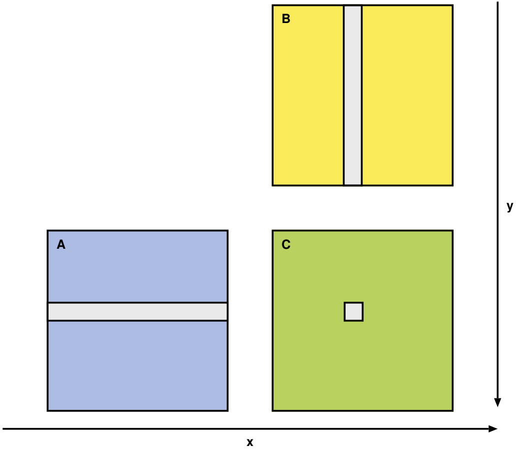
GPU PROGRAMMING WITH CUDA

Matrix Multiplication

April 09, 2024 | Carolin Penke, Kaveh Haghighi Mood, Jochen Kreutz | JSC

CUDA MATRIX MULTIPLICATION

Distribution of work

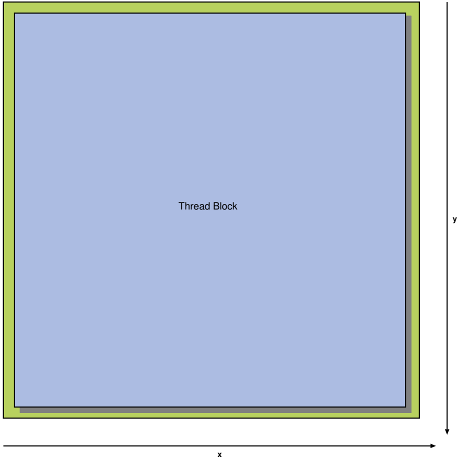


$$C_{row,col} = \sum_{i=1}^{n} A_{row,i} * B_{i,col}$$

- $n \times n$ **threads** needed for matrix C of size $n \times n$
- Thread `(x,y)` computes result element $C_{y,x}$ using row y of A and column x of B

CUDA MATRIX MULTIPLICATION

Execution Grid Layout

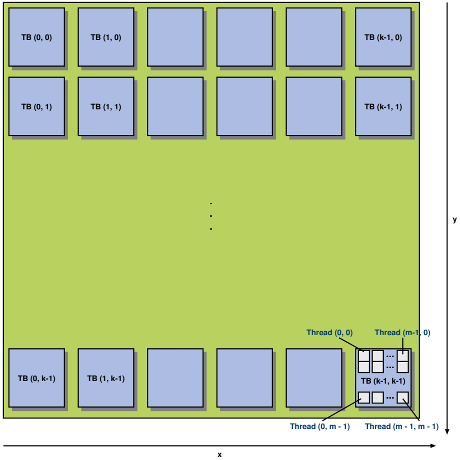


- Naive idea: One big thread **block** to cover all result elements
- Using only one block decreases performance (due to reduced device occupancy)
- Blocks are limited in size

→ Several blocks needed to cover the full matrix C

CUDA MATRIX MULTIPLICATION

Execution Grid Layout

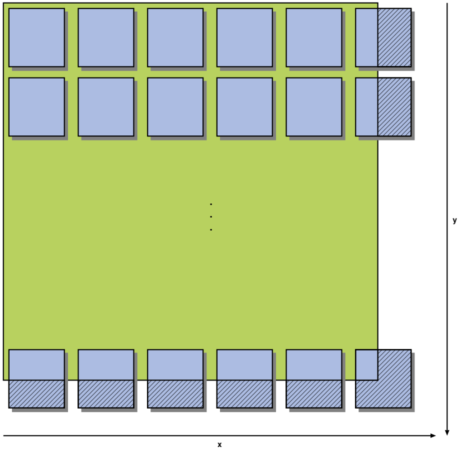


- Cover C of size $n \times n$ with 2D kernel execution grid with $k \times k$ thread blocks (TB).
- Fixed block size $m \times m$.
  - Optimal value for m architecture-dependant.

$$k = \begin{cases} n/m & \text{if } n \text{ divisible by } k \\ n/m + 1 & \text{else} \end{cases}$$

