VUB VRIJE UNIVERSITEIT BRUSSEL ULB UNIVERSITÉ LIBRE DE BRUXELLES

GPU Computing

»Lesson 2: Programming GPUs - part I

Gauthier Lafruit & Jan Lemeire

2022-2023

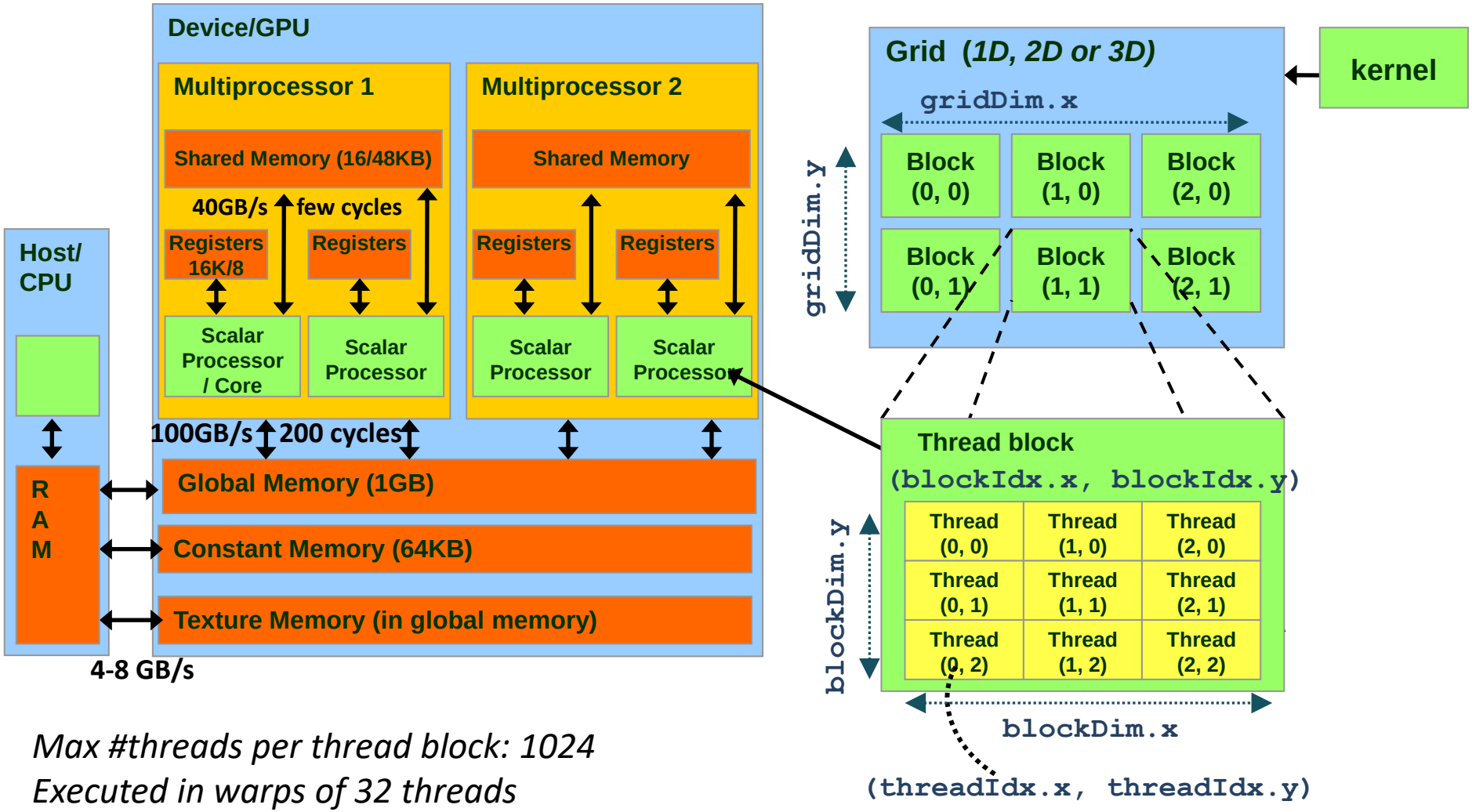
http://parallel.vub.ac.be/education/gpu

VRIJE UNIVERSITEIT BRUSSEL

Levels of Understanding

- Level 0
  - Host code
- Level 1
  - Parallel execution on the device
- Level 2 *=> explained later*
  - Device model and thread blocks
- Level 3 *=> explained later*
  - Hardware threads & SIMT

