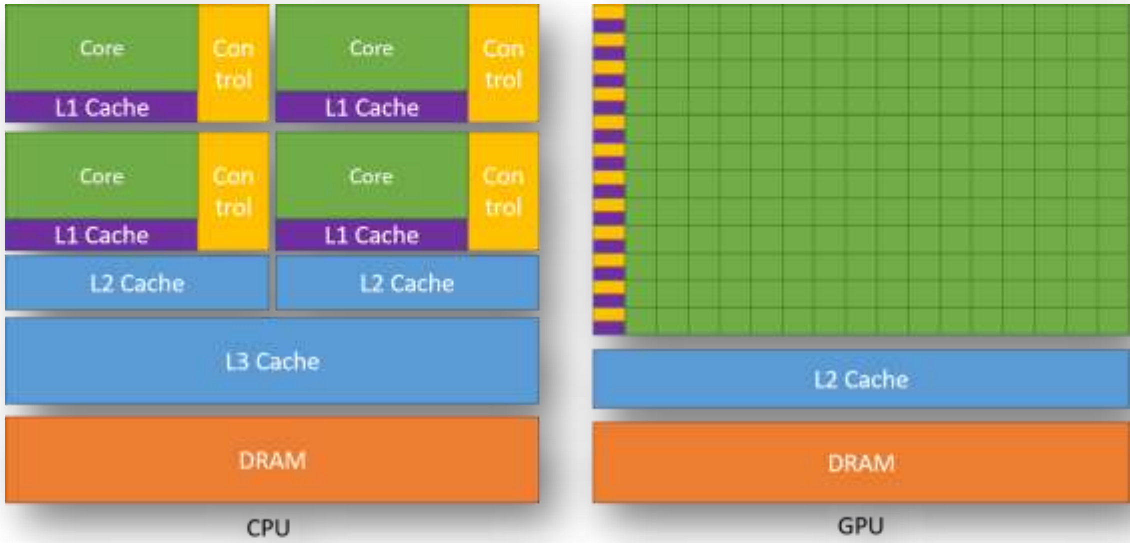
Programming Languages for Accelerators

Manos Pavlidakis[1,2]
manospavl@ics.forth.gr

[1] Institute of Computer Science, Foundation for Research and Technology - Hellas, Greece
[2] Computer Science Department, University of Crete, Greece

What is an accelerator?

- A device that performs some functions more efficiently than general-purpose CPU
  - Due to massive pallelism
  - GPUs are perfect for Vector Add
- General Purpose Graphic Processing Unit (NVIDIA, AMD)
- Field-Programmable Gate Array (Xilinx, Intel Altera)
- Application-Specific Integrated Circuit
  - TPU: Tensor Processing Unit (Google)



Programming Guide :: CUDA Toolkit Documentation (nvidia.com)

How we program accelerators?

Well known programming languages

- CUDA → NVIDIA GPUs

- HIP:
  - ROcM → AMD GPUs
  - CUDA → NVIDIA GPUs

- oneAPI/SYCL → Heterogeneous accelerators

CUDA: Compute Unified Device Architecture

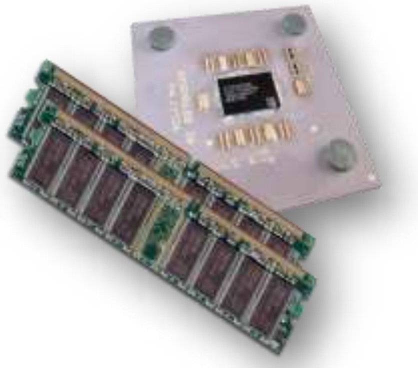
What is CUDA?

- CUDA
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming → Kernel code
  - APIs to manage devices → Transfers, Allocations etc.

Terminology

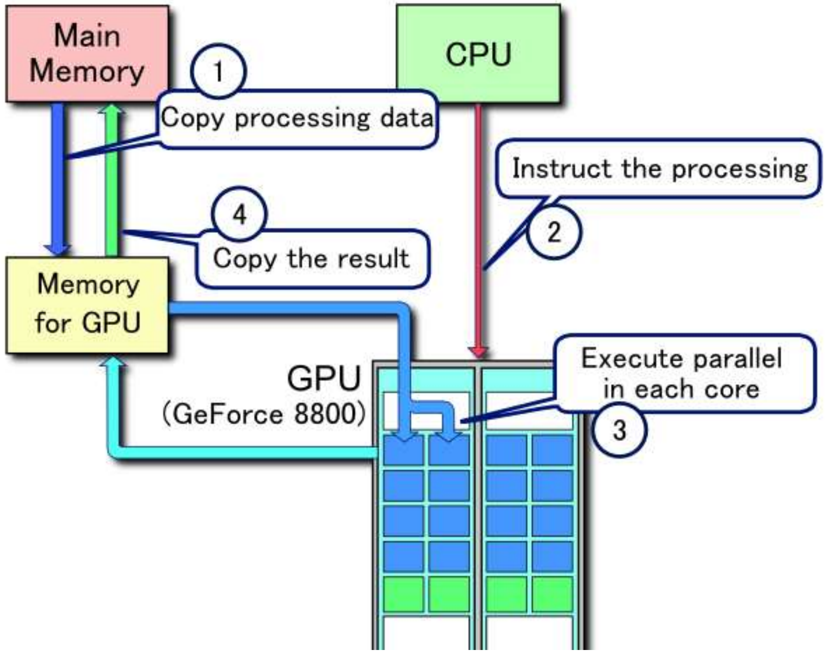
- Host: The CPU and its memory
- Device: The GPU/Accelerator and its memory

