A proposal for the implementation of High order Discontinuous Galerkin methods for Elliptic problems on GPUs

Paolo Gorlani

January 11, 2018

Mathematical motivations

- Finite Element Method
- Discontinuous Galerkin methods

Proposal

Implementation

- Data structure
- Strong Scaling
- Weak Scaling
- Early tests on FPGA

Conclusions

Finite Element Method

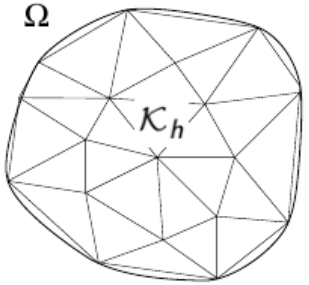
1. Given $f : \Omega \to \mathbb{R}$, find $u : \Omega \to \mathbb{R}$ such that:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{in } \partial\Omega \end{aligned}$$

2. Choose a *mesh* $\mathcal{K}_h$ and a *discrete space* V_h such that:

$$\overline{\Omega} = \bigcup_{K \in \mathcal{K}_h} K$$

$$V_h = \{ v_h \in C^0(\overline{\Omega}) : \ v|_K \in \mathbb{P}^p \quad \forall K \in \mathcal{K}_h \}$$



3. Find $u_h \in V_h$ such that:

$$\int_\Omega \nabla u_h \nabla v_h = \int_\Omega f v_h \quad \forall v_h \in V_h$$

Which is equivalent to the solution of the linear system

$$A\vec{\mathbf{u}} = \vec{\mathbf{f}}$$

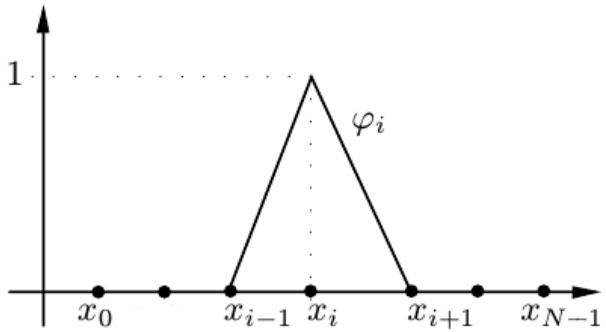


Figure 1: Example of a mono-dimensional basis function φ_i (with $\mathbb{P}^1$).

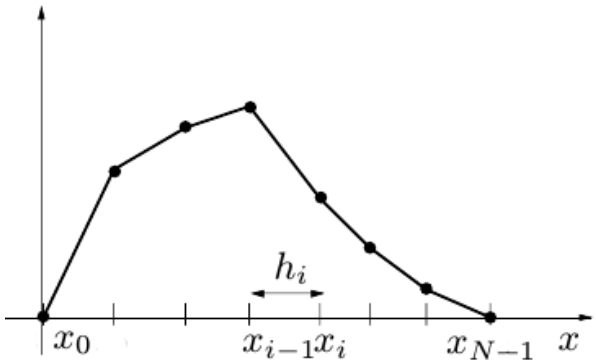


Figure 2: Example of a mono-dimensional function $u_h \in V_h$ (with $\mathbb{P}^1$).

Let $N = \dim(V_h)$ and $\{\varphi_i\}_{0 \leq i < N}$ a set of *basis function* on V_h.

$$A\vec{\mathbf{u}} = \vec{\mathbf{f}}$$

- Stiffness matrix $A \in \mathbb{R}^{N \times N}$

$$A_{ij} = \int_\Omega \nabla\varphi_i \nabla\varphi_j$$

- Right hand side $\vec{\mathbf{f}} \in \mathbb{R}^N$

$$f_i = \int_\Omega f\varphi_j$$

- Solution $\vec{\mathbf{u}} \in \mathbb{R}^N$

$$u_h = \sum_0^{N-1} u_i\varphi_i \in V_h$$

Finite Element Method

Finite Element Method shows two critical aspects in the implementation on GPU:

- Stiffness matrix A assembly implies concurrent access to memory locations assigned to the degrees of freedom shared by different elements.
- The stiffness matrix A is **sparse**, i.e. most of the elements are zero. The solution of the linear system $A\vec{\mathbf{u}} = \vec{\mathbf{f}}$ with iterative methods involves huge number of multiplications between A and generic vectors.

