

**POLITECNICO**  
MILANO 1863

**GPUs and Heterogeneous Systems**  
**(programming models and architectures)**

**Introduction to CUDA**

# What is CUDA?

- CUDA (Compute Unified Device Architecture) has been introduced in 2006 with NVIDIA Tesla architecture
- C/C++-like language to write programs that can use GPU as a processing resource to accelerate functions
- CUDA's abstraction level closely matches the characteristics and organization of NVIDIA GPUs

# What is CUDA?

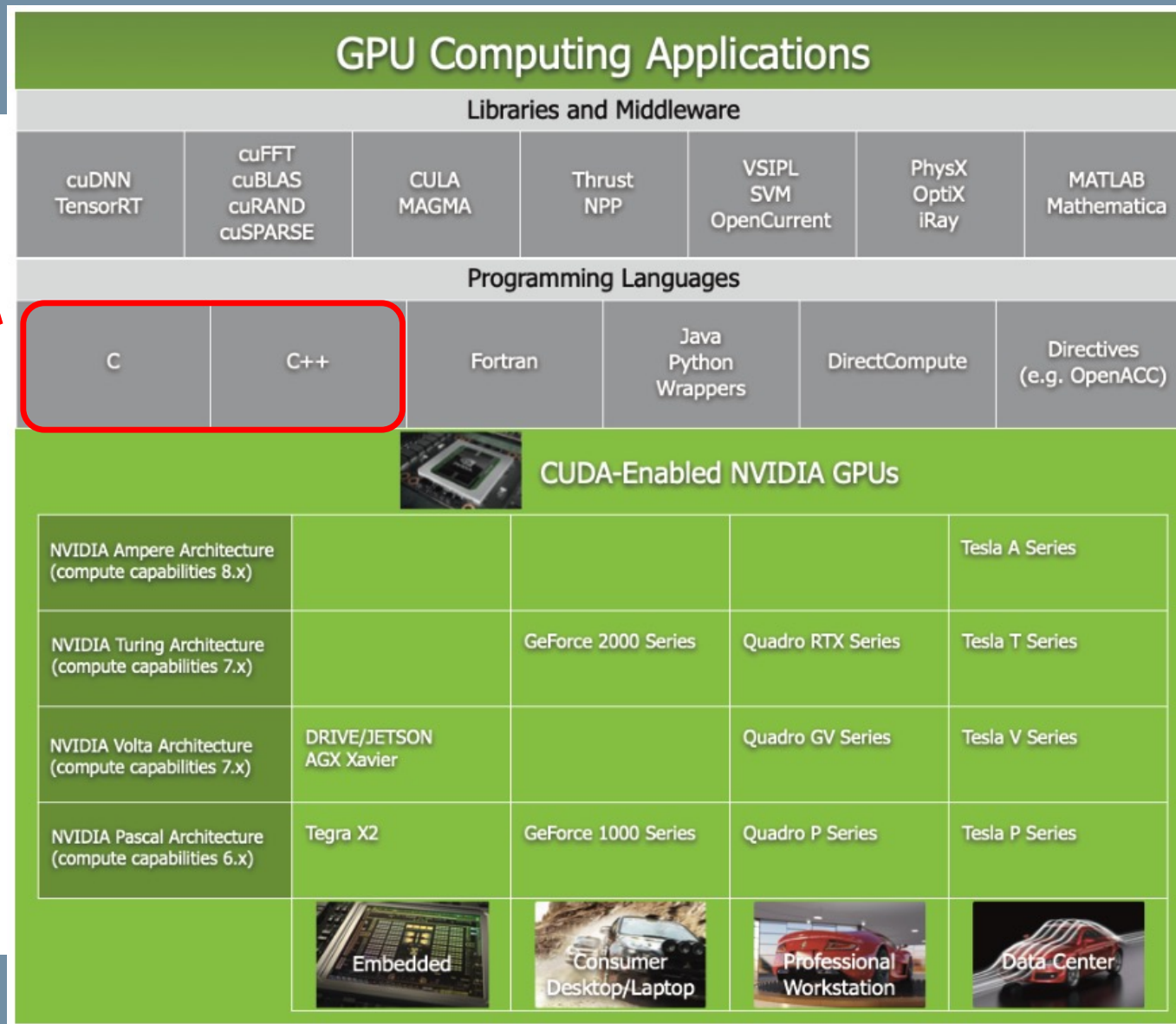
- A platform designed jointly at software and hardware levels to make use of GPUs in general-purpose computing:
  - **Software** –CUDA allows to program the GPU with minimal extensions to enable heterogeneous programming and attain an efficient and scalable execution
  - **Firmware** – CUDA offers a driver oriented to GPGPU programming and APIs to manage devices, memory, etc.
  - **Hardware** – CUDA exposes GPU parallelism for general-purpose computing via several multiprocessors endowed with cores and a memory hierarchy

# What is CUDA?

- From an application perspective, CUDA defines:
  - **Architecture model**
    - with many processing cores grouped in multiprocessors who share a SIMT control unit
  - **Programming model**
    - Based on massive data parallelism and fine-grained parallelism
    - **Scalable**: The code is executed on a different number of cores without recompiling it
  - **Memory model**
    - More explicit to the programmer, where caches are not transparent anymore

