

# CM2 - First steps in GPU computing with CUDA

E. Carlinet & J. Chazalon

*Fall 2024*



# GPU Computing Applications

## Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica
-------------------	---------------------------------------	---------------	---------------	-----------------------------	------------------------	-----------------------

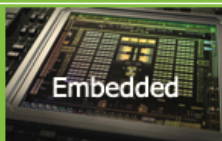
## Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)
---	-----	---------	----------------------------	---------------	------------------------------



## CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series



Embedded



Consumer  
Desktop/Laptop



Professional  
Workstation



Data Center

note: (missing series L and H)

# Enabling GPU acceleration in applications

- Option 1: Using libraries that uses CUDA as backend (e.g. Thrust, cuBLAS, cuDNN)

✓: Already implemented

✗: Limited flexibility if the feature is not implemented

- Option 2: Use compiler directives to offload computation to GPU (e.g. OpenACC)

Example:

```
#pragma acc parallel loop
for (int i = 0; i < n; ++i)
    h_C[i] = h_A[i] + h_B[i];
```

✓: Easy to use to adapt existing code

✗: Limited flexibility to take full advantage of GPU

- Option 3: Write your own CUDA kernels

✓: Full control over the GPU

✗: Requires learning a new language and API

# What is CUDA ?

## A product

It enables using NVidia GPUs for computation

## A C/C++ variant

Mostly C++17-compatible, with extensions (but also some restrictions!)

## A SDK

- A set of compilers and toolchains for various architectures

- Performance analysis tools

## A runtime

- An assembly specification
- Computation libraries (e.g. libcu++)

## A new industry standard

- Used by every major deep learning framework
- Replacing OpenCL as Vulkan is replacing OpenGL

# Your first kernel with CUDA

Summing two vectors in Regular C++:

```
// compute vector sum C = A + B
void vecAdd(float *A, float *B, float *C, int n)
{
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}

int main()
{
    // MISSING: Allocation for A, B and C
    // MISSING: I/O to read n elements of A and B
    vecAdd(A, B, C, n);
}
```

Rough equivalent in CUDA, the loop is removed. Instead,  $P$  processes execute the **kernel** (marked with **global**) in parallel on the GPU.



```
__global__ void vecAdd(float *A, float *B, float *C, int n)
{
    int i = "index of the process";
    if (i < n)
        C[i] = A[i] + B[i];
}

int main()
{
    // Missing: Allocation for A, B and C
    vecAdd<<<...P...>>>(A, B, C, n);
}
```

We *just* need to specify the number of "processes"  $P$  to execute in parallel ( $P > n$ )

# Splitting the work

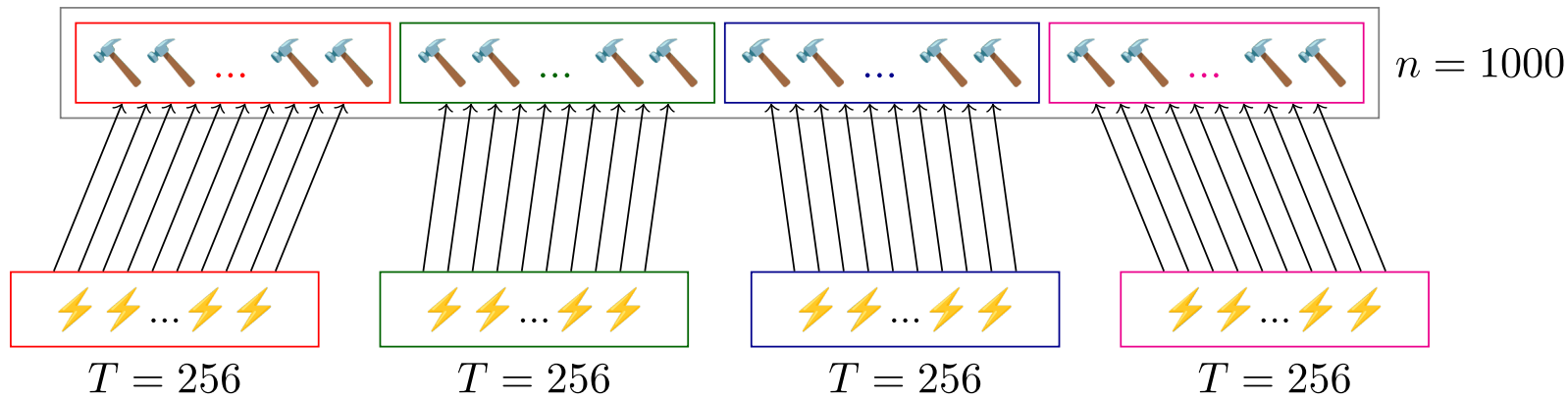
The elements to consider when writing a CUDA kernel:

