hochschule mannheim

Jörn Fischer
j.fischer@hs-mannheim.de

Willkommen zur Vorlesung

Algorithmen für moderne Rechnerarchitekturen

Vorstellung

Zu meiner Person...

Überblick

Inhalt

1 - Organisatorisches

1 - Organisatorisches

Wie läufts ab?

- Vorlesungsfolien auf meiner Homepage
- Großer Teil der Vorlesung ist Projektarbeit in 1er - 3er Gruppen!
- Projektarbeit wird mit mindestens 4 Seiten IEEE Paper (deutsch oder englisch) dokumentiert (Studienleistung)
- Prüfung besteht aus 15-20 Min. Vortrag pro Person

1 - Organisatorisches

Bitte registrieren Sie sich zu ALR unter folgendem Link:

http://www.informatik.hs-mannheim.de/~fischer/serverprogs/registration/ALR/index.php

1 - Organisatorisches

Literatur

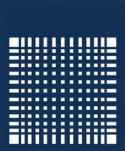
- NVIDIA, “NVIDIA CUDA C Programming Guide”, NVIDIA website
- NVIDIA, “CUDA C BEST PRACTICES GUIDE“, NVIDIA website
- David Kirk, “Programming massively parallel processors”, ELSEVIER Verlag, ISBN-13: 978-0-12-381472-2
- Thomas Rauber, Gundula Rünger, "Parallele Programmierung", 2. Auflage, Springer Verlag, ISBN:978-3-540-46549-2
- Thomas Beierlein, Olaf Hagenbruch, "Taschenbuch Mikroprozessortechnik", 2. Auflage, Fachbuchverlag Leipzig, ISBN:3-446-21686-3
- M.Allen, B.Wilkinson, “Parallel Programming”, Prentice-Hall, ISBN-13: 978-0131405639

1 - Organisatorisches

Fragen?

Auch jederzeit per email...

2 – Einführung parallele Computer



2 - Einführung parallele Computer

2 - Einführung parallele Computer

2 - Einführung parallele Computer

2.1 - Warum parallele Computer?

...Probleme effizienter lösen...

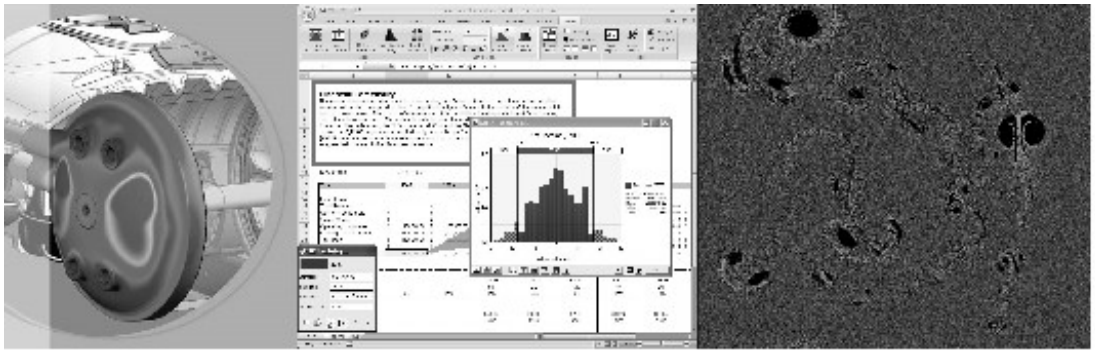
- Aufgaben schneller lösen
- Genauigkeit erhöhen
- Größere Instanzen eines Problems lösen

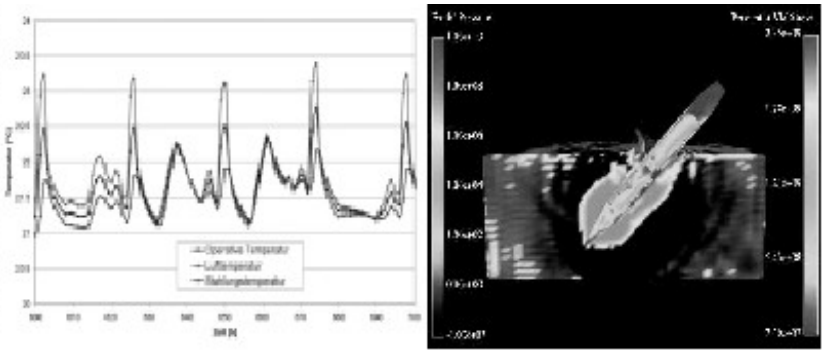
2 - Einführung parallele Computer

2.1 - Warum parallele Computer?

... Ereignisse simulieren...

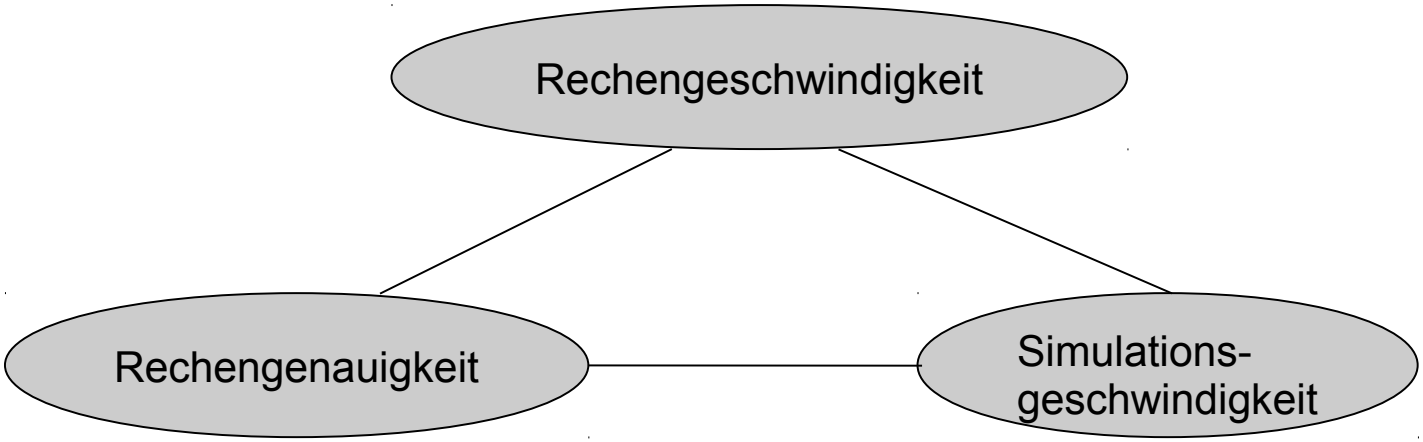
- Realität lässt sich oft genauer nachbilden als in typischen Versuchsanordnungen
- Simulation ist i.a. Kostengünstiger
- Parameter sind einfacher zu ändern
- Manche Dinge lassen sich kaum in der Realität ohne Simulation nachbilden (Wetter, Börsenkurse etc...)





2 - Einführung parallele Computer

Für Simulationen gelten folgende Abhängigkeiten:



2 - Einführung parallele Computer

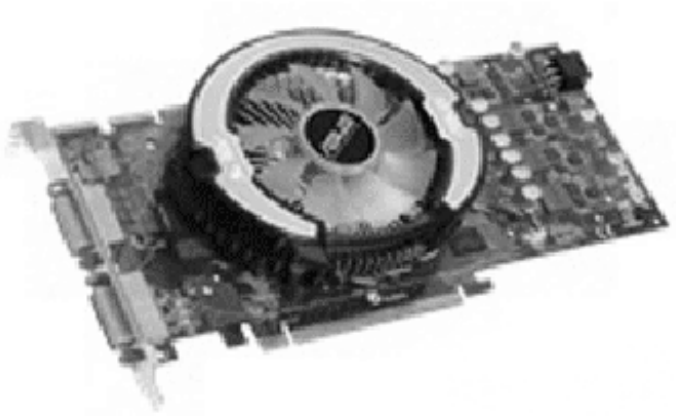
Wie kann man die Rechengeschwindigkeit erhöhen?

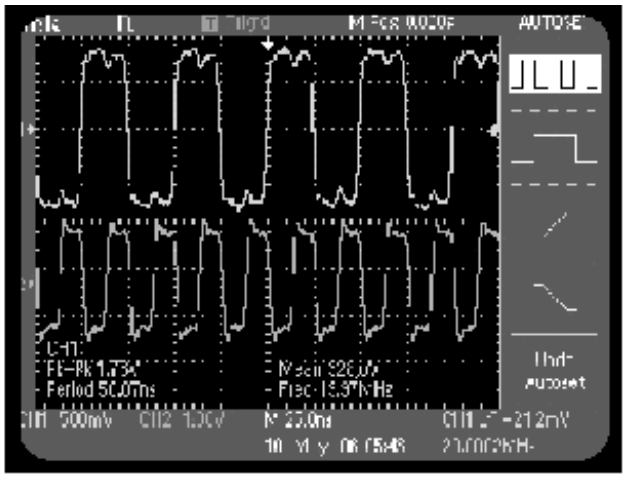
- Methode **1**) Taktrate erhöhen
- Methode **2**) Parallelisierung

2 - Einführung parallele Computer

Methode 1) Taktrate erhöhen...

- Erhöhte Leistungsaufname
- Wärmeentwicklung pro Chipfläche kritisch!!!
- Physikalische Grenzen kritisch!!!





2 - Einführung parallele Computer

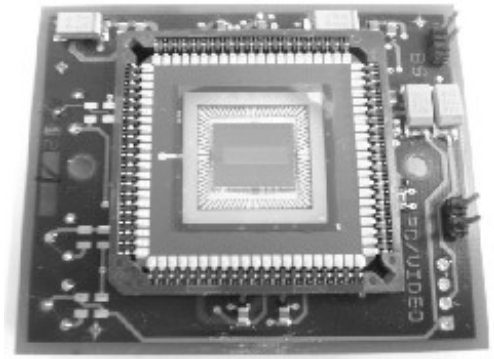
Methode 1) Physikalische Grenzen:

- Prozessor mit Taktrate von 3 Ghz
- ...entspricht einer Zykluszeit von 0.333 ns
- in dieser Zeit kann ein Signal maximal

$$0.333\ \ 10^{(-9)}\,s\ \ 0.3\ \ 10^{9}\,\frac{m}{s}\ \approx\ 10\,cm$$

zurücklegen

- Bei Verzehnfachung ist die Größenordnung des Chipdurchmesser erreicht!



2 - Einführung parallele Computer

Methode 1) Problem der Speicherzugriffszeit...

1980-1988 Effizienzsteigerung	pro Jahr
Integer-Operationen	55%
Floating Point Operationen	75%
Speicherkapazität DRAM	60%
Zugriffszeiten DRAM	25%

2 - Einführung parallele Computer

Methode 1) Entschärfung von “Speicherzugriffszeiten“

...durch Einsatz von Caches

- CC-NUMA: Cache Coherent Non-Uniform Memory Access
- NC-NUMA: Non-Coherent Non-Uniform Memory Access

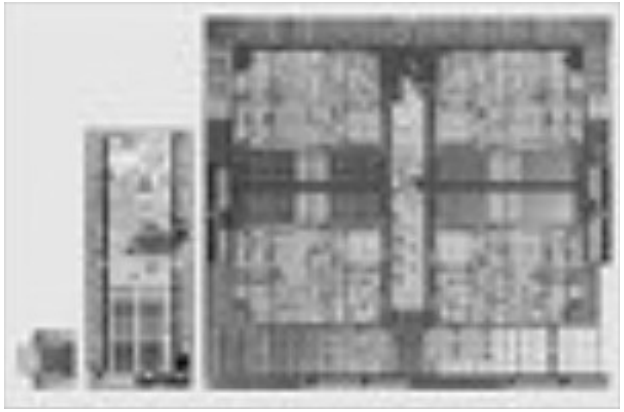
(Daten können sind z.B. Tabellen, die nur gelesen werden)

- COMA-Rechner: Cache only Memory Access

2 - Einführung parallele Computer

Methode 2) Parallelisierung

- Erhöhte Leistungsaufname
- Wärmeentwicklung
- Platzbedarf
- Aufwendigere Programmierung
- Aber machbar!

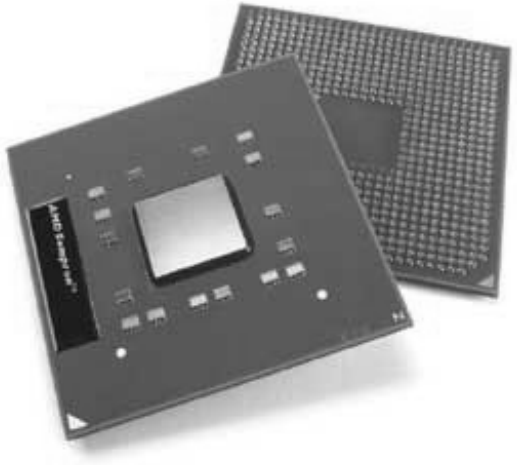


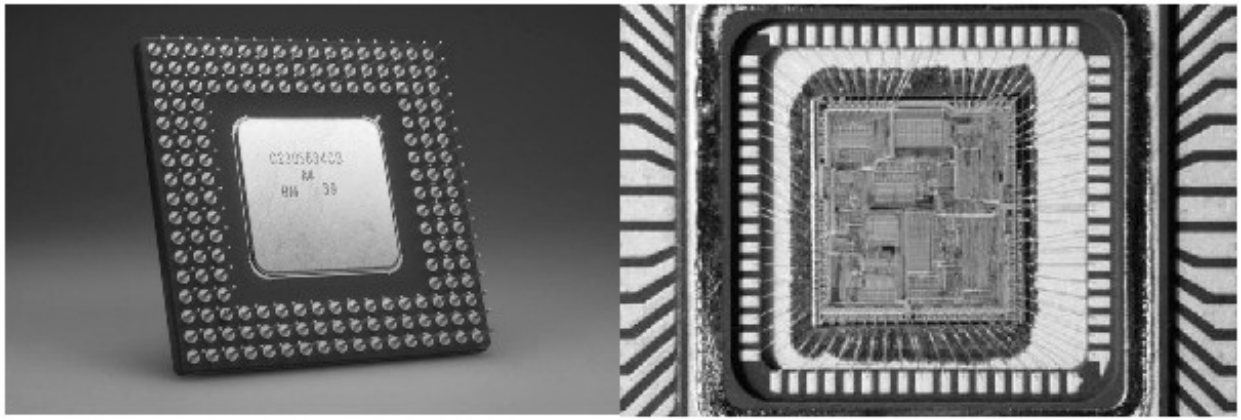
ARM, ATOM, INTEL DualCore

2 - Einführung parallele Computer

Methode 2) Parallelisierung innerhalb des Prozessors

- Parallelität auf Bitebene
- Parallelität durch Pipelining
- Parallelität durch mehrere Funktionseinheiten Superskalare Prozessoren (Very Large Instruction Word)
- Parallelität auf Prozess bzw. Threadebene





2 - Einführung parallele Computer

Methode 2) Multithreading/ Hyperthreading

- Interleaved Multithreading

  Verdeckung der Verzögerungszeiten (Latenzzeiten) des Memories durch wechseln des Threads bei jedem Befehl (feinkörnig und feste Anzahl von Threads)

- Blockorientiertes Multithreading

  Thread wechselt erst bei Speicherzugriff auf nicht lokalen (langsamen) Speicher

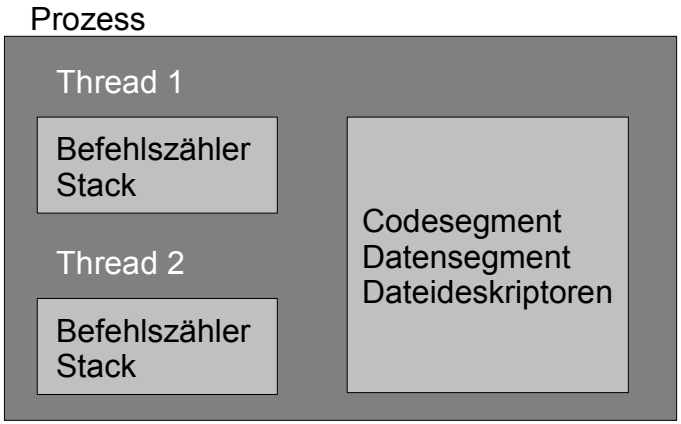
=> Hyperthreading (intel Prozessoren):

Threadwechsel wird von der Hardware unterstützt !!!

