

# Accelerate scientific applications using GPGPU computing

Alessio De Rango  
Department of Environmental Engineering  
University of Calabria, Arcavacata, 87036 Rende (CS), Italy  
Email: [alessio.derango@unical.it](mailto:alessio.derango@unical.it)



UNIVERSITÀ  
DELLA  
CALABRIA

DIPARTIMENTO  
DI INGEGNERIA  
DELL'AMBIENTE

**DIAM**



UNIVERSITÀ  
DELLA  
CALABRIA

DIPARTIMENTO  
DI MATEMATICA  
E INFORMATICA

**DeMaCS**

# Just some information about myself

I hold a PhD in Mathematics and  
Computer Science

I am currently a researcher at the  
Department of Environmental  
Engineering

My main research fields are:

- High-performance computing

- Modeling and simulation with

- Cellular Automata

- Artificial Intelligence applied to  
environmental science



UNIVERSITÀ  
DELLA  
CALABRIA

DIPARTIMENTO  
DI INGEGNERIA  
DELL'AMBIENTE

DIAM



UNIVERSITÀ  
DELLA  
CALABRIA

DIPARTIMENTO  
DI MATEMATICA  
E INFORMATICA

DeMaCS



## About the Course

This slides come from the **course Massively Parallel Programming on GPUs** arranged by Prof. D'Ambrosio and I would like thank you to him.

The course illustrates the fundamental concepts and algorithms of **Massively Parallel Programming on Graphics Processing Units (GPUs)** by the [CUDA](#) technologies.



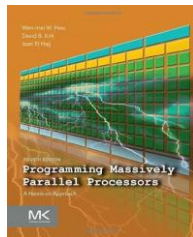
**CUDA** is used to perform General-Purpose computation on GPUs (GPGPU).

# Course resources

## Reference Book

Wen-mei W. Hwu, David B. Kirk, Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach - 4th Edition*. Morgan Kaufmann – Elsevier, 2022.

- Chapter 1. Introduction
- Chapter 2. Heterogeneous data parallel computing
- Chapter 3. Multidimensional grids and data
- Chapter 4. Compute architecture and scheduling
- Chapter 5. Memory architecture and data locality
- Chapter 6. Performance considerations
- Chapter 7. Convolution: An introduction to constant memory and caching
- Chapter 8. Stencil
- Chapter 9. Parallel histogram (Atomic operations)
- Chapter 10. Reduction
- Chapter 20. Programming a heterogeneous computing cluster: An introduction to CUDA streams



## CUDA Programming Guide

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> (html)

[https://docs.nvidia.com/cuda/pdf/CUDA\\_C\\_Programming\\_Guide.pdf](https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf) (pdf)

## CUDA Documentation (including how to install CUDA)

<https://docs.nvidia.com/cuda/>

Further resources will be made available on this git repo [here](#).



# The Boada Workstation

- Students can use the workstation to practice CUDA, develop the project for the exam, and for performance assessment.
- The following accounts were created for you: `siri1001`, `siri1002`, ...
- Access the workstation as (\$ is the shell prompt):
- ```
$ ssh siri{1001|1002|...}@boada.ac.upc.edu
```

There you will find the initial password for your account.

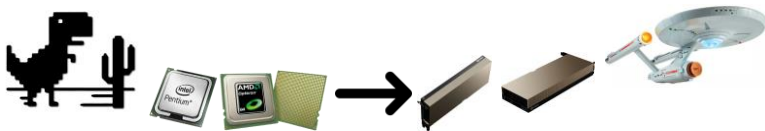
Please, change it at the first access. Do not annotate the new password in the shared document!

# Heterogeneous Parallel Computing

## Heterogeneous Parallel Computing

## A bit of history

- Old **single-core CPUs** were able to reach few GFLOPS<sup>2</sup> to the desktop and TFLOPS<sup>3</sup> to high-performance computers and datacenters.
- Since year 2003, due to energy consumption and heat dissipation issues, vendors have switched to **multi-core** architectures to increase the processing power, with **multithreading** emerging as the reference programming model (**concurrency revolution**).



- Multi-core** architectures (tens of cores) seeks to maintain the execution speed of sequential programs while moving into multiple cores.
- Many-core**, or many-thread architectures (thousands of cores) focuses more on the execution throughput of parallel applications.
- In terms of raw speed, **many-core processor, like GPUs, are two or three order of magnitude faster than multi-core solutions**, even though CPUs perform faster on some problems.

<sup>2</sup>GFLOPS: Giga Floating-Point Operations per Second

<sup>3</sup>TFLOPS: Tera Floating-Point Operations per Second



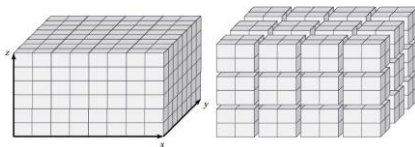
# Data Parallelism and GPU-Suitable Applications

- Most applications process a **huge amount of data**, with millions to trillions of:
  - pixels (Images/videos)
  - grid cells (fluid-dynamics applications)
  - atoms (molecular dynamics applications), and so on...
- Often, most of these elements, can be dealt with largely **independently**.
  - Convert a color pixel to a greyscale requires only the data of that pixel;
  - Updating the fluid mass in a cell requires only the data of small neighborhood of the cell;
  - Atoms/particle dynamic can be often modeled without accounting for collisions (e.g., in the PIC model<sup>4</sup>);
  - Even a seemingly global operation, such as finding the average brightness of all pixels in an image, can be broken down into many smaller computations that can be executed independently.
- Such independent evaluation is the basis of **data parallelism**: *(re)organize the computation around the data, such that we can execute the resulting independent computations in parallel.*

<sup>4</sup>Particle-in-cell: <https://en.wikipedia.org/wiki/Particle-in-cell>

# GPU-Suitable Applications

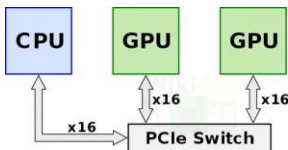
- A **data parallel application** is based on an usually large domain that can be partitioned in small elements (e.g., cells or nodes of a structured grid). Much of the computation can be done on different parts of the data in parallel, although in certain cases they will have to be reconciled at some point.