-   $n$  : Amount of work (number of tasks/jobs/elements to process)
-   $P$  : Number of physical resources available to work

Problem :  $P$  depends on the device

Solution : The hierarchical decomposition of the work affected into groups (=Block) of workers (=threads)





- Group size (Block size):  $T = 256$  threads
- Number of groups (Grid size):  $B = 4$  blocks
- Total number of threads:  $T \times B = 1024$  threads  $> n$

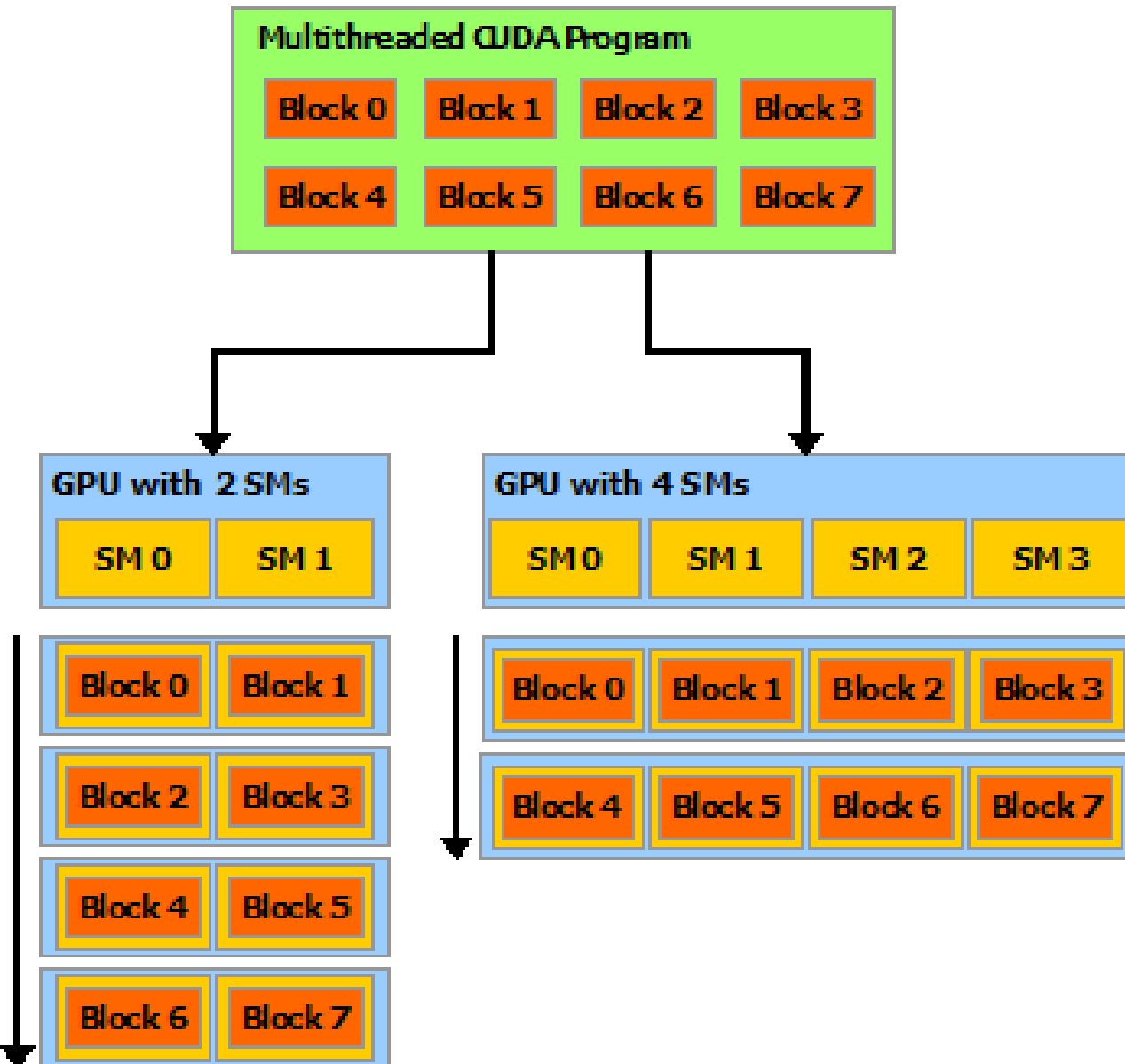
#### Note

- The number of threads per block ( $T$ ) is limited by the device (max=1024)
- The blocks are scheduled to be executed on the device in any order in parallel, **or not!**...