2 - Einführung parallele Computer

Prozess (Task) und leichtgewichtiger Prozess (Thread)

- Thread teilt sich Codesegment, Datensegment, Dateideskriptoren des Prozesses zu dem er gehört
- Thread hat eigenen Stack und Programmcounter
- Threads können untereinander kommunizieren



2 - Einführung parallele Computer

2.2 - Metriken

Wir definieren Metriken zum Vergleich unterschiedlicher Systeme:

- Speedup
- Effizienz
- Kommunikationsaufwand

2 - Einführung parallele Computer

Speedup Faktor

Maßzahl für die Performancesteigerung beim Umstieg auf ein Mehrprozessorsystem

$$S(p) = \frac{(Ausführungszeit\ auf\ single\ \ CPU)}{(Ausführugszeit\ auf\ Mehrprozessorsystem)} = \frac{T_s}{T_p}$$

Bei p Prozessoren ist der maximale Speedup Faktor:

$$S_{max}(p) = \frac{T_s}{(T_s / p)} = P \geq S(p)$$

ALR - Jörn Fischer - j.fischer@hs-mannheim.de - Büro: A112

2 - Einführung parallele Computer

Speedup Faktor (Amdahl's Gesetz)

Für den Speedup sind folgende Größen maßgeblich

- Idle Zeiten aller Prozessoren
- Zusätzlicher Ausführungscode (z.B. Konstanten neu berechnen)
- Kommunikationszeit

2 - Einführung parallele Computer

f sei der nicht parallelisierbare Teil des Programms

dann berechnet sich die Gesammtausführungszeit zu

$$T_p = f\,T_s + (1-f)\,T_s/p$$

und der Speedup zu:

$$S(p) = \frac{t_s}{(f\,t_s + (1-f)\,t_s/p)} = \frac{p}{(1+(p-1)\,f)}$$

(Amdahl's Gesetz)

2 - Einführung parallele Computer

Kann der Speedup auch größer werden ?

$$S(p) > p \quad ?$$

Ja, wenn

- Der Speicher des parallelen Systems größer ist
- Der sequentielle Algorithmus suboptimal ist
- Das Parallelsystem zusätzliche Features hat

ALR - Jörn Fischer - j.fischer@hs-mannheim.de - Büro: A112

2 - Einführung parallele Computer

Effizienz eines Multiprozessorsystems

Maßzahl für die Auslastung der Prozessoren

$$E = \frac{T_S}{(T_p \quad p)} = \frac{(S(p))}{p}$$

2 - Einführung parallele Computer

Ein System ist skalierbar wenn:

- Die Steigerung der Aufgabengröße mit mehr Prozessoren bewältigt werden kann
- Für die Erweiterung keine prinzipielle Architekturänderung notwendig ist

2 - Einführung parallele Computer

Kommunikationsaufwand

Der Datenaustausch zwischen den Prozessoren kann zu einem signifikanten Overhead führen...

$$T_p = T_{communication} + T_{computation}$$

Je mehr parallele Prozesse desto größer ist der Kommunikationsaufwand...

$$computation\ /\ communication\ ratio = \frac{T_{computation}}{T_{communication}}$$

2 - Einführung parallele Computer

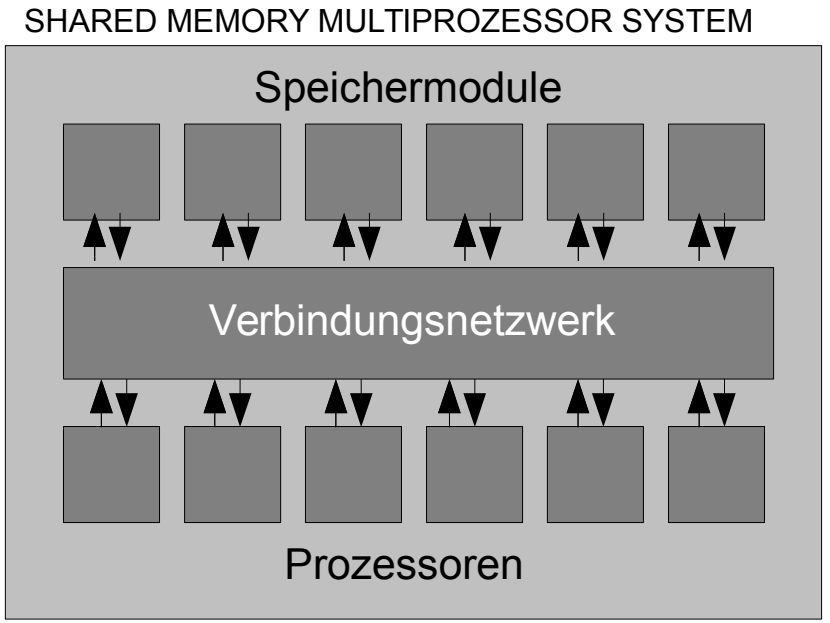
2.3 - Arten paralleler Computer

- Shared Memory Multiprozessor System
- Message-Passing Multicomputer
- Distributed Shared Memory Multiprozessor System

2 - Einführung parallele Computer

Shared Memory Multiprozessor System

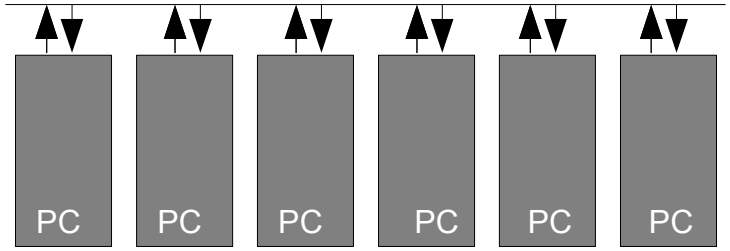
- Alle Prozessoren haben Zugriff auf alle Speichermodule
- Der Zugriff erfolgt über Verbindungsnetzwerk
- Für die Programmierung gibt es Spracherweiterungen wie z.B. OpenMP für C/C++, Unified Parallel C, Fortran
- Einfach für 2-4 Prozessoren
- Für mehr Prozessoren eher hierarchische oder verteilte Speicherstruktur
- Cache entschärft etwas das Zugriffsproblem



2 - Einführung parallele Computer

Message Passing Multicomputer

- Mehrere Computer, die untereinander vernetzt sind
- Prozessor hat eigenen Speicher
- Über Verbindungsnetzwerke werden Nachrichten und Daten ausgetauscht
- Durchmesser eines Netzwerks ist die minimale Anzahl physikalischer Links zwischen zwei Knoten und ist wichtigster Faktor zur Bestimmug der Verzögerung einer Nachricht zwischen den Knoten

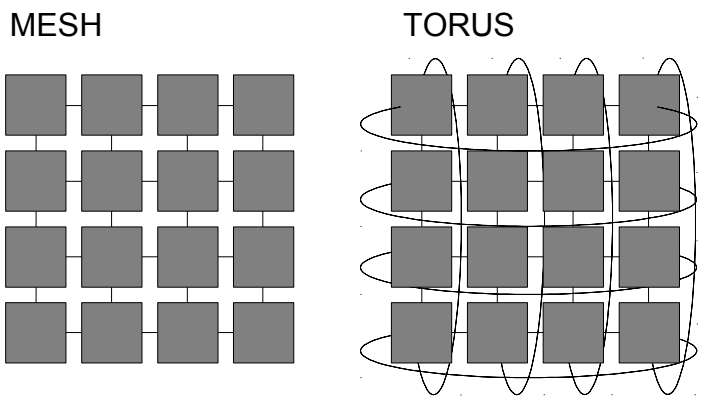


MESSAGE PASSING MULTICOMPUTER

2 - Einführung parallele Computer

Mesh

- Die Knoten werden in einem zweidimensionalen Netz angeordnet
- Duchmesser $=2(\sqrt{p}-1)$
- Verbindet man die gegen-überliegenden Seiten so nennt man das Torus
- Torusdurchmesser $=\sqrt{p}$
- Die 3-dim Variante hat für jeden Knoten in jeder Dimension 2 Verbindungen

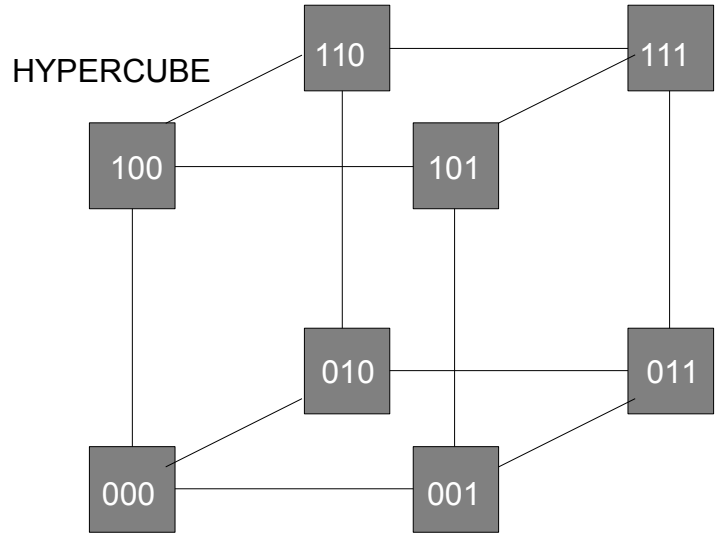


2 - Einführung parallele Computer

Hypercube

In einem d-dimensionalen binären Hypercube ist jeder Knoten mit einem Knoten in jeder Dimension verbunden

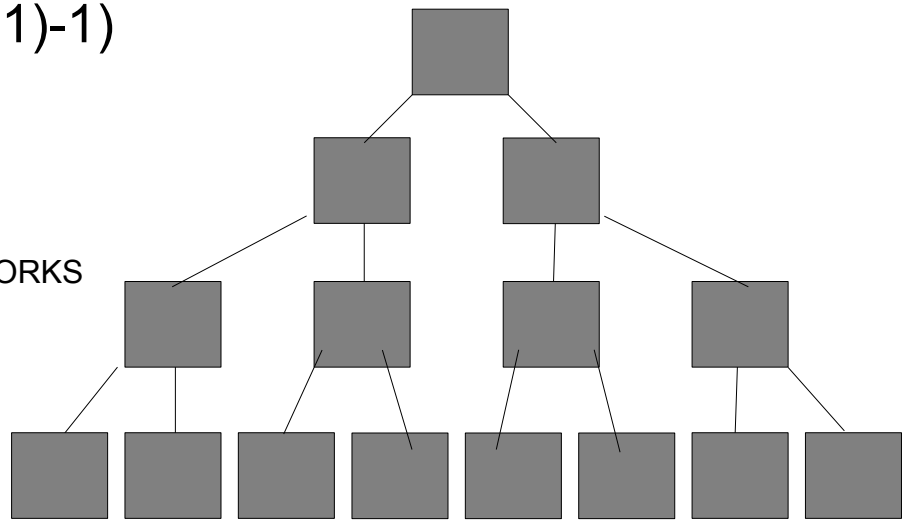
- Durchmesser $= \log_2 p$ also nur logarithmisches Wachstum bei Erweiterung des Netzes
- Je höherdimensional, desto mehr Verdrahtungs-aufwand



2 - Einführung parallele Computer

Tree Network

- Netzwerk mit Baumstruktur
- Netzwerkverkehr steigt an der Wurzel (Flaschenhals)
- Vorteil: die höhe des Baumes steigt nur logarithmisch mit der Anzahl der Prozessoren
- Durchmesser = $2\ ((\log_2(p+1)-1)$ beim binären Baum

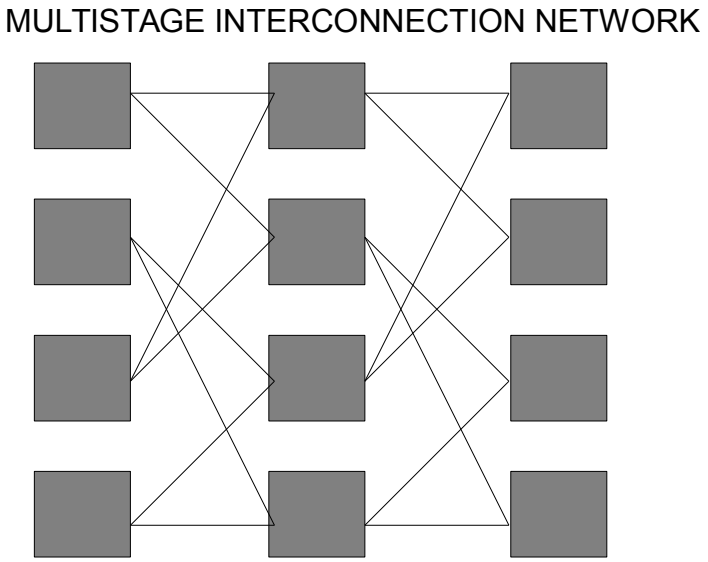


2 - Einführung parallele Computer

Multistage Interconnection Networks

- Netzwerkklassen, die eine bestimmte Anzahl von Switch-Ebenen haben

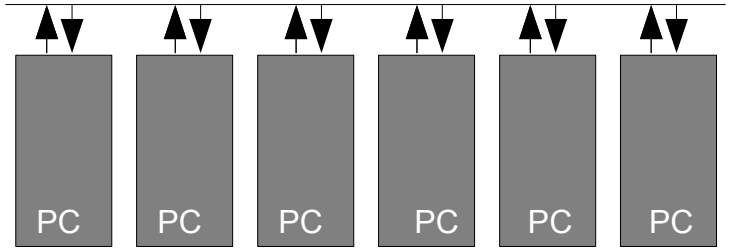
Bsp: Omeganetzwerk:
stark blockierend, aber
immer einen Weg zwischen
Eingang und Ausgang



2 - Einführung parallele Computer

Distributed Shared Memory Multiprocessor System

- Vernetzte Rechner oder Prozessoren
- Jeder Rechner hat eigenes Memory, welches aber von den anderen Rechnern mitbenutzt wird
- Architektur eher wie beim Message passing
- Software eher wie beim shared Memory



DISTRIBUTED SHARED MEMORY MULTIPROCESSOR SYSTEM

2 - Einführung parallele Computer

Kommunikation

- Circuit Switching: Der gesamte Verbindungspfad wird für die Zeit des Datentransfers reserviert
- Packet Switching: Die Pakete werden in einzelnen Knoten zwischengespeichert bis die ausgehende Verbindung frei ist
  - Locking
    - Livelock: wenn Paket im Netz umherirrt ohne Zielknoten zu finden (adaptives Routing)
    - Deadlock: wenn mehrere Knotenbuffer voll sind

2 - Einführung parallele Computer

Klassifikation nach Flynn (1996)

- SISD – Single Instruction Single Data
- MIMD – Multiple Instruction Multiple Data
- SIMD – Single Instruction Multiple Data
- MISD – Multiple Instruction Single Data

