École d'Ingénieurs Sup Galilée

ONERA

Carlos Carrascal Manzanares

Stage

21/09/2014

Calculs multi-GPU en Mécanique des Fluides

Encadré par

Jean-Marie LE GOUEZ

Grâce à l'accord avec

Universidad Autónoma de Madrid

Table des matières

1. Introduction

Ce document est rédigé dans le cadre d'un stage de l'École d'Ingénieurs Sup Galilée pour les études de Mathématiques Appliquées et Calcul Scientifique. Ce stage a eu lieu à l'Office National d'Études et de Recherches Aérospatiales pendant l'année 2014, et il a été encadré par l'ingénieur chercheur Jean-Marie Le Gouez.

Le stage, intitulé « Calculs multi-GPU en Mécanique des Fluides », contribue à un des nombreux projets réalisés par l'ONERA pour permettre en France et en Europe des avancées dans le calcul haute performance et le développement de la simulation numérique. Ces projets sont impulsés en particulier grâce à l'Initiative de Recherche pour l'Optimisation Acoustique Aéronautique.

Le but initial du projet est, en partant d'un code de calcul d'écoulements bidimensionnels et instationnaires de fluides avec maillages non structurés écrit en FORTRAN 90, d'ajouter une nouvelle dimension 3D homogène. Cette modélisation (2'5D) est caractéristique de nombreuses expériences de turbulence [ind] en soufflerie de recherche pour identifier les mécanismes de déstabilisation des écoulements en proche paroi.

La première étape du projet permet de réaliser différents types d'optimisation séquentielle, en particulier une vectorisation automatique pour CPU X86 en FORTRAN.

En suite, le projet analyse et met en oeuvre 3 formes différentes de parallélisation :

- L'outil d'accélération à mémoire partagée OpenMP
- L'outil de parallélisation à mémoire distribuée MPI
- Le hardware de parallélisation hiérarchique multi-niveaux GPU

Finalement, le projet combine tous les trois (FORTRAN 90, C et CUDA respectivement) et analyse les résultats obtenus en terme de vitesse d'exécution, espace de mémoire et précision par l'utilisation d'un nœud CPU de calcul Intel Westmere de 12 threads et un nœud GPU Tesla K20 de 2048 cœurs de calcul CUDA, avec une attention particulière à la comparaison entre l'exécution CPU et GPU.

Les changements respectent en tout moment la structure originale du programme en permettant la portabilité ultérieure aux grands réseaux de calcul composés de centaines des nœuds GPU et millions des cœurs, en même temps que la méthode numérique peut être perfectionné (précision en fonction du nombre de cellules de discrétisation) selon les besoins des investigateurs scientifiques.

2. Motivation

Pendant l'année 2012, plus de 832 millions d'européens ont pris un avion pour voyager entre les pays membres [1]. Il y a autour de 448 compagnies aériennes et 701 aéroports commerciaux [2]. Chaque année ces nombres augmentent, que ce soit pour le transport des passagers ou de marchandises. Mais il y a certains facteurs qui préoccupent véritablement, comme la consommation énergétique, la pollution (soit environnementale soit acoustique), les infrastructures aéroportuaires et la compétitivité.

Alors, en l'année 2000 l'ACARE (Advisory Council for Aviation Research and Innovation in Europe [3]) avec l'aide de plus de 40 associations et les pays membres de l'UE a publié le rapport « European Aeronautics : A vision for 2020 » [4], ou il établit les objectifs de l'industrie aéronautique européenne pour l'année 2020. On va détailler les objectifs les plus importants :

- Réduire de 50% l'émission de CO_2 dans le compte passager/kilomètre
- Réduire de 80% l'émission de NO_x pendant l'atterrissage et le décollage
- Réduire de 80% les accidents
- Augmenter de 300% la capacité de gestion des mouvements aériens
- Réduire de 50% le coût de création et d'opération des avions
- Réduire de 50% l'émission de bruit

Pour attendre le résultat de réduire l'émission de bruit, en France est crée en 2005 (et renouvelé en 2010) l'IROQUA (Initiative de Recherche pour l'Optimisation acoustiQUe Aéronautique [5]), afin de fédérer les travaux de recherche menés dans le pays pour l'industrie et les acteurs académiques. Les partenaires, par exemple Airbus, Dassault Aviation, Safran, Air France et Aéroports de Paris, sont coordonnés par l'ONERA (Office National d'Études et de Recherches Aérospatiales [6]). L'IROQUA a commencé une douzaine de projets de recherche destinés à améliorer la compréhension du bruit perçu des aéronefs [7]. Quelques exemples :

- COMATEC : travaille sur les matériaux qui absorbent les ondes sonores
- BRUCO : travaille sur le bruit de la combustion
- AEROCAV : travaille sur le bruit produit par les trains d'atterrissage et les dispositifs hypersustentateurs, lesquels représentent le 50% du bruit perçu à l'atterrissage.

Dans le cadre des investigations amenés pour l'amélioration des méthodes de calcul dans ces contextes, les 27 et 28 Mai 2013, l'ONERA a participé dans le « 2nd International Workshop on High-Order CFD Methods » [8], à Cologne (Allemagne). Ces 2 jours ont été complets avec différentes présentations des méthodes et des résultats sur une série de cas-tests, en utilisant les méthodes de différences finies, volumes finis, distributions de résiduels, Galerkin discontinue, etc.

Pour encourager les participants et promouvoir l'investigation, des nombreux cas d'étude ont eté proposés, avec l'objectif de présenter les résultats des nouveaux cas sur des configurations plus proches de celles intéressant les industriels, pour le troisième atelier (Janvier 2015). Parmi eux, on peut trouver le cas « Turbulent Flow over a 2D Multi-Element Airfoil » [9]. L'organisation a fourni les données géométriques sur la pièce, les constantes physiques de l'air et les conditions aérodynamiques de ce problème

Spécifiquement, le but est de calculer les turbulences produites autour une aile d'avion en configuration « hyper-sustenté » composé de 3 éléments. Visuellement la géométrie est :

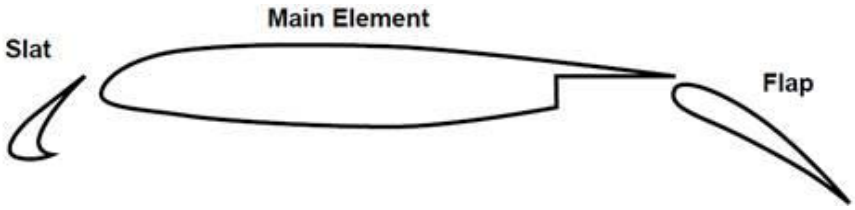


Image 1. MDA 30P-30N Géométrie d'une aile multiélément. Source : DLR

La géométrie proposée est une coupe verticale d'une hypothétique aile d'avion qui a été beaucoup étudiée en soufflerie à petite échelle, composée de 3 éléments. Donc la coupe transforme une aile en 3D en un problème en 2D. On peut créer un maillage avec le mailleur GMSH open source de l'université de Louvain [10] :

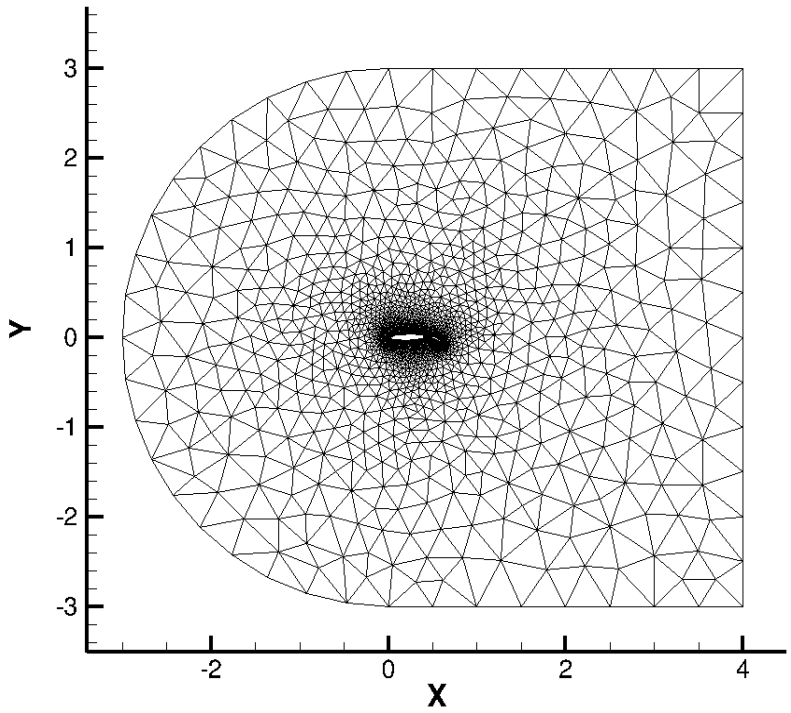


Image 2. Maillage d'environ 4500 nœuds et 8000 éléments triangulaires et raffinement de l'aile multiélément MDA 30P-30N. L'entrée des fluides (air) s'effectue de gauche à droite. Donc, le « slat » est le premier élément

L'image n'est pas très claire, puisque l'aile est très petite en taille par rapport au domaine total du problème. Donc, on va aider avec une représentation de la position du maillage. On rappelle que la partie gauche du maillage (le bord rond) doit être en avant :

Image 3. Coupe verticale d'une aile d'avion. Source: Sipa Press

La méthode numérique (dont on parle à la suite) existe déjà et elle est implémentée dans le programme NextFlow, utilisé dans le département DSNA (Département de Simulation Numérique des écoulements et Aéroacoustique) dans la branche d'investigation Mécanique des Fluides et Énergétique de l'ONERA.

3. NextFlow

NextFlow est un logiciel de recherche utilisé au sein du DSNA, non disponible à l'extérieur, orienté vers le calcul d'écoulements multidimensionnels (2D et 3D) et instationnaires de fluides, qui utilise des maillages non structurés [11]. Le code de calcul est écrit en FORTRAN 90, et il est composé de plusieurs modules correspondant aux étapes de l'algorithme. Par la suite, on va faire une petite explication de la méthodologie du problème, afin de pouvoir exposer plus tard les différents modules et son utilisation.

3.1. Méthodologie

La description du mouvement d'un fluide compressible et visqueux est représentée par les équations de Navier-Stokes, équations aux dérivées partielles du second ordre. NextFlow suppose que le fluide est newtonien et qu'il n'y a ni forces volumiques ni sources de chaleurs au sein du domaine. Pour le calcul, NextFlow utilise une méthode de volumes finis.

Les équations de Navier-Stokes sont :

- Équation de conservation de la masse

$$\partial_t \rho + div(\rho \vec{V}) = 0$$

- Équation de conservation de la quantité de mouvement

$$\partial_t \rho \vec{V} + div\left(pId + p\vec{V} \otimes \vec{V} - \tau\right) = 0$$

- Équation de conservation de l'énergie

$$\partial_t \rho E + div(\rho \vec{V} E + p\vec{V} - \tau\vec{V}) = 0$$

Où :

- $\partial_t = {}^{\partial}/_{\partial t}$
- $\tau = \begin{pmatrix} \tau_{xx} & \tau_{xy} \\ \tau_{yx} & \tau_{yy} \end{pmatrix}$ es le tenseur de contraintes visqueuses en 2 dimensions
- ρ est la masse volumique du fluide
- p est la pression
- $E = e + {}^{\vec{V^2}}/_2$ où e est l'énergie interne et ${}^{\vec{V^2}}/_2$ est l'énergie cinétique
- $\vec{V} = (u, v, w)$ est la vitesse d'une particule fluide

L'équation de conservation de quantité de mouvement peut se réécrire en deux dimensions sous la forme :

$$\begin{cases} \partial_t \rho u + \partial_x(\rho + \rho u^2 - \tau_{xx}) + \partial_y(\rho uv - \tau_{xy}) = 0 \\ \partial_t \rho v + \partial_y(\rho + \rho v^2 - \tau_{yy}) + \partial_x(\rho uv - \tau_{yx}) = 0 \end{cases}$$

On peut réécrire les équations de Navier-Stoker en notation de forme vectorielle :

$$W = (\,\rho \quad \rho u \quad \rho v \quad \rho E\,)^t$$

$$F = \begin{pmatrix} \rho u \\ p + \rho u^2 - \tau_{xx} \\ \rho uv - \tau_{yx} \\ u(\rho E + p) - \tau_{xx}u - \tau_{xy}v \end{pmatrix}$$

$$\mathrm{G} = \begin{pmatrix} \rho v \\ \rho uv - \tau_{xy} \\ p + \rho v^2 - \tau_{yy} \\ u(\rho E + p) - \tau_{yx}u - \tau_{yy}v \end{pmatrix}$$

Comme on a déjà dit, dans NextFlow les équations sont discrétisées sur des maillages non structurés à l'aide d'une méthode volumes finis où les inconnues sont des valeurs moyennes dans les cellules de discrétisation. Écrit sous forme intégrale, par le théorème de Green, le système de Navier-Stokes devient pour un maillage de cellules Ω_i (dans notre cas, cellules triangulaires) :

$$\frac{\partial}{\partial_t}\int_{\Omega_i} W\, d\Omega + \sum_{\Gamma \in \partial\Omega_i} \int_{\Gamma} H(W)\, n d\Gamma = 0$$

Où

- La cellule Ω_i est de frontière $\partial\Omega_\mathrm{i,}$ composé pour faces Γ (3 faces normalement)
- n es la normale extérieure à $\partial\Omega$
- $H(W) = [F(w) \quad G(W)]^t$ est la densité de flux de W

D'une part, le résidu au temps $(n+1)\Delta t$ est calculé en utilisant un schéma décentré en temps implicite précis à l'ordre 3, à quatre niveaux de temps pour approcher l'intégrale de volume, avec les développements limités suivants de W. Soit l'argument $(A) = (x, t + \Delta t)$:

$$W(x,t) = W(A) - \Delta t W_t(A) + \frac{\Delta t^2}{2} W_{tt}(A) - \frac{\Delta t^3}{6} W_{ttt}(A) + O(\Delta t^4)$$

$$W(x,t-\Delta t) = W(A) - 2\Delta t W_t(A) + 2\Delta t^2 W_{tt}(A) - \frac{8\Delta t^3}{6} W_{ttt}(A) + O(\Delta t^4)$$

$$W(x,t+\Delta t) = W(A) - 3\Delta t W_t(A) + \frac{9\Delta t^2}{2} W_{tt}(A) - \frac{27\Delta t^3}{6} W_{ttt}(A) + O(\Delta t^4)$$

En éliminant les dérivées d'ordre supérieur à 1, on obtient la formulation discrète linéaire :

$$(1) \quad \frac{11W^{n+1} - 18W^n + 9W^{n-1} - 2W^{n-2}}{6\Delta t} = \frac{\partial W}{\partial t} + O(\Delta t^4)$$

D'autre part, l'intégrale de surface est représentée par une approximation du flux à partir des valeurs moyennes des champs conservatifs W dans les cellules, par une reconstruction polynomiale du type moindres carrés (expliqué par la suite) :

Donc :

$$Résidu(W^{n+1}) = R(W^{n+1}) =$$

$$\frac{11W^{n+1} - 18W^{n} + 9W^{n-1} - 2W^{n-2}}{6\Delta t} |\Omega_i| + \sum_{\Gamma \in \partial\Omega_i} H((W^{n+1})_{int})|\Gamma|$$

Où

- $R(W^{n+1})$ est le résidu de Wdans le pas de temps physique $n+1$, et on cherche $R(W^{n+1}) \to 0$
- H est la densité de flux normale à la face Γ, d'aire |Γ|
- $(W^{n+1})_{int}$ est une interpolation de W^{n+1} dans les cellules, vers l'interface
- La variable à résoudre est W^{n+1}

La valeur W_{int} est calculée par un schéma décentré « par état » basé sur une décomposition dite « caractéristique », c'est á dire, en respectant le sens de transfert des ondes à travers l'interface de la gauche vers la droite ou inversement (ondes de convection, ondes acoustiques), de la manière suivante :

$$W_{int} = L(R_G W^G + R_D W^D)$$

Où

- $L = R^{-1}$
- R est la matrice des valeurs propres de $\frac{\partial H}{\partial W}$. Pour le calcul de R on utilise *la moyenne de Roe.* Consulter [11] pour plus information.
- R_G est la matrice des vecteurs propres des valeurs propres positives de R et R_D pareil mais pour les valeurs propres négatives

W^G et W^D sont les extrapolées à l'ordre élevé (>2) du champ W en Γ, depuis la gauche et la droite. Pour l'obtention des coefficients d'interpolation linéaire de W^G et W^D sur une interface Γ entre deux cellules, on doit utiliser les valeurs moyennes par chaque cellule et reconstruire la variation spatiale suivant un polynôme qu'on projette ensuite sur l'interface. Mais au-delà de la seule cellule droite ou gauche, on utilise un *voisinage* ou stencil plus vaste, dont la taille dépend du degré de voisinage à travers les faces choisies (degré nombre entier).

Un exemple visuel aidera à comprendre la méthode. Soit le degré de voisinage 2, on va indiquer quel est le degré de voisinage de chaque cellule près d'une face. L'exemple utilise des cellules triangulaires, qui sont les plus habituelles :

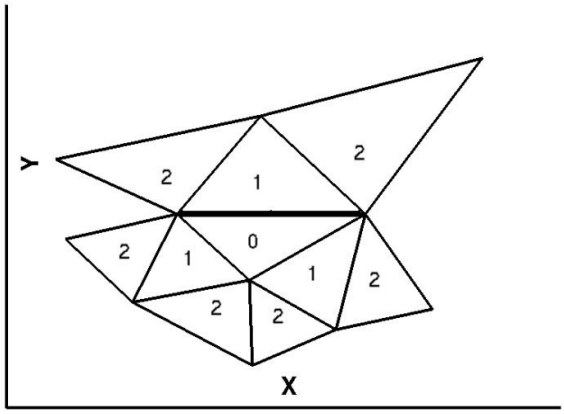


Image 4. Voisinage droite d'une face en 2D

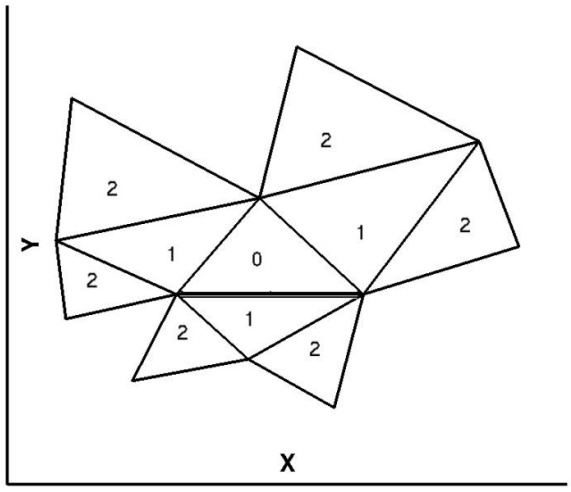


Image 5. Voisinage gauche d'une face en 2D

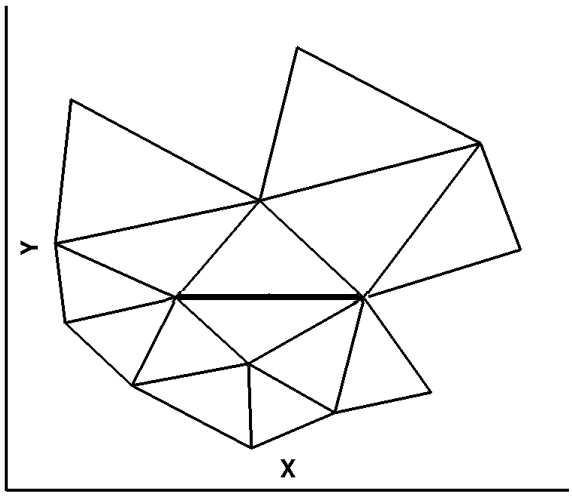


Image 6. Voisinage complète d'une face en 2D

On connait $(W^{n+1})_{int}$ et la première partie du résidu (différences en temps dans l'équation 1), parce qu'elle est linéaire. Mais résoudre le système complet avec :

$$(2) \quad \sum_{\Gamma \in \partial\Omega_i} H((W^{n+1})_{int})|\Gamma|$$

est plus difficile puisque c'est une fonction très non linéaire de (W^{n+1}) . On va voir comment s'y prendre pour la résolution.

Jusqu'ici on avait le concept de pas temps physique, Δt, identique pour toutes les cellules du calcul. La solution qu'on recherche (URANS, Unsteady Reynolds-Averaged Navier-Stokes) est instationnaires y compris pour la turbulence représentée par un modèle en moyenne de Reynolds (K-epsilon). Même en 2D on a besoin d'un tout petit pas du temps Δt.

Pour résoudre l'équation complète avec le terme (2), on doit faire des itérations. Alors, on va utiliser un pas de temps non physique, $\Delta tlocal$, ça veut dire, un pas de temps m différent pour chaque cellule qui optimise la vitesse de convergence $R(W^{n+1,m}) \to 0$. C'est non physique parce que chaque cellule « voyage » dans le « temps » à une vitesse différente, chose pas vraiment réaliste. Mais à convergence, l'effet de ces pas de « temps » locaux disparait et on a obtenu l'égalité entre les termes physiques de l'équation à la fin du pas de temps. Donc :

$$(3) \quad R(W^{n+1,m+1}) = \frac{W^{n+1,m+1} - W^{n+1,m}}{\Delta tlocal}$$

En plus, on va résoudre cette équation avec la méthode de Runge-Kutta de degré K. Alors, on va diviser le pas de temps non physiques en K étapes (chaque étape est notée d) :

$$R\left(W^{n+1,m+\frac{d}{K}}\right) = \frac{W^{n+1,m+\frac{d+1}{k}} - W^{n+1,m}}{\Delta tlocal} =$$

$$\frac{11W^{n+1,m+\frac{d}{K}} - 18W^{n,\infty} + 9W^{n-1,\infty} - 2W^{n-2,\infty}}{6\Delta tlocal} |\Omega_i| +$$

$$\sum_{\Gamma \in \partial\Omega_i} H\left(\left(W^{n+1,m+\frac{d}{K}}\right)_{int}\right)|\Gamma|$$

En déplaçant au second membre, on obtient :

$$(4) \quad W^{n+1,m+\frac{d+1}{k}} = W^{n+1,m} + \Delta tlocal * R\left(W^{n+1,m+\frac{d}{K}}\right)$$

Si on fait un petit rappel, on a parlé de 3 niveaux imbriqués d'itérations différents pour résoudre le problème:

1. Pas de temps physique (N itérations en total)
2. Pas de temps non physique (M itérations en total)
3. Etapes de Runge-Kutta (K itérations en total)

   À partir de ce moment, si on parle des itérations du calcul, on parle de la quantité :

$$N * M * K$$

3.2. Modules

On va décrire l'organisation du programme afin d'établir la fonctionnalité des différents modules et créer un schèma qui nous permettra de mieux comprendre les sections postérieures et les références. Dans cette explication on n'ajoute pas les modules qu'on a créés pendant le projet (messages, binder, etc.) puisque ils vont être mentionnés plus tard.

- Fort : Lecture des donnés. Initialisation des processus MPI (plus tard on expliquera).
- Compute : Module central. Il contrôle le flux de l'exécution et appelle le reste des modules. Il ne fait pas des calcules directement.
- Scheme : Module de préparation. Il alloue les structures, initialise les données, calcule les relations entre les éléments (voisinages, constantes, coefficients d'intégration spatiale, etc.). On ne l'appelle qu'une fois.
- Newtime : Il est chargé de faire des copies des variables à chaque pas de temps physique, de manière que quand on calcule le pas n+1 de l'équation (1) on a accès aux pas n, n-1 et n-2. On l'appelle N fois.
- Newiter : En similarité avec Newtime, il garde pour chaque pas de temps non physique une seule copie des données (on a accès à m quand on calcule m+1 dans l'équation 3). On l'appelle $N * M$ fois.
- Dtloc : Il recalcule pour chaque pas de temps non physique sa valeur locale dans chaque cellule. On l'appelle $N * M$ fois.
- Timeres : On évalue le terme linéaire (équation (1)) à l'aide des variables gardés par Scheme. On fait le calcul pour chaque itération de Runge-Kutta, alors on l'appelle $N * M * K$ fois.
- Fluxes : Le module plus important et plus couteux de programme. Il calcule les fluxes de chaque face, l'équation (2). On l'appelle $N * M * K$ fois.
- Fluxbal : Il calcule les bilans de fluxes, ça veut dire, additionne aux résidus obtenus en Timeres (équation 1) les nouveaux fluxes de Fluxes (équation 2). Alors, on obtient $R\left(W^{n+1,m+\frac{d}{K}}\right)$ un total de $N * M * K$ fois.
- Update : Finalement, on résout l'équation (4), et on obtient le résultat de chaque étape la plus interne Runge-Kutta (appellé $N * M * K$ fois).

