Georges-Henri Cottet, Jean-Matthieu Etancelin, Franck Perignon, Christophe Picard, Florian De Vuyst & Christophe Labourdette

Is GPU the future of Scientific Computing ?

Volume 20, n° 1 (2013), p. 75-99.

<http://ambp.cedram.org/item?id=AMBP_2013__20_1_75_0>

Annales mathématiques Blaise Pascal **20**, 75-99 (2013)

Is GPU the future of Scientific Computing ?

Georges-Henri Cottet
Jean-Matthieu Etancelin
Franck Perignon
Christophe Picard
Florian De Vuyst
Christophe Labourdette

Abstract

These past few years, new types of computational architectures based on graphics processors have emerged. These technologies provide important computational resources at low cost and low energy consumption. Lots of developments have been done around GPU and many tools and libraries are now available to implement efficiently softwares on those architectures.

This article contains the two contributions of the mini-symposium about GPU organized by Loïc Gouarin (Laboratoire de Mathématiques d'Orsay), Alexis Hérault (CNAM) and Violaine Louvet (Institut Camille Jordan). This mini-symposium was an opportunity to explore the upcoming role of hardware accelerators and how it will affect the way applications are designed and developed.

As the main issue of the mini-symposium was graphical cards, this document contains contributions about two feedbacks on the behavior of different numerical methods on GPU:

- ones on particle method for transport equations,
- the other on Lattice Boltzmann Methods for Navier–Stokes equations, Finite Volume schemes for Euler equations and particles methods for kinetic equations.

Keywords: GPU, méthode particulaire, EDP, Mécanique des Fluides, interaction, visualisation, calcul instantané, volumes finis, méthode Lattice Boltzmann, méthode particulaire, programmation multicœur.
Math. classification: 35L05, 65M08, 76M25, 76N15, 76P05, 97N40.

Le GPU est-il le futur du calcul scientifique ?

Résumé

Ces dernières années, de nouveaux types d'architectures basés sur les processeurs graphiques ont émergés. Ces technologies fournissent d'importantes ressources computationelles à faible coût et faible consommation d'énergie. Les nombreux dévelopements effectués sur le GPU ont alors permis la création et l'implémentation de logiciels sur ce type d'architecture.

Cet article contient les deux contributions de ce mini-symposium GPU organisé par Loïc Gouarin (Laboratoire de Mathématiques d'Orsay), Alexis Hérault (CNAM) et Violaine Louvet (Institut Camille Jordan). La premiere concerne les méthodes particulaires pour les équations de transport, la seconde concerne la résolution des équations de Navier-Stokes et des équations d'Euler.

Contents

1. Particle method on GPU 77
1.1. Previous work 77
1.1.1. Particle method 77
1.1.2. OpenCL computing 78
1.2. Particle method with OpenCL 79
1.2.1. Advection step 80
1.2.2. Remeshing step 81
1.3. Performances and results 81
1.3.1. Level set test case 81
1.3.2. Computational efficiency 82
1.4. Conclusions 84
2. Some examples of instant computations of fluid dynamics on GPU 84
2.1. GPU computing with instantaneous visualization and interaction in CFD. Experience feedback 84
2.1.1. Instantaneous computing and interactive visualization 84
2.1.2. Impact on the design of numerical methods 85
2.1.3. Impact on the data structures 94
2.1.4. General experience feedback and concluding remarks 95
2.2. Videos of instant GPU computations on youtube 97
Acknowledgements 97
References 97

1. Particle method on GPU

by *Georges-Henri Cottet, Jean-Matthieu Etancelin, Franck Perignon and Christophe Picard.*

In this part we present a graphics processing unit (GPU) implementation of a particle method for transport equations. More precisely the numerical method under consideration is a remeshed particle method. Not only remeshing particles makes simulations more accurate in flows with strong strain, but it leads to algorithms more regular in term of data structures. In this work, we develop a Python library using GPU through OpenCL standard that implements this remeshed particle method which already shows interesting performances.

1.1. Previous work

1.1.1. Particle method

In the present work, we focus on solving advection equation (1.1) by means of remeshed particle method

$$\partial_t u + \operatorname{div}(au) = 0\,. \tag{1.1}$$

Remeshed particle methods can be seen as forward semi-Lagrangian methods. For each time step, they consist in 2 sub-steps. An advection step where particles, carrying local masses of u, are advected with their local velocity, followed by a remeshing step where particles are restarted on a regular grid. In the advection step, one solves the following system of different equations

$$\frac{\mathrm{d}\tilde{x}_p}{\mathrm{d}t} = a(\tilde{x}_p, t)\,. \tag{1.2}$$

The remeshing step is performed with the following general formula

$$u_g = \sum_p \tilde{u}_p W\left(\frac{x_g - \tilde{x}_p}{h}\right)\,. \tag{1.3}$$

In the above formulas $\tilde{x}_p$, $\tilde{u}_p$ represent the particle locations and weights after advection, and x_g, u_g their location and weights after remeshing on a regular grid.

The algorithm to solve equation (1.1) for a time step $t = n\Delta t$ can be summarized as follows:

- Initialize particle locations and weights : $\tilde{x}_p^n \leftarrow x_g$, $\tilde{u}_p^n \leftarrow u_g^n$
- Solve equation (1.2) with a 2nd (or 4th) order Runge Kutta scheme $\tilde{x}_p^n \leftarrow \tilde{x}_p^n + \Delta t a^n(\tilde{x}_p^n + \frac{\Delta t}{2} a^n(\tilde{x}_p^n))$
- Remesh particles on grid : $u_g = \sum_p \tilde{u}_p W\left(\frac{x_g - \tilde{x}_p}{h}\right)$.

Note that depending on the problem at hand, particle velocities are either computed analytically or interpolated from grid-values.

In general this scheme is extended to any dimension using tensor-product formulas for the remeshing kernels. In [7], an alternating direction algorithm was proposed where particles are successively pushed and remeshed along axis directions. This method reduces the cost of the remeshing step, allowing to use high order interpolation kernels with large stencils. In this work we use the 6-point kernel M_6' [1]

$$M_6'(x) = \begin{cases} \frac{1}{12}\left(1-|x|\right)\left(25|x|^4 - 38|x|^3 - 3|x|^2 + 12|x| + 12\right) & \text{if } 0 < |x| < 1 \\ \frac{1}{24}\left(|x|-1\right)\left(|x|-2\right)\left(25|x|^3 - 114|x|^2 + 153|x| - 48\right) & \text{if } 1 < |x| < 2 \\ \frac{1}{24}\left(3-|x|\right)^3\left(5|x|-8\right)\left(|x|-2\right) & \text{if } 2 < |x| < 3 \\ 0 & \text{if } 3 < |x|\,. \end{cases}$$

A distinctive feature of remeshed particle methods is the time step does not depend on the number of particles. This allows to perform accurate high resolution simulations with large time steps.

