



DEEP
LEARNING
INSTITUTE

Accelerating Applications with CUDA C/C++

TOPICS

GPU-accelerated vs. CPU-only Applications

CUDA Kernel Execution

Parallel Memory Access

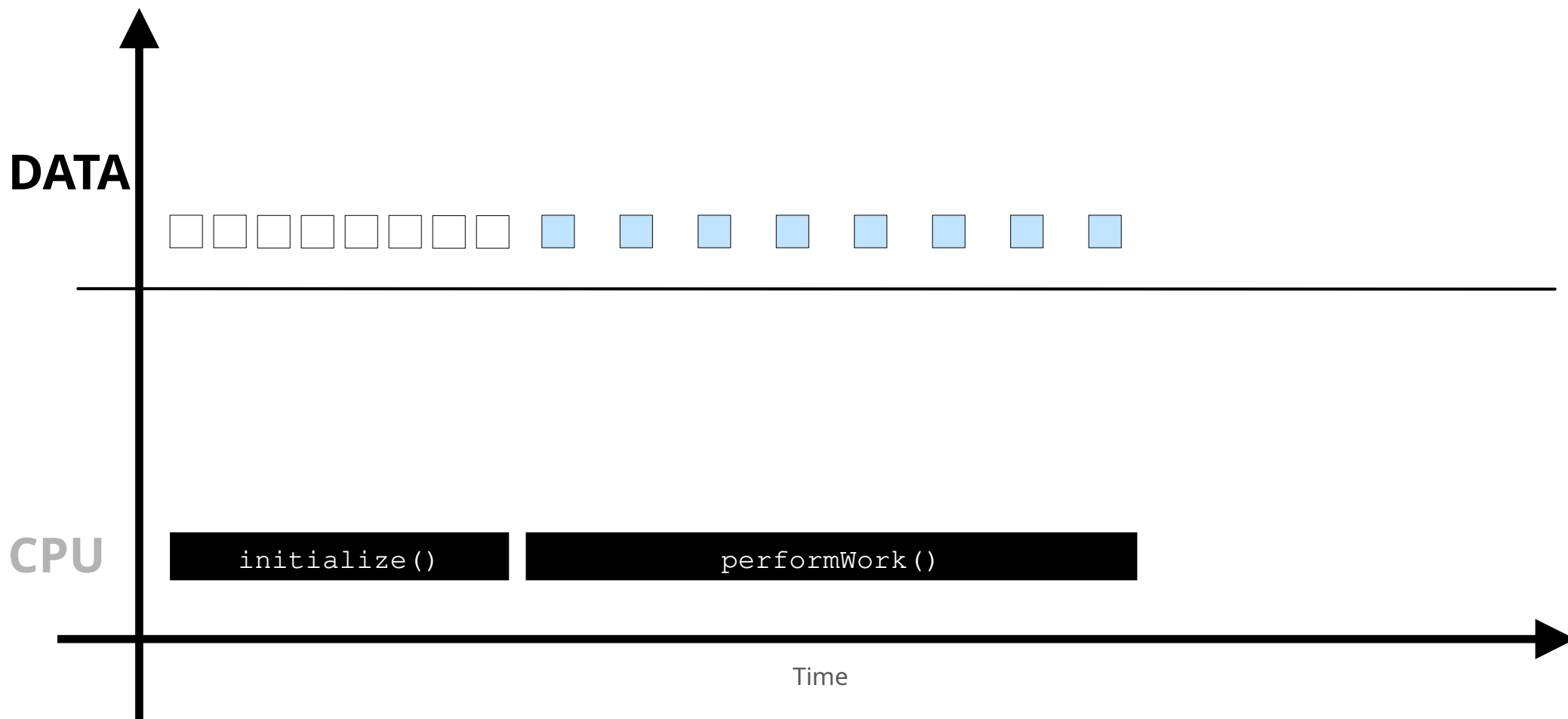
Appendix: Glossary

GPU-accelerated vs. CPU-only Applications

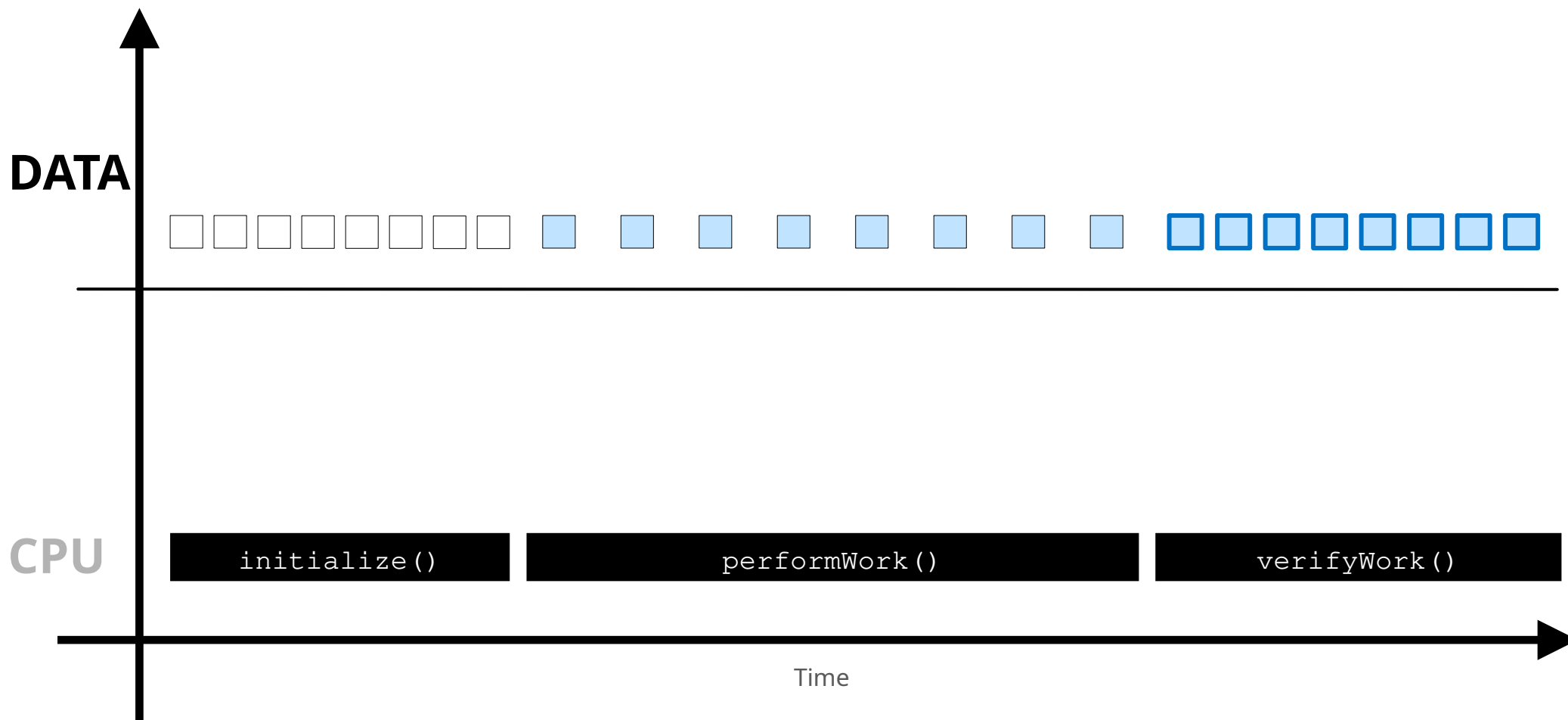
In **CPU-only applications** data is allocated on CPU



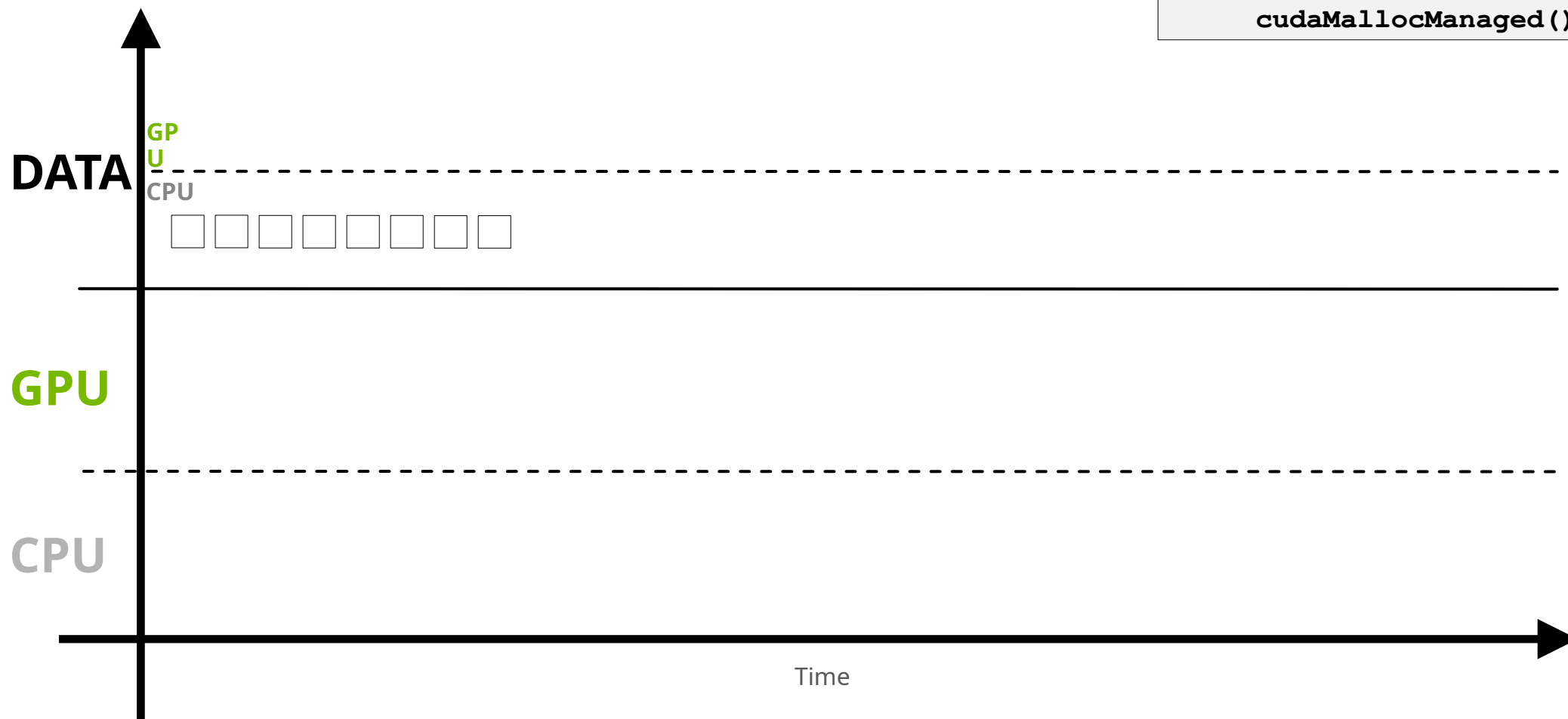
...and all work is performed on CPU



...and all work is performed on CPU



In **accelerated applications** data is
allocated with
cudaMallocManaged()

