GPU PROGRAMMING WITH CUDA

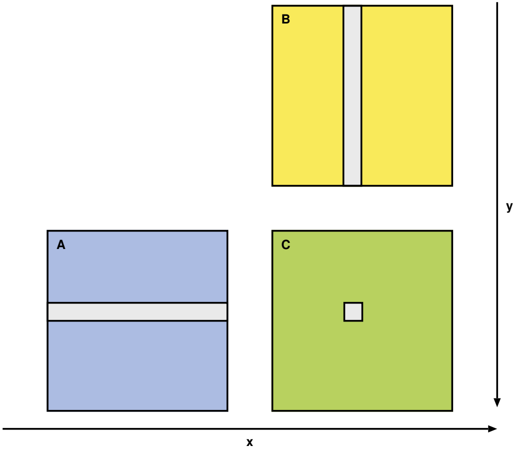
Matrix multiplication

April 26, 2022 | Kaveh Haghighi Mood, Jochen Kreutz | JSC

JÜLICH Forschungszentrum

CUDA MATRIX MULTIPLICATION

Distribution of work

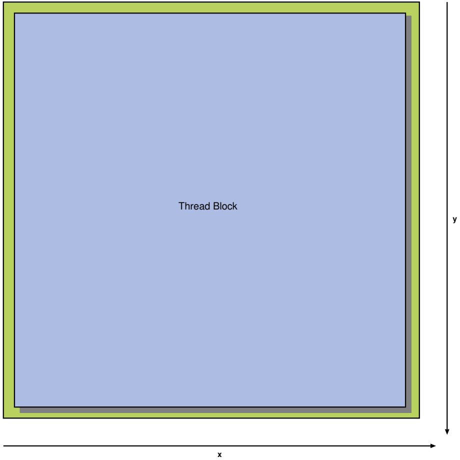


$$C_{row,col} = \sum_{i=1}^{N} A_{row,i} * B_{i,col}$$

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

CUDA MATRIX MULTIPLICATION

Execution grid layout

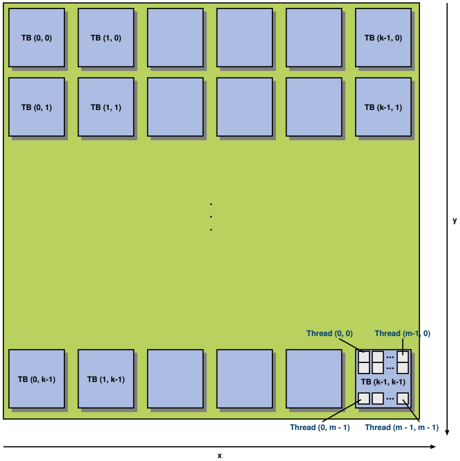


First naive idea:

- use one big thread block to cover all result elements
- Thread blocks are limited in size, thus we need several thread blocks to cover the full matrix C
- In addition, using only one thread block will decrease performance (due to reduced device occupancy)

CUDA MATRIX MULTIPLICATION

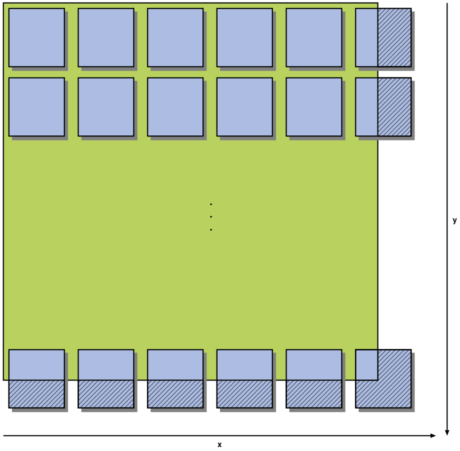
Execution grid layout



- Cover result matrix C of size n x n by using a 2d kernel execution grid with k * k thread blocks (TB)
- Use 2d thread blocks with fixed block size m
- k = n / m (n divisible by m)
- k = n / m + 1 (n not divisible by m)