1.1.2. **OpenCL computing**

OpenCL is an open standard for parallel programming of heterogeneous systems [8]. It provides application programming interfaces for managing hybrid platforms containing many CPUs and GPUs and a programming language based on C99 for writing instructions executed concurrently on the OpenCL devices.

OpenCL applications must define an execution model by setting a host program that executes on the host system and send OpenCL kernels to devices using a command queue. A kernel contains executable code that concurrently runs on devices compute units which are called work-items. A memory model need to be explicitly defined to manage data layout in the memory hierarchy. Details of these models such as work-item number in a work-group or memory access pattern have a very strong impact on program efficiency.

1.2. **Particle method with OpenCL**

Particle methods have already been implemented on GPU. In [11], a two dimensional solver for bluff body flows is developed using OpenGL and CUDA. The method allows to deal with particle distributions of up to 1024^2 at a speed greater than 20 fps. The accuracy of GPU computations was also addressed by comparing GPU results with high resolution double precision benchmark calculations on CPU.

Our implementation of the method presented in section 1.1.1 uses different abstract layers by means of a Python class hierarchy in order to have a well-defined program structure easy to use and develop. Computations are not performed on the host side of the program but on the devices in different kernels, unnoticed by user.

According to the method, the algorithm is split into two parts namely an advection and a remeshing step. These two parts are repeated several times to perform a dimensional splitting for each simulation time step. We depict in figure 1.1 the algorithm for one splitting direction. For simplicity we take the velocity as a constant. In the general case, the velocity field is computed once at every time step.

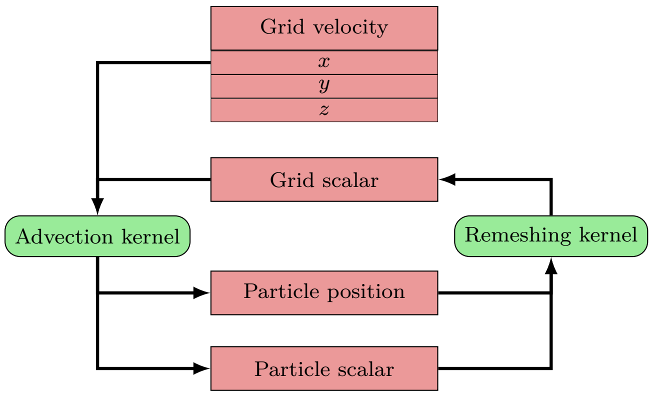


Figure 1.1. Execution layout on GPU. Memory objects are depicted in red and OpenCL kernels in green

The main constraints for implementations on GPU are to make a proper and optimized use of the memory size and bandwidth. In fact, in a tree-dimensional case, each particle needs 6 floats in global memory to be completely defined. For example, 1024^3 particles need 6GB memory in simple precision[1]. This problem will be tackled in future work using several GPUs. The memory access is detailed in the following sections. In figure 1.1, memory objects are either OpenCL Buffers or Images. The current work is to determine which type of object is best suited to our algorithm.

In order to take advantage of the splitting algorithm, the different one dimensional problems are distributed among work-items. In our implementation, one work-item is not in charge of one grid point but of one line of grid points in the splitting direction. Advantages of this distribution are detailed in the following parts. For example, 1024^3 particles will be computed over 1024^2 work-items, each one in charge of 1024 particles.

1.2.1. **Advection step**

In our splitting algorithm, only one component of velocity and particle position in the current splitting direction need to be considered. The other components are respectively unused or leaved unchanged. The particle position variable reduces to a scalar. In our method, particles are created on each grid point and initialized with the value on the grid. Grid points coordinates are re-computed each time from the global OpenCL index space thanks to buit-in functions.

Once particles are created and initialized, evolution ODEs are solved for particle position using the grid velocity field. This is done by means of a 2nd order Runge-Kutta scheme. The problem is to interpolate the grid velocity to compute intermediary steps in the time-stepping scheme because the needed data depend on the velocity field. Therefore the memory access pattern might not be linear, depending on the computation process.

A simple improvement for this point is to make data closer to work-items, by use of a copy of the needed grid data in private memory. Interpolations are then performed in private memory so data are read with the fastest memory access available.

A strong performance improvement was obtained by arranging data layout for the grid velocity. In fact, as data are accessed line by line, we

[1]Today's cards memory ranges between 128MB and 8GB.

make the data contiguous in a direction orthogonal to the splitting direction. Consequently, work-items can together read contiguous data in global memory and then avoid strided accesses. For scalar data, a similar memory layout can be used by transposing data from one splitting direction to the next. This implementation needs further improvements to cope with small private memory, or larger problems. In these cases, a new re-arrangement of tasks is required.

1.2.2. **Remeshing step**

As for advection, memory access pattern is execution dependant since a particle is remeshed on its nearest grid points. On top of that, two different particles can have exactly the same remeshing grid points. We need synchronization within particles to avoid concurrent memory writing access. More precisely, remeshing points overlap for particles that are in the same one dimensional line in the splitting direction under consideration. Thus, synchronization is done when only one work-item works on the line. A simple improvement is to write results in a local buffer and, once all particles are remeshed, copy this buffer into the global memory. This minimizes global memory access for remeshing.