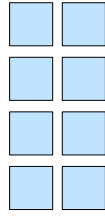


... where it can be accessed and worked on by the CPU



DATA

GPU
CPU



... and automatically migrated to the GPU where parallel work can be done

GPU

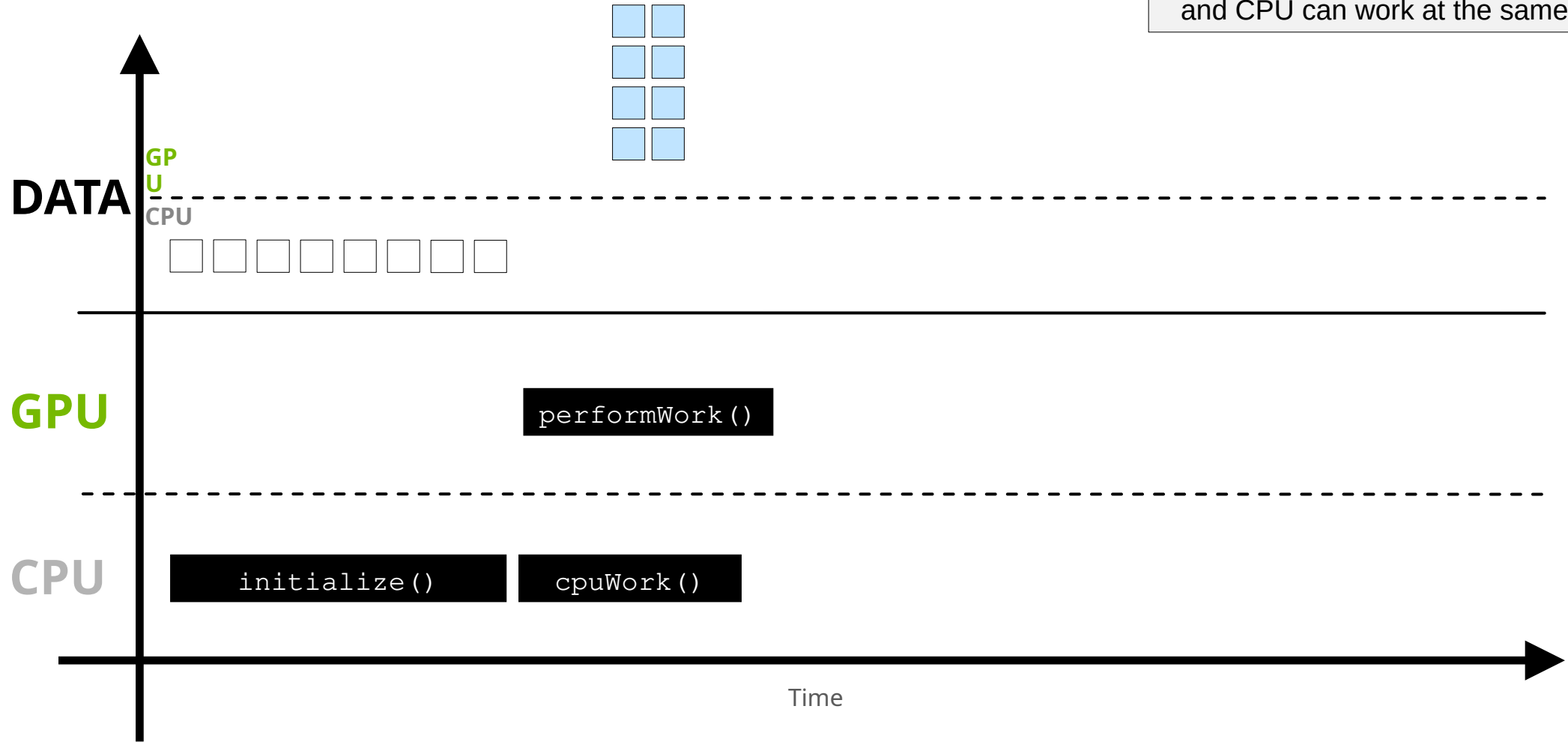
`performWork()`

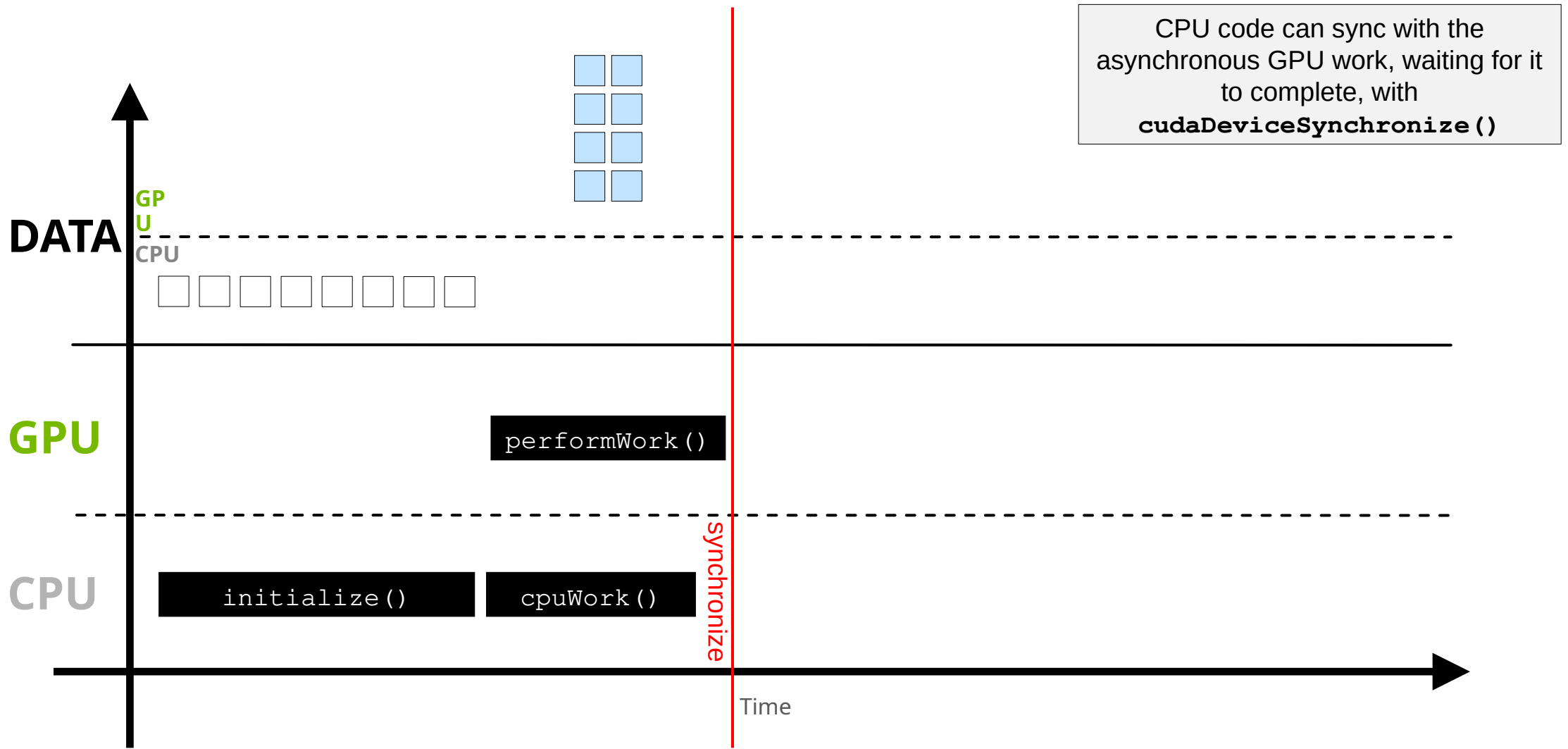
CPU

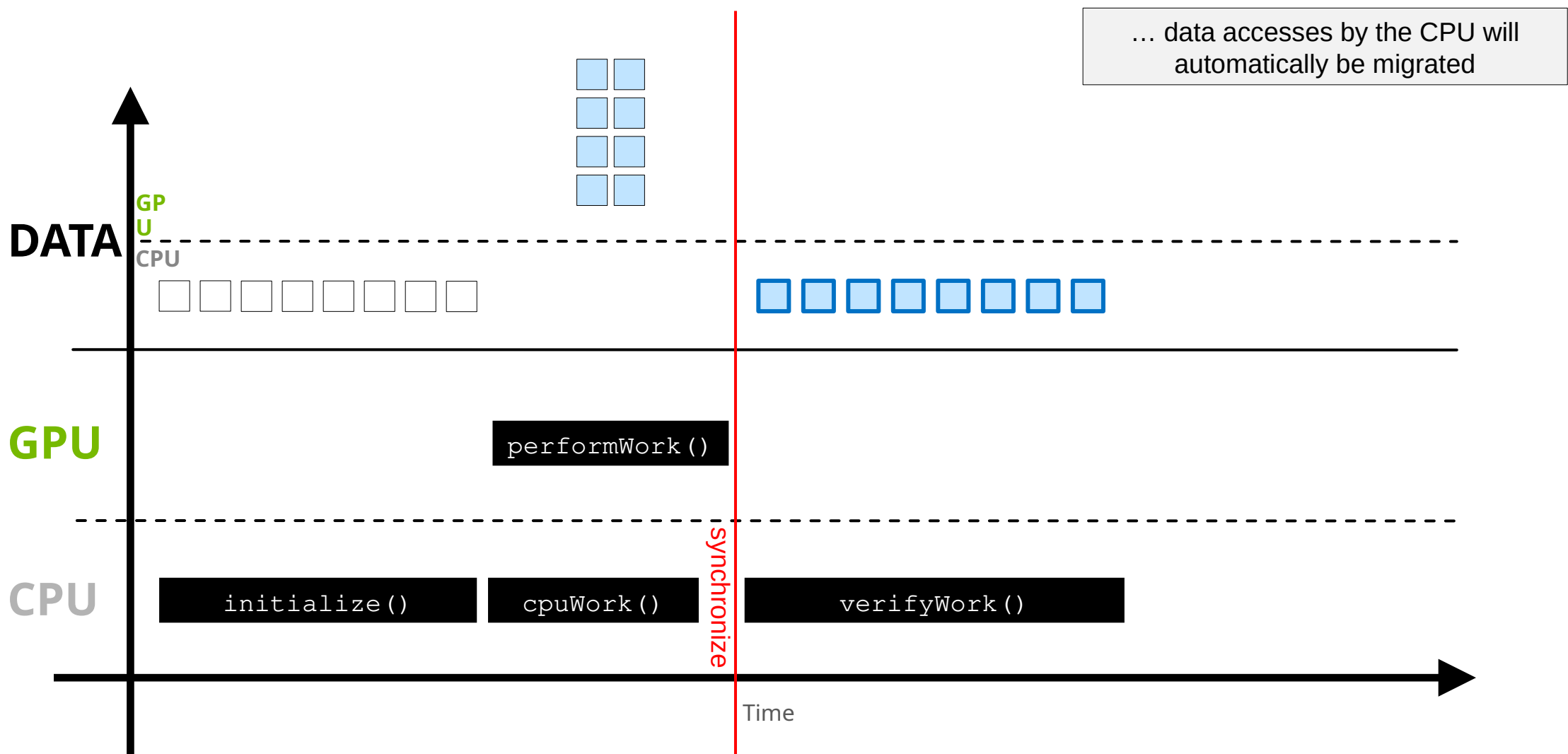
`initialize()`

Time

Work on the GPU is **asynchronous**,
and CPU can work at the same time

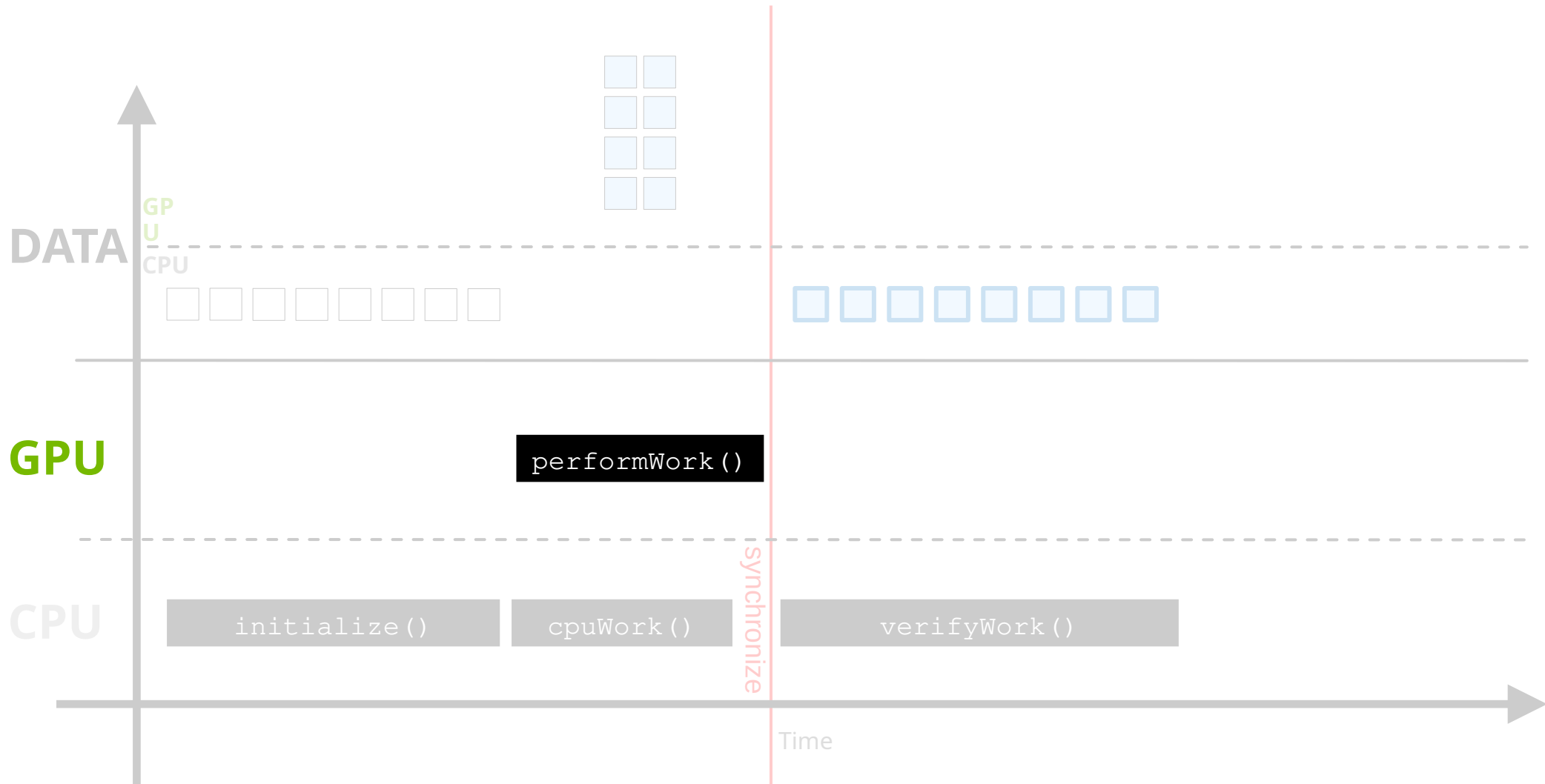




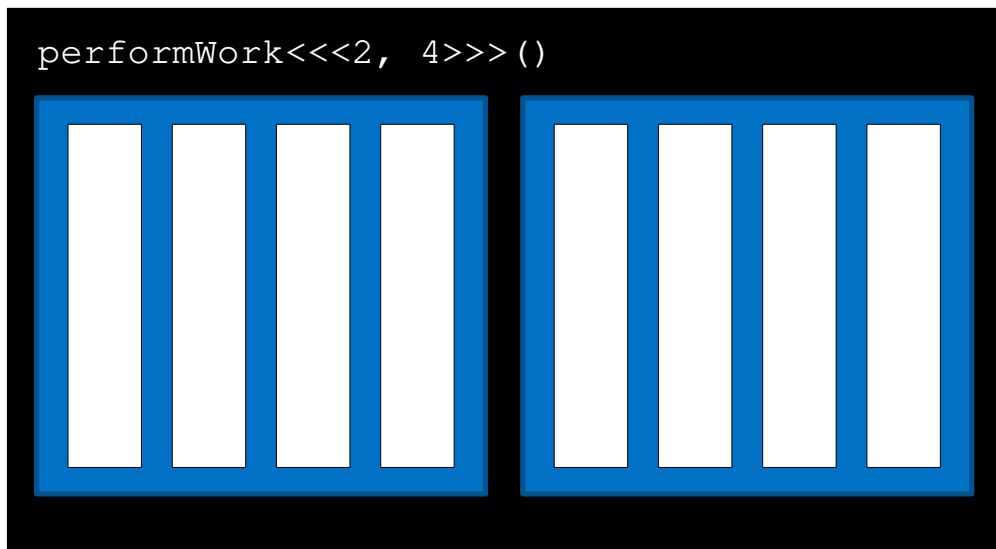


CUDA Kernel Execution

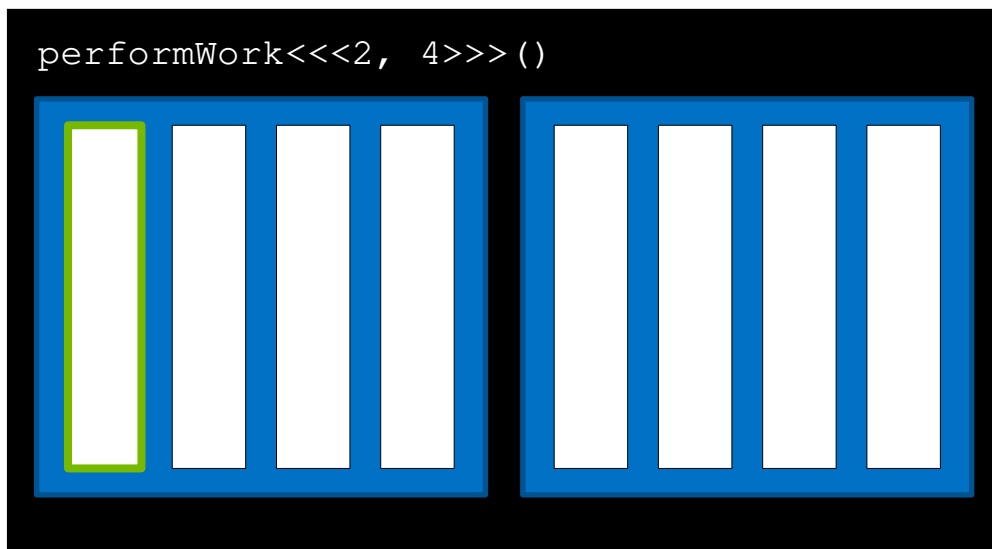
The background of the slide features a smooth gradient transitioning from a vibrant green on the left to a deep blue on the right. Overlaid on this gradient is a complex, abstract network of white dots and thin white lines, resembling a digital or neural network structure. The dots are of varying sizes and are interconnected by lines, creating a sense of depth and connectivity.



GPU



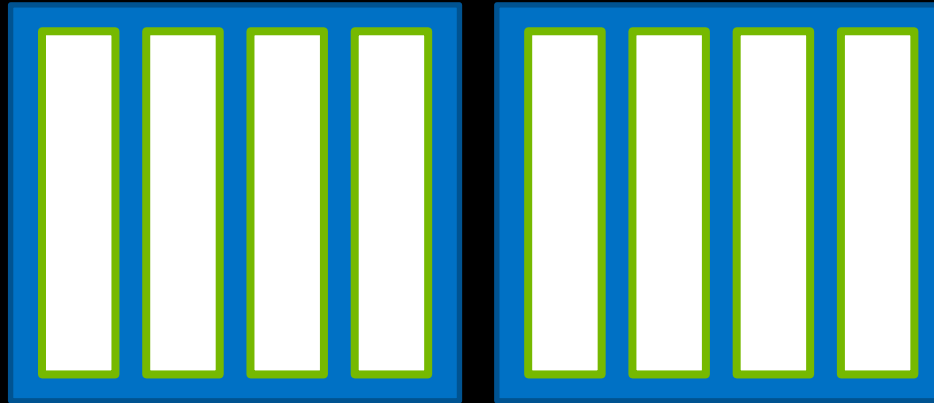
GPU



Many threads run in parallel

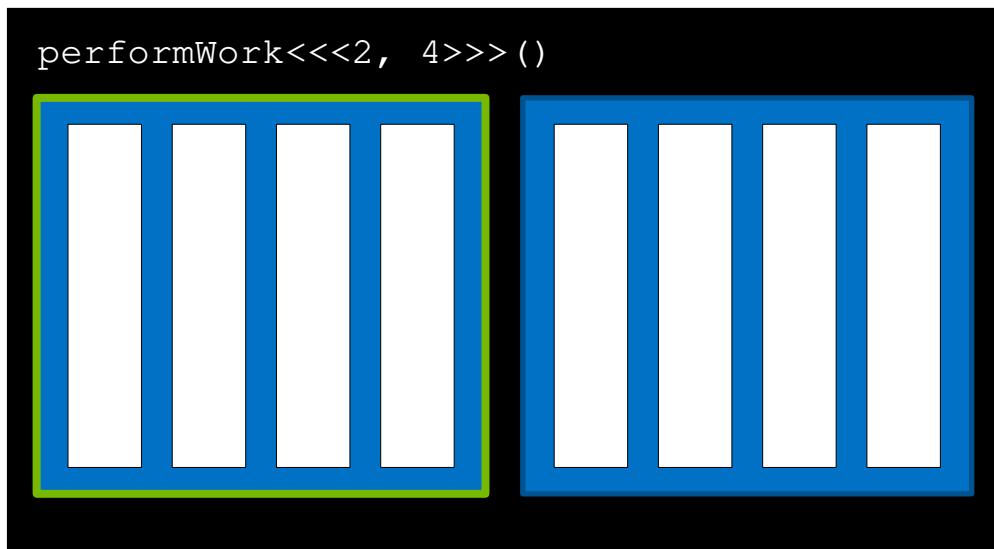
GPU

```
performWork<<<2, 4>>>()
```



A collection of threads is a **block**

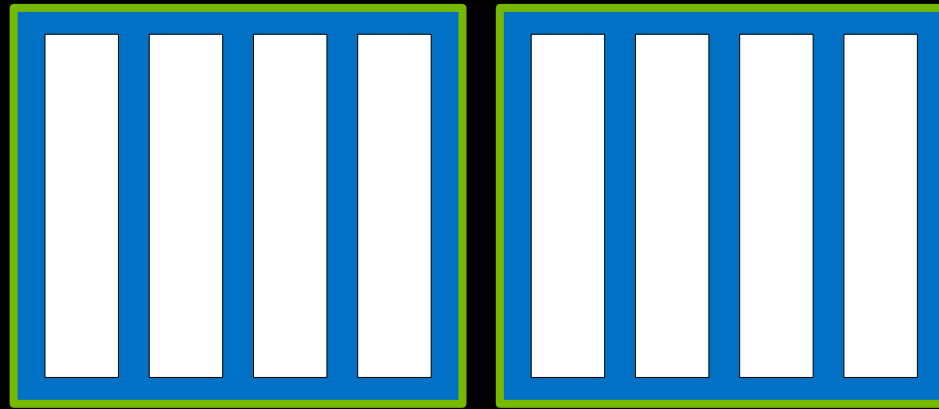
GPU



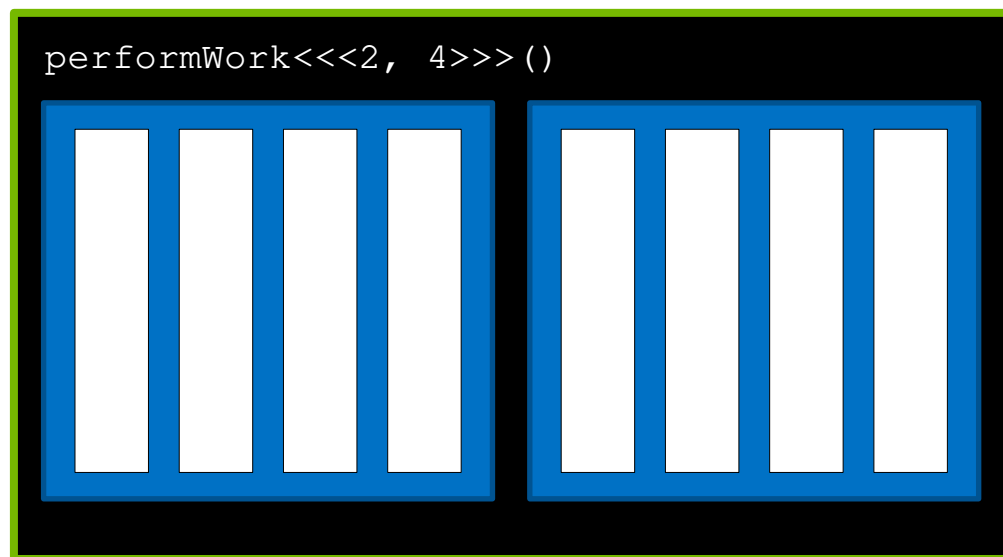
There are many blocks

GPU

```
performWork<<<2, 4>>>()
```

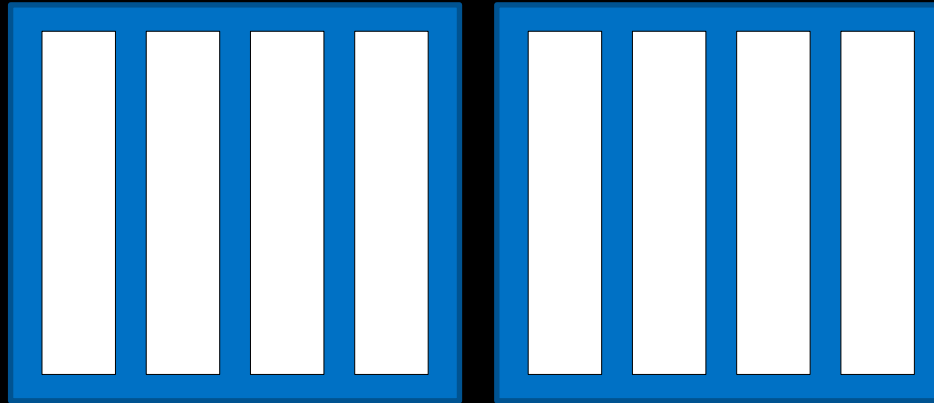


GPU



GPU

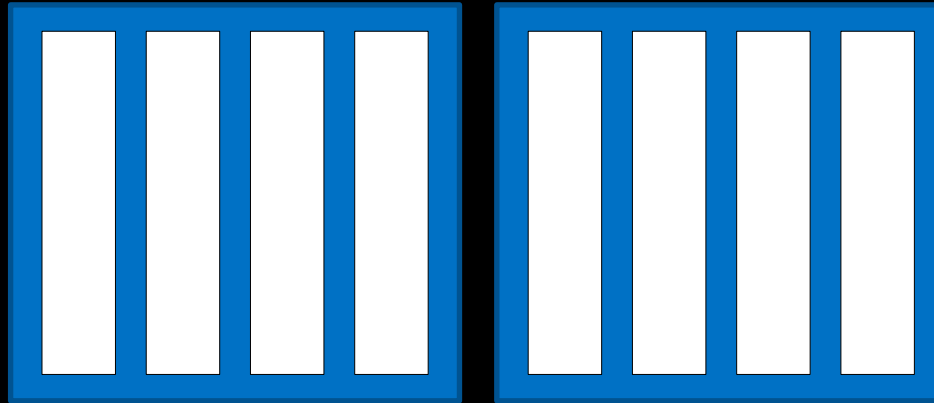
```
performWork<<<2, 4>>>()
```



Kernels are **launched** with an
execution configuration

GPU

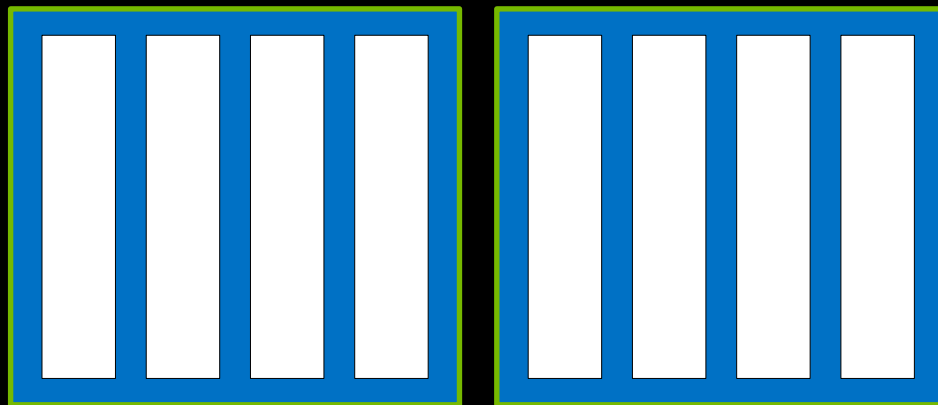
```
performWork<<<2, 4>>>()
```



The execution configuration defines
the number of blocks in the grid

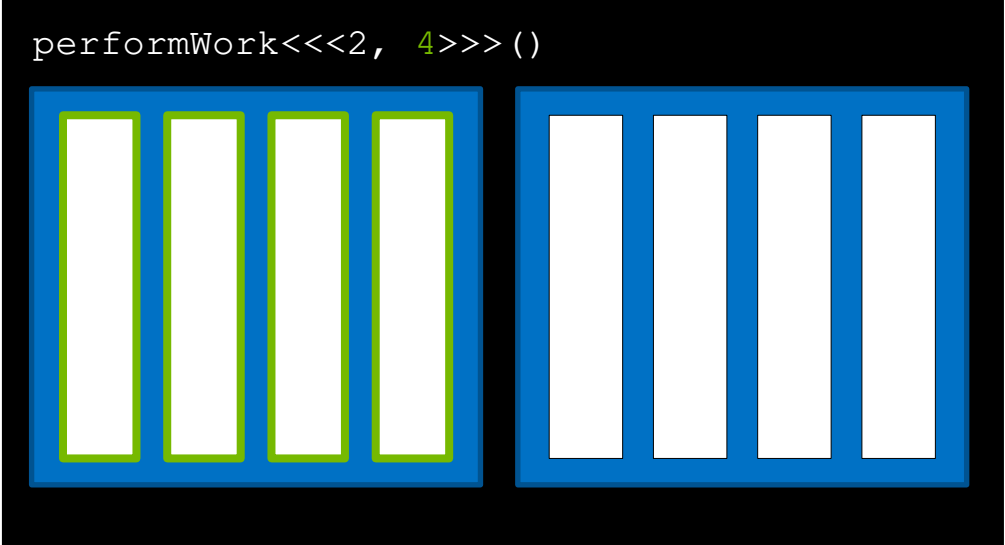
GPU

```
performWork<<<2, 4>>>()
```



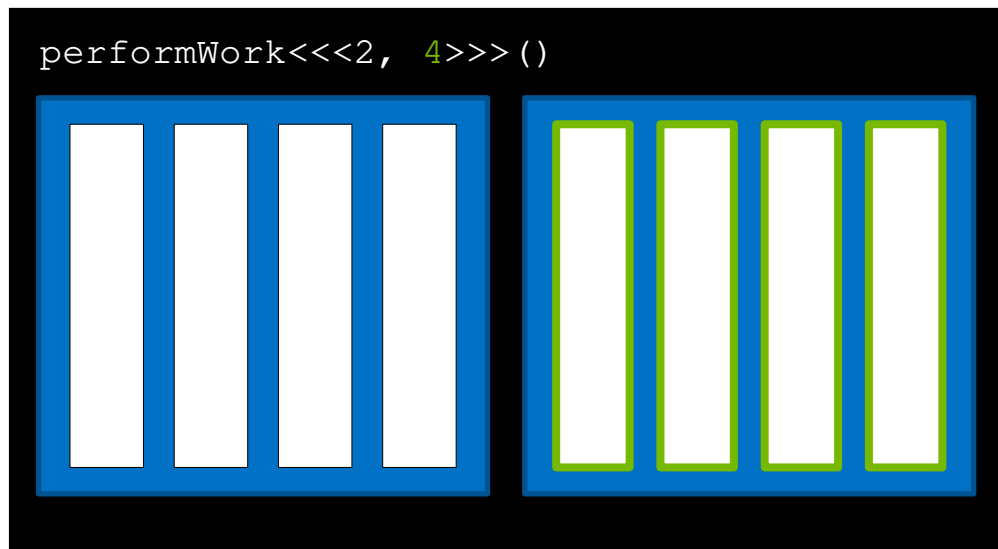
... as well as the number of threads in each block

GPU



Every block in the grid contains the same number of threads

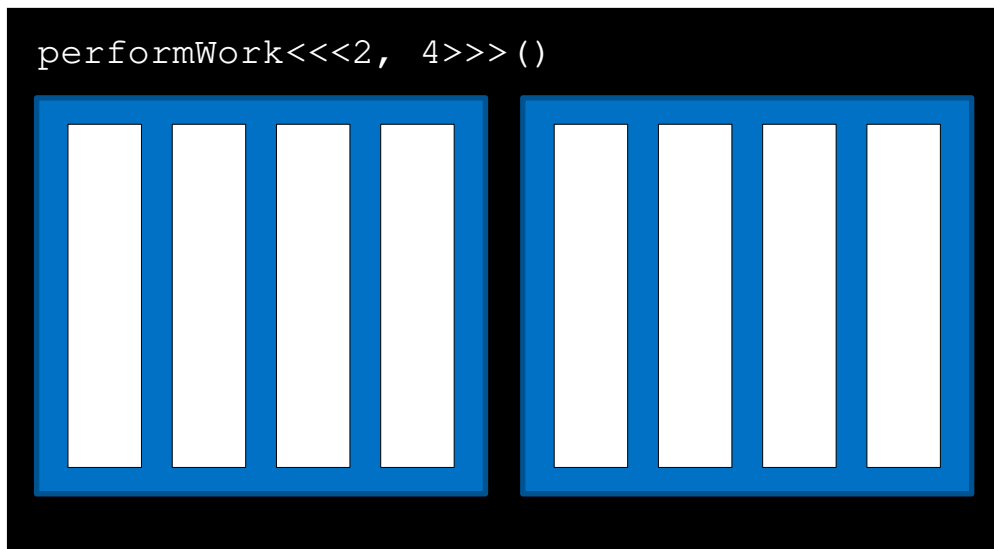
GPU



CUDA-Provided Thread Hierarchy Variables

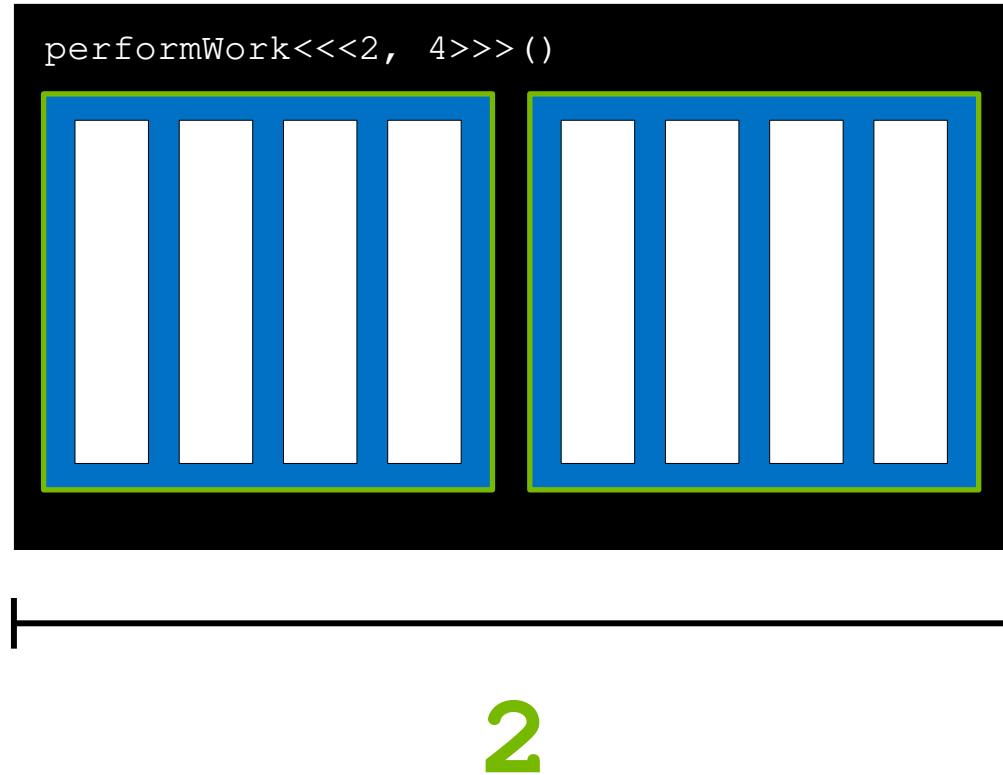
Inside kernels definitions, CUDA-provided variables describe its executing thread, block, and grid

GPU



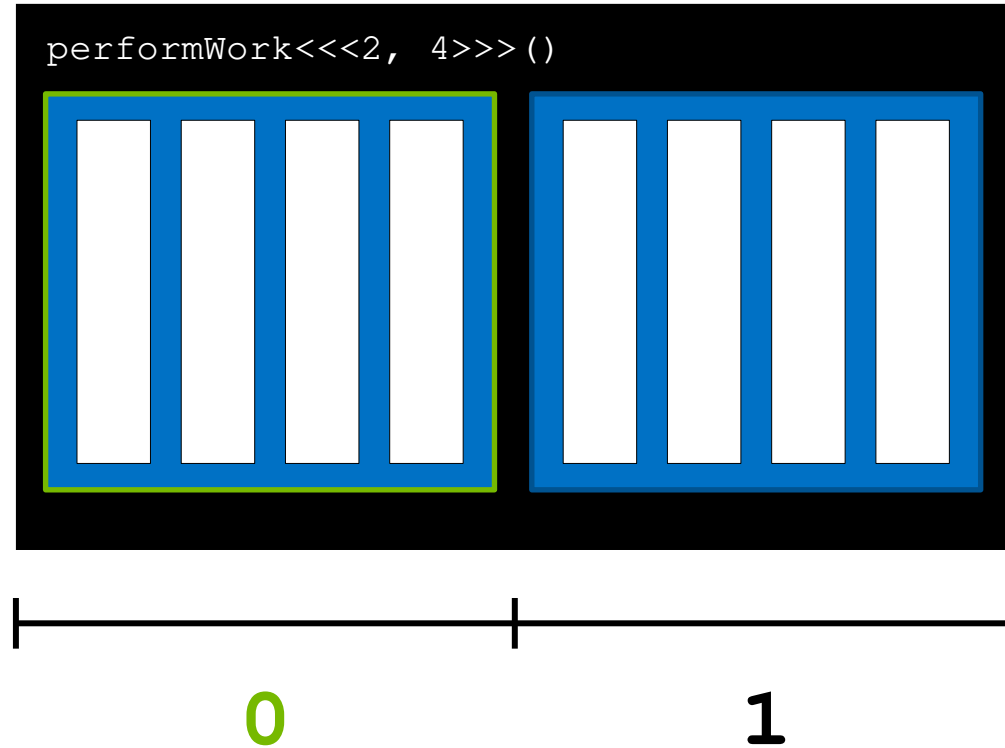
gridDim.x is the number of blocks in the grid, in this case **2**

