CINECA

Programming GPUs with CUDA: Introduction to CUDA

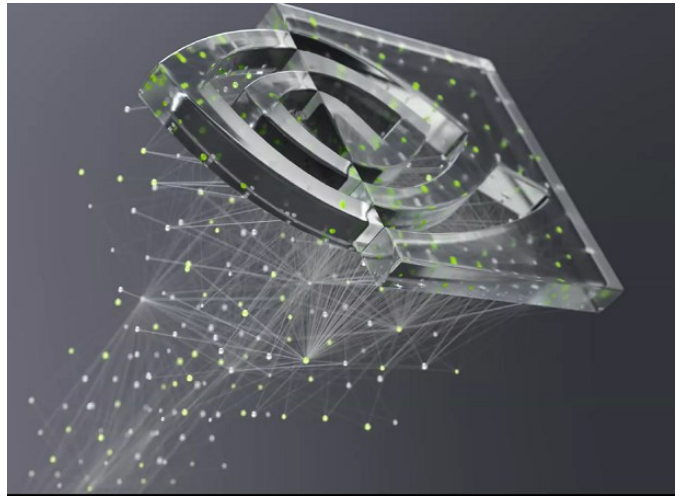
Lara Querciagrossa, Andrew Emerson, Nitin Shukla, Luca Ferraro, Sergio Orlandini

l.querciagrossa@cineca.it

July 12th, 2022

In this lecture…

- What is CUDA
- CUDA process flow
- CUDA kernel
- Compiling CUDA programs
- Execution model
- Kernel launch
- CUDA-provided thread hierarchy variables
- Global index



CUDA overview

Compute Unified Device Architecture

- CUDA is a **general purpose parallel computing platform and programming model** created by NVIDIA.
- It consists of:
  - a **hierarchical multi-threaded programming paradigm** that matches GPU hardware structure,
  - a set of extensions to **higher level programming languages** (C/C++ and Fortran) to use GPU as a coprocessor for heavy parallel task and to express thread parallelism within a familiar programming environment,
  - a **new architecture instruction set** called PTX (Parallel Thread eXecution) to match GPU typical hardware,
  - a **developer toolkit to compile**, debug, profile programs and run them easily in a heterogeneous systems,
  - a **set of GPU accelerated libraries** for common scientific algorithms.

Compute Unified Device Architecture

- **CUDA** C/C++ and FORTRAN **API** allow you to:
  - *control and query available* GPU devices,
  - *manage GPU memory* allocation and data transfers,
  - *manage* and control *independent work queues*,
  - define special language extension to *express thread parallelism* of selected functions (kernel) which will be executed on the GPU by thousands of threads.

- The ***SDK*** *includes*:
  - drivers, runtimes and API,
  - compiler wrappers for compiling cuda code (nvcc),
  - libraries (cuBLAS, cuFFT, cuSolver, etc),
  - debuggers (cuda-gdb, cuda-memcheck), profilers (nvprof, nView), etc.

CUDA process flow

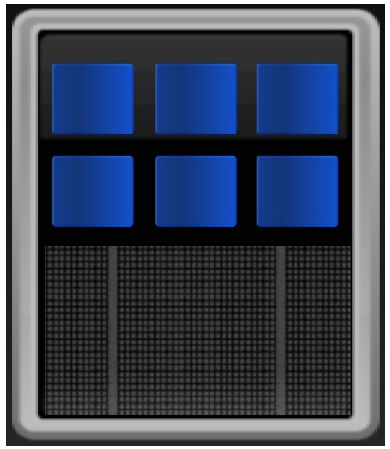
Basic 3-step process

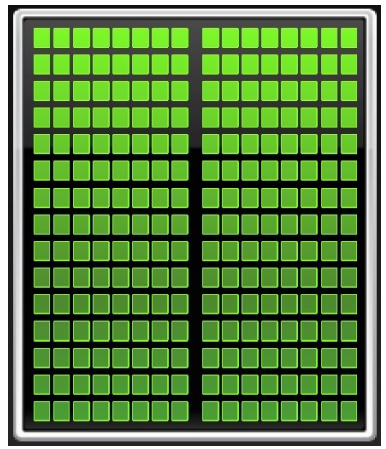
Heterogeneous computing

CPU

GPU

HOST

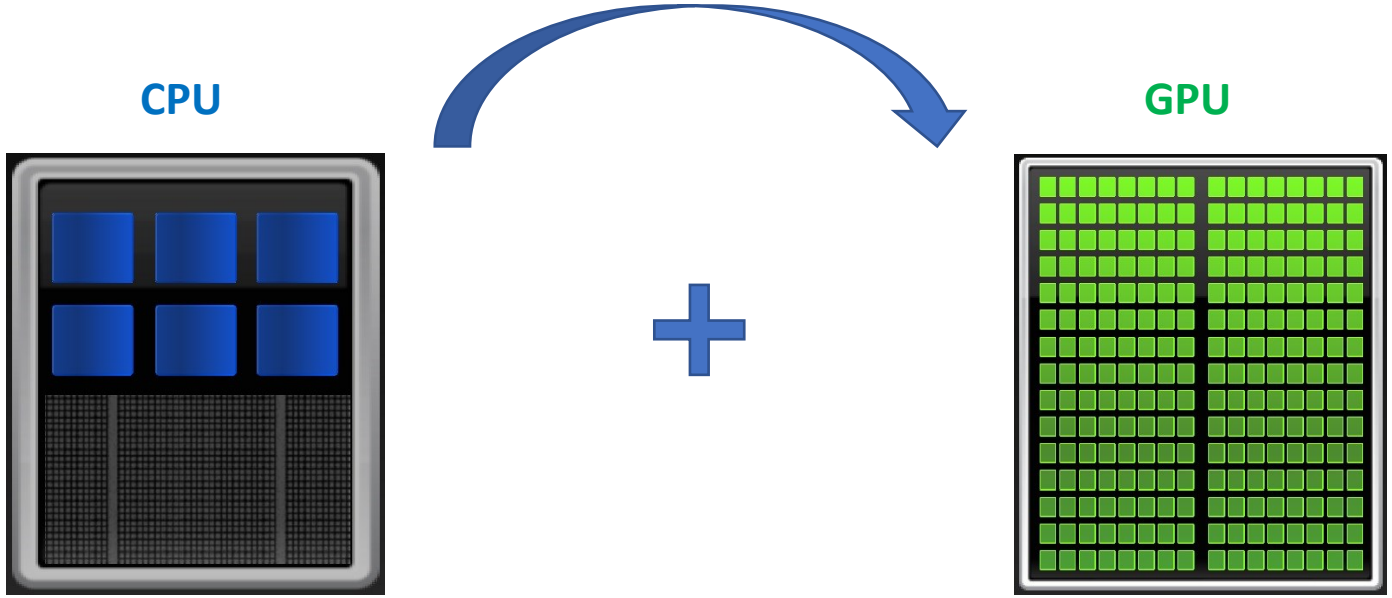




DEVICE

Basic 3-step process

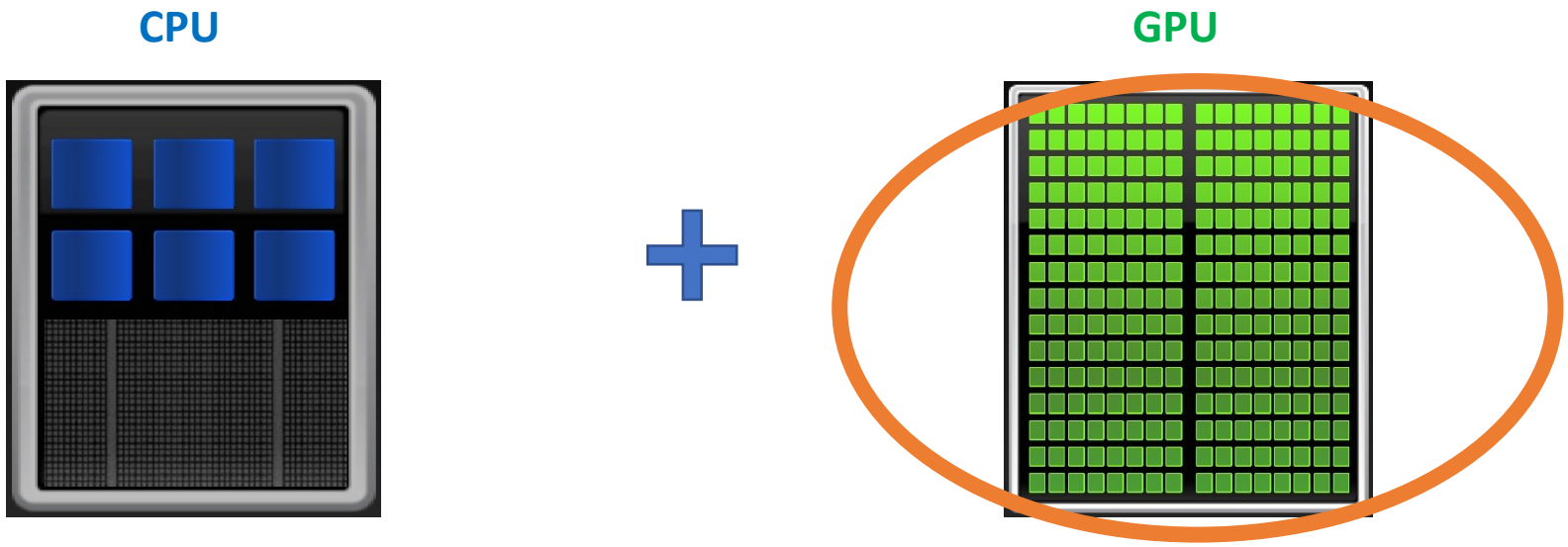
1. **Copy** input data from CPU memory **to GPU memory**



Basic 3-step process

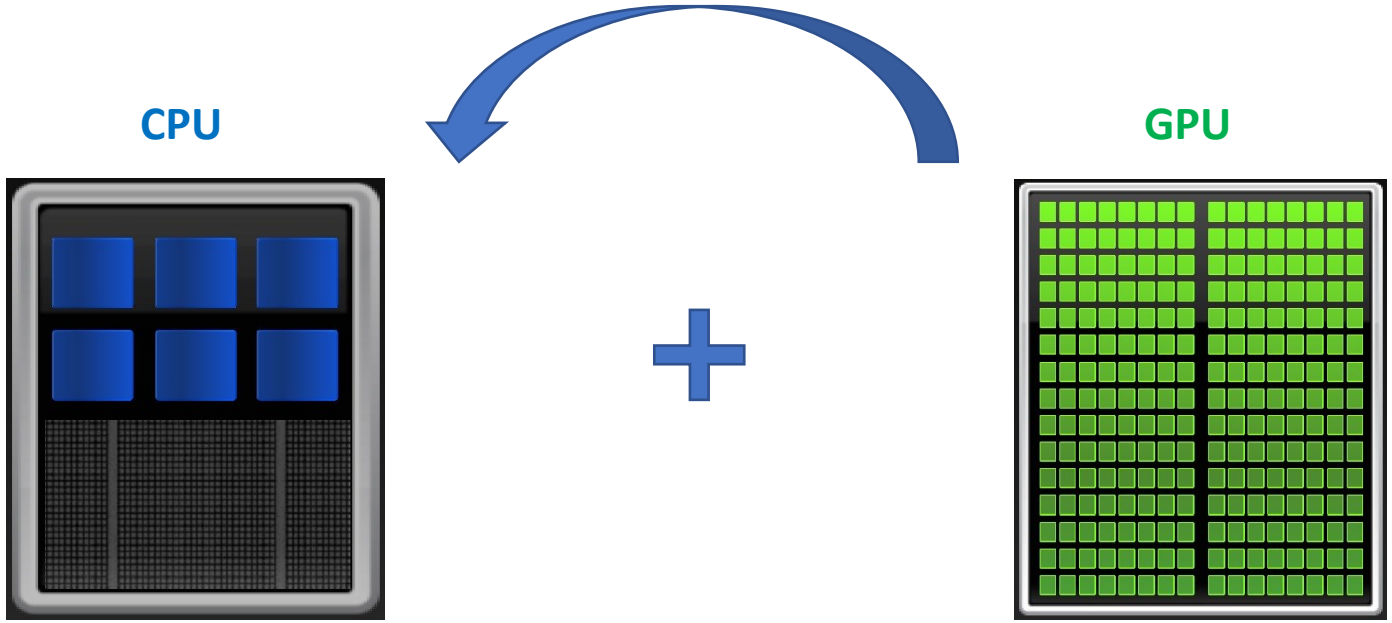
1. **Copy** input data from CPU memory **to GPU memory**
2. Load **GPU program** and **execute** it

