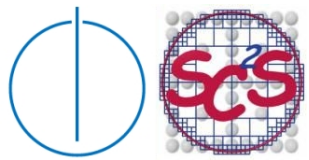
Masterpraktikum Scientific Computing

High-Performance Computing

Thomas Auckenthaler
Wolfgang Eckhardt
Prof. Dr. Michael Bader

Technische Universität München, Germany



Outline

Organisatorisches

Entwicklung General Purpose GPU Programming (GPGPU)

C for CUDA

Organisatorisches

- Diese Woche: Programmierung mit NVidia CUDA
- Di, 15.12.09: Besprechung der Lösung Blatt 3
- Di, 22.12.09: Ausgabe der Projektaufgabe
- Di, 12.01.09: Abgabe der Lösung zu CUDA

Outline

Organisatorisches

Entwicklung General Purpose GPU Programming (GPGPU)

C for CUDA

Herkömmliche Implementierung der Graphik-Pipeline

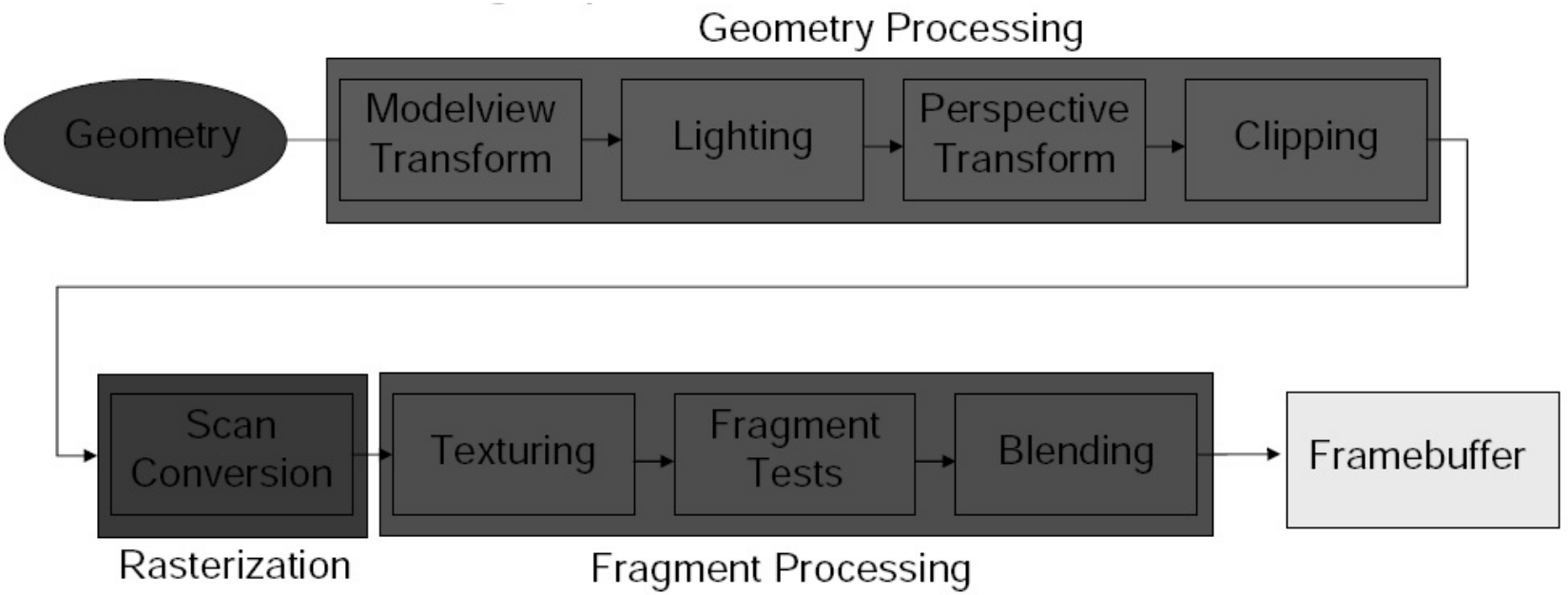
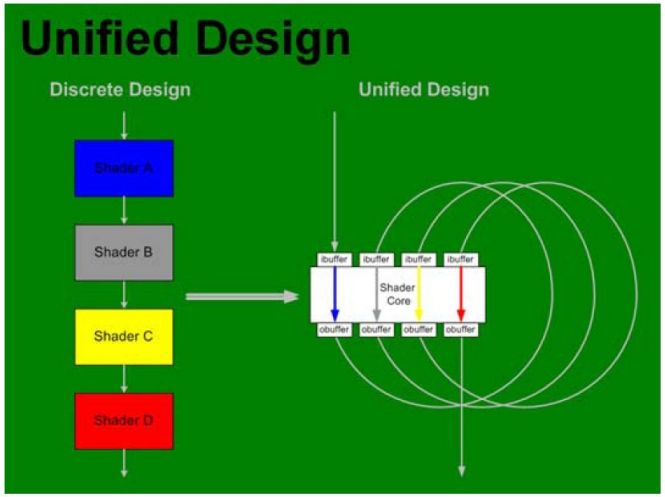
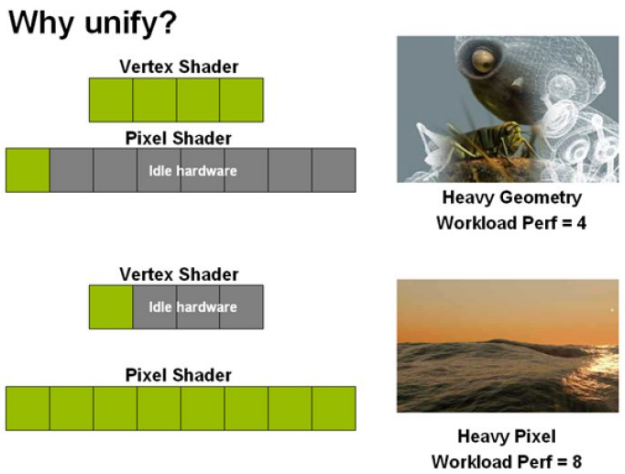


