Programming in CUDA

Part III. CUDA Kernels

Frédéric S. Masset

Instituto de Ciencias Físicas, UNAM

Lecture on CUDA

Outline

Outline

Reminder : what is kernel ?

A kernel, in CUDA's jargon, is a function that runs on the device.

It is a *lightweight* piece of program : little memory space for executable and registers in the multiprocessors.

Outline

Specifying that a function runs on the device

In the source code, a kernel looks pretty much like a normal C function, except that it has the qualifier `__global__` or `__device__` before it:

```
float myfunc (float foo) {
.... // A normal host function
}

__global__ void mykernel (float foo) {
... // My first kernel !
}
```

kernel qualifiers

- `__global__` means that the kernel can be called either from the host or the device. This is by far the most common case.
- `__device__` means that the kernel can only be called from the device (a kind of *subfunction* of another kernel).

Restrictions of kernels

- Contrary to normal C functions, kernels must return `void`
- Contrary to normal C functions, kernels are not recursive : a kernel cannot call itself (not too much of a restriction for standard HPC).
- The number of arguments is fixed (unlike some C functions like `printf ()`).

Kernel arguments

The arguments of a kernel may comprise scalar variables like `int`s or `float`s, but also pointers.

- The memory referred to by the pointers must be on the device, obviously. **How could the kernel access those values, otherwise ?**
- The scalar variables are host variables.

Kernel arguments

The arguments of a kernel may comprise scalar variables like `int`s or `float`s, but also pointers.

- The memory referred to by the pointers must be on the device, obviously.
- The scalar variables are host variables.

Kernel arguments

The arguments of a kernel may comprise scalar variables like `int`s or `float`s, but also pointers.

- The memory referred to by the pointers must be on the device, obviously.
- The scalar variables are host variables. **It cannot be otherwise, anyhow : there is no such thing as device variables.**

Kernel arguments

The arguments of a kernel may comprise scalar variables like `int`s or `float`s, but also pointers.

- The memory referred to by the pointers must be on the device, obviously.
- The scalar variables are host variables.

The value of these variables is sent to the device upon invocation of the kernel (when we “call” it). This is done automatically : no need to worry for host $\Longleftrightarrow$ device communications in this case.

Kernel and threads

When a kernel is launched, it runs in many different copies (*threads*) over the many cores of the GPU. There are many more threads than cores available, usually.

The “topology” of the threads obeys the division of a whole grid in blocks, and of blocks into threads, as seen in lecture 1.

Kernel and threads

When a kernel is launched, it runs in many different copies (*threads*) over the many cores of the GPU. There are many more threads than cores available, usually.

The “topology” of the threads obeys the division of a whole grid in blocks, and of blocks into threads, as seen in lecture 1.

How does a thread *know* which thread it is, during execution ?

Outline

Built-in variables

Inside a kernel, there are a few built-in variables, which are always available and that do not need to be declared. This variables are:

- `blockDim` : gives the size of a block, in threads.
- `blockIdx` : gives the location of the current block in the grid of blocks.
- `threadIdx` : gives the location of the current thread, within its parent block.
- `gridDim` : gives the total size of the grid, in blocks. Hardly used.

Built-in variables

Inside a kernel, there are a few built-in variables, which are always available and that do not need to be declared. This variables are:

- `blockDim` : gives the size of a block, in threads.
- `blockIdx` : gives the location of the current block in the grid of blocks.
- `threadIdx` : gives the location of the current thread, within its parent block.
- `gridDim` : gives the total size of the grid, in blocks. Hardly used.

Built-in variables

Inside a kernel, there are a few built-in variables, which are always available and that do not need to be declared. This variables are:

- `blockDim` : gives the size of a block, in threads.
- `blockIdx` : gives the location of the current block in the grid of blocks.
- `threadIdx` : gives the location of the current thread, within its parent block.
- `gridDim` : gives the total size of the grid, in blocks. Hardly used.

Built-in variables

Inside a kernel, there are a few built-in variables, which are always available and that do not need to be declared. This variables are:

- `blockDim` : gives the size of a block, in threads.
- `blockIdx` : gives the location of the current block in the grid of blocks.
- `threadIdx` : gives the location of the current thread, within its parent block.
- `gridDim` : gives the total size of the grid, in blocks. Hardly used.

Built-in variables

Inside a kernel, there are a few built-in variables, which are always available and that do not need to be declared. This variables are:

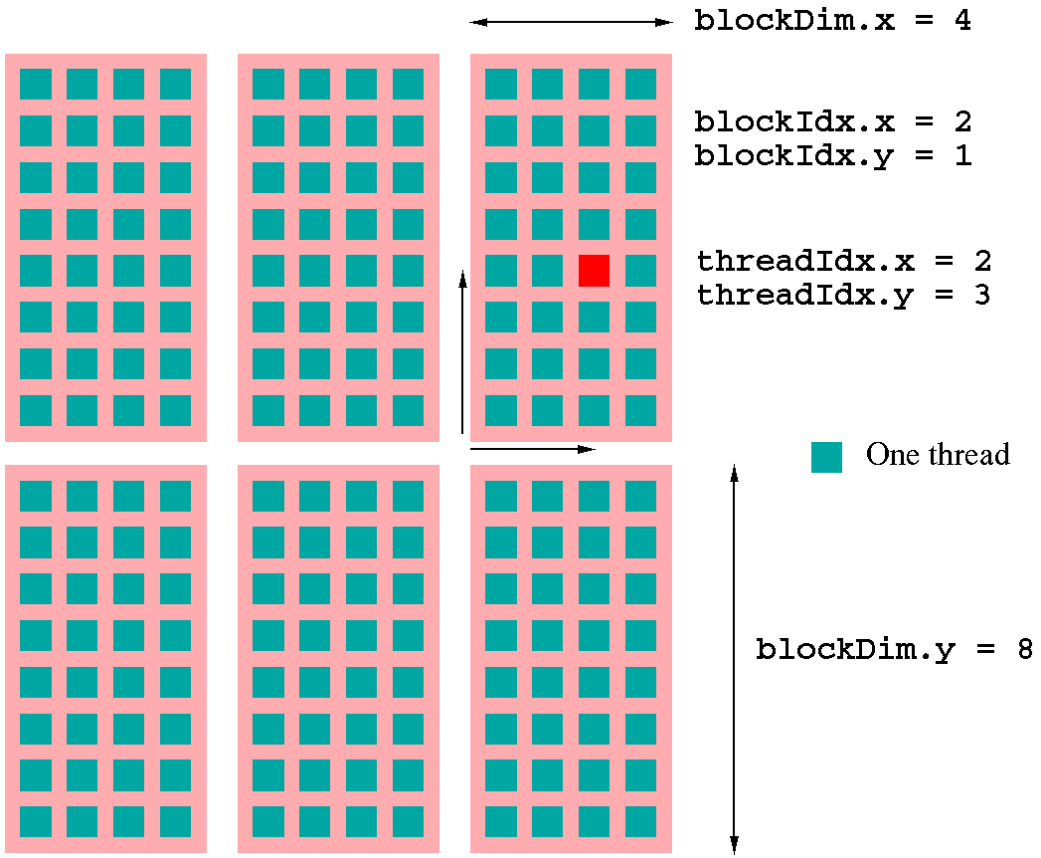
- `blockDim` : gives the size of a block, in threads.
- `blockIdx` : gives the location of the current block in the grid of blocks.
- `threadIdx` : gives the location of the current thread, within its parent block.
- `gridDim` : gives the total size of the grid, in blocks. Hardly used.

