

Masterpraktikum Scientific Computing

High-Performance Computing

Thomas Auckenthaler
Wolfgang Eckhardt

Technische Universität München, Germany



Outline

Entwicklung General Purpose GPU Programming (GPGPU)

C for CUDA

Herkömmliche Implementierung der Graphik-Pipeline

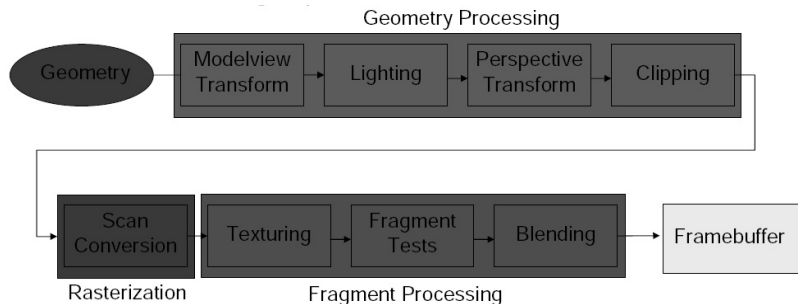
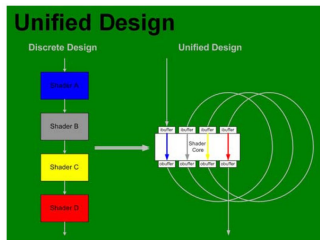


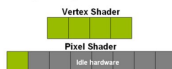
Figure: Klassische Graphikpipeline - Ein Prozessor (Shader) pro Stufe

“Unified Architecture“

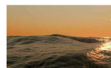
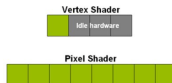
- Herkömmlich
Einschränkungen durch HW
(fixed-function pipeline),
gleichzeitig z.T. redundante
Logik
- NVidia Lösung: Unified
Device Architecture (UDA)
- Implementierungen: G80
(2006), GT200 (2007), Fermi
(2009)
- ATI / AMD: Unified Shader
Architecture, z.B. HD-2000
Reihe, HD-3000 Reihe



Why unify?

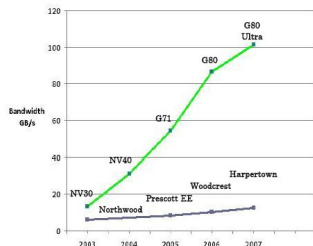
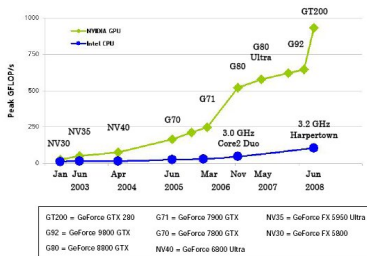


Heavy Geometry
Workload Perf = 4



Heavy Pixel
Workload Perf = 8

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

Table: Preise Mitte 2009

GPU Programming

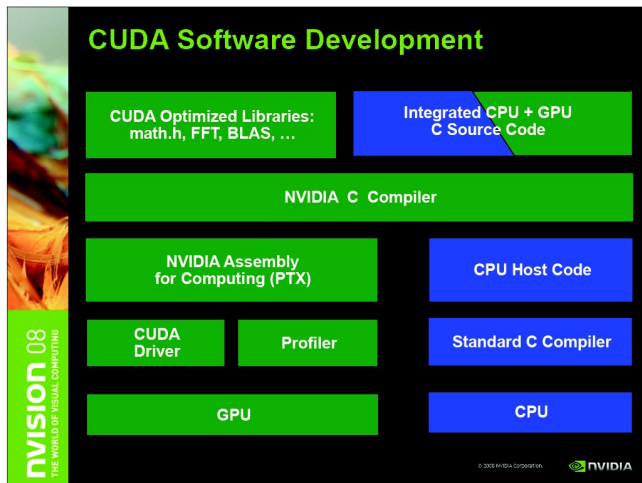
- Shader Programmierung, sehr hardwarenah:
 - Microsoft HLSL
 - NVidia Cg
- OpenCL (Open Computing Language)
 - Heterogene, parallele Plattformen mit CPU's, GPU's, DSP's,
...
 - 8. Dezember 2008
 - Apple, Intel, IBM, Nvidia, AMD
- Brook (Stanford University)
- NVidia: CUDA (C for Cuda, Cuda Driver API,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

Outline

Entwicklung General Purpose GPU Programming (GPGPU)

