EPSI Bordeaux
École Privée des Sciences Informatiques

Rapport de stage

Implémentation multi GPU de la méthode *Spectral Differences* pour un code de CFD

Adrien Cassagne

Sous la direction d'Isabelle d'Ast, Jean-François Boussuge,
Guillaume Puigt et Nadège Villedieu

Toulouse, le 31 janvier 2014

Réf. : WN-CFD-14-5

Remerciements

Je tiens à remercier dans un premier temps, toute l'équipe pédagogique de l'EPSI Bordeaux et les professeurs responsables du cycle d'ingénierie informatique, pour avoir assuré la partie théorique de ma formation.

Je veux aussi dire ma reconnaissance aux personnes suivantes, pour l'expérience enrichissante et pleine d'intérêt qu'elles m'ont faite vivre durant ces dix mois au sein du CERFACS :

M^me^ Isabelle D'Ast, ingénieure HPC, pour son accueil, son aide, ses conseils techniques, et la confiance qu'elle m'a accordée dès mon arrivée dans l'entreprise.

M^me^ Nadège Villedieu, post doctorante en CFD, pour m'avoir intégré rapidement au sein de le l'équipe CFD et m'avoir accordé toute sa confiance ; pour le temps qu'elle m'a consacré tout au long de cette période, sachant répondre à mes interrogations.

M. Jean-François Boussuge, mon maître de stage, chef de l'équipe CFD, pour le temps qu'il a consacré à la validation du code ; pour la confiance qu'il m'a accordé tout au long du stage.

M. Guillaume Puigt, chercheur sénior en CFD, pour ses explications sur les parties physiques du projet et ses conseils sur les directions à suivre.

M. Nicolas Monnier, directeur des systèmes d'information, pour la confiance et les moyens mis à ma disposition ; pour l'intérêt porté à mon projet et pour ses informations matérielles.

M^me^ Isabelle D'Ast, M^me^ Nadège Villedieu, M. Guillaume Puigt et M. Nicolas Monnier pour la relecture en profondeur et les corrections apportées à ce document.

M. Jean-François Boussuge, M^me^ Nadège Villedieu et M. Guillaume Puigt pour leur accueil chaleureux au sein d'une équipe très agréable et efficace ; pour les excellentes conditions de travail.

Ce stage de fin d'étude a été pour moi l'occasion de mettre mes connaissances en application dans le monde de l'entreprise. Le projet proposé était à la hauteur de mes attentes et m'a permis de monter en compétences. J'ai apprécié travailler sur plusieurs architectures (CPU et GPU) et je pense que c'est un atout d'avoir pu les comparer avec précision. Enfin et surtout, j'ai eu la chance d'être intégré par des équipes professionnelles et compétentes avec qui le dialogue a été simple et naturel.

Table des matières

I Introduction . . . 1
I.1 Introduction sur la mécanique des fluides numérique . . . 1
I.2 Contexte physique . . . 2
I.3 Contexte numérique . . . 2
I.4 Les équations d'Euler . . . 2
I.5 La méthode des Différences Spectrales . . . 4
II Présentation . . . 7
II.1 Le CERFACS . . . 7
II.2 La simulation dans l'industrie . . . 8
II.3 Le projet `JAGUAR` . . . 8
II.4 La technologie CUDA . . . 10
II.5 L'architecture *Kepler* . . . 10
II.6 Organisation du travail . . . 14
III Modifications algorithmiques . . . 15
III.1 Implémentation de la méthode des Différences Spectrales . . . 15
III.2 Méthodes de calcul numérique . . . 17
III.3 Réorganisation de la mémoire . . . 21
III.4 Algorithme glouton . . . 26
IV Implémentation GPU . . . 28
IV.1 Accès mémoire coalescés et grain de calcul . . . 28
IV.2 Répartition des temps de calcul par routine . . . 30
IV.3 Optimisation du solveur de Riemann . . . 31
IV.4 Optimisation des produits matriciels . . . 31
IV.5 Version multi GPU . . . 34
V Expérimentations . . . 38
V.1 Conditions de test . . . 38
V.2 Analyse des performances CPU . . . 41
V.3 Analyse des performances mono GPU . . . 43
V.4 Analyse des performances multi GPU . . . 45
VI Conclusion . . . 48
Bibliographie . . . 49

I Introduction

I.1 Introduction sur la mécanique des fluides numérique

La mécanique des fluides numérique (*Computational Fluid Dynamics* en Anglais) est une nouvelle science appliquée dont l'objectif est de déterminer les écoulements autour d'objets par la simulation numérique. Le principe est de partir des équations régissant le mouvement du fluide, de les discrétiser en fonction d'un paradigme mathématique et enfin d'implémenter cette approche dans un code de CFD. Les calculs permettent d'obtenir des informations fines sur le comportement aléatoire de l'écoulement : on parle alors d'écoulement turbulent. La turbulence se manifeste par l'apparition et la transformation d'un ensemble de tourbillons caractérisés par leur taille et leur énergie. Les plus gros tourbillons sont reliés à la taille de l'objet et les plus petits sont dissipés par effet de viscosité. La viscosité d'un fluide représente sa capacité à s'opposer à sa mise en mouvement. La turbulence se rencontre quotidiennement : les sillages d'avion et la fumée d'un panache d'usine sont turbulents, la météorologie résout les équations de la turbulence en milieu stratifié...

Comme cela a été introduit, la CFD se base sur une version discrétisée des équations. Sans rentrer dans tous les détails, on peut simplement dire que le domaine de calcul est décomposé en petits volumes élémentaires. Deux formalismes sont possibles :

- une approche **structurée** (cf. Fig. 1) qui consiste à décomposer le domaine de calcul en blocs héxaédriques et sur chaque bloc, à ranger ces volumes de manière structurée, suivant un remplissage pour chaque direction i, j, k de l'espace. Cette approche permet de construire des maillages [1] anisotropes (les propriétés ne sont pas toujours les mêmes pour les 3 directions) alignant la physique et le maillage. Toutefois, il n'est pas possible de construire automatiquement un maillage structuré, sauf cas très particulier. De plus, le temps de génération est directement relié à la complexité de la géométrie.
- une approche **non structurée** (cf. Fig. 2) qui consiste à décomposer le domaine de calcul en volumes élémentaires. Ici, il n'y a pas de direction privilégiée et cette approche est généralement choisie car des outils automatiques permettent de construire rapidement les maillages.

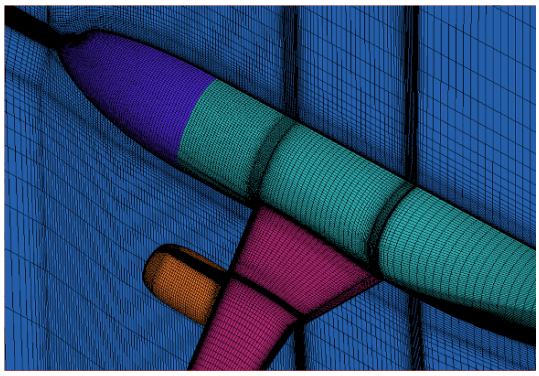

FIG. 1 – Maillage **structuré** d'un avion

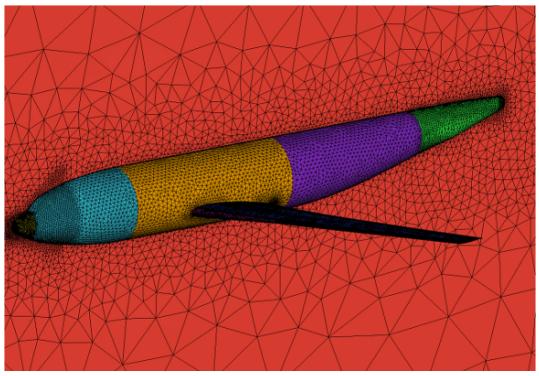

FIG. 2 – Maillage **non structuré** d'un avion

1. Maillage : discrétisation spatiale d'un milieu continu

I.2 Contexte physique

Le CERFACS [2] développe depuis plus de 20 ans des outils de résolution des équations de Navier-Stokes en régime turbulent. Plusieurs approches sont possibles mais ce mémoire se focalise sur **une approche nécessitant peu de modélisation**. Ainsi, cette approche est très précise. Comme cela a été indiqué, la turbulence est caractérisée par une taille de tourbillon et une énergie associée. Dans la suite, nous nous focalisons sur une approche dite *Large Eddy Simulation* ou Simulation aux Grandes Echelles dans laquelle tous les gros tourbillons sont effectivement calculés et les plus petits (ceux de taille inférieure à la maille) sont modélisés. La capture des phénomènes turbulents nécessite un maillage très fin et des temps de calcul très importants. Dans ce mémoire, le modèle régissant le fluide est relativement simpliste (équations d'Euler). L'enjeu est de l'optimiser au maximum afin d'être le plus performant possible.

I.3 Contexte numérique

Les calculs en LES nécessitent des **discrétisations précises** afin de bien capturer le spectre turbulent. La communauté de la CFD parle alors d'approches *High-Order*. En pratique, une approche est dite d'ordre élevée si l'écart séparant une fonction exacte de son approximation numérique est d'ordre supérieur ou égal à 3. Sur le plan mathématique, cela peut être représenté avec les développements de Taylor : on approche une fonction par un polynôme de degré au moins 3. En suivant le formalisme de Taylor, il est clair qu'il est plus facile de contrôler l'ordre de l'approximation sur un maillage structuré que sur un maillage non structuré. En effet, il suffit d'utiliser l'approximation de Taylor dans chacune des directions du bloc de maillage. En non structuré, il n'est plus possible d'utiliser simplement une approximation de type Taylor pour définir l'ordre du schéma numérique. La construction d'une approche d'ordre élevée en non structuré est donc très difficile. Toutefois, les besoins industriels se focalisent autour de la LES sur CAO [3] complexe.

Dans ce cadre, le CERFACS travaille sur une nouvelle approche pour maillages non structurés composés exclusivement d'hexaèdres : l'approche des Différences Spectrales (*Spectral Difference Method* dans la littérature anglaise). Avant d'introduire cette approche en Sec. I.5, nous commençons par introduire les équations d'Euler qui régissent le fluide.

I.4 Les équations d'Euler

Les équations d'Euler sont obtenues à partir de l'analyse de la masse, de la quantité de mouvement et de la variation de l'énergie totale à l'intérieur du domaine. Ces équations sont d'abord obtenues sous la forme d'intégrales. Cependant, ces dernières sont valides quelque soit le domaine d'étude : la forme intégrale peut alors être ignorée pour obtenir une formulation valable en tout point de l'espace et à tout instant. Les équations suivantes reposent sur plusieurs variables :

– ρ : la densité (en $kg.m^{-3}$),

2. CERFACS : Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique
3. CAO : Conception Assistée par Ordinateur

– $\overrightarrow{u}$: le vecteur vitesse et ses composantes u, v et w suivant les directions x, y et z (en $m.s^{-1}$),
– E : l'énergie totale (en $m^2.s^{-2}$),
– P : la pression statique (en Pa).

L'énergie totale E est définie par la somme de l'énergie interne e (liée à la température) et de l'énergie cinétique :

$$E = e + \frac{\|\vec{u}\|^2}{2}. \tag{1}$$

En complément de ces grandeurs physiques, il est bon d'introduire :

– *l'opérateur produit tensoriel.* Pour deux vecteurs $\overrightarrow{f}$ et $\overrightarrow{g}$, de composantes $(f_i)_{1\leq i\leq 3}$ et $(g_i)_{1\leq i\leq 3}$, on construit une matrice $f \otimes g$ définie par :

$$(f \otimes g)_{i,j} = f_i g_j, \tag{2}$$

– *l'opérateur divergence.* Pour une fonction $\mathcal{L}$ à 3 composantes $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$ suivant les directions x, y et z, la divergence de $\mathcal{L}$ s'écrit :

$$\nabla \cdot \mathcal{L} = \frac{\partial \mathcal{L}_1}{\partial x} + \frac{\partial \mathcal{L}_2}{\partial y} + \frac{\partial \mathcal{L}_3}{\partial z}, \tag{3}$$

– *l'opérateur gradient.* Pour une fonction scalaire f dépendant de (x, y, z), le gradient de f est un vecteur :

$$\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}). \tag{4}$$

Finalement, voici les équations d'Euler valides pour toute position d'espace et de temps :

$$\partial_t \rho + \nabla \cdot (\rho \overrightarrow{u}) = 0 \tag{5}$$
$$\partial_t (\rho \overrightarrow{u}) + \nabla \cdot (\rho \overrightarrow{u} \otimes \overrightarrow{u}) + \nabla P = 0 \tag{6}$$
$$\partial_t (\rho E) + \nabla \cdot (\overrightarrow{u}(\rho E + P)) = 0. \tag{7}$$

Le système composé des équations Eq. 5, Eq. 6 et Eq. 7 est un système ouvert avec plus d'inconnues que d'équations. Cependant, il peut être fermé en considérant d'autres propriétés physiques. Dans ce document, nous considérons que le fluide est l'air et qu'il est caractérisé par l'équation des gaz parfaits :

$$P = \rho R T, \tag{8}$$

où T est la température (en Kelvin) et R est une constante spécifique au fluide (pour l'air, $R = 287.05\ m^2 s^{-2} K^{-1}$). À ce niveau, le système est toujours ouvert puisqu'il n'y a pas de relation entre la température et l'énergie. Nous considérons alors que l'air est thermodynamiquement parfait, ce qui signifie que l'air est un gaz polytropique. Un gaz polytropique est caractérisé par une capacité thermique constante à pression et à volume constant (respectivement C_p et C_v). Dans notre cas, γ est la constante polytropique. C_p et C_v sont définis comme suit :

$$C_p = \frac{\gamma R}{\gamma - 1} \text{ et } C_v = \frac{R}{\gamma - 1}, \tag{9}$$

ce qui amène à $\gamma = C_p/C_v = 1.4$. De plus, il existe une relation linéaire entre l'énergie interne et la température :

$$e = C_v T. \tag{10}$$

Enfin, les équations d'Euler peuvent être écrites sous la forme compacte :

$$\frac{\partial Q}{\partial t} + \nabla \cdot \mathcal{F} = 0, \tag{11}$$

où Q est le vecteur des variables conservatives $Q = (\rho, \rho u, \rho v, \rho w, \rho E)^t$ et $\mathcal{F}$ peut être exprimé par ses trois composantes (F, G, H) en trois dimensions :

$$F = \begin{pmatrix} \rho u \\ P + \rho u^2 \\ \rho uv \\ \rho uw \\ u(\rho E + P) \end{pmatrix}, \qquad G = \begin{pmatrix} \rho v \\ \rho uv \\ P + \rho v^2 \\ \rho vw \\ v(\rho E + P) \end{pmatrix}, \qquad H = \begin{pmatrix} \rho w \\ \rho uw \\ \rho vw \\ P + \rho w^2 \\ w(\rho E + P) \end{pmatrix}. \tag{12}$$

Ce système d'équations est maintenant fermé en utilisant Eq. 8, Eq. 9 et Eq. 10.

I.5 La méthode des Différences Spectrales

Dans cette section, nous illustrons la méthode des Différences Spectrales dans le cas à une dimension. Cette dernière permet de résoudre les équations d'Euler présentées dans la section précédente. Notons que cette méthode est appliquée à chaque élément du maillage. Les solutions Q_x sont chacune constituées des 5 variables conservatives présentées dans la section I.4. Ces dernières sont connues au départ : elles représentent l'état physique initial (à l'instant t_0). L'objectif de la méthode est de calculer les solutions à l'instant $t_0 + \delta t$, donc après un intervalle de temps δt. Cette avance en temps se fait en différentes étapes.

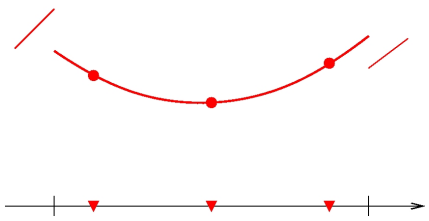

Fig. 3 – Étape 0 : les solutions (ronds rouges) sont disponibles en plusieurs points solution (triangles rouges)

On considère que la solution est disponible en plusieurs points appelés points solutions. Leur position spatiale est prédéfinie (triangles rouges) et les valeurs des champs Q_x sont représentés par les cercles rouges (Fig. 3). On appellera par la suite degré de liberté tout point solution. Sur la Fig. 3, il y a ainsi 3 degrés de liberté dans un segment. A l'aide des champs aux points solution, on peut définir un polynôme d'interpolation dont le degré maximal est directement lié au nombre de degrés de liberté. Ce polynôme est représenté en trait continu rouge sur la Fig. 3. Dans notre cas modèle, 3 degrés de liberté permettent

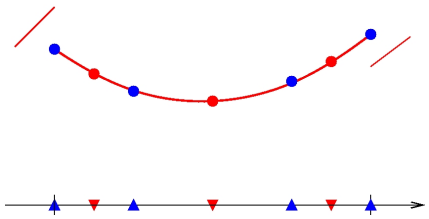

FIG. 4 – Étape 1 : extrapolation du polynôme passant par les solutions (ronds rouges) pour calculer les solutions extrapolés (ronds bleus)

de définir univoquement un polynôme de degré 2. Plus généralement, pour un polynôme de degré p, il y a $p+1$ degrés de liberté.

La méthode propose ensuite d'extrapoler le polynôme de degré p pour calculer $p+2$ champs en des points définis comme points flux (triangles bleus). Ces derniers sont situés soit entre 2 points solution, soit au bord du segment (Fig. 4).

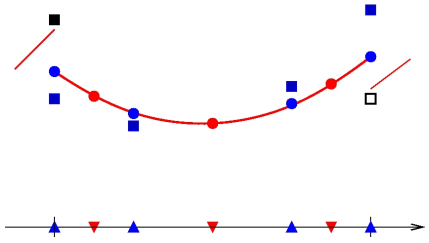

FIG. 5 – Étape 2 : calcul des flux (carrés bleus) grâce aux solutions extrapolés (ronds bleus)

Grâce aux $p+2$ champs solutions extrapolés (ronds bleus), on calcule les $p+2$ flux (F, G, H) correspondants (carrés bleus) (Fig. 5). En effet, on voit que les termes de Eq. 12 peuvent être aisément calculés à partir des champs conservatifs originaux $\rho, \rho\vec{u}, \rho E$. À l'interface entre deux éléments, les champs n'ont aucune raison d'être identiques par extrapolation, les flux associés sont donc généralement discontinus (carré noir et carré à fond blanc).

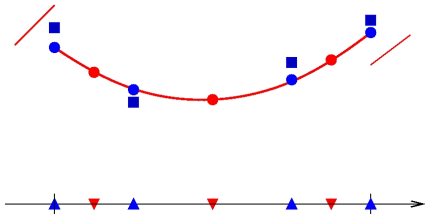

FIG. 6 – Étape 3 : calcul des flux (carrés bleus) physiquement corrects aux bords grâce au solveur de RIEMANN

Or, les flux aux bords permettent de faire transiter l'information entre 2 volumes adjacents. Cela explique qu'il est préférable, pour garantir la conservation des quantités que les flux de part et d'autre d'une interface soient identiques. Pour cela, on tient compte des écarts des champs extrapolés par résolution d'un problème de RIEMANN (Fig. 6).

