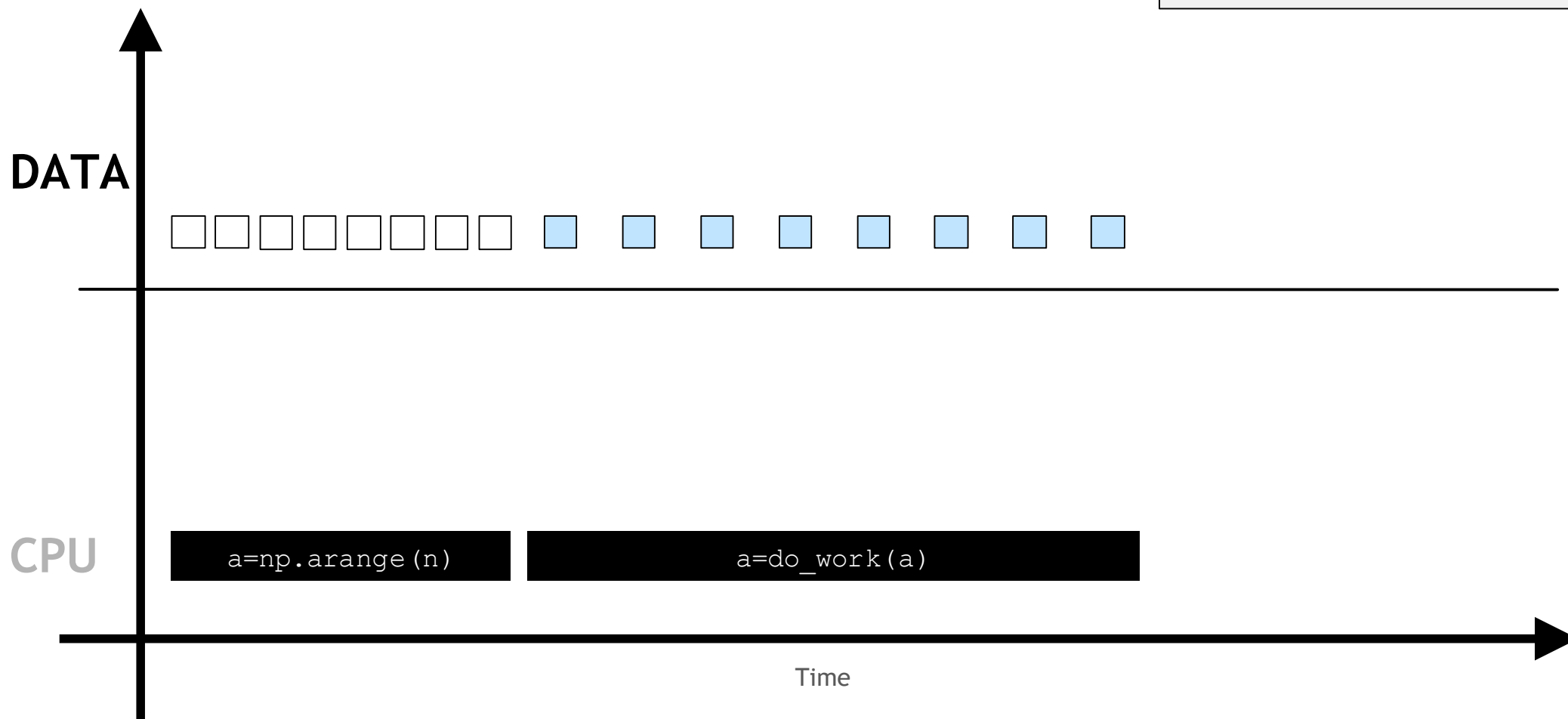


# GPU-accelerated vs. CPU-only Applications

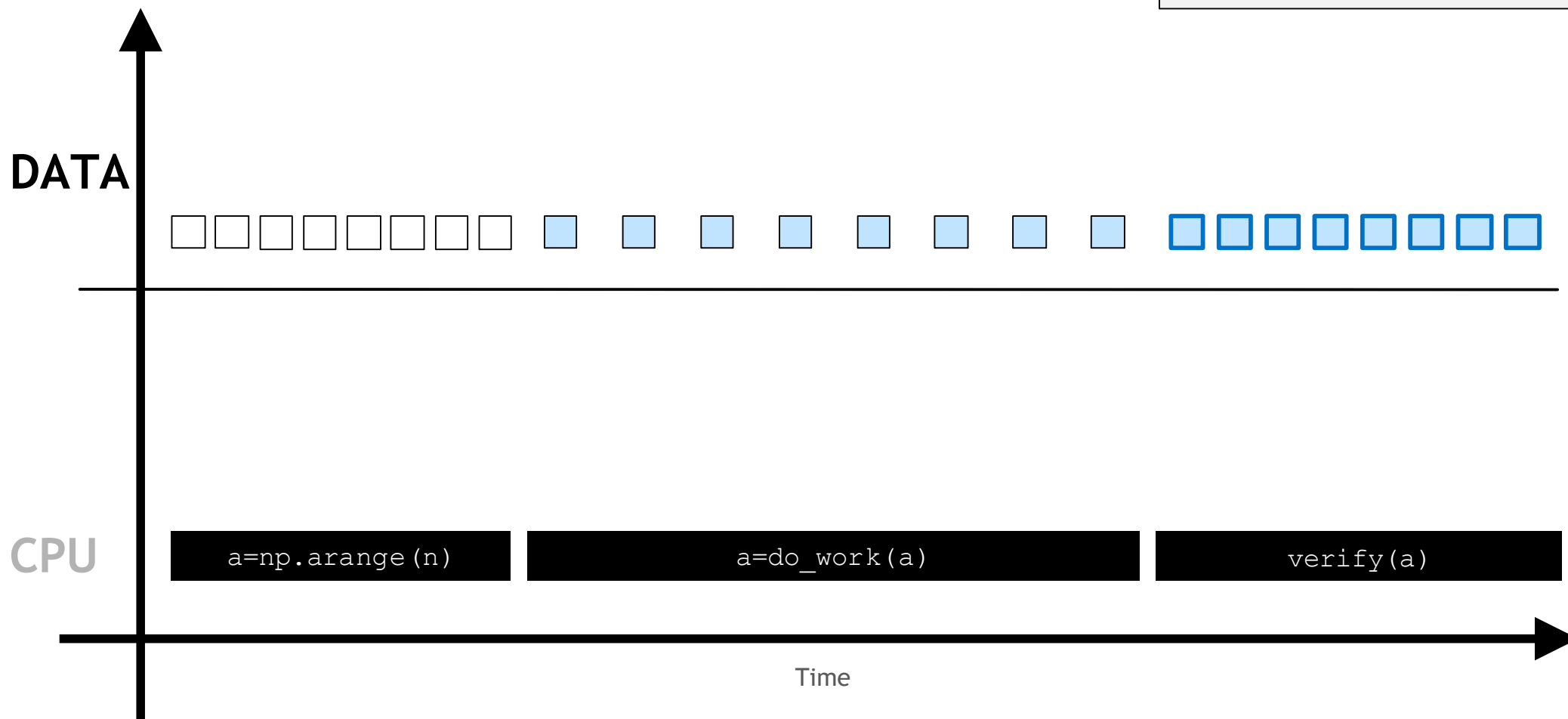
In **CPU-only applications** data is allocated on the CPU



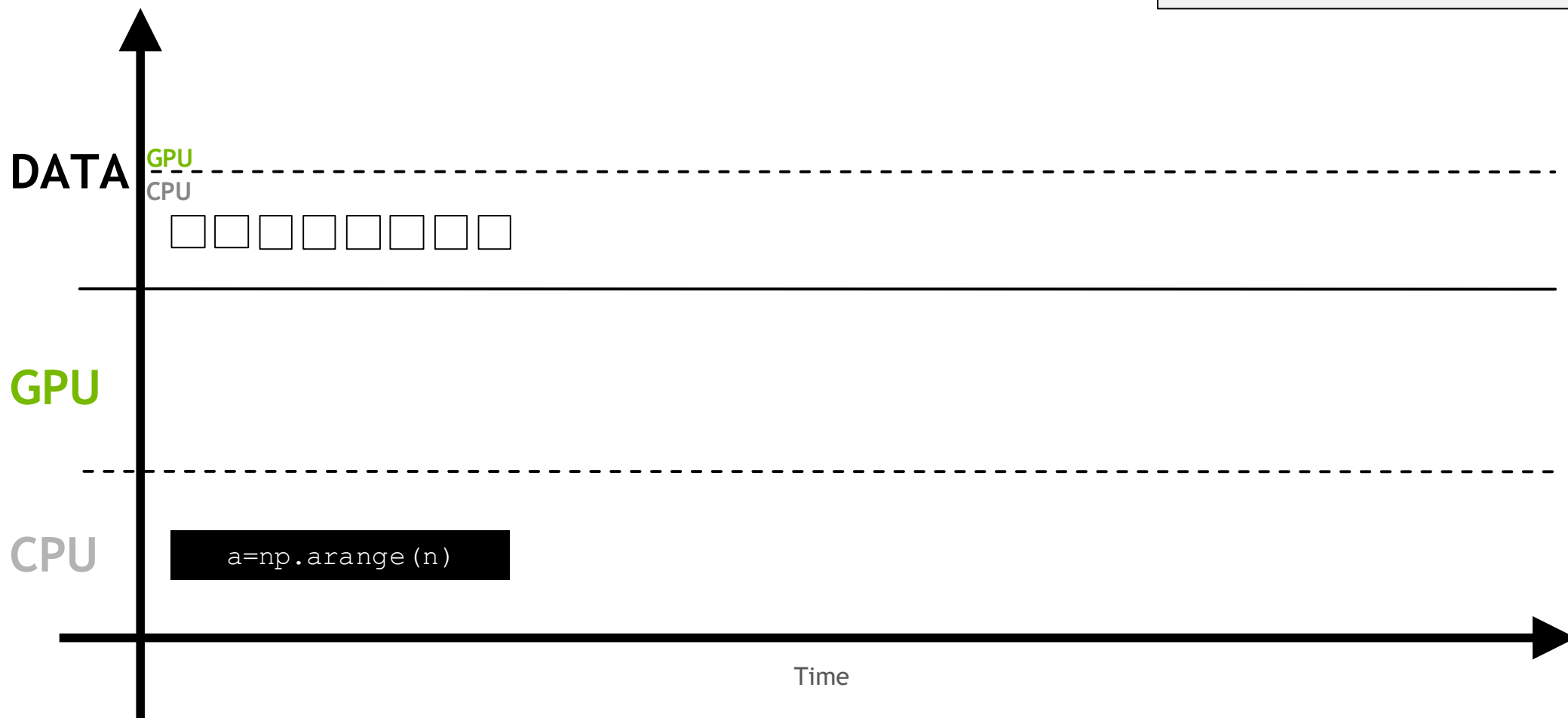
...and all work is performed serially on the CPU



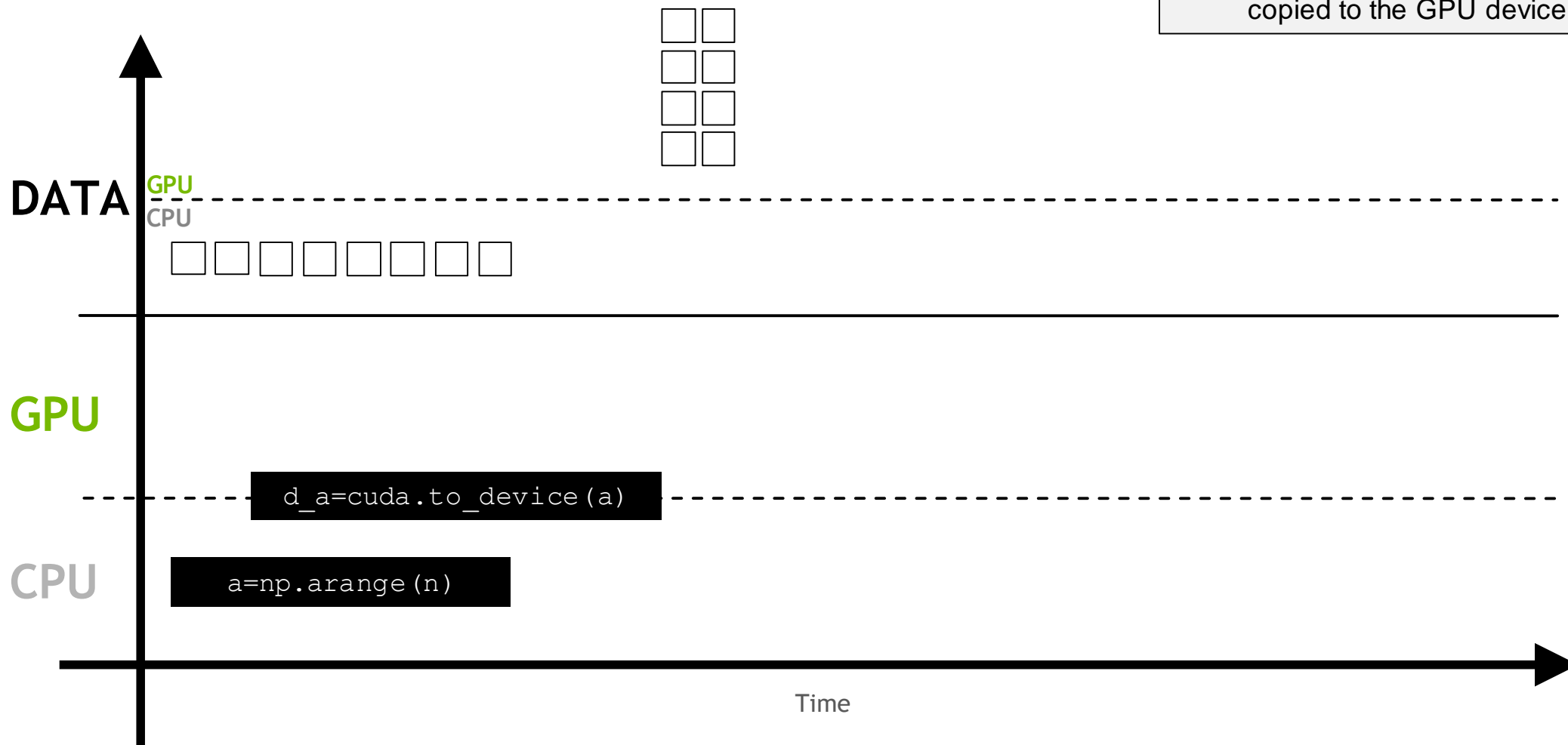
...and all work is performed serially on the CPU



In **accelerated applications** there is both host and device memory.



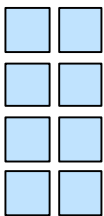
Data initialized on the CPU can be copied to the GPU device...



