CUDA-C PROGRAMMING - PROGRAMMING PATTERNS

Modern computing for physics

J.Pazzini
Padova University

Physics of Data
AA 2024-2025

UNIVERSITÀ DEGLI STUDI DI PADOVA

Memory sharing - Tiling

To overcome the memory limitations, a common strategy is partition the data into subsets so that each subset fits into the *shared memory*

This strategy takes advantage of re-using data across multiple threads, instead of loading it from global memory per every thread

➡ Memory can be shared only across a block, but not across multiple blocks

Memory sharing - Tiling

To overcome the memory limitations, a common strategy is partition the data into subsets so that each subset fits into the *shared memory*

This strategy takes advantage of re-using data across multiple threads, instead of loading it from global memory per every thread

➡ Memory can be shared only across a block, but not across multiple blocks

This is also referred to as ***Tiling***

➡ In a tiled algorithm, threads collaborate to load input elements into an on-chip memory and then access the on-chip memory for their subsequent use of these elements

➡ Blocks of threads will then collaborate and access the same shared data

➡ The collection of output elements processed by each block is often referred as an output tile

1D stencil

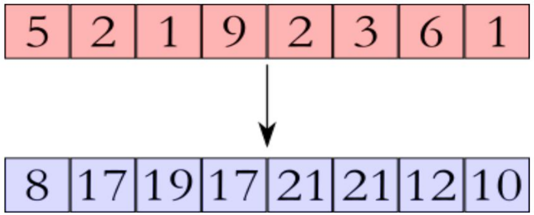
A *stencil* is a computational pattern commonly used in numerical methods and scientific computing to update elements in a one-dimensional array based on the values of neighboring elements.

This technique is often used in simulations of physical systems, such as solving differential equations, fluid dynamics, or heat diffusion problems.

A 1D convolution is a specific type of 1D stencil pattern.

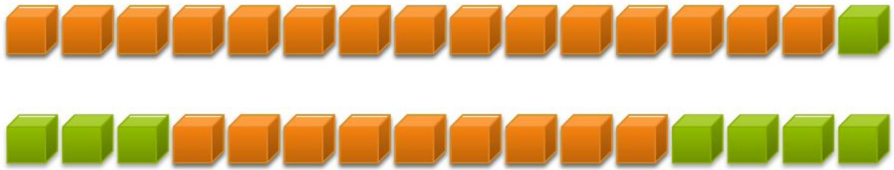
Let's consider applying a 1D stencil to a 1D array of elements
➡ Each output element is the sum of input elements within a radius

e.g. for radius = 2

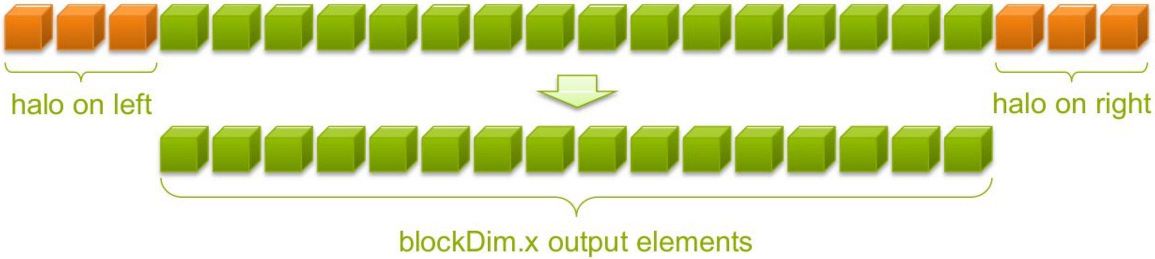


1D stencil

- Each thread processes one output element
- Input elements are read several times
  ➜ With radius 3, each input element is read seven times



- Per every block of threads:
  1. Read (**`blockDim.x + 2 * radius`**) input elements from global memory to shared memory
  2. Compute **`blockDim.x`** output elements and write to memory



