

# A fast particle-mesh integrator for galactic dynamics powered by GPGPUs

Dominique Aubert

Observatoire Astronomique de Strasbourg  
11 rue de l'Université  
67000, Strasbourg, France  
aubert@astro.u-strasbg.fr

Mehdi Amini

CECPV  
Pôle API Bd Sbastien Brant  
Illkirch, France  
amini@cecpv.u-strasbg.fr

Romarc David

CECPV  
Pôle API Bd Sbastien Brant  
Illkirch, France  
david@cecpv.u-strasbg.fr

## Abstract

We ported a particle-mesh N-Body integrator on Nvidia's GeForce 8800 GTX using CUDA. Relying on a grid-based description of the gravitational potential, it can simulate the evolution of  $128^3$  self-interacting 'stars' in order to model e.g. interacting galaxies or clusters. For this purpose we developed a Cuda version of 1/ an histogramming algorithm with CUDPP, 2/ of the resolution of the Poisson equation by means of FFT with CUFFT and multi-grid relaxation, 3/ of an optimized finite-difference scheme to compute the accelerations of stars and 4/ of an update procedure for positions and velocities. This four main steps have been ported on GPU, implying that high arithmetic intensity can be achieved without the requirement of expensive data transfers between the host and the GPU. We present several tests at different resolution, and obtain a factor 2 to 50 in performance depending on the resolution and the type of situations being simulated. It opens bright perspectives to a systematic use of GPUs for more complex N-Body calculations using e.g. AMR techniques and/or dispatched over several GPUs.

## I. INTRODUCTION

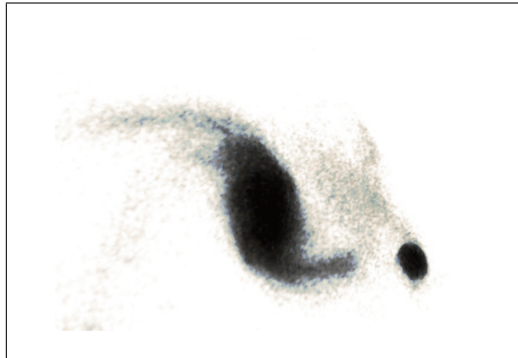


Fig. 1. An example of a PM simulation where a satellite triggers a bar and spiral arms in a galactic disc. This simulation involved  $128^3$  particles with a potential evaluation performed on  $128^3$  cells and ran on a single computer hosting an Nvidia 8800 GTX card.

By essence, astrophysics and cosmology lack of laboratory experiments and from this intrinsic limitation emerges the need to rely on numerical simulation in order to understand the observations. Among the different fields of astrophysics, galactic dynamics – the study of collective motion of stars and gas mainly driven by gravitation – has been a playground for numerical simulations for almost 50 years. More recently, it has been accompanied by numerical cosmology which ignited some of the largest scientific calculations ever made (see e.g. [1]–[3]). It shares with galactic dynamics the intensive use of N-Body integration techniques, which are the main topic of the current paper. Among these techniques, one can cite direct N-Body integration (Particle-Particle or PP hereafter), Particle-Mesh and its extensions (P3M, AP3M), Tree-codes and AMR integrators.

The recent introduction of ready-to-use API for General Purpose Graphical Processor Units (GPGPUs or GPUs hereafter) will strongly impact these domains by providing an easy way to boost the performances

of existing codes. Numerical experiments might be made faster and incidentally larger and hopefully more accurate. Several implementations of PP-methods ([4], [5]) have previously been made available. In such integrators pairwise interactions between stars are computed explicitly and are intrinsically limited by the  $O(n^2)$  complexity, thus limiting the number  $n$  of particles taken into account.

We present here an attempt to fully port a Particle-Mesh integrator on GPU for galactic dynamics. The overall speedup with respect to pure CPU computation spans from 2 to 50 thus promising interesting perspectives for future simulations.

The paper is organized as follows: first the principles of PM integrators are briefly described in section II. Then we provide the details of the parallelization using CUDA. Performances compared to CPU computation are presented in the following section. We discuss the limitations, the advantages and the perspectives in the last section.

## II. AN OVERVIEW OF THE PARTICLE-MESH INTEGRATOR

### A. Overall algorithm

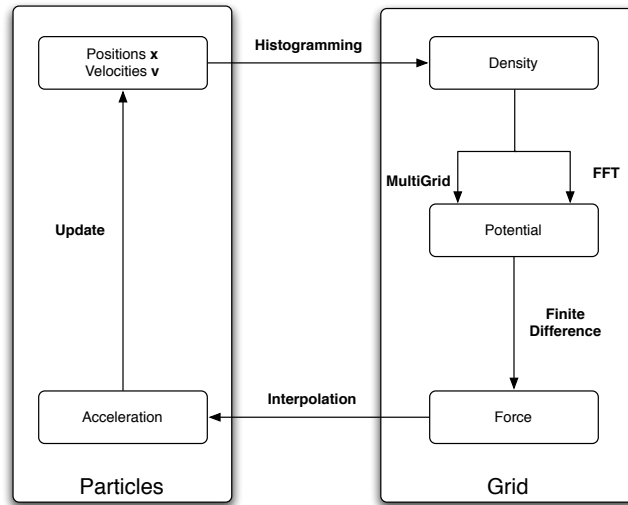


Fig. 2. Flow of operations in a PM calculation. One cycle corresponds to one time step. Operations on the left hand side of the diagram act on 'particles' data, while the right hand side operations act on fixed grid data.

The purpose of N-Body integrations is to simulate the mechanical evolution of a dynamical system due to its inner (and sometimes external) interactions. A well-known astronomical illustration of this problem is the dynamical evolution of a stellar system where stars interact with each other via the gravitational interaction ( $\vec{F}_{12} = -G \frac{m_1 m_2}{d^2} \vec{u}_{12}$  for two stars). For instance, these interactions lead to grand design spirals in galaxies (see fig. 1 for an illustrative case).

To simulate the evolution of the system, the positions and the motions of the stars have to be computed.

The motion of a star at position  $\mathbf{x}$  and velocity  $\mathbf{v}$  is modified as time goes by according to the laws of motions:

$$\mathbf{x}(t + dt) = \mathbf{x}(t) + \mathbf{v} \times dt \quad (1)$$

$$\mathbf{v}(t + dt) = \mathbf{v}(t) + \gamma \times dt \quad (2)$$

$$\gamma = -\nabla\phi, \quad (3)$$

where  $\gamma$  stands for the star's acceleration and  $\phi(\mathbf{x})$  stands for the gravitational *potential* applied to the star at its location. The potential is a scalar field and is related to the spatial distribution of matter via the

Poisson equation:

$$\nabla^2 \phi = 4\pi G \rho(\mathbf{x}), \quad (4)$$

where  $\rho(\mathbf{x})$  stands for the density of stars at a position  $\mathbf{x}$ . The density can be computed from the knowledge of the stars positions, which in turn makes it possible to predict the motion of these bodies through the evaluation of the potential. A handful of methods exist to solve this situation (see e.g. [6], [7] for a review in an astronomical context) and the following sections will focus on the so-called particle-mesh method (PM hereafter).

