Accelerating Applications with CUDA C/C++

TOPICS

GPU-accelerated vs. CPU-only Applications

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

DATA

CPU
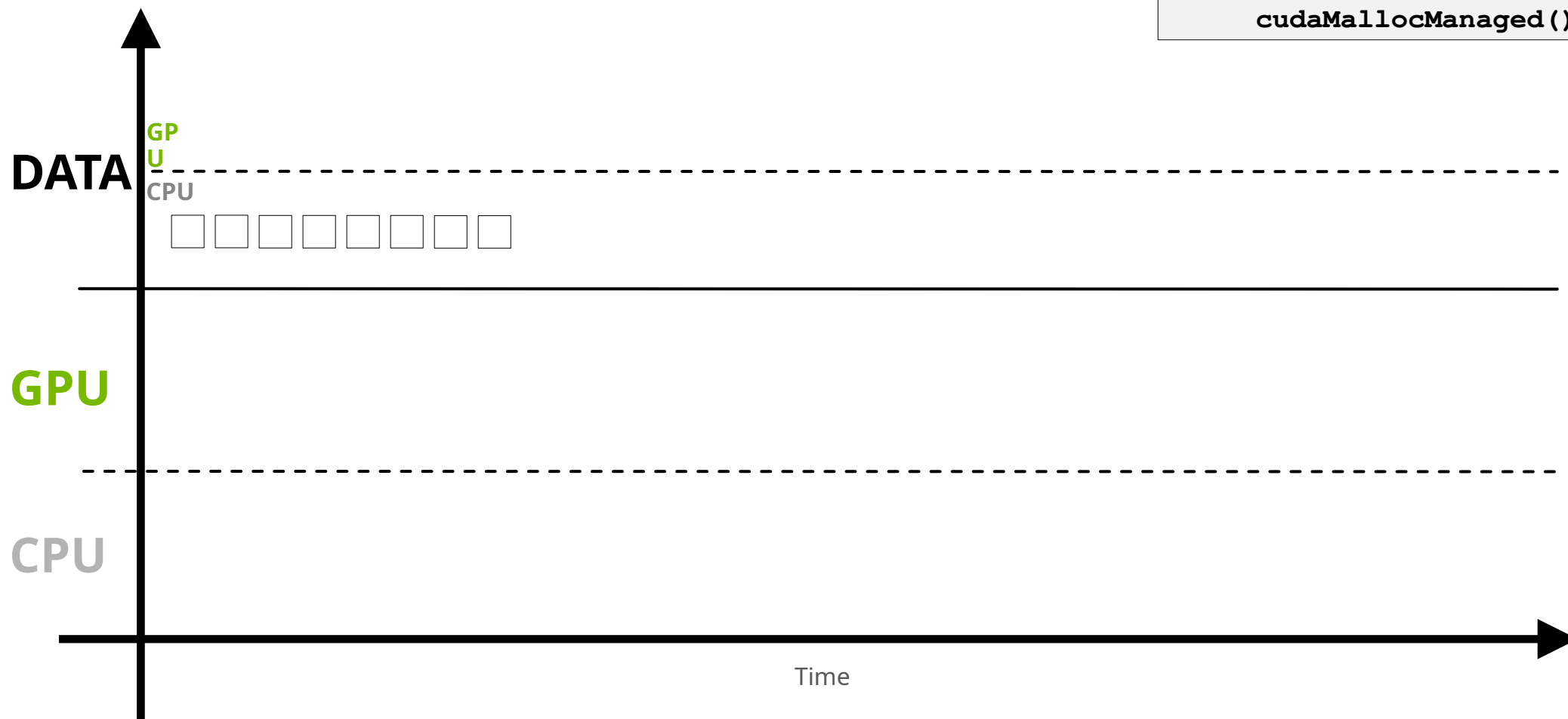
initialize()

Time

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

DATA

GPU
CPU

GPU

CPU
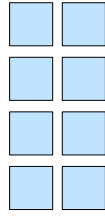
initialize()

Time

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

DATA
GPU
CPU

GPU

`performWork()`

CPU

`initialize()`

Time

CUDA Kernel Execution

GPU

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

GPU


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

GPU


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

GPU

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

GPU

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

GPU

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

GPU


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

GPU

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

GPU

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

CUDA-Provided Thread Hierarchy Variables

GPU

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

GPU

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

4

GPU

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

4   4

GPU

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


       0   1   2   **3**   0   1   2   3

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

GPU

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

0 1 2 3   0 **1** 2 3

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

GPU

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

0 1 2 3   0 1 2 3

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

GPU

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

0 1 2 3   0 1 2 3

Coordinating Parallel Threads
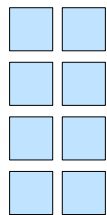
GP
U
DATA

GPU

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

0	4
1	5
2	6
3	7

GPU

DATA

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

4   4
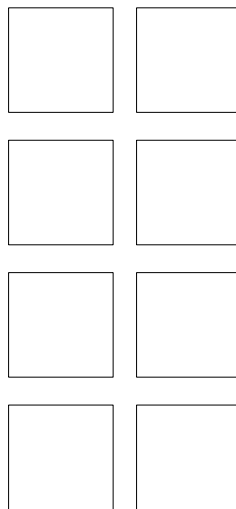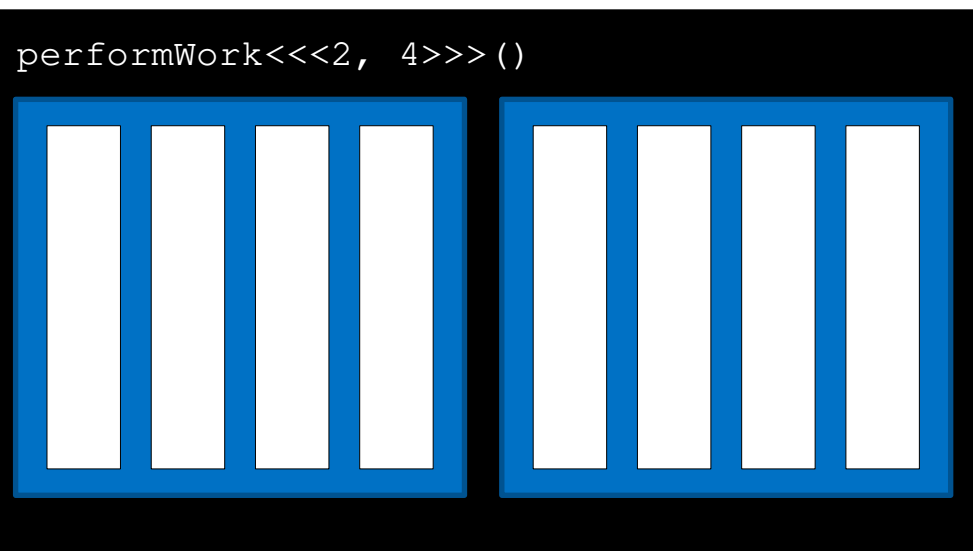
GP
U
DATA

