# Chapter 1
# Asynchronous OpenCL/MPI numerical simulations of conservation laws

Philippe Helluy[1,2], Thomas Strub[3], Michel Massaro[1,2], Malcolm Roberts[1,2]

**Abstract** Hyperbolic conservation laws are important mathematical models for describing many phenomena in physics or engineering. The Finite Volume (FV) method and the Discontinuous Galerkin (DG) methods are two popular methods for solving conservation laws on computers. Those two methods are good candidates for parallel computing:

- they require a large amount of uniform and simple computations,
- they rely on explicit time-integration,
- they present regular and local data access pattern.

In this paper, we present several FV and DG numerical simulations that we have realized with the OpenCL and MPI paradigms. First, we compare two optimized implementations of the FV method on a regular grid: an OpenCL implementation and a more traditional OpenMP implementation. We compare the efficiency of the approach on several CPU and GPU architectures of different brands. Then we give a short presentation of the DG method. Finally, we present how we have implemented this DG method in the OpenCL/MPI framework in order to achieve high efficiency. The implementation relies on a splitting of the DG mesh into sub-domains and sub-zones. Different kernels are compiled according to the zones properties. In addition, we rely on the OpenCL asynchronous task graph in order to overlap OpenCL computations, memory transfers and MPI communications.

**Key words:** OpenCL, MPI, task graph, conservation laws, discontinuous Galerkin approximation

[1] IRMA, Université de Strasbourg, [2] Inria Tonus, [3] AxesSim Illkirch, France

## 1.1 Introduction

Hyperbolic conservation laws are a particular class of Partial Differential Equations (PDE) models. They are present in many fields of physics or engineering. It is thus very important to have efficient software tools for solving such systems. The unknown of a system of conservation laws is a vector $W(x,t) \in \mathbb{R}^m$ that depends on a space variable $x = (x^1 \dots x^d)$ and time $t$. The vector $W$ is called the vector of conservative variables. In this work we shall consider a space dimension $d = 2$ or $d = 3$. Generally, the space variable $x$ belongs to a bounded domain $\Omega \subset \mathbb{R}^d$. The system of conservation reads

$$\partial_t W + \partial_k F^k(W) = 0. \tag{1.1}$$

In this formula, we use the following notations:

- The partial derivative operators are denoted by

$$\partial_t = \frac{\partial}{\partial_t}, \quad \partial_k = \frac{\partial}{\partial x^k}.$$

- We adopt the sum-on-repeated-indices convention

$$\partial_k F^k(W) = \sum_{k=1}^{d} \partial_k F^k(W).$$

- The functions $F^k(W) \in \mathbb{R}^m$, $k = 1 \dots d$, characterize the physical model that we wish to represent. It is classic to consider a space vector $n = (n_1 \dots n_d) \in \mathbb{R}^d$ and to also define the *flux* of the system

$$F(W,n) = F^k(W) n_k.$$

System (1.1) is supplemented by an initial condition

$$W(x,0) = W_0(x), \tag{1.2}$$

at time $t = 0$, and conditions on the boundary $\partial\Omega$ of $\Omega$. For example, one can prescribe the value of $W$ on the boundary

$$W(x,t) = W_b(x,t), \quad x \in \partial\Omega. \tag{1.3}$$

Generally, the system (1.1), (1.2), (1.3) admits a unique solution if it satisfies the hyperbolicity condition: the Jacobian matrix of the flux

$$\nabla_W F(W,n)$$

is diagonalizable with real eigenvalues for all values of $W$ and $n$.

The above mathematical framework is very general. It can be applied to electromagnetism, fluid mechanics, multiphase flows, magneto-hydro-dynamics (MHD), Vlasov plasmas, *etc.* Let us just give two examples:

- The Maxwell equations describe the evolution of the electric field $E(x,t) \in \mathbb{R}^3$ and the magnetic field $H(x,t) \in \mathbb{R}^3$. The conservative variables are the superimposition of these two vectors $W = (E^T, H^T)^T$ (thus $m = 6$) and the Maxwell flux is given by

$$F(W,n) = \begin{bmatrix} 0 & -n\times \\ n\times & 0 \end{bmatrix} W.$$

  In Section 3 we present numerical results obtained with the Maxwell equations.
- In fluid mechanics, the Euler equations describe the evolution of a compressible gas of density $\rho$, velocity $u = (u^1, u^2, u^3)$ and pressure $p$. The conservative variables are given here by

$$W = (\rho, \rho u^T, p/(\gamma - 1) + 1/2\rho u \cdot u)^T$$

  and the flux by

$$F(W,n) = (\rho u \cdot n, \rho u \cdot n u^T + p n^T, \{\gamma p/(\gamma - 1) + 1/2\rho u \cdot u\}\, u \cdot n)^T,$$

  where $\gamma > 1$ is the polytropic exponent of the gas. The MHD equations are a generalization of the Euler equations for taking into account magnetic effects in conductive compressible gas. The MHD system is a complicated system of conservation laws, with $m = 9$. It is not the objective of this work to detail the MHD equations. For this we refer for instance to [11]. In Section 2, we present numerical results obtained with the MHD equations.

Because of their numerous fields of application, many numerical methods have been developed for the resolution of hyperbolic conservation laws. For instance the finite volume (FV) and discontinuous Galerkin (DG) method are very popular. They are easy to program on a standard parallel computer thanks to subdomain decomposition. However, on new hybrid architectures, the efficient implementation of those methods is more complex. It appears that there is possibility of optimizations. In this paper, we explore several numerical experiments that we have made for solving conservation laws with the FV and DG methods on hybrid computers. OpenCL and MPI libraries are today available on a wide range of platform, making them a good choice for our optimizations. It is classic to rely on OpenCL for local computations and on MPI for communications between accelerators. In addition, in our work we will see that it is interesting to also use the OpenCL asynchronous task graph in order to overlap OpenCL computations, memory transfers and MPI communications.

In the first part of this paper, we compare a classic OpenMP optimization of a FV solver to an OpenCL implementation. We show that on a standard multicore CPU, we obtain comparable speedups between the OpenMP and the OpenCL implementation. In addition, using several GPU accelerators and MPI communications between them, we were able to make computations that would be unattainable with more classic architectures.

Our FV implementation is limited to regular grids. In the second part of the paper, we thus describe an efficient implementation of the DG algorithm on unstructured grids. Our implementation relies on several standard optimizations: local memory prefetching, exploitation of the sparse nature of the tensor basis, and MPI subdomain decomposition. Other optimizations are less common: idling work-item for minimizing cache prefetching and asynchronous MPI/OpenCL communication.

