CLAC: an hybrid OpenCL/MPI Discontinuous Galerkin solver for generic conservation laws

P. Helluy (and J. Jung, V. Loechner, M. Massaro, T. Strub)

Inria Tonus & IRMA Université de Strasbourg

July 10, 2014

Outlines

1. Efficient GPU programing.
2. Discontinuous Galerkin (DG) solver design.
3. Some applications.

Conservation laws

Solution $W(x, y, t) \in \mathbb{R}^m$ of Maxwell/fluids/MHD/Vlasov equations. System of conservation laws.

$$\partial_t W + \partial_x F^x(W) + \partial_y F^y(W) = 0.$$

Approximation $W_{i,j}^n$ of $W(i\Delta x, j\Delta y, n\Delta t)$. Finite volume method + Strang splitting

$$\frac{W_{i,j}^* - W_{i,j}^n}{\Delta t} + \frac{F_{i+1/2,j}^{x,n} - F_{i-1/2,j}^{x,n}}{\Delta x} = 0,$$

$$\frac{W_{i,j}^{n+1} - W_{i,j}^*}{\Delta t} + \frac{F_{i,j+1/2}^{y,n} - F_{i,j-1/2}^{y,n}}{\Delta y} = 0.$$

Numerical flux: $F_{i+1/2,j}^{x,n} = F_{\text{num}}^x(W_{i,j}^n, W_{i+1,j}^n)$, $F_{i,j+1/2}^{y,n} = F_{\text{num}}^y(W_{i,j}^n, W_{i,j+1}^n)$.

CPU Implementation

On large grids ($> 1024 \times 1024$). We compare:

- a naive C implementation without optimization on a CPU single core;
- the same program, but compiled with optimizations;
- the same program with an additional optimization (tiling for optimizing data locality);
- the same program with OpenMP parallelization on a 16-core CPU.

Comparison

Implementation	Time	Speedup
Naive code	30 days	1
Naive code + optim. compil.	146 h	5
Naive code + optim. compil. + tiling	97 h	8
OpenMP version (16 cores)	6.2 h	116

OpenCL

OpenCL model: an accelerator device (GPU or CPU) is made of

- Global memory (typically 2 GB);
- Compute units (typically 30).

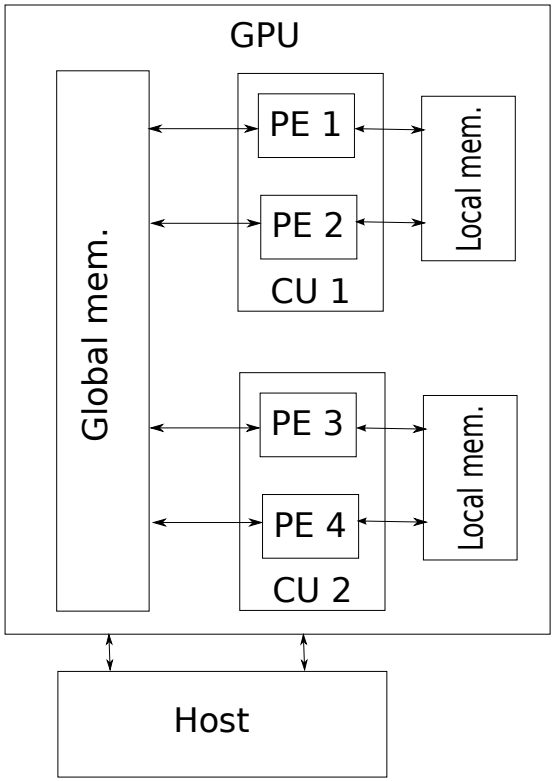
Each compute unit is made of

- Processing elements (typically 8);
- Local memory (typically 32 kb).

The same program (kernel) can be executed on all the processing elements at the same time.

- 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 two processing elements write at the same location at the same time, only one wins...
- The access to the global memory is slow while the access to the local memory is fast (generally...)

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



OpenCL

- OpenCL: "Open Computing Language". Library of C functions, called from the host, in order to drive the GPU + C-like language for the kernels. Available since 2009. Specification managed by the Khronos Group (OpenGL).
- Virtually, it allows to have as many compute units (work-groups) and processing elements (work-items) as needed.
- The threads are sent to the GPU thanks to a mechanism of command queues on the real compute units and processing elements. OpenCL manages events and a task graph for asynchronous out-of-order operations.
- Portable: the same program can run on a multicore CPU or a GPU. Drivers exist for: AMD CPU and GPU, Intel CPU and GPU, MIC, ARM, IBM Power7, StarPU, etc.