0	4
1	5
2	6
3	7
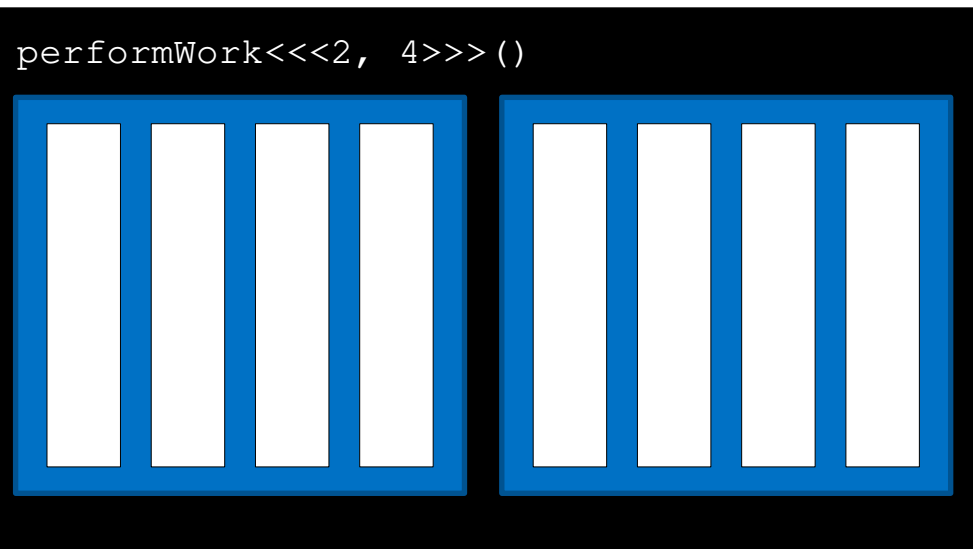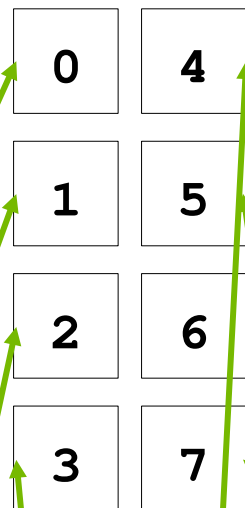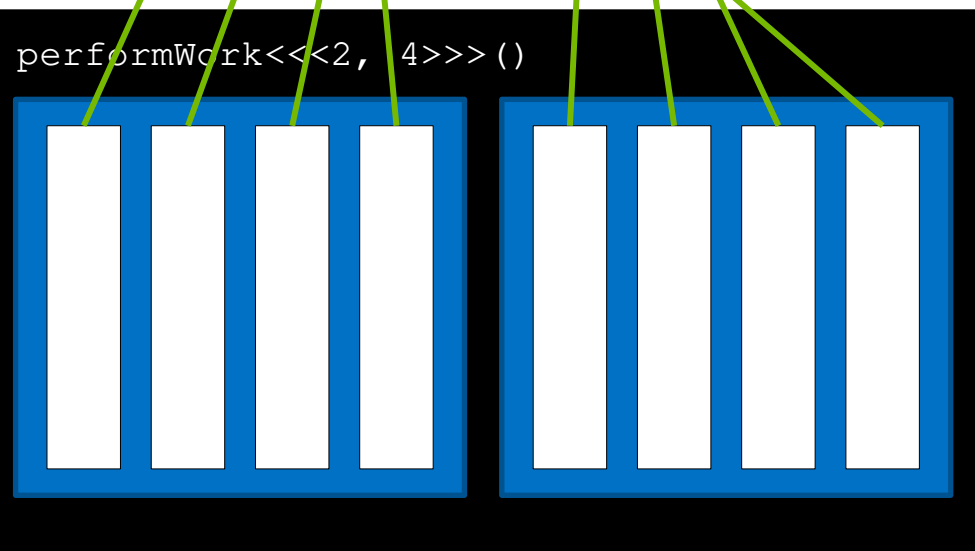
…and the index of its block within the grid via `blockIdx.x`

GPU

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

0

1

4   4

GP U DATA

0   4

1   5

2   6

3   7

GPU

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

0   0 1 2 3   4

1   0 1 2 3   4

DATA

0	4
1	5
2	6
3	7

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

dataIndex
?

GPU

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

0 | 0 | 1 | 2 | 3 | 4

1 | 0 | 1 | 2 | 3 | 4

GP
U

DATA

0	4
1	5
2	6
3	7

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

dataIndex
0

GPU

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

0
| 0 | 1 | 2 | 3 |

1
| 0 | 1 | 2 | 3 |

4   4

DATA

0	4
1	5
2	6
3	7

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

GPU

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

0

0	1	2	3

4

1

0	1	2	3

4

NVIDIA | DEEP LEARNING INSTITUTE

GPU DATA

0	4
1	5
2	6
3	7

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

dataIndex
1

GPU

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

0
0	1	2	3

4

1
0	1	2	3

4

GPU DATA

0	4
1	5
2	6
3	7

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

dataIndex
?

GPU

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

0
0	1	2	3

4

1
0	1	2	3

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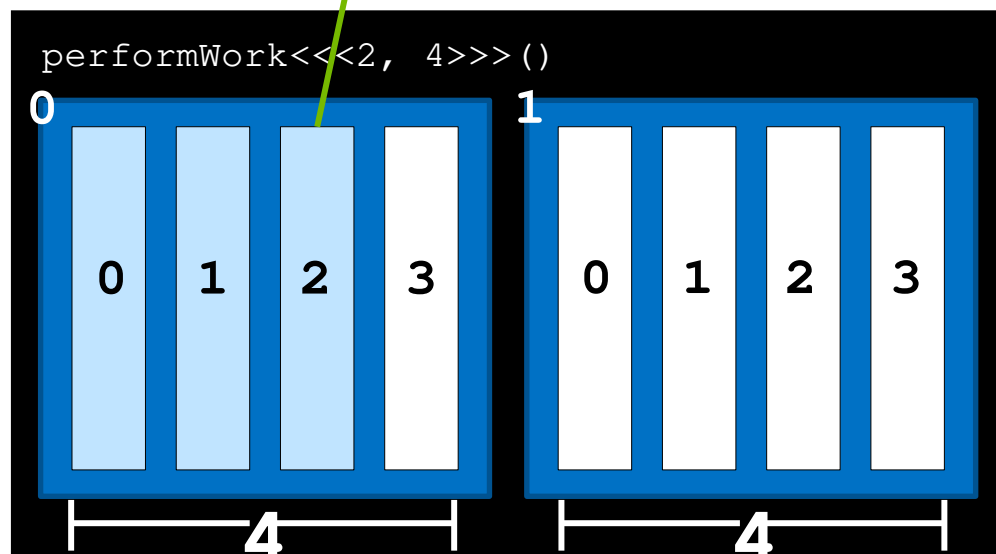
threadIdx.x	+	blockIdx.x	*	blockDim.x
3		0		4

dataIndex
?

