TDT4200 Parallel programming

PS5

Maren Wessel-Berg & Claudi Lleyda Moltó

October 2023

Practical information

Published: 10/10/23

Deadline: 24/10/23 at 22:00

Evaluation: Pass/Fail

- Completing the problem set is **mandatory**.
- The work must be done **individually** and without help from anyone but the TDT4200 staff.
- **Reference** all sources found on the internet or elsewhere.
- The **requirements**, and **how and what to deliver** is explained in the problem set description found on BlackBoard.
- **Start early!**

Where can you get help with the assignment?

- **Recitation lecture**: introduction to the problem set

  (Today)
  Slides will be made available online.

- **TA hours**: ask questions in person

  Friday, October 13, 10:00–12:00 in Cybele
  Monday, October 16, 13:00–15:00 in Cybele
  Friday, October 20, 10:00–12:00 in Cybele
  Monday, October 23, 13:00–15:00 in Cybele

- **Piazza**: question forum

  Ask questions any time (but give us time to answer).
  Select the ps5 folder for questions related to this problem set.
  Do not post full or partial solutions!

Today

- Introduce the problem set.
- Give an introduction to GPUs and CUDA programming.

  GPUs and GPU programming will be covered in the main lectures, but the assignment schedule is a little ahead of this schedule, so we will provide you with the information you need to solve the assignments in the recitation lectures.

Topic

Finite difference approximation of the 2D heat equation using CUDA

- You will work on the same code as in previous assignments, but this time you will take the sequential implementation of the Finite Difference Method (FDM) for solving the 2D heat equation and **parallelise it using CUDA**
- You will also **answer questions** about your implementation and the curriculum.

GPU vs. CPU

- The CPU must be good at many things.
  - Minimize the latency of a single thread by using caches and compex control flow logic.
- The GPU is optimized for massively parallel computations, i.e., it is specialized.
  - Maximize the throughput of all threads by running many threads in parallel and hiding latency with computations.

CUDA

- Platform and programming model developed by NVIDIA.
- Allows us to program NVIDIA GPUs through programming language extensions.
- Only compatible with NVIDIA GPUs– other GPUs require other programming models, e.g., OpenCL.

We will focus on CUDA in the assignments, but the general GPU concepts do not only apply to NVIDIA GPUs.

We will focus on three main topics

- Memory spaces and memory management.
- Execution spaces and launcing kernels.
- The thread hierarchy and thread identification.

Memory spaces

Host memory and device memory

- The CPU and the GPU have separate memory spaces.
  - We need to manage two different memory spaces.
  - Data is transferred between the GPU and the CPU (over PCIe or NVLink bus).
  - Dereferencing a pointer to GPU memory from the CPU will cause an error (and vice versa).

Memory management

`cudaMalloc`, `cudaMemcpy`, `cudaFree`

- The device memory is managed from the host and is similar to how CPU memory is managed.

Allocate device memory:

```
cudaMalloc ( void** devPtr, size_t size )
```

Memory management

`cudaMalloc`, `cudaMemcpy`, `cudaFree`

- The device memory is managed from the host and is similar to how CPU memory is managed.

Allocate device memory:

```
cudaMalloc ( void** devPtr, size_t size )
```

Copy data between host and device:

```
cudaMemcpy ( void* dst, const void* src,
             size_t count, cudaMemcpyKind kind )
```

Memory management

`cudaMalloc`, `cudaMemcpy`, `cudaFree`

- The device memory is managed from the host and is similar to how CPU memory is managed.

Allocate device memory:

```
cudaMalloc ( void** devPtr, size_t size )
```

Copy data between host and device:

```
cudaMemcpy ( void* dst, const void* src,
             size_t count, cudaMemcpyKind kind )
```

Free device memory

```
cudaFree ( void* devPtr )
```

Memory management

`cudaMalloc`, `cudaMemcpy`, `cudaFree`

- The device memory is managed from the host and is similar to how CPU memory is managed.

Copy data between host and device:

```
cudaMemcpy ( void* dst, const void* src,
             size_t count, cudaMemcpyKind kind )
```

Type of data copy (the cudaMemcpyKind enum)

```
cudaMemcopyHostToDevice = 1
cudaMemcpyDeviceToHost = 2
```

Kernels

- *Kernels* are **functions that run on the GPU** and specify what the threads should do.

  A function is declared as a kernel through the function execution space specifier `__global__`

- A kernel is **executed in parallel by all threads**, i.e., if there are N threads, the kernel is executed N times in parallel.

  The number of threads that should execute a kernel is specified by configuration parameters inside the construct `< < <...> > >` when the kernel is called.

Execution spaces

- In general, we want to distinguish between functions that are to be run on the CPU and functions that are to be run on the GPU.

Function Execution Space Specifiers:

```
// Host and device callable, device executed
__global__ void kernel(Params...) {}

// Device callable, device executed
__device__ void kernel(Params...) {}

 // Host callable, host executed (default)
__host__   void kernel(Params...) {}
```

The thread hierarchy

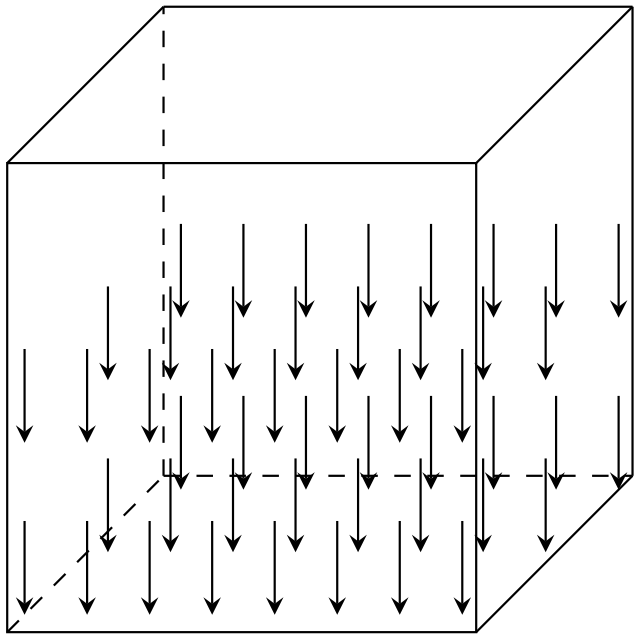
Organisation

Threads are contained within

The thread hierarchy

Organisation

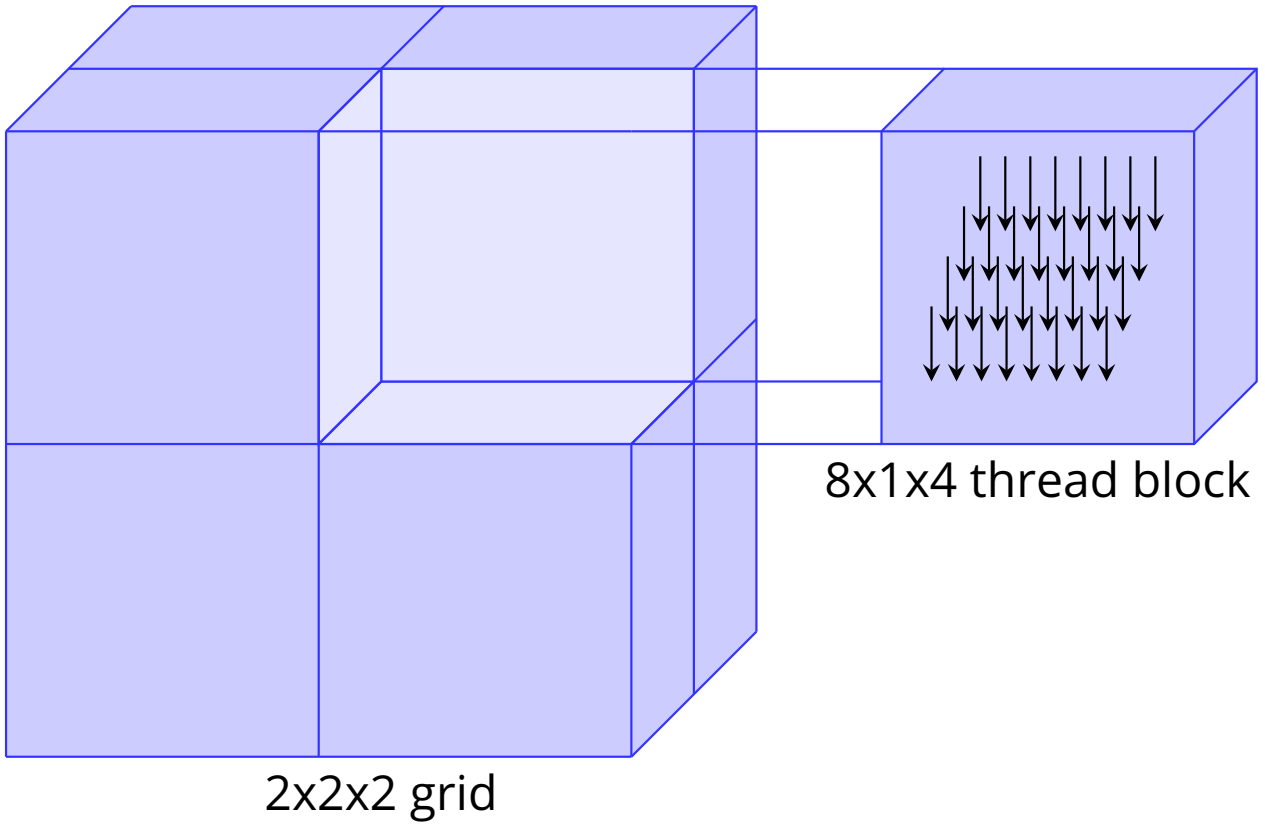
Threads are contained within **blocks** that are organized in a



