Optimization of a discontinuous finite element solver with OpenCL and StarPU

Philippe Helluy, Laura Mendoza, Bruno Weber

OPTIMIZATION OF A DISCONTINUOUS FINITE ELEMENT SOLVER WITH OPENCL AND STARPU

PHILIPPE HELLUY, LAURA MENDOZA, BRUNO WEBER

Abstract. `schnaps` is a finite element solver designed to simulate various physical phenomena. It is designed to run on hybrid computers made of several CPUs and GPUs. In order to address the hybrid architectures we rely on the StarPU runtime. StarPU allows to optimize in an incremental way a sequential algorithm in order to migrate to multicore parallelism and then to hybrid computing with OpenCL accelerators. StarPU proposes several task scheduling strategies in order to efficiently exploit the available computing power. We present the design of `schnaps` and the performance gain that we have obtained in electromagnetic simulations thanks to OpenCL codelets.

1. Introduction

The development of engineering simulation software is difficult. On the one hand, the user is interested in handling more and more complex physical phenomena, in devices with arbitrary geometries. These constraints require to adopt a generic and abstract software engineering approach in order to ensure the generality of the code. On the other hand, the user also wants to harness the full computational power of his hybrid computer. This requirement generally imposes to use low level hardware optimization hacks that are not always compatible with a generic and elegant approach. In addition, as the hardware evolves, optimizations of yesterday are not necessarily relevant for the devices of tomorrow...

OpenCL is a nice software environment for optimization purposes. It presents the good balance between an abstract view of the computing devices and some important hardware aspects, such as local memory for accelerating data transfers or subgroups for minimizing synchronization barriers. However depending on architecture, optimizations written for one accelerator may be completely irrelevant for another one. This is especially true with local memory optimizations. For instance, cache prefetching is generally efficient for discrete GPUs while inefficient for IGPs.

In this paper, we present our practical approach to this issue, with the help of the StarPU runtime system. StarPU is developed at Inria Bordeaux since 2006[1]. StarPU is a runtime C library based on the dataflow paradigm. The programmer has to split the computation workload into abstract computational tasks. Each task processes data buffers that can be in `read` (R), `write` (W) or `read/write` (RW) mode.

The tasks are then conveniently implemented into C *codelets*. It is possible (and recommended) to write several implementations of the same task into several codelets. For instance, one can imagine to write an unoptimized codelet for validation purposes and one or several optimized OpenCL codelets. The user then submits his tasks in a sequential way to StarPU. At runtime, StarPU is able

This work has obtained support from the ANR EXAMAG project and the BPI France HOROCH project.

[1]We are not involved in the creation and development of StarPU, but only users.

to construct a task graph based on buffer dependencies. The tasks are submitted to the available accelerators, in parallel if the dependencies allow it. In addition, StarPU automatically handles data transfers between the accelerators. It is also able to measure the efficiency of the different codelets' implementations in order to choose the best, according to the scheduling strategy.

With StarPU, implementing a complex dataflow of OpenCL kernels becomes easier. Indeed, the programmer does not have to handle the kernel dependencies with OpenCL events. In addition, it is possible to first write a well validated pure C version of the software, with only C codelets. Then, one can enrich the tasks with OpenCL codelets, which allows an incremental optimization of the code. At each stage, StarPU should be able to use the available codelets in an efficient way.

In this paper, we describe how we applied the StarPU philosophy in order to optimize a discontinuous finite element solver for conservation laws. The outlines are as follows: first we will present in its main lines the Discontinuous Galerkin (DG) scheme that is used in the solver. Then, after a short presentation of StarPU, we will explain how we have integrated the OpenCL optimizations into the DG solver. Finally, we will present some numerical experiments.