2 - Einführung parallele Computer

Programmierung

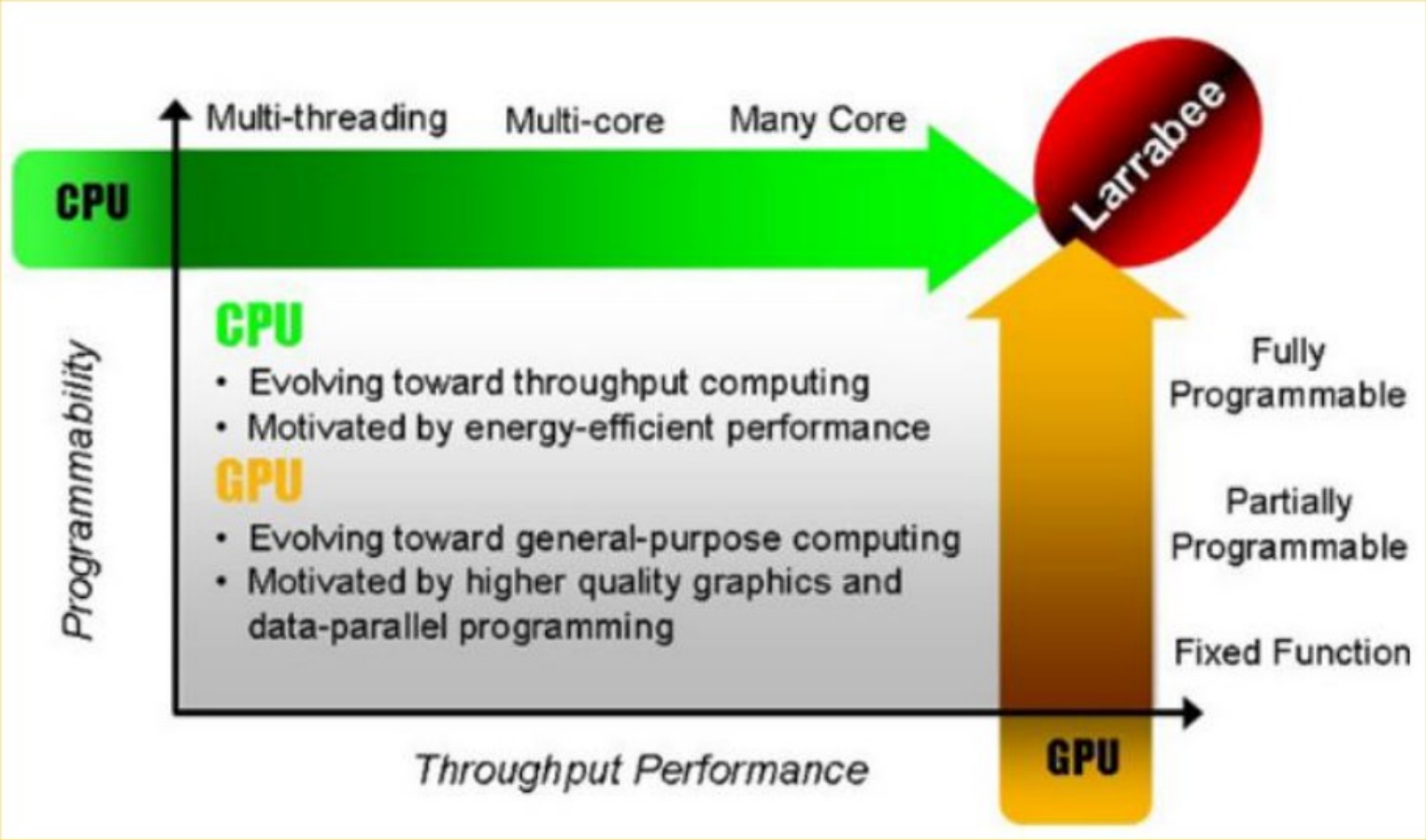
- Spezielle Programmiersprache
- Erweiterung der Sprachelemente
- Verwendung einer speziellen Library

Tools

- MPI (Message Passing Interface)
- openMP (für shared Memory Systeme)
- openCL (für CPU,GPU, DSP)
- CUDA (NVIDIA Grafikkarten)

3 - GPUs

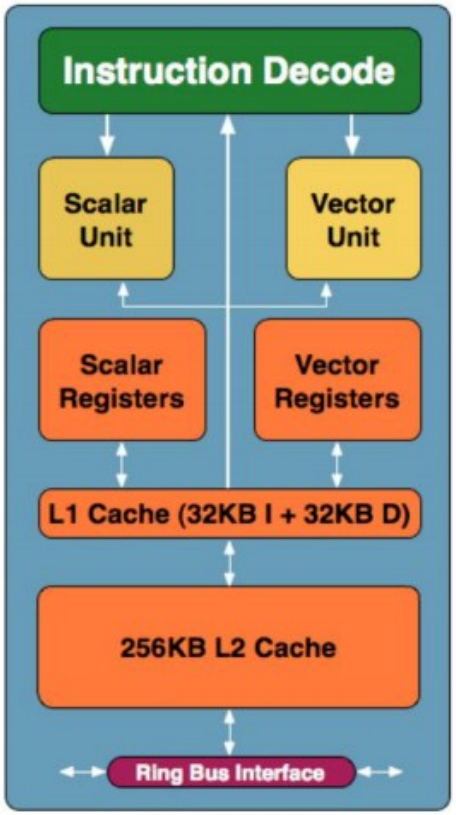
3.1 – CPU und GPU Trends



3 - GPUs

3.1 – CPU und GPU Trends (Beispiel: Larrabee)

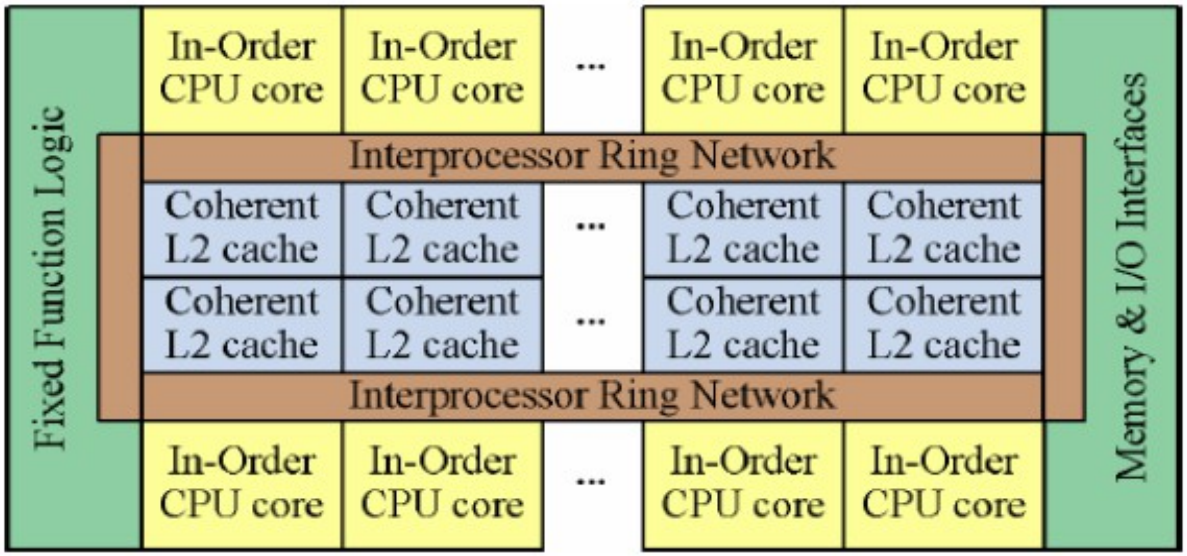
(aus Vortrag von Elmar Küsters)



- basiert auf Pentium-Architektur
- in-Order
- 4 Threads
- 32KB L1 Data Cache
- 256KB L2 Cache
- 64-bit X86
- 1 x 16-wide Vector Unit
- Erweiterter X86 Befehlssatz

3 - GPUs

3.1 – CPU und GPU Trends (Beispiel: Larrabee)



? Cores

? 256KB L2 Cache

2 x 512 Bit Ring Bus

Fixed Function Logic / Texture Filter Logic

3 - GPUs

3.1 – CPU und GPU Trends (Beispiel: Larrabee)

Vergleich mit Core 2 Duo

# CPU cores:	2 out-of-order	10 in-order
Instruction issue:	4 per clock	2 per clock
VPU per core:	4-wide SSE	16-wide
L2 cache size:	4 MB	4 MB
Single-stream:	**4 per clock**	**2 per clock**
Vector throughput:	**8 per clock**	**160 per clock**

Gleicher Stromverbrauch, gleiche DIE größe

Aber: Vector Processing Unit macht den Unterschied

3 - GPUs

3.1 – CPU und GPU Trends (Beispiel: Larrabee)

Programmierung GPU von AMD/ NVIDIA

- DirectX / OpenGL
- Direct Compute (DirectX11)
- NVIDIA : CUDA (Compute Unified Device Architecture)
- openCL

3.1 – CPU und GPU Trends (Beispiel: Larrabee)

Programmierung von Larrabee

- DirectX / OpenGL
- Direct Compute (DirectX11)
- bekannte x86 Programmierung
- Prozessoren könnten auch vom Betriebssystem genutzt werden
- SDK noch nicht verfügbar

3.1 – CPU und GPU Trends (Beispiel: Larrabee)

Performance

- November 2009 demonstrierte Intel auf der Super-computing Messe ein übertaktetes Larabee System mit 1TFLOPS.
- 2008 hatten bereits NVIDIA und AMD Grafikkarten dieser Leistung gezeigt
- 4.12.2009 entschied Intel Larrabee vorerst nicht zu vermarkten

3 - GPUs

3.2 Cayman- und Fermi- Grafikchiparchitekturen

- AMD (Cayman) verbaut 1536 Rechenkerne (4 Kerne gebündelt)
- NVIDIA (Fermi) verbaut nur 512 Rechenkerne

- AMD Chip/ Shadertakt 880Mhz/ 880 Mhz
- NVIDIA Chip/ Shadertakt 772 Mhz/ 1544 Mhz

- AMD Theoretische Rechenleistung 2703 Gflops
- NVIDIA Theoretische Rechenleistung 1581 GFlops

3 - GPUs

3.2 Cayman- und Fermi- Grafikchiparchitekturen

- AMD (Cayman) 256kb = 64k Regs pro VLIW (Very Long Instruction Word=64 Cores)
- NVIDIA (Fermi) 128kb = 32k Regs pro SM (Streaming Multiprozessor= 32 Cores)

- AMD (Cayman) L1 Cache 8kB pro VLIW
- NVIDIA (Fermi) L1 Cache 48kb oder 16kb
  (kann durch shared Memory geschmälert werden)

- AMD Shared Memory 32kb pro VLIW
- NVIDIA Shared Memory 16kb oder 48kb pro SM

3 - GPUs

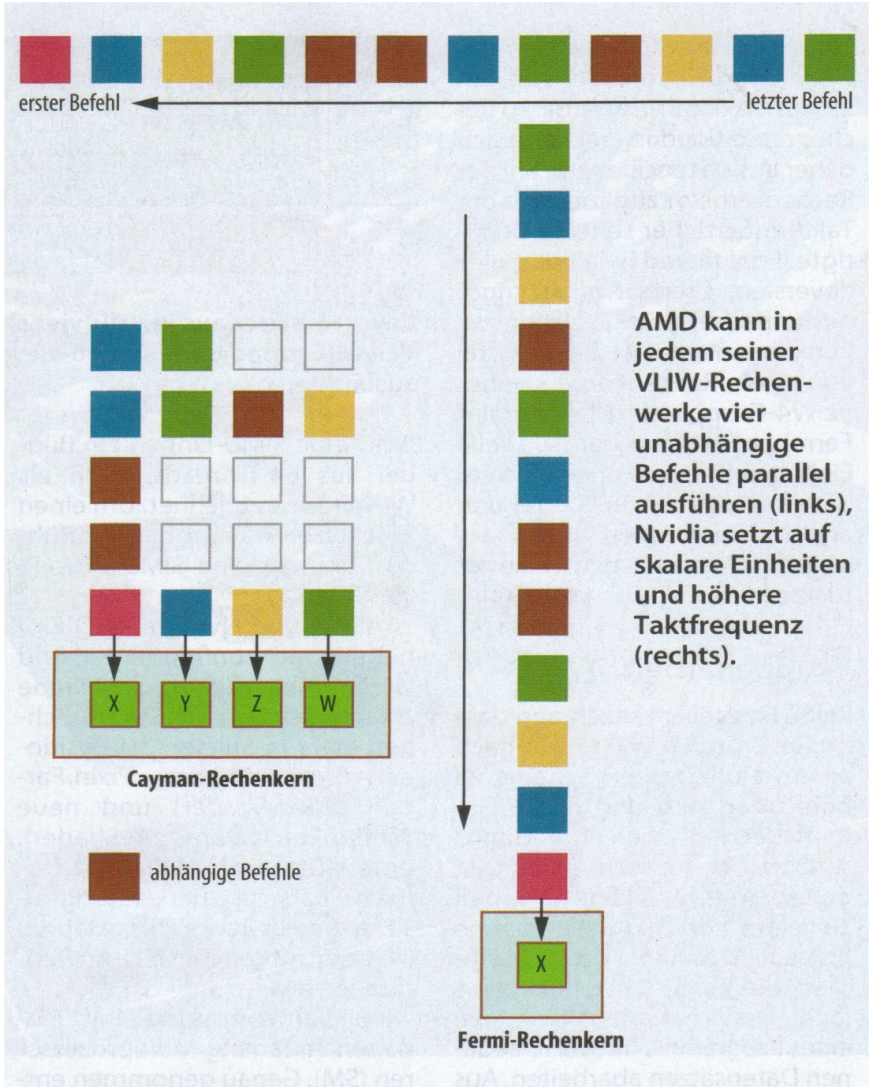


Abb. aus: C‘T 4 Magazin für Computertechnik 31.11.2011

3 - GPUs

3.2 Cayman- und Fermi- Grafikchiparchitekturen

- AMD (Cayman) 246 Wavefronts pro VLIW
- NVIDIA (Fermi) 512 Wavefronts pro SM

- AMD (Cayman) Cache ist Read only
- NVIDIA (Fermi) Cache ist Read/ Write

- NVIDIA Hardware für Cos, Sin...

3 - GPUs

3.3 Die Kepler Architektur (GK110)

- 1 Tflops double precision operations
- 3 x Performance pro Watt

- Features:
  - Dynamic Parallelism: GPU kann eigene Kernel starten
  - Hyper Q: Startet gleichzeitig Kernel von unterschiedlichen CPU Kernen aus
  - Grid Manag. Unit: Grid Management Controllsystem effizienter
  - Nvidia GPU Direct: Datenaustausch zwischen 2 GPUs auch über Netzwerk

4 - CUDA

4 - CUDA

Inhalt

4 - CUDA

Anmerkung:

In diese Vorlesung ist Material des CUDA Workshops von Oliver Mangold vom HLRS Stuttgart eingeflossen. Für die Erlaubnis dies tun zu dürfen möchte ich herzlich danken und auf die Internetseite des HLRS verweisen:

http://www.hlrs.de/no_cache/events/training/

Einführung

Warum GPUs benutzen?

- Sehr hohe floating point performance
- Hohe Memory Bandbreite
- Massenprodukt → billig

Probleme:

- Massive Parallelisierung notwendig
- Nutzen von doppelt genauen „double“ Zahlen ist 5-10 Fach langsamer
- Verschiedene Memoryarten
- Control flow Instruktionen können rechenaufwändig sein

Einführung

Was ist CUDA?