The thread hierarchy

Organisation

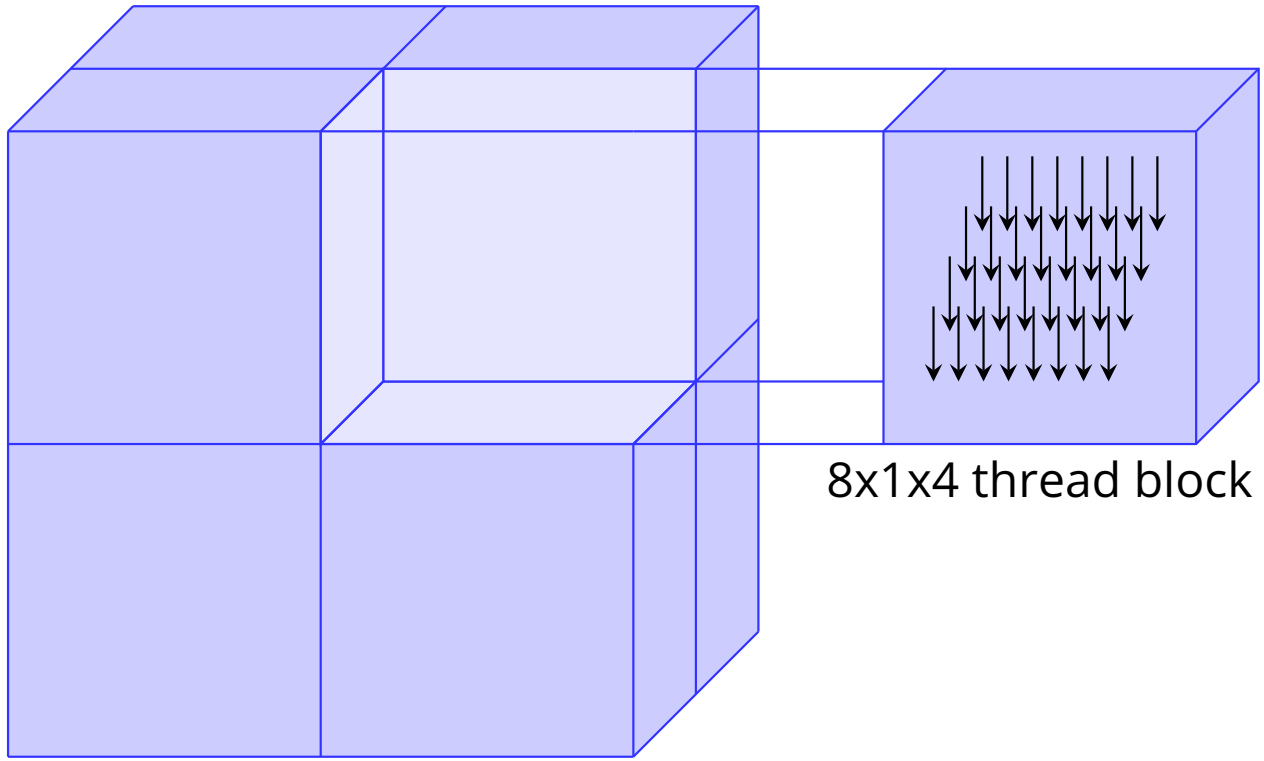
Threads are contained within **blocks** that are organized in a **grid**.



The thread hierarchy

Organisation

Threads are contained within **blocks** that are organized in a **grid**.



Note

A block can hold a maximum of 1024 threads.
A grid can hold a maximum of $2^{31} - 1$ blocks.

The thread hierarchy

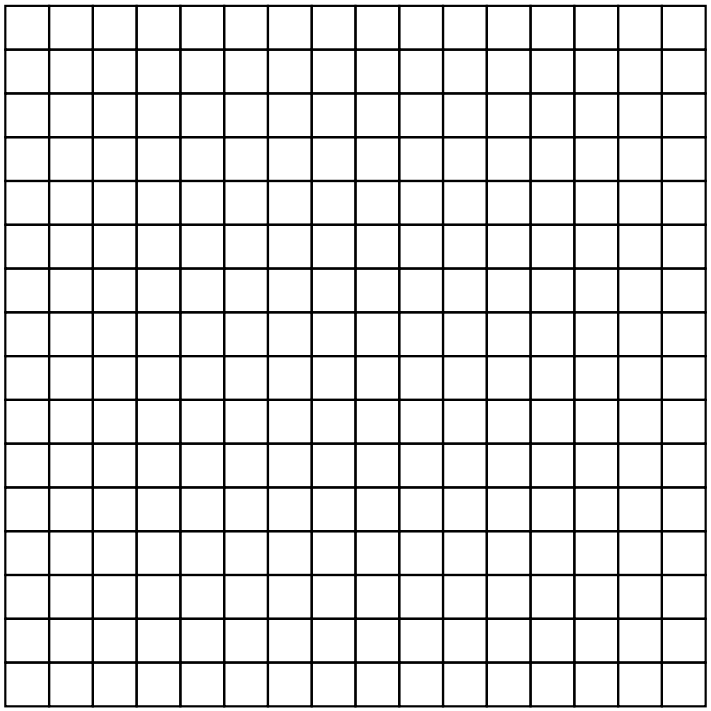
Identification

- A thread that executes a kernel has a **unique ID** that can be accessed from the kernel through built-in variables:

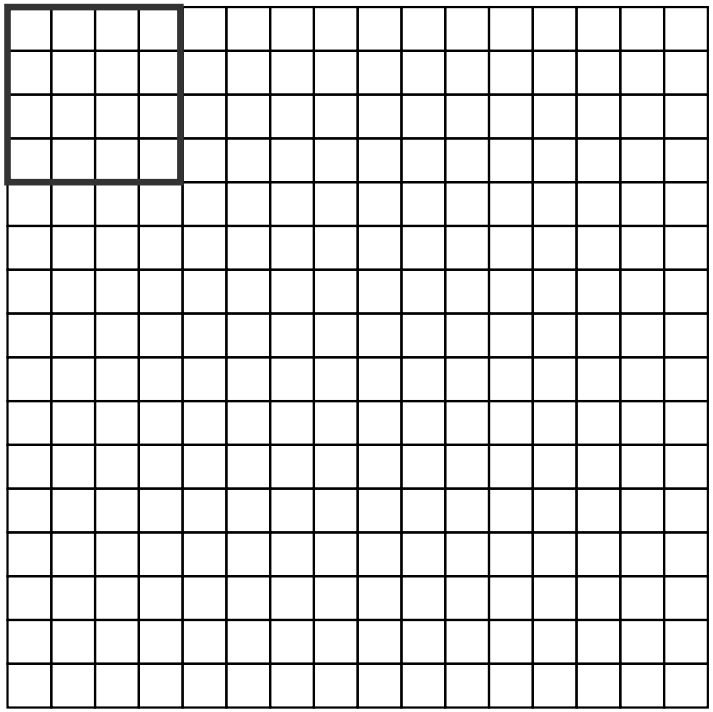
  `threadIdx`

  `blockIdx`

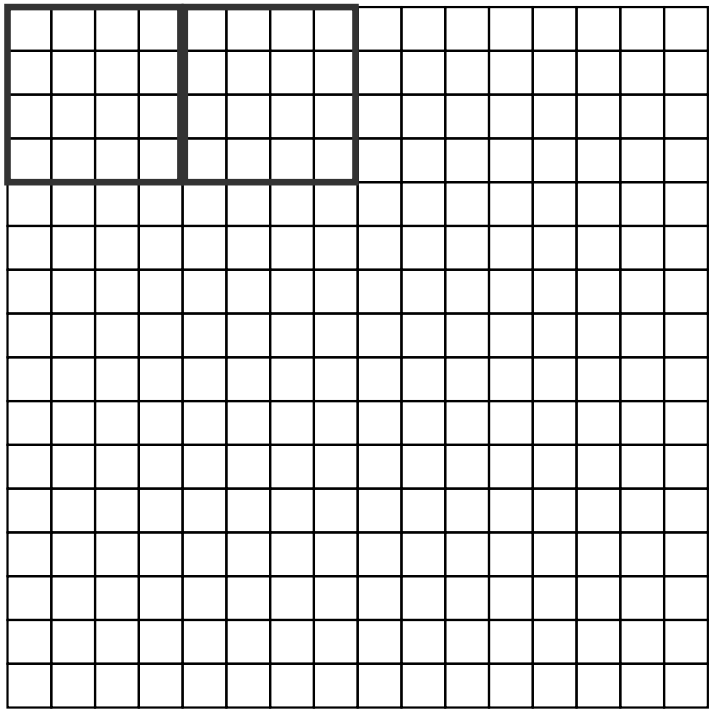
- This can be useful, for example, if we want each thread to operate on different data...



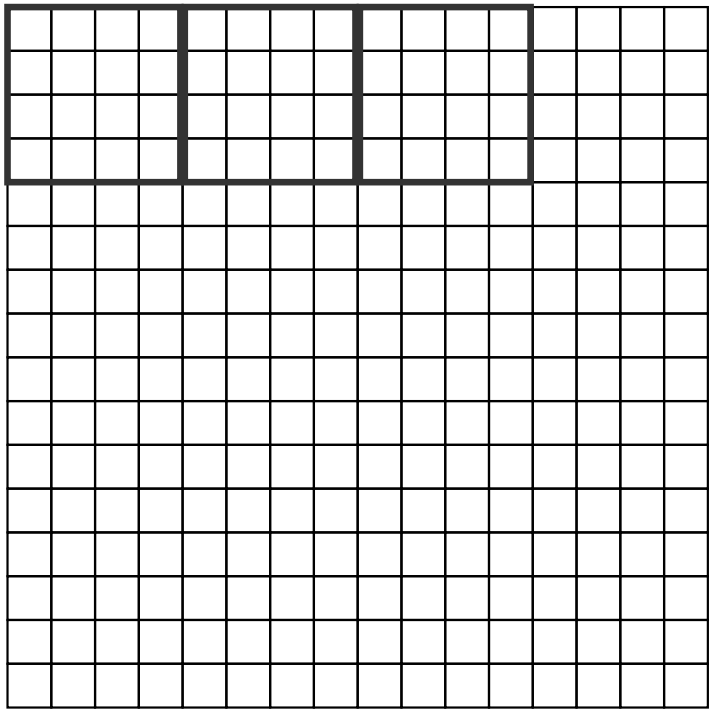
- Build your grid to match the physical domain.
- Determine the size of your thread blocks.
  Example: 4×4