GPU



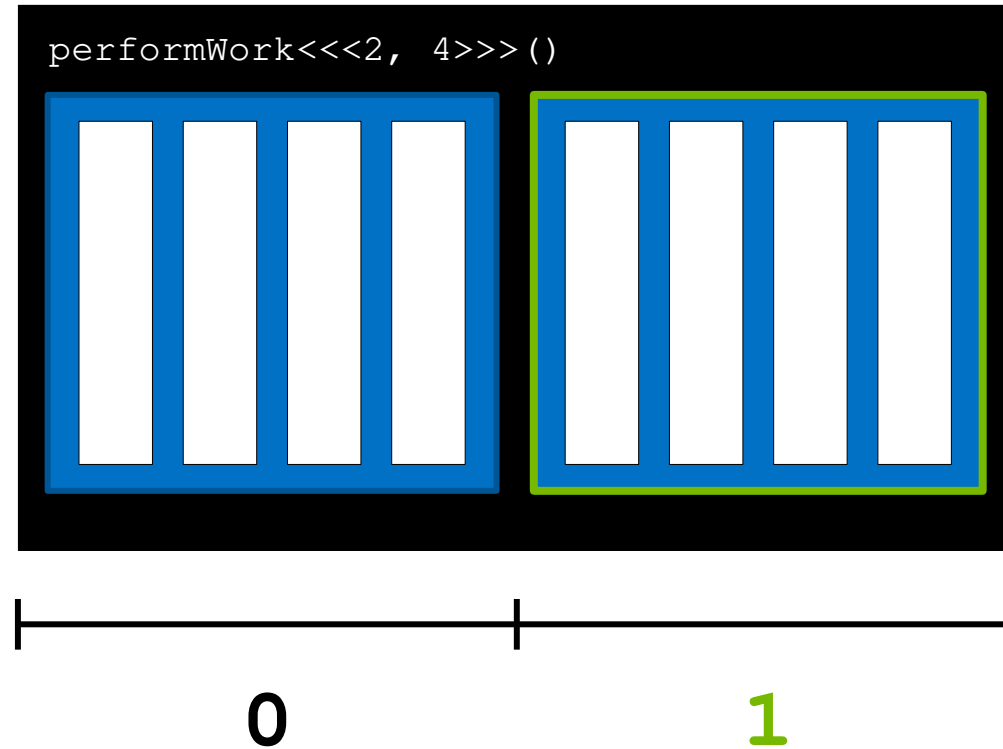
blockIdx.x is the index of the current block within the grid, in this case **0**

GPU



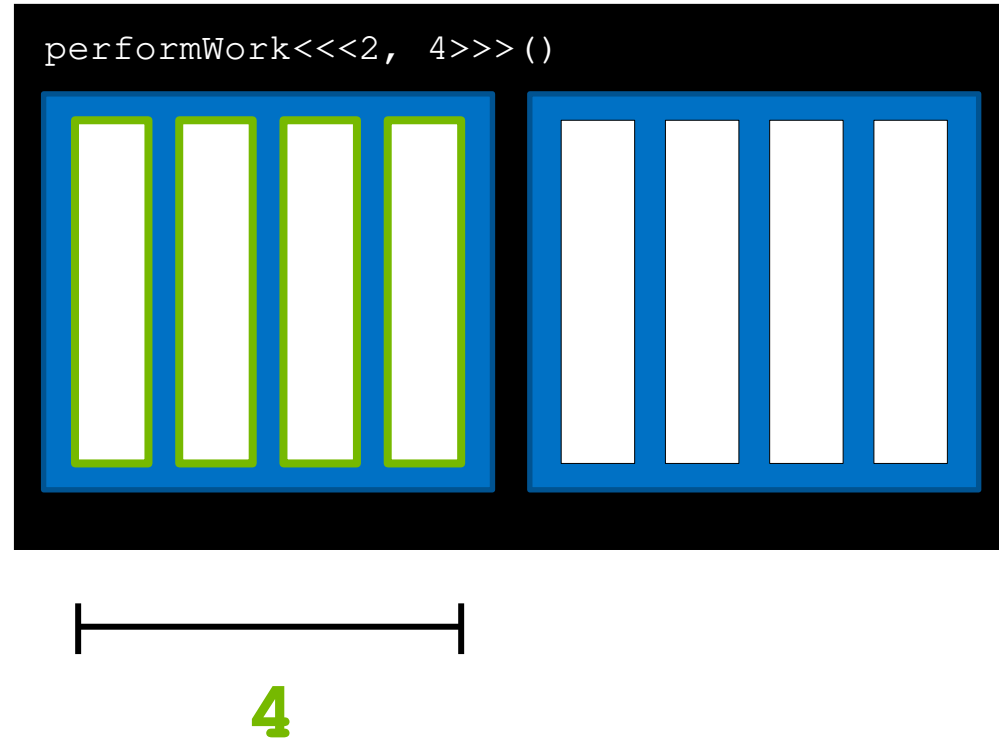
blockIdx.x is the index of the current block within the grid, in this case **1**

GPU



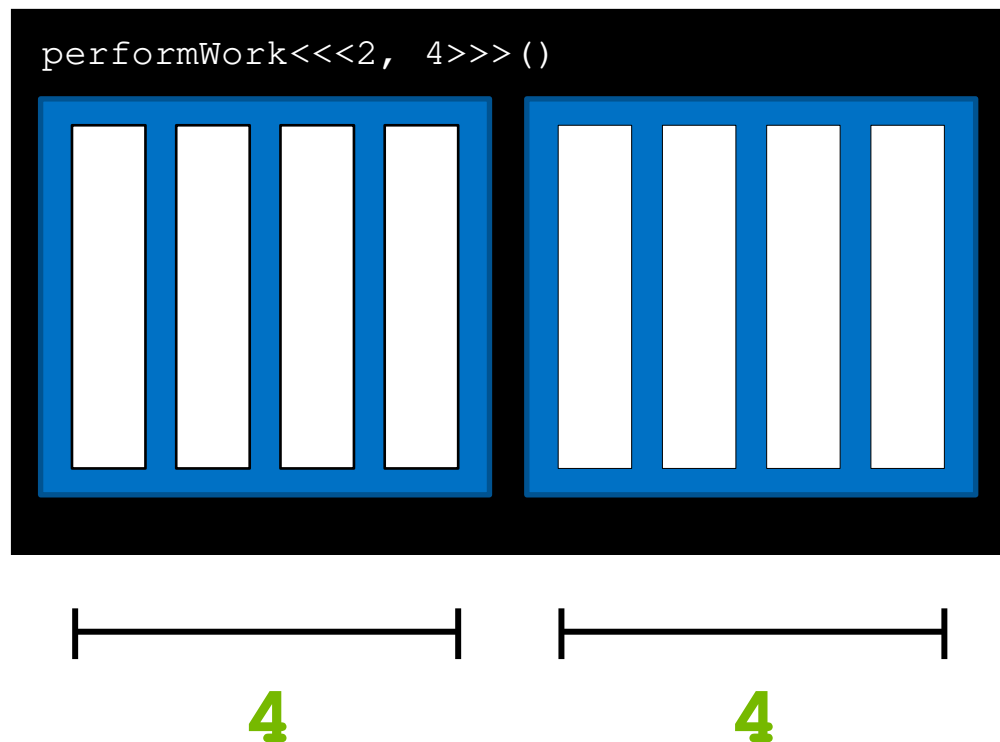
Inside a kernel `blockDim.x` describes the number of threads in a block. In this case **4**

GPU



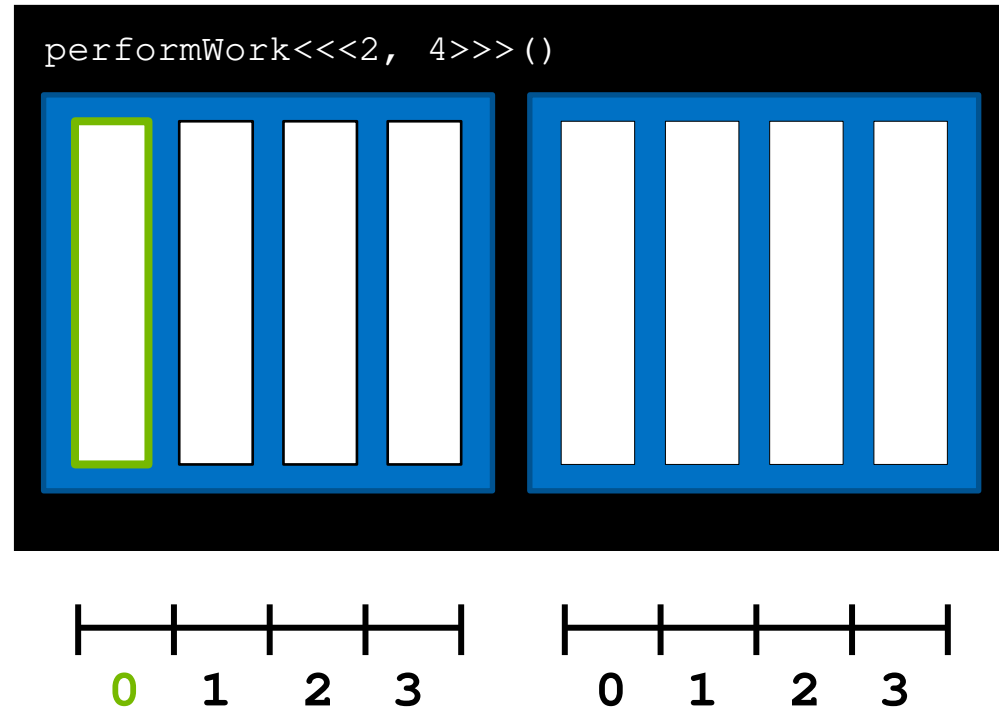
All blocks in a grid contain the same number of threads

GPU



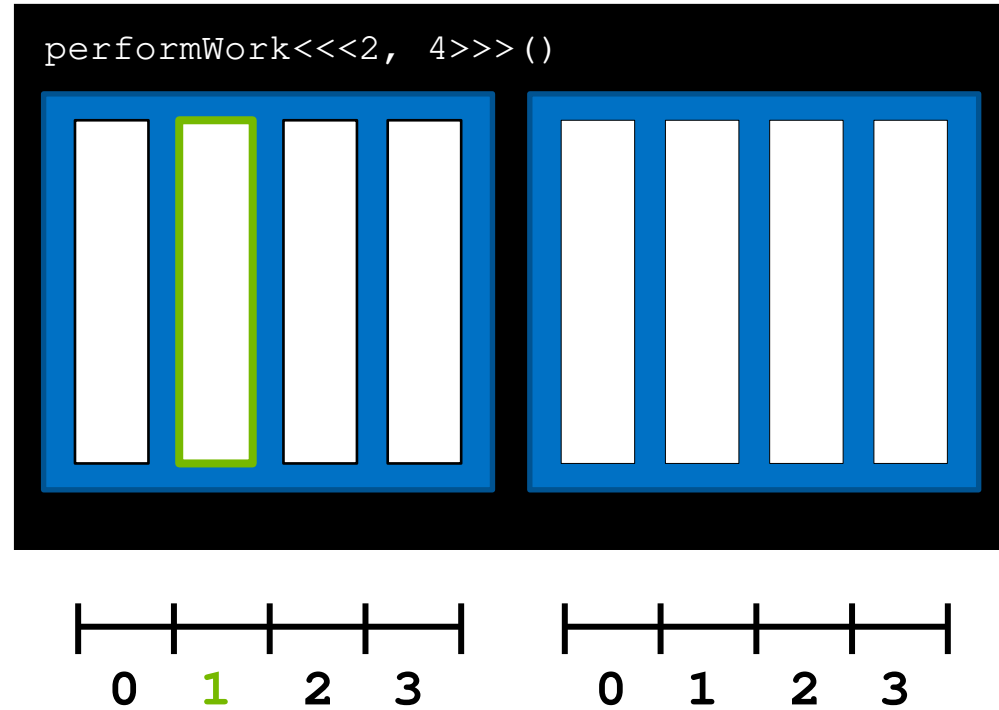
Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 0

GPU



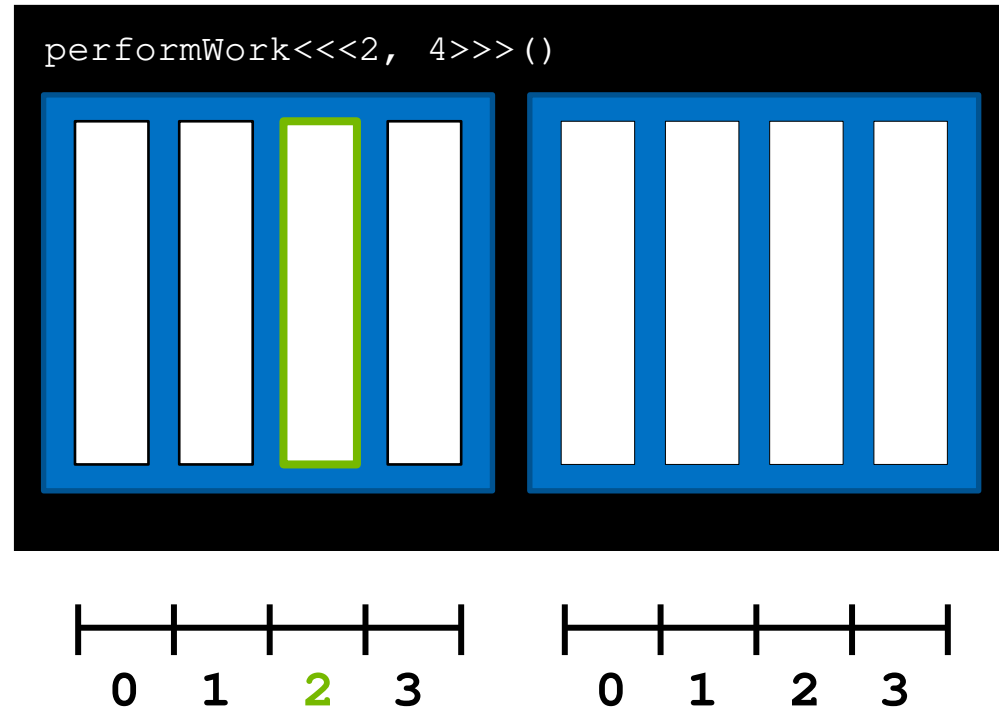
Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 1

GPU



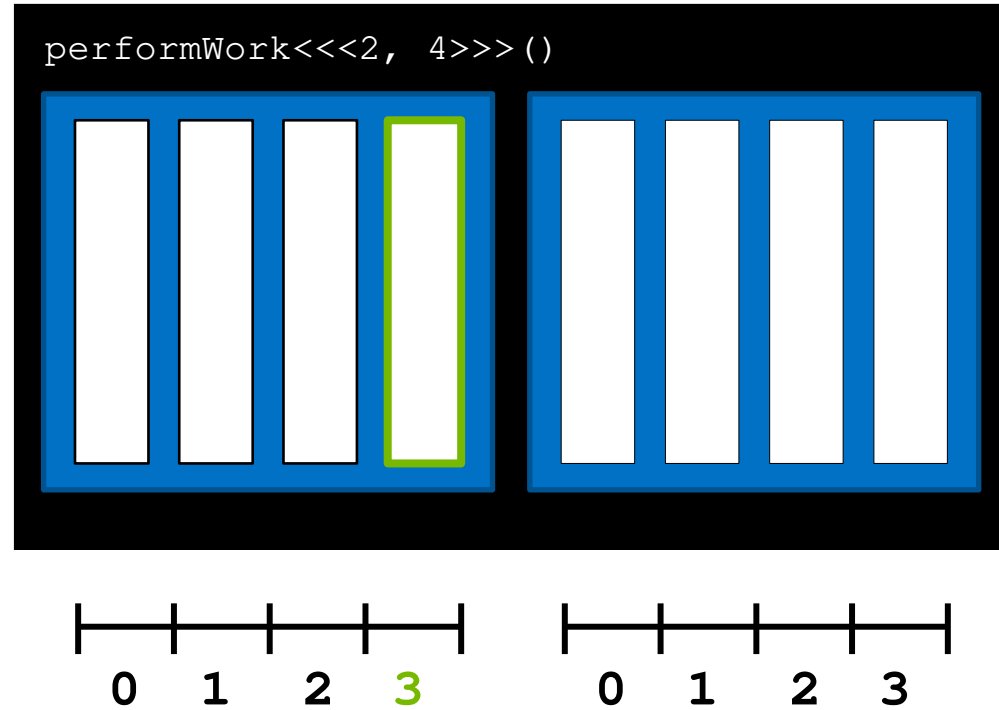
Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 2

GPU



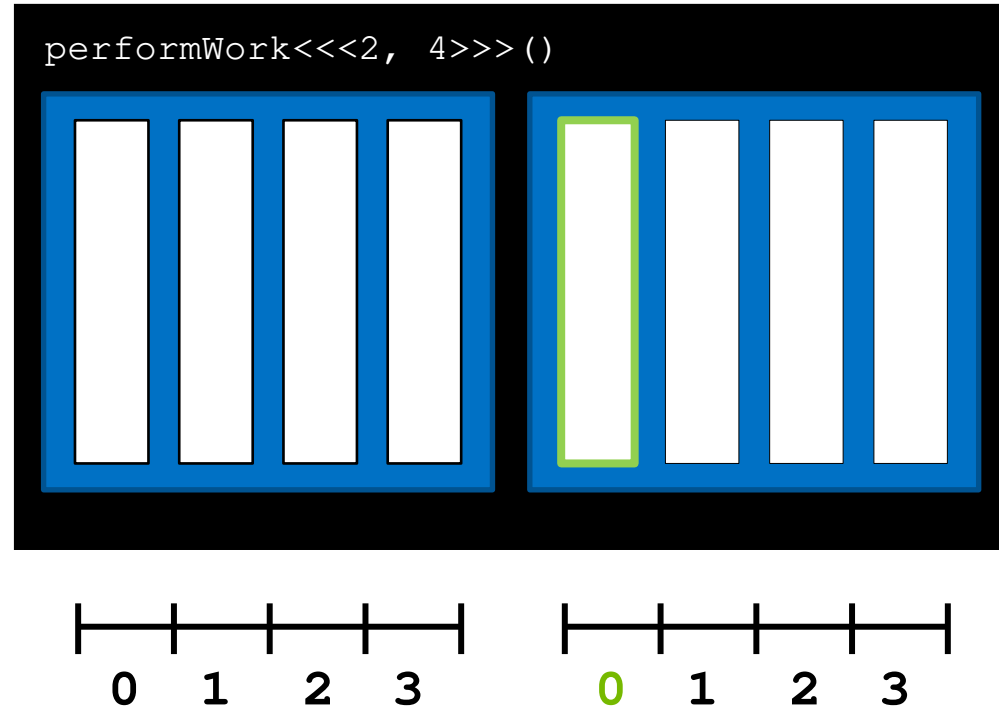
Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 3

GPU



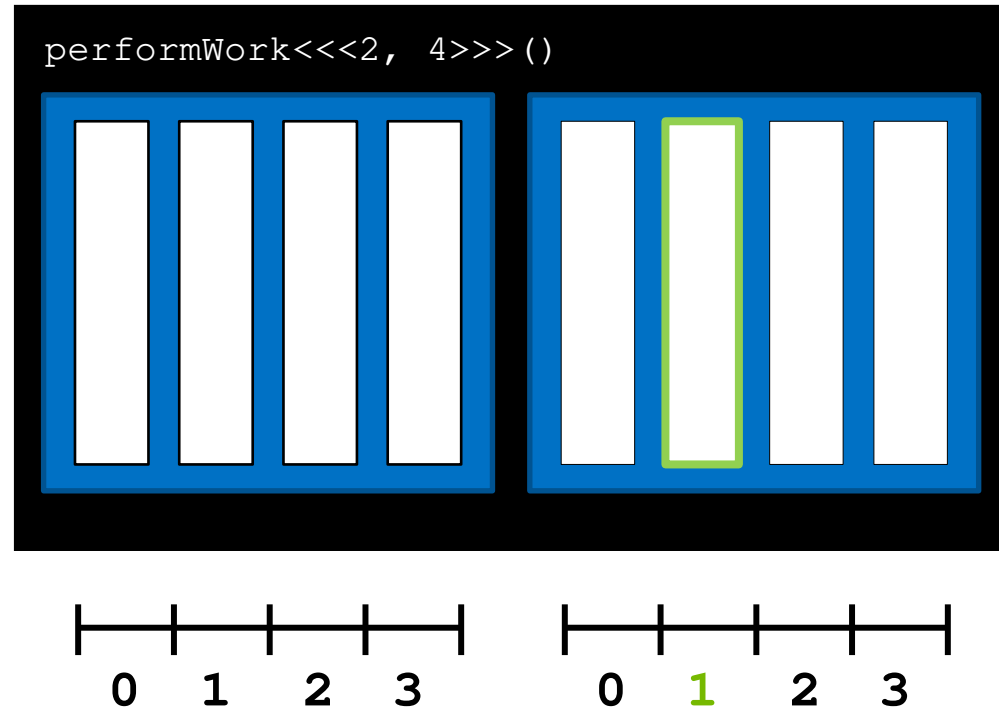
Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 0

GPU



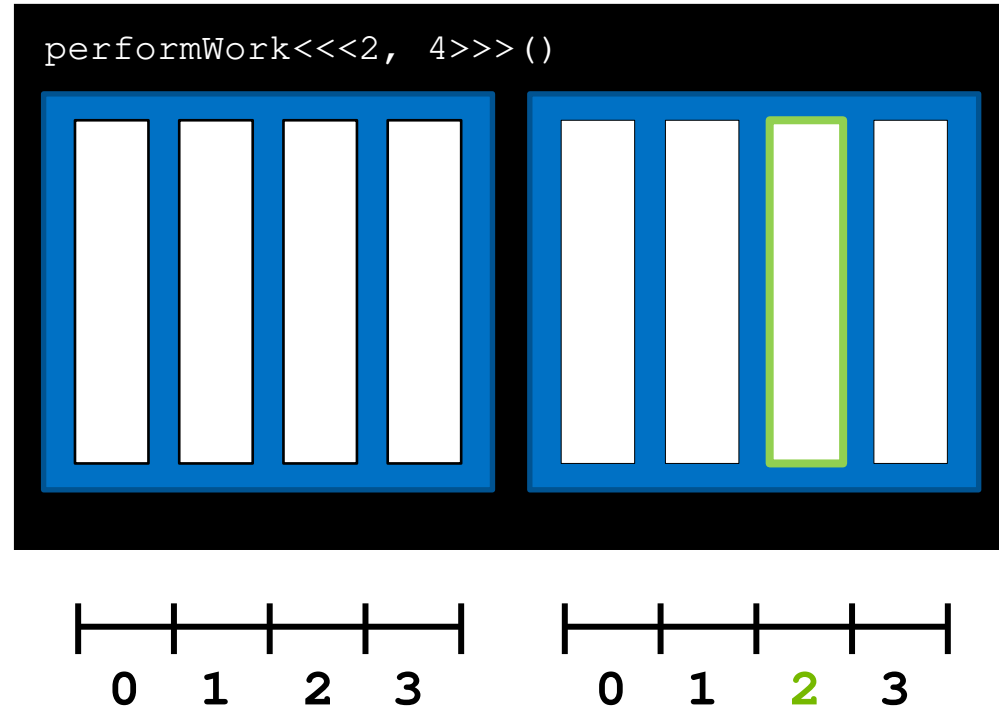
Inside a kernel `threadIdx.x`
describes the index of the thread
within a block. In this case 1

GPU



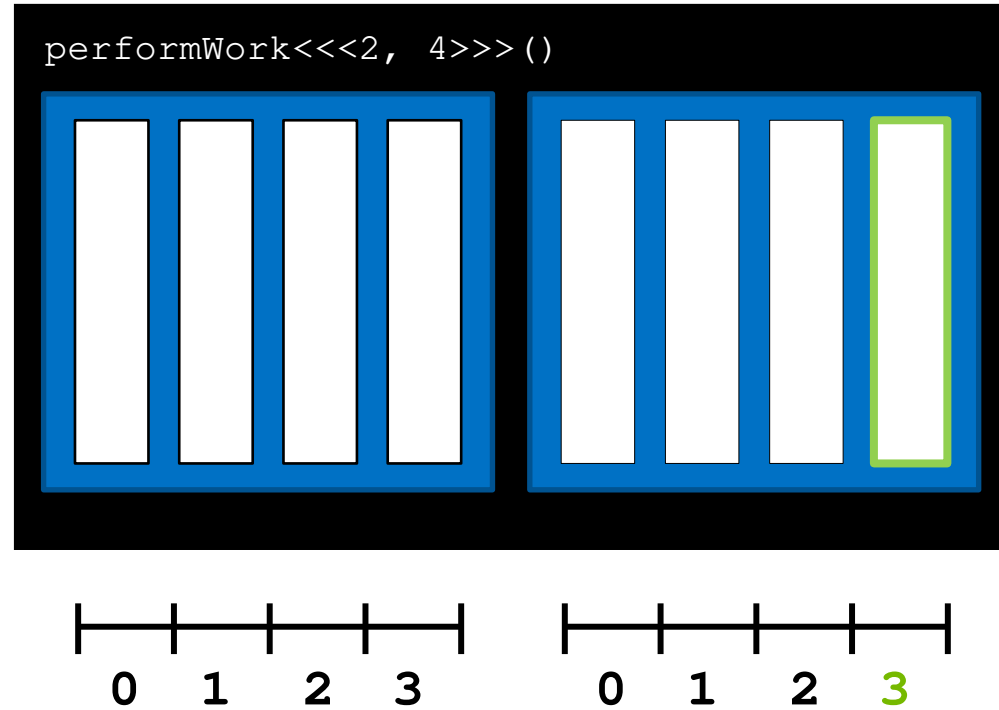
Inside a kernel `threadIdx.x`
describes the index of the thread
within a block. In this case 2