GPU DATA

GPU

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

0   0 1 2 3   1   0 1 2 3

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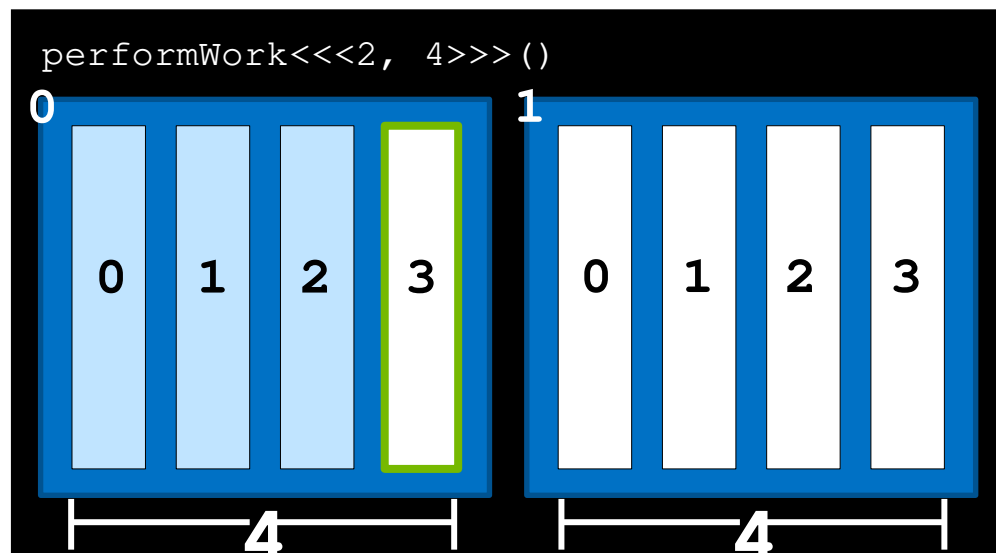
threadIdx.x	+	blockIdx.x	*	blockDim.x
3		0		4
dataIndex				
3				

GPU DATA

GPU

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

DATA

0	4
1	5
2	6
3	7

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

dataIndex
?

GPU

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

0

0	1	2	3

4

1

0	1	2	3

4

GPU
U
DATA

0	4
1	5
2	6
3	7

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

dataIndex
?

GPU

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

0
| 0 | 1 | 2 | 3 |

1
| 0 | 1 | 2 | 3 |

4   4

GPU DATA

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

dataIndex				
5				

GPU

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

0   0 1 2 3   4

1   0 1 2 3   4

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

GPU

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

0
0 1 2 3
4

1
0 1 2 3
4

GPU DATA

0	4
1	5
2	6
3	7

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

dataIndex
?

GPU

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

Block 0: [0 | 1 | 2 | 3] — 4

Block 1: [0 | 1 | 2 | 3] — 4

GPU DATA

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

dataIndex
7

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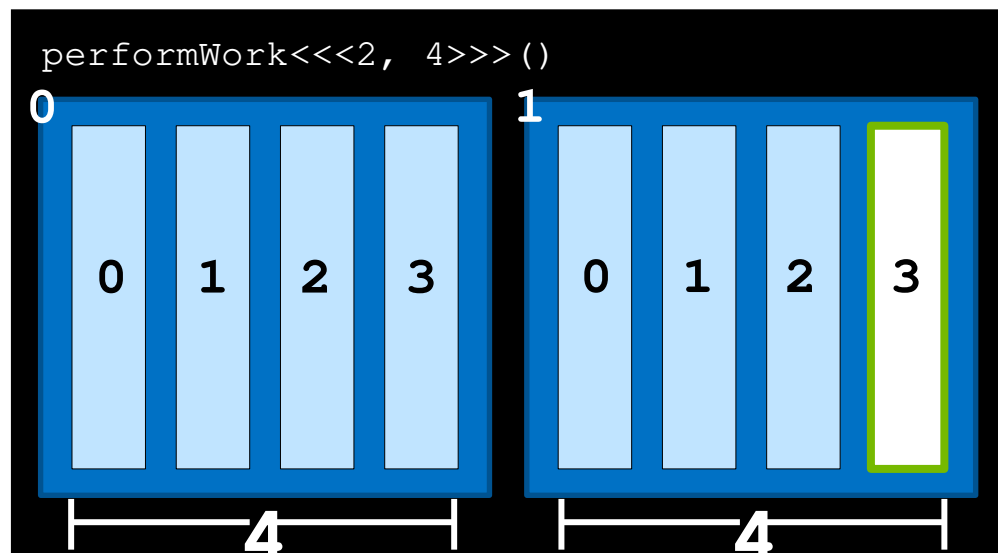
GPU