# Automatic scalability

Because the work is divided into **independent blocs** which can be run in **parallel** on each *streaming multiprocessor* (SM), the same code can be **automatically scaled** to architectures with more or less SMs...

as long as SMs architectures are compatibles (100% compatible with the same Compute Capabilities version --- a family of devices, careful otherwise).



```

__global__ void vecAdd(float *A, float *B, float *C, int n)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n)
        C[i] = A[i] + B[i];
}

int main()
{
    // Missing: Allocation for A, B and C
    int blockSize = 256; // threads per block
    int gridSize = FIXME; // number of blocks
    vecAdd<<<gridSize, blockSize>>>(A, B, C, n);
}

```

What is the value of `gridSize` ?

- `gridSize = (n + blockSize - 1) / blockSize`
- `gridSize = n / blockSize`
- `gridSize = [n / blockSize]`

# More work by thread

So far, 1 ⚡ = 1 🛠, what about augmenting the **work by thread**, e.g:

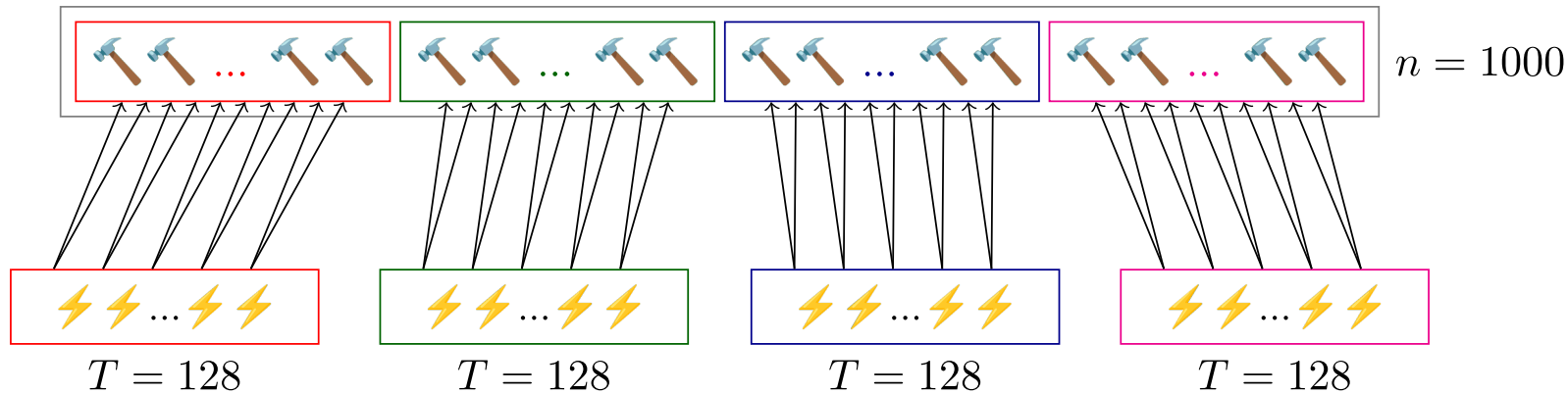
- 1 ⚡ = 2 🛠

```
__global__ void vecAdd(float *A, float *B, float *C, int n)
{
    int i = FIXME;
    int j = FIXME;
    if (i < n)
        C[i] = A[i] + B[i];
    if (j < n)
        C[j] = A[j] + B[j];
}
```

# Blocked arrangement

Threads process two consecutive elements.

```
i = blockIdx.x * blockDim.x + threadIdx.x * 2;  
j = blockIdx.x * blockDim.x + threadIdx.x * 2 + 1;
```

