ESAIM: PROCEEDINGS, Juillet 2013, Vol. 40, p. 51-62
C. Bourdarias, S. Gerbi, Editors

NUMERICAL RESOLUTION OF CONSERVATION LAWS WITH OPENCL

A. Crestetto[1], P. Helluy[2] and J. Jung[3]

Abstract. We present several numerical simulations of conservation laws on recent multicore processors, such as GPUs, using the OpenCL programming framework. Depending on the chosen numerical method, different implementation strategies have to be considered, for achieving the best performance. We explain how to program efficiently three methods: a finite volume approach on a structured grid, a high order Discontinuous Galerkin (DG) method on an unstructured grid and a Particle-In-Cell (PIC) method. The three methods are respectively applied to a two-fluid computation, a Maxwell simulation and a Vlasov-Maxwell simulation.

1. Introduction

Recent parallel CPU and GPU architectures are very efficient for scientific computing. Several programming frameworks, such as OpenMP, CUDA or OpenCL are developed for several years in order to help the implementation of classical algorithms on GPUs or multicore CPUs. The implementation is generally not so obvious and one has to care of some aspects in order to reach efficient simulations. The first point is that, because of the massive parallelism of some devices, the computations may become negligible compared to data memory transfers. The data organization into memory becomes the most important point of the algorithms. The second point is that simple operations in sequential program, such as computing an integral or a maximum on the computational domain, become non trivial on parallel computers because they imply memory write conflicts. Such operations have thus to be reorganized in order to perform efficiently in parallel.

In this paper we review three classical methods for solving hyperbolic systems of conservation laws: the structured Finite Volume method (FV), the high order Discontinuous Galerkin (DG) method and the Particle-In-Cell (PIC) method. We explain how to program them efficiently on GPUs, using the OpenCL framework. We also apply our simulations to realistic 2D test cases coming from physics

2. Multicore architectures and OpenCL

Several computer parallel architectures exist. The Single Instruction Multiple Data (SIMD) model is often interesting in scientific computing because it allows to perform the same operation on many data in parallel. The processors share the same instruction unit, which also allows excellent performance/watt ratio (green computing). Recent GPU devices are of this type, with several hundreds of processors sharing a few instruction units.

OpenCL is a programming framework for driving such devices. OpenCL is practically available since september 2009 [11]. The specification is managed by the Khronos Group, which is also responsible of the OpenGL API design and evolutions. OpenCL means “Open Computing Language”.

Many different architectures exist, but OpenCL proposes a unified model for programing generic SIMD machines, called “devices” in the OpenCL terminology. The generic device can be in practice a GPU, a multicore CPU or even a computer made of several multicore CPUs. In this model, a GPU is considered as a device plugged into a computer, called a “host”. See Figure 1. A device is made of (the typical hardware characteristics below are given for a NVIDIA GeForce GTX 280 GPU)

- Global memory (typically 1 Gb)

[1] CALVI, Inria Nancy-Grand Est & IRMA, Université de Strasbourg
[2] TONUS, Inria Nancy-Grand Est & IRMA, Université de Strasbourg, helluy@unistra.fr
[3] IRMA, Université de Strasbourg

Article published online by EDP Sciences and available at http://www.esaim-proc.org or http://dx.doi.org/10.1051/proc/201340004

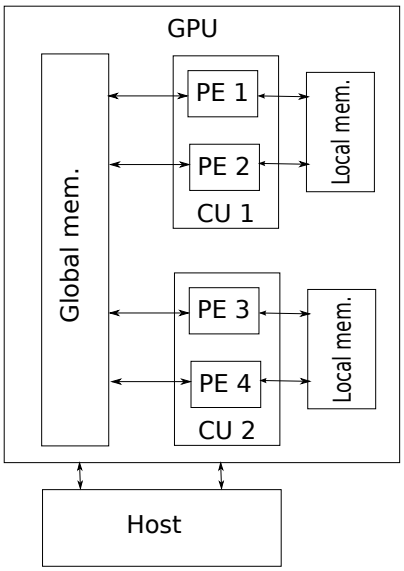

FIGURE 1. A (virtual) GPU with 2 Compute Units and 4 Processing Elements

- Compute units (typically 27).

Each compute unit is made of:

- Processing elements (typically 8).
- Local (or cache) memory (typically 16 kb)

The same program (a "kernel") can be executed on all the processing elements at the same time, with the following rules:

- All the processing elements have access to the global memory.
- The processing elements have only access to the local memory of their compute unit.
- If several processing elements write at the same location at the same time, only one write is successful.
- The access to the global memory is slow while the access to the local memory is fast. We can achieve faster reads/writes to the global memory if neighboring processors access neighboring memory locations.

OpenCL includes:

- A library of C functions, called from the host, in order to drive the GPU (or the multicore CPU);
- A C-like language for writing the kernels that will be executed on the processing elements.

Virtually, it allows having as many compute units and processing elements as needed. The virtual OpenCL compute units are called "work-groups" and the virtual processing elements are called "work-items". The work-groups and work-items are distributed on the real compute units and processing elements of the GPU thanks to a mechanism of command queues. The user does not have to bother of that mechanism because it is managed automatically by the OpenCL driver.

