CUDA-C PROGRAMMING - INTERMEDIATE CONCEPTS

Modern computing for physics

J.Pazzini
Padova University

Physics of Data
AA 2024-2025

UNIVERSITÀ DEGLI STUDI DI PADOVA

Matrix-Matrix multiplication - 1D Grid

Using `matrix_multiplication_1d.cu`

```
[...]

// CUDA kernel to perform matrix multiplication
__global__ void matrixMultiplication(const float* M, const float* N, float* P, const int width) {
   // Calculate the thread ID of the overall grid
   int idx = blockIdx.x * blockDim.x + threadIdx.x;

   // Each thread computes one element of the result matrix
   if (idx < width * width) {
       int row = idx / width;
       int col = idx % width;

       float sum = 0.;
       // Accessing all elements of a row of M and a column of N
       for (int k = 0; k < width; ++k) {
           sum += M[row * width + k] * N[k * width + col];
       }
       P[idx] = sum;
   }
}

[...]
```

Matrix-Matrix multiplication - 1D Grid

```
[...]

   // Compute the number of blocks and threads per block
   // Blocks are 1-dimensional
   int N_b   = ceil(float(WIDTH * WIDTH)/THREADS_PER_BLOCK);
   int N_tpb = THREADS_PER_BLOCK;

   // Launch CUDA kernel
   matrixMultiplication<<<N_b, N_tpb>>>(d_M, d_N, d_P, WIDTH);

[...]
```

Matrix-Matrix multiplication - 2D Grid

Using `matrix_multiplication_2d.cu`

```
[...]

// CUDA kernel to perform matrix multiplication
__global__ void matrixMultiplication(const float* M, const float* N, float* P, const int width) {
   // Calculate the thread ID of the overall grid
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;

   // Each thread computes one element of the result matrix
   if (row < width && col < width) {
       float sum = 0.;
       // Accessing all elements of a row of M and a column of N
       for (int k = 0; k < width; ++k) {
           sum += M[row * width + k] * N[k * width + col];
       }
       P[row * width + col] = sum;
   }
}

[...]
```

Matrix-Matrix multiplication - 2D Grid

```
[...]

   // Compute the dimensions of blocks and grid
   // Blocks are now 2-dimensional
   dim3 blockSize(THREADS_PER_BLOCK_X,THREADS_PER_BLOCK_Y);
   dim3 gridSize(ceil(float(WIDTH)/blockSize.x),ceil(float(WIDTH)/blockSize.y));

   // Launch CUDA kernel
   matrixMultiplication<<<gridSize, blockSize>>>(d_M, d_N, d_P, WIDTH);


[...]
```

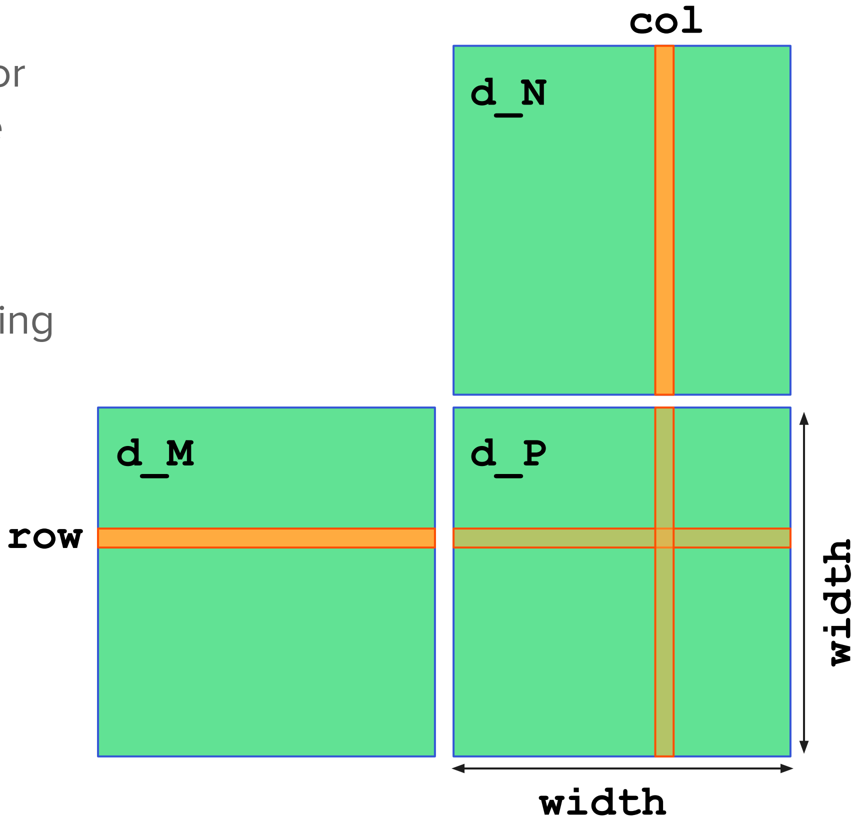
Matrix-Matrix multiplication - a sidenote