KERNEL



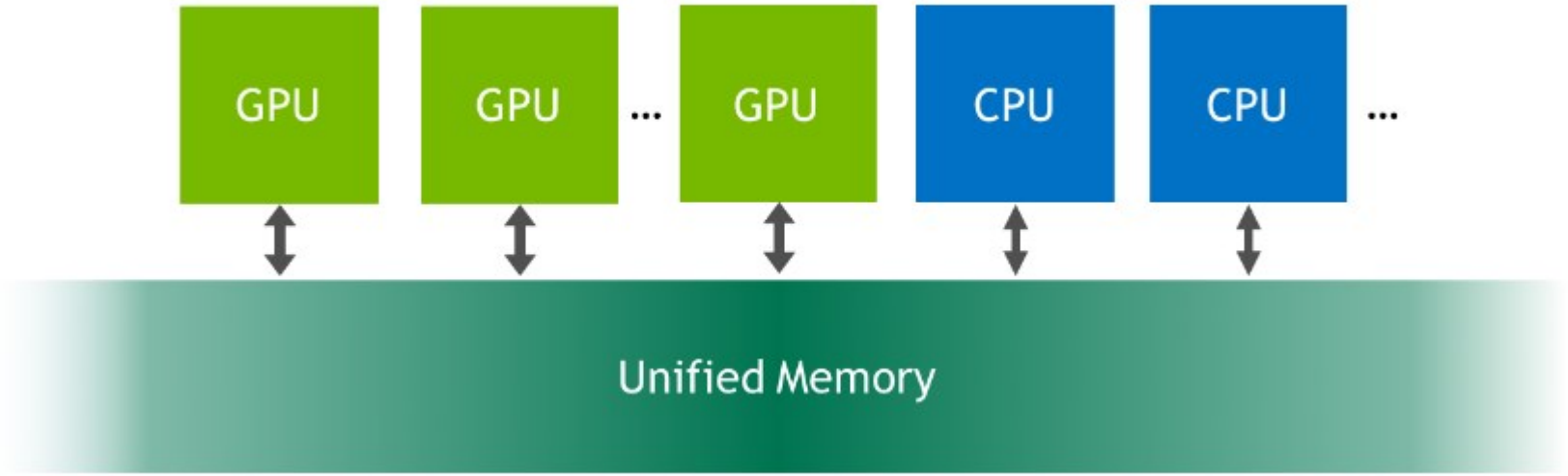
Basic 3-step process

1. **Copy** input data from CPU memory **to GPU memory**
2. Load **GPU program** and **execute** it
3. **Copy back** results from GPU memory **to CPU memory**



CUDA Unified Memory

- Introduced with CUDA 6.
- **Managed memory is accessible to both the CPU and GPU using a single pointer.**
- The **system automatically migrates data** allocated in Unified Memory between host and device.
- Up to GPU memory size (CUDA 8+: allocate up to system memory size).



https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

CUDA Unified Memory benefits

- **Simpler** Programming and Memory Model:
  - single allocation, single pointer, accessible everywhere,
  - eliminate the need of explicit copy,
  - simplify code porting.

- **Performance** Through Data Locality:
  - migrate data to accessing processor (complexity kept under the covers),
  - guarantee global coherence,
  - still allow explicit hand tuning (the CUDA runtime never has as much information as the programmer does about where data is needed and when!) → CUDA 8+ explicit prefetching API.

CUDA kernels

CUDA kernels

Functions running on GPU are called kernel

```c
#include <stdio.h>


void onCPU() {
    printf("This function runs on CPU\n");
}

int main() {
    onCPU();
}
```

```c
#include <stdio.h>

__global__
void onGPU() {
    printf("This function runs on GPU\n");
}

int main() {
    onGPU<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

CUDA kernels definition

.cu is the file extension for CUDA-accelerated program

Keyword `__global__`

- Indicates that the function will *run on GPU*.
- Can be *invoked globally* (both by CPU and GPU).
- It is required that a function defined with `__global__` returns `void`.
- **Device code.**

```
#include <stdio.h>

__global__
void onGPU() {
    printf("This function runs on
GPU\n");
}

int main() {
    onGPU<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

CUDA kernel launch

Kernel launch

- Function call to run on GPU.
- Must provide an **execution configuration** with the syntax <<<…>>> prior to passing kernel arguments.
- Specify the **thread hierarchy** for kernel launch:
  - number of **blocks** (group of threads),
  - number of **threads** per block (*limited to 1024!!*).

```
#include <stdio.h>

__global__
void onGPU() {
    printf("This function runs on
GPU\n");
}

int main() {
    onGPU<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

CUDA synchronization

- Launching kernels is **asynchronous**: CPU code will continue to execute without waiting for kernel to be completed.
- **`cudaDeviceSynchronize()`** will force the host to wait until device code completeness.

```
#include <stdio.h>

__global__
void onGPU() {
    printf("This function runs on
GPU\n");
}

int main() {
    onGPU<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

Compiling CUDA programs

Prepare your environment

- **Load CUDA**, on CINECA M100 cluster:
  `module load cuda hpc-sdk`

- **Information** on **CUDA device** (System Management Interface):
  `nvidia-smi`

- **Compiler**:
  ```
  nvcc -arch=sm_70  -o first_kernel 1_cuda_first_kernel.cu
  -run
  ```

  - The host code will be compiled by the host compiler; while the device code will be processed by NVIDIA compiler.
  - `-arch` flag indicates for which *architecture* the code should be compiled (`sm_70` is for TESLA V100 GPU).
  - `-run` flag *executes* the successfully compiled binary.

Your first exercise: 1_cuda_first_kernel

1. What is the output of your first CUDA program?

2. What happens if you remove the `__global__` keyword from the kernel?

3. What happens if your remove the configuration from your kernel launch?

4. What happens if you remove the call to `cudaDeviceSynchronize()` ?

Execution model

GPU work hierarchy

Grid: a *collection of thread blocks* (first level):

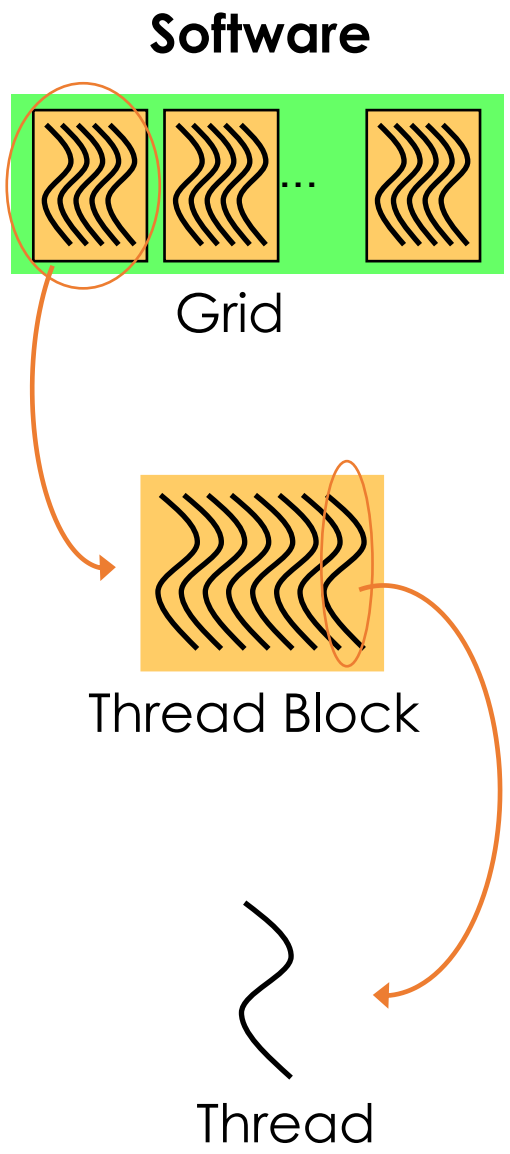
- thread blocks do not synchronize with each other;
- communication between blocks is expensive.

Thread Block: *a group of threads:*

- threads within a block can cooperate (light-weight synchronization).

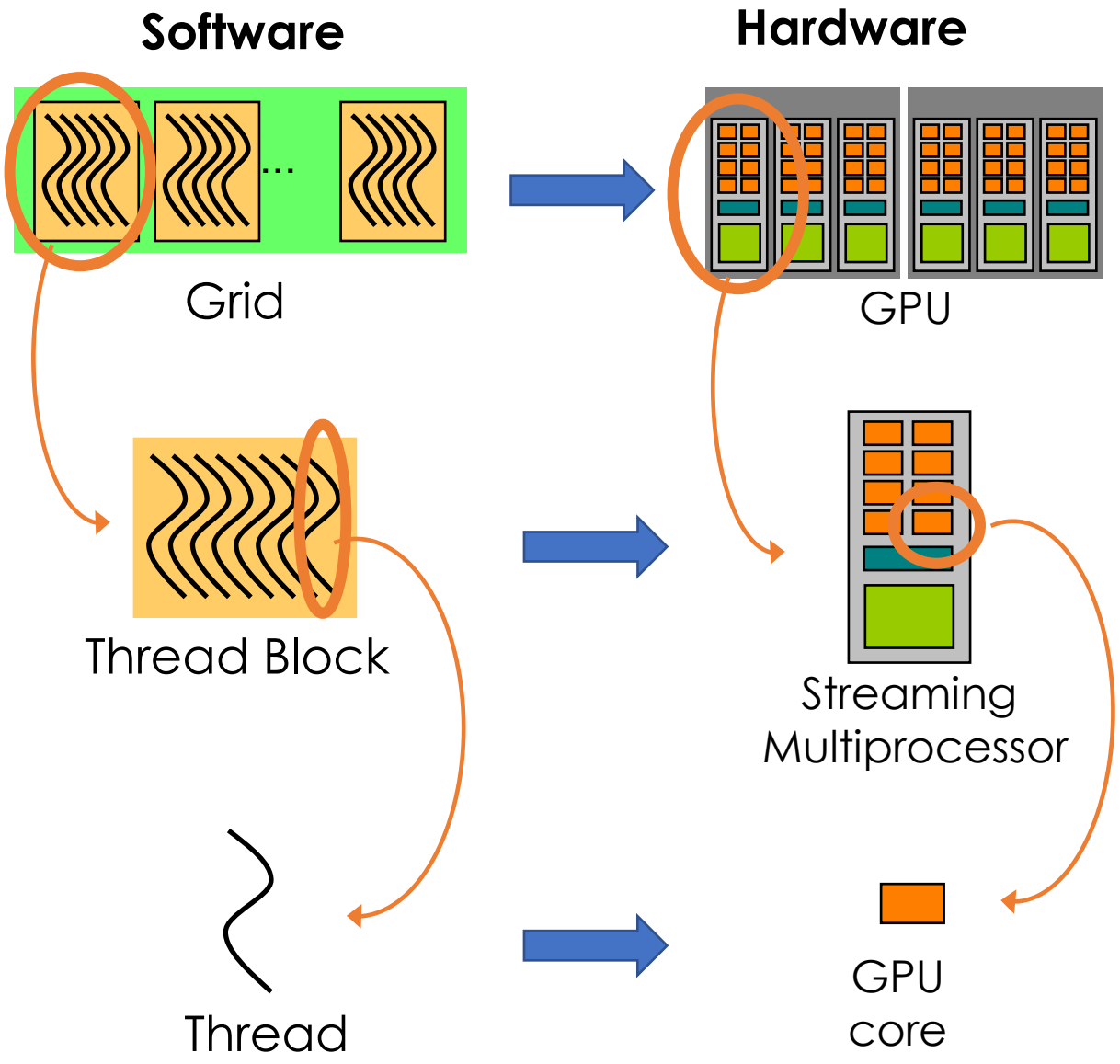
Thread: a *sequential execution unit:*

- all threads execute same sequential program,
- threads execute in parallel,
- a "**warp**" is a set of 32 threads which execute the same instruction.



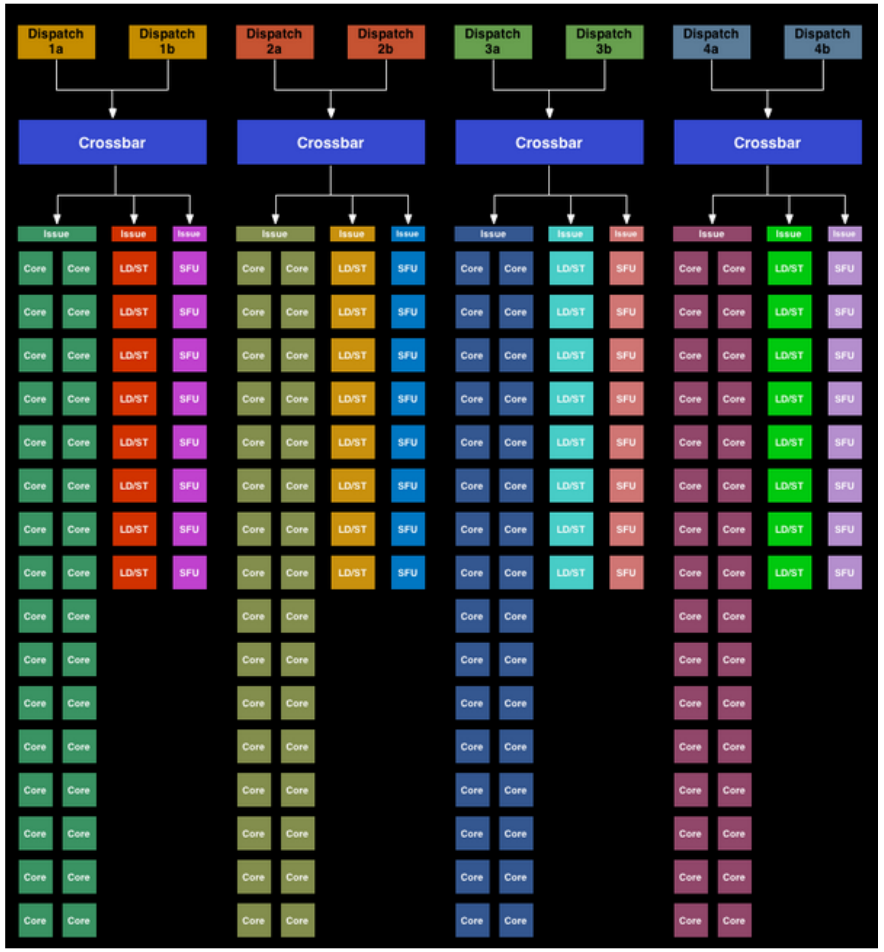
GPU execution model

- When a CUDA kernel is invoked:
- **thread blocks** are **assigned** to **SMs** in a round robin mode;
- the thread block remains on the SM until all its threads have finished (no communication between thread blocks).
- Threads of each thread block are partitioned into **warps** of consecutive threads.
- **Each thread** in a warp executes the same instruction simultaneously, with each thread being managed **by one GPU core.**



