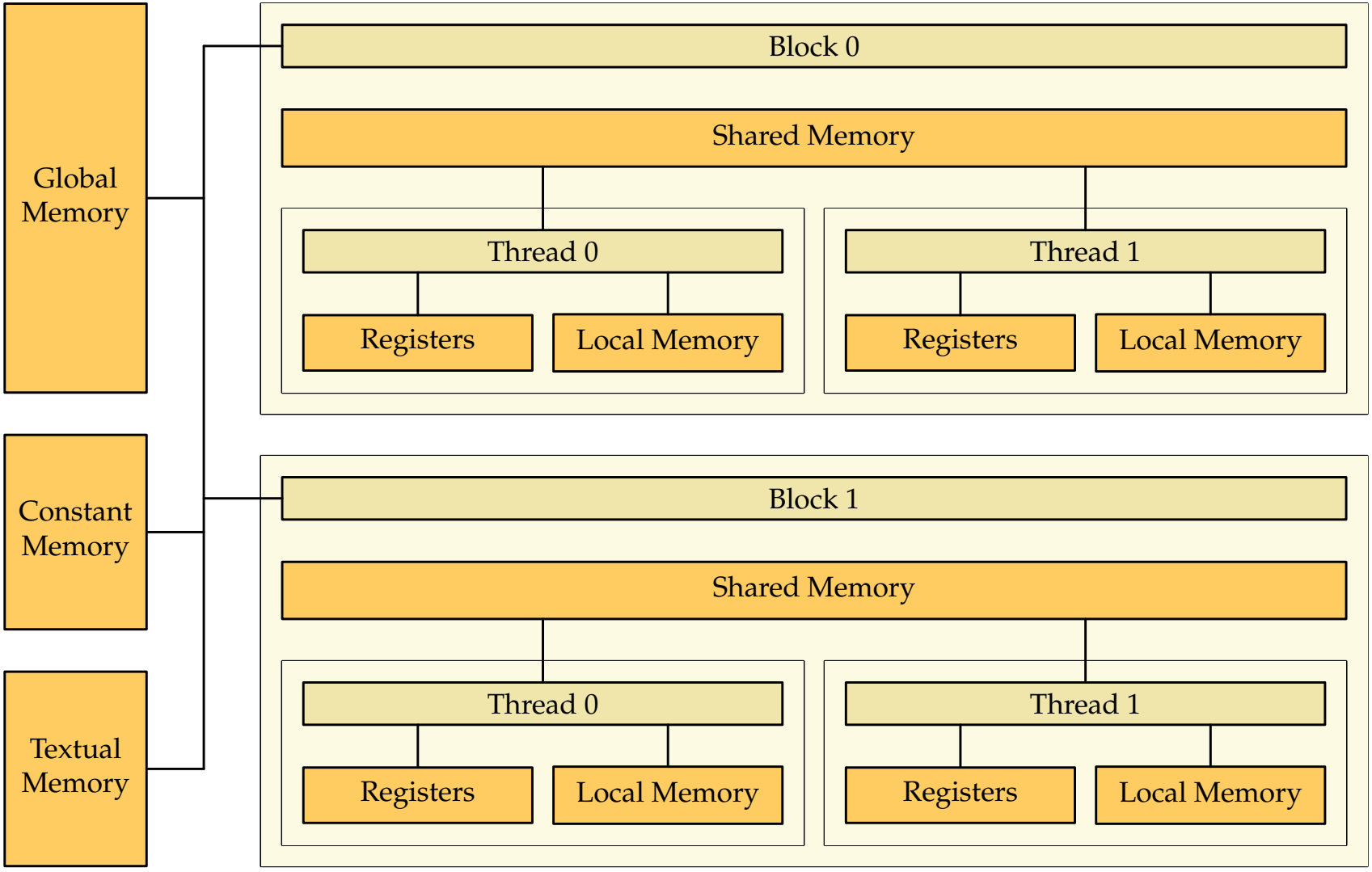
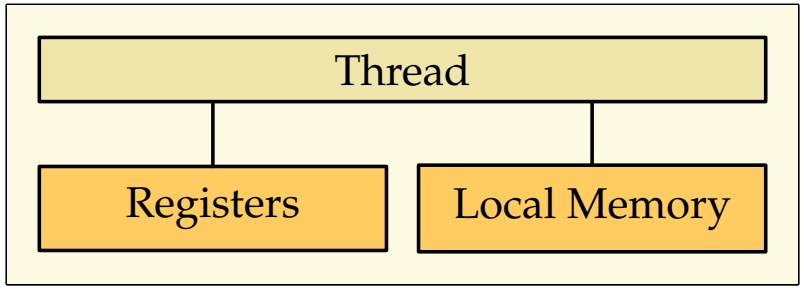
Speicherbereiche einer GPU

