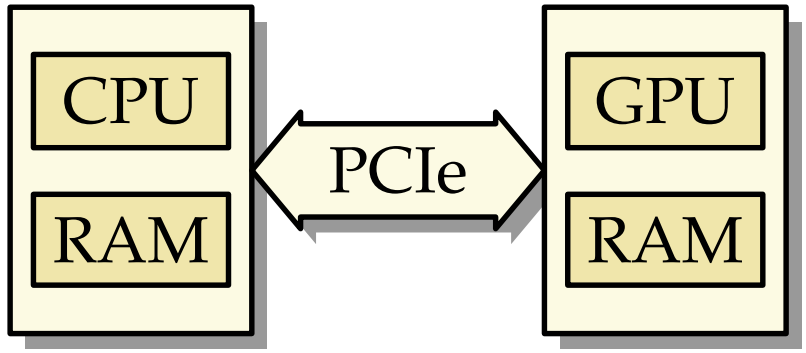
High Performance Computing I
WS 2015/2016

Andreas F. Borchert und Michael Lehn
Universität Ulm

8. Februar 2016

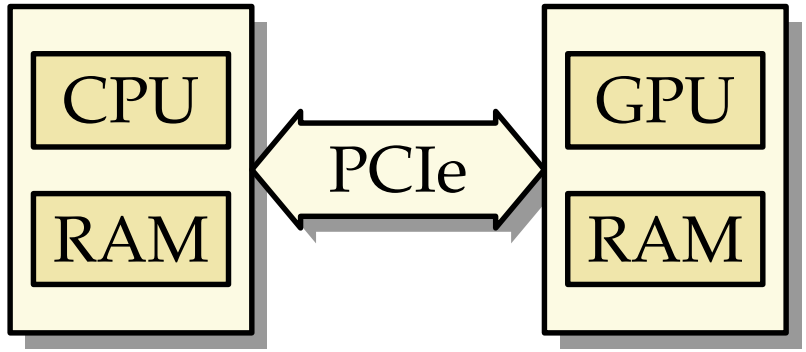
- Schon sehr früh gab es diverse Grafik-Beschleuniger, die der normalen CPU Arbeit abnahmen.
- Die im März 2001 von Nvidia eingeführte GeForce 3 Series führte programmierbares Shading ein.
- Im August 2002 folgte die Radeon R300 von ATI, die die Fähigkeiten der GeForce 3 deutlich erweiterte um mathematische Funktionen und Schleifen.
- Zunehmend werden die GPUs zu GPGPUs (*general purpose GPUs*).
- Zur generellen Nutzung wurden mehrere Sprachen und Schnittstellen entwickelt: OpenCL (Open Computing Language), DirectCompute (von Microsoft) und CUDA (Compute Unified Device Architecture, von Nvidia). Wir beschäftigen uns im Rahmen dieser Vorlesung mit CUDA, da es recht populär ist und bei uns auch zur Verfügung steht.

Beziehung zwischen CPU und GPU



- Überwiegend stehen GPUs als PCI-Express-Steckkarten zur Verfügung.
- Sie leben und arbeiten getrennt von der CPU und ihrem Speicher. Die Kommunikation erfolgt normalerweise über die PCI-Express-Verbindung. Neuere GPUs haben auch alternative Verbindungen mit höheren Kommunikationsraten (beispielsweise Nvidia NVLink).
- Meistens haben sie völlig eigenständige und spezialisierte Prozessorarchitekturen. Eine prominente Ausnahme ist die Larrabee-Mikroarchitektur von Intel, die sich an der x86-Architektur ausrichtete und der später die Xeon-Phi-Architektur folgte, die keine GPU mehr ist und nur noch dem High-Performance-Computing dient.

Beziehung zwischen CPU und GPU



- Eine entsprechende Anwendung besteht normalerweise aus zwei Programmen (eines für die CPU und eines für die GPU).
- Das Programm für die GPU muss in die GPU heruntergeladen werden; anschließend sind die Daten über die PCI-Express-Verbindung (oder den alternativen Kommunikationspfad) auszutauschen.
- Die Kommunikation kann (mit Hilfe der zur Verfügung stehenden Bibliothek) sowohl synchron als auch asynchron erfolgen.

- Für die GPUs stehen höhere Programmiersprachen zur Verfügung, typischerweise Varianten, die auf C bzw. C++ basieren.
- Hierbei kommen Spracherweiterungen für GPUs hinzu. Jedoch wird C bzw. C++ nicht umfassend unterstützt. Insbesondere stehen außer wenigen mathematischen Funktionen keine Standard-Bibliotheken zur Verfügung.
- Nach dem Modell von CUDA sind die Programmtexte für die CPU und die GPU in der gleichen Quelle vermischt. Diese werden auseinandersortiert und dann getrennt voneinander übersetzt. Der entstehende GPU-Code wird dann als Byte-Array in den CPU-Code eingefügt. Zur Laufzeit wird dann nur noch der fertig übersetzte GPU-Code zur GPU geladen.
- Neuere CUDA-Versionen unterstützen auch Übersetzungen zur Laufzeit.
- Beim Modell von OpenCL sind die Programmtexte getrennt, wobei der für die GPU bestimmte Programmtext erst zur Laufzeit übersetzt und mit Hilfe von OpenCL-Bibliotheksfunktionen in die GPU geladen wird.

CUDA ist ein von Nvidia für Linux, MacOS und Windows kostenfrei zur Verfügung gestelltes Paket (jedoch nicht *open source*), das folgende Komponenten umfasst:

- einen Gerätetreiber,
- eine Spracherweiterung von C bzw. C++ (CUDA C bzw. CUDA C++), die es ermöglicht, in einem Programmtext die Teile für die CPU und die GPU zu vereinen,
- einen Übersetzer *nvcc* (zu finden im Verzeichnis */usr/local/cuda-7.0/bin* auf Hochwanner und in */usr/local/cuda/bin* auf Olympia), der CUDA C bzw. CUDA C++ unterstützt,
- eine zugehörige Laufzeitbibliothek (*libcudart.so* in */usr/local/cuda-7.0/lib* auf Hochwanner und in */usr/local/cuda/lib* auf Olympia) und
- darauf aufbauende Bibliotheken (einschließlich BLAS und FFT).

URL: `https://developer.nvidia.com/cuda-downloads`

hpc/cuda/check.h

```
inline void check_cuda_error(const char* cudaop, const char* source,
       unsigned int line, cudaError_t error) {
   if (error != cudaSuccess) {
      fprintf(stderr, "%s at %s:%u failed: %s\n",
         cudaop, source, line, cudaGetErrorString(error)); exit(1);
   }
}

#define CHECK_CUDA(opname, ...) \
   hpc::cuda::check_cuda_error(#opname, __FILE__, __LINE__,
   opname(__VA_ARGS__))
```

properties.cu

```
#include <hpc/cuda/check.h>

int main() {
   int device; CHECK_CUDA(cudaGetDevice, &device);
   // ...
}
```

- Bei CUDA-Programmen steht die CUDA-Laufzeit-Bibliothek zur Verfügung. Die Fehler-Codes aller CUDA-Bibliotheksaufrufe sollten jeweils überprüft werden.

properties.cu

```
int device_count; CHECK_CUDA(cudaGetDeviceCount, &device_count);
if (device_count > 1) {
   std::printf("device %d selected out of %d devices:\n",
      device, device_count);
} else {
   std::printf("one device found:\n");
}
struct cudaDeviceProp p;
CHECK_CUDA(cudaGetDeviceProperties, &p, device);
// ...
```

- *cudaGetDeviceProperties* füllt eine umfangreiche Datenstruktur mit den Eigenschaften einer der zur Verfügung stehenden GPUs.
- Es gibt eine Vielzahl von Nvidia-Karten mit sehr unterschiedlichen Fähigkeiten und Ausstattungen, so dass bei portablen Anwendungen diese u.U. zuerst überprüft werden müssen.

```
hochwanner$ nvcc -o properties -I/home/numerik/pub/hpc/session24 \
> --gpu-architecture compute_20 properties.cu
hochwanner$ properties | sed 18q
one device found:
name: Quadro 600
pci id: 0000:0001:00
compute mode: default
compute capability: 2.1
multi processor count: 2
clock rate: 1280000 khz
total global memory: 1072889856
total constant memory: 65536
total shared memory per block: 49152
registers per block: 32768
L2 cache size: 131072
supporting caching of globals in L1 cache: yes
supporting caching of locals in L1 cache: yes
warp size: 32
max threads per block: 1024
max threads dim per block: 1024 x 1024 x 64
max grid dim: 65535 x 65535 x 65535
hochwanner$
```

```
hochwanner$ nvcc -o properties -I/home/numerik/pub/hpc/session24 \
> --gpu-architecture compute_20 properties.cu
```