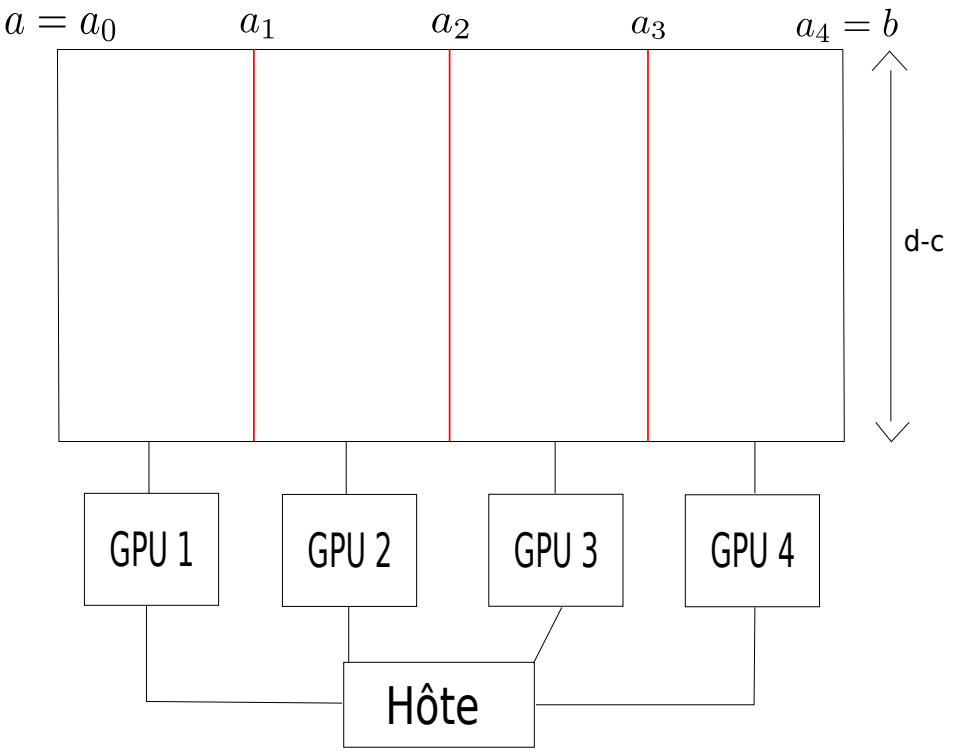
Implementation of the splitting scheme

We organize the data in a (x, y) grid and for each time step:

- we associate a work-item to each cell of the grid and a work-group to each row.
- we compute the fluxes balance in the x-direction for each cell of each row of the grid.
- we transpose the grid (exchange x and y) with an optimized memory transfer algorithm [5].
- we compute the fluxes balance in the y-direction for each row of the transposed grid. Memory access are optimal.
- we transpose again the grid.

OpenCL + MPI

We apply a subdomain decomposition for multi-GPU computing



Comparison

Implementation	Time	Speedup
Naive code	30 days	1
Naive code + optim. compil.	146 h	5
Naive code + optim. compil. + tiling	97 h	8
OpenMP (CPU Intel 8x2 cores)	6.2 h	116
OpenCL (CPU Intel 6x2 cores)	18 h	40
OpenCL (NVidia Tesla K20)	20 min	2160
OpenCL (AMD Radeon HD 7970)	16 min	2650
OpenCL + MPI (4 x AMD Radeon HD 7970)	5 min	7848

Essential: no test, optimized transposition (10x faster than the naive memory access)

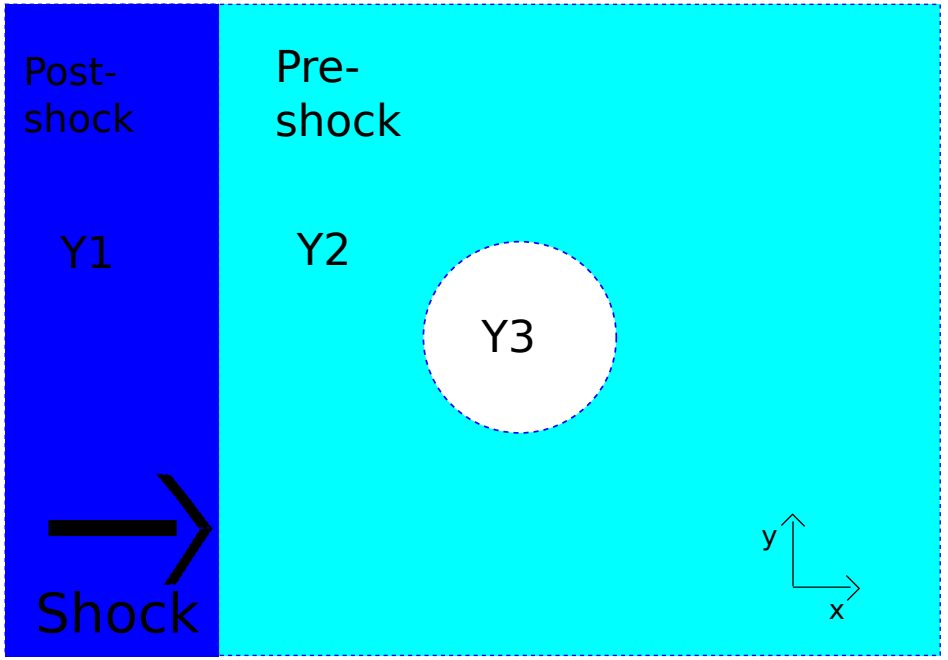
Partial conclusion

Well known conclusions

- GPU computing can be really faster,
- But: no test in OpenCL or CUDA kernels,
- Memory access are essentials.

Shock-bubble interaction

Two-phase flow conservation laws system. Density ρ, velocity (u, v), internal energy e, gas mass fraction ϕ.
$W = (\rho, \rho u, \rho v, \rho(e + u^2/2 + v^2/2), \rho\phi)$.

