GPGPU Programming

Donato D'Ambrosio

Department of Mathematics and Computer Science
Cubo 30B, University of Calabria, Rende 87036, Italy

Lecturer's email: donato.dambrosio [at] unical.it
Lecturer's homepage: http://www.mat.unical.it/~donato

Course's homepage: https://www.mat.unical.it/~donato/gpgpu.html
Course team page: https://bit.ly/3CU9xiR
Code to join the team: **3d98qzr**

Academic Year 2021/22

Table of contents

Data Parallel Computing

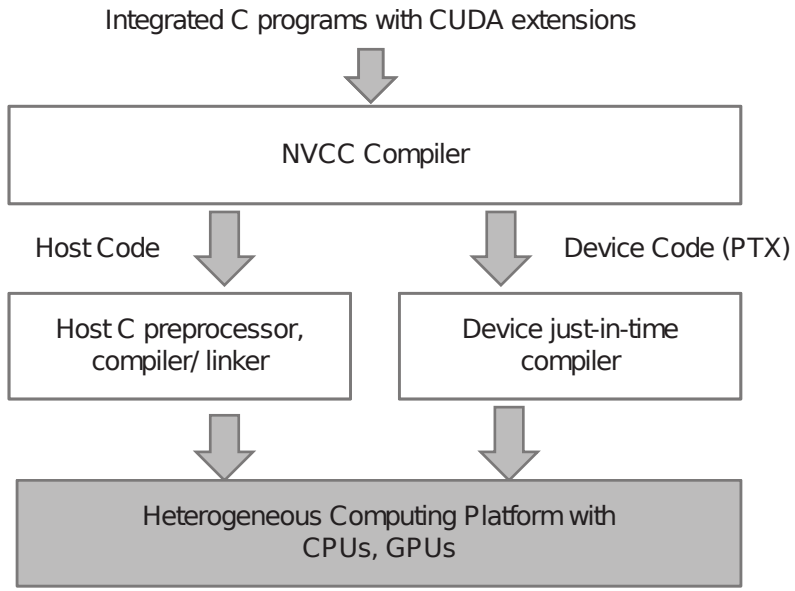
Data Parallel Computing

Data Parallelism

- Most applications process a huge amoutn 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, and so on...
  - 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.