La résolution d'un problème de RIEMANN étant complexe, on utilisera un solveur de RIEMANN approché appelé schéma de Roe [1]. Une étude plus complète et mathématique des solveurs de RIEMANN est disponible dans le livre de Toro [2].

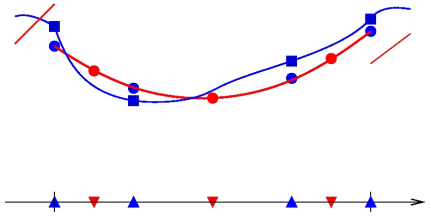

FIG. 7 – Étape 4 : définition d'un polynôme de degré $p+1$ passant par les flux

Une fois le flux obtenu au bord, on détermine un polynôme de degré $p+1$ qui passe par les $p+2$ flux (Fig. 7). Notons que le polynôme global du flux (sur tout le domaine) est par construction continu, alors que les champs originaux sont discontinus à l'interface entre 2 volumes.

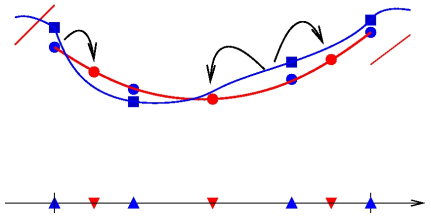

FIG. 8 – Étape 5 : dérivation des flux dans les solutions

Enfin, le polynôme de flux représente les termes non linéaires hors dérivée spatiale issus d'Eq. 12. Il faut donc calculer le terme en divergence, c'est-à-dire dériver le polynôme obtenu. La dérivée du polynôme de flux est simplement calculée aux points solution (Fig. 8). Ainsi, le polynôme de flux est d'ordre $p+1$ et passe par $p+2$ points, alors que sa dérivée est d'ordre p et est fournie en $p+1$ points.

Un schéma d'intégration en temps (schéma de RUNGE-KUTTA : basé sur une version améliorée de l'approximation de TAYLOR) est utilisé pour faire l'avance en temps :

$$\frac{\partial Q}{\partial t} \simeq \frac{Q(t+\delta t) - Q(t)}{\delta t}.$$

En dimension supérieure à 1, le traitement reste presque inchangé. Les points solution sont définis dans toutes les directions et le traitement 1D est effectué direction par direction.

Dans les sections suivantes, par abus de langage, nous parlerons parfois du degré d'un élément. Ce dernier correspond au degré du polynôme qui passe par les degrés de liberté d'un élément. S'il y a $p+1$ degrés de liberté alors le degré de l'élément est p.

Pour plus de détails sur la méthode des Différences Spectrales se référer à l'article [3] de Z. J. Wang.

II Présentation

II.1 Le CERFACS

Le CERFACS (Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique) est un centre de recherche dont l'objectif est de développer des méthodes de simulation numérique avancées ainsi que les solutions algorithmiques qui adressent les plus grands problèmes scientifiques et techniques abordés dans la recherche publique et industrielle. Ces simulations numériques requièrent l'utilisation des moyens de calcul les plus puissants. Le CERFACS est dirigé par un Conseil de Gérance dont les membres sont issus de chacun de ses actionnaires, il bénéficie par ailleurs des recommandations de son Conseil Scientifique. Le CERFACS compte sept actionnaires :

- Le CNES, Centre National d'Études Spatiales,
- EADS France, European Aeronautic and Defence Space company,
- EDF, Electricité De France,
- Météo-France,
- L'ONERA, centre français de recherche en aéronautique,
- SAFRAN, groupe international de haute technologie,
- TOTAL, multinationale du domaine de l'énergie.

En interne, le CERFACS est structuré en 5 équipes différentes : "Aviation et Environnement", "Modélisation du Climat et de son Changement Global", "Mécanique des Fluides Numérique", "Électromagnétisme et Acoustique", "Algorithmique Numérique Parallèle" et "Support Informatique". Durant mon stage j'ai principalement été en contact avec l'équipe CFD (*Computational Fluid Dynamics*) et l'équipe CSG (*Computing Support Group*). Les sous-sections suivantes détaillent ces deux groupes.

A L'équipe CFD aérodynamique

À mon arrivée, le 1er septempbre 2013, j'ai rejoint l'équipe CFD aérodynamique. Cette dernière est constituée de :

- Jean-François Boussuge : Chef d'équipe (maître de ce stage),
- Guillaume Puigt : Chercheur Sénior,
- Nadège Villedieu : Post Doctorante,
- Isabelle Marter : Stagiaire,
- Aurélien Genot : Stagiaire,

J'ai très souvent été en contact avec Nadège Villedieu, Guillaume Puigt et Jean-François Boussuge qui m'ont beaucoup aidé à comprendre le sujet sur lequel j'ai travaillé. Ils m'ont notamment permis de comprendre les bases physiques du code `JAGUAR` ainsi que son fonctionnement algorithmique. De plus, nous avons entretenu un dialogue constant afin de suivre l'évolution de mon stage et parfois de redéfinir la priorité des tâches à effectuer (voire en créer de nouvelles).

B L'équipe CSG

J'ai ensuite rencontré l'équipe CSG (Support Informatique) qui m'a aidé sur la partie technique. De plus, c'est cette équipe qui m'a permis d'accéder aux GPUs, CPUs et *Xeon Phi*. Cette dernière est constituée de :

– Nicolas Monnier : Directeur des Systèmes d'Information,
– Isabelle D'Ast : Ingénieure HPC,
– Gabriel Jonville : Ingénieur HPC,
– Gérard Dejean : Ingénieur Système,
– Fabrice Fleury : Ingénieur Système,
– Patrick Laporte : Ingénieur Réseau,
– Maurice Vidal : Technicien Informatique.

La vocation de cette équipe est, premièrement, d'apporter un support technique sur les outils mis à disposition puis, deuxièmement, d'assurer le bon fonctionnement des équipements informatiques (postes de travail, réseau, supercalculateurs, etc.). L'optimisation de code fait partie du support technique de CSG et c'est dans ce cadre qu'elle m'a apporté sont soutient. M^{me} Isabelle D'Ast, Ingénieure HPC, m'a très souvent aidé quand j'ai rencontré des problèmes techniques. Son expertise dans le domaine du HPC m'a été très précieuse. Elle a été une interlocutrice directe et m'a beaucoup conseillé sur les différents travaux que j'ai dû effectuer.

II.2 La simulation dans l'industrie

Les industriels ont compris que la simulation aux grandes échelles est un outil permettant de **se différencier de ses concurrents**. Dans un contexte de plus en plus contraint (par les normes de pollution et sonore), il est nécessaire de pouvoir calculer des écoulements turbulents, notamment pour pouvoir analyser et comprendre des phénomènes complexes. **Le coût des essais expérimentaux est rédhibitoire** lorsqu'il s'agit de faire une analyse paramétrique, alors que **le recours au calcul est maintenant à la portée de l'entreprise**.

C'est dans ce cadre que plusieurs projets industriels apparaissent : l'enjeu consiste en la capacité à tirer profit des plus gros supercalculateurs actuels ou à venir afin d'une part de diminuer autant que possible le temps de restitution des calculs et d'autre part de pouvoir traiter les applications les plus complexes possibles, ce qui demande des ressources de calcul et de mémoire très grandes.

II.3 Le projet `JAGUAR`

`JAGUAR` (*proJect of an Aerodynamic solver using General Unstructured grids And high ordeR schemes*) est un jeune code de dynamique des fluides développé au CERFACS par l'équipe CFD aérodynamique. Il utilise la méthode des Différences Spectrales (Sec I.5). Le projet est né en 2012 et est écrit en `Fortran 90`.

Ce choix a été motivé par plusieurs raisons :
– le `Fortran 90` est un langage compilé/performant et traditionnellement très utilisé en HPC (*High Performance Computing*),
– il est de plus haut niveau que le `C` : le code est généralement plus lisible (la syntaxe est relativement proche des notations mathématiques),
– `JAGUAR` est un code scientifique expérimental, il n'est donc pas question de se lancer dans des paradigmes complexes comme la programmation orientée objet.

L'application de la méthode SD pour les équations différentielles d'Euler se traduit par **l'utilisation d'un *stencil* d'ordre 1** : chaque élément a besoin d'informations provenant de

ses voisins directs (Fig. 9). Cet échange d'information intervient à l'étape 3 de la méthode SD : au moment où l'on utilise le solveur de RIEMANN.

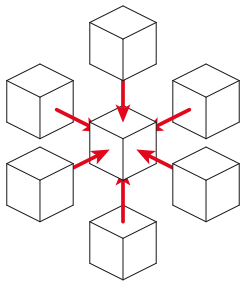

FIG. 9 – Exemple d'un élément et de ses voisins dans un *stencil* 3D

L'avantage des méthodes dites *High-Order* comme *Spectral Differences* est de proposer un grand nombre de calculs à l'intérieur de chaque élément : **la majorité des calculs sont locaux**. Cette méthode permet aussi de faire varier les degrés de liberté des éléments dans un même domaine. Le maillage est alors qualifié de multi p, où p est le degré d'un élément. Avec ce type de maillage, il est possible d'économiser du temps de calcul en diminuant le nombre de degrés de liberté des éléments que l'on ne souhaite pas observer avec précision.

De plus, **l'implémentation de la méthode SD s'adapte bien aux accélérateurs de calcul GPU** (*Graphical Process Unit*). Ces derniers ont été détournés de leur fonction première qui était de calculer des images. Au fur et à mesure des années, les GPUs ont montré un potentiel grandissant dans les applications massivement parallèles.

Plusieurs codes de CFD exploitent déjà la puissance de calcul des GPUs. Par exemple, P. CASTONGUAY a développé un solveur en trois dimensions pour les équations différentielles de NAVIER-STOKES (augmentation du modèle physique d'EULER avec la prise en compte du phénomène de viscosité) suivant une méthode proche de SD : *Flux Reconstruction* [4]. Le code fonctionne sur GPU et les performances sont nettement supérieures à celles des solveurs CPUs (*Central Processing Unit*) traditionnels. B. ZIMMERMAN, Z.J. WANG et M.R. VISBAL ont également développé leur solveur GPU suivant la même méthode que `JAGUAR` [5] (SD). Cependant ces codes ne permettent pas de calculer des maillages multi p. Le changement du degré en fonction des éléments est un réel obstacle sur GPU.

Dans un autre registre, l'Université de Cambridge a développé un code de CFD propriétaire pour les turbomachines : `Turbostream`. Ce dernier est aujourd'hui une référence dans le domaine et il utilise massivement les GPUs. `Turbostream` est un code abouti et compte parmis les meilleurs en terme de performance. G. PULLAN et T. BRANDVIK ont mis au point un *framework* capable de générer automatiquement le code de `Turbostream` pour GPU (et pour d'autres plateformes). L'article [6] décrit la méthode de génération du code ainsi que les résultats obtenus.

Dans `JAGUAR`, à mon arrivée, **le code n'était pas capable de traiter des maillages multi** p bien qu'une personne (Isabelle MARTER) travaillait déjà sur l'implémentation de cette fonctionnalité [7]. Une version parallèle à mémoire distribuée était fonctionnelle avec l'utilisation de la bibliothèque MPI (*Message Passing Interface*).

II.4 La technologie CUDA

Dans le monde des accélérateurs GPU, il y a principalement deux constructeurs : *AMD* (anciennement *ATI*) et *Nvidia*. Afin d'exploiter le potentiel des GPUs en matière de calcul, les constructeurs ont mis différents *framework* à disposition des développeurs : OpenCL (*Open Computing Language*) et CUDA (*Compute Unified Device Architecture*) sont les plus connus. OpenCL est un standard libre et utilisé par différents types de matériels (CPUs, GPUs, Xeon Phi, etc.). Les GPUs *AMD* et *Nvidia* sont compatibles avec cette norme. Cependant *Nvidia* développe son propre *framework* propriétaire depuis 2007 : CUDA. Ce dernier est uniquement disponible sur les GPUs *Nvidia* : il est très réputé et précurseur dans le monde du HPC. CUDA est très souvent en avance sur son concurrent OpenCL, il adopte très rapidement les nouvelles technologies proposées par les derniers GPUs. Le code `JAGUAR` a été développé en CUDA `Fortran` avec l'aide du guide de programmation de PGI [8] ainsi que du guide de programmation de *Nvidia* [9]. Aujourd'hui, en `Fortran`, seul le compilateur propriétaire de PGI permet de développer sur GPU. Ce compilateur est uniquement compatible avec CUDA et les GPUs *Nvidia*. Il n'existe pas de compilateur `Fortran` pour OpenCL.

II.5 L'architecture *Kepler*

A Rappels sur le fonctionnement et l'architecture générale des GPUs

Sur GPU, les traitements sont déportés vers ce que l'on appelle des "noyaux de calcul" (*kernels*). Ces derniers sont exécutés en simultané par plusieurs *threads*, le code est nativement SIMT (*Single Instruction Multiple Threads*). Sur le GPU comme sur le CPU il y a plusieurs types de mémoires :

- **la mémoire globale** : cette mémoire est lente mais de grande capacité (> 2 Go),
- **la mémoire partagée** : cette mémoire est partagée par un groupe de *threads*, elle est plus rapide que la mémoire globale mais sa capacité est inférieure à 64 Ko,
- **les registres** : ils sont propres à un *thread* et sont très rapides d'accès, cependant leur nombre est très limité : il varie entre 63 et 255 selon les modèles de GPU.

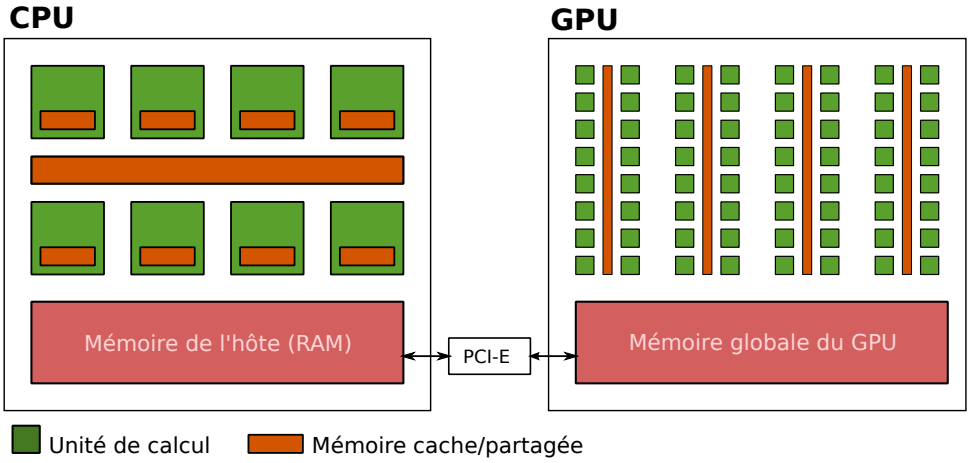

Fig. 10 – Architectures simplifiées d'un CPU et d'un GPU

La Fig. 10 présente les architectures simplifiées d'un CPU et d'un GPU. Sur GPU, il y a beaucoup plus d'unités de calcul que sur CPU. Cela a un impact sur la parallélisation du code : la granularité est différente. De plus, les transferts de données de la RAM vers la mémoire du GPU passe par le bus PCI-Express et ont un coût important : il faut essayer de minimiser ces transferts.

Dans `JAGUAR`, la stratégie a été de porter toutes les routines de la boucle de calcul principale sur GPU afin qu'il n'y ait plus aucun transfert de *buffers* pendant le calcul des solutions.

Logique d'exécution Les *threads* d'un noyau de calcul GPU sont organisés en grille de plusieurs blocs.

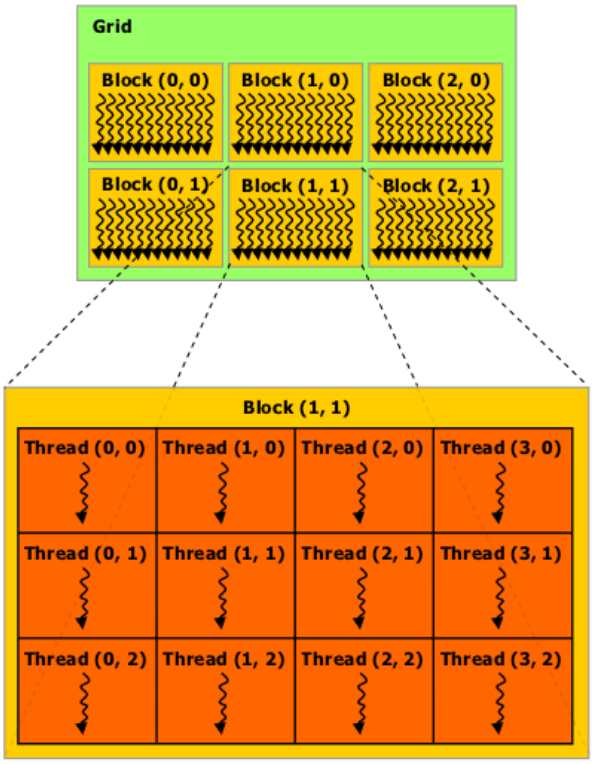

FIG. 11 – Exemple de grille 2D (2x3) avec blocs 2D (3x4)

La Fig. 11 illustre cette organisation : quand on exécute un *kernel*, il faut spécifier le nombre de blocs ainsi que la taille des blocs (nombre de *threads* par bloc). La grille représente la totalité des *threads* d'un noyau de calcul. Le bloc est un groupe de *threads* où les instructions vont être exécutées en même temps (parallélisme SIMD, *Single Instruction Multiple Data*). Sur les GPUs actuels, il est important de choisir des tailles de blocs qui sont multiples de 32 : cette spécificité vient du matériel qui ordonnance les blocs par groupe de 32 (appelé *warp*).

B Historique des architectures GPUs

Avant de rentrer dans les détails de l'architecture *Kepler*, la table 1 rappelle l'historique des architectures GPU *Nvidia* dédiées au calcul. Le terme *compute capability* est employé

par *Nvidia* pour désigner les possibilités matérielles de calcul des GPUs. Sans rentrer dans les détails de chaque architecture, voici les évolutions principales :

- de *Tesla* à *Fermi* : augmentation du nombre d'unités de calcul, possibilité d'exécuter plusieurs *kernels* en concurrence, apparition des unités de calcul doubles précisions, ajout des caches L1 et L2,
- de *Fermi* à *Kepler* : augmentation du nombre d'unités de calcul, partage d'un GPU entre plusieurs processus, création de *kernels* au sein d'un *kernel* (calcul adaptatif).