2. Discontinuous Galerkin method

The Discontinuous Galerkin (DG) method is a general finite element method for approximating systems of conservation laws of the form

$$\partial_t \mathbf{w} + \sum_{k=1}^{D} \partial_k \mathbf{f}^k(\mathbf{w}) = 0.$$

The unknown is the vector of conservative variables $\mathbf{w}(\mathbf{x}, t) \in \mathbb{R}^m$ depending on the space variable $\mathbf{x} = (x^1, \ldots, x^D) \in \mathbb{R}^D$ and on time t. In this paper, the space dimension is $D = 3$. We adopt the notations ∂_t for the partial derivative with respect to t and ∂_k for the partial derivative with respect to x^k. Let $\mathbf{n} = (n_1, \ldots, n_D) \in \mathbb{R}^D$ be a spatial direction, the flux in direction $\mathbf{n}$ is defined by

$$\mathbf{f}(\mathbf{w}, \mathbf{n}) = \sum_{k=1}^{D} n_k \mathbf{f}^k(\mathbf{w}).$$

For instance, in this work we consider the numerical simulation of an electromagnetic wave. In this particular case, the conservative variables are

$$\mathbf{w} = (\mathbf{E}^{\mathrm{T}}, \mathbf{H}^{\mathrm{T}}, \lambda, \mu)^{\mathrm{T}} \in \mathbb{R}^m, \quad m = 8.$$

where $\mathbf{E} \in \mathbb{R}^3$ is the electric field, $\mathbf{H} \in \mathbb{R}^3$ is the magnetic field and λ, μ are divergence cleaning potentials (Munz et al. [2000]). The flux is given by

$$\mathbf{f}(\mathbf{w}, \mathbf{n}) = \begin{pmatrix} -\mathbf{n} \times \mathbf{H} + \lambda \mathbf{n} \\ \mathbf{n} \times \mathbf{E} + \mu \mathbf{n} \\ c\mathbf{n} \cdot \mathbf{E} \\ c\mathbf{n} \cdot \mathbf{H} \end{pmatrix}$$

and $c > 1$ is the divergence cleaning parameter.

We consider a mesh $\mathcal{M}$ of Ω made of open sets, called *cells*, $\mathcal{M} = \{L_i,\ i = 1 \ldots N_c\}$. In the most general setting, the cells satisfy

(1) $L_i \cap L_j = \emptyset$, if $i \neq j$;
(2) $\overline{\cup_i L_i} = \overline{\Omega}$.

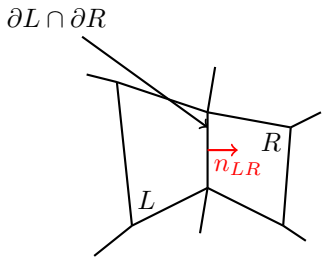

FIGURE 2.1. Convention for the L and R cells orientation.

In each cell $L \in \mathcal{M}$, we consider a basis of functions $(\varphi_{L,i}(\mathbf{x}))_{i=0\dots N_d-1}$ constructed from polynomials of order d. We denote by h the maximal diameter of the cells. With an abuse of notation we still denote by $\mathbf{w}$ the approximation of $\mathbf{w}$, defined by

$$\mathbf{w}(\mathbf{x},t) = \sum_{j=0}^{N_d-1} \mathbf{w}_{L,j}(t)\varphi_{L,j}(\mathbf{x}), \quad \mathbf{x} \in L.$$

The DG formulation then reads: find $\mathbf{w}_{L,j}$ such that for all cell L and all test function $\varphi_{L,i}$

$$\int_L \partial_t \mathbf{w}_L \varphi_{L,i} - \int_L \mathbf{f}(\mathbf{w}_L, \boldsymbol{\nabla}\varphi_{L,i}) + \int_{\partial L} \mathbf{f}(\mathbf{w}_L, \mathbf{w}_R, \mathbf{n})\varphi_{L,i} = 0. \tag{2.1}$$

In this formula (see Figure 2.1):

- R denotes the neighboring cell to L along its boundary $\partial L \cap \partial R$, or the exterior of Ω on $\partial L \cap \partial\Omega$.
- $\mathbf{n} = \mathbf{n}_{LR}$ is the unit normal vector on ∂L oriented from L to R.
- $\mathbf{w}_R$ denotes the value of $\mathbf{w}$ in the neighboring cell R on $\partial L \cap \partial R$.
- If L is a boundary cell, one may have to use the boundary values instead: $\mathbf{w}_R = \mathbf{w}_b$ on $\partial L \cap \partial\Omega$.
- $\mathbf{f}(\mathbf{w}_L, \mathbf{w}_R, \mathbf{n})$ is the standard upwind numerical flux encountered in many finite volume or DG methods.