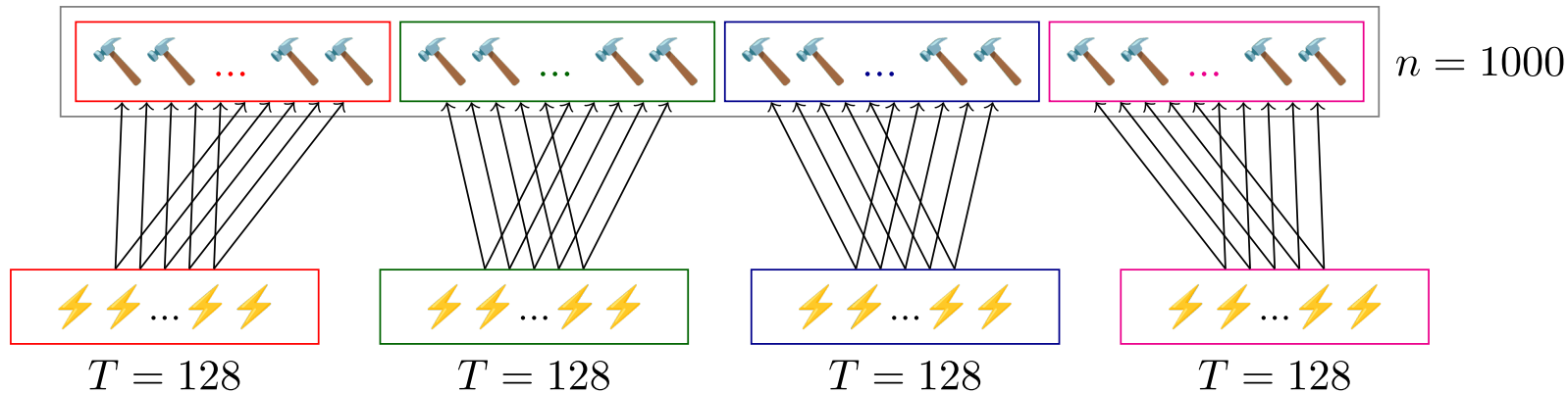



- Group size (Block size):  $T = 128$  threads
- Number of groups (Grid size):  $B = 4$  blocks
- Work by thread:  $C = 2$  ⚡
- $T \times B \times C = 1024 > n$

# Striped arrangement

Threads process two elements in "striped" fashion.

```
i = blockIdx.x * blockDim.x * 2 + threadIdx.x;  
j = blockIdx.x * blockDim.x * 2 + threadIdx.x + blockDim.x;
```



- Group size (Block size):  $T = 128$  threads
- Number of groups (Grid size):  $B = 4$  blocks
- Work by thread:  $C = 2$  
- $T \times B \times C = 1024 > n$

# Blocked vs Striped

## Note

- Blocked arrangements are desirable for algorithmic benefits (where long sequences of items can be processed sequentially within each thread).
- Striped arrangements are desirable for memory access benefits (where memory accesses can be coalesced).

We will see that in the next course 

# A multidimensional grid of threads (1/2)

Each thread uses indices (added by the compiler) to decide what data to work on:

- `blockIdx` (0 → `gridDim`): 1D, 2D or 3D
- `threadIdx` (0 → `blockDim`): 1D, 2D or 3D

Each index has `x`, `y` and `z` attributes to get the actual index in each dimension.

```
int i = threadIdx.x;  
int j = threadIdx.y;  
int k = threadIdx.z;
```