These make difficult to fully exploit the memory bandwidth of GPUs because memory operations do not coalesce.

Discontinuous Galerkin methods

The **Symmetric Interior Penalty Galerkin** method belongs to the class of Discontinuous Galerkin methods. The problem introduced previously has the following formulation.

- Choose a *discrete space* V_h such that:

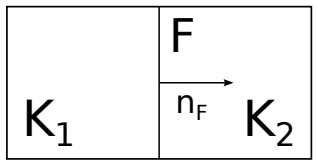
$$V_h = \{ v \in L^2(\Omega) : v|_K \in \mathbb{P}^p \quad \forall K \in \mathcal{K}_h \}$$

- Find $u_h \in V_h$ such that:

$$\sum_{K \in \mathcal{K}_h} \int_K \nabla u_h \nabla v_h - \sum_{F \in \mathcal{F}_h} \int_F \{\{\nabla u_h \cdot \mathbf{n}_F\}\} [\![v_h]\!] - \sum_{F \in \mathcal{F}_h} \int_F \{\{\nabla v_h \cdot \mathbf{n}_F\}\} [\![u_h]\!] + \sum_{F \in \mathcal{F}_h} \frac{\eta}{h_F} \int_F [\![u_h]\!] [\![v_h]\!] = \int_\Omega f v_h \qquad \forall v_h \in V_h$$

The average $\{\{\cdot\}\}$ and the jump $[\![\cdot]\!]$ operators are defined as

$$\{\{v\}\} = \tfrac{1}{2}(v|_{K_1} + v|_{K_2})$$
$$[\![v]\!] = (v|_{K_1} - v|_{K_2})$$



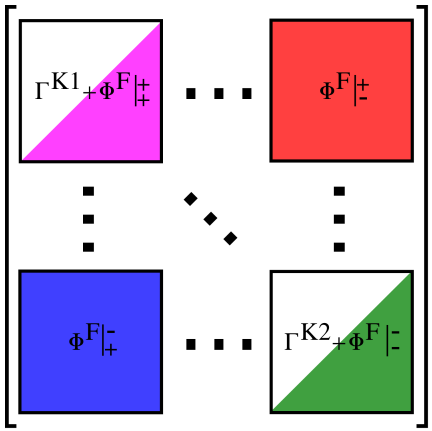
- It is possible to rewrite the previous formulation as

$$\sum_{K \in \mathcal{K}_h} \gamma^K(u,v) + \sum_{F \in \mathcal{F}_h^i} \phi^F(u,v) = \int_\Omega f v_h \qquad \forall v_h$$

$$\gamma^K(u,v) = \int_K \nabla u \nabla v$$

$$\phi^F(u,v) = \int_F \frac{\eta}{h_F} [\![u]\!][\![v]\!] - \{\{\nabla u \cdot \mathbf{n}_F\}\}[\![v]\!] - \{\{\nabla v \cdot \mathbf{n}_F\}\}[\![u]\!]$$

- The stiffness matrix A is build considering the following discrete operators, where B^K is the set of the basis functions on $K \in \mathcal{K}_h$ and $K_1, K_2 \in \mathcal{K}_h$ such that $F = K_1 \cap K_2 \in \mathcal{F}_h$.



$$\Gamma_{\mathbf{ij}}^K = \gamma^K(\varphi_\mathbf{i}, \varphi_\mathbf{j}) \quad \forall \varphi_\mathbf{i},\ \varphi_\mathbf{j} \in B^K$$