  Output : Quand on finit les calculs on garde les résultats dans des fichiers pour les visualiser avec l'aide de Tecplot.

On montre un organigrame qui aidera :

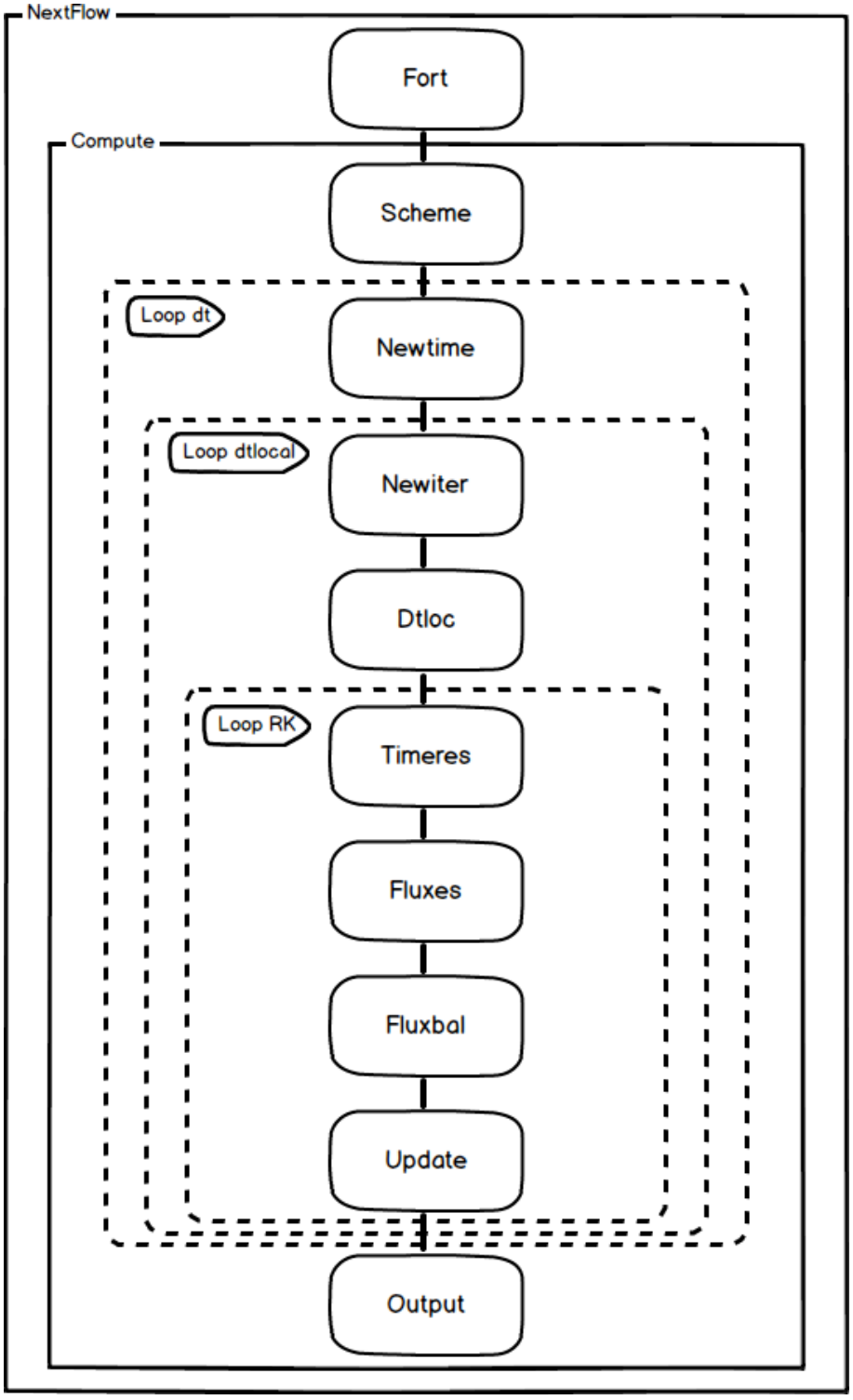


Image 7. Organigrame par modules et itérations du programme NextFlow avant des changements réalisés par le projet

4. Une nouvelle dimension

On a vu que le problème en 2D (axes X et Y, longueur et largeur) est une coupe verticale d'une aile d'avion, avec un maillage qui représente le domaine par des cellules triangulaires planes avec 3 faces chacune. On peut voir ces cellules triangulaires si on fait un zoom dans le maillage proposé, et en plus noter comment les cellules sont plus petites dans la zone proche à l'aile, puisque c'est un maillage raffiné :

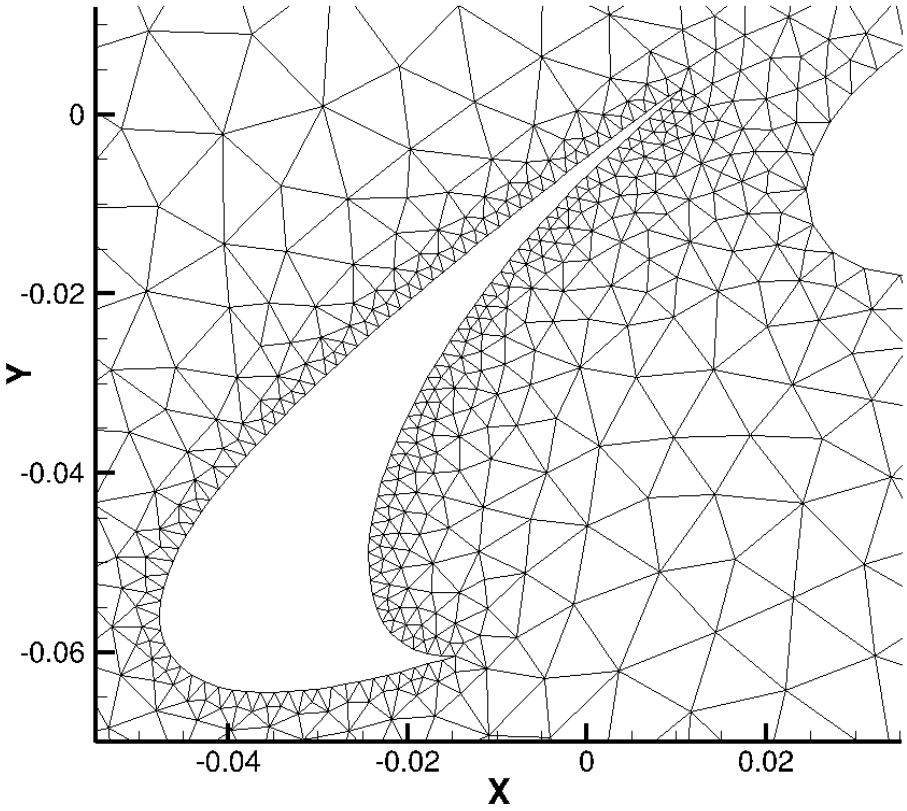


Image 8. Zone antérieure de l'aile d'avion dans le maillage proposé

On va étendre ces cellules à une troisième dimension (axe Z), mais de façon homogène, ça veut dire, avec une hauteur fixe pour toutes les cellules, de sorte que les cellules deviennent des prismes de base triangulaire, avec 5 faces chacune. Alors, les cellules volumiques situées le long d'un même axe Z son strictement similaires, avec des surfaces et des volumes de même taille exactement, de manière que les résultats puissent être étendues à des phénomènes physiques 3D, mais que la complexité reste acceptable. Par exemple, on peut choisir une hauteur de chaque cellule *deltaz* = 0,004 fois la longueur de l'aile, avec 256 cellules dans l'axe Z (cette quantité va être appelé *nspan*).

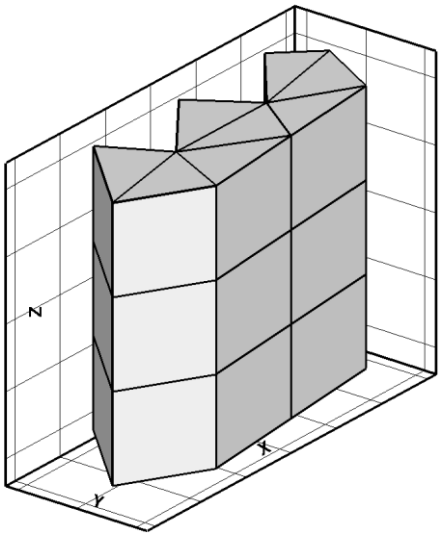


Image 9. Cellules-prismes d'hauteur proportionnelle

Cette nouvelle dimension homogène est considérée comme une fausse dimension ou dimension incomplète, pour cela on va parler de 2,5D, et non 3D. Dans une représentation 3D, chaque hauteur devrait être choisie en raison des raffinements du maillage, de même manière que selon les axes X et Y, où les cellules proches de l'aile ont des surfaces plus petites (Image 11).

Alors si en 2,5D on prend une hauteur pour créer des prismes « réguliers » proches de l'aile (Image 10), on va trouver des cellules éloignées avec des bases disproportionnées par rapport aux hauteurs, mais c'est un problème dont on ne va pas se préoccuper, puisqu'on est concerné par les structures turbulentes près de l'aile, où les cellules sont régulières, et les cellules très aplaties en s'éloignant n'ont pas d'effet sur la précision du calcul près des ailes.

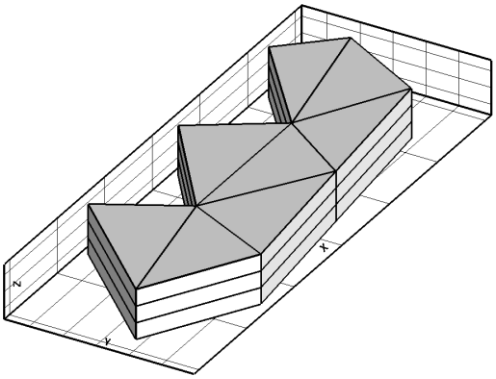


Image 10. Cellules-prismes aplaties

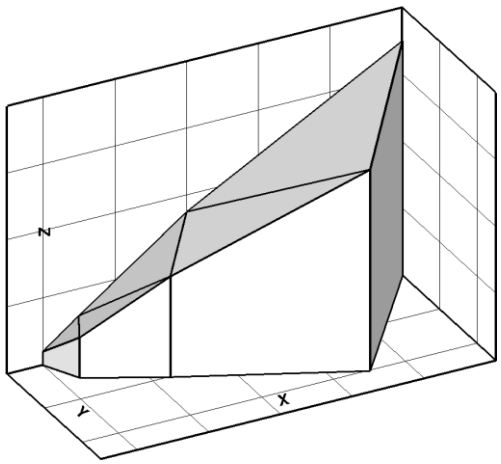


Image 11. Cellules d'hauteur raffinée

4.1. Changements appliqués

Le code initial de notre projet est NextFlow, qui résout les problèmes précisés dans la section 3 en 2D avec l'utilisation des variables physiques, conservatives, fluxes, résidus, etc. On va expliquer quels sont les changements plus importants effectués, parfois d'un point de vue logique, parfois pratique.

4.1.1. Changements pratiques

Ce sont les changements touchant le code, le nettoyage et plus proches du langage:

- Structures : Avant, on avait toutes les données représentées pour des tableaux de variables scalaires. Chaque variable était une matrice d'information, chaque information correspondant à une cellule différente. Maintenant on a créé certaines structures qui rassemblent les variables « ro », « u », « v », « w », etc., correspondant à une cellule. Chaque structure correspond donc à une cellule, et chaque variable est un vecteur, où chaque point correspond à un z dans la nouvelle dimension, donc un vecteur de taille *nspan*. Ainsi, toutes les opérations qui étaient effectuées dans chaque cellule seront effectuées de manière identique sur toute la nouvelle dimension. Et ceci est valable pour les nœuds, cellules, faces, surfaces, fluxes, etc.

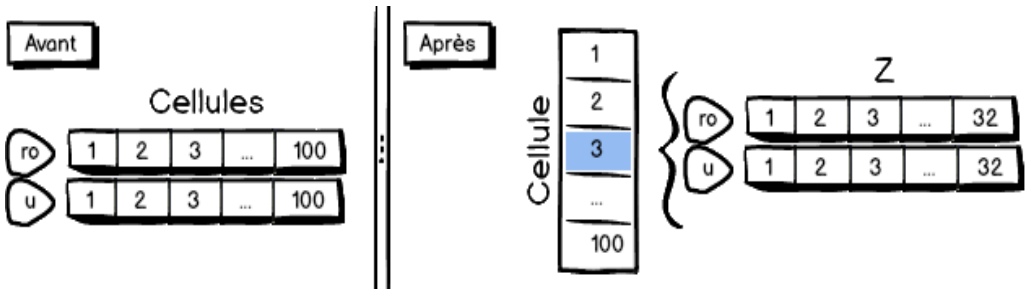


Image 12. Exemple de changement des structures

- Faces: Avant, on avait 3 faces (arêtes du triangle) pour chaque cellule, mais maintenant on a 5 pour chacune : 3 faces latérales (auparavant les arêtes des triangles sans surface), 1 base en haut et 1 base en bas. On va établir qu'on a besoin seulement d'une nouvelle face pour chaque cellule, la base en haut, puisque la base en bas est déjà calculée pour la cellule en bas. Alors, si on avait une quantité de 200 cellules avec 300 faces, maintenant on aura 500. Pour faciliter la numérotation, on a établi que les premières 200 structures faces correspondent aux bases des cellules, de forme que le numéro identificateur de cette face est le même que celui de la cellule (et c'est implicite), et les autres faces sont « décalées». Il existe déjà un « schèma » qui permet de connecter chaque cellule avec ses 3 facettes, qui doit seulement être modifié en ajoutant 200 (le nombre de cellules) à chaque identificateur.

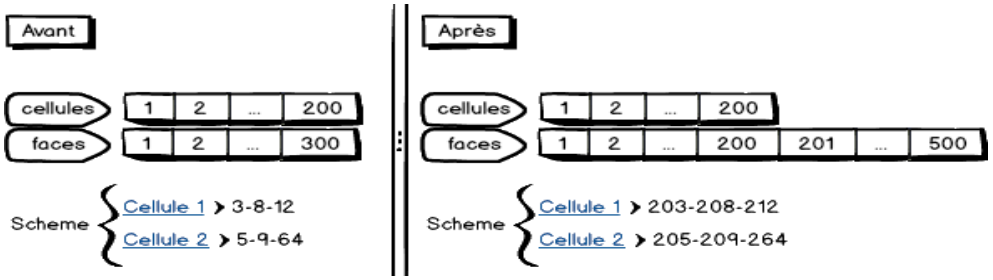


Image 13. Organisation des structures « faces » : exemple des décalages des faces

4.1.2. Changements logiques

Ce sont les changements touchant la nouvelle dimension, d'une manière théorique.

- Voisinage : Avant, pour certains calculs on avait besoin de calculer les voisinages droite et gauche d'une face, selon qu'on commence à calculer pour la cellule les interpolations spatiales à droite ou à gauche [12]. Les voisinages étaient étendus selon un degré de voisinage D, d'une forme générique, une face a dans son voisinage (ou stencil) toutes les cellules auxquelles elle peut arriver dans D pas (chaque pas consistant à traverser une arête entre deux cellules et où le pas 0 es la première cellule). Par le même fonctionnement, maintenant ces pas peuvent se faire à travers des arêtes en bas et en haut. Alors, le nouveau voisinage peut aller en toutes les dimensions de la manière suivante : le niveau N en vertical a un degré D-N en horizontal, pour un N entre 0 et D-1. On va montrer quelques exemples.

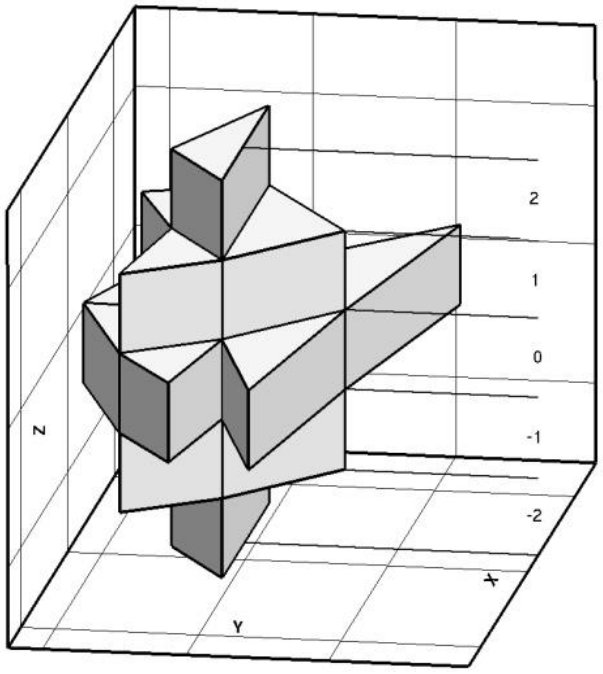


Image 14. Voisinage gauche d'une face en 2,5D (voir image 4)

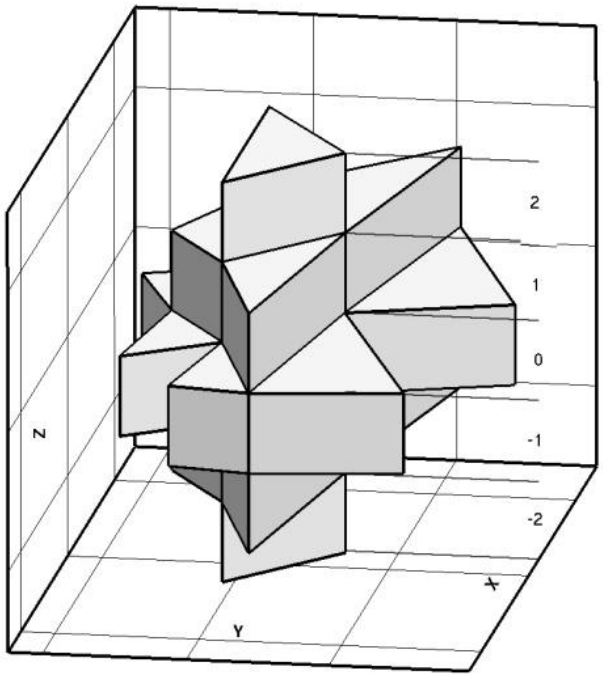


Image 15. Voisinage gauche d'une face en 2,5D (voir image 5)

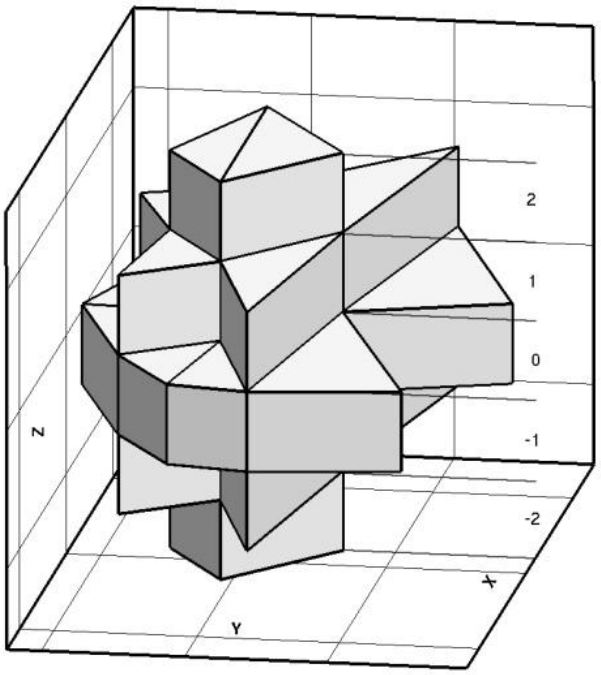


Image 16. Voisinage complet d'une face en 2,5D (union des voisinages gauche et droit: voir image 6)

- Ordre de faces : On ajoute les concepts de face « intérieure » et de « condition limite » ou « CL ». Les structures face sont ordonnées de façon qu'au début on trouve les bases des prismes qui sont toujours intérieures au domaine, après les anciennes arêtes qui ne sont pas les bords du domaine et finalement les arêtes qui sont les bords appelées CL. Les nouvelles bases ne sont pas CL parce qu'on cherche calculer l'axe Z cycliquement, pour permettre le développement des structures turbulentes dans la direction Z : la base en bas de la première cellule est la base en haut de la dernière, alors les fluxes qui sortent de l'une entrent dans l'autre.

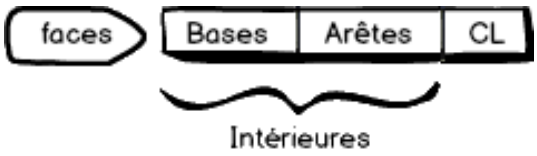


Image 17. Ordre des structures faces

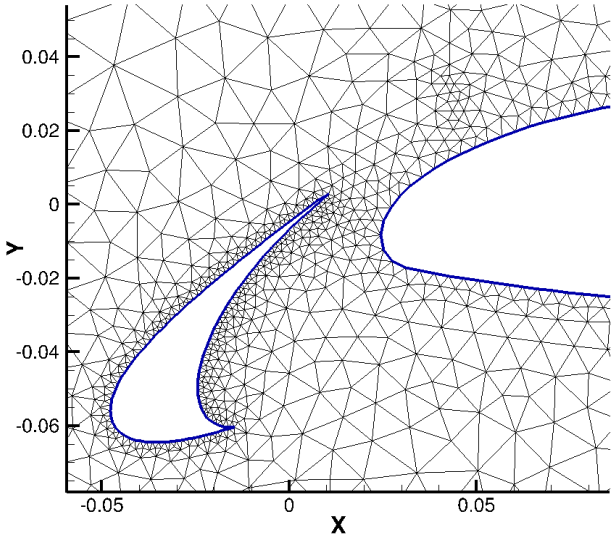


Image 18. Maille avec les bords CL en bleu

4.2. Résultats

Après avoir implémenté ces changements, on a exécuté le calcul avec un maillage d'autour 1500 nœuds, 2500 cellules et 3500 faces et 32 positions dans l'axe Z, éloignées de *dz* = 0,004 (longueur de l'aile égale à 1). Le temps de calcul reporté correspond à 6 étapes Runge-Kutta ($N * M = 6$). Le type de données est *float* (4 bytes). On a vérifié pour ce cas préliminaire que si l'initialisation était la même sur chaque z sur l'axe Z, les résultats doivent être indépendants de z, quelle que soit la position dans le plan XY.

Nspan	Mémoire (MB)	Temps (secondes)
1	6	22,17
32	38,44	544,98
Rapport	**640%**	**2458%**

On peut bien voir que l'augmentation de temps est énorme, 25 fois plus, puisque bien que pour chaque cellule maintenant on a 32 fois plus de données à calculer. Pour la place mémoire, les faces n'augmentent que de 70% (de 3500 à 6000), et il y a beaucoup d'autres données (nœuds, variables, constantes, coefficients du schéma spatial) qui n'augmentent pas avec la taille dans la 3ème dimension. On voit un effet important de traiter des variables vectorielles plutôt que des scalaires.

On peut faire une remarque très importante : pour le passage de 2D en 2 ,5D, on n'a du changer que les phases d'allocation et les boucles, les instructions de calcul sont absolument identiques dans la programmation qui utilise les structures ($ro(i)$ est devenu $cell(i)\%ro$) avec ro un scalaire en 2D et un vecteur de taille *nspan* et 2,5D). C'est le compilateur Fortran qui interprète que les calculs sont effectués de manière identique pour tous les indices des vecteurs en 2,5D.