## 1.2 Comparison of an OpenCL and an OpenMP solver on a regular grid

### *1.2.1 FV approximation of conservation laws*

The FV and DG method construct a discontinuous approximation of the conservative variables $W$. In the case of the FV method, the approximation is piecewise constant. In the case of the DG method, the approximation is piecewise polynomial. It is therefore necessary to extend the definition of the flux $F(W,n)$ at a discontinuity of the solution. We consider thus a spatial discontinuity $\Sigma$ of $W$. The discontinuity is oriented by a normal vector $n_{LR}$. We use the following convention: the "left" (L) of $\Sigma$ is on the side of $-n_{LR} = n_{RL}$ and the "right" (R) is on the side of $n_{LR}$. We denote by $W_L$ and $W_R$ the values of $W$ on the two sides of $\Sigma$. The numerical flux is then a function

$$F(W_L, W_R, n_{LR}).$$

A common choice is to take

$$F(W_L, W_R, n) = \frac{F(W_L, n) + F(W_R, n)}{2} - \frac{s}{2}(W_R - W_L), \qquad (1.4)$$

where $s$ is called the numerical viscosity. It is a supremum of all the wave speeds of the system. For more simplicity, in this section we consider the two-dimensional case $d = 2$ and a square domain $x = (x^1, x^2) \in \Omega = ]0, L[\times]0, L[$. The space step of the grid is $\Delta x = L/N$ where $N$ is a positive integer. The grid cells are squares of size $h \times h$. The cell centers are defined by $x_{i,j} = ((i + \frac{1}{2})\Delta x, (j + \frac{1}{2})\Delta x)$. We also consider a time step $\Delta t$ and the times $t^n = n\Delta t$. We look for an approximation $W^n_{i,j}$ of $W$ at the cell centers $x_{i,j}$ and at time $t^n$

$$W_{i,j}^n \simeq W(x_{i,j}, t^n). \tag{1.5}$$

Let $\nu^1$ and $\nu^2$ be normal vectors pointing in the $x^1$ and $x^2$ direction, respectively, so that

$$\nu^1 = (1,0)^T, \quad \nu^2 = (0,1)^T.$$

We adopt a Strang splitting strategy: for advancing the numerical solution from time step $t^n$ to time step $t^{n+1}$, we first solve the finite volume approximation in direction $x^1$

$$\frac{W_{i,j}^* - W_{i,j}^n}{\Delta t} + \frac{F(W_{i,j}^n, W_{i+1,j}^n, \nu^1) - F(W_{i-1,j}^n, W_{i,j}^n, \nu^1)}{\Delta x} = 0, \tag{1.6}$$

and then in direction $x^2$

$$\frac{W_{i,j}^{n+1} - W_{i,j}^*}{\Delta t} + \frac{F(W_{i,j}^n, W_{i,j+1}^n, \nu^2) - F(W_{i,j-1}^n, W_{i,j}^n, \nu^2)}{\Delta x} = 0. \tag{1.7}$$

On the boundary cells, we simply replace, in the previous formulas, the missing values of $W$ by the boundary values (1.3).

### *1.2.2 OpenMP implementation of the FV scheme*

The chosen numerical scheme is very simple. We have first written a sequential C implementation of the algorithm. We only consider results with single precision. We have also decided to apply the FV scheme to the ideal MHD system with divergence correction. The MHD system models the coupling of a compressible fluid with a magnetic field. It contains $m = 9$ conservative variables and the numerical flux can be a rather complex function. The numerical simulations thus require heavy computations and are well adapted to GPU hardware. For more details and bibliography on the MHD equations, we refer to [11].

In this first version we simply loop on all the rows $i$ of the grid and then on all the columns $j$ for scanning the grid points and applying the $x^1$-step (1.6). We compile the code with a recent version of gcc without optimizations. If we activate optimizations (-O3 compilation option of gcc), we obtain an easy speedup of 5. In order to observe cache access effects, we consider only large grids with sizes bigger than $1024 \times 1024$. It is clear that our first scanning strategy is not optimal, because it induces many cache misses. We have thus also implemented a tiling strategy, which consists in scanning smaller subgrids of the large grid. The subgrid size is chosen in such a way that cache misses are reduced. With this additional optimization we obtain a speedup of 8 compared to the initial naive implementation. The next optimization step is then to parallelize the numerical scheme. We have thus also implemented an OpenMP version of the FV algorithm. It consists simply in adding par-

allel OpenMP directives before the most external loop when scanning each subgrid. With the tiling+OpenMP version of our code we are able to reach a speedup of 116 on a two-CPU SMP computer compared to the naive sequential code.

We use the optimized tiled OpenMP implementation as our reference for comparisons with OpenCL implementations (see Table 1.1 where the different implementations are compared).

### *1.2.3 OpenCL implementation of the FV scheme*

#### 1.2.3.1 OpenCL

It is necessary to adapt our code to new SIMD accelerators, such as GPUs, in order to decrease computation cost. For this, we have chosen OpenCL, which is a recent programming framework for driving such accelerators. A nice feature of OpenCL is that multicore CPUs are also considered as accelerators. The same program can thus be run without modification on a CPU or a GPU.

OpenCL means "Open Computing Language". It consists in a library of C functions, called from the host, for driving one or several accelerators (GPU or multicore CPU). It contains also a C-like programming language for writing the programs (the "kernels") that will run on the accelerators. In the OpenCL paradigm, the accelerators are called "devices". Each device possesses its own memory ("global memory"). Each device is made of compute units of several processors ("processing elements") that share a small fast-access cache memory ("local memory").

For practical reasons, OpenCL allows one to program the accelerator as if it had an arbitrary number of compute units and processing elements. The "virtual" compute units and processing elements are respectively called "work-groups" and "work-items" in the OpenCL terminology. The kernels are launched on the actual devices, compute units and processing elements through a mechanism of "command queues". Some rules have to be respected for efficient OpenCL programming:

- The work-items can all access the global memory of the device but can only access the scarce local memory of their work-group.
- If two or more work-items try to write at the same memory location, only one succeeds.
- The access to the local memory is much faster than the access to the global memory. If access to the global memory is mandatory, it is advised for faster access that neighboring work-items read/write at neighboring memory locations. Such optimal access is called "coalescent access". When an algorithm requires non regular memory access, a classic strategy is thus to prefetch in a coalescent way the data from the global memory into the local memory, then work on the data in local memory. When the work is

finished, the data are copied back into the global memory in a coalescent fashion.
- Finally, on a GPU, data transfers between the host and the global memory are very slow, because they are generally transported through the PCIe bus. Consequently, they should be avoided.

As explained above OpenCL exposes useful abstractions for driving generic SIMD architectures. In this section, we exploit these features. However OpenCL contains other useful abstractions:

- It offers the possibility to launch different kernels or memory transfer tasks on command queues attached to different devices. Another level of parallelism is thus available. The tasks among different command queues can be launched asynchronously. A mechanism of events allows one to describe the task dependencies.
- The kernel sources are compiled at runtime. It is thus possible to customize the kernels depending on the accelerators detected by OpenCL during execution.