CUDA MATRIX MULTIPLICATION

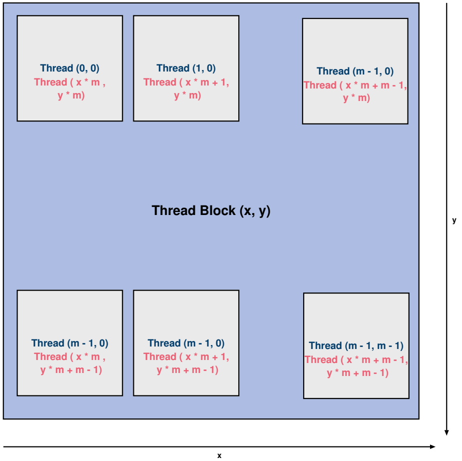
Execution grid layout



- k = n / m + 1 (n not divisible by m)
- All thread blocks have the same size
- Not possible to create ”partial blocks”
- To take care that some threads might not have to do any work (avoid out ouf bound memory access !)

CUDA MATRIX MULTIPLICATION

Execution grid layout



- Threads can be addressed via local index (block internal) and global index (full grid)
- Use available keywords in your kernel for targetting certain threads:
  - `blockIdx.[x, y, z]`
  - `blockDim.[x, y, z]`
  - `threadIdx.[x, y, z]`

CUDA MATRIX MULTIPLICATION

Execution grid layout

dim3 blockDim

```
dim3 blockDim { size_t blockDimX, size_t blockDimY, size_t blockDimZ }
```

On JUWELS Booster (Nvidia A100):

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

Example

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

CUDA MATRIX MULTIPLICATION

Execution grid layout

dim3 gridDim

```
dim3 gridDim { size_t gridDimX, size_t gridDimY, size_t gridDimZ }
```

On JUWELS Booster (Nvidia A100):

- Max. dim. of a grid: `2147483647 x 65535 x 65535`
- Use `cudaGetDeviceProperties()` to get device properties

Example

```
// problem dimension: 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); // don't need to write z = 1
```

CUDA MATRIX MULTIPLICATION

Calling the kernel

Define dimensions of thread blocks

```
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_name <<< dim3 gridDim, dim3 blockDim >>> ([kernel args])
```