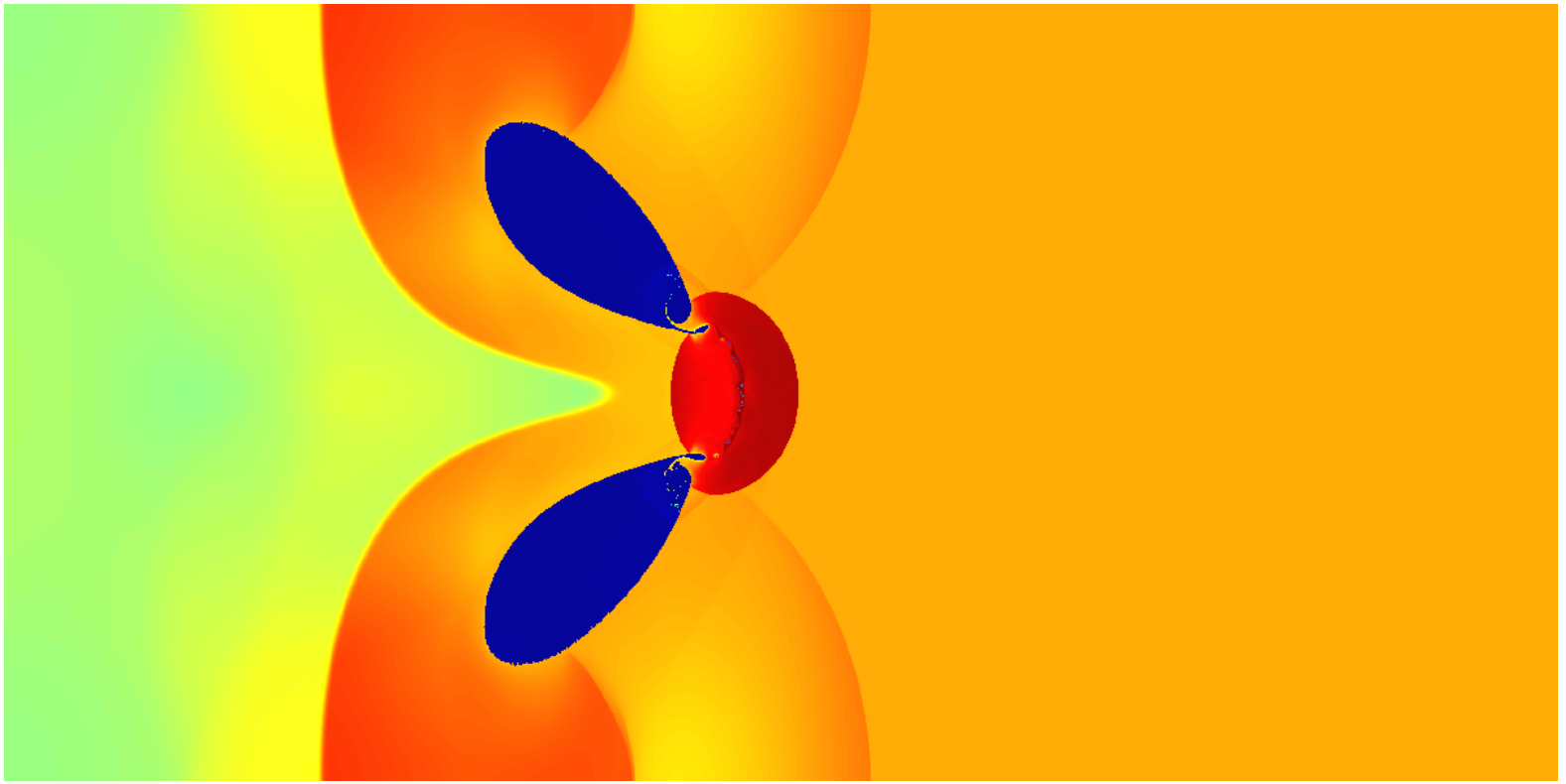


http://www.youtube.com/watch?v=c8hcqihJzbw

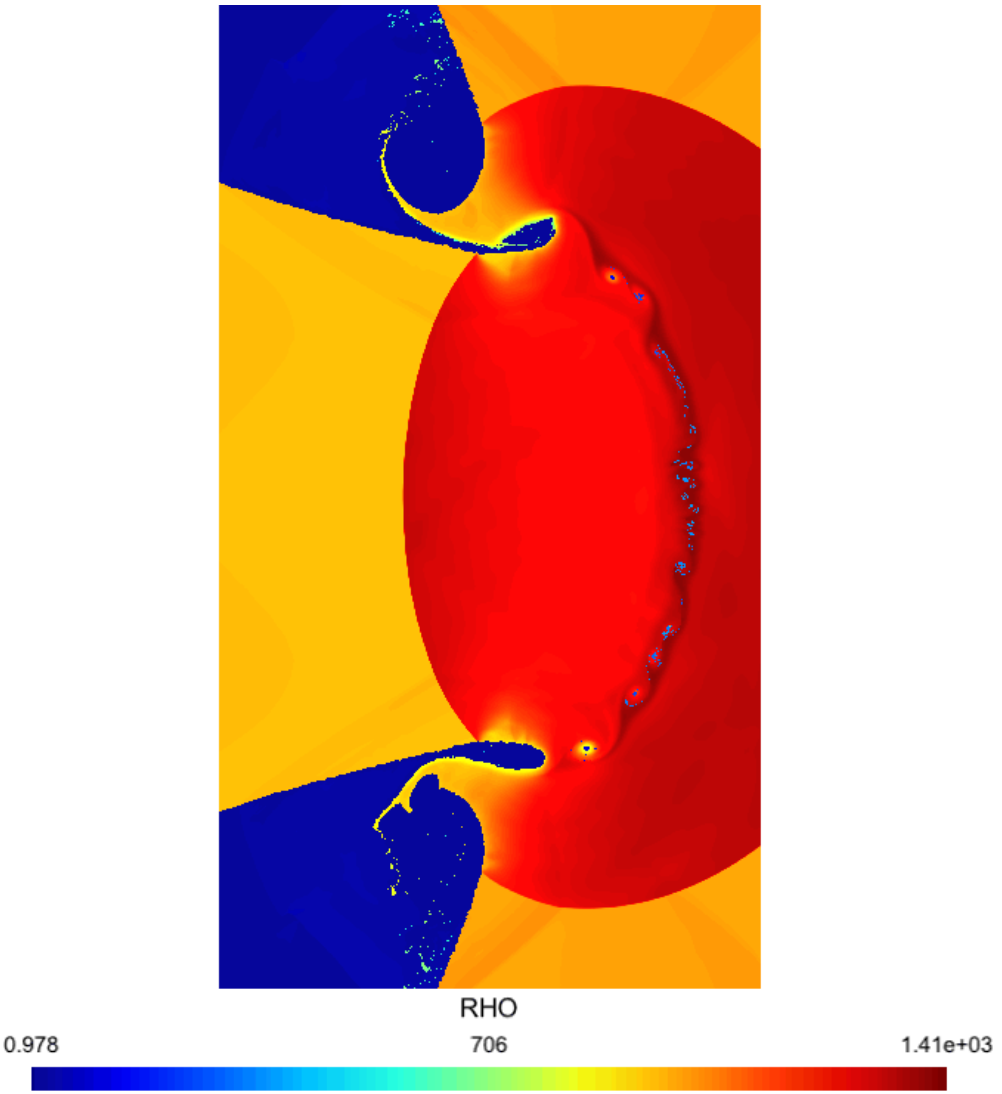
Numerical results

$t_{max} = 0.45$ ms
Grid: 40,000×20,000 (4 billions unknowns for each time step) [3]
GPU time: 30 h (10×NVIDIA K20)