Année de sortie	**Architecture**	**Compute capability**	**Chipset GPU**
2006	*Tesla*	1.0, 1.1, 1.2	G80, G86, GT218, ...
2008	*Tesla*	1.3	GT200, GT200b
2010	*Fermi*	2.0, 2.1	GF100, GF110, GF104, ...
2012	*Kepler*	3.0	GK104, GK106, GK107, ...
2013	*Kepler*	3.5	GK110

TAB. 1 – Historique des architectures GPU *Nvidia*

C L'architecture *Kepler*

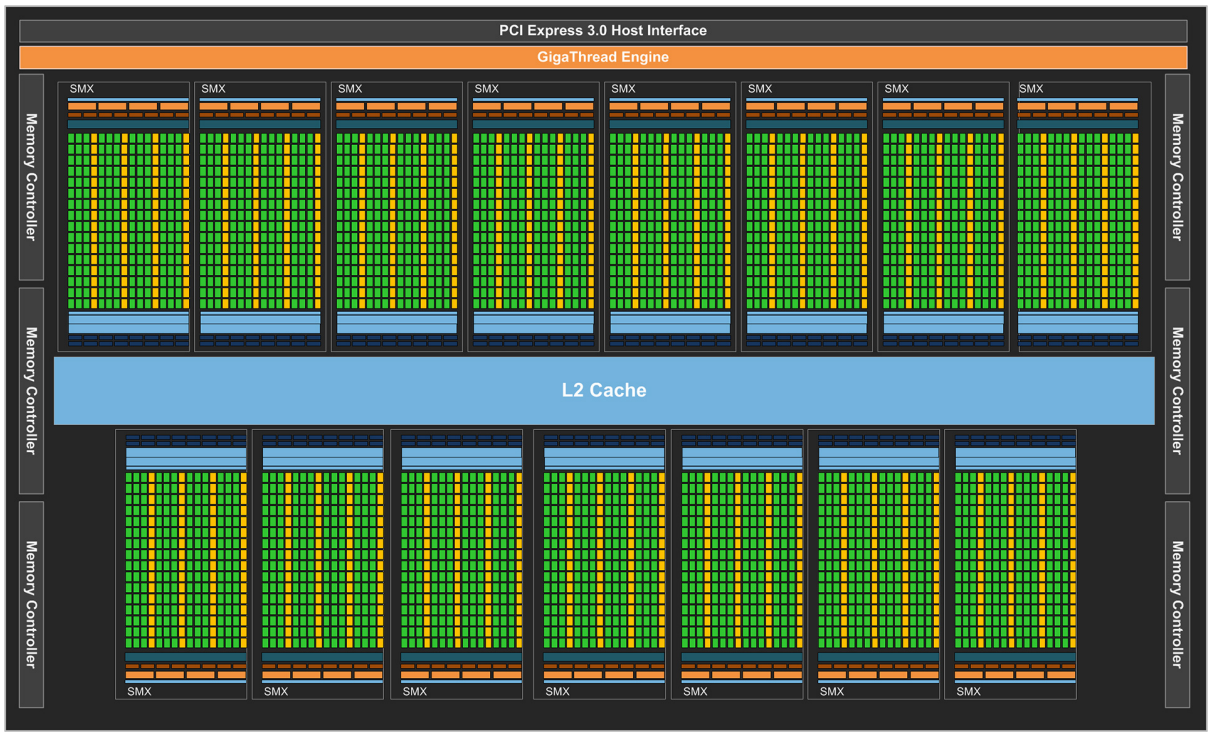

FIG. 12 – Répartition des unités de calcul dans les multiprocesseurs de flux (SMX) d'un GPU de type GK110

Kepler est la dernière architecture de *Nvidia*. La Fig. 12 illustre la répartition des unités de calcul dans les multiprocesseurs de flux (SMX, *Streaming Multiprocessor neXt*

generation). Les multiprocesseurs de flux peuvent être apparentés à un coeur CPU alors que les unités de calcul (carrés verts et jaunes) sont assez proches des unités vectorielles d'un CPU. La couleur verte représente les unités de calcul simples précisions et la couleur jaune représente les unités de calcul doubles précisions. Les SMX partagent un cache L2 et, de manière générale, chaque bloc d'une grille de calcul est envoyé à un SMX. S'il y a plus de blocs que de SMX, alors les SMX prennent plusieurs blocs en charge.

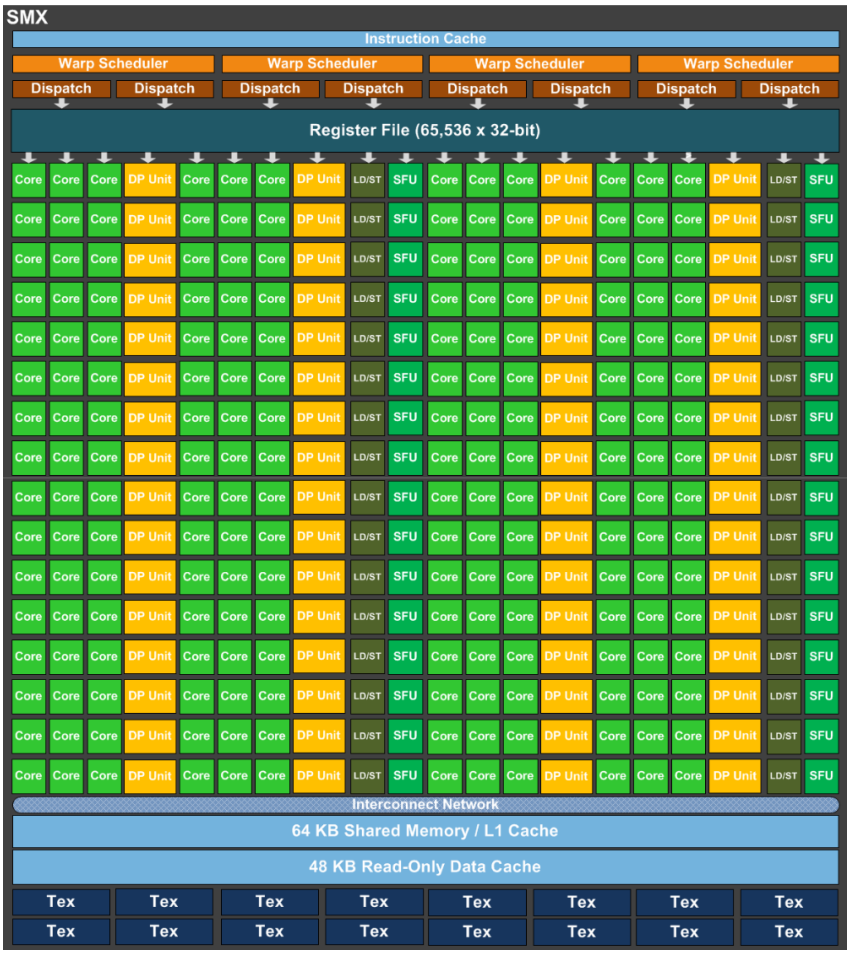

FIG. 13 – Détails d'un SMX

La Fig. 13 détaille l'architecture matérielle d'un multiprocesseur de flux. En plus des unités de calcul traditionnelles, il y a des unités SFUs (*Special Function Units*) dédiées aux fonctions spéciales (sinus, cosinus, racine carré, exponentielle, etc.). Pour résumer, dans un multiprocesseur de flux il y a 192 unités de calcul simples précisions, 64 unités de calcul doubles précisions et 32 unités de calcul spéciales. Le placement des *threads* sur les unités de calcul est assuré par 4 *Warp Scheduler* ce qui signifie qu'il peut potentiellement y avoir 4 noyaux de calculs différents qui s'exécutent en simultané sur un SMX. Enfin, pour ce qui est de la mémoire, il y a :

- 256 Ko dédiés aux registres (*Register File*), soit 65 536 variables de 32bit,
- 64 Ko de mémoire partagée (*Shared Memory*, une partie de cette mémoire est dédiée au cache L1),
- 48 Ko de mémoire cache en lecture seule (*Read-Only Data Cache*).

II.6 Organisation du travail

Le stage de fin d'étude s'est déroulé sur une durée de 5 mois : du 1er septembre 2013 au 31 janvier 2014. À mon arrivée, nous avons défini les objectifs principaux :

- **améliorer les performances de `JAGUAR`**,
- **porter le code sur GPU**.

Nous n'avons pas plannifié de tâches à l'avance car il aurait été trop difficile d'estimer leur durée ainsi que leurs implications. À la place, nous avons été en constante discussion sur les évolutions apportées au code afin de s'assurer que les décisions prises soient approuvées par l'ensemble de l'équipe. Le premier mois de travail a essentiellement consisté en la compréhension du code initial et en la création de sous projets élémentaires dédiés à des tests sur GPU. Cette première phase a permis de découvrir que certains algorithmes étaient perfectibles. Dans un second temps, le deuxième mois, nous avons modifié les structures de données ainsi que plusieurs traitements associés. Le troisième mois a été exclusivement réservé au portage du code complet sur GPU. La quatrième mois, nous nous sommes attachés à l'optimisation du code GPU et à la version multi GPU. Enfin, le dernier mois a permis de tester, corriger, mesurer, et expliquer les précédents travaux (cf. Fig. 14).

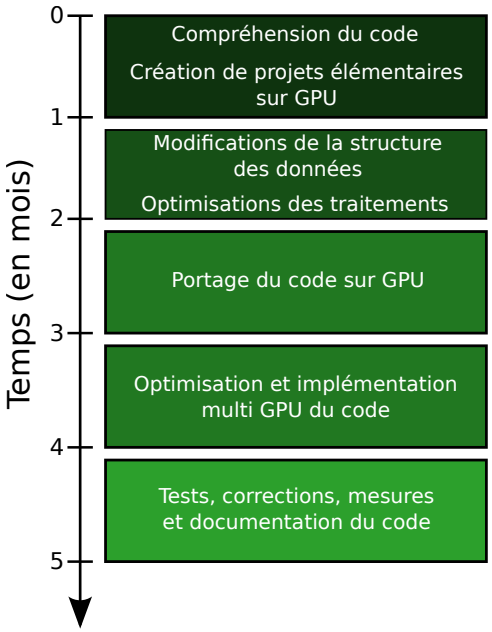

Fig. 14 – Déroulement du stage

Les développements sur le code `JAGUAR` ont été faits à l'aide du gestionnaire de version Git [4]. Dans ce cadre, les modifications ont d'abord été ajoutées sur une branche séparée puis, à la fin du stage, cette branche a été fusionnée avec la branche principale.

La suite de ce mémoire présente les différents travaux effectués dans l'objectif de rendre `JAGUAR` **compatible et performant avec les accélérateurs de calcul de type GPU**.

4. Git : `http://git-scm.com`

III Modifications algorithmiques

Cette section présente les différentes modifications algorithmiques apportées au code JAGUAR CPU dans l'objectif de le rendre plus performant et plus facile à porter sur GPU.

III.1 Implémentation de la méthode des Différences Spectrales

Dans les algorithmes 1 et 2, S, SB, SI, SF, FF, F sont des vecteurs flottants. Les vecteurs S, SB contiennent les solutions du maillage complet. Le vecteur SF contient les solutions extrapolées sur les faces. Le vecteur F contient les flux du maillage complet. Le vecteur FF contient les flux des faces. Le vecteur SI contient la divergence de F ($\texttt{SI} = \nabla \cdot \texttt{F}$). La variable dt est un scalaire flottant qui contient le pas de temps de la simulation. Enfin, les variables it, lastIt, iStep et rkSteps sont de type entier. La variable lastIt représente le nombre total d'itérations de la simulation. Les variables rkSteps et RKAlpha sont propres à la méthode de Runge-Kutta. Cette dernière ne sera pas détaillée dans ce document cependant il est bon de savoir qu'elle permet d'approximer des solutions d'équations différentielles. La thèse de A. Fosso [10] explique cette méthode en détail.

Algorithm 1 Spectral Differences Initial [SDI]

```
lastIt  ← readLastIteration()
rkSteps ← readRKSteps()
RKAlpha ← initRKAlpha( in::rkSteps )
S  ← readInitSolutions()
it ← 1
while (it ≤ lastIt) do
  dt ← computeTimeStep( in::S )
  SB ← S
  SI ← 0
  iStep ← 1
  while (iStep ≤ rkSteps) do
    SF ← scatterFlux( in::S )
    FF ← RiemannSolver( in::SF )
    F  ← gatherFlux( in::FF )
    SI ← computeFluxDivergence( in::S, inout::F )
    S  ← dt × SB × SI × RKAlpha[iStep]  ou updateSP
    iStep ← iStep + 1
  end while
  it ← it + 1
end while
```

L'algorithme 1 traduit l'implémentation de la méthode *Spectral Differences* dans JAGUAR. La routine computeTimeStep permet de calculer le pas de temps dt en fonction des solutions du maillage (le pas de temps est variable). La routine scatterFlux extrapole les points solutions sur les faces et les stocke dans le buffer SF. Le solveur de Riemann (RiemannSolver) calcule les flux sur les faces dans FF à partir des solutions extrapolées SF. C'est dans cette routine qu'intervient la notion de *stencil* : le

calcul des flux d'un élément sur les faces nécessite la connaissance des solutions des éléments voisins. La routine `gatherFlux` rassemble les flux des faces dans le buffer `F`. Enfin, `computeFluxDivergence` calcule les flux à l'intérieur des éléments puis détermine l'incrément des solutions (calcul de la divergence des flux). Deux étapes sont nécessaires au calcul des flux à l'intérieur : l'extrapolation des points solutions `S` puis le calcul des flux dans `F`. Les instructions $\texttt{dt} \times \texttt{SB} \times \texttt{SI} \times \texttt{RKAlpha[iStep]}$ permettent de mettre la solution à jour. Ces dernières seront appelées `updateSP` dans la suite du rapport.

Le problème de l'implémentation Alg. 1 est que l'extrapolation des points solutions est faite dans deux routines différentes (`scatterFlux` et `computeFluxDivergence`). Ce n'est pas nécessaire puisque le calcul est identique sur les faces et à l'intérieur. Nous avons alors rassemblé les extrapolations dans une même routine. Ce choix a été motivé par deux raisons : la simplification du code initial et l'amélioration des performances : le calcul global étant plus homogène, les mécanismes de cache et de prédiction de branchement du processeur sont plus efficaces. L'algorithme 2 adopte la nouvelle implémentation.

Algorithm 2 Spectral Differences [SD]

```
lastIt  ← readLastIteration()
rkSteps ← readRKSteps()
RKAlpha ← initRKAlpha( in::rkSteps )
S  ← readInitSolutions()
it ← 1
while (it ≤ lastIt) do
  dt ← computeTimeStep( in::S )
  SB ← S
  SI ← 0
  iStep ← 1
  while (iStep ≤ rkSteps) do
    F  ← computeInternFlux( in::S )
    SF ← scatterFlux( in::F )
    FF ← RiemannSolver( in::SF )
    F  ← gatherFlux( in::FF )
    SI ← computeFluxDivergence( in::F )
    S  ← dt × SB × SI × RKAlpha[iStep]  ou updateSP
    iStep ← iStep + 1
  end while
  it ← it + 1
end while
```

La nouvelle routine `computeInternFlux` extrapole toutes les solutions (internes et externes) dans le buffer `F`. Si les solutions sont internes à un élément, alors les flux correspondants sont calculés. Nous avons choisi de calculer directement les flux internes dans `computeInternFlux` pour des questions de performances : les solutions extrapolées étant dans le cache du processeur, c'est le meilleur moment pour calculer les flux. Dans l'algorithme Alg. 2, `scatterFlux` se contente d'extraire les solutions extrapolées sur les faces pour les sauver dans le buffer `SF`. La routine `computeFluxDivergence` dérive les flux `F` dans les solutions `SI`.

III.2 Méthodes de calcul numérique

L'analyse du temps de calcul des routines de `JAGUAR` (Alg. 2) montre que :
- `computeInternFlux` occupe **40%** du temps de calcul,
- `computeFluxDivergence` occupe **25%** du temps de calcul,
- `scatterFlux` et `gatherFlux` occupent **15%** du temps de calcul,
- `RiemannSolver` occupe **10%** du temps de calcul.

Pour résumer, environ **50%** du temps de calcul est passé dans l'extrapolation des points solutions et dans la dérivation des points flux (ce pourcentage est une estimation car il n'y a pas que des extrapolations dans `computeInternFlux`). Partant de ce constat, nous avons principalement modifié le code pour qu'il soit performant sur ces traitements.

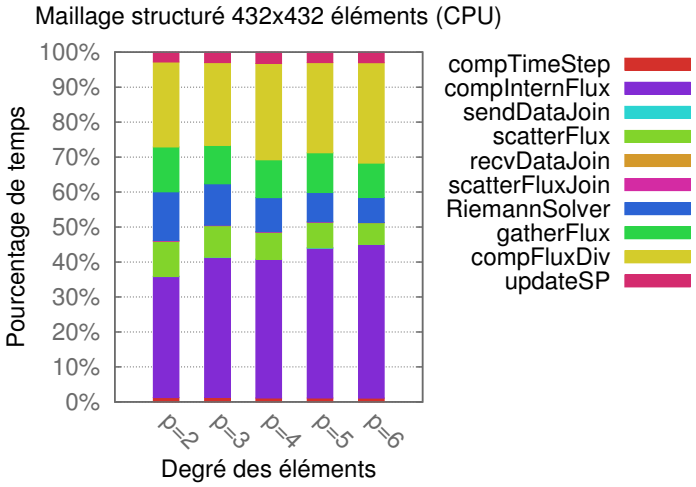

FIG. 15 – Pourcentage de temps occupé par routine (CPU)

La Fig. 15 propose d'observer le pourcentage de temps occupé par routine en fonction du degré de précision des éléments. La répartition des temps de calcul est identique en deux ou en trois dimensions pour la version séquentielle. Les routines dont le nom se termine par `Join` seront abordées plus tard et ne nous intéressent pas dans cette section. Attention, les temps qui ont permis de réaliser cet histogramme proviennent de la dernière version de `JAGUAR` (avec toutes les optimisations). Ce dernier est à titre indicatif et permet de présenter grossièrement la répartition des temps de calcul dans le code.

A Extrapolation des points solutions (`computeInternFlux`)

Le nombre de points où la solutions est extrapolée est identique au nombre de points flux. La figure 16 présente la répartition des points solutions (ronds rouges) et de la solution extrapolée (carrés verts) pour un élément en deux dimensions et de degré 1. Le calcul des solutions extrapolées **E1**, **E2**, et **E3** nécessite les solutions **S1** et **S2** alors que le calcul de **E4**, **E5** et **E6** nécessite **S3** et **S4**. **E1**, **E2**, **E3**, **E4**, **E5** et **E6** sont les solutions extrapolées suivant l'axe des ordonnées. De même, pour calculer **E7**, **E8** et **E9** il faut connaître **S1** et **S3**, etc. **E7**, **E8**, **E9**, **E10**, **E11** et **E12** sont les solutions extrapolées suivant l'axe des abscisses.