Each of these variables is a structure, with three fields : `x`, `y` and `z`. For instance, the *x* position of the thread in its block is `threadIdx.x`.

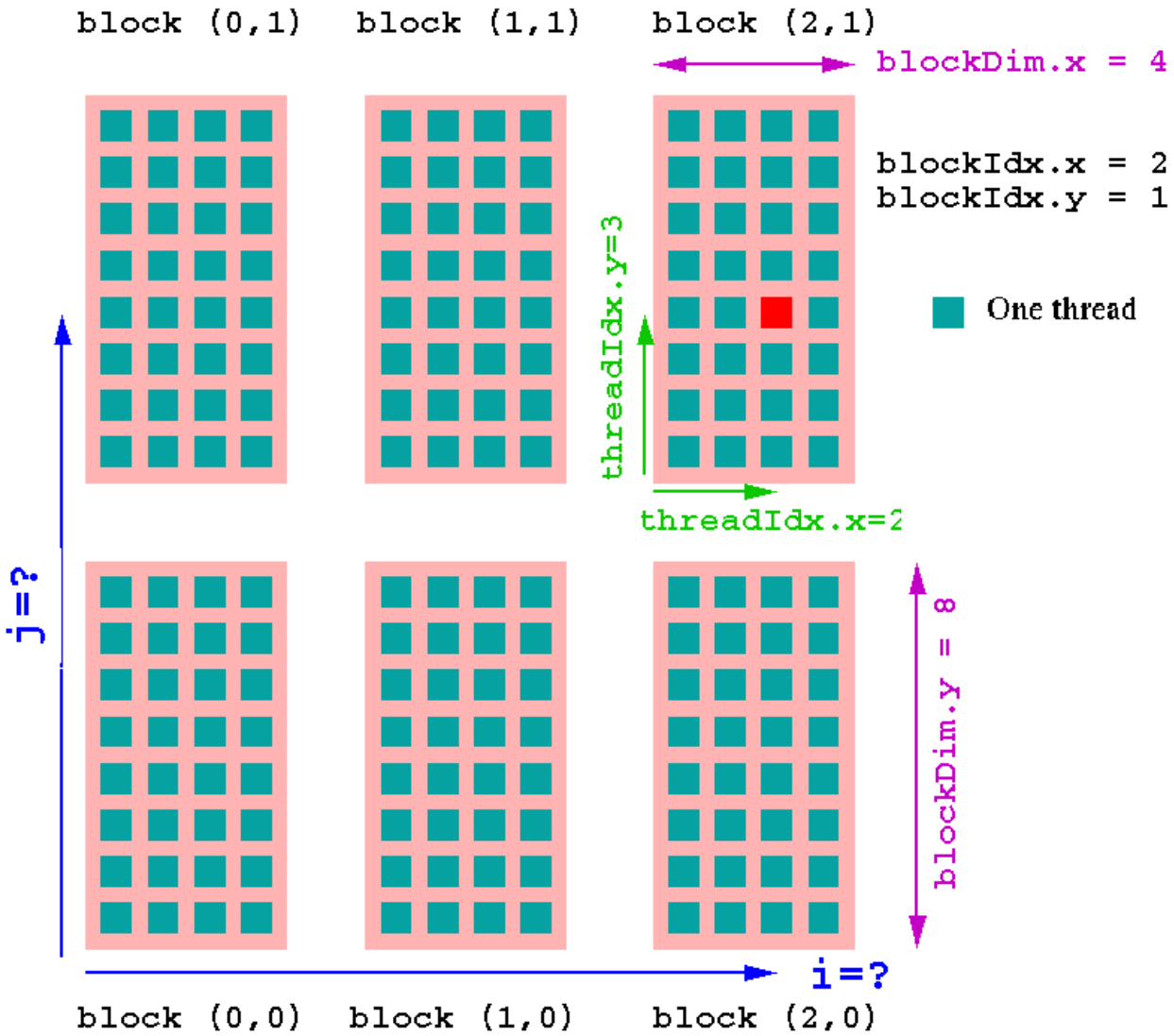
Outline

A typical use of a kernel

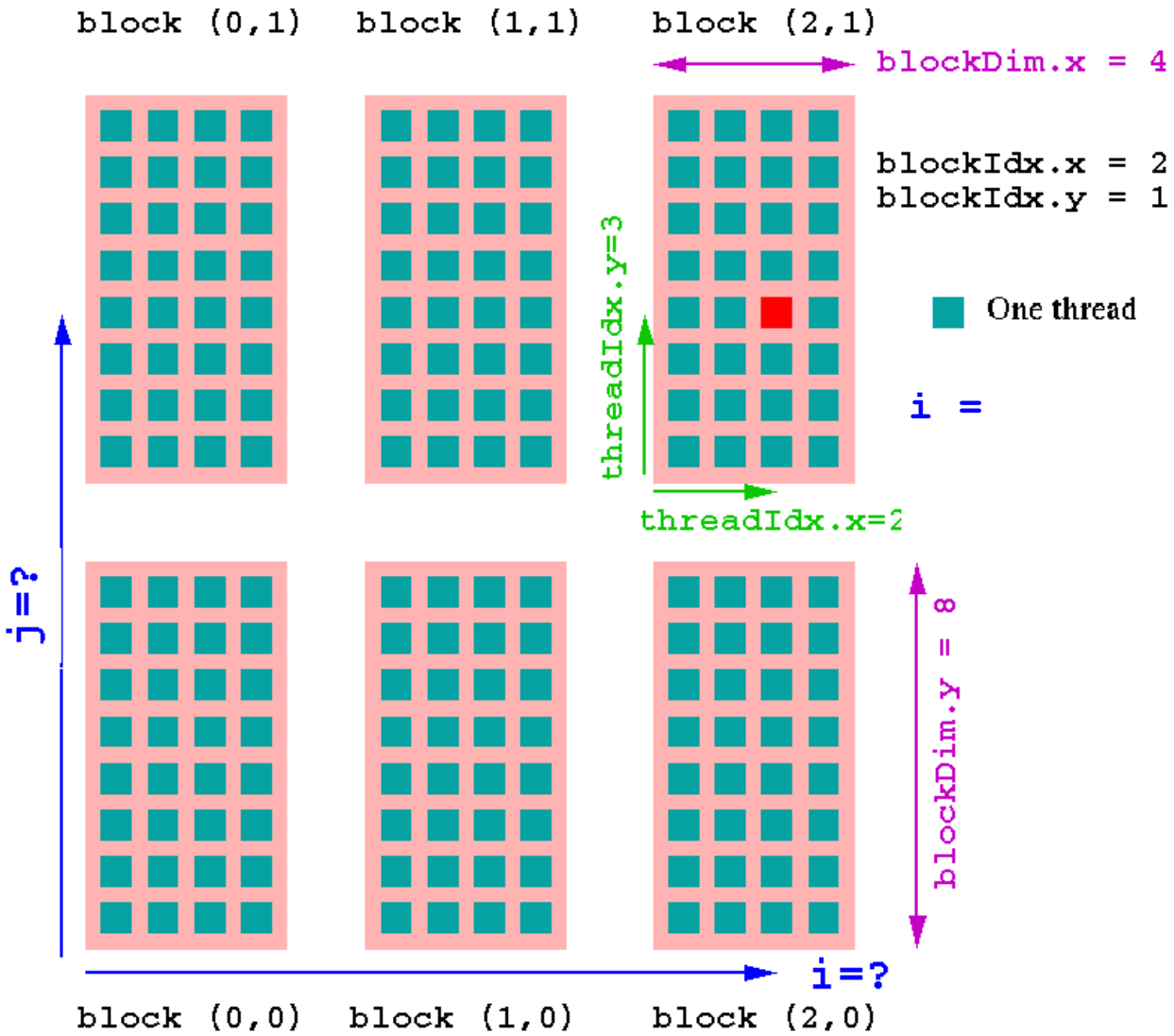
A kernel is typically used to perform operations on an array. Each thread treats one cell of the array and returns. All threads must therefore know where is the cell that they must deal with. The built-in variables allow to get the “absolute” position of the thread.