In our application, we consider hexahedral cells. We have a reference cell

$$\hat{L} =]-1,1[^D$$

and a smooth transformation $\mathbf{x} = \boldsymbol{\tau}_L(\hat{\mathbf{x}})$, $\hat{\mathbf{x}} \in \hat{L}$, that maps $\hat{L}$ on L

$$\boldsymbol{\tau}_L(\hat{L}) = L.$$

We assume that $\boldsymbol{\tau}_L$ is invertible and we denote by $\boldsymbol{\tau}'_L$ its (invertible) Jacobian matrix. We also assume that $\boldsymbol{\tau}_L$ is a direct transformation

$$\det \boldsymbol{\tau}'_L > 0.$$

In our implementation, $\boldsymbol{\tau}_L$ is a quadratic map based on hexahedral curved "H20" finite elements with 20 nodes. The mesh of H20 finite elements is generated by `gmsh` (Geuzaine and Remacle [2009]).

On the reference cell, we consider the Gauss-Lobatto points $(\hat{\mathbf{x}}_i)_{i=0\dots N_d-1}$, $N_d = (d+1)^D$ and associated weights $(\omega_i)_{i=0\dots N_d-1}$. They are obtained by tensor products of the $(d+1)$ one-dimensional

Gauss-Lobatto (GL) points on $]-1,1[$. The reference GL points and weights are then mapped to the physical GL points of cell L by

$$\mathbf{x}_{L,i} = \boldsymbol{\tau}_L(\hat{\mathbf{x}}_i), \quad \omega_{L,i} = \omega_i \det \boldsymbol{\tau}'_L(\hat{\mathbf{x}}_i) > 0. \tag{2.2}$$

In addition, the six faces of the reference hexahedral cell are denoted by F_ϵ, $\epsilon = 1 \dots 6$ and the corresponding outward normal vectors are denoted by $\hat{\mathbf{n}}_\epsilon$. One advantage of choosing the GL points is that the cells and the faces share the same quadrature points. We use the following notations to define the face quadrature weights:

- if a GL point $\hat{\mathbf{x}}_i \in F_\epsilon$, we denote by μ_i^ϵ the corresponding quadrature weight on face $\mathbf{w}_\epsilon$;
- we also use the convention that $\mu_i^\epsilon = 0$ if $\hat{\mathbf{x}}_i$ does not belong to face F_ϵ.

Let us remark that a given GL point $\hat{\mathbf{x}}_i$ can belong to several faces when it is on an edge or in a corner of $\hat{L}$. Because of symmetry, we observe that if $\mu_i^\epsilon \neq 0$, then the weight μ_i^ϵ does not depend on ϵ.

We then consider basis functions $\hat{\varphi}_i$ on the reference cell: they are the Lagrange polynomials associated to the Gauss-Lobatto points and thus satisfy the interpolation property

$$\hat{\varphi}_i(\hat{\mathbf{x}}_j) = \delta_{ij}.$$

The basis functions on cell L are then defined according to the formula

$$\varphi_{L,i}(\mathbf{x}) = \hat{\varphi}_i(\boldsymbol{\tau}_L^{-1}(\mathbf{x})).$$

In this way, they also satisfy the interpolation property

$$\varphi_{L,i}(\mathbf{x}_{L,j}) = \delta_{ij}. \tag{2.3}$$

In this paper, we only consider conformal meshes: the GL points on cell L are supposed to match the GL points of cell R on their common face. Dealing with non-matching cells is the object of a forthcoming work. Let L and R be two neighboring cells. Let $\mathbf{x}_{L,j}$ be a GL point in cell L that is also on the common face between L and R. In the case of conformal meshes, it is possible to define the index j' such that

$$\mathbf{x}_{L,j} = \mathbf{x}_{R,j'}.$$

Applying a numerical integration to (2.1), using (2.2) and the interpolation property (2.3), we finally obtain

$$\partial_t \mathbf{w}_{L,i}\omega_{L,i} - \sum_{j=0}^{N_d-1} \mathbf{f}(\mathbf{w}_{L,j}, \boldsymbol{\nabla}\varphi_{L,i}(\mathbf{x}_{L,j}))\omega_{L,j} + \sum_{\epsilon=1}^{6} \mu_i^\epsilon \mathbf{f}(\mathbf{w}_{L,i}, \mathbf{w}_{R,i'}, \mathbf{n}_\epsilon(\mathbf{x}_{L,i})) = 0. \tag{2.4}$$