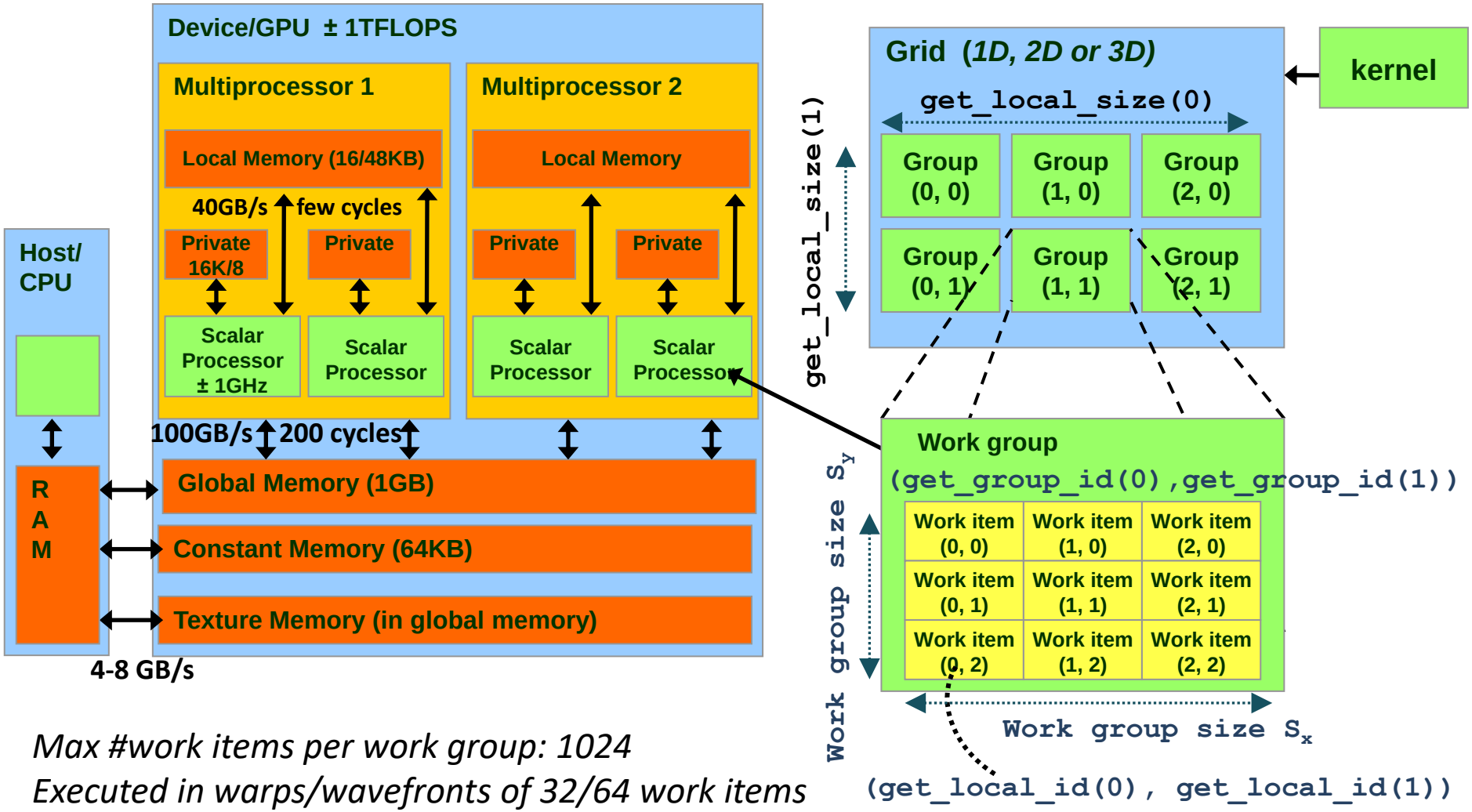
GPU Concepts



Max #threads per thread block: 1024
Executed in warps of 32 threads
Max thread blocks simultaneously on MP: 8
Max active warps on MP: 24/48