CUDA MATRIX MULTIPLICATION

Execution Grid Layout

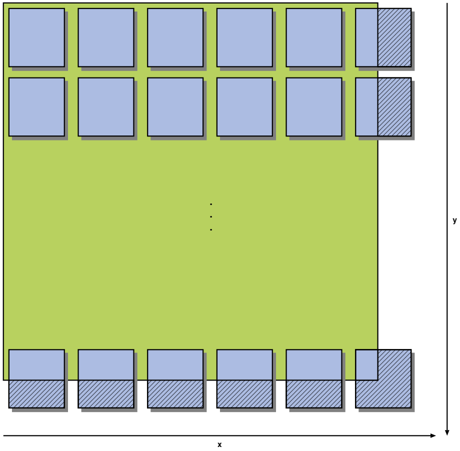


- Cover C of size $n \times n$ with 2D kernel execution grid with $k \times k$ thread blocks (TB).
- Fixed block size $m \times m$.
  - Optimal value for m architecture-dependant.

$$k = \begin{cases} n/m & \text{if } n \text{ divisible by } k \\ n/m + 1 & \text{else} \end{cases}$$

CUDA MATRIX MULTIPLICATION

Execution Grid Layout



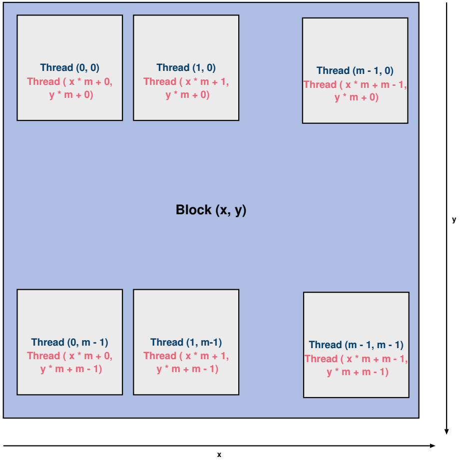
- Cover C of size $n \times n$ with 2D kernel execution grid with $k \times k$ thread blocks (TB).
- Fixed block size $m \times m$.
  - Optimal value for m architecture-dependant.

$$k = \begin{cases} n/m & \text{if } n \text{ divisible by } k \\ n/m + 1 & \text{else} \end{cases}$$

- Check if threads are out of bounds.

CUDA MATRIX MULTIPLICATION

Execution grid layout



- Threads can be addressed via local index (block internal) and global index (full grid)
- Keywords in kernel to get thread information:

```
blockIdx.x    blockIdx.y    blockIdx.z
threadIdx.x   threadIdx.y   threadIdx.z
blockDim.x    blockDim.y    blockDim.z
gridDim.x     gridDim.y     gridDim.z
```

RECAP: GRID AND BLOCK SIZES

See day 1 material

Define block sizes, grid sizes and launch kernel from host:

Example workflow

```
int Nx = 1000, Ny = 1000;
dim3 blockDim(16, 16); //store 2D configuration in blockDim
```

RECAP: GRID AND BLOCK SIZES

See day 1 material

Define block sizes, grid sizes and launch kernel from host:

Example workflow

```
int Nx = 1000, Ny = 1000;
dim3 blockDim(16, 16); //store 2D configuration in blockDim
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);  //store 2D configuration in gridDim
```

RECAP: GRID AND BLOCK SIZES

See day 1 material

Define block sizes, grid sizes and launch kernel from host:

Example workflow

```
int Nx = 1000, Ny = 1000;
dim3 blockDim(16, 16); //store 2D configuration in blockDim
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);  //store 2D configuration in gridDim
kernel<<<gridDim, blockDim>>>(); //launch kernel
```

CUDA MATRIX MULTIPLICATION

Kernel: Matrix Multiplication

```
__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 <<< gridDim, blockDim >>> (a, b, c, n);
```

EXERCISE

Simple matrix multiplication with Cuda

Detailed instructions

.../exercises/tasks/Cuda_MM_simple/Instructions.ipynb

1. Implement CUDA Matrix Multiplication

```
C[row*n + col] += A[row*n + i] * B[i*n + col];
```

2. Instead of writing to array `C`, write to local variable `cvalue`, write to `C` later.

```
cvalue += A[row*n + i] * B[i*n + col];
```

PERFORMANCE CONSIDERATIONS

Measured numbers

JUWELS Cluster: 1 x V100 (theoretical peak: 7 TFlops/s DP)
JUWELS Booster: 1 x A100 (theoretical peak: 9.7 TFlops/s DP, 19.5 with TC)

matrix size	64	1024	10240	64	1024	10240
	JW Cluster [GFlops/s)]			**JW Booster** [GFlops/s)]		
with `cvalue`	1.2	319	1146	1.1	286.2	1587.1
direct write	1.02	196	391	0.9	198.3	562.2