On constate donc que le calcul sur les vecteurs en Fortran est 1,28 fois plus rapide que sur les scalaires, sans aucune réécriture de code autre que la définition des structures, on obtient ainsi une optimisation séquentielle déjà très performante pour notre objectif de calcul 3D des structures turbulentes qui sont la source de bruit.

Ces résultats sont obtenus sur un seul cœur de calcul, on va mettre en place 2 niveaux de calcul parallèle pour l'exécution multi-cœur : OpenMP (existe déjà dans la version de départ de NextFlow) et MPI.

4.3. Optimisation OpenMP

On va utiliser l'outil Open Multi-Processing [13], une interface de programmation pour le calcul parallèle à mémoire partagée, au niveau interne des boucles de calcul par nœud, par face, par cellule. Avant toutes ces boucles du calcul on décrit le contexte OpenMP par des directives commençant par $C\$OMP$. On doit établir quels sont les variables partagées (*shared*) entre tous les processus et lesquels sont privées (*private*), et le nombre de processus (threads de calcul) qui vont travailler en parallèle en se partageant entre eux les indices de la boucle (ici 12 threads pour un nœud de 2 processeurs Westmere).

On peut voir un petit exemple avec les structures de cellules. Pour chaque cellule (numCell en total) on a l'indice du volume (ivf0) et on veut faire deux copies à la suite (ivf1, ivf2), chaque copie 50% plus grande : coefficient 1'5. La copie pour chaque thread du constant est *private*, le vecteur volume est partagé.

```
Integer :: constant
constant = 1,5
C$OMP   PARALLEL
C$OMP1  DEFAULT(PRIVATE)
C$OMP2  SHARED(cell, numCell,constant)
do ivf0 = 1, numCell
        const_copy = constant
        ivf1 = ivf0 + numCell
        ivf2 = ivf1 + numCell
        cell(ivf1)%vol = const_copy * cell(ivf0)%vol
        cell(ivf2)%vol = const_copy * cell(ivf1)%vol
enddo
C$OMP   END PARALLEL DO
```

Malgré la difficulté de l'implémentation, à cause des boucles complexes, avec une grande quantité des variables partagées et privées, les résultats sont prometteurs. Si las variables sont partagées, on trouve des problèmes d'accès concurrent, où les processus doivent s'attendre les uns les autres; si elles sont privées, chaque processus à une copie, alors la charge de mémoire augmente.

Nspan	**Temps (secondes)**
32	544,98
32 + OpenMP	62,95
Rapport	**12%**

Le temps a été accéléré presque 9 fois. Mais on va chercher aussi une optimisation plus centré en améliorer les opérations sur les vecteurs de l'axe Z que sur les boucles directes. La solution vient avec les optimisations de *gcc* à l'heure de compilation: -O3, qui permet tous les optimisations automatiques de *gcc* [14]. Parmi eux, unifier variables avec même valeurs, modifie sautes inutiles dans le code, optimise opérations couteux, etc. Mais il y a certaines optimisations qui concernent les résultats numériques, et on ne peut pas les faire sans attention. Alors, on doit ajouter aussi *-fp-model strict*, qui active la valeur sûre pendant les opérations avec données de virgule flottante (*float*, *double*), pour ne pas perdre une quantité importante de précision.

Nspan	**Temps (Secondes)**
32 + OpenMP	62,95
32 + OpenMP + O3	10,13
Rapport	**16%**

Et finalement, le temps et accéléré encore plus de 6 fois.

4.4. Conclusion

On a ajouté une nouvelle dimension avec certaines conditions, de cette manière on est passé d'un domaine en 2D à celui en 2,5D. Dans le chemin on a nettoyé le code, changé la logique et crée de nouvelles structures.

On a obtenu un agrandissement de charge de mémoire d'un facteur 6,4 (5 fois moindre que l'agrandissement du domaine réel sur *nspan* = 32). D'autre part, on a pu restreindre l'augmentation de temps de calcul d'un facteur 25 à le diminuer un facteur 2 grâce à OpenMP et aux optimisations automatiques du compilateur *gcc*.

Le bilan de cette partie est satisfaisant, mais on ne doit pas oublier qu'on n'a travaillé qu'avec un maillage assez simple, seulement 2500 cellules et *nspan* = 32. Notre objectif est beaucoup plus ambitieux, avec des maillages 2D composés 200 000 à 500 000 de cellules et, au minimum, un axe Z 16 fois plus grande, donc au total 256 000 000 de variables par équation. On doit commencer à explorer d'autres outils pour le calcul parallèle massif et des hardwares additionnels.

5. Outils : Partition du domaine

On doit se préoccuper d'un souci pas banal : la taille du domaine. En extrapolant par rapport à l'exemple précédent, si on passe de 2500 cellules (1400 nœuds) à 500 000 et d'une axe Z de *nspan* = 32 à *nspan* = 512, on pourrait se trouver avec un temps d'exécution de plusieurs heures pour une seule itération de pas de « temps » local. Il faut en faire plusieurs dizaines de milliers pour obtenir une bonne estimation des intensités de fluctuation turbulente des composantes de la vitesse. Mais, encore plus important, quelle mémoire peut-on utiliser? Si 6MB correspondent à 2500 cellules et 32 cotes *nspan*, 0,5 millions de cellules et *nspan* = 512 utilisent 120GB.

Un système conventionnel va être bien trop peu performant avec ce type de tailles. Un processeur Intel Westmere (le type utilisé à l'ONERA) a un cache interne de 12MB, largement insuffisant pour gérer cette tâche, ce qui provoque de très nombreux accès à la mémoire RAM centrale, avec la perte de temps que cela représente [12].

On doit explorer une autre option: le parallélisme de tache ou de processus, en mémoire distribuée sur plusieurs hardwares C'est la seule façon d'accéder à une taille mémoire très importante.

Le but est de diviser le domaine pour que chaque partition soit traitée de manière indépendante, sur différents hardwares de manière *asynchrone*, avec une quantité minimale de communication entre eux. On veut utiliser différents ordinateurs dans un réseau uniquement liés par une communication « externe », sans mémoire partagée, ce qu'on appelle *mémoire distribuée*.

5.1. Préprocesseur

Pour la division de domaines on va utiliser le partitioneur de graphe de maillage, METIS [15], intégré à GMSH. Un préprocesseur spécifique, intégré à NextFlow va lire toutes les données du maillage 2D et le fichier de connectivité HigLift.cnc. Dans ce fichier on peut trouver les données suivantes :

- Résumé : quantité total de nœuds, cellules et faces. Division des cellules et faces pour différents types : internes et conditions limites (glissement, adhérence et open) [11].
- Nœuds : Chaque nœud par ordre d'identificateur global, avec les 3 coordonnées (où la coordonné Z est toujours 0).
- Nœuds par cellule : l'identificateur global de chaque cellule suivie pour les identificateurs globaux des 3 nœuds qui le forment. Les dernières cellules sont les CL.
- Faces par cellule: l'identificateur global de chaque cellule suivies pour les identificateurs globales des 3 faces qui le forment.

- Cellules par face : l'identificateur global de chaque face suivies pour les identificateurs globales des 2 cellules qui sont à coté droite et gauche. Les dernières faces sont les CL, par l'ordre décrit avant : faces glissés, faces adhérentes et faces open.

Le préprocesseur reçoit quelques paramètres, dont la quantité des partitions. En faisant un calcul approximé, il commence l'adjudication des cellules aux différentes partitions, de façon que chaque partition aille une quantité similaire des cellules et que les partitions soient connexes.

Le préprocesseur reçoit quelques paramètres, dont la taille des partitions, et il effectue une renumérotation des cellules pour que chaque partition soit formée de cellules d'indices successifs. Ensuite, il identifie dans chaque partition quelles sont les cellules qui sont nécessaires pour calculer l'évolution des champs de variables dans chaque autre partition (zone appelée « Ghost », de part et d'autre de chaque frontière entre 2 partitions, dont on décrira le détail ultérieurement).

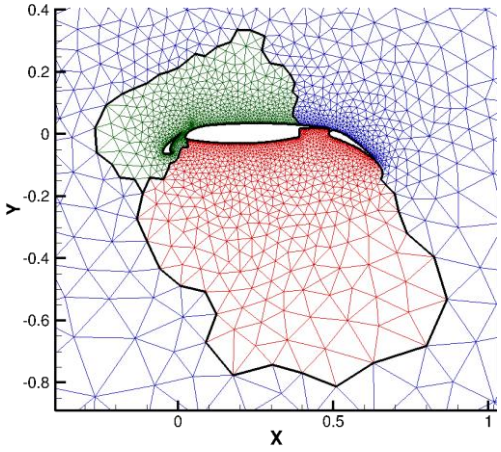


Image 19. Maille avec 3 partitions (bleu, rouge, vert) délimitées par noir

En plus, les différents éléments vont être « renumérotés », alors, ils vont recevoir un nouvel identificateur local, en raison de sa nouvelle partition.

Finalement, le préprocesseur va nous donner un fichier pour chaque partition, appelé HigLift_prtXXX.cnc, où XXX sont 3 chiffres avec le numéro de la partition. Le format de ces fichiers est similaire à celui de l'original, mais ici on a plus informations :

- Résumé : quantité total de nœuds, cellules et faces. Division des cellules et faces pour différents types : internes, CL et *ghost*.
- Nœuds : Chaque nœud par ordre de identificateur local, avec les 3 coordonnées (où la coordonné Z est encore 0).
- Nœuds par cellule : l'identificateur local de chaque cellule suivie pour les identificateurs locaux des 3 nœuds qui le forment et la partition auquel laquelle il appartient. L'ordre est internes, CL et *ghost*.
- Faces par cellule: l'identificateur local de chaque cellule suivies pour les identificateurs locales des 3 faces qui le forment.

- Cellules par face : l'identificateur global de chaque face suivies pour les identificateurs locales des 2 cellules qui sont à coté droite et gauche. L'ordre est internes, CL et *ghost*.
- Messages : Information sur les cellules *ghost*, dont la raison de son existence on va expliquer ensuite.

5.2. Cellules *ghost*

On a expliqué déjà que pendant les calculs on fait des interpolations pour certains valeurs dans les faces, généralement dans la forme d'une moyenne entre les valeurs des cellules qui entourent cette face (gauche et droite). Les 14 valeurs par cellule utilisées pour faire ces moyennes sont celles contenues dans la structure cellule : densité (ro), vitesse (3 directions : u, v, w), énergie totale (eto), énergie de turbulence (tke), taux de dissipation de l'énergie (eps), production de l'énergie (prodke), pseudo-pas du temps (dtloc), pseudo-masse volumique (r), pression (p), température (temp), viscosité dynamique (vis) et volume de cellule (vol).

Afin de faire cette interpolation de manière plus précise, on n'utilise pas seulement deux cellules, mais un stencil gauche et droit avec la méthode de voisinage, déjà commenté. Alors, pour chaque stencil on a plusieurs cellules proches à utiliser, et on aura besoin de calculer d'avantage les coefficients d'interpolation pour les cellules. Jusqu'ici aucun souci.

Mais des problèmes apparaissent avec des partitions indépendantes, si une partie du stencil appartient à une autre partition. Pour gérer ces interpolations, on utilise comme solution les cellules *ghost*.

Le préprocesseur augmente le domaine de chaque partition avec les cellules qui appartiennent aux autres partitions mais dont on a besoin parce qu'elles font partie d'un stencil d'une face interne ou CL (qui incluent les faces frontière entre 2 partitions). On a besoin de créer cellules et faces *ghost* afin de les remplir avec les données nécessaires pour l'interpolation, mais on ne va pas faire des calculs pour résoudre les équations de Navier-Stokes sur ces cellules Ghost, parce que chaque cellule *ghost* a une correspondance avec une cellule réelle (interne ou CL) dans une autre partition, et les calculs vont avoir lieu là-bas.

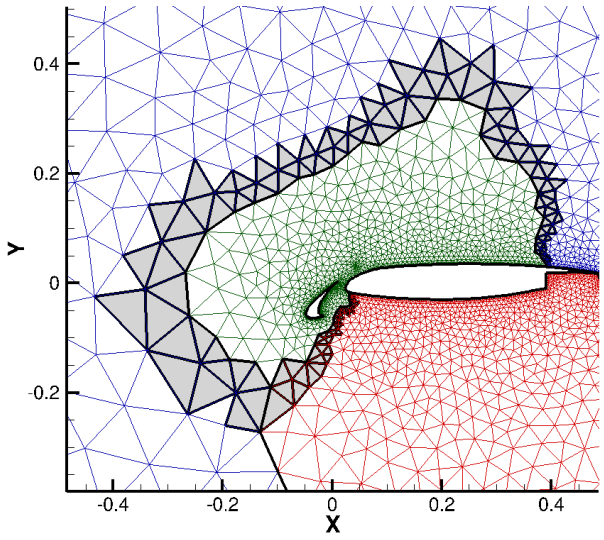


Image 20. Maille avec la partition vert centrée avec ses cellules *ghost* sur la partition bleu sont plus obscures

Les cellules et faces *ghost* sont numérotées localement, et sont ajoutées au final des structures pour les contrôler plus effectivement. On va montrer un exemple d'un domaine HigLift de 4203 nœuds, 6982 cellules et 9962 faces, dont 511 CL, découpé en 3 partitions :

Partition	Nœuds	Cellules	Internes CL *Ghost*	Faces	Internes CL *Ghost*	Glissement Adhérence Open
prt001.cnc	1444	2474	2157	3506	3175	0
			166		166	166
			151		165	0
prt002.cnc	1421	2499	2157	3578	3209	0
			119		119	71
			223		250	48
prt003.cnc	1546	2549	2157	3553	3148	0
			226		226	226
			166		179	0
Total	**4411**	**7522**	**6471**	**10637**	**9514**	**0**
			511		**511**	**392**
			540		**594**	**48**

Alors, on a 207 nœuds, 540 cellules (les *ghost*) et 675 faces supplémentaires. Mais pour les faces *ghost* on n'a que 594, alors, il y a 81 faces en plus créées pour une raison très simple : ce sont les faces qui divisent les partitions, qui ont la cellule droite et la gauche en différentes partitions (on a parlé d'elles avant).

Pour nous faire une idée plus exacte, on peut voir l'image à la suite, qui représente les partitions et les cellules *ghost* entre elles. On ne doit pas oublier que les cellules *ghost* sont cellules supplémentaires dans une partition, qui est originalement dans une autre, et pourtant elles sont représentées au-dessus de cellules réelles.

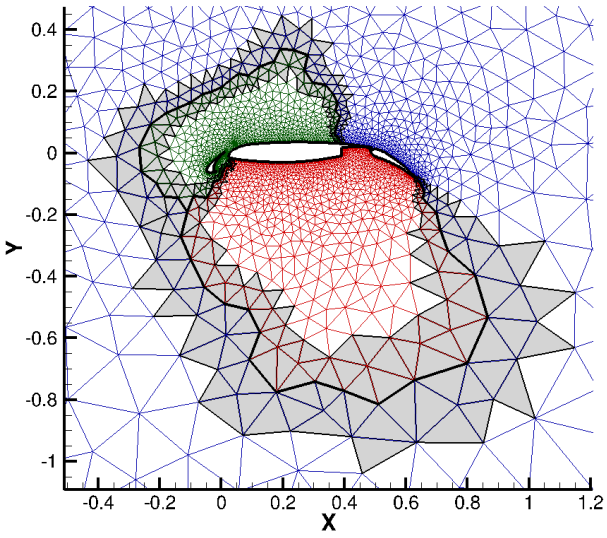


Image 21. Maille avec 3 partitions (bleu, vert, rouge) et les ghost plus obscures

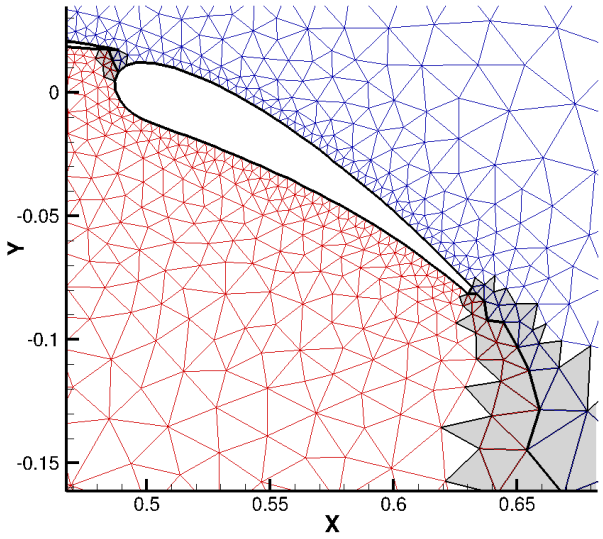


Image 22. Maille avec 3 partitions (bleu, vert, rouge) et les *ghost* plus obscures

5.3. MPI

On a les cellules *ghost*, mais on doit remplir chacune avec les 14 données physiques et conservatives de la cellule correspondant dans une autre partition. Alors, on doit avoir une communication bidirectionnelle entre tous les processus qui ont une frontière commune. Si A et B sont des partitions voisines, A va envoyer un message avec ses cellules à B afin que lui reçoit le message et remplisse ses *ghost*. Obligatoirement, A devra recevoir un message de B avec les données pour ses cellules *ghost*. Si les deux ont une frontière commune, les deux auront cellules *ghost* correspondants dans l'autre partition.

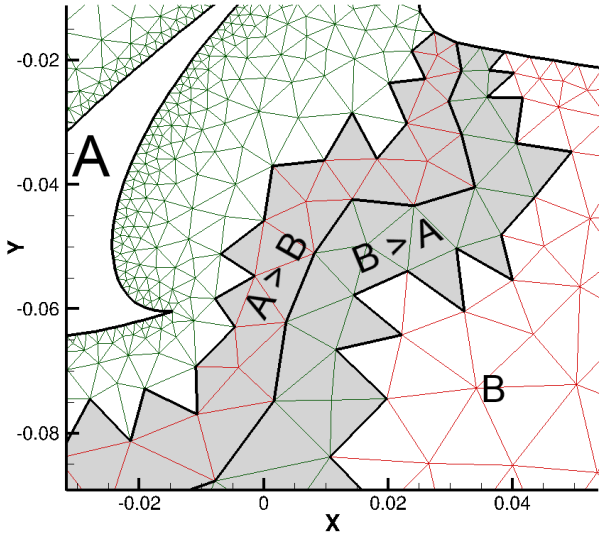


Image 23. Maille avec 2 partitions (A, B) et les cellules *ghost* selon la direction du message (>)

On va gérer cette communication avec l'outil MPI (*Messaging Passing Interface* [16]), une interface qui permet d'exploiter des hardwares distants (soit machines différentes, soit processeurs dans une même machine) avec passage des messages. MPI est le standard de communication le plus étendu pour exécutions parallèles sur systèmes de mémoire distribuée.

Grâce à MPI on va résoudre les deux soucis génères pour l'apparition des partitions du domaine : comment exécuter le programme parallèle en différents partitions et comment communiquer les données *ghost* nécessaires.

MPI dispose des fonctions pour envoyer et recevoir messages d'un autre processus [17], donc on doit connaître qui sont les partitions voisines. En plus, le message n'est qu'un vecteur dont on doit préciser le type de données et la taille, quand on envoie et on reçoit. C'est insuffisant pour connaître la destination des données dans le maillage de la partition qui le reçoit. On doit se mettre d'accord sur l'organisation des données dans ce message.

Alors on a besoin de :

- Les partitions voisines
- L'identificateur local des nos cellules *ghost* à recevoir pour chaque voisine
- L'identificateur local de nos cellules à envoyer pour chaque voisine
- La structure des messages pour les construire et interpréter

Et cette information est connue uniquement par le préprocesseur. Alors, il va ajouter à la fin de chaque fichier un paragraphe « messages » où on pourra lire les quantités et les identificateurs.

Par exemple, à la fin du fichier HigLift_prt001.cnc, on continue avec le même exemple que dans le point antérieur, on trouve:

```
Messages
(      2324        1 ) <=== (      1223        2 ) reception
(      2325        1 ) <=== (      1337        2 ) reception
(      2326        1 ) <=== (       332        2 ) reception
Reception depuis       2       98
(      2422        1 ) <=== (      1847        3 ) reception
(      2423        1 ) <=== (       938        3 ) reception
(      2424        1 ) <=== (       255        3 ) reception
Reception depuis       3       53
(       700        1 ) ===> (      2398        2 )  emission
(       915        1 ) ===> (      2399        2 )  emission
(       321        1 ) ===> (      2400        2 )  emission
Emission vers          2      102
(       840        1 ) ===> (      2384        3 )  emission
(       528        1 ) ===> (      2385        3 )  emission
(       855        1 ) ===> (      2386        3 )  emission
...
Emission vers          3       52
```

On connaît les voisines, les tailles du message et en plus en ordre ascendant pour les identificateurs locales des cellules *ghost*. Ces identificateurs on les connaissait déjà, mais maintenant on peut lire, dans le cas d'envoyer, les identificateurs locaux qu'on doit préparer ; dans les cas de recevoir, la quantité totale et la certitude qu'on va recevoir les cellules en ordre ascendante simple.

La structure de messages va être simple : pour chaque cellule on envoie en ordre les 14 informations nécessaires. Mais on ne peut pas oublier qu'à cause du 2,5D chaque cellule est composée d'une quantité *nspan* des valeurs. Alors, dans ce cas vraiment on va envoyer pour chaque cellule $13 \times nspan + 1$ données (la donnée *vol* n'est pas de taille *nspan* puisque le volume est toujours constant dans l'axe Z grâce à la dimension 2,5D). Ces messages vont être d'une taille assez petite par rapport à la quantité de mémoire traité. En plus, MPI n'a pas une limite de taille réelle dans ses messages et la taille où il est recommandable de diviser ces messages en *tokens* (messages de taille moindre) ne se rencontre jamais. [18]

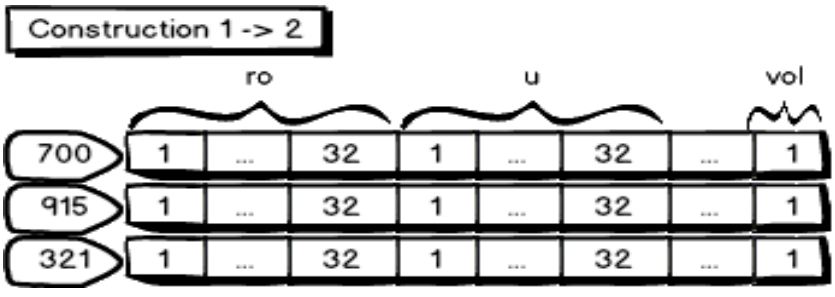


Image 24. Construction de la matrice du message de la partition 1 à partition 2, avec chaque ligne une cellule intérieure et pour chaque ligne 13 données de taille *nspan* = 32 et le donnée vol de taille 1

Pour la réception, rien plus facile : la partition qui a reçu doit seulement copier chaque cellule et chaque donnée à l'adresse correspondante :

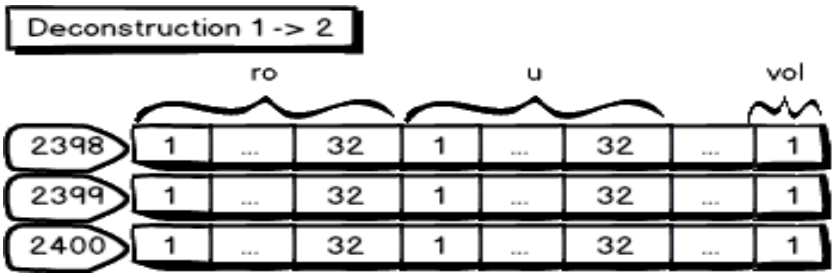


Image 25. Déconstruction de la matrice du message de la partition 1 à partition 2, avec chaque ligne une cellule *ghost* et pour chaque ligne 13 données de taille *nspan* = 32 et le donnée vol de taille 1

La construction et déconstruction sont détaillées, mais la communication directe peut être abordée des plusieurs manières. L'idée de MPI est avoir des partitions le plus autonomes possibles, qui peuvent se trouver dans différents points de l'exécution, ce qu'on appelle *asynchrone*. Mais toute communication confère un point de synchronisation, puisque si on va recevoir un message, sans doute on doit l'attendre.