The main advantage of OpenCL is its portability. The same program can run on multicore CPUs or GPUs of different brands. It is also possible to drive several devices at the same time for better efficiency. For instance, in a single PC, a part of the computations can be executed on the CPU while the other part is run on one or several GPUs. This allows very important computation power at a very low cost. Recent evolutions indicate that in the future, the OpenCL devices will share common memory buffers. This will greatly improve the efficiency of CPU-GPU cooperation by avoiding slow memory copies. Many resources are available on the web for learning OpenCL. For a tutorial and simple examples, see for instance [8].

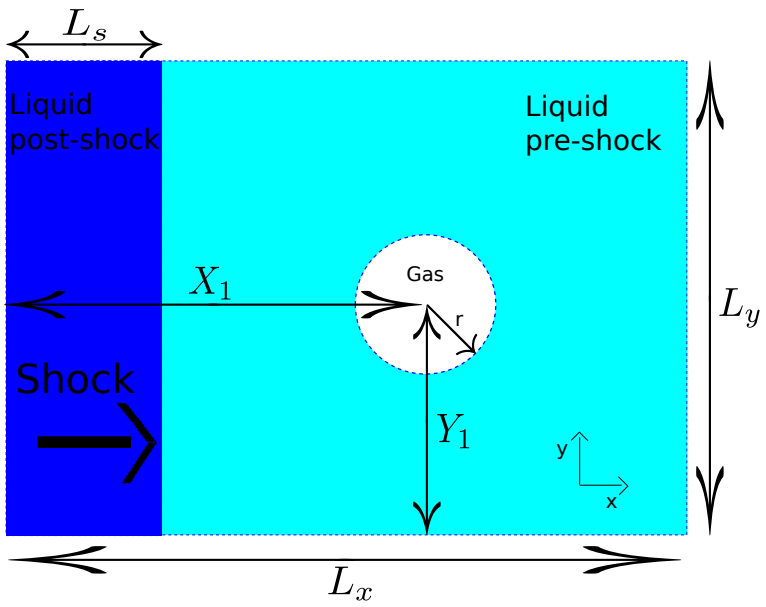

FIGURE 2. Liquid-gas shock-bubble interaction test. Description of the initial conditions.

3. A structured FV approximation of a compressible two-fluid flow

3.1. **Shock-bubble interaction**

Our first example is devoted to solving a 2D shock-bubble interaction in a compressible medium. The bubble is made of air inside liquid water. The liquid water is shocked at the left boundary and the planar shock wave will impinge the bubble. The initial condition of the problem is depicted on Figure 2 with the following parameters

$$L_x = 2\,\text{m},\ L_y = 1\,\text{m}, L_s = 0.04\,\text{m},\ X_1 = 0.5\,\text{m},\ Y_1 = 0.5\,\text{m},\ r = 0.4\,\text{m}.$$

The test case is also studied in [13, 17].

We first write the mathematical model, which is a hyperbolic system of conservation laws. We consider the vector of conservative variables $W = (\rho, \rho u, \rho v, \rho E, \rho\varphi)^T$ depending on the space variable $X \in \mathbb{R}^2$ and time t, where ρ is the density, $U = (u, v)^T$ is the velocity, E is the total energy and φ the color function ($\varphi = 0$ in the liquid and $\varphi = 1$ in the gas). The internal energy $e = E - u^2/2$ and the pressure $p = p(\rho, e, \varphi)$. Let $n \in \mathbb{R}^2$ be a normal vector, the flux of the system is given by

$$F(w) \cdot n = (\rho U \cdot n, \rho(U \cdot n)U^T + pn^T, (\rho E + p)U \cdot n, \rho\varphi U \cdot n)^T,$$

and the system of conservation laws reads

$$\partial_t W + \nabla_X \cdot F(W) = 0.$$

At the boundary we simply apply the data corresponding to the unperturbed initial shock wave.

The pressure law is a stiffened gas Equation Of State (EOS) where the parameters γ and π depend on the color function φ

$$\begin{aligned} p(\rho, e, \varphi) &= (\gamma(\varphi) - 1)\rho e - \gamma(\varphi)\pi(\varphi), \\ \frac{1}{\gamma(\varphi) - 1} &= \varphi\frac{1}{\gamma_{\text{gas}} - 1} + (1 - \varphi)\frac{1}{\gamma_{\text{liq}} - 1}, \\ \frac{\gamma(\varphi)\pi(\varphi)}{\gamma(\varphi) - 1} &= \varphi\frac{\gamma_{\text{gas}}\pi_{\text{gas}}}{\gamma_{\text{gas}} - 1} + (1 - \varphi)\frac{\gamma_{\text{liq}}\pi_{\text{liq}}}{\gamma_{\text{liq}} - 1}. \end{aligned}$$

The system is hyperbolic with eigenvalues $U \cdot n - c, U \cdot n, U \cdot n, U \cdot n, U \cdot n + c$ and $c = \sqrt{\gamma(p + \pi)/\rho}$.

The EOS parameters and initial data are given in Table 1.

For numerically solving this system we will apply Strang dimensional splitting. Therefore, we first we investigate the 1D framework.

Quantities	Liquid (post-shock)	Liquid (pre-shock)	Gas
$\rho\ (kg.m^{-3})$	1030.9	1000.0	1.0
$u\ (m.s^{-1})$	300.0	0	0
$v\ (m.s^{-1})$	0	0	0
p (Pa)	3×10^9	10^5	10^5
φ	0	0	1
γ	4.4	4.4	1.4
π	6.8×10^8	6.8×10^8	0

TABLE 1. Liquid-gas/bubble interaction test. Initial data.

