

CUDA Deep Dive: From Fundamentals to Advanced Techniques

N. Shukla | S. Orlandini | L. Ferraro

HPC Application Engineer

April 2nd 2025

Roadmap to basic to advanced

1

Why you should care about CUDA?

Grasping the basic elements of GPU programming

2

CUDA programming model

Kernel launch, Thread and Memory hierarchy

3

Performance consideration

Memory management, analysis Nsight and Nvidia

4

Streams and Concurrency

Overlapping kernel execution & data transfer on Single/Multi GPU

Topics is covered here

Identify the basic terminology used in CUDA

Compute Unified Device Architecture

CUDA programming Model

Program structure, Kernel launch, Managed memory, compilation workflow

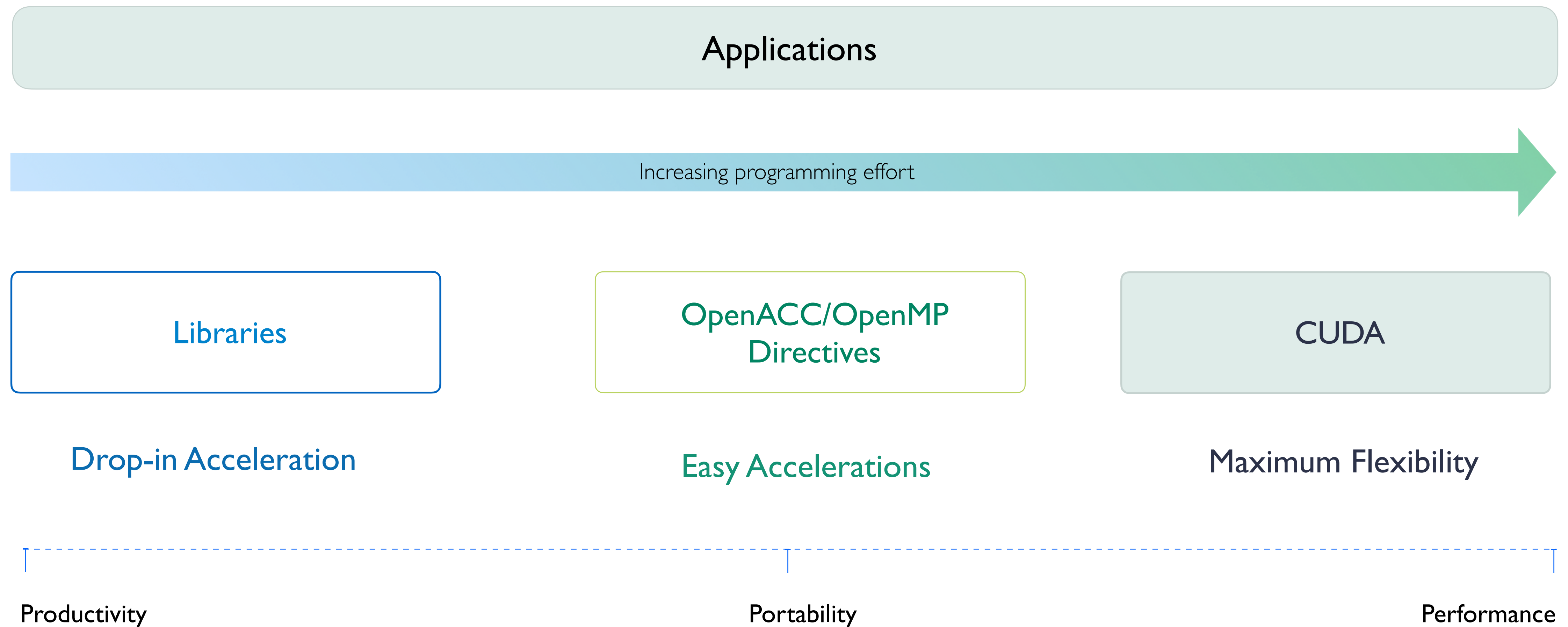
Measuring GPU performance

Execution Model

Understanding the Nature of Warp Execution, Thread hierarchy

 Why you should care about CUDA?

Ways to parallel an applications on Nvidia GPUs



Why CUDA matters

Best of both worlds

- **High-Level Productivity**
Use optimized libraries (cuBLAS, cuDNN) for AI, math, and analytics without reinventing the wheel. Frameworks rely on these under the hood!
- **Low-Level Flexibility**
Direct hardware access for fine-grained optimizations (memory management, kernel tuning) that frameworks often hide

Developer advantage

- **C++ Standard Support**
making GPU code more intuitive vs. compute shaders
- **Unified Codebase**
Write hybrid CPU/GPU code in one language, simplifying maintenance vs. Juggling separate shader/compute pipelines
- **Applications**
110000+ GPU accelerated apps
2M+ Global community

Performance Gains

- **100x speed increase**
Maximum performance boost for optimized applications
- **Energy Savings**
Typical reduction (80%) in power consumption vs CPU-only
- **Data Processing**
Process TB/s of information in real-time

Why CUDA matters

Performance

- Massive Parallelism: scale to 1000's of cores, 10000000's of parallel thread
- Massive Gain: substantial performance improvements in tasks that can be divided into smaller, concurrent operations

Scalability

- Efficiently maps to the GPU architecture: well-suited for leveraging GPU capabilities
- Wide Range of Hardware: applications can scale from small embedded devices to large supercomputers

Flexibility

- Programming Languages: supports various programming languages
- Easy to use: let programmers strip away complexity associated with parallel computing and focus on parallel algorithms

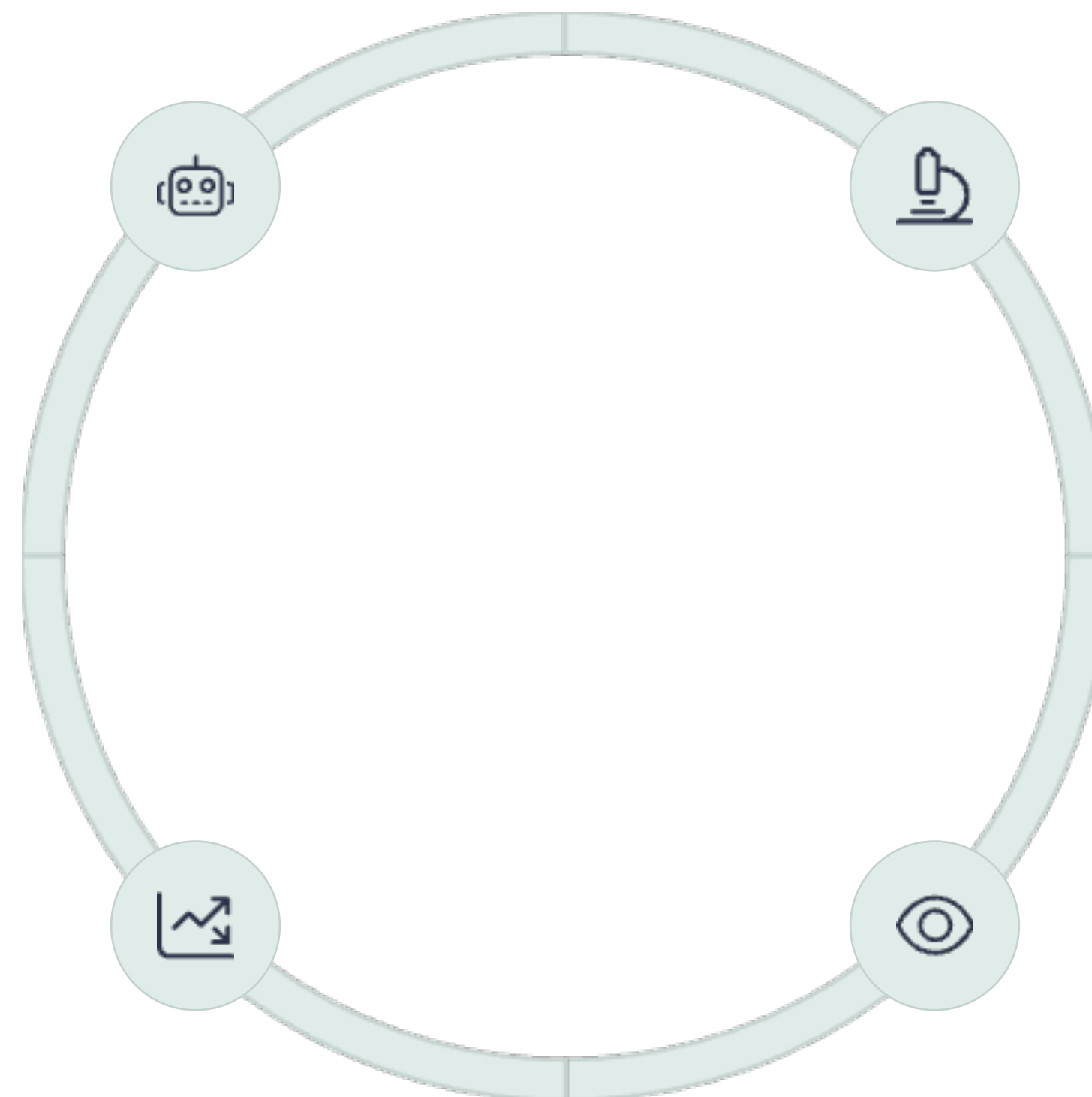
Where CUDA makes an impact

AI & Machine Learning

Powers neural networks and deep learning frameworks

Financial Analysis

Speeds up risk assessment and predictive models



Scientific Research

Accelerates complex simulations and modeling

Computer Vision

Enables real-time image and video processing

 What is CUDA?

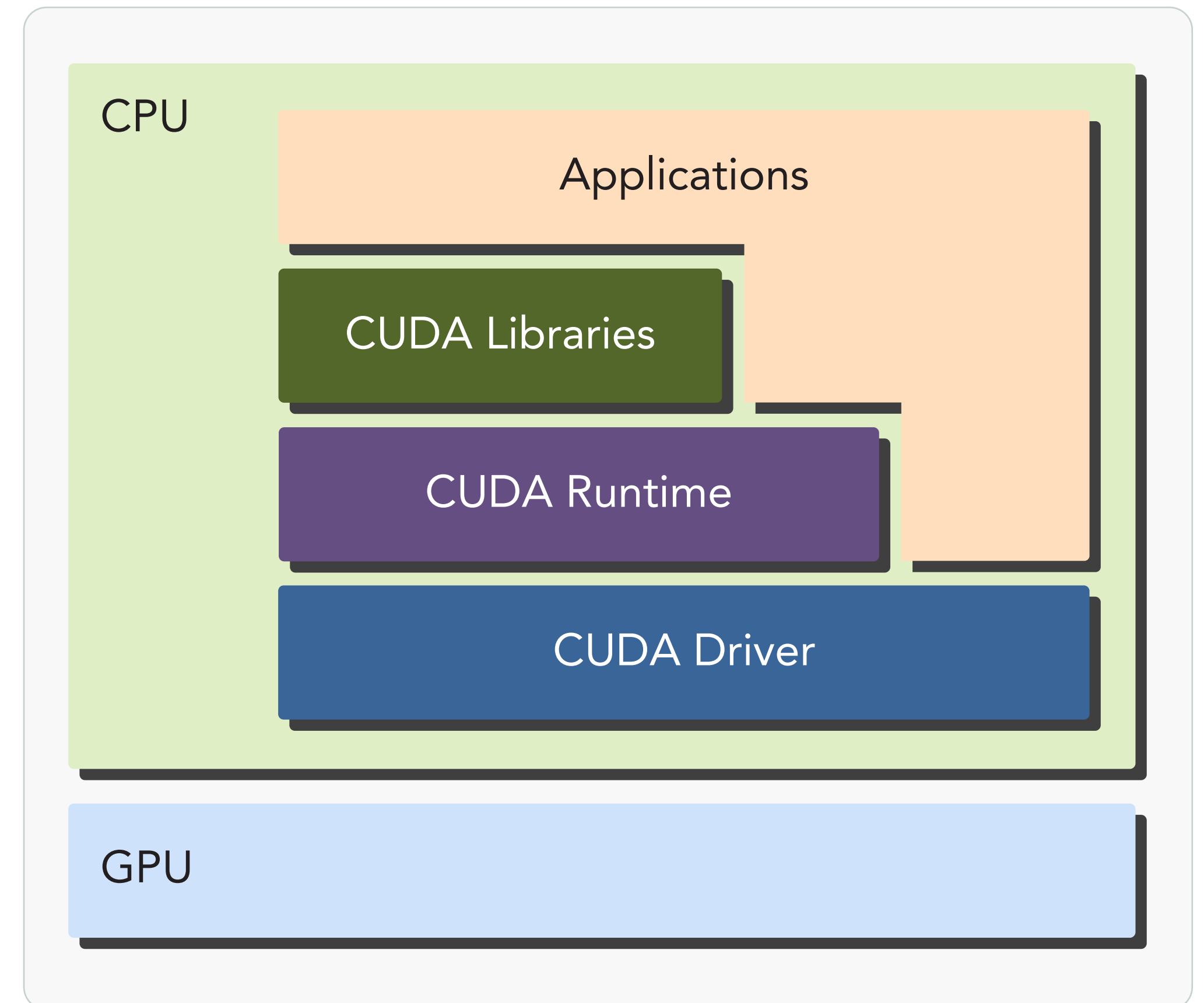
What is CUDA?

CUDA : Compute Unified Device Architecture

- Enable heterogeneous systems (i.e., CPU+GPU)
- A new architecture instruction set called PTX (Parallel Thread eXecution) to match GPU typical hardware
- Parallelism allows developers to use GPUs for general purpose processing (GPGPU)

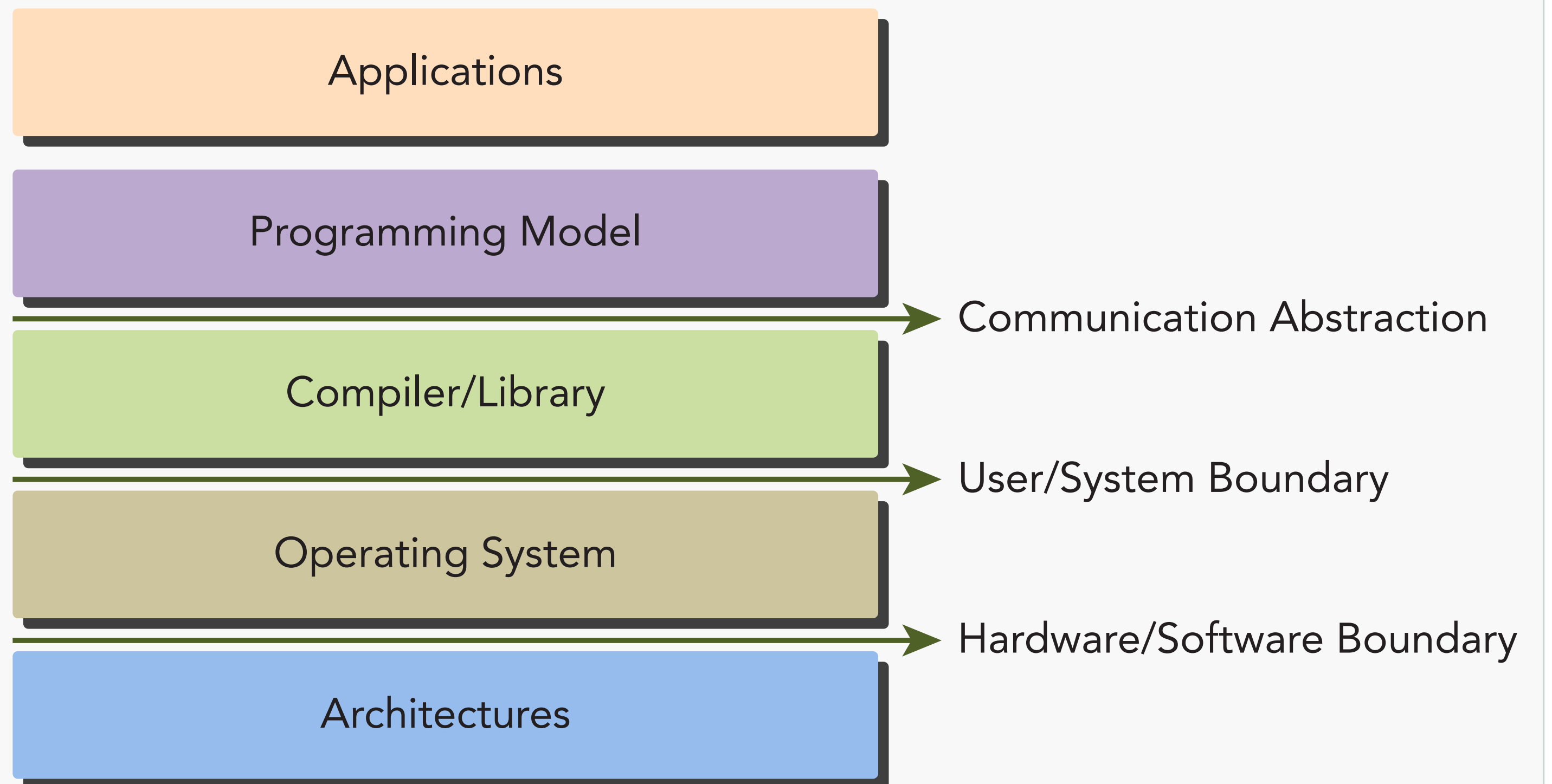
The SDK includes

- A Drivers, runtimes and API
- Compiler wrappers for compile cuda code (nvcc)
- Libraries (cuBLAS, cuFFT, cuSolver) debuggers (cuda-gdb, cuda-memcheck), profilers (nvprof, nView), etc
- CUDA-aware languages C/C++, Fortran, PyCUDA, CUDA.JI



CUDA programming model

- Abstraction of computer architectures
- Bridge between app and implementation
- Communication abstraction: program vs. model boundary
- Enabled by compilers/libraries, hardware, and OS
- Program dictates info sharing and coordination
- Offers logical view of computing architectures
- Embodied in languages or environments



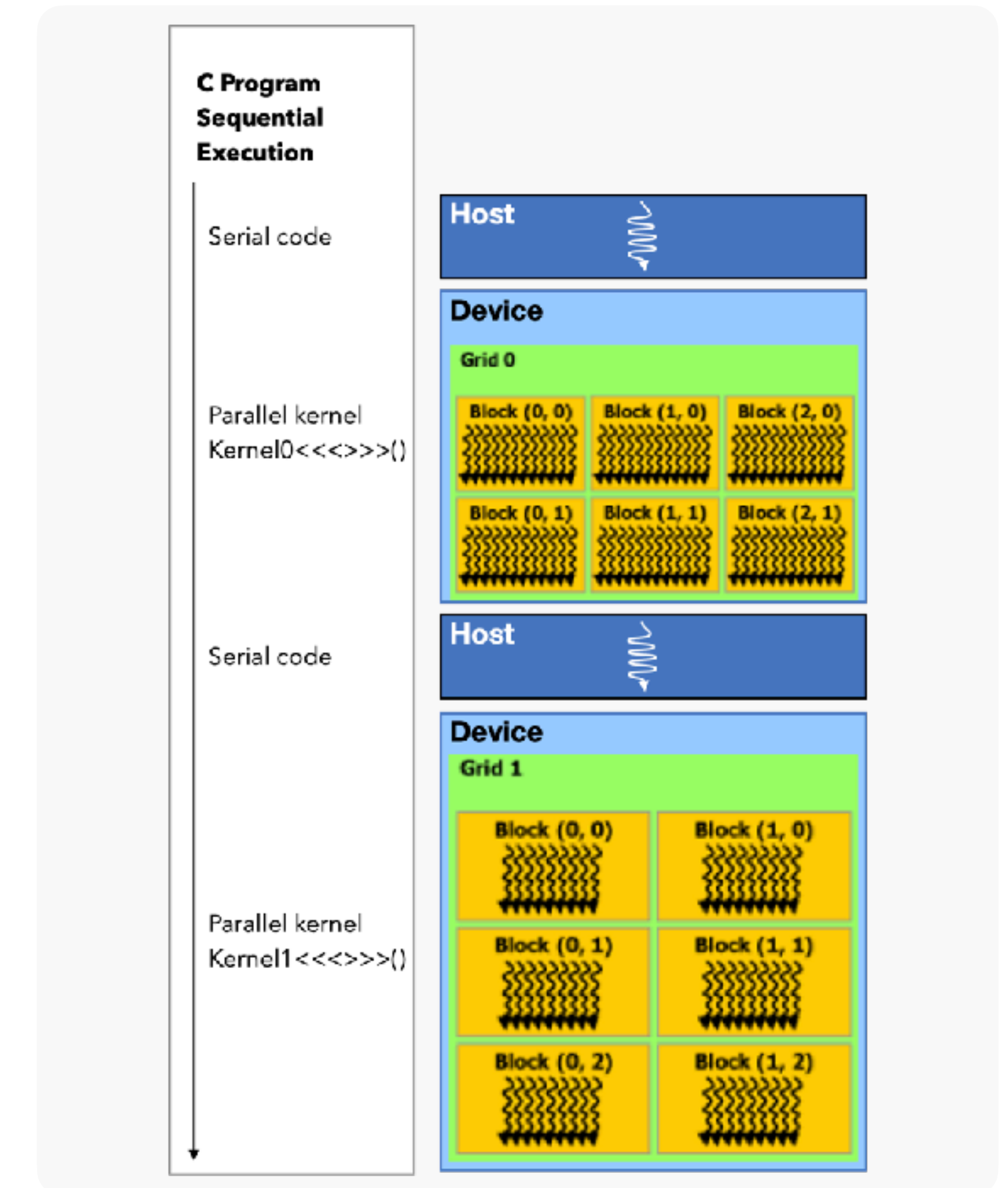
CUDA execution model

CUDA programmer perspective

- Heterogenous computing: combination of CPU and GPU
 - Host: The CPU and its memory
 - Device: The GPU and its memory
- Execution: Programs run a on the host and launch parallel code (kernels) on the device by many threads

Programming model view

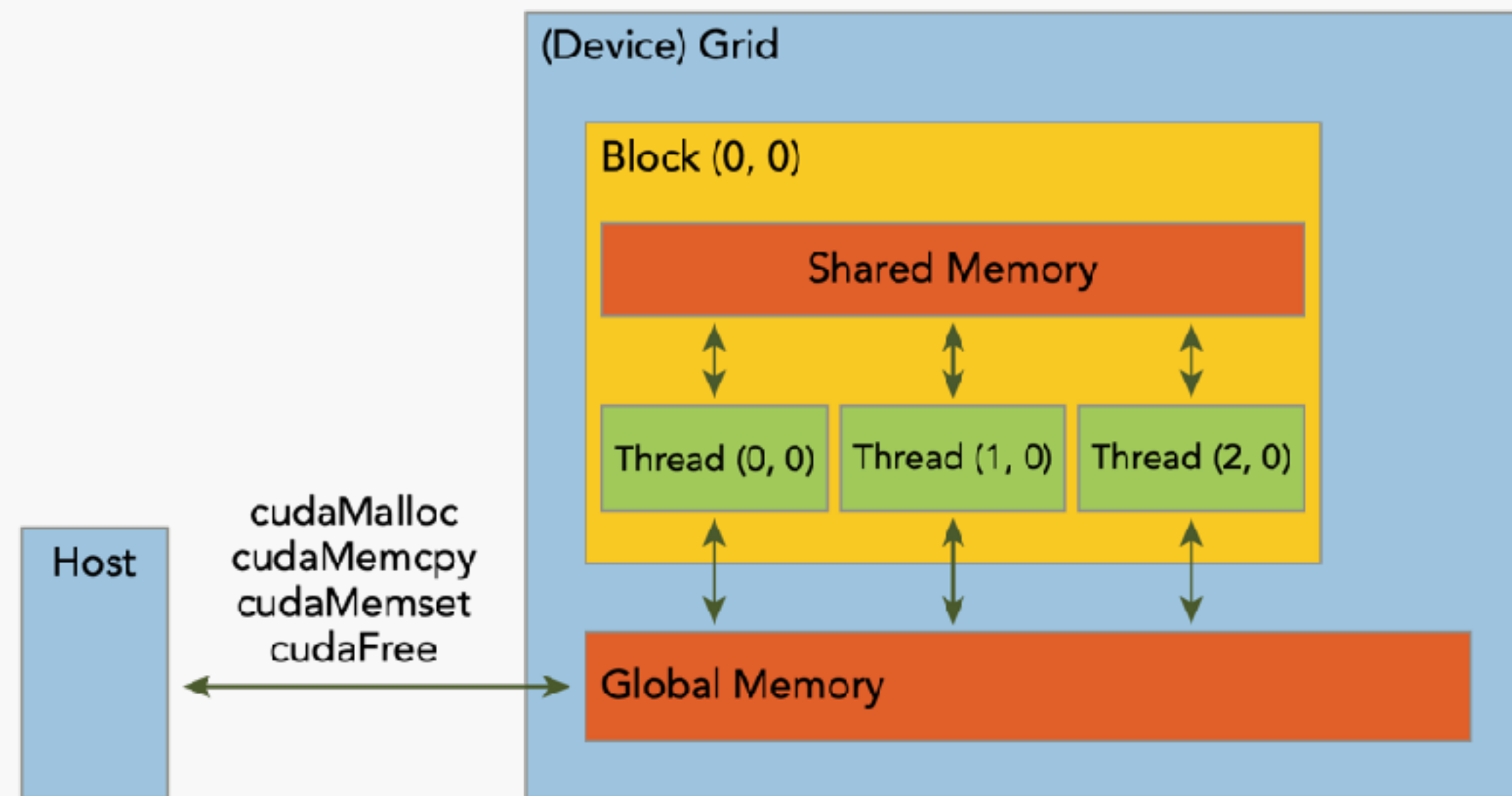
- Kernels: A function written in CUDA C/C++ and executed on the GPU
- Launch configurations:
 - Threads: Smallest unit of execution in CUDA
 - Block: A collection of threads
 - Grid: A collection of blocks
- Memory management: Allocate and transfer data between host (CPU) and device (GPU)



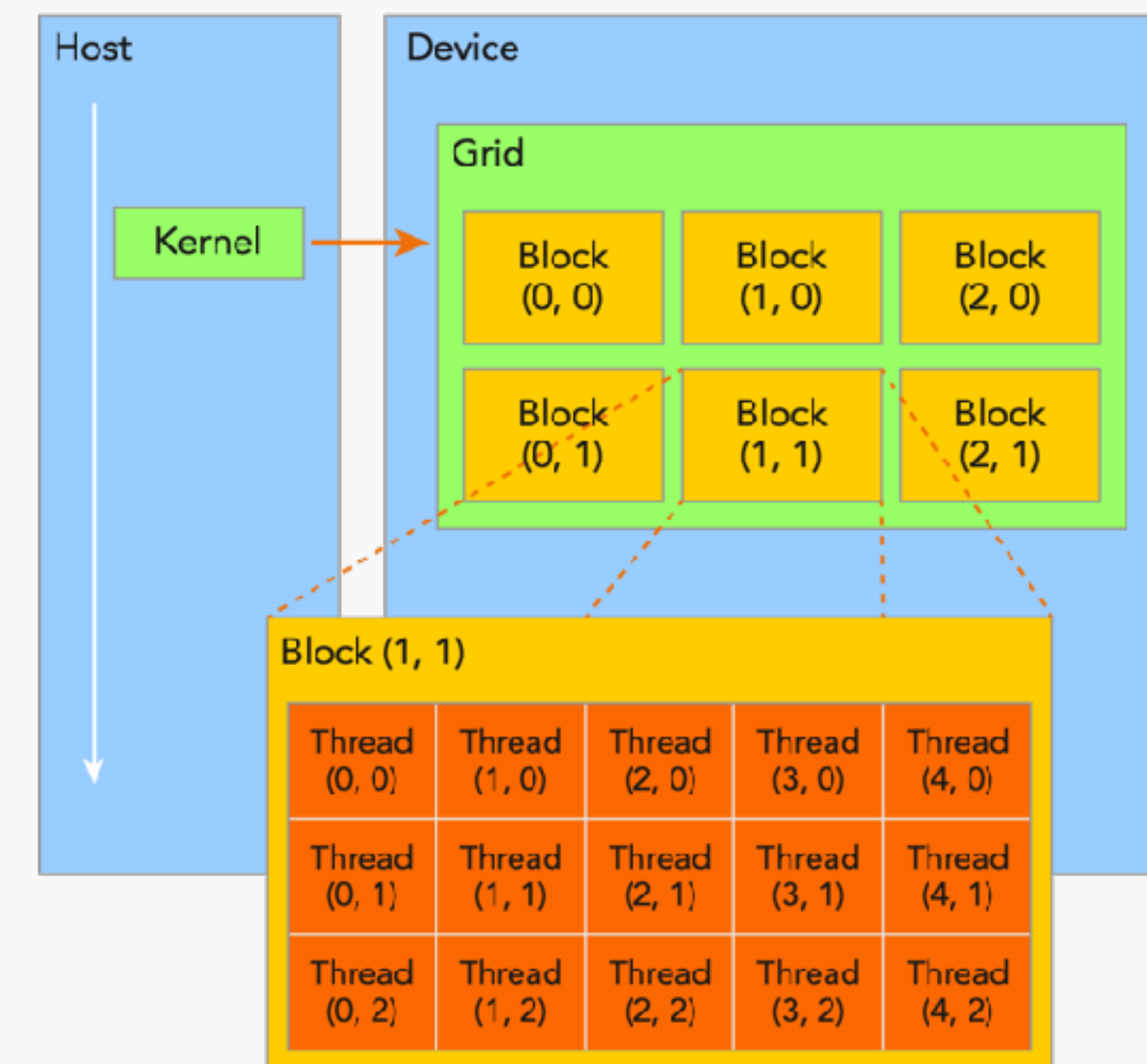
GPUs serve as a co-processor, not a standalone platform

CUDA enhances your control over memory and thread hierarchies, optimizing execution and scheduling with:

Memory hierarchy structure



Thread hierarchy structure



Embarrassing parallel code

Vector Addition

- Simple operation: a memory-bound operation
- Natural Fit for GPUs: Each element of a vector are independent
- Scalability: Larger vectors benefit from GPU or multi-core CPU parallelism, offering faster computation than serial processing.