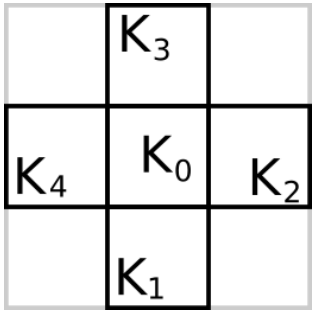
$$\Phi_{\mathbf{ij}}^F = \begin{cases} \Phi_{\mathbf{ij}}^F|_+^+ = \phi^F(\varphi_\mathbf{i}, \varphi_\mathbf{j}) & \text{if } \varphi_\mathbf{i},\ \varphi_\mathbf{j} \in B^{K_1} \\ \Phi_{\mathbf{ij}}^F|_-^+ = \phi^F(\varphi_\mathbf{i}, \varphi_\mathbf{j}) & \text{if } \varphi_\mathbf{i} \in B^{K_1},\ \varphi_\mathbf{j} \in B^{K_2} \\ \Phi_{\mathbf{ij}}^F|_-^- = \phi^F(\varphi_\mathbf{i}, \varphi_\mathbf{j}) & \text{if } \varphi_\mathbf{i},\ \varphi_\mathbf{j} \in B^{K_2} \\ \Phi_{\mathbf{ij}}^F|_+^- = \phi^F(\varphi_\mathbf{i}, \varphi_\mathbf{j}) & \text{if } \varphi_\mathbf{i} \in B^{K_2},\ \varphi_\mathbf{j} \in B^{K_1} \end{cases}$$

Proposal

- The **matrix vector product** is the key operation in any iterative solver used in numerical methods for partial derivative equations. It is also the most critical because it involves sparse matrices.
- Our proposal consists of **merging the generation of the local components and the assembly of the stiffness matrix into the matrix vector product**. The local matrix actions are applied directly to the vector of the degrees of freedom.
- In such a way, the storage of the stiffness matrix is avoided, its components are re-computed every time, saving device memory and preventing non-coalescing memory accesses.

$$\vec{y} = A\vec{x}$$

The stiffness matrix-vector products can be seen as the sum of local operators contributions.



$$\begin{aligned}\vec{y}^{K_0} = & \left(\Gamma^{K0} + \Phi^{F_1}|_{-}^{-} + \Phi^{F_2}|_{+}^{+} + \Phi^{F_3}|_{+}^{+} + \Phi^{F_4}|_{-}^{-} \right) \vec{x}^{K_0} \\ & + \Phi^{F_1}|_{+}^{-} \vec{x}^{K_1} \\ & + \Phi^{F_2}|_{-}^{+} \vec{x}^{K_2} \\ & + \Phi^{F_3}|_{-}^{+} \vec{x}^{K_3} \\ & + \Phi^{F_4}|_{+}^{-} \vec{x}^{K_4}\end{aligned}$$

$\vec{v}^K$ contains the values associated to the *dof* in $K \in \mathcal{K}_h$

Implementation

In order to test the potentials of the proposal, I developed a *prototype* solver for the numerical solution of the following Problem on a structured quadrilateral mesh.

It is possible to set

- the number of mesh elements
- the degree of the basis function.

In sake of this, I implemented codes for

- compute basis function values,
- handle quadrilateral meshes,
- performs the matrix-vector product (single and multi node),
- Conjugate Gradient solver (single and multi node),
- simple (sub-optimal) Jacobi preconditioner.

The code is written in CUDA, C++ and MPI.

Data structure

The vector of the *dof* is stored in **node-wise** ordering:
A1 A2 A3 A4 B1 B2 B3 B4 C1 C2 C3 C4 D1 D2 D3 D4

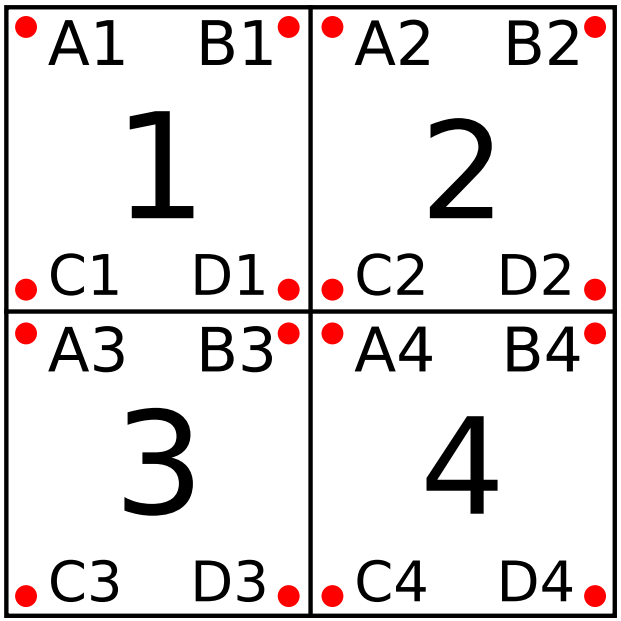


Figure 3: Four element mesh.

