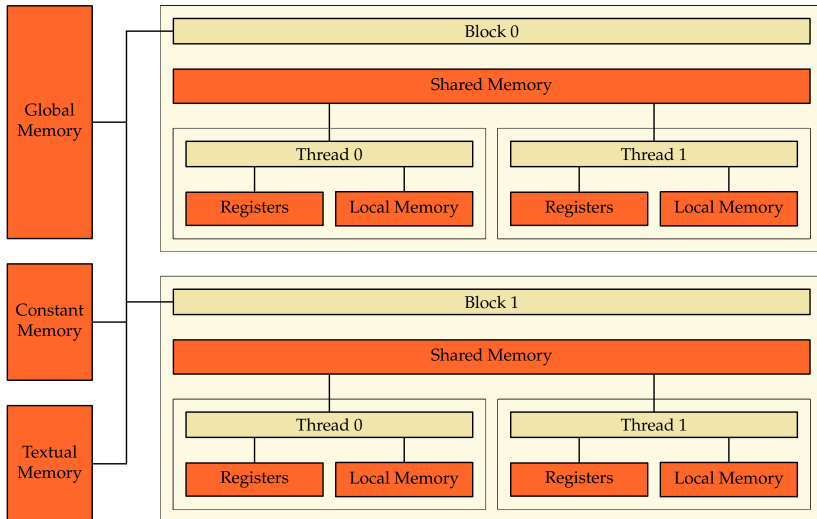
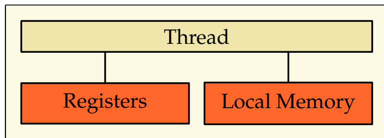
Speicherbereiche einer GPU

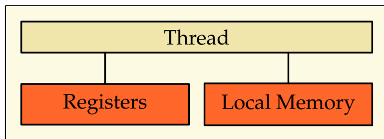


```
hochwanner$ make
nvcc -o simpson --gpu-architecture compute_20 -code sm_20 --ptxas-options=-v s
ptxas info    : 304 bytes gmem, 40 bytes cmem[14]
ptxas info    : Compiling entry function '_Z7simpsonddPd' for 'sm_20'
ptxas info    : Function properties for _Z7simpsonddPd
    0 bytes stack frame, 0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

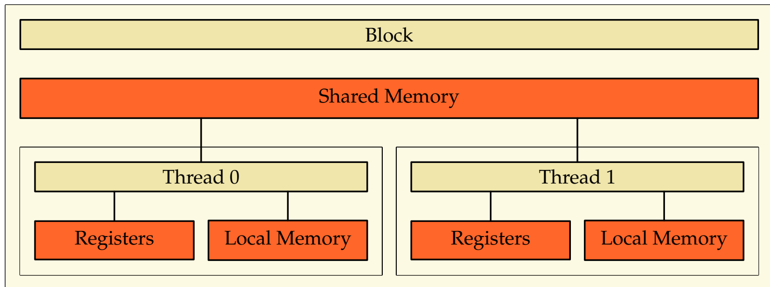
- *ptxas* dokumentiert den Verbrauch der einzelnen Speicherbereiche für eine Kernel-Funktion, wenn die entsprechende Verbose-Option gegeben wird.
- *gmem* steht hier für *global memory*, *cmem* für *constant memory*, das in Abhängigkeit der jeweiligen GPU-Architektur in einzelne Bereiche aufgeteilt wird.
- 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 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 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));
```

- Matrix-Matrix-Multiplikationen sind hochgradig parallelisierbar.
- Bei der Berechung von $C = A * B$ kann beispielsweise die Berechnung von $c_{i,j}$ an einen einzelnen Thread delegiert werden.
- Da größere Matrizen nicht mehr in einen Block (mit bei uns maximal 1024 Threads) passen, ist es sinnvoll, die gesamte Matrix in Blocks zu zerlegen.
- Dazu bieten sich 16×16 Blöcke mit 256 Threads an.
- O.B.d.A. betrachten wir nur quadratische $N \times N$ Matrizen mit $16 \mid N$.