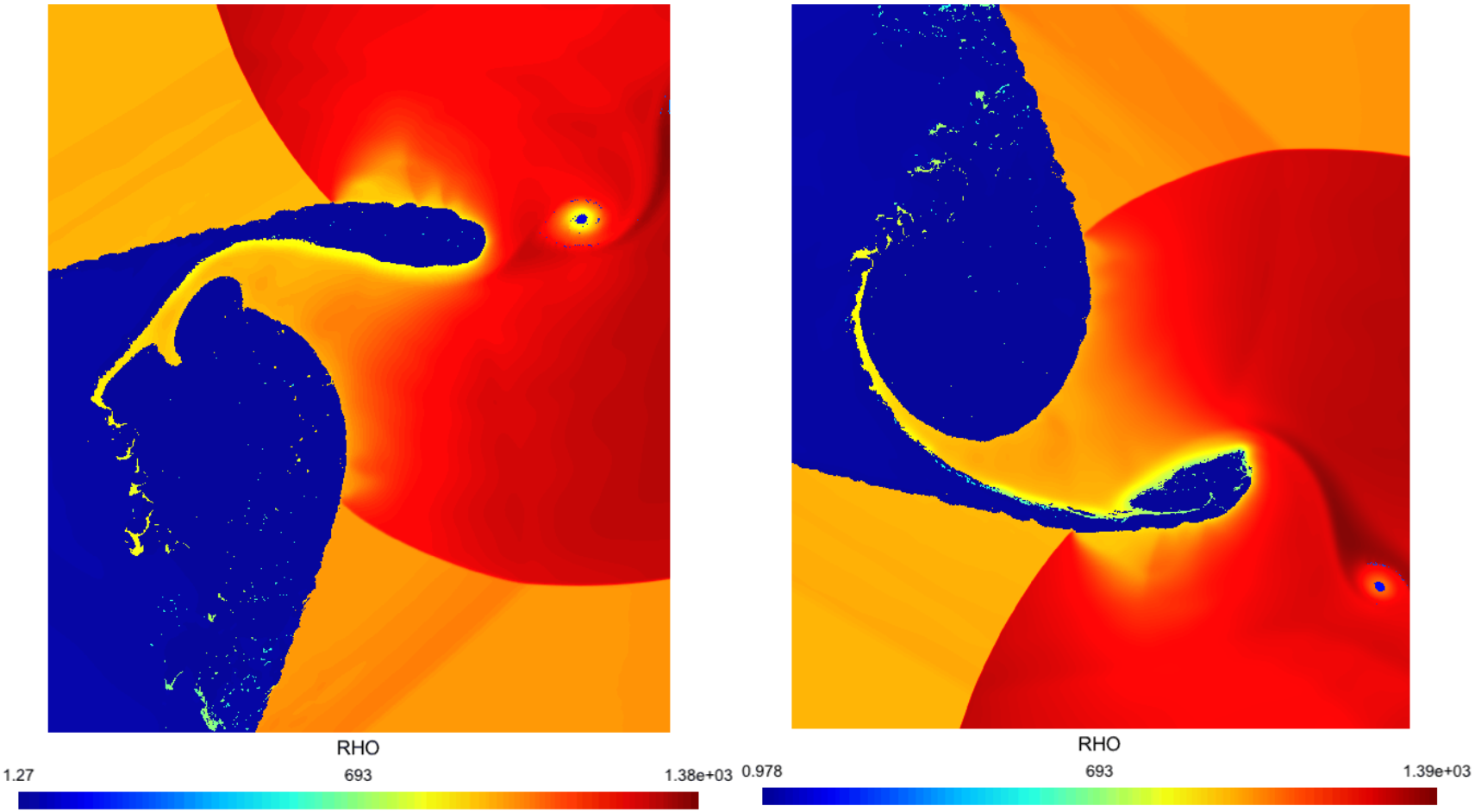
Density



Zoom 1

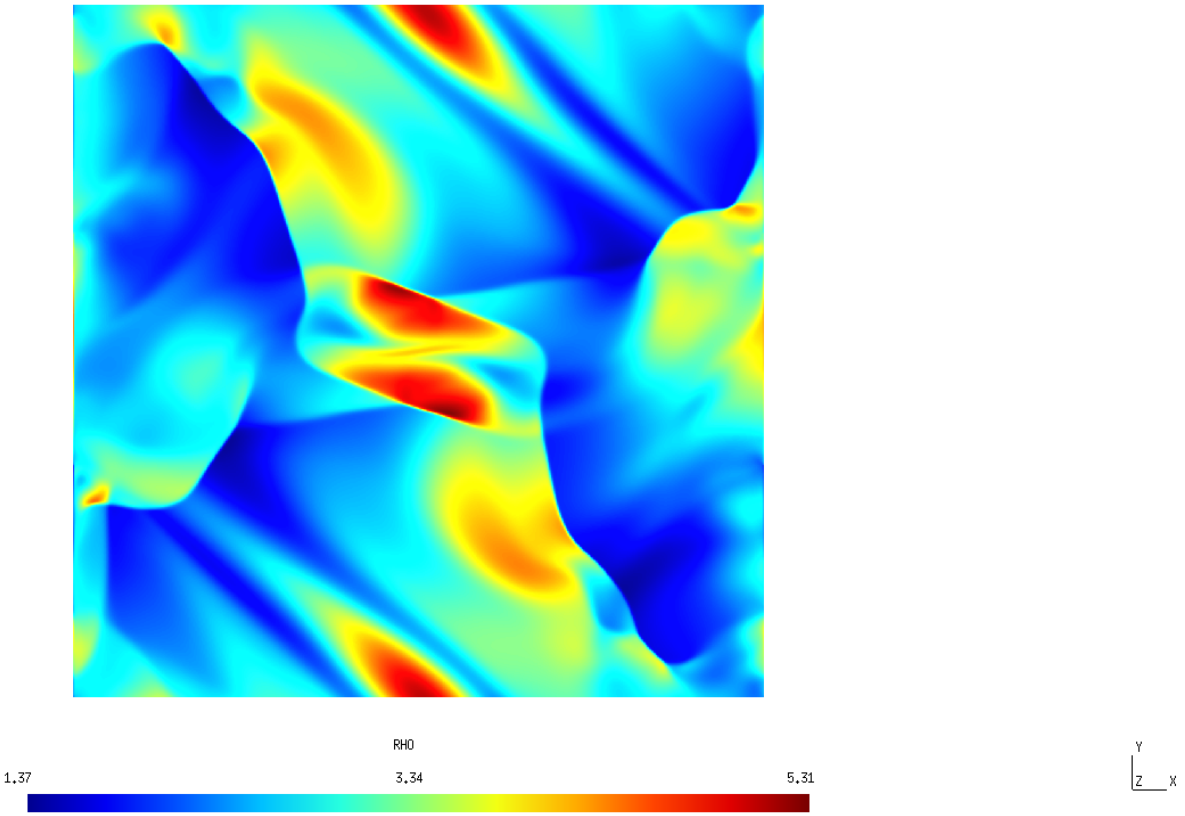


Zoom 2



MagnetoHydroDynamics (MHD)

$W = (\rho, \rho u, \rho v, \rho w, B_x, B_y, B_z, \rho(e + u^2/2 + v^2/2))$. Velocity (u, v, w), magnetic field (B_x, B_y, B_z).
Orszag-Tang vortex (grid size up to 15000×15000) [4].



DG solver

Generalization of the FV method, DG method in a 3D space, $X = (x, y, z)$. We consider a mesh of the computational domain. In a cell L of the mesh, the field is approximated by polynomial basis functions (sum on repeated indices)

$$W(X, t) = W_L^j(t)\varphi_j^L(X), \quad X \in L.$$

The numerical solution satisfies the DG approximation scheme

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

- R denotes the neighbor cells along ∂L.
- n_{LR} is the unit normal on ∂L oriented from L to R.
- $F(W_L, W_R, n)$ is the numerical flux (which satisfies $F(W, W, n) = F^x(W)n_x + F^y(W)n_y$).
- Time integration of a system of ordinary differential equations. Mass matrix $M_{i,j}^L = \int_L \varphi_i^L \varphi_j^L$

DG approach and GPU

advantages:

- varying order, mesh refinement.
- local stencil.
- high order $\Rightarrow$ high amount of local computations.
- many optimizations for hexahedrons meshes.