DATA

GPU

CPU



... where it can be worked on in parallel

GPU

```
d_a=do_work(d_a)
```

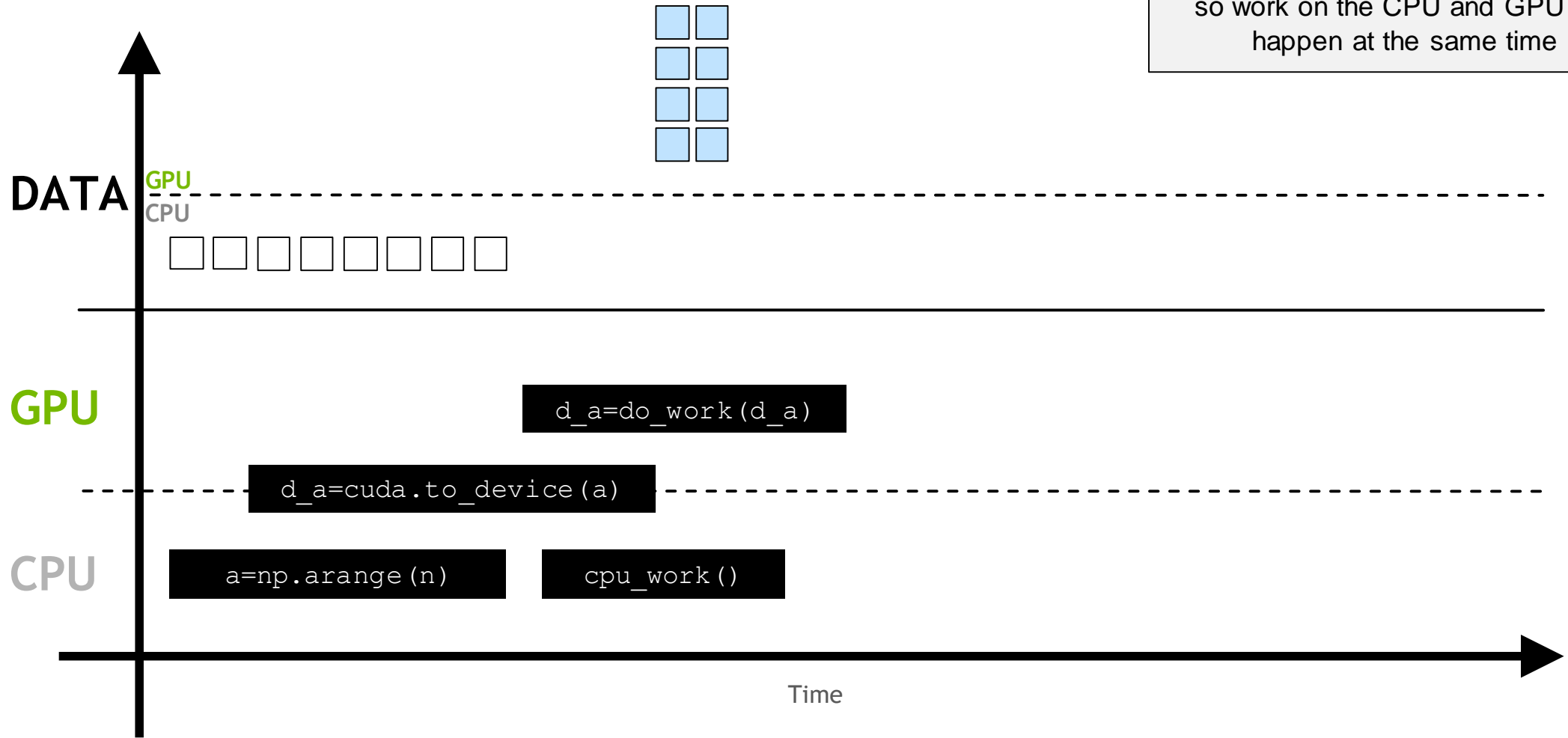
```
d_a=cuda.to_device(a)
```

CPU

```
a=np.arange(n)
```

Time

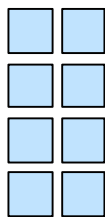
GPU work is asynchronous to the host,  
so work on the CPU and GPU can  
happen at the same time





DATA

GPU  
CPU



GPU

`d_a=cuda.to_device(a)`

`d_a=do_work(d_a)`

CPU

`a=np.arange(n)`

`cpu_work()`

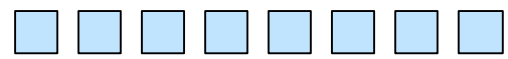
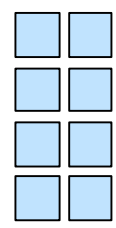
synchronize

Time

Programmers can indicate  
synchronization points between the  
CPU and GPU with  
`cuda.synchronize()`

DATA

GPU  
CPU



GPU

`d_a=do_work(d_a)`

`d_a=cuda.to_device(a)`

`a=d_a.copy_to_host()`

CPU

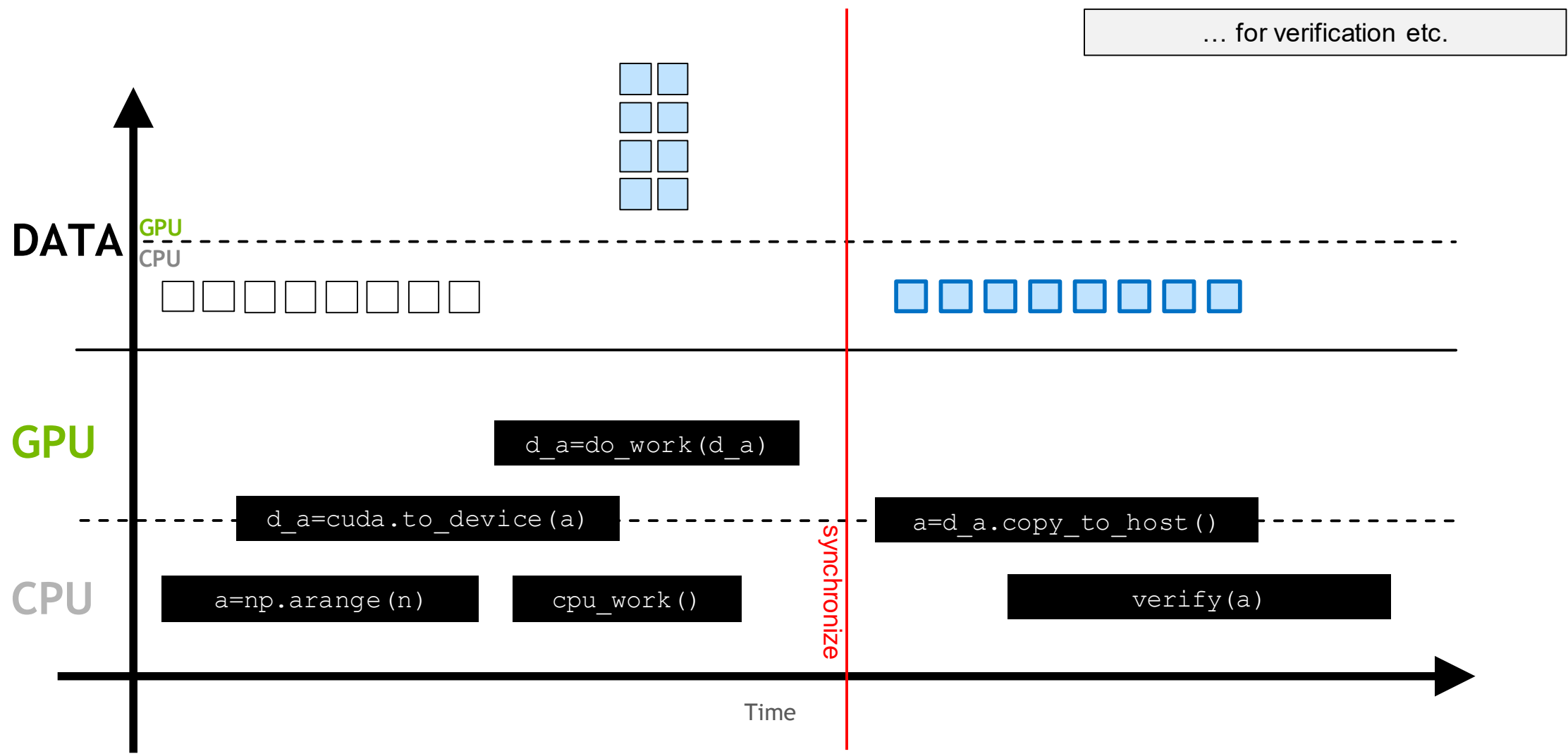
`a=np.arange(n)`

`cpu_work()`

synchronize

Time

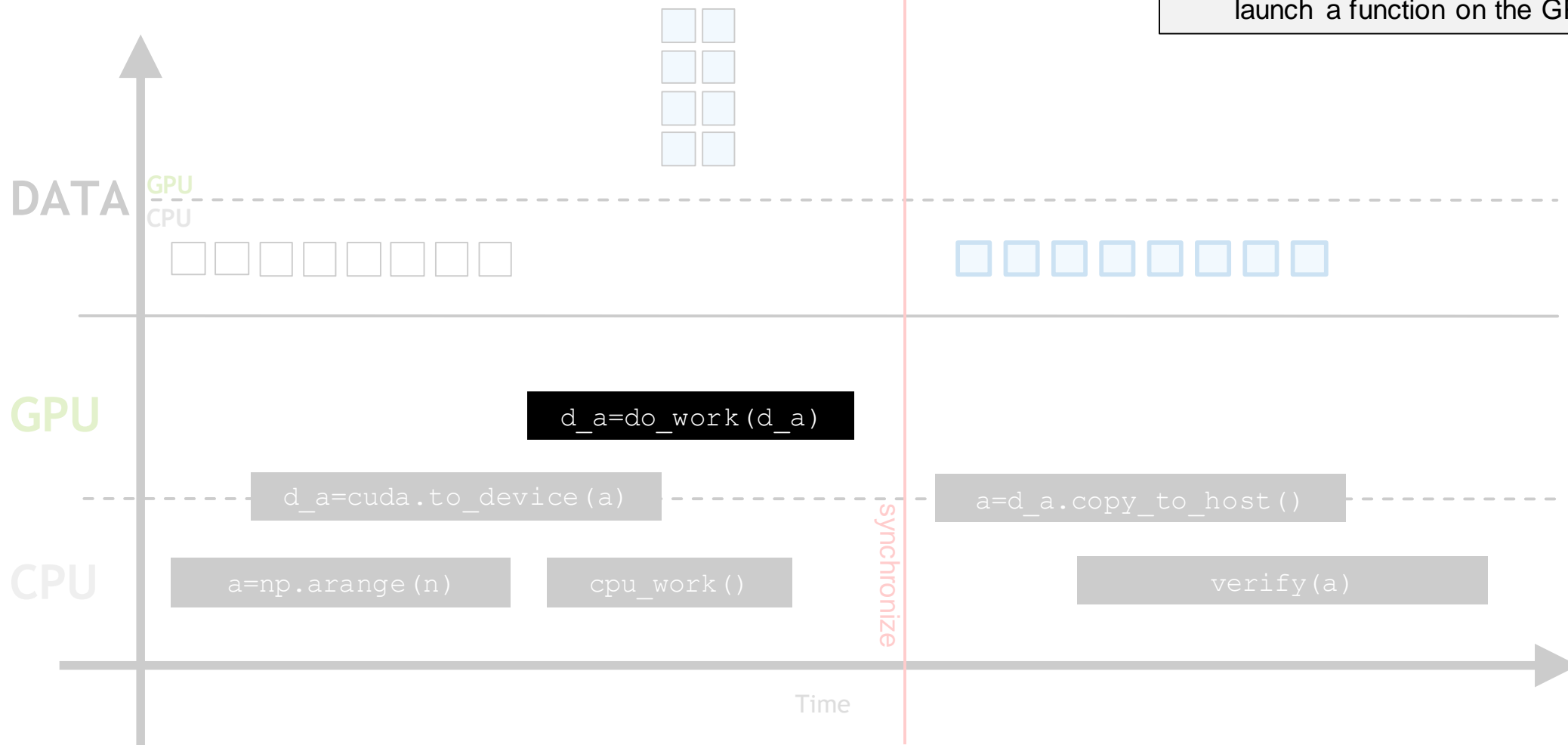
An data can be copied back to the CPU...



