Implémentation multi-GPU d'un code en volumes finis pour le calcul de haute performance des écoulements à surface libre

par

Vincent DELMAS

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE EN GÉNIE MÉCANIQUE
M. Sc. A.

MONTRÉAL, LE 25 AOÛT 2020

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. Azzeddine Soulaïmani, directeur de mémoire
Génie mécanique, École de Technologie Supérieure

M. Louis Lamarche, président du jury
Génie mécanique, École de Technologie Supérieure

M. Yousef Saad, examinateur externe
Computer Science, University of Minnesota, Minneapolis

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 13 AOÛT 2020

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Tout d'abord, je tiens à exprimer mes remerciements les plus vifs à mon directeur de recherche, professeur Azzeddine Soulaïmani, qui a encadré ce travail. Professeur Soulaïmani m'a permis de paralléliser un code de simulation numérique développé au cœur du groupe de recherche du GRANIT qu'il dirige à l'ETS et ainsi de faire un premier pas dans le domaine du calcul haute performance. Que ce soit concernant la programmation parallèle ou la définition physique et mathématique des problèmes traités, ses conseils ont toujours été très utiles et m'ont permis d'apprendre tant de choses.

J'aimerais remercier Pierre Dupuis, ingénieur hydraulicien au bureau de projet sur la gestion du risque d'inondation, pour les nombreux échanges que nous avons eus concernant la calibration d'un nouveau maillage de l'archipel de Montréal. Il a eu la patience de traiter et comparer à des relevés réels, les nombreux résultats de simulations que nous avons générés ce qui nous a permis d'améliorer notre méthode et je l'en remercie.

J'aimerais remercier Pierre Jaquier, étudiant à la maîtrise dans le groupe de recherche du GRANIT à l'ETS, avec qui les premières grandes bases de données de résultats pour le cas de la rivière des Mille Îles ont été générées. Son travail a permis de donner une application concrète au travail présenté ici et m'a donné la motivation nécessaire durant les longues phases de développement du code.

J'aimerais remercier professeur Jean-Marie Zokagoa qui m'a permis de me familiariser avec les premières versions du code traité dans ce travail en m'expliquant son fonctionnement interne.

J'aimerais remercier *Compute Canada* ainsi que *Calcul Quebec* pour l'accès qu'ils nous ont permis d'avoir sur leurs grappes de calculs sans lequel le travail proposé ici n'aurait pas pu voir le jour. Le service de support qu'ils ont mis en place a été d'une aide primordiale. Leurs réponses rapides et précises ont toujours été particulièrement utiles. Pour cela, j'aimerais remercier, parmi les nombreux membres du support de *Compute Canada* avec lesquels j'ai eu des interactions, Charles Coulombe, Maxime Boissonneault, Bart Oldeman et Huizhong Lu.

J’aimerais remercier professeur Louis Lamarche et professeur Yousef Saad pour la relecture et l’évaluation de ce travail.

J’aimerais remercier ma famille pour m’avoir soutenu dans la poursuite d’études supérieures et encouragé à venir étudier à l’ETS.

Mes remerciements vont aussi à l’ENSEIRB-MATMECA de Bordeaux en France, qui m’a offert la possibilité de faire un double diplôme à l’ETS à Montréal.

Implémentation multi-GPU d'un code en volumes finis pour le calcul de haute performance des écoulements à surface libre

Vincent DELMAS

RÉSUMÉ

Dans le cadre de ce mémoire, nous portons le code **CUTEFLOW** pour la résolution des équations de St-Venant par la méthode des volumes finis sur une architecture multi-GPU. Nous combinons dans ce but une version CUDA-*Aware* d'OpenMPI à du CUDA-Fortran. La bibliothèque METIS est utilisée pour faire la décomposition de domaine sur les maillages 2D triangulaires non structurés traités par le code. Une étude des accélérations et efficacités de la version multi-GPU est proposée, dans un premier temps, sur un cas de bris de barrage unidimensionnel, puis sur le domaine de la rivière des Mille Îles. Dans les deux cas des maillages allant jusqu'à 13 millions d'éléments sont utilisés. L'utilisation de 24 à 28 GPU pour la résolution de problèmes sur de tels maillages donne des efficacités supérieures à 80% ce qui est jugé optimal.

Mots-clés: multi-GPU, SWE, St-Venant, CFD, CUDA, MPI

Multi-GPU implemenation of a finite volume solver for the Shallow Water Equations

Vincent DELMAS

ABSTRACT

The main purpose of this master's thesis is the development of a multi-GPU version of the **CUTEFLOW** finite volume solver for the Shallow Water Equations (SWE). We use both MPI and CUDA-Fortran to use as many GPUs as we need. The METIS library is used in order to do a domain decomposition on the 2D unstructured triangular meshes. A CUDA-Aware OpenMPI version is used in order to speed up the messages between the MPI processes. Finally, a study of both speed-up and efficiency is carried over first, a classic onedimensional Dam-Break case, and then the Mille Îles river. In both cases, meshes with up to 13 million cells are used. Using 24 to 28 GPUs on these meshes lead to more than 80% efficacy.

Keywords: multi-GPU, SWE, St-Venant, CFD, CUDA, MPI

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

CFD	Computational Fluid Dynamics
GRANIT	Groupe de recherche sur les applications numériques en ingénierie et en technologie
CUDA	Compute Unified Device Architecture
SWE	Shallow Water Equations
MPI	Message Passing interface
GPU	Graphical Processing Unit
CPU	Central Processing Unit

LISTE DES SYMBOLES ET UNITÉS DE MESURE

Ω	Domaine de calcul
t	Temps
x	Position sur l'axe x
y	Position sur l'axe y
z	Position sur l'axe z
$\mathbf{V}$	Vecteur vitesse
u	Vitesse selon l'axe x
v	Vitesse selon l'axe y
w	Vitesse selon l'axe z
ρ	Masse volumique
p	Pression
σ	Tenseur des contraintes visqueuses
b	Bathymétrie
s	Hauteur de la surface libre
h	Hauteur d'eau
g	Accélération de la pesanteur
C_f	Coefficient de Chézy
n	Nombre de Manning
S_f	Terme source de friction
S_O	Terme source de bathymétrie

Indices :

i	Valeur dans la maille i
x	Dérivée par rapport à x

y	Dérivée par rapport à y
z	Dérivée par rapport à z

Indices supérieurs :

n	Valeur au temps n

Notations :

$\bar{u}$	Moyenne sur la hauteur d'eau de u, $\bar{u} = \frac{1}{h}\int_b^s u \partial z$

LISTE DES EXTRAITS DE CODE

Page

INTRODUCTION

Les inondations font partie des catastrophes naturelles les plus coûteuses. Qu'elles soient dues à des tsunamis, à des ruptures de barrages, ou simplement à de fortes précipitations, elles touchent de larges zones et nécessitent souvent l'évacuation de grandes quantités de personnes. La zone de l'archipel de Montréal est touchée de manière récurrente par des inondations dues aux fortes intempéries, c'est ce qui motive le travail mené ici.

Les mesures de prévention face à ces inondations sont multiples. Des protections physiques peuvent être mises en place, par exemple en canalisant le débit des cours d'eau ou en construisant des digues de protection. Des cartes d'inondations peuvent aussi être établies à titre de mesures préventives. L'objectif est de décrire la ligne d'inondation pour différentes situations critiques et permettre d'établir un plan d'évacuation d'urgence.

Dans ce but, l'utilisation de codes de simulations numériques s'est largement imposée parce qu'il est très difficile et coûteux de faire des expériences sur des modèles réduits. Le développement de codes permettant de simuler les écoulements à surface libre prenant place dans les cours d'eau a été un champ de recherche très actif durant les dernières décennies. On peut, entre autres, trouver le développement de tels codes dans : Audusse, Bouchut, Bristeau, Klein & Perthame (2004a) ; Bradford & Sanders (2002) ; Brufau, García-Navarro & Vázquez-Cendón (2004) ; Fortin, Manouzi & Soulaïmani (1993) ; Loukili & Soulaïmani (2007) ; Soulaïmani (1983) ; Toro (2001) ; Zokagoa & Soulaïmani (2010).

CUTEFLOW est un code de résolution des équations de St-Venant par méthode volumes finis qui a été développé dans le groupe de recherche du GRANIT de l'ETS. La première version séquentielle sur CPU a été développée dans un premier temps par Azzeddine Soulaïmani et Youssef Loukili (Loukili & Soulaïmani, 2007) puis par Jean-Marie Zokagoa (Zokagoa & Soulaïmani, 2010). Le code a par la suite été porté sur GPU en CUDA Fortran par Arun Kumar Suthar (Suthar & Soulaïmani, 2018). Finalement dans ce travail on a utilisé MPI pour porter le code

sur une architecture multi-GPU. L'objectif d'un tel code est de simuler des inondations sur des domaines de plus en plus grands dans le but d'établir des cartes d'inondations ou de prédire l'impact de digues ou autres constructions sur le cours d'eau.

On propose dans la suite de ce chapitre d'exposer les différents problèmes que posent les simulations d'inondations. On expliquera les raisons qui nous poussent à vouloir des simulations toujours plus rapides et la nécessité de l'utilisation de la programmation parallèle dans ce but.

0.1 La connaissance de la bathymétrie et le maillage

Lorsque l'on veut faire une simulation du cours d'eau d'une rivière, le premier obstacle est la connaissance précise de la bathymétrie. La bathymétrie correspond aux relevés de la profondeur de la rivière et sa connaissance précise est cruciale pour que les simulations numériques aient une chance de correspondre à la réalité.

Dans cet objectif, des relevés bathymétriques sont effectués dans les rivières et fleuves que l'on veut simuler. Ces campagnes de mesures sont souvent coûteuses, d'autant plus que la précision demandée est importante. Plusieurs méthodes peuvent être employées pour faire ces relevés. Des lasers peuvent être utilisés pour mesurer la profondeur de l'eau mais ils se limitent souvent à des eaux très peu profondes ou parfaitement limpides. Le reste du temps, ce sont des versions plus ou moins sophistiquées de sonar qui sont utilisées.

Ces relevés bathymétriques permettent de générer un maillage du fond de la rivière. Dans le cadre de l'étude d'inondations le domaine à mailler comprend le domaine fluvial ainsi que les rives qui pourront être inondées lors des simulations. Les données bathymétriques doivent donc être couplées à des données topographiques pour créer un maillage complet du domaine d'étude. La création d'un tel maillage nécessite aussi de prendre compte tout un tas d'obstacles dans la rivière, la présence d'îles, de piles de pont et toutes autres structures.

La création du maillage d'un domaine de simulation est une tâche longue et demandera souvent une étape de validation durant laquelle plusieurs résultats de simulations seront comparés à des relevés de hauteur d'eau réels pour assurer une bonne précision. On se servira donc, dans un premier temps, de simulations pour lesquelles des relevés de hauteur d'eau sont connus pour valider le maillage. On pourra, dans un second temps, faire une étude des inondations dans des situations plus critiques pour lesquelles des relevés de hauteur d'eau ne sont pas présents.

0.2 Différents objectifs de simulation

Lorsque l'on traite un nouveau domaine de simulation, il y a deux phases importantes à traiter.

La première consiste à générer une solution stabilisée, c'est-à-dire au régime permanent, sur le domaine pour un débit d'entrée moyen de la rivière. On considère que le régime permanent est atteint lorsque le débit de sortie de la rivière est égal au débit d'entrée pendant une longue période. Cette phase est souvent la plus complexe car l'initialisation de la solution sur tout le domaine pose plusieurs problèmes. On a rarement des relevés précis de hauteur d'eau dans toute la rivière et encore moins des relevés de vitesse. Il nous faut alors choisir arbitrairement ces paramètres pour les premières simulations. Les paramètres ainsi choisis mèneront plus ou moins rapidement à une solution stabilisée. Beaucoup de bancs couvrant-découvrant seront présents et mettront en difficulté la stabilité des différents schémas numériques. On pourra alors utiliser plusieurs stratégies d'initialisation pour converger le plus rapidement vers la solution permanente, parfois au prix d'une solution transitoire qui n'est pas physique.