- Generally, **data parallel applications** are suitable for GPU execution. Examples are:
  - Arrays and matrices based algorithms;
  - Simulation of complex phenomena in Physics, Biology, ...
  - Training of Deep Neural Networks;
  - Evolutionary Algorithms.
- 10x to 100x (and more speedup) can be achieved.
- However, most applications have portions that can be much better executed by the CPU. In such a case, a combined CPU/GPU implementations is the best choice.
- Eventually, other applications are simply more suitable for CPU execution.

# Heterogeneous Parallel Computing

- GPUs are installed into CPU-based **host systems** generally using the PCI-Express bus<sup>5</sup>.



- Applications generally use CPUs and GPUs together, executing the sequential parts on the CPU and numerically intensive parts on the GPUs (**Heterogeneous Parallel Computing**).
- Until 2006, GPGPU Programming was difficult since the OpenGL or Direct3D graphics APIs were the only available software solutions.
- The Nvidia [CUDA](#) programming model, as well as the Khronos [OpenCL](#) and the newer [SYCL](#) technologies, besides others, now support heterogeneous computing on CPU/GPU systems.


<sup>5</sup>Starting from the Pascal family, Nvidia GPUs also support NVLINK, a CPU–GPU and GPU–GPU interconnection faster than PCI-Express.

# Architecture of a Modern GPU

There is a fundamental difference between programming a CPU and a GPU (using CUDA).

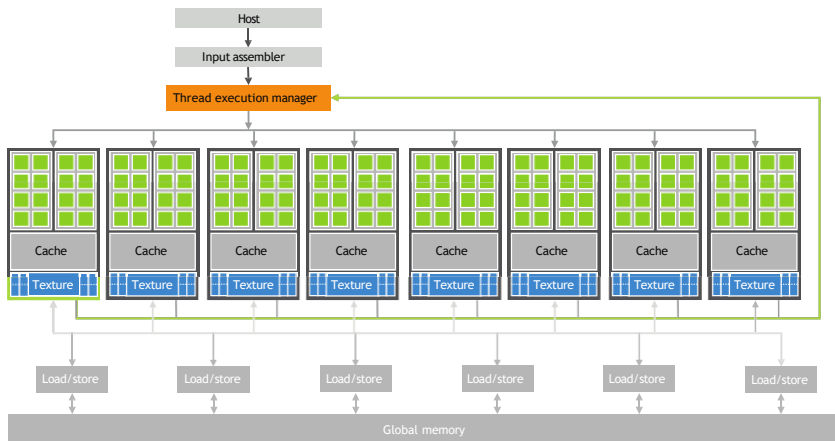
- For a CPU-based host system, the programmer can obtain high performance even ignoring the system architecture<sup>6</sup>.
- For the GPU, the programmer can not generally obtain high performance by ignoring the system architecture.
- Moreover, to run a CUDA application, at least a basic knowledge of the GPU architecture is required regarding the design of the computing and memory systems.
- In this respect, CUDA can be considered as a kind of low-level API.

---

<sup>6</sup>A knowledge of the CPU architecture is in any case needed for performance optimization purposes. 

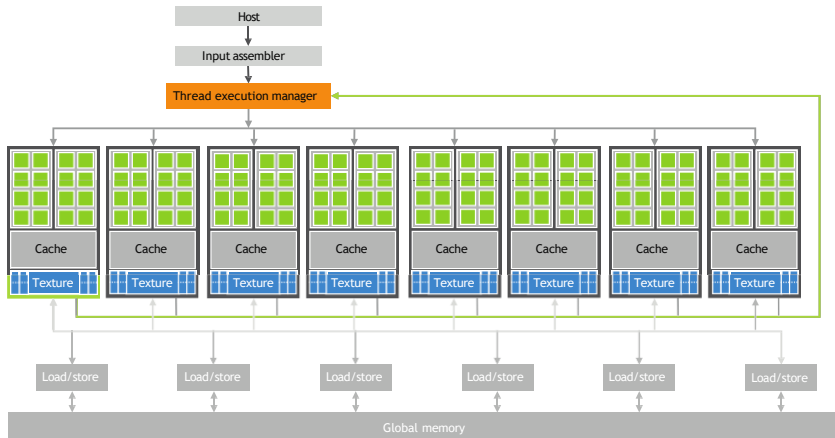
# Architecture of a Modern GPU

GPUs are designed to run thousands of threads and more! Typically they are organized into an **array of streaming multiprocessors (SMs)**.



# Architecture of a Modern GPU

Each SM has a number of **streaming processors (SPs)**.



For instance, the GTX 980 GPU has 16 SMs, each with 128 SPs, for a total of 2048 SPs (aka cores).