1D stencil

Using `stencil_1d.cu`

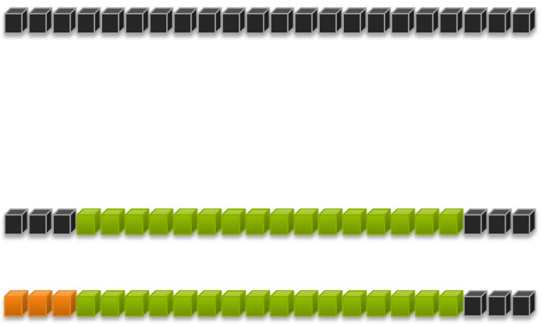
```
[...]

// CUDA kernel to perform 1d stencil summation
__global__ void stencil(const int* V, int* R, const int size, const int radius) {

   // Declare shared array of elements for the block
   // accounting for the two side radius
   __shared__ int tmp[THREADS_PER_BLOCK + 2 * RADIUS];

   // Calculate the global thread ID of the active kernel
   int g_idx = blockIdx.x * blockDim.x + threadIdx.x;

   // Calculate the indexes of the shared elements
   int s_idx = threadIdx.x + radius;

[...]
```

1D stencil

```
[...]

    // Fill the shared memory
    //
    // Copy an element from the global memory to the shared memory
    tmp[s_idx] = V[g_idx];
    // Check it the thread local index within its block
    // (threadIdx.x) is less than the radius ("left border")
    if (threadIdx.x < radius) {
        // Load elements that are radius positions to the left
        // of the current element in the global memory vector V
        tmp[s_idx - radius] = V[g_idx - radius];
        // Load element that is blockDim.x positions to the right
        // of the current element into shared memory
        // (also fill up the "right border")
        tmp[s_idx + blockDim.x] = V[g_idx + blockDim.x];
    }

[...]
```

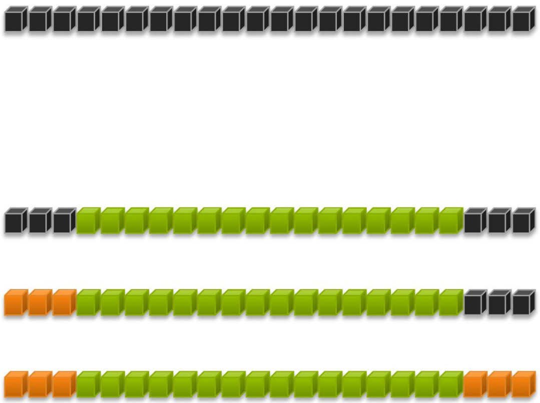
1D stencil

```
[...]

    // Fill the shared memory
    //
    // Copy an element from the global memory to the shared memory
    tmp[s_idx] = V[g_idx];
    // Check it the thread local index within its block
    // (threadIdx.x) is less than the radius ("left border")
    if (threadIdx.x < radius) {
        // Load elements that are radius positions to the left
        // of the current element in the global memory vector V
        tmp[s_idx - radius] = V[g_idx - radius];
        // Load element that is blockDim.x positions to the right
        // of the current element into shared memory
        // (also fill up the "right border")
        tmp[s_idx + blockDim.x] = V[g_idx + blockDim.x];
    }

[...]
```



1D stencil

```
[...]

    // Fill the shared memory
    //
    // Copy an element from the global memory to the shared memory
    tmp[s_idx] = V[g_idx];
    // Check it the thread local index within its block
    // (threadIdx.x) is less than the radius ("left border")
    if (threadIdx.x < radius) {
        // Load elements that are radius positions to the left
        // of the current element in the global memory vector V
        tmp[s_idx - radius] = V[g_idx - radius];
        // Load element that is blockDim.x positions to the right
        // of the current element into shared memory
        // (also fill up the "right border")
        tmp[s_idx + blockDim.x] = V[g_idx + blockDim.x];
    }

[...]
```



1D stencil

```
[...]

   // Synchronize (ensure all the data is available)
   __syncthreads();

   // Apply the stencil
   int result = 0;
   for (int offset = -radius ; offset <= radius ; offset++) {
       result += tmp[s_idx + offset];
   }

   // Store the result
   R[g_idx] = result;
}
```

`__syncthreads()`

Guarantees that all threads have completed filling up the shared memory before proceeding with the computation

➡ Avoids race conditions!