Figure: Klassische Graphikpipeline - Ein Prozessor (Shader) pro Stufe

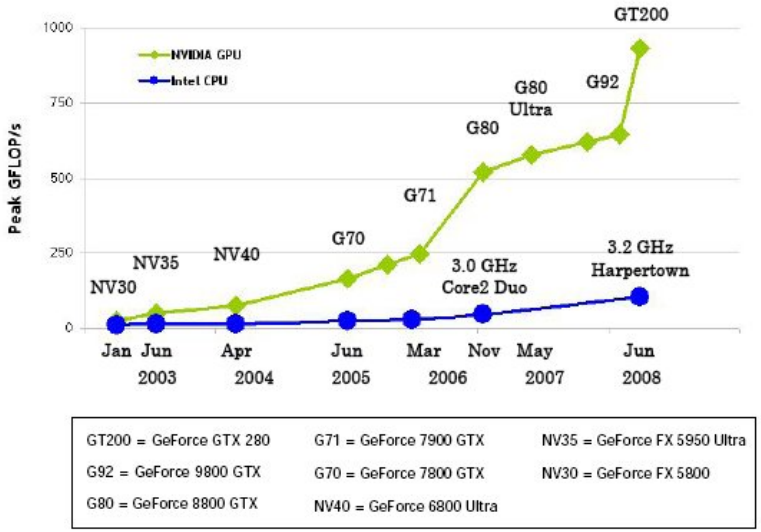
“Unified Architecture“

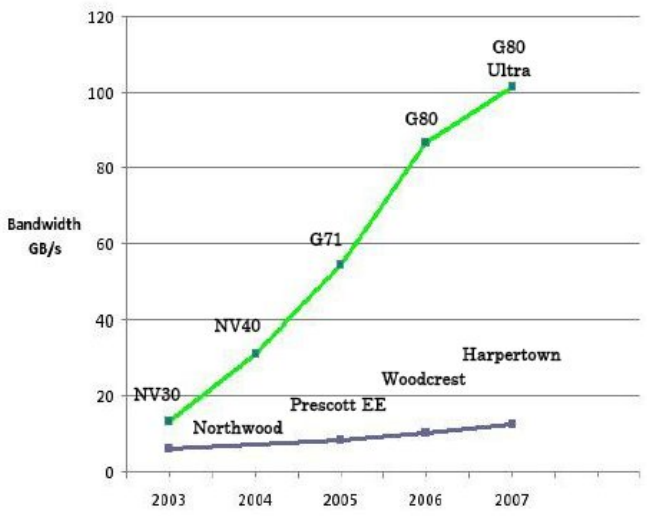
- Herkömmlich Einschränkungen durch HW (fixed-function pipeline), gleichzeitig z.T. redundante Logik
- NVidia Lösung: Unified Device Architecture (UDA)
- Implementierungen: G80 (2006), GT200 (2007), Fermi (2009)
- ATI / AMD: Unified Shader Architecture, z.B. HD-2000 Reihe, HD-3000 Reihe