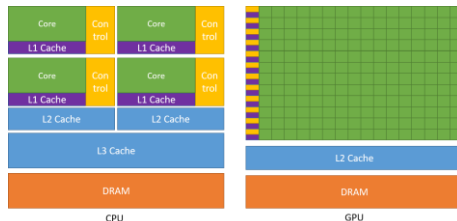
# Architecture of a Modern GPU: Execution Model

- A GPU can run more threads than its SPs.
  - A **scheduler** decides which threads must be in the **run** state, keeping the others in the **wait** or **ready** to run state<sup>7</sup>.
  - Therefore, **threads are only logically executed in parallel**. For instance, the GTX 980 can only keep 2048 threads running simultaneously, while can logically run in parallel a considerably greater number of threads.
  - Threads **consume GPU (memory) resources** to keep track of their state and data. Therefore, the maximum number of threads that can be run depends on the GPU design.
- Threads are subdivided in **blocks** and processed by the SMs. The set of blocks is called **grid**.
- Each block is further subdivided in **warps**, batches of 32 threads executed in lock-step (i.e., all of them fetch the same instruction and share the same control logic).

---

<sup>7</sup>This mechanism permits achieving the so-called latency hiding.

# CPUs vs GPUs



CPUs are optimized for **sequential/few-threads** performance.

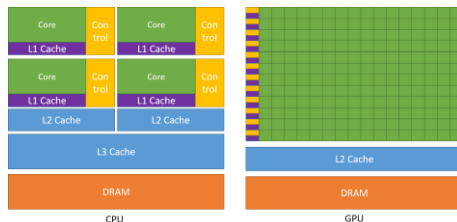
- **Sophisticated control logic** allow **serial instructions** to execute **in parallel** or even out of their sequential order.
- **Large Cache** (many MB) reduce the accesses to the higher-latency DRAM.

GPUs are **parallel, throughput-oriented** computing devices;

- Many threads can be quickly created/destroyed than on the CPUs.
- GPU's DRAM has a higher bandwidth than the CPU's DRAM (about 10x factor, so HBM).



# CPUs vs GPUs



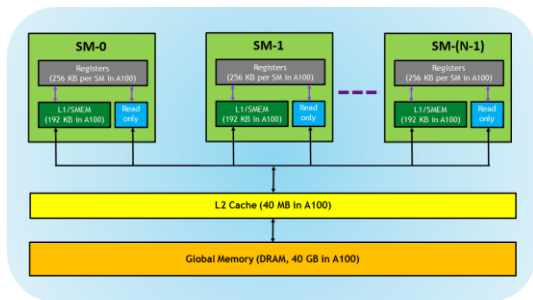
- Many threads allow for **latency hiding**<sup>8</sup>.
- Small Cache Memories**, either global (i.e., for all the SMs) and per SM, permit to reduce the accesses to the higher-latency DRAM.
- Per SM **Scratchpad Memories**<sup>9</sup> permit reducing DRAM accesses, if needed (e.g., when caching is not sufficient).

<sup>8</sup>The ability to find work to do for ready threads when others are waiting for long-latency memory accesses or arithmetic operations.

<sup>9</sup>Non-transparent cache-like memory.

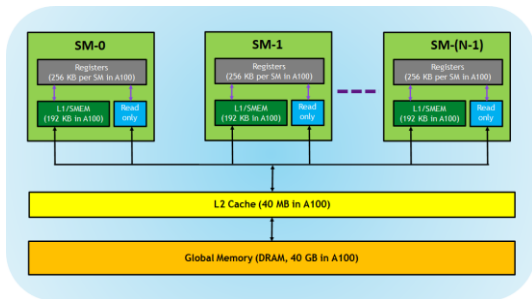
# Architecture: CUDA Physical Memory Model

- **Registers** (per thread fast on chip memory)
- **L1 Cache/Shared Memory** (per SM fast on-chip memory)
- **Read-only Memory** (per SM, for instructions, constant data, ...)
- **L2 Cache** (per GPU, visible globally)
- **DRAM** (per GPU, visible globally and accessible by the host)

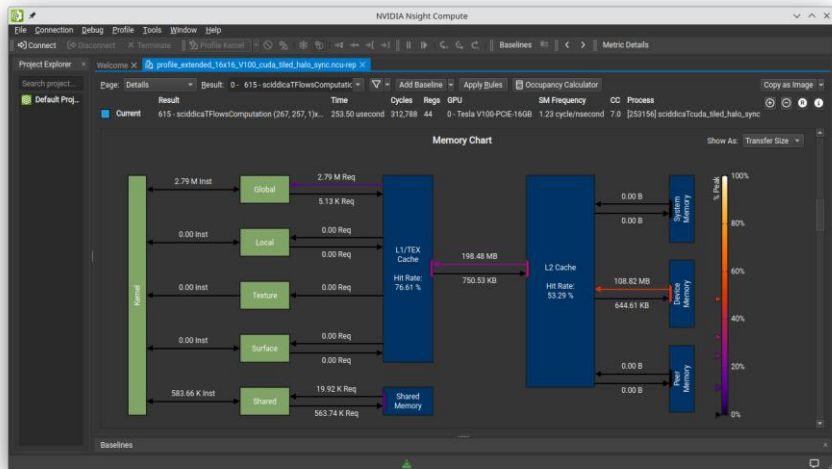


# Architecture: CUDA Logical Memory Model

- Local Memory (private variables, stored into Registers or DRAM)
- Shared Memory (per block data, stored in Shared Memory)
- Constant Memory (global read-only data, stored in DRAM and cached in L1)
- Global Memory (global data, stored in DRAM)