CPU function

```
sumArraysOnHost(float *A, float *B, float *C, const int N)
{
    for (int idx=0; idx<N; idx++)
        C[idx] = A[idx] + B[idx];
}

int main(int argc, char **argv)
{
    ..
    Start = cpuSecond();
    sumArrayOnCPU(h_A, h_B, h_C, N);
    Double cpuTime = cpuSecond() - start;
    printf("CPU Execution Time: %f second \n", cpuTime);
    ..
}
```

Declaring Host-Called, Device-Executed Functions

CUDA differentiates between these functions by using one of the following function type qualifiers as a prefix

- `__host__` functions called from host and executed on the host
- `__device__` functions called from device and execute on the device (a function that is called from a kernel needs the `__device__` qualifier)
- `__global__` qualifier for kernels that can be invoked globally

Step to Launching a CUDA Kernel

`__global__ void()`

Defines a kernel
can be invoked globally either from CPU or GPU

Execution configuration

Kernel_name <<<numBlocks, numThreads>>> (arguments);
Specifies grid and block dimensions

Synchronization

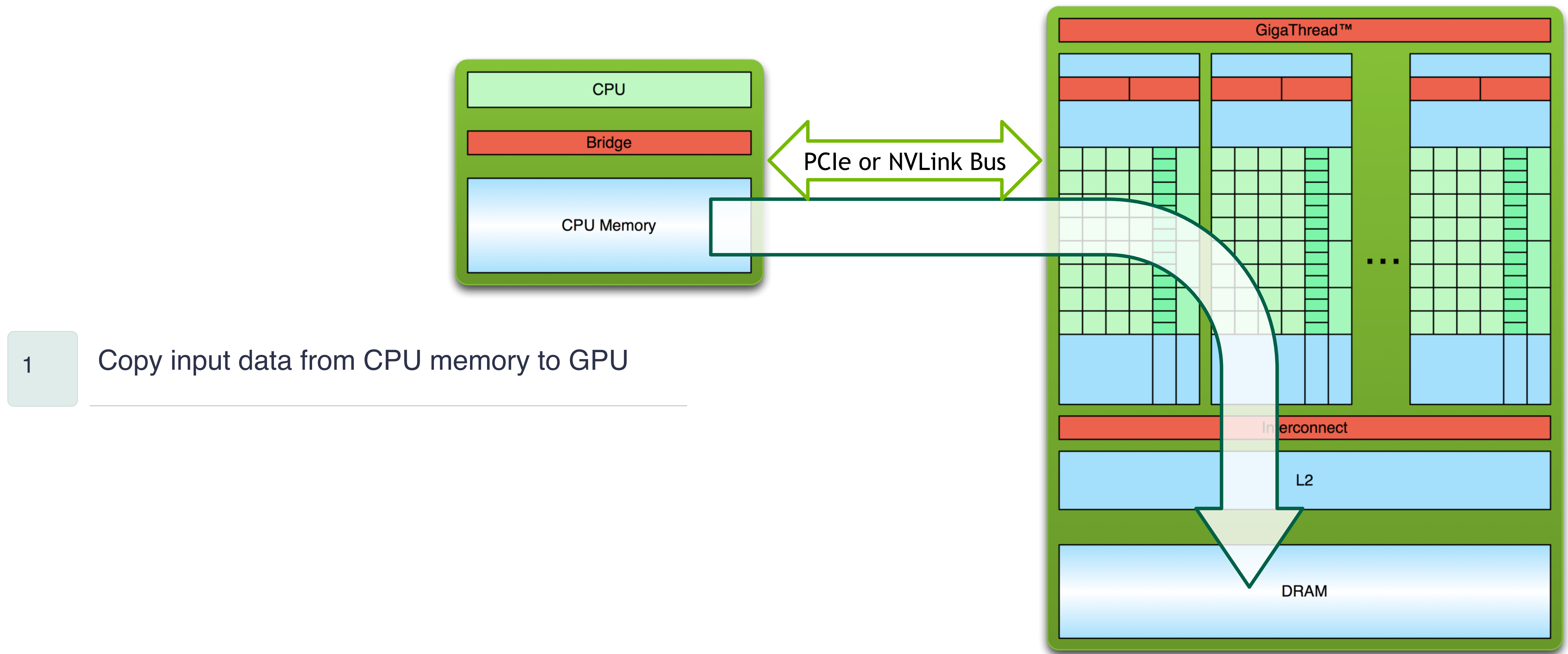
Launching kernel is asynchronous
`cudaDeviceSynchronize()`: wait until device code completeness

```
// Kernel
__global__
sumArraysOnDevice(float *A, float *B, float *C, const int N)
{
    for (int idx=0; idx<N; idx++)
        C[idx] = A[idx] + B[idx];
}

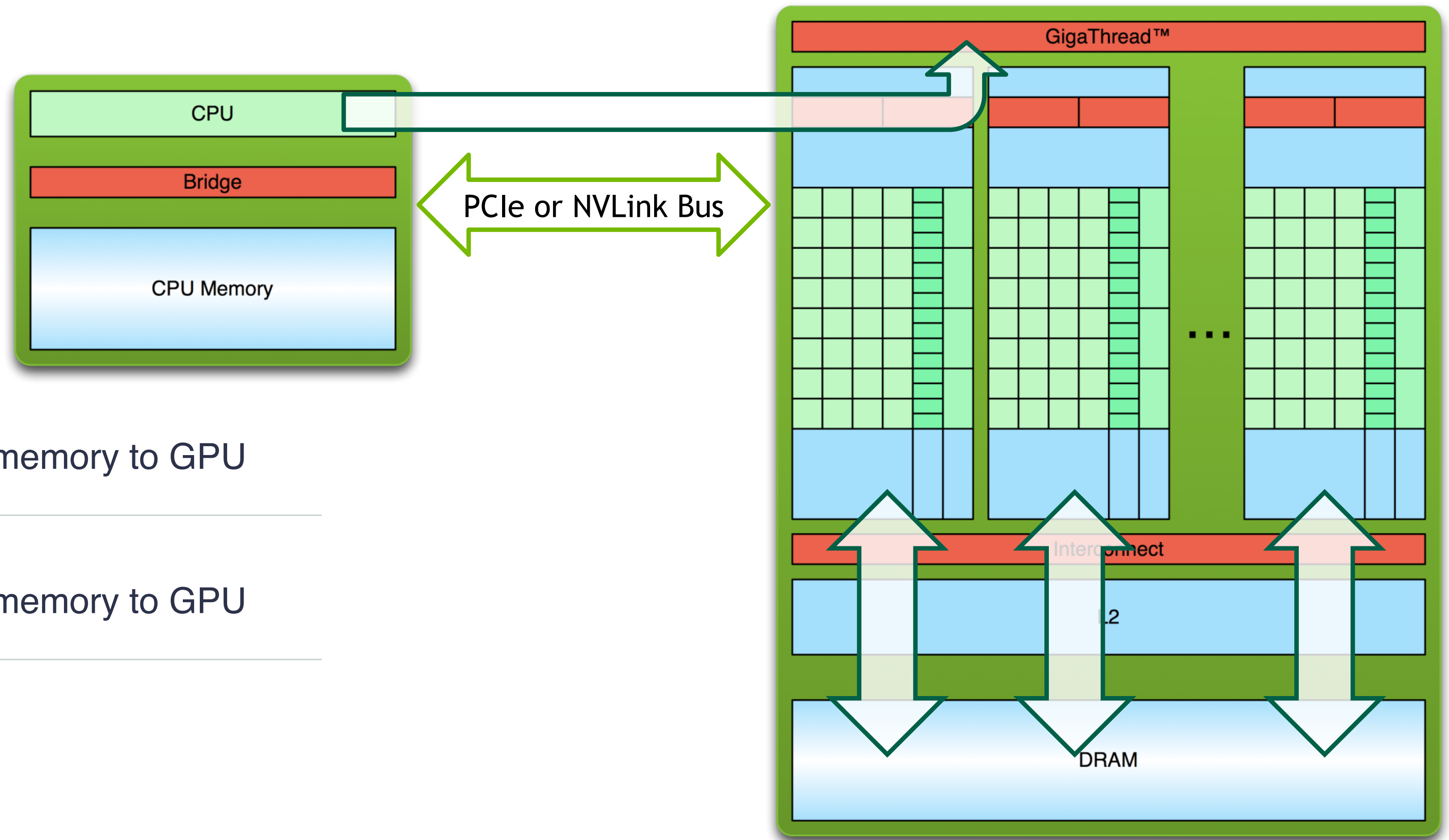
int main(int argc, char **argv)
{
    ..
    start = cpuSecond();
    sumArraysOnGPU<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);
    cudaDeviceSynchronize();
    double gpuTime = cpuSecond() - start;
    printf("GPU Execution Time: %f seconds\n", gpuTime);
    ..
}
```

Managing Memory

Three simple processing steps



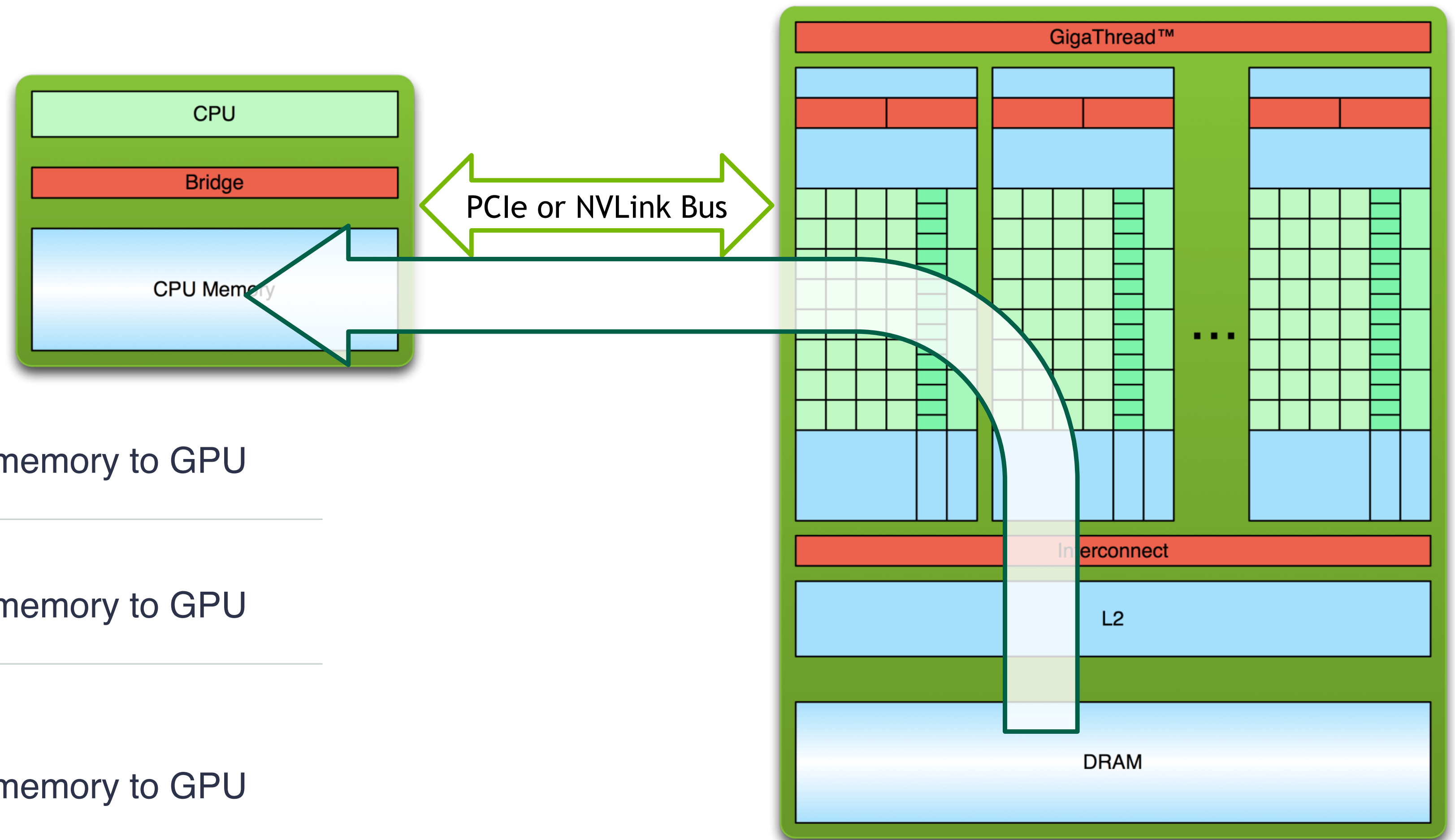
Three simple processing steps



1 Copy input data from CPU memory to GPU

2 Copy input data from CPU memory to GPU

Three simple processing steps



1 Copy input data from CPU memory to GPU

2 Copy input data from CPU memory to GPU

3 Copy input data from CPU memory to GPU

Data movement

1 Copy host to Device

2 Copy Device to host

3 Clean up memory for host and device

// Copy data from host to device

```
checkCuda( cudaMemcpy(d_A, h_A, size,  
cudaMemcpyHostToDevice) );  
checkCuda( cudaMemcpy(d_B, h_B, size,  
cudaMemcpyHostToDevice) );
```

// Copy result from device to host

```
checkCuda( cudaMemcpy(h_C_ref, d_C, size,  
cudaMemcpyDeviceToHost) );
```

// Clean up memory

```
checkCuda( cudaFree(d_A) );  
checkCuda( cudaFree(d_B) );  
checkCuda( cudaFree(d_C) );  
cleanup(h_A, h_B, h_C, h_C_ref);
```

Unified virtual memory (UVM)

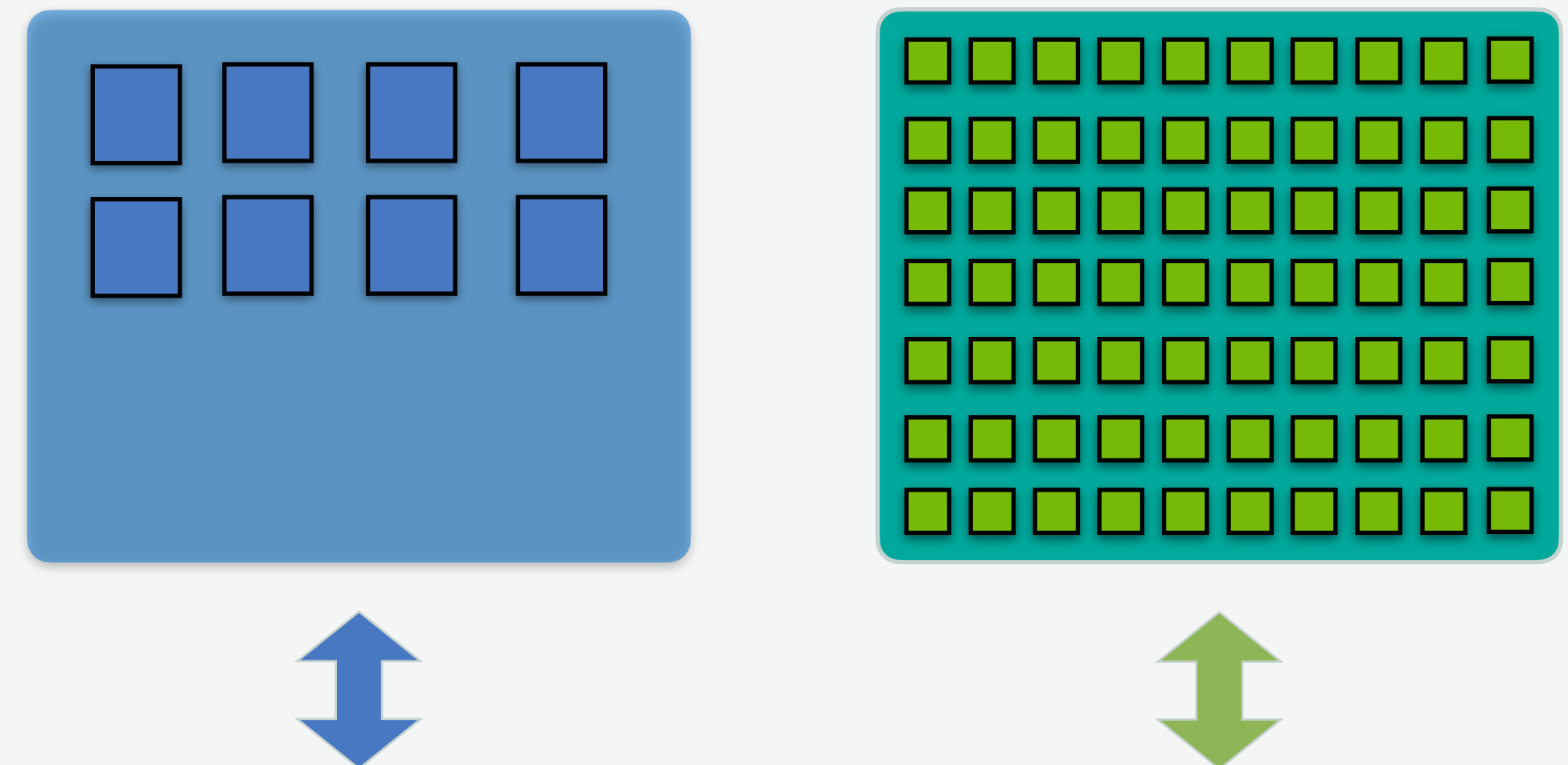
Increased memory latency

- Single allocation, single pointer, accessible everywhere
eliminate the need of explicit copy and simplify code porting
- Enables the sharing of memory which reduces overall usage

Limited control over memory placement

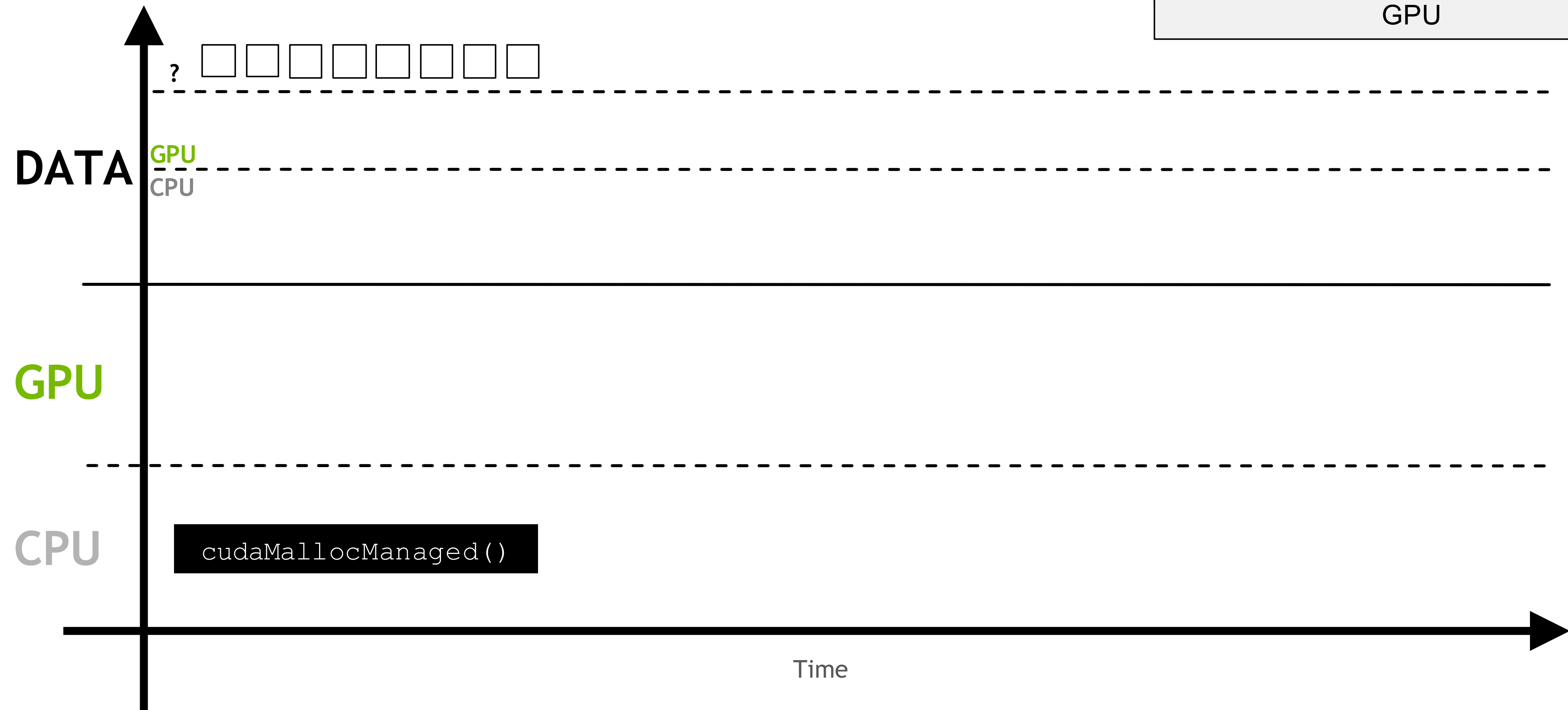
UVM automatically manages memory placement, which may not always be optimal for a given application

Developer view of GPU memory



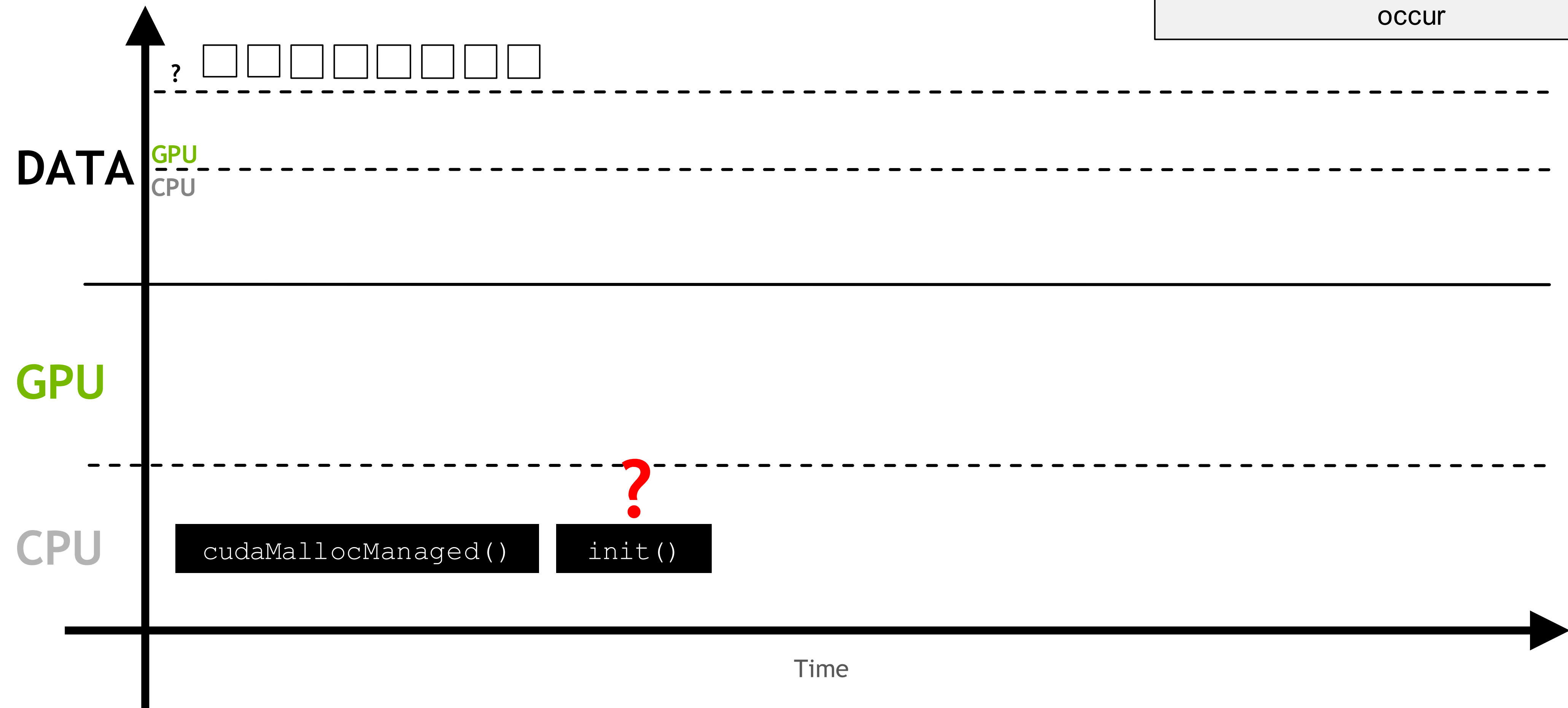
How does cudaMallocManaged actually works?

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



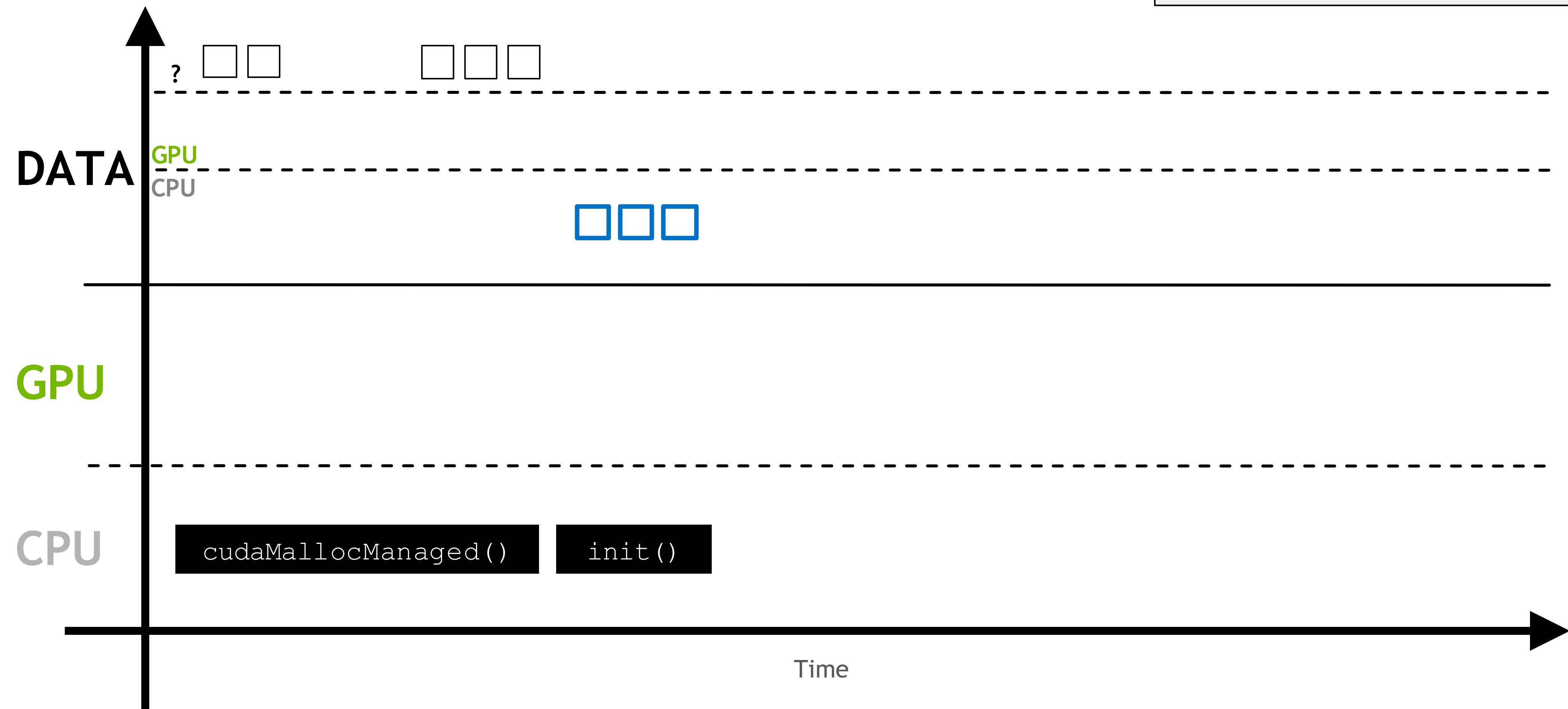
How does cudaMallocManaged actually works?

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



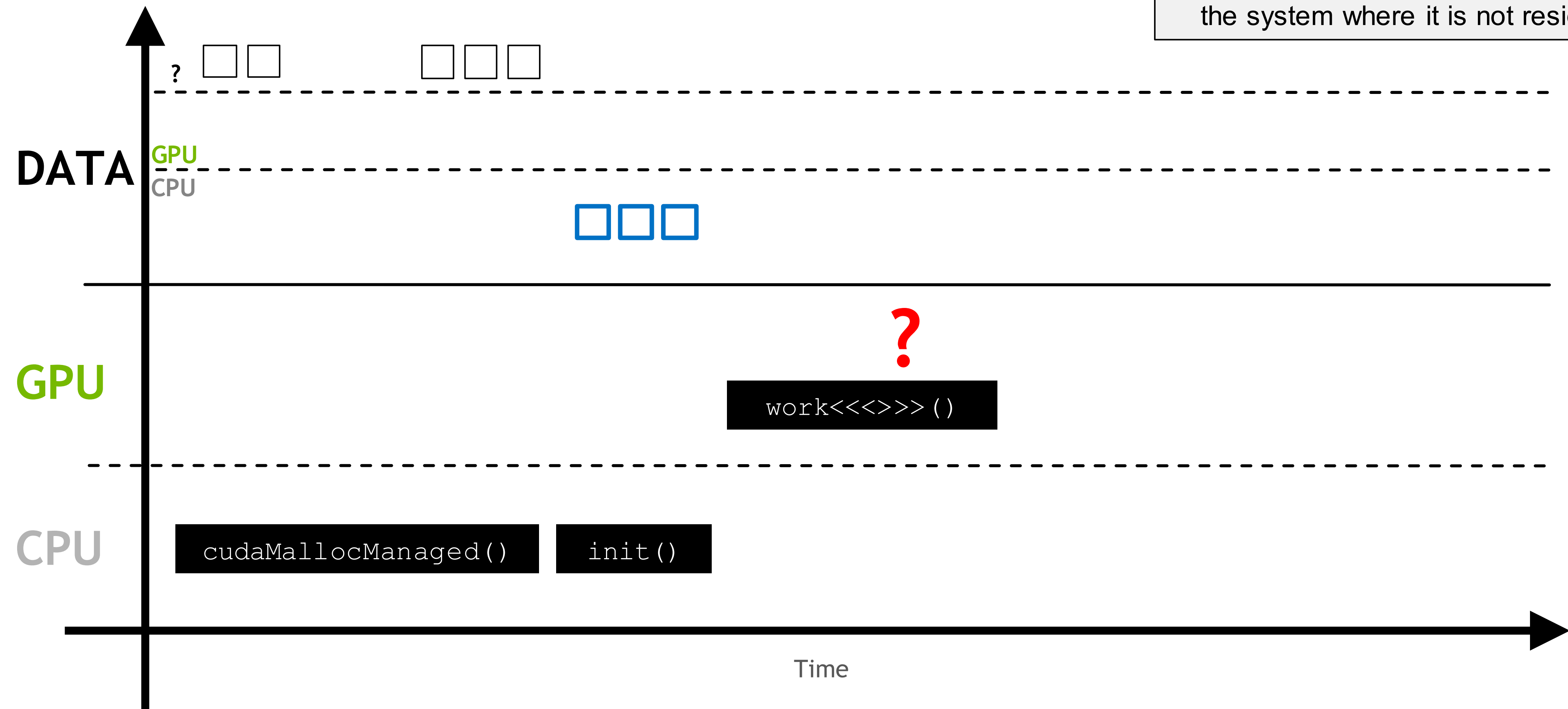
How does cudaMallocManaged actually works?

