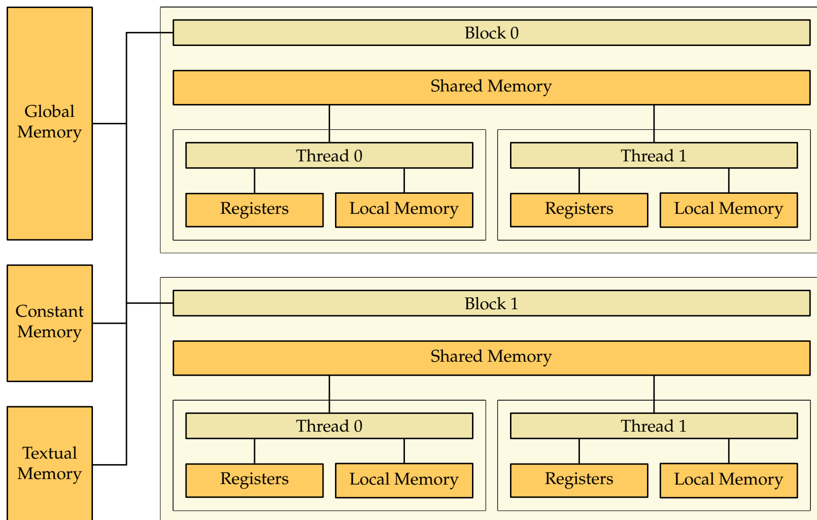
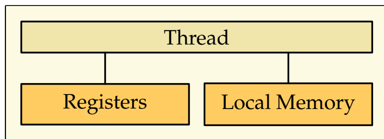
Speicherbereiche einer GPU

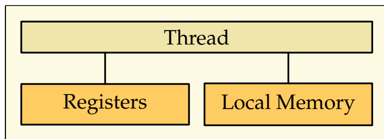


```
hochwanner$ make
nvcc -o simpson --gpu-architecture compute_20 -code sm_20 --ptxas-options=-v s
ptxas info    : 304 bytes gmem, 40 bytes cmem[14]
ptxas info    : Compiling entry function '_Z7simpsonddPd' for 'sm_20'
ptxas info    : Function properties for _Z7simpsonddPd
    0 bytes stack frame, 0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

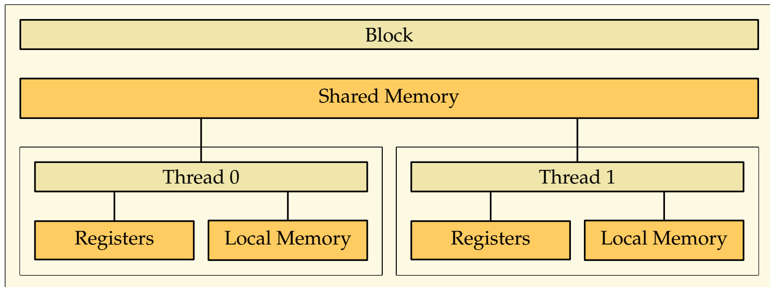
- *ptxas* dokumentiert den Verbrauch der einzelnen Speicherbereiche für eine Kernel-Funktion, wenn die entsprechende Verbose-Option gegeben wird.
- *gmem* steht hier für *global memory*, *cmem* für *constant memory*, das in Abhängigkeit der jeweiligen GPU-Architektur in einzelne Bereiche aufgeteilt wird.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

Triviale Umsetzung

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.

- 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.

Nutzung eines CUDA-Streams

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.

matrix.hpp

```
template<typename T>
class SquareMatrix {
   public:
      // ...
      void copy_to_gpu(cudaStream_t stream) {
         assert(data); allocate_cuda_data();
         CHECK_CUDA(cudaMemcpyAsync, cuda_data, data, get_size(),
            cudaMemcpyHostToDevice, stream);
      }

      void copy_from_gpu(cudaStream_t stream) {
         assert(cuda_data); if (!data) data = new T[N * N];
         CHECK_CUDA(cudaMemcpyAsync, data, cuda_data, get_size(),
            cudaMemcpyDeviceToHost, stream);
      }
      // ...
};
```

mmm-streamed.cu

```
cudaStream_t stream[COUNT];
for (unsigned int i = 0; i < COUNT; ++i) {
   CHECK_CUDA(cudaStreamCreate, &stream[i]);
   /* copy data to GPU */
   A[i].copy_to_gpu(stream[i]); B[i].copy_to_gpu(stream[i]);
   /* execute kernel on GPU */
   mmm<<<grid, block, 0, stream[i]>>>(A[i], B[i], C[i]);
   /* copy resulting data back to host */
   C[i].copy_from_gpu(stream[i]);
}
CHECK_CUDA(cudaDeviceSynchronize); // wait for everything to finish
```

- *COUNT* Matrix-Matrix-Multiplikationen werden hier parallel abgewickelt.
- Das entspricht hier dem Fork-And-Join-Pattern, wobei *cudaDeviceSynchronize* dem *join* entspricht.

