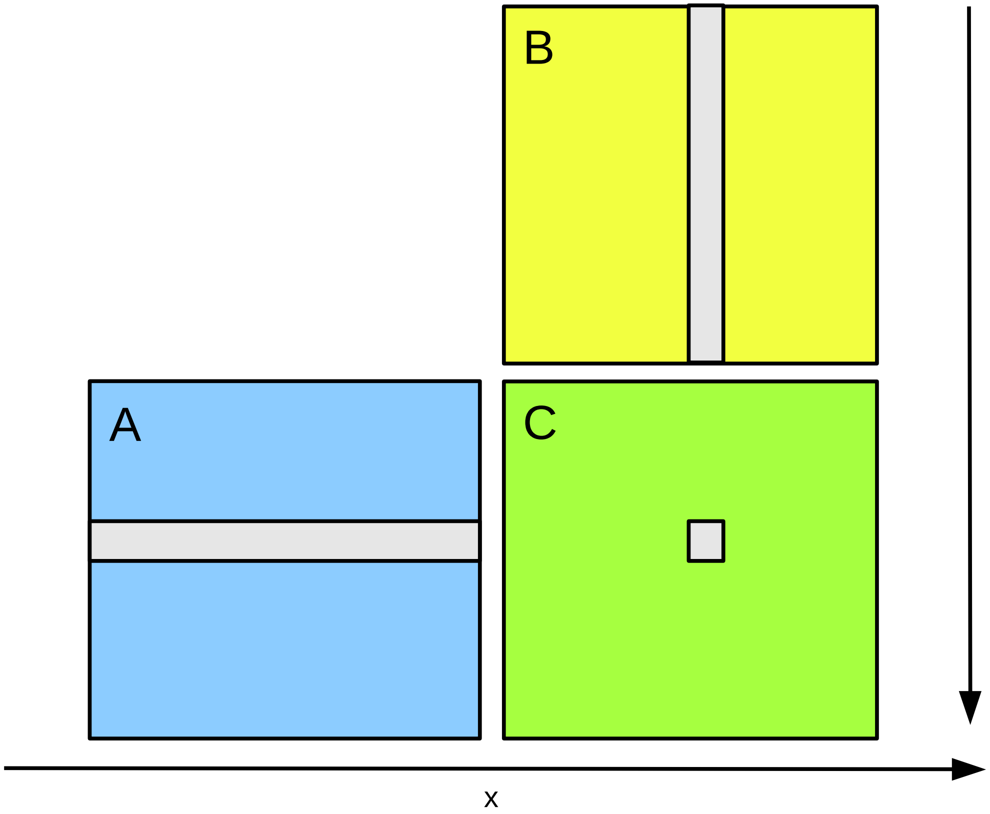
MATRIX MULTIPLICATION WITH CUDA

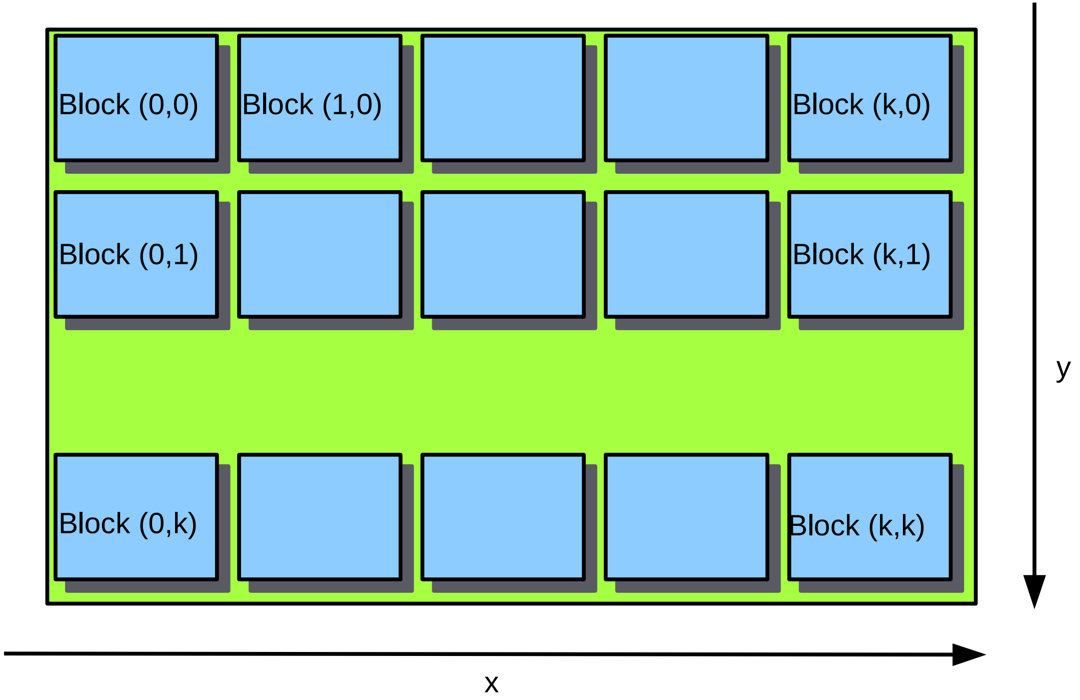
24. APRIL 2018 | JOCHEN KREUTZ

DISTRIBUTION OF WORK



- Each thread computes one element of the result matrix C
- n * n threads will be needed (for square matrix C)
- Indexing of threads corresponds to 2d indexing of the matrices
- Thread(x, y) will calculate element C(x, y) using row y of A and column x of B

DISTRIBUTION OF WORK

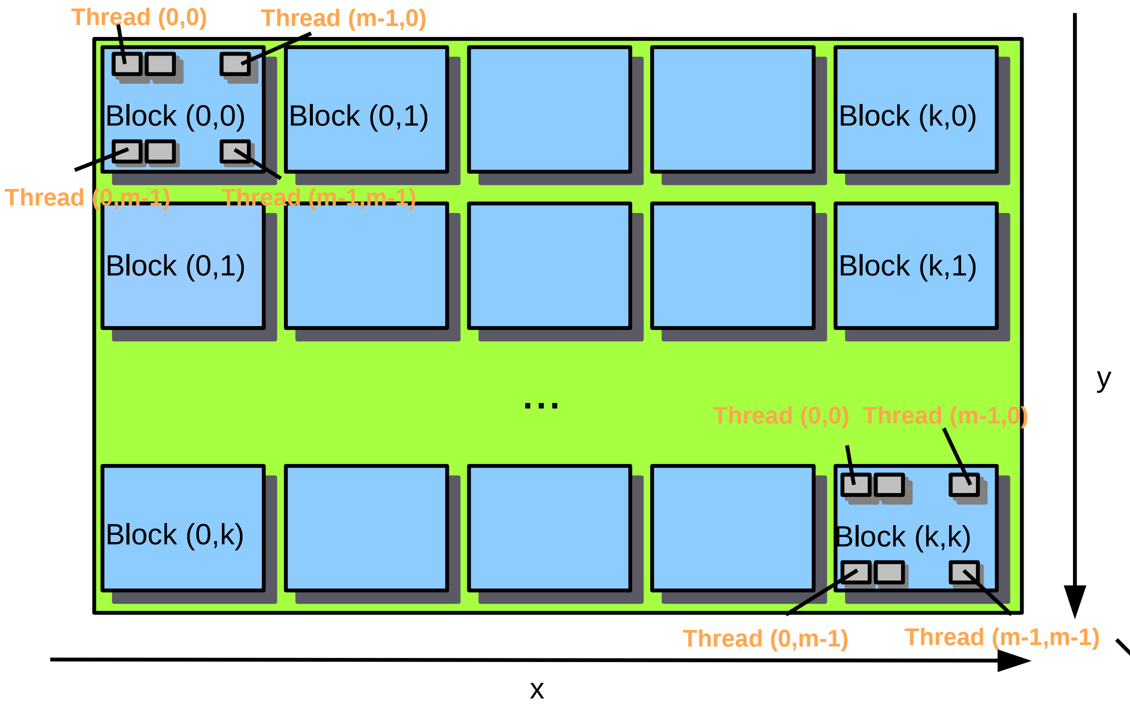


Result matrix C (n * n elements)

- Block dimensions are limited, hence several thread blocks will be needed
- Use 2d execution grid with

  k * k blocks

DISTRIBUTION OF WORK



Result matrix C (n * n elements)

- Use 2d execution grid with k * k blocks
- Use 2d thread blocks with fixed block size (m * m)
- k = n / m (n divisible by m)
- k = n / m + 1 (n not divisble by m)

block local thread index

JÜLICH Forschungszentrum

DEFINE DIMENSIONS OF THREAD BLOCK

DIM3 BLOCKDIM
`dim3 blockDim ( size_t blockDimX, size_t blockDimY, size_t blockDimZ )`

On Jureca (Tesla K80):

- Max. dim. of a block: 1024 x 1024 x 64
- Max. number of threads per block: 2048

Example:

```
// Create 3D thread block with 512 threads
dim3 blockDim(16, 16, 2);
```


