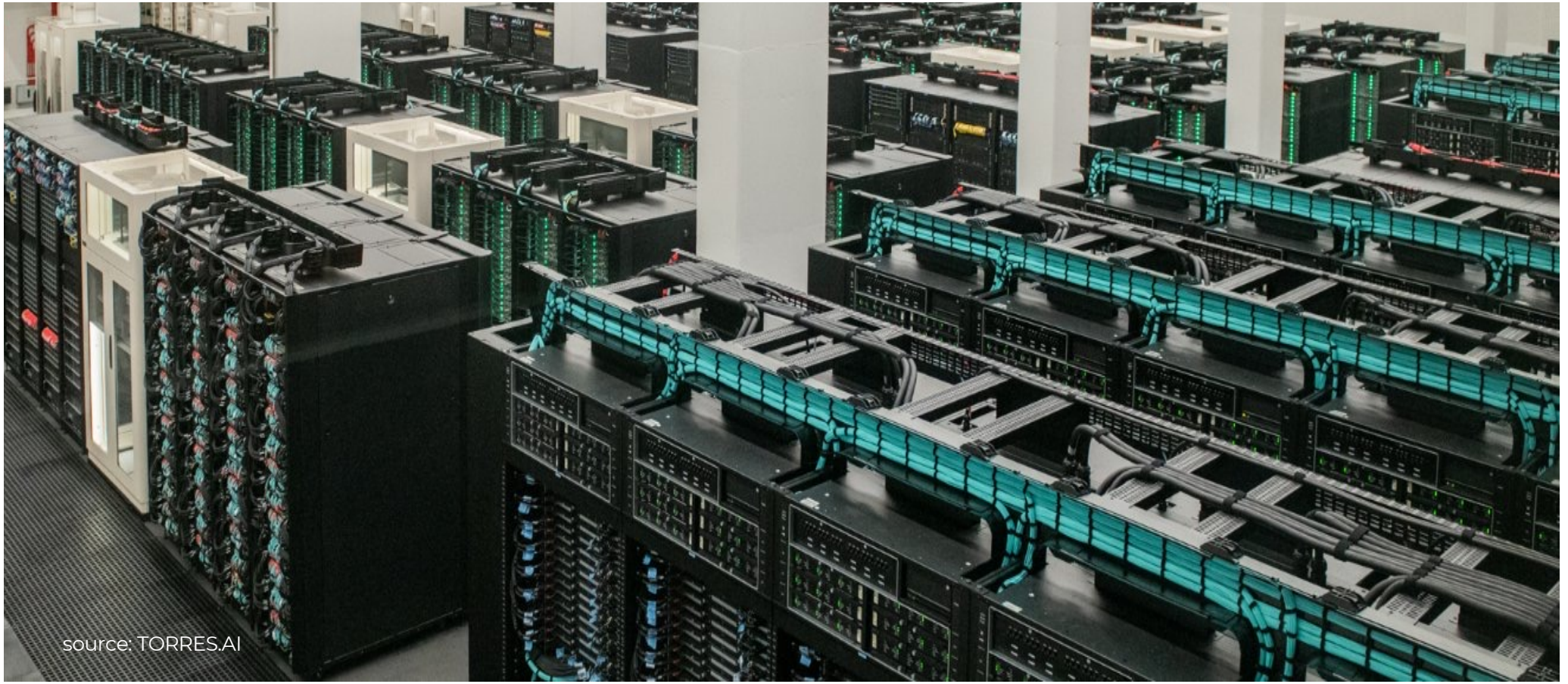
Teoria 5 Parallel Computing with CUDA

Computació d'Altes Prestacions

Josep Ll. Berral – Jordi Torres · Grau IA – FIB



Parallel Computing with CUDA

1. Basic elements of GPU programming
2. CUDA programming models basics
3. Thread organization
4. Memory organization
5. Launching a CUDA kernel
6. Case study: Matrix Multiplication
7. Timing the kernel

→Ex

What is an (AI) accelerator?

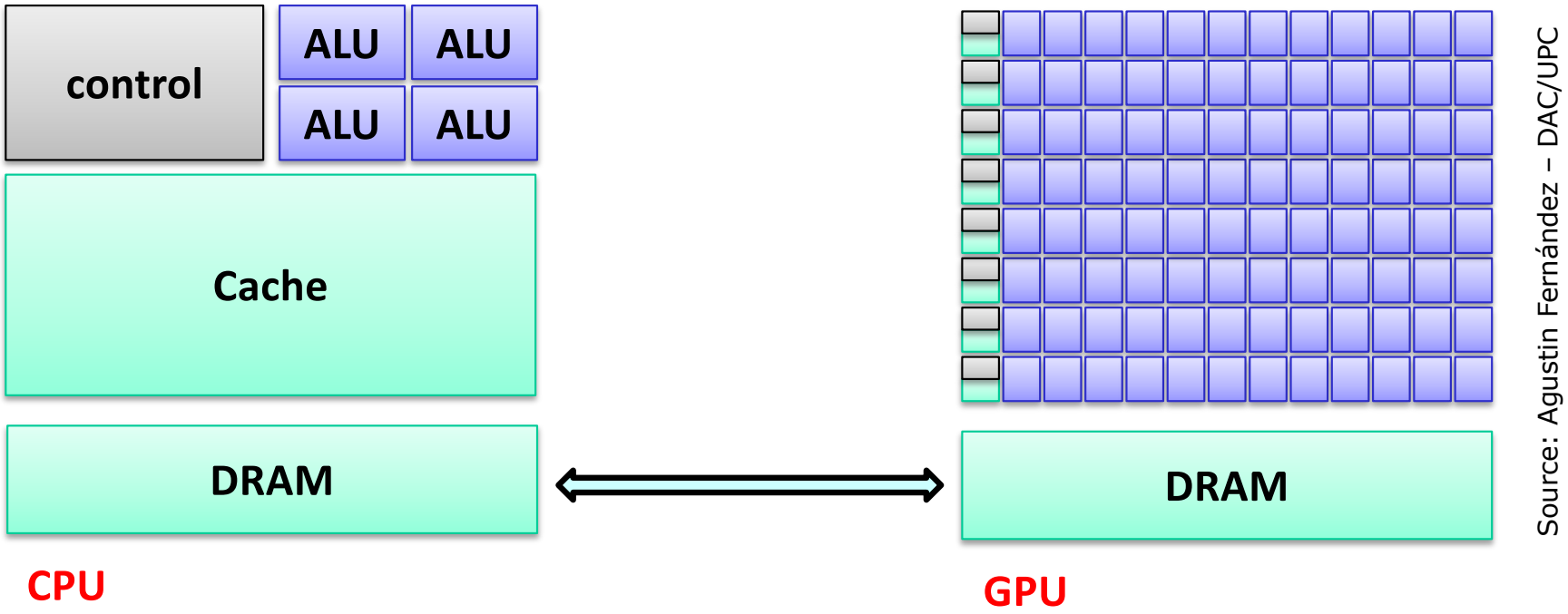
- An (AI) accelerator is a dedicated processor designed to accelerate machine learning tasks (and particularly its subset, deep learning).
  - They are primarily composed of a large number of linear algebra operations.

    ( These algebra operations can be easily parallelized)

- AI accelerator's goals:
  - Accelerate computations and improve performance
  - Reduce the cost of deploying models

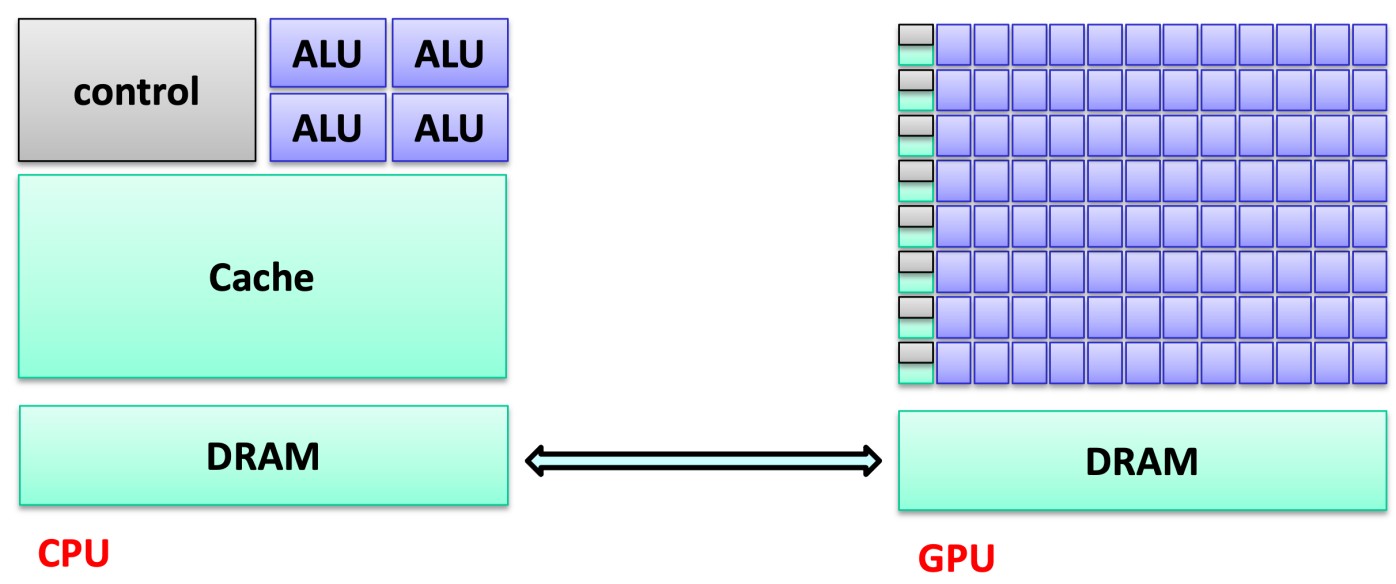
Accelerators?

- These are usually attached to the computer node of the system using industry-standard interfaces, such as **PCI Express** (or **NVLink)**



Source: Agustin Fernández – DAC/UPC

Accelerators?



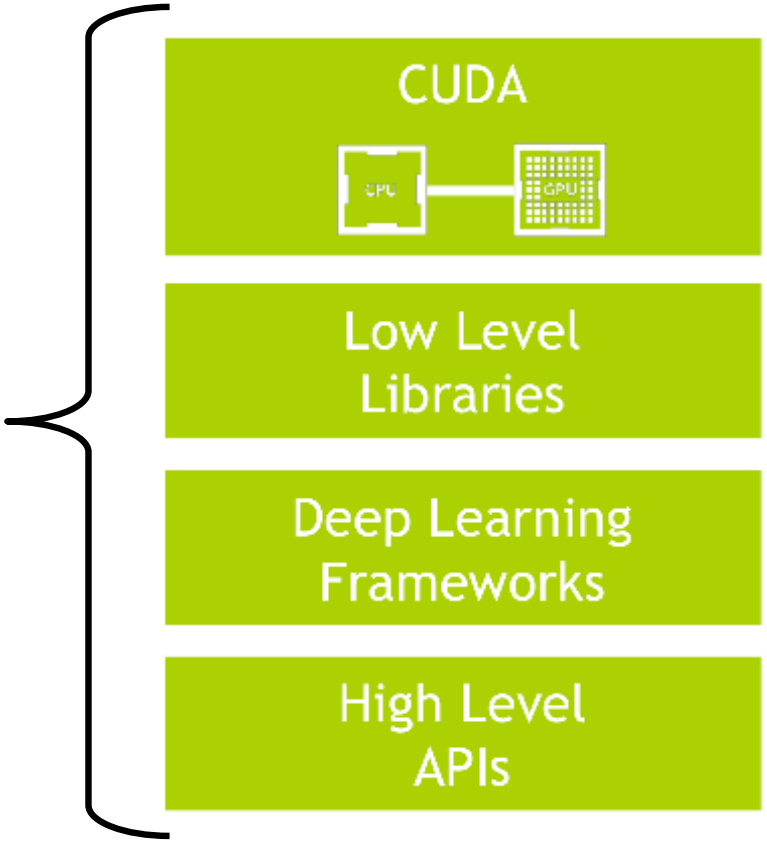
- **This separation of the two functional domains by a standard bus introduces several significant limitations to the control of the transfer and access of the data that the accelerators consume or produce.**

- In general, the main processor is the master control entity responsible for transferring required data from the accelerator memory to the main memory on the computer node.

Accelerators

- In Supercomputing, accelerators are typically employed to increase the computational throughput in terms of FLOPS, **although at the cost of programmability.**

different alternatives



Accelerator vs CPUs

- Instead of spending silicon on large caches, complex control logic for out-of-order execution, and other mechanisms that aim to speed up how quickly a single control flow can progress individually, accelerator architectures spend silicon on building simple. Still, many cores with large register files keep the execution contexts of many independent control flows resident on the chip to switch between rapidly.
- The accelerator also relies on a SIMD approach to further increase the fraction of silicon used for computation rather than control logic.
- Each control logic is associated with multiple arithmetic logic units (ALUs), allowing for the parallel execution of multiple control flows as long as they execute the exact instructions on different data.

