

Accelerated computing with CUDA C/C++

Dr. Jony Castagna

FAST team leader - Hartree Centre

NVidia Deep Learning Institute Ambassador

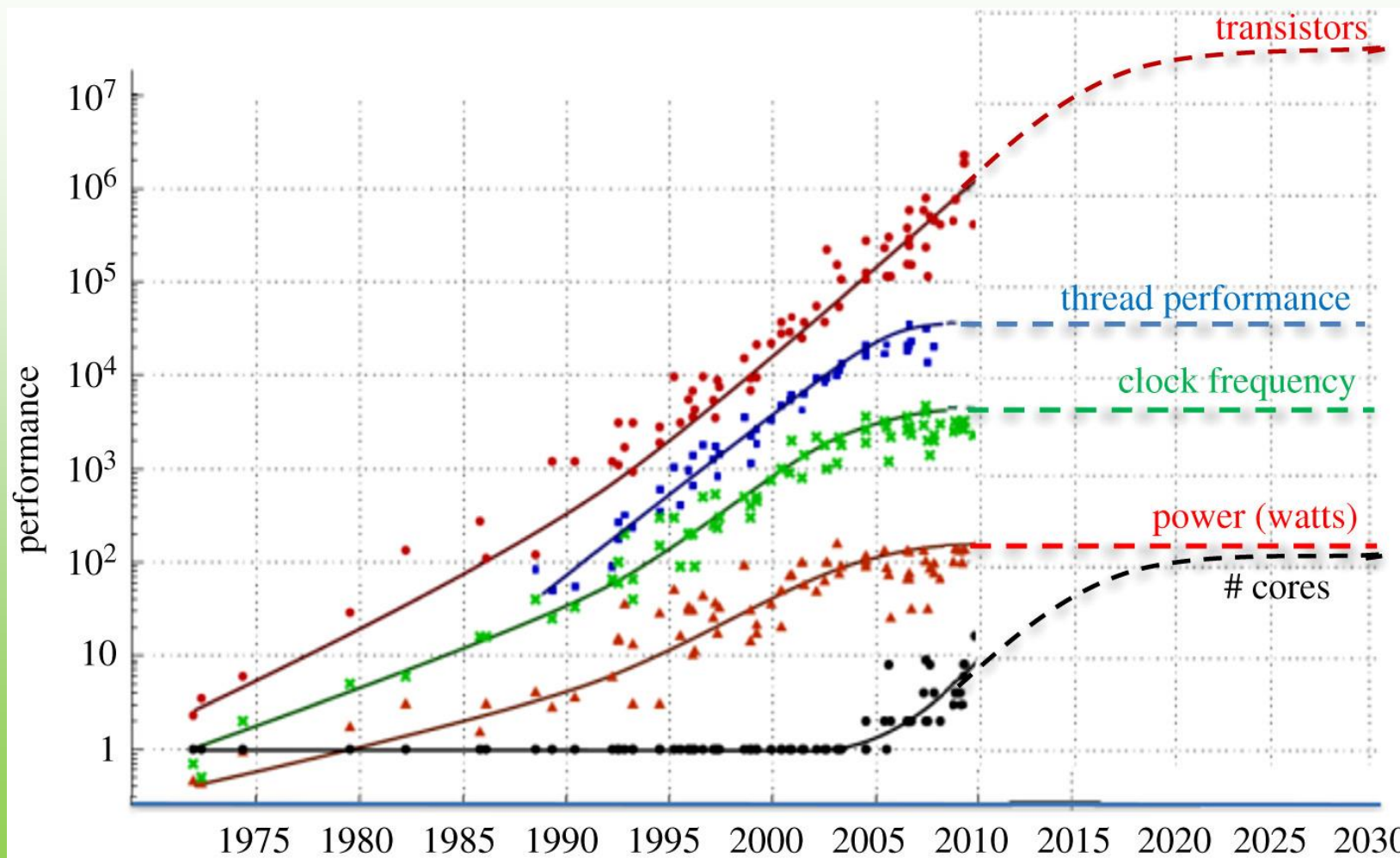


Agenda

- What are GPUs?
- What is CUDA?
- CUDA Threads
- Memory management
- CUDA Streams
- Alternative programming models
- GPU implementations in HEP

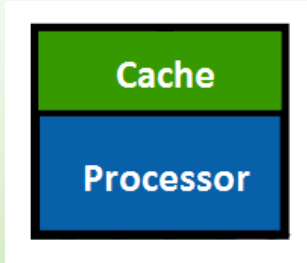


Why we have GPUs?



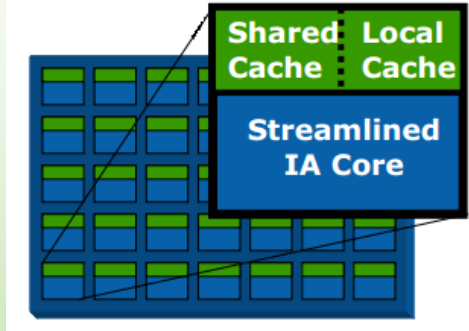
Modern and future architectures

Single Core Processor



Long pipelined,
out-of-order
execution

Many-core Processor



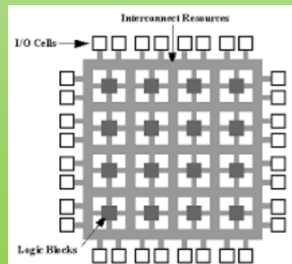
Short pipelined,
cache coherent

GPU

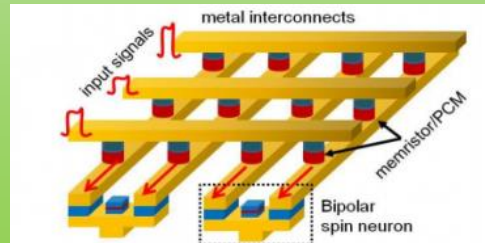


Shared instruction
control, small cache

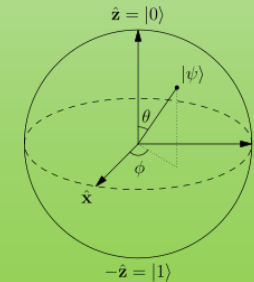
Field-Programmable
Gate Arrays



Neuromorphic
Computing



Quantum
Computing



First 10 of top 500 supercomputers

9/10
have
GPUs!

2 AMD
1 Intel
6 Nvidia

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier - HPE Cray EX2255a, AMD Optimized 3rd Generation EPYC 64C 26Hz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	Aurora - HPE Cray EX - Intel Extreme Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	Orion - Microsoft NDv5, Xeon Platinum 8480C 48C 26Hz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.26Hz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	LUMI - HPE Cray EX2255a, AMD Optimized 3rd Generation EPYC 64C 26Hz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107
6	Alps - HPE Cray EX2255a, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Swiss National Supercomputing Centre (OSOS) Switzerland	1,305,600	270.00	353.75	5,194
7	Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 80GB 3.0GHz, Infiniband HDR100 Infiniband, EVIDEN EuroHPC/CINECA Italy	1,824,768	241.20	306.31	7,494
8	MareNostrum 5 ACC - BullSequana XH2000, Xeon Platinum 8460Y+ 32C 2.36GHz, NVIDIA H100 60GB, Infiniband NDR, EVIDEN EuroHPC/BSC Spain	663,040	175.30	249.44	4,159
9	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
10	Eos - NVIDIA DGX SuperPOD - NVIDIA DGX H100, Xeon Platinum 8480C 48C 2.6GHz, NVIDIA H100, Infiniband NDR400, Nvidia NVIDIA Corporation United States	485,888	121.40	188.65	

account the Turbo CPU clock rate where it applies.

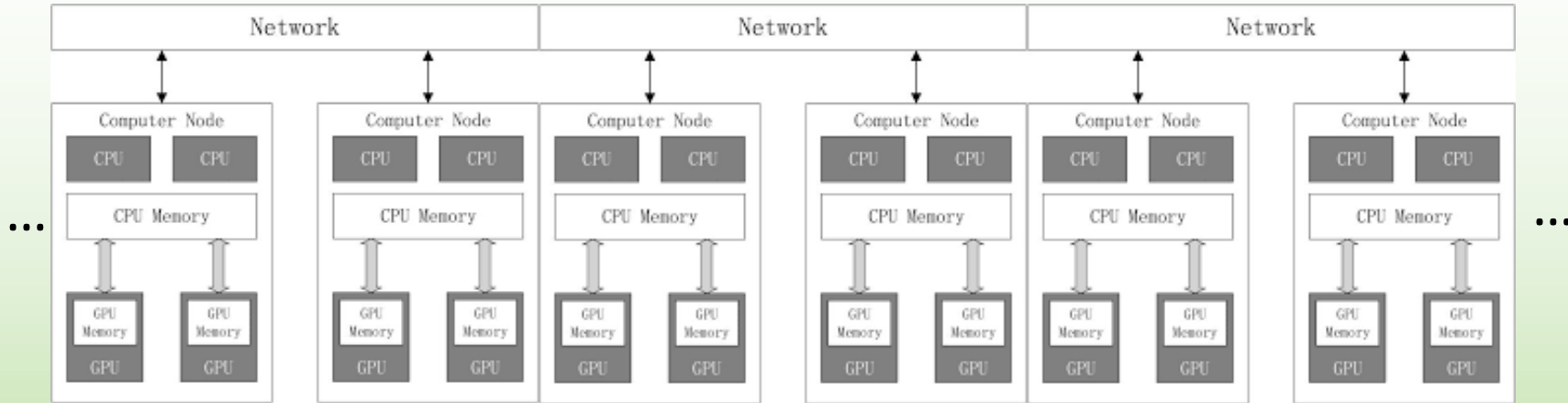
Green500 Data

Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	Power (kW)	Energy Efficiency (GFlops/watts)
1	189	JEDI - BullSequana XH2000, Grace Hopper Superchip 72C 3GHz, NVIDIA GH200 Superchip, Infiniband NDR200, ParTec/EVIDEN EuroHPC/FZJ Germany	19,584	4.50	67	72.723
2	128	Orion - AMD AI phase 1 - HPE Cray EX2255a, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE University of Bristol United Kingdom	34,272	7.42	117	68.835
3	55	Helios GPU - HPE Cray EX2255a, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cyfronet Poland	89,760	19.14	317	66.948
4	328	Henri - ThinkSystem S550C 1588 Xeon Platinum 8348C 2.8GHz, NVIDIA H100 80GB PCIe, Infiniband HDR, Lenovo Fujitsu Institute United States	1,000	2.88	44	65.396
5	71	preAlps - HPE Cray EX225a, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Swiss National Supercomputing Centre (CSCS) Switzerland	81,600	15.47	240	64.381
6	299	Orion - ThinkSystem S550C 1588 V3, AMD EPYC 9304 32C 3.25GHz, Nvidia H100 94Gb SXM5, Infiniband NDR200, Lenovo Forschungszentrum für Technologie (KIT) Germany	13,616	3.12	50	62.964
7	54	Frontier TDS - HPE Cray EX2255a, AMD Optimized 3rd Generation EPYC 64C 26Hz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	120,832	19.20	309	62.684
8	11	Venado - HPE Cray EX2255a, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE DOE/NNSA/Oak Ridge National Laboratory United States	481,440	98.51	1,662	59.287
9	20	Adastra - HPE Cray EX2255a, AMD Optimized 3rd Generation EPYC 64C 26Hz, AMD Instinct MI250X, Slingshot-11, HPE Grand Equipement National de Calcul Informatique - Centre Informatique National de l'Enseignement Supérieur (GENCI-CINES) France	319,072	46.10	921	58.021
10	28	Setonix - BRU - HPE Cray EX2255a, AMD Optimized 3rd Generation EPYC 64C 26Hz, AMD Instinct MI250X, Slingshot-11, HPE Supercomputing Centres, Kensington, Western Australia Australia	181,248	27.16	477	56.983

Green list
73 Gflops/Watts

6 Nvidia
4 AMD

Typical hybrid CPU-GPU supercomputer



Each node is made of
**1 (or more) CPUs and
1 (or more) GPUs**

Example: Frontier
1 CPUs AMD EPYC (64-core)
4 GPUs AMD Instinct 250X

What are GPUs?



you don't necessarily need a cluster!



37,888 [Instinct](#) MI250X GPUs!



NVidia GPUs

Tesla/Data Center
(HPC)

- Tesla (1.x)
- Fermi (2.x)
- Kepler (3.x)
- Maxwell (5.x)
- Pascal (6.x)
- Volta (7.x)
- Turing (7.5)
- Ampere (8.x)
- Lovelace (8.9)
- Hopper (9.x)
- Blackwell (10.x)

~~Quadro~~ RTX
(Visualization)

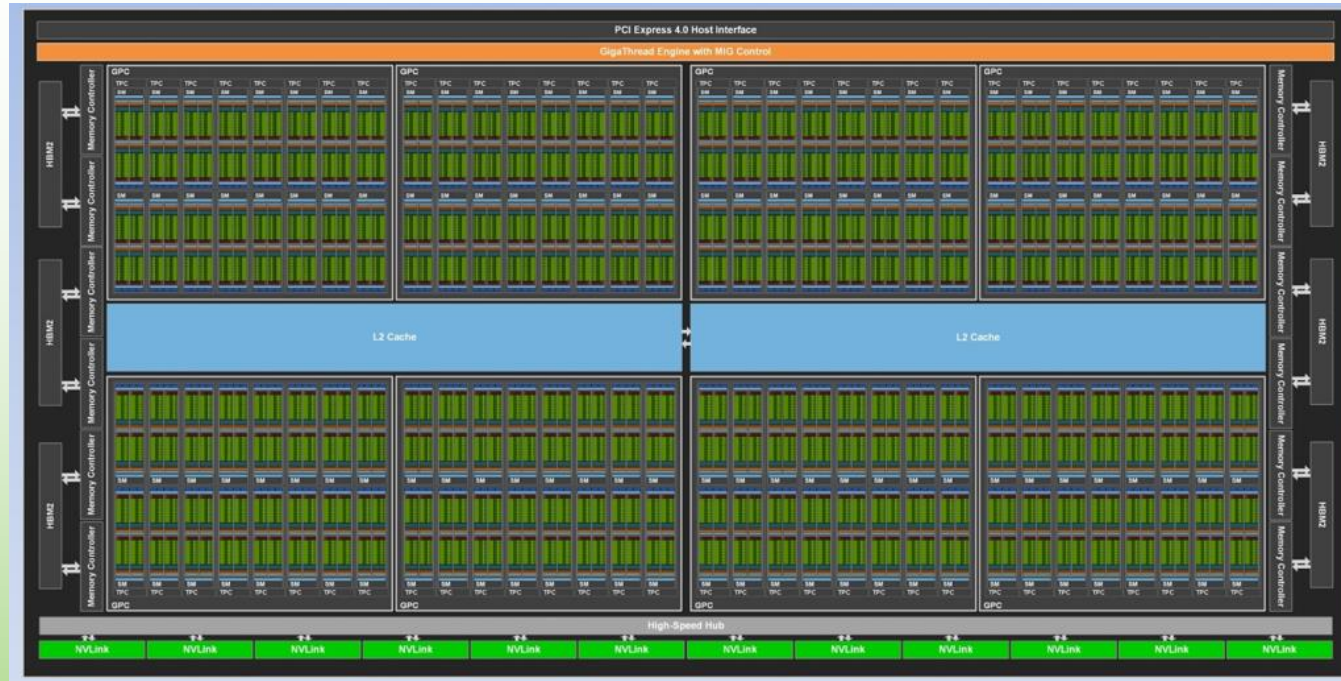
GeForce
(Gaming)

Jetson/Tegra
(edge/auto)

Architecture identifier also corresponding to the major number of Compute Capability index



NVidia H100



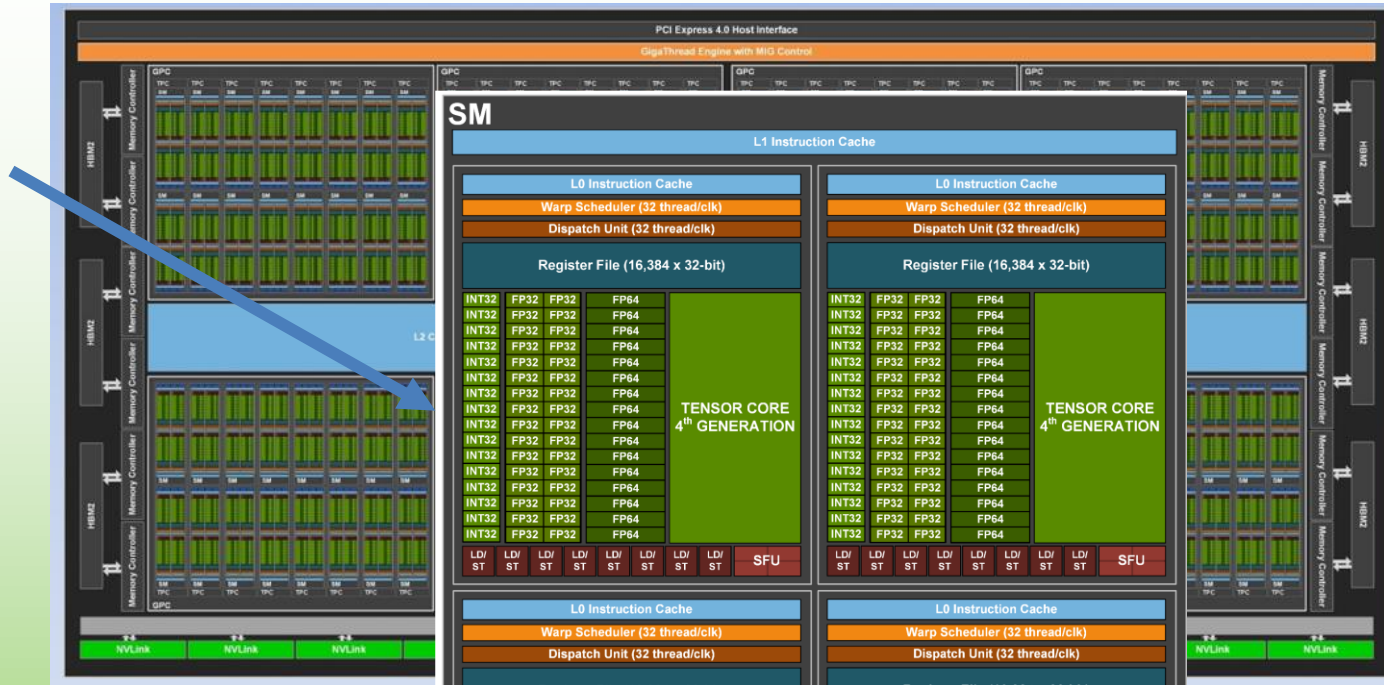
NVidia Tesla H100: 14592 CUDA cores!

- 8 GPCs, 72 TPCs (9 TPCs/GPC), 2 SMs/TPC, 144 SMs per full GPU
- 128 FP32 CUDA Cores per SM, 18432 FP32 CUDA Cores per full GPU
- 4 Fourth-Generation Tensor Cores per SM, 576 per full GPU
- 6 HBM3 or HBM2e stacks, 12 512-bit Memory Controllers
- 60 MB L2 Cache
- Fourth-Generation NVLink and PCIe Gen 5

H100 white paper:
<https://resources.nvidia.com/en-us-tensor-core>



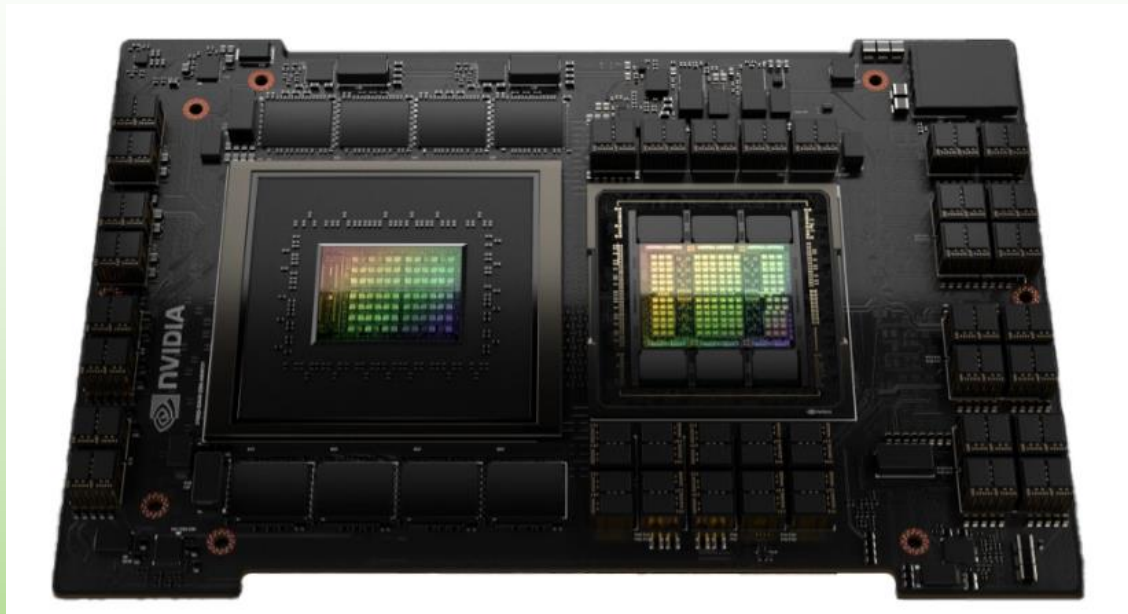
NVidia H100



Nvidia GH100 SM architecture

Full GPU has 144 SMs!

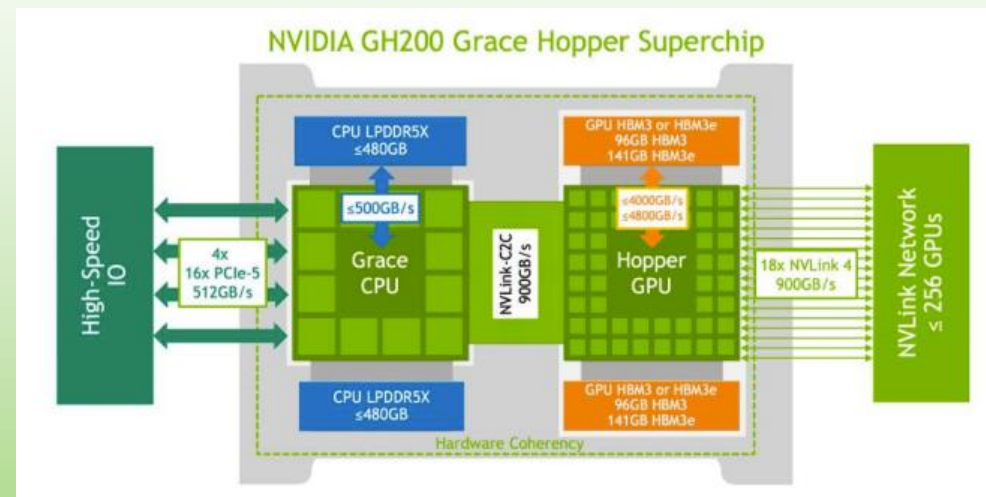
The Grace-Hopper superchip



GH200 White paper

<https://www.aspsys.com/wp-content/uploads/2023/09/nvidia-grace-hopper-cpu-whitepaper.pdf>

High bandwidth and memory coherence!



Useful blog

<https://developer.nvidia.com/blog/simplifying-gpu-programming-for-hpc-with-the-nvidia-grace-hopper-superchip/>



H100 vs H200

Table 2. NVIDIA MGX Grace Hopper Superchip vs. NVIDIA x86+Hopper

Feature per GPU	HGX H100 4-GPU (x86)	NVIDIA MGX GH200 with HBM3	NVIDIA MGX GH200 with HBM3e	NVIDIA DGX GH200
CPU Memory bandwidth (GB/s / GPU)	Up to 150	Up to 500	Up to 500	Up to 500
GPU Memory bandwidth (GB/s / GPU)	3000	4000	4800	4000
CPU Memory bandwidth to GPU Memory bandwidth ratio	5%	12.5%	10.4%	12.5%
GPU-CPU Link bi-directional bandwidth (GB/s / GPU)	128 (x16 PCIe Gen5)	900 (NVLink-C2C)	900 (NVLink-C2C)	900 (NVLink-C2C)
GPU-GPU bi-directional bandwidth inter node (GB/s / GPU)	100 (InfiniBand NDR400)	100 (InfiniBand NDR400)	900 (NVLink 4 for dual GH200 with HBM3e) 100 (InfiniBand NDR400)	900 (NVLink 4)

These improvements in CPU ratio, and NVLink-C2C and NVLink Switch System performance redefine how we achieve maximum performance from heterogeneous systems, enabling new applications, and efficient solutions to challenging problems.



DGX-GH200

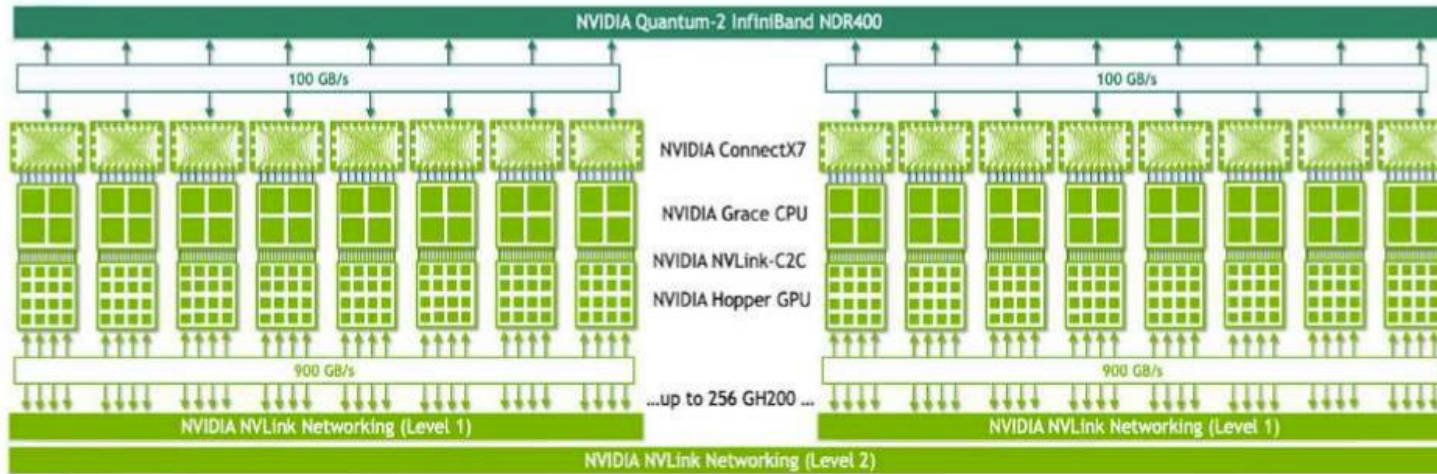


Figure 4. NVIDIA DGX GH200 with NVLink Switch System for strong-scaling giant ML workloads

up to 256 GPUs!

1 exaFlops FP8!

~£10M!



HPC and AI (I)



HPC

(\$45 Billion in 2022)



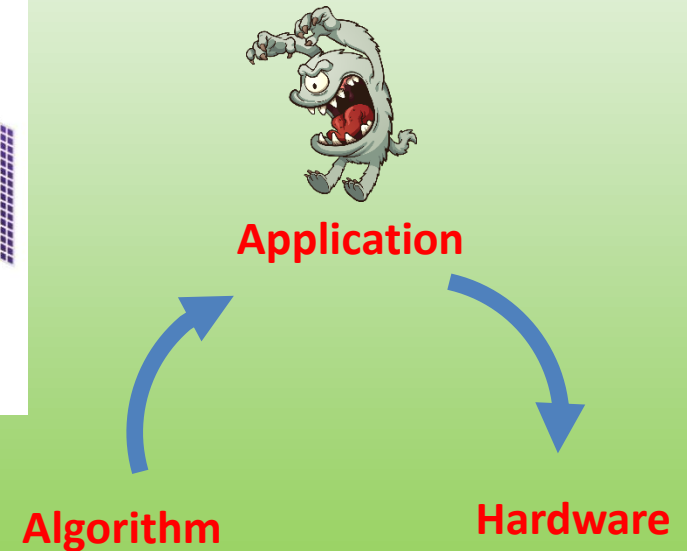
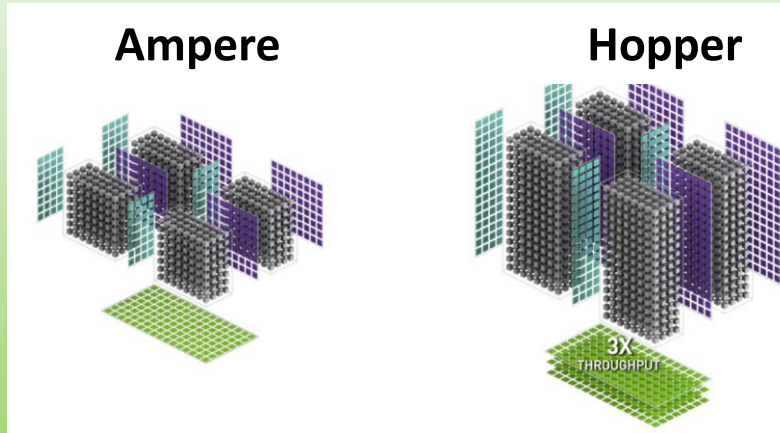
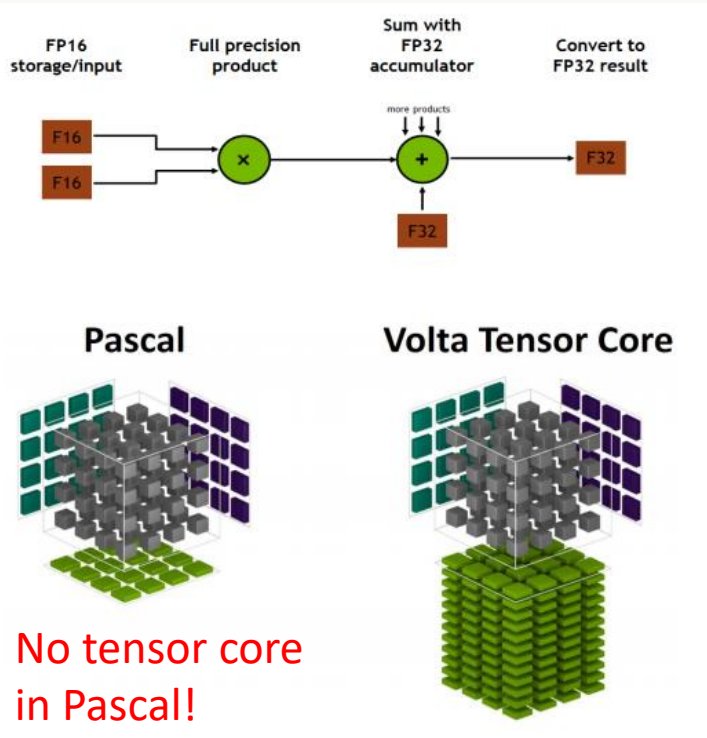
AI
(\$200 billions
in 2023!)

no worry... likes GPUs!



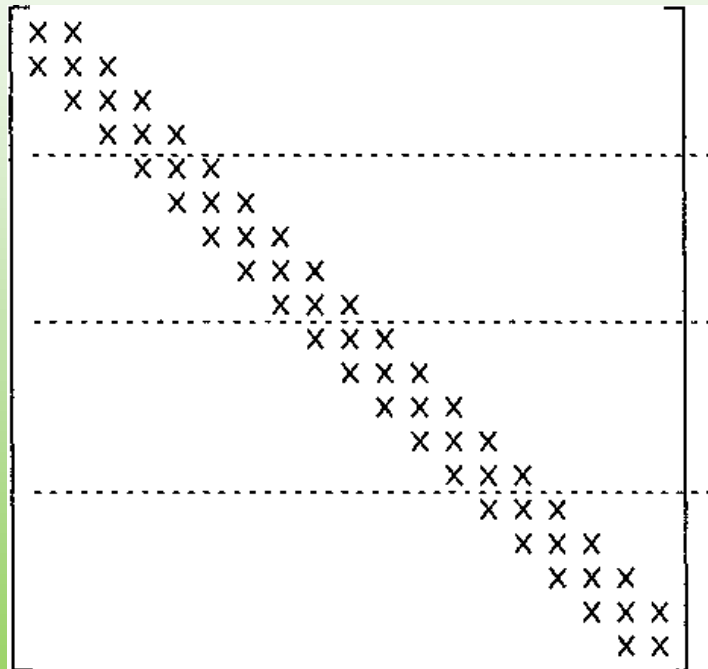
HPC and AI (II)

Deep Learning does NOT need double precision!



Why is important to know the GPU hardware? (I)

Solve a Tridiagonal matrix in parallel:

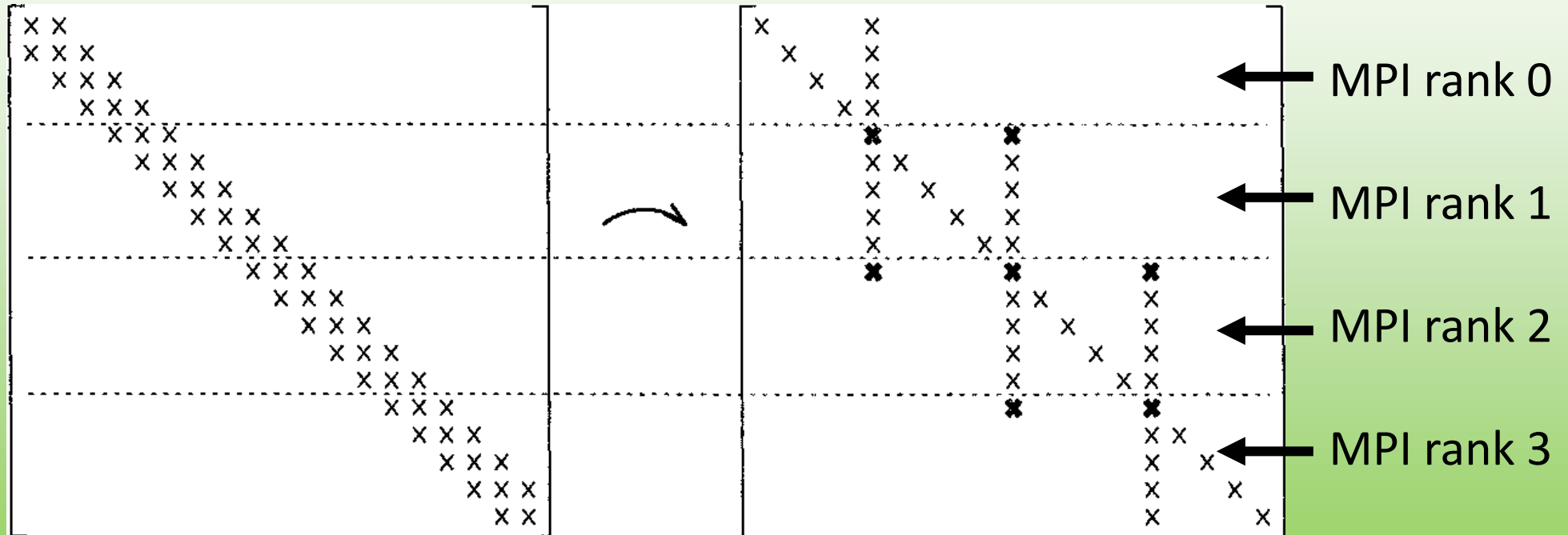


Thomas (sequential)
algorithm
for 1 CPU



Why is important to know the hardware? (II)

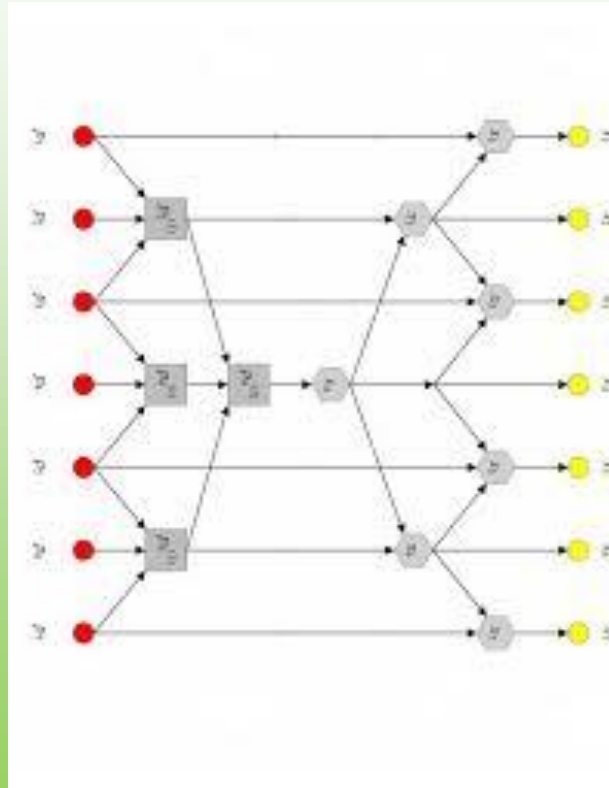
Solve a Tridiagonal matrix in parallel:



For MPI/OpenMP you can use a partition method

Why is important to know the hardware? (III)

Solve a Tridiagonal matrix in parallel:

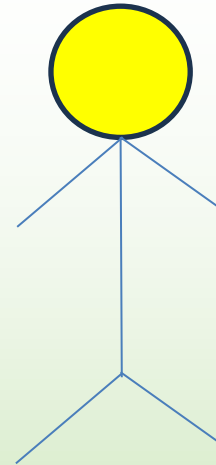
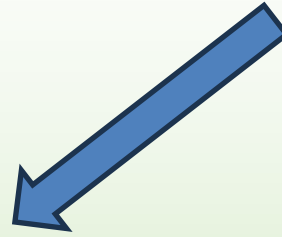
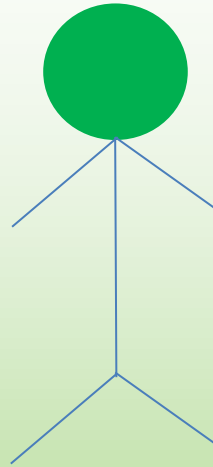
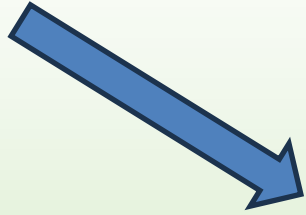
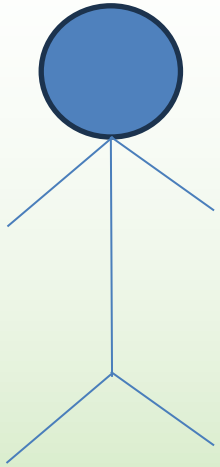


Each thread simplify
a pair of equations.

On GPU you MUST use a cyclic reduction method!



Your role!



Scientist (usually):

- counts from 1
- likes Fortran
- interested in solving PDE
- does not like AI solving everything

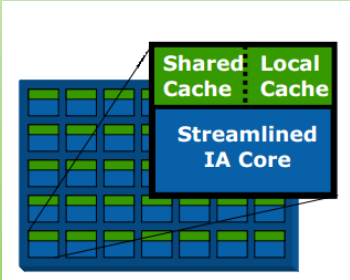
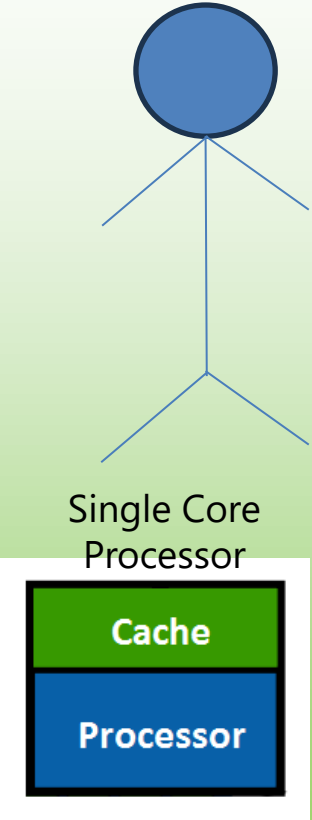
Computational scientist (usually):

- likes all languages
- wants to solve PDE efficiently (fast) and everywhere (portability)
- AI can help if combined with a physics background

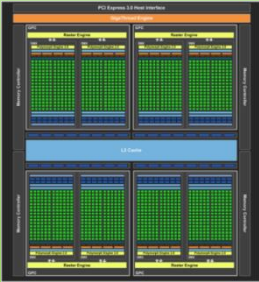
Computer scientist (usually):

- counts from 0
- likes C/C++/Python
- interested in performance/hardware
- likes an AI super-intelligence

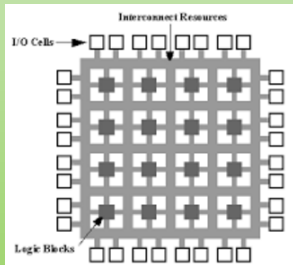
Modern and future computational scientist!



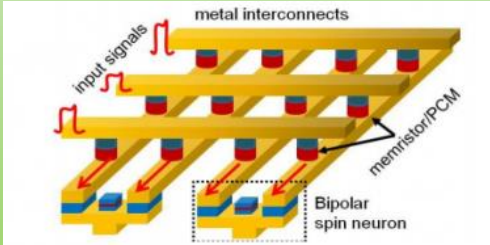
Many-core Processor



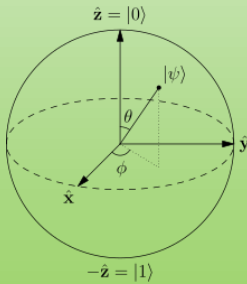
GPU



Field-Programmable Gate Arrays



Neuromorphic Computing



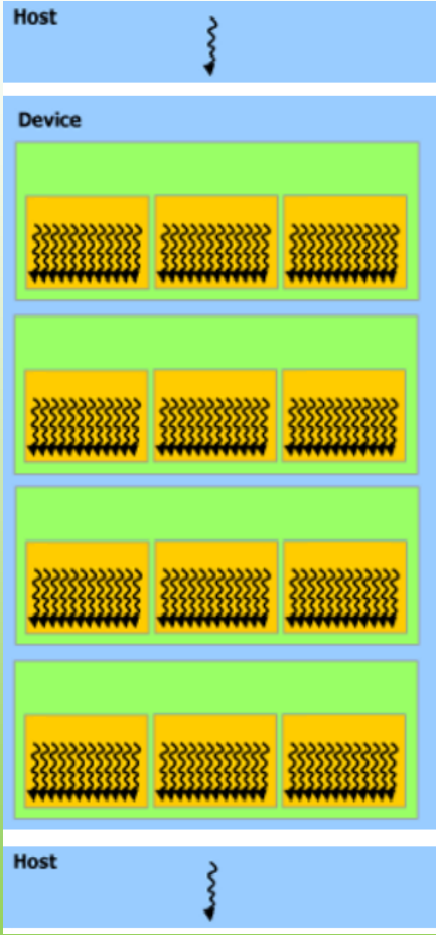
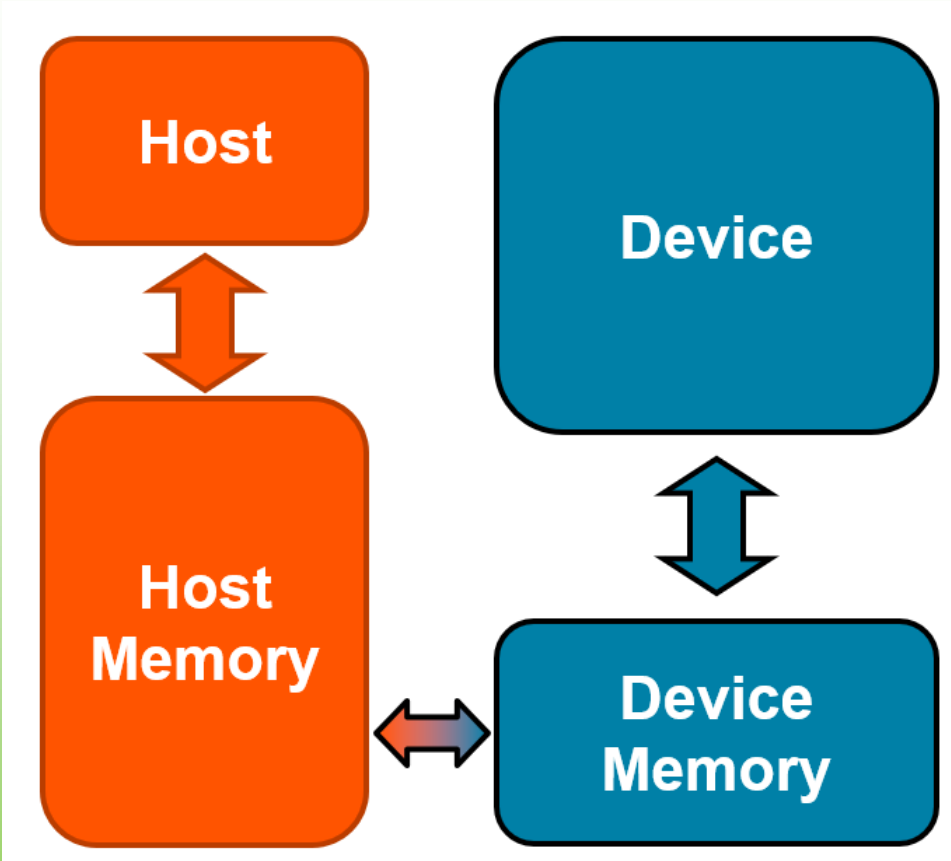
Quantum Computing



today we are here!



1st Main Concepts: host (CPU) and device (GPU)



Independent from the vendor!!!



2nd Main Concepts : software abstraction

Technical specifications	Compute capability (version)																
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0 (7.2?)	7.5
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4			32		16	128	32	16		128
Maximum dimensionality of grid of thread blocks	2				3												
Maximum x-dimension of a grid of thread blocks	65535					2 ³¹ - 1											
Maximum y-, or z-dimension of a grid of thread blocks	65535																
Maximum dimensionality of thread block	3																
Maximum x- or y-dimension of a block	512				1024												
Maximum z-dimension of a block	64																
Maximum number of threads per block	512				1024												
Warp size	32																

you can have billions of CUDA threads despite you only have ~15000 CUDA cores on a single GPU!

3rd Main Concepts : Fine Grain Parallelism (I)

A

56	1	33	34	35	2	33	4	5	45	3	4
----	---	----	----	----	---	----	---	---	----	---	---

B

42	1	34	9	7	4	77	6	55	43	2	2
----	---	----	---	---	---	----	---	----	----	---	---

C

98	2	67	43	42	6	110	10	60	88	5	6
----	---	----	----	----	---	-----	----	----	----	---	---



MPI
rank 0
core 0



MPI
rank 1
core 1



MPI
rank 2
core 2



3rd Main Concepts : Fine Grain Parallelism (I)



3rd Main Concepts : Fine Grain Parallelism (II)

A

56	1	33	34	35	2	33	4	5	45	3	4
----	---	----	----	----	---	----	---	---	----	---	---

B

42	1	34	9	7	4	77	6	55	43	2	2
----	---	----	---	---	---	----	---	----	----	---	---

C

98	2	67	43	42	6	110	10	60	88	5	6
----	---	----	----	----	---	-----	----	----	----	---	---



1 CUDA threads per each element of the array!

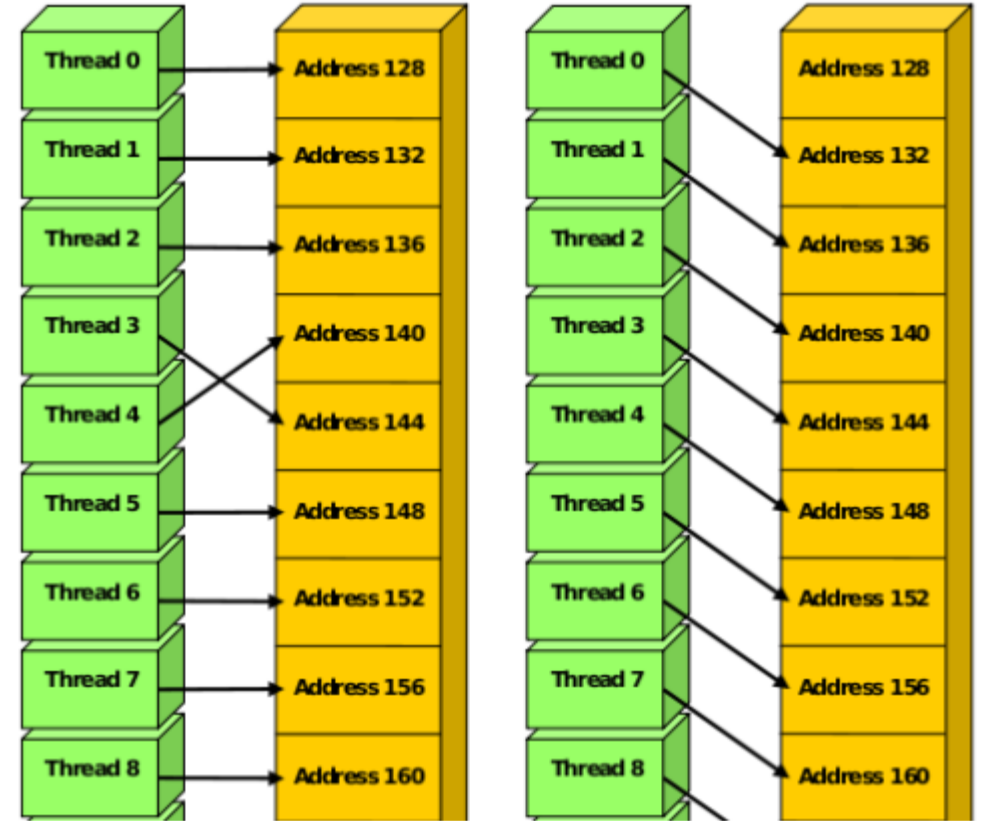
you can have millions of CUDA threads despite you only have ~15000 CUDA cores on a single GPU!



4th Main Concepts : memory coalescent



Coalescent Access



Uncoalescent Access

Resume Main Concepts

- Host (CPU) and device (GPU)
- Software Abstraction
- Fine Grain Parallelism Paradigm
- Memory Coalescent Access

The real challenge is NOT porting to CUDA-C

The real challenge is to satisfy those main concepts: an algorithm change may be required!



Very often from the algorithm change will benefit also others architectures (vectorization, etc...)



Questions?



What is CUDA?

Compute Unified Device Architecture: is a parallel computing platform and application programming interface (API) model created by NVidia.