JÜLICH Forschungszentrum

DEFINE DIMENSIONS OF GRID

DIM3 GRIDDIM

```
dim3 gridDim ( size_t blockDimX, size_t blockDimY, size_t blockDimZ )
```

On Jureca (Tesla K80):

- Max. dim. of a grid: (2147483647, 65535, 65535)

Example:

```
// Dimension of problem: nx * ny = 1000 * 1000
dim3 blockDim(16, 16) // Don't need to write z = 1
int gx = (nx % blockDim.x==0) ? nx / blockDim.x : nx / blockDim.x + 1
int gy = (ny % blockDim.y==0) ? ny / blockDim.y : ny / blockDim.y + 1
dim3 gridDim(gx, gy);
```

Watch out!

CALLING THE KERNEL

DEFINE DIMENSIONS OF THREAD BLOCK

```
dim3 blockDim ( size_t blockDimX, size_t blockDimY, size_t blockDimZ )
```

DEFINE DIMENSIONS OF EXECUTION GRID

```
dim3 gridDim ( size_t gridDimX, size_t gridDimY,  size_t gridDimZ )
```

LAUNCH THE KERNEL

```
kernel<<<dim3 gridDim, dim3 blockDim>>>([arg]*)
```

KERNEL (CUDA)

KERNEL FUNCTION

```
__global__ void mm_kernel(float* A, float* B, float* C, int n){
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < n && col < n) {
      for (int i = 0; i < n; ++i) {
        C[row * n + col] += A[row * n + i] * B[i * n + col];
      }
    }
}

mm_kernel <<< dimGrid, dimBlock >>> (d_a, d_b, d_c, n);
```

EXERCISE

Simple Cuda MM implementation

.../exercises/tasks/Cuda_MM_simple

LIMITING FACTOR

KERNEL FUNCTION

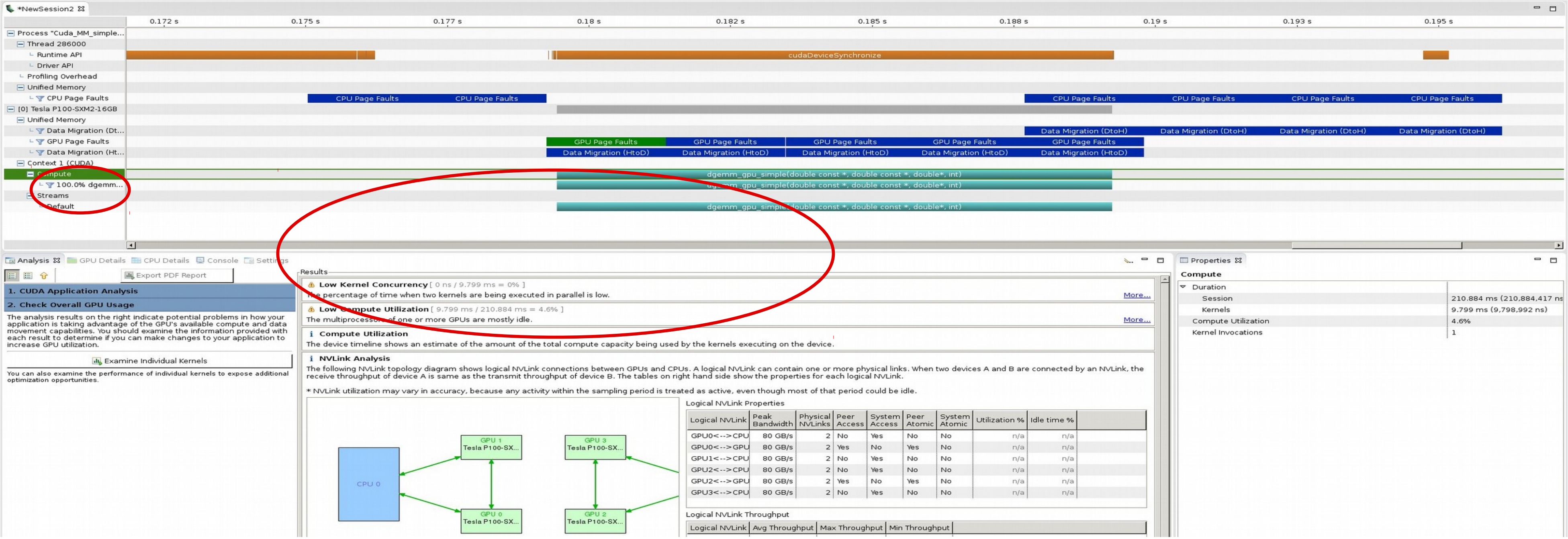
```
void mm_kernel ( float* A, float* B, float* C,int n )
{
    for (int k = 0; k < n; ++k){
        C[i * n + j] += A[i * n + k] * B[k * n + j];
    }
}
```

