Accelerating Applications with CUDA C/C++

TOPICS

GPU-accelerated vs. CPU-only Applications

CUDA Kernel Execution

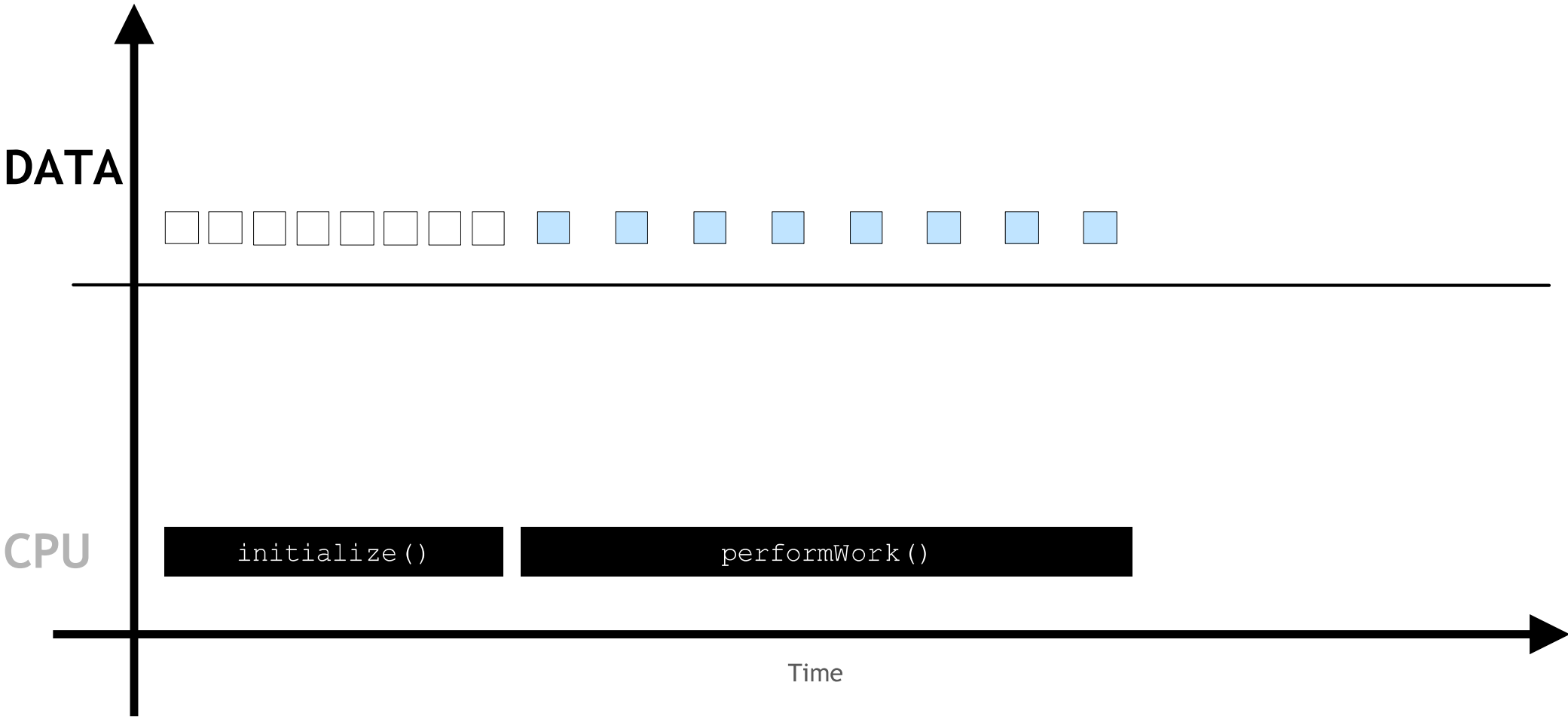
Parallel Memory Access

Appendix: Glossary

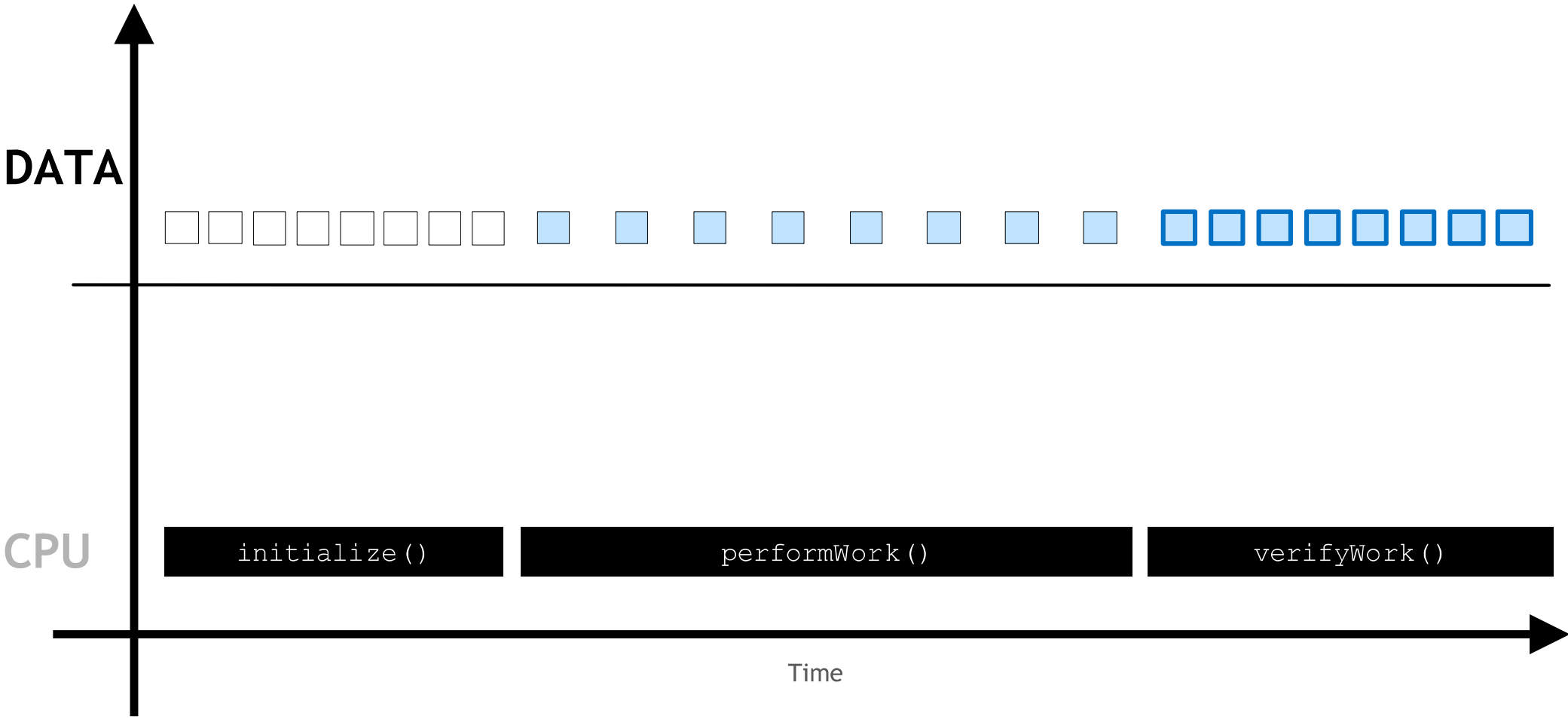
GPU-accelerated vs. CPU-only Applications

In **CPU-only applications** da
allocated on CPU

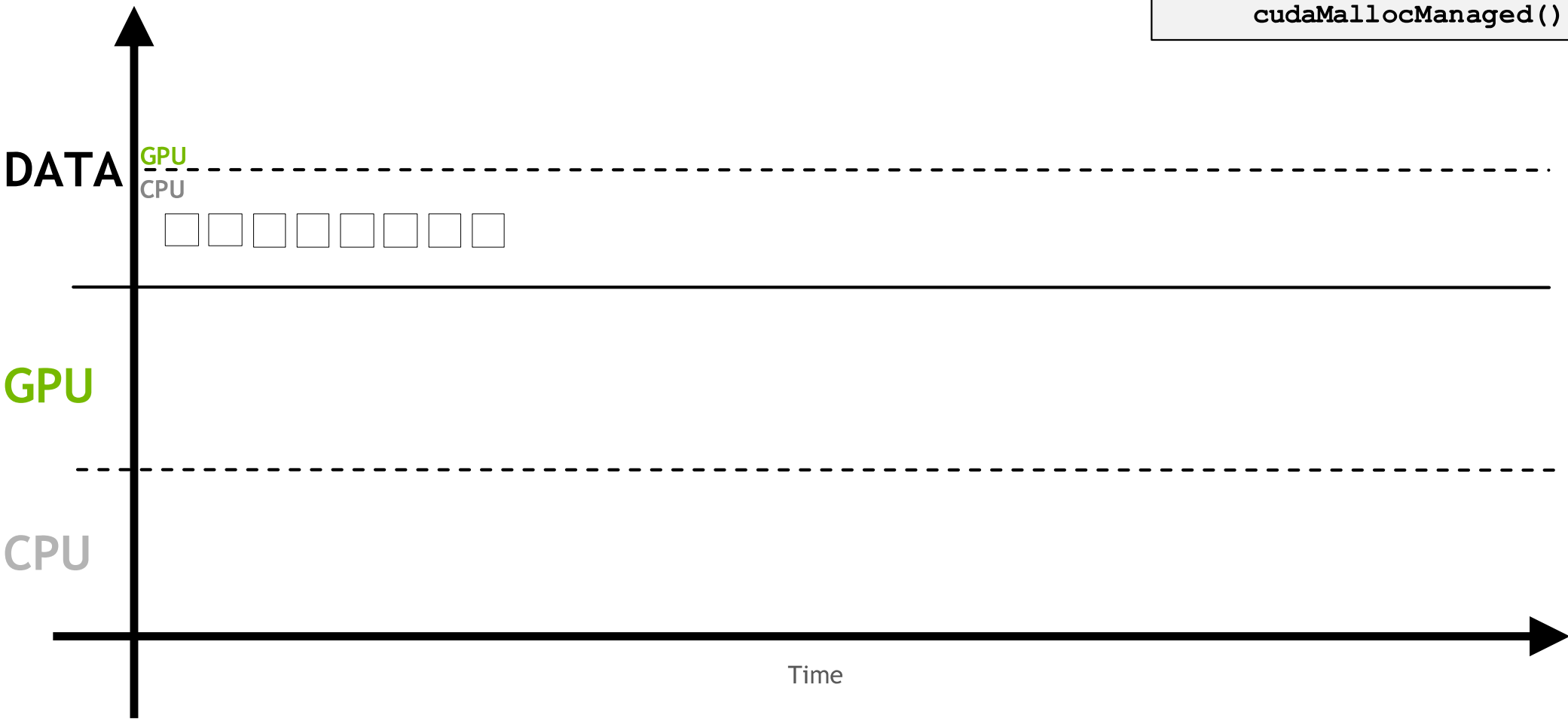
…and all work is performed on



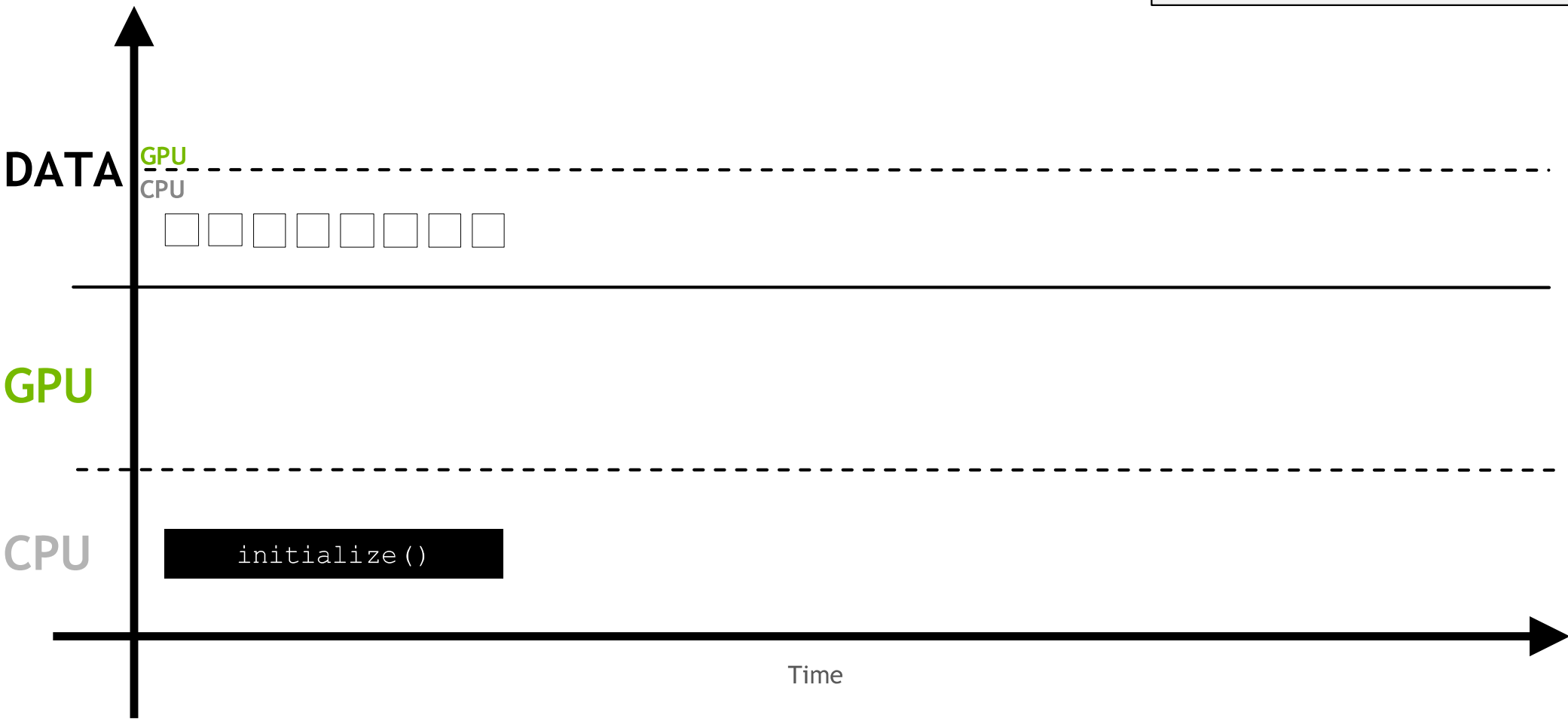
…and all work is performed on



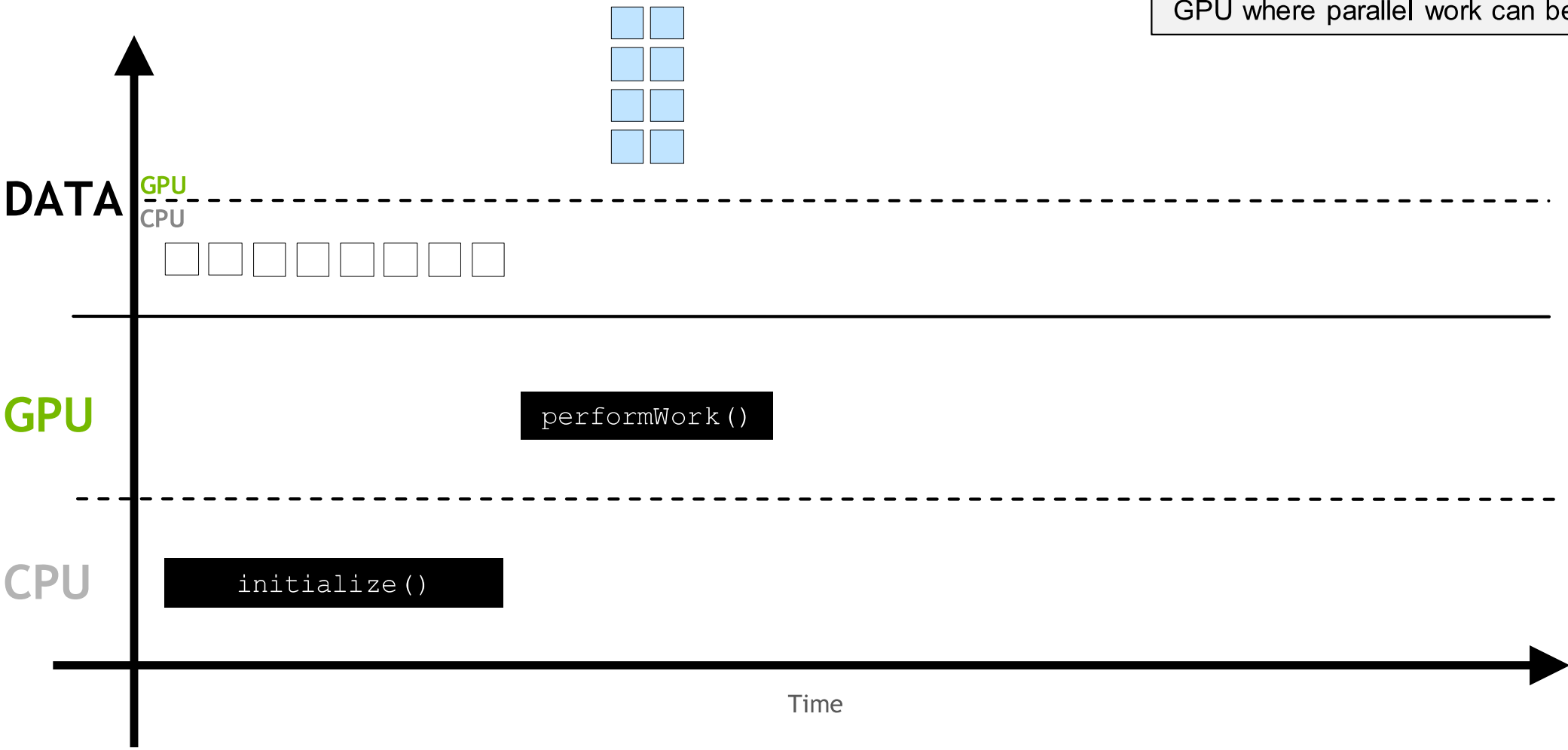
In **accelerated applications** d
allocated with
`cudaMallocManaged()`

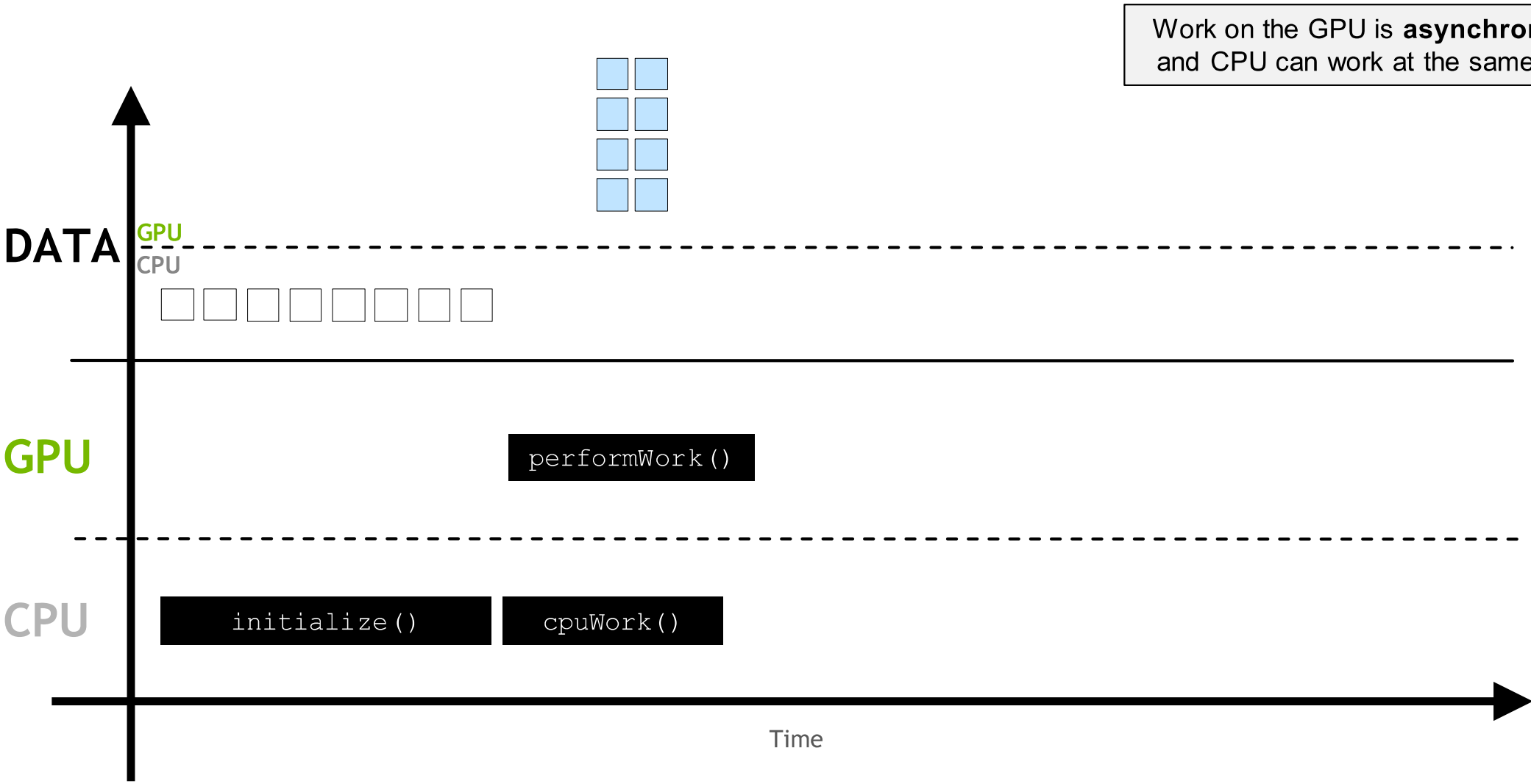


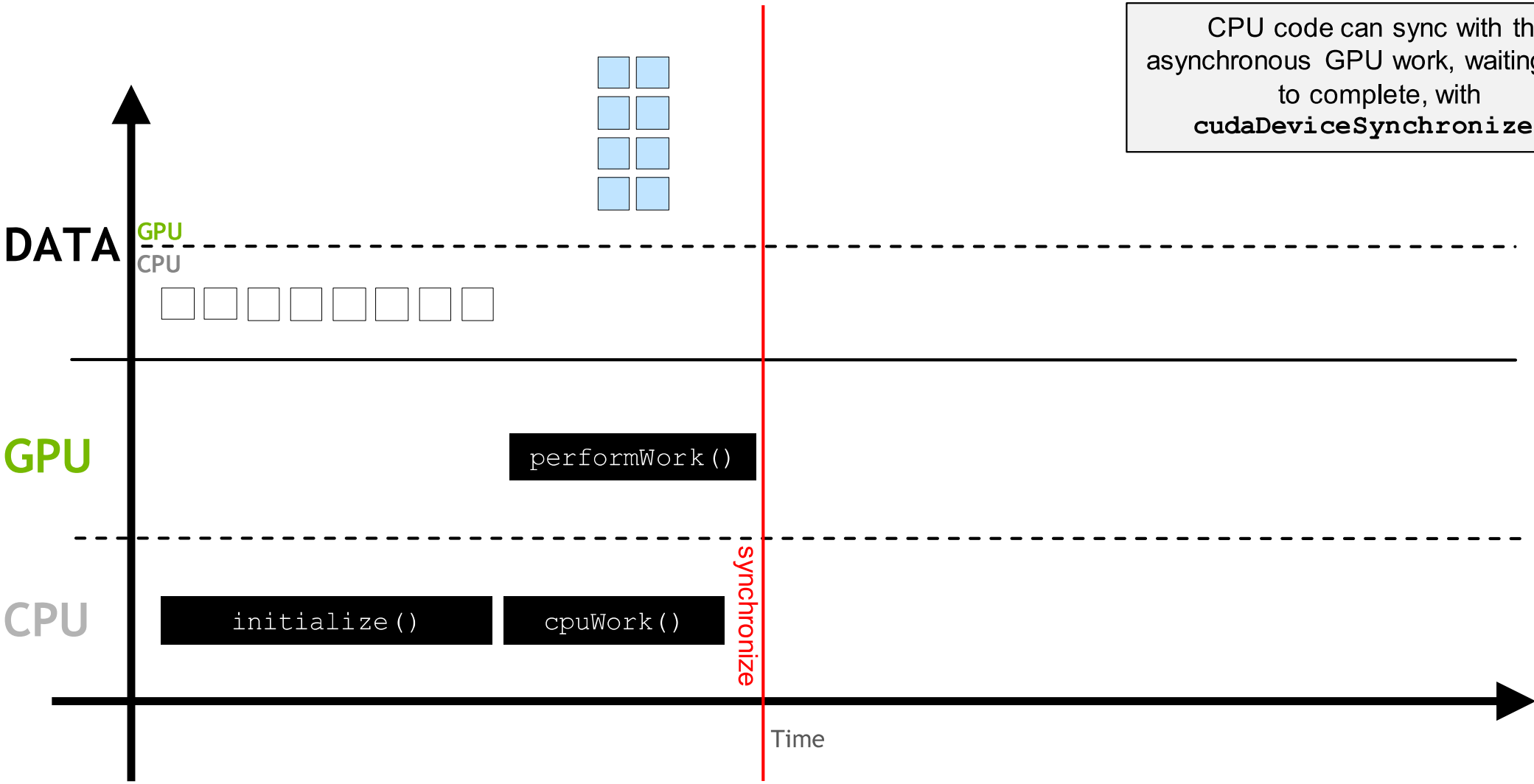
… where it can be accessed
worked on by the CPU

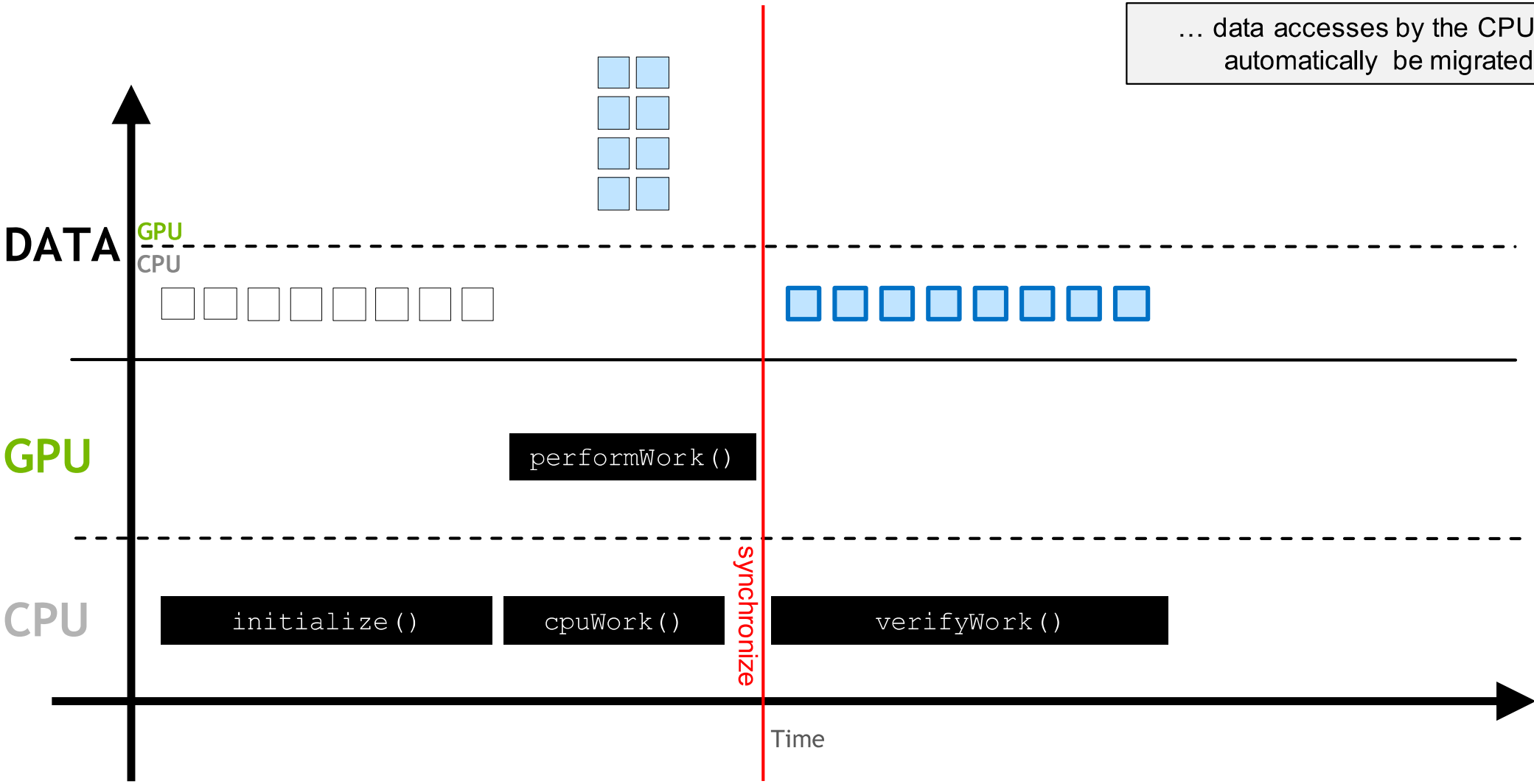


… and automatically migrated t
GPU where parallel work can be

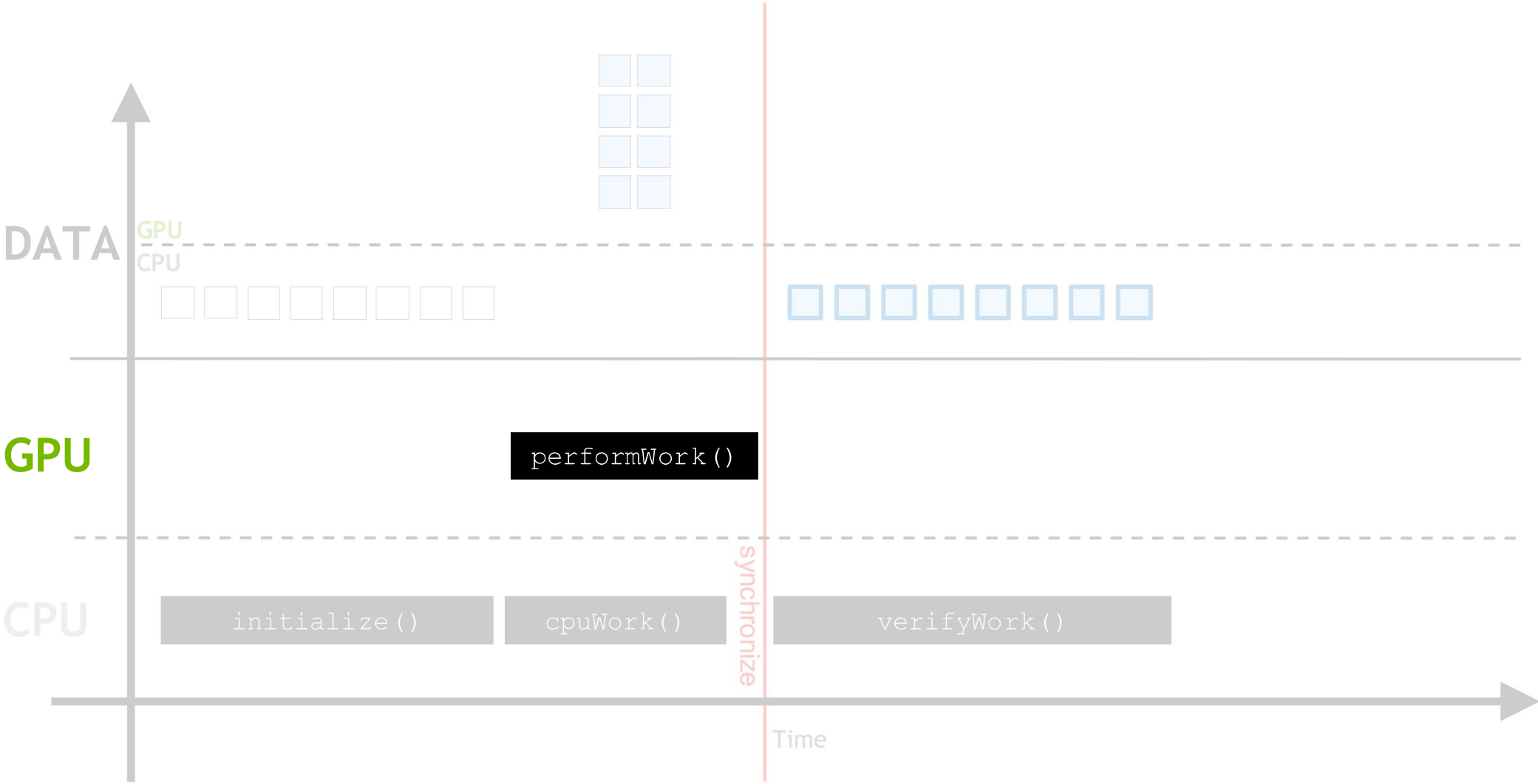



Work on the GPU is asynchro
and CPU can work at the same
DATA
GPU
CPU
GPU
performWork()
CPU
initialize()
cpuWork()
Time