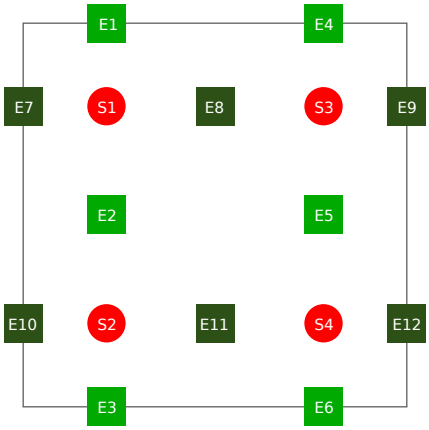

FIG. 16 – Points solutions (cercles) et points solutions extrapolées (carrés) d'un élément de degré 1 en 2D

Pour extrapoler les points solutions, on utilise des coefficients d'extrapolation appelés `ExtraCoefs` : ils sont précalculés à l'initialisation du programme. Afin d'améliorer la lisibilité et les performances du code, nous avons formalisé le calcul des extrapolations en une combinaison de produits matriciels.

Extrapolation des points solutions de l'élément Fig. 16 selon l'axe des ordonnées :

$$E_j = \texttt{ExtraCoefs} \times S \Leftrightarrow \begin{pmatrix} E1 & E4 \\ E2 & E5 \\ E3 & E6 \end{pmatrix} = \begin{pmatrix} C1 & C2 \\ C3 & C4 \\ C5 & C6 \end{pmatrix} \times \begin{pmatrix} S1 & S3 \\ S2 & S4 \end{pmatrix}$$

Extrapolation des points solutions de l'élément Fig. 16 selon l'axe des abscisses :

$$E_i = \texttt{ExtraCoefs} \times S^t \Leftrightarrow \begin{pmatrix} E7 & E10 \\ E8 & E11 \\ E9 & E12 \end{pmatrix} = \begin{pmatrix} C1 & C2 \\ C3 & C4 \\ C5 & C6 \end{pmatrix} \times \begin{pmatrix} S1 & S2 \\ S3 & S4 \end{pmatrix}$$

B Dérivation des points flux (`computeFluxDivergence`)

La dérivation des flux aux points solutions est un abus de langage : en réalité, les flux sont interpolés puis dérivés dans les points solutions. Les mouvements de données engendrés par la dérivation des flux sont inversés par rapport à l'extrapolation. Au lieu de calculer des points solutions vers la solution extrapolée, on calcule des points flux vers les points solutions. La figure 17 présente l'organisation des points solutions et des points flux. Nous avons formalisé la dérivation en une combinaison de produits matriciels (comme pour l'extrapolation).

Pour dériver les points flux dans les points solutions, on utilise des coefficients de dérivation (`DerivCoefs`) précalculés à l'initialisation du programme. Lors du calcul, on dérive les points flux suivant les directions dans un buffer solution unique. Il faut donc conserver les valeurs des solutions entre les différentes dérivations.

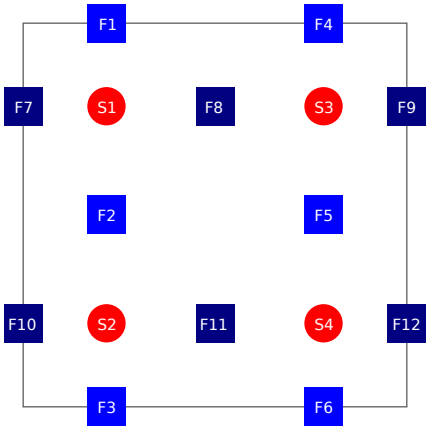

FIG. 17 – Points solutions (cercles) et points flux (carrés) d'un élément de degré 1 en 2D

Dérivation des points flux de l'élément Fig. 17 selon l'axe des ordonnées :

$$S = S + \texttt{DerivCoefs} \times F_j$$
$$\Leftrightarrow$$
$$\begin{pmatrix} S1 & S3 \\ S2 & S4 \end{pmatrix} = \begin{pmatrix} S1 & S3 \\ S2 & S4 \end{pmatrix} + \begin{pmatrix} C1 & C2 & C3 \\ C4 & C5 & C6 \end{pmatrix} \times \begin{pmatrix} F1 & F4 \\ F2 & F5 \\ F3 & F6 \end{pmatrix}$$

Dérivation des points flux de l'élément Fig. 17 selon l'axe des abscisses :

$$S^t = S^t + \texttt{DerivCoefs} \times F_i$$
$$\Leftrightarrow$$
$$\begin{pmatrix} S1 & S2 \\ S3 & S4 \end{pmatrix} = \begin{pmatrix} S1 & S2 \\ S3 & S4 \end{pmatrix} + \begin{pmatrix} C1 & C2 & C3 \\ C4 & C5 & C6 \end{pmatrix} \times \begin{pmatrix} F7 & F10 \\ F8 & F11 \\ F9 & F12 \end{pmatrix}$$

C Produits matriciels et performances

L'écriture de l'extrapolation et de la dérivation sous forme matricielle permet aux compilateurs de mieux optimiser le code. En effet, ces derniers sont très efficaces pour reconnaître ce type de problème. De plus, cette nouvelle formalisation permet de s'abstraire du problème physique pour le transformer en un problème d'algèbre linéaire. Traditionnellement, dans le domaine du HPC, on préfère traiter des problèmes généralistes (problèmes connus). Cette approche rend possible la réutilisation des travaux effectués dans un domaine pour d'autres.

En termes de performances, dans `JAGUAR`, l'écriture matricielle nous a permis de mieux organiser les données de façon à ce que les accès mémoire soient parfaitement contigus. De ce fait, la vectorisation [5] du code est plus efficace : il y a beaucoup moins de problèmes de dispersion et de regroupement des données (*scatter* et *gather*).

Enfin, nous n'avons pas utilisé de bibliothèque BLAS (*Basic Linear Algebra Subprograms*) car les matrices calculées dans `JAGUAR` sont de petites tailles (2×2 à 10×10). Le

5. Vectorisation : http://fr.wikipedia.org/wiki/Vectorisation_(informatique)

temps d'appel d'une routine BLAS est trop coûteux par rapport à l'exécution de cette même routine. Nous avons préféré réécrire les produits matriciels.

D Calcul en trois dimensions

Les exemples précédents ne détaillent pas le cas 3D car il est plus complexe à expliquer et à schématiser. Ce dernier se décompose en une succession de cas 2D. Les traitements matriciels sont effectués dans plusieurs plans et suivant les trois directions (i, j, k). Le nombre de plans est déterminé en fonction du degré des éléments.

III.3 Réorganisation de la mémoire

Afin d'améliorer les performances des produits matriciels (extrapolation et dérivation), nous avons réorganisé la mémoire dans `JAGUAR`. Avant de détailler ces modifications, il faut préciser que les produits matriciels sont effectués 5 fois par solution. Cela correspond aux 5 variables conservatives $(\rho, \rho u, \rho v, \rho w, \rho E)$ (cf.Sec. I.4). Les coefficients d'extrapolation/dérivation restent identiques quelle que soit la variable conservative.

L'exemple suivant présente l'extrapolation des solutions de l'élément Fig. 16 suivant l'axe des ordonnées (le numéro de la variable conservative est noté en exposant) :

$$E_j^1 = \texttt{ExtraCoefs} \times S^1 \Leftrightarrow \begin{pmatrix} E1^1 & E4^1 \\ E2^1 & E5^1 \\ E3^1 & E6^1 \end{pmatrix} = \begin{pmatrix} C1 & C2 \\ C3 & C4 \\ C5 & C6 \end{pmatrix} \times \begin{pmatrix} S1^1 & S3^1 \\ S2^1 & S4^1 \end{pmatrix}$$

$$E_j^2 = \texttt{ExtraCoefs} \times S^2 \Leftrightarrow \begin{pmatrix} E1^2 & E4^2 \\ E2^2 & E5^2 \\ E3^2 & E6^2 \end{pmatrix} = \begin{pmatrix} C1 & C2 \\ C3 & C4 \\ C5 & C6 \end{pmatrix} \times \begin{pmatrix} S1^2 & S3^2 \\ S2^2 & S4^2 \end{pmatrix}$$

$$E_j^3 = \texttt{ExtraCoefs} \times S^3 \Leftrightarrow \begin{pmatrix} E1^3 & E4^3 \\ E2^3 & E5^3 \\ E3^3 & E6^3 \end{pmatrix} = \begin{pmatrix} C1 & C2 \\ C3 & C4 \\ C5 & C6 \end{pmatrix} \times \begin{pmatrix} S1^3 & S3^3 \\ S2^3 & S4^3 \end{pmatrix}$$

$$E_j^4 = \texttt{ExtraCoefs} \times S^4 \Leftrightarrow \begin{pmatrix} E1^4 & E4^4 \\ E2^4 & E5^4 \\ E3^4 & E6^4 \end{pmatrix} = \begin{pmatrix} C1 & C2 \\ C3 & C4 \\ C5 & C6 \end{pmatrix} \times \begin{pmatrix} S1^4 & S3^4 \\ S2^4 & S4^4 \end{pmatrix}$$

$$E_j^5 = \texttt{ExtraCoefs} \times S^5 \Leftrightarrow \begin{pmatrix} E1^5 & E4^5 \\ E2^5 & E5^5 \\ E3^5 & E6^5 \end{pmatrix} = \begin{pmatrix} C1 & C2 \\ C3 & C4 \\ C5 & C6 \end{pmatrix} \times \begin{pmatrix} S1^5 & S3^5 \\ S2^5 & S4^5 \end{pmatrix}$$

Le principe est le même pour le calcul des divergences.

Dans les sous sections suivantes (Sec. A et Sec. B), on considère un maillage multi p de deux éléments en 2D. Le premier élément du maillage est de degré 1 (Fig. 18) et le deuxième élément est de degré 2 (Fig. 19).

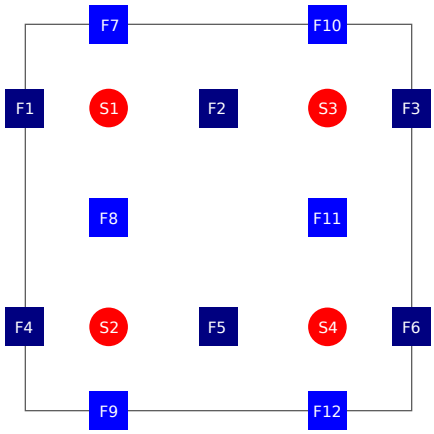

FIG. 18 – Points solutions (cercles) et points flux (carrés), élément de degré 1 (2D)

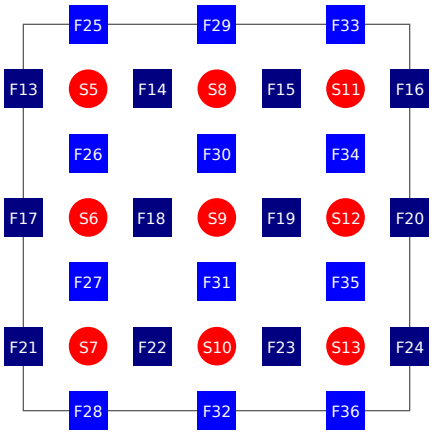

FIG. 19 – Points solutions (cercles) et points flux (carrés), élément de degré 2 (2D)

A Réorganisation des solutions

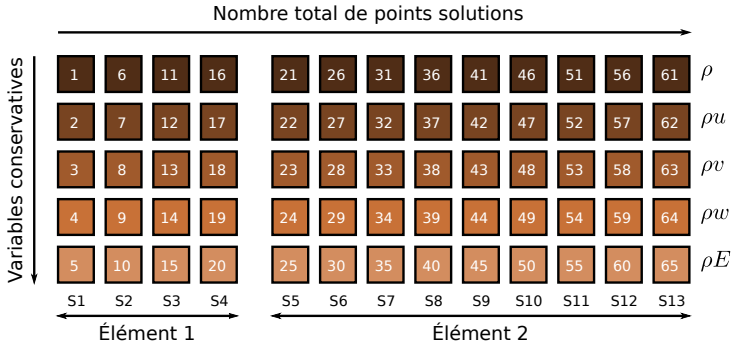

FIG. 20 – Organisation mémoire des solutions pour un maillage 2D de deux éléments (l'élément 1 est de degré 1 (cf. Fig. 18) et l'élément 2 est de degré 2 (cf. Fig. 19))

La Fig. 20 présente l'organisation initiale des solutions en mémoire. Les données sont stockées par colonne comme le montre la numérotation des cases mémoires. Chaque colonne contient les 5 variables conservatives d'une solution. En `Fortran`, l'accès à la quatrième variable conservative de la septième solution s'écrit naturellement :

```
tmp = Solutions(4, 7).
```

Pour chaque élément, la mémoire est accédée en 5 étapes (selon les 5 variables conservatives). Cela revient à faire des accès par ligne alors que les données sont stockées par colonne (`Fortran`). Les matrices suivantes représentent les accès à la mémoire pour les 5 produits matriciels (élément 1 Fig. 20) :

$$\left[\begin{pmatrix} 1 & 11 \\ 6 & 16 \end{pmatrix}, \begin{pmatrix} 2 & 12 \\ 7 & 17 \end{pmatrix}, \begin{pmatrix} 3 & 13 \\ 8 & 18 \end{pmatrix}, \begin{pmatrix} 4 & 14 \\ 9 & 19 \end{pmatrix}, \begin{pmatrix} 5 & 15 \\ 10 & 20 \end{pmatrix}\right]$$

On remarque qu'il y a un saut de 5 entre chaque accès. Même si le saut est régulier, cette organisation ne permet pas d'atteindre les performances optimales.

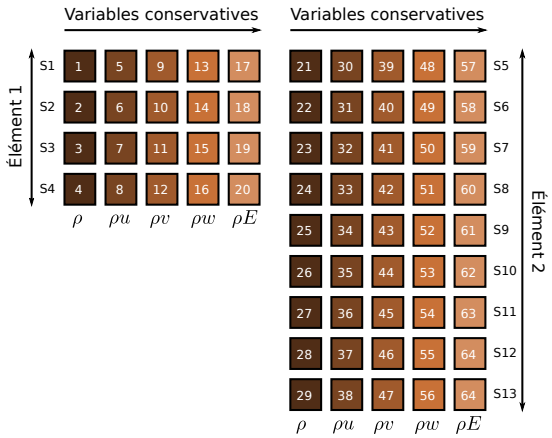

FIG. 21 – Nouvelle organisation mémoire des solutions pour un maillage 2D de deux éléments (l'élément 1 est de degré 1 (cf. Fig. 18) et l'élément 2 est de degré 2 (cf. Fig. 19))

La Fig. 21 propose la nouvelle organisation des solutions en mémoire. Les variables conservatives de même ordre et d'un même élément sont stockées les unes après les autres. Les solutions d'un élément sont stockés par colonne. Pour l'élément 1, les accès aux 5 matrices sont parfaitement contigus :

$$\left[\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}, \begin{pmatrix} 5 & 7 \\ 6 & 8 \end{pmatrix}, \begin{pmatrix} 9 & 11 \\ 10 & 12 \end{pmatrix}, \begin{pmatrix} 13 & 15 \\ 14 & 16 \end{pmatrix}, \begin{pmatrix} 17 & 19 \\ 18 & 20 \end{pmatrix}\right]$$

Cette réorganisation est très bénéfique aux accélérateurs de calcul car ils sont très sensibles aux accès mémoire.

En contrepartie, en `Fortran`, la syntaxe permettant d'accéder aux données est moins simple qu'avec l'organisation initiale. Il n'est plus possible de déclarer un tableau en deux dimensions et d'y accéder avec deux indices différents. Le code suivant permet d'accéder à la quatrième variable conservative de la septième solution (la septième solution globale correspond à la troisième solution de l'élément 2) :

```
1 integer nSolPElt1 = 4 ! nombre de solutions de l'element 1
2 integer nSolPElt2 = 9 ! nombre de solutions de l'element 2
3 integer offElt    = 0 ! saut memoire de positionnement sur l'element 2
4 integer nVars     = 5 ! nombre de variables conservatives
5 real    tmp           ! variable de stockage temporaire
6
7 offElt = offElt + nSolPElt1 * nVars
8 tmp    = Solutions(offElt + 3 + 4 * nSolPElt2)
```

B Réorganisation des flux

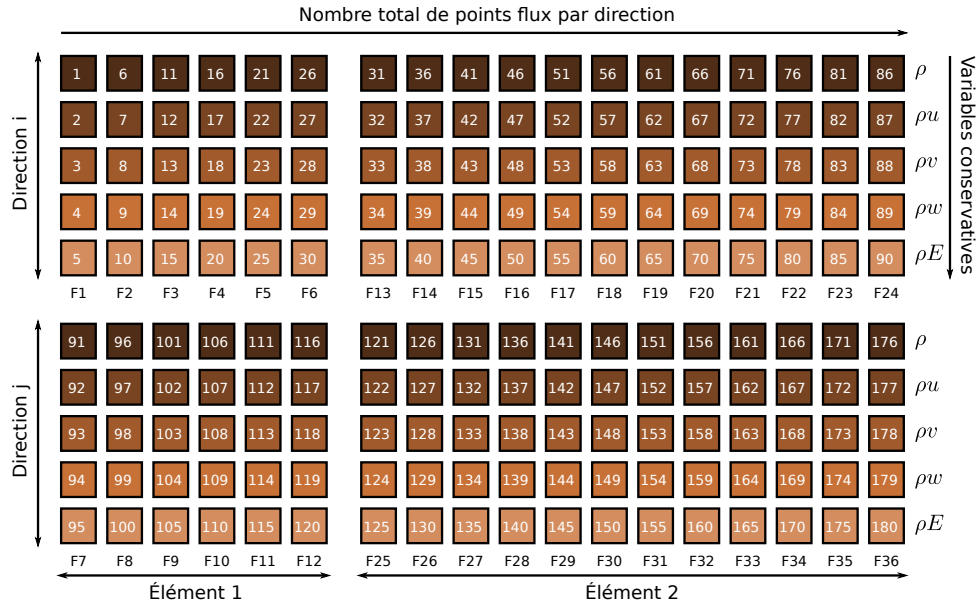

Fig. 22 – Organisation mémoire des flux pour un maillage 2D de deux éléments (l'élément 1 est de degré 1 (cf. Fig. 18) et l'élément 2 est de degré 2 (cf. Fig. 19))

La Fig. 22 présente l'organisation initiale des flux en mémoire. Les données sont stockées par colonne comme le montre la numérotation des cases mémoires. Chaque colonne

contient les 5 variables d'un flux. À la différence des solutions, les flux sont stockés par direction.