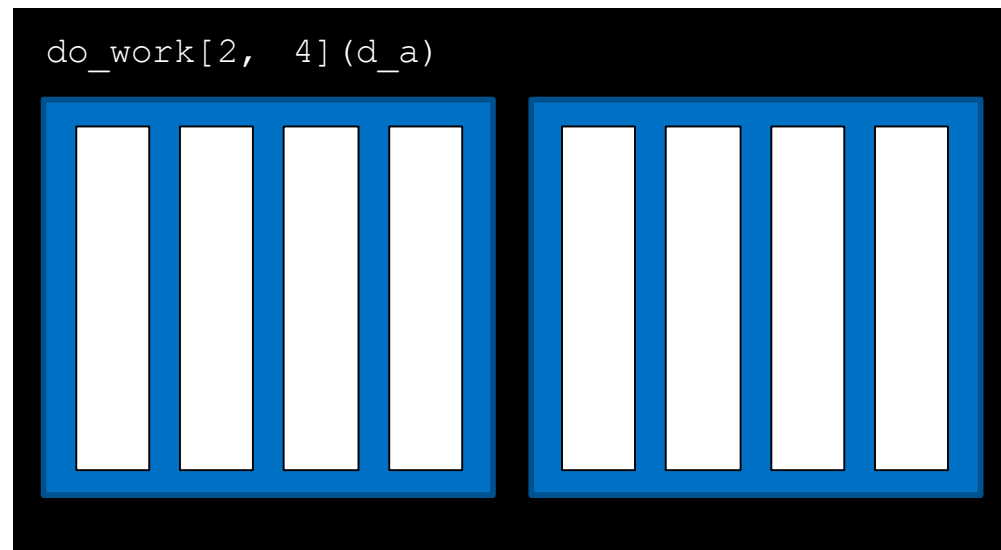
# CUDA Thread Hierarchy



Let's dig into what happens when we launch a function on the GPU

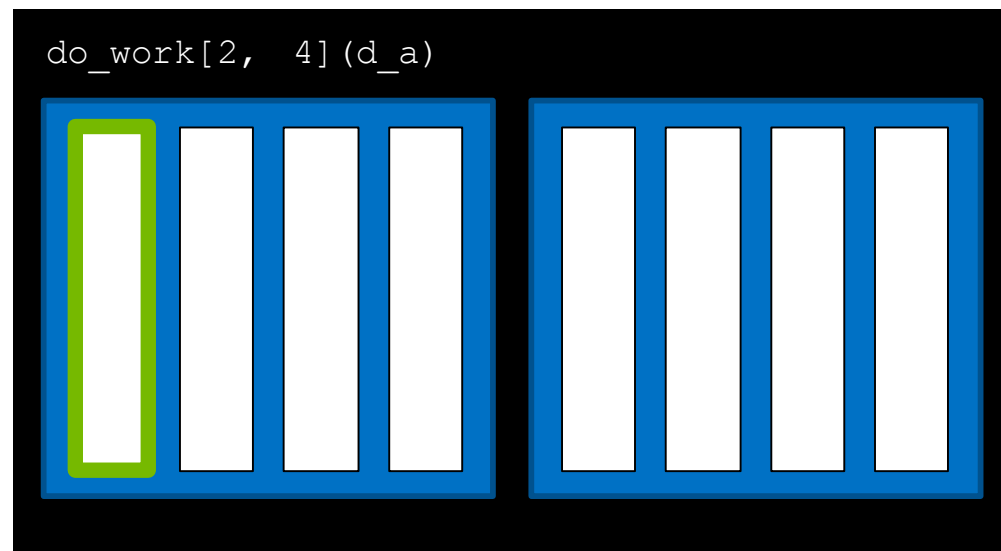


GPU



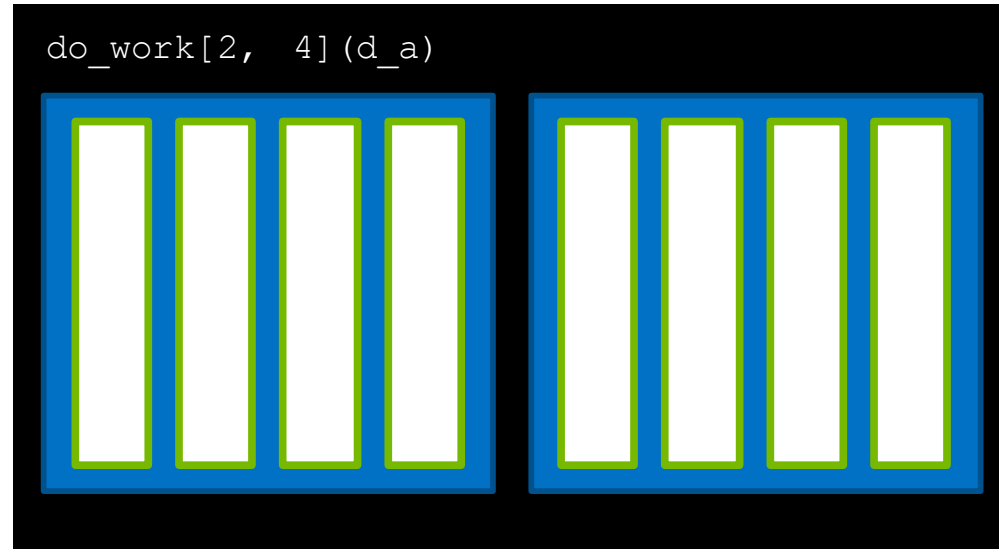
GPU work is done in a **thread**

GPU



Many threads run in parallel

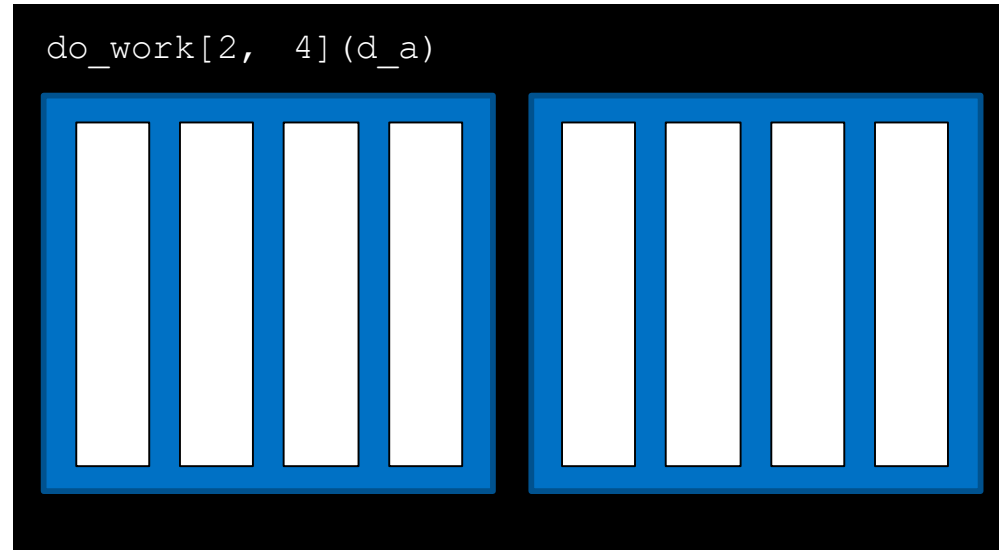
GPU





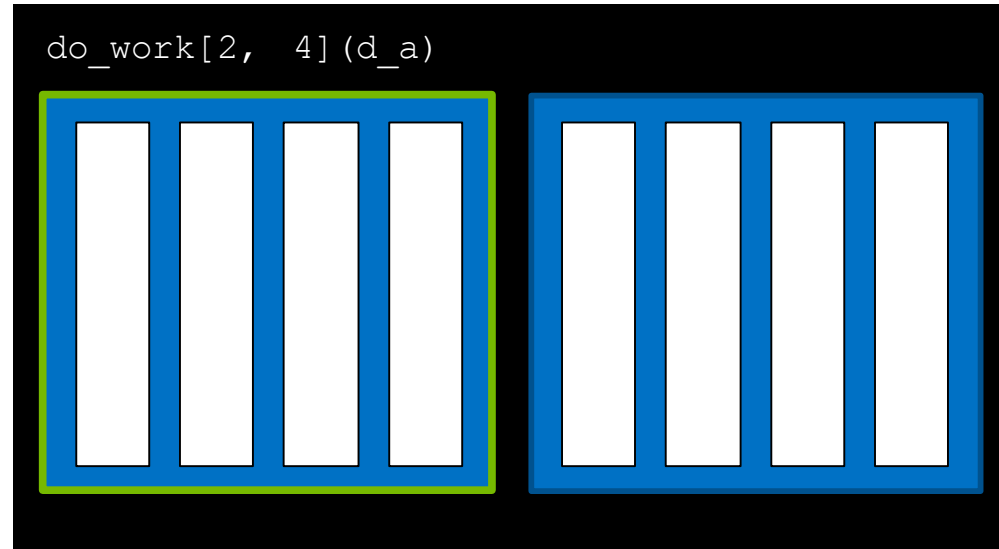
CUDA can process thousands of threads in parallel. The sizes are greatly reduced in these images for simplicity.

GPU



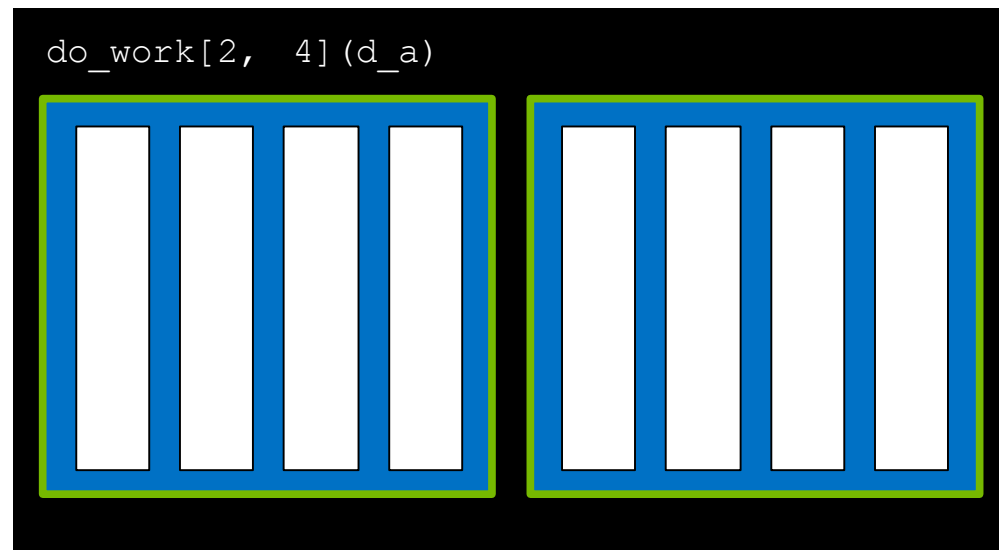
A collection of threads is a **block**

GPU



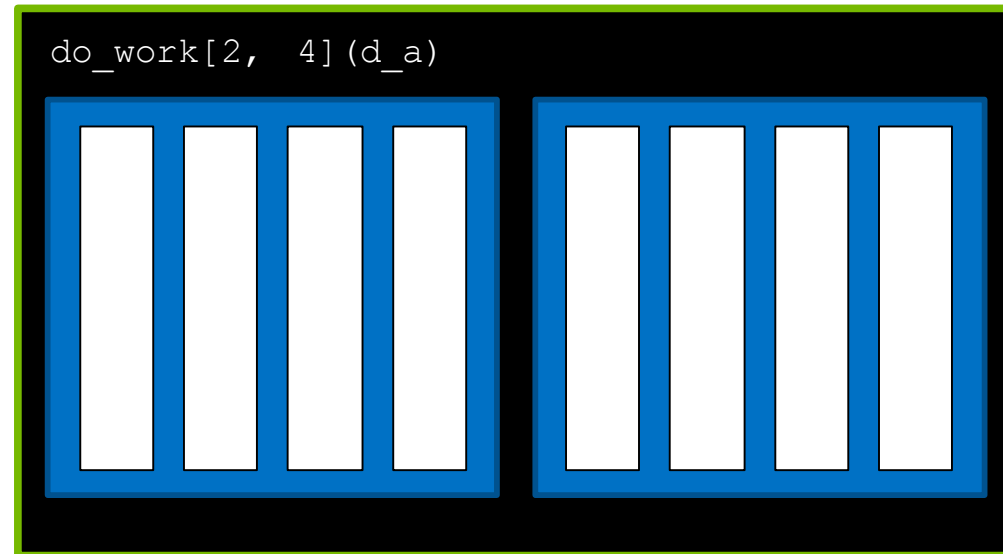
There can be many blocks

GPU

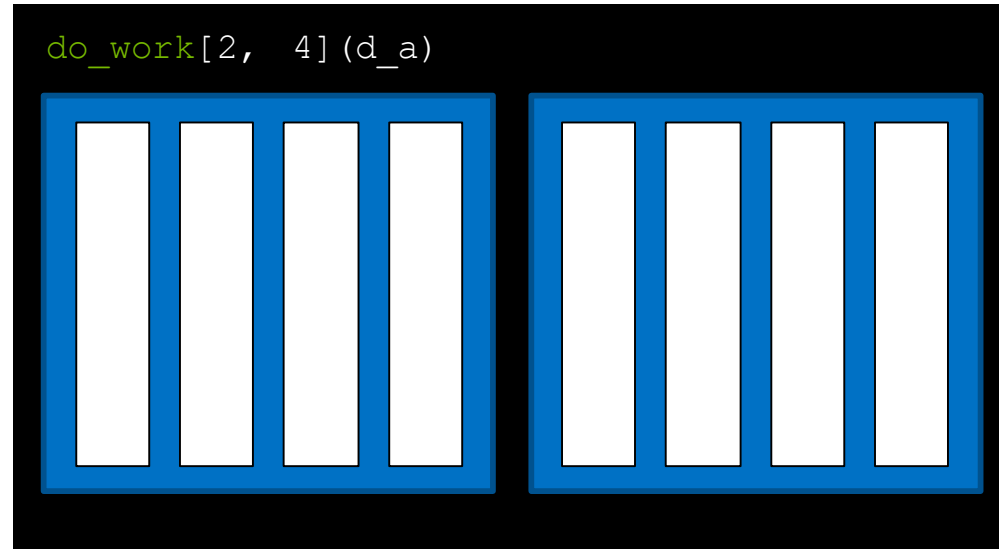


A collection of blocks associated with a given kernel launch is a **grid**

GPU

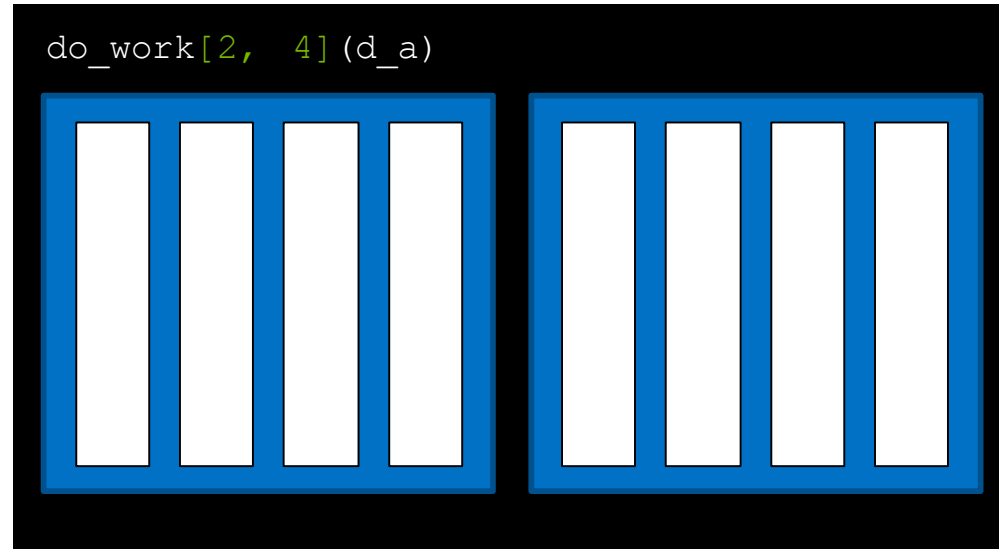


GPU



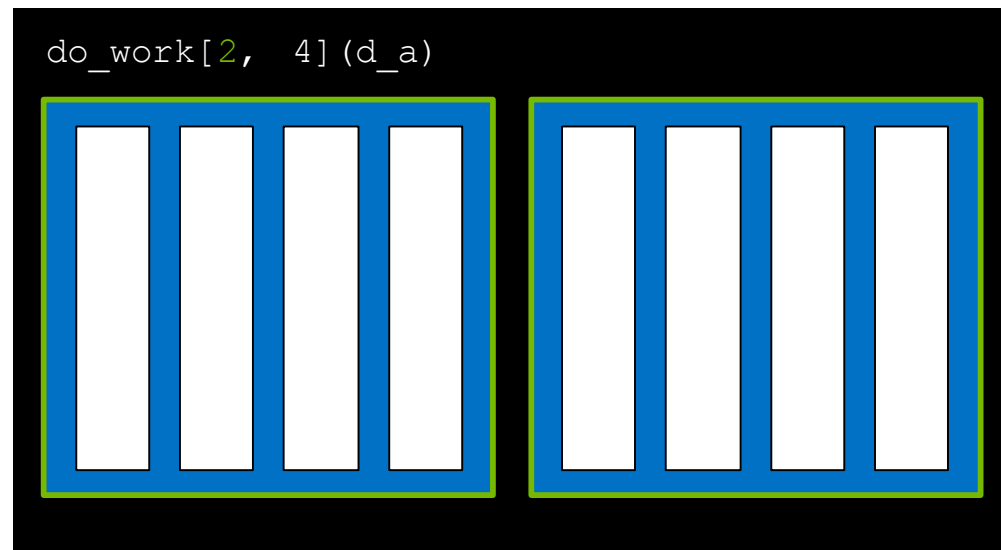
Kernels are **launched** with an  
**execution configuration**

GPU



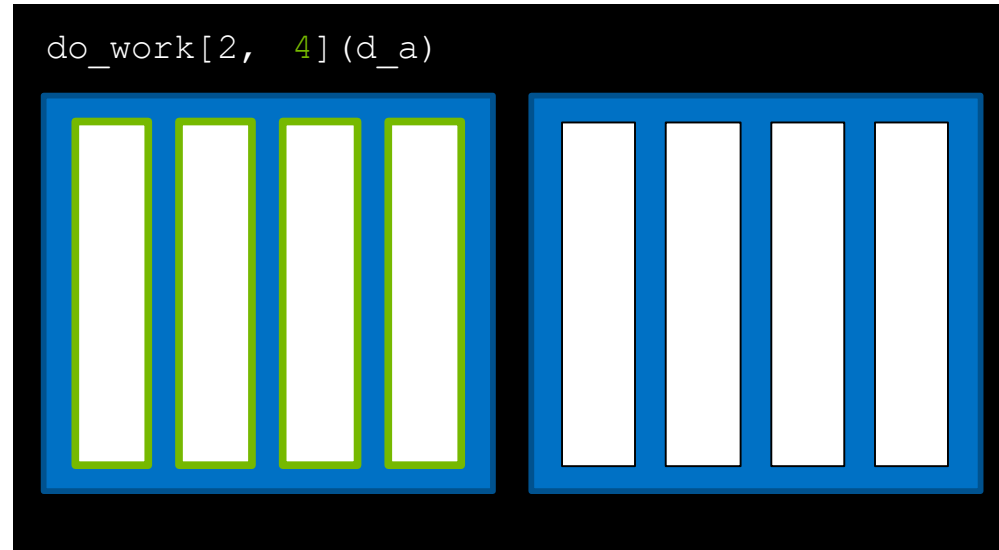
The execution configuration defines  
the number of blocks in the grid

