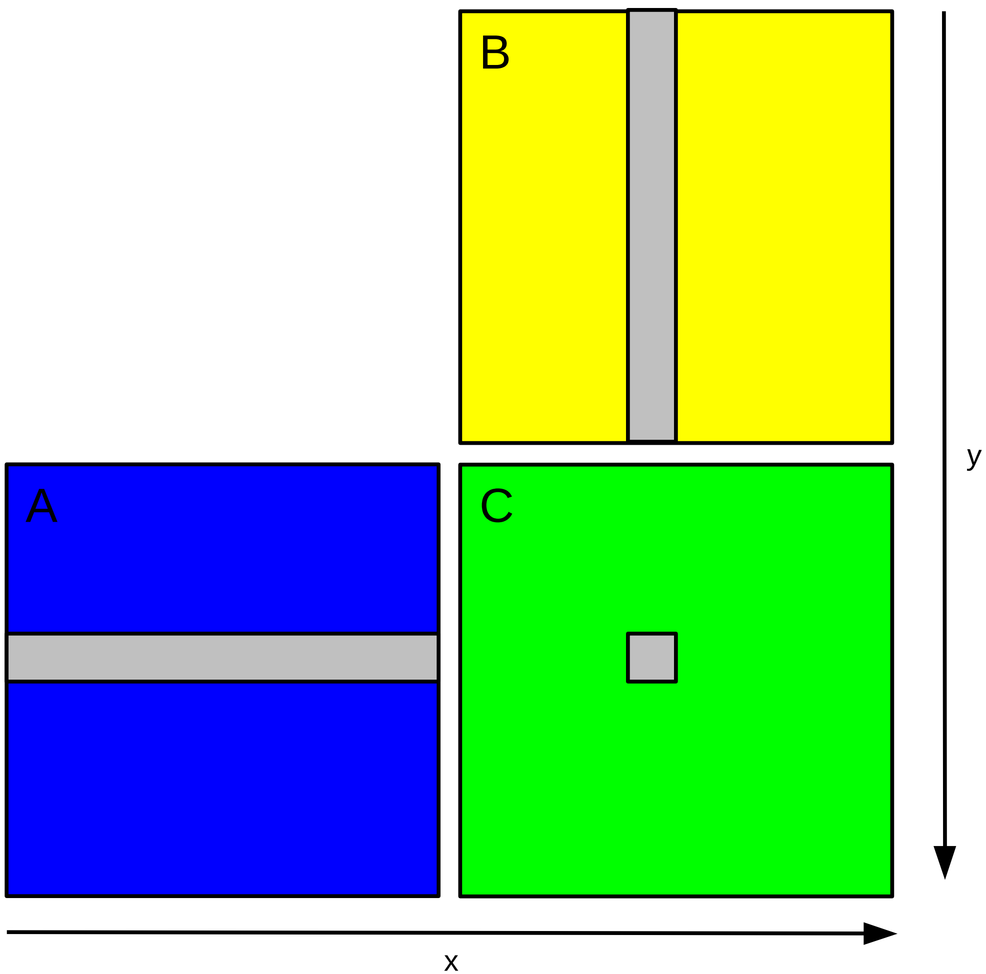
Mitglied der Helmholtz-Gemeinschaft

Matrix multiplication with Cuda

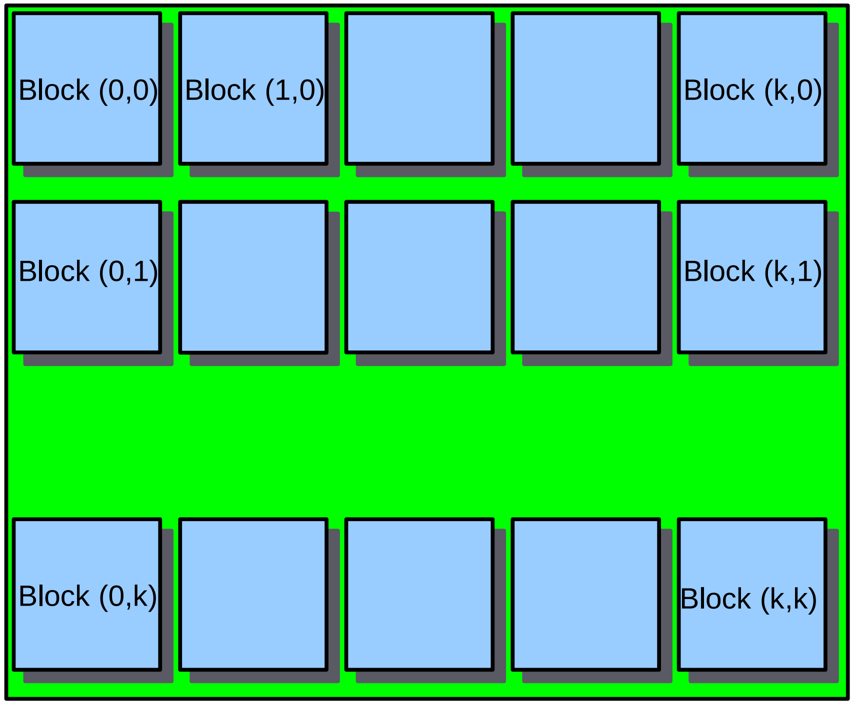
Jochen Kreutz (JSC)