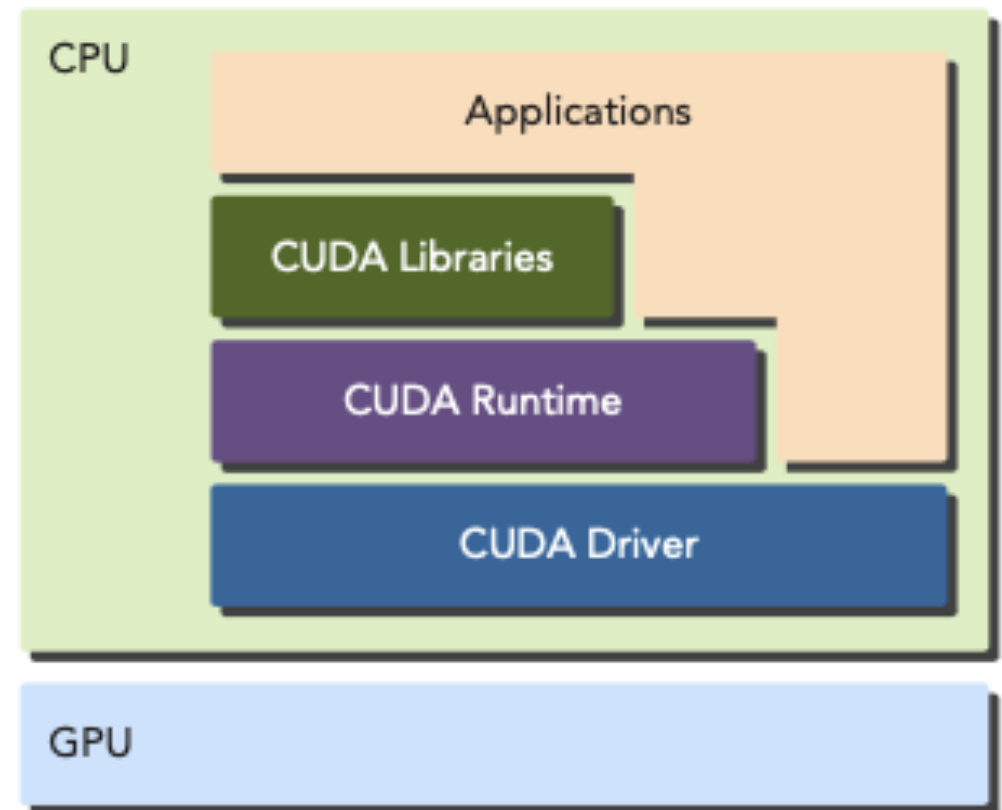


# CUDA software platform



# CUDA software platform

- CUDA provides 2 APIs for manage the GPU device:
  - CUDA Driver
  - CUDA Runtime ←
- They are mutually exclusive
- On top of them, CUDA offers libraries for executing “popular” functions/algorithms



# CUDA version and compute capabilities

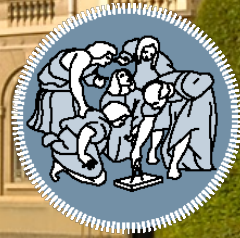
- **Compute capability**: version number identifying the set of features supported by the GPU hardware
  - Major version number identifies the GPU architecture
  - Minor version number identifies incremental improvements
- **CUDA version**: version number identifying the revision of the CUDA software platform
  - CUDA platform typically also include software features that are independent of hardware generation

In this course we will use:  
Compute capabilities  $\geq 3.7$   
CUDA version  $\geq 9.0$

# Parallelism in CUDA

- **Data parallelism** – many data items can be operated on with the same algorithm at the same time
- **Task parallelism** – various tasks or functions can be operated independently in parallel
- CUDA mainly exploits data parallelism and also task parallelism to achieve high performance