mmm.cu

```
int main(int argc, char** argv) {
   cmdname = *argv++; --argc;
   if (argc != 2) usage();
   Matrix A; if (!read_matrix(*argv++, A)) usage(); --argc;
   Matrix B; if (!read_matrix(*argv++, B)) usage(); --argc;
   std::cout << "A = " << std::endl << A << std::endl;
   std::cout << "B = " << std::endl << B << std::endl;
   if (A.N != B.N) {
      std::cerr << cmdname << ": sizes of the matrices do not match"
         << std::endl; exit(1);
   }
   if (A.N % BLOCK_SIZE) {
      std::cerr << cmdname << ": size of matrices is not a multiply of "
         << BLOCK_SIZE << std::endl;
      exit(1);
   }

   A.copy_to_gpu(); B.copy_to_gpu();
   Matrix C; C.resize(A.N); C.allocate_cuda_data();

   dim3 block(BLOCK_SIZE, BLOCK_SIZE);
   dim3 grid(A.N / BLOCK_SIZE, A.N / BLOCK_SIZE);

   mmm<<<grid, block>>>(A, B, C);

   C.copy_from_gpu();
   std::cout << "C = " << std::endl << std::setprecision(14)
      << C << std::endl;
}
```

- Es ist sinnvoll, eine Klasse für Matrizen zu verwenden, die die Daten sowohl auf der CPU als auch auf der GPU je nach Bedarf hält.
- Diese Klasse kann dann auch das Kopieren der Daten unterstützen.
- Generell ist es sinnvoll, Kopieraktionen soweit wie möglich zu vermeiden, indem etwa Zwischenresultate nicht unnötig von der GPU zur CPU kopiert werden.
- Eine Klasse hat auch den Vorteil, dass die Freigabe der Datenflächen automatisiert wird.

mmm.cu

```
struct Matrix {
   unsigned int N;
   double* data;
   double* cuda_data;
   bool cloned;

   Matrix() : N(0), data(0), cuda_data(0), cloned(false) {
   }
   // ...
};
```

- *N* ist die Größe der Matrix, *data* der Zeiger in den Adressraum der CPU, *cuda_data* der Zeiger in den Adressraum der GPU.
- Mit *cloned* merken wir uns, ob es sich dabei um eine Kopie handelt.

mmm.cu

```
Matrix(const Matrix& other) : N(other.N), data(other.data),
      cuda_data(other.cuda_data), cloned(true) {
}
~Matrix() {
   if (!cloned) {
      if (data) delete data;
      if (cuda_data) release_cuda_data();
   }
}
```

- Ein Objekt kann auch als Parameter an eine Funktion übergeben werden.
- Dies geht mit dem Kopierkonstruktor.
- Zu beachten ist, dass die Kopie auf der CPU anschließend wieder dekonstruiert wird. Hier muss darauf geachtet werden, dass wir die Speicherflächen zu früh und am Ende doppelt freigeben.

mmm.cu

```
   void copy_to_gpu() {
      if (!cuda_allocated) {
         allocate_cuda_data();
      }
      CHECK_CUDA(cudaMemcpy, cuda_data, data, N * N * sizeof(double),
         cudaMemcpyHostToDevice);
   }

   void copy_from_gpu() {
      assert(cuda_allocated);
      CHECK_CUDA(cudaMemcpy, data, cuda_data, N * N * sizeof(double),
         cudaMemcpyDeviceToHost);
   }
```

- *copy_to_gpu* und *copy_from_gpu* kopieren die Matrix zur GPU und zurück.

mmm.cu

```
void allocate_cuda_data() {
   if (!cuda_allocated) {
      double* cudap;
      CHECK_CUDA(cudaMalloc, (void**)&cudap, N * N * sizeof(double));
      cuda_data = cudap;
      cuda_allocated = true;
   }
}

void release_cuda_data() {
   if (cuda_data) {
      CHECK_CUDA(cudaFree, cuda_data);
      cuda_data = 0;
   }
}
```

- Mit *allocate_cuda_data* wird die Matrix im Adressraum der GPU belegt, mit *release_cuda_data* wieder freigegeben.

mmm.cu

```
void resize(unsigned int N_) {
   if (N != N_) {
      double* rp = new double[N_ * N_];
      if (data) delete data;
      release_cuda_data();
      data = rp; N = N_;
   }
}
```

- Mit *resize* wird die Größe festgelegt bzw. verändert.