Distribution of work



- Each thread computes one element of the result matrix C
- n * n threads will be needed
- 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

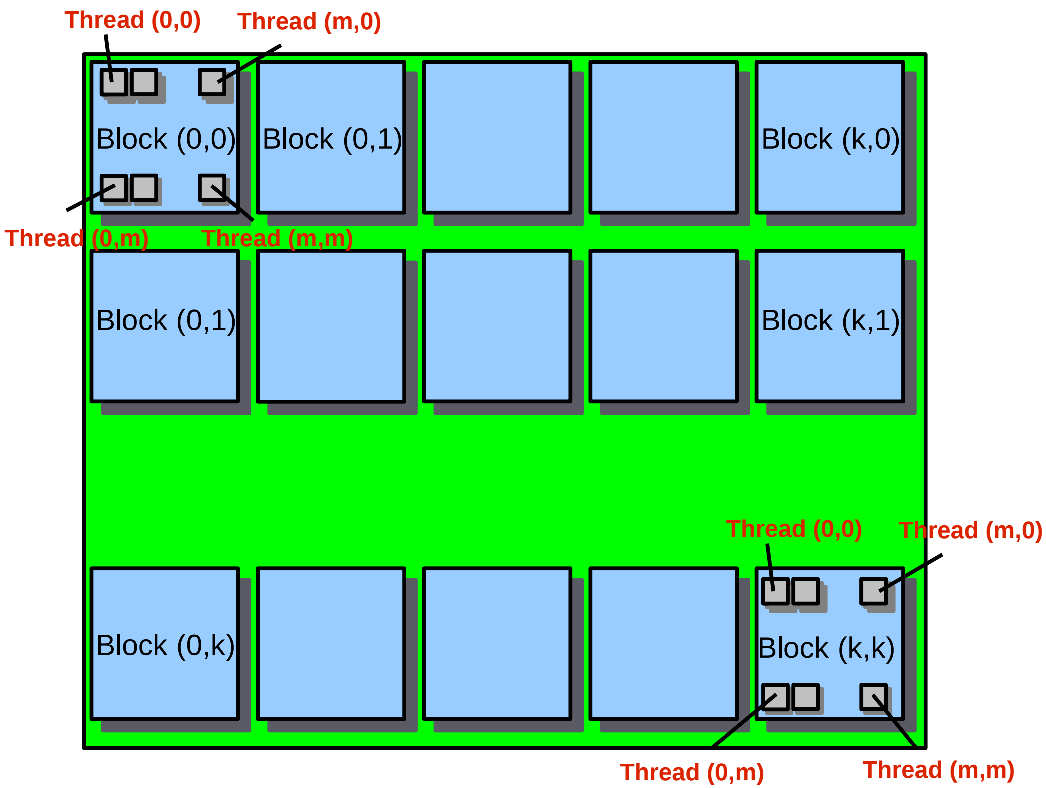


Result matrix C (n * n elements)

- 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 divisible by m)