Host



Device

Host and Device code



Host code

Device code

(CUDA kernel)

```
#define THREADS_PER_BLOCK 256
const int N=2048;

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
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space on host
    a = (int*)malloc(size); random_ints(a, N);
    b = (int*)malloc(size); random_ints(b, N);
    c = (int*)malloc(size);

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

Executed by one **Host** Thread

Executed by multiple **CUDA** Threads

CUDA by an example

Add two integers with CUDA

- 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

Add two integers with CUDA

- 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 for a, b, c

Memory Management

- Host and device memory are separate entities
- Device pointers point to GPU memory
  - May be passed to/from **host** code
- Host pointers point to CPU memory
  - May be passed to/from **device** code
- Simple CUDA API for handling device memory
  - cudaMalloc(), cudaFree(), cudaMemcpy()
  - Similar to the C equivalents malloc(), free(), memcpy()

Add two integers: main()

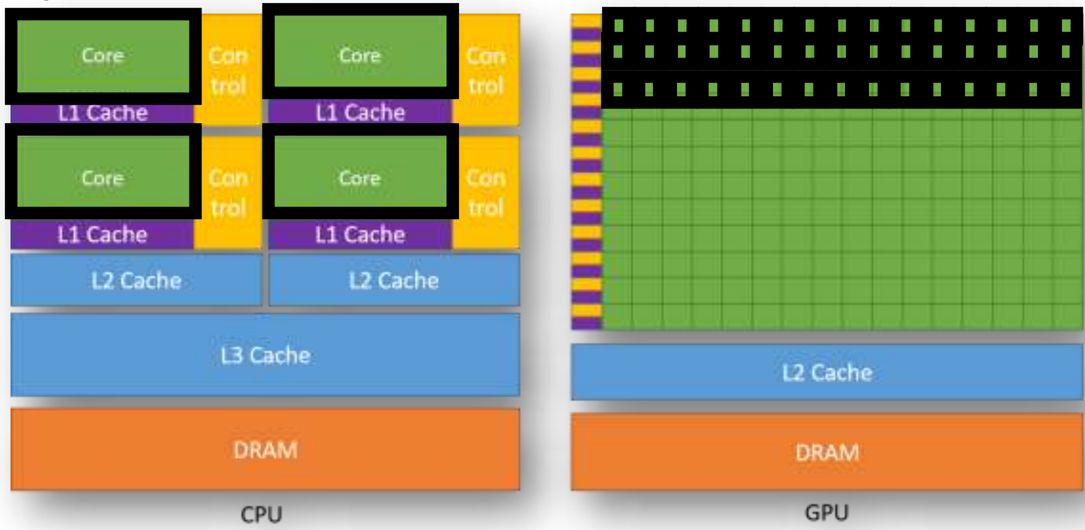
```
int main(void){
        int a, b, c;                    // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = sizeof(int);
        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void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

Add two integers: 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 results back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
} // End of main()
```

Use parallelism

- Performance gain of GPUs is based on massive parallelism
  - CPUs have several cores (i.e. hundreds)
  - GPUs have many cores (i.e. thousands)



- How we run the add() kernel on many cores?

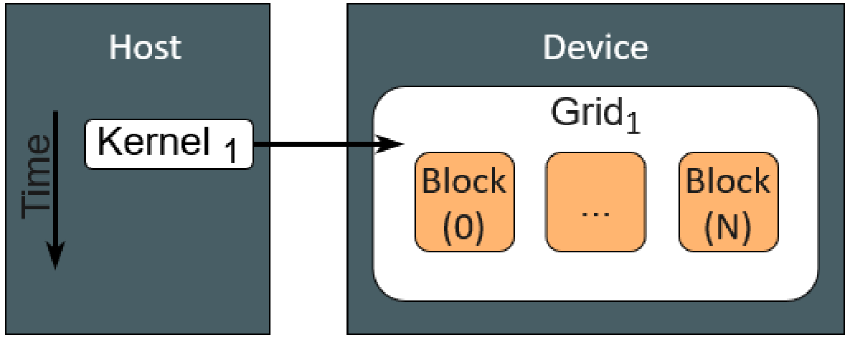
  add<<< 1, 1 >>>();

  ⬇

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

- Instead of executing add() once, execute N times in parallel!

Vector addition on the Device



- With add() running in parallel we can do vector addition
- Each parallel invocation of add() is referred to as a Block
- The set of Blocks is referred to as a Grid
- Each invocation can refer to its block index using blockIdx.x

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

- By using blockIdx.x to index into the array, each block handles a different index

Sequential Vector Addition using C++