La seconde phase est celle qui présente le réel intérêt pour faire l'étude des inondations. Lors de cette phase on initialisera la simulation à partir de la solution stable calculée précédemment. On lancera ensuite plusieurs simulations en faisant varier certains paramètres. On pourra par exemple simuler de fortes intempéries en augmentant le débit d'entrée de la rivière. Le but de cette seconde phase est de générer une base de données de résultats de simulations lancées avec

différents paramètres (débit d'entrée, hauteur de sortie, coefficient de frottement). On pourra par la suite utiliser cette base de données pour faire des études statistiques des inondations (Abdedou & Soulaïmani, 2018 ; Fahsi, Soulaïmani & Tchamen, 2010 ; Zokagoa & Soulaïmani, 2012). On peut aussi utiliser ces bases de données pour entraîner des algorithmes de *machine learning* (Jacquier, Abdedou, Delmas & Soulaïmani, 2020) pour prédire en un temps très faible des situations qui n'ont pas été simulées.

0.3 Le besoin d'un calcul rapide

Que ce soit pour la première ou la seconde phase des simulations, on veut toujours pouvoir traiter des domaines les plus grands possibles le plus rapidement possible.

La possibilité de traiter de très grands maillages comprenant un nombre de cellules très important est cruciale pour avoir une représentation précise de la bathymétrie du domaine. Plus le maillage aura un grand nombre d'éléments, plus il sera simple d'y traduire les relevés bathymétriques et plus le maillage pourra être raffiné dans les zones critiques de la rivière (barrages, piles de ponts, îles). Un maillage plus raffiné combiné à des mesures de bathymétrie précises permettra à coup sûr d'avoir de meilleures solutions.

D'autre part, si les calculs sont plus rapides on pourra simuler bien plus de cas d'inondations différents dans un même temps. Plus on simulera de cas différents, plus les bases de données seront fournies ce qui permettra de diminuer les incertitudes sur les prédictions.

Pour atteindre ces objectifs on se tourne vers de la programmation parallèle pour distribuer les calculs et/ou la mémoire sur plusieurs unités de calculs. Une revue de littérature des différentes méthodes utilisées pour faire de la programmation parallèle dans le cadre de la résolution des équations de St-Venant sera présentée dans le chapitre 1. On souhaite pour finir cette introduction

présenter les raisons qui nous poussent à vouloir utiliser des GPU, puis pourquoi on veut coupler leur utilisation à MPI.

0.4 Pourquoi utiliser un GPU ?

Le GPU : *Graphical Processing Unit* est un coprocesseur qui a été historiquement utilisé pour faire du calcul graphique beaucoup plus rapidement que le CPU : *Central Processing Unit*. Dans cet objectif le GPU est construit avec une architecture SIMD : *Single Instruction Multiple Data* qui lui permet d'effectuer les mêmes opérations sur une grande quantité de données.

Il a vite été remarqué qu'une telle architecture pouvait être particulièrement adaptée à certaines simulations numériques dans lesquelles les mêmes opérations sont répétées sur des données différentes.

Dans notre cas, avec une méthode volumes finis explicite en temps, porter le code sur GPU n'est pas une tâche très complexe, bien que l'optimisation requise pour atteindre les plus hauts *speed-up* peut être longue. Comme on le verra, ce qui rend l'adaptation d'un code sur GPU relativement facile est la manière incrémentale avec laquelle ce portage peut être fait.

Finalement, il est relativement simple de trouver de nos jours des ordinateurs fixes ou même portables qui contiennent un GPU et sur lesquels un tel code peut être lancé. Le code est donc relativement portable même si des étapes de configuration du système sont requises.

0.5 Pourquoi utiliser MPI et une décomposition de domaine ?

Même si l'utilisation d'un GPU permet déjà d'obtenir des *speed-up* remarquables, on souhaite dans ce travail utiliser plusieurs GPU pour chaque simulation.

Cette volonté vient du fait que l'on veut pouvoir traiter des domaines toujours plus grands sans perdre en précision du maillage. On veut par exemple faire l'étude couplée de plusieurs tronçons de rivières plutôt que de les étudier séparément. Dans ce cadre, il est assez naturel de vouloir utiliser MPI : *Message Passing Interface* couplée à une décomposition de domaine pour distribuer les calculs et la mémoire sur plusieurs CPU, chacun utilisant un GPU pour faire leurs calculs.

On verra dans la suite du travail qu'un maillage fourni par la Communauté métropolitaine de Montréal tombe complètement dans cette catégorie et même si le nombre d'éléments du maillage n'est pas suffisamment élevé, c'est pour résoudre ce genre de problèmes que **CUTEFLOW** a été porté sur une architecture multi-GPU.

En contre-partie, l'utilisation de plusieurs GPU rend le code moins portable comme il requiert l'accès à des grappes de calculs (*clusters*) de haute performance. Cela nous permet, en plus de la puissance de calcul, d'avoir accès à un grand espace de stockage pour les différents résultats de simulations.

0.6 Objectifs du mémoire

L'objectif général du mémoire est d'améliorer les performances du code **CUTEFLOW** pour la résolution des équations de St-Venant. On décide dans ce but d'utiliser MPI et CUDA-Fortran pour porter le code sur une architecture multi-GPU.

On aura alors plusieurs objectifs spécifiques :

- Développer un pré-traitement pour faire une décomposition de domaine en utilisant la librairie METIS,
- Utiliser OpenMPI dans **CUTEFLOW** pour faire les échanges mémoire entre sous-domaines,

- Développer des codes de post-traitement pour recombiner les solutions de chaque sous-domaine sur le domaine original,
- Participer à l'amélioration globale du code **CUTEFLOW** pour l'utilisation de domaines de plus en plus grands, par exemple en ajoutant le traitement de plusieurs entrées et sorties.

0.7 Plan du mémoire

Le mémoire sera divisé en plusieurs chapitres. En premier, nous présenterons les équations de St-Venant et leurs résolutions par une méthode volumes finis explicite en temps. Nous présenterons les solveurs de Riemann utilisés lors de cette étape et combinerons plusieurs travaux pour avoir les meilleurs résultats possibles.

Nous présenterons ensuite les concepts classiques de la programmation GPU. Nous aborderons les différents concepts via un exemple simple que nous porterons sur GPU en CUDA-Fortran. Nous ferons le lien avec l'utilisation de la programmation GPU qui est faite dans **CUTEFLOW**.

Nous rentrerons ensuite dans le cœur du portage multi-GPU en commençant par présenter le pré-traitement mis en place pour faire la décomposition de domaines. Nous utiliserons METIS (Karypis & Kumar, 2009) pour faire la décomposition et traiterons avec une attention particulière la numérotation des mailles à envoyer et recevoir par chaque sous-domaine.

L'utilisation d'OpenMPI (Gabriel, Fagg, Bosilca, Angskun, Dongarra, Squyres, Sahay, Kambadur, Barrett, Lumsdaine, Castain, Daniel, Graham & Woodall, 2004) sera ensuite rapidement présentée. Une discussion sur l'utilité d'une version CUDA-*Aware* ainsi que sur la difficulté d'avoir une telle version sera tenue.

Finalement, après avoir présenté le fonctionnement général du code **CUTEFLOW**, nous présenterons les résultats obtenus sur différents domaines, en premier sur un cas de bris de barrage classique puis sur le domaine de la rivière des Mille Îles et sur le domaine de l'archipel

de Montréal. Nous calculerons les *speed-up* et efficacités pour différentes tailles de maillages et conclurons quant aux performances de la version multi-GPU de **CUTEFLOW** développée ici.

CHAPITRE 1

REVUE DE LITTÉRATURE

Dans cette revue de littérature on se concentre sur l'utilisation de la programmation parallèle sur GPU dans le cadre de la résolution numérique d'équations aux dérivées partielles par des méthodes explicites, en particulier dans le domaine de la CFD.

1.1 L'utilisation de GPU pour accélérer les calculs

Comme on l'a dit en introduction, les GPU : *Graphical Processing Unit*, ont historiquement été créés pour faire des calculs graphiques bien plus rapidement que les CPU : *Central Processing Unit*. Ils atteignent cet objectif avec un architecture SIMD : *Single Instruction Multiple Data* qui leur permet d'effectuer les mêmes calculs sur des données différentes. Il a vite été remarqué que ce genre d'architecture pourrait être très puissante dans le cadre de simulations numériques mais jusqu'en 2006 il était très difficile de les programmer, les langages comme OpenGL et Direct3D étaient uniquement adaptés au calcul graphique.

Cela change en 2007 avec la première version de CUDA : *Compute Unified Device Architecture* présenté par NVIDIA. CUDA est une architecture qui permet aux GPU d'être programmés pour faire des calculs généraux. Au départ, NVIDIA présente un compilateur pour le CUDA C et le CUDA C++, plus tard, à la suite d'une collaboration entre PGI et NVIDIA en 2009, un compilateur pour le CUDA Fortran est développé.

Bien que CUDA ait été la première façon de programmer des GPU pour des calculs généraux, on peut de nos jours programmer ces GPU de différentes manières. Par exemple avec des directives compilateurs, en utilisant OpenACC, spécialement développé pour faciliter la programmation des GPU, ou encore, en utilisant OpenCL, une API développé par le *Khronos Group* conçu pour programmer des systèmes hétérogènes, et en autres des GPU. Ce dernier est très proche du CUDA C et présente l'avantage d'être utilisable sur toutes sortes d'accélérateurs.

Le CUDA C,C++ ou Fortran n'est utilisable qu'avec les GPU de NVIDIA. On pourrait penser que cela réduit son champ d'application. Cependant, on peut trouver sur le site (https://www.top500.org) des statistiques sur les 500 super-calculateurs les plus puissants dans le monde. En date de Juin 2020, ils utilisent en majorité des GPU, et ce sont alors toujours des GPU de NVIDIA. L'utilisation de CUDA C,C++ ou Fortran n'est donc en rien restrictive lorsque l'on projette d'utiliser ce genre de grappes de calculs.

Depuis la première version de CUDA en 2007, beaucoup de travaux ont été effectués pour évaluer les performances des GPU dans le cadre de simulations numériques. On peut trouver dans l'introduction de Brodtkorb, Sætra & Altinakar (2012) une explication des difficultés à comparer les performances des GPU à celles des CPU, on en reprend les grandes lignes ici.

Comparer une version séquentielle sur CPU à une version sur GPU n'est pas chose facile parce que les deux processeurs ont des architectures très différentes, souvent les accélérations, *speed-up*, reportés sont à prendre avec précaution. Par exemple on peut trouver sur NVIDIA (2020), différents articles qui reportent des *speed-up* impressionnants pour des versions GPU par rapport à leur équivalent sur CPU. Beaucoup d'entre eux favorisent les versions GPU, soit en utilisant des versions CPU non optimisées soit en utilisant des CPU d'ancienne génération et il faut donc être particulièrement attentif aux résultats présentés.

Même si une attention particulière doit être portée aux comparaisons entre des versions séquentielles sur CPU et des versions sur GPU, on trouve dans la littérature des *speed-up* importants en particulier dans la résolution par la méthodes des volumes finis explicites en temps des équations de St-Venant. On se concentre ici sur des méthodes explicites en temps, une discussion de la difficulté de porter des méthodes implicites sur GPU est faite dans Aissa, Verstraete & Vuik (2017).

On peut trouver dans De la Asunción, Mantas & Castro (2010) ; De la Asunción, Castro, Fernández-Nieto, Mantas, Acosta & González-Vida (2013) des *speed-up* de l'ordre de 10 à 20 des versions sur GPU par rapport aux versions séquentielles sur CPU. Dans Niksiar, Ashrafizadeh, Shams & Madani (2014) des *speed-up* de l'ordre de 40 sont rapportés. Avec l'utilisation de

meilleurs GPU, dans Vacondio, Dal Palù & Mignosa (2014) des *speed-up* encore plus importants sont atteints.

On trouve l'utilisation de GPU pour la résolution des équations de St-Venant dans Brodtkorb, Hagen, Lie & Natvig (2010) ; Brodtkorb *et al.* (2012) ; Escalante, Morales de Luna & Castro (2018) ; Smith & Liang (2013). Dans Brodtkorb *et al.* (2010,1) ; Escalante *et al.* (2018) c'est du CUDA C,C++ ou Fortran qui a été utilisé alors que dans Smith & Liang (2013) c'est OpenCL qui a été choisi.