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);
```

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

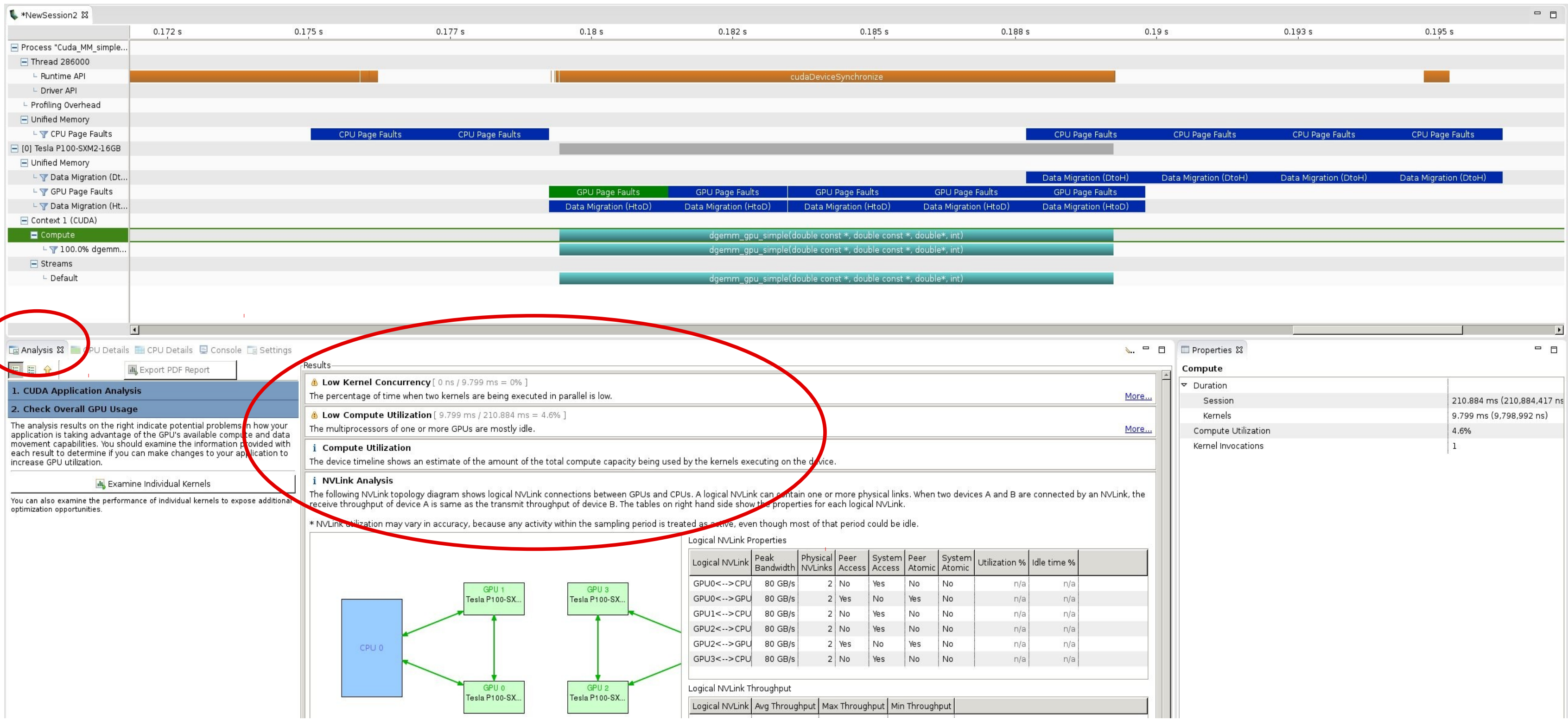
Kernel function