3.2. **One-dimensional scheme**

We set $X = (x, y)^T$, and we suppose that $W = W(x, t)$, $n = (1, 0)^T$. The system of conservation laws become

$$\partial_t W + \partial_x (F(W) \cdot n) = 0. \tag{1}$$

As is well known, standard conservative finite volume schemes for solving such a system give very low precision results (see for instance [3] and included references). We have proposed in [2] a new Lagrange and remap scheme that we will apply here. We first recall how it works. As usual, we consider a 1D mesh of cells $C_i =]x_{i-1/2}, x_{i+1/2}[$. The size of cell C_i is $h_i^n = x_{i+1/2} - x_{i-1/2}$. The time step is noted $\tau_n = t_{n+1} - t_n$ and $W_i^n \simeq W(x_i t_n)$.

Each time step of the scheme is made of two stages: an Arbitrary Lagrangian Eulerian (ALE) step and a remap step for going back to the Eulerian mesh. In the first stage, we approximate the solution with an ALE scheme

$$h_i^{n+1,-} W_i^{n+1,-} - h_i^n W_i^n + \tau_n \left(F_{i+1/2}^n - F_{i-1/2}^n \right) = 0.$$

For a left state W_L, a right state W_R and a velocity x/t, we denote by $R(W_L, W_R, x/t)$ an exact or approximate Riemann solver for the problem (1). The Lagrange numerical flux is then defined by $W_{i+1/2}^n = R(W_i^n, W_{i+1}^n, x/t = u_{i+1/2}^n)$ and

$$\begin{aligned} F_{i+1/2}^n &= F(W_{i+1/2}^n) - u_{i+1/2}^n W_{i+1/2}^n, \\ W_{i+1/2}^n &= R(W_i^n, W_{i+1}^n, u_{i+1/2}^n), \end{aligned}$$

where the cell boundary $x_{i+1/2}$ moves at the (yet unknown) velocity $u_{i+1/2}^n$

$$x_{i+1/2}^{n+1,-} = x_{i+1/2} + \tau_n u_{i+1/2}^n.$$

At the end of the ALE step, the new cell size is thus

$$h_i^{n+1,-} = x_{i+1/2}^{n+1,-} - x_{i-1/2}^{n+1,-} = h_i^n + \tau_n (u_{i+1/2}^n - u_{i-1/2}^n).$$

For returning to the initial Euler mesh, we use a random remap step, which is justified in [2]. We construct a sequence of random or pseudo-random numbers $\omega_n \in [0, 1[$. According to this number we take [6]

$$\begin{aligned} W_i^{n+1} = W_{i-1}^{n+1,-} \text{ if } \omega_n < \frac{\tau_n}{h_i} \max(u_{i-1/2}^n, 0), \\ W_i^{n+1} = W_{i+1}^{n+1,-} \text{ if } \omega_n > 1 + \frac{\tau_n}{h_i} \min(u_{i+1/2}^n, 0), \\ W_i^{n+1} = W_i^{n+1,-} \text{ if } \frac{\tau_n}{h_i} \max(u_{i-1/2}^n, 0) \le \omega_n \le 1 + \frac{\tau_n}{h_i} \min(u_{i+1/2}^n, 0). \end{aligned}$$

A good choice for the pseudo-random sequence ω_n is the (k_1, k_2) van der Corput sequence, computed by the following C algorithm

```
float corput(int n,int k1,int k2){
  float corput=0;
  float s=1;
  while(n>0){
    s/=k1;
```

```
    corput+=(k2*n%k1)%k1*s;
    n/=k1;}
  return corput;
}
```

In this algorithm, k_1 and k_2 are two relatively prime numbers and $k_1 > k_2 > 0$. As advised in [18], in practice we consider the $(5, 3)$ van der Corput sequence.

The scheme is completely described if we provide the interface ALE velocities $u^n_{i+1/2}$. In the resolution of the Riemann problem $R(W^n_i, W^n_{i+1}, x/t)$ we find four waves. The characteristic fields 2 and 3 are linearly degenerated and $\lambda_2(w) = \lambda_3(w) = u$, thus the velocity is constant across these waves. It is natural to consider u as the interface velocity. We denote it by $u^*(W_i, W_{i+1})$. Our choice for the ALE velocity is then

$$u^n_{i+1/2} = \begin{cases} u^*(W_i, W_{i+1}) \text{ if } \varphi^n_i \neq \varphi^n_{i+1}, \\ 0 \text{ if } \varphi^n_i = \varphi^n_{i+1}. \end{cases} \tag{2}$$

Remarks:

- the natural full Lagrange choice

$$u^n_{i+1/2} = u^*(W_i, W_{i+1}),$$

  leads to a scheme that is not BV stable (see [2]).
- If $u^n_{i+1/2} = 0$, we recover a standard Eulerian scheme. It means that with the choice (2) we follow a Lagrange approach at the interface and a Eulerian approach elsewhere. The random remap is only performed at the interface.

For achieving more robustness, we have also constructed an approximate Riemann solver based on relaxation techniques. The construction is adapted from [5]. The resulting scheme has a full set of desired properties:

- it is positive and handles vacuum.
- It is entropy dissipative (at least in the pure fluid regions).
- The constant (u, p) states are exactly preserved.
- The gas fraction is not smeared at all: $\varphi^n_i \in \{0, 1\}$.
- It is statistically conservative.

