Einführung CUDA

Ralf Seidler

Lehrstuhl Rechnerarchitektur und Advanced Computing

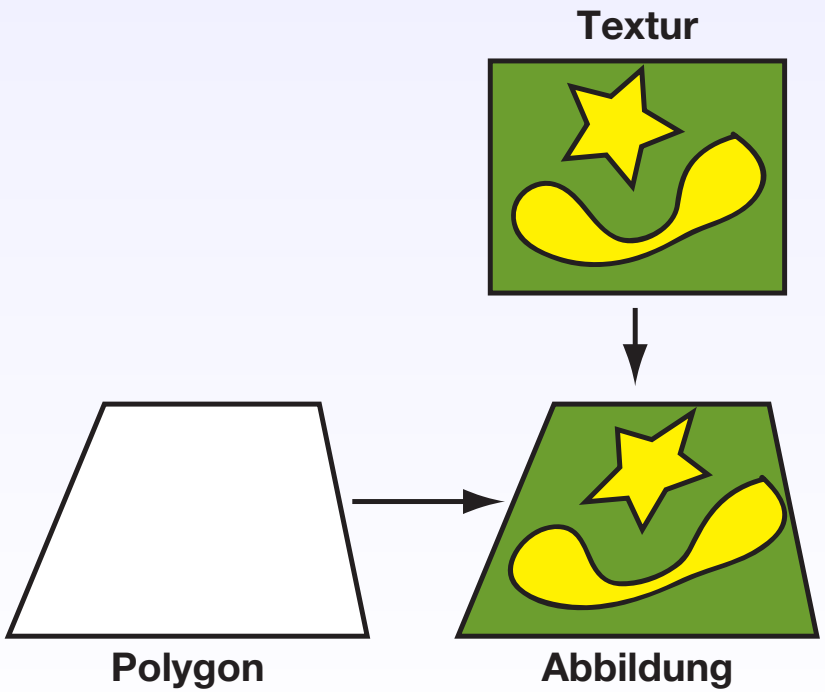
15. Juni 2015

Outline

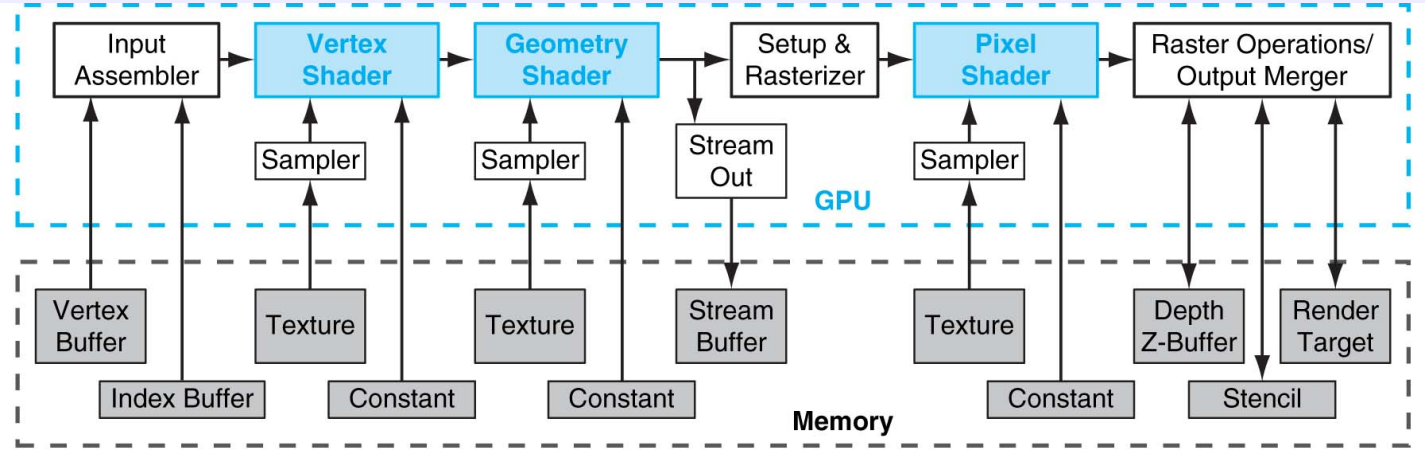
Outlook

Eine kurze Geschichte der Grafikkarten

- Ursprünglich: *Video Card* steuert Monitor an
- Mitte 80er: Grafikkarten mit 2D-Beschleunigung
  - angelehnt an Arcade- und Home-Computer
- Frühe 90er: erste 3D-Beschleunigung:
  - Matrox Mystique, 3dfx Voodoo
  - Rastern von Polygonen