```
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**

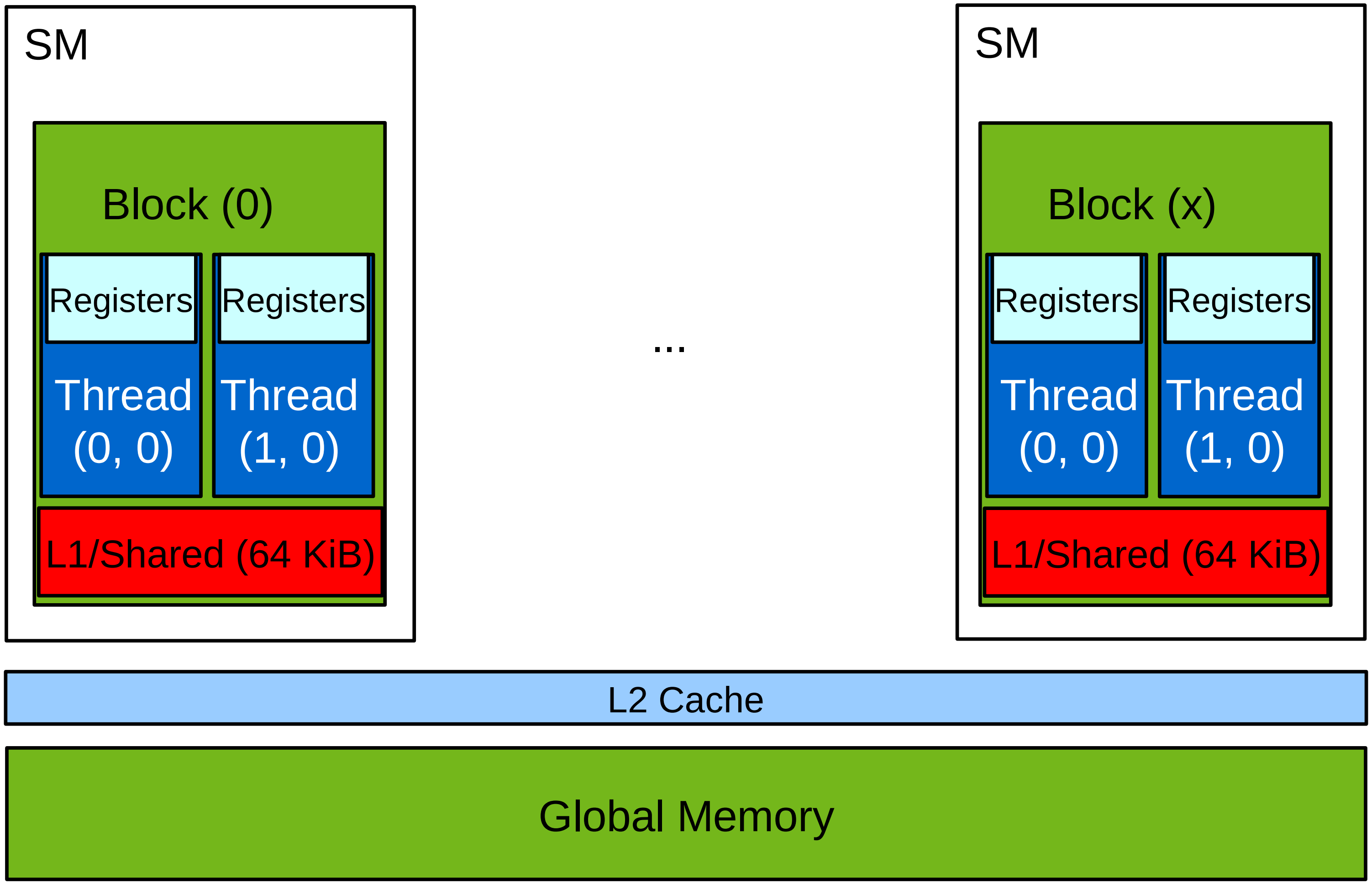
Limiting Factor



Logical NVLink	Peak Bandwidth	Physical NVLinks	Peer Access	System Access	Peer Atomic	System Atomic	Utilization %	Idle time %
GPU0<-->CPU	80 GB/s	2	No	Yes	No	No	n/a	n/a
GPU0<-->GPU	80 GB/s	2	Yes	No	Yes	No	n/a	n/a
GPU1<-->CPU	80 GB/s	2	No	Yes	No	No	n/a	n/a
GPU2<-->CPU	80 GB/s	2	No	Yes	No	No	n/a	n/a
GPU2<-->GPU	80 GB/s	2	Yes	No	Yes	No	n/a	n/a
GPU3<-->CPU	80 GB/s	2	No	Yes	No	No	n/a	n/a

- **Check hints from Visual Profiler for further performance issues**

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

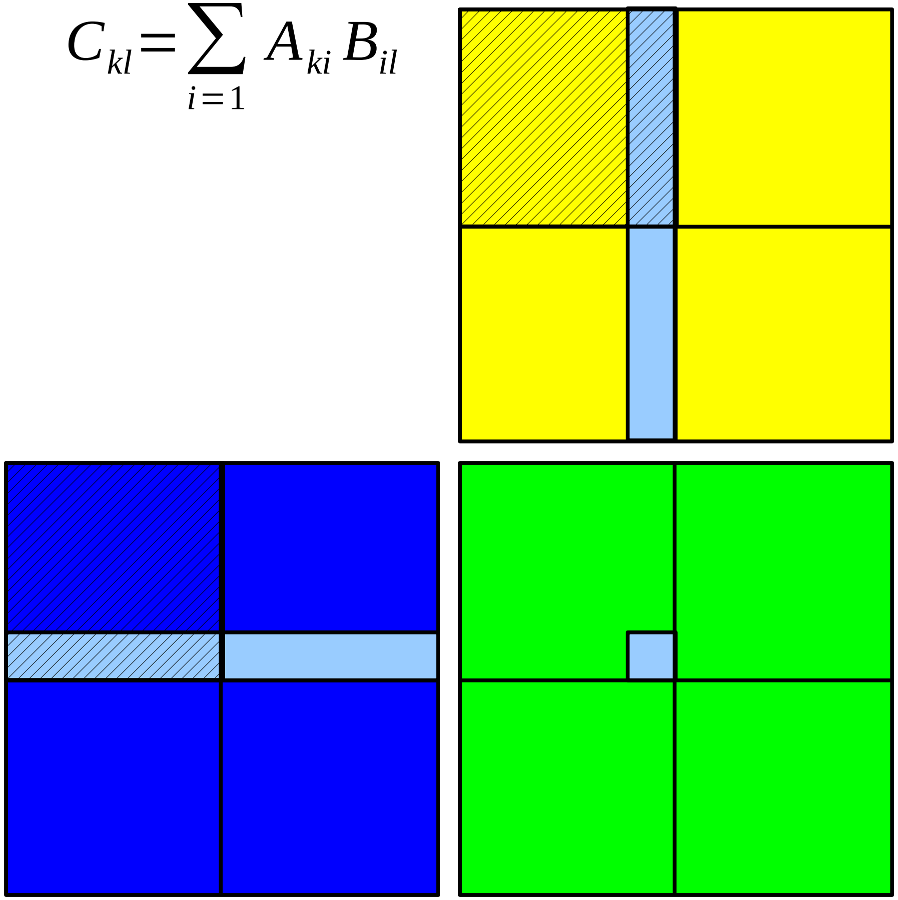
```
// fetch data from global to shared memory
Msub[threadIdx.y][threadIdx.x] = M[TidY * width + TidX];
```