We have to detail how the gradients and normal vectors are computed in the above formula. Let $\mathbf{A}$ be a square matrix. We recall that the cofactor matrix of $\mathbf{A}$ is defined by

$$\mathrm{co}(\mathbf{A}) = \det(\mathbf{A}) \left(\mathbf{A}^{-1}\right)^{\mathrm{T}}. \tag{2.5}$$

The gradient of the basis function is computed from the gradients on the reference cell using (2.5)

$$\boldsymbol{\nabla}\varphi_{L,i}(\mathbf{x}_{L,j}) = \frac{1}{\det \boldsymbol{\tau}'_L(\hat{\mathbf{x}}_j)} \mathrm{co}(\boldsymbol{\tau}'_L(\hat{\mathbf{x}}_j)) \hat{\nabla}\hat{\varphi}_i(\hat{\mathbf{x}}_j).$$

In the same way, the scaled normal vectors $\mathbf{n}_\epsilon$ on the faces are computed by the formula

$$\mathbf{n}_\epsilon(\mathbf{x}_{L,i}) = \mathrm{co}(\boldsymbol{\tau}'_L(\hat{\mathbf{x}}_i))\hat{\mathbf{n}}_\epsilon.$$

We introduce the following notation for the cofactor matrix

$$\mathbf{c}_{L,i} = \mathrm{co}(\boldsymbol{\tau}'_L(\hat{\mathbf{x}}_i)).$$

The DG scheme then reads

$$\partial_t \mathbf{w}_{L,i} - \frac{1}{\omega_{L,i}} \sum_{j=0}^{N_d-1} \mathbf{f}(\mathbf{w}_{L,j}, \mathbf{c}_{L,j}\hat{\nabla}\hat{\varphi}_i(\hat{\mathbf{x}}_j))\omega_j + \frac{1}{\omega_{L,i}} \sum_{\epsilon=1}^{6} \mu_i^\epsilon \mathbf{f}\left(\mathbf{w}_{L,i}, \mathbf{w}_{R,i'}, \mathbf{c}_{L,i}\hat{\mathbf{n}}_\epsilon\right) = 0. \tag{2.6}$$

In formula (2.6) we identify volume terms that are associated to Gauss-Lobatto points in the volume

$$V_{i,j} = \mathbf{f}(\mathbf{w}_{L,j}, \mathbf{c}_{L,j}\hat{\nabla}\hat{\varphi}_i(\hat{\mathbf{x}}_j))\omega_j \tag{2.7}$$

and surface terms that are associated to cell boundaries

$$S_{i,i'} = \mu_i^\epsilon \mathbf{f}\left(\mathbf{w}_{L,i}, \mathbf{w}_{R,i'}, \mathbf{c}_{L,i}\hat{\mathbf{n}}_\epsilon\right). \tag{2.8}$$

Then, once these terms have been accumulated, one has to apply the inverse of the mass matrix. Here, this matrix is diagonal and it corresponds to a simple multiplication of $\partial_t \mathbf{w}_{L,i}$ by

$$\frac{1}{\omega_{L,i}}. \tag{2.9}$$

Generally, i and i' are associated to unknowns of internal cells, *i.e.* cells that do not touch the boundaries. On boundary GL points, the value of $\mathbf{w}_{R,i'}$ is given by the boundary condition

$$\mathbf{w}_{R,i'} = \mathbf{w}_b(\mathbf{x}_{L,i}, t), \quad \mathbf{x}_{L,i} = \mathbf{x}_{R,i'}.$$

For practical reasons, it can be interesting to also consider $\mathbf{w}_{R,i'}$ as an artificial unknown in a fictitious cell. The fictitious unknown is then a solution of the differential equation

$$\partial_t \mathbf{w}_{R,i'} = \partial_t \mathbf{w}_b(\mathbf{x}_{L,i}, \cdot). \tag{2.10}$$

In the end, if we put all the unknowns in a large vector $\mathbf{W}(t)$, (2.6) and (2.10) read as a large system of coupled differential equations

$$\partial_t \mathbf{W} = \mathbf{F}(\mathbf{W}). \tag{2.11}$$

This set of differential equations is then numerically solved by a Runge-Kutta numerical method. In practice, we use a second order Runge-Kutta method (RK2).