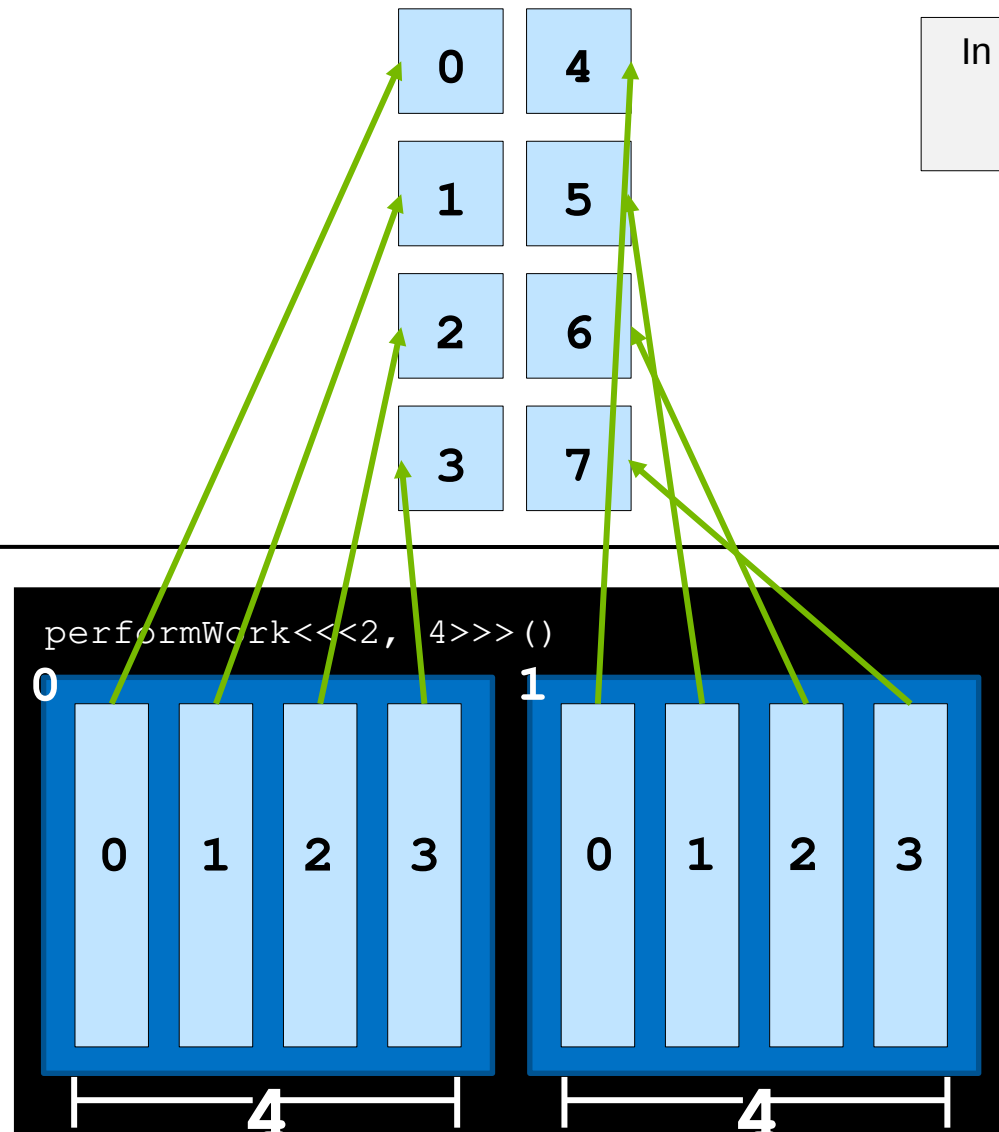# Grid Size Work Amount Mismatch

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

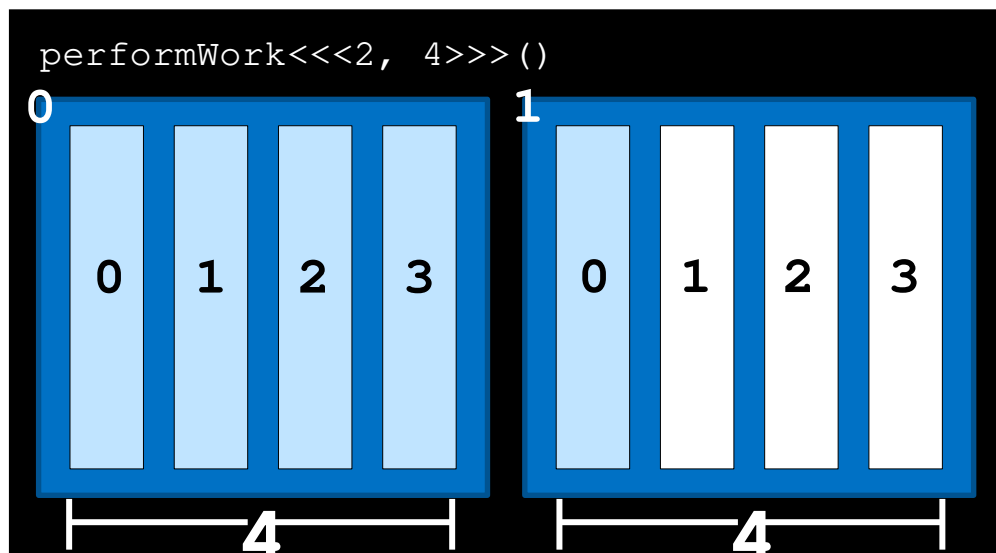`performWork<<<2, 4>>>()`
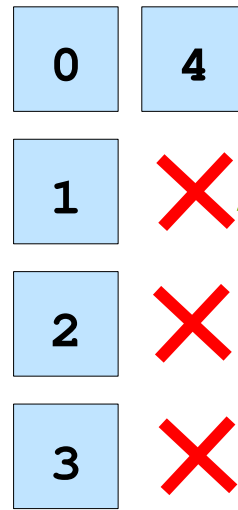
GP
U

DATA

0	4
1	
2	
3	

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

dataIndex	<	N	=	Can work
4		5		?

GPU

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

0
0	1	2	3

4

1
0	1	2	3

4

GPU DATA

| 0 | 4 |

1

2

3

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

GPU

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

0
| 0 | 1 | 2 | 3 |

1
| 0 | 1 | 2 | 3 |

4

4

GPU DATA

0	4

1

2

3
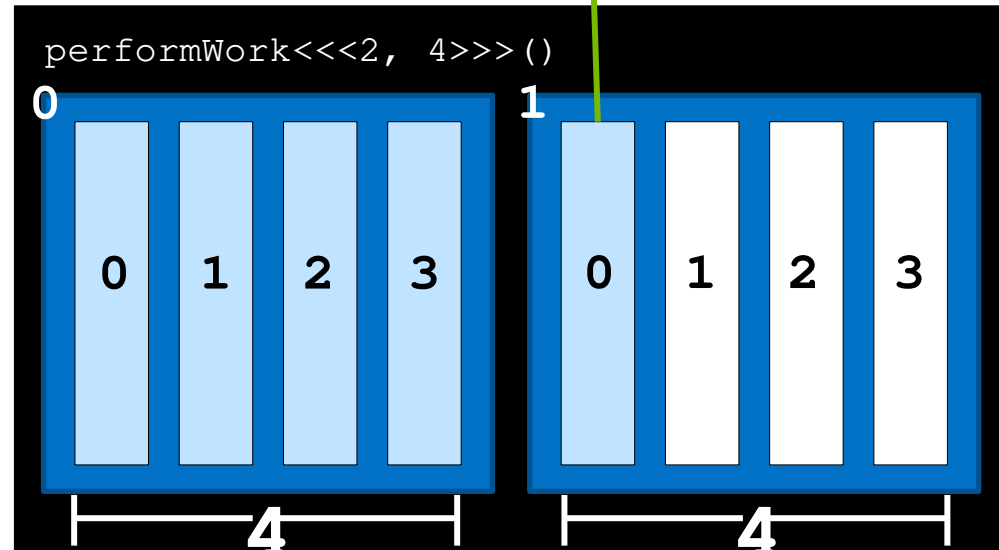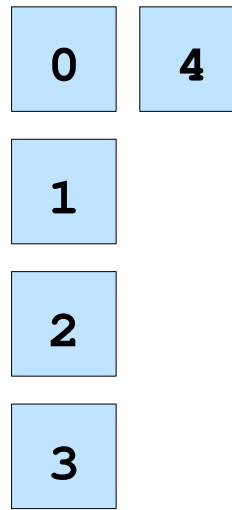
threadIdx.x	+	blockIdx.x	*	blockDim.x
1		1		4
dataIndex	<	N	=	Can work
5		5		**false**