Synchronize threads

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

Matrix-matrix multiplication with blocks

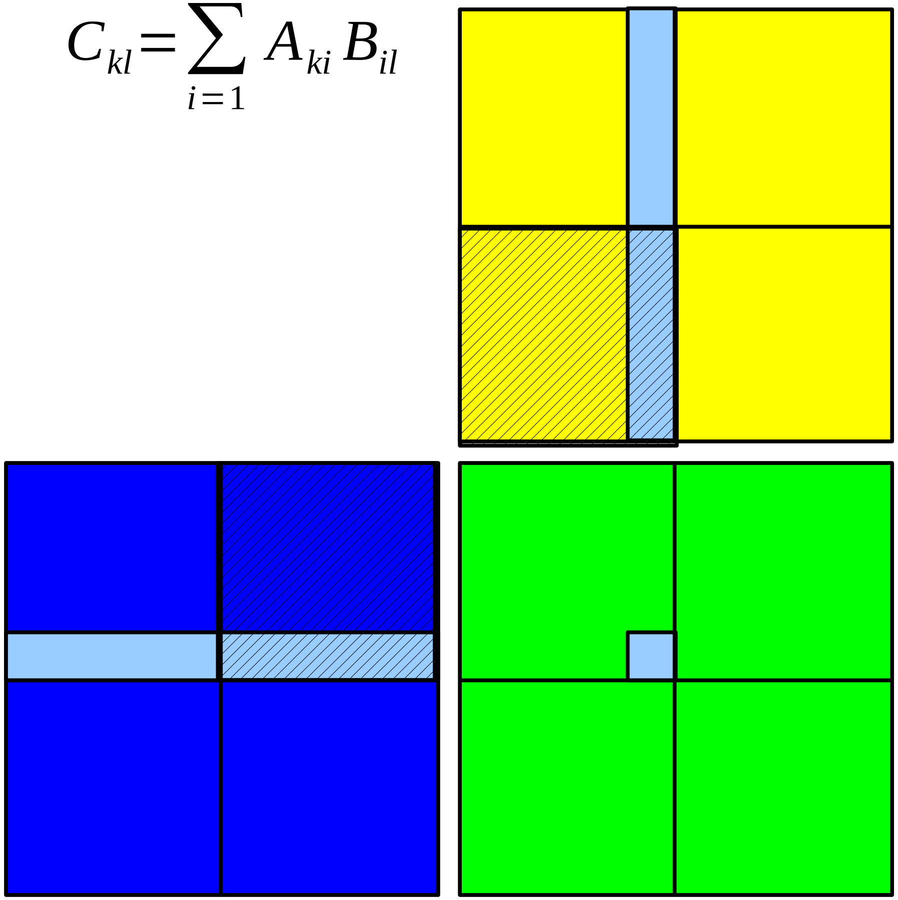
$$C_{kl}=\sum_{i=1}^{N} A_{ki} B_{il}$$



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

Matrix-matrix multiplication with blocks

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



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

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

For each element

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

An Example

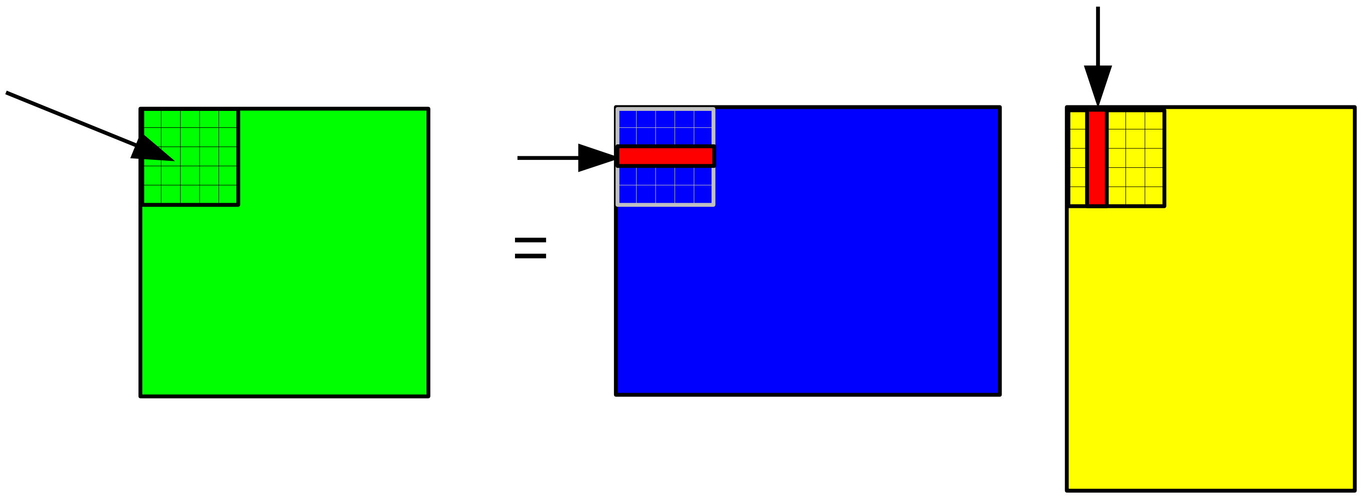
$$A=\begin{pmatrix}1&2&3&4\\4&1&2&3\\3&4&1&2\\2&3&4&1\end{pmatrix}\qquad B=\frac{1}{40}\begin{pmatrix}-9&11&1&1\\1&-9&11&1\\1&1&-9&11\\11&1&1&-9\end{pmatrix}\qquad C=AB$$