In numerical simulations, PM-driven N-Body integrations loop over a well-defined sequence of elementary steps. Let us consider a set of  $N$  stars<sup>1</sup> at position  $\mathbf{x}_i$  and velocities  $\mathbf{v}_i$  at time  $t$  and we aim at computing the same quantities at time  $t + dt$ . The integration sequence we set up can be listed as follow and is summarized in figure 2. For critical steps, we mention the algorithms involved.

- 1) Density evaluation : knowing the position of the particles, the density  $\rho$  is evaluated on a 3D regular grid. This is a standard histogramming step. A typical astrophysical situation aims at dispatching  $128^3$  or more particles over a grid that contains  $128^3$  cells or more.
- 2) Potential evaluation: the density  $\rho$  being available, the potential is computed on the same 3D grid via the resolution of the Poisson equation. This is the critical step of N-Body integration and must be made as efficient and 'painless' as possible. We present two standard types of resolution based on FFT and multi-grid relaxation.
- 3) Accelerations calculation: they are directly available from the potential using differentiation. At this stage accelerations are available on the 3D grid.
- 4) Interpolation: the data representation switches back to a particle description. Each body is being assigned an acceleration by interpolation at its position.
- 5) Velocities and positions update: the accelerations lead to the update of the velocities and the velocity update allows to update the positions. Several well-documented schemes exist to perform this step accurately and efficiently. From this point, a new density can be computed and a new time step can be started.

### *B. Motivation of our work*

Running a PM integrator with realistic problem sizes ( $128^3$  and larger) on the researchers desktop machine can be time consuming. Through the current project, we aim at setting the foundations of a fast PM integrator able to run large simulations on desktop machines. It would ease the access to the results of large and more accurate simulations and/or accelerates the mass production of simulated catalogs by quickly providing large sets of small numerical experiments. Also in the prospect of simulations computed on a grid, one requires a code able to produce efficiently results on the smallest number of nodes (ideally 1) and would greatly benefit from any type of acceleration. A large number of simulations would be quickly performed simultaneously on the grid's nodes. To achieve these goals, we implemented all of the steps described in section II-A on GPU with Cuda. We noticed that the SIMD programming mode and the libraries of the Cuda toolkit could fit the steps of our method :

- 1) Density evaluation: Density/ $\rho$  computation : CUDA histogramming routines are available (see [8]) but considering the size of the situations handled here (32000 to more than two million bins) we developed our own SIMD histogram computation using parts of the CuDPP library.
- 2) Potential evaluation: It is usually achieved with FFT or multi-grid relaxation. Parallelizations of both have been widely studied and implemented (see [9], [10]). We developed both versions on the GPU, using the CuFFT API (similar to FFTw) for FFT or writing the whole multi-grid solver.

<sup>1</sup>For sake of simplicity we will consider that the stars share the same mass  $m$ , but the following could easily be generalized to non uniform distributions of masses.

- 3) Accelerations calculation: accelerations are directly available from the potential using derivation. We adapted this entire step to the GPU and optimized it from a SIMD point-of-view while taking into account data locality problems.
- 4) Velocities and positions update: This step is quite similar to the previous one and parallel by nature as each particle is being assigned an acceleration and therefore a velocity and a position.

These four steps have been ported on GPU and, more important, the entire sequence has been ported on GPU in order to limit performance losses due to data transfer from/to the CPU or main memory. That means that, as soon as the input data (initial positions and velocities) are calculated or read from a file, the data is sent to the GPU *once and for all* and needs to be brought back on the host only to be written to the output file. We describe the parallelization and the GPU port in details in the next section.

### III. PARALLELIZATION OF A PM INTEGRATOR WITH CUDA

#### A. CUDA

The PM integrator was implemented on Nvidia's GeForce 8800 GPUs. The GE8 architecture is now well-known. Let us briefly remember it consists in 16 groups (or *multi-processor*) of 8 scalar processors, for a total of 128 processors on a single card. Each multi-processor can execute up to 32 threads simultaneously at a frequency of 675 MHz. A floating-point operation (flop) takes typically 2 cycles to be performed, therefore one can expect a GPU to compute  $32 \times 16 = 512$  flops in 2 cycles, or 256 flops/cycle on average at 675 MHz. In comparison, a Core 2 Duo is able to compute 16 flops/cycle but at higher frequency. In the end, performances of up to 150-350 GFlops may theoretically be expected from 8800 GTX, and depending on the type of operations computed. It may be an order of magnitude more powerful than the best CPUs available in 2008.

Programs executed on GPUs are called *kernels*. Kernels are executed in parallel on the GPU by simple so-called *threads*. For numbering sake, threads are grouped in *blocks*. Blocks themselves are grouped in a *grid* of blocks.

Before any kernel execution, the data must be sent from the memory of the machine hosting the GPU to the device's own memory. The memory hierarchy of the GPU is organized as follows :

- *global memory*, not cached, 768 MB on our GeForce 8800 GTX. Each thread can access this memory.
- *shared memory*, 16KB, several per graphic device, shareable among threads of a given block. Using this memory is the only way threads can *communicate*
- *registers*, belonging to individual threads

#### B. Histogramming

Knowing the positions of the  $N$  particles, the density has to be evaluated on 3D regular grid. Several schemes exists such as nearest-grid point (NGP), cloud-in-cell (CIC) or triangular shaped cloud (TSC) (details can be found in [6]). For this first attempt to port PM on GPUs, we choose to use the simple NGP assignment procedure. In this approach, each particle fully belongs to a single cell. Therefore,  $H_i$  being the number of particles in the grid cell  $i$ ,  $x_p$  the position of the  $p$ -th particle and  $\Delta x$  the cell size, the histogramming is performed following:

```

for  $p = 1$  to  $N$  do
   $i \leftarrow x_p / \Delta x$ 
   $H_i \leftarrow H_i + 1$ 
end for

```

This simple assignment scheme turns out to be unsuitable to an SIMD calculation using GPU. First the whole histogram must be accessible in memory since one cannot predict in which bucket a given particle will fall. Since a typical simulation deals with at least a million of cells, it would imply to perform the assignment directly in global memory. The lack of atomic operations in this memory on our GPUs forbids

us to simply split the histogramming loop over separate threads. Even though such operations are available on the next generation of Nvidia GPUs, two methods were implemented to overcome these issues for the current project.