Accelerators, yes …. but …

Allow the incorporation of accelerated hardware in practically any supercomputer ….. if ….

has sufficient incoming power to supply energy to the accelerator

And suitable cooling support to eliminate the heat generated by the accelerator.

Origin of Accelerators?

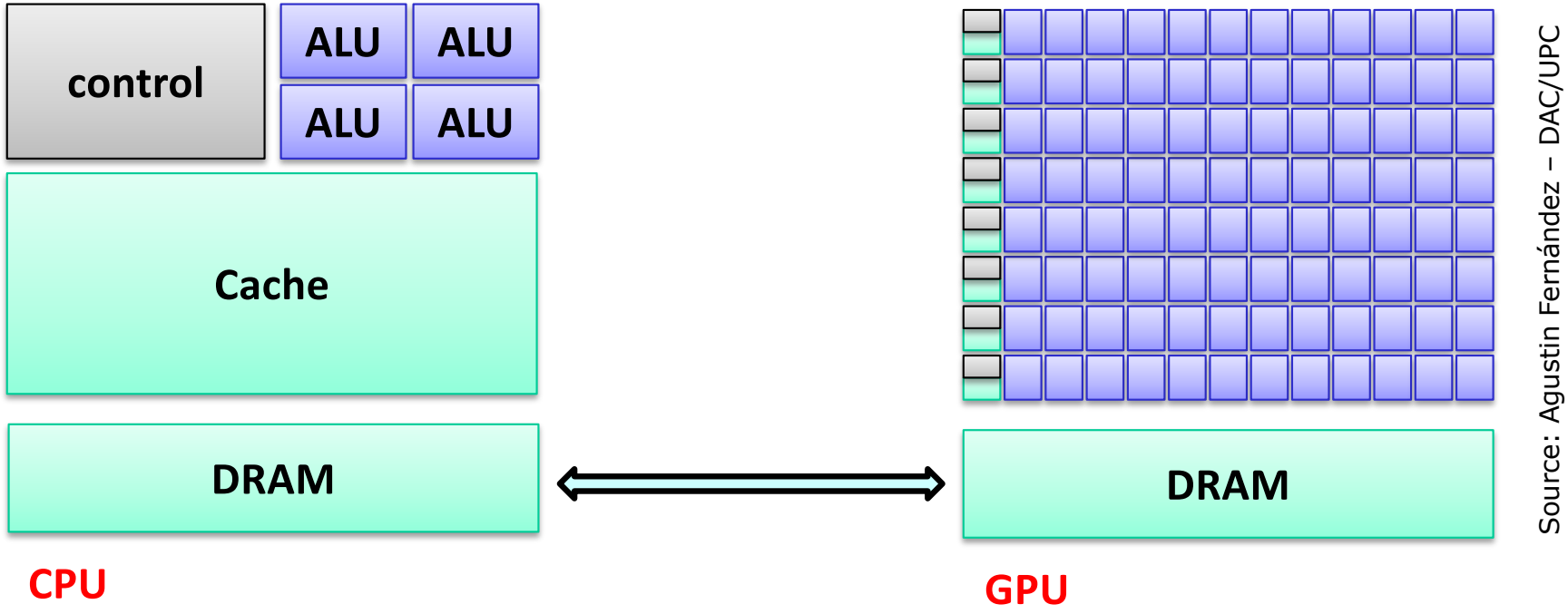
- The first GPU (Graphic Processing Unit) was introduced by NVIDIA in **1999**.
- Originally, GPUs were specialized computing devices that performed multiple processing functions associated with image generation (**rendering of three-dimensional scenes**) due to the volume of computations required and their real-time nature.

Origin of Accelerators?

- Three-dimensional **scene rendering** is a highly parallel task and can be subdivided into many concurrent computations operating on their section of the image.
  - To accomplish this, GPUs included many FPUs and execution logic supporting many **processing threads** plus **substantial memory bandwidth** to feed the data at sustained high rates.
  - The FPUs were optimized for single-precision arithmetic, as this level of accuracy is sufficient for most graphics operations. However, in the late 2000s, they started to include double-precision hardware to extend the range of GPU applications to areas beyond computer graphics, such **as Supercomputing**.

Host vs device

- GPUs must operate in conjunction with a CPU-based host through a PCI-Express bus.



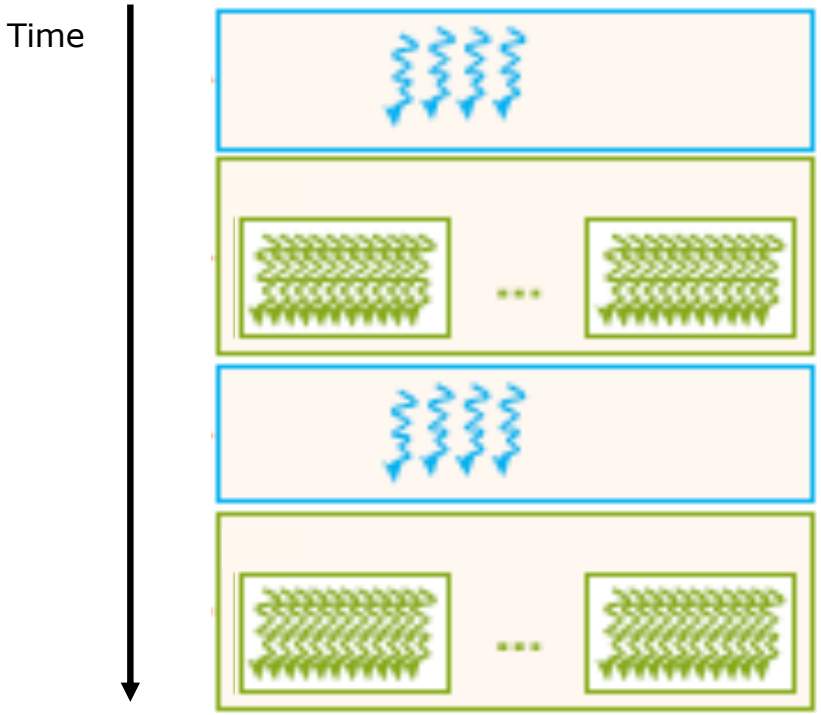
Source: Agustin Fernández – DAC/UPC

- In GPU computing terms,
  - the CPU is called the **host**
  - the GPU is called the **device**

Heterogeneous Computing

- The increased application design complexity currently limits the effective use of such systems.

- A heterogeneous application consists of two parts:



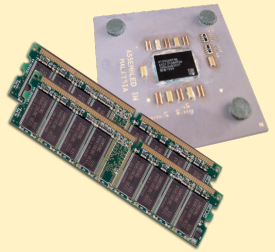
Heterogeneous Computing

- A heterogeneous application consists of two parts:
  - **Host code** that runs on CPUs

    The CPU code is responsible for managing the device's environment, code, and data before loading compute-intensive tasks on it.

  - **Device code** that runs on GPUs

    GPUs are used to accelerate the execution of the intensive data parallelism portion.



Host

Device

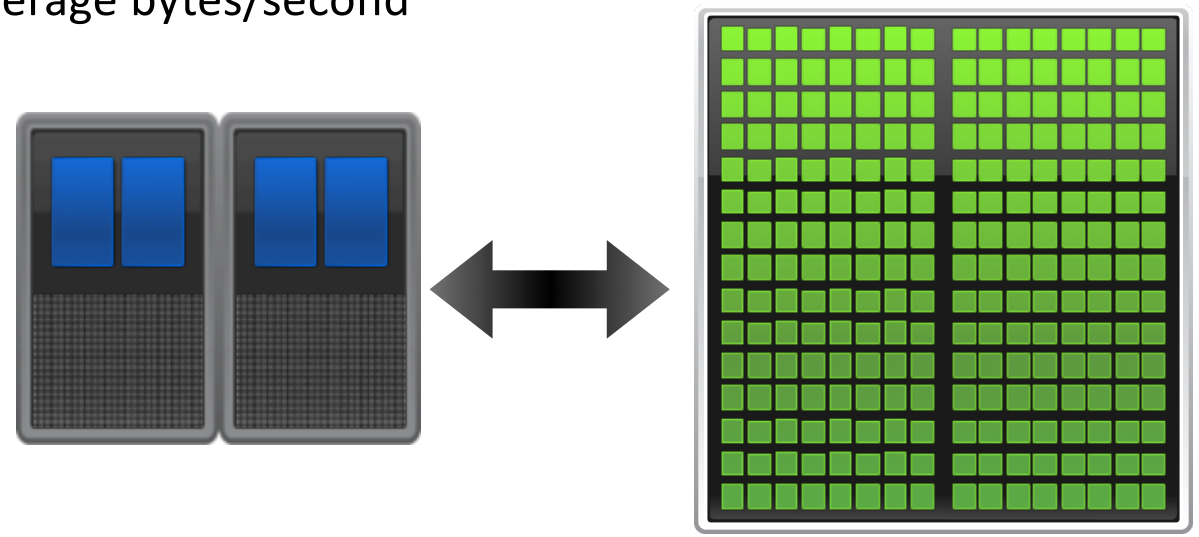
CUDA programming model

- These accelerators exploit many-core architectures that can exploit coarse and fine-grained parallelism.
- However, the traditional problem with using GPUs and other accelerators has been the ease of programming them.
- To this end, NVIDIA designed the **Compute Unified Device Architecture (CUDA)**

to support joint CPU + GPU execution of an application

Heterogeneous Parallel Computing

- Key Performance Issues
- latency: time for first byte
- throughput: average bytes/second



Latency Optimized CPU

Fast Serial Processing

Throughput Optimized GPU

Scalable Parallel Processing

Source: Forrest Iandola & Bryan Catanzaro from NVIDIA

Latency vs Throughput

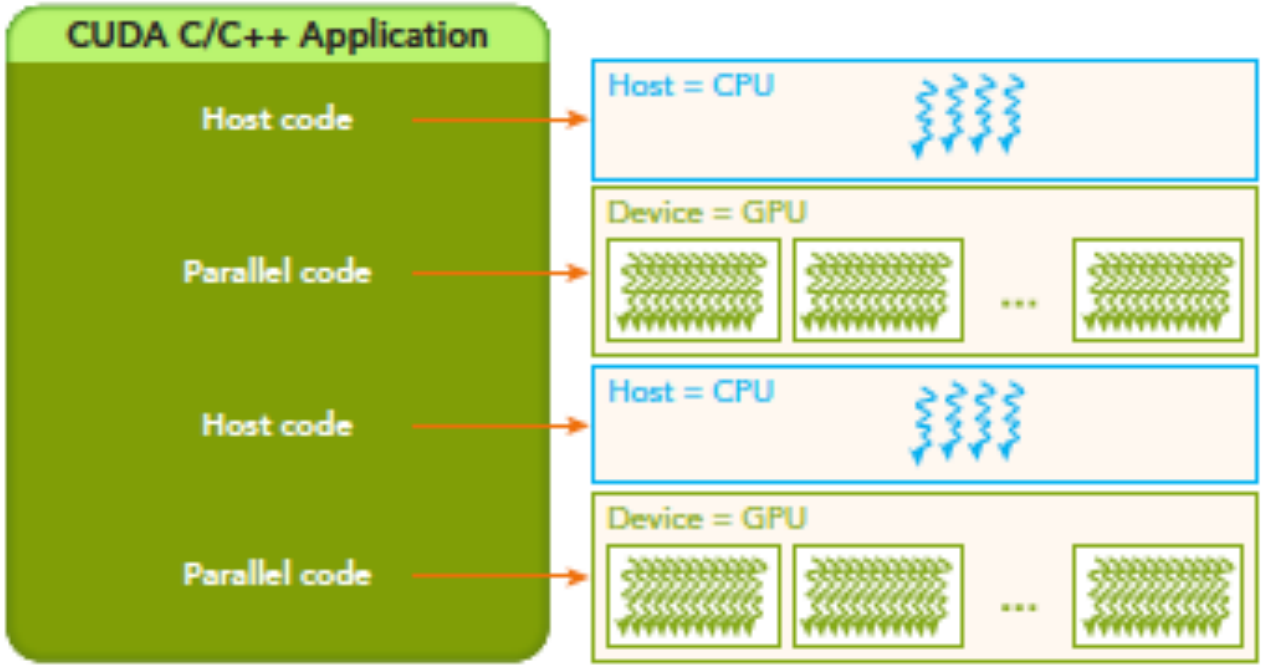
- **Different goals produce different designs.**
  - Throughput cores: assume the workload is highly parallel
  - Latency cores: assume the workload is primarily sequential

- **Latency goal: minimize latency experienced by one thread**
  - There are lots of big on-chip caches
  - extremely sophisticated control, branch prediction

- **Throughput goal: maximize throughput of all threads**
  - lots of big ALUs
  - multithreading can hide latency … so skip the big caches
  - more straightforward control, cost amortized over ALUs via SIMD

Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
- GPUs for parallel parts where throughput wins