GPU

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

0

0	1	2	3

4

1

0	1	2	3

4

GPU DATA

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

GPU

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

0: 0 1 2 3 — 4

1: 0 1 2 3 — 4

GPU DATA

0	4

1

2

3

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

GPU

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

0					1			
0	1	2	3		0	1	2	3

4   4

GPU DATA

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

0 4

1

2

3

GPU

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

0 0 1 2 3   **1** 0 1 2 3

4   4

Grid-Stride Loops

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

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

| 0 | 1 | 2 | 3 | | 0 | 1 | 2 | 3 |

In such scenarios threads cannot work on only one element
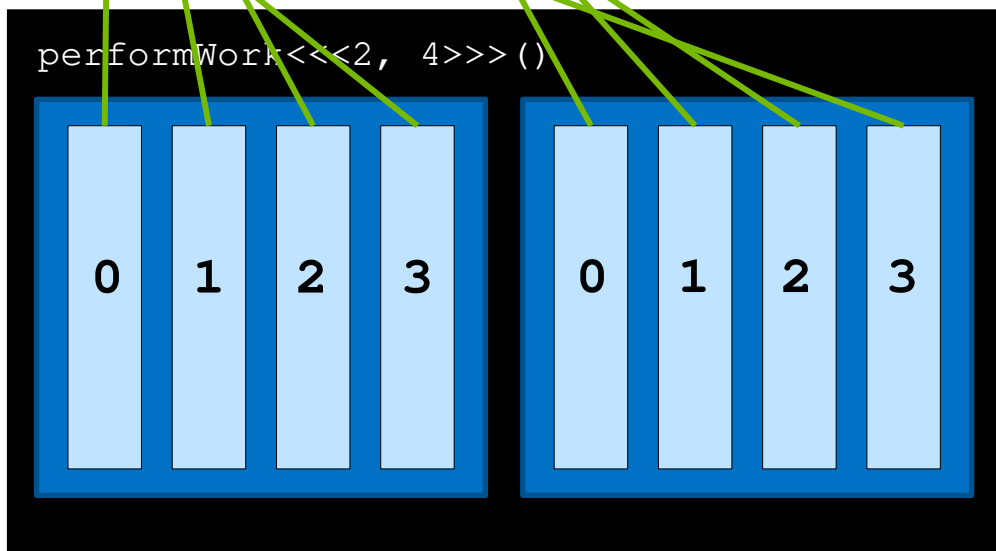
In a grid-stride loop, the thread's first element is calculated as usual, with `threadIdx.x + blockIdx.x * blockDim.x`
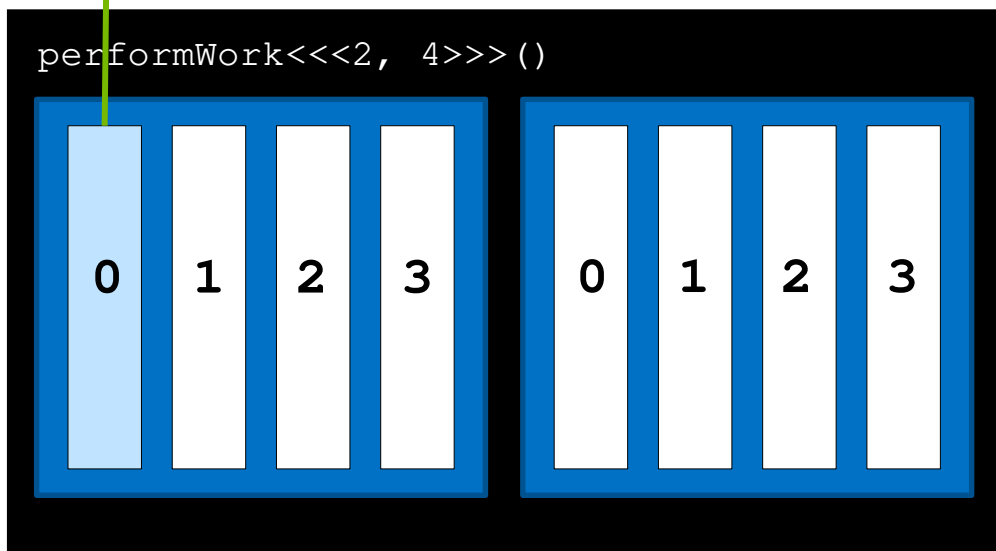
performWork<<<2, 4>>>()

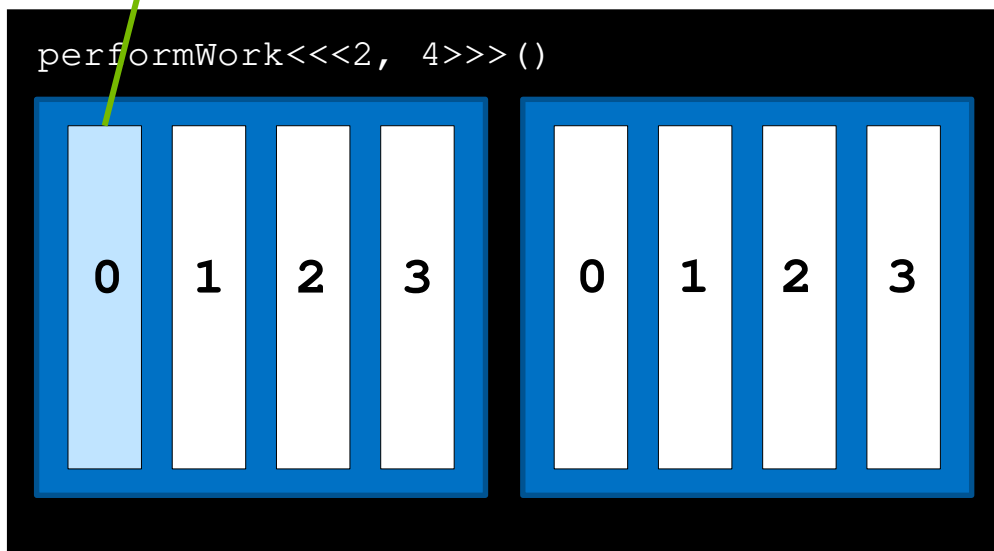GPU DATA

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

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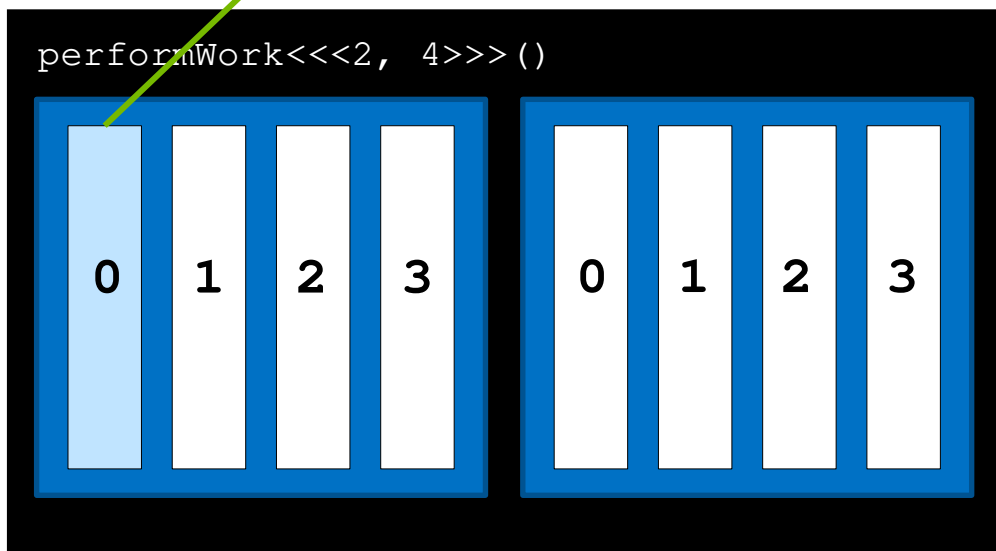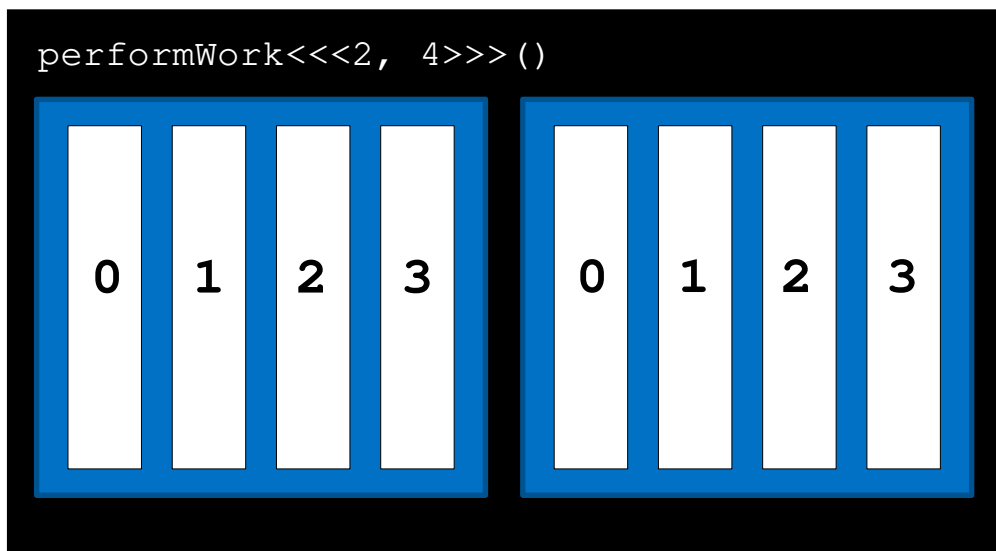
GPU