CUDA terminology

GPU Concepts



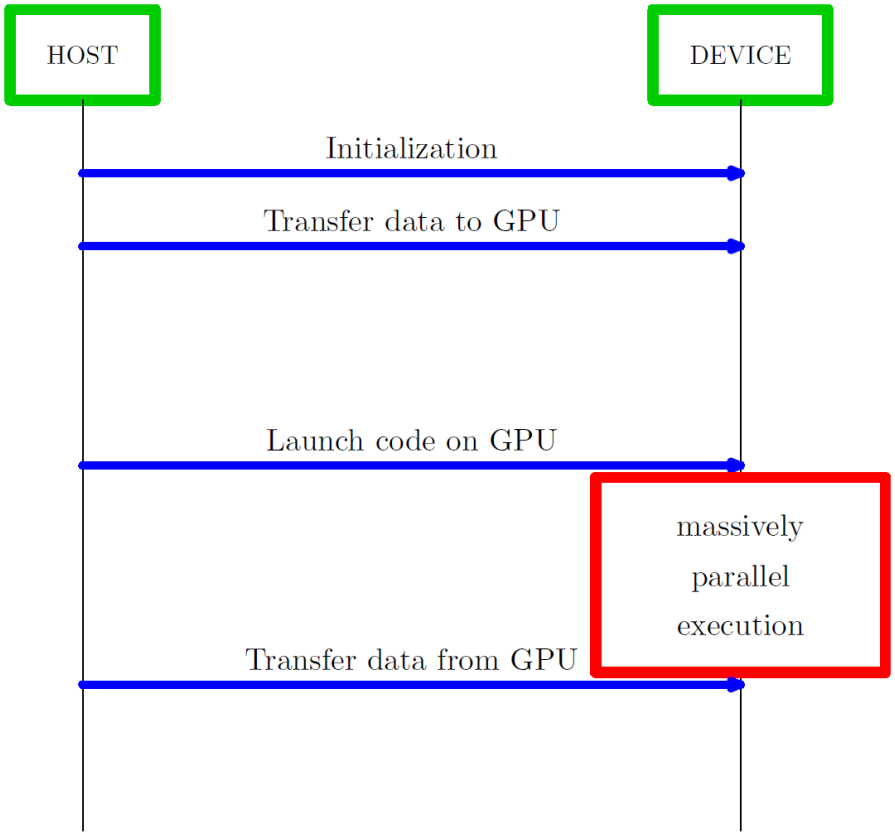
Max #work items per work group: 1024
Executed in warps/wavefronts of 32/64 work items
Max work groups simultaneously on MP: 8
Max active warps/wavefronts on MP: 24/48

OpenCL terminology

Level 0
Host Code

VRIJE UNIVERSITEIT BRUSSEL

Typical Sequence of Events



CUDA Working Group

Will be used in this course

OpenCL Working Group

- **Diverse industry participation**
  - Processor vendors, system OEMs, middleware vendors, application developers
- **Many industry-leading experts involved in OpenCL's design**
  - A healthy diversity of industry perspectives
- **Apple initially proposed and is very active in the working group**
  - Serving as specification editor
- **Here are some of the other companies in the OpenCL working group**

VRIJE UNIVERSITEIT BRUSSEL

OpenCL Resources

A small sample

- www.khronos.org
- www.iwocl.org (*)
- www.streamcomputing.eu (*)
- developer.amd.com/tools-and-sdks/opencl-zone/
- www.eriksmistad.no/category/opencl/
- www.youtube.com
  - AJ Guillon

(*) These sites include references to books

VRIJE UNIVERSITEIT BRUSSEL

CUDA

- We need a way to
  - Modify our program to use accelerators
  - Specify the code that needs to run on the accelerators

- CUDA
  - A host API: functions starting with prefix ‘cuda’
  - CUDA C language
  - A model of a GPU

VRIJE UNIVERSITEIT BRUSSEL

HOST API

- We need only a little knowledge:
  1. Select the appropriate GPU.
  2. Allocate memory on the GPU.
  3. Transfer data between CPU and GPU.
  4. Compile and run code for/on the GPU.
- Understand what has to be modified.

VRIJE UNIVERSITEIT BRUSSEL

CUDA is almost C

- CUDA is very similar to C, we only add some keywords:
  - Type qualifiers:
    - __global__ to declare a function to be executed on the GPU
    - __device__ to declare a function to be called by another function on the GPU
    - __shared__ to declare shared memory in a kernel
    - __local__ [see doc]
    - __constant__ [see doc]
  - Keywords: threadIdx, blockIdx, gridDim, blockDim to access elements in a kernel
  - Intrinsics:
    - __syncthreads(); // to synchronize threads between warps (within a block) and in all the warps (thus all threads..)
    - atomicAdd(); // to synchronize blocks (see atomic ops: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions)
    - dim2, dim3, uint2, uint3 // units for cuda : dim2 hello(16,32);
  - Runtime API: to allocate memory on the GPU and transfer data, check errors, etc.. [doc]
  - Kernel functions to launch code to the GPU from the CPU: <<<X,Y,Z>>>(.) (special syntax detailed later)

deviceProperties.major

1a. Print the CUDA-enabled GPUs

```
int nDevices;

  cudaGetDeviceCount(&nDevices);
  for (int i = 0; i < nDevices; i++) {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, i);
    printf("Device Number: %d\n", i);
    printf("  Device name: %s\n", prop.name);
    printf("  Compute Capability: %d.%d\n", prop.major, prop.minor);
    printf("  Memory Clock Rate (KHz): %d\n", prop.memoryClockRate);
    printf("  Peak Memory Bandwidth (GB/s): %f\n\n",
           2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
  }
```