3. Data-based parallelism and StarPU

StarPU is a runtime system library developed at Inria Bordeaux (Augonnet et al. [2011, 2012]). It relies on the data-based parallelism paradigm. The user has first to split its whole problem into elementary computational tasks. The elementary tasks are then implemented into *codelets*, which are simple C functions. The same task can be implemented differently into several codelets. This allows the user to harness special accelerators, such as vectorial CPU cores or OpenCL devices, for example. In the StarPU terminology these devices are called *workers*. If a codelet contains OpenCL kernel submissions, special utilities are available in order to map the StarPU buffers to OpenCL buffers.

For each task, the user also has to describe precisely what are the input data, in `read` mode, and the output data, in `write` or `read/write` mode. The user then submits the task in a sequential way to the StarPU system. StarPU is able to construct at runtime a task graph from the data dependencies. The task graph is analyzed and the tasks are scheduled automatically to the available workers (CPU cores, GPUs, *etc.*). If possible, they are executed in parallel into concurrent threads.

The data transfer tasks between the threads are automatically generated and managed by StarPU. OpenCL data transfers are also managed by StarPU.

When a StarPU program is executed, it is possible to choose among several schedulers. The simplest `eager` scheduler adopts a very simple strategy: the tasks are executed in the order of submission by the free workers, without optimization. More sophisticated schedulers, such as the `dmda` or `dmdar` scheduler, are able to measure the efficiency of the different codelets and the data transfer times, in order to apply a more efficient allocation of tasks.

Recently a new data access mode has been added to StarPU: the `commute` mode. In a task, a buffer of data can now be accessed in `commute` mode, in addition to the `write` or `read/write` modes. A `commute` access tells to StarPU that the execution of the corresponding task may be executed before or after other tasks containing commutative access. This allows StarPU to perform additional optimizations.

There also exists a MPI version of StarPU. In the MPI version, the user has to decide an initial distribution of data among the MPI nodes. Then the tasks are submitted as usual (using the function starpu_mpi_insert_task instead of starpu_insert_task). Required MPI communications are automatically generated by StarPU. For the moment, this approach does not guarantee a good load balancing. It is the responsibility of the user to migrate data from one MPI node to another to improve the load balancing, if necessary. The MPI version of StarPU is not tested in this work.

3.1. **Macrocell approach.** StarPU is quite efficient, but there is an unavoidable overhead due to the task submissions and to the on-the-fly construction and analysis of the task graph. Therefore it is important to ensure that the computational tasks are not too small, in which case the overhead would not be amortized, or not too big, in which case some workers would be idle. To achieve the right balance, we decided not to apply the DG algorithm at the cell level but to groups of cells instead.

The implementation of the DG scheme has been made into the `schnaps` software[2] which is a C99 software dedicated to the numerical simulation of conservation laws. In `schnaps`, we first construct a *macromesh* of the computational domain. Then each *macrocell* of the macromesh is split into *subcells*. See Figure 3.1. We also arrange the subcells into a regular sub-mesh of the macrocells. In this way, it is possible to apply additional optimizations. For instance, the subcells L of a same macrocell $\mathcal{L}$ share the same geometrical transformation $\boldsymbol{\tau}_L$, which saves memory access.

In `schnaps` we also defined an *interface* structure in order to manage data communications between the macrocells. An interface contains a copy of the data at the Gauss-Lobatto points that are common to the two neighboring macrocells.

To solve one time step of the DG method, here are the most important tasks:

(1) Interface extraction: this task simply extracts the values of $\mathbf{w}$ coming from the neighboring macrocells to the interface. In this task, the macrocell buffers are accessed in `read` mode and the interface data in `write` mode.
(2) Interface fluxes: this task only computes the numerical fluxes (2.8) that are located at the Gauss-Lobatto points on the interface. The fluxes are then applied to the neighboring macrocells. In this task, the interface data are accessed in `read` mode and the macrocell data in `read/write` mode. For a better efficiency, we also assume a `commute` access to the