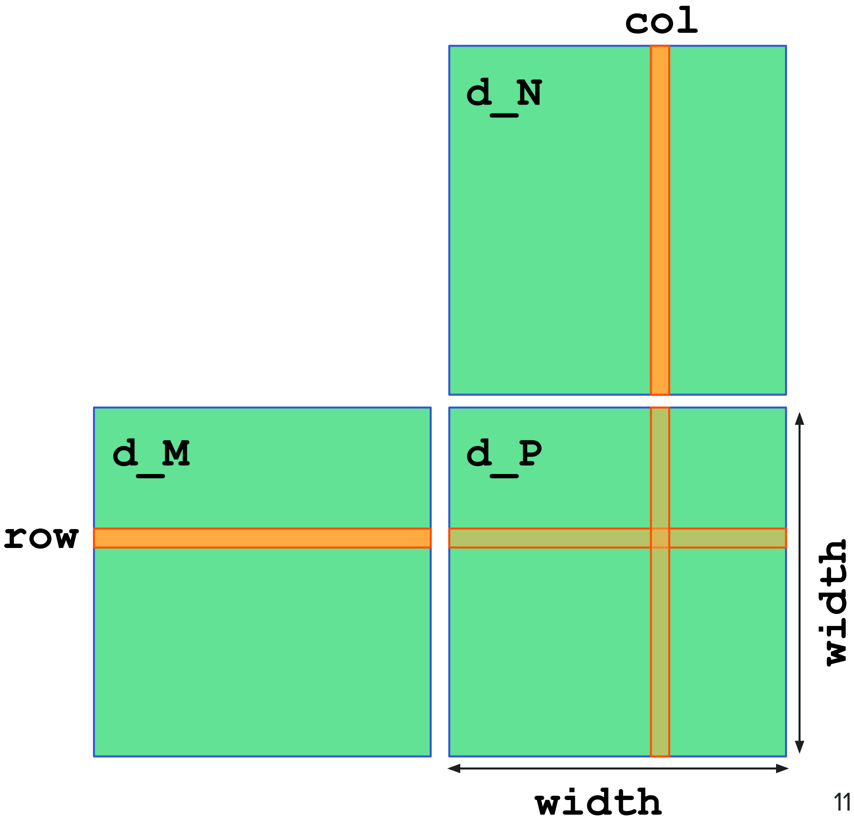
Tiled Matrix-Matrix multiplication

Shared memory is limited (typically around 64KB)

It's impossible to share large amounts of data, but it's possible to break down the computation in such a way that we may only need to store a smaller amount of data at a time.

Getting back to matrix-matrix multiplication, in combination with the usage of shared memory

⇒ refactor the CUDA thread blocks to compute all elements of a given N_{Block}xN_{Block} subset of matrix `P`