- MIMD/SIMD parallelism. Subdomains (MPI), elementary computations (OpenMP, CUDA, OpenCL) [1]

possible issues:

- memory access (unstructured mesh).
- branch tests in kernels (physical models, boundary conditions, etc.)
- MPI communications imply GPU$\leftrightarrow$Host memory transfers.

CLAC

CLAC means Conservation Laws Approximation on many Cores. It is a C++ library developed with AxesSim company in Strasbourg. It is actually used for actual electromagnetic simulations.

- "Reasonable" C++: a few templates, almost no inheritance.
- Google coding rules `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`
- Git on Inria GForge.
- Doxygen.
- scons.
- Boost: unit tests, graphs.

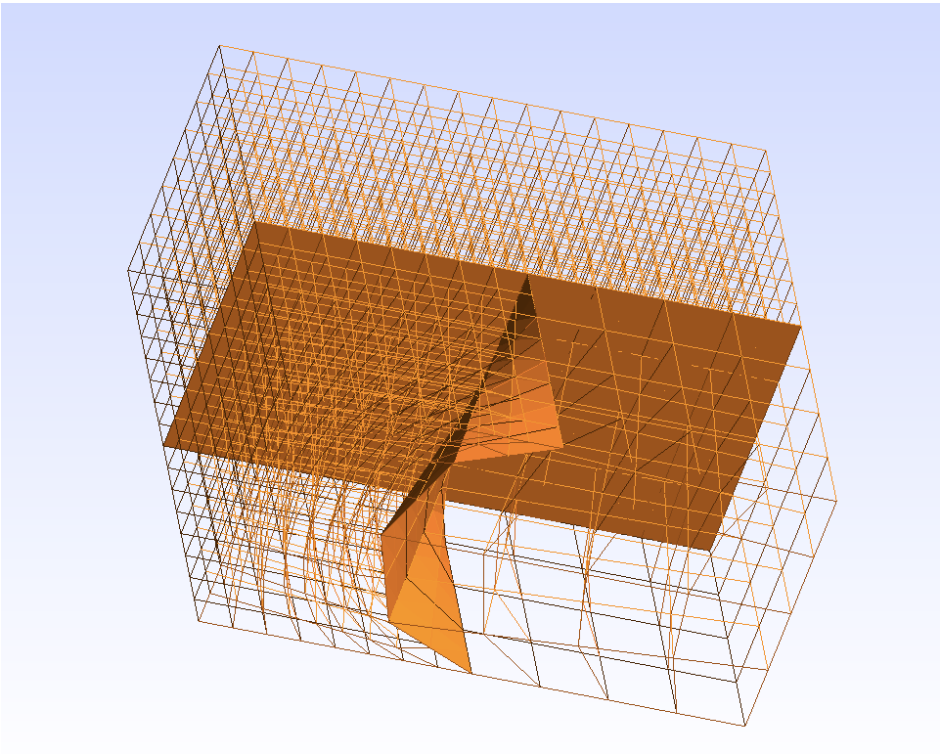
Would be useful: continuous integration...

CLAC data structure

- Subdomain decomposition: each domain is associated to a MPI node. Each MPI node is associated to an OpenCL device (CPU or GPU).
- Zone decomposition: each subdomain is split into volume zones and interface zones. A zone possess identical elements (same order, same geometry, same physical model). A computation kernel is compiled for each zone (for avoiding branch tests).
- (Simple) non-conformity between zones is allowed.
- Geometry and interpolation are separated (possibility to replace memory access by computations).