Regardless of using a 1D or 2D grid to launch kernels, in both cases 1 kernel is responsible for calculating the value of a single element of the `d_P` matrix

A large number of threads are launched but prompt intercepted by the if statement protecting against attempting to fetch non-existing data from memory

It’s now time for a couple of considerations:

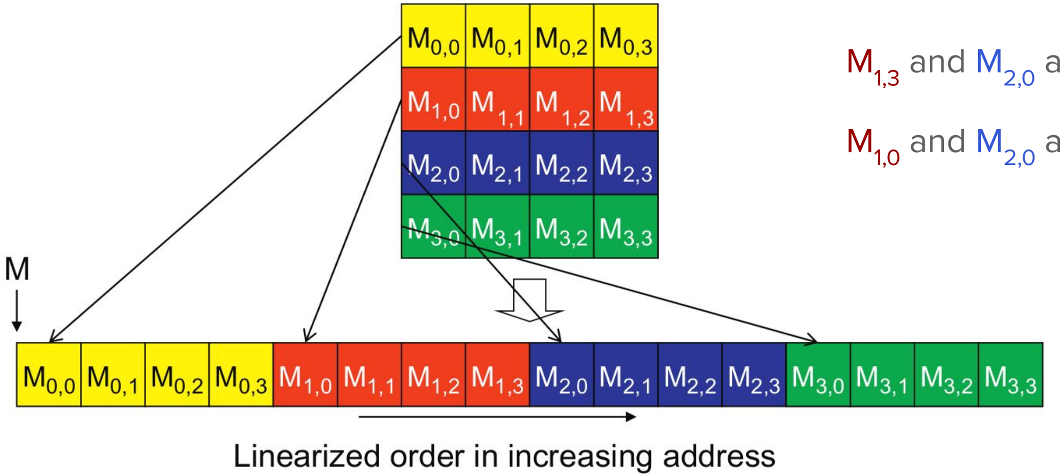
- Coalescing
- Tiling



Memory coalescing

On the memory:

- The address location of nD arrays is linearized in memory (row-major in C/C++)
- Global memory access is slow, and it's even slower if two consecutive data accesses happen for address of data that are far from one another (related to how the DRAM works...)



$M_{1,3}$ and $M_{2,0}$ are close in memory ➔ fast access

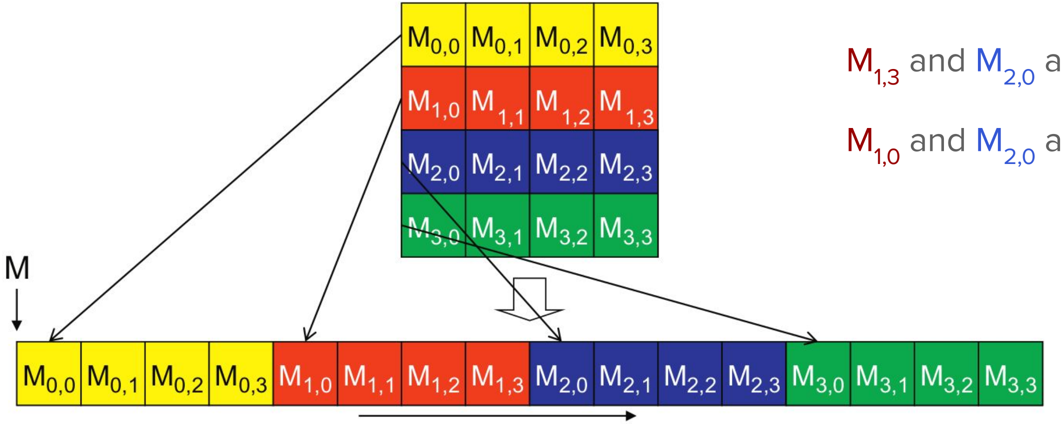
$M_{1,0}$ and $M_{2,0}$ are far in memory ➔ slow access

Memory coalescing

On top of this:

- Threads of the same warp are executed simultaneously
- Each thread operates on a given set of data elements

→ In CUDA, when multiple threads in a warp (a group of 32 threads) access memory, the GPU tries to combine these accesses into a single transaction if possible. This is called coalesced memory access, and it is much faster than individual, scattered accesses.



Linearized order in increasing address

$M_{1,3}$ and $M_{2,0}$ are close in memory → fast access

$M_{1,0}$ and $M_{2,0}$ are far in memory → slow access

Memory coalescing

⇒ Organizing data such that simultaneously-running threads are fetching from memory “close by” elements would result in shorter access time and better performance!