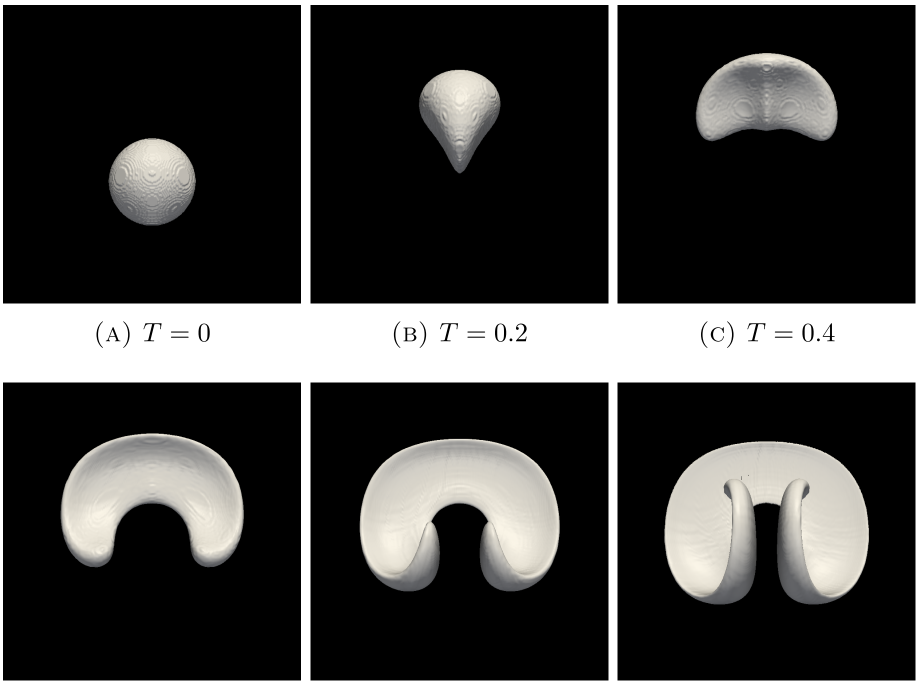
1.3. **Performances and results**

1.3.1. **Level set test case**

Our method is tested with a classical and challenging problem for level set methods, namely a sphere subjected to a incompressible velocity field in a periodic cube $[0;1]^3$. This test consists in the advection of a passive scalar initialized with value $u=1$ inside a sphere of radius 0.15 and $u=0$ elsewhere. The advection velocity is given by the following formula

$$a(x) = \begin{pmatrix} 2\sin(\pi x)^2 \sin(2\pi y)\sin(2\pi z) \\ -\sin(2\pi x)\sin(\pi y)^2 \sin(2\pi z) \\ -\sin(2\pi x)\sin(2\pi y)\sin(\pi z)^2 \end{pmatrix}.$$

One of the tests implemented in [7] and the references therein is presented in Figure 1.2. This simulation is performed with $N = 256^3$ and a time step value which would correspond to a CFL number equal to 25


(a) $T = 0$ (b) $T = 0.2$ (c) $T = 0.4$

(d) $T = 0.6$ (e) $T = 0.8$ (f) $T = 1.0$

Figure 1.2. Iso-surface of level 0.5 at different times, $CFL = 25$, $dt = 0.05$.

at this resolution. Time $T = 1$ is reached in 20 iterations and took 25 seconds.

1.3.2. **Computational efficiency**

OpenCL kernels can be compute-bound or memory-bound. Our advection kernel is memory-bound since the operational intensity equals to 2.25 operations per byte of data accessed from the memory and remeshing kernel is nearly compute-bound with a operational intensity of 9.5.

In figure 1.3, we present profiling results and timings. For larger problems, almost all compute time is spent in kernels that could be optimized. The initialization and host code are sequential codes quite independent of the problem size. The initialization part consists in reading problems data and creating python objects structure. The host code tasks are to set the OpenCL execution layout and to launch the kernels. For 256^3 particles, the whole computation is performed in 25 seconds, which corresponds to 1.25 seconds per time step.

As a comparison, a Fortran/MPI solver performs the same simulation on 4 Intel Core i7 running at 2.4 GHz in 62 seconds, which corresponds to 12.4 seconds per time step. This shows a speedup of nearly 10 against the parallel Fortran code.

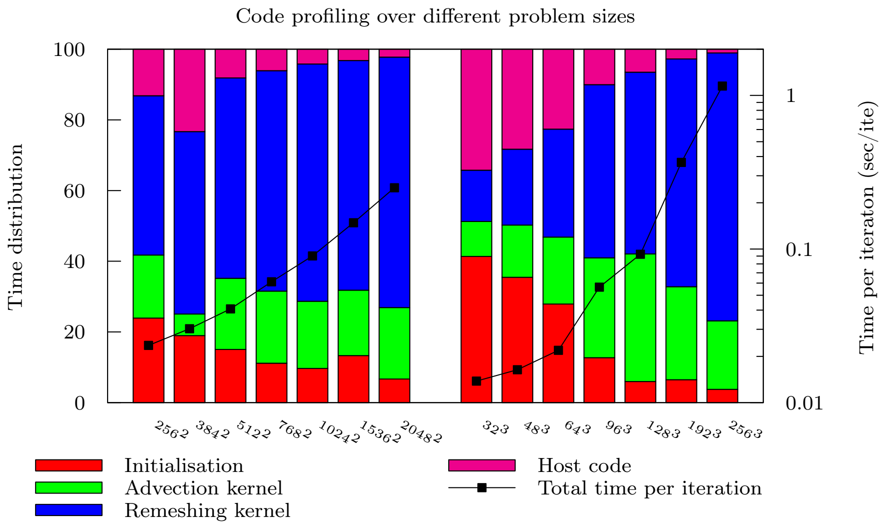


Figure 1.3. Profiling data on ATI Radeon HD 6770M

To give another comparison, the computational times showed in [11] are about 0.048 seconds per time step for one million particle for the whole Navier-Stokes solver in simple precision. In our case, we obtain a computing time of 0.055 for problems of similar size. Note however that the problems are rather different, since the problems considered in [11] were two-dimensional and used lower order remeshing schemes but involved non-local field evaluations (through FFT).

1.4. Conclusions

In this work we showed implementations of 3D particle methods in GPUs. A splitting algorithm together with a high order remeshing kernel were considered for a transport equation discretized on a single GPU with about 16 million particles.

Ongoing work concerns further optimizations of our code and its Python implementation on several GPUs to tackle larger problems.

2. Some examples of instant computations of fluid dynamics on GPU

by *Florian De Vuyst and Christophe Labourdette*

This section is a summary of our experience feedback on GPU and GPGPU computing for two-dimensional computational fluid dynamics using fine grids and three-dimensional kinetic transport problems. The choice of the computational approach is clearly critical for both performance speedup and efficiency. In our numerical experiments, we used a Lattice Boltzmann approach (LBM) for the incompressible Navier-Stokes equations, a finite volume Flux Vector Splitting (FVS) method for the compressible Euler equations and a lagrangian particle approach for a linear kinetic problem.

2.1. GPU computing with instantaneous visualization and interaction in CFD. Experience feedback

2.1.1. Instantaneous computing and interactive visualization

High performance computing (HPC) knows an important growth since recent years. Theoretical peak processing performance and storage capacity in supercomputers gained three or four orders of magnitude in less than a decade. However, scientists and computational engineers would like more flexibility in terms of delay of response and ease of use. Manufacturers develop cluster computers to exceed the petaflop (see the exaflop !), but the cost and planning of very large computations imposes workflow constraints in batch mode.