- CUDA = Compute Unified Device Architecture
- NVIDIA SDK enthält Treiber, C/C++ Compiler und Runtime Framework für nVIDIA Grafikkarten für Windows, Linux, MacOS
- NVCC (Compiler) enthält eine minimale C/C++ Erweiterung, so dass CPU und GPU effizient ausgelastet werden können
- Vielen 1000 Threads werden auf der GPU parallel abgearbeitet

Einführung

Was muss noch behandelt werden?

- Das Programmiermodell ist eine Abstraktion der nVIDIA-Hardware
- Um Performanz zu bekommen ist es nötig die Hardware zu kennen
  - Paralleler scheduler
  - Single Instruction Multiple Thread
  - Memory Architecture
  - …
- Die Anzahl der Hardwarefeatures wächst kontinuierlich
  - Einige Features brauchen neuere Hardware
  - compute capability

Einführung

- Die Hardware ist mehrstufig parallel:
  - (GPU - Multiprozessoren - Skalarprozessoren)
- Programmiermodell ist mehrstufig parallel:
  - (grid - blocks - threads)
- Verschiedene Speicherarten
  - Globale Memory, Shared Memory, Texture Memory, Constant Memory, Register

Einführung (Begrifflichkeiten)

- Host
  - CPU und Memory des Mainboards
- Device
  - Videokarte bzw GPU und Videomemory
- Global memory/ Device Memory
  - Speicher auf der Videokarte
  - Ist jedoch nicht auf dem Chip selbst
- Multiprozessor
  - Ein CORE der GPU
  - Kann mehrere Threads mit identischem Code zur selben Zeit ausführen.
- Compute Capability
  - Hardwareversion des Grafikprozessors
  - Fermi hat compute capability 2.0

Einführung (Programmiermodell)

- CUDA Programme bestehen aus:
  - Ein „Host“ Programm
  - Kontrolliert den wesentlichen Programmablauf
  - Ist für den Datenaustausch zwischen CPU und GPU verantwortlich

- „Accelerator“ Programmiermodell
  - CPU = Master
  - GPU = Slave
  - CPU und GPU arbeiten parallel, CPU hat die Kontrolle über die GPU

Einführung (Programmiermodell)

- Kernels
  - Ist nur vom Host aus aufrufbar
  - Beispiel: __global__ void getSum(float* inputA, float* sum);
  - Man nennt diese Funktionen CUDA Kernel
  - Die Funktion muss immer vom Typ void sein.
- Device Funktionen
  - Ist nur vom device aus aufrufbar
  - Beispiel: __device__ float max(float x, float y);
  - Kann nicht rekursiv sein für compute capability < 2.0
- Andere Funktionen
  - Alle anderen Funktionen werden als Host Code compiliert

Einführung (Programmiermodell)

- Kernels können vom Hostcode wie jede andere Funktion aufgerufen werden.
- Bemerkung:
  - Kernel aufrufe sind asynchron d.h. , dass die Funktion sofort zurückkehrt bevor die Bearbeitung der Funktion erfolgt ist.

- Beispiel:

```
main(){
    …
    myCudaFunction<<<gridSize, blockSize>>> (inputA, inputB,
    output, size);
    cudaThreadSynchronize();
    …
    return 0;
}
```

Einführung (Kernel Aufrufskonfiguration)

- Was heißt Kernel Aufrufskonfiguration?
  <<<gridSize, blockSize >>>

- Block von logischen Threads
  - Werden parallel ausgeführt
  - Werden in unterschiedlichen Skalarprozessoren des selben Multiprozessors ausgeführt
- Grid of Blocks
  - Werden unabhängig von-einander seriell oder parallel ausgeführt
  - Werden auf unterschiedliche Multiprozessoren verteilt

Einführung (Kernel Aufrufskonfiguration)

- Fortgeschrittene Konfigurationssyntax:

```
dim3 blockSize(16,8,4); // → 512 threads per block
dim3 gridSize(64,32);    // 2048 blocks
MyKernelFunction <<<gridSize, blockSize>>>();
```

  - Threads können 1,2 oder 3 Dimensional organisiert sein
  - Es dürfen maximal 512 Threads pro Block sein
  - Blocks können 1 und 2 Dimensional organisiert sein
  - Jede Block Dimension darf maximal 65535 sein
  - Alle unspezifizierten Komponenten von dim3 werden mit 1 initialisiert

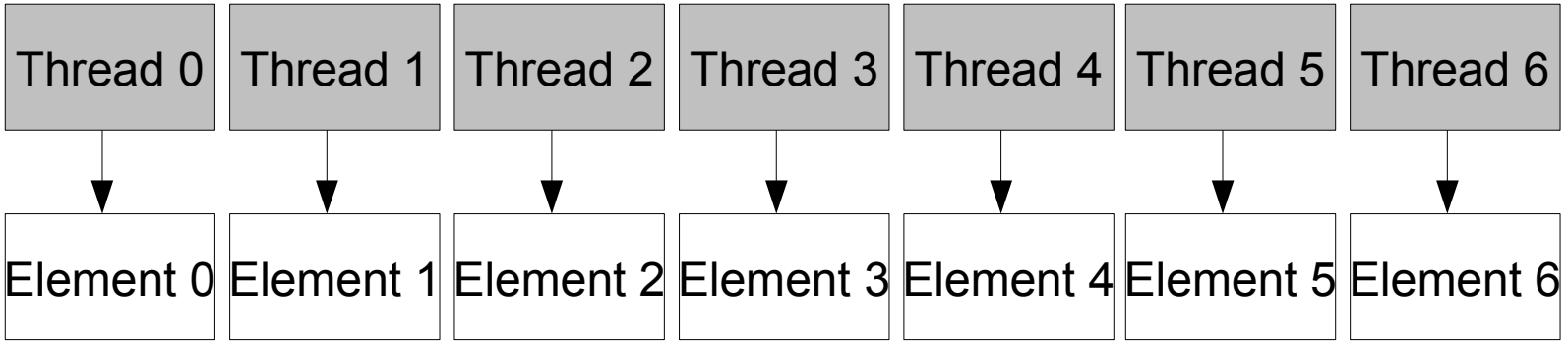
Einführung (Kernel Aufrufskonfiguration)

- Wie bekommt man die thread/block Information im Kernel Code
  - `threadIdx.x (threadIdx.y, threadIdx.z)` → `index des threads innerhalb des Blocks`
  - `blockDim.x (blockDim.y, blockDim.z)` → `Anzahl der Threads pro Block`
  - `blockIdx.x (blockIdx.y, blockIdx.z)` → `index des Blocks`
  - `gridDim.x (gridDim.y, gridDim.z)` → `Anzahl der Blocks im Grid`

Beispiel: addiere 2 Vektoren auf einem Multiprozessor

- Wenn vektor size = block size:

```
__global__ void VectorSum(float* inputA, float* inputB, float* output){
    output[threadIdx.x] = inputA[threadIdx.x] + inputB[threadIdx.x]
}
```

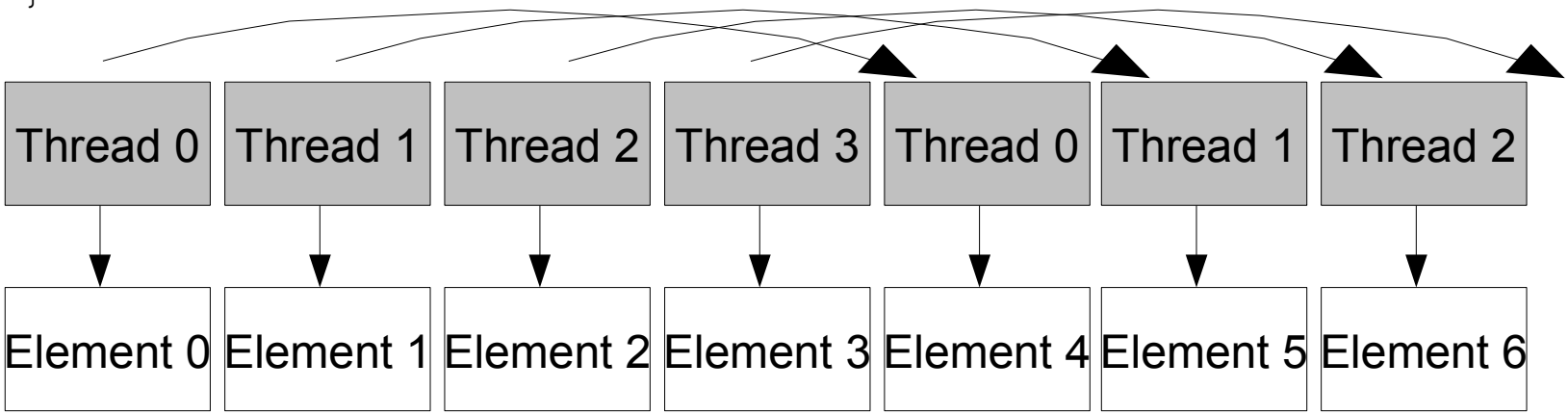


Jeder thread greift auf genau ein Element des Vektors zu

Beispiel: addiere 2 Vektoren auf einem Multiprozessor

- Wenn vektor size != block size:

```
__global__ void VectorSum(float* inputA, float* inputB, float* output){
   for (int i=threadIdx.x; i<size; i+=blockDim.x){
       output[i] = inputA[i] + inputB[i]
   }
}
```

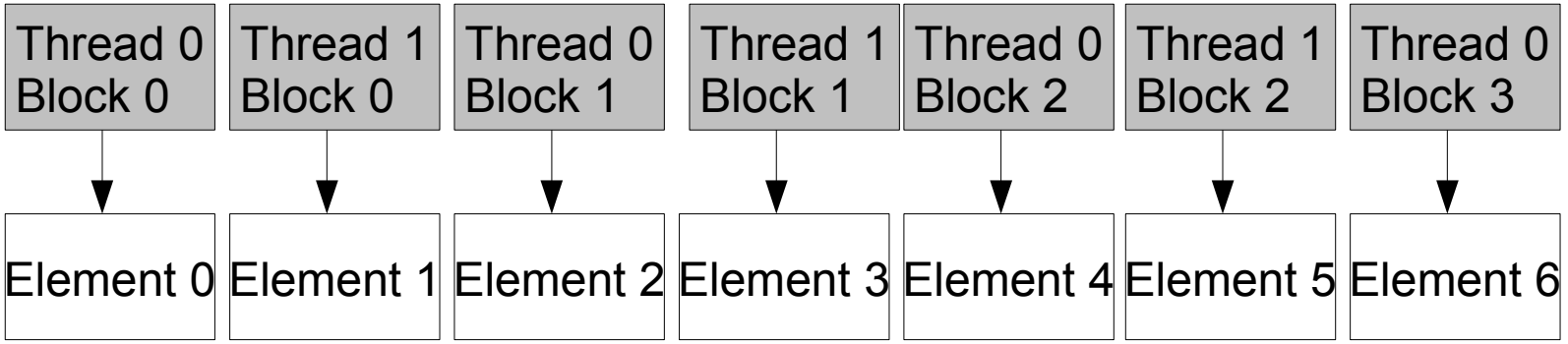


Jeder thread greift auf ein Element des Vektors pro Schleifendurchlauf zu.
Im Bild sind es genau 4 Threads.

Beispiel: addiere 2 Vektoren auf mehreren Multiprozessoren

- Wenn vektor size = block size * grid size:

```
__global__ void VectorSum(float* inputA, float* inputB, float* output){
      // lokale Variablen sind für jeden Thread lokal
      int myIndex = blockIdx.x*blockDim + threadIdx.x;
      output[myIndex] = inputA[myIndex] + inputB[myIndex]
}
```

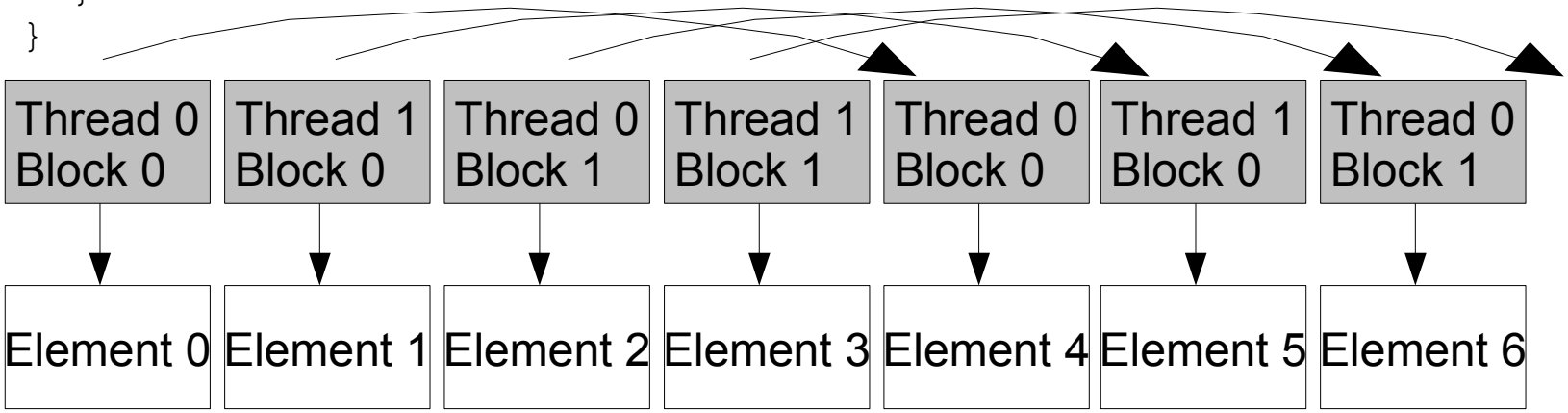


Jeder thread in jedem Block greift auf genau ein Element des Vektors zu
Im Bild ist die Anzahl der threads = 2

Beispiel: addiere 2 Vektoren auf mehreren Multiprozessoren

- Wenn vektor size != block size * grid size:

```
__global__ void VectorSum(float* inputA, float* inputB, float* output){
   for (int i=blockIdx.x*blockDim.x+threadIdx.x; i<size;
                                                 i+=blockDim.x*gridDim.x){
       output[i] = inputA[i] + inputB[i]
   }
}
```



Jeder thread greift auf ein Element des Vektors pro Schleifendurchlauf zu.
Im Bild sind es genau 2 Threads und 2 Blocks.