3.3. **GPU implementation and numerical results**

In order to perform 2D computations, we use dimensional splitting. For advancing a time step τ, we first numerically solve

$$\frac{W^* - W^n}{\tau} + \partial_x F^1(W^n) = 0, \tag{3}$$

and then

$$\frac{W^{n+1} - W^*}{\tau} + \partial_y F^2(W^*) = 0, \tag{4}$$

with the Lagrange and remap scheme. An interesting point is that we can keep the same pseudo-random number ω_n for the two sub-steps.

For performance reasons, we implement the algorithm on a GPU. An important point is to decide how to distribute the computations to the work-groups and work-items of the GPU. We organize the data in a (x, y) grid, which can also be considered as a matrix. For GPU computations, it is generally advised to perform fine grain parallelism and to keep the computing kernels as simple as possible. Therefore, we associate one work-item to exactly one cell of the grid. In addition, we consider that the work-items of the same row are in the same work-group.

We have to take care of the memory access. Indeed, when we solve the problem (3) in the $x-$direction, memory access are very fast because two successive work-items access successive memory locations. This kind of memory access is said to be coalescent. If nothing is done, when solving (4), the $y-$direction memory access are slow because two neighboring work-items access different rows. The memory access are no more coalescent.

Between the x and y steps, we have therefore to perform a transposition of the grid data. This transposition is performed in an optimized way, using cache memory. The transposition algorithm is described for instance in [16]. The principle is very simple: it consists in splitting the matrix into 32×32 sub-matrices. Each sub-matrix is then read in a coalescent way from global memory by work-groups made of 32 work-items and loaded into the cache memory of the work-group. Once in the cache, the sub-matrix is transposed. Then, the transposed sub-matrix is copied in a coalescent way, at the good position in the global memory.

	time (s)
AMD Phenom II x4 945 (1 core)	192
AMD Phenom II x4 945 (4 cores)	59
AMD Radeon HD5850	1.43
NVIDIA GTX 460	2.48
NVIDIA Geforce GTX470	0.93

TABLE 2. Simulation times for several CPU and GPU devices

Without this trick, the whole algorithm is still fast. But it is approximately ten times slower than the optimized algorithm. This behavior is classical in GPU programing: because the computations are very fast, the limiting factor is often imposed by data read or write operations into the GPU global memory.

For each time step the algorithm is thus:

- compute the fluxes balance in the x-direction for each cell of each row of the grid. A row is associated to one work-group and one cell to one work-item;
- transpose the grid (exchange x and y) with an optimized memory transfer algorithm;
- compute the fluxes balance in the y-direction for each row of the transposed grid. Memory access are optimal.
- transpose again the grid.

In Table 2, we give the computation time of the resulting algorithm for different CPU and GPU devices. The mesh is a 256x256 grid. We have used exactly the same OpenCL implementation. We observe an excellent scaling of our implementation on massively parallel GPU devices.

In Figure 3, we present the density and pressure profiles for the shock-bubble interaction at time $t = 600\mu s$. The grid for this computation is made of 3000x1000 cells.

4. An unstructured DG approximation of electromagnetic waves

We will now present a different application of GPU simulation. Figure 4 represents a diode. The diode is used for generating electron beams. If the beam is intense enough, it is possible to generate X-rays when the electrons impinge the anode. At first, we model only the electromagnetic waves. The electrons will be addressed in Section 5.

We consider the electric field $E = (E_1, E_2, 0)^T$, the electric current $J = (j_1, j_2, 0)^T$ and the magnetic field $H = (0, 0, H_3)^T$. We write the 2D Maxwell equations in the so-called Transverse Electric (TE) mode as a first order linear hyperbolic system

$$W = (E_1, E_2, H_3)^T,$$
$$\partial_t W + A^i \partial_i W = J \quad \text{(sum on repeated indices)} \tag{5}$$
$$A^1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \quad A^2 = \begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}, \quad J = \begin{pmatrix} j_1 \\ j_2 \\ 0 \end{pmatrix},$$

with the boundary conditions

$$H_3 - n_1 E_2 + n_2 E_1 = s \text{ on } \Gamma_S, \quad -n_1 E_2 + n_2 E_1 = 0 \text{ on } \Gamma_M.$$