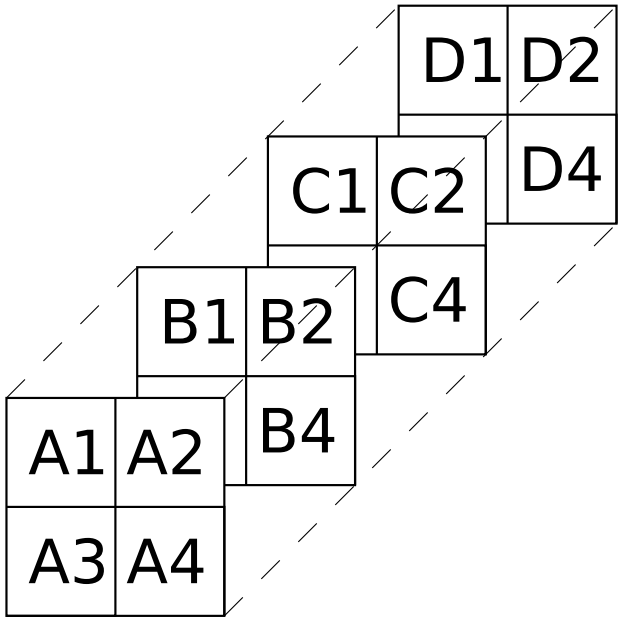
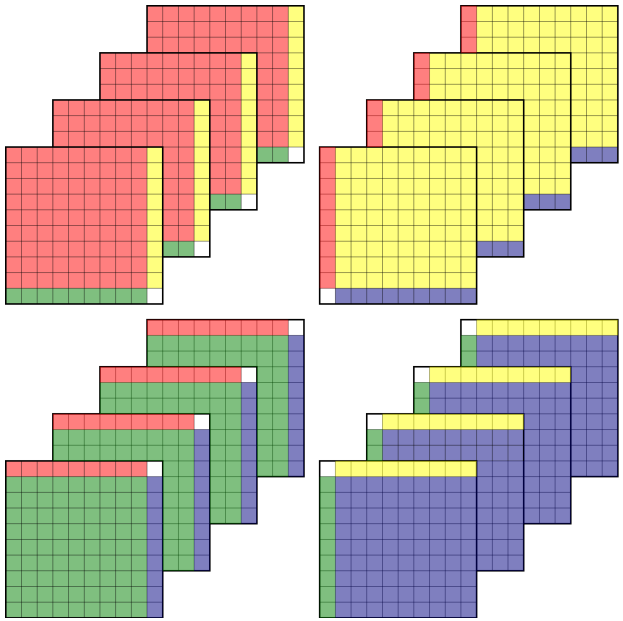


Figure 4: Degree of freedom arrangement.