CPU code can sync with th
asynchronous GPU work, waiting
to complete, with
cudaDeviceSynchronize
DATA
GPU
CPU
GPU
performWork()
CPU
initialize()
cpuWork()
synchronize
Time

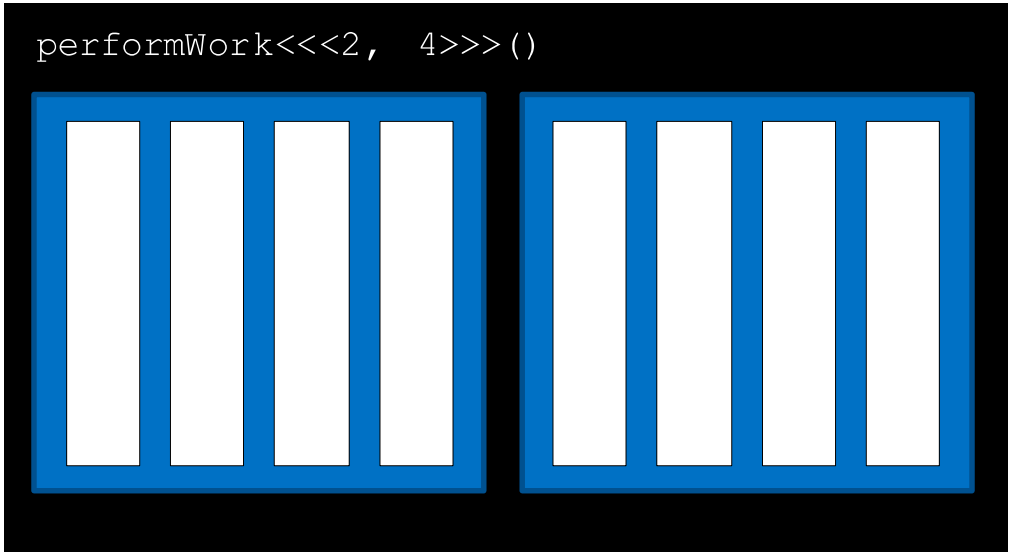

… data accesses by the CPU automatically be migrated
DATA
GPU
CPU
GPU
performWork()
CPU
initialize()
cpuWork()
synchronize
verifyWork()
Time

CUDA Kernel Execution


DATA
GPU
CPU
GPU
performWork()
CPU
initialize()
cpuWork()
synchronize
verifyWork()
Time

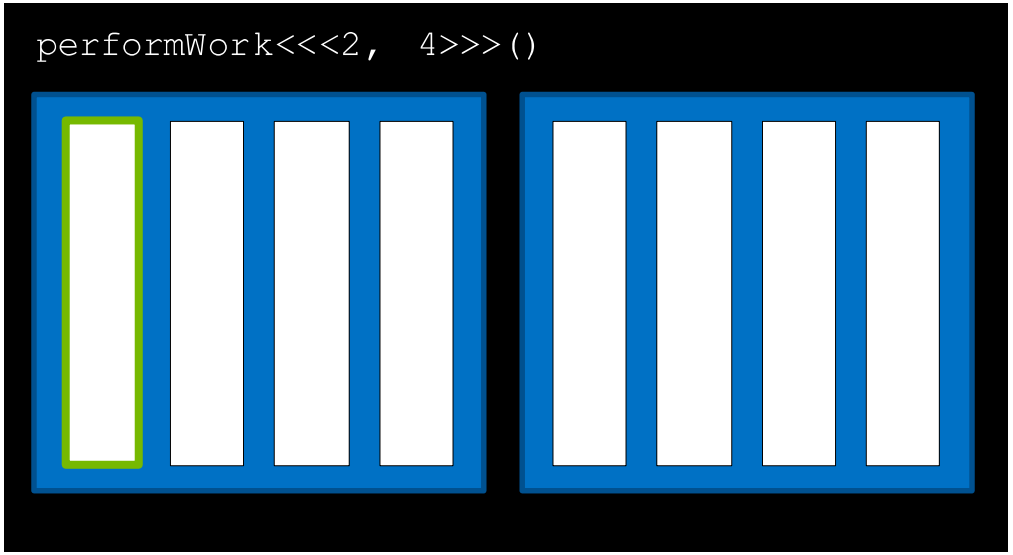
GPUs do work in parallel

GPU



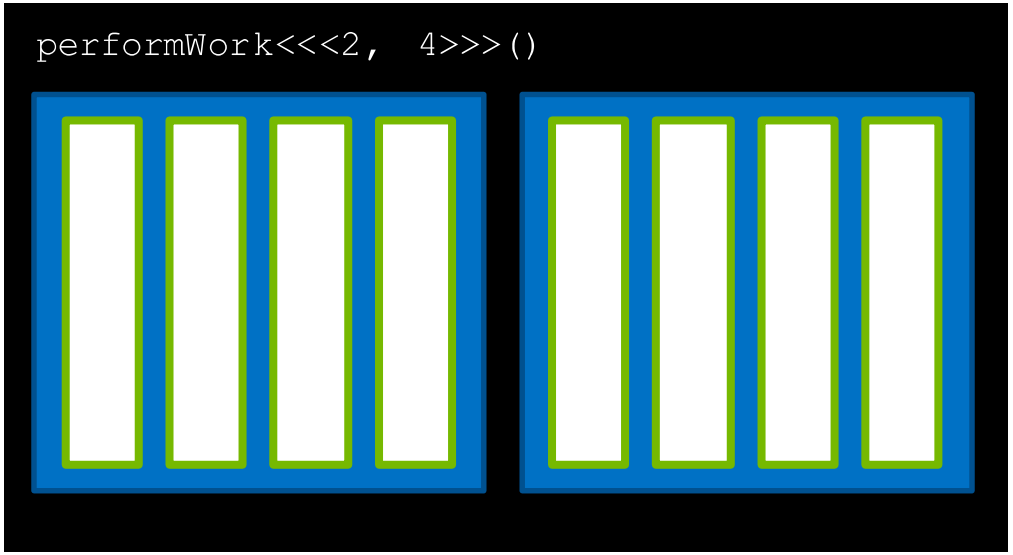
GPU work is done in a **threa**

GPU



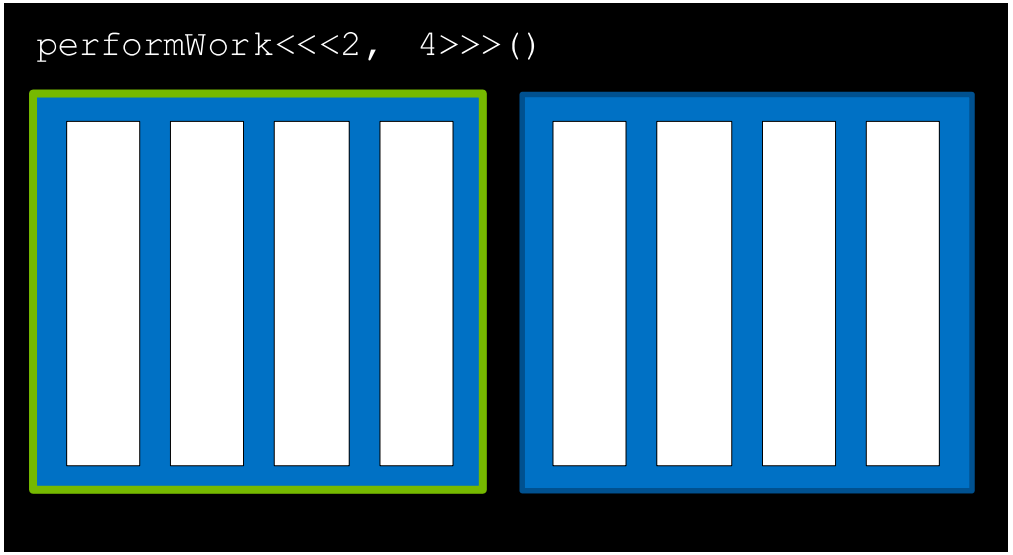
Many threads run in paralle

GPU



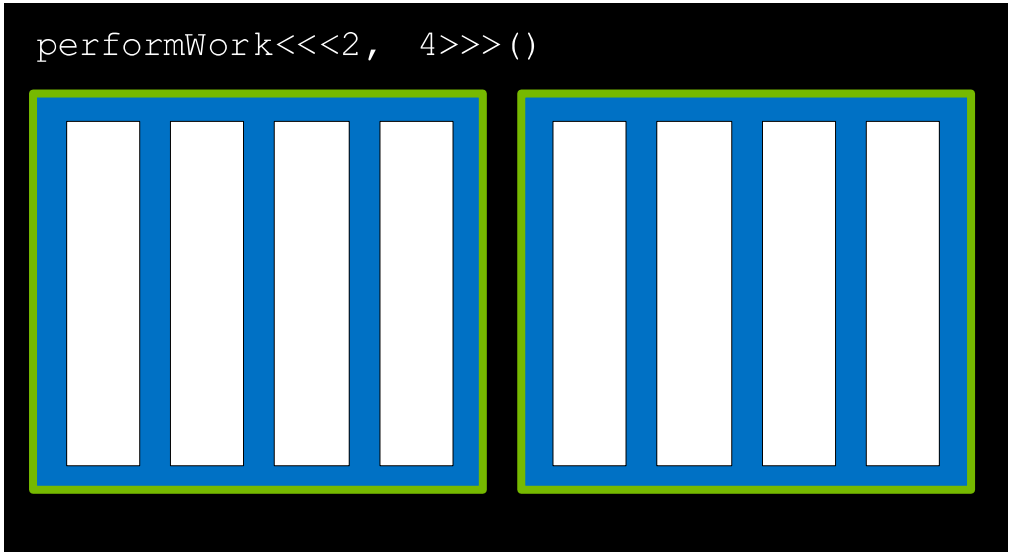
A collection of threads is a **bl**

GPU



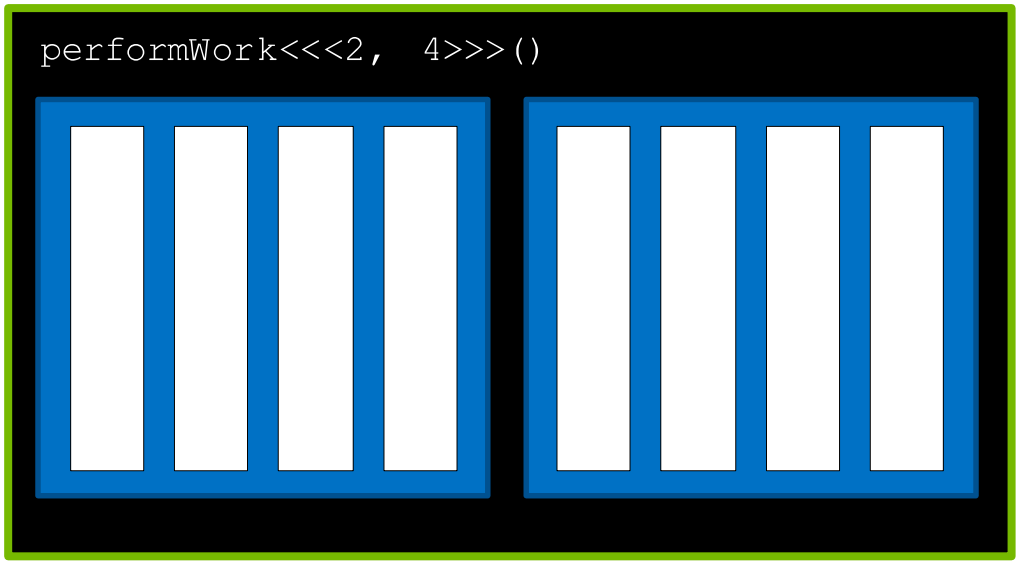
There are many blocks

GPU



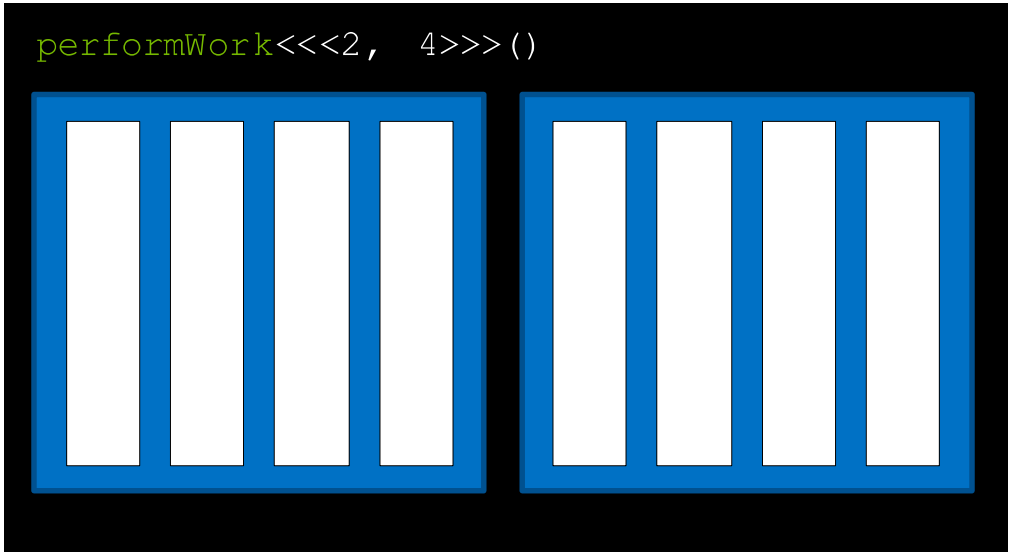
A collection of blocks is a **gr**

GPU



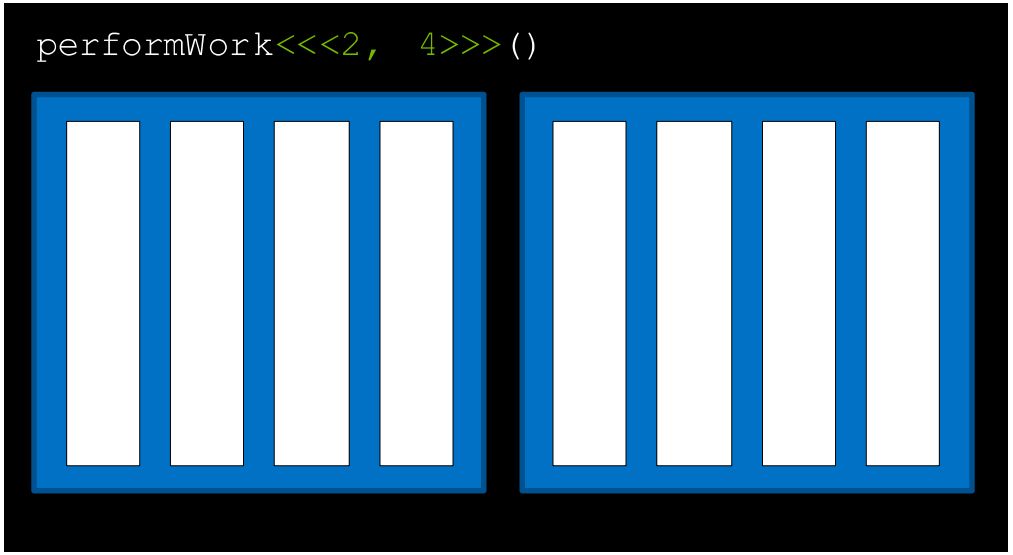
GPU functions are called **ker**

GPU



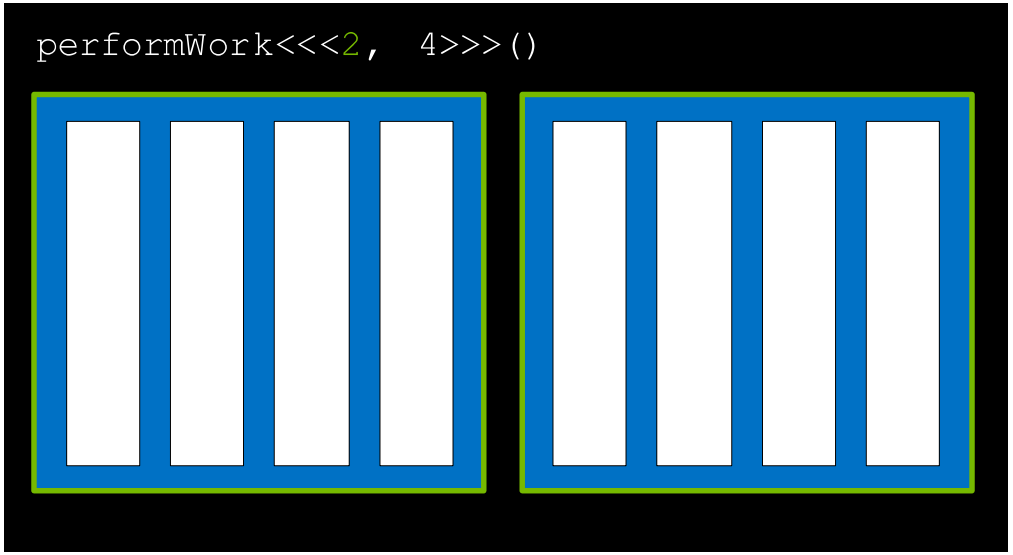
Kernels are **launched** with
execution configuration

GPU



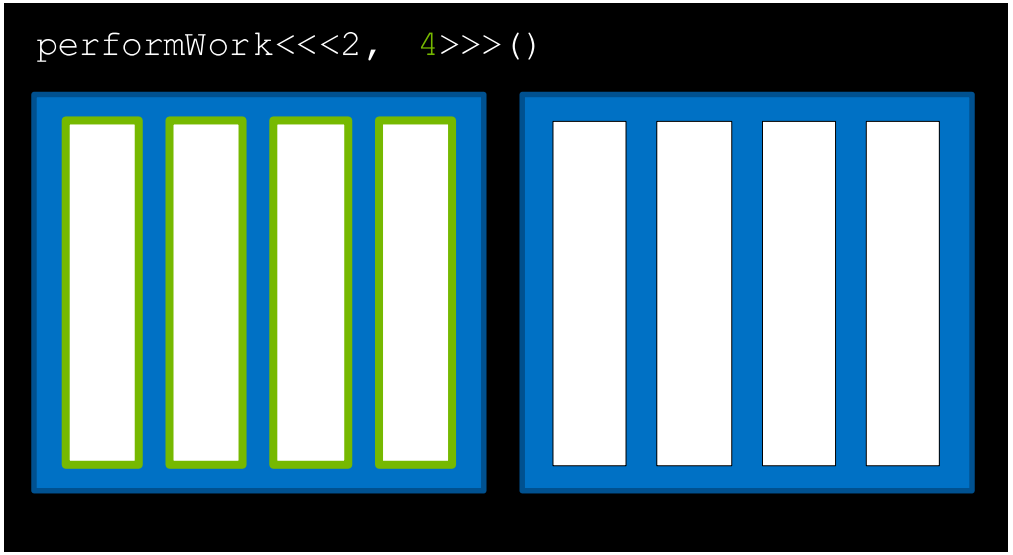
The execution configuration de
the number of blocks in the g

GPU



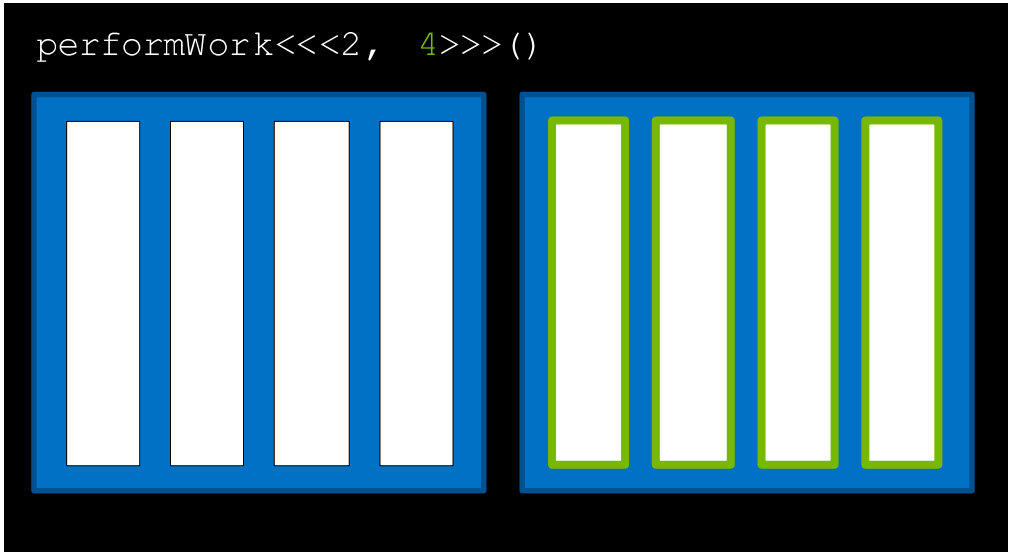
… as well as the number of thre
each block

GPU



Every block in the grid contains
same number of threads

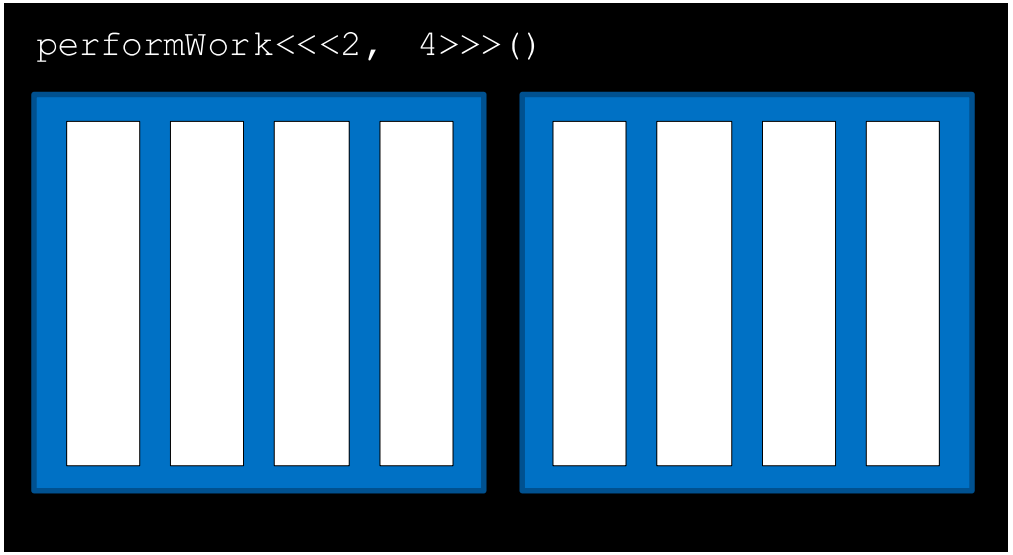
GPU



CUDA-Provided Thread Hierarchy Variables

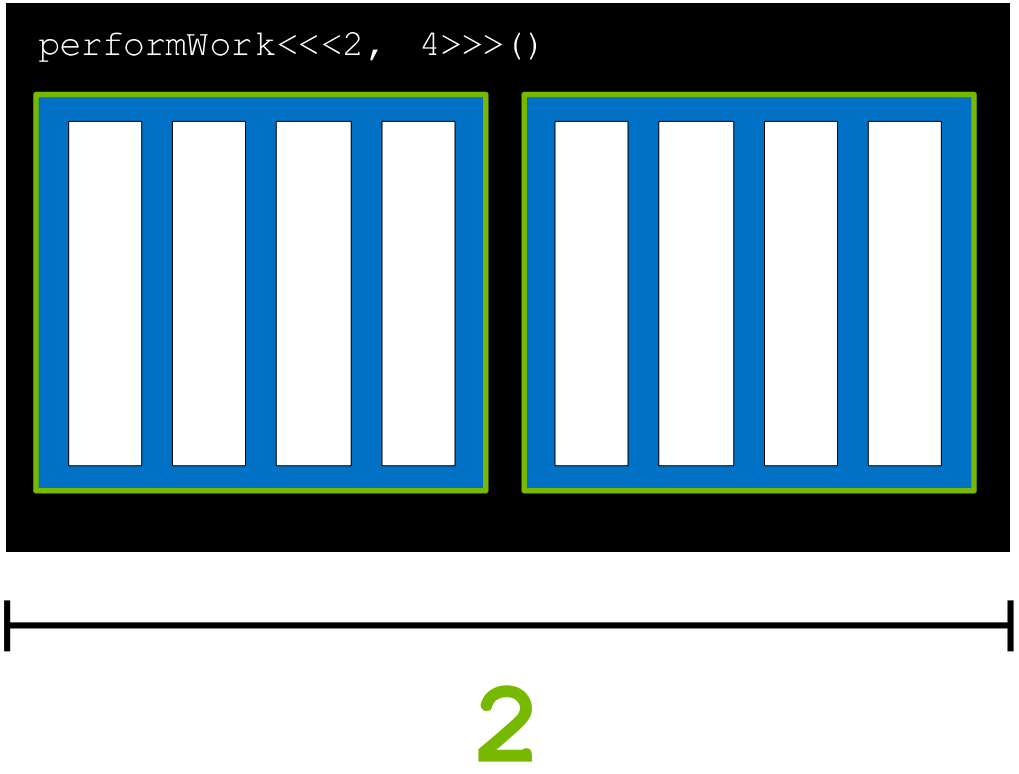
Inside kernels definitions, CU
provided variables describe
executing thread, block, and

GPU



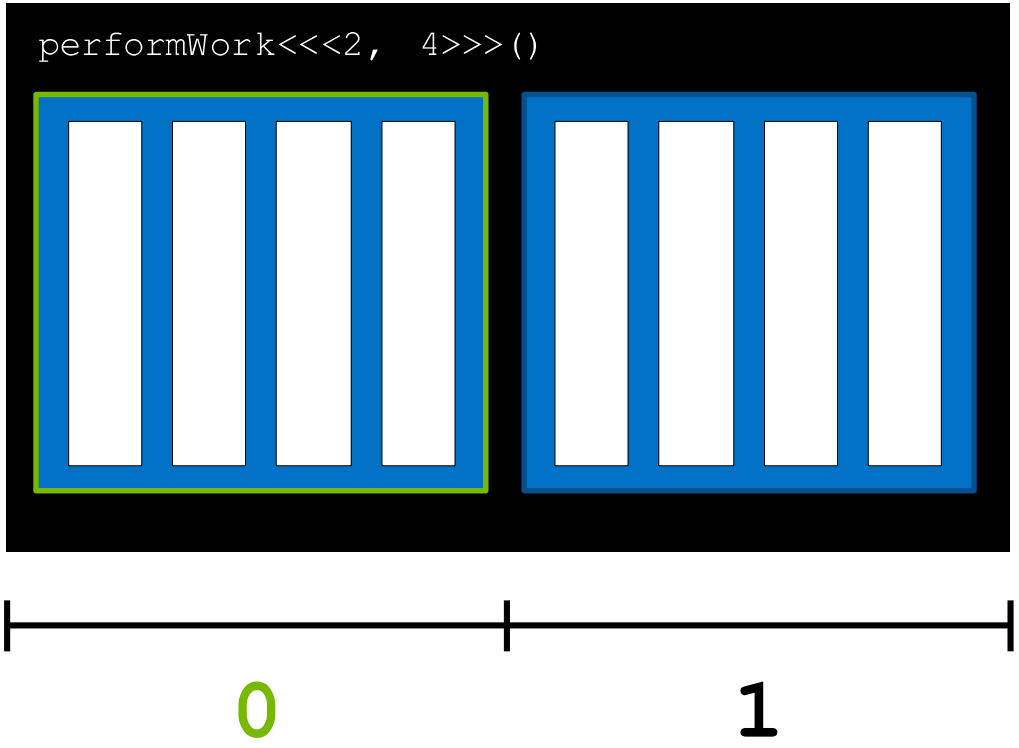
gridDim.x is the number of blo
the grid, in this case **2**