GPU



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

GPU



Coordinating Parallel Threads

DATA

GPU

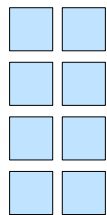
CPU

GPU

CPU

synchronize

Time



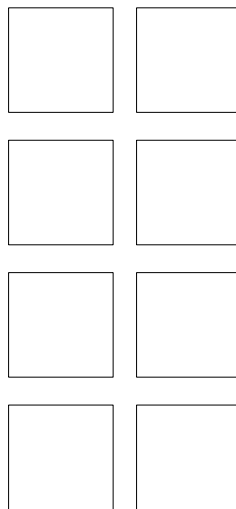
`performWork()`

`initialize()`

`cpuWork()`

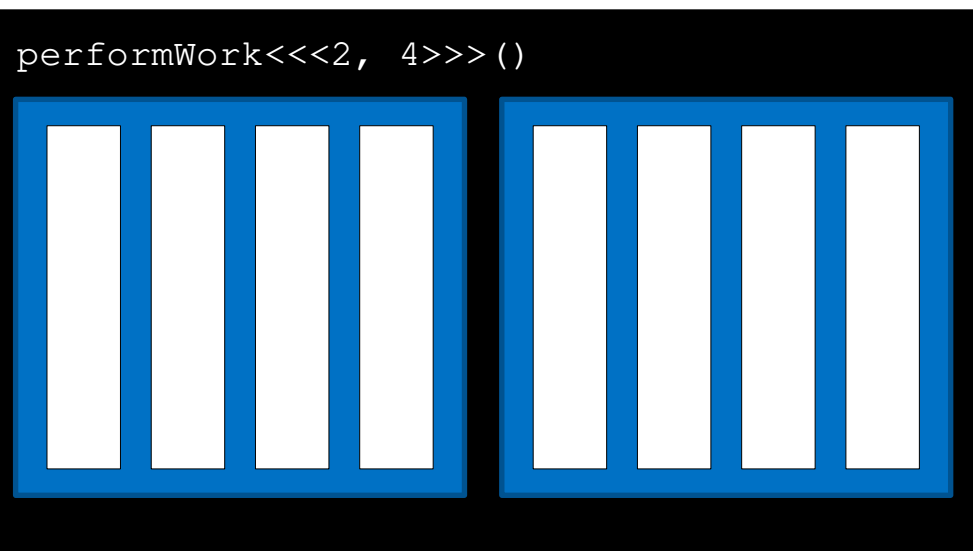
`verifyWork()`

GPU
DATA



Assume data is in a 0 indexed vector

GPU

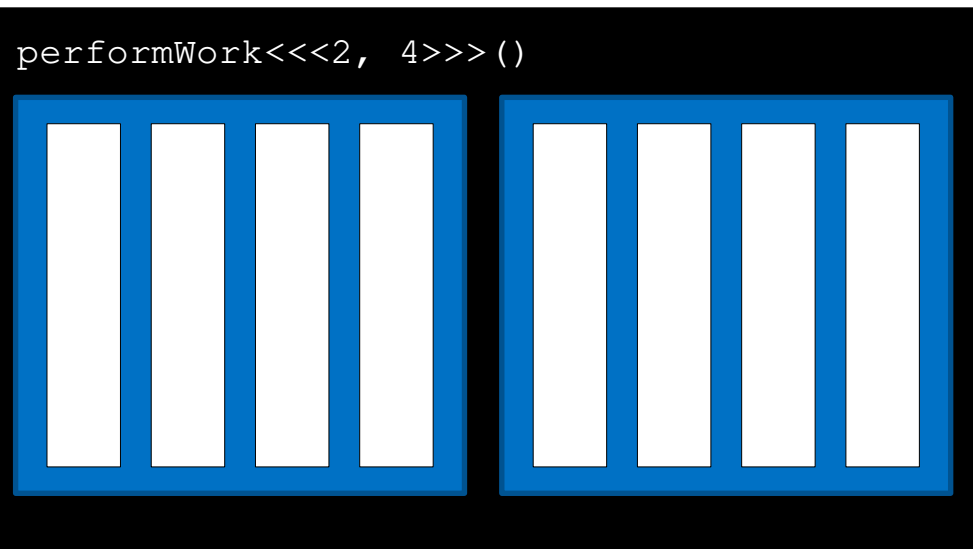


GPU DATA

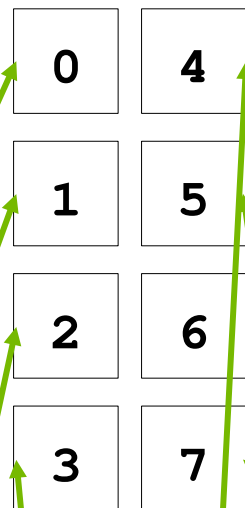
0	4
1	5
2	6
3	7

Assume data is in a 0 indexed vector

GPU

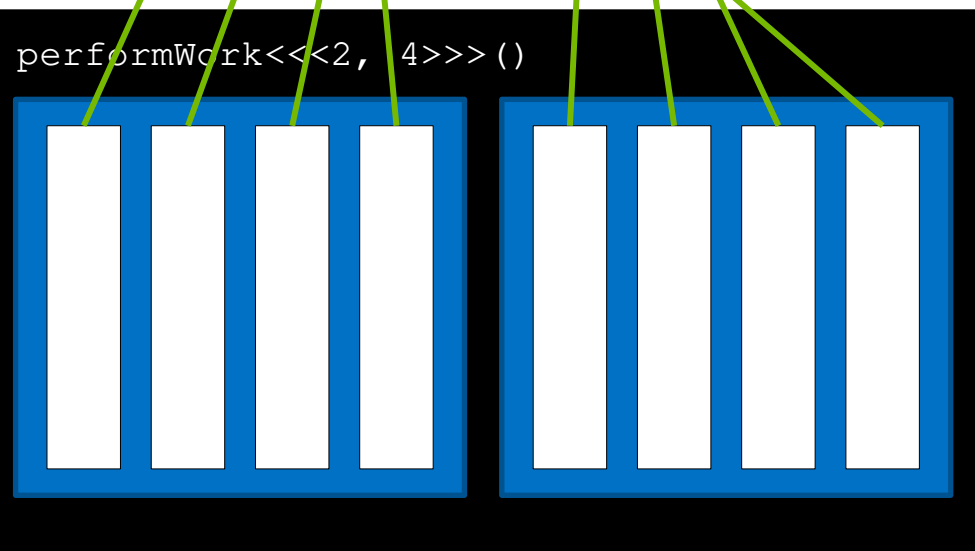


GPU
DATA



Somehow, each thread must be mapped to work on an element in the vector

GPU

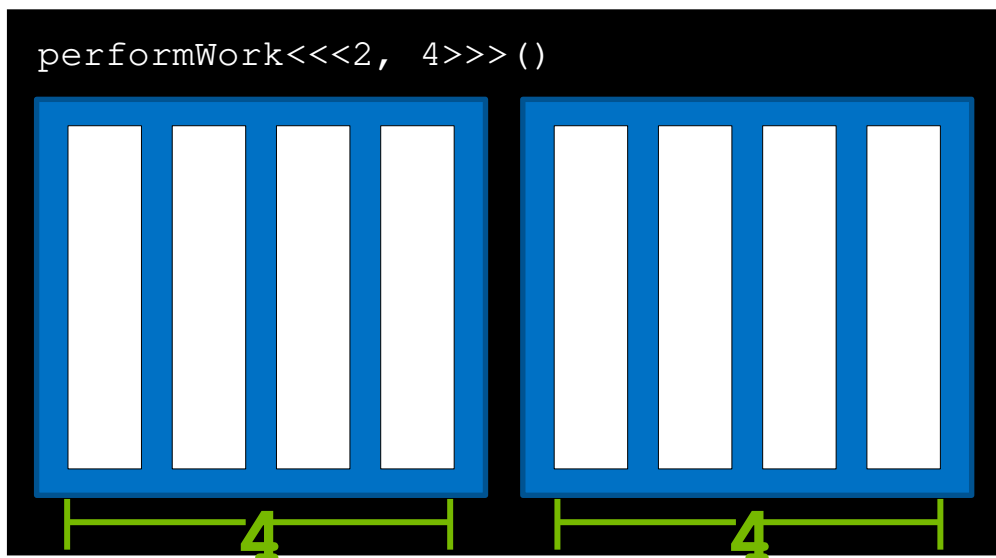


GPU DATA

0	4
1	5
2	6
3	7

Recall that each thread has access to the size of its block via `blockDim.x`

GPU

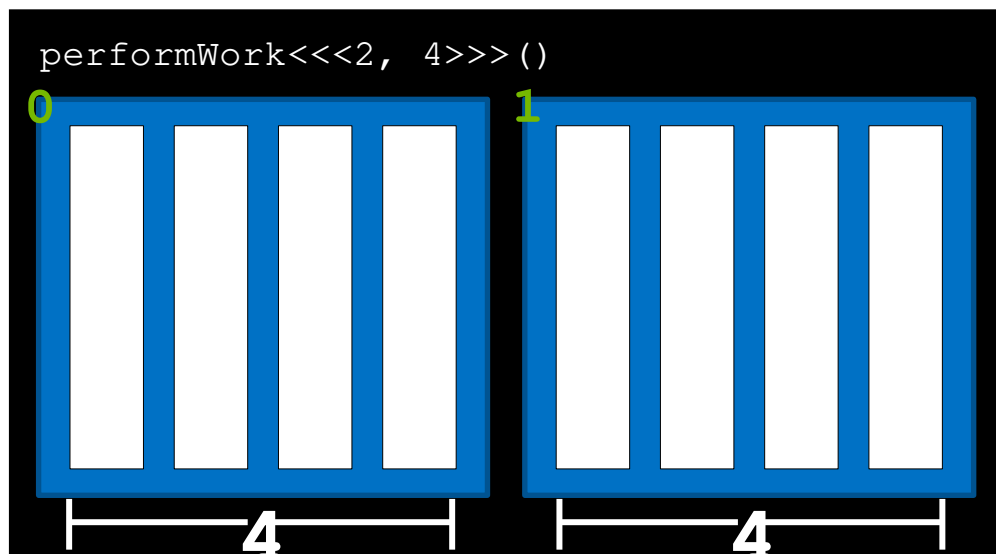


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

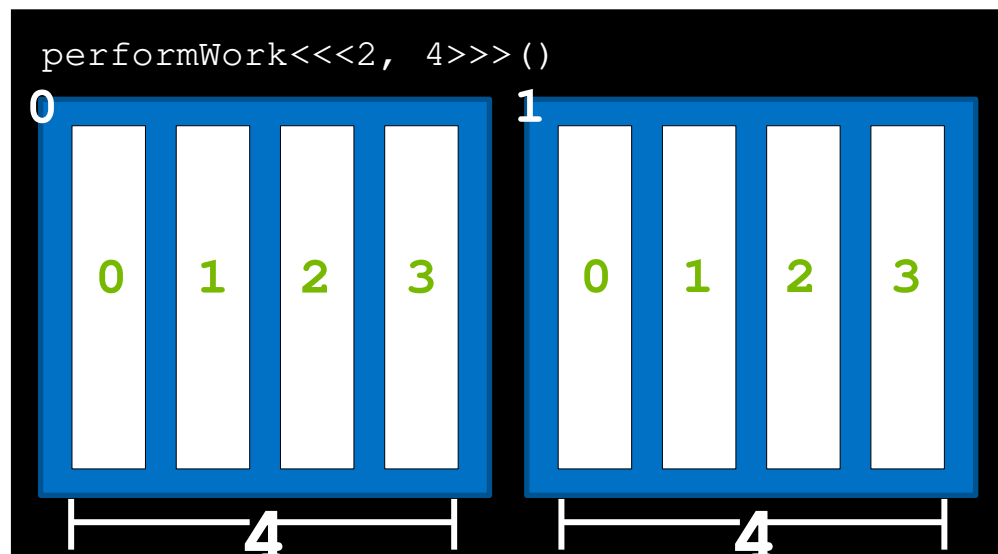


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

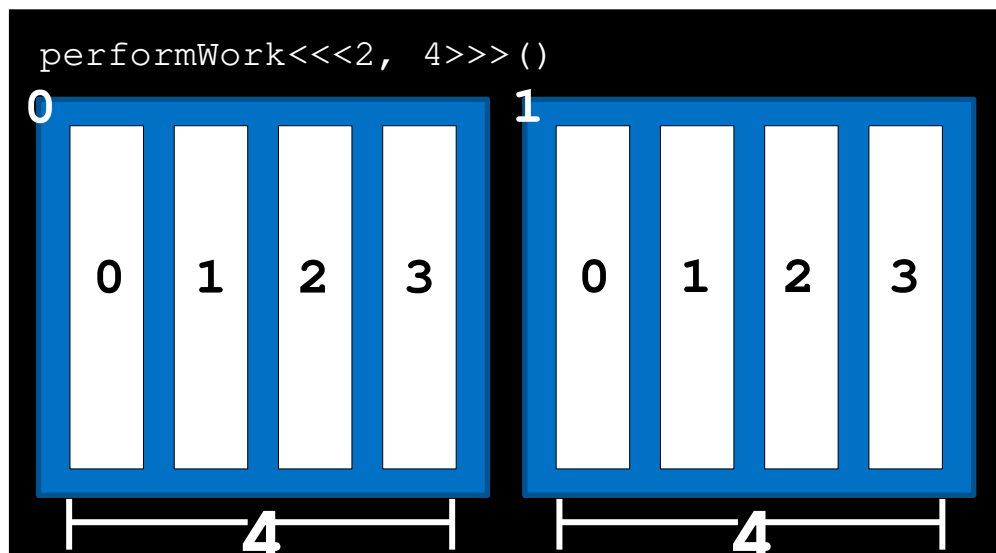


GPU DATA

0	4
1	5
2	6
3	7

Using these variables, the formula **threadIdx.x + blockIdx.x * blockDim.x** will map each thread to one element in the vector