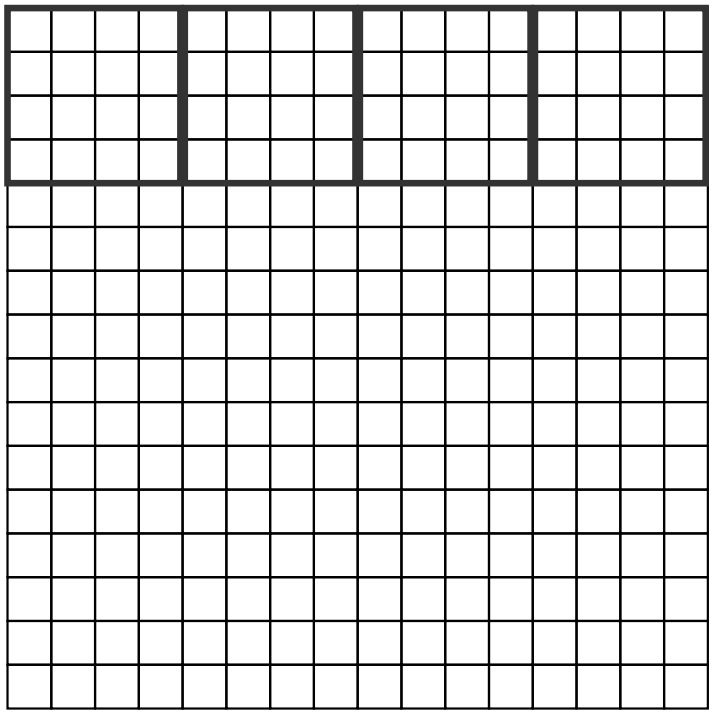
- Build your grid to match the physical domain.
- Determine the size of your thread blocks.
  Example: 4×4



- Build your grid to match the physical domain.
- Determine the size of your thread blocks. **Example**: 4×4

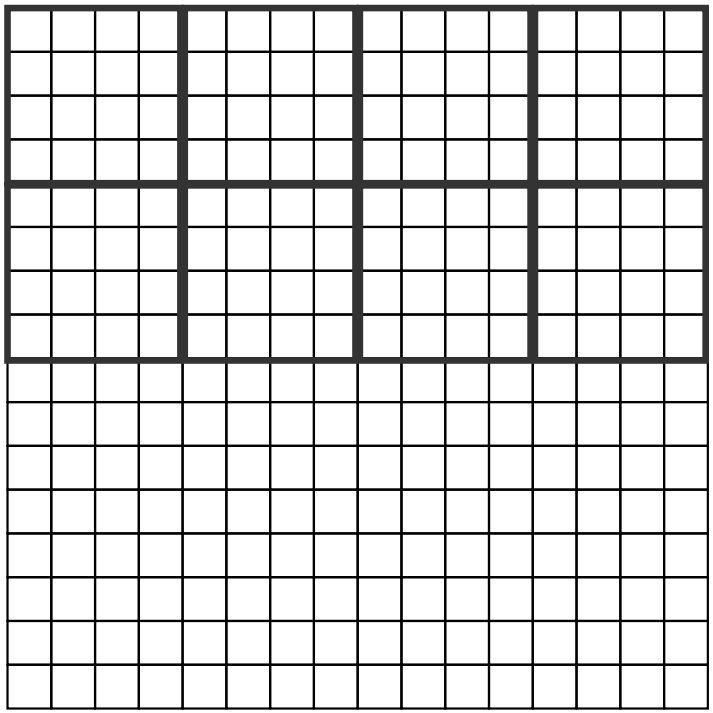


- Build your grid to match the physical domain.
- Determine the size of your thread blocks.
  Example: 4×4



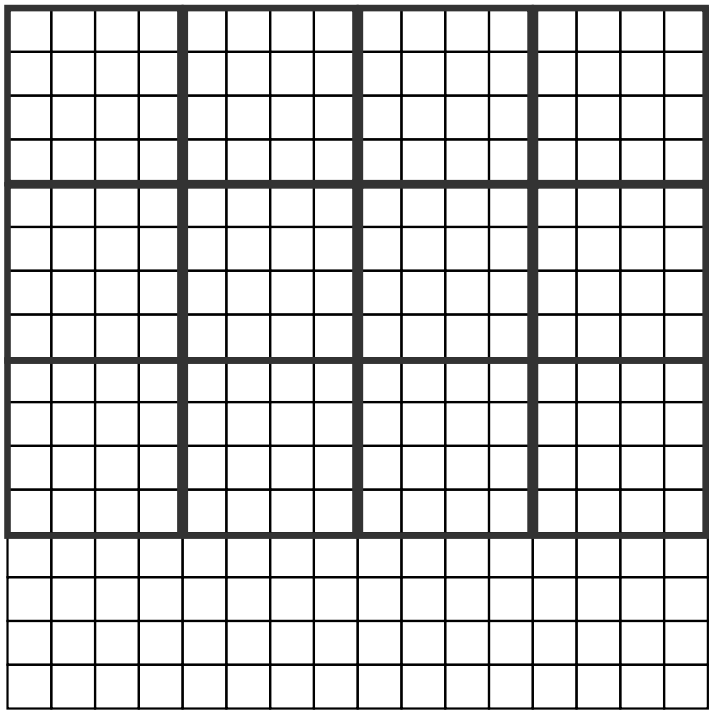
- Build your grid to match the physical domain.
- Determine the size of your thread blocks.
  Example: 4×4

  $grid_x = \lceil n_x/4 \rceil$



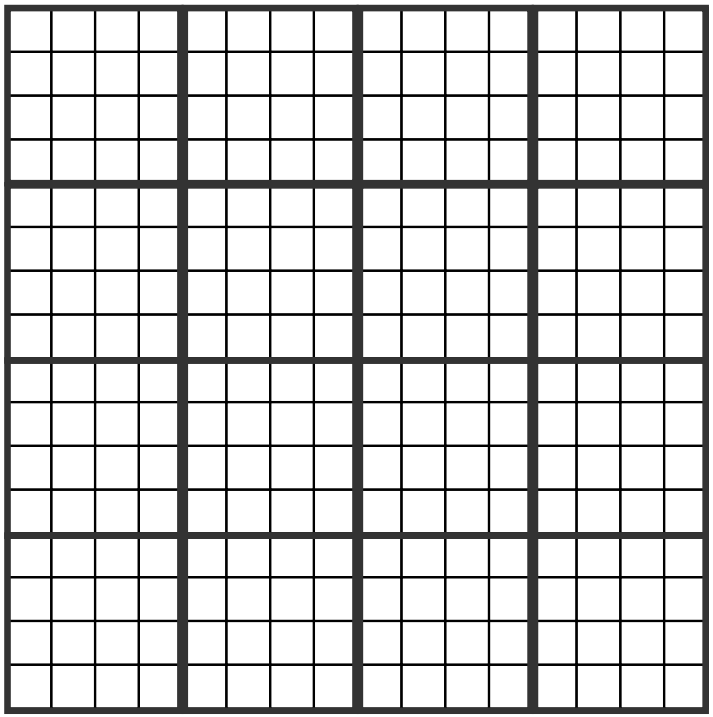
- Build your grid to match the physical domain.
- Determine the size of your thread blocks.
  Example: 4×4

  $grid_x = \lceil n_x/4 \rceil$



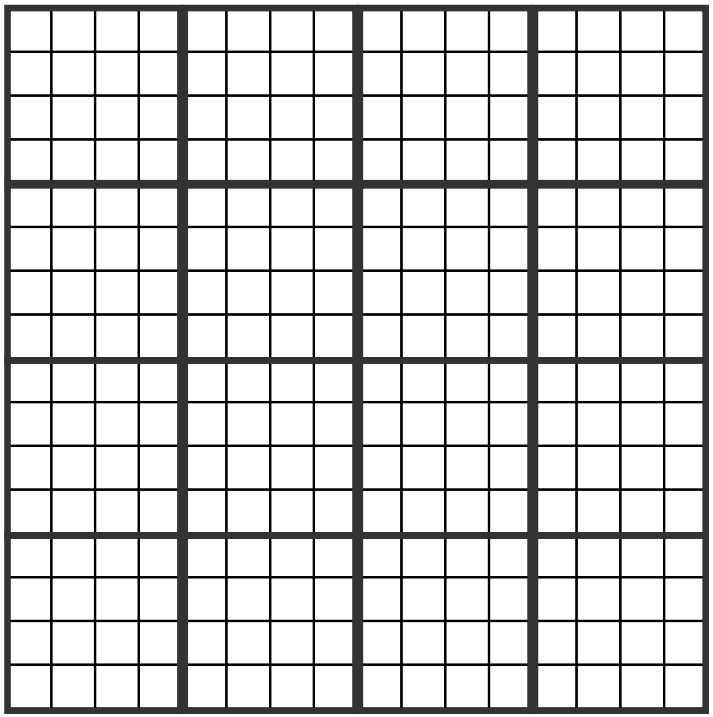
- Build your grid to match the physical domain.
- Determine the size of your thread blocks.
  Example: 4×4

  $grid_x = \lceil n_x/4 \rceil$



- Build your grid to match the physical domain.
- Determine the size of your thread blocks.
  Example: 4×4