- One floating point operation per memory access
- One double: 8 bytes
- Limited global memory bandwidth
- **Check hints from Visual Profiler for further performance issues**

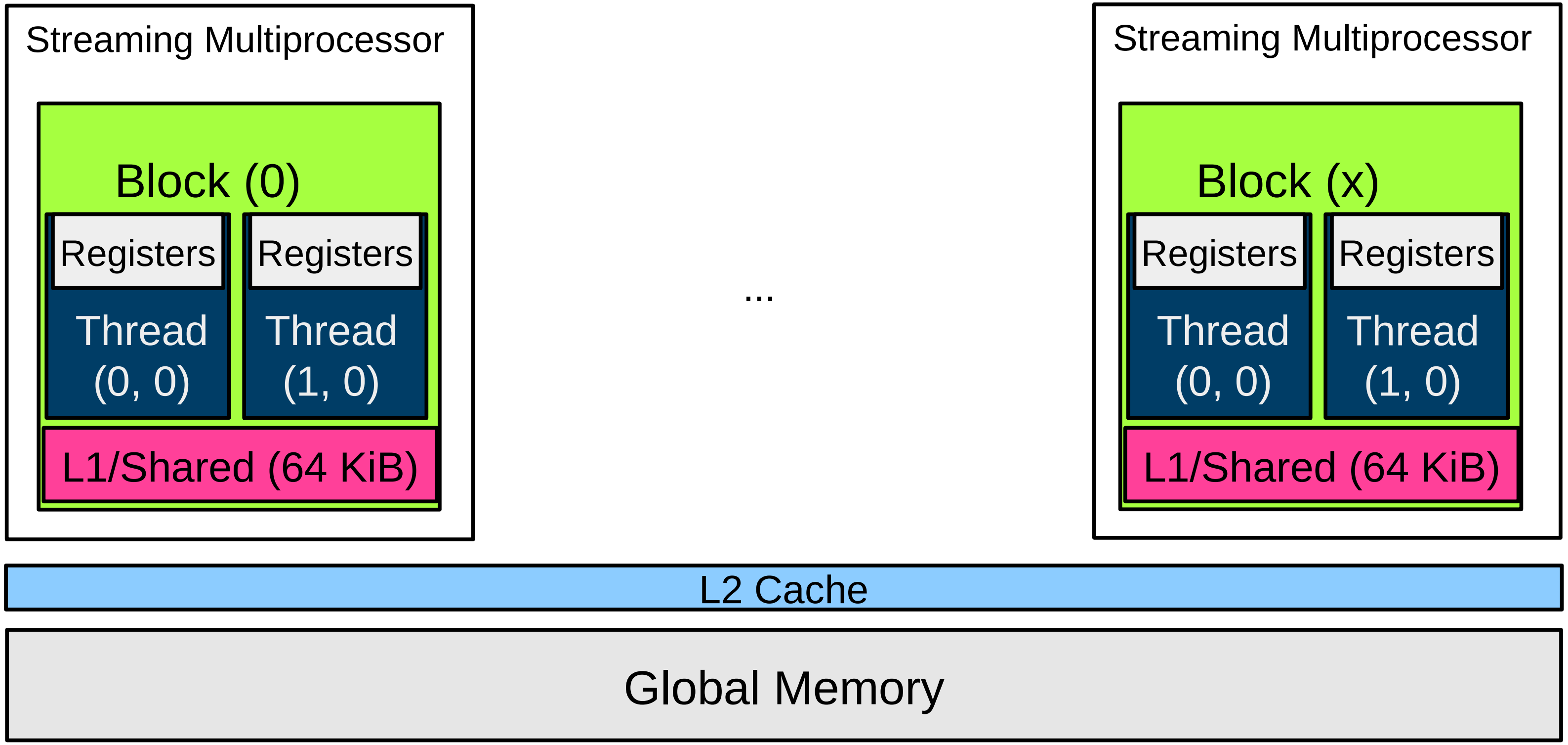
JÜLICH Forschungszentrum

LIMITING FACTOR

- Visual Profiler hints for simple MM



GPU MEMORY (SCHEMATICS)



USING SHARED MEMORY

ALLOCATE SHARED MEMORY

```
// allocate vector in shared memory
__shared__ float[size];

// can also define multi-dimensional arrays: BLOCK_SIZE is length (and width) of a thread block here
__shared__ float Msub[BLOCK_SIZE][BLOCK_SIZE];
```