JÜLICH Forschungszentrum

PERFORMANCE CONSIDERATIONS

Profiler hints for simple matrix multiplication

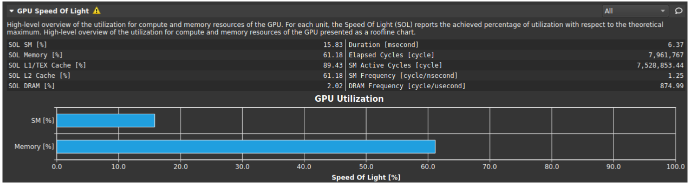


Figure: Kernel profiling in Nsight Compute

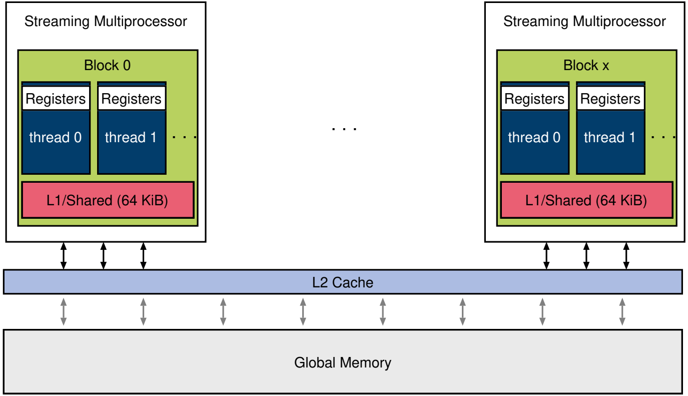
- NVIDIA Nsight Systems gives overview timeline.
- NVIDIA Nsight Compute analyzes kernels.

→ useful hints, hotspots, potential performance issues

- indicates very low compute utilization
- `dgemm` kernel is memory-bound, waits for data

PERFORMANCE CONSIDERATIONS

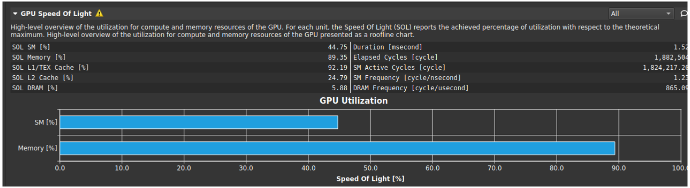
GPU memory layout



- array `C` located in global memory
- `cvalue` located in registers on SM: faster write operations

PERFORMANCE CONSIDERATIONS

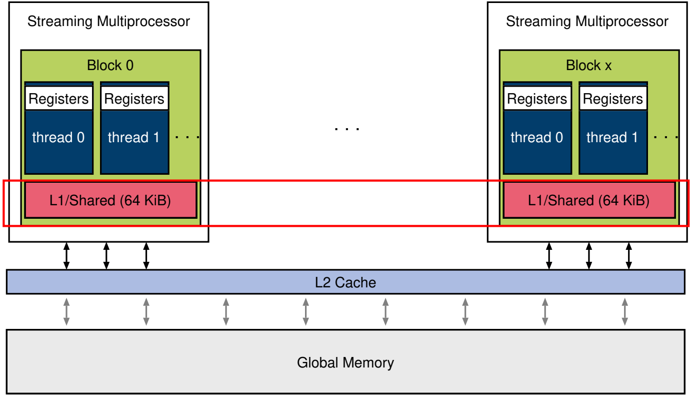
Profiler hints for simple matrix multiplication



- Using `cvalue` reduces the access to the global memory

PERFORMANCE CONSIDERATIONS

GPU memory layout (schematics)



How to make use of Shared Memory?

- matrix array `C` located in global memory
- `cvalue` located in registers on SM: faster write operations

SHARED MEMORY

How to use inside your kernels

Allocate shared memory

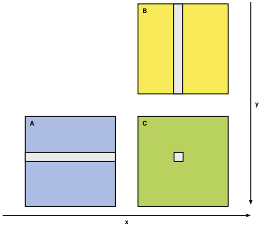
```
// allocate vector in shared memory
__shared__ float[size];
// allocate 2D array
__shared__ float Msub[BLOCK_SIZE][BLOCK_SIZE];
```

Copy data from globalto shared memory

```
Msub[threadIdx.y][threadIdx.x] = M[threadIdx.y * width + threadIdx.x]
```

Remember: only shared between threads within the same thread block !