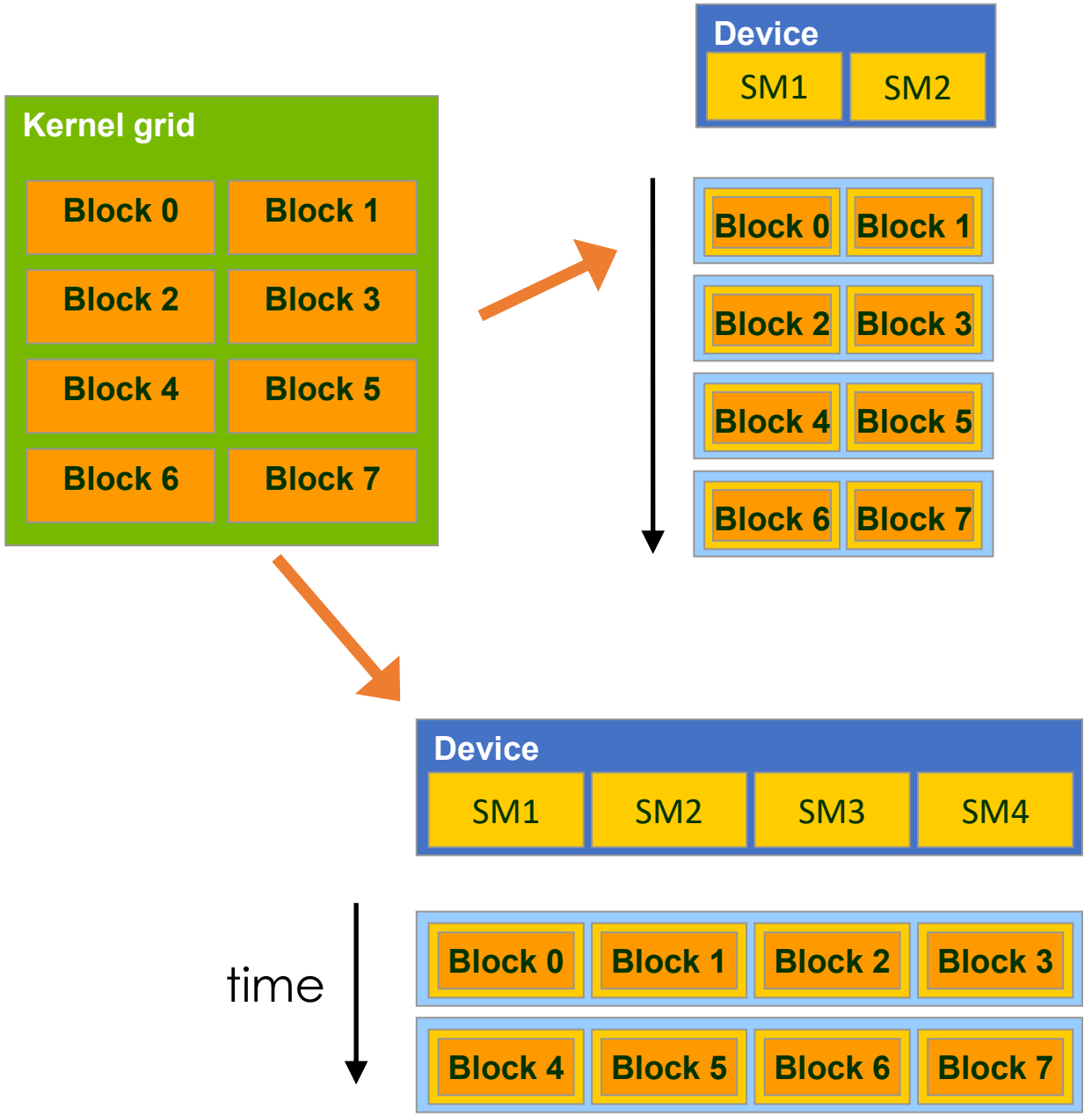
Warp scheduler

- NVIDIA SM schedules threads in **warps** (groups of 32 threads).
- **Warp schedulers** allows warps to be issued and executed concurrently if hardware resources are available.
- **Volta** SM has 4 warp schedulers, each one is responsible for:
  - feeding 32 CUDA cores,
  - 8 load/store units,
  - 8 special functions unit.



Transparent scalability

- The GPU runtime system can **execute blocks in any order** relative to each other.
- This flexibility enables to execute the **same application** code **on hardware** with **different** numbers of SMs.



Hiding latencies

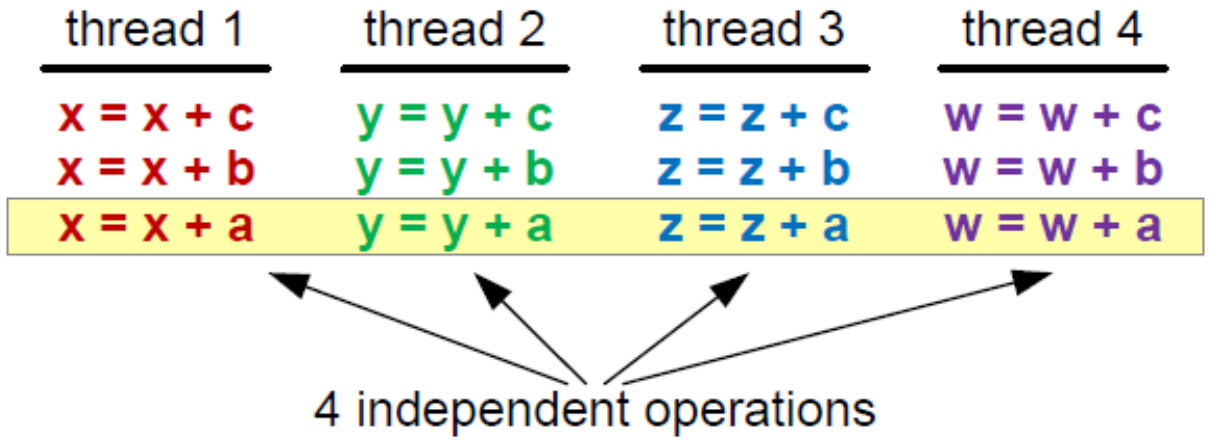
- **Latency** is the n*umber of clock cycles needed to complete an instruction*, aka the number of cycles we need to wait for before another dependent operation can start:
  - arithmetic latency (~ 18-24 cycles),
  - memory access latency (~ 400-800 cycles).

- Latency can not be discarded (hardware limitation), but its effect can be controlled (**hidden**) by:
  - **saturating computational pipelines** in computational bound problems,
  - **saturating bandwidth** in memory bound problems.

Hiding latencies

- The code need to be organized to **provide the scheduler a sufficient number of independent operations.**

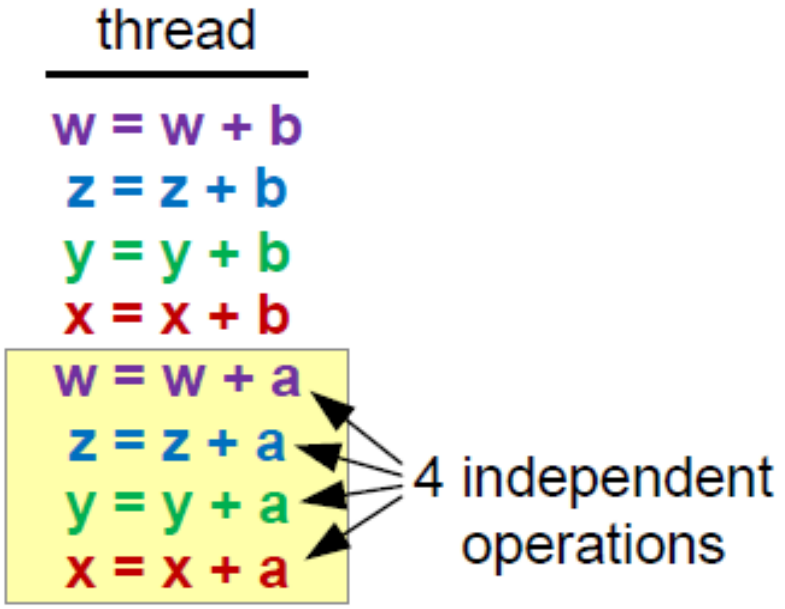
- There are **two possible ways** and paradigms to use, that can be combined.
  - ***Thread-Level Parallelism (TLP)***:
    - **high SM occupancy**: provide as much threads per SM as possible (when a scheduler is free it will find easily a warp to execute),
    - best approach for low number of independent operations per CUDA kernel.



Hiding latencies

- The code need to be organized to **provide the scheduler a sufficient number of independent operations.**

- There are **two possible ways** and paradigms to use, that can be combined:
  - *Thread-Level Parallelism (TLP)*
  - ***Instruction-Level Parallelism (ILP)***
    - **high number of multiple independent operations** inside CUDA kernel (each kernel act on a lot of data),
    - this will grant the scheduler to stay on the same warp and **fully load** each **hardware pipeline.**

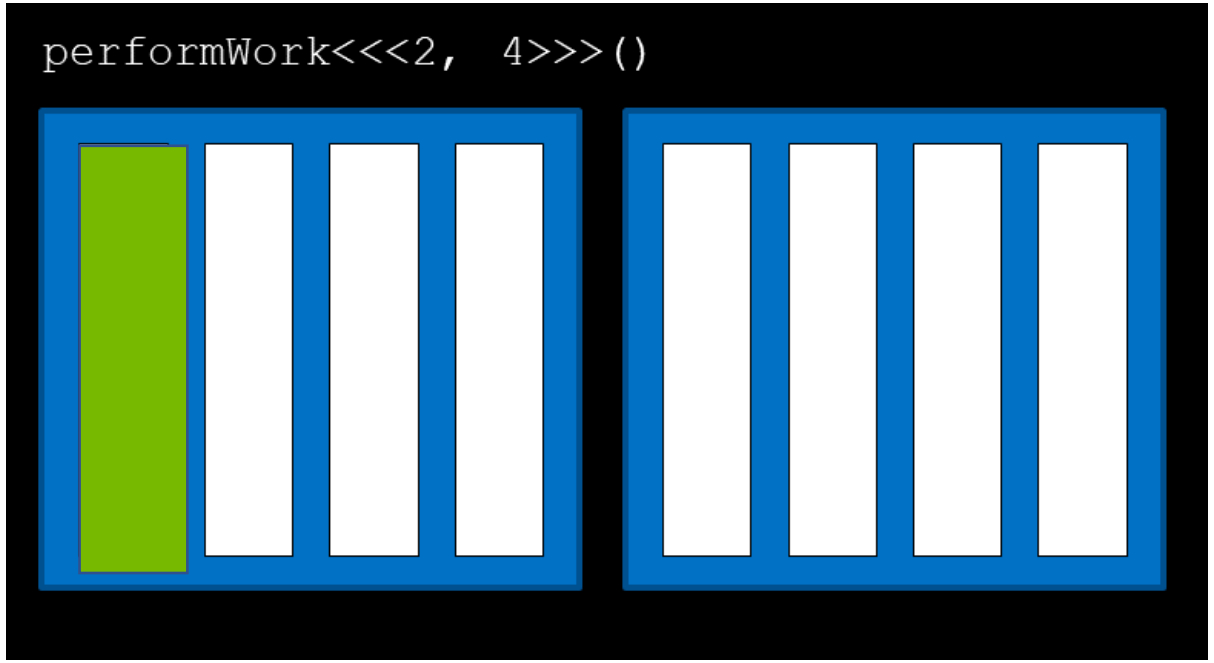


Kernel launch

Kernel execution

```
kernel<<<number_of_blocks, number_of_threads_per_block>>>();
```

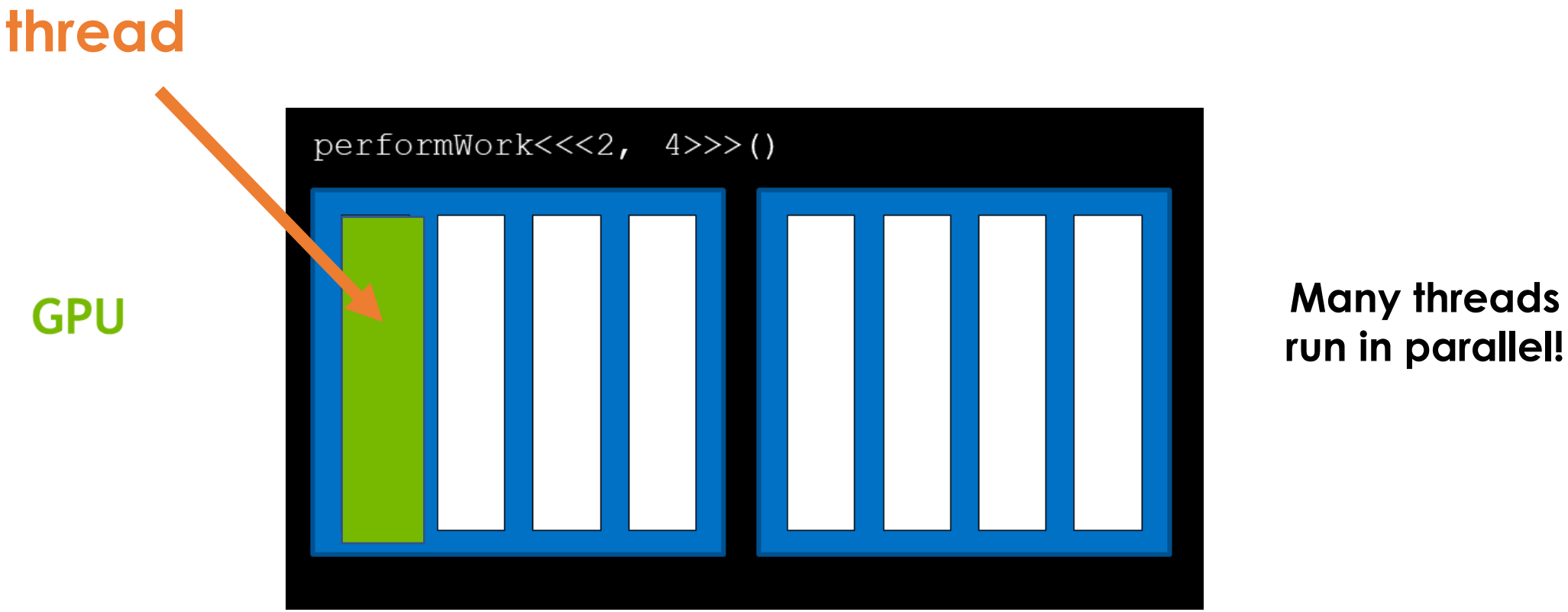
GPU



Many threads run in parallel!