VRIJE UNIVERSITEIT BRUSSEL

1b. Select the appropriate GPU

```
int devicesCount;
cudaGetDeviceCount(&devicesCount);
string desiredDeviceName = "GTX XYZ";
for(int deviceIndex = 0; deviceIndex < devicesCount; ++deviceIndex)
{
        cudaDeviceProp deviceProperties;
        cudaGetDeviceProperties(&deviceProperties, deviceIndex);
        if (deviceProperties.name == desiredDeviceName)
        {
                cudaSetDevice(deviceIndex);
                break;
        }
}
```

VRIJE UNIVERSITEIT BRUSSEL

2. Allocate memory on the GPU

- Explicit memory allocation returns pointers to GPU memory
  - cudaMalloc(), cudaFree()

VRIJE UNIVERSITEIT BRUSSEL

3. Transfer data between CPU and GPU

- Explicit memory copy for host ↔ device, device ↔ device
- cudaMemcpy(), cudaMemcpy2D(), ...

VRIJE UNIVERSITEIT BRUSSEL

CPU-GPU Memory Management

- As we are going to deal with pointers to the CPU and to the GPU main memory, it is useful to start every pointer name by h_ if it is a pointer to the host's memory (CPU) or d_ if it is a pointer to the GPU's memory

```
int N = 16;
int num_bytes = N*sizeof(int);
int *d_a, *h_a = 0;

h_a = (int*) malloc(num_bytes);
cudaMalloc ( (void**) &d_a, num_bytes);
cudaMemset( d_a, 0, num_bytes);
cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
free(h_a);
cudaFree(d_a);
```

We often use flags such as:
cudaMemcpyDeviceToHost or cudaMemcpyHostToDevice

VRIJE UNIVERSITEIT BRUSSEL

4. Compile and run code for/on the GPU

- Declare a kernel (details discussed in the next section):
  - put __global__ in the beginning of the prototype of a function!
  - The Nvidia compiler (nvcc) will know what to do

```
__global__ void mySuperKernel(float * A, float * B, float *C) {
    …
}
```

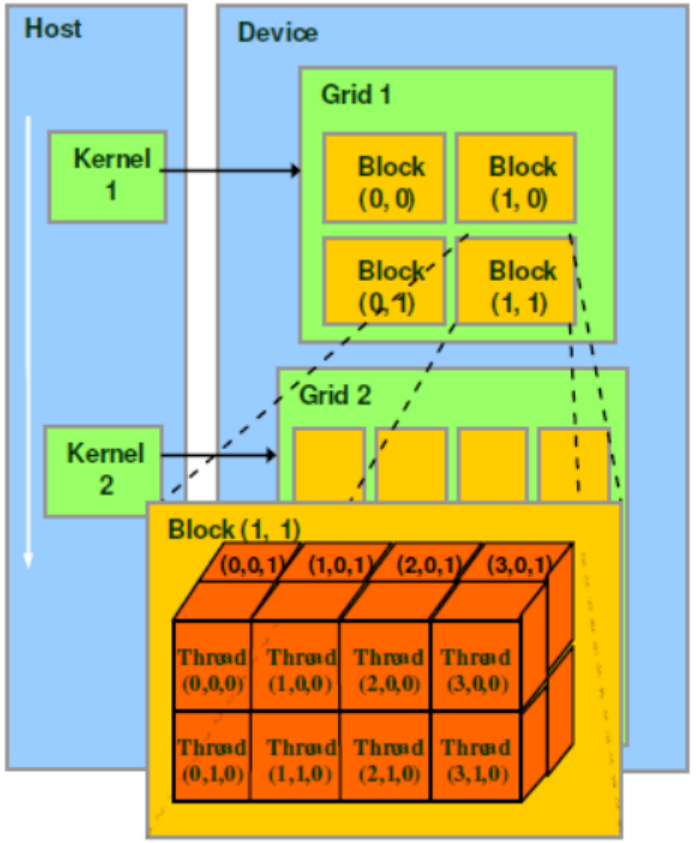
- Start a GPU program:

```
mySuperKernel <<<num_blocks, num_threads>>>(params…)
```

  - Call a kernel function
  - Provide parameters (scalars and pointers to GPU memory)
  - Provide the number of threads through the GRID dimensions
    - With the special operator <<< >>>