```
int main(void) {
        int a, b, c;
        int size = sizeof(int);
        // Allocate and initialize a, b, c
        a = (int*)malloc(size); random_ints(a, N);
        b = (int*)malloc(size); random_ints(b, N);
        c = (int*)malloc(size);
        for(int i =0; i<size; i++) // Add arrays
                c[i] = a[i] + b[i];
        free(a); free(b); free(c);
        return 0;
}
```

Vector addition with 4 blocks

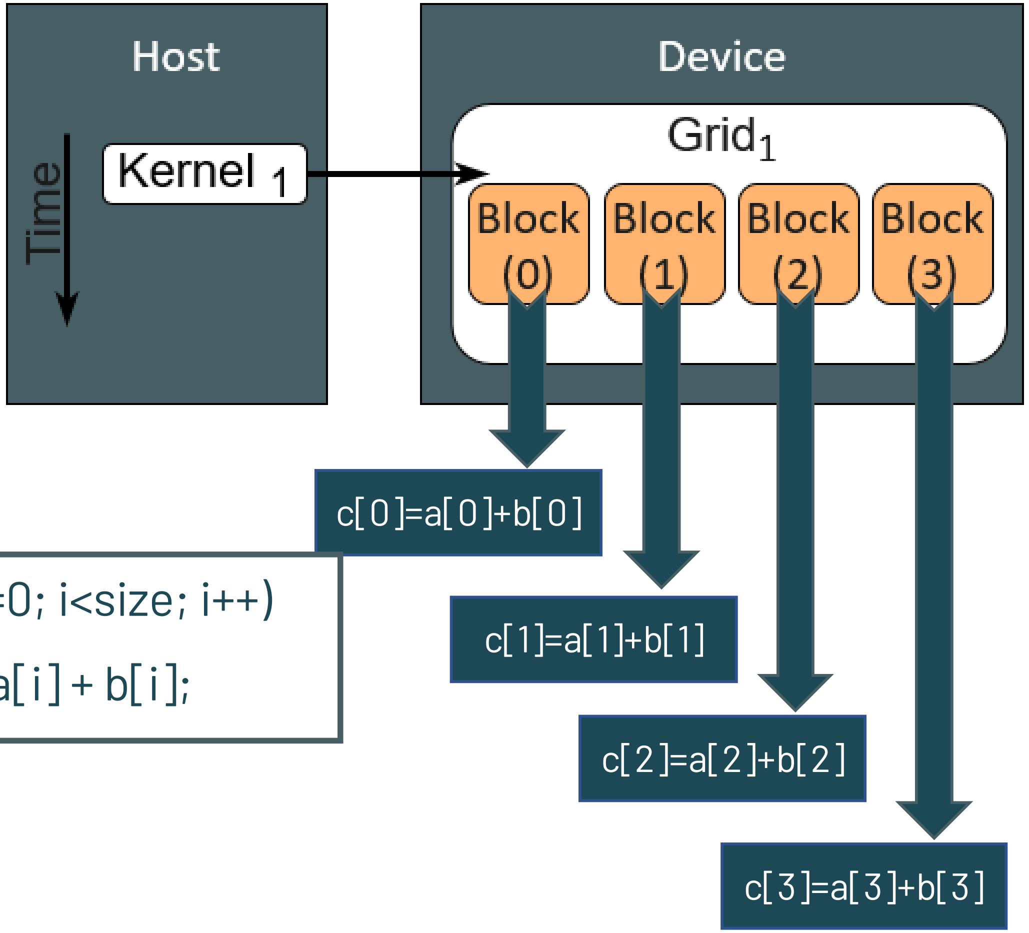
- If we want to create 4 parallel blocks
- We will call the kernel from host with N=4

```
add<<< 4, 1 >>>(...);
```

- On the device the kernel code

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

```
for (int i =0; i<size; i++)
    c[i] = a[i] + b[i];
```



Vector addition main()

```
#define N 4
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 random input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

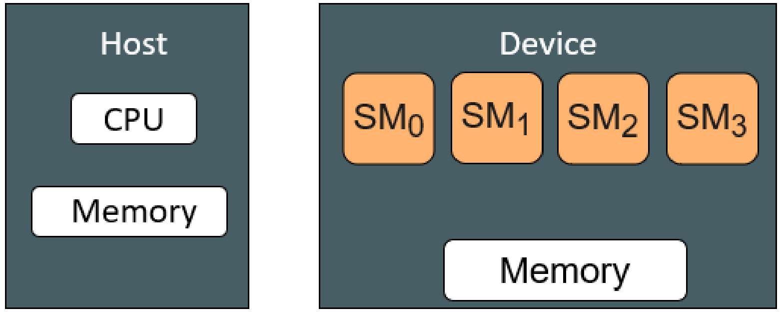
Vector addition main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);
    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
    // Free device and host memory
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