Pour minimiser ce souci, on va utiliser les variantes asynchrones de MPI [19]. Ces fonctions permettent de lancer l'envoi des messages et de continuer avec d'autres tâches sans attendre la confirmation du processus récepteur, et de même manière lancer la réception et continuer même si le message n'a pas vraiment arrivé.

Quand un processus MPI arrive dans la section des messages, le premier acte est de lancer la réception des messages, MPI_Irecv, mais il ne va pas attendre la réception, il va continuer en envoyant ses propres messages. De cette manière, le système interne de communication de la machine va être active automatiquement pour recevoir les messages, même si le processus ne les veux pas interpréter immédiatement.

L'envoi est fait de manière similaire, MPI_Isend les envoie mais on n'attend aucune confirmation, le processus passe directement à déconstruire (interpréter) les messages reçus. Ici il doit faire une confirmation réelle de la réception, puisque on pourrait traiter un message vide. Alors, ce point est le moment de synchronisme qu'on ne peut pas éviter : sans que tous les messages soient arrivés, on ne peut pas continuer. On doit utiliser la fonction MPI_Test. Pour minimiser le temps d'attente, comme chaque processus attend plusieurs messages, il va chercher en boucle tous les messages qu'on doit recevoir pour déconstruire les messages dans l'ordre dans lequel ils arrivent (on utilise la populaire méthode de FIFO, acronyme de *First in, First Out*, on traite les messages en ordre chronologique*)*. Donc, le processus n'est jamais bloqué. Bien sûr, si l'autre processus est très lent, il devra attendre son envoi, mais dans ce temps il a déjà déconstruit tous les autres messages.

Avec l'utilisation des fonctions asynchrones on cherche à gagner du temps, mais surtout éviter l'interblocage (ou *deadlock* [20] en anglais), en anglais), phénomène trouvé en programmation concurrente dans laquelle deux processus s'attendent mutuellement. Dans ce cas, si les deux processus sont bloqués en attendant la réception du message de l'autre, ils ne vont jamais sortir de cet état, en provoquant une situation catastrophique.

On montre un organigramme du module de communication MPI :

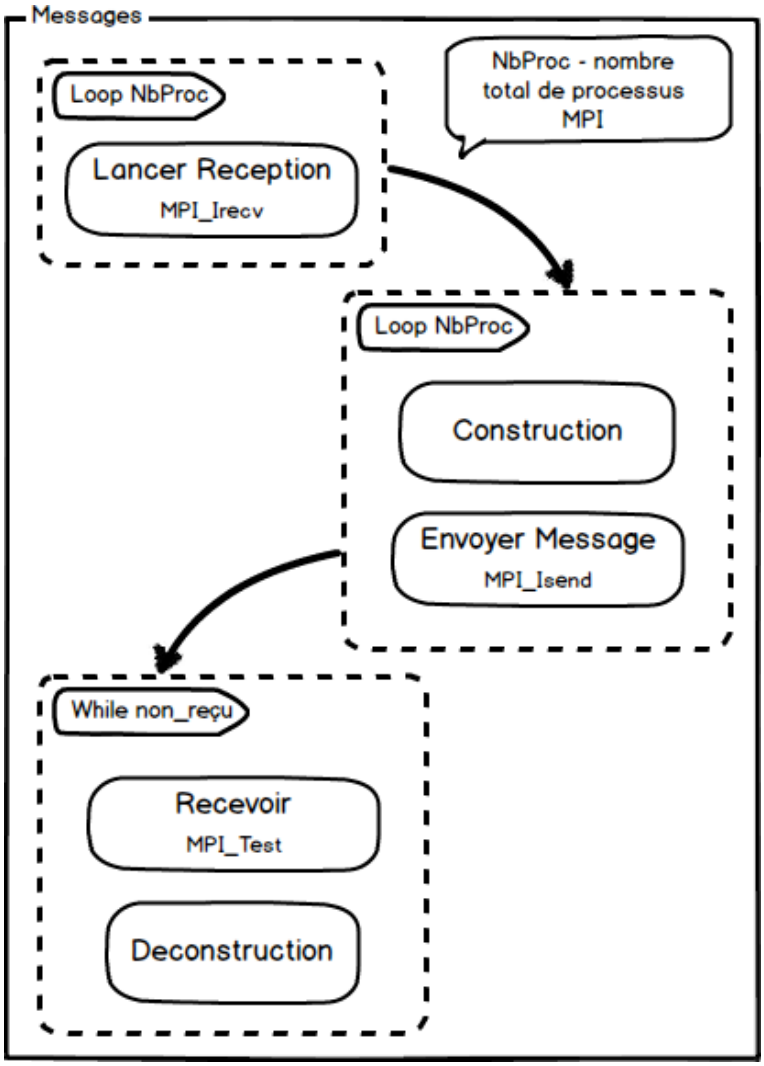


Image 26. Organigramme du module de messages avec fonctions asynchrones

5.4. Résultats

On va montrer plusieurs exemples par rapport au temps et espace, et analyser vraiment si la parallélisassions MPI est utile. Il faut noter qu'on avait 12 threads OpenMP pour les tests, donc on a divisé la quantité des threads entre les processus. On continue avec l'exemple de 4203 nœuds et *nspan* 32, avec une, trois ou six partitions :

Itérations	100		
Partitions	1	3	6
Calcul (s)	17,35	12,95	10,89
Messages (s)	-	0,92	0,76
% messages	-	7,10%	6,98%
Rapport	-	75%	63%

Itérations	300		
Partitions	1	3	6
Calcul (s)	50,55	40,76	31,77
Messages (s)	-	3,54	3,12
% messages	-	8,68%	9,82%
Rapport	-	81%	62%

Itérations	600		
Partitions	1	3	6
Calcul (s)	100,09	79,90	65,11
Messages (s)	-	9,43	7,50
% messages	-	11,80%	11,52%
Rapport	-	80%	65%

Si on observe les 3 exemples, des itérations 100, 300 et 600, on trouve que le temps final après la division en 3 partitions est environ 80% du temps original et le 62% pour 6 partitions.

L'objective de réduire le temps est réussi, mais l'espace en mémoire est aussi importante. On va voir une petite table avec seulement 3 partitions pour montrer l'espace extra nécessaire pour gérer les messages. Pour chaque partition, on doit allouer une zone mémoire pour les messages qu'on va recevoir et les messages qu'on va envoyer. Les messages sont composés pour information des cellules, et la quantité des cellules dépend des zones *ghost* que les partitions ont en commun. Pour chaque cellule, on doit envoyer 13 données de taille *nspan* et une donnée de taille 1, alors $13 * 32 + 1 = 413$ (chaque donnée est un *float* de 4 bytes) :

Partition	1	2	3	Tot
Cell. recevoir	151	223	166	540
Cell. envoyer	154	212	174	540
Espace (MB)	0,49	0,69	0,54	**1,72**
Mémoire de travail	0,08	0,08	0,08	**0,24**

On ajoute 1,96 MB en mémoire alloué pour la gestion des messages aux 120 MB de mémoire totale. Cette mémoire est allouée la première fois qu'on l'utilise, est libérée au final, de manière qu'on ne perde du temps avec cette allocation qu'une fois pour exécution.

5.5. Conclusion

La création des partitions MPI apporte un degré de complexité à l'exécution, puisque on doit « prétraiter » le maillage pour créer les différentes partitions et on doit modifier le code pour permettre le parallélisme. Le plus important est créer un nouveau module pour gérer toute la communication, en cherchant à minimiser l'incidence de la synchronisation et le temps dans la construction et déconstruction des messages.

Le gain de temps est indiscutable, surtout si on observe que le temps ajouté pour la communication est complètement masqué. Alors, l'utilisation de MPI comme méthode de parallélisation est fortement recommandée.

Finalement, les nouvelles partitions comportent la création de nouvelles structures (*ghost*, faces, nœuds) qui doivent être alloués en mémoire. Par exemple, on va montrer pour le maillage exemple de 4203 nœuds la quantité de mémoire nécessaire par rapport au maillage complet et si on partitionne en 3 ou 6:

Partitions	**1**	**3**	**6**
Données	111	119,19	152,90
Messages	-	1,96	4,75
Total	111	121,15	157,65
Total / partition	**111**	**40,38**	**26,28**

*Information en MB

Donc, on se trouve avec 111MB dans un cas général, 6,23MB à cause des 3 partitions et 1,96MB générés pour la gestion de la communication, et des quantités un peu plus grands avec 6 partitions. Alors, la quantité ajoutée est minime. En plus, chaque partition a autour d'un tiers, 40,38MB, ou de un sixième, 26,28MB. Cette quantité est un petit exemple du but de ce projet, qui est gérer plutôt un maillage d'un million des points avec *nspan* 512, et non 4203 points et *nspan* 32. On avait dit au début de cette section que passer qu'on pourrait travailler avec 120GB, et maintenant cette quantité est devenue un tiers, 40GB.

L'adaptation à mémoire distribuée va nous permettre utiliser réseaux des machines afin d'accéder à centaines des cœurs de calcul.

6. Hardware : GPU

Grâce à l'outil MPI on a découpé le domaine de nôtre problème en plusieurs parties assez indépendantes de manière qu'on puisse exécuter le problème en différentes systèmes. On a obtenu une puissance d'utilisation que les processeurs conventionnels ne peuvent pas utiliser. Donc, maintenant est le moment d'explorer un changement de hardware, adapter nôtre programme NextFlow qui utilise actuellement la CPU 2 processeurs Intel Westmere avec 6 cœurs chacun [12] à l'utilisation des puissantes machines de calcul GPU (Graphics Processing Unit ou Processeur Graphique) avec 2496 cœurs [21].

Image 27. Carte GPU Tesla K20 de taille 26,26 x 11,12 cm. Source: PNY Europe

6.1. GPU Nvidia Tesla K20

Un GPU est un circuit intégré originalement présent dans une carte graphique qui assure les fonctions de calcul de l'affichage. Grâce à son extraordinaire structure hautement parallèle quelques entreprises ont créé une famille des GPU pour le calcul haute performance, soit pour l'analyse de données en masse, soit pour les calculs scientifiques à grande échelle.

Alors, les GPU sont des processeurs auxiliaires, où les CPU ou *host* qui exécutent le code principal peuvent charger les calculs plus lourds, et attendre le résultat ou même continuer avec son exécution, jusqu'à que le calcul soit fini.

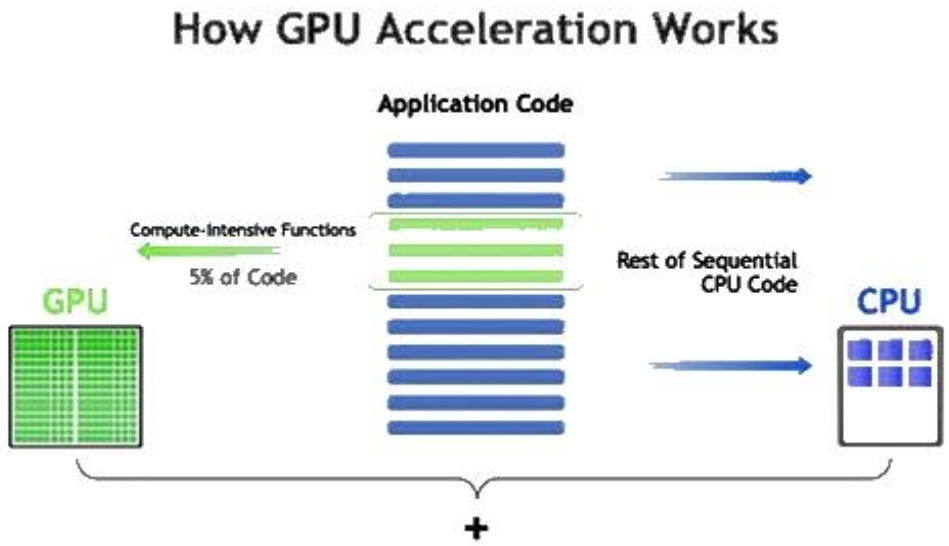


Image 28. Le code de l'application est pour la plus grande part exécuté en CPU, et seulement une partie de 5% du code source, qui utilise plus de 99% du CPU, est envoyé à GPU. Source: NVIDIA

Ici on va utiliser la famille GPU Nvidia Tesla, en particulier, Tesla K20 [22], contrôlé pour la CPU Intel Westmere déjà connue. Les spécifications techniques sont:

Caractéristique TK20	Valeur
Max. Double Précision (*double*)	1,17 TF*
Max. Single Précision (*float*)	3,52 TF*
Memory Size	5 GB
Memory Bandwith	208 GB/s
CUDA Cores	2496
Architecture	GK110

* TF = TeraFlops, où Flops est l'acronyme de "FLoating point Operations Per Second"

Sur l'architecture GK110 [23], on doit impérativement parler sur les 3 avances plus significatives:

- SMX : On améliore la performance en privilégiant les cœurs de traitement à la logique de commande. Chaque contrôleur SMX gère un groupe des instructions et dispose de 192 cœurs CUDA de single précision (*float*), avec 6 *warps*, ça veut dire, l'option d'exécuter au maximum 6 instructions au même temps et de dédier 32 cœurs à chacune. La GPU dispose de 13 SMX et chacune d'un cache interne de 64KB. Pour minimiser l'accès à la mémoire, on doit essayer de charger moins de 64KB avant de lancer chaque calcul. Si on a 13 SMX, avec 6 *warps* chacune, et 32 cœurs par *warp*, on a vraiment 2048 cœurs de calcul *float*. Le reste jusqu'à 2496 sont cœurs de input/output ou pour fonctions spécifiques.

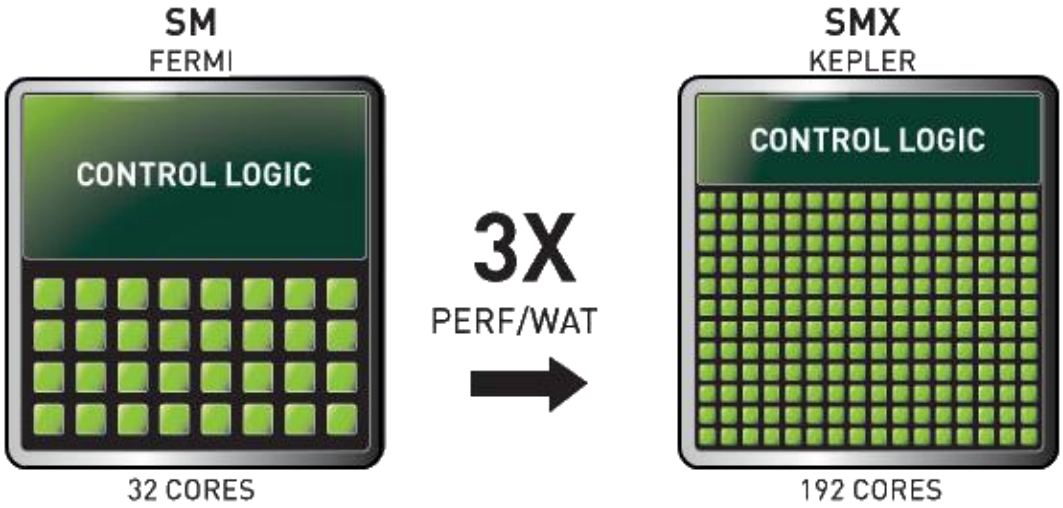


Image 29. Comparative entre la famille des GPU Fermi et la famille Kepler. Un seul contrôleur de logique commande 192 cœurs et non 32, de forme que la performance pour watt (flux énergétique) est triplé. Source: NVIDIA

- Dynamique de parallélisassions : La GPU peut exécuter dynamiquement de nouveaux threads sans faire appel au *host*. La GPU est d'une certaine manière intelligente pour optimiser ses ressources. [24]

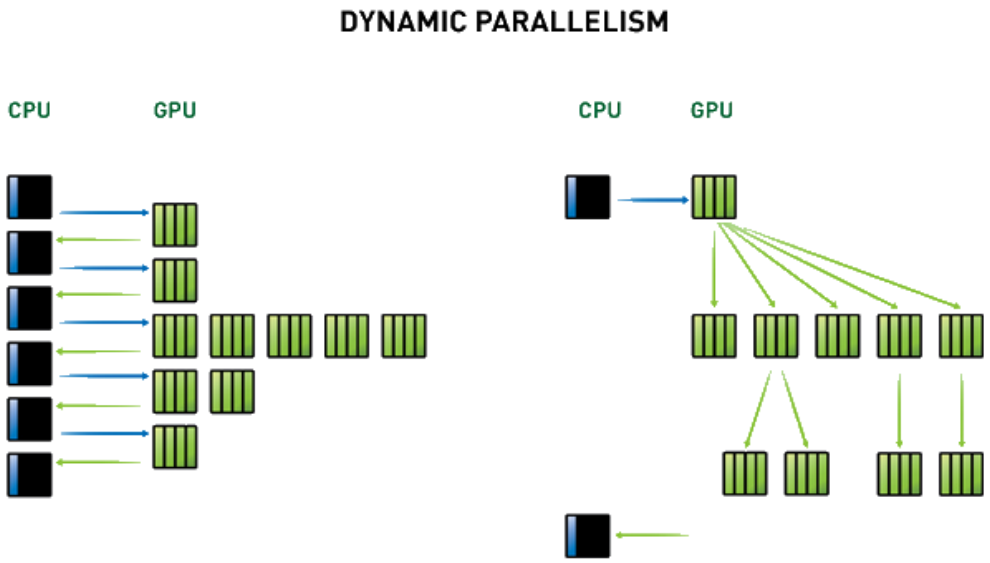


Image 30. Comparative entre les appelles CPU-GPU. Avant, à gauche, chaque GPU était créé pour un CPU. Avec parallélisme dynamique, à droite, les GPU peuvent s'appeler directement. Source: NVIDIA

- Hyper-Q : Plusieurs *host* peuvent utiliser en même temps un GPU, en réduisant l'inactivité des *host* et en utilisant tout la puissance de calcul GPU. Donc, on pourrait exécuter, par exemple, 3 partitions de nôtre problème en 1 ou 3 GPU. Donc, chaque partition utilise un *device* différent, même si plusieurs *devices* sont partie d'un seul GPU. Les différentes *devices* se répartissent les 5GB de mémoire global.

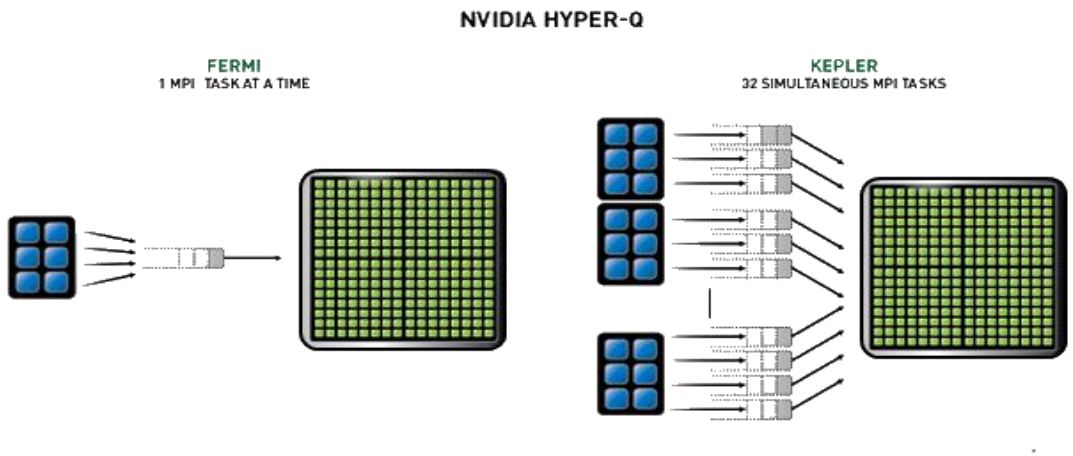


Image 31. Comparaison entre la famille des GPU Fermi et la famille Kepler. Un seul contrôleur de logique commande 192 cœurs et non 32, de forme que la performance pour watt (flux énergétique) est triplé. Source: NVIDIA

Comme curiosité, le superordinateur Titan, le supercalculateur plus puissant du monde, utilise une architecture hybride à base de 18.688 processeurs AMD (16 cœurs chacun, en total 299006 coeurs) et 18.688 accélérateurs GPU Nvidia Tesla K20. Titan peut arriver à 20 petaflops avec un mémoire de 710 terabytes. [25]

6.2. CUDA

On a dit que Tesla K20 dispose de 2496 nœuds CUDA. CUDA est l'acronyme de Compute Unified Device Architecture [26]. Développé en 2006, CUDA est une technologie de GPGPU (General-Purpose Computing on Graphic Processing Units [27]), c'est-à dire, la manière d'utiliser des GPU pour calculs généraux (pas seulement graphiques) habituellement exécutes par CPU.

CUDA existe dans plusieurs langages, mais on va utiliser l'adaptation de C, en tant que la plus étendue entre les développeurs [28]. Alors, tout le code de NextFlow a été traduite à code C, et à continuation adapté au « binders » ou « kernels », selon nos besoins:

- Binder: C'est un module situé entre le programme NextFlow en Fortran et l'exécution dans la GPU en CUDA, c'est qu'on a appelé *host*. Il est composé des fonctions intermédiaires en C qui permettent initialiser un *device* pour chaque partition MPI, gérer la mémoire dynamique, initialiser les données dans le *device*, appeler à chaque kernel, gérer la communication des données entre le *device* et la CPU (inclus les messages MPI, l'écriture des fichiers, le calcul des erreurs) et libérer la mémoire finalement. Un exemple très simplifié (exemple complet du module Fluxbal dans l'annexe 2):

```
void nxt_copy_binder_( )
{
   dim3 dimBlock (BLOCK_X, BLOCK_Y, 1);
   dim3 dimGrid (GRID_X, GRID_Y, 1);

    /* Kernel */
   nxt_copy_kernel <<<dimGrid, dimBlock>>> (new, old);

   return;
}
```

- Kernel – chaque fonction de calcul exécuté dans la GPU. Les kernels sont initialisés par leurs correspondants fonction binder dans le *host*, de sorte qu'on peut choisir combien de threads vont être utilisés au total. Bien sûr, le code doit être adapté pour cette exécution parallèle où un kernel est un « grid » avec plusieurs « blocks » et « threads » (bien qu'il soit possible en 3D, on va utiliser seulement en 2D) :

  - Grid : C'est une matrice en 2D (gridDim.x, gridDim.y) où chaque position est un Block.
    Une grid a son propre espace de travail indépendant aux grids initialisés pour autres kernels.
    On va lier un grid avec un kernel ou *device*.

  - Block : C'est une matrice en 2D (blockDim.x, blockDim.y) où chaque position est un Thread.
    On peut connaître la position de chaque block dedans un grid pour les valeurs (blockIdx.x, blockIdx.y). Un block est situé dans un SMX (on peut avoir plusieurs blocks dans 1 SMX, mais jamais 1 block dans plusieurs SMX), alors il a mémoire partagé interne (caché L1 et L2) mais il est complètement indépendante aux autres blocks. Un block peut avoir au maximum 1024 threads, gérés pour les 6 *warps* en groupes de 32 [23].
    On va lier l'idée du block avec une cellule.

  - Thread : C'est une tâche séquentielle. On peut connaître la position de chaque thread dedans un block pour les valeurs (threadIdx.x, threadIdx.y).
    Les threads des différentes blocks sont indépendantes et seulement peuvent se communiquer à travers de mémoire global du *device* (situation à éviter). Chaque thread dispose de 255 registres (alors, pour un calcul ils peuvent gérer au même temps 255 valeurs *floats* sans accéder au mémoire global du *device*). Un grid peut avoir 2048 threads en total [21].
    On va lier chaque thread avec une hauteur différente dans une cellule, puisque toutes les opérations sont égales pour chaque cellule et on facilite la gestion de tâches au SMX. En plus, la taille *nspan* doit être multiple de 32 pour mieux optimiser chaque *warp*.