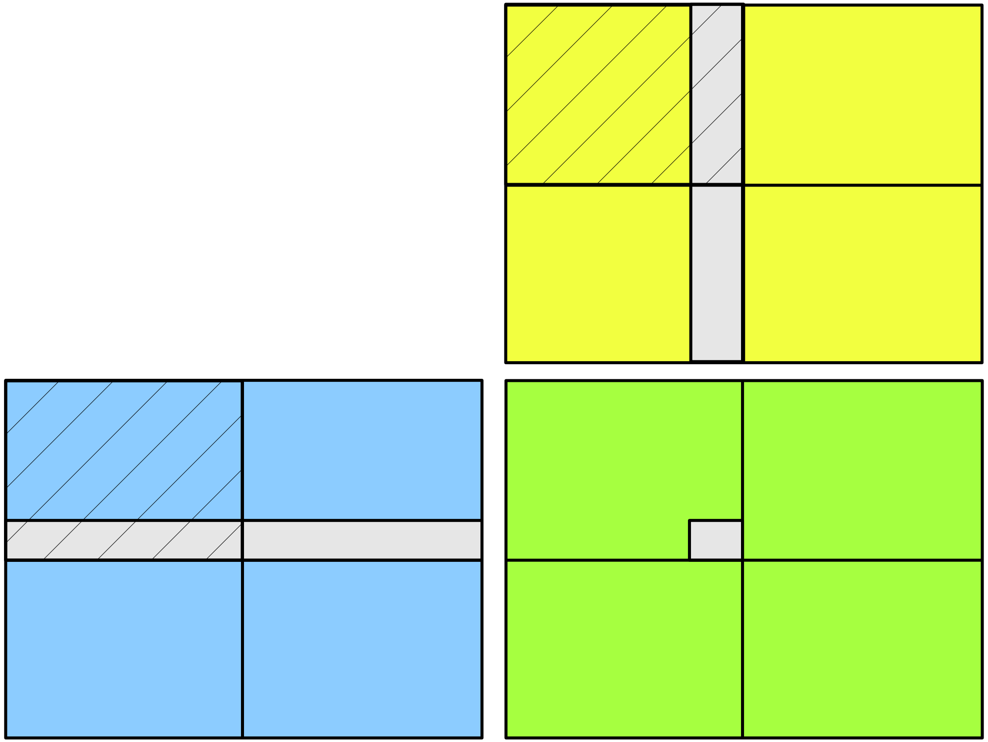
COPY DATA TO SHARED MEMORY

```
// fetch data from global to shared memory
Msub[threadIdx.y][threadIdx.x] = M[TidY * width + TidX];
```

SYNCHRONIZE THREADS

```
// ensure that all threads within a block had time to read / write data
__syncthreads();
```