1.2 Le passage au multi-GPU pour utiliser plusieurs GPU

Il est apparu assez clairement que pour pouvoir traiter les plus gros problèmes il faudrait combiner l'utilisation de plusieurs CPU et de plusieurs GPU. Dans Smith & Liang (2013) ou une implémentation sur un GPU est présentée, l'implémentation multi-GPU fait partie des améliorations envisagées en conclusion.

Parmi les raisons qui poussent à vouloir utiliser plusieurs GPU, la quantité limitée de mémoire, en particulier sur les premiers GPU, est un facteur important. De nos jours la mémoire des GPU a bien augmenté mais il reste beaucoup à gagner à partager la résolution d'un problème sur plusieurs GPU.

Plusieurs méthodes sont disponibles pour utiliser plusieurs GPU. Si on se restreint à un seul nœud de calcul on peut se contenter du CUDA C,C++ ou Fortran, ou y combiner OpenMP pour avoir aussi plusieurs *threads* sur le CPU. De nos jours, les systèmes les plus courants offrent 4 GPU par nœud, c'est par exemple le cas sur BELUGA la grappe de calcul la plus récente de *Compute Canada*. Si notre objectif est d'utiliser plus d'un nœud de calcul, plus de 4 GPU sur BELUGA, il va falloir passer à l'utilisation de MPI. Beaucoup de travaux sont allés dans cette direction.

Un des premiers articles utilisant MPI et CUDA pour résoudre de gros problèmes est Komatitsch, Erlebacher, Göddeke & Michéa (2010). Dans cet article l'utilisation de MPI permet d'utiliser

plusieurs processeurs sur des nœuds de calculs différents, chacun étant associé à un GPU. Ils utilisent alors jusqu'à 192 GPU pour faire des simulations de propagation d'ondes sismiques. Ils couplent MPI à une décomposition de domaine pour répartir correctement la mémoire entre les processeurs. Dans cet article des GPU avec 4G de mémoire sont utilisés et la taille du problème est dictée par le nombre de GPU disponibles.

De la même façon dans Jacobsen, Thibault & Senocak (2010), la programmation en CUDA et MPI est utilisée pour accélérer des calculs d'écoulements compressibles. L'utilisation de grappes de calculs et de programmation multi-GPU est jugée très prometteuse et les auteurs concluent en postulant le fait que ce type de programmation sera largement utilisée dans le futur. A la vue du top 500 (https://www.top500.org) en Juin 2020, on peut dire qu'ils ne se sont pas trompés. On peut trouver le même résultat dans Lai, Li, Tian & Zhang (2019) où MPI est couplé à du CUDA C dans le cadre de calculs d'écoulements hypersoniques.

L'utilisation de MPI et CUDA est présentée dans Viñas, Lobeiras, Fraguela, Arenaz, Amor, García, Castro & Doallo (2013) dans le cadre de la résolution des équations de St-Venant. Des maillages sous forme de grille sont utilisés avec un nombre d'éléments allant jusqu'à un million, en utilisant 4 GPU, ce qui permet de garder des temps de calculs largement inférieurs au temps de simulation. Ils calculent dans ce cas 7 jours de simulation en seulement 28 minutes. Évidement, ici aussi, il faut prendre les résultats avec précaution, ce temps peu fortement varier selon le maillage, il suffit que celui-ci soit raffiné dans une zone pour contraindre le pas de temps dans tout le domaine et donner des temps de calculs bien plus longs.

Récemment dans Turchetto, Dal Palù & Vacondio (2020) on trouve une méthode générale qui combine aussi une décomposition de domaine à MPI et CUDA. L'exemple de la résolution des équations de St-Venant est pris et des maillages cartésiens sont utilisés. La décomposition de domaine est faite à l'aide de courbes de Hilbert et même si le maillage est sous forme de grille, il peut être raffiné dans certaines zones.

Bien que ces articles utilisent MPI et CUDA, la plupart n'utilisent pas une version CUDA-*Aware* de MPI. Une telle version permet d'échanger directement la mémoire des GPU en évitant des

copies mémoire inutiles entre CPU et GPU. La raison souvent mise en avant est le fait qu'utiliser du CUDA-*Aware* MPI rend le code moins portable (Turchetto *et al.*, 2020). Cependant plusieurs études ont montré la supériorité du CUDA-*Aware* MPI pour avoir des échanges les plus rapides possibles. On trouve une telle étude dans Khorassani, Chu, Subramoni & Panda (2019) où il est clairement mis en avant la puissance de CUDA-*Aware* MPI. Malgré certaines limitations liées aux différentes implémentations et au support des différentes architectures des grappes de calculs, les librairies supportent de mieux en mieux le CUDA-Aware MPI et permettent d'avoir des échanges mémoire toujours plus rapides.

Les approches citées jusqu'à présent se concentrent sur l'utilisation d'un ou plusieurs GPU en laissant très peu de calculs sur le CPU. Certaines approches comme dans, Xu, Zhang, Deng, Fang, Wang, Cao, Che, Wang & Liu (2014), ont une vision plus collaborative de la répartition de calculs sur des grappes de calculs. Dans ce genre d'approche, l'objectif et de répartir la charge de calcul de la manière la plus optimisée possible entre les CPU et les GPU en prenant en compte leurs forces et leurs faiblesses. Ce genre de méthode permet d'avoir de meilleurs temps de calcul au prix d'un temps de programmation et d'optimisation très long.

Pour conclure cette revue de littérature, on constate que l'utilisation de plusieurs GPU est très présente dans le domaine de la CFD, en particulier l'utilisation de MPI et CUDA pour résoudre de problèmes de grandes tailles. Dans ce travail on propose de porter le code **CUTEFLOW** sur une architecture multi-GPU en utilisant MPI et CUDA et en utilisant une version CUDA-*Aware* d'OpenMPI. On travaillera avec des maillages non structurés et on fera la décomposition de domaine grâce à la bibliothèque METIS (Karypis & Kumar, 2009) comme suggéré dans Komatitsch *et al.* (2010).

CHAPITRE 2

MODÉLISATION DES ÉCOULEMENTS À SURFACE LIBRE À FAIBLE PROFONDEUR

2.1 Présentation des équations de St-Venant

Dans cette section on part des principes de conservation de la masse et de la quantité de mouvement, c'est-à-dire, les équations de Navier-Stokes, pour trouver les équations de St-Venant qui décrivent les écoulements à surface libre à faible profondeur (de tels écoulement se trouvent dans les rivières, les lacs, les estuaires). C'est une démonstration très classique et nous nous inspirerons en partie de celle faite dans Toro (2001).

Sous leur forme générale on peut écrire les équations de Navier-Stokes de la façon suivante,

$$\rho_t + \nabla.(\rho \mathbf{V}) = 0 \tag{2.1}$$

$$(\rho \mathbf{V})_t + \nabla.[(\rho \mathbf{V}) \otimes \mathbf{V} + pI - \sigma] = \rho \mathbf{g} \tag{2.2}$$

où ρ est la masse volumique, $\mathbf{V}$ le champ de vitesse et σ est le tenseur des contraintes visqueuses. Sous forme indicielle on a,

$$\begin{cases} \rho_t + u\rho_x + v\rho_y + w\rho_z + \rho(u_x + v_y + w_z) & = & 0, \\ u_t + uu_x + vu_y + wu_z + \frac{1}{\rho}p_x - \frac{1}{\rho}\sigma_{11x} - \frac{1}{\rho}\sigma_{12y} - \frac{1}{\rho}\sigma_{13z} & = & g_1, \\ v_t + uv_x + vv_y + wv_z + \frac{1}{\rho}p_y - \frac{1}{\rho}\sigma_{21x} - \frac{1}{\rho}\sigma_{22y} - \frac{1}{\rho}\sigma_{23z} & = & g_2, \\ w_t + uw_x + vw_y + ww_z + \frac{1}{\rho}p_z - \frac{1}{\rho}\sigma_{31x} - \frac{1}{\rho}\sigma_{32y} - \frac{1}{\rho}\sigma_{33z} & = & g_3. \end{cases} \tag{2.3}$$

On utilise ces équations pour définir le problème d'écoulement à surface libre. On choisit l'axe z vertical et on définit le fond du domaine par la fonction

$$z = b(x, y)$$

et la surface libre est définie par

$$z = s(x, y, t) = b(x, y) + h(x, y, t) \tag{2.4}$$

où $h(x, y, t)$ est la hauteur de la colonne d'eau en un point.

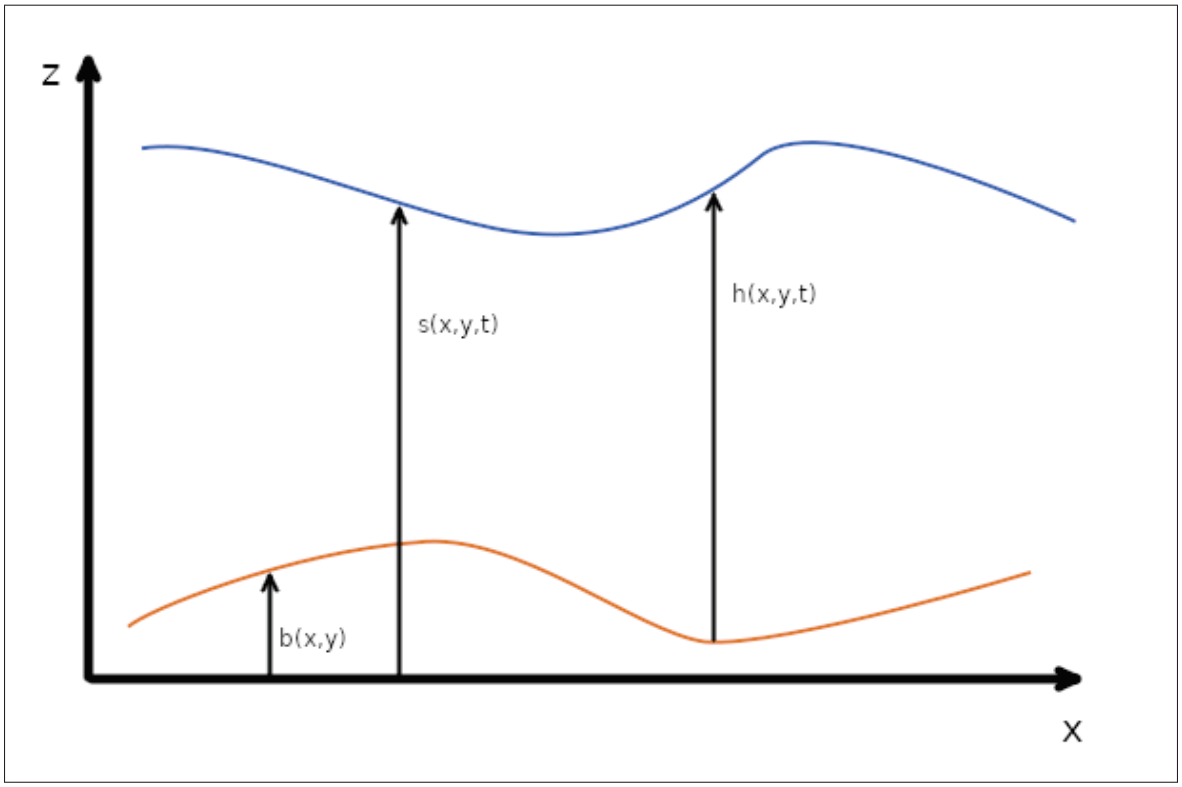


Figure 2.1 Illustration des notations choisies.