GPU



... as well as the number of threads in each block

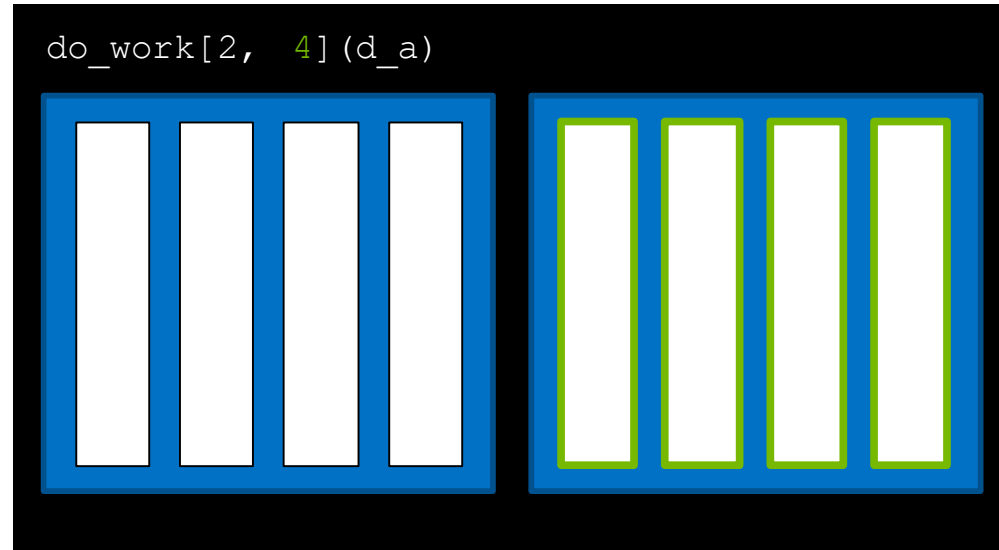
GPU





Every block in the grid contains the same number of threads

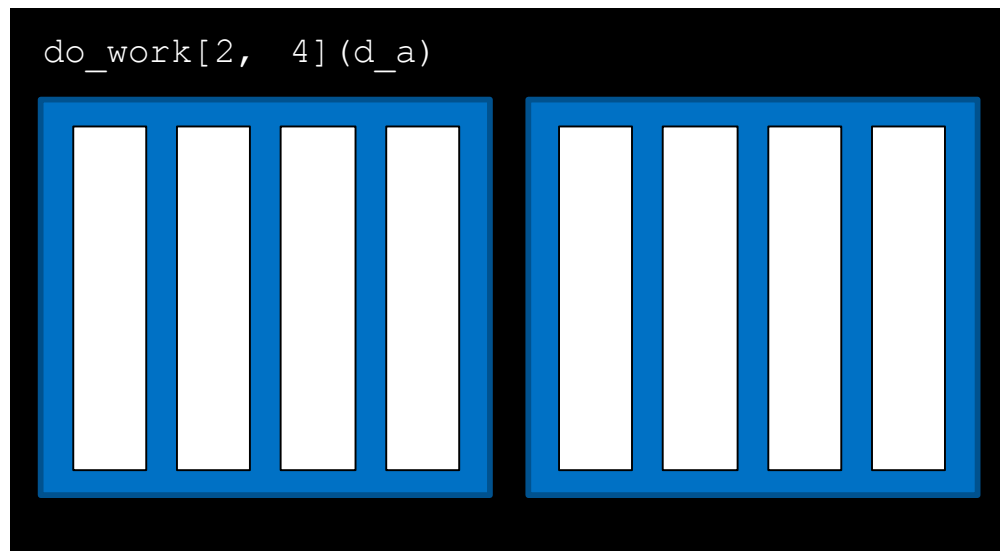
GPU



# CUDA-Provided Thread Hierarchy Variables

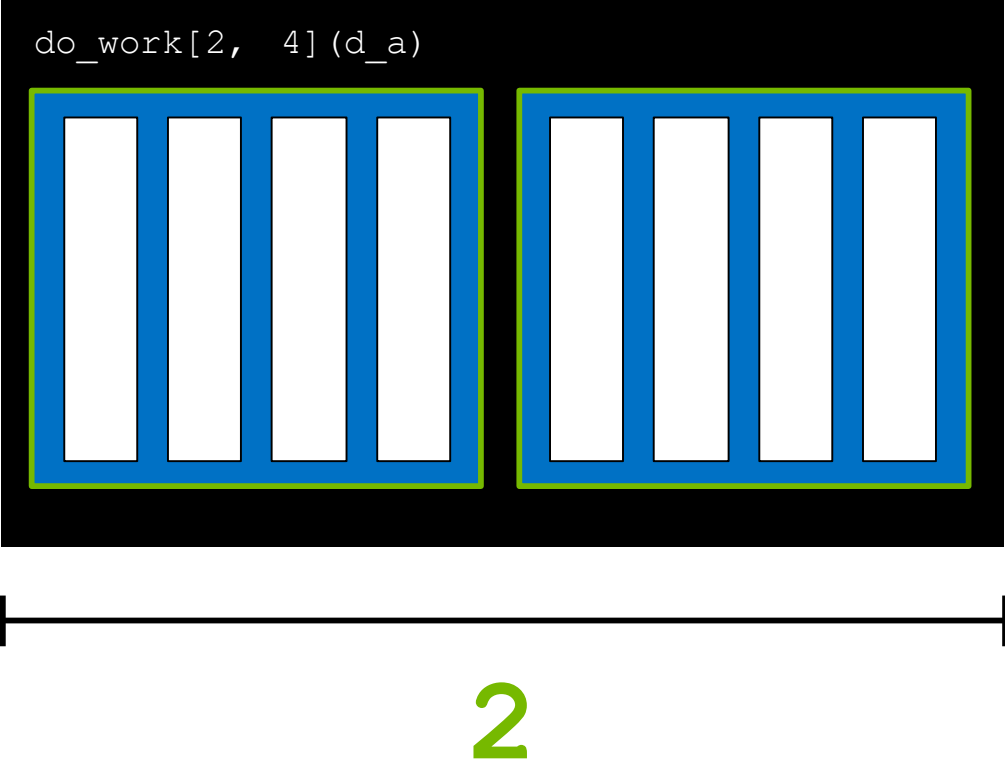
Inside kernel definitions, CUDA-provided variables describe its executing thread, block, and grid

GPU



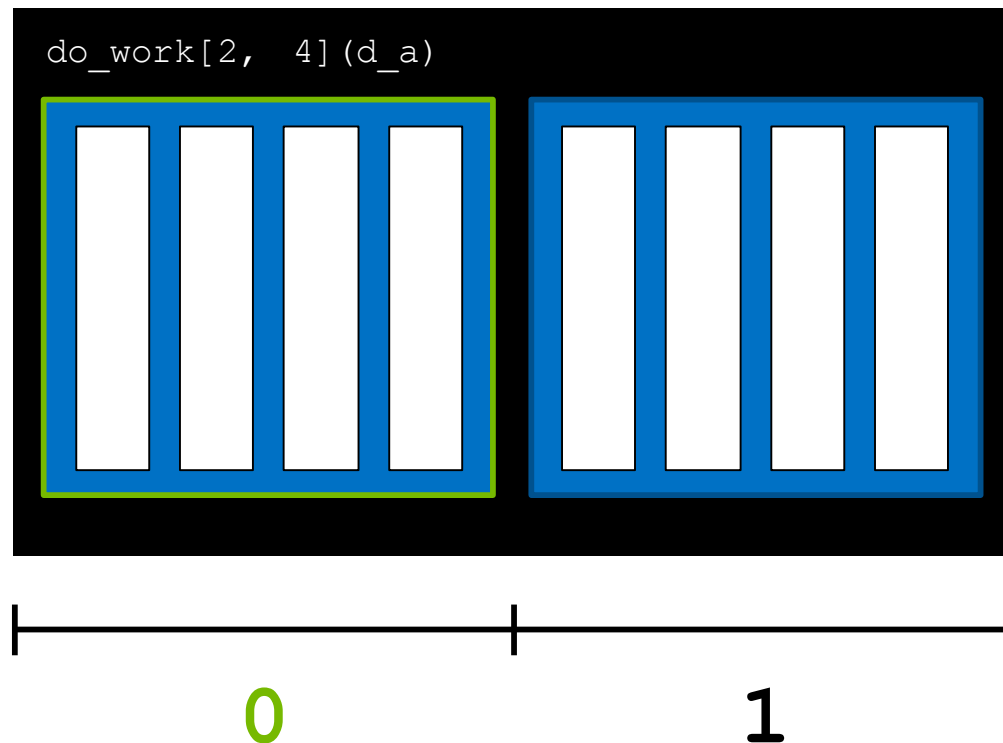
`gridDim.x` is the number of blocks in the grid, in this case 2

GPU



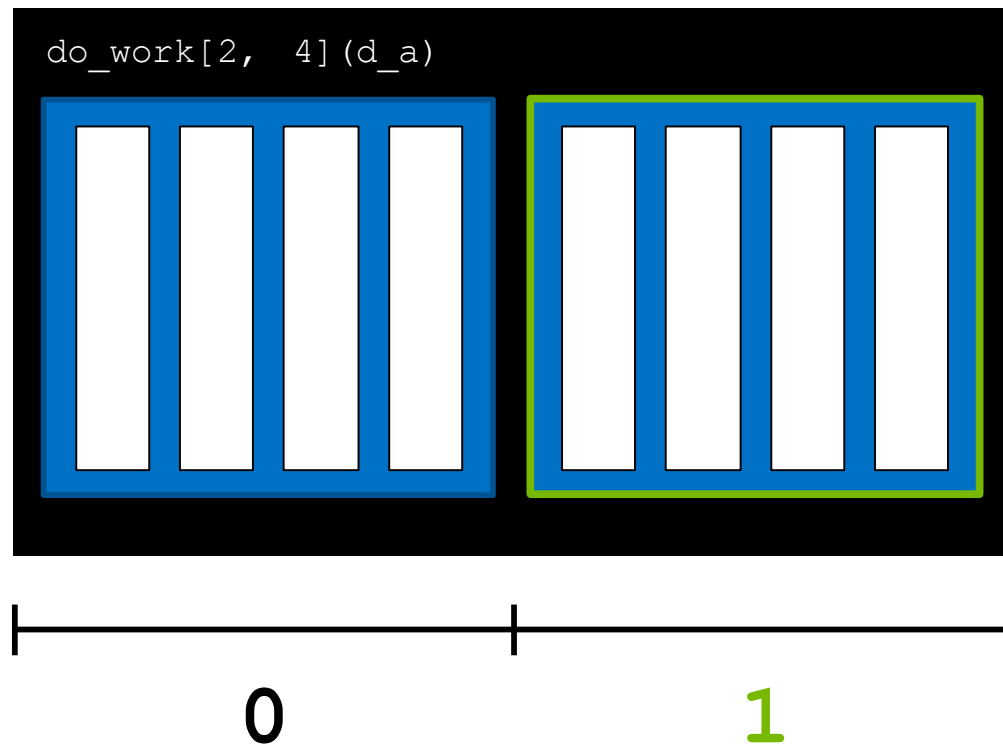
`blockIdx.x` is the index of the current block within the grid, in this case 0

GPU



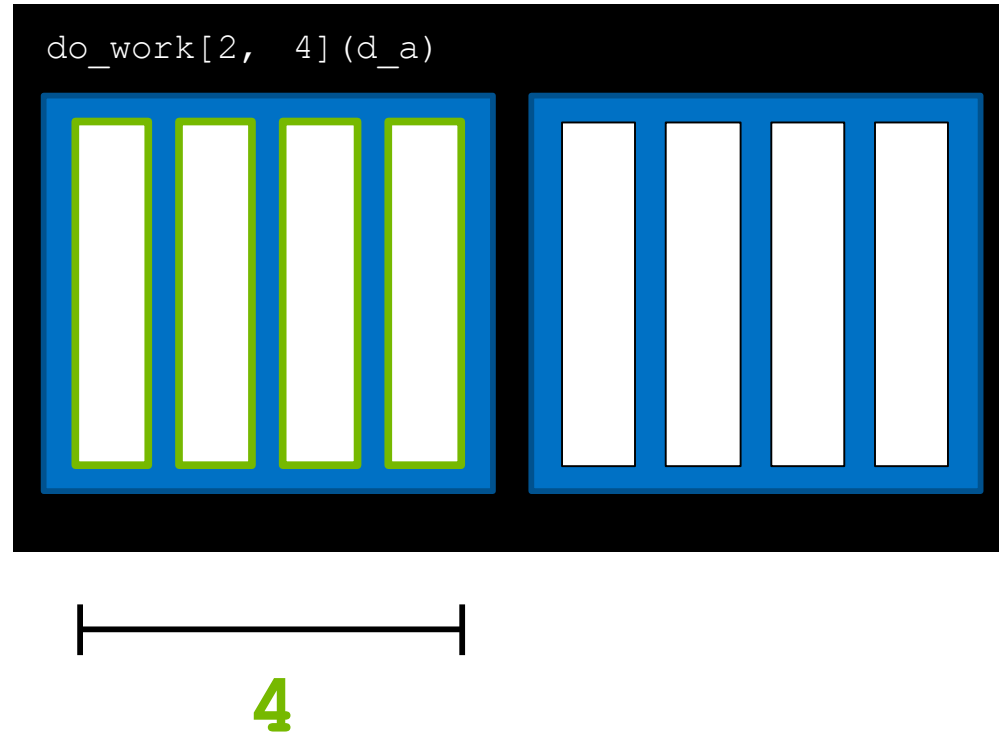
`blockIdx.x` is the index of the current block within the grid, in this case 1

GPU



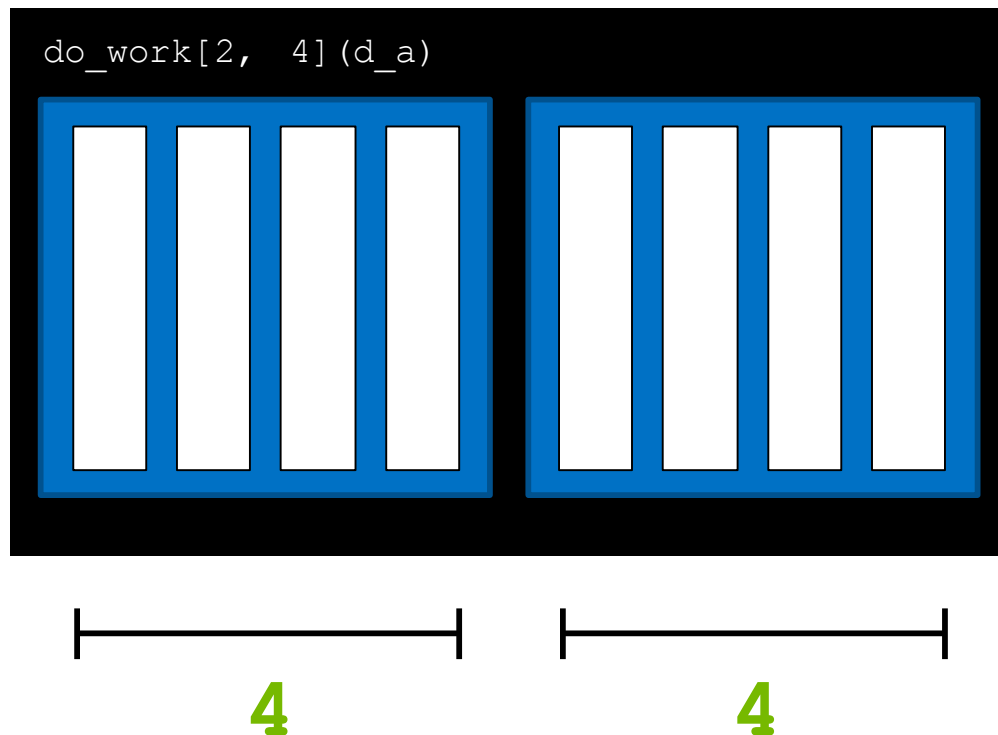
Inside a kernel `blockDim.x` describes the number of threads in a block. In this case 4

GPU



All blocks in a grid contain the same number of threads

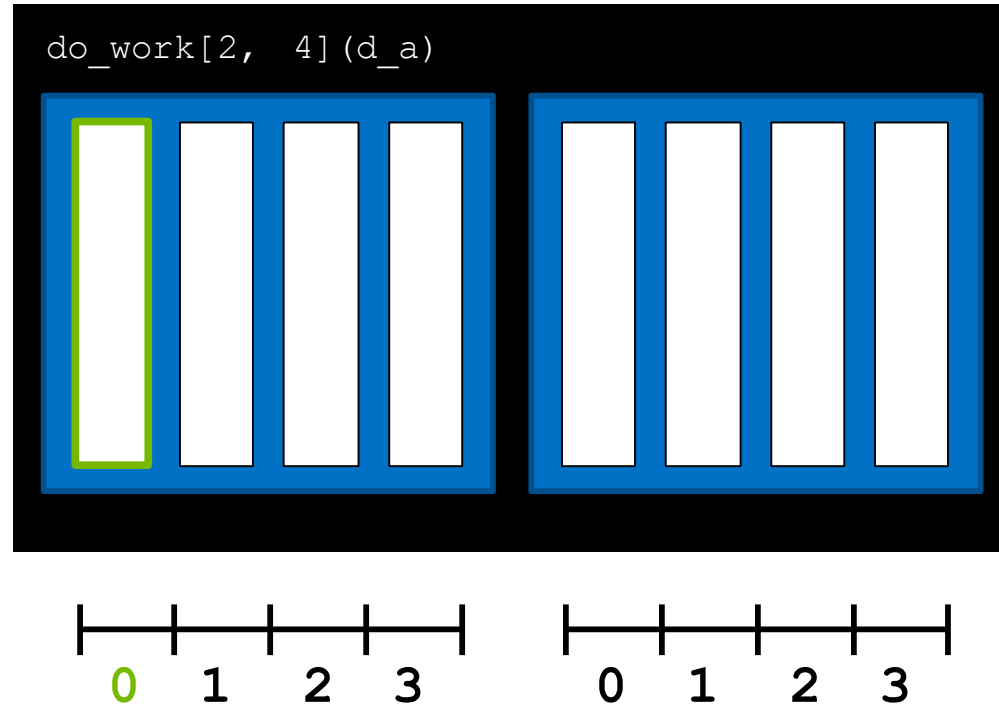
GPU





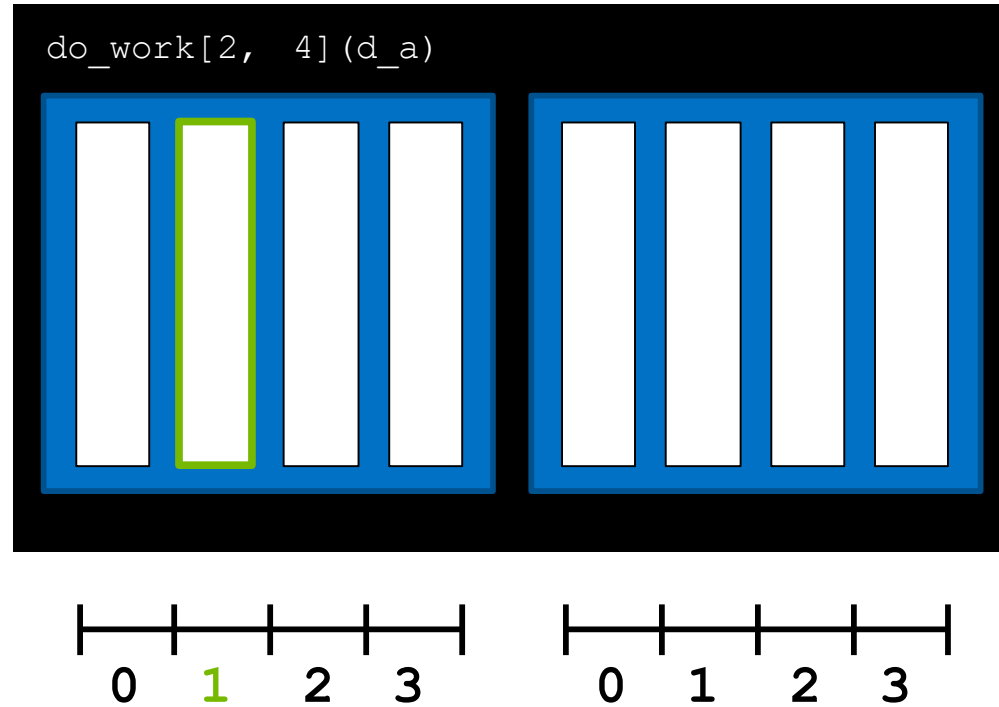
Inside a kernel `threadIdx.x`  
describes the index of the thread within  
a block. In this case 0