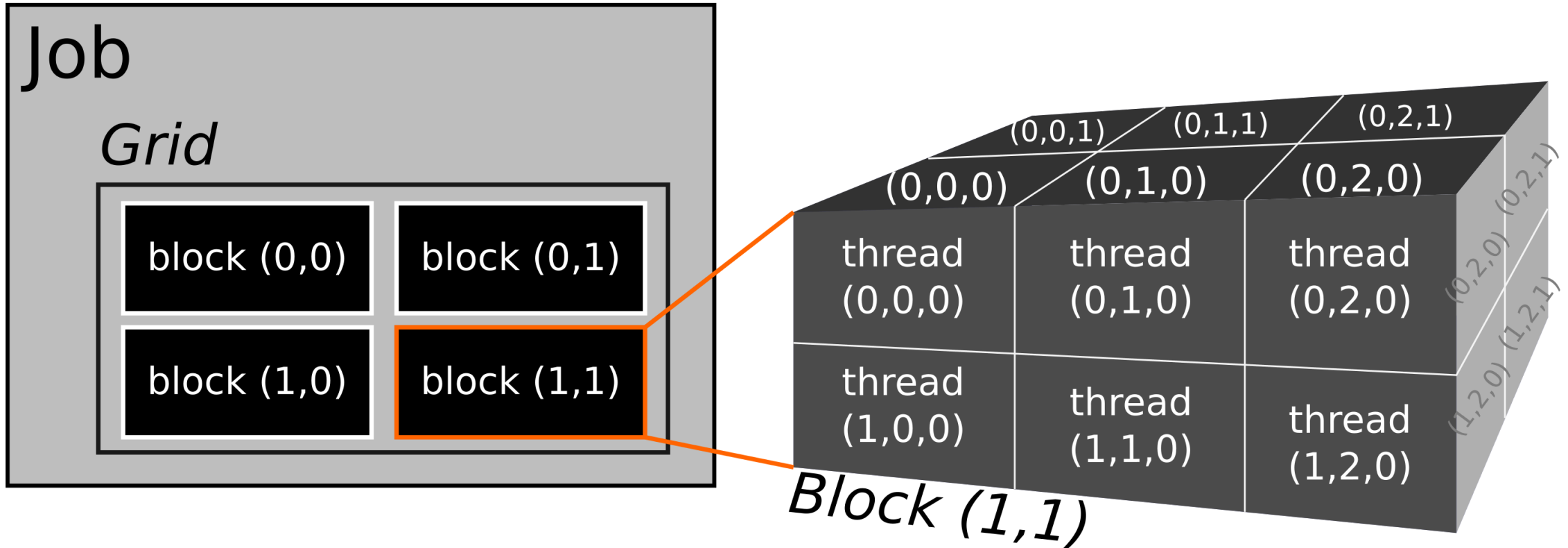
Simplifies memory addressing when processing multidimensional data:

- *image processing*
- *solving PDE on volumes*



# A multidimensional grid of computation threads (2/2)

Grid and blocks can have different dimensions, but they are usually two levels of the same work decomposition.



# VecAdd in 2D

```
__global__ void matAdd(float *A, float *B, float *C, int width, int height)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < width && j < height)
        C[j * width + i] = A[j * width + i] + B[j * width + i];
}
```

```
int main()
{
    int bx = 8, by = 8;
    dim3 blockSize(bx, by); // threads per block (blockSize)
    dim3 gridSize(..., ...); // number of blocks
    matAdd<<<gridSize, blockSize>>>(A, B, C, w, h);
}
```

What is the value of `gridSize` ?

- `gridSize = ([w / bx], [h / by]) = ((w+bx-1) / bx, (h+by-1) / by)`

# Memory Management in CUDA

```

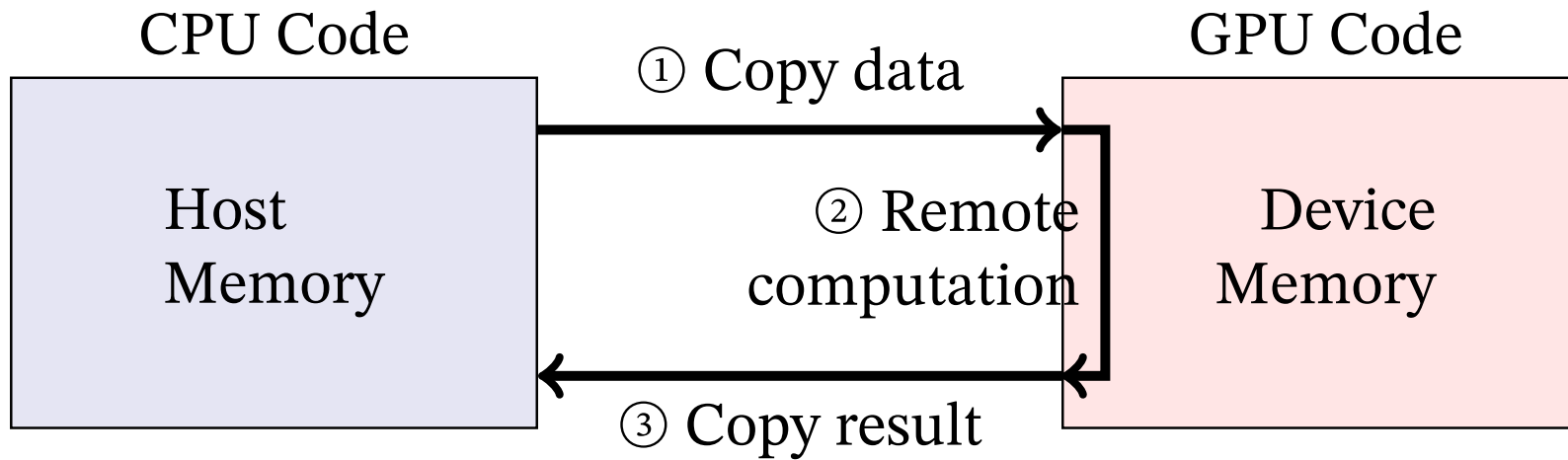
__global__ void vecAdd(float *A, float *B, float *C, int n)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n)
        C[i] = A[i] + B[i]; // ⚠️ DEVICE MEMORY ACCESS
}

int main() // HOST CODE
{
    float* A, *B, *C; // ⚠️ HOST MEMORY POINTER
    int blockSize = 256;
    int gridSize = n + blockSize - 1 / blockSize;
    vecAdd<<<gridSize, blockSize>>>(A, B, C, n);
}

