About GPUs and CUDA usage: general introduction and some examples.

Matteo Ottaviani, PhD candidate

19th December 2018, CoMoS meeting,
Sant'Anna School of Advanced Studies, Pisa

Historical introduction on: **Graphic Processing Unit**

- **‘70s:**
  - co-processors utilization for integrating primary processor functions (video games tasks)
  - for any pixel: one color —> evolution towards parallel reduced *cores* (industrial design, 2-D pictures)
- **‘80s:**
  - OpenGL Application Programming Interface (API): versatile interface for programmers
  - first non-graphic usage in physics (*Ising spin glasses Monte Carlo simulations*)
- **2007:** nVidia released FERMI architecture and CUDA first version for C/C++
  - many computational problems with **intrinsic parallelism** could be solved from this point on in considerable less computational time
- …

what does CUDA mean?

- CUDA is a **parallel computing platform** and **programming model** developed by NVIDIA for general computing on GPUs.

- CUDA usage is carried out by extensions in the form of a **few basic keywords**. for programming languages: **C**, **C++**, **Fortran**, **Python** and **MATLAB**.

- In GPU-accelerated codes:

  - the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance.

  - the compute intensive portion of the application runs on thousands of GPU cores in parallel.

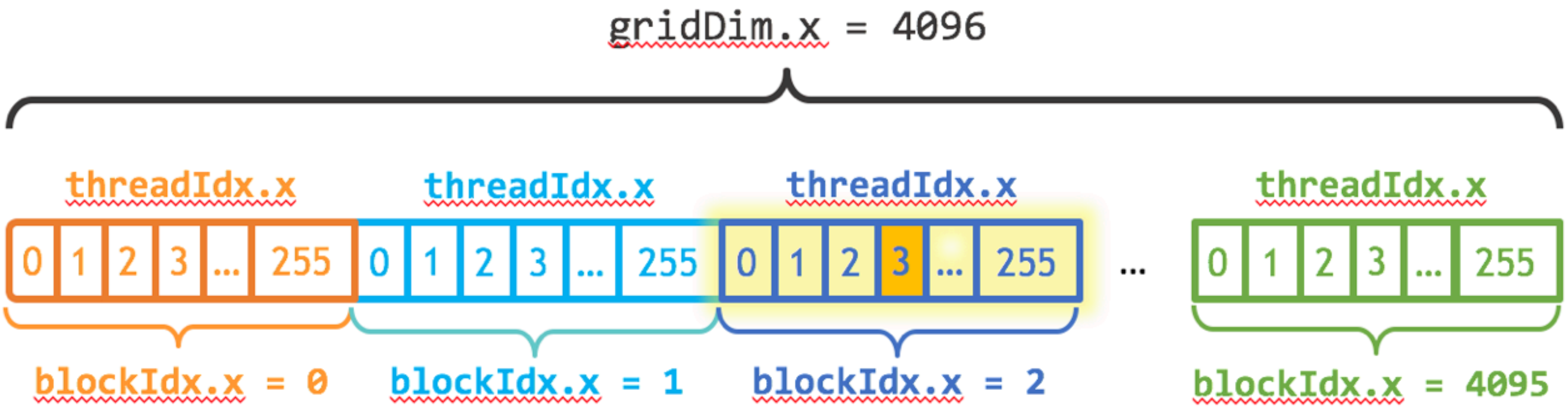
CUDA basic structure

Through extensions to the programming code —> **KERNEL:**

- special functions running directly on GPU and CPU;
- an **instance**, a given object (i.e. variable declaration, etc), is handled by a **THREAD:**
  - **thread block**: it is a set of threads set in a cubic lattice; every thread has three *built-in* variables ***threadId.x, threadId.y*** and ***threadId.z*** . There is a limited number for any direction, depending of the GPU; threads communicate one another only if belonging to the same block (because same microprocessor): *shared memory* , otherwise *global memory.*
  - **block grid:** it is a thread block set; even in this case I could have 3 directions: ***blockId.x, blockId.y*** and ***blockId.z*** . Running order may vary for any different kernel call.

CUDA basic structure

Example:
for the main axis, the CUDA parallel thread hierarchy is:



CUDA basic structure

nVIDIA programming model: **Single Instruction Multiple Threads:**
any core can execute the same operation on different data in input for a given size called **warp.**
Warp is the basic computational unit for nVIDIA GPUs.
(Usually 1 warp = 32 threads)

Thread —> Warp —> Block —> Grid

Other **KERNEL** features**:**

- they are asynchronous: it is allowed to overlap kernel call on the GPU (*device*) and another process on the CPU (*host*); that is because the workflow supervision is due to host.

- different kernels can be executed on the same device: *stream*. This feature lets programmers distribute on a **cluster of GPUs** the workflow of their code.

nVIDIA "Fermi architecture" GPU

global memory (DRAM):
main memory of the GPU; any memory location is accessible from any thread. High latency is the main drawback.

shared memory:
same MC threads accessible only.

