GPU computing for 2-d spin systems

CUDA vs OpenGL

F. Di Renzo

V. Anselmi, G. Conti

Università di Parma and *INFN, Parma, Italy*

Outline

- GPGPU
  General Purpose computations on Graphics Processing Units
- A look at the current scenario, both hardware (an impressive growth!) and software (OpenGL, CUDA)
- An exercise: (2-d) XY spin model by Hybrid MonteCarlo
  - CUDA (Compute Unified Device Architecture) implementation
  - OpenGL (GLSL) (Open Graphics Library) implementation
- Conclusions

A growing interest for GPGPU ...

In the end: what are PC's built for? what are they useful for?

In recent years the more and more powerful GPU's available on the PC market have attracted attention as a cost effective solution for parallel (SIMD) computing. The birth of the GPGPU acronym is there as a witness.

• At the beginning one could look at GPGPU as one of the many examples of IT-DIY

• Quite soon scientific applications made their entrance

• Actually, big experts are in our community! Egri, Fodor, Hoebling, Katz, Nogradi, Szabo, *Lattice QCD as a video game, Comput. Phys. Comm. 177 (631) 2007*

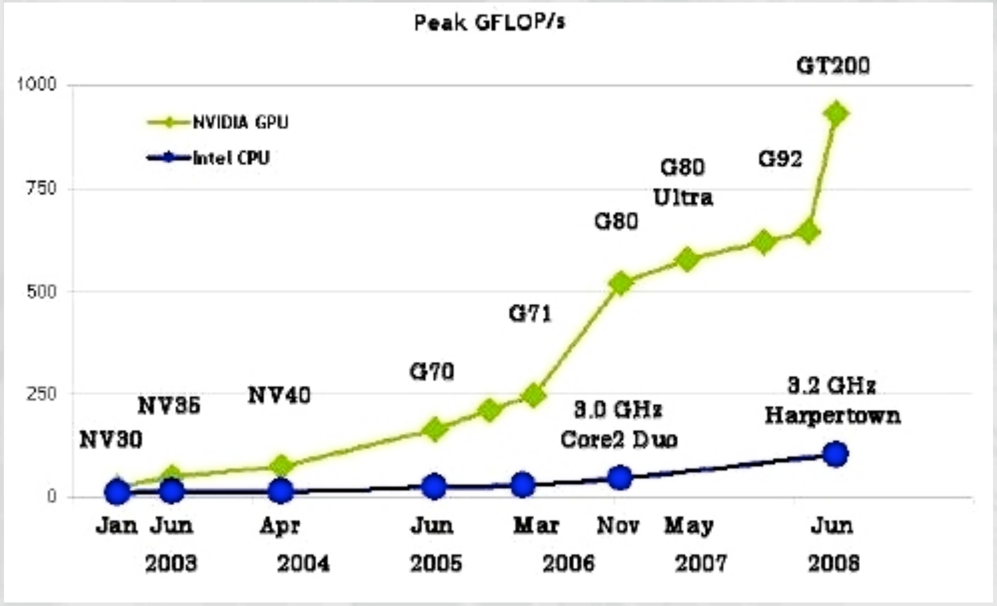
• Big hw companies have been becoming quite iterested players

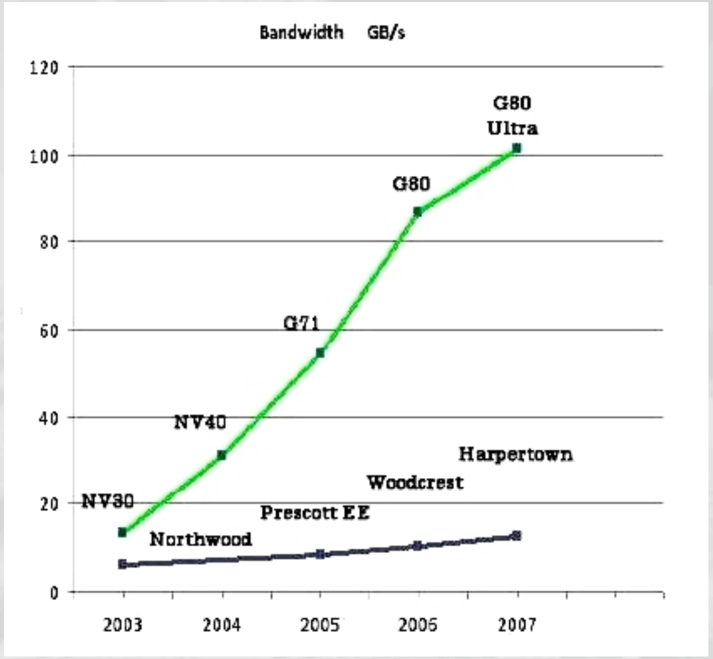
• A good example is provided by *Nvidia* (CUDA hw/sw architecture, Tesla products)