GPU



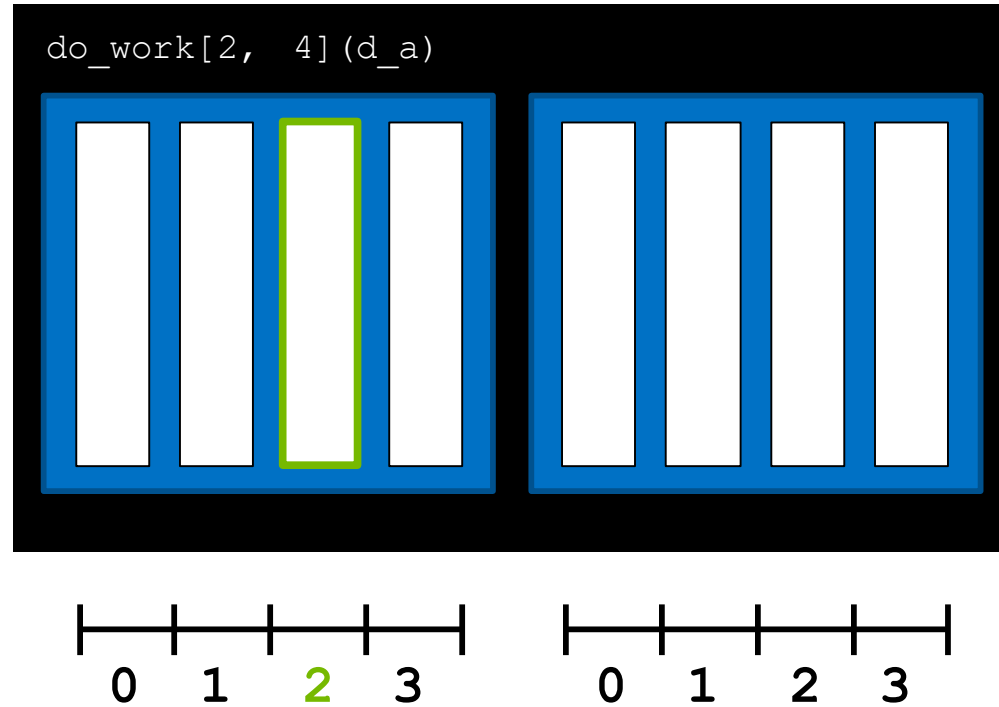
Inside a kernel `threadIdx.x`  
describes the index of the thread within  
a block. In this case 1

GPU



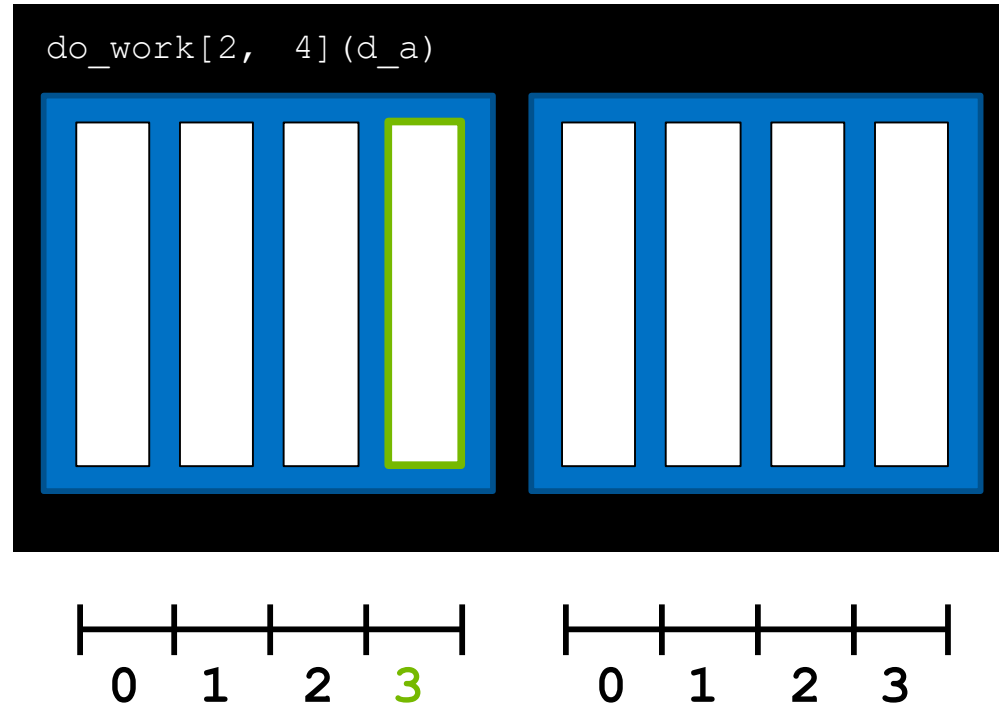
Inside a kernel `threadIdx.x`  
describes the index of the thread within  
a block. In this case 2

GPU



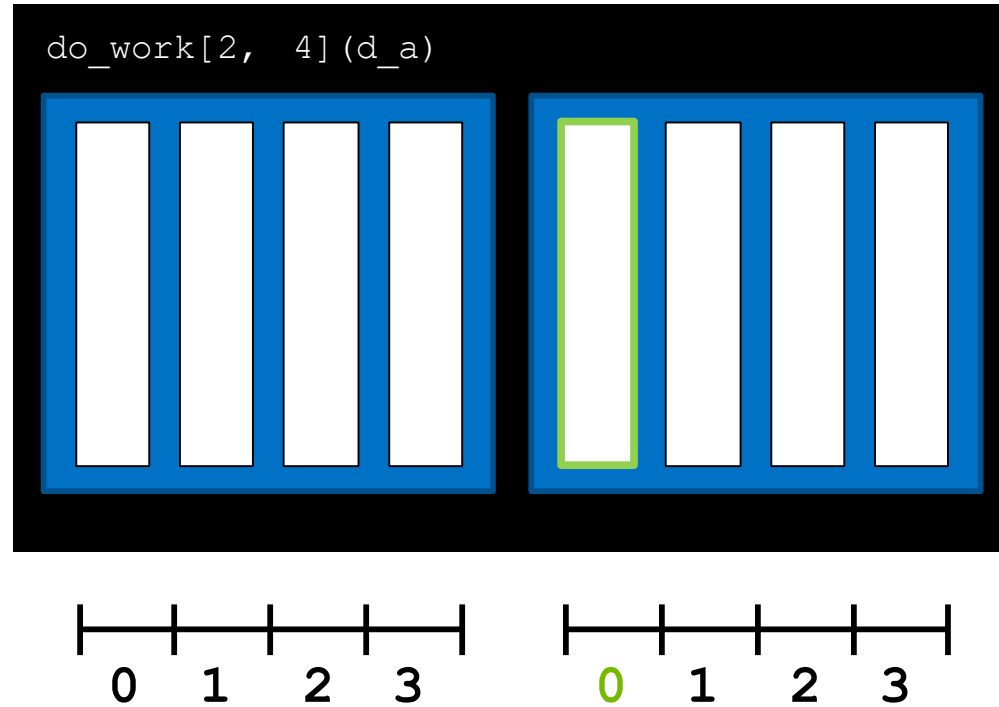
Inside a kernel `threadIdx.x`  
describes the index of the thread within  
a block. In this case 3

GPU



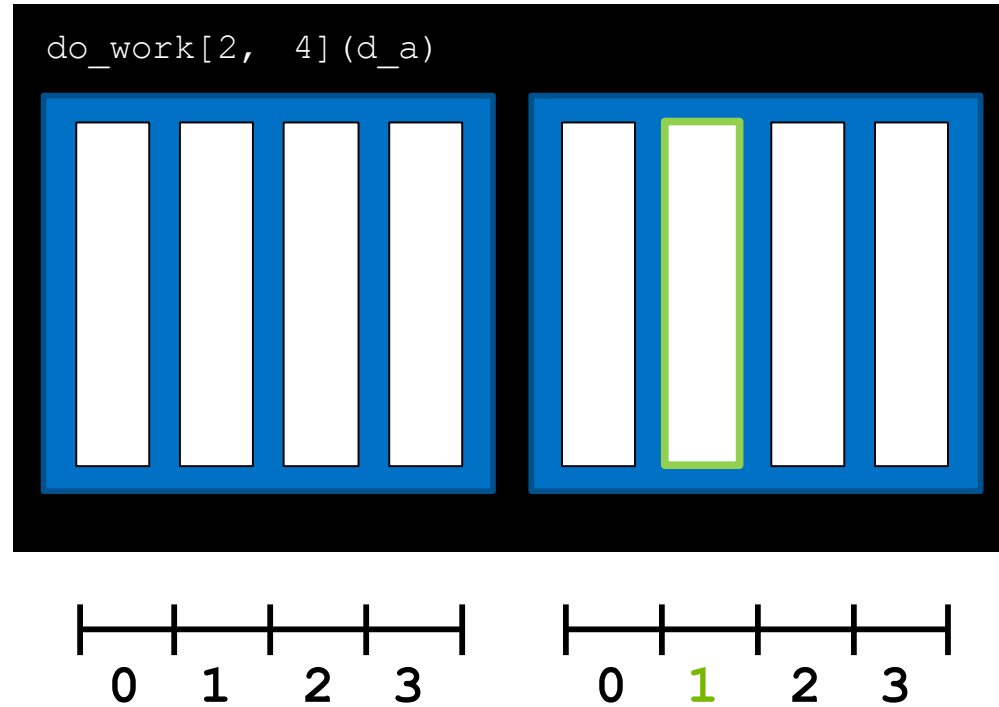
Inside a kernel `threadIdx.x`  
describes the index of the thread within  
a block. In this case 0

GPU



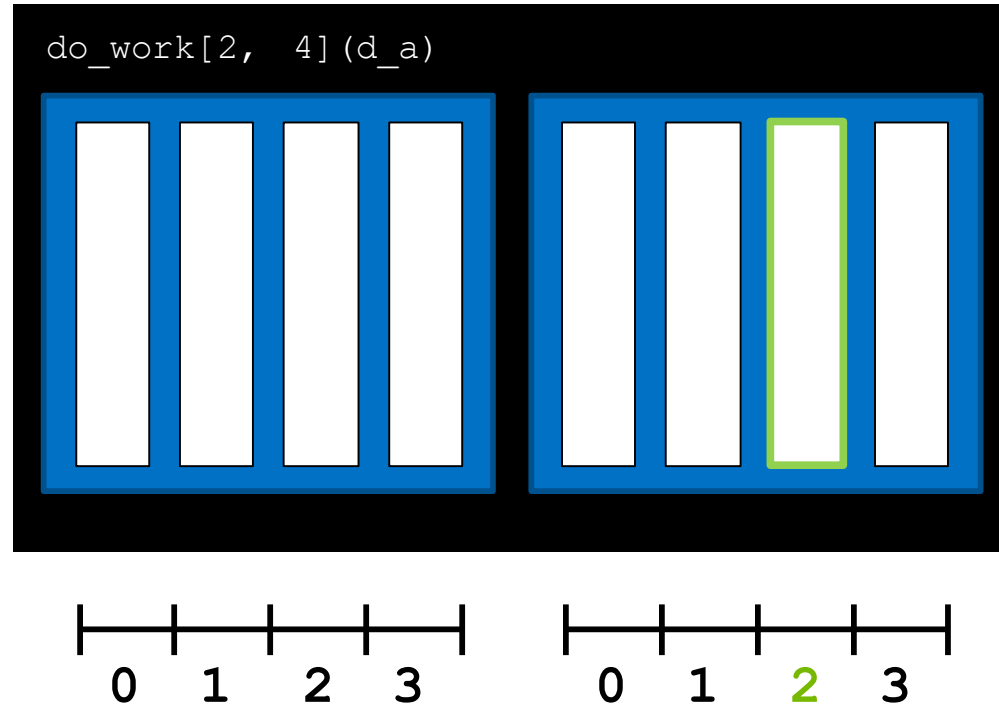
Inside a kernel `threadIdx.x`  
describes the index of the thread within  
a block. In this case 1

GPU



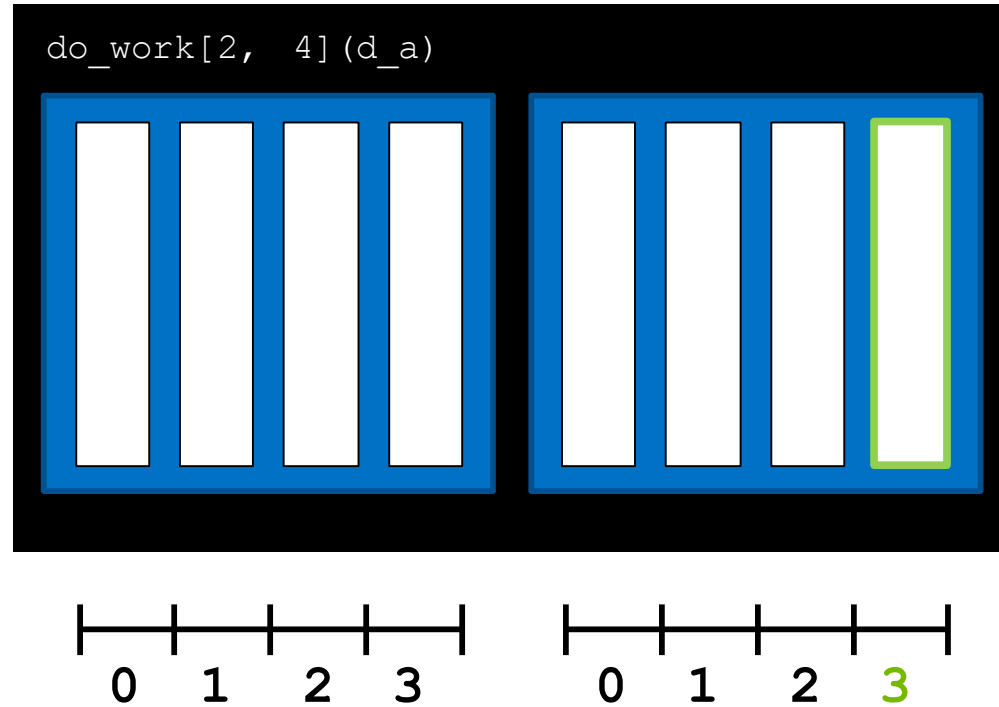
Inside a kernel `threadIdx.x`  
describes the index of the thread within  
a block. In this case 2

GPU



Inside a kernel `threadIdx.x`  
describes the index of the thread within  
a block. In this case 3