The first one consists in splitting the histogramming routine in two parts. The cell assignment (i.e. finding the cell a given particle belongs to) is performed on the GPU, and it leads to a data array with the cell indexes. Then this array is copied to the host memory and the histogram incrementation is performed using the CPU. Then, the histogram is sent back to the GPU. This method is simple but seems at first glance limited by latencies and transfer rates between the main memory and the graphic device. As shown in section IV, it turns out to be nevertheless the most interesting option in some of the situations we tested.

The second method takes advantage of the CUDPP library developed by [11] and provided by NVidia. The principle is described in figure 3 and consists in computing first the cumulative histogram then the histogram itself. First, an array containing the indexes of the particles' cells is computed on the GPU. It is sorted by means of the CUDPP radix sort. Once the sorted array is available, each value that differs from the next one is flagged. The flag array is used to compact an array of increasing integers, using the CUDPP compact routine, returning the cumulative histogram. The final histogram is obtained by a simple differentiation of the cumulative histogram.

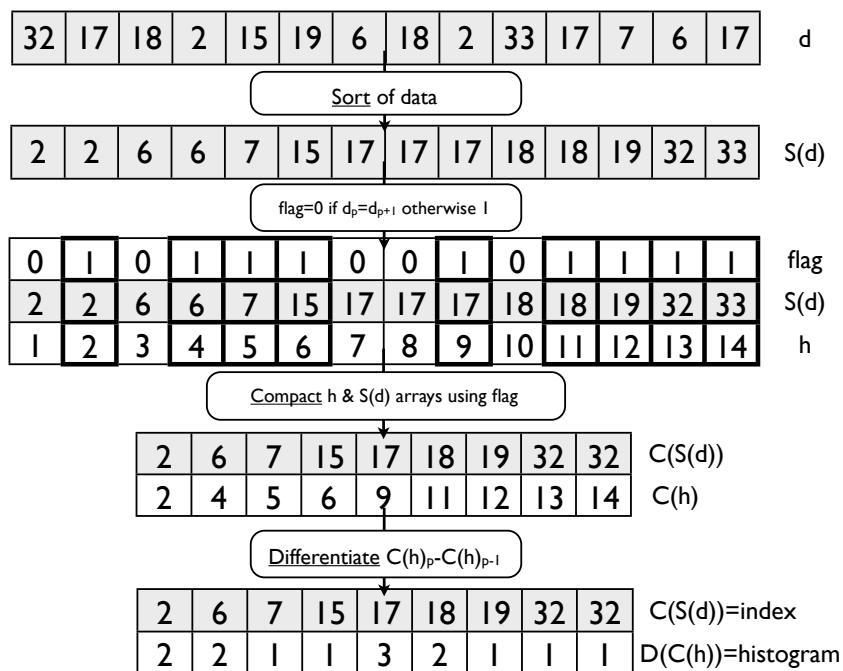


Fig. 3. The histogram computation, obtained from the cumulative histogram. It relies on two CUDPP routines, radix sort and compact (underlined here). Particles lie in the cells given by the  $d$  array. The  $h$  array is a simple array of increasing integers.  $S$ ,  $C$  and  $D$  stand respectively for the sort, compact and differentiation operations (see text for further details).

The four successive operations that lead to the histogram can all be performed on the GPU. CUDPP radix sort and compact rely on the parallel scan algorithm, while the flag and differentiation step are performed by independent threads. This method suppresses completely the data transfers between the host and the GPU during the simulation. However it relies on complex sorting and compact procedures, which are extremely costly in terms of computation time. Situations exist where this method is the fastest

of the two, but let us emphasize that it should rather be considered as our first attempt to develop a full-GPU histogramming method for large grids rather than a definitive algorithm.

Table I summaries the speedups we obtained, step by step. Finding the cell a particle belongs to is denoted as *Cell Assign*. Histogram computations are called *Histo* and *Histo Mix* for respectively the GPU version using CUDPP and the simple CPU-based procedure with data transfers. In table I *Density* refers to the multiplication of the histogram by a certain factor in order to obtain a physical density.

### C. Poisson Solver

The potential is computed from the density using the Poisson equation (see eq. 4) that can be finite differentiated as (in 1D for the demonstration, 3D in our application):

$$\frac{\phi_{i+1} + \phi_{i-1} - 2\phi_i}{\Delta x^2} = 4\pi G\rho_i. \quad (5)$$

This equation can be solved by means of Fourier transforms or by relaxation methods. We implemented both of them.

1) *Fourier transforms*: if periodic boundary conditions are assumed, the Poisson equation can be solved exactly in Fourier space:

$$4\pi G\rho_k = \phi_k \frac{\exp(ik\Delta x) + \exp(-ik\Delta x) - 2}{\Delta x^2} \quad (6)$$

$$= -\phi_k \frac{4 \sin^2(k\Delta x/2)}{\Delta x^2}. \quad (7)$$

From the last relation, the potential can be seen as a filtered (smoother) version of the density. Therefore, obtaining the potential is performed by applying the inverse filter to the Fourier transform of the density and transforming back to real space. Our implementation uses the CUDA FFT libraries, CUFFT, while the filtering in Fourier space is performed by individual threads. The filtering (i.e. the division by the sin term resulting from Eq. 7) is performed by blocks of threads that upload a sub-grid in shared memory.

FFT-based Poisson solvers are popular because they are fast, provide an exact solution to the field equation and are simple to implement. This method has nonetheless some drawbacks. It assumes periodic boundary conditions, which can be severely wrong in case of physical modeling. Zero-padding can limit the effects of periodic artifacts but it results inevitably in a lower resolution. Also in the prospect of a distributed simulation over several GPUs, the slab-based domain decomposition of parallel FFTs is not suited to simulations where particles moves quickly throughout the whole computation volume with strong inhomogeneities (such as the example of fig. 1). That's why we implemented relaxation methods.