…

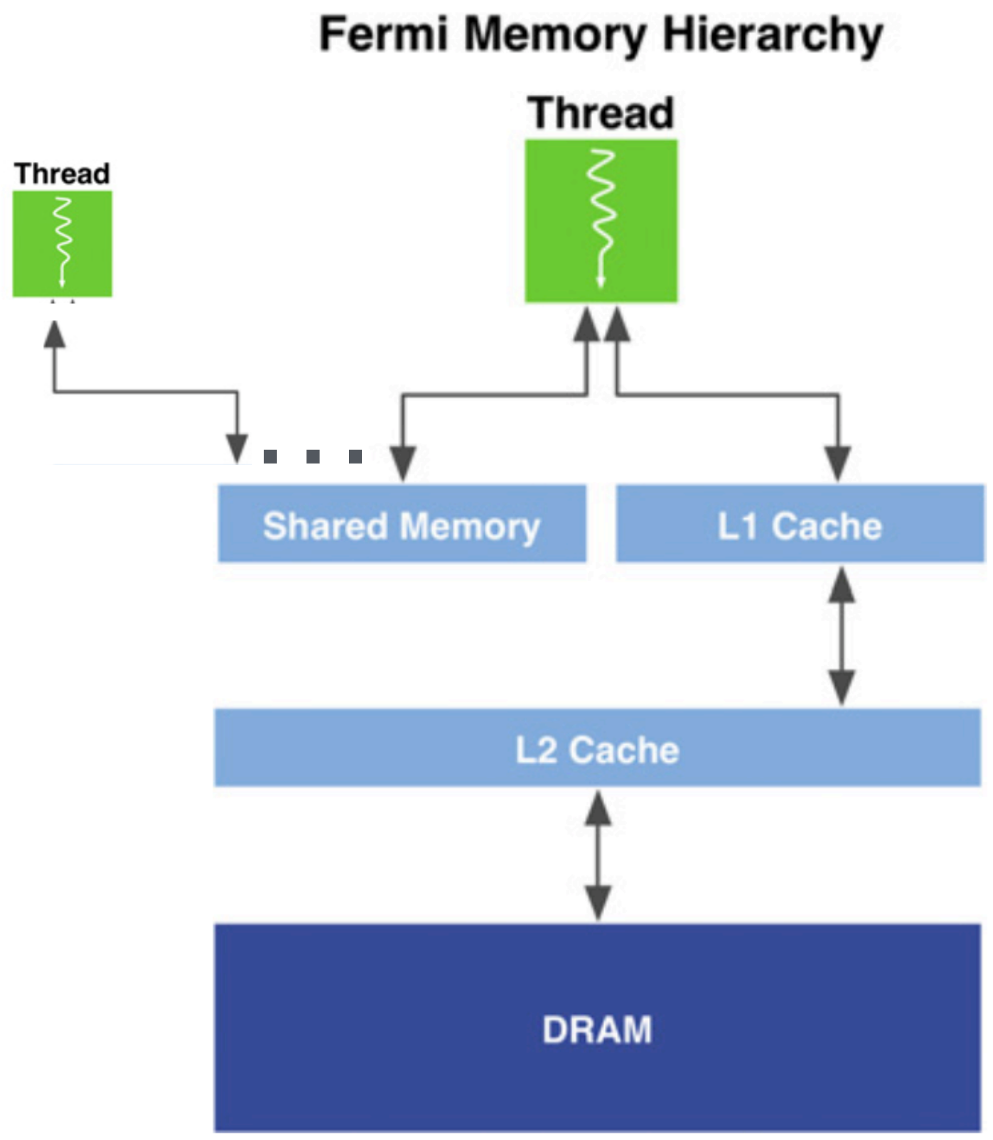


Fig. 1. Memory hierarchy in the Fermi architecture.

Simple illustrative example of CUDA usage

simple C++ program on *host*:
adds the elements of two arrays with a million elements each.

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

I want to get this computation running (in parallel) on the many cores of a GPU.

Simple illustrative example of CUDA usage

I have to turn `add` function into a function that the GPU can run, i.e. a *kernel* in CUDA.

`__global__` function is known as *kernel*, it tells the CUDA C++ compiler this is a function that runs on the GPU and can be called from CPU code.

```
// CUDA Kernel function to add the elements of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

Simple illustrative example of CUDA usage

To compute on the GPU, I need to allocate memory accessible by the GPU.

Unified Memory provides a single memory space accessible by all GPUs and CPUs in your system.

```
  // Allocate Unified Memory -- accessible from CPU or GPU
  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  ...

  // Free memory
  cudaFree(x);
  cudaFree(y);
```

Finally, I need to launch the `add()` kernel, which invokes it on the GPU.

```
add<<<1, 1>>>(N, x, y);
```

I need the CPU to wait until the kernel is done before it accesses the results. To do this I just call `cudaDeviceSynchronize()` .

Simple illustrative example of CUDA usage

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two
arrays
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y;

  // Allocate Unified Memory – accessible
from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);

  // Wait for GPU to finish before accessing
on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be
3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError,
fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError <<
std::endl;

  // Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

this kernel is only correct for a single thread, since every thread that runs it will perform the add on the whole array.

Simple illustrative example of CUDA usage

how do you make it parallel?

The key is in CUDA's `<<<1, 1>>>` syntax: **execution configuration**.

It tells the CUDA runtime how many parallel threads to use for the launch on the GPU.

Let's start changing the second one: the **number of threads in a thread block**.