GPU



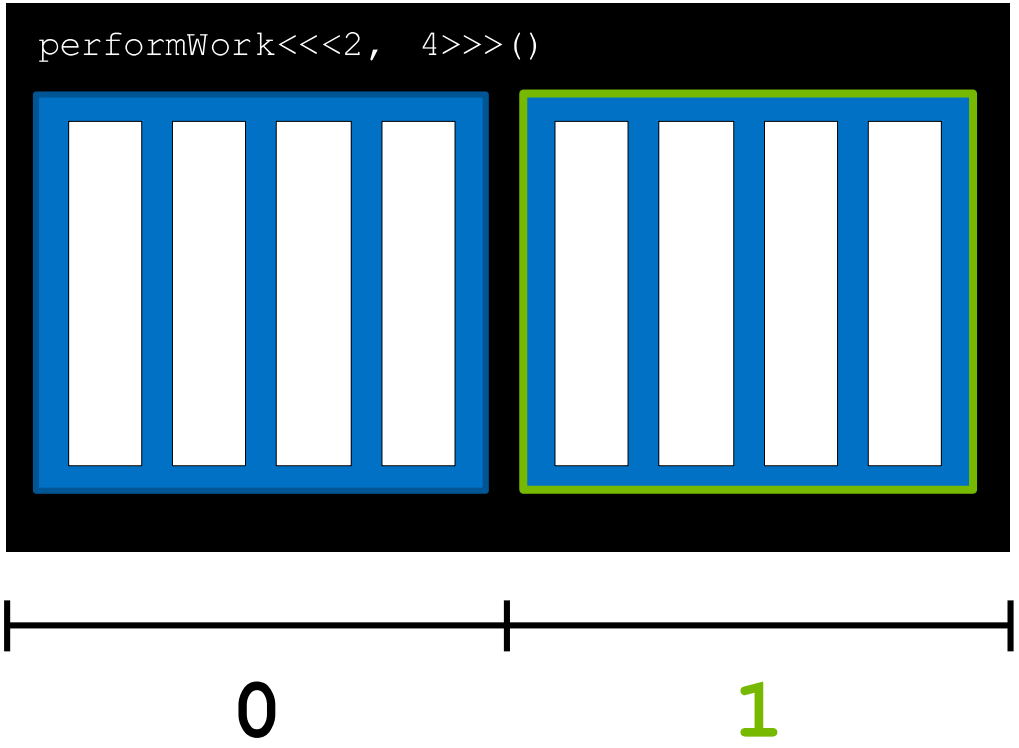
`blockIdx.x` is the index of current block within the grid, in case **0**

GPU



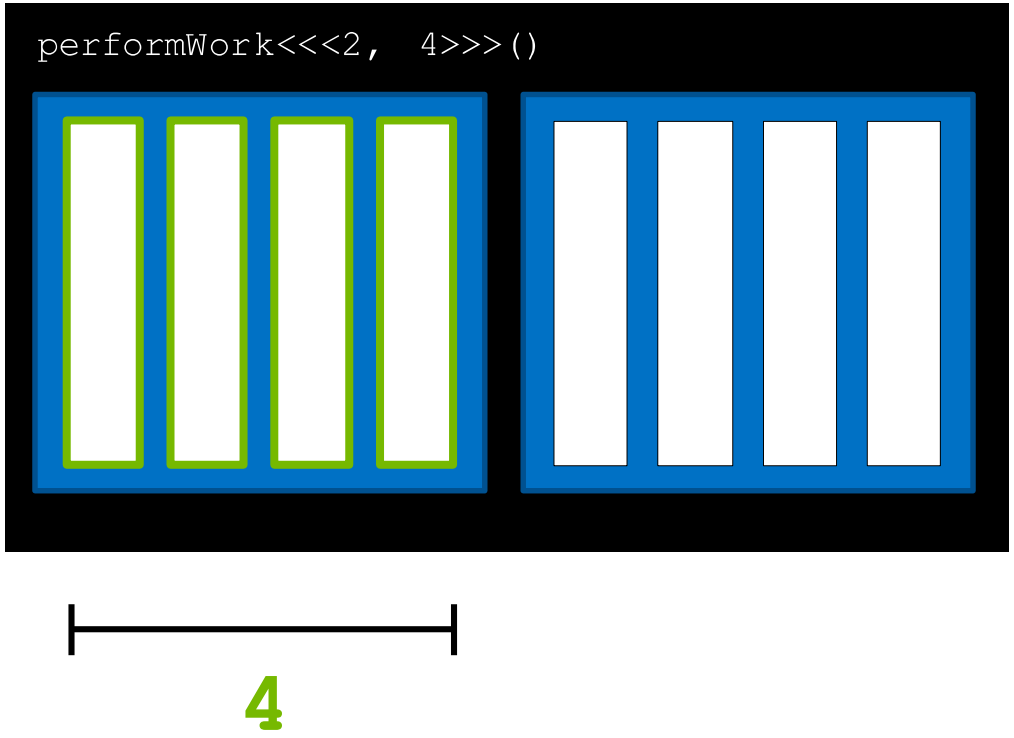
`blockIdx.x` is the index of current block within the grid, in case **`1`**

GPU



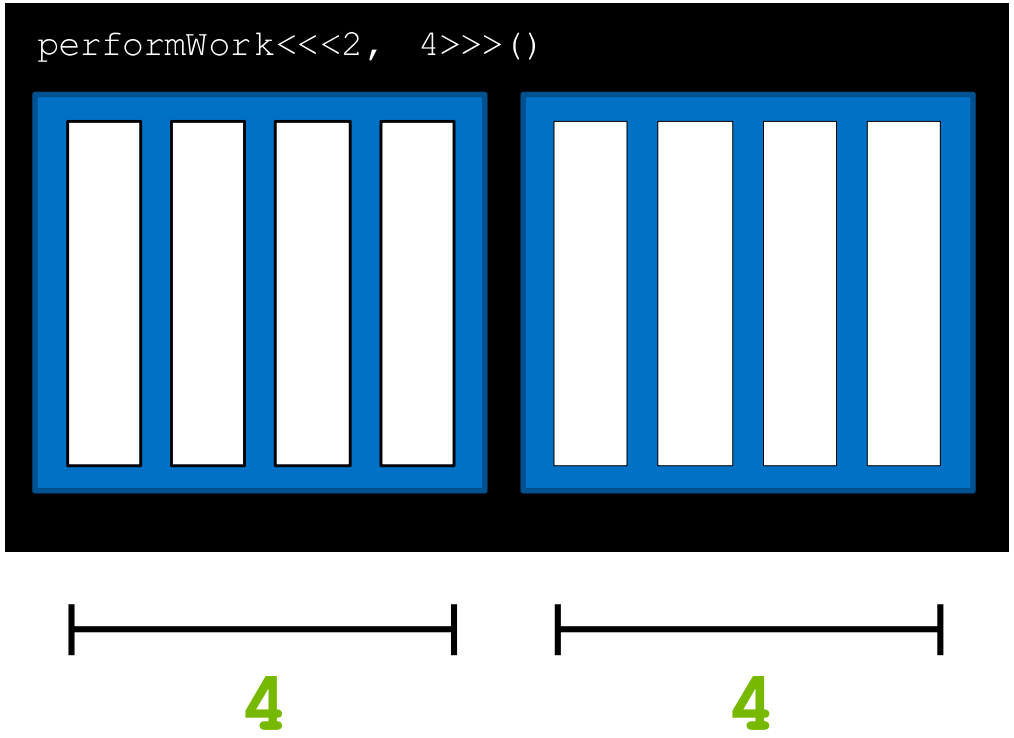
Inside a kernel **`blockDim.x`** des
the number of threads in a bloc
this case **`4`**

GPU



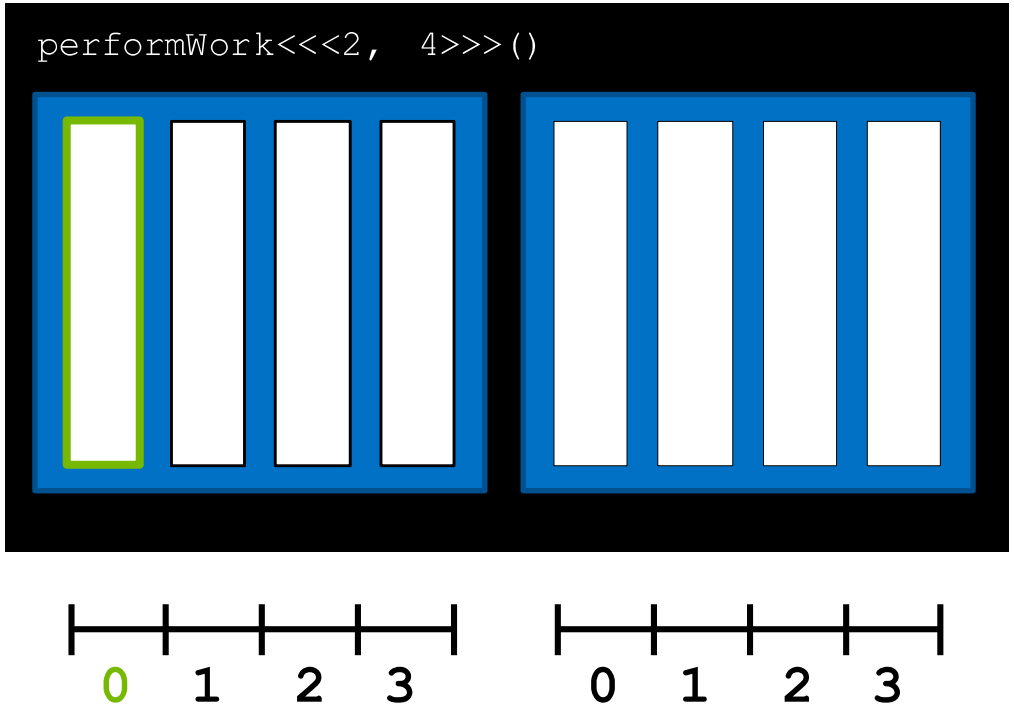
All blocks in a grid contain the s
number of threads

GPU



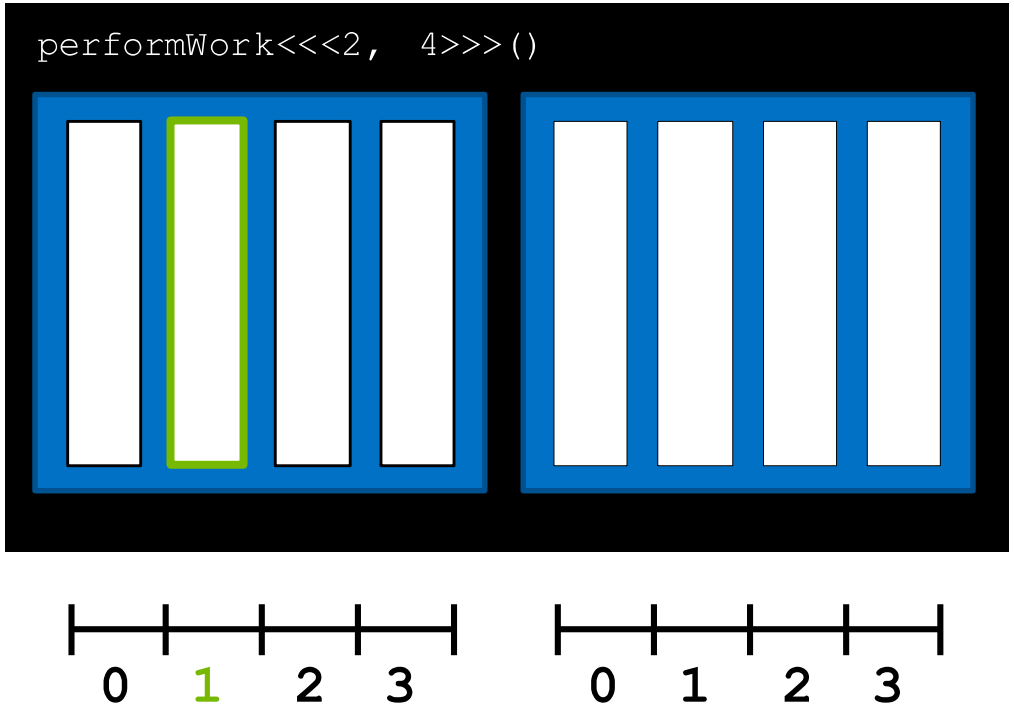
Inside a kernel **`threadIdx`**
describes the index of the thread
a block. In this case `0`

GPU



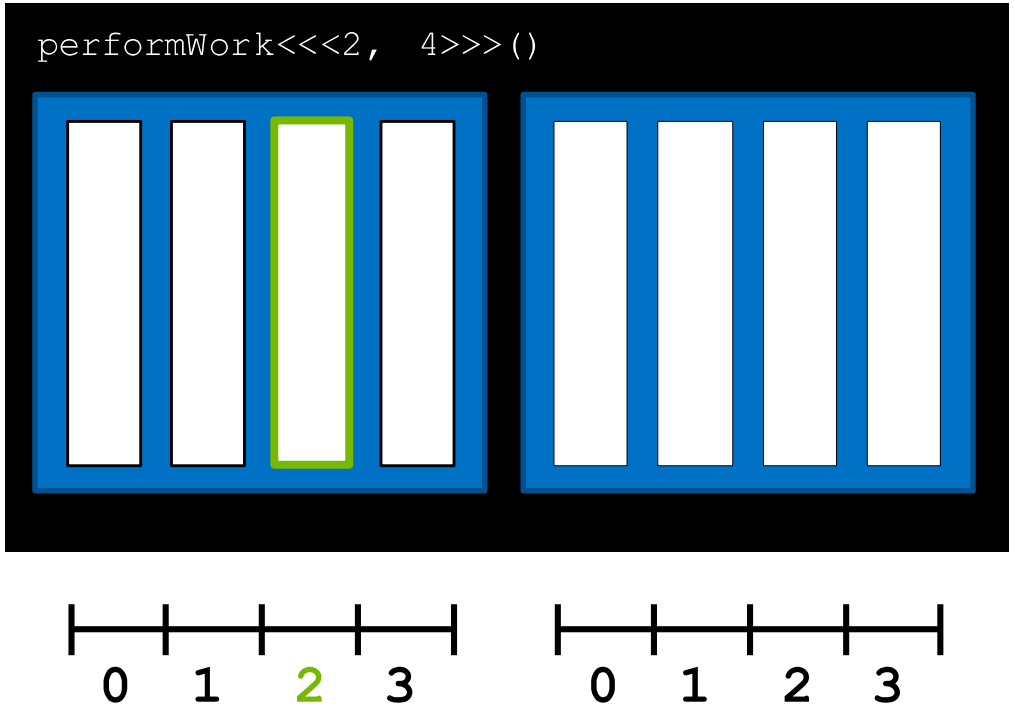
Inside a kernel **`threadIdx`**
describes the index of the thread
a block. In this case **`1`**

GPU



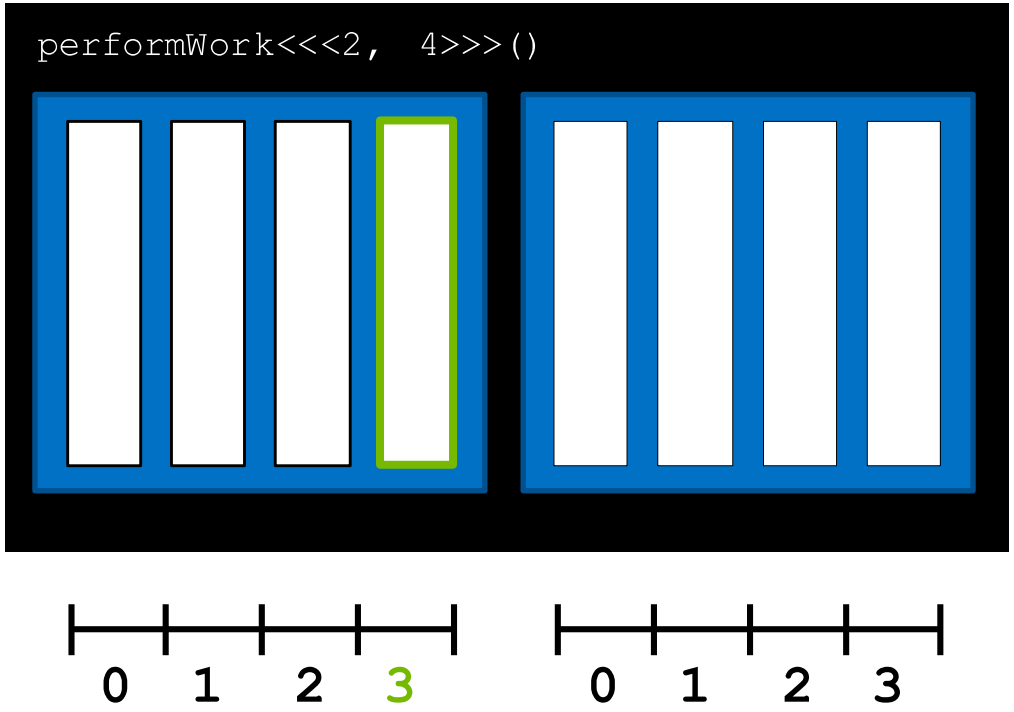
Inside a kernel **threadIdx**
describes the index of the thread
a block. In this case **2**

GPU



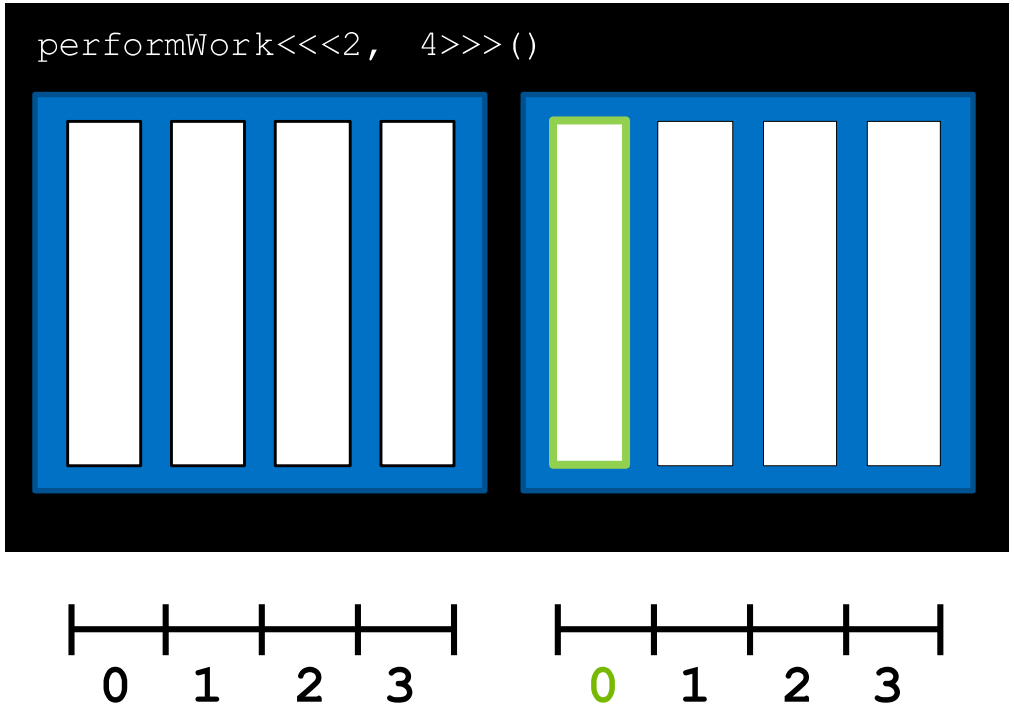
Inside a kernel **threadIdx**
describes the index of the thread
a block. In this case **3**

GPU



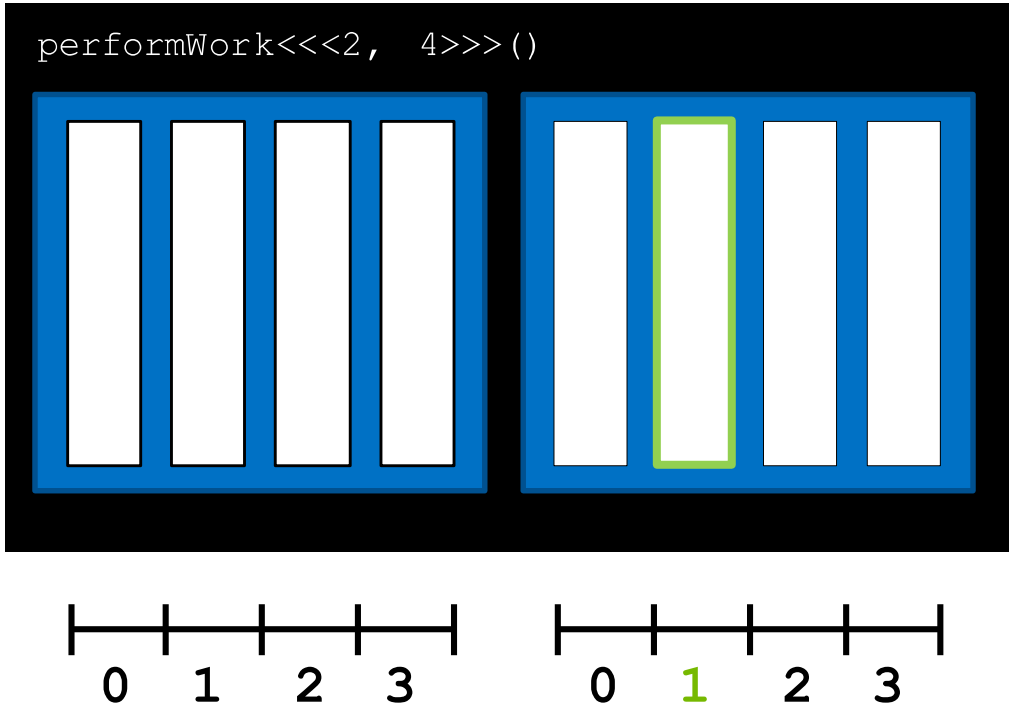
Inside a kernel **`threadIdx`**
describes the index of the thread
a block. In this case `0`

GPU



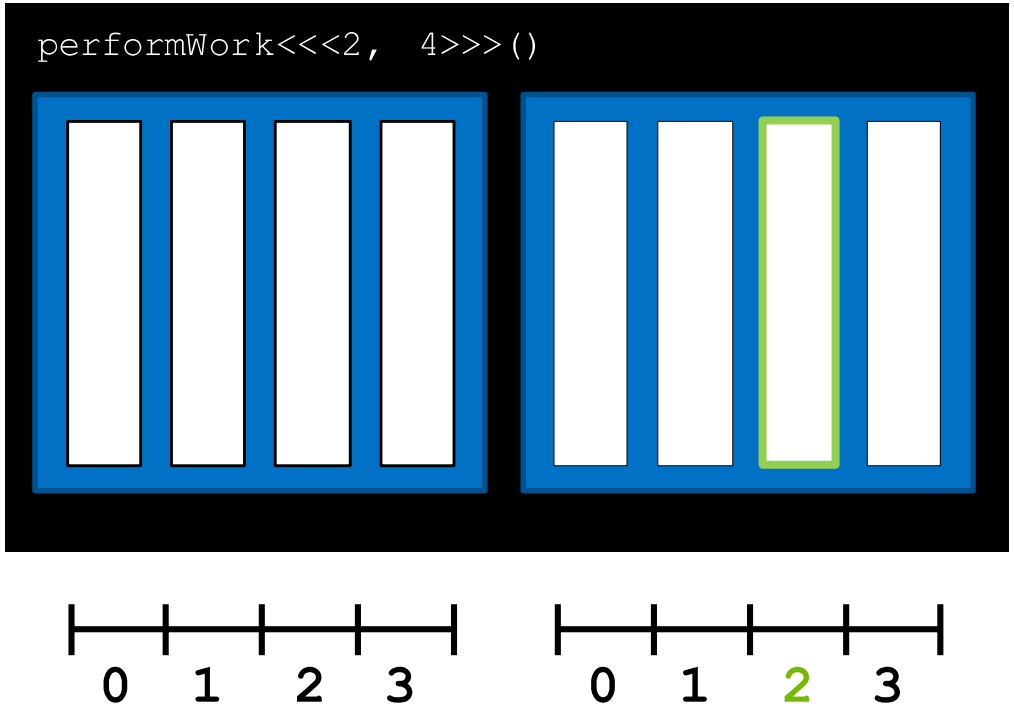
Inside a kernel **`threadIdx`**
describes the index of the thread
a block. In this case **`1`**

GPU



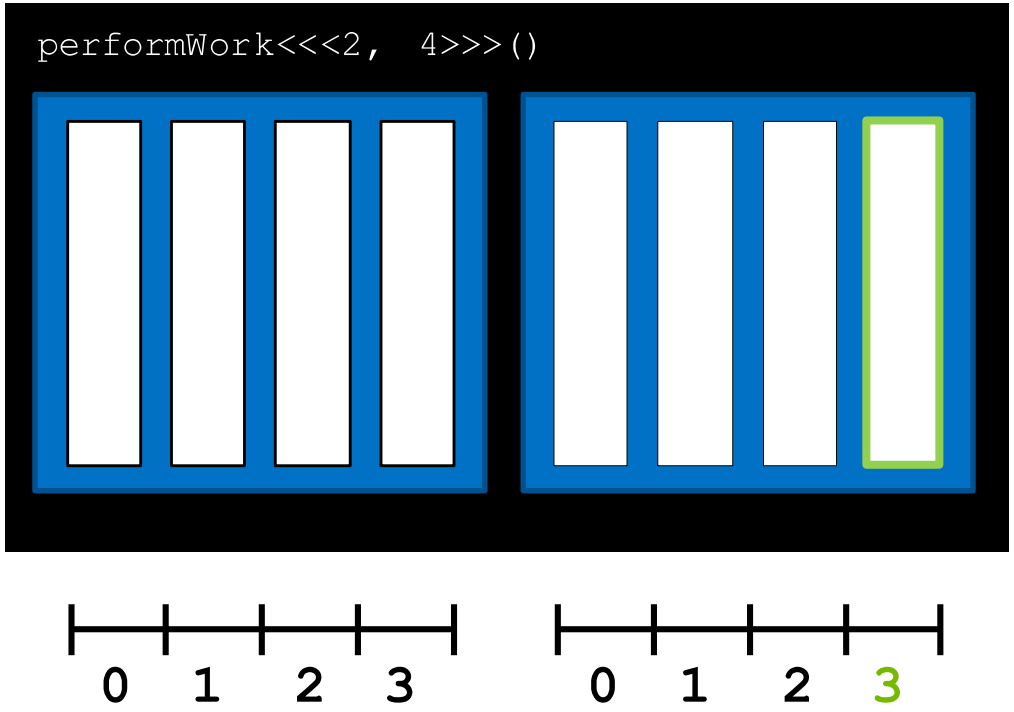
Inside a kernel **threadIdx**
describes the index of the thread
a block. In this case **2**

GPU

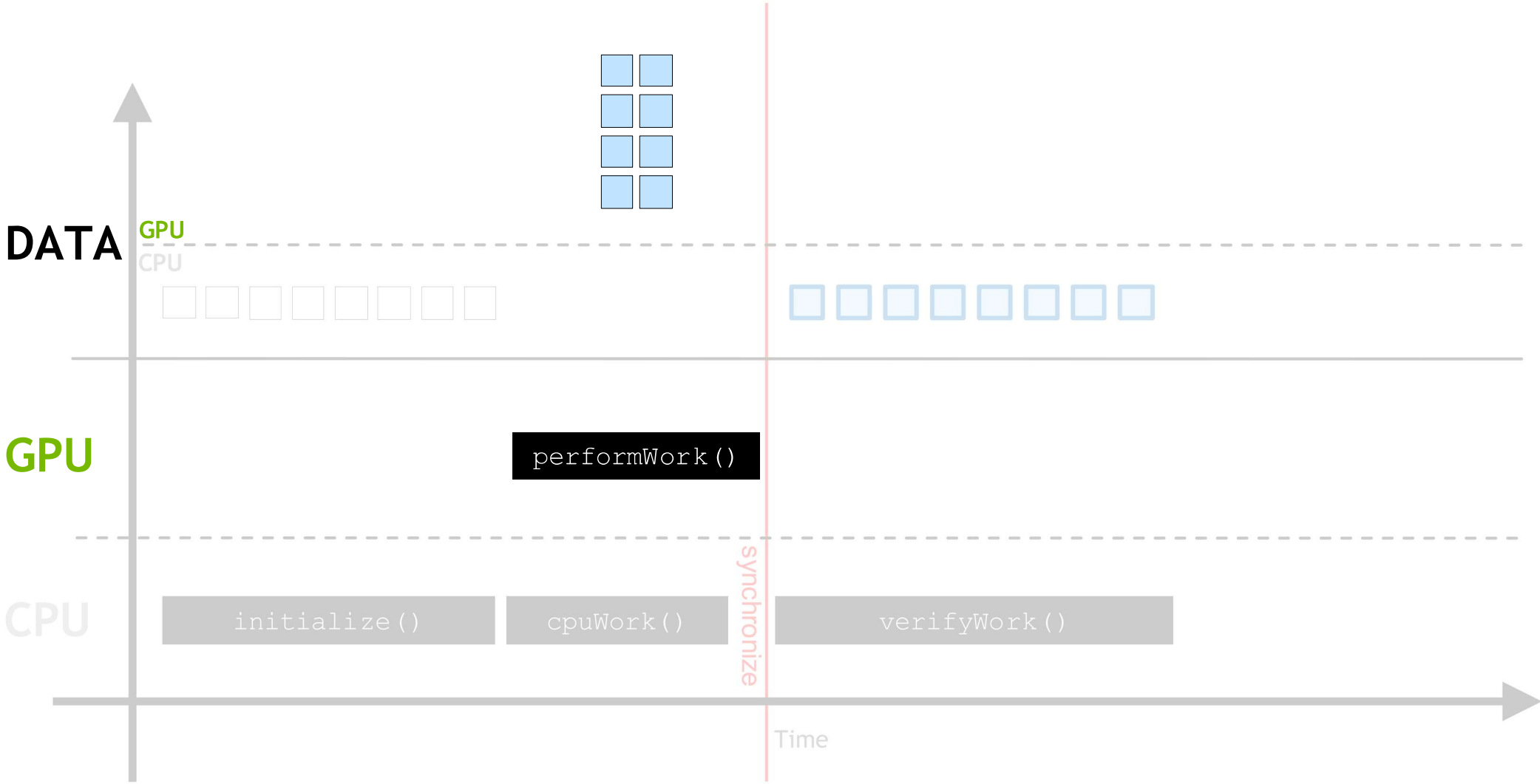


Inside a kernel **threadIdx**
describes the index of the thread
a block. In this case **3**