Eine kurze Geschichte der Grafikkarten

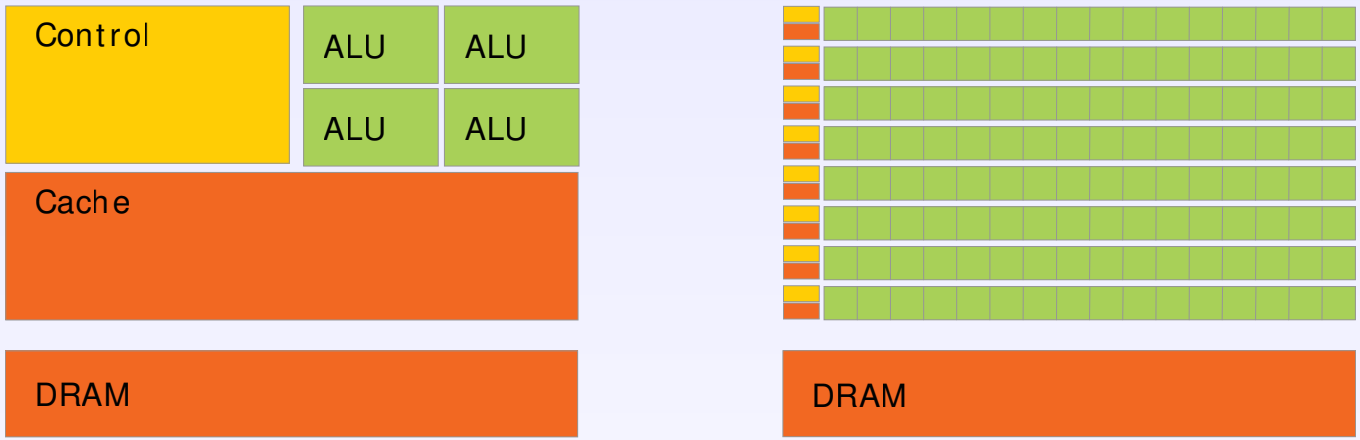


Direct3D 10 Pipeline

Eine kurze Geschichte der Grafikkarten

- 2000er:
  - Zunächst nur Fixed-Function-Pipeline
  - Shader-Programme bieten mehr Flexibilität
  - Shader-Programme ursprünglich nur einfache Listen
  - 2002: ATI Radeon 9700 kann Loops in Shadern ausführen
- Heute:
  - Shader turing-vollständig
  - Hersteller: Intel, ATI und NVIDIA
  - Massenmarkt $\rightarrow$ niedrige Preise

CPU vs GPU



- CPU: Wenige ALUs, große Steuerung und riesiger Cache
- GPU: Viele kleine ALUs, einfache Steuerung, kaum Cache

Klassifikation nach Flynn

	Single Instruction (SI)	Multiple Instruction (MI)
Single Data (SD)	SISD	MISD
Multiple Data (MD)	SIMD	MIMD