En `Fortran`, les flux sont représentés par un tableau de rang 3 : le premier rang pour les variables, le deuxième pour les flux et le troisième pour les directions. L'accès à la quatrième variable du septième flux suivant la deuxième direction s'écrit :

```
tmp = Flux(4, 7, 2).
```

Comme pour le stockage initial des solutions, le stockage initial des flux ne permet pas d'avoir des accès contigus lors du calcul des produits matriciels.

La Fig. 23 montre la nouvelle organisation des flux. Cette dernière permet des accès mémoire contigus lors du calcul des produits matriciels.

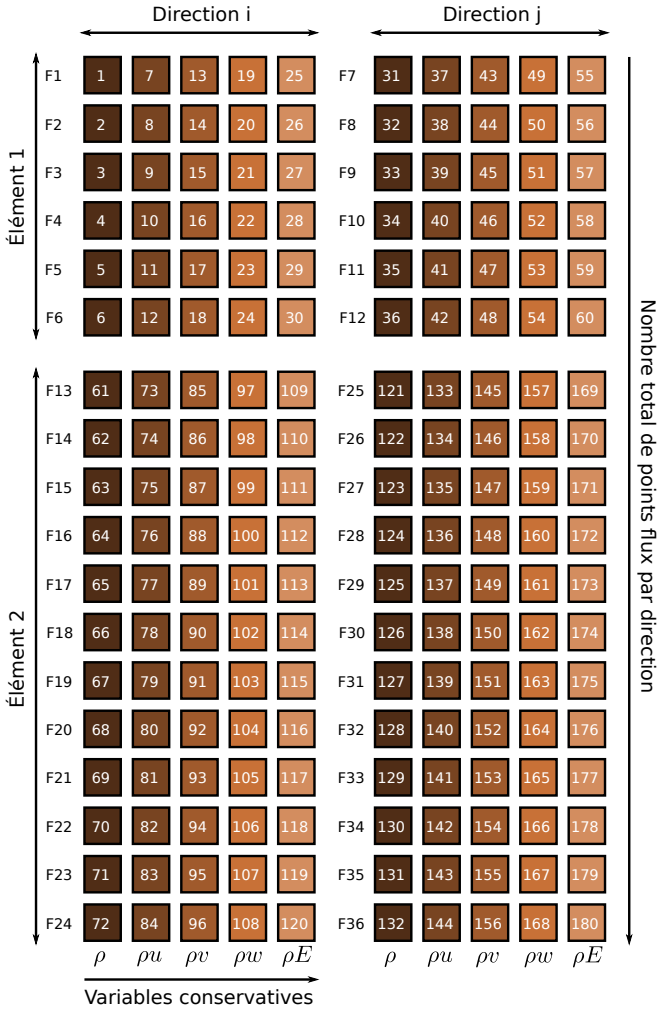

Fig. 23 – Nouvelle organisation mémoire des flux pour un maillage 2D de deux éléments (l'élément 1 est de degré 1 (cf. Fig. 18) et l'élément 2 est de degré 2 (cf. Fig. 19))

Cependant, comme pour les solutions, ce stockage est moins simple à utiliser. Le code suivant permet d'accéder à la quatrième variable du septième flux suivant la deuxième direction (le septième flux global correspond au premier flux de l'élément 2) :

```
integer nFluxPElt1 = 6  ! nb. de flux de l'element 1 (par direction)
integer nFluxPElt2 = 12 ! nb. de flux de l'element 2 (par direction)
integer offElt     = 0  ! saut memoire de positionnement sur l'element 2
integer nVars      = 5  ! nombre de variables conservatives
integer dim        = 2  ! nombre de dimensions
real    tmp             ! variable de stockage temporaire

offElt = offElt + nFluxPElt1 * nVars * dim
tmp    = Flux(offElt + 1              +        ! 1er flux de l'element 2
                       4 * nFluxPElt2 +        ! 4eme variable
                       2 * nFluxPElt2 * nVars) ! 2eme direction
```

C Le cas en trois dimensions

Le stockage des solutions en 3D est identique au stockage 2D. Pour un élément de degré fixé, les solutions sont stockées les unes après les autres. En 3D, il y a juste plus de solutions.

Pour les flux, il suffit de prendre en compte la direction k. Les flux d'un élément sont alors regroupés suivant les trois directions (i, j, k) : en premier les flux selon i, en second les flux selon j et en dernier les flux selon k.

III.4 Algorithme glouton

Afin de rendre les routines `computeInternFlux` et `computeFluxDivergence` plus modulaires et plus efficaces, nous avons regroupé les traitements des éléments de même degré dans des tâches (cf. Alg. 3). Les composantes de l'algorithme sont :

- `nTotalElt` : le nombre total d'éléments du maillage,
- `P` : un vecteur contenant les degrés des éléments du maillage,
- `p` : le degré du groupe d'élément courant,
- `iStartElt` : le premier élément du groupe,
- `iStopElt` : le dernier élément du groupe,
- `executeRegularTask` : une routine de calcul homogène.

Algorithm 3 Lanceur Glouton de Tâches Régulières [LGTR]

Require: `nTotalElt` > 0, `init(P)`, `sizeof(P)` $=$ `nTotalElt`
  `iStartElt` $\leftarrow$ `iStopElt` $\leftarrow 1$
  while (`iStartElt` $\leq$ `nTotalElt`) **do**
    `p` $\leftarrow$ `P[iStartElt]`
    while (`iStopElt` $+ 1 \leq$ `nTotalElt`) $\wedge$ (`P[iStopElt` $+ 1$`]` $=$ `p`) **do**
      `iStopElt` $\leftarrow$ `iStopElt` $+ 1$
    end while
    `executeRegularTask( in::p, in::iStartElt, in::iStopElt )`
    `iStartElt` $\leftarrow$ `iStopElt` $\leftarrow$ `iStopElt` $+ 1$
  end while

La Fig. 24 illustre l'affectation des éléments aux différentes tâches dans un maillage multi p en deux dimensions. Les éléments (grands carrés bleus) sont stockés par colonne. Plus un élément est foncé plus son degré p est élevé. Les petits carrés de couleurs vives montrent l'organisation des tâches : plusieurs petits carrés de même couleur qui se suivent sont apparentés à une même tâche.

A Calibrage des tâches et parallélisme

L'algorithme glouton Alg. 3 ne permet pas d'améliorer les performances directement. Cependant nous l'avons modifié pour mieux utiliser les caches du processeur (technique de *cache blocking*). La nouvelle implémentation subdivise les tâches en sous-tâches de tailles adaptées au cache (calibrage des tâches). Cette optimisation est particulièrement efficace dans la routine `computeInternFlux` (cf. Sec. III.1). L'extrapolation et le calcul des flux sont réunis dans une même tâche. Pour rappel, l'extrapolation est effectuée en premier et le calcul des flux est effectué en second. La taille des données utilisées dans les sous-tâches ne dépasse pas la taille du cache L2 du processeur. De cette façon, les solutions extrapolées sont encore dans le cache pour le calcul des flux.

Enfin, cette méthode permet de lancer des tâches dont les temps de calcul sont à peu près identiques. Cette amélioration de Alg. 3 permet une meilleure répartition de la charge sur différents *threads*. Chaque tâche travaille sur des données différentes des autres tâches. Cela permet d'éviter les problèmes liés au partage des données : chaque *thread* exécute des tâches indépendantes. Lors de nos tests, le parallélisme de tâche s'est montré bien plus *scalable* que le parallélisme SIMD (*Single Instruction Multiple Data*)

traditionnel. Il s'est aussi montré performant sur les architectures NUMA (*Non Uniform Memory Access*).

La Fig. 25 présente un exemple d'affectation des éléments aux différentes sous-tâches en considérant qu'il faut :

- moins de 5 éléments bleus clairs pour tenir dans le cache,
- moins de 3 éléments bleus pour tenir dans le cache,
- moins de 2 éléments bleus foncés pour tenir dans le cache.

Il est bon de rappeler que plus la couleur d'une cellule est foncée, plus le degré est élevé et plus la quantité de travail à effectuer est importante.

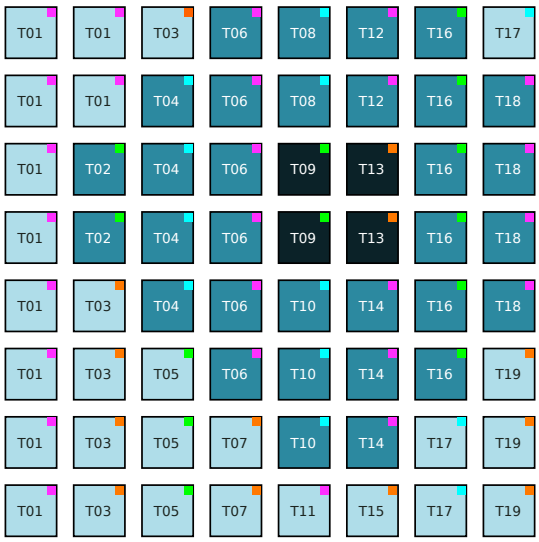

FIG. 24 – Affectation des éléments aux différentes tâches dans un maillage 2D multi p suivant l'algorithme Alg. 3

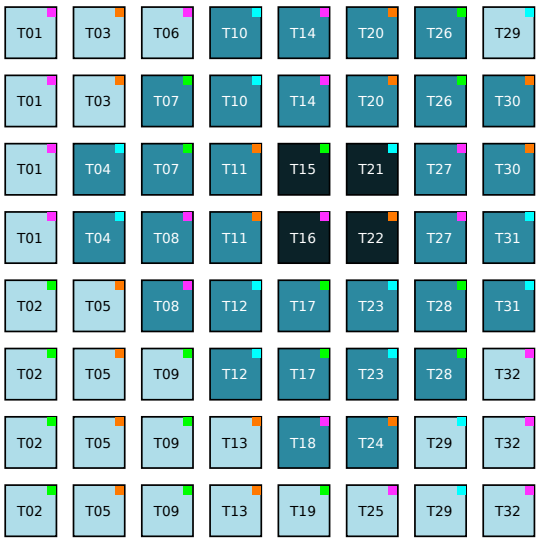

FIG. 25 – Affectation des éléments aux différentes sous-tâches dans un maillage 2D multi p

B Efficacité sur les accélérateurs de calcul

L'algorithme glouton Alg. 3 est particulièrement adapté aux GPUs. Chaque tâche rassemble des éléments de même degré ce qui permet un véritable parallélisme SIMD au sein d'une tâche.

Sur *Xeon Phi*[6], le calibrage des tâches permet de bien répartir la charge entre les 61 coeurs physiques. Grâce à ce mécanisme, le temps d'exécution sur *Xeon Phi* est identique au temps d'exécution sur un CPU *Sandy Bridge*. Même si ce résultat n'est pas très intéressant aujourd'hui, il est prometteur pour les accélérateurs de demain.

6. Intel Xeon Phi : www.intel.com/xeonphi

IV Implémentation GPU

Cette section décrit le portage du code `JAGUAR` sur GPU. Afin de mieux comprendre les enjeux de cette implémentation, la première sous section explique les bonnes pratiques à adopter sur ce type d'accélérateur.

IV.1 Accès mémoire coalescés et grain de calcul

Sur GPU, il y a beaucoup de *threads* matériels disponibles et le coût de création d'un *thread* est négligeable. De plus, la mémoire globale du GPU est bien plus performante que la RAM du CPU mais il faut que les accès soient effectués selon des *patterns* (motifs) bien définis. Toute la subtilité d'un noyau de calcul efficace réside dans le choix d'un grain de calcul adapté pour que les accès mémoire soient efficaces.

Sur `JAGUAR`, la vitesse d'exécution est principalement limitée par les accès à la mémoire, il est donc nécessaire de soigner ces accès pour qu'ils soient les plus rapides possible. Il faut effectuer des accès mémoire coalescés pour avoisiner les débits maximum.

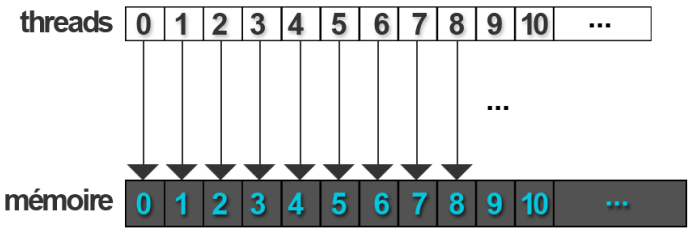

FIG. 26 – Accès mémoire coalescés

La Fig. 26 présente le motif à suivre pour que les accès à la mémoire globale soient parfaitement coalescés : chaque *thread* accède la zone mémoire qui suit directement celle du *thread* précédent. Dans cette situation, le SMX effectue une seule requête dans la mémoire globale pour tous les *threads* dont il est responsable. Cette méthodologie va influencer le choix du grain de calcul : pour que les accès mémoires soient coalescés il faut adapter le nombre de *threads*.

A Exemple de code

Prenons l'exemple de la multiplication d'un scalaire *alpha* avec un vecteur X de 8 éléments ($X = alpha \times X$), le code suivant calcule la multiplication avec un seul *thread* :

```
1 subroutine scal_v1(alpha, X)
2   do i = 1, 8
3     X(i) = alpha * X(i)
4   end do
5 end subroutine scal_v1
```

Cela n'est pas très judicieux sur GPU puisqu'il y a beaucoup de *threads* disponibles et qu'un *thread* est relativement lent. Considérons maintenant le cas où l'on utilise deux *threads* pour traiter le problème :

```
1 subroutine scal_v2(alpha, X)
2   id = getMyId() ! retourne 0 ou 1 selon le thread
3   do i = (id * 4) + 1, (id * 4) + 4
4     X(i) = alpha * X(i)
5   end do
6 end subroutine scal_v2
```

La Fig. 27 présente les accès mémoire de la deuxième version (`scal_v2`) : en rouge les zones mémoires accédées par le *thread* 0 et en vert les zones mémoires accédées par le *thread* 1. Cette version est à gros grain car il y a peu de *threads* et ils effectuent une grande partie du travail chacun.

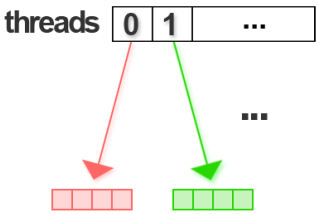

FIG. 27 – Accès mémoire de `scal_v2` (version parallèle à gros grain)

La version 2 n'est pas très efficace sur GPU car les accès mémoire ne sont pas coalescés (il y a au minimum 4 accès à la mémoire globale). Voici le code d'une troisième version pensée pour GPU avec 8 *threads* :

```
1 subroutine scal_v3(alpha, X)
2   id = getMyId() ! retourne 0, 1, 2, 3, 4, 5, 6 ou 7 selon le thread
3   X(id) = alpha * X(id)
4 end subroutine scal_v3
```

Avec 8 *threads* (taille de X) la boucle suivant les éléments de X disparaît : un *thread* traite un élément. Les accès mémoire des *threads* 0 à 4 sont shématisés Fig. 28.

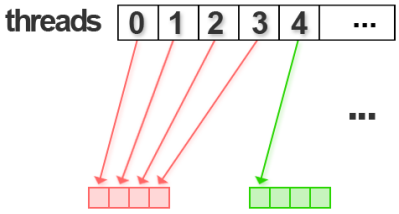

FIG. 28 – Accès mémoire de `scal_v3` (version parallèle à grain fin)

Dans la troisième version (`scal_v3`), les accès à la mémoire globale sont parfaitement coalescés ce qui minimise le nombre de transactions à effectuer (il y a un seul accès mémoire au lieu de quatre dans la version 2). Cette dernière version est à grain fin car il y a beaucoup de *threads* et ils effectuent une petite part du travail.

Cet exemple illustre parfaitement la nécessité d'adapter le grain de calcul en fonction des données (pour maximiser la coalescence des accès). De manière générale, sur GPU, les codes efficaces sont à grain fin.

Dans `JAGUAR`, toutes les routines sur GPU ont été adaptées pour que le parallélisme soit à grain fin. Par contre, la coalescence des accès n'est pas toujours parfaite puisque les routines utilisent souvent plusieurs *buffers* et il n'est pas toujours possible ni facile d'avoir des accès coalescés sur tous les *buffers* en même temps.

IV.2 Répartition des temps de calcul par routine

La Fig. 29 permet d'observer la répartition des temps de calcul par routine sur GPU.

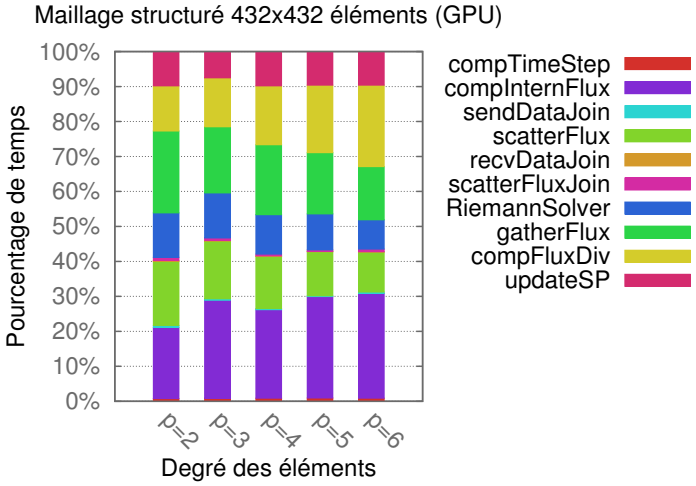

FIG. 29 – Pourcentage de temps occupé par routine (GPU)

À la différence de la répartition sur CPU (cf. Fig. 15), les routines `computeInternFlux` et `computeFluxDivergence` occupent significativement moins de temps sur GPU. Alors que les routines `scatterFlux` et `gatherFlux` occupent, elles, beaucoup plus de temps. Notons que, comme pour la Fig. 15, la Fig. 29 a été construite à partir de la version finale de `JAGUAR` GPU. Il est donc normal que les temps des routines `computeInternFlux` et `computeFluxDivergence` soient assez faibles par rapport aux autres routines puisque ces dernières ont été optimisées et leur performance est limitée par le calcul (or, sur GPU, les codes limités par le calcul s'exécutent plus rapidement que les codes limités par la mémoire). En contrepartie, les routines `scatterFlux` et `gatherFlux` sont limitées par la mémoire et elles n'ont pas été autant optimisées que `computeInternFlux` et `computeFluxDivergence`.

Les sections suivantes proposent de détailler les principales optimisations effectuées sur les routines `RiemmanSolver`, `computeInternFlux` et `computeFluxDivergence` pour améliorer les performances de `JAGUAR` sur GPU. Nous ne détaillerons pas le portage des routines `scatterFlux` et `gatherFlux` car leur implémentation est relativement triviale.

IV.3 Optimisation du solveur de Riemann

Cette section présente le portage de la routine `RiemannSolver` et permet de comprendre la stratégie décrite en Sec. IV.1 sur un cas concret.