- Übersetzt wird mit *nvcc*.
- Hierbei werden die für die GPU vorgesehenen Teile übersetzt und in Byte-Arrays verwandelt, die dann zusammen mit den für die CPU bestimmten Programmteilen dem *gcc* zur Übersetzung gegeben werden.
- Die Option „–gpu-architecture compute_20“ ermöglicht die Nutzung wichtiger später hinzu gekommener Erweiterungen wie z.B. die Unterstützung des Datentyps **double** auf der GPU.
- Wenn die Option „–code sm_20“ hinzukommt, wird beim Übersetzen auch der Code für die CPU erzeugt. Ansonsten geschieht dies bei neueren CUDA-Versionen erst zur Laufzeit im JIT-Verfahren.

simple.cu

```
template<typename Index, typename Alpha, typename TX, typename TY>
__global__ void axpy(Index n, Alpha alpha,
      const TX* x, Index incX, TY* y, Index incY) {
   for (Index i = 0; i < n; ++i) {
      y[i*incY] += alpha * x[i*incX];
   }
}
```

- Funktionen (oder Methoden), die für die GPU bestimmt sind und von der CPU aus aufrufbar sind, werden Kernel-Funktionen genannt und mit dem CUDA-Schlüsselwort **__global__** gekennzeichnet.
- Es darf sich dabei auch um Template-Funktionen handeln.
- Die Template-Funktion *axpy* in diesem Beispiel berechnet $\vec{y} \leftarrow \vec{y} + \alpha\vec{x}$.
- Alle Zeiger verweisen hier auf GPU-Speicher.

simple.cu

```
/* execute kernel function on GPU */
axpy<<<1, 1>>>(N, 2.0, cuda_a, cuda_b);
```

- Eine Kernel-Funktion kann direkt von dem auf der CPU ausgeführten Programmtext aufgerufen werden.
- Zwischen dem Funktionsnamen und den Parametern wird in „<<<...>>>“ die Kernel-Konfiguration angegeben. Diese ist zwingend notwendig. Die einzelnen Konfigurationsparameter werden später erläutert.
- Elementare Datentypen können direkt übergeben werden (hier N und der Wert 2.0), Zeiger müssen aber bereits auf GPU-Speicher zeigen, d.h. dass ggf. Daten zuvor vom CPU-Speicher zum GPU-Speicher zu transferieren sind.

simple.cu

```
double a[N]; double b[N];
for (unsigned int i = 0; i < N; ++i) {
   a[i] = i; b[i] = i * i;
}

/* transfer vectors to GPU memory */
double* cuda_a;
CHECK_CUDA(cudaMalloc, (void**)&cuda_a, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, cuda_a, a, N * sizeof(double),
   cudaMemcpyHostToDevice);
double* cuda_b;
CHECK_CUDA(cudaMalloc, (void**)&cuda_b, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, cuda_b, b, N * sizeof(double),
   cudaMemcpyHostToDevice);
```

- Mit der CUDA-Bibliotheksfunktion *cudaMalloc* kann Speicher bei der GPU belegt werden.
- Es kann davon ausgegangen werden, dass alle Datentypen auf der CPU und der GPU in gleicher Weise repräsentiert werden. Anders als bei MPI kann die Übertragung ohne Konvertierungen erfolgen.

Kopieren zwischen CPU und GPU

simple.cu

```
double a[N]; double b[N];
for (unsigned int i = 0; i < N; ++i) {
   a[i] = i; b[i] = i * i;
}

/* transfer vectors to GPU memory */
double* cuda_a;
CHECK_CUDA(cudaMalloc, (void**)&cuda_a, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, cuda_a, a, N * sizeof(double),
   cudaMemcpyHostToDevice);
double* cuda_b;
CHECK_CUDA(cudaMalloc, (void**)&cuda_b, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, cuda_b, b, N * sizeof(double),
   cudaMemcpyHostToDevice);
```

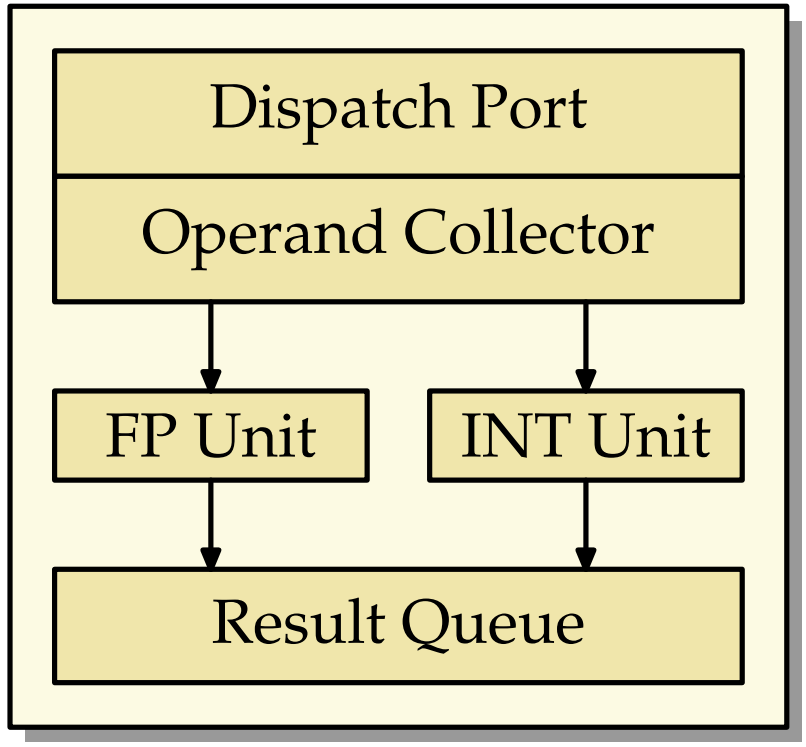
- Mit *cudaMemcpy* kann von CPU- in GPU-Speicher oder umgekehrt kopiert werden. Zuerst kommt (wie bei *memcpy*) das Ziel, dann die Quelle, dann die Zahl der Bytes. Zuletzt wird die Richtung angegeben.
- Ohne den letzten Parameter weiß *cudaMemcpy* nicht, um was für Zeiger es sich handelt.

simple.cu

```
/* transfer result vector from GPU to host memory */
CHECK_CUDA(cudaMemcpy, b, cuda_b, N * sizeof(double),
   cudaMemcpyDeviceToHost);
/* free space allocated at GPU memory */
CHECK_CUDA(cudaFree, cuda_a);
CHECK_CUDA(cudaFree, cuda_b);
```

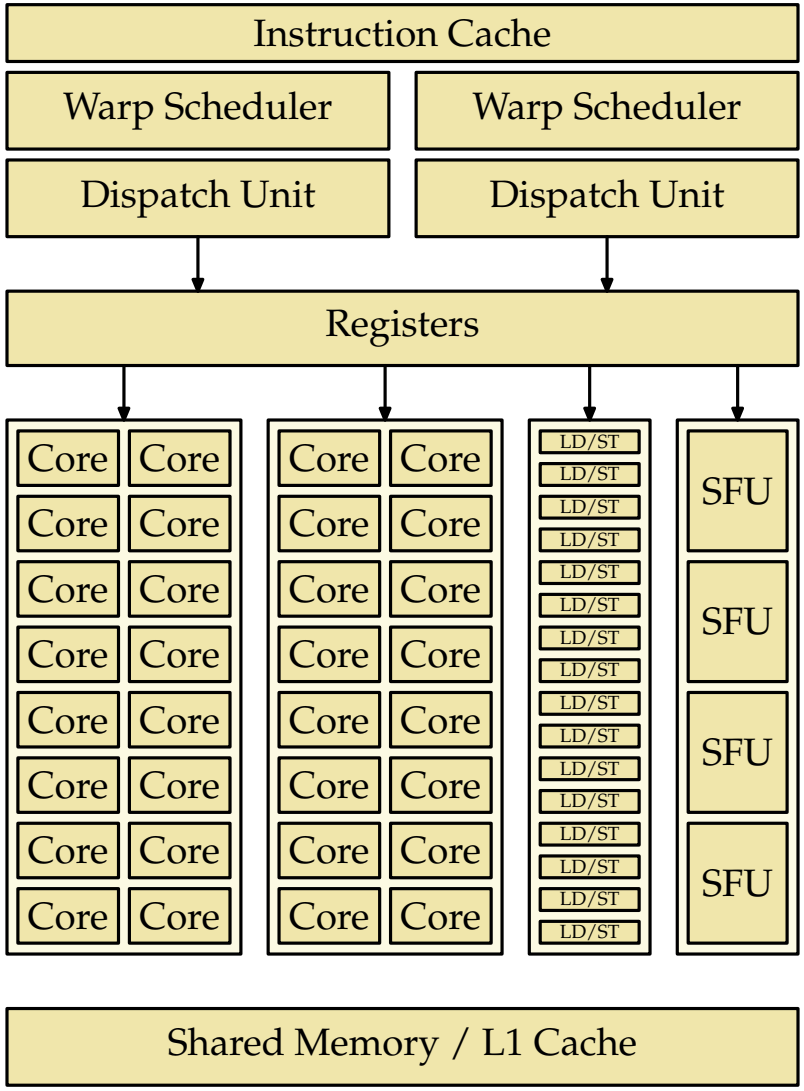
- Nach dem Aufruf der Kernel-Funktion kann das Ergebnis zurückkopiert werden.
- Kernel-Funktionen laufen asynchron, d.h. die weitere Ausführung des Programms auf der CPU läuft parallel zu der Verarbeitung auf der GPU. Sobald die Ergebnisse jedoch zurückkopiert werden, findet implizit eine Synchronisierung statt, d.h. es wird auf die Beendigung der Kernel-Funktion gewartet.
- Wenn der Kernel nicht gestartet werden konnte oder es zu Problemen während der Ausführung kam, wird dies über die Fehler-Codes bei der folgenden synchronisierenden CUDA-Operation mitgeteilt.
- Anschließend sollte der GPU-Speicher freigegeben werden, wenn er nicht mehr benötigt wird.