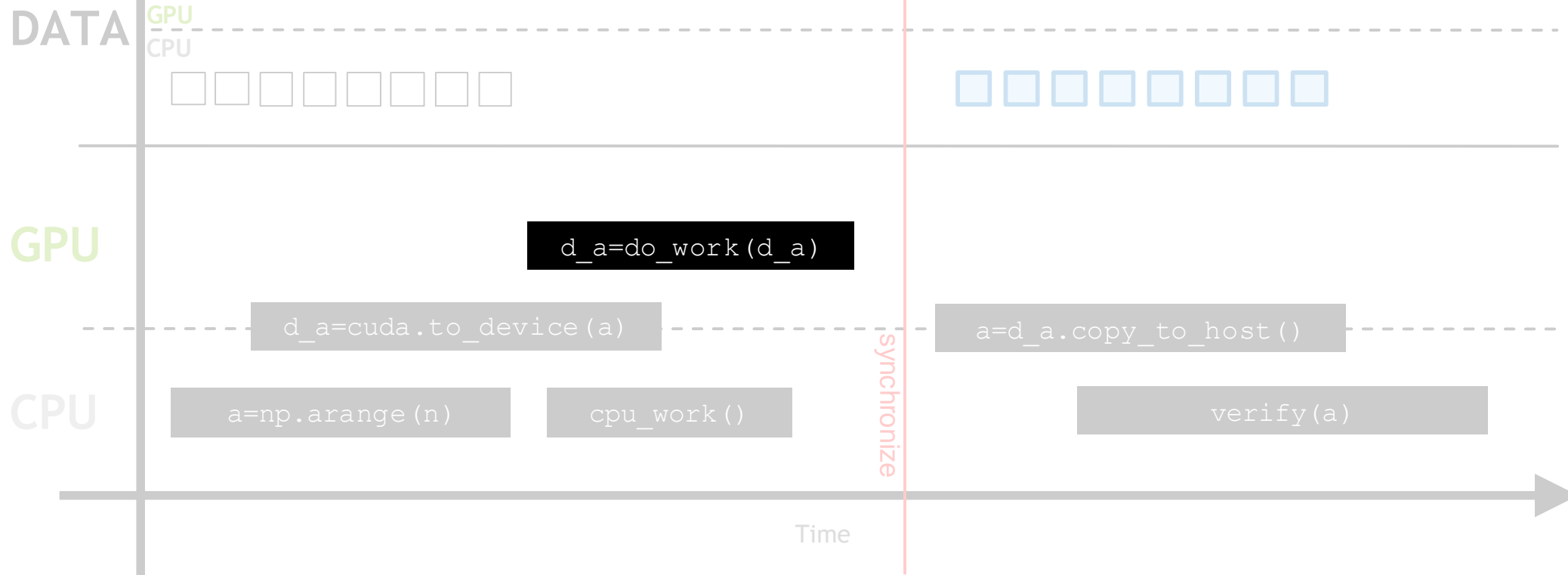
GPU



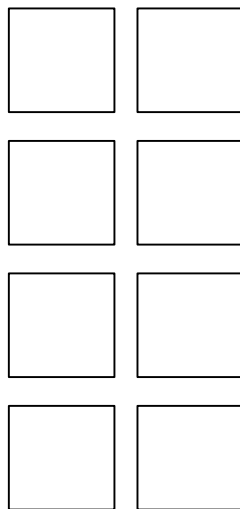


# Coordinating Parallel Threads

Let's look at some basic ways to coordinate parallel threads to work on a data set



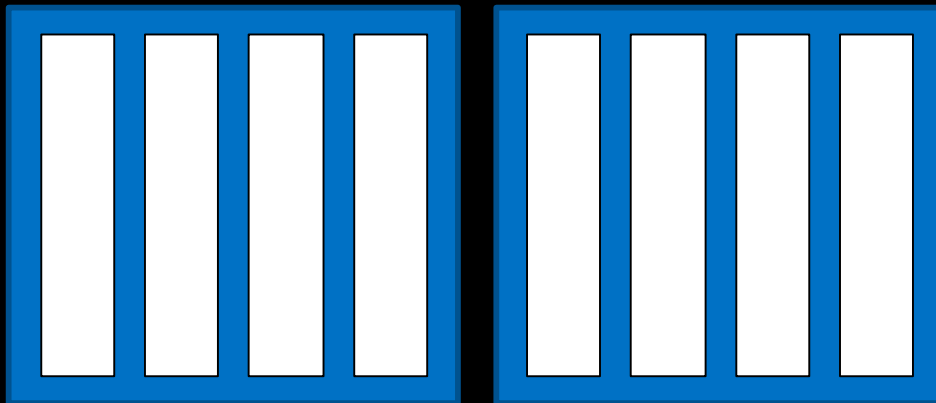
GPU  
DATA



Assume data is in a 0 indexed vector

GPU

```
do_work[2, 4] (d_a)
```



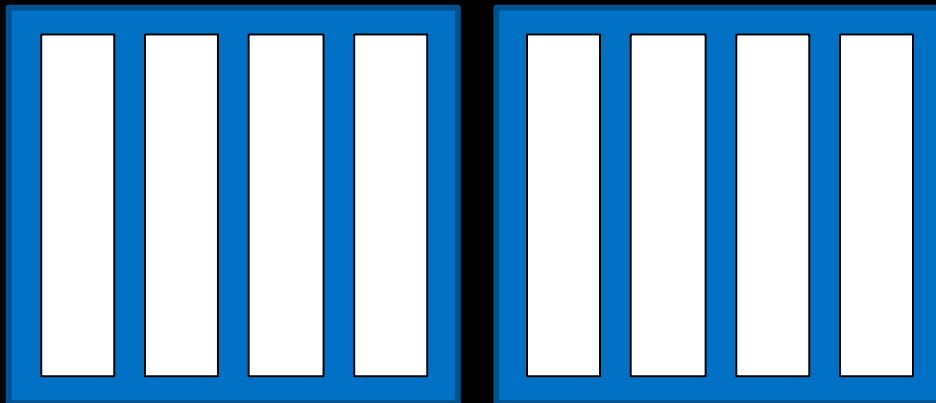
## GPU DATA

0	4
1	5
2	6
3	7

Assume data is in a 0 indexed vector

## GPU

```
do_work[2, 4] (d_a)
```

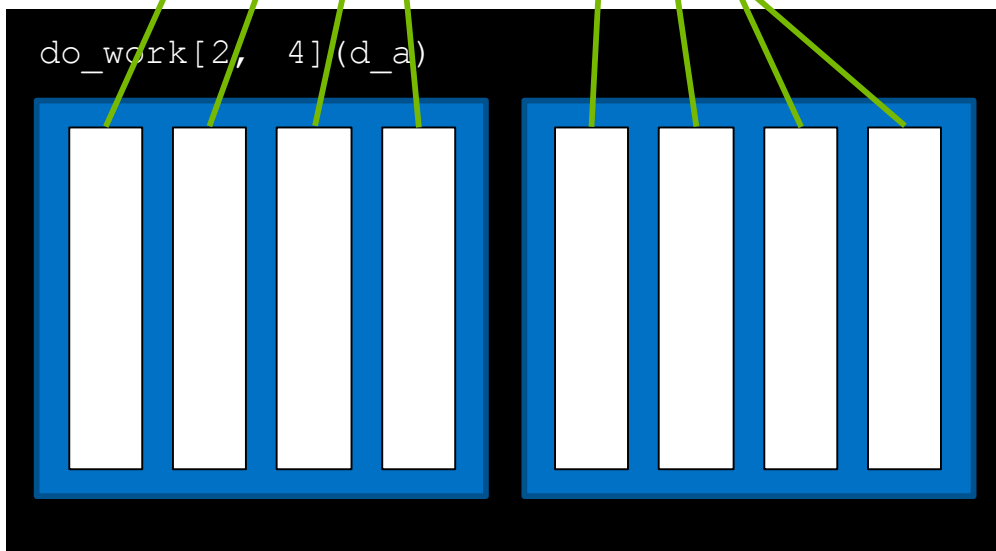


GPU  
DATA

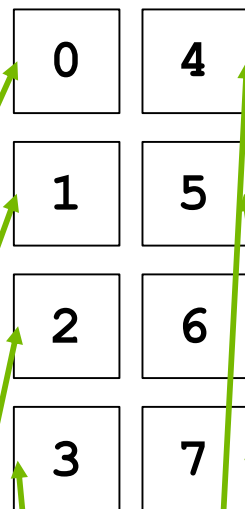
0	4
1	5
2	6
3	7

Somehow, each thread must be mapped to work on elements in the data

GPU

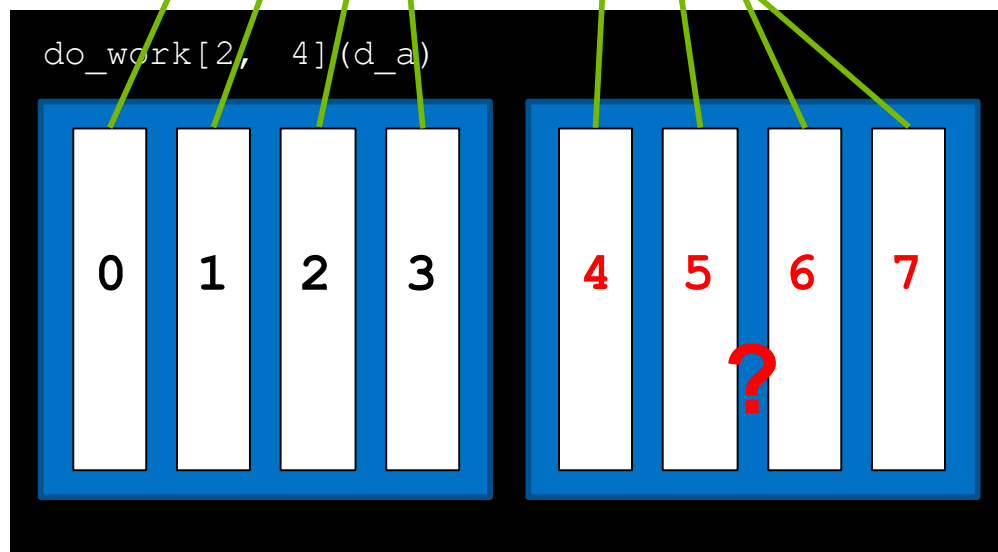


GPU  
DATA

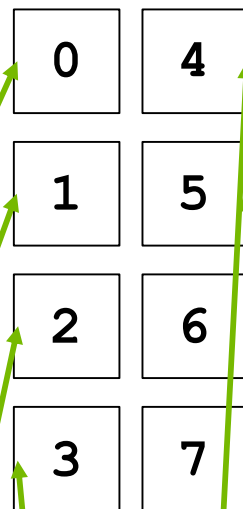


... if we can calculate a thread's index  
*within the entire grid*, then we could  
map that index to an index in the data

GPU

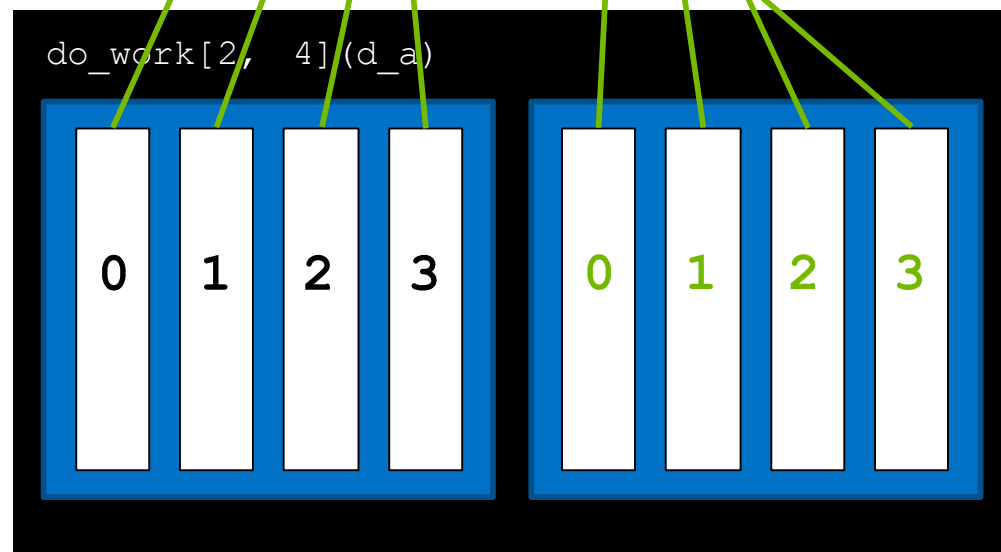


GPU  
DATA



... unfortunately CUDA does not provide a single variable to capture this, only thread indices *within the block*

GPU

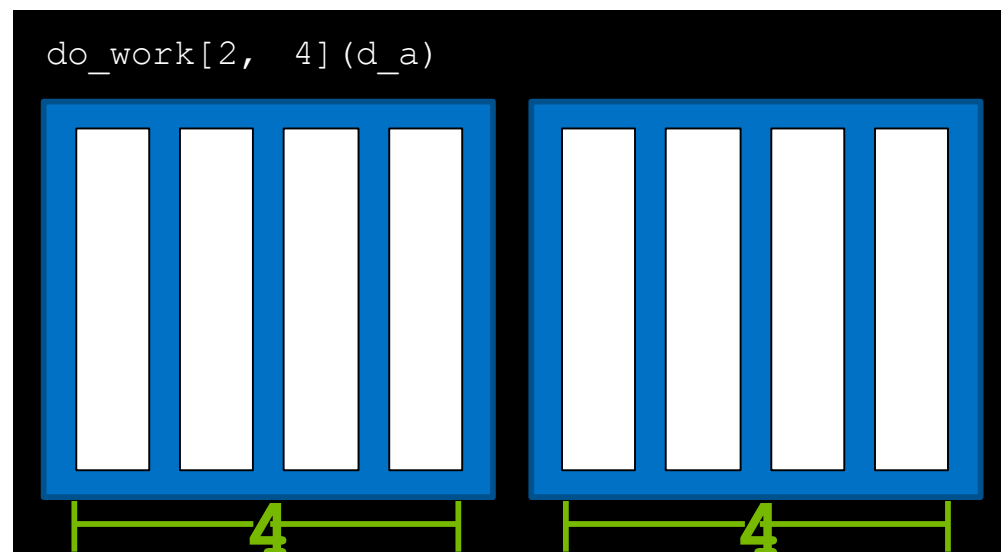


## GPU DATA

0	4
1	5
2	6
3	7

There is an idiomatic way to calculate this value, however. Recall that each thread has access to the size of its block via `blockDim.x`

## GPU



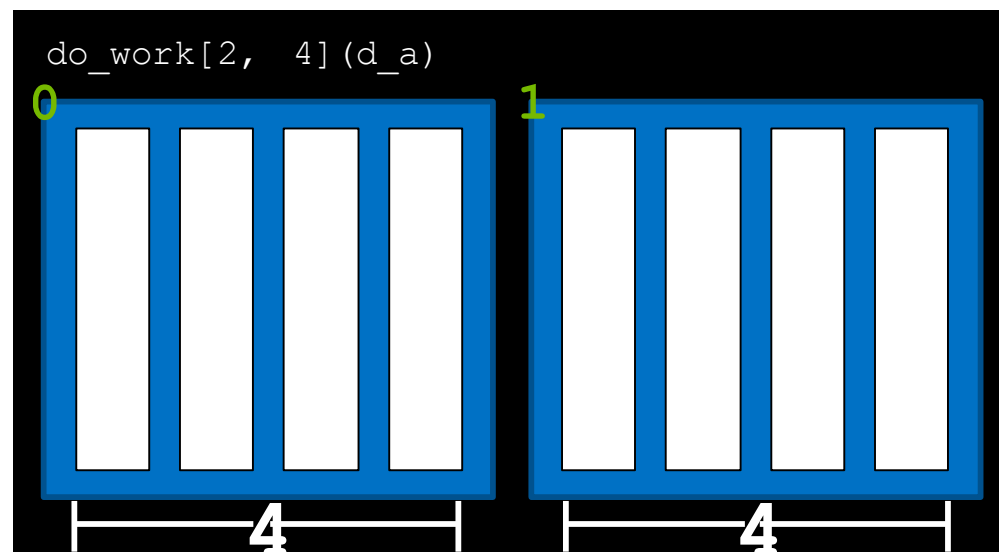


## GPU DATA

0	4
1	5
2	6
3	7

...and the index of its block within the grid via `blockIdx.x`

## GPU

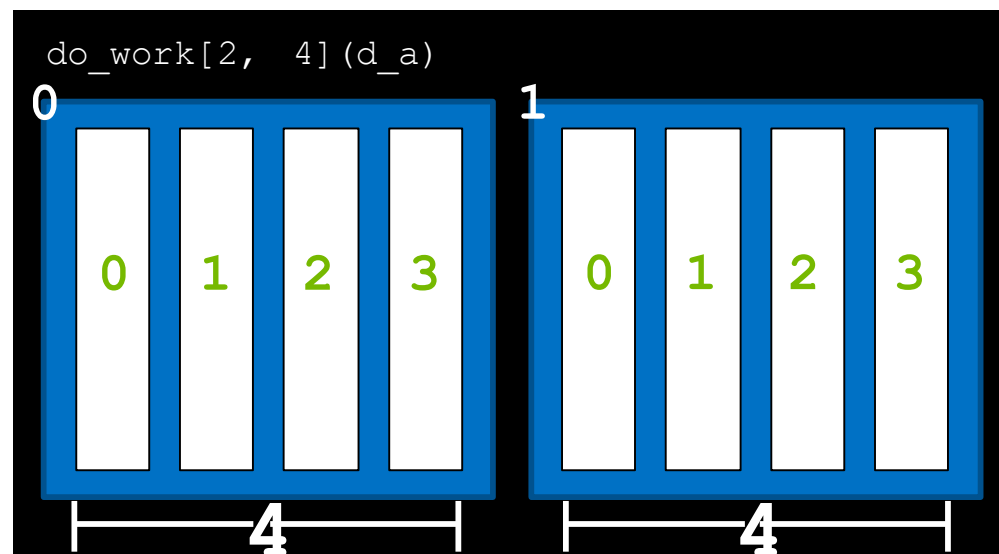


## GPU DATA

0	4
1	5
2	6
3	7

...and its own index within its block via  
`threadIdx.x`

## GPU

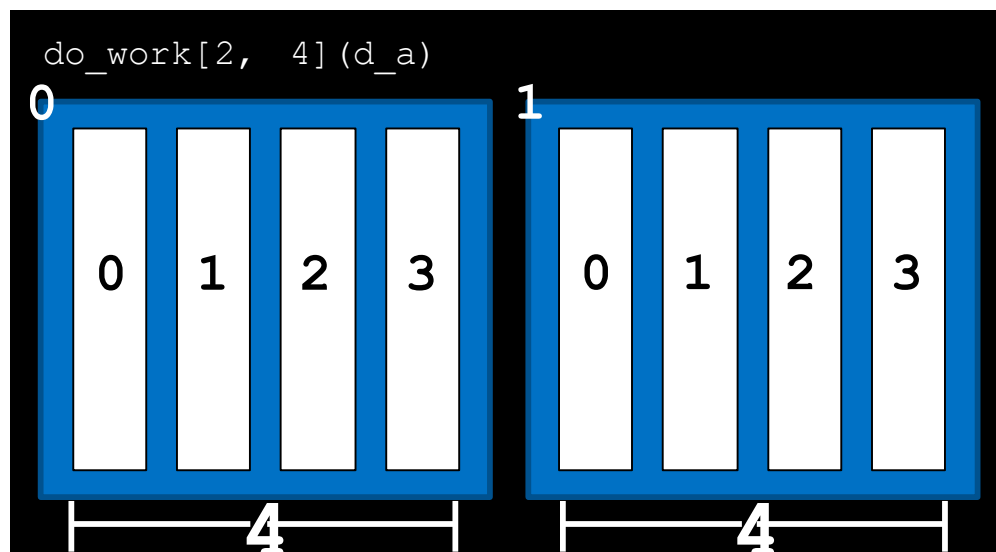


## GPU DATA

0	4
1	5
2	6
3	7

Using these variables, the formula `threadIdx.x + blockIdx.x * blockDim.x` will return the thread's unique index in the whole grid, which we can then map to data elements.