Le solveur de Riemann calcule la valeur des flux (*buffer* `faceFlux`) aux interfaces des éléments à partir des solutions extrapolées à gauche (*buffer* `primitiveL`) et à droite (*buffer* `primitiveR`). Chaque calcul de flux est indépendant et ne fait intervenir que la solution extrapolée à droite et la solution extrapolée à gauche correspondante. Chaque solution extrapolée et chaque flux est composé de 5 valeurs (selon les variables conservatives $\rho, \rho u, \rho v, \rho w, \rho E$).

Nous avons alors créé autant de *threads* que de flux sur les faces pour lancer le noyau de calcul sur GPU. La taille des blocs a ensuite été fixé à 192 threads : cette configuration est optimale sur les GPUs d'architecture *Kepler*. Le nombre de blocs est calculé à partir du nombre de *threads* total et de la taille des blocs :

$$nbBlocs = \lceil \frac{nbThreads}{tailleBlocs} \rceil.$$

Chaque bloc ayant obligatoirement le même nombre de *threads*, le dernier bloc se retrouve très souvent avec trop de *threads* par rapport aux données qu'il a à traiter. Les *threads* inutiles attendent que les autres aient fini. Cette perte d'unités de calcul est négligable par rapport à la grille de *threads* totale.

Dans la structure initiale des *buffers* `faceFlux`, `primitiveL` et `primitiveR`, les flux et les solutions sont stockés les uns après les autres. De ce fait, les 5 variables conservatives d'un flux sont stockées de manière contiguë. Lorsque les *threads* accèdent aux données cela se traduit par un *stride* (un saut) entre chaque accès : si chaque *thread* accède à la première variable conservative de son flux alors il y a un saut de 5 entre tous les accès mémoire. Pour palier ce problème nous avons stocké les valeurs de ρ de chaque flux en premier, puis les valeurs de ρu de chaque flux en deuxième, puis les valeurs de ρv de chaque flux en troisième, etc. De cette façon les accès à la mémoire sont devenus parfaitement coalescés.

La version GPU du solveur de Riemann est aujourd'hui la routine la plus performante du code. Cela provient de sa nature : les accès sont parfaitement coalescés pour tous les *buffers* et il y a une grande quantité de calculs à effectuer. Pour donner un ordre de grandeur, sur un GPU de type *Nvidia Tesla K20c*, la routine atteint 100 Gflops/s alors que le code global avoisine seulement les 50 Gflops/s.

IV.4 Optimisation des produits matriciels

Les routines `computeInternFlux` et `computeFluxDivergence` sont un peu particulières et ont subi des optimisations supplémentaires. Ces dernières comportent des produits matriciels et permettent la réutilisation de certaines données.

A Optimisation de `computeInternFlux` pour GPU

Avant de commencer à écrire un noyau de calcul il faut déterminer la bonne granularité. La routine `computeInternFlux` extrapole les solutions et calcule les flux internes.

Dans un premier temps, nous avons fixé la taille des blocs suivant le nombre de flux contenus dans un élément. De cette façon, la grille représente l'ensemble des éléments et un bloc traite un élément spécifique. Cette organisation des *threads* est naturellement alignée sur les flux et par conséquent elle n'est pas alignée sur les solutions. Les accès peuvent donc être parfaitement coalescés au moment de l'écriture des flux (ou des solutions extrapolés), en contrepartie, cela n'est pas possible lors de la lecture des solutions. Dans `computeInternFlux` sur GPU, chaque *thread* calcule une solution extrapolée. Si cette solution extrapolée est interne à l'élément, alors le *thread* calcule aussi le flux associé.

Afin d'améliorer les performances, dans chaque bloc (équivalent à un élément), nous avons recopié les solutions de la mémoire globale vers la mémoire partagée. La mémoire partagée est bien plus rapide que la mémoire globale et elle est beaucoup plus efficace pour des accès hétérogènes. De plus, les solutions sont requises par plusieurs *threads* en même temps, il est donc judicieux d'utiliser la mémoire "partagée". Enfin, l'utilisation de la mémoire partagée permet d'effectuer des transactions les plus coalescés possible dans la mémoire globale (recopie des données) pour ensuite effectuer des accès irréguliers (mais efficace) en mémoire partagée. Pour les mêmes raisons, nous avons aussi utilisé la mémoire partagée pour stocker les coefficients d'extrapolation et les flux (ou solutions extrapolées).

Cependant cette première solution n'est pas très efficace : la taille des blocs n'est pas un multiple de 32. Les unités de calcul sont alors sous exploitées et il y a un grand nombre de *threads* inactifs (dépendant directement du nombre de flux). Pour palier à ce problème et rentabiliser au maximum les unités de calcul disponibles, nous avons traité plusieurs éléments par bloc de manière à ce que le nombre de flux total soit proche d'un multiple de 32.

Les Tab. 2 et Tab. 3 présentent le nombre d'éléments que nous avons associé à chaque bloc en fonction du degré p. Pour de petites valeurs de p, il est assez facile de trouver un coefficient multiplicateur permettant de se rapprocher d'un multiple de 32, cependant quand p augmente cela devient de plus en plus difficile. La colonne "flux par élément" représente le nombre de flux pour un élément de degré p selon une direction. Le nombre de *threads* est propre à un bloc et il est obtenu en multipliant le nombre de flux par le nombre d'éléments. La colonne "mémoire partagée" représente la quantité de mémoire utilisée par un bloc (elle comprend les solutions, les flux et les coefficients d'extrapolation). Le nombre d'éléments par bloc a été déterminé de manière empirique et il donne des résultats logiques : le nombre de *threads* est toujours proche d'un multiple de 32.

Degré	**Flux par élément**	**Éléments par bloc**	**Nb. threads**	**Mem. partagée**
1	$(1+2)\times(1+1)^1 = 6$	16	96	3.3 Ko
2	$(2+2)\times(2+1)^1 = 12$	8	96	3.7 Ko
3	$(3+2)\times(3+1)^1 = 20$	8	160	6.4 Ko
4	$(4+2)\times(4+1)^1 = 30$	2	60	2.7 Ko
5	$(5+2)\times(5+1)^1 = 42$	3	126	5.5 Ko
6	$(6+2)\times(6+1)^1 = 56$	2	112	5.2 Ko

Tab. 2 – Nb. d'éléments par bloc en fonction du degré de l'extrapolation en 2D

Degré	Flux par élément	Éléments par bloc	Nb. threads	Mem. partagée
1	$(1+2)\times(1+1)^2 = 12$	8	96	3.3 Ko
2	$(2+2)\times(2+1)^2 = 36$	7	252	8.6 Ko
3	$(3+2)\times(3+1)^2 = 80$	2	160	6.4 Ko
4	$(4+2)\times(4+1)^2 = 150$	1	150	6.3 Ko
5	$(5+2)\times(5+1)^2 = 252$	1	252	10.7 Ko
6	$(6+2)\times(6+1)^2 = 392$	1	392	16.7 Ko

Tab. 3 – Nb. d'éléments par bloc en fonction du degré de l'extrapolation en 3D

B Optimisation de `computeFluxDivergence` pour GPU

Les optimisations apportées à la version GPU de `computeFluxDivergence` sont assez proches des optimisations précédentes. Lors de la dérivation, on calcule les solutions à partir des flux. Le nombre de *threads* par bloc est basé sur le nombre de solutions (au lieu du nombre de flux pour `computeInternFlux`). Nous avons utilisé la mémoire partagée pour stocker les solutions, les flux et les coefficients de dérivation. Les Tab. 4 et Tab. 5 présentent le nombre d'éléments que nous avons associé à un bloc en fonction du degré des éléments.

Degré	Solutions par élément	Éléments par bloc	Nb. threads	Mem. partagée
1	$(1+1)^2 = 4$	16	64	1.3 Ko
2	$(2+1)^2 = 9$	14	126	2.4 Ko
3	$(3+1)^2 = 16$	8	128	2.4 Ko
4	$(4+1)^2 = 25$	5	125	2.4 Ko
5	$(5+1)^2 = 36$	7	252	2.6 Ko
6	$(6+1)^2 = 49$	5	245	2.6 Ko

Tab. 4 – Nb. d'éléments par bloc en fonction du degré pour la dérivation en 2D

Degré	Solutions par élément	Éléments par bloc	Nb. threads	Mem. partagée
1	$(1+1)^3 = 8$	16	128	2.5 Ko
2	$(2+1)^3 = 27$	7	189	3.6 Ko
3	$(3+1)^3 = 64$	2	128	2.4 Ko
4	$(4+1)^3 = 125$	1	125	2.4 Ko
5	$(5+1)^3 = 216$	1	216	4.0 Ko
6	$(6+1)^3 = 343$	1	343	6.2 Ko

Tab. 5 – Nb. d'éléments par bloc en fonction du degré pour la dérivation en 3D

La quantité de mémoire partagée utilisée par bloc est moins importante que dans la routine `computeInternFlux`. Dans `computeFluxDivergence` la quantité de mémoire

partagée dédiée au stockage des solutions est divisée par 5 (le nombre de variables conservatives) : au lieu de récupérer entièrement les solutions en une seule fois, il y a 5 accès séparés afin de diminuer la place occupée en mémoire partagée.

IV.5 Version multi GPU

La version multi GPU de `JAGUAR` s'appuie sur la bibliothèque MPI (*Message Passing Interface*). Le principe est relativement simple, quand il y a plusieurs nœuds de calcul, le maillage est découpé en autant de blocs. Chaque bloc possède plusieurs frontières. Aujourd'hui, `JAGUAR` traite uniquement des écoulements périodiques : il n'y a pas de conditions limites physiques, mais seulement des conditions limites de type numérique. Ces dernières sont traitées comme les raccords de blocs pour les calculs parallèles. Cela simplifie le problème, il n'y a qu'un seul type de frontière.

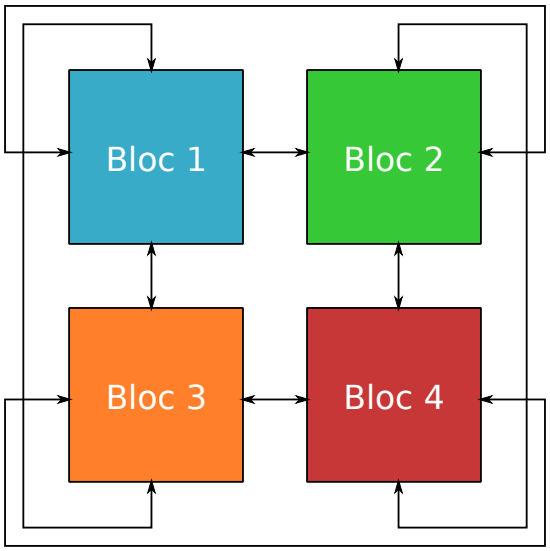

Fig. 30 – Découpage d'un maillage 2D en 4 blocs

La Fig. 30 illustre le découpage d'un maillage en 4 parties : chaque GPU calcule un bloc. Les communications nécessaires au calcul des solutions sont représentées par les flêches. L'écoulement calculé étant périodique, il y a 4 communications par bloc : une pour chaque côté.

Dans le code initial, l'envoi des solutions frontières était effectué après l'extrapolation et le calcul des flux internes et avant le regroupement des solutions extrapolées sur les faces ; cet envoi était non bloquant. La réception était effectuée directement après le regroupement des solutions extrapolées sur les faces et juste avant le solveur de Riemann : la réception était bloquante.

L'algorithme 4 précise l'implémentation des communications pour la méthode des Différences Spectrales. Les routines dédiées aux communications sont `sendDataJoin` pour l'envoi et `recvDataJoin` pour la réception (en gras sur Alg. 4). Les versions initiales des routines `recvDataJoin` et `sendDataJoin` sont détaillées dans l'Alg. 5 et l'Alg. 6. La réception des solutions extrapolées aux frontières ne peut pas être recouverte par le temps de calcul de `scatterFlux` car la réception des données est entièrement faite dans la routine `recvDataJoin` (après `scatterFlux`).

Algorithm 4 Spectral Differences MPI [SDM]

```
lastIt  ← readLastIteration()
rkSteps ← readRKSteps()
RKAlpha ← initRKAlpha( in::rkSteps )
S  ← readInitSolutions()
it ← 1
while (it ≤ lastIt) do
  dt ← computeTimeStep( in::S )
  SB ← S
  SI ← 0
  iStep ← 1
  while (iStep ≤ rkSteps) do
    F  ← computeInternFlux( in::S )
    sendDataJoin( in::F )
    SF ← scatterFlux( in::F )
    SF ← recvDataJoin( in::SF )
    FF ← riemannSolver( in::SF )
    F  ← gatherFlux( in::FF )
    SI ← computeFluxDivergence( in::F )
    S  ← dt × SB × SI × RKAlpha[iStep]  ou updateSP
    iStep ← iStep + 1
  end while
  it ← it + 1
end while
```

Algorithm 5 `sendDataJoin(in::F)` Initial [SDJI]

```
JS ← extractJoinAtSolutions( in::F )  du GPU vers le CPU
MPI_Isend( in::JS )  non bloquant
```

Algorithm 6 `recvDataJoin(in::SF)` Initial [RDJI]

```
JR    ← MPI_Recv()  bloquant
newSF ← addJoinToFaceSolutions( in::JR, in::SF )  du CPU vers le GPU
return newSF
```

La Fig. 31 montre la répartition des temps de calcul par routine quand on augmente le nombre de GPUs (version sans recouvrement : utilisation de Alg. 5 et Alg. 6).

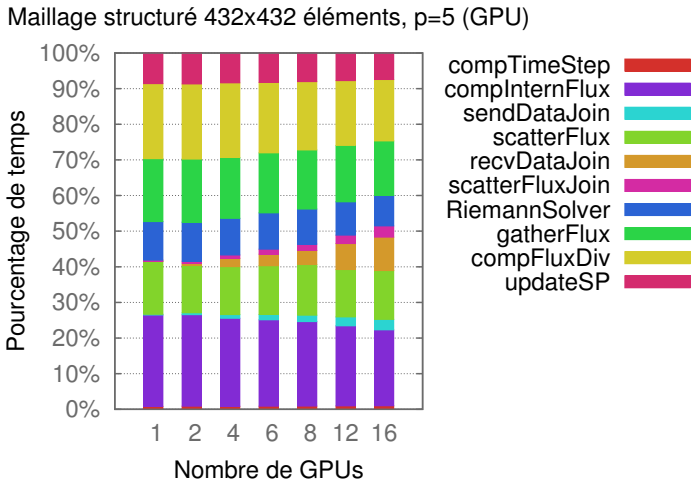

FIG. 31 – Pourcentage de temps occupé par routine (GPU MPI sans recouvrement)

On remarque que le pourcentage de temps pris par la routine `recvDataJoin` augmente considérablement avec le nombre de GPUs. Afin de palier ce problème, nous avons recouvert les temps de communication par le temps de calcul de la routine `scatterFlux`. Pour y parvenir nous avons utilisé des communcations complètement non bloquantes avec l'appel aux primitives `MPI_IRecv` et `MPI_ISend`. L'Alg. 7 et l'Alg. 8 décrivent respectivement les nouvelles implémentations de `sendDataJoin` et `recvDataJoin`.

Algorithm 7 `sendDataJoin(in::F)` [SDJ]

`MPI_Irecv( out::JR )` *non bloquant*
`JS` ← `extractJoinAtSolutions( in::F )` *du GPU vers le CPU*
`MPI_Isend( in::JS )` *non bloquant*

Algorithm 8 `recvDataJoin(in::SF)` [RDJ]

`MPI_Wait()` *bloquant*
`newSF` ← `addJoinToFaceSolutions( in::JR, in::SF )` *du CPU vers le GPU*
return `newSF`

Avec cette nouvelle version, les communications peuvent être effectuées pendant l'exécution de la routine `scatterFlux`. Nous avons aussi utilisé les communications persistantes pour essayer de diminuer les coûts d'appel à MPI (les communications persistantes ne sont pas détaillées dans ce rapport). Ici seuls les temps de transfert entre CPU et GPU ne peuvent pas être évités. Nous n'avons pas utilisé de bibliothèque MPI spécialisée pour CUDA dans `JAGUAR`.

La Fig. 32 montre la nouvelle répartition des temps de calcul par routine avec le recouvrement (utilisation de Alg. 7 et Alg. 8). On remarque assez clairement que le

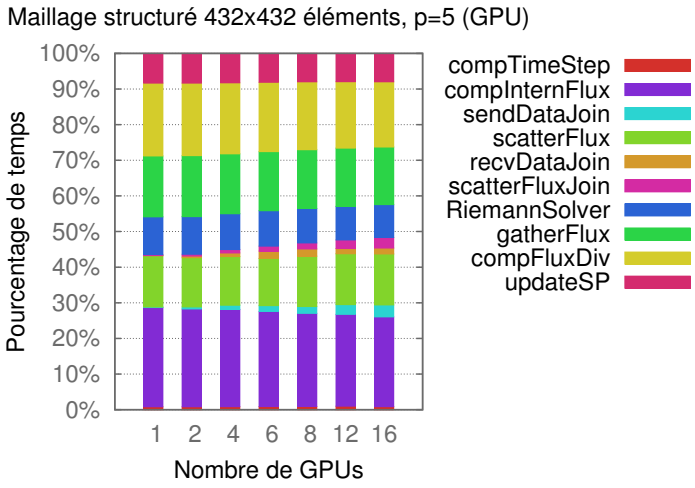

FIG. 32 – Pourcentage de temps occupé par routine (GPU MPI avec recouvrement)

pourcentage de temps pris par la routine `sendDataJoin` est bien inférieur à celui de la version initiale.

La Fig. 33 met en évidence l'importance de ce recouvrement en comparant les *speedups* des versions avec et sans recouvrement. Nous ne nous attarderons pas ici sur les performances multi GPU qui seront bien plus détaillées dans la section V.4, cependant il est bon de retenir l'importance du recouvrement quand le partitionnement devient important.

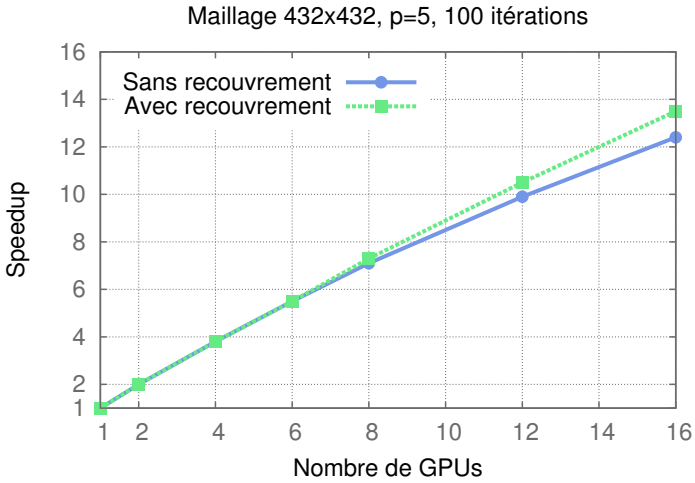

FIG. 33 – *Speedup* de la version multi GPU de `JAGUAR` (avec et sans recouvrement)

V Expérimentations

Dans cette section, nous discutons des différents résultats obtenus sur CPU et sur GPU. Les conditions de tests sont détaillées dans la première sous section puis, par la suite, les résultats sont présentés en s'appuyant sur des courbes et des histogrammes récapitulatifs.