En supposant la densité constante on réécrit les équations du problème de la façon suivante.

$$\begin{cases} u_x + v_y + w_z &= 0, \\ u_t + uu_x + vu_y + wu_z &= -\frac{1}{\rho}p_x + \frac{1}{\rho}\sigma_{11x} + \frac{1}{\rho}\sigma_{12y} + \frac{1}{\rho}\sigma_{13z}, \\ v_t + uv_x + vv_y + wv_z &= -\frac{1}{\rho}p_y + \frac{1}{\rho}\sigma_{21x} + \frac{1}{\rho}\sigma_{22y} + \frac{1}{\rho}\sigma_{23z}, \\ w_t + uw_x + vw_y + ww_z &= -g - \frac{1}{\rho}p_z + \frac{1}{\rho}\sigma_{31x} + \frac{1}{\rho}\sigma_{32y} + \frac{1}{\rho}\sigma_{33z}. \end{cases} \tag{2.5}$$

Avant de dériver les équations d'écoulement en eau peu profonde il faut s'intéresser à la définition de la surface libre et du lit de l'écoulement. Pour la surface libre on définit,

$$\psi_{surface}(x, y, z, t) = z - s(x, y, t) \tag{2.6}$$

et pour le lit on définit

$$\psi_{lit}(x, y, z, t) = z - b(x, y) \tag{2.7}$$

On impose ensuite deux conditions aux limites, la condition cinématique,

$$\frac{d}{dt}\psi(x, y, z, t) = \psi_t + u\psi_x + v\psi_y + w\psi_z = 0 \tag{2.8}$$

et la condition dynamique

$$p(x, y, z, t)_{z=s(x,y)} = p_{atm} = 0. \tag{2.9}$$

Pour les contraintes visqueuses, au fond du domaine on a,

$$\begin{aligned} \sigma_{b1} &= \sigma_{11}|_{z=b} b_x + \sigma_{12}|_{z=b} b_y - \sigma_{13}|_{z=b} \\ \sigma_{b2} &= \sigma_{21}|_{z=b} b_x + \sigma_{22}|_{z=b} b_y - \sigma_{23}|_{z=b} \end{aligned} \tag{2.10}$$

et à la surface,

$$\begin{aligned} \sigma_{s1} &= -\sigma_{11}|_{z=s} s_x - \sigma_{12}|_{z=s} s_y + \sigma_{13}|_{z=s} \\ \sigma_{s2} &= -\sigma_{21}|_{z=s} s_x - \sigma_{22}|_{z=s} s_y + \sigma_{23}|_{z=s} \end{aligned} \tag{2.11}$$

σ_s correspond aux contraintes sur la surface libre, par exemple, celles dues au vent.
σ_b correspond aux contraintes de frottements au fond du domaine qui sont données par l'équation de Chezy,

$$\begin{cases} \sigma_{b1} = \dfrac{gu\sqrt{u^2+v^2}}{C_f^2}, \\ \sigma_{b2} = \dfrac{gv\sqrt{u^2+v^2}}{C_f^2} \end{cases} \tag{2.12}$$

où C_f est le coefficient de Chezy, qui peut aussi s'exprimer en fonction du nombre de Manning n comme $C_f = h^{1/6}/n$.

Pour dériver les équations d'écoulement en eau peu profondes à partir de 2.5 et des conditions aux limites 2.8 et 2.9, on suppose que la composante verticale de l'accélération donnée par

$$\frac{dw}{dt} = w_t + uw_x + vw_y + ww_z \tag{2.13}$$

est négligeable. En utilisant ce résultat dans la dernière équation de 2.5 on obtient

$$p_z = -\rho g \tag{2.14}$$

Puis en utilisant la condition dynamique 2.9 on a finalement,

$$p = \rho g(s - z) \tag{2.15}$$

La différenciation de 2.15 donne

$$p_x = \rho g s_x, \quad p_y = \rho g s_y. \tag{2.16}$$

Ce qui donne dans 2.5,

$$u_t + uu_x + vu_y + wu_z = -gs_x + \frac{1}{\rho}\sigma_{11x} + \frac{1}{\rho}\sigma_{12y} + \frac{1}{\rho}\sigma_{13z} \tag{2.17}$$

et

$$v_t + uv_x + vv_y + wv_z = -gs_y + \frac{1}{\rho}\sigma_{21x} + \frac{1}{\rho}\sigma_{22y} + \frac{1}{\rho}\sigma_{23z} \tag{2.18}$$

Une étape importante pour avoir les équations d'écoulements en eau peu profonde sous leur forme classique est de transformer l'équation de continuité en l'intégrant sur la hauteur du domaine de la manière suivante,

$$\int_b^s (u_x + v_y + w_z)dz = 0, \tag{2.19}$$

ce qui donne

$$w|_{z=s} - w|_{z=b} + \int_b^s u_x dz + \int_b^s v_y dz = 0. \tag{2.20}$$

On applique ensuite 2.8 à $\psi_{surface}$ définie en 2.6 pour déterminer les termes $w|_{z=s}$ et $w|_{z=b}$.

$$(-s_t - us_x - vs_y + w)|_{z=s} = 0 \tag{2.21}$$

Il en est de même pour ψ_{lit},

$$(-ub_x - vb_y + w)|_{z=b} = 0. \tag{2.22}$$

On obtient donc

$$w|_{z=s} = (s_t + us_x + vs_y)|_{z=s}, \tag{2.23}$$

et

$$w|_{z=b} = (ub_x + vb_y)|_{z=b}. \tag{2.24}$$

Finalement on peut réécrire 2.20 de la façon suivante,

$$(s_t + us_x + vs_y)|_{z=s} - (ub_x + vb_y)|_{z=b} + \int_b^s u_x dz + \int_b^s v_y dz = 0. \tag{2.25}$$

On utilise ensuite la formule de Leibniz pour simplifier cette expression,

$$\frac{d}{d\alpha}\int_{\xi_1(\alpha)}^{\xi_2(\alpha)} f(\xi,\alpha)d\xi = \int_{\xi_1(\alpha)}^{\xi_2(\alpha)} \frac{\partial f}{\partial \alpha}d\xi + \frac{d\xi_2}{d\alpha}f(\xi_2,\alpha) - \frac{d\xi_1}{d\alpha}f(\xi_1,\alpha) \tag{2.26}$$

ce qui nous permet d'écrire les termes intégraux de 2.25 de la façon suivante,

$$\int_b^s u_x dz = \frac{d}{dx}\int_b^s u dz - s_x u|_{z=s} + b_x u|_{z=b} \tag{2.27}$$

et

$$\int_b^s v_y dz = \frac{d}{dy}\int_b^s v dz - s_y v|_{z=s} + b_y v|_{z=b} \tag{2.28}$$

ce qui donne dans 2.25,

$$s_t + \frac{d}{dx}\int_b^s u dz + \frac{d}{dy}\int_b^s v dz = 0 \tag{2.29}$$

On définit alors les vitesses moyennées sur la hauteur d'eau comme,

$$\bar{u} = \frac{1}{h}\int_b^s u dz, \text{ et, } \bar{v} = \frac{1}{h}\int_b^s v dz \tag{2.30}$$

Ensuite comme $s = b + h$ et $b_t = 0$, on peut réécrire 2.29 sous sa forme finale,

$$h_t + (h\bar{u})_x + (h\bar{v})_y = 0. \tag{2.31}$$

On procède ensuite de la même manière pour les équations 2.17 et 2.18. On les intègre sur la hauteur d'eau et on applique la formule de Leibniz, on peut trouver le développement dans l'annexe I. Ensuite en négligeant les contraintes $\sigma 11$, $\sigma 12$ et $\sigma 22$ par rapport aux contraintes $\sigma 13$ et $\sigma 23$ dans les équations 2.10 et 2.11, on obtient, pour 2.17,

$$(h\bar{u})_t + (h\bar{u}^2 + \frac{1}{2}gh^2)_x + (h\bar{u}\bar{v})_y = -ghb_x + \frac{1}{\rho}\left[\sigma_{s1} - \sigma_{b1}\right], \tag{2.32}$$

et pour 2.18

$$(h\bar{v})_t + (h\bar{u}\bar{v})_x + (h\bar{v}^2 + \frac{1}{2}gh^2)_y = -ghb_y + \frac{1}{\rho}\left[\sigma_{s2} - \sigma_{b2}\right]. \tag{2.33}$$

Les trois équations 2.31, 2.32 et 2.33 peuvent s'écrire sous leurs formes conservatives dans une seule équation vectorielle,

$$U_t + F(U)_x + G(U)_y = S(U), \tag{2.34}$$

avec

$$U = \begin{bmatrix} h \\ h\bar{u} \\ h\bar{v} \end{bmatrix}, \qquad F(U) = \begin{bmatrix} h\bar{u} \\ h\bar{u}^2 + \frac{1}{2}gh^2 \\ h\bar{u}\bar{v} \end{bmatrix},$$

$$G(U) = \begin{bmatrix} h\bar{v} \\ h\bar{u}\bar{v} \\ h\bar{v}^2 + \frac{1}{2}gh^2 \end{bmatrix}, \qquad S(U) = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix}.$$

On peut remettre cette équation sous forme intégrale pour accepter les discontinuités de la manière suivante,

$$\frac{\partial}{\partial t}\int_\Omega U dV + \int_{\partial\Omega} n.H(U) dS = \int_\Omega S(U) dV, \tag{2.35}$$

avec $H(U) = (F(U), G(U))$

Le terme source de l'équation 2.34 peut être séparé en deux, une partie pour la bathymétrie et une partie pour la friction. On notera dans la suite,

$$S(U) = S_O(U) + S_f(U) \tag{2.36}$$

Le terme de bathymétrie S_O dérive directement de 2.32 et 2.33,

$$S_O = (0, -ghb_x, -ghb_y). \tag{2.37}$$

Le terme de friction S_f que nous utiliserons est obtenu à partir de 2.32 et 2.33, en négligeant σ_s, et en utilisant l'expression 2.12 pour σ_b, ce qui donne,

$$S_f = (0, -gh\frac{n^2\bar{u}\sqrt{\bar{u}^2+\bar{v}^2}}{h^{4/3}}, -gh\frac{n^2\bar{v}\sqrt{\bar{u}^2+\bar{v}^2}}{h^{4/3}}). \tag{2.38}$$

avec n le nombre de manning.

2.2 Résolution numérique des équations de St-Venant par la méthode des volumes finis

La résolution numérique de ces équations a été faite par plusieurs méthodes : SPH dans Ata & Soulaïmani (2005a) ; éléments finis dans Dhatt, Soulaimani, Ouellet & Fortin (1986) ; Fortin *et al.* (1993) ; Soulaïmani (1983) ; volumes finis dans Ata, Pavan, Khelladi & Toro (2013) ; Audusse & Bristeau (2005) ; Loukili & Soulaïmani (2007). On utilise dans ce travail une méthode volumes finis centrée sur les cellules en combinant le schéma HLLC et la semi-implicitation des termes de frictions de Loukili & Soulaïmani (2007) avec le traitement des termes de bathymétrie présenté dans Ata *et al.* (2013) s'inspirant de Audusse & Bristeau (2005).

2.2.1 Discrétisation par des schémas de volumes finis

La méthode des volumes finis se base sur un pavage du domaine d'étude en volumes qu'on prendra ici triangulaires. On commence par intégrer les équations de St-Venant sur chaque volume Ω_i ce qui donne en appliquant le théorème de la divergence,

$$\int_{\Omega_i} \frac{\partial U}{\partial t} dV + \int_{\partial\Omega_i} H(U).n_i dS = \int_{\Omega_i} S(U) dV \tag{2.39}$$

avec $H(U) = (F, G)^T$, et n_i la normale unitaire de $\partial\Omega_i$ sortante de Ω_i

On fait ensuite les approximations,

$$U_i = \frac{1}{|\Omega_i|} \int_{\Omega_i} U dV \tag{2.40}$$

et,

$$S_i(U) = \frac{1}{|\Omega_i|} \int_{\Omega_i} S(U) dV \tag{2.41}$$

Ce qui donne pour 2.39,

$$|\Omega_i| \frac{dU_i}{dt} = -\sum_{j=1}^{3} L_{ij} H(U).n_{ij} + |\Omega_i| S_i(U) \tag{2.42}$$

avec $|\Omega_i|$ l'aire de l'élément Ω_i (triangulaire), L_{ij} la longueur du côté j de l'élément Ω_i et n_{ij} la normale au coté j sortante de Ω_i.

