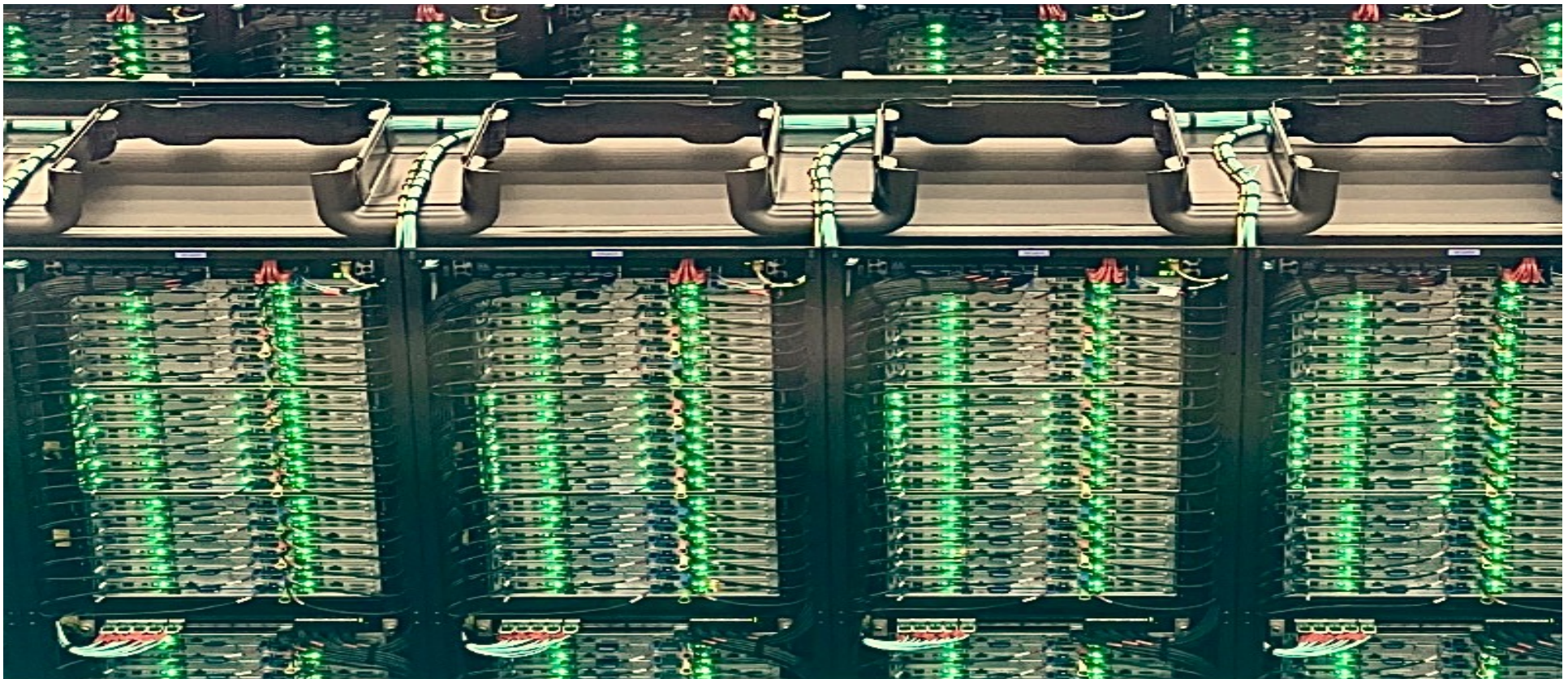


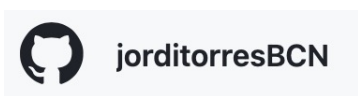
05. GPU Programming and CUDA

Supercomputing for Artificial Intelligence
Foundations, Architectures, and Scaling Deep Learning Workloads

Jordi **TORRES.AI**

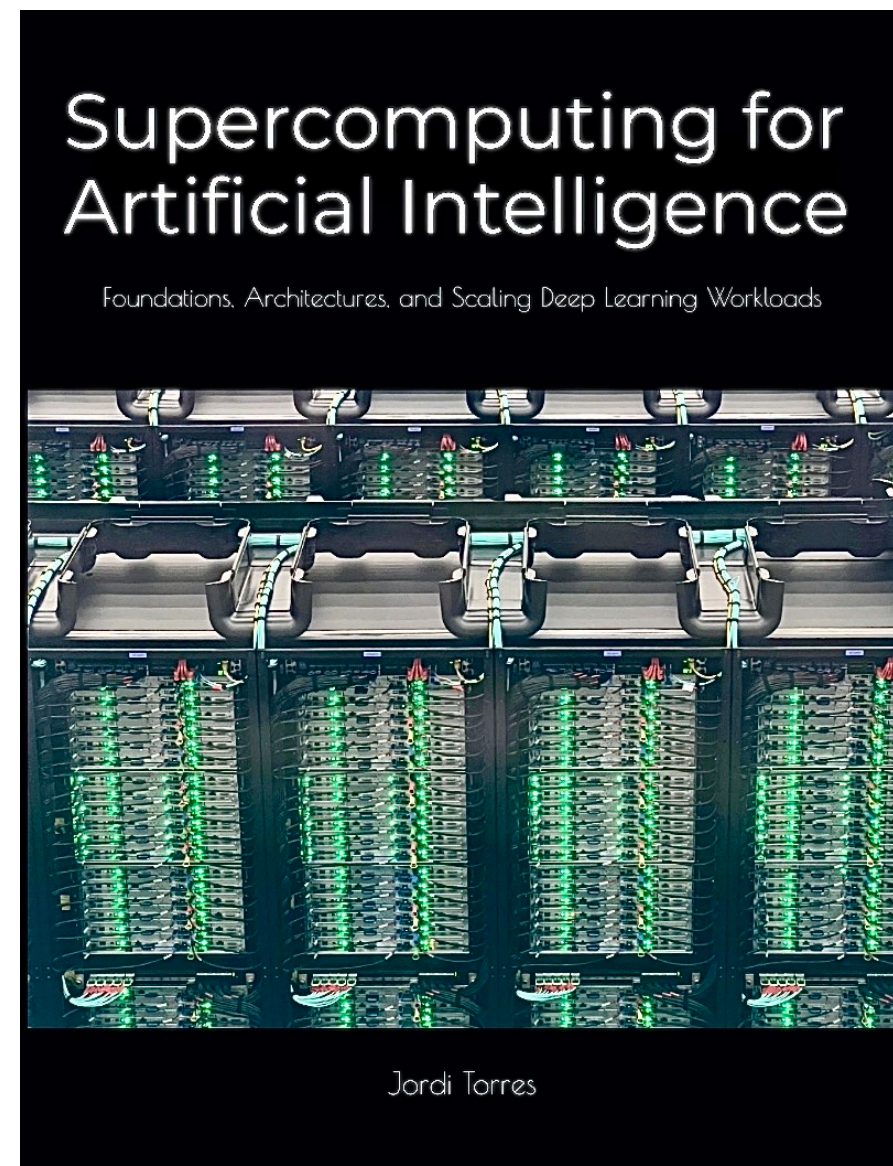


5 GPU PROGRAMMING AND CUDA	215
5.1 Foundations of GPU Acceleration and CUDA Programming	215
Accelerators in Supercomputing	215
CUDA: A Programming Platform for GPUs	220
Core CUDA Libraries for Accelerated Computing	221
5.2 Getting Started with CUDA: Threads, Kernels, and Memory	223
Hello World in CUDA	223
Threads Organization	226
Memory Management	231
Launching a CUDA Kernel	236
Looking Ahead: Topics Beyond the Basics	241
5.3 Case study: Matrix Multiplication in CUDA	243
Handling Errors	244
Managing Flattened Matrices	244
Allocating a GPU with SLURM	248
Timing the kernel	249
Performance and Scalability Analysis Using nvprof	252
5.4 Key Takeaways from Chapter 5	255



HPC4AIbook / Chapter.05 /

Jordi Torres and Jordi Torres Last update	
Name	Last commit message
..	
add.cu	Last update
add_vectors.cu	Last update
checkIndex.cu	Last update
hello.cu	Last update
helloGPU.cu	Last update
matrixMulN.cu	Last update
matrixMulN.slurm	Last update
matrixMulN_all.cu	Last update
matrixMulN_all.slurm	Last update



Content

5.1 Foundations of GPU Acceleration and CUDA


- Programming
- Accelerators in Supercomputing
- CUDA: A Programming Platform for GPUs
- Core CUDA Libraries for Accelerated Computing

5.2 Getting Started with CUDA

- Hello World in CUDA
- Threads Organization
- Memory Management
- Launching a CUDA Kernel
- Looking Ahead: Topics Beyond the Basics

5.3 Case study: Matrix Multiplication in CUDA

- Handling Errors
- Managing Flattened Matrices
- Allocating a GPU with SLURM
- Timing the kernel
- Performance and Scalability Analysis Using `nvprof`