```

# Separate memory spaces

- **Host memory:** accessible by the CPU
- **Device memory:** accessible by the GPU
- Need to allocate memory on the device and copy data to/from the device



```
int main() // HOST CODE
{
    float* A, *B, *C; // ⚠ HOST MEMORY POINTER
    float* d_A, *d_B, *d_C; // ⚠ DEVICE MEMORY POINTER
    cudaMalloc(&d_A, n * sizeof(float)); // Allocate device memory for A
    // Idem for B and C
    cudaMemcpy(d_A, A, n * sizeof(float), cudaMemcpyHostToDevice); // ① Copy A Host → Device
    cudaMemcpy(d_B, B, n * sizeof(float), cudaMemcpyHostToDevice); // ① Copy B Host → Device

    vecAdd<<<gridSize, blockSize>>>(A, B, C, n); // ② Launch kernel

    cudaMemcpy(C, d_C, n * sizeof(float), cudaMemcpyDeviceToHost); // ③ Copy C Device → Host
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C); // Free device memory
}
```

Host memory	Device memory
<code>ptr = malloc(nbytes)</code>	<code>cudaMalloc(&amp;ptr, nbytes)</code>
<code>free(ptr)</code>	<code>cudaFree(ptr)</code>
<code>memcpy(dst, src, nbytes)</code>	<code>cudaMemcpy(dst, src, nbytes, cudaMemcpyDefault)</code>
<code>memset(ptr, val, nbytes)</code>	<code>cudaMemset(ptr, val, nbytes)</code>

### Note

`cudaMemcpyDefault` can be replaced by `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`

# Unified Memory

Introduced in CUDA 6.0, it allows the GPU to access the host memory directly with a unified address space (transparently managed by the CUDA driver).

The transfer is done on demand, and the data is cached on the device.

```
int main() // HOST CODE
{
    float* A, *B, *C; // ⚠️ HOST MEMORY POINTER accessible by the GPU
    cudaMallocManaged(&A, n * sizeof(float)); // Allocate unified memory for A
    cudaMallocManaged(&B, n * sizeof(float)); // Idem for B
    cudaMallocManaged(&C, n * sizeof(float)); // Idem for C

    vecAdd<<<gridSize, blockSize>>>(A, B, C, n); // Launch kernel

    // No need to copy data back to the host
    // Use C as if it was a host pointer
    // ...

    cudaFree(A); cudaFree(B); cudaFree(C); // Free unified memory
}
```



# 2D and 3D variants

	1D	2D	3D
<b>Allocate</b>	<code>cudaMalloc()</code>	<code>cudaMallocPitch()</code>	<code>cudaMalloc3D()</code>
<b>Copy</b>	<code>cudaMemcpy()</code>	<code>cudaMemcpy2D()</code>	<code>cudaMemcpy3D()</code>
<b>On-device init.</b>	<code>cudaMemset()</code>	<code>cudaMemset2D()</code>	<code>cudaMemset3D()</code>
<b>Reclaim</b>	<code>cudaFree()</code>		

Why 2D and 3D variants?