⇒ When accessing global memory, we would like to make sure that concurrent threads in the same warp access nearby memory locations

Let’s visualize this with our matrix-matrix multiplication example:

```
// Each thread computes one element of the result matrix
if (row < width && col < width) {
    float sum = 0.;
    // Accessing all elements of a row of M and a column of N
    for (int k = 0; k < width; ++k) {
        sum += M[row * width + k] * N[k * width + col];
    }
    P[row * width + col] = sum;
}
```

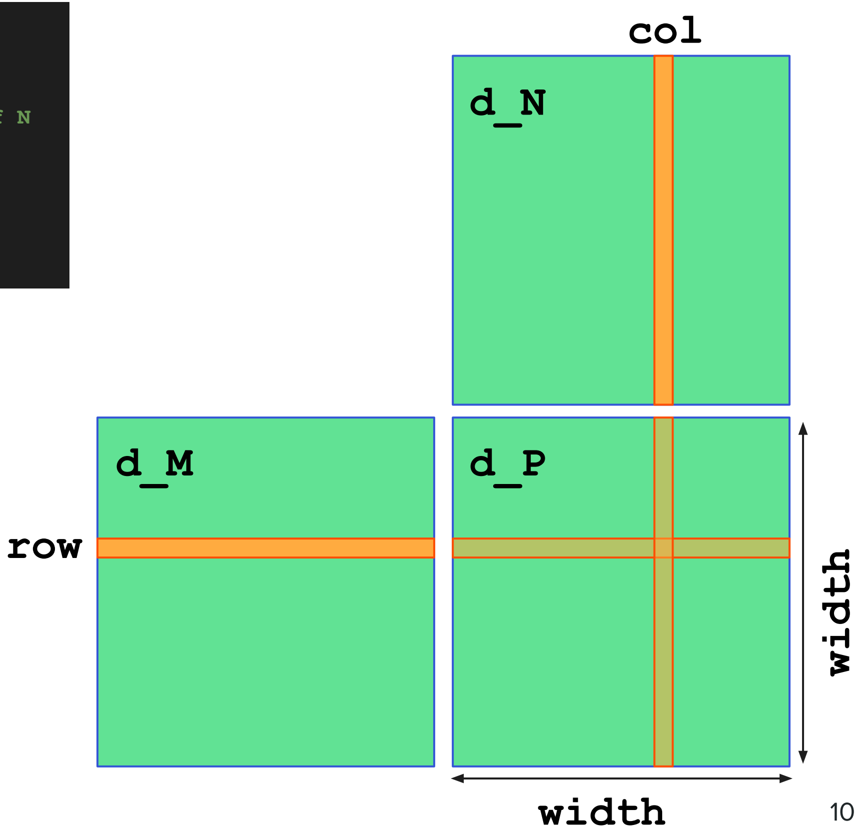
1 thread → 1 Element of the matrix **`P`**

`M[row * width + k]` accesses elements of the matrix M row by row

`N[k * width + col]` accesses elements of the matrix N column by column

Memory coalescing

```
// Each thread computes one element of the result matrix
if (row < width && col < width) {
    float sum = 0.;
    // Accessing all elements of a row of M and a column of N
    for (int k = 0; k < width; ++k) {
        sum += M[row * width + k] * N[k * width + col];
    }
    P[row * width + col] = sum;
}
```