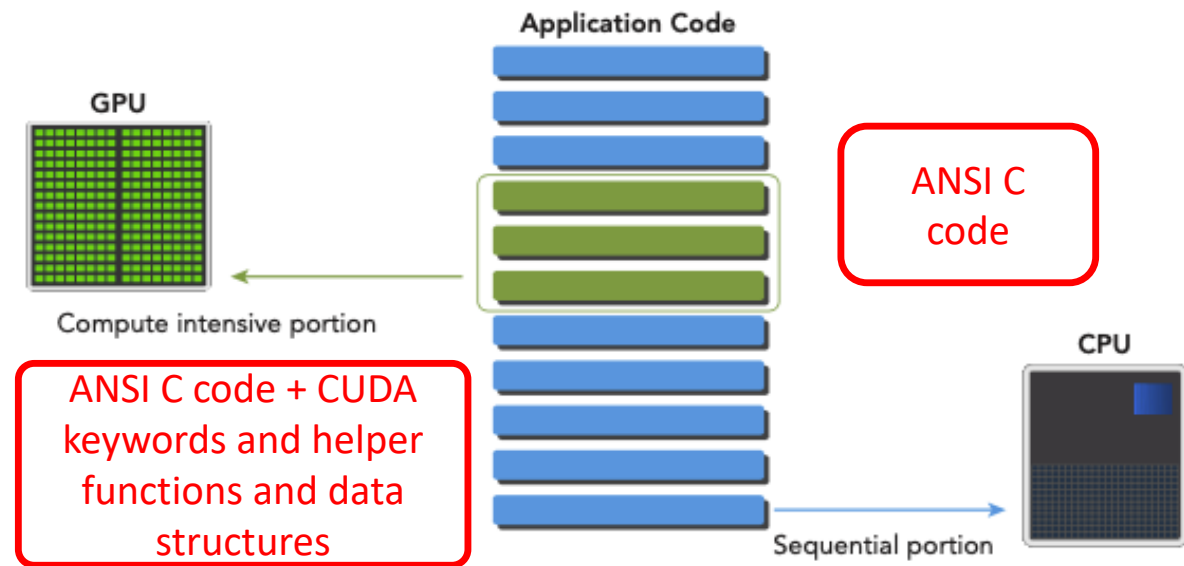
**POLITECNICO**  
MILANO 1863

**GPUs and Heterogeneous Systems**  
(programming models and architectures)

**CUDA programming model**

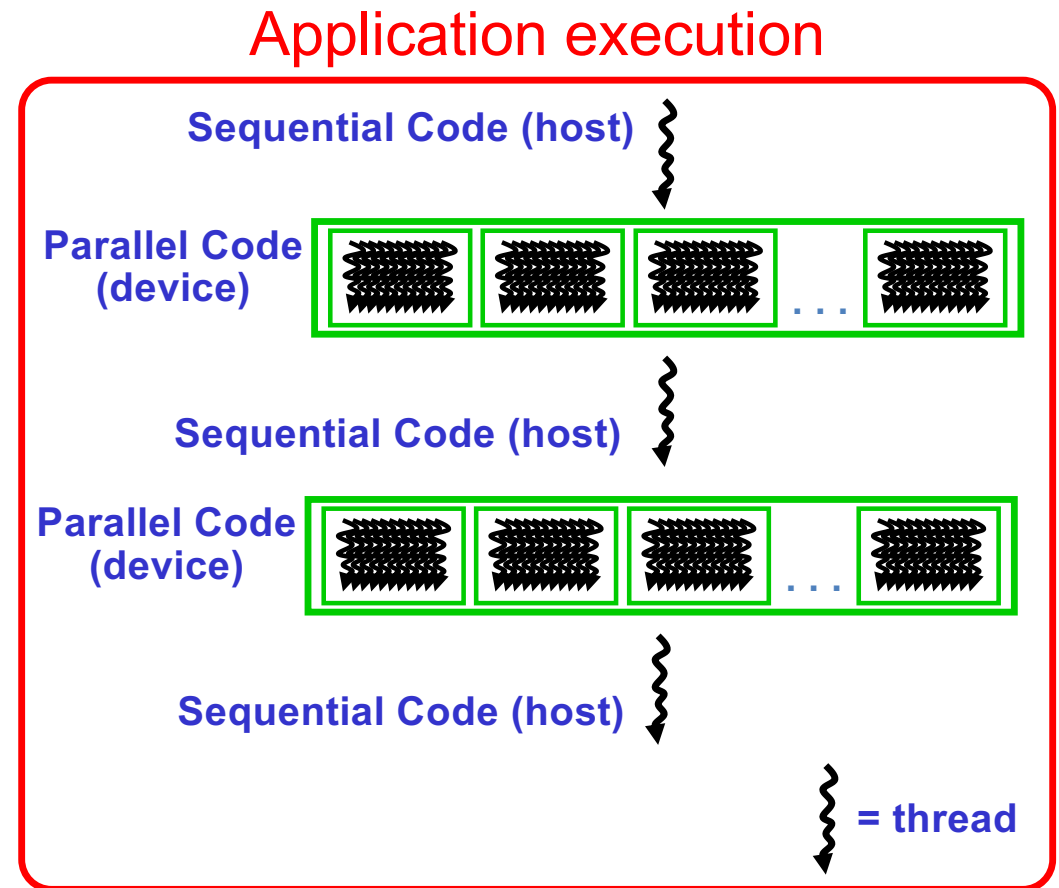
# CUDA program structure

- A **CUDA program** is organized in
  - The **serial code** executed on the **host** (i.e., the CPU)
  - One or **many independent functions** (called **kernels**) parallelized on the **device** (i.e., the GPU)



# CUDA program structure

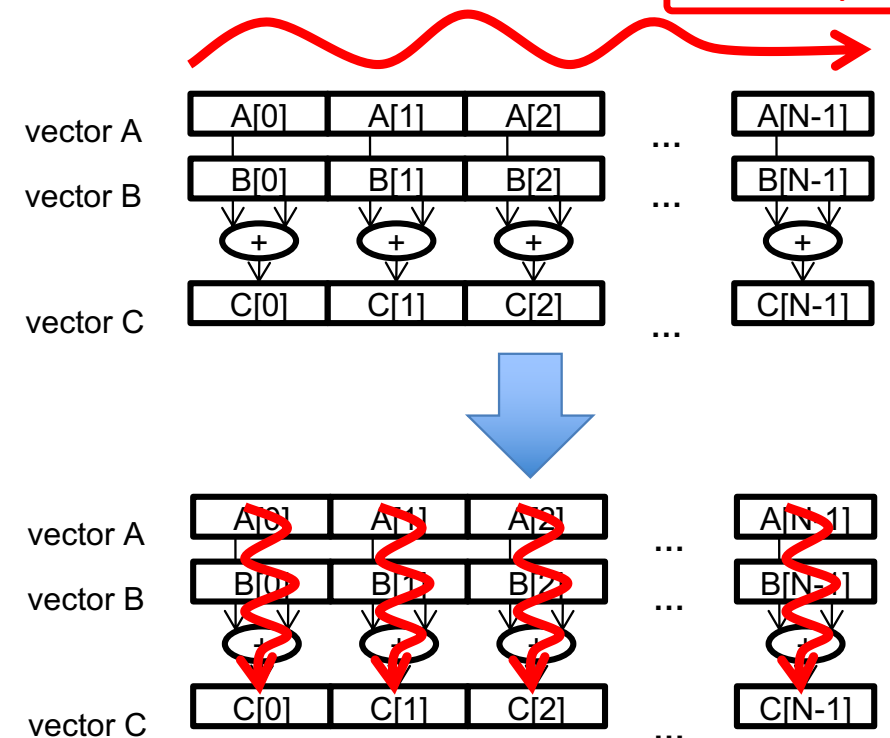
- A CUDA program is organized in
  - The serial code executed on the **host** (i.e., the CPU)
  - One or many independent functions (called **kernels**) parallelized on the **device** (i.e., the GPU)



# Parallelization of a kernel

- **Single-Program Multi-Data (SPMD)** parallel paradigm
  - The set of input data is decomposed into a stream of separate data elements
  - The kernel function is defined to operate on the single data element
  - The thread executes the kernel code on a single data element