Winning Applications Use Both CPU and GPU

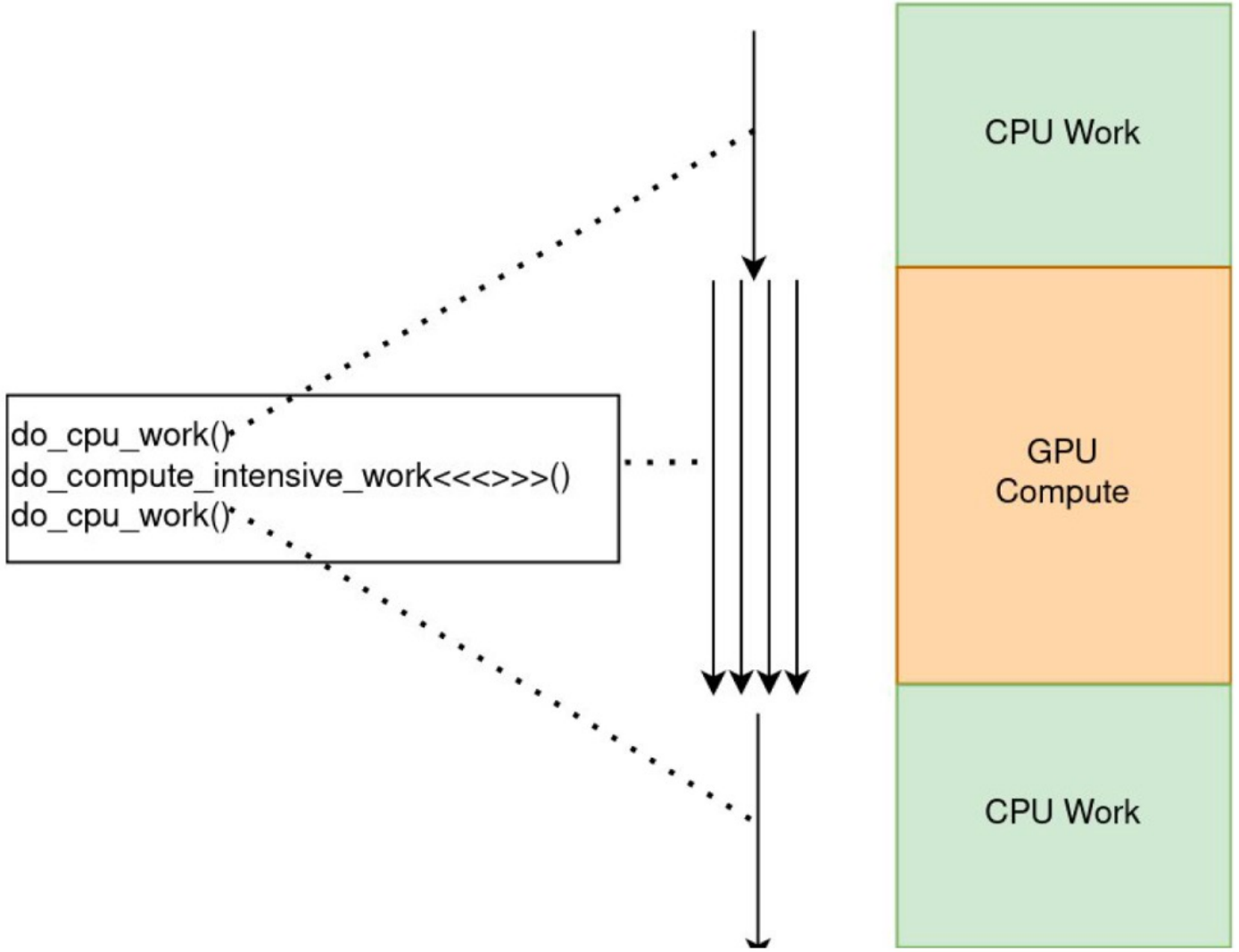


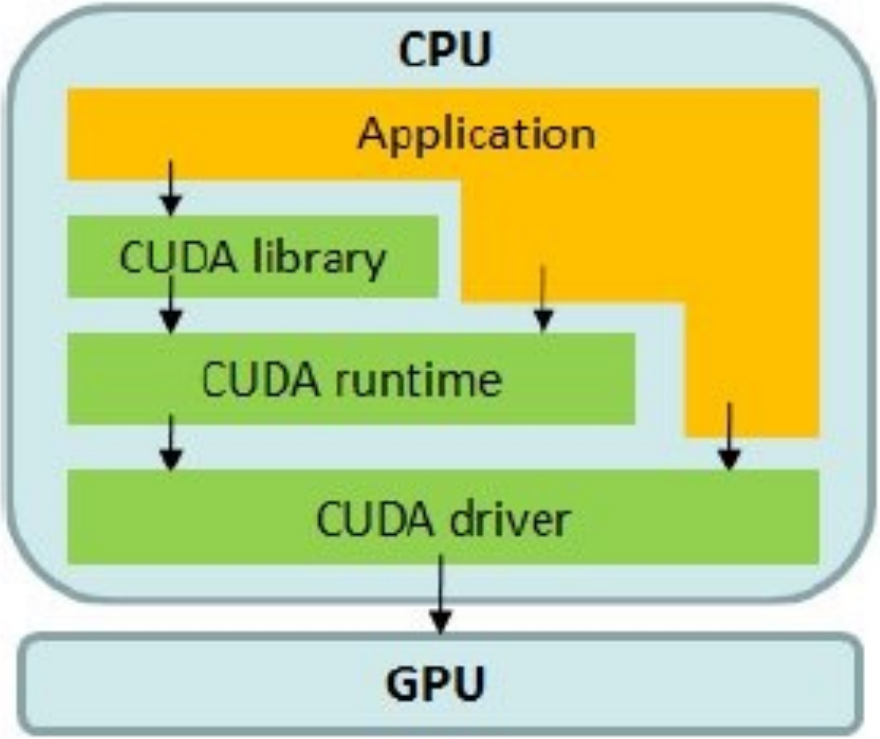
image source: BSC

CUDA as a parallel computing platform

- CUDA is a general-purpose parallel computing **platform** and **programming model**:
  - Application programming interfaces
  - CUDA-accelerated libraries
  - compiler directives
  - extensions to industry-standard programming languages (including C, C++, Fortran, and Python)
  - …

Application programming interface

- CUDA provides two API levels for managing the GPU device and organizing threads:
  - CUDA Driver API (low-level API)
  - CUDA Runtime API (high-level API)



What is CUDA? (Compute Unified Device Architecture)

- **Developed by NVIDIA**: A parallel computing platform and programming model.
- **Purpose**: Allows developers to use NVIDIA GPUs for general-purpose processing (GPGPU).
- **Key Features:**
  - **Parallel Computing**: Executes many calculations simultaneously.
  - **Speed and Efficiency**: Significantly accelerates applications by offloading compute-intensive tasks to the GPU.
  - **Extensive Libraries**: Provides optimized libraries for various applications (e.g., deep learning, scientific computing). *ex: cuBLAS, cuSparse, CUDA C++ Core Libraries (cccL)*.
- **Components:**
  - **CUDA Toolkit**: Includes compilers, libraries, and debugging tools.
  - **CUDA Runtime API**: Simplifies the development of GPU-accelerated applications.
  - **CUDA Driver API**: Provides lower-level control over the GPU.
- **Applications:**
  - **Machine Learning and AI**: Accelerates training and inference processes.
  - **Scientific Research**: Enhances simulations and data analysis.
  - **Gaming and Graphics**: Improves rendering and visual effects.

GPU-Accelerated Libraries

- **Provide highly optimized algorithms and functions**:
  - With NVIDIA's libraries, you get highly efficient implementations of regularly extended and optimized algorithms. Whether you are building a new application or trying to speed up an existing application, NVIDIA's libraries provide the easiest way to get started with GPUs

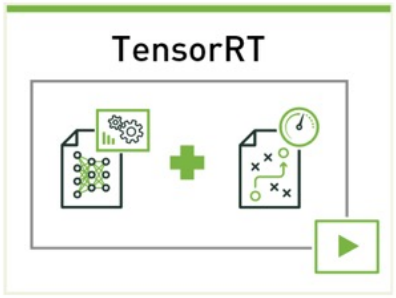
Visit: https://developer.nvidia.com/gpu-accelerated-libraries

GPU-Accelerated Libraries

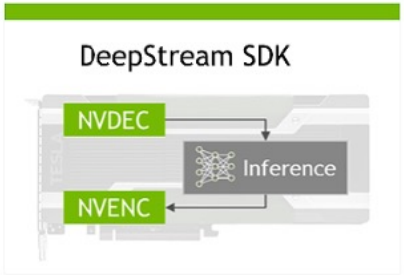
- **Deep Learning libraries**

cuDNN

GPU-accelerated library of primitives for deep neural networks



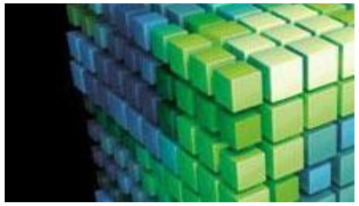
GPU-accelerated neural network inference library for building deep learning applications



Advanced GPU-accelerated video inference library

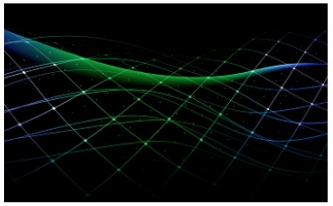
GPU-Accelerated Libraries

- **Linear Algebra and Math Libraries**



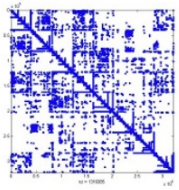
cuBLAS

GPU-accelerated standard BLAS library



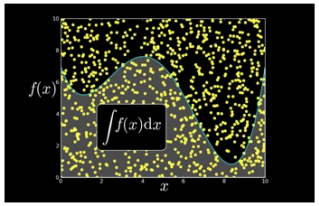
CUDA Math Library

GPU-accelerated standard mathematical function library



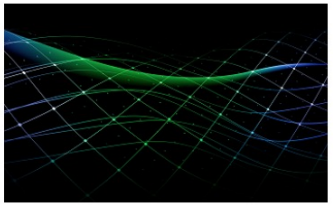
cuSPARSE

GPU-accelerated BLAS for sparse matrices



cuRAND

GPU-accelerated random number generation (RNG)



cuSOLVER

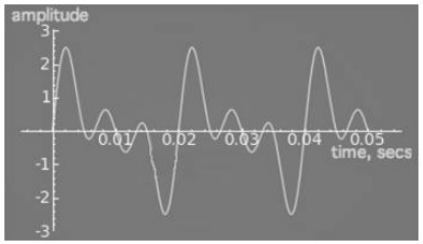
Dense and sparse direct solvers for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications

AmgX

GPU accelerated linear solvers for simulations and implicit unstructured methods

GPU-Accelerated Libraries

- **Signal, Image and Video Libraries**



cuFFT

GPU-accelerated library for Fast Fourier Transforms

NVIDIA Performance Primitives

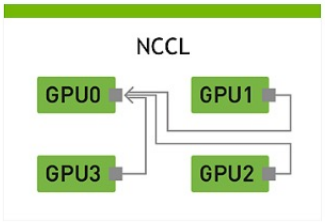
GPU-accelerated library for image and signal processing

NVIDIA Codec SDK

High-performance APIs and tools for hardware accelerated video encode and decode

GPU-Accelerated Libraries

- **Parallel Algorithm Libraries**



NCCL

Collective Communications Library for scaling apps across multiple GPUs and nodes

nvGRAPH

GPU-accelerated library for graph analytics

Thrust

GPU-accelerated library of parallel algorithms and data structures

GPU-Accelerated Libraries

- **Partner Libraries**

GPU-accelerated open-source library for computer vision, image processing and machine learning, now supporting real-time operation

Open-source multi-media framework with a library of plugins for audio and video processing

GPU-accelerated open source library for matrix, signal, and image processing

GPU-accelerated linear algebra routines for heterogeneous architectures, by Magma

GPU-accelerated open-source Fortran library with functions for math, signal and image processing, statistics, by RogueWave

Library for graph-processing designed specifically for the GPU

GPU-accelerated functions for sparse direct solvers, included in SuiteSparse linear algebra package authored by Prof.