CUDA MATRIX MULTIPLICATION

Cuda kernel

Example

```
__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 matrix multiplication with Cuda

Location:

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

PERFORMANCE CONSIDERATIONS

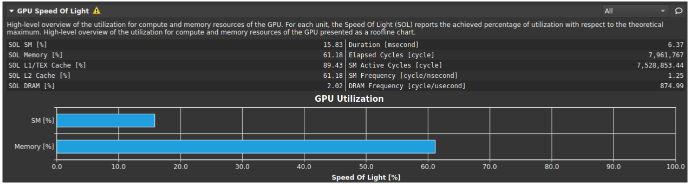
Measured numbers

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

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

PERFORMANCE CONSIDERATIONS

Profiler hints for simple matrix multiplication

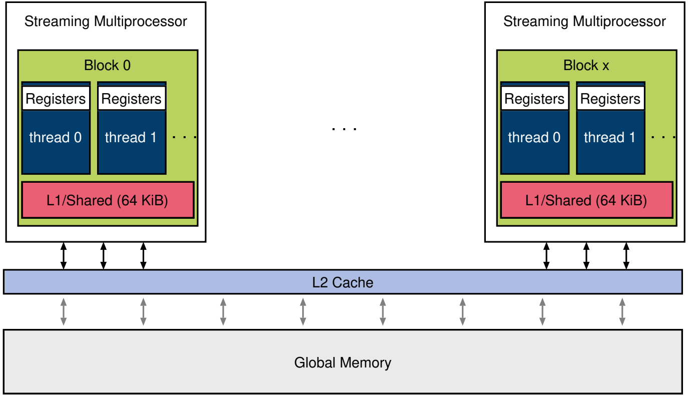


- get useful hints from profiler
- helps to identify hotspots and potential performance issues
- get an overview timeline using Nsight Systems
- can analyse kernels individually using Nsight Compute
- indicates very low compute utilization
- `dgemm` kernel is memory-bound (GPU cores spend lots of time waiting for data)

JÜLICH Forschungszentrum

PERFORMANCE CONSIDERATIONS

GPU memory layout (schematics)

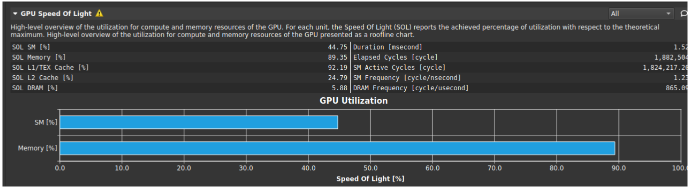


- matrix array `C` located in global memory
- `cvalue` located in registers on SM: faster write operations

JÜLICH Forschungszentrum

PERFORMANCE CONSIDERATIONS

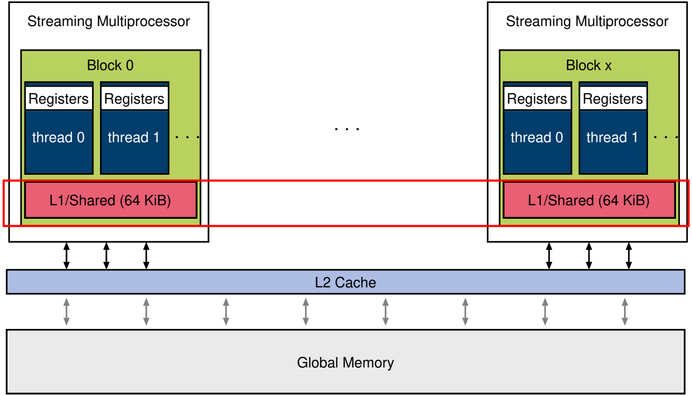
Profiler hints for simple matrix multiplication



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

PERFORMANCE CONSIDERATIONS

GPU memory layout (schematics)



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

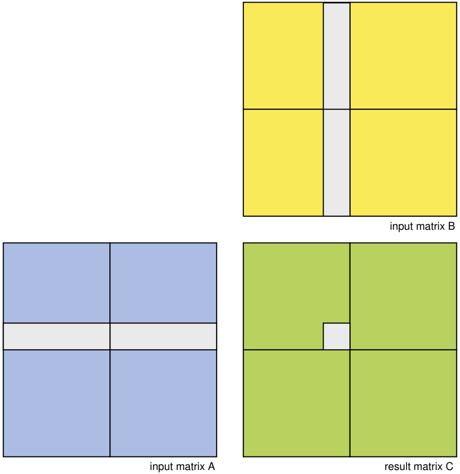
Copy data into shared memory

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

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

BLOCK MATRIX MULTIPLICATION

Idea

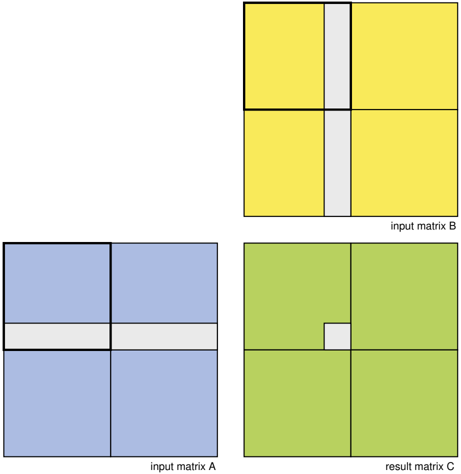


Split computation of result element into parts (assume N is even here)

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