- Das erste Beispiel zeigt den typischen Ablauf vieler Anwendungen:
  - Speicher auf der GPU belegen
  - Daten zur GPU transferieren
  - Kernel-Funktion aufrufen
  - Ergebnisse zurücktransferieren
  - GPU-Speicher freigeben
- Eine echte Parallelisierung hat das Beispiel nicht gebracht, da die CPU auf die GPU wartete und auf der GPU nur ein einziger GPU-Kern aktiv war...

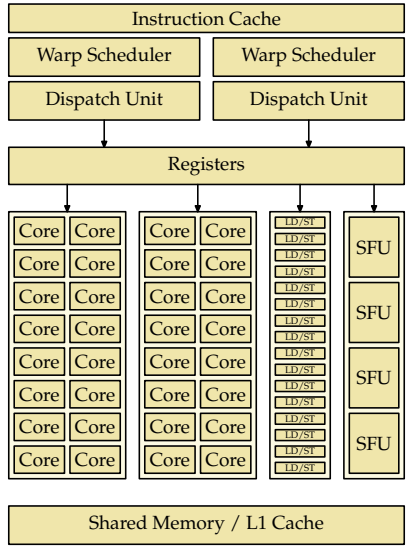


- Die elementaren Recheneinheiten einer GPU bestehen im wesentlichen nur aus zwei Komponenten, jeweils eine für arithmetische Operationen mit Gleitkommazahlen und eine mit ganzen Zahlen.
- Mehr Teile hat ein GPU-Kern nicht. Die Instruktion und die Daten werden angeliefert (*Dispatch Port* und *Operand Collector*), und das Resultat wird bei der *Result Queue* abgeliefert.

Multiprozessor einer GPU

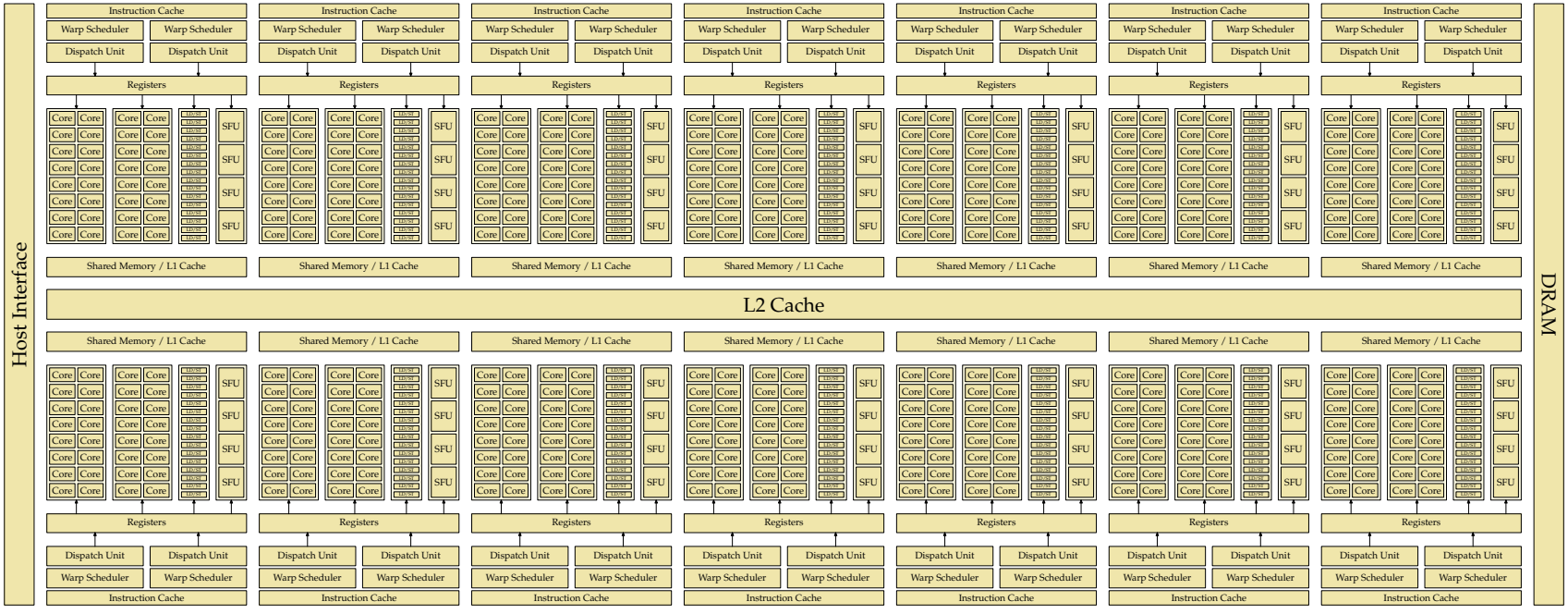


Multiprozessor einer GPU



- Je nach Ausführung der GPU werden zahlreiche Kerne und weitere Komponenten zu einem Multiprozessor zusammengefasst.
- Auf Olympia hat ein Multiprozessor 32, auf Hochwanner 48 Kerne.
- Hinzu kommen Einheiten zum parallelisierten Laden und Speichern (*LD/ST*) und Einheiten für spezielle Operationen wie beispielsweise *sin* oder *sqrt* (*SFU*).

GPU mit mehreren Multiprozessoren



- Je nach Ausführung der GPU werden mehrere Multiprozessoren zusammengefasst.
- Olympia bietet 14 Multiprozessoren mit insgesamt 448 Kernen an. Bei Hochwanner sind es nur 2 Multiprozessoren mit insgesamt 96 Kernen.

- Die Kerne operieren nicht unabhängig voneinander.
- Im Normalfall werden 32 Kerne zu einem Warp zusammengefasst. (Unterstützt werden auch halbe Warps mit 16 Kernen.)
- Alle Kerne eines Warps führen synchron die gleiche Instruktion aus – auf unterschiedlichen Daten (SIMD-Architektur: Array-Prozessor).
- Dabei werden jeweils zunächst die zu ladenden Daten parallel organisiert (durch die *LD/ST*-Einheiten) und dann über die Register den einzelnen Kernen zur Verfügung gestellt.

- Der Instruktionssatz der Nvidia-GPUs ist proprietär und bis heute wurde abgesehen von einer kleinen Übersicht von Nvidia kein umfassendes öffentliches Handbuch dazu herausgegeben.
- Mit Hilfe des Werkzeugs *cuobjdump* lässt sich der Code disassemblieren und ansehen.
- Die Instruktionen haben entweder einen Umfang von 32 oder 64 Bits. 64-Bit-Instruktionen sind auf 64-Bit-Kanten.
- Arithmetische Instruktionen haben bis zu drei Operanden und ein Ziel, bei dem das Ergebnis abgelegt wird. Beispiel ist etwa eine Instruktion, die in einfacher Genauigkeit $d = a * b + c$ berechnet (FMAD).

Wie können bedingte Sprünge umgesetzt werden, wenn ein Warp auf eine if-Anweisung stößt und die einzelnen Threads des Warps unterschiedlich weitermachen wollen? (Zur Erinnerung: Alle Threads eines Warps führen immer die gleiche Instruktion aus.)

- Es stehen zwei Stacks zur Verfügung:
- Ein Stack mit Masken, bestehend aus 32 Bits, die festlegen, welche der 32 Threads die aktuellen Instruktionen ausführen.
- Ferner gibt es noch einen Stack mit Zieladressen.
- Bei einer bedingten Verzweigung legt jeder der Threads in der Maske fest, ob die folgenden Instruktionen ihn betreffen oder nicht. Diese Maske wird auf den Stack der Masken befördert.
- Die Zieladresse des Sprungs wird auf den Stack der Zieladressen befördert.
- Wenn die Zieladresse erreicht wird, wird auf beiden Stacks das oberste Element jeweils entfernt.