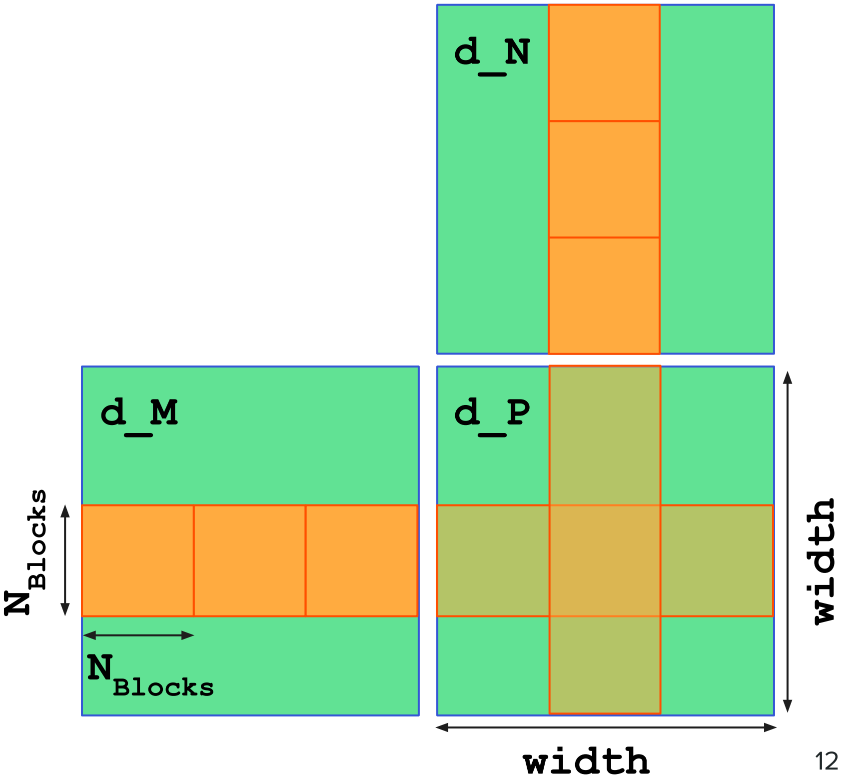
Tiled Matrix-Matrix multiplication

Shared memory is limited (typically around 64KB)

It's impossible to share large amounts of data, but it's possible to break down the computation in such a way that we may only need to store a smaller amount of data at a time.

Getting back to matrix-matrix multiplication, in combination with the usage of shared memory

⇒ refactor the CUDA thread blocks to compute all elements of a given $N_{Block}xN_{Block}$ subset of matrix $\mathbb{P}$



Tiled Matrix-Matrix multiplication

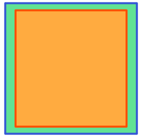
Using `matrix_multiplication_2d_tiled.cu`

```
// CUDA kernel to perform matrix multiplication
__global__ void matrixMultiplication(const float* M,
                                     const float* N,
                                     float* P,
                                     const int width) {
   // Declare shared matrices of size block*block (tiles)
   __shared__ float M_tile[TILE_WIDTH][TILE_WIDTH];
   __shared__ float N_tile[TILE_WIDTH][TILE_WIDTH];

   // For the sake of simplifying the notation
   // assign registers for thread_x,y
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // Calculate the row and column index of the current element
   int row = blockIdx.y * TILE_WIDTH + ty;
   int col = blockIdx.x * TILE_WIDTH + tx;

   // Initialize the intermediate P value
   float sum = 0.;

[...]
```

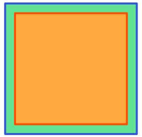
Tiled Matrix-Matrix multiplication

```
[...]

    // Fill the shared memory
    //
    // Loop over the tiles of the input matrices
    for (int t = 0; t < width / TILE_WIDTH; ++t) {
        // Load (in collaboration with other threads)
        // the tiles into shared memory
        if ( (row < width) && (t * TILE_WIDTH + tx < width) )
            M_tile[ty][tx] = M[row * width + t * TILE_WIDTH + tx];
        else
            M_tile[ty][tx] = 0.;

        if ( (t * TILE_WIDTH + ty < width) && (col < width) )
            N_tile[ty][tx] = N[(t * TILE_WIDTH + ty) * width + col];
        else
            N_tile[ty][tx] = 0.;

        // Synchronize (ensure the tile is loaded in shared memory)
        __syncthreads();

    [...]
```