These more advanced features are exploited in Section 3.

#### 1.2.3.2 Implementation

For the OpenCL version of our FV algorithm, we organize the data in a $(x_1, x_2)$ grid: each conservative variable is stored in a two-dimensional $(i, j)$ array. For advancing from time step $t^n$ to time step $t^{n+1}$:

- In principle, we associate a work-item to each cell of the grid and a work-group to each row. But OpenCL drivers generally impose a maximal work-group size. Thus when the row is too long it is also necessary to split the row and distribute it on several work-groups.
- We compute the flux balance in the $x_1$-direction for each cell of each row of the grid (see formula (1.6)).
- We then transpose the grid, which amounts to exchanging the $x_1$ and $x_2$ coordinates. The $(i, j) \to (j, i)$ transposition is performed on the two-dimensional array of each conservative variable. For ensuring coalescent memory access we adopt an optimized memory transfer algorithm [13] (see also [12]).
- We can then compute the fluxes balance in the $x_2$-direction (1.7) for each row of the transposed grid. Thanks to the previous transposition, memory access is coalescent.
- We again transpose the grid.

Let us mention that other strategies are possible. For instance in [12] the authors describe GPU computations of scalar ($m = 1$) elastic waves. The algorithm is based on two-dimensional tiling of the mesh into cache memory and registers in order to ensure fast memory access. However the tile size is

limited by the cache size and the number of unknowns $m$ in each grid cell. In our case for the MHD system we have $m = 9$ and the adaptation of the algorithm given in [12] would probably be inefficient.

We have tested this OpenCL implementation in several configurations and we can make the following comments:

- We can run the OpenCL code on a two-CPU SMP computer or GPUs of different brands, without modification. In addition, we obtain interesting speedups on SMP architectures. The OpenCL speedup for CPU accelerator is approximately 70% of the OpenMP speedup. It remains very good considering that the transposition algorithm probably deteriorates the memory access efficiency on CPU architectures. The fact that OpenCL is a possible alternative to OpenMP on multicore CPU has already been discussed in [14].
- On AMD or NVIDIA GPUs, the same version of our code achieves excellent performance.
- If we replace the optimized transposition by a naive unoptimized transposition algorithm the code runs approximately 10 times slower on GPUs. The coalescent memory access is thus an essential ingredient of the efficiency.

### *1.2.4 OpenCL/MPI FV solver*

We now modify the OpenCL implementation in order to address several GPU accelerators at the same time. This could theoretically be achieved by creating several command queues, one for each GPU device. However, as of today, when GPUs are plugged into different nodes of a supercomputer, the current OpenCL drivers are not able to drive at the same time GPUs of different nodes. Therefore, we have decided to rely on the MPI framework for managing the communications between different GPUs. This strategy is very common (see for instance [1, 4, 7] and included references).

We split the computational domain $\Omega$ into several subdomains in the $x^1$ direction. An example of splitting with four subdomains is presented on Figure 1.1. Then, each subdomain is associated to one MPI node and each MPI node drives one GPU. For applying the finite volume algorithm on a subdomain, it is necessary to exchange two layers of cells between the neighboring subdomains at the beginning of each time step. The layers are shaded in grey in Figure 1.1. On each MPI node, an exchange thus requires a GPU to CPU memory transfer of the cell layers, a MPI send/recv communication and a CPU to GPU transfer for retrieving the neighbor layers. The exchanged cells represent a small amount of the total grid cells, however, the transfer and communication time represent a non-negligible amount of the computation cost.

In our first OpenCL/MPI implementation, the exchange task is performed in a synchronous way: we wait for the exchange to be finished before com-

puting the fluxes balance in the subdomains. This explains why the speedup between the OpenCL code and the OpenCL/MPI code with four GPUs is approximately 3.5 (the ideal speedup would be 4). See Table 1.1).

Despite the synchronous approach, the OpenCL/MPI FV solver on structured grid is rather efficient. It has permitted us to perform computations on very fine grids that would be unreachable with standard parallel computers. For instance, we have performed two-fluid computations of shock-bubble interaction with grid size up to $40,000 \times 20,000$ in [6]. An animation (on a coarser grid) of this test case can be seen at `www.youtube.com/watch?v=c8hcqihJzbw`. This animation uses the possibility to share GPU buffers between OpenGL and OpenCL drivers (OpenGL/OpenCL interops). We have also performed magneto-hydro-dynamics (MHD) simulation on very fine grids in [11].

Now we would like to address the following drawbacks of the FV solver:

- The FV method is limited to first or second order approximation. In some applications, it is important to have access to higher order schemes.
- MPI and host/GPU communications take time, so it is important to provide asynchronous implementations for scalability with more MPI nodes.
- The previously described approach is limited to structured grids. We wish also to extend the method to arbitrary geometries.

In the next section we describe our implementation of a Discontinuous Galerkin (DG) solver that allows to achieving higher order, addressing general geometries, and overlapping computations and communications.

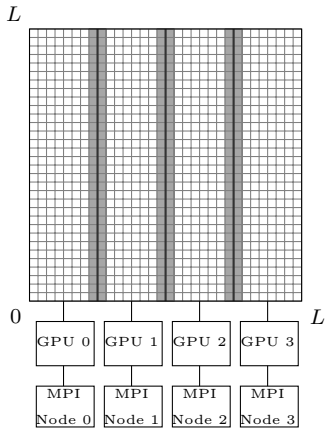


**Fig. 1.1** Subdomain MPI decomposition

| Implementation | Time | Speedup |
|---|---|---|
| OpenMP (Intel CPU 12 cores) | 717 s | 1 |
| OpenCL (Intel CPU 12 cores) | 996 s | 0.7 |
| OpenCL (NVIDIA K20) | 45 s | 16 |
| OpenCL (AMD HD7970) | 38 s | 19 |
| OpenCL + MPI (4 x NVIDIA K20) | 12 s | 58 |

**Table 1.1** Comparison of the different implementations of the FV scheme on a structured grid. Hardware : 2× Intel(R) Xeon(R) E5-2630 (6 cores, 2.3GHz), AMD Radeon HD 7970, NVidia K20m. On Intel CPUs hyperthreading was deactivated.

## 1.3 Asynchronous OpenCL/MPI Discontinuous Galerkin solver

We now present the Discontinuous Galerkin Method and explain our software design for keeping high performance in the GPU implementation.