BLOCK MATRIX MULTIPLICATION

Idea



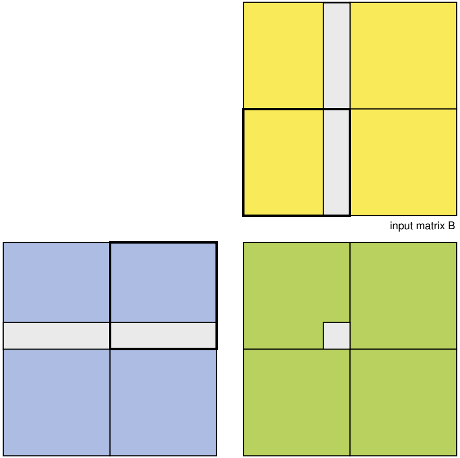
Split computation of result element into parts (assume N is even here)

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

$$= \sum_{i=1}^{\frac{N}{2}} A_{row,i} * B_{i,col}$$

BLOCK MATRIX MULTIPLICATION

Idea



Split computation of result element into parts (assume N is even here)

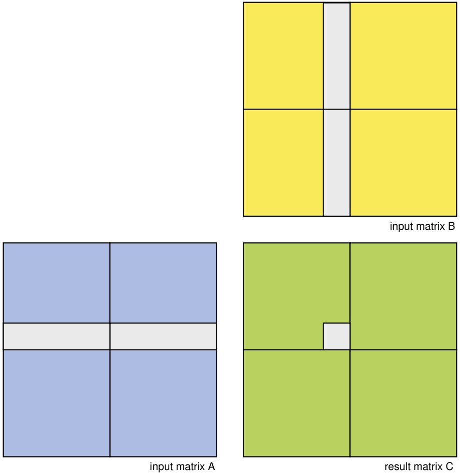
$$C_{row,col} = \sum_{i=1}^{N} A_{row,i} * B_{i,col}$$

$$= \sum_{i=1}^{\frac{N}{2}} A_{row,i} * B_{i,col}$$

$$+ \sum_{i=\frac{N}{2}+1}^{N} A_{row,i} * B_{i,col}$$

BLOCK MATRIX MULTIPLICATION

Idea



$$C_{row,col} = \sum_{i=1}^{\frac{N}{2}} A_{row,i} * B_{i,col} + \sum_{i=\frac{N}{2}+1}^{N} A_{row,i} * B_{i,col}$$

consider all result elements within the same block in C:

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

$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$$

$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$$

$$C_{22} = A_{21} * B_{12} + A_{22} * B_{21}$$

BLOCK MATRIX MULTIPLICATION

Example

$$C = A * B \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

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

$$C_{11} = \begin{pmatrix} 1 & 2 \\ 4 & 1 \end{pmatrix} * \frac{1}{40} \begin{pmatrix} -9 & 11 \\ 1 & -9 \end{pmatrix} + \begin{pmatrix} 3 & 4 \\ 2 & 3 \end{pmatrix} * \frac{1}{40} \begin{pmatrix} 1 & 1 \\ 11 & 1 \end{pmatrix} \tag{1}$$

BLOCK MATRIX MULTIPLICATION

Example

$$C = A * B$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$A = \begin{pmatrix} \begin{pmatrix} 1 & 2 \\ 4 & 1 \end{pmatrix} & \begin{pmatrix} 3 & 4 \\ 2 & 3 \end{pmatrix} \\ \begin{pmatrix} 3 & 4 \\ 2 & 3 \end{pmatrix} & \begin{pmatrix} 1 & 2 \\ 4 & 1 \end{pmatrix} \end{pmatrix}$$

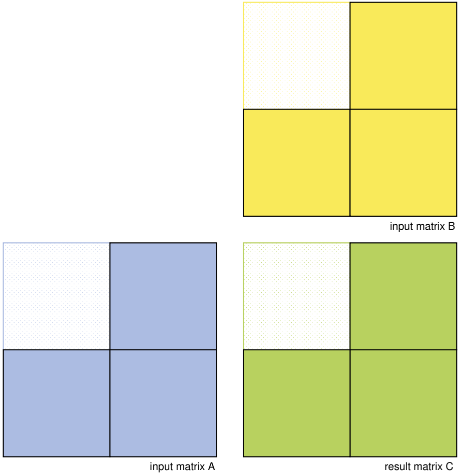