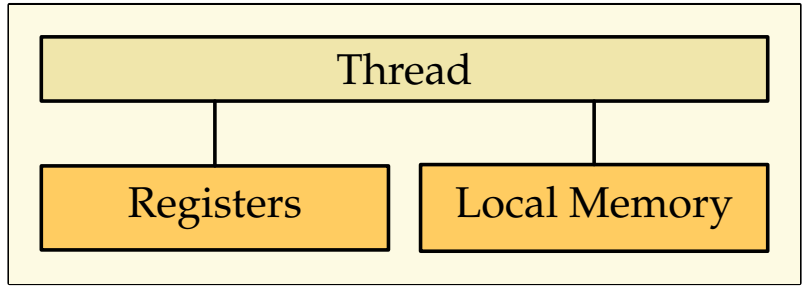


```
livingstone$ make
nvcc -o simpson -I/home/numerik/pub/pp/ss19/lib --ptxas-options=-
v simpson.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z7simpsonddPd' for 'sm_30'
ptxas info    : Function properties for _Z7simpsonddPd
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 26 registers, 344 bytes cmem[0], 40 bytes cmem[2]
livingstone$
```

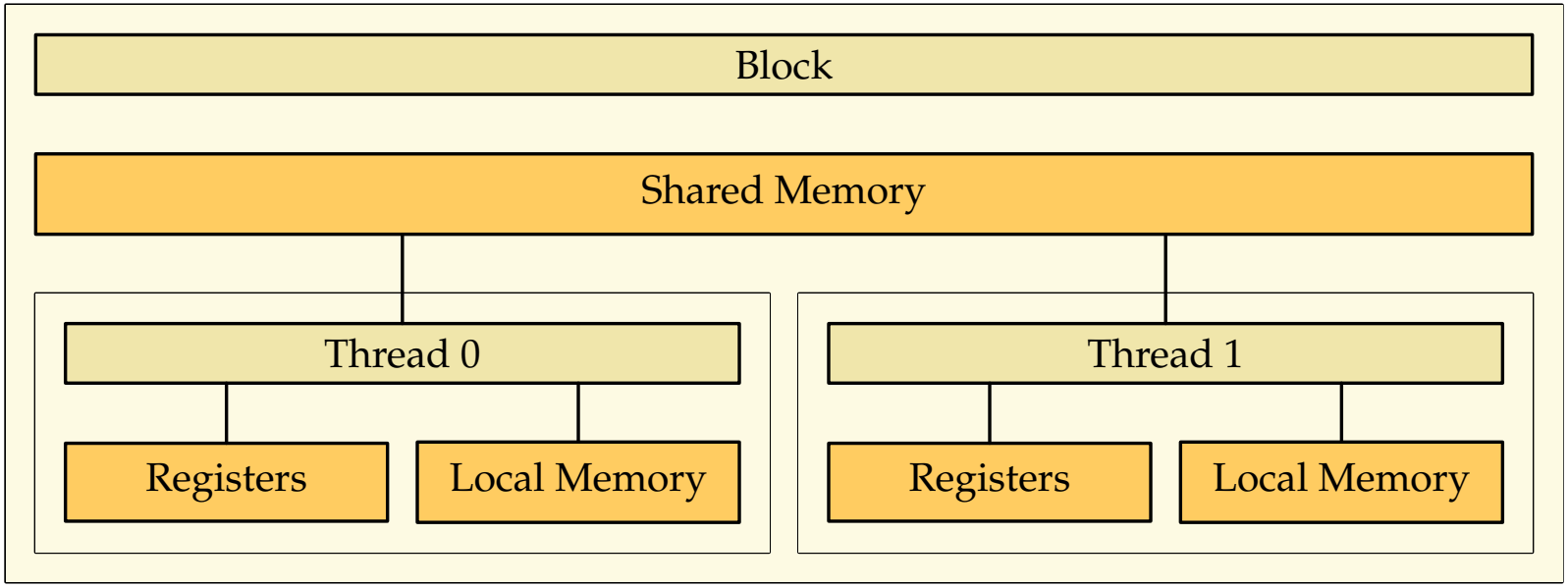
- *ptxas* dokumentiert den Verbrauch der einzelnen Speicherbereiche für eine Kernel-Funktion, wenn die entsprechende Verbose-Option gegeben wird.
- *gmem* steht hier für *global memory*, *cmem* für *constant memory*, das in Abhängigkeit der jeweiligen GPU-Architektur in einzelne Bereiche aufgeteilt wird.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



- 32-Bit-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 Livingstone bietet beispielsweise 65536 Register per Block. Wenn das Maximum von 1024 Threads pro Block ausgeschöpft wird, verbleiben nur 64 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 Livingstone stehen nur 48 KiB per Block zur Verfügung, insgesamt jedoch 96 KiB per Multiprozessor (bei dem möglicherweise mehrere Blöcke parallel laufen).