MATRIX-MATRIX MULTIPLICATION WITH BLOCKS

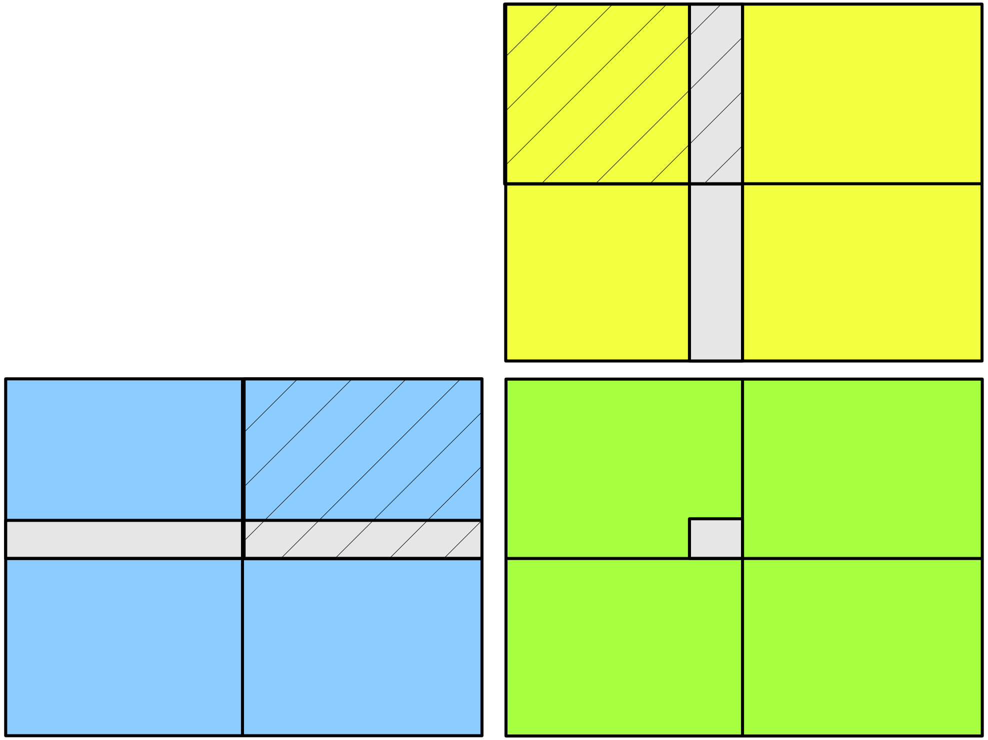


$$C_{kl} = \sum_{i=1}^{N} A_{ki} B_{il}$$

Split computation into partial computions

$$C_{kl} = \sum_{i=1}^{N/2} A_{ki} B_{il} + \sum_{i=N/2+1}^{N} A_{ki} B_{il}$$

MATRIX-MATRIX MULTIPLICATION WITH BLOCKS



$$C_{kl} = \sum_{i=1}^{N/2} A_{ki} B_{il} + \sum_{i=N/2+1}^{N} A_{ki} B_{il}$$

For each result element:

- Set element to zero
- For each pair of blocks
  - Copy data
  - Do partial sum
  - Add result of partial sum to element

JÜLICH Forschungszentrum

AN EXAMPLE

$$A=\begin{pmatrix}1&2&3&4\\4&1&2&3\\3&4&1&2\\2&3&4&1\end{pmatrix}\qquad B=\frac{1}{40}\begin{pmatrix}-9&11&1&1\\1&-9&11&1\\1&1&-9&11\\11&1&1&-9\end{pmatrix}\qquad C=AB$$

$$A=\begin{pmatrix}\overset{\mathrm{A}_{11}}{\begin{pmatrix}1&2\\4&1\end{pmatrix}}&\overset{\mathrm{A}_{12}}{\begin{pmatrix}3&4\\2&3\end{pmatrix}}\\\underset{\mathrm{A}_{21}}{\begin{pmatrix}3&4\\2&3\end{pmatrix}}&\underset{\mathrm{A}_{22}}{\begin{pmatrix}1&2\\4&1\end{pmatrix}}\end{pmatrix}\qquad B=\frac{1}{40}\begin{pmatrix}\overset{\mathrm{B}_{11}}{\begin{pmatrix}-9&11\\1&-9\end{pmatrix}}&\overset{\mathrm{B}_{12}}{\begin{pmatrix}1&1\\11&1\end{pmatrix}}\\\underset{\mathrm{B}_{21}}{\begin{pmatrix}1&1\\11&1\end{pmatrix}}&\underset{\mathrm{B}_{22}}{\begin{pmatrix}-9&11\\1&-9\end{pmatrix}}\end{pmatrix}\qquad C=\begin{pmatrix}C_{11}&C_{12}\\C_{21}&C_{22}\end{pmatrix}$$

$$C_{11}=A_{11}B_{11}+A_{12}B_{21}$$

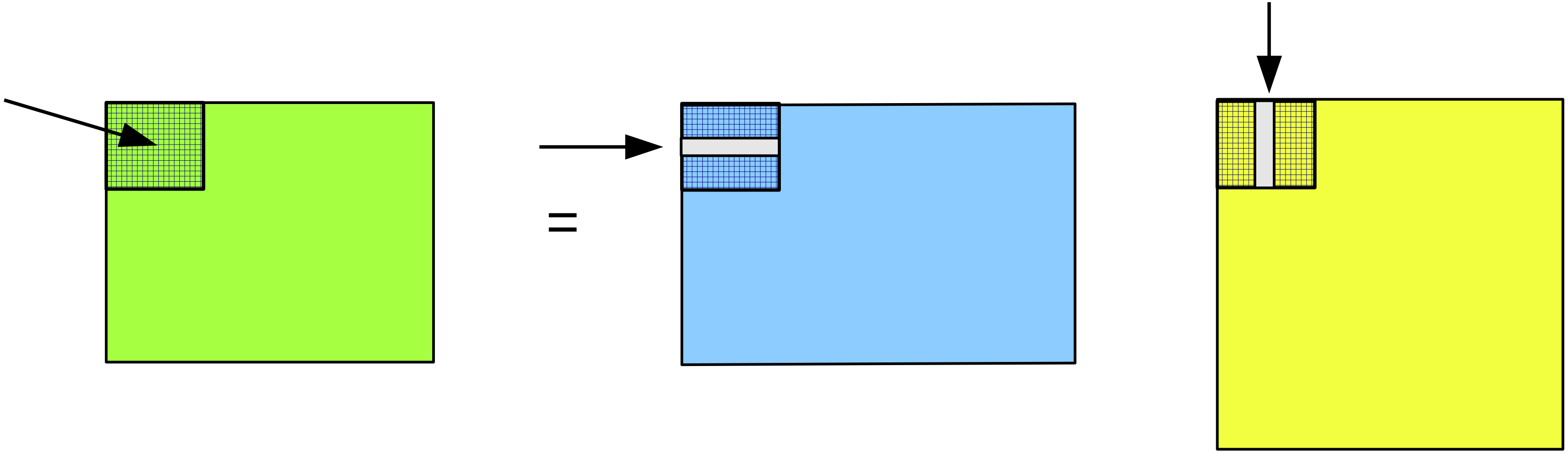
$$=\frac{1}{40}\begin{pmatrix}1&2\\4&1\end{pmatrix}\begin{pmatrix}-9&11\\1&-9\end{pmatrix}+\frac{1}{40}\begin{pmatrix}3&4\\2&3\end{pmatrix}\begin{pmatrix}1&1\\11&1\end{pmatrix}=\frac{1}{40}\begin{pmatrix}-9+2&11-18\\-36+1&44-9\end{pmatrix}+\frac{1}{40}\begin{pmatrix}3+44&3+4\\2+33&2+3\end{pmatrix}$$