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

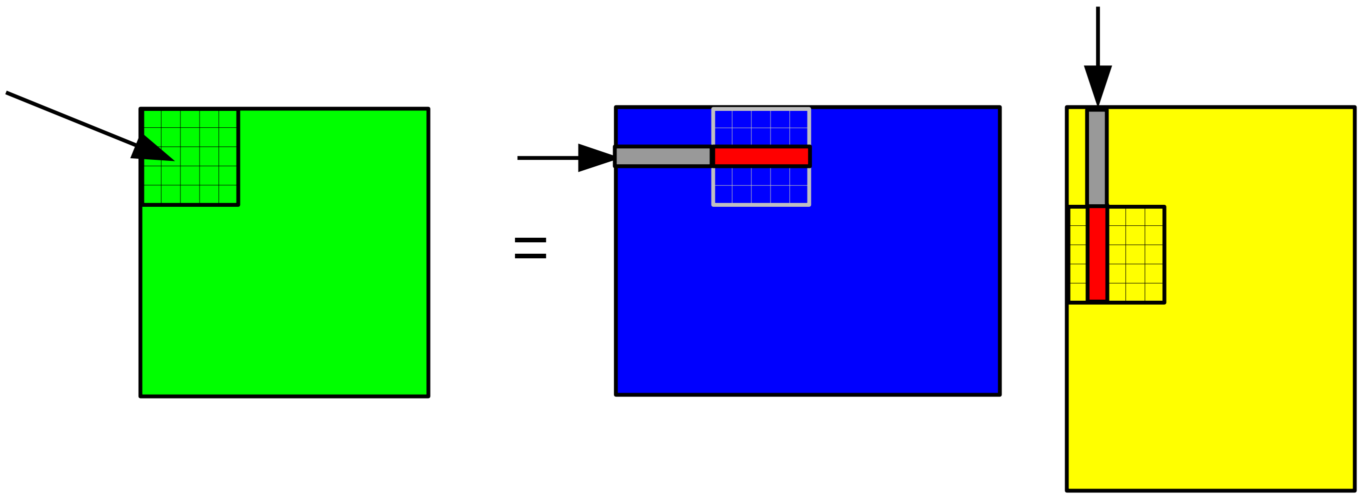
$$\begin{aligned}C_{11}&=A_{11}B_{11}+A_{12}B_{21}\\&=\frac{1}{40}\begin{pmatrix}1&2\\4&1\end{pmatrix}\begin{pmatrix}-9&11\\1&-9\end{pmatrix}+\frac{1}{40}\begin{pmatrix}3&4\\2&3\end{pmatrix}\begin{pmatrix}1&1\\11&1\end{pmatrix}\\&=\frac{1}{40}\begin{pmatrix}-9+2&11-18\\-36+1&44-9\end{pmatrix}+\frac{1}{40}\begin{pmatrix}3+44&3+4\\2+33&2+3\end{pmatrix}\\&=\frac{1}{40}\begin{pmatrix}-7&-7\\-35&35\end{pmatrix}+\frac{1}{40}\begin{pmatrix}47&7\\35&5\end{pmatrix}\\&=\frac{1}{40}\begin{pmatrix}40&0\\0&40\end{pmatrix}=\begin{pmatrix}1&0\\0&1\end{pmatrix}\end{aligned}$$