- Strong alignment requirements in device memory
  - Enables correct loading of memory chunks to SM caches
- Proper striding management in automated fashion

# Intermission: *Can I use memory management functions inside kernels?*

**No:** `cudaMalloc()`, `cudaMemcpy()` and `cudaFree()` shall be called from host only.

However, kernels may allocate, use and reclaim memory dynamically using regular `malloc()`, `memset()`, `memcpy()` and `free()` functions.

Note that if some device code allocates some memory, it must free it.

## **Warning**

how many threads are going to call `malloc()` ?

# Your first kernel launch

## Code `hello.cu`

```
#include <stdio>

__global__ void print_kernel() {
    printf(
        "Hello from block %d, thread %d\n",
        blockIdx.x, threadIdx.x);
}

int main() {
    print_kernel<<<2, 3>>>();
}
```

## Compile and run

```
$ nvcc hello.cu -o hello
$ ./hello    # no output WTF 🤔
$
```

### ⚡ Danger

The `cudaDeviceSynchronize()` function is missing in the code.

It is required to ensure that the kernel has finished executing before the program exits. Kernel launches are asynchronous by default.

```
__global__ void print_kernel() {  
    printf(  
        "Hello from block %d, thread %d\n",  
        blockIdx.x, threadIdx.x);  
}  
  
int main() {  
    print_kernel<<<2, 3>>>();  
    cudaDeviceSynchronize();  
}
```

## Compile and run

```
$ nvcc hello.cu -o hello  
$ ./hello  
Hello from block 1, thread 0  
Hello from block 1, thread 1  
Hello from block 1, thread 2  
Hello from block 0, thread 0  
Hello from block 0, thread 1  
Hello from block 0, thread 2
```

# Summary

## **Host vs Device → Separate memory**

GPUs are computation units which require explicit usage, as opposed to a CPU

Need to load data to and fetch result from device (explicitly or managed by the driver)

## **Replace loops with kernels**

Kernel = Function computed in relative isolation on small chunks of data, on the GPU

## **Divide the work**

Problem → Grid → Blocks → Threads

# Cooperative work with CUDA

Sometimes, the threads have to collaborate to solve a problem.

Available tools :

- **Shared memory:** a small memory space shared by threads in the same block
- **Barriers:** a synchronization point where all threads in a block must reach before continuing
- **Atomic operations:** operations that are guaranteed to be executed without interruption



# Problem: counting non-zero values in an array

```
__global__ void count_non_zero(int *A, int n, int *result)
{
    FIXME
}

int main() {
    int n = 1000;
    int* A;
    int* result;
    cudaMallocManaged(&A, n * sizeof(int));
    cudaMallocManaged(&result, sizeof(int));
    // MISSING: Initialize A with random values
    count_non_zero<<<4, 256>>>(A, n, result);
}
```

1. Count the number of non-zero values in the block
2. Add this number to the global result

## Info

We need a shared variable between threads of the same block.

# Reminder about memory types

Memory	On chip	Cached	Access	Scope	Lifetime
Register	✓		R/W	1 thread	Thread
Local		✓	R/W	1 thread	Thread
Shared	✓		R/W	All threads in block	Block
Global		✓	R/W	All threads + host	Host allocation
Constant		✓	R	All threads + host	Host allocation

```
__device__  int global_var;    // Global variable
__constant__ int lookup_table; // Constant variable

__global__ void kernel() {
    __shared__ int shared_var; // Shared variable
    int local_var;             // Local variable (register)
    int aux_data[128];         // Local variable (maybe local)
}
```

*Where to declare variables? Can host access it?*

- Yes: **global** and **constant**  
Declare outside of any function
- No: **register** and **shared**  
Use or declare in the kernel

# Counting in shared memory

```
__global__ void count_non_zero(int *A, int n, int *result) {  
    __shared__ int count;  
    count = 0;  
  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        count += (A[i] != 0);  
  
    *result += count;  
}
```

## ⚡ Danger

Many problem in this code ! Can you spot them?

- `count` , `result` is shared between threads, but not protected
- We did not synchronize the threads, they all add the result

# Synchronization

- Memory fences: `__threadfence()`, `__threadfence_block()`, `__threadfence_system()`

Memory fences are used to ensure that memory operations are visible to all threads in the block.

- Barriers: `__syncthreads()`

Barriers are used to synchronize all threads in a block.