Kernel execution

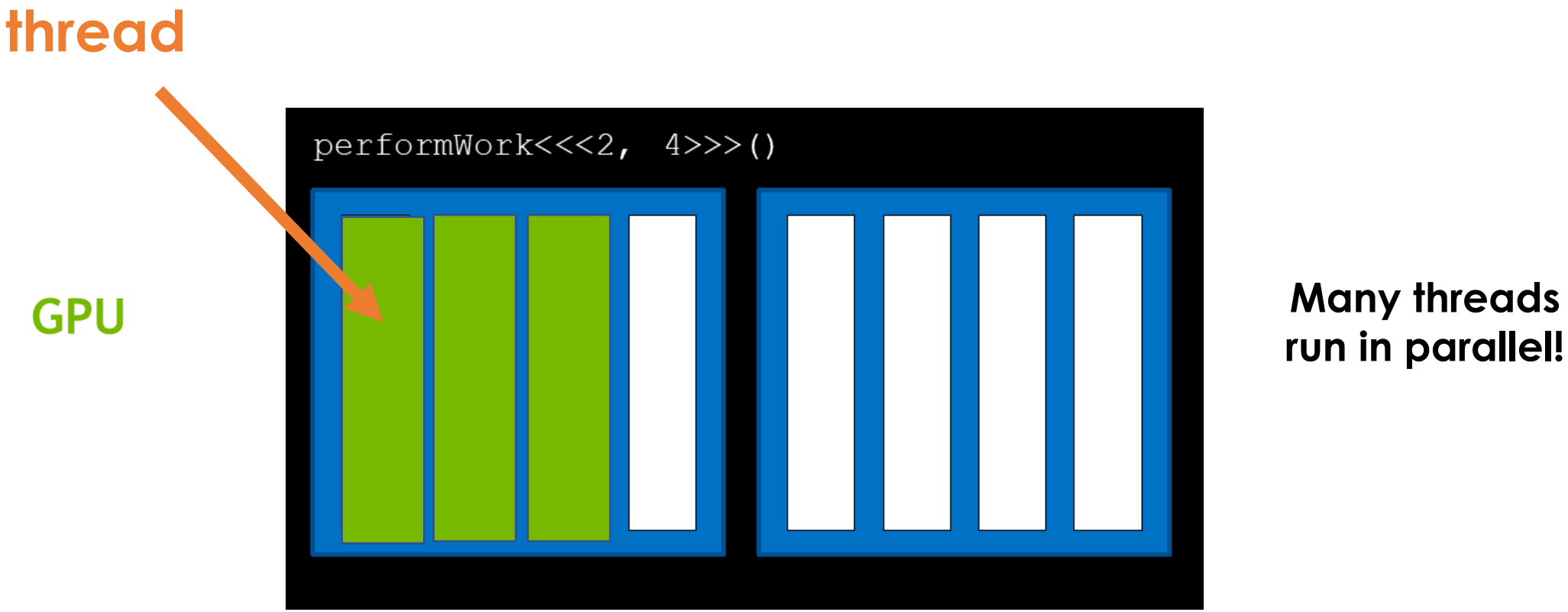
```
kernel<<<number_of_blocks, number_of_threads_per_block>>>();
```



Many threads run in parallel!

Kernel execution

```
kernel<<<number_of_blocks, number_of_threads_per_block>>>();
```

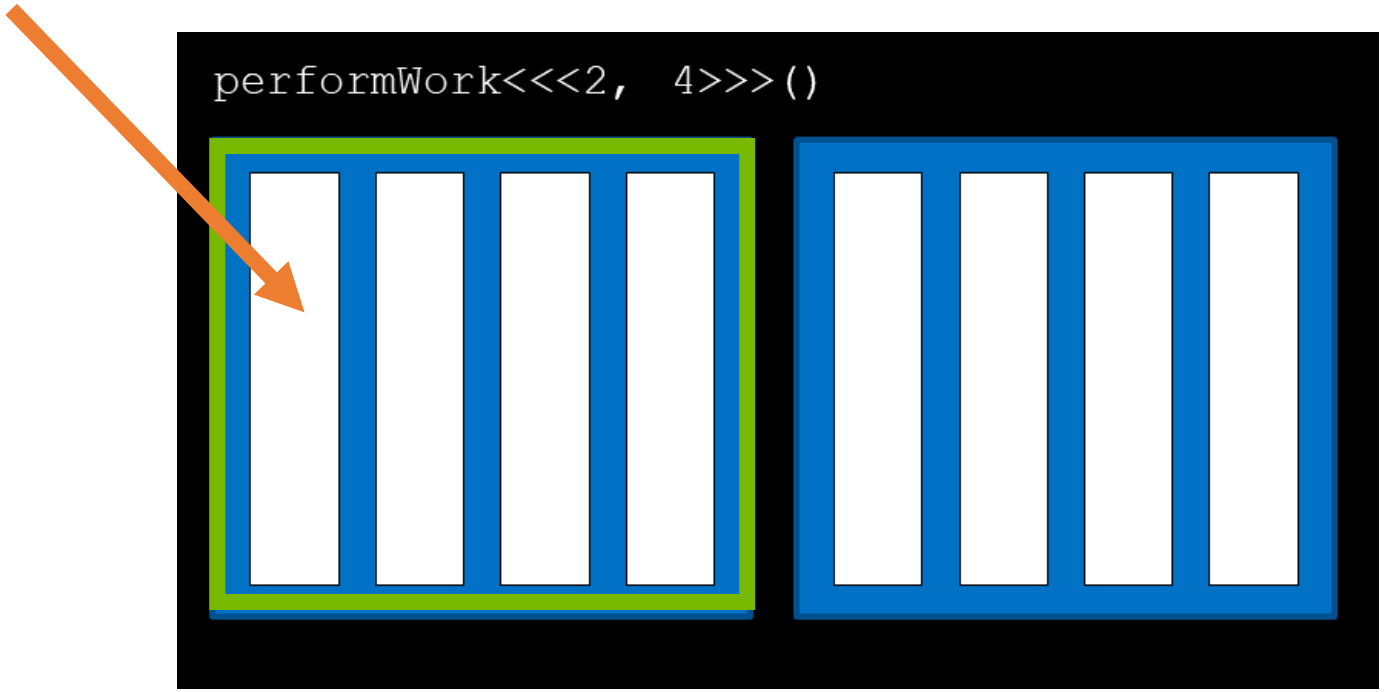


Many threads run in parallel!

Kernel execution

```
kernel<<<number_of_blocks, number_of_threads_per_block>>>();
```

block: collection of threads

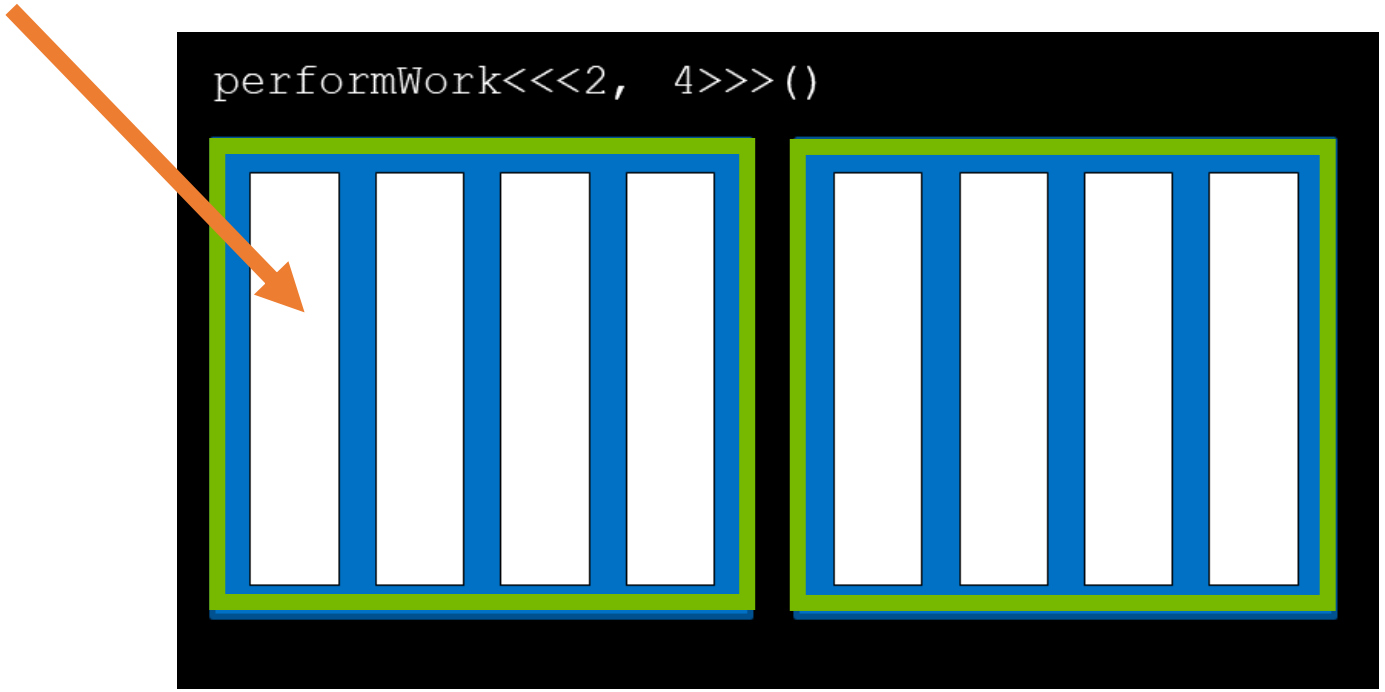


Every block of the grid contains the same number of threads!

Kernel execution

```
kernel<<<number_of_blocks, number_of_threads_per_block>>>();
```

block: collection of threads

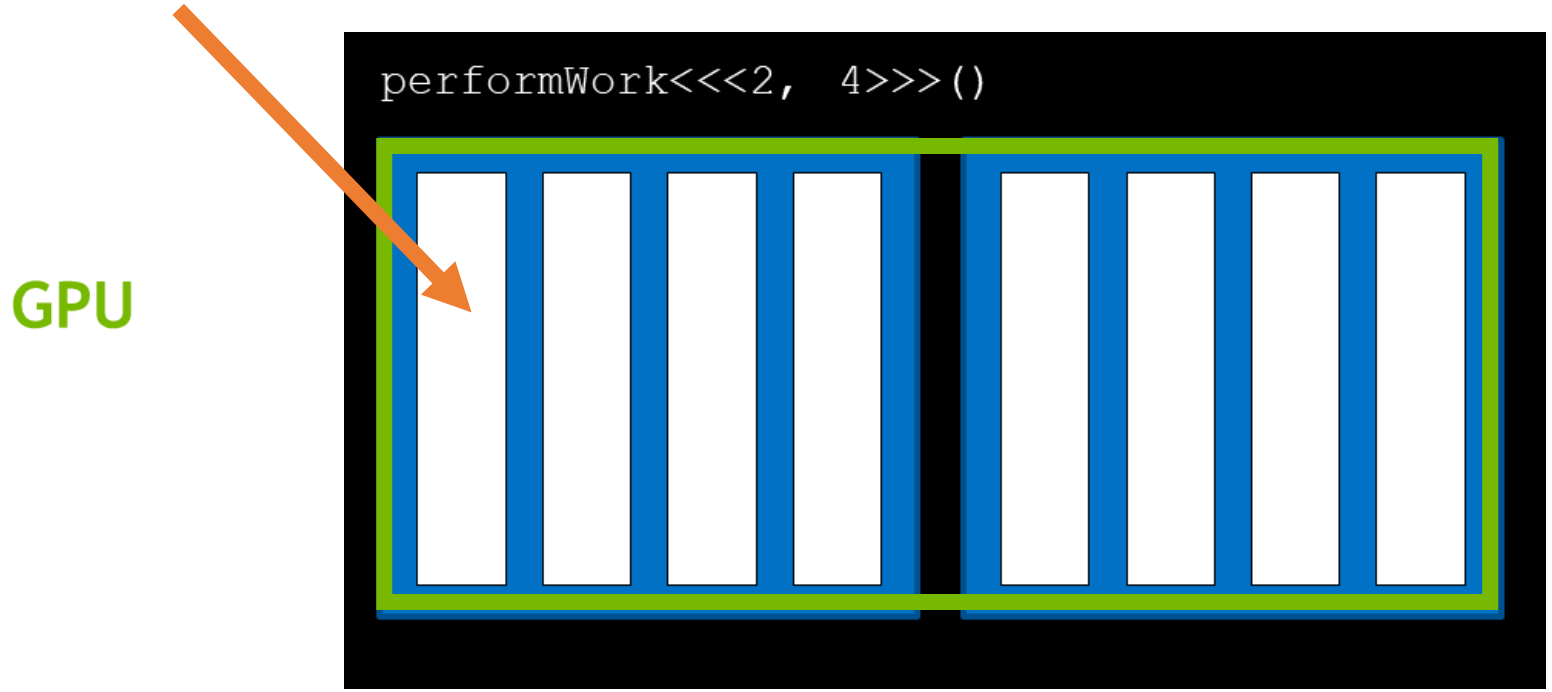


Every block of the grid contains the same number of threads!