Grid and Blocks dimensions

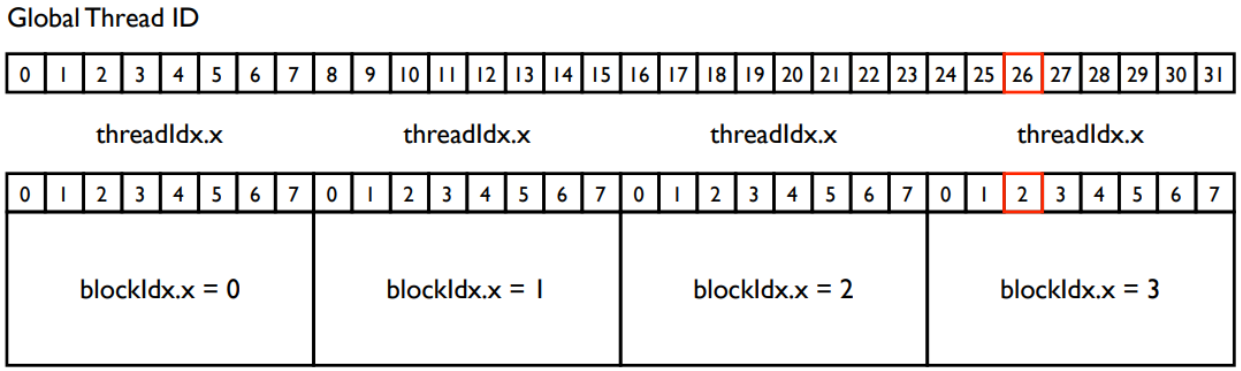
- Threads are ordered in a hierarchy:
  - A Grid consists of blocks, a blocks
- A lot of problems in science are not expressed in 1D but are rather natural in 2 or 3 dimensions.
  - Images work well in 2D
  - Medical scans work well in 3D
- Examples:
  - mySuperKernel <<<1, 1>>>(A, B, C); // executes 1 block composed of 1 thread
  - mySuperKernel<<<B, 1>>>(A, B, C); // executes B blocks composed of 1 thread
  - mySuperKernel<<<B, M>>>(A, B, C); // Executes B blocks composed of M threads each.



VRIJE UNIVERSITEIT BRUSSEL

Indexing

- CUDA Built-In Variables
  - blockIdx.{x,y,z} block ID in the {x,y,z}-axis of the block
  - threadIdx.{x,y,z} thread ID in the {x,y,z}-axis of the thread
  - blockDim.{x,y,z} block dimensions
  - threadDim.{x,y,z} thread dimensions
- The full global thread ID in x dimension can be computed by
  - globalThreadID = blockIdx.x * blockDim.x + threadIdx.x;



VRIJE UNIVERSITEIT BRUSSEL

More on kernel invocation

- Kernel_function<<<num_blocks, num_threads>>>(params…)
  - Use variables for num_blocks and num_threads, you are going to tune them.
  - num_threads is the number of threads in each block we want
    - The number of threads in a block is limited: 512, 1024, …
  - Parameters can be passed via registers or constant memory
    - If passed by registers: eg: 128 threads + 3 parameters : 128 x 3 = 384 registers
      - A GPU has a lot of registers, at least 8192 per SM. So potentially you could use 8192/128=64 registers per thread. But it is a bad idea: prefer using multiple blocks (otherwise the SM are going to be idle as soon as you access the memory)
  - The number of threads in a block should ALWAYS be a multiple of the warp size (32, 64, …)

Level 1
Parallel Execution on the Device

VRIJE UNIVERSITEIT BRUSSEL

Example

- Implement a vector addition
  - Assume three lists A, B and C
  - Element i of C:
    - $C_i = A_i + B_i$;
  - Each thread processes one data item
- Extension:
  - One work item processes more than one data item