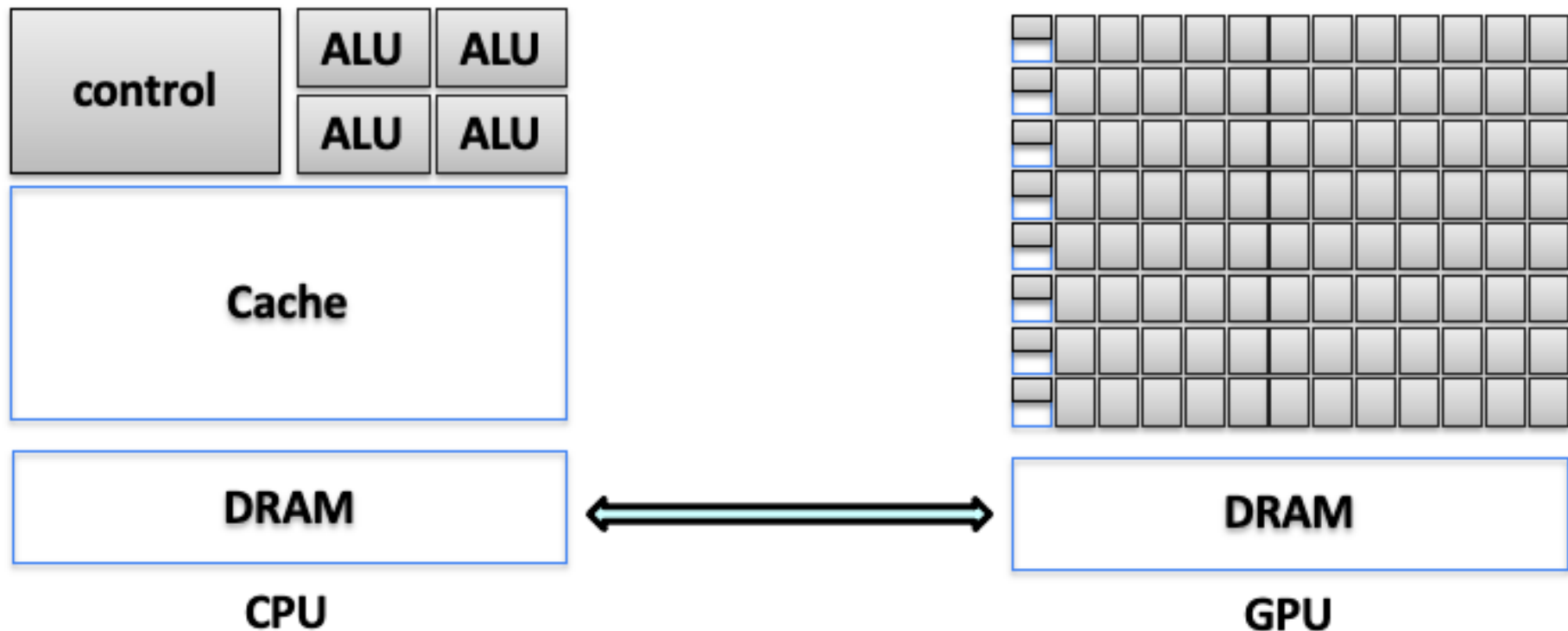
5.1 Foundations of GPU Acceleration and CUDA

GPUs in HPC and AI

- **1999: NVIDIA introduced the first GPU, designed for real-time 3D rendering**
- **GPUs are now central to supercomputing infrastructures**
 - Widely used for machine learning and scientific workloads
 - Neural network training relies on matrix multiplications and convolutions, which map well to GPU parallelism
- **Primary goals of GPU integration:**
 - Maximize throughput (large volume of parallel operations)
 - Minimize execution time and energy consumption

Host–Device Model

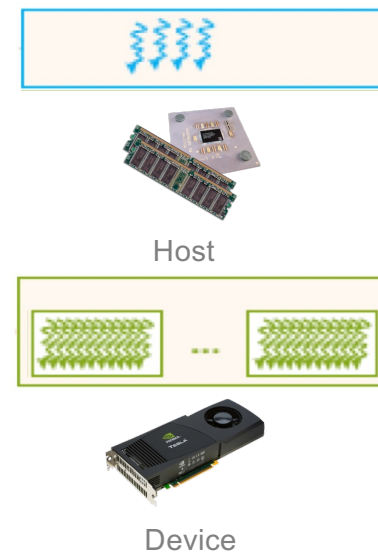
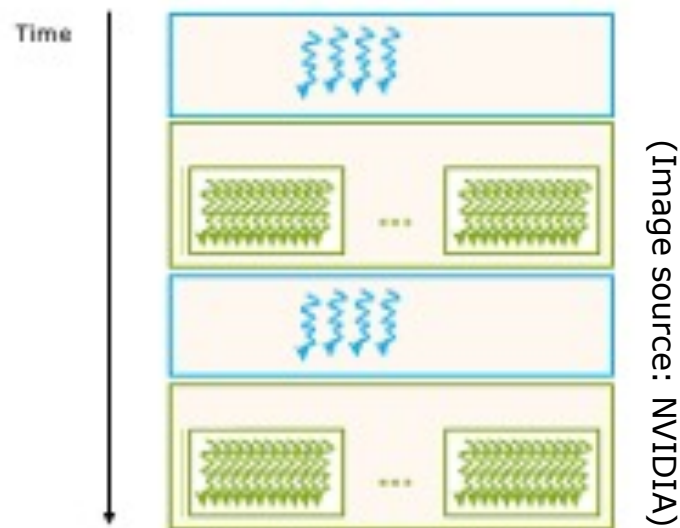
- Visual CPU (Host) – GPU (Device) split and memory hierarchies



(Adapted from Agustín Fernández – DAC/UPC)

Heterogeneous Computing

- **Applications partitioned across CPU + GPU**
 - **Host** code (CPU): control, memory orchestration, kernel launches
 - **Device** code (GPU): parallel kernels over thousands of threads





CUDA: A Programming Platform for GPUs

CUDA

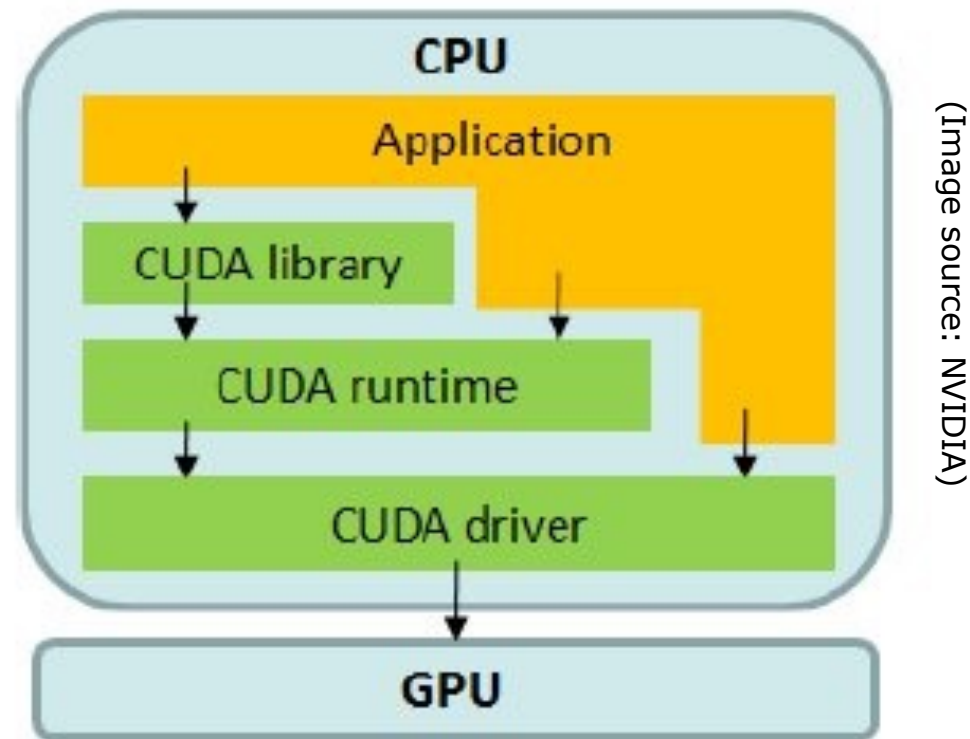
- Introduced by NVIDIA to support general-purpose GPU computing



- **CUDA (Compute Unified Device Architecture):**
 - Parallel computing platform
 - Programming model for heterogeneous systems
- **Goal:**
 - unlock GPU acceleration for a broad range of applications
- **Integrated into C/C++ workflows**
 - Developers can offload parallel tasks to GPUs without leaving familiar programming paradigms

The CUDA Software Stack

- CUDA provides a layered software ecosystem



CUDA software stack: libraries, runtime, and driver layers enabling host-device interaction (Image source NVIDIA)

The CUDA Software Stack

- **Levels of abstraction:**

- CUDA Libraries:
 - pre-built, optimized algorithms with near-peak performance
- CUDA Runtime API:
 - higher-level management of memory, kernel launches, and synchronization (used in most applications)
- CUDA Driver API:
 - low-level access for fine-grained control and system-level integration

- **This modular design allows both productivity for beginners and performance tuning for experts**

Core CUDA Libraries for Accelerated Computing

- **Main libraries for Deep Learning and AI**

- **cuDNN**: primitives for convolution, pooling, normalization, activations, RNNs, transformers
- **cuBLAS**: dense linear algebra (GEMM ops central to neural networks)
- **CUTLASS**: customizable matrix operations, fusion, modular design
- **NCCL**: optimized collective communication (AllReduce, Broadcast, AllGather) across GPUs

- **Widely integrated in AI frameworks**

- PyTorch, TensorFlow, JAX, Hugging Face

Core CUDA Libraries for Accelerated Computing

- **CUDA supports many additional fields:**
 - cuRAND: random number generation
 - cuSOLVER: linear solvers and decompositions
 - cuSPARSE: sparse matrix operations
 - cuFFT: Fast Fourier Transforms
 - NPP: image & video processing primitives
 - Thrust: STL-like C++ parallel algorithms
 - nvGRAPH: graph analytics
 - Video Codec SDK: accelerated video encoding/decoding

Framework Integration & Ecosystem

- **Modern frameworks abstract CUDA details:**
 - Automatic GPU memory management
 - Automatic kernel execution & synchronization
 - Multi-GPU support via distributed APIs (e.g., torch.distributed, tf.distribute)
- **NVIDIA ecosystem support:**
 - Deep Learning SDK
 - NVIDIA GPU Cloud (NGC)
 - preconfigured containers, inference engines, profiling tools, optimized libraries



5.2

Getting Started with CUDA: Threads, Kernels, and Memory

Hello World in CUDA: Introduction

■ **Compilation process:**

- Host code → compiled with standard C compiler
- Device code → compiled with CUDA extensions
- Linked together with CUDA runtime libraries