The boundary term s is given, for instance, by the incident field

$$s = H_3^{\text{inc}} - n_1 E_2^{\text{inc}} + n_2 E_1^{\text{inc}}.$$

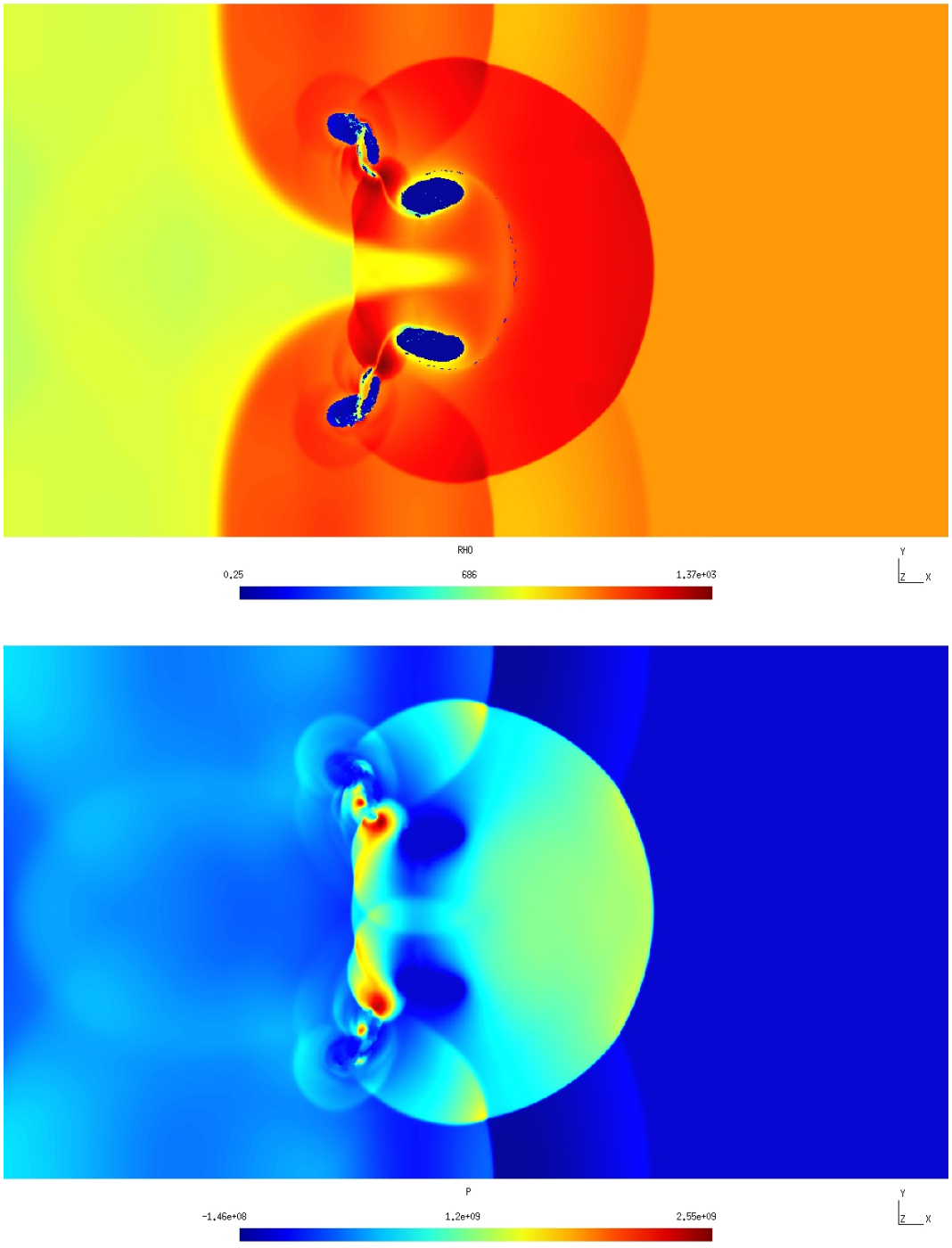

FIGURE 3. Shock-bubble interaction at time $t = 600\mu s$ on a 3000x1000 grid. Density (top) and pressure (bottom).

For simplicity, we have written only the Maxwell equations in cartesian geometry. It is also possible to consider the axisymmetric equations (see [7]). The numerical results presented in Figure 5 are done in axisymmetric geometry.

We consider a mesh of the domain Ω, such as in Figure 4. In each cell L the fields are approximated by

$$W_L(x,t) \simeq w_{L,j}(t)\psi_{L,j}(x), \quad \{\psi_{L,j}\} \text{ basis of } P_2(\mathbb{R}^2)^3$$

The Discontinuous Galerkin (DG) upwind weak formulation [14], [10], [4] reads

$$\int_L \partial_t W_L \cdot \psi_L - \int_L W_L \cdot A^i \partial_i \psi_L + \int_{\partial L} (A^i n_i^+ W_L + A^i n_i^- W_R) \cdot \psi_L = \int_L J \cdot \psi_L, \tag{6}$$

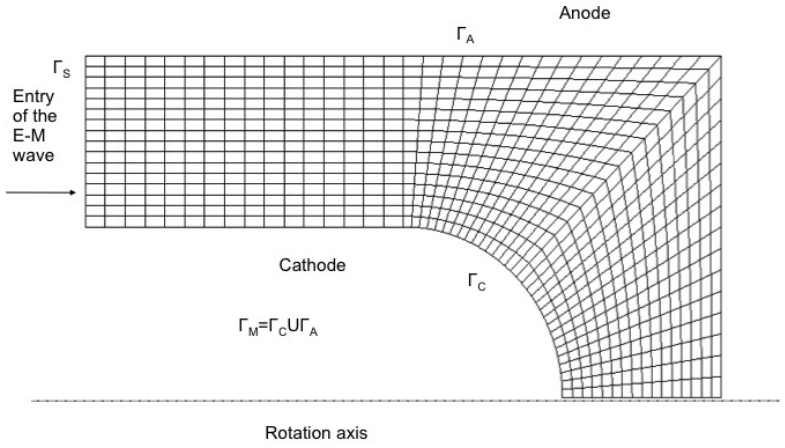

FIGURE 4. Diode in axisymmetric geometry. The entry of the electromagnetic pulse is at the left. The (positive) cathode and (negative) anode are represented by metallic boundary conditions. The space between the cathode and the anode is at vacuum. The mesh of the computational domain is also represented.

where n is the normal vector on ∂L oriented from the cell L to the neighboring cells R and

$$x^+ = \max(0, x), \quad x^- = \min(0, x).$$

Setting $\psi_L = \psi_{L,j}$ in 6, we end up with a system of ordinary linear differential equations for the $w_{L,j}(t)$. We list below a few features of our implementation:

- The cells are quadratic curved quadrilaterals.
- We use an exact numerical integration (16 Gauss-Legendre quadrature points in the cells and 4 points on each edge).
- We use a divergence-cleaning correction, described in [15], in order to enforce the divergence condition on the electric field.