### *1.3.1 The DG method*

#### 1.3.1.1 Interpolation on unstructured hexahedral meshes

The DG method is a generalization of the FV method. We suppose that dimension $d = 3$. We consider a mesh of the computational domain $\Omega$ made of cells $L_i$, $i = 1 \dots N_c$. In a cell $L$ of the mesh, the field is approximated by a linear combination of basis functions $\psi_j^L$

$$W(x,t) = W_L^j(t)\psi_j^L(x), \quad x \in L. \tag{1.8}$$

Each cell $L$ of the mesh is obtained by a geometrical mapping $\tau_L$ that transforms a reference element $\hat{L}$ into $L$. In theory the shape of the reference element $\hat{L}$ may be arbitrary. A classic choice is to consider tetrahedra [9]. In this work we prefer hexahedra, as in [5]. Building a tetrahedral mesh of $\Omega$ is generally easier. The basis functions of hexahedral cell are constructed from tensor products of one-dimensional functions. The tensor nature of the basis allows many optimizations of the algorithm that are not possible with tetrahedra.

We now give some details on the construction of the basis function. Let $D$ be the interpolation degree. We consider the $(D+1)$ Gauss-Legendre points in the interval $]0,1[$, $\xi_p$, $p = 0 \dots D$.

To these points, we can associate Lagrange polynomials of order $D$, $\ell_p(\xi)$ satisfying $\ell_p(\xi_q) = \delta_{p,q}$, where $\delta$ is the Kronecker symbol ($\delta_{p,q} = 1$ if $p = q$ and $\delta_{p,q} = 0$ otherwise).

We can also associate to each Gauss point the integration

$$\omega_p = \int_0^1 \ell_p(\xi) d\xi$$

The $(D+1)^3$ three-dimensional Gauss-Legendre points on the reference cube are obtained by tensor product. More precisely, for an integer $i \in \{0 \dots (D+1)^3 - 1\}$ there is a unique triplet $(p^1, p^2, p^3)$ in $\{0 \dots D\}^3$ such that $i = p^1 + (D+1)p^2 + (D+1)^2 p^3$. The corresponding Gauss-Legendre point and weight are then

$$\hat{x}_i = (\xi_{p^1}, \xi_{p^2} \xi_{p^3}), \quad \hat{\omega}_i = \omega_{p^1} \cdot \omega_{p^2} \cdot \omega_{p^3}.$$

We use the same kind of definition for the Gauss-Legendre points on the faces of the reference cube. In Figure 1.2 we have represented the Gauss-Legendre points for an order $D = 2$. The volume Gauss points are blue and the face Gauss points are green.

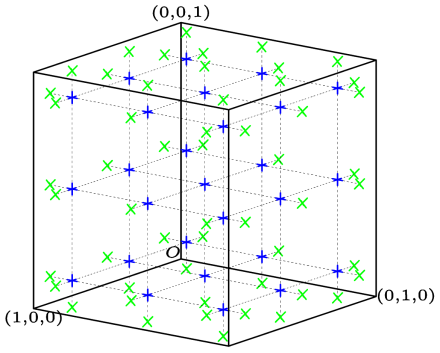


**Fig. 1.2** Volume and face Gauss-Legendre points in the reference cube.

Let $\hat{h}(\hat{x})$ be a function defined on the reference cell $\hat{L}$. We can then approximate the integral of this function by the integration rule

$$\int_{\hat{L}} \hat{h}(\hat{x}) \simeq \sum_k \hat{\omega}_k \hat{h}(\hat{x}_k). \tag{1.9}$$

A similar formula holds for the integration on the faces of the reference cube.

To each Gauss point $\hat{x}_i$ we associate a reference basis function that we define thanks to a tensor product of one-dimensional Lagrange polynomials

$$\hat{\psi}_i(\hat{x}) = \ell_{p^1}(\hat{x}^1) \ell_{p^2}(\hat{x}^2) \ell_{p^3}(\hat{x}^3).$$

With this choice, we have the interpolation property

$$\hat{\psi}_j(\hat{x}_i) = \delta_{i,j}.$$

For defining the transformation $\tau_L$ that maps the reference cell $\hat{L}$ to the current cell $L$ we first define the eight nodes $\hat{N}^k$ of the reference element

$$(\hat{N}^1,\hat{N}^2,\hat{N}^3,\hat{N}^4,\hat{N}^5,\hat{N}^6,\hat{N}^7,\hat{N}^8) =$$

$$\begin{pmatrix} 0&1&0&1&0&1&0&1\\ 0&0&1&1&0&0&1&1\\ 0&0&0&0&1&1&1&1 \end{pmatrix}.$$

As in the finite element method, we define the shape functions

$$\begin{aligned}
\phi_1(\hat{x}) &= (1-\hat{x}^1)(1-\hat{x}^2)(1-\hat{x}^3),\\
\phi_2(\hat{x}) &= \hat{x}^1(1-\hat{x}^2)(1-\hat{x}^3),\\
\phi_3(\hat{x}) &= (1-\hat{x}^1)\hat{x}^2(1-\hat{x}^3),\\
\phi_4(\hat{x}) &= \hat{x}^1\hat{x}^2(1-\hat{x}^3),\\
\phi_5(\hat{x}) &= (1-\hat{x}^1)(1-\hat{x}^2)\hat{x}^3,\\
\phi_6(\hat{x}) &= \hat{x}^1(1-\hat{x}^2)\hat{x}^3,\\
\phi_7(\hat{x}) &= (1-\hat{x}^1)\hat{x}^2\hat{x}^3,\\
\phi_8(\hat{x}) &= \hat{x}^1\hat{x}^2\hat{x}^3.
\end{aligned}$$

The shape functions also satisfy a nodal property

$$\phi_i(\hat{N}^j) = \delta_{i,j}.$$

We now denote by $N_L^k$ the nodes of the current cell $L$ and the mapping is defined by

$$\tau_L(\hat{x}) = \phi_k(\hat{x}) N_L^k.$$

The basis function of cell $L$ are then defined by transporting the reference basis functions:

$$\psi_j^L(x) = \hat{\psi}_j(\tau_L^{-1}(x)) \Leftrightarrow \psi_j^L(\tau_L(\hat{x})) = \hat{\psi}_j(\hat{x}).$$

Let $h(x)$ be a function defined on the cell $L$. For computing the integral of $h$ on $L$, we apply the formula

$$\int_L h(x) = \int_{\hat{L}} h(\tau_L(\hat{x}))\det\tau_L'(x)$$

and the quadrature rule (1.9).

Let us remark that from the above definitions, we have simply

$$W(\tau_L(\hat{x}_i),t) = W_L^i(t),$$

or in other words, the coefficients in the linear combination (1.8) are the values of the conservative variables $W$ at the Gauss points of cell $L$. For this reason the chosen basis is often called a nodal basis [8].