Kernel execution

```
kernel<<<number_of_blocks, number_of_threads_per_block>>>();
```

grid: collection of blocks



Kernel execution configuration

```
kernel<<<number_of_blocks, number_of_threads_per_block>>>();
```

- `kernel<<<1, 1>>>()` is configured to run in a single thread block which has a single thread, therefore it will run only once.
- `kernel<<<1, 10>>>()` is configured to run in a single thread block which has 10 threads, therefore it will run 10 times.
- `kernel<<<10, 1>>>()` is configured to run in 10 thread blocks which each have a single thread, therefore it will run 10 times.
- `kernel<<<10, 10>>>()` is configured to run in 10 thread blocks each of which has 10 threads, therefore it will run 100 times.

Exercise: 2_first_parallel.cu

1. Refactor `firstParallel` function to be launched as CUDA kernel.

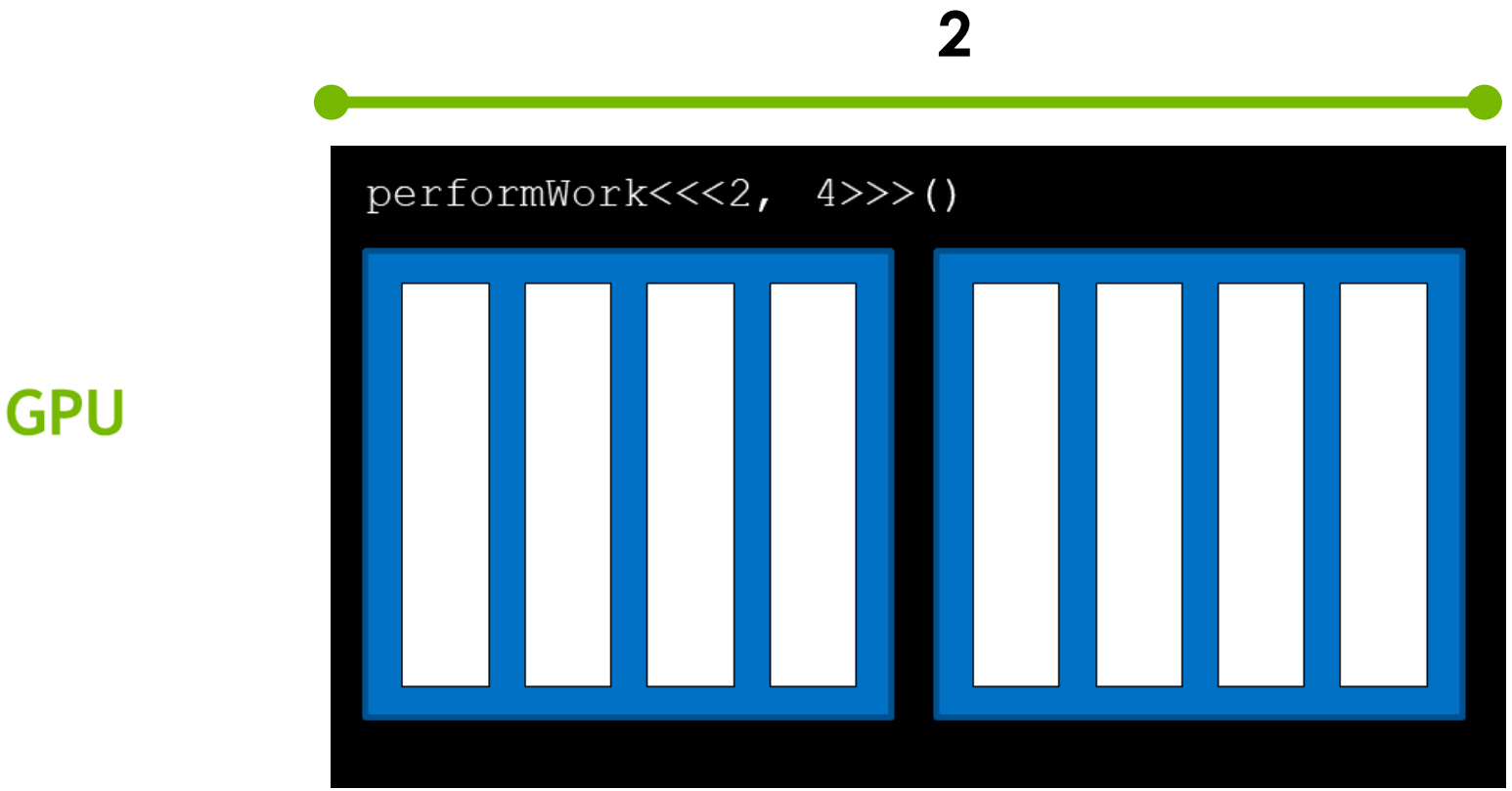
2. Rearrange the execution configuration to run the kernel in parallel on 5 threads, in a single thread block. How many times is the message printed?

3. Rearrange the execution configuration to run the kernel in parallel on 3 thread blocks, each containing 5 threads. How many times is the message printed?