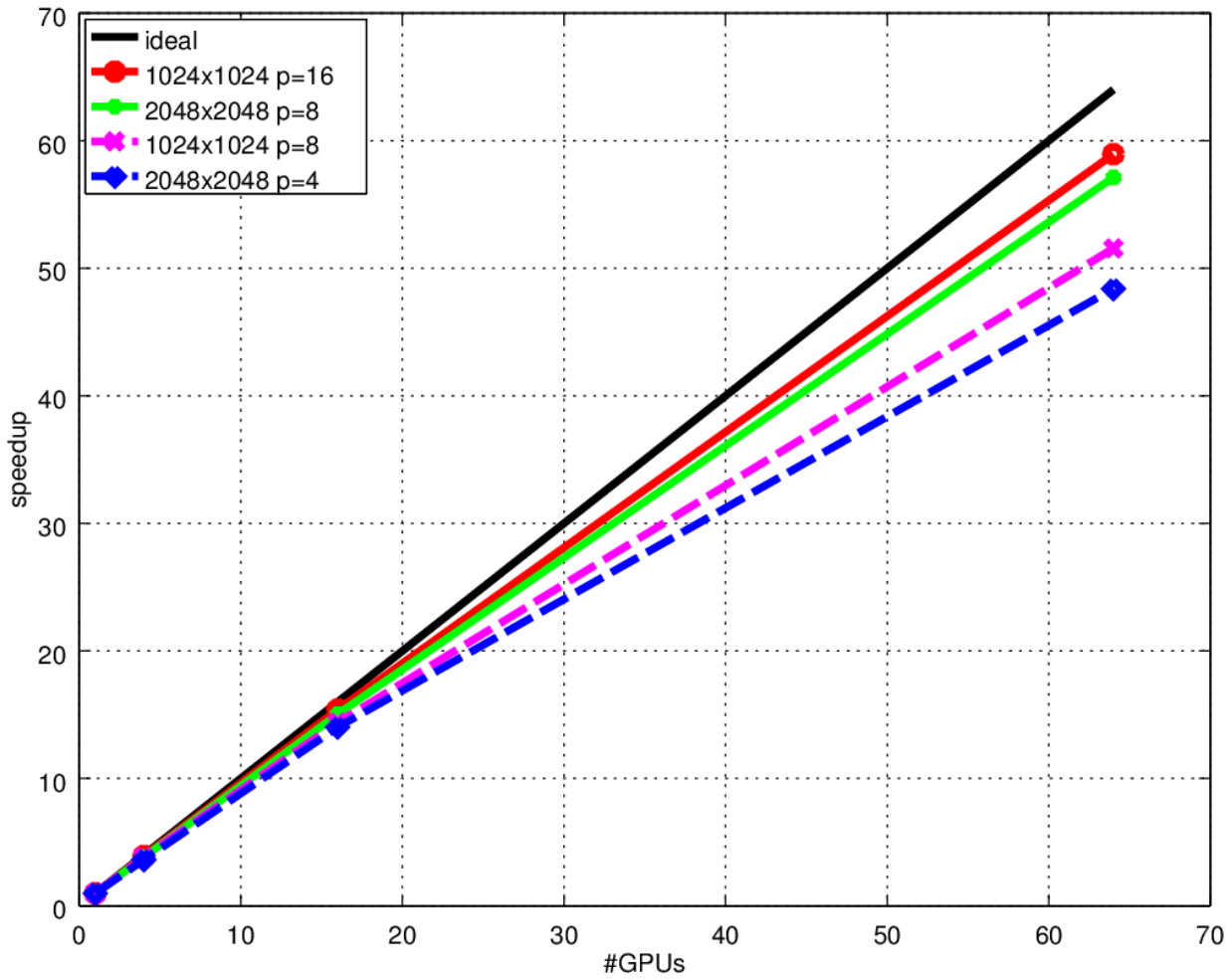
Data structure

This kind of data structure makes easy to spread the computation along different nodes.