More parallelism using CUDA cores

- NVIDIA GPUs consists of:
  - Streaming Multiprocessors (SM)
  - An SM consists of CUDA cores

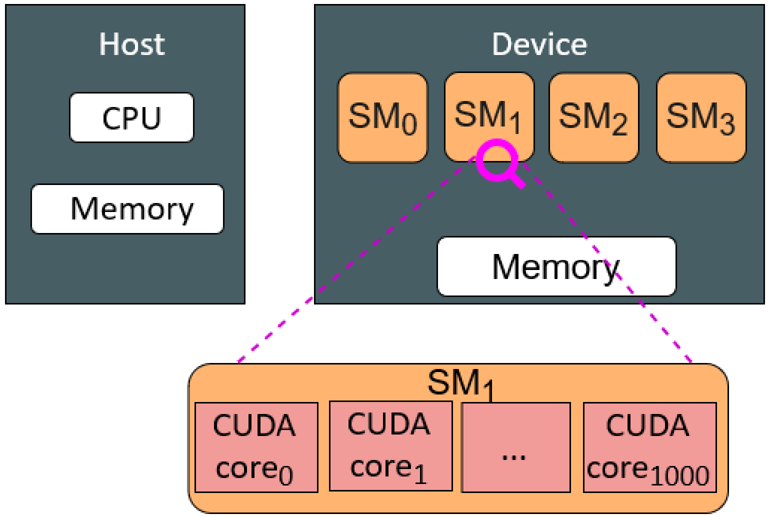
Hardware



More parallelism using CUDA cores

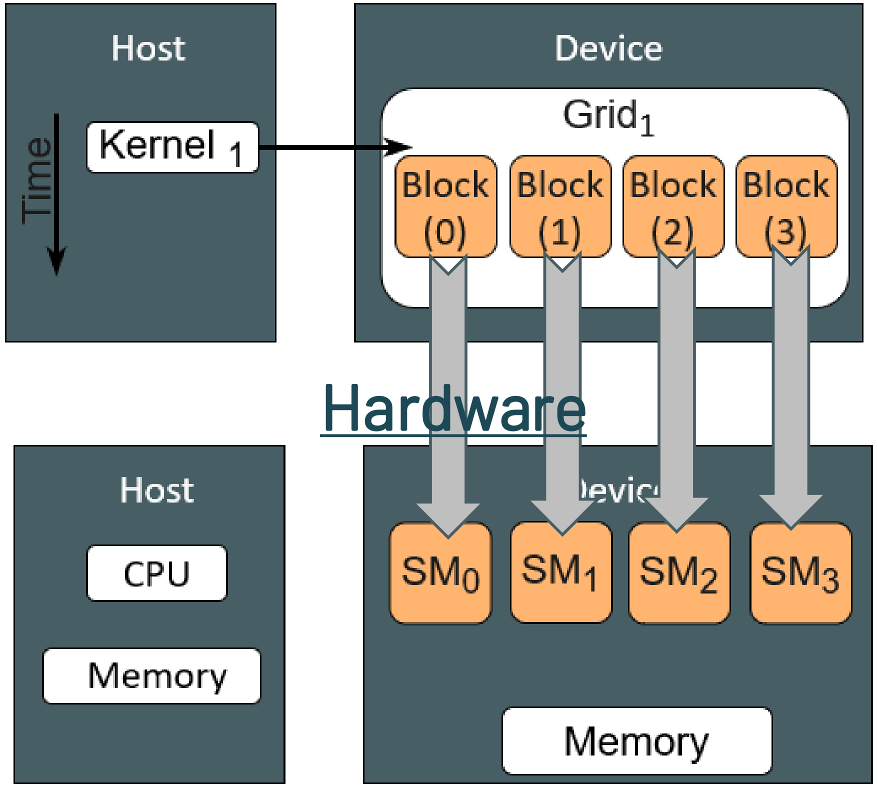
- NVIDIA GPUs consists of:
  - Streaming Multiprocessors (SM)
  - An SM consists of CUDA cores
  - A CUDA core is like a **thin** processor
- A GPU has:
  - Hundreds of SMs (e.g. A100 GPU has **128**)
  - Thousand CUDA cores (e.g. A100 GPU has **7000**)