VRIJE UNIVERSITEIT BRUSSEL

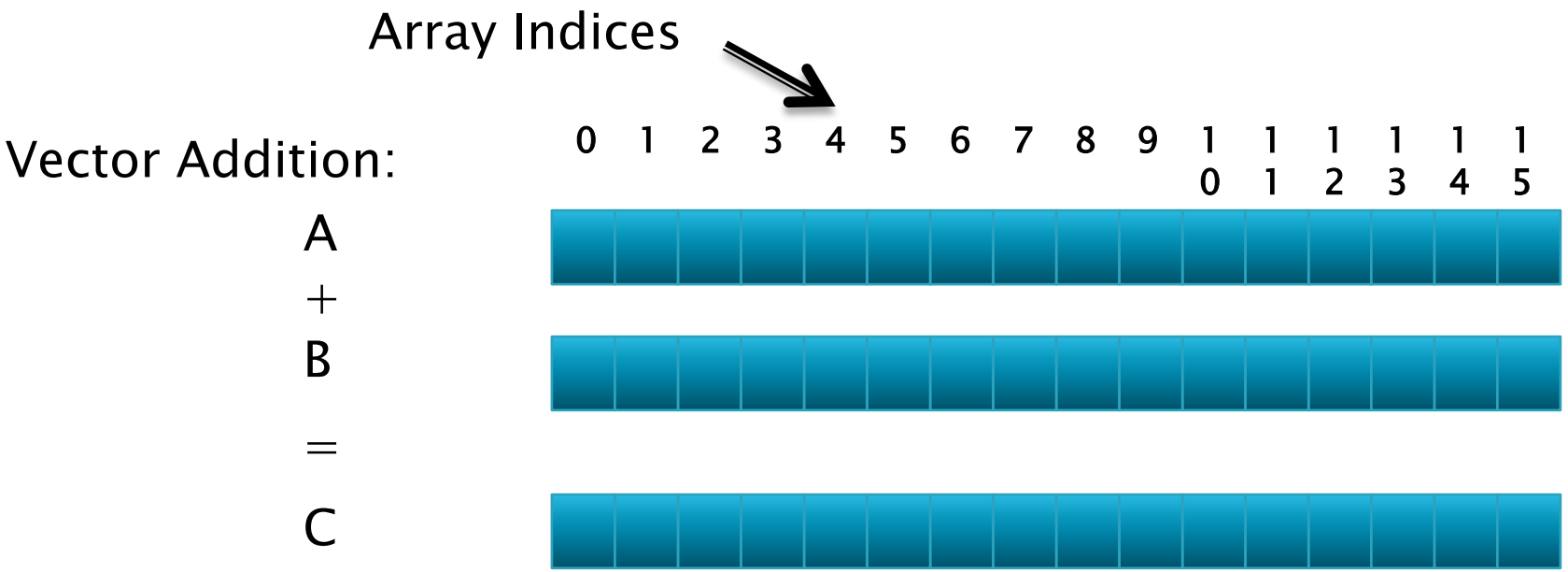
Threads executes kernel

- Massively parallel programs are usually written so that each kernel thread computes one part of a problem
  - For vector addition, we will add corresponding elements from two arrays, so each thread will perform one addition
  - If we think about the thread structure visually, the threads will usually be arranged in the same shape as the data