[2] `http://schnaps.gforge.inria.fr/`

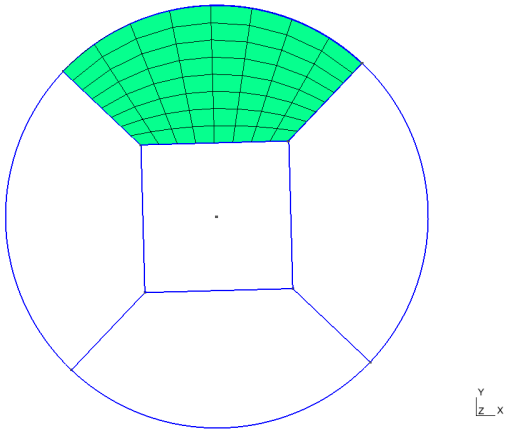

FIGURE 3.1. Macrocell approach: an example of a mesh made of five macrocells. Each macrocell is then split into several subcells. Only the subcells of the top macrocell are represented here (in green).

Algorithm 1 DG algorithm

for all interface:
Extract interface (copy the data from the neighboring macrocells to the interface)
If the interface is a boundary interface, compute the boundary fluxes (2.8) and apply them to the neighboring macrocell
Else the interface is an internal one. Compute the numerical fluxes (2.8) and apply them to the two neighboring macrocells
End for
Then, for each macrocell:
Compute and apply the numerical fluxes (2.8) between the subcells
Compute the volume terms (2.7) inside the subcells
Apply the inverse of the mass matrix (2.9) inside the subcells
End for

macrocell data. In this way the interface fluxes can be assembled in any order, which can help the StarPU scheduling.

(3) Interface boundary fluxes: this task computes the boundary data and only numerical fluxes (2.8) that are located at the boundary interface. The fluxes are applied to the neighboring macrocell. In this task, the interface data are accessed in `read` mode and the macrocell data in `read/write` and `commute` mode.
(4) Volume terms: this task applies the volumic terms (2.7) in a given macrocell. The macrocell data are accessed in `read/write/commute` mode.
(5) Subcell fluxes: this task applies the numerical subcell fluxes (2.8) that are internal to a given macrocell. The macrocell data are accessed in `read/write/commute` mode.

Additional simple tasks are needed to apply the Runge-Kutta algorithm. We do not describe them.

The general sequential DG algorithm is then given in Algorithm 1. Thanks to StarPU, this algorithm can be submitted in a sequential way. StarPU then uses the data dependency in order to distribute the tasks in parallel on the available workers.

In the next section we give more details about the implementation of the OpenCL codelets.

4. Hybrid C/OpenCL codelets

In order to attain better performance, we programmed an OpenCL version of the previously described codelets. As it is often the case, special care has to be given in order to improve the coalescence of memory access. The values of $\mathbf{w}$ at the Gauss-Lobatto points are stored into what we call a *field* structure. A field is attached to a macrocell. A given component of $\mathbf{w}$ in a field is located thanks to its subcell index `ic`, a Gauss-Lobatto index `ig` in the cell, and a component index `iw`. If the macrocell is cut into n_r subcells in each direction, if the polynomial order is d and for a system of m conservation laws, these indices have the following bounds

$$0 \leq \texttt{ic} < n_r^3, \quad 0 \leq \texttt{ig} < (d+1)^3, \quad 0 \leq \texttt{iw} < m.$$

The values of $\mathbf{w}$ in a given field are stored into a memory buffer `wbuf`. In order to test several memory arrangements, the index in the buffer is computed by a function `varindex(ic,ig,iw)` that we can easily change. For instance, we can consider the following formula, in which the electromagnetic components at a given Gauss-Lobatto point are grouped in memory

$$\texttt{varindex}(\texttt{ic},\texttt{ig},\texttt{iw}) = \texttt{iw}+m*(\texttt{ig}+(d+1)^3*\texttt{ic}), \tag{4.1}$$

or this formula

$$\texttt{varindex}(\texttt{ic},\texttt{ig},\texttt{iw}) = \texttt{ig}+(d+1)^3*(\texttt{iw}+m*\texttt{ic}), \tag{4.2}$$

in which the electromagnetic components are separated in memory. We have programmed an OpenCL codelet for each task described in Section 3.1. The most time consuming kernels are: (i) the kernel associated to the computations of the volume terms (2.7) inside the macrocells, *volume* kernel, and (ii) the kernel that computes the subcell fluxes, the *surface* kernel. Boundary and interface terms generally involve less computations.