$$=\frac{1}{40}\begin{pmatrix}-7&-7\\-35&35\end{pmatrix}+\frac{1}{40}\begin{pmatrix}47&7\\35&5\end{pmatrix}$$

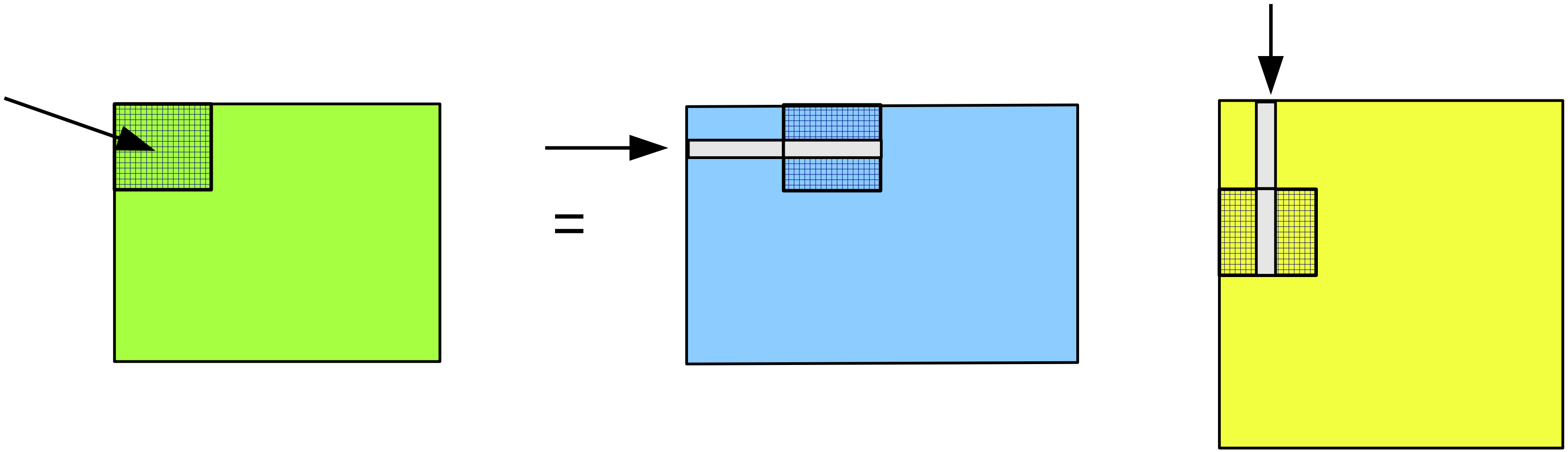
Do C_12, C_13, and C_14 the same way.

$$=\frac{1}{40}\begin{pmatrix}40&0\\0&40\end{pmatrix}=\begin{pmatrix}1&0\\0&1\end{pmatrix}$$