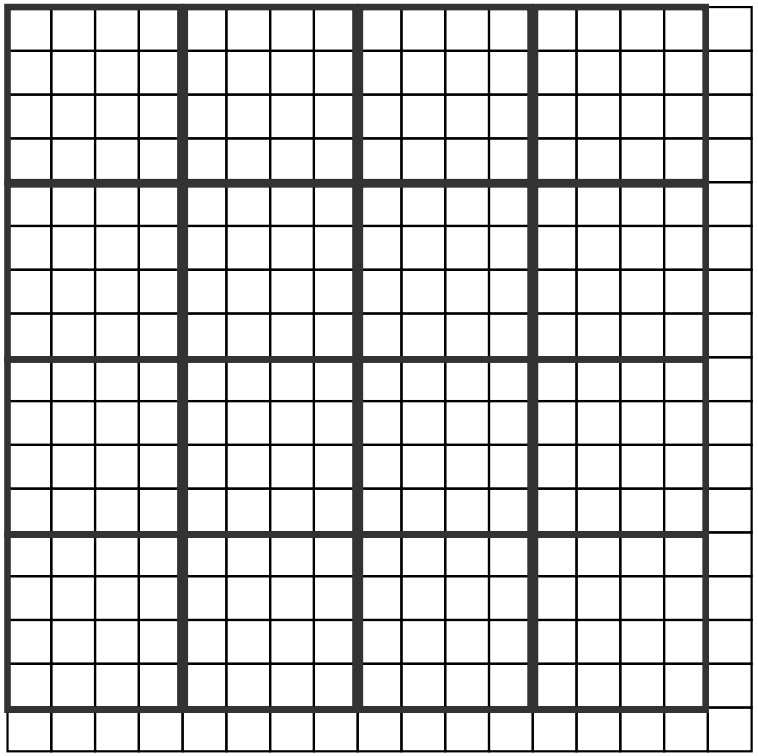
  $$grid_x = \lceil n_x/4 \rceil$$
  $$grid_y = \lceil n_y/4 \rceil$$



- Build your grid to match the physical domain.
- Determine the size of your thread blocks.
  Example: 4×4

  $grid_x = \lceil n_x/4 \rceil$

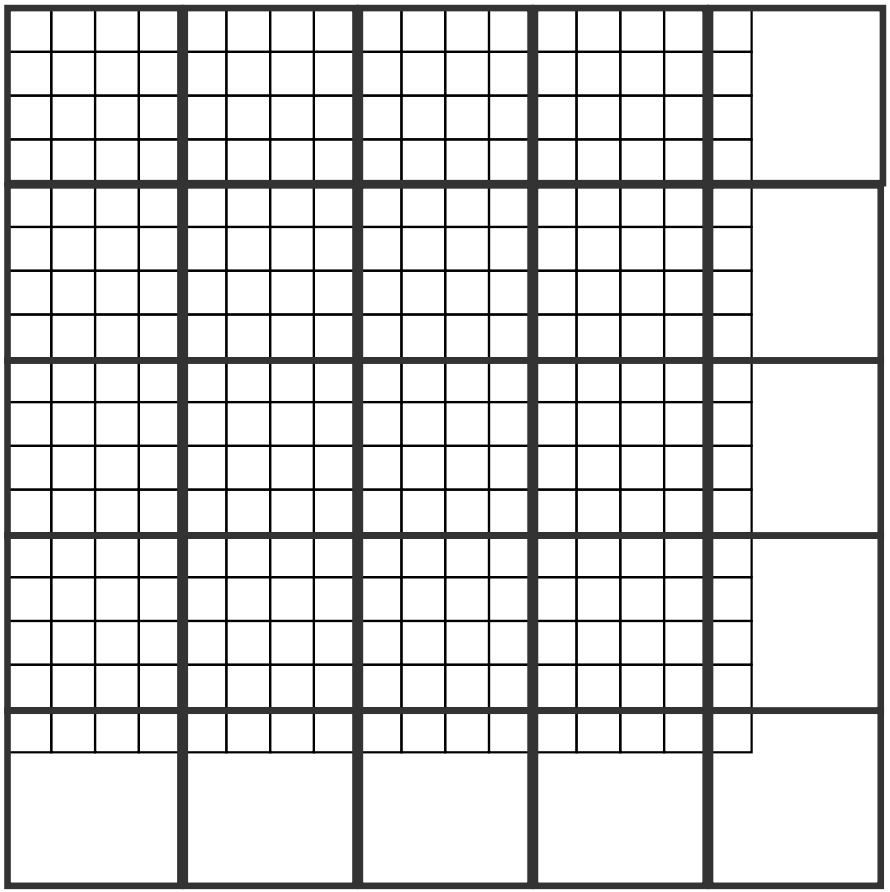
  $grid_y = \lceil n_y/4 \rceil$
- Why should you round up the number of blocks?



- Build your grid to match the physical domain.
- Determine the size of your thread blocks.
  Example: 4×4

  $grid_x = \lceil n_x/4 \rceil$

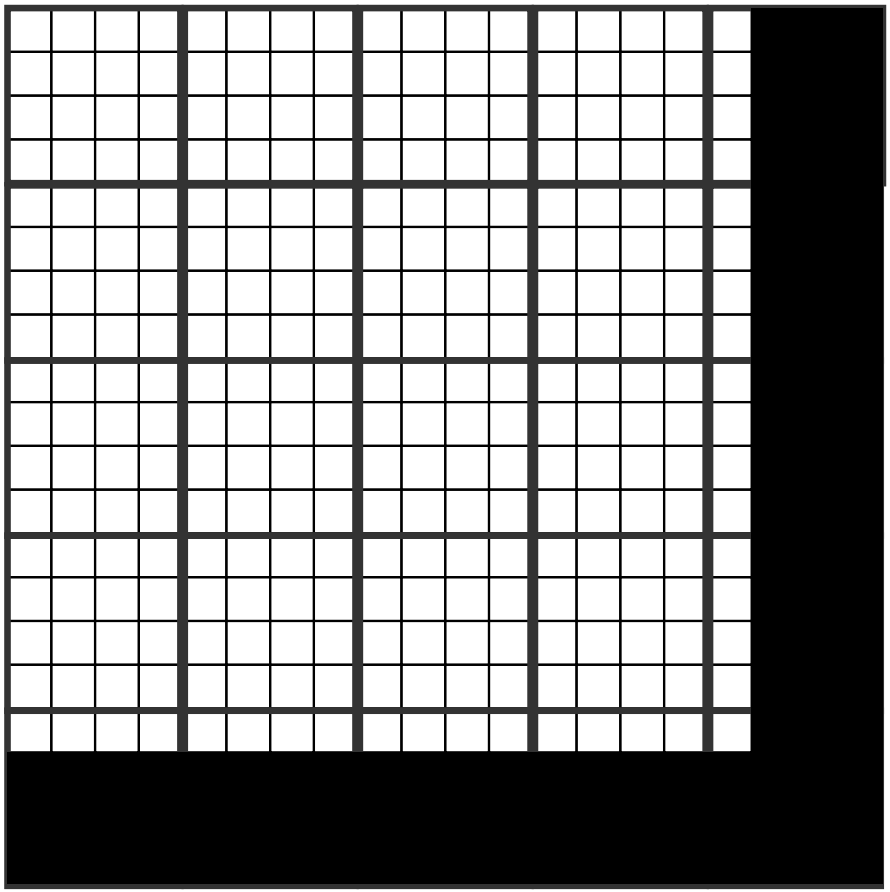
  $grid_y = \lceil n_y/4 \rceil$
- Why should you round up the number of blocks?



- Build your grid to match the physical domain.
- Determine the size of your thread blocks.
  Example: 4×4

  $grid_x = \lceil n_x/4 \rceil$

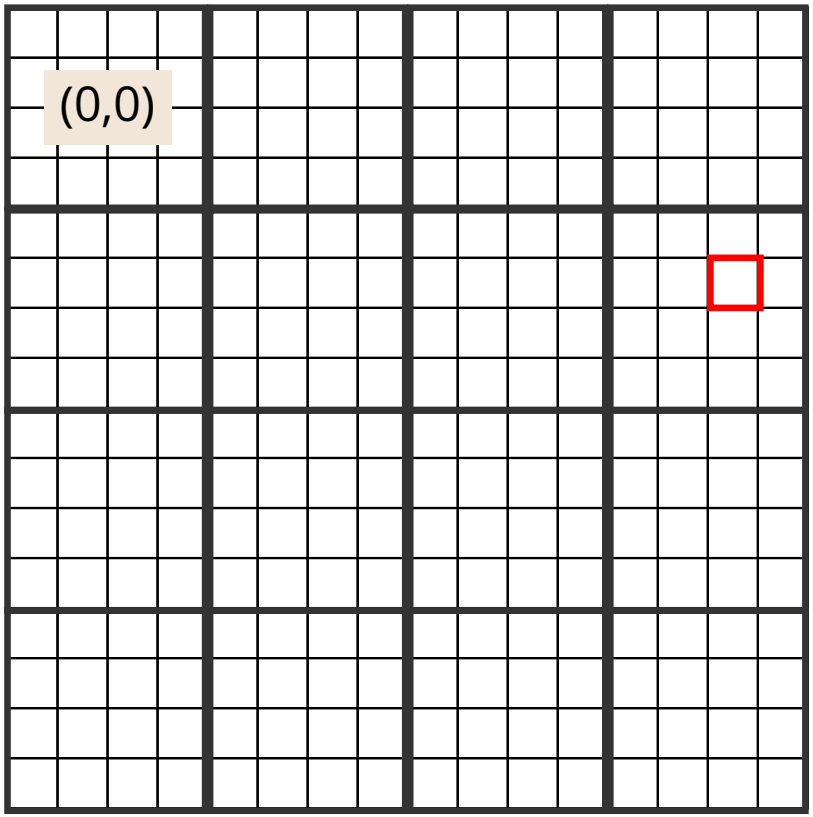
  $grid_y = \lceil n_y/4 \rceil$
- Why should you round up the number of blocks?



