- Matrix-Matrix-Multiplikationen sind hochgradig parallelisierbar.
- Bei der Berechung von $C \leftarrow \alpha AB + \beta C$ 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.

gemm1.hpp

```
template<typename Alpha, typename MA, typename MB,
   typename Beta, typename MC>
__global__ void
gemm_kernel(const Alpha alpha, const MA A, const MB B,
      const Beta beta, MC C) {
   using Index = typename std::common_type<typename MA::Index,
      typename MB::Index, typename MC::Index>::type;
   using T = typename std::common_type<typename MA::ElementType,
      typename MB::ElementType, typename MC::ElementType>::type;

   Index i = threadIdx.x + blockIdx.x * blockDim.x;
   Index j = threadIdx.y + blockIdx.y * blockDim.y;

   if (i < C.numRows && j < C.numCols) {
      T sum{};
      for (Index k = 0; k < A.numCols; ++k) {
         sum += A(i, k) * B(k, j);
      }
      sum *= alpha;
      sum += beta * C(i, j);
      C(i, j) = sum;
   }
}
```

gemm1.hpp

```
Index i = threadIdx.x + blockIdx.x * blockDim.x;
Index j = threadIdx.y + blockIdx.y * blockDim.y;

if (i < C.numRows && j < C.numCols) {
   T sum{};
   for (Index k = 0; k < A.numCols; ++k) {
      sum += A(i, k) * B(k, j);
   }
   sum *= alpha;
   sum += beta * C(i, j);
   C(i, j) = sum;
}
```

- Dies ist die triviale Implementierung, bei der jeder Thread $c_{row,col}$ direkt berechnet.
- Die Threads eines Warps bzw. Half-Warps unterscheiden sich nur durch *threadIdx.x*, haben *threadIdx.y* jedoch gemeinsam.
- Der Zugriff auf A und möglicherweise auch auf B und C sind hier ineffizient, da ein Warp bzw. Half-Warp nicht auf konsekutiv im Speicher liegende Werte zugreift.

gemm1.hpp

```
template<typename Alpha, typename MA, typename MB,
   typename Beta, typename MC>
void cuda_gemm(const Alpha alpha, const MA& A, const MB& B,
      const Beta beta, MC& C) {
   using Index = typename std::common_type<typename MA::Index,
      typename MB::Index, typename MC::Index>::type;
   assert(A.numRows == C.numRows && A.numCols == B.numRows &&
      B.numCols == C.numCols);
   constexpr Index blockdim = 16;
   dim3 block(blockdim, blockdim);
   Index M = C.numRows; Index N = C.numCols; Index K = A.numCols;
   using namespace hpc::aux;
   dim3 grid(ceildiv(M, blockdim), ceildiv(N, blockdim));
   gemm_kernel<<<grid, block>>>(alpha,
      A(0, 0, M, K), B(0, 0, K, N), beta, C(0, 0, M, N));
}
```

- An die Kernel-Funktion werden hier Views übergeben, d.h. Matrix-Objekte ohne eigene Datenhaltung.

- Prinzipiell können Matrix-Klassen eingerichtet werden, die Matrizen je nach Ausprägung entweder auf der GPU oder auf der CPU halten.
- Kopieroperationen ermöglichen das Verschieben der Daten. Effizient gelingt dies nur, wenn die jeweiligen Matrizen gleichartig im Speicher organisiert sind.
- Objekte des Typs *hpc*::*cuda*::*GeMatrix* leben nur auf der CPU-Seite, deren Daten liegen aber auf der GPU-Seite.
- Die zugehörigen Views des Typs *hpc*::*cuda*::*GeMatrixView* können sowohl auf der CPU- als auch der GPU-Seite verwendet werden, wobei wiederum die Daten nur auf der GPU-Seite zugänglich sind.

test-gemm.cu

```
using T = double;
using Index = std::size_t;
constexpr std::size_t M = 512;
constexpr std::size_t N = 512;
constexpr std::size_t K = 512;
T alpha = 1.0; T beta = 1.5;

hpc::matvec::GeMatrix<T, Index> A_host(M, K, hpc::matvec::RowMajor);
hpc::matvec::GeMatrix<T, Index> B_host(K, N, hpc::matvec::RowMajor);
hpc::matvec::GeMatrix<T, Index> C1_host(M, N, hpc::matvec::RowMajor);
hpc::matvec::GeMatrix<T, Index> C2_host(M, N, hpc::matvec::RowMajor);

hpc::cuda::GeMatrix<T, Index> A_dev(M, K, hpc::cuda::RowMajor);
hpc::cuda::GeMatrix<T, Index> B_dev(K, N, hpc::cuda::RowMajor);
hpc::cuda::GeMatrix<T, Index> C2_dev(M, N, hpc::cuda::RowMajor);

init_matrix(A_host); hpc::cuda::copy(A_host, A_dev);
init_matrix(B_host); hpc::cuda::copy(B_host, B_dev);
init_matrix(C1_host); hpc::cuda::copy(C1_host, C2_dev);

auto start = std::chrono::high_resolution_clock::now();
cuda_gemm(alpha, A_dev, B_dev, beta, C2_dev);
CHECK_CUDA(cudaDeviceSynchronize); // wait for the kernel to finish
auto finish = std::chrono::high_resolution_clock::now();

hpc::cuda::copy(C2_dev, C2_host);
```

gemm2.hpp

```
Index i = threadIdx.y + blockIdx.y * BLOCK_DIM;
Index j = threadIdx.x + blockIdx.x * BLOCK_DIM;

/* ... */

if (i < C.numRows && j < C.numCols) {
   C(i, j) = sum;
}
```

- Im folgenden gehen wir davon aus, dass sowohl A, B als auch C jeweils in *row major* organisiert sind.
- Dann ist es sinnvoll, den Spaltenindex, der auf benachbarte Speicherzellen zugreift, mit *threadIdx.x* zu verknüpfen.
- Damit wird sichergestellt, dass ein Warp bzw. Half-Warp auf benachbarte Daten zugreift.

gemm2.hpp

```
__shared__ T ablock[BLOCK_DIM][BLOCK_DIM];
__shared__ T bblock[BLOCK_DIM][BLOCK_DIM];
```

- 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.

gemm2.hpp

```
Index K = A.numCols;
Index rounds = (K + BLOCK_DIM - 1) / BLOCK_DIM;
T sum{};
for (Index round = 0; round < rounds; ++round) {
   T val;
   if (i < A.numRows && round*BLOCK_DIM + threadIdx.x < A.numCols) {
      val = A(i, round*BLOCK_DIM + threadIdx.x);
   } else {
      val = 0;
   }
   ablock[threadIdx.y][threadIdx.x] = val;
   if (round*BLOCK_DIM + threadIdx.x < B.numRows && j < B.numCols) {
      val = B(round*BLOCK_DIM + threadIdx.y, j);
   } else {
      val = 0;
   }
   bblock[threadIdx.y][threadIdx.x] = val;
   __syncthreads();

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

- 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.

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.

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.

- 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.