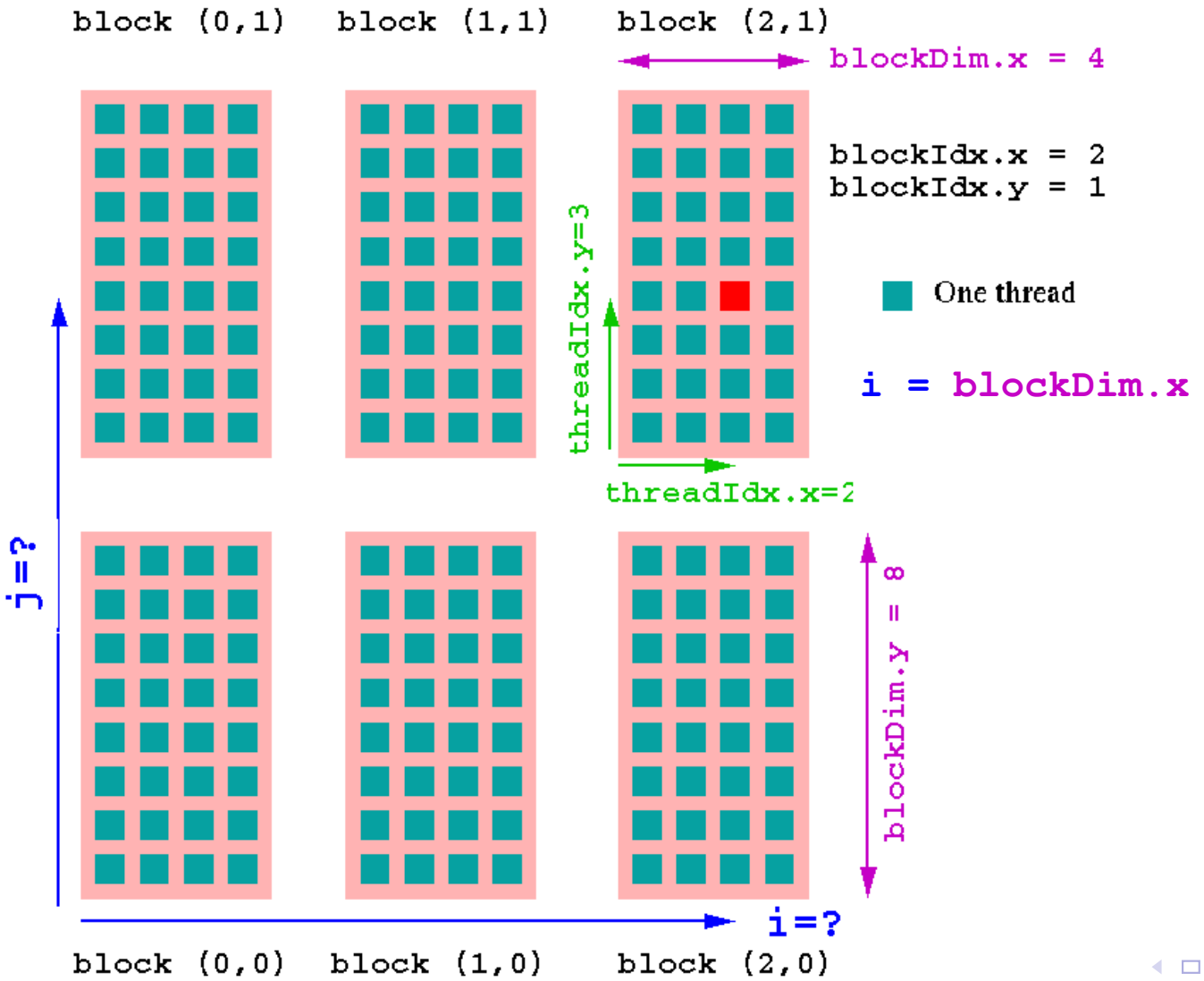
Absolute position of a thread



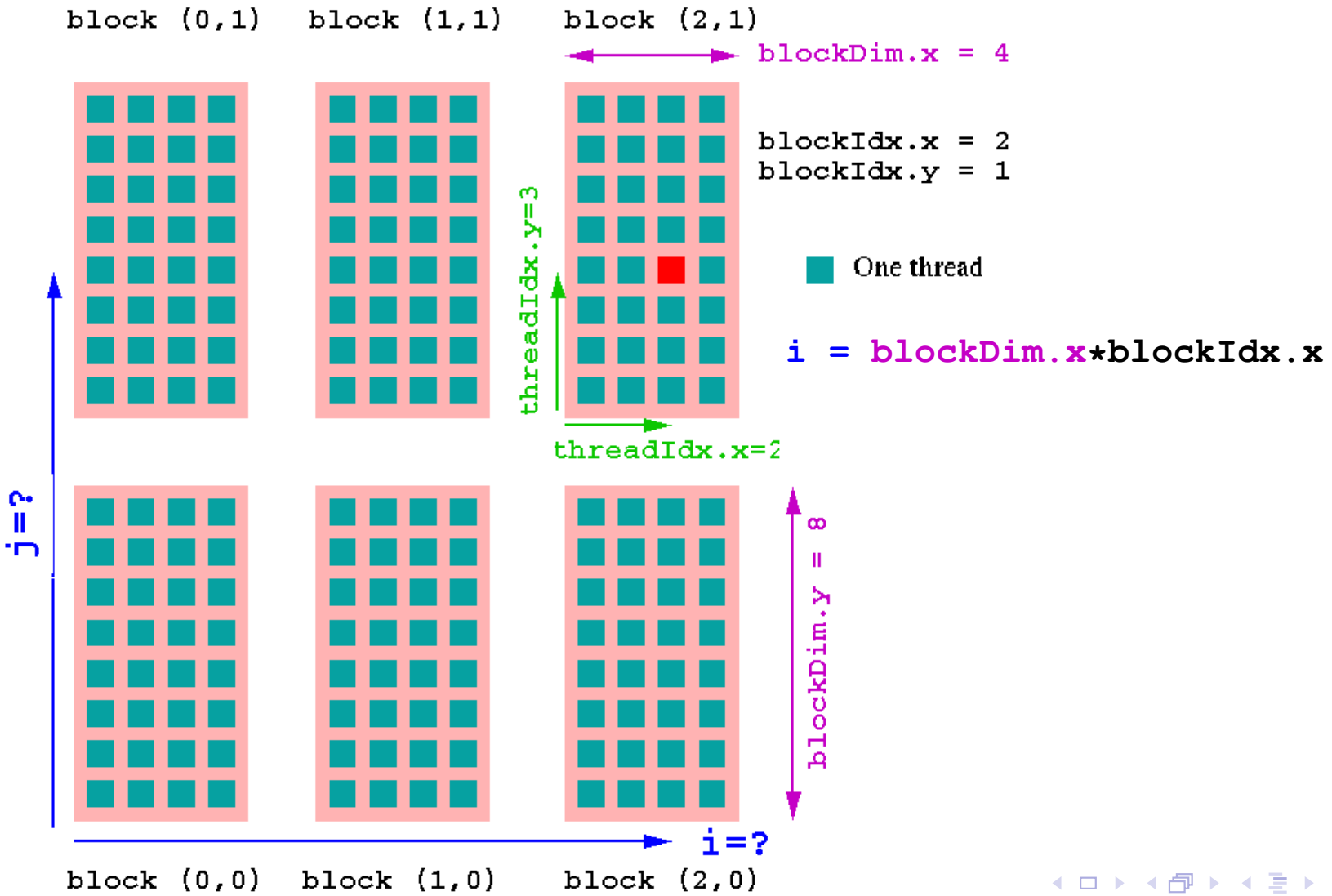
Absolute position of a thread



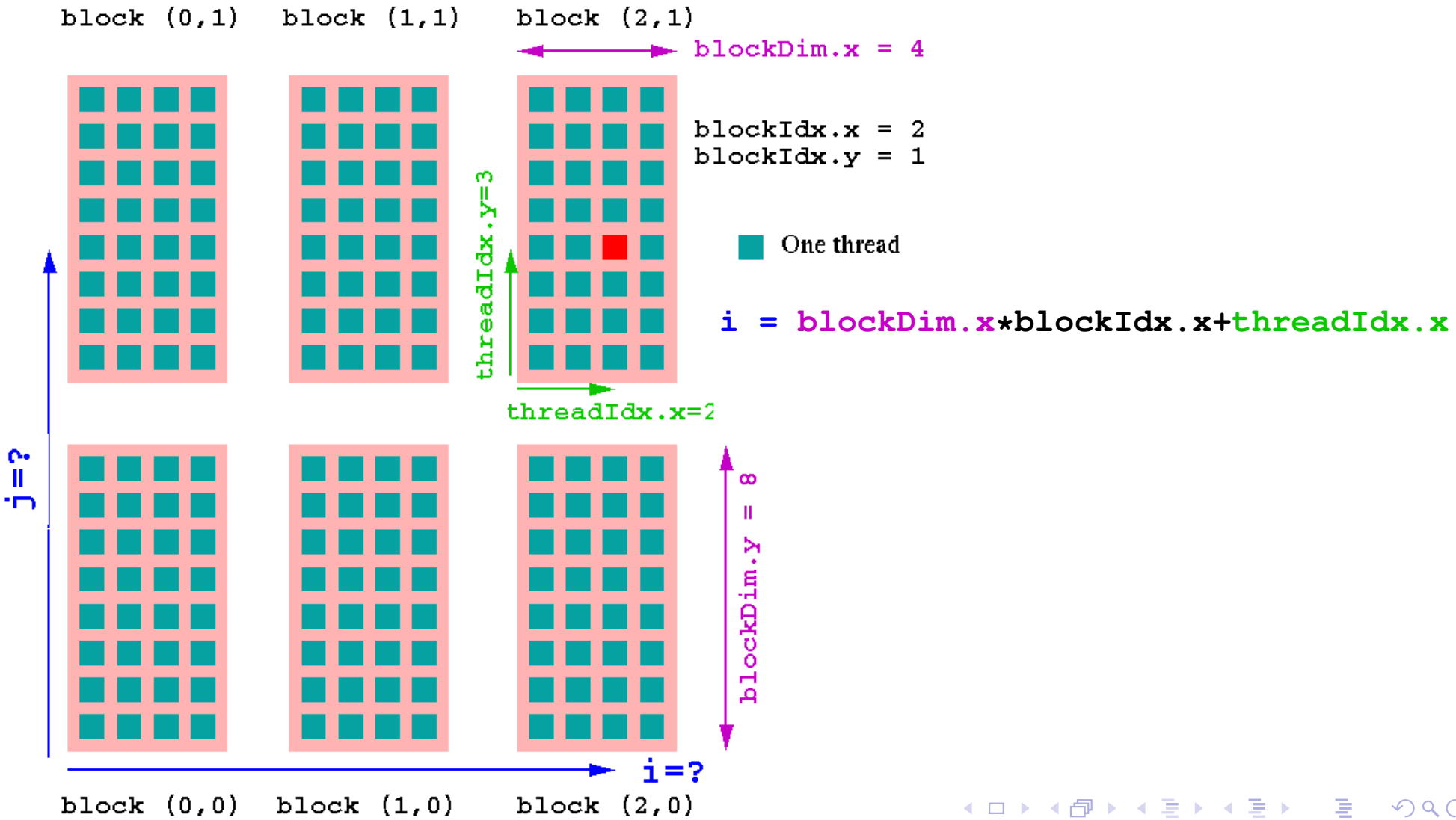
Absolute position of a thread



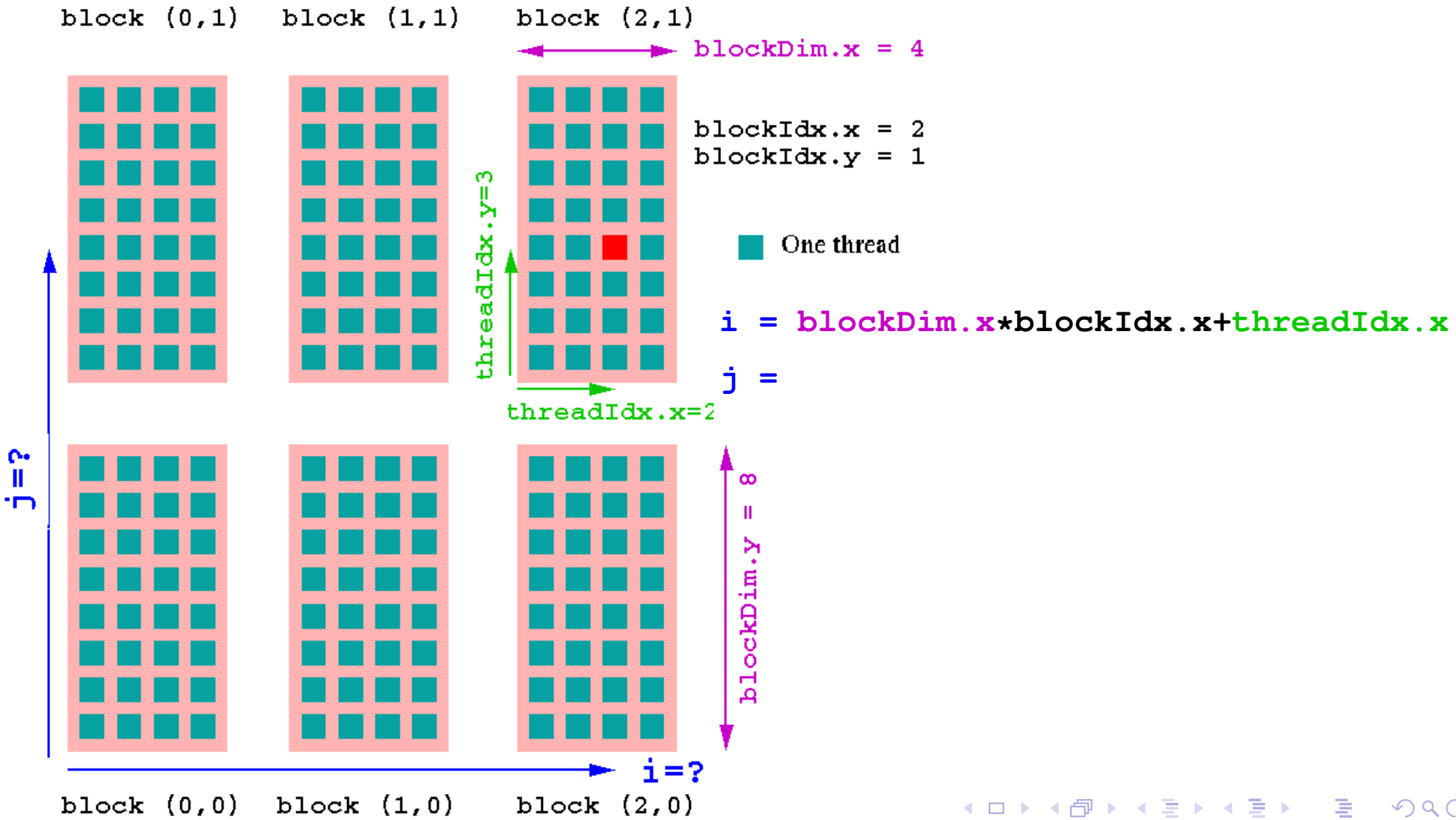
Absolute position of a thread



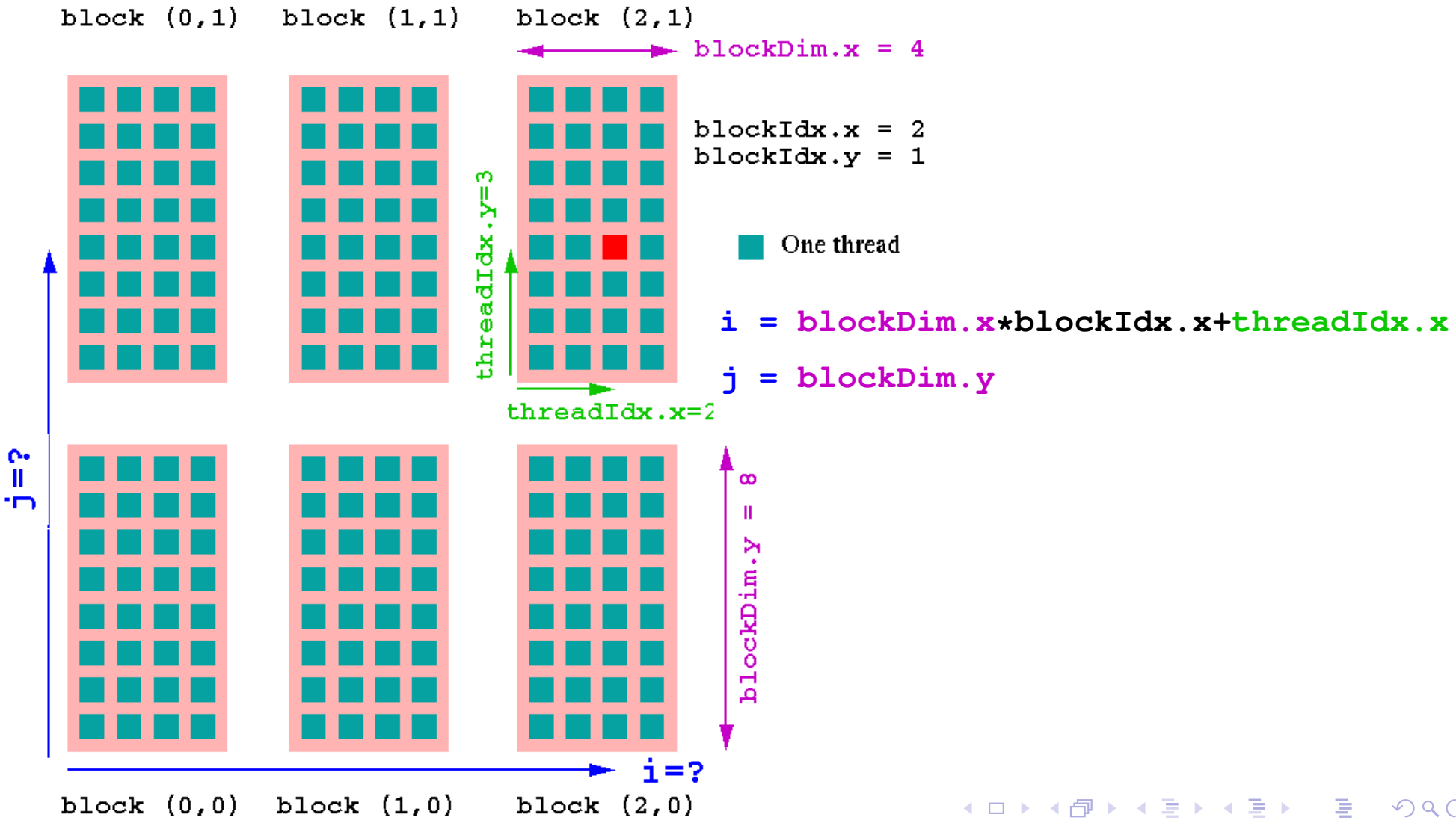
Absolute position of a thread



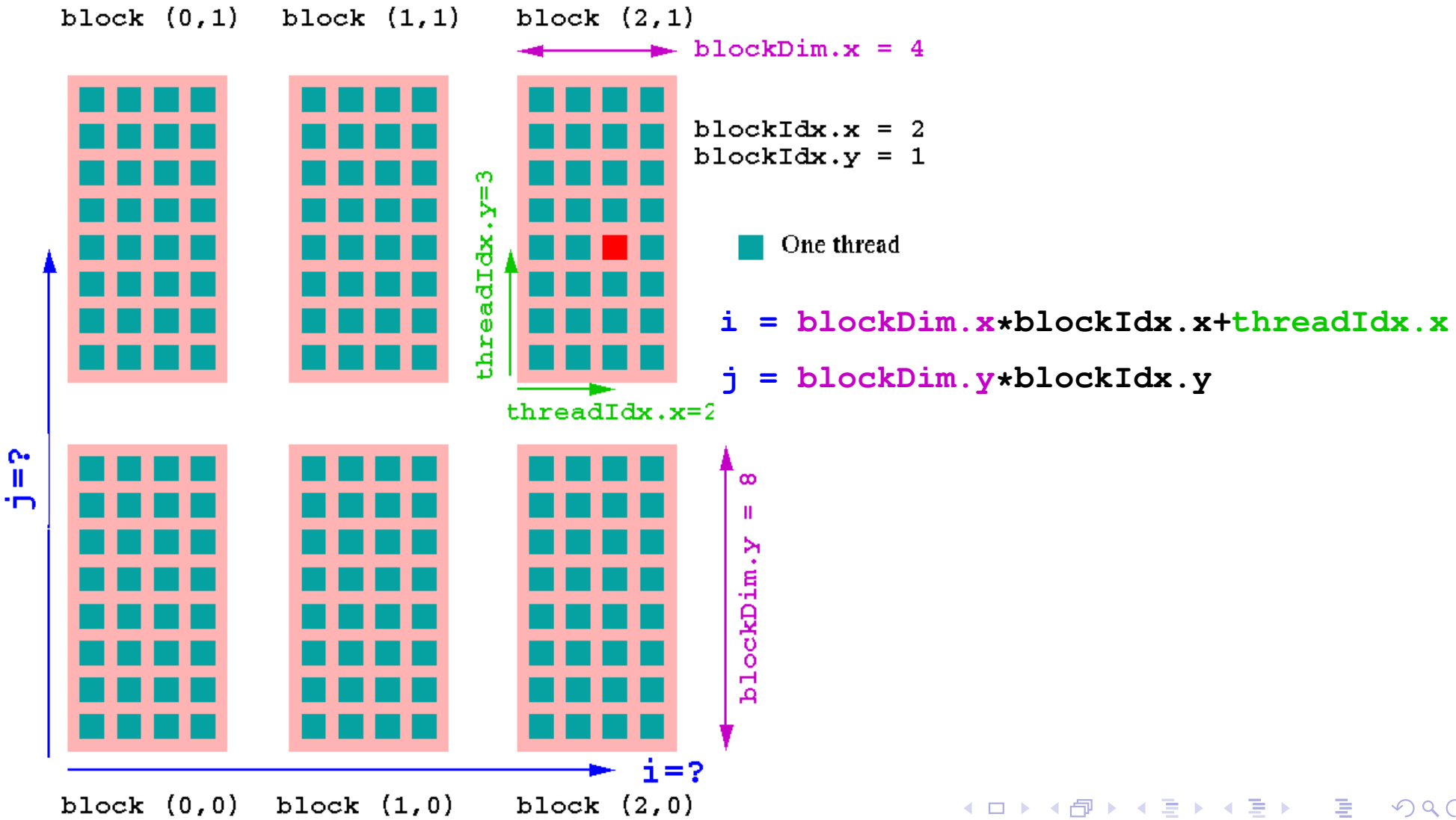
Absolute position of a thread



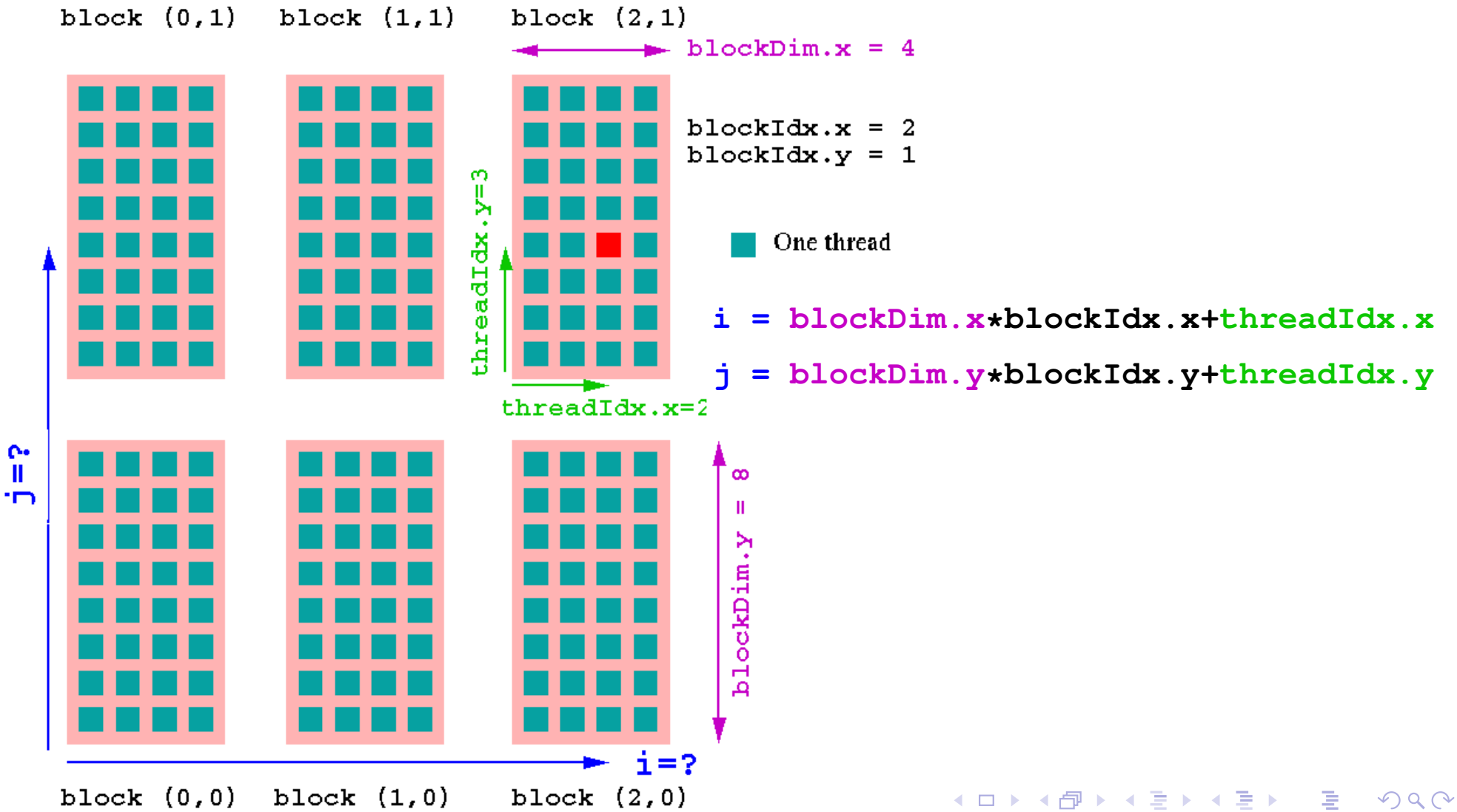
Absolute position of a thread



Absolute position of a thread