## ⚡ Danger

`__syncthreads()` must be called by all threads in the block, otherwise, the program will hang

```
if (...)
    __syncthreads(); // ⚠ LIKELY WRONG
```

# Counting in shared memory (2)

```
__global__ void count_non_zero(int *A, int n, int *result) {
    __shared__ int count;
    count = 0;

    __syncthreads(); // All threads must reach this point

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        count += (A[i] != 0);

    __syncthreads(); // Synchronize threads
    if (threadIdx.x == 0)
        *result += count;
}
```

## ⚡ Danger

The code is still wrong, can you spot the problem?

- `count` & `result` are shared between threads, but not protected

# Atomic operations

Atomic functions perform a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

Most of the atomic functions are available for all the numerical types : `int` , `uint` , `uint64` , `float` , `double` , `half` , etc.

Function	Description
<code>old = atomicAdd(&amp;x, v)</code>	<code>old ← x; x ← x + v;</code>
<code>old = atomicSub(&amp;x, v)</code>	<code>old ← x; x ← x - v;</code>
<code>old = atomicExch(&amp;x, v)</code>	<code>old ← x; x ← v;</code>
<code>old = atomicMin(&amp;x, v)</code>	<code>old ← x; x ← min(x, v);</code>
<code>old = atomicCAS(&amp;x, cmp, v)</code>	<code>old ← x; if (x == cmp) x ← v;</code>

# Counting with atomic operations

```
__global__ void count_non_zero(int *A, int n, int *result) {  
    __shared__ int count;  
    count = 0;  
  
    __syncthreads(); // All threads must reach this point  
  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        atomicAdd(count, A[i] != 0);  
  
    __syncthreads(); // Synchronize threads  
    if (threadIdx.x == 0)  
        atomicAdd(result, count);  
}
```



# Exercise

CUDA has vector types ( `float2` , `float3` , `float4` , `int2` , `int3` , `int4` , etc.) that can be used to store multiple values in a single variable.

Adapt the previous code so that the work by thread is doubled by processing two elements at a time.

```
__global__ void count_non_zero(int *A, int n, int *result) {
    FIXME
}

int main() {
    int n = 1000;
    int* A;
    int* result;
    cudaMallocManaged(&A, n * sizeof(int));
    cudaMallocManaged(&result, sizeof(int));
    count_non_zero<<<FIXME>>>(A, n, result);
}
```

# Host & Device functions

	Executed on the:	Only callable from the:
<code>__host__ float HostFunc()</code>	host	host
<code>__global__ void KernelFunc()</code>	device	host <sup>1</sup>
<code>__device__ float DeviceFunc()</code>	device	device

- `__global__` defines a kernel function
  - Each "`__`" consists of two underscore characters
  - A kernel function must return void
  - <sup>1</sup> It may be called from another kernel for devices of compute capability 3.2 or higher (Dynamic Parallelism support)
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

# Exercise

Adapt the previous code to get the ID of the block having the most non-zero values.

1. Wrap the function that counts the number of non-zero values in a `device` function
2. Adapt the code, so that blocks update the global result with the block ID having the most non-zero values
3. The last block store the result in the global result

```
__device__ int count_non_zero_block(int *A, int n) {  
    __shared__ int count;  
    count = 0;  
  
    __syncthreads(); // All threads must reach this point  
  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        atomicAdd(count, A[i] != 0);  
  
    __syncthreads(); // Synchronize threads  
    return count;  
}
```

```

__global__ int      gProcessedBlocks = 0; // Store the block ID
__global__ int      gCount[]; // Store the total per block

__device__ int get_max_nonzero(int *A, int n, int *result) {
    int count = count_non_zero_block(A, n, result);

    __shared__ int is_last_block;
    if (threadIdx.x == 0) { // Only one thread per block
        gCount[blockIdx.x] = count;
        __threadfence(); // Ensure that the value is written before the signal
        int old = atomicAdd(&gProcessedBlocks, 1); // Increment the number of processed blocks
        is_last_block = (old == gridDim.x - 1);
    };

    __syncthreads(); // Synchronize threads

    store_max_index_of_block(gCount, gridDim.x, result); // To implement
}

```

# Conclusion

- Cooperative work is possible but difficult to manage (requires advanced tools: shared memory, barriers, atomic operations)
- At grid level, it is even more complex (no primitives to synchronize blocks)
- The introduction of cooperative groups in [CUDA 9.0 \(CUDA Cooperative Groups\)](#) simplifies the management of groups of threads