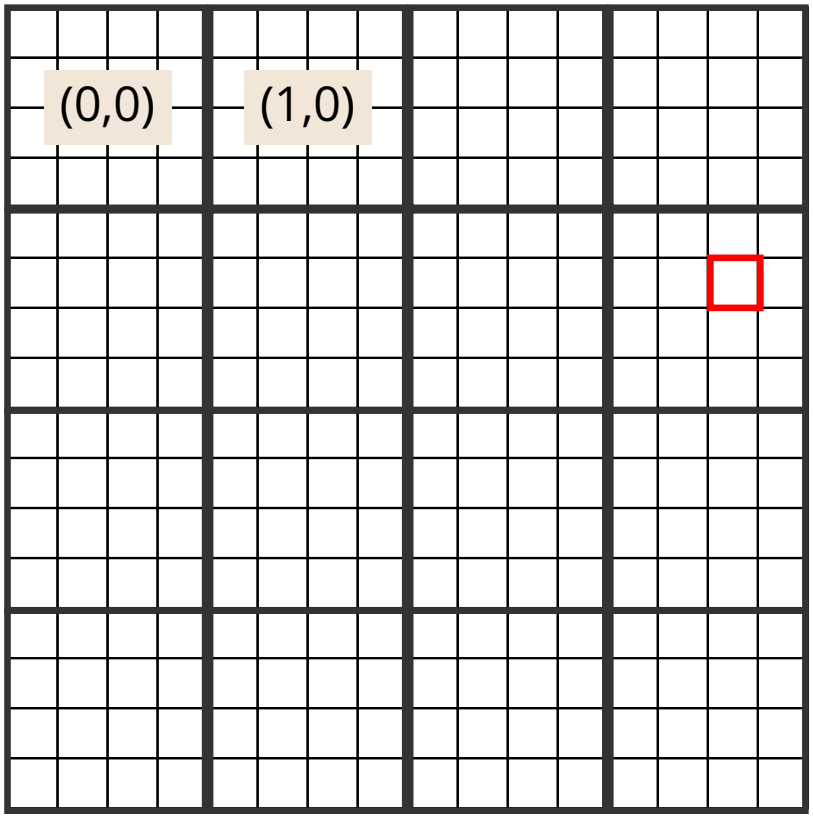
- Build your grid to match the physical domain.
- Determine the size of your thread blocks.
  Example: 4×4

  $grid_x = \lceil n_x/4 \rceil$

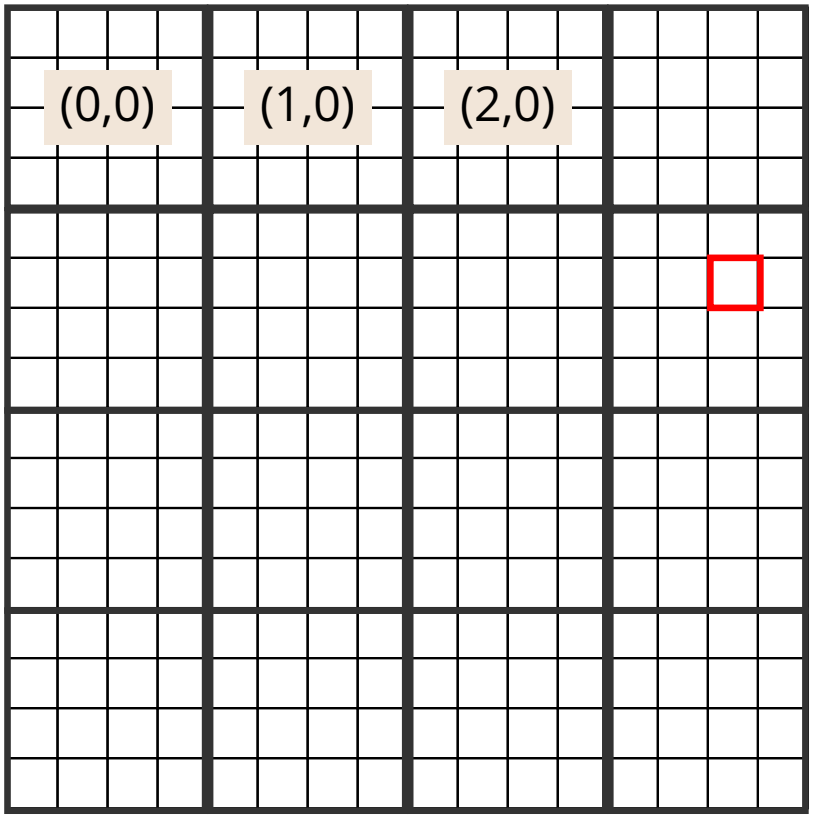
  $grid_y = \lceil n_y/4 \rceil$
- Why should you round up the number of blocks?
- What do you do with threads with an ID "outside" the domain?



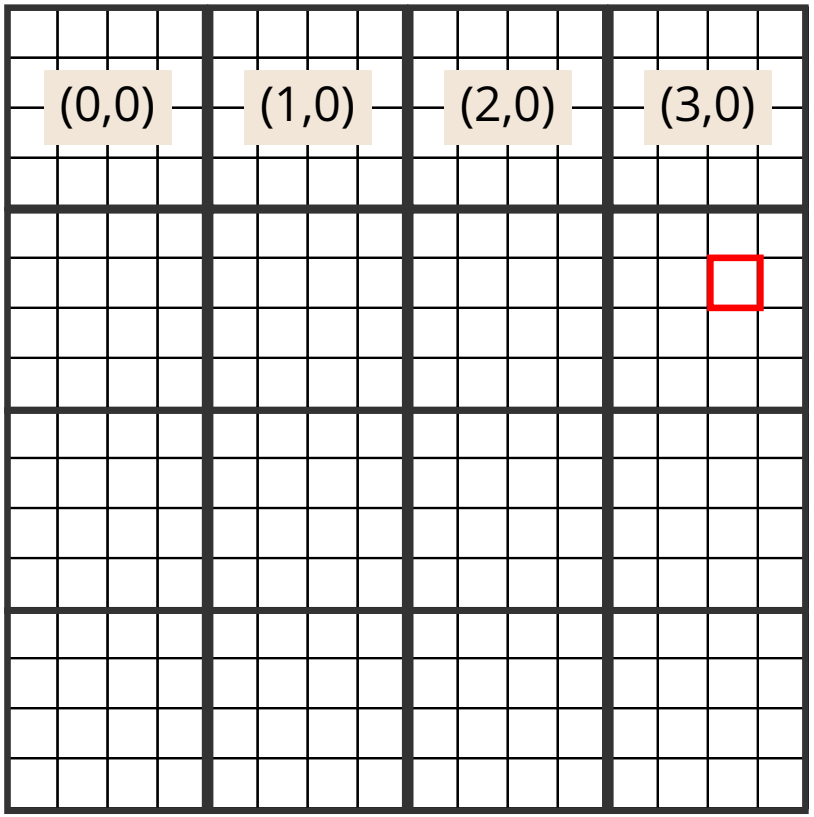
- What thread is this?



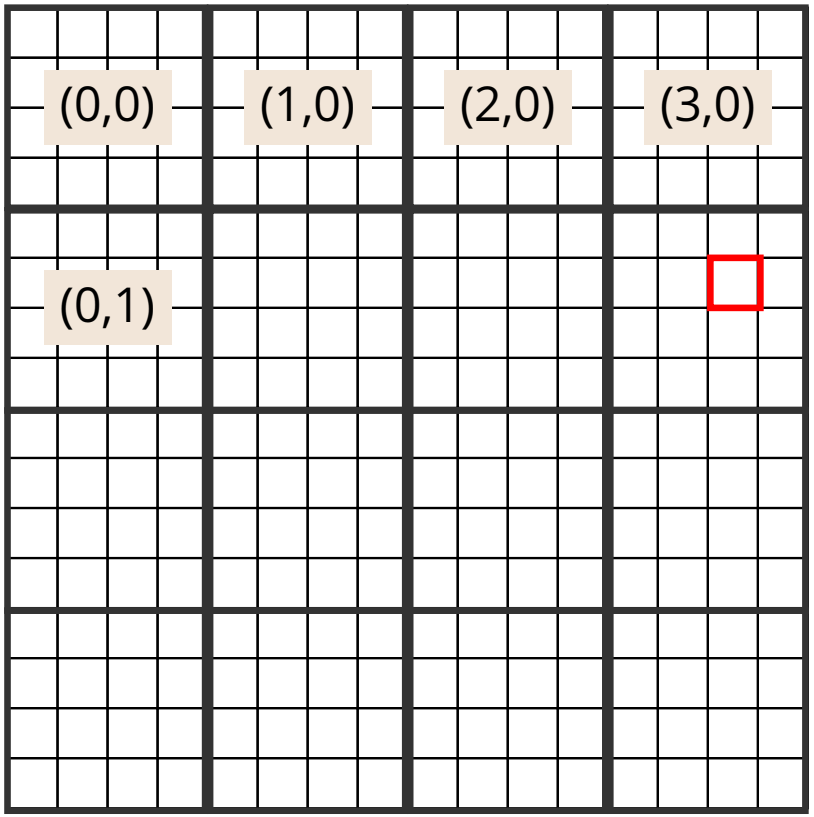
- What thread is this?



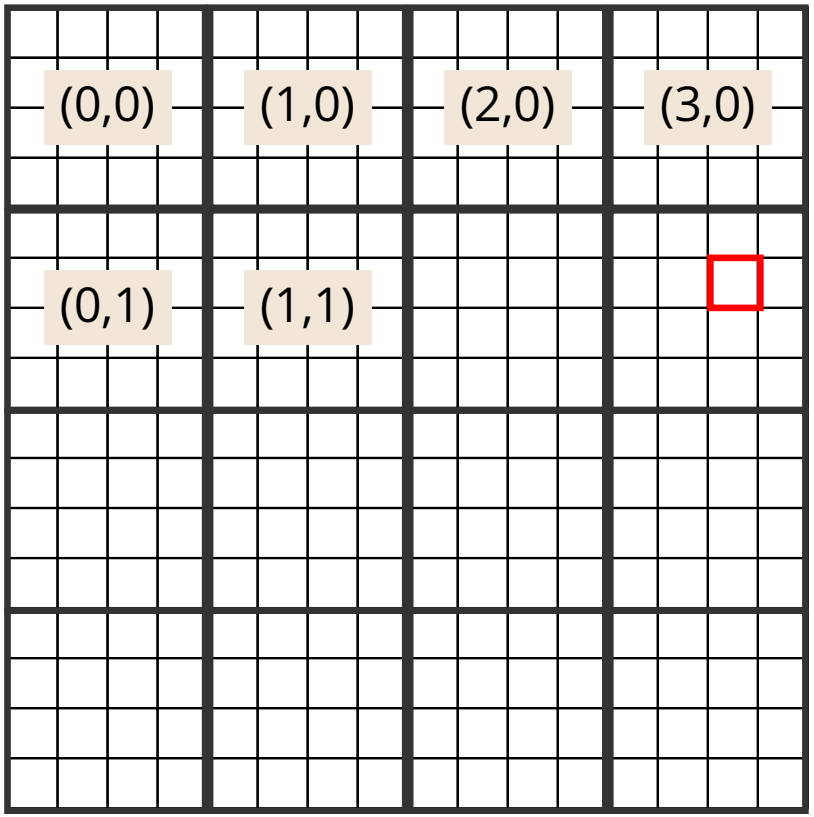
- What thread is this?