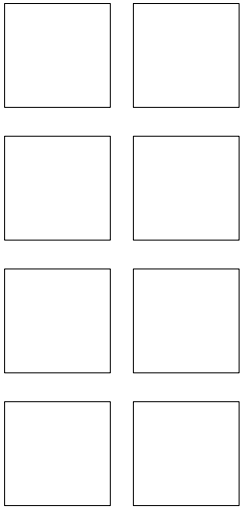
GPU



Coordinating Parallel Threads

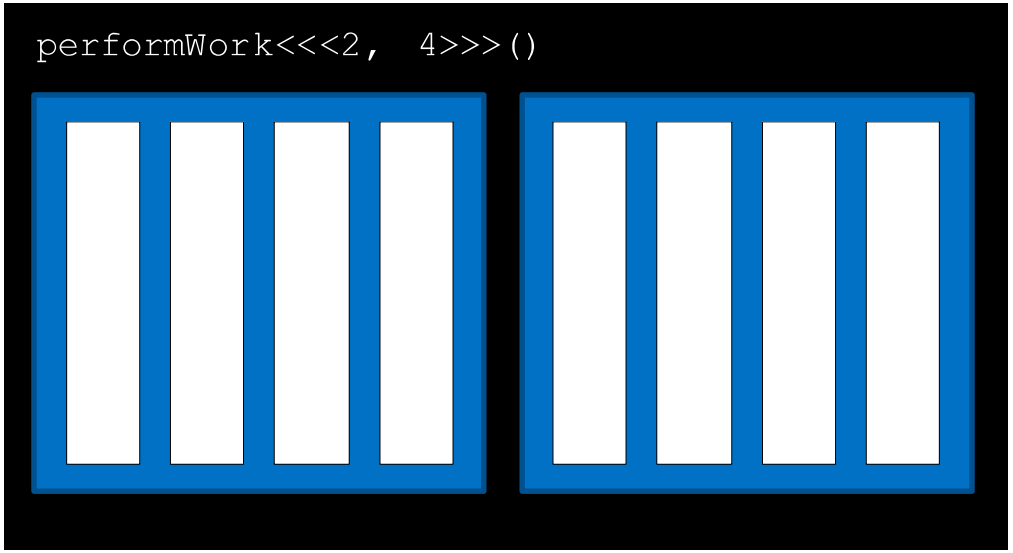

DATA
GPU
CPU
GPU
performWork()
CPU
initialize()
cpuWork()
synchronize
verifyWork()
Time

GPU
DATA



Assume data is in a 0 indexed v

GPU

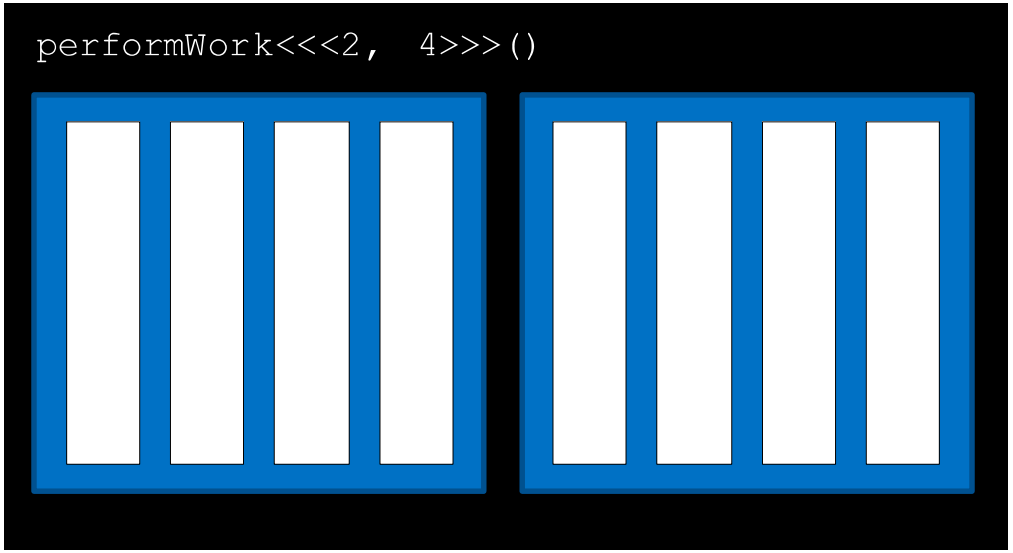


GPU
DATA

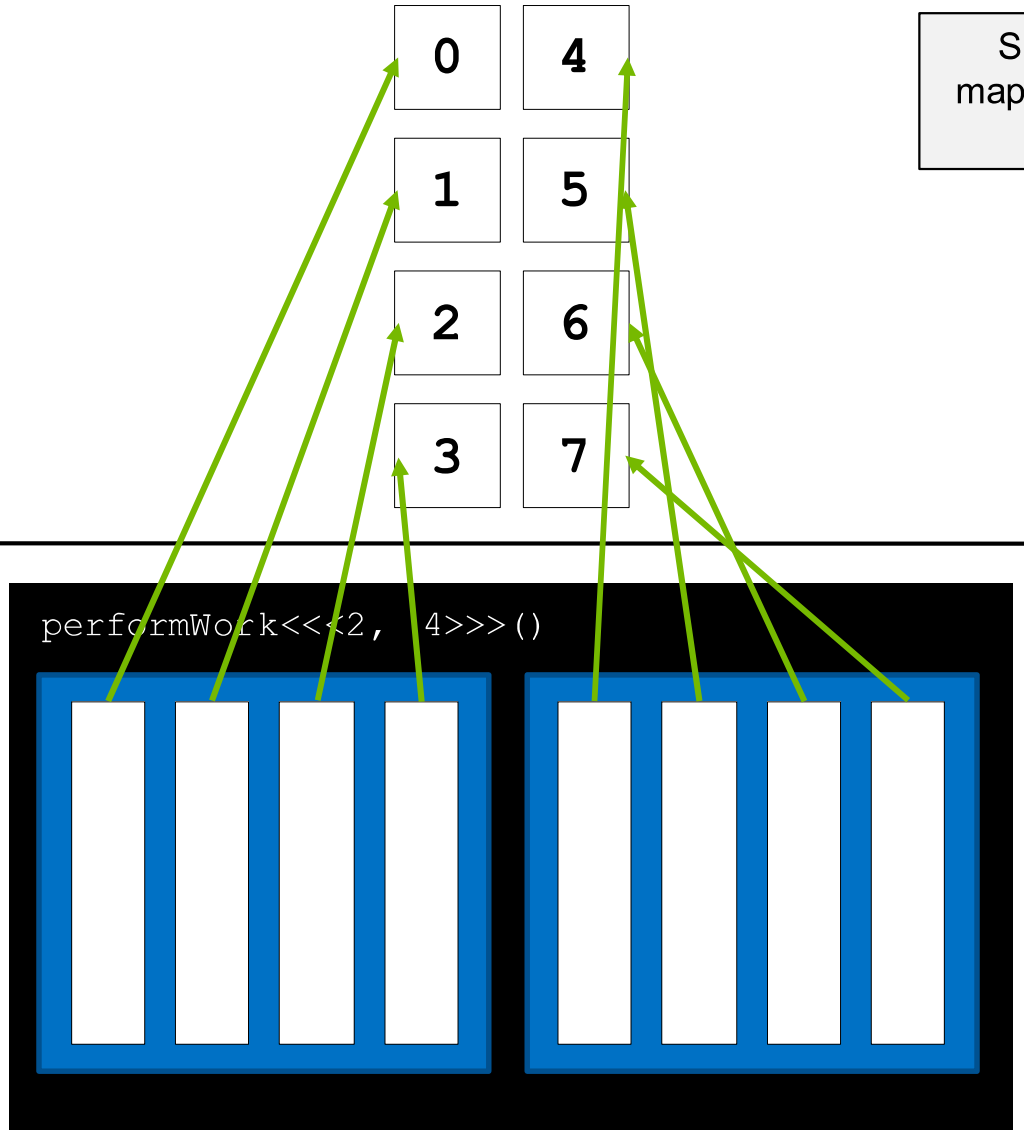
0	4
1	5
2	6
3	7

Assume data is in a 0 indexed v

GPU



GPU
DATA



GPU

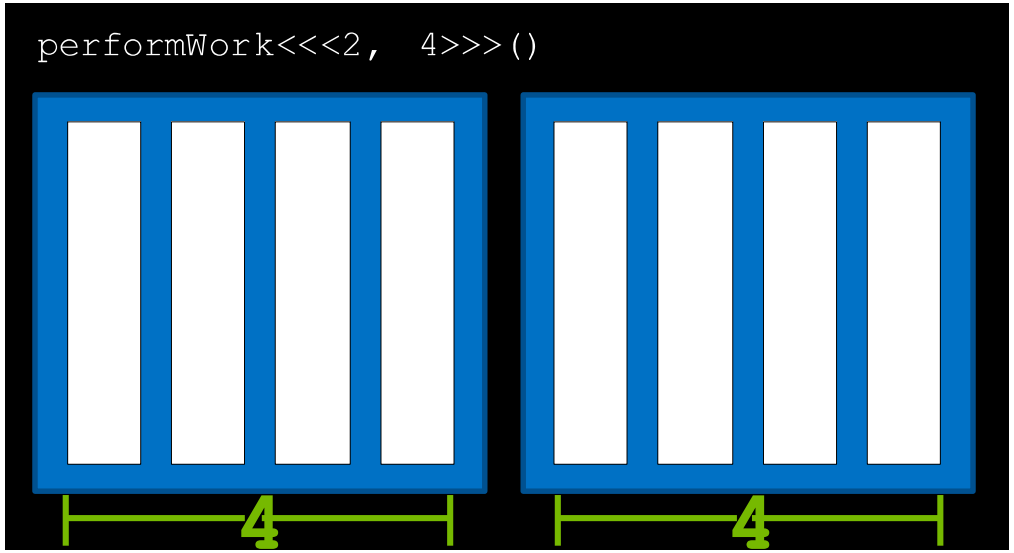
Somehow, each thread must be mapped to work on an element of a vector

GPU
DATA

0	4
1	5
2	6
3	7

Recall that each thread has acc
the size of its block via **blockD**

GPU

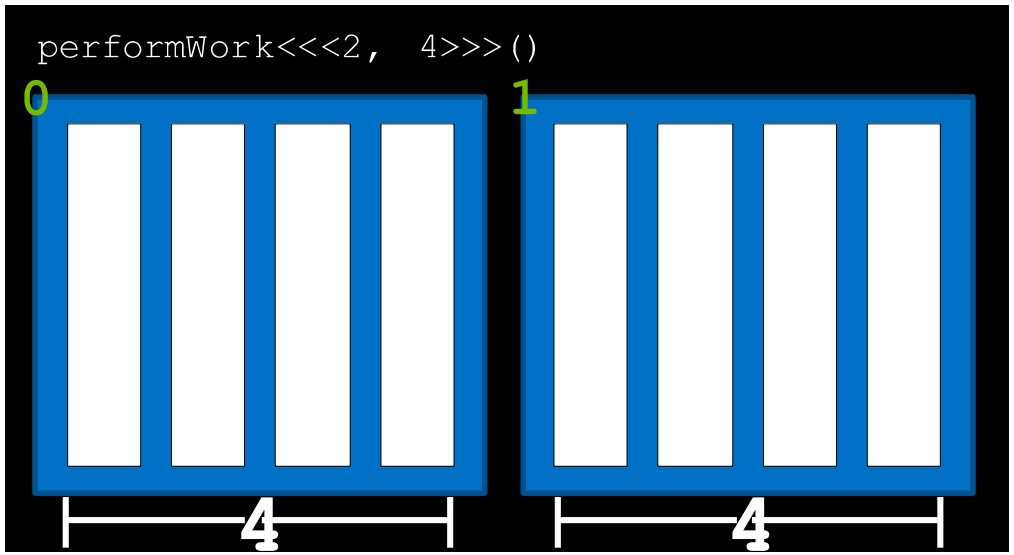


GPU
DATA

0	4
1	5
2	6
3	7

…and the index of its block with
grid via **`blockIdx.x`**

GPU

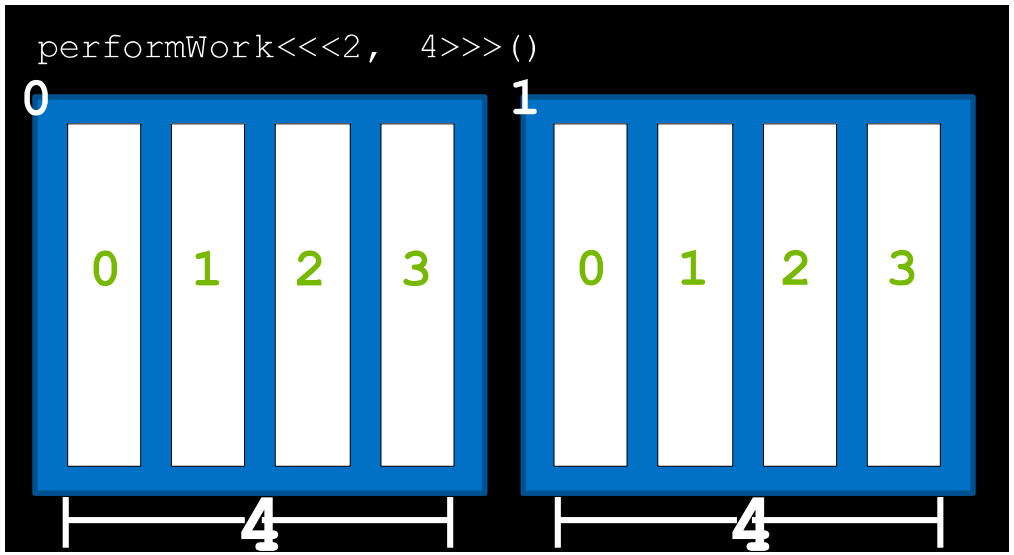


GPU
DATA

0	4
1	5
2	6
3	7

…and its own index within its blo
threadIdx.x

GPU

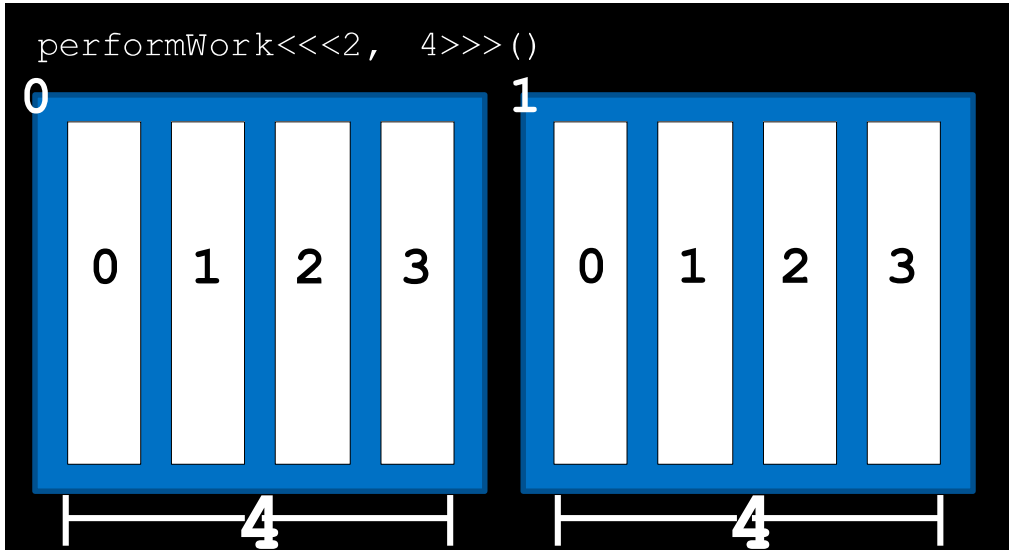


GPU
DATA

0	4
1	5
2	6
3	7

Using these variables, the form
`threadIdx.x + blockIdx.`
`blockDim.x` will map each thre
one element in the vector

GPU



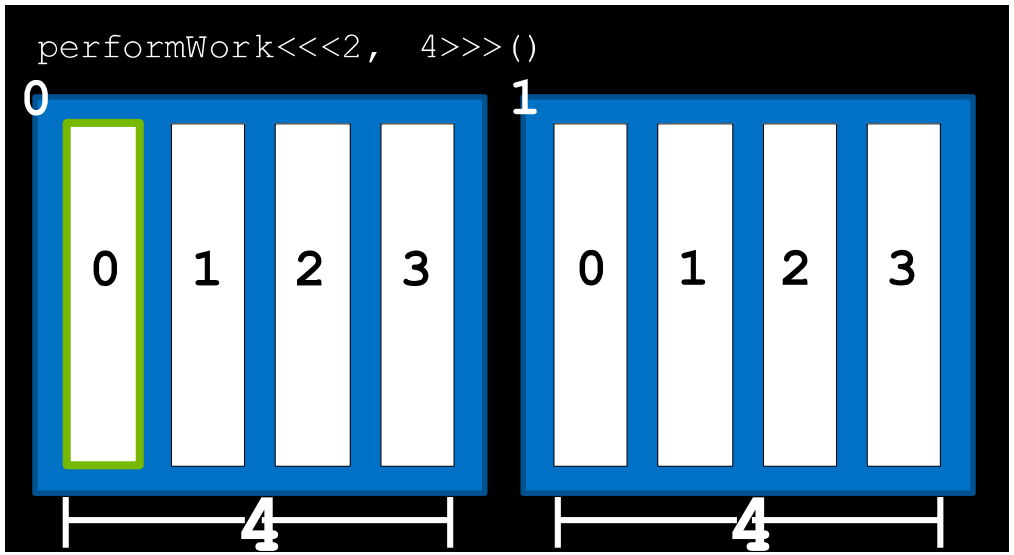
GPU
DATA

0	4
1	5
2	6
3	7

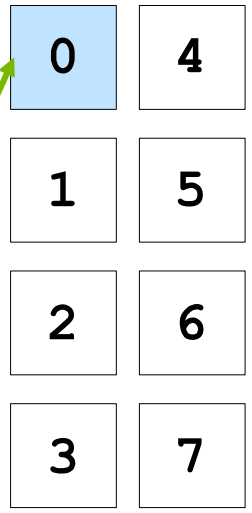
threadIdx.x	+	blockIdx.x	*	blockD
0		0		

dataIndex
?

GPU



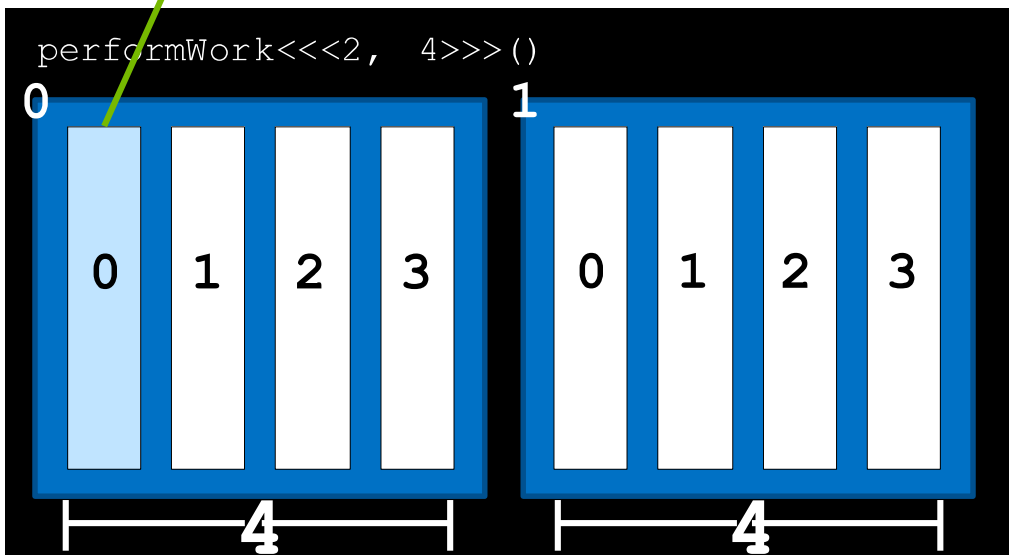
GPU
DATA



threadIdx.x	+	blockIdx.x	*	blockD
0		0		

dataIndex
0

GPU



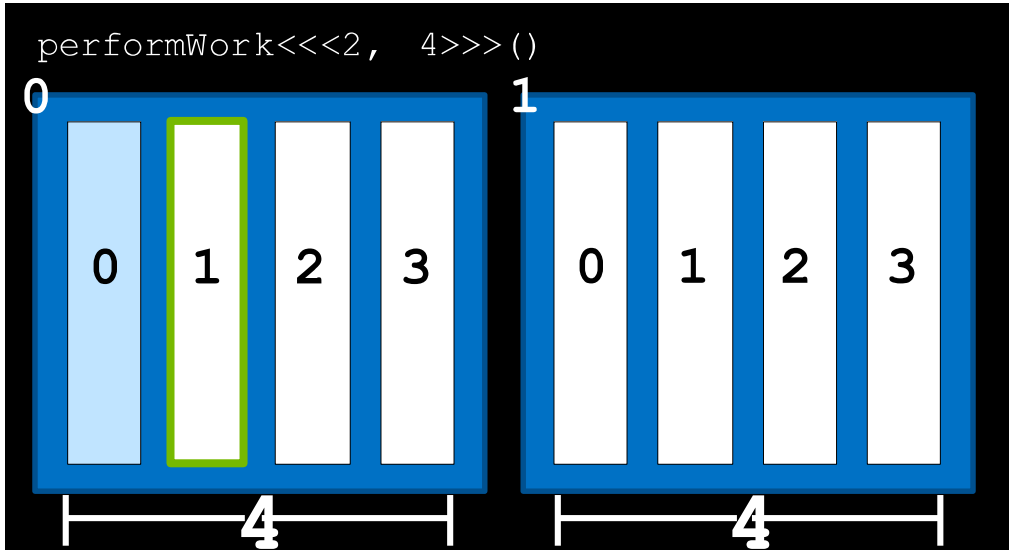
GPU
DATA

0	4
1	5
2	6
3	7

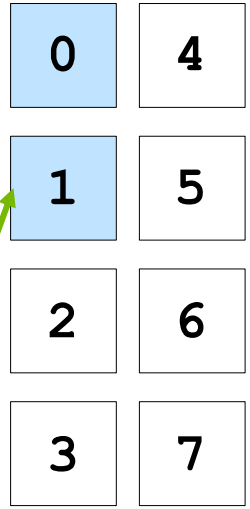
threadIdx.x	+	blockIdx.x	*	blockD
1		0		

dataIndex
?

GPU



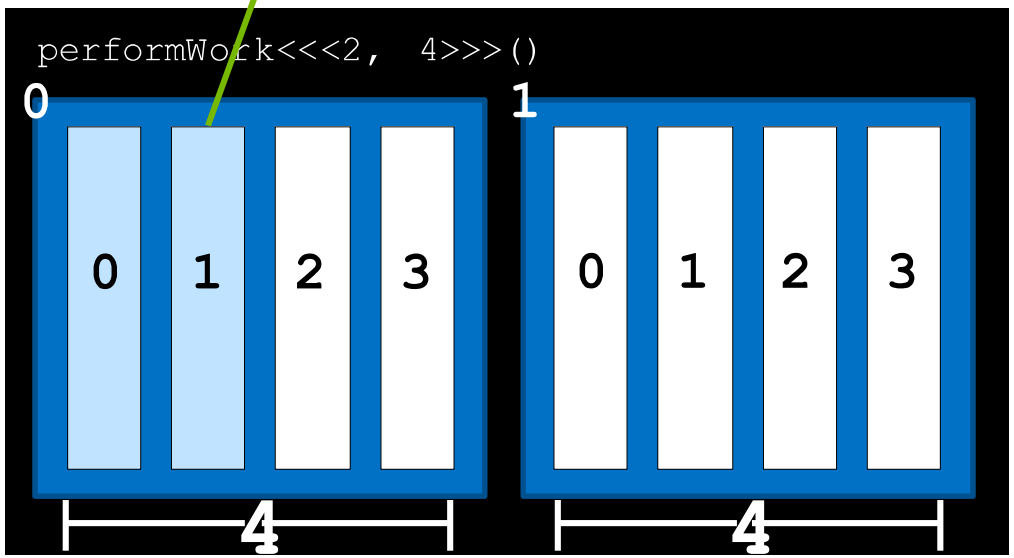
GPU
DATA



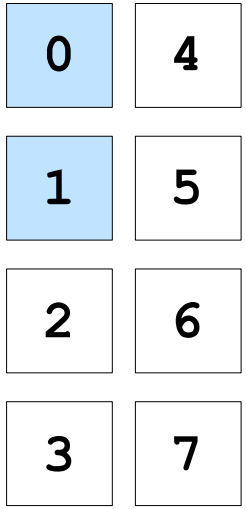
threadIdx.x	+	blockIdx.x	*	blockD
1		0		

dataIndex
1

GPU



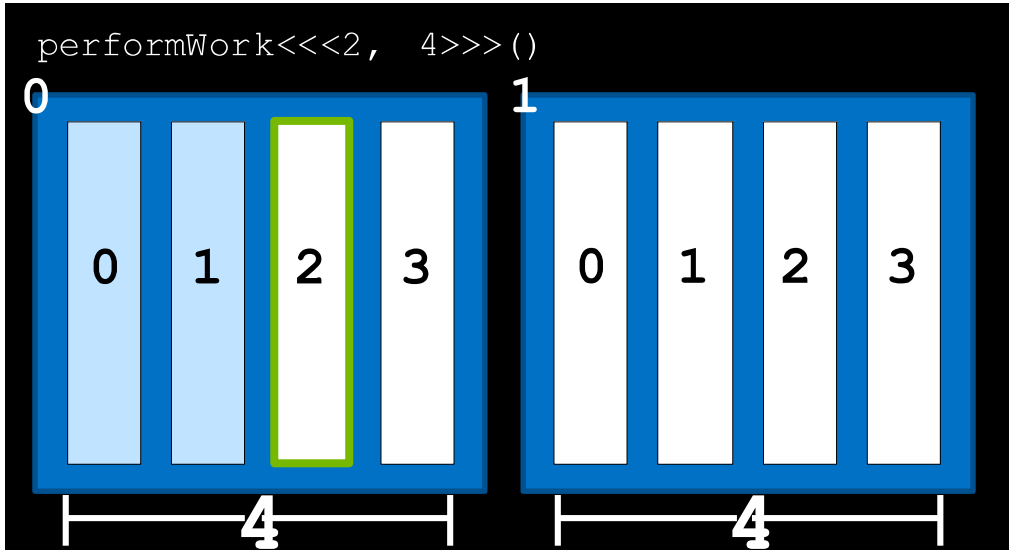
GPU
DATA



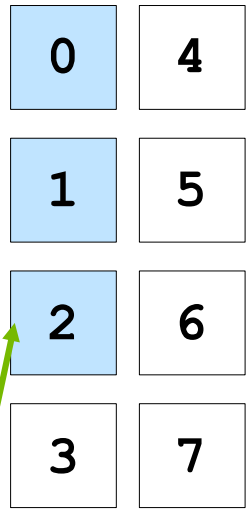
threadIdx.x	+	blockIdx.x	*	blockD
2		0		

dataIndex
?

GPU



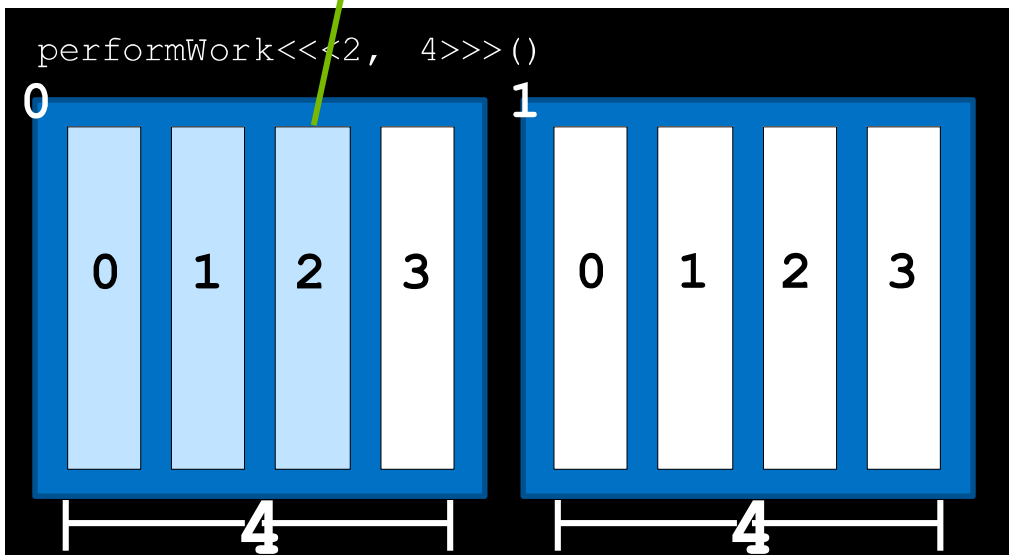
GPU
DATA



threadIdx.x	+	blockIdx.x	*	blockD
2		0		

dataIndex
2

GPU



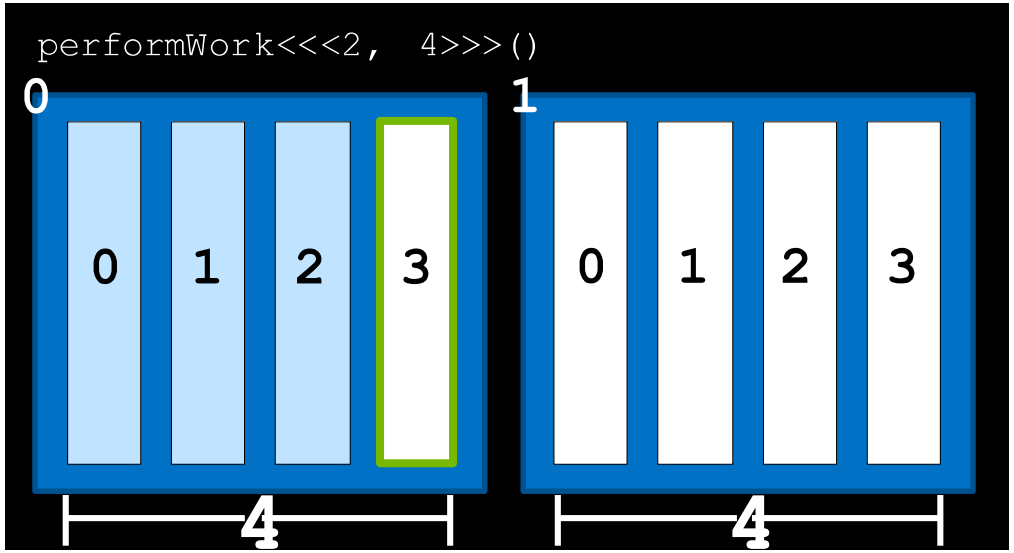
GPU
DATA

0	4
1	5
2	6
3	7

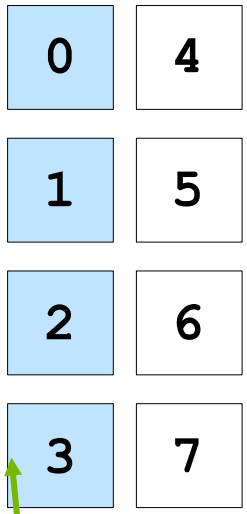
threadIdx.x	+	blockIdx.x	*	blockD
3		0		

dataIndex
?