... and a growing power, actually!

Once again, keep in mind: what are PC's built for? what are they useful for?
For sure, GPU documentations provide beautiful plots (most of the figures in the following come from Nvidia)

The GPU bargain: a lot of Flops at low prices, growing faster than in the case of CPU.
Beware: as now, delivered in single precision!



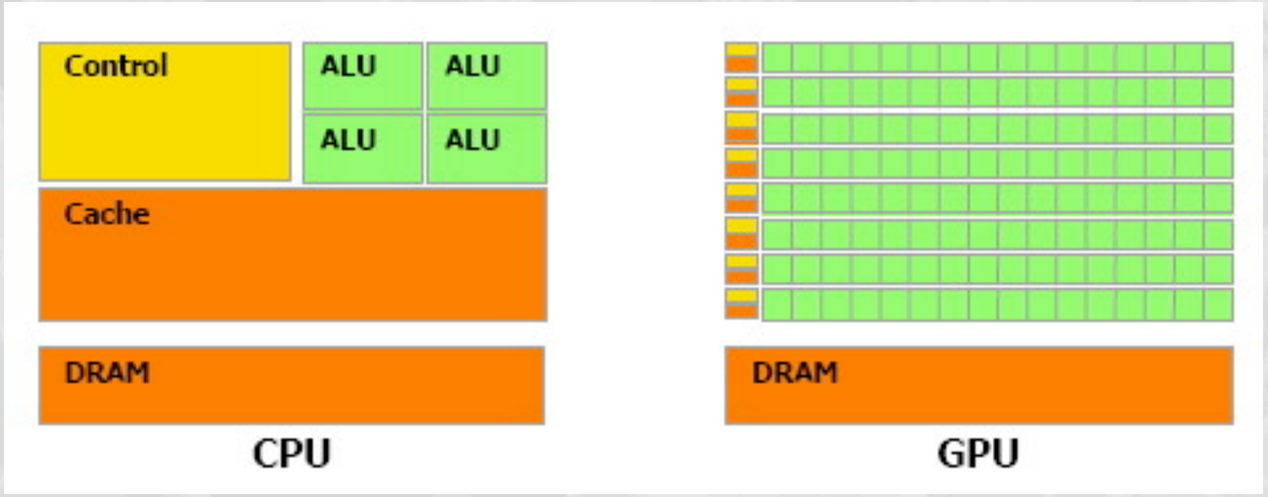


And keep in mind: also bandwith is growing faster than for CPU.

The bottom line: a basic SIMD architecture

And resources allocated in a very different way with respect to CPU

In the end, every computer is from a logical point of view control, data-path and storage. In the end, every computer design is an allocation of resources.