Recently, the design of manycore processors like graphics processing units GPU, general purpose graphics processing units GPGPU and many-integrated cores MIC allow one to get theoretical teraflop performance into a simple office workstation. This potential flexibility of use with only one user let us imagine new ways of computing and use cases like interactive simulation and instant computations. The applied mathematicians are often little concerned with the very large calculations, they are more interested in the design of methods and algorithms for performing the calculations. That's why we emphasize here on ways of instant computing, in particular on GPU or GPGPU. We especially focus on fluid dynamics problems where time scales of interest allow for interaction with the simulation, and where the models can be controlled by changing parameters and operating conditions with effects viewed instantaneously.

The spin-off effect of such applications is the ease with which a user may "play" with the computational method and the underlying Physics thanks to the visualization. We believe that the coupling between instantaneous computing and interactive visualization really brings an extra dimension to better understand and evaluate methods or schemes. It is a new valuable tool for the applied mathematician. GPU computing today seems to be an inevitable affordable way to build such kinds of applications. Of course there is a price to pay to fully take advantage of GPU resources. We need to reconsider conventional methods and design new innovative computational algorithms to really take advantage of the theoretical peak performance.

2.1.2. **Impact on the design of numerical methods**

As a simple statement of fact, standard sophisticated discretization methods for partial differential problems or optimization algorithms are not really suited for high-performance parallelization on manycore GPU-like architectures. For example, implicit methods lead to a large (sparse) linear system which is solved either by a direct method involving a sparse factorization and a sequential descent/roll up, or by an iterative algorithm which is also sequential by nature. Of course, one can find BLAS-like libraries on many-cores (like cuBLAS on nVIDIA GPU), but today reported speedups are partially satisfactory. For that reason, explicit methods are certainly much more suitable on GPU architectures. Another aspect is the memory access to neighboring degrees of freedom, which is a common issue for PDE-based problems. It is important to notice that there are strongly optimized data structures and methods for fixed neighbor stencil patterns access. This makes cartesian structured grids very suitable candidates. For complex geometries, one can imagine immersed boundary methods (IBM) into cartesian uniform meshes.

Our belief at CMLA is that we have to reconsider both models and methods, in order to derive efficient single-program multiple-data (SPMD) algorithms with communications rates that do not affect the floating point performance too much. For most of the classical PDEs, there are tracks to achieve manycore-suited computational approaches. For example, Lattice Boltzmann (LB) methods are a particular class of cellular automata able to discretize classical equations like the heat equations, convection-diffusion equations and even the unsteady Navier-Stokes equations or more complicated coupled systems. Because of the explicit nature of the method and the uniform local spatial pattern/stencil of discretization, the LB methods are excellent SPMD computational approaches. In the next section, we will focus and give more details on LB methods for solving the incompressible Navier-Stokes equations.

Another track is the underlying microscopic dynamics behind macroscopic models. Generally PDEs are nothing else but a deterministic macroscopic representation of some microscopic or mesoscopic dynamics with "uncertainty" taken into account (stochastic effects like brownian motion). The Laplace operator for example is the macroscopic diffusion operator of the microscopic brownian random walk. Generally, at the microscopic scale, there is an underlying transport process and a collision/interaction phenomenon. On the other hand, the possible numerical simulation at microscopic scale will require a large number of "individuals" in order to derive accurate statistics able to return the macroscopic effects accurately. Manycore architectures are excellent hardware candidates to run population-based computational approaches (like particle methods) on a very large number of individuals. In the sequel we shall give some illustrative examples.

Incompressible flows. Consider the two-dimensional incompressible Navier-Stokes equations defined on a bounded spatial domain Ω of $\mathbb{R}^2$:

$$\nabla \cdot \boldsymbol{u} = 0, \quad t > 0,\ x \in \Omega, \tag{2.1}$$

$$\rho\left(\partial_t \boldsymbol{u} + \boldsymbol{u} \cdot \nabla \boldsymbol{u}\right) + \nabla p - \rho\nu\Delta u = 0, \quad t > 0,\ x \in \Omega, \tag{2.2}$$

where ρ is the (constant) density, $\boldsymbol{u}$ the fluid velocity, p the pressure and $\nu > 0$ the kinematic viscosity of the fluid. There is lot of literature on how to discretize this system of equations. Because of the implicit incompressibility constraint $\nabla \cdot \boldsymbol{u} = 0$, generally an implicit solver is used leading to the solution of a large linear system to solve at each time step, what is not very directly suitable for GPU.

Since two decades, a new family of discrete explicit methods, namely the Lattice Boltzmann methods (LBM) (see the book [12] for example or the review paper [9]) are a kind of kinetic-based cellular automata. They are based on a discretization of the Boltzmann equation

$$\partial_t f + \boldsymbol{e} \cdot \nabla_x f = (\partial_t f)_{coll}$$

governing the distribution $f = f(\boldsymbol{x}, \boldsymbol{e}, t)$ of gas particles having speed $\boldsymbol{e}$ at position $\boldsymbol{x}$ and time t. The term $(\partial_t f)_{coll}$ models all the possible pairwise particle collisions. It is expected that the system fulfils the so-called H-theorem, stating that the entropy functional $H(f) = \int f \log f \, d\boldsymbol{e}$ decreases in time. The equilibrium steady state returns the well-known Maxwellian distribution (see for example [3] for a general theory).

Lattice Boltzmann methods have the advantage to be implemented very easily and even to deal with complex geometries using an immersed boundary approach while being potentially very accurate. Let us consider the 2D Navier-Stokes case: the basic LB method is the so-called Lattice BGK (LBGK) method that uses a BGK collision operator

$$(\partial_t f)_{coll} = \frac{f^{eq} - f}{\tau}$$

for an equilibrium distribution f^{eq} and a characteristic collision time scale $\tau > 0$. The discretization process first deals with a finite set of discrete velocities $\{\boldsymbol{e}_i\}_{i=1,\ldots,N}$, $N > 1$. This leads to a coupled system of spatial transport equations

$$\partial_t f_i + \boldsymbol{e}_i \cdot \nabla_x f_i = \frac{f_i^{eq} - f_i}{\tau}, \quad i = 1, ..., N$$

with $f_i(x,t) \approx f(x, \boldsymbol{e}_i, t)$. The standard D2Q9 lattice makes use of a uniform spatial grid with constant space step per direction $\Delta x = 1$, and (only) $N = 9$ discrete microscopic velocities $\{\boldsymbol{e}_i\}_{i=0,\ldots,8}$ (with $\boldsymbol{e}_0 = 0$) as shown in figure 2.1. Then we have nine discrete transport-collision equations to solve. Using the characteristic method for integrating transport term and a first order explicit Euler dicretization for the collision term, we get