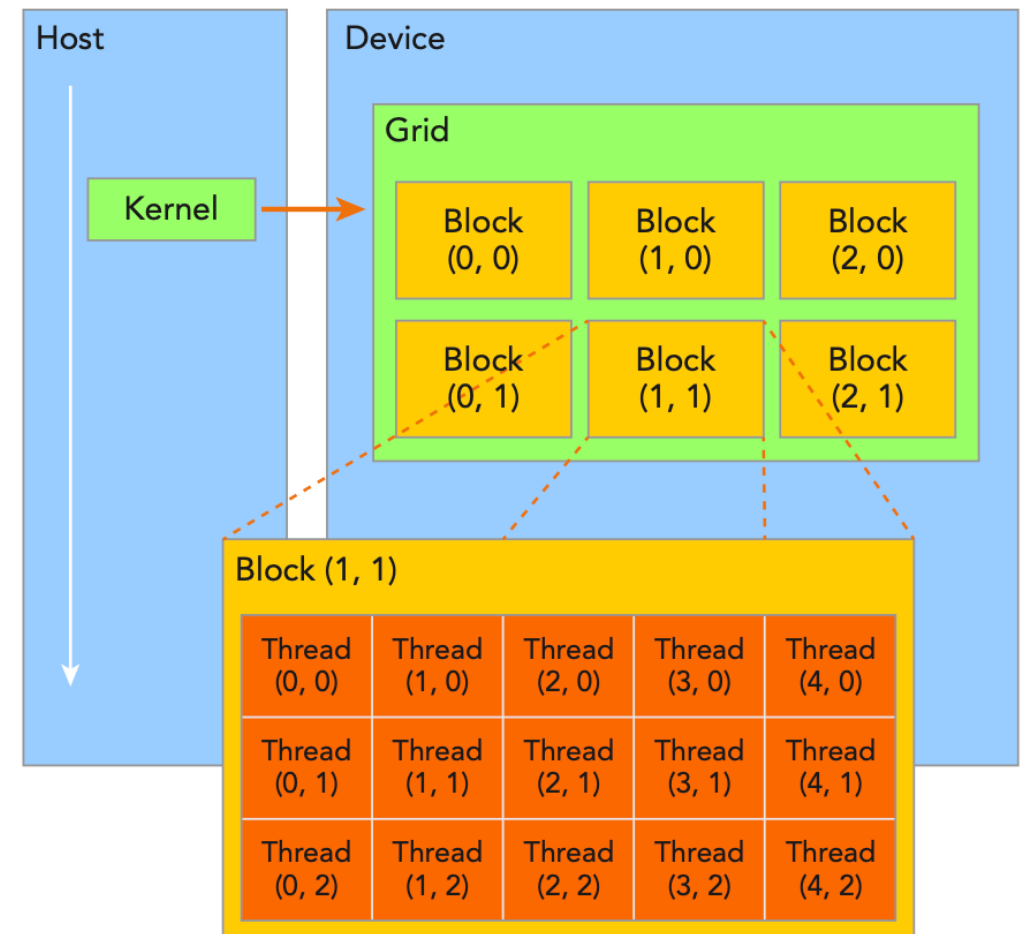
Vector sum example: for loop



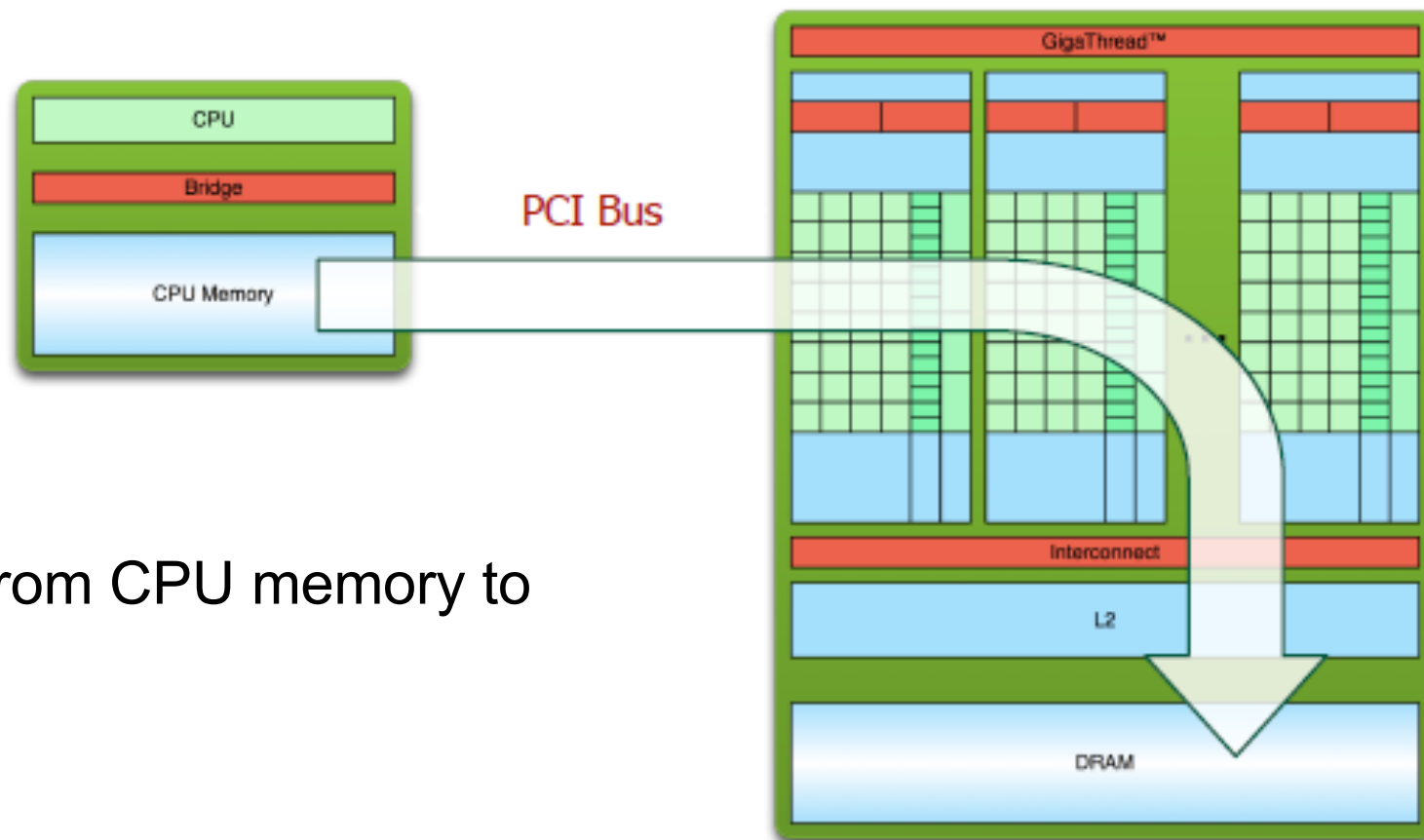


# CUDA thread hierarchy

- A kernel executes in parallel across a set of parallel threads
- Threads are grouped in thread **blocks**
- Thread blocks are organized in a **grid**
- Blocks and grids are organized as an N-dimensional (up to 3) array