On peut ici aussi séparer le terme source, S_i, en deux parties, S_{O_i} pour la bathymétrie, et S_{f_i} pour la friction,

$$\begin{cases} S_{O_i}(U) = & \frac{1}{|\Omega_i|} \int_{\Omega_i} S_O(U) dV \\ S_{f_i}(U) = & \frac{1}{|\Omega_i|} \int_{\Omega_i} S_f(U) dV \end{cases} \tag{2.43}$$

Ce qui donne donc pour 2.42,

$$|\Omega_i| \frac{dU}{dt} = -\sum_{j=1}^{3} L_{ij} H(U).n_{ij} + |\Omega_i| S_{O_i}(U) + |\Omega_i| S_{f_i}(U) \tag{2.44}$$

On utilise l'invariance par rotation entre G et H de chaque côté d'une arête (Hemker & Spekreijse (1985) ; Loukili & Soulaïmani (2007) ; Toro (2001)) qui donne,

$$H(U).n_{ij} = T_{n_{ij}}^{-1} G(T_{n_{ij}} U), \quad T_{n_{ij}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & n_{ij}^1 & n_{ij}^2 \\ 0 & -n_{ij}^2 & n_{ij}^1 \end{bmatrix} \tag{2.45}$$

Dans cette approche on suppose une méthode des volumes finis centrée sur les cellules avec les variables constantes par morceaux sur les cellules. Pour approximer les flux on va résoudre un problème de Riemann unidirectionnel dans la direction n_{ij}. On peut ainsi réécrire 2.44 de la façon suivante,

$$|\Omega_i| \frac{dU_i}{dt} = -\sum_{j=1}^{3} L_{ij} T_{n_{ij}}^{-1} \tilde{G}(T_{n_{ij}} U_i, T_{n_{ij}} U_j) + |\Omega_i| S_{O_i}(U) + |\Omega_i| S_{f_i}(U) \tag{2.46}$$

avec $\tilde{G}(T_{n_{ij}} U_i, T_{n_{ij}} U_j) = \tilde{G}(U_L, U_R)$ un flux discret que l'on trouve en résolvant un problème de Riemann avec $U_L = T_{n_{ij}} U_i$ et $U_R = T_{n_{ij}} U_j$ comme états initiaux, soit

$$\begin{cases} \dfrac{\partial U}{\partial t} + \dfrac{\partial G(U)}{\partial x_n} = 0, \\ U(x,0) = \begin{cases} U_L \text{ si } x_n < 0, \\ U_R \text{ si } x_n > 0. \end{cases} \end{cases} \tag{2.47}$$

En ce qui concerne les conditions aux limites, il y en a de plusieurs types et on choisit ici aussi de procéder comme dans Loukili & Soulaïmani (2007). Une condition transmissive sera résolue en supposant un état $U_R = U_L$ dans la résolution du problème de Riemann. Pour une condition avec un débit entrant Q on calcule le flux directement avec $\tilde{G}(U_L) = (Q, \frac{Q^2}{h_l} + \frac{g h_l^2}{2}, 0)^T$. Pour une condition de mur non transmissive on utilise le calcul précédant avec un débit entrant nul, soit $\tilde{G}(U_L) = (0, \frac{g h_l^2}{2}, 0)^T$.

La discrétisation temporelle est traitée avec une méthode d'Euler explicite. Cela permet de ne pas avoir à résoudre un système linéaire au prix d'un choix particulier du pas de temps pour

respecter une condition de stabilité. L'analyse de stabilité donne la condition CFL suivante (Loukili & Soulaïmani (2007)),

$$CFL = \Delta t \frac{max(\sqrt{gh} + \sqrt{u^2 + v^2})}{min(d_{L,LR})} \tag{2.48}$$

avec $d_{L,LR}$ est la distance entre le centre de la cellule et le centre de l'interface L et R. Cependant pour plus de simplicité on choisira ici de prendre $d_{L_LR} = R_L$ le rayon du cercle inscrit dans la cellule L. C'est une condition très conservative qui a montré une très grande stabilité pour $CFL = 0.9$. Avec la discrétisation temporelle l'équation 2.46 devient,

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = -\frac{1}{|\Omega_i|} \sum_{j=1}^{3} L_{ij} T_{n_{ij}}^{-1} \tilde{G}(U_L^n, U_R^n) + |\Omega_i| S_{O_i}^n(U) + |\Omega_i| S_{f_i}^n(U) \tag{2.49}$$

où l'exposant n désigne que la valeur est prise au temps t_n.

2.2.2 Schéma HLL et HLLC

On rappelle ici les flux du schéma HLL (Harten, Lax & Leer, 1983 ; Loukili & Soulaïmani, 2007 ; Toro, 2001),

$$\tilde{G}^{HLL} = \begin{cases} G(U_L), S_L \geq 0, \\ G(U_R), S_R \leq 0, \\ \tilde{G}(U_*), S_L \leq 0 \text{ et } S_R \geq 0. \end{cases} \tag{2.50}$$

avec $\tilde{G}(U_*)$ le flux dans la région étoile donné par

$$\tilde{G}(U_*) = (S_R G(U_L) - S_L G(U_R) + S_R S_L (U_R - U_L))/(S_R - S_L) \tag{2.51}$$

avec S_R et S_L les vitesses des ondes droites et gauches estimées de la façon suivante,

$$S_L = u_L - a_L p_L, S_R = u_R + a_R p_R \tag{2.52}$$

ou pour $k = L, R$, $a_k = \sqrt{g h_k}$ et

$$p_k = \begin{cases} [h^*(h^* + h_k)/2)]^{1/2}/2, & h^* > h_k \\ 1, & h^* \leq h_k \end{cases} \tag{2.53}$$

avec h^* la hauteur d'eau dans la région étoile évaluée de la façon suivante. Une première approximation permet de savoir si on a une onde de choc ou de raréfaction,

$$h_0^* = \frac{h_L + h_R}{2} - \frac{(U_R - U_L)(h_L + h_R)}{4(a_R + a_L)} \tag{2.54}$$

Si $h_0^* \leq min(h_L, h_R)$ alors c'est une onde de raréfaction et on a,

$$h^* = [(a_R + a_L)/2 + (U_L - U_R)/4]^2/g \tag{2.55}$$

Si $h_0^* > min(h_L, h_R)$ alors c'est une onde de choc et on a,

$$\begin{aligned} h^* &= (h_L g_L + h_R g_R + u_L - u_R)/(g_L + g_R), \\ g_k &= \left[\frac{g(h_0^* + h_k)}{2h_0^* h_k}\right]^{1/2}, k = L, R \end{aligned} \tag{2.56}$$

On peut ensuite modifier ce schéma pour avoir le schéma HLLC (Ata *et al.*, 2013 ; Harten, 1983 ; Loukili & Soulaïmani, 2007 ; Toro, 2001) en prenant en compte la vitesse S^* dans la zone étoile.

La modification n'a d'impact que dans la dernière composante du flux,

$$\tilde{G}_3^{HLLC} = \begin{cases} \tilde{G}_1^{HLL}(U_L, U_R)v_L, S^* \leq 0 \\ \tilde{G}_1^{HLL}(U_L, U_R)v_R, S^* \geq 0 \end{cases} \tag{2.57}$$

avec,

$$S^* = \frac{s_L h_R(u_R - S_R) - s_R h_L(u_L - S_L)}{h_R(u_R - S_R) - h_L(u_L - S_L)} \tag{2.58}$$

en considérant les situations suivantes de lit sec,

$$\begin{aligned} h_L = 0 : S_L = U_R - 2a_R, \quad S_R = u_R + a_R, S^* = S_L, \\ h_L = 0 : S_L = U_L - a_L, \quad S_R = u_L + 2a_L, S^* = S_R. \end{aligned} \tag{2.59}$$

Avec ce flux le schéma 2.49 devient

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = -\frac{1}{|\Omega_i|} \sum_{j=1}^{3} L_{ij} T_{n_{ij}}^{-1} \tilde{G}^{HLLC}(U_L^n, U_R^n) + S_{O_i}^n(U) + S_{f_i}^n(U) \tag{2.60}$$

2.2.3 Traitement des termes sources

On présente dans cette partie le traitement des termes sources bathymétrie et de friction.

2.2.3.1 Bathymétrie

Pour le traitement des termes sources de bathymétrie on reprend la façon de faire de Ata *et al.* (2013) basée sur Audusse & Bristeau (2005). Il s'agit d'une méthode qui peut être utilisée avec n'importe quel flux numérique consistant (Ata *et al.* (2013)).

Les étapes principales sont les suivantes :

- Définir la bathymétrie à l'interface entre les cellules i et j avec $z_{ij} = z_{ji} = max(z_i, z_j)$

- Définir la hauteur d'eau à l'interface $h_{ij}^{**} = max(0, h_i + z_i - z_{ij})$ et ainsi obtenir des nouvelles variables à l'interface :

$$U_{ij}^{**} = (h_{ij}^{**}, h_{ij}^{**}u_i, h_{ij}^{**}v_i)^T \tag{2.61}$$

- Faire l'hypothèse, $\nabla s = 0$, soit, $\nabla(gh^2/2) = -gh\nabla b$. On peut alors développer le calcul du terme de friction. Par définition on avait dans 2.37,

$$S_O(U) = \begin{pmatrix} 0 \\ -gh\nabla b \end{pmatrix} \tag{2.62}$$

  Ce qui donne avec l'approximation $-gh\nabla b = \nabla(gh^2/2)$,

$$S_O(U) = \begin{pmatrix} 0 \\ \nabla(gh^2/2) \end{pmatrix} \tag{2.63}$$

  Ensuite en utilisant la définition 2.43 ($S_{O_i}(U) = \frac{1}{|\Omega_i|}\int_{\Omega_i} S_O(U)$), on choisit une nouvelle discrétisation en utilisant les variables 2.61 pour obtenir (Audusse, Bouchut, Bristeau, Klein & Perthame, 2004b),

$$S_{O_i}(U) = \frac{1}{|\Omega_i|}\sum_{j=1}^{3} L_{ij}\begin{pmatrix} 0 \\ g(h_{ij}^{**^2} - h_i^2)\mathbf{n_{ij}} \end{pmatrix} \tag{2.64}$$

- Finalement, on remplacera S_{O_i} dans 2.60 par 2.64.

On note ici que la relation $\nabla(gh^2/2) = -gh\nabla b$ est choisie pour vérifier un état stable où le gradient de la surface libre est nul, normalement, on devrait avoir,

$$\nabla(gh^2/2) = gh\nabla(h) = gh\nabla(s - b) \tag{2.65}$$

2.2.3.2 Friction

On rappelle que,

$$S_{f_i}(U) = \frac{1}{|\Omega_i|} \int_{\Omega_i} S_f(U) \partial\Omega \tag{2.66}$$

avec,

$$S_f(U) = (0, -gh\frac{n^2 \bar{u}\sqrt{\bar{u}^2 + \bar{v}^2}}{h^{4/3}}, -gh\frac{n^2 \bar{v}\sqrt{\bar{u}^2 + \bar{v}^2}}{h^{4/3}}). \tag{2.67}$$

Pour traiter le terme lié à la friction on reprend la semi-implicitation proposée dans Loukili & Soulaïmani (2007). Cette méthode consiste à prendre,

$$S_f = \frac{S_f^{n+1} + S_f^n}{2} \tag{2.68}$$

puis d'utiliser l'approximation,

$$S_f^{n+1} \simeq S_f^n + J_f(U^{n+1} - U^n) \tag{2.69}$$

avec

$$J_f^n = \frac{\partial S_f^n}{\partial U} = \begin{pmatrix} 0 & 0 & 0 \\ \partial S_{fx}^n/\partial h & \partial S_{fx}^n/\partial(hu) & \partial S_{fx}^n/\partial(hv) \\ \partial S_{fy}^n/\partial h & \partial S_{fy}^n/\partial(hu) & \partial S_{fy}^n/\partial(hv) \end{pmatrix} \tag{2.70}$$

On peut trouver le développement de 2.70 dans l'annexe II la forme finale étant A II-2.