Do C_{12}, C_{13}, and C_{14} the same way.

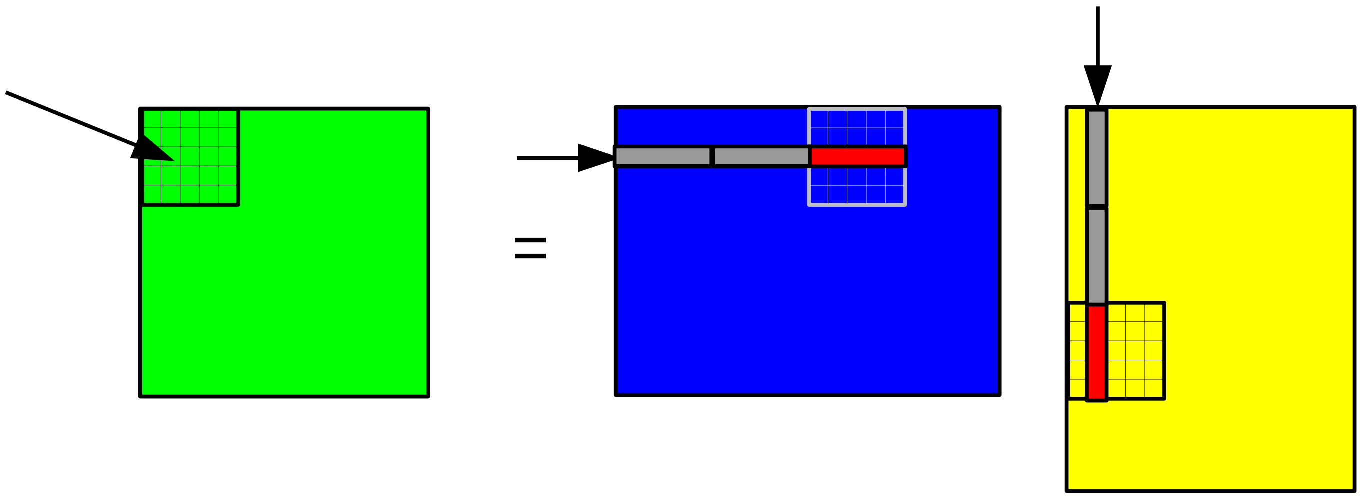
Blockwise Matrix-Matrix Multiplication



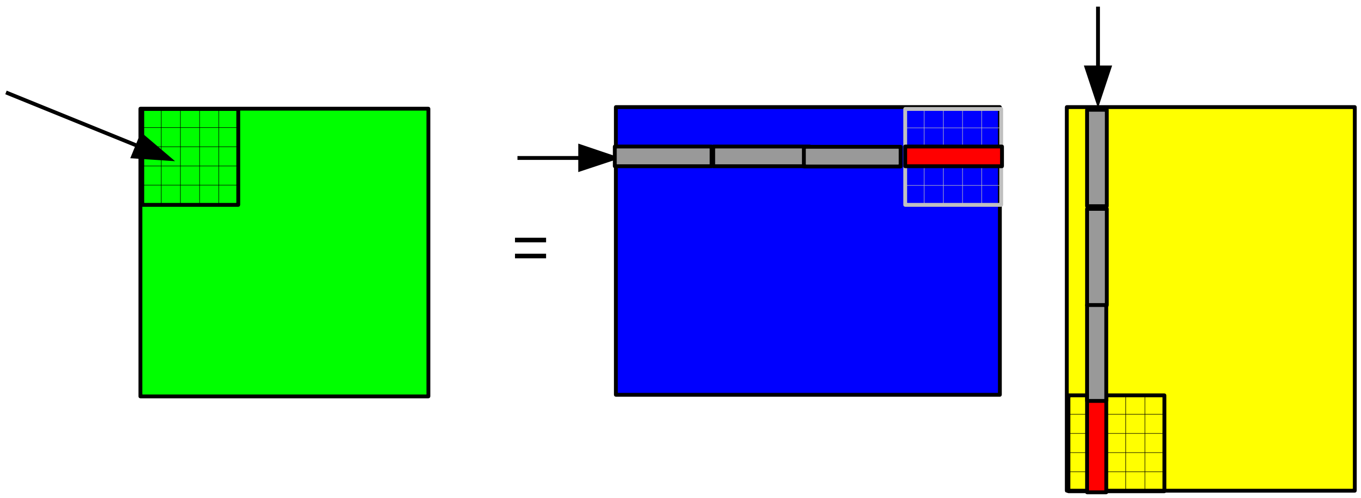
Blockwise Matrix-Matrix Multiplication



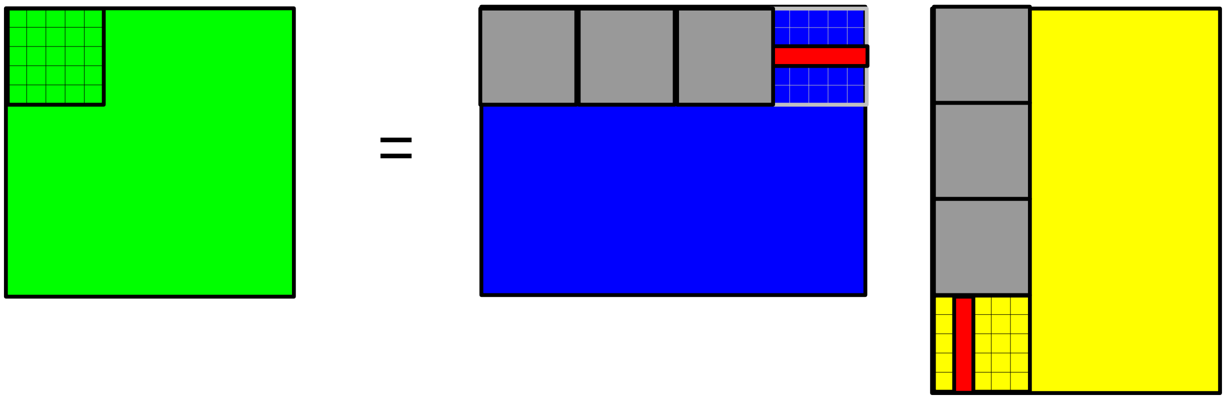