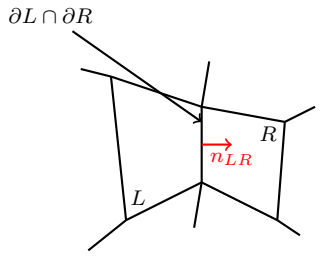


**Fig. 1.3** Mesh: notation conventions.

#### 1.3.1.2 DG formulation

The numerical solution satisfies the DG approximation scheme

$$\forall L, \forall i \quad \int_L \partial_t W \psi_i^L - \int_L F(W, W, \nabla \psi_i^L) + \int_{\partial L} F(W_L, W_R, n_{LR}) \psi_i^L = 0. \tag{1.10}$$

In this formula,

- $R$ denotes the neighbor cells along $\partial L$.
- $n_{LR}$ is the unit normal vector on $\partial L$ oriented from $L$ to $R$. See Figure 1.3.
- $F(W_L, W_R, n)$ is the numerical flux, which satisfies $F(W, W, n) = F^k(W) n_k$.

Inserting expansion (1.8) into (1.10) we obtain a system of differential equations satisfied by the $W_L^j(t)$. This system of differential equations can be solved numerically with a standard Runge-Kutta method.

The choice of interpolation we have described in the previous section is well adapted to the DG formulation.

- For instance, the nodal basis property ensures that we have direct access to the values of $W$ at the Gauss points. Consequently the mass matrix is diagonal.
- In the computation of the volume term $\int_L F(W, W, \nabla \psi_i^L)$ it is not necessary to loop on all the volume Gauss points. Indeed, the gradient of $\psi_i$ is nonzero only at the points that are aligned with point $i$ (see Figure 1.4).
- Finally, for computing the face integrals

$$\int_{\partial L} F(W_L, W_R, n_{LR})\psi_i^L$$

we have to extrapolate the values of $W$, which are known on the volume Gauss points, to the interfacial Gauss points. On tetrahedra, all the volume Gauss points would be involved in the interpolation. With our nodal hexahedral basis, only the volume Gauss points aligned with the considered interfacial Gauss point are needed (see Figure 1.2: for computing $W$ at a green point, we only need to know $W$ at the blue points aligned with this green point).

In the end, exploiting the tensor basis properties, the DG formulation (1.10) in a cell $L$ requires computations of complexity $\sim D^4$ instead of $\sim D^6$. For high orders, this is a huge improvement.

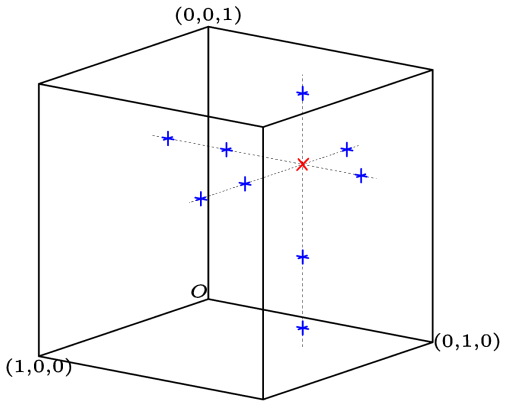


**Fig. 1.4** Non-zero values of the basis functions. The gradient of the basis function associated to the red point is nonzero only on the blue points

Beyond these useful optimizations that are also applied in sequential implementations, The DG method presents many advantages:

- It is possible to have different orders on different cells. No conformity is required between the cell and mesh refinement is thus simplified.
- The computations inside a cell only depend on the neighboring cells. The stencil is more compact than for high order FV methods. Memory accesses are thus adapted to GPU computations.
- High order inside a cell implies a high amount of local computations. This property is well adapted to GPU computations.
- Two level of parallelism can be easily exploited: a coarse grain parallelism, at the subdomain level, well adapted to MPI algorithms; and a fine grain parallelism, at the level of a single cell, well adapted to OpenCL or OpenMP.

But there are also possible issues that could make an implementation inefficient:

- We have first to take care of memory bandwidth, because unstructured meshes may imply non coalescent memory access.
- In addition, a general DG solver has to manage many different physical models, boundary conditions, interpolation basis, etc. If the implementation is not realized with care it is possible to end up with poorly coded kernels with many barely used variables or branch tests. Such wastage may remain unseen on standard CPUs with many registers and large cache memory, but is often catastrophic on GPUs.
- Finally, as we have already seen, MPI communications imply very slow GPU to Host and Host to GPU memory transfers. If possible, it is advised to hide communication latency by an overlapping with computations.

### *1.3.2 OpenCL kernel for a single GPU*

We first wrote optimized OpenCL kernels for computing, on a single cell $L$, the terms appearing in the DG formulation (1.10). After several experiments, we have found that an efficient strategy is to write a single kernel for computing the $\partial L$ and $L$ integration steps.

More precisely we construct a kernel with two steps.

In the first step ("flux step"), we compute the fluxes at the faces Gauss points and store those fluxes in the cache memory of the work-group. The work-items are distributed on the faces Gauss points. In this stage, $6(D+1)^2$ work-items are activated.

After a sync barrier, in the second stage ("collecting step"), we associate a work-item to each volume Gauss point $i$ and we collect the contributions of the other volume Gauss points $k$ coming from the numerical integration. We also collect the contribution from the faces fluxes stored in the first step. In this stage, $(D+1)^3$ work-items are activated.

We observe that when the order $D < 5$, which is always the case in our computations, $(D+1)^3 < 6(D+1)^2$ and then some work-items are idling in the collecting step.

We have also tried to split the computation into two kernels, one for the flux step and one for the collecting step, but it requires saving the fluxes into global memory, and in the end it appears that the idling work-items method is more efficient.

### *1.3.3 Asynchronous MPI/OpenCL implementation for several GPUs*

#### 1.3.3.1 Subdomains and zones

We have written a generic C++ DG solver called CLAC ("Conservation Laws Approximation on many Cores") for solving large problems on general hexahedral meshes. Practical industrial applications require a lot of memory and computations. It is thus necessary to address several accelerators in an efficient way.

We describe some features of the CLAC implementation.

First, the physical models are localized in the code: in practice, the user has to provide the numerical flux plus a few functions for applying boundary conditions, source terms, etc. With this approach it is possible to apply CLAC to very different physics: Maxwell equations, compressible fluids, MHD, etc. This approach is similar to the approach of A. Klöckner in [10].

We also adopt a domain decomposition strategy. The mesh is split into several domains, each of which is associated to a single MPI node, and each MPI node is associated to an OpenCL device (CPU or GPU).

In addition to the domain decomposition, in each domain we split the mesh into zones. We consider volume zones made of hexahedral cells and also interface zones made of cells faces. The role of a volume zone is to apply the source terms and the fluxes balance between cells inside the zone. The interface zones are devoted to computing the fluxes balance between cells of different volume zones. When an interface zone is at the boundary of the computational domain, it is used for applying boundary conditions. When it is situated at an interface between two domains, it is also in charge of the MPI communications between the domains. Interface zones also serves to manage mesh refinements between two volume zones. A simple example of mesh with two subdomains, three volume zones and five interface zones is given in Figure 1.5 and a schematic view of the same mesh is represented in Figure 1.6. We observe in this figure that simple non-conformities are allowed between volume zones (for instance neighboring volume zones 2 and 3 do not have the same refinement).