CUDA-provided thread hierarchy variables

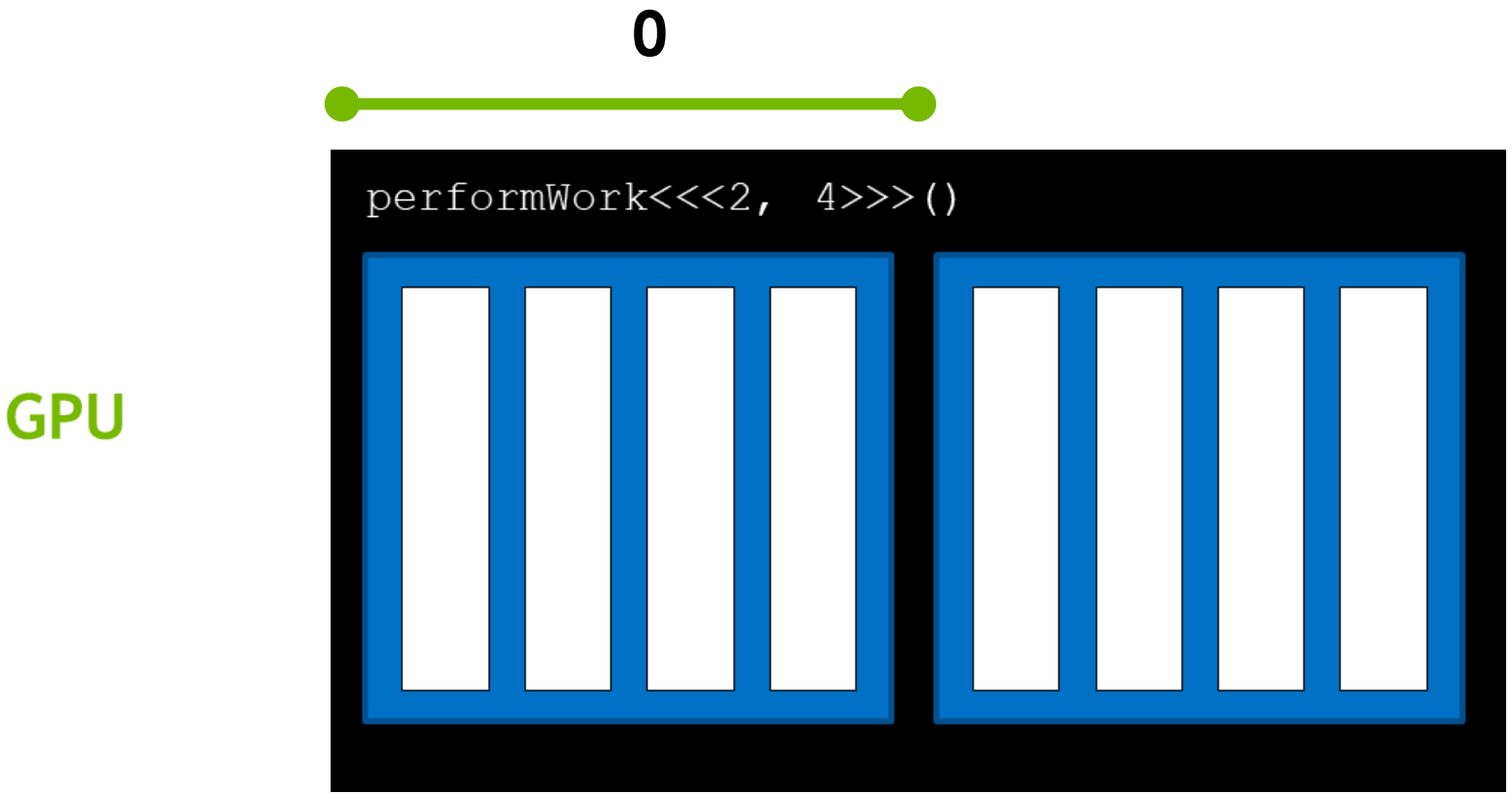
Thread hierarchy variables

gridDim.x: number of blocks in the grid



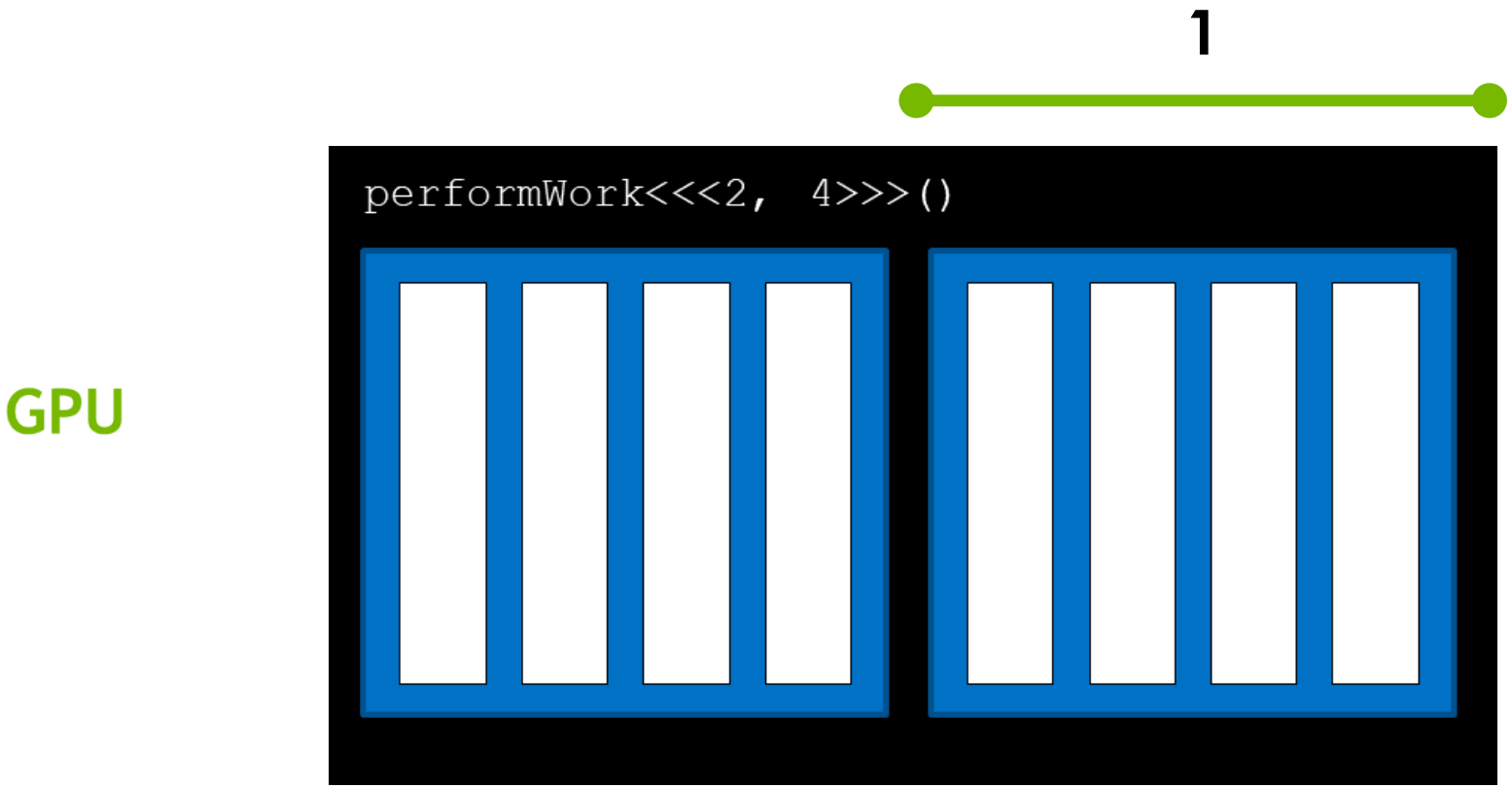
Thread hierarchy variables

blockIdx.x: index of current block within the grid



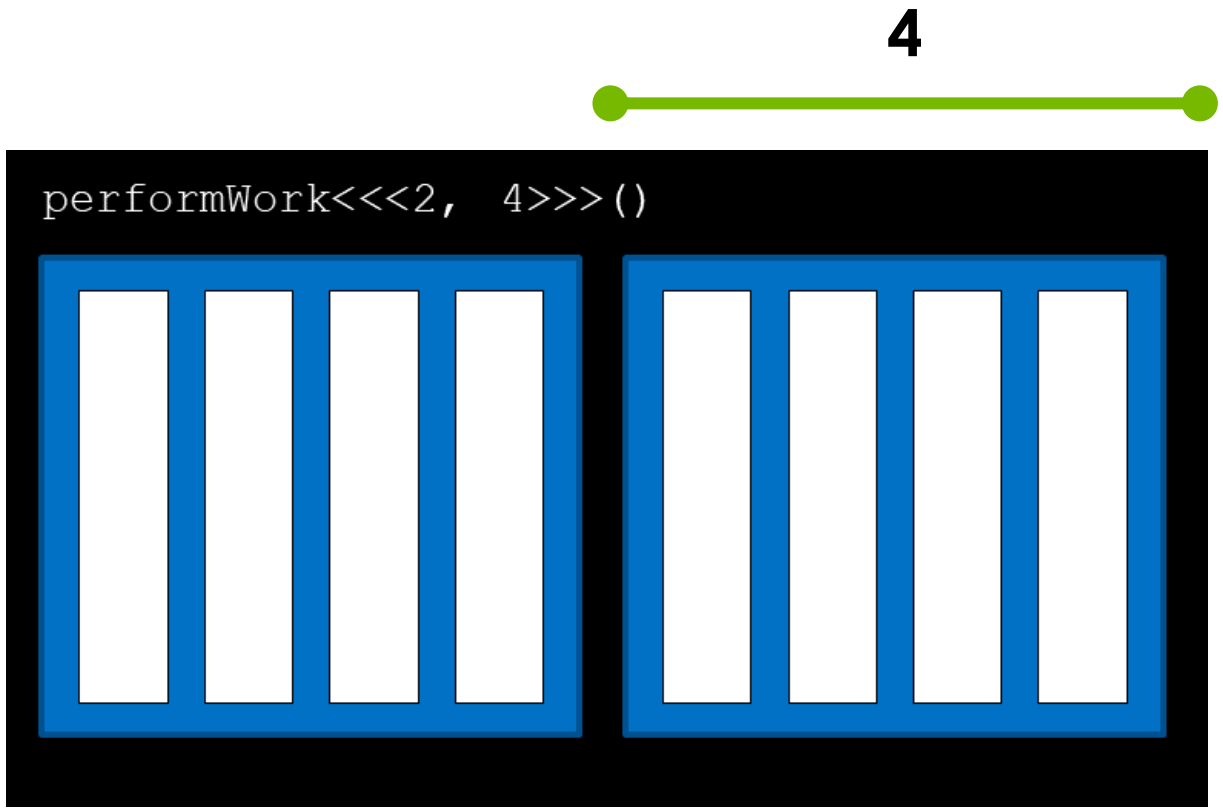
Thread hierarchy variables

blockIdx.x: index of current block within the grid



Thread hierarchy variables

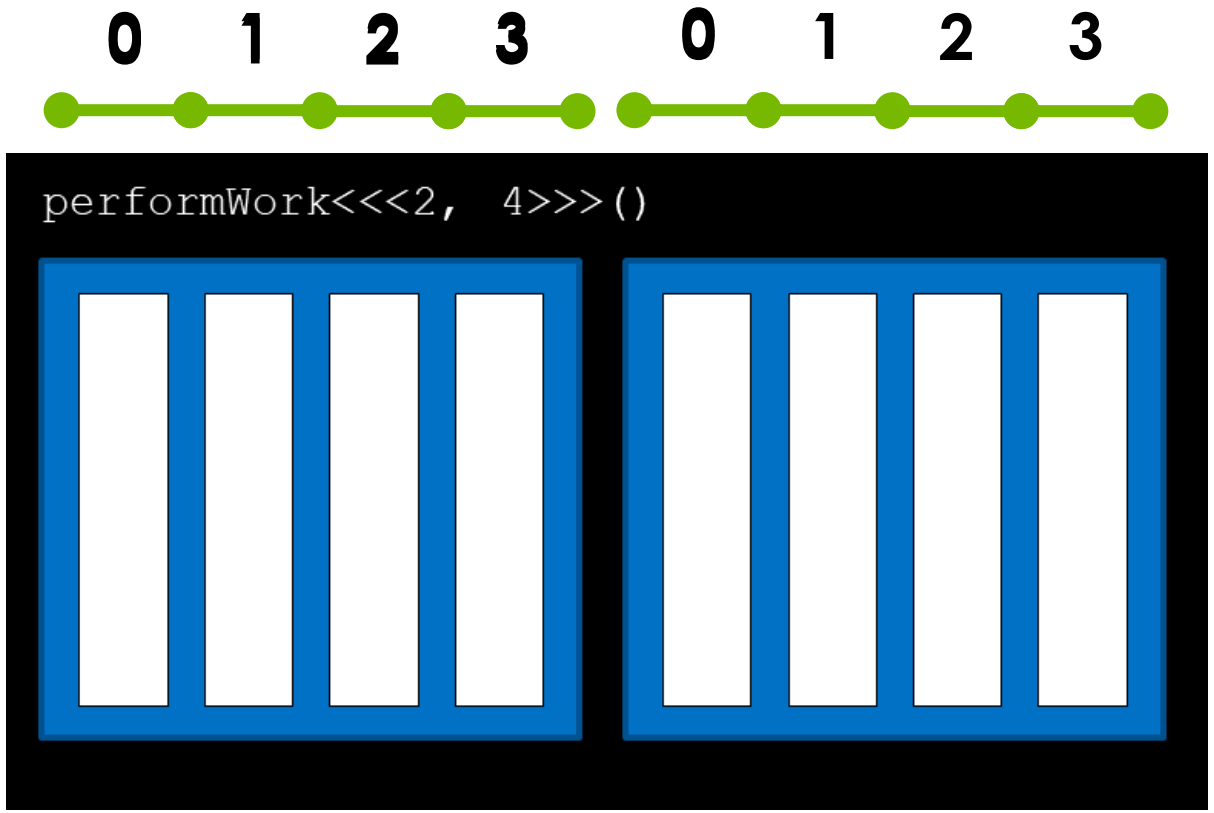
blockDim.x: number of threads per block



All blocks in a grid contain the same number of threads.

Thread hierarchy variables

threadIdx.x: index of the thread within a block



All blocks in a grid contain the same number of threads.

Exercise: 3_understand_threadIdx_blockIdx.cu

Refactor the code so that a "`Success`" message is printed.

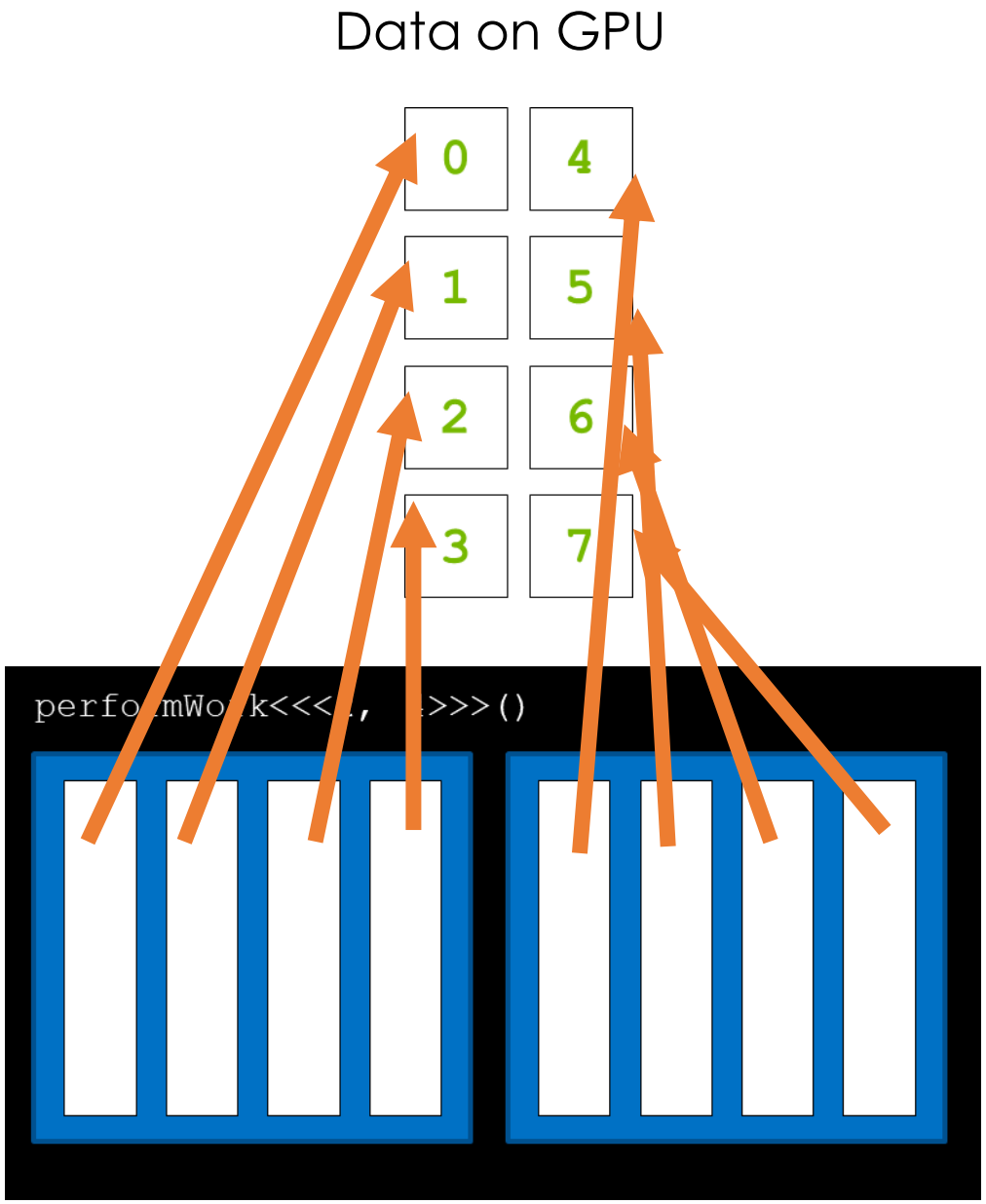
Exercise: 4_accelerating_loops.cu

1. Refactor the code so that the function `doLoop()` will run on GPU.
2. Refactor the kernel so that each iteration can be run in parallel. The new kernel should only do the work of one iteration of the original loop. Use a single block of threads. Numbers from 0 to `N` should be printed, as for the CPU code.

Global index

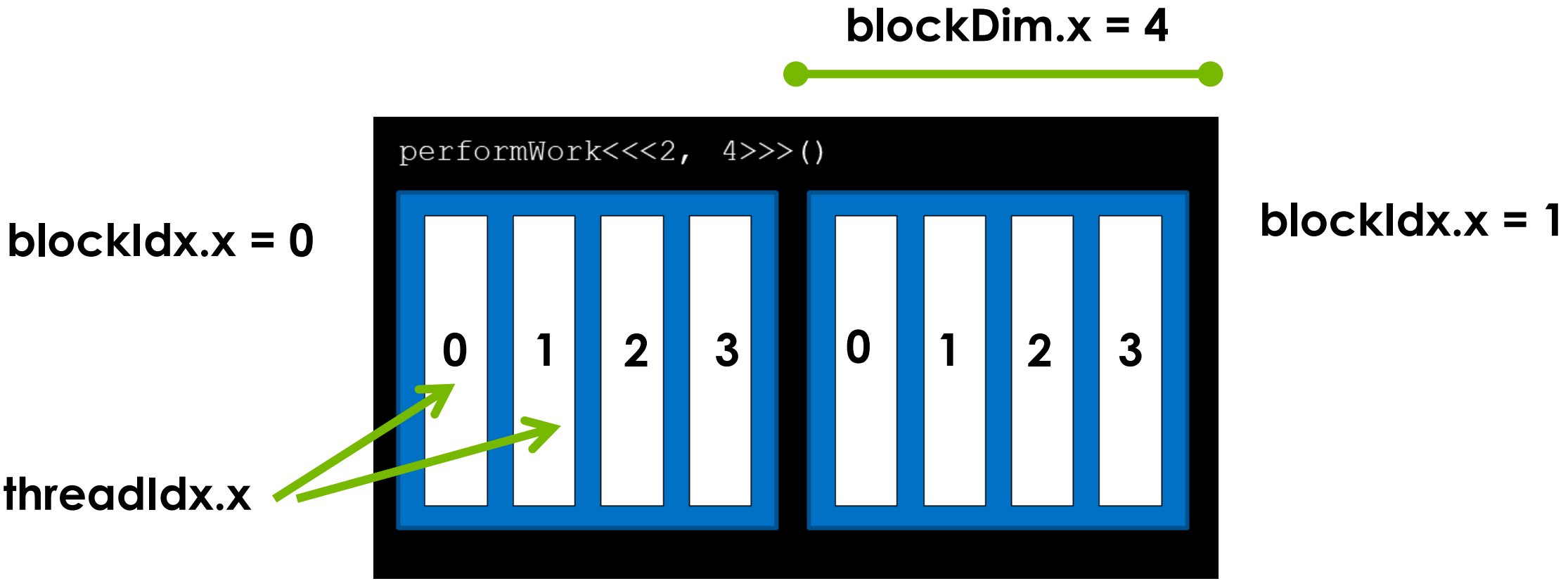
Coordinate parallel threads

- A **limited number of threads** (**1024**) can fit inside a thread block.
- To increase parallelism, we need to **coordinate** work **among thread blocks**.
- This is achieved by **mapping** element of data vector to threads using **global index**.