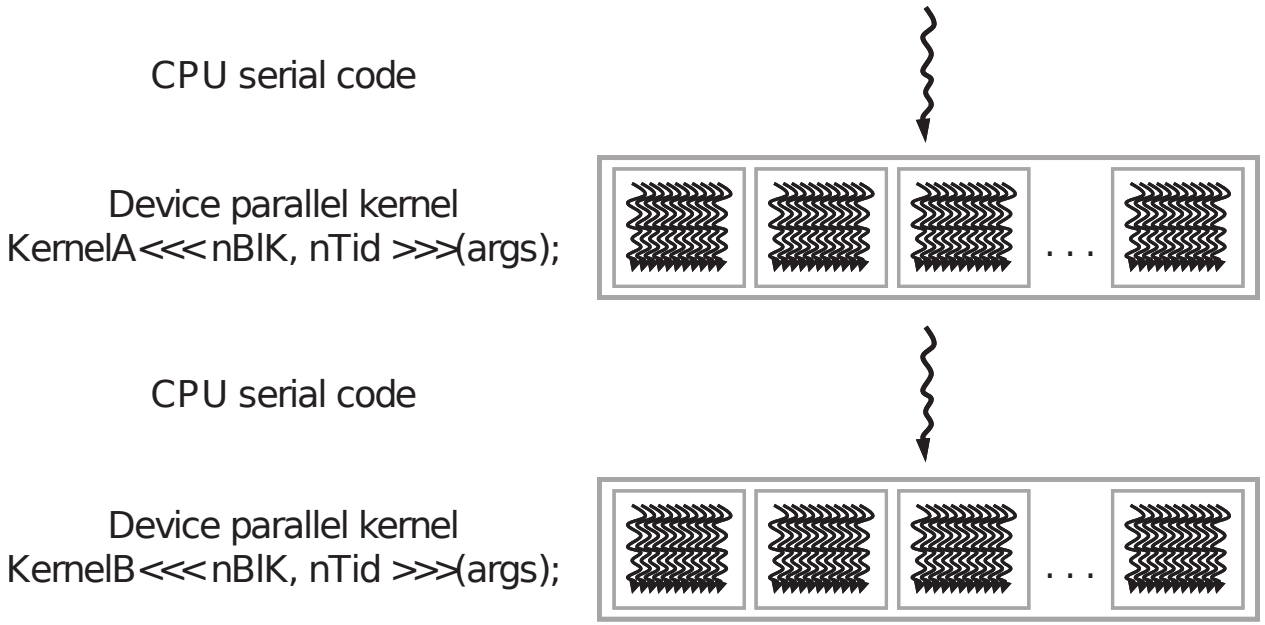
CUDA C

- CUDA C from Nvidia is an extention of the C programming language that permits to exploit the data parallelism for faster execution on Nvidia many-core devices.
- 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.

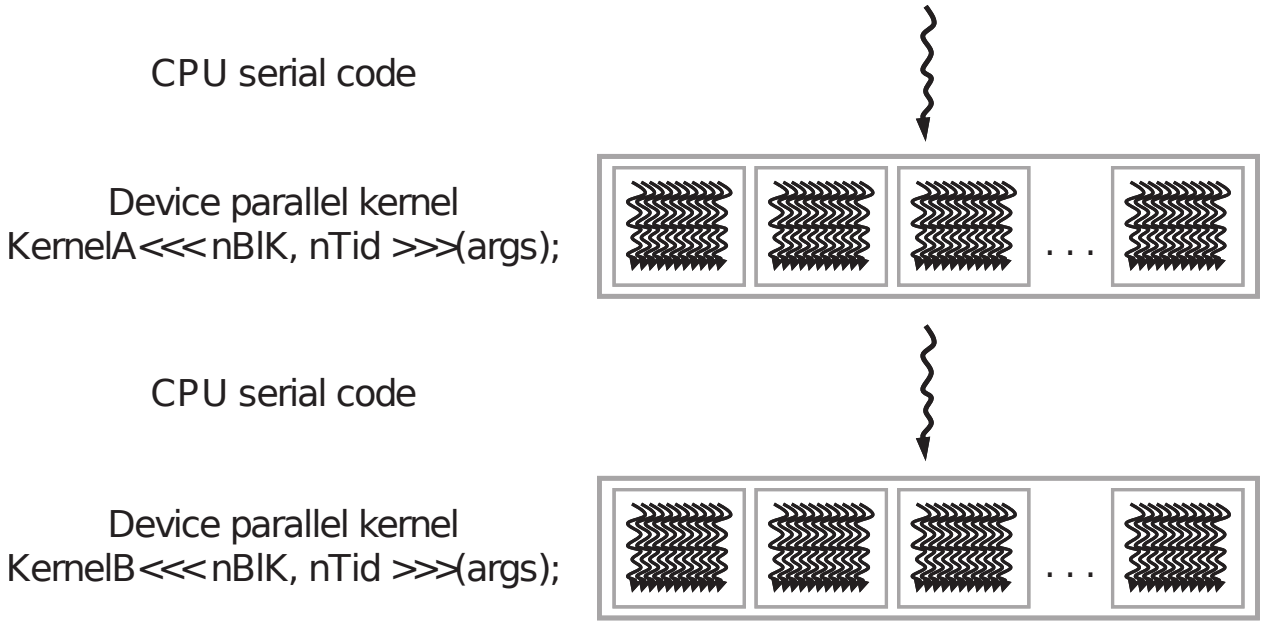


CUDA C

- 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 ***grid of threads*** on a device.