CLAC data structure

Example of a domain made of two subdomains, three volume zones and three interface zones. the mesh is non-conforming.
Subdomain 1: only one big refined volume zone. Two interface zones.
Subdomain 2: two small volume zones (coarse and refined). Three interface zones.



Nodal interpolation

- Numerical integration: Gauss-Legendre integration points G_k^L, $G_k^{\partial L}$ and weights ω_k^L, $\omega_k^{\partial L}$ on hexaedrons

$$\int_L h(X)dX \simeq \sum_k \omega_k^L h(G_k^L).$$

  Nodal basis function $\varphi_i^L(G_k^L) = \delta_{i,k}$.
- Several optimizations: diagonal mass matrix, complexity $(d+1)^3 \rightarrow 3(d+1)$), etc. [2]

Implementation details

- A single kernel for ∂L and L integration steps. Intermediate results stored in the cache of the compute unit. One processor per Gauss point. The number of Gauss points is different on ∂L and $L \Rightarrow$ some processors are idling in the volume integration step.
- A function class (pointer to the actual function + headers and sources: needed for the OpenCL compilation at runtime). We generally hide memory access into function calls.
- Customized kernels are assembled and compiled for each zone.

CLAC task graph

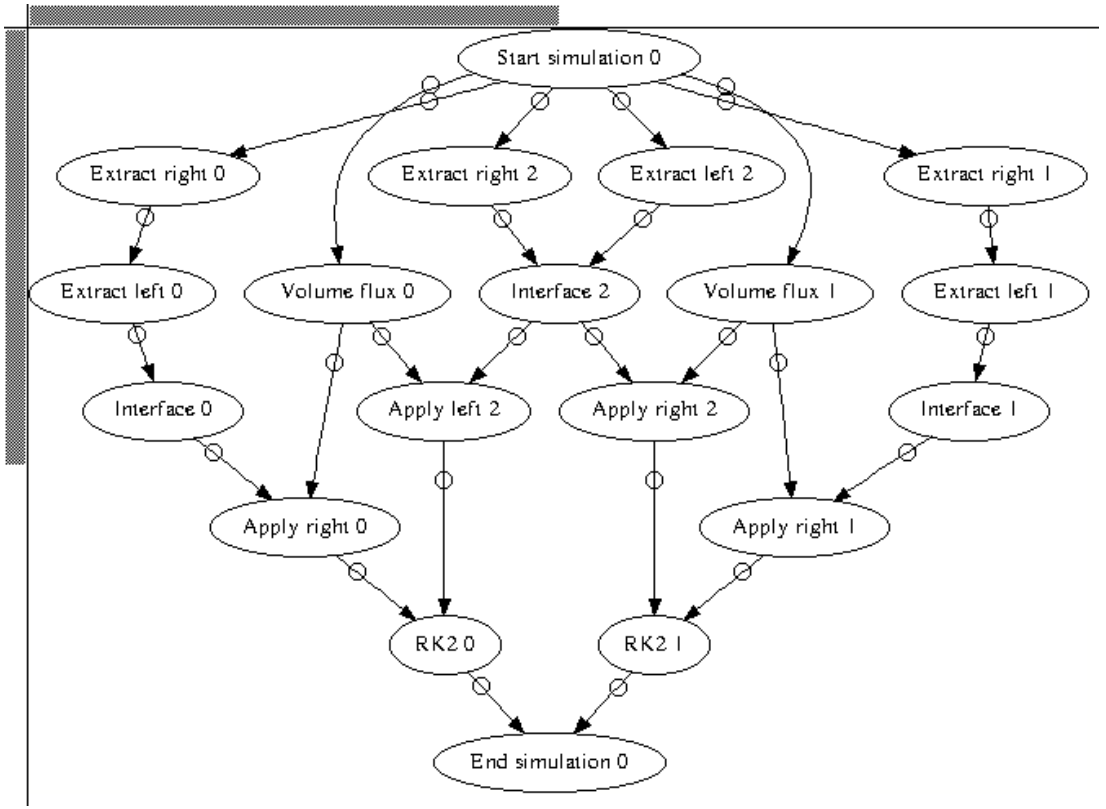
Important kernels:

- Volume zone: internal fluxes and sources assembly ("Volume flux").
- Interface zone: field extraction from right and left volume zones ("Extract left", "Extract right"). The extraction may imply MPI communications.
- Interface zone: flux computations ("Interface").
- Interface zone: boundary fluxes assembly on the left and right volume zones ("Apply right" or "Apply left").
- Volume zones: RK2 integration step ("RK2").

GPU-host transfers occur only in the extraction task at an interface between two subdomains.

Task dependency graph

For the moment, we use Boost Graph.



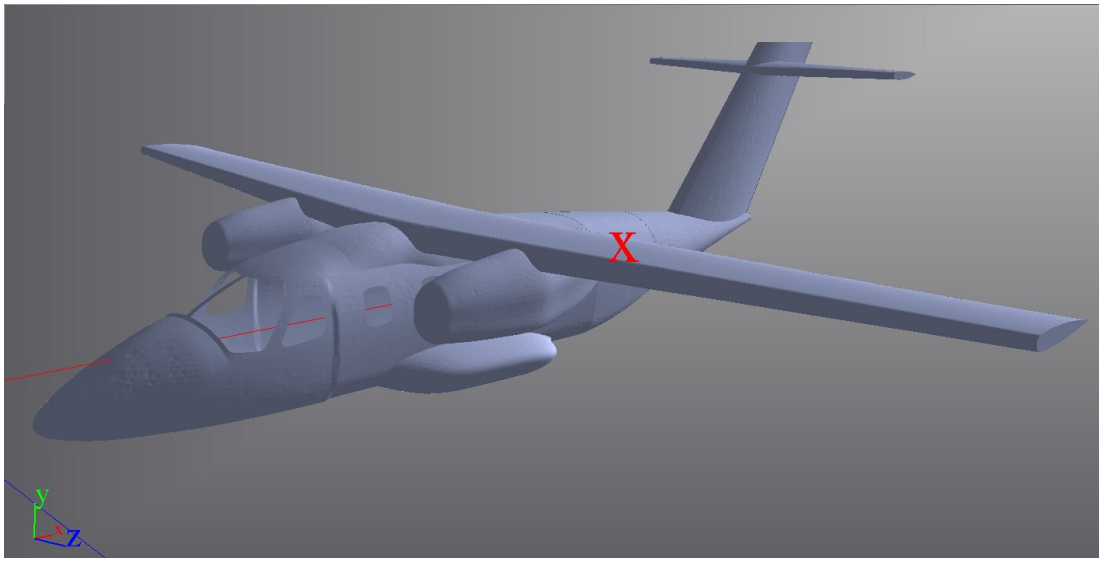
MPI/OpenCL events management

Problem: how to express the dependency between MPI and OpenCL operations ?