Global index

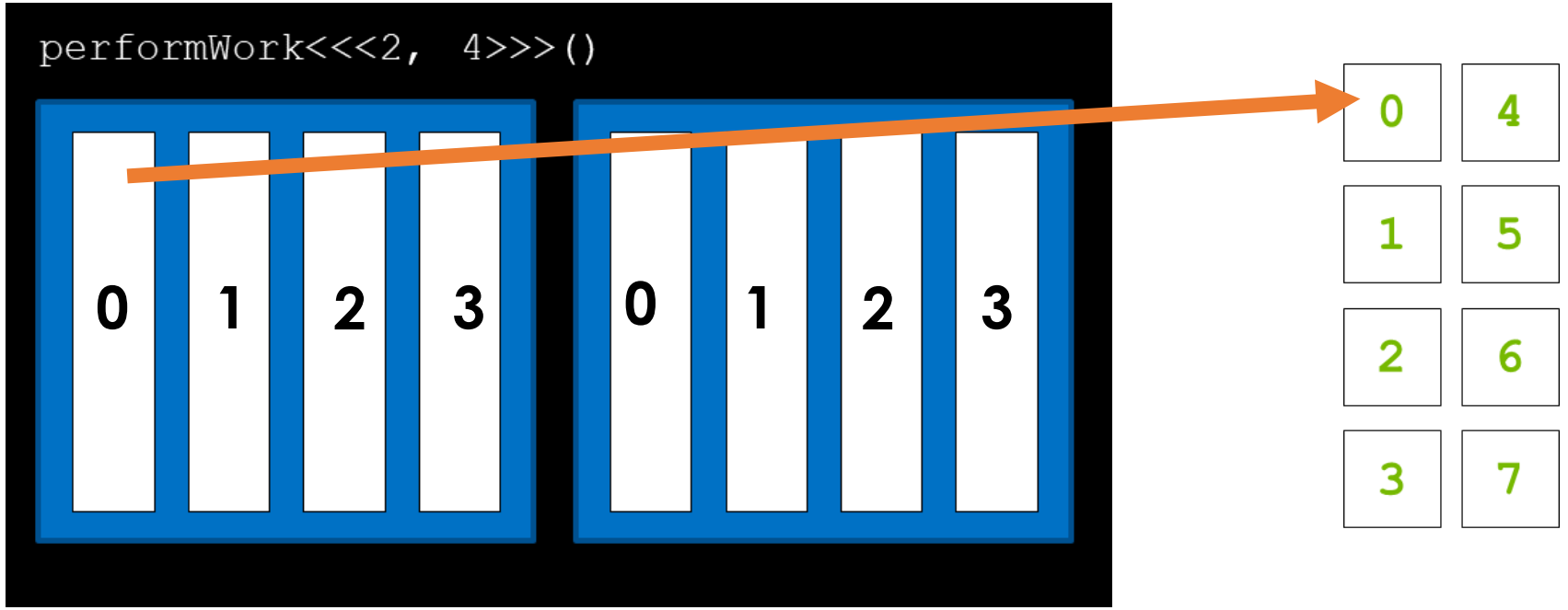
```
threadIdx.x + blockIdx.x * blockDim.x
```