CUDA C



- Many threads are generally needed to exploit data parallelism.
- Threads take few cycles to schedule due to efficient hardware[1].
- When all threads of a kernel finish, the corresponding grid terminates and the execution continues on the host.

[1] This is in contrast with CPU threads that typically take thousands of cycles.

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

- Here is the skeleton of a cuda equivalent version...

```
#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, it is necessary to allocate memory on the device and transfer data from the host to the device memory[2].

```
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 therefore launced with the syntax <<< >>>

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

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

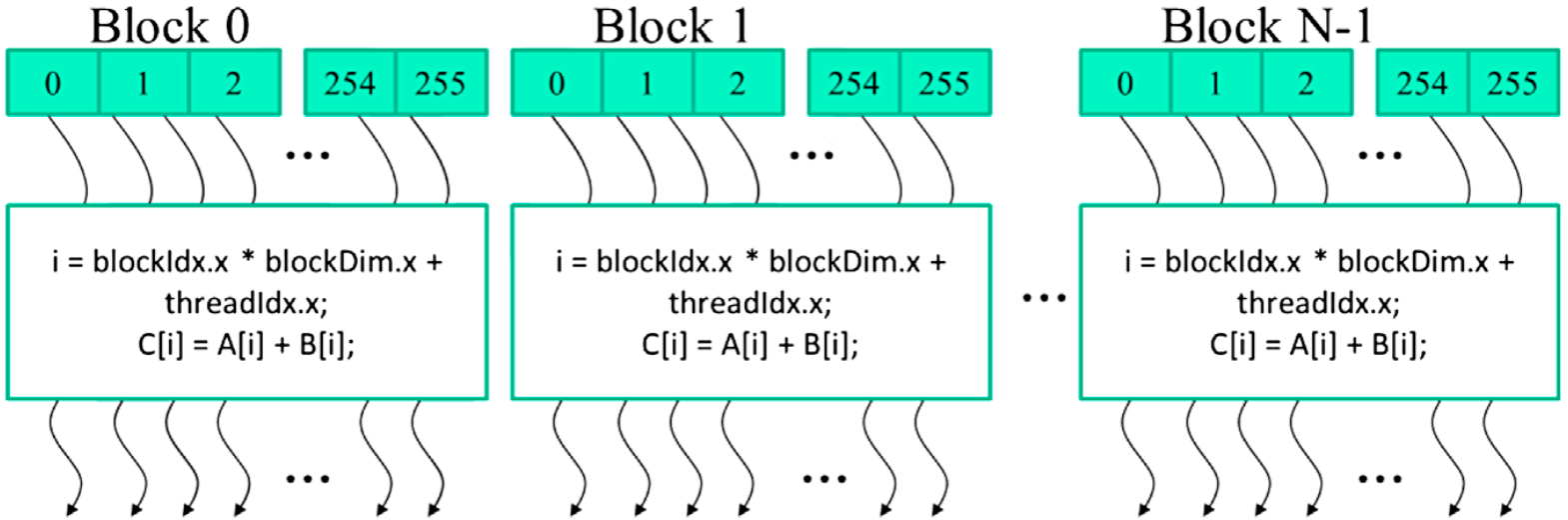
- Eventually, the data is copied back to the host and the device memory is released.

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

[2] The NVIDIA GTX 980 comes with 4 GB of DRAM

Kernel Functions

- In CUDA, a kernel function specifies the code to be executed by the threads of a Single-Program Multiple-Data (SPMD) phase.
- When the host launches a kernel, the CUDA run-time system generates a grid of threads that can be 1D, 2D, or 3D.
- The grid replaces the serial loop, being a 0-based thread index (i.e., from 0 to the grid size - 1) assigned to each thread.
- Each grid is split in blocks of the same size, up to 1024 threads. Here is an example of a 1D grid with blocks of 256 threads