2) *Relaxation methods*: Equation 5 can be further modified to an iterative scheme where:

$$\frac{\phi_{i+1}^p + \phi_{i-1}^p - 2\phi_i^p}{\Delta x^2} - 4\pi G\rho_i = \frac{\phi_i^{p+1} - \phi_i^p}{\Delta t}, \quad (8)$$

where  $p$  denotes the  $p$ -th evaluation of the potential and  $\Delta t$  is fictitious time step and the 'real' potential appears as the stationary solution. This iterative scheme is stable for  $\Delta t \leq \Delta x^2/2$ . Choosing the extreme value of  $\Delta t$  leads to a simple iterative formula (in 1D for the explanation, 3D in our application):

$$\phi_i^{p+1} = \frac{\phi_{i+1}^p + \phi_{i-1}^p}{2} - 2\pi G\rho_i \Delta x^2, \quad (9)$$

which can in principle be solved by means of classic smoothers such as Jacobi, Gauss-Seidel or successive over-relaxation. In practice, convergence can be long to achieve and must be accelerated using for instance multi-grid techniques (MG). We implemented the latter in the broader context of the development of a full-approximation storage (FAS) non-linear multi-grid solver, even though the current version included in the PM has been restricted to a simple multi-grid solver, with trilinear interpolation prolongation and a

red-black Gauss-Seidel smoother. The two-color scheme of the iterator ensured that simultaneous threads can compute the next iteration values without causing interferences. Restriction and prolongation were parallelized with each thread being attributed a coarse grid cell: each thread gathers information of fine cells to restrict on coarse grid, while a given thread interpolates on all the finer cells included in its own coarse cell (see Fig. 4). The current version do not rely on shared memory to speedup the restriction-prolongation operations. Relaxation methods do not perform as well as Fourier techniques on the periodic Poisson equation (in terms of speed and accuracy), but unlike FFT-related methods, arbitrary boundary conditions can easily be envisioned. As already mentioned, MG methods can solve non linear version of the Poisson equation such as the one encountered in modified Newtonian gravity (see e.g. [12]). Finally in the prospect of an AMR version of the code with embedded grids, the development of such accelerated iterators is essential.

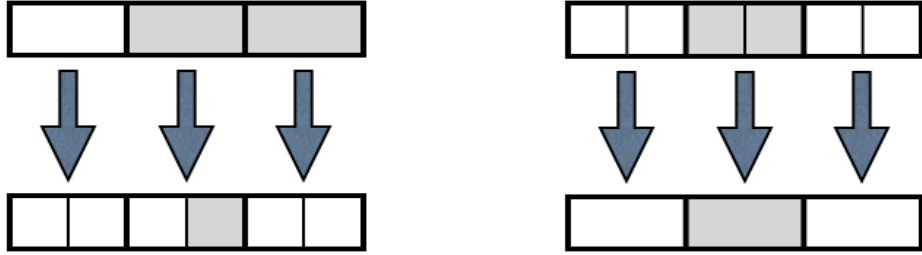


Fig. 4. Prolongation (left) and restriction (right) operations of the multi-grid relaxation method. The operations are shown in 1D for sake of clarity. Each arrow represents a single thread. During the prolongation (left scheme), two coarse cells contribute to a single fine cell (gray) and a thread computes values for the two cells within a coarse one. Similarly, two fine cells contribute to a coarse cell during restriction (right scheme).

In table I *Poiss. FFT* and *Poiss. MG* refer to the two methods for the Poisson resolution and *Potential* refers to a factor multiplication to obtain a physical potential.

#### D. Acceleration, Velocity and position update

The acceleration on the grid nodes is provided by finite differentiation of the potential, following e.g.:

$$(a_{ijk})_x = -\frac{\phi_{i+1jk} - \phi_{i-1jk}}{2\Delta x}, \quad (10)$$

for the acceleration along the  $x$  direction at cell  $ijk$ . From the potential, this operation can be performed on separate threads without any cooperation among them. However, in order to increase this step efficiency, the finite differencing scheme kernel first uploads the data from global memory to shared memory. Since neighboring cells access to close values of the potential, several threads of the same block will access the same region in shared memory. Also, the differentiations in the three spatial directions differ as they follow different schemes to access the memory. Three different parallelization strategies were set up in order to favor coalescent memory access for this purpose (see Fig. 5).

The assignment of an acceleration to a given particle is usually performed in terms of interpolation. The interpolation scheme should be identical to the cell assignment scheme used for the density evaluation. Here we used the simplest of these schemes (NGP), therefore each particle feels an acceleration equals to the acceleration computed in the cell it belongs to. The finite difference steps and interpolation are called *forceX,Y,Z* in table I (the acceleration and the force differs only by a factor, namely the mass).

Finally, position and velocities must be updated. We use the leapfrog scheme, where velocity and positions are computed in a staggered fashion. The acceleration serves to compute the velocity at half a

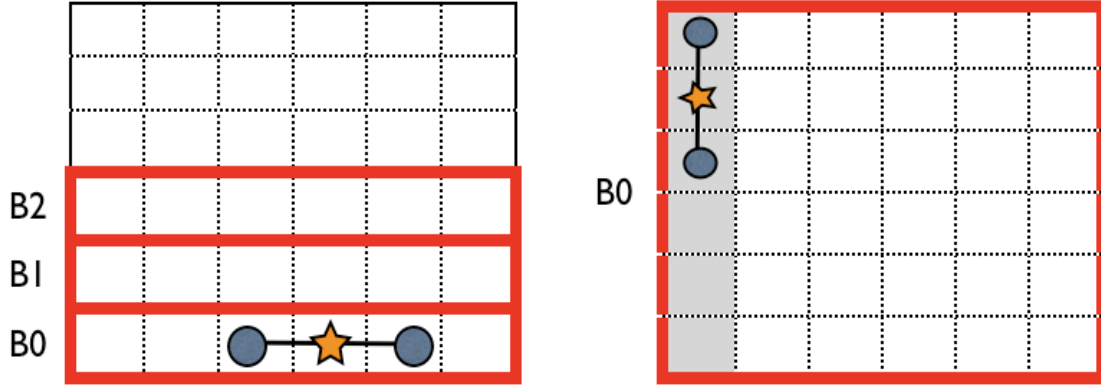


Fig. 5. Threads and blocks configuration for the finite difference scheme (see Eq. 10). *Left*: finite difference scheme along the memory's storage direction. Cells limits are shown as dotted lines. Shared memory within blocks is shown using thick red lines, each thread computes the acceleration in a single cell (star) using the information stored in neighbors (circles). Three blocks are shown here. *Right*: finite difference scheme orthogonal to the memory's storage direction. A block covers the entire plane (shown in thick red line) and a given thread computes values for an entire column (shown in gray). According to our timings, these schemes optimize the usage of memory coalescence.

time step

$$v_{t+\Delta t/2} = v_{t-\Delta t/2} + \Delta t \times a, \quad (11)$$

and the positions are updated following

$$x_{t+\Delta t} = x_t + \Delta t \times v_{t+\Delta t/2}. \quad (12)$$

This update is done by independent threads for each particle as its acceleration is known, leading to its velocity and finally its position. The time step is completed and a whole cycle can be restarted. Usually, state-of-the-art simulation includes adaptive time step. Here we chose at the current to use a fixed time step along the course of the simulation, but such a feature will be included in future versions.

These updates are called *updateVelX,Y,Z* in table I while timings for the positions updated are gathered in *updatePos* in the same table.