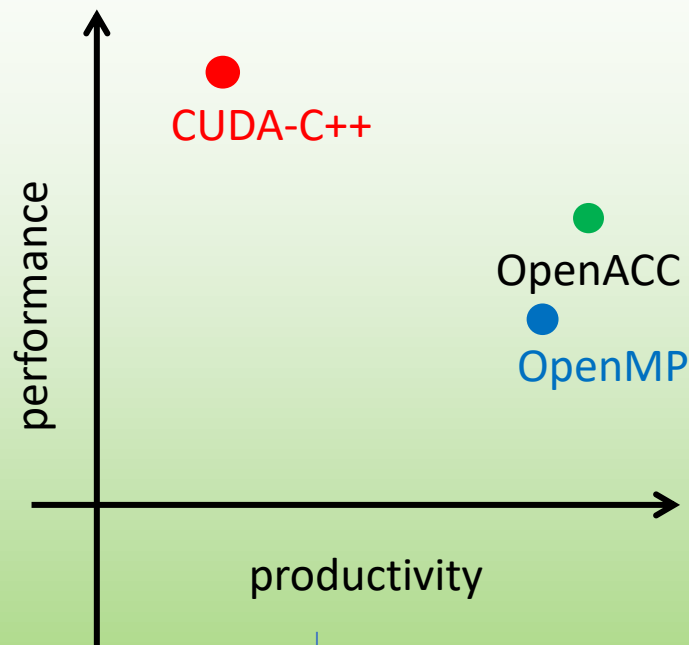
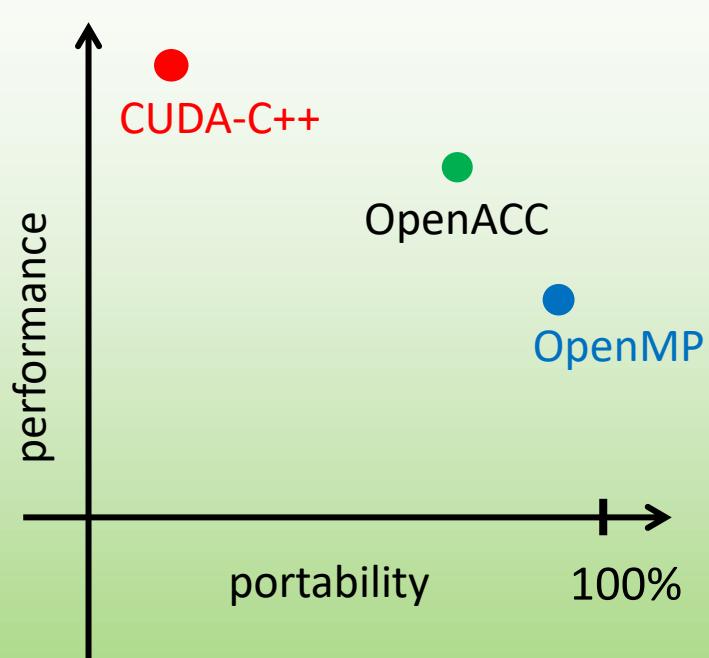
- CUDA is based on C/C++ language
- CUDA Fortran is a Fortran wrapper for CUDA C
- OpenACC are directives to offload kernels on GPU. It translates to CUDA-C.

Main References:

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>



Why CUDA?



Compilers currently supporting OpenACC

- NVidia (PGI) (Fortran/C/C++)
- Cray (Fortran)
- GCC (Fortran/C/C++)



The future of parallel programming

Standard Languages | Directives | Specialized Languages

```
std::transform(par, x, x+n, y, y,  
              [=] (float x, float y) {  
                  return y + a*x;  
              });
```

```
do concurrent (i = 1:n)  
  y(i) = y(i) + a*x(i)  
enddo
```

GPU Accelerated
C++ and Fortran

```
#pragma acc data copy(x,y)  
{  
  ...  
  std::transform(par, x, x+n, y, y,  
                [=] (float x, float y) {  
                    return y + a*x;  
                });  
  ...  
}
```

Incremental Performance
Optimization with Directives

```
global  
void saxpy(int n, float a,  
          float *x, float *y) {  
  int i = blockIdx.x*blockDim.x +  
          threadIdx.x;  
  if (i < n) y[i] += a*x[i];  
}  
  
int main(void) {  
  ...  
  cudaMemcpy(d_x, x, ...);  
  cudaMemcpy(d_y, y, ...);  
  
  saxpy<<<(N+255)/256,256>>>(...);  
  
  cudaMemcpy(y, d_y, ...);  
}
```

Maximize GPU Performance
with CUDA C++/Fortran

The NVIDIA HPC compilers split execution of an application across multicore CPUs and NVIDIA GPUs using standard language constructs, directives, or CUDA.



However...

CPU

AMD EPYC 7713 processor 2 GHz 256 MB L3

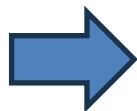
£1,097.41 ex VAT

£1,316.89 inc VAT

Specifications

PROCESSOR		MEMORY	
Processor manufacturer	AMD	Memory types supported by processor	DDR4-SDRAM
Processor model	7713	Memory clock speeds supported by processor	3200 MHz
Processor base frequency	2 GHz	Memory channels	Octa-channel
Processor family	AMD EPYC	Memory bandwidth (max)	204.8 GB/s
Processor cores	64		

price ratio ~ 10
energy ratio ~ 2
memory bandwidth ~ 15



**At least >2x
and ideally
>10x!!!**

GPU

about 10k USD

The price of an AMD Instinct MI250X is about 10k USD, so 10000 of them are about 100 million USD. This would suggest that the actual workhorse of the supercomputer is only 20% of its cost (since Frontier cost 600 million USD). 18 Jun 2022

AMD Instinct™ MI250X Accelerators

AMD Instinct™ MI250X accelerators are designed to supercharge HPC workloads and p era of exascale

Peak Double Precision Matrix (FP64) Performance 95.7 TFLOPs

GPU Memory

Dedicated Memory Size 128 GB

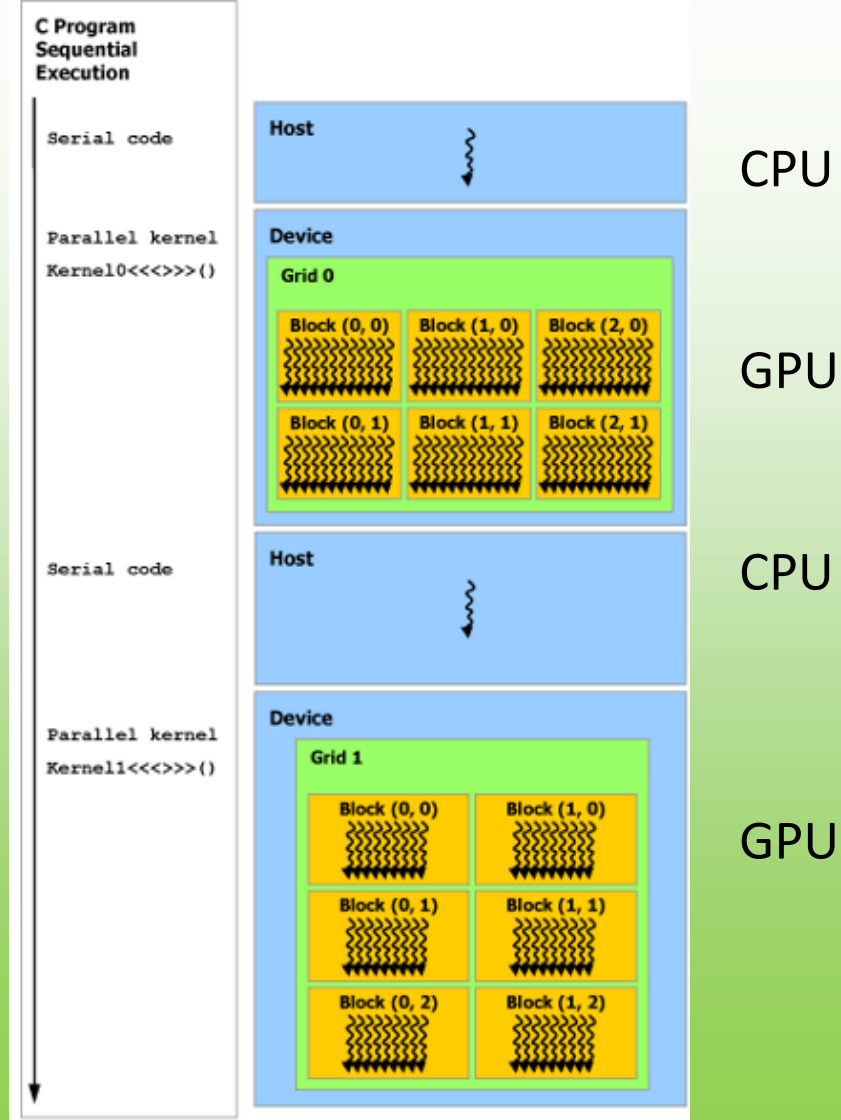
Dedicated Memory Type HBM2e

Memory Interface 8192-bit

Memory Clock 1.6 GHz

Peak Memory Bandwidth 3.2 TB/s

How it works



Transfer CPU (Host) to GPU (Device) is slow: try to avoid as much as possible!



How it works

Standard C Code

```
void saxpy(int n, float a,
           float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

C with CUDA extensions

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

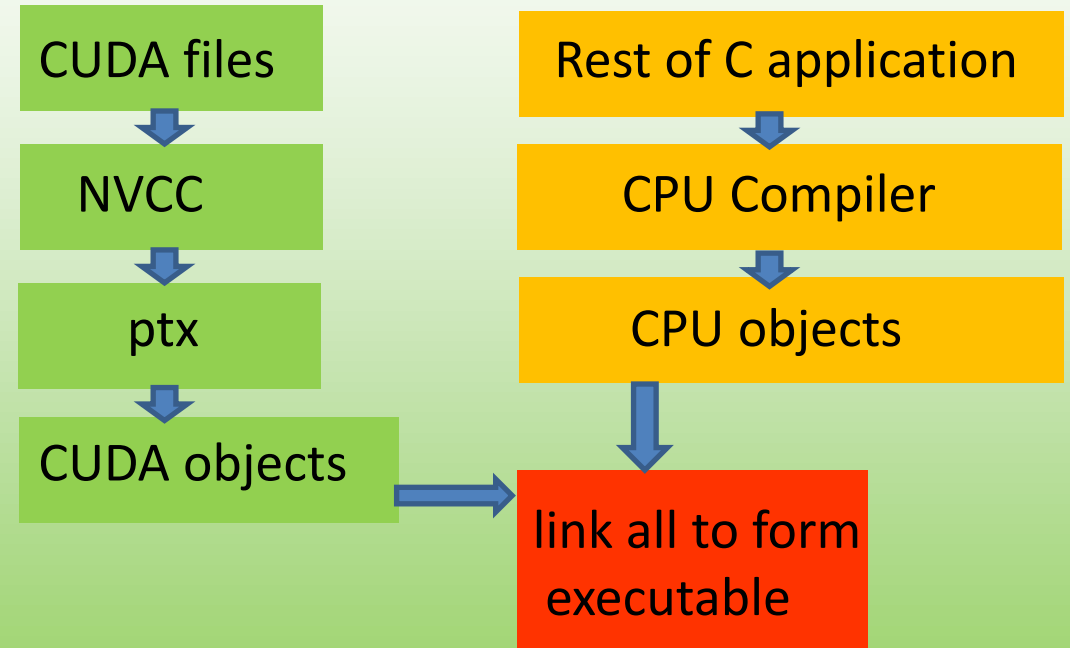
cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```



How it works

C with CUDA extensions

```
__global__  
void saxpy(int n, float a,  
          float *x, float *y)  
{  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    if (i < n) y[i] = a*x[i] + y[i];  
}  
  
int N = 1<<20;  
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);  
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);  
  
// Perform SAXPY on 1M elements  
saxpy<<<4096,256>>>(N, 2.0, x, y);  
  
cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```



CUDA compiler

NVCC (NVIDIA CUDA compiler)

- Very robust!
- Backward compatible
- Released with the **CUDA toolkit (12.6)** or **NVIDIA HPC SDK (24.7)**
- Current version **12.6**

CUDA toolkit version: do NOT confuse with CUDA architecture or Compute Capability index!

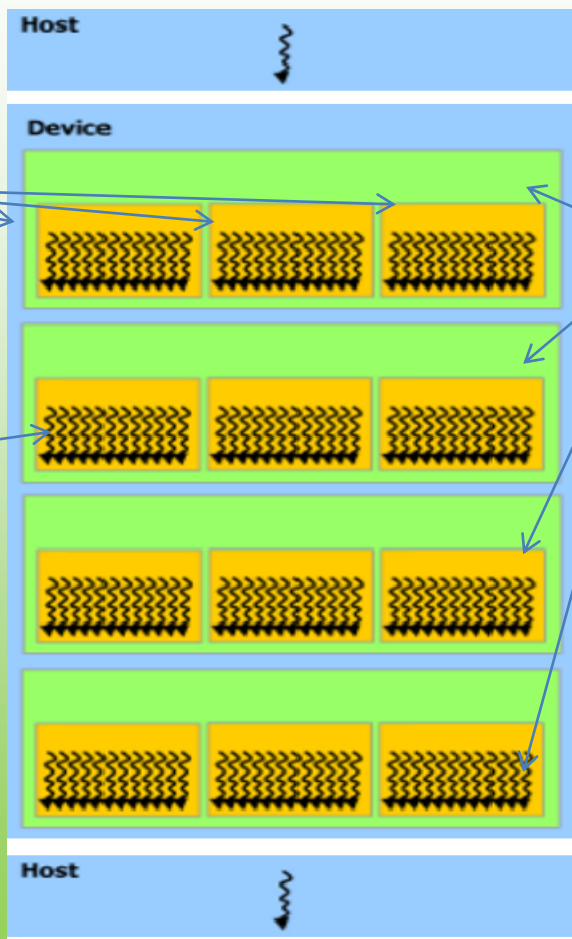


CUDA software abstraction I

blocks

threads

blocks and
threads can be
3D!

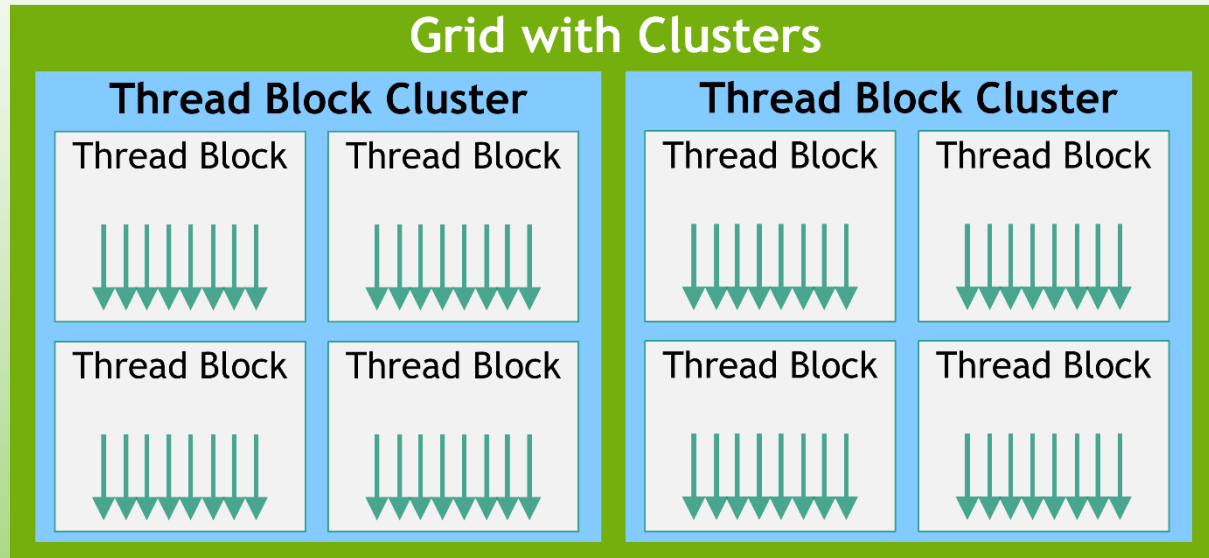


grids

These are
irrespective of the
number of cores!



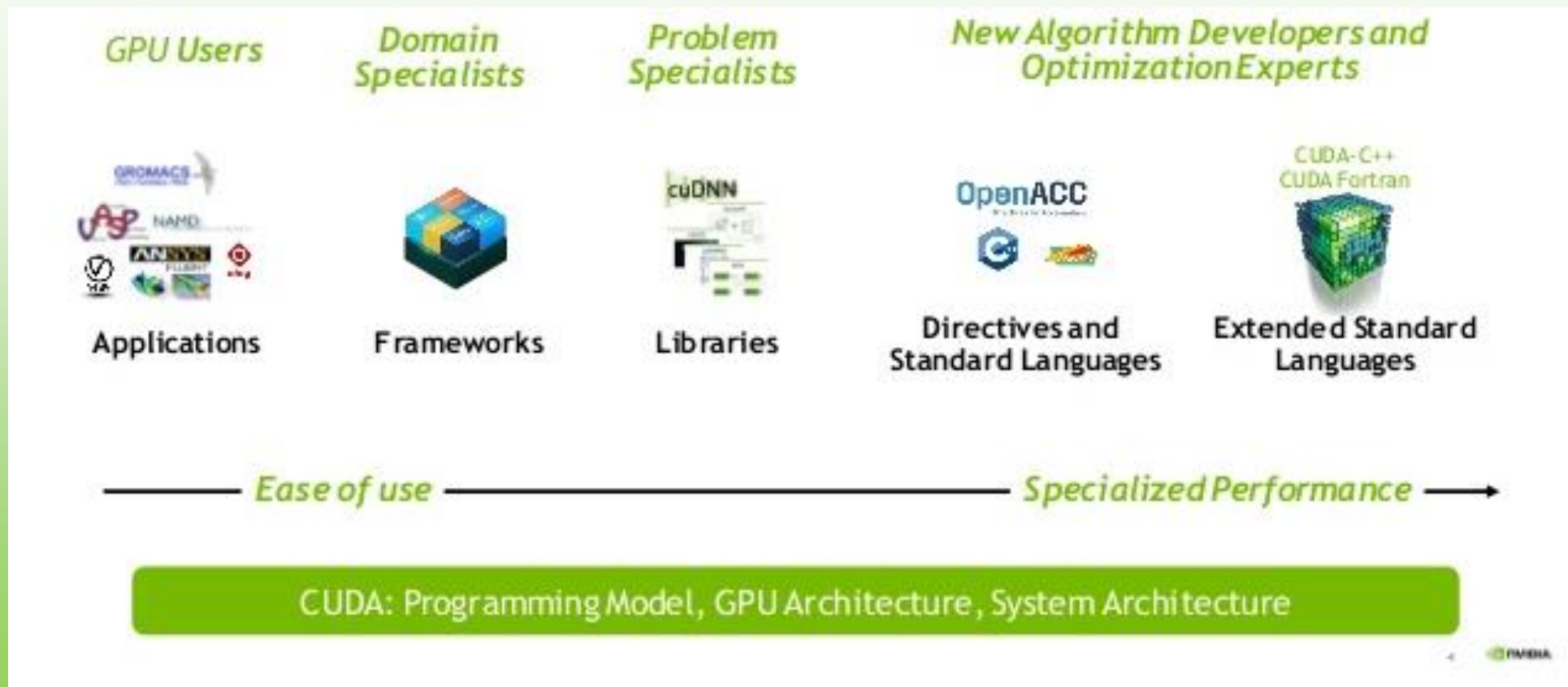
CUDA software abstraction II



**From CUDA
compute
capability 9.0
onwards only!**

**also these can
be 3D!**

CUDA Development Ecosystem (I)



CUDA Development Ecosystem (II)

Tools & Ecosystem



Accelerated Solutions

GPUs are accelerating many applications across numerous industries.

[Learn more >](#)



Performance Analysis Tools

Find the best solutions for analyzing your application's performance profile.

[Learn more >](#)



Key Technologies

Learn more about parallel computing technologies and architectures.

[Learn more >](#)



GPU-Accelerated Libraries

Application accelerating can be as easy as calling a library function.

[Learn more >](#)



Debugging Solutions

Powerful tools can help debug complex parallel applications in intuitive ways.

[Learn more >](#)



Accelerated Web Services

Micro services with visual and intelligent capabilities using deep learning.

[Learn more >](#)



Language and APIs

GPU acceleration can be accessed from most popular programming languages.

[Learn more >](#)



Data Center Tools

Software Tools for every step of the HPC and AI software life cycle.

[Learn more >](#)



Cluster Management

Managing your cluster and job scheduling can be simple and intuitive.

[Learn more >](#)

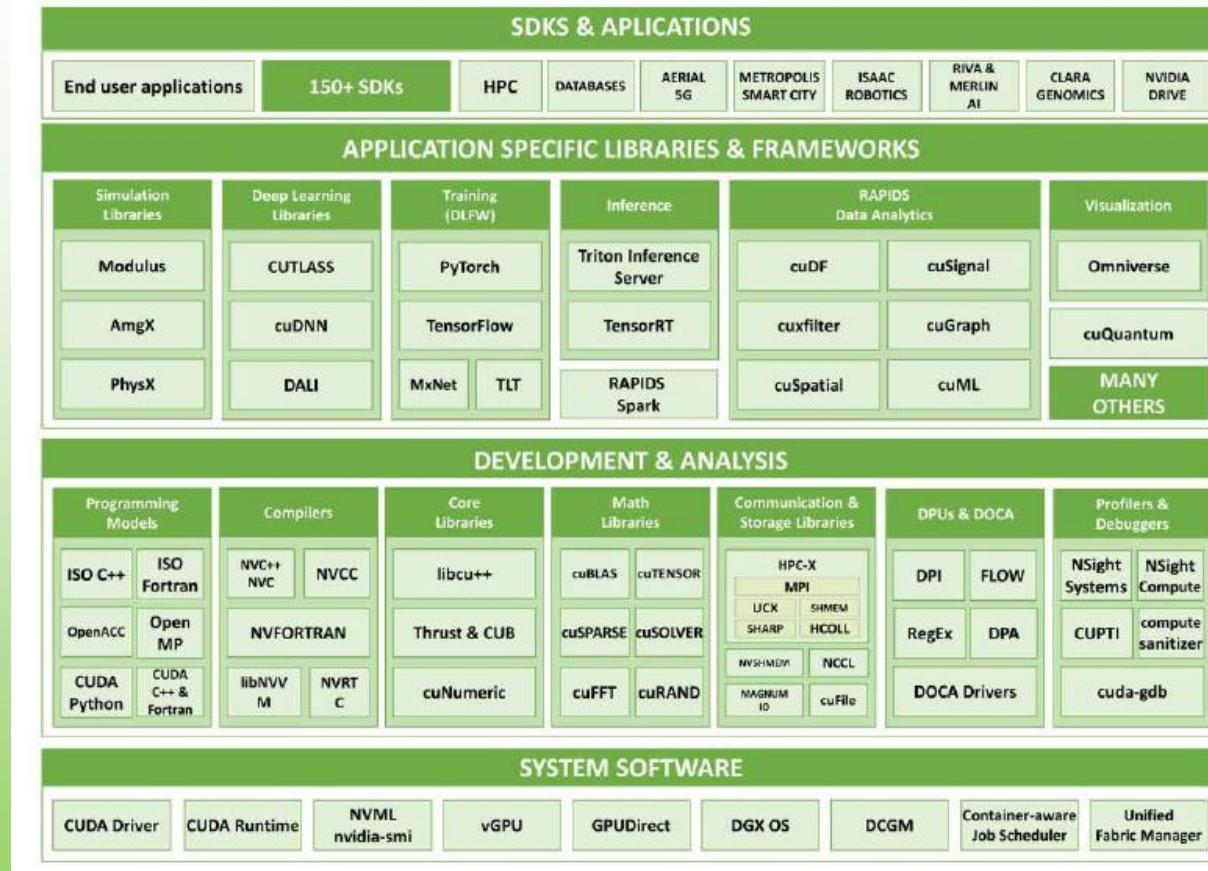
<https://developer.nvidia.com/tools-ecosystem>



Hartree Centre

Science & Technology Facilities Council

CUDA Development Ecosystem (III)



<https://developer.nvidia.com/tools-ecosystem>



The CUDA tools

- NSight (debugger)
- NVPP (performance)
- Code samples
- ...and many more!!!

Ex.: Deep Learning Frameworks



CuDNN (CUDA Deep Neural Network library)



CUDA



NVidia drivers



GPU Hardware

Questions?



TOPICS

CUDA threads

GPU-accelerated vs. CPU-only Applications

CUDA Kernel Execution

Parallel Memory Access

GPU occupancy

Kernel occupancy

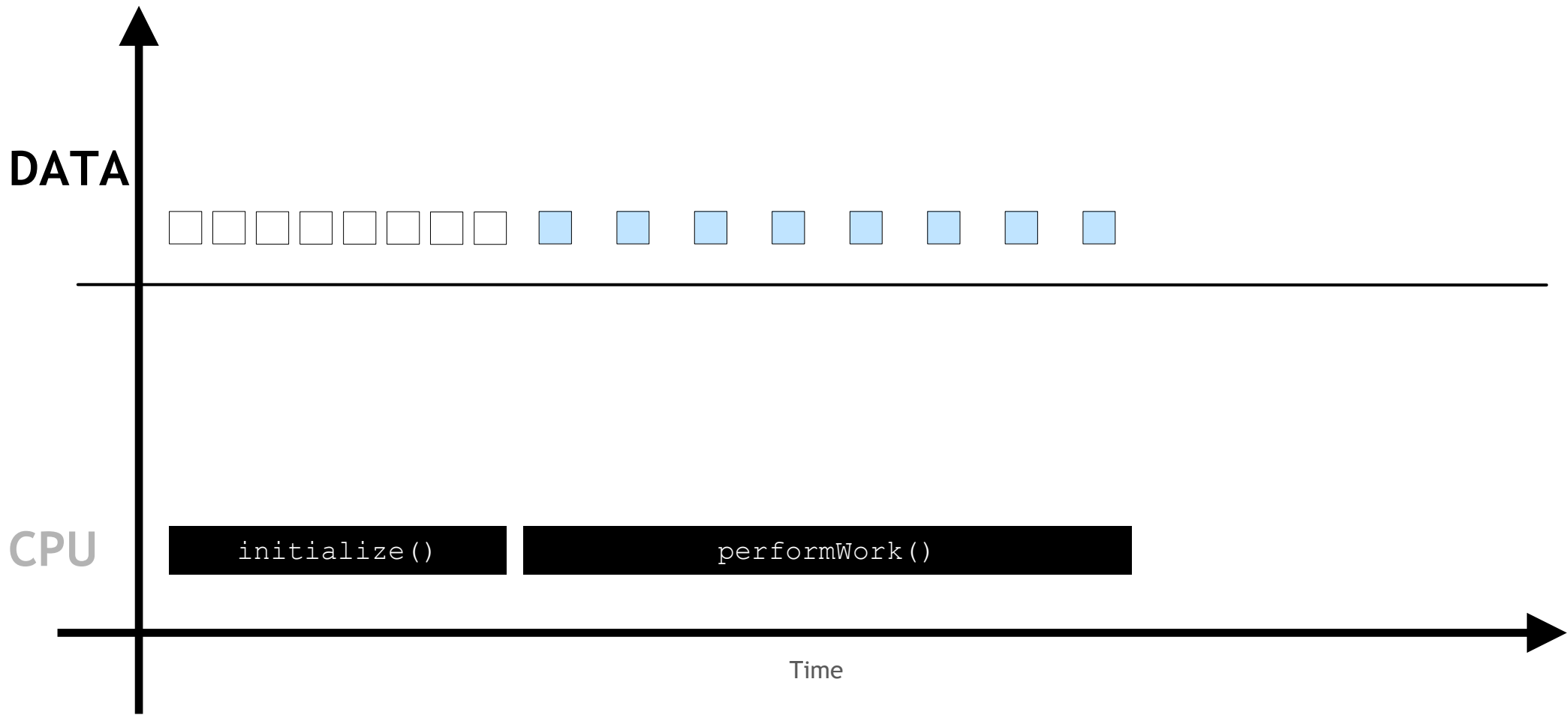
Appendix: Glossary

GPU-accelerated vs. CPU-only Applications

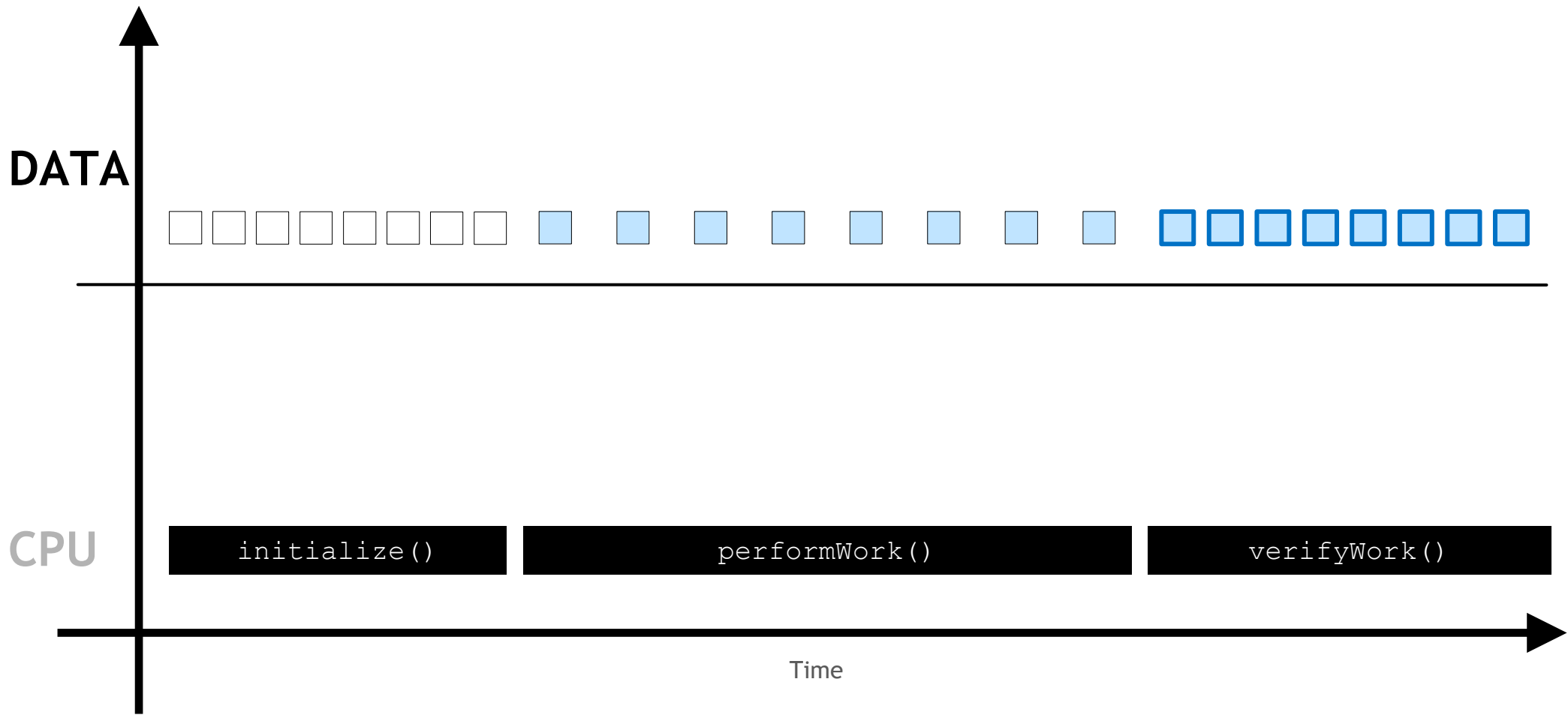
In **CPU-only applications** data is allocated on CPU



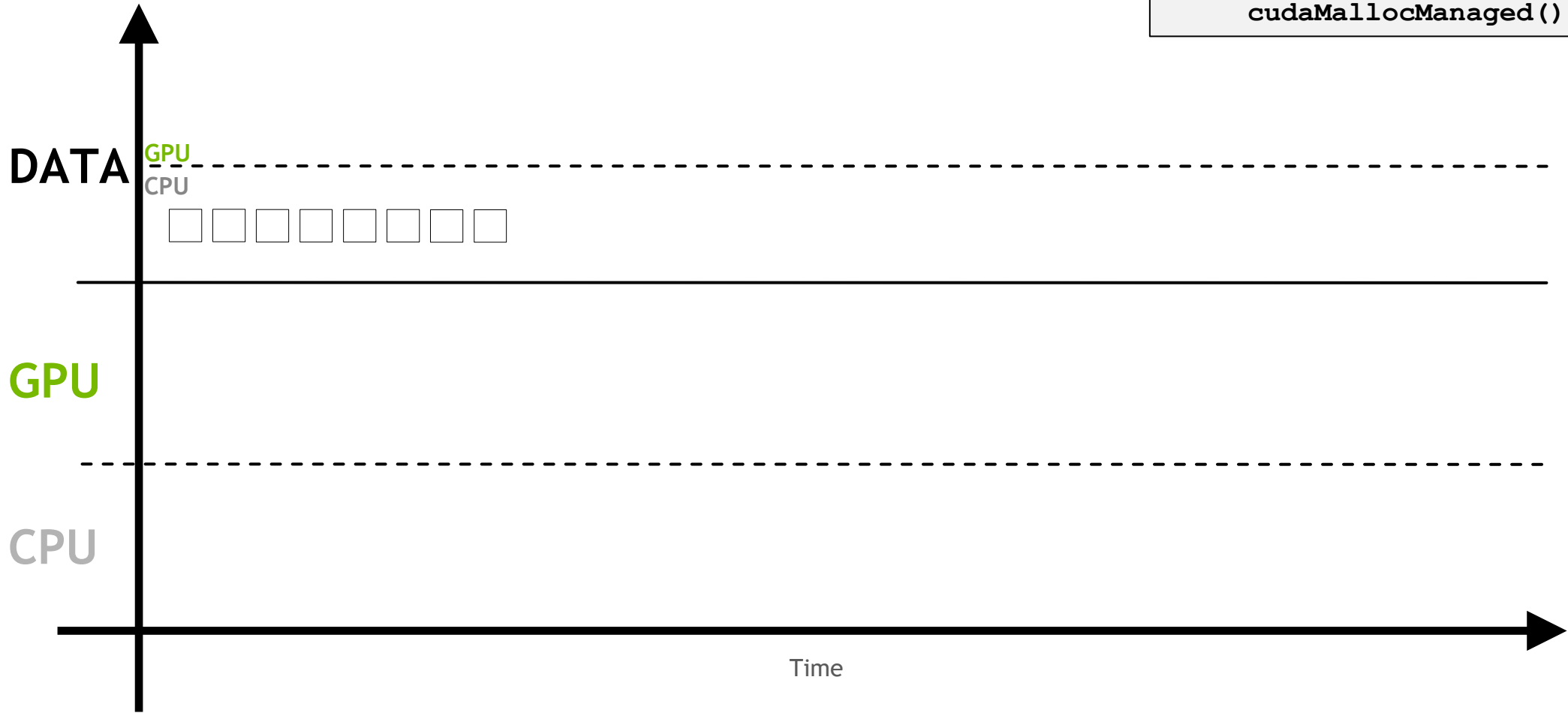
...and all work is performed on CPU



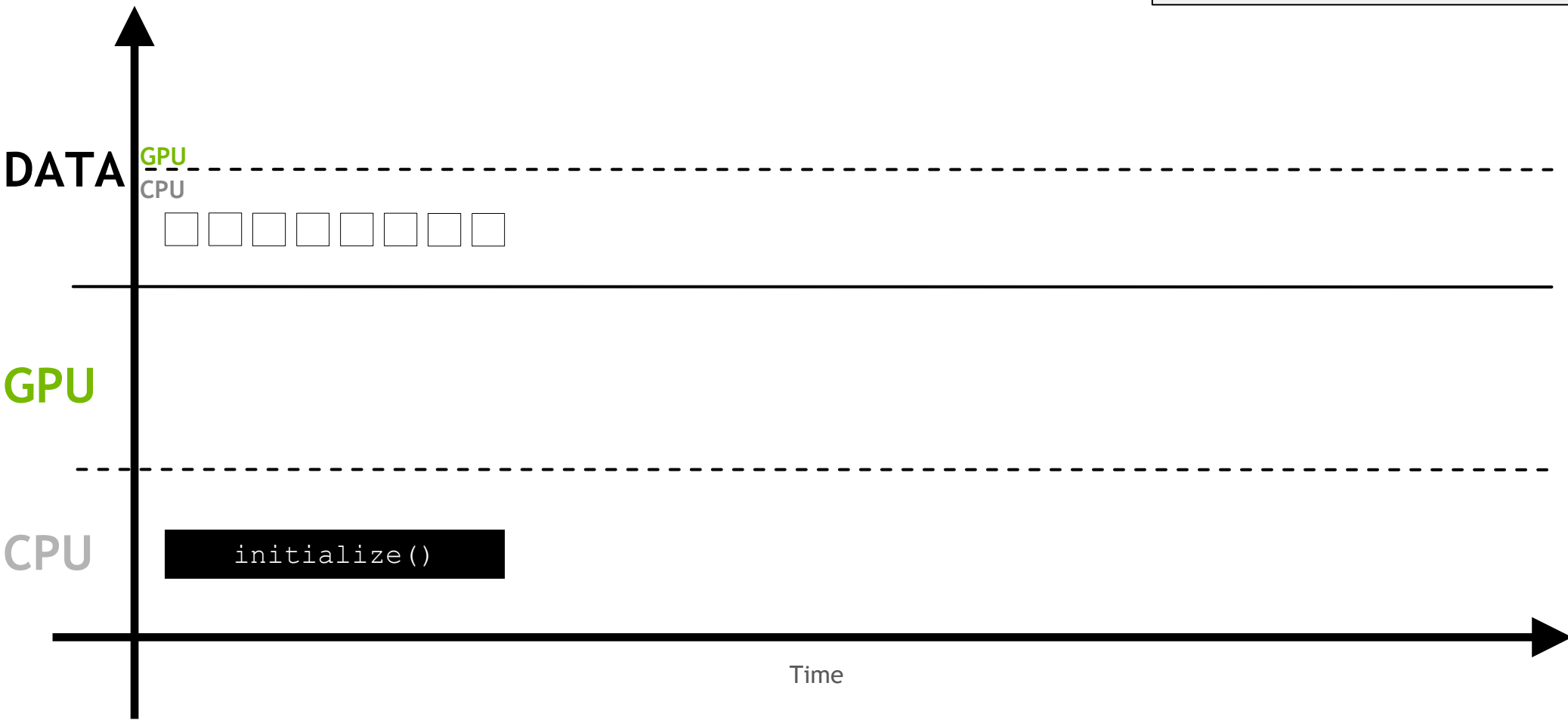
...and all work is performed on CPU



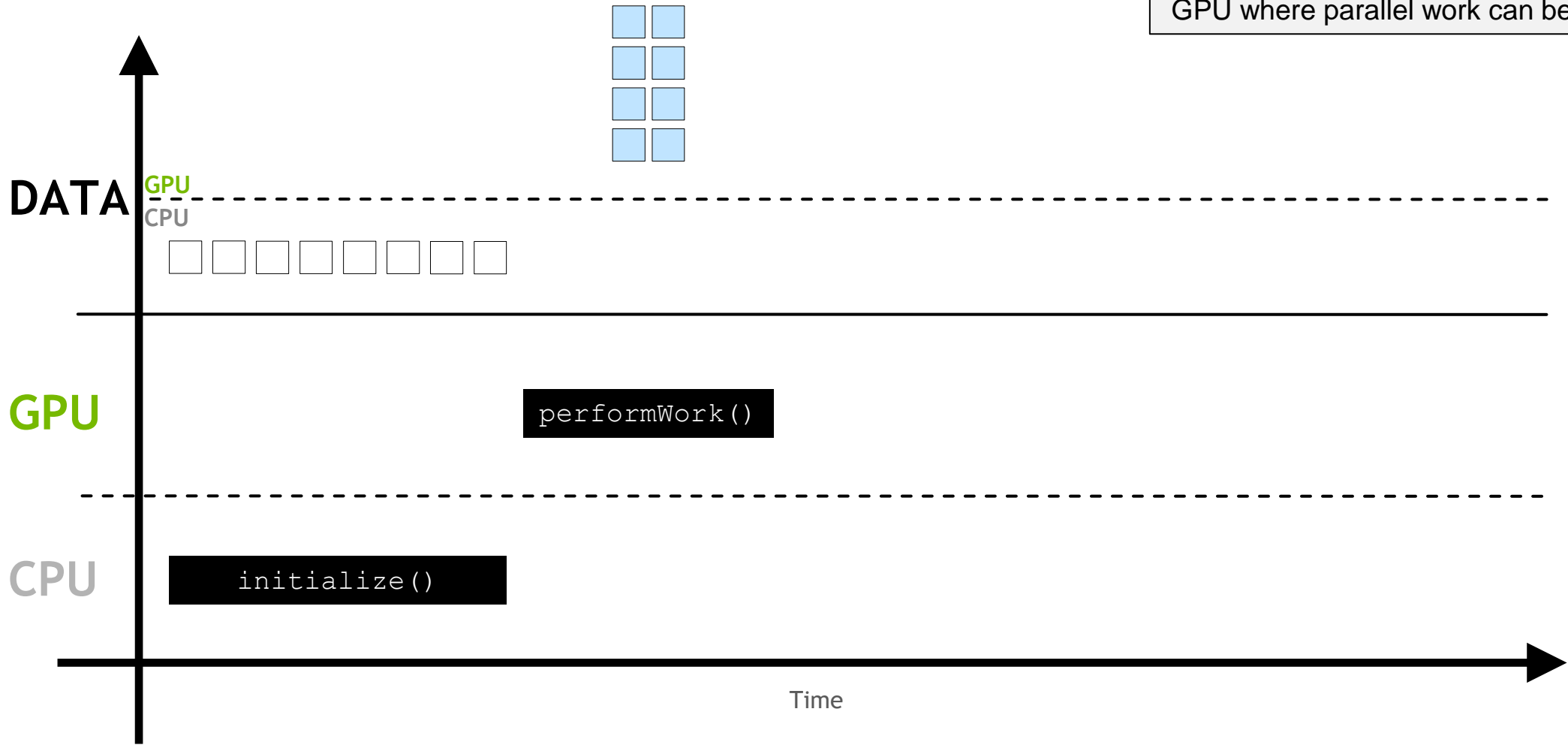
In **accelerated applications** data is allocated with `cudaMallocManaged()`



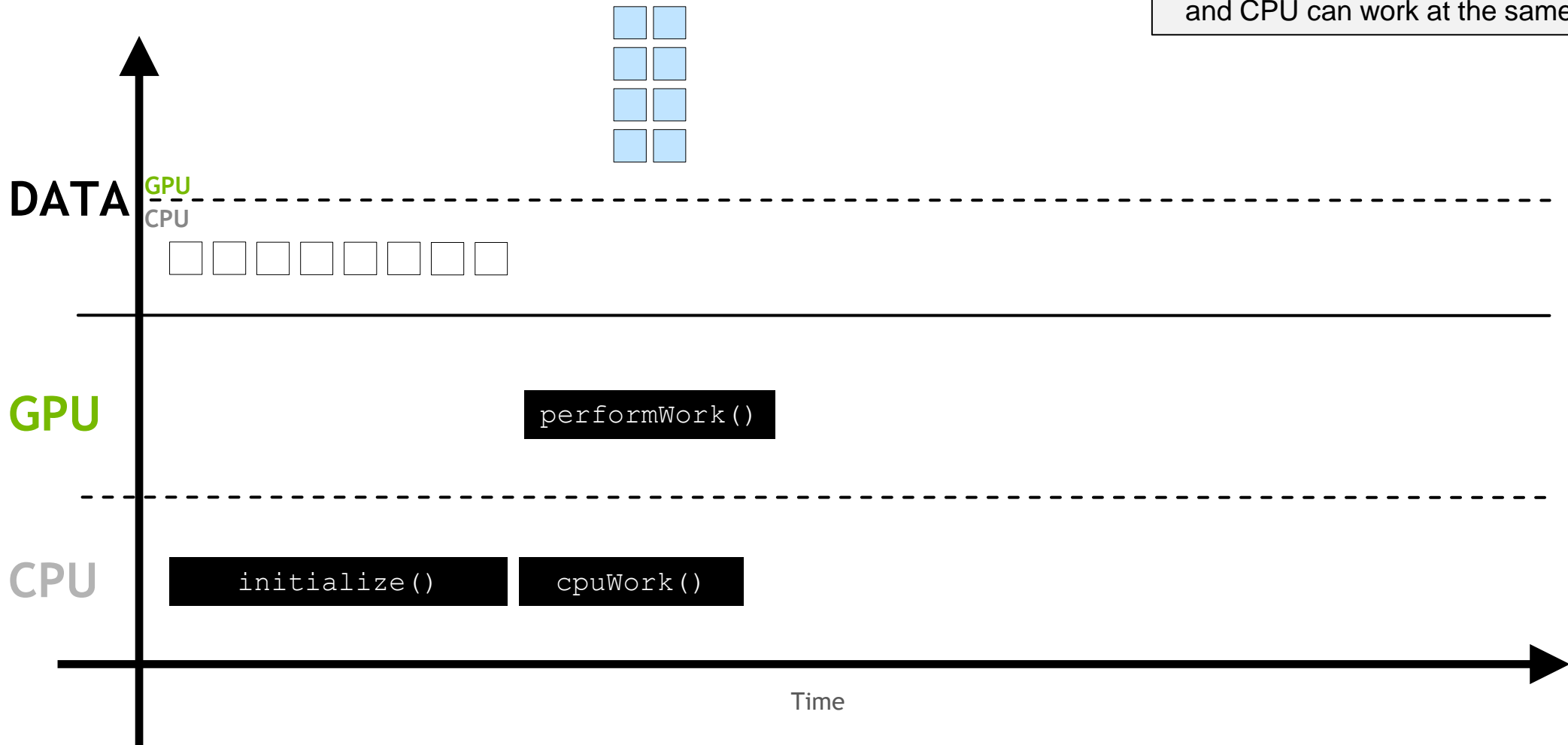
... where it can be accessed and worked on by the CPU

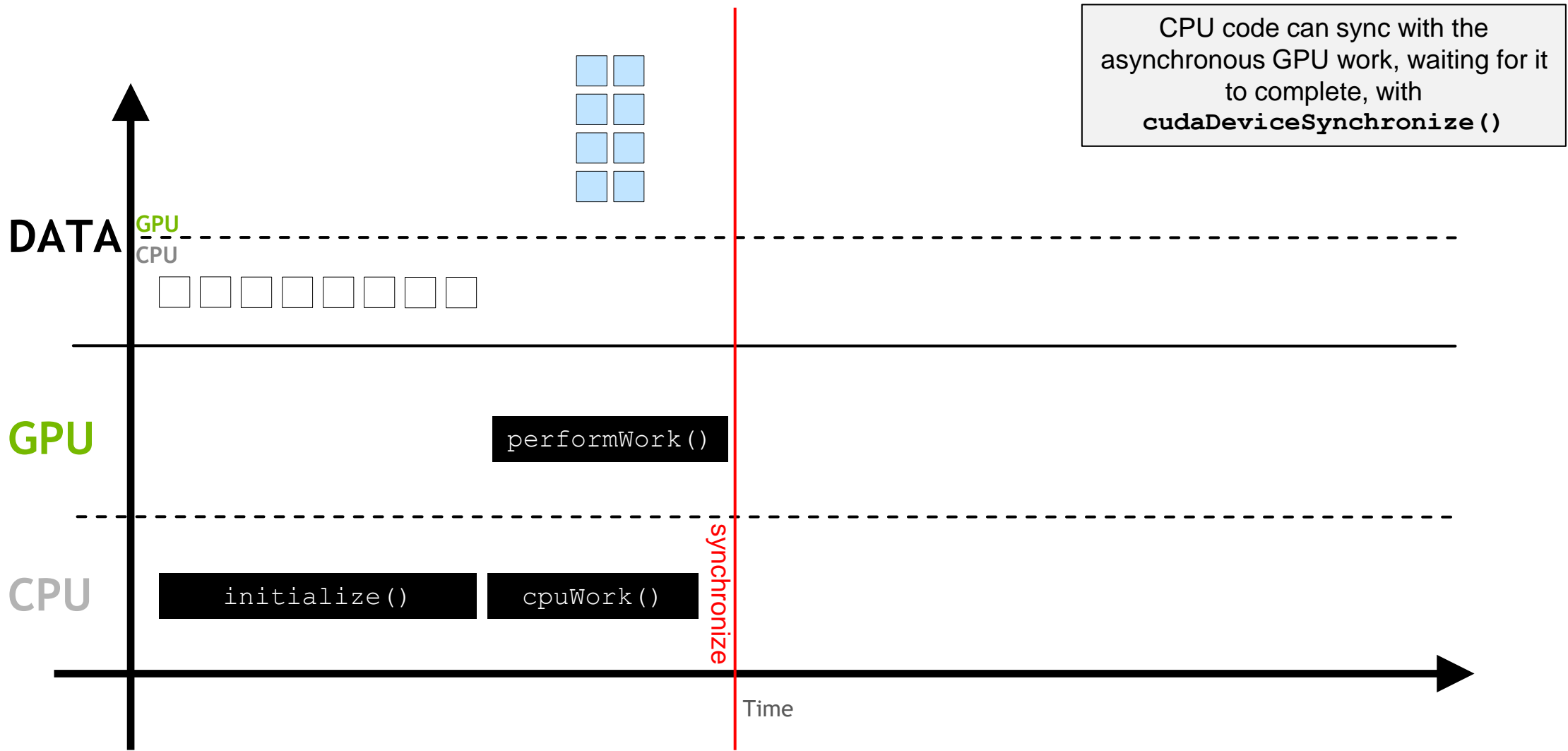


... and automatically migrated to the GPU where parallel work can be done

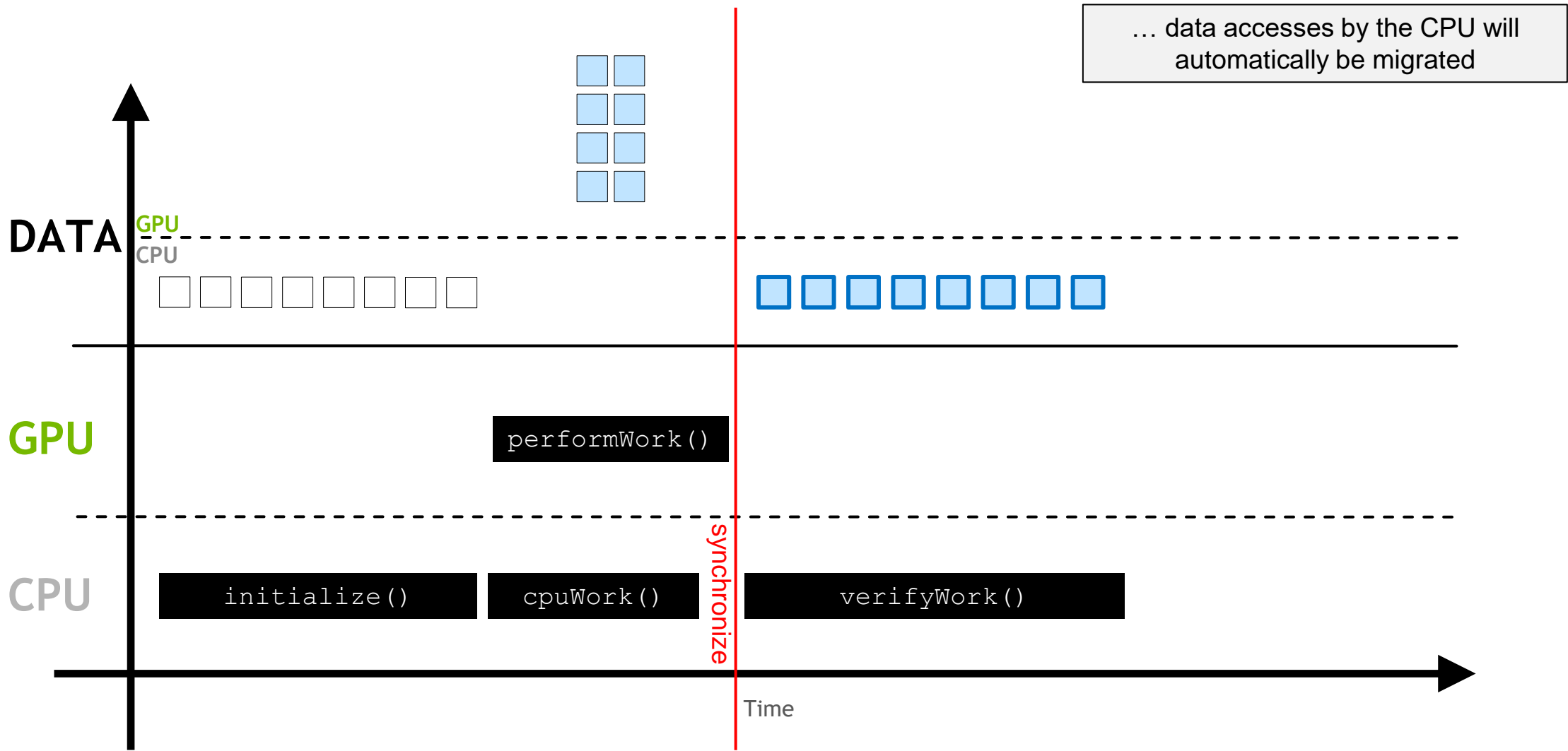


Work on the GPU is **asynchronous**,
and CPU can work at the same time

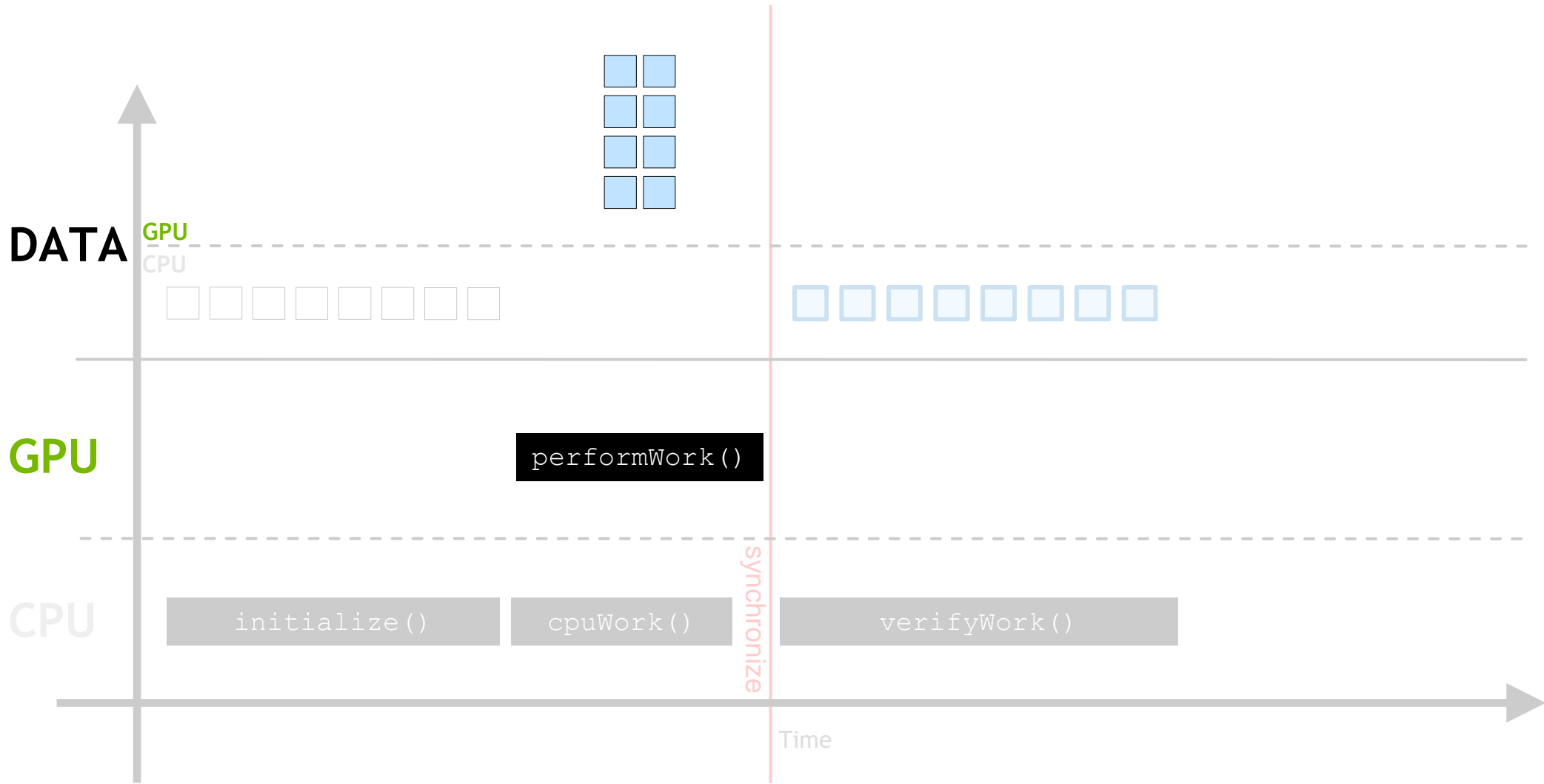




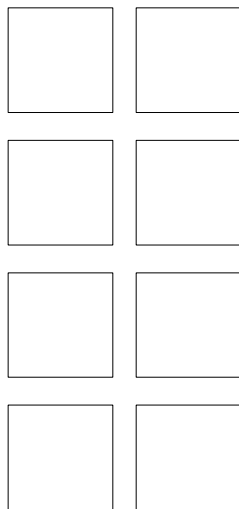
CPU code can sync with the asynchronous GPU work, waiting for it to complete, with `cudaDeviceSynchronize()`