$$f_i(\boldsymbol{x} + \boldsymbol{e}_i \Delta t, t + \Delta t) - f_i(\boldsymbol{x}, t) = \frac{\Delta t}{\tau} \left[f_i^{eq}(\boldsymbol{x}, t) - f_i(\boldsymbol{x}, t) \right], \quad i = 0, ..., 8, \tag{2.3}$$

with $\tau > 0$ the characteristic collision time, for each lattice point $\boldsymbol{x}$. The zeroth-order and first-order moments allow us to retrieve both density and momentum. Denoting by $\mathcal{S} = \{0, ..., 8\}$, we have

$$\sum_{i \in \mathcal{S}} (1,\, \boldsymbol{e}_i)\, f_i(x, t) = (\rho,\, \rho \boldsymbol{u})(x, t). \tag{2.4}$$

From a formal Chapman-Enskog expansion of the discrete density probability functions f_i,

$$f_i = f_i^{(0)} + \varepsilon f_i^{(1)} + \varepsilon^2 f_i^{(2)} + ...$$

where ε is a lattice Knudsen number, for $\varepsilon \ll 1$ it is possible retrieve the macroscopic Fluid Mechanics equations. In order to reproduce the Navier-Stokes equations, only the first two approximations $f_i^{(0)}$ and $f_i^{(1)}$ are required [9]. The zero-th order term $f_i^{(0)}$ identifies with the equilibrium discribution f_i^{eq}. Let us consider a dimensionless lattice size $\Delta x = 1$ and a lattice speed $c = 1$ ($\Delta t = 1$). By choosing the equilibrium density function

$$f_i^{eq} = w_i\, \rho \left(1 + 3 \boldsymbol{e}_i \cdot \boldsymbol{u} + \frac{9}{2} (\boldsymbol{e}_i \cdot u)^2 - \frac{3}{2} |\boldsymbol{u}|^2 \right), \tag{2.5}$$

with weighting factors $w_0 = 4/9$, $w_k = 1/9$ for $k = 1, ..., 4$ and $w_k = 1/36$ for $k = 5, ..., 8$, then for $|u| \ll 1$, we get (up to a scaling) second order accurate approximations of the incompressible Navier-Stokes equations with a kinematic viscosity ν equal to

$$\nu = \frac{1}{3} \left(\tau - \frac{1}{2} \right). \tag{2.6}$$

Actually, for a given fluid kinematic viscosity ν, we compute $\tau > \frac{1}{2}$ such that (2.6) holds. It can be shown that the LBGK method is linearly stable as soon as $\tau > 1/2$. Practically, it becomes unstable for τ close to $\frac{1}{2}$ (high Reynolds number), but stabilization methods exist in this case (MRT, entropy fix, positivity preserving, etc, see [2]).

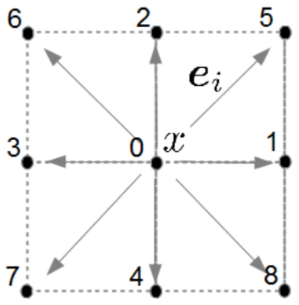


Figure 2.1. The two-dimensional D2Q9 lattice pattern.

LBM code porting on GPU. It is easy to check that LBM can be rewritten as a two-step fractional step method, with i) a collisionless transport evolution, ii) a pure collision process. The GPU collision step can be perfectly done in parallel because of only pointwise operations. The transport step requires communications with the direct first neighboring lattice points. But, because this communication pattern is uniform over the whole computational domain, there are potentially important ways of improvement and performance gain of memory access. On NVIDIA GPU boards, using CUDA programming, one can use `texture` memory (both structures and access methods) that are optimized for fixed memory patterns.

We implemented the D2Q9 LBGK scheme with a stabilization method proposed by Brownlee et al. in [2]. We used Pixel Buffer Object (PBO) for openGL instant visualization and binding between CUDA structures and PBO. On a lattice grid of typical size 1024×1024, we observe speedup factors of about 100 compared to a single-thread CPU sequential computation, allowing for interactivity, visual appearance and evolution of von Karman vortex sheddings. Flow interaction is made possible by adding new obstacles during computation with the mouse (see figure 2.2). This is easily handled by the GPU programming using a solid mask array.

Compressible flows. Let us now consider a compressible fluid. The Euler equations govern the dynamics of a perfect fluid. The mass, momentum

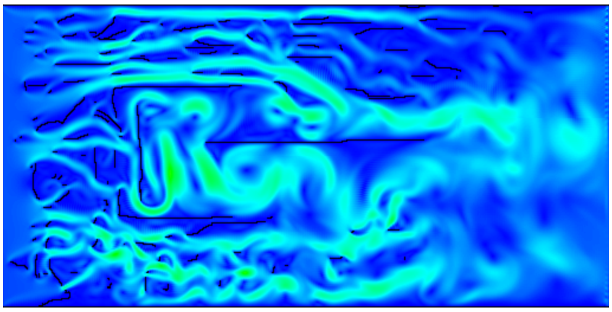


Figure 2.2. Instant Lattice Boltzmann GPU computation of the Navier-Stokes equations on a cartesian grid of typical size 1024×512. Flow interaction is made possible by adding new obstacles during computation with the mouse.

and total energy conservation equations read

$$\partial_t \rho + \nabla \cdot (\rho \boldsymbol{u}) = 0, \tag{2.7}$$

$$\partial_t (\rho \boldsymbol{u}) + \nabla \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u}) + \nabla p = 0, \tag{2.8}$$

$$\partial_t (\rho E) + \nabla \cdot ((\rho E + p)\boldsymbol{u}) = 0 \tag{2.9}$$

with density ρ, velocity vector $\boldsymbol{u}$, pressure p and specific total energy E. The energy E is the sum of the kinetic energy $|u|^2/2$ and the internal energy e. For the perfect gas with constant specific heat ratio γ, $\gamma \in (1,3]$, we have

$$E = e + \frac{1}{2}|\boldsymbol{u}|^2, \quad e = \frac{1}{\gamma - 1}\frac{p}{\rho}. \tag{2.10}$$

The above system can be written in condensed vector form

$$\partial_t U + \nabla \cdot \boldsymbol{F}(U) = 0, \quad U = (\rho, \rho\boldsymbol{u}, E)^T. \tag{2.11}$$

This system is known to be hyperbolic on its admissible state space ([4]).