GPU



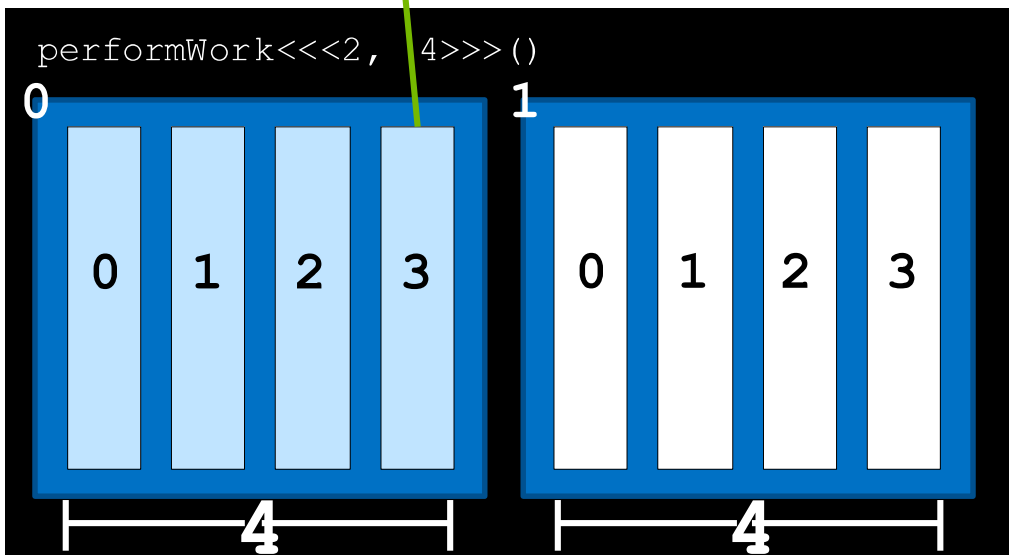
GPU
DATA



threadIdx.x	+	blockIdx.x	*	blockD
3		0		

dataIndex
3

GPU



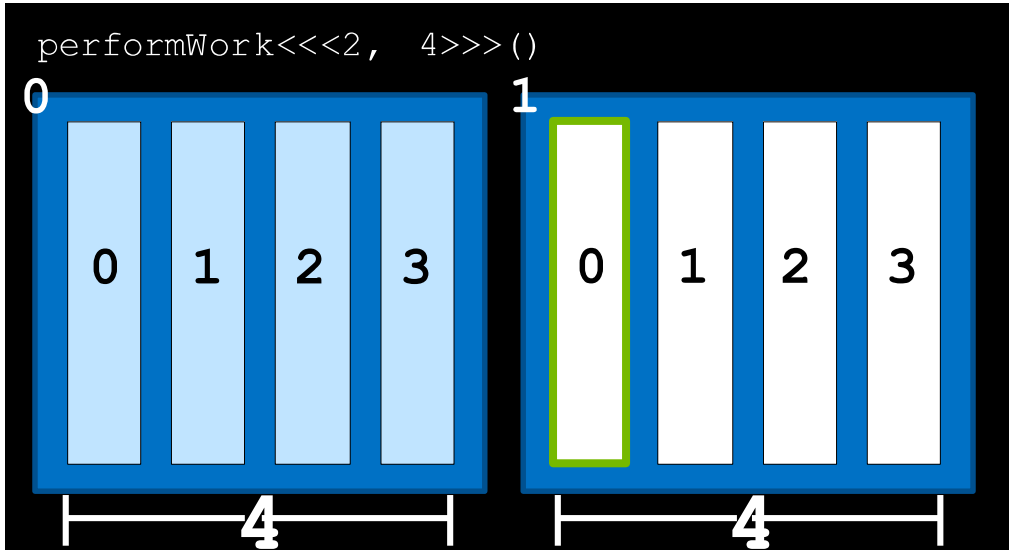
GPU
DATA

0	4
1	5
2	6
3	7

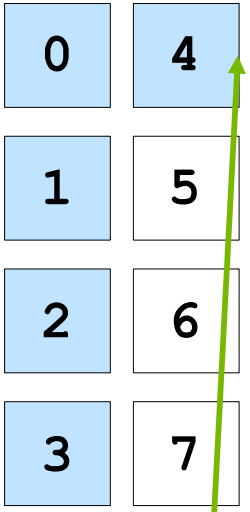
threadIdx.x	+	blockIdx.x	*	blockD
0		1		

dataIndex
?

GPU



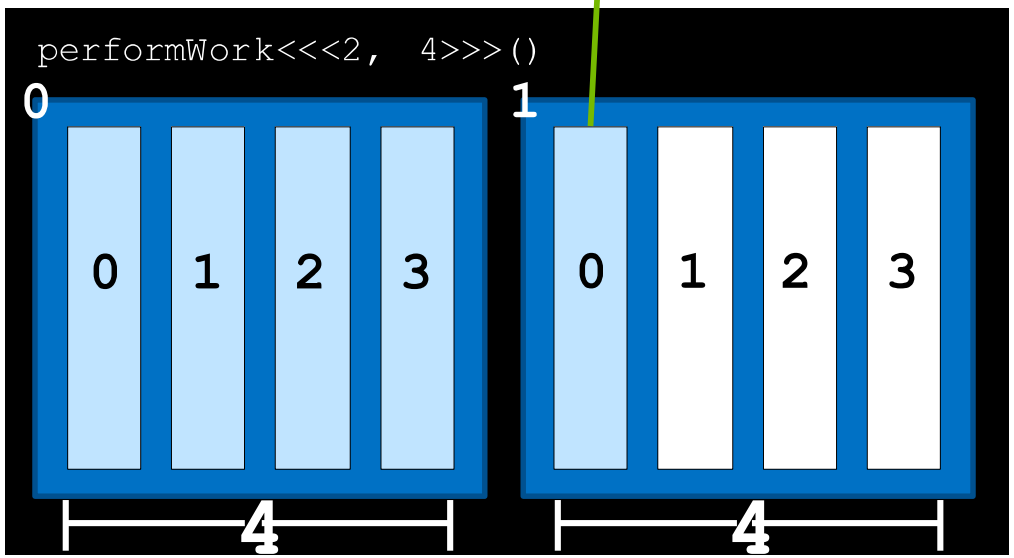
GPU
DATA



threadIdx.x	+	blockIdx.x	*	blockD
0		1		

dataIndex
4

GPU



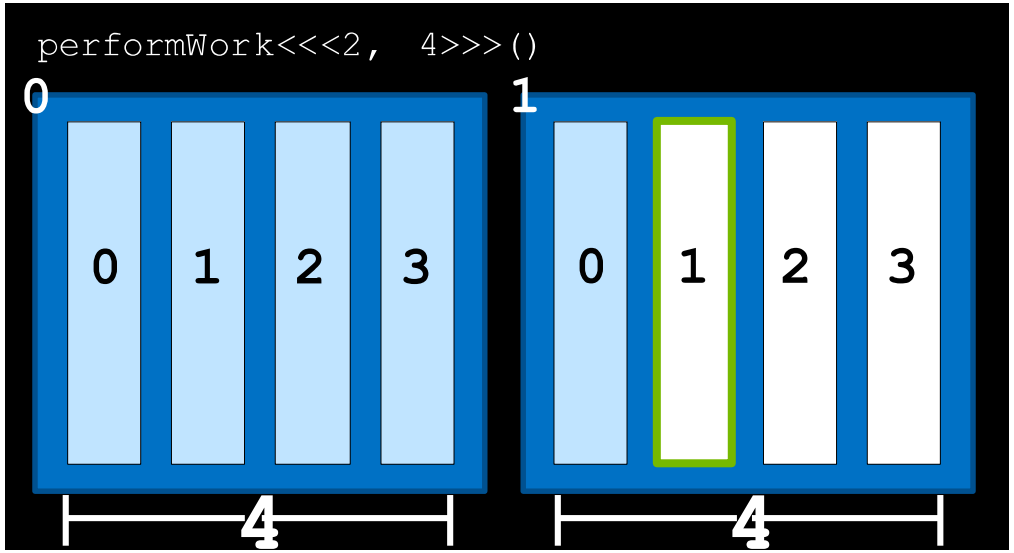
GPU
DATA

0	4
1	5
2	6
3	7

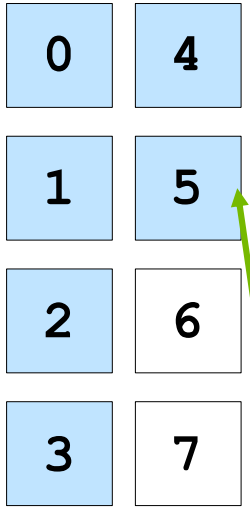
threadIdx.x	+	blockIdx.x	*	blockD
1		1		

dataIndex
?

GPU



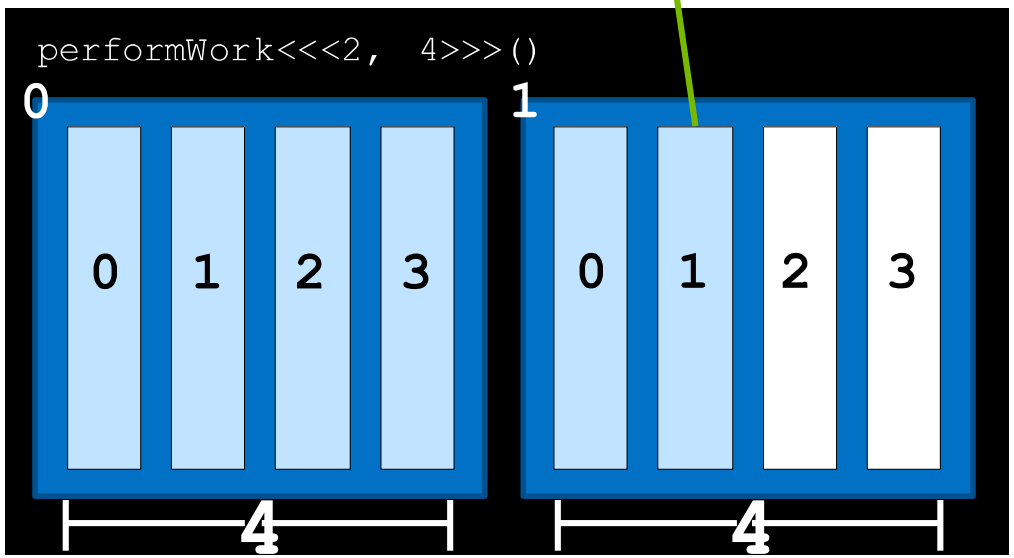
GPU
DATA



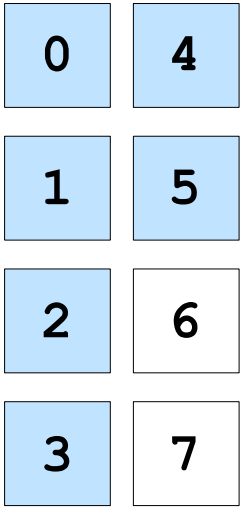
threadIdx.x	+	blockIdx.x	*	blockD
1		1		

dataIndex
5

GPU



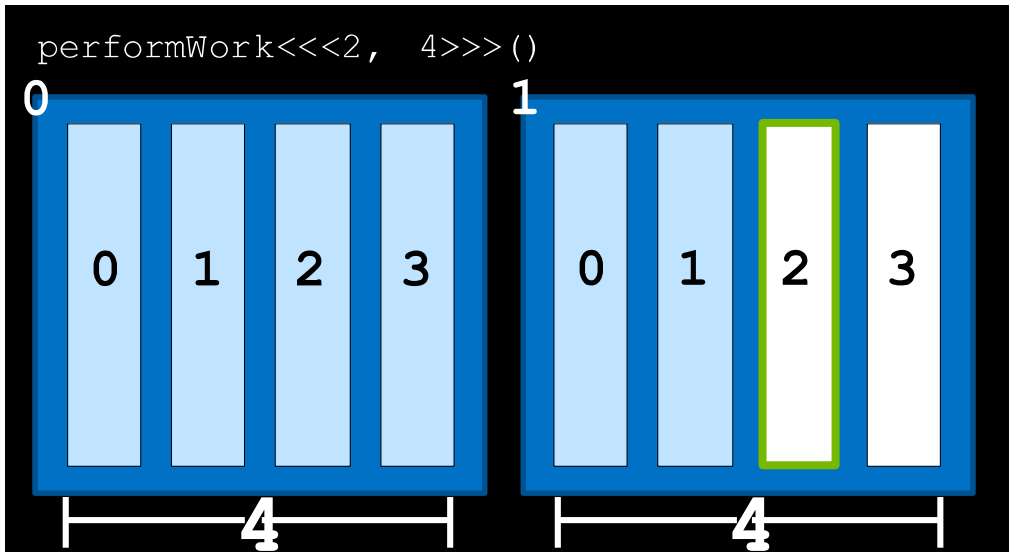
GPU
DATA



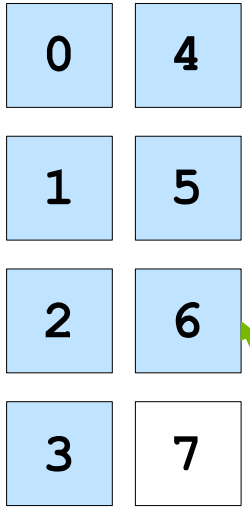
threadIdx.x	+	blockIdx.x	*	blockD
2		1		

dataIndex
?

GPU



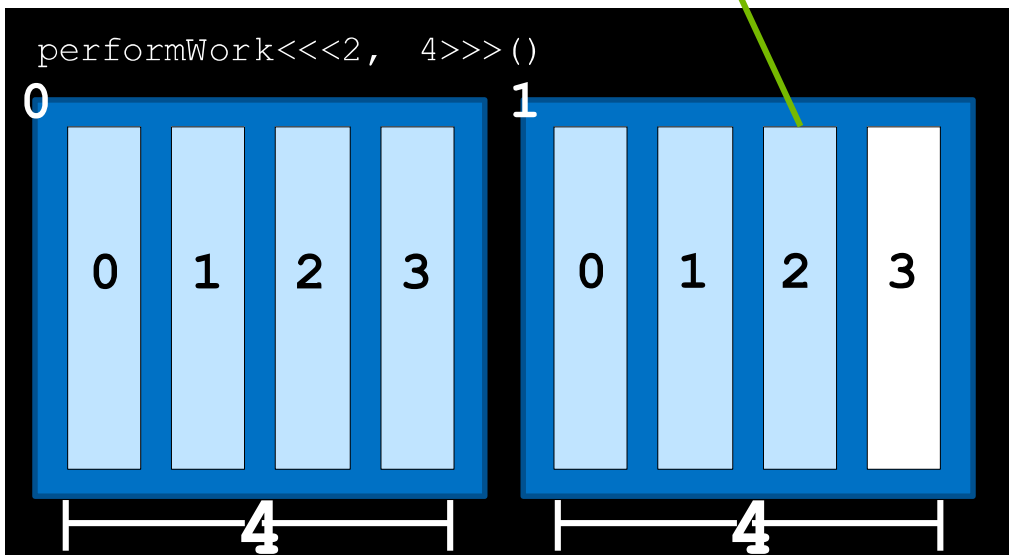
GPU
DATA



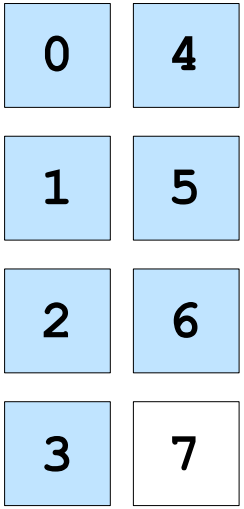
threadIdx.x	+	blockIdx.x	*	blockD
2		1		

dataIndex
6

GPU



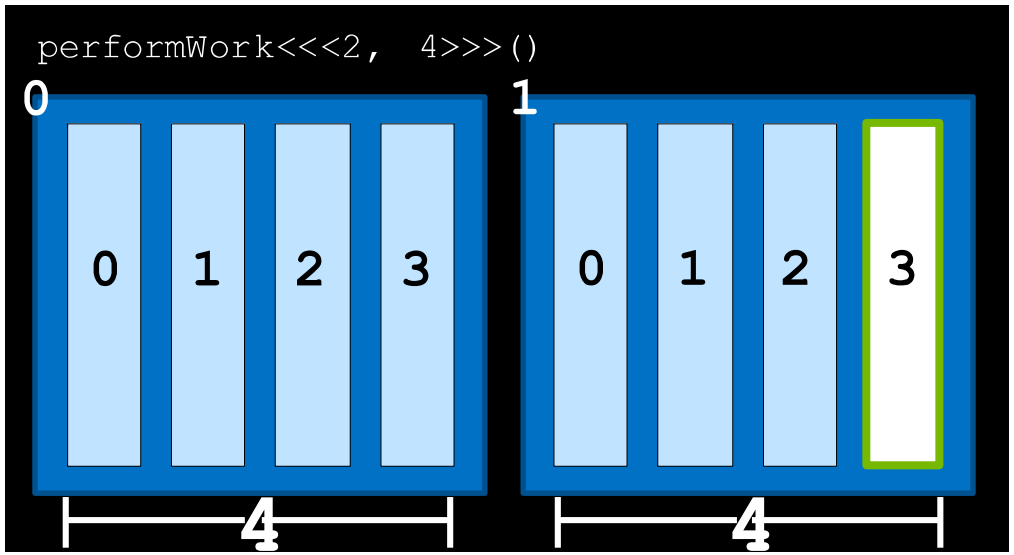
GPU
DATA



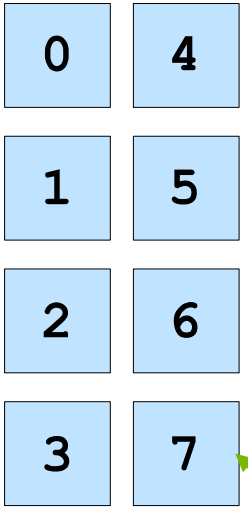
threadIdx.x	+	blockIdx.x	*	blockD
3		1		

dataIndex
?

GPU



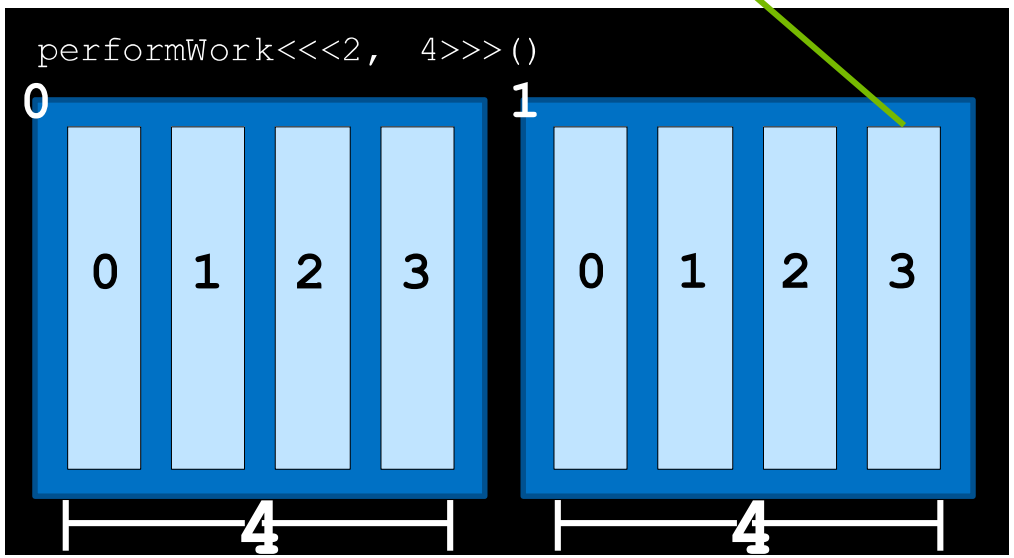
GPU
DATA



threadIdx.x	+	blockIdx.x	*	blockD
3		1		

dataIndex
7

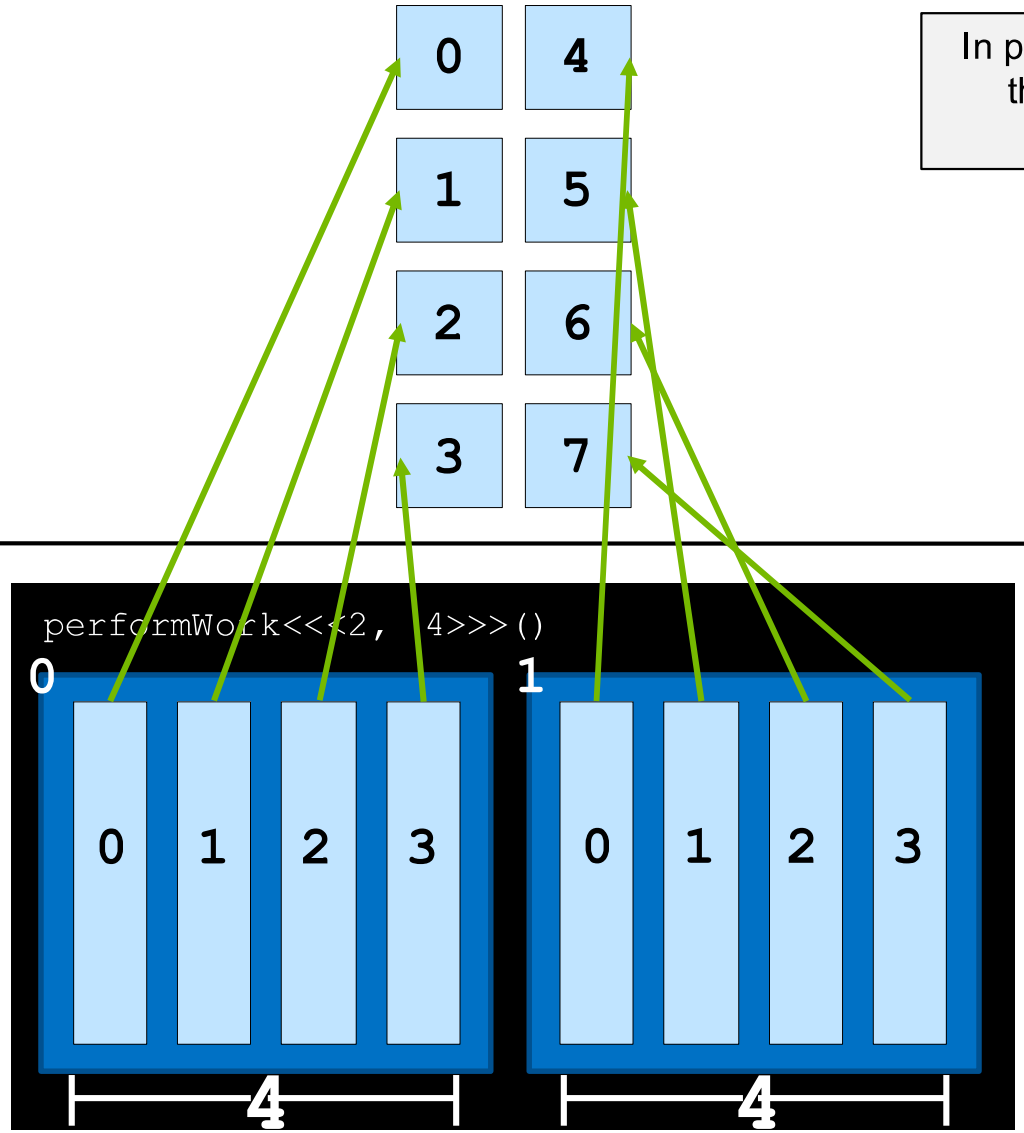
GPU



Grid Size Work Amount Mismatch

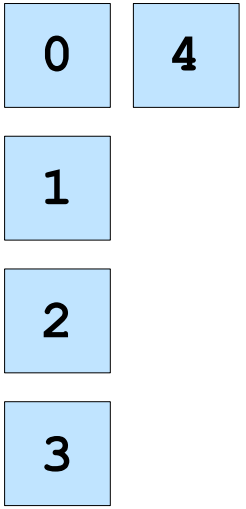
In previous scenarios, the numb
threads in the grid matched t
number of elements exactl

GPU
DATA



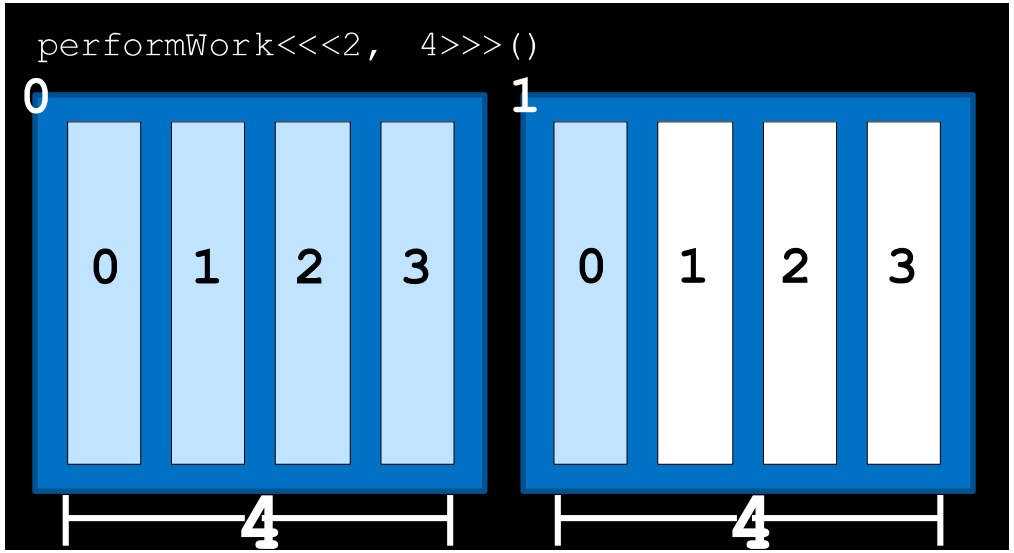
GPU

GPU
DATA

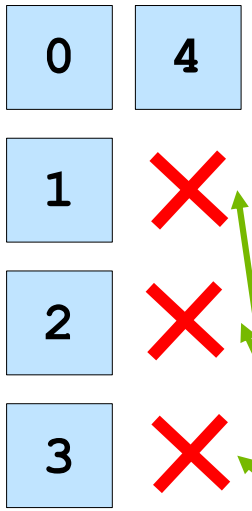


What if there are more threads than work to be done?