■ **Compilation on MN5:**

```
module load cuda
```

```
nvcc <filename>.cu -o <filename>
```

```
which nvcc
```


Hello World in CUDA: Introduction

- Hello World from CPU

```
// hello.cu
#include <stdio.h>

int main(void)
{
    printf("Hello World from CPU\n");
}
```

Hello World in CUDA: Introduction

- Execution (Hello World **from GPU**)

```
$nvcc hello.cu -o hello  
$ ./hello
```

Hello World from CPU

- Although compiled with `nvcc`, this example Hello World program runs only on the CPU
- Behavior is identical to a standard C program

Hello World in CUDA: Introduction

■ Hello World from GPU

```
// helloGPU.cu
#include <stdio.h>

__global__ void helloFromGPU(void)
{
    printf("Hello World from GPU\n");
}

int main(void)
{
    printf("Hello World from CPU\n");
    helloFromGPU<<<1, 3>>>>(); // Launch 1 block with 3 threads
    cudaDeviceReset();
    return 0;
}
```

Hello World in CUDA: Introduction

- Execution (Hello World **from GPU**)

```
$ nvcc helloGPU.cu -o helloGPU  
$ ./helloGPU
```

```
Hello World from CPU  
Hello World from GPU  
Hello World from GPU  
Hello World from GPU
```

Understanding the Output:

- "Hello World from CPU" printed once by the CPU main thread
- GPU kernel `helloFromGPU` launched with `<<<1, 3>>>`
1 block, 3 threads
Each thread prints independently → **3 messages from GPU**
- `cudaDeviceReset()` ensures proper cleanup of GPU resources

Task 5.1

■ Task 5.1 – Your First Hello World in CUDA

Reproduce the results presented in this section. Doing so will help you verify that the CUDA environment is correctly set up and that you are able to compile and execute a basic CUDA program on your system.



Thread Organization

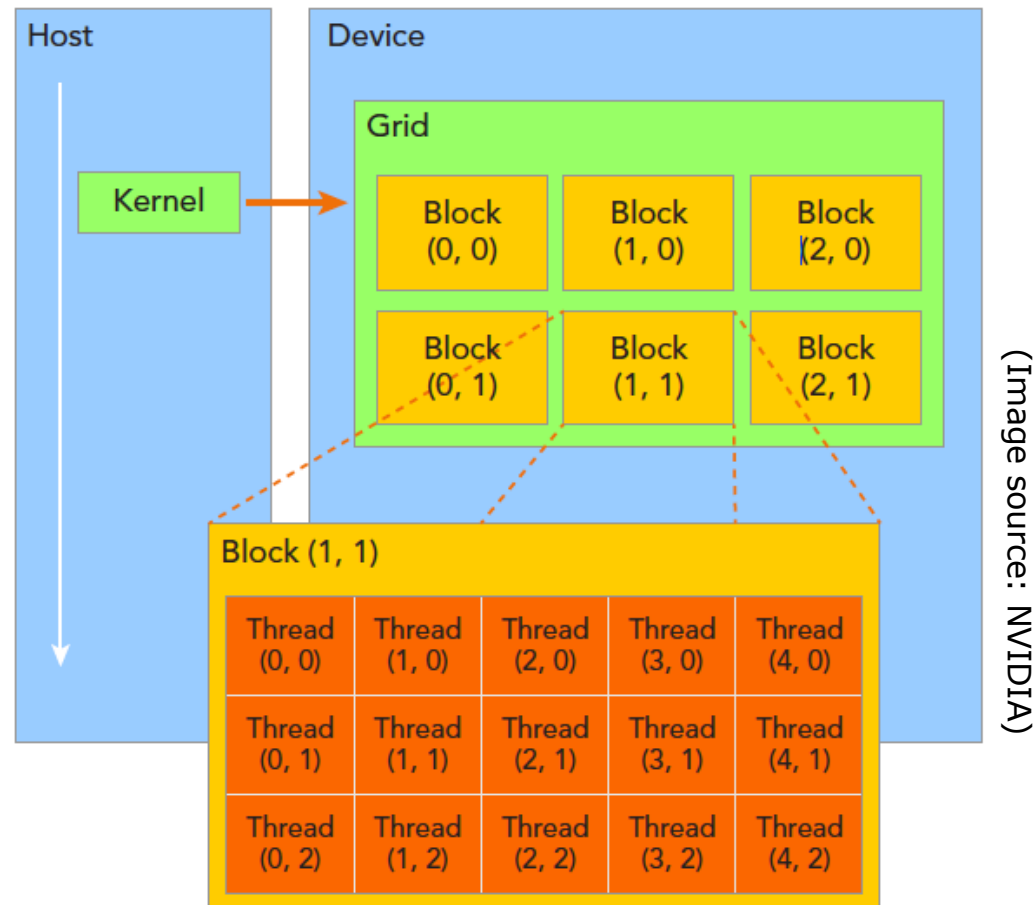
Threads Organization: Overview

- **In CUDA, a thread is the smallest unit of execution**
 - Each thread performs a computation independently
 - GPUs can launch hundreds or thousands of threads in parallel
- **CUDA exposes a **two-level thread** hierarchy decomposed into blocks of threads and grids of blocks:**
 - **Grids** → collections of blocks
 - **Blocks** → collections of threads
- **All threads execute the same kernel code but operate on different data elements**

Threads Organization: Overview

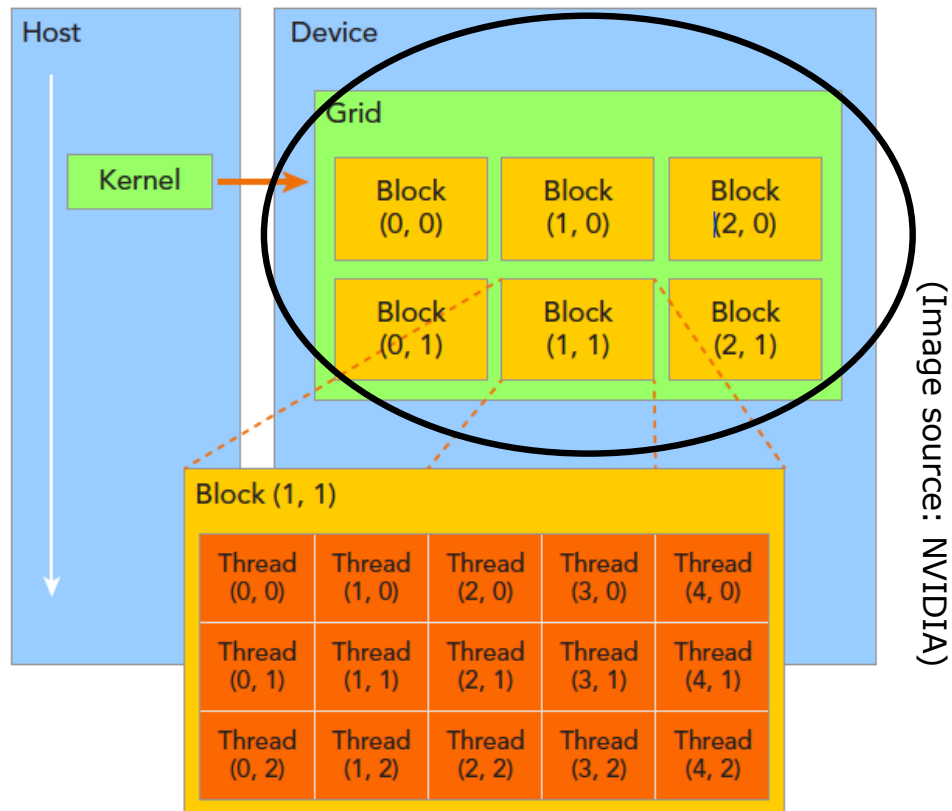
■ Visual Example

CUDA thread hierarchy: 2D grid of 3×2 blocks, each block 5×3 threads)



Organizing Threads

- All threads spawned by a single kernel launch are collectively called a **grid**.