The page fault will trigger the migration of the demanded memory



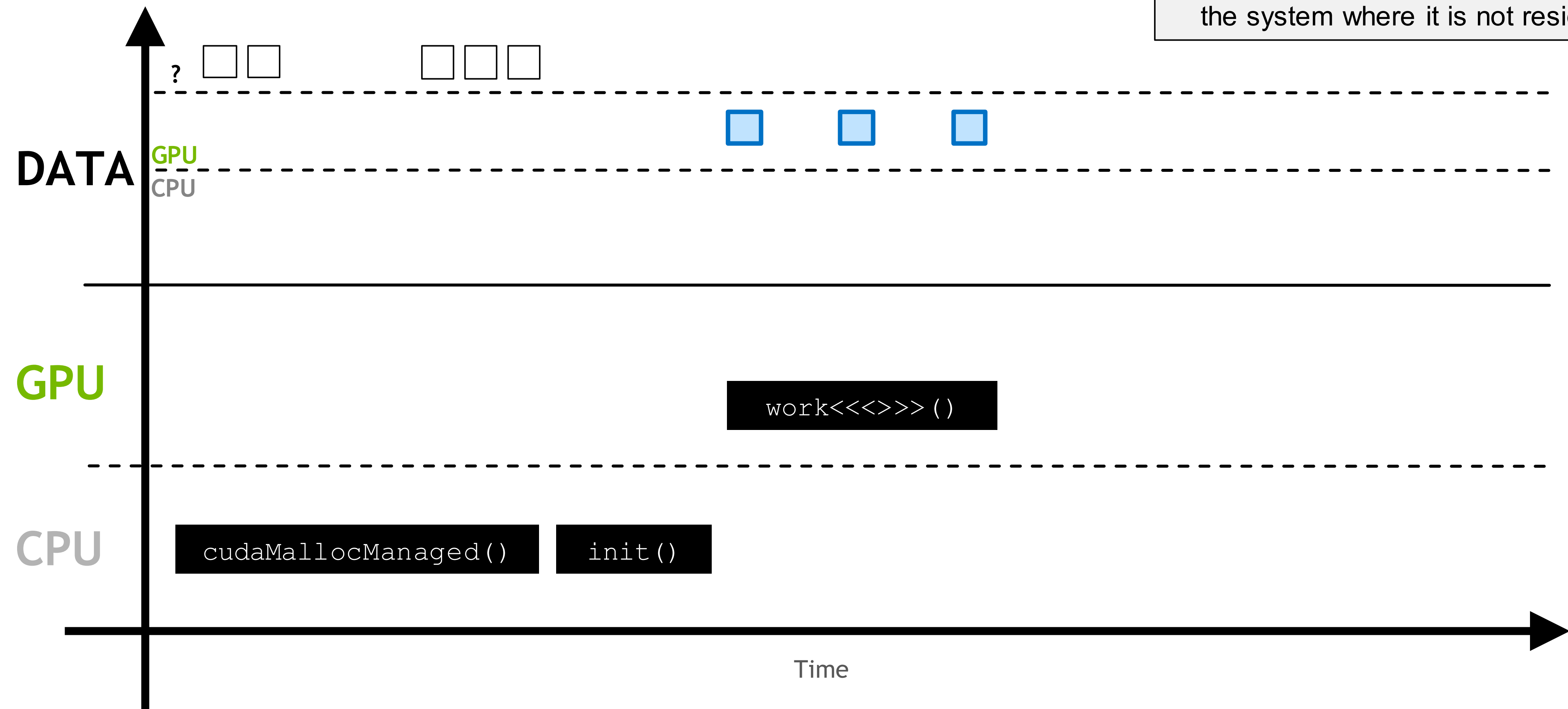
How does cudaMallocManaged actually works?

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



How does cudaMallocManaged actually works?

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



Simplified memory management code

Allow to **allocate** and **free memory**

CPU code

```
int N = 10000;  
size_t size = N*sizeof(int);
```

```
int *a;
```

```
a = (int*)malloc(size);
```

```
free(a);
```

CUDA Code with UM

```
int N = 10000;  
size_t size = N*sizeof(int);
```

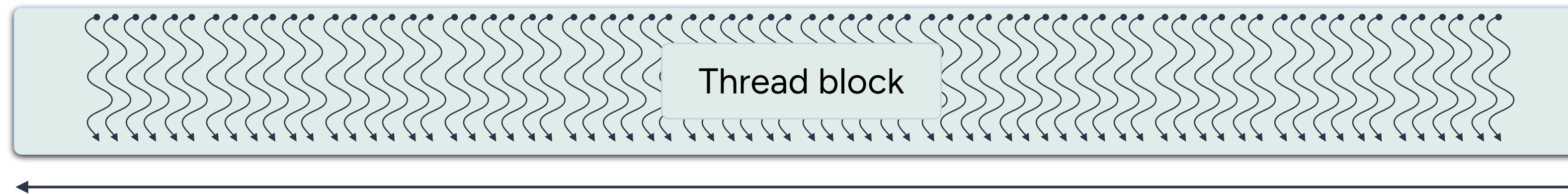
```
int *a;
```

```
cudaMallocManaged(&a, size);
```

```
cudaFree(a);
```

 CUDA Threads organization

CUDA launches arrays of parallel threads



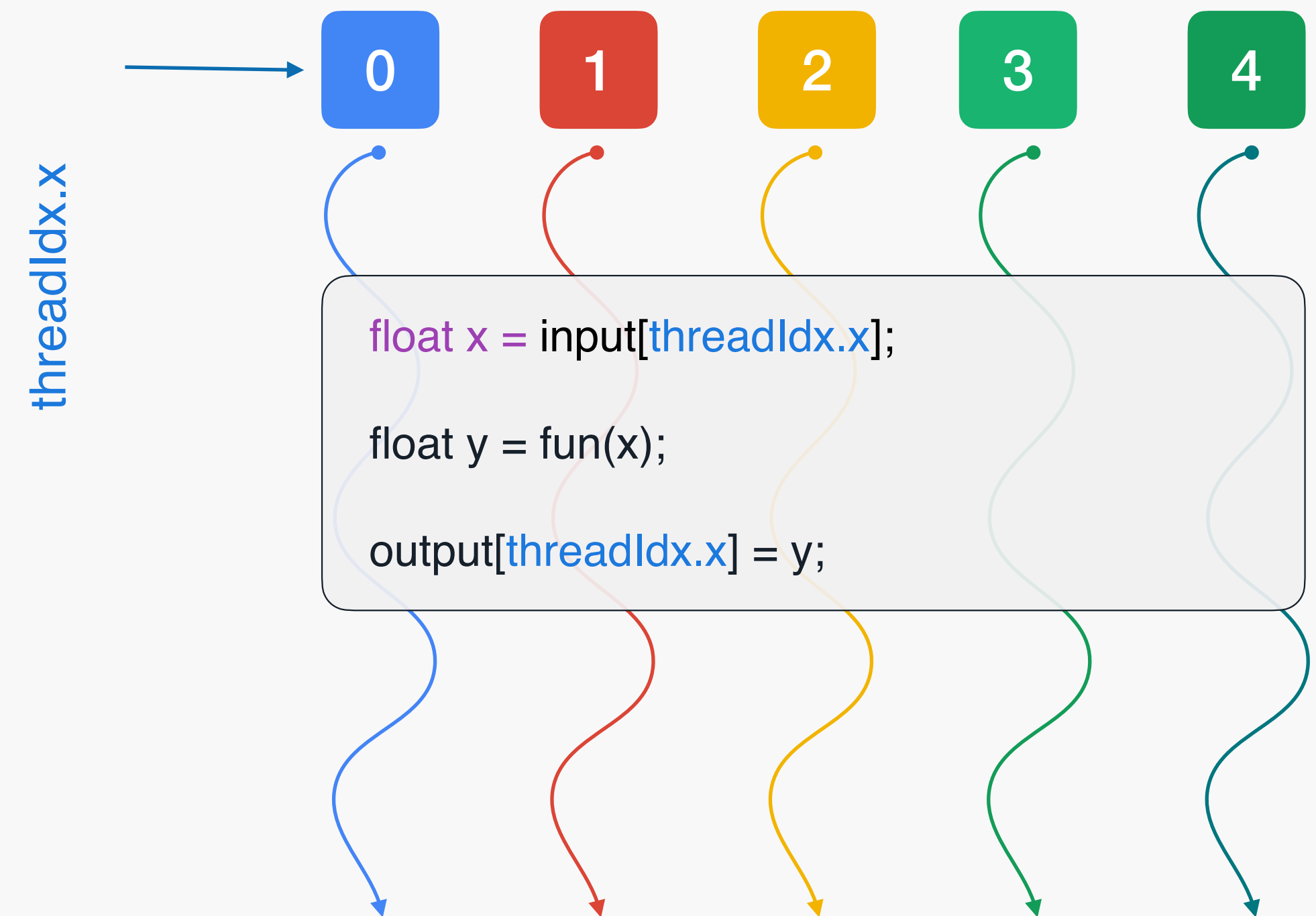
A block has a fixed number of threads which are guaranteed to be running simultaneously on the same SM

CUDA launches arrays of parallel threads

For fully utilisation of the parallel processing power of the GPU

A CUDA kernel is executed as a grid (array) of threads

- All threads in a grid run the same kernel code
- Each thread has a unique ID: `threadIdx`
- Threads are similar to data-parallel tasks.
- Threads independently execute the same operation on a data subset
- Follows SPMD model i.e the Single Program Multiple Data => SIMT Single Instructions Multiple threads



Error handling

Validate GPU results by comparing with CPU results

```
// Validate results

bool validateResults(float *hostRef, float *gpuRef, int nElem) {
    bool correct = true;
    for (int i = 0; i < nElem; i++) {
        if (fabs(hostRef[i] - gpuRef[i]) > 1e-5) {
            correct = false;
            printf("Mismatch at index %d: CPU = %f, GPU = %f\n", i, hostRef[i], gpuRef[i]);
            break;
        }
    }

    if (correct) {
        printf("Results match!\n");
    }
    return correct;
}
```

Kernel Launch Errors

- Error handling in accelerated CUDA code is essential.
- All CUDA API returns an error code of type `cudaError_t`
 - Special value `cudaSuccess` means that no error occurred
- An error message can be printed with `cudaGetErrorString`

```
cudaError_t err;  
err = cudaMallocManaged(&a, N);  
if(err != cudaSuccess) { printf("Error: %s \n", cudaGetErrorString(err)); }
```

- To check for errors occurring at the time of kernel launch, CUDA provides the `cudaGetLastError` function, which does return a value of type `cudaError_t`

```
someKernel <<<1, -1 >>>();    // - 1 is not a valid number of threads  
cudaError_t err;  
err = cudaGetLastError();  
if(err != cudaSuccess) { printf("Error: %s \n", cudaGetErrorString(err)); }
```

CUDA Error Handling Function

- A macro that wraps CUDA function calls for checking errors could be useful
- Can be wrapped around any function that returns a `cudaError_t`

```
#include <stdio.h>
#include <assert.h>

inline cudaError_t checkCuda(cudaError_t result) {
    if (result != cudaSuccess) {
        fprintf(stderr, "CUDA Runtime Error: %s\n", cudaGetErrorString(result));
        assert(result == cudaSuccess); }
    return result; }

int main() {
    /* The macro can be wrapped around any function returning
    * a value of type `cudaError_t`.
    */
    checkCuda( cudaDeviceSynchronize() )
}
```

Asynchronous errors

To catch errors that occur in **asynchronous part** of the code (for example during the execution of an asynchronous kernel), check the **status returned by a subsequent synchronizing CUDA runtime API call**, such as `cudaDeviceSynchronize`.

```
cudaError_t asyncErr;  
asyncErr = cudaDeviceSynchronize(); if (asyncErr != cudaSuccess)  
{  
    printf("Error: %s\n", cudaGetErrorString(err));  
}
```

 How to compile CUDA enable application?

CUDA components: Source Files and Compilation

1

CUDA Driver

A critical piece of software that acts as the interface between your application and the NVIDIA GPU hardware

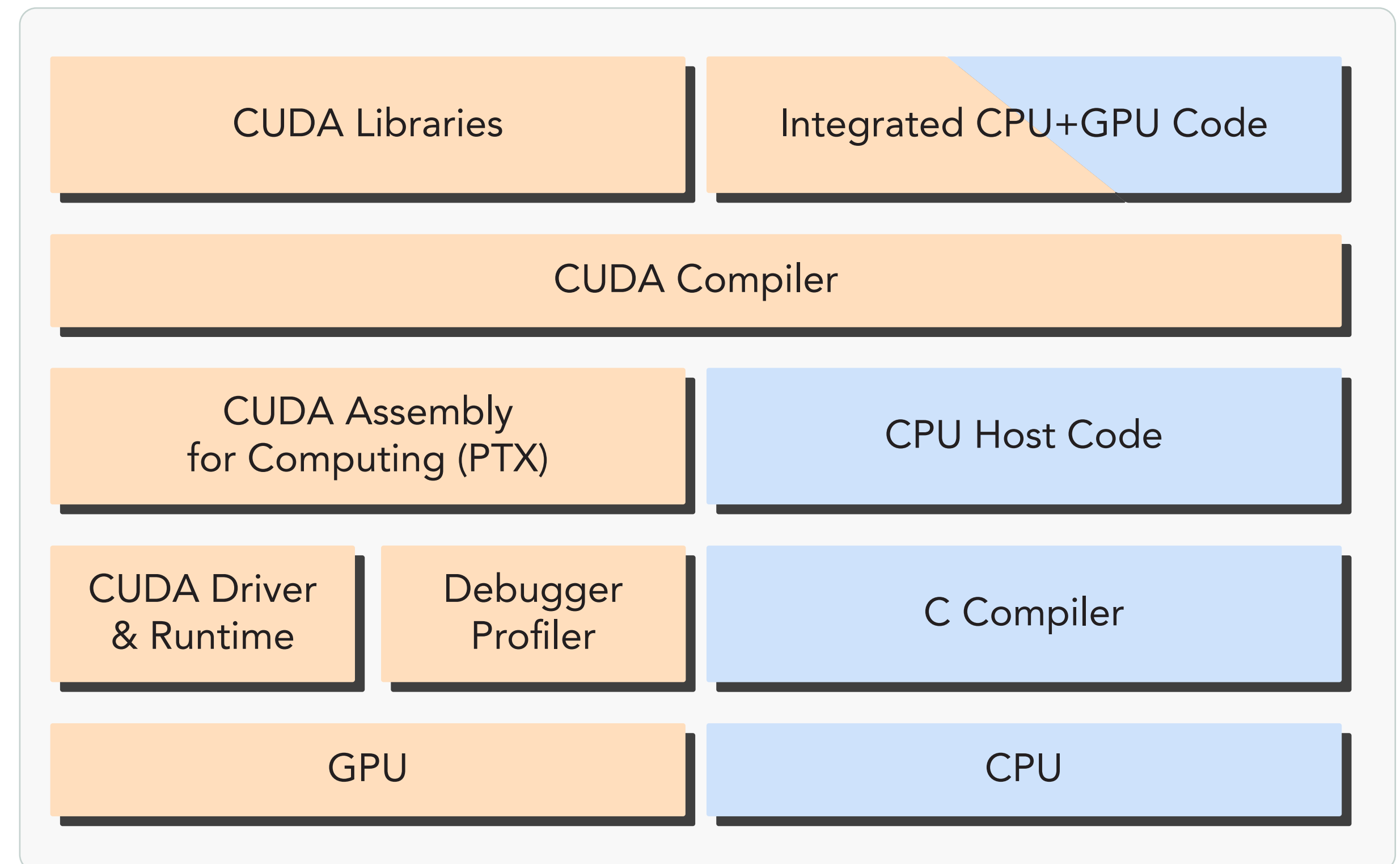
2

The CUDA Toolkit

NVHPC Compiler: translate CUDA into optimised machine instructions for NVIDIA GPUs

Libraries: Comprehensive libraries like cuBLAS and cuDNN are provided

Debugging tools: robust debugging tools



NVCC compiler

1

Compilation process

Code for host and device in some.cu file

CUDA compiler separates source code into host and device components

Based LLVM open source compiler infrastructure

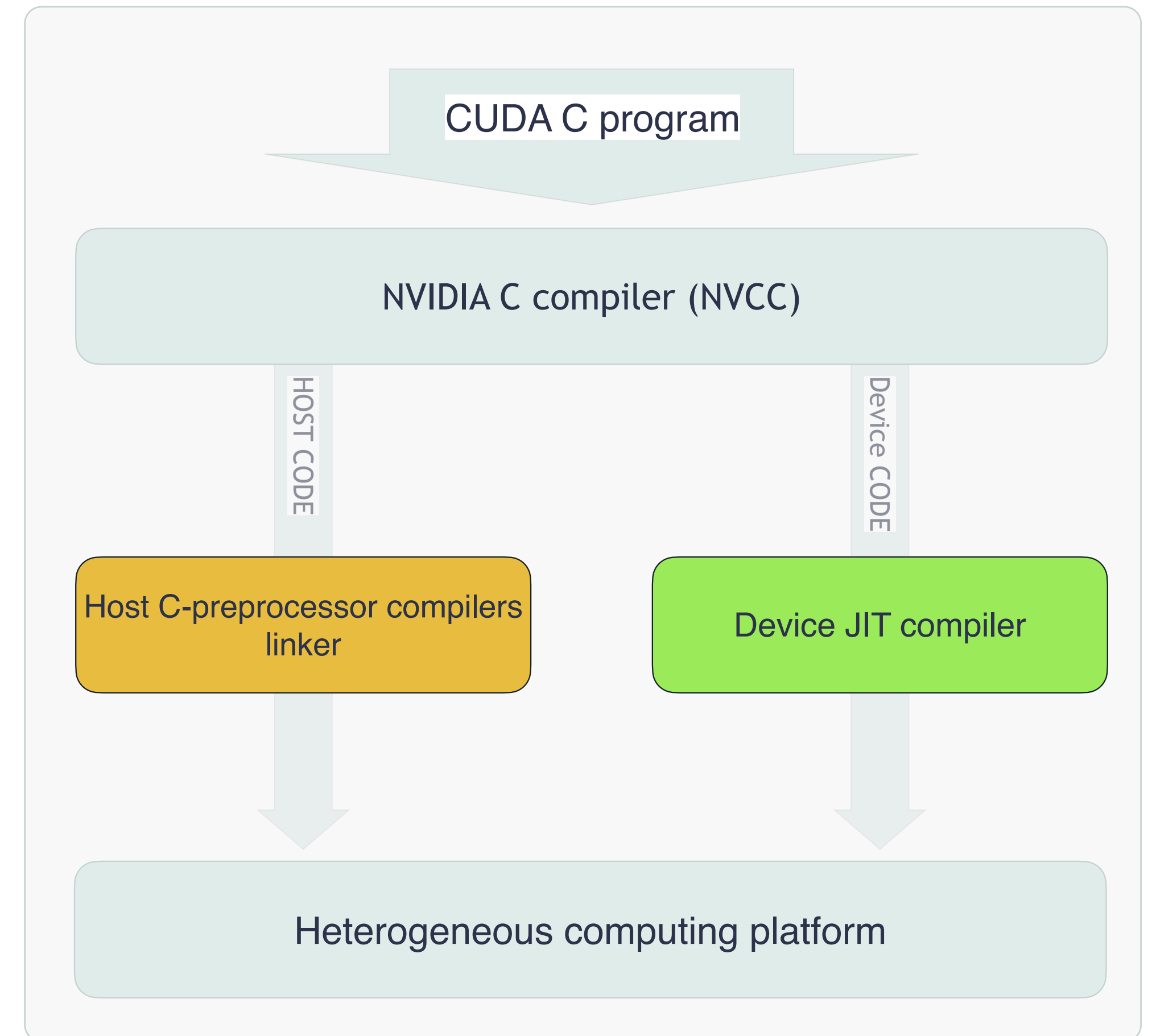
2

`nvcc -arch=sm_80 -o out some-CUDA.cu -run`

- arch: indicates for which architecture the files must be compiled (sm_80 is for TESLA A100 GPU)

- run: execute the successfully compiled binary

- Information on CUDA device: `nvidia-smi`, `deviceQuery`



Code samples for CUDA with CMake

Good support for CUDA projects!

```
cmake_minimum_required(VERSION 3.20)
```

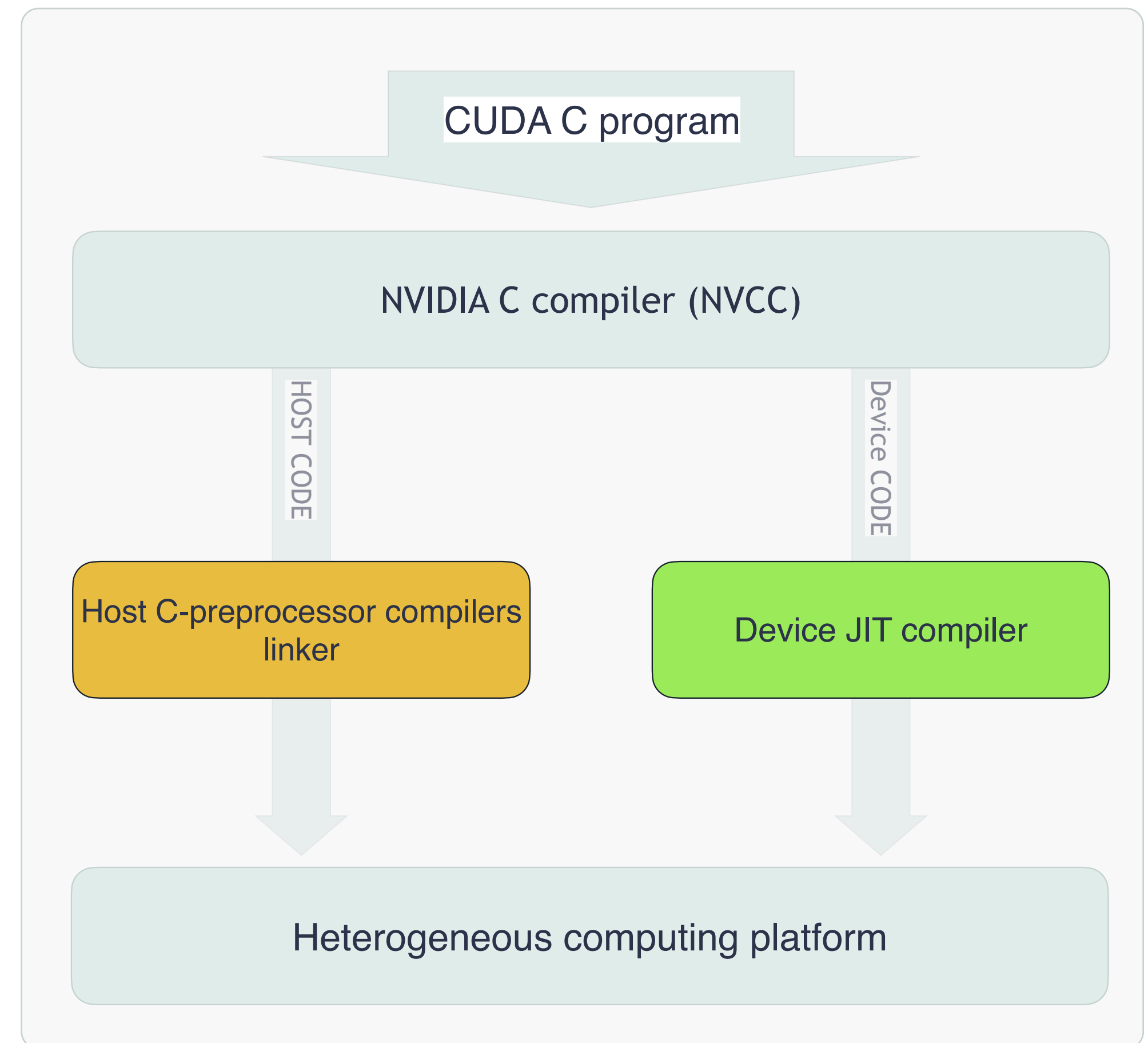
```
project(CUDA_Tutorial LANGUAGES CXX CUDA)
```

```
add_executable(gpuArray gpuArray.cu)
```

```
target_include_directories(gpuArray PRIVATE $  
{CMAKE_CURRENT_SOURCE_DIR})
```

Features

- Variables for CUDA Toolkit, targeted architectures
- E.g., CMake 3.20: automatically detects default GPU architecture that NVCC builds for



 Checkpoint-1

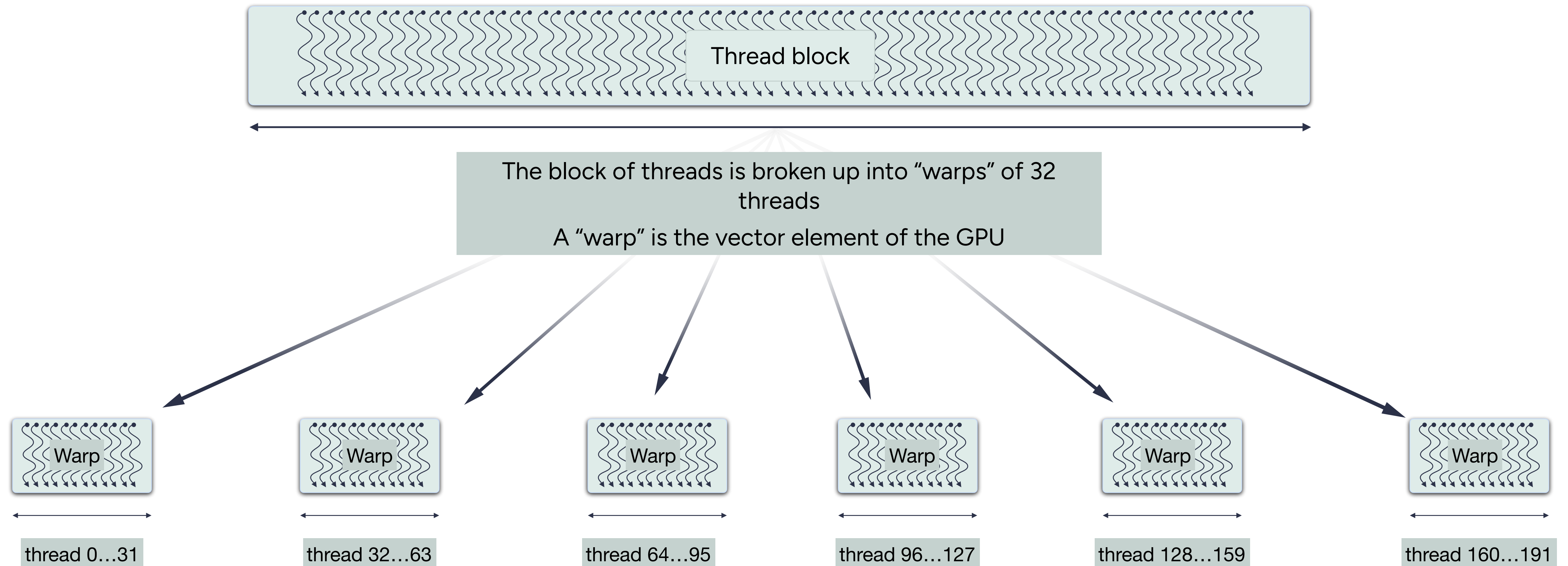
Things to do

In this exercise, initialize an array using managed memory and process it on the GPU:

- T1. Structure the function to call the GPU function.
- T2. Allocate managed memory (accessible by both CPU and GPU)
- T3. Launch kernel with a single block of threads
- T4. Wait for GPU to finish before proceeding
- T5. Free the managed memory

Launching Parallel Kernels

CUDA launches arrays of parallel threads



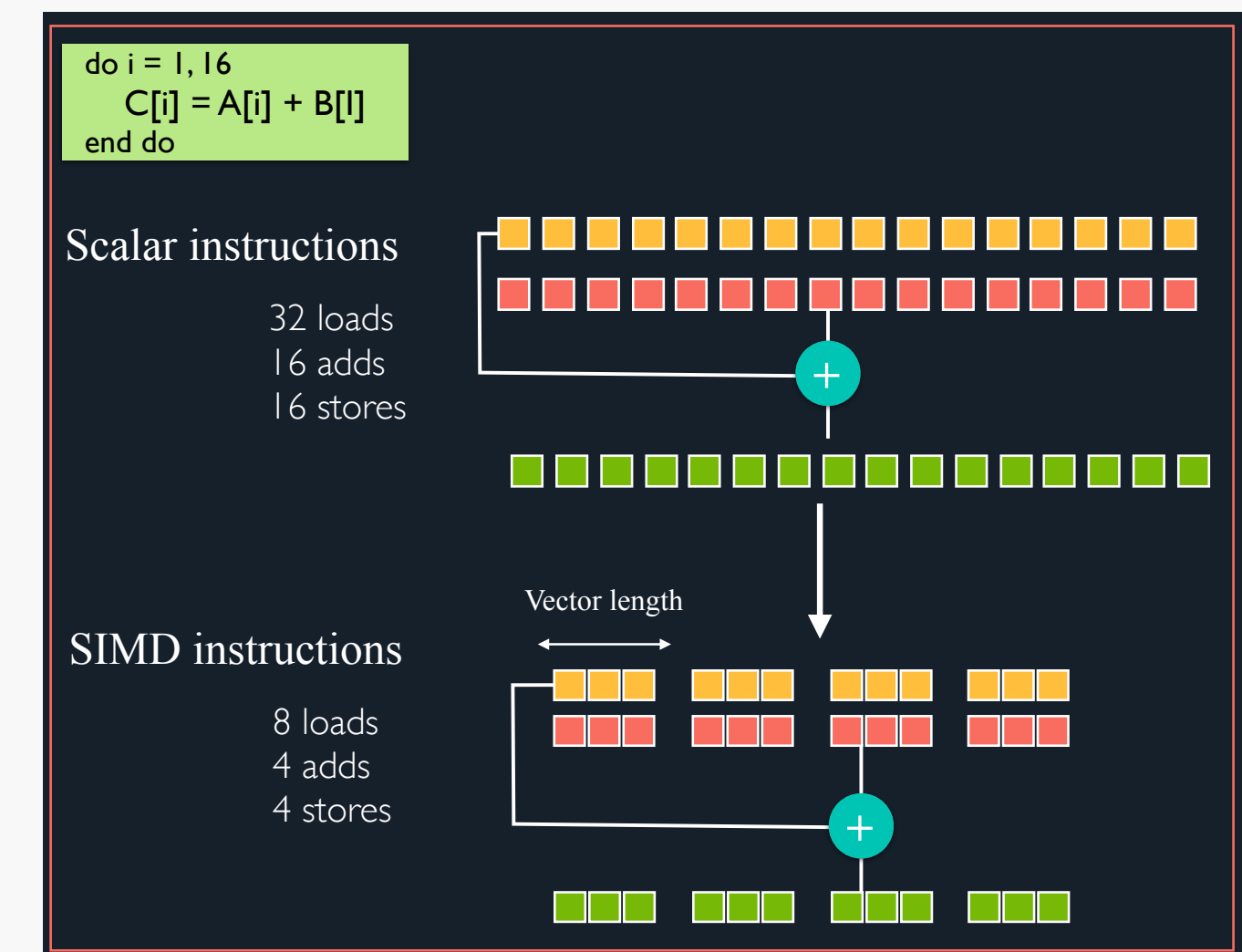
SIMT VS. SIMD execution model

Both SIMD and SIMT achieve parallelism by broadcasting a single instruction to multiple execution units

Consider how computations will be distributed between threads for the following loop (N >> threads count):

```
float *A, *B, *C = ... ; for (int I = 0; I < N; I++ ) A[I] = B[I] + C[I]
```

- SIMD describes a class of instructions which perform the same operations on multiple registers simultaneously
- Converting an algorithm to use SIMD is usually called “Vectorizing”
- a **SIMD register** (or a **vector register**) can hold many values (2 - 16 values or more) of a single type
- Vectorisation helps you write code which has good access patterns to maximise bandwidth



SIMT VS. SIMD execution model

Both SIMD and SIMT achieve parallelism by broadcasting a single instruction to multiple execution units

A loose extension of SIMD which is what CUDA’s computational model is, although there is key differences

- Single instruction, multiple registers
- Single instructions multiple addresses
i.e. parallel memory access!
- Single instruction, multiple flow paths
if statements are allowed!