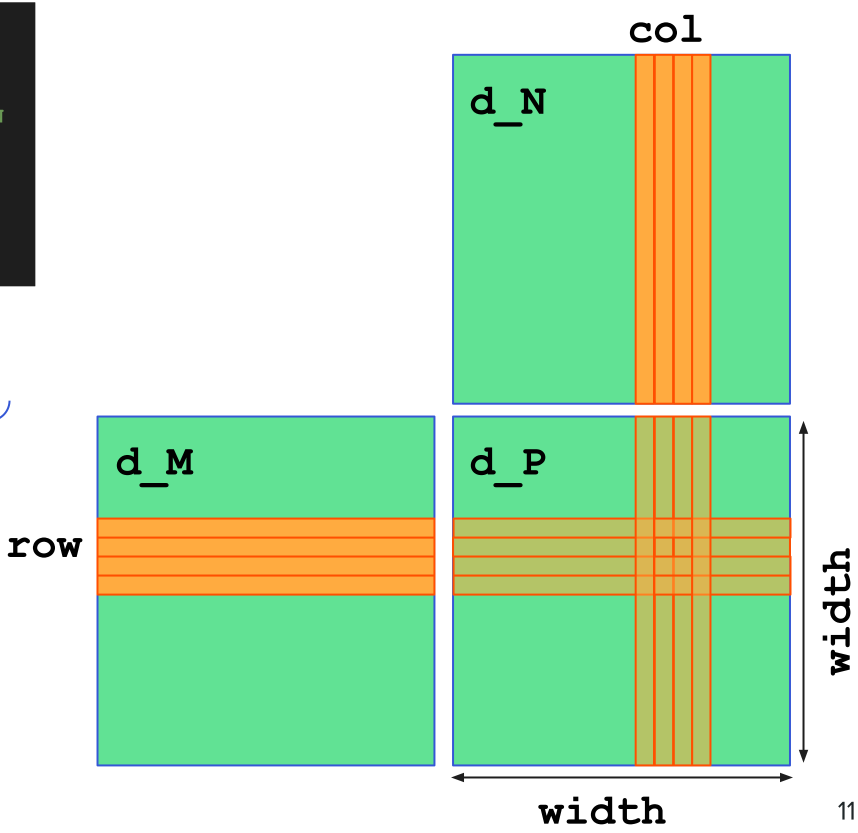


Memory coalescing

```
// Each thread computes one element of the result matrix
if (row < width && col < width) {
    float sum = 0.;
    // Accessing all elements of a row of M and a column of N
    for (int k = 0; k < width; ++k) {
        sum += M[row * width + k] * N[k * width + col];
    }
    P[row * width + col] = sum;
}
```

1 warp → N threads → N elements of the matrix `P`

All simultaneously executed (SIMD)



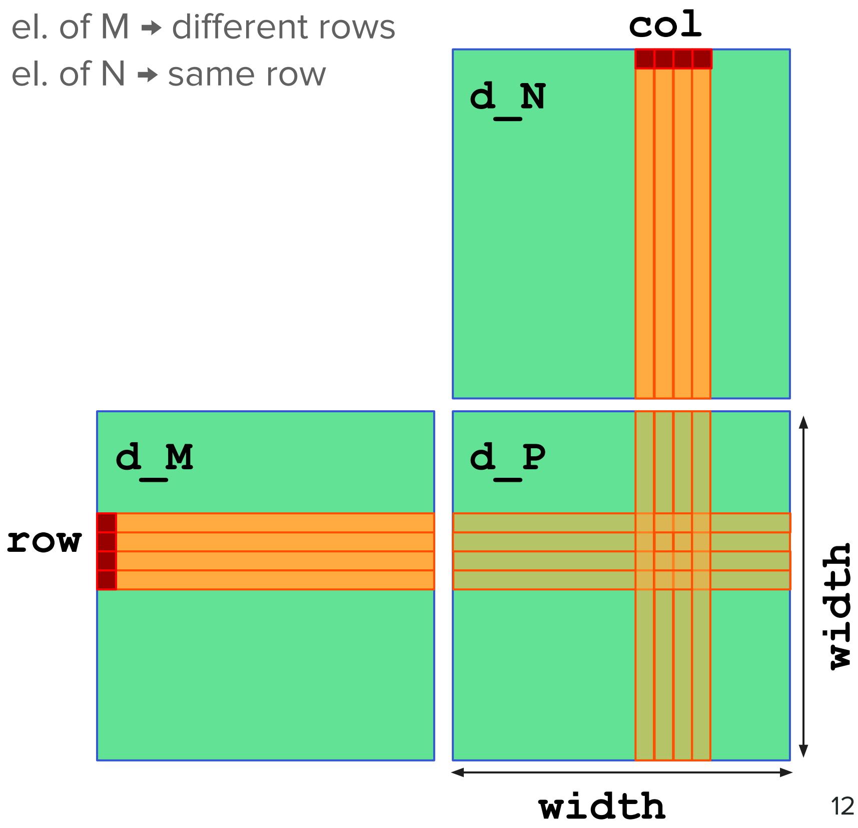
Memory coalescing

```
// Each thread computes one element of the result matrix
if (row < width && col < width) {
    float sum = 0.;
    // Accessing all elements of a row of M and a column of N
    for (int k = 0; k < width; ++k) {
        sum += M[row * width + k] * N[k * width + col];
    }
    P[row * width + col] = sum;
}
```

k=0

4 el. of M ➔ different rows

4 el. of N ➔ same row



Memory coalescing

```
// Each thread computes one element of the result matrix
if (row < width && col < width) {
    float sum = 0.;
    // Accessing all elements of a row of M and a column of N
    for (int k = 0; k < width; ++k) {
        sum += M[row * width + k] * N[k * width + col];
    }
    P[row * width + col] = sum;
}
```