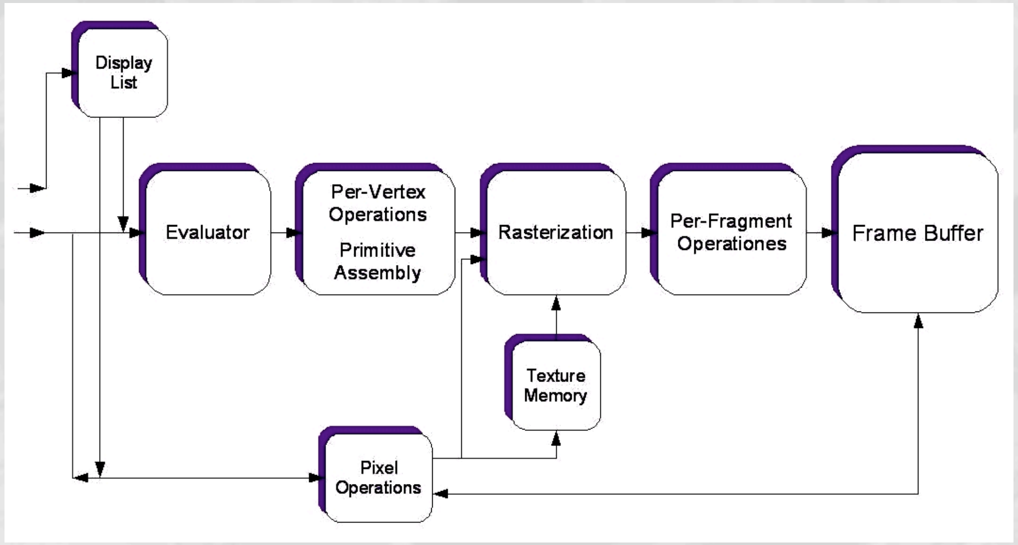
Basic points:

• a SIMD architecture (parallel processing of threads) is well suited for taking care of the mapping of basic graphic elements (pixels, vertices, …)

• memory latencies to be hidden by computations more than by mean of a big cache

A couple of items to be kept in mind

In the end, there is something GPU's are intended for …



The graphic pipeline

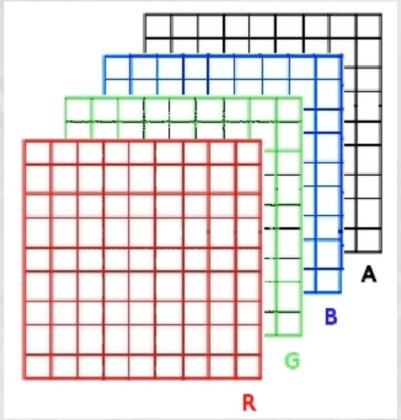
Many steps are needed in order to get a 2-d bitmap for a 3-d image: many SIMD resources are devoted to them. As always in a pipeline, the output of a stage is input for the following one.

Since the fragment shading stage is a very powerful one, usage is made of those resources.

Basic data types are textures

Basically, they are (2-d) matrices. Roughly speaking: images should be eventualy mapped to a bunch of pixels on the screen.

Vector types are a natural choice: RGBA textures account for colours and alpha channel (transparency).



A couple of approaches

There's almost always more than one solution

In the following we will report on two different implentations of the same MC simulation. They are representative of two approaches at our disposal.

✔ The first approach to GPGPU is based on OpenGL (standard graphics library). Basically, *you talk to GPU as you were performing standard image processing*. In other words, your computation enters the graphic pipeline. Standard choice is to enter the fragment stage. Both upload of your code and data and collection of results are a bit funny (again, you are pretending to perform standard graphics).

✔ The second implementation was in the framework of CUDA. Nvidia provides a hw/sw architecture to actually access the GPU as a (parallel) coprocessor.

Our lab: the (2-d) XY spin model by Hybrid MonteCarlo

$$\mathrm{H} = -\sum_{<i,j>} \cos(\theta_i - \theta_j)$$

A disclaimer: preliminary investigations, so that improvements are possible. Having said that, let's pin down some common features of both implementation:

✔ As for generation of momenta: flat random numbers were generated on CPU, conversion to gaussian performed on GPU.

✔ Some operations are critical with respect to single/double precision: global sums of the energy were performed on CPU (as well as Metropolis step).

✔ The core of the parallel computation is the leap-frog integration of equations of motion.

✔ Results were cross-checked with series expansions in the high temperature regime (we are friends of P. Butera, after all) and with a reference HMC (this was done also for acceptance).

✔ The reference serial code was run on an Intel Conroe and was also taken as a reference (at fixed HMC parameters) for performance evaluation.