Hardware



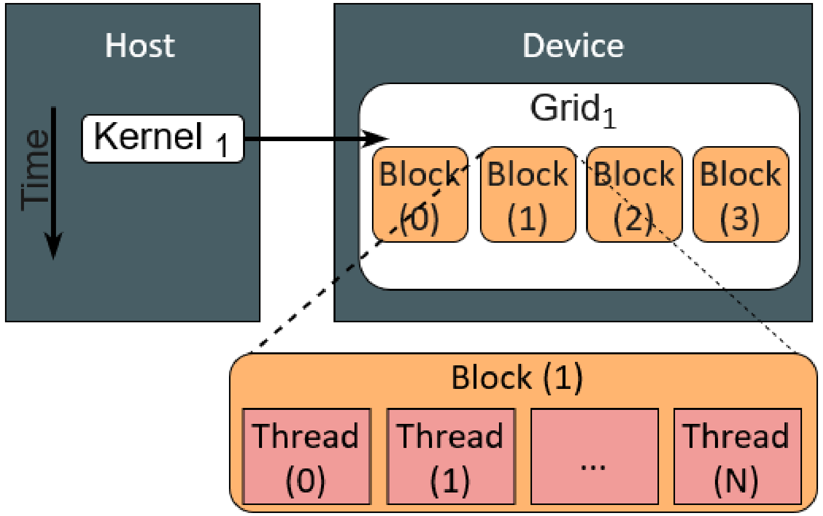
More parallelism using CUDA cores

- NVIDIA GPUs consists of:
  - Streaming Multiprocessors (SM)
  - An SM consists of CUDA cores
  - A CUDA core is like a **thin** processor
- A GPU has:
  - Hundreds of SMs (e.g. A100 GPU has **128)**
  - Thousand CUDA cores (e.g. A100 GPU has **7000)**
- In our example we use only Blocks
  - A Block is assigned to an SM
  - Blocks in the same SM execute concurrently!
    - Not in parallel!!
    - If we have 4xSMs only 4xBlocks run in parallel
- Why not use CUDA cores !



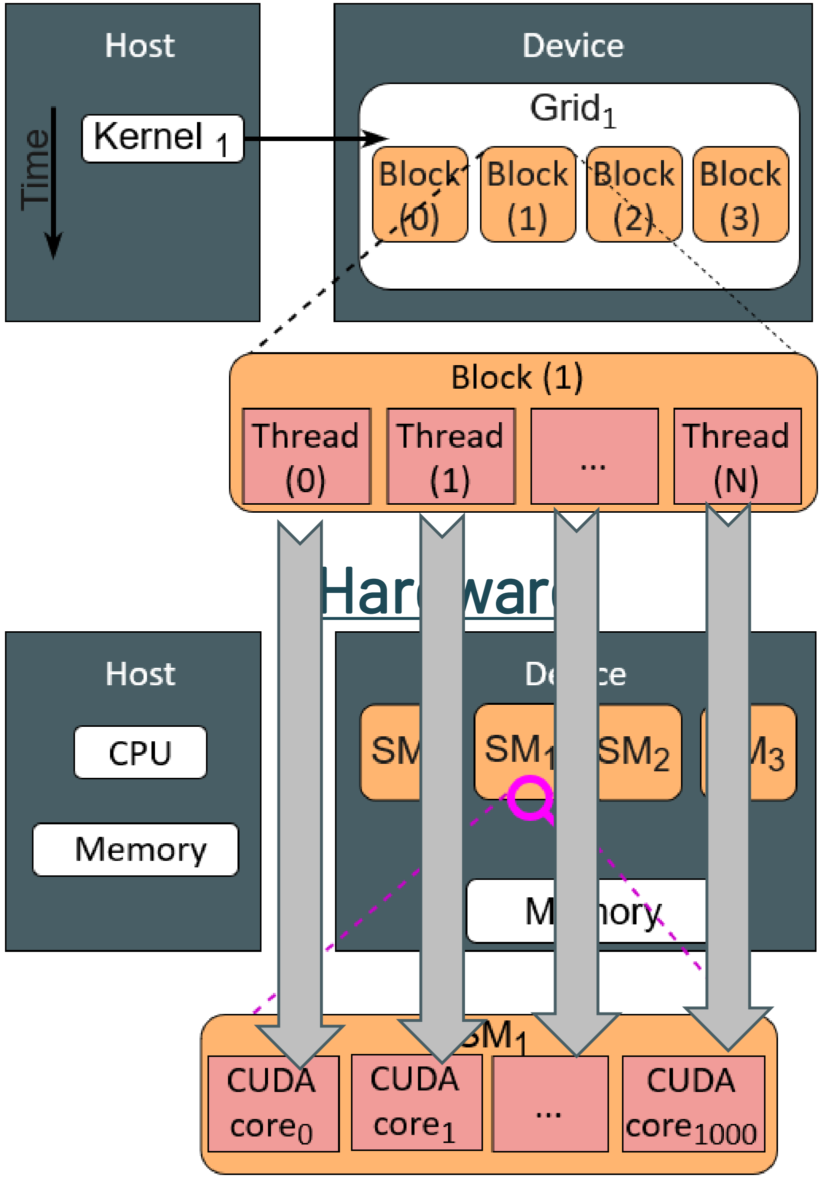
CUDA Threads → CUDA cores

- A Block can be split into parallel Threads
  - Threads in the same block can cooperate
  - Threads have unique ids (i.e. threadId.x)