GPU

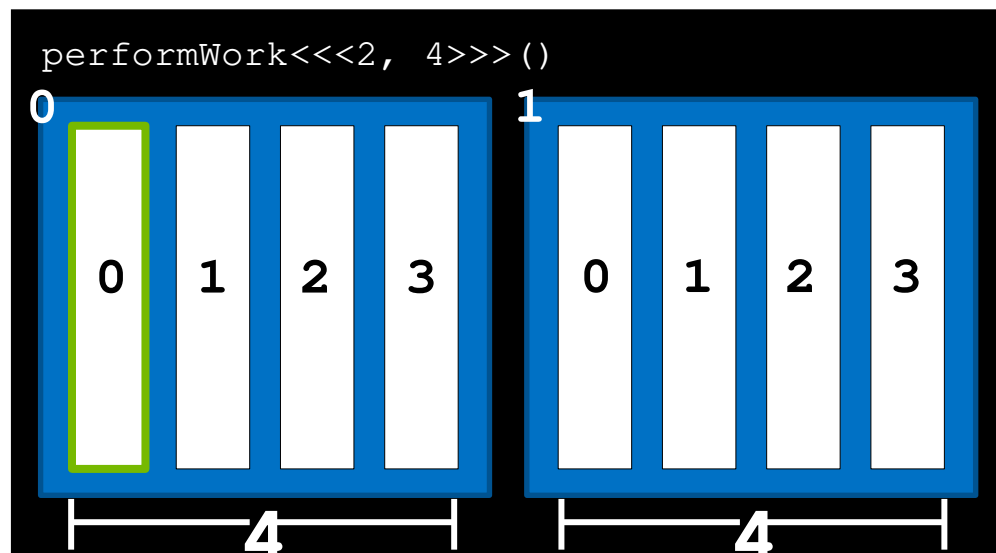


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

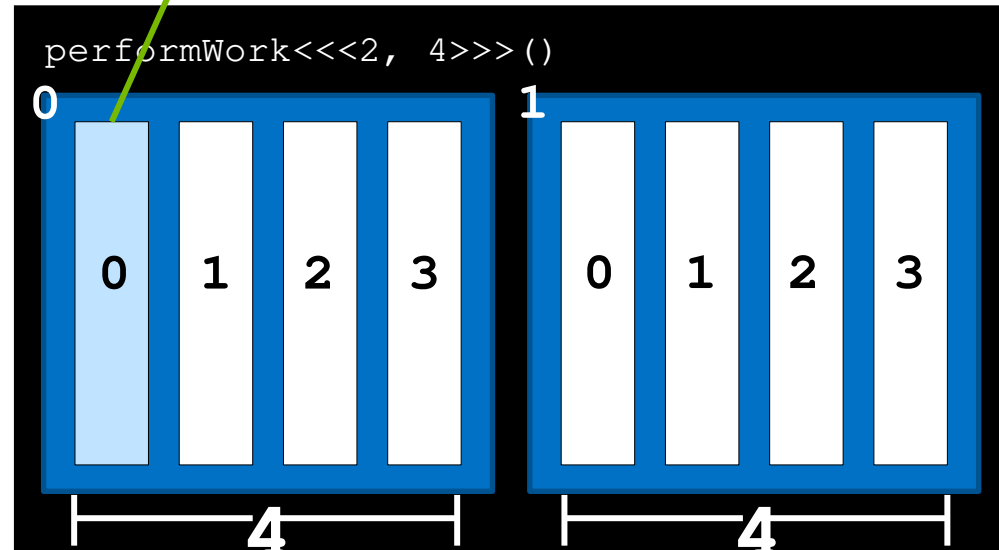


GPU
DATA

0	4
1	5
2	6
3	7

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

GPU

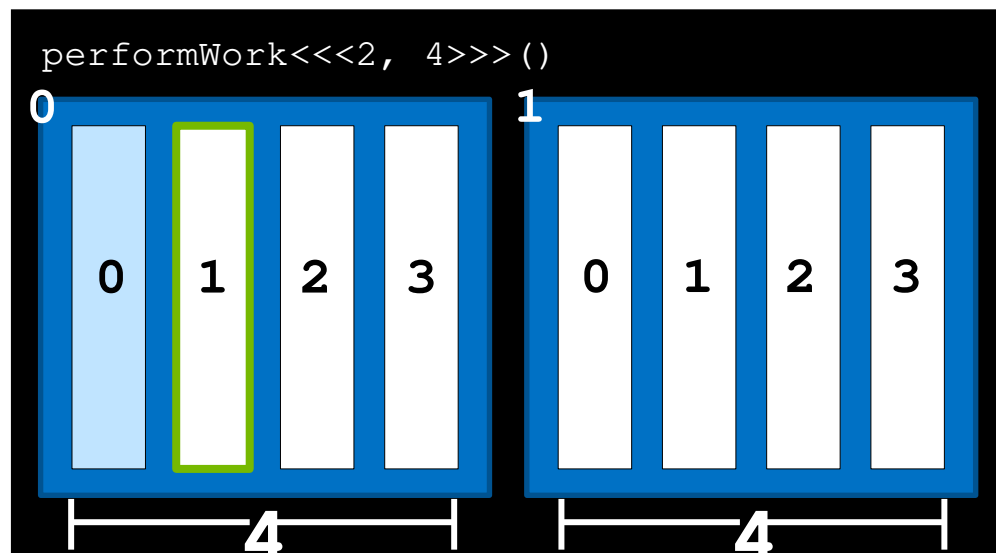


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

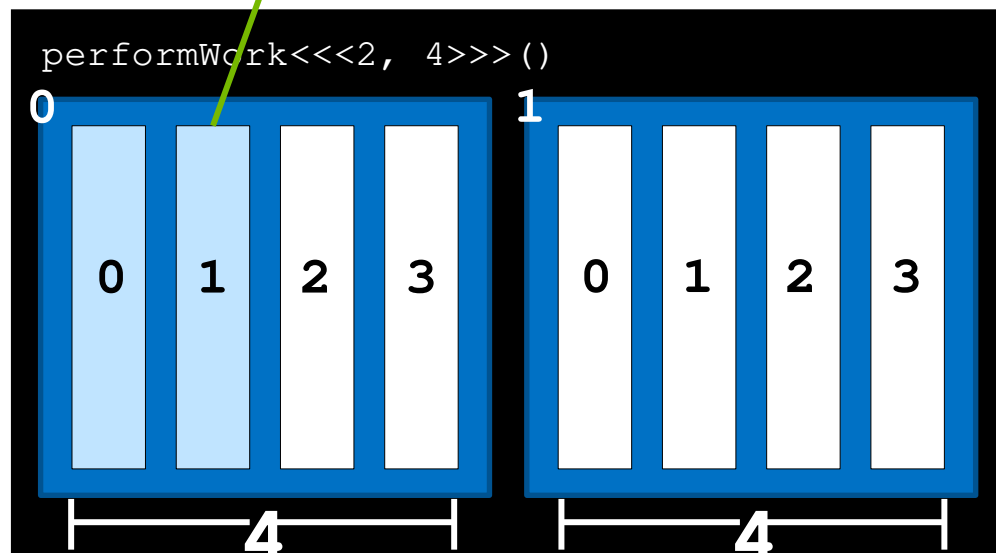


GPU
DATA

0	4
1	5
2	6
3	7

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

GPU

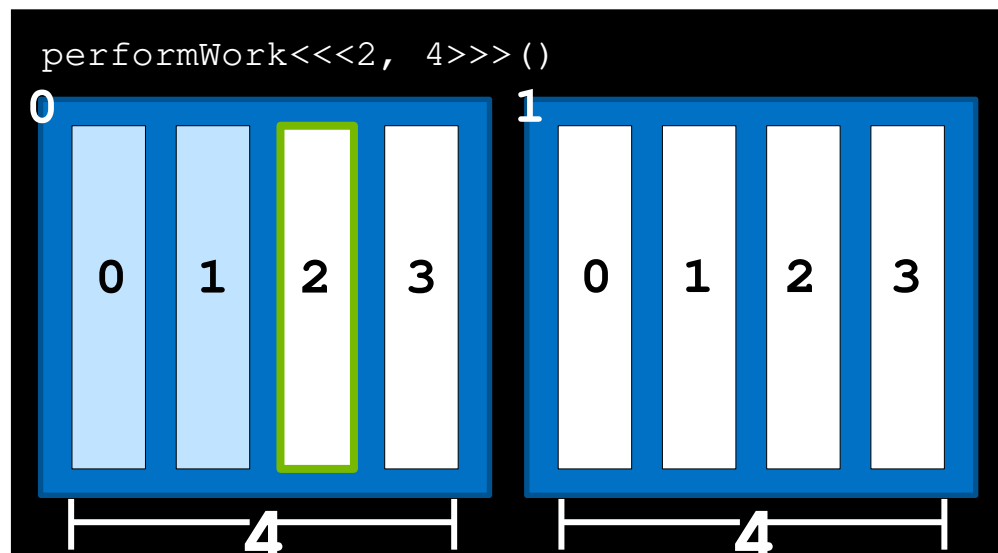


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

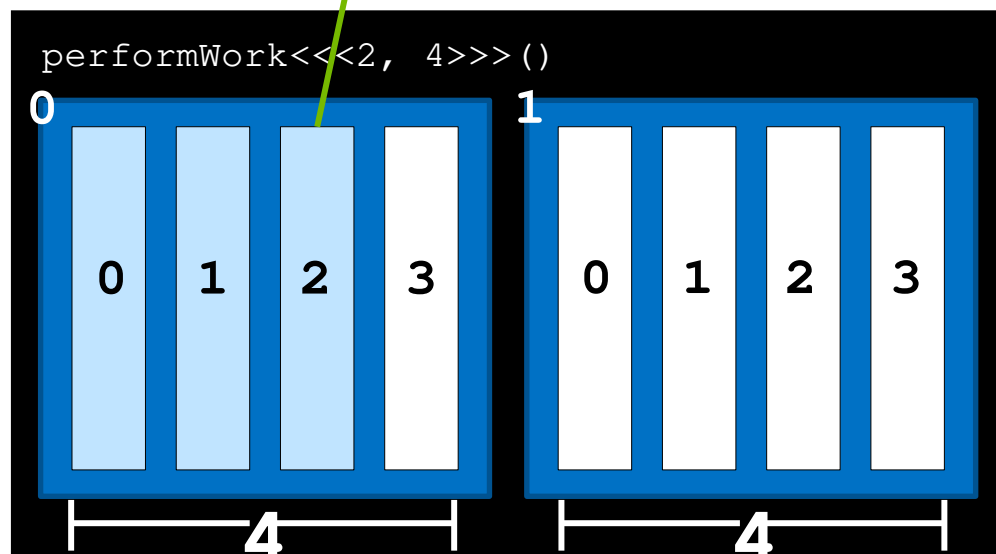


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

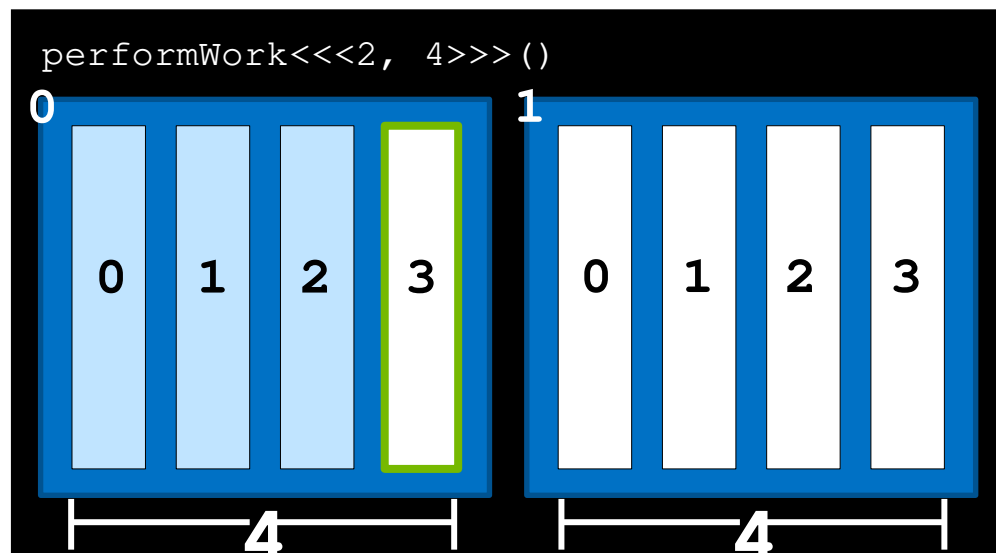


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

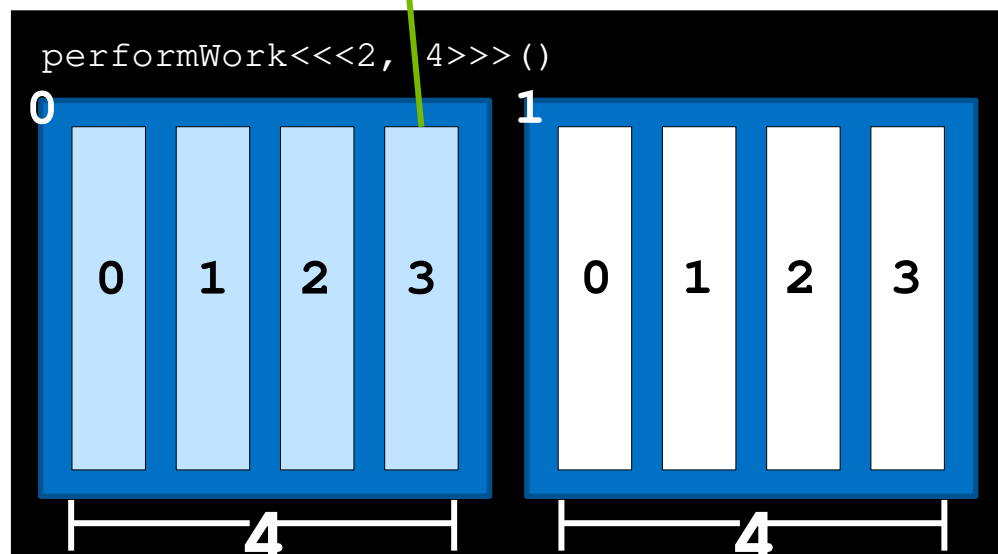


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

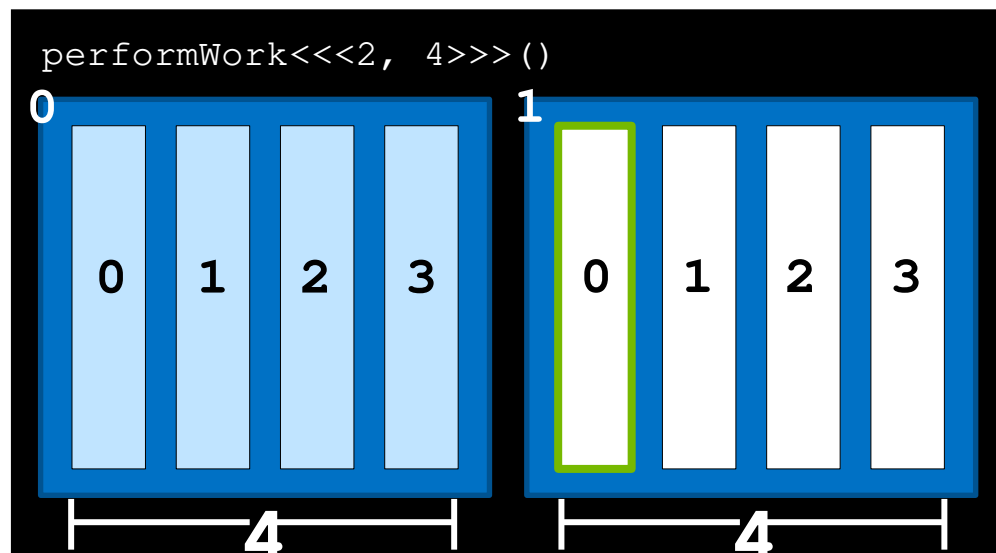


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

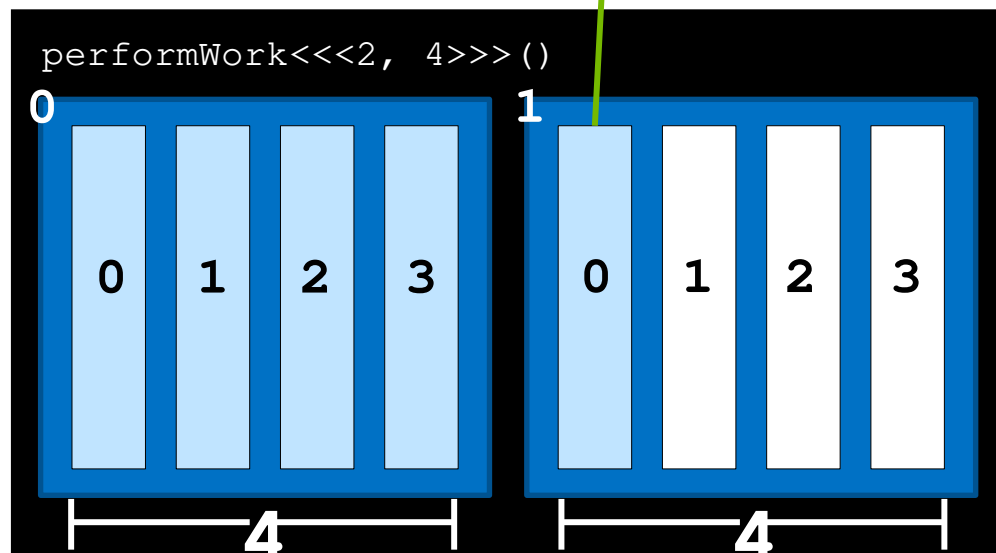


GPU
DATA

0	4
1	5
2	6
3	7

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

GPU

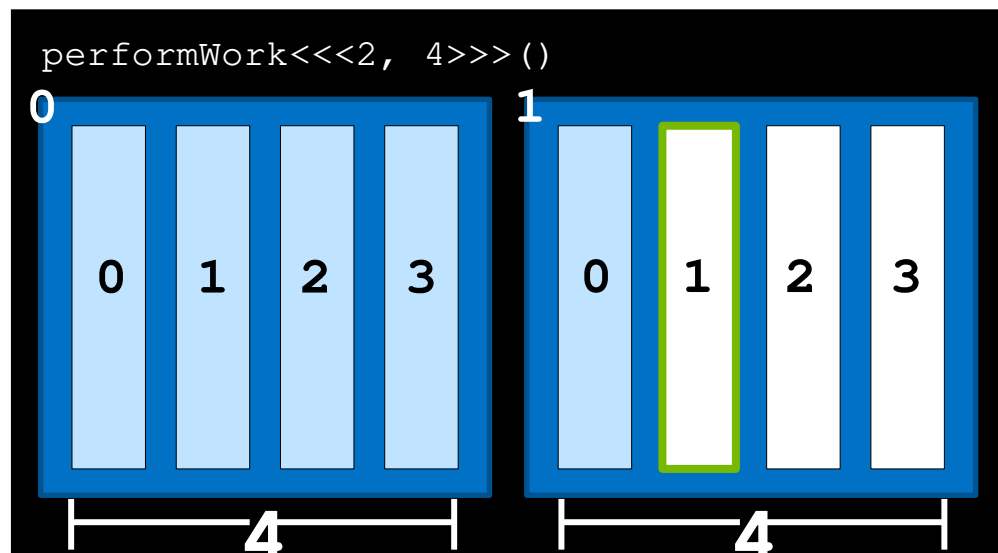


GPU DATA

0	4
1	5
2	6
3	7

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

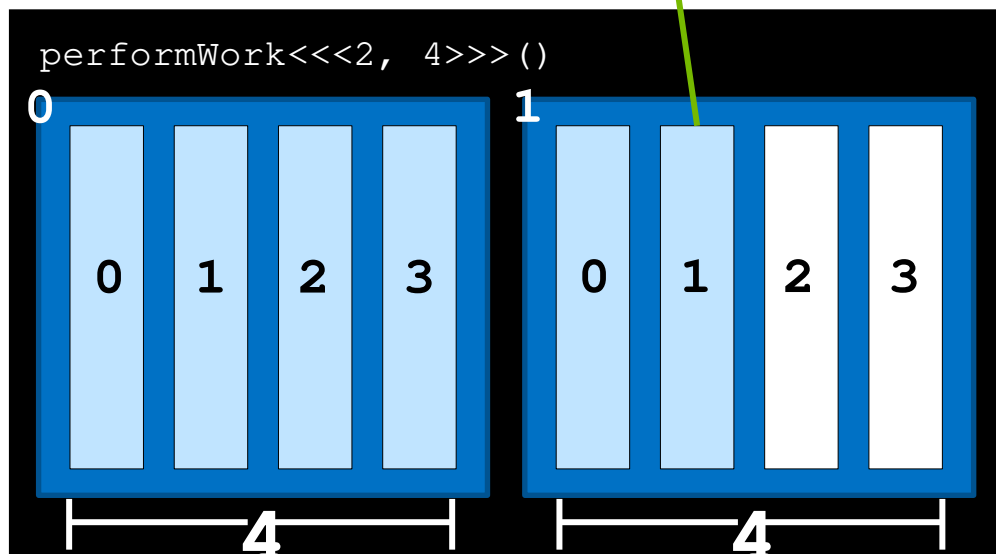
GPU



GPU DATA

0	4	threadIdx.x	+	blockIdx.x	*	blockDim.x
1	5	1		1		4
2	6	dataIndex				
3	7	5				

GPU

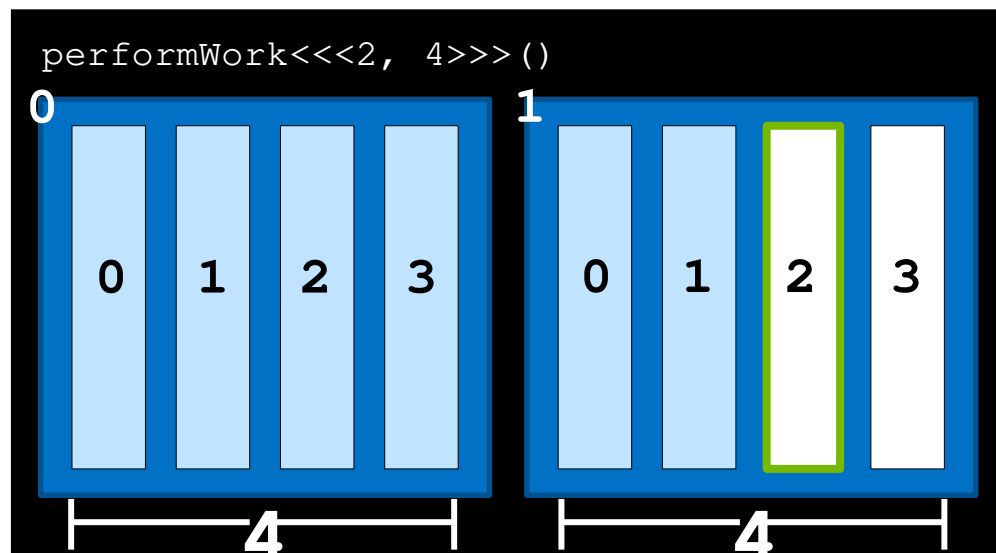


GPU DATA

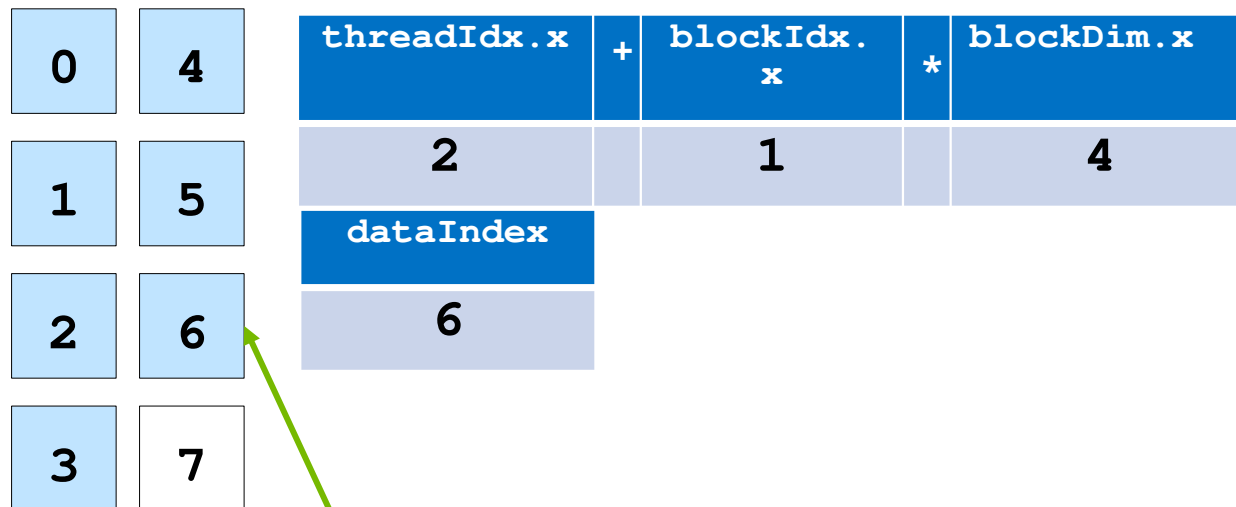
0	4
1	5
2	6
3	7

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

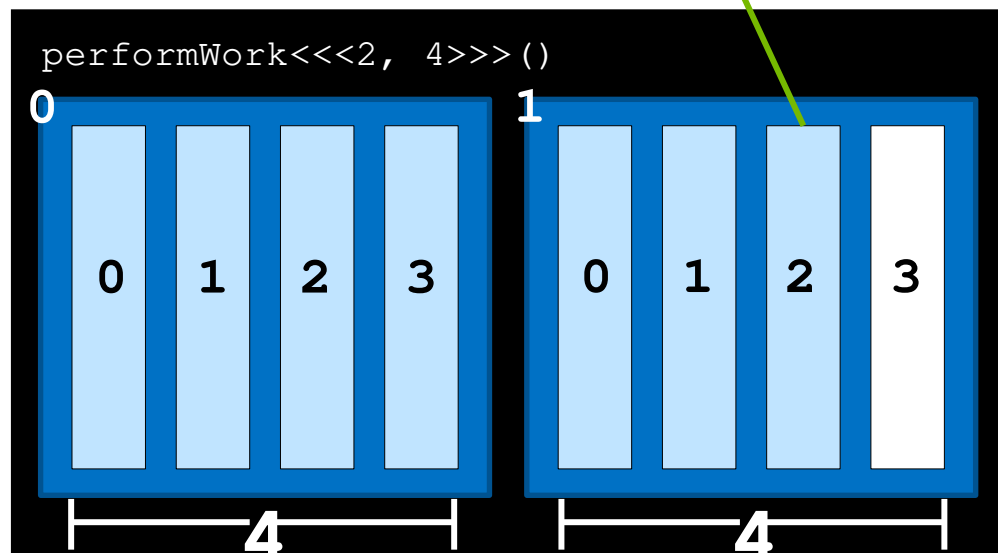
GPU



GPU DATA



GPU

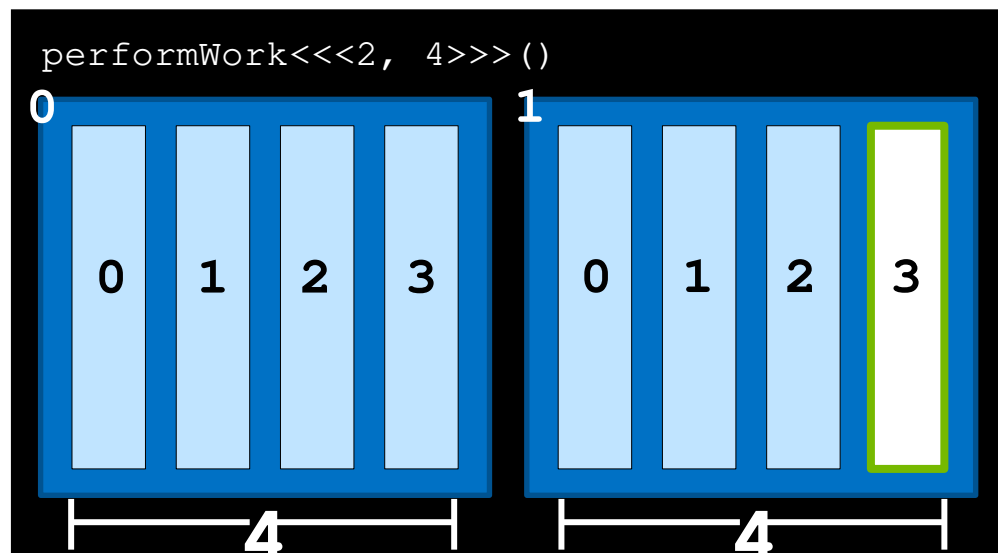


GPU DATA

0	4
1	5
2	6
3	7

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

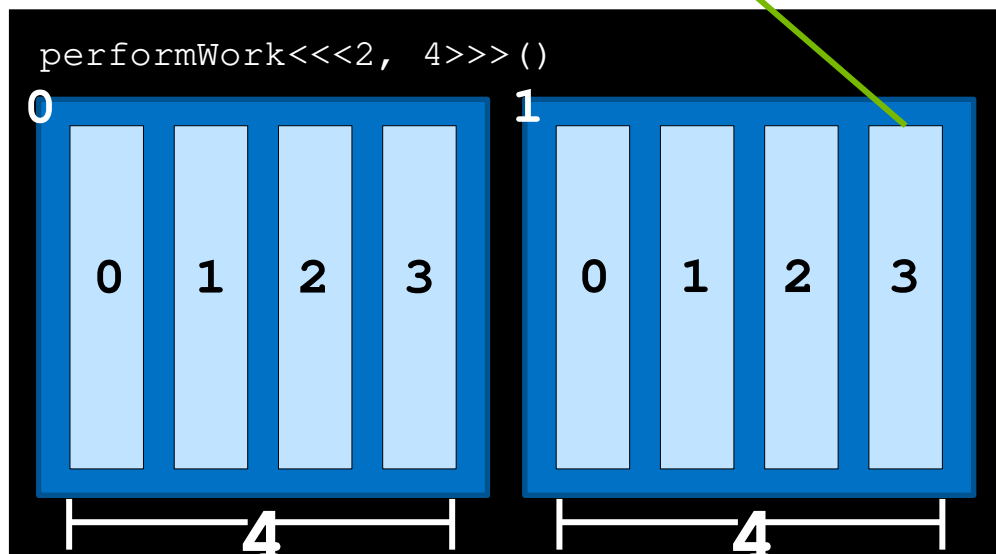
GPU



GPU DATA

0	4	threadIdx.x	+	blockIdx.x	*	blockDim.x
1	5	3		1		4
2	6	dataIndex				
3	7	7				

GPU

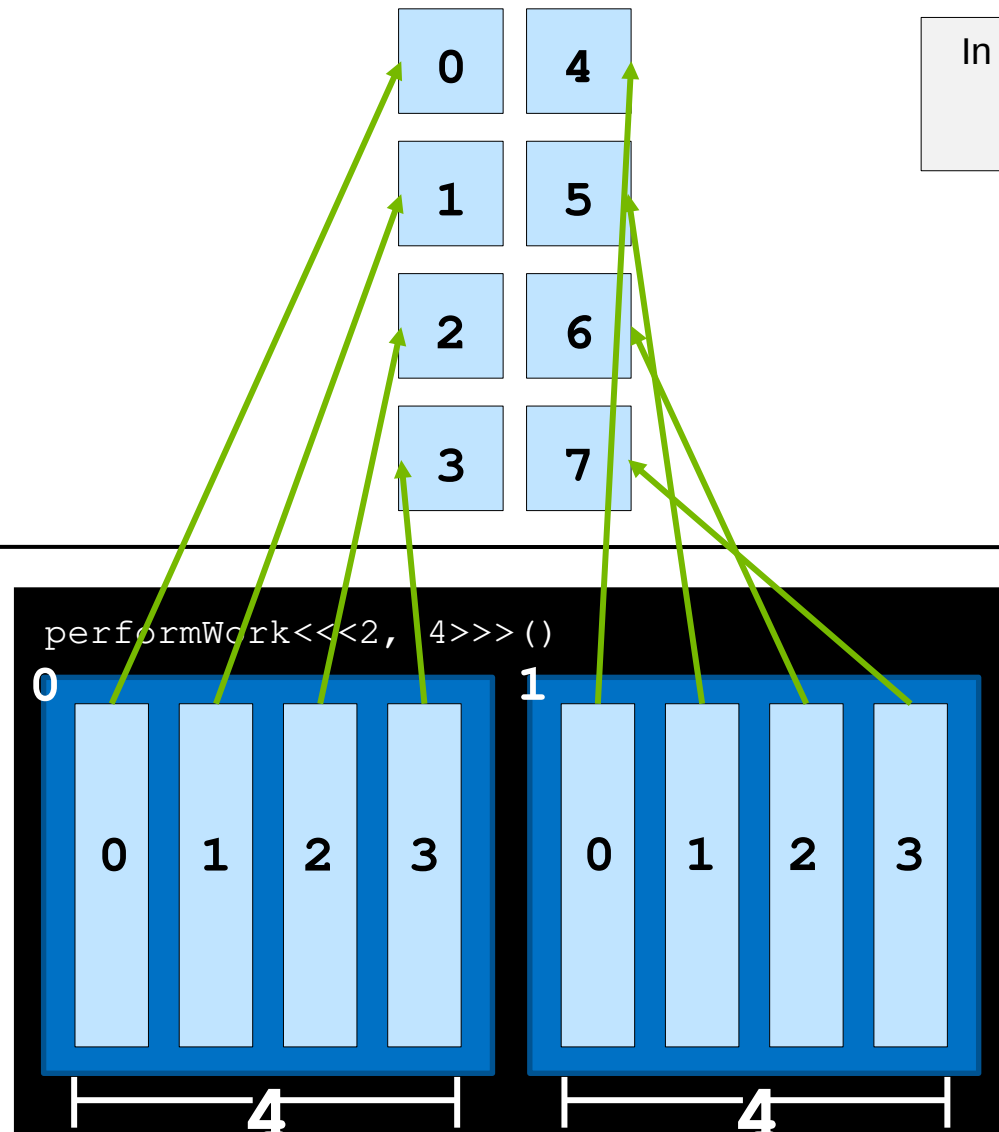


Grid Size Work Amount Mismatch

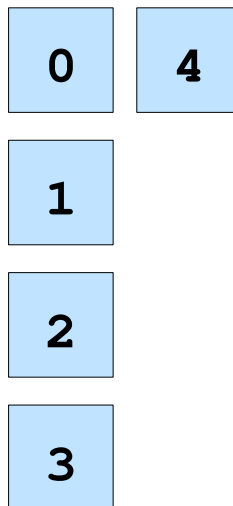
GPU
DATA

In previous scenarios, the number of threads in the grid matched the number of elements exactly

GPU

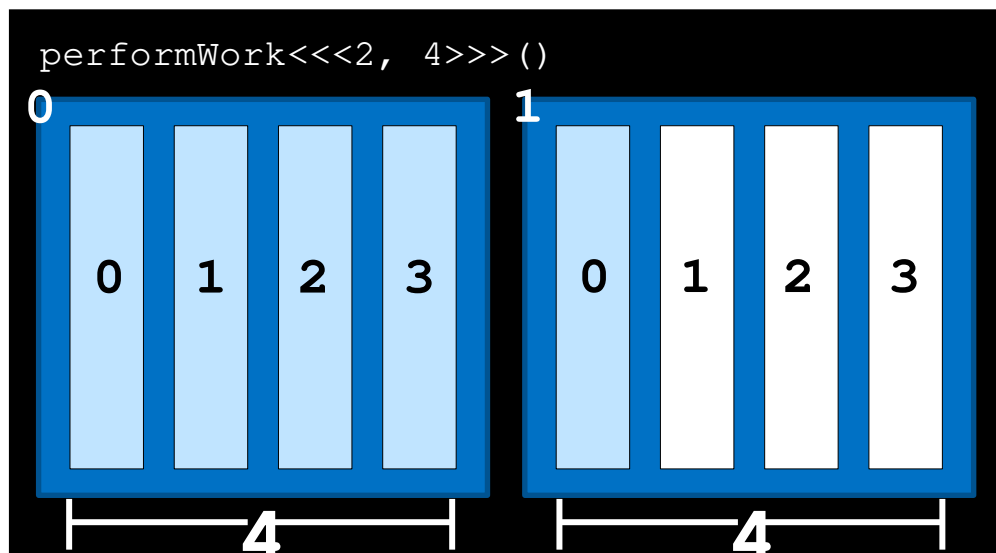


GPU DATA

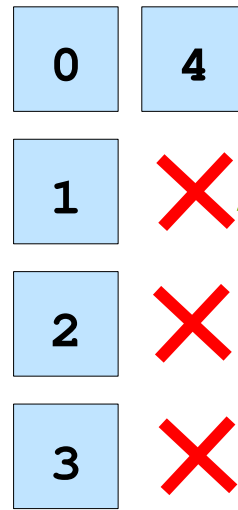


What if there are more threads than work to be done?