Vergleich CPU - GPU





	Intel Xeon X5355	NVidia Tesla C1060 (T10)
Clock Speed	2.66 GHZ	1296
#Cores / SPEs	4	240
theor. Peak Perf.	85	933
Price	ca. 1100 USD	1200 EUR

GPU Programming

- Shader Programmierung, sehr hardwarenah:
  - Microsoft HLSL
  - NVidia Cg
- OpenCL (Open Computing Language)
  - Heterogene, parallele Platformen mit CPU's, GPU's, DSP's, ...
  - 8. Dezember 2008
  - Apple, Intel, IBM, Nvidia, AMD
- Brook (Stanford University)
- NVidia: CUDA (C for Cuda, Cuda Driver API, CUBLAS, CUFFT)
- ...

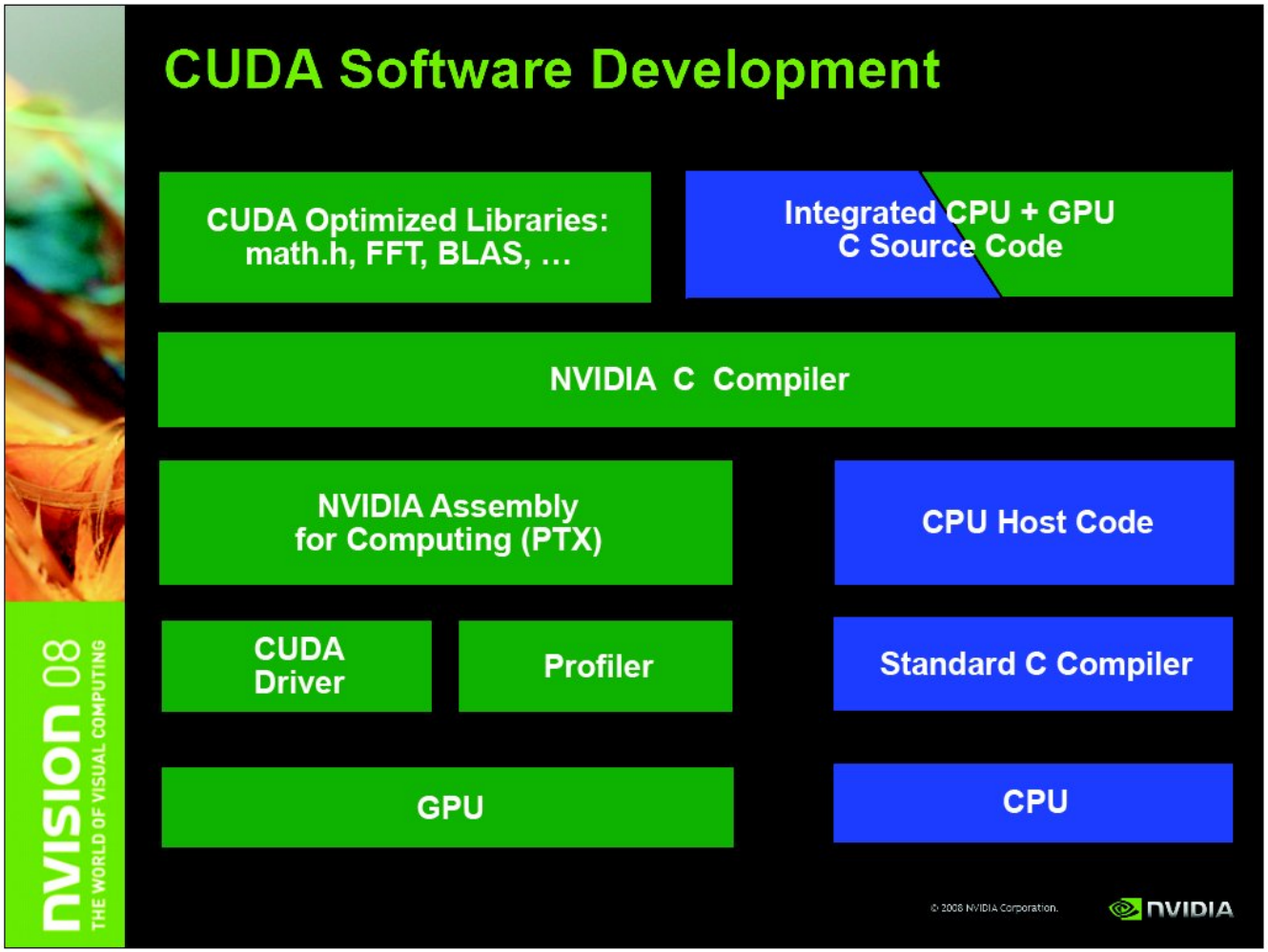
Outline

Organisatorisches

Entwicklung General Purpose GPU Programming (GPGPU)