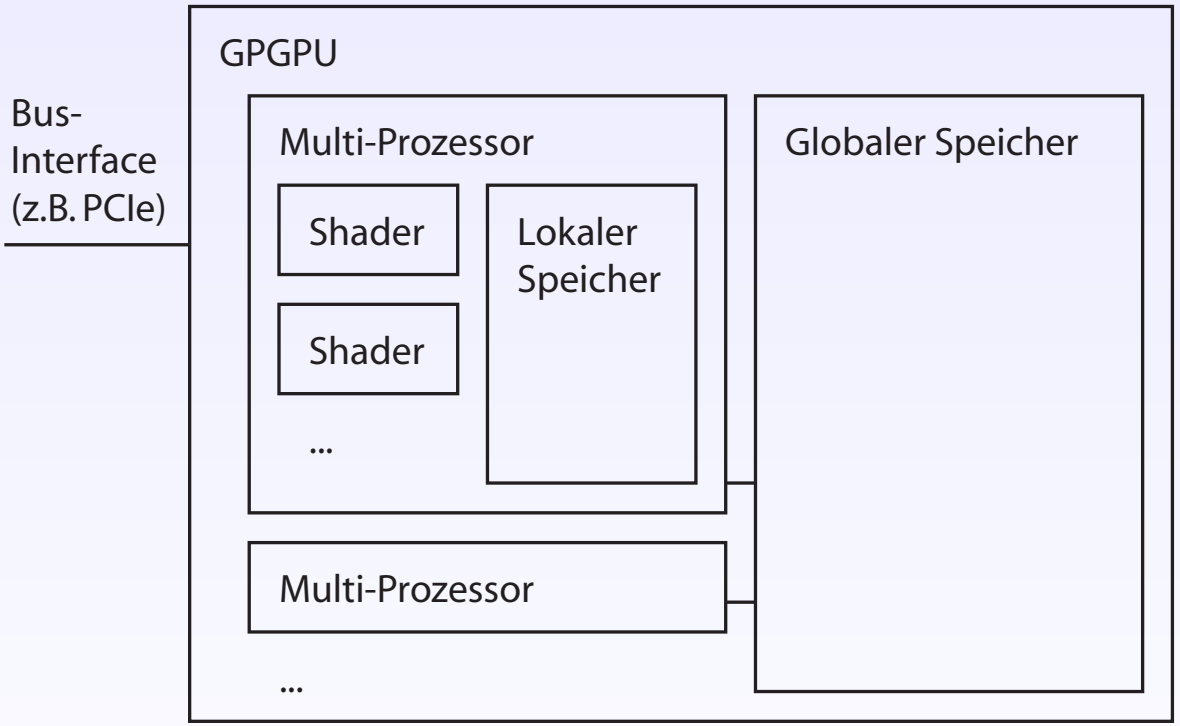
Klassifikation nach Flynn

	Single Instruction (SI)	Multiple Instruction (MI)
Single Data (SD)	μC CPU	Fehlertolerante Systeme
Multiple Data (MD)	GPUs SSE	Multicore-CPU

GPGPUs

- GPGPU = General Purpose Graphics Processing Unit
- Grafikkarten zunehmend flexibler programmierbar
- Stetig wachsende Leistung
- Geeignet für Streamprocessing:
  - Geringes Verhältnis IO-zu-Rechenlast
  - Datenparallelität (SIMD-Verarbeitung)
- *Single precision* (SP) wichtiger als *double precision* (DP)