## IV. PERFORMANCES

### A. Setup of the experiments

The subsequent experiments were performed on three sets of initial conditions. First a Plummer sphere ([13]) where the particles are distributed isotropically. The velocity distribution is chosen in order to balance the gravity and sphere remains coherent over long period of time. Second, we simulated an exponential disk, where particles are distributed in a thin plane and velocities are similar to the one measured in real disc-like galaxies. This experiment is set up to be unstable by nature, allowing spiral arms to develop for instance. The third type of simulations consists simply in particles randomly distributed in a cubic space with random velocities. Even though, it does not correspond to any real astronomical system, it is similar to cosmological simulations that are ignited from a quasi-uniform distribution of matter. It thus provide insights in the prospect of future cosmological simulations. All the simulations are performed on the same volume using  $32^3$ ,  $64^3$  and  $128^3$  particles/density cells : smaller problems are of little interests while the next power of 8 ( $256^3$ ) surpasses the current capacity some routines such as cuFFT and CUDPP sort and

compact. These larger situations will be addressed on short term as hardware and software improve and our set of simulation already provides a good insight on the perspectives offered by GPU ports.

The time step is chosen in order to achieve an energy conservation of  $\Delta E/E \sim 10^{-3}$  over a time unit, where the energy is defined as :

$$E = \sum_{\text{particles}} \frac{v^2}{2} - \phi(x). \quad (13)$$

For comparison, we developed a CPU version of the PM integrator, written in C, compiled using the Intel C compiler. CPU-driven simulations were performed on Opteron Dual-Core at 2.2 GHz, which also hosts the GPU we used. Let us emphasize that *by no means* the CPU version should be considered as fully optimized version, even though loops were parallelized on the fly by the compiler and common optimization flags were used. The following results should be more considered as a demonstration of the quick gains that can be achieved on GPUs with moderate development skills.

On the CPU, Fourier transforms were performed with the FFTW 3.1.2 library using the single-float precision version and multi-threading was not enabled. On both the GPU and CPU, FFTs were performed using complex-to-complex transform. The multi-grid calculation involved 3 V-cycles with five levels of restrictions, using 5 pre- and post-recursion smoothing steps. It ensures the same level of energy conservation as the FFT calculation  $\Delta E/E \sim 10^{-3}$ . The energy fluctuations were found to be identical between the CPU and GPU versions.

Each of the steps of the integrator is timed separately and ran 1000 times. As explained in section III-B, we used two different histogramming procedures, one that involves a partial CPU calculation ("GPU Mixed Histo" hereafter) and one "Full GPU" version that suppresses all CPU calculations at the cost of complex sorting and compact routines.

### B. Overall performance

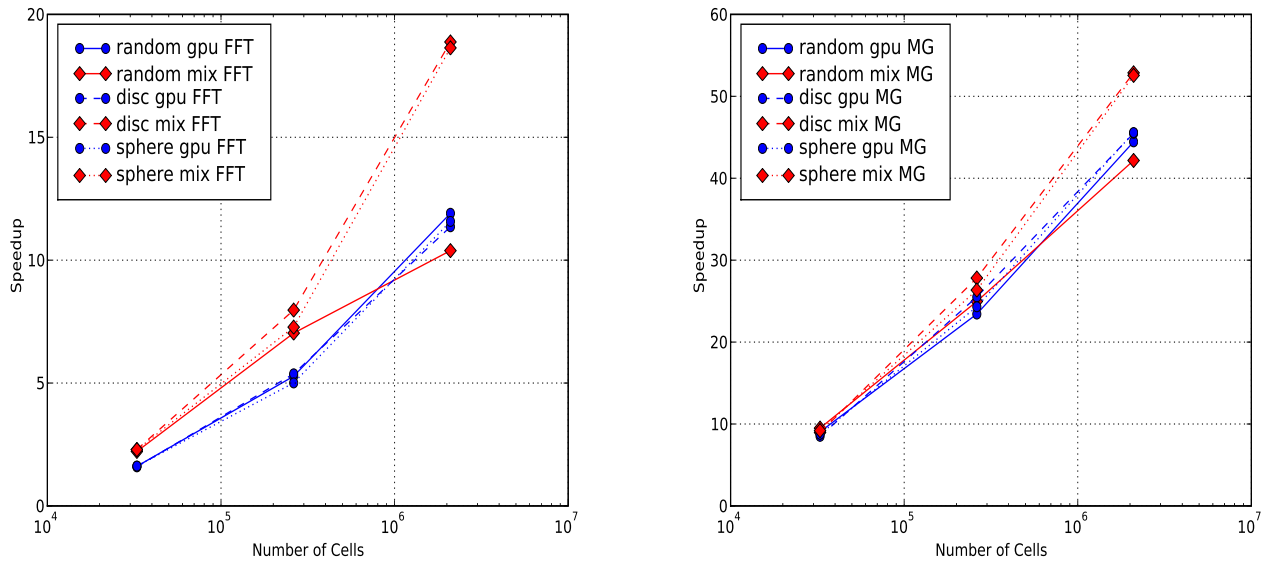


Fig. 6. The overall speedup compared to the CPU version using an FFT (left panel) or a multi-grid (right panel) solver, shown as a function of the number of cells. Please note that y-scales are different in the two panels. Blue lines with circular symbols stand for the full-GPU calculations, while red ones with diamonds stand for the GPU simulations with the histogramming partially performed on the host.

According to our experiments, the GPU versions of the PM integrator can be significantly accelerated, depending of the type of simulation and the number of cells.

As a first broad picture, the figure 6 presents the overall speedup of Full-GPU and GPU-Mixed Histo calculations compared to the CPU version for  $32^3$ ,  $64^3$  and  $128^3$  simulations. Using FFTs for the Poisson equations, the gain measured for the Full GPU ranges from a factor 1.6 to 11.5 while the mixed version experiences a speedup from 2.5 for the smallest versions to 18 for the largest simulation. Using the multi-grid relaxation, the speedup is at least a factor 10 for both versions of the histogramming and reaches a level of 43 for the Full-GPU versions and 52 for the GPU-Mix  $128^3$  calculations. Focusing on the  $128^3$  simulations, the GPU-mixed versions are almost a factor of 2 faster than the GPU versions, while Multi-Grid based calculations are almost equivalent if one considers the Full-GPU or GPU-mixed calculations. One can also note that the random calculations are less efficient when the histogram is performed on the CPU. All these trends result from different limitations and specificities that are detailed hereafter.