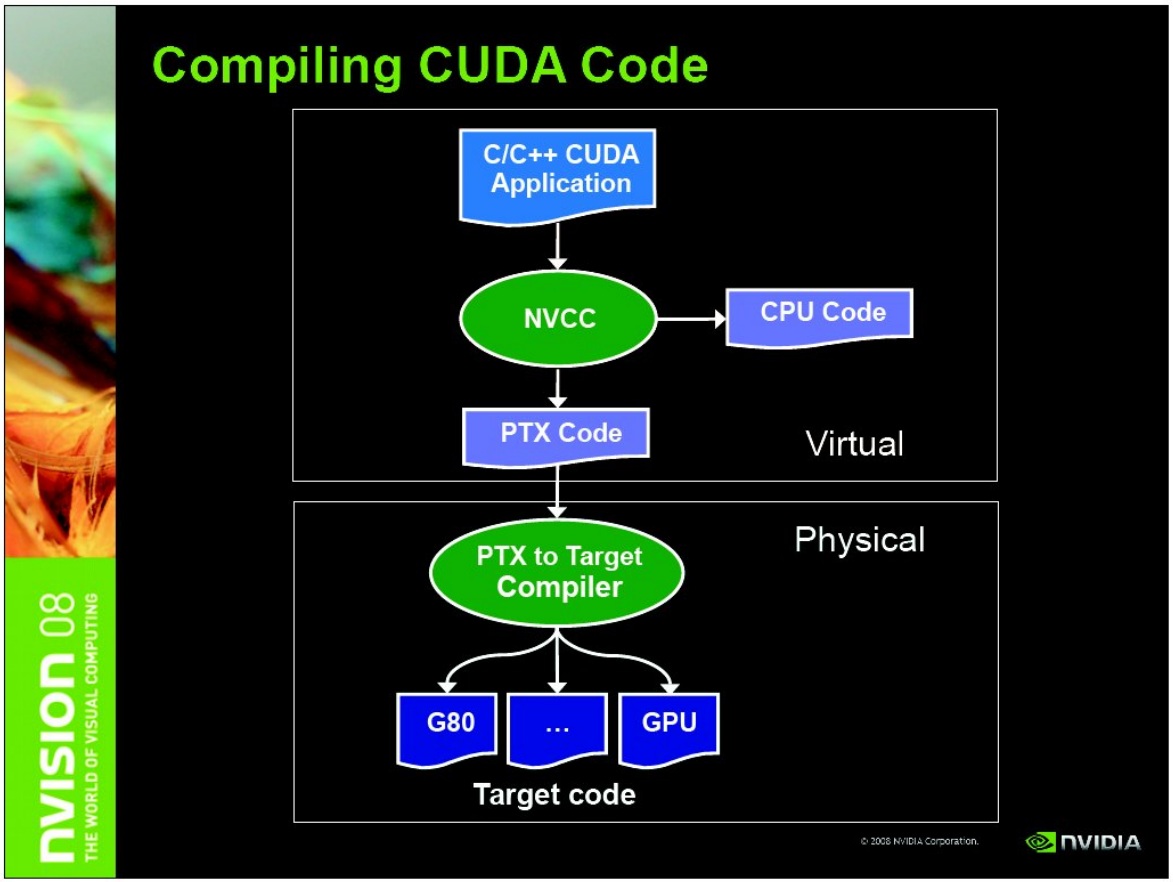
C for CUDA

C for CUDA


CUDA Software Development
CUDA Optimized Libraries: math.h, FFT, BLAS, …
Integrated CPU + GPU C Source Code
NVIDIA C Compiler
NVIDIA Assembly for Computing (PTX)
CPU Host Code
CUDA Driver
Profiler
Standard C Compiler
GPU
CPU
NVISION 08
THE WORLD OF VISUAL COMPUTING
© 2008 NVIDIA Corporation.
NVIDIA

C for CUDA - Übersetzung

- Cuda-Programm wird als C-Programm mit geringfügig erweiterter Syntax geschrieben.
- nvcc test.cu -o test

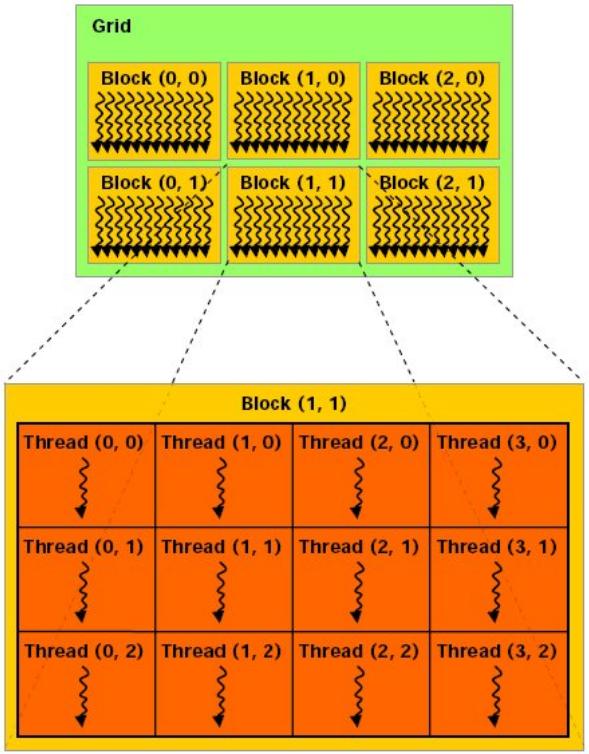


Kernels: Bsp. Vektoraddition

```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

- Kernel wird parallel von N Threads ausgeführt
- In Kernel Built-in Variables verfügbar (**threadIdx.x**)
- Kernels sind C-Funktionen mit zusätzlichem Modifier **__global__**
- Modifier gibt an, wo Funktion lebt:
  - __global__ : wird auf Graphikkarte ausgeführt, von Host aus aufgerufen
  - __device__ : kann nur von Kernels aufgerufen werden
  - __host__ : kann nur auf Host aufgerufen werden

Thread-Hierachie



- Threads werden in 1d/2d/3d Blöcke unterteilt, Blöcke liegen in 1d/2d Gittern; wird bei Aufruf des Kernels spezifiziert