```
add<<<1, 256>>>(N, x, y);
```

&

modify the kernel through keywords
that let kernels get the indices of the running threads

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

`threadIdx.x`	**index of the current thread**
`blockDim.x`	**number of threads in the block**

Here, I went from 1 thread to 256 threads.

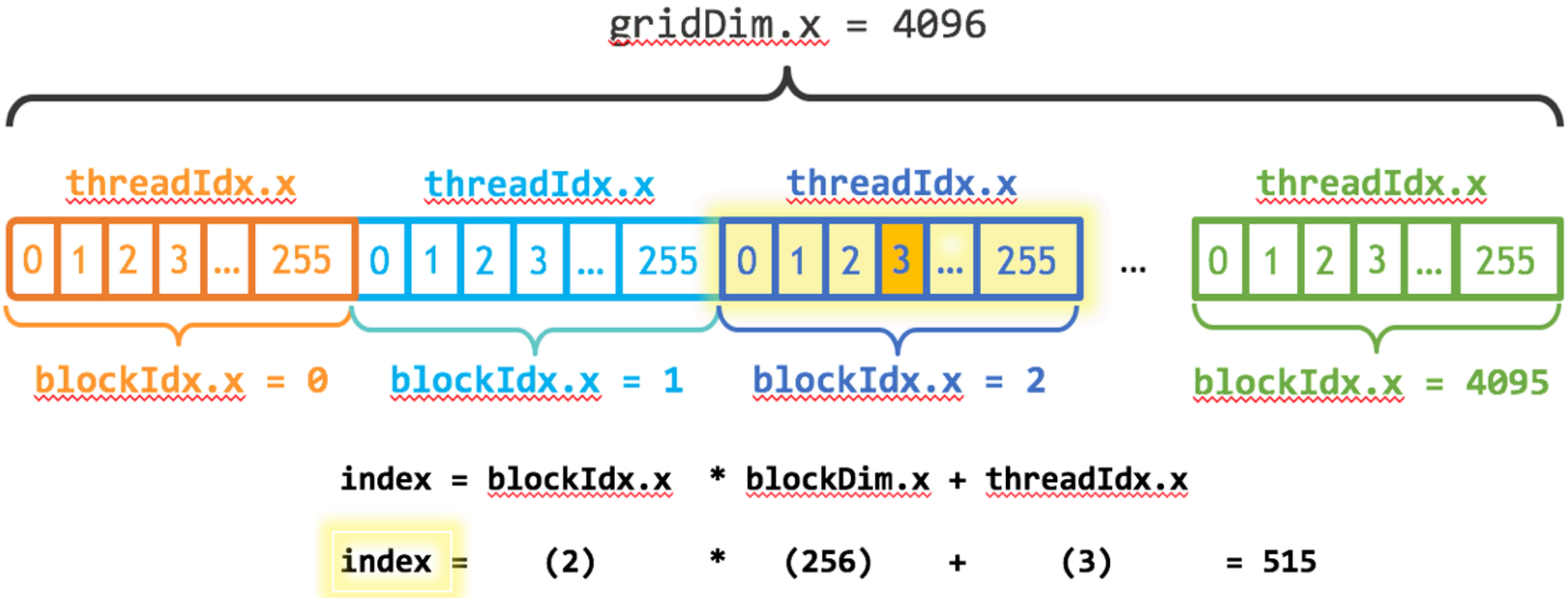
(463ms down to 2.7ms)

Simple illustrative example of CUDA usage

CUDA GPUs have many parallel processors grouped into **Streaming Multiprocessors** or cores, each of them can run multiple concurrent thread blocks.

Since I have `N` elements to process, and 256 threads per block, I just need to calculate the number of blocks to get at least N threads.

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```



