Matrix-Matrix-Multiplikation

- 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 organisiert werden.
- Da größere Matrizen nicht mehr in einen Block (mit bei uns maximal 1536 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;
   cout << "A = " << endl << A << endl;
   cout << "B = " << endl << B << endl;
   if (A.N != B.N) {
      cerr << cmdname << ": sizes of the matrices do not match" << endl;
      exit(1);
   }
   if (A.N % BLOCK_SIZE) {
      cerr << cmdname << ": size of matrices is not a multiply of "
         << BLOCK_SIZE << 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.cuda_data, B.cuda_data, C.cuda_data);

   C.copy_from_gpu();
   cout << "C = " << endl << setprecision(14) << C << 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 das die Freigabe der Datenflächen automatisiert wird.

mmm.cu

```
struct Matrix {
   unsigned int N;
   Real* data;
   bool cuda_allocated;
   Real* cuda_data;

   Matrix() :
      N(0), data(0), cuda_allocated(false), cuda_data(0) {
   }
   ~Matrix() {
      if (data) delete data;
      if (cuda_allocated) release_cuda_data();
   }

   // ...
};
```

- *N* ist die Größe der Matrix, *data* der Zeiger in den Adressraum der CPU, *cuda_data* der Zeiger in den Adressraum der GPU.

mmm.cu

```
bool copy_to_gpu() {
   if (!cuda_allocated) {
      if (!allocate_cuda_data()) return false;
   }
   return cudaMemcpy(cuda_data, data, N * N * sizeof(Real),
            cudaMemcpyHostToDevice) == cudaSuccess;
}

bool copy_from_gpu() {
   assert(cuda_allocated);
   return cudaMemcpy(data, cuda_data, N * N * sizeof(Real),
            cudaMemcpyDeviceToHost) == cudaSuccess;
}
```

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

mmm.cu

```
bool allocate_cuda_data() {
   if (cuda_allocated) return true;
   Real* cudap;
   if (cudaMalloc((void**)&cudap, N * N * sizeof(Real)) !=
         cudaSuccess) {
      return false;
   }
   cuda_data = cudap;
   cuda_allocated = true;
   return true;
}

void release_cuda_data() {
   if (cuda_data) {
      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

```
bool resize(unsigned int N_) {
   if (N == N_) return true;
   Real* rp = new Real[N_ * N_];
   if (!rp) return false;
   if (data) delete data;
   release_cuda_data();
   data = rp; N = N_;
   return true;
}

Real& operator()(unsigned int i, unsigned int j) {
   return data[i*N + j];
}

const Real& operator()(unsigned int i, unsigned int j) const {
   return data[i*N + j];
}
```

- Mit *resize* wird die Größe festgelegt bzw. verändert. Die beiden ()-Operatoren dienen dem indizierten Zugriff (auf der Seite der CPU).

mmm-ab.cu

```
#define ELEMENT(m,i,j) ((m)[(i) * stride + (j)])

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

   Real sum = 0;
   for (int k = 0; k < BLOCK_SIZE * gridDim.y; ++k) {
      sum += ELEMENT(a, row, k) * ELEMENT(b, k, col);
   }
   ELEMENT(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__ Real ablock[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.

Blockweise Multiplikation

mmm.cu

```
#define ELEMENT(m,i,j) ((m)[(i) * stride + (j)])

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

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

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