A natural distribution of the workload is to associate one subcell to each OpenCL work-group and the computations at a Gauss point to a work-item. With this natural distribution, formula (4.2) ensures optimal coalescent memory access in the volume kernel, but the access is no more optimal in the surface kernel, because neighboring Gauss points on the subcell interfaces are not necessarily close in memory. Therefore, we prefer considering formula (4.1).

5. Numerical results

In this section, we present some practical experiments that we realized with `schnaps`. The first experiment deals with the efficiency of the StarPU C99 codelets on a multicore CPU. The second experiment deals with the efficiency of the OpenCL codelets. Then we test the code in a CPU/GPU configuration.

We approximate by the DG algorithm an exact plane wave solution of the Maxwell equations

$$\mathbf{E} = \frac{c}{r}\begin{pmatrix} -v \\ u \\ 0 \end{pmatrix}, \quad \mathbf{H} = \frac{c}{r}\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \quad \lambda = 0, \quad \mu = 0.$$

config.	CPU	# cores	mem. CPU	GPU	mem. GPU
"PC"	AMD FX-8320E 3.2 GHz	8	8 GB	NVIDIA GTX 1050 Ti	4 GB
"WS"	Intel Xeon E5-2609 1.7 GHz	2×8	64 GB	NVIDIA QUADRO P6000	24 GB

TABLE 1. Configurations of the computers used for the tests.

n_r	n_{cpu}	Time (s)	Ideal time	Efficiency
4	2	704	704	1
4	4	365	352	0.96
4	7	209	201	0.96
6	7	689	679	0.99
8	4	2902	2816	0.97
8	7	1650	1609	0.98

TABLE 2. Efficiency of the CPU codelets for the "PC" configuration with the `dmdar` scheduler.

with $c = -\cos(\pi/2(ux^1 + vx^2 - t))$ and $(u, v) = (\cos(\pi/4), \sin(\pi/4))$.

We consider the propagation of the previous electromagnetic plane wave into a torus-shaped domain Ω represented on Figure 5.1. In the DG solver, we consider only third order polynomials: $d = 3$. The domain is split into $M = 400$ macrocells. The macrocells are split into n_r subcells in each direction. The number of degrees of freedom is thus

$$n_{\text{d.o.f.}} = Mm(d+1)^3 n_r^3, \quad M = 400, \quad m = 8, \quad d = 3.$$

StarPU will have to take decisions on how to distribute efficiently the tasks on the available accelerators. We have performed our experiments on two different computers. The first configuration "PC" correspond to a standard desktop personnal computer. The second configuration "WS" is made of a more powerful two-CPU workstation The technical details are listed in Table 1.

5.1. **Multicore CPU tests.** In the first test, we only activate the C99 codelets, and thus the GPU is not activated. We also vary the number of computing CPU cores from 2 to 7 for the "PC" configuration and from 2 to 15 for the "WS" configuration. One CPU core is anyway reserved for the main StarPU thread.

We perform a fixed number of n_t iterations of the RK2 algorithm. We assume that the number of elementary computing operations increases as $O(n_r^3)$. Ideally, with infinitely fast operations at interfaces, and with n_a identical CPU cores, the computations time would behave like

$$T \sim C n_t \frac{n_r^3}{n_a} \tag{5.1}$$

where C is fixed constant, depending on the hardware. In our benchmark the *efficiency* is the ratio of the measured execution time over this ideal time. Normally it should be smaller than one (because of communications). If it is close to one, the algorithm is very efficient. In Table 2 and 3, we observe an excellent efficiency of the StarPU task distribution on the CPU cores even on a machine with two NUMA nodes.

n_r	n_{cpu}	Time (s)	Ideal time	Efficiency
4	2	763	763	1
4	4	382	381	0.99
4	8	196	191	0.97
4	15	110	102	0.92
8	8	1586	1526	0.96
8	15	828	814	0.98

TABLE 3. Efficiency of the CPU codelets for the "WS" configuration with the `dmdar` scheduler.

n_r	config.	t_{CPU} (s)	t_{GPU}(s)	$t_{\text{CPU}}/t_{\text{GPU}}$	$t_{\text{CPU+GPU}}$ (s)	$t_{\text{CPU}}/t_{\text{CPU+GPU}}$
4	PC	209	73	2.86	32	6.53
6	PC	689	86	8.01	64	10.77
8	PC	1650	171	9.65	130	12.69
4	WS	110	86	1.28	36	3.05
8	WS	828	88	9.41	77	10.75

TABLE 4. Efficiency of the CPU/GPU implementation.