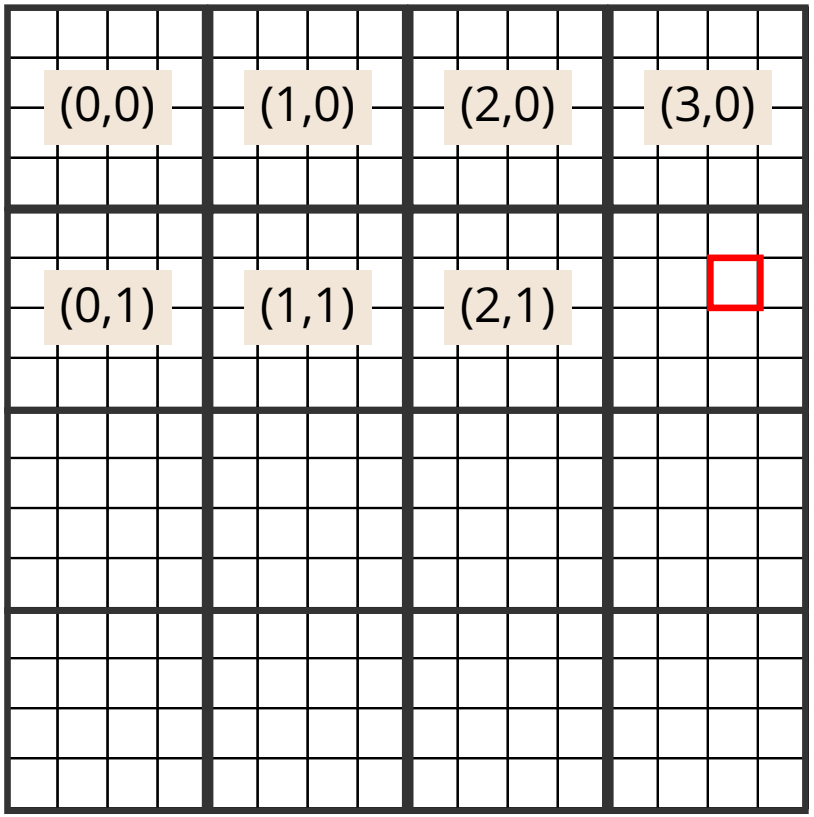
- What thread is this?



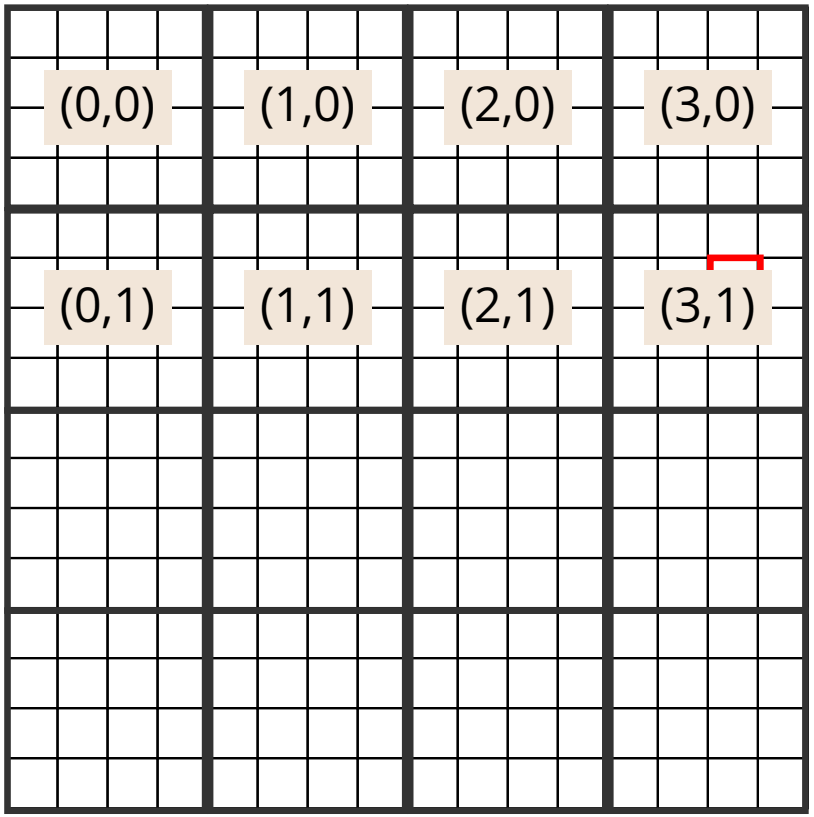
- What thread is this?



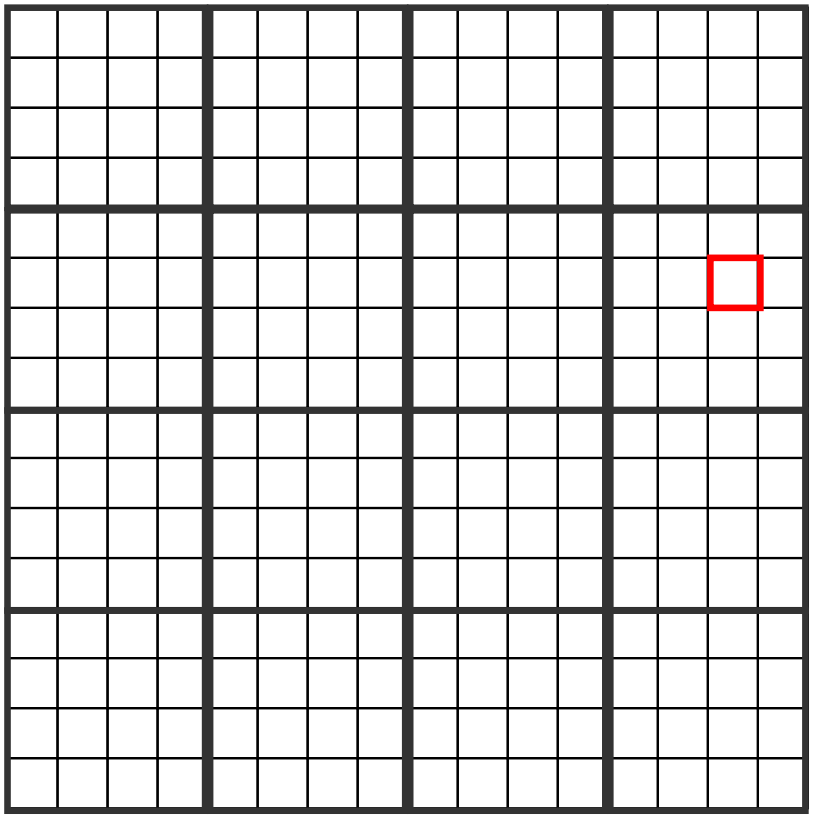
- What thread is this?



- What thread is this?

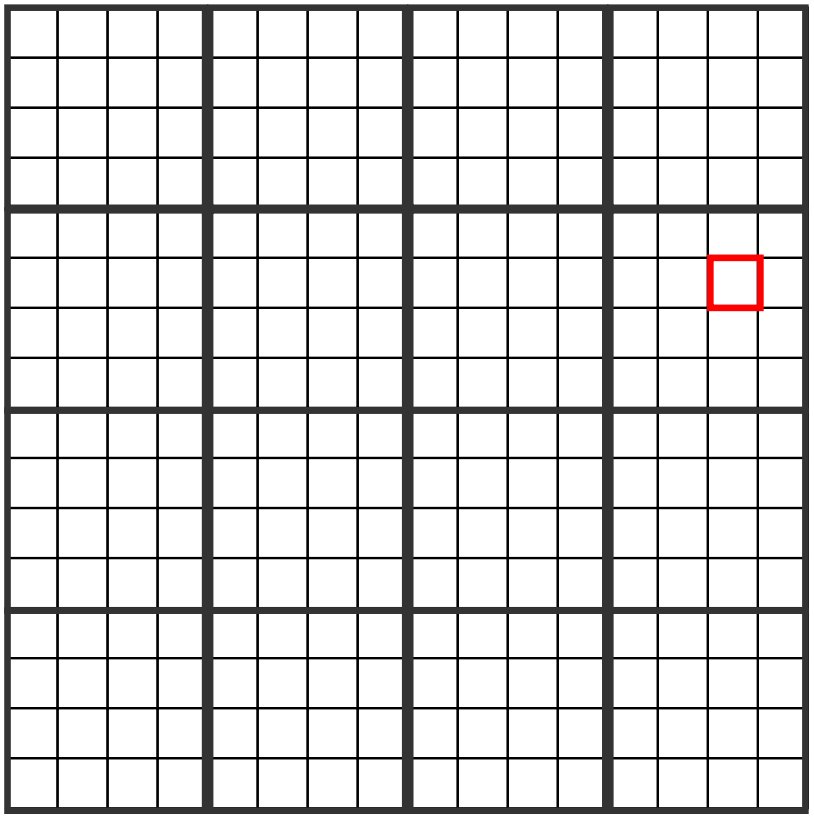


- What thread is this?



- What thread is this?