GPU

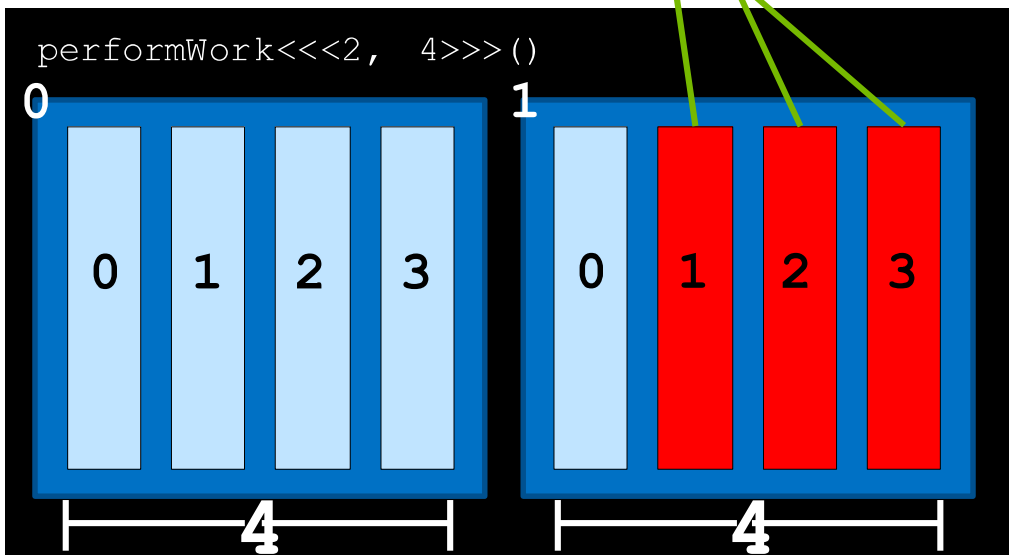


GPU
DATA

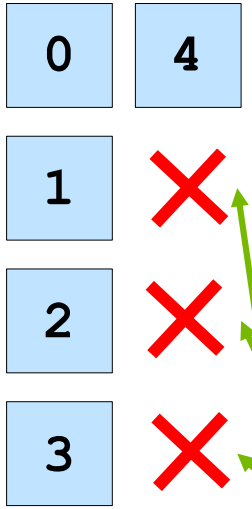


Attempting to access non-exis
elements can result in a runtime

GPU

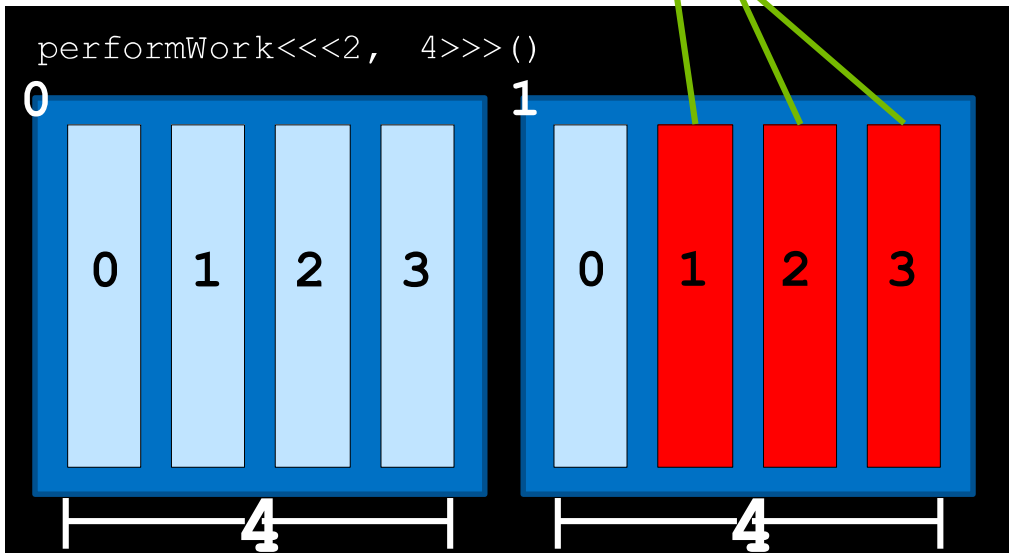


GPU
DATA

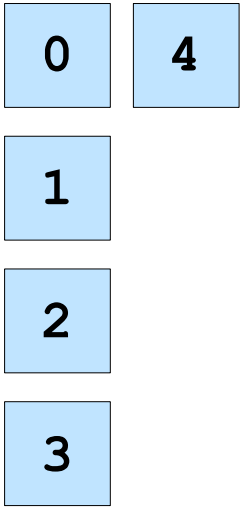


Code must check that the **data**
calculated by **threadIdx.x**
blockIdx.x * blockDim.x
than **N**, the number of data elem

GPU



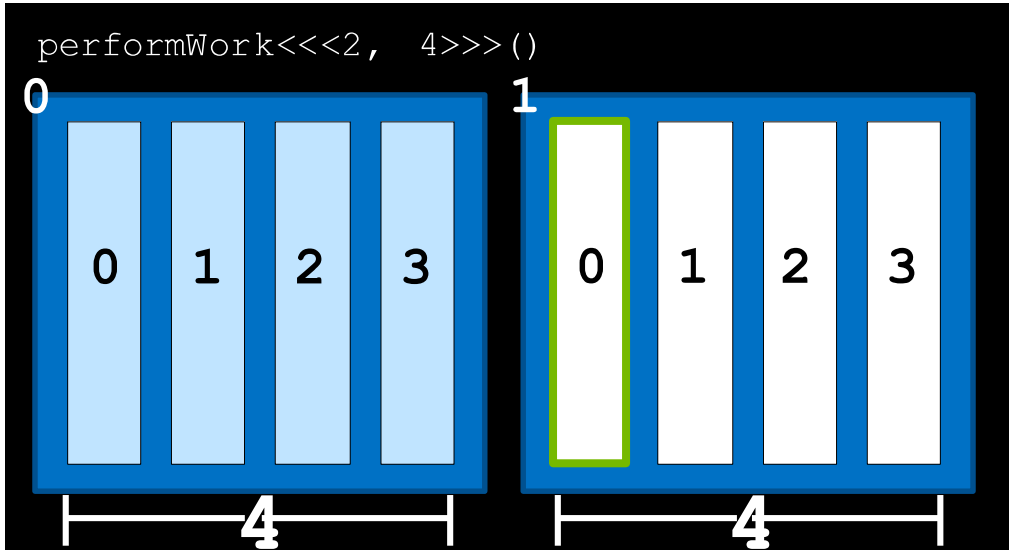
GPU
DATA



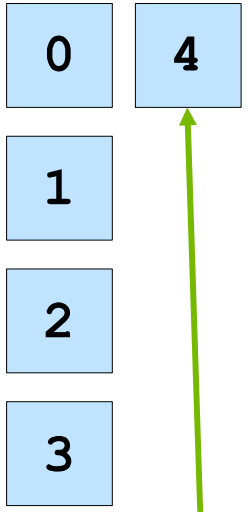
threadIdx.x	+	blockIdx.x	*	blockD
0		1		

dataIndex	<	N	=	Can
4		5		

GPU



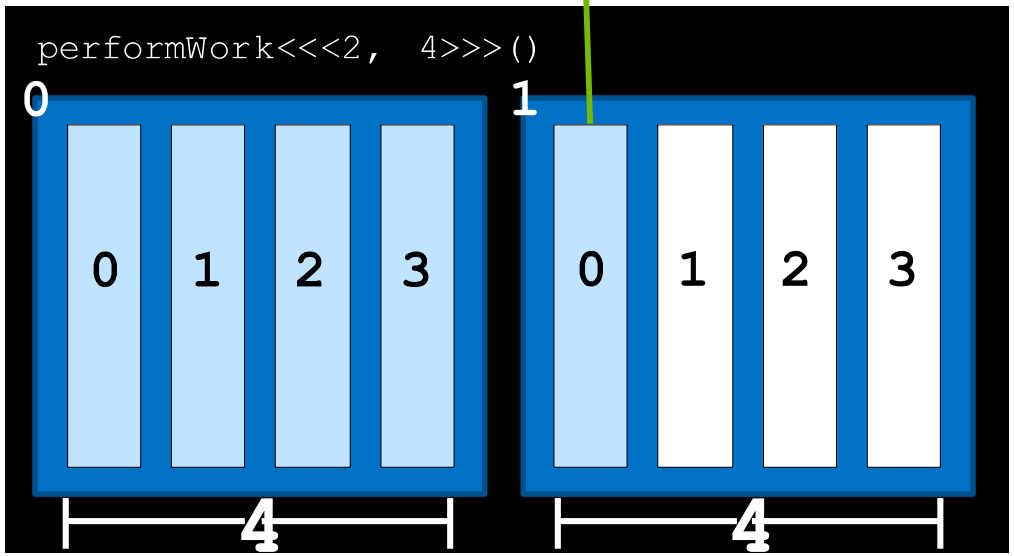
GPU
DATA



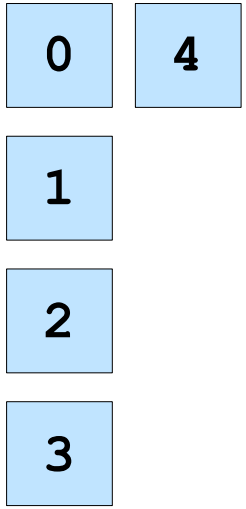
threadIdx.x	+	blockIdx.x	*	blockD
0		1		

dataIndex	<	N	=	Can
4		5		tr

GPU



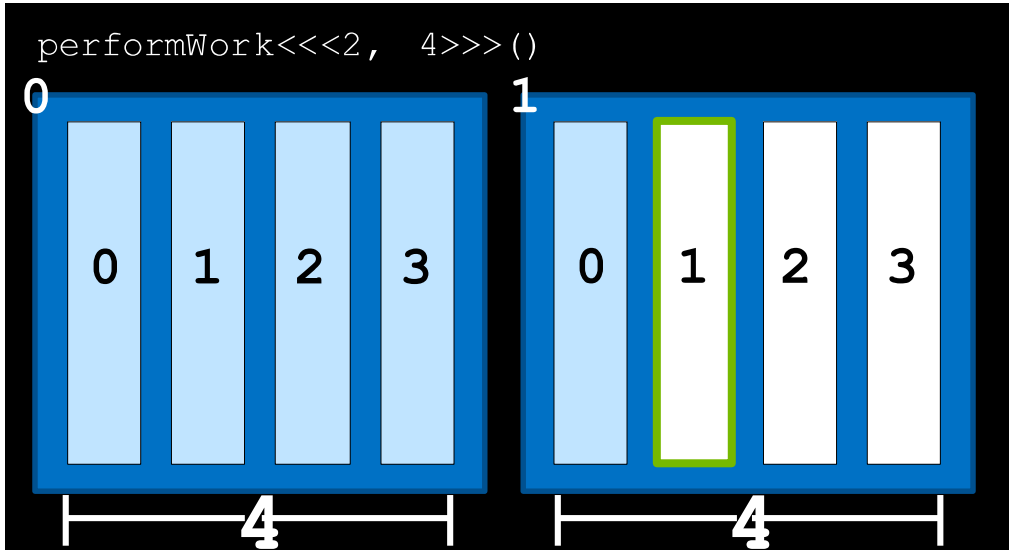
GPU
DATA



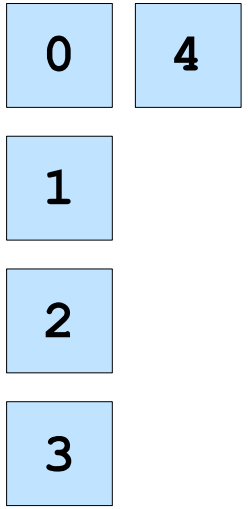
threadIdx.x	+	blockIdx.x	*	blockD
1		1		

dataIndex	<	N	=	Can
5		5		

GPU



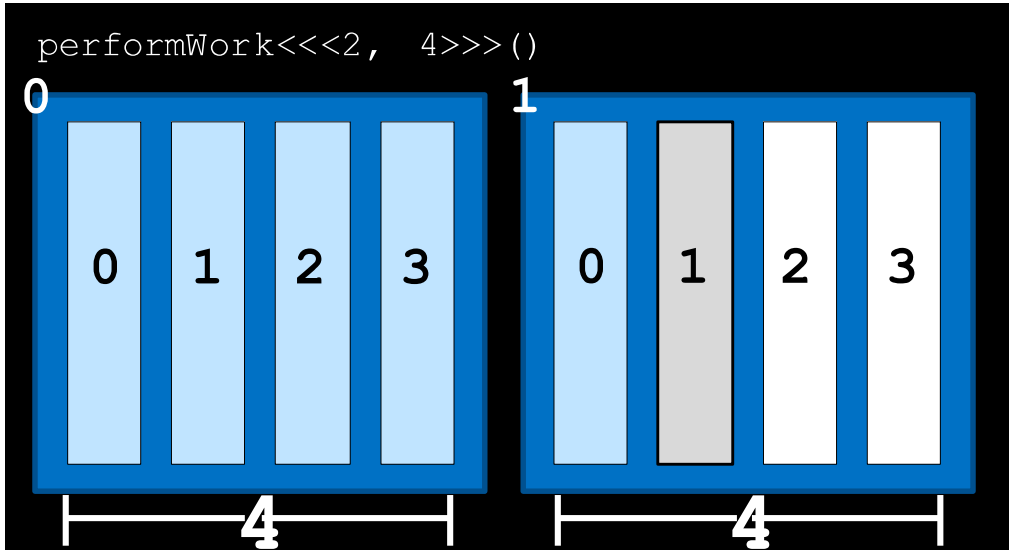
GPU
DATA



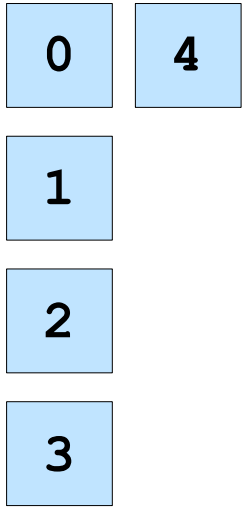
threadIdx.x	+	blockIdx.x	*	blockD
1		1		

dataIndex	<	N	=	Can
5		5		fa

GPU



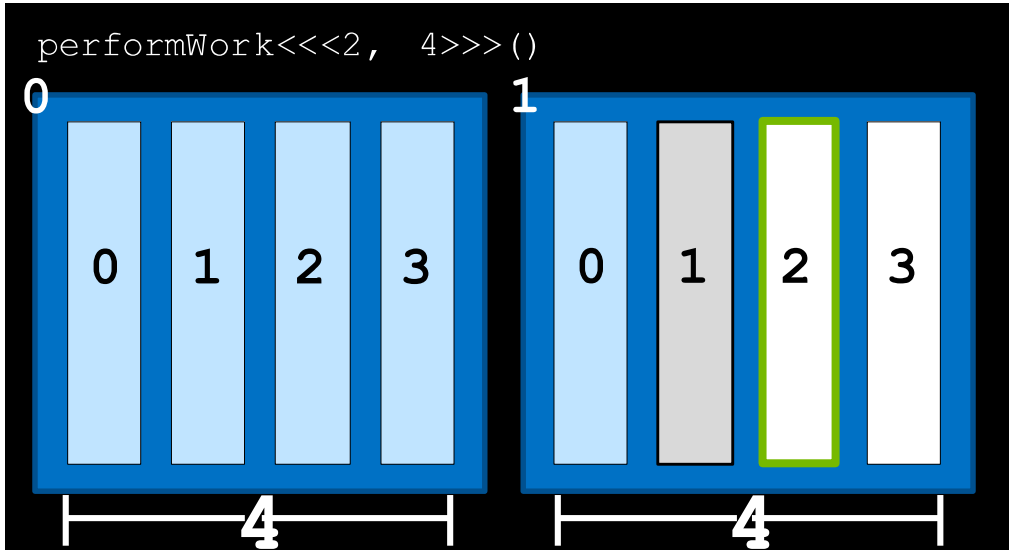
GPU
DATA



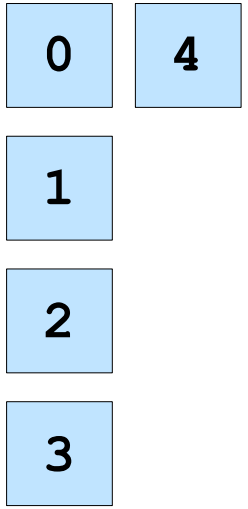
threadIdx.x	+	blockIdx.x	*	blockD
2		1		

dataIndex	<	N	=	Can
6		5		

GPU



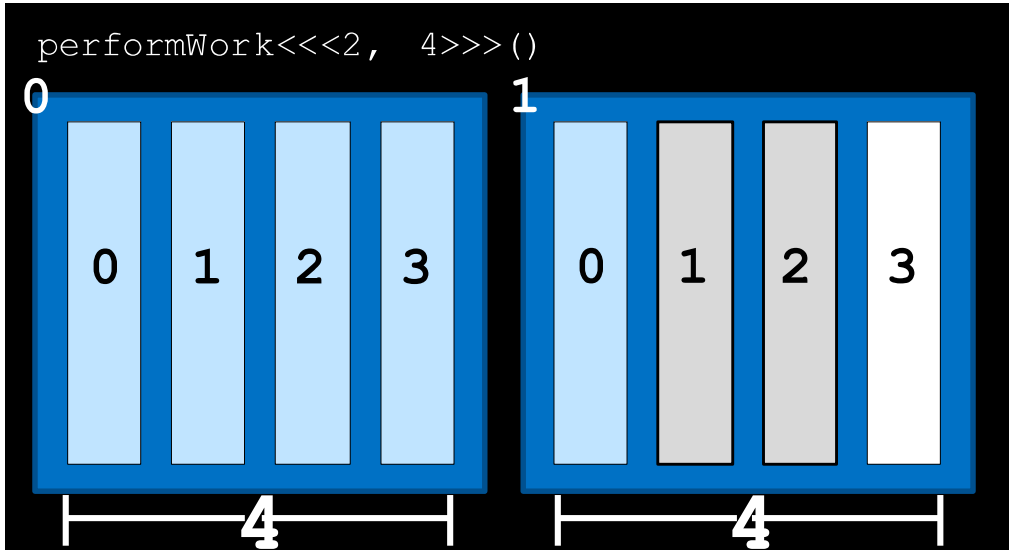
GPU
DATA



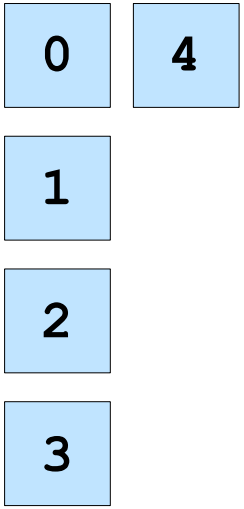
threadIdx.x	+	blockIdx.x	*	blockD
2		1		

dataIndex	<	N	=	Can
6		5		fa

GPU



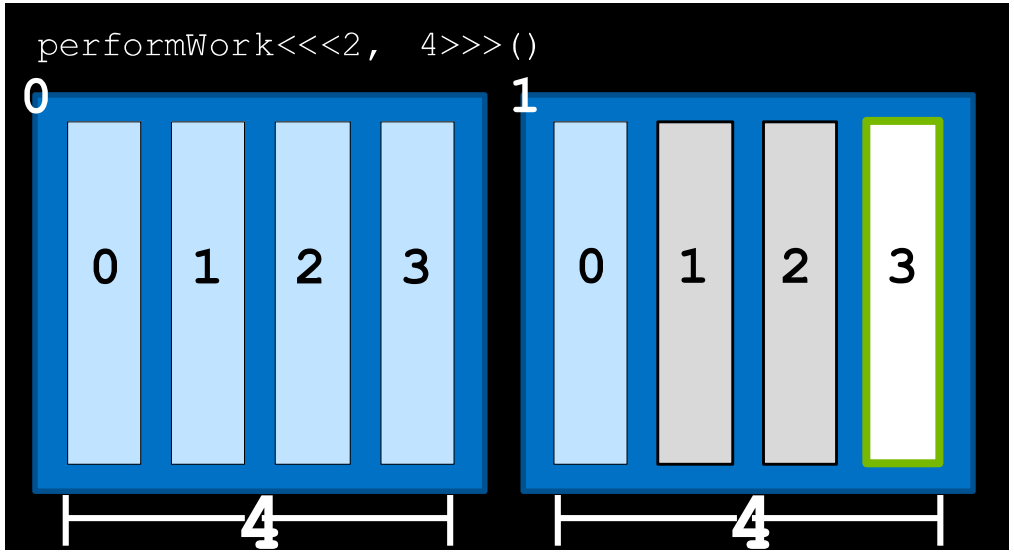
GPU
DATA



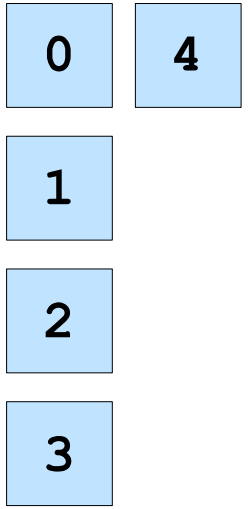
threadIdx.x	+	blockIdx.x	*	blockD
2		1		

dataIndex	<	N	=	Can
6		5		

GPU



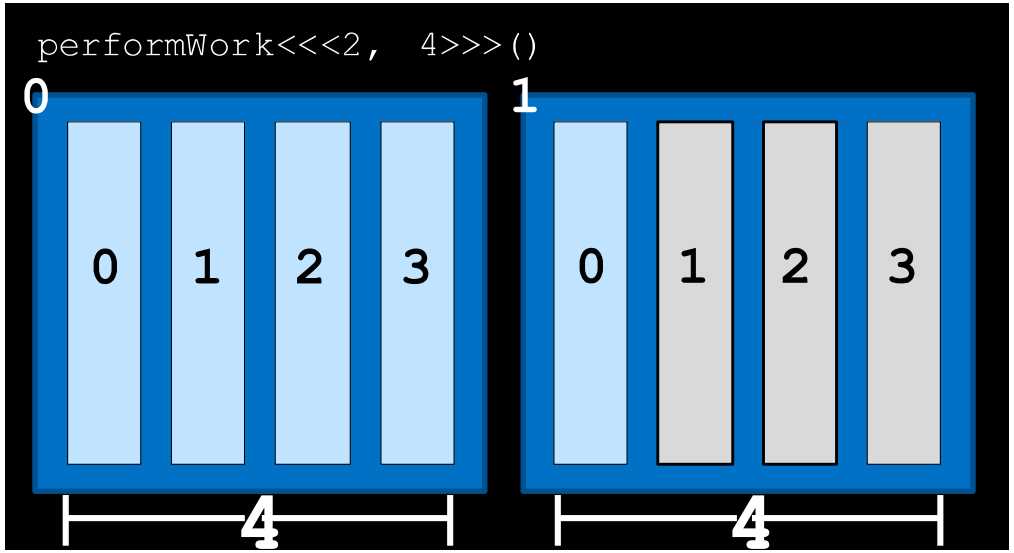
GPU
DATA



threadIdx.x	+	blockIdx.x	*	blockD
2		1		

dataIndex	<	N	=	Can
6		5		fa

GPU



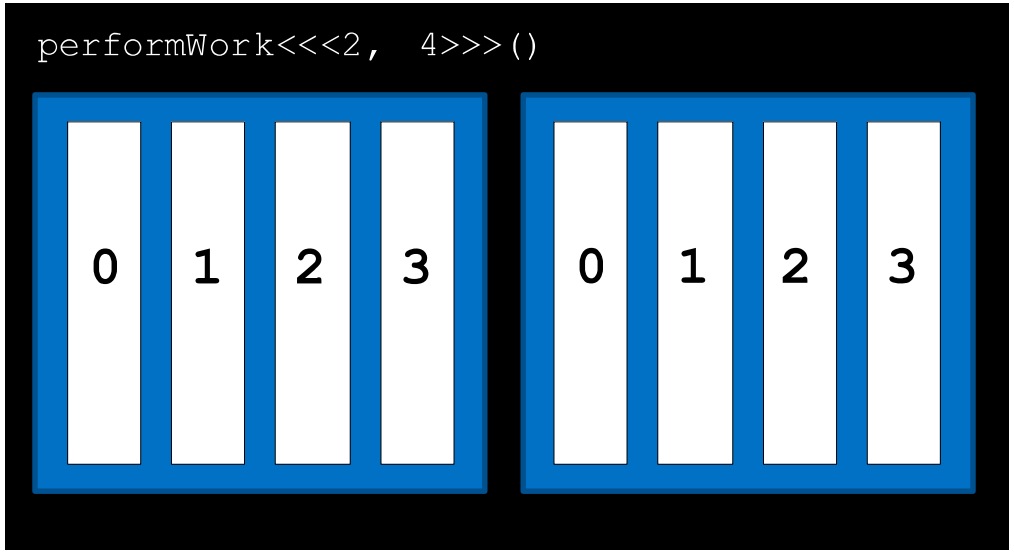
Grid-Stride Loops

GPU
DATA

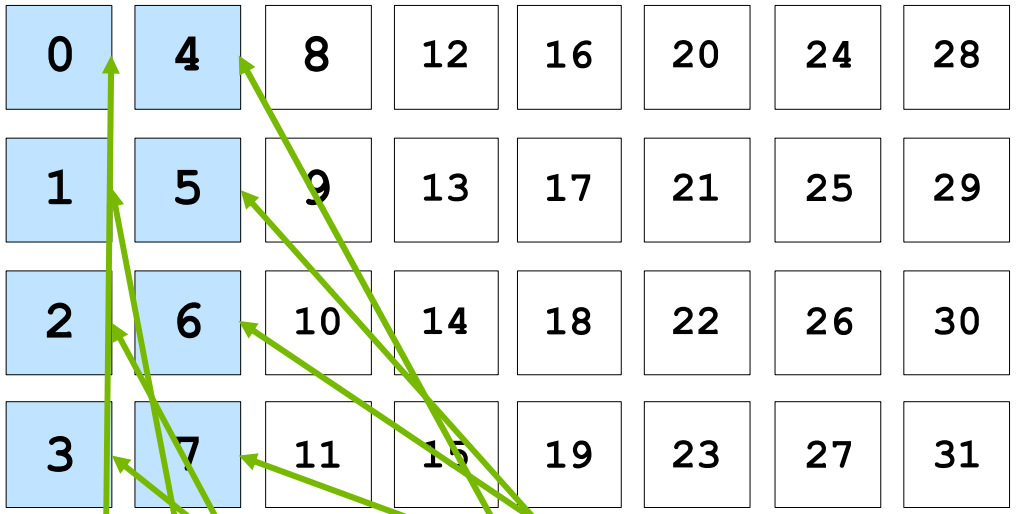
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

Often there are more elements than there threads in the gri

GPU

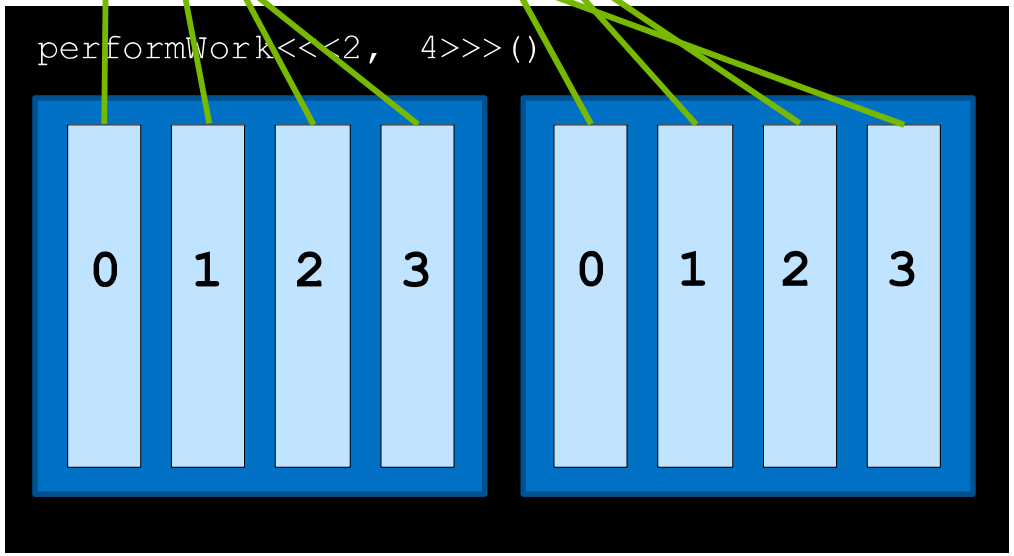


GPU
DATA

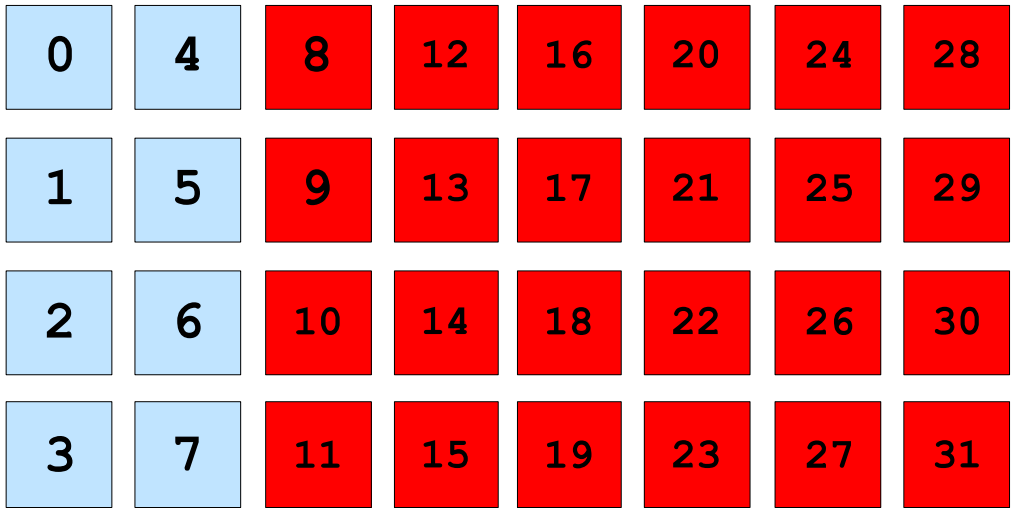


In such scenarios thr
cannot work on only
element

GPU

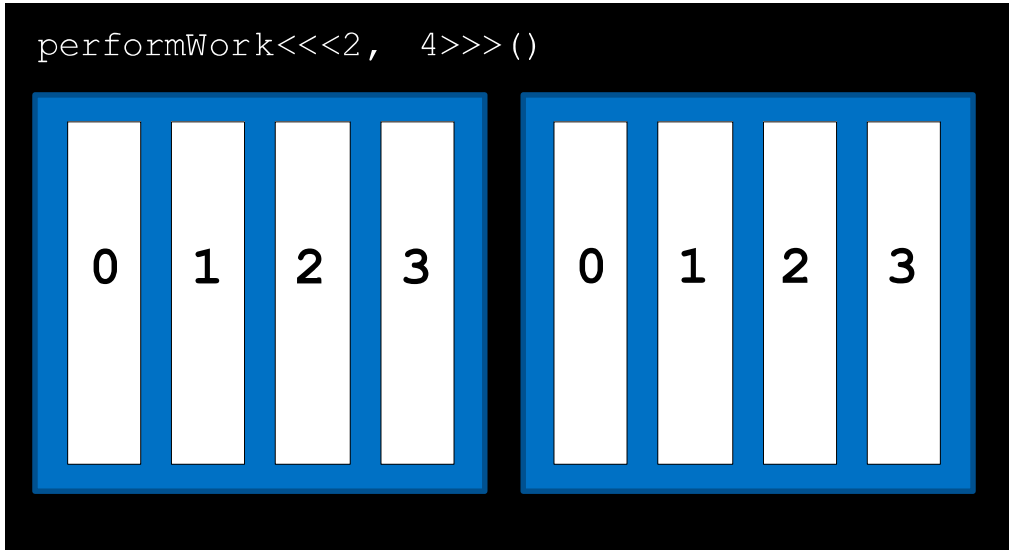


GPU
DATA



… or else work is undone

GPU

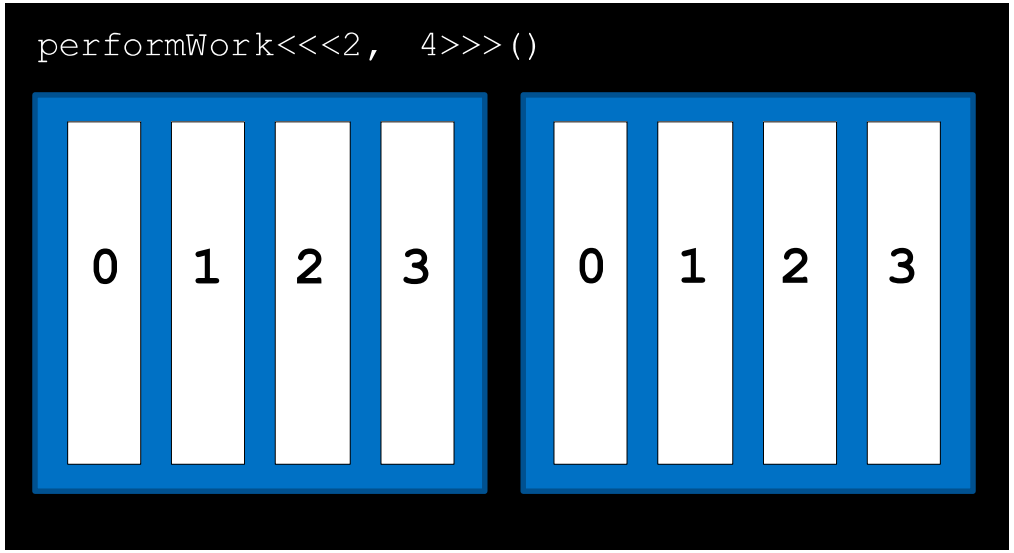


GPU
DATA

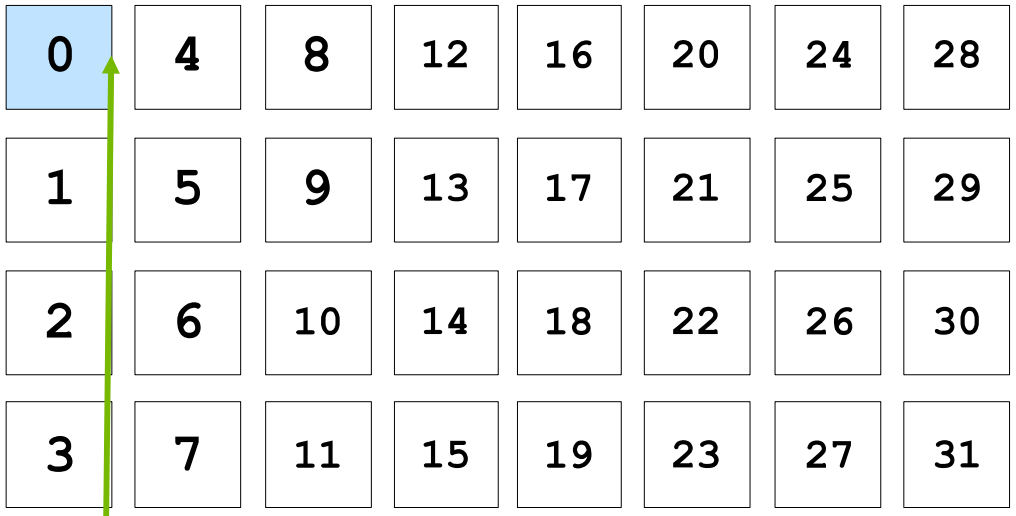
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