SHARED MEMORY



Shared memory is limited, the whole matrices do not fit in all at once.

How can we rewrite Matrix Multiplication, s.t. data in shared memory is reused efficiently?

Solution: Tiling (very common in all matrix-based algorithms)

BLOCK MATRIX EXAMPLE

2×2 blocks

A matrix can be divided into blocks:

$$A = \left[\begin{array}{cc|cc} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ \hline 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{array}\right] = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

BLOCK MATRIX EXAMPLE

2×2 blocks

A matrix can be divided into blocks:

$$A = \left[\begin{array}{cc|cc} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ \hline 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{array}\right] = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

If block sizes align, matrix multiplication can be rewritten in block form:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

BLOCK MATRIX EXAMPLE

2×2 blocks

A matrix can be divided into blocks:

$$A = \left[\begin{array}{cc|cc} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ \hline 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{array}\right] = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

If block sizes align, matrix multiplication can be rewritten in block form:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

BLOCK MATRIX EXAMPLE

2×2 blocks

A matrix can be divided into blocks:

$$A = \left[\begin{array}{cc|cc} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ \hline 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{array}\right] = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

If block sizes align, matrix multiplication can be rewritten in block form:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

BLOCK MATRIX EXAMPLE

2×2 blocks

A matrix can be divided into blocks:

$$A = \left[\begin{array}{cc|cc} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ \hline 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{array}\right] = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

If block sizes align, matrix multiplication can be rewritten in block form:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

BLOCK MATRIX EXAMPLE

2×2 blocks

A matrix can be divided into blocks:

$$A = \left[\begin{array}{cc|cc} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ \hline 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{array}\right] = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

If block sizes align, matrix multiplication can be rewritten in block form:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

BLOCK MATRIX EXAMPLE

2×2 blocks

A matrix can be divided into blocks:

$$A = \left[\begin{array}{cc|cc} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ \hline 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{array}\right] = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

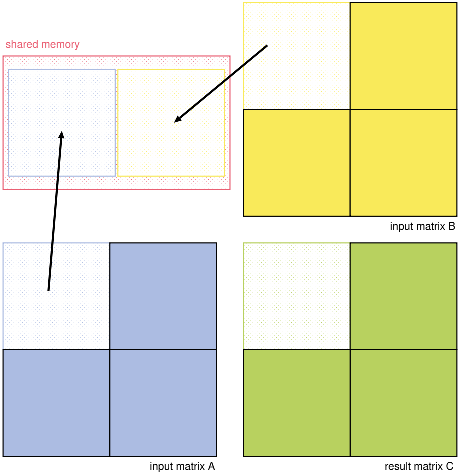
If block sizes align, matrix multiplication can be rewritten in block form:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

The whole matrices do not fit into shared memory, but we tile the matrix so that blocks do!

BLOCK MATRIX EXAMPLE

2×2 blocks, using shared memory



We map: CUDA Thread Block = Matrix Block.
One block computes

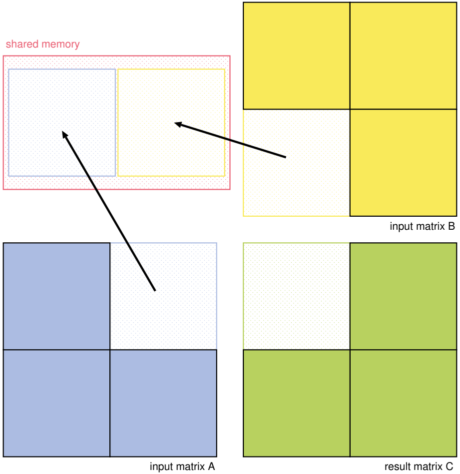
$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

Implementation:

1. Load A_{11}, B_{11} into shared memory
2. $C_{11} \leftarrow A_{11}B_{11}$

BLOCK MATRIX EXAMPLE

2×2 blocks, using shared memory



We map: CUDA Thread Block = Matrix Block.
One block computes

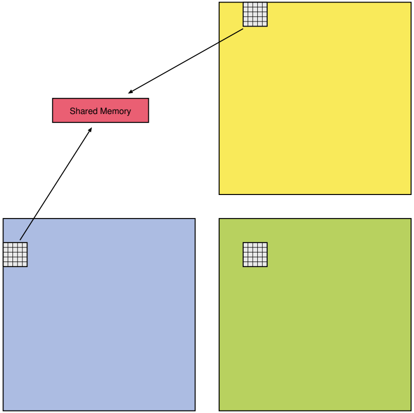
$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