- Zunächst wurden nur ganzzahlige Datentypen (32 Bit) und Gleitkommazahlen (**float**) unterstützt.
- Erst ab Level 1.3 kam die Unterstützung von **double** hinzu. Die GeForce GTX 470 auf Olympia unterstützt Level 2.0, die Quadro 600 auf Hochwanner unterstützt Level 2.1. (Beim Übersetzen mit *nvcc* sollte immer die Option „–gpu-architecture compute_20“ angegeben werden, da ohne die Option **double** nicht zur Verfügung steht.)
- Ferner werden Zeiger unterstützt.
- Zugriffe sind (auch per Zeiger ab Level 2.0) möglich auf den gemeinsamen Speicher der GPU (*global memory*), auf den gemeinsamen Speicher eines Blocks (*shared memory*) und auf lokalen Speicher (Register).

- Ein Block ist eine Abstraktion, die mehrere Warps zusammenfasst.
- Bei CUDA-Programmen werden Blöcke konfiguriert, die dann durch den jeweiligen *Warp Scheduler* auf einzelne Warps aufgeteilt werden, die sukzessive zur Ausführung kommen.
- Ein Block läuft immer nur auf einem Multiprozessor.
- Threads eines Blockes können sich untereinander synchronisieren und über den gemeinsamen Speicher kommunizieren.
- Entsprechend ist der gemeinsame Speicherbereich immer mit einem Block assoziiert.
- Ein Block kann (bei uns auf Olympia und Hochwanner) bis zu 32 Warps bzw. 1024 Threads umfassen.

vector.cu

```
template<typename Index, typename Alpha, typename TX, typename TY>
__global__ void axpy(Index n, Alpha alpha,
      const TX* x, Index incX, TY* y, Index incY) {
   Index index = threadIdx.x + blockIdx.x * blockDim.x;
   if (index < n) {
      y[index*incY] += alpha * x[index*incX];
   }
}
```

- Die **for**-Schleife ist entfallen. Stattdessen führt die Kernel-Funktion jetzt nur noch eine einzige Addition durch.
- Mit *threadIdx.x* erfahren wir hier, der wievielte Thread (beginnend ab 0) unseres Blocks wir sind.
- Die Kernel-Funktion wird dann für jedes Element aufgerufen, wobei dies im Rahmen der Möglichkeiten parallelisiert wird.

vector.cu

```
/* execute kernel function on GPU */
axpy<<<1, N>>>(N, 2.0, cuda_a, 1, cuda_b, 1);
```

- Beim Aufruf der Kernel-Funktion wurde jetzt die Konfiguration verändert.
- $<<<1, N>>>$ bedeutet jetzt, dass ein Block mit N Threads gestartet wird.
- Entsprechend starten die einzelnen Threads mit Werten von 0 bis $N - 1$ für *threadIdx.x*.
- Da die Zahl der Threads pro Block beschränkt ist, kann N hier nicht beliebig groß werden.

Übersetzungsprozess: Generierung von PTX 28

vector.ptx

```
        mov.u32         %r5, %tid.x;
        mov.u32         %r6, %ntid.x;
        mov.u32         %r7, %ctaid.x;
        mad.lo.s32      %r1, %r6, %r7, %r5;
        setp.ge.u32     %p1, %r1, %r4;
        @%p1 bra        BB0_2;
        cvta.to.global.u64      %rd3, %rd1;
        mul.lo.s32      %r8, %r1, %r2;
        mul.wide.u32    %rd4, %r8, 8;
        add.s64         %rd5, %rd3, %rd4;
        ld.global.f64   %fd2, [%rd5];
        mul.lo.s32      %r9, %r1, %r3;
        cvta.to.global.u64      %rd6, %rd2;
        mul.wide.u32    %rd7, %r9, 8;
        add.s64         %rd8, %rd6, %rd7;
        ld.global.f64   %fd3, [%rd8];
        fma.rn.f64      %fd4, %fd2, %fd1, %fd3;
        st.global.f64   [%rd8], %fd4;
BB0_2:
        ret;
```

- PTX steht für *Parallel Thread Execution* und ist eine Assembler-Sprache für einen virtuellen GPU-Prozessor. Dies ist die erste Zielsprache des Übersetzers für den für die GPU bestimmten Teil.

- PTX steht für *Parallel Thread Execution* und ist eine Assembler-Sprache für einen virtuellen GPU-Prozessor. Dies ist die erste Zielsprache des Übersetzers für den für die GPU bestimmten Teil.
- Der PTX-Instruktionssatz ist öffentlich:
  `http:www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf`
- PTX wurde entwickelt, um eine portable vom der jeweiligen Grafikkarte unabhängige virtuelle Maschine zu haben, die ohne größeren Aufwand effizient für die jeweiligen GPUs weiter übersetzt werden kann.
- PTX lässt sich mit Hilfe des folgenden Kommandos erzeugen:
  ```
  nvcc -o vector.ptx -ptx -gpu-architecture compute_20
  vector.cu
  ```

Übersetzungsprozess: Generierung von CUBIN 30

vector.disas

```
/*0000*/         MOV R1, c[0x1][0x100];
/*0008*/         S2R R2, SR_TID.X;
/*0010*/         S2R R0, SR_CTAID.X;
/*0018*/         IMAD R0, R0, c[0x0][0x8], R2;
/*0020*/         ISETP.GE.U32.AND P0, PT, R0, c[0x0][0x20], PT;
/*0028*/     @P0 BRA.U 0x98;
/*0030*/    @!P0 IMUL R2, R0, c[0x0][0x38];
/*0038*/    @!P0 MOV32I R3, 0x8;
/*0040*/    @!P0 IMUL R0, R0, c[0x0][0x48];
/*0048*/    @!P0 IMAD.U32.U32 R8.CC, R2, R3, c[0x0][0x30];
/*0050*/    @!P0 IMAD.U32.U32.HI.X R9, R2, R3, c[0x0][0x34];
/*0058*/    @!P0 IMAD.U32.U32 R2.CC, R0, R3, c[0x0][0x40];
/*0060*/    @!P0 LD.E.64 R6, [R8];
/*0068*/    @!P0 IMAD.U32.U32.HI.X R3, R0, R3, c[0x0][0x44];
/*0070*/    @!P0 MOV R10, c[0x0][0x28];
/*0078*/    @!P0 LD.E.64 R4, [R2];
/*0080*/    @!P0 MOV R11, c[0x0][0x2c];
/*0088*/    @!P0 DFMA R4, R6, R10, R4;
/*0090*/    @!P0 ST.E.64 [R2], R4;
/*0098*/         EXIT;
```

- *ptxas* übersetzt den PTX-Text in Maschinen-Code:
  `ptxas -output-file vector.cubin -gpu-name sm_21 vector.ptx`

- In neueren Versionen stellt Nvidia mit *cuobjdump* ein Werkzeug zur Verfügung, der den CUBIN-Code wieder disassembliert:
  `cuobjdump -dump-sass vector.cubin >vector.disas`
- Bei uns (d.h. Hochwanner und Olympia) ist in der zugehörigen Dokumentation die Übersicht zu Fermi relevant.

bigvector.cu

```
/* execute kernel function on GPU */
unsigned int threads_per_block = 128;
unsigned int num_blocks = (N + threads_per_block - 1) /
   threads_per_block;
axpy<<<num_blocks, threads_per_block>>>(N, 2.0,
   cuda_a, 1, cuda_b, 1);
```

- Wenn die Zahl der Elemente größer ist als die maximale Zahl der Threads pro Block, wird es notwendig, die Aufgabe auf mehrere Blöcke aufzusplitten.
- Bei uns liegt die maximale Zahl der Threads pro Block bei 1.024.
- Wenn nicht das Maximum ausgereizt wird, stehen jeweils mehr Register und gemeinsamer Speicher zur Verfügung, da diese Ressourcen einem Block zugeordnet werden.
- Wenn mehr als ein Multiprozessor zum Einsatz kommt, führt dies auch zu einer weiteren Parallelisierung.

bigvector.cu

```
template<typename Index, typename Alpha, typename TX, typename TY>
__global__ void axpy(Index n, Alpha alpha,
      const TX* x, Index incX, TY* y, Index incY) {
   unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
   if (tid < n) {
      y[tid*incY] += alpha * x[tid*incX];
   }
}
```

- *threadIdx.x* gibt jetzt nur noch den Thread-Index innerhalb eines Blocks an und ist somit alleine nicht mehr geeignet, die eindeutige Thread-ID im Bereich von 0 bis $n-1$ zu bestimmen.
- *blockIdx.x* liefert jetzt zusätzlich noch den Block-Index und *blockDim.x* die Zahl der Threads pro Block.
- Wenn n nicht durch die verwendete Blockgröße teilbar ist, dann erhalten wir Threads, die nichts zu tun haben. Um einen Zugriff außerhalb der Array-Grenzen zu vermeiden, benötigen wir eine entsprechende **if**-Anweisung.