GPU

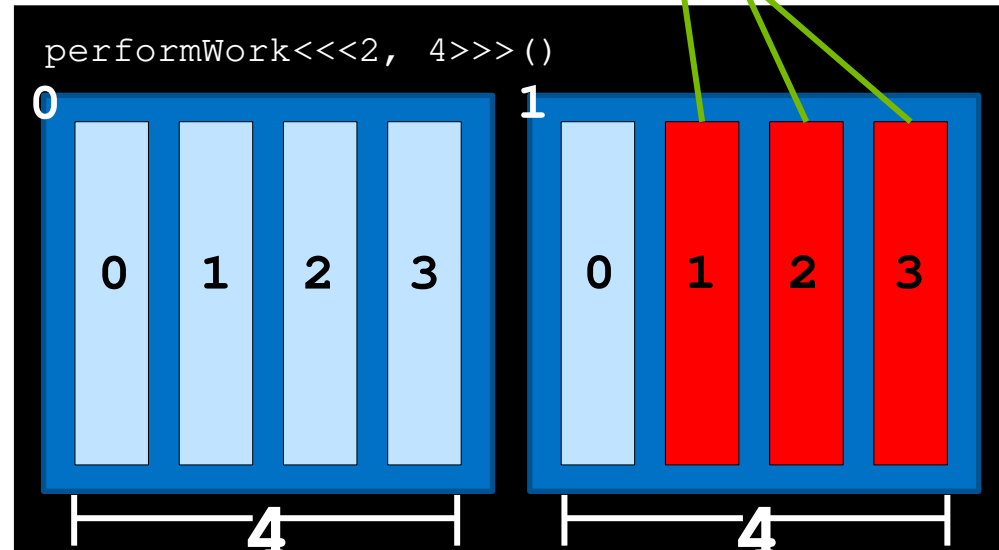


GPU DATA



Attempting to access non-existent elements can result in a runtime error

GPU

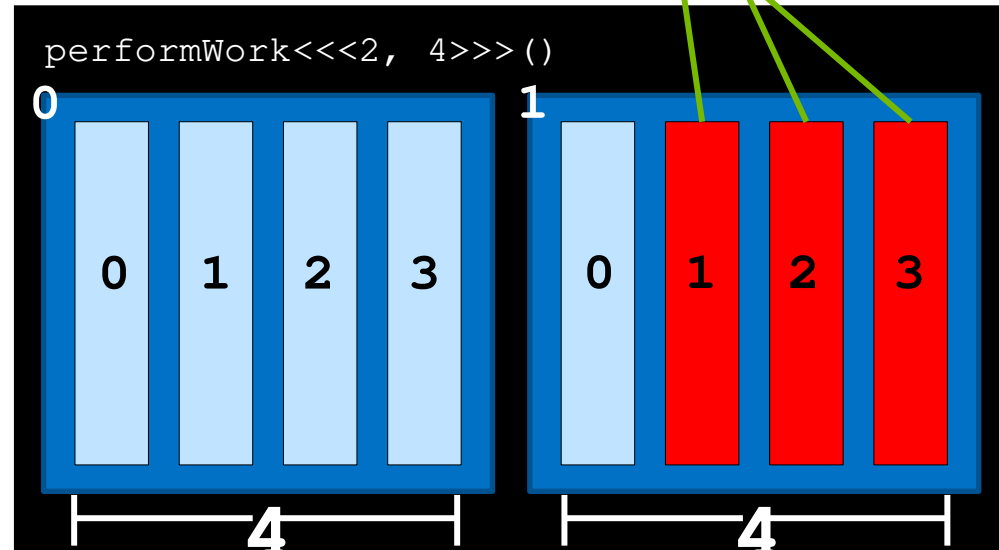


GPU DATA

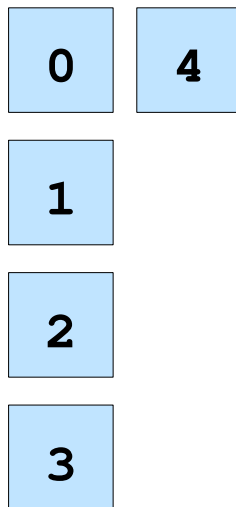
0	4
1	×
2	×
3	×

Code must check that the **dataIndex** calculated by **threadIdx.x + blockIdx.x * blockDim.x** is less than **N**, the number of data elements.

GPU

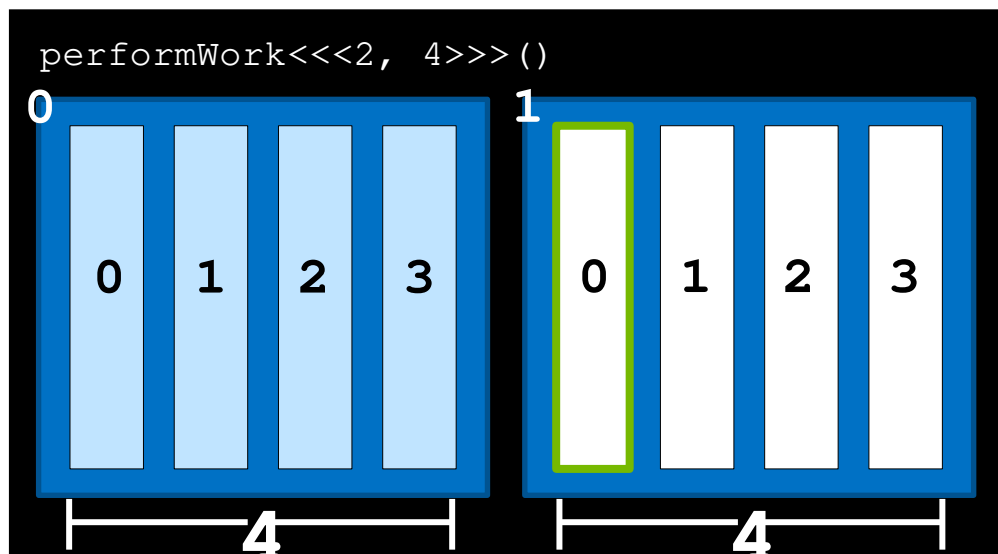


GPU DATA

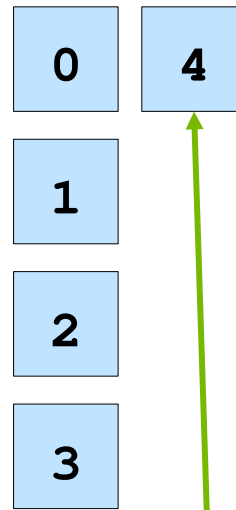


threadIdx.x	+	blockIdx.x	*	blockDim.x
0		1		4
dataIndex	<	N	=	Can work
4		5		?

GPU

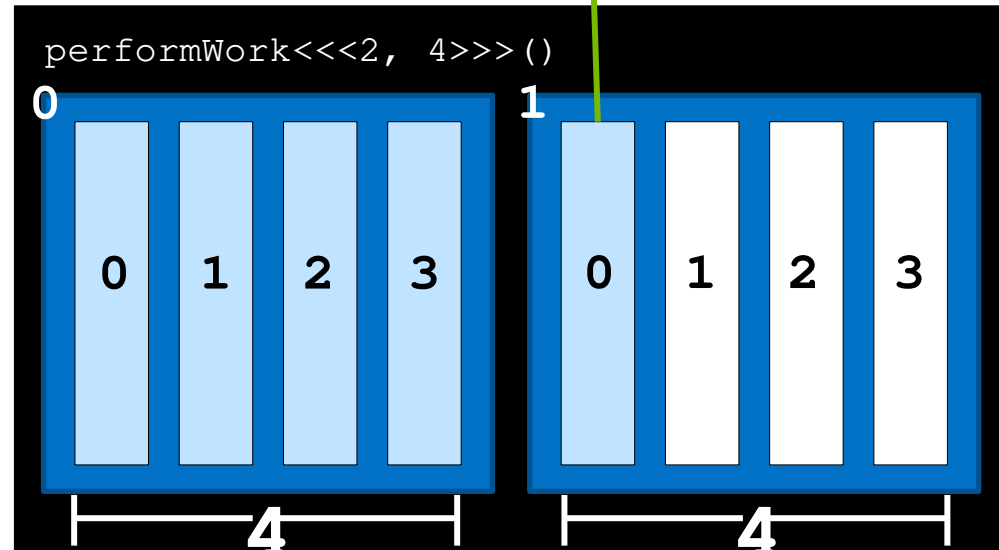


GPU
DATA

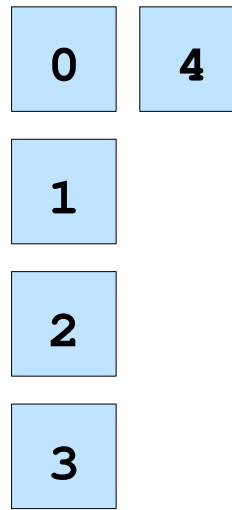


threadIdx.x	+	blockIdx.x	*	blockDim.x
0		1		4
dataIndex	<	N	=	Can work
4		5		true

GPU

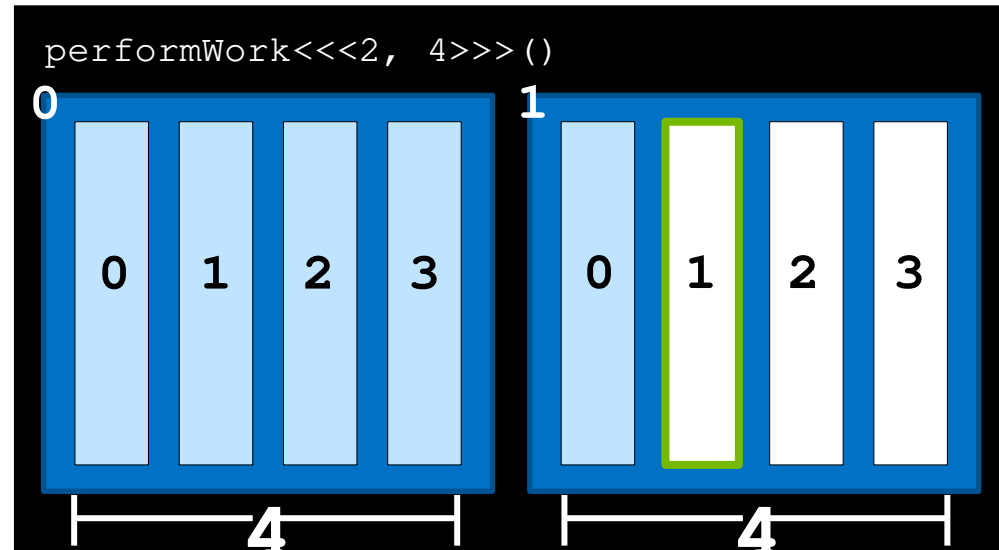


GPU DATA

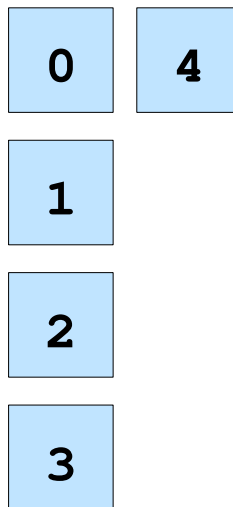


threadIdx.x	+	blockIdx.x	*	blockDim.x
1		1		4
dataIndex	<	N	=	Can work
5		5		?

GPU

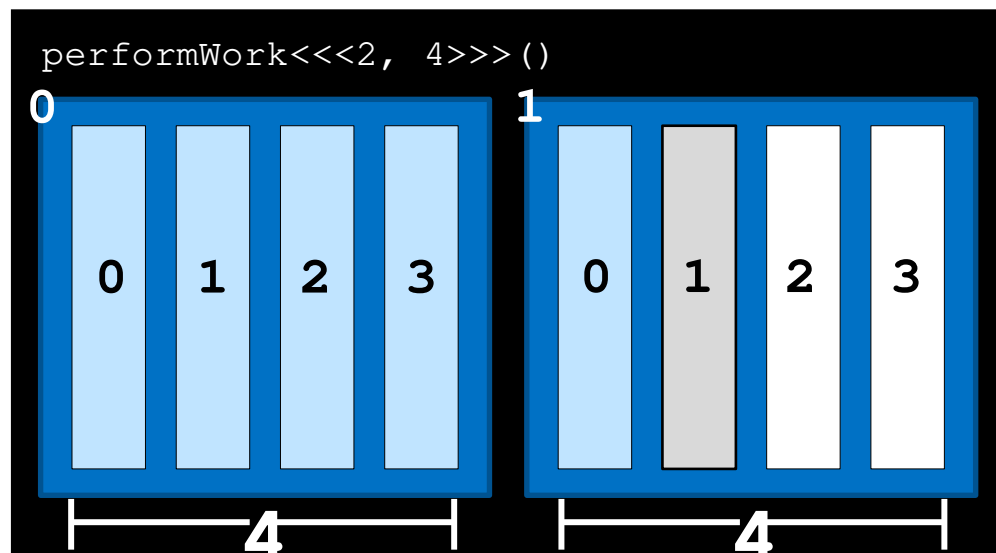


GPU DATA

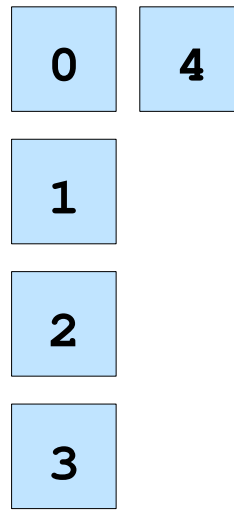


threadIdx.x	+	blockIdx.x	*	blockDim.x
1		1		4
dataIndex	<	N	=	Can work
5		5		false

GPU

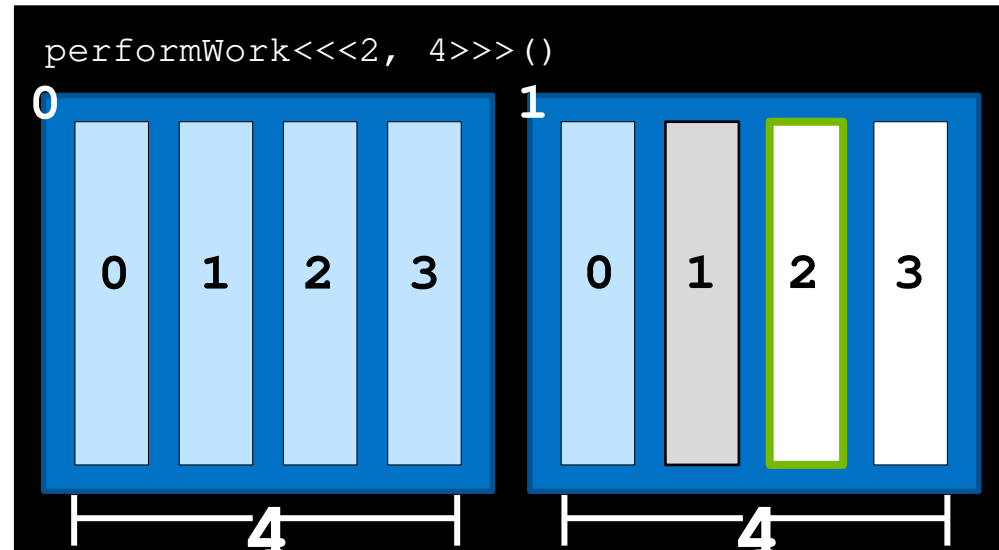


GPU DATA

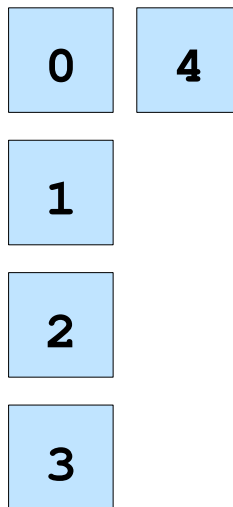


threadIdx.x	+	blockIdx.x	*	blockDim.x
2		1		4
dataIndex	<	N	=	Can work
6		5		?

GPU

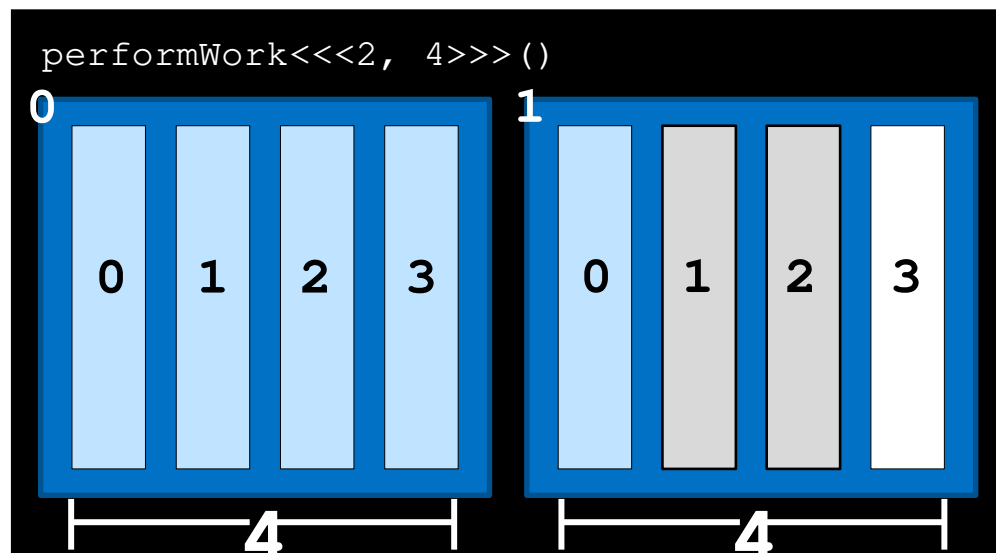


GPU DATA

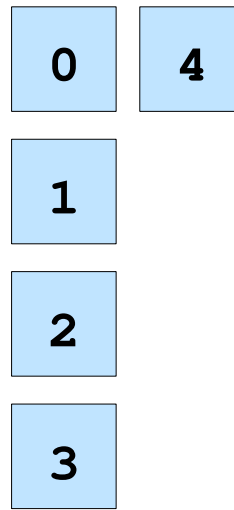


threadIdx.x	+	blockIdx.x	*	blockDim.x
2		1		4
dataIndex	<	N	=	Can work
6		5		false

GPU

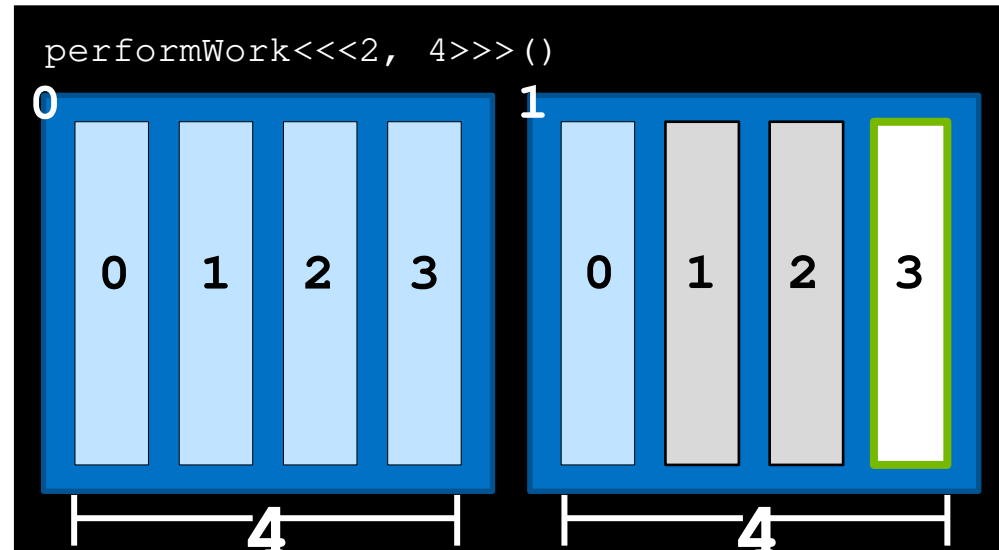


GPU DATA

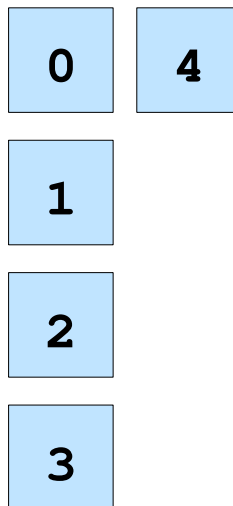


threadIdx.x	+	blockIdx.x	*	blockDim.x
2		1		4
dataIndex	<	N	=	Can work
6		5		?