```
dim3 block(w, h, d);
dim3 grid(w2, h2);
myKernel<<<grid, block>>>( ...);
```

- Konfiguration innerhalb Kernels durch built-in Variablen zugänglich

threadIdx	Thread-Index
blockIdx	Block-Index
blockDim	Dimensionen eines Blocks
gridDim	Dimensionen des Grids

- Synchronisation innerhalb eines Blockes mit `__syncthreads()`

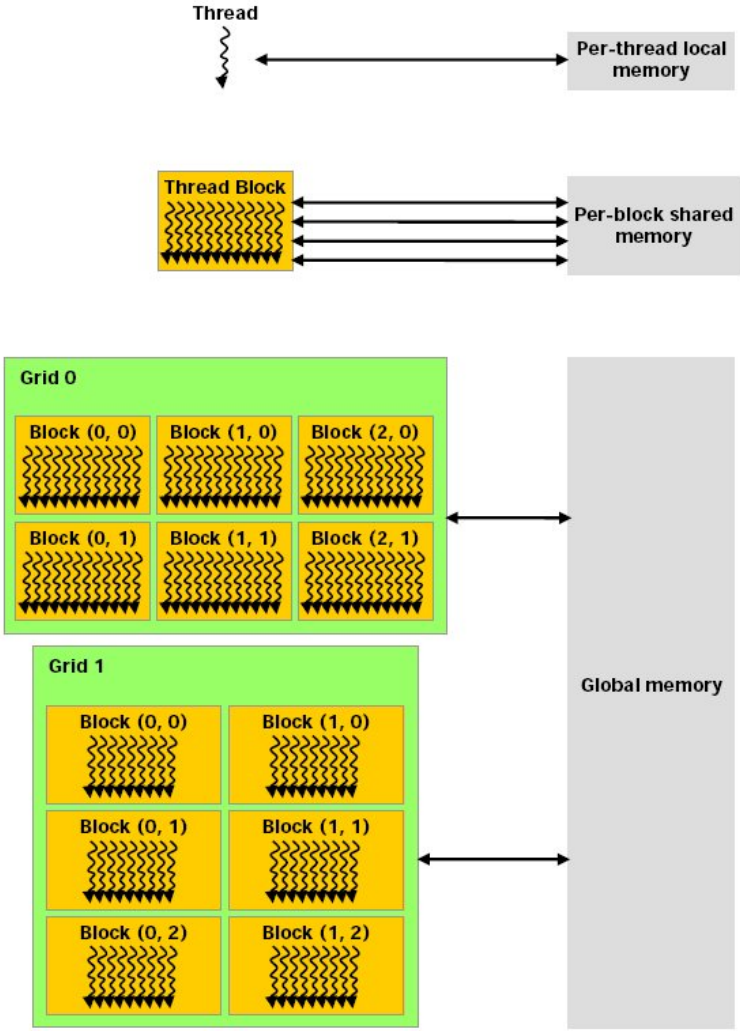
Kernels

```
__global__ void vecAddGPU(float* A,
                          float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

__host__ void vecAdd(int n, float* A,
                     float* B, float* C) {
    dim3 grid(1);
    dim3 block(n);
    vecAddGPU<<<gid,block>>>(A, B, C);
}
```

- Wie könnte eine Aufteilung für die Addition von $n \times n$–Matrizen aussehen?
- Beachte: Einschränkungen durch HW (z.B. nur 512 Threads pro Block)

Speicher-Hierachie



- globaler Speicher langsam, nicht gecached
- shared memory: modifier `__shared__`
- außerdem: Textur-Speicher, Constant-Memory

Speicher-Verwaltung

- Allokation von globalem Graphikspeicher

```
float* ptr;
cudaMalloc(&ptr, n * sizeof(float));
```

- Kopieren

```
cudaMemcpy(void* target, void* source,
           int size, enum cudaMemcpyKind);
```

  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

- Freigabe von globalem Graphikspeicher

```
cudaFree(void* ptr);
```

Error Checking and Debugging

- Alle Runtime-Funktionen geben Fehlercode zurück.
- Problematisch bei asynchronen Funktionsaufrufen. Daher: `cudaThreadSynchronize()` in Verbindung mit `cudaGetLastError()`.
- Debugging: übersetze Code mit `--device-emulation`: Graphikkarte wird emuliert, Makro `__DEVICE_EMULATION__` definiert
- z.B. printf() oder Debugger möglich