```
hochwanner$ LD_LIBRARY_PATH=/usr/local/cuda-5.5/lib64 nvprof mmm
==16203== NVPROF is profiling process 16203, command: mmm
GPU time in ms: 2034.15
==16203== Profiling application: mmm
==16203== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 90.38%  1.64119s        10  164.12ms  164.08ms  164.14ms  mmm(SquareMatrix<double>, SquareMatrix<double
  5.19%  94.167ms        20  4.7083ms  4.6672ms  5.0939ms  [CUDA memcpy HtoD]
  4.44%  80.588ms        10  8.0588ms  8.0159ms  8.1020ms  [CUDA memcpy DtoH]
hochwanner$ LD_LIBRARY_PATH=/usr/local/cuda-5.5/lib64 nvprof mmm-without-transfers
==16214== NVPROF is profiling process 16214, command: mmm-without-transfers
GPU time in ms: 1641.09
==16214== Profiling application: mmm-without-transfers
==16214== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  1.64106s        10  164.11ms  164.09ms  164.12ms  mmm(SquareMatrix<double>, SquareMatrix<double
hochwanner$ hochwanner$ LD_LIBRARY_PATH=/usr/local/cuda-5.5/lib64 nvprof mmm-streamed
==16223== NVPROF is profiling process 16223, command: mmm-streamed
GPU time in ms: 1850.16
==16223== Profiling application: mmm-streamed
==16223== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 90.36%  1.64105s        10  164.11ms  164.09ms  164.14ms  mmm(SquareMatrix<double>, SquareMatrix<double
  5.19%  94.228ms        20  4.7114ms  4.6703ms  4.8342ms  [CUDA memcpy HtoD]
  4.45%  80.833ms        10  8.0833ms  8.0303ms  8.1465ms  [CUDA memcpy DtoH]
hochwanner$
```

- 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.

Nicht auslagerbarer Speicher

pinned-matrix.hpp

```
   void allocate_data() {
      if (!data) {
         CHECK_CUDA(cudaMallocHost, (void**)&data, get_size());
      }
   }
   void release_data() {
      if (data) {
         CHECK_CUDA(cudaFreeHost, data);
         data = 0;
      }
   }
```

- Die *Matrix*-Klasse kann entsprechend angepasst werden.

```
hochwanner$ LD_LIBRARY_PATH=/usr/local/cuda-5.5/lib64 nvprof mmm-streamed-and-
==16783== NVPROF is profiling process 16783, command: mmm-streamed-and-pinned
GPU time in ms: 1712.28
==16783== Profiling application: mmm-streamed-and-pinned
==16783== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 95.87%  1.64091s        10  164.09ms  164.07ms  164.10ms  mmm(SquareMatrix<do
  2.76%  47.267ms        20  2.3633ms  2.3505ms  2.4718ms  [CUDA memcpy HtoD]
  1.37%  23.461ms        10  2.3461ms  2.3449ms  2.3534ms  [CUDA memcpy DtoH]
hochwanner$
```

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?

- 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.

mapped-matrix.hpp

```
void allocate_data() {
   if (!data) {
      CHECK_CUDA(cudaHostAlloc, (void**)&data, get_size(),
         cudaHostAllocMapped);
   }
}

void allocate_cuda_data() {
   /* not actually allocating but
      accessing mapped host memory */
   if (!cuda_data) {
      allocate_data();
      CHECK_CUDA(cudaHostGetDevicePointer,
         (void**)&cuda_data, (void*)data, 0);
   }
}
```

- In dieser Fassung der *Matrix*-Implementierung entfallen die Transfer-Operationen bei *copy_to_gpu* und *copy_from_gpu*.

```
hochwanner$ LD_LIBRARY_PATH=/usr/local/cuda-5.5/lib64 nvprof mmm-mapped
==20142== NVPROF is profiling process 20142, command: mmm-mapped
GPU time in ms: 3047.22
==20142== Profiling application: mmm-mapped
==20142== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  3.04127s        10  304.13ms  302.77ms  314.53ms  mmm(SquareMatrix<do
hochwanner$
```

- Die Laufzeit hat sich katastrophal verschlechtert von 1712.28 ms bei effizienten Datentransfers mit nicht auslagerbaren Hauptspeicher zu 3047.22 ms bei dem direkten Zugriff auf den Hauptspeicher.
- Die Ursache ist hier darin zu finden, dass die beiden Ausgangsmatrizen nicht nur einmal, sondern N-fach ausgelesen werden, wenn es sich um eine $N \times N$-Matrix handelt.
- Entsprechend profitieren die Matrix-Zugriffe sehr von einem Cache, da auch die Chance sehr groß ist, dass parallel laufende Blöcke teilweise auf die gleichen Bereiche der Ausgangsmatrizen zugreifen.

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.

```
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.