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

| 0 | 1 | 2 | 3 | | 0 | 1 | 2 | 3 |

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

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

GPU DATA

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

GPU

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

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
```

GPU

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

```
0  1  2  3      0  1  2  3
```

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

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

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

GPU
DATA

GPU

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

0 1 2 3   0 1 2 3

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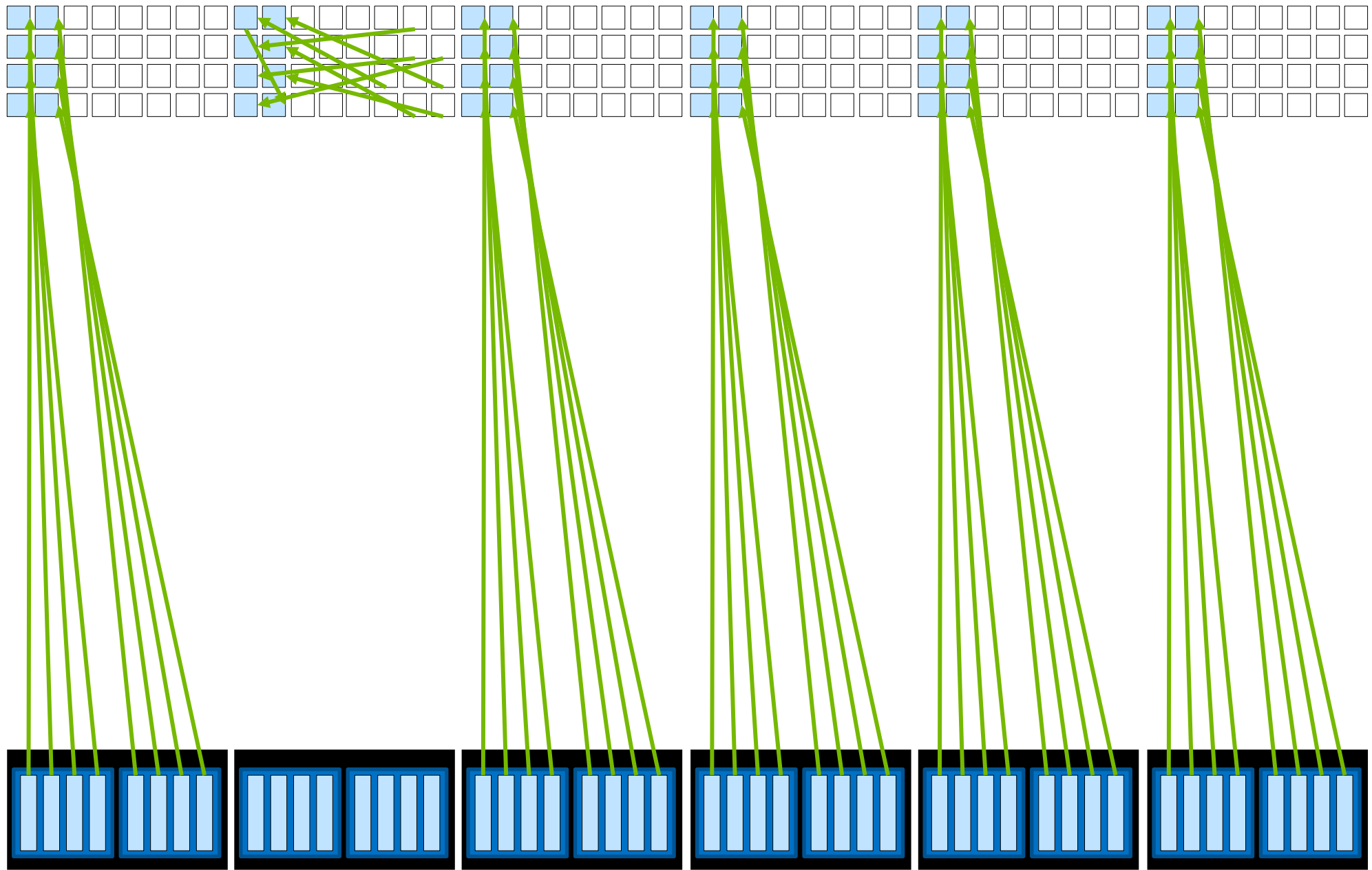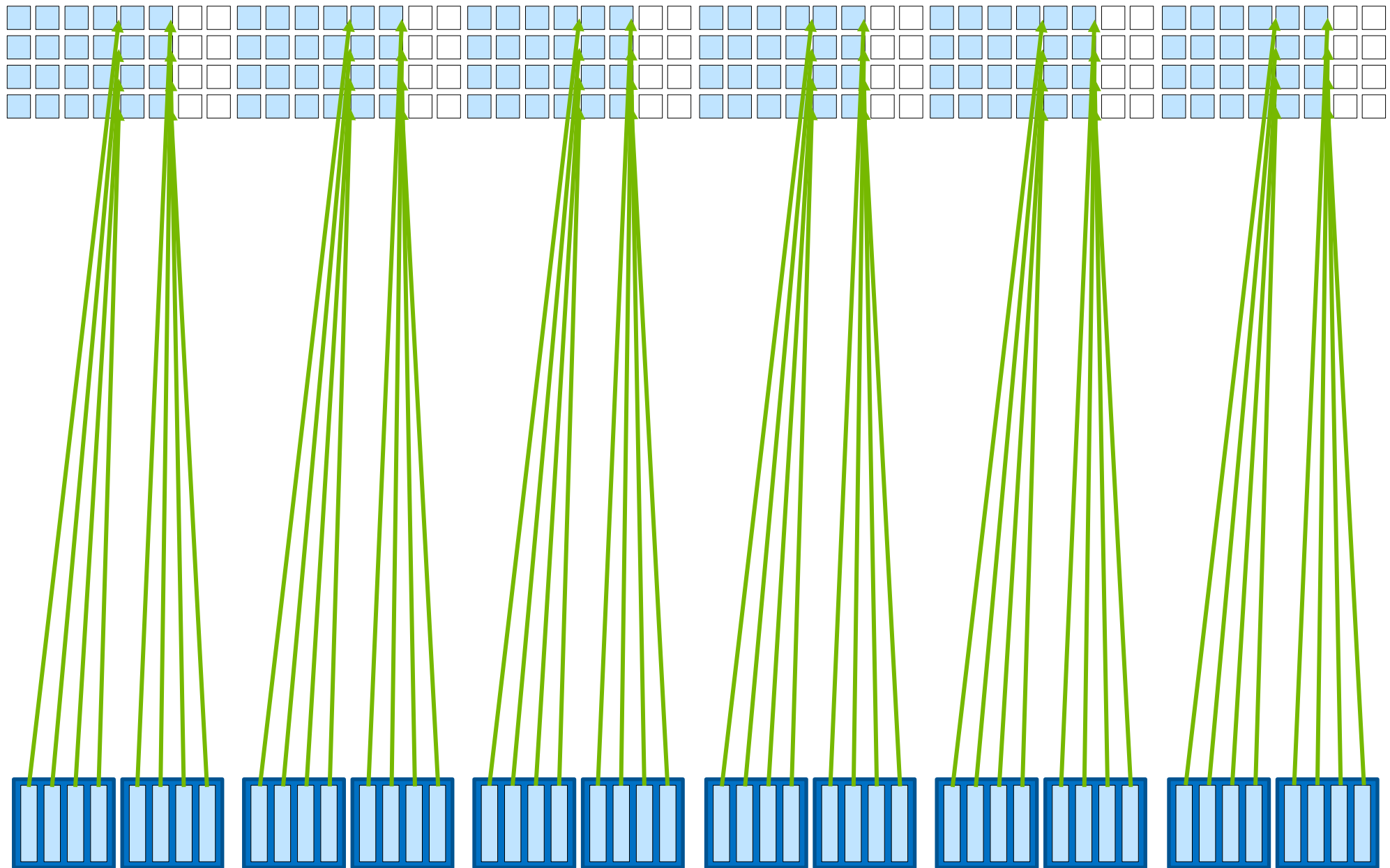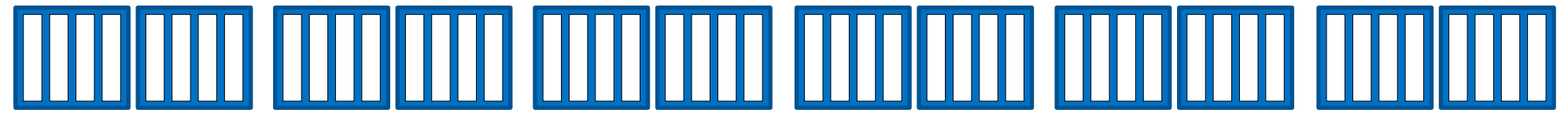
Glossary