In der allgemeinen Form akzeptiert die Konfiguration vier Parameter. Davon sind die beiden letzten optional:
$<<<$ *Dg*, *Db*, *Ns*, *S* $>>>$

- *Dg* legt die Dimensionierung des Grids fest (ein- oder zwei- oder dreidimensional).
- *Db* legt die Dimensionierung eines Blocks fest (ein-, zwei- oder dreidimensional).
- *Ns* legt den Umfang des gemeinsamen Speicherbereichs per Block fest (per Voreinstellung 0 bzw. vom Übersetzer bestimmt).
- *S* erlaubt die Verknüpfung mit einem Stream (per Voreinstellung mit dem NULL-Stream, d.h. es erfolgt immer eine Synchronisierung).
- *Dg* und *Db* sind beide vom Typ *dim3*, der mit eins bis drei ganzen Zahlen initialisiert werden kann.
- Vorgegebene Beschränkungen sind bei der Dimensionierung zu berücksichtigen. Sonst kann der Kernel nicht gestartet werden.

In den auf der GPU laufenden Funktionen stehen spezielle Variablen zur Verfügung, die die Identifizierung bzw. Einordnung des eigenen Threads ermöglichen:

threadIdx.x	x-Koordinate innerhalb des Blocks
threadIdx.y	y-Koordinate innerhalb des Blocks
threadIdx.z	z-Koordinate innerhalb des Blocks
blockDim.x	Dimensionierung des Blocks für x
blockDim.y	Dimensionierung des Blocks für y
blockDim.z	Dimensionierung des Blocks für z
blockIdx.x	x-Koordinate innerhalb des Gitters
blockIdx.y	y-Koordinate innerhalb des Gitters
blockIdx.z	z-Koordinate innerhalb des Gitters
gridDim.x	Dimensionierung des Gitters für x
gridDim.y	Dimensionierung des Gitters für y
gridDim.z	Dimensionierung des Gitters für z

Simpson-Verfahren auf der GPU

simpson.cu

```
// to be integrated function
__device__ double f(double x) {
   return 4 / (1 + x*x);
}

// numerical integration according to the Simpson rule
// for f over the i-th subinterval of [a,b]
__global__ void simpson(double a, double b, double* sums) {
   const int N = blockDim.x * gridDim.x;
   const unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
   double xleft = a + (b - a) / N * i;
   double xright = xleft + (b - a) / N;
   double xmid = (xleft + xright) / 2;
   sums[i] = (xright - xleft) / 6 * (f(xleft) + 4 * f(xmid) + f(xright));
}
```

- Die Kernel-Funktion berechnet hier jeweils nur ein Teilintervall und ermittelt mit Hilfe von $blockDim.x * gridDim.x$, wieviel Teilintervalle es gibt.
- Funktionen, die auf der GPU nur von anderen GPU-Funktionen aufzurufen sind, werden mit dem Schlüsselwort **__device__** gekennzeichnet.

simpson.cu

```
double* cuda_sums;
CHECK_CUDA(cudaMalloc, (void**)&cuda_sums, N * sizeof(double));
simpson<<<nof_blocks, blocksize>>>(a, b, cuda_sums);

double sums[N];
CHECK_CUDA(cudaMemcpy, sums, cuda_sums,
   N * sizeof(double), cudaMemcpyDeviceToHost);
CHECK_CUDA(cudaFree, cuda_sums);
double sum = 0;
for (int i = 0; i < N; ++i) {
   sum += sums[i];
}
```

- Es gibt keine vorgegebenen Aggregierungs-Operatoren, so dass diese „von Hand“ durchgeführt werden müssen.

Blockweise Aggregierung

simpson2.cu

```
#define THREADS_PER_BLOCK 256 /* must be a power of 2 */

// numerical integration according to the Simpson rule
// for f over the i-th subinterval of [a,b]
__global__ void simpson(double a, double b, double* sums) {
   /* compute approximative sum for our sub-interval */
   const int N = blockDim.x * gridDim.x;
   const unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
   // ...
   double sum = (xright - xleft) / 6 * (f(xleft) +
      4 * f(xmid) + f(xright));

   /* store it into the per-block shared array of sums */
   unsigned int me = threadIdx.x;
   __shared__ double sums_per_block[THREADS_PER_BLOCK];
   sums_per_block[me] = sum;

   // ...
}
```

- Innerhalb eines Blocks ist eine Synchronisierung und die Nutzung eines gemeinsamen Speicherbereiches möglich.
- Das eröffnet die Möglichkeit der blockweisen Aggregierung.

simpson2.cu

```
/* store it into the per-block shared array of sums */
unsigned int me = threadIdx.x;
__shared__ double sums_per_block[THREADS_PER_BLOCK];
sums_per_block[me] = sum;
```

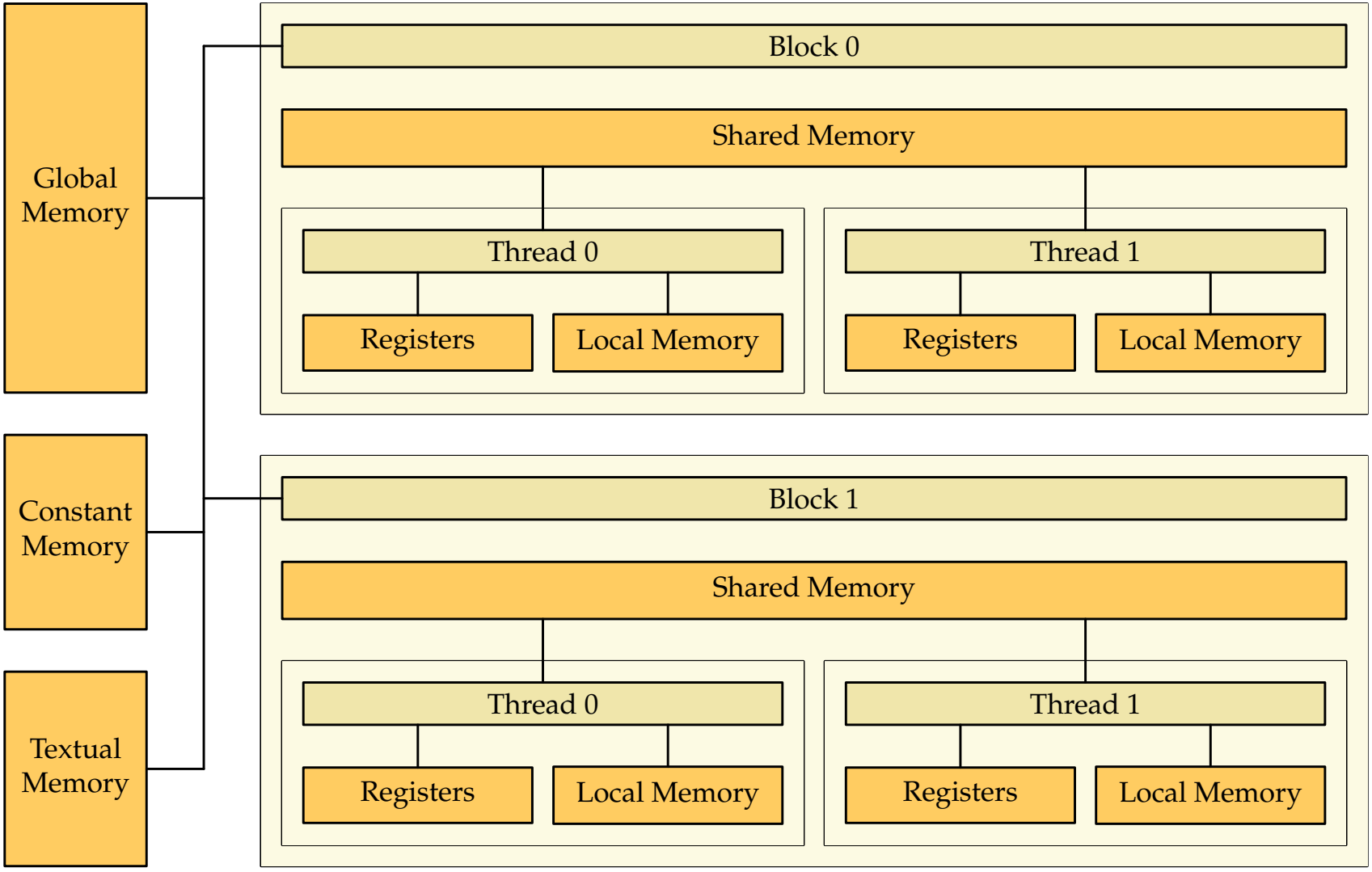
- Mit **__shared__** können Variablen gekennzeichnet werden, die allen Threads eines Blocks gemeinsam zur Verfügung stehen.
- Die Zugriffe auf diese Bereiche sind schneller als auf den globalen GPU-Speicher.
- Allerdings ist die Kapazität begrenzt. Auf Hochwanner stehen pro Block nur 48 KiB zur Verfügung.