Finalement, on fait l'approximation suivante pour 2.66,

$$S_{f_i}(U) = S_f(U_i) \tag{2.71}$$

On peut ensuite remplacer 2.68 dans 2.60, on obtient la forme finale de l'approximation,

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = \left[I - \frac{\Delta t}{2} J_f \right]^{-1} \left[-\frac{1}{|\Omega_i|} \sum_{j=1}^{3} L_{ij} T_{n_{ij}}^{-1} \tilde{G}^{HLLC}(U_L^n, U_R^n) + S_f(U_i^n) + S_{O_i}^n(U_i^n, U_{ij}^{**^n}, n_{ij}) \right] \tag{2.72}$$

CHAPITRE 3

LES BASES DE LA PROGRAMMATION GPU

L'architecture CUDA a été développée par NVIDIA pour permettre de faire des calculs généraux sur le GPU, la première version date de 2007. Comme on l'a présenté dans le chapitre 1, il y a plusieurs façons d'utiliser l'architecture CUDA. On utilise ici le CUDA Fortran, développé à la suite d'une collaboration entre PGI et NVIDIA en 2009.

Plusieurs livres ont été écrits sur le sujet, et on présente dans la suite les concepts les plus basiques. Pour une meilleure compréhension de la programmation GPU, le lecteur peut entre autre se référer aux trois livres sur lesquels nous basons notre travail, Sanders & Kandrot (2010), Fatica & Ruetsch (2014) et Kirk & mei W. Hwu (2017). On retrouve la plupart des différents concepts dans chaque livre approchés sous des formes différentes.

Le fil conducteur de la section à venir sera le passage sur GPU de l'extrait de code 3.1.

Extrait 3.1 Programme simple sur CPU

```
1  module fonctions
2    implicit none
3
4  contains
5    subroutine increment(a,b,n)
6      implicit none
7      integer, intent(in) :: n
8      integer, dimension(:), intent(inout):: a,b
9      integer :: i
10
11     do i=1,n
12       a(i) = a(i) + 2*b(i)
13     end do
14
15   end subroutine increment
16 end module fonctions
17
18 program main
19   use fonctions
```

```
20    implicit none
21
22    integer, parameter :: n=1e6
23
24    integer, dimension(:), allocatable :: a,b
25
26    allocate(a(n),b(n))
27
28    a=1
29    b=2
30
31    call  increment(a,b,n)
32
33    if(any(a/=5)) then
34      print*,"Program␣Failed"
35    else
36      print*,"Program␣Passed"
37    end if
38
39    deallocate(a,b)
40  end program main
```

Ce code fait le calcul $a = a + 2b$ grâce à la fonction *increment* présente dans le module *fonctions*. Dans la pratique on peut imaginer qu'on aura un calcul plus complexe à faire mais c'est une illustration et on montre dans la suite les étapes pour lancer cette fonction sur le GPU.

3.1 Code hôte et *kernels*

Lorsque l'on fait de la programmation GPU en utilisant du CUDA Fortran on va en fait seulement changer des fonctions du code pour les rendre exécutables sur le GPU. On appelle alors ces fonctions des *kernels*. Il faudra alors que le code hôte qui s'exécute toujours sur le CPU lance ces *kernels* sur le GPU. Ces *kernels* diffèrent alors des fonctions classiques par plusieurs aspects. Au minimum, les *kernels* :

- Sont lancés de manière asynchrone par le CPU sur le GPU ;
- Doivent être lancés avec une certaine configuration de *threads* et *blocks* ;
- Sont automatiquement assignés les variables *threadIdx*, *blockIdx* et *blockDim* ;

- Ne peuvent être appelés qu'avec des variables explicitement déclarées sur le GPU.

Si un *kernel* vise à être appelé par le code hôte il faudra le déclarer avec,

Extrait 3.2 Déclaration d'un *kernel* qui sera appelé depuis le CPU

```
attributes(global) subroutine kernel_from_cpu()
...
end subroutine kernel_from_cpu
```

alors que si on veut pouvoir l'appeler depuis un autre *kernel* il faudra le déclarer de la façon suivante,

Extrait 3.3 Déclaration d'un *kernel* qui sera appelé depuis le GPU

```
attributes(device) subroutine kernel_from_gpu()
...
end subroutine kernel_from_gpu
```

3.2 Configuration de lancement

Pour chaque *kernel* on doit spécifier une configuration de *threads* et *blocks* qui va dicter le lancement de ce même *kernel* en parallèle sur le GPU. Les *threads* sont des fils d'exécution qui seront organisés en un certain nombre de *blocks*. Concrètement si on lance un *block* de *n threads* le *kernel* sera lancé *n* fois sur le GPU.

Ce qui nous permet de modifier l'exécution du *kernel*, en fonction de sa position dans la configuration *threads-blocks*, sont les variables *threadIdx*, *blockIdx* et *blockDim* qui sont accessibles dans tous les *kernels*. Concrètement pour avoir l'identifiant *gi* d'un *kernel* il nous suffit de faire,

$$\begin{aligned} ti &= threadIdx \\ gi &= (blockIdx - 1) * blockDim + ti \end{aligned} \tag{3.1}$$

En notant qu'en Fortran *threadIdx* et *blockIdx* commencent à 1.

Une subtilité supplémentaire vient du fait que les variables *threadId*, *blockIdx* et *blockDim* sont des variables de type *dim3*. Cela veut dire que la configuration des *blocks* et *threads* peut être faite avec 3 dimensions. Il est en effet courant de chercher à avoir un *thread* par élément d'une matrice par exemple, il serait difficile avec une seule dimension d'attribuer un *thread* à chaque élément de la matrice. On peut donc utiliser 2 dimensions pour, par exemple, lancer un *kernel* avec $(1, 1, 1)$ *blocks* et $(n, n, 1)$ *threads*. De cette façon on lance un *block* avec n^2 *threads* par *blocks* numérotés sur un grille de taille $n * n$. On pourra ensuite facilement avoir le numéro de chaque *thread* de la façon suivante,

$$\begin{aligned} ti &= threadIdx\%x \\ gi &= (blockIdx\%x - 1) * blockDim\%x + ti \\ tj &= threadIdx\%y \\ gj &= (blockIdx\%x - 1) * blockDim\%x + ti \end{aligned} \tag{3.2}$$

Où *threadIdx%x* permet d'accéder au numéro du *thread* selon la première dimension de la grille, *threadIdx%y* et *threadIdx%z* pour les dimensions 2 et 3.

On pourra de cette façon accéder à tous les éléments d'une matrice carrée de taille n, chaque élément aura les coordonnées (gi, gj).

Pour la simplicité de l'explication on choisit ici de ne faire varier que le nombre de *threads* par *blocks* mais en pratique il faudra choisir ce rapport avec précaution. En effet les *threads* d'un *block* auront la possibilité de se synchroniser et auront accès à la même mémoire partagée. Le nombre de *threads* est en pratique toujours pris comme une puissance de 2 pour accommoder l'attribution de mémoire partagée et faciliter les réductions dont on parlera dans la suite. En pratique si on veut avoir un *thread* par élément d'un vecteur on va plutôt sélectionner un nombre de *threads* par *blocks* par exemple, 256, puis sélectionner le bon nombre de *blocks* pour couvrir tout le vecteur.

Si on veut une configuration qui nous permette d'associer à chaque *thread* un élément des vecteurs *a* et *b* de l'extrait 3.1 on peut faire,

Extrait 3.4 Configuration de lancement pour un vecteur de taille *n*

```
use cudafor !! Pour utiliser le type dim3
type(dim3) :: threads, blocks

threads = dim3(256, 1, 1)
blocks = dim3(ceiling(n/real(threads%x)), 1, 1)

call increment<<<threads, blocks>>>(a, b)
```

Avec ce choix de configuration il y aura sûrement des *threads* dans le dernier *block* dont l'indice *gi* sera plus grand que la taille du vecteur *n*. Ce n'est pas un problème, il suffira à l'intérieur du *kernel* de vérifier que *gi* <= *n* avant de faire la copie.

Si on tente d'utiliser l'extrait 3.4 dans 3.1 on va avoir une erreur car les variables *a* et *b* que l'on passe à *increment* sont définies sur CPU et non sur GPU. C'est ce dont on parle dans la suite.

3.3 Copies mémoires CPU-GPU : mémoire *pageable*, *pinned* et *constant*

Les mémoires du CPU et du GPU sont complètement séparées et les *kernels* n'ont accès qu'aux variables déclarées sur le GPU. Il faudra donc déclarer spécifiquement des variables sur le GPU en début de programme puis il faudra copier les variables du CPU vers leurs équivalents sur le GPU avant de lancer un *kernel* puis à la fin de son exécution on copiera le résultat depuis de GPU vers le CPU. En pratique comme dans les exemples qui vont suivre si on a une variable qui a un certain *nom* sur le CPU alors on déclarera son équivalent sur le GPU avec *nom_d*. Ce n'est pas obligatoire mais c'est une convention qui est très souvent respectée.

Pour déclarer une variable sur le GPU il suffit de la déclarer avec l'attribut *device* depuis le code hôte, par exemple

```
real, device, dimension(:), allocatable : a_d
```

De cette façon la variable est déclarée sur le GPU et pourra être modifiée par un *kernel*. Si on veut avoir des variables constantes sur le GPU modifiables uniquement par le CPU on peut aussi déclarer des variables de la façon suivante,

```
integer, constant :: n_d
```

L'attribut *constant* sert à déclarer une variable sur GPU qui ne sera modifiable que depuis le code hôte en copiant une nouvelle valeur dans cette variable. L'avantage est un accès plus rapide à ce type de variables sur le GPU comme ces variables constantes seront présentes dans la mémoire cache sur le GPU.

Une fois qu'on a déclaré ces variables sur le GPU on aura probablement besoin de copier les variables qu'on aura initialisées sur le CPU vers le GPU. Le prototype de la routine de copie est le suivant.

```
integer function cudaMemcpy(dst, src, count)
```

avec *dst* = *destination*, *src* = *source* et *count* = *nombre d'éléments*

La routine 3.3 et une copie qui va implicitement synchroniser le CPU et GPU, c'est-à-dire qu'avant de procéder à la copie, le CPU va attendre que tous les *kernels* aient fini leur exécution sur le GPU. On pourra éviter cette synchronisation en utilisant des *streams*, c'est ce dont on parle en section 3.6.

Ces copies du CPU vers le GPU sont à éviter au maximum comme elles ralentissent l'exécution. On cherchera donc à faire une unique copie du CPU vers le GPU en début de programme, une fois les différents vecteurs initialisés sur le CPU, et une seule copie en retour du GPU vers le CPU en fin de programme une fois tous les *kernels* exécutés.

Si on a régulièrement besoin de faire des copies du CPU vers le GPU et inversement il est intéressant de déclarer les variables sur le CPU avec l'attribut *pinned*. En effet, par défaut les variables sur le CPU sont définies comme *pageable* or les transferts mémoires entre le GPU et le CPU ne peuvent se faire que si la mémoire sur le CPU est définie comme *pinned*. Ce qui se passe alors en arrière-plan lorsque les variables ne sont pas explicitement définies comme *pinned* est qu'une copie est faite depuis la mémoire *pageable* vers la mémoire *pinned* sur le CPU puis que la copie est faite depuis la mémoire *pinned* sur le CPU vers la mémoire du GPU, la figure 3.1 en est une illustration.