VRIJE UNIVERSITEIT BRUSSEL

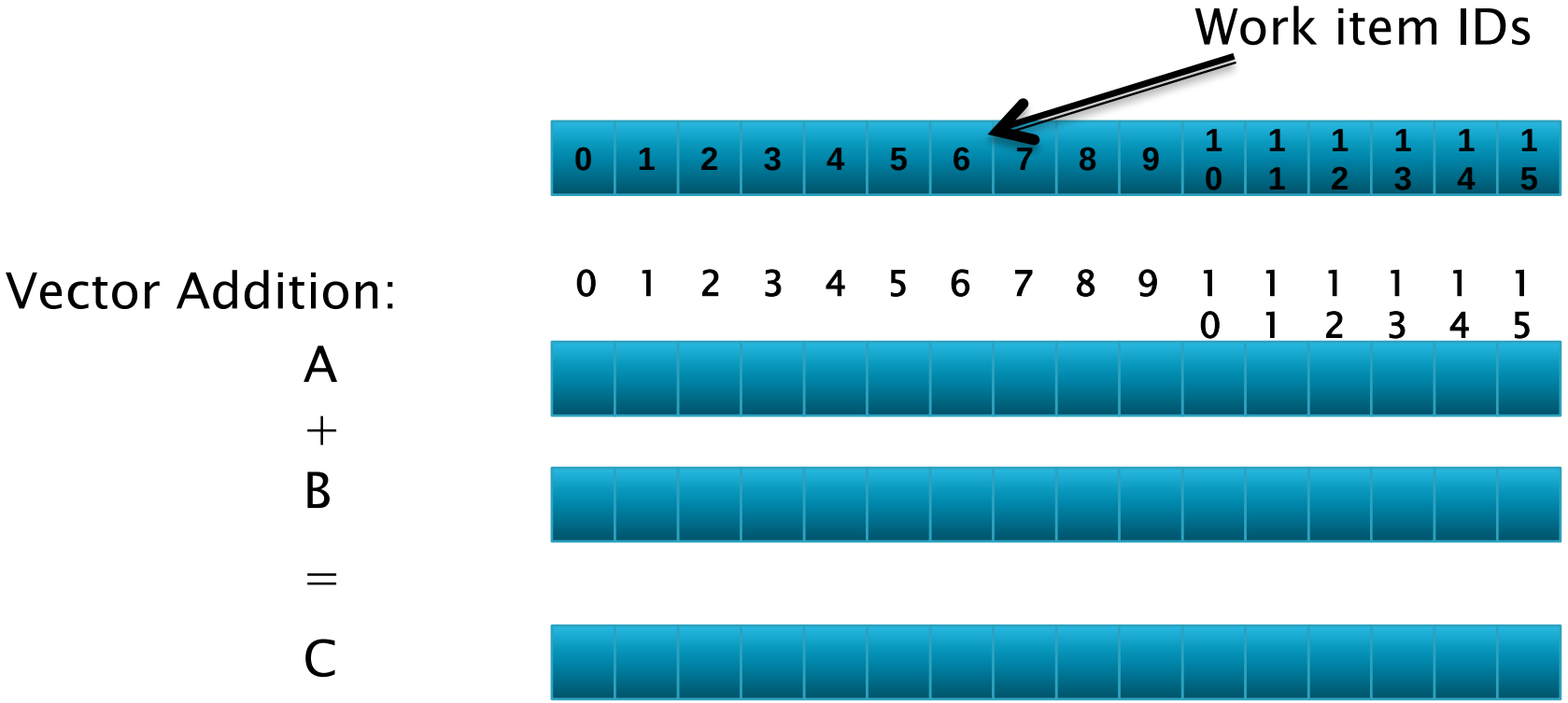
Vector addition

- Consider a simple vector addition of 16 elements
  - 2 input buffers (A, B) and 1 output buffer (C) are required



Vector addition

- Create thread structure to match the problem
  - 1-dimensional problem in this case



Vector addition

- Each thread is responsible for adding the indices corresponding to its ID

Vector Addition:

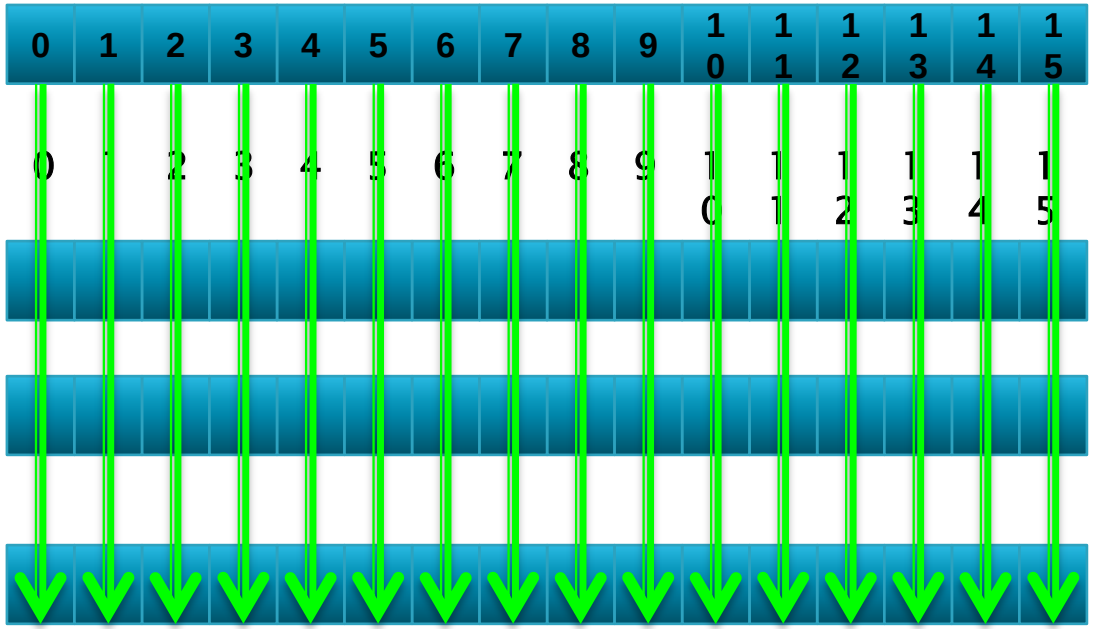
A

+

B

=

C



A kernel defines the code for a thread

VRIJE UNIVERSITEIT BRUSSEL

CUDA Kernel code

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

- __global__ to declare a function to be executed on the GPU
- Variables `blockDim`, `blockIdx`, `threadDim` and `threadIdx` to determine position of thread in the grid
  - Type: dim3 struct with fields x, y and z

VRIJE UNIVERSITEIT BRUSSEL

Host (CPU) code

```
// allocate host (CPU) memory
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
… initalize h_A and h_B …
// allocate device (GPU) memory
float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),cudaMemcpyHostToDevice));
cudaMemcpy( d_B, h_B, N * sizeof(float),cudaMemcpyHostToDevice));
// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
// copy result array back to host
float* h_C = (float*) malloc(N * sizeof(float));
cudaMemcpy( h_C, d_C, N * sizeof(float),cudaMemcpyDeviceToHost));
```