The parallel algorithm is then as follows:

- Initialization: we compute and invert on the CPU the local mass matrix of each cell. These matrices come from the following term of the DG formulation (6)

$$\int_L \partial_t W_L \cdot \psi_L.$$

  We then send all the data to the GPU.
- In the first pass of each time step, we associate to each Gauss point of each edge one work-item. One work-group corresponds to exactly one cell. We compute the flux and store it into global memory.
- In the second pass, we associate to each cell a work-group and to each basis function of the cell a work-item. We compute the time derivative of the $w_{L,j}$ using the volume terms in the DG weak formulation (6), the previously computed fluxes and the stored inverted mass matrices. It is not possible to perform the fluxes pass and the volumic pass at the same time, because with this approach two different processors would try to add their contributions at the same memory location at the same time, which leads to wrong results. That's why we have to store the intermediate results and "reorganize the parallelism" between the two passes.
- For the time integration, we use a simple second order Heun scheme. This step is parallelized in the obvious way (one work-item per vector location).

We can make the following remarks on our implementation:

- We do not use the linearity of the equations, so that our algorithm could be reused for other hyperbolic systems.

	time (s)
AMD Phenom II x4 945 (1 core)	563
AMD Phenom II x4 945 (4 cores)	185
NVIDIA NV320M	146
AMD Radeon HD5850	23
NVIDIA Geforce GTX260	17
NVIDIA Geforce GTX470	3

TABLE 3. DG Maxwell solver simulation times for several CPU and GPU devices.

- For the diode test case, we use a curved structured grid, but our implementation can also handle fully unstructured grids.
- Our approach is different from [12]. In [12], the algorithm is three-dimensional and relies explicitly on the linearity of the Maxwell equations. In addition, the computations are made in only one kernel. This avoids the intermediate storage of the fluxes. But it implies that some processing elements are not optimally exploited, because the number of Gauss points on the edges and in the cells are not necessarily the same.

It must be noted that the implementation is rather different compared to the shock-bubble interaction. We associate a processor to each Gauss point instead of each cell. This choice is here again imposed by the memory access pattern. Because of the high order approximation, each cell owns a relatively high quantity of data. If we had associated a processor to each cell, two neighboring work-items would not access neighboring data in memory. With a finer grain parallelism, we achieve better memory bandwidth.

In Table 3, we compare the same OpenCL implementation for several multicore devices. We obtain very interesting speedups for GPU devices.

5. A PIC approximation of charged particle beams

In this section, we explain how we compute the electron beam and how we couple it with the previously presented Maxwell solver. We have to approximate the electron distribution function $f(x, v, t)$, which measures the quantity of electrons having the velocity v and position x at time t. The distribution function satisfies the Vlasov equation (see (7) below). In addition, the motions of electrons generate a current J at the right hand side of the Maxwell equations (5). In principle, the Vlasov equation is a first order hyperbolic conservation law that could be solved also by a DG scheme. However, because it is set in four-dimensional phase space, it is very expensive to solve it with this method. We will follow a more economical approach based on particle approximation: the Particle-In-Cell method.

In this method, the distribution function of electrons f is approximated by weighted particles (δ is the Dirac measure)

$$f(x, v, t) = \sum_k \omega_k \delta(x - x_k(t))\delta(v - x_k'(t)).$$

The particles move according to the Newton's equations of motion

$$m x_k'' = q(E + x_k' \wedge H), \quad E = (E_1, E_2, 0)^T, \quad H = (0, 0, H_3)^T.$$

And the electronic current is a measure given by

$$j(x, t) = \sum_k \omega_k \delta(x - x_k(t)) x_k'(t).$$

We can recover the following equations

- Vlasov equation
$$\partial_t f + v \cdot \nabla_x f + \frac{q}{m}(E + v \wedge H) \cdot \nabla_v f = 0. \tag{7}$$
- Charge conservation
$$\partial_t \rho + \nabla \cdot j = 0, \quad \rho(x, t) = q \int_v f(x, v, t) dv.$$

Nb de mailles	Nb d'itér.	Nb de part.	CPU 1 cœur	CPU 4 cœurs	Nvidia GTX 470	CPU_4/GPU
1024	2530	46000	723 s.	446 s.	21 s.	**21.2**
1024	2530	92000	914 s.	611 s.	31 s.	**19.7**
4096	5060	70000	4450 s.	2319 s.	65 s.	**35.7**
4096	5060	110000	4809 s.	2652 s.	97 s.	**27.3**

TABLE 4. PIC-DG coupling algorithm. Performance comparison of several CPU and GPU devices.

- Electric field divergence condition (if true at time $t = 0$)

$$\nabla \cdot E = \rho.$$

At the initial time, there is no charge in the diode. The electrons are emitted at the cathode if the normal electric field is strong enough, until it cancels (this is the so-called Child-Langmuir law).