GPU-accelerated linear algebra library by EM Photonics

GPU-accelerated linear algebra (LA) routines for the R platform for statistical computing supporting heterogeneous

GPU-accelerated computational geometry engine for advanced GIS, EDA, computer vision, and motion planning, by Fixstars

GPU-accelerated library for sparse iterative methods by Paralution

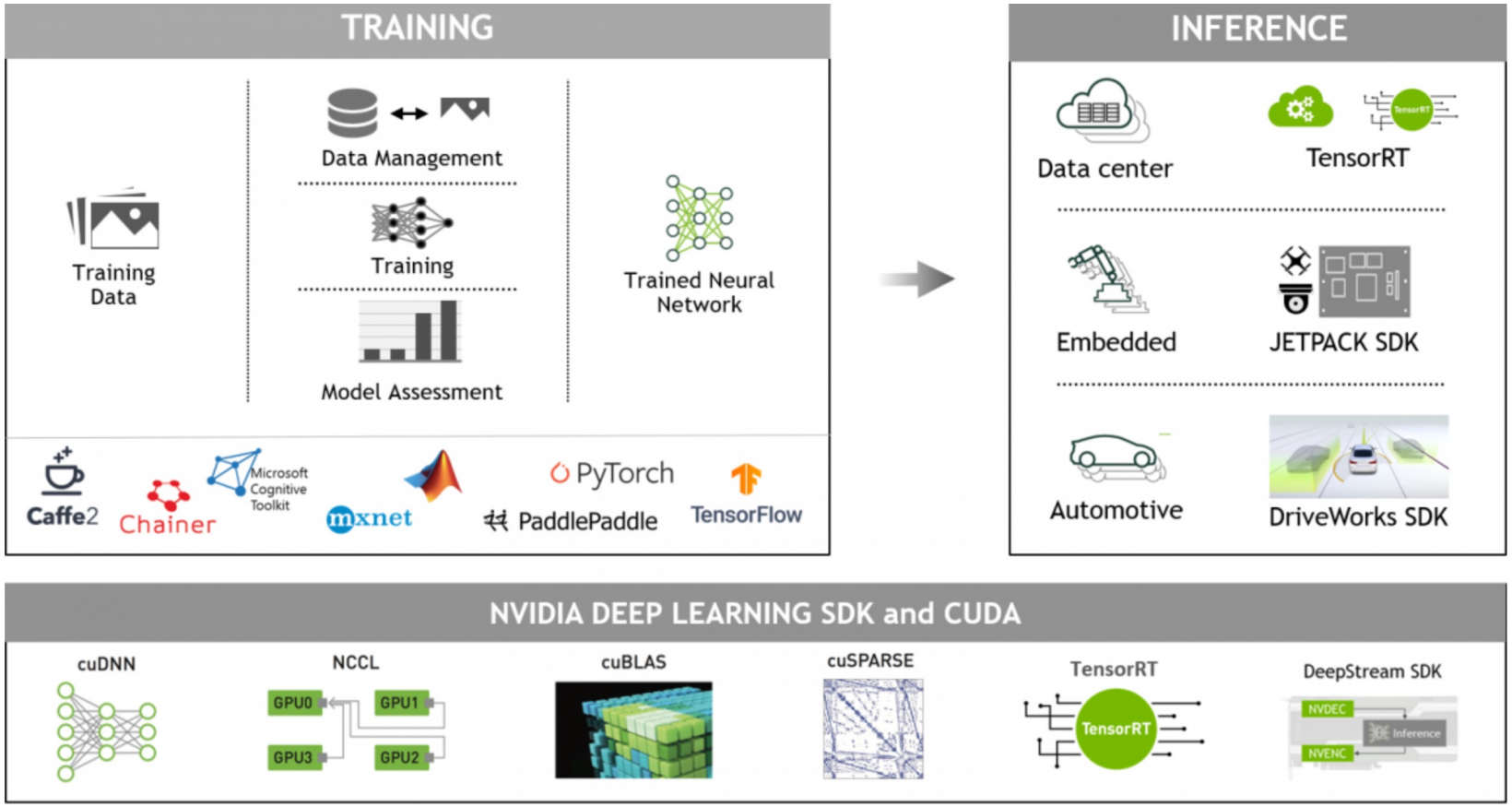
Real-time visual simulation of oceans, water bodies in games, simulation, and training applications, by Triton

Deep Learning Frameworks

- Deep learning frameworks offer building blocks for designing, training, and validating deep neural networks through a high-level programming interface.
- **Visit https://developer.nvidia.com/deep-learning-software**

NVIDIA Deep Learning SDK

- Provides powerful tools and libraries for designing and deploying GPU-accelerated deep learning applications.
- It includes libraries for deep learning primitives, inference, video analytics, linear algebra, sparse matrices, and multi-GPU communications.



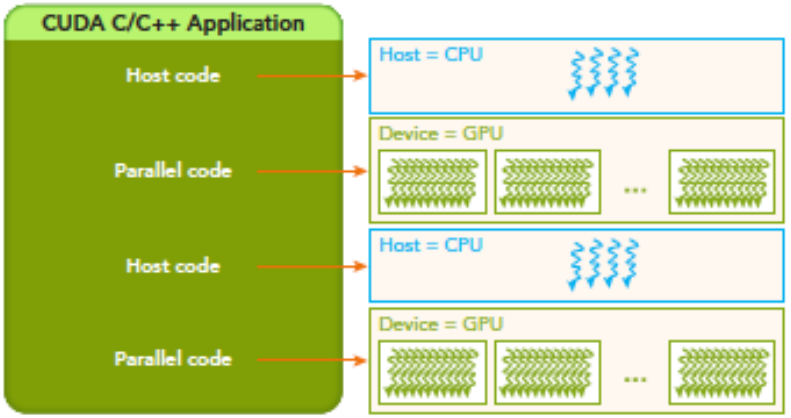
Parallel Computing with CUDA

1. Basic elements of GPU programming
2. CUDA programming models basics
3. Thread organization
4. Memory organization
5. Launching a CUDA kernel
6. Case study: Matrix Multiplication
7. Timing the kernel

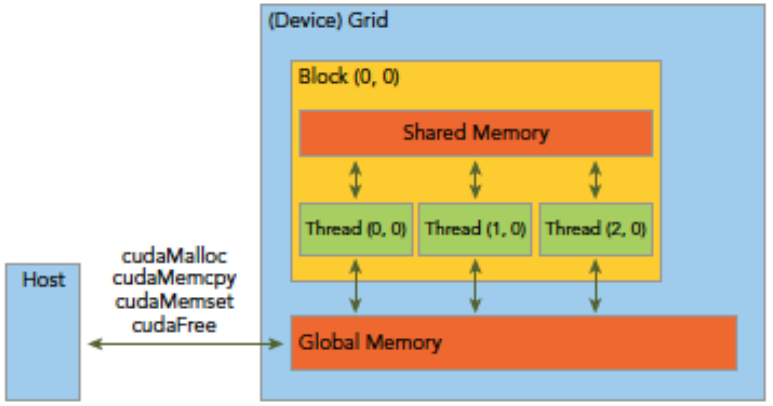
→Ex

CUDA programming model basics

- Two main features:
  - A way to **organize threads** on the GPU through a hierarchy structure



  - A way to **access memory** on the GPU through a hierarchy structure



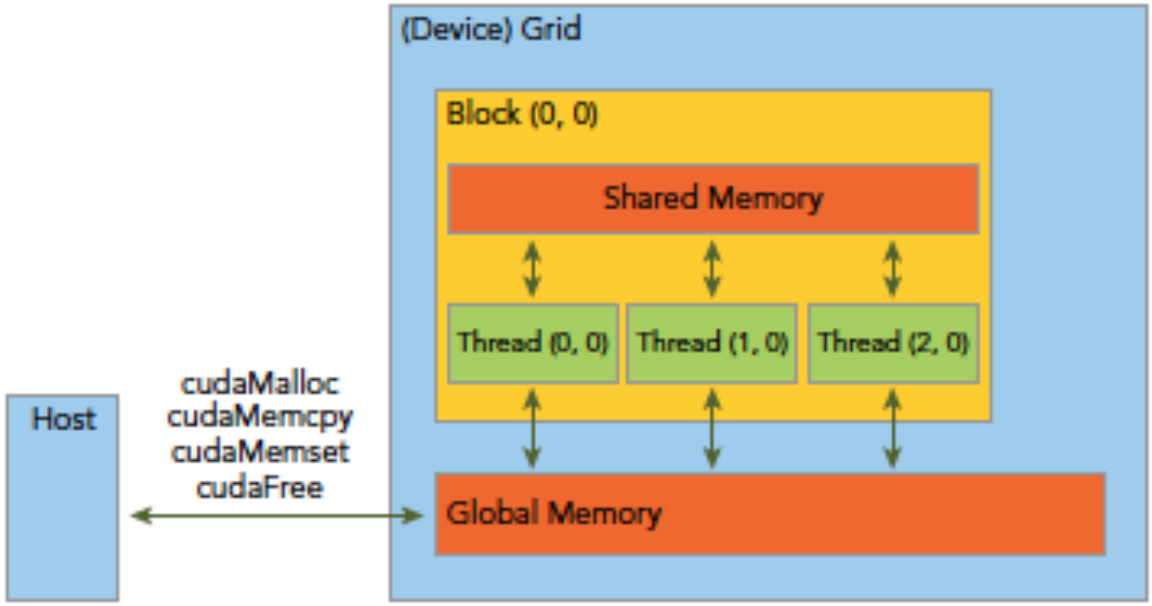
CUDA programming model basics

- Henceforth, we will make the following distinction in the code:
  - **Host**: the CPU and its memory (host memory referred to with `h_`)
  - **Device**: the GPU and its memory (device memory referred to with `d_`)

- A key component of the CUDA programming model: **kernel**
  - The code that runs on the GPU device

Simplified memory structure

- The CUDA programming model **exposes an abstraction** of memory hierarchy from the GPU architecture.
- A simplified GPU memory structure, containing two major ingredients:
  - global memory and
  - shared memory



Parallel Computing with CUDA

1. Basic elements of GPU programming
2. CUDA programming models basics
3. Thread organization
4. Memory organization
5. Launching a CUDA kernel
6. Case study: Matrix Multiplication
7. Timing the kernel

→Ex

Organizing Threads

- How does CUDA create the hundreds or thousands of parallel threads required?

- And how does CUDA initialize the individual threads so that each one does a different part of the work?

  →

  CUDA creates **all the threads as a group**, running the same function with a set of parameters that apply to all.

Organizing Threads

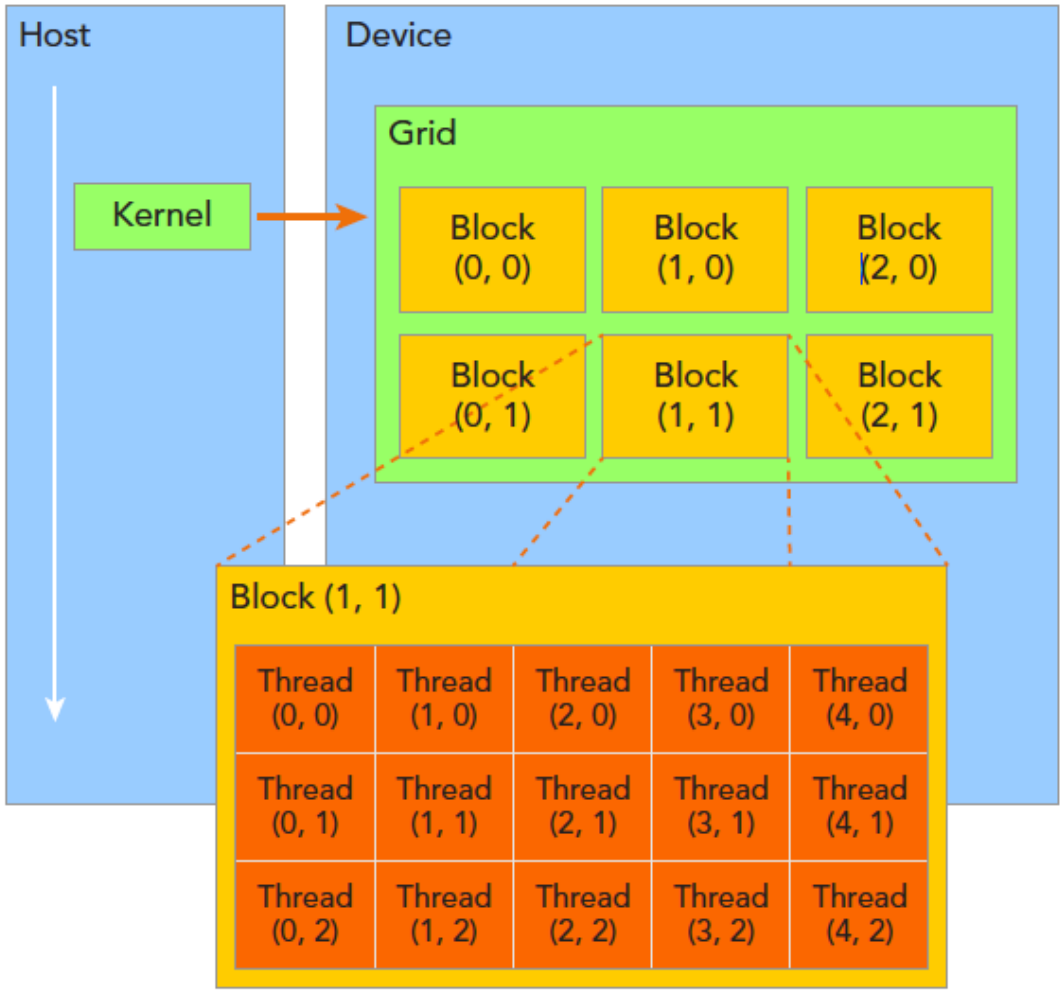
→

Then, CUDA creates the threads in a multidimensional structure, where each thread is **aware of its position** in the overall structure thanks to a set of intrinsic structure variables.

With this information, a thread can map its position to the subset of the assigned data.