N_tile

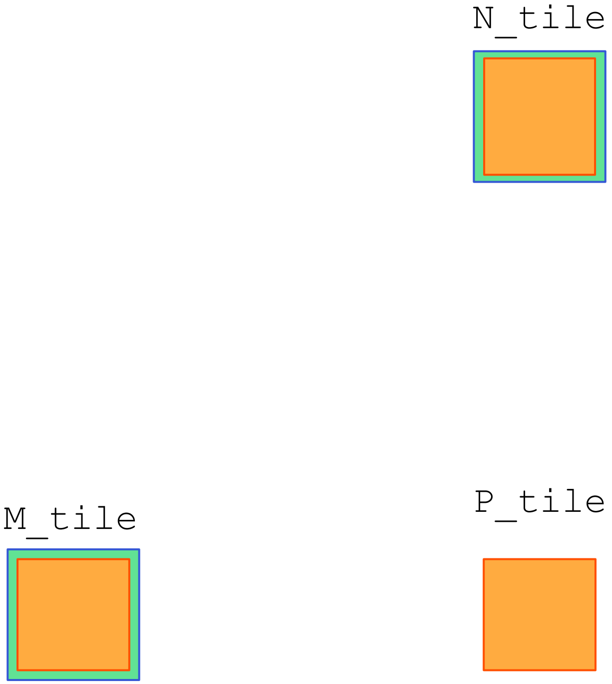


M_tile



Tiled Matrix-Matrix multiplication

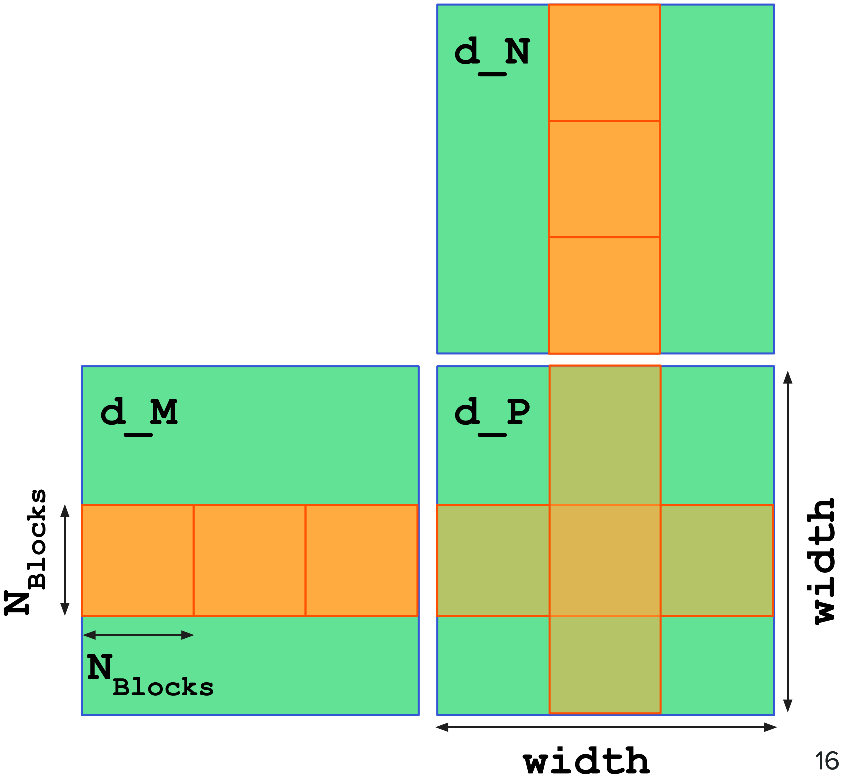
```
    [...]

        // Perform the multiplication for this tile
        for (int k = 0; k < TILE_WIDTH; ++k) {
            sum += M_tile[ty][k] * N_tile[k][tx];
        }

        // Ensure all threads are done computing
        // before loading the next tile
        __syncthreads();
    }

    // Write the result back to the global memory
    if (row < width && col < width) {
        P[row * width + col] = sum;
    }
}
```



➜ continue the loop for all tiles before writing the result to global memory

Tiled Matrix-Matrix multiplication

```
    [...]

        // Perform the multiplication for this tile
        for (int k = 0; k < TILE_WIDTH; ++k) {
            sum += M_tile[ty][k] * N_tile[k][tx];
        }

        // Ensure all threads are done computing
        // before loading the next tile
        __syncthreads();
    }

    // Write the result back to the global memory
    if (row < width && col < width) {
        P[row * width + col] = sum;
    }
}
```



Work at home/lab

- Smoothing of 1D function with Gaussian-like kernels
- 2D convolution of image with blur/edge detection filters

```
[0.1, 0.15, 0.4, 0.15, 0.1]
```

Ridge or edge detection	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$

Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$
Gaussian blur 5 × 5 (approximation)	$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$