SIMT allows

- CUDA GPU to perform “vector” computations on *scalar cores*
- Much easier to vectorise than getting compiler to autovectorize on CPU

<https://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html>

SIMT thread registers			
a[l]	a[l+1]	a[l+2]	a[l+3]
b[l]	a[l+1]	b[l+2]	b[l+3]
a	a	a	a
b	b	b	b
l	l+1	l+2	l+3
...

SIMT VS. SIMD execution model

Both SIMD and SIMT achieve parallelism by broadcasting a single instruction to multiple execution units

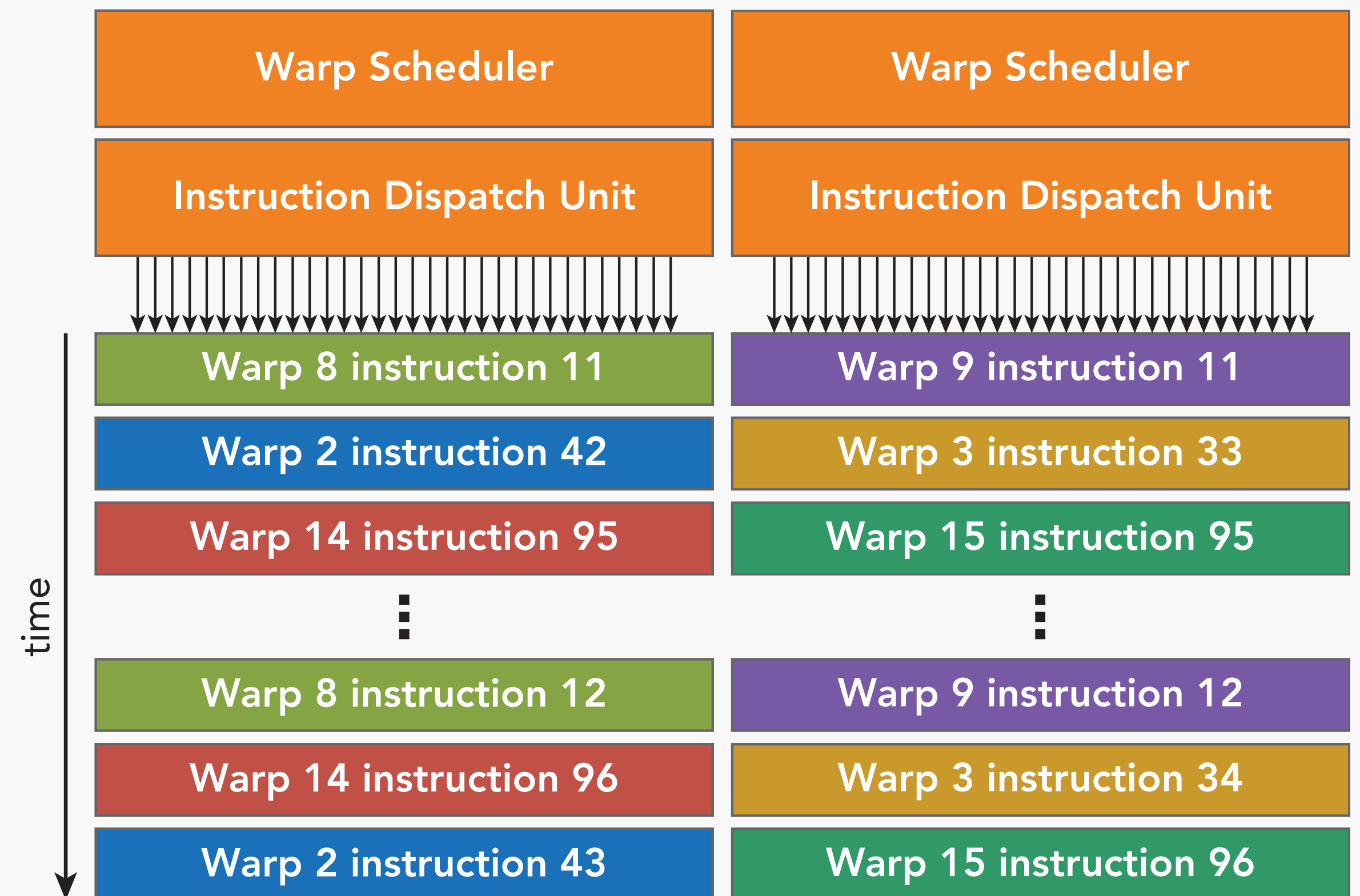
Feature	SIMD	SIMT
Architecture	Traditional CPUs	Utilized by NVIDIA GPUs
Execution Unit	Multiple data lanes	Multiple threads (warps)
Flexibility	Low	High
Branch Handling	No support for divergence	Supports thread divergence
Best Suited For	Homogeneous data operations	Dynamic control flow applications
Common Usage	CPU computing	Vector processing on GPUs

 What is warp, and why is it important?

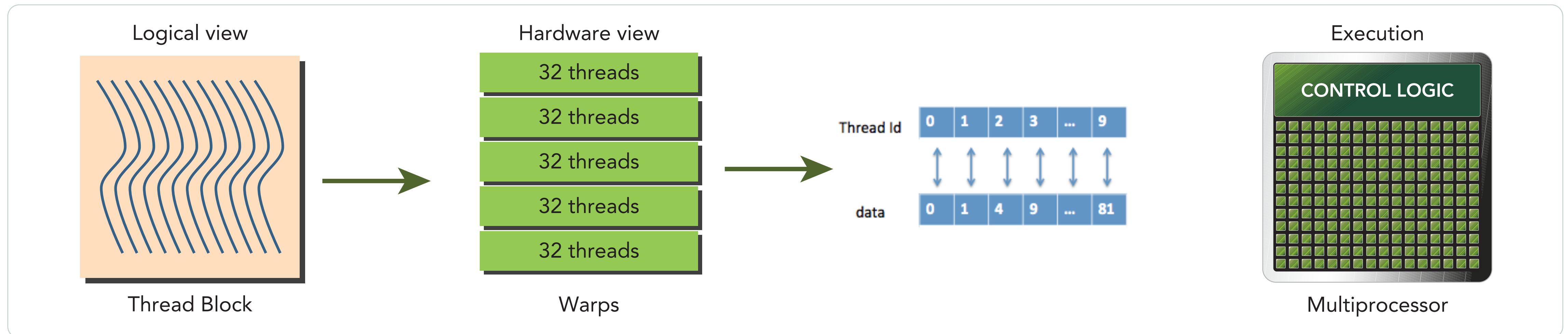
What is WARP?

Hardware Multithreading

- NVIDIA SM schedules threads in warps (groups of 32 threads)
- Warp simply means a group of threads that are scheduled together to execute the same instructions in lockstep.
- Execution context stays on chip
- No overhead for switching warps
- Volta SM has 4 warp schedulers, each one is responsible for
 - feeding 32 CUDA cores
 - 8 load/store units
 - 8 special functions unit



Warps as Scheduling Units



Groups (vectors) of 32 consecutive threads of a block that are executed in parallel in hardware

- An implementation technique, not part of the CUDA programming model
- basic unit of execution in an SM

Warp 0: thread 0, thread 1, thread 2, ... thread 31
Warp 1: thread 32, thread 33, thread 34, ... thread 63
Warp 3: thread 64, thread 65, thread 66, ... thread 95
Warp 4: thread 96, thread 97, thread 98, ... thread 127

Why do we need to have so many warps in an SM?

Latency hiding

- **Memory Access Latency:** Multiple warps can hide memory access latency by switching to another ready warp when one warp is waiting for data
- **Instruction Pipeline Latency:** Keeps the execution units busy while other warps are stalled due to dependencies or resource constraints

Resource Utilisation

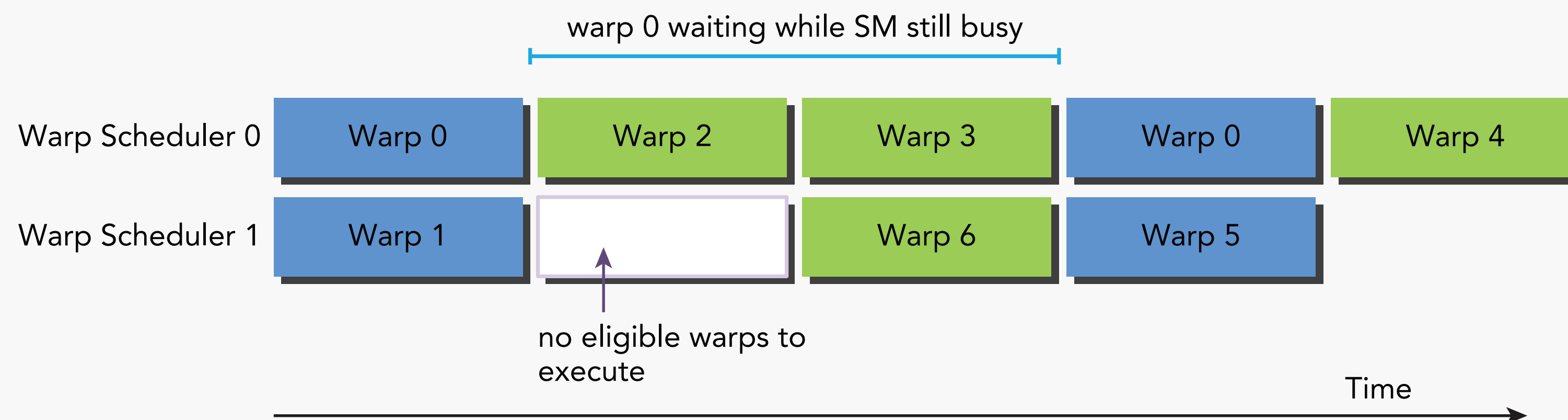
- **Maximizing Throughput:** More warps allow for better utilization of SM resources (ALUs, memory bandwidth)
- **Load Balancing:** Distributes the workload evenly across the available execution units

Parallelism

- **Enhancing Parallel execution:** Multiple warps increase the parallelism, enabling more threads to be processed concurrently
- **Improved Performance:** Higher parallelism leads to better performance and throughput for data-intensive applications

GPUs are designed to hide latency

- **Latency** is the number of clock cycles needed to complete an instruction, aka the number of cycles we need to wait for before another dependent operation can start
 - Arithmetic latency (~18-24 cycles)
 - Memory access latency (~400 -800 cycles)
- It can't be discarded (**hardware limitation**), but its effect can be controlled (**hidden**) by
 - Saturating computational pipelines in computational bound problems
 - Saturating band width in memory bound problems



Active block: when compute resources such as registers, shared memory has been allocated

Active warp: warp it contains are called. Active warps can be classified:

- Selected warp
- Stalled warp
- Eligible warp

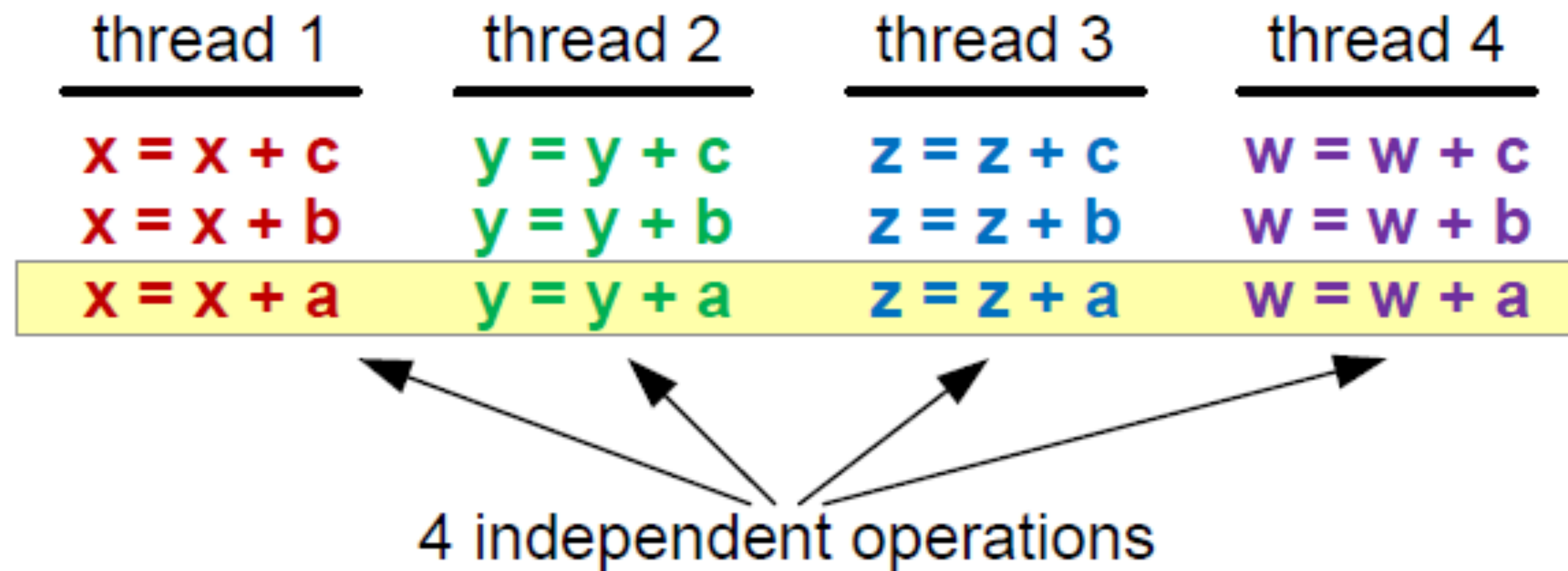
Latency can be hidden in two possible ways

- The code need to be organised to **provide the scheduler a sufficient number of independent operations**
 - i.e. the more warps are available, the more context-switch can hide latency
 - And therefore proceed with other useful operations
- There are two possible ways and paradigms to use (**can be combined too!**)
 - Thread-Level Parallelism (TLP)
 - Instruction-Level Parallelism (ILP)

More concurrently eligible threads

- **Thread-level parallelism (TLP)**

- ▶ Strive for high SM occupancy: provide as much threads as SM as possible (when a scheduler is free it will find easily a warp to execute)
- ▶ Best approach for low number of independent operations per CUDA kernel



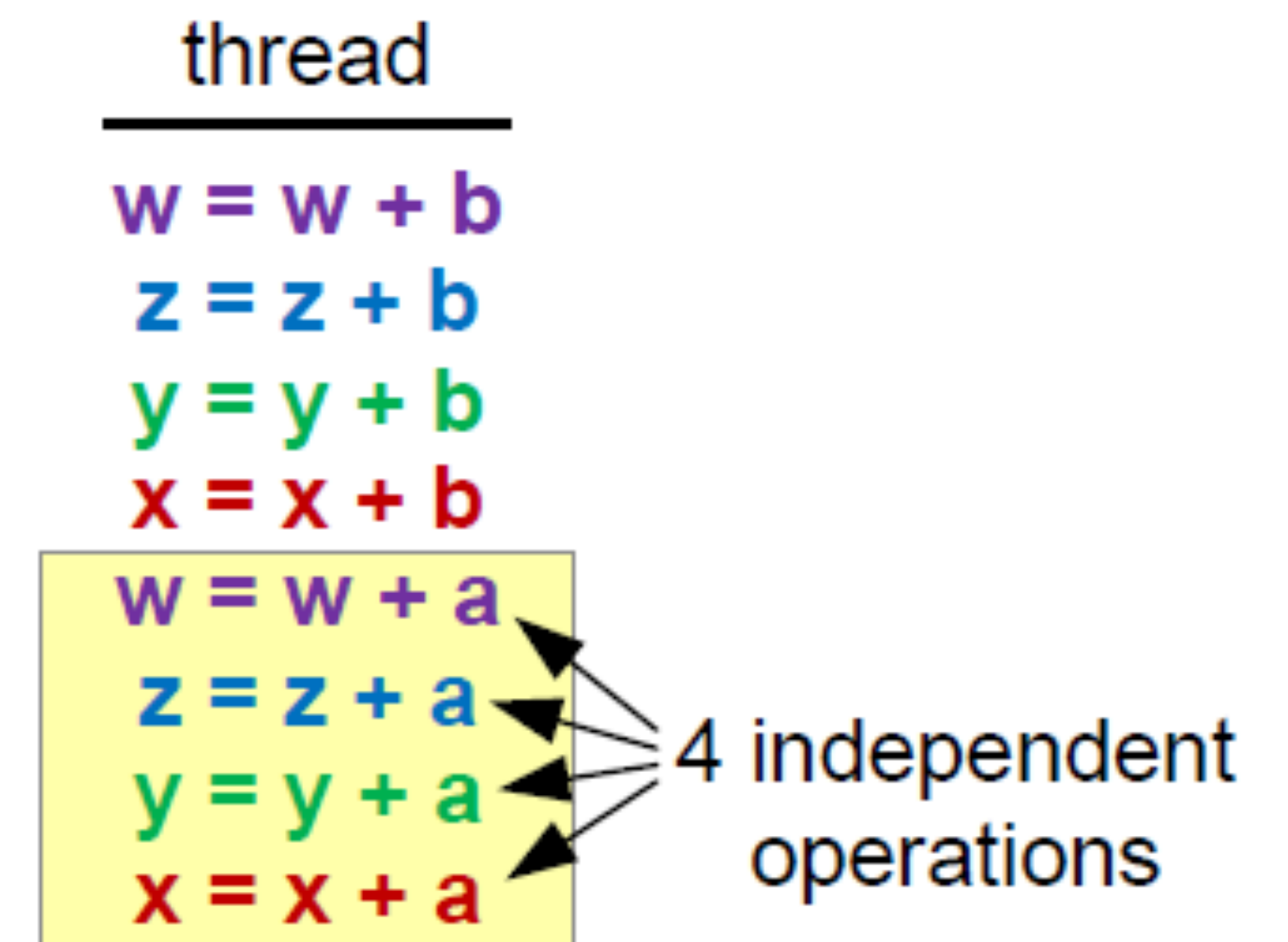
More concurrently eligible threads

- **Instruction-level parallelism (ILP)**

- ▶ High-number of multiple independent operations inside CUDA kernel (each kernel act on a lot of data)
- ▶ This will grant the scheduler to stay on the same warp and fully load each hardware pipeline

- **NOTE**

- ▶ The scheduler will not select a new warp until there are eligible instructions ready to execute on the current warp

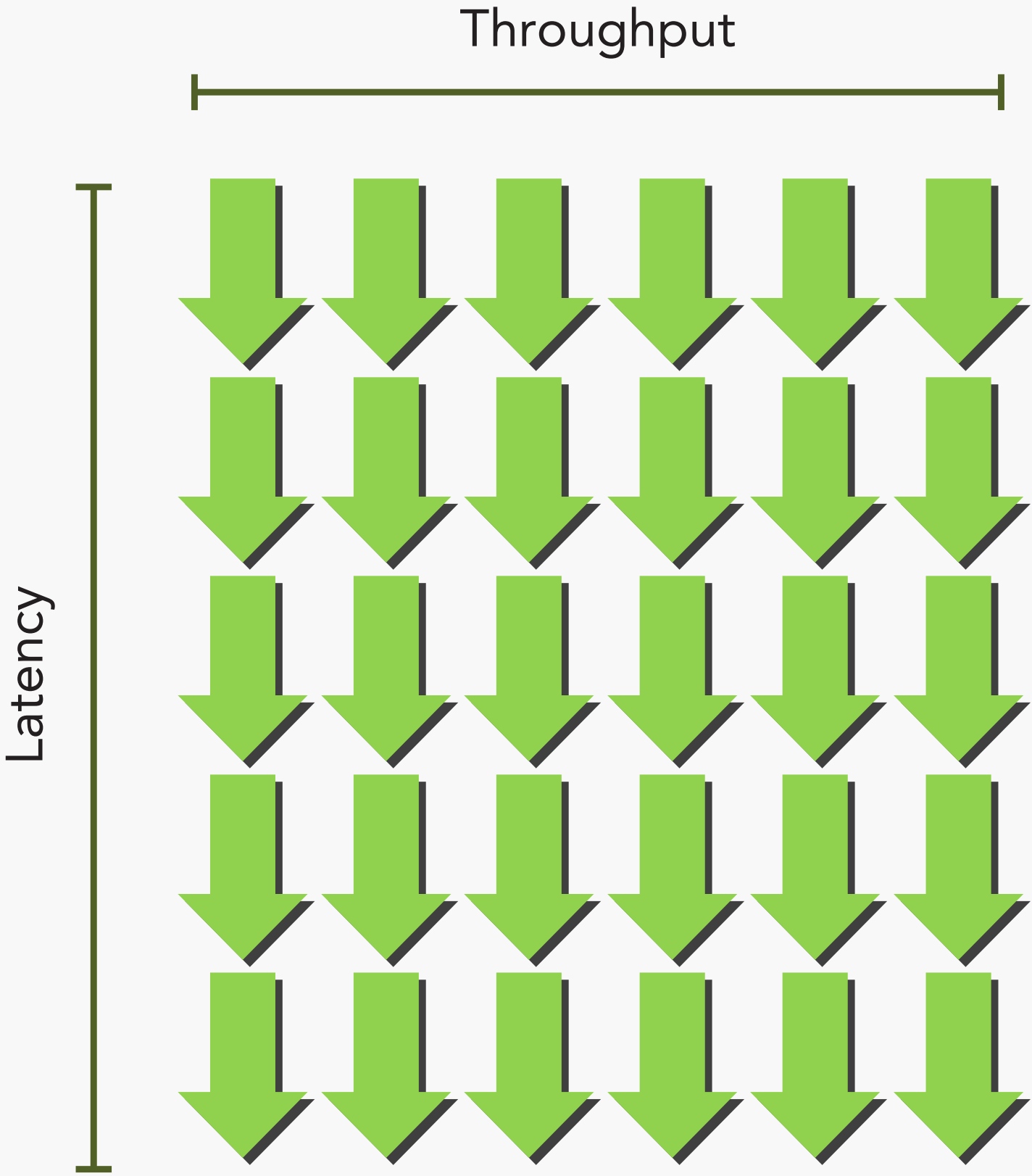


Throughput and Bandwidth

How to estimate the number of active warps required to hide latency

$$\text{Number of Required Warps} = \text{Latency} \times \text{Throughput}$$

Bandwidth = "What's possible"	Throughput = "What's achieved"
Theoretical peak data transfer rate (e.g., GPU memory ↔ cores)	Achieved rate of useful work (e.g., instructions, operations, or data processed per second).
Focused on data movement capacity (e.g., memory, PCIe bus)	Can measure computational efficiency (e.g., FLOPS, IPC).
Hardware-limited (e.g., GDDR6X specs, bus width)	Depends on software/hardware synergy (e.g., kernel efficiency, parallelism)



SM Parallelism Required to Maintain Full Arithmetic Utilization

FP32 FMA on A100 vs. Older Architectures

GPU Architecture	FP32 FMA Throughput (ops/cycle/SM)	Instruction Latency (cycles)	Required Parallelism
Fermi (2010)	32 ops/cycle/SM	~20 cycles	~ 640 ops
Kepler (2012)	192 ops/cycle/SM	~18 cycles	~ 3456 ops
Ampere (A100)	256 ops/cycle/SM	~4 cycles	~ 1024 ops

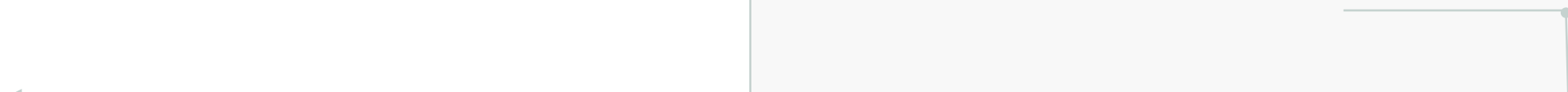
CUDA Kernels

The error is expected:

- You can not launch more than 1024 threads
- ... in a thread block, which is a HW limit

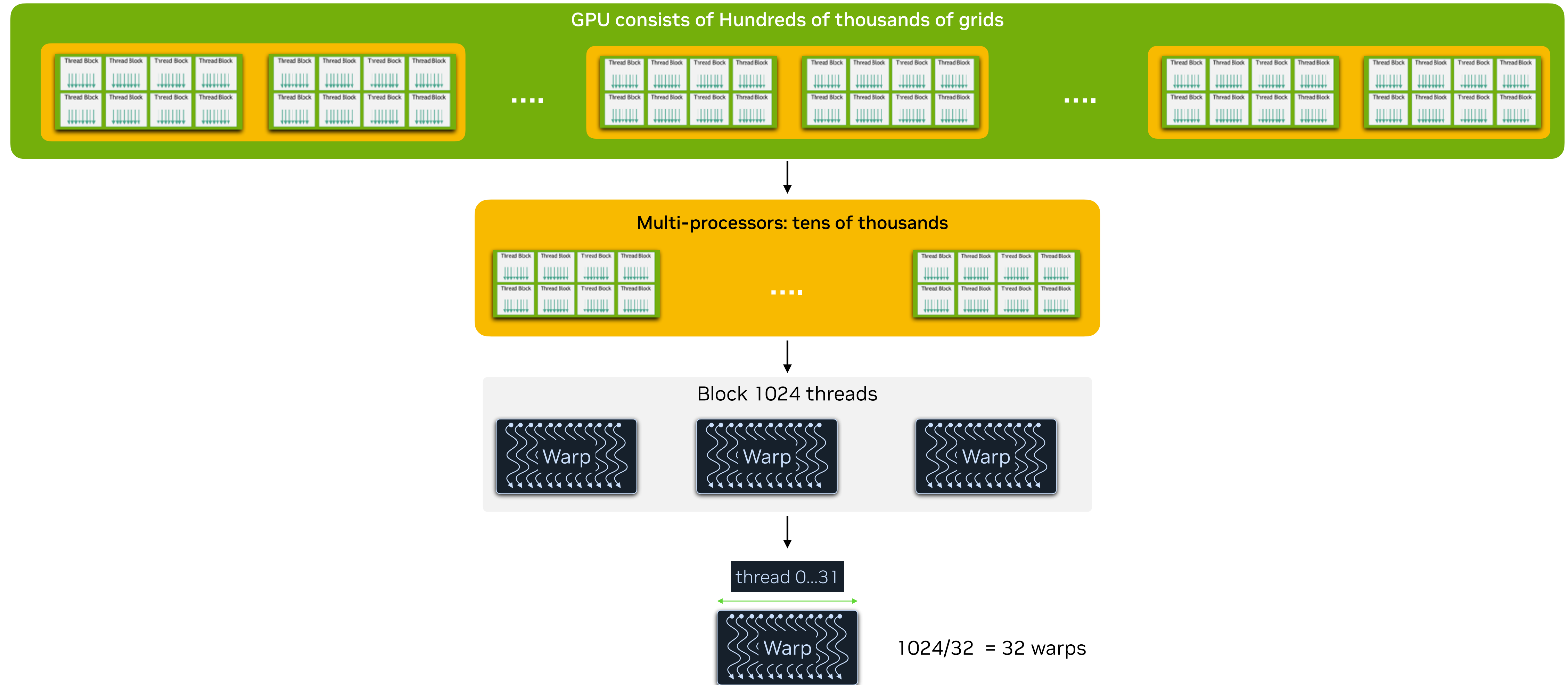
Warning: Trying to launch 2048 threads, which exceeds the maximum allowed per block (1024 threads)

```
__global__ void processArray(int *arr, int N) {  
    // Get the index of the thread  
    int idx = threadIdx.x;  
  
    // Ensure the index is within bounds  
    if (idx < N) {  
        arr[idx] *= 2; // Multiply each element by 2  
    }  
}  
  
int main() { ...  
    // Launch kernel with a single block of threads  
    processArray<<< 1, number_of_threads >>>(arr, N);  
}
```