k=0

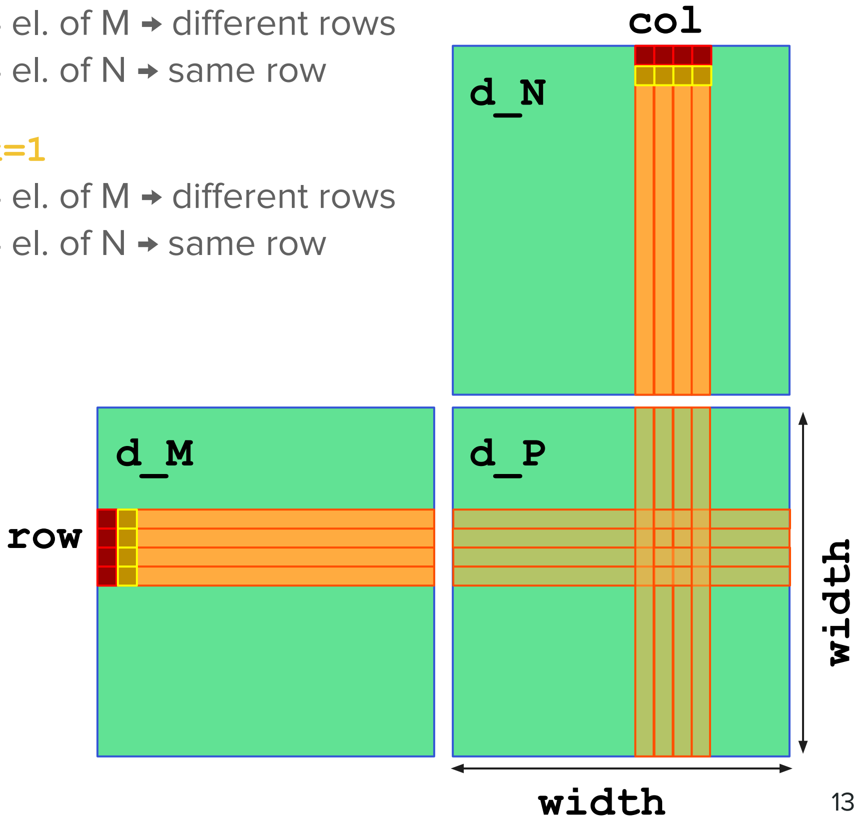
4 el. of M ➔ different rows

4 el. of N ➔ same row

k=1

4 el. of M ➔ different rows

4 el. of N ➔ same row



Memory coalescing

```
// Each thread computes one element of the result matrix
if (row < width && col < width) {
    float sum = 0.;
    // Accessing all elements of a row of M and a column of N
    for (int k = 0; k < width; ++k) {
        sum += M[row * width + k] * N[k * width + col];
    }
    P[row * width + col] = sum;
}
```