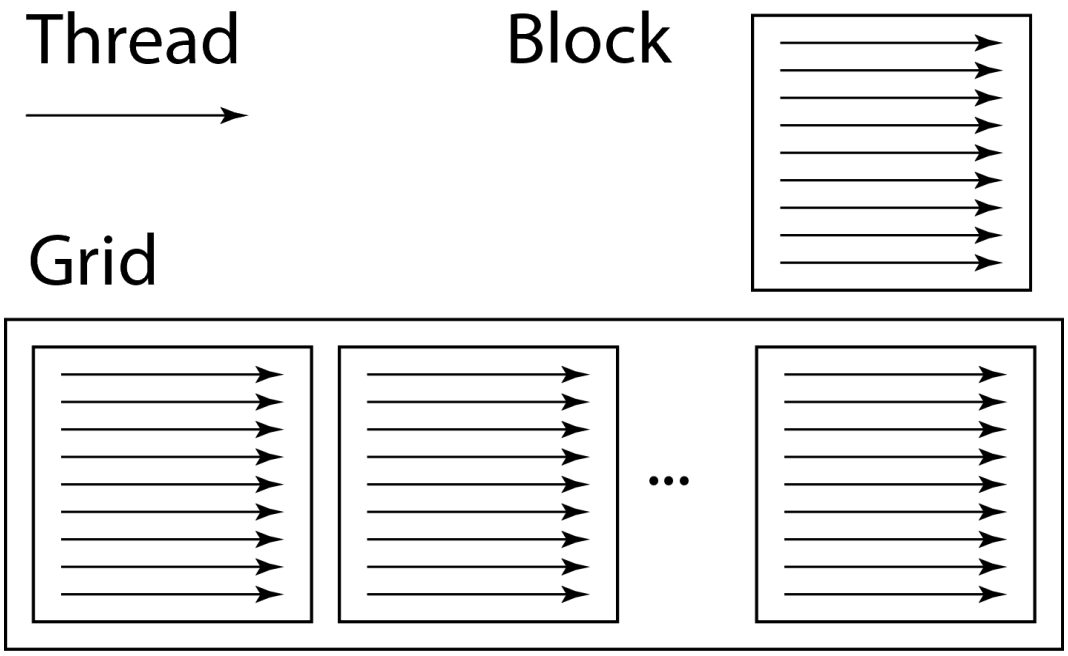


Image 32. Représentation visuel des relations entre threads, blocks et grids

En nous guidant pour la Thèse du Docteur Matthieu Lefebvre [29], on a décidé faire des tests pour établir quelle taille de blocks per grid et des threads per block améliore le temps d'exécution. On ne doit pas oublier qu'on dispose d'un GPU de 2048 cœurs, partagé pour 3 *device*s (3 partitions MPI), et on a choisi un *nspan* de 32 pour être associé à un *warp*. En plus, un block est associé à un SMX et le Tesla K20 n'a pas que 13 SMX, alors si on choisisse plus de 13 blocks, on aura plus d'un block par SMX.

Le maille utilisé est le même de 4203 nœuds initiales, 3 partitions et *nspan* = 32, mais on a établi un calcul plus long pour bien trouver le temps :

Taille Grid	Temps GPU (s)	Rapport
10	226,95	**1,00**
20	166,19	**1,37**
40	61,95	**3,66**
60	43,96	**5,16**
80	29,35	**7,73**
100	24,24	**9,36**
120	38,30	**5,96**
140	34,10	**6,66**
160	30,40	**7,47**
180	28,53	**7,95**
200	31,43	**7,22**
220	34,61	**6,56**

Alors, pour chaque kernel, en général, on peut calculer au même temps sur 100 cellules d'une taille *nspan* puisque avec 100 blocks on obtienne la plus grande vitesse. On montre un petit exemple pour l'utilisation (exemple plus complet dans l'annexe 2):

```
/*
* @param [in]   copy Matrice taille (max_cell, nspan) avec données à copier
* @param [out] paste Matrice taille (max_cell, nspan) avec données actualisées
*/
__global__ void nxt_copy_kernel(float ** copy, float ** paste, int max_cell)
{
    /* Identificateur threads */
    unsigned int thread  = blockDim.x * threadIdx.y + threadIdx.x;        /* Valeurs de 0 à 31*/
    unsigned int maxThread = blockDim.x * blockDim.y ;                    /* Total = nspan = 32 */
    unsigned int block = gridDim.x * blockIdx.y + blockIdx.x;             /* Valeurs de 0 à 99*/
    unsigned int maxBlocks = gridDim.x * gridDim.y;                       /* Total = 100 */

    int cellule = 0;

    for(cellule = block; cellule < max_cell; cellule += maxBlocks)
        old[cellule][thread] =  new[cellule][thread];

    return;
}
```

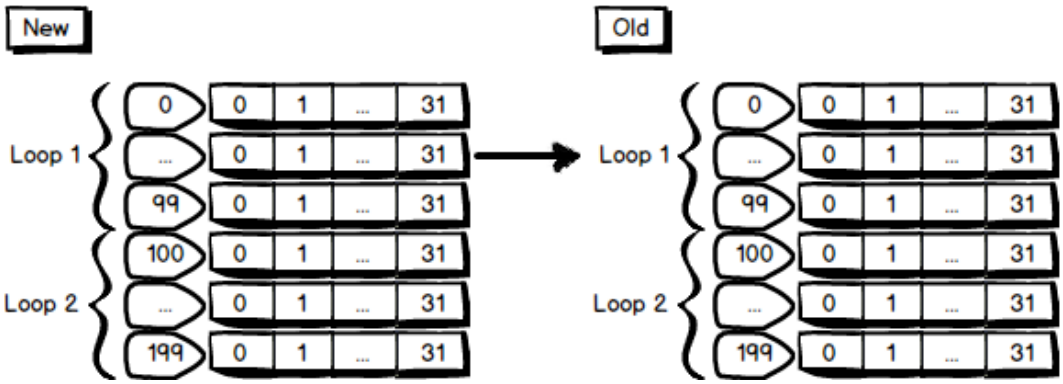


Image 33. Explication visuel de la fonction exemple. On fait la copie de New à Old des X premières cellules de taille *nspan* = 32 au même temps.

Le block qui opère la cellule 0 va faire aussi la cellule 100, 200, 300, etc, jusqu'à le fin de la matrice, mais le thread d'identificateur 0 va faire toujours la position *nspan* = 0 des différentes cellules.

Si max_cell = 2000, on devrait faire de manière séquentielle $2000 * 32 = 64000$ tâches. Avec le système du kernel on a en travaille $100 * 32 = 3200$ threads parallèles, alors on fait la copie en $^{64000}/_{3200} = 20$ tâches.

6.3. Messages MPI

Pendant la section 5.3 on a expliqué l'apparition des données *ghost* à cause de la partition du domaine et l'utilisation de MPI pour résoudre le souci dans la CPU, et la construction et déconstruction des messages.

Bien sûr, la GPU a besoin de communication MPI, parce que les différents *devices*, même s'ils sont dans la même GPU, n'ont pas mémoire global commune. Le but d'envoyer 14 données est déjà traité, mais maintenant il est plus intéressant de discuter des différents types de mémoire et comment on va les gérer. On va montrer un petit résumé [26]:

- Host Memory (HM): la mémoire typique du CPU, pas accessible pour la GPU. Ici sont les données initiales.
- Global Memory (GM): la version de host memory sur le GPU mais seulement accessible pour la GPU en utilisant les kernels. Ce sont les 5GB utilisés pour les différentes *devices*. Pendant l'initialisation, une fonction du *host* copie ici tous les données nécessaires pour l'exécution.
- Pinned Memory (PM): host memory spécialement préparé pour être accessible aussi pour un *device*. De cette manière, copier des données du pinned memory à global memory est plus efficace que depuis host memory. Par contre, ce type de mémoire n'est pas disponible ailleurs le *host*. Cette mémoire dynamique doit être allouée d'une manière spécifique, alors il n'est pas recommandé de l'utiliser en grande quantité [26], et elle devient une région commune qui accélère les copies de mémoire.

Alors, les devices ne peuvent pas utiliser les messages MPI pour se communiquer. Par contre, les *host* peuvent le faire, mais les données modifiées sont en GM, pas accessible pour eux. On pourrait utiliser PM pour toute l'information, accessible pour les deux, et simplifier le codage, mais la quantité nécessaire est trop grande et devienne trop lourde pour le *host*. Donc, chaque fois qu'il faut envoyer des messages on doit:

1. Le *host* alloue PM et appelle le device.
2. Le device va copier dans cette PM les données à envoyer qui sont en GM, et retourne le control au *host*.
3. Le *host* va construire le message en HM avec l'information de la PM.
4. MPI va envoyer et recevoir les messages.
5. Le *host* va déconstruire le message en HM et garder l'information en PM.
6. Le *device* va copier dans GM cette PM avec les données *ghost*.
7. Le *host* garde la même PM alloué pour toute la communication du programme, puisque la taille ne change pas et on l'utilise souvent, et seulement va la libérer à la fin.

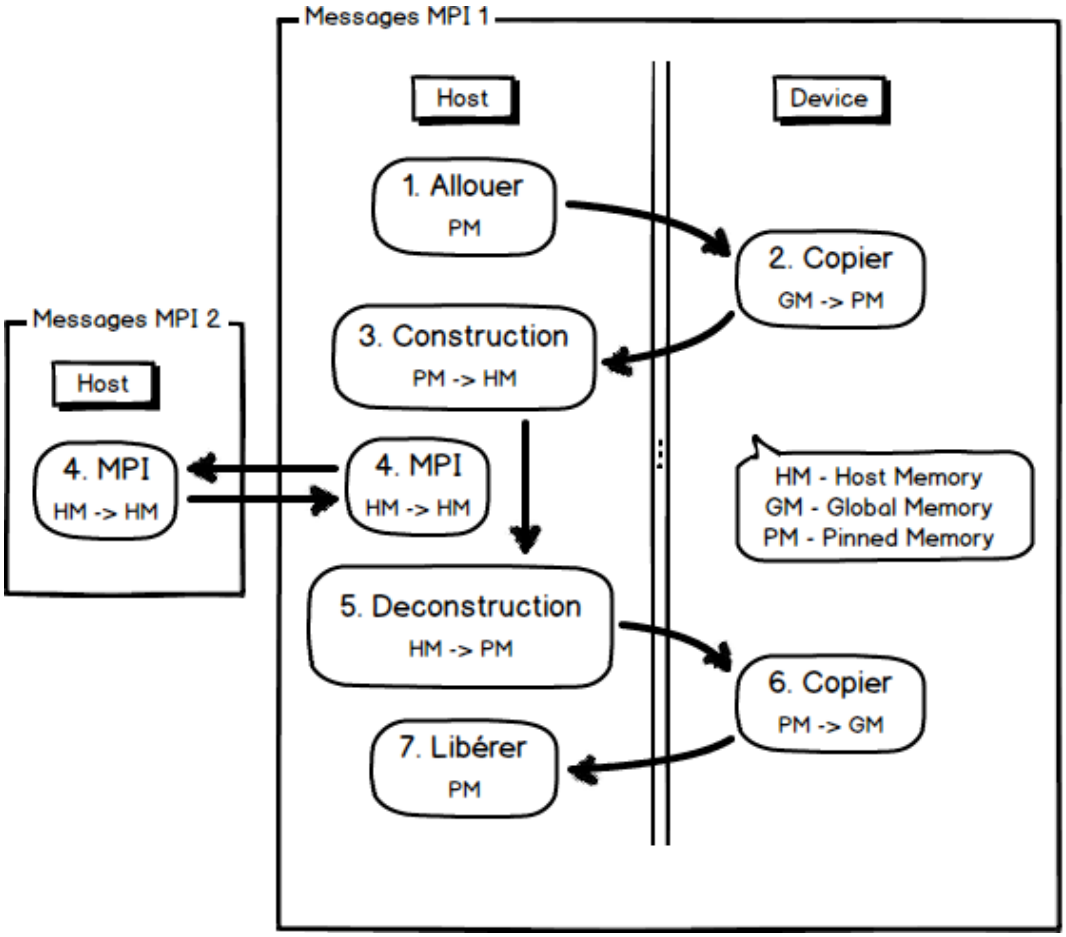


Image 34. Organigrame suivi pour *host* (CPU) et *device* (GPU) afin de réaliser la communication MPI entre le processus 1 et le processus 2

6.4. Améliorations

Il est vrai que la plus grande partie de l'écriture du code CUDA, si le code déjà existe en autre langage, Fortran dans notre cas, n'est pas trez difficile, puisque c'est une espèce de traduction en code C (on rappelle, CUDA est une librairie spécifique disponible en plusieurs langages). Chaque kernel est une partie de nos calculs, et on ajoute le module de control binder. Mais pour utiliser le vrai potentiel d'une GPU on doit modifier plusieurs détails en profitant de ses caractéristiques techniques. Par exemple, on a dit pendant le point MPI antérieur qu'on a profité de la « pinned memory » pour accélérer les copies de mémoire. Il est recommandé de consulter [30] pour voir les temps d'accès et transfert entre les différents types de mémoire d'un GPU K20 et entre GPU et CPU. On va préciser les quelques détails les plus importants pour l'accélération entre toutes les variantes testées:

- Utilisation de mémoire cache: Dans le host on a les données séparées en tableaux de structures (cellules, faces, nœuds, etc.). Chaque structure a les données associées à une cellule ou une face (flux, vitesse, densité, etc.). Si on travaille dans la CPU, c'est la manière la plus simple et intuitive d'accéder aux données. Mais dans la GPU c'est différent, puisque on doit considérer l'existence de la mémoire cache. Entre la mémoire globale d'un GPU et ses cœurs de calcul, existent deux niveaux de cache (niveau SMX et niveau block), qui sont accessibles pour les cœurs d'une manière beaucoup plus rapide. Si un cœur a besoin d'une donnée, on la cherche dans la mémoire globale et on la garde dans la cache avec les données qui sont immédiatement à côté, parce que c'est très possible que le cœur va les utiliser à la suite, donc ils sont déjà rapidement accessibles.
  On a dit que chaque block va traiter une cellule (ou face ou nœud…), alors est vraiment intéressant assurer que toute l'information d'une même cellule est en mémoire adjacente, de manière que un block qui demande l'information d'une cellule pourra accéder au reste des données rapidement. Alors, les structures ne sont plus utiles parce qu'on veut imposer que les adresses soient adjacentes en mémoire alors que ce n'est pas assuré avec les structures. On va utiliser le type de matrice qu'on a détaillé pour les messages MPI : matrices où chaque ligne correspond à une cellule, où sont alignées les données les unes après les autres. Par exemple, la matrice (on ne doit pas oublier qu'une matrice n'est qu'un tableau alloué en mémoire!) avec les variables conservatives d'une cellule (normalement accédées ensemble) a en chaque ligne la densité, les trois composants de la vitesse, l'énergie totale, etc. Et n'oubliez pas que chaque donnée est en général elle-même un tableau de taille *nspan*.

- Kernels asynchrones : Quand le *host* lance une fonction kernel qui va être exécuté dans le *device*, il peut continuer avec son exécution sans attendre la finalisation, et en plus il peut appeler plusieurs kernels, de manière que le *device* va tous les exécuter de manière autonome. On doit prêter beaucoup d'attention, la plupart de fois la synchronisation est obligatoire si les données qui doivent être modifiés dans le premièr kernel sont nécessaires dans les kernels suivants. En fait, seul le module de calcul *fluxes* permet la division asynchrone des tâches. On calcule le flux des fluides indépendamment entre les faces intérieures et les faces CL (open et adhérent). Alors, on peut lancer 3 kernels, un pour chaque type. La séparation des faces qu'on a détaillée dans la section 4.1. va être vraiment utile. Il faut donc séparer en trois kernels (3 fichiers) les différents calculs et faire que chacun soit exécuté sur une partie différente de la matrice *flux* (matrice provenant de la structure face faite pour assurer que les différentes données des fluxes d'une même face soient adjacentes). Ce changement est révélé comme vraiment utile, puisque comme on verra plus tard le module de calcul *fluxes* est le plus couteaux du programme.

6.5. Résultats

On va montrer finalement quelques résultats en temps et espace des différentes exécutions dans un GPU K20, en utilisant le maillage de 4203 nœuds entre 1 seule partition, 3 et 6.

Itérations	**6000**		
Partitions	**1**	**3**	**6**
Calcul (s)	114,37	149,82	158,23
Messages (s)	-	27,70	30,11
Pourcentage	-	18,49%	19,03%
Rapport	**-**	**125%**	**138%**

On observe une augmentation d'environ 25 % entre l'utilisation d'un GPU pour calculer une partition ou trois partitions MPI. On aurait préféré une diminution du temps, mais les résultats sont facilement explicables.

La GPU est une machine vraiment puissante pas comparable aux CPU. Dans un GPU faire les calculs entre 4203 nœuds ou un tiers de cette quantité est pratiquement identique, on gagne du temps mais pas de manière si évidente. On pourrait voir plus de différence avec des quantités beaucoup plus grandes des données. En tout cas, ici on utilise qu'une GPU pour les trois partitions, donc ils se divisent la mémoire et la puissance de calcul, donc les résultats sont perturbés.

Sans doute, le problème plus important est la communication. Dans un CPU la communication est directe, mais ici on doit faire deux copies de mémoire global à mémoire host et en plus attendre les messages gérés pour CPU. Même avec l'astuce de la « pinned memory », le temps de la communication ne peut pas être diminué, la GPU ne peut pas faire une autre chose qu'attendre.

On peut observer un exemple avec plus itérations, où on voit le même souci :

Itérations	**60000**		
Partitions	**1**	**3**	**6**
Calcul (s)	1153,36	1170,53	1179,36
Messages (s)	-	267,93	285,25
Pourcentage	-	22,89%	24,19%
Rapport	**-**	**101%**	**102%**

6.6. Conclusion

Si avec des partitions les temps de restitution globaux sont plus importants, pourquoi utiliser MPI ? On va comprendre mieux avec un exemple de l'espace total alloué :

Partitions	1	3	6
Données GPU	111	119,19	152,09
Données host	11,44	12,42	13,82
Messages host	-	1,96	4,75

* Information en MB

La quantité d'espace allouée dans le *host* pour l'exécution est petite. Mais la quantité dans le *device* est de 39,73MB pour 3 partitions ou 25,34MB pour 6 partitions (on rappelle que la GPU est la carte générale et que chaque *device* est initialisé à l'intérieur d'une GPU pour chaque partition). Si on passe á un million des nœuds et *nspan* 512, avec trois partitions on peut avoir pour chaque *device* 76GB. C'est complètement impossible puisque on n'a que 5GB en total pour chaque GPU…

Alors, si on veut utiliser les GPU, qui vont nous permettre de réduire les temps d'exécution et de profiter d'une puissance de calcul extraordinaire, on doit obligatoirement faire des partitions avec une mémoire total de l'ordre de 5GB afin d'occuper une GPU avec une seule partition.

7. Comparaison GPU vs CPU

On a vu les changements concernant l'espace mémoire et le temps de restitution global des calculs, trouvés pour chaque type de parallélisation, mais dans cette section, la dernière, on va analyser vraiment les différences obtenues dans plusieurs cas, en ciblant trois facteurs : mémoire, temps et erreurs, en particulier entre les CPU et GPU.

Dans la section 3.2 on peut trouver un organigramme simplifié du programme NextFlow original, avec les différents modules et les itérations. On explique par la suite les différences avec l'organigramme actualisé de notre version du programme NextFlow. Le schéma actuel est le suivant :

- Le module « Fort » est chargé de lire les fichiers et aussi de gérer la création des processus MPI. On ajoute aussi un module « Close ». Avant, il y avait aussi la finalisation du programme, fermeture des fichiers et gestion de mémoire allouée, mais maintenant il est vraiment important de bien gérer le final de chaque processus MPI et de fermer correctement les *devices* initialisés dans la GPU.
- On montre dans cet organigramme les deux types de mémoire de manière générale, CPU et GPU, puisque on sait pour la section 6.3 qu'on utilise plusieurs types de mémoire.
- On peut voir le module « Binder », qui masque complètement la gestion de GPU avec le langage CUDA au module principal en Fortran.
- Si le module « Scheme » initialise les données dans la CPU, on a besoin d'un module comme « Init Device » qui charge ces données dans le *device*.
- Afin de pouvoir comparer correctement les exécutions entre CPU et GPU, le module « Compute » a été doublé et maintenant est chargé d'appeler les autres modules dans la version CPU et GPU, en collectant les données pour les comparaisons de la mémoire et le temps d'exécution. Les deux versions peuvent être exécutées simultanément sur CPU et GPU pendant un nombre élevé d'itérations, avant de demander la comparaison précise de chaque champ calculé en CPU et GPU pour suivre l'accumulation des erreurs d'arrondi différentes dans les deux versions des bibliothèques mathématiques.
- Pour simplifier l'organigramme, on ne montre qu'une grande boucle quand il y a trois réellement (les trois types d'itérations, voir final de la section 3.3).
- On montre un module « Calcul » qui représente l'union des modules de calcul, pour simplifier si possible. Il y a la version des fonctions de calcul en CPU et la version des kernels dans la GPU.
- On montre également le module de communication MPI répandue de manière simplifiée, puisque réellement la communication s'exécute comme expliqué dans le module 6.3.
- Le module « Erreur » est chargé d'utiliser les résultats obtenus dans la CPU et GPU et montrer l'erreur relative. On explique dans la section 7.3 avec plus détaille.

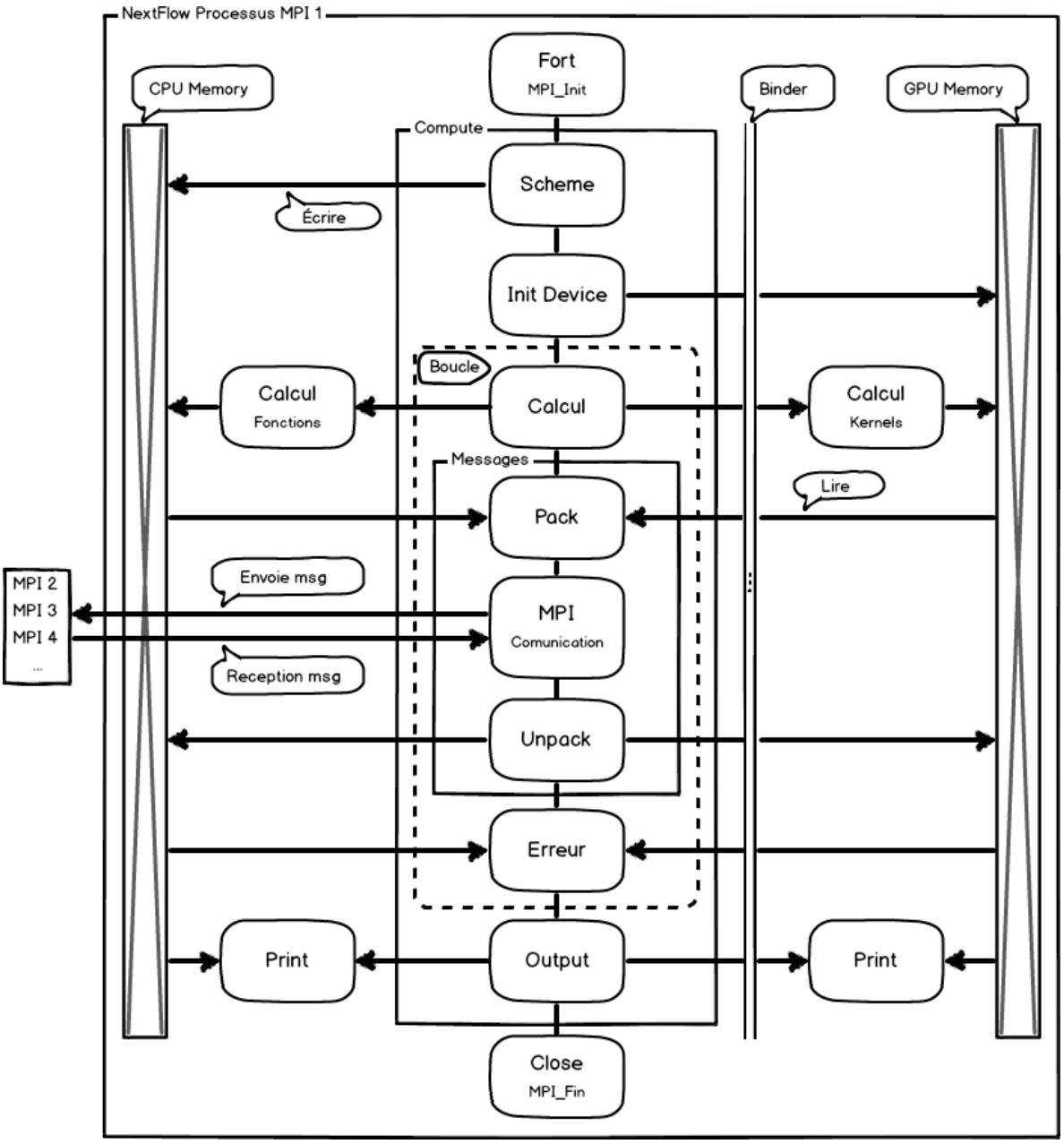


Image 35. Organigramme par modules du programme NextFlow après les changements réalisés par le projet. La section gauche est le domaine d'influence de CPU et la droite, de GPU.