Absolute position of a thread



A simpler example: 1D array

The simplest case corresponds naturally to a 1D grid of blocks, each block being a 1D row of threads. In that case, we have:

```
blockDim.y = 1
blockDim.z = 1
blockIdx.y = 0
blockIdx.z = 0
threadIdx.y = 0
threadIdx.z = 0
```

A simpler example: 1D array

The simplest case corresponds naturally to a 1D grid of blocks, each block being a 1D row of threads. In that case, we have:

```
blockDim.y = 1
blockDim.z = 1   always true
blockIdx.y = 0
blockIdx.z = 0   always true
threadIdx.y = 0
threadIdx.z = 0
```

Remember that the grid of blocks is at most 2D

A simpler example: 1D array

The simplest case corresponds naturally to a 1D grid of blocks, each block being a 1D row of threads. In that case, we have:

```
blockDim.y = 1
blockDim.z = 1
blockIdx.y = 0
blockIdx.z = 0
threadIdx.y = 0
threadIdx.z = 0
```

In that case the "absolute" index of the thread is just:

```
i = blockDim.x*blockIdx.x + threadIdx.x
```

Outline

Determining the blocks / threads topology

- When do we determine the size of the grid of blocks ?
- When do we determine the size of the blocks of threads ?
- Answer : at runtime, when we invoke the kernel (when we *call* it).

Determining the blocks / threads topology

- When do we determine the size of the grid of blocks ?
- When do we determine the size of the blocks of threads ?
- Answer : at runtime, when we invoke the kernel (when we *call* it).

Determining the blocks / threads topology

- When do we determine the size of the grid of blocks ?
- When do we determine the size of the blocks of threads ?
- Answer : at runtime, when we invoke the kernel (when we *call* it).

Determining the blocks / threads topology

- When do we determine the size of the grid of blocks ?
- When do we determine the size of the blocks of threads ?
- Answer : at runtime, when we invoke the kernel (when we *call* it).

Therefore, there is no indication of a given, fixed block / thread topology within the kernel source.

A given kernel can be called at different places in a program, each time with a different block / thread topology.