# ncu-ui Memory Chart

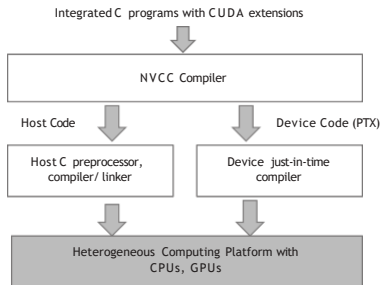


# Data Parallel Computing

## Data Parallel Computing

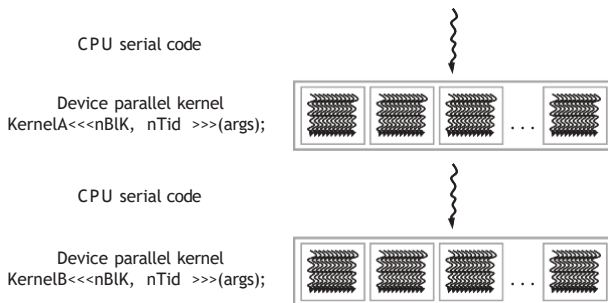
# CUDA C

- **CUDA C** is a data parallel extension of the C programming language for Nvidia GPUs.
- The structure of a CUDA C program reflects the coexistence of a **host** (CPU) and one or more **devices** (GPUs) in the computer.
- The **nvcc** compiler processes a CUDA C program, using the CUDA keywords to separate the host code and device code.

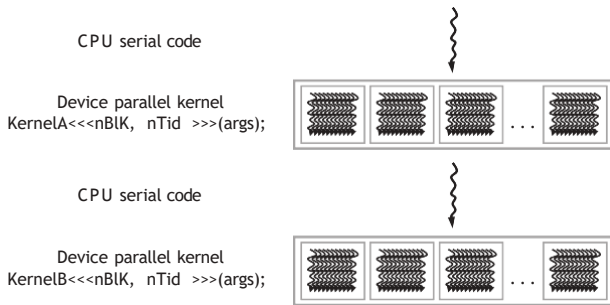


# CUDA C

- The device code is marked with CUDA keywords for data parallel functions, called **kernels**, and their associated helper functions and data structures.
  - The device code is further compiled by a run-time component of **nvcc** and executed on a GPU device.
- The execution starts with CPU host serial code. When a kernel function is called, it is executed by a **pool of threads** on a device.



# CUDA C



- CUDA maintains a default **execution queue (stream)**, which is transparent to the programmer<sup>10</sup>. Each time a kernel or other GPU tasks are invoked by the host, they are added to the queue and processed **in order**. The host call can be **synchronous** or **asynchronous**.

<sup>10</sup>More streams can be created explicitly for concurrent operations.



# Vector Addition

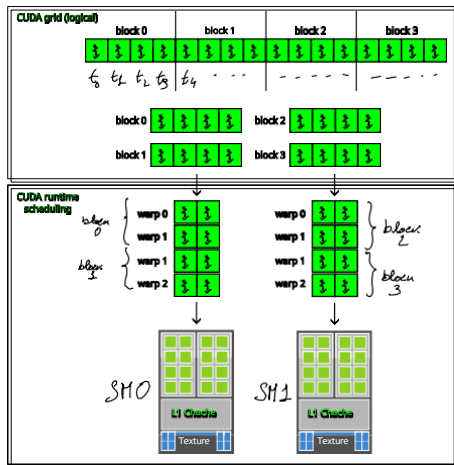
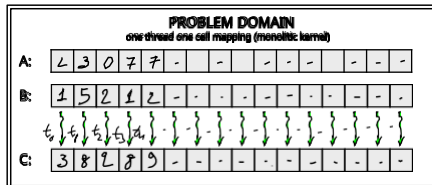
- The vectors to be added are stored in arrays **A** and **B**, while the output vector is in array **C**.
- **N** is the length of each vector.
- **h\_** and **d\_** mark data stored on the host and device, respectively.
- Here is the serial implementation...

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    for (int i = 0; i < n; i++)
        h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C, I/O, etc...

    vecAdd(h_A, h_B, h_C, N);
}
```

# Vector Addition



# Vector Addition

## CUDA Vector Addition Host-side Function

```
#include <cuda.h>

void vecAdd(float* A, float* B, float* C, int n)
{
    float *d_A, *d_B, *d_C; int size = n* sizeof(float);
    int block_size = 32, number_of_blocks = ceil(n/block_size);

    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<number_of_blocks, block_size>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

# Device Global Memory and Data Transfer

- Preliminarily, memory is allocated on the device [**Asynch**] and data transferred to the device [**Synch**]<sup>11</sup>

```
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);
cudaMalloc((void**)&d_C, size);
cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
```

- The kernel is enqueued using <<< >>> [**Asynch**]

```
vecAddKernel<<<number_of_blocks, block_size>>>(d_A, d_B, d_C, n);
```

The total number of threads, i.e. *number\_of\_blocks* · *block\_size*, must be  $\geq$  of the elements to be processed (**monolithic execution**).

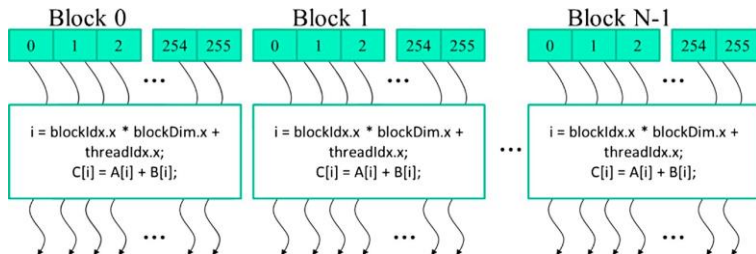
- Eventually, the data is copied back to the host [**Synch**] and the device memory is released [**Asynch**].

```
cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
```

<sup>11</sup> cudaMemcpy is not always blocking.

# Kernel Functions

- A kernel specifies the code to be executed by the threads.
- The kernel is executed by a uniform **grid of thread blocks**.
- A **0-based index** is assigned to each block in the grid, and to each thread in a block. A **global thread index**,  $i$ , can be easily obtained, which can be used to access the data elements to be processed.