Coordinating Parallel Threads

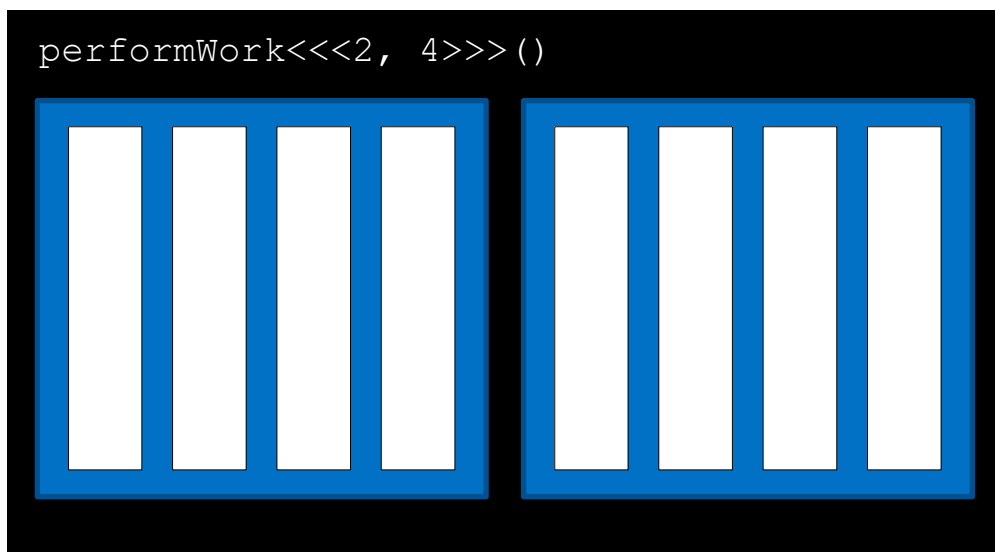


GPU
DATA



Assume data is in a 0 indexed vector

GPU



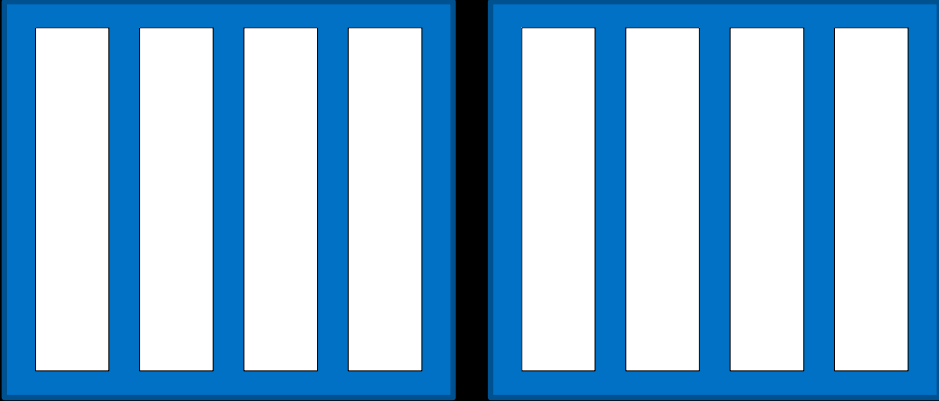
GPU DATA

0	4
1	5
2	6
3	7

Assume data is in a 0 indexed vector

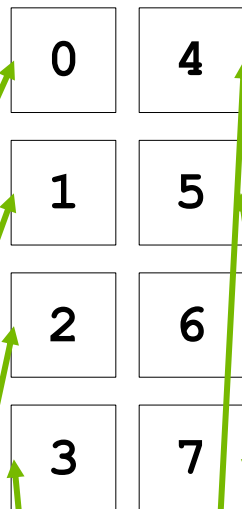
GPU

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



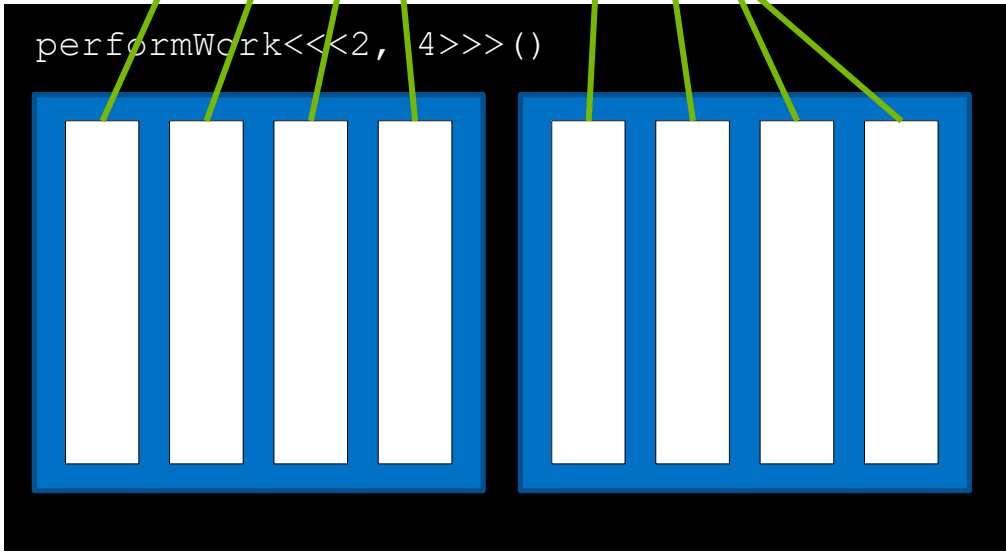
The diagram illustrates the GPU architecture. It features a black rectangular background. At the top, the code `performWork<<<2, 4>>>()` is written in white. Below the code, there are two identical groups of four vertical blue bars, each representing a GPU core. The bars are arranged in two rows of two, with a gap between the two groups.

GPU
DATA



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

GPU

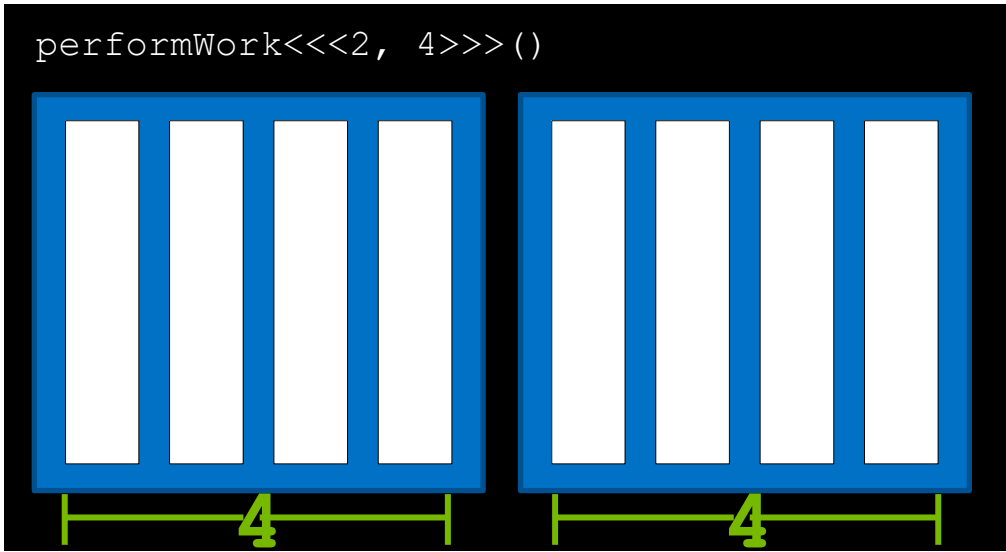


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

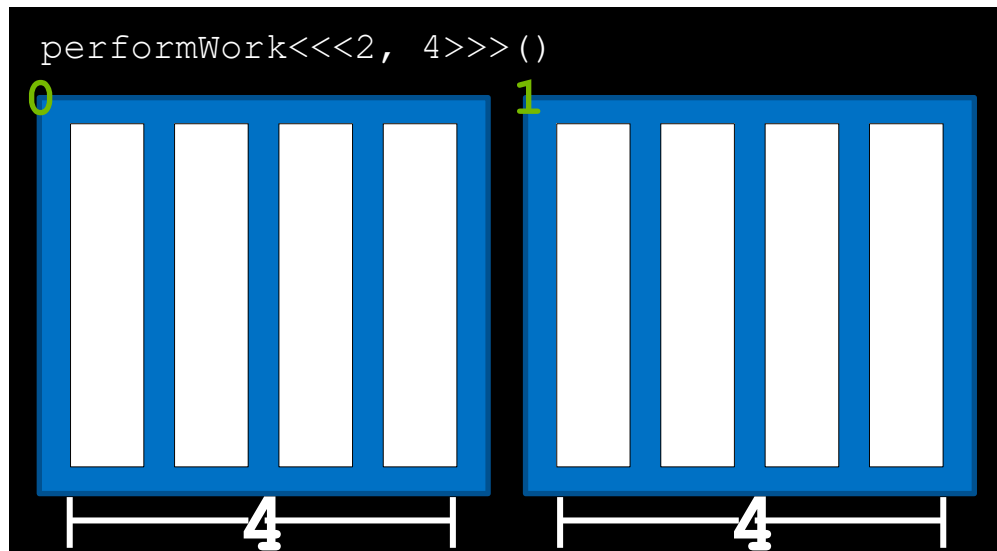


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

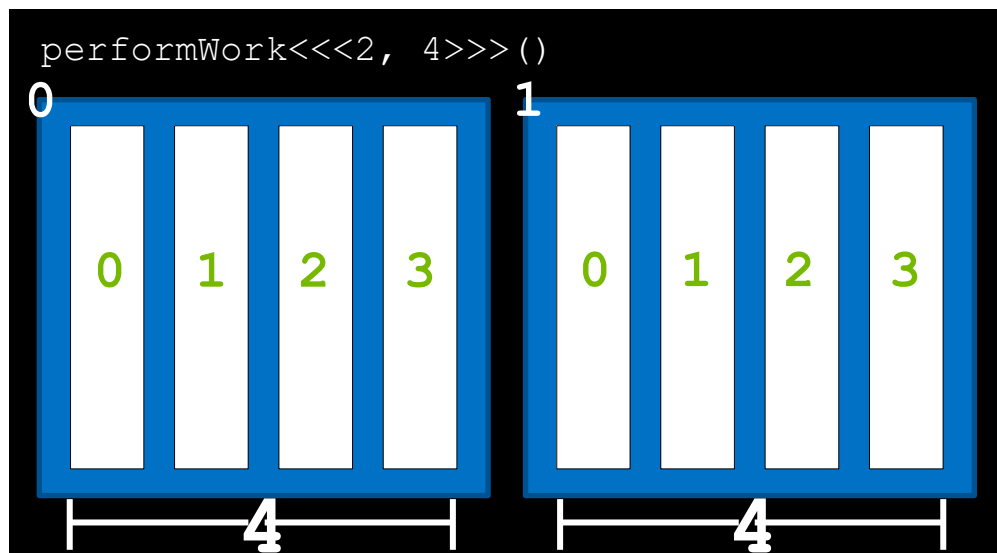


GPU DATA

0	4
1	5
2	6
3	7

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

GPU

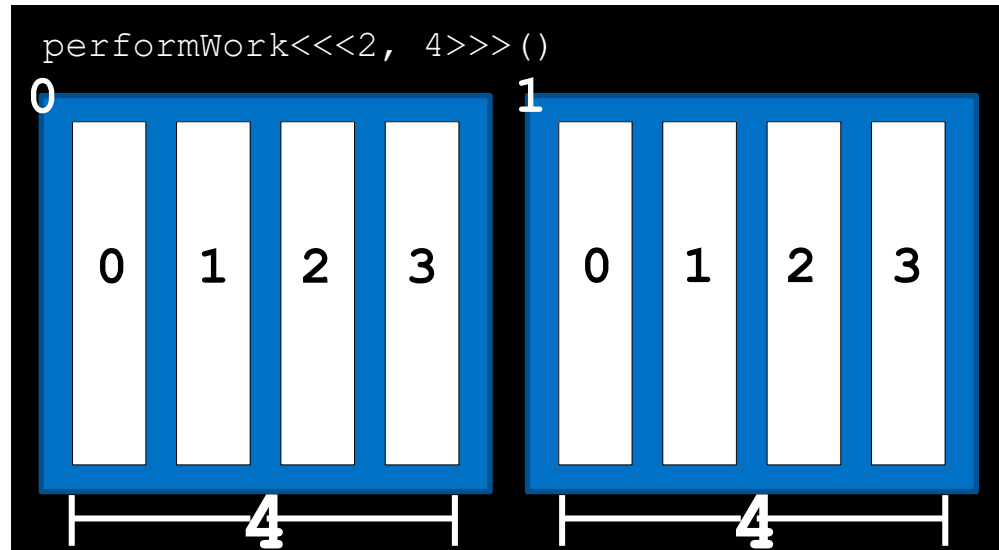


GPU DATA

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



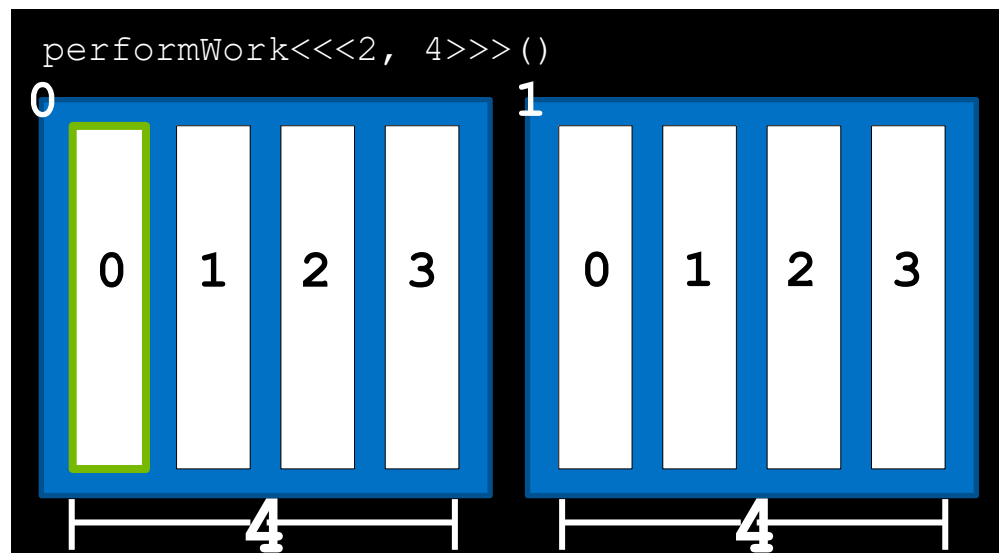
GPU DATA

0	4
1	5
2	6
3	7

<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
0		0		4

<code>dataIndex</code>
?

GPU



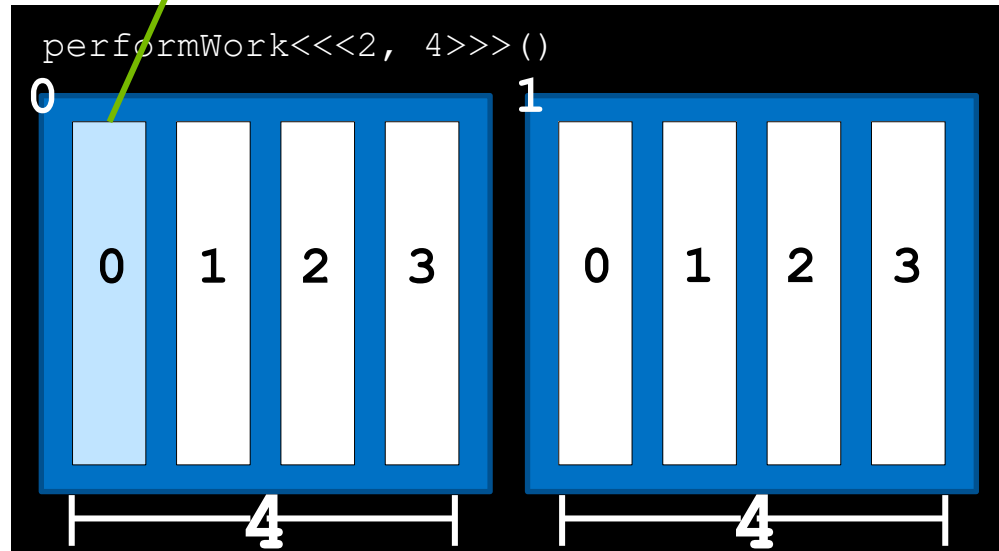
GPU
DATA

0	4
1	5
2	6
3	7

<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
0		0		4

<code>dataIndex</code>
0

GPU



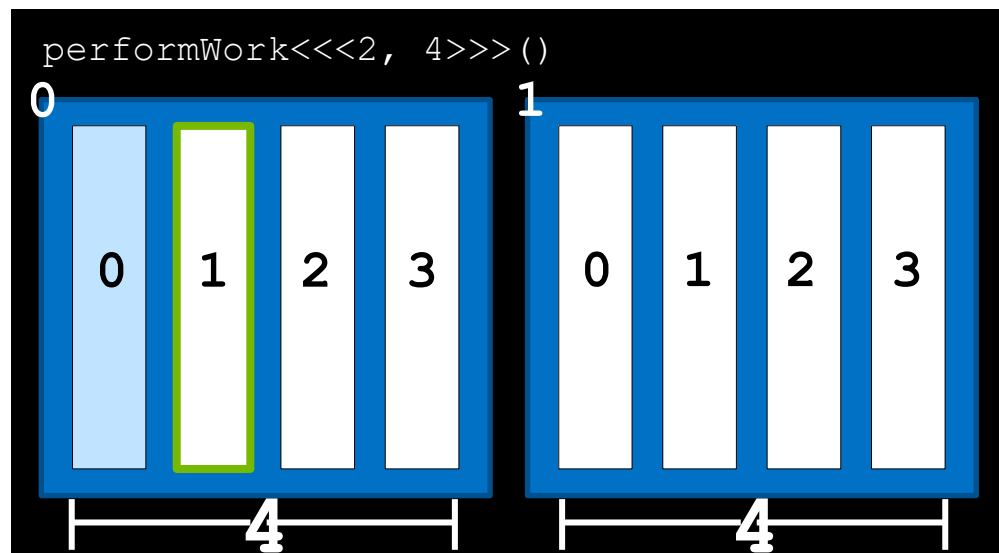
GPU DATA

0	4
1	5
2	6
3	7

<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
1		0		4

<code>dataIndex</code>
?

GPU



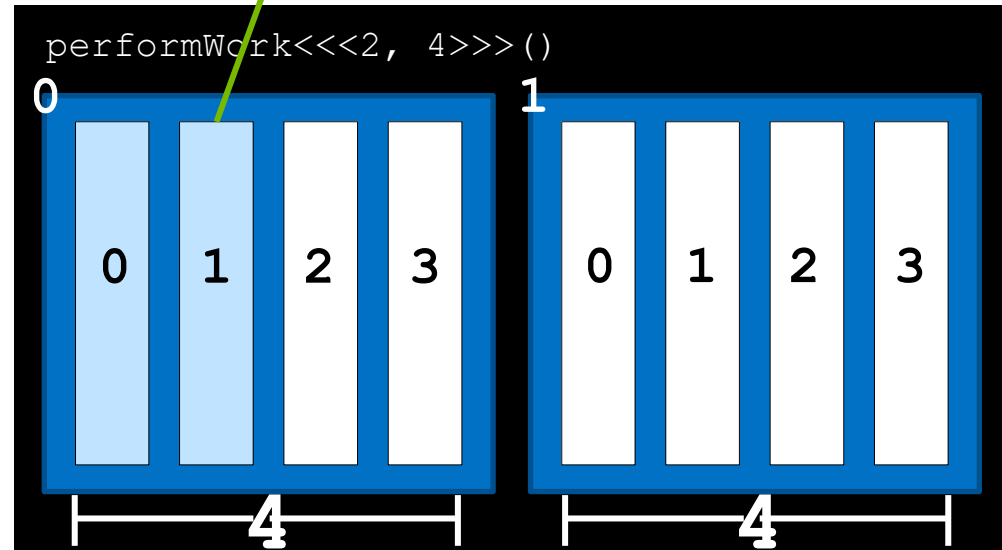
GPU
DATA

0	4
1	5
2	6
3	7

<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
1		0		4

<code>dataIndex</code>
1

GPU



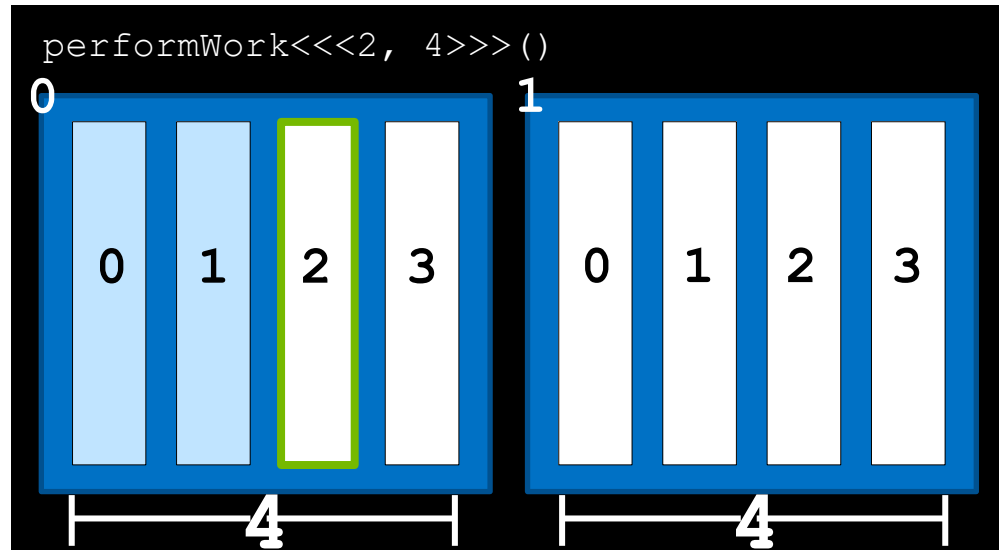
GPU
DATA

0	4
1	5
2	6
3	7

<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
2		0		4

<code>dataIndex</code>
?

GPU



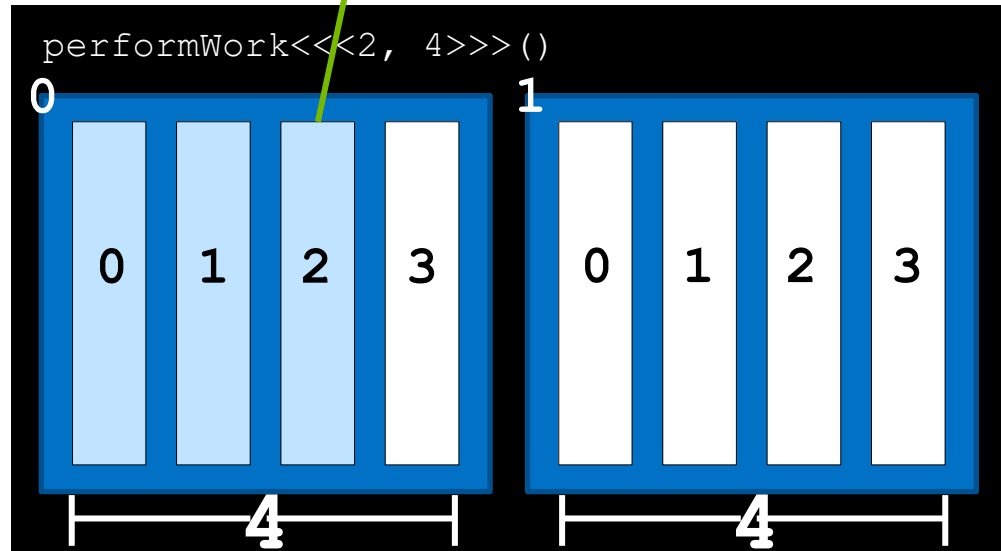
GPU
DATA

0	4
1	5
2	6
3	7

<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
2		0		4

<code>dataIndex</code>
2

GPU



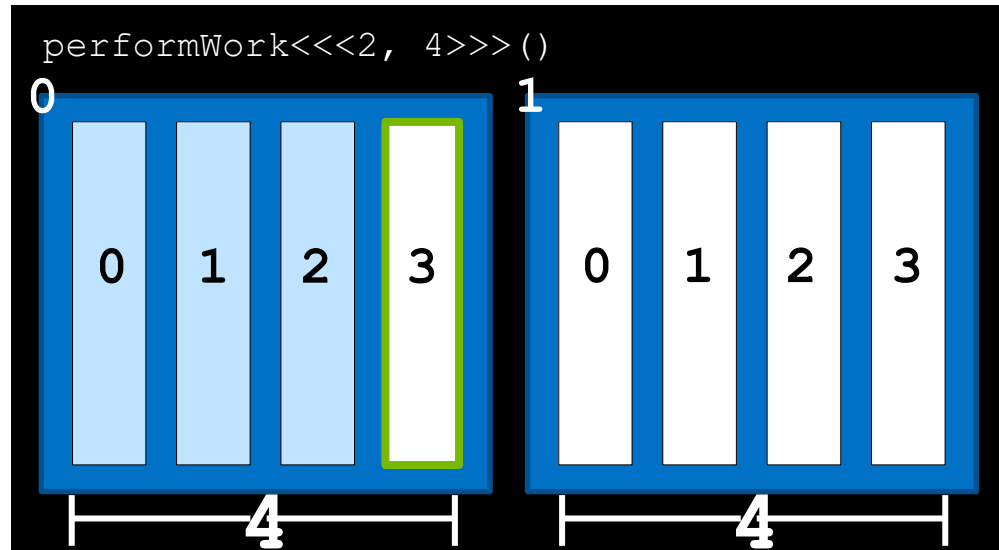
GPU
DATA

0	4
1	5
2	6
3	7

<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
3		0		4

<code>dataIndex</code>
?

GPU



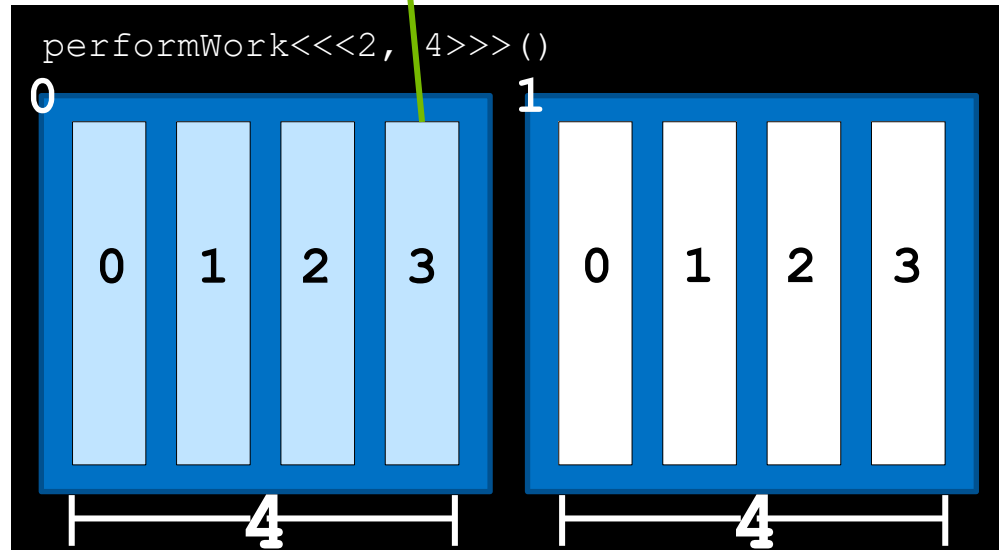
GPU
DATA

0	4
1	5
2	6
3	7

<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
3		0		4

<code>dataIndex</code>
3

GPU



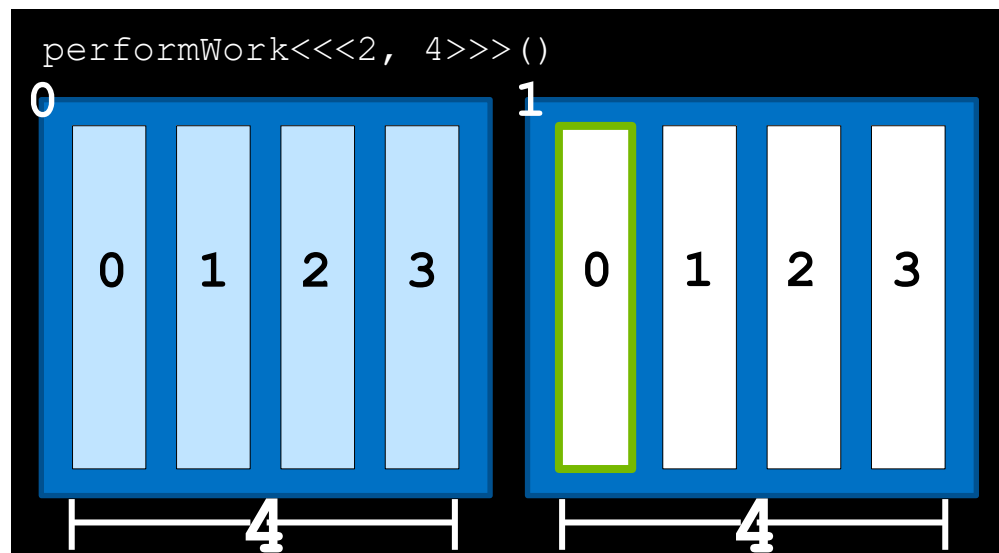
GPU DATA

0	4
1	5
2	6
3	7

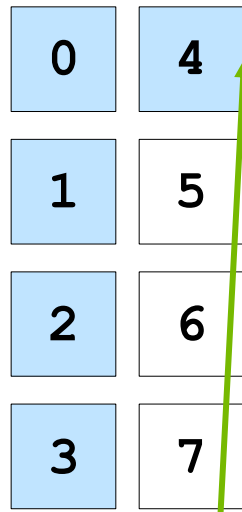
<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
0		1		4

<code>dataIndex</code>
?

GPU



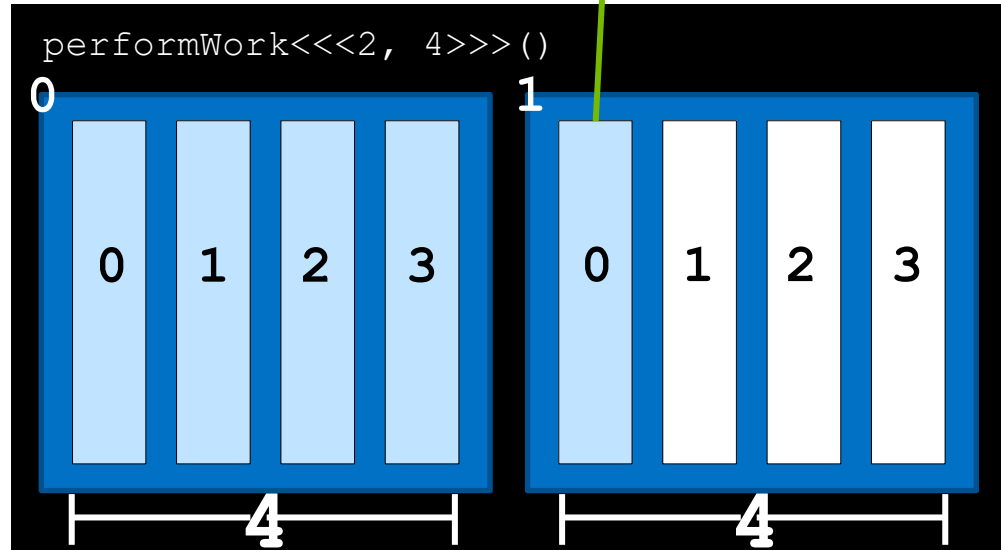
GPU
DATA



<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
0		1		4

<code>dataIndex</code>
4

GPU



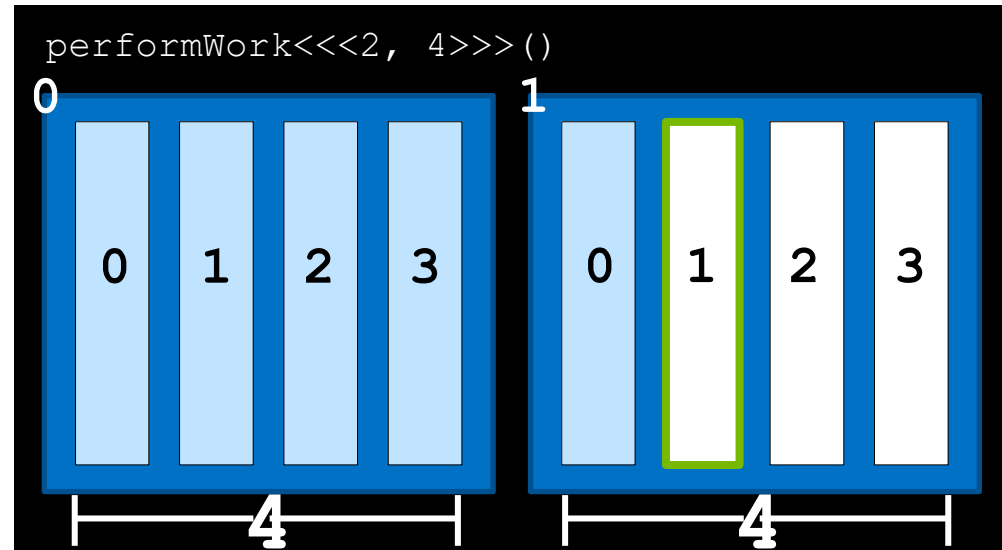
GPU
DATA

0	4
1	5
2	6
3	7

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

<code>dataIndex</code>
?

GPU



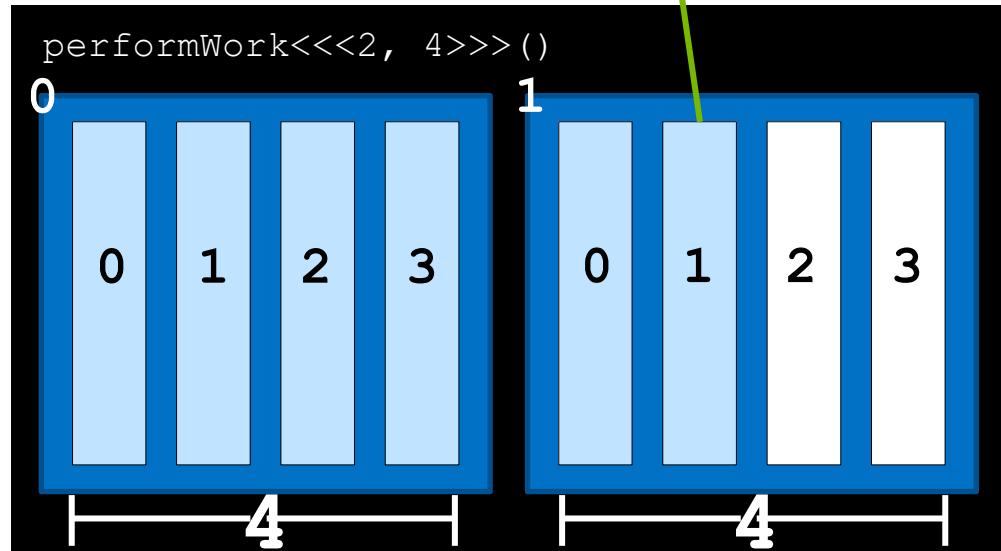
GPU
DATA

0	4
1	5
2	6
3	7

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

<code>dataIndex</code>
5

GPU



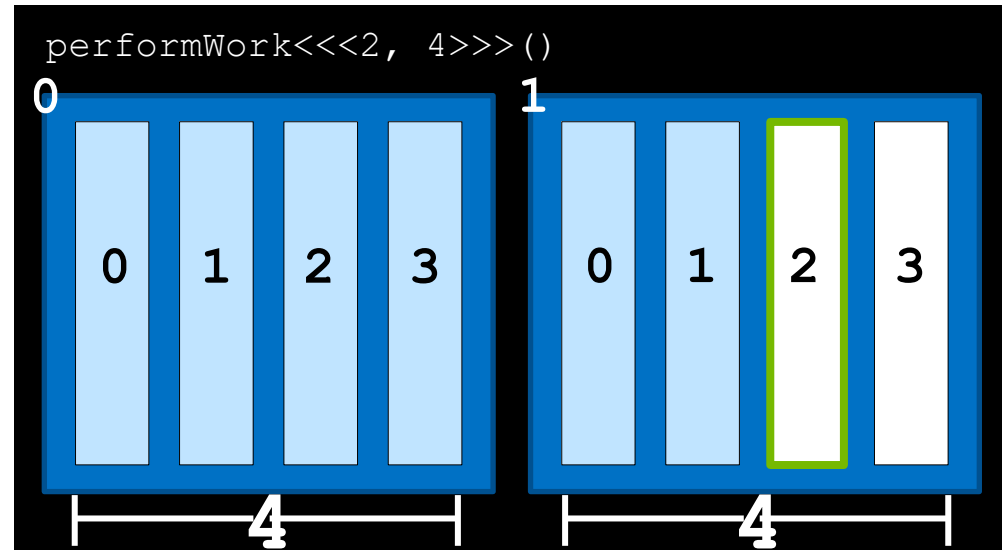
GPU DATA

0	4
1	5
2	6
3	7

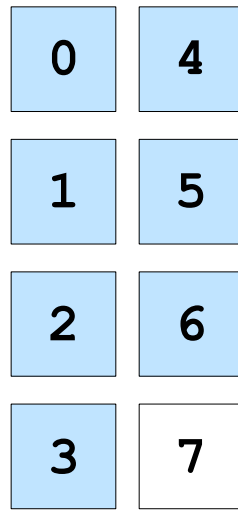
<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
2		1		4

<code>dataIndex</code>
?

GPU



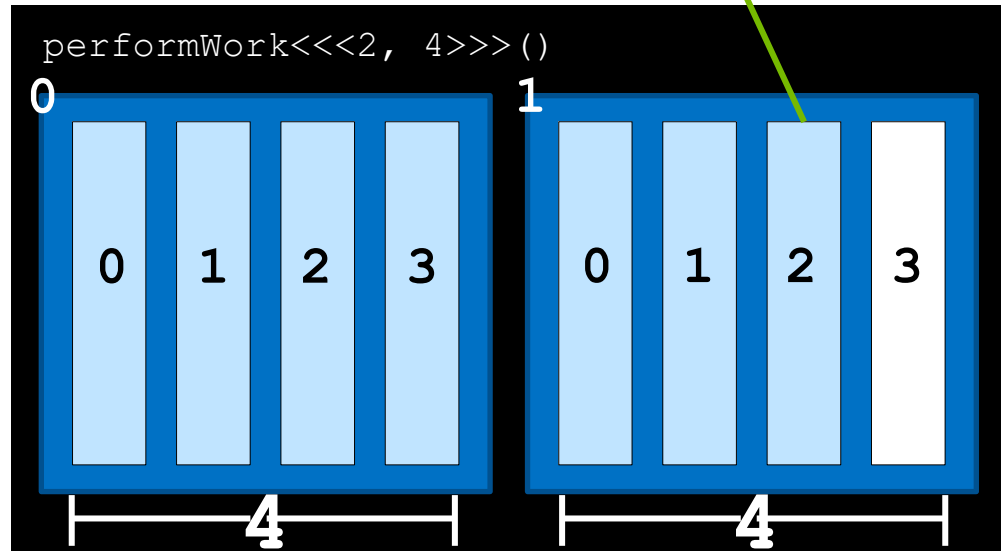
GPU
DATA



<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
2		1		4

<code>dataIndex</code>
6

GPU



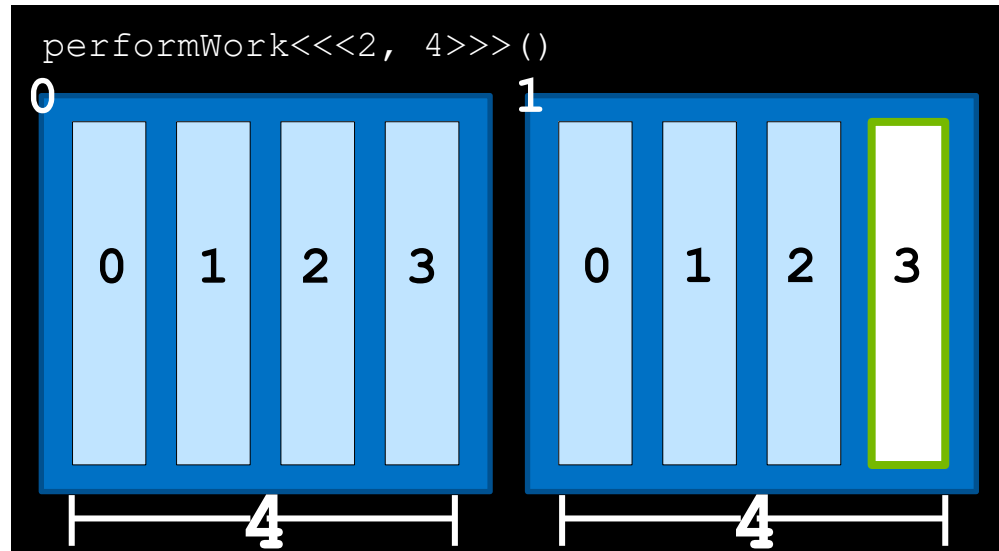
GPU
DATA

0	4
1	5
2	6
3	7

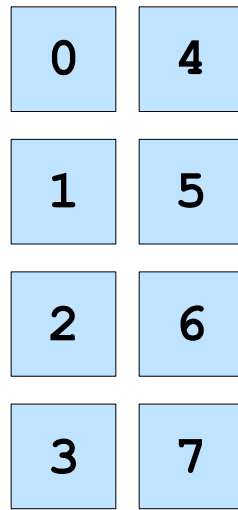
<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
3		1		4

<code>dataIndex</code>
?

GPU

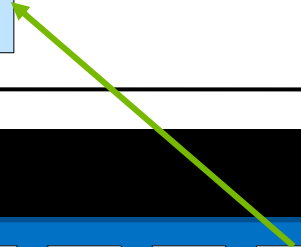


GPU
DATA

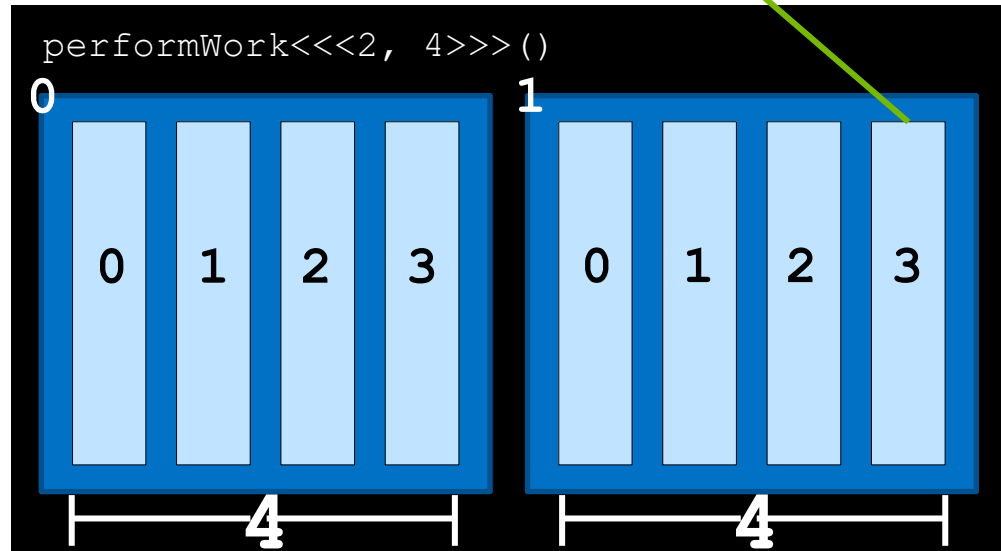


<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
3		1		4

<code>dataIndex</code>
7

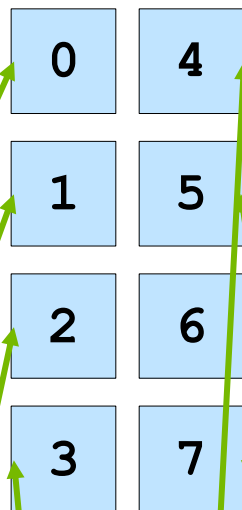


GPU



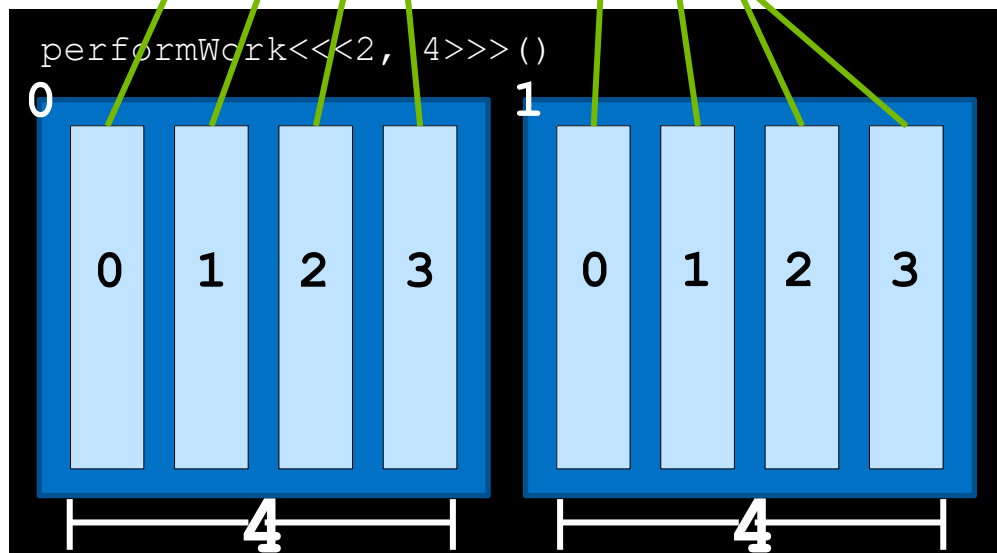
Grid Size Work Amount Mismatch

GPU
DATA

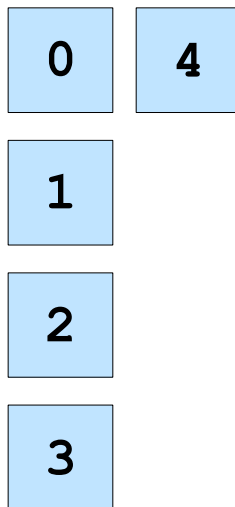


In previous scenarios, the number of threads in the grid matched the number of elements exactly

GPU

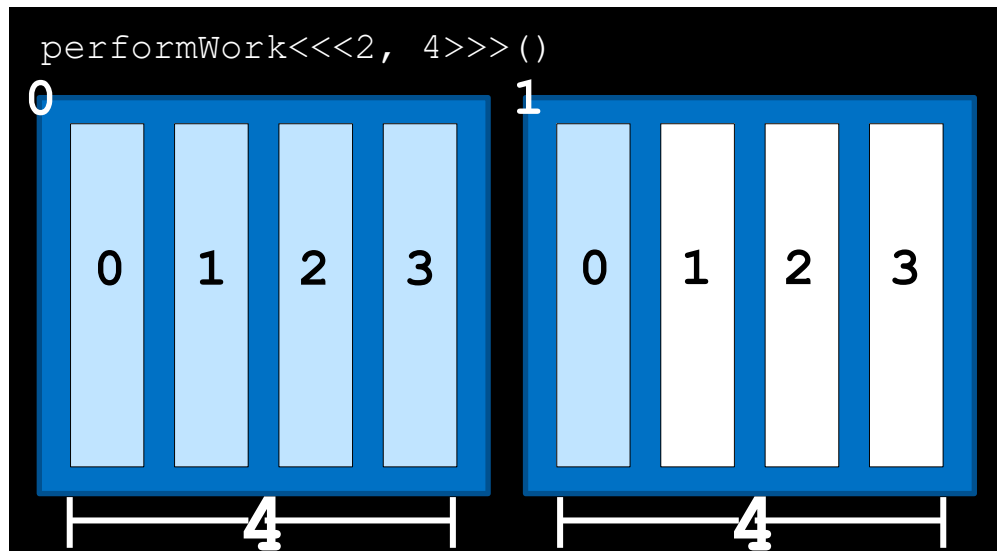


GPU
DATA

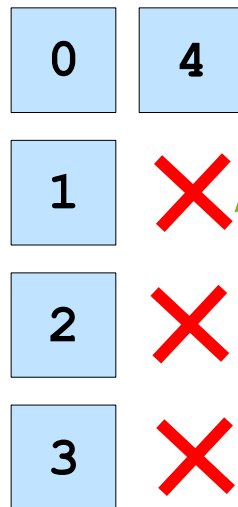


What if there are more threads than work to be done?