Aufbau GPGPU



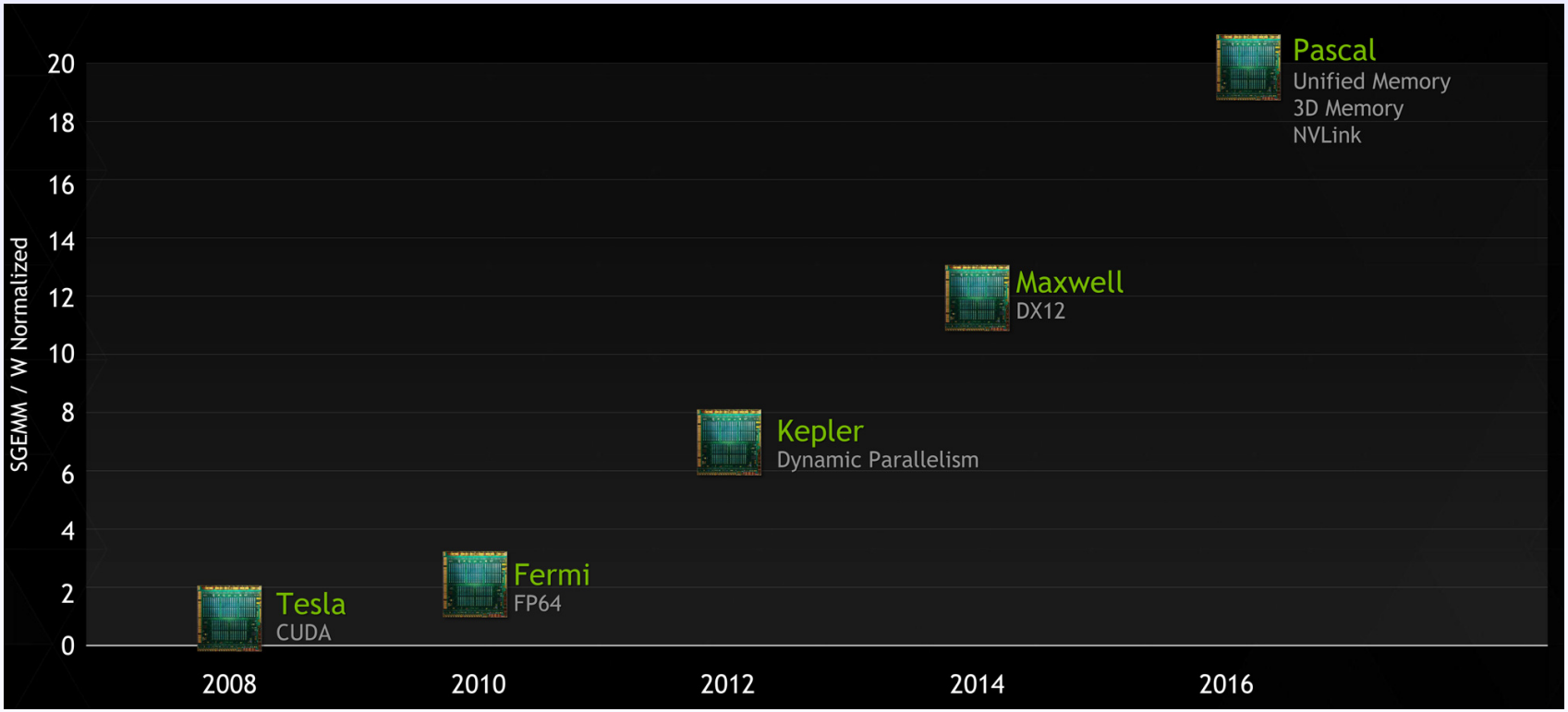
Eigenschaften von GPGPUs

- Viele einfache *Cores*, genannt Skalarprozessoren (SPs)
- Keine Sprungvorhersage etc.
- Gruppiert in Multi-Prozessoren (Vektorprozessoren)
- Probleme bei nicht einheitlichen Sprüngen
- Viele Register
- Großer, langsamer, globaler Speicher
  (Latenz: 400 – 800 Taktzyklen)
- Kleine, schnelle on-chip Shared-Memory-Blöcke

Grafikkartengenerationen

- 2006 G80 (Tesla) - Erster Entwurf einer Grafikkarte mit CUDA Unterstützung
- 2008 GT200 (Tesla) - einfache Double Precision Unterstützung
- 2010 GF100 (Fermi) - Grundlegende Neugestaltung der Multiprozessoren, Steigerung der DP Performance
- 2012 GK100 (Kepler) -Dynamic Parallelism, Massive Performance/Watt Steigerung
- 2014 GM200 (Maxwell) - Effizienzsteigerungen für 3D Grafikanwendungen
- 2016 GP? (Pascal) - 3D-Stacked DRAM, Unified Memory

Grafikgenerationen

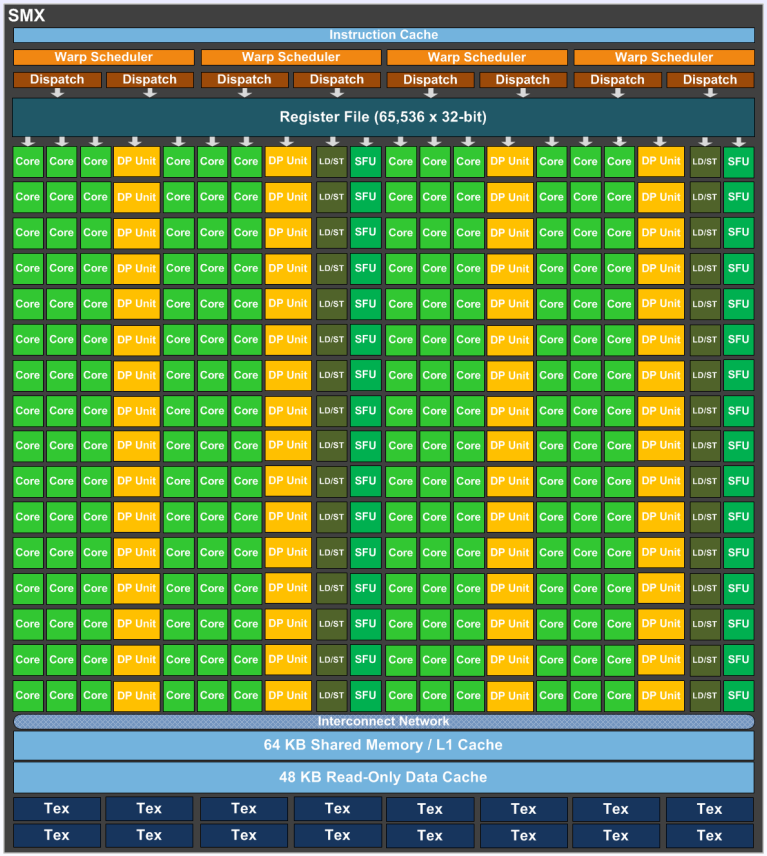


Nvidia GK110 - Zahlen

- 7.1 Mrd. Transistoren (28nm)
- Daten für Maximalausbau in Tesla K40 (≈6500 EUR)
  - 2880 SPs (15 Multiprozessoren x 192 SPs)
  - 12 GB GDDR5 RAM mit 384 Bit Speicherinterface (288 GB/s)
  - Performance bis zu 4290 GFlops (Single Precision)
  - Double Precision 1430 GFlops
- “Kleinster” GK110 in Geforce GTX 780 (≈500 EUR)
  - 2304 SPs (12 Multiprozessoren)
  - 3 GB GDDR5 RAM (288 GB/s)
  - 3976 GFlops (SP), 165 GFlops (DP)
- Architekturbeschreibung:
  `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`