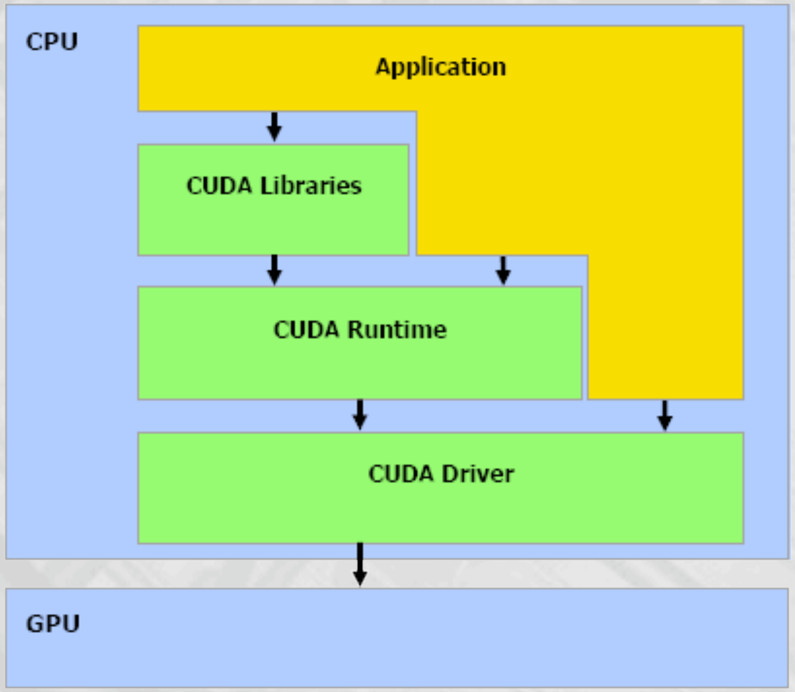
The CUDA environment

Nvidia's entrance to parallel computing

Nvidia calls CUDA a hw/sw architecture which is intended to enable parallel programming on GPU. In the end, they distribute a driver to access the device (list of compatible GPU models extends to last three generations) and a toolkit enabling a programming environment which is basically an extension to C.

Users are provided with

- A nvcc C compiler (code on the device).
- Debugging and profiling tools.
- FFT and BLAS CUDA libraries.
- Documentation.
- A collection of worked-out examples.



A look at a recent GPU architecture

Nvidia wants you to be aware of the basic hw ...

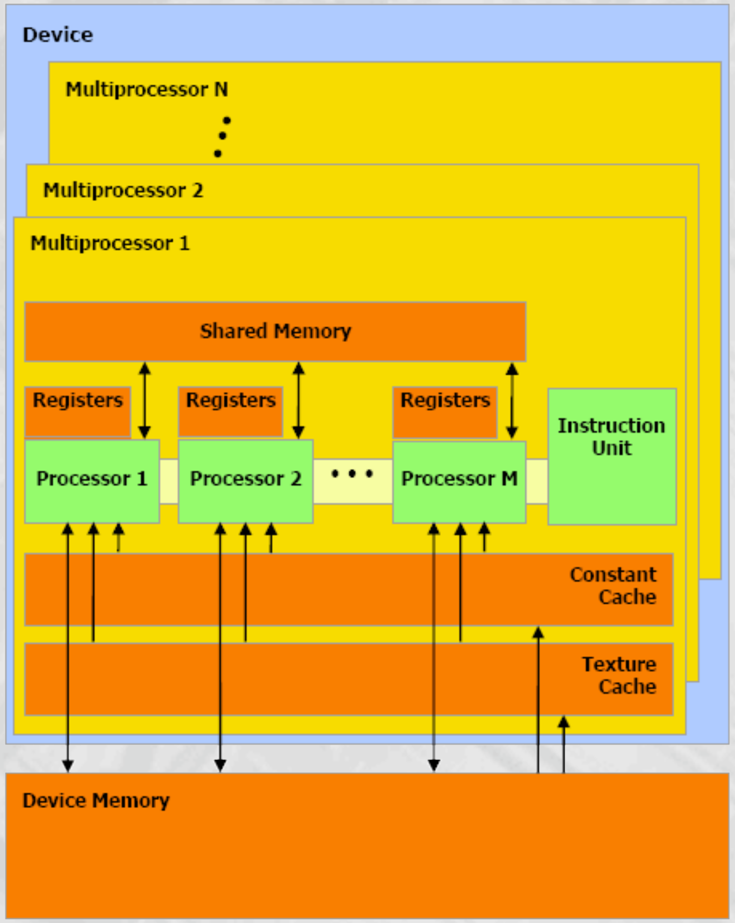
Basically, Nvidia wants you to be aware that a recent GPU is collection of multiprocessors, each made of several processors.