Variablendeklaration und Memory

- Variablen, die ausserhalb der __device__ Funktionen deklariert sind, sind globale Hostvariablen
- Variablen, die mit __device__ global deklariert sind gehören zum globalen device memory
- Variablen, die innerhalb von __device__ Funktionen deklariert sind werden typischerweise als GPU Register oder als Globaler GPU Speicher umgesetzt. Sie sind lokal innerhalb eines Thread
- Variablen, die mit __constant__ in globalem Scope deklariert sind, sind nur lesbar von allen threads und allen Blöcken(wird später behandelt)
- Variablen, die mit __shared__ innerhalb von __device__ Funktionen deklariert sind lokal pro Block (wird später behandelt)

Zeiger

- Pointer im Host zeigen per default aufs Host-Memory
- Pointer im Device zeigen auf Device-Memory

- Bitte keine Pointer, die aufs Host Memory zeigen in den Device übertragen
- Genauso wenig Pointer, die aufs device Memory zeigen auf den Host übertragen

Speicher reservieren und kopieren

- Device Memory Allozieren: `cudaError_t cudaMalloc( void** pointer, size_t size );`
- Device Memory freigeben: `cudaError_t cudaFree( void* pointer );`
- Memory kopieren `cudaError_t cudaMemcpy( void* dat, const void* arc, size_t count, enum cudaMemcpyKind kind );`
  - Kind kann sein: `cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice`
- Reservieren und freigeben von page-locked host Memory

```
cudaError_t cudaMallocHost(void** pointer, size_t size);
cudaError_t cudaFreeHost(void* pointer);
```

- Warum page locked memory?
  - Memory kann schneller von und zum device kopiert werden
  - Zugriff vom Device ist möglich
  - Asynchrones kopieren ist möglich

Synchronisation

- Wie kann die Ausführung der Threads und Blocks synchronisiert werden?
  - Die device Funktion __syncthreads() synchronisiert alle threads des gleichen Blocks (wenn ein thread den Befehl nicht erreicht gibt es einen deadlock!)
  - Die host Funktion cudaThreadSynchronize(); blockiert die Host CPU bis alle GPU Kernelfunktionen abgearbeitet ist
  - Implizite Barrieren
    - Verschiedene Kernelfunktionen überlappen sich nicht für Grafikkarten der Compute Capability <2.0 (Fermi kann 4 kernels gleichzeitig bearbeiten)
    - cudaMemcpy() überlappt Kernelfunktionen nur für host → host
    - cudaMemcpy() blockiert den host thread bis der transfer beendet ist

Fehlerbehandlung

- Die funktion `cudaError_t cudaGetLastError();` gibt den letzten Fehler eines cuda Funktionsaufrufes oder eines kernelaufrufs wieder
- Die Funktion `char* cudaGetErrorString(cudaError_t error);` gibt die Fehlermeldung in lesbarer Form zurück
- Mehrere Cuda Funktionen geben `cudaError_t` direkt zurück
- Wenn kein Fehler auftrat ist der Rückgabewert `cudaSuccess`

Vereinfachte Fehlerbehandlung

- Das File „`cuda_utils.h`“ bietet ein einfacheres Interface zur Fehlerbehandlung
- Wenn der Ausdruck x vom Typ cudaError_t mit dem Makro `cu_verify(x)` aufgerufen wird, wird falls es eine Fehlermeldung gibt, die Meldung ausgegeben und das Programm abgebrochen.
- Das Makro `cu_verify_void(x)` ruft Statement x auf und dann cudaGetLastError().
    - Wenn das Ergebnis ein Fehler ist, wird der String ausgegeben und das Programm abgebrochen. Bei Kernelaufrufen bitte doppelte Klammern setzen `cu_verify_void((myKernel<<<>>>()));` sonst kommt der Präprozessor durcheinander.

Beispiel Host Programm

```
int main() {

  int vectorSize = 10000000;
  int memorySize = sizeof(float)*vectorSize;

  float *vectorHostA;
  float *vectorHostB;
  float* vectorHostResult;

  cudaMallocHost((void**)&vectorHostA,
                 memorySize);
  cudaMallocHost((void**)&vectorHostB,
                 memorySize);
  cudaMallocHost((void**)&vectorHostResult,
                 memorySize);

  ... initialize vectors A,B ...

  float* vectorDeviceA;
  float* vectorDeviceB;
  float* vectorDeviceResult;

  cudaMalloc((void**)&vectorDeviceA,
             memorySize);
  cudaMalloc((void**)&vectorDeviceB,
             memorySize);
  cudaMalloc((void**)&vectorDeviceResult,
             memorySize);

  cudaMemcpy(vectorDeviceA, vectorHostA,
             memorySize, cudaMemcpyHostToDevice);
  cudaMemcpy(vectorDeviceB, vectorHostB,
             memorySize, cudaMemcpyHostToDevice);

  VectorSum<<<gridSize, blockSize>>>
            (vectorDeviceA, vectorDeviceB,
             vectorDeviceResult, vectorSize);

  cudaMemcpy(vectorHostResult,
             vectorDeviceResult,
             memorySize, cudaMemcpyDeviceToHost);

  ... print result ...

  cudaFree(vectorDeviceA);
  cudaFree(vectorDeviceB);
  cudaFree(vectorDeviceResult);

  cudaFreeHost(vectorHostA);
  cudaFreeHost(vectorHostB);
  cudaFreeHost(vectorHostResult);

  return 0;
}
```

Aus dem CUDA Workshop von Oliver Mangold am HLRS: http://www.hlrs.de/no_cache/events/training/

Beispiel Device Programm

- In folgendem Device Programm werden 2 Vektoren addiert.

```
__global__ void vectorAdd(float* inA, float* inB, float* out){
  for (int i=blockIdx.x*blockDim.x+threadIdx.x; i<size; i+=blockDim.x*gridDim.x){
      out[i] = inA[i] + inB[i];
  }
}
```

Was kann innerhalb der Kernelfunktionen benutzt werden

- Was kann genutzt werden
  - Alle mathematischen Funktionen
  - Kontrollfluss Befehle (if, for, while, case, goto)
  - Aufrufe von device Funktionen
  - Pointer
  - Structs und statisch deklarierte arrays
  - CUDA spezifische Funktionen
  - C++ templates
  - C++ Funktionsüberladung

Was kann innerhalb der Kernel nicht benutzt werden

- Was kann nur in neueren Devices benutzt werden
  - Double precision floats (compute capability>1.3)
  - Rekursive Funktionen (compute capability >= 2.0)
  - Funktionspointer (compute capability>=2.0, CUDA >3.1
  - C++ nicht-polymorphe Klassen (keine virtuellen Funktionen) (c.c.>2.0)
  - Printf (c.c.>=2.0, CUDA 3.1)
- Was kann zur Zeit nicht in device-code benutzt werden?
  - C99/C++ arrays mit dynamischer Größe
  - C++ polymorphe Klassen (Virtuelle Fkt., dynamic casts etc.)
  - System calls, I/O Memory management (fopen, fprintf, malloc...)
  - Long double floats

Beispiel: Vektor Skalarmultiplikation

Aufgabe 1: serielle Funktion auf der GPU

- Aufgabe 1: Vector-scalarmultiplikation
  - CPU-Code:

```
const int size = 10000;
float a[size];
const float b;
float b[size];
…
for (int i=0;i<size;i++){
     c[i]=a[i]*b;
}
```

  - Die Multiplikation bitte auf der GPU mit einem Thread in einem Block

- Vorgehensweise:
  - Reserviere Speicher auf dem device mit cudaMalloc()
  - Reserviere Speicher auf dem host mit cudaMallocHost()
  - Benutze cudaMemcpy() um den Input Vektor auf die Grafikkarte zu kopieren
  - Rufe den Kernel mit GridSize=1 und BlockSize=1 auf
  - Benutze cudaMemcpy() um den Output Vektor auf den Host zu kopieren

Aufgabe 2: parallele Funktion auf der GPU

- Ändere die Lösung aus Aufgabe 1 so, dass mehrere Blocks und Threads benutzt werden um das Ergebnis zu erzeugen
  - Dazu bitte threadIdx.x, blockDim.x, blockIdx.x, gridDim.x benutzen um den Array Index zu ermitteln
  - Können Sie die Funktion so umschreiben, dass statt einer Schleife viele Blöcke benötigt werden um das Ergebnis zu berechnen?

CUDA Programme compilieren

- NVIDIA compiler:
  - Nvcc -arch=compute_13 <source file> -o <executable>
    - -arch=compute_13 heisst „compute capability 1.3“
- Emulationsmodus
  - Alle host Funktionen können benutzt werden
    nvcc -deviceemu <source file> -o <executable>
  - Gibt es nur für CUDA < 3.0 !
- Andere debugging Tools:
  - cuda -gdb <executable> (debugger, braucht compilierung mit ‘ -g -G‘
  - Cuda-memcheck <executable> (um Memoryleaks zu finden)

Performance Optimierung I

GPU Architektur

- Zwei Stufen der Parallelität:
  - Multiprozessoren (Fermi: 15 oder 16)
  - Jeder Multiprozessor hat mehrere Skalarprozessoren (Fermi: 32)
- Die Arbeit muss auf mehrere
  - Multiprozessoren und mehrere
  - Skalarprozessoren verteilt werden.

Warps

- Single Instruction Multiple Thead Ausführung
  - Jede Gruppe von 32 Threads des selben Blocks führen die gleiche Instruktion zur selben Zeit aus.
  - Diese Gruppe von Threads werden Warps genannt
    - Half-Warps nennt man die ersten oder letzten 16 threads eines Warps
    - Einige Operationen wie Speicherzugriffe werden für jeden Half-Warp kombiniert
- Befehle werden auf dem Skalarprozessor gepipelined, so dass
  - 4 Befehle in 4 Zyklen - nicht aber 1 Befehl in 1 Zyklus ausgeführt wird

Single Instruction Multiple Data (SIMT)

- Was passiert wenn der Kontrollfluss divergiert

```
if ( threadIdx.x >  5){
        output[threadIdx.x] = ...
 else{
        output[threadIdx.x]=...
 }
```

- Funktioniert aber Verzweigungen werden seriell abgearbeitet
- Nur Verzweigungen, die von mindestens einem Thead durchlaufen werden werden bearbeitet

Anmerkung zu SIMT Ausführung

- Wie viele threads pro Block sollte man nehmen?
  - Threads sind in 32‘er Blöcken organisiert.
  - 4 threads brauchen die gleiche Zeit wie 32 Threads (wenn sie auf GPU Registern arbeiten)
  - Also ist es Ideal ein Vielfaches von 32 Threads zu nehmen
  - Die Anzahl der Register eines Multiprozessors ist begrenzt (Fermi: 32768)

Aus dem CUDA Workshop von Oliver Mangold am HLRS: http://www.hlrs.de/no_cache/events/training/

Speichermodell

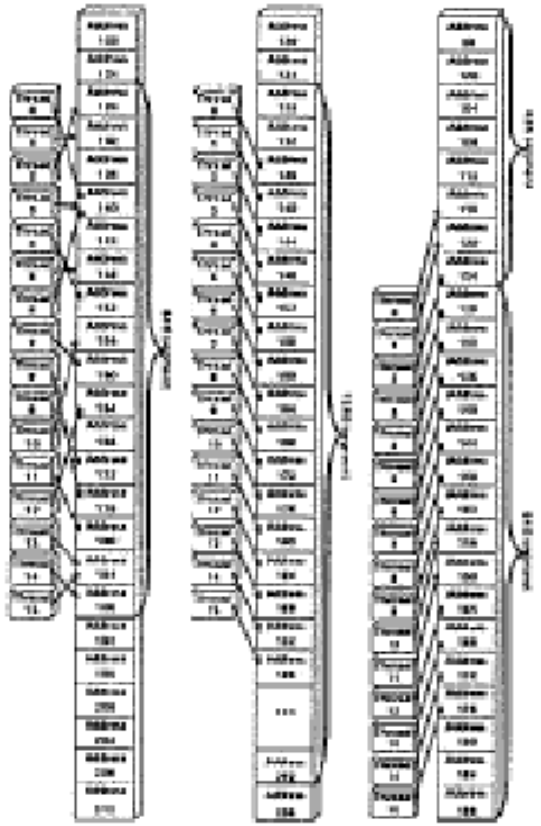
- Es gibt verschiedene Arten von Memory:
  - Auf das **Host Memory** kann nur langsam über den PCI Bus zugegriffen werden
  - **Globale Memory:** (Hauptmemory der Grafikkarte) kann von allen Multiprozessoren erreicht werden
    - Hohe Bandbreite (>100 GB/s)
    - Große Latenzzeit (>100 Zyklen)
  - **GPU Register:** benutzt für Daten eines einzelnen Threads (Zugriffszeit: 1 Zyklus)
  - **Lokales Memory:** physikalisch gleich dem globalen Memory, aber lokal pro thread (wird genutzt wenn dem Thread die Register ausgehen)
  - **Shared, Constant, Texture Memory:** später diskutiert

Bemerkung zu globalem Memory Zugriff

- „Verschmolzene“ Zugriffe (Coalesced access)
  - Vereinigen von Memoryzugriffen steigert die Performanz gravierend
  - Wenn jeder thread eines Half-Warps das Selbe Memory Segment (128/64/32 Bytes) zugreift, wird der Zugriff zu einer Transaktion kombiniert
    - Half-Warps sind entweder die ersten 16 oder die letzten 16 Threads eines Warps
- Bemerkung: Memoryzugriffe auf das Globale– und das Shared-Memory werden
  - Pro Half-Warp auf der T10 Architektur behandelt
  - Pro Warp behandelt auf der Fermi Architektur

Bemerkung zu globalem Memory Zugriff

- „Verschmolzene“ Zugriffe (Coalesced access)
  - Linkes: 64 Byte Memory Zugriff
  - Mitte: 128 Byte Memory Zugriff
  - Rechts: zwei Memory Zugriffe (32- und 64 Bytes)
- „unverschmolzene“ Zugriffe (Uncolesced access)
  - Zugriffe, in denen derselbe Half-Warp auf unterschiedliche 64-/128 Byte Speicherblöcke zugreift. Das gilt auch für Zugriffe über den Cache



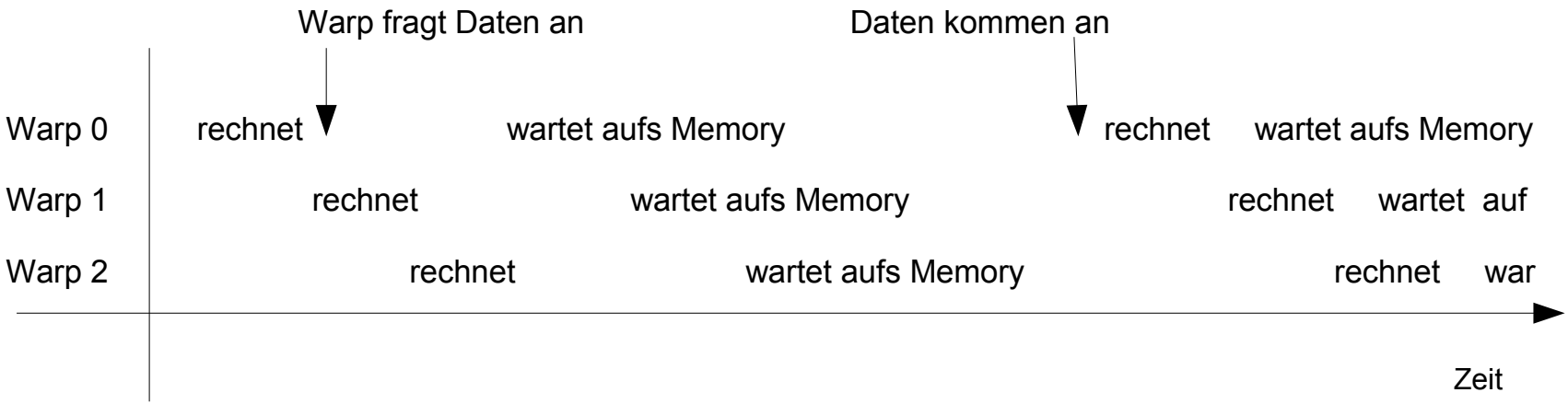
Aus dem CUDA Workshop von Oliver Mangold am HLRS: http://www.hlrs.de/no_cache/events/training/

Bemerkung zu block size und grid size

- Wie viele Blocks pro grid sollte man benutzen?
  - Um alle Multiprozessoren zu benutzen sollten die Anzahl der Blocks >= Anzahl der Multiprozessoren sein
  - Zugriffe aufs globale Memory haben große Latenzzeiten: verstecken der Latenzzeiten ist notwendig um volle Memory-Bandbreite zu bekommen
    - Wenn die Anzahl der Blocks größer ist als die Anzahl der Multiprozessoren, wird die Blockauführung überlappt (multitasking)
- Gleiches für Threads:
  - Wenn die Anzahl der Threads größer ist als die Warp-Size (32) wird die Warp-Ausführung überlappt (multitasking)

Verstecken der Latenzzeiten

- Wie funktioniert das Verstecken der Latenzzeiten
- Jeder Multiprozessor kann zwischen verschiedenen Warps umschalten
- Wenn ein Thread aufs globale Memory zugreift wartet er 500-600 Zyklen
- Während der Zeit fürt der Multiprozessor Code eines anderen Warps aus.