```
#ifdef __DEVICE_EMULATION__
  printf("Variable i: %d\n", i);
#endif
```

Performance - Hinweise

- Speicher:
  - Verschmelzen von Zugriffen auf globalen Speicher
  - Speicherart
  - Bankkonflikte

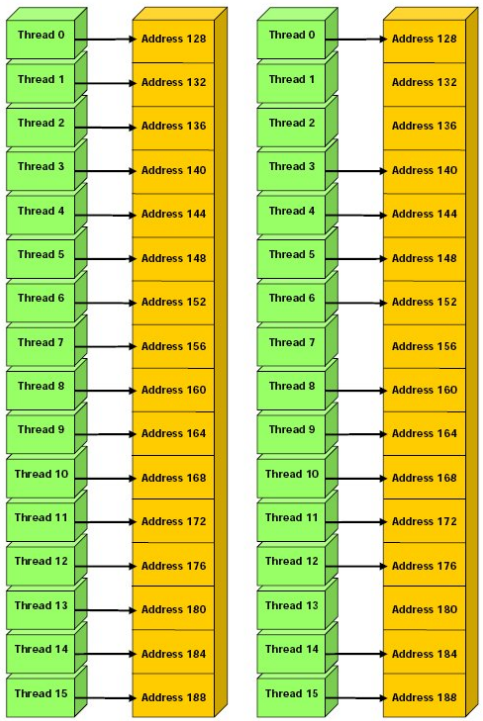


Figure: zu einem Speicherzugriff verschmolzen

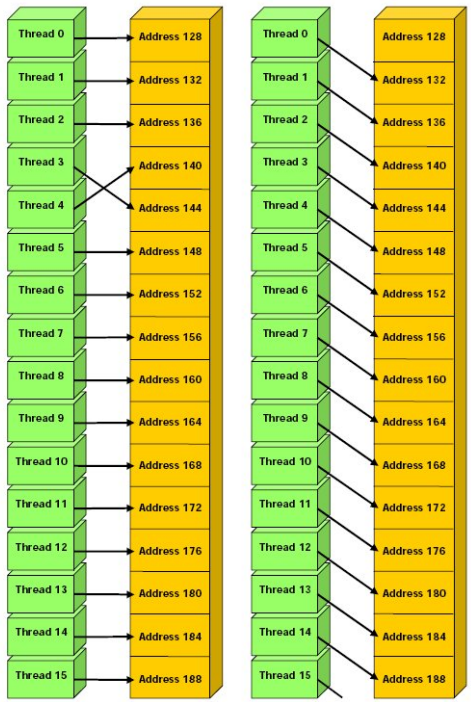
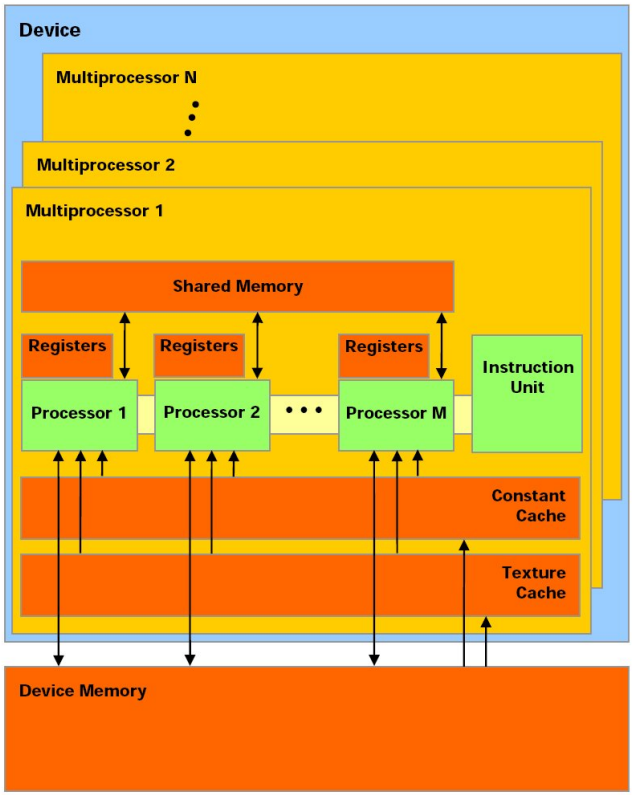


Figure: resultiert in 16 Speicherzugriffen (CUDA 1.0/1.1

Performance - Hinweise

- Threads eines Blockes unterteilt in *Warps* a 32 Threads
- Code-Verzweigungen / ifs...
- genügend Blöcke
- Latenzen verbergen



Hinweise - Aufgaben

- Nutzung des Codes zum Einlesen einer .pgm-Datei möglich, nachdem die Implementierung zum .cpp umbenannt wurde
- Entwicklung auf `atsccs30` (Quadro NVS 290)
- Test auf `gvs2.lrz-muenchen.de` (Quadro FX 5800)
- Auf gvs2 existiert kein Reservierungssystem, daher:
  - prüfe mit `top` last
  - prüfe mit `lsof /dev/nvi*`, dass niemand sonst die Graphikkarten benutzt (Ausgabe 4 Zeilen)
  - `module load cuda`
  - `module load gcc`
- Nutzung von 4 GPUs auf gvs2 möglich!
- Große Bilder z.B. unter `http://visibleearth.nasa.gov`