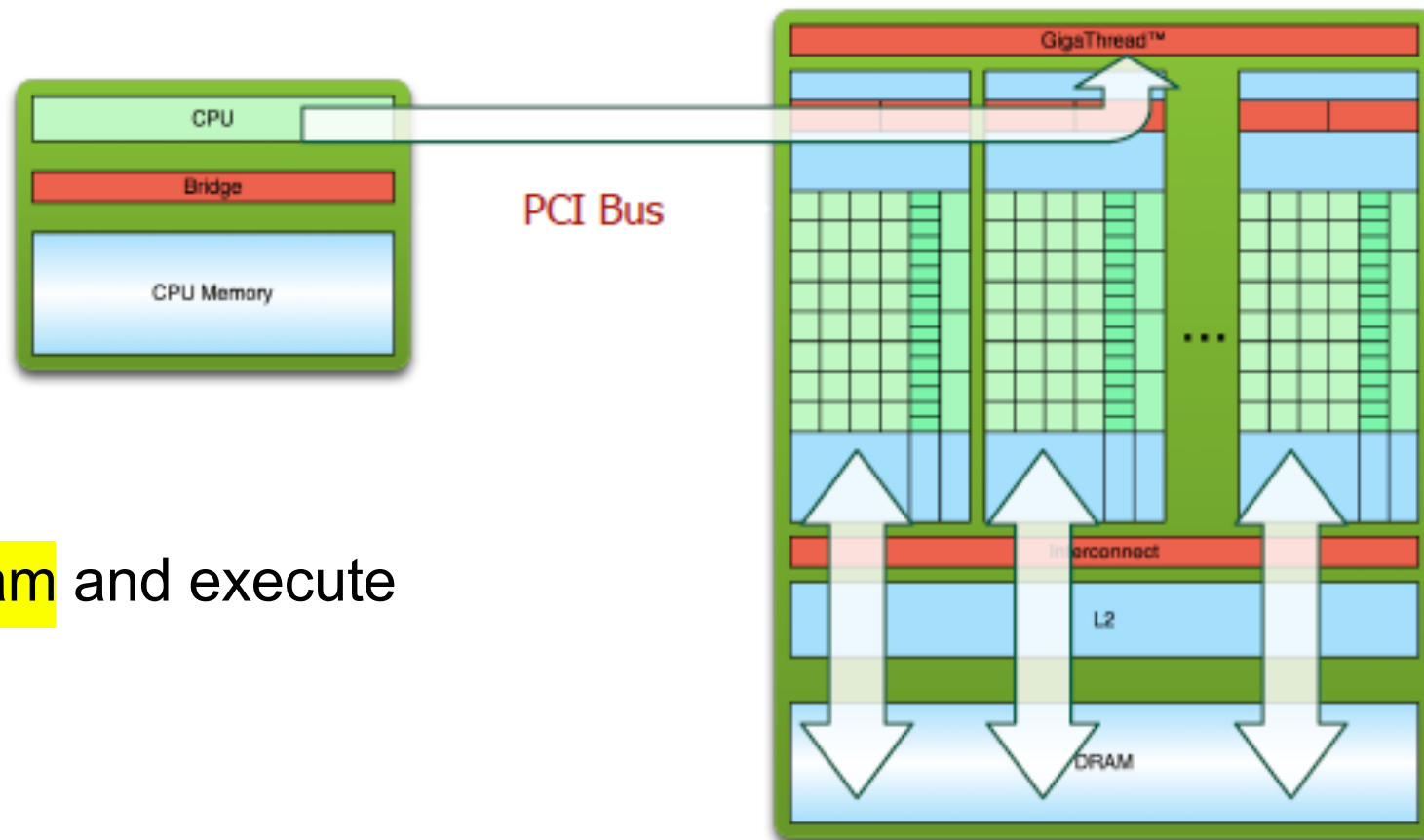
# CUDA program execution flow



1. Copy input data from CPU memory to GPU memory

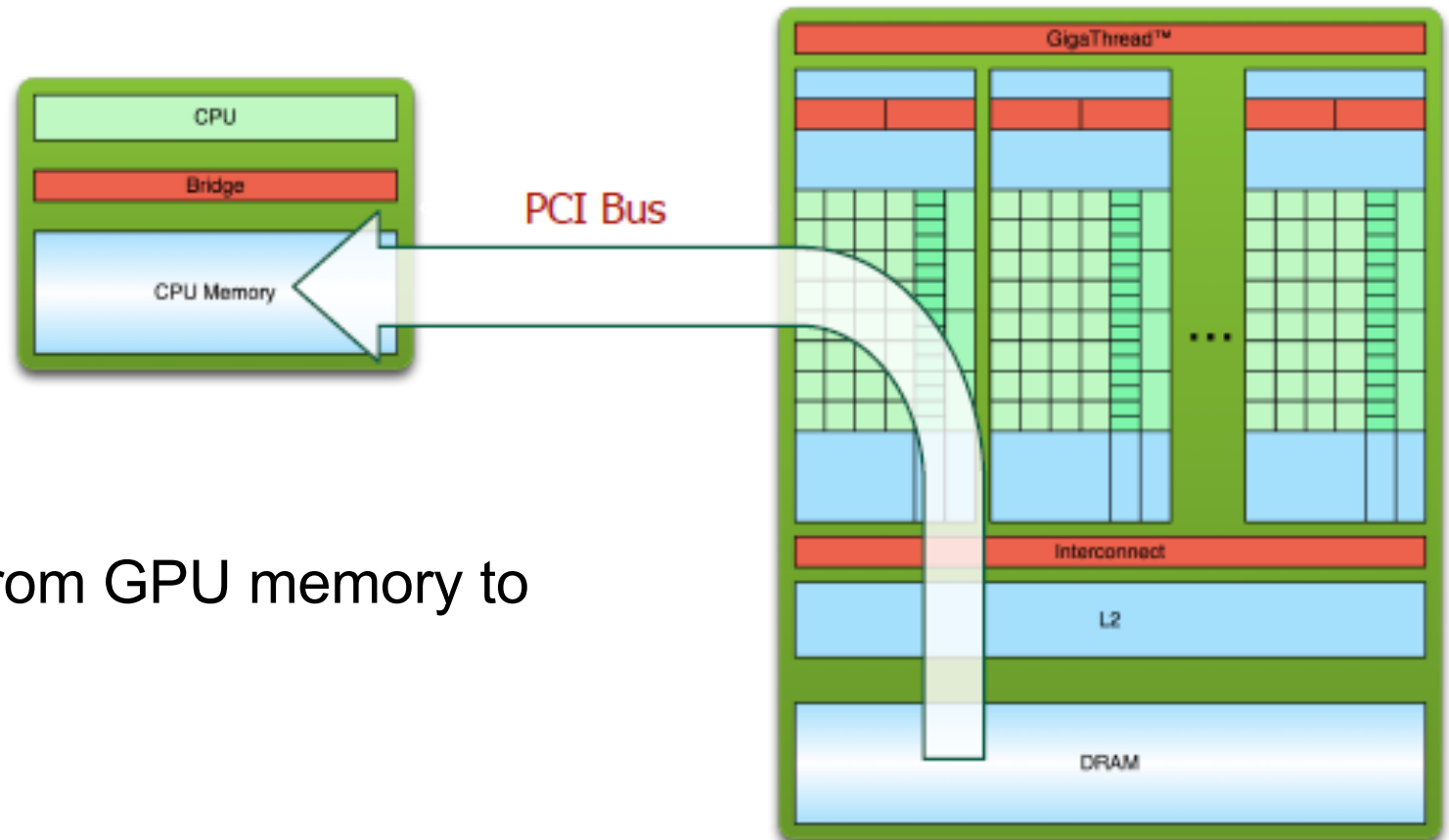


# CUDA program execution flow



2. Load GPU program and execute

# CUDA program execution flow



3. Transfer results from GPU memory to CPU memory

# A running example

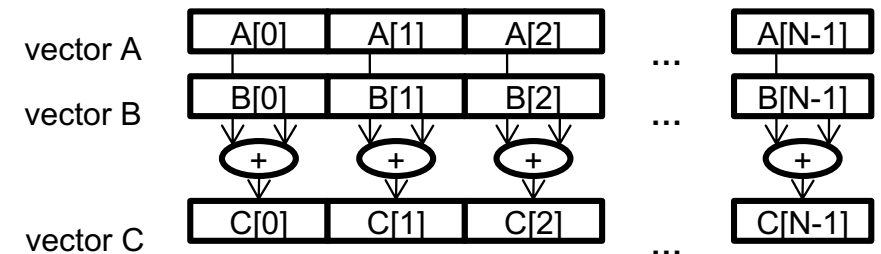
- C program computing the sum of two integer vectors:

```
#define N 1024

void vsum(int *a, int *b, int *c){
    int i;
    for (i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}

int main(){
    int va[N], vb[N], vc[N];
    /*... input data acquisition ...*/
    vsum(va, vb, vc);
    /*... output data visualization ...*/
}
```