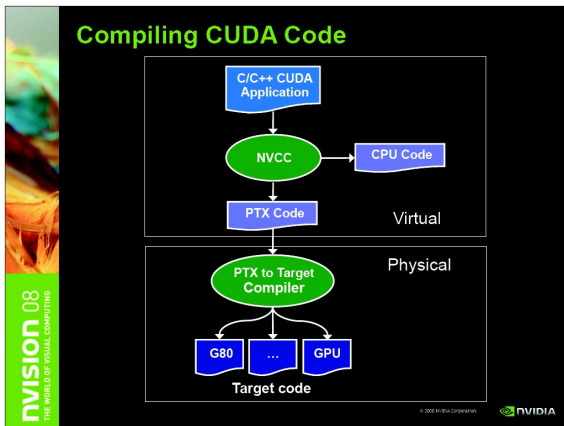
C for CUDA

C for CUDA



C for CUDA - Übersetzung

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

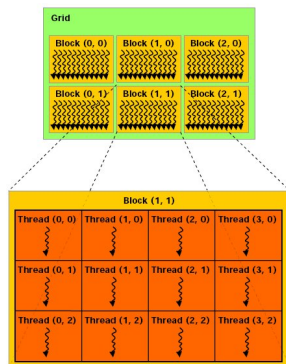


Kernels: Bsp. Vektoraddition

```
__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-Hierarchie



- 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()`

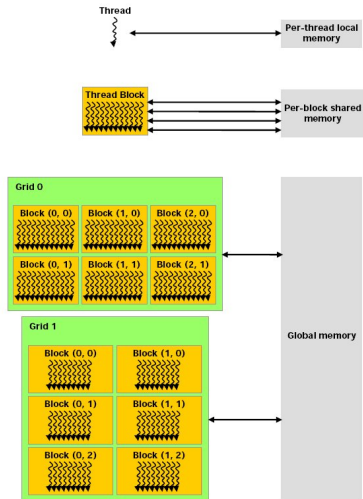
Kernels

```
__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<<<grid,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-Hierarchie



- 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

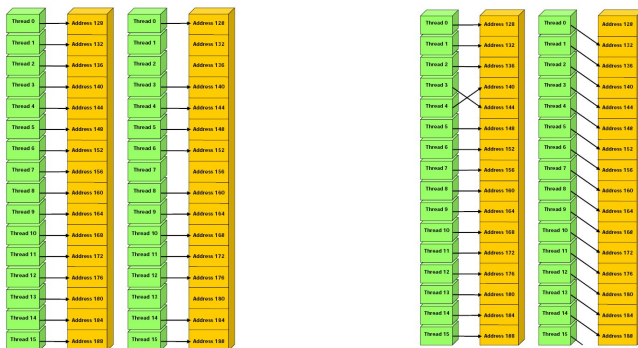
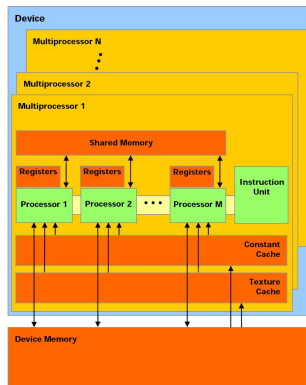


Figure: zu einem Speicherzugriff verschmolzen

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



Fermi / CUDA 3.0

- kein device emulation mode mehr
- cuda-gdb (erfordert Graphikkarte ohne X)
- memory-checker:

```
Cuda error: Kernel Execution
: unspecified launch failure.
===== Invalid read of size 4
=====          at 0x00000108 in gauss_cuda_kernel
=====          by thread 917504 in block 1
===== Address 0x000186a8 is out of bounds
=====
===== ERROR SUMMARY: 1 errors
```

- printf() als Befehl in HW unterstützt
- konfigurierbare, "echte" Caches
- ECC Speicherschutz
- 32 Threadprozessoren pro SM
- Gleichzeitige Ausführung von Kernels

Hinweise - Aufgaben

- Evtl. Nutzung von `cudaMallocPitch()`
- Test auf `gvs1.lrz-muenchen.de` und `gvs2.lrz-muenchen.de` (Quadro FX 5800)
- Auf `gvs1/2` existiert kein Reservierungssystem, daher:
 - prüfe mit `top last`
 - prüfe mit `lsOf /dev/nvi*`, dass niemand sonst die Graphikkarten benutzt (Ausgabe 2 Zeilen)
 - `module load cuda (default 2.3)`
 - `module load gcc`
- Nutzung von 2 GPUs auf `gvs2` möglich!
- Große Bilder z.B. unter `http://visibleearth.nasa.gov`