Organizing Threads

- CUDA exposes a **two-level thread** hierarchy decomposed into **blocks** of threads and **grids** of blocks:



Organizing Threads: **Grid**

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

- A Grid can a 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 ≤ 1024)

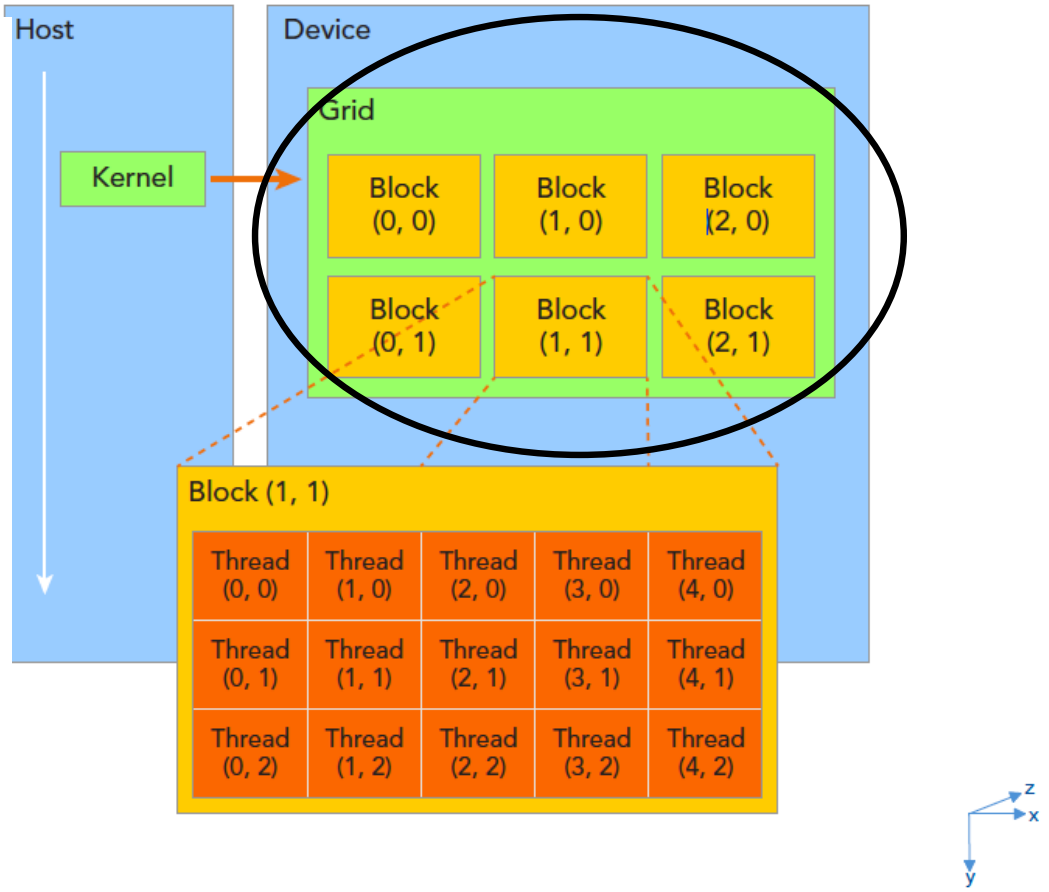
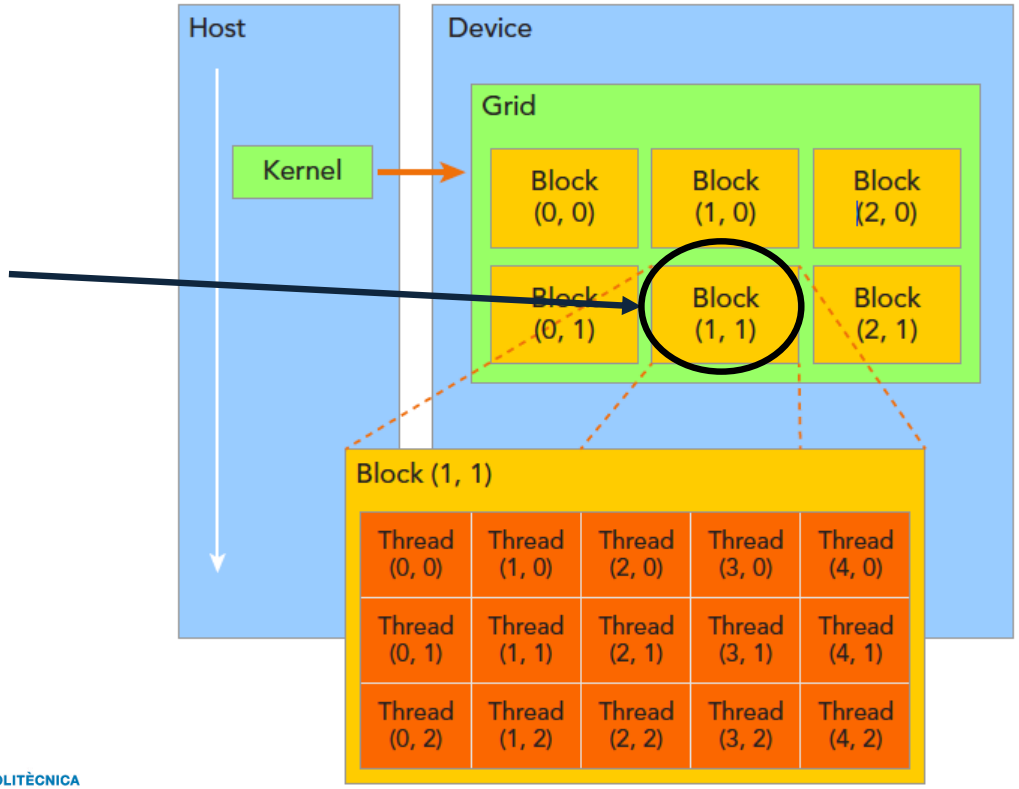


image from Nvida Cuda programming Guide

Organizing Threads: **Grid**

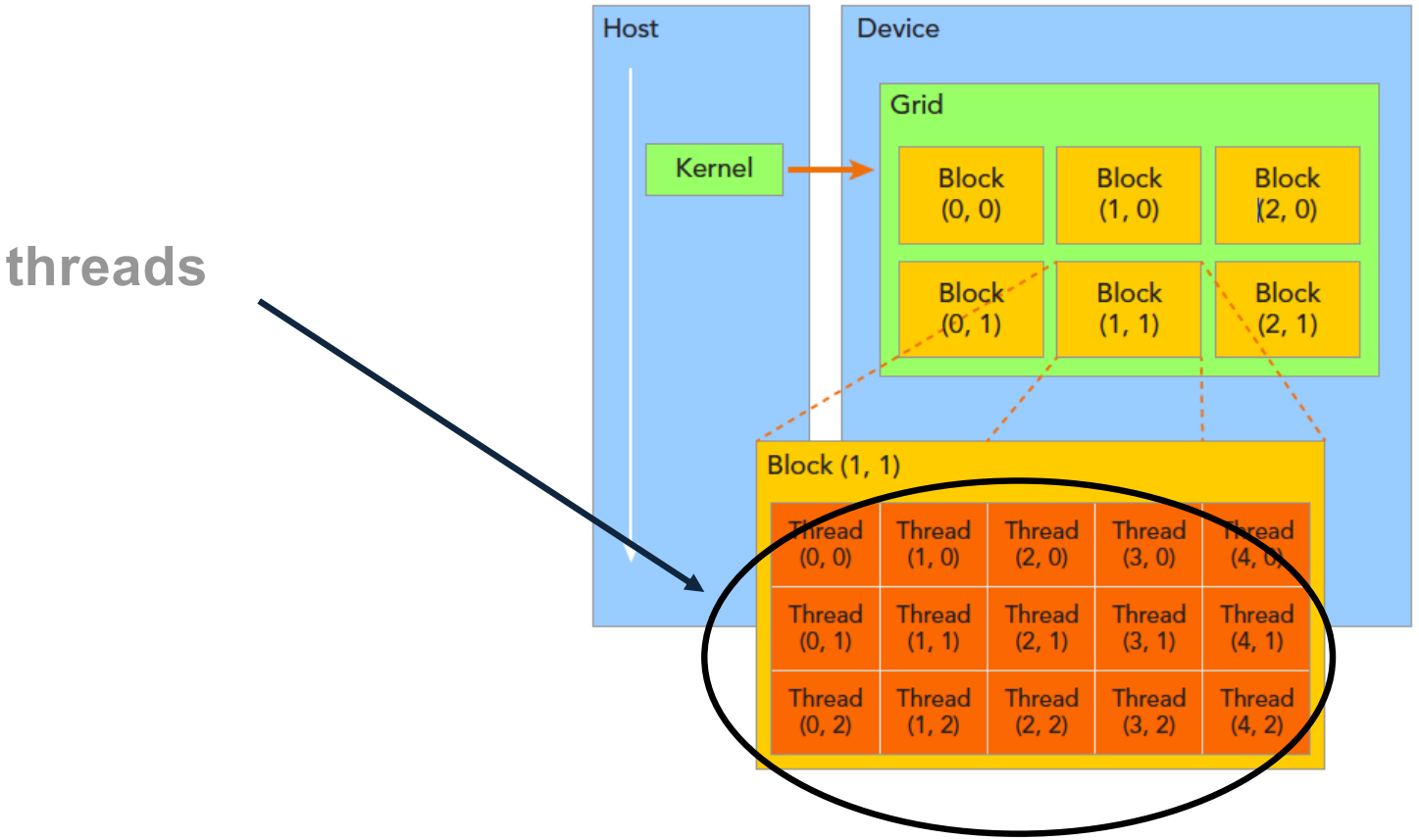
- A grid is made up of many thread blocks:
  - All threads in a grid share the same global memory space.

thread block



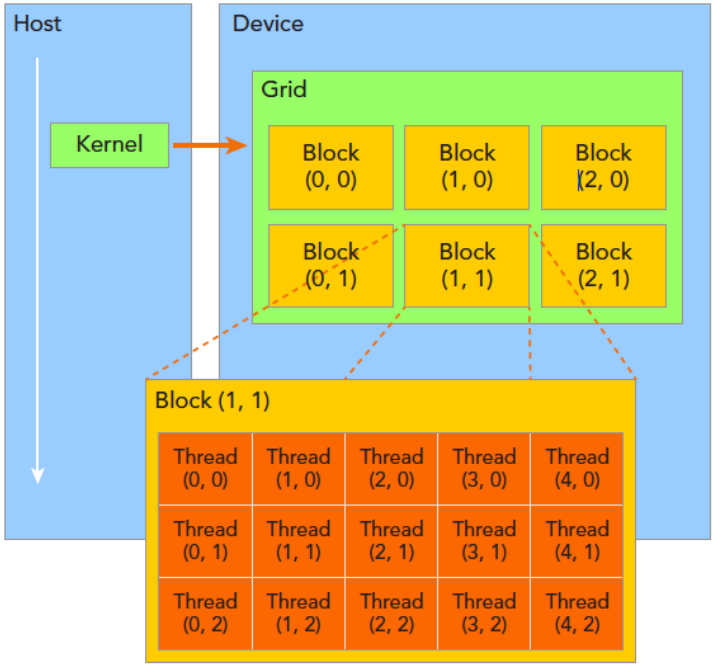
Organizing Threads: **Grid**

- 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: **Grid**

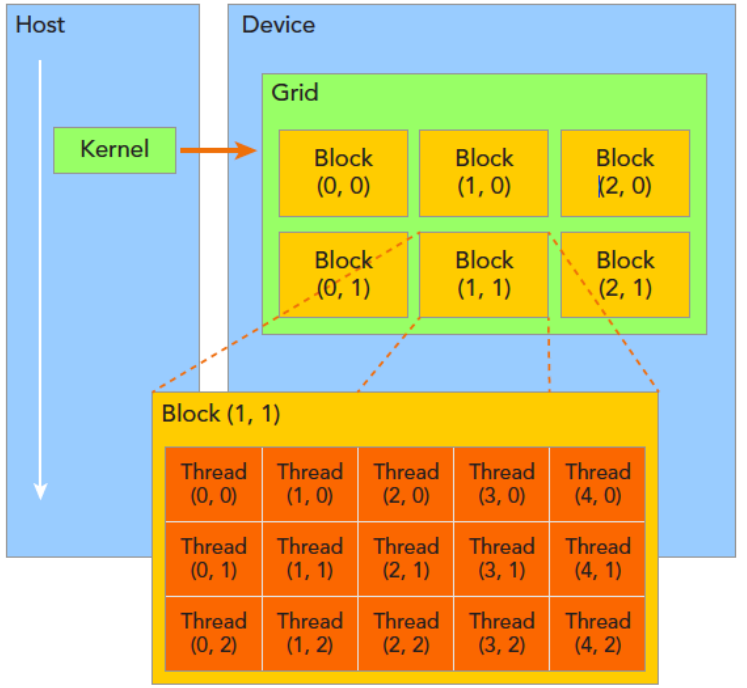
- Threads rely on two unique coordinates to distinguish themselves from each other: (*)
  - **`blockIdx`** : block index within a grid
  - **`threadIdx`** : thread index within a block



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

Organizing Threads: **Grid**

- 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

CUDA Execution Model Summary

- **Thread: Sequential execution unit**
  - All threads execute the same sequential program
  - Threads execute in parallel
- **Thread Block: a group of threads**
  - Executes on a single Streaming Multiprocessor (SM)
  - Threads within a block can cooperate
- **Grid: a collection of thread blocks**
  - Thread blocks of a grid execute across multiple SMs
  - Thread blocks do not synchronize with each other
  - Communication between blocks is expensive

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

Launching a CUDA Kernel SPOILER

- C function call syntax:

```
function_name (argument list);
```

- A CUDA kernel call is a direct extension to the C function syntax that adds a kernel's execution configuration inside triple-angle-brackets:

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

  - `grid:` number of blocks to launch.
  - `block:` the number of threads within each block.