For discretization, we consider a conservative finite volume scheme built on an unstructured finite volume mesh made of cells K. We will denote A_{KL} the edge separating the two volumes K and L and ν_{KL} the unit exterior vector orthogonal to A_{KL}. A general explicit first-order finite volume scheme reads

$$U_K^{k+1} = U_K^n - \frac{\Delta t}{|K|} \sum_{L \in \mathscr{V}(K)} |A_{KL}|\, \Phi(U_K^n, U_L^n, \nu_{KL}), \tag{2.12}$$

with a numerical flux $\Phi(U_K^n, U_L^n, \nu_{KL})$ having at least Lipschitz-continuous regularity, and being consistent i.e. $\Phi(U, U, \nu) = \boldsymbol{F}(U)\nu$. For stability purposes, numerical fluxes are generally built to fulfil an upwinding property. In this context, two main families of upwind flux are identified (see [6]): the Flux Difference Splitting (FDS) one, and the Flux Vector Splitting (FVS) one. FDS fluxes (including Godunov, Osher, Roe, HLLE, etc...) are written in the form

$$\begin{aligned}&\Phi(U_K^n, U_L^n, \nu_{KL}) = \\ &\frac{1}{2}\left(F(U_K^n,\, \nu_{KL}) + F(U_L^n,\, \nu_{KL})\right) - \frac{1}{2}\int_{\Gamma_{KL}^n} |A(U(s),\, \nu_{KL})|\, U'(s)\, ds\end{aligned} \tag{2.13}$$

where $A(U, \nu)$ denote the (diagonalizable) Jacobian matrix of the flux in the direction ν, and $\Gamma_{KL}^n = \Gamma(U_K^n, U_L^n, s)$ is a Lipschitz continuous path linking the states U_K^n and U_L^n with a curvilinear parameter $s \in [0, 1]$. The second term of the RHS of (2.13) corresponds to the upwind artificial viscosity term.

From the GPU computational point of view, FDS schemes require at each time step i) the computation of the FDS flux with memory reads of the cell states; ii) cell updates with memory reads of the FDS fluxes (see figure 2.3). Memory transfer may be a limiting performance factor for GPUs if the DRAM bandwidth is saturated.

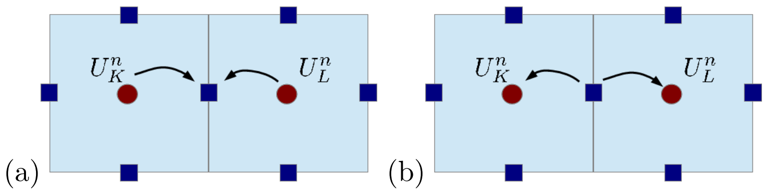


Figure 2.3. FV scheme with Flux Difference Splitting FDS schemes. FDS require two memory transfers: (a) computation of the numerical flux with memory reads of states; (b) state update into control volumes with memory reads of numerical fluxes.

The Flux Vector Splitting (FVS) family [6] has numerical fluxes written in the form

$$\Phi(U_K^n, U_L^n, \nu_{KL}) = F^+(U_K^n,\, \nu_{KL}) + F^-(U_L^n,\, \nu_{KL}) \tag{2.14}$$

with F^+ representing the leftward part of the flux and F^- representing the rightward part. Consistency requirements involve the identity

$$F^+(U,\nu) + F^-(U,\nu) = \boldsymbol{F}(U)\nu.$$

What is peculiar with FVS is that $F^+(U_K^n,\, \nu_{KL})$ can be computed without any knowledge of the neigboring state U_L^n, and conversely. Then the GPU computation of $F^+(U_K^n,\, \nu_{KL})$ may be seen as a pointwise, cell-centered computation, perfectly done in parallel. For that reason, one has only to send F^+ and F^- to the neighboring cells for state updates, thus reducing memory reads and DRAM transfer (see figure 2.4). Moreover, FVS fluxes

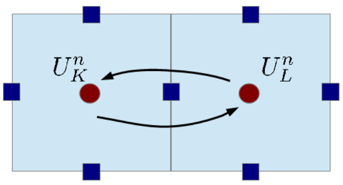


Figure 2.4. FV scheme with Flux Vector Splitting (FVS) fluxes. FVS only require one memory transfers: F^+ ou F^- are sended to the neighboring cells for state update.

generally do not require neither eigenstructure decomposition nor matrix-vector products, what improves the whole performance. For example, the van Leer's FVS with Hänel-Schwane energy-flux modification [5] leads to the scripts (written here for $\nu = (1,0)$):

$$F_\rho^+ = \frac{\rho c}{4}(M+1)^2\, 1_{(|M|\le 1)} + \max(u,0)\, 1_{(|M|>1)}, \tag{2.15}$$

$$p^+ = \frac{p}{4}(M+1)^2(2-M)\, 1_{(|M|\le 1)} + p\, 1_{(M>1)}, \tag{2.16}$$

$$F_{\rho u}^+ = u\, F_\rho^+ + p^+,\;\; F_{\rho v}^+ = v\, F_\rho^+, \tag{2.17}$$

$$F_{\rho E}^+ = (E + p/\rho)\, F_\rho^+ \tag{2.18}$$

where $c = \sqrt{\gamma p/\rho}$ is the speed of sound and $M = u/c$ is the normal Mach number. For these reasons, FVS are clearly better candidates for GPU implementation and high-performance computation [13]. On figure 2.5, we show an instant computation of the well-known 2D Mach 3 forwarding step case on a very fine grid, using (2.15)-(2.18) as FVS scheme. Again, mouse interactivity allows us to add obstacles on-the-fly and observe the fluid response. This is of great interest for training, education and comprehension of Fluid Dynamics.

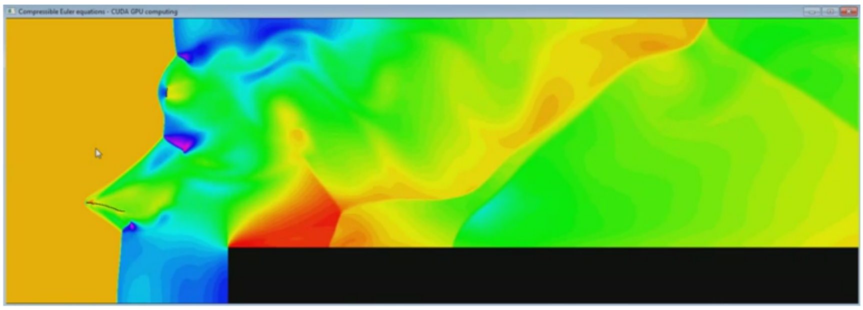


Figure 2.5. Instant GPU computation on the well-known Mach 3 forward step channel 2D problem. Hänel's FVS is here used. The flow can be perturbed by directly adding new wall obstacles with the mouse pointer.