Verstecken der Latenzzeiten

- Die Maximale Anzahl von Warps, die von einem Multiprozessor gehandhabt werden können sind
  - 32 auf der T10 Architektur
  - 64 auf der Fermi Architektur
- Die Maximale Anzahl von Blöcken, die von einem Multiprozessor gehandhabt werden können sind 8
- Heißt: BlockSize >32 hat Vorteile beim verstecken der Latenzzeiten

Nutzung der Register

- Achtung: Umschalten von Warps (Multitasking) findet nur statt, wenn die nötige Anzahl von Registern und das nötige shared Memory im Multiprozessor vorhanden ist.
- Wie kann ich raus finden wie viele Register pro Thread mein Kernel benutzt?

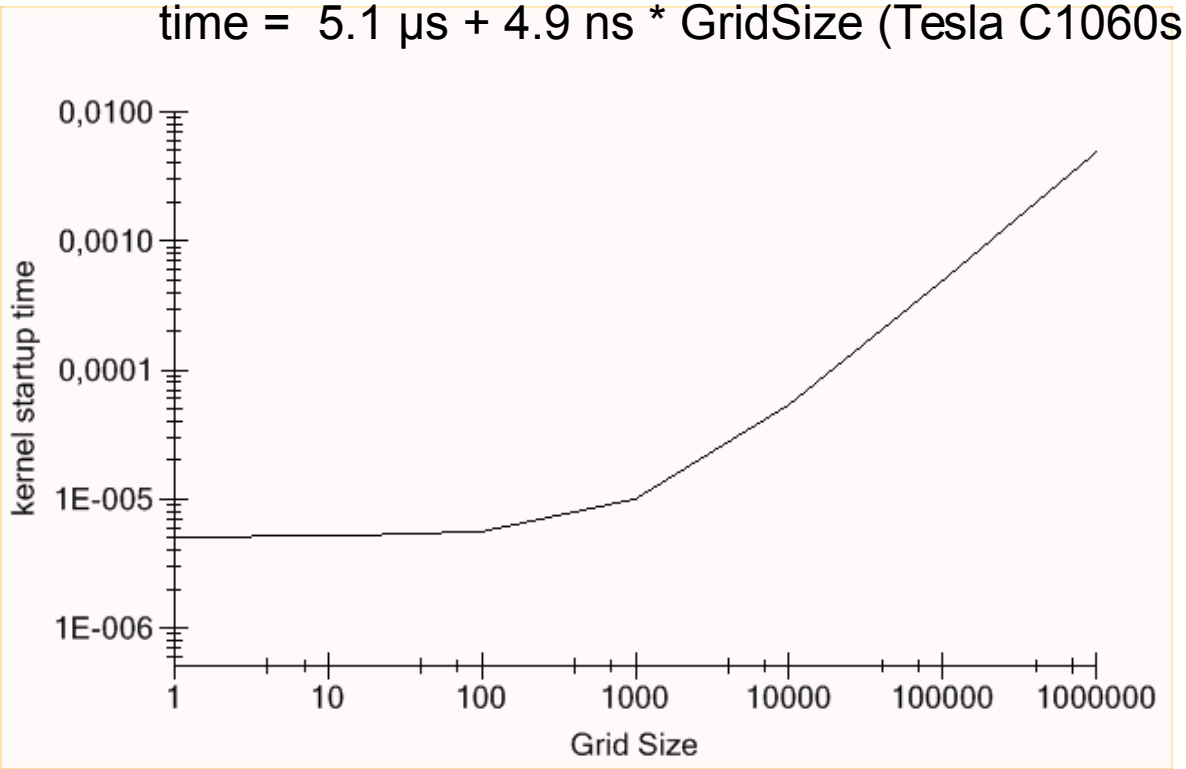
--code=sm_13 –ptxas-options -v (Fermi:--code=sm_20 –ptxas-options -v)

- Kann ich die Anzahl der benutzten Register begrenzen?
  - Ja, wenn man im Kernel folgendes benutzt:
  - __launch_bounds__(maxthreadsPerBlock,MinBlocksPerMP)
  - Oder bei der compilierung: --maxrregcount <N>

CUDA Kernel Startzeit

- Wie lange braucht es vom Host einen Kernel zu starten?

  time = 5.1 µs + 4.9 ns * GridSize (Tesla C1060s)



Bemerkung: Der Overhead pro Block beträgt nur wenige Zyklen → es ist ok eine grosse Anzahl von Blöcken zu nehmen und sollte die bevorzugte Lösung sein um die Latenzzeit zu verstecken.

Memory Tansfer zwischen Host und Device

- Wie lange dauert der Memorytransfer zwischen Host und Device
  - Etwa 3.3 µs + Datengröße * 6Gb/s (hängt vom PCI express Bus ab)

Empfehlung:

- Vermeiden Sie unnötigen Datentransfer
- Falls große Datenmengen verarbeitet werden ist es sinnvoll den Datentransfer mit Rechenzeit zu überdecken
  - Die Funktion

```
cudaMemcpyAsync( void* dst, const void*arc, size_t count,
                 enum cudaMemcpyKind kind, cudaStream_t stream);
```

    kann benutzt werden um Datentransfer mit CPU bzw. GPU Berechnungen zu überlappen
  - Dafür müssen streams benutzt werden, die später behandelt werden.

Fließkommazahlen mit doppelter Genauigkeit

- Double addition und Multiplikation sind auf Consumer Grafikkarten um einen Faktor 8 langsamer und bei Fermi Karten um den Faktor 2
- Wenn möglich einfache Genauigkeit verwenden
- Merke: Um doubles auf dem device nutzen zu können muss die Compute Capability>1.3 sein. Ausserdem muss für die Compilation ‚-arch=compute_13‘ oder „-arch=compute_20‘ genutzt werden
- Konstanten in C/C++ sind ohne suffix automatisch vom Typ double. Floats haben den suffix f z.B.: 1.45f

Befehlsdurchsatz

- 1 Multiprozessor benutzt auf der Tesla T10 Architektur folgende Anzahl von Cycles (..) in einem Warp:
  - 32 Bit float Befehle
    - Add oder multiply-add (4)
    - 1/x (16)
    - x/y (36)
    - 1/sqrt(x) (16)
    - sqrtf(x) (32)
    - sinf(x),cosf(x),logf(x),expf(x) (viele!)
  - 32 Bit integer Befehle
    - Add (4)
    - Multiply (16)
    - Comparison, &, | (max 4)
  - __syncthreads() (4)

Befehlsdurchsatz

- 64 Bit floats
  - Add, multiply-add (32)
  - Andere double Befehle (typischerweise viele)
- Merke: Es gibt schnellere aber weniger akkurate Versionen von float Befehlen
  - __sinf(x),__cosf(x),__expf(x) (32)
  - __logf(x) (16)
  - __fdivdef(x,y) (20)
- Da die T10 GPU intern 32 bit floats nutzt sind 24 bit Integer Multiplikationen schneller mit:
  - __mult24(x,y),...umul24(x,y) (4)
  - Fermi nutzt 32bit integer Multiplikationen und ist damit schneller!

Pipelining und Datenabhängigkeiten

- Die Tesla Architektur kreiert für das Pipelining 4 mal größere Warps als die Anzahl der Skalarprozessoren
- Die GPU kann bereits neue Befehle starten bevor die alten abgeschlossen sind
- Das funktioniert nur, wenn es keine Datenabhängigkeiten gibt.
- Das Beispiel 2 läuft aus diesem Grund wesentlich schneller

```
1)float val;
  for (int t=0;t<iterations;t++){
      val = 2.f * val * (1.f - val);
      val = 2.f * val * (1.f - val);
  }
2)float value1, value2;
  for (int t=0;t<iterations;t++){
      val1 = 2.f * val1 * (1.f - val1);
      val2 = 2.f * val2 * (1.f - val2);
  }
```

Zusammenfassung über Performance

- Beschäftige möglichst alle Skalarprozessoren und alle Multiprozessoren
- Vermeide Unterschiedliche Verzweigungen innerhalb der Threads eines Warps
- Gebrauche viele Blöcke aber wenig Register und wenig shared Memory um die Latenzzeiten zu verstecken
- Gebrauche „Verschmolzene“ Zugriffe (Coalesced access) zum globalen Memory
- Halte die Daten so lange wie möglich auf dem Device
- Benutze 32 Bit floats
- Benutze schnelle __mathematische Befehle wenn möglich

Übung „verschmolzene Zugriffe“

Vektor Addition

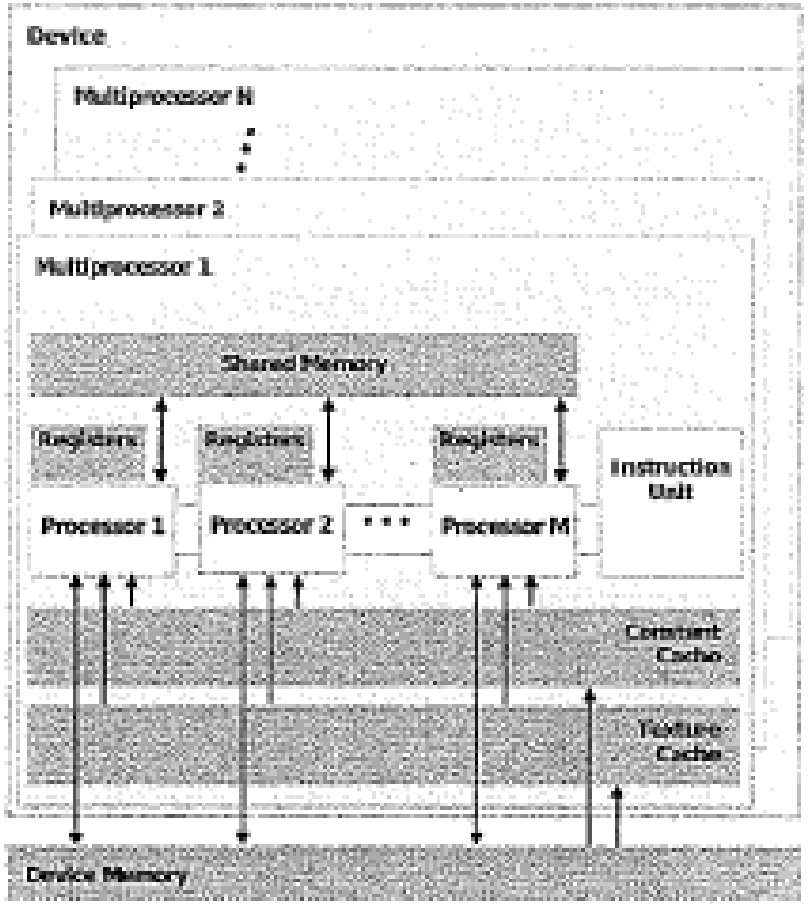
- Compiliere das Vektoradditionsbeispiel (unten) und messe die Performance
- Welche ist die beste Ausführungs Konfiguration
- Für welche Block/ Grid Grösse bekommt man gute Performance?

```
__global__ void vectorAdd(float* inA, float* inB, float* out){
  for (int i=blockIdx.x*blockDim.x+threadIdx.x; i<size; i+=blockDim.x*gridDim.x){
      out[i] = inA[i] + inB[i];
  }
}
```

Shared memory

Was ist shared memory

- Das shared Memory wird unter allen Threads des gleichen Blocks aufgeteilt.
- T10 → 16kb pro Multiprozessor
- Fermi → 16kb pro Multiprozessor
- Ist in 16 oder 32 Bänke eingeteilt
- Ist so schnell wie Register (Wenn keine Bankkonflikte auftreten)
- Gebraucht meist als: Zwischenbuffer zum Globalen Memory



Aus dem CUDA Workshop von Oliver Mangold am HLRS: http://www.hlrs.de/no_cache/events/training/

Wie kann shared Memory genutzt werden?

- Lokales Memory kann mit __shared__ im shared Memory deklariert werden.

```
__global__void myKernel(){
    __shared__ float data[size];
    ...
}
```

Pointer zum Shared Memory

- Pointer können zum shared- und zum globalen Memory zeigen

```
__ global__ void myKernal(float* input){
     __shared__ float sharedData[size];
     float* a = input;
     float* b = sharedData;
    ….
}
```

- Einschränkungen gibt es bei Grafikkarten der compute capabilty <2.0:

  Es muss zur Laufzeit bekannt sein zu welchem Memory der Pointer zeigt, sonst gibt der Compiler eine Warning aus.

Dynamisch reserviertes shared memory

- Shared Memory kann dynamisch allokiert werden, indem man ein extern `__shared__` Array von unbekanner Größe anlegt.

```
__global__ void myKernel(){
  __shared__ float a[32];
  extern __shared__ char dynamicSharedData[];
  …
}
```

- Die Größe des dynamisch angelegten Memories in Byte muss in der Ausführungskonfiguration mit angegeben werden:

myKernel<<<gridSize,blockSize,dynamicSharedSize>>>();

- Merke: Alle dynamischen shared Arrays zeigen auf die gleiche Adresse. Die Adresse muss per Hand korrigiert werden.

Synchonisation

• Wie kann man sicherstellen, dass der read Befehl zum shared (oder Globalen) Memory von einem Thread nach dem Schreibbefehl des anderen Threads kommt?

```
__global__ void myKernel(){
  __shared__ float a[..];
  // thread 0 writes a[0], thread[1] writes a[1],...
  a[threadIdx.x]=...;
  // thread 0 reads a[1]...
  … = a[threadIdx.x+1];
}
```

Synchonisation

- Lösung: Die Funktion __syncthreads() ist eine Barrierensynchronisation für jeden Thread.

```
__global__ void myKernel(){
  __shared__ float a[...];
  // thread 0 writes a[0], thread[1] writes a[1],...
  a[threadIdx.x]=...;

  __syncthreads();

  // thread 0 reads a[1]...
  … = a[threadIdx.x+1];
}
```

Beispiel: Diffusionsgleichung in 1 Dimension

- Jeder Punkt benötigt die Information seiner Nachbarn

$$f_i^{t+1} = f_i^t + a\left(f_{i-1}^t + f_{i+1}^t - 2 f_i^t\right)$$

- Einfache Implementation:

```
__global__ void diffusion(float* input, float* output,int size){
  for ( int i=blockIdx.x*bloackDim.x+threadIdx.x;
                        i<size;i+=blockDim.x*gridDim.x ){
      output[i] = input[i]*(1.-2. * a) + a*(input[i-1]+input[i+1]);
  }

}
```

Problem: Jeder Wert wird 3 mal aus dem globalen Memory gelesen

Beispiel: Diffusionsgleichung in 1 Dimension

- Implementation mit Buffering im shared Memory

```
__global__ void diffusion(int size, float*fieldIn, float* fieldOut){
   const int blockStart=(size-2)*blockIdx.x/gridDim.x+1;
   const int blockEnd=(size-2)*(blockIdx.x+1)/gridDim.x+1;
   for ( int i=blockStart;i<blockEnd;i+=sharedSize-2 ){
       //read shared values to shared memory
       for (int j=i-1;k=0;k<sharedSize;j+=blockDim.x,k+=blockDim.x){
           sharedMemory[k] = fieldIn[j];
       }
       __syncthreads();
       // calculate sharedSize-2 result values
       for (int j=i,k=1;k<sharedSize-1;j+=blockDim.x,k+=blockDim.x){
           fieldOut[j] = (1.-2.*a)*sharedMemory[k]+a*(sharedMemory[k-1]
                         +sharedMemory[k+1]);
       }
       __syncthreads();
   }
}
```

Reduktions Operationen

- Um Werte zu kombinieren, die in unterschiedlichen Threads ausgerechnet wurden wie beispielsweise die Summe oder das Produkt dieser Werte...
  - Es kann beispielsweise pro Block in Teilsummen aufgeteilt und das Ergebnis ins globale Memory geschrieben werden.
  - Dann können die Teilresultate in einem zusätzlichen Kernel zu einem Endresultat verrechnet werden
- Merke: Atomic operations können für integer Werte genommen werden. (Programming Guide B10.1.1)
  - Fermi hat auch atomicAdd() für single precision floats

Constant bzw. Texture Memory

Was ist Texture Memory?