✔ In particular, one should notice the hierarchy of memories at our disposal.

✔ Basic CUDA tool is a driver enabling the access the device in a natural way. In particular, the language enables you to upload/download data to/from the device memories.

✔ You are then entitled to make your choice on where you want your data to reside.

✔ In the end, resources are limited and all the game goes back to their allocation.



CUDA processes organization

Quite a natural threads environment

A process running on CPU (host) can start (several) kernels on GPU (device).

✔ Basic organization is blocks of threads which come in a grid of blocks.

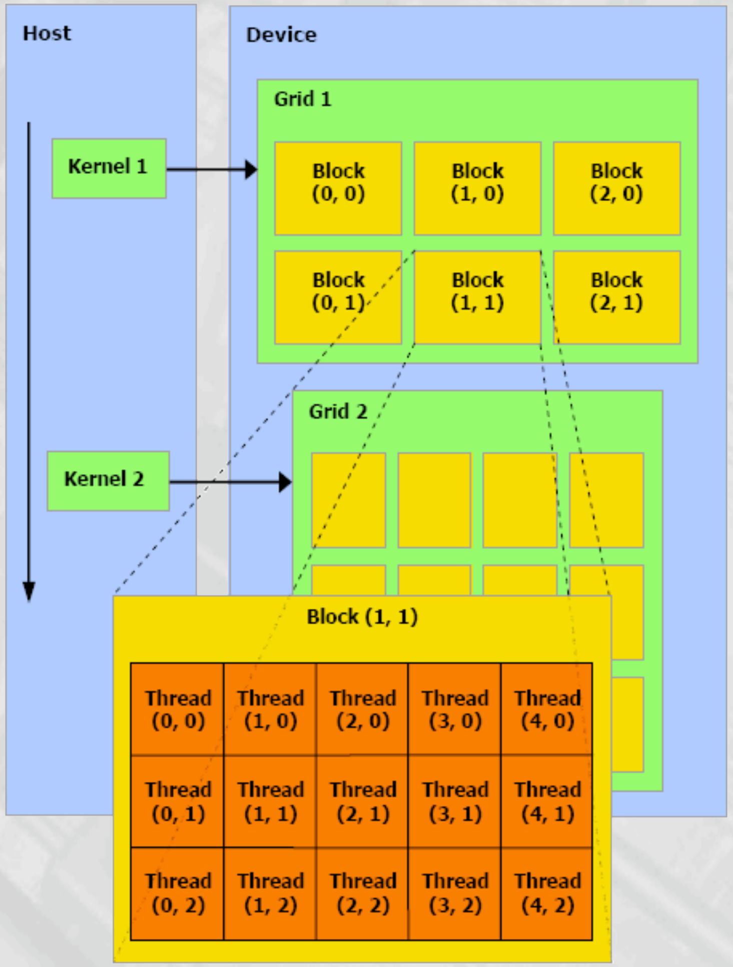
✔ You have commands to upload/download data to/from the device memory.

✔ To execute a kernel your call will be

My_kernel<<<dimG,dimB>>>(my_arg_1, …, my_arg_n)

✔ Threads within a block can be synchronized (they are assigned by the system to the same multiprocessor) and they typically access shared memory.

✔ There are limitations to the number of threads within a block and of blocks within a grid. Then there are limitations imposed by shared memory dimension. Actual allocation of resources is up to the system.