simpson2.cu

```
/* aggregate sums within a block */
int index = blockDim.x / 2;
while (index) {
   __syncthreads();
   if (me < index) {
      sums_per_block[me] += sums_per_block[me + index];
   }
   index /= 2;
}
/* publish result */
if (me == 0) {
   sums[blockIdx.x] = sums_per_block[0];
}
```

- Zwar werden die jeweils einzelnen Gruppen eines Blocks zu Warps zusammengefasst, die entsprechend der SIMD-Architektur synchron laufen, aber das umfasst nicht den gesamten Block.
- Wenn alle Threads eines globalen Blocks synchronisiert werden sollen, geht dies mit der Funktion *__syncthreads*, die den aufrufenden Thread solange blockiert, bis alle Threads des Blocks diese Funktion aufrufen.

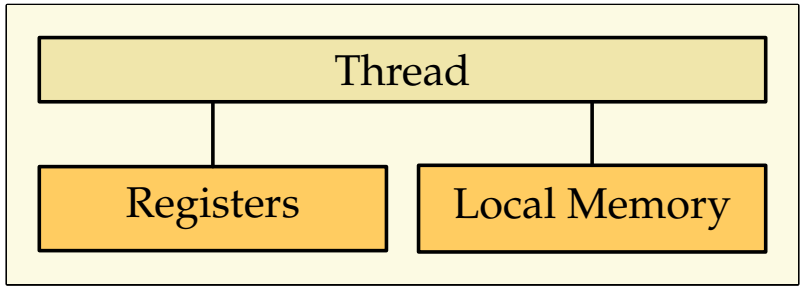
Speicherbereiche einer GPU



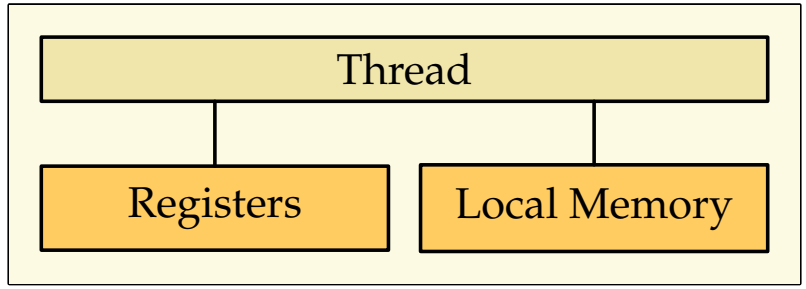
Ausgabe des Speicherplatzverbrauchs


```
hochwanner$ make
nvcc -o simpson --gpu-architecture compute_20 -code sm_20 --ptxas-options=-v s
ptxas info    : 304 bytes gmem, 40 bytes cmem[14]
ptxas info    : Compiling entry function '_Z7simpsonddPd' for 'sm_20'
ptxas info    : Function properties for _Z7simpsonddPd
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 25 registers, 56 bytes cmem[0], 12 bytes cmem[16]
hochwanner$
```

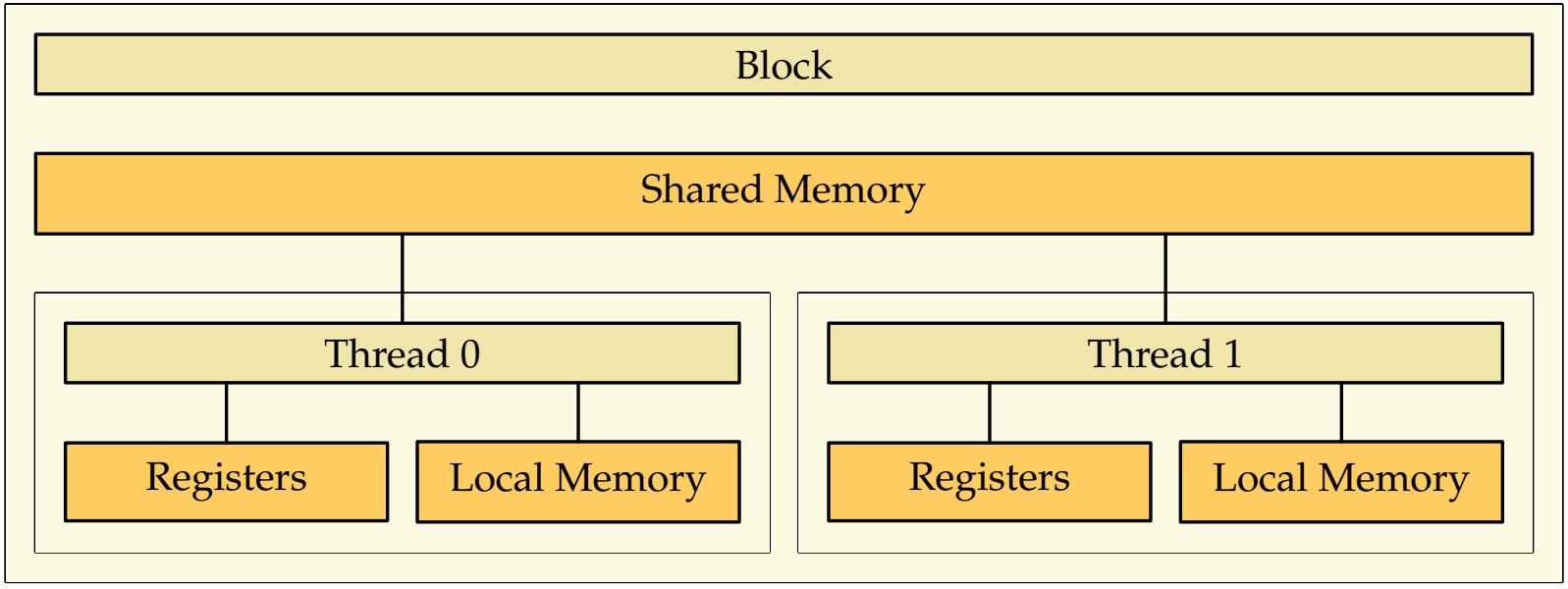
- *ptxas* dokumentiert den Verbrauch der einzelnen Speicherbereiche für eine Kernel-Funktion, wenn die entsprechende Verbose-Option gegeben wird.
- *gmem* steht hier für *global memory*, *cmem* für *constant memory*, das in Abhängigkeit der jeweiligen GPU-Architektur in einzelne Bereiche aufgeteilt wird.
- Lokaler Speicher wird verbraucht durch das *stack frame* und das Sichern von Registern (*spill stores*). Die *spill stores* und *spill loads* werden aber nur statisch an Hand des erzeugten Codes gezählt.



- Register gibt es für ganzzahlige Datentypen oder Gleitkommazahlen.
- Lokale Variablen innerhalb eines Threads werden soweit wie möglich in Registern abgelegt.
- Wenn sehr viel Register benötigt werden, kann dies dazu führen, dass weniger Threads in einem Block zur Verfügung stehen als das maximale Limit angibt.
- Die Hochwanner bietet beispielsweise 32768 Register per Block. Wenn das Maximum von 1024 Threads pro Block ausgeschöpft wird, verbleiben nur 32 Register für jeden Thread.



- Für den lokalen Speicher wird tatsächlich globaler Speicher verwendet.
- Es gibt allerdings spezielle cache-optimierte Instruktionen für das Laden und Speichern von und in den lokalen Speicher. Dies lässt sich optimieren, weil das Problem der Cache-Kohärenz wegfällt.
- Normalerweise erfolgen Lese- und Schreibzugriffe entsprechend nur aus bzw. in den L1-Cache. Wenn jedoch Zugriffe auf globalen Speicher notwendig werden, dann ist dies um ein Vielfaches langsamer als der gemeinsame Speicherbereich.
- Benutzt wird der lokale Speicher für den Stack, wenn die Register ausgehen und für lokale Arrays, bei denen Indizes zur Laufzeit berechnet werden.



- Nach den Registern bietet der für jeweils einen Block gemeinsame Speicher die höchste Zugriffsgeschwindigkeit.
- Die Kapazität ist sehr begrenzt. Auf Hochwanner stehen nur 48 KiB pro Block zur Verfügung.