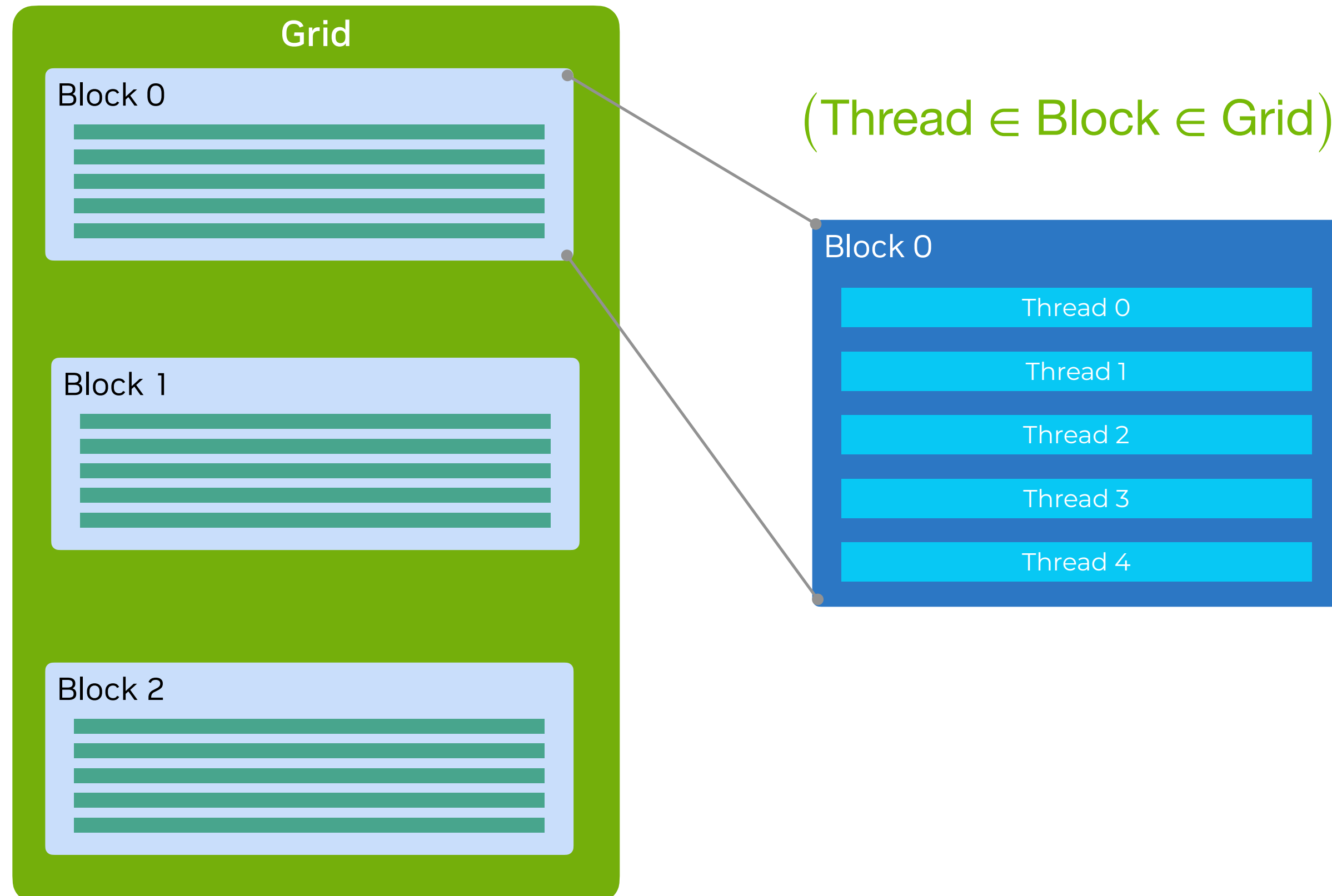


GPU Thread hierarchy

GPU Thread hierarchy



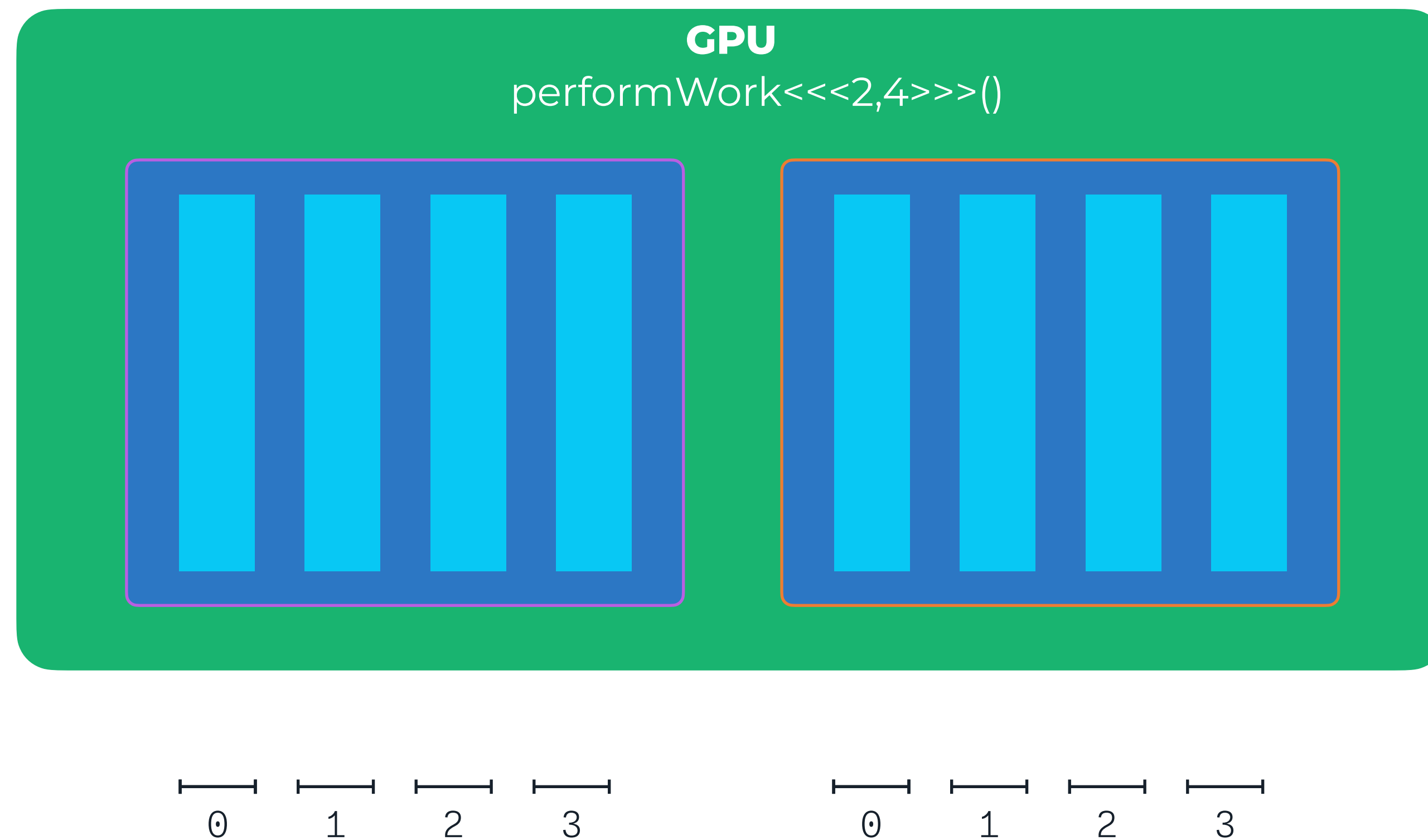
Kernel execution across Thread, Block, and Grid



- In order to compute N elements on the GPU in parallel, at least N concurrent threads must be created on the device
- GPU threads are grouped together in teams or blocks of threads
- Threads belonging to the same block or team can cooperate together exchanging data through a shared memory cache area
- Each block of threads will be executed independently
- No assumption is made on the blocks execution order

Kernel execution across Thread, Block, and Grid

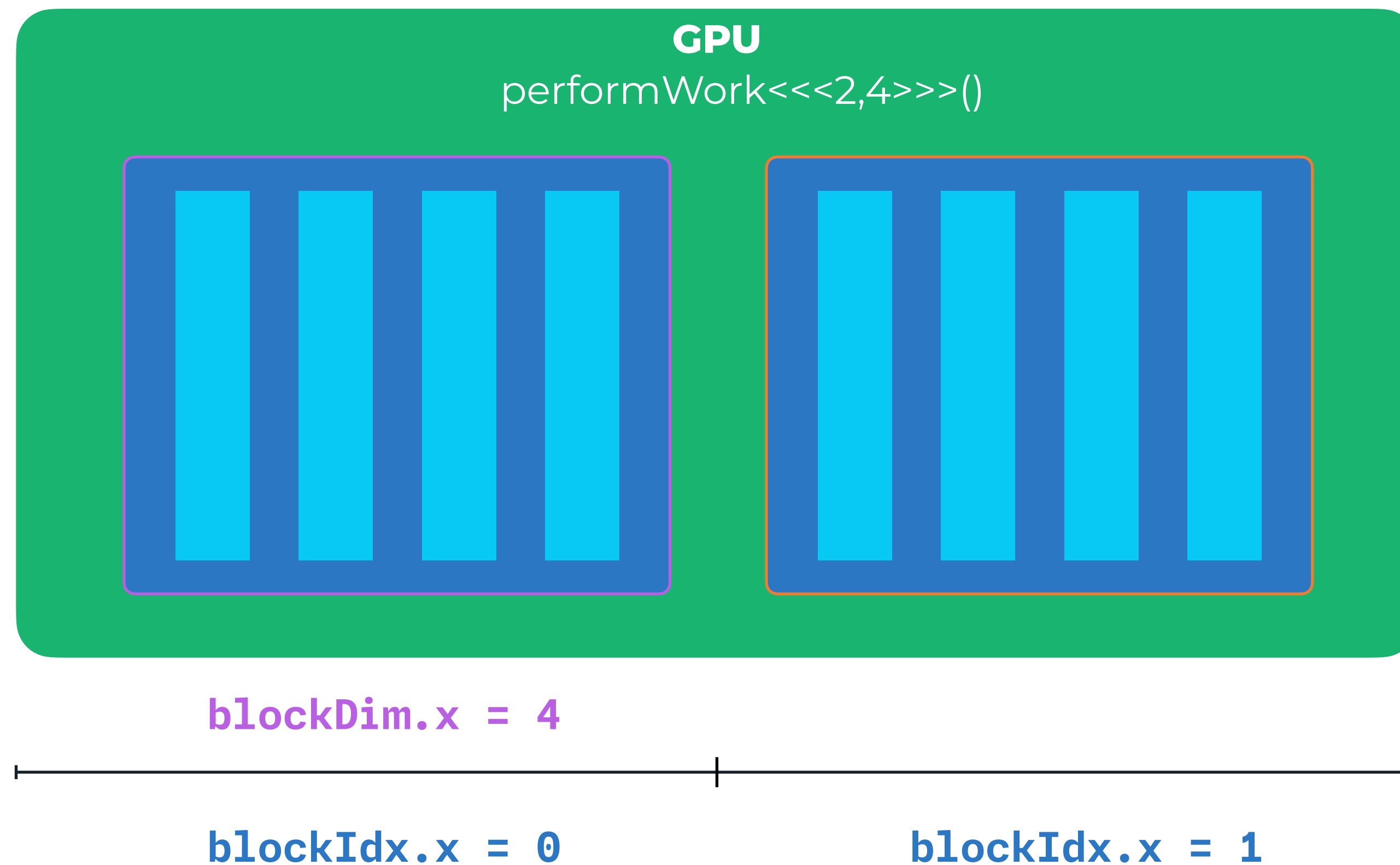
`threadIdx.x`: index of the thread with a block



Kernel execution across Thread, Block, and Grid

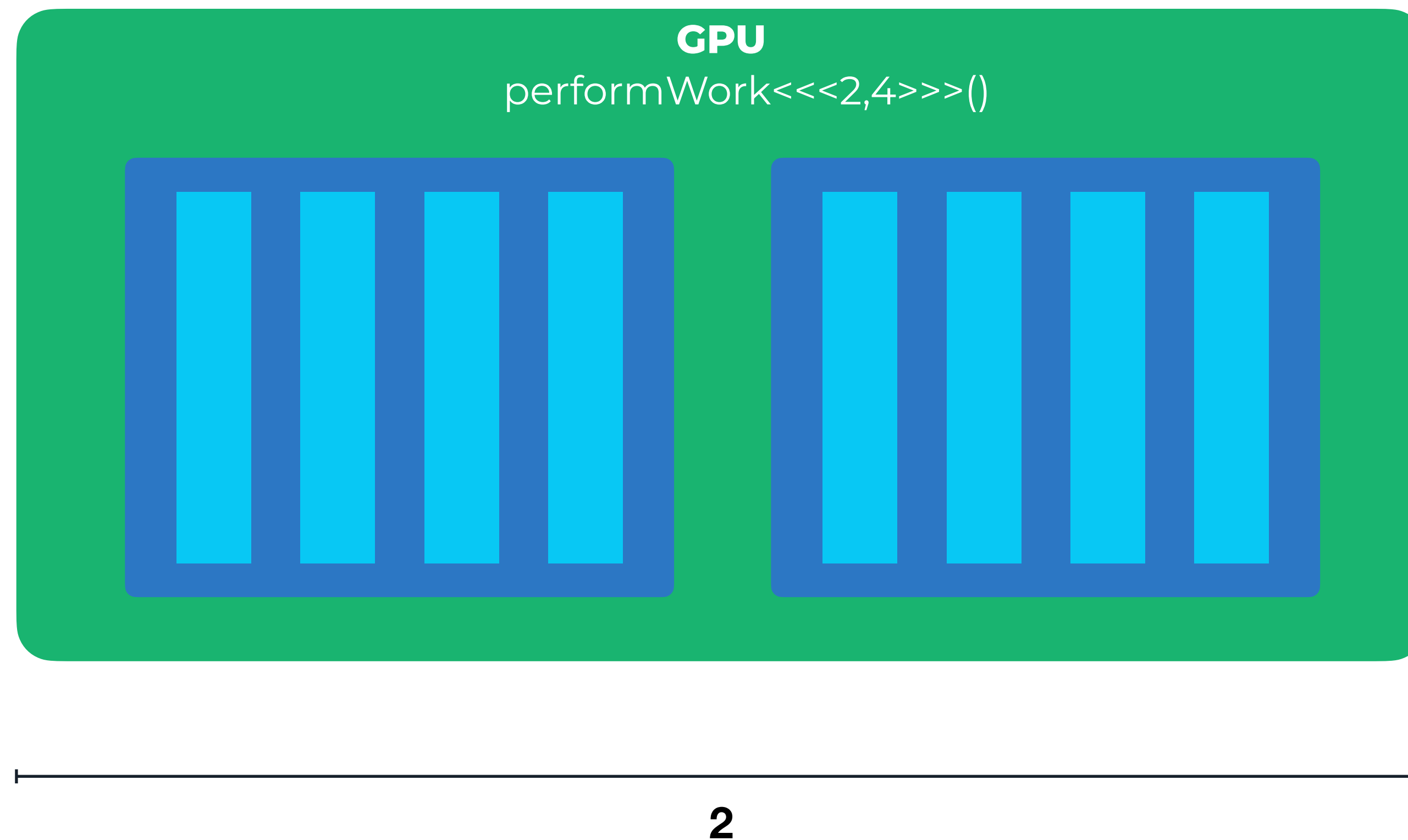
blockIdx.x: index of a blocks in a grid

blockDim.x: number of threads per block



Kernel execution across Thread, Block, and Grid

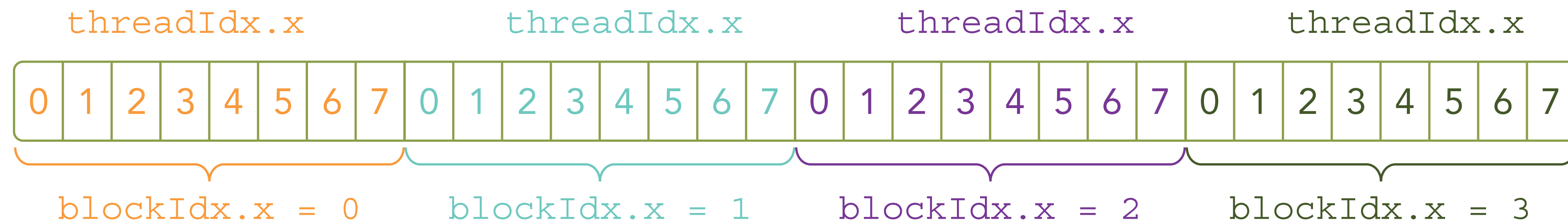
gridDim.x: number of blocks in the grid, in this case 2



Kernel execution across Thread, Block, and Grid

Choose the optimal block size

- A **limited number of threads (1024)** can fit inside a thread block
- To increase parallelism, we need to **coordinate** work **among thread blocks**.
- This is achieved by **mapping** element of data vector to threads using **global index = threadIdx.x + blockIdx.x*blockDim.x**

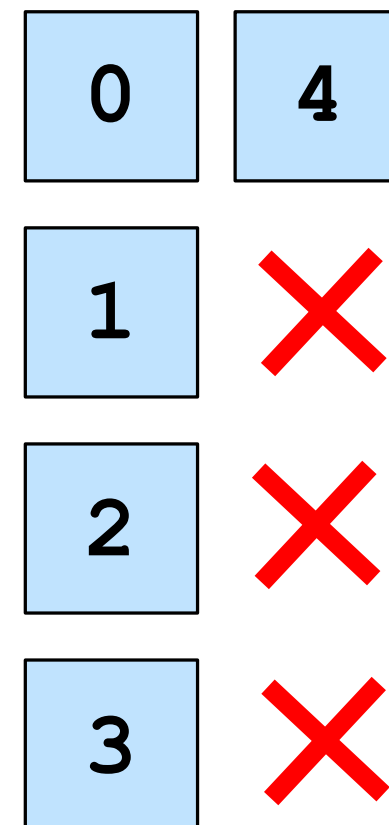


```
for blockIdx.x = 0  
  i = 0 * 8 + threadIdx.x = { 0, 1, 2, ... , 7 }
```

```
for blockIdx.x = 3  
  i = 0 * 8 + threadIdx.x = { 0, 1, 2, ... , 7 }
```

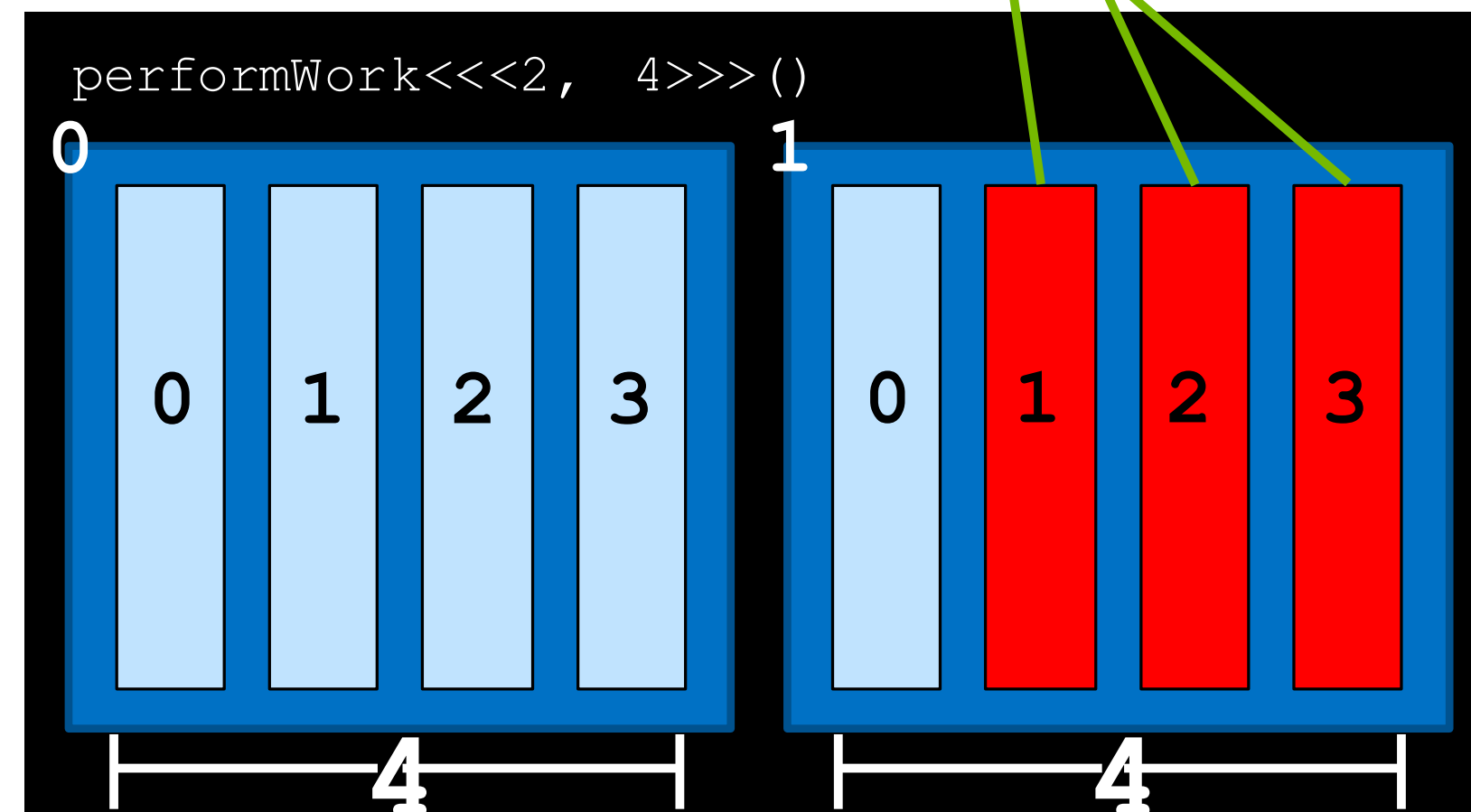
Grid size larger than data set

GPU
DATA



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

GPU



Choosing the optimal grid size

Choose the optimal block size

- best performance for blocks that contain a number of threads that is a **multiple of 32**, due to GPU hardware traits
- Example: we need to run 1000 parallel task with blocks containing 256 threads. How do we choose the optimal block size?

```
int N = 100000; size_t threads_per_block = 256;

size_t number_of_blocks = (N + threads_per_block - 1) / threads_per_block;

kernel<<<number_of_blocks, threads_per_block>>>(N);
```

- This calculation ensures that the number of blocks is sufficient to cover all the threads, even if the total number of threads is not evenly divisible by the threads per block
- The “-1” term is added to round up the division if necessary

Choosing the optimal grid size

Choose the optimal block size

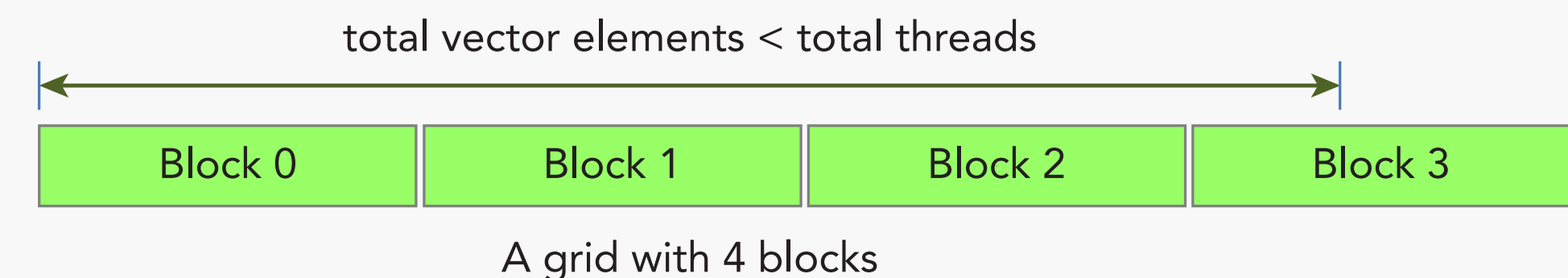
- Write an execution configuration that creates more threads than necessary
- Pass a value as an argument into the kernel (N) that represents that total size if the data set to be processed/total threads needed to complete the work
- Calculate the global index and if it does not exceed N perform the kernel work

```
// Coalesced access example
__global__ vectorSum(int N)
int idx = threadIdx.x + blockIdx.x * blockDim.x;
{
    if(idx < N){ // only do work if it does}
}
```

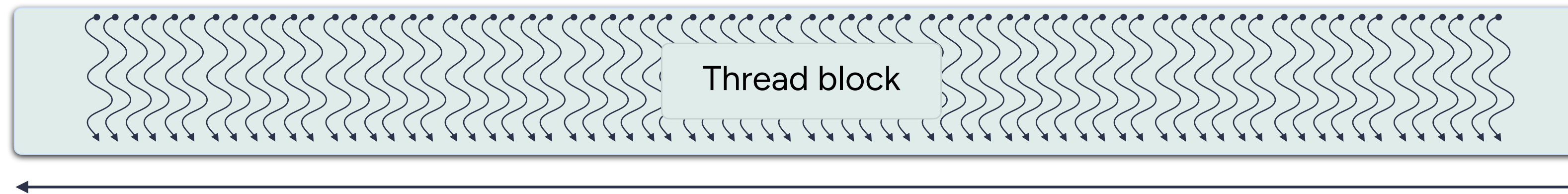
Know your limitations

Maximum size at each level of the thread hierarchy is device dependent. On A100 typical you get :

- Maximum number of threads per block : 1024
- Maximum sizes of x-, y-, and -z dimensions of threads block 1024 x 1024 x 64
- Maximum sizes of each dimension of grid of thread blocks: 65535 x 65535 x 65535 (about 280,000 billion blocks)



Every thread runs exactly the same program



A **limited number of threads (1024)** can fit inside a thread block



To increase parallelism, we need to **coordinate** work **among thread blocks**



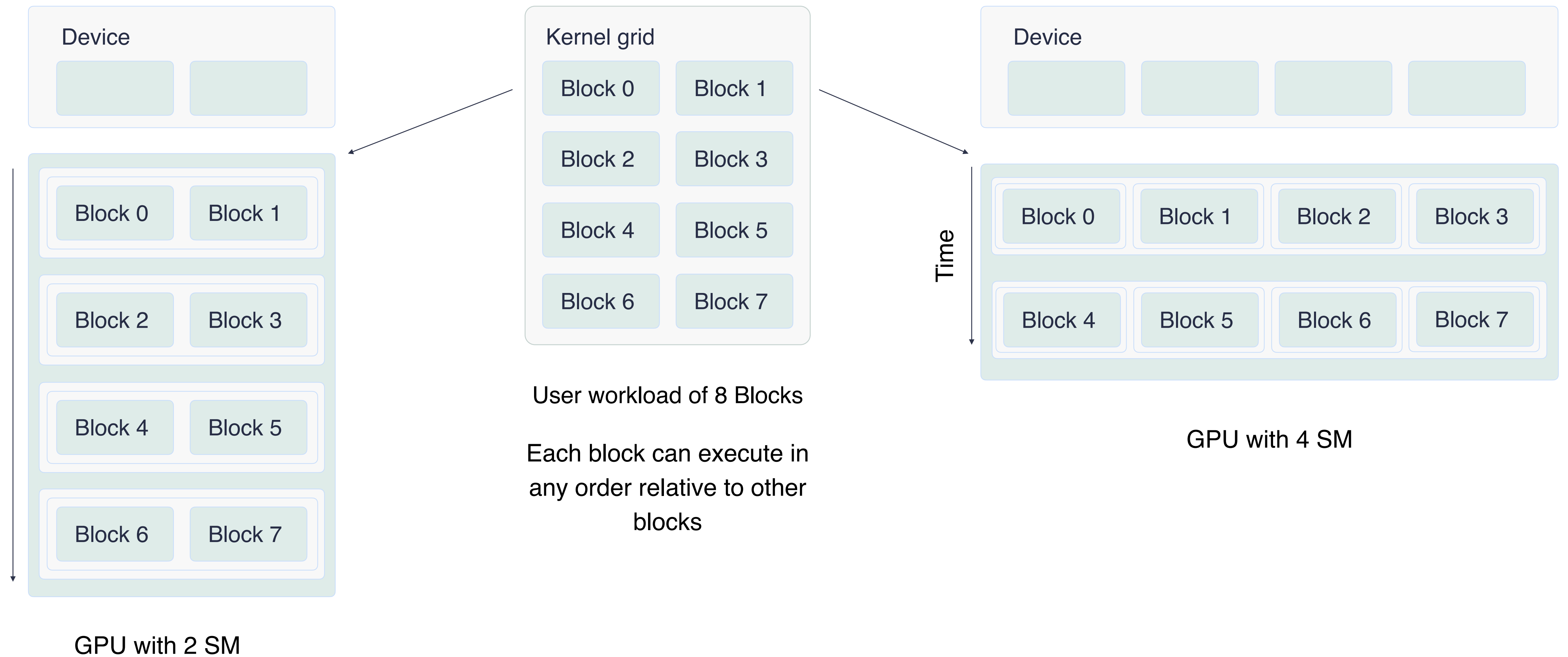
This is achieved by **mapping** element of data vector to threads using **global index**

```
int index = threadIdx.x + (blockIdx.x * blockDim.x)
```



All about this one line code

Transparent scalability



Mapping to hardware

1

CUDA invokes kernel grid

Host kicks off the execution of a kernel grid which contains blocks of threads

2

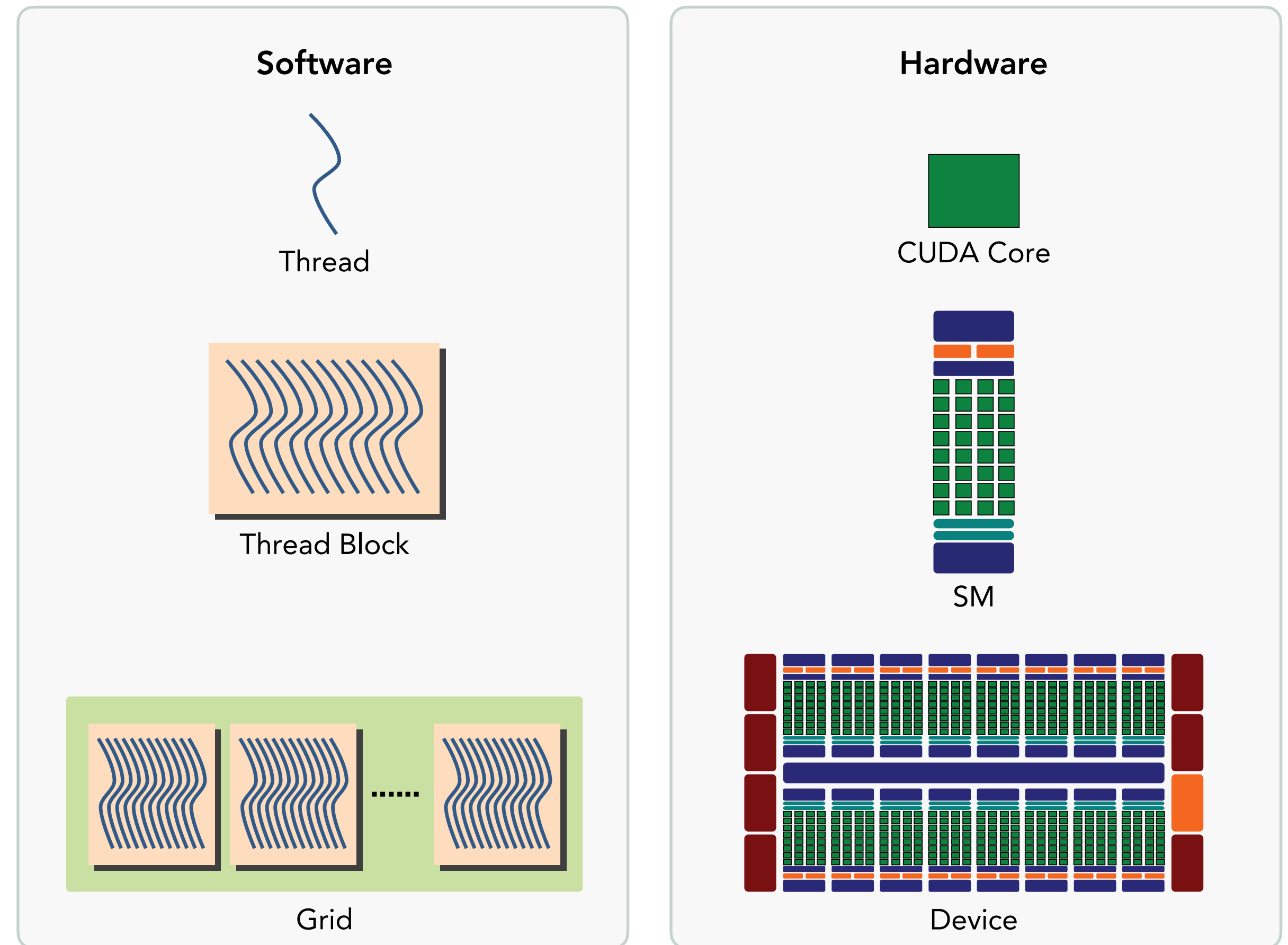
Execute concurrently

Each SM runs multiple thread blocks
Each SP runs on thread from a thread blocks

3

Grid blocks distributed to SMs

Shared cache, register and memory
Global memory shared by all SMs



Checkpoint-1 : Vector Sum

Things to do

The starting code contains a CPU vector addition function. Using what you have learned so far, accelerate the `addVectorsInto` function to run as a CUDA kernel on the GPU and to do its work in parallel. Consider the following guidelines.