GPU

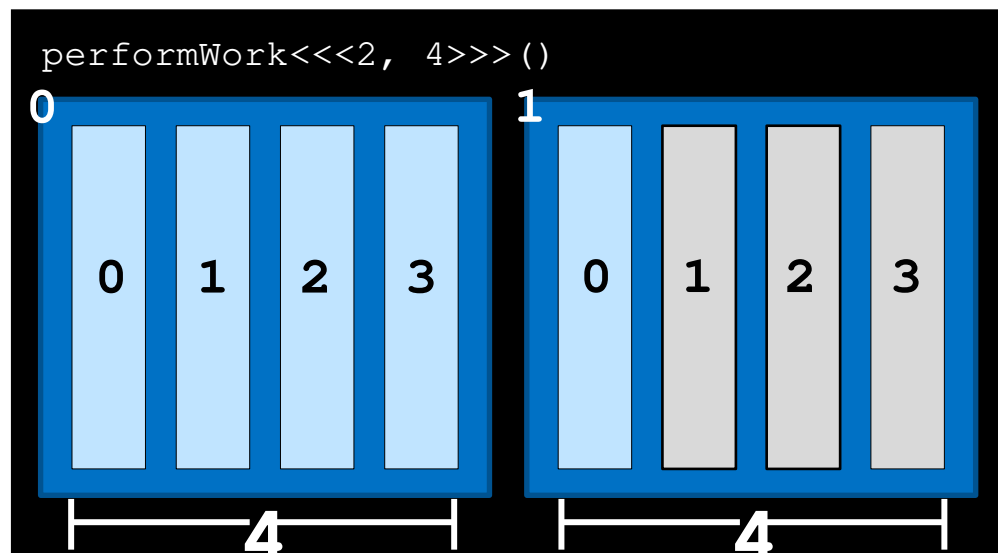


GPU DATA



threadIdx.x	+	blockIdx.x	*	blockDim.x
2		1		4
dataIndex	<	N	=	Can work
6		5		false

GPU



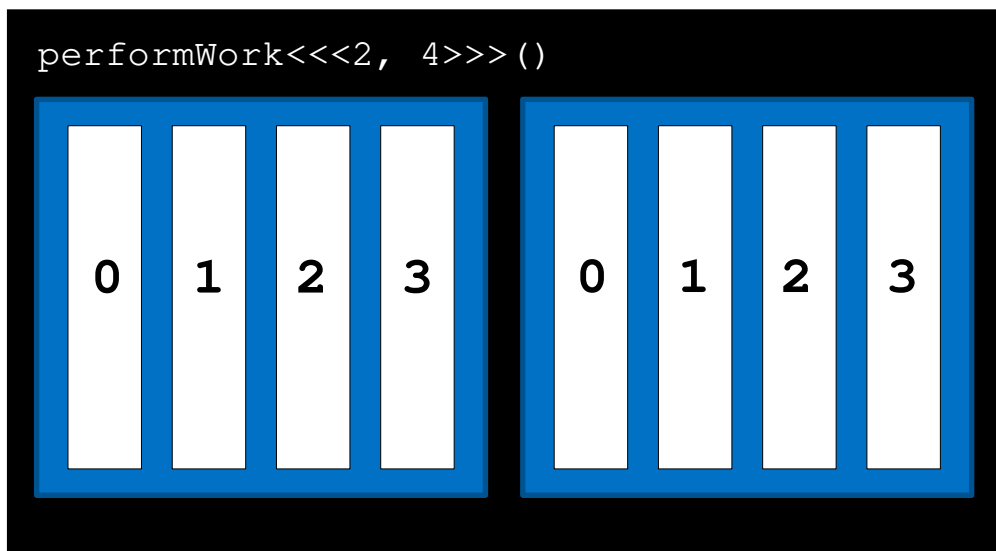
Grid-Stride Loops

GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

Often there are more data elements than there are threads in the grid

GPU

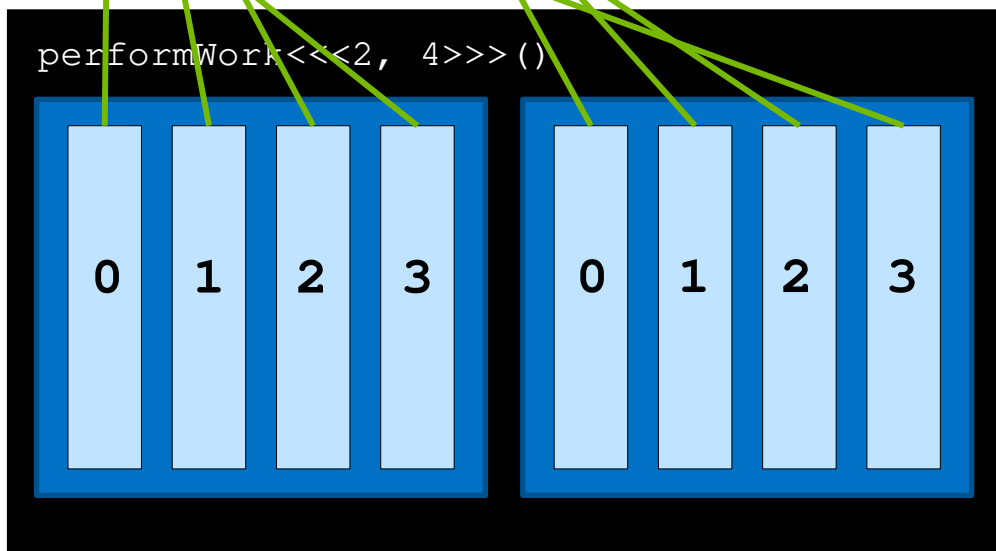


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

In such scenarios threads cannot work on only one element

GPU



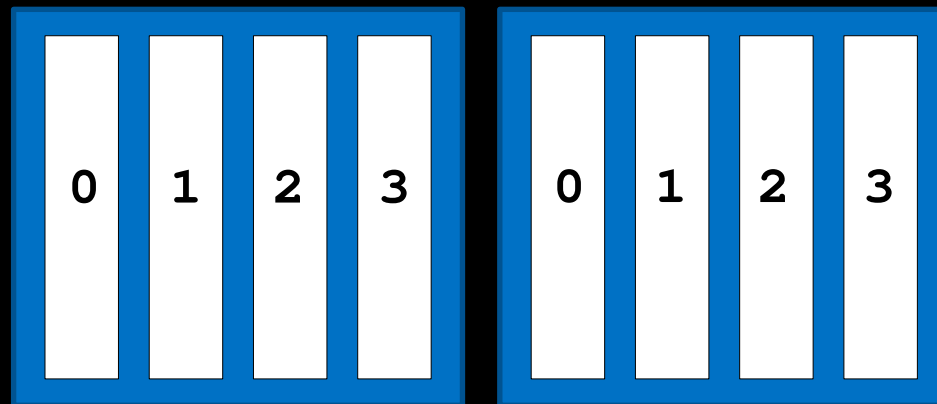
GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

... or else work is left
undone

GPU

```
performWork<<<2, 4>>>()
```

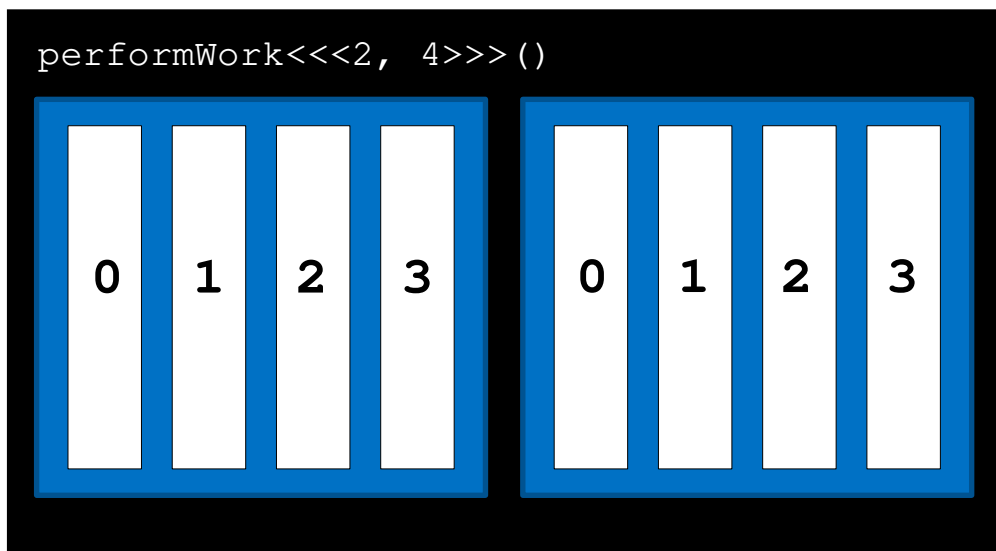


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

One way to address this programmatically is with a **grid-stride loop**

GPU



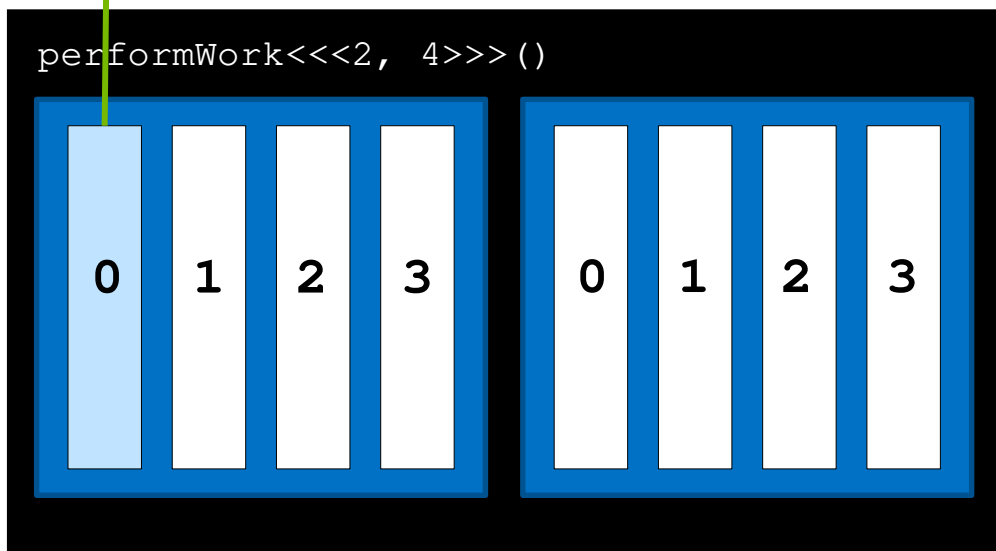
GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

In a grid-stride loop, the thread's first element is calculated as usual, with

$$\text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}$$

GPU

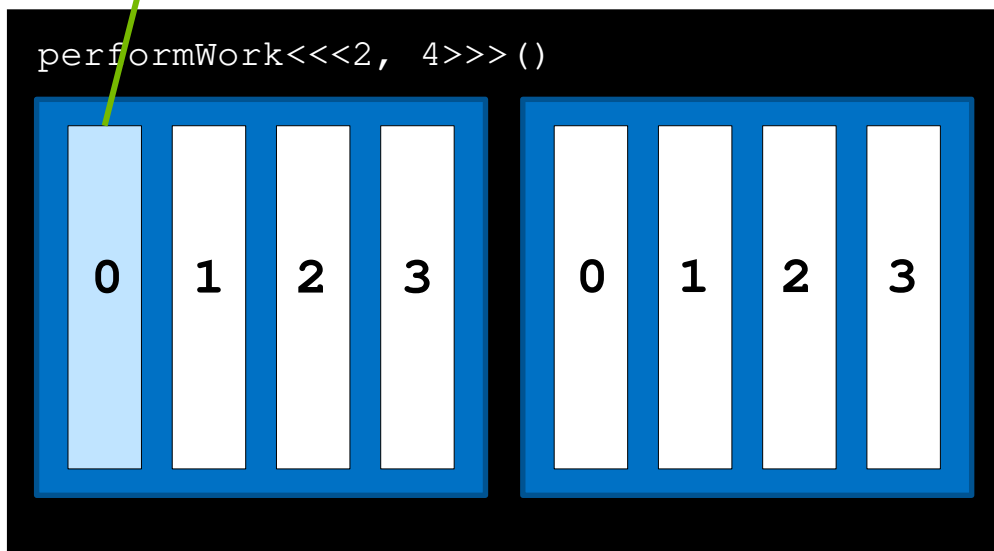


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

The thread then strides forward by the number of threads in the grid
(`blockDim.x * blockDim.y`), in this case
8

GPU

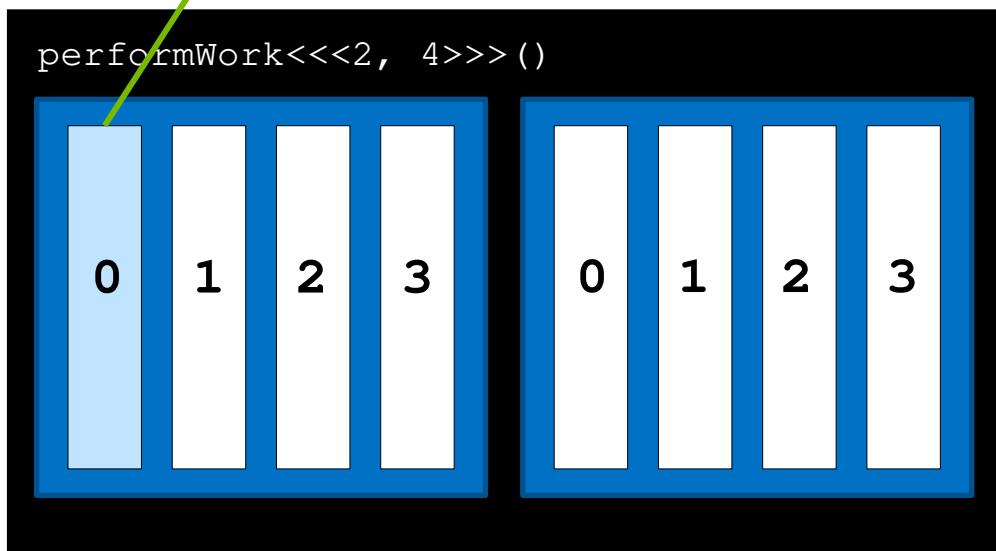


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

It continues in this way until its data index is greater than the number of data elements

GPU

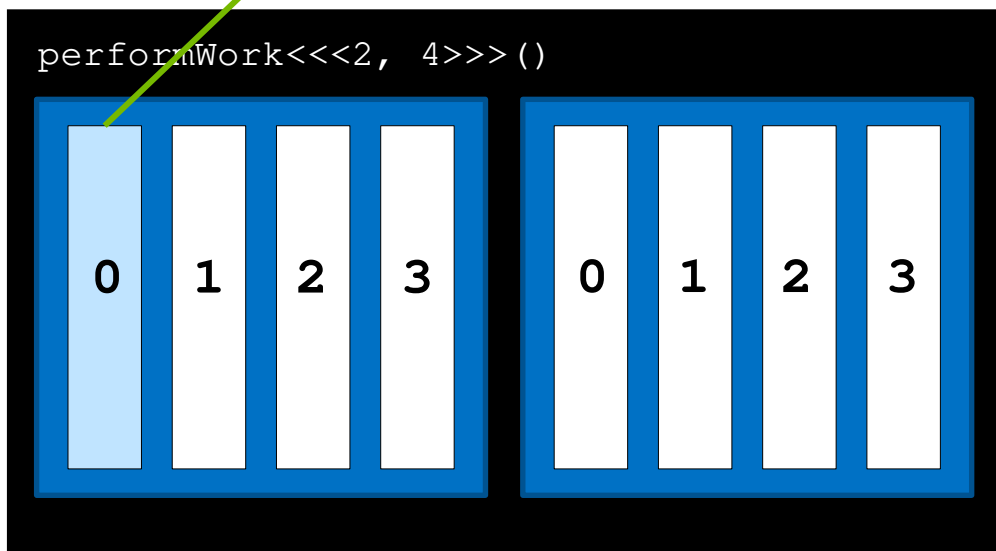


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

It continues in this way until its data index is greater than the number of data elements

GPU

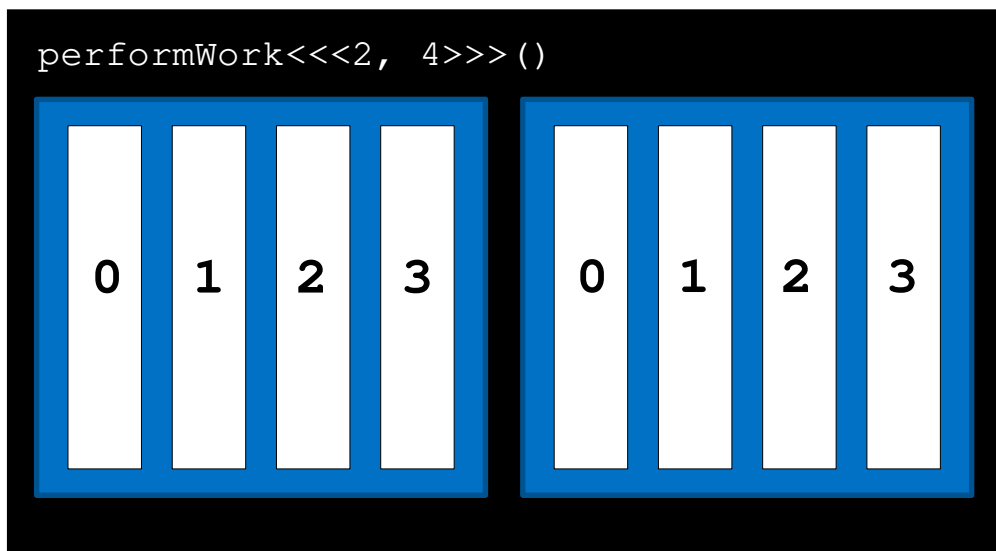


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads working in this way, all elements are covered

GPU

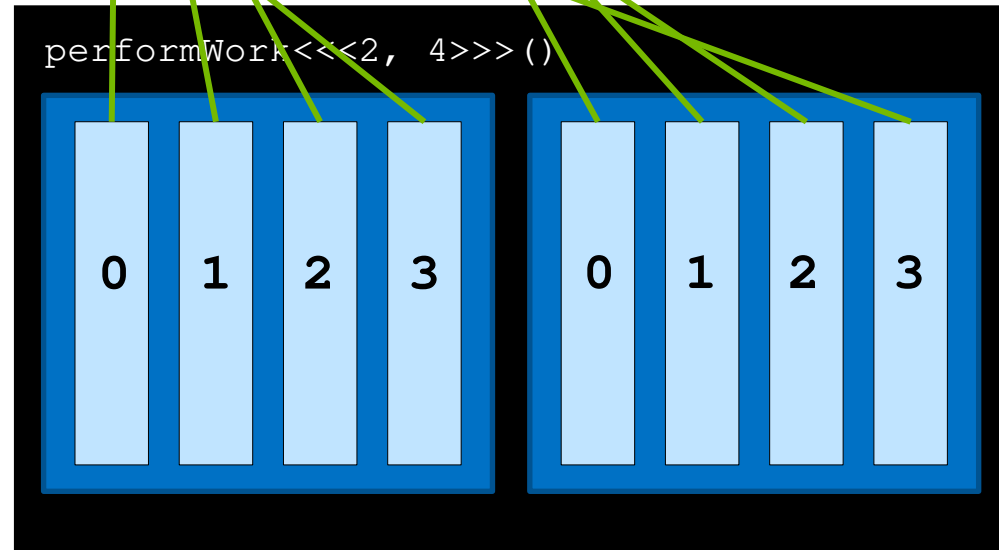


GPU
DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads working in this way, all elements are covered

GPU

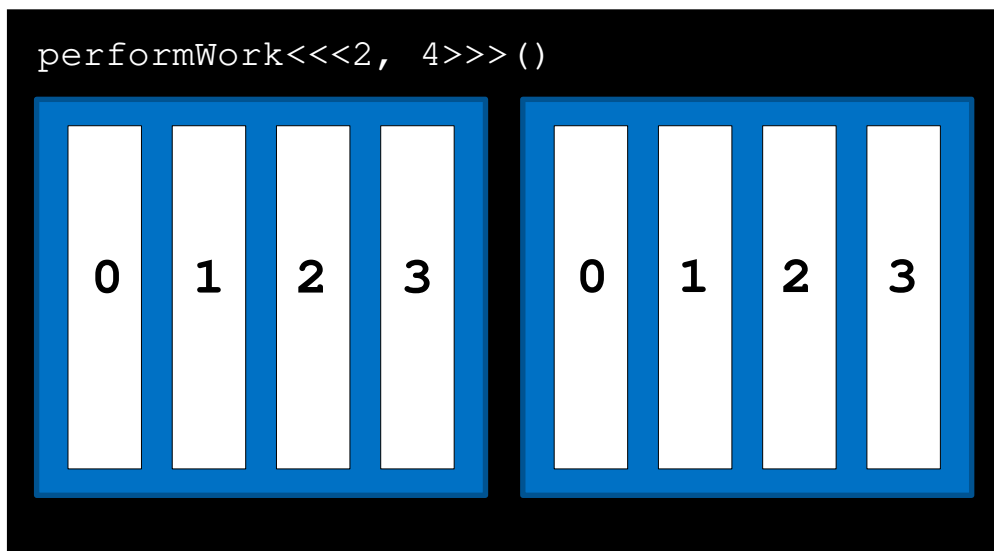


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads working in this way, all elements are covered

GPU

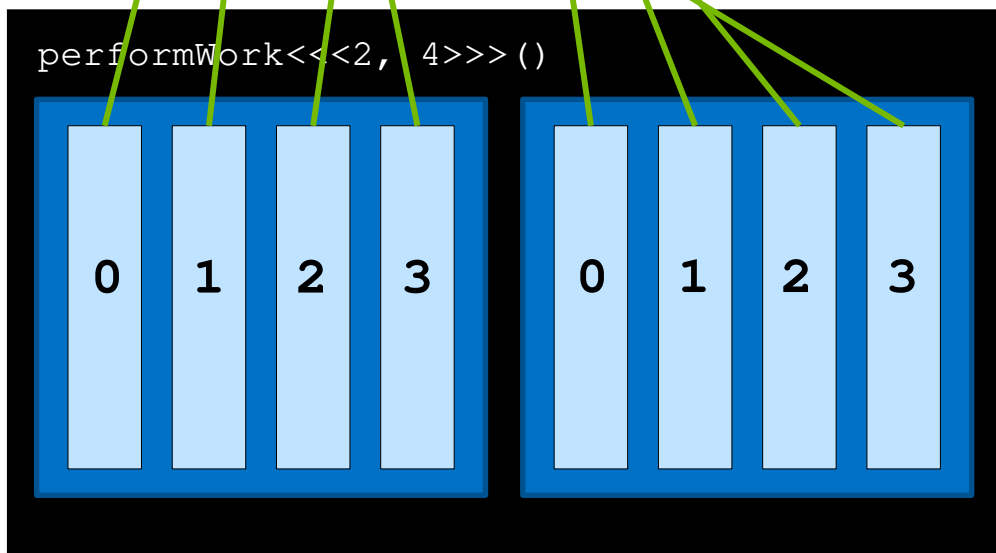


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads working in this way, all elements are covered

GPU

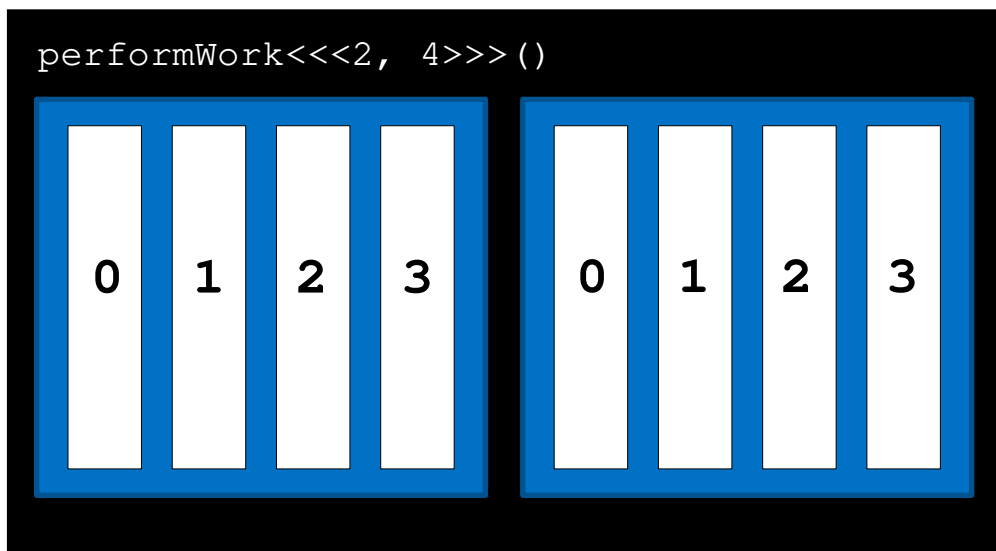


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads working in this way, all elements are covered

GPU

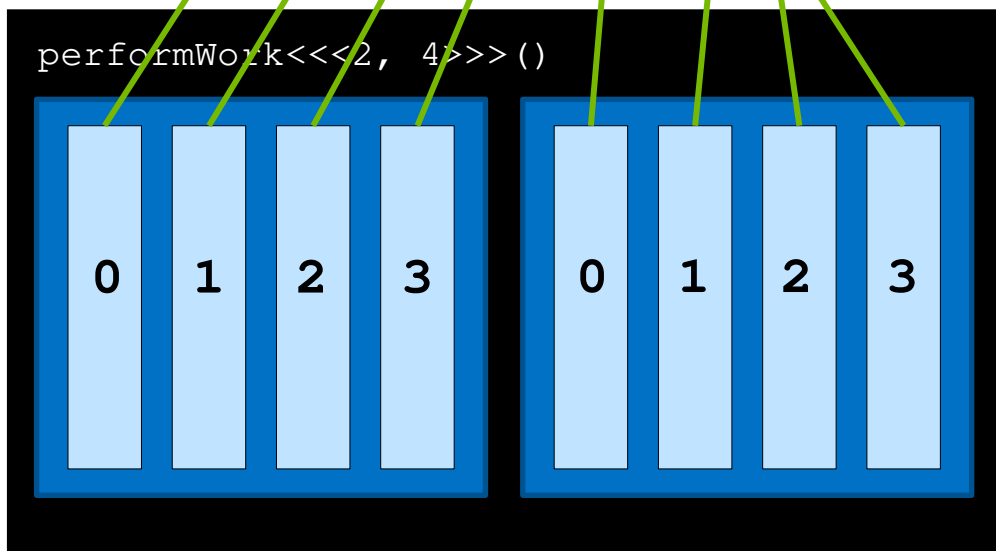


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads working in this way, all elements are covered

GPU

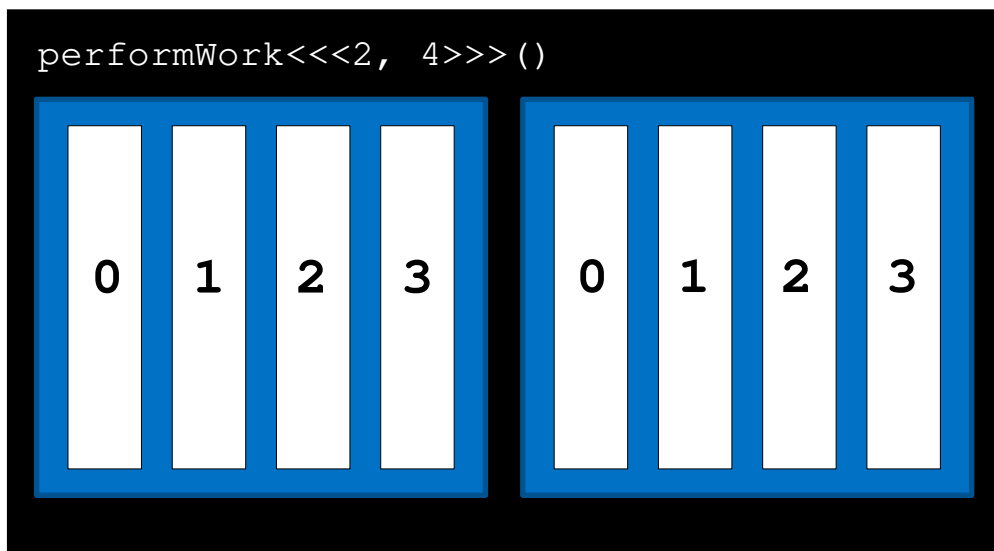


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads working in this way, all elements are covered