7.1. Mémoire

On peut voit quelques exemples de mémoire alloué pour tailles différentes des maillages et divisions des domaines en 1, 3 ou 6 partitions. Toutes les données sont disponibles dans l'annexe 1.

Les données choisisses pour montrer sont : la mémoire alloué en cas d'utiliser CPU, la mémoire alloué en version GPU, qui est l'union de la mémoire du *device* (dans la GPU) et la mémoire du *host* (dans la CPU). Aussi, la taille des messages dans le cas d'avoir partitions, qui est la même taille soit CPU soit GPU. Les nombres ne sont pas la quantité totale de toutes les partitions, ils sont la taille moyenne de mémoire par partition.

En plus, il faut noter qu'on a choisit donnés de type *float* (4 bytes), si on veut utiliser type *double* (8 bytes) la mémoire est presque exactement doublé, puisque les entiers ne changent pas de taille, mais la plupart, avec différence, des donnés sont réels.

Noeuds	**4500**	**nspan**	**32**
Partitions	**1**	**3**	**6**
Données CPU	126,91	45,83	25,48
Données device	111	39,73	21,84
Données host	11,44	4,14	2,30
Messages	-	0,65	0,79

*Information en MB

Si on utilise partitions, la mémoire pour chaque partition est pareille que diviser la quantité totale pour la quantité des processus. Si on utilise GPU, la mémoire est un peut plus petite, et la quantité ajouté dans le *host* est minimal.

Noeuds	**25000**	**nspan**	**32**
Partitions	**1**	**3**	**6**
Données CPU	864,43	300,31	154,24
Données device	758,82	262,32	134,31
Données host	77,46	26,93	13,84
Messages	-	2,22	1,86

*Information en MB

Dans un cas avec 25000 nœuds, on note déjà que en utilisant partitions on ne divise pas exactement la mémoire, puisque on ajoute la section *ghost* qui est assez grande, mais la taille ajoutée n'est pas notable par rapport à la taille totale. Les messages sont minuscules et l'utilisation de GPU donne encore une petite diminution.

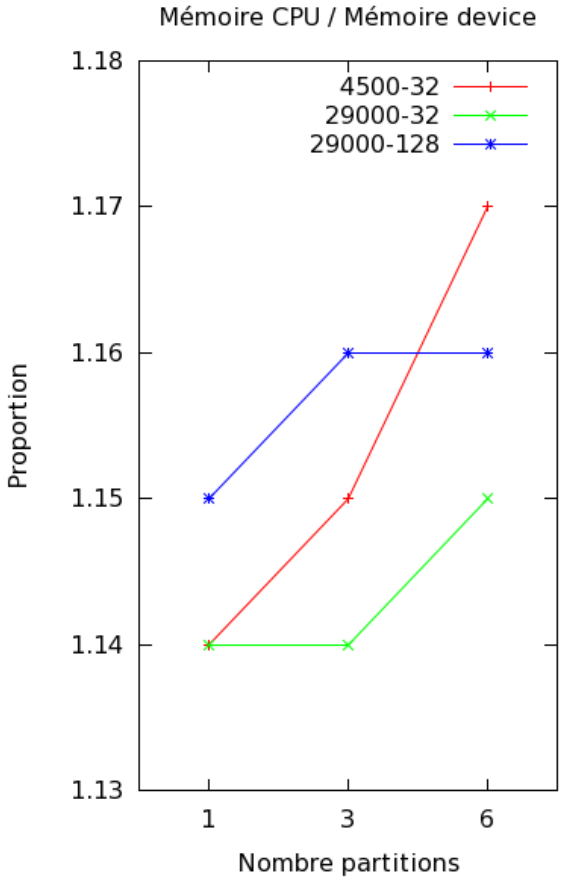
Noeuds	25000	nspan	128
Partitions	**1**	**3**	**6**
Données CPU	3120,47	1084,17	556,86
Données device	2701,93	935,37	479,34
Données host	309,28	107,53	55,25
Messages	-	8,87	7,41

*Information en MB

Finalement, on peut voir un exemple avec une augmentation de *nspan* de 4 fois. Toutes les données n'ont pas forme de vecteur, alors les nouvelles tailles de mémoire allouées ne sont pas exactement 4 fois plus grandes. Les messages sont composés pour information des cellules, qui ont toujours (moins la valeur *vol*) forme de vecteur *nspan*, donc la taille est presque multiplié par 4. L'utilisation en GPU est encore plus petite.

En terme général, on doit conclure que par rapport à mémoire alloué, l'utilisation de la parallélisation MPI est satisfaisante, et la présence double du cas GPU dans deux machines différentes (*device* et *host*) permet une diminution de la mémoire et, en conséquence, permet l'inclusion des maillages plus fins.

On montre une graphique finale avec le gain de mémoire dans le 3 cas :



7.2. Temps

On montre les différents résultats de temps d'exécution obtenus en un maillage 4500 nœuds, *nspan* de 32, et avec une quantité de 10000 itérations. On va se centrer dans les différences obtenues entre l'utilisation de CPU et GPU, 1 et 3 partitions et le choix d'utiliser valeurs *float* ou *double*. Pour OpenMP on a 12 threads, et avec la distribution choisie on a 13 threads pour 1 partition et 5 threads pour chacune des 3 partitions.

Les données choisisses pour montrer sont : le temps de calcul total (compute), le temps d'exécution du module Fluxes (le plus couteux en général), le temps d'exécution du module Messages et ses proportions par rapport au total. Dans le cas de 3 partitions, on montre le temps moyen. L'information complète est disponible dans l'annexe 1. Dans le calcul ne sont pas inclus les temps de « Scheme » ni l'initialisation et fermeture des *devices* et messages.

Données	float	Partitions	1
Type	**CPU**		**GPU**
Compute	1774,21		205,45
Fluxes	1297,60 (73,14%)		137,86 (67,10%)
Messages	-		-

*Information en seconds

Quand on utilise une seule partition on voit un temps beaucoup plus faible pour la GPU. Surtout par rapport au module Fluxes, qui a une diminution de rapport 9. L'explication est : plus un module est coûteux en termes d'opérations, plus on peut profiter des capacités d'une GPU. Lancer un calcul dans un *device* prend un temps minimal, alors, on doit profiter chaque fois qu'on entre pour faire la plus grande quantité d'opérations. En plus, Fluxes a été parallélisé (voir paragraphe 6.4 : calculs asynchrones), et cela a un effet dans l'amélioration.

Données	float	Partitions	3
Type	**CPU**		**GPU**
Compute	1331,25		189,41
Fluxes	1020,63 (76,67%)		90,52 (47,86%)
Messages	133,15 (10,00%)		55,16 (29,17%)

*Information en seconds

Si on passe à 3 partitions, on voit que le temps dans la CPU a diminué, mais pas beaucoup dans le cas de la GPU. C'est très normal, puisque on utilise qu'une machine GPU pour les 3 partitions, et on vient de dire que on profite mieux une GPU avec grande quantité des calculs, et si on fait 3 partitions avec une taille si petite (seulement 4500 nœuds) on n'utilise pas son potentiel. Il y a un seuil de temps dans une GPU qu'on ne peut pas diminuer à cause du cout inhérent à l'initialisation du GPU et à l'accès du module binder.

Données	double	Partitions	1
Type	**CPU**		**GPU**
Compute	2167,80		252,80
Fluxes	1710,04 (78,88%)		177,57 (70,24%)
Messages	-		-

*Information en seconds

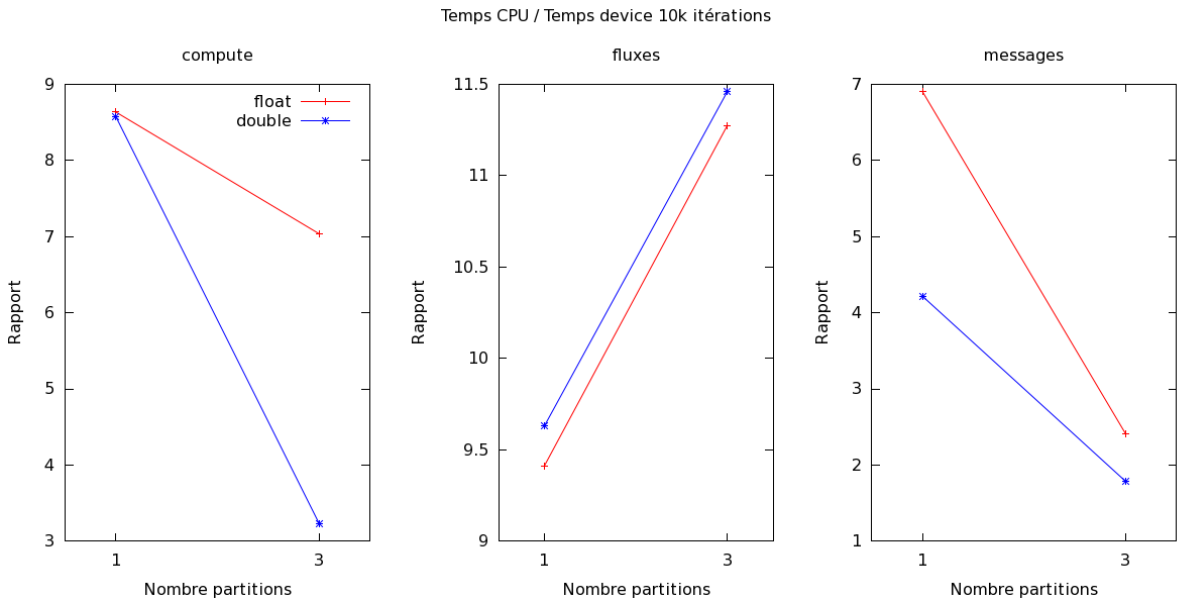
On passe dans un cas de d*ouble* et une seule partition. Par rapport aux cas *float* on voit une augmentation dans la CPU et une autre mineure dans la GPU. C'est dû à ce que la GPU Tesla K20 a quelques nœuds de calcul spécialisés en *double*, et son architecture permet d'unir deux nœuds de *float* et les traiter comme *double*. Encore, le plus, le mieux.

Données	double	Partitions	3
Type	**CPU**		**GPU**
Compute	2108,97		651,40
Fluxes	993,93 (47,13%)		94,95 (15,68%)
Messages	943,00 (44,71%)		527,31 (80,95%)

*Information en seconds

Ici on a une table avec les temps de 3 partitions et réels de 8 bytes. On observe déjà que les messages sont un problème pour tous les deux, on ne peut pas oublier que la GPU utilise aussi le même moyen de communication, et malgré son gain de temps dans la création des messages, elle doit attendre un 81% de son temps.

Finalement on montre trois graphiques avec le gain de temps de *float* et *double* en fonction du nombre de partitions pour les modules Compute (le temps total), Fluxes et Messages, pour faire un résumé :



7.3. Erreur

On va montrer la différence des résultats obtenus entre CPU et GPU. Chaque fois qu'on lance un calcul numérique on va obtenir des résultats différents, alors on ne va pas considérer une erreur de la GPU par rapport à la CPU puisque les résultats de la CPU vont changer chaque fois, et ce n'est pas correct penser que ces résultats sont les bonnes directement. Objectivement, on ne peut pas établir des résultats corrects et comparer de la même manière qu'on a faite avec la mémoire et le temps.

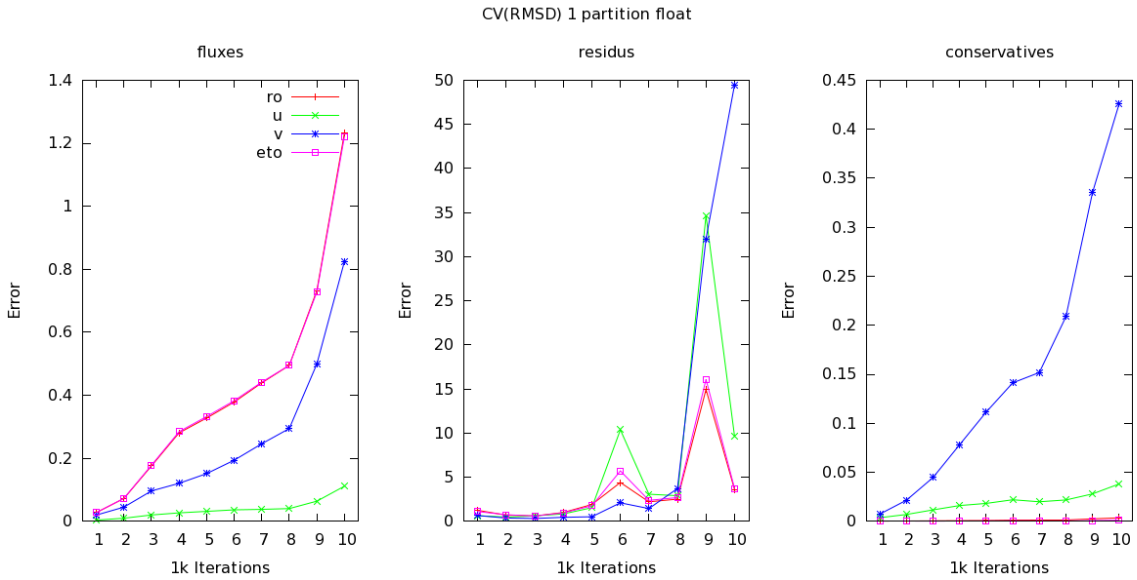
Alors, on va utiliser la formule de « Root-Mean-Square Error » (RMSE) qui représente la déviation standard de la différence entre les valeurs originales x_t et les valeurs obtenues y_t , où N est la quantité total des cellules ou faces (ça dépend de quelles variables on va calculer) :

$$RMSD = \sqrt{\sum_{t=0}^{N} \frac{(x_t - y_t)^2}{N}}$$

En plus, pour bien comparer les RMSE des différents variables, qui sont parfois d'ordres très différentes, on va calculer le Coefficient de Variation du RMSE (CV), qui simplement divise le RMSE pour la moyenne, de sorte qu'on obtienne une valeur normalisée :

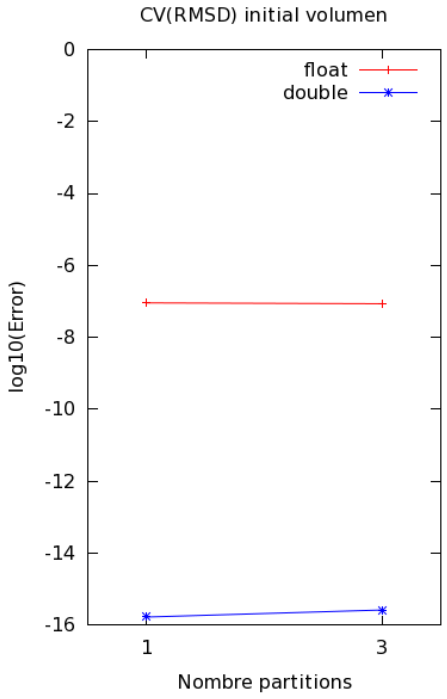
$$CV(RMSD) = \frac{RMSD}{\bar{X}}$$

On va montrer pour un cas de 4500 nœuds, *nspan* 32 et 10000 itérations l'erreur $CV(RMSD)$ entre les résultats de CPU et GPU de 1 partition et en utilisant *float*. Les variables choisies sont variables pour le calcul de fluxes et résidus, alors, variables intermédiaires, et les variables conservatives des cellules, qui sont une partie des vrais résultats qu'on cherche : densité (ro), vitesse (2 directions : u, v) et énergie totale (eto).



On peut observer qu'il y a deux variables de fluxes qui commencent à changer (*v*, *eto*) et ces variables vont changer plus tarde les résidus (surtout de la vitesse). Mais les variables conservatives, les résultats cherchés, ne changent pas 30 ou 50 fois ses valeurs originales, puisque le résidu a un impact très petit dans les variables pour chaque itération.

Et comment expliquer cette différence ? On doit chercher dans les architectures des machines. La manière de traiter et mémoriser les nombres réels dans la GPU provoque une petite perturbation qui est étendue avec chaque opération. On a choisi une valeur réelle de control, qui ne doit changer jamais, le volume des cellules (*vol*), et on a observé dans la première itération ça :

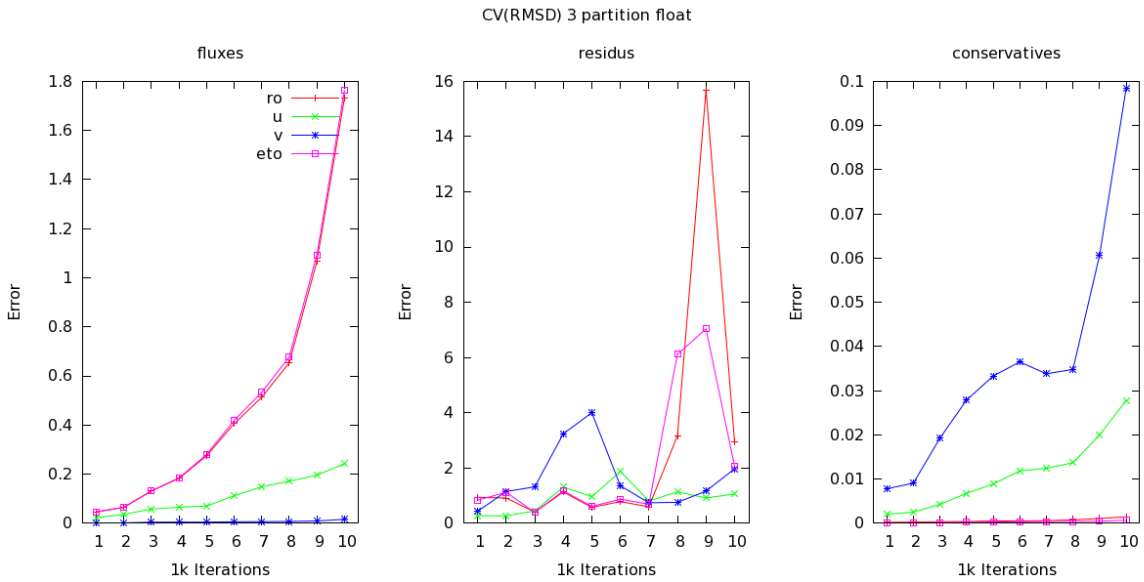


On peut observer qu'il y a différences. Si on utilise *float* l'erreur est d'ordre 10^{-7}, si ont utilise *double*, il est d'ordre 10^{-15}, dans le deux cas la dernière chiffre significative (*double* utilise 8 bytes, donc il a 8 décimales plus). Et de même manière avec toutes les autres variables, donc il est très normal avoir une erreur si considérable d'entrée. En plus, ce souci n'est pas seulement applicable à GPU, on peut l'observer dans n'importe quelle paire des architectures différentes.

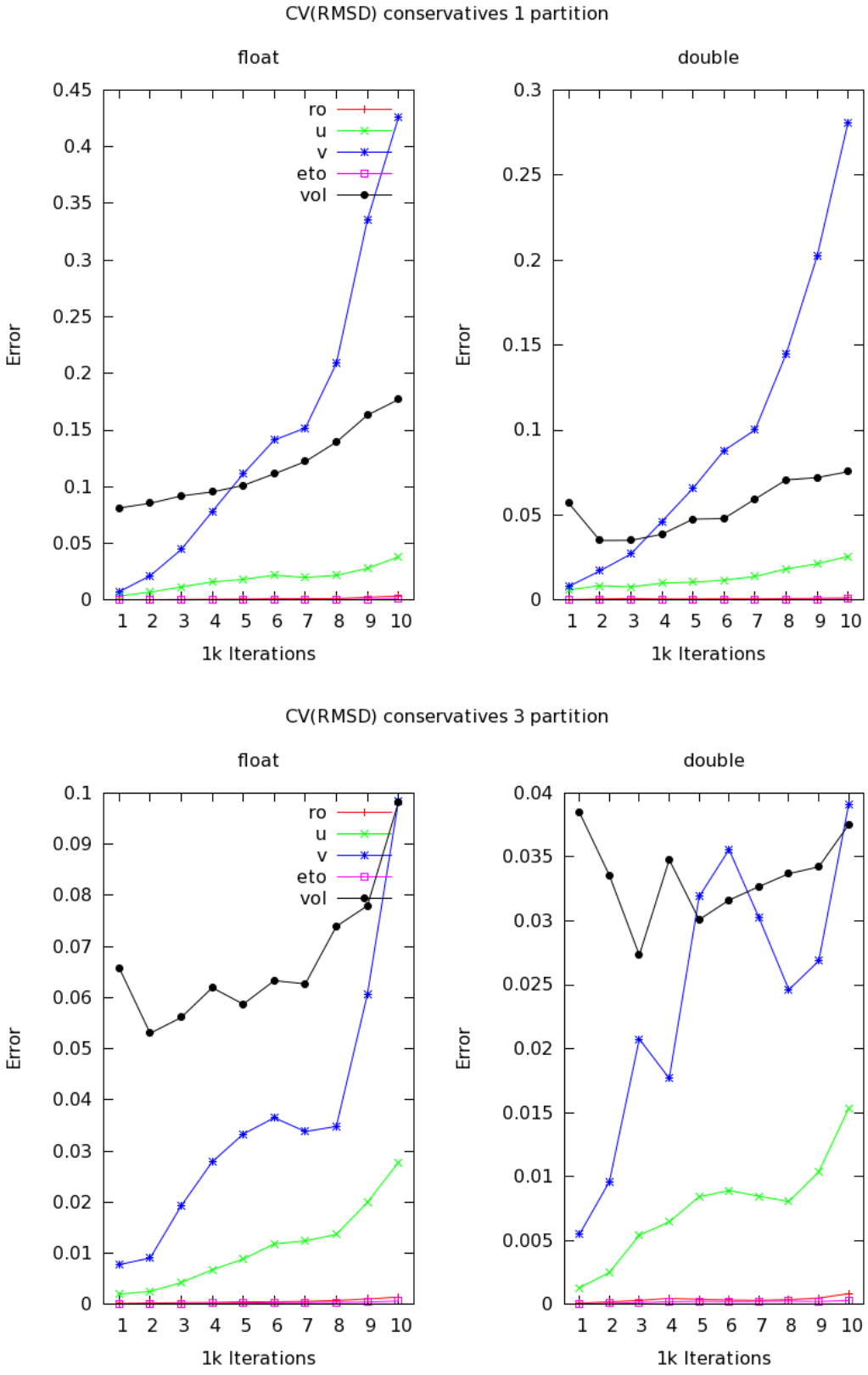
On doit être disposés à troquer précision pour vitesse, mais on peut essayer de minimiser la perdre. On ne peut pas changer l'architecture interne d'une GPU, mais on peut utiliser *flags* de CUDA dans le moment de la compilation pour éviter une perdre de précision plus élevée :

- ftz = false – Il désactive le *fast mode*, ça veut dire, une manière spécial de allouer et copier les réels qui est plus rapide mais qui provoque une perte des décimaux
- prec-div = true – Il active les divisions des réels sans perdre beaucoup de précision, mais plus lentement
- prec-sqrt = true – Il active les racines carrées des réels sans perdre beaucoup de précision, mais plus lentement

Dans la suite, on montre un cas avec 3 partitions, où on peut observer les mêmes conclusions mais produites sur *ro* et *eto*. Attention, les résultats montrés sont les résultats de la première partition. On n'a pas fait dans ce cas la moyenne entre les résultats de toutes les partitions :



On va continuer avec quelques graphiques qui montrent l'erreur du même exemple et 10000 itérations, pour voir les différences entre 1 et 3 partitions (dans ce cas, encore la première partition) et *float* et *double,* mais maintenant on va se centrer dans les variables conservatives, qui sont les plus intéressantes, et en ajoutant toujours la valeur de control de *vol*, pour nous faire une idée de l'erreur minimale qu'on va trouver toujours seulement pour utiliser une architecture différente.