- T1. Augment the `vecSum` definition so that it is a **CUDA kernel**.
- T2. Choose a working execution configuration for `vecSum`.
- T3. Use `cudaMallocManaged` to manage the data transfer

 Performance consideration

Timing GPU code

1 Kernels are executed asynchronously

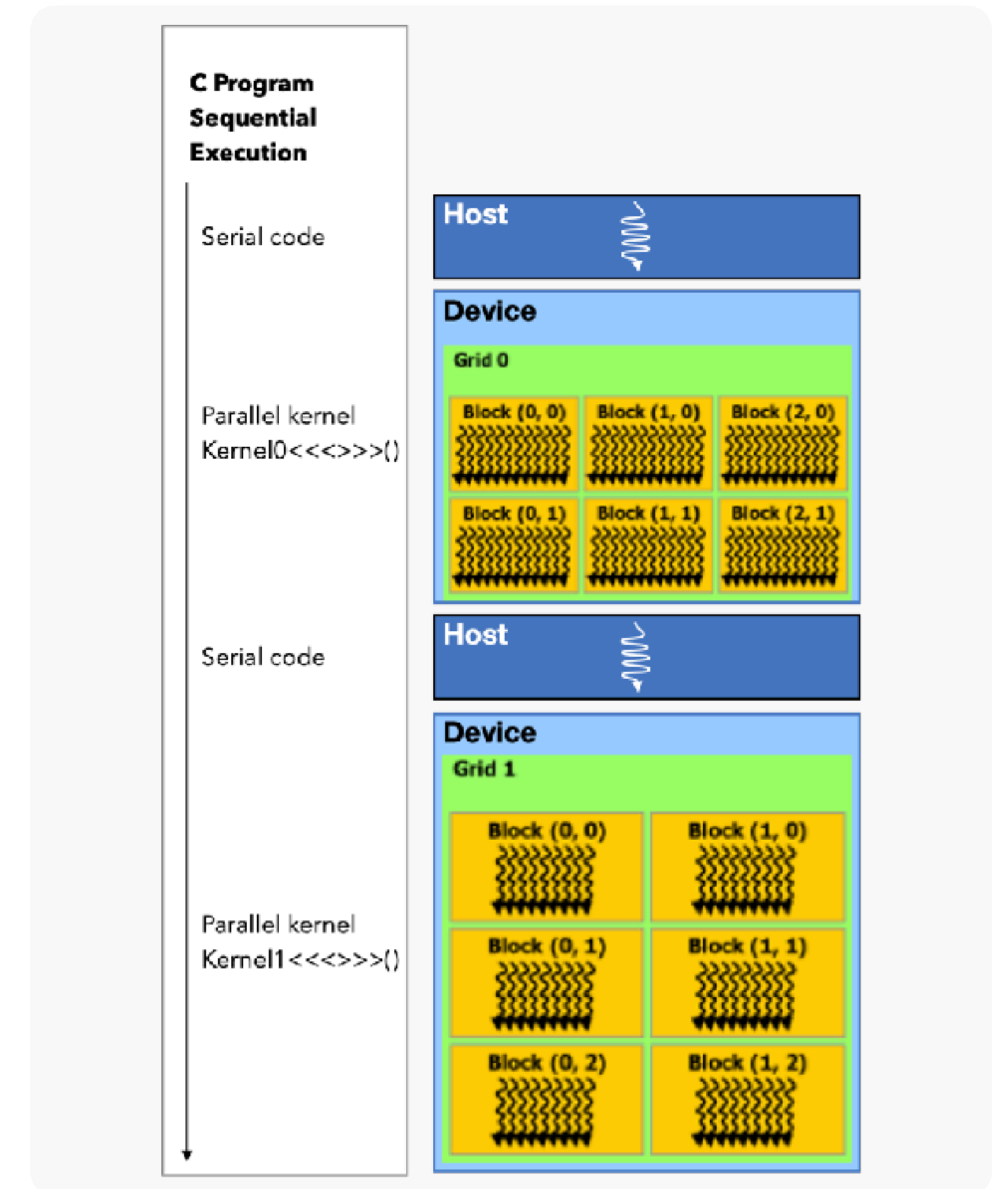
The cpu continues executing while kernels runs

2 Native CUDA events

An event in CUDA is essentially a GPU time stamp that is recorded at a user-specified point in time

3 Use profile

Nsight system, Toal view, Extra-e or SCOREP



Measuring performance with events

How to time code using CUDA events

```
cudaEvent_t start, stop;  
float time;  
cudaEventCreate(&start);  
cudaEventRecord(&stop);  
  
cudaEventRecord( start, 0 );  
kernel<<<grid, threads>>> ( d_odata, d_idata, size_x, size_y, NUM_REPS);  
  
// do some work on the GPU  
cudaEventRecord( stop, 0 );  
cudaEventSynchronize( stop );  
  
cudaEventElapsedTime( &time, start, stop );  
cudaEventDestroy( start );  
cudaEventDestroy( stop );
```

Create events to record

Record events to timestamp

Get time between events

Destroy when done

Validate GPU results by comparing with CPU results

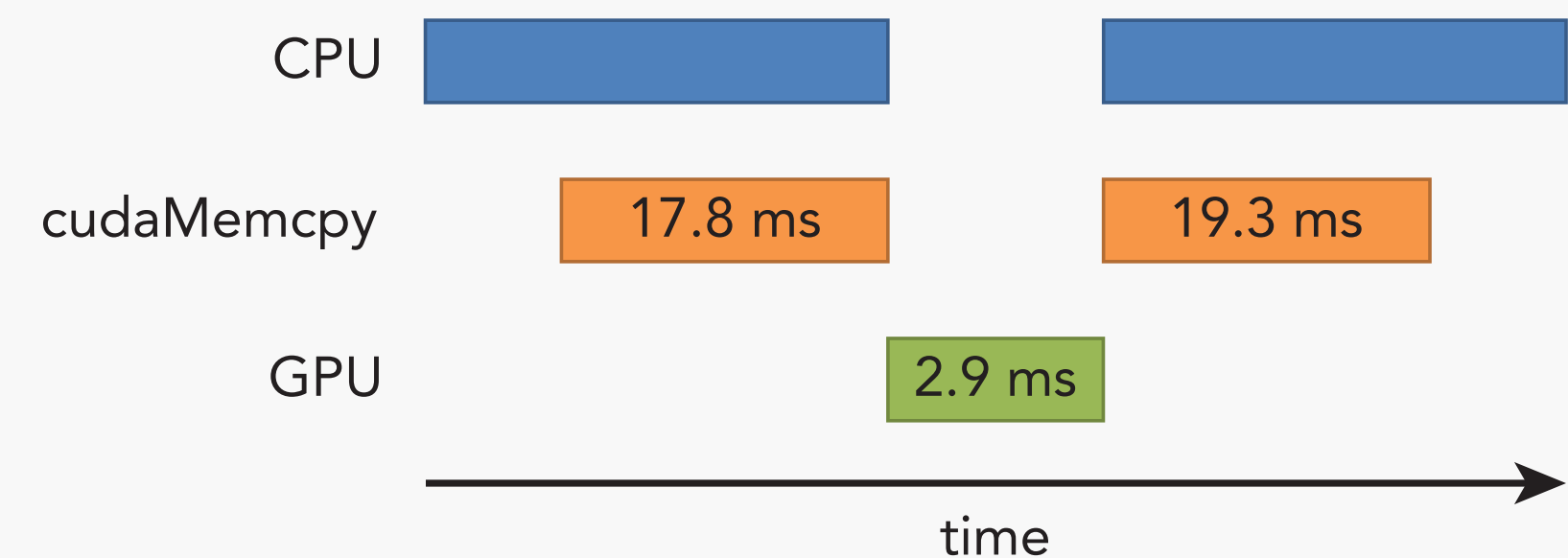
```
// Validate results

bool validateResults(float *hostRef, float *gpuRef, int nElem) {
    bool correct = true;
    for (int i = 0; i < nElem; i++) {
        if (fabs(hostRef[i] - gpuRef[i]) > 1e-5) {
            correct = false;
            printf("Mismatch at index %d: CPU = %f, GPU = %f\n", i, hostRef[i], gpuRef[i]);
            break;
        }
    }

    if (correct) {
        printf("Results match!\n");
    }
    return correct;
}
```

Timing your kernel

```
double cpuSecond() {  
    struct timespec ts;  
    timespec_get(&ts, TIME_UTC);  
    return ((double)ts.tv_sec + (double)ts.tv_nsec * 1.e-9);  
}  
  
/* Measure time for CPU execution */  
double start = cpuSecond();  
sumArraysOnCPU(h_A, h_B, hostRef, nElem);  
double cpuTime = cpuSecond() - start;  
printf("CPU Execution Time: %f seconds\n", cpuTime);  
  
/* Measure time for GPU execution  
double start = cpuSecond();  
sumArraysOnGPU<<<gridSize, blockSize>>>(d_A, d_B, d_C, nElem);  
checkCuda( cudaDeviceSynchronize() ); // Ensure GPU kernel finishes  
double gpuTime = cpuSecond() - start;  
printf("GPU Execution Time: %f seconds\n", gpuTime);
```



Nsight Systems and Compute

Nsight product family

Nsight System

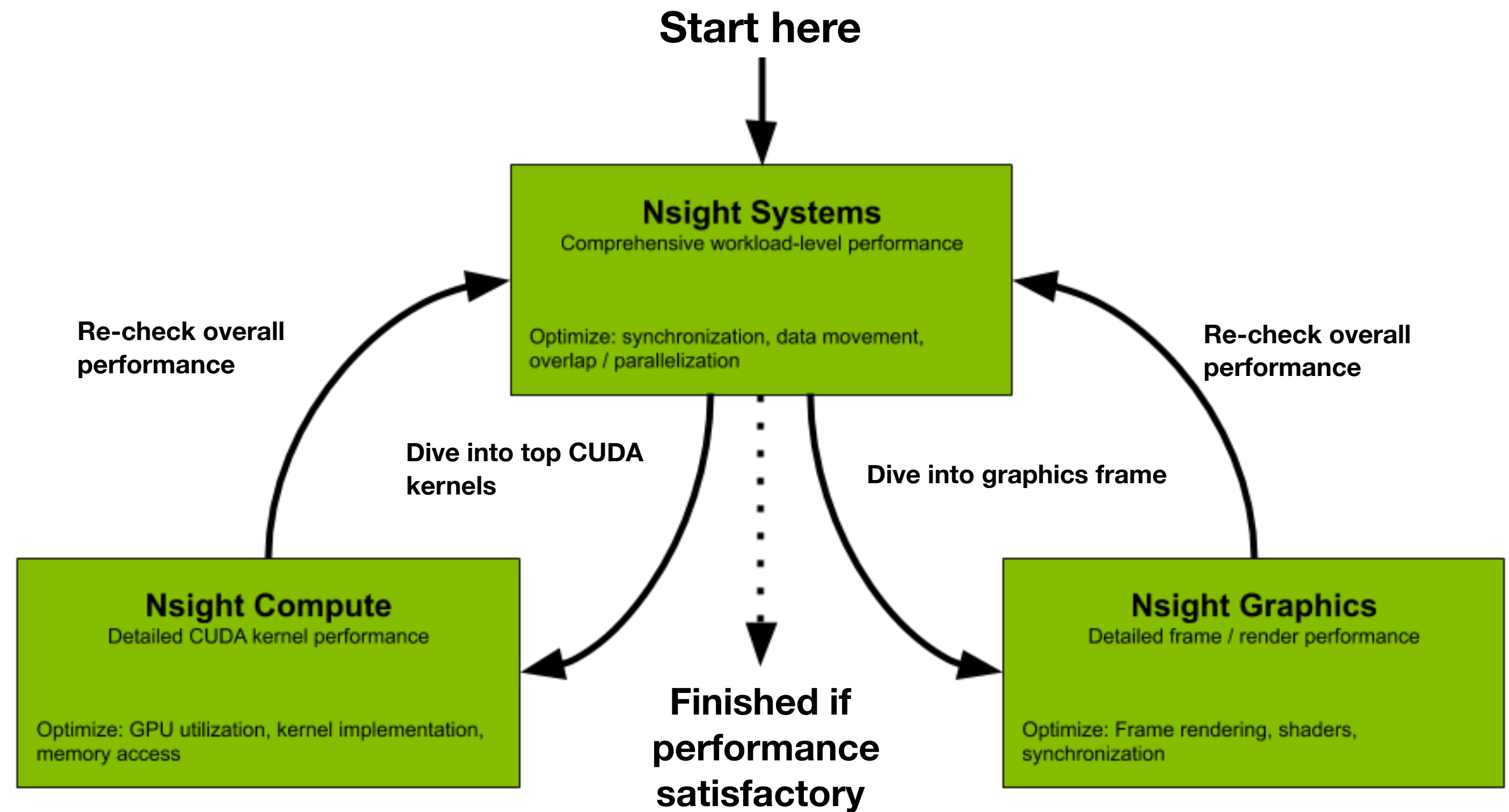
Analyze application algorithm system-wide

Nsight Compute

Debug/ Optimise CUDA kernel

Nsight Graphics

Debug/ Optimise graphics workloads



At First NSIGHT: Recording an application timeline

Notable flags for nays profile

```
nsys profile -t cuda,nvtx,mpi,openacc --stats=true --force-overwrite true -o my_report ./myapp
```

CUDA API Statistics:							
Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	StdDev	Name
80.8	2,389,597,601	1	2,389,597,601.0	2,389,597,601	2,389,597,601	0.0	cudaDeviceSynchronize
18.9	560,567,324	3	186,855,774.7	37,749	560,460,726	323,551,379.2	cudaMallocManaged
0.3	8,408,411	3	2,802,803.7	2,302,933	3,332,155	515,243.9	cudaFree
0.0	79,831	1	79,831.0	79,831	79,831	0.0	cudaLaunchKernel

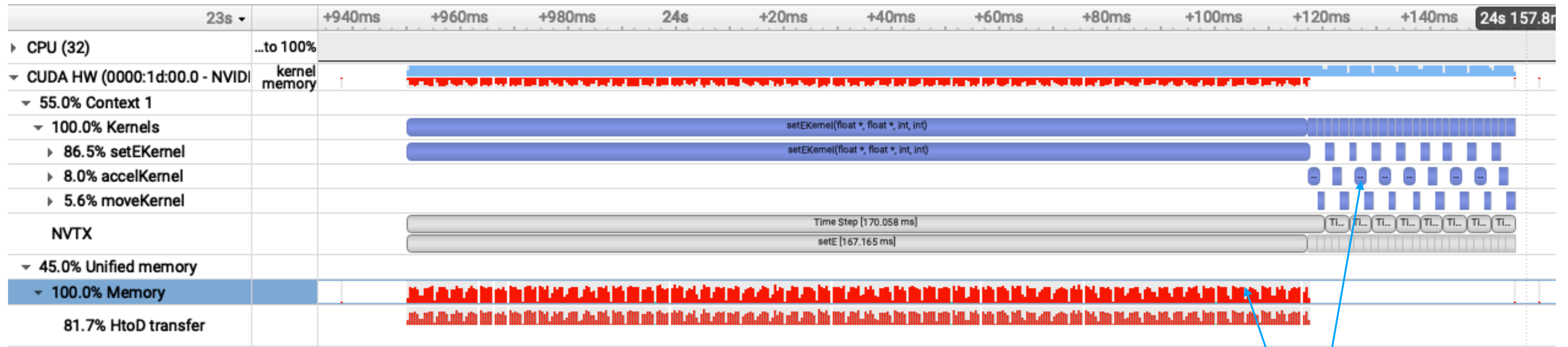
CUDA Memory Operation Statistics (by time):							
Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	StdDev	Operation
75.4	10,016,498	1,920	5,216.9	2,846	17,792	4,223.6	[CUDA Unified Memory memcpy HtoD]
24.6	3,269,861	640	5,109.2	2,815	16,862	4,121.6	[CUDA Unified Memory memcpy DtoH]

nsys --help or nsys [specific command] --help

- profile – start a profiling session
- -t: Selects the APIs to be traced (cuda, cublas, nvtx, mpi openmp and openacc in this example)
- —cuda-memory-usage = true or false
- --stats: if true, it generates summary of statistics after the collection
- --force-overwrite: if true, it overwrites the existing generated report
- -o – name for the intermediate result file, created at the end of the collection (.qdrep filename)

NOTE:
When using:
In -s \$TMPDIR /tmp/nvidia

Nsight system report



Look at this pattern

Performance consideration

1

Optimal Thread Utilization

Maximize occupancy by launching enough threads per multiprocessor

2

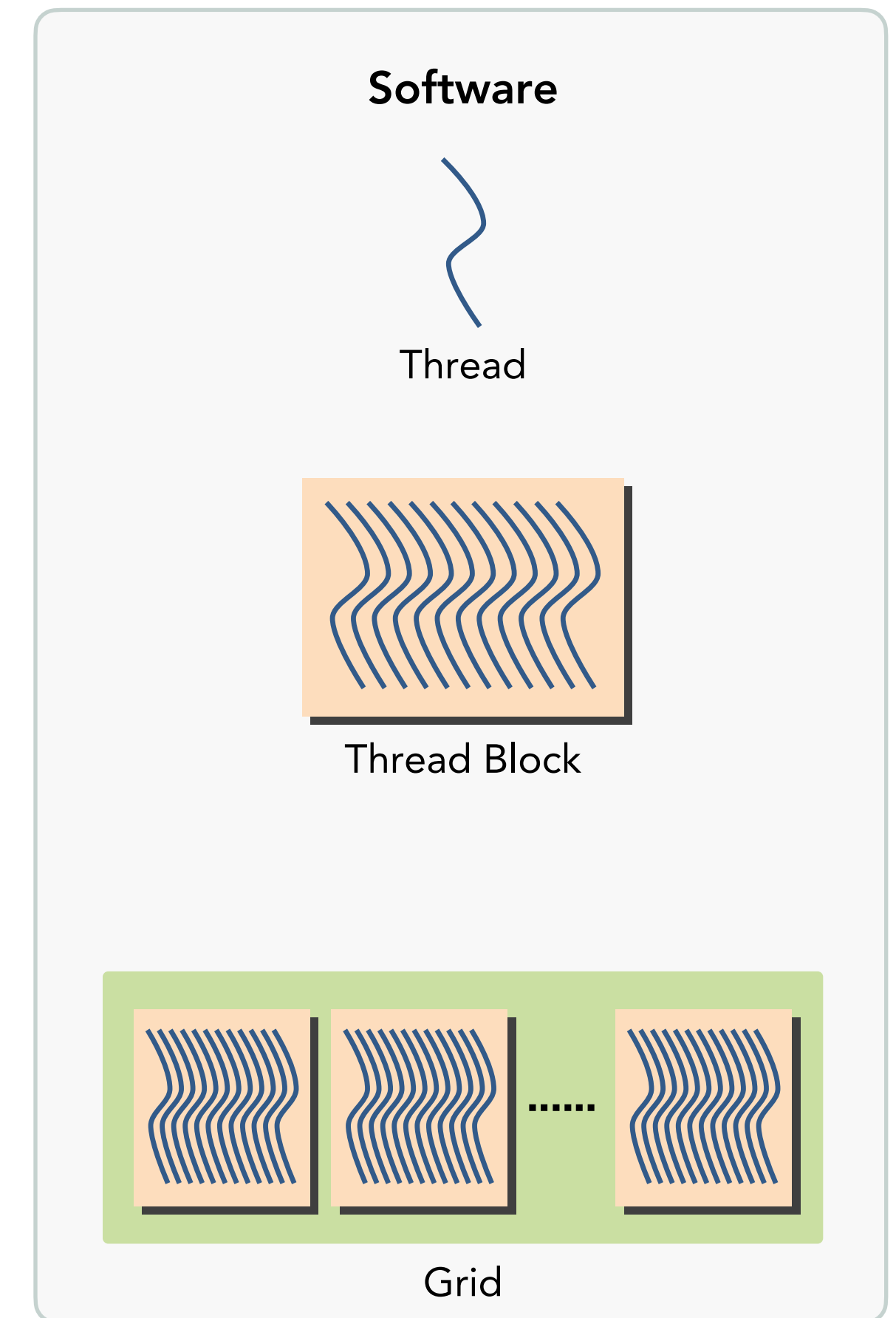
Resource Allocation

Optimize performance by using block sizes that are multiples of 32

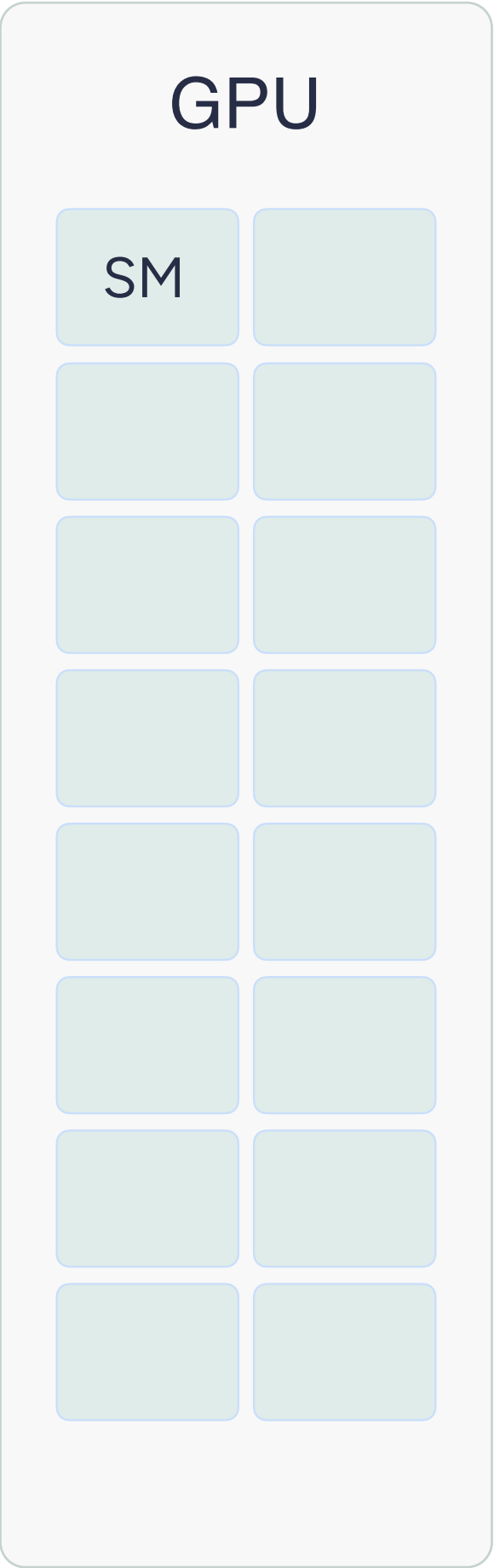
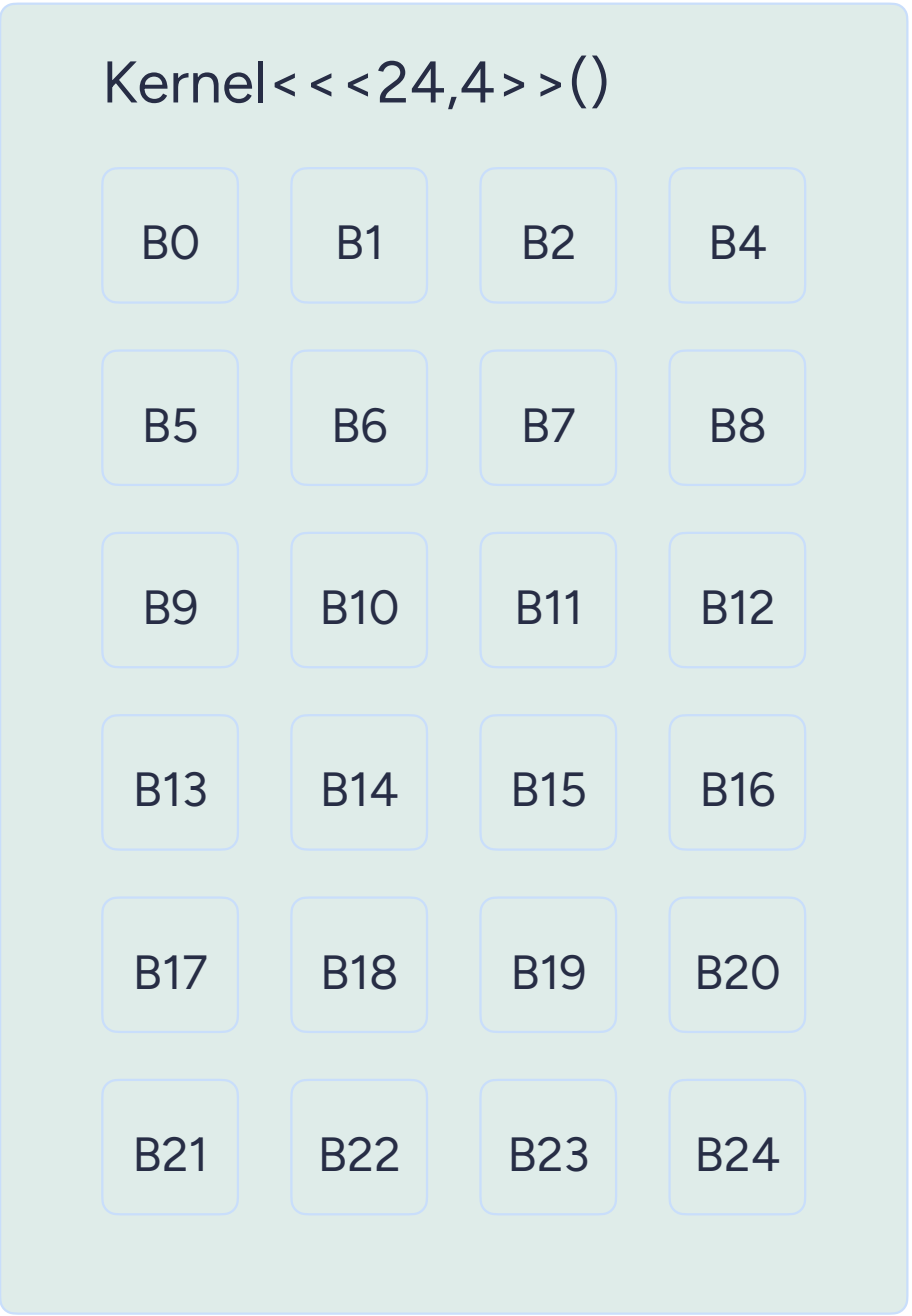
3

Load Balancing

Matching grid size to GPU SMs can boost performance



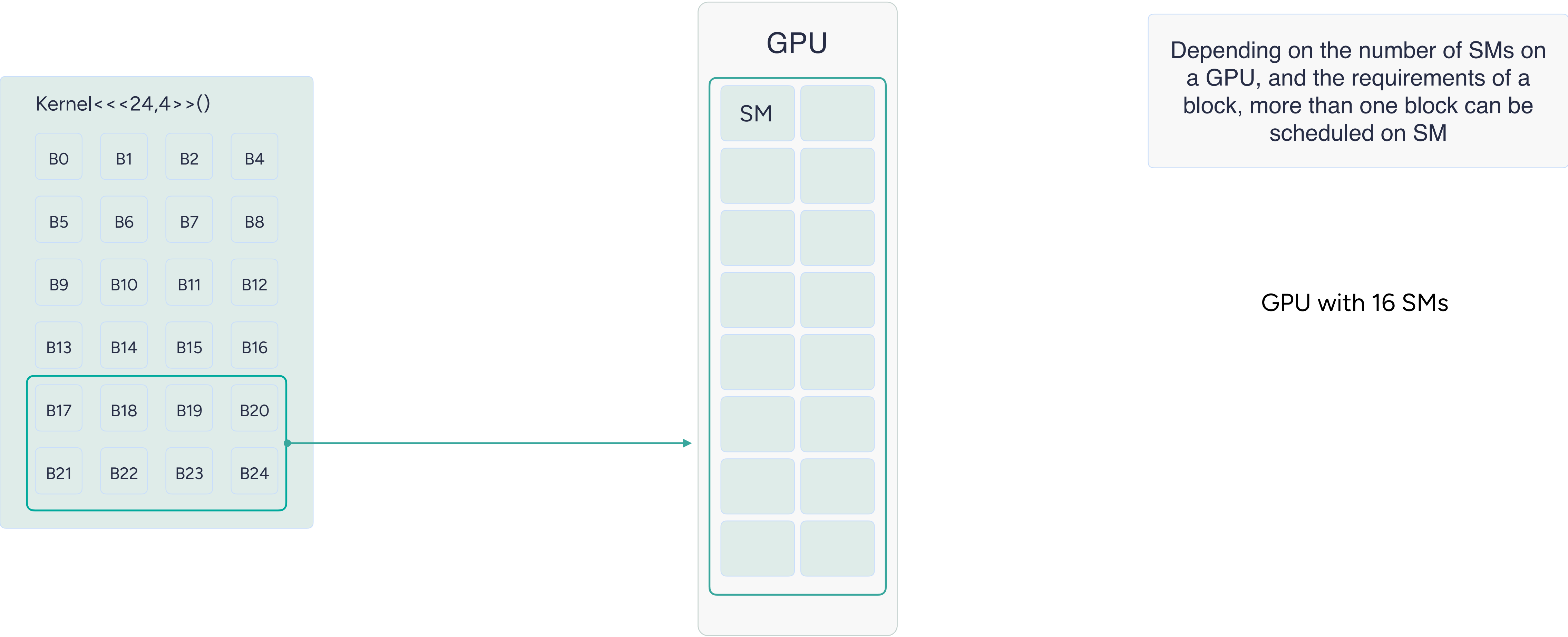
NVIDIA GPUs contain functional units called SMs