Nvidia GK110 - Struktur

Nvidia GK110 - Struktur SMX



- 192 SP Cores
- 64 DP Units
- 32 Load/Store Units (LD/ST)
- 32 Special Function Units (SFU)
- 4 Warp Scheduler (je 2 Dispatch)
- 65536 32-Bit Register
- 64 kB Shared Memory / L1 Cache
- 48 kB Read-Only Cache

Programmierung

- Sehr viele, kurzlebige Threads
- Threads in Blöcken gruppiert
- Blöcke auf Multi-Prozessoren verteilt
- Standards:
  - CUDA (NVIDIA, Marktführer)
  - OpenCL (offener Standard, entsprechend zu OpenGL)
  - DirectCompute (Microsoft)
- CUDA Dokumentation:
  `http://docs.nvidia.com/cuda/index.html`

GPU Systeme am Lehrstuhl

- {gpu01,gpu02,gpu03}@inf-ra.uni-jena.de mit je
- Intel Core i7 4770
- 16 GB DDR3-RAM
- Nvidia Geforce GTX 780
  - 2304 SP Cores
  - 3 GB GDDR5 RAM
- Ubuntu 12.10 mit CUDA Toolkit 5.5

Login:

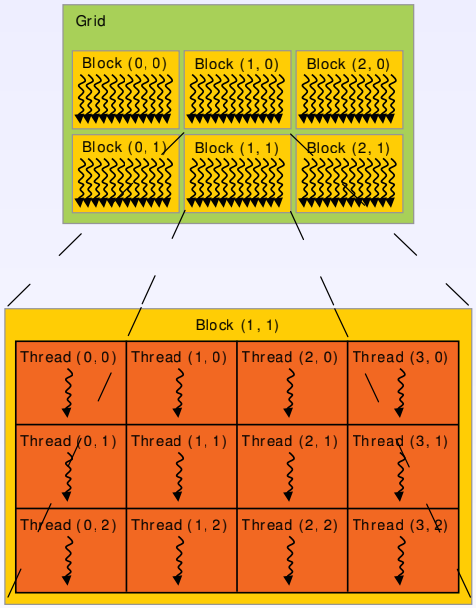
```
ssh gpu0X.inf-ra.uni-jena.de
./mount-frz.sh
cd frz
```

Outlook

CUDA - Compute Unified Device Architecture

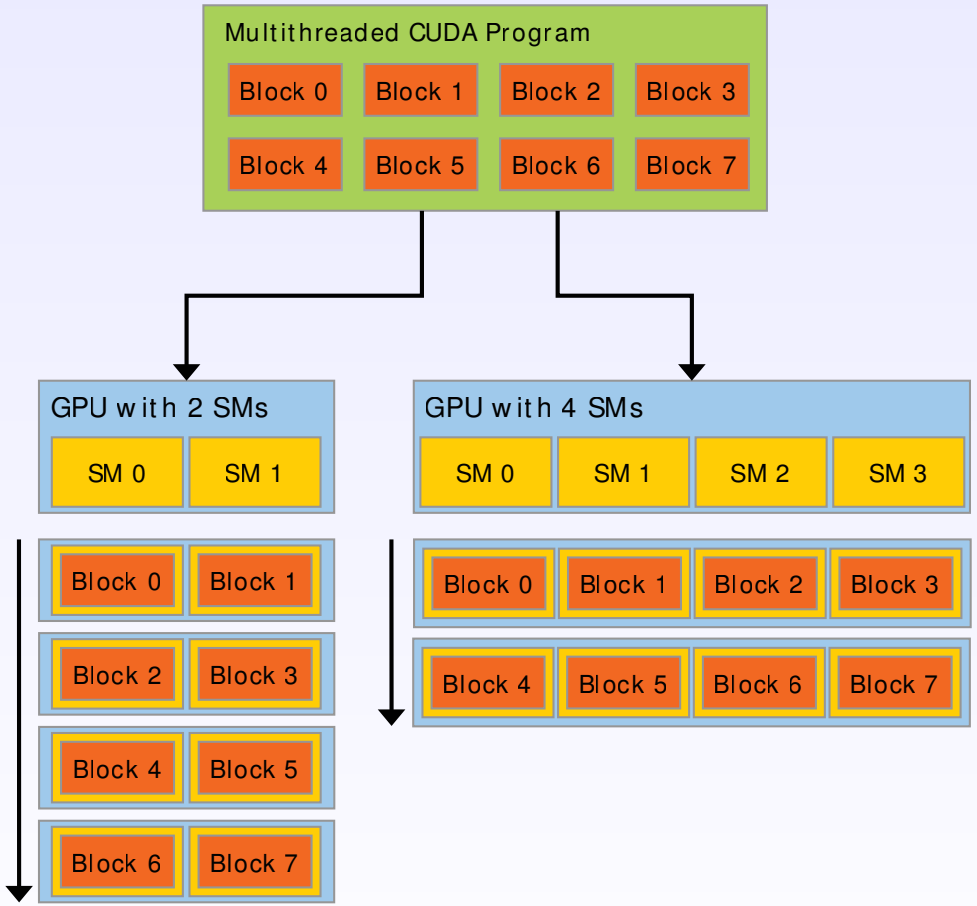
- Programmierung in C (CUDA for C)
- einzelne Funktionen laufen auf GPU in sog. Kernels (Function-Offloading)
- Compiler `nvcc` separiert Code, baut auf gcc auf
- Programm wird in Host-Code (Standard C) und Device-Code (CUDA) unterteilt
- Unterscheidung der Funktionen durch Keywords
  - `__host__` als CPU Funktionen (nur von Host aufrufbar)
  - `__device__` GPGPU Funktionen (nur von Device aufrufbar)
  - `__global__` definieren Kernel auf GPU (von Host-Code aus aufrufbar)