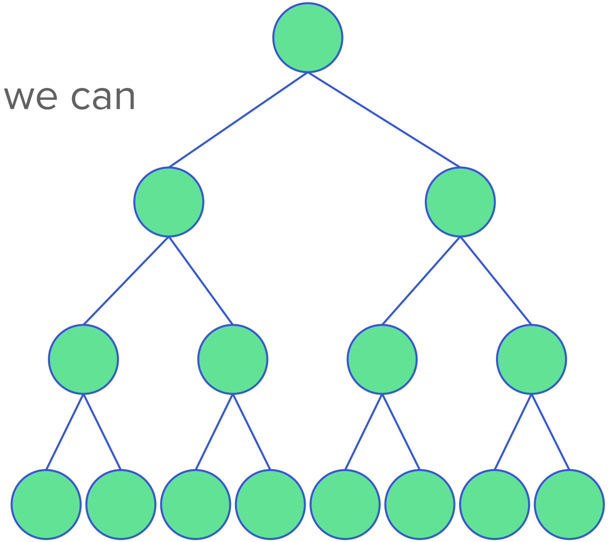
Parallel Reduction

Reduction algorithms are something we already discussed in MAPD-B…
It's a commonly used strategy for processing large input data sets (see MapReduce)

A reduction algorithm derives a single value from an array of input values

Apart from the "standard' functions (e.g. sum, max, min, product) we can define custom reductions operators, which have to respect the requirements of:

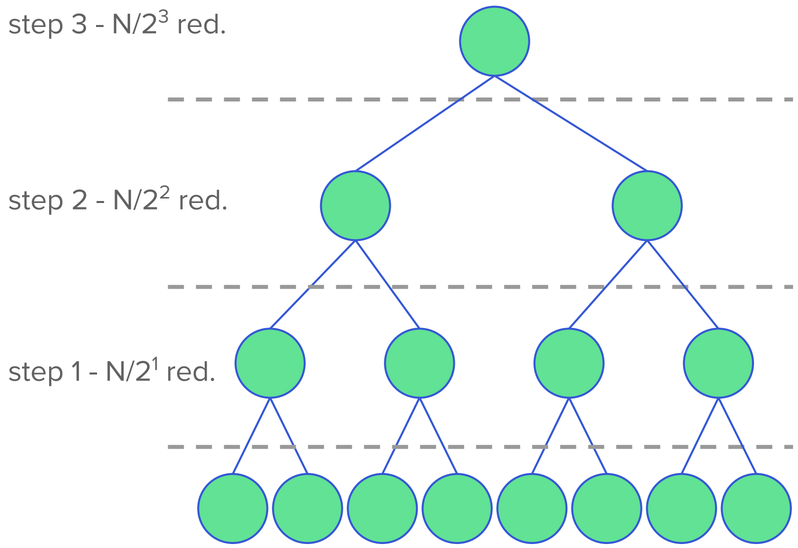
- being a binary associative and commutative operator
- having a well-defined identity (e.g. 0 for the sum)



Parallel Reduction

Given N inputs, a parallel reduction algorithms:

- with every step, halves the number of elements
- “visits” every input value only once
- takes N-1 reduction operations in total
- takes *log(N)* steps to complete