```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =     (2)     *   (256)    +    (3)      = 515
```

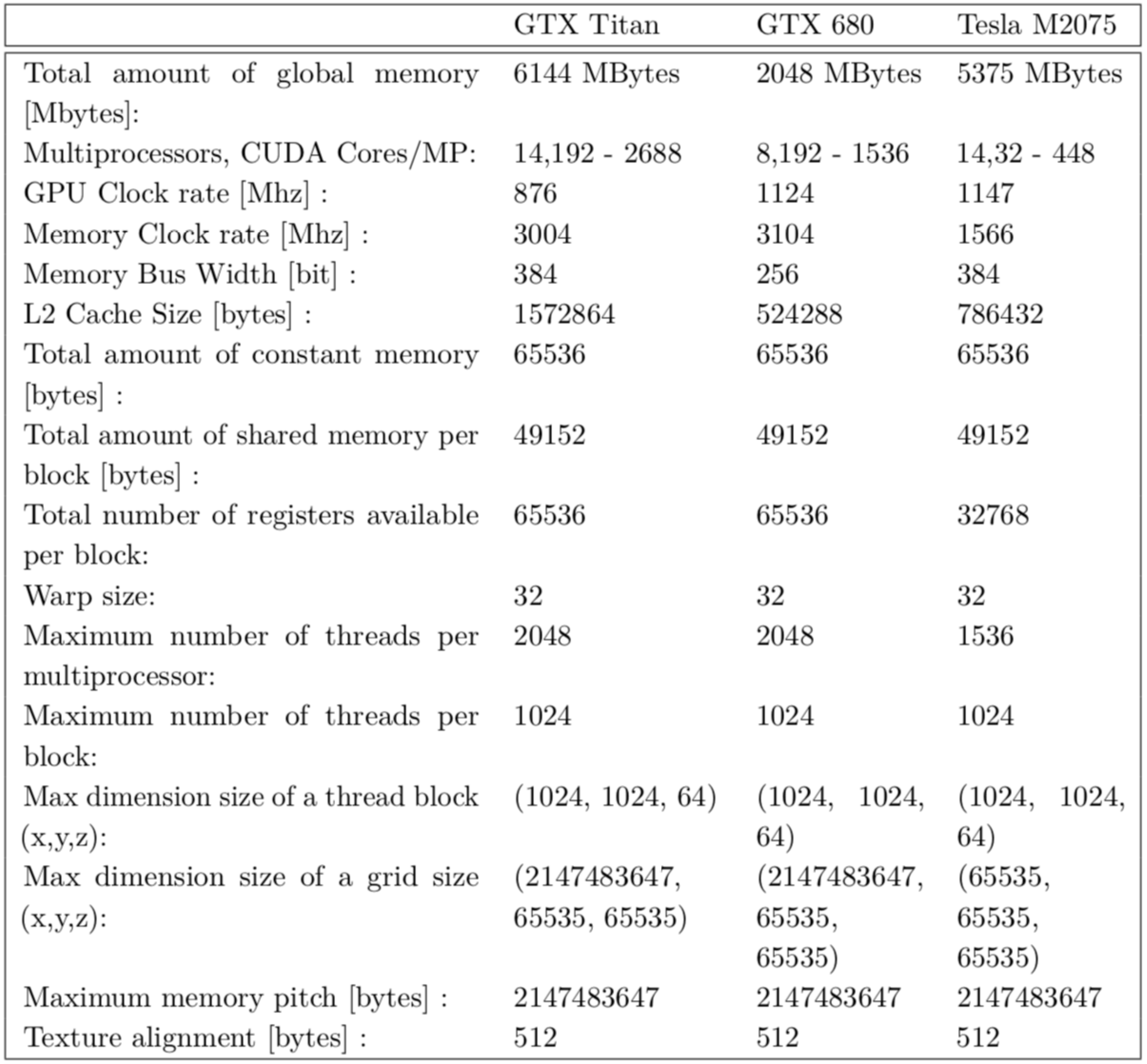
Simple illustrative example of CUDA usage

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

grid-stride loop

Version	Time
1 CUDA Thread	463ms
1 CUDA Block	2.7ms
Many CUDA Blocks	0.094ms

3 different GPUs

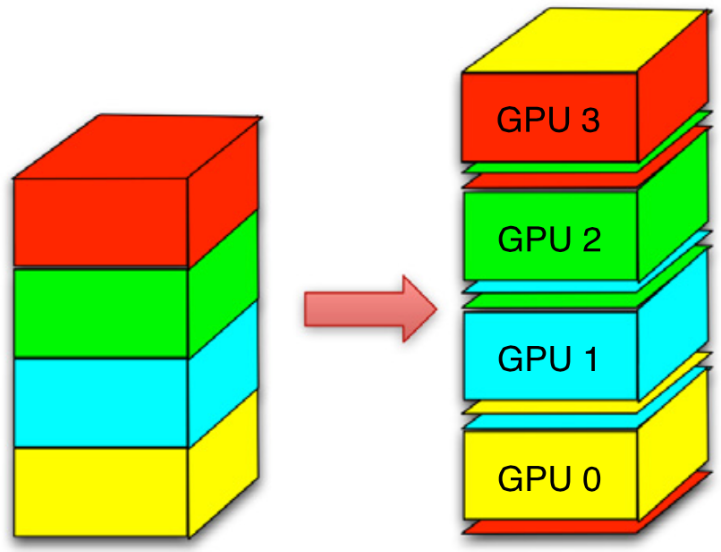


	GTX Titan	GTX 680	Tesla M2075
Total amount of global memory [Mbytes]:	6144 MBytes	2048 MBytes	5375 MBytes
Multiprocessors, CUDA Cores/MP:	14,192 - 2688	8,192 - 1536	14,32 - 448
GPU Clock rate [Mhz] :	876	1124	1147
Memory Clock rate [Mhz] :	3004	3104	1566
Memory Bus Width [bit] :	384	256	384
L2 Cache Size [bytes] :	1572864	524288	786432
Total amount of constant memory [bytes] :	65536	65536	65536
Total amount of shared memory per block [bytes] :	49152	49152	49152