k=0

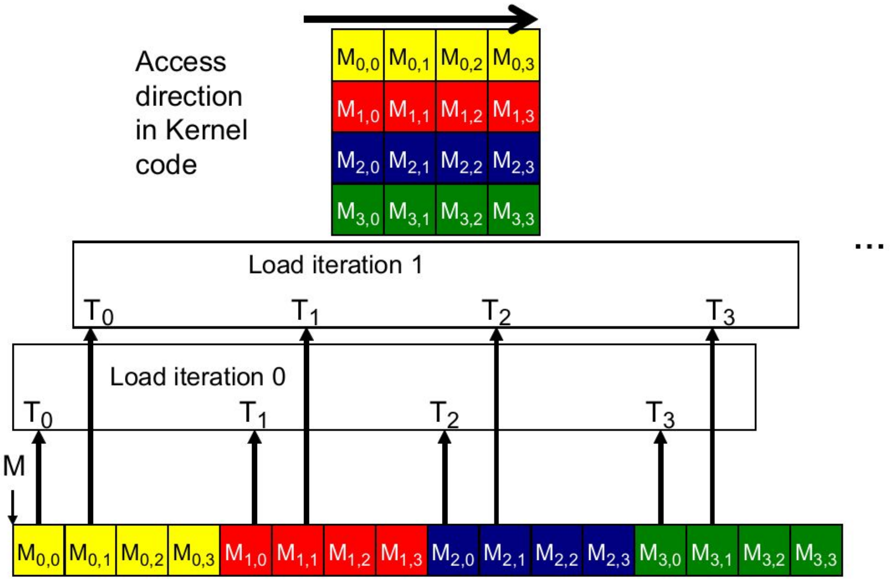
4 el. of M → different rows

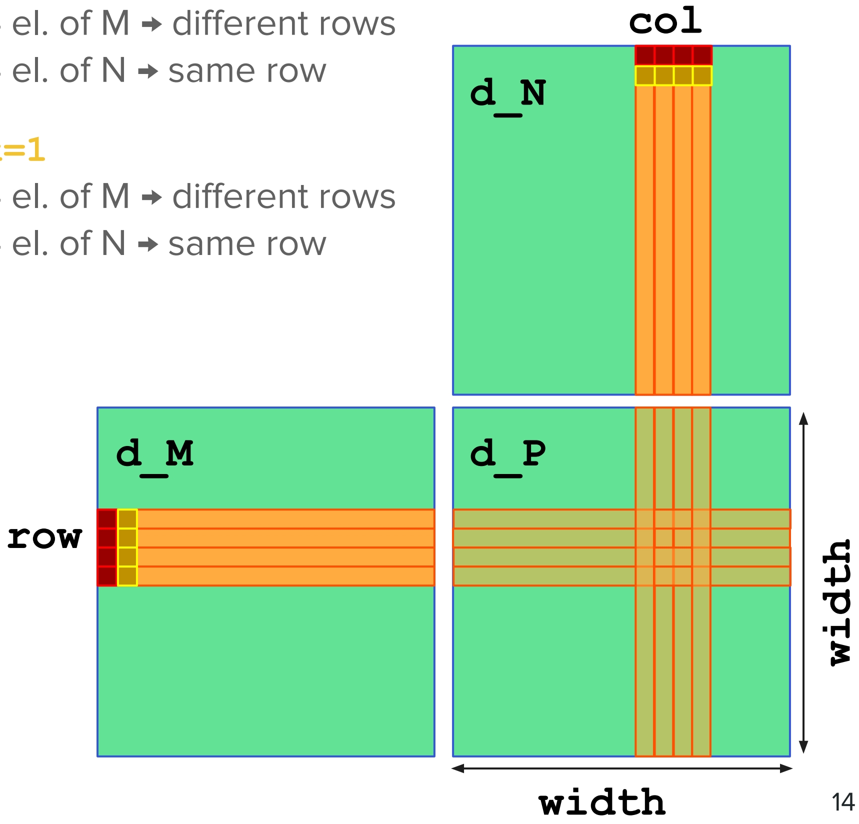
4 el. of N → same row

k=1

4 el. of M → different rows

4 el. of N → same row





Memory coalescing

```
// Each thread computes one element of the result matrix
if (row < width && col < width) {
    float sum = 0.;
    // Accessing all elements of a row of M and a column of N
    for (int k = 0; k < width; ++k) {
        sum += M[row * width + k] * N[k * width + col];
    }
    P[row * width + col] = sum;
}
```