- Der gemeinsame Speicher ist zyklisch in Bänke (*banks*) aufgeteilt. Das erste Wort im gemeinsamen Speicher (32 Bit) gehört zur ersten Bank, das zweite Wort zur zweiten Bank usw. Auf Hochwanner gibt es 32 solcher Bänke. Das 33. Wort gehört dann wieder zur ersten Bank.
- Zugriffe eines Warps auf unterschiedliche Bänke erfolgen gleichzeitig. Zugriffe auf die gleiche Bank müssen serialisiert werden, wobei je nach Architektur Broad- und Multicasts möglich sind, d.h. ein Wort kann gleichzeitig an alle oder mehrere Threads eines Warps verteilt werden.

- Der globale Speicher ist für alle Threads und (mit Hilfe von *cudaMemcpy*) auch von der CPU aus zugänglich.
- Anders als der reguläre Hauptspeicher findet bei dem globalen GPU-Speicher kein Paging statt. D.h. es gibt nicht virtuell mehr Speicher als physisch zur Verfügung steht.
- Auf Hochwanner steht 1 GiB zur Verfügung und ca. 1,2 GiB auf Olympia.
- Zugriffe erfolgen über L1 und L2, wobei (bei unseren GPUs) Cache-Kohärenz nur über den globalen L2-Cache hergestellt wird, d.h. Schreib-Operationen schlagen sofort auf den L2 durch.
- Globale Variablen können mit **__global__** deklariert werden oder dynamisch belegt wereden.

Zugriffe auf globalen Speicher sind unter den folgenden Bedingungen schnell:

- Der Zugriff erfolgt auf Worte, die mindestens 32 Bit groß sind.
- Die Zugriffsadressen müssen aufeinanderfolgend sein entsprechend der Thread-IDs innerhalb eines Blocks.
- Das erste Wort muss auf einer passenden Speicherkante liegen:

Wortgröße	Speicherkante
32 Bit	64 Byte
64 Bit	128 Byte
128 Bit	256 Byte

Bei der Fermi-Architektur (bei uns auf Hochwanner und Olympia) erfolgen die Zugriffe durch den L1- und L2-Cache:

- Die Cache-Lines bei L1 und L2 betragen jeweils 128 Bytes. (Entsprechend ergibt sich ein Alignment von 128 Bytes.)
- Wenn die gewünschten Daten im L1 liegen, dann kann innerhalb einer Transaktion eine Cache-Line mit 128 Bytes übertragen werden.
- Wenn die Daten nicht im L1, jedoch im L2 liegen, dann können per Transaktion 32 Byte übertragen werden.
- Die Restriktion, dass die Zugriffe konsekutiv entsprechend der Thread-ID erfolgen müssen, damit es effizient abläuft, entfällt. Es kommt nur noch darauf an, dass alle durch einen Warp gleichzeitig erfolgenden Zugriffe in eine gemeinsame Cache-Line passen.

- Wird von dem Übersetzer verwendet (u.a. für die Parameter der Kernel-Funktion) und es sind auch eigene Deklarationen mit dem Schlüsselwort **__constant__** möglich.
- Zur freien Verwendung stehen auf Hochwanner und Olympia 64 KiB zur Verfügung.
- Die Zugriffe erfolgen optimiert, weil keine Cache-Kohärenz gewährleistet werden muss.
- Schreibzugriffe sind zulässig, aber innerhalb eines Kernels wegen der fehlenden Cache-Kohärenz nicht sinnvoll.

tracer.cu

```
__constant__ char sphere_storage[sizeof(Sphere)*SPHERES];
```

- Variablen im konstanten Speicher werden mit **___constant___** deklariert.
- Datentypen mit nicht-leeren Konstruktoren oder Destruktoren werden in diesem Bereich jedoch nicht unterstützt, so dass hier nur die entsprechende Fläche reserviert wird.
- Mit *cudaMemcpyToSymbol* kann dann von der CPU eine Variable im konstanten Speicher beschrieben werden.

tracer.cu

```
Sphere host_spheres[SPHERES];
// fill host_spheres...
// copy spheres to constant memory
CHECK_CUDA(cudaMemcpyToSymbol, sphere_storage,
   host_spheres, sizeof(host_spheres));
```

- Bislang wurden überwiegend alle CUDA-Aktivitäten sequentiell durchgeführt, abgesehen davon, dass die Kernel-Funktionen parallelisiert abgearbeitet werden und der Aufruf eines Kernels asynchron erfolgt.
- In vielen Fällen bleibt so Parallelisierungspotential ungenutzt.
- CUDA-Streams sind eine Abstraktion, mit deren Hilfe mehrere sequentielle Abläufe definiert werden können, die voneinander unabhängig sind und daher prinzipiell parallelisiert werden können.
- Ferner gibt es Synchronisierungsoperationen und das Behandeln von Ereignissen mit CUDA-Streams.

Folgende Aktivitäten können mit Hilfe von CUDA-Streams unabhängig voneinander parallel laufen:

- CPU und GPU können unabhängig voneinander operieren
- der Transfer von Daten und die Ausführung von Kernel-Funktionen.
- Mehrere Kernel können auf der gleichen GPU konkurrierend ausgeführt werden (bei Hochwanner können vier Kernel parallel laufen).
- Wenn mehrere GPUs zur Verfügung stehen, können diese ebenfalls parallel laufen.

Insbesondere bietet es sich an, den Datentransfer und die Ausführung der Kernel-Funktionen zu parallelisieren. Dabei können insbesondere Datentransfers vom Hauptspeicher zur GPU und in umgekehrter Richtung von der GPU zum Hauptspeicher ungestört parallel laufen.

Sobald ein CUDA-Stream erzeugt worden ist, können einzelne Operationen oder der Aufruf einer Kernel-Funktion einem Stream zugeordnet werden:

cudaError_t cudaStreamCreate (cudaStream_t∗ stream)
: Erzeugt einen neuen Stream. Bei *cudaStream_t* handelt es sich um einen Zeiger auf eine nicht-öffentliche Datenstruktur, die beliebig kopiert werden kann.

cudaError_t cudaStreamSynchronize (cudaStream_t stream)
: Wartet bis alle Aktivitäten des Streams beendet sind.

cudaError_t cudaStreamDestroy (cudaStream_t stream)
: Wartet auf die Beendigung der mit dem Stream verbundenen Aktivitäten und anschließende Freigabe der zum Stream gehörenden Ressourcen.

Für die Datentransfers stehen asynchrone Operationen zur Verfügung, die einen Stream als Parameter erwarten:

cudaError_t cudaMemcpyAsync(**void**∗ *dst*, **const void**∗ *src*, *size_t count*, **enum** *cudaMemcpyKind kind*, *cudaStream_t stream*)

Funktioniert analog zu *cudaMemcpy*, synchronisiert jedoch nicht und reiht den Datentransfer in die zu dem Stream gehörende Sequenz ein.

Beim Aufruf eines Kernels kann bei dem letzten Parameter der Konfiguration ein Stream angegeben werden:

- $<<<$ *Dg*, *Db*, *Ns*, *S* $>>>$
- Der letzte Parameter *S* ist der Stream.
- Bei *Ns* kann im Normalfall einfach 0 angegeben werden.

- Grundsätzlich kann auch ein 0-Zeiger (bzw. **nullptr**) als Stream übergeben werden.
- In diesem Fall werden ähnlich wie bei *cudaDeviceSynchronize* erst alle noch nicht abgeschlossenen CUDA-Operationen abgewartet, bevor die Operation beginnt.
- Das erfolgt aber asynchron, so dass die CPU dessen ungeachtet weiter fortfahren kann.
- Datentransfers und der Aufruf von Kernel-Funktionen ohne die Angabe eines Streams implizieren immer die Verwendung des NULL-Streams. Entsprechend wird in diesen Fällen implizit synchronisiert.
- Wenn versucht wird, mit Streams zu parallelisieren, ist darauf zu achten, dass nicht versehentlich durch die implizite Verwendung eines NULL-Streams eine Synchronisierung erzwungen wird.

Wenn mehrere voneinander unabhängige Kernel-Funktionen hintereinander aufzurufen sind, die jeweils Daten von der CPU benötigen und Daten zurückliefern, lohnt sich u.U. ein Pipelining mit Hilfe von Streams:

- Für jeden Aufruf einer Kernel-Funktion wird ein Stream angelegt.
- Jedem Stream werden drei Operationen zugeordnet:
  - Datentransfer zur GPU
  - Aufruf der Kernel-Funktion
  - Datentransfer zum Hauptspeicher

Wenn der Zeitaufwand für die Datentransfers geringer ist als für die eigentliche Berechnung auf der GPU fällt dieser dank der Parallelisierung weg bei der Berücksichtigung der Gesamtzeit, abgesehen von dem ersten und letzten Datentransfer.