GPU

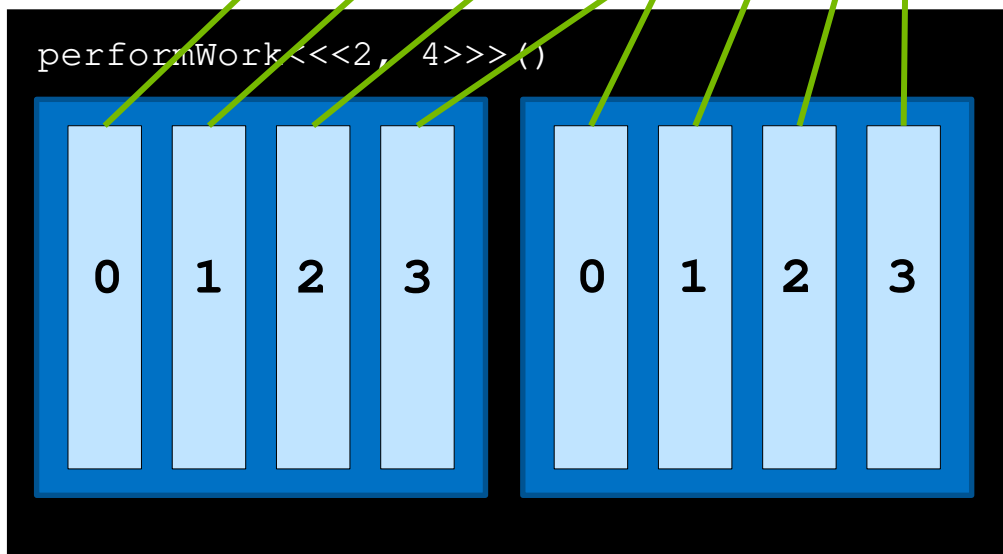


GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads working in this way, all elements are covered

GPU

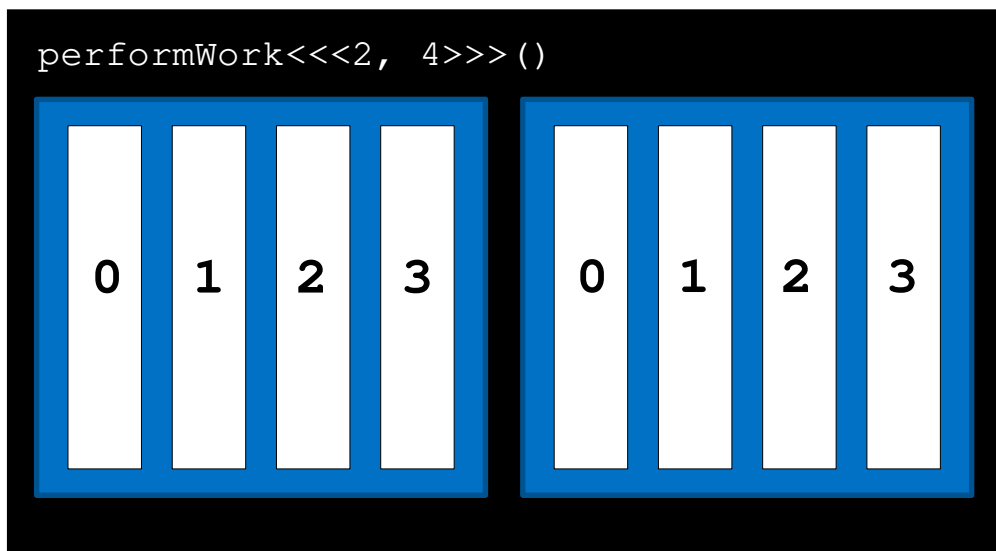


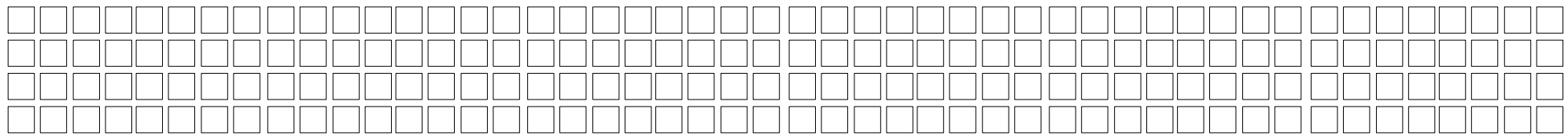
GPU DATA

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads working in this way, all elements are covered

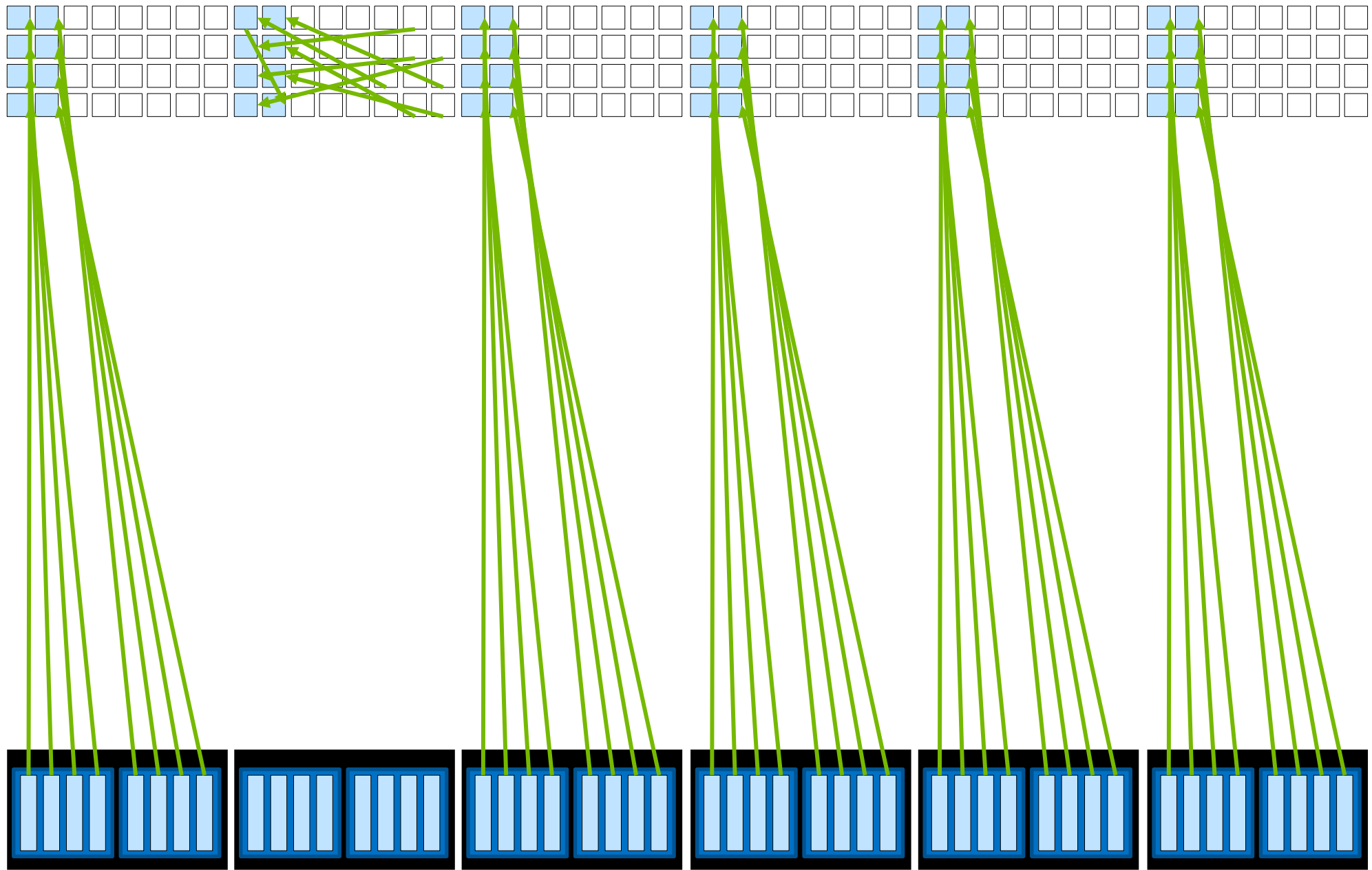
GPU

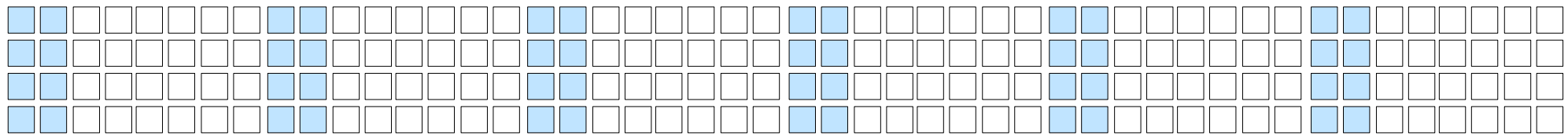


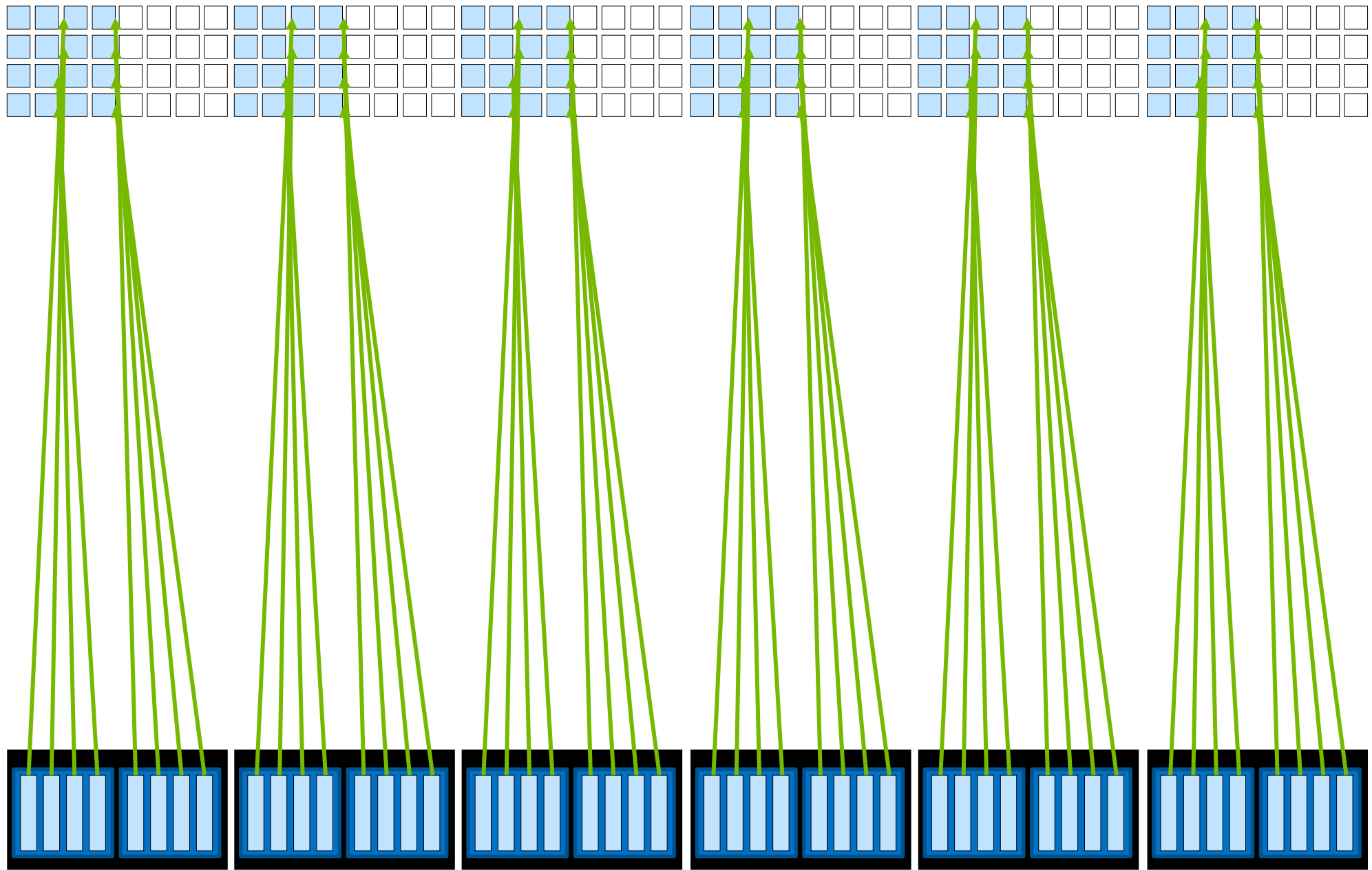


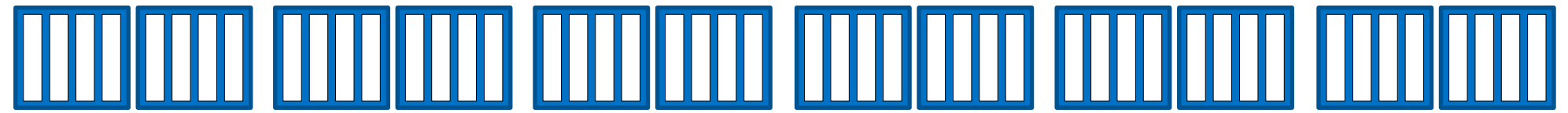
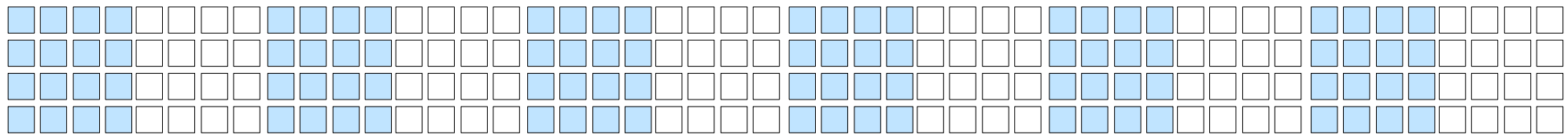
CUDA runs as many blocks
in parallel at once as the
GPU hardware supports, for
massive parallelization

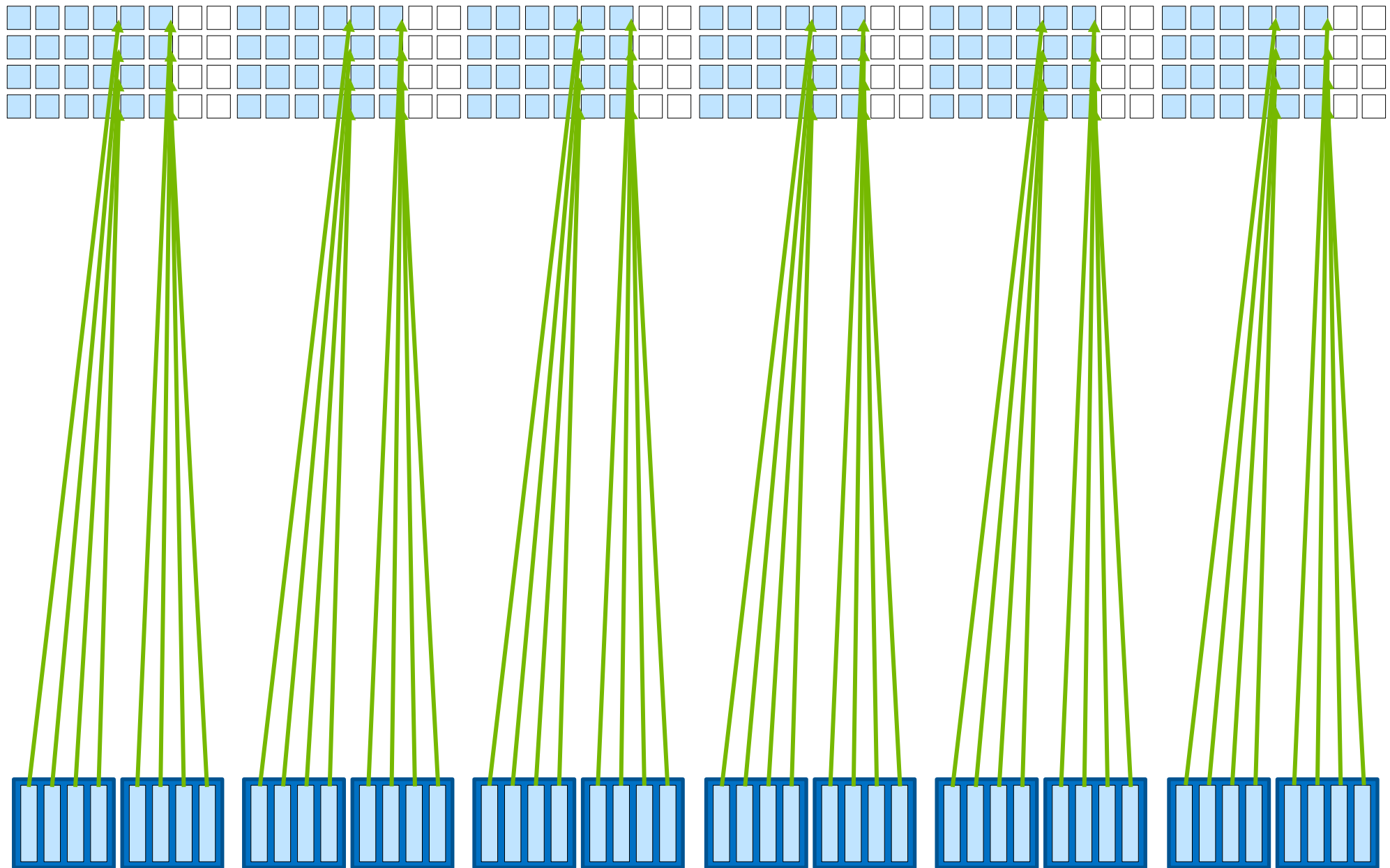


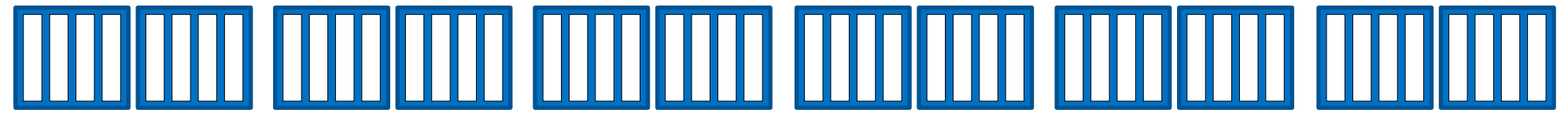
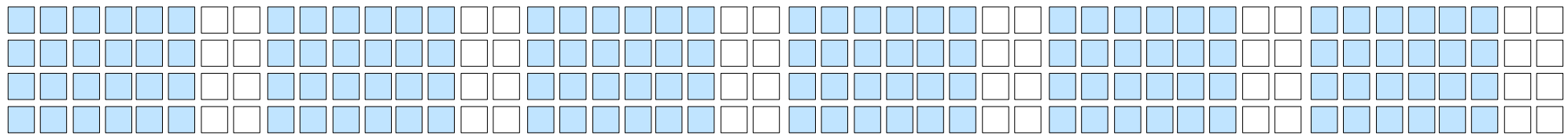


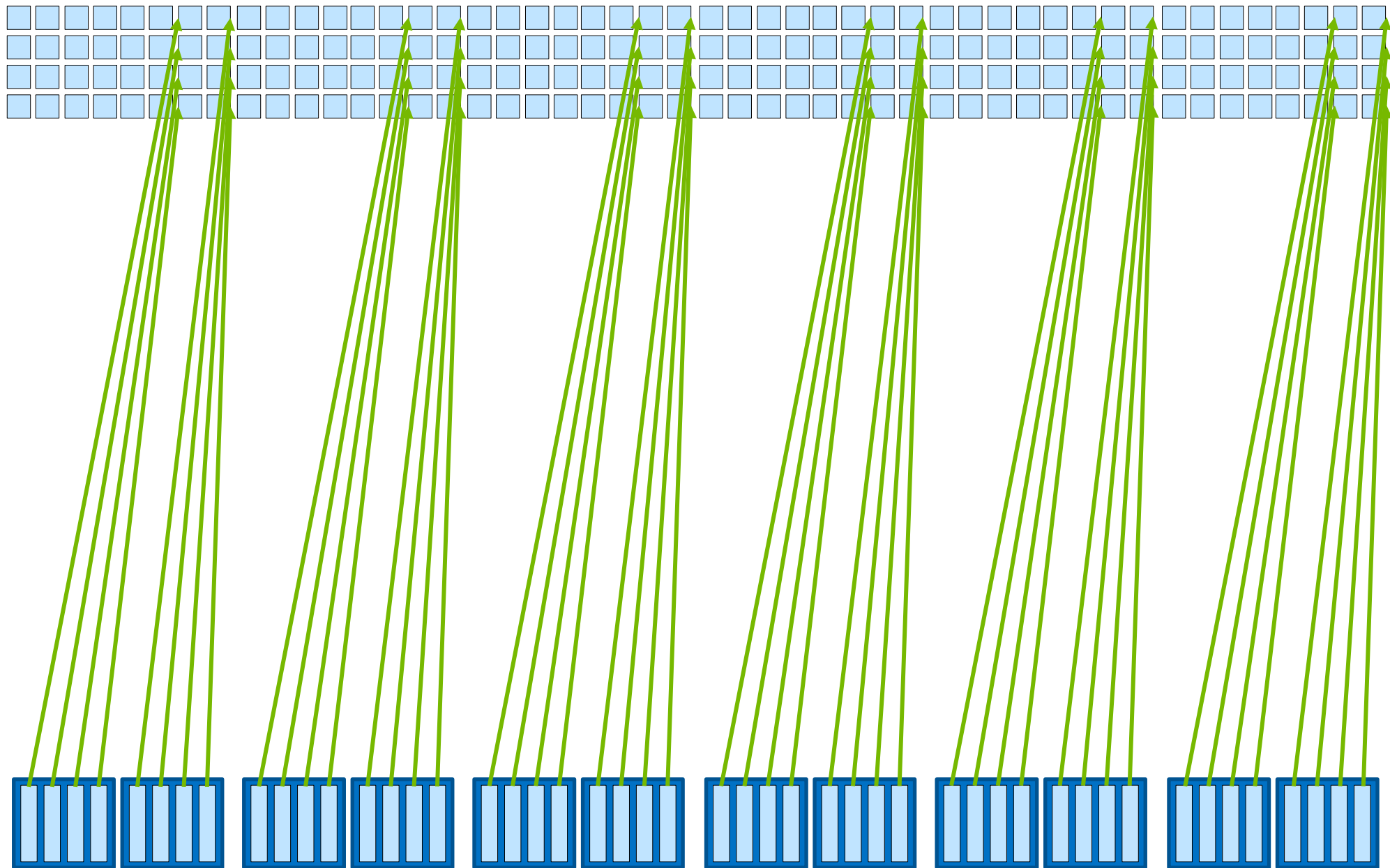


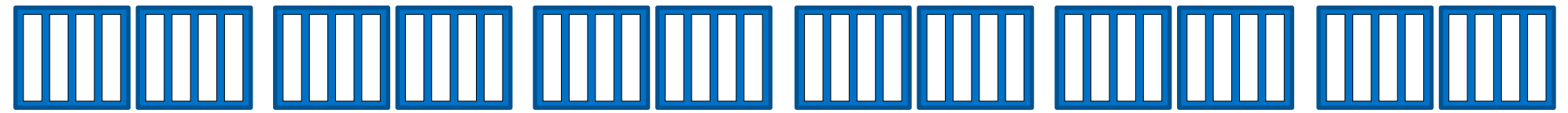
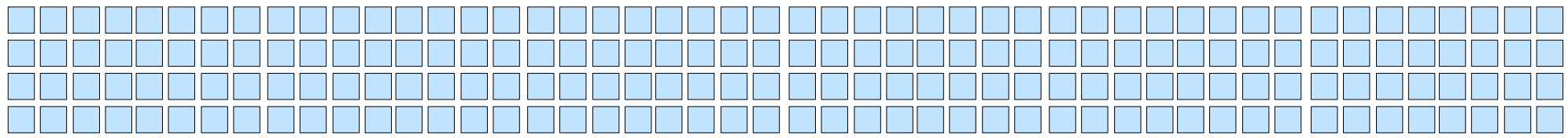












Glossary

Glossary

- **cudaMallocManaged()**: CUDA function to allocate memory accessible by both the CPU and GPUs. Memory allocated this way is called *unified memory* and is automatically migrated between the CPU and GPUs as needed.
- **cudaDeviceSynchronize()**: CUDA function that will cause the CPU to wait until the GPU is finished working.
- **Kernel**: A CUDA function executed on a GPU.
- **Thread**: The unit of execution for CUDA kernels.
- **Block**: A collection of threads.
- **Grid**: A collection of blocks.
- **Execution context**: Special arguments given to CUDA kernels when launched using the `<<<...>>>` syntax. It defines the number of blocks in the grid, as well as the number of threads in each block.
- **gridDim.x**: CUDA variable available inside executing kernel that gives the number of blocks in the grid
- **blockDim.x**: CUDA variable available inside executing kernel that gives the number of threads in the thread's block
- **blockIdx.x**: CUDA variable available inside executing kernel that gives the index the thread's block within the grid
- **threadIdx.x**: CUDA variable available inside executing kernel that gives the index the thread within the block
- **threadIdx.x + blockIdx.x * blockDim.x**: Common CUDA technique to map a thread to a data element
- **Grid-stride loop**: A technique for assigning a thread more than one data element to work on when there are more elements than the number of threads in the grid. The stride is calculated by **gridDim.x * blockDim.x**, which is the number of threads in the grid.



DEEP
LEARNING
INSTITUTE

www.nvidia.com/dli