Programmiermodell



- 1-,2- oder 3-dimensionale Thread-Anordnung
- Unterscheidung zwischen Thread-Block und Grid
- Thread-Block:
  - Zusammenschluss von bis zu 1024 Threads
  - Threads innerhalb eines Thread-Blocks synchronisierbar
  - Shared Memory zum Datenaustausch
  - 32 Threads zu Warp zusammengefasst
- Grid:
  - Zusammenschluss der Thread-Blöcke
  - maximal $2^{32} - 1$ in einer Dimension

Abbildung auf Multiprozessoren



Beispiel

Einfaches Beispiel:
Speichere die Werte aus Array a inkrementiert um 1 in Array b!
Auf Host ohne CUDA:

```
int increment(int a)
{
    return a+1;
}

void host(int *a, int *b, int size)
{
    for (int i=0;i<size;i++)
        b[i]=increment(a[i]);
}
```

Beispiel

```
//increment wird auf host und device definiert
__host__ __device__ int increment(int a)
{
     return a+1;
}

__global__ void kernel(int *a, int *b, int size)
{
    int tid = threadIdx.x;      //lokaler Thread Index
    int bid = blockIdx.x;       //Index des Blockes
    int bdim= blockDim.x;       //Anzahl an Threads pro Block

    int i = tid+bid*bdim;       //Globale Adresse

    if (i<size)                 //Fehlerbehandlung
      b[i]=increment(a[i]);     //Increment
}
```

Hostcode

1. Allokation von globalem Speicher (`cudaMalloc()`)
2. Transfer System-RAM $\rightarrow$ GPU-RAM (`cudaMemcpy()`)
3. Kernel
   - Aufruf mit Konfiguration in Spitzen Klammern `<<<grid,threads_per_block,sm_size,stream>>>`
   - Nur `grid` und `threads_per_block` benötigt, rest optional
4. Transfer GPU-RAM $\rightarrow$ System-RAM (`cudaMemcpy()`)
5. Deallokation der Speicher (`cudaFree()`)

Beispiel

```
int main(int argc, char** argv)
{
    int size=1024;
    int *a_host, *b_host, *a_dev, *b_dev;;
    a_host = (int*)malloc(size*sizeof(int));
    b_host = (int*)malloc(size*sizeof(int));
    fillA(a_host,size);
    cudaMalloc((void**)&a_dev,size*sizeof(int));
    cudaMalloc((void**)&b_dev,size*sizeof(int));
    cudaMemcpy(a_dev,a_host,size*sizeof(int),cudaMemcpyHostToDevice);
    dim3 threads(256);
    dim3 grid(size/threads.x);
    kernel<<<grid,threads>>>(a_dev,b_dev,size);
    cudaMemcpy(b_host,b_dev,size*sizeof(int),cudaMemcpyDeviceToHost);
    checkResult(a_host,b_host,size);
    cudaFree(a_dev);
    cudaFree(b_dev);
    free(a_host);
    free(b_host);
    return 0;
}
```

Beispiel

Code dafür liegt im System unter `01_beispiel`
Übersetzen mit

```
nvcc -o beispiel beispiel.cu -arch=sm_35
```