  - By specifying the `grid` and `block` dimensions, we configure:
    - The total number of threads for a kernel
    - The layout of the threads you want to employ for a kernel

Organizing Threads: Example

- First, define the data size to be used in the program:

```
int nElem = 6;
```

- Next, define the block size and calculate the grid size based on the block and data size:

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

(*) In this example, a 1D block containing 3 threads and a 1D grid with the number of blocks defined based on the data and block sizes are defined. (grid size is rounded up to the multiple of the block size).

Access Grid/Block Variables: From the host and device side

- In order to checks the grid and block dimensions:

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

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

```
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);
```

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

Hands-on: Organizing Threads Example

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

/*
 * Display the dimensionality of a thread block and grid from the host and
 * device.
 */

__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);
}
```

Hands-on: Organizing Threads Example

```
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);


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


    // Ensure that the kernel has completed execution
    cudaDeviceSynchronize();

    // reset device before you leave
    cudaDeviceReset();
```

source: https://github.com/jorditorresBCN/Marenostrum5/blob/main/checkIndex.cu

Hands-on: Organizing Threads Example

```
% ssh nct01002@alogin1.bsc.es

[nct01002@alogin1 ~]$ vi checkIndex.cu
[nct01002@alogin1 ~]$ module load nvidia-hpc-sdk/23.11-cuda11.8
[nct01002@alogin1 ~]$ nvcc checkIndex.cu -o checkIndex
[nct01002@alogin1 ~]$ ./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)
```

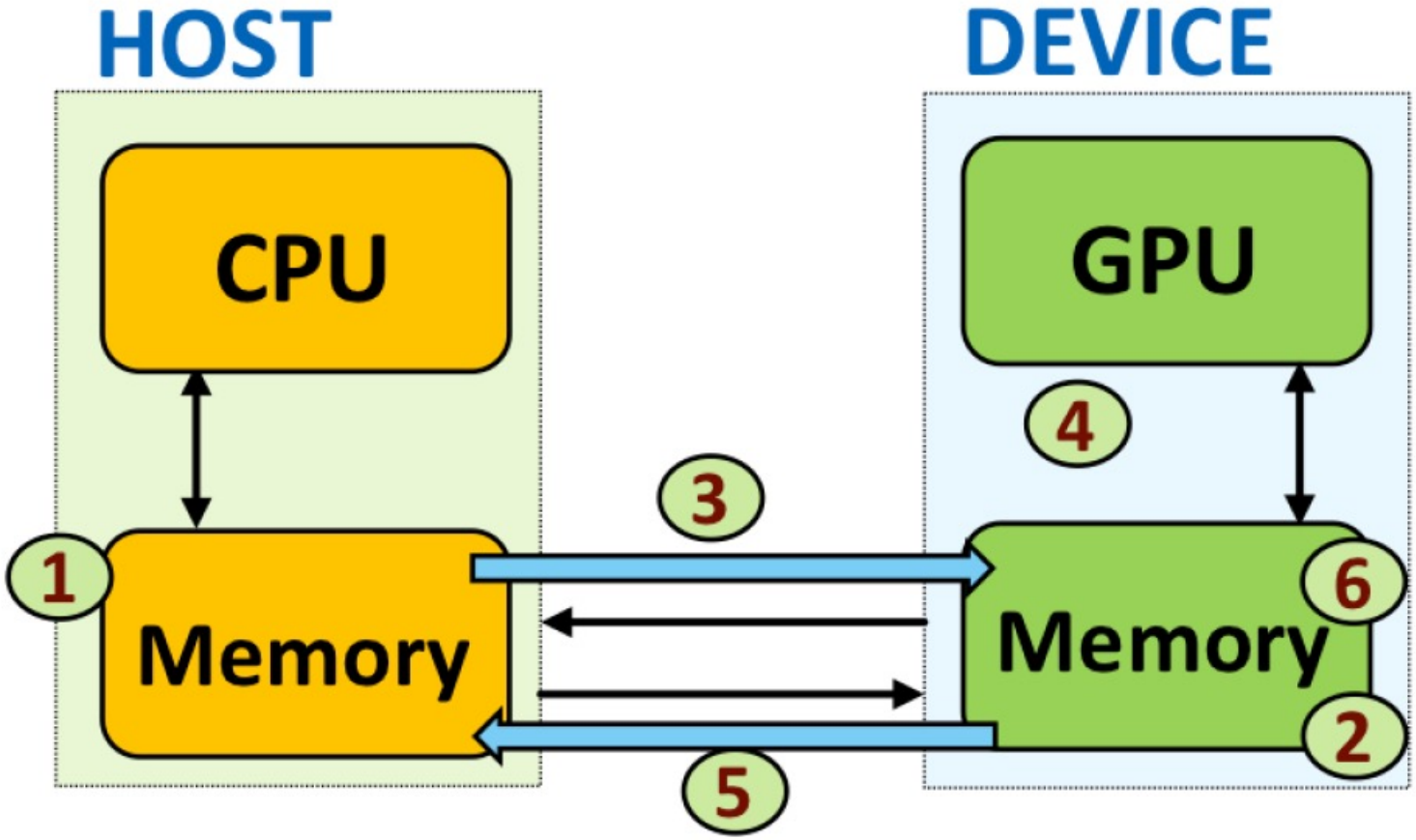
Parallel Computing with CUDA

1. Basic elements of GPU programming
2. CUDA programming models basics
3. Thread organization
4. Memory organization
5. Launching a CUDA kernel
6. Case study: Matrix Multiplication
7. Timing the kernel

→Ex

CUDA memory management flow summary

- The CUDA programming model assumes a system composed of a host and a device, each with its own separate memory.



Memory management

- The pattern of a CUDA processing flow :
  - Copy data from CPU memory to GPU memory.
  - Invoke kernels to operate on the data stored in GPU memory.
  - Copy data back from GPU memory to CPU memory.

- Host and Device Memory Functions:

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

GPU memory allocation

```
cudaError_t cudaMalloc(void** devPtr, size_t size)
```

This function allocates a linear range of device memory with the specified size in bytes. The allocated memory is returned through `devPtr`

Transfer data between the host and device

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

- This function copies the specified bytes from the source memory area, pointed to by `src`, to the destination memory area, pointed to by `dst`.
- The direction is specified by kind (can take one of the following types):
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

(*) This function exhibits synchronous behavior because the host application blocks until cudaMemcpy returns and the transfer is complete.

Example: data movement between host and device

- **A simple kernel to add two integers**

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- **`__global__` is a CUDA C/C++ keyword meaning**
  - `add()` will execute on the device
  - `add()` will be called from the host

Addition on the Device

- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory

- We need to allocate memory on the GPU (device)

Addition on the Device: `main()`

```
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;
```

Addition on the Device: `main()`

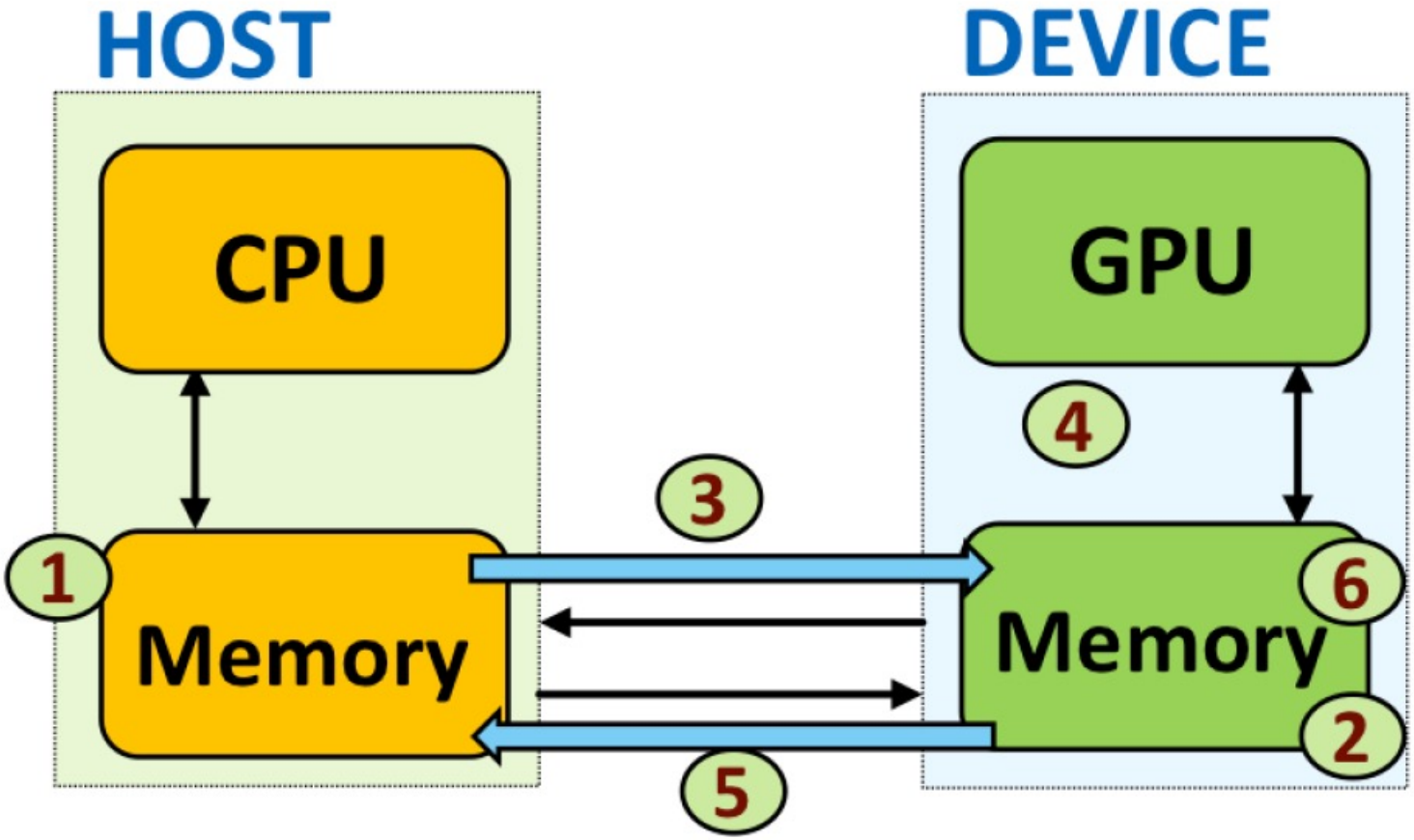
```
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // 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);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

CUDA memory management flow



Addition on the Device: `main()`

```
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;
```

1

2

Addition on the Device: `main()`

3

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

4

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

5

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

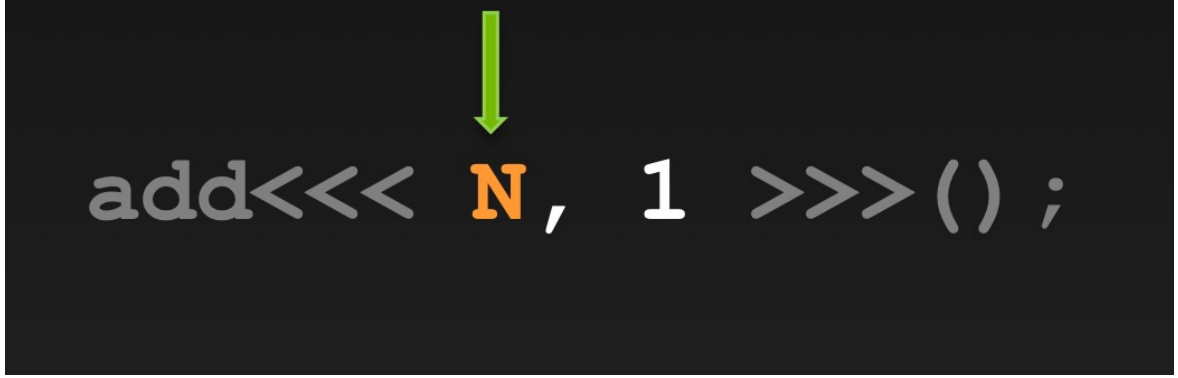
6

```
// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Moving to Parallel

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