- The grid replaces the serial loop.

# Kernel Functions

- Both the grid and the blocks can be **1D**, **2D**, or **3D**. The CUDA built-in variables **blockIdx**, **blockDim**, and **threadIdx** have the **x**, **y** and **z** fields that store the per-dimension information. In the multidimensional case, the global indices are obtained as:

```
ix = blockIdx.x * blockDim.x + threadIdx.x //1D, 2D, and 3D grids  
iy = blockIdx.y * blockDim.y + threadIdx.y //2D and 3D grids  
iz = blockIdx.z * blockDim.z + threadIdx.z //3D grids
```

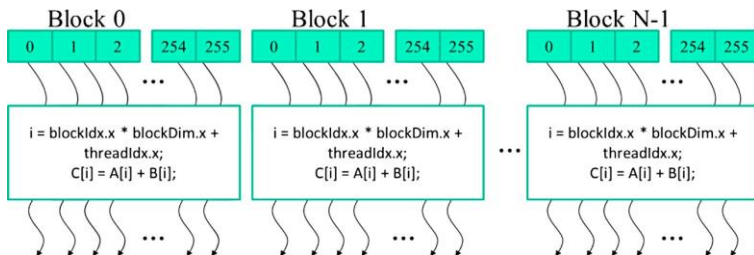
- Each device define the **max number of threads in a block** and the **max number of blocks in a grid**.
  - The GTX 980, allows for up to 1024 threads per block and  $2^{31}$  blocks in the grid in the 1D case.
- Other important information are the **max number of threads per SM** and the **max number of blocks per SM**
  - The GTX 980, allows for 2048 threads and 32 blocks per SM.
  - Problems of at least 32768 data elements are needed to exploit the 16 SMs of the GTX 980 GPU, each one able to run 2048 threads.
- The max number of threads per SM should be maximized to achieve best performance.**

# Vector Addition Kernel

## Vector Addition Device-side Kernel

```
// The serial loop is replaced by the pool of threads
// Each thread performs one pair-wise addition
__global__ void vecAddKernel(float* A, float* B, float* C, int n)
{
    //i is a private variable
    int i = blockDim.x*blockIdx.x + threadIdx.x;

    if (i < n) // to be sure to "intercept" data
        C[i] = A[i] + B[i];
}
```



## Vector Addition Kernel

- The `__global__` keyword means that the kernel is launched from the host and executed on the device. CUDA function keywords are:

|                                 | Callable from  | Executed on | Executed by                |
|---------------------------------|----------------|-------------|----------------------------|
| <code>__host__</code> (default) | Host           | Host        | Caller host thread         |
| <code>__global__</code>         | Host or Device | Device      | New grid of device threads |
| <code>__device__</code>         | Device         | Device      | Caller device thread       |

- The loop is now replaced by the CUDA grid: Each thread corresponds to one iteration of the original loop (*loop parallelism*).
- The automatic (local) variable `i` is private to each thread, i.e., an instance will be generated for every thread.
- The `if` statement is necessary because not all vector lengths can be expressed as multiples of the block size.

### Example

Assume that we picked 32 as block size (i.e., the smallest efficient CUDA block size). Four blocks, for a total of 128 threads, are needed to process all the 100 vector elements. Since all threads execute the same code, all will test their `i` values against `n` (i.e., 100). With the `if (i < n)` statement, the first 100 threads will perform the addition whereas the last 28 will not.



# Error Handling

- **Synchronous error** happens when the host thread knows the kernel is illegal or invalid.
  - For example, when the block size or grid size of a kernel is too large, an error is resulted immediately after the kernel launch call.
  - This error could be captured by runtime error capturing API calls, such as `cudaGetLastError`, right after the kernel launch call.
- **Asynchronous error** happens during kernel execution or CUDA runtime asynchronous API execution on GPU (current execution aborts and an error is sent back to host thread). Examples are:
  - Invalid memory access in the late stage of kernel execution <sup>12</sup>.
  - Invalid memory access during `cudaMemcpyAsync` execution.
  - This error could be (inefficiently) captured by `cudaGetLastError` after an explicitly synchronization using calls such as `cudaDeviceSynchronize`, `cudaStreamSynchronize`, or `cudaEventSynchronize`.

---

<sup>12</sup>Kernel launch is asynchronous, meaning that the host thread issues a request to execute the kernel on GPU, then it continues without waiting for the kernel to complete.

# Error Handling

- **Sticky error.** Subsequent CUDA calls will return the same error (the CUDA context is corrupted).
  - For instance, if a kernel accesses invalid memory address, it will result in a sticky error. Therefore, it is better to check...

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

- **Non-sticky error.** Subsequent CUDA calls will NOT return the error (the CUDA context is not corrupted).
  - For instance, `cudaMalloc` returns a non-sticky error if the GPU memory is insufficient. Therefore, it is better to check...

```
cudaError_t err = cudaMalloc((void **) &d_A, size);  
if (err != cudaSuccess) {  
    printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__, __LINE__);  
    exit(EXIT_FAILURE);  
}
```

# To do list

- Read this article about *error handling*: <https://leimao.github.io/blog/Proper-CUDA-Error-Checking/> (on which i based the previous two slides about error checking).
- Checkout the *vectorAdd* example.
- Read this article about *monolithic* kernels and kernels with *grid-stride loops*: <https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/>
- Read this article regarding the CUDA *Unified Memory*: <https://developer.nvidia.com/blog/unified-memory-cuda-beginners/>

# Scalable Parallel Execution

## Scalable Parallel Execution

# Cuda Thread Organization

CUDA threads:

- Execute different instances of the same kernel function;
- Use the thread index to access data element to process.