- Endung `*.cu` für C-Code mit CUDA (kann auch normalen Hostcode beinhalten)
- `-arch=sm_35`: GPU-Code soll für Device für Compute Capability 3.5 und neuer erzeugt werden

Compute Capability gibt Fähigkeiten der GPU an

CUDA - Compute Capability

Beispiele für Compute Capability (Shader Model):

Compute Capability	1.0 G80	1.3 GT200	2.0 GF100	3.0 GK104	3.5 GK110
Geforce GTX	8800	280	480	680	780
SP Cores	128	240	480	1536	2304
SP/MP	8/16	8/30	32/15	192/8	192/12
Release	2006	2008	2010	2012	2013
double	-	o[1]	x	x	x
atomics	-	o[2]	x	x	x
printf	-	-	x	x	x
Dynamic Par.	-	-	-	-	x
L1-Cache	Tex	Tex	x	x	x
Reg/Thread	128	128	63	63	255

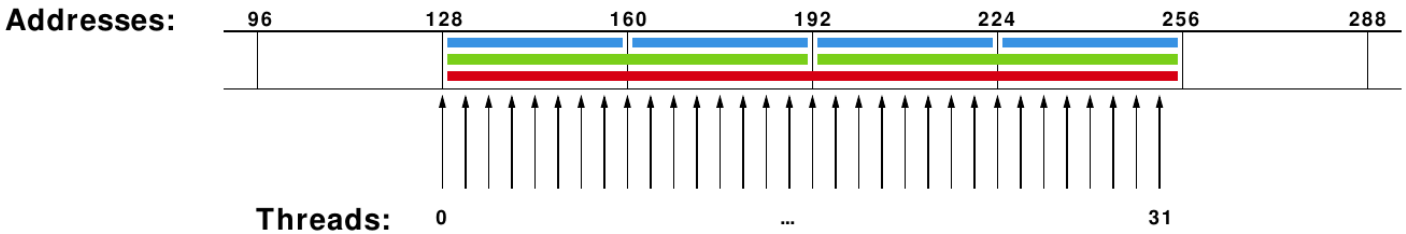
[1] Nicht IEEE 754

[2] Nur int32

Globale-Speicherzugriffe I

Zugriff auf aufeinander folgende Speicheradressen ideal

Aligned and sequential

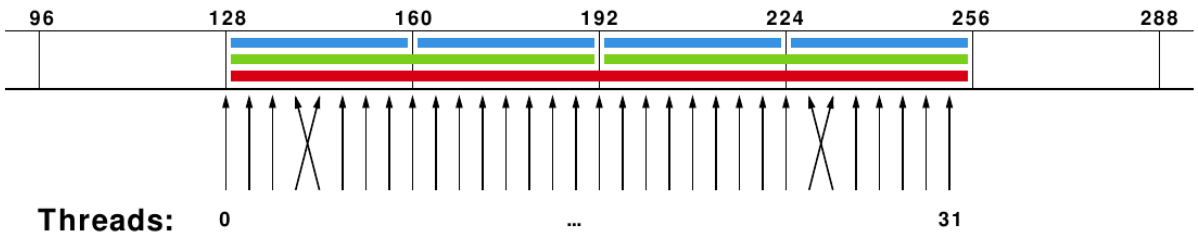


Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	1x 64B at 128 1x 64B at 192	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Globale-Speicherzugriffe II

Neuere GPUs haben kein Problem mit überkreuzenden Zugriffen

Aligned and non- sequential

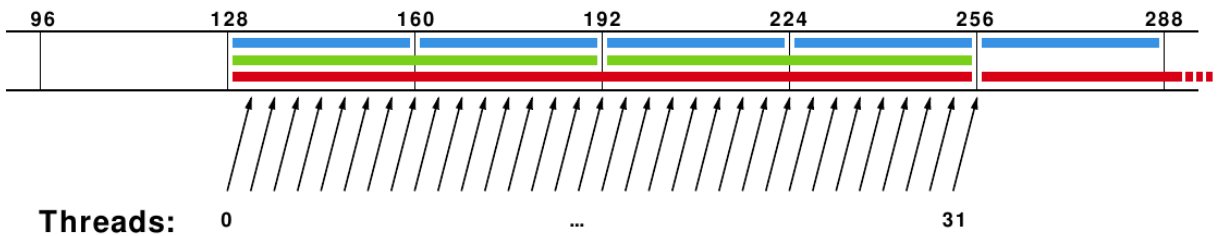


Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	8x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Globale-Speicherzugriffe III

Aber: Verschobene Zugriffe sind immer schlecht

Mis- aligned and sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	7x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224 1x 32B at 256	1x 128B at 128 1x 64B at 192 1x 32B at 256	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