Finally, a zone possesses identical elements (same order, same geometry, same physical model). Thus, different computation kernels are compiled for each zone, in order to save registers and branch tests. We have observed that this aspect is very important for achieving high efficiency. For example, it is possible to simplify the kernel that compute the fluxes balance at an interface zone between two volume zones with conforming meshes. At an interface between volume zones with different refinements, the kernel is more complex, because the Gauss integration points are not aligned (see Interface zone 3 on Figure 1.6). The specialized kernels take advantage of the Gauss points alignment and store interpolation and geometrical data in constant

arrays or preprocessor macros. The speedup obtained using the specialized kernels as opposed to the generic kernels is reported in Table 1.2 for different interpolation orders.

| Order | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Speedup | 1.6 | 1.8 | 2.8 | 3.6 | 5.5 |

**Table 1.2** Speedup obtained with the specialized kernels

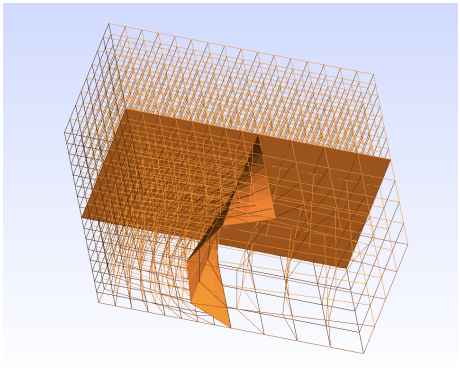

**Fig. 1.5** A simple non conforming mesh.

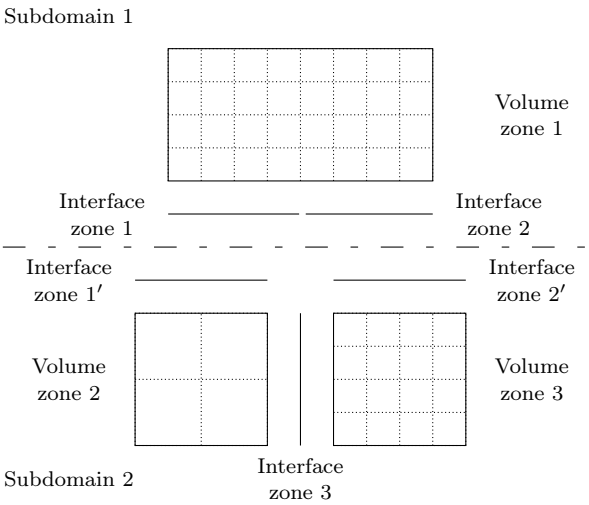


**Fig. 1.6** Schematic view of the simple mesh.

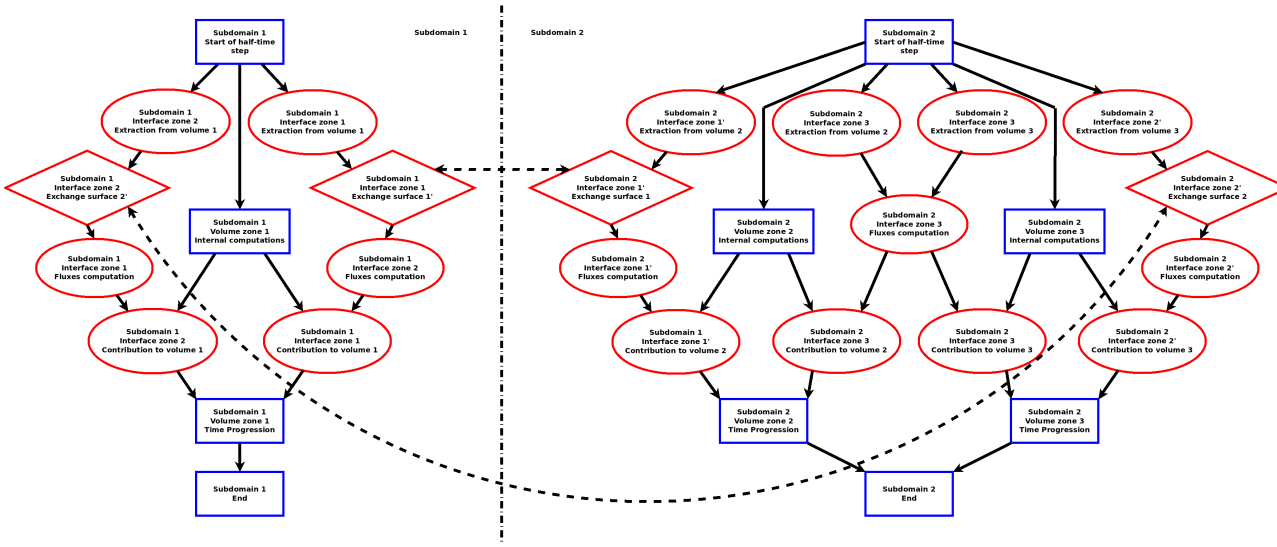


**Fig. 1.7** Task graphs for the simple mesh. One task graph for each MPI node.

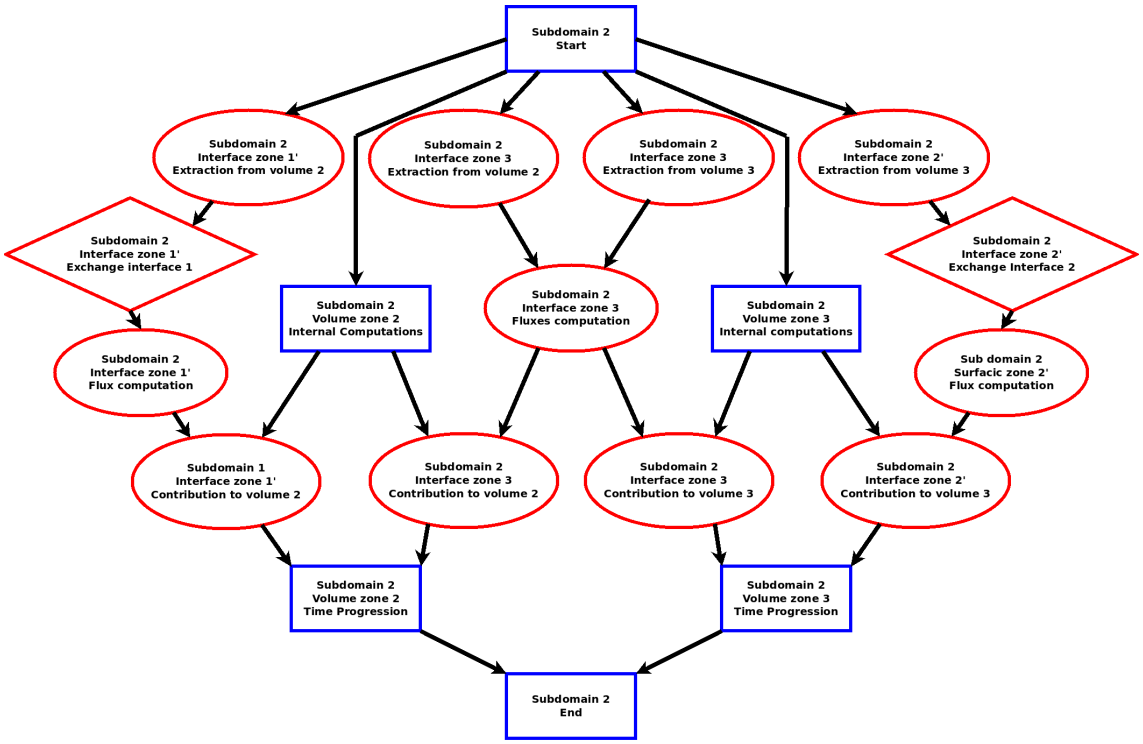


**Fig. 1.8** Task graph for subdomain 2

#### 1.3.3.2 Task graph

The zone approach is very useful to express the dependency between the different tasks of the DG algorithm.

We have identified tasks attached to volume or interface zones that have to be executed for performing a Runge-Kutta substep with the DG formulation. Those tasks are detailed in Table 1.3.

| Name | Attached to | Description |
|---|---|---|
| Extraction | Interface | Copy or extrapolate the values of $W$ from a neighboring volume zone |
| Exchange | Interface | GPU/Host transfers and MPI communication with an interface of another domain |
| Fluxes | Interface | Compute the fluxes at the Gauss points of the interface |
| Sources | Volume | Compute the internal fluxes and source terms inside a volume zone |
| Boundaries | Interface | Apply the fluxes of an interface to a volume zone |
| Time | Volume | Apply a step of the Runge-Kutta time integration to a volume zone |
| Start | Volume | Fictitious task: beginning of the Runge-Kutta substep |
| End | Volume | Fictitious task: end of the Runge-Kutta substep |

**Table 1.3** Tasks description

We express the dependencies between the tasks in a graph, and construct a task graph per subdomain. For instance, we have represented the graphs associated to the simple mesh of Figure 1.6 on Figure 1.7. For a better readability, we have also represented the task graph associated to subdomain 2 on Figure 1.8. The volume tasks are represented in blue rectangles, the interface tasks in red ellipses. The interface tasks that require MPI communication are in red rhombuses.

We observe in these figures that it is possible to perform the exchange tasks and the internal computations at the same time. It is thus possible to overlap communications and GPU/Host transfers by computations.

OpenCL contains events objects for describing task dependencies between the operations sent to command queues. It is also possible to create user events for describing interactions between the OpenCL command queues and tasks that are executed outside of a call to the OpenCL library. We have decided to rely only on the OpenCL event management for constructing the task dependencies.

Using asynchronous MPI communication requires calling `MPI_Wait` before launching tasks that depend on the completion of communication. We thus face a practical problem, which is to express the dependency between MPI and OpenCL operations in a non-blocking way. A possibility would have been to use an OpenCL "Native Kernel" containing MPI calls. A native kernel is a standard function compiled and executed on the host side, but that can be inserted into the OpenCL task graph. As of today, the native kernel feature is not implemented properly in all the OpenCL drivers. We thus had to adopt another approach in order to circumvent this difficulty.