## GPU



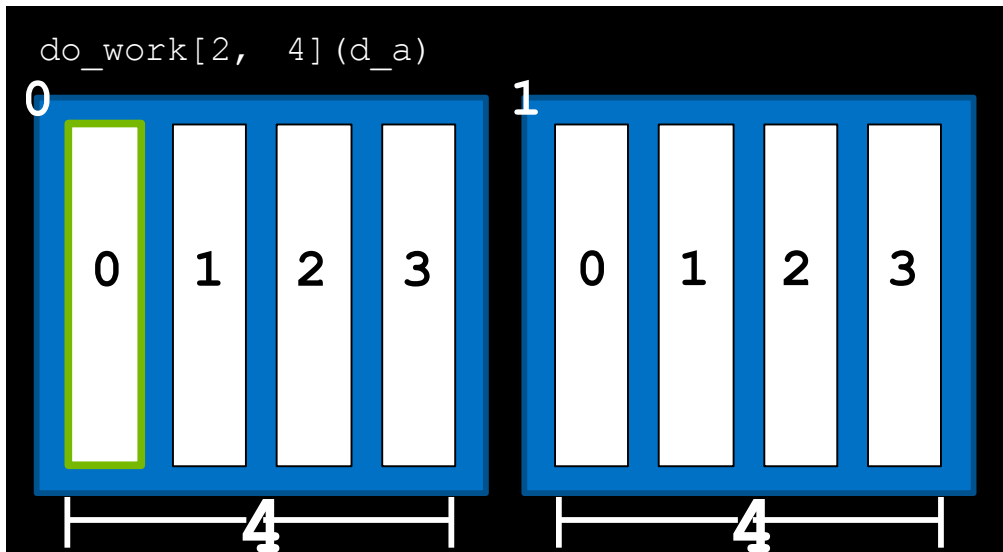
GPU  
DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
0		0		4

data_index
?

GPU



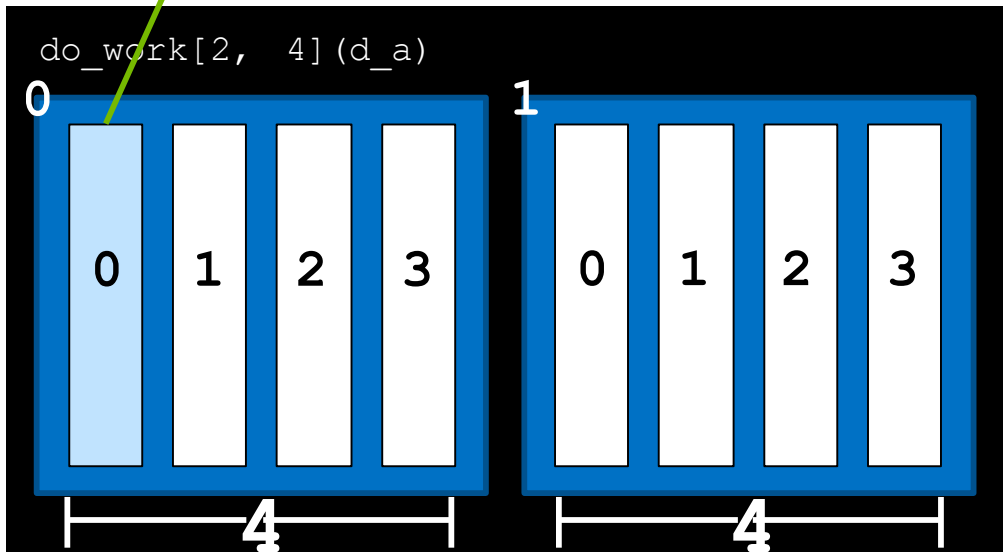
GPU  
DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
0		0		4

data_index
0

GPU



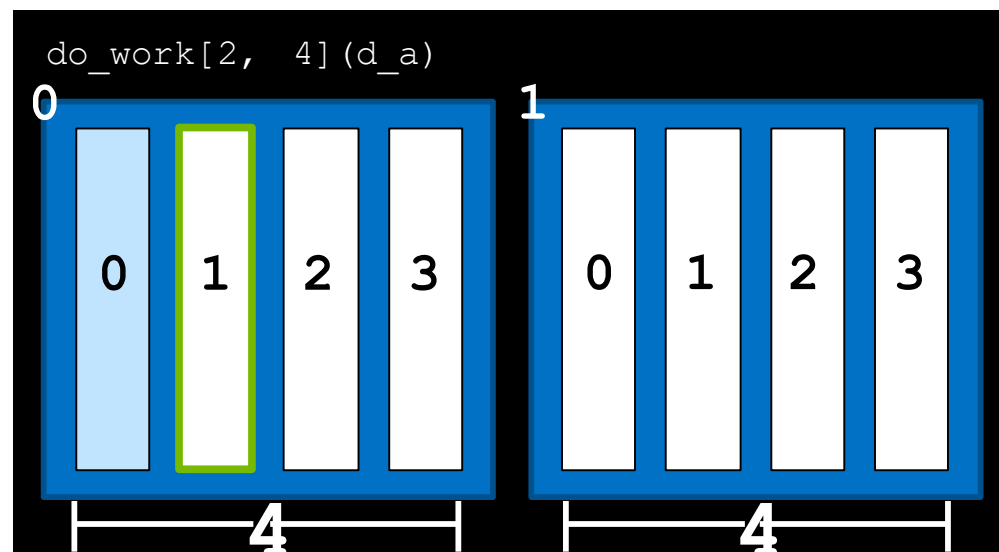
# GPU DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
1		0		4

data_index
?

# GPU



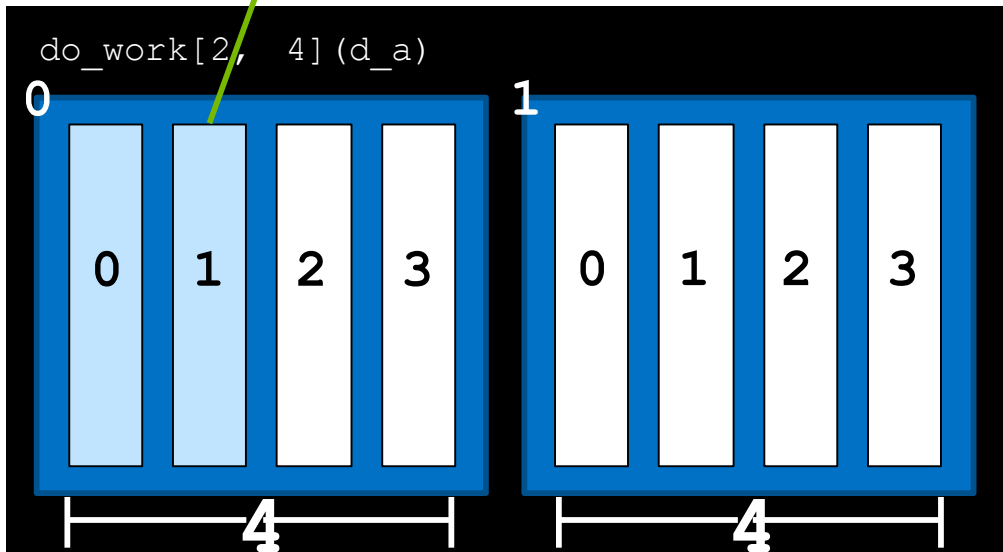
GPU  
DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
1		0		4

data_index
1

GPU



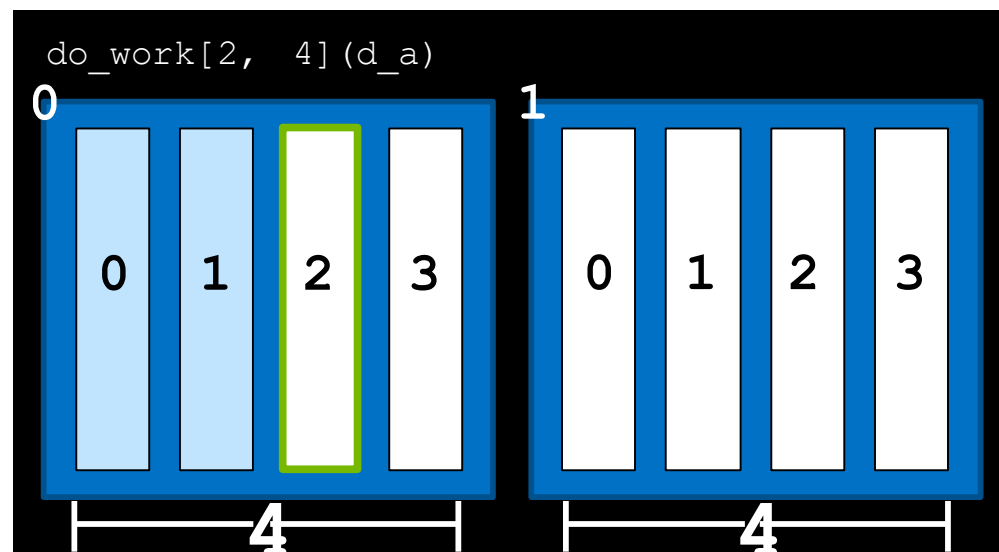
## GPU DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
2		0		4

data_index
?

## GPU





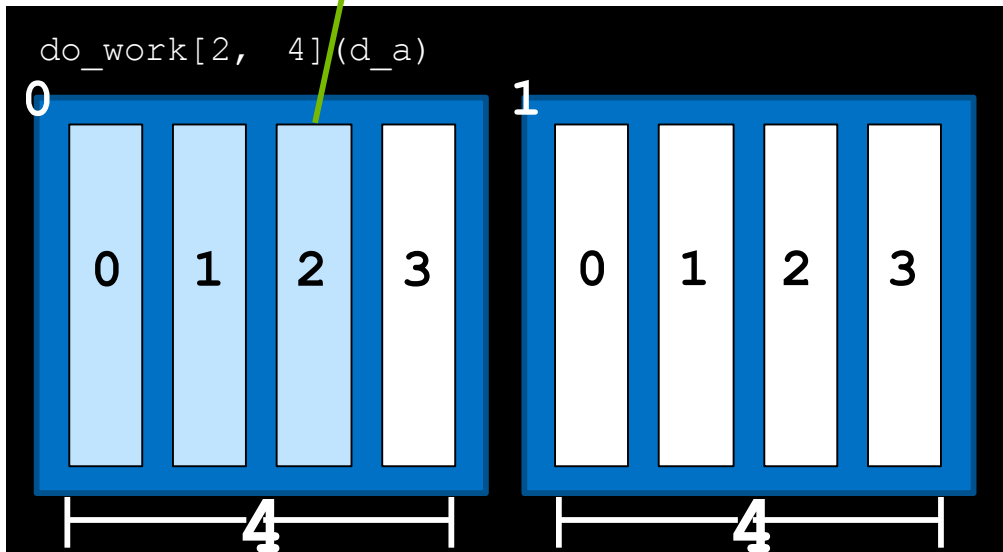
GPU  
DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
2		0		4

data_index
2

GPU



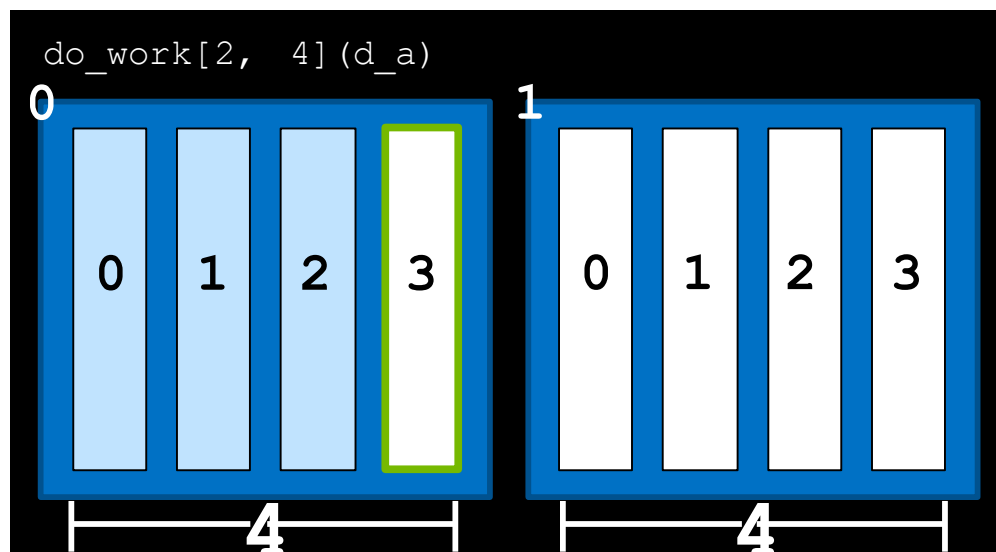
# GPU DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
3		0		4

data_index
?

# GPU



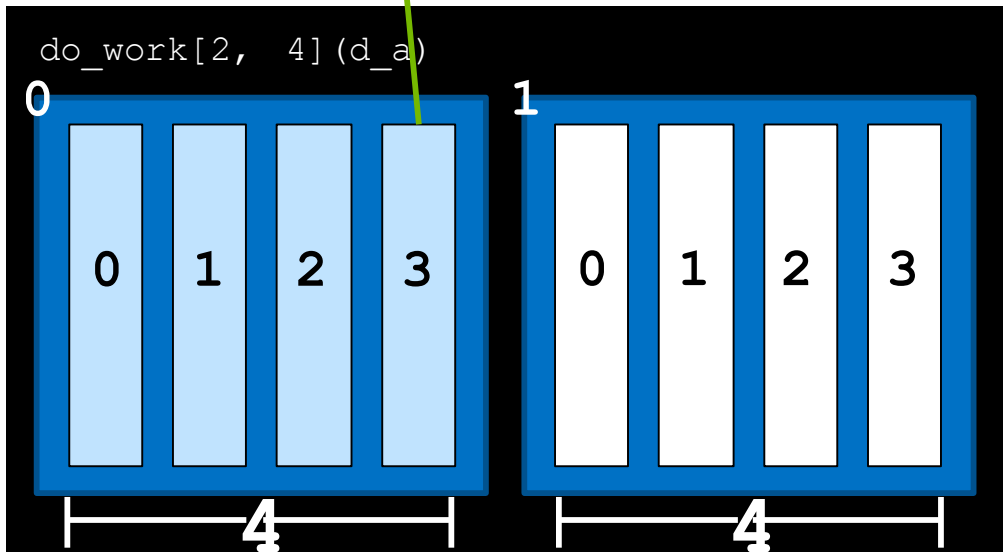
GPU  
DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
3		0		4

data_index
3

GPU



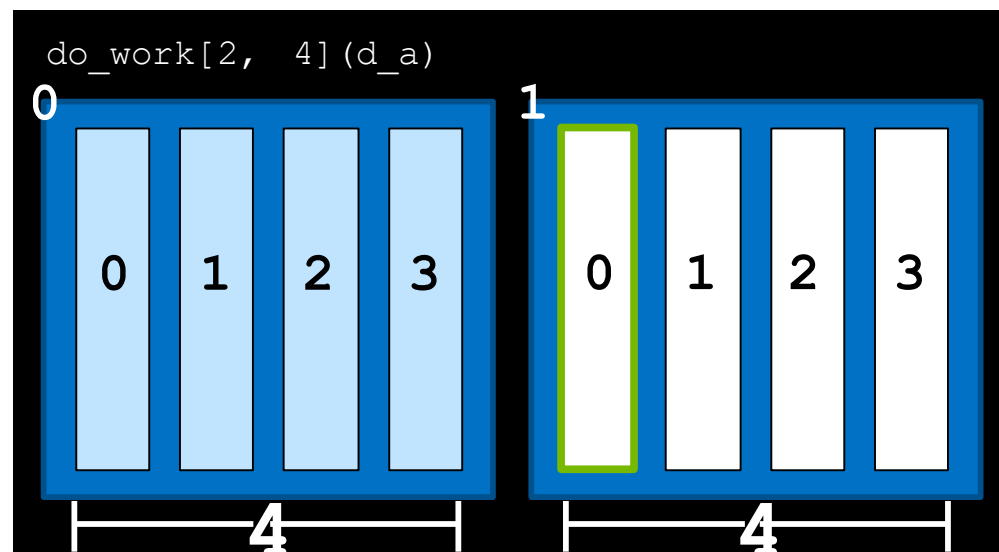
## GPU DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
0		1		4

data_index
?