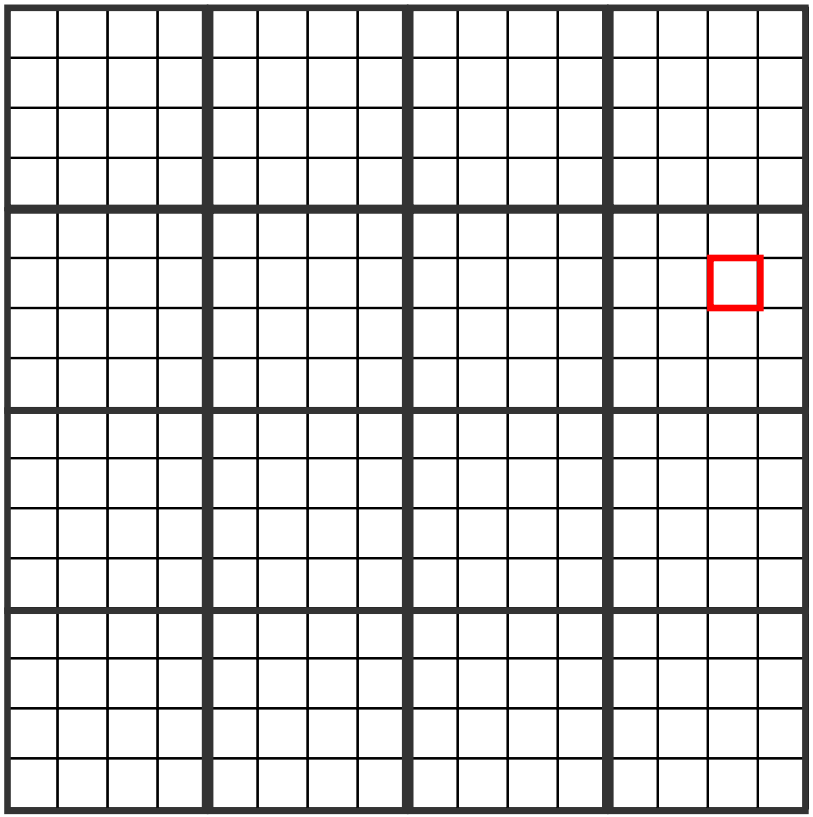
  Thread block: $(3, 1)$



- What thread is this?

  Thread block: $(3, 1)$

  Local thread index: $(2, 1)$



- What thread is this?

  Thread block: $(3, 1)$

  Local thread index: $(2, 1)$

- Use the built-in variables:

```
gridDim.x;     // 4
gridDim.y;     // 4
blockDim.x;    // 4
blockDim.y;    // 4
blockIdx.x;    // 3
blockIdx.y;    // 1
threadIdx.x;   // 2
threadIdx.y;   // 1
```

```
int x = blockDim.x * blockIdx.x + threadIdx.x; // 14
int y = blockDim.y * blockIdx.y + threadIdx.y; // 5
```

The thread hierarchy

Identification

- By using the thread IDs and the calculation on the previous slide we can map the grid of thread blocks to the physical domain so that one thread is responsible for calculating one grid point.

```
int x = blockDim.x * blockIdx.x + threadIdx.x;
int y = blockDim.y * blockIdx.y + threadIdx.y;

// ...
```

The thread hierarchy

Identification

- By using the thread IDs and the calculation on the previous slide we can map the grid of thread blocks to the physical domain so that one thread is responsible for calculating one grid point.
- Remember to add guards!

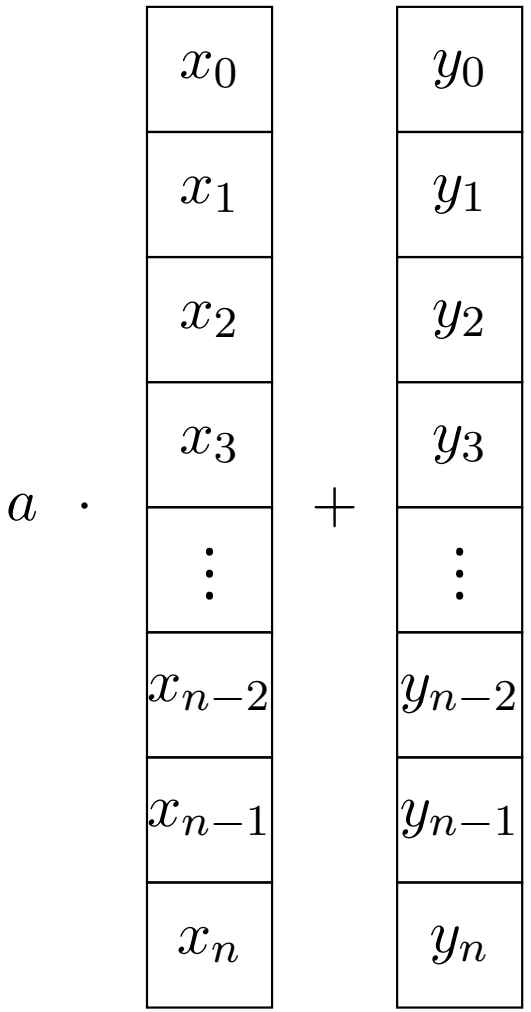
```
int x = blockDim.x * blockIdx.x + threadIdx.x;
int y = blockDim.y * blockIdx.y + threadIdx.y;

if ( /* GUARD */ ) {
    // ...
}
```

Example: SAXPY

Single-precision AX Plus Y

- A vector x scaled by a added with another vector y.
- We want to distribute one component per thread.



Example: SAXPY

Single-precision AX Plus Y

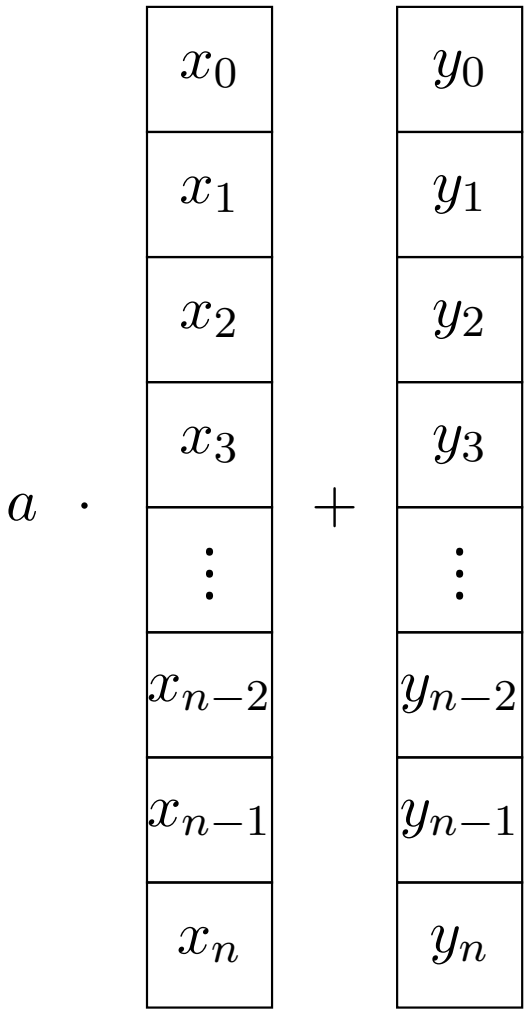
- A vector x scaled by a added with another vector y.
- We want to distribute one component per thread.
- $y_i = a \cdot x_i + y_i$
- How many threads per block?

$$a \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} + \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-2} \\ y_{n-1} \\ y_n \end{bmatrix}$$

Example: SAXPY

Single-precision AX Plus Y

- A vector x scaled by a added with another vector y.
- We want to distribute one component per thread.
- $y_i = a \cdot x_i + y_i$
- How many threads per block?
- How are they organized within each block?



Example: SAXPY

Single-precision AX Plus Y

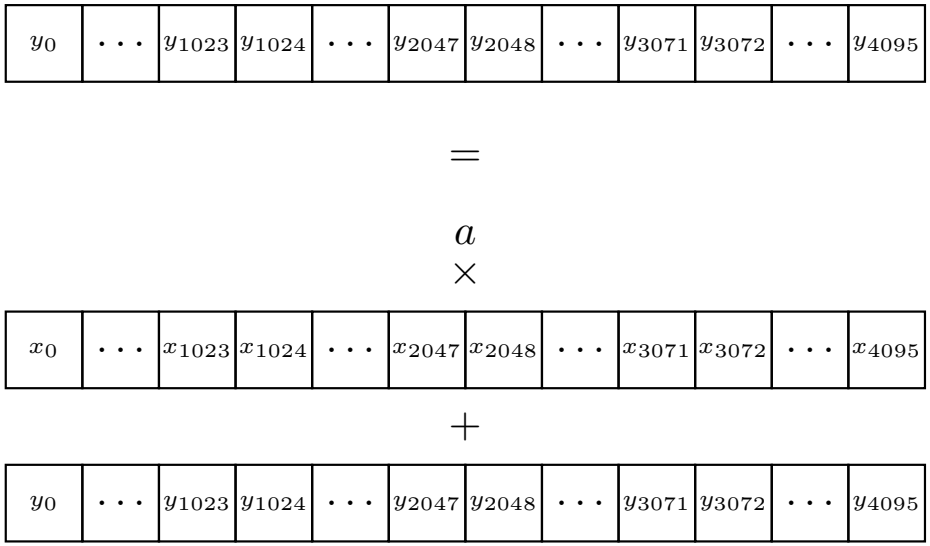
- A vector x scaled by a added with another vector y.
- We want to distribute one component per thread.
- $y_i = a \cdot x_i + y_i$
- How many threads per block?
- How are they organized within each block?
- **Proposition**: Organize as many threads as possible within one-dimensional blocks since the vectors are inherently one-dimensional.

$$a \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} + \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-2} \\ y_{n-1} \\ y_n \end{bmatrix}$$

Example: SAXPY

Building the grid (n = 4096)