- The block size is specified by the host when a kernel is launched[3].
- The built-in blockDim variable stores the block size. Its three fields, x, y and z, permit to retrieve the grid size in each dimension.
- A 0-based blockIdx is assigned to each block. The x, y and z fields are also available.
- A 0-based threadIdx is also assigned to each thread in a block.
- The global thread index can be easily evaluated as:

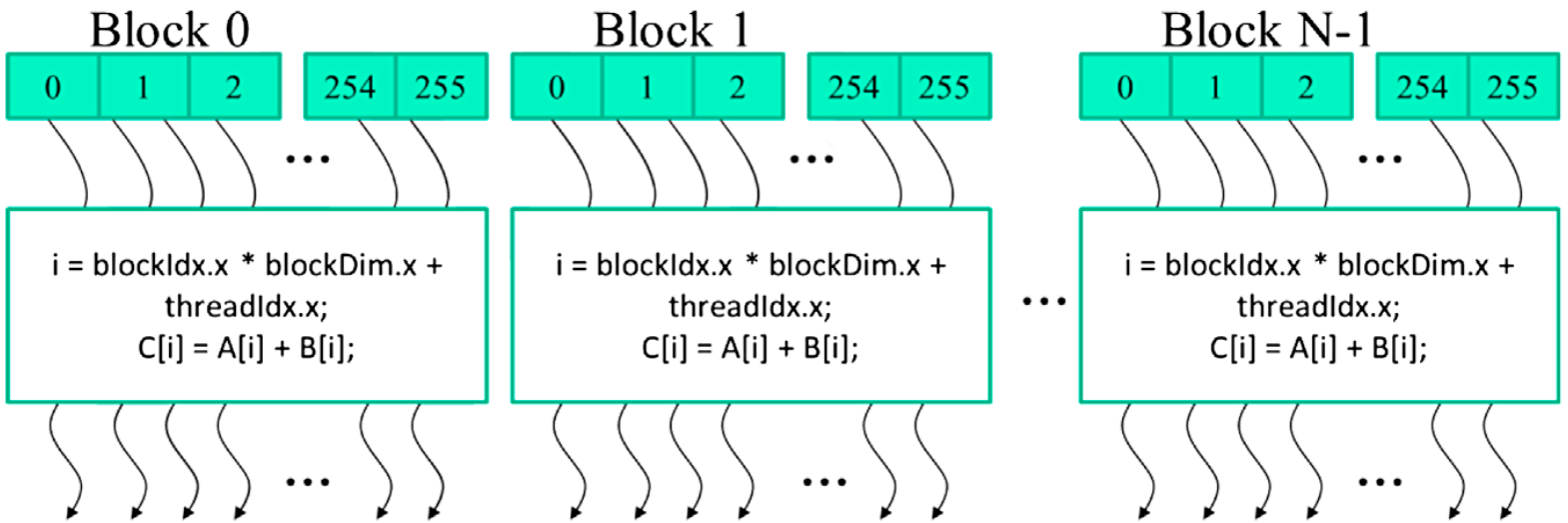
```
ix = blockIdx.x * blockDim.x + threadIdx.x //1D, 2D, and 3D grids
iy = blockIdx.y * blockDim.y + threadIdx.y //2D and 3D grids
iz = blockIdx.z * blockDim.z + threadIdx.z //3D grids
```

[3]Threads in each dimension of thread blocks should be multiples of 32 due to hardware efficiency reasons.
Note that the same kernel can be launched with different numbers of threads at different parts of the host code.

Vector Addition Kernel

```
// The serial loop is replaced by the grid 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
`__global__`	host	device
`__device__`	device	device
`__host__`	host	host

- The loop is now replaced with the grid of threads.
  - The entire grid forms the equivalent of the loop. Each thread in the grid 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.

Vector Addition Kernel

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

Error Handling and Complete Source Code

- For brevity, we did not take into account error checking code. However, we should test for error condition and print out error. For instance, in spite of `cudaMalloc((void **) &d_A, size);` we should write something like:

```
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:
  - 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/