- 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 Livingstone 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 beim regulären Hauptspeicher findet bei dem globalen GPU-Speicher kein Paging statt. D.h. es gibt nicht virtuell mehr Speicher als physisch zur Verfügung steht.
- Auf Livingstone stehen 1,95 GiB zur Verfügung.
- 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.
- Der L2 hat auf Livingstone 512 KiB.
- Globale Variablen können mit **__global__** deklariert werden oder dynamisch belegt wereden.

Zugriffe auf globalen Speicher sind bei älteren GPU-Architekturen 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 Pascal-Architektur (bei uns auf Livingstone) erfolgen die Zugriffe normalerweise nur über den L2, für Zugriffe durch den L1 müssen spezielle Übersetzungsoptionen gewählt werden.

- Die Cache-Lines bei L1 und L2 betragen jeweils 128 Bytes. (Entsprechend ergibt sich ein Alignment von 128 Bytes.)
- Eine Cache-Line besteht aus 4 Segmenten zu je 32 Bytes, bei einem Cache-Miss werden Segmente gefüllt, jedoch nicht zwangsläufig die gesamte Cache-Line.
- 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.
- Auf Livingstone stehen hierfür 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 $C \leftarrow \alpha AB + \beta C$ 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 Beta,
   template<typename> class MatrixA,
   template<typename> class MatrixB,
   template<typename> class MatrixC,
   typename T,
   Require<
      DeviceGe<MatrixA<T>>, DeviceView<MatrixA<T>>,
      DeviceGe<MatrixB<T>>, DeviceView<MatrixB<T>>,
      DeviceGe<MatrixC<T>>, DeviceView<MatrixC<T>>,
   > = true
>
__global__ void gemm_kernel(const Alpha alpha,
      const MatrixA<T> A,  const MatrixB<T> B,
      const Beta beta, MatrixC<T> C) {
   std::size_t i = threadIdx.x + blockIdx.x * blockDim.x;
   std::size_t j = threadIdx.y + blockIdx.y * blockDim.y;

   if (i < C.numRows() && j < C.numCols()) {
      T sum{};
      for (std::size_t 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

```
std::size_t i = threadIdx.x + blockIdx.x * blockDim.x;
std::size_t j = threadIdx.y + blockIdx.y * blockDim.y;

if (i < C.numRows() && j < C.numCols()) {
   T sum{};
   for (std::size_t 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 Beta,
   template<typename> class MatrixA,
   template<typename> class MatrixB,
   template<typename> class MatrixC,
   typename T,
   Require<
      DeviceGe<MatrixA<T>>,
      DeviceGe<MatrixB<T>>,
      DeviceGe<MatrixC<T>>
   > = true
>
void cuda_gemm(const Alpha alpha,
      const MatrixA<T>& A,  const MatrixB<T>& B,
      const Beta beta, MatrixC<T>& C) {
   assert(A.numRows() == C.numRows() && A.numCols() == B.numRows() &&
      B.numCols() == C.numCols());
   constexpr std::size_t blockdim = 16;
   dim3 block(blockdim, blockdim);
   std::size_t M = C.numRows();
   std::size_t N = C.numCols();
   std::size_t K = A.numCols();
   using namespace hpc::aux;
   dim3 grid(ceildiv(M, blockdim), ceildiv(N, blockdim));
   gemm_kernel<<<grid, block>>>(alpha,
      A.view(), B.view(), beta, C.view());
}
```

gemm1.hpp

```
gemm_kernel<<<grid, block>>>(alpha,
   A.view(), B.view(), beta, C.view());
```

- An die Kernel-Funktion werden hier Views übergeben, d.h. Matrix-Objekte ohne eigene Datenhaltung.
- Die Views werden *by value* übergeben und enthalten nur die Dimensionierung der Matrix, einen Zeiger auf die Daten und Informationen zur Organisation der Matrix im Speicher.

- 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 *DeviceGeMatrix* leben nur auf der CPU-Seite, deren Daten liegen aber auf der GPU-Seite.
- Die zugehörigen Views des Typs *DeviceGeMatrixView* 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;
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;

using namespace hpc::cuda;
using namespace hpc::matvec;

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

DeviceGeMatrix<T> A_dev(M, K, Order::RowMajor);
DeviceGeMatrix<T> B_dev(K, N, Order::RowMajor);
DeviceGeMatrix<T> C2_dev(M, N, Order::RowMajor);

init_matrix(A_host); copy(A_host, A_dev);
init_matrix(B_host); copy(B_host, B_dev);
init_matrix(C1_host);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();

copy(C2_dev, C2_host);
```

gemm2.hpp

```
std::size_t i = threadIdx.y + blockIdx.y * BLOCK_DIM;
std::size_t 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

```
std::size_t K = A.numCols();
std::size_t rounds = (K + BLOCK_DIM - 1) / BLOCK_DIM;
T sum{};
for (std::size_t 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 (std::size_t k = 0; k < BLOCK_DIM; ++k) {
      sum += ablock[threadIdx.y][k] * bblock[k][threadIdx.x];
   }
   __syncthreads();
}
```