GPU

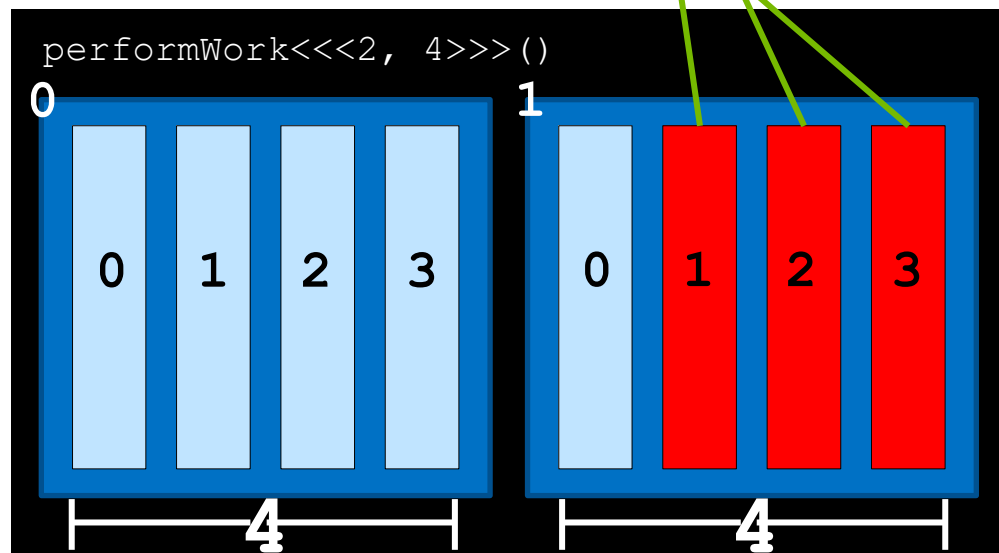


GPU DATA

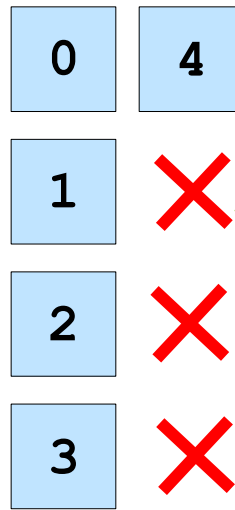


Attempting to access non-existent elements can result in a runtime error

GPU

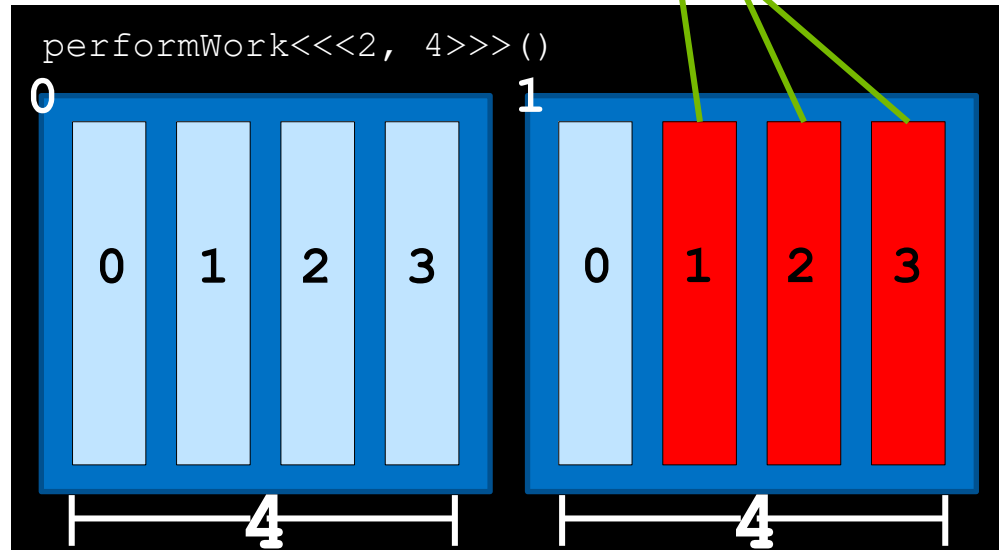


GPU DATA

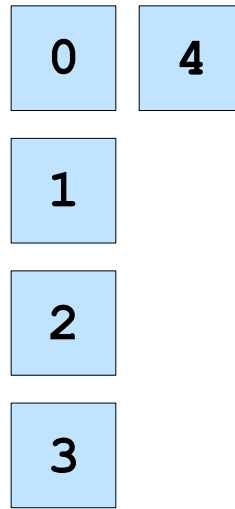


Code must check that the `dataIdx` calculated by `threadIdx.x + blockIdx.x * blockDim.x` is less than `N`, the number of data elements.

GPU



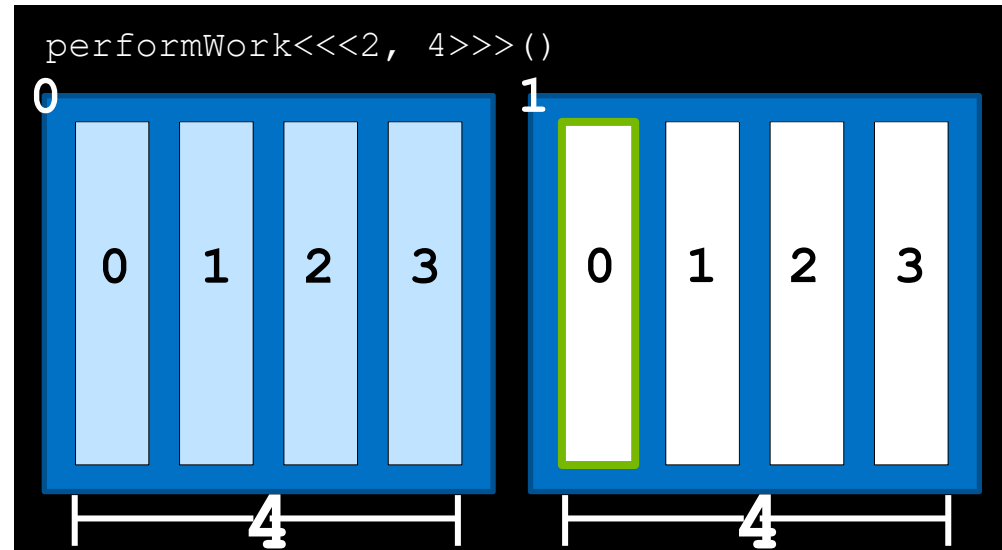
GPU DATA



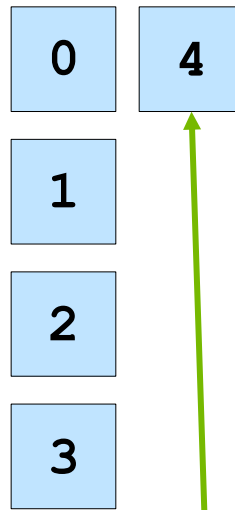
<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
0		1		4

<code>dataIndex</code>	<	N	=	Can work
4		5		?

GPU



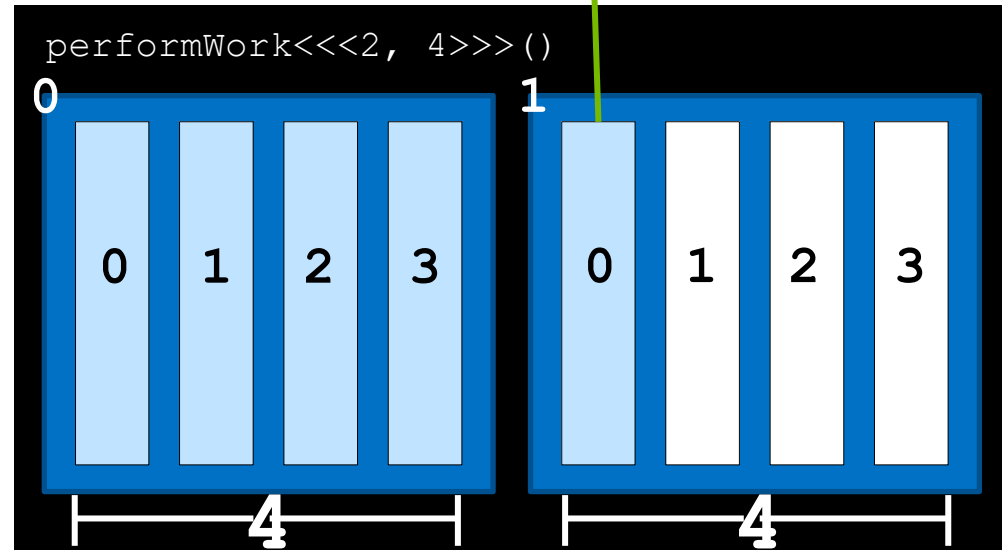
GPU DATA



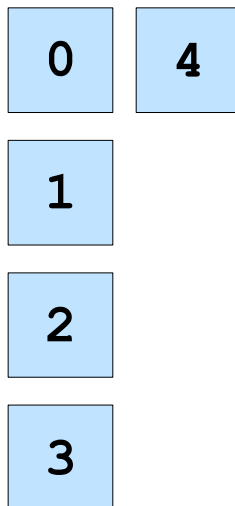
<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
0		1		4

<code>dataIndex</code>	<	N	=	Can work
4		5		true

GPU



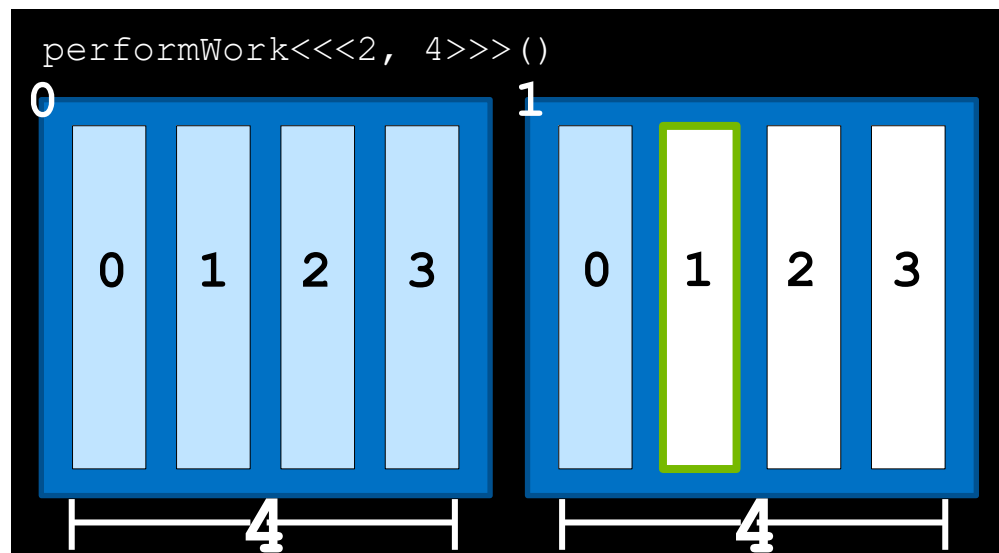
GPU DATA



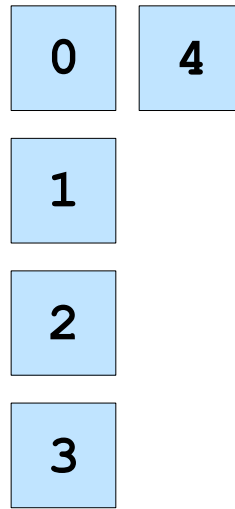
<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
1		1		4

<code>dataIndex</code>	<	N	=	Can work
5		5		?

GPU



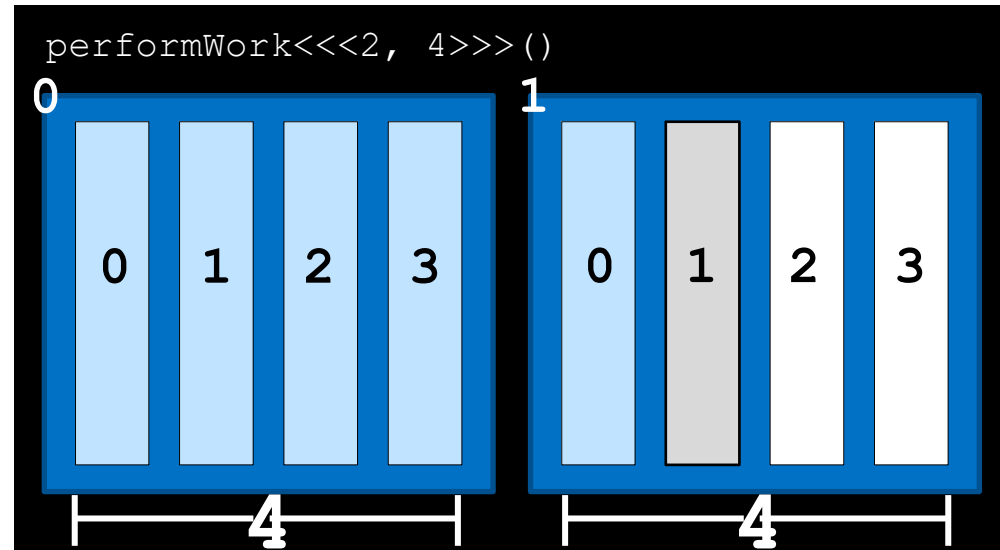
GPU DATA



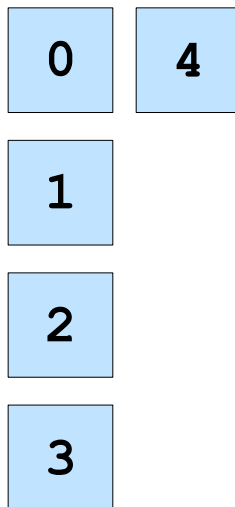
<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
1		1		4

<code>dataIndex</code>	<	N	=	Can work
5		5		false

GPU



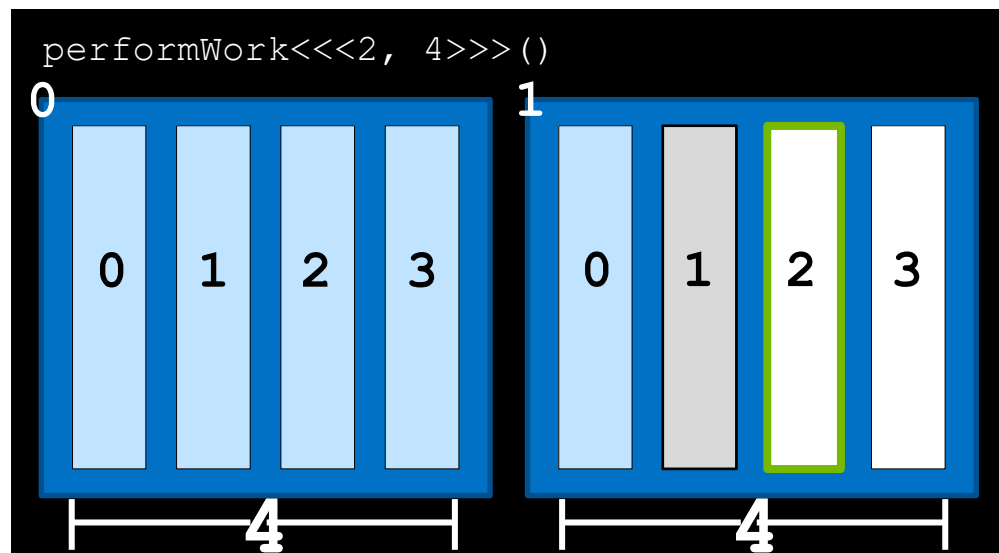
GPU DATA



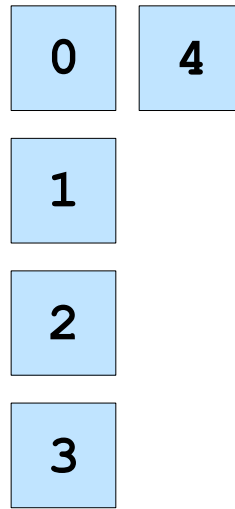
<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
2		1		4

<code>dataIndex</code>	<	N	=	Can work
6		5		?

GPU



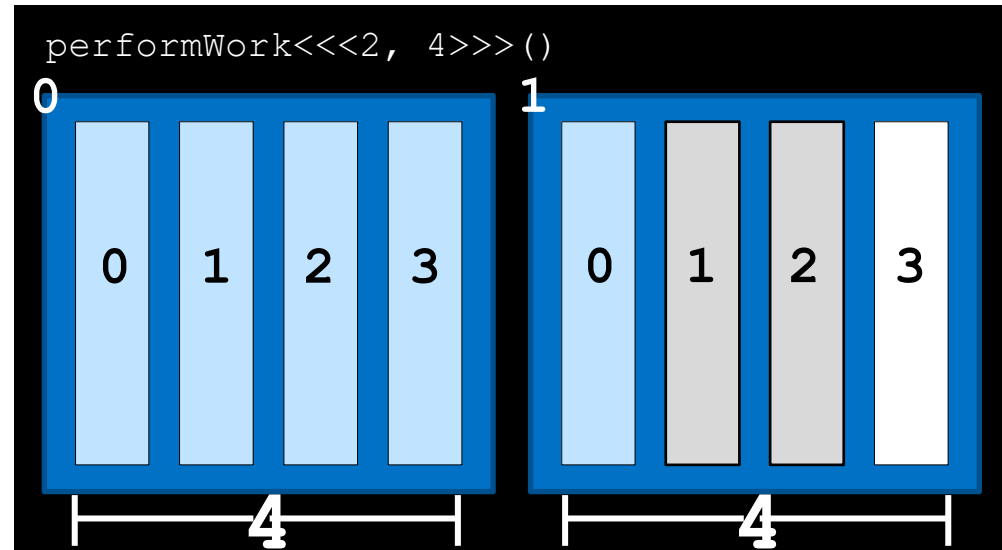
GPU
DATA



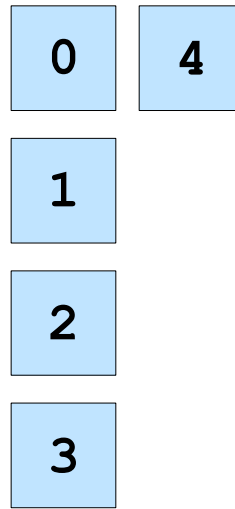
<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
2		1		4

<code>dataIndex</code>	<	N	=	Can work
6		5		false

GPU



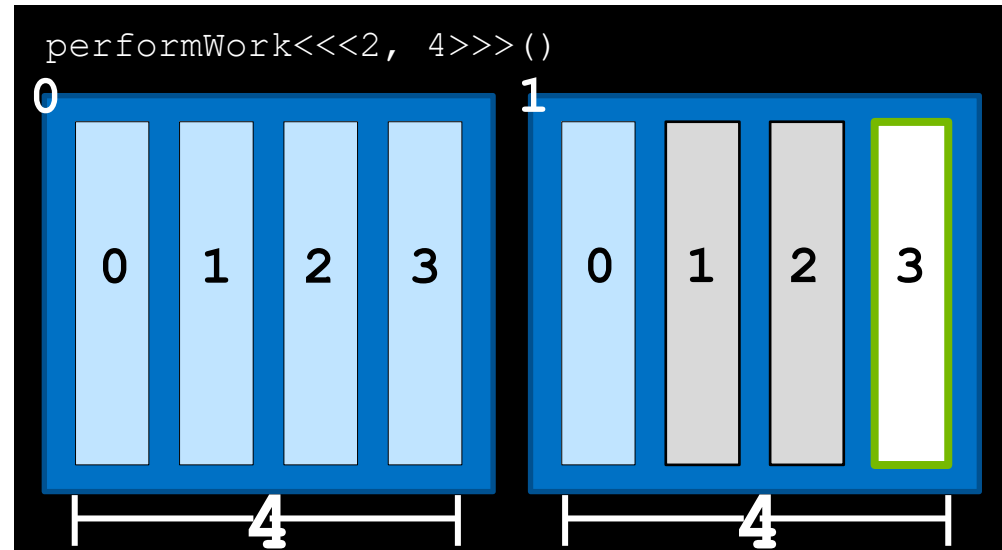
GPU DATA



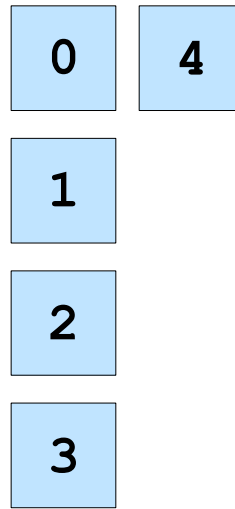
<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
2		1		4

<code>dataIndex</code>	<	N	=	Can work
6		5		?

GPU



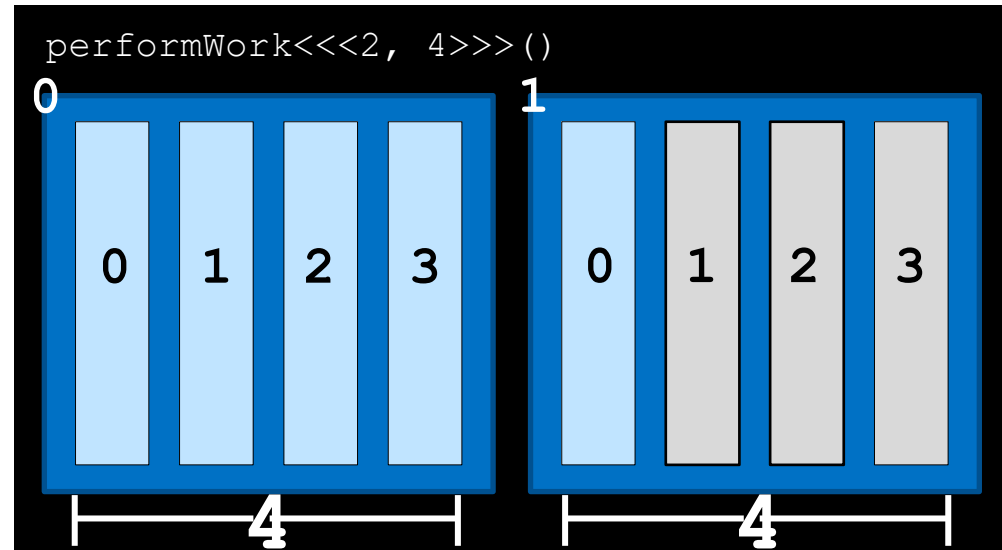
GPU
DATA



<code>threadIdx.x</code>	+	<code>blockIdx.x</code>	*	<code>blockDim.x</code>
2		1		4

<code>dataIndex</code>	<	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

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

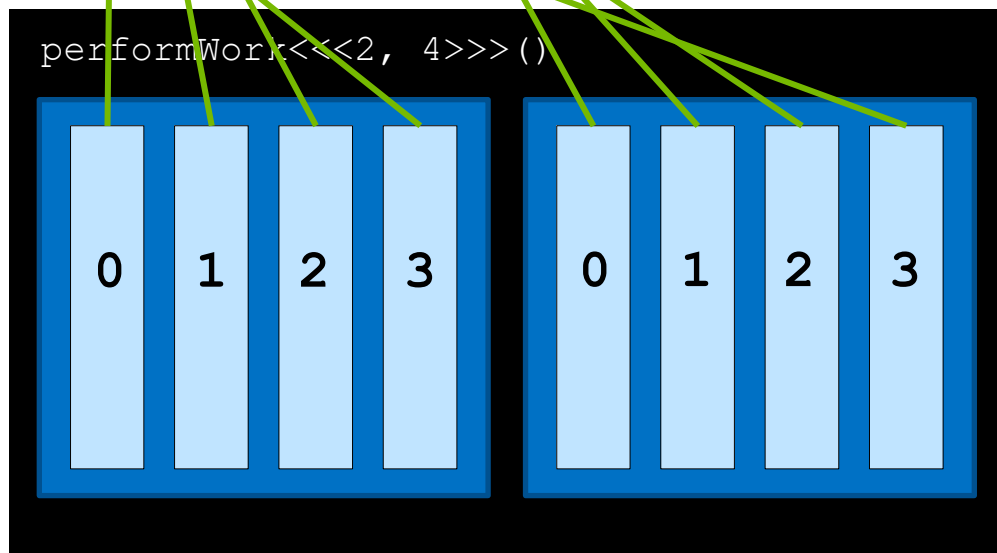
The diagram shows two thread blocks, each containing four threads. The threads are numbered 0, 1, 2, and 3. This represents a 2x4 thread grid configuration.