More precisely, we use the following algorithm for a cell L that touches the cathode (n is the outward normal vector on the cell boundary):

- if $E \cdot n > 0$ on $\partial L \cap \Gamma_C$ then compute

$$\delta_L = \rho_L - \int_{\partial L \setminus \Gamma_C} E \cdot n$$

  where $\rho_L = \sum_{x_k \in L} \omega_k$ is the charge in the cell L
- if $\delta_L < 0$, create n_e random particles in the cell L with weights δ_L / n_e

When the particles arrive at the anode, they simply leave the domain.

The PIC algorithm is made of two parts. First, we have to move the particles. Then, we have to compute the contribution of each particle to the electric current of each cell. The first part is easy to program, while the other is more tricky, because we have to avoid concurrent memory access.

The easy part of the PIC algorithm consists in the following steps:

- emission: we use the previously given algorithm. The random positions are given by independent van der Corput sequences.
- Particle moves: at each time step we associate one work-item to each particle. We move the particle and find its new cell location. We assume that the particle does not cross more than one cell. The algorithm works on an unstructured grid.

Then we have to compute the contribution of the electric current in the right hand side of the DG formulation (6). This is more subtle because we have to avoid concurrent memory write operations. It is not possible to keep the same distribution of the particles to the work-item. Indeed, two particles in the same cell could then try to add their contribution to the current at the same time, which would lead to wrong results. The electric current algorithm is rather the following:

- we first sort the list of particles according to their cell numbers. For this, we use an OpenCL optimized radix sorting algorithm described in [9]. Then, it is easy to know how many particles are in each cell. This approach is classical in parallel PIC implementations (see for instance [1]).
- we associate to each cell a work-item. Then, for each cell it is possible to loop on its particles in order to compute their contributions to the current

$$\int_L J \cdot \psi_L = \sum_{x_k \in L} \omega_k (x'_{1,k}(t), x'_{2,k}(t), 0)^T \cdot \psi_L(x_k(t)).$$

We compare the full DG-PIC coupling algorithm for several CPU and GPU devices in Table 4. We still observe good GPU performances, but because of the necessary sorting in the PIC current computation, the speedups are less interesting than for the full DG algorithm.

We also present in Figure 5 the electron beam and the radial electric field at three different times.

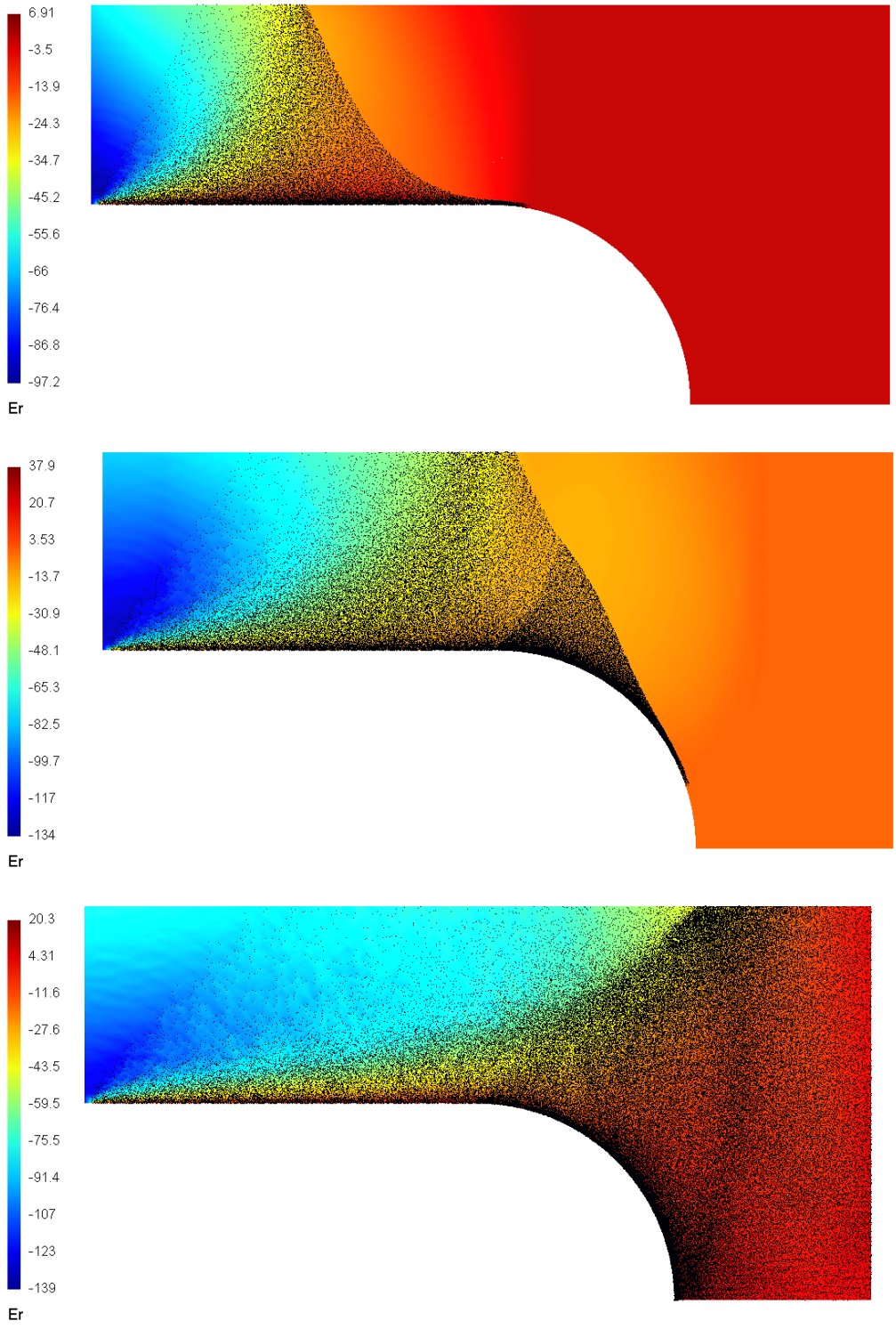

FIGURE 5. PIC-DG coupling. The colors represent the radial component of the electric field. The black dots represent the positions of the particles. The simulation times are $t = 0.22$ (top), $t = 0.33$ (middle) $t = 0.55$ (bottom).