Our solution uses the C++ standard thread class. It is also necessary to use an MPI implementation that provides the `MPI_THREAD_MULTIPLE` option. For programming the "Exchange" task, we first create an OpenCL user event. Then we launch a thread and return from the task. The main program flow is not interrupted and other operations can be enqueued. Meanwhile, in the thread, we start a blocking send/recv MPI operation for exchanging data between the boundary interface zones. Because the communication is launched in a thread, its blocking or non-blocking nature is not very important. When the communication is finished, we mark the OpenCL event as completed and exit the thread. The completion of the user event triggers the beginning of the enqueued tasks that depend on the exchange.

As we will see in the next section, this simple solution offers very good efficiency.

### *1.3.4 Efficiency analysis*

In this section we measure the efficiency of the CLAC implementation. Recently the so-called roofline model has been introduced for analyzing the efficiency of algorithm implementation on a single accelerator [15]. This model is based on several hardware parameters:

- First, we need to know the peak computation performances of the accelerator. This peak is measured with an algorithm with high computational intensity and very little memory access. It can be measured with a program that only requires register access. For instance, for a NVIDIA K20 accelerator, the peak performance is $P = 3.5\text{TFLOP/s}$.
- Another parameter is the memory bandwidth $B$ that measures the transfer speed of the global memory. For a NVIDIA K20 $B = 208\text{GB/s}$.

Not all algorithms are well adapted to GPU computing. Consider an algorithm (A) in which we count $N_{\text{ops}}$ operations (add, multiply, etc.) and $N_{\text{mem}}$ global memory operations (read or write). In [15], the computational intensity of the algorithm is defined by

$$I = \frac{N_{\text{ops}}}{N_{\text{mem}}}.$$

The maximal attainable performance of one GPU for this algorithm is then given by the roofline formula:

$$P_A = \max(P, B \times I).$$

We have counted the computational and memory operations of our DG implementation. The results are plotted in Figure 1.9. We observe that for order 1, the DG method is limited by the memory bandwidth. For higher orders, the method is limited by the peak performance of the GPU. The figure confirms that the DG method is well adapted to GPU architectures. We have also performed this analysis for the FV method described in Section 2. For large grids, the efficiency of the FV scheme is approximately 20 FLOP/B. The FV algorithm is thus also limited by the peak performance of the GPU. Our implementation of the FV scheme reaches approximately 800 GFLOP/s on a single K20 GPU.

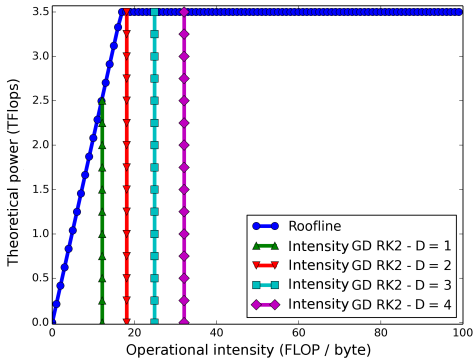


**Fig. 1.9** Roofline model and DG method. Abscissa: computational intensity $I$ (FLOP/B). Ordinate: Algorithm performance (TFLOP/s).

In Table 1.5, we present the results that we have measured with the asynchronous MPI/OpenCL implementation with 1, 2, 4 and 8 GPUs. For comparison, we also give in Table 1.4 the results of the synchronous execution (we wait that each task is completed before launching the next one). The computational domain $\Omega$ is a cube. The chosen model is the Maxwell system ($m = 6$). The mesh is made of several subdomains of $90^3$ cells. We perform single precision computations. The interpolation is of order $D = 3$. The algorithm requires storing three time steps of the numerical solution. With these parameters the memory of each K20 board is almost entirely filled. Indeed the storage of the electromagnetic field on one subdomain requires approximately 3.4 GB.