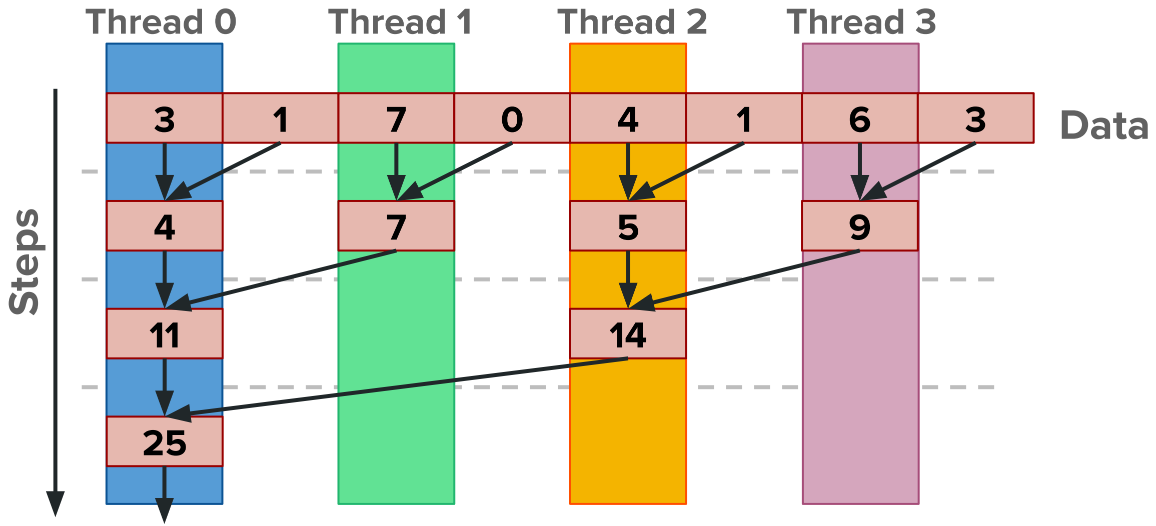
A parallel implementation should include:

1. An initialization of the result as the identity value for the reduction operation
2. The iteration through the inputs to perform the reduction operation

Parallel Reduction

A naive thread-to-data mapping reduction algorithm:

- At step 0, each thread (½ threads as input elements) is responsible for the reduction
- After each step, ½ of the threads are no longer required (are they in the same block/warp?)
- One of the inputs is always from the location of responsibility
- In each step, one of the inputs comes from an increasing distance away



Parallel Reduction

Using `reduction_naive.cu`

```
#define WIDTH 2048                              // Define the vector width
#define N_BLOCKS  256                           // Define the number of blocks
#define THREADS_PER_BLOCK WIDTH/N_BLOCKS/2      // Define the number of threads in a block

[...]

// CUDA kernel to perform parallel reduction
__global__ void reduction_naive(const int* V, int* R, const int width) {
   // Local index (within block) of thread
   int bdim = blockDim.x;
   int bx = blockIdx.x;
   int tx = threadIdx.x;

   // Global index of the data element
   int start_idx = 2 * bdim * bx; // 2x as we are launching 1 thread every 2 items

   // Shared memory to store partial sums
   // Limited to block, thus overall size 2x
   __shared__ int partialSum[2 * THREADS_PER_BLOCK];

[...]
```

½ threads as input elements

Parallel Reduction

```
[...]

  // Fill partial sum with two elements:
  // - the one corresponding to the thread
  // - the one 1 block size away from it
  partialSum[tx] = V[start_idx + tx];
  partialSum[tx + bdim] = V[start_idx + tx + bdim];

  // Loop over the shared memory doubling the stride
  // and sum values in place
  for (int stride = 1; stride <= bdim; stride *= 2) {

      // Ensure all elements of partial sums have been
      // generated before proceeding to the next step
      __syncthreads();

      // If the thread is active at this step, sum
      if (tx % stride == 0)
          partialSum[2 * tx] += partialSum[2 * tx + stride];
  }

[...]
```

Parallel Reduction

```
[...]

    // Ensure all threads are done computing before
    // offloading the result to global memory
    __syncthreads();
    // Write the result of this block to the output
    if (tx == 0) {
        R[blockIdx.x] = partialSum[0];
    }
}
```