For loop sequentially  
scanning the arrays



# Structure of a CUDA program

```
#include <cuda_runtime.h>
#define N 1024

/* kernel function */

int main() {
    int h_va[N], h_vb[N], h_vc[N];
    /*... input data acquisition ...*/
    /* device memory allocation */
    /* CPU->GPU data transmission */
    /* kernel launch */
    /* GPU->CPU data transmission */
    /* device memory freeing */
    /*... output data visualization ...*/
}
```

# Structure of a CUDA program

```
#include <cuda_runtime.h>
#define N 1024

/* kernel function */

int main() {
    int h_va[N], h_vb[N], h_vc[N];
    /*... input data acquisition ...*/
    /* device memory allocation */
    /* CPU->GPU data transmission */
    /* kernel launch */
    /* GPU->CPU data transmission */
    /* device memory freeing */
    /*... output data visualization ...*/
}
```

Host code

Device code

- It is a C/C++ program with CUDA extensions
- The kernel function has some restrictions:
  - Access to device memory only
  - void return type
  - Asynchronous behavior
  - No variable number of parameters
  - No static variables
  - No function pointers

# Kernel function

```
#include <cuda_runtime.h>
#define N 1024

__global__ void vsumKernel(int *a, int *b, int *c){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

int main(){
    /* ... */
}
```



# Kernel function

```
#include <cuda_runtime.h>
#define N 1024

__global__ void vsumKernel(int *a, int *b, int *c){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

int main(){
    /* ... */
}
```

## Single Program Multiple Data (SPMD) parallel paradigm

- The kernel function works on the single data element
- The kernel function includes the code executed by each thread
- The for loop is replaced by a grid of threads, each one working on a single data element

# Kernel function

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

```
#define N 1024
```

```
__global__ void vsumKernel(int *a, int *b, int *c){
```

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

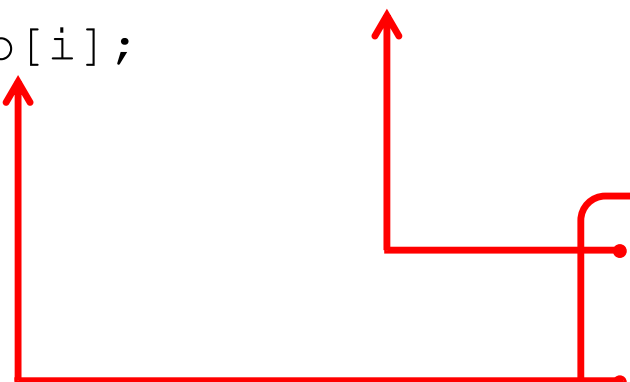
```
    c[i] = a[i] + b[i];
```

```
}
```

```
int main(){
```

```
    /* ... */
```

```
}
```

- 
- Mapping of each thread to the single data element
  - Elaboration on the data element

# Kernel function

```
#include <cuda_runtime.h>
#define N 1024

__global__ void vsumKernel(int *a, int *b, int *c){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

int main(){
    /* ... */
}
```

blockIdx: id of the block in the grid  
blockDim: size of the block (#threads)  
threadIdx: id of the thread in the block  
gridDim: size of the grid (#blocks)

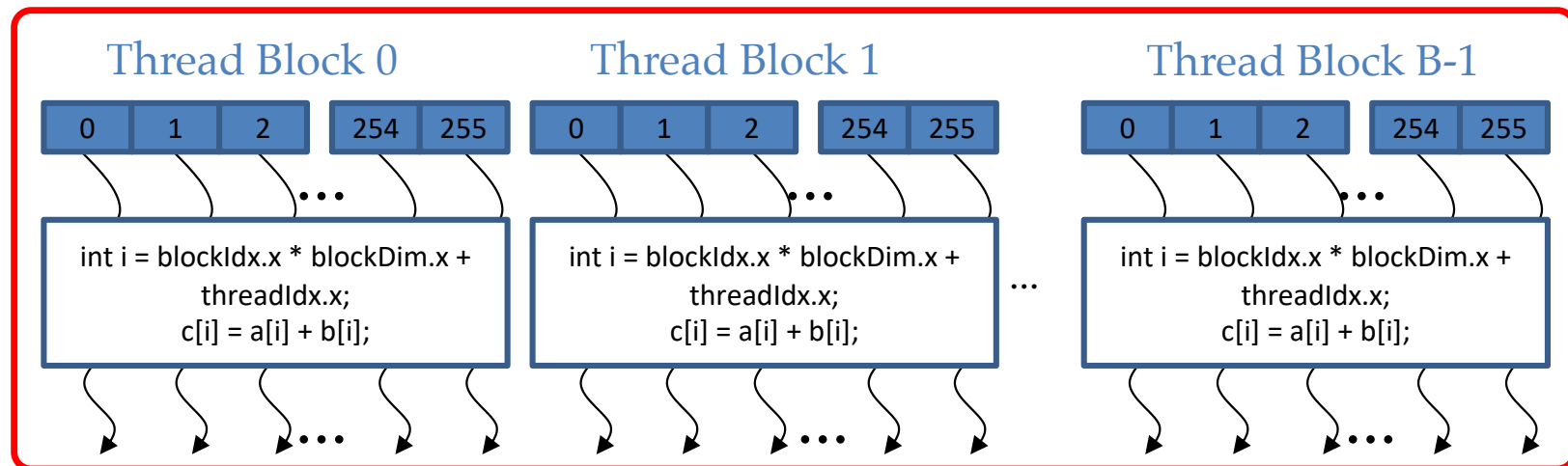
CUDA built-in struct variables with three fields: x, y, z (3D grid!)