### C. Timestep analysis

Before a thorough analysis of the speedup component by component, it should be emphasized that CPU and GPU versions are limited by different bottlenecks. Fig. 7 details the sequence of execution of a given time step and the fraction of computational time spent in each of its sub components for the FFT Poisson solver, while figure 8 present the same quantities for simulations driven by a MG Poisson solver. The CPU version is ‘limited’ by the performance of the Poisson solver: about 75 % of a timestep is spent solving the Poisson equation when the FFT solver is called while this proportion is close to a 100% when the MG solver is used. Considering the FFT version first, the Full-GPU version is limited by the histogramming stage, as expected from an algorithm not naturally suited to SIMD calculations and considering that complex operations (sorting/compact) are involved in the procedure we set up: 60 % of its computational time is spent constructing the density. The Mixed-Histo GPU manages to divide by two the relative importance of histogramming in the overall calculation. Therefore if data transfer between the host and the GPU are hidden by other computations this option can be significantly faster than the full GPU version and Fig 6 seems to confirm this statement. If the Poisson equation is solved by means of the multi-grid relaxation, the fraction of time spent in the histogramming is lowered to levels of 10% to 20%. As a consequence, performing the histogramming on the CPU is not as interesting as it is in the FFT-based simulations : it confirms the global speedups results shown in Fig. 6 where Full-GPU calculations are comparable to mixed calculations when Multi-Grid is called.

### D. Sub components analysis

The absolute timings and speedup of the sub components of a time step are shown in Figs. 9 and 10. Speedups are summarized in table I. From now on, the focus is set on  $128^3$  simulations as they are the closest to realistic size simulations. Histogramming on the GPU and Mixed histogramming are shown side by side even though only one these implementations is used in a given simulation run. The same remark holds for the FFT-based and the MG-based Poisson solvers.

We deduce that the overall speedup of the GPU version compared to the CPU results mainly from a large gain in the resolution of the Poisson equation, moderated by the low efficiency of the histogramming and sustained by the speedups achieved in all the other steps of the calculation.

We now focus on the sub components one by one.

1) *Poisson solver:* From Fig. 9, it can be noted that the resolutions of the Poisson equations are extremely time consuming for the CPU versions and among them the multi-grid calculations are ten times slower than FFTs. The same difference can be noted on the GPU versions, even though the absolute durations are 40 times (for the FFT)/ 60 times (for the Multi-Grid) faster than the CPU versions. Let us emphasize that the FFT-driven resolution of the Poisson equation involves two Fourier transforms in opposite directions and an isotropic filtering. If we consider only the FFTs, our measurements showed that CUFFT is 2, 16 and 40 times faster than FFTW for the  $32^3$ ,  $64^3$  and  $128^3$  experiments. It differs from measurements of [14] with a different device but are consistent with the tests of [15].

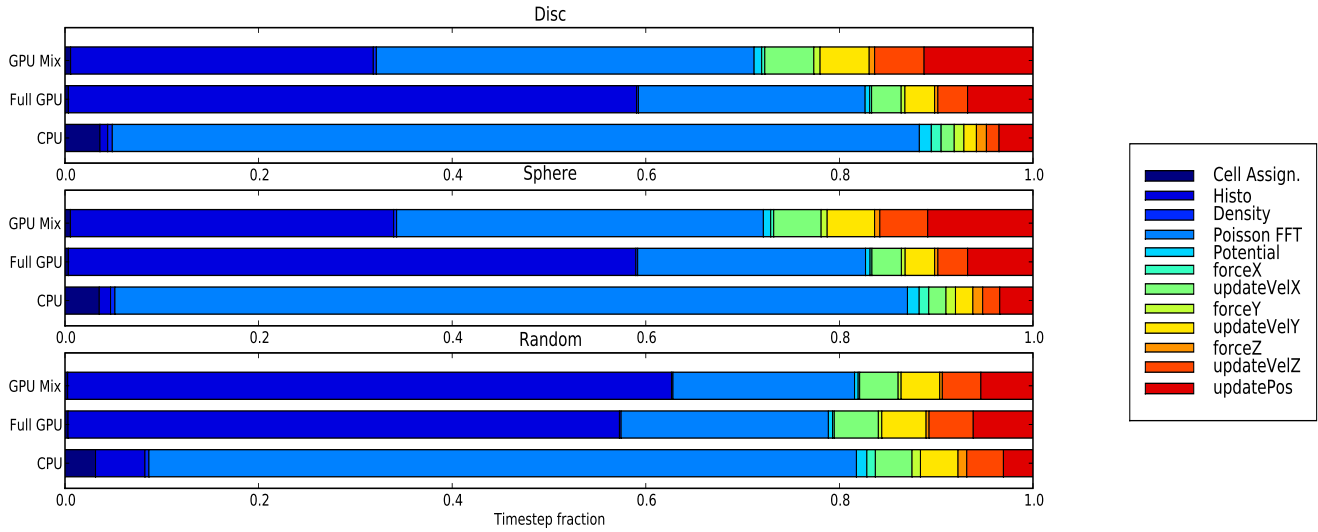


Fig. 7. The time line of a single time step for the three different versions of the PM code (CPU, Full-GPU, and GPU Mixed) measured on the three type of simulations for a  $128^3$  grid driven by a FFT Poisson solver.

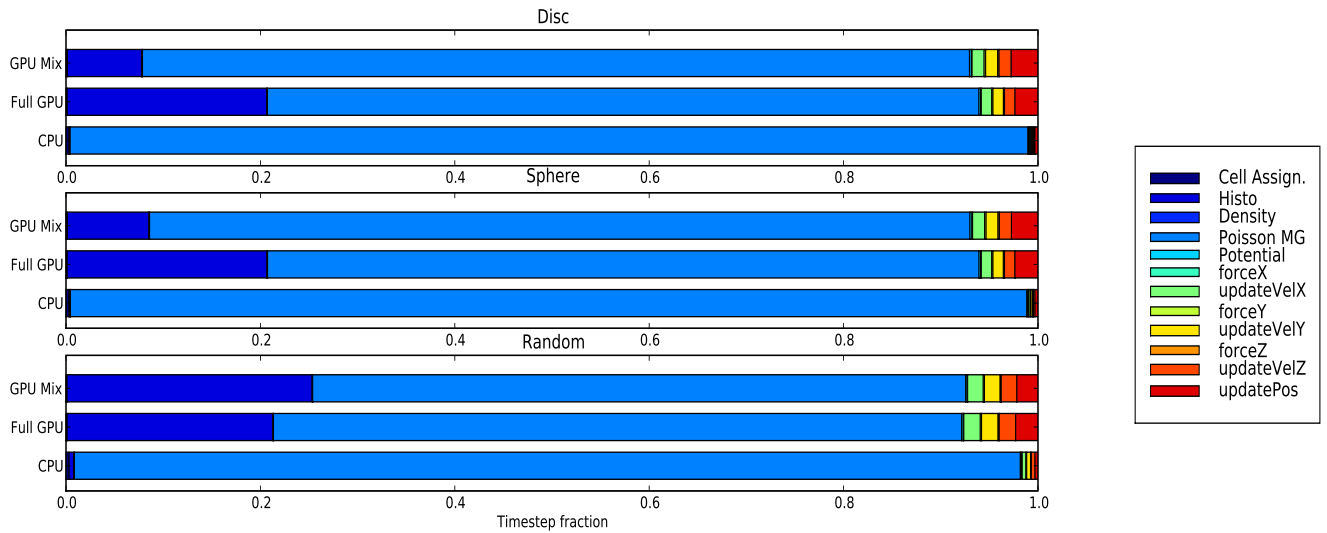


Fig. 8. The time line of a single time step for the three different versions of the PM code (CPU, Full-GPU, and GPU Mixed) measured on the three type of simulations for a  $128^3$  grid driven by a MG Poisson solver.