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

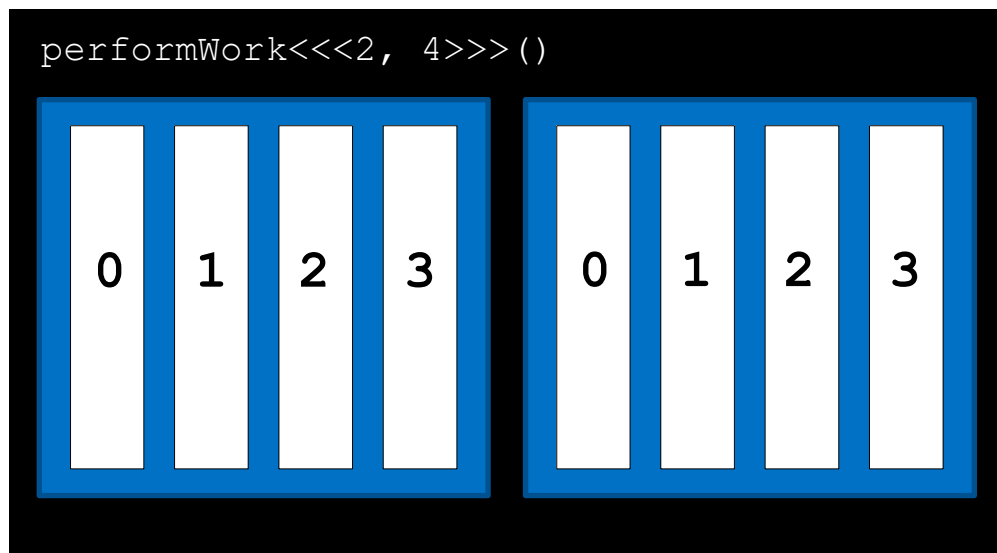


GPU
DATA

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

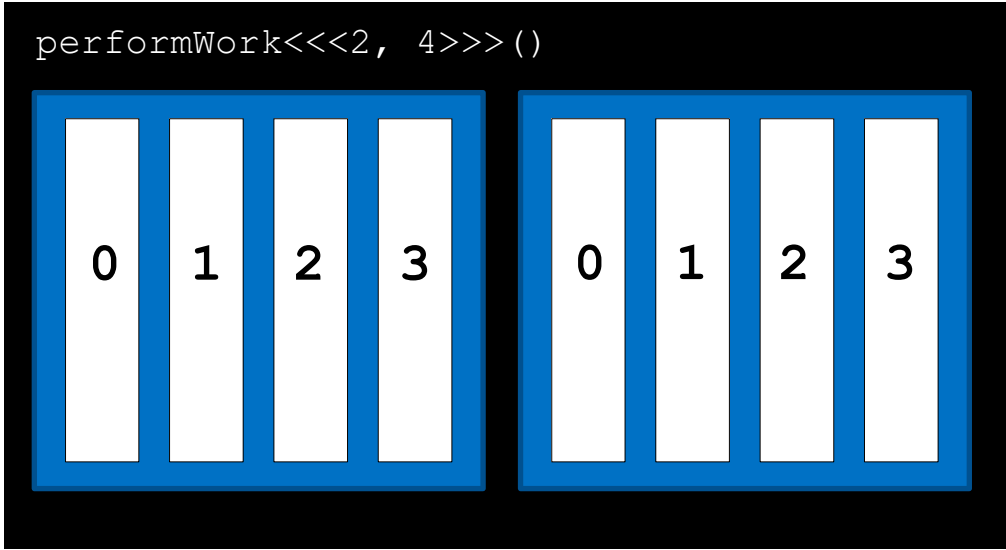


GPU DATA

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

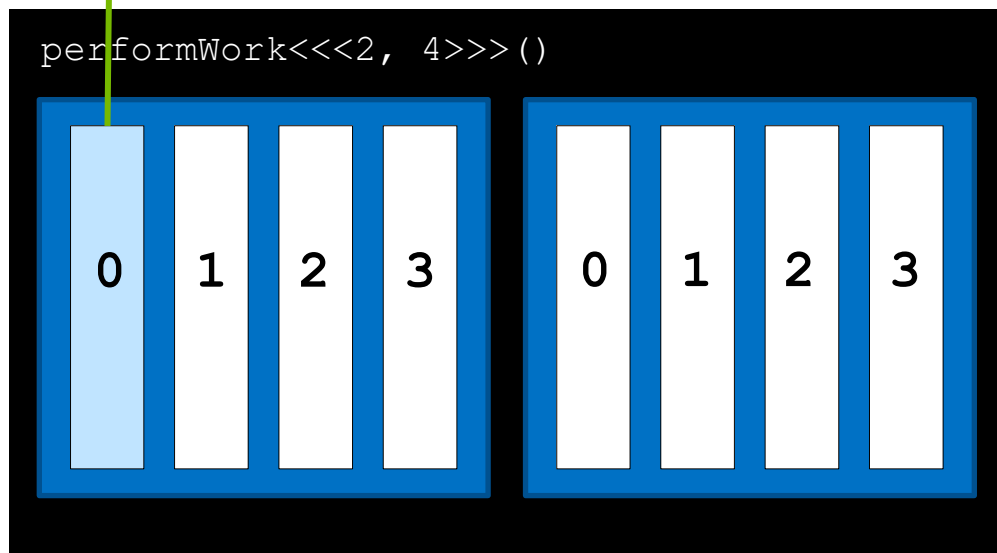


GPU DATA

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

GPU

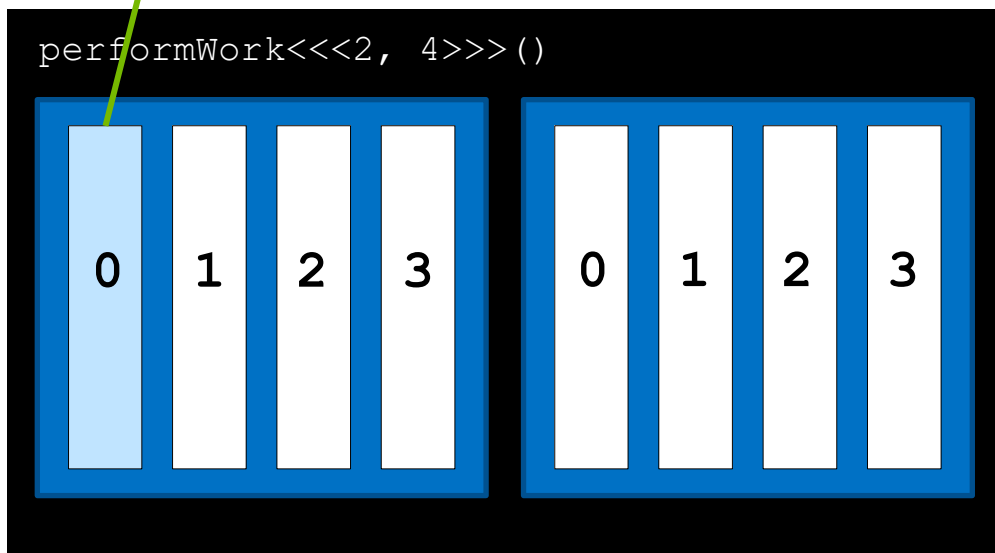


GPU
DATA

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

GPU

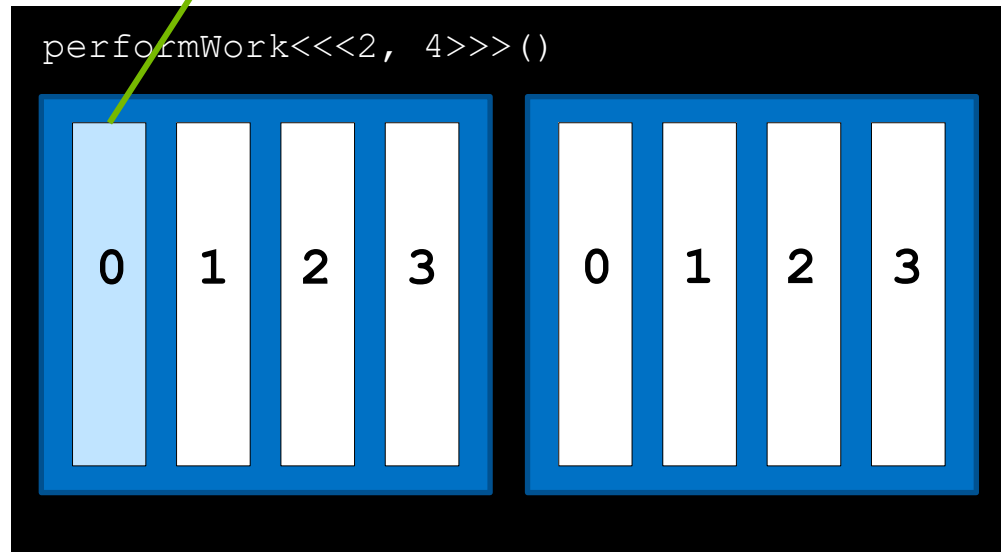


GPU
DATA

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

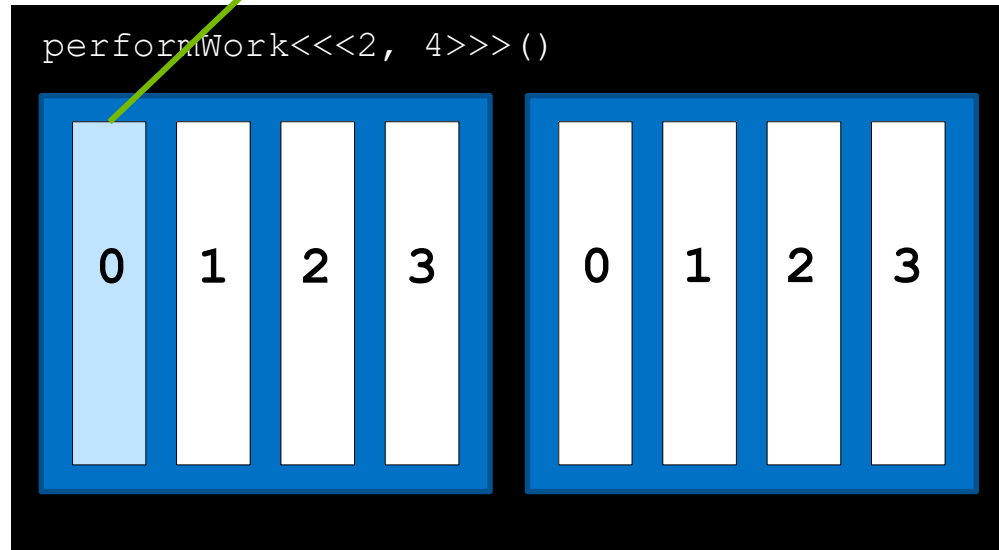


GPU
DATA

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

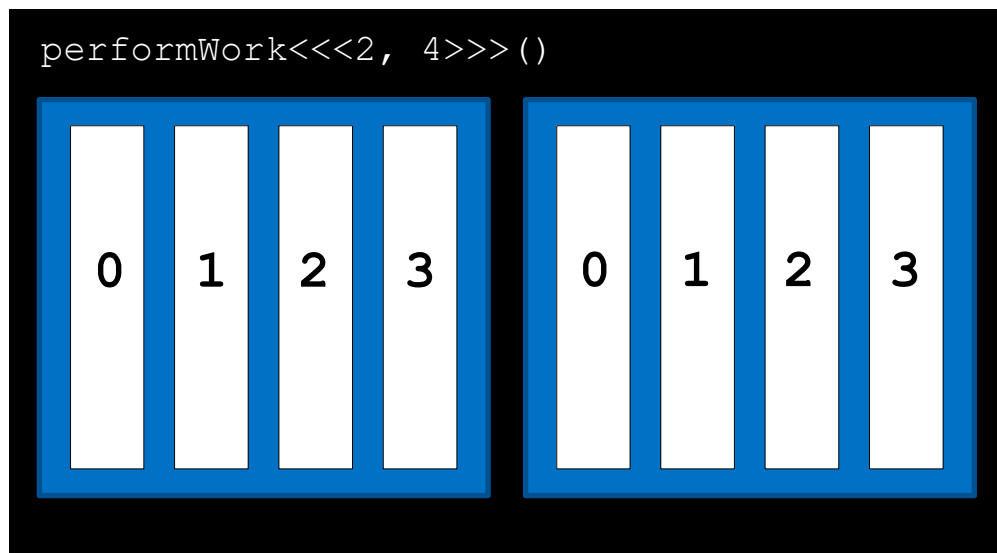


GPU DATA

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

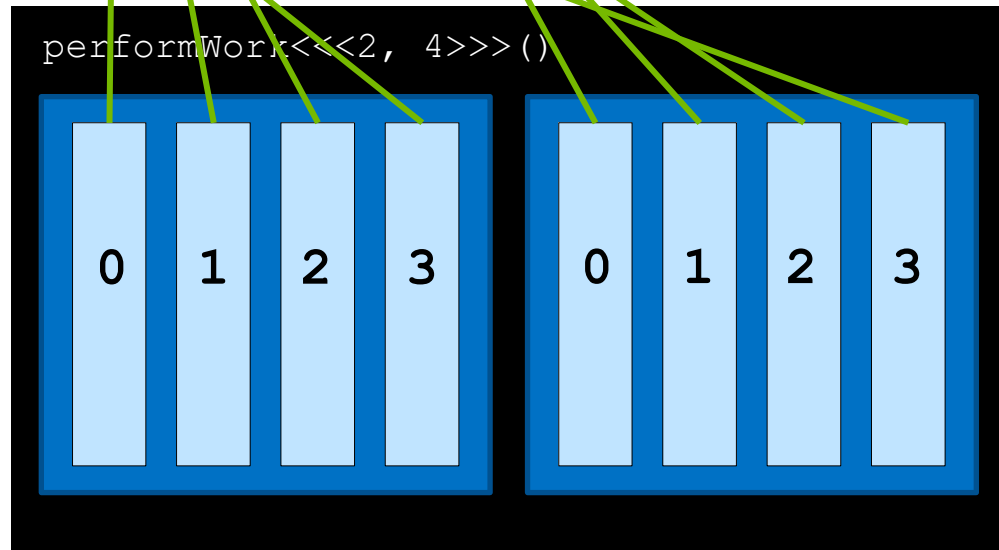


GPU
DATA

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

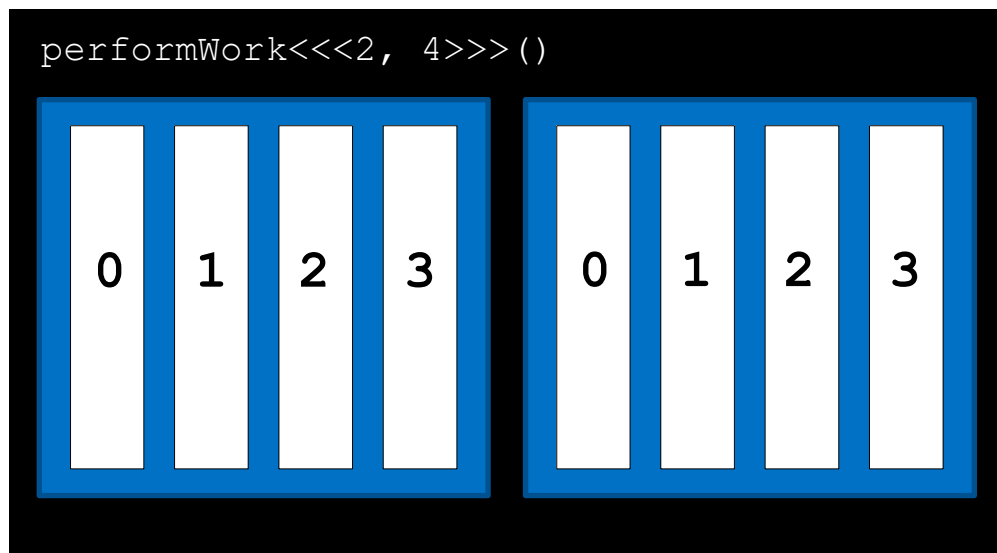


GPU
DATA

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

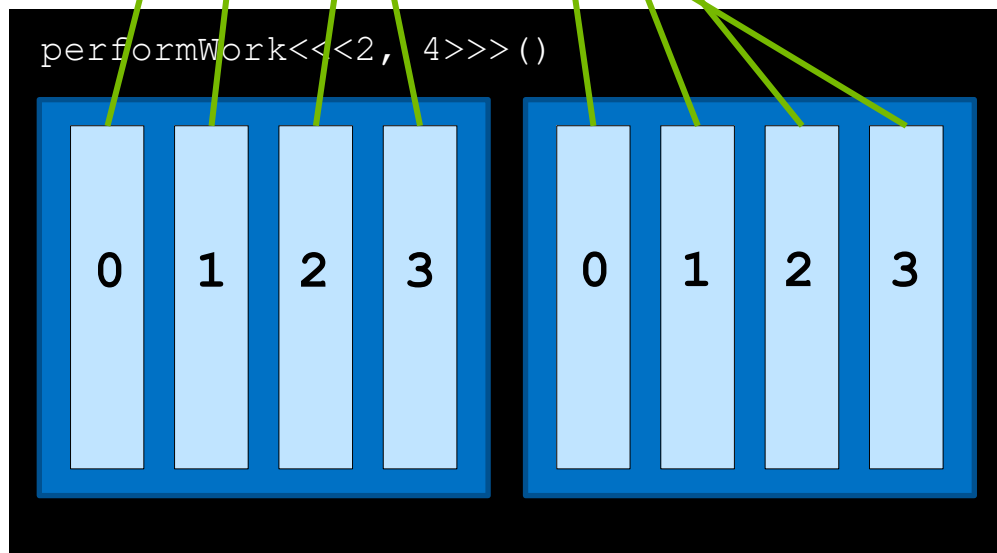


GPU
DATA

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

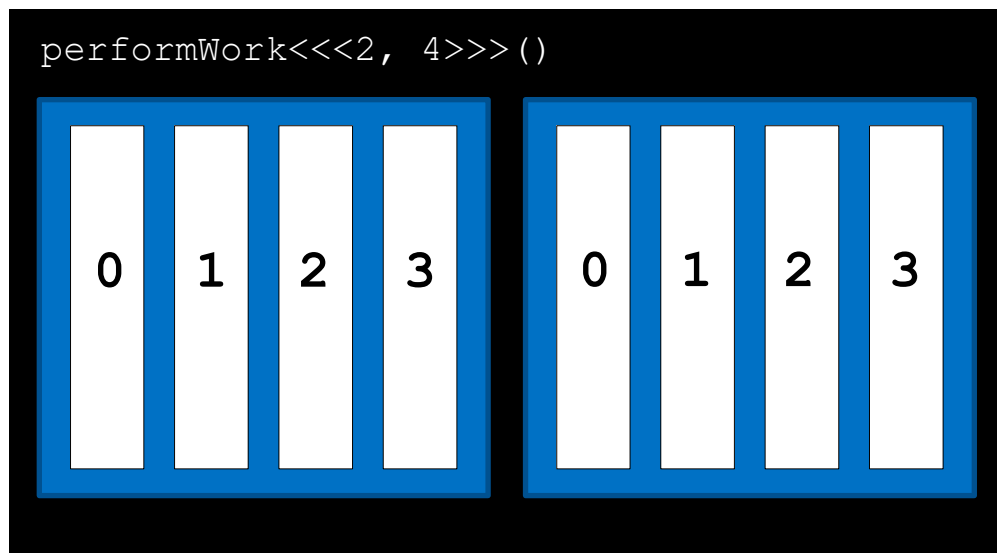


GPU DATA

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

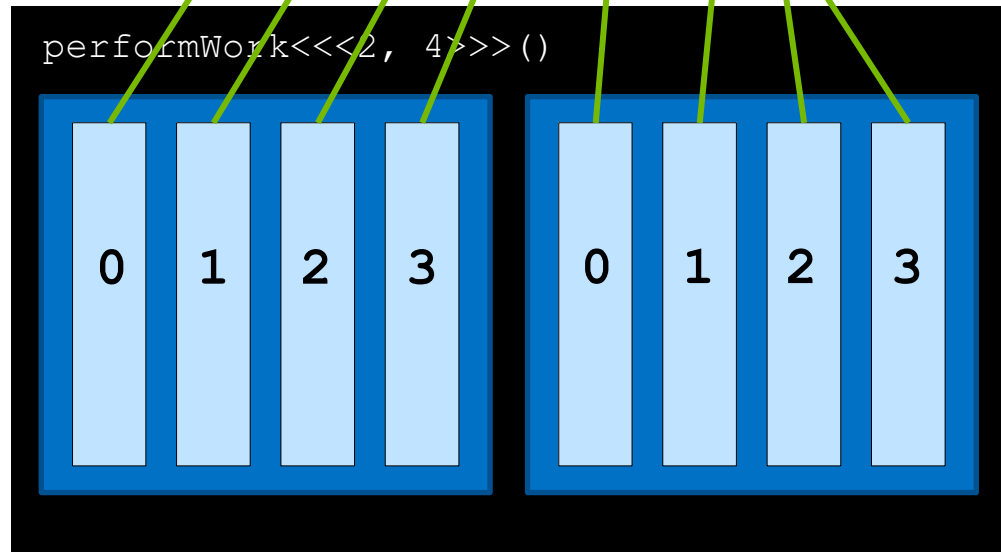


GPU
DATA

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

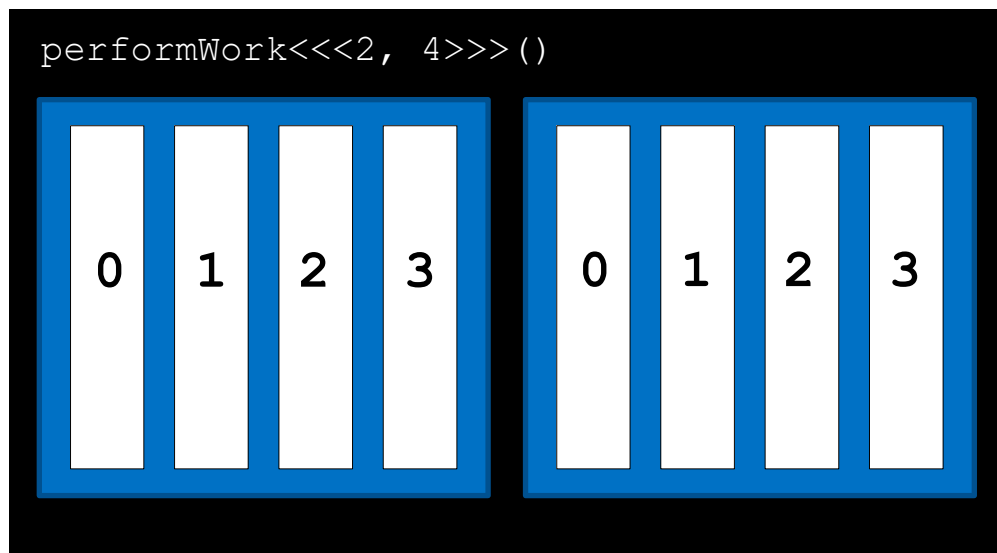


GPU
DATA

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

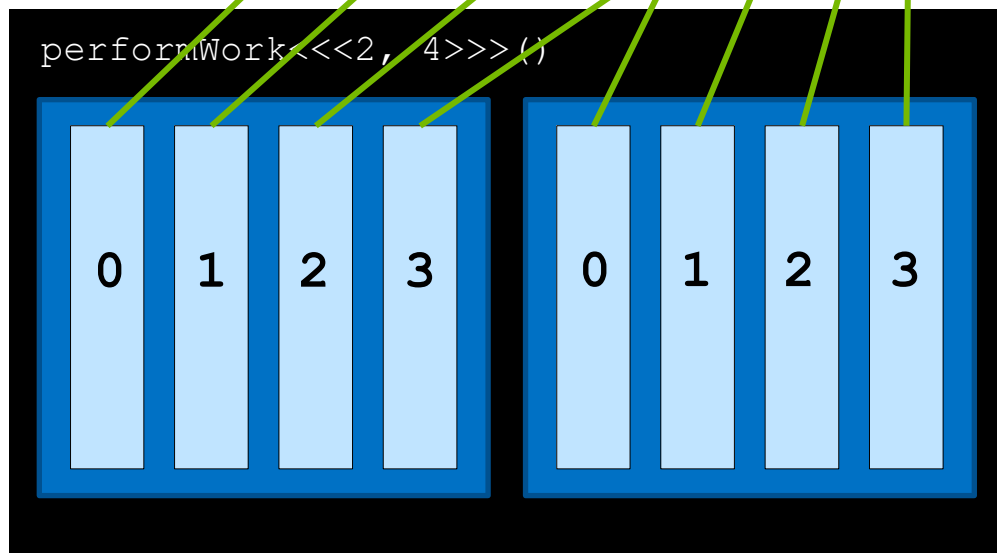


GPU
DATA

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

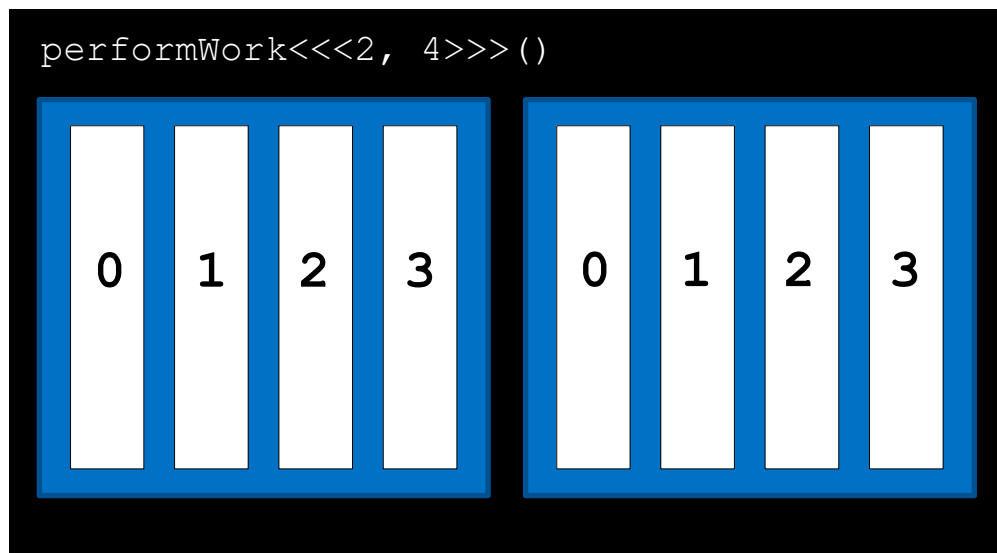


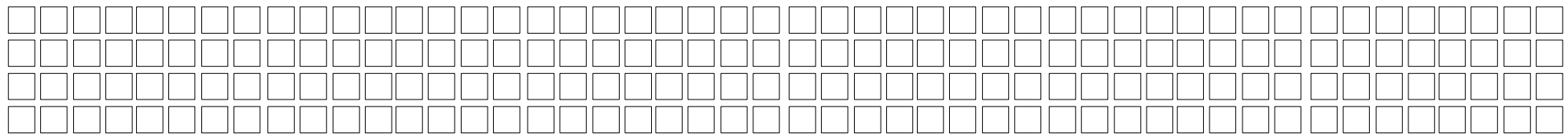
GPU DATA

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

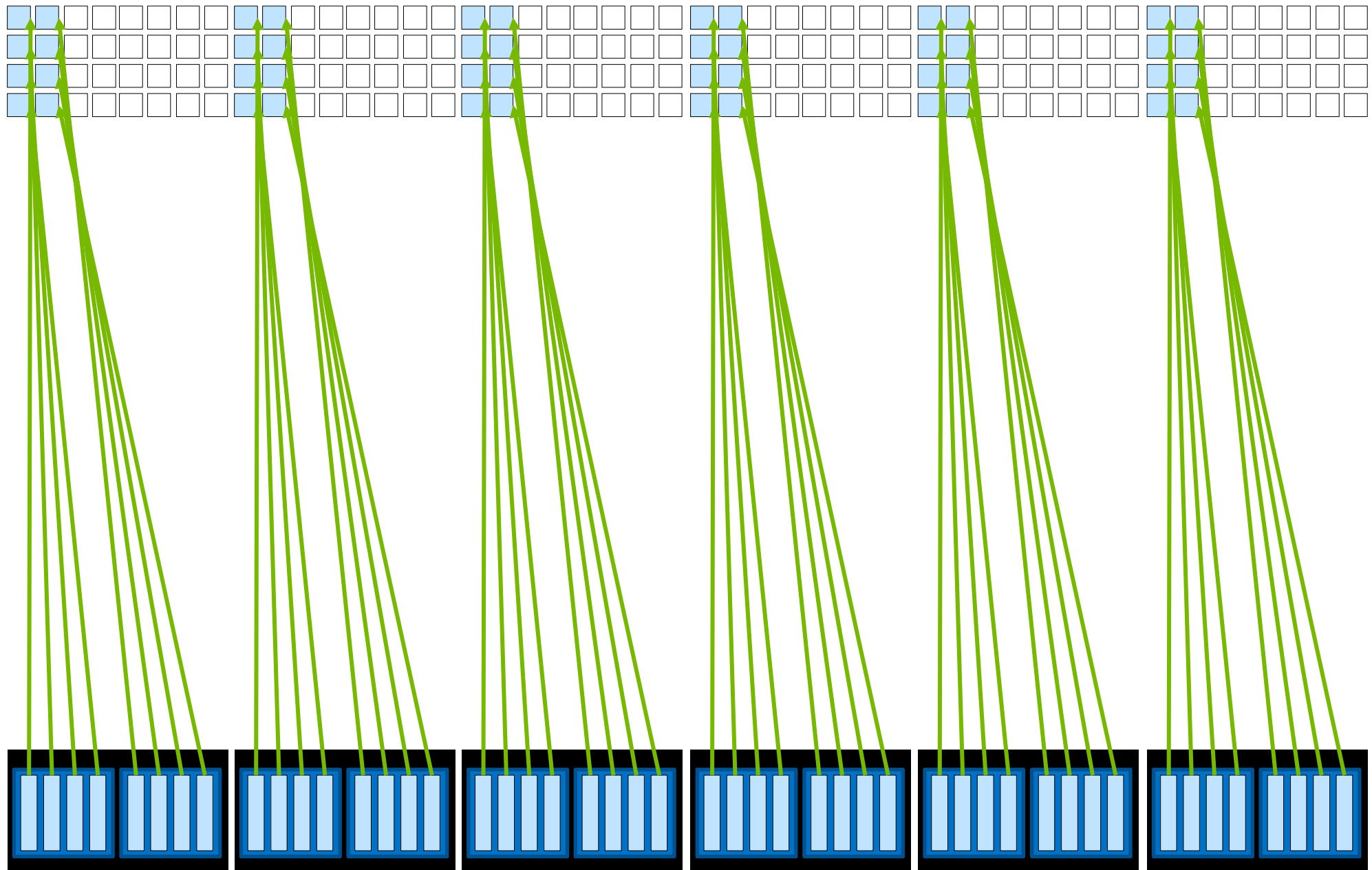
GPU

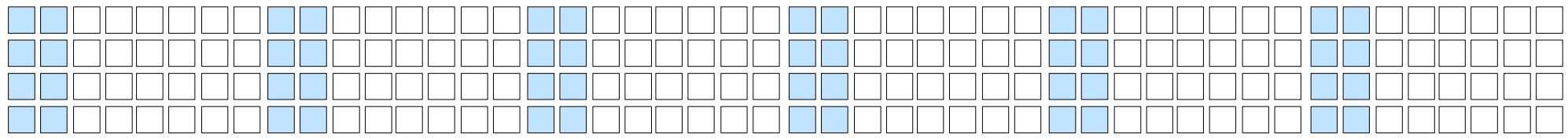


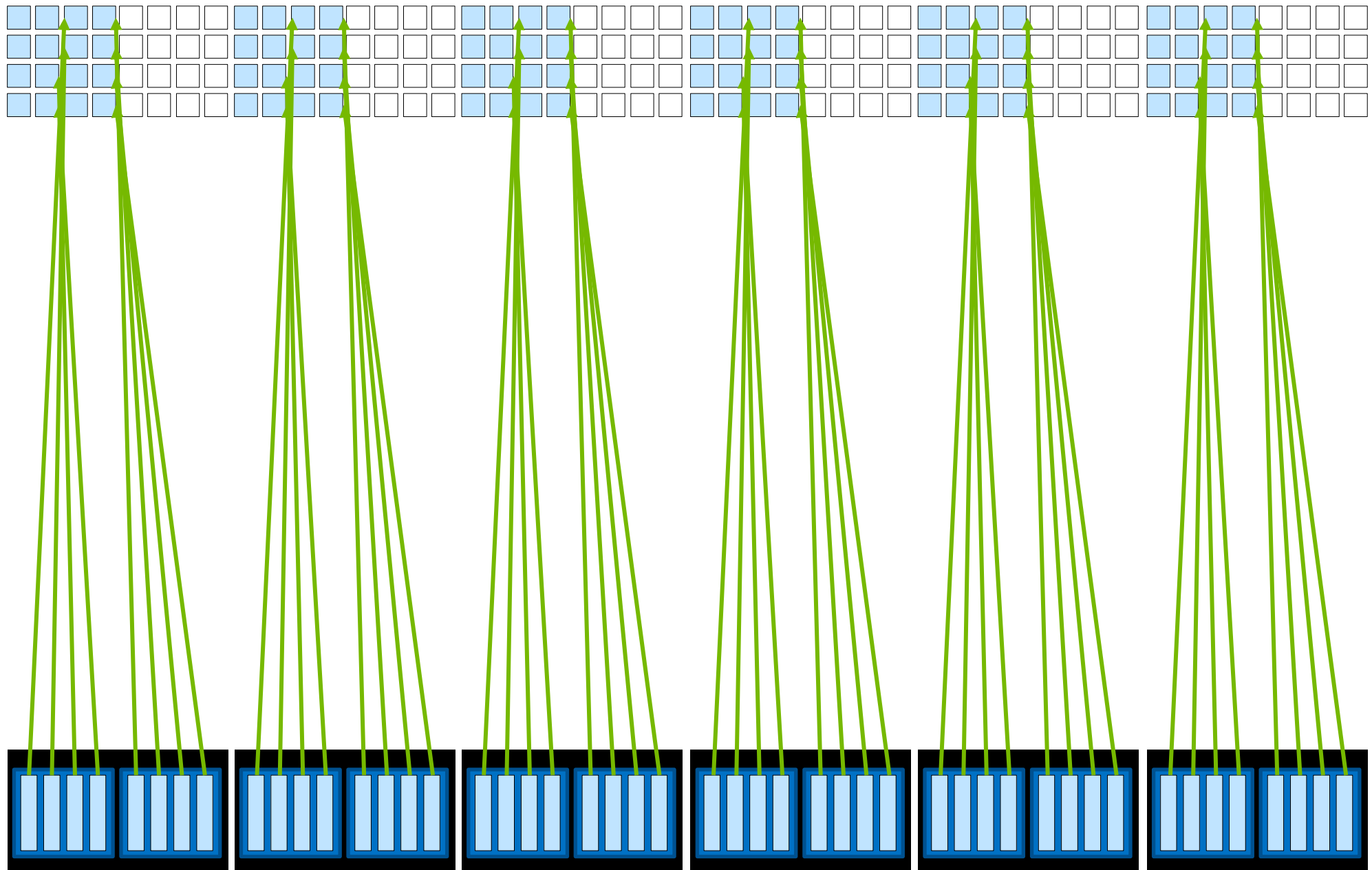


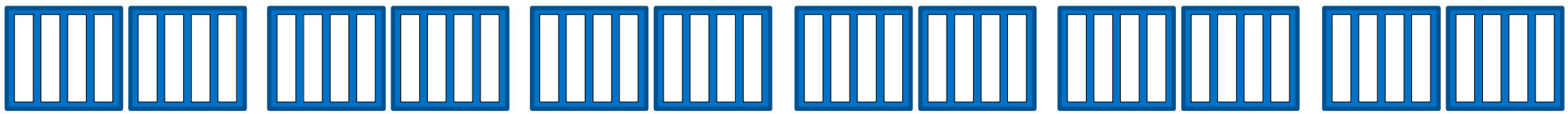
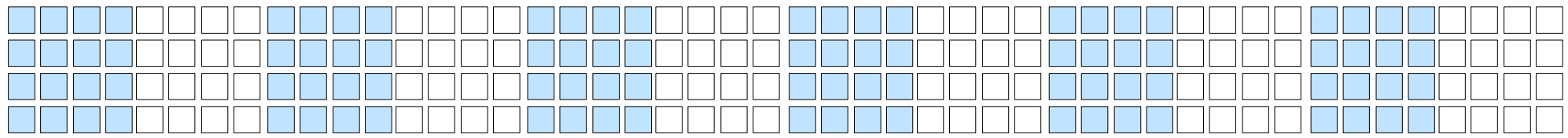
CUDA runs as many blocks in parallel at once as the GPU hardware supports, for massive parallelization

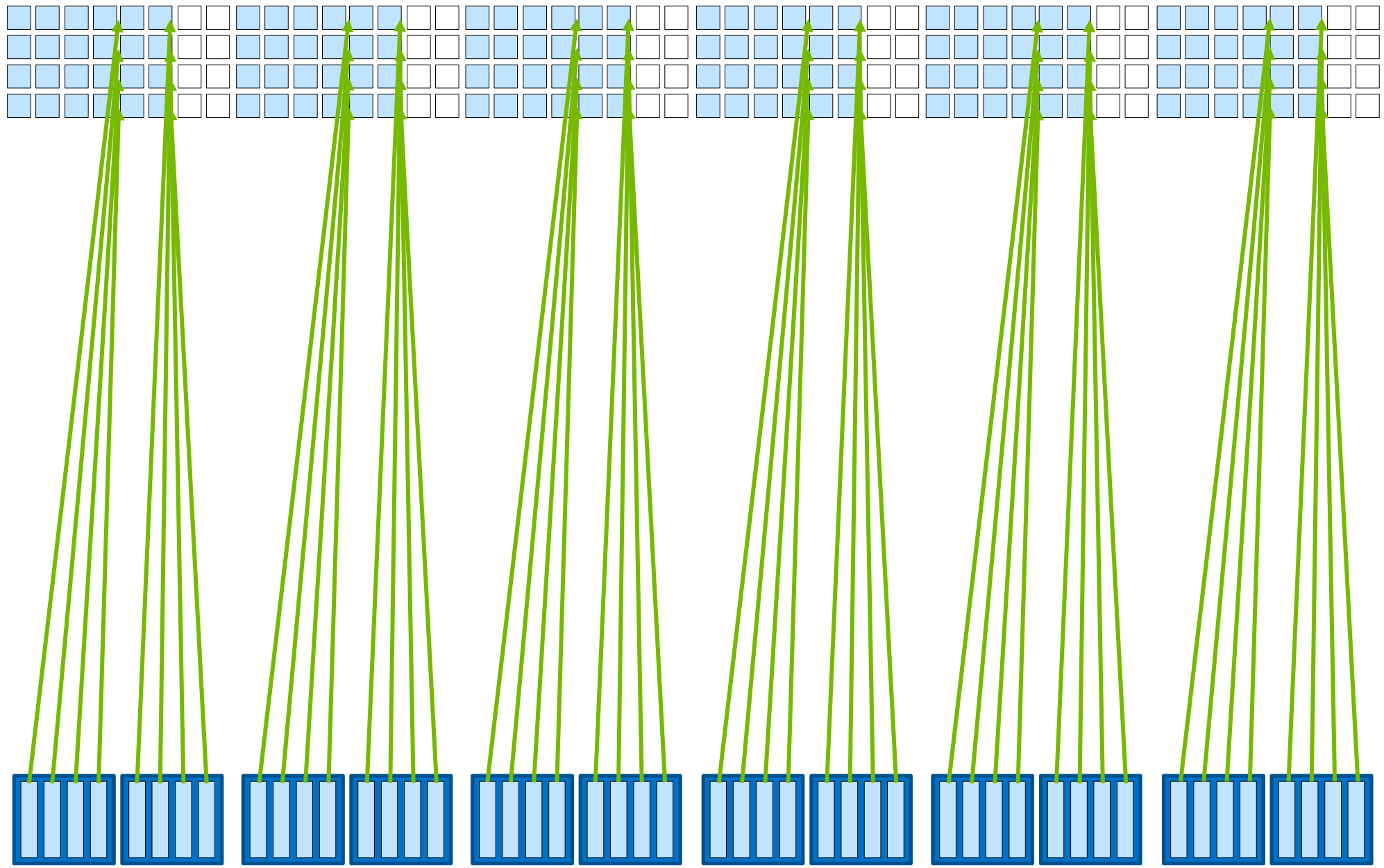


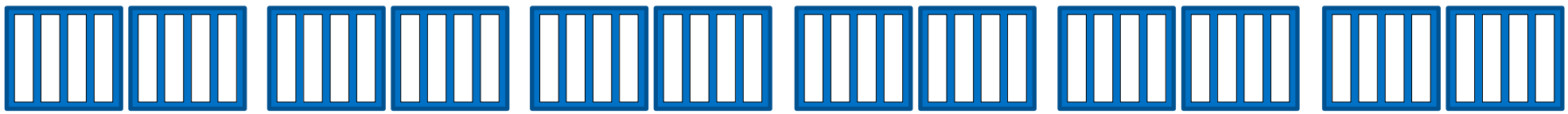
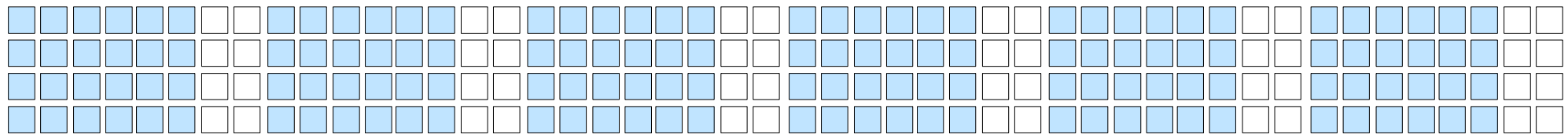


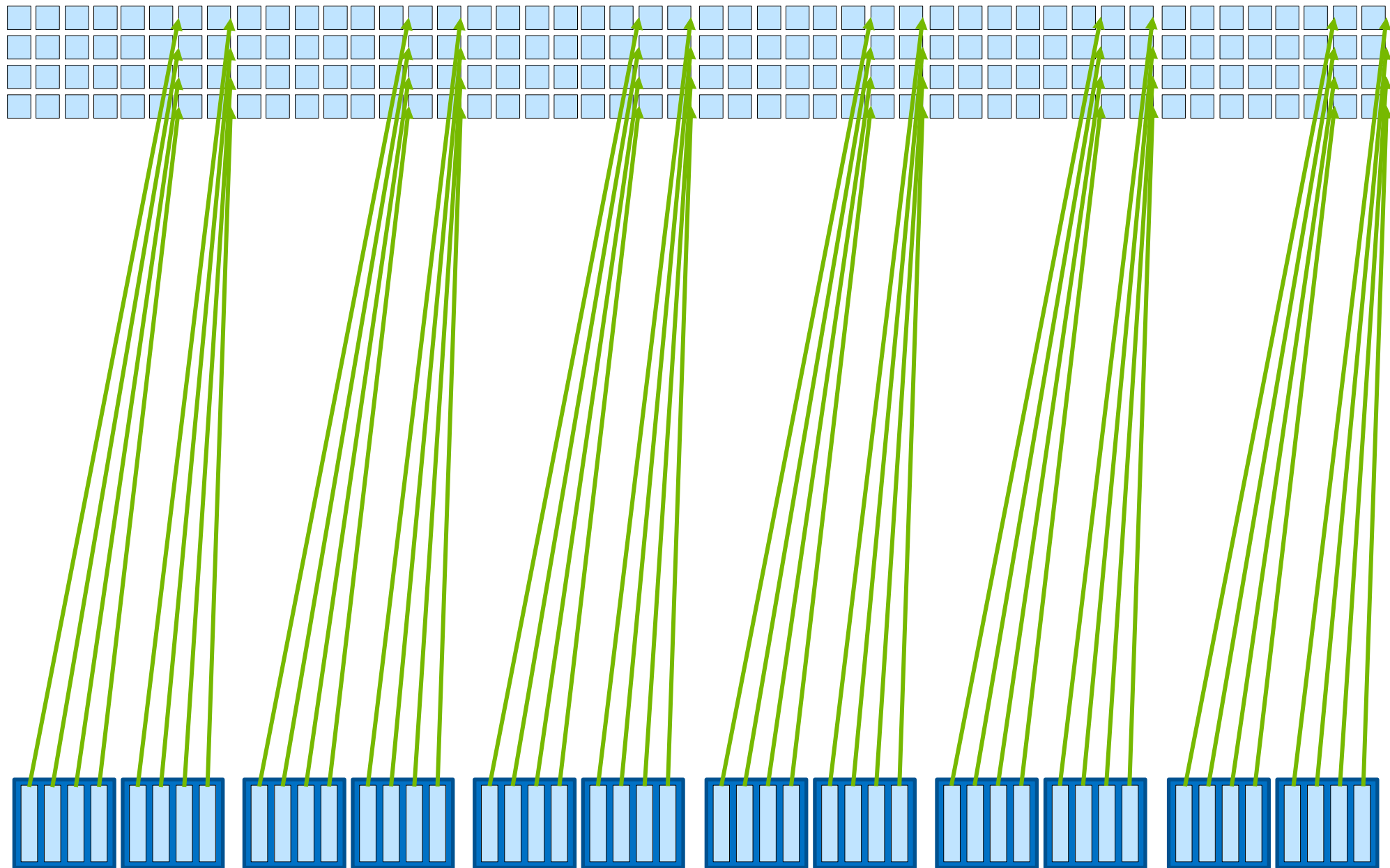


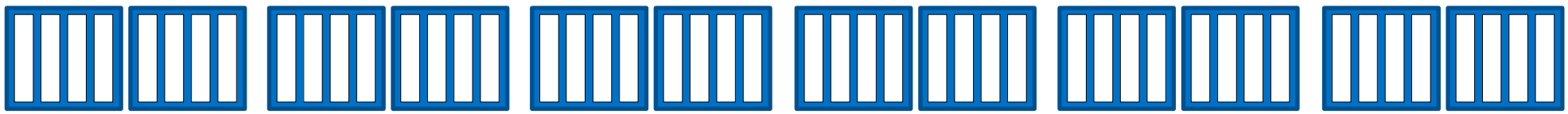
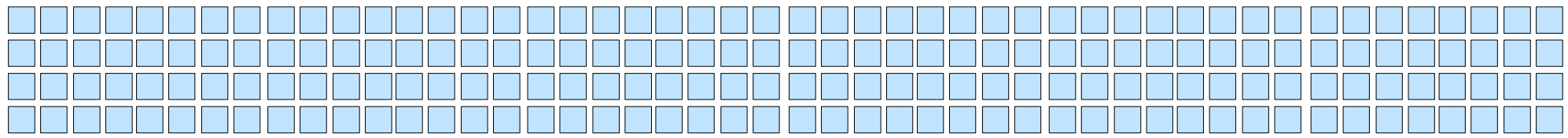












Occupancy

- GPU Occupancy
- Kernel Occupancy

GPU

SM

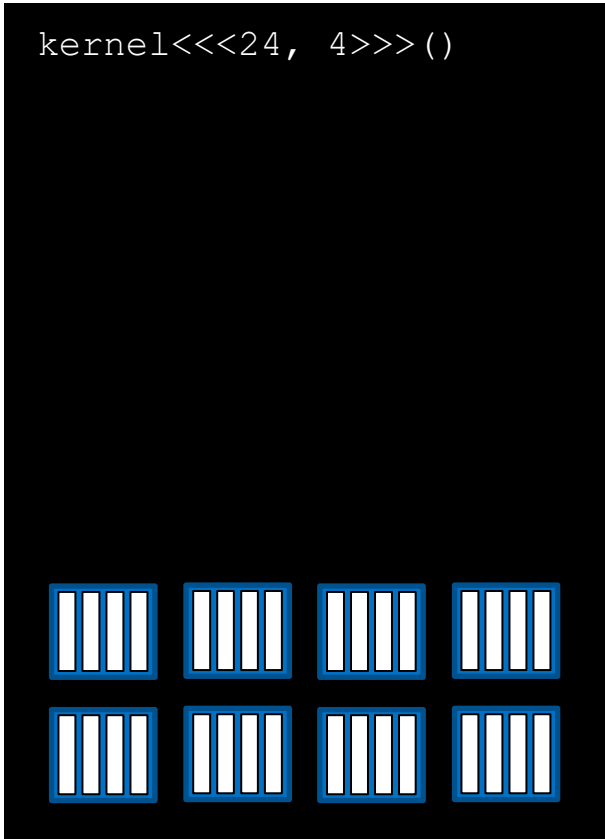
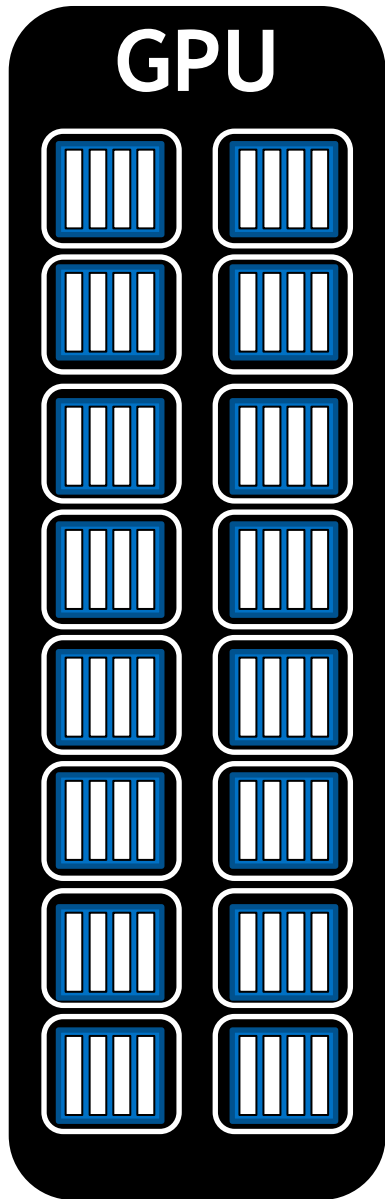
NVIDIA GPUs contain functional units called **Streaming Multiprocessors, or SMs**

GPU

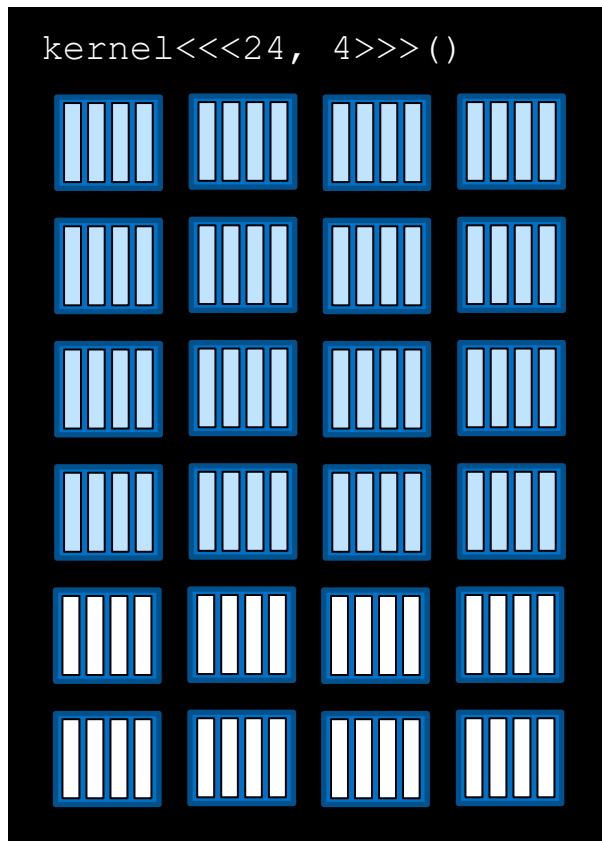
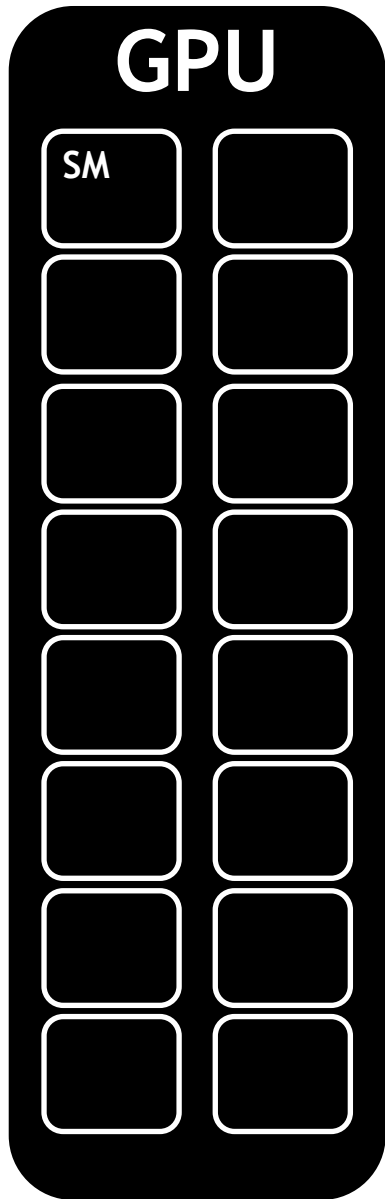
SM

```
kernel<<<24, 4>>>()
```

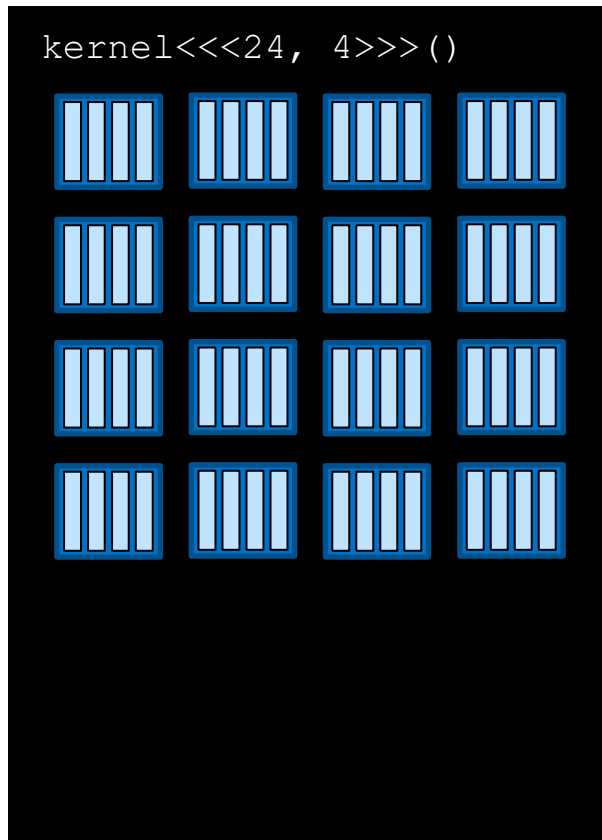
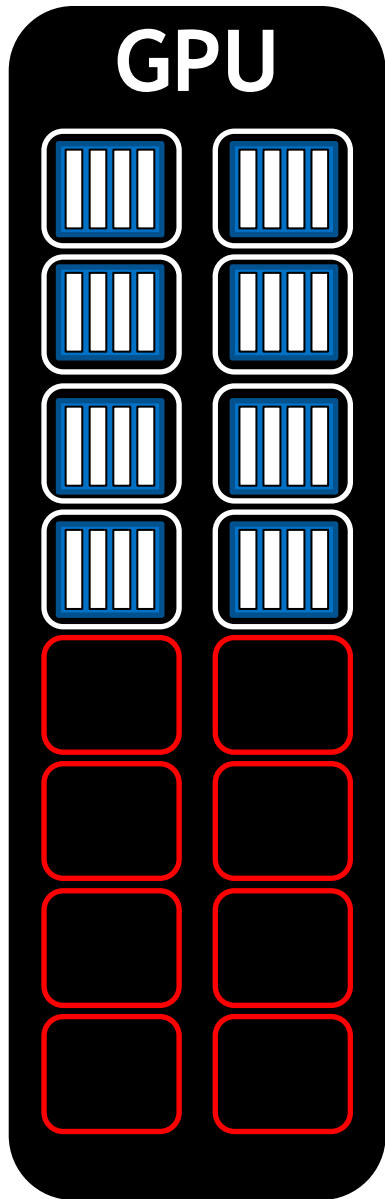
Blocks of threads are scheduled to run on SMs



Depending on the number of SMs on a GPU, and the requirements of a block, more than one block can be scheduled on an SM



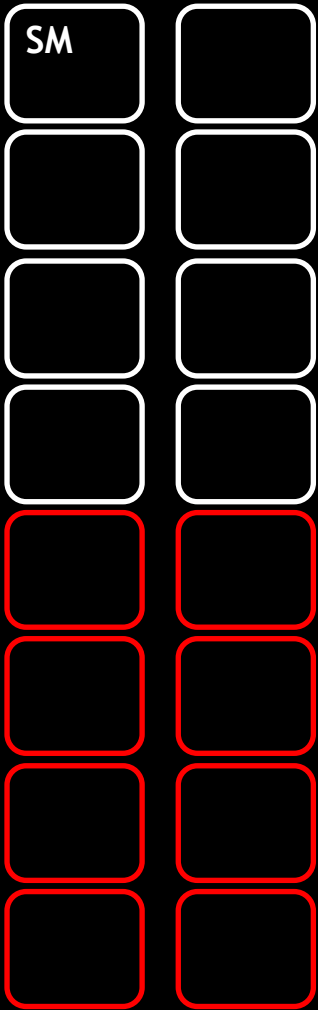
Depending on the number of SMs on a GPU, and the requirements of a block, more than one block can be scheduled on an SM



Grid dimensions divisible by the number of SMs on a GPU can promote full SM utilization

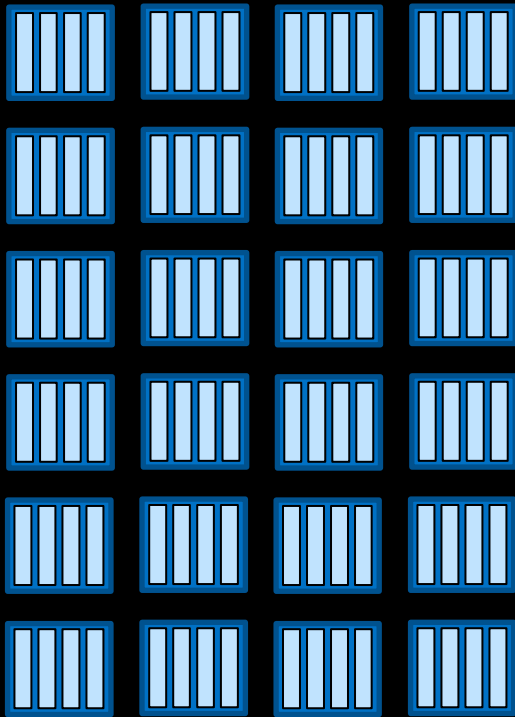
GPU

SM



Here there are follow SMs

```
kernel<<<24, 4>>>()
```



Kernel Occupancy

compile your code with the following nvcc option:

--ptxas-options=-v

you will get something like this:

ptxas info : 0 bytes gmem

ptxas info : Compiling entry function
'_Z28MyKernelhS0_S0_PK3CDRS0_Ph' for 'sm_20'

ptxas info : Function properties for
'_Z28MyKernelhS0_S0_PK3CDRS0_Ph'

24 bytes stack frame, 0 bytes spill stores, 0 bytes spill
loads

ptxas info : Used 29 registers, 80 bytes cmem[0], 4
bytes cmem[16]

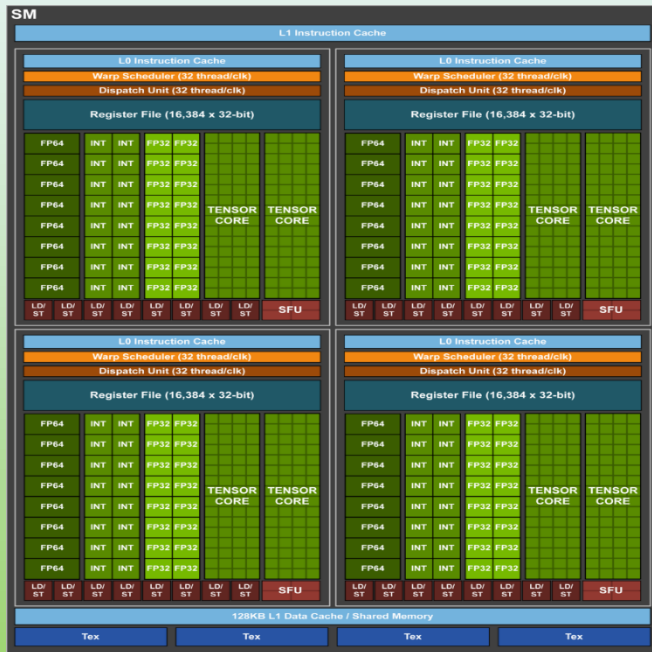
Remember:

**In the H100 you have 256KB
register per multiprocessor.**

**The total number of “threads X
register per threads” cannot
exceed this value!!**

Kernel Occupancy

Warp = 32 threads



Technical specifications	Compute capability (version)																
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0 (7.2?)	7.5
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16	4	32				16	128	32	16	128		
Maximum dimensionality of grid of thread blocks	2				3												
Maximum x-dimension of a grid of thread blocks	65535					$2^{31} - 1$											
Maximum y-, or z-dimension of a grid of thread blocks	65535																
Maximum dimensionality of thread block	3																
Maximum x- or y-dimension of a block	512				1024												
Maximum z-dimension of a block	64																
Maximum number of threads per block	512				1024												
Warp size	32																
Maximum number of resident blocks per multiprocessor	8				16				32				16				
Maximum number of resident warps per multiprocessor	24	32	48	64												32	
Maximum number of resident threads per multiprocessor	768	1024	1536	2048												1024	
Number of 32-bit registers per multiprocessor	8 K	16 K	32 K	64 K			128 K			64 K							
Maximum number of 32-bit registers per thread block	N/A			32 K	64 K	32 K	64 K				32 K	64 K	32 K	64 K			
Maximum number of 32-bit registers per thread	124			63			255										

$$\text{Occupancy} = \frac{\text{max number of active warps}}{\text{max number of warps per SM}}$$

Block size should be a multiple of 32 (128,256 usually)!

Spreadsheet calculator

<https://xmartlabs.github.io/cuda-calculator/>

Or google for:

CUDA spreadsheet calculator excel

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 $\text{gridDim.x} * \text{blockDim.x}$, which is the number of threads in the grid.

Questions?

How to access to the course online

1. WIFI Info: **eduroam**
2. Browser Recommendation: **Chrome**
3. websocketstest.courses.nvidia.com
4. <https://learn.nvidia.com/dli-event>
5. Create an Nvidia Developer Account (if you have not done yet)
6. Event code: **STFC_CUDA_AMBASSADOR_SE24**
7. Work through the Introduction Section and 'Start' launching your first GPU task

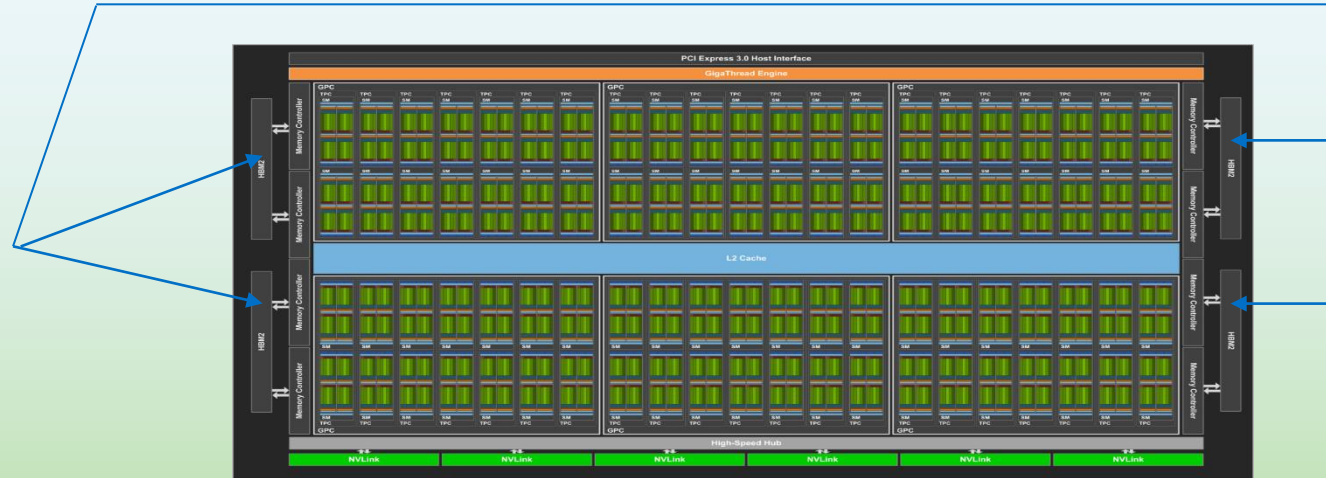
Memory Management

- Type of Memories
- Unified memory behaviour
- Non-Unified memory
- `cudaMemcpyAsync`

Type of Memories

Device Memory:

- Global Memory
- Texture Memory
- Local Memory
- Constant Memory



DRAM

On Chip Memory:

- Shared Memory (L1 Cache)
- Registers



Device Memory

Memory	Location	Access	Scope	Lifetime	CUDA qualifier
Global	DRAM	RW	All threads and host	Application	cudaMalloc cudaMallocManaged cudaMallocHost
Local	DRAM	RW	1 thread	Thread	
Constant	DRAM	R	All threads and host	Application	__constant__
Texture	DRAM	R	All threads and host	Application	texture

- Global memory is the largest memory on the GPU
- Local memory are local registers spilled into the global memory
- Constant memory is useful to store constant read in values (ex. g , c , R , etc.)
- Texture memory allows interpolation on 2D constant matrix values (ex. PVT table)

On chip Memory

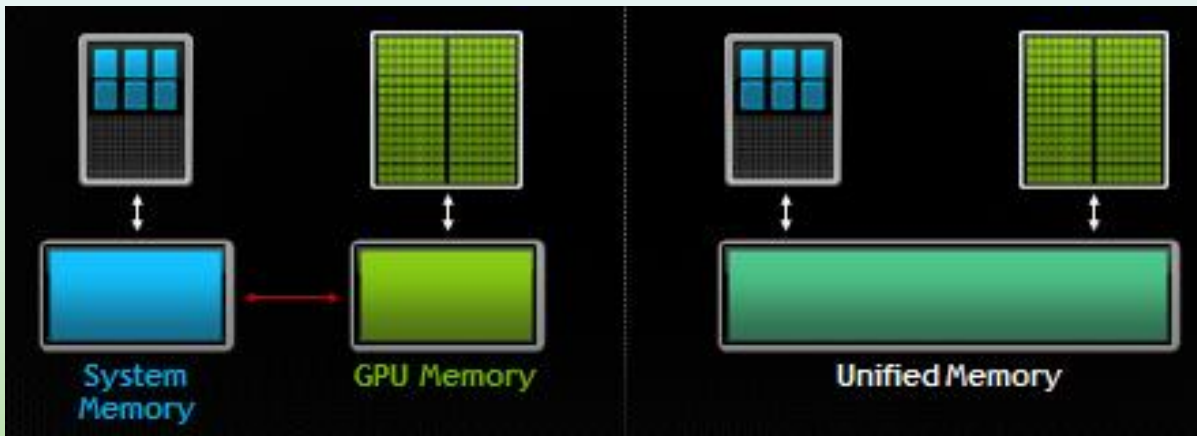
Memory	Location	Access	Scope	Lifetime	CUDA qualifier
Register	On chip	RW	1 thread	Thread	
Shared	On chip	RW	All threads in a block	Block	__shared__

- Registers per thread can be found at compile time
- Shared memory can enhance performance when locality is high

Unified Memory (2014)

Before CUDA 6

After CUDA 6



Simplifies code writing

Applies to Global Memory only!

CPU Code

```
void sortfile(FILE *fp, int N) {
    char *data;
    data = (char *)malloc(N);

    fread(data, 1, N, fp);

    qsort(data, N, 1, compare);

    use_data(data);

    free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged(&data, N);

    fread(data, 1, N, fp);

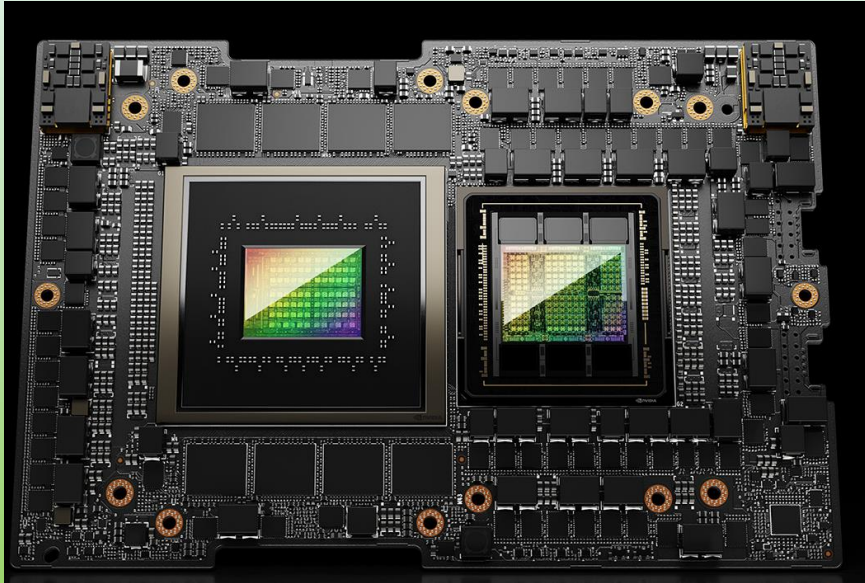
    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();

    use_data(data);

    cudaFree(data);
}
```

Grace-Hopper Unified Memory (2022)

Careful!! This concept is NOT the unified memory referred here!