- We decided to rely only on the OpenCL events management.
- The beginning of a task depends on the completions of a list of OpenCL events. The task is itself associated to an OpenCL event.
- At an interface zone between two subdomains, an extraction task contains a GPU to host memory transfer, a MPI send/receive communication and a host to GPU transfer.
- we create an OpenCL user event, and launch a MPI blocking sendrecv in a thread. At the end of the communication, in the thread, the OpenCL event is marked as completed. Using threads avoids blocking the main program flow.
- Work in progress: performance evaluations (problem with NVIDIA devices).

Aircraft

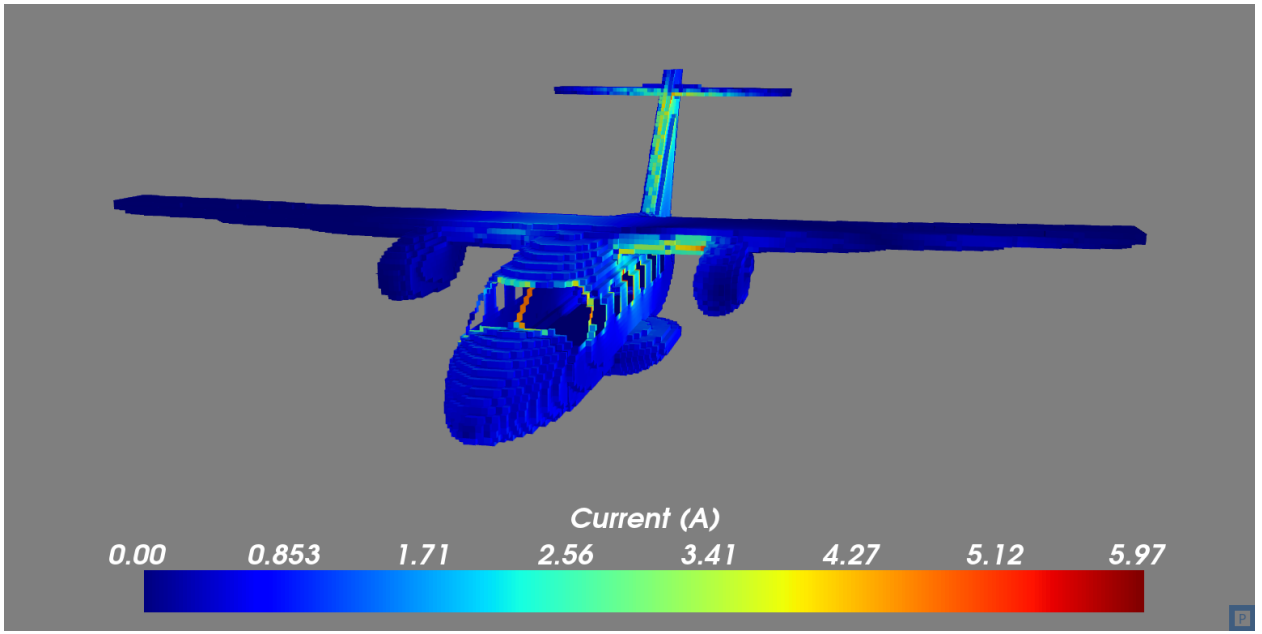
$W = (E_x, E_y, E_z, H_x, H_y, H_z)$: electric and magnetic field. Maxwell equations.
Plane wave with gaussian profile.



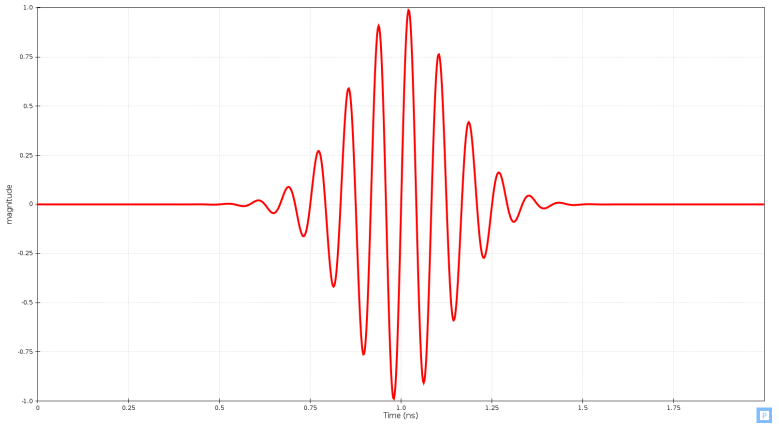
Aircraft

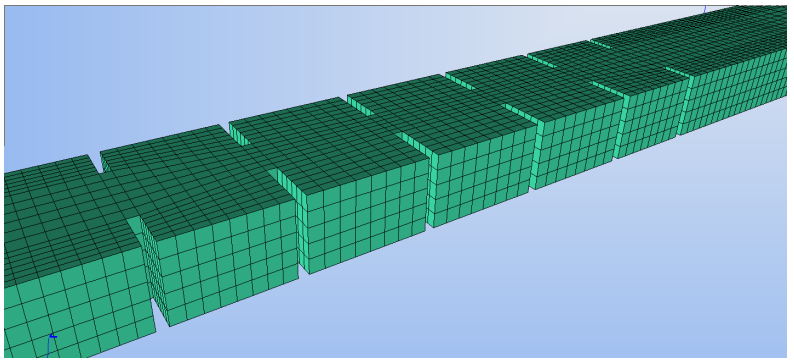
- Aircraft geometry described with $3,337,875$ hexaedrons ($\simeq$1 billion unknowns per time step). Several PML layers at the boudaries.
- We use 8 GPUs to perform the computation. The simulation does not fit into a single GPU memory. 400 Gflops.
- In this test case we spend about 30% of the computation time in the memory transfers between the CPU and the GPU and about 20% in the MPI communications. We expect much better speedups with the asynchronous task graph.