```
add<<< 1, 1 >>>();
```



```
add<<< N, 1 >>>();
```

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

Parallel Computing with CUDA

1. Basic elements of GPU programming
2. CUDA programming models basics
3. Thread organization
4. Memory organization
5. Launching a CUDA kernel
6. Case study: Matrix Multiplication
7. Timing the kernel

→Ex

Launching a CUDA Kernel Basics

- C function call syntax:

```
function_name (argument list);
```

- A CUDA kernel call is a direct extension to the C function syntax that adds a kernel's execution configuration inside triple-angle-brackets:

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

  - `grid:` number of blocks to launch.
  - `block:` the number of threads within each block.

  - By specifying the `grid` and `block` dimensions, we configure:
    - The total number of threads for a kernel
    - The layout of the threads you want to employ for a kernel

Launching a CUDA Kernel Basics

- The threads within the same block can easily communicate with each other, and threads that belong to different blocks cannot cooperate.

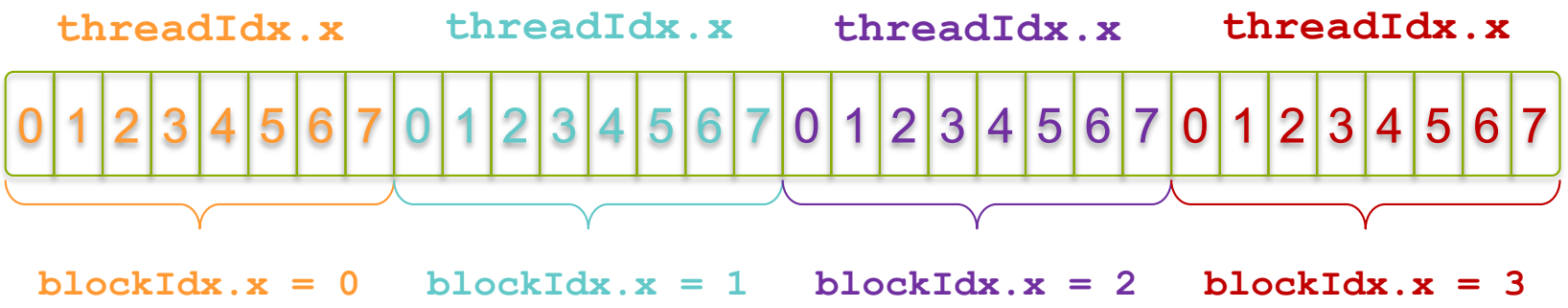
- For a given problem, you can use a different grid and block layout to organize your threads.

- For example, suppose we have 32 data elements for a calculation and we group 8 elements into each block, and launch 4 blocks:

```
kernel_name <<<4, 8>>>(argument list);
```

Indexing Arrays: Example

- Layout of threads in the above configuration:

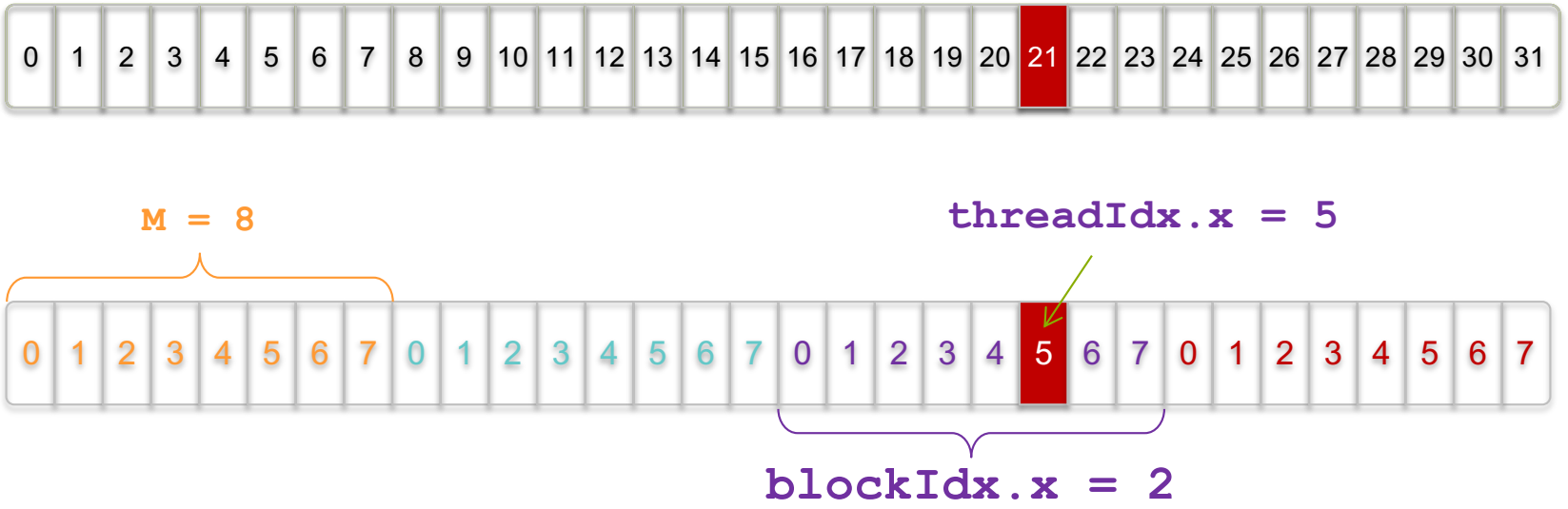


- Indexing Arrays with Blocks and Threads:
  - With M threads/block a unique index for each thread is given by:

```
index = threadIdx.x + blockIdx.x * M;
```

Indexing Arrays: Example

- Which thread will operate on the red element?




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

Slide source: NVIDIA 2013

Vector Addition with Blocks and Threads

Vector Addition example:

- 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 index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- Vector Addition `main()`?

Source: NVIDIA

Addition with Blocks and Threads: `main()`

```
#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

Addition with Blocks and Threads: `main()`

```
    // 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;
}
```

Handling Arbitrary Vector Sizes

Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

```
__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];
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

Last but not least, other topics

- **CUDA programming of GPU platforms covers a wide variety of topics not covered in this "Getting Started with CUDA"**
- **Last but not least, some important topics focused on improving our codes using the CUDA programming model:**
  - **Related to the execution model**: Warp Execution, Branch Divergence, Unrolling Loops, Dynamic Parallelism, Streams and Concurrency, etc.
  - **Related to memory model:** Advanced memory management, Access Patterns, Shared Memory and Constant Memory, etc.
  - **Related to performance:** Tuning Instruction-level primitives, GPU-accelerated CUDA Libraries, OpenACC, etc.
  - **Related to Multi-GPU Programming:** subdividing computation, Peer-to-peer communication, scaling applications, etc.

Parallel Computing with CUDA

1. Basic elements of GPU programming
2. CUDA programming models basics
3. Thread organization
4. Memory organization
5. Launching a CUDA kernel
6. Case study: Matrix Multiplication
7. Timing the kernel

→Ex

Handling Errors?

- Since many CUDA calls are asynchronous, it may be difficult to identify which routine caused an error.

- How to handle errors?

  → Defining an error-handling **macro** and function to wrap all CUDA API calls simplifies the error checking process.

Handling Errors

- We propose using the following approach:

```
#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));
      }
}
```

- For example, you can use it on the following code:

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

If the memory copy or a previous asynchronous operation caused an error, the macro reports the error code, prints a human readable message, and then stops the program.

Case Study: Matrix Multiplication

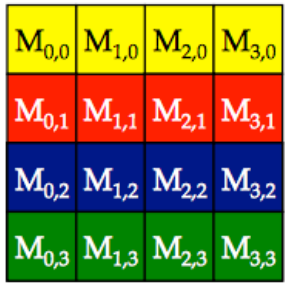
- Full global thread ID in x and y dimensions can be computed by:

```
x = blockIdx.x * blockDim.x + threadIdx.x;
y = blockIdx.y * blockDim.y + threadIdx.y;
```

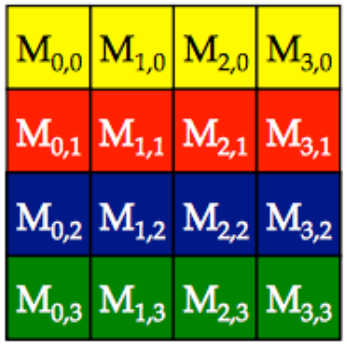
- Generally, memory is allocated dynamically on the device (GPU) , and we can not use two-dimensional indices to access matrices. **Matrices are flattened into linear memory.**

Case Study: Matrix Multiplication

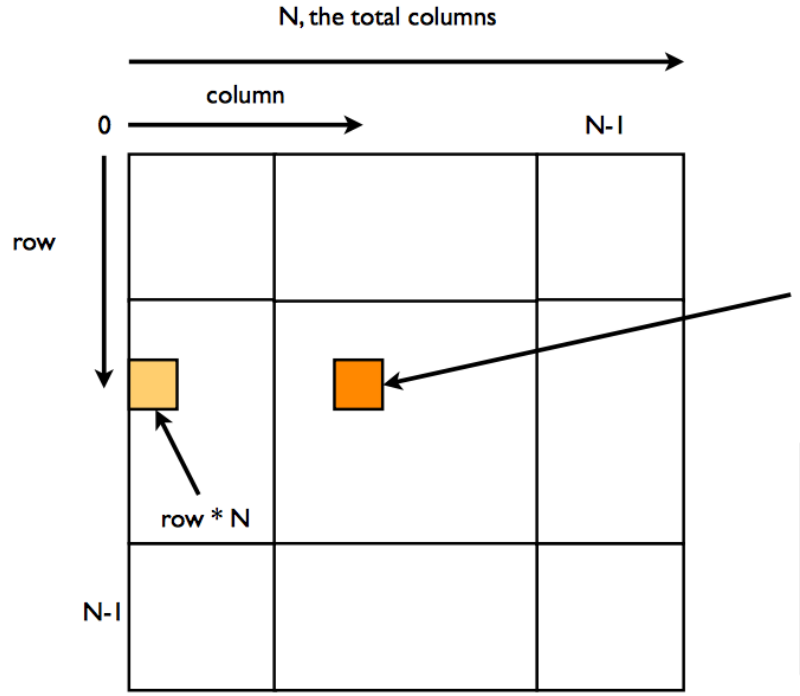
- **Matrices are flattened into linear memory**
  - the programmer needs to know how the matrix is laid out in memory and then compute the distance from the beginning of the matrix.
  - C language uses row-major order, where rows are stored one after the other in memory.



Case Study: Matrix Multiplication



- **Accessing Matrices in Linear Memory**



a[row][column] = a[offset]

offset = column + row * N

In CUDA:

```
int col = blockIdx.x*blockDim.x+threadIdx.x;
int row = blockIdx.y*blockDim.y+threadIdx.y;
int index = col + row * N;
A[index] = …
```

Case Study: 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;
          }
}
```

Case Study: Matrix Multiplication

- The main() function, that runs in the host, begins with the declaration of 3 matrices and 3 pointers.

  By convention when we refer to host variables we use "h_" and when we refer to device variables we use "d_".

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define N 512
#define BLOCK_SIZE_DIM 16

int main() {

 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);
```

Case Study: Matrix Multiplication

- After initializing matrices we have to allocate device memory with cudaMalloc that allows the programmer to allocate dynamic memory in the graphic card global memory.
- The next step is to copy the matrices from the main host memory to the device memory. This can be done with cudaMemcpy and its parameter cudaMemcpyHostToDevice .

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

// Copy the information in the device
checkCuda(cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice));
checkCuda(cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice));
```