- Global aber Read Only!
  - Merke: Mit Cuda 3.1 und Fermi erlaubt das ‚surface memory' READ/WRITE
  - Und mit 8 kB cache pro Multiprozessor
- Typische Anwendungen:
  - Stark benutzte Read Only Daten mit unregelmäßigen Zugriffsmustern
  - Beschleunigung von nichtlinearen Lesezugriffen auf das globale Memory
- Kann für optimiertes Caching konfiguriert werden für 1-, 2- und 3-Dimensionale Zugriffsmuster

Wie kann das Texture Memory benutzt werden?

- Ein Teil des globalen Memories kann als Texture Memory definiert werden:
- Schritt 1: Deklaration einer Texture Referenz

```
texture<Datatype, Dimension> texRef;
```

  Merke: Der Parameter Dimension ist optional.
- Merke: Texture Referenzen müssen als reine globale Variablen deklariert werden und können nicht als Kernelargument übergeben werden.

Wie kann das Texture Memory benutzt werden?

- Beispiel:

```
texture<float> textureRef1;
#define mySize 100
__global__ void myKernel(){
        int index = 10;
        // --- use texture memory
        float sum = tex1D(textureRef1,index);// correlates to: sum = deviceArray[10]
}
int main(int argc, char **argv){
        // --- reserve texture Memory and bind texture reference
        float *deviceArray;
        cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
        cudaMalloc( &deviceArray, sizeof(float) * N);
        cudaBindTexture(NULL,&textureRef1,deviceArray,&channelDesc,sizeof(float)*mySize);
        // --- call kernel
        myKernel<<<blocks,threadsPerBlock >>>();
};
```

Wie kann das Texture Memory benutzt werden?

- 2 Dimensional

```
cudaChannelFormatDesc channelDesc =
cudaCreateChannelDesc<Datatype>();
```

```
cudaBindTexture2D( Null, &texRef, memoryPtr, &channelDesc,
memorySize, arraySizeX, arraySizeY, pitch );
```

  mit pitch als offset zwischen einer Linie und der nächsten Linie. Die Adresse berechnet sich dann folgendermassen:

  address = baseAdr + pitch*indexY + sizeof(Datatype)*indexX

- Merke: pitch muss ein Vielfaches von der Pagesize sein (Tesla → 32 Bytes)

Wie kann das Texture Memory benutzt werden?

- Andere Methode: 1D-,2D- oder 3D textures mit Hilfe eines CUDA-arrays:

```
cudaChannelFormatDesc channelDesc =
cudaCreateChannelDesc<Datatype>();
CudaExtent extent = make_cudaExtant( width, height, depth );
CudaArray* arrayPtr;
CudaMalloc3DArray( &arrayPtr, channelDesc,extent );
CudaBindTextureToArray( texRef, arrayPtr );
```

3D Texturen müssen zu CUDA arrays gebunden werden

Cuda Arrays werden später kurz erklärt

Wie kann das Texture Memory benutzt werden?

- Schritt 3: Nun kann das texture Memory innerhalb des Kernels benutzt werden.

```
__global__ void mykernel(){
   …
   … = … tex1D(tex_ref_1D, index) …;
   … = … tex1Dfetch( texref_1D_ld, index)...;
   … = … tex2D(tex_ref_2d, indexX, indexY)...;
   … = … tex3D( tex_ref_3D, indexX, indexY, indexZ ) …;
}
```

- Merke: die Funktion tex1D,tex2D,tex3D haben einen 32 bit float Wert als Index
- Für große 1D-arrays sollte die Funktion tex1Dfetch genommen werden um Rundungsfehler zu vermeiden!

Wie kann das Texture Memory benutzt werden?

- Texture memory ist Read Only Memory im Globalen Memory
- So ist es möglich auf das Texture Memory zu schreiben mit Hilfe eines Pointers zum globalen Memory
- Wenn man vom Kernel aus über einen Pointer ins Texture Memory schreibt, welches vom selben Kernel ausgelesen wird, so ist das Ergebnis undefiniert. Das Texture Memory ist über einen Read Only Non-Coherent Cache realisiert
- Man kann jedoch von einem Kernel auf das texture Memory schreiben um es zu einem späteren Zeitpunkt von einem anderen Kernel wieder auszulesen.

Beschränkungen für das texture memory?

- 1D texture kann eine maximale Größe haben von:
  - 2^27 wenn es an eine globale Memoryregion gebunden ist
  - 8192 wenn es an ein Cuda Array gebunden ist
- 2D texturen können eine maximale Breite von 65536 und eine maximale Höhe von 32768 haben
- 3D Textures können eine maximale Breite, Höhe und Tiefe von 2048 haben
- Die base address pointer müssen für die Funktionen cudaBindTexture() und cudaBindTexture2D zu einer gegebenen Pagegröße Aligned sein (T10: 256 Bytes).
  - Die Pointer, die cudaMalloc() zurückgeben erfüllen dieses Alignment!
- Erlaubte Datentypen sind
  - 8,16,32 Bit integer
  - 32 Bit floats

CUDA Arrays

- CUDA Arrays sind 1-,2- oder 3- Dimensional passend zum Gebrauch für Texturen
- Die Funktion

```
cudaError_t cudaMalloc3DArray(
    struct cudaArray arrayptr,
    const struct cudaChannelFormatDesc desc,
    struct cudaExtent extent)
```

alloziert ein 1D Array, wenn Tiefe und Höhe beide 0 sind

alloziert ein 2D Array, wenn die Tiefe 0 ist.

Alloziert ein 3D Array, wenn die Tiefe und Höhe ungleich 0

CUDA Arrays

• Auf Daten in einem CUDA Array kann nicht direkt zugegriffen werden. Sie können mit cudaMemcpy kopiert werden:

- cudaMemcpyFromArray
- cudaMemcpyToArray
- cudaMemcpy2DFromArray
- cudaMemcpy2DToArray
- cudaMemcpy3D....
- ...

Weitere Features des texture memories

- Nutzung unterschiedlicher Datentypen
- Unterschiedliche Adress-Modes (clamp, wrap)
- Unterschiedliche Read-Modes (normalize...)
- Koordinaten Normierung (Koordinaten laufen von[0,1])
- Lineares Filtern der Texturen

Constant Memory

- Separater Constant Memory Bereich
  - Read Only
  - Mit 8 kB cache pro Multiprozessor
- Einschränkungen
  - Nicht dynamisch allozierbar
  - Maximal 64kB pro Device

Wie kann Constant Memory genutzt werden?

- Globale Variablen können als __constant__ deklariert werden:

```
__constant__ float data[size];
```

Wie kann Constant Memory genutzt werden?

- Auf Constant Memory kann wie auf globales oder lokales Memory zugegriffen werden.

```
__constant__ float data[size];
__ global__ void myKernel(){
  ...=...data[...];
}
```

Constant Memory vom Host Code beschreiben...?

• Der Host kann auf das Constant Memory nicht dirkt zugreifen. Es muss kopiert werden:

```
cudaError_t cudaMemcpyToSymbol(const char* symbol,const void* src,
size_t count, size_t offset, enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyFromSymbol(void dst,const char* symbol,
size_t count, size_t offset, enum cudaMemcpyKind kind
```

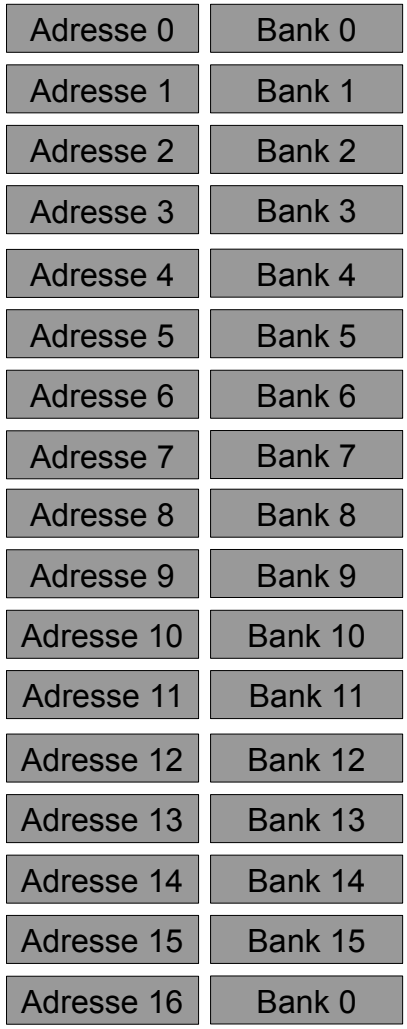
Merke: Im Host Code sind „Constant“ Variablen keine Variablen, sondern „symbols“

- ein „symbol“ ist vom Typ const char*
- versuche nicht Pointerarithmetik auf „symbols“ anzuwenden

Performance Optimierung II

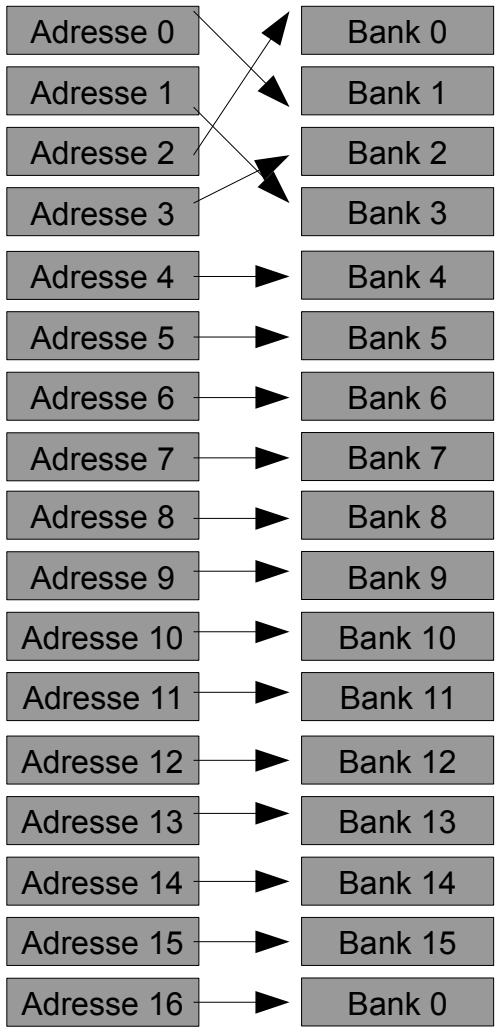
Shared Memory Bänke

- Shared Memory ist in 16 Bänke aufgeteilt:
  - Aufeinander folgende 32 Bit Wörter gehören zu aufeinander folgende Bänke
  - Wenn keine Bankkonflikte vorhanden sind, ist shared Memory so schnell wie Register
- Regel zur Auflösung von Konflikten:
  - In einem Half-Warp sollten keine 2 Threads auf die selbe Bank zugreifen
  - Ausnahme: ein und dasselbe 32 Bitwort kann von vielen Threads gelesen werden



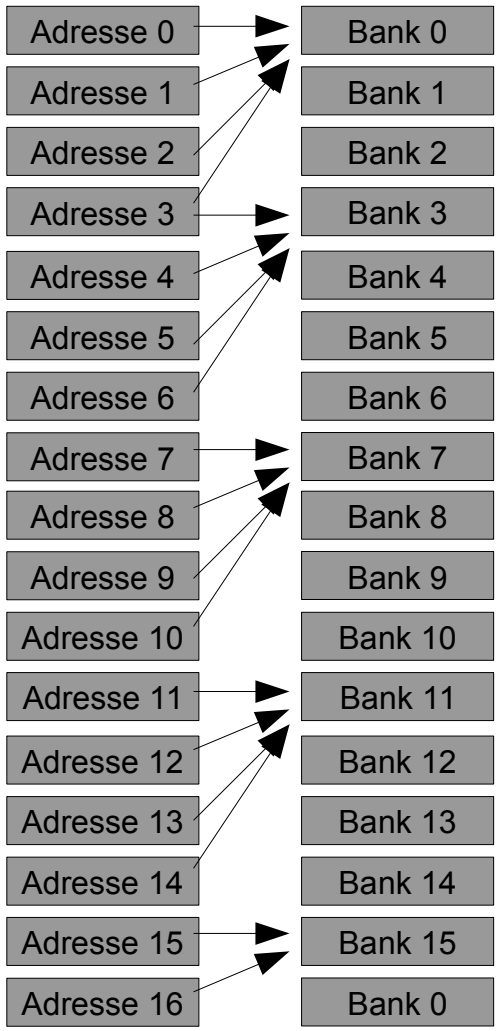
Konfliktfreier Memory Zugriff

- Konfliktfreier Zugriff
  - Jeder Thread greift auf unterschiedliche Bank zu
  - Permutationen des Zugriffs sind erlaubt



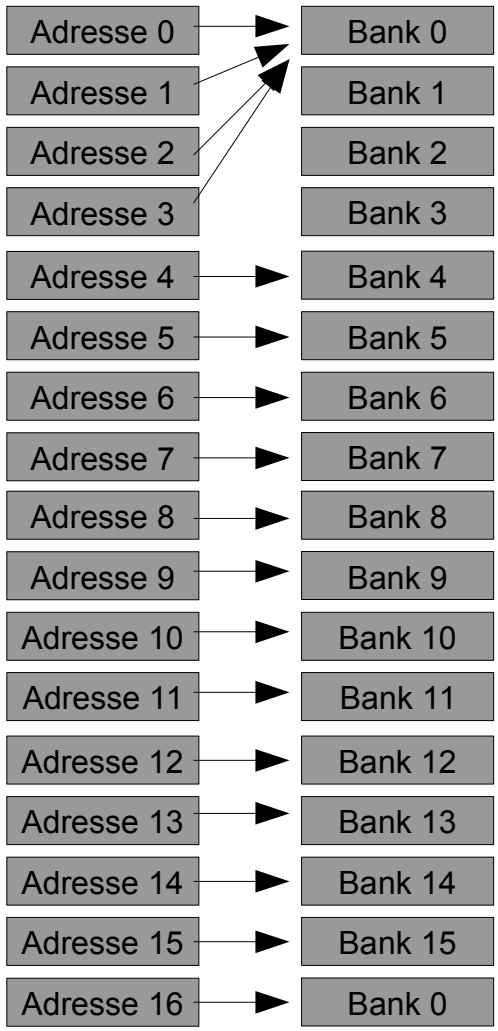
Konfliktbehafteter Memory Zugriff

- Konfliktbehafteter Zugriff
  - Mehrere Threads greifen auf die gleiche Bank zu



Shared Memory Zugriff

- Broadcast Mechanismus
  - Verschiedene Threads greifen auf die gleiche Adresse zu
  - Nur ein Speicherwort kann übertragen werden
  - Broadcast kann mit Zugriff auf konfliktfreie Banken gemischt werden
  - Ist so schnell wie konfliktfreier Zugriff!



Strided Shared Memory Zugriff

- Wenn man beim Zugriff auf das shared Memory Bänke überspringt:

```
__shared__ float x[blockDim.x*n];
… x[threadIdx.x*n]...
```

dann gibt es Bankkonflikte sobald n ein Vielfaches von 2 ist

- Dies gilt auch für 2 Dimensionale Arrays, wenn man threadIdx.x als ersten array Index benutzt.
- Wenn man strided Memory Zugriffe benötigt, sollte man Zugriffe auf ein vielfaches von 2 vermeiden...