V.1 Conditions de test

Les différentes expérimentations ont été faites sur **Neptune** pour la partie CPU, sur **Airain** pour la partie mono GPU et sur **Airain** et **Curie** pour la partie multi GPU.

A Neptune

Neptune est un supercalculateur interne au CERFACS. Les nœuds standards de Neptune possèdent deux CPUs de type *Intel Xeon E5-2670*[7] @ 2.60 GHz et 32 Go de mémoire vive. Le processeur *Intel Xeon E5-2670* est d'architecture *Sandy Bridge*[8] : il possède 8 coeurs et 20Mo de cache. Sa puissance théorique en opérations flottantes (double précision) est de 83.2 gigaflops/s alors que sa bande passante mémoire maximale est de 51.2 Go/s. Les processeurs *Sandy Bridge* sont capables d'effectuer 4 opérations flottantes (double précision) en un cycle d'horloge grâce au mécanisme de vectorisation (jeu d'instructions AVX-256). Lors des tests, les technologies *Intel Hyper-Threading*[9] et *Intel Turbo Boost*[10] étaient désactivées.

B Airain

Airain est un supercalculateur civil du CEA[11]. Ce dernier est constitué de 752 nœuds standards ainsi que de 8 nœuds hybrides. Les noeuds standards sont à peu près similaires à ceux de Neptune mais nous ne les avons pas utilisés. En revanche, chaque nœud hybride est équipé de deux GPUs (soit 16 GPUs disponibles) de type *Nvidia Tesla K20c*[12] (architecture *Kepler*). Ce GPU possède 13 SMX et 832 coeurs CUDA (double précision) ainsi que 5 Go de mémoire vive. Sa puissance théorique en opérations flottantes (double précision) est de 1.17 teraflops/s alors que sa bande passante mémoire maximale est de 208 Go/s (avec l'*ECC*[13] désactivé). Lors de nos tests l'ECC était activé : la bande passante et la quantité mémoire étaient donc légèrement diminuées par rapport au pic.

La table 6 permet de comparer les performance crêtes du CPU et du GPU que nous avons utilisés lors de nos tests. La consommation électrique (TDP) maximale d'un GPU est à peu près le double de celle d'un CPU.

Le coût d'un nœud équipé de deux *Tesla K20c* est proche de 10 000€ alors que le prix d'une *Tesla K20c* est de 2 000€ : deux *Tesla K20c* représentent 40% du prix total. Le coût de la maintenance de ce type de carte est nul.

7. Intel Xeon E5-2670 : http://ark.intel.com/fr/products/64595/
8. Intel Sandy Bridge : http://en.wikipedia.org/wiki/Sandy_Bridge
9. Intel Hyper-Threading :http://fr.wikipedia.org/wiki/Hyper-Threading
10. Intel Turbo Boost : http://en.wikipedia.org/wiki/Intel_Turbo_Boost
11. CEA : Commissariat à l'énergie atomique et aux énergies alternatives
12. Nvidia Tesla K20c : http://en.wikipedia.org/wiki/Nvidia_Tesla
13. ECC : http://en.wikipedia.org/wiki/ECC_memory

Modèle	Unités vectorielles	Performances	Débit mémoire	TDP max
Xeon E5-2670	32 @ 2,6 Ghz	83,2 Gflops/s	51,2 Go/s	115 watts
Tesla K20c	832 @ 1,0 Ghz	1170,0 Gflops/s	208,0 Go/s	225 watts

TAB. 6 – Performances crêtes du Xeon E5-2650 et de la Tesla K20c

C Curie

Curie est un supercalculateur militaire du CEA. Sa particularité est de proposer 256 nœuds hybrides. Chacun de ces nœuds est constitué de deux GPUs de type *Nvidia Tesla M2090* (architecture *Fermi*). Ce GPU possède 16 *Streaming Multiprocessors* (SMs) et 6 Go de mémoire vive. Chaque SM possède 32 cœurs CUDA simple précision et 16 cœurs CUDA double précision (soit un total de 256 cœurs double précision).

Sa puissance théorique en opérations flottantes (double précision) est de 666.1 gigaflops/s alors que sa bande passante mémoire maximale est de 177 Go/s (avec l'*ECC* désactivé). Lors de nos tests l'ECC était activé : la bande passante et la quantité mémoire étaient donc légèrement diminuées par rapport au pic.

Modèle	Unités vectorielles	Performances	Débit mémoire	TDP max
Tesla K20c	832 @ 1,0 Ghz	1170,0 Gflops/s	208,0 Go/s	225 watts
Tesla M2090	256 @ 1,3 Ghz	666,1 Gflops/s	177,0 Go/s	225 watts

TAB. 7 – Performances crêtes de la Tesla K20c et la Tesla M2090

La table 7 permet de comparer les performances de la *Tesla M2090* d'architecture *Fermi* avec la *Tesla K20c* d'architecture *Kepler* (plus récente). On remarque que le pic de performance (opérations flottantes) de la *K20c* est bien plus élevé que celui de la *M2090*. En contrepartie, la différence entre les débits mémoire n'est pas très importante.

Nous avons fait la majorité des tests multi GPU sur Curie en raison du grand nombre de GPUs disponibles.

D Validité des résultats

L'architecture des GPUs est différente de celle des CPUs : elle n'est pas compatible x86[14]. Les calculs sont effectués différemment et souvent à l'avantage de la performance (et non de l'exactitude). Cependant les résultats GPUs sont très proches des résultats CPU ce qui les rend complètement exploitables.

E Le cas test

Les expérimentations suivantes reposent sur un cas test bien défini : la convection (translation) d'un vortex (ou tourbillon). Ce dernier est initialisé dans une boîte au centre du maillage et se déplace de la gauche vers la droite. Au bout d'un certain nombre d'itérations, le vortex doit revenir à sa position initiale puisque l'écoulement est périodique.

14. Famille x86 : http://fr.wikipedia.org/wiki/X86

Les valeurs des solutions initiales sont connues et quand le vortex a effectué "un tour", il est facile de mesurer la marge d'erreur des solutions calculées.

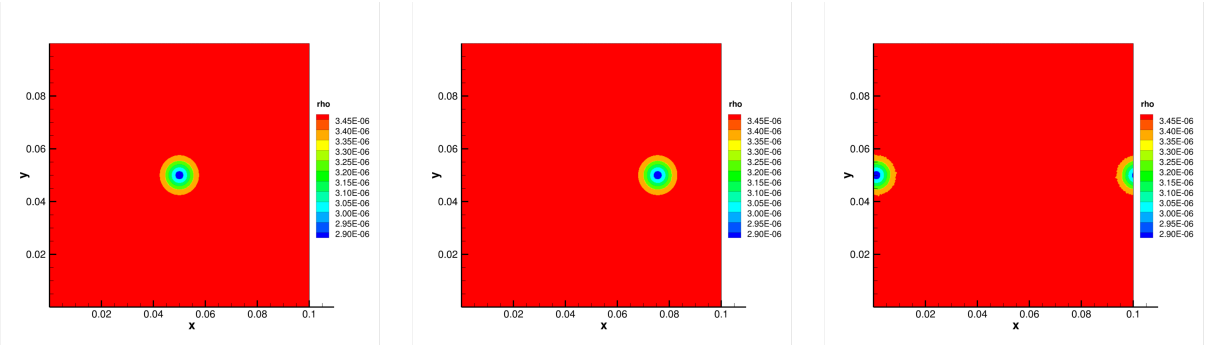

FIG. 34 – Étude des variations de densité (en kg.m^{-3}) d'un vortex pour un maillage 2D (plus la couleur est froide, plus la densité est faible). La première figure représente l'état initial, la deuxième représente la densité après 903 itérations et la troisième représente la densité après 2408 itérations.

La Fig. 34 illustre le tourbillon/vortex et son déplacement dans l'espace (suivant la variation de densité ρ). Dans la dernière figure, on constate que le vortex est séparé en deux parties : cela est dû au caractère périodique de l'écoulement.

V.2 Analyse des performances CPU

Dans cette section nous analysons l'impact des modifications présentées en Sec. III sur le code `JAGUAR` en séquentiel. Les temps relevés sont ceux de la boucle principale de calcul, l'initialisation des données et l'écriture des résultats ne sont pas pris en compte. Pour compiler le code, nous avons utilisé le compilateur `Fortran` d'*Intel* (ifort version 13.1.0). Pour calculer le nombre d'opérations flottantes par seconde (Gflops/s) nous avons utilisé deux méthodes différentes :

- *likwid-perfctr* : un outil de mesure automatique qui s'appuie sur les compteurs matériels du CPU (méthode automatique),
- nous avons compté les opérations flottantes une par une dans le code (méthode manuelle).

Les résultats obtenus avec ces deux méthodes sont identiques à 50 Mflops/s près même si *likwid-perfctr* à tendance à légèrement surévaluer les performances par rapport au calcul manuel.

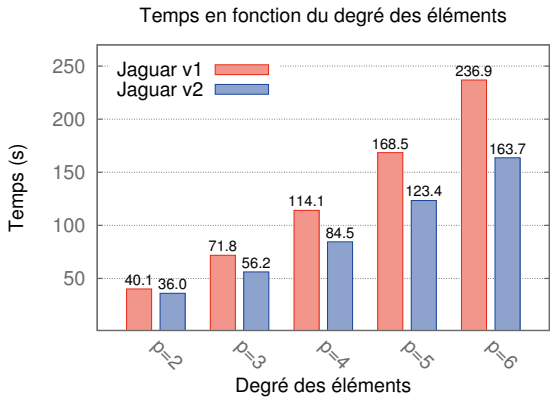

FIG. 35 – Temps d'exécution en fonction du degré des éléments (maillage 2D structuré de 128x128 éléments, 100 itérations)

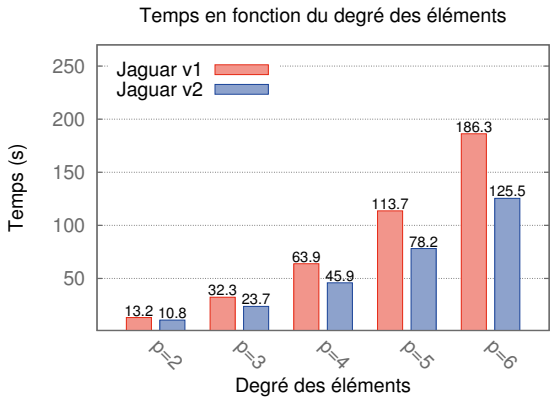

FIG. 36 – Temps d'exécution en fonction du degré des éléments (maillage 3D structuré de 25x25x3 éléments, 100 itérations)

Dans Fig. 35, Fig. 36, Fig. 37 et Fig. 38 "Jaguar v1" est la version initiale (sans les modifications Sec. III) alors que "Jaguar v2" est la dernière version du code (avec les modifications). Dans Fig. 37 et Fig. 38 nous avons retenu les Gflops/s de la méthode automatique car il aurait été trop complexe d'instrumenter "Jaguar v1" avec la méthode manuelle. De plus, la méthode automatique permet de différencier les opérations flottantes scalaires des opérations flottantes vectorielles.

Dans Fig. 35 et Fig. 36 plus le temps d'exécution est faible, meilleures sont les performances. En deux dimensions, on remarque que plus le degré des éléments augmente plus l'écart entre les temps d'exécution se creuse (de 10% en moins pour `p=2` à 30% en moins pour `p=6`). En trois dimensions, le constat est identique même si les gains de temps sont légèrement supérieurs. Cette petite différence de gain s'explique par la différence de quantité de calcul : en 3D il y a plus de calculs matriciels et donc les performances sont légèrement revues à la hausse par rapport au cas 2D.

Les Fig. 37 et Fig. 38 exposent les performances des deux versions en terme d'opérations flottantes. Dans la première version de `JAGUAR`, les Gflops/s totaux stagnent alors

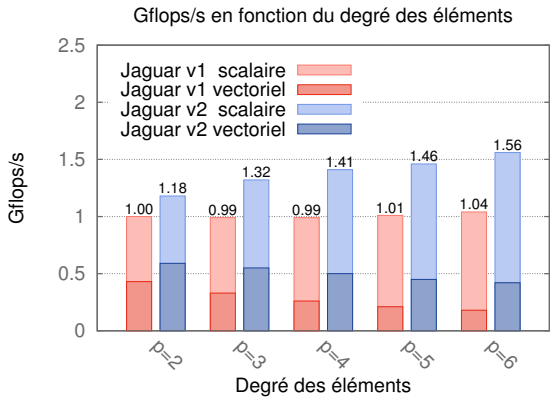

FIG. 37 – Gflops/s en fonction du degré des éléments (maillage 2D structuré de 128x128 éléments, 100 itérations)

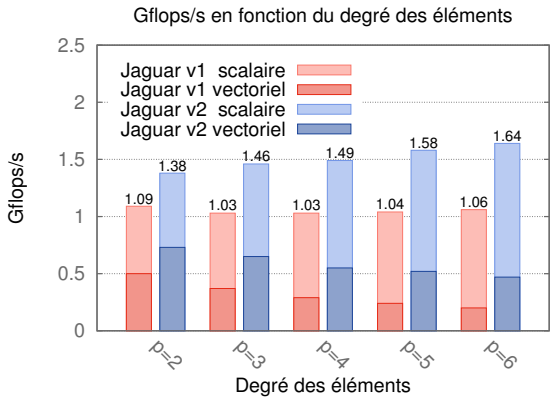

FIG. 38 – Gflops/s en fonction du degré des éléments (maillage 3D structuré de 25x25x3 éléments, 100 itérations)

que dans la deuxième version ils augmentent avec le degré p. Les Gflops/s scalaires sont un peu supérieurs dans la nouvelle version mais ils restent assez proche de la version initiale. Par contre, les Gflops/s vectoriels sont nettement supérieurs dans la version 2 : la réécriture du code en produits matriciels permet au compilateur de mieux vectoriser le code. On remarque aussi que quand p augmente, les Gflops/s scalaires augmentent et les Gflops/s vectoriels diminuent. Cette diminution des Gflops/s vectoriels est difficile à expliquer : le code est identique quel que soit le degré des éléments. La boucle qui est vectorisée effectue `p+1` ou `p+2` itérations (cela dépend des routines). Il est possible que le compilateur soit optimisé pour traiter des boucles de petites tailles (optimisations en dur) mais cette explication n'a pas été pas vérifiée.

Les performances présentées dans les figures précédentes sont satisfaisantes et font de `JAGUAR` un des codes de CFD les plus rapide développé au CERFACS. La puissance crête du CPU *Intel Xeon E5-2670* est de 83,2 Gflops/s pour 8 cœurs, on peut alors considérer que la puissance crête d'un cœur est de $83,2 \div 8 = 10.4$ Gflops/s. Le code `JAGUAR` exploite donc 15% de la puissance crête du processeur. Ce résultat n'est pas exceptionnel mais il est à mettre en perspective avec la nature des calculs : le problème traité est un problème d'algèbre linéaire creuse. Il est donc difficile de tirer parti des caches du processeur et les performances s'en ressentent.

V.3 Analyse des performances mono GPU

Dans cette section nous étudions les performances de la version GPU de `JAGUAR`. Nous avons compilé le code avec le compilateur de PGI (pgfortran version 13.4) car il est le seul à proposer la compatibilité avec CUDA (en `Fortran`). La version séquentielle du code (cf. Sec. V.2) sert de référence pour l'analyse des performances GPU. Sur GPU nous n'avons pas trouvé d'outil permettant de calculer automatiquement le nombre d'opérations flottantes : nous l'avons donc calculé manuellement.

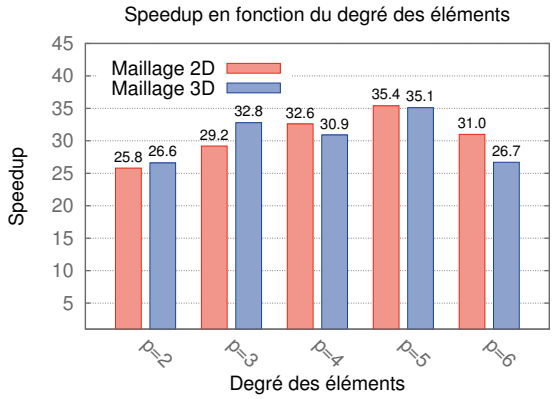

FIG. 39 – *Speedup* en fonction du degré des éléments (maillage 2D structuré de 512x512 éléments, maillage 3D structuré de 30x30x30 éléments, 100 itérations)

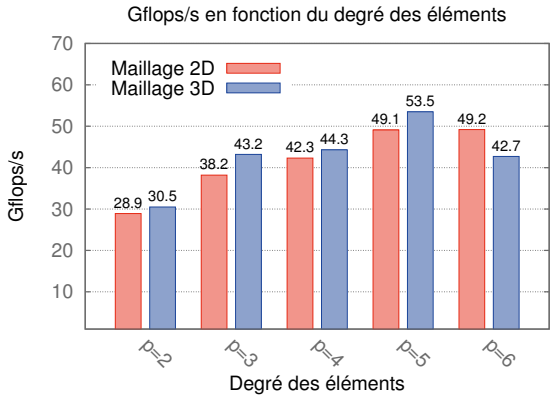

FIG. 40 – Gflops/s en fonction du degré des éléments (maillage 2D structuré de 512x512 éléments, maillage 3D structuré de 30x30x30 éléments, 100 itérations)

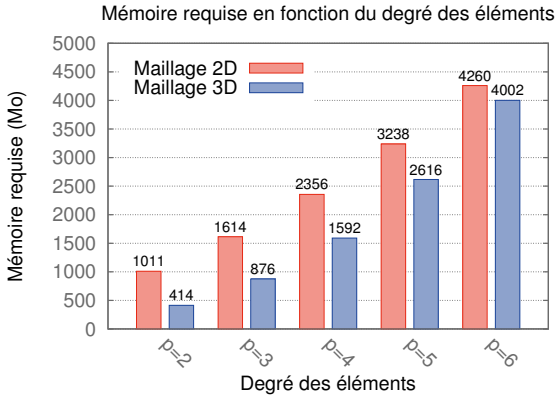

FIG. 41 – Mémoire requise en fonction du degré des éléments (maillage 2D structuré de 512x512 éléments, maillage 3D structuré de 30x30x30 éléments, 100 itérations)

La Fig. 39 présente les différentes accélérations obtenues en fonction du degré p des éléments. Les maillages 2D (512x512 éléments) et 3D (30x30x30 éléments) sont volontairement de grande taille :

- aujourd'hui, le nombre de GPUs sur les supercalculateurs est limité, il faut remplir au maximum la mémoire disponible pour en tirer parti sur des cas concrets,
- les performances sont meilleures quand le GPU doit effectuer une grande quantité de calcul.