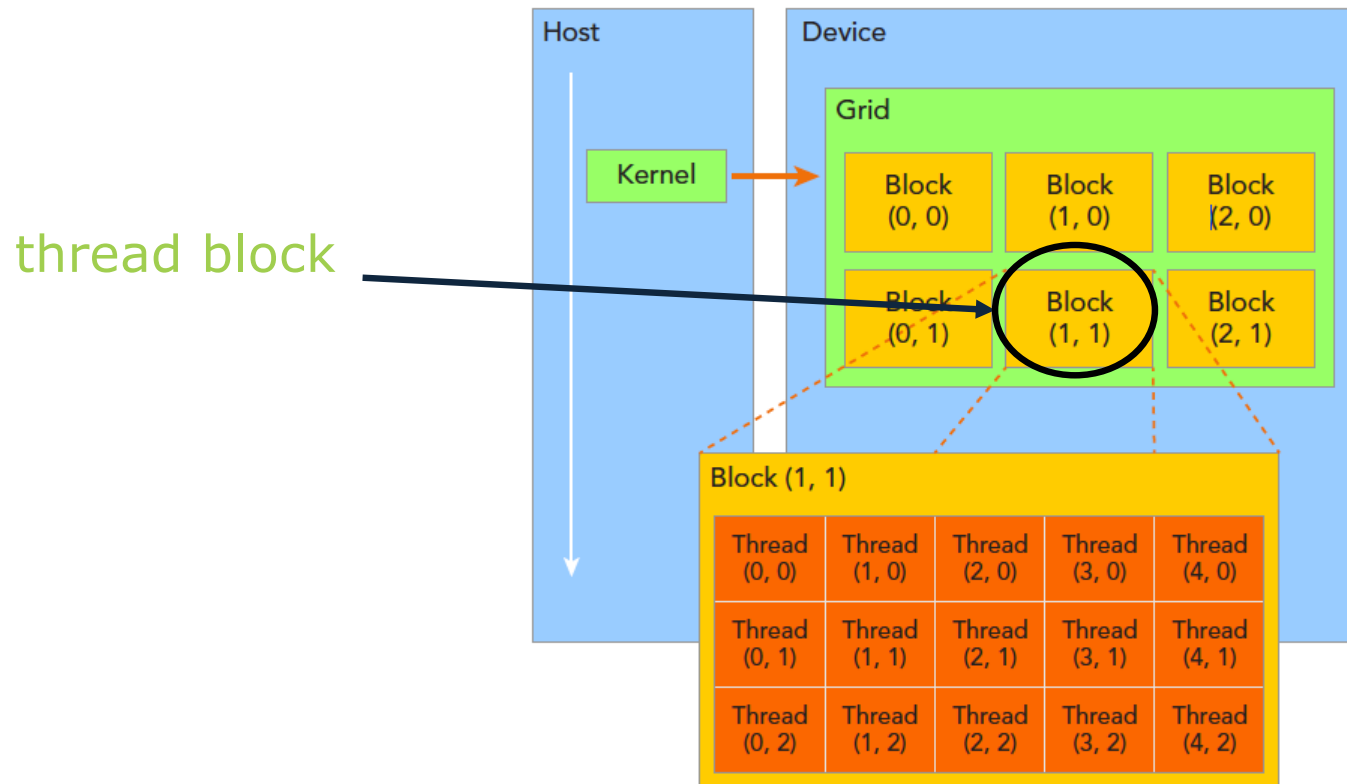


- A Grid can be a 3D structure of blocks
Max grid dim (x,y,z): $(2^{31}-1, 2^{16}-1, 2^{16}-1)$
- A block can be a 3D structure of threads
Max block dims (x,y,z): $(2^{10}, 2^{10}, 2^6)$
 - There is a limit on maximum no. of threads per block so (in H100 it is 1024 which implies that a kernel with block dim (x,y,z), has to have $x+y+z \leq 1024$)



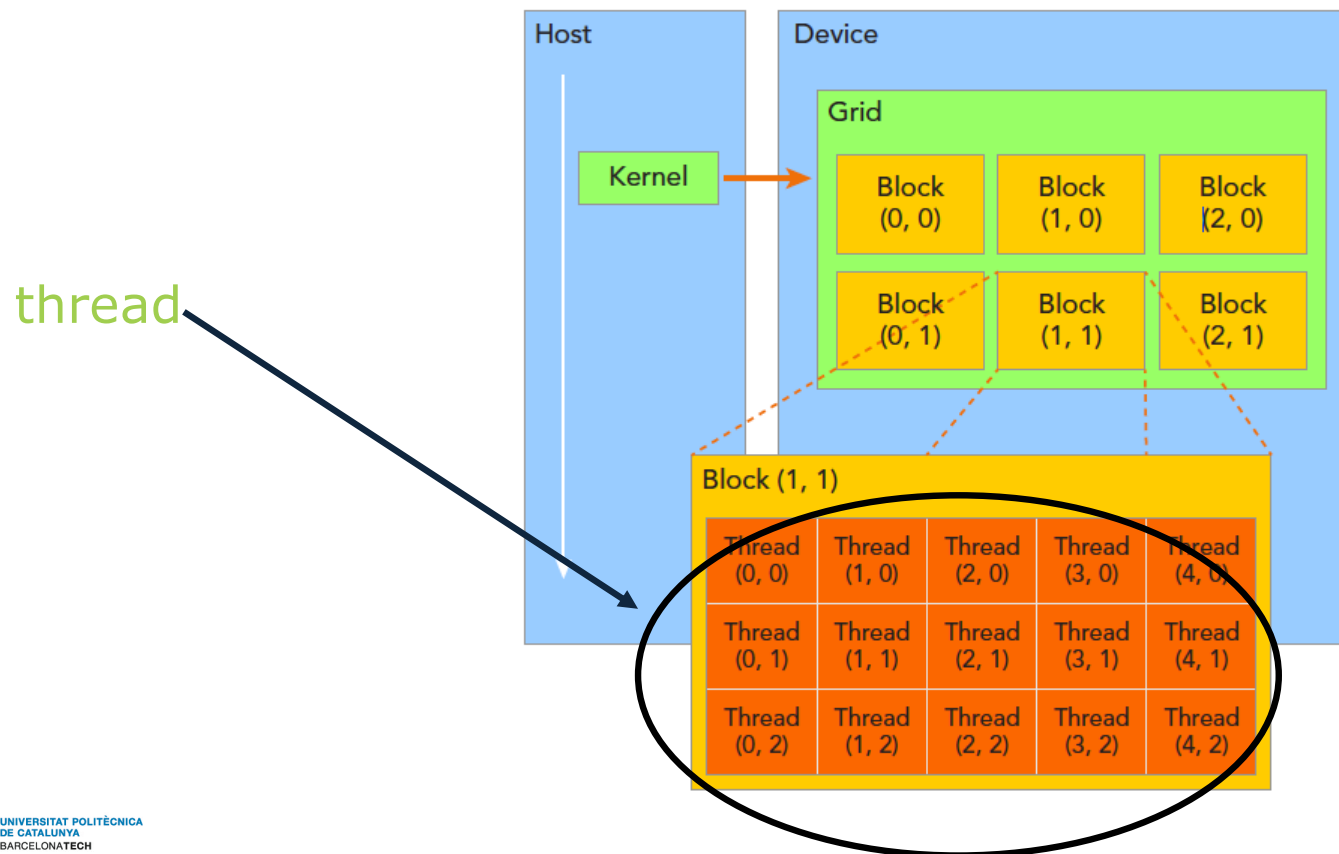
Organizing Threads

- A grid is made up of many **thread blocks**:
 - All threads in a block **share** the same **global memory** space.



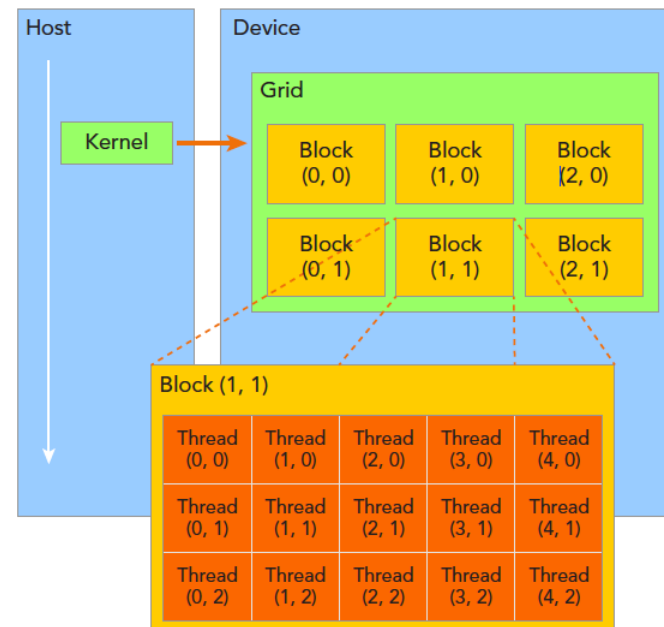
Organizing Threads

- A **thread block** is a group of threads that **can cooperate** with each other using Block-local synchronization, or Block-local shared memory.
- **Threads from different blocks cannot cooperate.**



Organizing Threads

- Threads rely on two unique coordinates to distinguish themselves from each other: (*)
 - `blockIdx` : block index within a grid
 - `threadIdx` : thread index within a block



(*) These variables are assigned to each thread by the CUDA runtime and can be accessed within kernel functions.

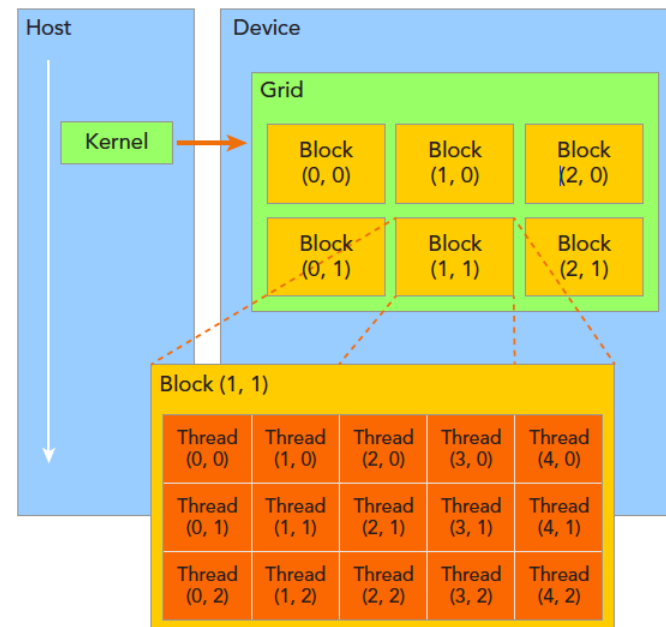
Organizing Threads

- The coordinate variable is of type **uint3**, a CUDA built-in vector type derived from the basic integer type:

- `blockIdx.x`
- `blockIdx.y`
- `blockIdx.z`

- `threadIdx.x`
- `threadIdx.y`
- `threadIdx.z`

Visual example: a thread hierarchy structure with a **2D grid** containing **2D blocks**



Organizing Threads: **blockDim & gridDim**

- The dimensions of a grid and a block are specified by:
 - **blockDim**: block dimension, measured in threads.
 - **gridDim**: grid dimension, measured in blocks.
- These variables are of type **dim3** (*)
- And are accessible through its x, y, and z fields:
 - **blockDim.x**
 - **blockDim.y**
 - **blockDim.z**
 - **gridDim.x**
 - **gridDim.y**
 - **gridDim.z**

(*) When defining a variable of type dim3, any component left unspecified is initialized to 1

Example:

Configuring 1D Grid and Block

■ `checkIndex.cu`

```
#include <cuda_runtime.h>
#include <stdio.h>
/*
 * Display the dimensionality of a thread block and grid
 * from the host and device.
 */

int main(int argc, char **argv)
{
    // define total data element
    int nElem = 6;

    // define grid and block structure
    dim3 block(3);
    dim3 grid((nElem + block.x - 1) / block.x);
```

Example:

Configuring 1D Grid and Block

■ `checkIndex.cu` (cont)

```
// check grid and block index from host side
printf("\ncheck grid and block dimension from host side-->\n grid.x=%d grid.y=%d grid.z=%d\n | block.x=%d block.y=%d block.z=%d\n", grid.x, grid.y, grid.z , block.x, block.y, block.z);

// check grid and block dimension from the device side
printf("\n check grid/block index from kernel side-->\n");
checkIndex<<<grid, block>>>();

// reset the device before you leave
cudaDeviceReset();
return 0;
}
```

Example:

Configuring 1D Grid and Block

■ Initialization

```
int nElem = 6;  
dim3 block(3);  
dim3 grid((nElem + block.x - 1) / block.x);
```

- nElem = 6 data elements
- Block with 3 threads
- Grid size computed using ceiling division
- Ensures all data elements are processed, even if not a perfect multiple of block size

CODE SOURCE : Book Professional CUDA C Programming, by John Cheng, Max Grossman, Ty McKercher. Wrox Ed - Wiley 2014

Example: Printing Indices

- Host code defines grid and block, launches kernel, and prints information

```
printf("grid.x = %d grid.y = %d grid.z = %d\n", grid.x, grid.y, grid.z);  
printf("block.x = %d block.y = %d block.z = %d\n", block.x, block.y, block.z);
```

Example: Printing Indices

- **In the kernel function**, each thread can print out its own thread index, block index, block dimension, and grid dimension as follows:

```
__global__ void checkIndex(void)
{
    printf("threadIdx:(%d, %d, %d) blockIdx:(%d, %d, %d)
           blockDim:(%d, %d, %d) gridDim:(%d, %d, %d)\n",
           threadIdx.x, threadIdx.y, threadIdx.z,
           blockIdx.x, blockIdx.y, blockIdx.z,
           blockDim.x, blockDim.y, blockDim.z,
           gridDim.x, gridDim.y, gridDim.z);
}
```

- On the device side, within a kernel, each thread can access:
 - threadIdx to get its position in the block
 - blockIdx to get the block's position in the grid
 - blockDim and gridDim to know the size of each dimension

Example Output

```
% ssh nct01002@allogin1.bsc.es
```

```
[nct01002@allogin1 ~]$ vi checkIndex.cu
```

```
[nct01002@allogin1 ~]$ nvcc checkIndex.cu -o checkIndex
```

```
[nct01002@allogin1 ~]$ ./checkIndex
```

check grid and block dimension from host side-->

```
grid.x=2 grid.y=1 grid.z=1 | block.x=3 block.y=1 block.z=1
```

check grid and block dimension from kernel side-->

```
threadIdx:(0, 0, 0) blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
```

```
threadIdx:(1, 0, 0) blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
```

```
threadIdx:(2, 0, 0) blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
```

```
threadIdx:(0, 0, 0) blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
```

```
threadIdx:(1, 0, 0) blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
```

```
threadIdx:(2, 0, 0) blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
```


Task 5.2

■ Task 5.2 – Dimensionality of a Thread Block and Grid

1. Write a CUDA program that prints the dimensionality of the thread block and grid from both the host and device sides.
2. Compile and execute the program interactively in your MN5 terminal.
3. Confirm that the output matches expectations, and that all thread indices are correctly reported.



Memory Management

Memory Management in CUDA

- **Data for GPU must be explicitly transferred (have separate memory spaces)**
 - host → device → host
- **Memory management is crucial for:**
 - **Correctness** (data available in right memory)
 - **Performance** (avoiding unnecessary transfers)
- **Efficient CUDA programs:**
 - Minimize transfers
 - Overlap transfers with computation

CUDA Memory Management Functions

- **cudaMalloc:** allocate memory on device
- **cudaFree:** free device memory
- **cudaMemcpy:** transfer data between host & device

```
cudaError_t cudaMemcpy ( void* dst, const void* src,  
                        size_t count, cudaMemcpyKind kind )
```

Transfer direction specified with kind:

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

STANDARD C FUNCTIONS	CUDA C FUNCTIONS
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

Example: Integer Addition

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

```
// CUDA kernel function to add two numbers  
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
    printf("GPU: computed %d + %d = %d\n", *a, *b, *c);  
}
```

```
int main(void) {  
    int a, b, c; // host copies of a, b, c  
    int *d_a, *d_b, *d_c; // device copies of a, b, c  
    int size = sizeof(int);  
  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **)&d_a, size);  
    cudaMalloc((void **)&d_b, size);  
    cudaMalloc((void **)&d_c, size);  
    // Setup input values  
    a = 2;  
    b = 7;
```

Example: Integer Addition

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Print the result on the host
printf("CPU: received result %d + %d = %d\n", a, b, c);

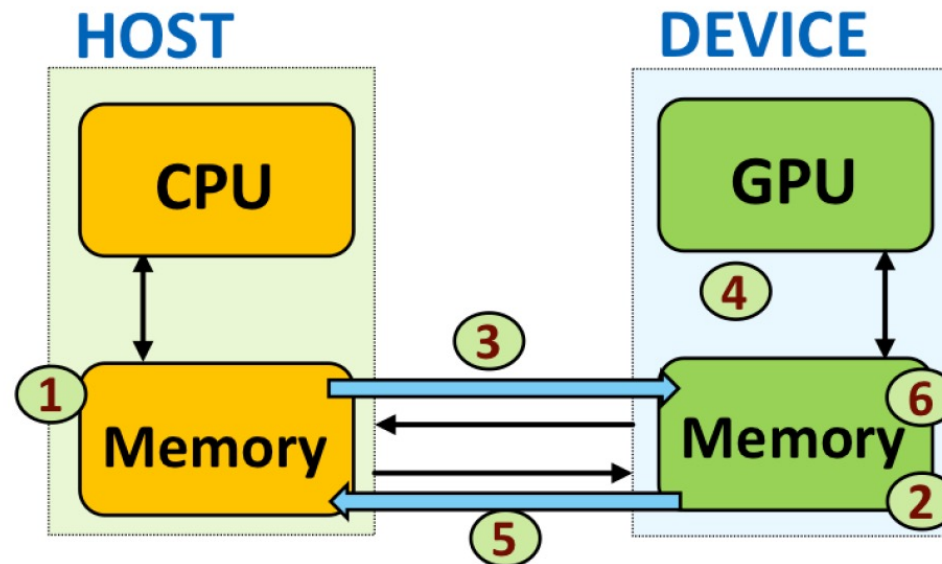
// Cleanup
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return 0;

}
```


Example: Integer Addition

- Order of memory transfer steps in a typical CUDA program.



- Output:

```
$ nvcc add.cu -o add
$ ./add
```

```
GPU: computed 2 + 7 = 9
CPU: received result 2 + 7 = 9
```

Task 5.3

■ Task 5.3 – Investigating Parallel Execution with Multiple Threads

- Modify previous CUDA program to launch kernel with 4 threads:

```
add<<<1, 4>>>(d_a, d_b, d_c);
```

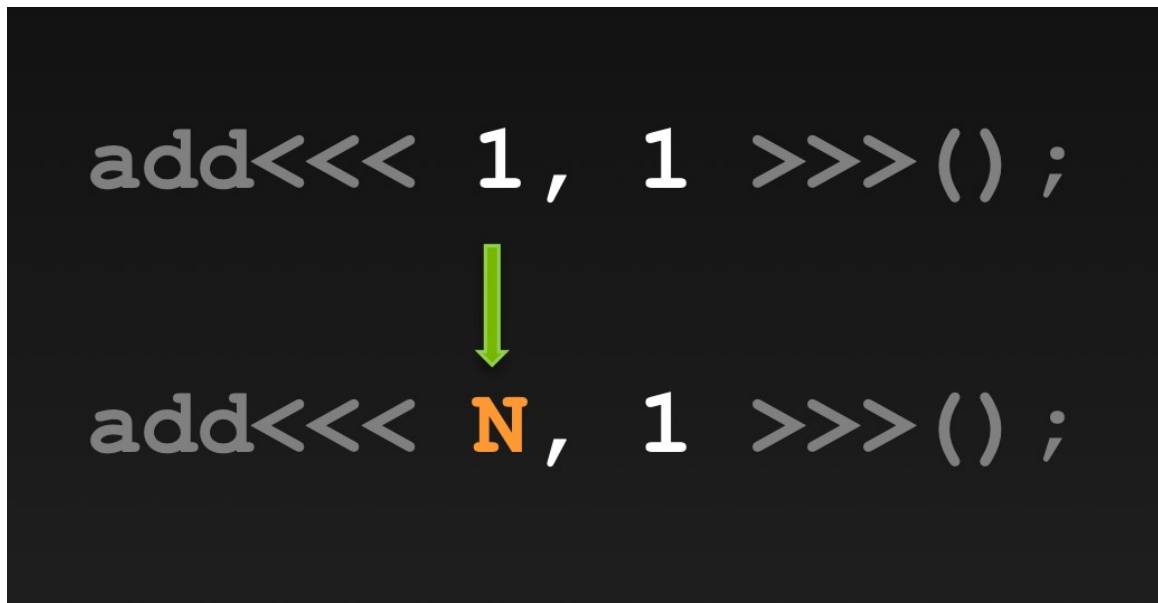
- Compile and execute
- Observe the output:
 - Why does each thread print the same operation and result?
 - What happens if multiple threads access the same memory position without coordination?



Launching a CUDA Kernel

Moving to Parallel

- GPU computing is about massive parallelism
 - So , how do we run code in parallel on the device?



The diagram illustrates the concept of parallelism in GPU computing. It shows two lines of code on a dark background. The first line is `add<<< 1, 1 >>> () ;`. A green arrow points down from the number '1' in the first parameter to the letter 'N' in the second line of code, `add<<< N, 1 >>> () ;`. The 'N' is highlighted in orange, indicating that the operation is now executed N times in parallel.

- Instead of executing `add()` once, execute N times in parallel
- With `add()` running in parallel we can do vector addition, etc.

CUDA Kernel

- A CUDA kernel is a special function executed concurrently by many threads on the GPU.
- Launched with the triple-chevron syntax:

```
kernel_name <<<grid, block>>>(argument list);
```

- **Parameters:**
 - `grid` = number of blocks
 - `block` = number of threads per block
 - `argument_list` = kernel input data
- **This configuration defines the total number of threads and their organization.**

Mapping Threads to Array Elements

- **CUDA provides hierarchical organization:**

- `threadIdx.x` → position of the thread inside its block
- `blockIdx.x` → block position inside the grid

- **Global index formula:**

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

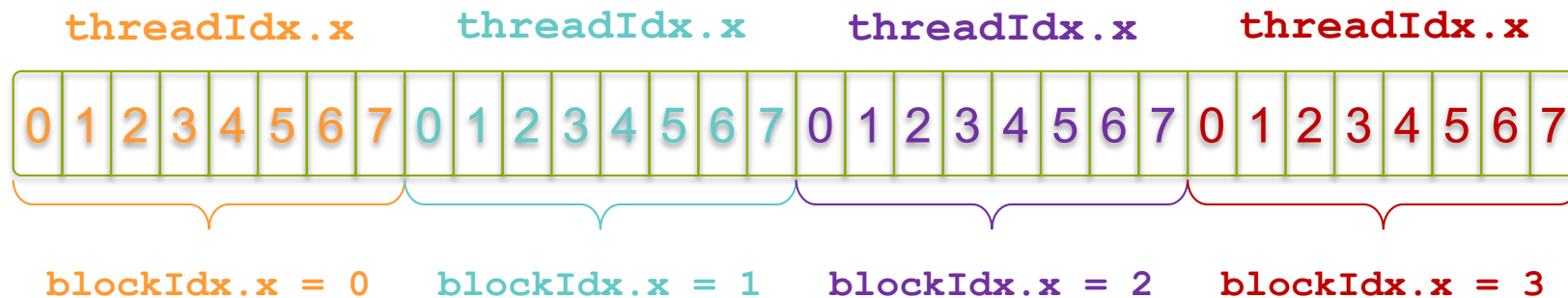
- **Example:**

```
kernel_name<<<4, 8>>>(...)
```

- 4 blocks × 8 threads = 32 threads
- Each thread processes one element of the array

Indexing Arrays: Example

- Layout of threads in the above configuration:



- Indexing Arrays with Blocks and Threads:

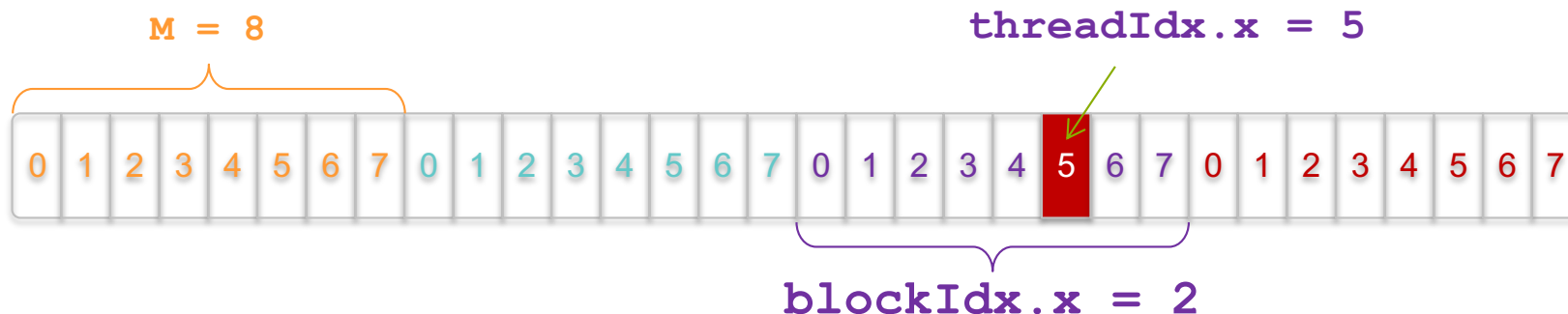
- With M threads/block a unique index for each thread is given by:

$$\text{index} = \text{threadIdx.x} + \text{blockIdx.x} * M;$$

Indexing Arrays: Example

■ Example of Thread Index Calculation

- Suppose: `threadIdx.x = 5` in `blockIdx.x = 2` with `blockDim.x = 8`
- Global index = $5 + 2 * 8 = 21$
- This corresponds to the 21st element in the array



```
int index = threadIdx.x + blockIdx.x * M;  
          =      5      +      2      * 8;  
          = 21;
```

Example: Vector Addition

- **How to index?**

- Use the built-in variable `blockDim.x` for threads per block.

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

- **Combined version of `add()` to use parallel threads *and* parallel blocks**

```
__global__ void add(int *a, int *b, int *c, int n) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if (index < n)  
        c[index] = a[index] + b[index];  
}
```

- **Launching kernel:**

```
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>>(d_a, d_b, d_c, N);
```

- **Each thread computes one element independently.**

Example: Vector Addition

```
#define N 4194304
#define THREADS_PER_BLOCK 512

int main(void) {
    int *a, *b, *c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Source: NVIDIA

Example: Vector Addition

```
// Copy inputs to device
```

```
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

```
// Launch add() kernel on GPU
```

```
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

```
// Copy result back to host
```

```
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```

```
// Cleanup
```

```
free(a); free(b); free(c);
```

```
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

```
return 0;
```

```
}
```

Source: NVIDIA

Handling Arbitrary Vector Sizes

- Often N is not divisible by the block size.
- Safeguard with conditional check inside the kernel:

```
if (index < n) {  
    c[index] = a[index] + b[index];  
}
```

- Grid size computed with ceiling division:

```
int blockSize = THREADS_PER_BLOCK;  
int gridSize = (N + blockSize - 1) / blockSize;  
add<<<gridSize, blockSize>>>(d_a, d_b, d_c, N);
```

Task 5.4

■ Task 5.4 – Element-wise Vector Addition Using CUDA

1. Write a complete CUDA program that performs vector addition as described.
2. Initialize two input vectors on the host.
3. Allocate memory and copy them to the device.
4. Launch the kernel and retrieve the result.
5. Print both input and output vectors to verify correctness.
6. Run the program interactively in your MN5 terminal.



Topics Beyond the Basics

Expanding the CUDA Skillset

- **CUDA development extends far beyond basic kernel execution and memory copying.**
- **Advanced topics include:**
 - Optimizing memory usage – leveraging shared, constant, and texture memory
 - Improving memory access patterns – ensuring coalesced and aligned memory transactions
 - Fine-tuning performance – exploiting instruction-level parallelism and GPU profiling tools
- **These techniques are key to achieving performance close to the theoretical limits of the hardware.**

Optimizing Memory Usage

- **CUDA provides multiple memory spaces, each with distinct performance traits:**
 - Shared memory: on-chip, low-latency, cooperative among threads in a block
Constant memory: cached, ideal when all threads read the same data
 - Texture memory: optimized for spatial locality and read-heavy workloads
- **Techniques such as tiling minimize redundant global memory accesses.**
- **Effective memory design often provides greater performance gains than algorithmic changes.**

Memory Coalescing and Alignment

- Threads in a warp should access contiguous, properly aligned memory addresses.
- This allows coalesced memory access, where multiple reads/writes merge into a single transaction.
- **Benefits:**
 - Lower latency
 - Reduced bandwidth pressure
 - Improved overall kernel efficiency
- **Misaligned or scattered accesses can drastically reduce throughput.**

Performance Tuning and Profiling

- **Achieving maximum performance requires deep understanding of GPU execution.**
- **Common optimization techniques:**
 - Loop unrolling
 - Instruction-level parallelism
 - Fused multiply-add operations
- **Profiling tools:**
 - Nsight Compute and Nsight Systems (from NVIDIA)
 - CUPTI interface for low-level metrics
 - Paraver (at BSC): for visualizing Nsight traces with fine-grained temporal analysis

Portability and High-Level Programming Models

- **CUDA is not the only way to program GPUs efficiently.**
- **Directive-based models such as:**
 - OpenACC
 - OpenMP Offloading
- **Advantages:**
 - Simplify GPU acceleration for large or legacy codebases
 - Enable incremental parallelization
 - Facilitate adoption in multidisciplinary teams
- **Developers can combine these models with CUDA to balance control and productivity.**

The Evolving CUDA Ecosystem

- **Each GPU generation introduces architectural improvements:**
 - New memory hierarchies
 - Enhanced scheduling
 - Expanded Tensor Core capabilities
- **The CUDA software stack evolves in parallel:**
 - Updated APIs and libraries
 - Advanced debuggers and profilers
 - Integration with containerized and cloud-based workflows
- **Staying up to date with these tools ensures scalable and future-ready GPU development.**



5.3

Case study: Matrix Multiplication in CUDA

Handling Errors in CUDA

- **CUDA API calls are often asynchronous, which complicates debugging.**
- **Solution:** wrap calls with a custom error-checking macro and function.

```
#define err(format, ...) do { fprintf(stderr, format, ##__VA_ARGS__); exit(1); } while (0)

inline void checkCuda(cudaError_t e) {
    if (e != cudaSuccess) {
        err("CUDA Error %d: %s\n", e, cudaGetErrorString(e));
    }
}
```

- **Example usage:**

```
checkCuda(cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice));
```

- If an error occurs, it prints a human-readable message and stops execution.