CUDA Threads → CUDA cores

- A Block can be split into parallel Threads
  - Threads in the same block can cooperate
  - Threads have unique ids (i.e. threadId.x)
- Threads are assigned to CUDA cores



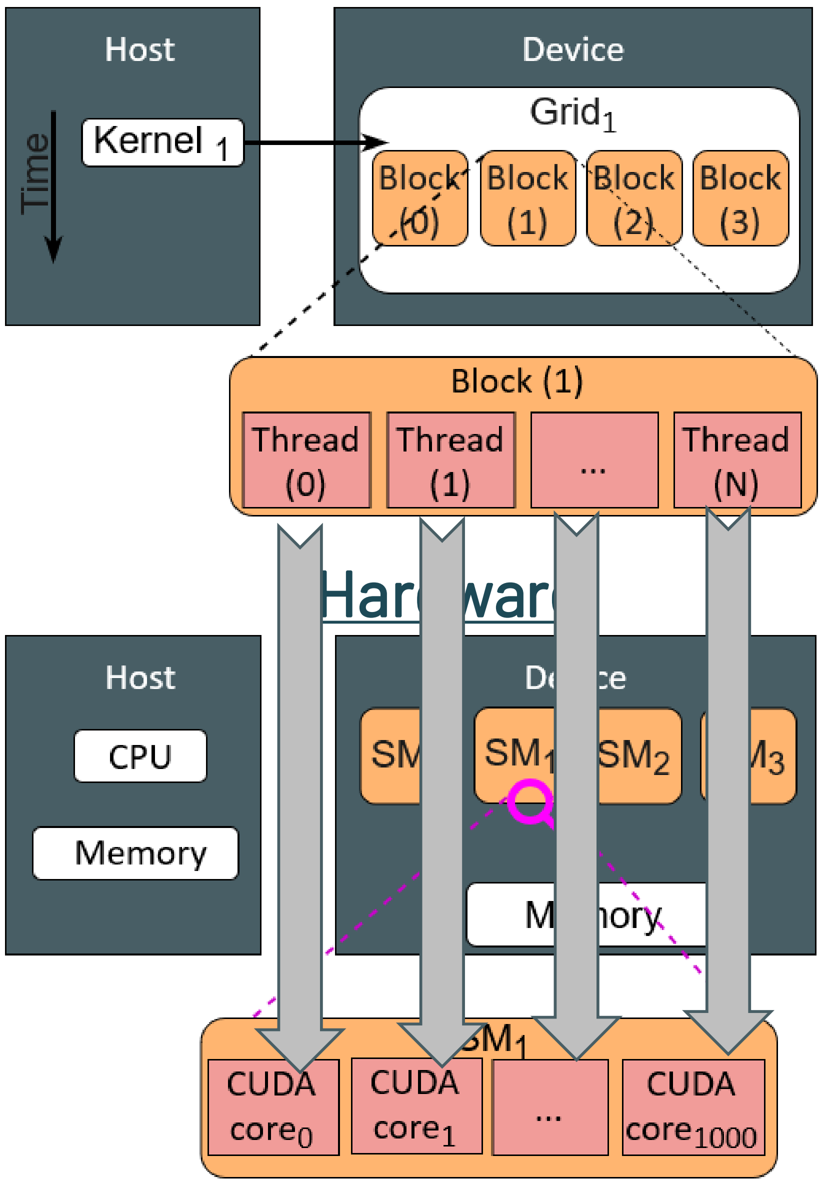
CUDA Threads → CUDA cores

- A Block can be split into parallel Threads
  - Threads in the same block can cooperate
  - Threads have unique ids (i.e. threadId.x)
- Threads are assigned to CUDA cores
- We have to modify kernel code to use Threads, instead of blocks:

```
__global__ void add(int *a, int *b, int *c) {
        c[threadId.x] = a[threadIdx.x] + b[threadIdx.x]; }
```

- We use the threadId.x instead of blockIdx.x
- In main we have to change the kernel call:

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



CUDA Threads → CUDA cores

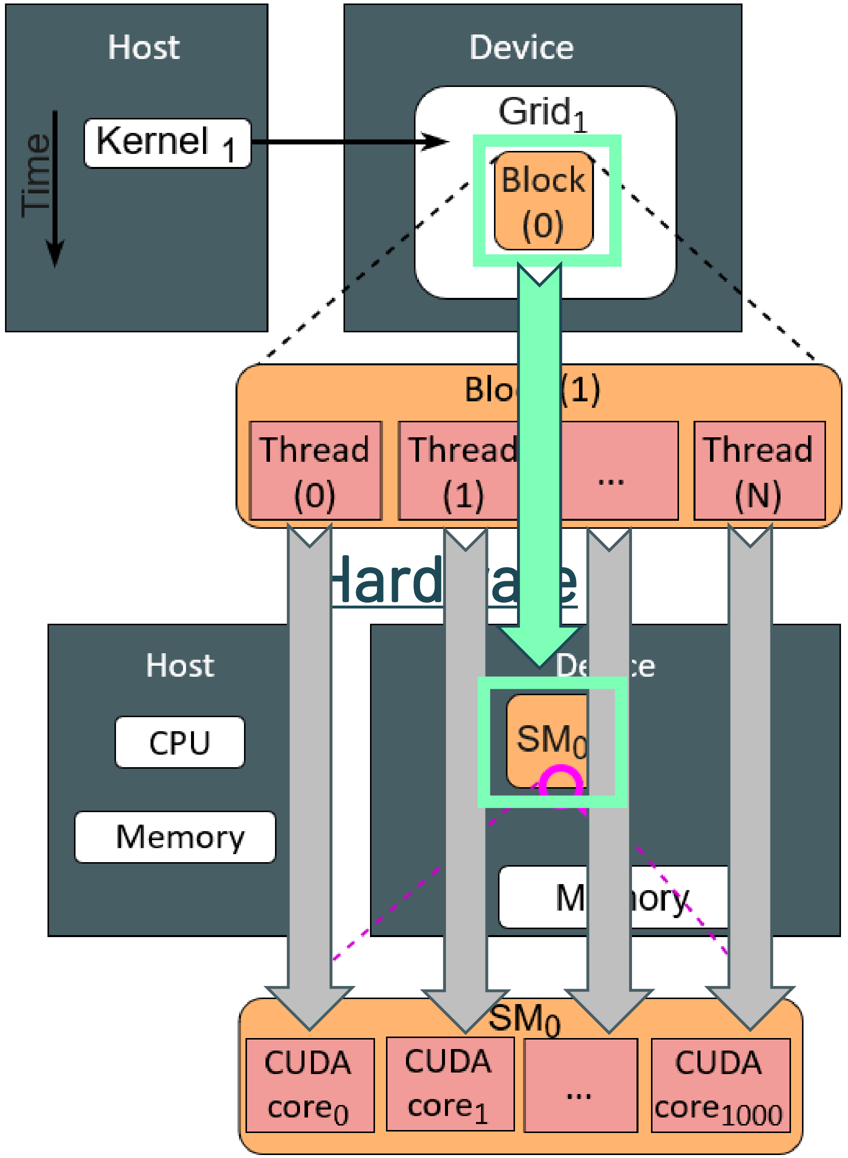
- A Block can be split into parallel Threads
  - Threads in the same block can cooperate
  - Threads have unique ids (i.e. threadId.x)
- Threads are assigned to CUDA cores
- We have to modify kernel code to use Threads, instead of blocks:

```
__global__ void add(int *a, int *b, int *c){
        c[threadId.x] = a[threadIdx.x] + b[threadIdx.x]; }
```

- We use the threadId.x **instead of** blockIdx.x
- In main we have to change the kernel call:

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

- Now we call the kernel with **1xBlock** and NxThreads

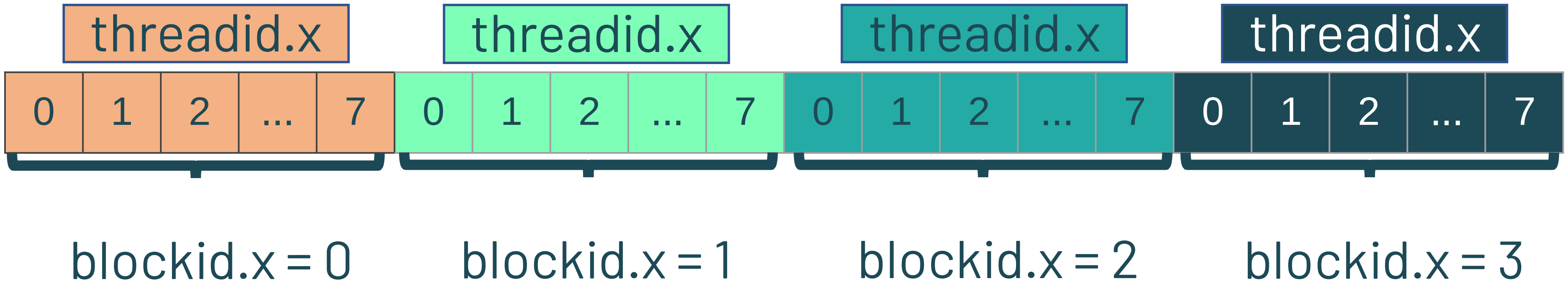


Combining Threads & Blocks

- Until now we have seen parallel vector addition using
  - Many Blocks with **one Thread** each
  - One **Block** with many **Threads**
- Now let's get more parallelism (= performance) by using
  - Many Blocks with many Threads
- But before let's discuss data indexing

Indexing Arrays with Blocks & threads

- Consider indexing an array with one element per thread (8 threads/block)