Aircraft



Waveguide filter



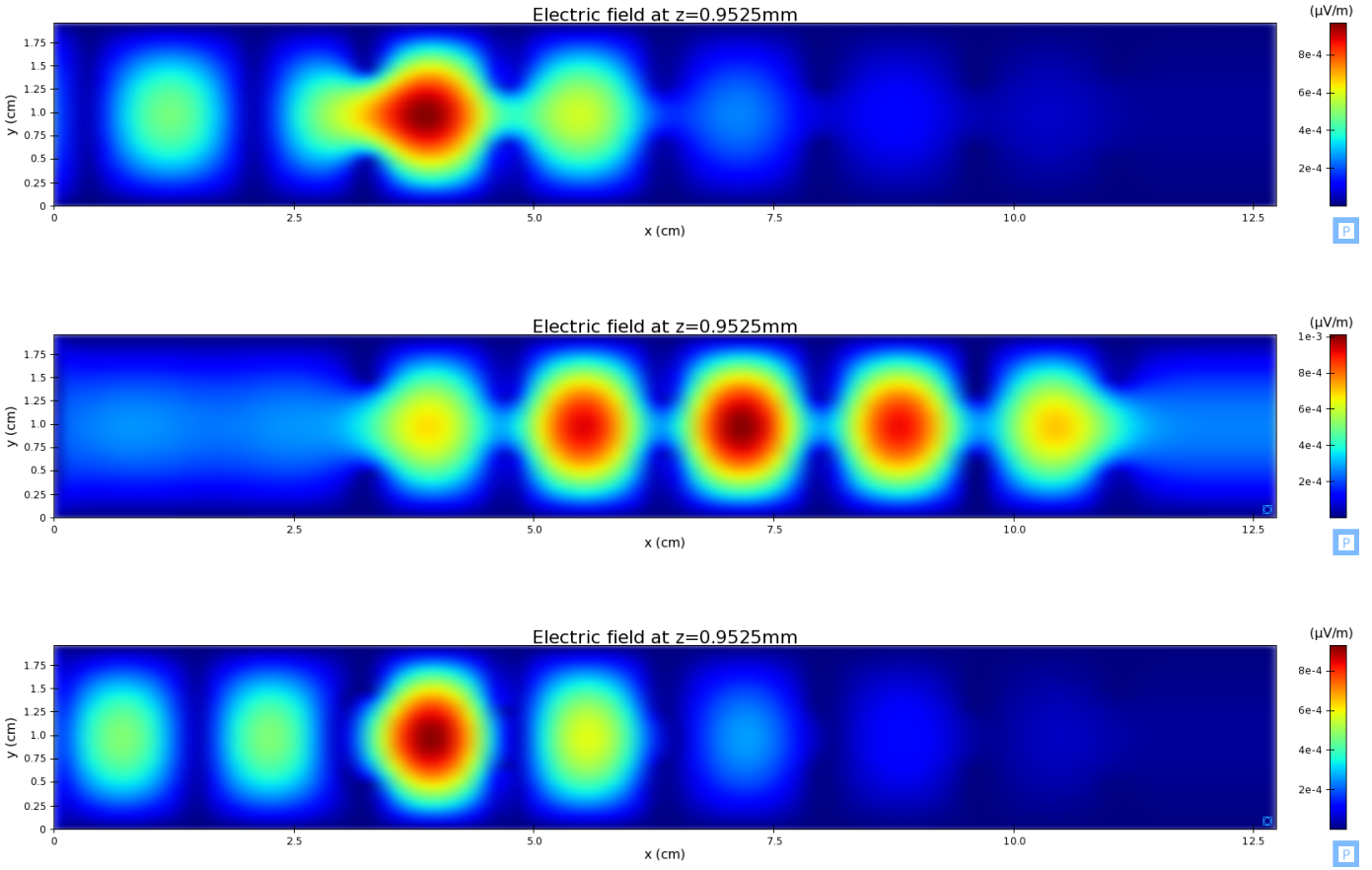


Waveguide filter

- 8370 cells, 3 millions unknowns (second order nodal interpolation).
- 10 cPML layers on two sides.
- dt = 1.36e-13s et dx = 8.14316e-04m. tmax = 25e-9s. 184076 iterations. 19.5mm$\times$9.525mm.
- GPU time 4092s on one NVIDIA GTX 680.

Waveguide filter

Ez at 11.5 GHz, 12 GHz and 12.4 GHz.



Conclusion

- CLAC: asynchronous hybrid DG solver based on OpenCL and MPI.
- It works...
- Work in progress: asynchronous command queues on NVidia GPU, Gauss-Lobatto integration, memory transfer optimization (zone transpositions), etc. Summer CEMRACS project `http://smai.emath.fr/cemracs/cemracs14/lessiv.pdf`
- Task graph: we are reaching our position of incompetence. StarPU, SOCL ? PhD project in C2S@exa.


Tristan Cabel, Joseph Charles, and Stephane Lanteri.
Multi-gpu acceleration of a dgtd method for modeling human exposure to electromagnetic waves.
2011.


Gary Cohen, Xavier Ferrieres, and Sébastien 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.


Philippe Helluy and Jonathan Jung.
Two-fluid compressible simulations on gpu cluster.
2014.


Michel Massaro, Philippe Helluy, and Vincent Loechner.
Numerical simulation for the mhd system in 2d using opencl.
2013.


Greg Ruetsch and Paulius Micikevicius.
Optimizing matrix transpose in cuda.
Nvidia CUDA SDK Application Note, 2009.