$$B = \begin{pmatrix} \begin{pmatrix} -9 & 11 \\ 1 & -9 \end{pmatrix} & \begin{pmatrix} 1 & 1 \\ 11 & 1 \end{pmatrix} \\ \begin{pmatrix} 1 & 1 \\ 11 & 1 \end{pmatrix} & \begin{pmatrix} -9 & 11 \\ 1 & -9 \end{pmatrix} \end{pmatrix} * \frac{1}{40}$$

$$= \frac{1}{40} * \begin{pmatrix} -7 & -7 \\ -35 & 35 \end{pmatrix} + \frac{1}{40} * \begin{pmatrix} 47 & 7 \\ 35 & 5 \end{pmatrix} = \frac{1}{40} * \begin{pmatrix} 40 & 0 \\ 0 & 40 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

do C_{12}, C_{21} and C_{22} the same way

BLOCK MATRIX MULTIPLICATION

Using shared memory

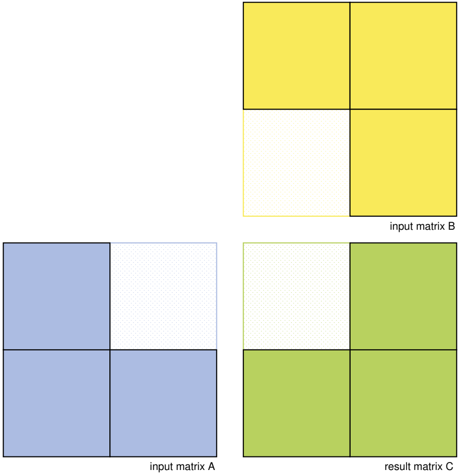


consider all result elements within the same block:

- all threads dealing with those elements will have to access input data from the same blocks of A and B for the first part of the computation

BLOCK MATRIX MULTIPLICATION

Using shared memory

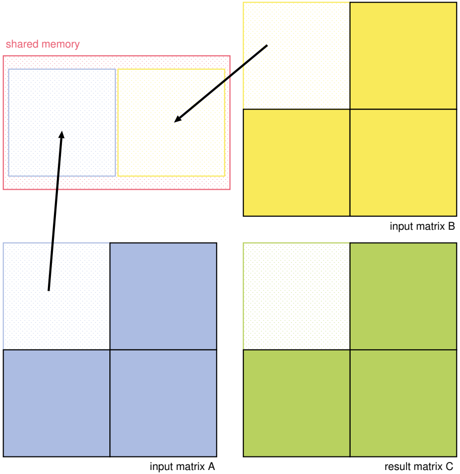


consider all result elements within the same block:

- all threads dealing with those elements will have to access input data from the same blocks of A and B for the first part of the computation
- same counts for the successing compute parts

BLOCK MATRIX MULTIPLICATION

Using shared memory

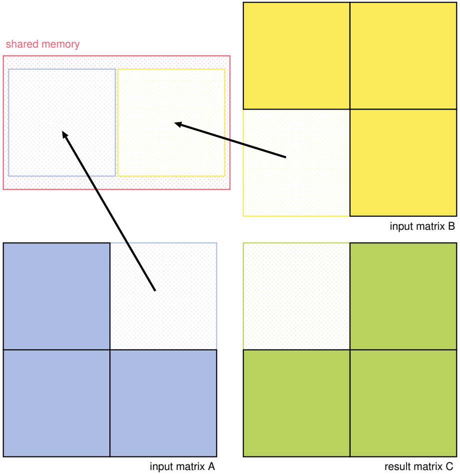


consider all result elements within the same block:

- all threads dealing with those elements will have to access input data from the same blocks of A and B for the first part of the computation
- same counts for the successing compute parts
- hence store a data copy of the input blocks into shared memory
- this prevents repeated reads from the global memory

BLOCK MATRIX MULTIPLICATION

Using shared memory

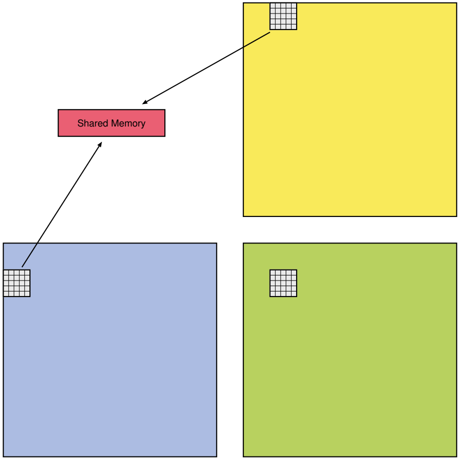


consider all result elements within the same block:

- mapping logical matrix blocks to your Cuda thread blocks ensures that all threads in your result blocks see the same shared memory
- each thread reads 1 element of A and 1 element of B and stores in into the shared memory

BLOCK MATRIX MULTIPLICATION

Workflow

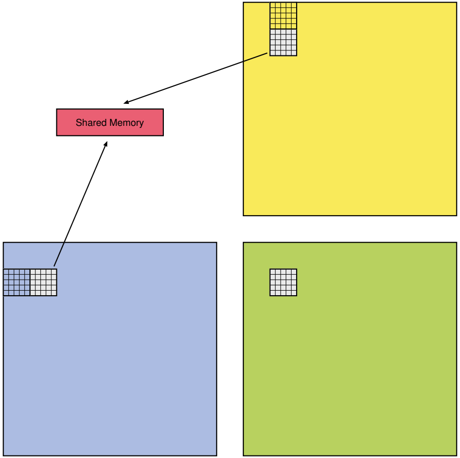


for each result element (thread):

- set result element to zero
- for each pair of blocks
  - copy input data to shared memory (one element from A and B)
  - do partial sum using shared memory
  - add partial sum to result element

BLOCK MATRIX MULTIPLICATION

Workflow

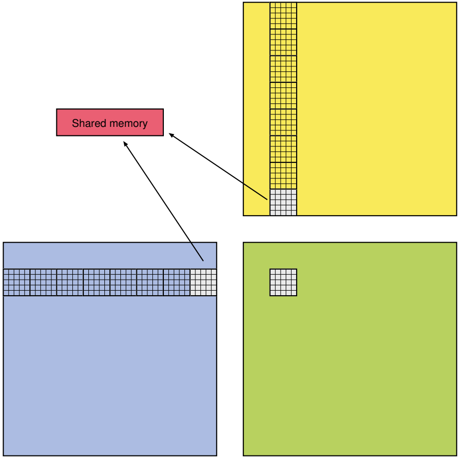


for each result element (thread):

- set result element to zero
- for each pair of blocks
  - copy input data to shared memory (one element from A and B)
  - do partial sum using shared memory
  - add partial sum to result element

BLOCK MATRIX MULTIPLICATION

Workflow

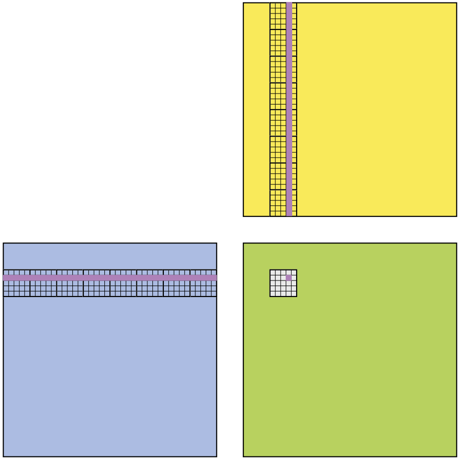


for each result element (thread):

- set result element to zero
- for each pair of blocks
    - copy input data to shared memory (one element from A and B)
    - do partial sum using shared memory
    - add partial sum to result element

BLOCK MATRIX MULTIPLICATION

Workflow



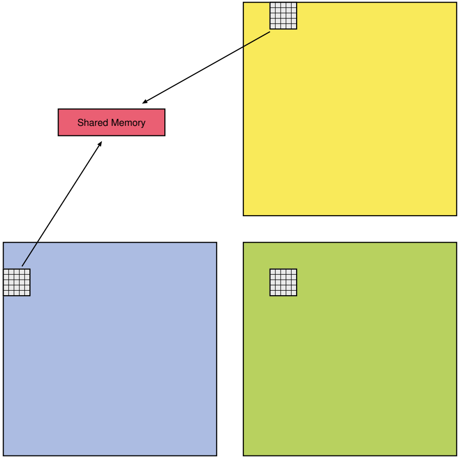
for each result element (thread):

- set result element to zero
- for each pair of blocks
  - copy input data to shared memory (one element from A and B)
  - do partial sum using shared memory
  - add partial sum to result element

JÜLICH Forschungszentrum

BLOCK MATRIX MULTIPLICATION

Thread synchronization



Thread synchronization

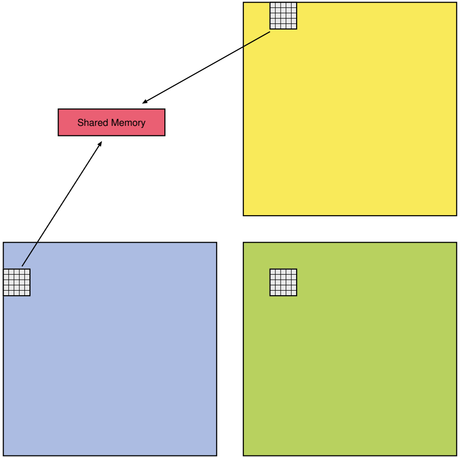
Threads within a thread block might be executed one after the other. Hence, synchronization is needed !

Synchronize threads within a thread block