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.
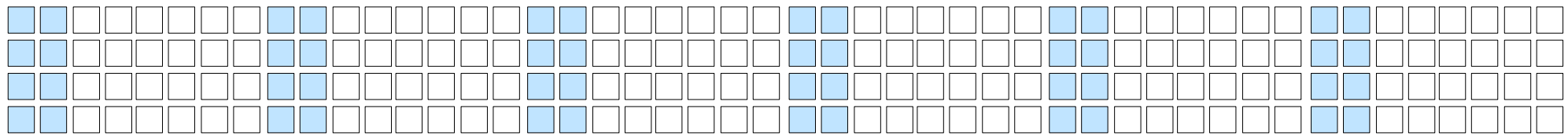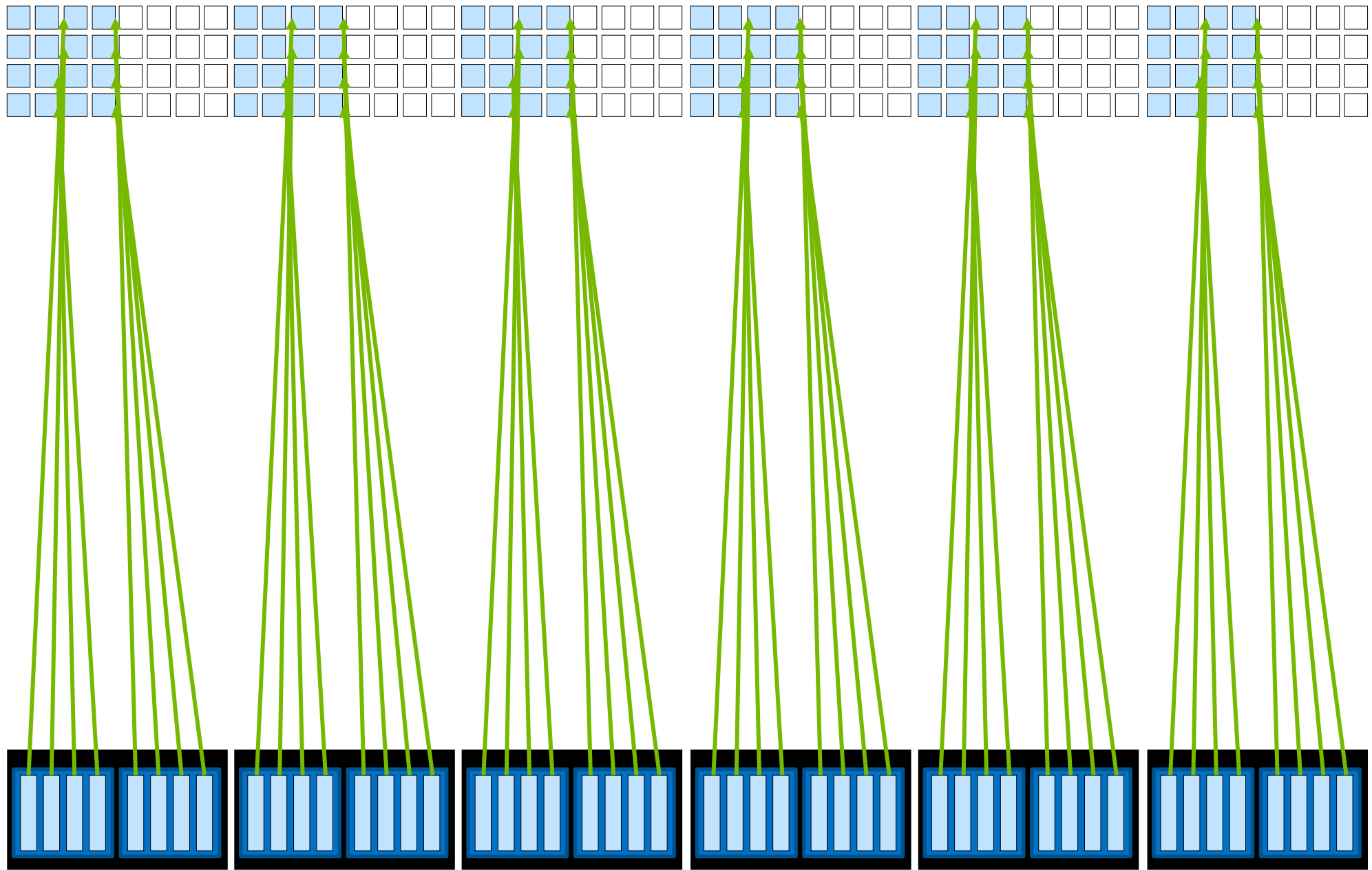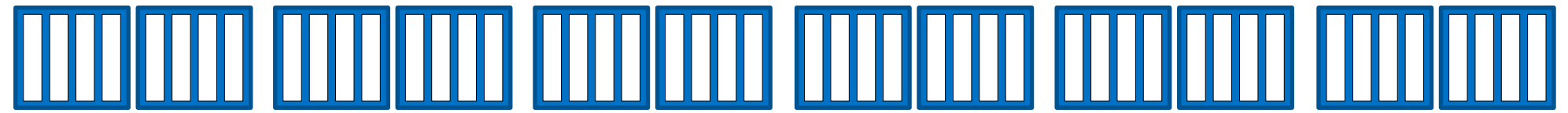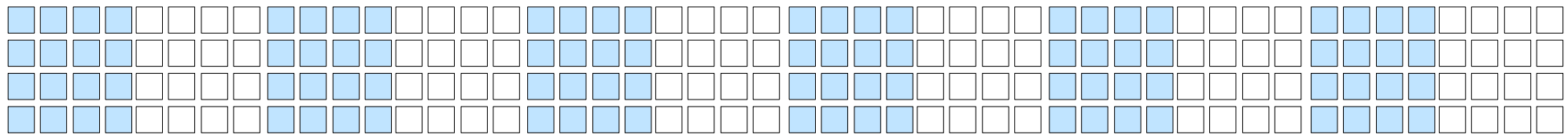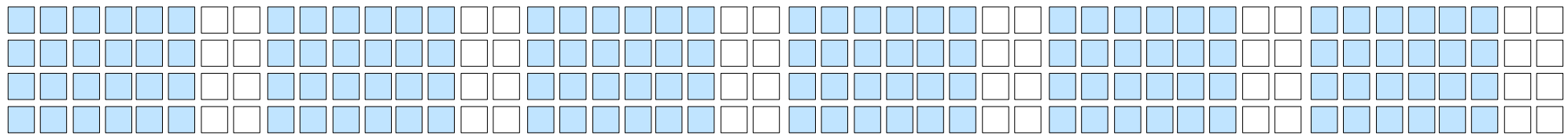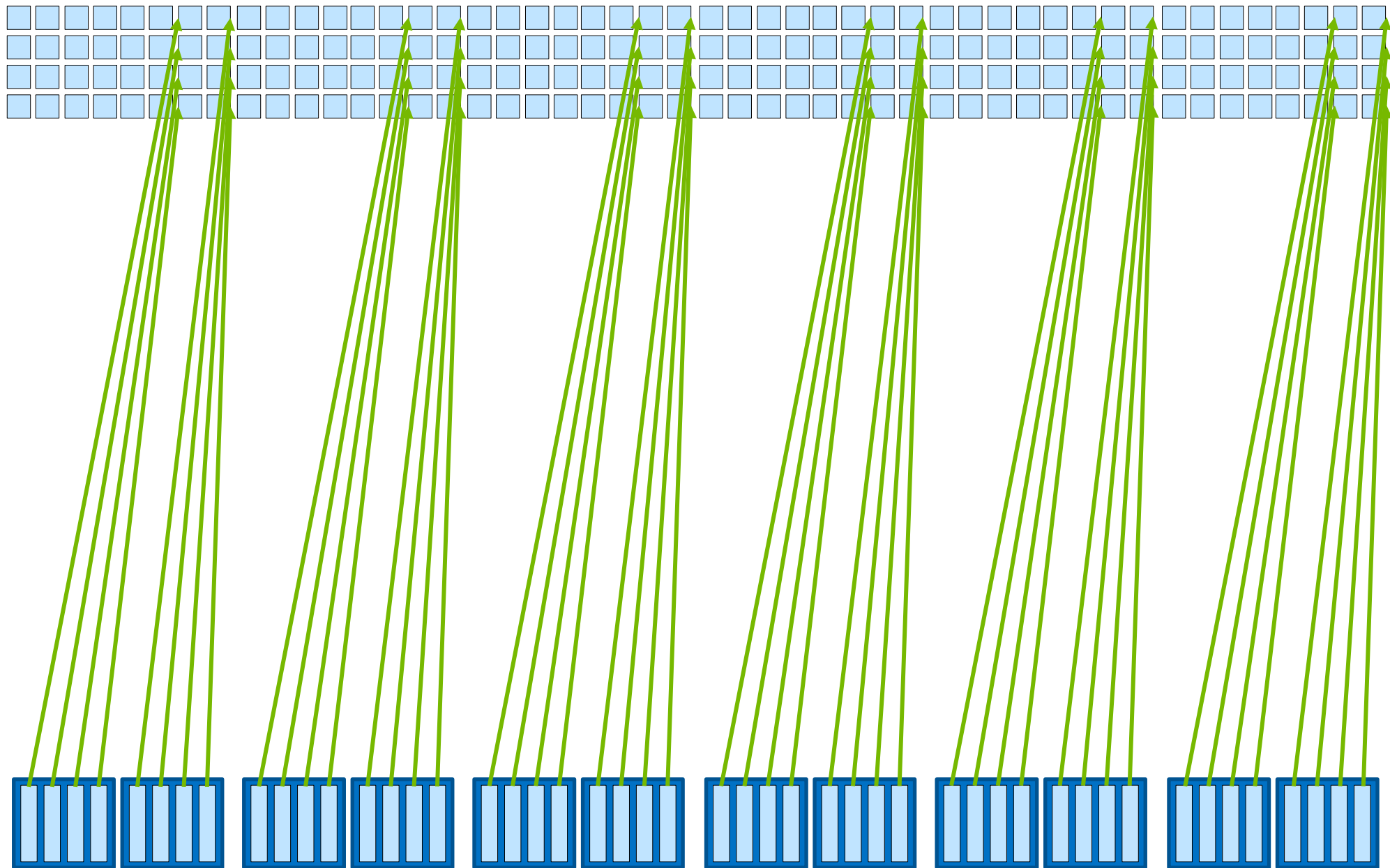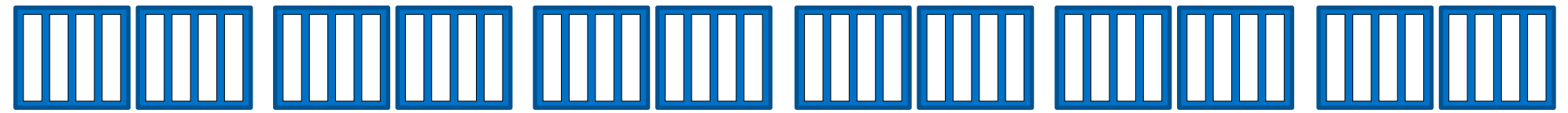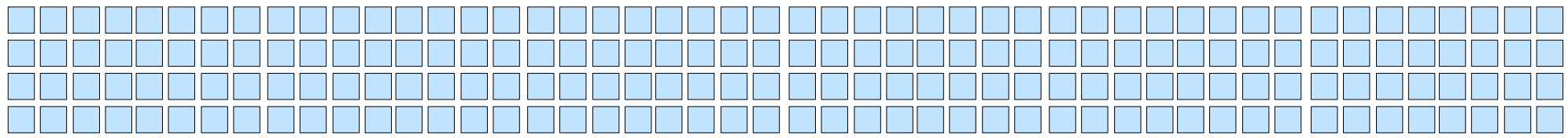
— **Kernel:** A CUDA function executed on a GPU.

— **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.