Implementation:

1. Load A_{11}, B_{11} into shared memory
2. $C_{11} \leftarrow A_{11}B_{11}$
3. Load A_{12}, B_{21} into shared memory
4. $C_{11} \leftarrow C_{11} + A_{11}B_{21}$

BLOCK MATRIX MULTIPLICATION

Workflow, $k \times k$ blocks



Each thread (global index (x, y), local index (s, t) in block (u, v)) does:

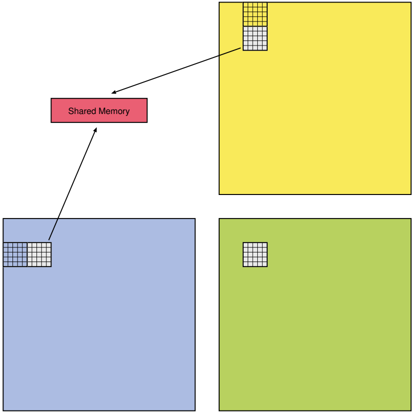
$C_{y,x} \leftarrow 0$
for $l = 1$ **to** k **do**
Copy input data A_{vi}, B_{iu} to shared memory (one element per thread)

Compute value (t, s) in $A_{vi}B_{iu}$.
Add this value to $C_{y,x}$

end for

BLOCK MATRIX MULTIPLICATION

Workflow, $k \times k$ blocks



Each thread (global index (x, y), local index (s, t) in block (u, v)) does:

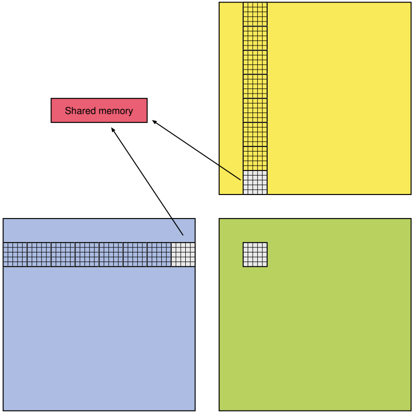
$C_{y,x} \leftarrow 0$
for $l = 1$ **to** k **do**
 Copy input data A_{vl}, B_{lu} to shared memory (one element per thread)

 Compute value (t, s) in $A_{vl}B_{lu}$.
 Add this value to $C_{y,x}$

end for

BLOCK MATRIX MULTIPLICATION

Workflow, $k \times k$ blocks



Each thread (global index (x, y), local index (s, t) in block (u, v)) does:

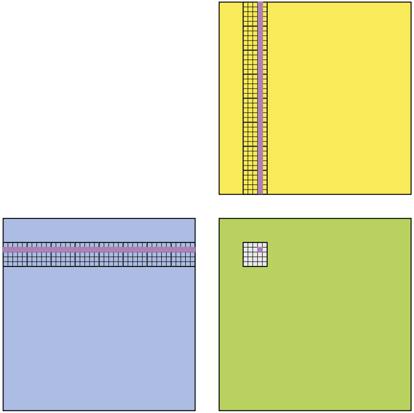
$C_{y,x} \leftarrow 0$
for $l = 1$ **to** k **do**
  Copy input data A_{vi}, B_{iu} to shared memory (one element per thread)

  Compute value (t, s) in $A_{vi}B_{iu}$.
  Add this value to $C_{y,x}$

end for

BLOCK MATRIX MULTIPLICATION

Workflow, $k \times k$ blocks



Each thread (global index (x, y), local index (s, t) in block (u, v)) does:

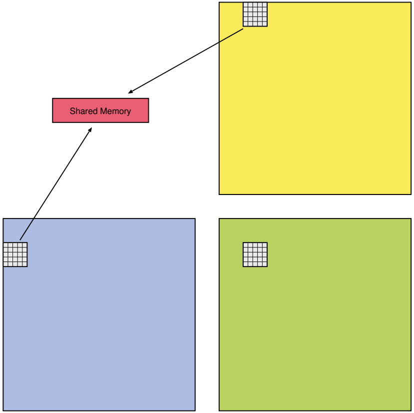
$C_{y,x} \leftarrow 0$
for $l = 1$ **to** k **do**
  Copy input data A_{vi}, B_{iu} to shared memory (one element per thread)

  Compute value (t, s) in $A_{vi}B_{iu}$.
  Add this value to $C_{y,x}$

end for

BLOCK MATRIX MULTIPLICATION

Thread synchronization



Thread synchronization

- Threads within a block may not be completely in synch.