We observe in Table 1.5 that the asynchronous implementation is rather efficient and that the communications are well overlapped by the GPU com-

| | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs |
|---|---|---|---|---|
| TFLOP/s | 1.01 | 1.84 | 3.53 | 5.07 |
| Speedup | 1 | 1.83 | 3.53 | 5.01 |

**Table 1.4** Weak scaling of the synchronous MPI/OpenCL implementation

| | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs |
|---|---|---|---|---|
| TFLOP/s | 1.01 | 1.96 | 3.78 | 7.34 |
| Speedup | 1 | 1.94 | 3.74 | 7.26 |

**Table 1.5** Weak scaling of the asynchronous MPI/OpenCL implementation

putations. In addition, we observe that with CLAC we attain approximately 30% of the roofline limit. This result is not too bad, because CLAC handles unstructured meshes and some non-coalescent memory access are unavoidable.

### *1.3.5 Numerical results*

For finishing this paper, we would like to present numerical results that we have obtained from a real-world application. The objective is to compute the reflection of an electromagnetic plane wave with Gaussian profile over an entire aircraft. The aircraft geometry is displayed in Figure 1.10. The mesh is made of $3,337,875$ hexahedrons. We used an order $D = 2$ approximation and 8 GPUs (NVIDIA K20). The interior and the exterior of the aircraft are meshed. In order to approximate the infinite exterior model, we use a Perfectly Matched Layers (PML) model [3]. The PML model is an extension of the Maxwell model. The possibility to use different models in different zones is here exploited for applying the PML model. In a PML zone, the Maxwell equations are coupled with a system of six ordinary differential equations. This coupling induces an additional cost reported in Table 1.6.

| Order | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 5 layers (%) | 7.14 | 4.29 | 15.9 | 16.5 | 15.0 |
| 10 layers (%) | 7.95 | 6.49 | 19.0 | 20.6 | 18.1 |

**Table 1.6** Additional cost for 5 and 10 PML expressed in percentage of the total computation time.

The current density on the aircraft is given in Figure 1.11 at a chosen time.

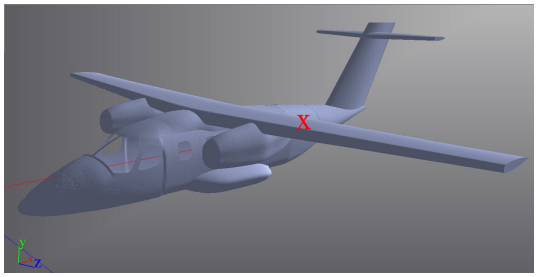

**Fig. 1.10** Aircraft. Only the mesh skin is represented.

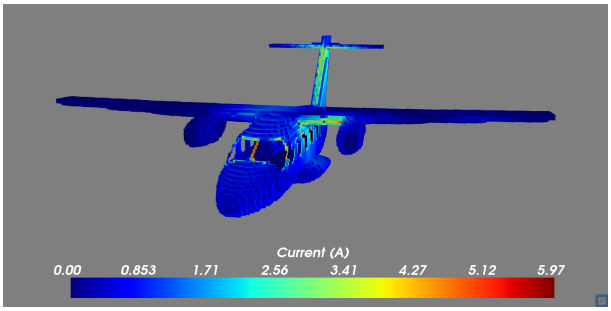


**Fig. 1.11** Current density on the aircraft skin.

## 1.4 Conclusions

In this work we have reviewed several methods for solving hyperbolic conservation laws. Such models are very useful in many fields of physics or engineering. We have presented a finite volume OpenCL/MPI implementation. We have seen that coalescent memory access is essential for obtaining good efficiency. The synchronous MPI communication does not allow an optimal scaling with several GPUs. However the MPI extension allows addressing computations that would not fit into a single accelerator.

We have then presented a more sophisticated approach: the Discontinuous Galerkin method on unstructured hexahedral meshes. We have also written an OpenCL/MPI implementation of the method. Despite the unstructured mesh and some non-coalescent memory accesses, we reach 30% of the peak performance.

In future works we intend to change the description of the mesh geometry in order to minimize the memory access: we can for instance share a higher order geometrical transformation $\tau$ between several cells. We also plan to implement a local-time stepping algorithm in order to be able to deal with locally refined meshes. Finally, we would like to describe the task graph in a more abstract manner in order to distribute the computation more effectively

on the available resources. An interesting tool for performing such distribution could be for instance the StarPU environment [2].

## 1.5 Acknowledgments

This work has benefited from several supports: from the French Defense Agency DGA, from the Labex ANR-11-LABX-0055-IRMIA and from the AxesSim company. We also thank Vincent Loechner for his helpful advice regarding the optimization of the OpenMP code.

## References

1. D. Aubert. Numerical cosmology powered by GPUs. *Proceedings of the International Astronomical Union*, 6(S270):397–400, 2010.
2. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.
3. J.-P. Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of computational physics*, 114(2):185–200, 1994.
4. T. Cabel, J. Charles, and S. Lanteri. Multi-GPU acceleration of a DGTD method for modeling human exposure to electromagnetic waves, 2011.
5. G. Cohen, X. Ferrieres, and S. Pernet. A spatial high-order hexahedral discontinuous Galerkin method to solve Maxwell's equations in time domain. *Journal of Computational Physics*, 217(2):340–363, 2006.
6. P. Helluy and J. Jung. Interpolated pressure laws in two-fluid simulations and hyperbolicity. In *Finite Volumes for Complex Applications VII-Methods and Theoretical Aspects*, pages 37–53. Springer International Publishing, 2014.
7. P. Helluy and J. Jung. Two-fluid compressible simulations on GPU cluster. *ESAIM: Proceedings and Surveys*, 45:349–358, 2014.
8. J. S. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, volume 54. Springer Science & Business Media, 2007.
9. A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *J. Comput. Phys.*, 228(21):7863–7882, 2009.
10. A. Kloeckner. Hedge: Hybrid and Easy Discontinuous Galerkin Environment `http://mathema.tician.de/software/hedge/`, 2010.
11. M. Massaro, P. Helluy, and V. Loechner. Numerical simulation for the MHD system in 2D using OpenCL. *ESAIM: Proceedings and Surveys*, 45:485–492, 2014.
12. D. Michéa and D. Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophysical Journal International*, 182(1):389–402, 2010.
13. G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in CUDA. *Nvidia CUDA SDK Application Note*, 2009.
14. J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance gaps between OpenMP and OpenCL for multi-core CPUs. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 116–125. IEEE, 2012.
15. S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.