Three-dimensional free transport kinetic equations. This case was designed to evaluate GPU performance of particle methods. Let us consider the following homogeneous Vlasov equation in 3D: find $f = f(\boldsymbol{x}, \boldsymbol{v}, t)$, $\boldsymbol{x} \in \Omega(t) \subset \mathbb{R}^3$, $\boldsymbol{v} \in \mathbb{R}^3$, $t > 0$, solution of

$$\partial_t f + \boldsymbol{v} \cdot \nabla_x f + a(\boldsymbol{x}) \nabla_v f = 0, \quad \boldsymbol{x} \in \Omega(t) \subset \mathbb{R}^3,\ \boldsymbol{v} \in \mathbb{R}^3,\ t > 0 \tag{2.19}$$

with $f(.,.,0) \in L^1(\Omega \times \mathbb{R}^3)$. Standard eulerian discretization methods would involve a mesh in a space of dimension 6, what is still not realistic to address at the present time. An alternative approach is to reformulate the transport dynamics behind this equation. Let us consider the following system of motion equations defined on a large set of particles $\{\boldsymbol{x}_k\}_k$:

$$\begin{cases} \dot{\boldsymbol{x}}_k(t) = \boldsymbol{v}_k, \\ \dot{\boldsymbol{v}}_k(t) = \boldsymbol{a}(\boldsymbol{x}_k) \end{cases} \tag{2.20}$$

and the discrete measure-valued distribution

$$f = \sum_{k=1}^{N} \omega_k \, \delta(\boldsymbol{x} - \boldsymbol{x}_k(t)) \, \delta(\boldsymbol{v} - \boldsymbol{v}_k(t)) \tag{2.21}$$

for some given weighting factors $\{\omega_k\}_k$, $\omega_k > 0$. We want to evaluate the L^1 norm on f in the time-dependent domain $\Omega(t)$ (a pulsating sphere for example, see [14]). For that we have to compute at each time step

$$\|f(t)\|_{L^1(\Omega(t))} = \sum_{k=1}^{N} \omega_k \, 1_{(x_k \in \Omega(t))}. \tag{2.22}$$

The summation has to be "optimized" on a GPU architecture. For that we used the sum-reduction algorithm proposed in the GPU code examples of the CUDA SDK. We have observed parallel speedup factors of about 25-30 for particles sets between 1 million and 8 million particles on a GPGPU TESLA C2070.

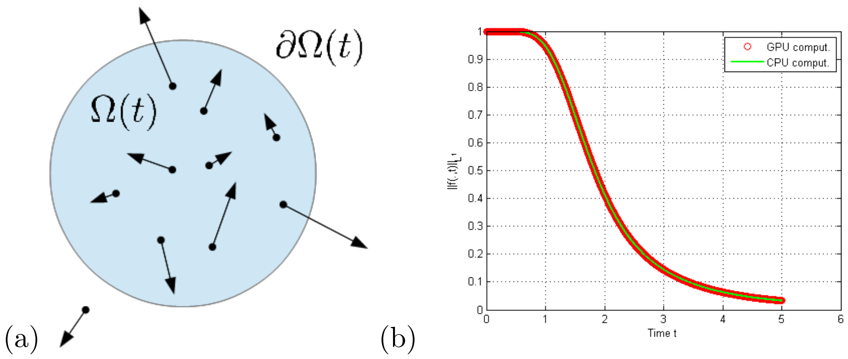


Figure 2.6. Validation of GPU acceleration of free transport equations on a moving domain with parallel reduction at each time step for L^1-norm computation. (a) Schematic of the particles and moving domain; (b) Discrete $L^1(\Omega(t))$-norm of the distribution during time t.

2.1.3. Impact on the data structures

In the above sections, we have seen how GPU computing may change the way of thinking both physical models and computational methods. Beyond pure computational aspects, there is of course the programming and code optimization dimensions. The obtained "speedup" and the necessary work of specidifc GPU programming are subject to a search of thrust performance. Parallel computing strongly alters the balance of power in the classical duality memory-computation. A first simple rule is to try to keep as much as possible data on the parallel ”device” and transfer data as less as possible because of limited bandwidth. To illustrate, it is even often cheaper to recompute a result than transmitting it.

Communication performance is also strongly linked to the way to handle complex data. Data coalescence is a critical keypoint for optimal performance. To offset a large latency of global memory a rational way is to read consecutive blocks in memory (coalescing). There are two kinds of non-coalescing memory accesses, described in the figure 2.7:

- either the *threads* cannot access to neighboring fields in the right order;
- or there is an alignment problem, the first *thread* of a *warp* must access one memory multiple of 32, 64 or 128 (depending on the data), see [10].

It is difficult when looking to optimized performance, to work with very sophisticated data models. The notion of array perfectly fits to this scheme, data types more developed as structures or classes instead readily scatter in the data. For example it is much more efficient to manipulate Structures of Arrays (SoA) that Arrays of Structures (AoS). To be efficient, it is necessary to stay close to the data and always keep in mind the specific hardware architecture of GPU and the constraints it imposes to the data. This is probably the main reason why we think that today CUDA is one of the most appropriate language to obtain optimal performance on GPU.

2.1.4. **General experience feedback and concluding remarks**

Let us conclude by some humble advices. Today’s GPU for scientific computing is a question a tradeoff between performance and design and/or implementation effort. Reasonable (suboptimal) speedup (say 10 or 20) is easy to reach. GPU parallel programming is far easier for computational methods feined on cartesian grids or meshfree particle methods. For stronger performance, the way is to fing a good tradeoff between implementation effort, code readability and performance. GPU computing requires a real reflection on the choice of data structures, especially for