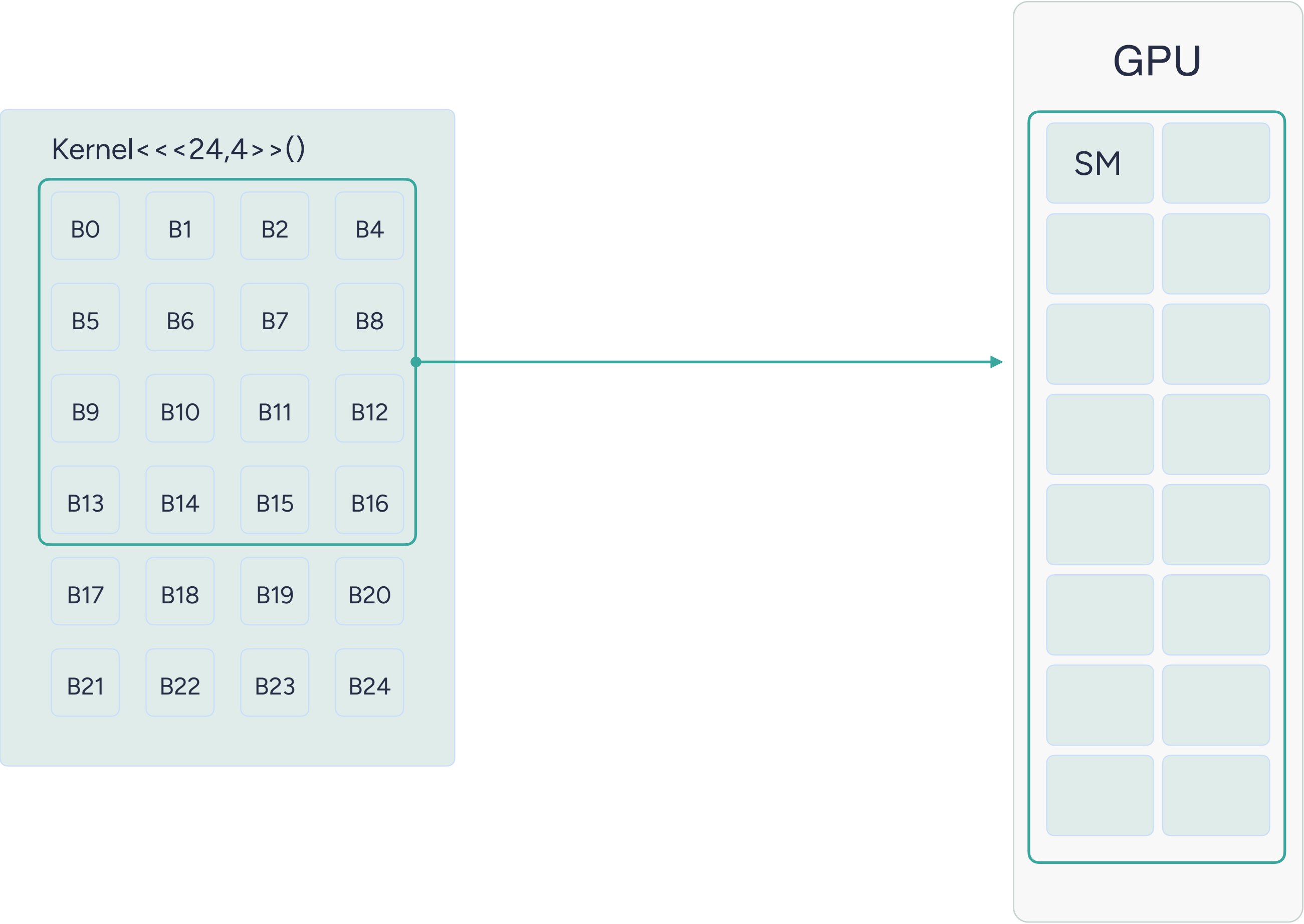
Blocks of threads are
scheduled to run on SMs

GPU with 16 SMs

More than one block can be schedule on an SM



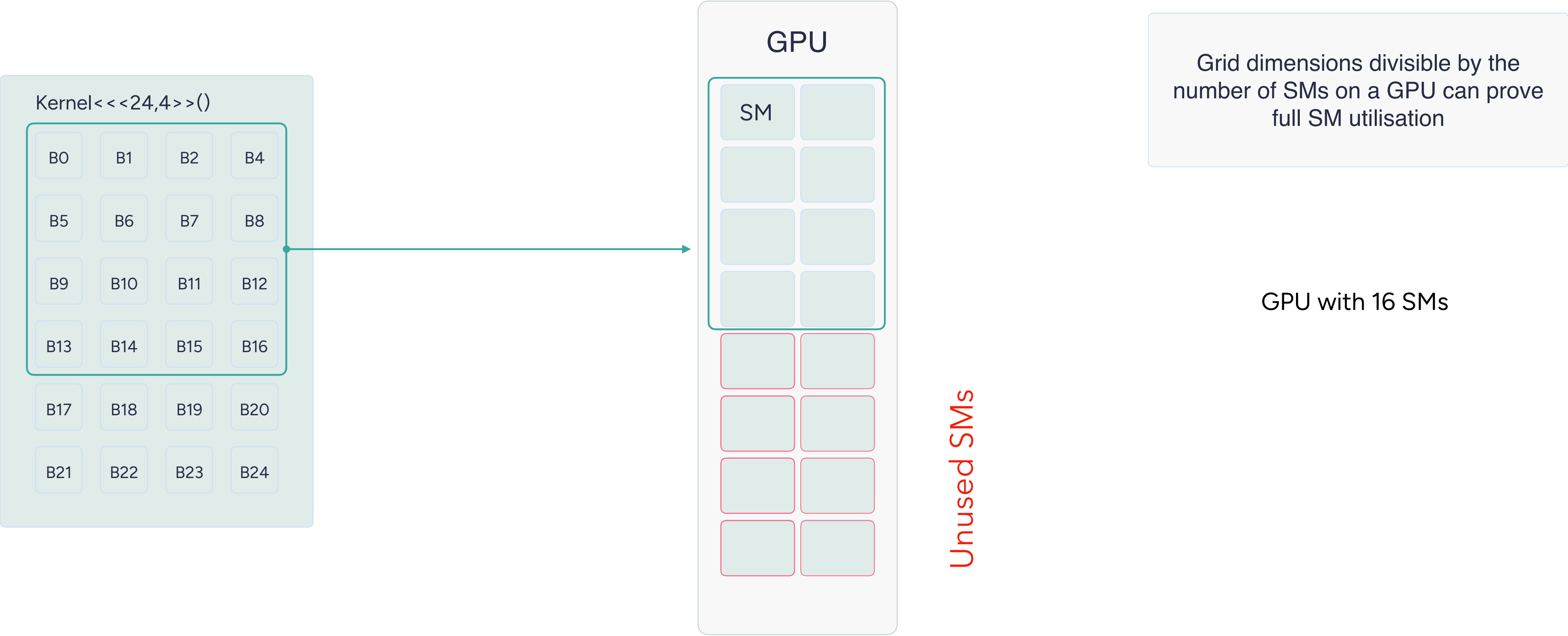
More than one block can be schedule on an SM



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

GPU with 16 SMs

More than one block can be schedule on an SM



Programmatically querying GPU device properties

- **The number of SMs on a GPU** can differ depending on the specific GPU being used, so the number of SMs should **not be hard-coded** into a code bases
- This information should be acquired **programmatically**.
- To obtain the id of the currently active GPU:

```
int deviceId;  
cudaGetDevice(&deviceId);
```

- To obtain a C struct which contains many properties about the currently active GPU device, including its number of SMs:

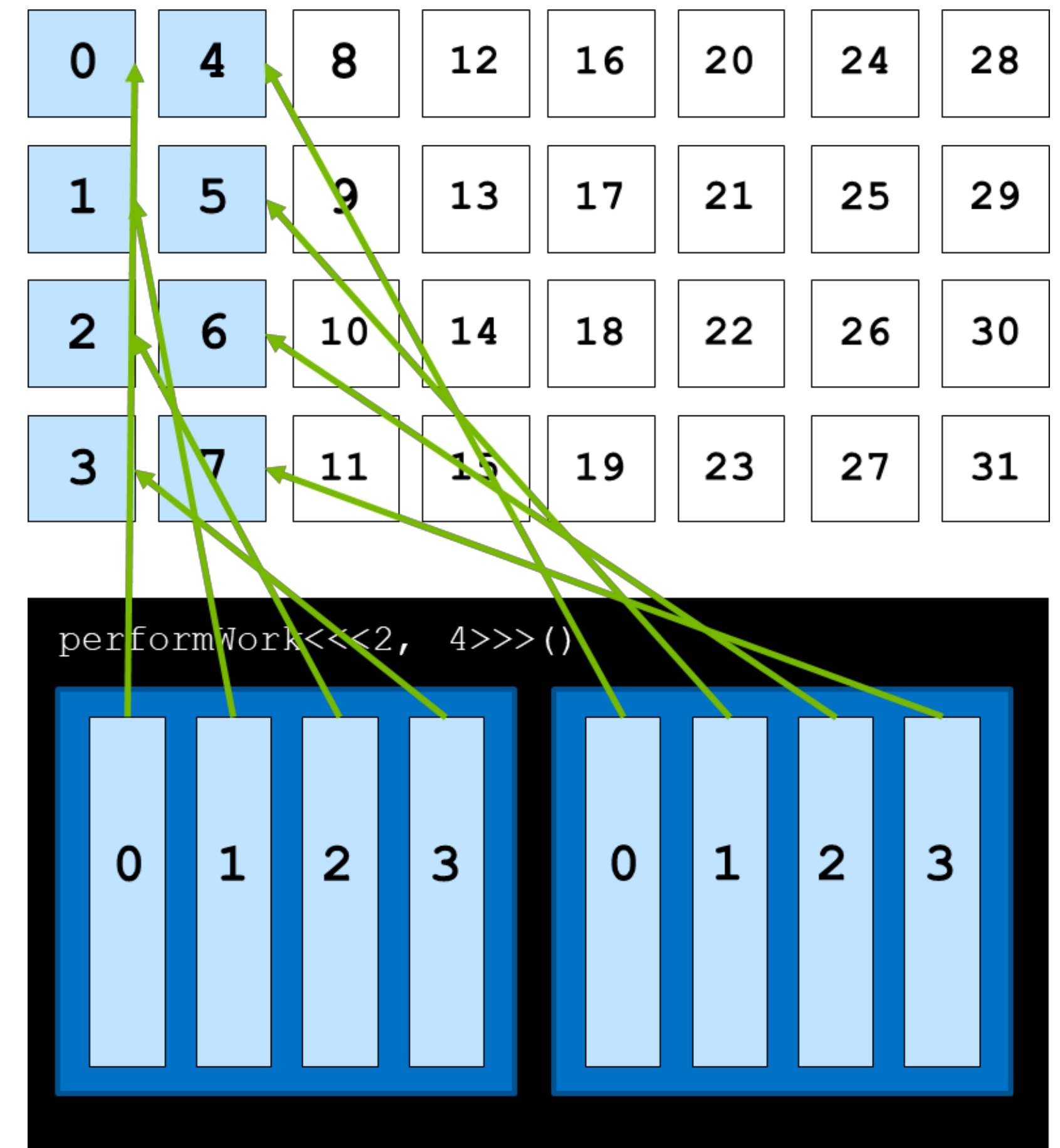
```
cudaDeviceProp props;  
cudaGetDeviceProperties(&props, deviceId);
```



Are there hardware constraints on threads per block and blocks per grid?

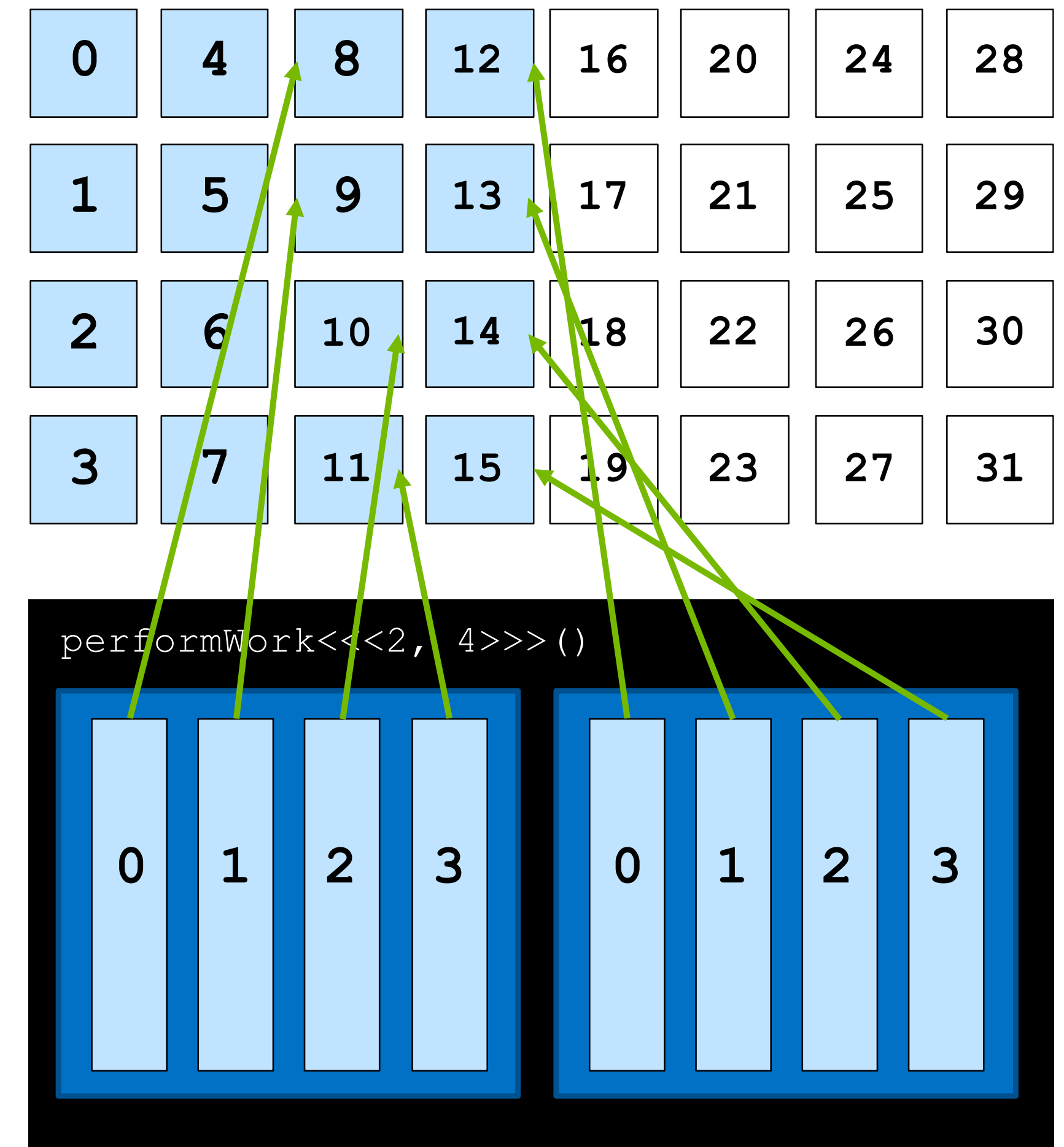
When the data set is larger than grid size?

- In this scenario, each thread should work on more elements.
- Work can be assigned programmatically with a grid- stride loop.



When the data set is larger than grid size?

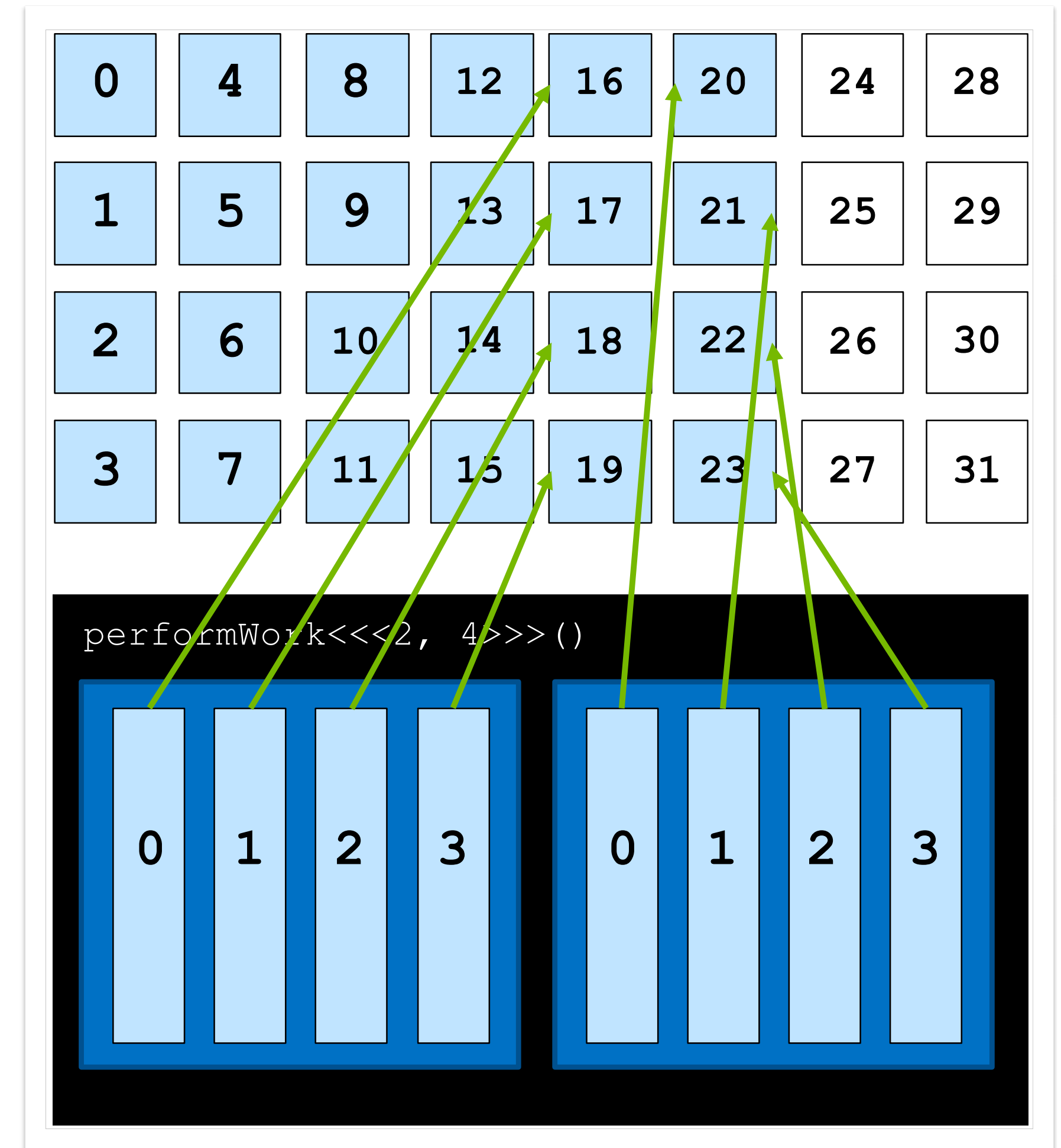
- In this scenario, each thread should work on more elements.
- Work can be assigned programmatically with a grid- stride loop
 - the first element to be assigned to a thread is calculated via the global index



When the data set is larger than grid size?

- In this scenario, each thread should work on more elements.
- Work can be assigned programmatically with a grid- stride loop
 - the first element to be assigned to a thread is calculated via the global index

```
int globalIndex = threadIdx.x + blockIdx.x * blockDim.x;
```



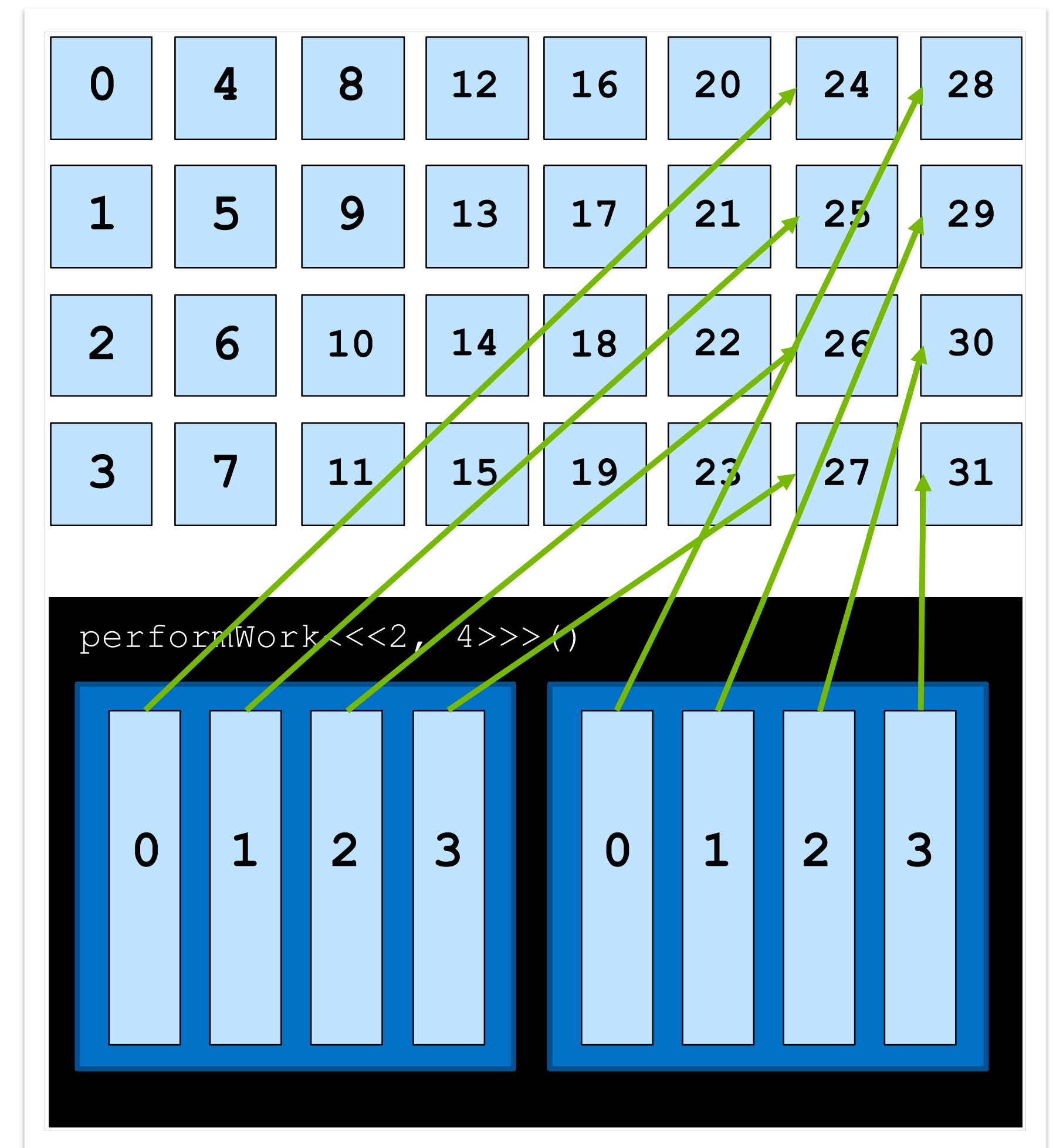
When the data set is larger than grid size?

- In this scenario, each thread should work on more elements.
- Work can be assigned programmatically with a grid- stride loop
 - the first element to be assigned to a thread is calculated via the global index

```
int globalIndex = threadIdx.x + blockIdx.x * blockDim.x;
```

- the next one is obtained by summing the number of threads in the grid

```
stride = blockDim.x * gridDim.x
```



Data set larger than grid size: grid-stride loop

Operation inside the kernel will be executed in a grid-stride loop:

```
__global__ void kernel(int *a, int N) {  
    int indexWithinTheGrid = threadIdx.x + blockIdx.x * blockDim.x;  
    int gridStride = gridDim.x * blockDim.x;  
  
    for (int i = indexWithinTheGrid; i < N; i += gridStride)  
    {  
        // do work on a[i];  
    }  
}
```

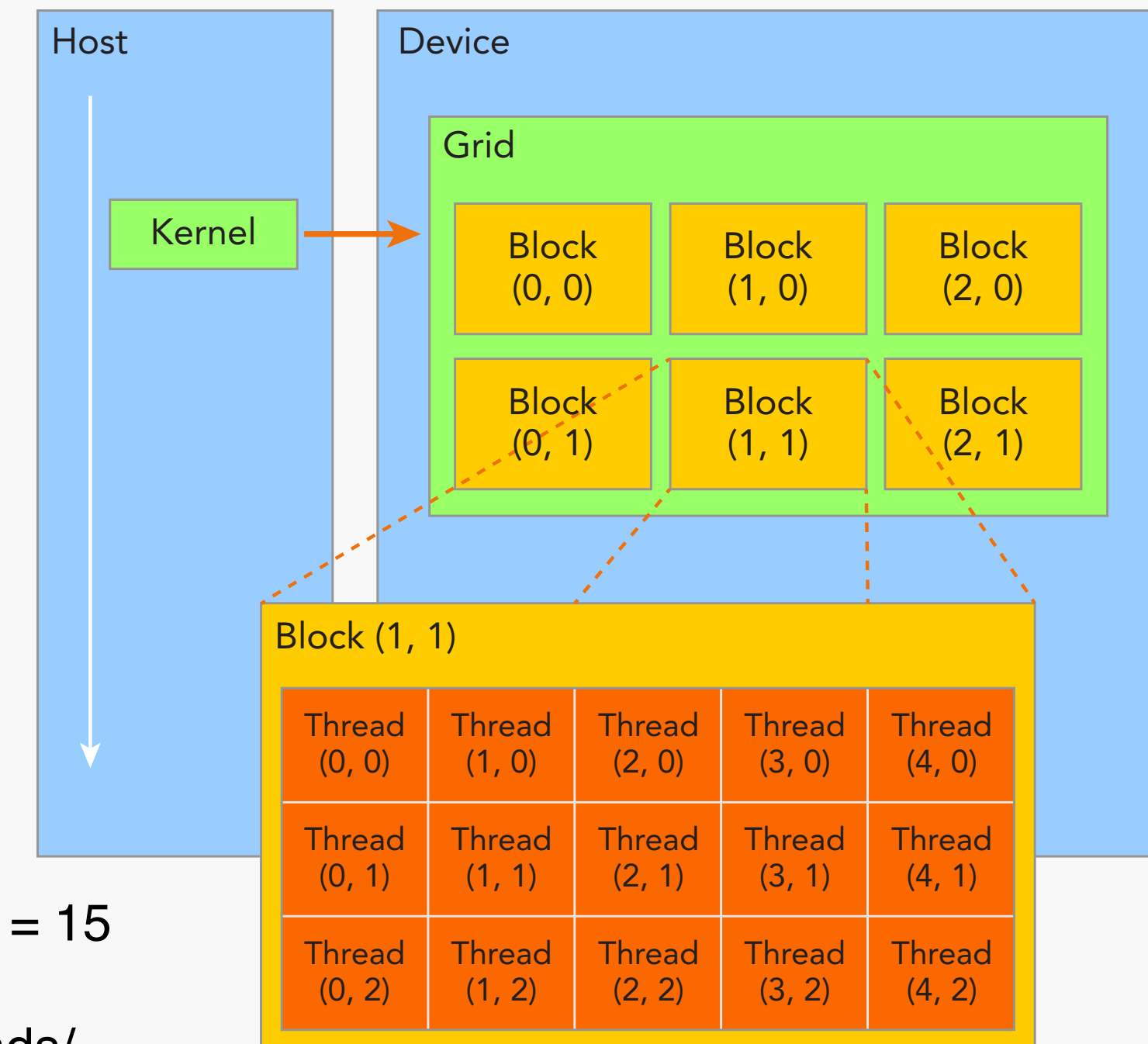
Multidimensional blocks and grids

Multidimensional Blocks and Grids

Host program specifies “grid-block-threads” configuration for kernel at run time

- All threads spawned by a single kernel launch are collectively called a *grid*
- All threads in a grid share the same global memory space
- A grid is made up of many thread blocks
- Kernel needs to know run-time configuration
- Built-in-three-dimensional type for threads (uint3) and blocks (dim3)
 - `threadIdx.x, threadIdx.y, threadIdx.z`
 - `blockIdx.x, blockIdx.y, blockIdx.z`
 - `blockDim.x, blockDim.y, blockDim.z`

Grid Dimension: $3 \times 2 = 6$ Blocks



Block Dimension: $5 \times 3 = 15$ Threads/Blocks
(6 Blocks) \times (15 Threads/Blocks) = 90 Total threads in Grid

Device Run-time Configuration

Type	Variable	Description
dim3	gridDim	Dimensions of grid
uint3	blockIdx	Index of block within grid
dim3	blockDim	Dimensions of block
uint3	ThreadId	Index of thread within block

Dimension	Variable	ID
1D	(Dx)	x
2D	(Dx, Dy)	y + y*Dx
3D	(Dx, Dy, Dz)	z + y*Dx + z*DxDy

CUDA compute grid

CUDA compute grid supports 1-3 dimensions

```
gpu_kernel<<<4,2>>>(...)
```

```
gpu_kernel<<<dim3(8, 4, 1), dim3(4,2,1) >>>(...)
```

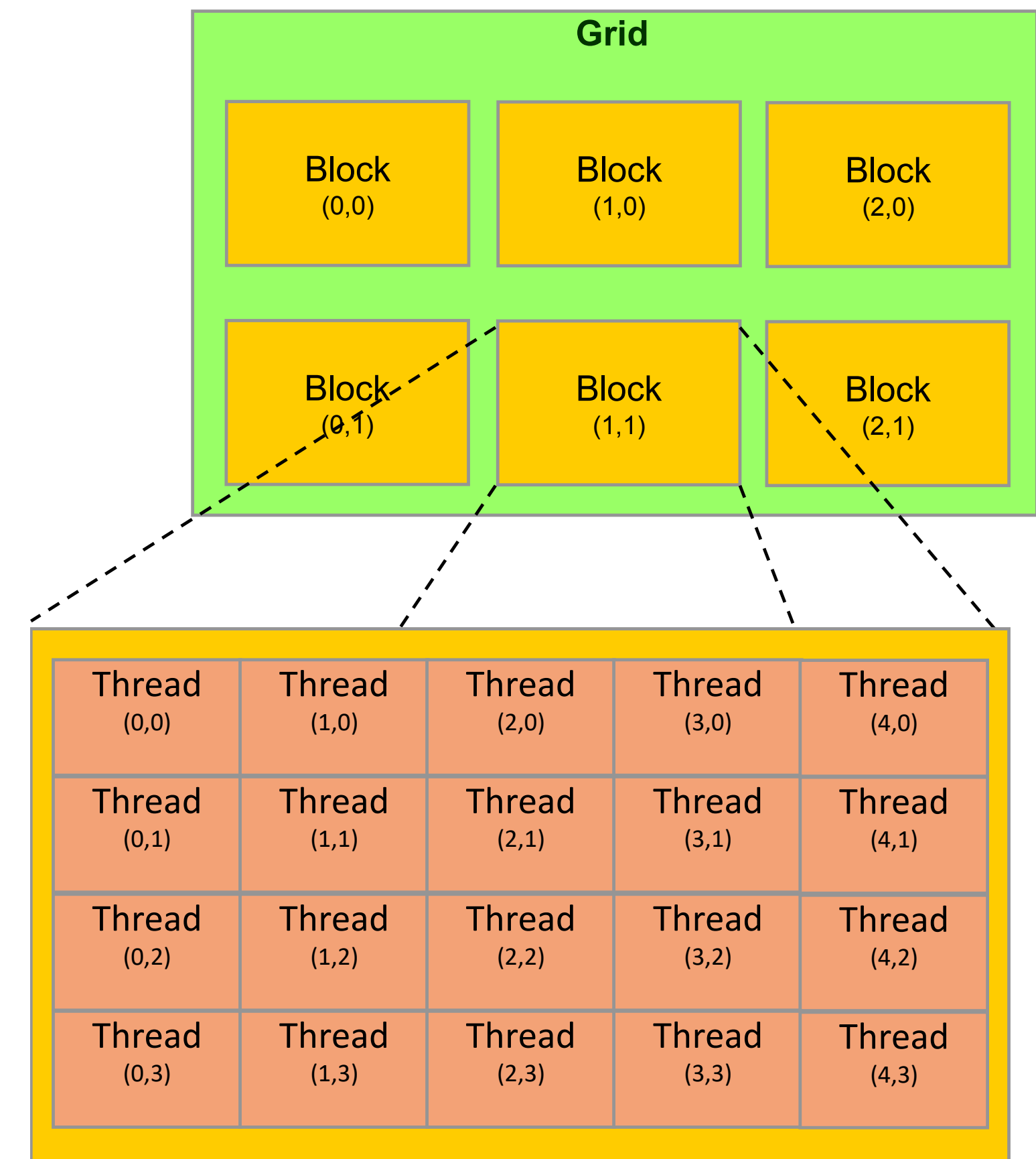
```
gpu_kernel<<<dim3(16, 8, 4), dim3(8, 4, 2) >>>(...)
```

Useful for when

Dealing with multidimensional data

CUDA's dim3 type for both 2D and 3D grids and blocks

CUDA variables: gridDim.x, gridDim.y, gridDim.z, gridBlock.z,...