In general, a grid is a three-dimensional array of blocks, and each block is a three-dimensional array of threads<sup>13</sup>

The grid is defined when the kernel is launched, as shown below

## Example of kernel launch

```
dim3 block_size(256, 1, 1);  
dim3 number_of_blocks(ceil(n/(float)block_size.x), 1, 1);  
vecAddKernel<<<number_of_blocks, block_size>>>(...);
```

Both `number_of_blocks` and `block_size` parameters are of type `dim3`, which is a C struct with three unsigned integer fields: x, y, and z.

---

<sup>13</sup>The programmer can use fewer than three dimensions by setting the size of the unused dimensions to 1

# Cuda Thread Organization

For convenience, CUDA C allows to use an integer expression instead of `dim3` for 1D grids and blocks. In the example

## Example of kernel launch

```
dim3 block_size(256, 1, 1);  
dim3 number_of_blocks(ceil(n/(float)block_size.x), 1, 1);  
vecAddKernel<<<number_of_blocks, block_size>>>(...);
```

the number of threads per block is fixed to 256, while the number of blocks is a function of both the data size (`n`) and the block size (`block_size`) in order to have as many threads as the data elements to be processed.

- If `n` is 1000, the grid will consist of 4 blocks, each of 256 threads, for a total of 1024 threads;
- If `n` is 4000, the grid will have 16 blocks, and so on.

Once `vecAddKernel` is launched, the grid and block dimensions will remain the same until the entire grid finishes execution.

# Cuda Thread Organization

Grid and Block Limits (for the GTX 980 - they can vary with the device)

Grid size limits:  $(2^{31}, 2^{16}, 2^{16})$

Block size limits<sup>a</sup>:  $(2^{10}, 2^{10}, 2^6)$

<sup>a</sup>For the block size, the overall limit of  $2^{10} = 1024$  threads must be in any case satisfied.

## Examples of allowed blocks

```
block_size(512, 1, 1)
```

```
block_size(8, 16, 4)
```

```
block_size(32, 16, 2)
```

## Examples of NOT allowed blocks

```
block_size(32, 16, 4) ( $32 \cdot 16 \cdot 4 = 2048 > 1024$ )
```

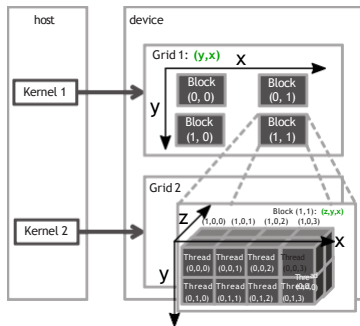
```
block_size(32, 32, 2) ( $32 \cdot 32 \cdot 2 = 2048 > 1024$ )
```

```
block_size(16, 16, 16) ( $16 \cdot 16 \cdot 16 = 4096 > 1024$ )
```

# Cuda Thread Organization

## Example of CUDA Grid

```
dim3 block_size(4, 2, 2); //(x, y, z)
dim3 number_of_blocks(2, 2, 1); //(x, y, z)
cudaKernel<<<number_of_blocks, block_size>>>(...);
```



Note that indices are exposed as 2D or 3D for convenience but they actually are 1D at lower level. The convention adopted by CUDA is to have **x**, **y**, **z** representing the **column**, **row**, **layer** indices, respectively<sup>14</sup>

<sup>14</sup>In the figure, blocks are labeled as `(blockIdx.y, blockIdx.x)`, e.g., Block(1,0) has `blockIdx.y=1` and `blockIdx.x=0`. The authors think this is better to illustrate the mapping of thread coordinates into data indexes in accessing multidimensional data!

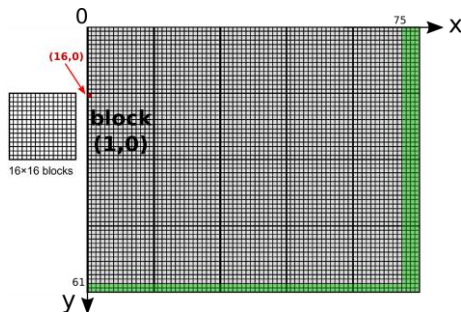


# Mapping Threads to Multidimensional Data

2D grids of 2D blocks are often convenient for 2D raster data

Let consider a 76×62 picture (76 pixels along x, 62 pixels along y)

If we use a 16×16 block we need 5 blocks in the x direction and 4 blocks in the y direction, resulting in 5×4=20 blocks.



The element processed by thread(0,0) of block(1,0) is given by:

$$(\text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y}, \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}) = (16, 0)$$

# Mapping Threads to Multidimensional Data

If  $m$  and  $n$  are the number of pixels in the  $x$  and  $y$  directions, the following code can be used to launch a 2D kernel to process the image

## Example

```
dim3 number_of_blocks(ceil(m/16.0), ceil(n/16.0), 1);  
dim3 block_size(16, 16, 1);  
colorToGrey<<<number_of_blocks,block_size>>>>(d_Pin,d_Pout,m,n);
```

Before commenting the kernel code, we need to understand how to access data in global memory, since in CUDA data is stored as linear buffer, so that a single index must be used (this is due to the fact that CUDA C is compliant with a C standard prior to the C99<sup>15</sup>).