Zeitmessung auf der GPU

- Am einfachsten über Events
- Datentyp `cudaEvent_t`
- Erzeugen/Vernichten: `cudaEventCreate()`/`cudaEventDestroy()`
- `cudaEventRecord()` für Event auf GPU
- `cudaEventSynchronize()` synchronisiert CPU mit GPU auf angegebenen Event (für Asynchrone Abarbeitung)
- `cudaEventElapsedTime()` Zeit zwischen 2 Events in ms
- Auch zur Synchronisierung zwischen verschiedenen Host-Threads/GPUs gedacht

Zeitmessung auf der GPU - Beispiel

```
int main(int argc, char** argv)
{
    ...
    float time;
    cudaEvent_t start,end;
    cudaEventCreate(&start);
    cudaEventCreate(&end);
    cudaEventRecord(start);
    kernel<<<grid,threads>>>(a_dev,b_dev,size);
    cudaEventRecord(end);
    cudaEventSynchronize(end);
    cudaEventElapsedTime(&time,start,end);
    ...
}
```

Neue Aufgaben - Matrix-Vektor-Produkt

1. Setzen Sie die Matrix-Vektor Multiplikation $Ax = y$ mit dem Ihnen zur Verfügung stehenden Wissen mittels CUDA um und messen sie die Zeiten auf der GPU für eine Matrix $A \in R^{n \times n}$ mit $n = 1024 \cdot i, i \in \{1, 4, 8, 16\}$.
   Jeder Thread soll dabei eine komplette Zeile bearbeiten (1 dimensionale Grid, Threads und Arrays).
2. Nutzen Sie die gleiche Matrix A für die Berechnung $A^T x = y$ indem Sie die Indizes der Matrix A vertauschen. Welchen Einfluss hat dies auf die Berechnungszeit?
3. Vergleichen Sie die Zeiten mit einer reinen CPU Implementierung (Seriell).

Nutzen Sie die Templates in `02_matvec`.

Fehlerbehandlung

- Jede CUDA Host Funktion (`cudaMemcpy`, `cudaMalloc`, ...) hat als Rückgabewert `cudaError_t`
- Damit kann auf Fehler reagiert werden (Fehler bei `cudaError_t != 0`)
- `cudaGetLastError()` um Fehler von Kernel Aufrufen zu überprüfen
- `cudaGetErrorString()` um aussagekräftigere Fehlermeldung auszugeben

```
kernel<<<grid,threads>>>(a_dev,b_dev,size);
cudaError_t err=cudaGetLastError();
printf("CUDA status: %s\n",cudaGetErrorString(err));
```

Shared Memory Zugriffe

- Shared Memory bestehen aus je 32 Bänken
- Aufeinanderfolgende 32-Bit Wörter in aufeinanderfolgenden Bänken
- Bandbreite: Je Bank 32 Bits pro 2 Takte
- Bankkonflikte treten nur auf, wenn zwei oder mehr Threads unterschiedliche 32 Bit Wörter einer Bank lesen/schreiben

Outlook

Weiterführende Details I

- Atomic-Funktionen für Thread-Exklusiven Zugriff auf Speicherbereiche:
  - `atomicMin/atomicMax`
  - `atomicAdd/atomicSub`
  - `atomicInc/atomicDec`
  - Bis Compute-Capability 1.3 nur auf Integer Daten definiert
  - 2.0 unterstützt auch float $\rightarrow$ Assoziativität?
- L1-Cache/Shared Memory konfigurierbar:
  - 64 kByte pro SM verfügbar
  - `cudaFuncSetCacheConfig(function, cudaFuncCachePreferL1|cudaFuncCachePreferShared)`
  - L1 48k/Shared 16k | L1 16k/Shared 48k | L1 32k/Shared 32k

Weiterführende Details II

- Host-Funktionen von Kernel ausführbar
  - Seit Compute-Capability 2.0 unterstützt
  - `printf()`
    - Einfacheres Debugging möglich
    - Achtung: Ausgabe erfolgt PRO Thread, Puffer sehr schnell voll
  - `malloc()/free()`
    - Maximale Heapsize vorher angeben (Standard 8MB) `cudaThreadSetLimit(cudaLimitMallocHeapSize, size)`
    - Jeder Thread kann dann Speicher vom Heap allokieren
    - Bleibt über gesamte Ausführung (auch über Kernel-Calls) bestehen und MUSS mit `free()` freigegeben werden

Weiterführende Details II

- Dynamic Parallelism
  - Seit Compute-Capability 3.5 unterstützt
  - Rekursives aufrufen von Kernels