CV(RMSD) conservatives 1 partition
float
double
ro
u
v
eto
vol
Error
1k Iterations
CV(RMSD) conservatives 3 partition
float
double
ro
u
v
eto
vol
Error
1k Iterations

7.4. Conclusion

En parlant du programme, les modifications qu'on a faites sur NextFlow ne changent pas son fonctionnement, elles ajoutent des fonctionnalités qui sont toujours optionnelles, de manière qu'on ne doit pas s'adapter obligatoirement ou utiliser d'autres versions antérieures.

Par rapport à la nouvelle dimension, on peut choisir la taille du domaine dans la direction d'envergure, dans l'axe Z, donc on peut toujours établir *nspan* = 1 et les résultats vont être les mêmes qu'avant.

Bien que cela ne soit pas montré dans l'organigramme au début de cette section, il est important de savoir que le programme permet exécuter seulement la version CPU, seulement la GPU ou tous les deux ensemble. Grâce à ces options, on peut soit tester les différences entre CPU et GPU, soit utiliser le programme sans l'obligation d'avoir une GPU disponible ou soit seulement la version GPU avec une ou plusieurs machines. De même manière, il y a aussi l'option d'utiliser OpenMP ou pas. Et sans oublier les futures utilisations de ce projet, tout le projet est documenté de manière assez détaillée avec l'outil *Doxygen* [31].

En parlant du projet, on a fait un changement fondamental en ajoutant la dimension 2,5D, et on a utilisé :

- Un mécanisme d'optimisation pour l'exécution séquentielle : vectorisation automatique sans recodage par Fortran en introduisant des structures, tableaux de longueur multiples de 32 et les opérations directes sur tous les éléments d'un tableau spécifiées en une seule ligne de code

- 3 différents types de parallélisation :

    - OpenMP afin d'améliorer de manière automatique le calcul sur plusieurs cœurs d'un même nœud CPU et l'optimisation des opérations du flag O3

    - MPI afin de partitionner l'espace et le temps d'exécution sur un réseau de nœuds de calculs

    - GPU afin d'accéder à un nombre gigantesque de cœurs de calculs 32 et 64 bits

On a détaillé les pas et considérations plus importants du projet en ajoutant des tests pour montrer les effets de chacun : OpenMP facteur 9 de vitesse, O3 facteur 6 de vitesse, MPI facteur 3 de réduction de mémoire et temps de la communication masqué, GPU facteur 9 de vitesse (1 GPU Tesla K20 par rapport à une exécution sur 2 sockets Westmere, 12 cœurs Intel hyperthreadés). Pour finir, on s'est concentré dans la comparaison entre l'exécution CPU avec OpenMP et GPU, avec différents quantités des partitions MPI et tailles du maille et *nspan*.

Avec ces données et résultats, on confirme l'utilité des procédures suivies, l'intérêt indiscutable d'approfondir dans les outils de parallélisation et la convenance d'adapter le calcul numérique à des réseaux de calcul comprenant des millions de cœurs afin de l'amener à un autre niveau.

La mise en place d'une parallélisation hiérarchique multi-niveaux et d'une optimisation séquentielle vectorielle (sur cœurs CPU Intel et Cœurs CUDA respectivement) ont été possibles en un temps réduit sur une maquette complète d'un code de calcul Navier-Stokes avec un codage FORTRAN, C et MPI, CUDA.

Les noyaux de calcul CUDA sont très proches des codes FORTRAN initiaux, avec des en-têtes qui gèrent l'interface avec les schedulers CUDA (dans les binders C), pour l'identification des indices de boucles à traiter par chaque CUDA-thread.

L'étape suivante du projet, lorsque les derniers bugs introduits par le portage de environ 2000 lignes de code auront été éliminés, est de réaliser des calculs complets de LES avec une longueur dans la 3ème dimension d'envergure de l'ordre de la longueur de l'aile. Pour cela il faudra de l'ordre du million de cellules 2D et *nspan* = 512 ou 1024.

Une configuration qui pourra être mise en opération sur un méga GPU-Ordinateur du type de la machine ROMEO de l'université de Champagne Ardenne [32] auprès de laquelle un projet scientifique a été déposé (260 nœuds de calcul GPU Tesla K20, environ 530000 nœuds de calcul CUDA).

Image 36. Méga GPU-Ordinateur ROMEO de l'université de Champagne Ardenne. SOURCE : Université de Reims

8. Références et contributions

[1] European Comission, Passenger Transport Statistics - Statistics Explained, eurostat, 2014.

[2] Advisory Council for Aviation Research and Innovation in Europe, About Acare, ACARE, 2010.

[3] Advisory Council for Aviation Research and Innovation in Europe, «ACARE,» [En ligne]. Available: www.acare4europe.com.

[4] Advisory Council for Aviation Research and Innovation in Europe, European Aeronautics : A vision for 2020, ACARE, 2001.

[5] IROQUA, «Initiative de Recherche pour l'Optimisation Acoustique Aéronautique,» [En ligne]. Available: www.iroqua.fr.

[6] Office National d'Études et Recherches Aérospatiales, «ONERA - Le centre français de recherche aérospatiale,» [En ligne]. Available: www.onera.fr.

[7] ONERA, Le secteur aéronautique se mobilise pour lutter contre le bruit de aéronefs, Châtillon: Communiqué de presse, 2010.

[8] DLR & AIAA & AFOSR, «2nd International Workshop on High-Order CFD Methods,» 2013. [En ligne]. Available: http://www.dlr.de/as/hiocfd.

[9] DLR & AIAA & AFOSR, «Turbulent Flow over a 2D Multi-Element Airfoil,» 2013. [En ligne]. Available: http://www.as.dlr.de/hiocfd/case_c3.1.html.

[10] C. Geuzaine et J.-F. Remacle, Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities, Université catholique de Louvain, 2002.

[11] J. Mallet, Manuel Utilisateur de NextFlow, ONERA, 2009.

[12] C. Carrascal, Développement d'une surcouche MPI, ONERA & École d'Ingenieurs Sup Galilée, 2014.

[13] OpenMP Architecture Review Board, «The OpenMP API Specification for Parallel Programming,» [En ligne]. Available: www.openmp.org.

[14] GCC Team, Options That Control Optimization, GCC, the GNU Compiler Collection.

[15] Karypis Lab, «Family of Graph and Hypergraph Partitioning Software,» [En ligne]. Available: http://glaros.dtc.umn.edu/gkhome/views/metis.

[16] MPI Forum, «Message Passing Interface Forum,» [En ligne]. Available: www.mpi-

forum.org.

[17] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard. Version 3.0, Knoxville: University of Tennessee, 2012.

[18] P. Kapinos, H. Rüdiger et A. Aures, MPI Caveats, RWTH Aachen University, 2012.

[19] I. Dupays, M. Flé, J. Gaidamour et D. Lecas, Message Passing Interface (MPI), IDRIS MPI Cours, 2014.

[20] D. Padua, Encyclopedia of Parallel Computing, Springer, 2011.

[21] NVIDIA Corporation, NVIDIA Tesla K-Series Datasheet, NVIDIA, 2013.

[22] NVIDIA Corporation, Tesla K20 GPU Active Accelerator, NVIDIA, 2013.

[23] NVIDIA Corporation, Kepler GK110 Architecture Whitepaper, NVIDIA.

[24] NVIDIA Corporation, Dynamic Parallelism In Cuda TechBrief, NVIDIA.

[25] Oak Ridge Leadership Computing Facility, Titan Overview, Oak Ridge National Laboratory.

[26] NVIDIA Corporation, Cuda Toolkit Guide, NVIDIA, 2014.

[27] M. Harris, «GPGPU,» 2002. [En ligne]. Available: www.gpgpu.org.

[28] NVIDIA Corporation, Cuda C Programming Guide v6.5, NVIDIA, 2014.

[29] M. Lefebvre, algorithmes sur GU pour la simulation numérique en mécanique des fluides, École d'Ingénieurs Sup Galilée, 2012.

[30] V. Haydin, NVIDIA Tesla K20 Benchmark: Facts, Figures and Some Conclusions, 2012.

[31] Doxygen, «Doxygen,» [En ligne]. Available: http://www.stack.nl/~dimitri/doxygen/index.html.

[32] Université de Reims, «Centre de Calcul de Champagne-Ardenne ROMEO,» [En ligne]. Available: https://romeo.univ-reims.fr/.

Contributions :

- Jean-Marie LE GOUEZ, Ingénieur chercheur, DSNA, ONERA.
- Emeric MARTIN, Ingénieur chercheur, DSNA, ONERA.

9. Annexes

9.1. Annexe 1 : Tests

Mémoire (MB)

Noeuds	4500	nspan	32
Partitions	1		
Données CPU	126,907		
Données device	110,997		
Données host	11,443		
Messages	0,000		

Noeuds	4500	nspan	32
Partitions	1	2	3
Données CPU	46,106	45,544	45,851
Données device	39,939	39,493	39,752
Données host	4,178	4,112	4,132
Messages	0,536	0,614	0,809

Noeuds	4500		nspan		32	
Partitions	1	2	3	4	5	6
Données CPU	24,942	24,959	27,481	25,678	25,107	24,734
Données device	21,420	21,443	23,369	22,010	21,553	21,264
Données host	2,254	2,265	2,475	2,311	2,273	2,238
Messages	0,664	0,582	1,258	0,962	0,637	0,641

Noeuds	25000	nspan	32
Partitions	1		
Données CPU	864,433		
Données device	758,824		
Données host	77,458		
Messages	0,000		

Noeuds	25000	nspan	32
Partitions	1	2	3
Données CPU	297,973	302,608	300,336
Données device	260,497	264,093	262,376
Données host	26,716	27,161	26,913
Messages	1,826	2,360	2,478

Noeuds	25000		nspan		32	
Partitions	1	2	3	4	5	6
Données CPU	154,641	153,335	153,398	154,709	153,982	155,346
Données device	134,648	133,596	133,695	134,666	134,105	135,151
Données host	13,914	13,745	13,731	13,883	13,817	13,933
Messages	1,469	1,768	2,063	1,923	1,764	2,142

Noeuds	25000	nspan	128
Partitions	1		
Données CPU	3120,472		
Données device	2701,932		
Données host	309,279		
Messages	0,000		

Noeuds	25000	nspan	128
Partitions	1	2	3
Données CPU	1075,723	1092,612	1084,174
Données device	928,601	942,074	935,444
Données host	106,673	108,451	107,461
Messages	7,291	9,426	9,896

Noeuds	25000		nspan		128	
Partitions	1	2	3	4	5	6
Données CPU	558,533	553,545	553,667	558,588	555,954	560,84
Données device	480,871	476,619	476,865	480,675	478,595	482,405
Données host	55,556	54,883	54,826	55,436	55,169	55,633
Messages	5,869	7,063	8,238	7,679	7,044	8,555

Temps (s)

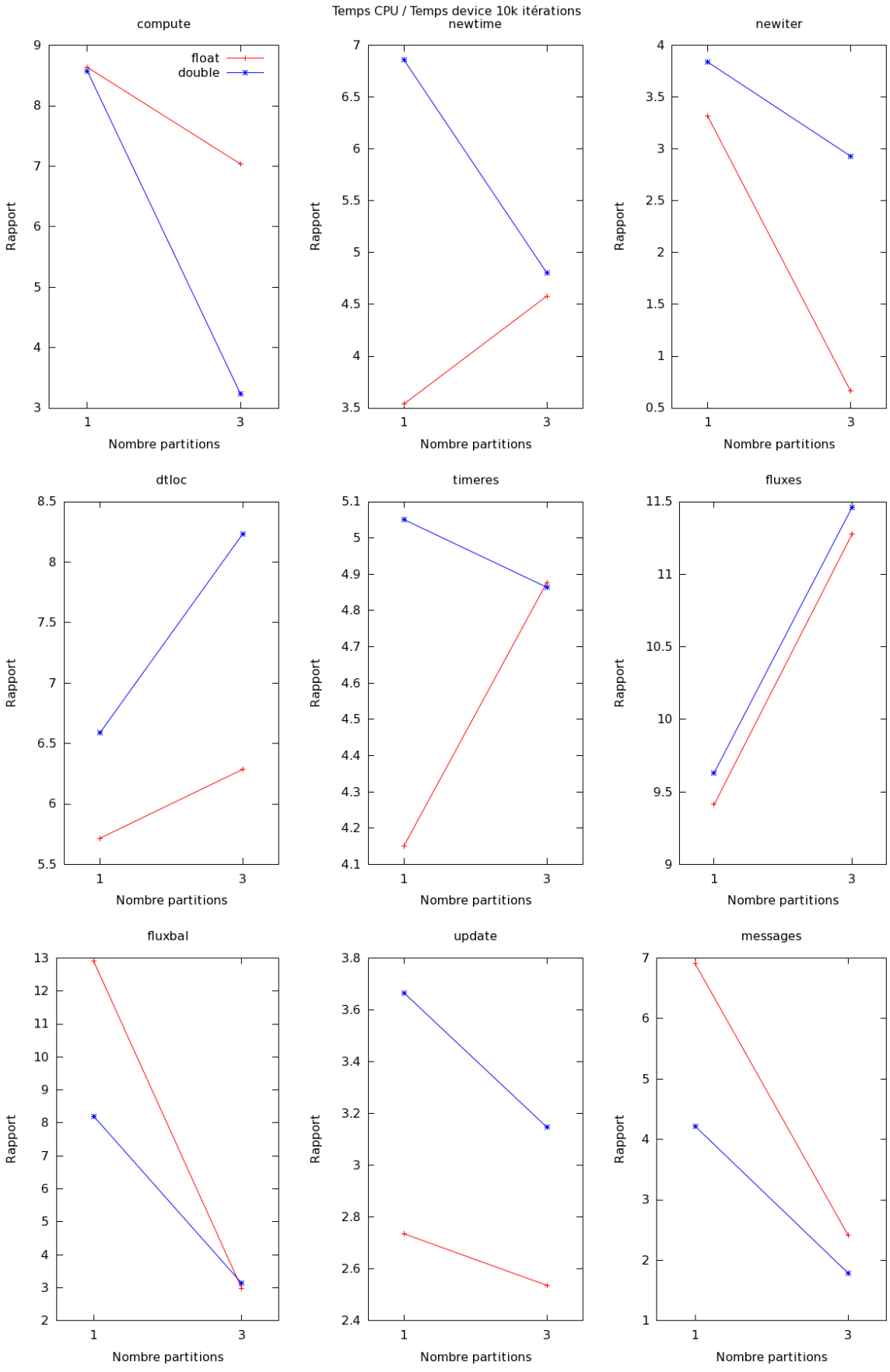
Données		float		Partitions		1
Type		CPU	%	GPU	%	Rapport
Compute		1774,20558	100	205,45199	100	8,63562
Newtime		0,52701	0,02970	0,14892	0,07249	3,53875
Newiter		3,31958	0,18710	1,00142	0,48742	3,31487
Dtloc		109,63100	6,17916	19,17744	9,33427	5,71667
Timeres		33,93916	1,91292	8,17695	3,97998	4,15059
Fluxes		1297,60087	73,13701	137,86405	67,10281	9,41218
Fluxbal		282,04617	15,89704	21,85500	10,63752	12,90534
Update		47,13028	2,65642	17,22654	8,38470	2,73591
Messages		0,01152	0,00065	0,00167	0,00081	6,90679

Données		float		Partitions		3
Type		CPU	%	GPU	%	Rapport
Compute	1	1340,70635	100	179,78613	100	7,45723
	2	1327,12011	100	193,11141	100	6,87230
	3	1325,91423	100	194,53123	100	6,81595
Newtime	1	0,40870	0,03048	0,07652	0,04256	5,34132
	2	0,36421	0,02744	0,09399	0,04867	3,87513
	3	0,39331	0,02966	0,08692	0,04468	4,52479
Newiter	1	0,40870	0,03048	0,60903	0,33875	0,67108
	2	0,36421	0,02744	0,56242	0,29124	0,64758
	3	0,39331	0,02966	0,58915	0,30286	0,66759
Dtloc	1	78,46160	5,85226	12,46876	6,93533	6,29265
	2	81,18967	6,11773	13,38076	6,92904	6,06764
	3	85,22868	6,42792	13,09892	6,73358	6,50654
Timeres	1	25,70036	1,91693	4,73336	2,63277	5,42963
	2	19,07830	1,43757	4,95300	2,56484	3,85187
	3	25,19415	1,90013	4,66203	2,39655	5,40412

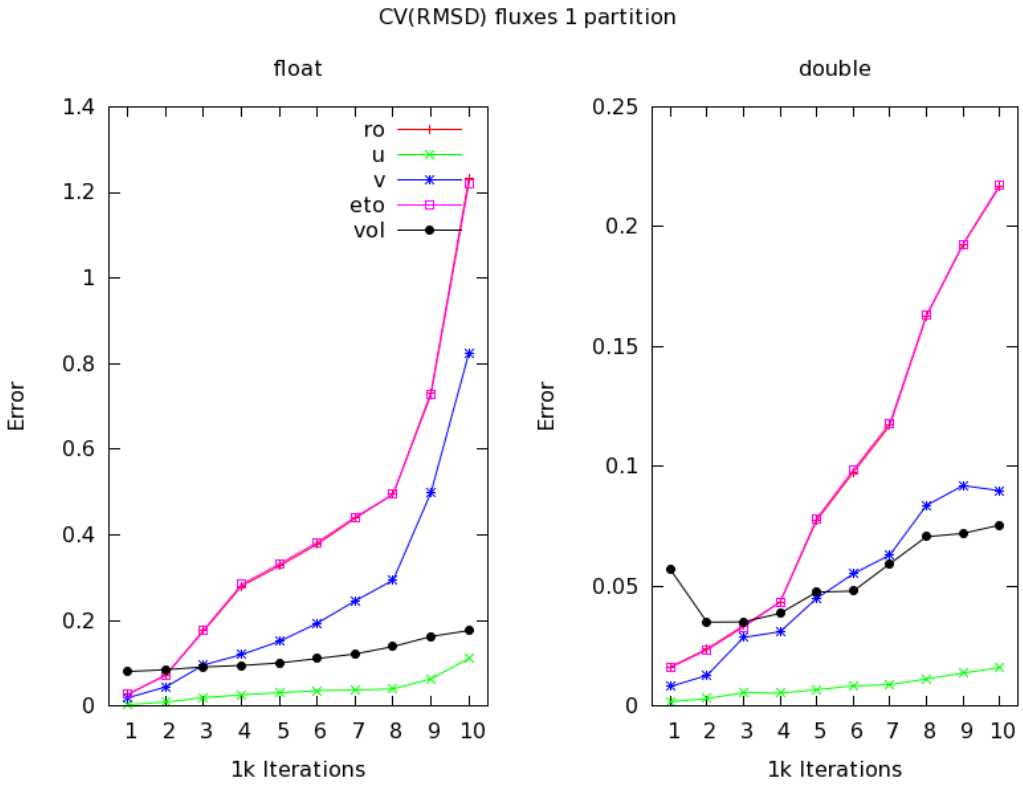
Fluxes	**1**	987,79661	73,67733	84,92912	47,23897	11,63084
	2	1028,98831	77,53543	91,58460	47,42578	11,23539
	3	1045,10115	78,82117	95,05795	48,86513	10,99436
Fluxbal	**1**	46,04916	3,43469	13,21995	7,35315	3,48331
	2	31,97059	2,40902	16,29331	8,43726	1,96219
	3	50,17355	3,78407	13,36853	6,87218	3,75311
Update	**1**	28,14393	2,09919	10,14353	5,64200	2,77457
	2	23,25536	1,75232	12,83758	6,64776	1,81151
	3	30,20525	2,27807	9,18581	4,72202	3,28825
Messages	**1**	171,70241	12,80686	53,60587	29,81647	3,20305
	2	140,85055	10,61325	53,40576	27,65541	2,63737
	3	86,90582	6,55441	58,48192	30,06300	1,48603

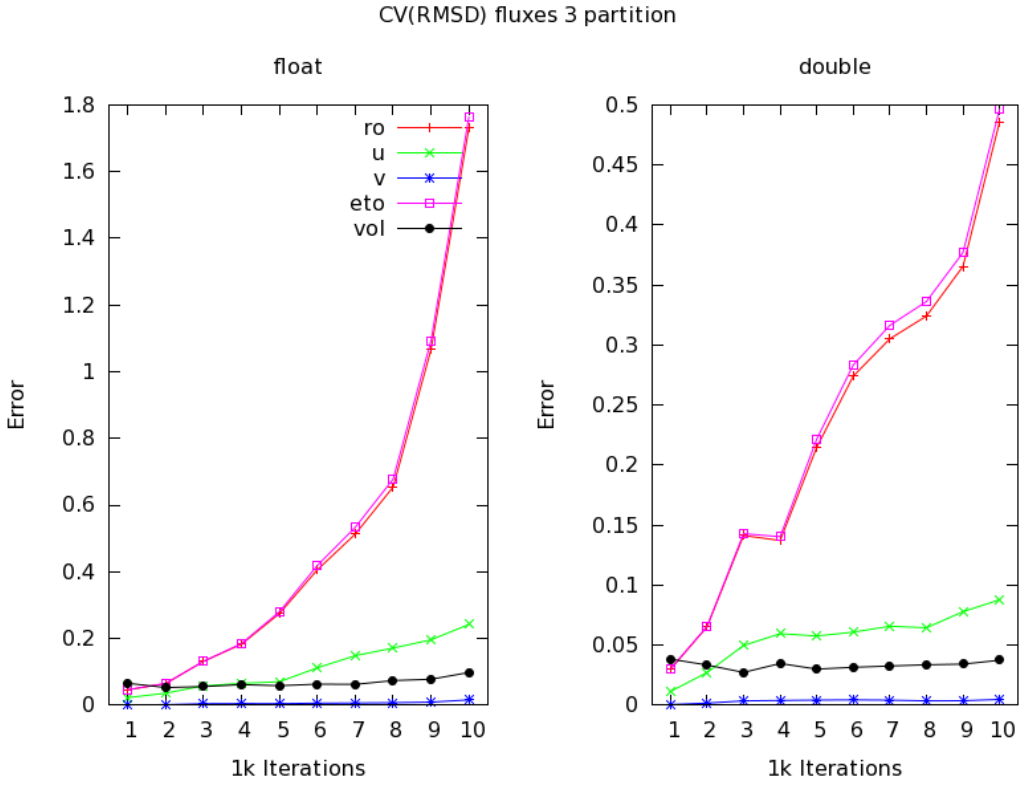
Données	**double**		**Partitions**		**1**
Type	**CPU**	**%**	**GPU**	**%**	**Rapport**
Compute	2167,79629	100	252,80274	100	8,57505
Newtime	1,06824	0,04928	0,15570	0,06159	6,86081
Newiter	4,26443	0,19672	1,11108	0,43950	3,83809
Dtloc	127,22891	5,86904	19,31125	7,63886	6,58833
Timeres	52,07273	2,40210	10,31007	4,07831	5,05067
Fluxes	1710,03629	78,88362	177,56724	70,23944	9,63036
Fluxbal	199,86995	9,21996	24,36375	9,63745	8,20358
Update	73,24418	3,37874	19,98091	7,90376	3,66571
Messages	0,01156	0,00053	0,00274	0,00108	4,21883