mmm.cu

```
   __device__ __host__ double& operator()(unsigned int i,
         unsigned int j) {
#ifdef __CUDA_ARCH__
      return cuda_data[i*N + j];
#else
      return data[i*N + j];
#endif
   }

   __device__ __host__ const double& operator()(unsigned int i,
         unsigned int j) const {
#ifdef __CUDA_ARCH__
      return cuda_data[i*N + j];
#else
      return data[i*N + j];
#endif
   }
```

- Mit **__device__ __host__** lassen sich Methoden und Funktionen auszeichnen, die sowohl für die CPU als auch die GPU übersetzt werden.
- Das Präprozessor-Symbol *__CUDA_ARCH__* ist nur dann definiert, wenn der Programmtext für die GPU übersetzt wird.

mmm-ab.cu

```
__global__ void mmm(Matrix a, Matrix b, Matrix c) {
   unsigned int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
   unsigned int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;

   double sum = 0;
   for (int k = 0; k < BLOCK_SIZE * gridDim.y; ++k) {
      sum += a(row, k) * b(k, col);
   }
   c(row, col) = sum;
}
```

- Dies ist die triviale Implementierung, bei der jeder Thread $c_{row,col}$ direkt berechnet.
- Der Zugriff auf a ist hier ineffizient, da ein Warp hier nicht auf konsekutiv im Speicher liegende Werte zugreift.

mmm.cu

```
   __shared__ double ablock[BLOCK_SIZE][BLOCK_SIZE];
   __shared__ double bblock[BLOCK_SIZE][BLOCK_SIZE];
```

- Wenn kein konsekutiver Zugriff erfolgt, kann es sich lohnen, dies über Datenstruktur abzuwickeln, die allen Threads eines Blocks gemeinsam ist.
- Die Idee ist, dass dieses Array gemeinsam von allen Threads eines Blocks konsekutiv gefüllt wird.
- Der Zugriff auf das gemeinsame Array ist recht effizient und muss nicht mehr konsekutiv sein.
- Die Matrix-Matrix-Multiplikation muss dann aber blockweise organisiert werden.

mmm.cu

```
__global__ void mmm(Matrix a, Matrix b, Matrix c) {
   __shared__ double ablock[BLOCK_SIZE][BLOCK_SIZE];
   __shared__ double bblock[BLOCK_SIZE][BLOCK_SIZE];
   unsigned int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
   unsigned int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;

   double sum = 0;
   for (int round = 0; round < gridDim.y; ++round) {
      ablock[threadIdx.y][threadIdx.x] =
         a(row, round*BLOCK_SIZE + threadIdx.x);
      bblock[threadIdx.y][threadIdx.x] =
         b(round*BLOCK_SIZE + threadIdx.y, col);
      __syncthreads();

      #pragma unroll
      for (int k = 0; k < BLOCK_SIZE; ++k) {
         sum += ablock[threadIdx.y][k] * bblock[k][threadIdx.x];
      }
      __syncthreads();
   }
   c(row, col) = sum;
}
```

mmm.cu

```
   for (int round = 0; round < gridDim.y; ++round) {
      ablock[threadIdx.y][threadIdx.x] =
         a(row, round*BLOCK_SIZE + threadIdx.x);
      bblock[threadIdx.y][threadIdx.x] =
         b(round*BLOCK_SIZE + threadIdx.y, col);
      __syncthreads();

      #pragma unroll
      for (int k = 0; k < BLOCK_SIZE; ++k) {
         sum += ablock[threadIdx.y][k] * bblock[k][threadIdx.x];
      }
      __syncthreads();
   }
```

- Jeder Block multipliziert *BLOCK_SIZE* Zeilen aus *A* mit *BLOCK_SIZE* Spalten aus *B*.
- Innerhalb jeder Runde werden bei einem Block die Zwischensummen für das Produkt zweier Teilmatrizen der Größe *BLOCK_SIZE* $\times$ *BLOCK_SIZE* berechnet.

mmm.cu

```
      #pragma unroll
      for (int k = 0; k < BLOCK_SIZE; ++k) {
         sum += ablock[threadIdx.y][k] * bblock[k][threadIdx.x];
      }
```

- Mit der Anweisung **#pragma** unroll wird der Übersetzer gebeten, auf die Generierung der folgenden Schleife zu verzichten und stattdessen die *BLOCK_SIZE* Additionen hintereinander zu generieren.
- Auf diese Weise fallen die bedingten Sprünge weg.