Case Study: Matrix Multiplication

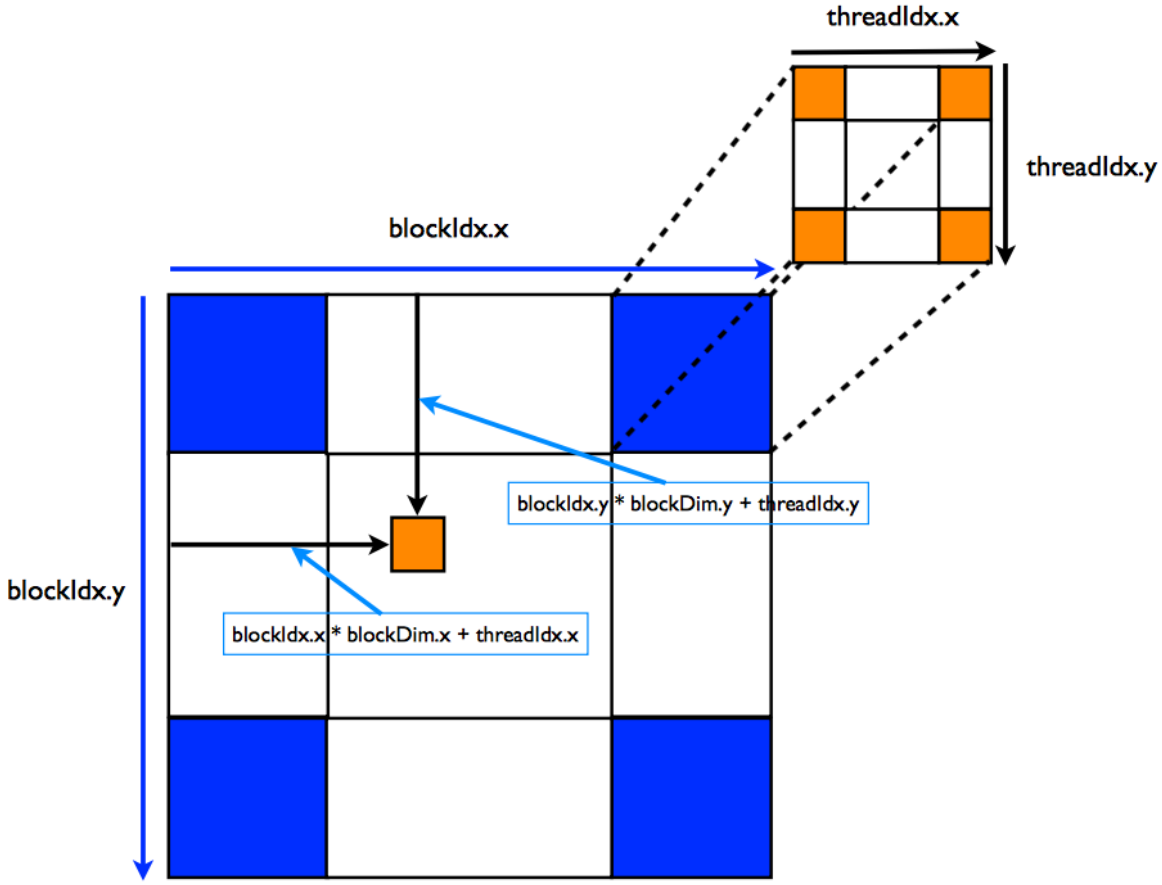
- Remember:
  - A kernel is the function that will be run by all the threads in a grid. During its invocation, one can specify the thread hierarchy with a special execution configuration syntax <<< >>>.
  - With this call, the programmer indicates the number of blocks in each dimension in the first parameter and the threads per block in each dimension in the second one.
- In our example:

```
// CUDA threads structure definition
dim3 dimGrid((N + BLOCK_SIZE_DIM -1) / BLOCK_SIZE_DIM,
             (N + BLOCK_SIZE_DIM -1) / BLOCK_SIZE_DIM);
dim3 dimBlock(BLOCK_SIZE_DIM, BLOCK_SIZE_DIM);

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

Case Study: Matrix Multiplication

- In this example we define a one block grid. The block have the same dimensions like the final product matrix. In this way, every thread will calculate one number of the final matrix.



Case Study: Matrix Multiplication

- If the host requires that the GPU computation is done before proceeding (e.g. , results are needed),
- →we can use **cudaDeviceSynchronize**() as an explicit barrier that waits to all threads are finished.

Case Study: Matrix Multiplication

- Finally we have to bring the result matrix from device memory to host memory with cudaMemcpy instruction and its parameter cudaMemcpyDeviceToHost.

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

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

    checkCuda(cudaFree(d_a));
    checkCuda(cudaFree(d_b));
    checkCuda(cudaFree(d_c));

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

Case Study: Matrix Multiplication

- In our toy example we could initialize the matrices with rand() function.

```
void initializeMatrices(double matrix_a[N][N], double matrix_b[N][N]) {
          srand(time(NULL));
          for (int i=0; i<N; i++) {
                    for (int j=0; j<N; j++) {
                              matrix_a[i][j] = rand()%100;
                              matrix_b[i][j] = rand()%100;
                    }
          }
}
```

Case Study: Matrix Multiplication

- And show the results with:

```
void showResults(double matrix_a[N][N], double matrix_b[N][N], double matrix_c[N][N]) {
        printf("***** MATRIX A ***** \n");
        for (int i=0; i<N; i++) {
                for (int j=0; j<N; j++) {
                        (j % N == N-1) ? printf("%f \n", matrix_a[i][j]) :
                        printf("%f,", matrix_a[i][j]);
                }
        }
        printf("***** MATRIX B ***** \n");
        for (int i=0; i<N; i++) {
                for (int j=0; j<N; j++) {
                        (j % N == N-1) ? printf("%f \n", matrix_b[i][j]) :
                        printf("%f,", matrix_b[i][j]);
                }
        }
        printf("***** MATRIX C ***** \n");
        for (int i=0; i<N; i++) {
                for (int j=0; j<N; j++) {
                        (j % N == N-1) ? printf("%f \n", matrix_c[i][j]) :
                        printf("%f,", matrix_c[i][j]);
                }
        }
```

Parallel Computing with CUDA

1. Basic elements of GPU programming
2. CUDA programming models basics
3. Thread organization
4. Memory organization
5. Launching a CUDA kernel
6. Case study: Matrix Multiplication
7. Timing the kernel

→Ex

Timing with `nvprof`

- Command-line profiling tool:

```
$ nvprof [nvprof_args] <application> [application_args]
```

- Collect timeline information from CPU and GPU activity, including kernel execution, memory transfers, and CUDA API calls.

- More information about nvprof options can be found by using the following command:

```
$ nvprof --help
```

nvprof: "Hello World"

```
[bsc31991@nvblogin1 CUDAlabs]$ nvprof ./helloGPU
Hello World from CPU
==7671== NVPROF is profiling process 7671, command: ./helloGPU
Hello World from GPU
Hello World from GPU
Hello World from GPU
Hello World from GPU
Hello World from GPU
Hello World from GPU
==7671== Profiling application: ./helloGPU
==7671== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  221.79us         1  221.79us  221.79us  221.79us  helloFromGPU(void)

==7671== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 71.77%  125.63ms         1  125.63ms  125.63ms  125.63ms  cudaLaunch
 27.74%  48.561ms         1  48.561ms  48.561ms  48.561ms  cudaDeviceReset
  0.39%  675.69us       166  4.0700us     170ns  151.92us  cuDeviceGetAttribute
  0.05%  82.786us         2  41.393us  38.525us  44.261us  cuDeviceTotalMem
  0.04%  68.116us         2  34.058us  30.130us  37.986us  cuDeviceGetName
  0.01%  19.419us         1  19.419us  19.419us  19.419us  cudaConfigureCall
  0.00%  2.6680us         2  1.3340us     341ns  2.3270us  cuDeviceGetCount
  0.00%  1.3050us         4     326ns     192ns     577ns  cuDeviceGet
```

The first half of the message contains output from the program, and the second half contains output from nvprof.

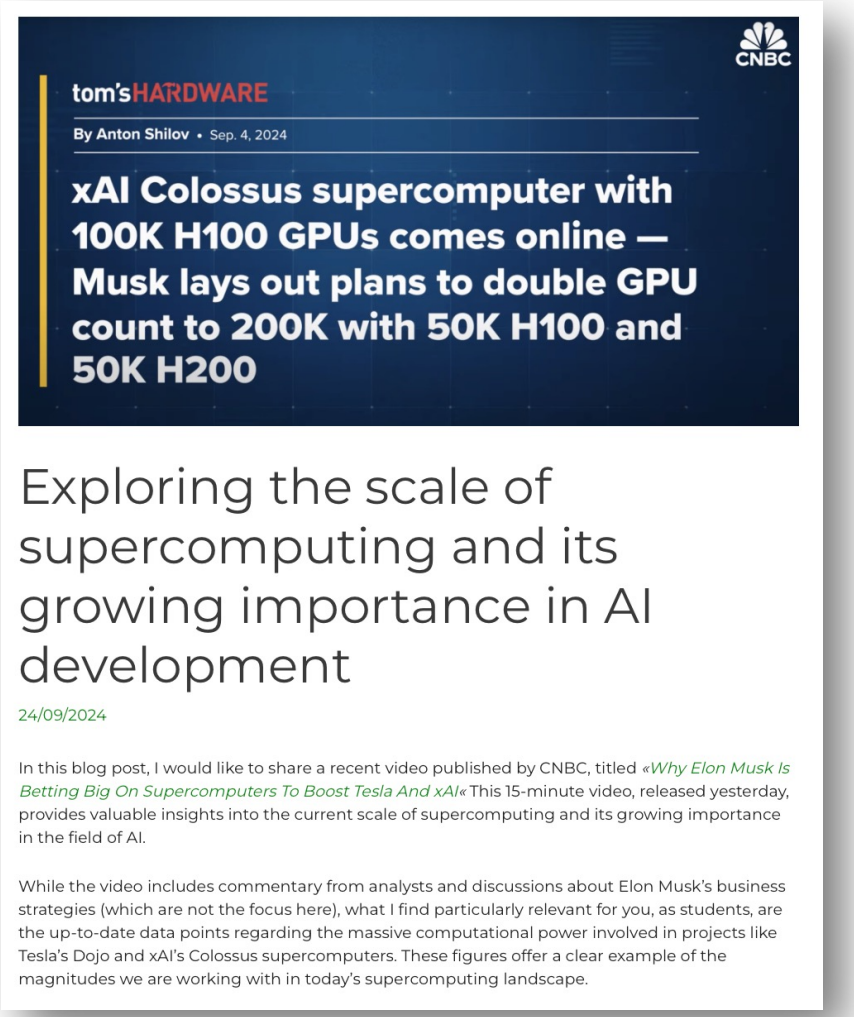
Laboratori 5: Getting started with CUDA

PR02: Presentation

PR02: PRESENTATION

- **Exploring the scale of supercomputing and its growing importance in IA development (*)**
- Type of presentation:
  - Slides presentation (**8-12** minutes)
  - All students should submit a PDF version of slides to the Racó FIB (Ex02 inbox)
- Deadline and presentation day:
  - Monday **21/10/2024** One student will present (randomly) (**)

(**) For maximum grade of this homework the student must attend the class during the presentation day



(*) https://torres.ai/exploring-the-scale-of-supercomputing-and-its-growing-importance-in-ai-development/

PR01: PRESENTATION

- Presentation of the section:
  HPC and AI convergence
- Types of presentation:
  - Slides presentation (5-10 minutes)
  - All students should submit a PDF version of slides to the Racó (as a PR01) Wednesday 14/10/2024
- Presentation day:
  - Monday ~~14/10/2024~~ **21/10/2024**
  - One student will present (randomly) (*)

(*) For maximum grade of this homework the student must attend the class during the presentation day

PR: Student PResentations

- Good practical experience for students!

  and … a way to stimulate homework accomplishment

- Short presentation: “X” minutes + slides (to support presentation)

- 1 student **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)
```

PR: Student PResentations

- All students should submit a version of slides to the Racó (as a Homework) before the deadline
  - Given the design of the course to maximize student learning, **exercises must be completed with the understanding that any student may be selected to present their work to the class**, followed by a discussion.

  - **To receive full credit, students must attend the presentation class. Failure to attend will result in the exercise grade being halved.**

List of students and their assigned “lucky” number

1	ALVAREZ ARAGONÉS, RUBÉN
2	ÁLVAREZ GÓMEZ-CALCERRADA, LUCÍA
3	ANDREU LOPEZ, RAMON MANEL
4	ATIENZA VILELA, CARLA
5	AUBACH ALTES, ARTUR
6	BAIGES TRILLA, ROGER
7	BARNADAS CONANGLA, EDUARD
8	BARRENECHEA PEREA, PABLO
9	BENNÀSSAR MARTÍN, JOAN
10	BERNAUS CASADESÚS, JOAN
11	BIASIZZO SERRA, ENZO
12	BONET VILA, VIOLETA
13	BRICHS RALLÓ, MIREIA
14	CANTARERO CARRERAS, ADRIÀ
15	CARRIÓN BASTIDA, MARTA
16	CASANOVAS POIRIER, ANNA
17	CHEN, HAO
18	CHEN, PENGCHENG
19	CHEN, ZHIHAO
20	DURÁN LAPLAZA, NILS
21	FIGUERAS FERNÁNDEZ, ALBA
22	FLORES ALBÓ, ADRIÀ
23	FURRIOLS LLIMARGAS, LLUC
24	GIL CASAS, MARIA
25	GONZÁLEZ MONFORT, PABLO
26	GRANJA I BAYOT, JORDI
27	GUTIÉRREZ KITAJIMA, LUIS-KAZUTO
28	HIDALGO PUJOL, PAU
29	IBARS MINGUELLA, ALEIX
30	JEREZ CUBERO, ALBERTO
31	JUNCAROL PI, MARTA
32	LLOPART FERNANDEZ, NURIA
33	LÓPEZ GARCÍA, DANIEL
34	MACIÀ CODERA, NIL
35	MARGARIT FISAS, POL
36	MARTÍNEZ MARTÍNEZ, EVA
37	MEJIA ROTA, CESAR ELIAS
38	MEYA MORALES, MÁXIMO
39	MIRA GARCÍA, MAX
40	MONROY MIR, LOLA
41	MORA LADÀRIA, JAUME
42	NADAL PAR, MARTA
43	NAVARRO NAVARRO, ALEX
44	PEREZ PRADES, POL
45	PRAT MORENO, PAU
46	PUMARES BENAIGES, IRENE
47	RISSO MATAS, ABRIL MARÍA
48	RODRÍGUEZ SANSALONI, MIQUEL
49	ROPERO SERRANO, MIQUEL
50	SELVAS SALA, CAI
51	ZHOU, ZHIQIAN