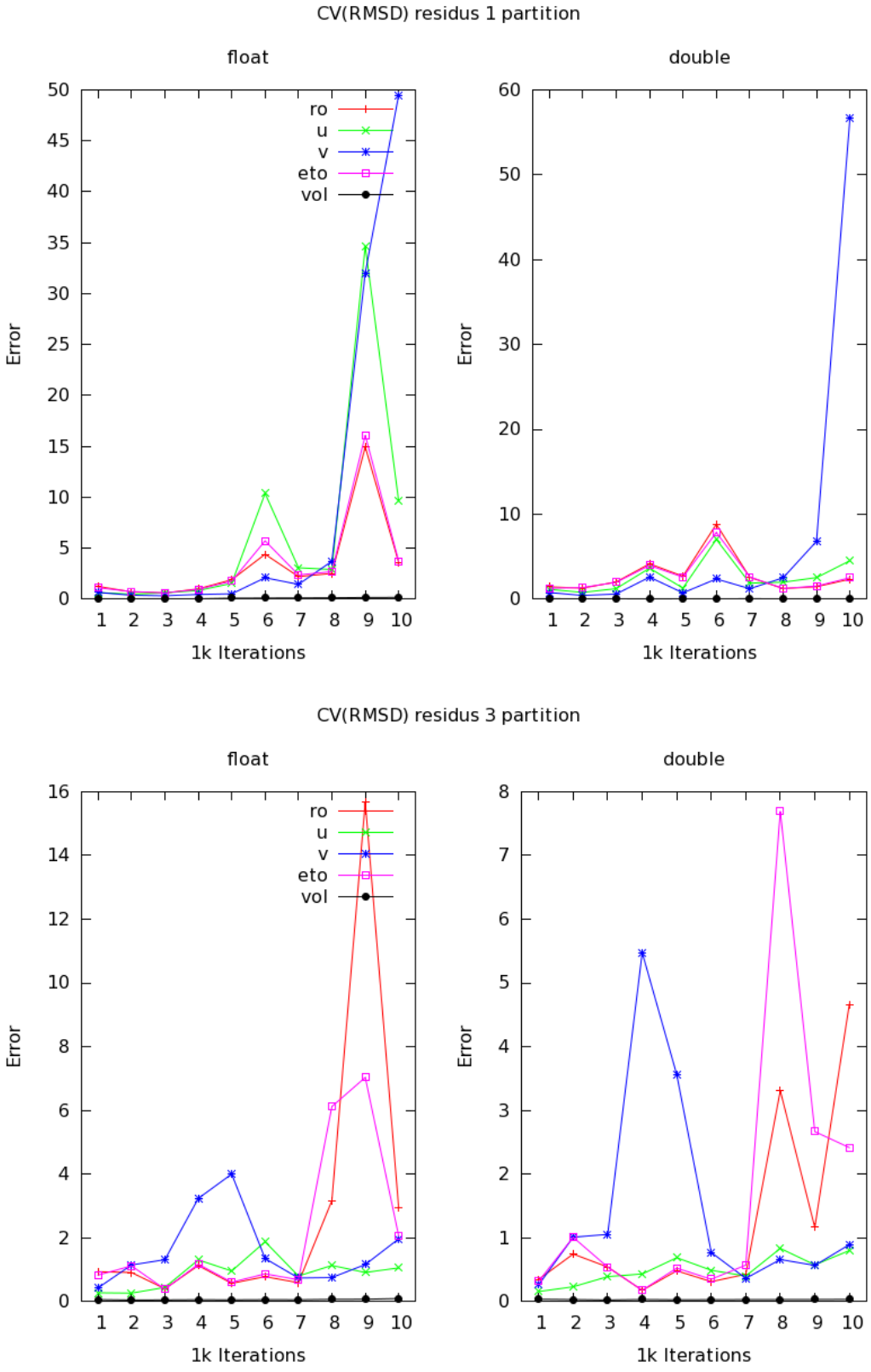
Données		double		Partitions		3
Type		CPU	%	GPU	%	Rapport
Compute	1	2518,16459	100	241,94942	100	10,40781
	2	1654,32742	100	1106,38488	100	1,49525
	3	2154,42114	100	605,87122	100	3,55591
Newtime	1	0,40385	0,01604	0,07639	0,03157	5,28648
	2	0,28453	0,01720	0,07570	0,00684	3,75860
	3	0,42884	0,01991	0,07991	0,01319	5,36664
Newiter	1	2,01534	0,08003	0,63479	0,26236	3,17483
	2	1,62705	0,09835	0,71956	0,06504	2,26118
	3	2,31118	0,10728	0,67785	0,11188	3,40956
Dtloc	1	67,77016	2,69125	10,72816	4,43405	6,31703
	2	77,17675	4,66514	7,17761	0,64874	10,75243
	3	75,34982	3,49745	8,84421	1,45975	8,51967
Timeres	1	30,71471	1,21973	5,53225	2,28653	5,55194
	2	20,92104	1,26463	5,54567	0,50124	3,77250
	3	26,24364	1,21813	4,93357	0,81429	5,31941
Fluxes	1	950,32254	37,73870	97,09432	40,13001	9,78762
	2	1040,08465	62,87054	68,15067	6,15976	15,26155
	3	991,37444	46,01581	94,95420	15,67234	10,44055
Fluxbal	1	38,73186	1,53810	11,73977	4,85216	3,29920
	2	32,58113	1,96945	10,53301	0,95202	3,09324
	3	40,20238	1,86604	13,21993	2,18197	3,04104
Update	1	33,45740	1,32864	13,94624	5,76412	2,39903
	2	33,47571	2,02352	8,87305	0,80199	3,77274
	3	32,42485	1,50504	8,74952	1,44412	3,70590
Messages	1	1394,74874	55,38751	102,19750	42,23920	13,64758
	2	448,17657	27,09116	1005,30961	90,86437	0,44581
	3	986,08600	45,77035	474,41203	78,30245	2,07854

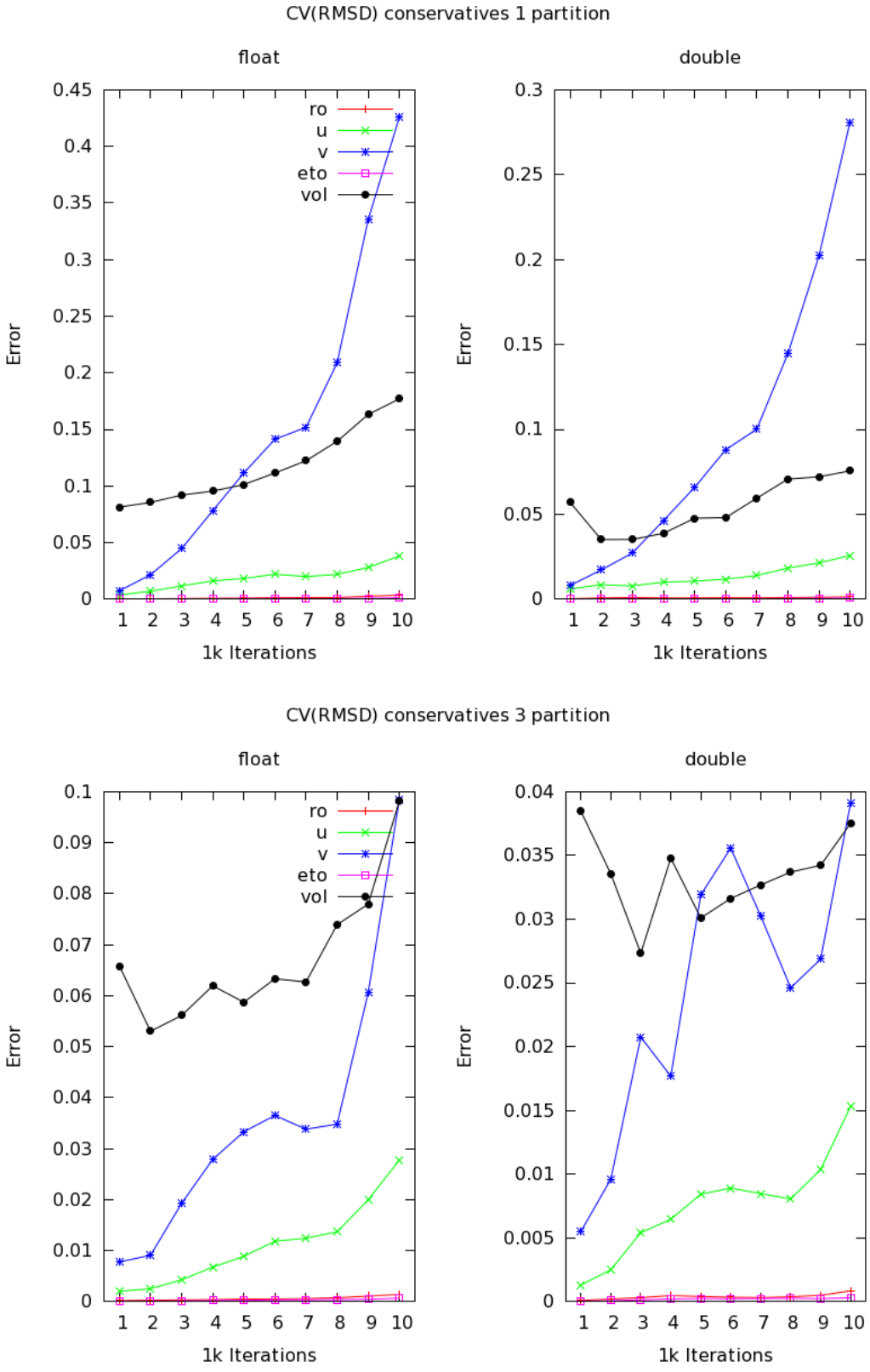

Temps CPU / Temps device 10k itérations
compute
float
double
Rapport
Nombre partitions
newtime
newiter
dtloc
timeres
fluxes
fluxbal
update
messages

Error (CV(RMDS))






CV(RMSD) residus 1 partition
float
double
ro
u
v
eto
vol
Error
1k Iterations
CV(RMSD) residus 3 partition
float
double
ro
u
v
eto
vol
Error
1k Iterations


CV(RMSD) conservatives 1 partition
float
double
ro
u
v
eto
vol
Error
1k Iterations
CV(RMSD) conservatives 3 partition
float
double
ro
u
v
eto
vol
Error
1k Iterations

9.2. Annexe 2 : Fluxbal

CPU

Dans la version CPU (*cpu_bol = TRUE)* , le module Compute (*nxt_compute*.f) appelle la fonction « nxt_fluxbal » et envoie comme arguments las structures nécessaires : « cell » avec les cellules, « face » avec las faces et « cnc » avec information des connectivités entre faces et cellules.

```
C
C       *** Bilan des flux
C
        if(cpu_bool) then
                ompclk_s(0) = omp_get_wtime()
                call nxt_fluxbal(cell,face,cnc)
                ompclk_e(0) = omp_get_wtime()
                ompclk(9,1) = ompclk(9,1) + ompclk_e(0) - ompclk_s(0)
        endif
```

On utilise les fonctions d'OpenMP pour calculer le temps, de manière que les résultats est le temps réel et non le temps de calcul (la somme des temps des threads d'OpenMP). Le résultat est ajouté dans une valeur *double* chaque itération afin de connaître le temps total dépensé dans la fonction *nxt_fluxbal*.

```
!> @file nxt_fluxbal.f
!> @brief Fichier pour calculer la balance des fluxes
!> @author Jean-Marie Le Gouez
!> @date 13-05-2014
!
C ********************************************************************
!> Fonction pour calculer la balance des fluxes
!! @param cell Struct type @ref commun::cell_type
!! @param face Struct type @ref commun::face_type
!! @param cnc Struct type @ref commun::cnc_type
        subroutine nxt_fluxbal(cell,face,cnc)
C
        USE commun
        USE omp_lib
        IMPLICIT NONE
```

Dans le fichier nxt_fluxbal.f on trouve la fonction pour calculer la balance des fluxes. Les commentaires commencés par "!" sont partie de la documentation Doxygen. La subroutine commence et ajoute les modules « commun.f » avec les variables globales du programme, le module OpenMP et l'option qui bloque la déclaration automatique des variables permit normalement pour FORTRAN 90.

En suite, on alloue la mémoire pour les arguments et pour les variables qu'on va utiliser pendant le calcul. En réalité, les arguments sont déjà alloués, mais la subroutine a besoin de connaître les tailles : « nvfmx » est la quantité maximale des cellules, « nfamx » la quantité maximale des faces. Pour les allocations on utilise « nspan », la taille de l'axe Z, et le type « real », qui permet être *float* ou *double* selon si le *flag –r4* ou *–r8* est actif à l'heure de la compilation.

```
C          *** Allocation structures
           type(cell_type) cell(nvfmx*4)
           type(face_type) face(nfamx)
           type(cnc_type)  cnc(nvfmx)
C
           real :: prodtk(nspan), xlfrtw(nspan)
           real :: rtet(nspan), tketet(nspan), epstet(nspan)
           real :: utau(nspan), cstdtl(nspan)
C
           real :: ssid,voltte
           integer :: ivffl,nbnv,nbfv,iff,ifac,isid,ifacadh
C
C$OMP   PARALLEL DO IF(omp_bool)
C$OMP1  DEFAULT(PRIVATE)
C$OMP2  SHARED(co1eps,co2eps,refmu,nvffl,nbfavf,
C$OMP3         cstage,cpri0,deltim,ielesnd,
C$OMP4         cell,face,cnc,rank)
C
           do ivffl = 1, nvffl
C
           [CALCUL]
C
           enddo
C$OMP   END PARALLEL DO

           return
           end
```

Après la déclaration des variables, on trouve l'initialisation de l'optimisation OpenMP (*omp_bool = TRUE*), avec les variables *private* ou *shared*. Le boucle de calcul est effectué sur « nvffl », la quantité des cellules internes (no CL, no *ghost*). On ferme l'optimisation OpenMP et on retourne le control des opérations au module Compute.

GPU

Pour la version GPU (*gpu_bool = TRUE*) le comportement du calcul du temps est pareil, mais l'appel à la fonction *binder* n'as pas des arguments. La fonction *nxt_fluxbal_binder ,* en CUDA, est chargé de gérer l'exécution dans la GPU, avec les données qui sont dans la mémoire du *host* et *device*.

```
C
C       *** Bilan des flux
C
        if(gpu_bool) then
                ompclk_s(0) = omp_get_wtime()
                call nxt_fluxbal_binder()
                ompclk_e(0) = omp_get_wtime()
                ompclk(9,2) = ompclk(9,2) + ompclk_e(0) - ompclk_s(0)
        endif
```

La fonction *binder* peut être trouvée dans le fichier *nxt_binder.cu*. Les commentaires initials sont pour la documentation Doxygen. On doit préparer l'initialisation du *device* pour exécuter le kernel. La taille des blocks est choisit pour la division entre 32, puisque on connait l'architecture d'une GPU et le control est optimal en groupes des threads de 32. Dans le cas que la taille de *nspan* soit plus petite, on choisi directement cette taille. Les dimensions de grid est 100x1 (tout déjà expliqué).

```
/**
 * @brief Fonction binder pour executer @ref nxt_fluxbal dans le device.
 * Cette fonction appelle @ref nxt_fluxbal_kernel
 */
void nxt_fluxbal_binder_()      {
        /* Taille Device */
        if(parameters_.nspanz < 32)     {
                BLOCK_X = parameters_.nspanz;
                BLOCK_Y = 1;
        }
         else     {
                BLOCK_X = 32;
                BLOCK_Y = (parameters_.nspanz+BLOCK_X-1)/BLOCK_X;
        }
        GRID_X = 100;
        GRID_Y = 1;

         dim3 dimBlock(BLOCK_X, BLOCK_Y, 1);
        dim3 dimGrid(GRID_X, GRID_Y, 1);
```

On appelle la fonction *nxt_fluxbal_kernel* avec les arguments nécessaires, qui sont en tout cas les directions de mémoire global du device des données. La structure des cellules est divisé dans les matrices « conser » des variables conservatives, « phisiq » des variables physiques et « residu » des résidus. La structure des faces est divisé en matrices «diswal », « infwal », « flux » (fluxes) et « xlfwal ». Par contre, la structure des connections a une seule matrice « cnc ». Les autres sont structures qui dans la version CPU existent dans le module *commun.f*, ajouté directement, sont passés aussi par argument.

```
        /* Kernel */
        nxt_fluxbal_kernel <<<dimGrid, dimBlock>>> (conser_gpu, residu_gpu,
                phisiq_gpu, diswal_gpu, infwal_gpu, flux_gpu, xlfwal_gpu,
                cnc_gpu, allocsize_gpu, connect_gpu, turbuke_gpu,
                unsteady_gpu, fluidprop_gpu, tailles_gpu, parameters_gpu);

         _gpuControl(cudaDeviceSynchronize(), "Synchronize fluxbal");

        return;
}
```

Après les calculs, on fait une synchronisation des threads de la GPU avant de sortir avec la fonction *cudaDeviceSynchronize()*. La fonction *_gpuControl()* est chargé de trouver les possibles erreurs des fonctions CUDA concernant le *device* qu'on fait dés le *binder* et imprimer si nécessaire un message. On retourne le control au module Compute.

Le kernel est trouvable dans le fichier *nxt_fluxbal_kernel.cu*. On peut trouver les commentaires de la documentation Doxygen. Pour indiquer que cette fonction va être exécute dans le device on doit ajouter *__global__*. On doit ajouter *nxt_fort.h* parce que là-bas sont les déclarations en CUDA des structures qui sont en *commun.f* (comme *block_parameters*).

```
/**
 * @file nxt_fluxbal_kernel.cu
 * @brief Fichier pour calculer le balance des fluxes en device
 * @author Carlos Carrascal Manzanares
 * @date 24-06-2014
 */

#include "nxt_fort.h"
```

On peut trouver aussi «reelPrecision », qui est déclare en *fort.h* comme *float* ou *double*. Il doit être obligatoirement la même option que la choisisse avec les *flags –r4* ou *–r8* dans la partie CPU.

```
/**
* @brief Function pour executer @ref nxt_fluxbal dans le device
* @param [in] conser Pointer a @ref conser_gpu
 […]
* @param [in] parameters Pointer a @ref parameters_gpu
*/
__global__ void nxt_fluxbal_kernel( reelPrecision * conser, […],
                                    block_parameters * parameters)
{
```

Quand on initialise le *device* est on a entré dans le *kernel*, il y a 100 blocks de taille multiple de 32, au minimum 32000 threads qu'on doit identifier afin de commencer à travailler. On doit identifier le *thread*, ou position de l'axe Z qu'ils vont travailler (obligatoire que la dimension des blocks soit exactement la même que *nspan* ! our permettre l'acces coalescent à la m'emoire, indispensable pour exploiter toute la bande passante memoire) ; et *block*, ou la cellule qui vont calculer. Quand on a une matrice de taille NxM mais en forme de vecteur, on peut accéder à la position XxY avec le calcul $position = Y * M + X$, et dan notre cas le calcul est exactement pareil.

```
/* Identificateur threads */
unsigned int thread  = blockDim.x * threadIdx.y + threadIdx.x;
unsigned int block = gridDim.x * blockIdx.y + blockIdx.x;
unsigned int maxBlocks = gridDim.x * gridDim.y;

 /* Structures */
int nspanz = parameters->nspanz;
int nvffl = allocsize->nvffl;
reelPrecision co1eps = turbuke->co1eps;
 reelPrecision co2eps = turbuke->co2eps;
[…]

/* Indices */
unsigned int ivffl = 0, ifac = 0;
unsigned int ivf_con = 0, ivf_res = 0, ivf_cnc = 0, ivf_inf = 0, ivf_dis = 0;
unsigned int ivf_phi = 0, ivf_flu = 0, ifac_flu = 0, ifac_xlf = 0, nbnv = 0;

 /* Valeurs */
reelPrecision prodtk = 0, xlfrtw = 0, rtet = 0, tketet = 0, epstet = 0;
reelPrecision epsovk = 0, coeimp = 0, rhoeps = 0;
[…]
```

En suite, on garde dans variables locales les valeurs des structures en mémoire global qu'on va utiliser. Chaque processus a disponibles à 225 registres internes, très rapides à accéder en comparaison avec l'accès à mémoire global. Donc on utilise ces registres pour garder les identificateurs, les valeurs des structures et autres variables qu'on va utiliser.

```
/* Tailles unions arrays */
unsigned int con_size = tailles->con_size; /* conservative */
unsigned int r_pos = tailles->ro_pos; /* ro */
unsigned int k_pos = tailles->tke_pos; /* tke */

unsigned int cnc_size = tailles->cnc_size; /* cnc */
unsigned int nbnovf_pos = tailles->nbnovf_pos; /* nbnovf */

unsigned int res_size = tailles->res_size; /* residus */
unsigned int rhsr_pos = tailles->rhsr_pos; /* rhsr */
unsigned int rhsu_pos = tailles->rhsu_pos; /* rhsu */
[…]
```

Avant de commencer les calculs, il faut aussi garder dans les registres les valeurs de la structure *tailles*. Cette structure est chargée de l'utilisation des matrices (vecteurs adjacents en mémoire) avec les donnés : *phisiq*, *conser*, *residu*, *cnc*, *faces*, *fluxes*, etc. Par exemple, la matrice *conser* (données conservatives des cellules) est formée par lignes qui sont cellules où chaque ligne a adjacents les données *ro*, *u*, *v, w*, *tke, eps*, etc. Donc la valeur *con_size* est la longueur totale de chaque ligne, et *r_pos* est, par exemple, la position où commence la donnée *ro* dans une ligne. De même manière avec *k_pos*, *e_pos*, etc.

```
for(ivffl = block; ivffl < nvffl; ivffl += maxBlocks)        {
        ivf_con = ivffl * con_size + thread;
        ivf_res = ivffl * res_size + thread;
        ivf_phi = ivffl * phi_size + thread;
        ivf_cnc = ivffl * cnc_size;
        […]
        conser[ivf_con+prodke_pos] = 0.;
        residu[ivf_res+rhsu_pos] = 1.5;
        nbnv = cnc[ivf_cnc+nbnovf_pos];
        phisiq[ivf_phi+vol_pos-thread] = 6;
        [CALCUL]
}
return;
}
```

La boucle de calcul va être sur les cellules (de 0 à *nvffl*). Chaque thread va commencer avec la cellule correspondant à son identificateur *block*. Pour chaque itération, les threads vont sauter à la prochaine cellule n'ont fait, ça veut dire, un saute de *maxBlocks* (quantité totale des cellules travaillées au même temps, dans nôtre cas, 100).

Chaque thread connait la cellule à travailler (*ivffl*), mais il faut calculer la position exacte dans les matrices ($position = Y * M + X$). Pour la matrice *conser*, *ivf_con* est la position géneral, où *thread* correspond avec un déplacement entre 0 et 31 (si *nspan* est 32). Plus tard, il faudra ajouter *prodke_pos* si on veut accéder dans la valeur *prodke*. Parfois une matrice est composé des valeurs scalaires, pas vecteurs de taille *nspan*, et dans ce cas on ne doit pas ajouter *thread* dans le calcul (de cette manière tous les threads accèdent dans la même position).

L'unique cas spécial est *vol* (volume) dans la matrice *phisique*, où le volume est scalaire et ne doit pas avoir *thread*. Si l'indice d'accès *ivf_phi* a ajouté *thread* afin de trouver autres données vecteurs, il faut retirer cette quantité pour chercher le volume.

Pour finir, on va voir un petit exemple de traduction CUDA (scalaires parce qu'on accède avec chaque thread) et FORTRAN 90 (vecteurs) :

cell(ivffl)%prodke	←→	conser[ivf_con + prodke_pos]
cell(ivffl)%rhsu	←→	residu[ivf_res + rhsu_pos]
cnc(ivffl)%nbnovf	←→	cnc[ivf_cnc + nbnovf_pos]
cell(ivffl)%vol	←→	phisiq[ivf_phi + vol_pos-thread]