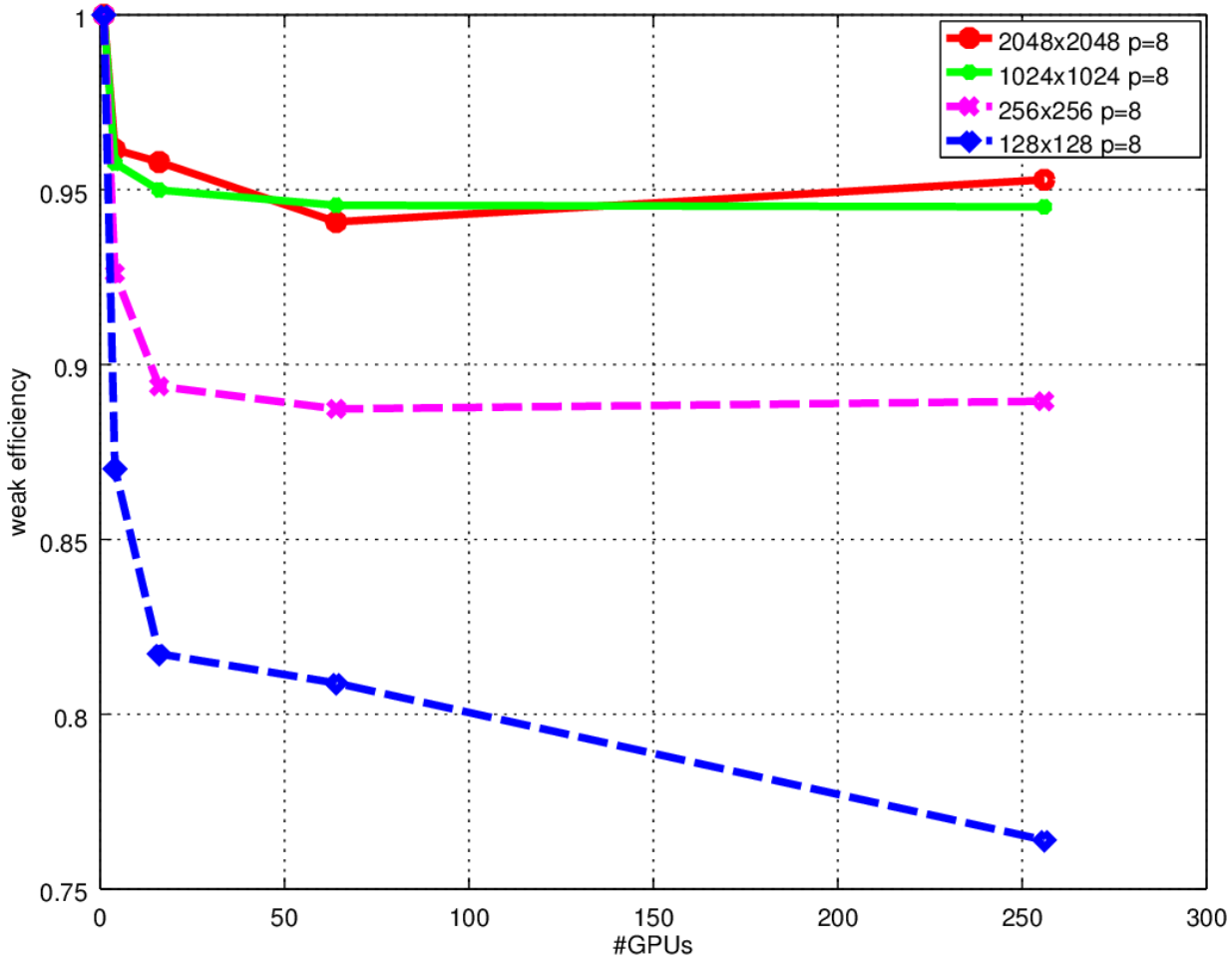
A multi-node (1 MPI node = 1 GPU) version of the code is implemented. The halo exchange is done using GCL library (provided by Mauro Bianco, CSCS) which is a part of GridTools.

Strong Scaling



The plot shows the speed-up of the *matrix vector product* performed on multiple nodes (1 node = 1 GPUs [K20]) of Piz Daint (CSCS).

Weak Scaling



The plot shows the weak scaling efficiency of the *matrix vector product* performed on multiple nodes (1 node = 1 GPUs [K20]) of Piz Daint (CSCS).

Early tests on FPGA

I ported the kernel that applies the action of the Γ operator to FPGA using the *SDAccel Development Environment*. Then, I tested it on a *Virtex UltraScale+ VU9P FPGA* provided by an Amazon AWS F1 instance.

High Level Synthesis reports:

- All memory accesses performed in bursts.
- All loops pipelined with II=1.

p	2	4	8	16
fpga [VU9P]	34.86	92.42	301.27	1684.44
gpu [K20x]	0.24	0.97	3.84	61.36

Figure 5: Execution times [ms] varying the basis degree p.

Conclusions

- Matrix vector product is seen as a stencil code.
- The developed kernels reach a high occupancy and exploit most of the available memory bandwidth.
- The properties of the proposed approach seems to be well-suited also for FPGA.