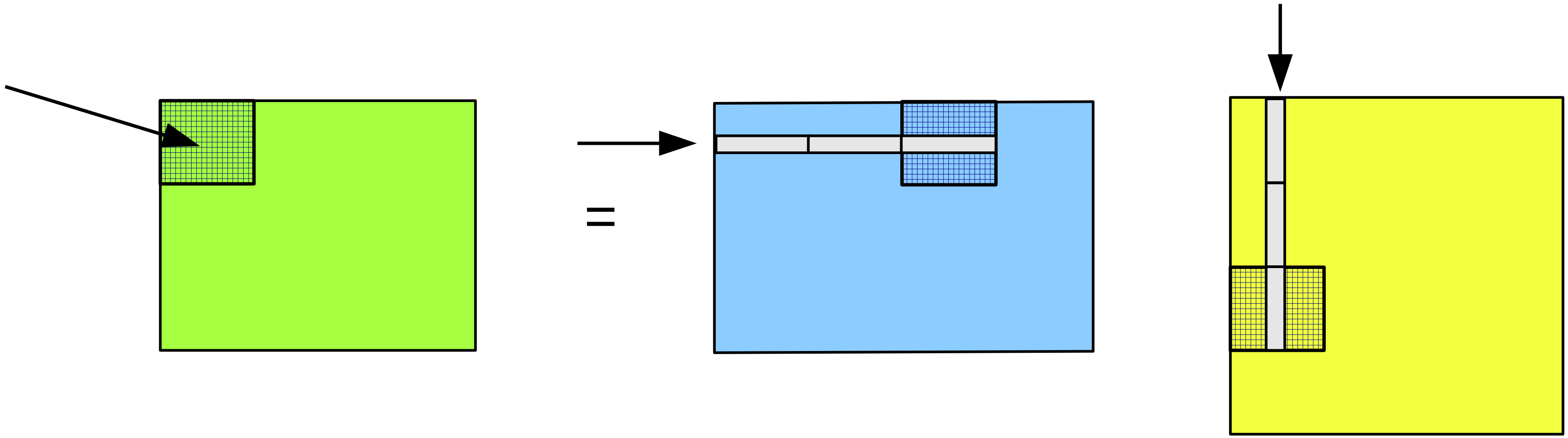
BLOCKWISE MATRIX-MATRIX MULTIPLICATION



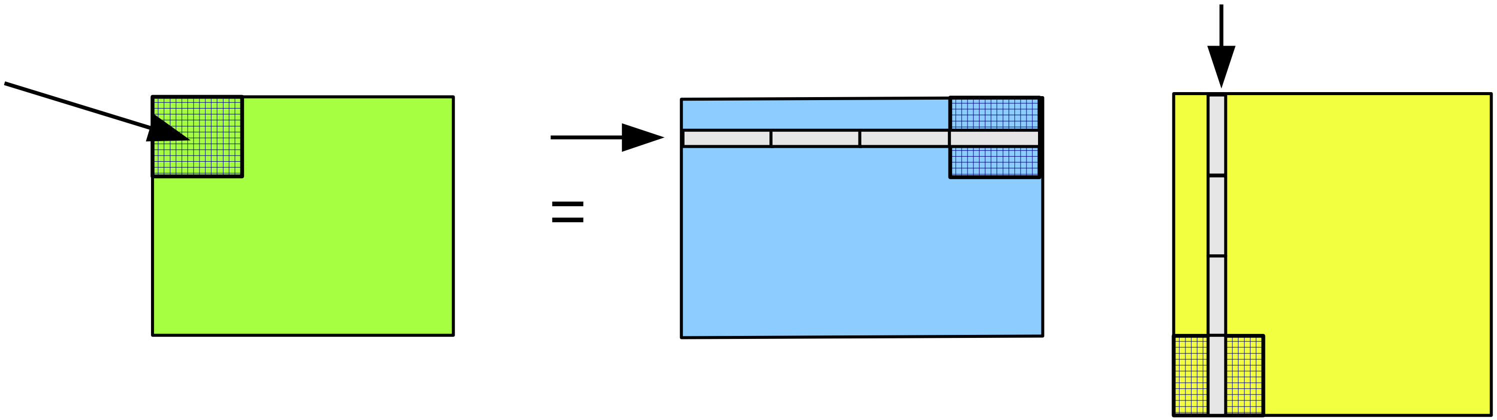
BLOCKWISE MATRIX-MATRIX MULTIPLICATION



BLOCKWISE MATRIX-MATRIX MULTIPLICATION



BLOCKWISE MATRIX-MATRIX MULTIPLICATION



BLOCK MATRIX-MULTIPLICATION APPROACH

Thread block loops over blocks in blue and yellow matrix:

- *Calculate upper left corner*
- *Load data into shared memory*
- *Do calculation (one thread is still responsible for an element)*
- *Add partial sum to result*

EXERCISE

Shared memory Cuda MM implementation

.../exercises/tasks/Cuda_MM_shared