Shared Memory Anwendungen

- In der Tesla Architektur (T10) ist das shared Memory Ersatz für den first level cache.
- Zerteilen von Problemen in Teilprobleme und Pufferung der Daten im Shared Memory kann einen enormen Speedup bedeuten.
- Genereller Ansatz:
  - Kopiere Memory vom globalen Memory ins shared Memory
  - Berechne viel auf dem shared Memory
  - Kopiere das Memory zurück
- Beispiele:
  - Dense Matrix Multiplication
  - FFT

Texture Memory

- Zugriff aufs Texture Memory ist nicht so schnell, wie der Zugriff aufs shared Memory.
- Aber texture Zugriffe sind immer „verschmolzen“ und gecached
- 2D oder 3D Texture
    - Cache ist optimiert für 2D bzw. 3D Zugriffe
    - Cache lines sind nicht linear sondern in 2D bzw. 3D organisiert

Unroll pragma

- Overhead beim Kontrollfluß ist auf der GPU wichtiger als auf der CPU
- So gibt es ein unroll pragma, welches hilft die Rechenzeit zu optimieren
- Beispiel:

#pragma unroll 8

```
for (int i=0;t<n;i++){
    …
}
```

PTX Assembler Output

- PTX Code ist eine Art Intermediate Code, aus dem der Device abhängige CUBIN Code generiert wird.
    - z.B. benutzt PTX virtuelle Register, die dann erst auf dem Device zugeordnet werden
- Das einsehen des PTX Outputs mag helfen, „Performance Bremsen“ zu identifizieren
- Die Option ‘-ptx‘ forciert den Compiler ein PTX Output-File zu erzeugen

Was ändert sich mit der Fermi Architektur?

- Read-Write-Cache
  - Benutzung des shared und texture memorys ist meist nicht nötig
  - L1 cache ist schneller als texture cache
  - → Man kann statt dessen Globales Memory nutzen
  - Faktor 2 Mehr Register, mehr shared Memory 48 kB
  - Mehr Blöcke bzw. Warps können parallel gescheduled werden.
- Shared Memory findet pro Warp statt pro Half-Warp statt
  - 32 statt 16 memory Bänke
  - Konflikte können zwischen ersten und 2. Halfwarp stattfinden
- 32 Bit Integer Multiplikation
  - 32 Bit Integer Multiplikation ist schneller als __mul24()
- Atmomic Operations für floats einfacher Genauigkeit

Zusammenfassung der Optimierungsstrategie

- Memory Nutzung:
  - Teilen von Blöcken im Shared Memory
  - Caching im Texture Memory
- Reduktion in 2 Kernel
  - 1. von Skalarprozessoren pro Thread, dann pro Multiprozessor
  - 2. über alle Multiprozessoren
- Fermi
  - Die meisten Memory Optimierungen sind unnötig

4 - CUDA

Debugging und Profiling

CUDA-GDB

- CUDA Device code kann geDebugged werden mit einer geänderten Version des GDB (CUDA-GDB)
- Der code muss mit nvcc -g -G ...compiliert werden
- Breakpoints, Variablen observierung etc funktioniert im Kernel normal

CUDA-GDB

- Getting started:
  - nvcc -g -G … -o <executable> (mit debug code compilieren)
  - cuda -gdb <executable> (debugger starten)
  - br <linenumber> (setzt breakpoints in Zeilennummern) oder
  - br <function> (breakpoint in Funktionen)
  - n (next line/ nächste Zeile) oder
  - c (continue/ weiter)
  - print <variable oder Expression>
  - display <variable oder Expression>
  - thread <<<(bx,by),(tx,ty,tz)>>> (schaltet den thread um
  - help (zeigt die Hilfe an)

CUDA-GDB

- CUDA spezifische features:
  - cuda block (schreibt den aktuellen Blockindex auf den Bildschirm)
  - cuda thread (schreibt den aktuellen threadindex auf den Bildschirm)
  - cuda block thread (schreibt beides)
  - print gridDim, print blockDim
  - cuda block(<x><y>) (schaltet zu block x,y um)
  - Cuda thread (<x><y><z>) (schaltet zu thread x,y,z um)

CUDA-memory checking

- 2 Möglichkeiten:
  - Methode 1: innerhalb cuda-gdb cuda memcheck auf on schalten
  - Methode 2: Das tool cuda-memcheck <executable>

cuPrintf

- Printf Funktion für device code
- Auf Fermi Architektur (Compute Capability>=2.0) kann normales printf verwendet werden

Kernel time Funktion

- Im kernel code gibt die Funktion clock() den GPU time counter zurück
- Ergebnis ist die Core Clock Zyklen
- Die Taktrate bekommt man mit: cudasetDeviceProperties()

NVIDIA Visual Profiler

- Nimmt einige GPU Performance relewandte Informationen während der Ausführung des Programms auf
- Gebrauch:
  - Cudaprof
  - File → New
  - Wähle Project File und Directory
  - Im Tab ‘Session‘ muss noch der Ausführungspfad und optional Arbeitsverzeichnis und Programmargumente übergeben werden
  - In Tab ,‘Profiler Counters‘ und ,‘other Options‘ müssen noch die Informationen angegeben werden, die aufgenommen werden sollen.
  - Start drücken

NVIDIA Visual Profiler

gld uncoa esced	Number of non-coalesced global memory loads
gld coalesced	Number of coalesced global memory loads
gld request	Number of global memory load
gld_32/64/128b	Number of 32 byte, 64 byte and 128 byte global memory load transactions
gst uncoalesced	Number of non-coalesced global memory stores
gst coalesced	Number of coalesced global memory stores
gst request	Number of global memory store requests
gst_32/64/128b	Number of 32 byte, 64 byte and 128 byte global memory store transactions
local load	Number of local memory loads
local store	Number of local memory stores
tlb hit	Number of instruction or constant memory cache hits
sm cta launched	Number of instruction or constant memory cache misses
branch	Number of threads blocks launched on a multiprocessor
divergent branch	Number of divergent branches within a warp
instructions	Number of instructions executed
warp serialize	Number of thread warps that serialize on address conflicts to either sharec or constant memory
cta launched	Number of threads blocks executed

Aus dem CUDA Workshop von Oliver Mangold am HLRS: http://www.hlrs.de/no_cache/events/training/

NVIDIA Visual Profiler

- Die Hardware hat eine limitierte Anzahl von Profilingregister
- Da Programm wird mehrere Male ausgeführt um die Informationen zu sammeln
- Die Ergebnisse mögen unbrauchbar sein, wenn sie nicht gleich ablaufen
    - Zufallsgenerator oder OpenMP dynamisches scheduling
- Interessante Felder sind
    - Profiler output
        - Static und shared ;Memory pro Block
        - Register pro Thread
        - Occupancy (Anzahl der parallelen Warps/ Maximal mögliche parallele warps
- Summary Table
    - Global mem...throughput
    - GPU time plot
        - Computation time vs. Data transfertime

Profiler counter Funktion

- Jeder Profiler hat 16 profiler counter
- Counter 0-7 können vom Programmierer benutzt werden
- Counter 8-15 sind vom System reserviert
- Die Funktion __prof_trigger(int counter); inkrementiert den counter mit index counter um 1 für jeden Thread, der diesen Befehl ausführt
- Einzusehen im Tab ‘Profiler Counters‘
- Merke: Die Ergebnisse beziehen sich nur auf den ersten Multiprozessor

Command Line Profiling

- CUDA Profiling kann auch in der Kommandozeile ohne visual Profiler aktiviert werden
- Setze die environment Variable: CUDA_PROFILE=1
- Profiling informationen werden in ein Log File geschrieben
- Zusätzliche Optionen
  - CUDA_PROFILE_CSV=1 (output im CSV Format für den Import in Visual Profiler)
  - CUDA_PROFILE_LOG=<filename> setzt den Log-Filename
  - CUDA_PROFILE_CONFIG=<filename> setzt Profiler Konfigurationsfile

Profiler Konfigurationsfile

- Das Profiler Konfigurationsfile ist eine einfache Liste von records und countern, die ins Logfile geschrieben werden sollen:

```
timestamp, gpustarttimestamp, gpuendtimestamp, streamid,
gridsize,threadblocksize,dynsmemperblock,stasmemperblock,regperthread,
memtransferdir, memtransfersize, memtransferhostmemtype,
local_load, local store, gld_request, gst_request,
divergent_branch, branch, sm_cta_launched,
gld_incoherent, gld_coherent, gld_32b, gld_64b, gld_128b,
gst_incoherent, gst_coherent, gst_32b, gst_64b, gst_128b,
instructions, warp_serialize, cta_launched,
prof_trigger_00...prof_trigger_07,
tex_cache_hit, tex_cache_miss, shared_load, shared_store,
inst_issued, inst_executed, warps_launched, threads_launched,
l1_global_load_hit, l1_global_load_miss
```

- Devices der Compute Capability 1.x können maximal 4 counter verwenden

Aus dem CUDA Workshop von Oliver Mangold am HLRS: http://www.hlrs.de/no_cache/events/training/

4 - CUDA

Streams und weitere Features

Hardware Information abfragen

- Anzahl der devices im System:

```
cuDeviceGetCount(int *count);
```

- Name der Hardware:

```
cuDeviceGetName(char* name, int len, Cudevice dev);
```

- Device-Eigenschaften

```
cudaGetDeviceProperties(cudaDeviceProp* prop, Cudevice dev);
```

Interesting members of `cudaDeviceProp` (incomplete list):

```
char name[256];
size_t totalGlobalMem; size_t sharedMemPerBlock;
int regsPerBlock; int warpSize; int maxThreadsPerBlock;
int maxThreadsDim[3]; int maxGridSize[3]; // maximum block/grid sizes
int clockRate; // GPU core clock frequency
size_t totalConstMem;
int major; int minor; // compute capability
size_t textureAlignment; // alignment of texture base addresses
int deviceOverlap; // can overlap copy with kernel execution
int multiProcessorCount;
int canMapHostMemory; // zero-copy host memory access works
int maxTexture1D; int maxTexture2D[2]; // maximum texture sizes
int maxTexture3D[3];
int maxTexture2DArray[3];
int concurrentKernels; // multiple kernels can execute concurrently (Fermi)
```

Aus dem CUDA Workshop von Oliver Mangold am HLRS: http://www.hlrs.de/no_cache/events/training/

Streams

- Streams erlauben gleichzeitig Memory Kopier-Transaktionen und Kernel Aufrufe
- cudaStream_t stream;
- Jede auf dem device ist an einen stream gehängt. Aufrufe, die kein stream handle mitgeben werden an den default stream ‘stream 0‘ gehängt
- Operationen, die an ein und dem selben Stream gehängt sind werden nacheinander ausgeführt
- Operationen verschiedener Streams werden parallel ausgeführt

Nutzung von Streams

- Erzeugung:

```
cudaStream_t sttream;
cudaStreamCreate(&stream);
```

- Cleanup:

```
cudaStreamDestroy(&stream);
```

- Asynchrones Kopieren an einen stream hängen:

```
cudaMemcpyAsync(dest, src, size, type, stream);
```

Merke: Parallel arbeitet dieser Befehl nur mit page locked Memory

- Kernel call an einen stream gehängt:

```
MyKernel<<<grid, block, shared_mem, stream>>>(....);
```

- Wartet auf alle Operationen eines bestimmten Streams

cudaThreadSynchronize();

Nutzung von Streams

- Parallelität zwischen Kernel Ausführungen und dem kopieren von Daten von/auf den device ist nur möglich, wenn die device Property ‘deviceOverlap‘ gesetzt ist.
- Parallelität zwischen verschiedenen Kernel Ausführungen ist nur möglich, wenn die device Property ‘concurrentKernels‘ gesetzt ist.
- Wenn man die environment Variable CUDA_LAUNCH_BLOCKING=1 setzt, wird die parallele Ausführung unterschiedlicher Kernel ausgeschaltet

Nutzung mehrerer GPUs

- Einschränkungen
  - Jeder Host Thread kann nur eine GPU zu einer Zeit nutzen
  - → man starte so viele Host Threads wie GPUs im System sind.

Atomic Befehle

- Die Folgenden atomic Befehle stehen in CUDA Device Code zur Verfügung
  - atomicAdd(int32,uint32,uint64)
  - float auf Fermi aber kein double
  - atomicSub(int32,uint32)
  - atomicExch(int32,uint32,uint64,float)
  - atomicMin, atomicMax (int32,uint32)
  - atomicInc,atomicDec (uint32)
  - atomicAnd, atomicOr, atomicXor (int32, uint32)
- 32 Bit Befehle stehen für globales und shared Memory zur Verfügung
- 64 Bit Befehle stehen nur für globales Memory zur Verfügung
- Anwendungen
  - Reduktionen z.B. von grossen Summen
  - Globale Zähler

Host Memory Zugriffe ohne kopieren

- Auf Page Locked Host Memory kann direkt von der GPU zugegriffen werden, wenn sie mit cudaHostAlloc() reserviert und das cudaHostAllocMapped Flag gesetzt ist.
- Bevor der CUDA Aufruf stattfinden kann muß das Flag cudaDeviceMapHost gesetzt werden
- Die Funktion cudaHostGetDevicePointer() kann die Host Adresse zu einer device Adresse konvertieren.
- Beispiel:

```
cudaDeviceFlags(cudaDeviceMapHost);
float * aHost, *adevice;
vudaHostAlloc((void**)&aHost, size,cudaHostAllocMapped);
cudaHostGetDevicePointer((void**)&aDevice,aHost,0);
...kernel call ...
```

C++ classes

- Für Devices der Compute Capability >=2.0 ist es möglich nicht polymorphe C++ Klassen zu nutzen
- Syntax:

```
class test{
  int x,y,z;
  __device__ test(...){...}// constructor als device Funktion
  __device__ void f(...){...} // method as device function
}
```

Numerische Libraries

- CuFFT
  - Fast Fourier Transform Library from NVIDIA
- CuBLAS
  - Linear Algebra Unterprogramme
- CUDA-accellerated LAPACK: CULA (Kommerziell)
  - http://www.culatools.com (High level Lineare Algebra library)

4 - CUDA

Open CL als Alternative

Alternative zu CUDA - OpenCL

- Open-CL ist ähnlich zu CUDA aber:
  - Mehr Initialisierungsoverhead
  - Prinzipiell Platformunabhängig
    - Effiziente Implementierung könnte nicht möglich sein
    - SIMT Ausführungsmodell ist dem User nicht transparent dargelegt
      - Es gibt kein Warp Konzept

Open-CL Beispiel

```
// create a compute context with GPU
devicecontext = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// create a command queue
queue = clCreateCommandQueue(context, NULL, 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(float)*2*num_entries, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*2*num_entries, NULL, NULL);

// create the compute program
program = clCreateProgramWithSource(context, 1, &fft1D_1024_kernel_src, NULL, NULL);

// build the compute program executable
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the compute kernel
kernel = clCreateKernel(program, "fft1D_1024", NULL);

// set the args values
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
clSetKernelArg(kernel, 2, sizeof(float)*(local_work_size[0]+1)*16, NULL);
clSetKernelArg(kernel, 3, sizeof(float)*(local_work_size[0]+1)*16, NULL);

// create N-D range object with work-item dimensions and execute kernel
global_work_size[0] = num_entries;
local_work_size[0] = 64;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size, local_work_size, 0, NULL, NULL);
```

Aus dem CUDA Workshop von Oliver Mangold am HLRS: http://www.hlrs.de/no_cache/events/training/