Total number of registers available per block:	65536	65536	32768
Warp size:	32	32	32
Maximum number of threads per multiprocessor:	2048	2048	1536
Maximum number of threads per block:	1024	1024	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)	(1024, 1024, 64)	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)	(2147483647, 65535, 65535)	(65535, 65535, 65535)
Maximum memory pitch [bytes] :	2147483647	2147483647	2147483647
Texture alignment [bytes] :	512	512	512

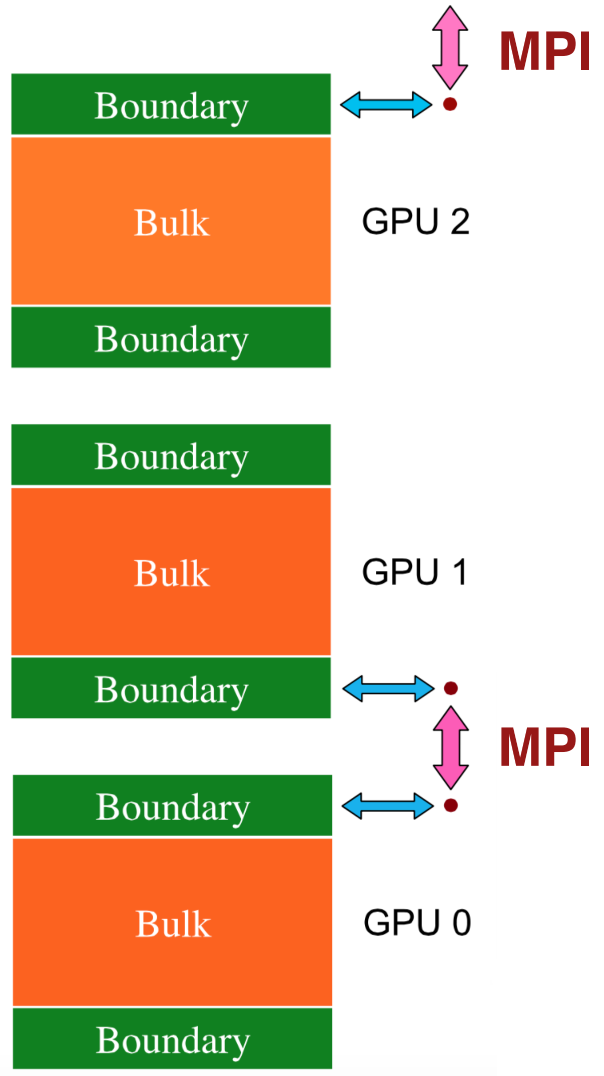
Tabella 3.1. Specifiche principali delle GPUs nVidia utilizzate per le nostre simulazioni numeriche; le GTX Titan e le GTX 680 sul cluster *Piovra* e le Tesla M2075 su *QUonG*.

Multi-GPU simulation of HSG

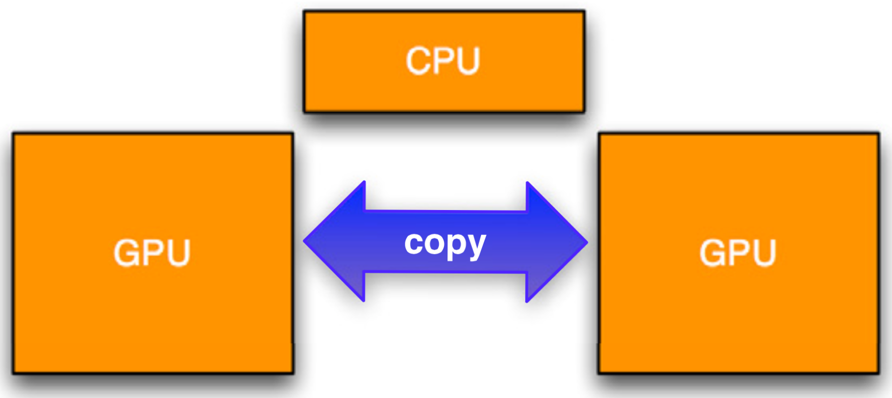
C++/CUDA implementation and optimization of codes[6] on single and multi GPUs using two different platforms of communication