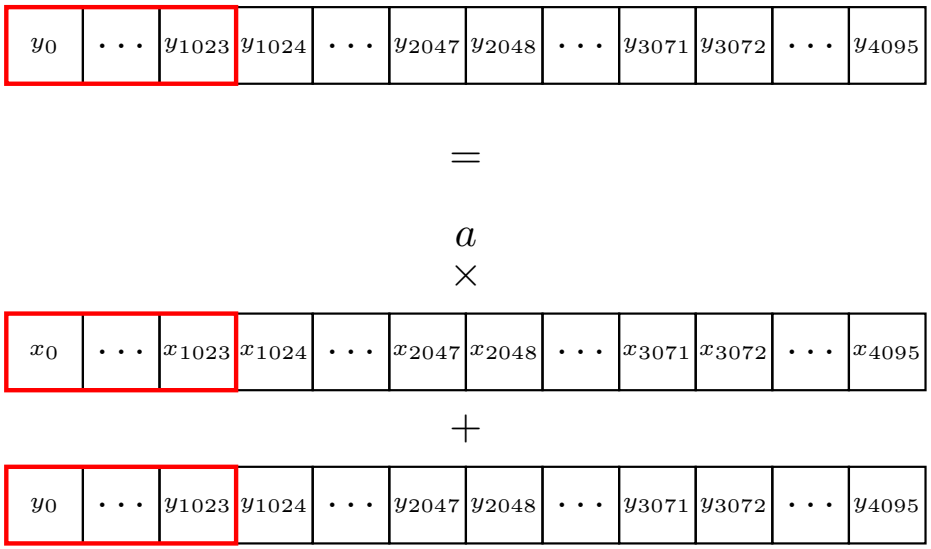
- Elements per vector: $n = 4096$
- Max threads per block: 1024



Example: SAXPY

Building the grid (n = 4096)

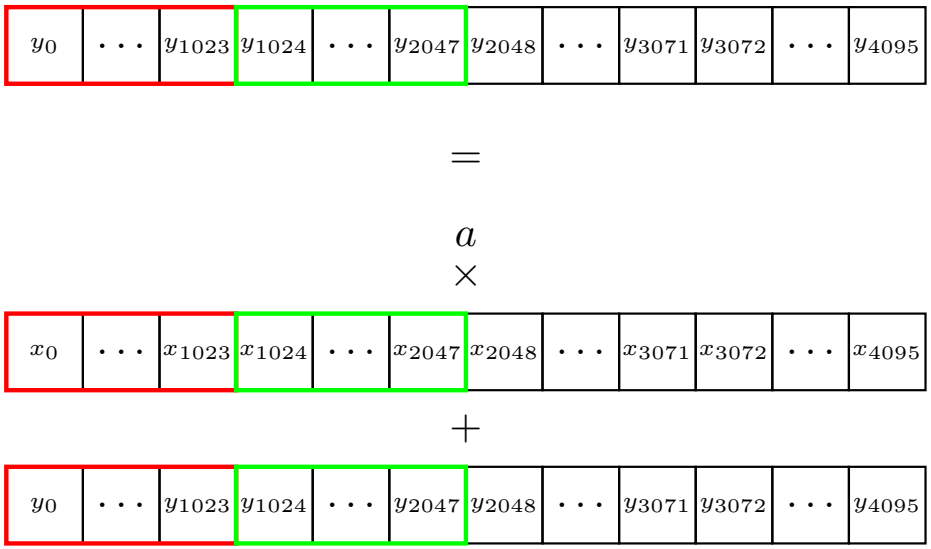
- Elements per vector: $n = 4096$
- Max threads per block: 1024
- Threads per block: $N_t = min(n, 1024)$



Example: SAXPY

Building the grid (n = 4096)

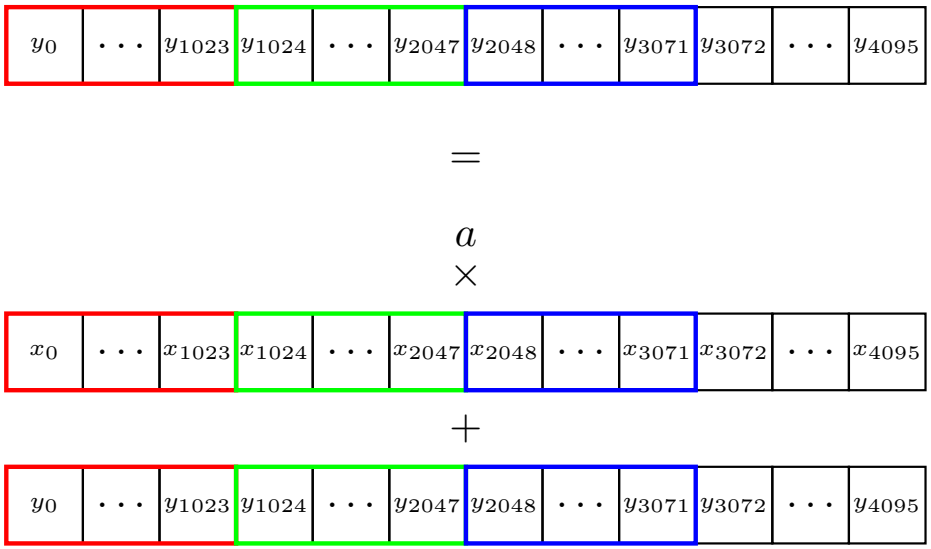
- Elements per vector: $n = 4096$
- Max threads per block: 1024
- Threads per block: $N_t = min(n, 1024)$



Example: SAXPY

Building the grid (n = 4096)

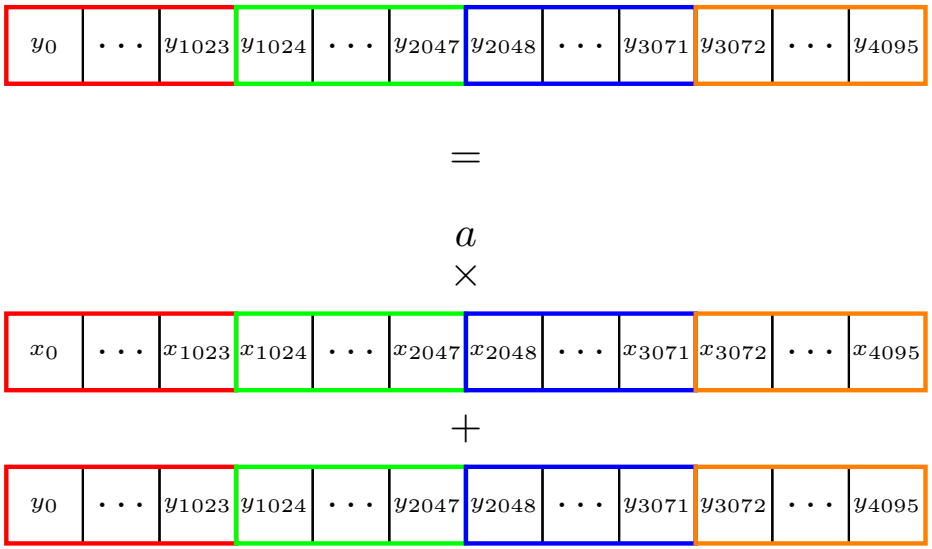
- Elements per vector: $n = 4096$
- Max threads per block: 1024
- Threads per block: $N_t = min(n, 1024)$



Example: SAXPY

Building the grid (n = 4096)

- Elements per vector: $n = 4096$
- Max threads per block: 1024
- Threads per block: $N_t = min(n, 1024)$
- Thread blocks: $N_b = \lceil \frac{n}{N_t} \rceil = 4$



Example: SAXPY

Implementation

```
int n = 4096;
float a, *h_x, *h_y, *d_x, *d_y;

cudaMalloc ( &d_x, sizeof(float) * n );
cudaMalloc ( &d_y, sizeof(float) * n );
cudaMemcpy ( d_x, h_x, sizeof(float) * n,
             cudaMemcpyHostToDevice );
cudaMemcpy (d_y, h_y, sizeof(float) * n,
             cudaMemcpyHostToDevice );

dim3 threadBlockDims = {1024, 1, 1};
dim3 gridDims = {ceil(n/1024), 1, 1};

saxpy <<<gridDims, threadBlockDims>>> ( n, a, d_x, d_y );

cudaMemcpy ( h_y, d_y, sizeof(float) * n,
             cudaMemcpyDeviceToHost );
cudaFree ( d_x );
cudaFree ( d_y );
```

Example: SAXPY

Implementation

```
__global__ saxpy ( const int n, const float a,
                   const float *x, const float *y )
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    if (idx >= n) return;
    y[idx] = a*x[idx] + y[idx];
}
```

Pro-tips

Variable names

```
real_t *h_mass; // Host variable
real_t *d_mass; // Device variable
```

Error handling

```
cudaError_t status;
status = cudaMalloc(&devPtr, /* size */);

if ( status != cudaSuccess ) {
    fprintf(stderr,
        "GPU Error: %s %s %d\n",
        cudaGetErrorString(code), __FILE__, __LINE__);
    // If you want to abort at this point, add an abort
    exit(EXIT_FAILURE);
}
```

Your tasks

1. Setup
   - Allocate memory on the host (CPU) and the device (GPU).
   - Transfer data from the host to the device.
   - Specify grid and block layout.

Your tasks

1. Setup
   - Allocate memory on the host (CPU) and the device (GPU).
   - Transfer data from the host to the device.
   - Specify grid and block layout.
2. Write kernels and device functions to parallelise the `time_step` and `boundary_condition` functions.

Your tasks

1. Setup
   - Allocate memory on the host (CPU) and the device (GPU).
   - Transfer data from the host to the device.
   - Specify grid and block layout.
2. Write kernels and device functions to parallelise the `time_step` and `boundary_condition` functions.
3. Handle results and teardown
   - Transfer data from the device to the host.
   - Free previously allocated memory.

Where should you run the code?

- You might have an NVIDIA GPU in your **personal computer**.

  You can check with the command

  ```
  lspci | grep -i nvidia
  ```

  Use this guide to install CUDA.

- **Oppdal and Selbu** have NVIDIA T4 GPUs.

  The document under *Sources and Syllabus* in Blackboard explains how to connect to and use the Snotra cluster.

  Note that to compile the code on Oppdal or Selbu you need to update the Makefile so that the correct compiler is used:

  ```
  #PARALLEL_CC:=nvcc
  PARALLEL_CC:=/usr/local/cuda-12.2/bin/nvcc
  ```

Extra resources

CUDA C++ Programming Guide

CUDA Runtime API Reference

Debugging with CUDA-GDB