Global index

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

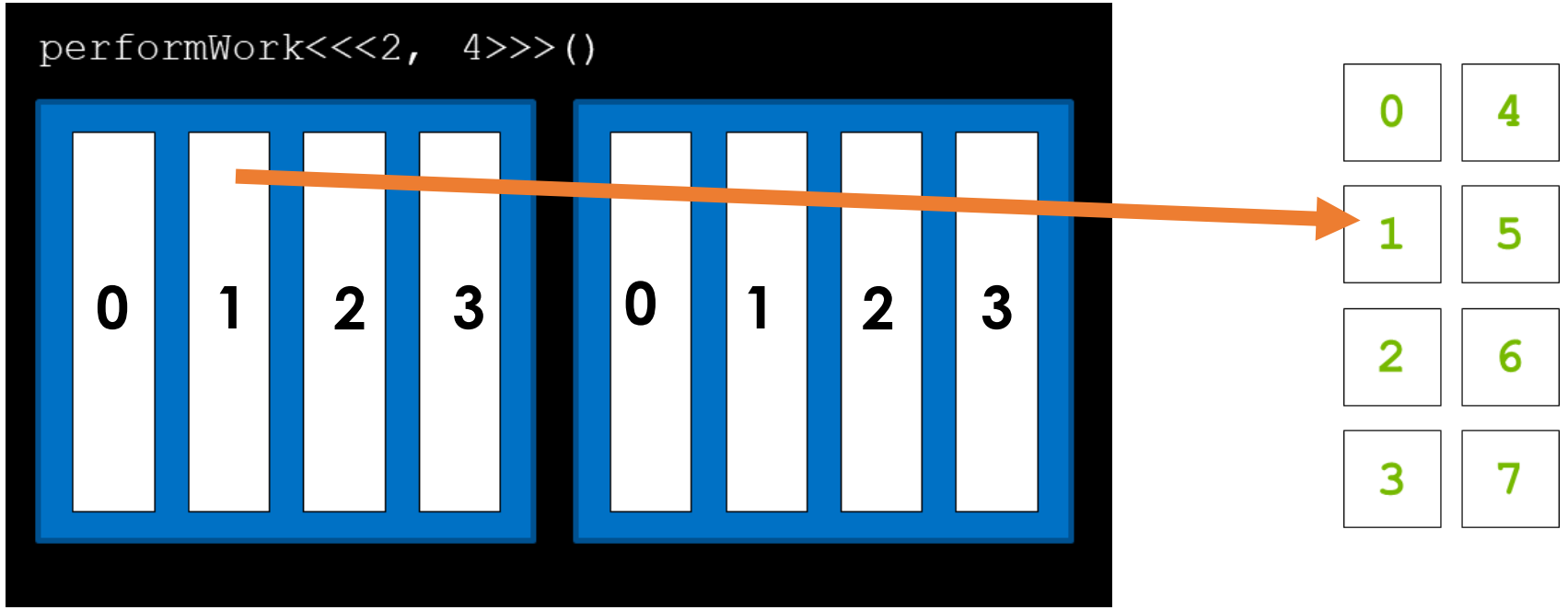
```
dataIdx = 0 + 0 * 4 = 0
```



Global index

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

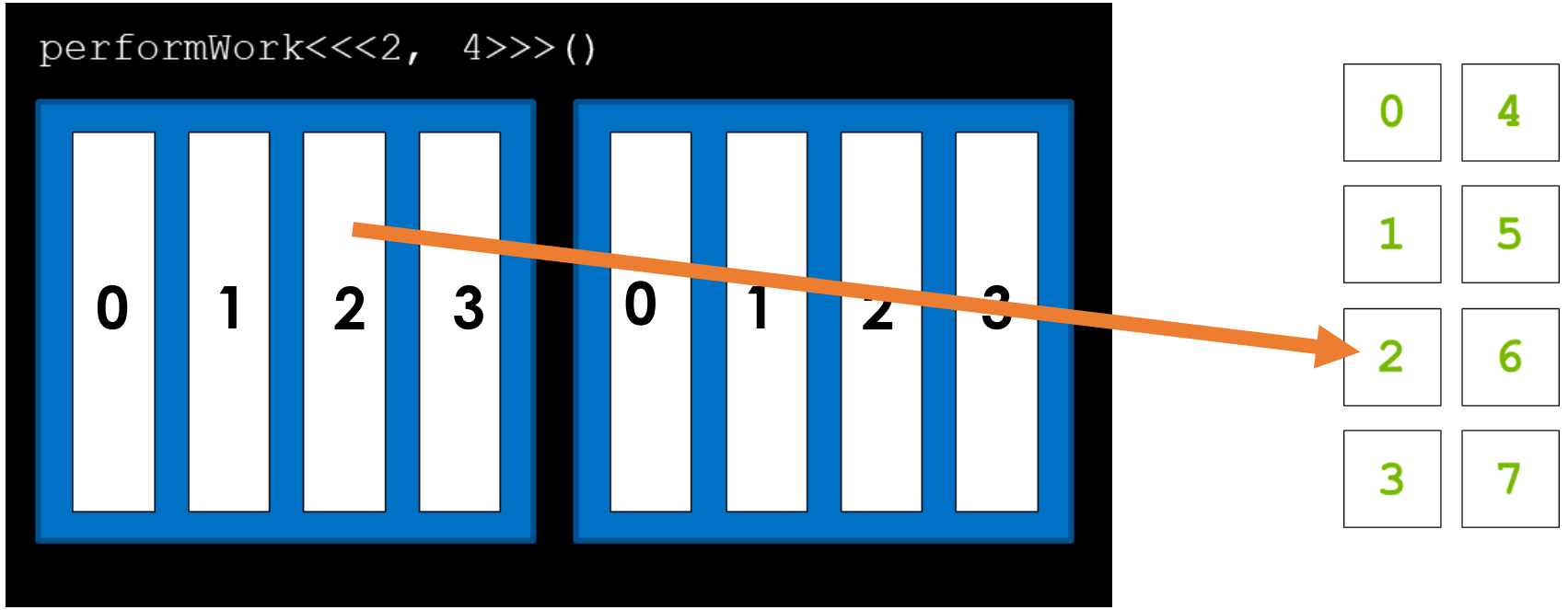
```
dataIdx = 1 + 0 * 4 = 1
```



Global index

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

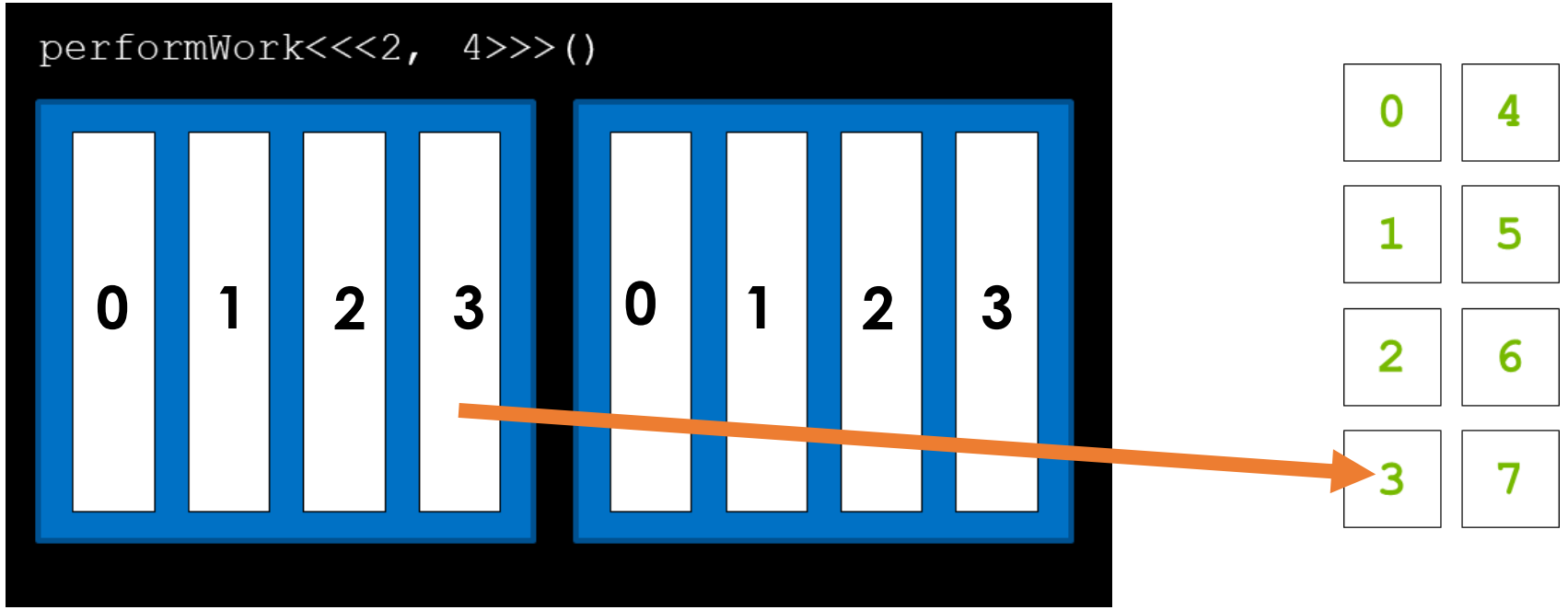
```
dataIdx = 2 + 0 * 4 = 2
```



Global index

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

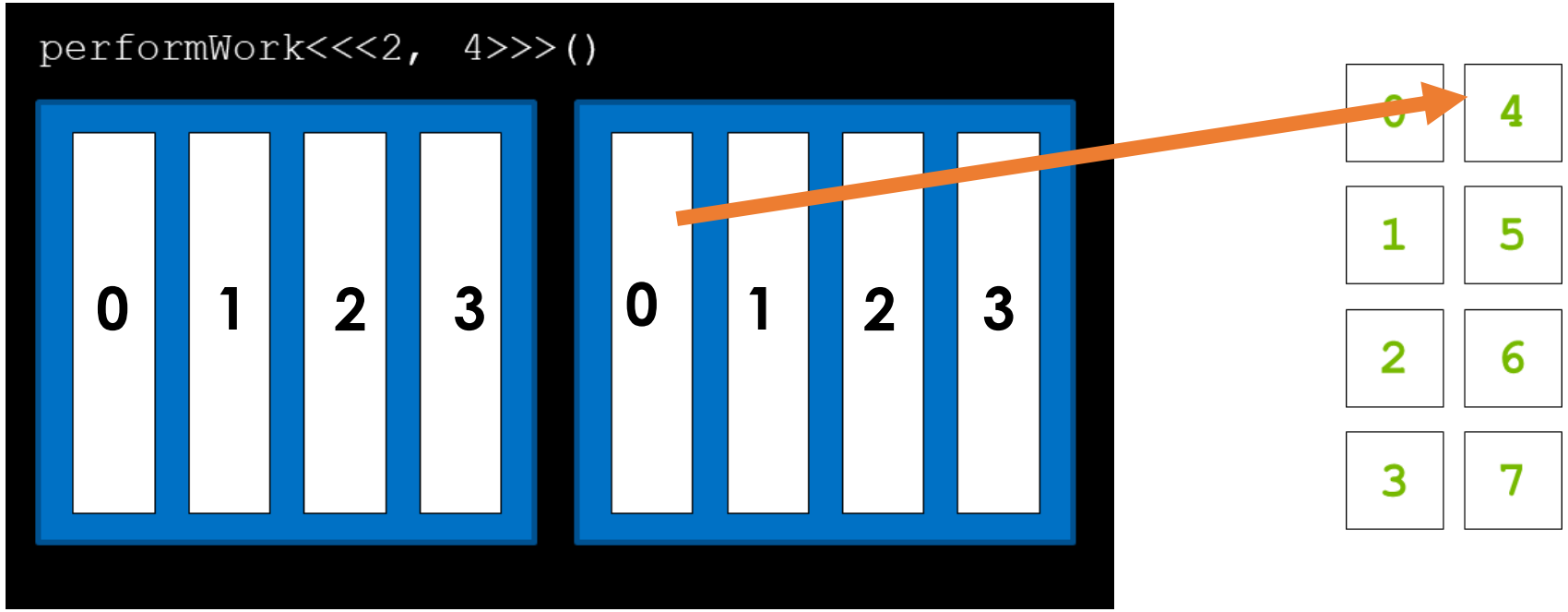
```
dataIdx = 3 + 0 * 4 = 3
```



Global index

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

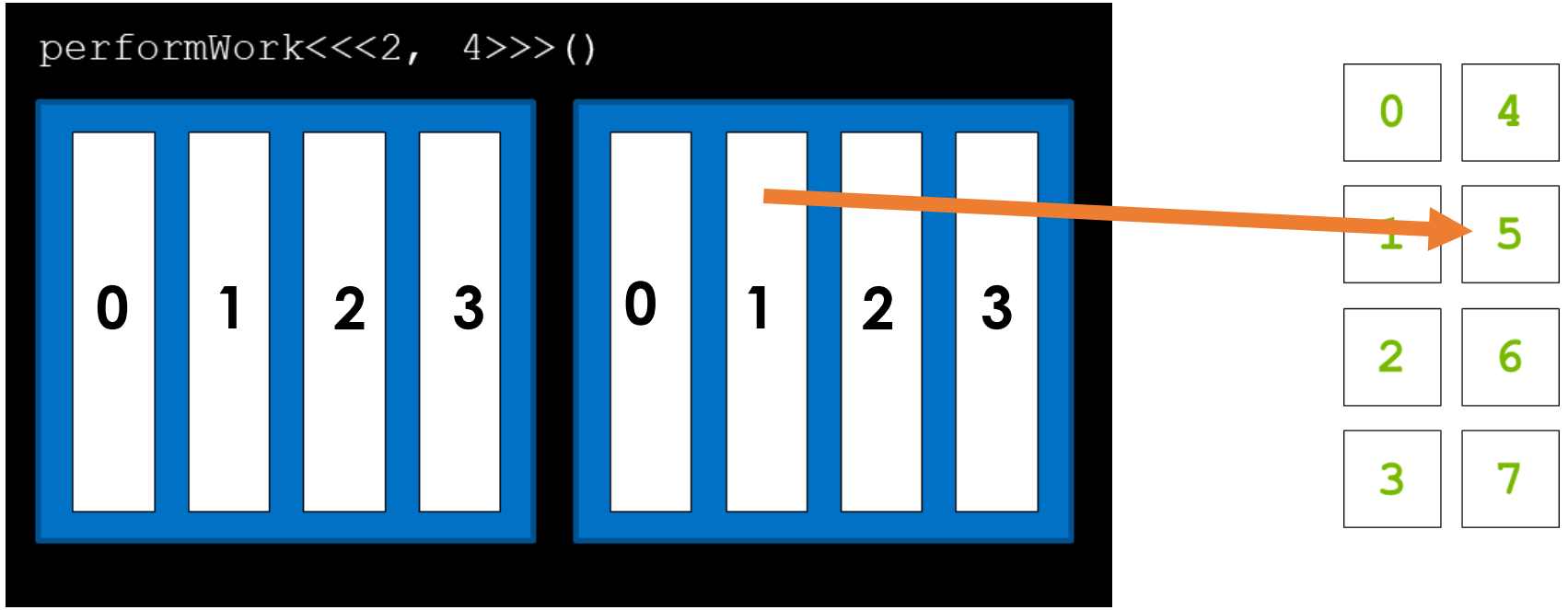
```
dataIdx = 0 + 1 * 4 = 4
```



Global index

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

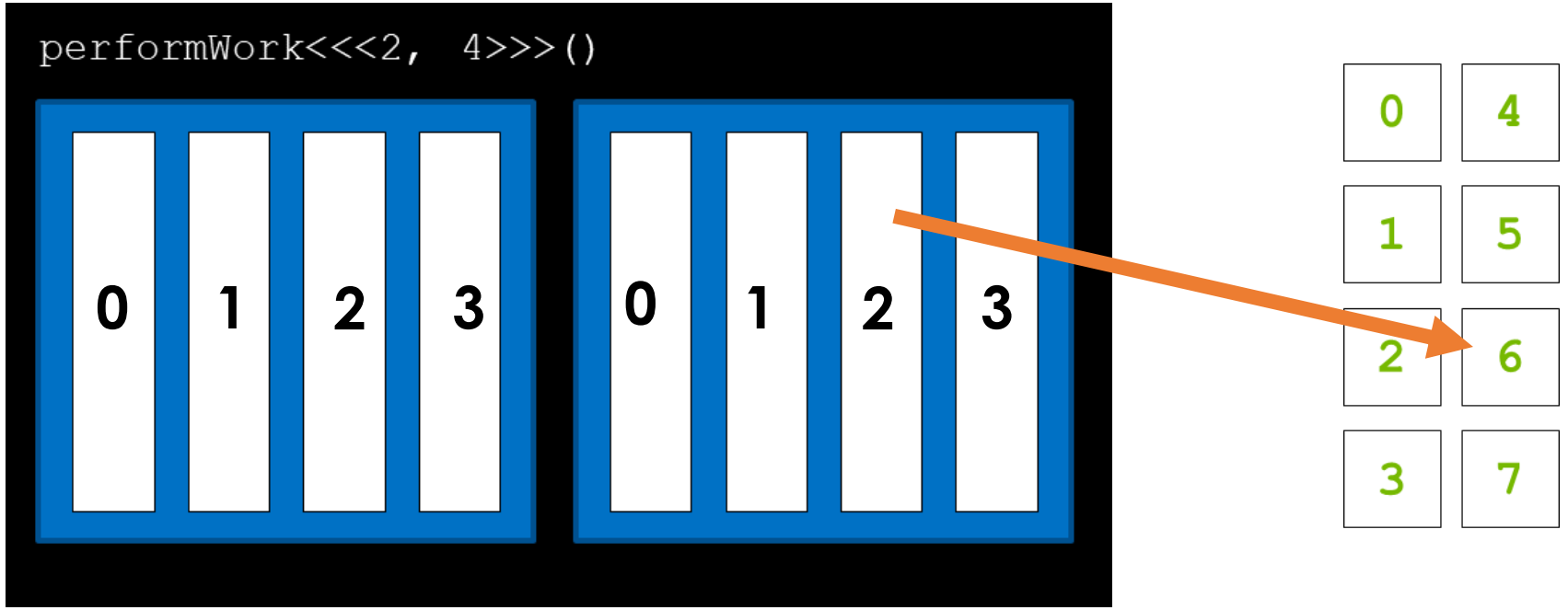
```
dataIdx = 1 + 1 * 4 = 5
```



Global index

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

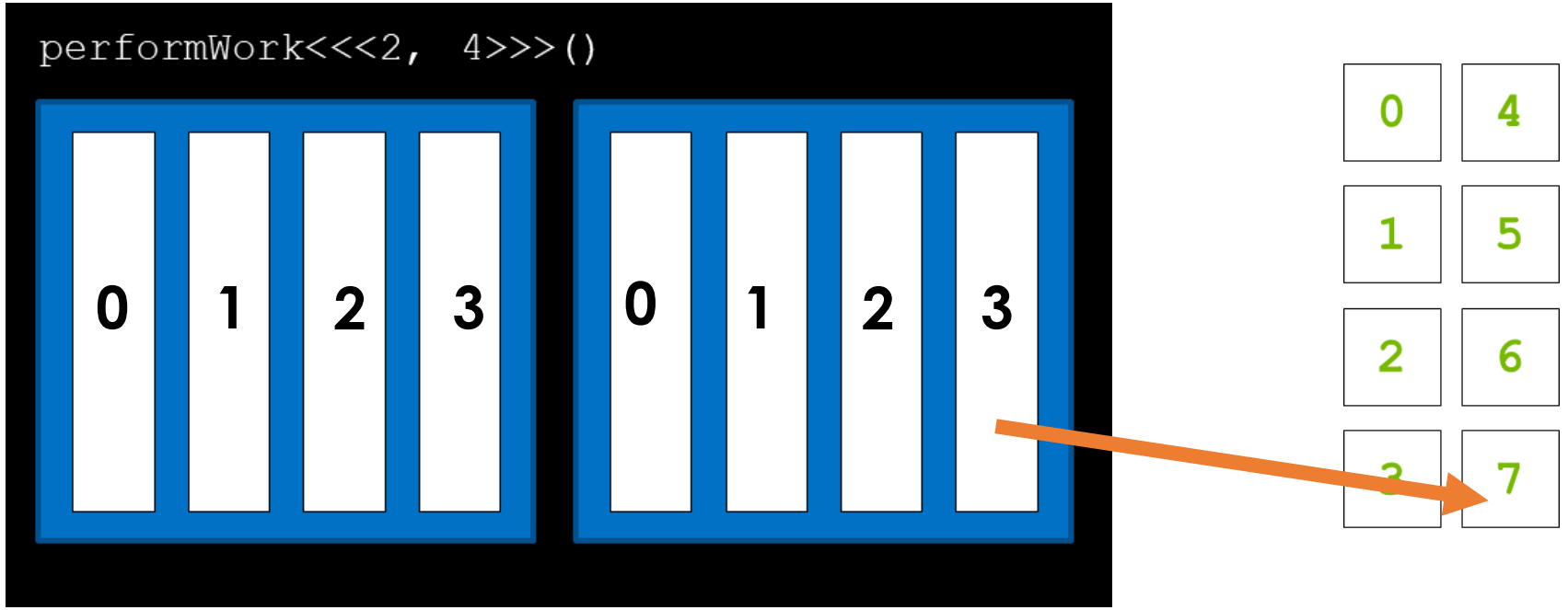
```
dataIdx = 2 + 1 * 4 = 6
```



Global index

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

```
dataIdx = 3 + 1 * 4 = 7
```



Exercise: 5_multi_block_loops.cu

1. Refactor the code so that the function `doLoop()` will run on GPU (same starting point as exercise 4_accelerating_loop.cu).

2. Refactor the kernel so that each iteration can be run in parallel. The new kernel should only do the work of one iteration of the original loop. Numbers from 0 to `N` should be printed, as for the CPU code.
   Additional constraint: use a kernel configuration that launches at least 2 blocks of threads.
   How many kernel configurations can be used to solve this problem? Only one or more than one?

References

References

- Previous editions of this school at CINECA
- Oakridge National Laboratory's "Introduction to CUDA C++": https://www.olcf.ornl.gov/calendar/introduction-to-cuda-c/
- NVIDIA DL Institute Online Course: **main source of exercises**
- blogs.nvidia.com
- unsplash.com
- Wikipedia

THANK YOU!

Lara Querciagrossa
l.querciagrossa@cineca.it