Important speedups are measured for the multi-grid relaxation, where both GPU and CPU code were written from scratch. Speedup is achieved using the high-level parallelism of the computations involved in the restrictions, prolongations and the Gauss-Seidel iterations. We think greater speedups might be reached by fine-tuning the GPU routines, especially with a greater use of shared memory for the redundant operations of restrictions/prolongation.

2) *Histogramming*: On the downside, no gain can be observed for the histogramming step on the GPUs. Even worse, this computation can be 5 times slower on  $128^3$  simulations and can go down to 10 times slower on the full GPU version for  $32^3$  particles simulations (not studied here otherwise). The GPU-mix histo-version performs the most expensive step on the host but still, the data transfer (transfer rate and

latency) results in this computation step being twice slower than computation performed only on the CPU. The mixed version fill the gap for the sphere and disc simulations, but are at least 15% slower than the CPU versions.

Interestingly, the random case simulations (i.e. the closest to a cosmological simulation) are much more favorable to the Full-GPU version in terms of histogramming: even though it is more complex, the fact that particles are spread in all the computation box implies that CPU cache cannot be as efficient as it writes data in memory (cache misses). Indeed, in the two previous cases, the particles were confined in certain sub regions (disc or sphere), ensuring a certain level of cache hits. The GPU histogramming routine is SIMD by nature and its performances do not depend strongly on the particle distribution, while it is clearly not the case for the simpler but more cache-dependent CPU or mixed version.

3) *Accelerations and updates*: All the other stages of the calculation are significantly speeded up on GPU, with speedups ranging from 5 to 120 compared to the CPU versions. We noticed that the speedups of velocity updates increase as the particles are spread in a larger portion of the grid, especially comparing disc simulations to random simulations. To compute the velocity of a particle, a given thread (or the CPU) finds the cell it belongs to and uses the associated acceleration. If particles are strongly clustered, memory access scheme gets close to neighboring accesses to the memory, which are more likely to be cached on the CPU. Consequently, speedups are larger as the CPU fails to cache the memory accesses, as it is the case for the random simulation. Also, computing the force/acceleration is faster along the  $x$  direction and results from the storage of the grids in memory which favors certain directions in terms of contiguous memory access among threads.

TABLE I  
SPEEDUPS FOR  $128^3$  SIMULATIONS

Type	Random	Sphere	Disc
Cell Assign.	118.0	117.6	117.6
Histo	1.1	0.2	0.2
HistoMix	0.9	0.7	0.5
Density	28.4	27.8	28.4
Poiss. FFT	40.5	40.3	40.3
Poiss.MG	61.1	61.3	61.2
Potential	29.7	29.7	29.7
forceX	59.7	59.6	59.8
updateVelX	9.9	6.7	5.0
forceY	29.3	29.3	29.3
updateVelY	10.1	6.8	4.8
forceZ	34.9	34.9	34.8
updateVelZ	9.9	6.6	4.8
updatePos	5.9	5.9	5.9

## V. CONCLUSION AND PERSPECTIVES

Using the CUDA API developed by Nvidia for its hardware we developed a Particle-Mesh N-Body integrator that runs on common graphic devices. All the steps of the algorithm were ported on this hardware and we obtain speedups that ranges from 2 to 50 depending on the size of the problem and the techniques involved along the course of a simulation. Histogramming (for the density computation) and the solver for the Poisson equation are the critical part of the implementation. Thus, we developed a full GPU histogramming algorithm and solve the Poisson equation by means of FFT and Multi-Grid

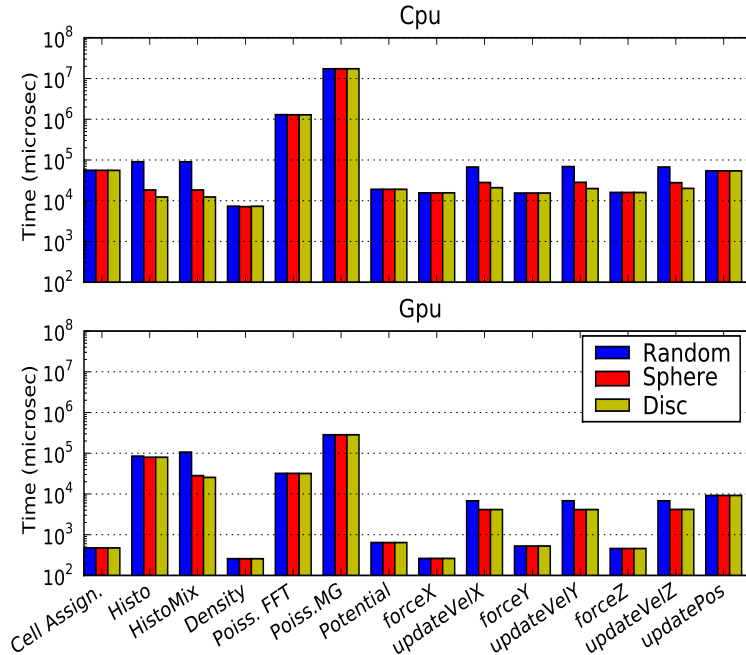


Fig. 9. Absolute timings for the different stages of a single time step for the CPU and the GPU versions. For a given run, histogramming is performed either on the GPU (Histo) or on the CPU (HistoMix). Also the Poisson equation is solved using FFT or Multi-Grid relaxation (MG). For the CPU version, the Histo and HistoMix routines are identical.

relaxation. For a typical data set ( $128^3$  particles dispatched in a disc), we achieve speedups of 20 for FFT based calculations and up to 50 for Multi-Grid based ones.

Our experiments showed that histogramming is more efficient on the CPU if particles are confined to well-defined sub-volume by means of cached access to the memory. On the other hand, GPU histogramming is as efficient as the CPU version if particles are spread in all the computing volume such as situations observed in cosmological simulations. For large problems, the resolution of the Poisson equation is at least 40 times faster on the GPU using FFT and 60 times faster using multi-grid relaxation. Even though histogram computation is hardly accelerated on GPU, the performance achieved on the Poisson resolution plus the 10-100 speedup obtained on all the other steps (cell assignment, acceleration computation/interpolation, velocity/position update) results in a significant acceleration of the code.