Message Passing Interface (MPI)



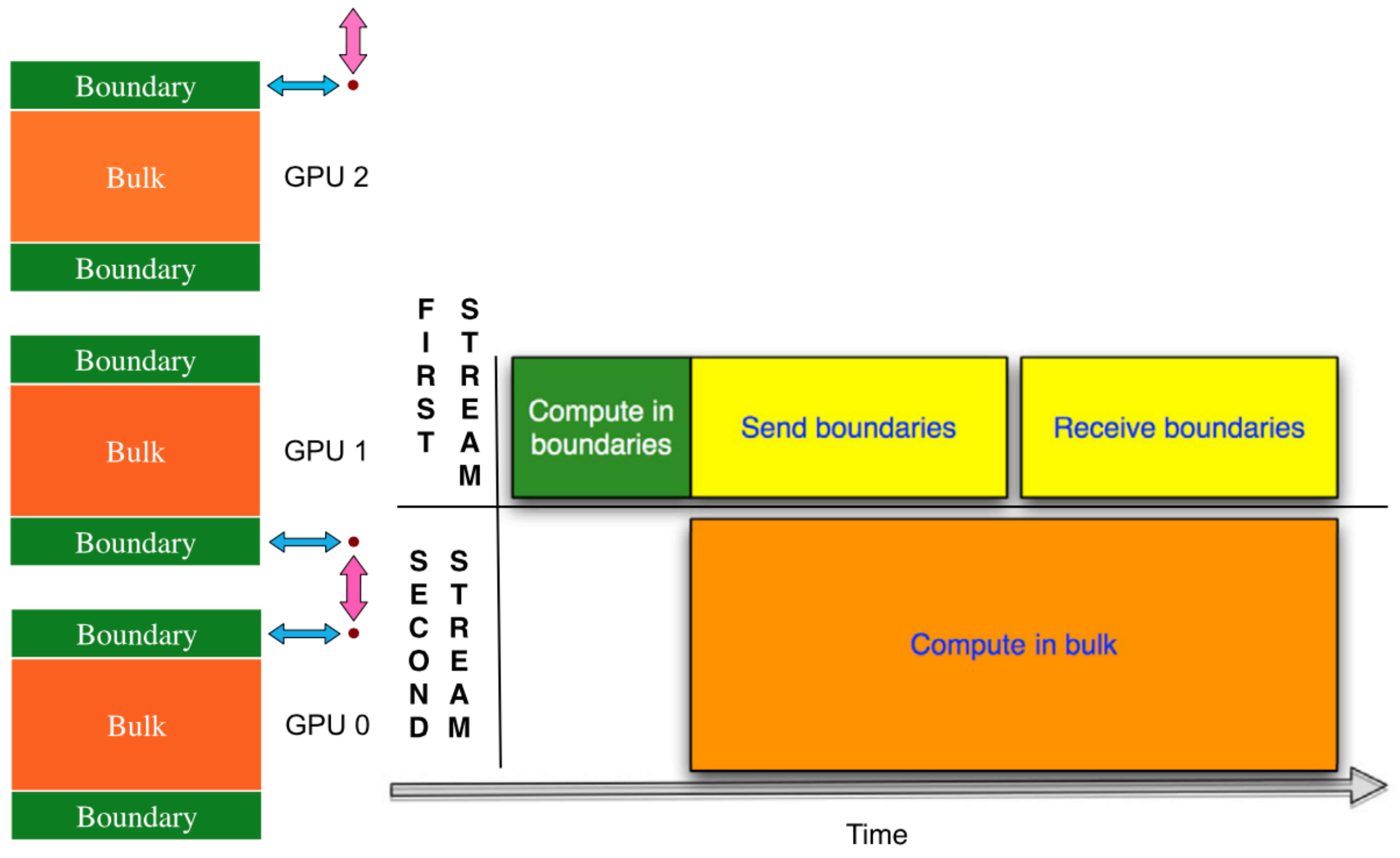
Peer to Peer (P2P)



[6]M. Bernaschi, M. Fatica, G. Parisi, and L. Parisi. *Multi-gpu codes for spin systems simulations*. Computer Physics Communications, vol. 183, no. 7, pp. 1416 – 1421(2012).

Multi-GPU simulation of HSG

stream:

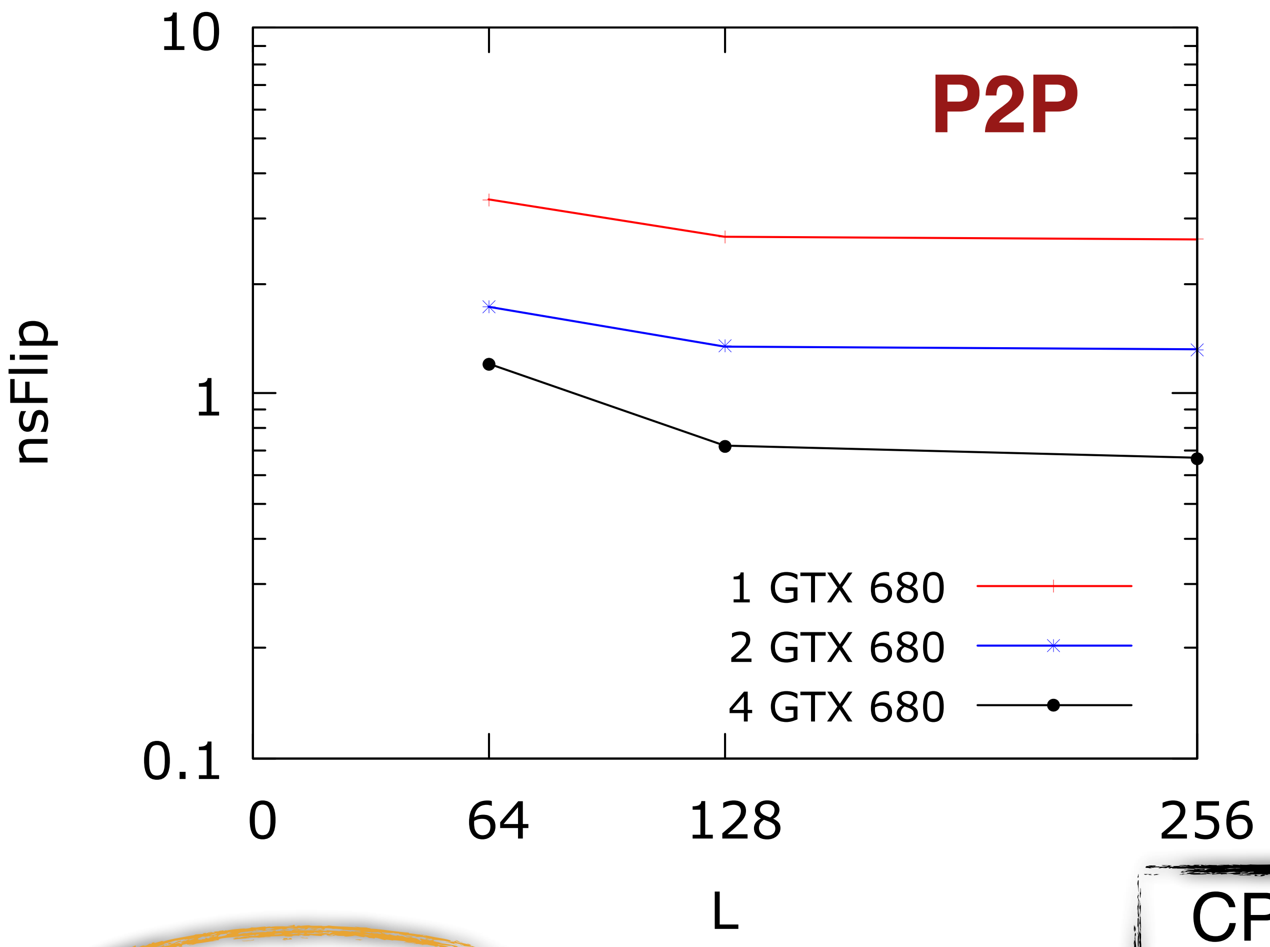


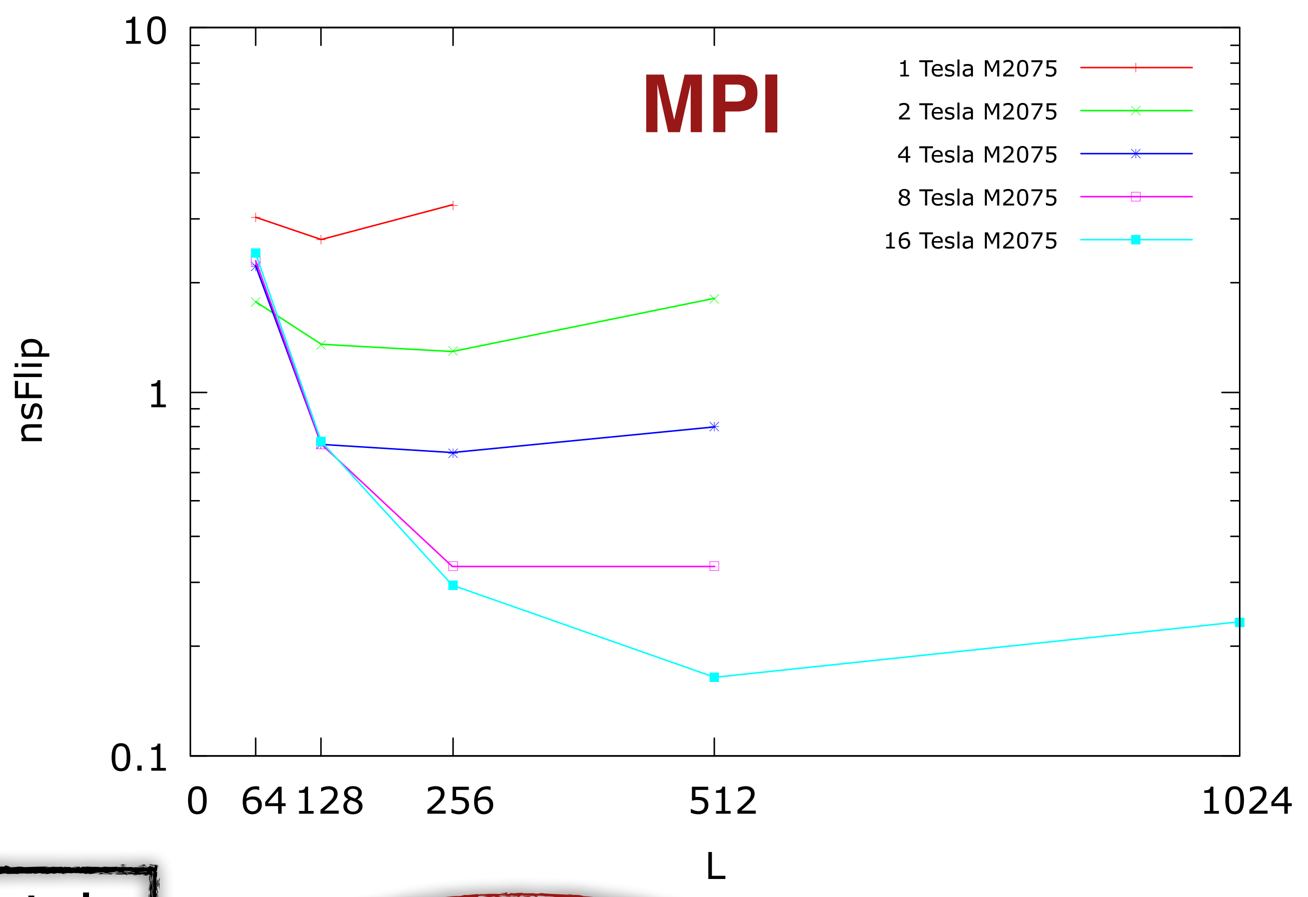
Performance of multi-GPUs simulation

nsFlip : mean time in nanoseconds for single spin update using Heat Bath algorithm in double precision

P2P communication performed on PIOVRA cluster[7]

MPI communication performed on QUonG cluster[8]





$\tau_\sigma \simeq 2.6\ ns$

1 nVidia GTX 680

$L = 128$

CPU Intel Q9650

$\tau_\sigma \simeq 242\ ns$

$\tau_\sigma \simeq 0.16\ ns$

16 nVidia Tesla M2075

$L = 512$

[7]PIOVRA cluster, CHIMERA research group, Physics Department of "La Sapienza", Rome. http://chimera.roma1.infn.it/