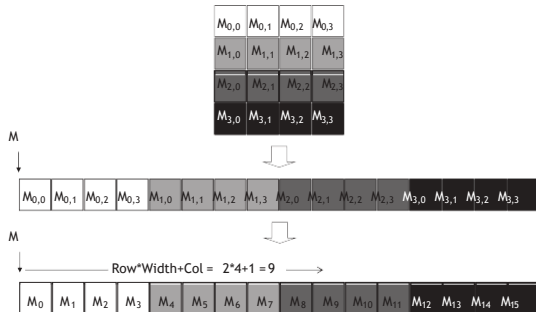
A two-dimensional array can be linearized in *row-major* order, by placing the rows one after another into the memory space.

---

<sup>15</sup>The newer C99 standard allows multidimensional syntax for dynamically allocated arrays. Future CUDA C versions may support multidimensional syntax for dynamically allocated arrays.

# Mapping Threads to Multidimensional Data

CUDA C represents two-dimensional arrays in *row-major* order, by placing the rows one after another into the linear memory space.



For efficiency, **it is important that CUDA thread indices and data indices are consistent** (we will discuss a topic called coalesced accesses later).

# Mapping Threads to Multidimensional Data

## Example of a kernel converting an RGB image to gray-scale

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGrey(unsigned char* Pout, unsigned char* Pin, int width, int height) {
    int Col = threadIdx.x + blockIdx.x * blockDim.x;
    int Row = threadIdx.y + blockIdx.y * blockDim.y;
    if (Col < width && Row < height) {
        // get 1D coordinate for the grayscale image
        int greyOffset = Row*width + Col;
        // one can think of the RGB image having
        // CHANNEL times columns than the grayscale image
        int rgbOffset = greyOffset*CHANNELS;
        unsigned char r = Pin[rgbOffset]; // red value for pixel
        unsigned char g = Pin[rgbOffset + 1]; // green value for pixel
        unsigned char b = Pin[rgbOffset + 2]; // blue value for pixel
        // perform the rescaling and store it
        // We multiply by floating point constants
        Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

## Image Blur: A More Complex Kernel

Blurring smooths out the variation of pixel values by updating the pixel values with a weighted sum of the surrounding pixel values (**convolution**).

Example blurred image



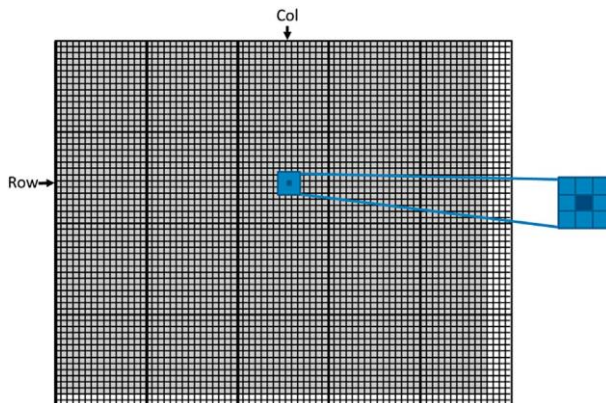
We consider the average value of the  $N \times N$  patch of pixels surrounding, and including, our target pixel<sup>16</sup>.

---

<sup>16</sup>To keep the algorithm simple, we will not consider the distance from the target pixel, which is common in a convolution blurring such as Gaussian blur.

# Image Blur: A More Complex Kernel

An example of patch



# Image Blur: A More Complex Kernel

```
#define BLUR_SIZE 5
__global__ void blurKernel(unsigned char* in, unsigned char* out, int w, int h)
{
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    int Row = blockIdx.y * blockDim.y + threadIdx.y;

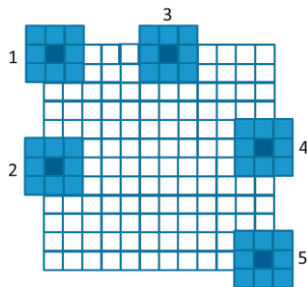
    if (Col < w && Row < h)
    {
        int pixR = 0; int pixG = 0; int pixB = 0;
        int pixels = 0;

        for (int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; blurRow++)
            for (int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; blurCol++)
            {
                int curRow = Row + blurRow;
                int curCol = Col + blurCol;
                if (curRow > -1 && curRow < h && curCol > -1 && curCol < w)
                {
                    pixR += in[3*(curRow * w + curCol)];
                    pixG += in[3*(curRow * w + curCol) + 1];
                    pixB += in[3*(curRow * w + curCol) + 2];
                    pixels++;
                }
            }

        out[3*(Row * w + Col)] = (unsigned char)(pixR / pixels);
        out[3*(Row * w + Col) + 1] = (unsigned char)(pixG / pixels);
        out[3*(Row * w + Col) + 2] = (unsigned char)(pixB / pixels);
    }
}
```

## Image Blur: A More Complex Kernel

The `if (curRow > -1 && curRow < h && curCol > -1 && curCol < w)` statement permitted to properly manage boundary pixels.



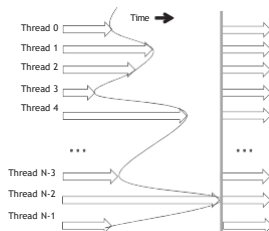


# Synchronization and Transparent Scalability

We will now study a basic thread coordination mechanism.

CUDA allows **threads in the same block** to coordinate their activities by using a **barrier** synchronization function

`__syncthreads ()`<sup>17</sup>



Synchronization is **not possible among threads of different blocks**<sup>18</sup>.

<sup>17</sup> Using `__syncthreads ()` in an `if/if else` statement can be dangerous since some threads could diverge from the barrier call.

<sup>18</sup> Threads in a block can share the resources needed for the sync since they run on a specific SM (Streaming Multiprocessor), while threads of different blocks can not. Using the global memory to share resource for all the running threads could be inefficient and consume too much resources.