- With **M threads per block**, a unique index for each thread is given by:
  - int index = threadIdx.x + blockIdx.x * M;
  - M = Number of threads per block or block size
  - Use the built-in variable blockDim.x for threads per block
  - int index = threadIdx.x + blockIdx.x * blockDim.x;

Vector addition with Blocks & Threads

Host code

```
#define THREADS_PER_BLOCK 256
const int N=2048;

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
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc spzce on host
    // 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);

    // 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;
}
```

Block size: 256

Array size: 2048

add <<< 8, 256 >>>(...);

Kernel code

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

8 x Blocks, 256 x Threads/Block = **2048**

Handling arbitrary vector sizes

- Typical problems are not friendly multiples of blockDim.x (e.g. N = 5000)

```
#define THREADS_PER_BLOCK 256
const int N=5000;

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);

    ...

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK + 1,THREADS_PER_BLOCK>>>

    ...

    return 0;
}

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

Array size: 5000

Update kernel launch

Avoid accessing beyond the end of arrays

Compile a CUDA program

- nvcc is the CUDA compiler
- With CUDA both Host and Device code can be in the same file
  - With suffix ".cu"
- nvcc separates source code into host and device components
  - Device functions (e.g. add()) processed by NVIDIA compiler
  - Host functions (e.g. main()) processed by standard host compiler (g++/gcc)
- Compile vector addition example:
  - nvcc vectorAdd.cu –o vectorAdd
- Run:
  - ./vectorAdd

Thank you

Questions?

Manos Pavlidakis
manospavl@ics.forth.gr

Find the example in: https://github.com/manospavlidakis/VectorAdditionCUDA.git

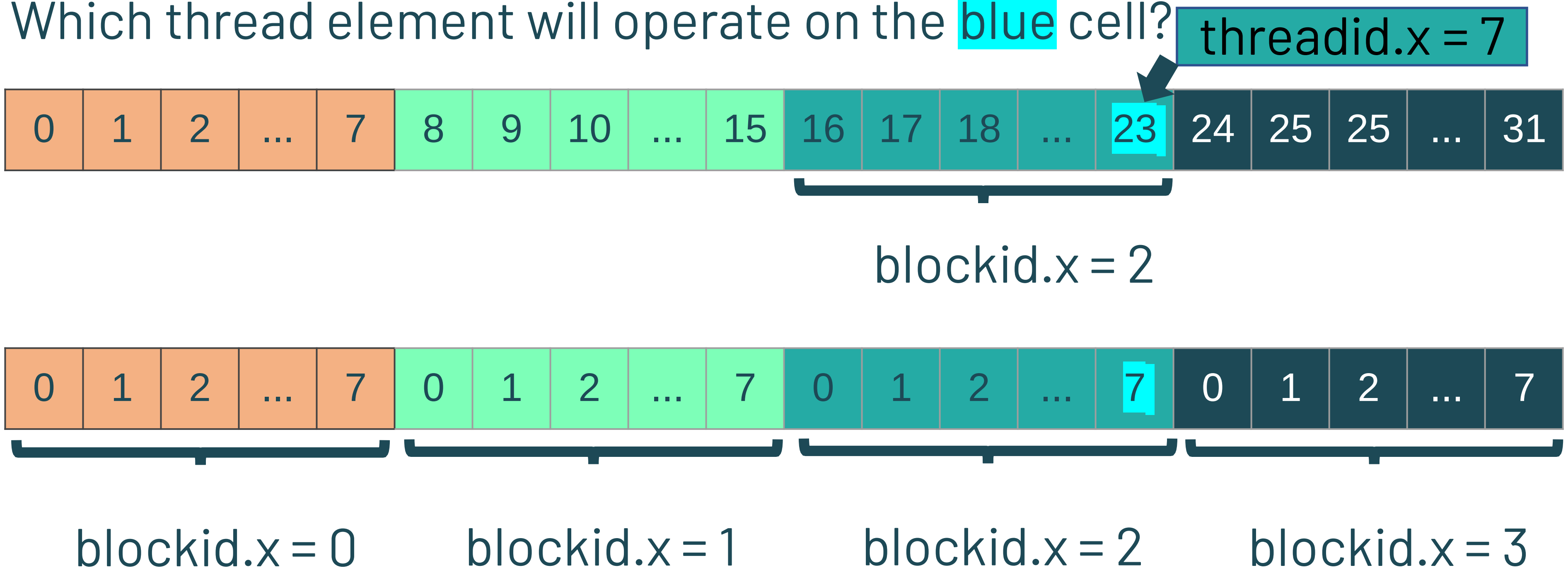
A simple example

- Which thread element will operate on the blue cell?

0	1	2	...	7	8	9	10	...	15	16	17	18	...	23	24	25	25	...	31

A simple example

- Which thread element will operate on the blue cell?



- With **M threads per block = 8**
  - int index = threadIdx.x + blockIdx.x * M = 7 + 2 * 8 = 23;