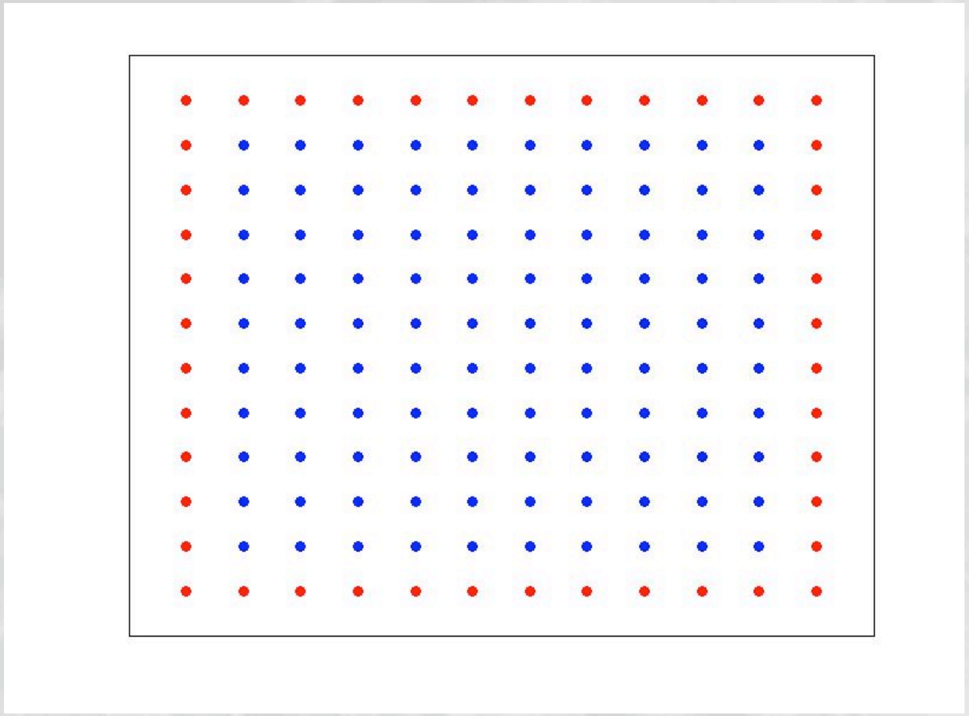
Lattice allocation

First candidate is a quite a natural one

✔ Allocating the lattice can be really straightforward: map a site to a thread, take care of the sublattices borders and try to take advantage of the shared memories.

✔ Nearest neighbors are accessed in a very natural way once you get a border.

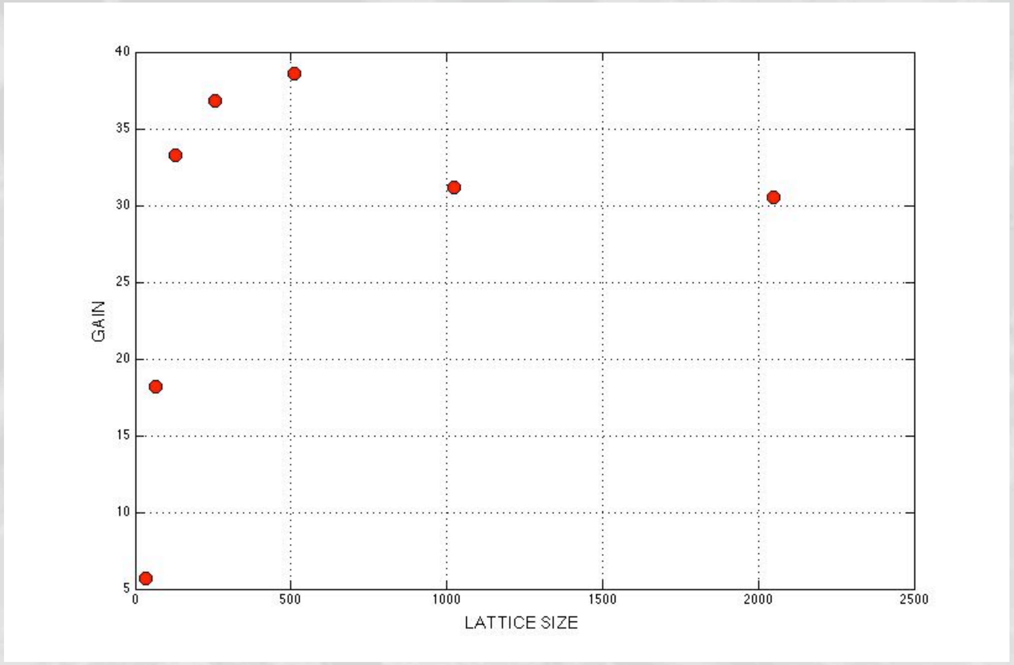
✔ We have to keep your own balance optimizing the usage of resources (well, implementation was so easy and fast that it might well be that we can do better than we did!)