# Kernel function

```
#include <cuda_runtime.h>
#define N 1024

__global__ void vsumKernel(int *a, int *b, int *c){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}
```

```
int main(){
    /* ... */
}
```



# Kernel function

```
#include <cuda_runtime.h>
#define N 1024

__global__ void vsumKernel(int *a, int *b, int *c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

int main() {
    /* ... */
}
```

Pointers to the device memory

Thread local variable

- Host and device codes have separate scopes
- Local variables are per-thread private
- The memory locations pointed by the parameters are accessed by all the threads

# Kernel function

```
#include <cuda_runtime.h>
#define N 1024

__global__ void vsumKernel(int *a, int *b, int *c){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

int main(){
    /* ... */
}
```

## Function qualifier

	Executed on the:	Only callable from the:
device	device	device
<u>global</u>	device	host
<u>host</u>	host	host



# Device memory allocation

```
#include <cuda_runtime.h>
#define N 1024
/*...*/

int main(){
    int h_va[N], h_vb[N], h_vc[N];
    int *d_va, *d_vb, *d_vc;
    /* device memory allocation */
    cudaMalloc(&d_va, N*sizeof(int));
    cudaMalloc(&d_vb, N*sizeof(int));
    cudaMalloc(&d_vc, N*sizeof(int));
    /* ... */
}
```

# Device memory allocation

```
#include <cuda_runtime.h>
#define N 1024
/*...*/

int main(){
    int h_va[N], h_vb[N], h_vc[N];
    int *d_va, *d_vb, *d_vc;
    /* device memory allocation */
    cudaMalloc(&d_va, N*sizeof(int));
    cudaMalloc(&d_vb, N*sizeof(int));
    cudaMalloc(&d_vc, N*sizeof(int));
    /* ... */
}
```

Device memory has to be allocated similarly to host memory (heap)

Device memory is separate from the host one! -> pointers to device memory CANNOT be used in the host program

- Pointer storing the address of the allocated memory
- Size in bytes to be allocated

# Data transfer to the device

```
/*...*/
```

```
int main() {  
    /*...*/  
    /* CPU->GPU data transmission */  
    cudaMemcpy(d_va, h_va, N*sizeof(int),  
               cudaMemcpyHostToDevice);  
    cudaMemcpy(d_vb, h_vb, N*sizeof(int),  
               cudaMemcpyHostToDevice);  
    /* ... */  
}
```

# Data transfer to the device

```
/*...*/
```

Data have to be transferred explicitly to the device memory

```
int main() {  
    /*...*/  
    /* CPU->GPU data transmission */  
    cudaMemcpy(d_va, h_va, N*sizeof(int),  
               cudaMemcpyHostToDevice);  
    cudaMemcpy(d_vb, h_vb, N*sizeof(int),  
               cudaMemcpyHostToDevice);  
    /* ... */  
}
```

- Pointer to the destination memory
- Pointer to the source memory
- Size in bytes to be transmitted
- Transmission direction

# Launch the kernel function

```
/*...*/  
int main() {  
    /*...*/  
    /* kernel launch */  
    dim3 blocksPerGrid(N/256, 1, 1);  
    dim3 threadsPerBlock(256, 1, 1);  
    vsumKernel<<<blocksPerGrid, threadsPerBlock>>>(d_va, d_vb,  
                                                    d_vc);  
  
    /* ... */  
}
```

# Launch the kernel function

```
/*...*/  
int main() {  
    /*...*/  
    /* kernel launch */  
    dim3 blocksPerGrid(N/256, 1, 1);  
    dim3 threadsPerBlock(256, 1, 1);  
    vsumKernel<<<blocksPerGrid, threadsPerBlock>>>(d_va, d_vb,  
                                                    d_vc);  
    /* ... */  
}
```

• Set execution configuration  
parameters: block and grid dimensions  
• dim3 is a struct with three fields: x, y, z

Launch the kernel

The parameters are the  
pointers to device memory



# Launch the kernel function

```
/*...*/  
int main() {  
    /*...*/  
    /* kernel launch */  
    vsumKernel<<<N/256, 256>>>(d_va, d_vb, d_vc);  
    /* ... */  
}
```

Alternative compact version for  
1D kernel launch



# Data transfer to the host

```
/*...*/
```

```
int main() {  
    /*...*/  
    /* GPU->CPU data transmission */  
    cudaMemcpy(h_vc, d_vc, N*sizeof(int),  
               cudaMemcpyDeviceToHost);  
  
    /* ... */  
}
```

# Data transfer to the host

```
/* ... */
```

```
int main() {  
    /* ... */  
    /* GPU->CPU data transmission */  
    cudaMemcpy(h_vc, d_vc, N*sizeof(int),  
               cudaMemcpyDeviceToHost);  
    /* ... */  
}
```

Data have to be transferred explicitly to the host memory

- Pointer to the destination memory
- Pointer to the source memory
- Size in bytes to be transmitted
- Transmission direction

4 possible directions:

`cudaMemcpyDeviceToHost`, `cudaMemcpyHostToDevice`,  
`cudaMemcpyDeviceToDevice`, `cudaMemcpyHostToHost`

# Device memory freeing

```
/*...*/

int main() {
    /*...*/
    /* device memory freeing */
    cudaFree(d_va);
    cudaFree(d_vb);
    cudaFree(d_vc);
    /* ... */
}
```

# Device memory freeing

```
/* ... */

int main() {
    /* ... */
    /* device memory freeing */
    cudaFree(d_va);
    cudaFree(d_vb);
    cudaFree(d_vc);
    /* ... */
}
```

Device memory has to be explicitly released by the application

cudaDeviceReset() may be called to reset all resources of the device used by the process