6. CONCLUSION

We have implemented three different conservation laws solvers with OpenCL that works both for multicore CPU and GPU. The solvers are: a first order Finite Volume solver, a high order Discontinuous Galerkin solver, a Particle-In-Cell solver. For achieving good performance, we had to deal with several important features of multicore programming. The most important feature is related to memory access. Because of the high number of processors, the computations are very fast and the memory access becomes the limiting factor. We have to

organize the data into memory in such a way that neighboring processors access neighboring data. This leads to different implementations of the first order scheme and the high order scheme.

Another important feature is that the GPUs are SIMD computers: the processors share the same memory. We have thus to avoid concurrent memory access that would give wrong numerical results. In some cases this leads to complicated algorithms, such as particle sorting in the PIC method.

Anyway, despite these complications, the resulting performances are very impressive. The observed speedups are of the order 100 compared to a standard single core CPU implementation. It is also possible to solve real life problems on complex geometries.

This leads us to believe that multicore processors and corresponding software frameworks, such as OpenCL, could not be avoided in the future of scientific computing.

References

[1] Aubert, D.; Amini, M.; David, R. A Particle-Mesh Integrator for Galactic Dynamics Powered by GPGPUs. Lecture Notes in Computer Science, 5544, 874–883, (2009).
[2] Bachmann, M.; Helluy, P.; Jung, J.; Mathis, H. and Müller, S. Random sampling remap for compressible two-phase flows, 2011, submitted. Preprint at http://hal.archives-ouvertes.fr/hal-00546919
[3] Barberon, T; Helluy, P. and Rouy, S. Practical computation of axisymmetrical multifluid flows. Int. J. Finite Vol., 1(1):34, 2004.
[4] Bourdel, F.; Mazet, P.-A. and Helluy, P. Resolution of the non-stationary or harmonic maxwell equations by a discontinuous finite element method. In 10th international conference on computing methods in applied sciences and engineering, pages 1–18. Nova Science Publishers, Inc., New York, 1992.
[5] Bouchut, F. Nonlinear stability of finite volume methods for hyperbolic conservation laws and well-balanced schemes for sources. Frontiers in Mathematics. Birkhäuser Verlag, Basel, 2004.
[6] Chalons, C. and Coquel, F. Computing material fronts with Lagrange-Projection approach. HYP2010 Proc. http://hal.archives-ouvertes.fr/hal-00548938/fr/.
[7] Crestetto, A. Optimisation de méthodes numériques pour la physique des plasmas. Application aux faisceaux de particules chargées. PhD thesis. Strasbourg, 2012.
[8] Crestetto A. and Helluy P. An OpenCL tutorial. 2010. http://www-irma.u-strasbg.fr/~helluy/OPENCL/tut-opencl.html
[9] Helluy, P. A portable implementation of the radix sort algorithm in OpenCL, http://hal.archives-ouvertes.fr/hal-00596730/fr/. Technical report, (2011).
[10] Johnson, C. and Pitkäranta, J. An analysis of the discontinuous Galerkin method for a scalar hyperbolic equation. Math. Comp. 46 (1986), no. 173, 1–26
[11] Khronos Group. OpenCL online documentation. http://www.khronos.org/opencl/
[12] Klockner, A.; Warburton, T.; Bridge, J. and Hesthaven, J. S. Nodal discontinuous Galerkin methods on graphics processors, Journal of Computational Physics, Volume 228, Issue 21, 20 November 2009, Pages 7863-7882
[13] Kokh, S. and Lagoutière, F. An anti-diffusive numerical scheme for the simulation of interfaces between compressible fluids by means of the five-equation model. J. Computational Physics, 229, 2773-2809, 2010.
[14] LeSaint, P.; Raviart, P. A. On a finite element method for solving the neutron transport equation, Mathematical aspects of finite elements in partial differential equations (C. de Boor, Ed.), Academic Press, 89–145, (1974).
[15] Munz, C.-D.; Omnes, P.; Schneider, R.; Sonnendrücker, E.; Voß, U. Divergence Correction Techniques for Maxwell Solvers Based on a Hyperbolic Model, Journal of Computational Physics **161**, 484–511, (2000).
[16] Ruetsch, G. and Micikevicius, P. Optimizing Matrix Transpose in CUDA. NVIDIA GPU Computing SDK, 1 – 24. 2009.
[17] Saurel, R. and Abgrall, R. A simple method for compressible multifluid flows. SIAM J. Sci. Comput. 21 (1999), no. 3, 1115–1145
[18] Toro, E. F. Riemann solvers and numerical methods for fluid dynamics, 2nd edition. Springer, 1999