Synchronization is possible only at the level of 1 block

Across blocks there is no real synchronization

➡ 1 output per block from this naive implementation

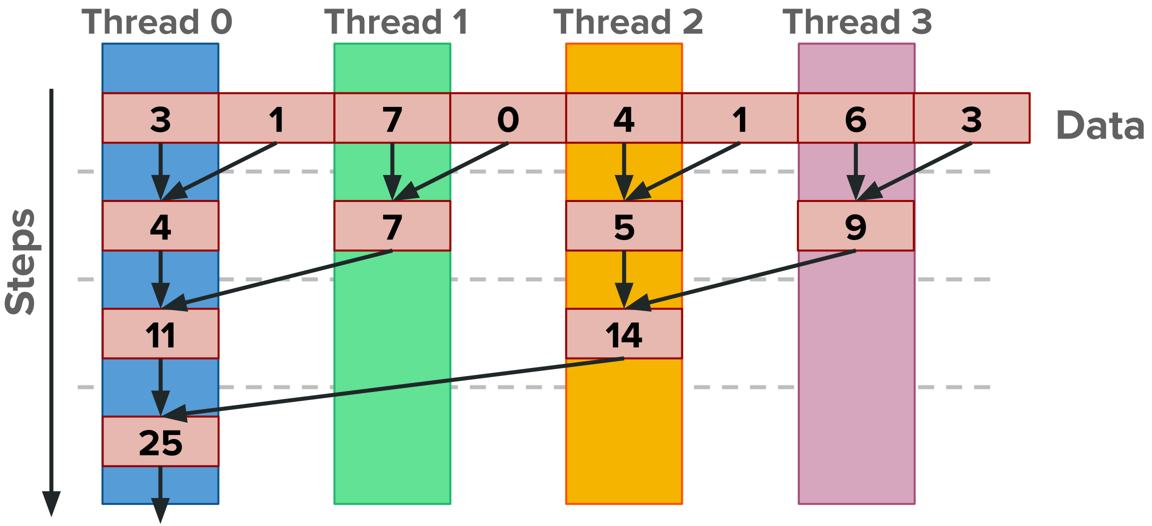
⇒ either use CPU for the final aggregation, or use another kernel to do it on the GPU

Parallel Reduction

What's wrong with the naive implementation?

- In each step, there is going to be some thread that do work, and some that are idle
  ➜ very inefficient as threads that don't work still consume resources (ALUs, registers)
- Already after the 1st step, ½ of the threads are "dead"
- After the 5th step, entire warps may stop working… very inefficient

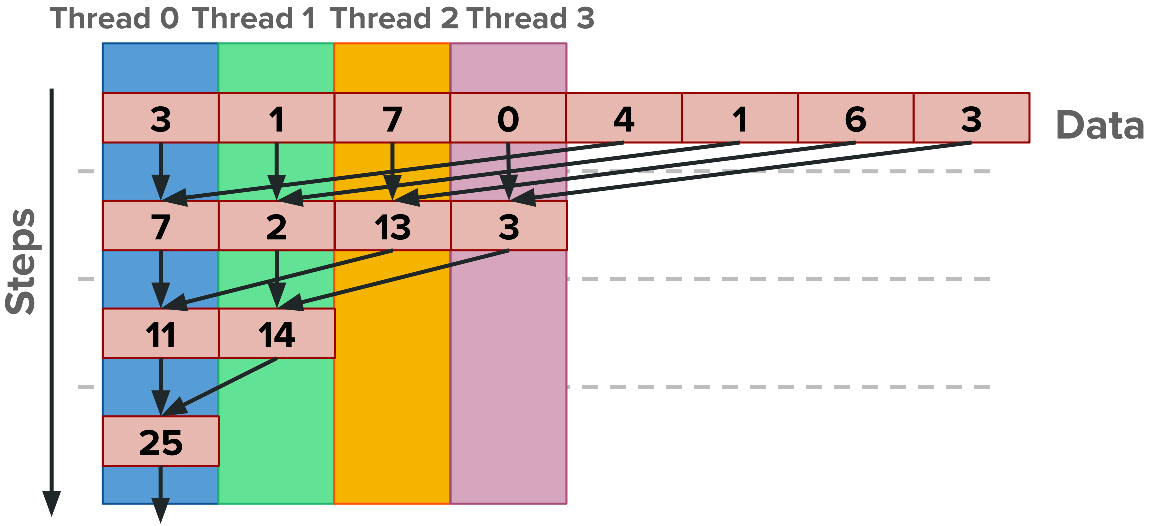
➜ this is referred to as a "divergent execution"



Parallel Reduction

How can we improve the situation?

- Change the indexing to better utilize the available resources
- Compact the partial sums into the front locations in the `partialSum[ ]` array
- Keep the active threads consecutive



Work at home/lab

- implement the improved reduction algorithm