Device Run-time Configuration

```
27 int main(int argc, char **argv)
28 {
29     const int b_x = 2, b_y = 3, b_z = 4;
30     const int t_x = 3, t_y = 3, t_z = 3;
31
32     int blocks_per_grid = b_x * b_y * b_z;
33     int threads_per_block = t_x * t_y * t_z;
34
35     printf("%d blocks/grid\n", blocks_per_grid);
36     printf("%d threads/block\n", threads_per_block);
37     printf("%d total threads\n", blocks_per_grid * threads_per_block);
38
39     dim3 blocksPerGrid(b_x, b_y, b_z);
40     dim3 threadsPerBlock(t_x, t_y, t_z);
41
42     whoami<<<blocksPerGrid,threadsPerBlock>>>();
43     cudaDeviceSynchronize();
44
45     return 0;
46 }
```

[cuda-whoami.cu](https://github.com/andrewdodds/cuda-whoami)

Output:
24 blocks/grid
27 threads/block
648 total threads

GPU on which this code ran has 384 cores
CUDA can run (a lot) more threads than cores!

Device Run-time Configuration

```
1  #include <stdio.h>
2
3  __global__ void whoami() {
4      int block_id =
5          blockIdx.x +
6          blockIdx.y * gridDim.x +
7          blockIdx.z * gridDim.x * gridDim.y;
8
9      int block_offset =
10         block_id *
11         blockDim.x * blockDim.y * blockDim.z;
12
13     int thread_offset =
14         threadIdx.x +
15         threadIdx.y * blockDim.x +
16         threadIdx.z * blockDim.x * blockDim.y;
17
18     int id = block_offset + thread_offset;
19
20     printf("%04d | Block(%d %d %d) = %3d | thread(%d %d %d) = %3d\n",
21         id,
22         blockIdx.x, blockIdx.y, blockIdx.z, block_offset,
23         threadIdx.x, threadIdx.y, threadIdx.z, thread_offset);
24 }
```

cuda-whoami.cu

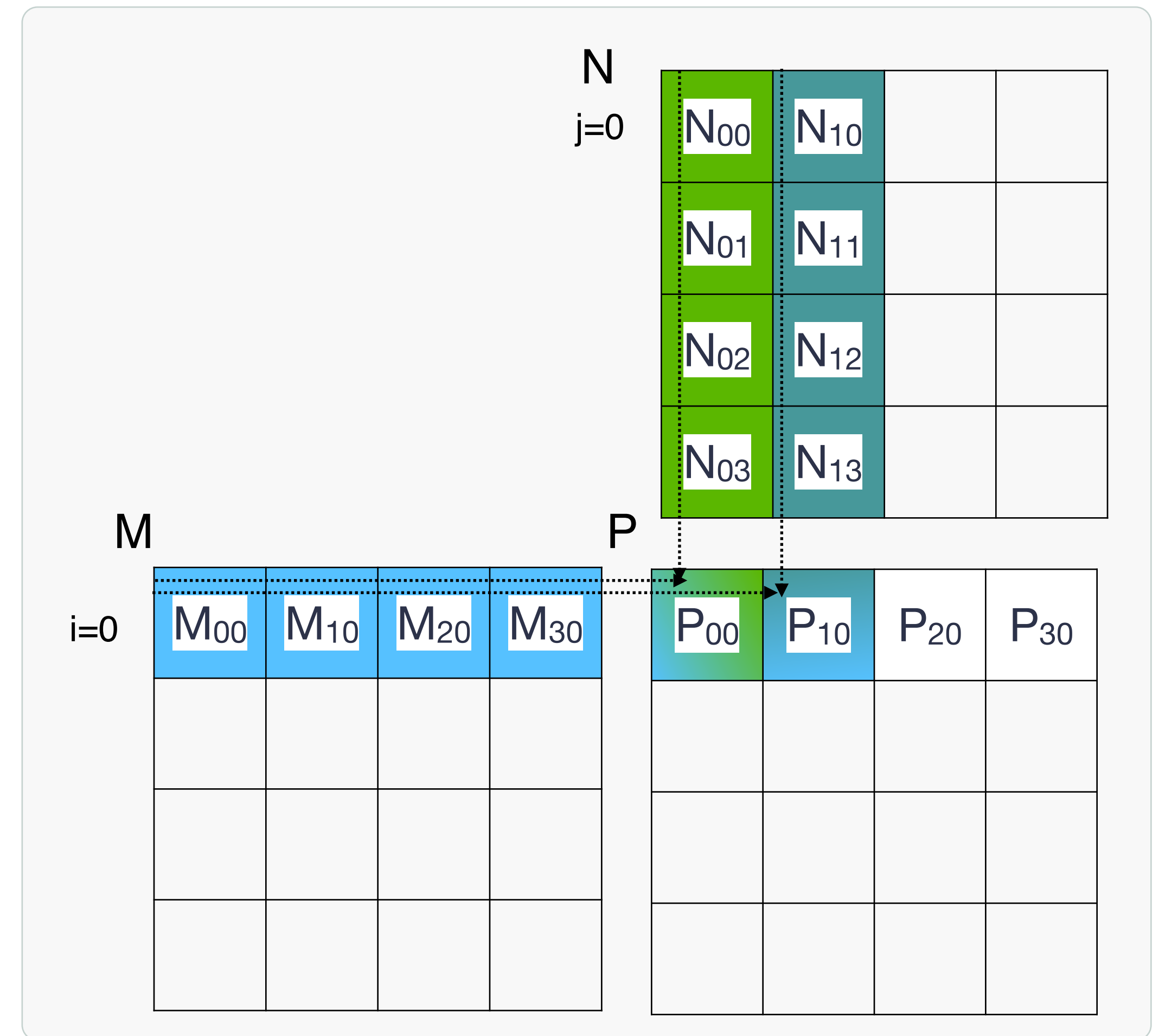
```
0621 | Block(1 2 3) = 621 | thread(0 0 0) = 0
0622 | Block(1 2 3) = 621 | thread(1 0 0) = 1
0623 | Block(1 2 3) = 621 | thread(2 0 0) = 2
0624 | Block(1 2 3) = 621 | thread(0 1 0) = 3
0625 | Block(1 2 3) = 621 | thread(1 1 0) = 4
...
...
0644 | Block(1 2 3) = 621 | thread(2 1 2) = 23
0645 | Block(1 2 3) = 621 | thread(0 2 2) = 24
0646 | Block(1 2 3) = 621 | thread(1 2 2) = 25
0647 | Block(1 2 3) = 621 | thread(2 2 2) = 26
```

Two matrix multiplication

$$P_{ij} = \sum_{k=1}^n M_{ik} \cdot N_{kj}$$

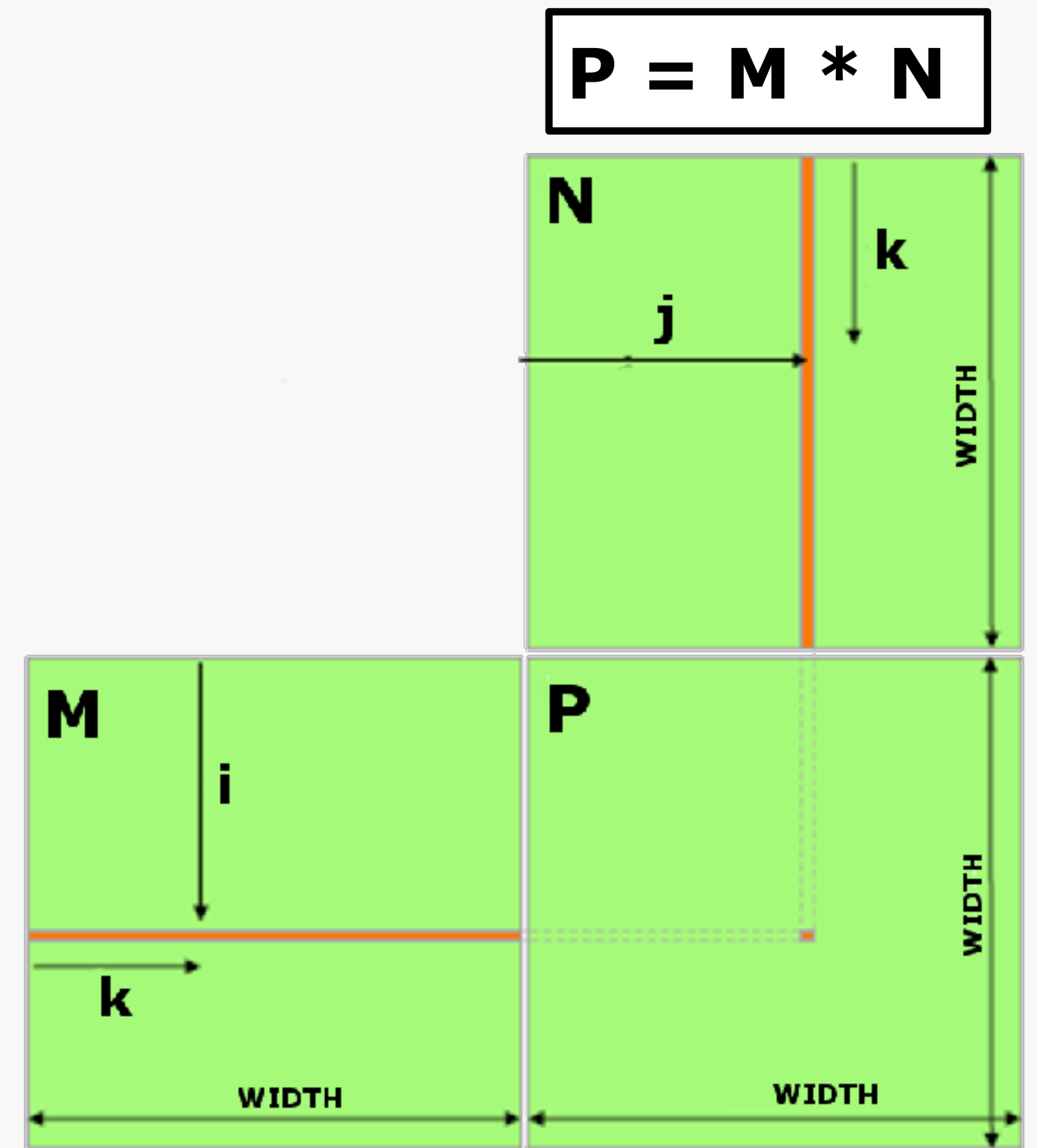
$$P_{10} = M_{00} * N_{10} + M_{10} * N_{11} + M_{20} * N_{12} + + M_{30} * N_{13}$$

$$P_{00} = M_{00} * N_{00} + M_{10} * N_{10} + M_{20} * N_{20} + + M_{30} * N_{30}$$



Two matrix multiplication

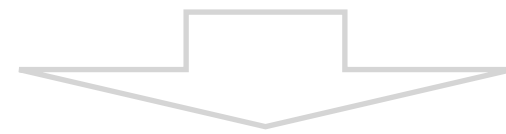
```
void matrixMultOnHost(float* M, float* N, float* P, int Width){  
    for (int row = 0; row < Width; ++row){  
        for (int col = 0; col < Width; ++col){  
  
            // accumulate element-wise products  
            float pval = 0;  
            for (int k = 0; k < Width; ++k){  
                float a = M[row*Width + k];  
                float b = N[k*Width + col];  
                pval += a*b;  
            }  
            P[row*width + col] = pval;  
        }  
    }  
}
```



CUDA compute grid supports 1-3 dimensions

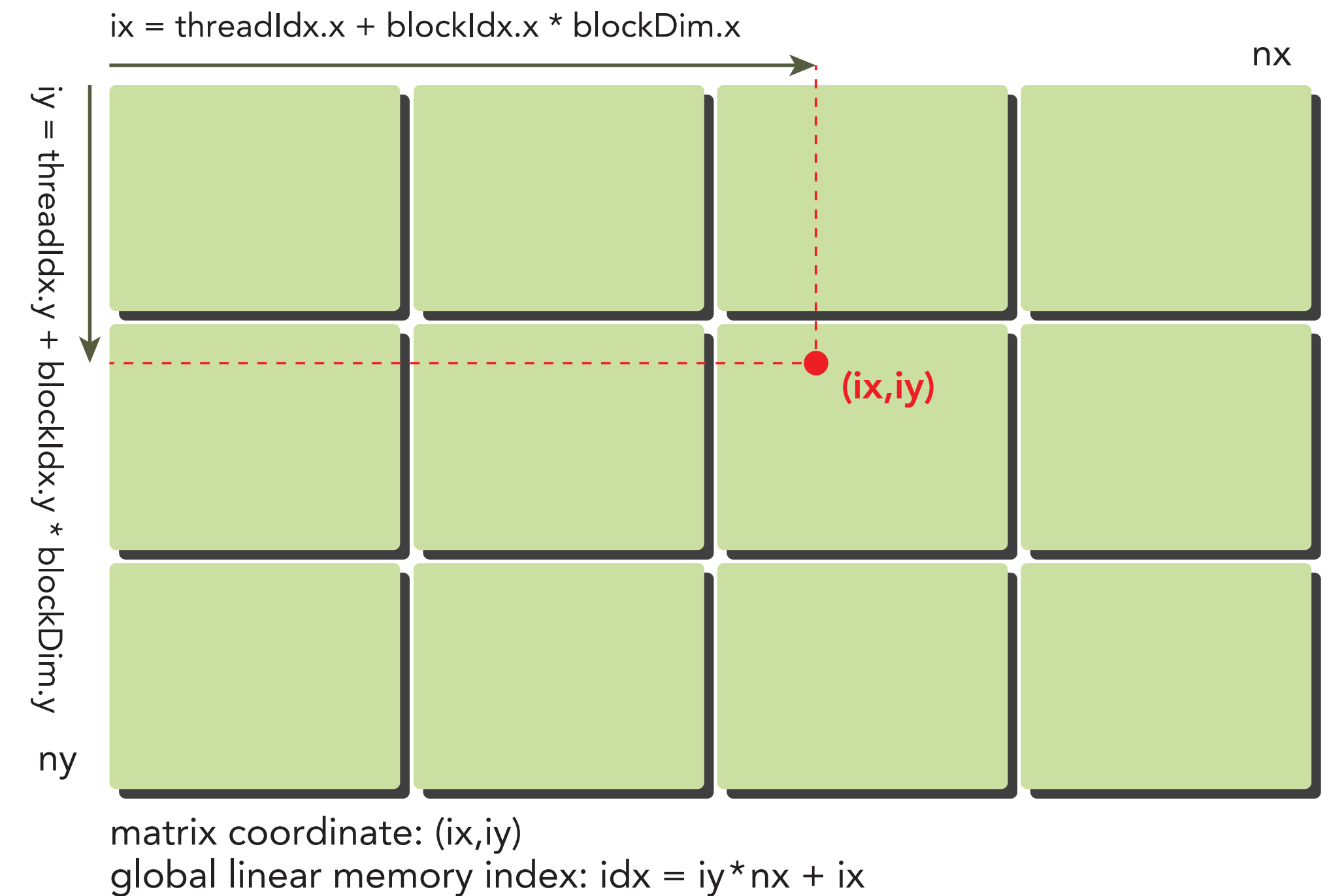
2D

```
int i = blockIdx.x * blockDim.x + threadIdx.x;  
int j = blockIdx.y * blockDim.y + threadIdx.y;
```



3D

```
int i = blockIdx.x * blockDim.x + threadIdx.x;  
int j = blockIdx.y * blockDim.y + threadIdx.y;  
int k = blockIdx.z * blockDim.z + threadIdx.z;
```



- CUDA “hides” loop headers into kernel launch parameters
- Ranges are distributed between threads and blocks of threads
- Blocks number is rounded up to handle the remainder

Two matrix multiplication on GPU

N	Methods	Time execution	Speedup
2048x2048	Serial	25,18	1
	CUDA	0,063	398,29

 Checkpoint-3 : 2D_matrix_multiplication.cu

Things to do

The starting point of this exercise contains a working host function, called `matrixMulCPU`. Your task is to build out the `matrixMulGPU` CUDA kernel. The source code will execute the matrix multiplication with both functions, and compare their answers to verify the correctness of your CUDA kernel.

Follow these guidelines.

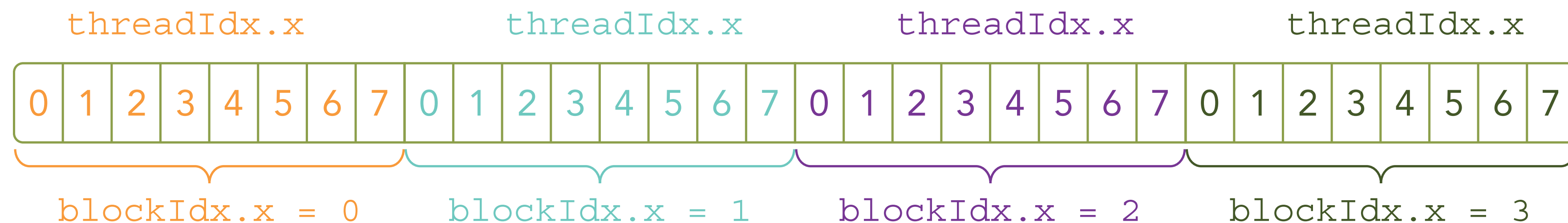
- T1. Create an execution configuration whose arguments are both `dim3` values with the x and y dimensions set to greater than 1.
- T2. Inside the body of the kernel, establish the running thread's unique index within the grid as usual, but you should define two indices for the thread: one for the x axis of the grid, and one for the y axis of the grid.
- T3. Use managed memory

Unrolling loops

Kernel execution across Thread, Block, and Grid

Choose the optimal block size

- A **limited number of threads** (1024) can fit inside a thread block
- To increase parallelism, we need to **coordinate** work **among thread blocks**.
- This is achieved by **mapping** element of data vector to threads using **global index** = **threadIdx.x** + **blockIdx.x*blockDim.x**



```
for blockIdx.x = 0  
  i = 0 * 8 + threadIdx.x = { 0, 1, 2, ... , 7 }
```

```
for blockIdx.x = 3  
  i = 0 * 8 + threadIdx.x = { 0, 1, 2, ... , 7 }
```

Unrolling loops

```
__global__ void unrolledMatrixMultiplicationKernel(float *A, float *B, float *C, int n, int m, int p) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // Row index of C
    int j = blockIdx.y * blockDim.y + threadIdx.y; // Column index of C

    if (i < n && j < p) {
        float sum = 0; // Changed to float
        for (int k = 0; k < m - 3; k += 4) {
            sum += A[i * m + k] * B[k * p + j] + A[i * m + k + 1] * B[(k + 1) * p + j] +
                A[i * m + k + 2] * B[(k + 2) * p + j] + A[i * m + k + 3] * B[(k + 3) * p + j];
        }
        // Handle remaining elements
        for (int k = (m / 4) * 4; k < m; k++) {
            sum += A[i * m + k] * B[k * p + j];
        }
        C[i * p + j] = sum;
    }
}
```

Two matrix multiplication on GPU

N	Methods	Time execution	Speedup
2048x2048	Serial	25,18	1
	CUDA	0,063	398,29
	Unrolled loop	0,055491	453,92

What Bandwidth can a kernel achieve?

Matrix transpose problem

0	1	2	3
4	5	6	7
8	9	10	11

```
void transposeHost(float *out, float *in, const int nx, const int ny) {  
    for (int iy = 0; iy < ny; ++iy) {  
        for (int ix = 0; ix < nx; ++ix) {  
            out[ix*ny+iy] = in[iy*nx+ix];  
        }  
    }  
}
```

0	4	8
1	5	9
2	6	10
3	7	11

transposed

data layout of original matrix

0	1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	---	----	----

data layout of transposed matrix

0	4	8	1	5	9	2	6	10	3	7	11
---	---	---	---	---	---	---	---	----	---	---	----

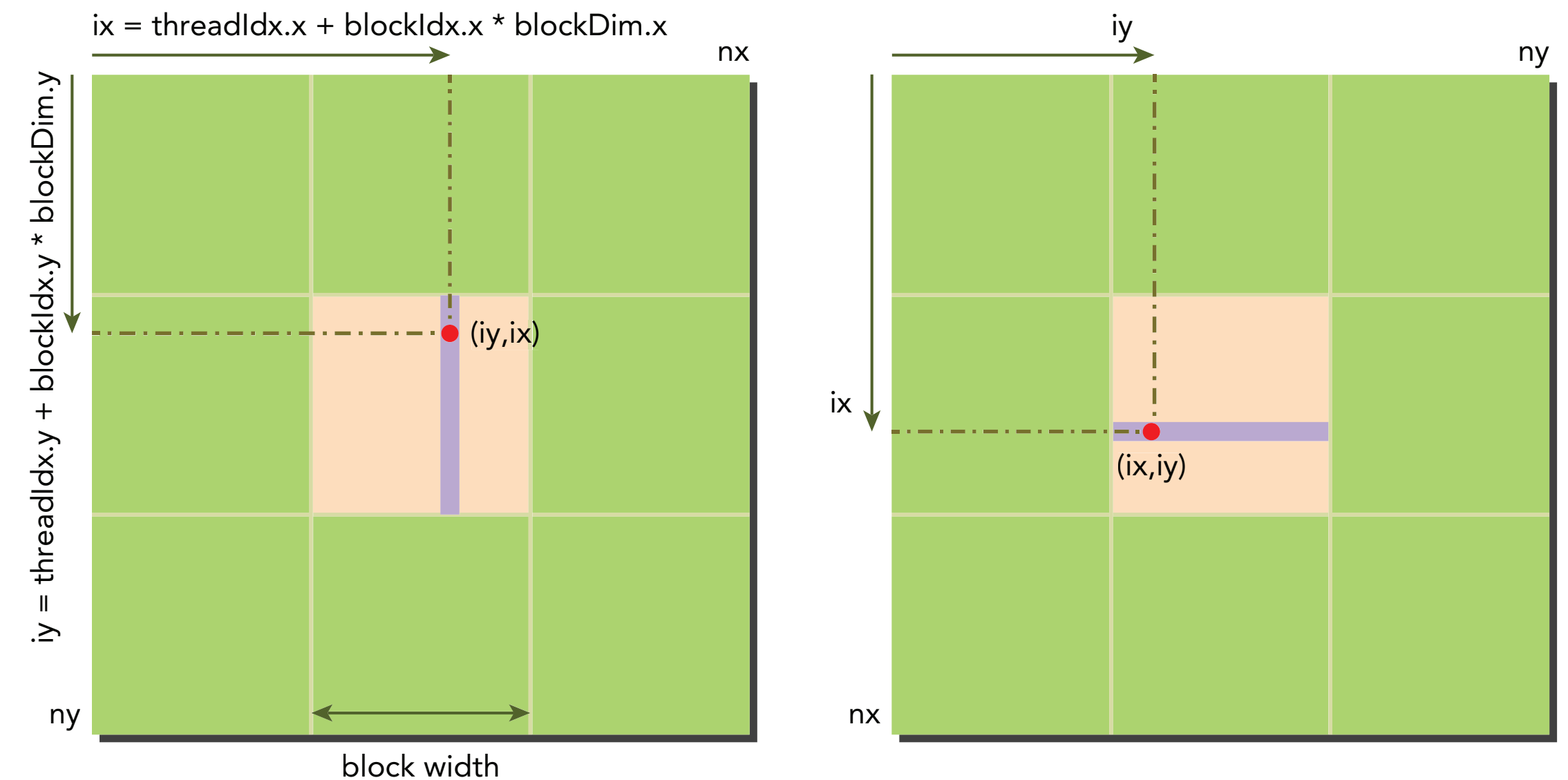
CUDA Matrix transpose

__global__

```
void tranposeRow(float *out, float *in, const int nx, const int ny) {  
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;  
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;  
  
    if (ix < nx && iy < ny) { out[iy*nx + ix] = in[iy*nx + ix];}  
}
```

__global__

```
void tranposeCol(float *out, float *in, const int nx, const int ny) {  
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;  
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;  
  
    if (ix < nx && iy < ny) { out[ix*ny + iy] = in[ix*ny + iy]; }  
}
```



Effective Bandwidth of Kernels

BLOCKSIZE	KERNEL	BANDWIDTH [GB/s]	RATIO TO PEAK BANDWITDH (%)
	Theoretical peak bandwidth	900,0	
16 X16	copyRow: Load/store using rows	626,60	69,62
	copyCol: Load/store using cols	275,42	30,60
32X32	copyRow: Load/store using rows	376,32	41,81
	copyCol: Load/store using cols	170,14	18,90

Naive Transpose: Reading Rows versus Reading Columns

__global__

```
void tranposeNRow(float *out, float *in, const int nx, const int ny) {  
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;  
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;  
  
    if (ix < nx && iy < ny) { out[ix * ny + iy] = in[iy * nx + ix]; }  
}
```

__global__

```
void tranposeNCol(float *out, float *in, const int nx, const int ny) {  
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;  
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;  
  
    if (ix < nx && iy < ny) { out[iy*nx + ix] = in[ix*ny + iy]; }  
}
```

BLOCKSIZE	KERNEL	BANDWIDTH [GB/s]	RATIO TO PEAK BANDWITDH (%)
	Theoretical peak bandwidth	900,0	
16 X16	copyRow: Load/store using rows	273,09	30,34
	copyCol: Load/store using rows	296,09	32,90

Unrolling Transpose: Reading Rows versus Reading Columns

```
__global__ void transposeUnroll4Row(float *out, float *in, const int nx,
const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x*4 + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int ti = iy*nx + ix; unsigned int to = ix*ny + iy;

    // access in columns
    if (ix+3*blockDim.x < nx && iy < ny) {
        out[to] = in[ti];
        out[to + ny*blockDim.x] = in[ti+blockDim.x];
        out[to + ny*2*blockDim.x] = in[ti+2*blockDim.x];
        out[to + ny*3*blockDim.x] = in[ti+3*blockDim.x];
    }
}
```

```
__global__ void transposeUnroll4Col(float *out, float *in, const int nx,
const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x*4 + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int ti = iy*nx + ix; unsigned int to = ix*ny + iy;

    // access in columns
    if (ix+3*blockDim.x < nx && iy < ny) {
        out[ti] = in[to];
        out[ti + blockDim.x] = in[to+ blockDim.x*ny];
        out[ti + 2*blockDim.x] = in[to+ 2*blockDim.x*ny];
        out[ti + 3*blockDim.x] = in[to+ 3*blockDim.x*ny];
    }
}
```

Effective Bandwidth of Kernels

BLOCKSIZE	KERNEL	BANDWIDTH [GB/s]	RATIO TO PEAK BANDWITDH (%)
	Theoretical peak bandwidth	900,0	
16 X16	NaiveRow: Load/store using rows	317,29	35,25
	NaiveCol: Load/store using rows	742,74	82,53
32X32	NaiveRow: Load/store using rows	160,73	17,86
	NaiveCol: Load/store using rows	492,21	54,69

 Final Project

Things to do

In this exercise, you will accelerate an application that simulates the thermal conduction of silver in 2 dimensional space.

T1. Convert the `step_kernel_mod` function to execute on the GPU.

T2. Modify the main function to properly allocate data for use on CPU and GPU

The `step_kernel_ref` function executes on the CPU and is used for error checking. Because this code involves floating point calculations, different processors, or even simply reordering operations on the same processor, can result in slightly different results. For this reason, the **error checking** code uses an error threshold, instead of looking for an exact match.

Take away message

CUDA gives each thread a unique ThreadID to distinguish between each other even though the kernel instructions are the same

- Grids map to GPUs
- Threads map to Stream Processors (SP)
- Warps are groups of (32) threads that execute simultaneously
- Blocks map to the Streaming MultiProcessors (SMP)

On NVIDIA GPU typically you get

- Maximum number of threads per block: 1024
- Maximum sizes of x-, y-, and z- dimensions of thread block: 1024 x 1024 x 64
- Maximum size of each dimension of grid of thread blocks: 65535 x 65535 x 65535
(about 280,000 billion blocks)

Recommended Resources