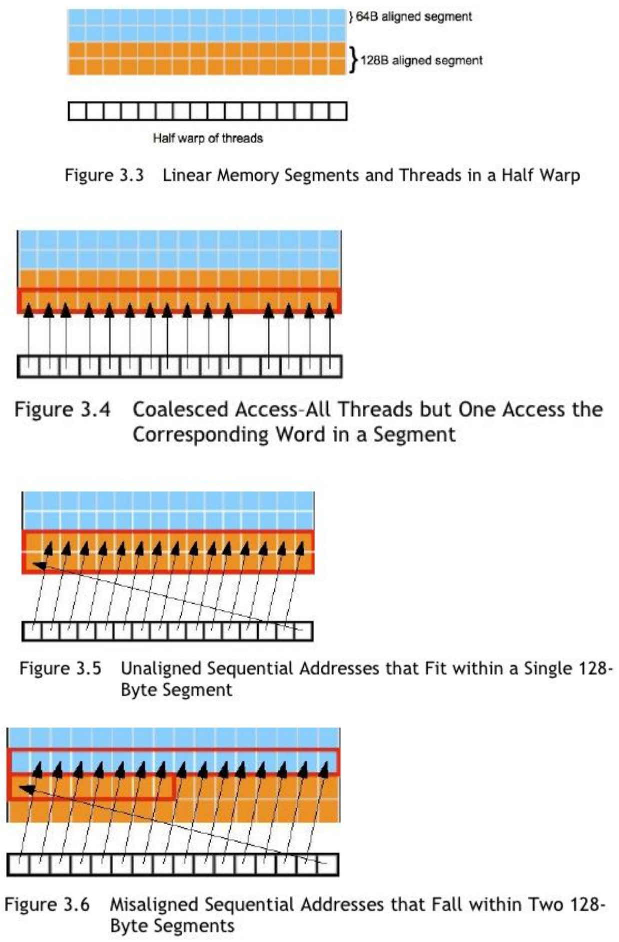

64B aligned segment
128B aligned segment
Half warp of threads
Figure 3.3 Linear Memory Segments and Threads in a Half Warp
Figure 3.4 Coalesced Access-All Threads but One Access the Corresponding Word in a Segment
Figure 3.5 Unaligned Sequential Addresses that Fit within a Single 128-Byte Segment
Figure 3.6 Misaligned Sequential Addresses that Fall within Two 128-Byte Segments

Figure 2.7. Schematic of data coalescence, extracted from the "CUDA C Best Practices Guide", NVIDIA 2012 [10]

the sake of memory coalescence: arrays of structures AoS versus structures of arrays SoA, byte alignment, etc. In the same spirit, the strategy/use of intermediate cache memory level (per streaming multiprocessor for example) is also of great importance for high performance. Texture memory is particularly suited for local partial differential operators involving uniform spatial stencils.

Our belief at CMLA is that GPU/manycore processors will deeply impact the numerical solvers in the next years. We have to think about paradigm shifts for both modeling and discretization for strong better GPU performance.

2.2. Videos of instant GPU computations on youtube

All the interactive computations can be found at the following URL:
`http://www.youtube.com/user/floriandevuyst/videos`.

Acknowledgements

This work has been partly supported by the FARMAN Institute of ENS Cachan and by a NVIDIA Equipment Grant in 2011.

References

[1] M. Bergdorf & P. Koumoutsakos – "A Lagrangian particle-wavelet method", *Multiscale Models. Simul.* **5** (2006), no. 3, p. 980–995.

[2] R. Brownlee, A. Gorban & J. Levesley – "Stabilization of the lattice boltzmann method using the Ehrenfests' coarse-graining data", *Physical Review E* **74** (2006), p. 037703.

[3] C. Cercignani – *The boltzmann equation and its applications*, vol. 67, Springer-Verlag, 1988.

[4] E. Godlewski & P.-A. Raviart – *Numerical approximation of hyperbolic conservation laws*, vol. 118, Applied Mathematical Sciences, Springer-Verlag, Boston, 1996.

[5] D. Hanel & R. Schwane – "An implicit flux-vector splitting scheme for the computation of viscous hypersonic flow", *AIAA Paper* **25** (1989), Paper 89-0274.

[6] A. Harten, P. Lax & B. van Leer – "On upstream differencing and godunov-type schemes for hyperbolic conservation laws", *SIAM Review* **25** (1983), p. 35–61.

[7] A. Magni & G. Cottet – "Accurate, non-oscillatory, remeshing schemes for particle methods", *J. Comput. Phys.* **231** (2012), no. 1, p. 152–172.

[8] A. Munshi et al. – "The OpenCL Specification", *Khronos OpenCL Working Group* (2011).

[9] R. Nourgaliev, T. Dinh, T. Theofanous & D. Joseph – "The lattice boltzmann equation method: theoretical interpretation, numerics and implications", *Int. J. of Multiphase Flow* **29** (2003), p. 117–169.

[10] *Cuda c best practices guide 4.1, nvidia* – 2012.

[11] D. Rossinelli, M. Bergdorf, G. Cottet & P. Koumoutsakos – "GPU accelerated simulations of bluff body flows using vortex methods", *J. Comput. Phys.* **229** (2010), no. 9, p. 3316–3333.

[12] S. Succi – *The lattice boltzmann equation for fluid dynamics and beyond*, Oxford, 2001, ISBN:0-19-850398-9.

[13] F. D. Vuyst – "A flux vector splitting method that preserves stationary contact discontinuities", *Acta Mathematicae Applicandae* (2013), accepted, under revision.

[14] F. D. Vuyst & F. Salvarani – "Gpu-accelerated numerical simulations of the knudsen gas on time-dependent domains", *Computer Physics Communications* **184** (2013), no. 3, p. 532–536.

Georges-Henri Cottet
Laboratoire Jean Kuntzmann
Université Joseph Fourier
BP 53
38041, Grenoble Cedex 9
France
Georges-Henri.Cottet@imag.fr

Jean-Matthieu Etancelin
Laboratoire Jean Kuntzmann
Université Joseph Fourier
BP 53
38041, Grenoble Cedex 9
France
Jean-Matthieu.Etancelin@imag.fr

GPU Computing

FRANCK PERIGNON
Laboratoire Jean Kuntzmann
Université Joseph Fourier
BP 53
38041, Grenoble Cedex 9
France
franck.perignon@imag.fr

CHRISTOPHE PICARD
Laboratoire Jean Kuntzamnn
Université Joseph Fourier
BP 53
38041, Grenoble Cedex 9
France
Christophe.Picard@imag.fr

FLORIAN DE VUYST
Centre de Mathématiques
et de leurs Applications
CMLA CNRS UMR 8536
61, avenue du Président Wilson
94235 Cachan CEDEX
FRANCE
devuyst@cmla.ens-cachan.fr

CHRISTOPHE LABOURDETTE
Centre de Mathématiques
et de leurs Applications
CMLA CNRS UMR 8536
61, avenue du Président Wilson
94235 Cachan CEDEX
FRANCE
labour@cmla.ens-cachan.fr