Blockwise Matrix-Matrix Multiplication



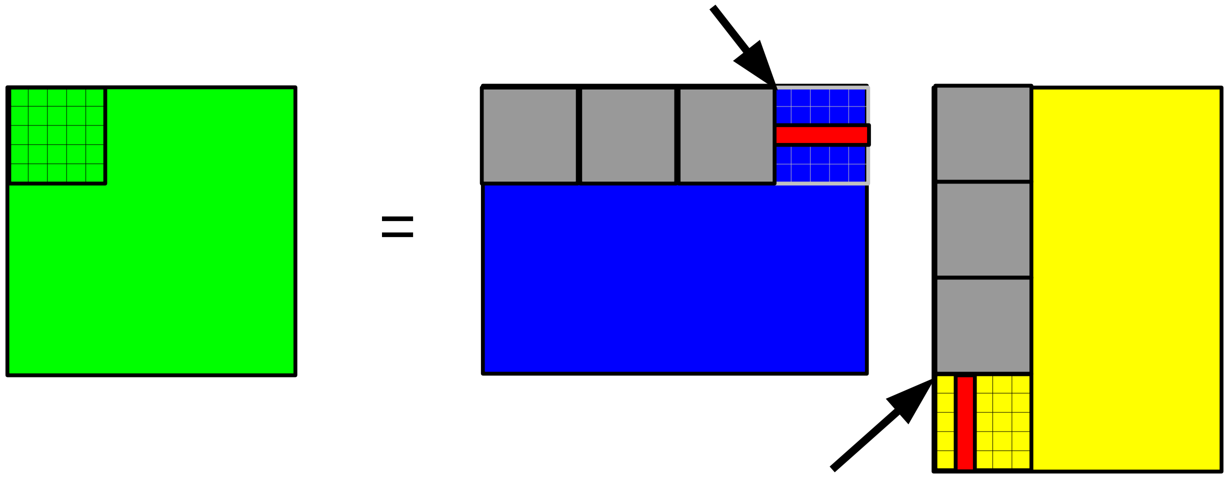
Blockwise Matrix-Matrix Multiplication



Blockwise Matrix-Matrix Multiplication



Blockwise Matrix-Matrix Multiplication



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