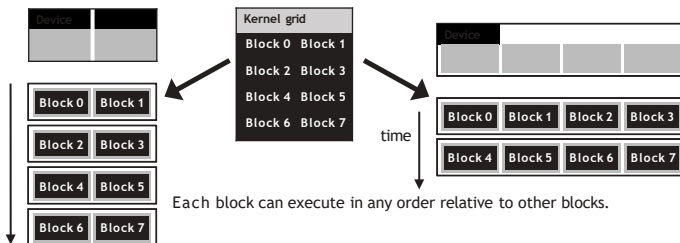
# Synchronization and Transparent Scalability

By not allowing inter-block threads synchronization, **blocks can be executed in any order**. This flexibility enables scalable implementations (**transparent scalability**).

## Example of transparent scalability

In a low-cost system with only a few execution resources, one can execute a small number of blocks simultaneously.

In a high-end implementation with more execution resources, one can execute a large number of blocks simultaneously.

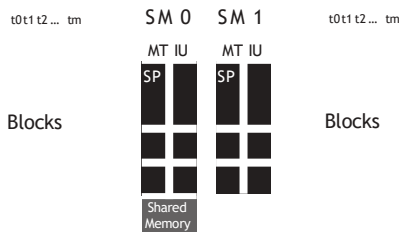


# Resource Assignment

Once a kernel is launched, the CUDA runtime system generates the corresponding grid.

As discussed, these threads are assigned to execution resources (Streaming Multiprocessors - SMs) on a block-by-block basis.

## Example of resource assignment



As we already know, **multiple thread blocks can be assigned to each SM.**

## Resource Assignment

Each device sets a **limit on the number of blocks that can be simultaneously assigned to each SM for concurrent execution.**

- For instance, let us consider a CUDA device that may allow up to 32 blocks to be assigned to each SM<sup>19</sup>.
- **In case of shortage of resources** needed for the simultaneous execution of 32 blocks, **CUDA automatically reduces the number of blocks assigned to each SM** until their combined resource usage falls below the limit.

The number of blocks that can be actively executed in a CUDA device is therefore limited.

Most grids contain many more blocks than those that could be simultaneously processed by the available SMs. **In that cases, the runtime system maintains a list of blocks that need to execute** and assigns new blocks to SMs as previously assigned blocks complete execution.

---

<sup>19</sup>The GTX 980 GPU can execute up to 32 blocks.

# Resource Assignment

One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled.

It takes hardware resources (built-in registers) for SMs to maintain the thread and block indexes and track their execution status. Therefore, each generation of hardware sets a limit on the number of blocks and number of threads that can be assigned to an SM.

- For instance in the GTX 980, up to 32 blocks and 2048 threads can be assigned to each SM. This could be in the form of 2 blocks of 1024 threads each, 4 blocks of 512 threads each, and so on.
- Having the GTX 980 16 SMs where each SM can accommodate up to 2048 threads, the device can have up to 32,768 threads simultaneously residing in the CUDA device for execution.

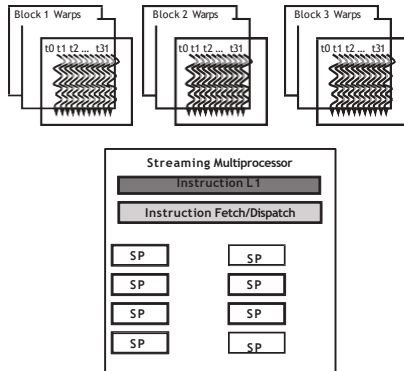
# Querying Device Properties

- In CUDA C, a built-in mechanism exists for a host code to query the properties of the devices available in the system.
- The CUDA runtime system (device driver) API function `cudaGetDeviceCount` gives the number of available CUDA devices in the system, while `cudaGetDeviceProperties` gets device info:

```
int dev_count; cudaDeviceProp dev_prop;
cudaGetDeviceCount(&dev_count);
for (int i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties(&dev_prop, i);
    printf("... %d ...\n", dev_prop.multiProcessorCount);
    printf("... %d ...\n", dev_prop.maxThreadsPerBlock);
    printf("... %d ...\n", dev_prop.maxThreadsDim[0];
    printf("... %d ...\n", dev_prop.maxThreadsDim[1];
    printf("... %d ...\n", dev_prop.maxThreadsDim[2];
    printf("... %d ...\n", dev_prop.maxGrisSize[0];
    printf("... %d ...\n", dev_prop.maxGrisSize[1];
    printf("... %d ...\n", dev_prop.maxGridSize[2];
    printf("... %d ...\n", dev_prop.warpSize); //See next slides...
}
```

# Thread Scheduling And Latency Tolerance

- In the majority of implementations, a block of threads is further divided into units called **WARPS**.
- The size of warps is implementation-specific, usually **32**.
- The warp is the unit of thread scheduling in SMs.



# Thread Scheduling And Latency Tolerance

- SMs execute all **threads in a warp** following the Single Instruction, Multiple Data (**SIMD**) model - i.e., at any instant in time, one instruction is fetched and executed for all threads in the warp.
  - These threads will apply the same instruction to different portions of the data. Consequently, **all threads in a warp will always have the same execution timing** (branches are not good, remember?).
- Assigning **many warps to an SM**, even if it can only execute a small subset of them at any instant in time, **is advantageous**:
  - When an instruction to be executed by a warp needs to wait for the result of a previously long-latency operation (e.g., memory transfer, floating-point arithmetic), the warp is not selected for execution. Instead, **another resident warp that is no longer waiting for results will be executed**.
  - This mechanism of filling the latency of operations with work from other threads is called **latency tolerance** or **latency hiding**.