k=0

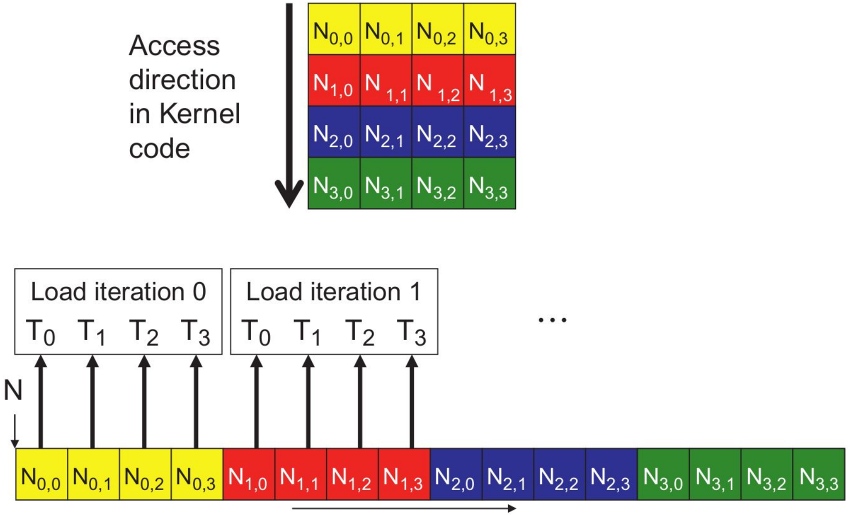
4 el. of M ➔ different rows

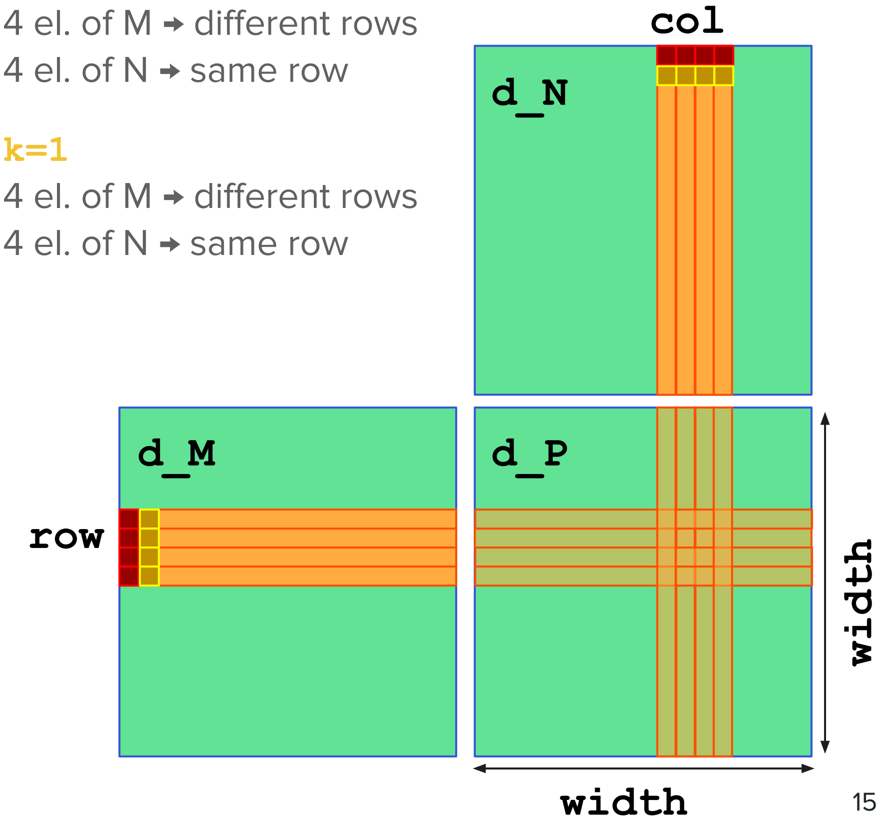
4 el. of N ➔ same row

k=1

4 el. of M ➔ different rows

4 el. of N ➔ same row

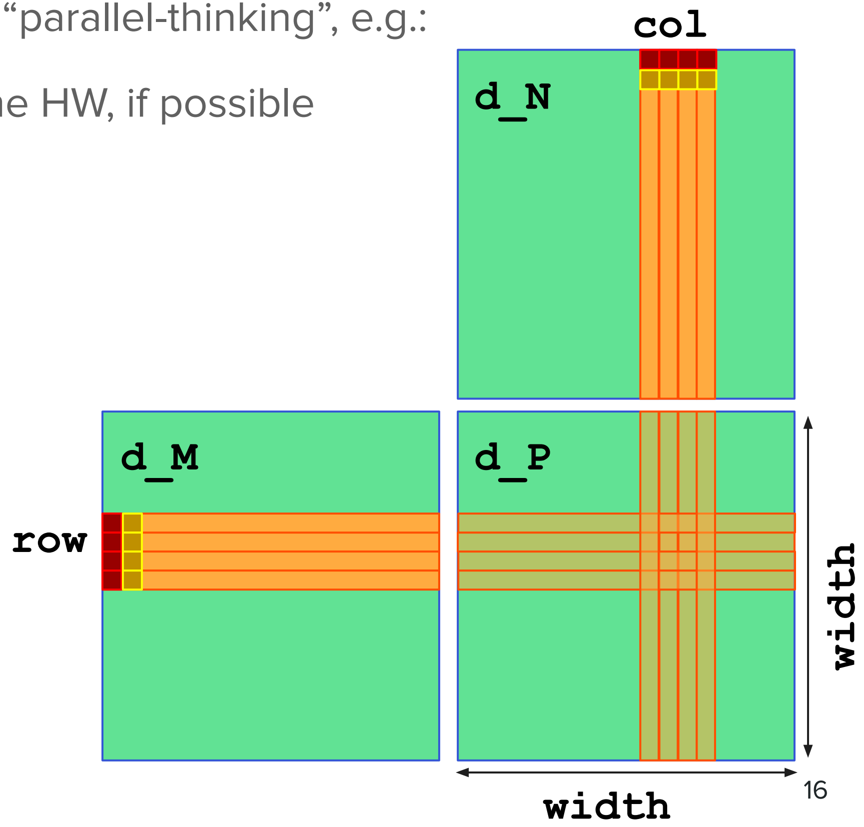




Overcoming the limitations

To overcome the memory limitations induced by non-coalesced memory access we have to rethink our problem, and modify it by applying some "parallel-thinking", e.g.:

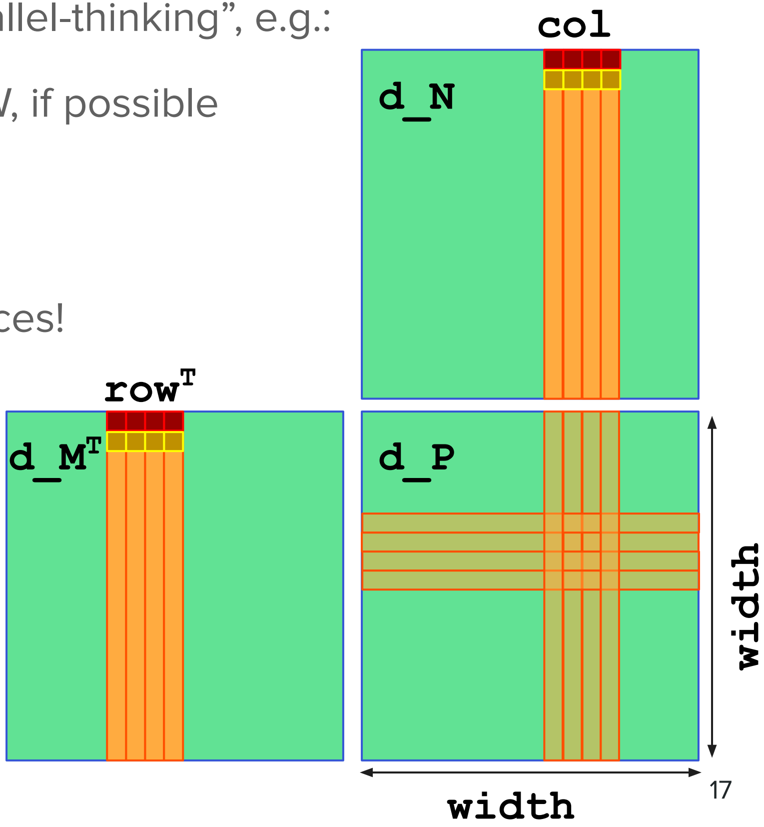
1. Reshape the data and algorithm to better suit the HW, if possible



Overcoming the limitations

To overcome the memory limitations induced by non-coalesced memory access we have to rethink our problem, and modify it by applying some “parallel-thinking”, e.g.:

1. Reshape the data and algorithm to better suit the HW, if possible
   - What if instead of storing matrix `M` in memory, we stored its transposed version M^T?
   - If we could also change the kernel accordingly, we would gain coalesced access for both matrices!

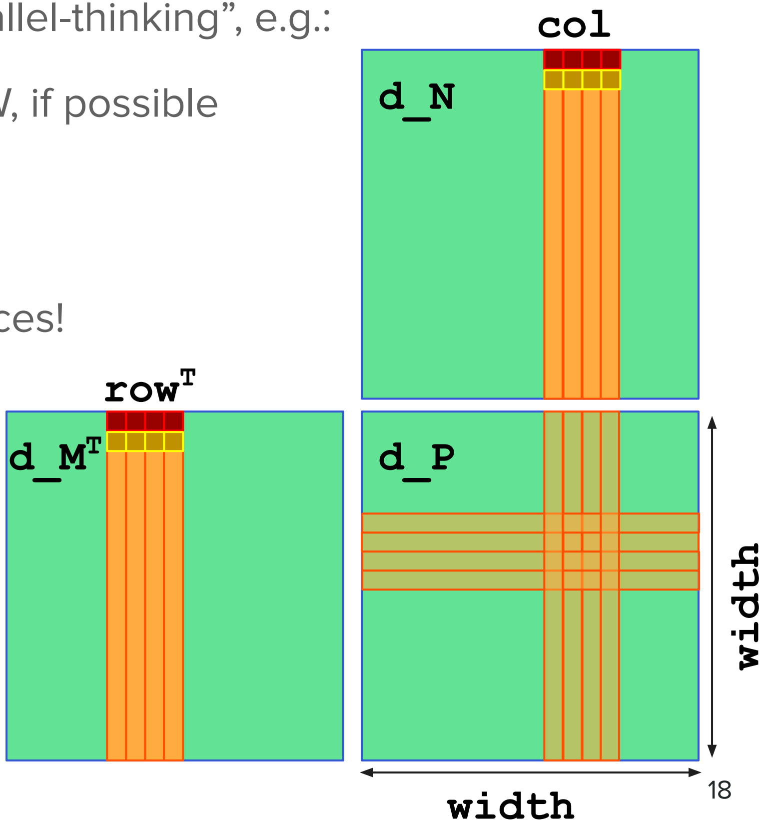


Overcoming the limitations

To overcome the memory limitations induced by non-coalesced memory access we have to rethink our problem, and modify it by applying some “parallel-thinking”, e.g.:

1. Reshape the data and algorithm to better suit the HW, if possible
   - ➔ What if instead of storing matrix `M` in memory, we stored its transposed version M^T?
   - ➔ If we could also change the kernel accordingly, we would gain coalesced access for both matrices!

2. Investigate the data usage of our algorithm and plan to share data across all active threads



Work at home/lab

- generalization to non-square matrix-matrix multiplications
- square matrix multiplication with transposition
- converting image from RGB (2D x 3 color channels) to grayscale