In a near future, we plan first to assess larger problems ( $256^3$ ,  $512^3$ ) in order to reach astrophysically relevant resolutions. Using devices with larger memory capabilities and supporting atomic operations, it should be within our reach using the Multi-Grid Poisson solver and an improved version of the histogramming routine. For instance, along the course of this paper's redaction, [16] described an efficient histogramming algorithm implemented in CUDA, for plasma physics simulations. Our project would clearly benefit from such an algorithm. If such large situations can be handled, running large multi-GPU simulations would be the next step to reach the billion particles with GPU speedups. However, it remains still unclear how network based communication may lower the speedups obtained with a single device. From a methodological point of view, such a code should be extended to adaptive grid refinement techniques, where no *a priori* GPU-related limitations exist to prevent such a development.

From an astrophysical point of view, the results shown in the current paper are clearly encouraging. Furthermore, this PM development is accompanied by two other codes : GPU version of a non-linear FAS

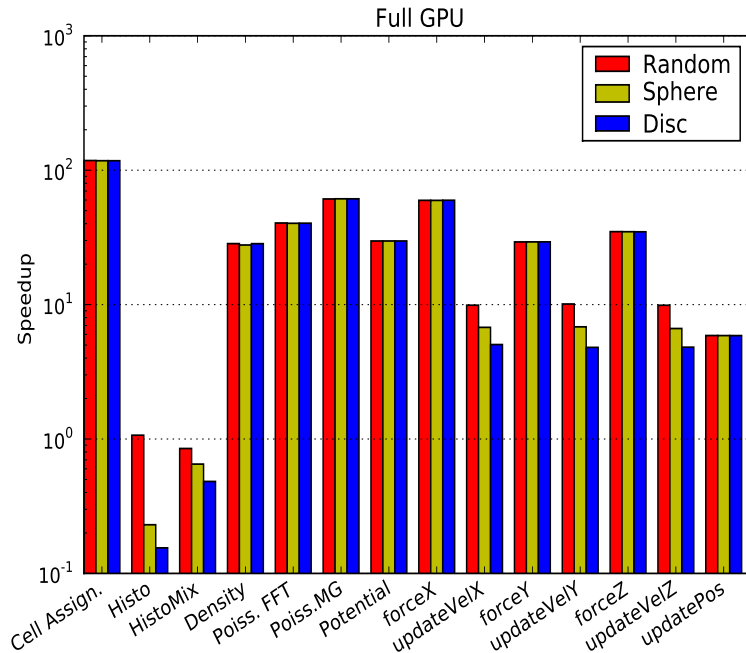


Fig. 10. Speedup for the different stages of a single time step for the the GPU-Mix and full GPU simulations on a  $128^3$  grid.

Multi-Grid solver for the modified Newtonian dynamics and the CUDA transcription of a cosmological radiative transfer code. For these two codes speedups range from 10 to 80 compared to CPU versions. For astrophysicists, it opens the bright perspective of an easy access to HPC-like calculations in their desktop machines with a set of well-suited API's enabled for GPUs.

#### ACKNOWLEDGMENTS

D.A. would like to thank R. Teyssier for patience and fruitful comments along the course of this development and T.Keller for technical support. This work is supported by a grant from the *Conseil Scientifique* de l'Université Louis Pasteur.

#### REFERENCES

- [1] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce, "Simulations of the formation, evolution and clustering of galaxies and quasars," *Nature*, vol. 435, pp. 629–636, Jun. 2005.
- [2] I. T. Iliev, P. R. Shapiro, G. Mellema, H. Merz, and U.-L. Pen, "Simulating Cosmic Reionization," *ArXiv e-prints*, vol. 806, Jun. 2008.
- [3] R. Teyssier, S. Pires, S. Prunet, D. Aubert, C. Pichon, A. Amara, K. Benabed, S. Colombi, A. Refregier, and J.-L. Starck, "Full-Sky Weak Lensing Simulation with 70 Billion Particles," *ArXiv e-prints*, vol. 807, Jul. 2008.
- [4] S. F. Portegies Zwart, R. G. Belleman, and P. M. Geldof, "High-performance direct gravitational N-body simulations on graphics processing units," *New Astronomy*, vol. 12, pp. 641–650, Nov. 2007.
- [5] T. Hamada and T. Iitaka, "The Chamomile Scheme: An Optimized Algorithm for N-body simulations on Programmable Graphics Processing Units," *ArXiv Astrophysics e-prints*, Mar. 2007.
- [6] R. W. Hockney and J. W. Eastwood, *Computer simulation using particles*. Bristol: Hilger, 1988, 1988.
- [7] E. Bertschinger, "Simulations of Structure Formation in the Universe," *ARAA*, vol. 36, pp. 599–654, 1998.
- [8] R. Shams and N. Barnes, "Speeding up mutual information computation using nvidia cuda hardware," *Digital Image Computing Techniques and Applications, 9th Biennial Conference of the Australian Pattern Recognition Society on*, pp. 555–560, Dec. 2007.
- [9] T. Sorensen, T. Schaeffter, K. Noe, and M. Hansen, "Accelerating the nonequispaced fast fourier transform on commodity graphics hardware," *IEEE Transactions on Medical Imaging*, vol. 27, no. 4, pp. 538–547, Oct. 2008.

- [10] D. Göttsche, R. Strzodka, and S. Turek, "Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in fem simulations," Int. J. Parallel Emerg. Distrib. Syst., vol. 22, no. 4, pp. 221–256, 2007.
- [11] M. Harris, S. Sengupta, and J. D. Owens, Parallel Prefix Sum (Scan) with CUDA (In Collection). Addison Wesley, 2007, 2007.
- [12] R. Brada and M. Milgrom, "Stability of Disk Galaxies in the Modified Dynamics," ApJ, vol. 519, pp. 590–598, Jul. 1999.
- [13] J. Binney and S. Tremaine, Galactic Dynamics: Second Edition. Galactic Dynamics: Second Edition, by James Binney and Scott Tremaine. ISBN 978-0-691-13026-2 (HB). Published by Princeton University Press, Princeton, NJ USA, 2008., 2008.
- [14] H. Merz. (2008) CUFFT Vs FFTW Comparison [www.science.uwaterloo.ca/~hmerz/CUDAbenchFFT/](http://www.science.uwaterloo.ca/~hmerz/CUDAbenchFFT/) -.
- [15] P. Demores. (2007) GPU Benchmarking [www.cv.nrao.edu/~pdemores/gpu/](http://www.cv.nrao.edu/~pdemores/gpu/).
- [16] G. Stantchev, W. Dorland, and N. Gumerov, "Fast parallel particle-to-grid interpolation for plasma pic simulations on the gpu," Journal of Parallel and Distributed Computing, vol. 68, no. 10, pp. 1339–1349, Oct. 2008.