Quel que soit le maillage (2D ou 3D), le *speedup* est assez stable : il est proche de 30 (Fig. 39). Quand p vaut 2 les performances sont moins bonnes car la quantité de travail n'est pas assez importante pour occuper correctement la carte. Le cas `p=6` en 3D est anormalement moins bon que le cas 2D : cela s'explique par un nombre de threads par bloc éloigné d'un multiple de 32 et une surconsommation de mémoire partagée (cf. Tab. 3 et Tab. 5). Les résultats restent quand même encourageants mais il ne faut pas oublier que les CPUs d'aujourd'hui possèdent eux aussi beaucoup de cœurs de calcul. La version GPU est 30 fois plus rapide que la version séquentielle : une version de `JAGUAR` qui utilise 1 cœur sur les 8 disponibles. On peut alors considérer qu'avec 8 cœurs on obtient un *speedup* de 8 sur CPU (*speedup* parfait). Dans ce cas, la version GPU est 4 fois plus rapide (environ) que la version multi-cœurs CPU. Cette performance est à mettre en regard avec la consommation électrique : le CPU *Intel Xeon E5-2670* consomme au maximum 115 watts et la carte *Nvidia Tesla K20c* consomme au maximum 225 watts. Si le GPU va 4 fois plus vite que le CPU, alors il consomme environ deux fois moins pour une quantité de travail fixée. L'analyse précédente est uniquement basée sur les consommations électriques crêtes et non sur des relevés réels, il y a de grandes chances pour que la réalité soit un peu différente.

Les performances en termes d'opérations flottantes sont assez mauvaises (entre 40 et 50 Gflops/s, cf. Fig. 40). Le pic théorique de la *Tesla K20c* est de 1170 Gflops/s et la version GPU de `JAGUAR` avoisine seulement 5% de la crête. Cependant, il ne faut pas oublier que le code est limité par les accès à la mémoire (dû au caractère creux de l'algorithme). En comparaison, le code de P. Castonguay [4] atteint 40 Gflops/s avec la précédente génération de GPUs (*Nvidia Tesla C2050*). Le code GPU est perfectible et il est très probable que des optimisations futures apportent encore des gains significatifs.

V.4 Analyse des performances multi GPU

Comme pour la version mono GPU, nous avons compilé la version multi GPU avec le compilateur `Fortran` de PGI (pgfortran version 13.4).

A Supercalculateur Airain, sans recouvrement calcul/communication

La version multi GPU actuelle de `JAGUAR` ne permet pas d'utiliser efficacement plus d'un GPU par nœud. C'est pour cette raison que les mesures suivantes n'utilisent pas plus de 8 GPUs (alors qu'il y en avait 16 de disponibles sur Airain). Ces mesures ont été faites sur une version de `JAGUAR` où les temps de communication n'étaient pas recouverts par du calcul.

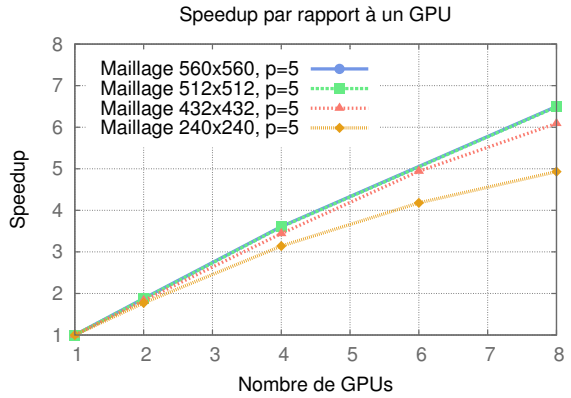

FIG. 42 – *Speedup* par rapport à un GPU (maillages 2D structurés, le degré des éléments est fixé à 5, 100 itérations)

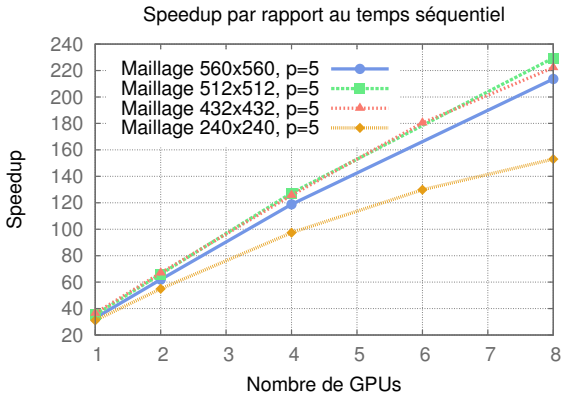

FIG. 43 – *Speedup* par rapport au temps séquentiel (maillages 2D structurés, le degré des éléments est fixé à 5, 100 itérations)

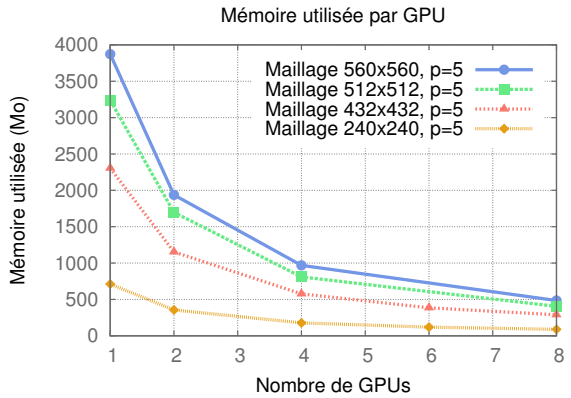

FIG. 44 – Consommation mémoire pour un GPU en fonction du nombre de GPUs (maillages 2D structurés, le degré des éléments est fixé à 5, 100 itérations)

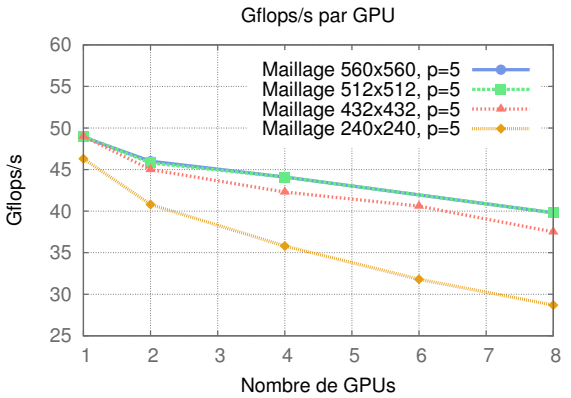

FIG. 45 – Gflops/s pour un GPU en fonction du nombre de GPUs (maillages 2D structurés, le degré des éléments est fixé à 5, 100 itérations)

Les Fig. 42 et Fig. 43 montrent l'accélération par rapport à un GPU et par rapport au temps séquentiel. Sur Fig. 42 on remarque que l'évolution du *speedup* est logarithmique :

l'efficacité diminue avec le nombre de GPUs. Pour 8 GPUs, on obtient un *speedup* proche de 6 pour les maillage 560x560 et 512x512 et 432x432, cependant, quand le maillage est plus petit le *speedup* chute fortement (maillage 240x240 : pour 8 GPUs le *speedup* atteint seulement 5). Plus le maillage est gros, plus le *speedup* se rapproche de l'optimal. La Fig. 43 reprend les mêmes temps GPUs que la Fig. 42, seul le temps de référence est différent (temps séquentiel CPU). Le *speedup* avec le maillage 560x560 est anormalement moins bon que le *speedup* avec les maillages 432x432 et 512x512 : cela est dû au temps séquentiel de la version 560x560 qui est meilleur que pour les maillages plus petits.

Les Fig. 44 et Fig. 45 présentent respectivement la mémoire utilisée par GPU en fonction du nombre total de GPUs utilisés et les Gflops/s atteints par GPU en fonction du nombre total de GPUs utilisés. On remarque que pour un maillage donné, la quantité de mémoire utilisée par un GPU diminue avec l'augmentation du nombre total de GPUs : c'est assez naturel puisque le maillage est découpé en n parts s'il y a n GPUs. Quand la quantité de données à traiter diminue, la quantité de travail par GPU diminue aussi et on constate que les performances en Gflops/s chutent (cf. Fig. 45). Cela met en évidence deux phénomènes :

- sur GPU, il faut une assez grosse quantité de travail pour tirer parti de la puissance,
- le *speedup* est difficilement optimal puisque les performances sont directement liées à la quantité de travail.

Cependant, il faut modérer les affirmations précédentes : dans `JAGUAR`, les parties "préparation des communications" et "communications" sont simplistes et non optimisées pour GPU. Quand la quantité de travail est faible, les routines dédiées aux communications prennent un temps trop important par rapport au temps calcul.

B Supercalculateur Curie, avec recouvrement calcul/communication

Ces mesures ont été faites avec la dernière version de `JAGUAR`. Les temps de communication sont recouverts par du calcul (cf. Sec. IV.5).

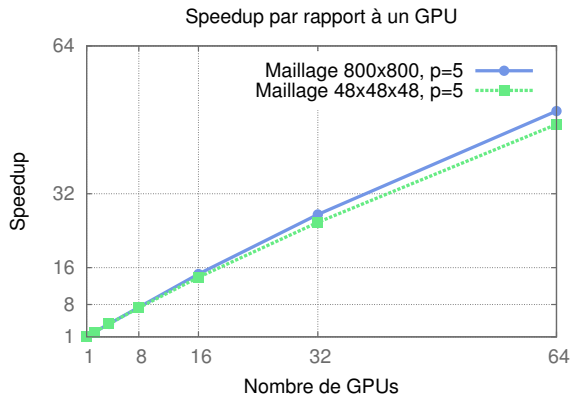

Fig. 46 – *Speedup* par rapport à un GPU (maillages 2D et 3D structurés, 100 itérations)

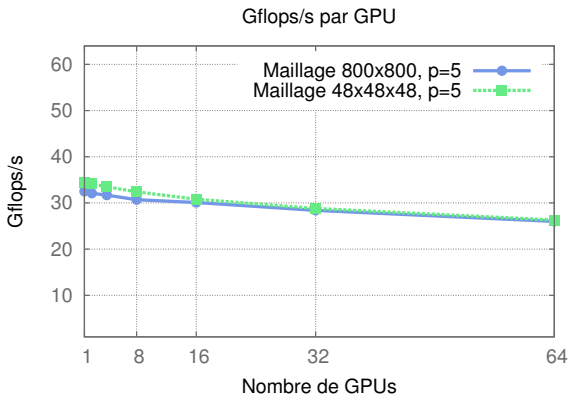

Fig. 47 – Gflops/s par GPU en fonction du nombre de GPUs (maillages 2D et 3D structurés, 100 itérations)

Dans cette sous section nous avons utilisé de "gros" maillages : un maillage 2D de 800×800 éléments et un maillage 3D de $48 \times 48 \times 48$ éléments avec $p = 5$. Ainsi, le

nombre de degrés de liberté (*Degrees Of Freedom*, DOFs) est à peu près identique pour le maillage 2D et le maillage 3D :

- maillage 2D : $800 \times 800 \times (5+1)^2 = 23\ 040\ 000$ DOFs,
- maillage 3D : $48 \times 48 \times 48 \times (5+1)^3 = 23\ 887\ 872$ DOFs.

La Fig. 46 illustre l'évolution du *speedup* en fonction du nombre de GPUs. Les maillages que nous avons utilisé pour ces mesures étant trop gros pour être lancés sur un GPU, nous avons "inventé" les temps de référence. Nous nous sommes basés sur les ratios obtenus avec des maillages plus petits : pour 2 GPUs nous avons considéré que le *speedup* était de 1,95 et pour 4 GPUs nous avons considéré que le *speedup* était de 3,75. Les résultats obtenus sont satisfaisants puisque les performances en terme d'opérations flottantes (cf. Fig. 47) ne chutent pas trop quand le nombre de GPUs augmente. Au final, on obtient un *speedup* proche de 50 pour 64 GPUs.

Sur la Fig. 46, on remarque que le *speedup* de la version 2D est légèrement supérieur à celui de la version 3D. Cela s'explique par la quantité des données échangées : en 2D on communique $4n$ données alors qu'en 3D on communique $6n$ données. De même, la taille des transferts entre CPU et GPU est de $4n$ en 2D et de $6n$ en 3D. Les Fig. 48 et Fig. 49 mettent ce phénomène en évidence : les routines `sendDataJoin` et `recvDataJoin` prennent un temps plus important en 3D.

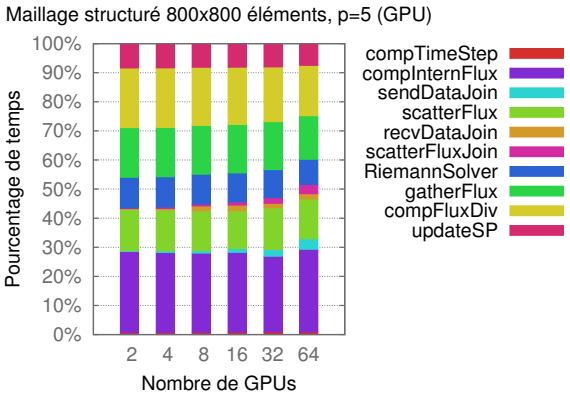

FIG. 48 – Pourcentage de temps occupé par routine en 2D (GPU MPI avec recouvrement)

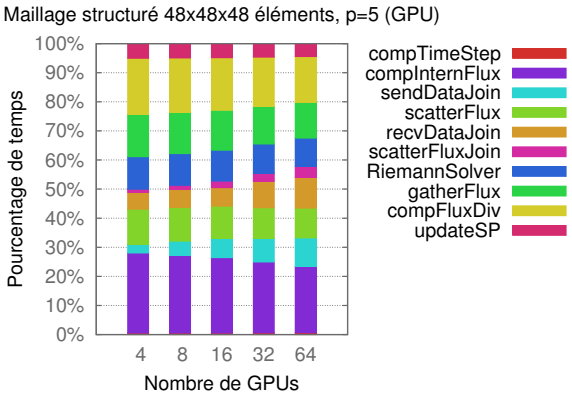

FIG. 49 – Pourcentage de temps occupé par routine en 3D (GPU MPI avec recouvrement)

Il est aussi intéressant de comparer les performances entre la nouvelle génération de GPUs (*Kepler*, cf. Sec. V.3 et Sec. A) et la précédente génération (*Fermi*, cf. Sec. B). On remarque que pour $p = 5$ on atteint 50 Gflops/s avec la *Tesla K20c* et 30 Gflops/s avec la *Tesla M2090*. Les performances de la nouvelle génération ne doublent donc pas les performances de la précédente ce qui montre bien que le code `JAGUAR` est en partie (mais pas complètement) limité par la bande passante mémoire. Le gain de performance apporté par la nouvelle génération de GPUs est d'environ 65%.

VI Conclusion

Le projet `JAGUAR` étant relativement jeune, **il a été possible de travailler sur une grande partie du code en profondeur**. Tout au long de mon stage, j'ai été directement en contact avec les principaux développeurs de `JAGUAR` ce qui m'a permis de progresser rapidement et de comprendre les clefs. Avec l'aide de l'équipe CFD, **j'ai été amené à réécrire une grande partie du cœur du solveur**.

Le travail accompli sur la version CPU du code est satisfaisant, les performances entre la version initiale et finale de `JAGUAR` en témoignent. Les modifications CPU ont toujours été dans l'objectif de porter le code sur GPU. L'écriture en produits matriciels a permis de tirer des performances intéressantes sur GPU. Même si le code n'est pas encore parfaitement optimisé, **la version GPU est 30 fois plus rapide que la version séquentielle**. Cette accélération est suffisante pour justifier l'utilisation des GPUs. De plus, **le code GPU traite aussi bien des maillages en deux ou en trois dimensions** ce qui le rend parfaitement utilisable dans des conditions réelles. Les travaux effectués sur la version multi GPU de `JAGUAR` sont eux aussi prometteurs avec **un *speedup* de 50 pour 64 GPUs**.

Les évolutions futures du code sont multiples. La nouvelle architecture GPU (*Kepler*) est capable de traiter efficacement des maillages multi p avec la création de *kernels* au sein d'un *kernel* (calcul adaptatif). **Le code `JAGUAR` GPU ne permet pas de calculer des maillages multi p** : il est à améliorer pour pouvoir tirer parti des possibilités matérielles. De manière générale, **les noyaux de calcul qui dispersent et regroupent les données (`scatter` et `gather`) ne sont pas très efficaces sur GPU.** De plus, les *buffers* de communication passent par la mémoire RAM du CPU et cela pourrait être évité. La technologie GPU Direct RDMA (*Remote Direct Memory Access*) permet des transferts directs entres plusieurs GPUs placés sur des nœuds différents. **Le code GPU actuel ne permet pas d'exploiter efficacement plusieurs GPUs sur un même nœud** alors que les configurations ont tendance à multiplier le nombre d'accélérateurs par nœud. Notons aussi que la structure des données peut être améliorée pour que certains accès soient mieux coalescés.

En définitive, le travail accompli est perfectible mais il offre une solution parfaitement utilisable.

Bibliographie

[1] P.L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43 :357–372, 1981.

[2] E.L. Toro. *Riemann solvers and Numerical Methods for Fluid Dynamics*. 1999.

[3] Y. Sun, Z. J. Wang, and Y. Liu. High-order multidomain spectral difference method for the Navier-Stokes equations on unstructured hexahedral grids. *Communications In Computational Physics*, 2(2) :310–333, april 2007.

[4] P. Castonguay, D. M. Williamsy, P. E. Vincentz, M. Lopez, and A. Jameson. On the development of a high-order, multi-gpu enabled, compressible viscous flow solver for mixed unstructured grids. In *20th AIAA Computational Fluid Dynamics Conference, Honolulu, Hawaii, AIAA Paper 2011-3229*, june 2011.

[5] Z.J. Wang B. Zimmerman and M.R. Visbal. High-order spectral difference : Verification and acceleration using gpu computing. In *21th AIAA Computational Fluid Dynamics Conference, San Diega, Canada, AIAA Paper 2013-2941*, june 2013.

[6] T. Brandvik and G. Pullan. Sblock : A framework for efficient stencil-based PDE solvers on multi-core platforms. In *10th IEEE International Conference on Computer and Information Technology*, pages 1181–1188, 2010.

[7] I. Marter. Handling different h and p refinements in the framework of Spectral Difference Method. september 2013.

[8] The Portland Group. CUDA Fortran programming guide and reference. `http://www.pgroup.com/doc/pgicudaforug.pdf`, 2013.

[9] NVIDIA Corporation. CUDA C Programming guide. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2013.

[10] A. Fosso-Pouangué. *Schémas Volumes Finis précis : application à l'aéroacoustique de jets subsoniques- TH/CFD/11/21.* PhD thesis, Université Pierre et Marie Curie, Paris, 2011. phd.