Handling Errors in CUDA

- **Benefits of Centralized Error Handling**
 - Simplifies debugging by localizing error reporting.
 - Provides clear diagnostic messages for asynchronous API calls.
 - Helps identify:
 - Invalid memory access
 - Out-of-memory conditions
 - Kernel launch failures
- **Use `checkCuda()` systematically after all CUDA API calls.**

Managing Flattened Matrices

- CUDA device memory is 1D; it does not support direct 2D indexing.

- Matrices are stored as flattened arrays:

`matrix[i * N + j]`

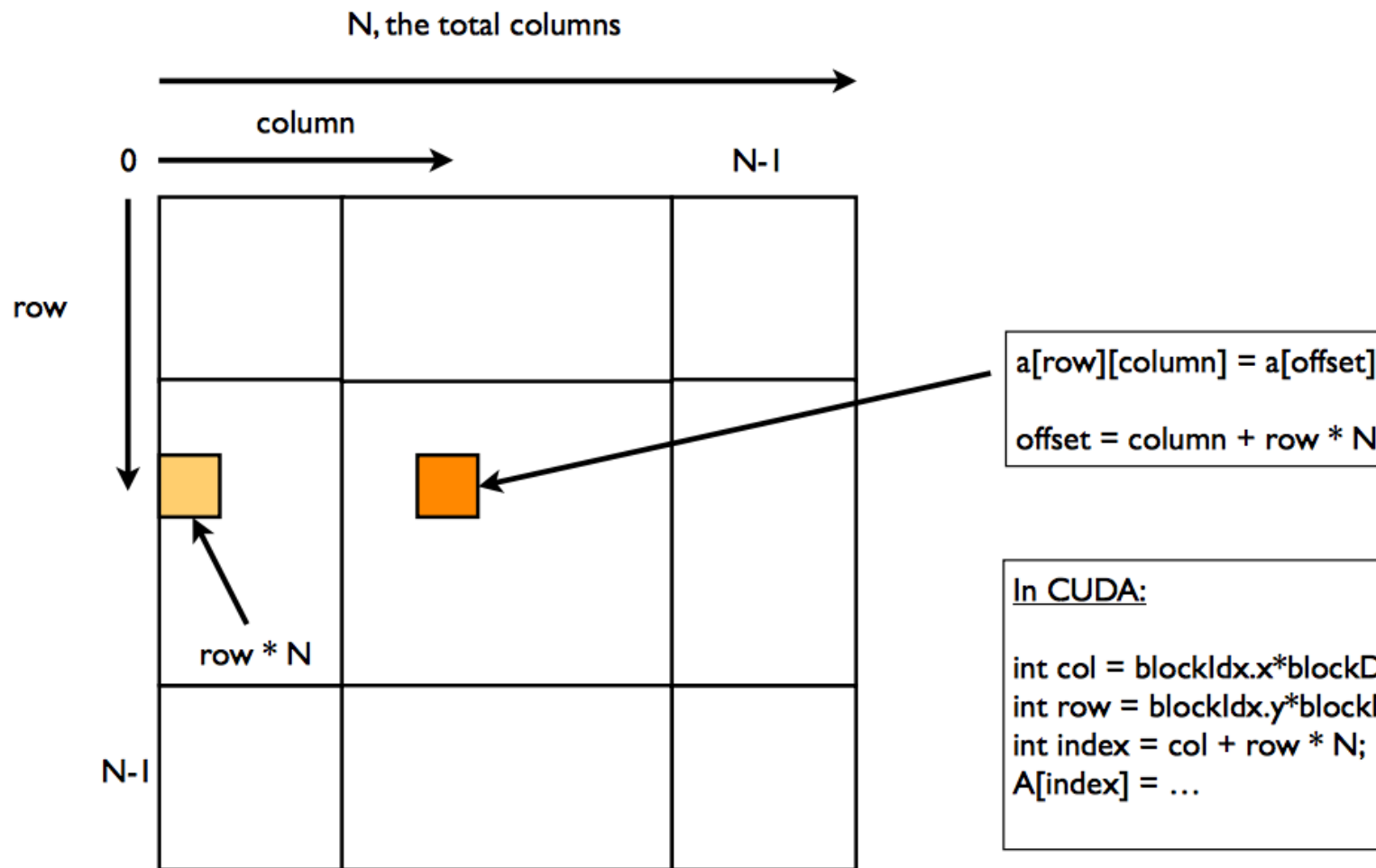
$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

- Each thread computes one element of matrix C.

Case Study: Matrix Multiplication

■ Accessing Matrices in Linear Memory



CUDA Kernel for Matrix Multiplication

```
__global__ void matrixProduct(double *matrix_a, double *matrix_b,
                              double *matrix_c, int width) {

    double sum = 0;
    int row = threadIdx.y + blockDim.y * blockIdx.y;
    int col = threadIdx.x + blockDim.x * blockIdx.x;
    if (col < width && row < width) {
        for (int k=0; k<width; k++) {
            sum += matrix_a[row * width + k] * matrix_b[k * width + col];
        }
        matrix_c[row * width + col] = sum;
    }
}
```

Each GPU thread multiplies one row of A by one column of B.

Host Code: Execution Flow

1. Initialize matrices A and B with random values.
2. Allocate memory on the GPU.
3. Copy matrices A and B to the device.
4. Configure grid and block dimensions:

```
dim3 dimBlock(BLOCK_SIZE_DIM, BLOCK_SIZE_DIM);  
dim3 dimGrid((N + BLOCK_SIZE_DIM - 1) / BLOCK_SIZE_DIM,  
             (N + BLOCK_SIZE_DIM - 1) / BLOCK_SIZE_DIM);
```

5. Launch kernel:

```
matrixProduct<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, N);
```

Synchronization and Data Transfer

- CUDA kernel launches are asynchronous relative to the host.

- **Always synchronize before accessing results:**

```
checkCuda(cudaDeviceSynchronize());  
checkCuda(cudaGetLastError());
```

- **Copy result matrix back to host:**

```
checkCuda(cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost));
```

- **Finally, free device memory and reset the GPU.**

Host code I

```
int main() {
    size_t size = N * N * sizeof(double);
    double h_a[N][N], h_b[N][N], h_c[N][N];
    double *d_a, *d_b, *d_c;

    initializeMatrices(h_a, h_b);

    // Allocate memory on device
    checkCuda(cudaMalloc((void **) &d_a, size));
    checkCuda(cudaMalloc((void **) &d_b, size));
    checkCuda(cudaMalloc((void **) &d_c, size));

    // Copy matrices to device
    checkCuda(cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice));
    checkCuda(cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice));

    // Define grid and block dimensions
    dim3 dimBlock(BLOCK_SIZE_DIM, BLOCK_SIZE_DIM);
    dim3 dimGrid((N + BLOCK_SIZE_DIM - 1) / BLOCK_SIZE_DIM,
                 (N + BLOCK_SIZE_DIM - 1) / BLOCK_SIZE_DIM);
```

Host code II

```
// Launch kernel
matrixProduct<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, N);

// Wait for completion and check errors
checkCuda(cudaDeviceSynchronize());
checkCuda(cudaGetLastError());

// Copy result back to host
checkCuda(cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost));

// Free device memory
checkCuda(cudaFree(d_a));
checkCuda(cudaFree(d_b));
checkCuda(cudaFree(d_c));

showResults(h_a, h_b, h_c);
cudaDeviceReset();
return 0;
}
```


Naming Conventions

- To improve code readability:

Type	Prefix	Example
Host variables	h_	h_a, h_c
Device variables	d_	d_a, d_c

- Consistent naming clarifies which memory space a variable belongs to, reducing programming errors.

Task 5.5

■ Task 5.5: Parallel Matrix Multiplication with CUDA

- Objective: Implement a program that computes $C = A \times B$ for square matrices of size $N = 4$.
- Instructions:
 - Initialize matrices A and B with random values.
 - Allocate GPU memory and copy both matrices to the device.
 - Launch the kernel using a 2D grid and block configuration.
 - Copy the resulting matrix back to the host.
 - Print matrices A, B, and C to verify results.
- Note:

Execution time for $N = 512$ is modest—use an interactive login node for development.

Allocating a GPU with SLURM

■ GPU Allocation Directive:

```
#SBATCH --gres=gpu:1
```

- The flag `--gres` (generic resource) tells SLURM to allocate a specific GPU resource.
- `gpu:1` reserves one GPU for your job, ensuring exclusive access during execution.

■ Purpose:

- This guarantees that the CPU process running the CUDA application is correctly associated with a GPU device, enabling proper offloading of compute tasks.

Complete SLURM Configuration

- In addition to GPU allocation, define the number of CPU tasks for the job:

```
#SBATCH --ntasks=1
```

- --ntasks specifies the number of **CPU-side** processes.

- For most CUDA applications, a single process is sufficient.

- This process handles:
 - CUDA initialization
 - Memory allocation and transfers
 - Kernel launching on the GPU
 - Result retrieval to host memory

Task 5.6

■ Task 5.6 – Running CUDA Jobs with SLURM

- Repeat the matrix multiplication experiment from Task 5.5, but this time submit it as a batch job using SLURM.
- You should:
 - Request one GPU with `--gres=gpu:1`
 - Launch one task with `--ntasks=1`
 - Load the CUDA environment: `module load cuda`
 - Execute your compiled program
- Goal:

Understand how to execute CUDA applications in a production HPC environment, ensuring proper GPU allocation and CPU–GPU binding.