VRIJE UNIVERSITEIT BRUSSEL

Multi-dimensional Grid

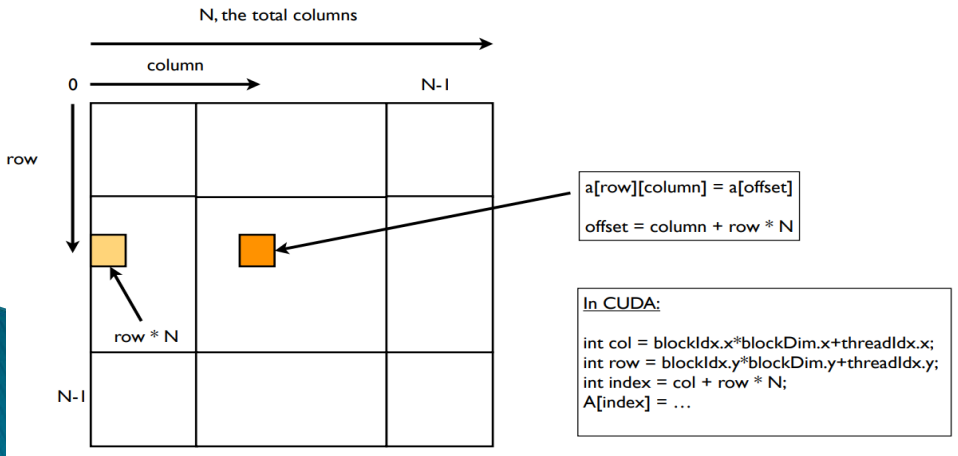
- Grid of blocks in 2 dimensions
- Block of threads in 3 dimensions

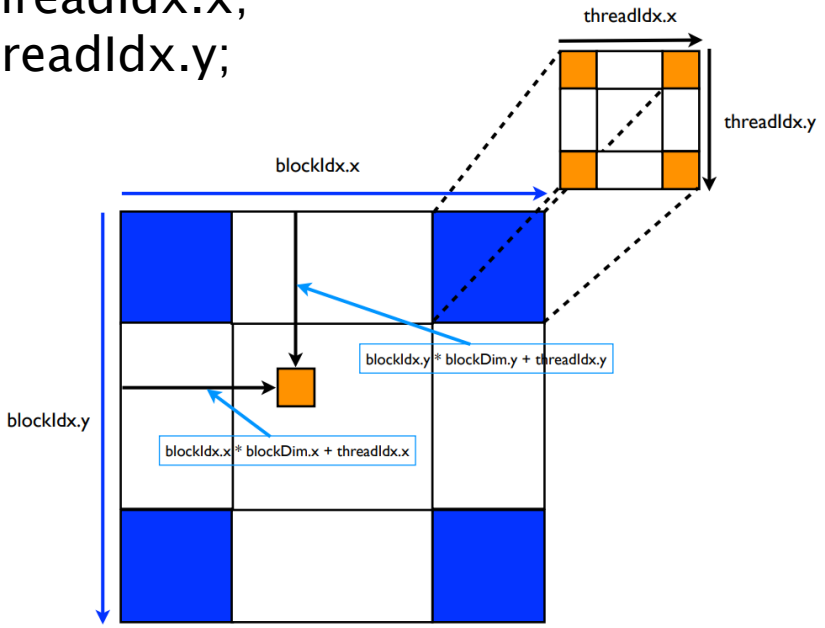
```
__global__ void KernelFunc(...);

main()
{
    dim3 DimGrid(100, 50); // 5000 thread blocks
    dim3 DimBlock(4, 8, 8); // 256 threads per block
    KernelFunc<<< DimGrid, DimBlock>>>(...);
}
```

Indexing – Higher dimensions

- Number of blocks in 2D grid: gridDim.x * gridDim.y
- Number of threads in 3D block: blockDim.x * blockDim.y * blockDim.z
- Example of initialization:
  - dim3 grid(16,16); // grid = 16x16 blocks
    dim3 block(32,32); // block = 32x32 threads
    myKernel<<<grid, block>>>(…);
  - x = blockIdx.x * blockDim.x + threadIdx.x;
    y = blockIdx.y * blockDim.y + threadIdx.y;





VRIJE UNIVERSITEIT BRUSSEL

Matrix Multiplication

- 2D GRID

```
__global__ void MatrixMulKernel(float* A, float* B, float* C, int Width)
{
        int row = blockIdx.y*blockDim.y + threadIdx.y;
        int col = blockIdx.x*blockDim.x + threadIdx.x;

        // each thread computes one element of the result matrix
        float sumOfProducts = 0;
        for (int k = 0; k < Width; ++k)
                sumOfProducts += A[row*Width+k] * B[k*Width+col];
        C[row*Width+col] = sumOfProducts;
}
```