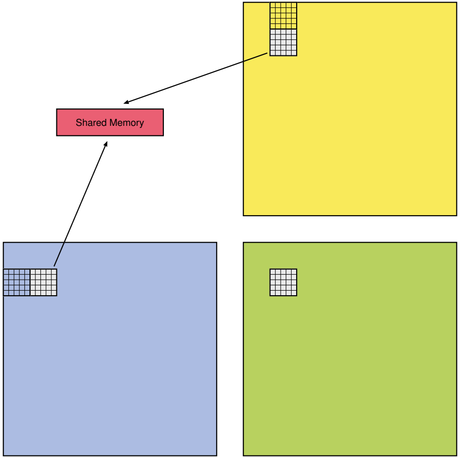
→ Synchronization is needed!

Synchronize threads within a block

```
__syncthreads ();
```

BLOCK MATRIX MULTIPLICATION

Workflow, $k \times k$ blocks

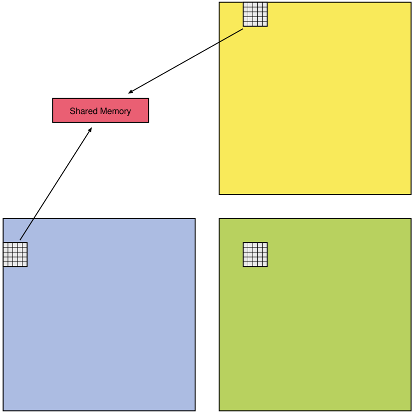


Each thread (global index (x, y), local index (s, t) in block (u, v)) does:

$C_{y,x} \leftarrow 0$
for $i = 1$ **to** k **do**
  Copy input data A_{vi}, B_{iu} to shared memory (one element per thread)
  Wait until all threads in block have copied their data
  Compute value (t, s) in $A_{vi}B_{iu}$.
  Add this value to $C_{y,x}$
  Wait until all threads in block have finished computation
end for

BLOCK MATRIX MULTIPLICATION

Offsets and indexes



Use (2D coordinates of) upper left corner of input blocks as reference.

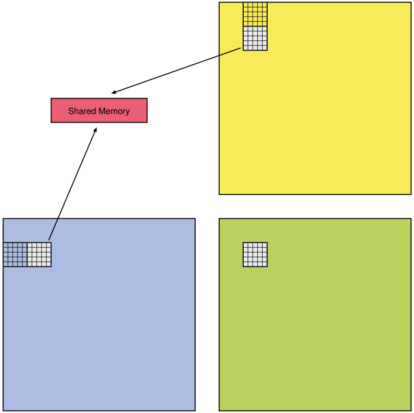
For i=1,...k:

A-block row	`blockIdx.y * block_size`
A-block column	`i * block_size`
B-block row	`i * block_size * n`
B-block column	`blockIdx.x * block_size`

Relative position inside the block corresponds to the local (block internal) thread index

BLOCK MATRIX MULTIPLICATION

Offsets and indexes



Use (2D coordinates of) upper left corner of input blocks as reference.

For i=1,...k:

A-block row	`blockIdx.y * block_size`
A-block column	`i * block_size`
B-block row	`i * block_size * n`
B-block column	`blockIdx.x * block_size`

Relative position inside the block corresponds to the local (block internal) thread index

EXERCISE

Matrix multiplication with CUDA using shared memory

Detailed instructions

.../exercises/tasks/Cuda_MM_shared/Instructions.ipynb

- Implement a matrix multiplication with CUDA using shared memory.

EXERCISE

Measured numbers

Results on JUWELS Booster (GFlops/s):

matrix size	1024	4096	8192	16384
Simple	286	1186	1554	1769
Shared memory(16,16)	296	952	1560	1742
Shared memory(32,32)	339	1369	1945	2205

EXERCISE
Measured numbers

Results on JUWELS Booster (GFlops/s):

matrix size	1024	4096	8192	16384
Simple	286	1186	1554	1769
Shared memory(16,16)	296	952	1560	1742
Shared memory(32,32)	339	1369	1945	2205

Thank you for your attention!