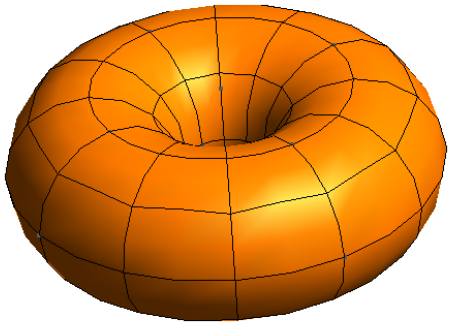

FIGURE 5.1. Computational domain Ω. Only the macrocells of the mesh are visible. The represented mesh contains 240 macrocells (poloidal refinement set to $n_{pol} = 3$ and toroidal refinement set to $n_{tor} = 3$. In the numerical experiments, we take $n_{pol} = 3$ and $n_{tor} = 5$, which leads to a mesh of 400 macrocells). Each macrocell is then cut regularly into the three directions.

5.2. **OpenCL codelets.** In this section, we compare the efficiency of the CPU/C99 and GPU/OpenCL codelets on the two configurations "PC" and "WS". We first perform a CPU-only computation with all the cores activated (7 for the "PC" platform and 15 for the "WS" platform) and a GPU-only computation. The results are given in Table 4. We observe that the OpenCL codelets are faster than the CPU codelets. The highest efficiency is achieved for the finer meshes ($n_r = 8$).

5.3. **Hybrid CPU/GPU computations.** Now that we are equipped with verified OpenCL codelets we can try to run the code with StarPU in order to see if it is able to distribute the tasks efficiently on the available accelerators.

One difficulty is to manage the data transfers efficiently between the accelerators. Several scheduling strategies are available in StarPU. With the `eager` scheduler, the tasks are distributed in the order of submission to the inactive device, without taking into account the cost of memory transfers. This strategy is obviously not optimal.

It is better to choose another scheduler, such as the `dmdar` scheduler. With the `dmdar` scheduler, the computational and memory transfer costs are evaluated in a preliminary benchmarking phase. Then an optimized scheduling is activated in order to better overlap computations and communications.

We perform computations by letting StarPU decide how to distribute the computations on the available CPU cores and GPU.

The results are given in the last two columns of Table 4. We observe that in all the situations, StarPU is able to get an additional gain from the CPU cores, even if the GPU codelets are faster.

6. Conclusion

We have proposed an optimized implementation of the Discontinuous Galerkin method on hybrid computer made of several GPUs and multicore CPUs. In order to manage the heterogeneous architecture easily and efficiently, we rely on OpenCL for the GPU computations and on the StarPU runtime for distributing the computational tasks on the available devices.

OpenCL programming becomes much easier because the task dependency is computed by StarPU. We only had to concentrate on the optimization of the individual OpenCL kernels and not on data distribution or memory transfers. We first tested the efficiency of our OpenCL kernels. We verified that the macrocell approach and cache prefetching strategy, while not optiaml, give good results.

In addition, with a good choice of scheduler, and for heavy computations, we have shown that StarPU is able to overlap computations and memory transfer in a quite efficient way. It is also able to use the available CPU codelets to achieve even higher acceleration.

References

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. Starpu-mpi: Task programming over clusters of machines enhanced with accelerators. In *European MPI Users' Group Meeting*, pages 298–299. Springer, 2012.

Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering*, 79(11):1309–1331, 2009.

C-D Munz, Pascal Omnes, Rudolf Schneider, Eric Sonnendrücker, and Ursula Voss. Divergence correction techniques for maxwell solvers based on a hyperbolic model. *Journal of Computational Physics*, 161(2):484–511, 2000.

Université de Strasbourg, Inria Tonus, AxesSim. philippe.helluy@unistra.fr