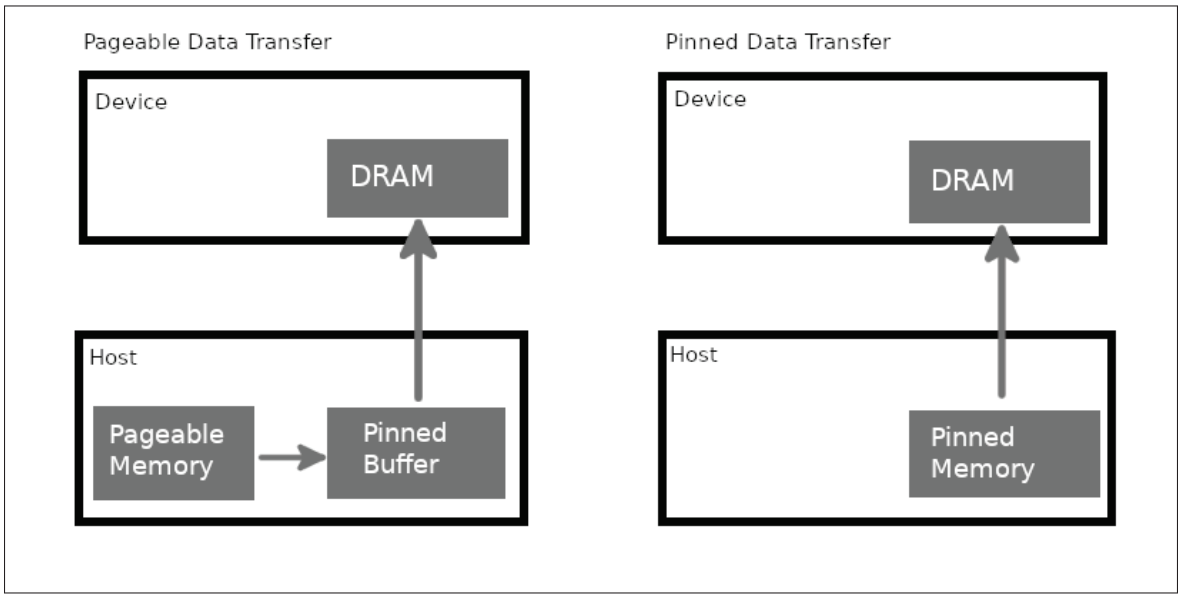


Figure 3.1 Illustration de l'échange de mémoire *pinned* et *pageable* sur le CPU, inspiré de (Fatica & Ruetsch, 2014, p. 45)

Cette copie supplémentaire ralentit encore l'échange. Les variables qui sont souvent copiées doivent donc être définies explicitement comme *pinned* sur le CPU. Pour une explication plus complète le lecteur peut se tourner vers (Fatica & Ruetsch, 2014, p. 45).

3.4 Première version d'un programme sur GPU

A ce stade, on a les éléments qui nous permettent de faire une première version de 3.1 qu'on présente dans l'extrait 3.5.

Extrait 3.5 Programme simple de copie sur GPU

```
1  module kernels
2    implicit none
3  contains
4    attributes(global) subroutine increment(a,b,n)
5      implicit none
6      integer, intent(in) :: n
7      integer, dimension(:), intent(inout):: a,b
8
9      integer :: ti,gi
10
11     ti = threadidx%x
12     gi = (blockIdx%x-1)*blockdim%x + ti
13
14     if(gi<=n) then
15       a(gi) = a(gi) + 2*b(gi)
16     end if
17
18   end subroutine increment
19 end module kernels
20
21 program main
22   use cudafor
23   use kernels
24   implicit none
25
26   integer, parameter :: n=1e6
27   integer, device :: n_d
28   integer :: ierr
29
30   integer, pinned, dimension(:), allocatable :: a,b
31   integer, device, dimension(:), allocatable :: a_d,b_d
32
33   type(dim3) :: threads,blocks
34   threads=dim3(256,1,1)
35   blocks=dim3(ceiling(n/real(threads%x)),1,1)
36
```

```
37    allocate(a(n),b(n))
38    allocate(a_d(n),b_d(n))
39
40    a=1
41    b=2
42
43    ierr = cudaMemcpy(n_d,n,1)
44    ierr = cudaMemcpy(a_d,a,n)
45    ierr = cudaMemcpy(b_d,b,n)
46
47    call  increment<<<blocks, threads>>>(a_d,b_d,n_d)
48
49    ierr = cudaMemcpy(a,a_d,n)
50    ierr = cudaMemcpy(b,b_d,n)
51
52    if(any(a/=5)) then
53      print*,"Program␣Failed"
54    else
55      print*,"Program␣Passed"
56    end if
57
58    deallocate(a_d,b_d)
59    deallocate(a,b)
60  end program main
```

On retrouve sur les lignes,

- 4 la déclaration du *kernel* pour être appelé depuis le code hôte,
- 11-12 le calcul de l'indice de chaque *thread*,
- 14 la condition pour éviter un problème pour les *threads* en trop,
- 30 la déclaration des vecteurs en mémoire *pinned* sur le CPU,
- 27 et 31 la déclaration des variables sur GPU avec l'attribut *device*,
- 33-35 la déclaration de la configuration de lancement pour les *kernels*,
- 43-45 la copie de a, b et n depuis le CPU vers a_d, b_d et n_d sur le GPU,
- 47 le lancement du *kernel increment* sur le GPU,
- 49-50 la copie de a_d et b_d depuis le GPU vers a et b sur le CPU.

On peut remarquer qu'à l'intérieur du *kernel* on n'a pas besoin de déclarer les variables avec l'attribut *device*. Comme ce code est exécuté sur le GPU les variables seront automatiquement déclarées sur le GPU.

Finalement, on remarque dans cet exemple qu'on n'a pas déclaré n_d comme de la mémoire *constant* sur le GPU alors que cette variable s'y prête. C'est intentionnel pour montrer un phénomène particulier qui se produit lors de l'exécution de ce programme et qu'on pourra facilement empêcher en déclarant des variables globales sur le GPU. A ce moment là nous utiliserons l'attribut *constant* pour les variables qui s'y prêtent. C'est ce dont on parle dans la section suivante.

3.5 *Profiler* et échanges mémoires implicites

Lorsque l'on exécute un programme il est courant d'avoir recours à un *profiler* pour avoir des informations sur l'exécution du code. Dans le cadre de la programmation GPU en CUDA Fortran le *profiler* que nous allons utiliser est *pgprof*, développé par PGI, qui est un dérivé de *nvprof*, le *profiler* pour le CUDA C développé par NVIDIA. Ce genre de *profiler* nous permet d'avoir un retour visuel sur l'exécution du code en plus de nous permettre de connaître le temps d'exécution des différents *kernels* et la durée des échanges mémoires. On va souvent y avoir recours pour expliquer différentes optimisations possibles. On présente ici l'utilisation du *profiler* sur l'extrait de code 3.5 que nous avons vu dans la partie précédante.

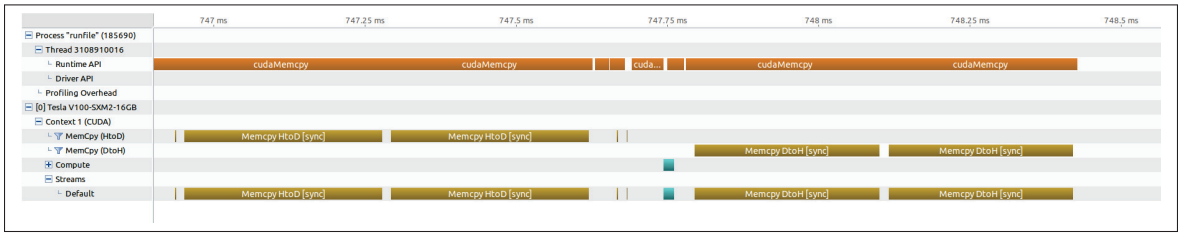


Figure 3.2 *Profiler* pour l'extrait de code 3.5

On peut voir sur la figure 3.2 les différentes étapes de 3.1. Les lignes qui nous intéressent ici sont les trois lignes dans la partie *Context* :

- *Memcpy (HToD)* : Copies mémoire Hôte vers Device donc CPU vers GPU,
- *Memcpy (DToH)* : Copies mémoire Device vers Hôte donc GPU vers CPU,
- *Compute* : Exécution des *kernels* sur le GPU.

On peut donc voir sur la figure 3.2, 5 copies mémoire CPU vers GPU avant le lancement du *kernel* puis 2 copies mémoires GPU vers CPU une fois le *kernel* terminé. On remarque directement que les copies mémoire sont bien plus longues que l'exécution du *kernel*, même en utilisant de la mémoire *pinned* c'est pour cela qu'il faut au maximum les éviter.

On peut facilement voir que les 2 copies une fois le *kernel* terminé correspondent aux lignes 49-50 de l'extrait 3.5. Ce sont des copies plutôt longues même si la mémoire du CPU est *pinned* car il faut copier tous les éléments du vecteur. On peut ensuite en déduire que les trois premières copies de 3.2 correspondent aux lignes 43-45 de l'extrait 3.5 avec en premier la copie de l'entier *n* qui est très rapide puis la copie des deux vecteurs.

Il reste donc 2 copies mémoire juste avant le lancement du *kernel* que nous n'avons pas expliquées. Il s'agit de copies implicites que nous n'avons pas demandées dans l'extrait 3.5. Si on pouvait aller voir de plus près dans le *profiler* on remarquerait que ces copies sont précédées par une allocation mémoire sur le GPU. Ces deux copies ne paraissent pas ralentir tant que ça l'exécution du code mais comme elles sont faites implicitement on n'a aucun contrôle dessus et elles sont sur le *stream* par défaut ce qui les rend bloquantes, on y viendra dans la suite.

Ces 2 copies sont en fait dues à la ligne 7 dans l'extrait 3.5 qu'on rappelle ici,

```
integer, dimension(:), intent(inout) :: a,b
```

Le fait de spécifier *dimension(:)* dans une subroutine est pratique comme on peut alors prendre des vecteurs de tailles diverses. Cependant cella provoque ici une allocation mémoire pour stocker la taille des vecteurs ainsi que leur copie sur le GPU de manière synchrone.

On a plusieurs choix ici pour régler le problème, on pourrait simplement remplacer la ligne 7 de l'extrait 3.5 par

```
integer, dimension(n), intent(inout) :: a,b
```

comme on donne déjà la taille des vecteurs n en argument de la routine.

On peut faire mieux que cela en définissant n_d comme une variable globale *constant* sur le GPU. De cette façon tous les *kernels* auront accès à cette variable. Dans notre code **CUTEFLOW** on gère cela avec un module spécial qui contient toutes ces variables globales *constant* sur le GPU. Cette modification donne une nouvelle version de l'extrait 3.5,

Extrait 3.6 Programme de copie sur GPU avec variables globales *constant*

```
 1 module global_constant
 2   implicit none
 3   integer, constant :: n_d
 4 end module global_constant
 5
 6 module kernels
 7   implicit none
 8 contains
 9   attributes(global) subroutine increment(a,b)
10     use global_constant
11     implicit none
12     integer, dimension(n_d), intent(inout):: a,b
13
14     integer :: ti,gi
15
16     ti = threadidx%x
17     gi = (blockIdx%x-1)*blockdim%x + ti
18
19     if(gi<=n_d) then
20       a(gi) = a(gi) + 2*b(gi)
21     end if
22
23   end subroutine increment
24 end module kernels
25
```

```
26 program main
27   use cudafor
28   use global_constant
29   use kernels
30   implicit none
31
32   integer, parameter :: n=1e6
33   integer :: ierr
34
35   integer, pinned, dimension(:), allocatable :: a,b
36   integer, device, dimension(:), allocatable :: a_d,b_d
37
38   type(dim3) :: threads,blocks
39   threads=dim3(256,1,1)
40   blocks=dim3(ceiling(n/real(threads%x)),1,1)
41
42   allocate(a(n),b(n))
43   allocate(a_d(n),b_d(n))
44
45   a=1
46   b=2
47
48   ierr = cudaMemcpy(n_d,n,1)
49   ierr = cudaMemcpy(a_d,a,n)
50   ierr = cudaMemcpy(b_d,b,n)
51
52   call  increment<<<blocks, threads>>>(a_d,b_d)
53
54   ierr = cudaMemcpy(a,a_d,n)
55   ierr = cudaMemcpy(b,b_d,n)
56
57   if(any(a/=5)) then
58     print*,"Program␣Failed"
59   else
60     print*,"Program␣Passed"
61   end if
62
63   deallocate(a_d,b_d)
64   deallocate(a,b)
65 end program main
```

Avec le résultat du *profiler* pour l'extrait 3.6 en figure 3.3.

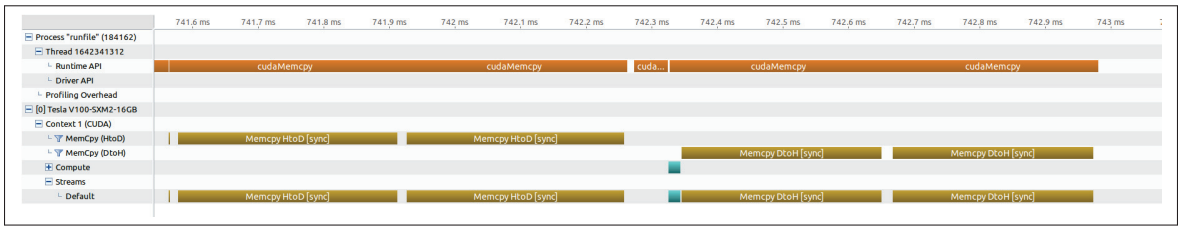


Figure 3.3 *Profiler* pour l'extrait de code 3.6

On constate bien que les copies implicites ne sont plus présentes et on a donc le contrôle sur toutes les copies. On peut donc par exemple faire des copies asynchrones sur des *streams* différents pour superposer ces copies avec l'exécution de *kernels*, c'est ce dont on parle dans la partie suivante.