- Der GPU steht über die PCIe-Schnittstelle *direct memory access* (DMA) zur Verfügung.
- Dies wäre recht schnell, kommt aber normalerweise nicht zum Zuge, da dazu sichergestellt sein muss, dass die entsprechenden Kacheln im Hauptspeicher nicht zwischenzeitlich vom Betriebssystem ausgelagert werden.
- Alternativ ist es möglich, ausgewählte Bereiche des Hauptspeichers zu reservieren, so dass diese nicht ausgelagert werden können (*pinned memory*).
- Davon sollte zurückhaltend Gebrauch gemacht werden, da dies ein System in die Knie zwingen kann, wenn zuviele physische Kacheln reserviert sind.

Nicht auslagerbarer Speicher (*pinned memory*) muss mit speziellen Funktionen belegt und freigegeben werden:

cudaError_t cudaMallocHost(**void**∗∗ *ptr*, *size_t size*)
: belegt ähnlich wie *malloc* Hauptspeicher, wobei hier sichergestellt wird, dass dieser nicht ausgelagert wird.

cudaError_t cudaFreeHost(**void**∗)
: gibt den mit *cudaMallocHost* oder *cudaHostAlloc* reservierten Speicher wieder frei.

Neuere Grafikkarten und Versionen der CUDA-Schnittstelle (einschließlich Hochwanner) erlauben die Abbildung nicht auslagerbaren Hauptspeichers in den Adressraum der GPU:

cudaError_t cudaHostAlloc(**void**∗∗ *ptr*, *size_t size*, **unsigned int** *flags*)
belegt Hauptspeicher, der u.a. in den virtuellen Adressraum der GPU abgebildet werden kann. Folgende miteinander kombinierbare Optionen gibt es:

cudaHostAllocDefault emuliert *cudaMallocHost*, d.h. der Speicher wird nicht abgebildet.

cudaHostAllocPortable macht den Speicherbereich allen Grafikkarten zugänglich (falls mehrere zur Verfügung stehen).

cudaHostAllocMapped bildet den Hauptspeicher in den virtuellen Adressraum der GPU ab

cudaHostAllocWriteCombined ermöglicht u.U. eine effizientere Lesezugriffe der GPU zu Lasten der Lesegeschwindigkeit auf der CPU.

Neuere CUDA-Versionen unterstützen *unified virtual memory* (UVM), bei dem die Zeiger auf der CPU- und GPU-Seite für abgebildeten Hauptspeicher identisch sind. (Dies gilt auch dann, wenn wenn die CPU mit 64-Bit- und die GPU mit 32-Bit-Zeigern arbeitet.)
Ohne UVM müssen die Zeiger abgebildet werden:

cudaError_t cudaHostGetDevicePointer(**void**∗∗ *pDevice*, **void**∗ *pHost*, **unsigned int** *flags*)

liefert für Hauptspeicher, der in den Adressraum der GPU abgebildet ist (*cudaDeviceMapHost* wurde angegeben) den entsprechenden Zeiger in den Adressraum der GPU. Bei *unified virtual memory* (UVM) sind beide Zeiger identisch. (Bei den *flags* ist nach dem aktuellen Stand der API immer 0 anzugeben.)

Ob UVM unterstützt wird oder nicht, lässt sich über das Feld *unifiedAddressing* aus der **struct** *cudaDeviceProp* ermitteln, die mit *cudaGetDeviceProperties* gefüllt werden kann.

Wann lohnt es sich, Hauptspeicher abzubilden? 63

- Bei integrierten Systemen wird jeglicher Kopieraufwand vermieden (siehe das Feld *integrated* in der **struct** *cudaDeviceProp*).
- Bei nicht-integrierten Systemen werden die Daten jeweils implizit per *direct memory access* transferiert.
- Wenn der Speicher auf der GPU sonst nicht ausreicht.
- Wenn jede Speicherzelle nicht mehr als einmal in konsekutiver Weise gelesen oder geschrieben wird (ansonsten sind Zugriffe durch einen Cache effizienter).
- Reine Schreibzugriffe sind günstiger, da hier die Synchronisierung wegfällt, d.h. die Umsetzung einer Schreib-Operation und die Fortsetzung der Kernel-Funktion erfolgen parallel.
- Bei Lesezugriffen ist der Vorteil geringer, da hier gewartet werden muss.

Die CUDA-Schnittstelle bietet auch die Möglichkeit, auf konventionelle Weise belegten Speicher (etwa mit **new** oder *malloc*) nachträglich gegen Auslagerung zu schützen:

cudaError_t cudaHostRegister(**void**∗ *ptr*, *size_t size*, **unsigned int** *flags*)
: schützt die Speicherfläche, auf die *ptr* verweist, vor einer Auslagerung. Zwei Optionen werden unterstützt:

  cudaHostRegisterPortable
  : die Speicherfläche wird von allen GPUs als nicht auslagerbar erkannt.

  cudaHostRegisterMapped
  : die Speicherfläche wird in den Adressraum der GPU abgebildet. (Achtung: Selbst bei UVM kann nicht auf *cudaHostGetDevicePointer* verzichtet werden.)

cudaError_t cudaHostUnregister(**void**∗ *ptr*)
: beendet den Schutz vor Auslagerung.

Die CUDA-Schnittstelle unterstützt Ereignisse, die der Synchronisierung und der Zeitmessung dienen:

cudaError_t cudaEventCreate(cudaEvent_t event)*
: legt ein Ereignis-Objekt an mit der Option *cudaEventDefault*, d.h. Zeitmessungen sind möglich, eine Synchronisierung erfolgt jedoch im *busy-wait*-Verfahren.

cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream)
: fügt in die zu *stream* gehörende Ausführungssequenz die Anweisung hinzu, das Eintreten des Ereignisses zu signalisieren.

cudaError_t cudaEventDestroy(cudaEvent_t event)
: gibt die mit dem Ereignis-Objekt verbundenen Ressourcen wieder frei.

CUDA-Event-Operationen zur Synchronisierung und Zeitmessung:

cudaError_t cudaEventSynchronize(cudaEvent_t event)

der aufrufende Thread wartet, bis das Ereignis eingetreten ist. Wenn jedoch *cudaEventRecord* vorher noch nicht aufgerufen worden ist, kehrt dieser Aufruf sofort zurück. Wenn die Option *cudaEventBlockingSync* nicht gesetzt wurde, wird im *busy-wait*-Verfahren gewartet.

*cudaError_t cudaEventElapsedTime(***float**** ms, cudaEvent_t start, cudaEvent_t end)*

liefert die Zeit in Millisekunden, die zwischen den Ereignissen *start* und *end* vergangen ist. Die Auflösung der Zeit beträgt etwa eine halbe Mikrosekunde. Diese Zeitmessung ist genauer als konventionelle Methoden, weil hierfür die Uhr auf der GPU verwendet wird.

Zeitmessung mit CUDA-Events


```
cudaEvent_t start_event; CHECK_CUDA(cudaEventCreate, &start_event);
cudaEvent_t end_event; CHECK_CUDA(cudaEventCreate, &end_event);
CHECK_CUDA(cudaEventRecord, start_event);

// kernel invocations, data transfers etc.

CHECK_CUDA(cudaEventRecord, end_event);
CHECK_CUDA(cudaDeviceSynchronize); // wait for everything to finish

float timeInMillisecs;
CHECK_CUDA(cudaEventElapsedTime, &timeInMillisecs,
   start_event, end_event);
std::cerr << "GPU time in ms: " << timeInMillisecs << std::endl;
CHECK_CUDA(cudaEventDestroy, start_event);
CHECK_CUDA(cudaEventDestroy, end_event);
```

- Zu beachten ist hier, dass *cudaEventRecord* asynchron abgewickelt wird und das Ereignis erst signalisiert, wenn alle laufenden CUDA-Operationen abgeschlossen sind. Die CPU muss sich also danach immer noch mit *cudaDeviceSynchronize* synchronisieren.
- Zeitmessungen sollten immer auf Ereignissen beruhen, die mit dem NULL-Stream verbunden sind. Das Messen der Realzeit einzelner CUDA-Streams ist nicht sinnvoll, da sich jederzeit andere Operationen dazwischenschieben können.

- Neuere Nvidia-Grafikkarten (ab 3.5, nicht auf Hochwanner) geben Kernel-Funktionen die Möglichkeit, eine Reihe der CUDA-Funktionalitäten zu nutzen, die bislang nur auf der CPU-Seite zur Verfügung stehen.
- Insbesondere können Kernel-Funktionen von Kernel-Funktionen aufgerufen werden.
- Damit entfällt hier die Notwendigkeit, zeitaufwendig zur CPU zu synchronisieren und von der CPU eine Kernel-Funktion zu starten.
- Davon profitiert insbesondere das Master/Worker-Pattern.