It was easy to gain with respect to the serial code

Again, maybe too easy and optimal code can be still away …

As said, the implementation was very fast, optimization is most probably not complete, but getting substantial gain was easy. Implementation on a Nvidia GeForce 8800GTX. Plot refers to a version which does not even perform gaussian generation step on the GPU.



Let’s now talk OpenGL

The first, tricky approach to GPGPU which has been used

GLSL (Shading Language): another (less direct) extension to C, providing an environment for access to GPU in the OpenGL framework.

✔ Vector variables are a natural choice (vec2, vec3, vec4) . vec4 are an obvious choice for RGBA textures.

✔ Special output variables (e.g. vec4 gl_FragColor): always keep in mind you are supposed to process images.

✔ math library available!

Your code will be essentially devoted to enter the graphic pipeline!

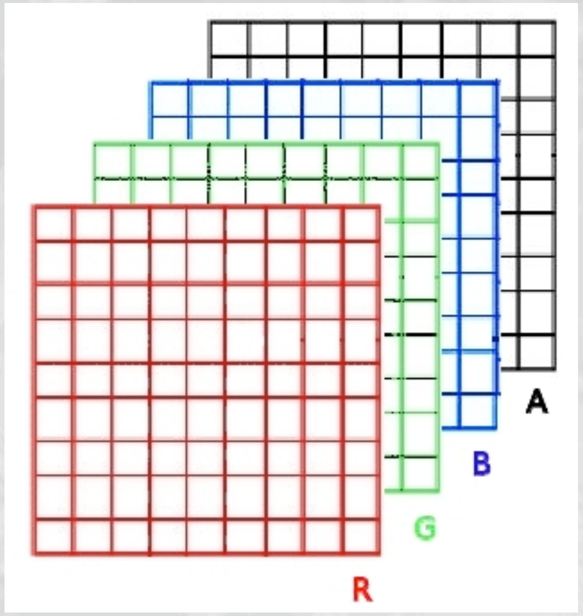
✔ GLSL is initialized runtime, while your kernel has to be “prepared” and then enabled to enter the rendering pipeline.

✔ Input basically boils down to binding textures to texture units, while output is attaching the target texture to a FBO. Finally, remember that we need a filled quad in order to “draw” …

✔ In a pipeline output of one stage is regarded as input for next one. This restricts RW access: PING-PONG technique!

Lattice allocation

Things can be much more natural than they appear (2-d …)



✔ Texels can be your spins.

✔ Texels make a texture like spins make a lattice.