In CUDA:

-gpu=unified

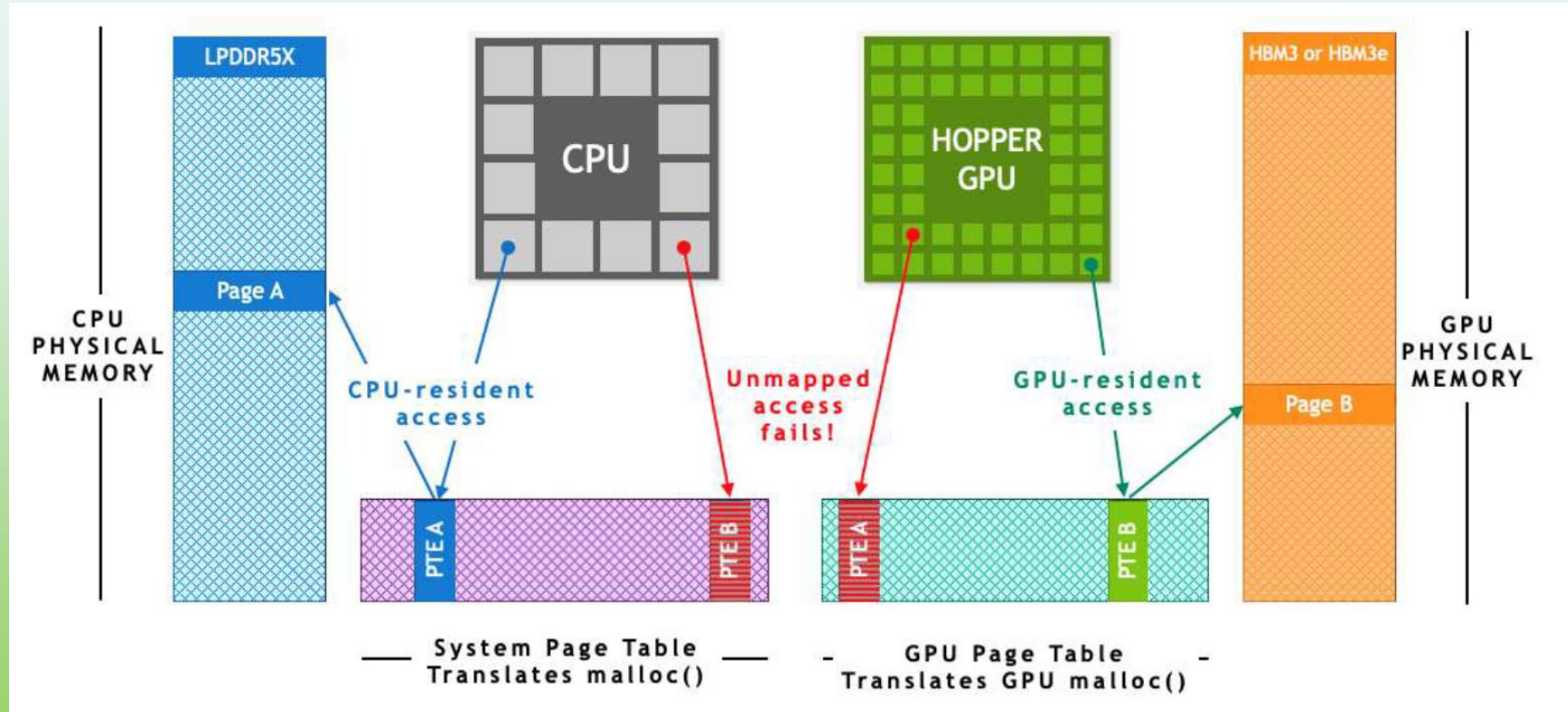
-gpu=unified (implies managed)

<https://developer.nvidia.com/blog/unified-memory-cuda-beginners/>

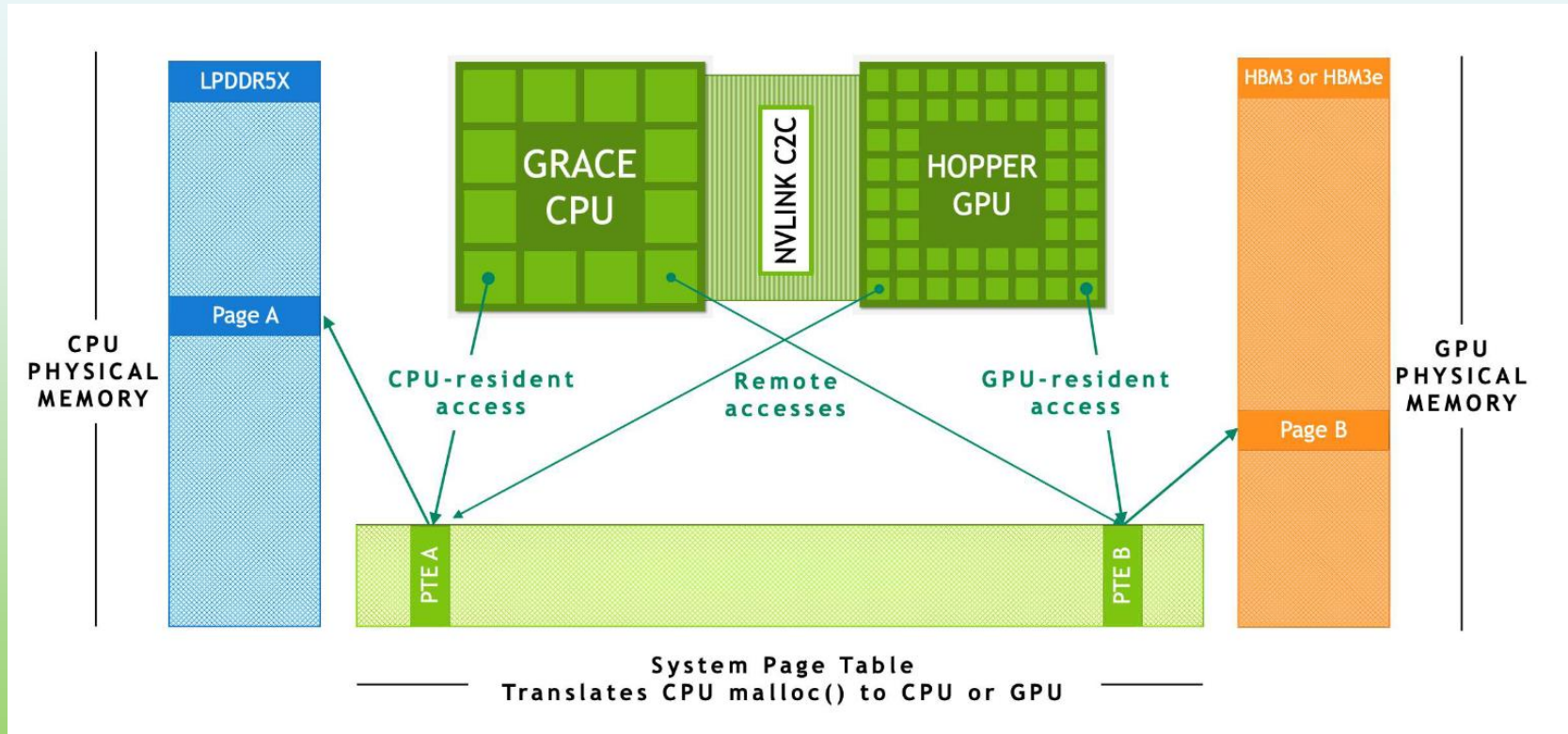
In OpenACC

-ta=tesla:managed

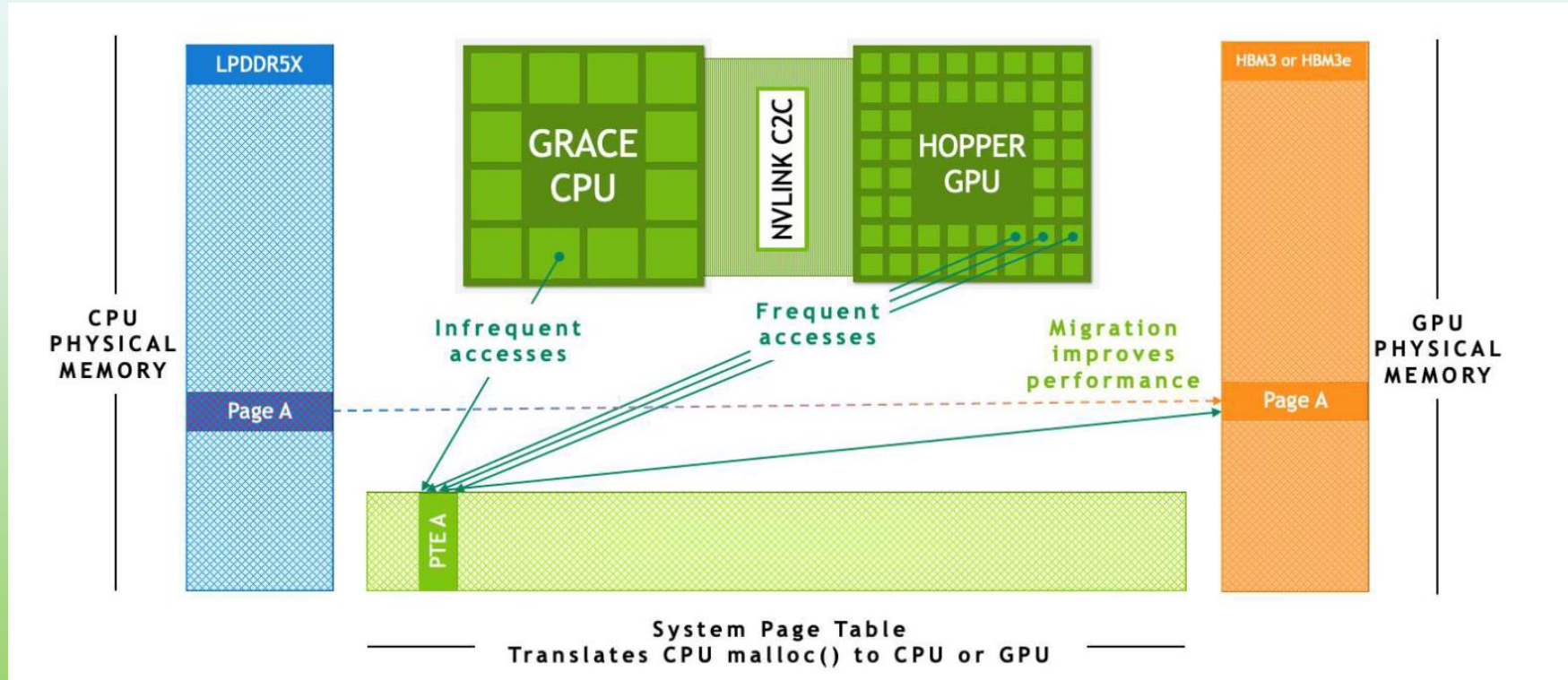
New unified memory (I)



New unified memory (II)

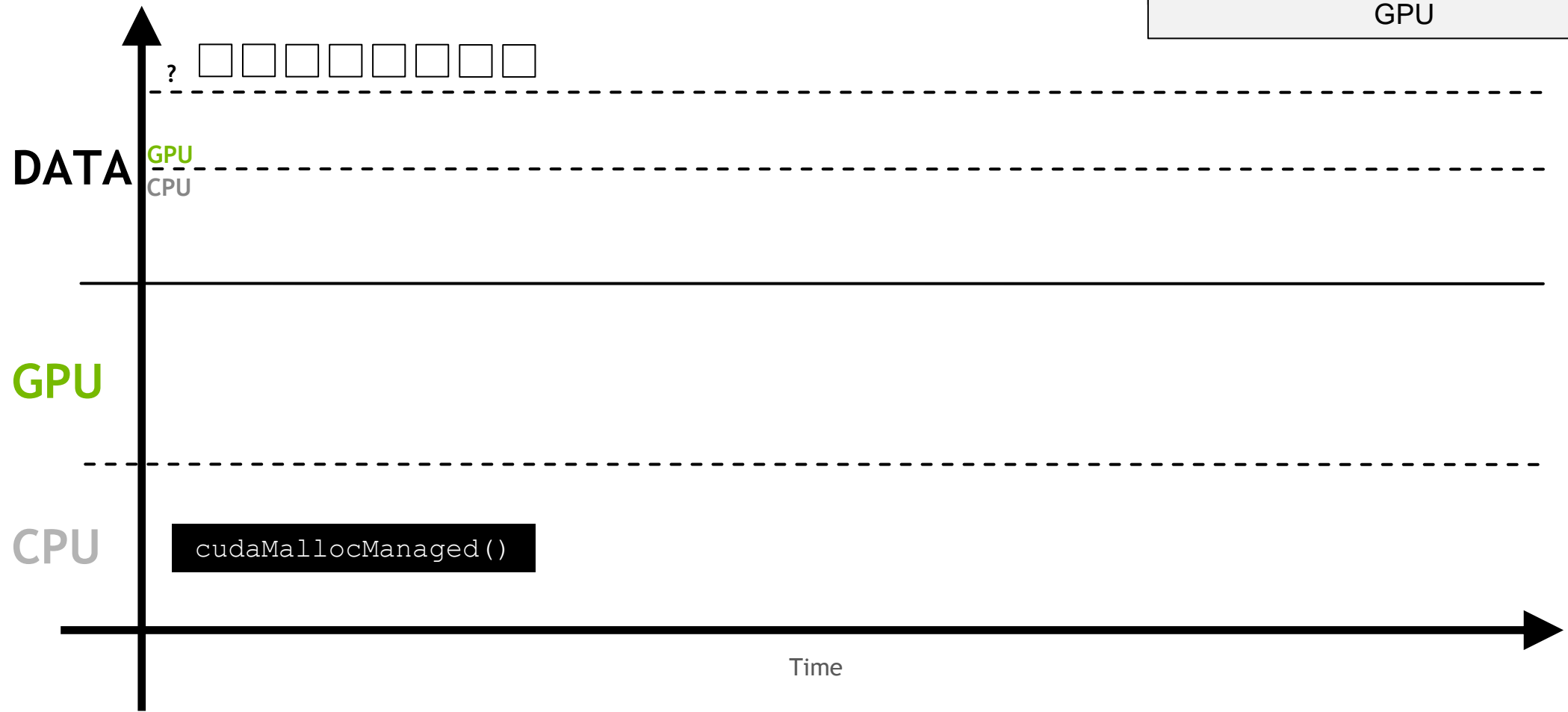


New unified memory (III)

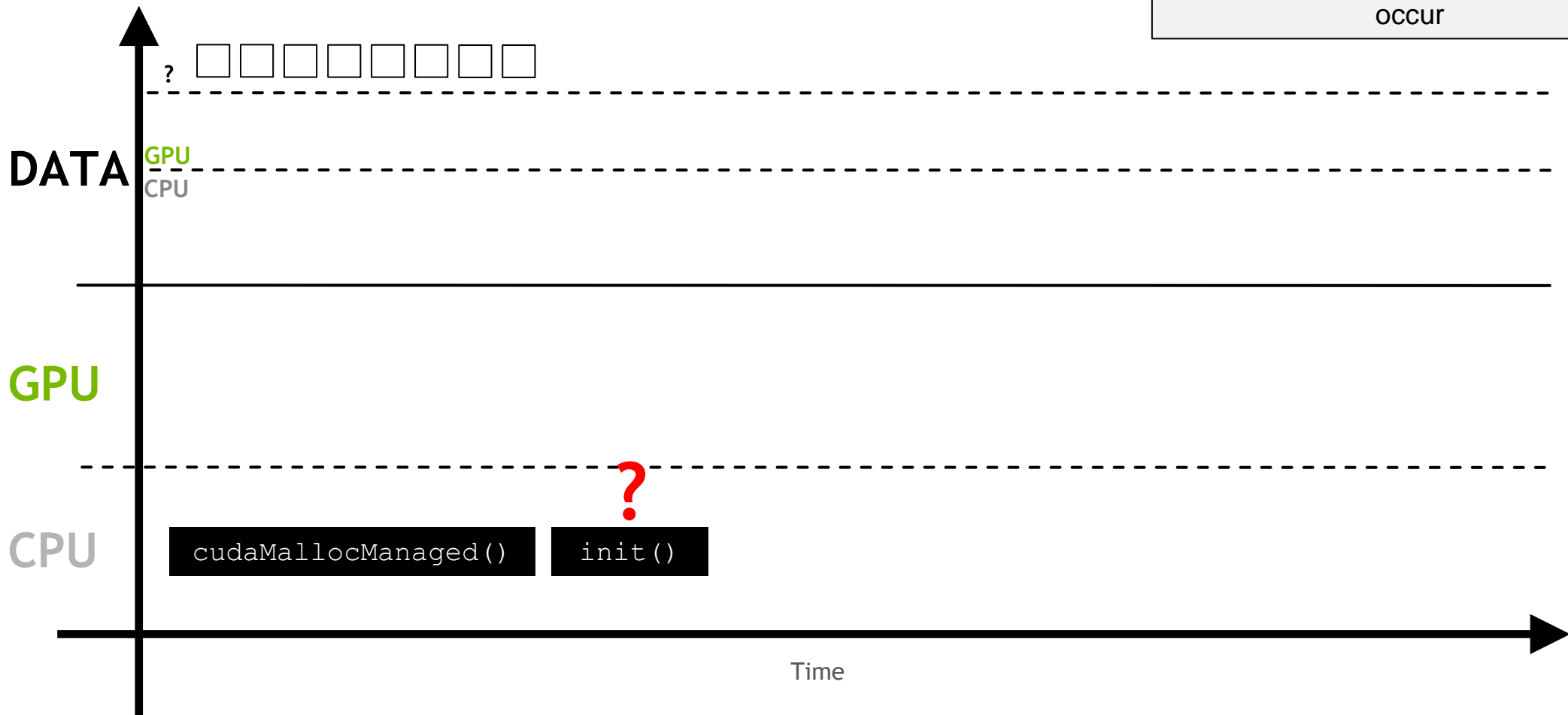


Unified Memory Behavior

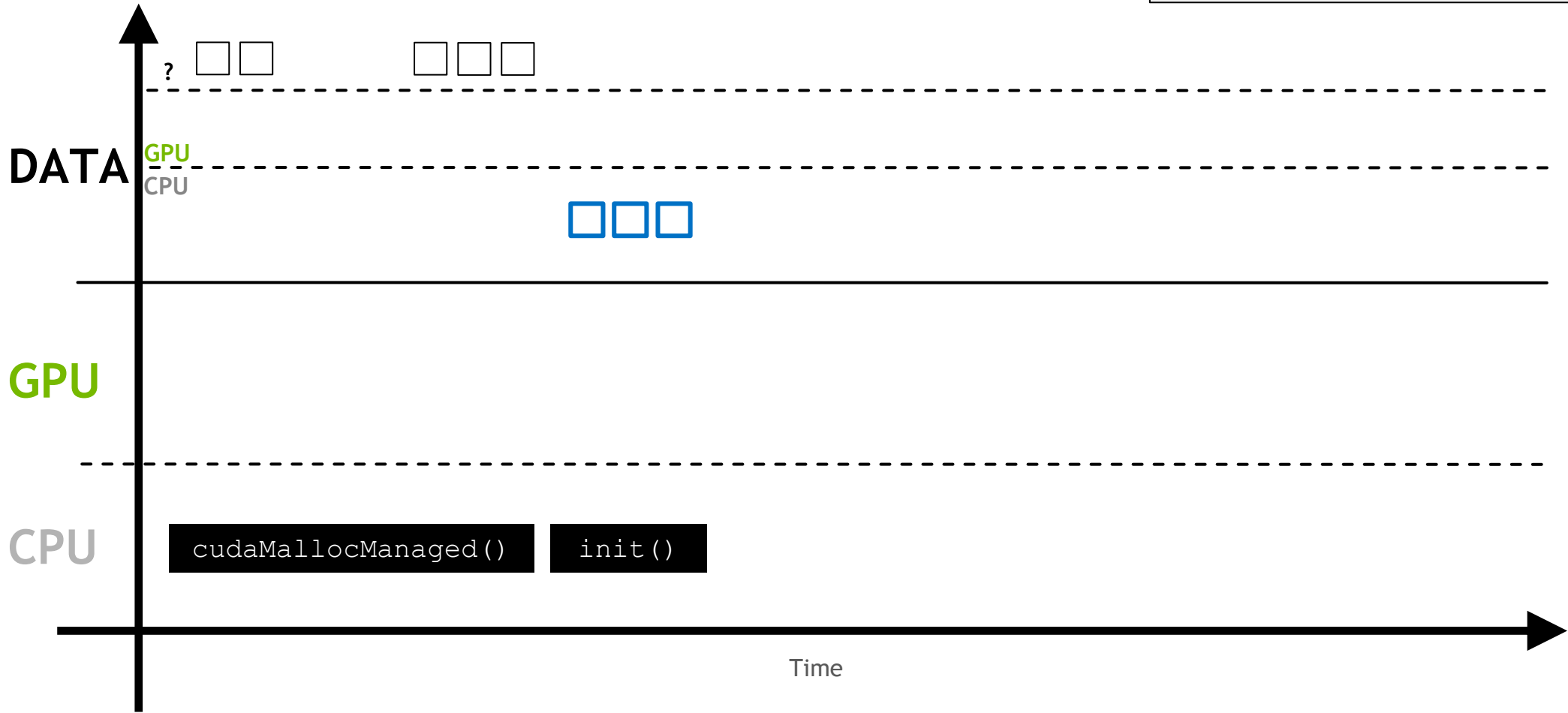
When **UM** is allocated, it may not be resident initially on the CPU or the GPU



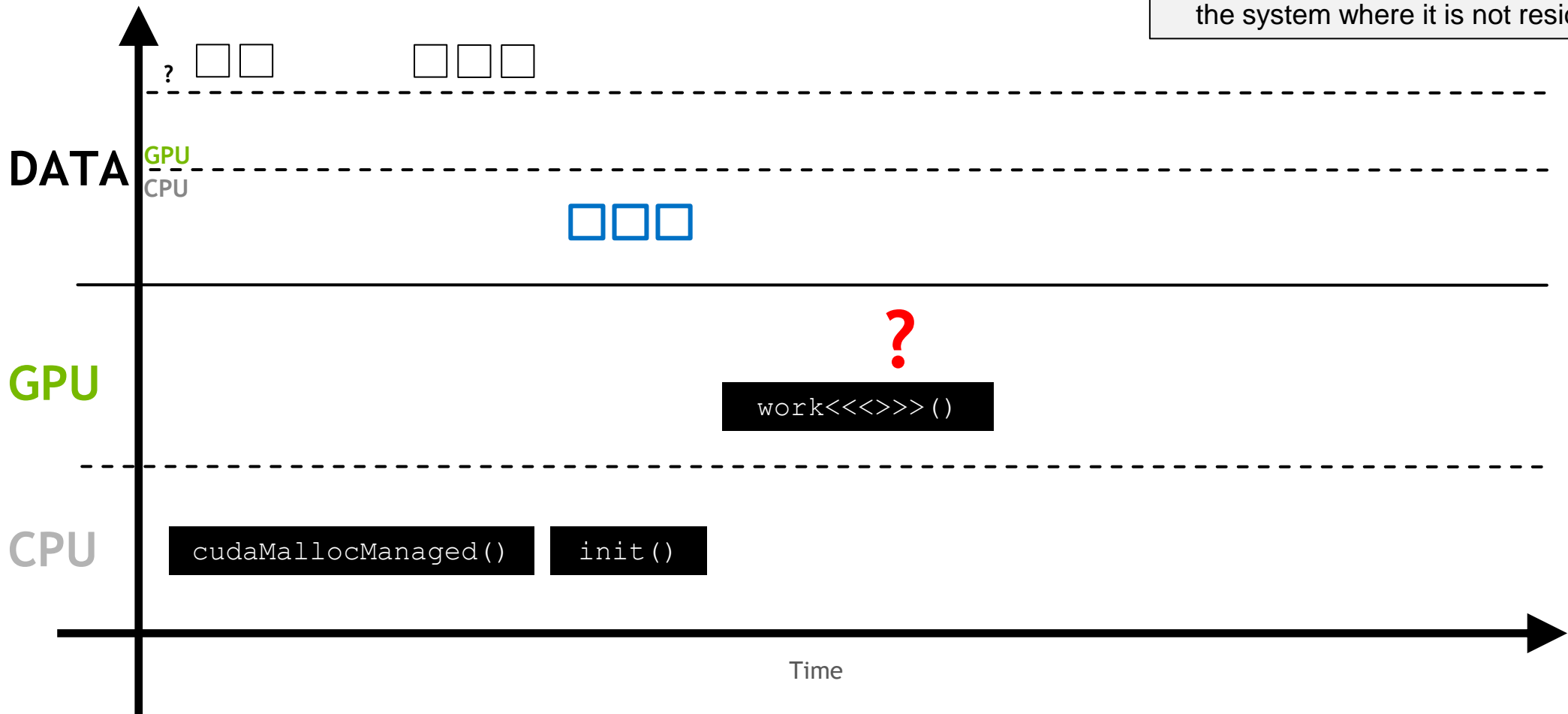
When some work asks for the memory for the first time, a **page fault** will occur



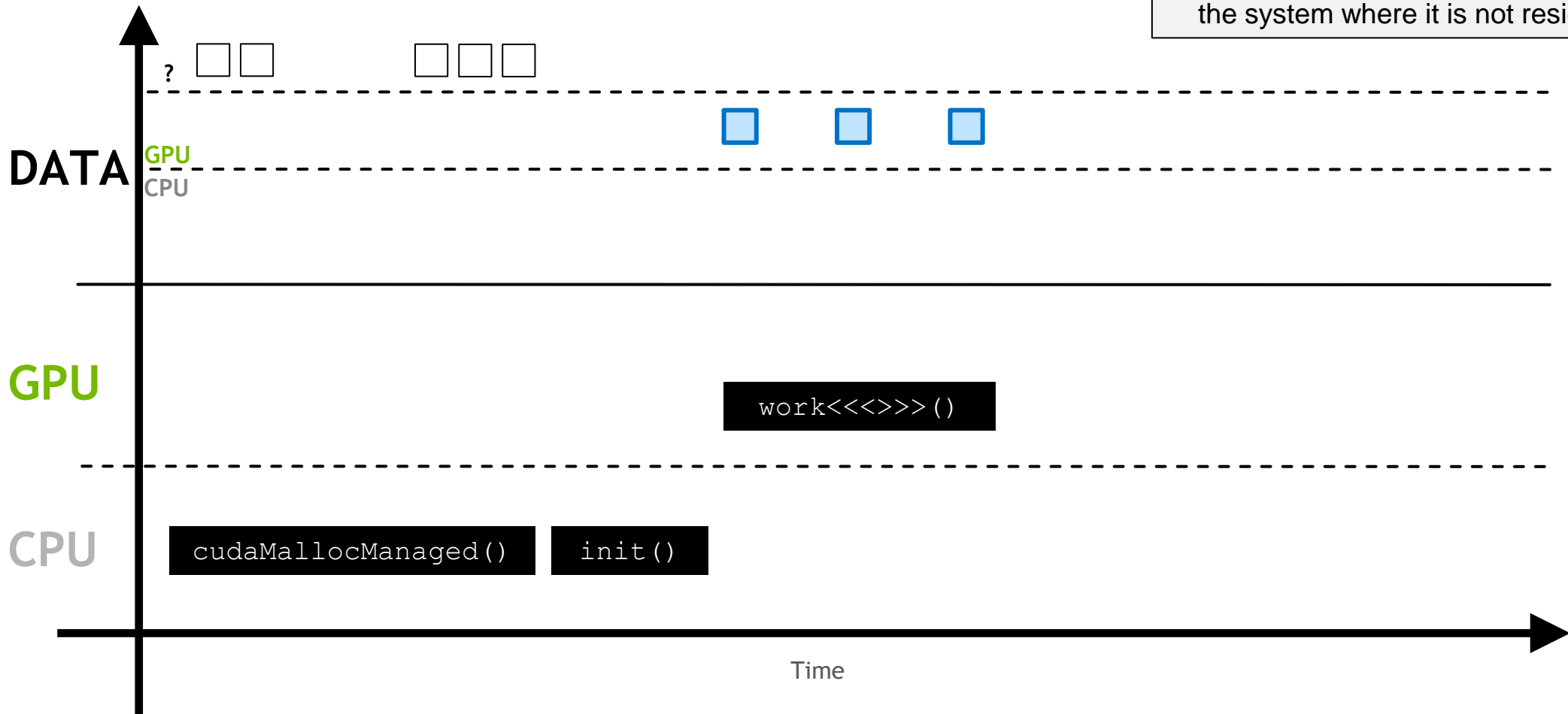
The page fault will trigger the migration of the demanded memory



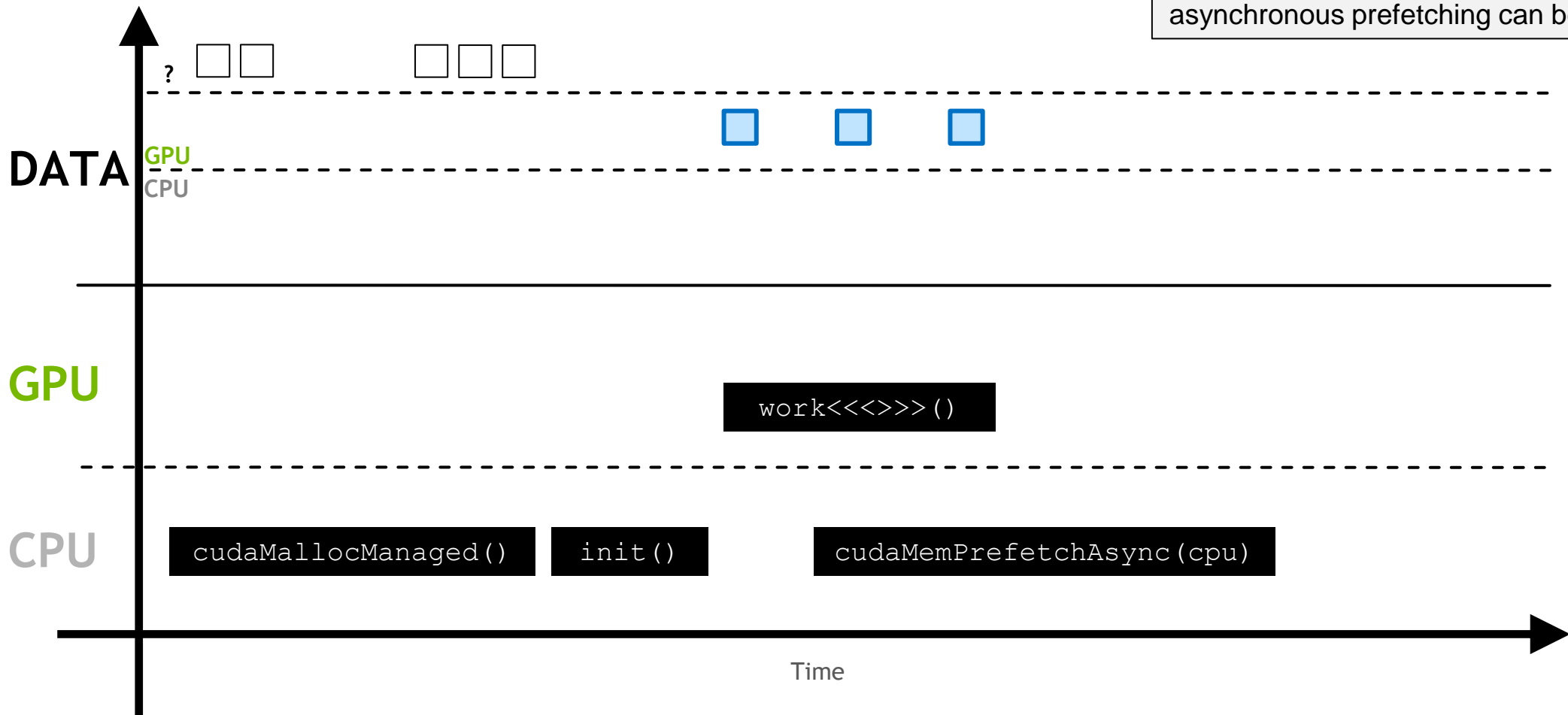
This process repeats anytime the memory is requested somewhere in the system where it is not resident



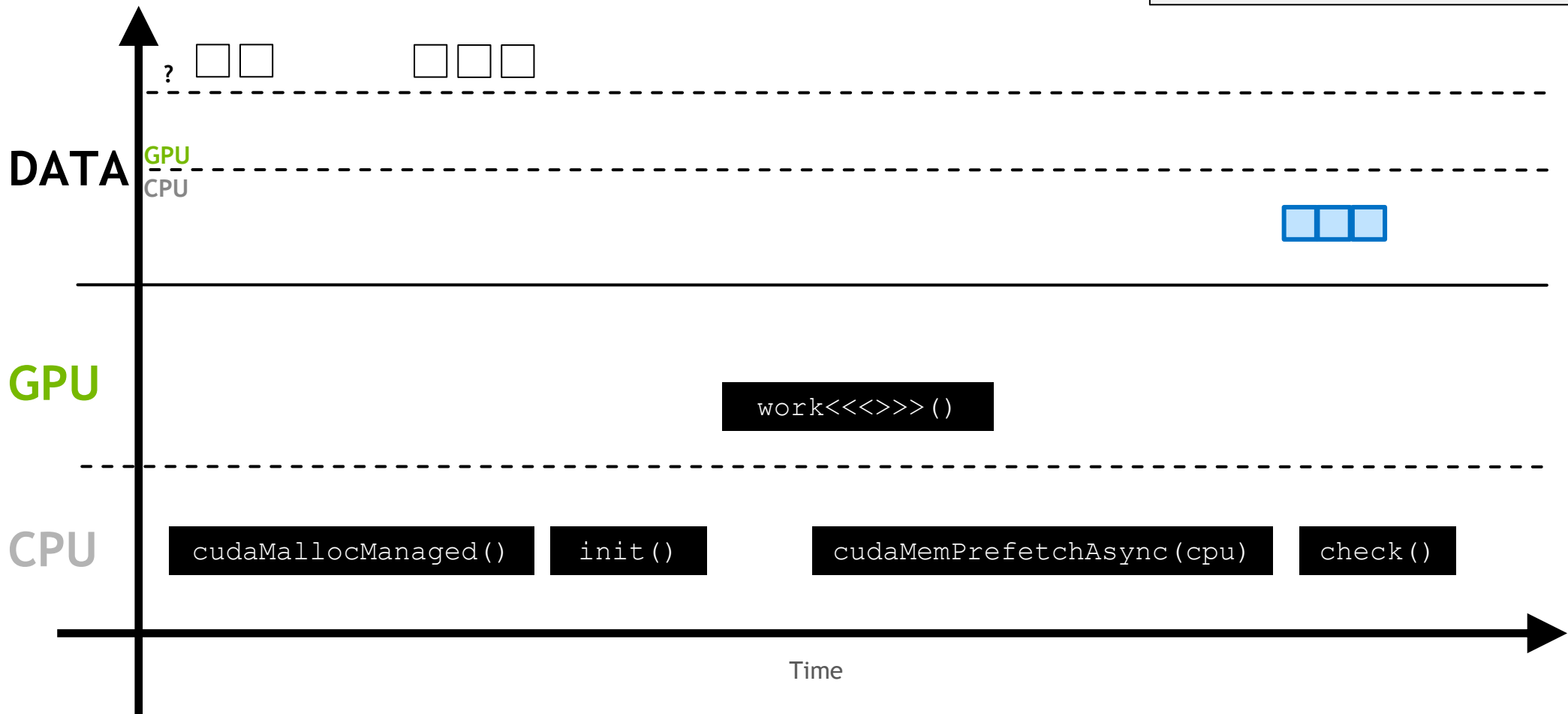
This process repeats anytime the memory is requested somewhere in the system where it is not resident



If it is known that the memory **will be** accessed somewhere it is not resident, asynchronous prefetching can be used

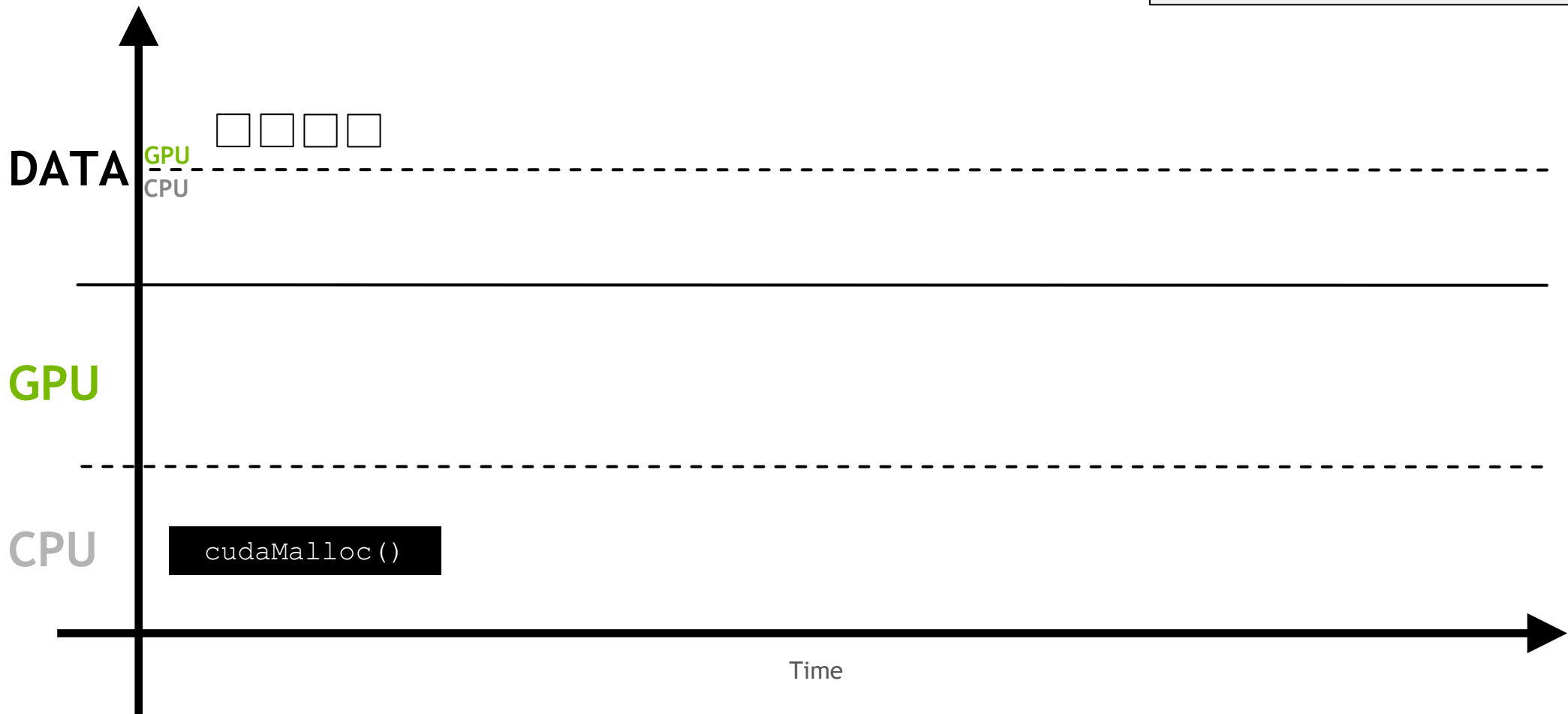


This moves the memory in larger batches, and prevents page faulting

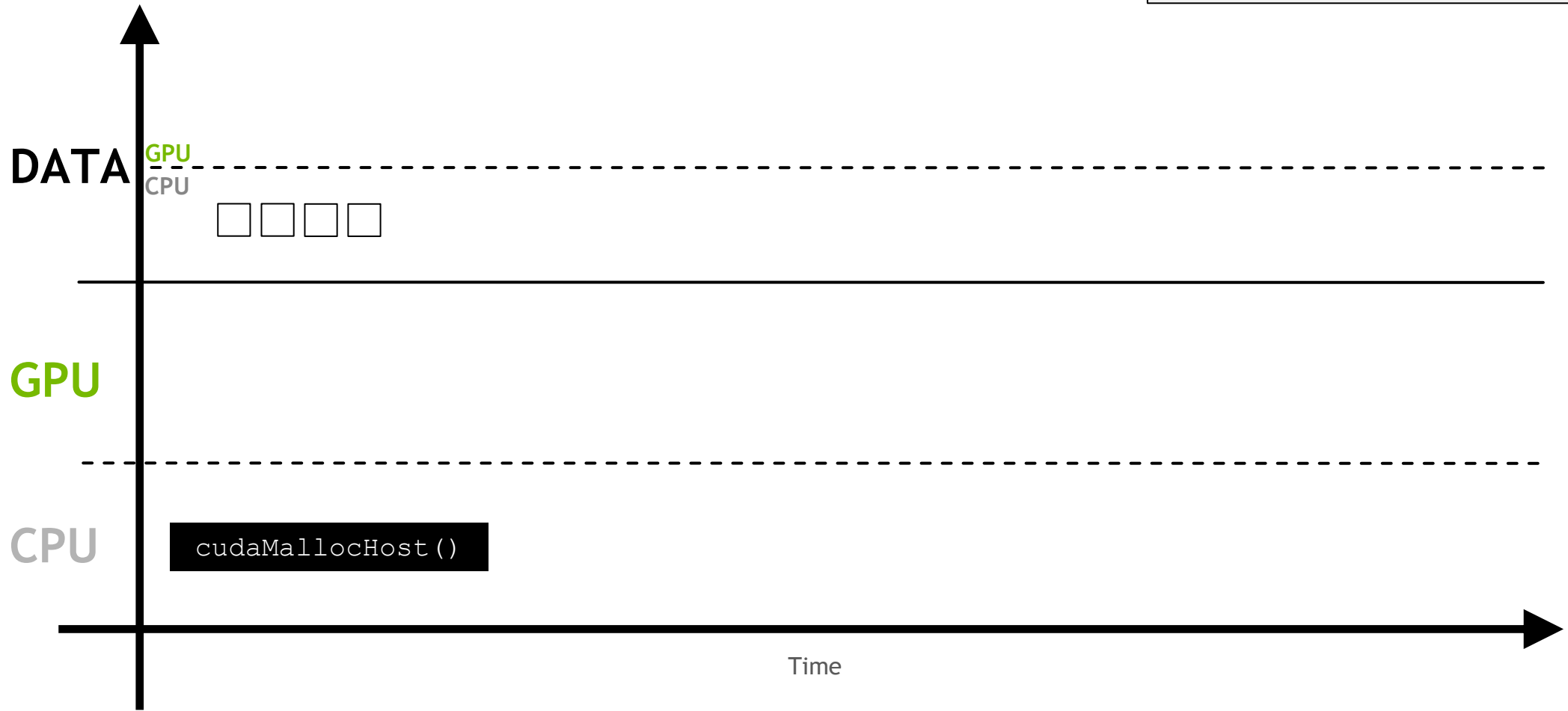


Non-Unified Memory

Memory can be allocated directly to the GPU with `cudaMalloc`

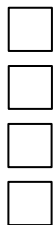


Memory can be allocated directly to the host with `cudaMallocHost`



DATA

GPU
CPU



Memory allocated in either of these ways can be **copied** to other locations in the system with `cudaMemCpy`

GPU

CPU

`cudaMallocHost ()`

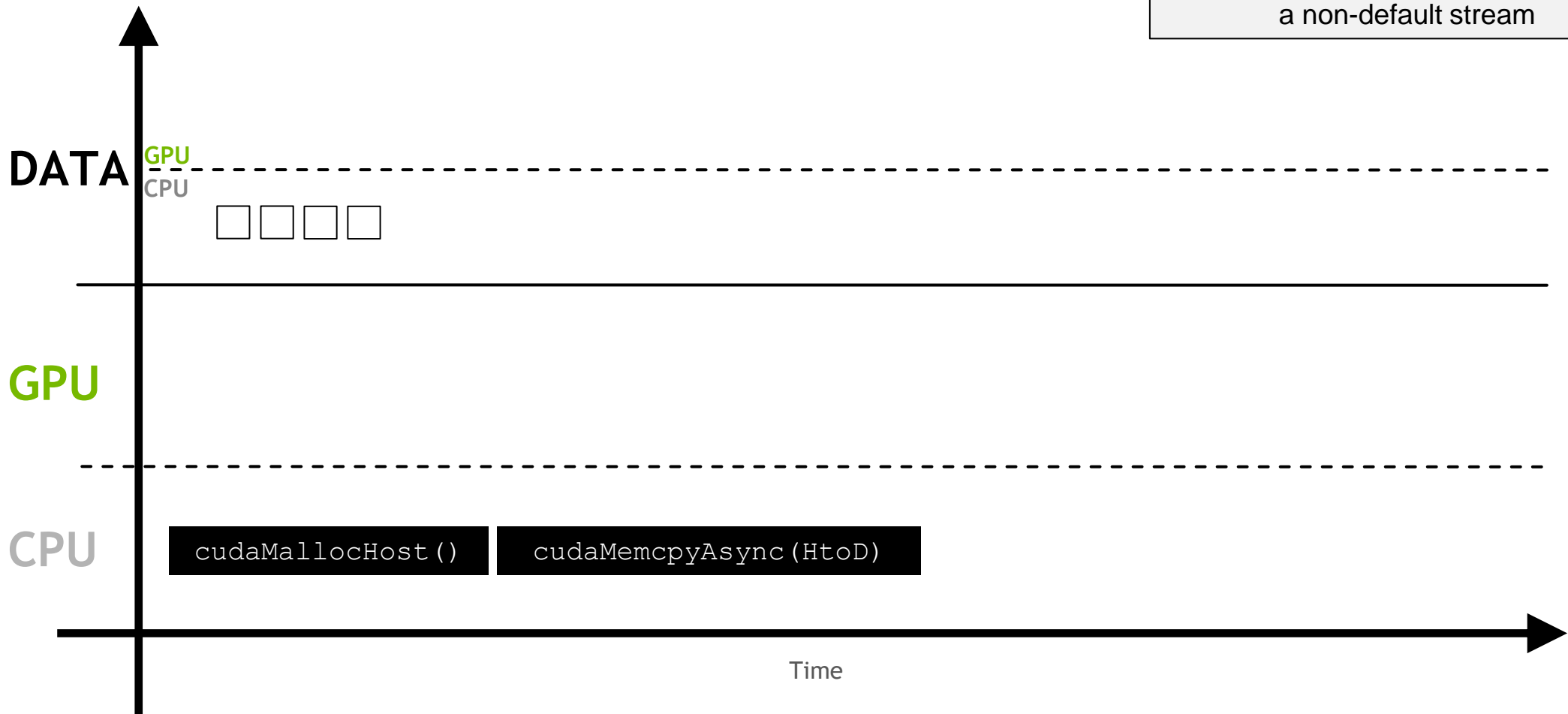
`cudaMemcpy (HtoD)`

Time



cudaMemcpyAsync

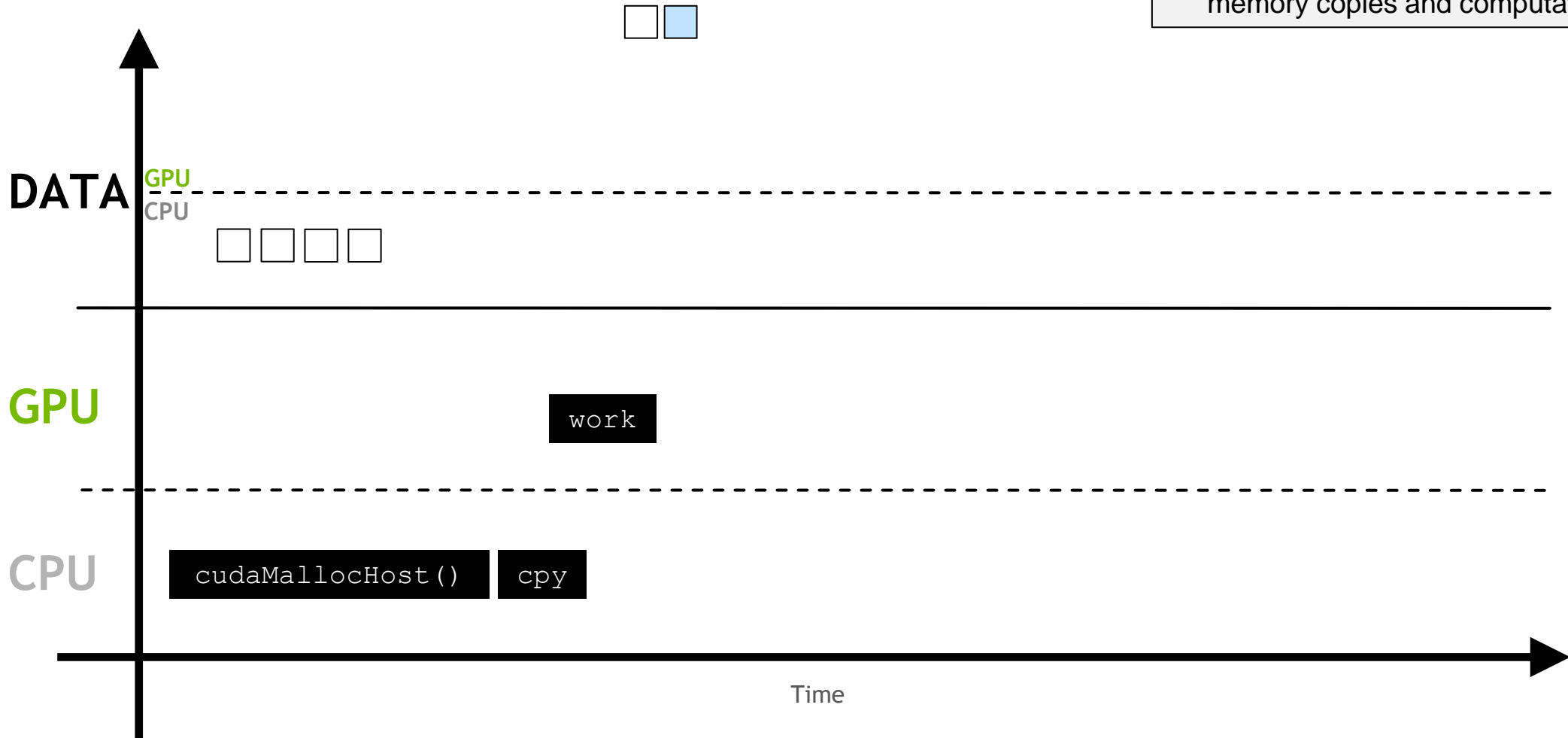
`cudaMemcpyAsync` can asynchronously transfer memory over a non-default stream



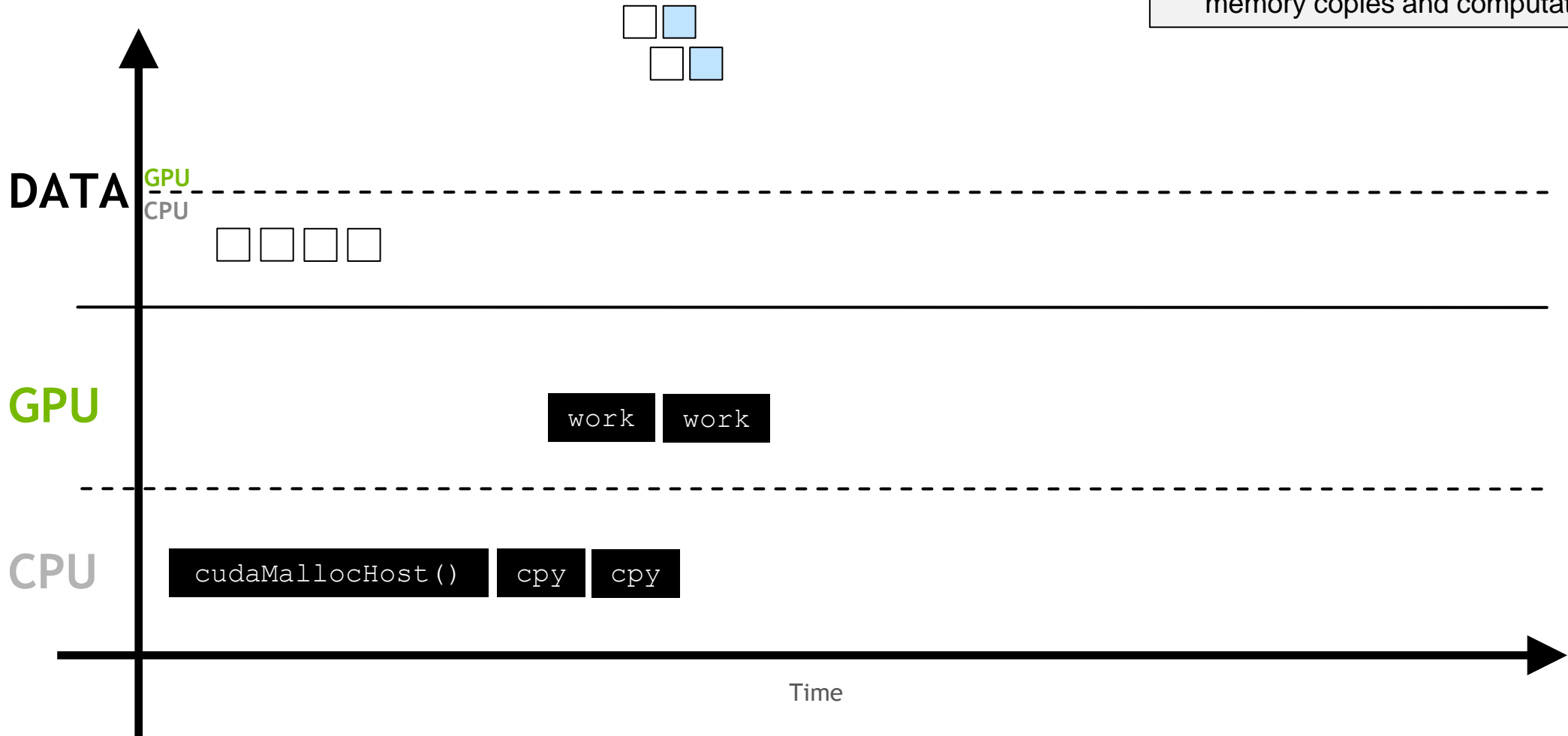
This can allow the **overlapping** memory copies and computation



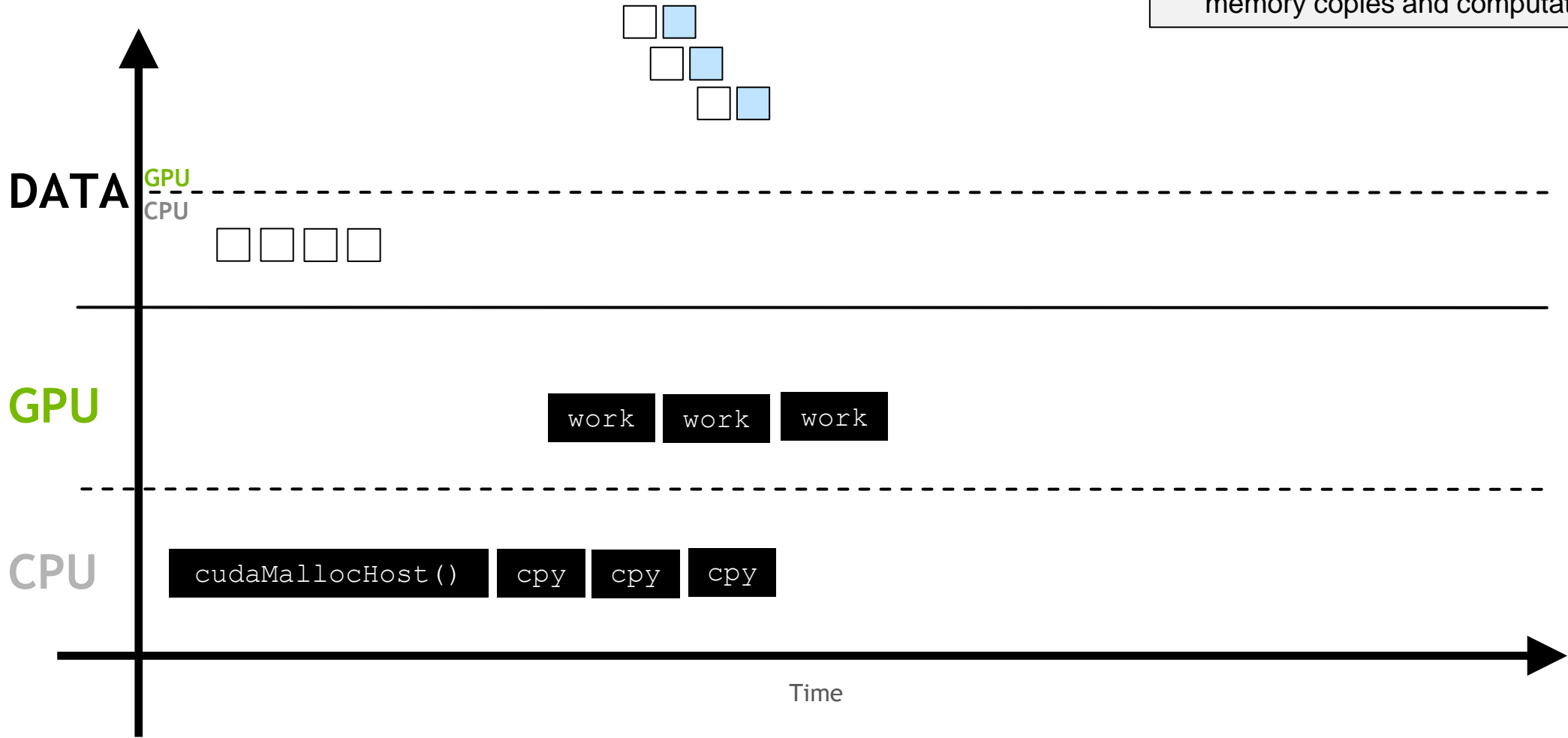
This can allow the **overlapping** memory copies and computation