## GPU



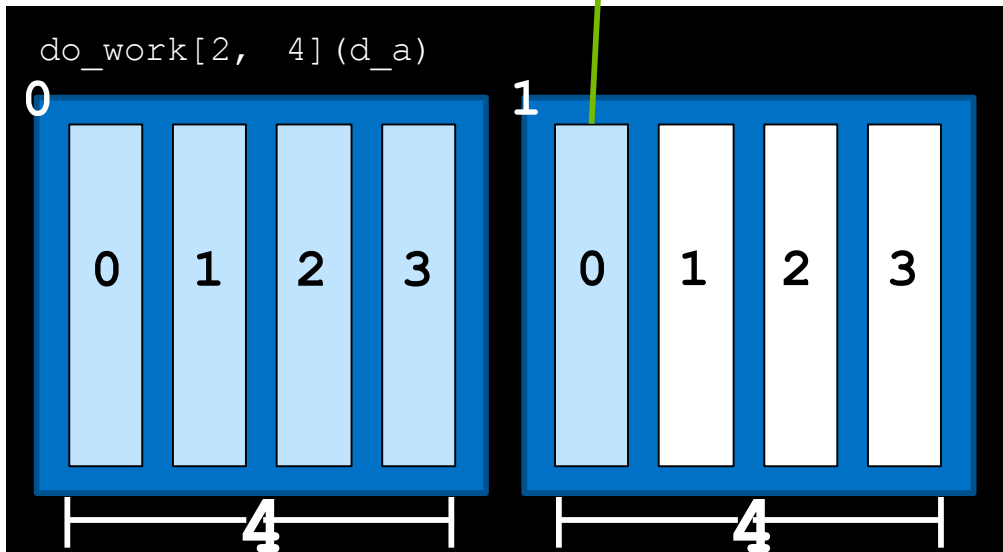
GPU  
DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
0		1		4

data_index
4

GPU



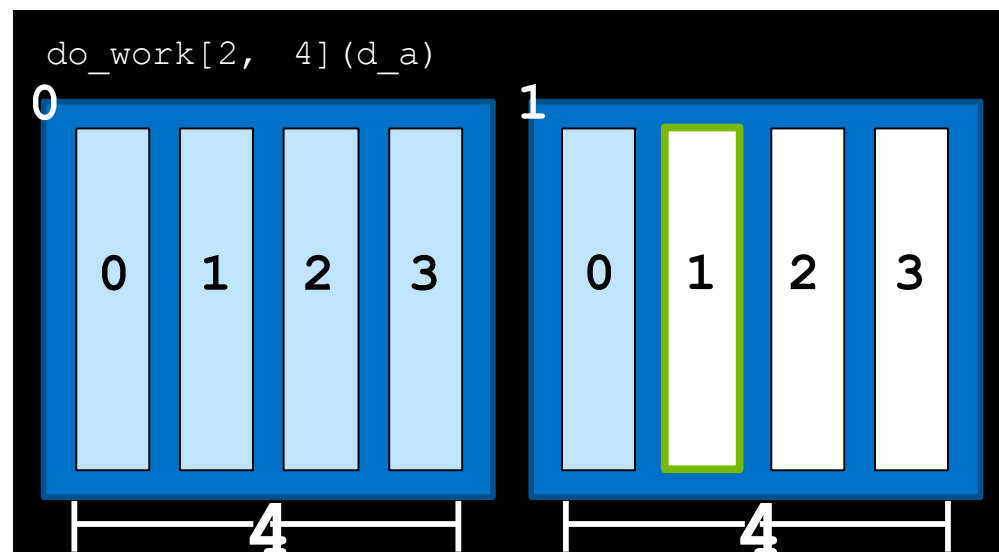
## GPU DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
1		1		4

data_index
?

## GPU



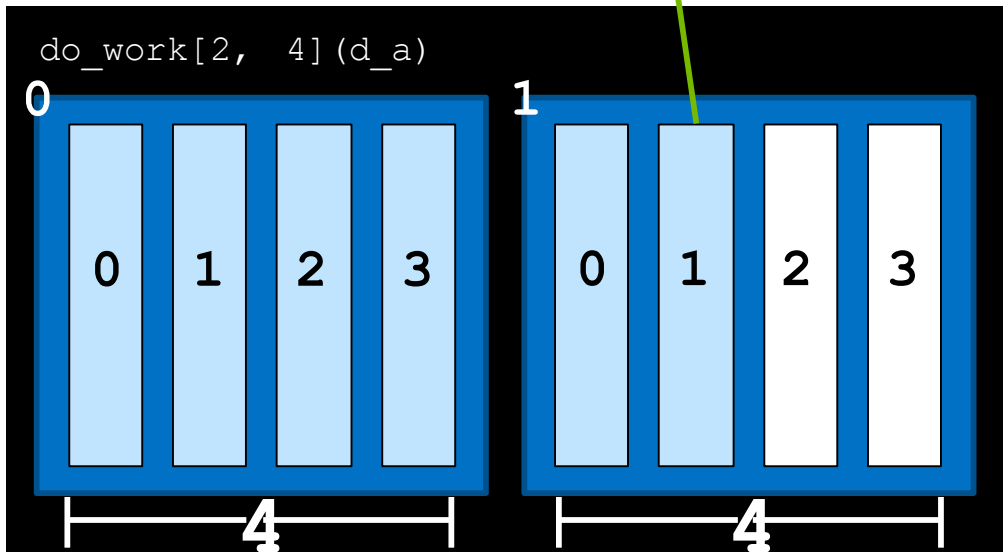
GPU  
DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
1		1		4

data_index
5

GPU



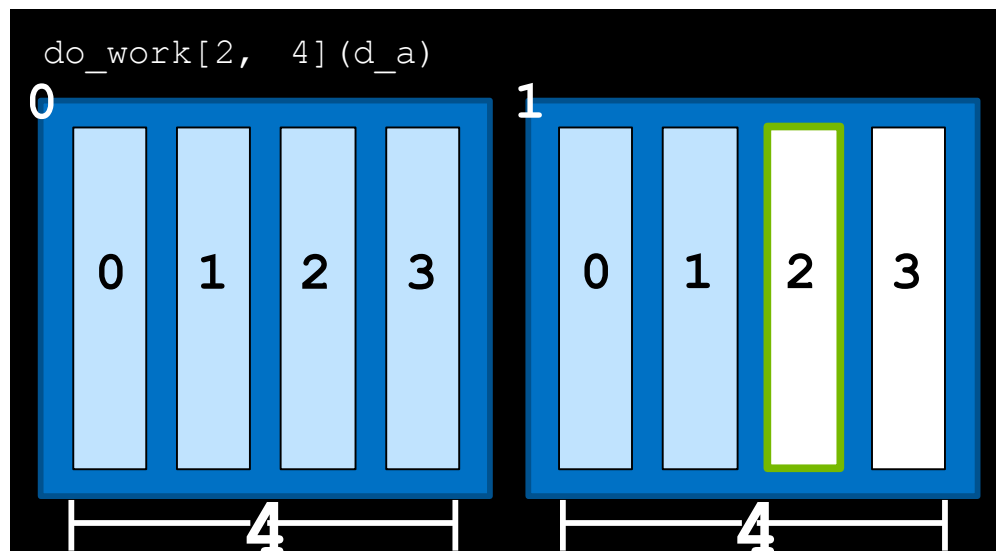
# GPU DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
2		1		4

data_index
?

# GPU





GPU  
DATA

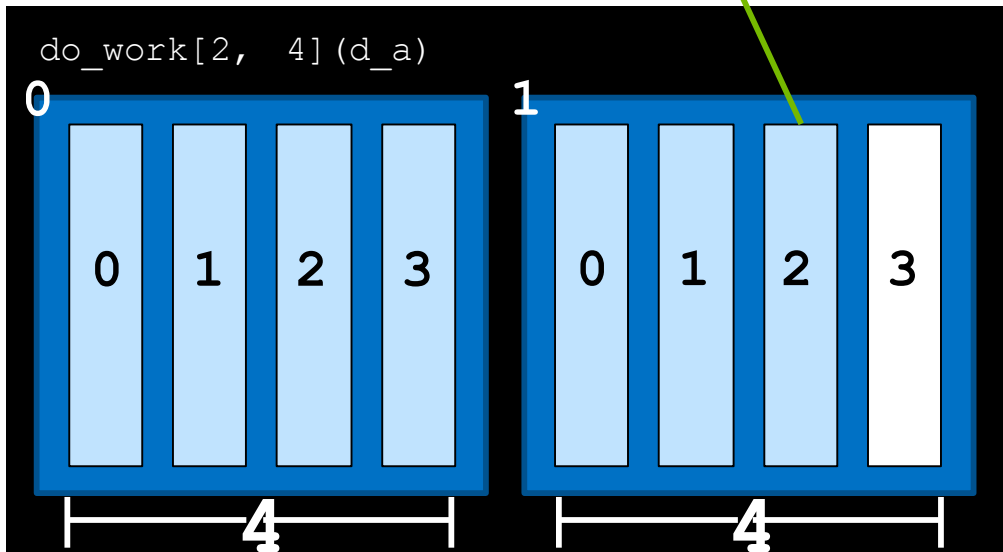
0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
2		1		4

data_index
6



GPU



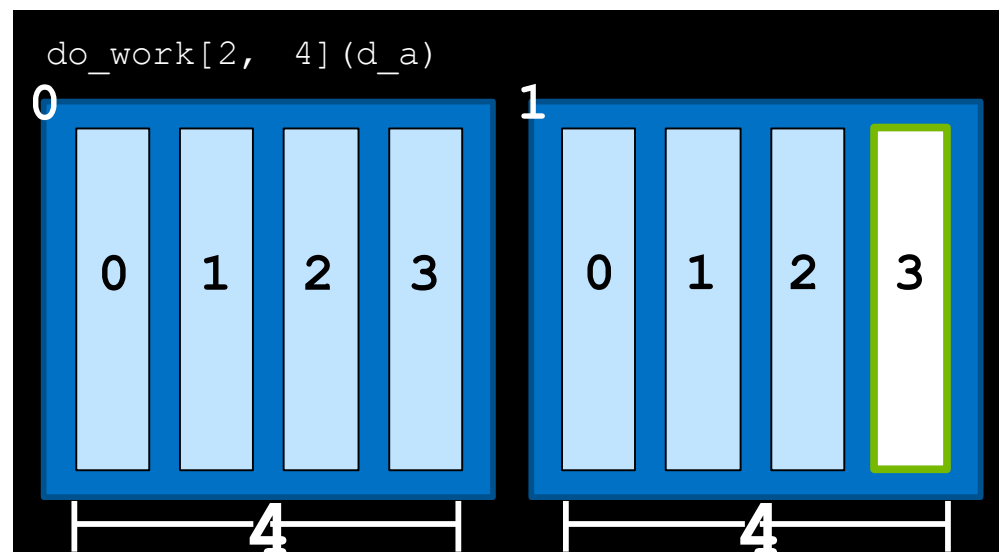
## GPU DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
3		1		4

data_index
?

## GPU



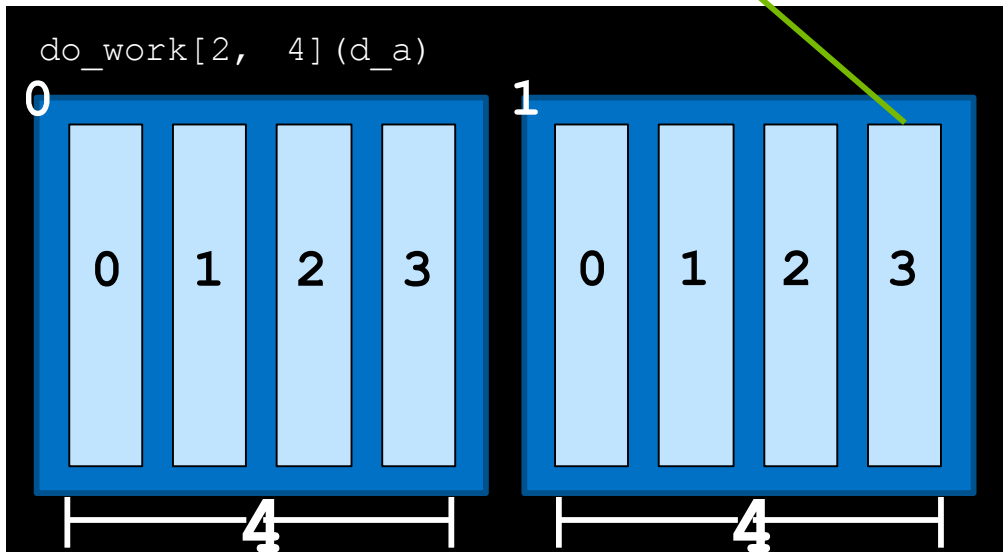
GPU  
DATA

0	4
1	5
2	6
3	7

threadIdx.x	+	blockIdx.x	*	blockDim.x
3		1		4

data_index
7

GPU



## GPU DATA

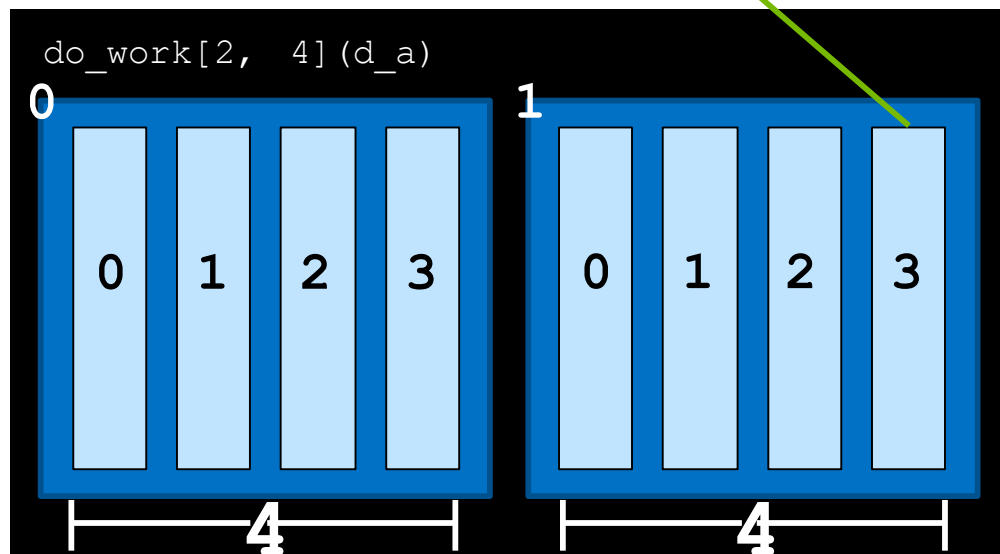
0	4
1	5
2	6
3	7

<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
3		1		4

<code>data_index</code>
7
<code>grid(1)</code>
7

As a convenience, Numba provides the `cuda.grid()` function, which will return a thread's unique index in the grid.

## GPU





DEEP  
LEARNING  
INSTITUTE

[www.nvidia.com/dli](http://www.nvidia.com/dli)