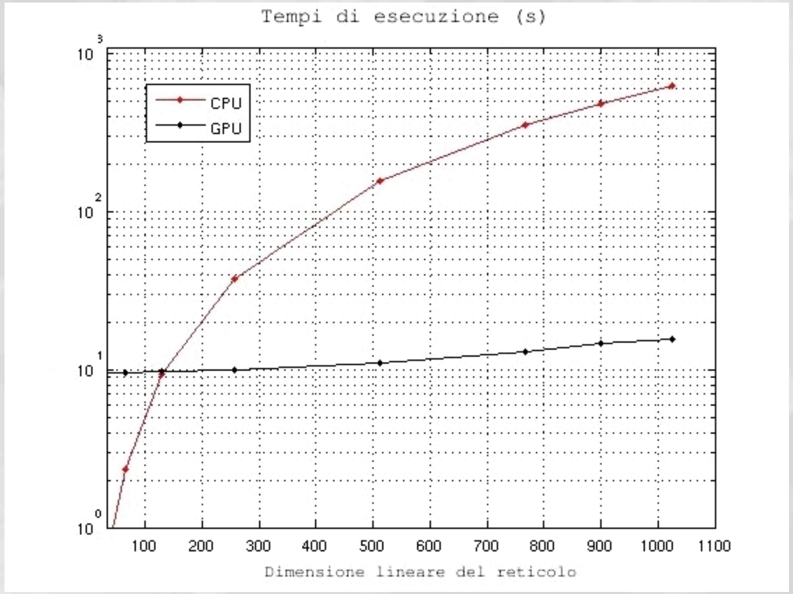
✔ A RGBA texture easily accomodates 4 independent replicas of a lattice (maybe at different temperatures). You only need to be careful on Metropolis acceptance step (easily done: $Y = z\, Y_{new} + (1\text{-}z)\, Y_{old}$).

✔ Nearest neighbors are also easily accomodated: in 2-d they are 4, so they fit in a RGBA texture as well.

2-d systems are actually a gold plated application …

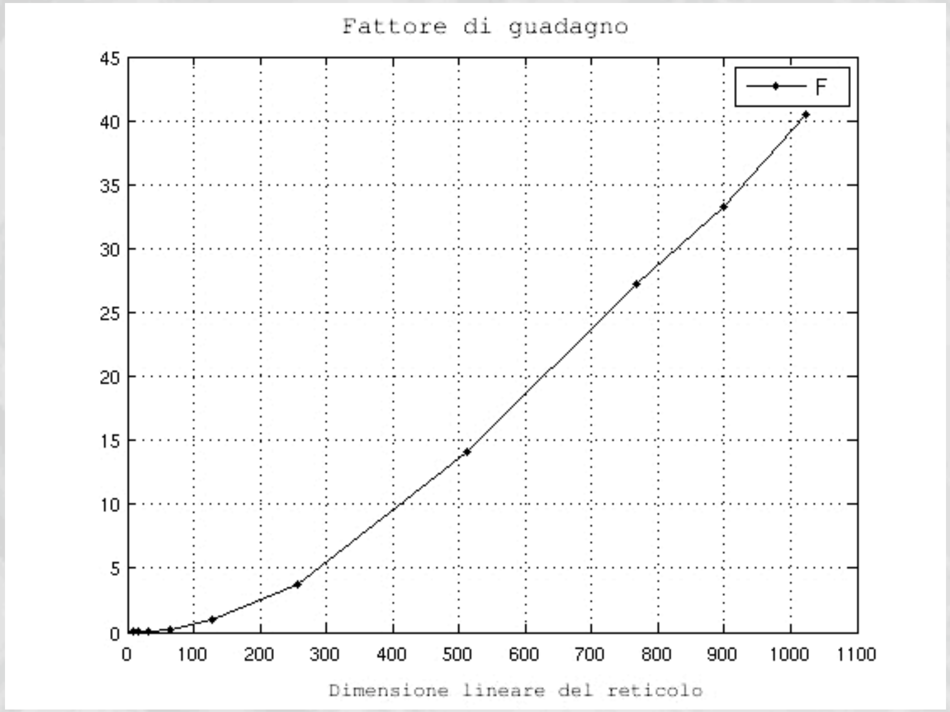
It was even easier to gain!

The more you can fit the better you do …



Despite the fact that we used an old GPU (Nvidia series 6), the speedup was good.

Actually, we obtained the best performance on the largest lattice we could allocate.



↑ Execution times of a fixed number of sweeps vs lattice size

T_{CPU}/T_{GGP} vs lattice size →

Conclusions

- Work was intended as a benchmark exercise (XY model was really only a lab). It might well be that one can do better than this.

- CUDA environment is really friendly: it is easy to get a cost-effective, fairly good performance.

- OpenGL (GLSL) implementation was actually easier than expected! And it was really good performance on a cheap device!