One way to address
programmatically is
grid-stride loop

GPU

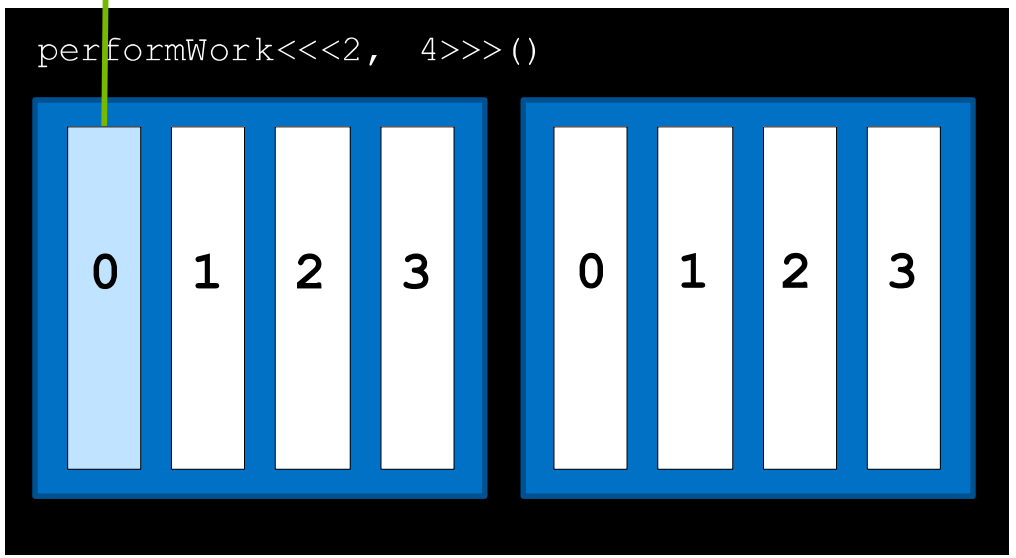


GPU
DATA



In a grid-stride loop
thread's first eleme
calculated as usual,
`threadIdx.x`
`blockIdx.x *`
`blockDim.x`

GPU

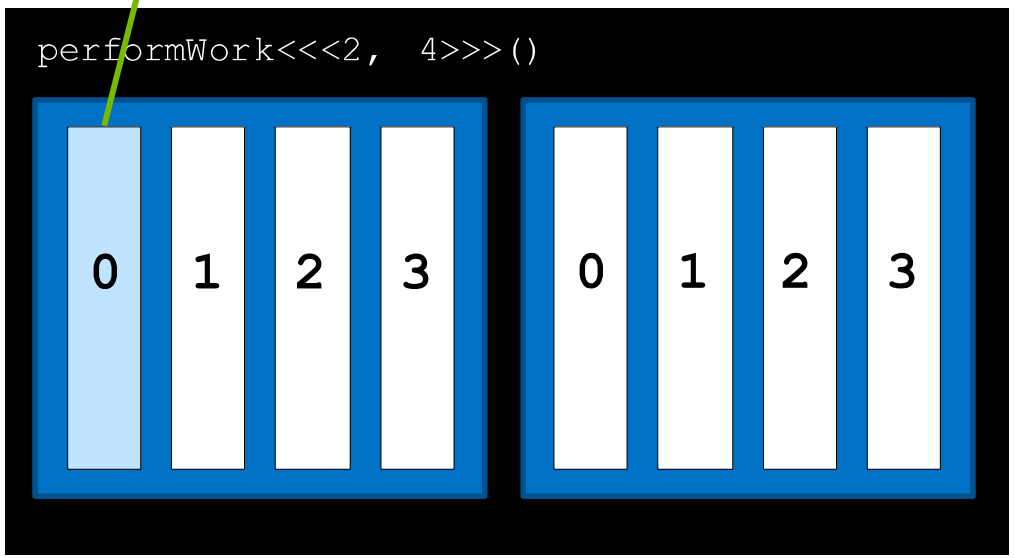


GPU
DATA

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 str
forward by the numb
threads in the gri
(**blockDim.x**
gridDim.x), in this
8

GPU

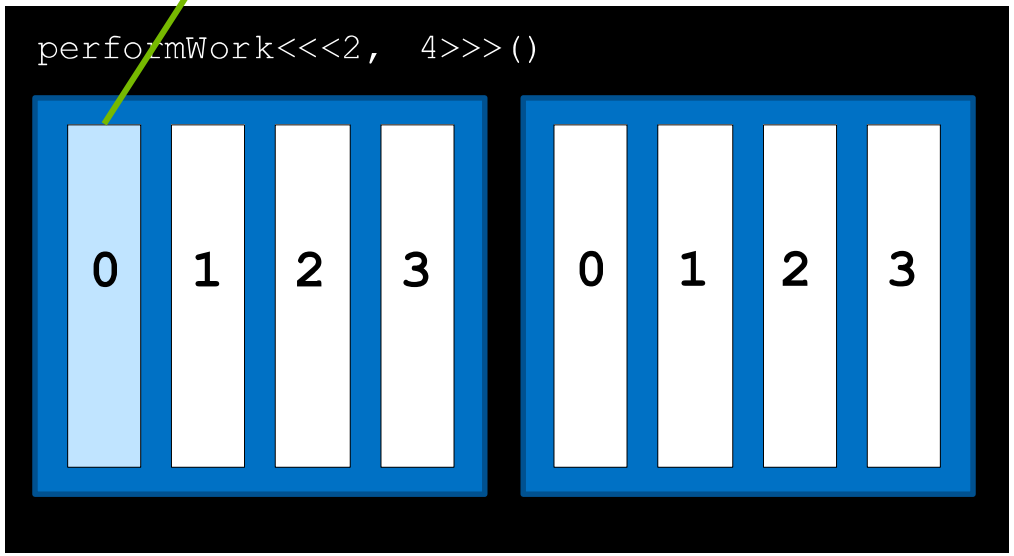


GPU
DATA

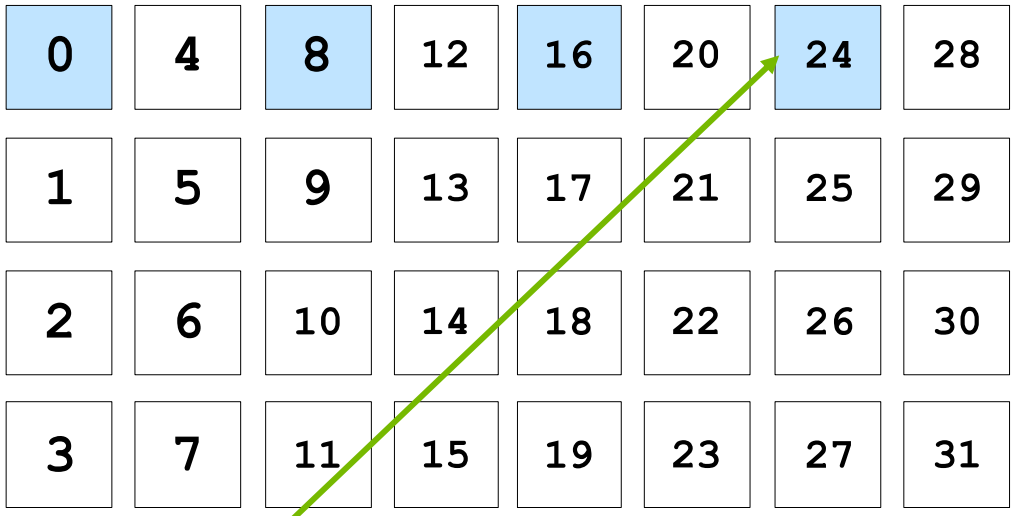
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

It continues in this wa
its data index is greate
the number of da
elements

GPU

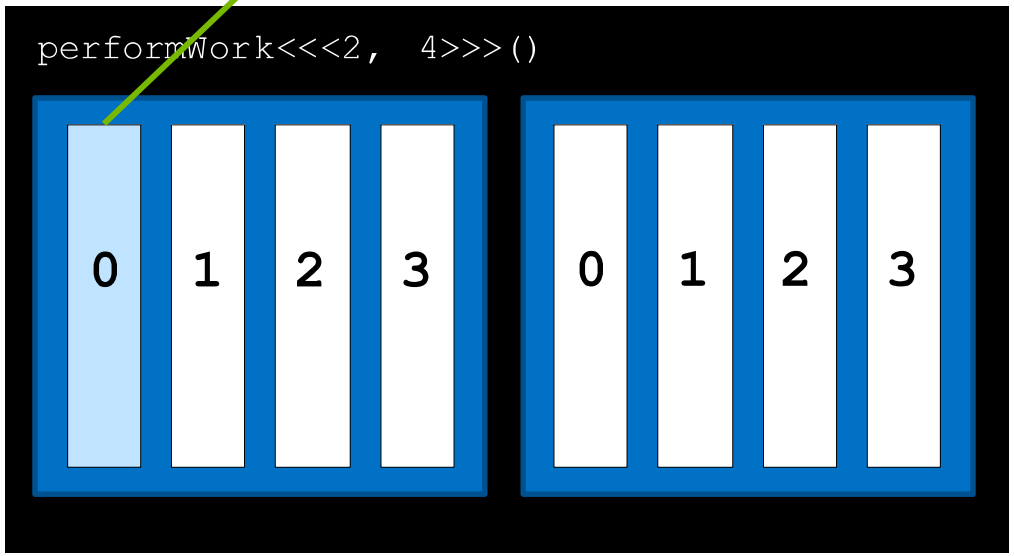


GPU
DATA



It continues in this wa
its data index is greate
the number of da
elements

GPU

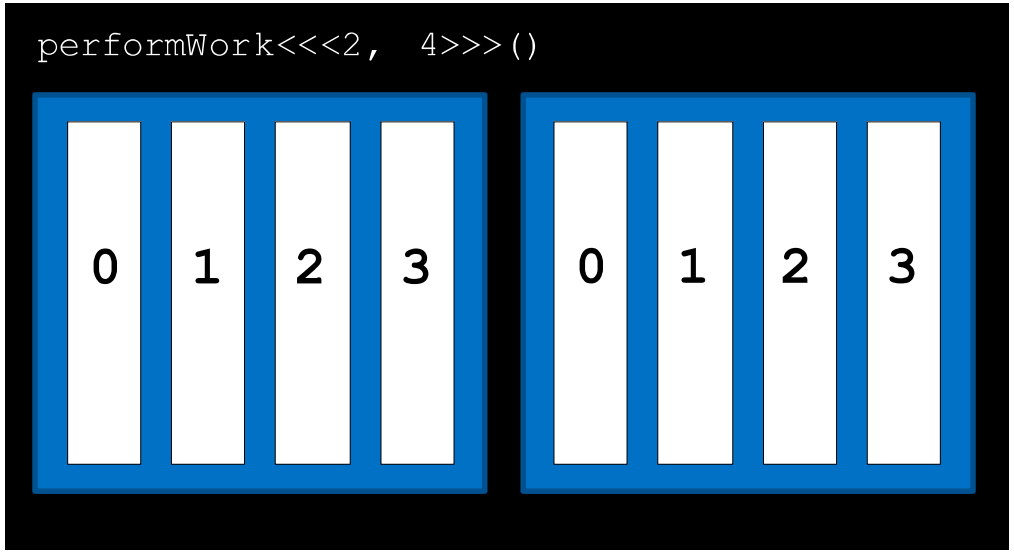


GPU
DATA

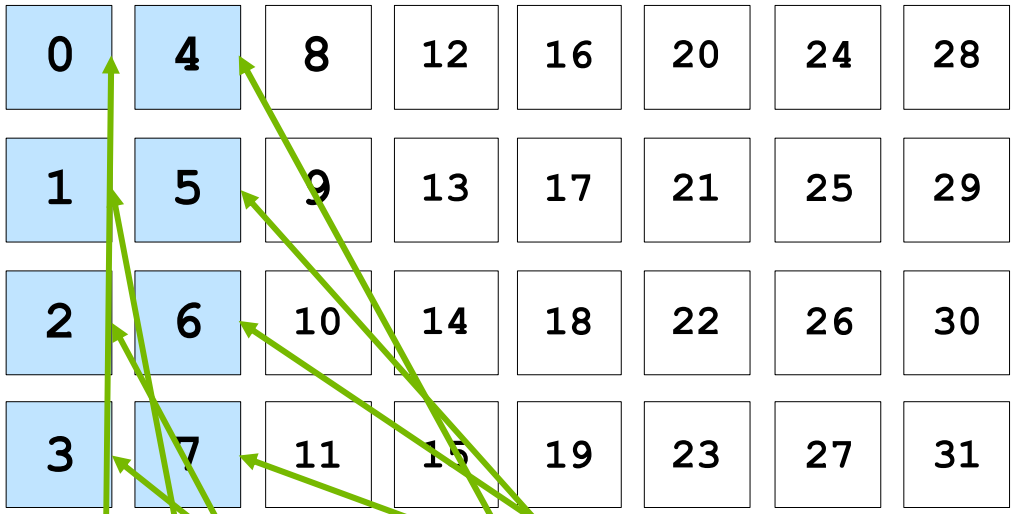
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads work this way, all element covered

GPU

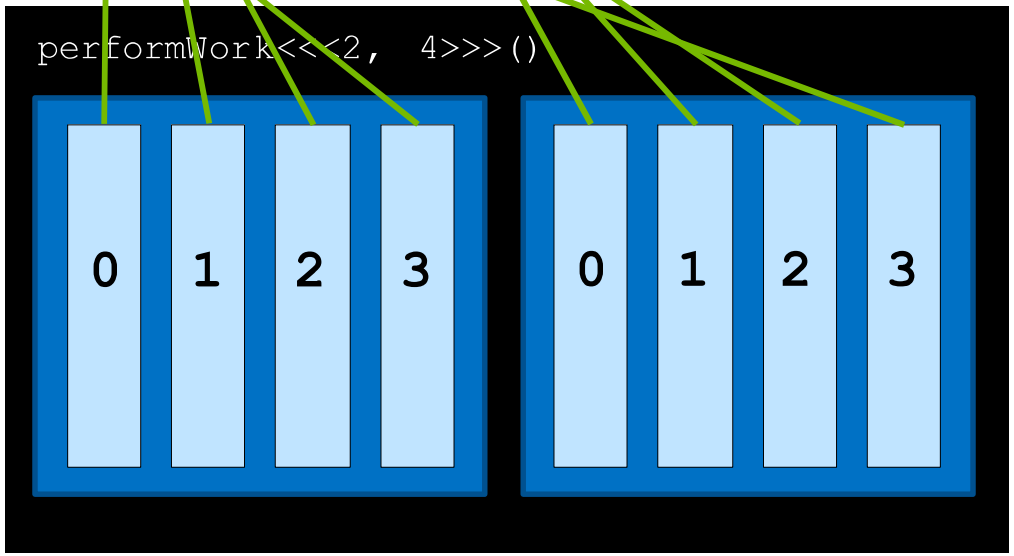


GPU
DATA



With all threads work this way, all element covered

GPU

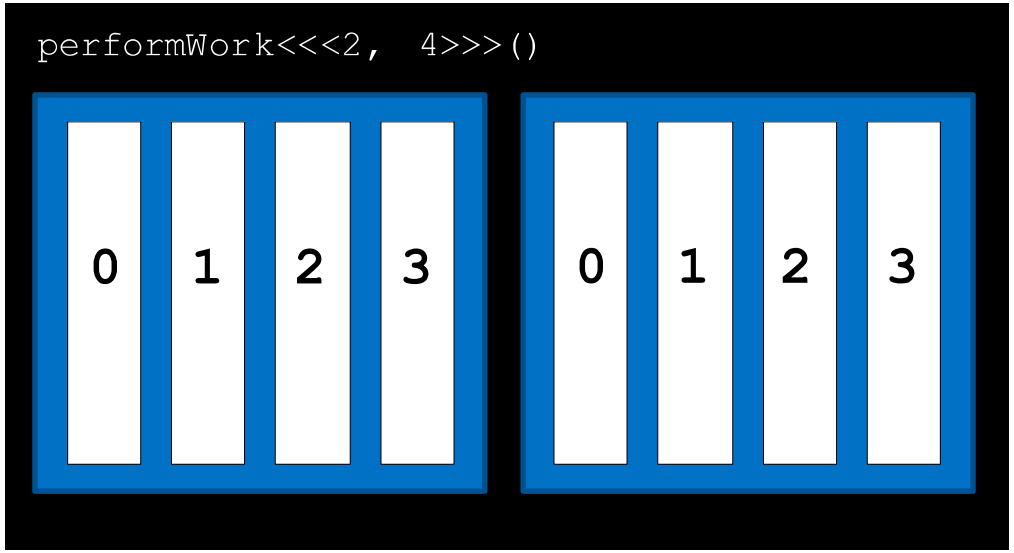


GPU
DATA

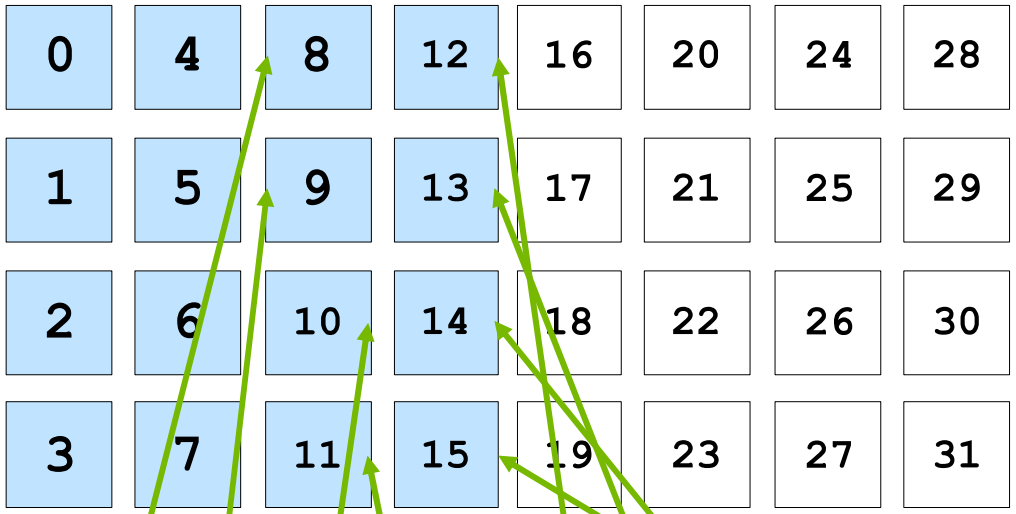
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads work this way, all element covered

GPU

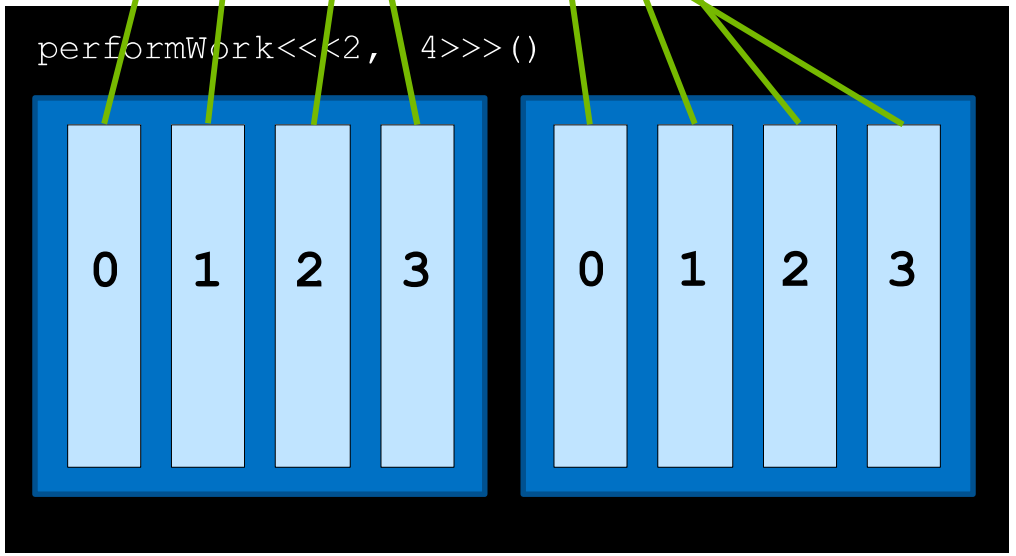


GPU
DATA



With all threads work
this way, all element
covered

GPU

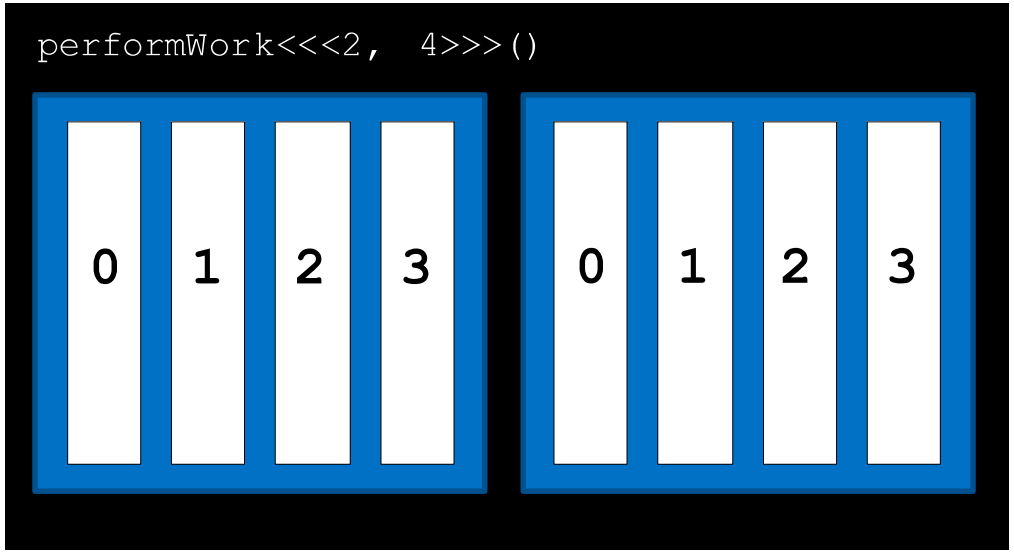


GPU
DATA

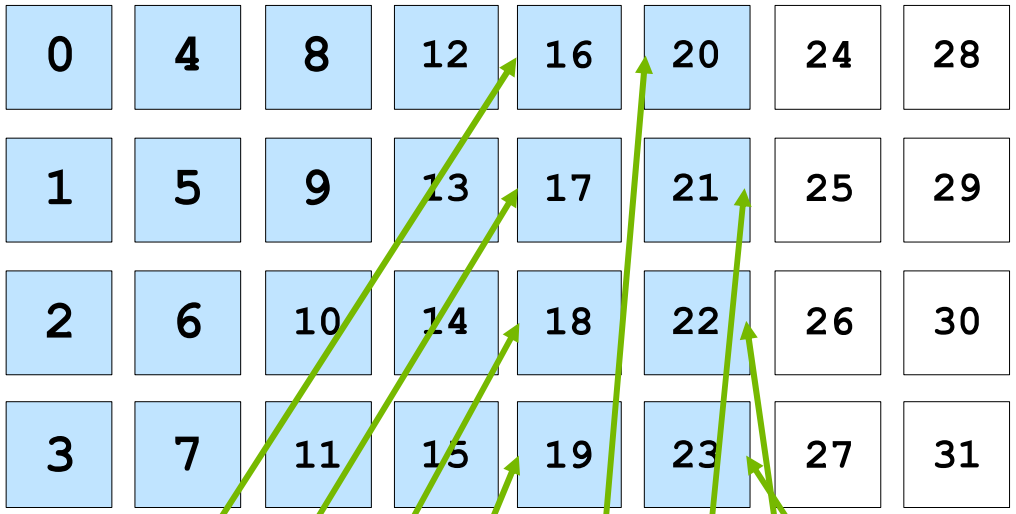
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads work this way, all element covered

GPU

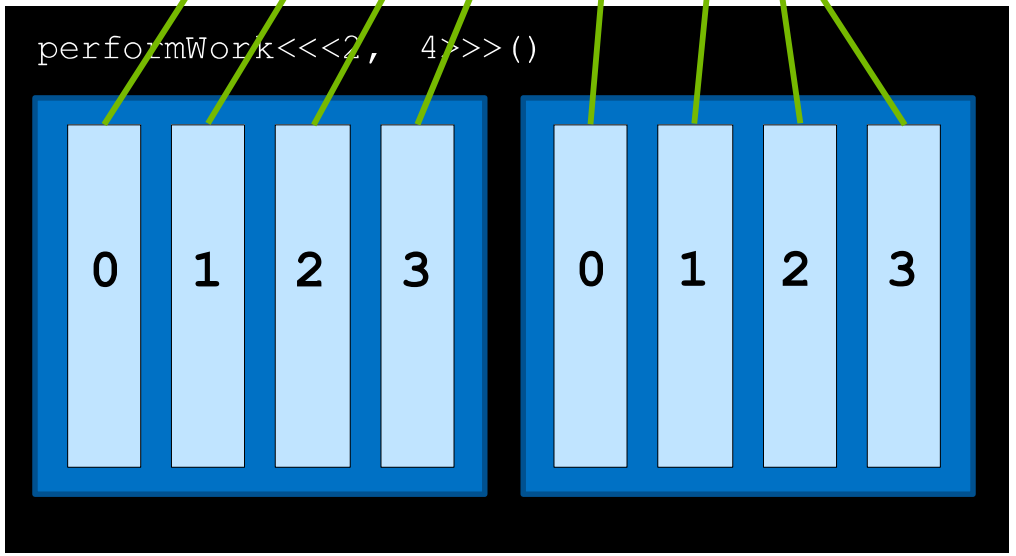


GPU
DATA



With all threads work this way, all element covered

GPU

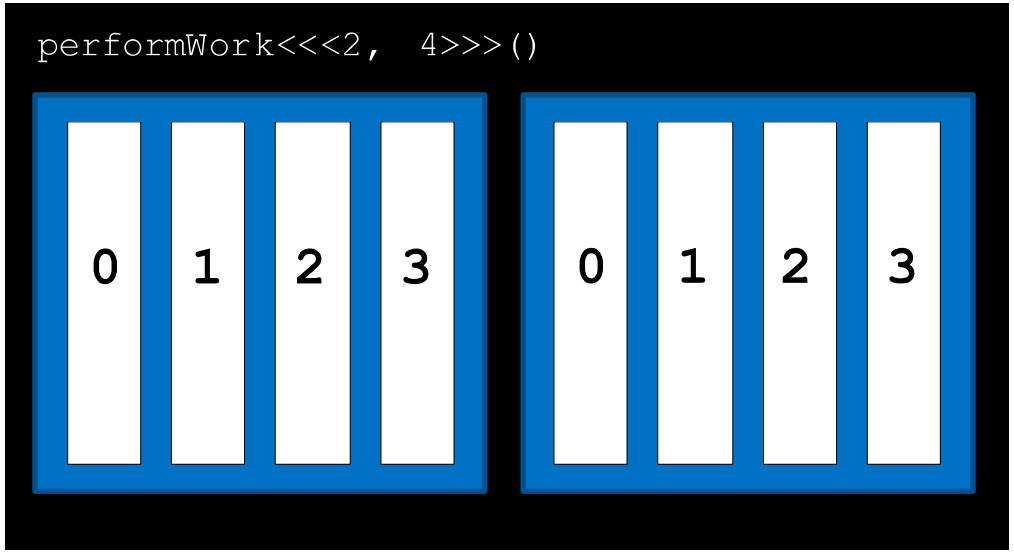


GPU
DATA

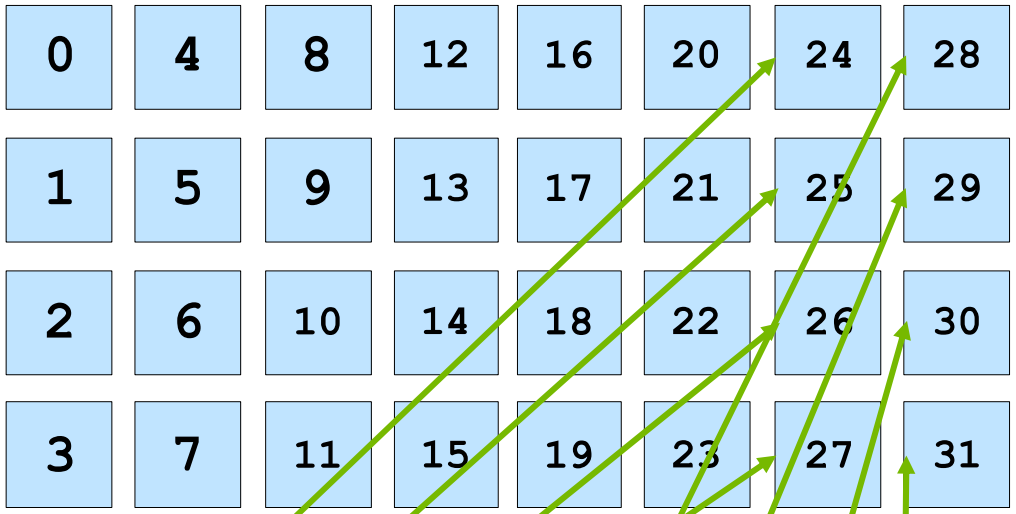
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