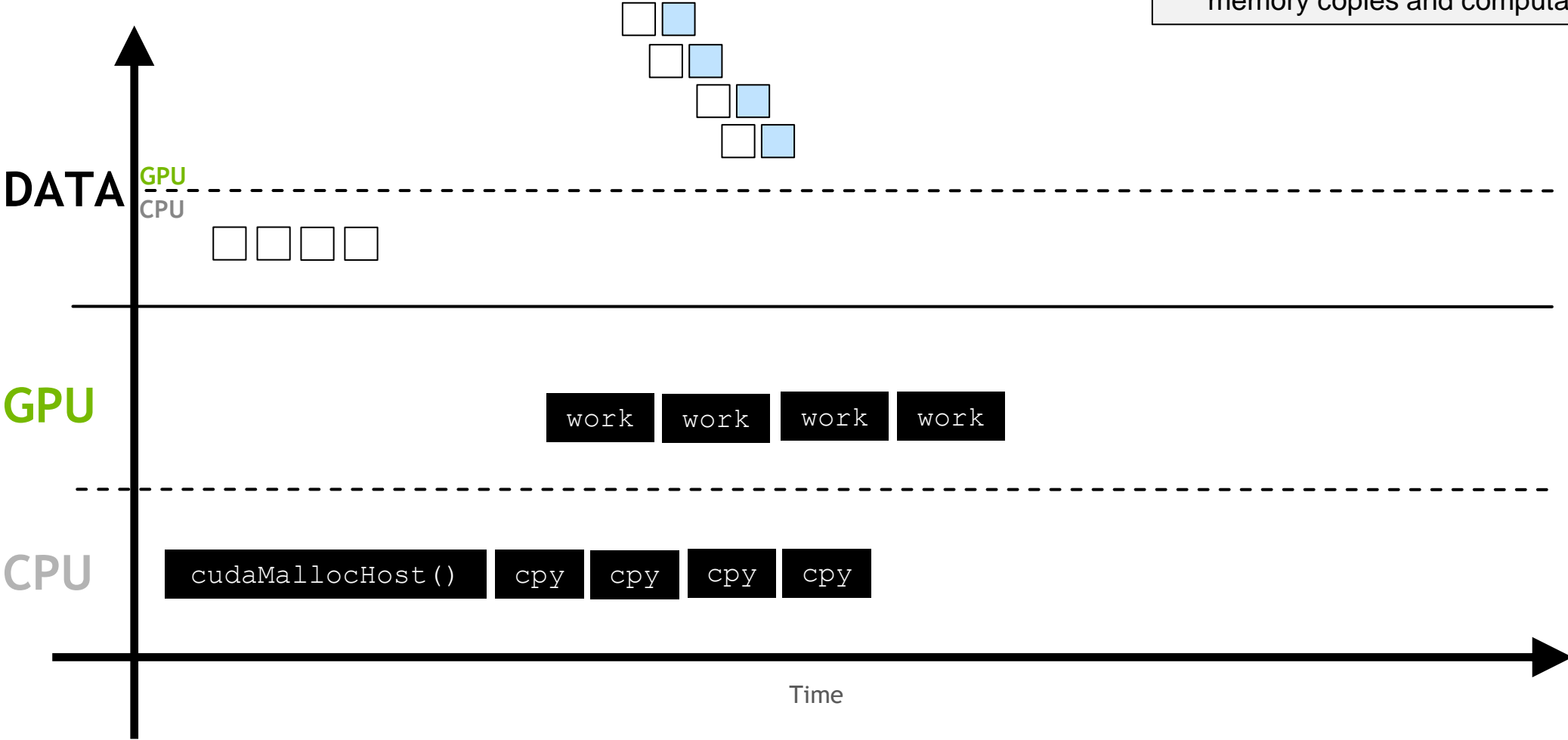
This can allow the **overlapping** memory copies and computation



This can allow the **overlapping** memory copies and computation



This can allow the **overlapping** memory copies and computation



CUDA streams

- **Default stream**
- **Concurrent streams**

Default Stream

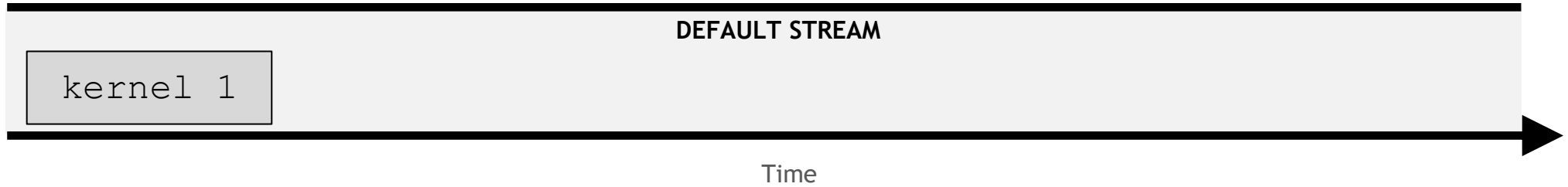
A **stream** is a series of instructions,
and CUDA has a **default stream**



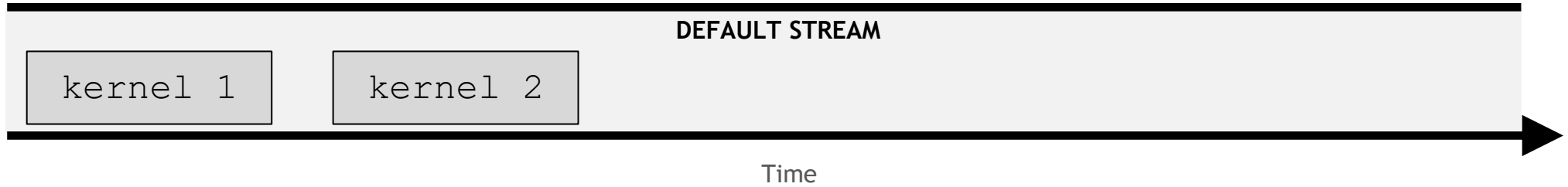
DEFAULT STREAM

Time

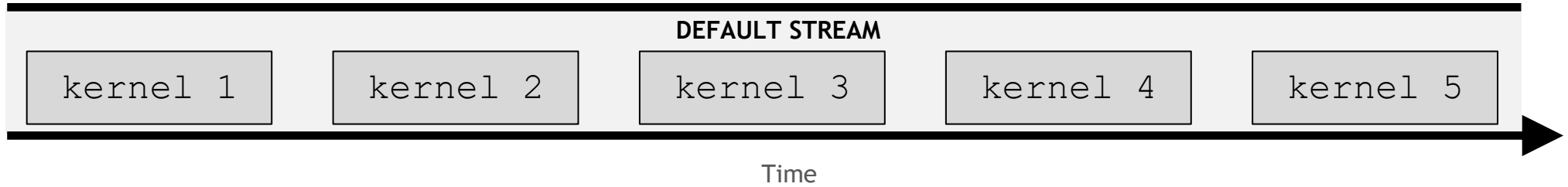
By default, CUDA kernels run in the **default stream**



In any stream, including the default, an instruction in it (here a kernel launch) must complete before the next can begin

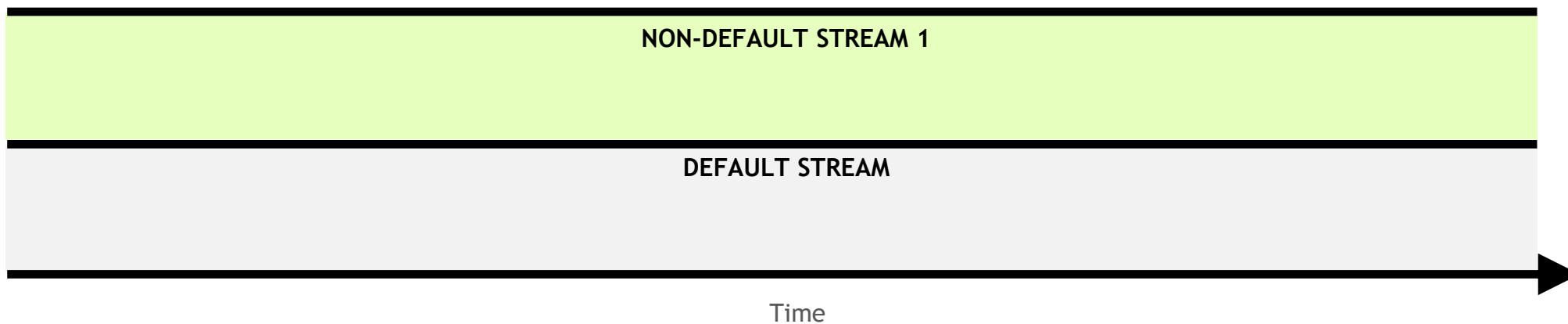


In any stream, including the default, an instruction in it (here a kernel launch) must complete before the next can begin

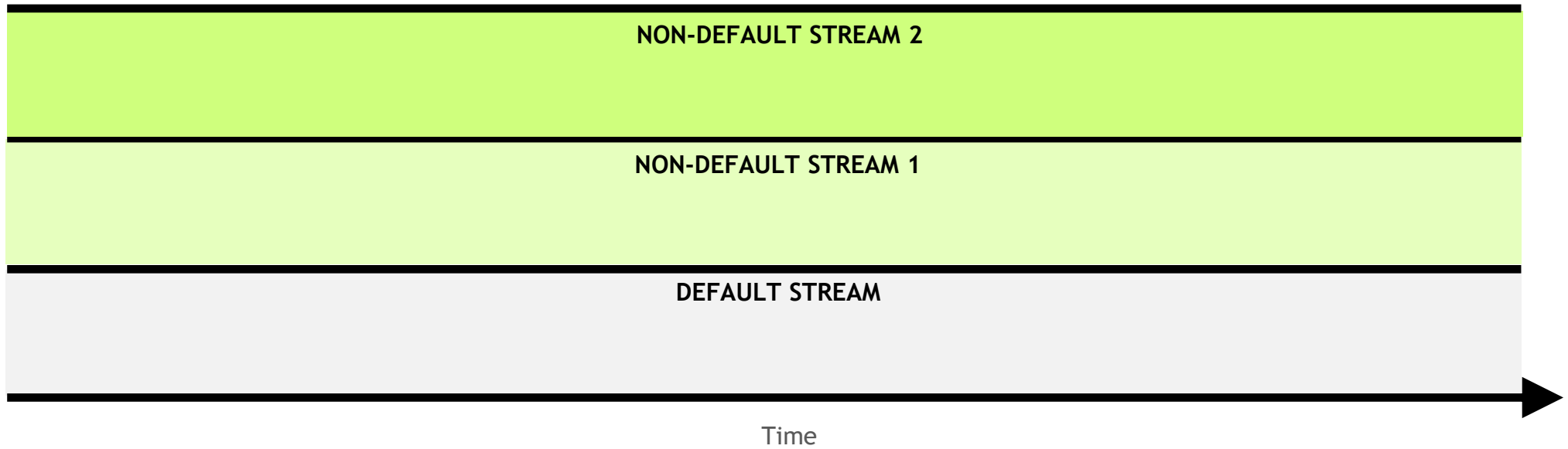


Concurrent streams

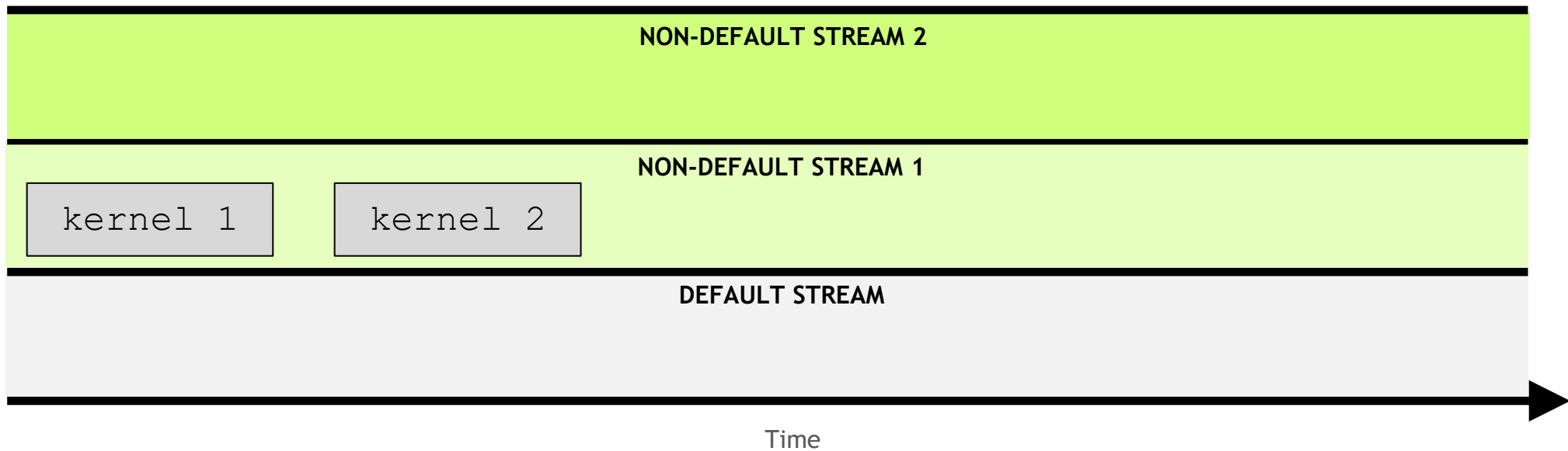
Non-default streams can also be created for kernel execution



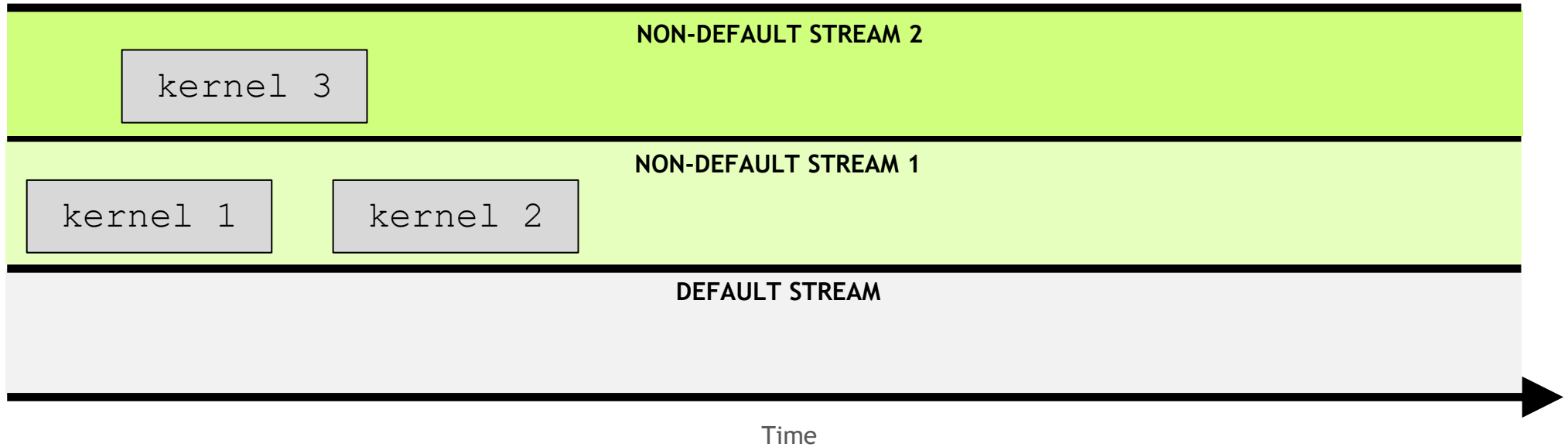
Non-default streams can also be created for kernel execution



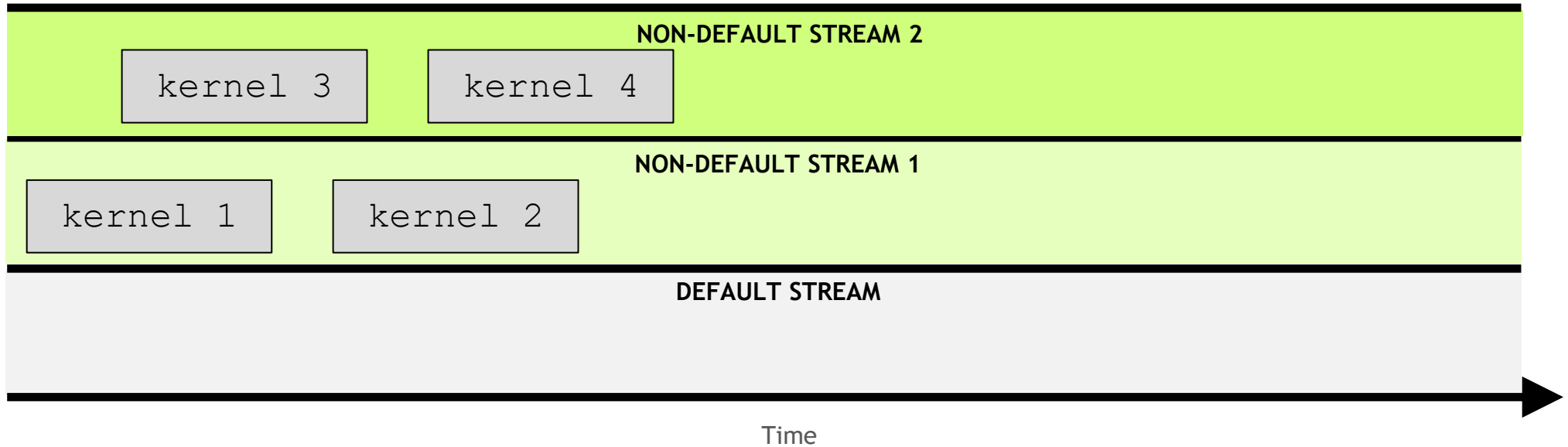
Kernels within any single stream must execute in order



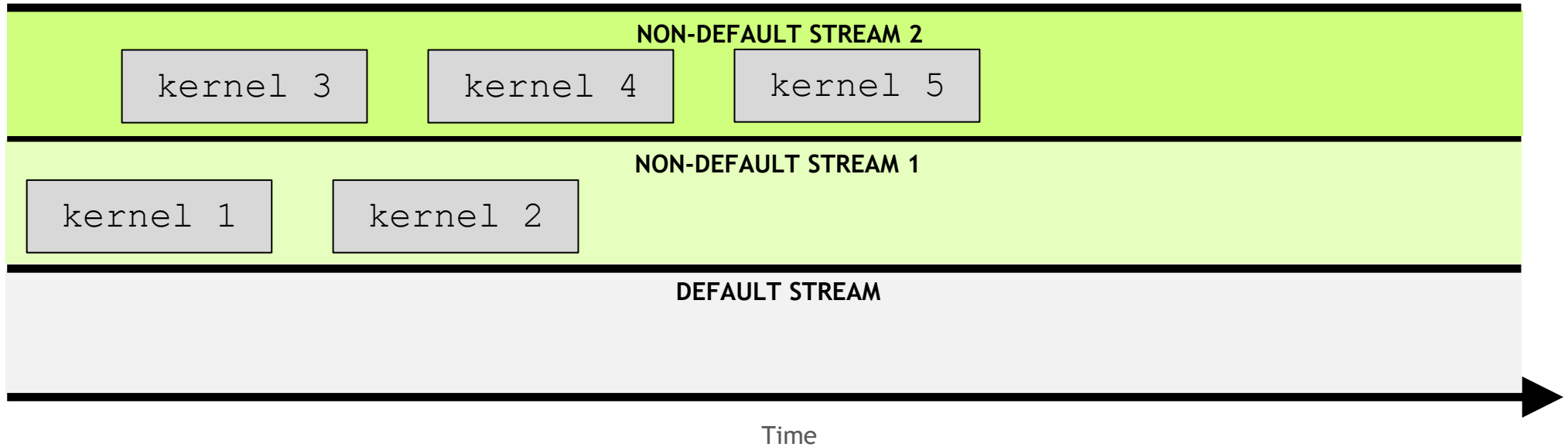
However, kernels in **different, non-default streams**, can interact concurrently



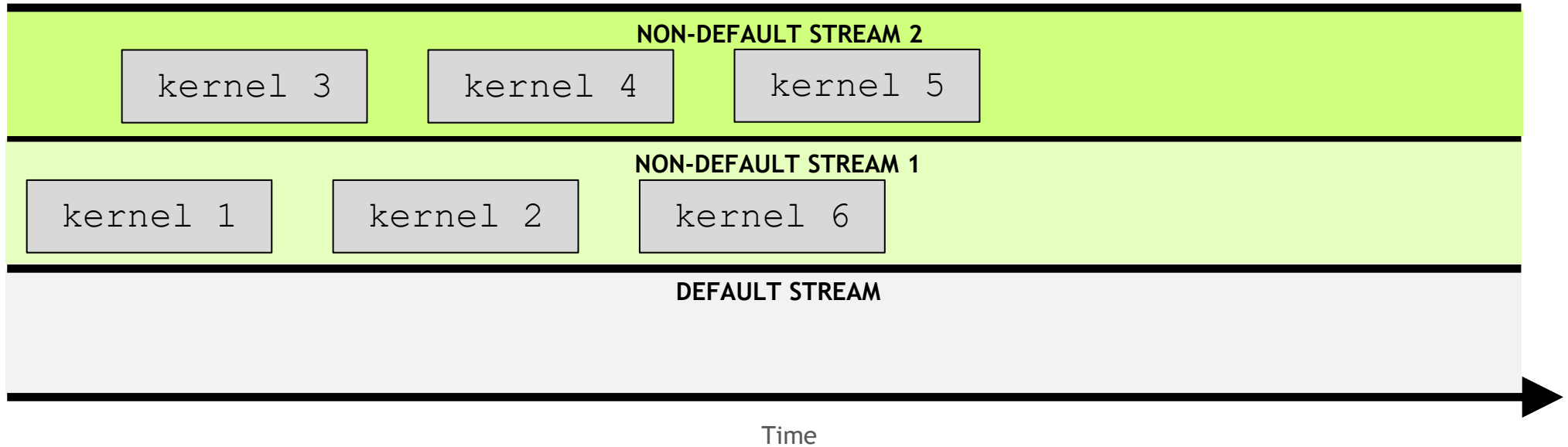
However, kernels in **different, non-default streams**, can interact concurrently



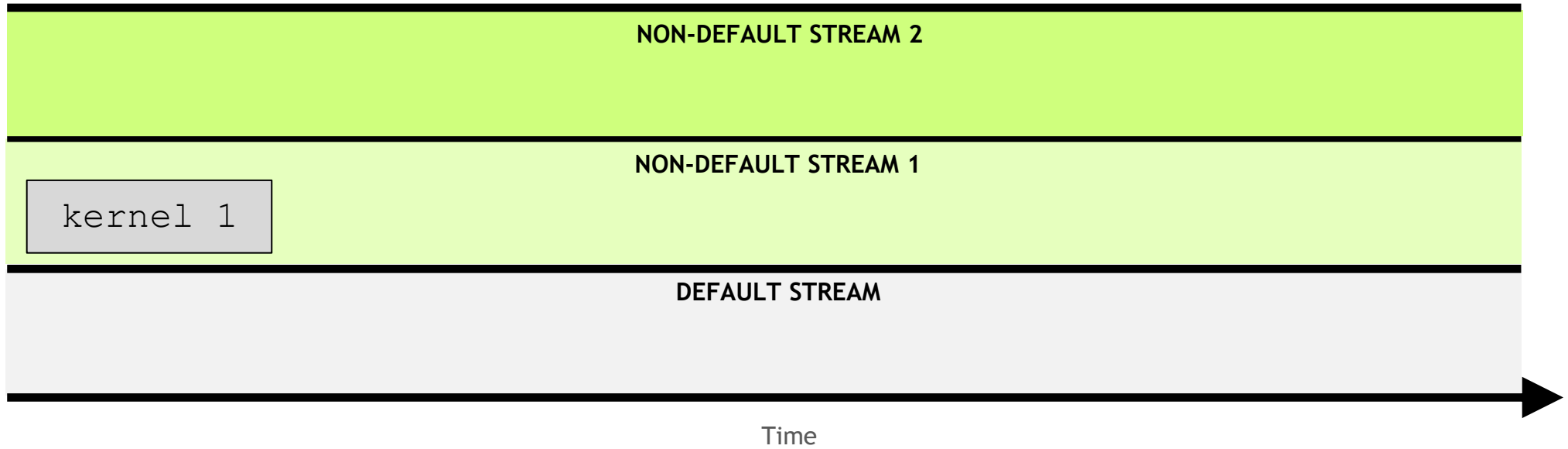
However, kernels in **different, non-default streams**, can interact concurrently



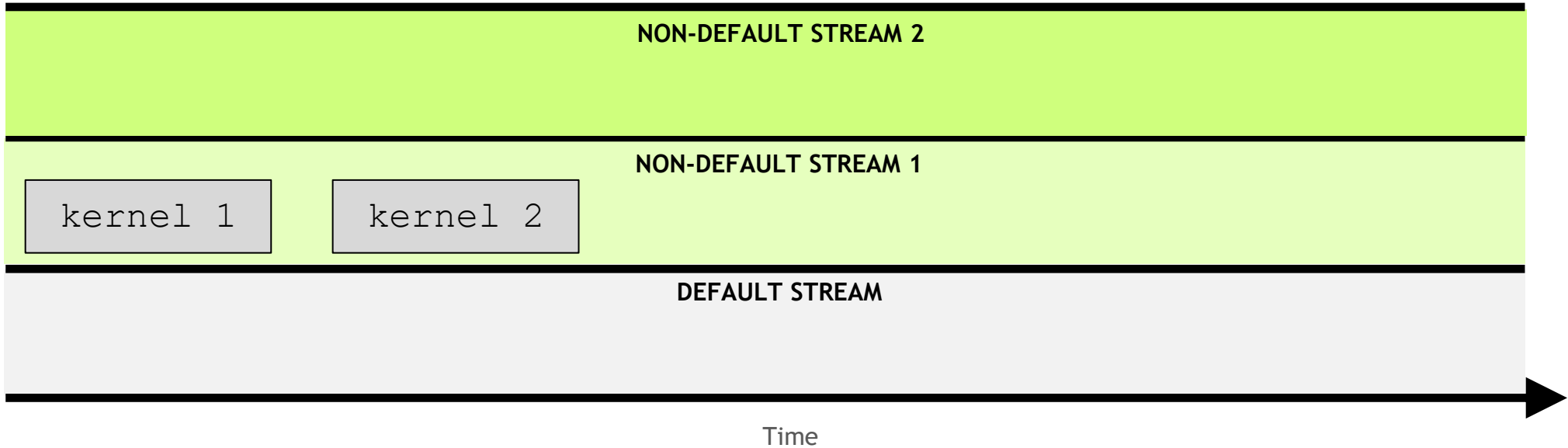
However, kernels in **different, non-default streams**, can interact concurrently



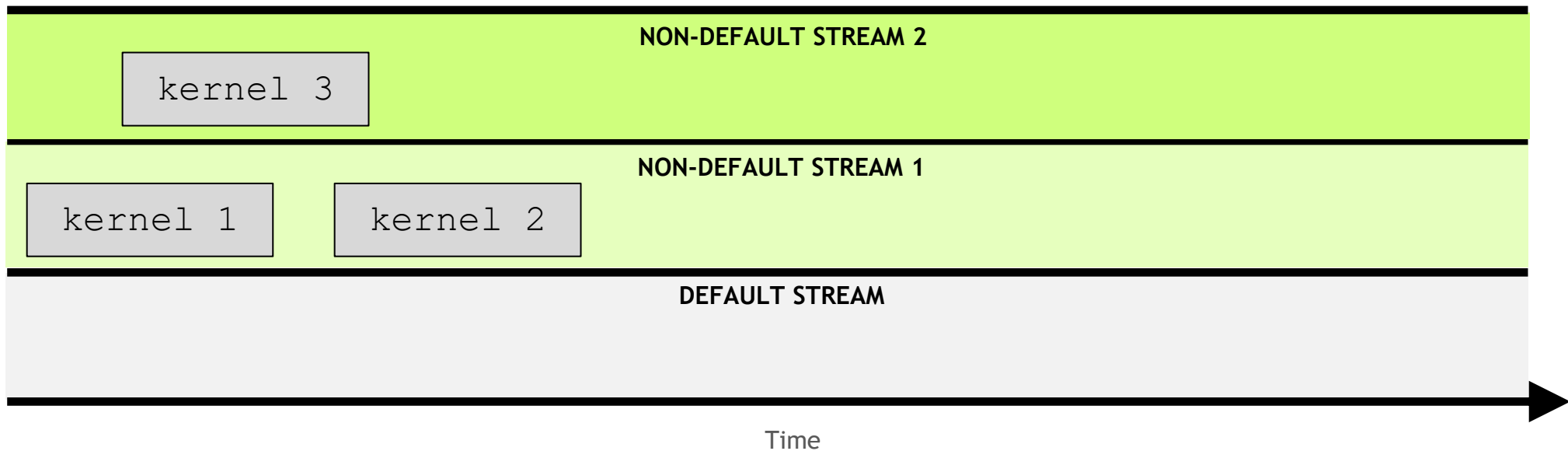
The default stream is special: **it blocks all kernels in all other streams**



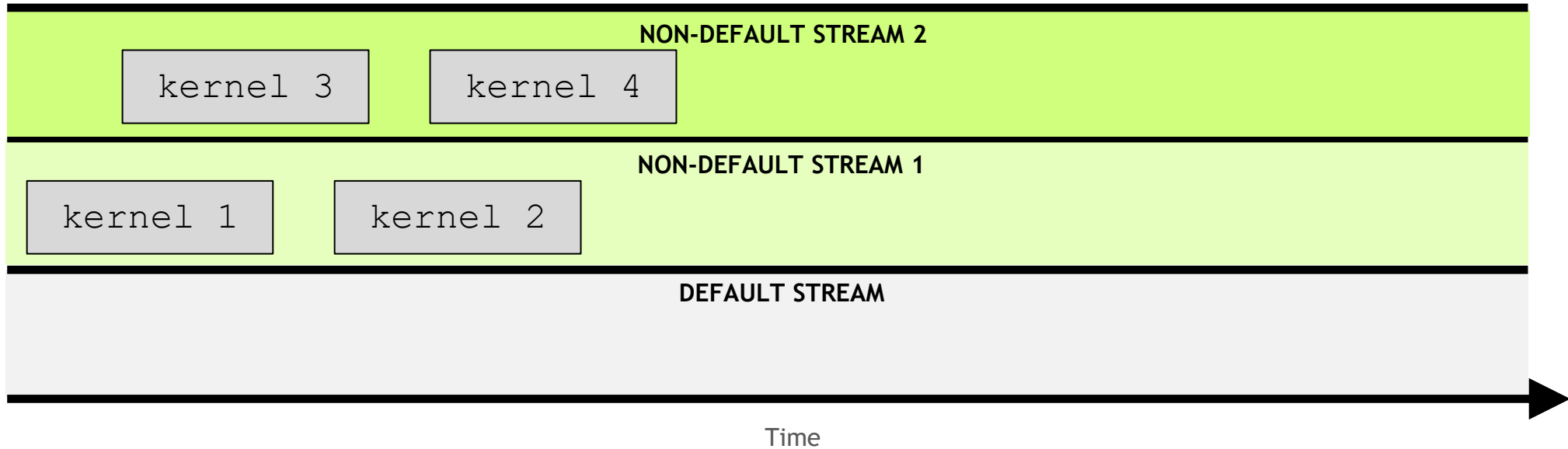
The default stream is special: **it blocks all kernels in all other streams**



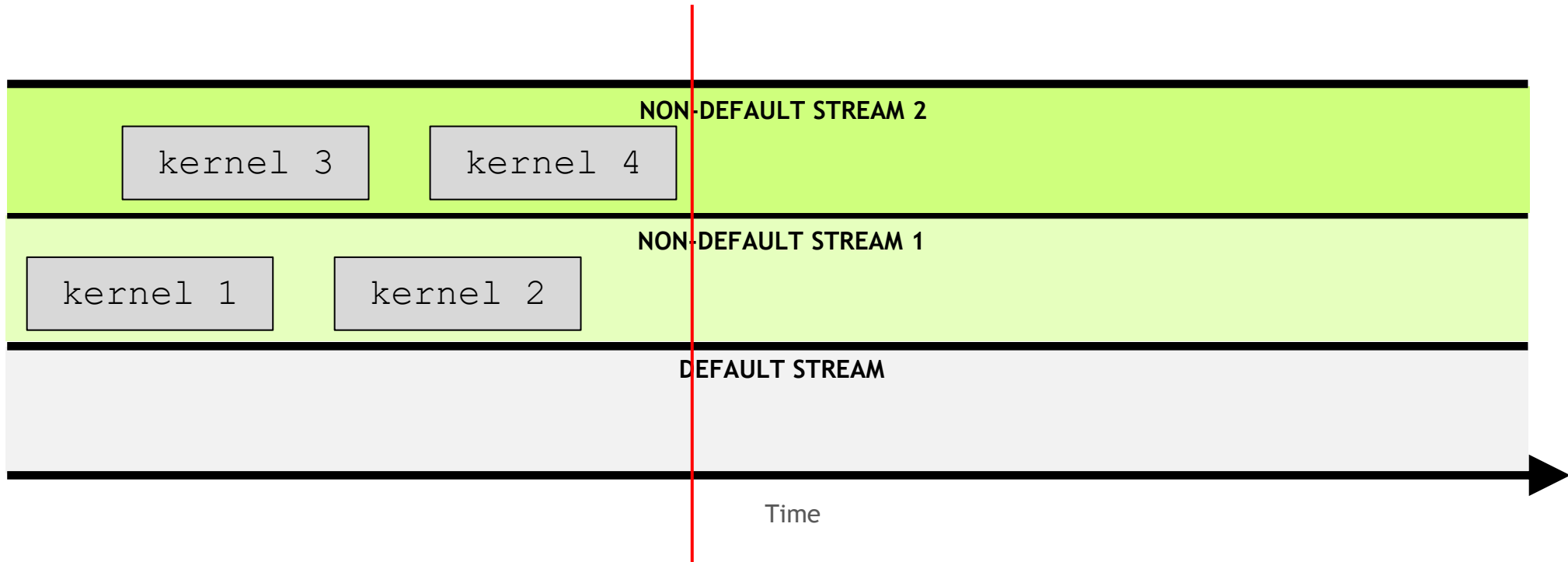
The default stream is special: **it blocks all kernels in all other streams**



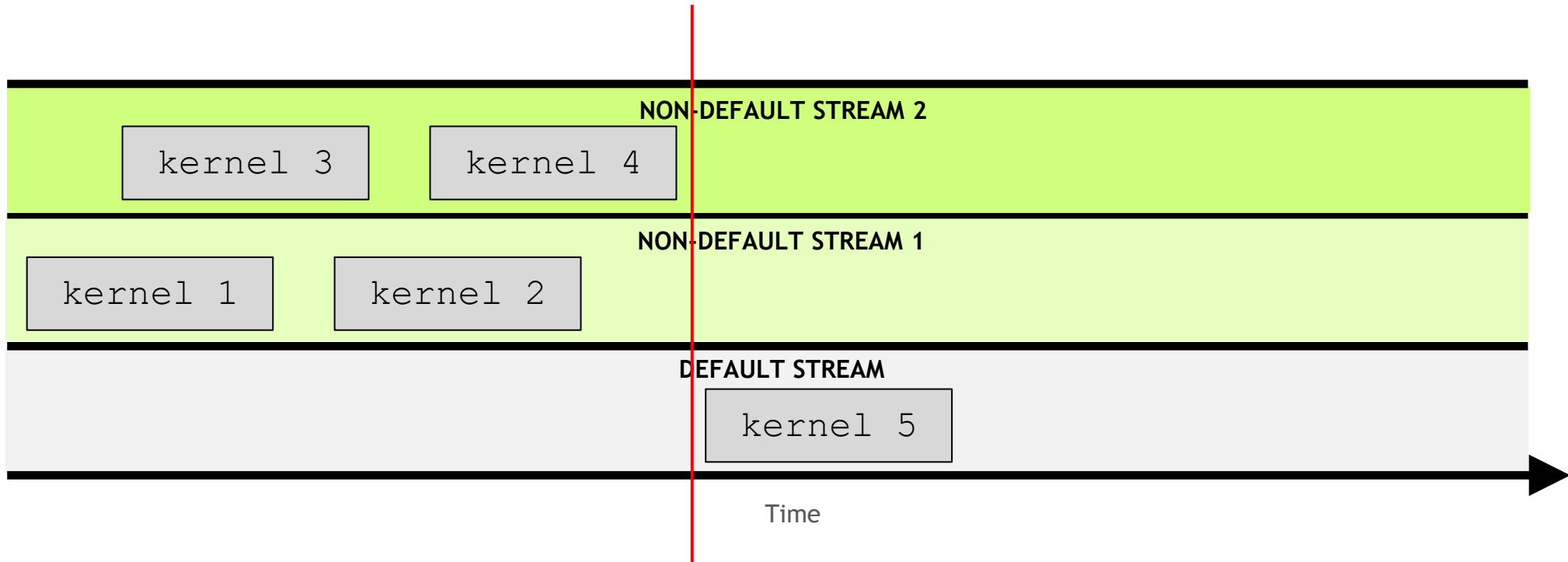
The default stream is special: **it blocks all kernels in all other streams**



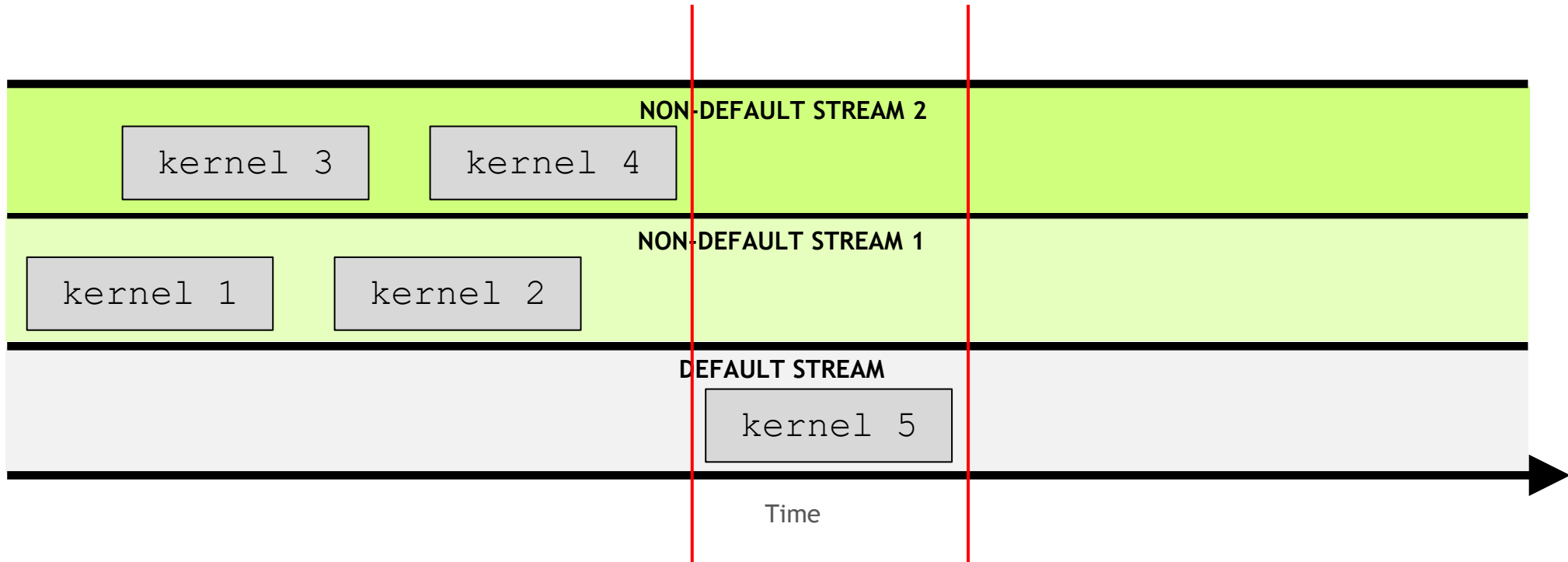
The default stream is special: **it blocks all kernels in all other streams**



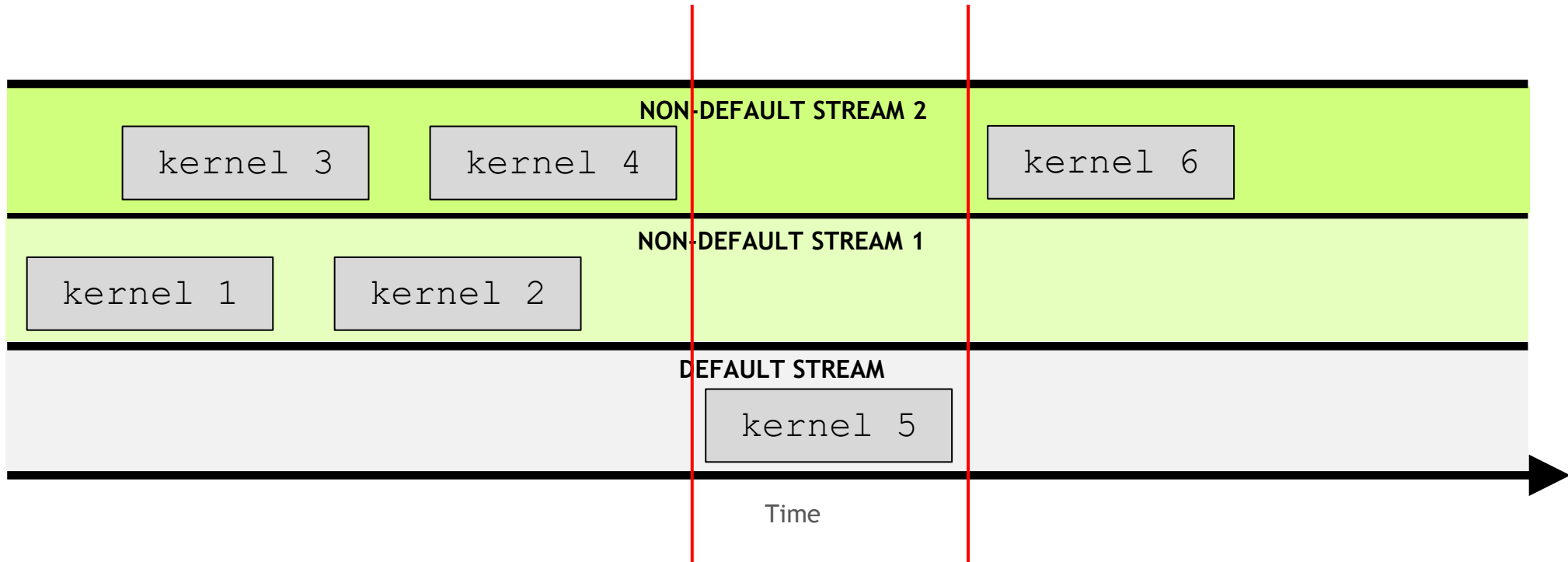
The default stream is special: **it blocks all kernels in all other streams**



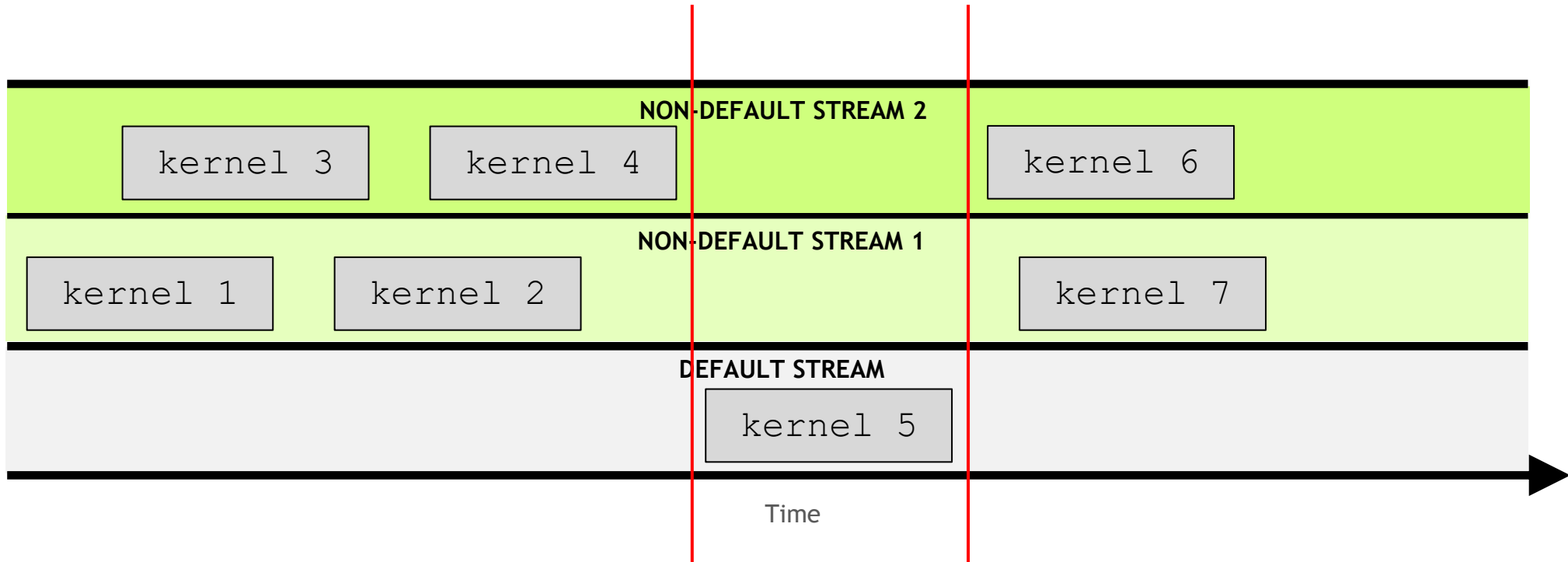
The default stream is special: **it blocks all kernels in all other streams**



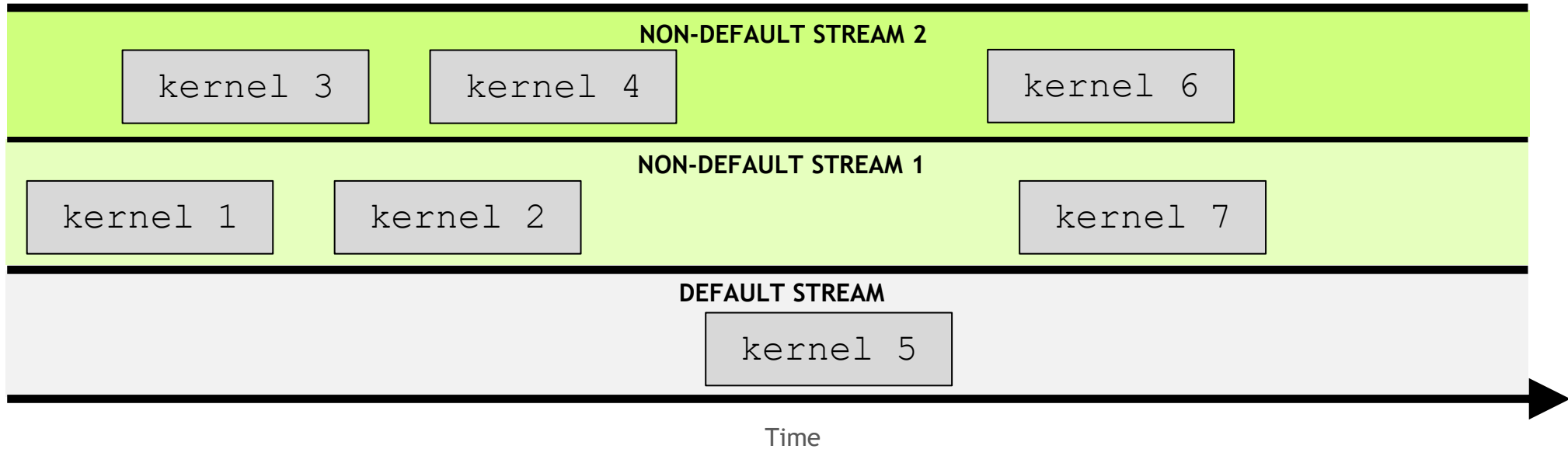
The default stream is special: **it blocks all kernels in all other streams**



The default stream is special: **it blocks all kernels in all other streams**



The default stream is special: **it blocks all kernels in all other streams**



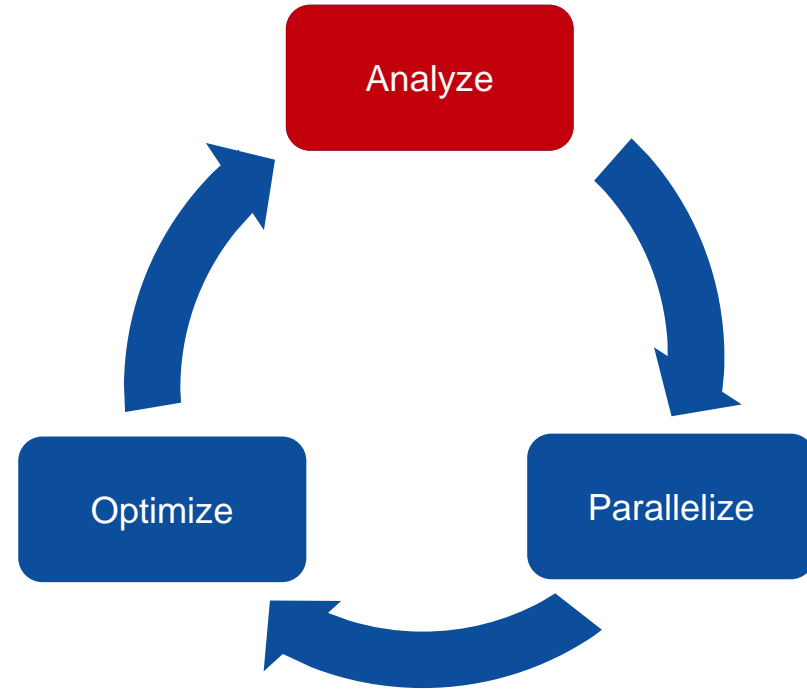
Questions?