Timing the Kernel (profiling quickstart)

- **CUDA profiling today: Nsight Systems (nsys) & Nsight Compute (ncu)**
- **For command-line + zero setup on most clusters: use nsys nvprof mode**
 - Ships with CUDA toolkits; works on MN5 login/compute nodes
- **What you get:**
 - Kernel timelines & durations
 - cudaMemcpy transfer stats (HtoD / DtoH)
 - CUDA API call costs (e.g., cudaMalloc, cudaDeviceSynchronize)
- **Run**

```
nsys nvprof ./matrixMulN
```



```
# or include args: nsys nvprof ./matrixMulN <args>
```

Sample nsys nvprof report

[4/7] Executing 'cuda_api_sum' stats report

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
72.1	90467036	3	30155678.7	60989.0	57064	90348983	52128930.7	cudaMalloc
25.8	32375626	1	32375626.0	32375626.0	32375626	32375626	0.0	cudaDeviceReset
1.2	1546393	3	515464.3	236710.0	225481	1084202	492573.3	cudaMemcpy
0.3	405734	444	913.8	90.0	73	45814	5775.5	cuDeviceGetAttribute
0.3	401131	1	401131.0	401131.0	401131	401131	0.0	cudaLaunchKernel
0.2	230855	3	76951.7	60610.0	58217	112028	30400.6	cudaFree
0.1	72769	1	72769.0	72769.0	72769	72769	0.0	cudaDeviceSynchronize
0.0	7034	4	1758.5	1303.5	1233	3194	959.3	cuDeviceGetName
0.0	5756	4	1439.0	132.5	103	5388	2632.8	cuDeviceTotalMem_v2
0.0	3149	1	3149.0	3149.0	3149	3149	0.0	cuCtxSynchronize
0.0	1509	1	1509.0	1509.0	1509	1509	0.0	cudaGetLastError
0.0	1439	1	1439.0	1439.0	1439	1439	0.0	cuModuleGetLoadingMode
0.0	458	4	114.5	80.5	79	218	69.0	cuDeviceGet
0.0	437	4	109.3	83.5	81	189	53.2	cuDeviceGetUuid
0.0	355	1	355.0	355.0	355	355	0.0	cuDeviceGetCount

[5/7] Executing 'cuda_gpu_kern_sum' stats report

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
100.0	70848	1	70848.0	70848.0	70848	70848	0.0	matrixProduct

[6/7] Executing 'cuda_gpu_mem_time_sum' stats report

Time (%)	Total Time (ns)	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation
83.0	244222	2	122111.0	122111.0	120991	123231	1583.9	[CUDA memcpy HtoD]
17.0	50176	1	50176.0	50176.0	50176	50176	0.0	[CUDA memcpy DtoH]

[7/7] Executing 'cuda_gpu_mem_size_sum' stats report

Total (MB)	Count	Avg (MB)	Med (MB)	Min (MB)	Max (MB)	StdDev (MB)	Operation
4.194	2	2.097	2.097	2.097	2.097	0.000	[CUDA memcpy HtoD]
2.097	1	2.097	2.097	2.097	2.097	0.000	[CUDA memcpy DtoH]

Reading the nsys nvprof report (what matters)

- **CUDA API summary → host-side calls**
 - `cudaMalloc` often dominates API time (tens of ms per call)
- **Kernel summary → device execution**
 - `matrixProduct` ran once; e.g., ~70–100 μ s for $N=512$ (H100)
- **GPU mem time summary**
 - H→D copies (two inputs) vs D→H copy (one output)
- **GPU mem size summary**
 - Confirms bytes moved (e.g., ~2 MB per 512×512 matrix doubles)
- **Key takeaways**
 - Data motion frequently dominates; reduce transfers & allocations.
 - Keep allocations outside hot paths; reuse device buffers.
 - Profile repeatedly if you need robust statistics.

Task 5.7

■ Task 5.7 – Profiling Matrix Multiplication on the GPU

- Goal

Time your matrix multiplication for $N=1024$ using `nsys nvprof` on MN5.

- Do

- Submit via SLURM; save the profiler output.
- Identify where time is spent:
 - kernel vs `cudaMemcpy` (HtoD/DtoH) vs `cudaMalloc`
- Briefly note optimization ideas (e.g., reuse buffers, fuse copies, larger N).

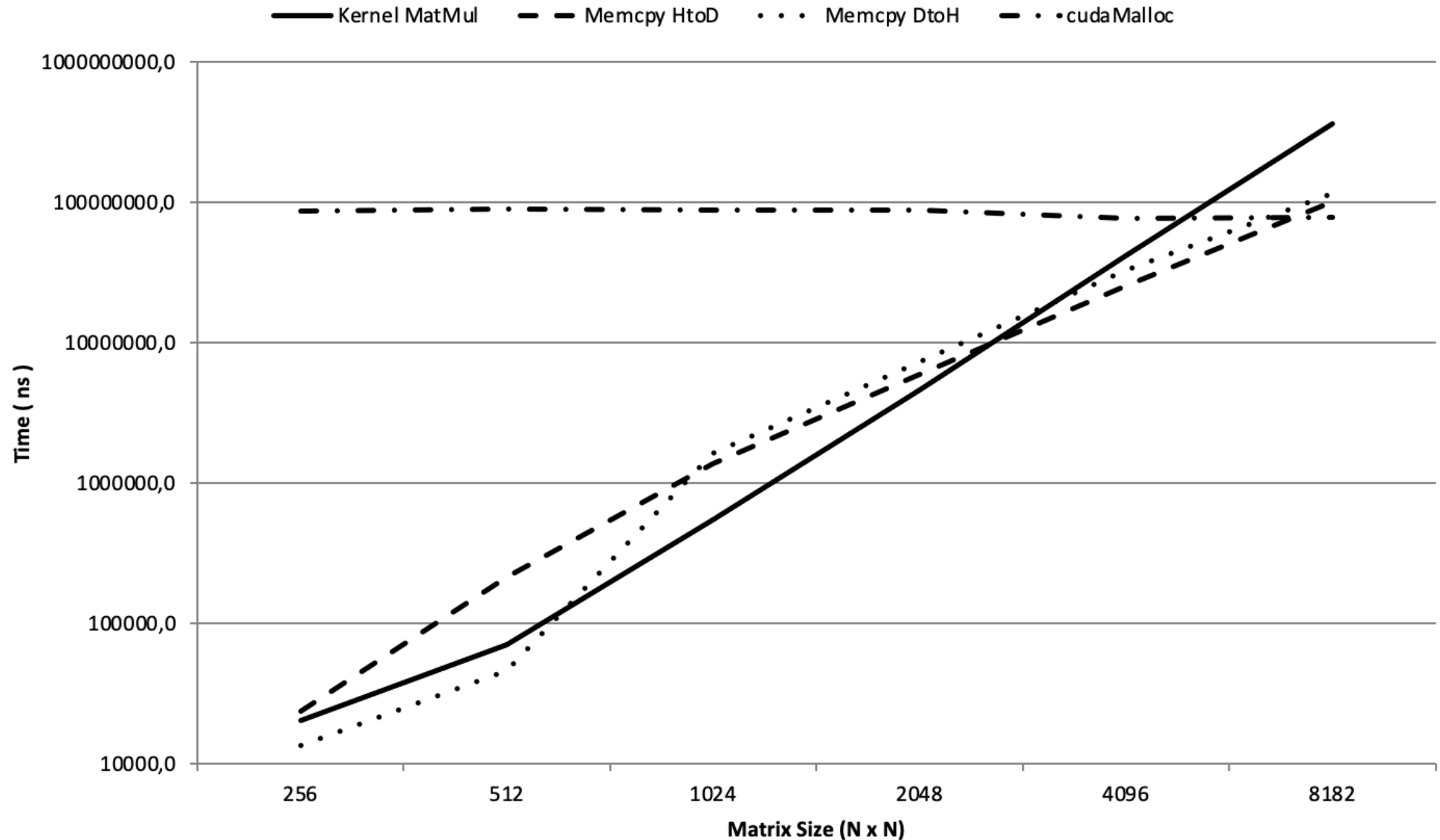
Performance & Scalability with nsys nvprof

■ Experiment setup

- Problem sizes $N = 64, 128, 256, 512, 1024, 2048$ (and beyond)
- Build per size (or pass N as an argument)
- Launch config: block 16×16 , grid $((N+15)/16, (N+15)/16)$
- Profile each run:

```
nsys nvprof ./matrixMulN
```

Execution time vs matrix size (log scale)



Interpreting the scaling plot

- Kernel (solid) grows $\sim O(N^3)$ → compute-bound for large N
- cudaMemcpy HtoD/DtoH (dashed/dotted) grow with bytes moved $\sim O(N^2)$
- cudaMalloc (dash-dot) roughly constant per call (size-independent)
- **Crossover point (memory-bound → compute-bound)**
 - Small N: transfers dominate → GPU underutilized
 - Large N: kernel dominates → arithmetic intensity wins
- **Implications**
 - For small problems, batching & minimizing transfers matter most.
 - For large problems, optimize kernel math & memory access patterns.

Task 5.8

■ Task 5.8 – Compute-Bound vs Memory-Bound

- Goal

Replicate the scaling study and decide where your run becomes compute-bound.

- Do

- Implement/build your CUDA MatMul; validate on a small N.
- Run `nsys nvprof` for $N = 256 \dots 8192$ (as resources allow).
- Collect per run:
 - Kernel time (`matrixProduct`)
 - `cudaMemcpy` times (HtoD & DtoH)
 - `cudaMalloc` time
- Plot on log-scale Y; analyze trends & the crossover.
- Conclude: when does your workload shift from memory-bound to compute-bound, and why?

Pd: GPU programming and CUDA

- **Tasks included: task 5.1 – task 5.8**
- **Deliverable:**
 - Upload a single PDF (per group) to the intranet racó@FIB containing one slide per task. Each slide should report results or briefly explain how the task was completed.
 - In class (evaluation day), one group (chosen at random) will give a “*elevator pitch*”-style— clear, concise, and straight to the point.

Labs presentation

- Good practical experience for students!
and ... a way to stimulate homework accomplishment
- 1 group **will be randomly chosen**
 - We'll sum 4 numbers from randomly chosen students and use the '%' function with the total number of students to find the winner in the list.

```
>>> nums_to_add = ...+...+...+...  
>>> winner= nums_to_add % num_students +1  
>>> print (winner)
```