Syntax of invocation

Assume I want to call the kernel `mykern` with a grid of 32×16 blocks, each block being a 3D grid of $8 \times 4 \times 8$ threads.

```
dim3 grid (32,16);
dim3 block (8,4,8);
mykern <<< grid, block >>> (...arguments...);
```

Syntax of invocation

Assume I want to call the kernel `mykern` with a grid of 32×16 blocks, each block being a 3D grid of $8 \times 4 \times 8$ threads.

```
dim3 grid (32,16);
dim3 block (8,4,8);
mykern <<< grid, block >>> (...arguments...);
```

Syntax of invocation

Assume I want to call the kernel `mykern` with a grid of 32×16 blocks, each block being a 3D grid of $8 \times 4 \times 8$ threads.

```
dim3 grid (32,16); dim3 is a type consisting of 3 integers
dim3 block (8,4,8);
mykern <<< grid, block >>> (...arguments...);
```

Syntax of invocation

Assume I want to call the kernel `mykern` with a grid of 32×16 blocks, each block being a 3D grid of $8 \times 4 \times 8$ threads.

```
dim3 grid (32,16); No need to write trailing 1's
dim3 block (8,4,8);
mykern <<< grid, block >>> (...arguments...);
```

Syntax of invocation

Assume I want to call the kernel `mykern` with a grid of 32×16 blocks, each block being a 3D grid of $8 \times 4 \times 8$ threads.

```
dim3 grid (32,16);
dim3 block (8,4,8);
mykern <<< grid, block >>> (...arguments...);
```

`<<<...>>>` is the standard syntax for kernel invocations. You must use it to specify the block/threads topology.

Special case: 1D topology

In the particular case of a 1D topology, there is no need to use `dim3` variables. You can give directly the figures specifying the number of blocks, and the number of threads per block:

```
int a=256
mykern <<< 32, a >>> (...arguments...)
```

Kernel calls are asynchronous

Kernel calls are asynchronous : this means that straight away after invocation, the control is given back to the CPU that can continue its execution flow, even though the kernel is still running on the GPU.

Under most circumstances, you may be interested in waiting for the kernel to complete its task. This is done by inserting:

```
cudaThreadSynchronize ();
```

right after the kernel invocation. The CPU execution flow will be blocked until the kernel “returns”.

Outline

Our first kernel

Usual practice requires a first “hello world !” sand box program when learning a new language. As emphasized previously, however, GPUs do not have input/output directives which would allow to print a string to a terminal.

Here our sand box program will consist of a kernel which sets all elements of a vector to a given value.

Practice `sandbox1`

Our program consists of two source files : `main.c` and `setvec.cu`.
Please read them carefully (begin with `main.c`). Note that `main.c` is a standard C program without any mention of CUDA.
You'll need to edit both files to make it work.
In order to compile, issue (your mileage may vary):

```
cc -c main.c  produces object file main.o
nvcc -c setvec.cu produces object file setvec.o
nvcc main.o setvec.o Linking stage
```

An executable `a.out` is produced.

Practice `sandbox1`

The `main.c` contains a call to the CUDA file (not directly to the kernel !) Then it compares the result with the expectation (here all values should be 11.0), and issue either `PASSED` or `FAILED`.

Note that a large number of examples in the SDK work essentially the same way (`PASSED / FAILED`).
Your task consists in :

- Evaluate the absolute index of the thread in the kernel
- Write the kernel invocation in the C++ wrapper `set_value ()`
- Transfer the data from the GPU to the host, where it will be analyzed by `main ()`.

Once it works...

- Change the length of the vector from 1024 to 1100
- Does it still work ? Why ? Find a workaround.
- When it works again, edit back your source file to make the same mistake as initially... Does it work or fail ? Why ?? Suggest a workaround.

Variations on `sandbox1` — 1

What should be the program output if I change the line

```
gpu_ptr[i] = value;
```

with:

- `gpu_ptr[i] = blockIdx.x`
- `gpu_ptr[i] = blockDim.x`
- `gpu_ptr[i] = blockDim.z`
- `gpu_ptr[i] = threadIdx.x`

Check your expectations by editing the source and running the program for each case.

Variations on `sandbox1` — 2

The choice that each thread treats one cell is completely arbitrary. Modify the program so that we have less threads, but each thread treats two subsequent cells.

For instance, if the vector length is 1024, we want 512 threads. Thread 0 treats cells 0 and 1, thread 1 treats cell 2 and 3, and so on....

Keep in a separate directory these modified sources. We shall use them when dealing with the profiler.

Outline

Practice `sandbox2`

This program is composed of two source files : `addition.c` which contains the function `main ()`, and the CUDA file `addongpu.cu`.

Read them carefully.

The intended execution flow is as follows:

- allocate 3 matrices on CPU, initialize 2 of them with random numbers
- perform the addition of the two random matrices on GPU, then send result back to CPU.
- compare result with expectations $\Rightarrow$ `PASSED / FAILED`

You now have a `makefile`. No longer need to compile manually. Just issue `make` to build the executable.

Work requested in `sandbox2`

The file `addition.c` does not require any changes. You should edit the file `addongpu.cu`, at the locations where you find the comment `UP TO YOU`, in order to:

- properly evaluate the variable `index` in the kernel
- allocate the matrices on the GPU
- copy the matrices from CPU to GPU
- evaluate the correct number of blocks in x and y
- invoke the kernel
- copy the result from GPU to CPU

From loops to threads

Note how the GPU implementation differs from a CPU implementation.

- On the CPU, our program would consist of a loop (or two nested loops)
- On the GPU, there is no longer a loop. Rather, an index calculation that differs from thread to thread.
- The “heart” of the calculation remains the same (here a simple addition).

Once it works...

- Edit the kernel so that `j=0` $\Rightarrow$ only the leftmost column will be dealt with.
- Rebuild. Does it fail ? Why ?

Find a workaround and deduce a rule that you must follow when developping a CUDA program.

A bit of benchmarking — `sandbox2`

So far we have not worried about performance. We can start comparing the execution time for the matrix sum on GPU and CPU. For this purpose, we need to import the `chrono ()` function that we wrote in the `bandwidth` practice. You'll need to:

- take large matrices
- make a loop so that you invoke the kernel many times, in order to achieve a better accuracy
- transform the `main ()` function so that you perform a sum in the final loop, rather than a comparison. Surround this loop with invocations of the chronometer, in order to get the execution time on CPU.
- What performance gain do you obtain ?

A bit of benchmarking — `sandbox2`

Check how this performance gain changes when:

- you change the matrix size
- you change the thread block size
- you include the communications with the host in the time measurements

`sandbox2` — For the most advanced

Once you have instrumented the program for benchmarking, alter the kernel in the same way as we did in `sandbox1`:

- get half of the threads
- each thread treats two consecutive columns in the same row

How is the performance impacted ?