Profiler and other tools

- **NSight**
- **Debugger, cuda-memcheck, etc.**

The deployment cycle

- **Analyze** your code to determine most likely places needing parallelization or optimization.
- **Parallelize** your code by starting with the most time consuming parts, check for correctness and then analyze it again.
- **Optimize** your code to improve observed speed-up from parallelization.

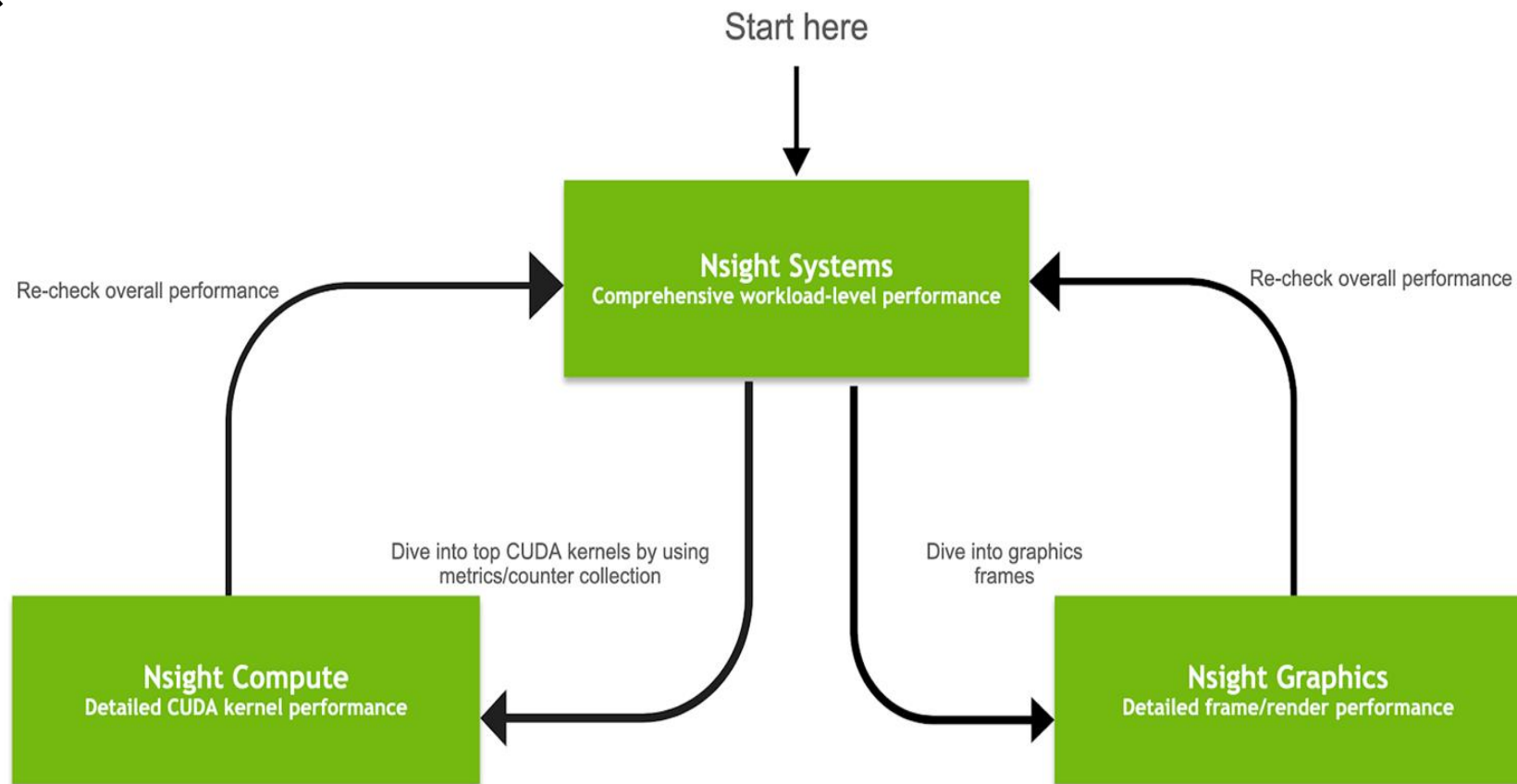


NVidia tools

Nsight Systems - Analyze application algorithm system-wide

Nsight Compute - Debug/optimize CUDA kernel

Nsight Graphics - Debug/optimize graphics workloads



Profiling via command line

Using Command Line Interface (CLI)

NVIDIA Nsight Systems CLI provides

- Simple interface to collect data
- Can be copied to any system and analysed later
- Profiles both serial and parallel code
- For more info enter `nsys --help` on the terminal

To profile a serial application with NVIDIA Nsight Systems, we use NVIDIA Tools Extension (NVTX) API functions in addition to collecting backtraces while sampling.

Profiling using NVTX (I)

NVIDIA Tools Extension API (NVTX) library

What is it?

- A C-based Application Programming Interface (API) for annotating events
- Can be easily integrated to the application
- Can be used with NVIDIA Nsight Systems

Why?

- Allows manual instrumentation of the application
- Allows additional information for profiling (e.g: tracing of CPU events and time ranges)

How?

- Import the header only C library `nvToolsExt.h`
- Wrap the code region or a specific function with `nvtxRangePush()` and `nvtxRangePop()`

Profiling using NVTX (II) SEQUENTIAL

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include "laplace2d.h"
#include <nvtx3/nvToolsExt.h>

int main(int argc, char** argv)
{
    const int n = 4096;
    const int m = 4096;
    const int iter_max = 1000;

    const double tol = 1.0e-6;
    double error = 1.0;

    double *restrict A = (double*)malloc(sizeof(double)*n*m);
    double *restrict Anew = (double*)malloc(sizeof(double)*n*m);

    nvtxRangePushA("init");
    initialize(A, Anew, m, n);
    nvtxRangePop();

    printf("Jacobi relaxation Calculation: %d x %d mesh\n", n, m);

    double st = omp_get_wtime();
    int iter = 0;

    nvtxRangePushA("while");
    while ( error > tol && iter < iter_max )
    {
        nvtxRangePushA("calc");
        error = calcNext(A, Anew, m, n);
        nvtxRangePop();

        nvtxRangePushA("swap");
        swap(A, Anew, m, n);
        nvtxRangePop();

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

        iter++;
    }
    nvtxRangePop();

    double runtime = omp_get_wtime() - st;
    printf(" total: %f s\n", runtime);

    deallocate(A, Anew);

    return 0;
}
```

jacobi.c
(starting and ending of ranges are highlighted with the same color)

- t Selects the APIs to be traced (nvtx in this example)
- status if true, generates summary of statistics after the collection
- b Selects the backtrace method to use while sampling. The option dwarf uses DWARF's CFI (Call Frame Information).
- force-overwrite if true, overwrites the existing results
- o sets the output (qdrep) filename

```
moszhgank@prmr-dgx-32:~/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel/nsys profile -t nvtx --stats=true -b dwarf --force-overwrite true -o laplace-seq ./laplace-seq
Collecting data...
Jacobi relaxation Calculation: 4096 x 4096 mesh
 0, 0.250000
100, 0.002397
200, 0.001204
300, 0.000804
400, 0.000603
500, 0.000483
600, 0.000403
700, 0.000345
800, 0.000302
900, 0.000269
total: 55.754501 s
Processing events...
Capturing symbol files...
Saving intermediate "/home/mozhgank/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel/laplace-seq.qdstrm" file to disk...
Importing [=====100%]
Saved report file to "/home/mozhgank/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel/laplace-seq.qdrep"
Exporting 70802 events: [=====100%]
Exported successfully to
/home/mozhgank/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel/laplace-seq.sqlite
Generating NVTX Push-Pop Range Statistics...
NVTX Push-Pop Range Statistics (nanoseconds)
```

Time (%)	Total Time	Instances	Average	Minimum	Maximum	Range
49.9	55754497966	1	55754497966.0	55754497966	55754497966	while
26.5	29577817696	1000	29577817.7	29092956	65008545	calc
23.4	26163892482	1000	26163892.5	25761418	60129514	swap
0.1	137489808	1	137489808.0	137489808	137489808	init

NVTX range statistics

"calc" region (calcNext function) takes 26.6%
"swap" region (swap function) takes 23.4% of
total execution time

Open laplace-seq.qdrep with
Nsight System GUI to view the
timeline

Profiling using NVTX (III) PARALLEL

```
#include <math.h>
#include <stdlib.h>

#define OFFSET(x, y, m) ((x)*(m)) + (y)

void initialize(double *restrict A, double *restrict Anew, int m, int n)
{
    memset(A, 0, n * m * sizeof(double));
    memset(Anew, 0, n * m * sizeof(double));

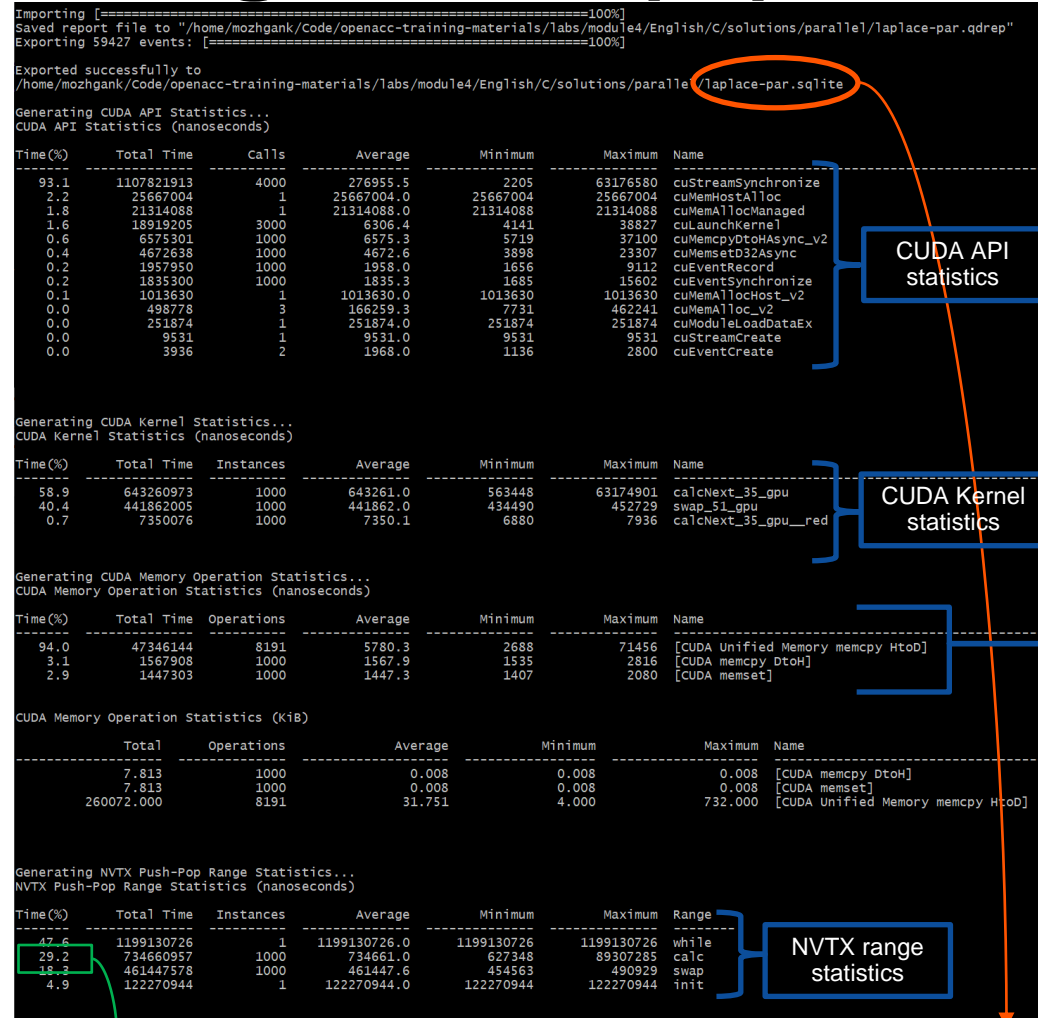
    for(int i = 0; i < m; i++){
        A[i] = 1.0;
        Anew[i] = 1.0;
    }
}

double calcNext(double *restrict A, double *restrict Anew, int m, int n)
{
    double error = 0.0;
    #pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++)
    {
        #pragma acc loop
        for( int i = 1; i < m-1; i++ )
        {
            Anew[OFFSET(j, i, m)] = 0.25 * ( A[OFFSET(j, i+1, m)] + A[OFFSET(j, i-1, m)]
            + A[OFFSET(j-1, i, m)] + A[OFFSET(j+1, i, m)]);
            error = max( error, fabs(Anew[OFFSET(j, i, m)] - A[OFFSET(j, i, m)]));
        }
    }
    return error;
}

void swap(double *restrict A, double *restrict Anew, int m, int n)
{
    #pragma acc parallel loop
    for( int j = 1; j < n-1; j++)
    {
        #pragma acc loop
        for( int i = 1; i < m-1; i++ )
        {
            A[OFFSET(j, i, m)] = Anew[OFFSET(j, i, m)];
        }
    }
}

void deallocate(double *restrict A, double *restrict Anew)
{
    free(A);
    free(Anew);
}
```

laplace2d.c
(Parallellised using OpenACC parallel directives (pragmas highlighted))



“calc” region (calcNext function) takes 29.2%
“swap” region (swap function) takes 18.3% of
total execution time

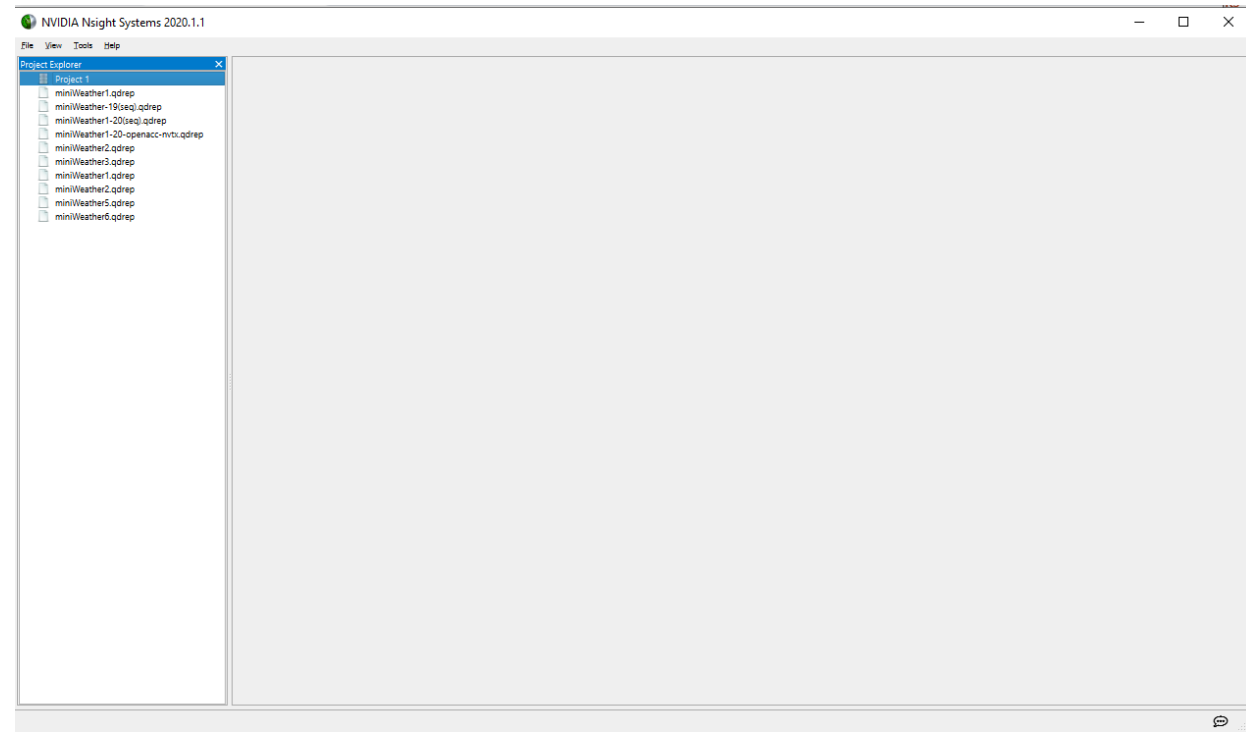
Open laplace-par.qdrep
with Nsight System GUI to
view the timeline

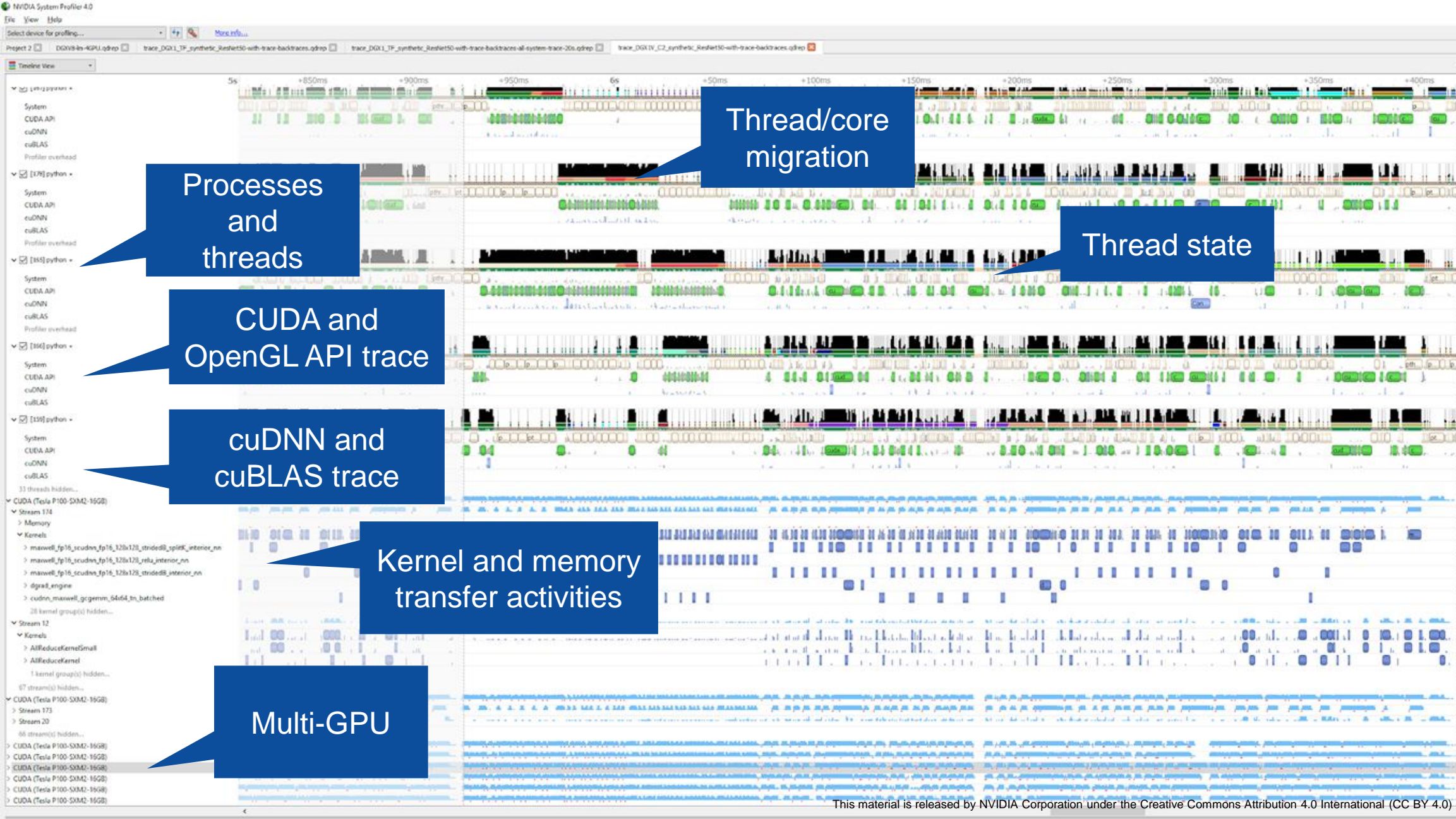
Profiling using Nsight

Using Nsight Systems

Open the generated report files (*.qdrep) from command line in the Nsight Systems profiler.

File > Open





Processes and threads

Thread/core migration

Thread state

CUDA and OpenGL API trace

cuDNN and cuBLAS trace

Kernel and memory transfer activities

Multi-GPU

Other tools

If you can, do not debug via print!!!

You could use instead
Visual Studio Code with Nsight



Arm Forge Debugger (formerly Alinea DDT)

Provides application developers with a single tool that can debug hybrid MPI, OpenMP, CUDA and OpenACC applications on a single workstation or GPU cluster.



CUDA-GDB

Delivers a seamless debugging experience that allows you to debug both the CPU and GPU portions of your application simultaneously. Use CUDA-GDB on Linux or MacOS, from the command line, DDD or EMACS.



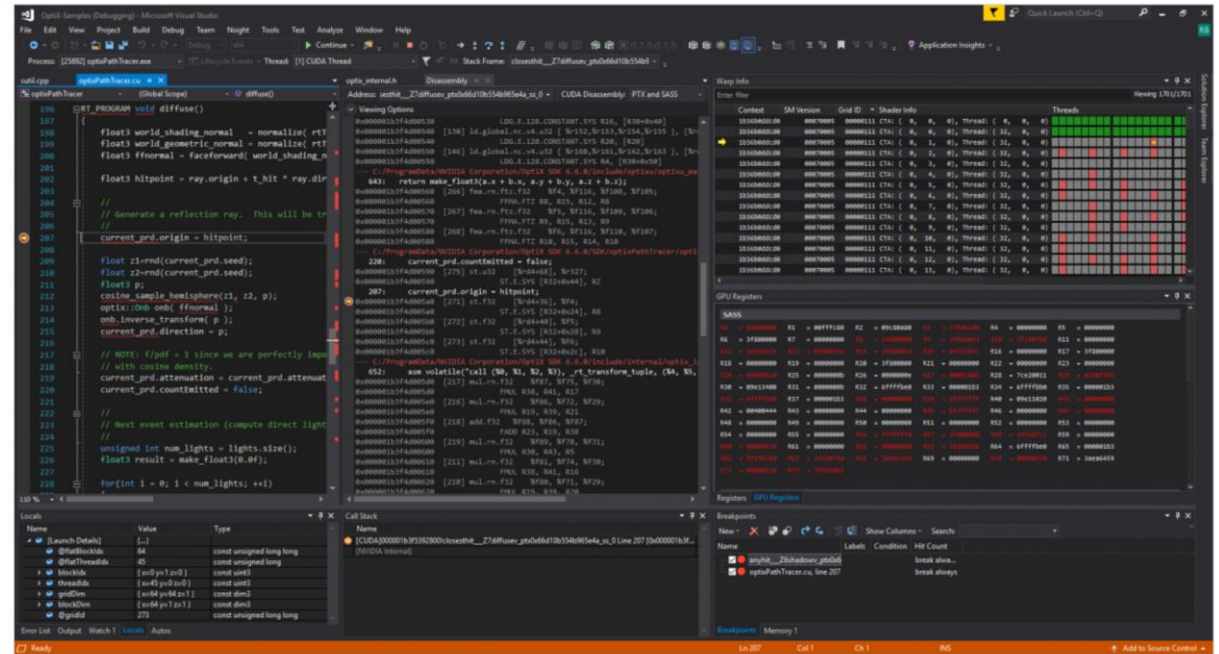
TotalView

A GUI-based tool that allows you to debug one or many processes/threads with complete control over program execution, from basic debugging operations like stepping through code to concurrent programs that take advantage of threads, OpenMP, MPI, or GPUs.



COMPUTE SANITIZER

Compute Sanitizer is a functional correctness checking suite included in the CUDA toolkit. This suite contains multiple tools that can perform different type of checks. The memcheck tool is capable of precisely detecting and attributing out of bounds and misaligned memory access errors in CUDA applications. The tool can

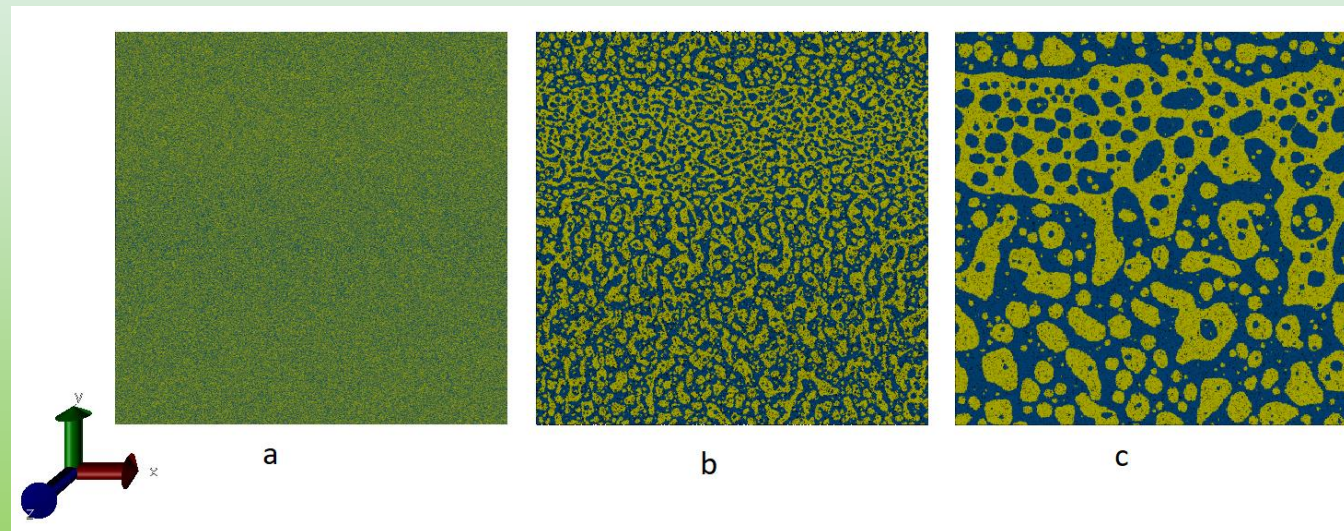


<https://developer.nvidia.com/nsight-visual-studio-code-edition>

Examples

- **Porting DL_MESO to CUDA**
- **DL_MESO on multi-GPU**

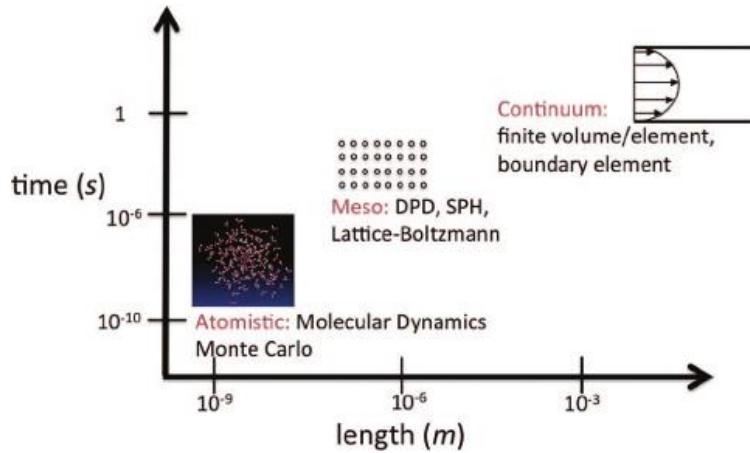
An Hartree application: porting DL_MESO to CUDA



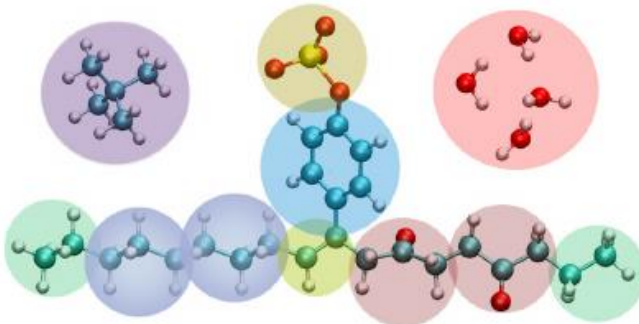
Porting DL_MESO to GPUs

What is DPD and DL_MESO?

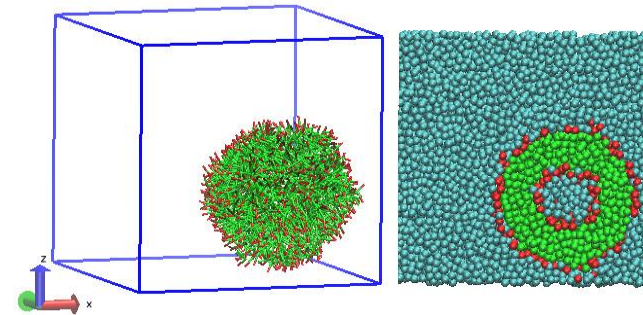
Dissipative Particle Dynamics (DPD)...



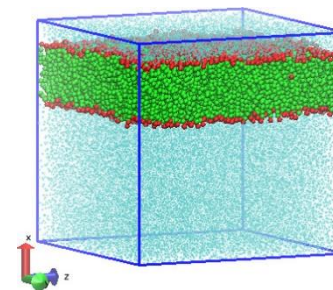
...coarse grain representation using beads



DL_MESO: software package developed by
Michael Seaton at DL



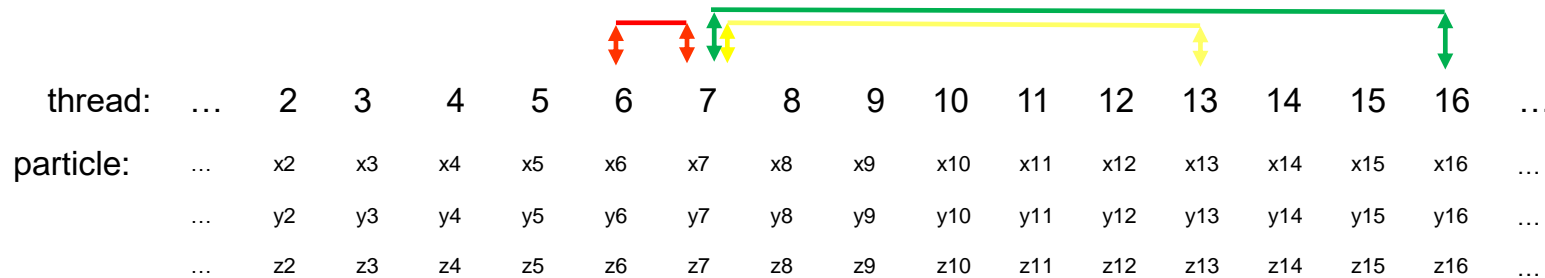
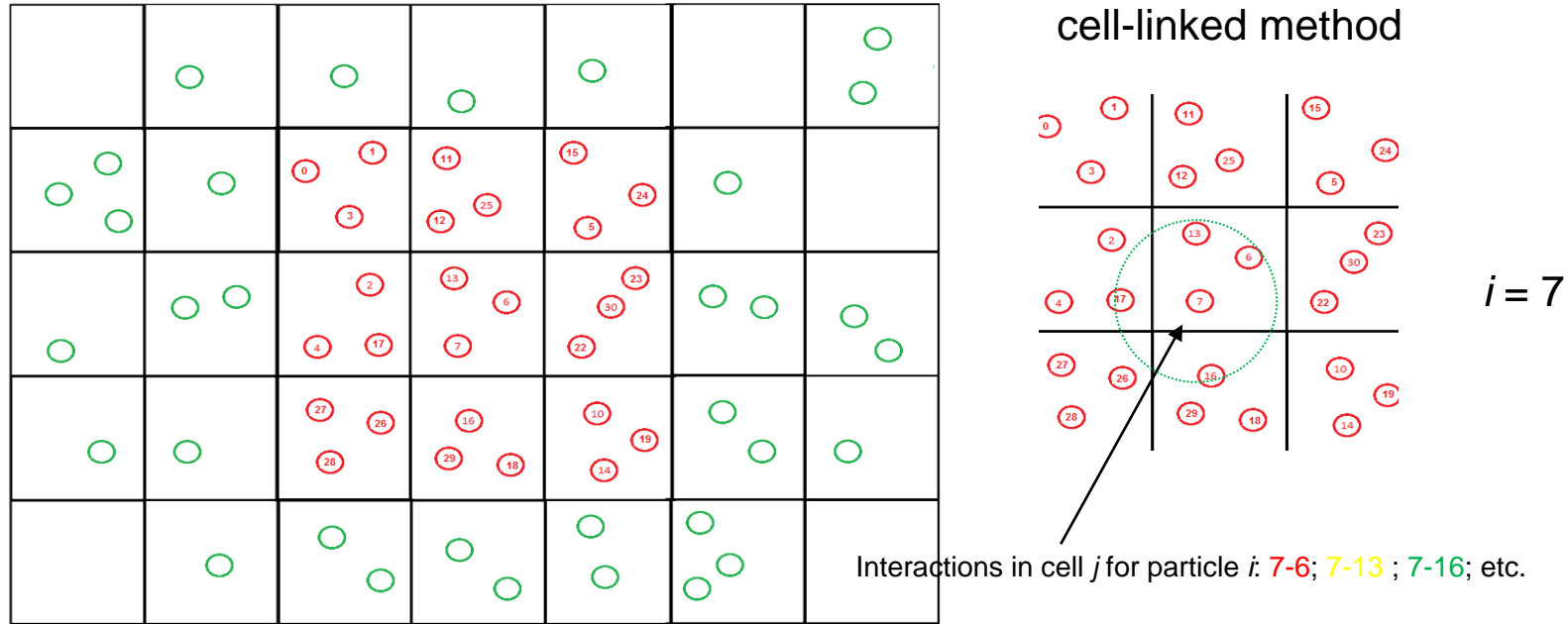
Vesicle Formation



Lipid Bilayer

DL_MESO: highly scalable mesoscale simulations
Molecular Simulation 39 (10) pp. 796-821, 2013

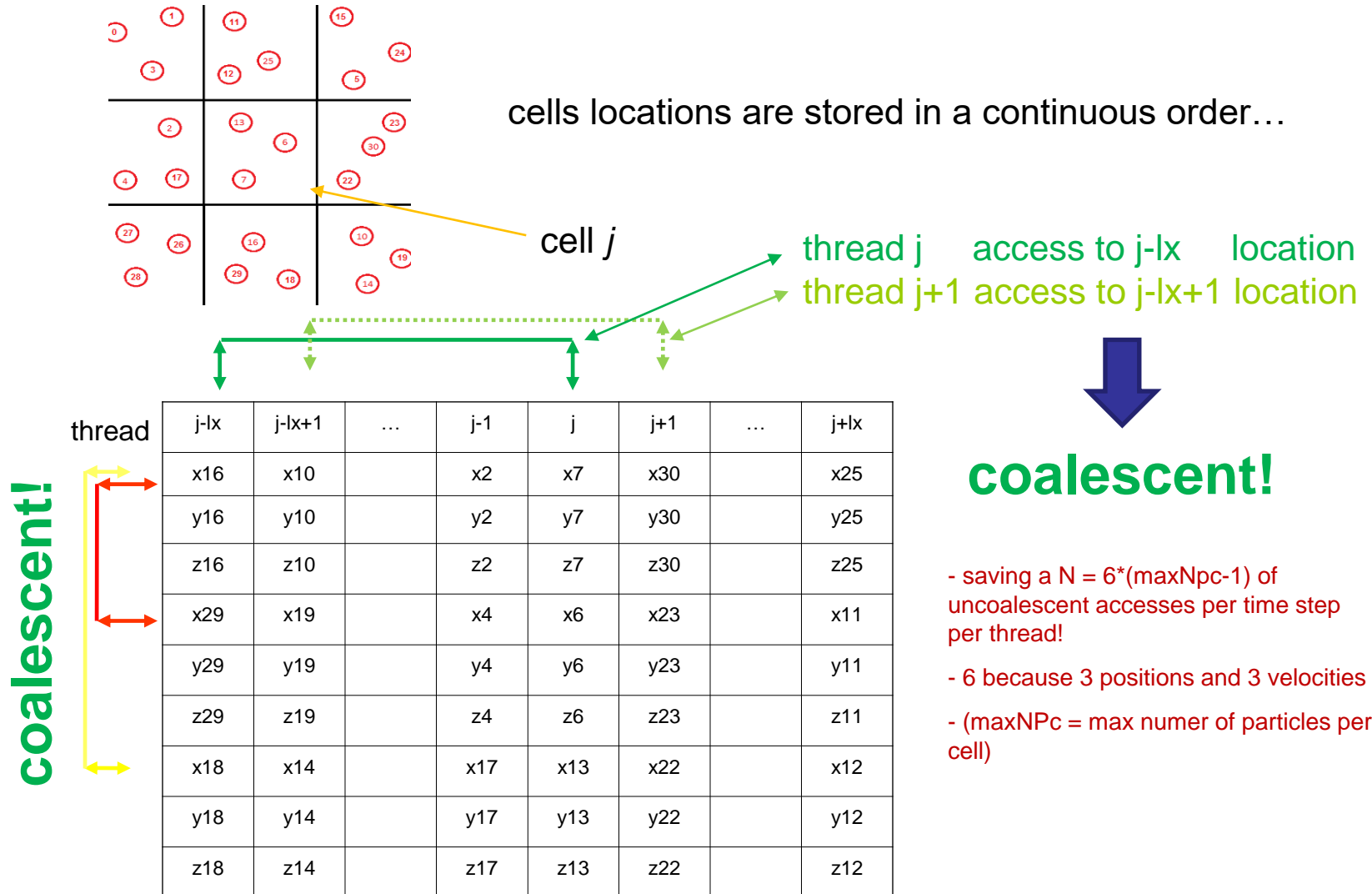
Main problem: memory access pattern



particle locations are stored in a continuous order...

Very uncoalescent access to the memory!

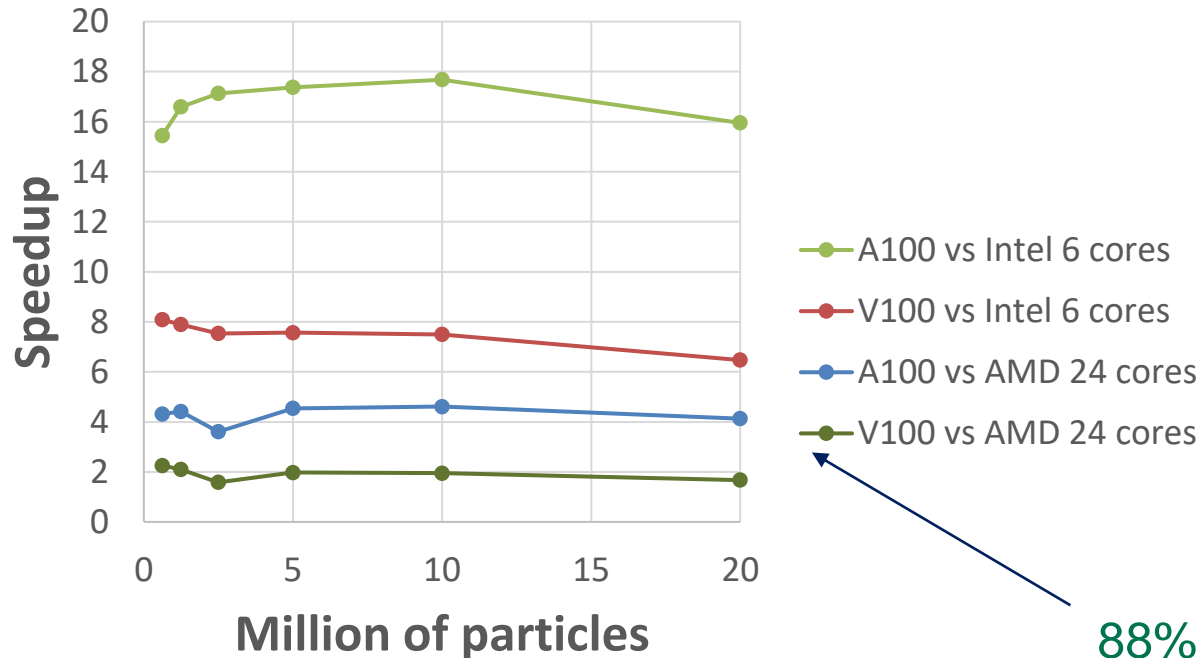
Reorganize the cell-linked array



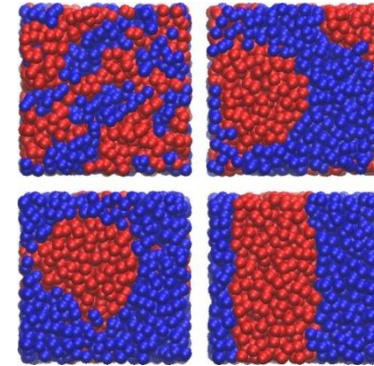
Speedup on single GPU

Speedup on GPU vs:

- AMD EPYC 7402 (Rome) **24 cores**
- Intel Xeon(R) W-2133 CPU @ 3.60GHz **6 cores**



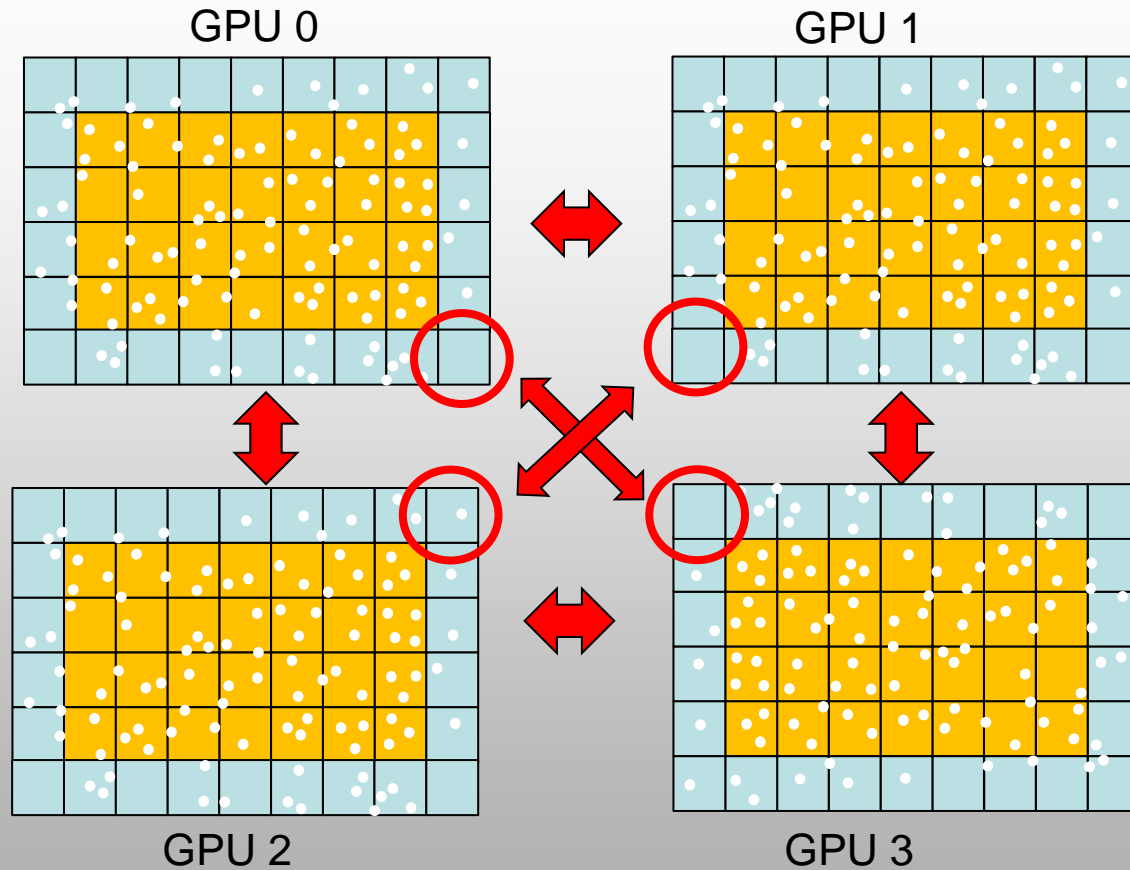
Test case: large mixture with constant the ratio particles/volume = 4



88% memory used!

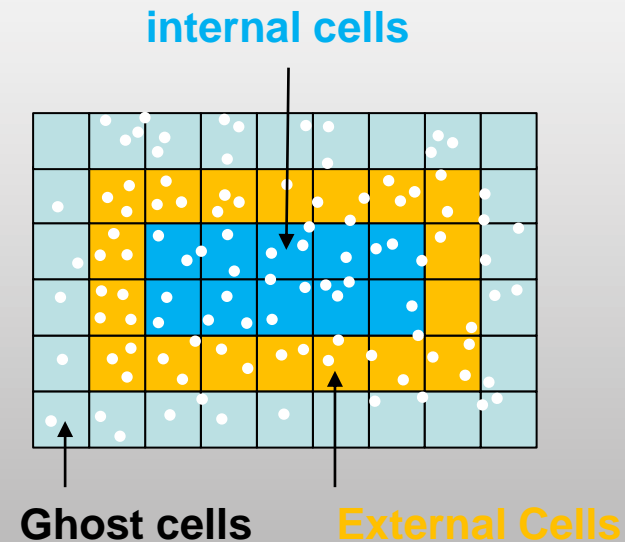
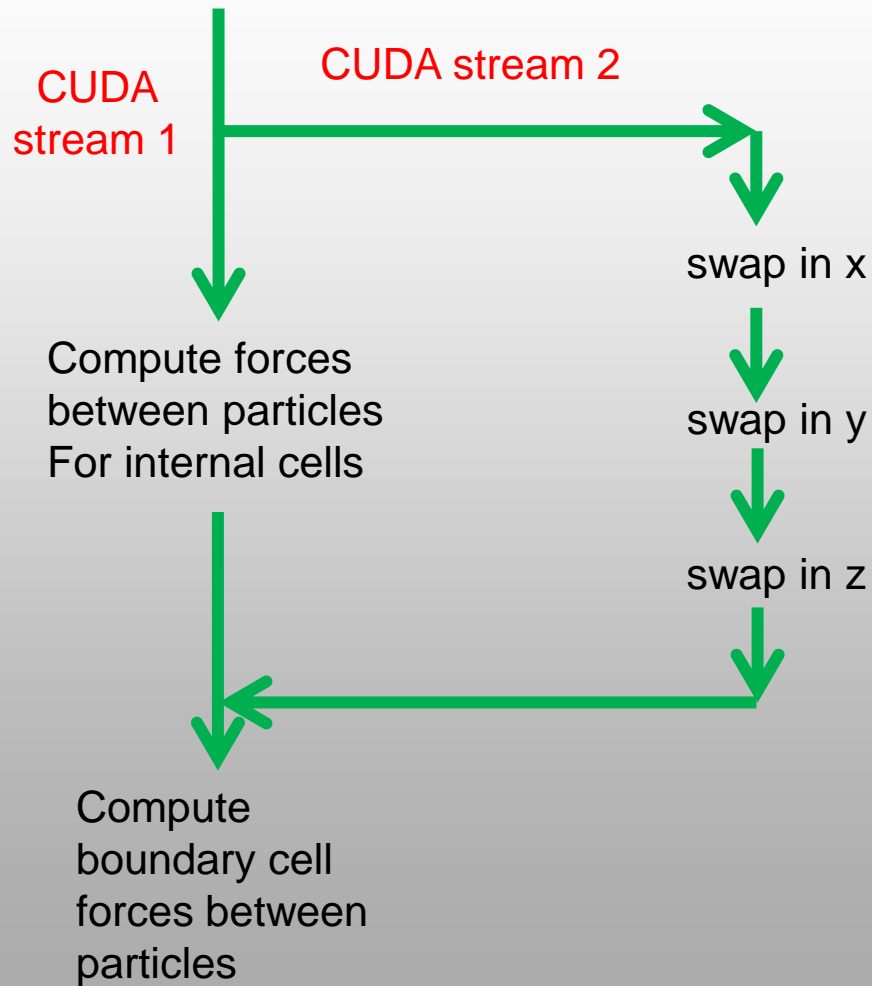
Castagna et al. "Towards Extreme Scale Dissipative Particle Dynamics Simulations using Multiple GPGPUs" Comput. Phys. Comm. (2020)

Multi GPU version



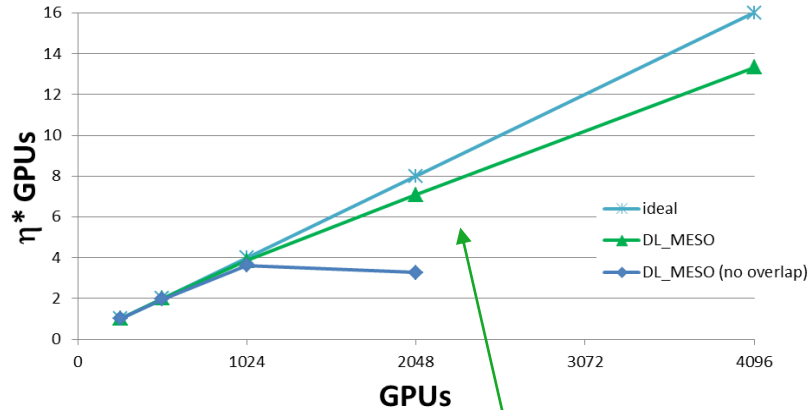
**in 3D you have
26 GPUs/GPU!!**

Overlap computation with communication



Scaling on different supercomputers

Strong scaling: 4096 GPUs (Piz Daint)

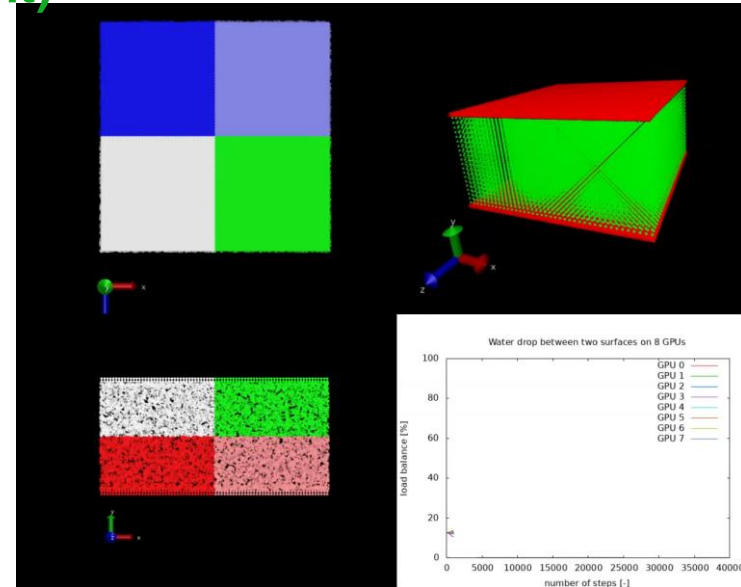


proper overlap computation-communication
has a strong impact on scaling!

Largest simulation: 14 billion particles!

D. Di Giusto and J. Castagna et al. "Scalable algorithm for many-body Dissipative Particle Dynamics using multiple General Purpose Graphic Processing Units" *Comput. Phys. Comm.* (2022)

Adding Load Balance



Animation **Water Drop** formation on 8 GPUs showing the impact of load balancing routine ALL (ALL is from Julich Supercomputer Centre)

Resume

- DL_MESO has been ported to single and multi-GPU **nNvidia GPUs** using **CUDA** language
- Good scaling up to **4096** GPU
- We can now run very large **DPD** simulations (**14 billions**)
- **Load balance** allows to run simulations without out of memory on the GPU, as well as save computational time
- 2 publications on **journal paper** (CPC)

Questions?