With all threads work this way, all element covered

GPU

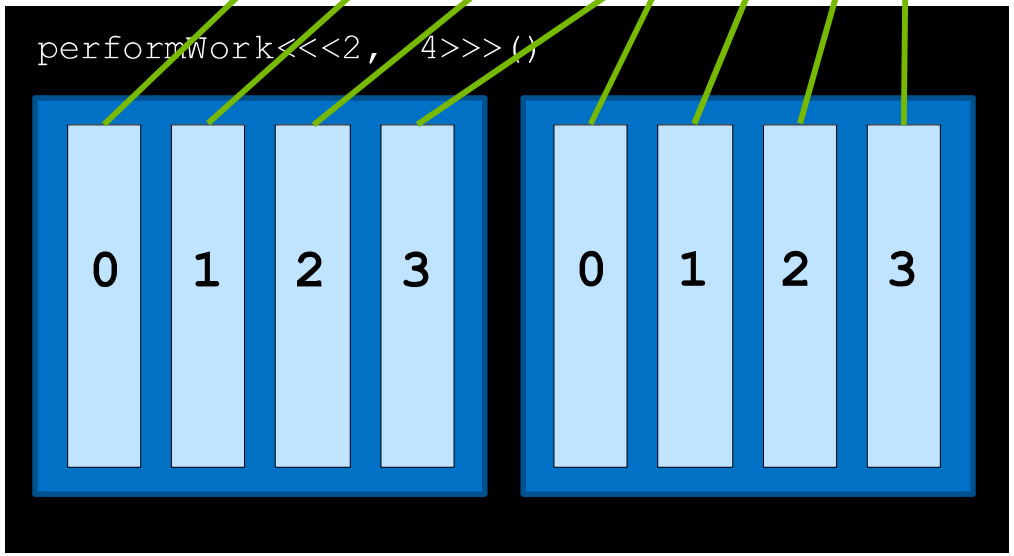


GPU
DATA



With all threads work this way, all element covered

GPU

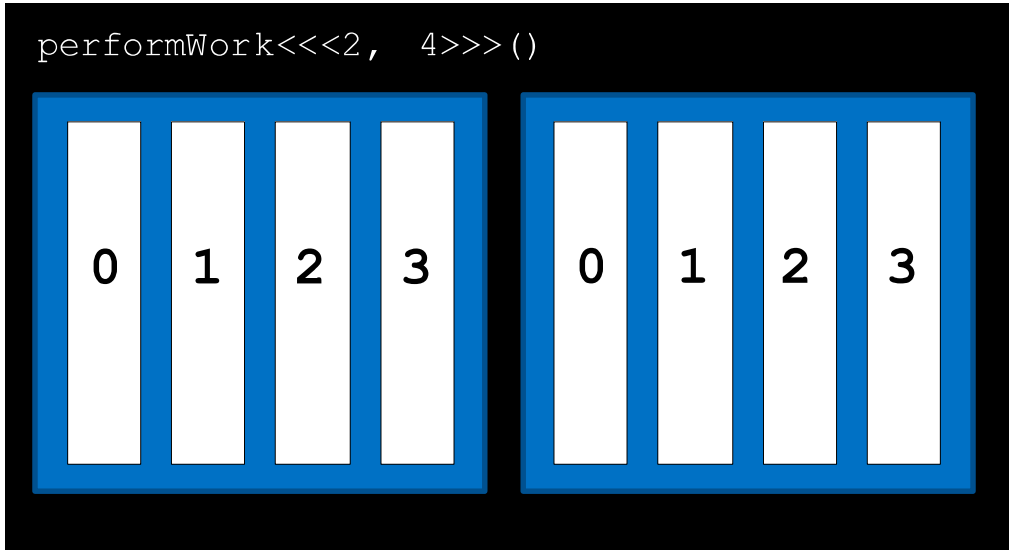


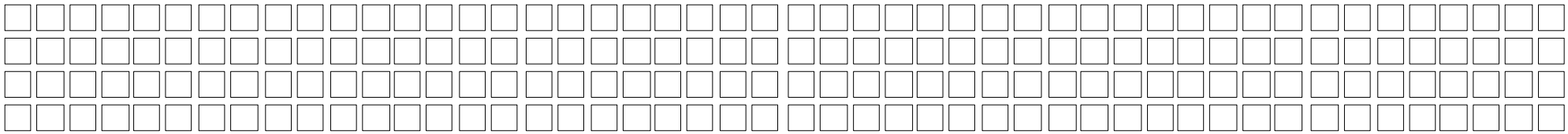
GPU
DATA

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

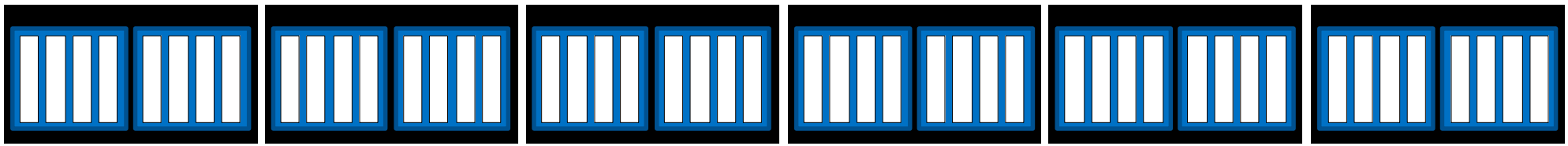
With all threads work this way, all element covered

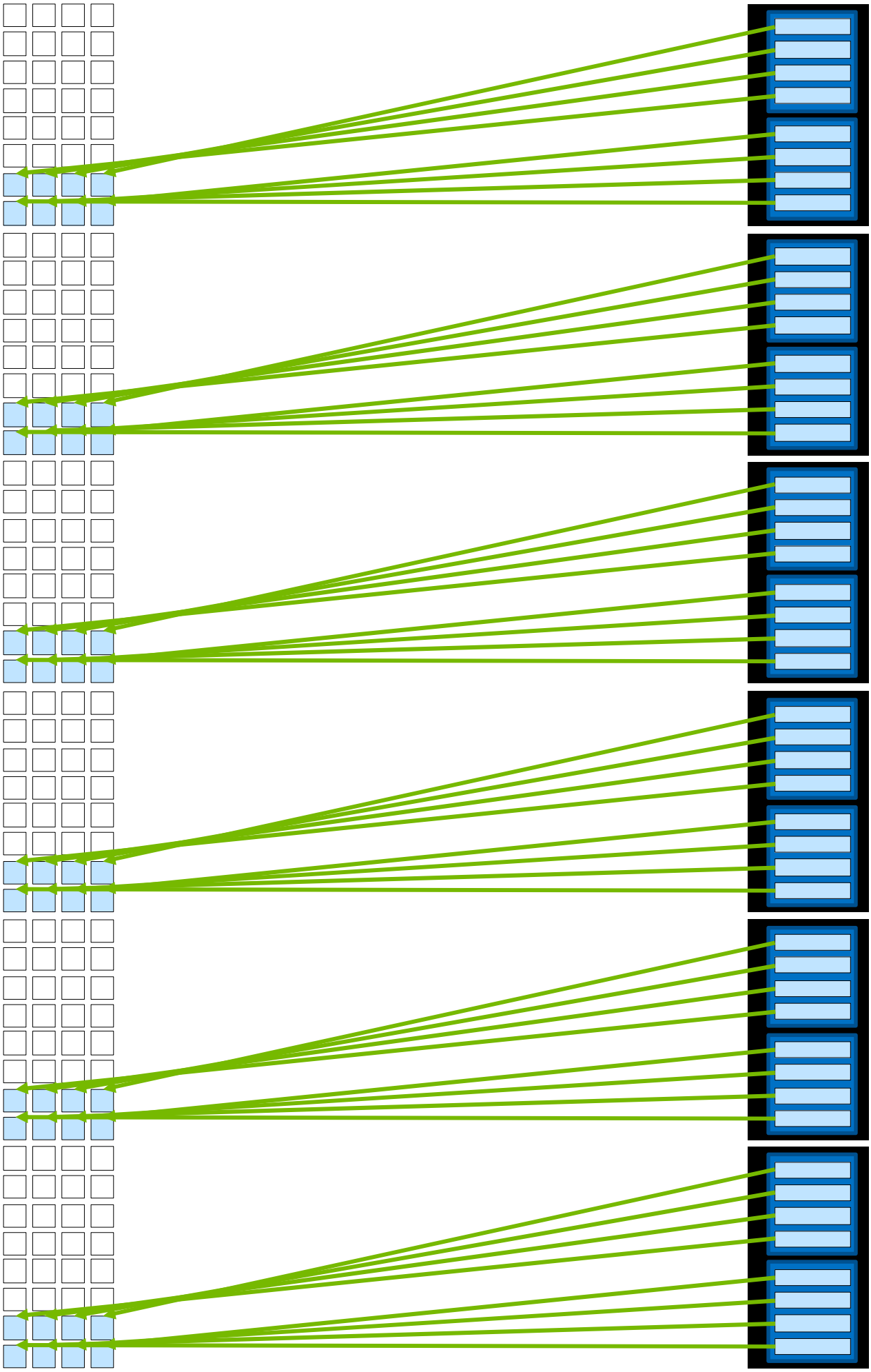
GPU

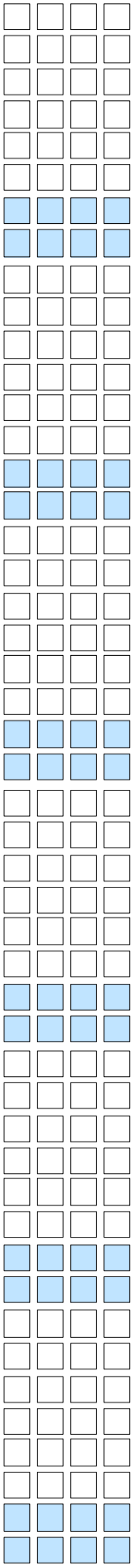


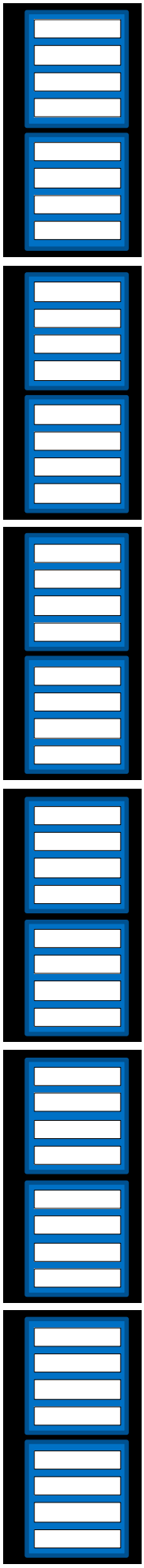


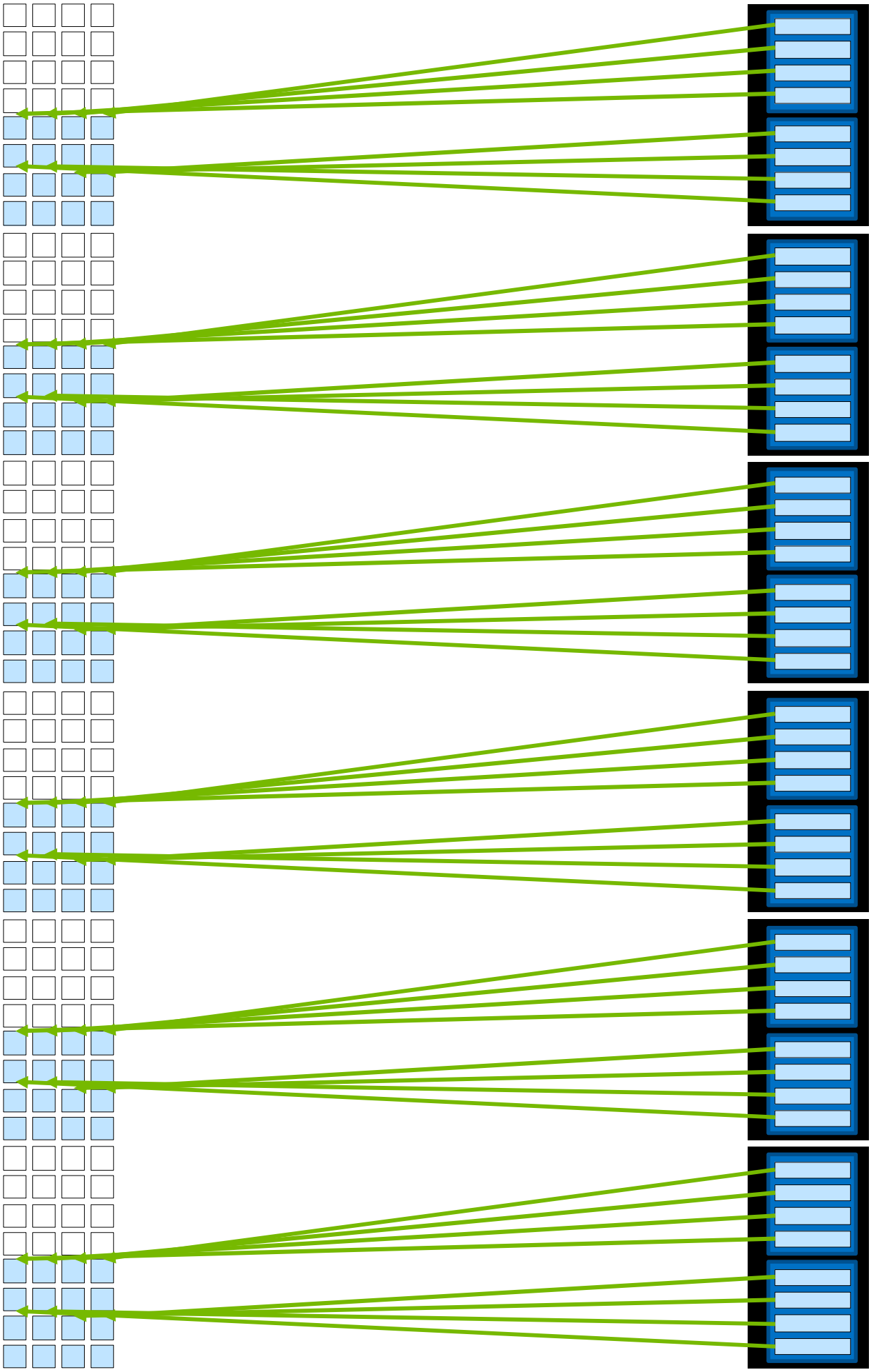
CUDA runs as many blocks in parallel at once as the GPU hardware supports, for massive parallelization

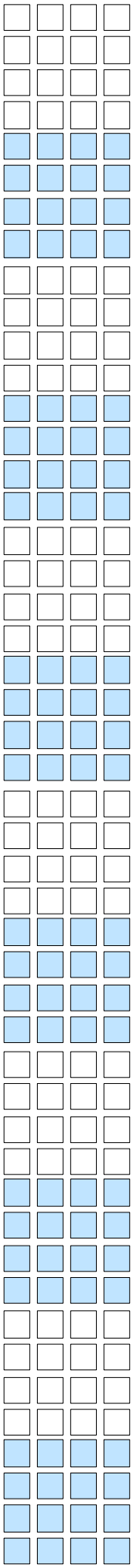


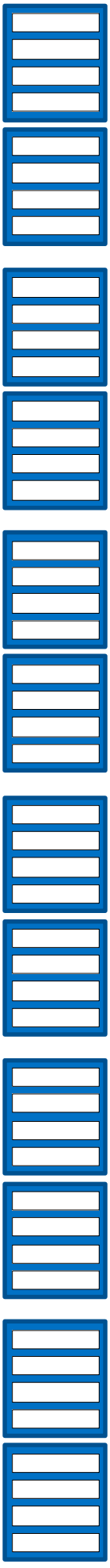


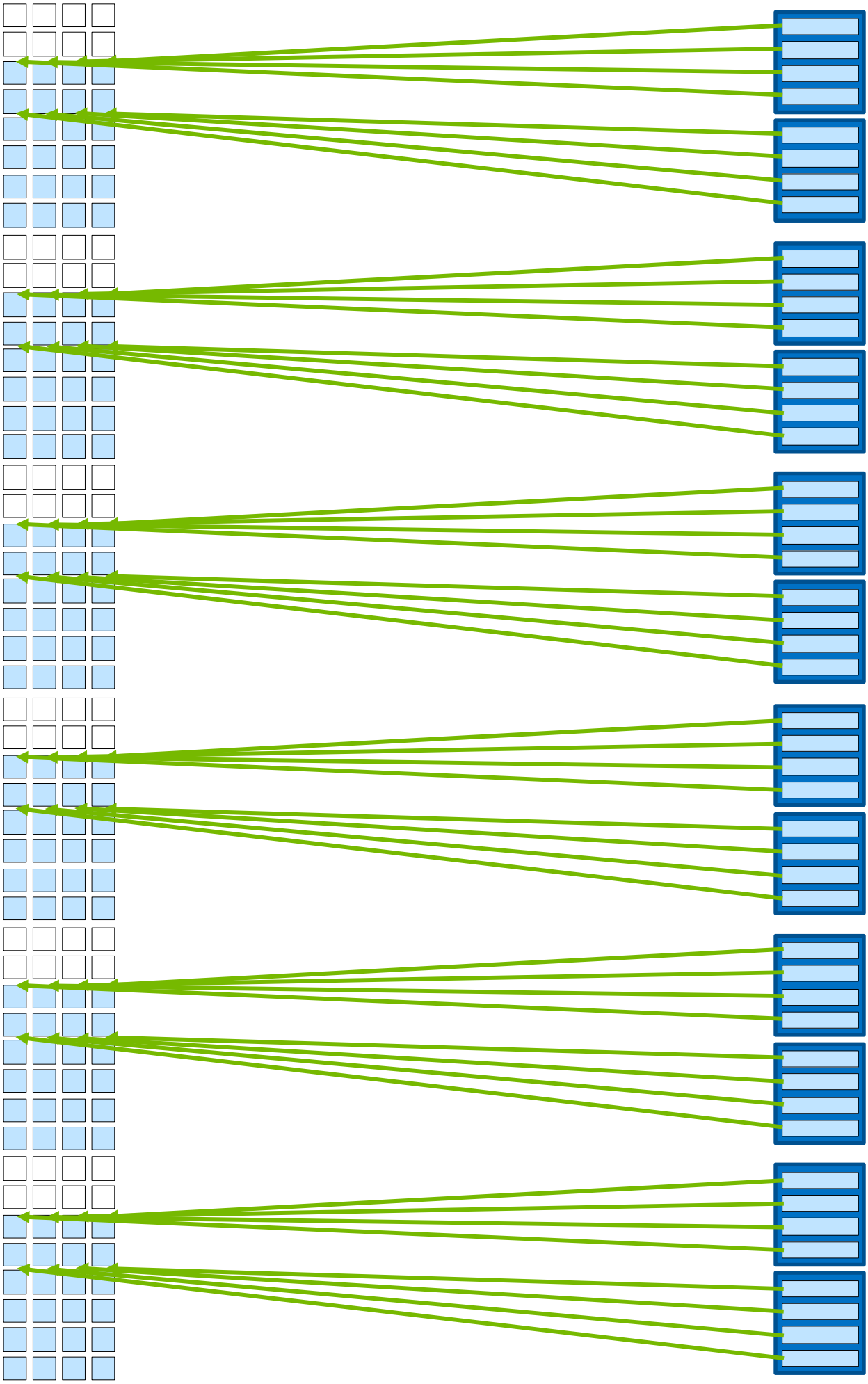


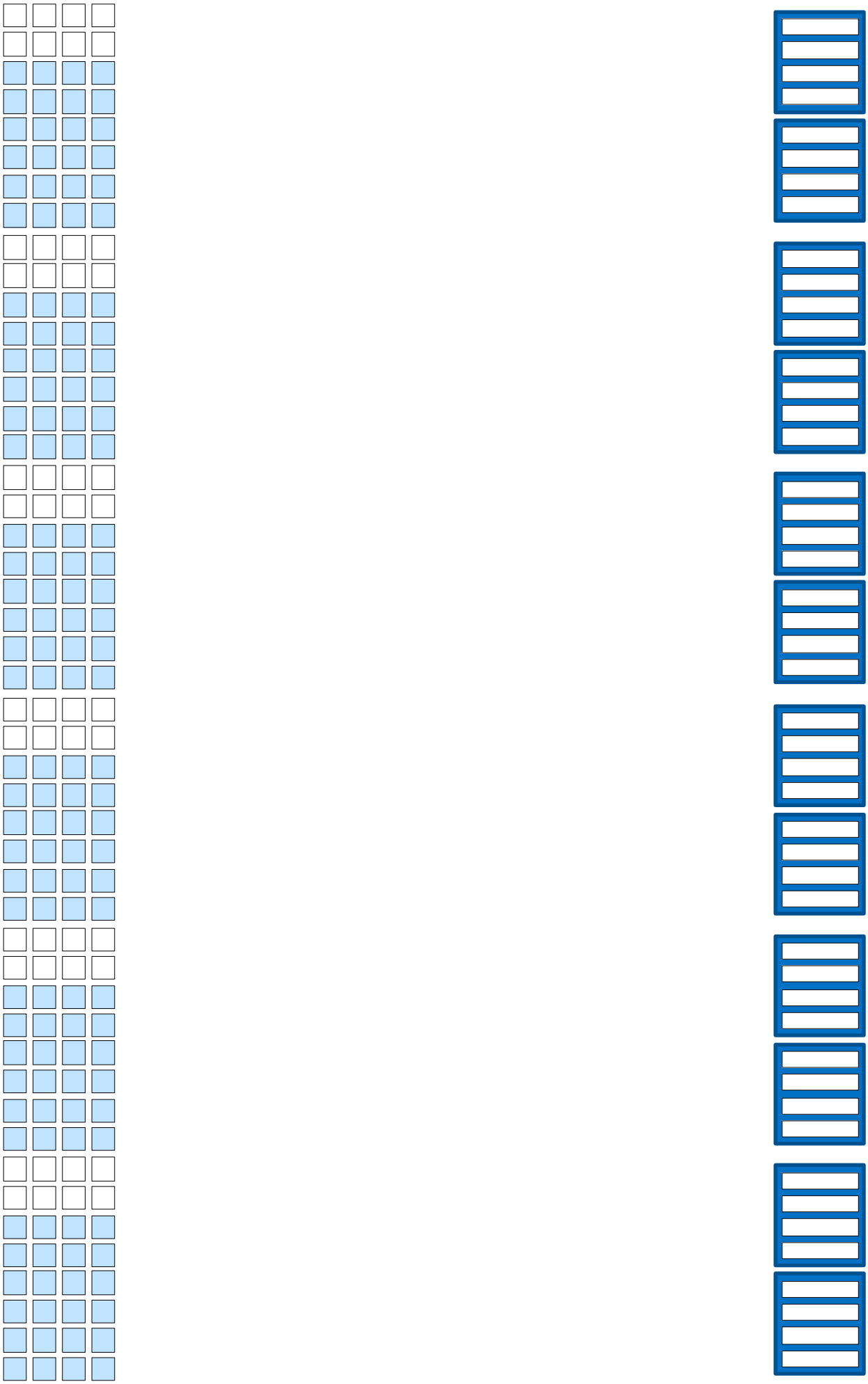


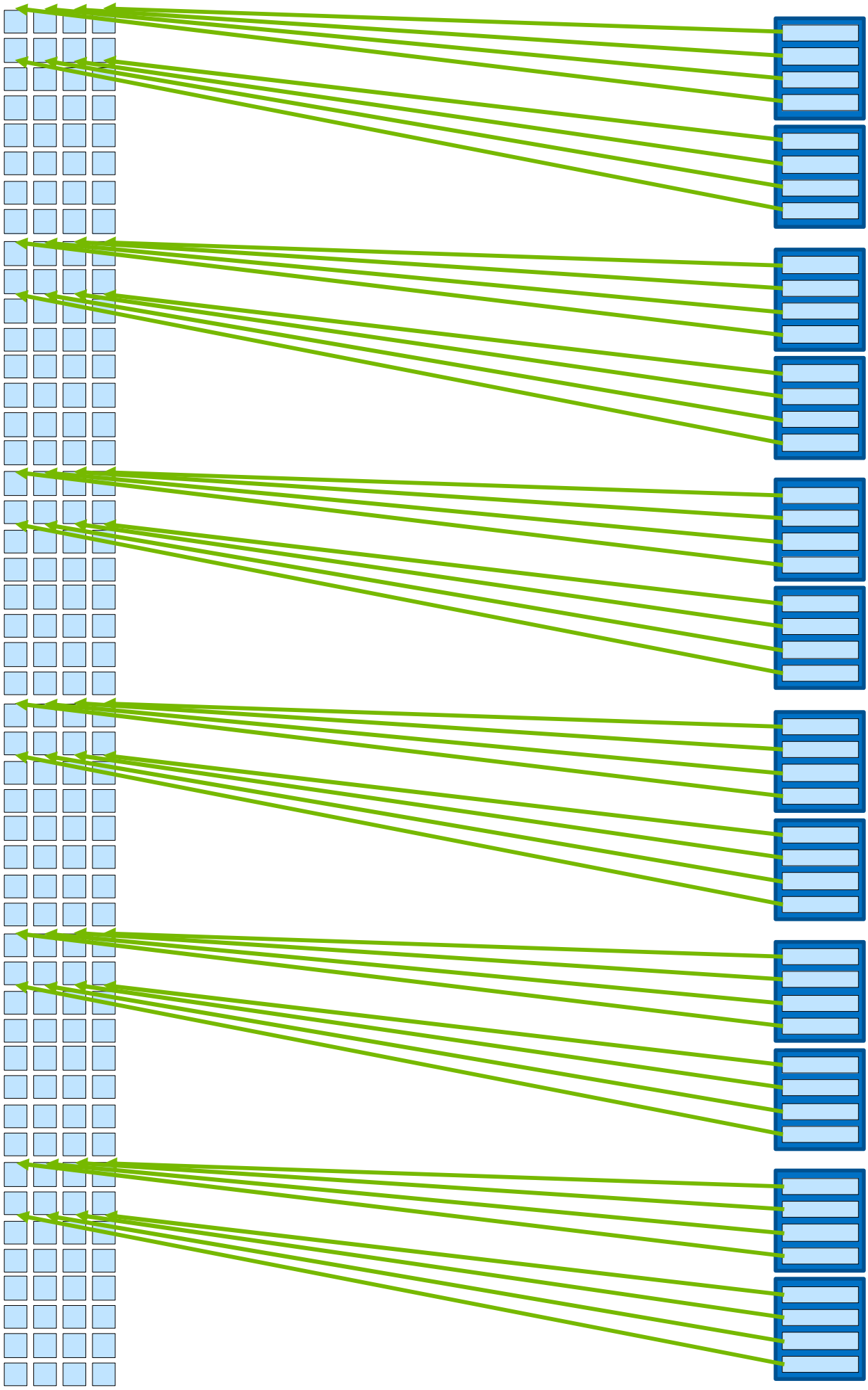


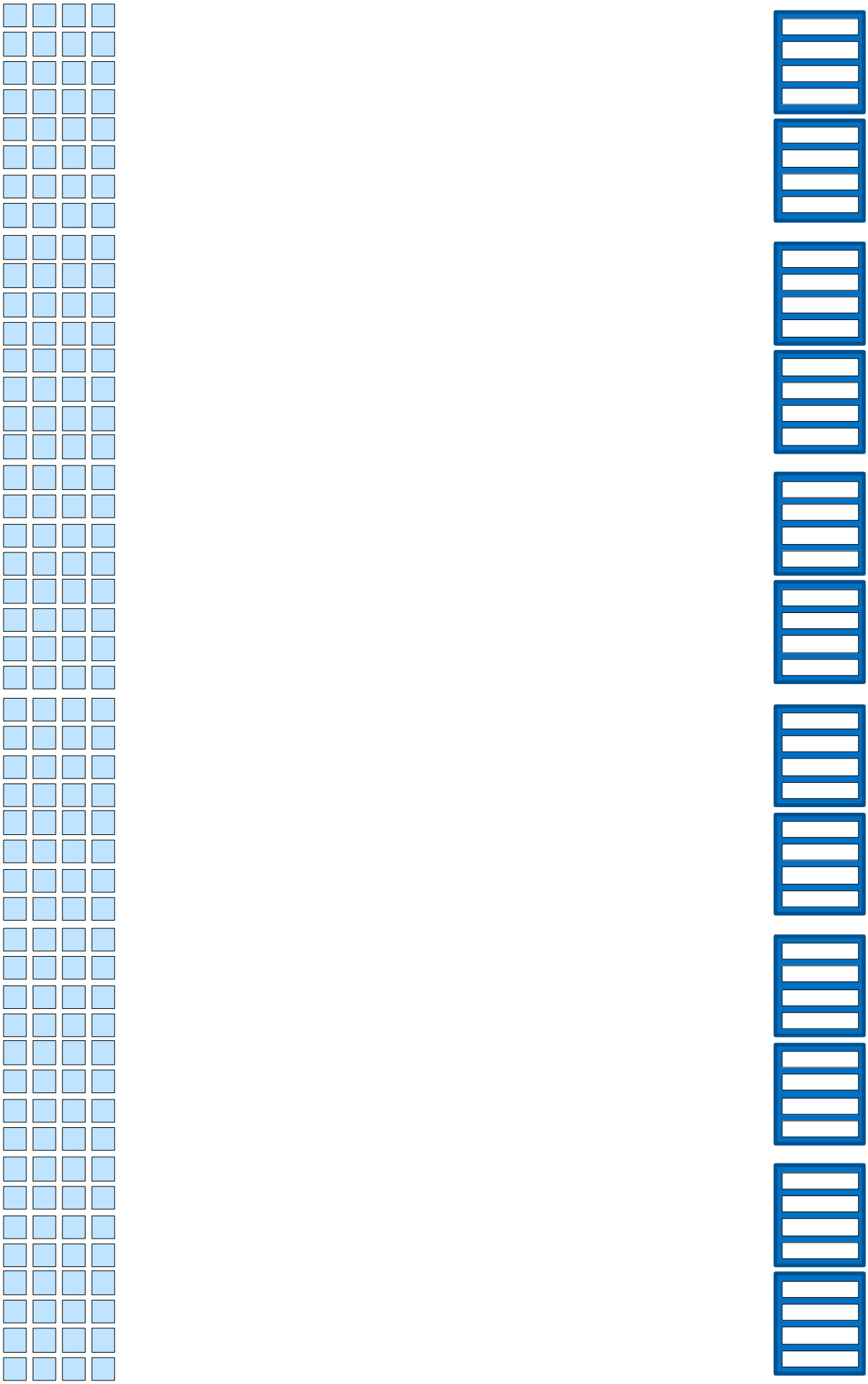












Glossary

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 define 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 the number of threads in the grid.

www.nvidia.com/dli