# The overall code

```
#include <cuda_runtime.h>
#define N 1024

__global__ void vsumKernel(int *a, int *b, int *c){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

int main(){
    int h_va[N], h_vb[N], h_vc[N];
    int *d_va, *d_vb, *d_vc;

    /*... input data acquisition ...*/

    cudaMalloc(&d_va, N*sizeof(int));
    cudaMalloc(&d_vb, N*sizeof(int));
    cudaMalloc(&d_vc, N*sizeof(int));

    cudaMemcpy(d_va, h_va, N*sizeof(int),
               cudaMemcpyHostToDevice);
    cudaMemcpy(d_vb, h_vb, N*sizeof(int),
               cudaMemcpyHostToDevice);
```

```
    dim3 blocksPerGrid(N/256, 1, 1);
    dim3 threadsPerBlock(256, 1, 1);
    vsumKernel<<<blocksPerGrid, threadsPerBlock>>>
        (d_va, d_vb, d_vc);

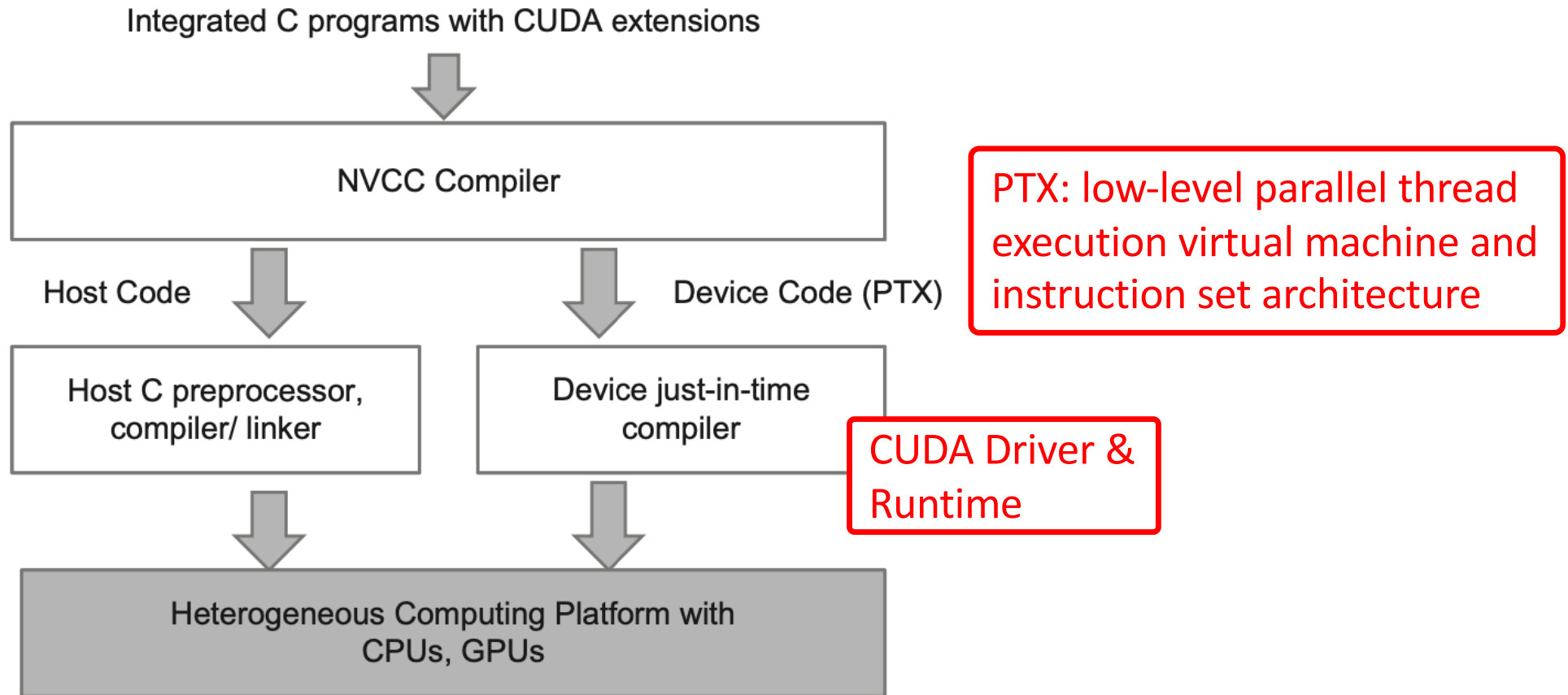
    cudaMemcpy(h_vc, d_vc, N*sizeof(int),
               cudaMemcpyDeviceToHost);

    cudaFree(d_va);
    cudaFree(d_vb);
    cudaFree(d_vc);

    /*... output data visualization ...*/
    return 0;
}
```

Typical CUDA C extension of  
source code file is .cu

# Compilation process



# Basic commands to compile and execute

- To compile:  
`nvcc vector_sum.cu -o vector_sum`
- To execute:  
`./vector_sum`



# Error handling

- Every CUDA function returns a flag
- The flag can be used for error handling

MACRO  
definition:

```
#define CHECK(call) \
{\
    const cudaError_t err = call;\
    if (err != cudaSuccess) {\
        printf("%s in %s at line %d\n", cudaGetErrorString(err), \
        __FILE__, __LINE__); \
        exit(EXIT_FAILURE);\
    }\
}
```

and use:

```
CHECK(cudaMalloc(&d_va, N*sizeof(int)));
```

# Error handling

- Kernel invocation returns `void`!
- To check errors at kernel launch:

Errors during kernel execution cannot be checked in this way... we talk about that in the next class

**MACRO  
definition:**

```
#define CHECK_KERNELCALL().\n{\n    const cudaError_t err = cudaGetLastError();\n    if (err != cudaSuccess) {\n        printf("%s in %s at line %d\\n", cudaGetErrorString(err),\n               __FILE__, __LINE__); \n        exit(EXIT_FAILURE);\n    }\n}
```

**and use:**

```
vsumKernel<<<blocksPerGrid, threadsPerBlock>>>(d_va, d_vb, d_vc);\nCHECK_KERNELCALL();
```

# Querying GPU information

- CUDA provides the API to query available GPU devices

```
/*...*/  
int dev;  
cudaDeviceProp devProp;  
cudaGetDevice(&dev);  
cudaGetDeviceProperties(&devProp, dev);  
/*...*/
```

# Querying GPU information

- CUDA provides the API to query available GPU devices

```
/*...*/  
int dev;  
cudaDeviceProp devProp;  
cudaGetDevice(&dev);  
cudaGetDeviceProperties(&devProp, dev);  
/*...*/
```

Get the id of the  
currently used device

Query the properties of  
the selected device

cudaDeviceProp is a C struct

# Querying GPU information

- CUDA provides the API to query available GPU devices

```
/*...*/  
printf("Major revision number:           %d\n",   devProp.major);  
printf("Minor revision number:          %d\n",   devProp.minor);  
printf("Name:                           %s\n",   devProp.name);  
printf("Total global memory:             %lu\n",  
devProp.totalGlobalMem);  
/*...*/
```

cudaDeviceProp is a C struct  
containing various HW-related information

# Querying GPU information

- CUDA provides the API to query available GPU devices

```
/*...*/  
printf("Total registers per block:      %d\n",  
devProp.regsPerBlock);  
printf("Maximum threads per block:      %d\n",  
devProp.maxThreadsPerBlock);  
for (int i = 0; i < 3; ++i)  
    printf("Maximum dimension %d of block:  %d\n", i,  
                                                  devProp.maxThreadsDim[i]);  
/*...*/
```

cudaDeviceProp is a C struct  
containing various HW-related information

# Querying GPU information

- It is possible also to query available GPU devices by means of `nvidia-smi` command-line tool
  - E.g.: `nvidia-smi -q -i 0` returns details about GPU 0

```
Timestamp                : Sun Jan 16 15:39:38 2022
Driver Version            : 460.32.03
CUDA Version              : 11.2

Attached GPUs             : 1
GPU 00000000:00:04.0
    Product Name           : Tesla K80
    Product Brand           : Tesla
    Product Architecture    : Kepler
    Display Mode            : Disabled
    Display Active          : Disabled
    Persistence Mode        : Disabled
    MIG Mode
        Current             : N/A
        Pending             : N/A
```

Not supported on Jetson boards

# References

- Slides mainly based on:
  - W.-m. W. Hwu , D. B. Kirk, I. El Hajj, **Programming Massively Parallel Processors: A Hands-on Approach, Chapter 2**
  - J. Chen, M. Grossman, T. McKercher, **Professional Cuda C Programming, Chapter 2**