HAL
open science

Is GPU the future of Scientific Computing?

Georges-Henri Cottet, Jean-Matthieu Etancelin, Franck Pérignon, Christophe Picard, Florian de Vuyst, Christophe Labourdette

To cite this version:

Georges-Henri Cottet, Jean-Matthieu Etancelin, Franck Pérignon, Christophe Picard, Florian de Vuyst, et al.. Is GPU the future of Scientific Computing?. Annales Mathématiques Blaise Pascal, 2013, 20 (1), pp.75-99. 10.5802/ambp.322 . hal-00780565

HAL Id: hal-00780565

https://hal.science/hal-00780565

Submitted on 24 Jan 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Annales mathématiques Blaise Pascal **Version de travail – 24 janvier 2013**

Particle method on GPU

Jean-Matthieu Etancelin
Georges-Henri Cottet
Christophe Picard
Franck Perignon

Particle method on GPU

Abstract. In this article 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. Introduction

Particle methods are well adapted to numerical simulations of fluid dynamics, especially for turbulent flows where accurate simulations are required to represent small scales. Particle advection and remeshing [2] lead to a simple algorithm, working on extremely regular data structures, that is suited to a graphics processing unit (GPU) implementation. Today's supercomputers are based on hybrid architectures mixing thousands of multi-core processors and GPUs. Our goal is therefore to develop GPU implementations within a framework allowing to combine different type of software. The paper is organized as follows. In Section 2, we describe the method and in Section 3 we detail the GPU implementation. Section 4 is dedicated to numerical illustrations and performances results. Finally, a conclusion is drawn in Section 5.

Mots-clés: GPU, méthode particulaire, EDP.
Classification math.: 35L05, 65M08, 76M25, 76N15, 76P05, 97N40, 00X99.

2. Previous work

2.1. Particle method

In the present work, we focus on solving advection equation (2.1) by means of remeshed particle method.

$$\partial_t u + \mathrm{div}(au) = 0 \tag{2.1}$$

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.

$$\frac{\mathrm{d}\tilde{x}_p}{\mathrm{d}t} = a(\tilde{x}_p, t) \tag{2.2}$$

The remeshing step is performed with the following general formula.

$$u_g = \sum_p \tilde{u}_p W\left(\frac{x_g - \tilde{x}_p}{h}\right) \tag{2.3}$$

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 (2.1) 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 (2.2) 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 [2], 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.

2.2. **OpenCL computing**

OpenCL is an open standard for parallel programming of heterogeneous systems [3]. 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.

3. **Particle method with OpenCL**

Particle methods have already been implemented on GPU. In [4], 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 2.1 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 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.

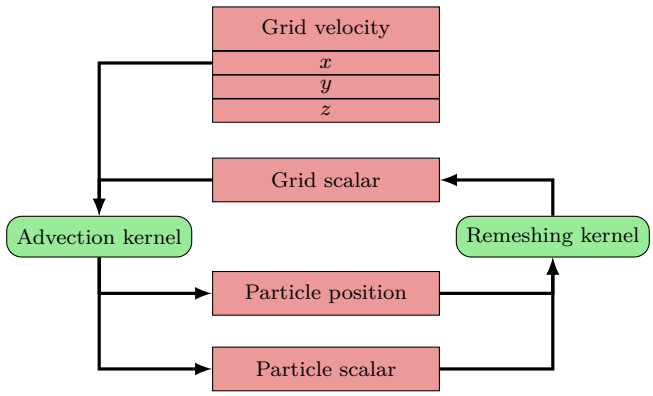


Figure 1. Execution layout on GPU. Memory objects are depicted in red and OpenCL kernels in green

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, 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

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

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.

3.1. **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 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.

3.2. **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.

4. Performances and results

4.1. 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.