```
__syncthreads ();
```

BLOCK MATRIX MULTIPLICATION

Thread synchronization

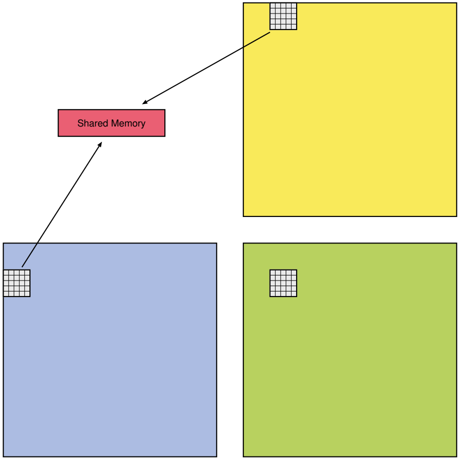


for each result element (thread):

- set result element to zero
- for each pair of blocks
  - copy input data to shared memory (one element from A and B)
  - wait until all threads have copied their data
  - do partial sum
  - wait until all threads finished computation on current data
  - add partial sum to result element

BLOCK MATRIX MULTIPLICATION

Offsets and indexes



idea: use (2d coordinates of) upper left corner of input blocks as reference

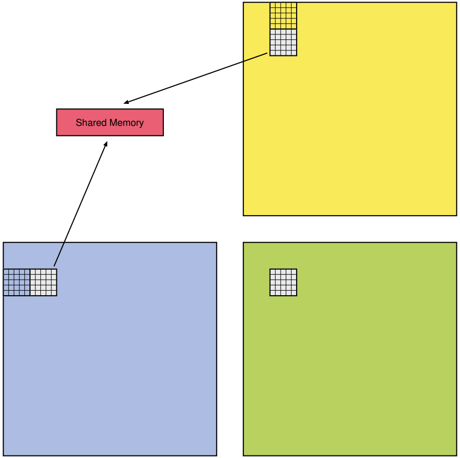
- relative positions inside the input blocks correspond to the local (block internal) thread indexes

starting point:

- row in A (`blockAy`):
  `blockIdx.y * block_size`
- col in B (`blockBx`):
  `blockidx.x * block_size`
- `blockAx` and `blockBy` will be `0` at start

BLOCK MATRIX MULTIPLICATION

Offsets and indexes



idea: use (2d coordinates of) upper left corner of input blocks as reference

moving input blocks:

- A moving to x direction by adding: `block_size`
- B moving to y direction by adding: `n * block_size`

shared memory blocks:

- use local (block internal) thread indexes to select correct row and column

JÜLICH Forschungszentrum

EXERCISE

Matrix multiplication with Cuda using shared memory

Location:

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

EXERCISE

Measured numbers

Results on JUWELS Booster (gflops):

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):

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!