[8]Roberto Ammendola, Andrea Biagioni, Ottorino Frezza, Francesca Lo Cicero, Alessandro Lonardo, Pier Stanislao Paolucci, Davide Rossetti, Francesco Simula, Laura Tosoratto, Piero Vicini. *QUonG: A GPU-based HPC System Dedicated to LQCD Computing. Application Accelerators in High-Performance Computing*, Symposium on, pp. 113-122, 2011 Symposium on Application Accelerators in High-Performance Computing (2011).

Summing Up

- **computing on GPUs is an excellent device for simulating huge systems:**
  (about two/three orders of magnitude less than CPU, if well optimized), **as long as:**
  - **the simulated systems possess intrinsic parallelism;**
  - **simulating taking into account the allocation in number of Threads;**
  - **translate arithmetical operations in bitwise ones (like:OR,AND,XOR,NOT, >>,<<) as much as possible (es. $L = 2^n$, %->AND);**
  - **C/C++, Python and MATLAB usage.**

Thank you for your attention!

What is a spin glass?

a magnetic system with disorder and frustation

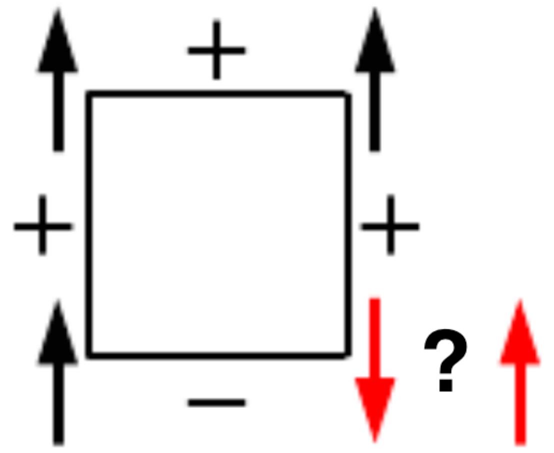
theory:

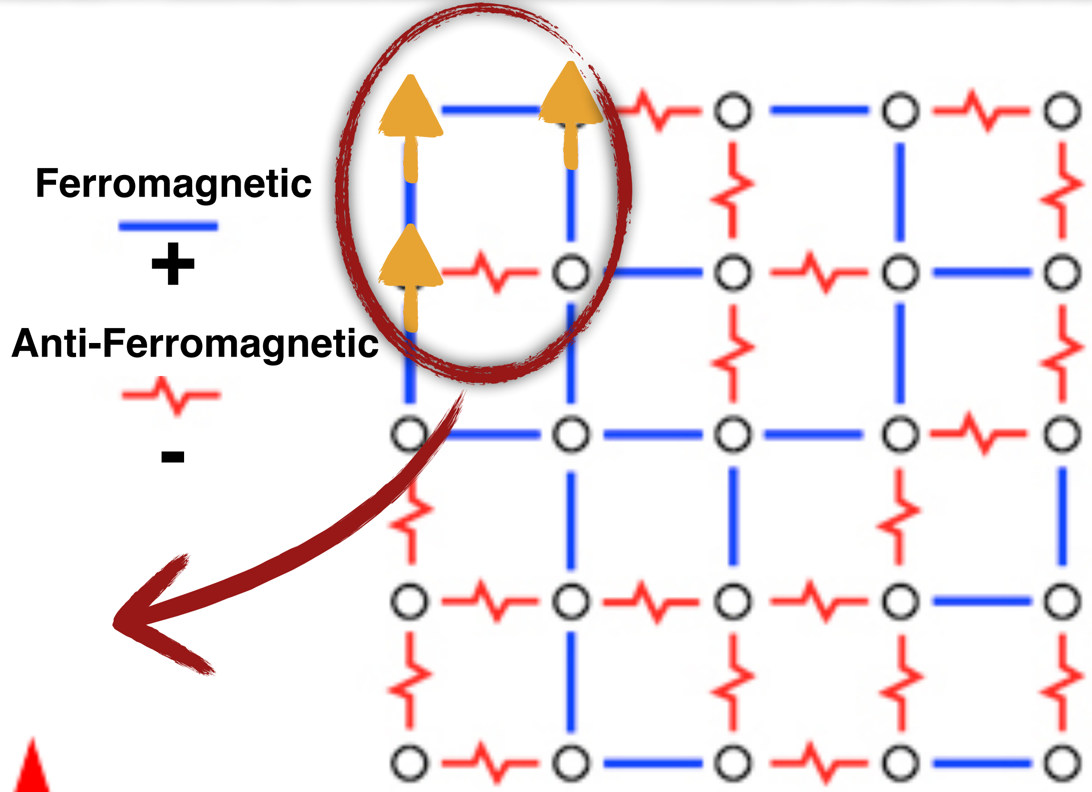
random bonds

$\{J_{ij}\}$ **gaussian, or** $\pm J$

$$H[\sigma] = -\sum_{\langle ij\rangle} J_{ij}\sigma_i\sigma_j$$

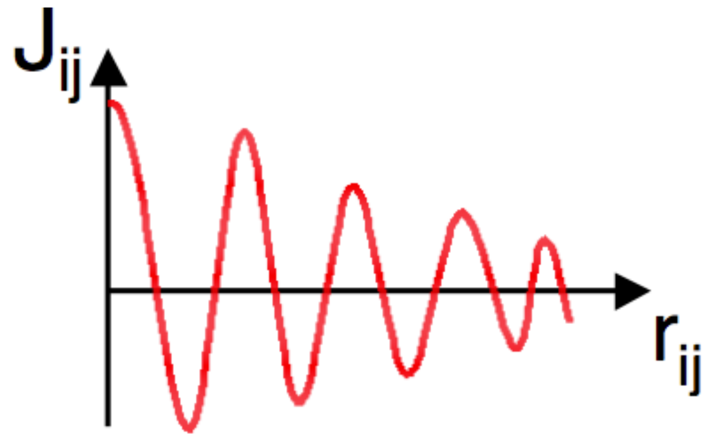
$\sigma_i = \pm 1$ Ising model





"real" spin glass: **RKKY interaction, sign oscillates with distance**

random dilution of magnetic ions
e.g. Mn (magnetic atoms) in Cu (non-magnetic metal)



Data transferring from host (my computer) to device (GPU or cluster of):

- data structures are initialized on the host and copied on the device;
- kernel functions are executed on device;
- execution result is copied from the device to the host.