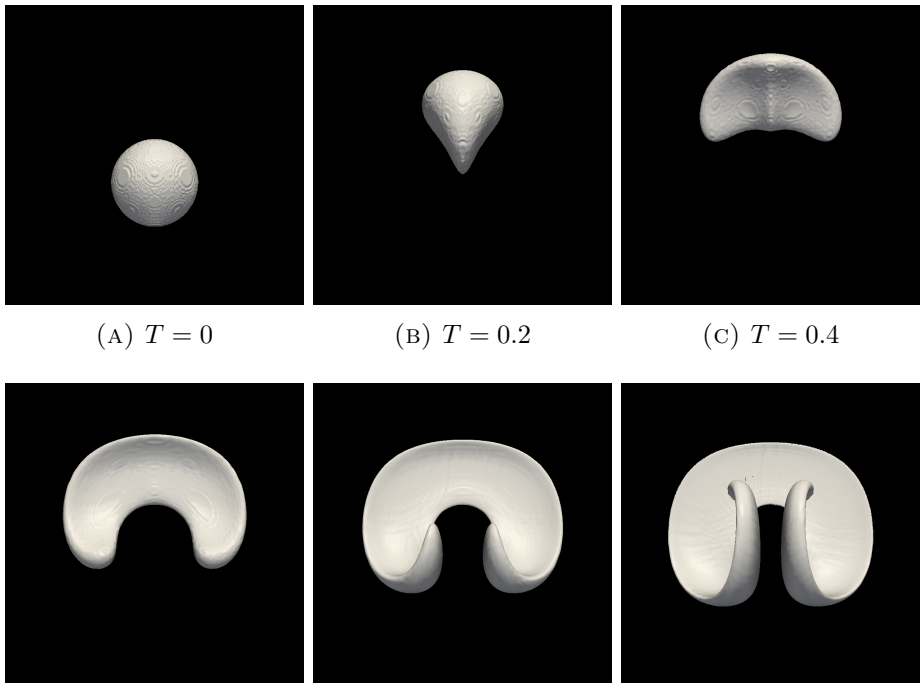
$$a(x) = \begin{pmatrix} 2\sin(\pi x)^2 \sin(2\pi y)\sin(2\pi z) \\ -\sin(2\pi x)\sin(\pi y)^2 \sin(2\pi z) \\ -\sin(2\pi x)\sin(2\pi y)\sin(\pi z)^2 \end{pmatrix}$$

One of the tests implemented in [2] and the references therein is presented in Figure 2. This simulation is performed with $N = 256^3$ and a time step value which would correspond to a CFL number equal to 25 at this resolution. Time $T = 1$ is reached in 20 iterations and took 25 seconds.

4.2. 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 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



(A) $T = 0$ (B) $T = 0.2$ (C) $T = 0.4$

(D) $T = 0.6$ (E) $T = 0.8$ (F) $T = 1.0$

FIGURE 2. Iso-surface of level 0.5 at different times, $CFL = 25$, $dt = 0.05$.

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.

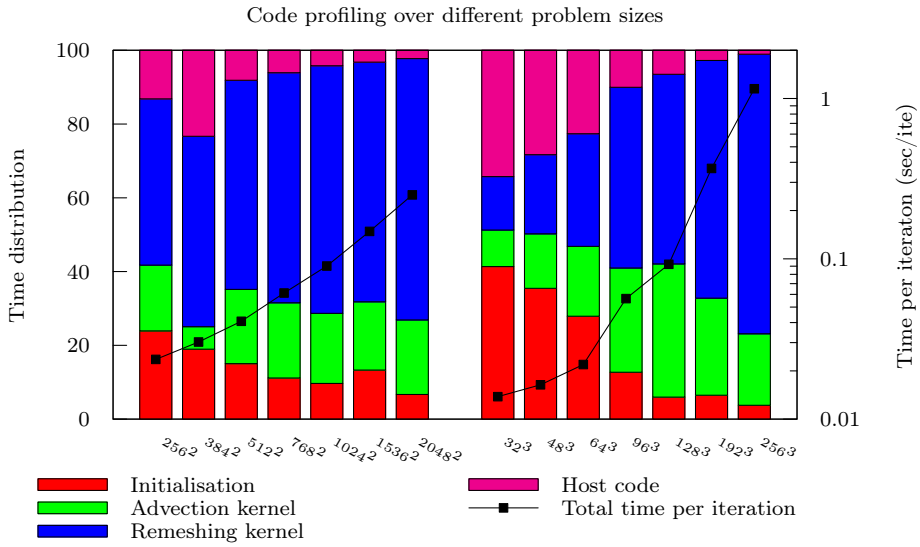


Figure 3. Profiling data on ATI Radeon HD 6770M

To give another comparison, the computational times showed in [4] 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 [4] were two-dimensional and used lower order remeshing schemes but involved non-local field evaluations (through FFT).

5. 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.

Bibliographie

[1] M. Bergdorf and P. Koumoutsakos. A Lagrangian particle-wavelet method. *Multiscale Models. Simul.*, 5(3) :980–995, 2006.

[2] A. Magni and G.H. Cottet. Accurate, non-oscillatory, remeshing schemes for particle methods. *J. Comput. Phys.*, 231(1) :152–172, 2012.

[3] A. Munshi et al. The OpenCL Specification. *Khronos OpenCL Working Group*, 2011.

[4] D. Rossinelli, M. Bergdorf, G.H. Cottet, and P. Koumoutsakos. GPU accelerated simulations of bluff body flows using vortex methods. *J. Comput. Phys.*, 229(9) :3316–3333, 2010.

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

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

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

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