3.6 Synchronisation Hôte-Device et superposition de *kernels* avec des *streams*

Comme on l'a dit, le lancement dans le code hôte des *kernels* se fait de manière asynchrone. Cela signifie que le code hôte n'attend pas la fin du *kernel* pour continuer l'exécution du programme. Comme on l'a vu, la copie d'éléments faite de manière classique provoque une synchronisation entre le CPU et le GPU. Si besoin on peut demander explicitement cette synchronisation avec la fonction *CudaDeviceSynchronize()* qui permet de bloquer le code hôte jusqu'à ce que toutes les opérations sur le GPU soient terminées.

Ce qui nous intéresse dans cette partie sont les *streams* qui sont en fait des files d'exécution sur le GPU. Le code hôte peut lancer des copies mémoire ou des *kernels* dans un certain *stream* en particulier. Il y a plusieurs choses importantes à noter :

- le *stream* par défaut bloque l'exécution des autres *streams*,
- l'exécution est séquentielle à l'intérieur d'un *stream*,
- les *streams* autres que le *stream* par défaut s'exécutent de manière asynchrone,
- on peut synchroniser le code hôte avec un *stream* avec *cudaStreamSynchronize(stream)*.

On va pouvoir créer des *streams* en faisant,

```
integer :: ierr
integer(kind=cuda_stream_kind) :: stream
ierr = cudaStreamCreate(stream)
```

Où le paramètre *ierr* permet juste de savoir si la création a bien fonctionné.

On pourra ensuite lancer un *kernel* ou une copie mémoire sur ce *stream*, on donne en exemple les lignes 48-55 de l'extrait de code 3.6.

Extrait 3.7 Utilisation de *streams* version 1

```
ierr = cudaMemcpyAsync(a,a_d,n,stream)
ierr = cudaMemcpyAsync(b,b_d,n,stream)

call increment<<<threads, blocks, 0, stream>>>(a_d,b_d)

ierr = cudaMemcpyAsync(a_d,a,n,stream)
ierr = cudaMemcpyAsync(b_d,b,n,stream)
```

On utilise ici la fonction *cudaMemcpyAsync* pour effectuer une copie asynchrone qui s'effectuera sur le *stream* voulu. Cependant dans l'extrait de code 3.7 on utilise un unique *stream*, et comme le lancement est séquentiel à l'intérieur d'un *stream* cela n'aura aucun impact sur le code.

On note au passage que lors de l'appel du *kernel increment* dans l'extrait 3.7 on spécifie la configuration de *threads*, *blocks* et le *stream* mais aussi un paramètre qu'on met à 0 ici, il s'agit de la quantité de mémoire partagée dont on parlera en section 3.7.

On pourrait alors se dire qu'il nous suffit de créer un second *stream* et de mettre la copie de *a* sur le *stream1* et la copie de *b* sur le *stream2* pour que les 2 copies se fassent en même temps sur le GPU. On aurait alors,

Extrait 3.8 Utilisation de *streams* version 2

```
ierr = cudaMemcpyAsync(a,a_d,n,stream1)
ierr = cudaMemcpyAsync(b,b_d,n,stream2)
```

```
ierr = cudaDeviceSynchronize()
call increment<<<threads, blocks, 0, stream1>>>(a_d,b_d)
ierr = cudaStreamSynchronize(stream1)

ierr = cudaMemcpyAsync(a_d,a,n,stream1)
ierr = cudaMemcpyAsync(b_d,b,n,stream2)
```

Le choix du *stream* pour le *kernel increment* n'a pas vraiment d'importance car on doit attendre la fin des copies avant de lancer le *kernel*. Juste avant le lancement du *kernel* on va donc utiliser la fonction *cudaDeviceSynchronize()* pour attendre la fin de la copie sur chaque *stream* avant de lancer le *kernel*. Une fois le *kernel* lancé il nous suffit d'attendre que son exécution sur le *stream1* soit terminée, on utilise donc *cudaStreamSynchronize(stream1)*. On aurait aussi pu utiliser à nouveau un *cudaDeviceSynchronize()*. On peut visualiser cette nouvelle version sur la figure 3.4

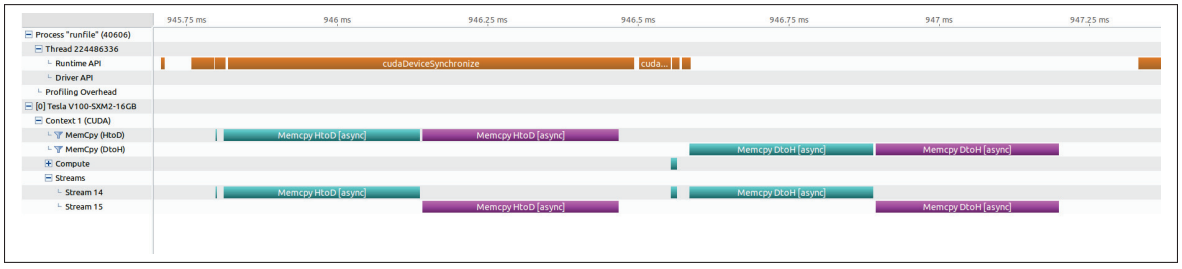


Figure 3.4 *Profiler* pour l'extrait de code 3.8

On remarque bien sur les deux dernières lignes qui concernent les *streams* que l'on a bien 2 *streams*, le *stream* 14 et le *stream* 15 qui correspondent aux *steam1* et *stream2* de l'extrait de code 3.8. Cependant on constate que les copies ne se superposent pas, que se passe-t-il ?

Il reste une dernière subtilité lorsque l'on parle de *streams* qui est que **les échanges mémoires dans la même direction ne peuvent pas se superposer** quoi que l'on fasse. On peut donc superposer des *kernels* avec un échange mémoire, ou un échange mémoire GPU vers CPU avec un échange CPU vers GPU mais on ne peut pas superposer plusieurs échanges GPU vers CPU. L'utilisation de *streams* comme faite dans l'extrait 3.7 ne sert donc pas à grand-chose. Cependant

on peut reprendre la même idée et l'appliquer au lancement de deux *kernels*, on présente cela dans l'extrait de code 3.9.

Extrait 3.9 Programme de copie avec superposition de deux *kernels*

```
1  module global_constant
2    implicit none
3    integer, constant :: n_d
4  end module global_constant
5
6  module kernels
7    implicit none
8  contains
9    attributes(global) subroutine increment(a,b)
10     use global_constant
11     implicit none
12     integer, dimension(n_d), intent(inout):: a,b
13
14     integer :: ti,gi
15
16     ti = threadidx%x
17     gi = (blockIdx%x-1)*blockdim%x + ti
18
19     if(gi<=n_d) then
20       a(gi) = a(gi) + 2*b(gi)
21     end if
22
23   end subroutine increment
24 end module kernels
25
26 program main
27   use cudafor
28   use global_constant
29   use kernels
30   implicit none
31
32   integer, parameter :: n=1e6
33   integer :: ierr
34
35   integer, pinned, dimension(:), allocatable :: a1,b1,a2,b2
36   integer, device, dimension(:), allocatable :: a1_d,b1_d
37   integer, device, dimension(:), allocatable :: a2_d,b2_d
38
39   integer(kind=cuda_stream_kind) :: stream1,stream2
```

```
40
41    type(dim3) :: threads,blocks
42    threads=dim3(256,1,1)
43    blocks=dim3(ceiling(n/real(threads%x)),1,1)
44
45    ierr = cudaStreamCreate(stream1)
46    ierr = cudaStreamCreate(stream2)
47
48    allocate(a1(n),b1(n),a2(n),b2(n))
49    allocate(a1_d(n),b1_d(n),a2_d(n),b2_d(n))
50
51    a1=1
52    b1=2
53    a2=1
54    b2=4
55
56    ierr = cudaMemcpyAsync(n_d,n,1,stream1)
57    ierr = cudaDeviceSynchronize()
58
59    ierr = cudaMemcpyAsync(a1_d,a1,n,stream1)
60    ierr = cudaMemcpyAsync(b1_d,b1,n,stream1)
61
62    call increment<<<blocks, threads, 0, stream1>>>(a1_d,b1_d)
63
64    ierr = cudaMemcpyAsync(a2_d,a2,n,stream2)
65    ierr = cudaMemcpyAsync(b2_d,b2,n,stream2)
66
67    call increment<<<blocks, threads, 0, stream2>>>(a2_d,b2_d)
68
69    ierr = cudaMemcpyAsync(a1,a1_d,n,stream1)
70    ierr = cudaMemcpyAsync(b1,b1_d,n,stream1)
71
72    ierr = cudaMemcpyAsync(a2,a2_d,n,stream2)
73    ierr = cudaMemcpyAsync(b2,b2_d,n,stream2)
74
75    ierr = cudaDeviceSynchronize()
76
77    if(any(a1/=5) .or. any(a2/=9)) then
78      print*,"Program␣Failed"
79    else
80      print*,"Program␣Passed"
81    end if
82
```

```
83    deallocate(a1,b1,a2,b2)
84    deallocate(a1_d,b1_d,a2_d,b2_d)
85  end program main
```

L'extrait de code 3.9 contient la déclaration de variables pour le lancement de deux *kernels* de copies sur les variables $a1$ et $b1$ pour le premier *kernel* et $a2\_d$ et $b2\_d$ avec le second. On remarque que comme l'exécution à l'intérieur d'un *stream* est séquentielle, on a juste besoin d'une synchronisation en ligne 74 à la fin du programme. Il donne avec le profiler la figure 3.5

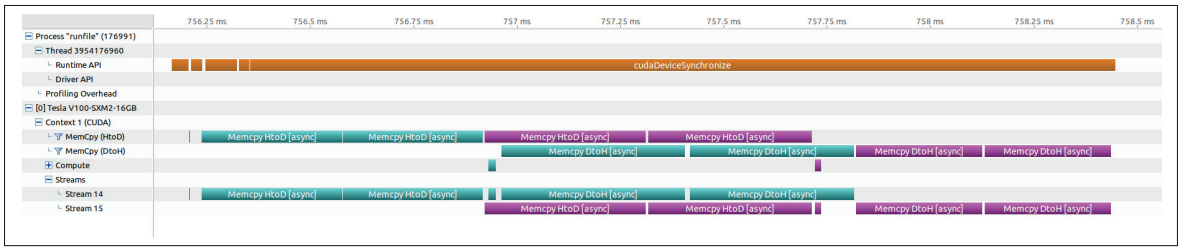


Figure 3.5 *Profiler* pour l'extrait de code 3.9

On obtient enfin le résultat souhaité. Pendant que le *kernel* 1 s'exécute la copie des variables pour les *kernel* 2 est en train de se faire, puis lors de l'échange GPU vers CPU du *kernel* 1 l'échange CPU vers GPU est en train de se faire pour le *kernel* 2. Ce genre de superposition est très puissant. C'est là où on apprécie le fait de ne plus avoir les échanges mémoire implicites causés par la ligne $dimension(:)$ dans les *kernels*. Ces copies auraient complètement empêché la superposition des échanges mémoire que l'on voit ici en étant automatiquement attribués sur le *stream* défaut qui est bloquant.

Pour finir cette partie, il faut préciser que l'utilisation de *streams* ne va pas magiquement créer des ressources supplémentaires sur le GPU. Ce qu'on veut dire est que si on avait essayé de superposer uniquement les *kernels* 1 et 2 l'un avec l'autre, en faisant les copies mémoires avant de synchroniser puis lancer chaque *kernel* sur son *stream*, on aurait vu une très faible superposition des *kernels*, comme le montre la figure 3.6 où on fait un zoom sur les deux *kernels*.

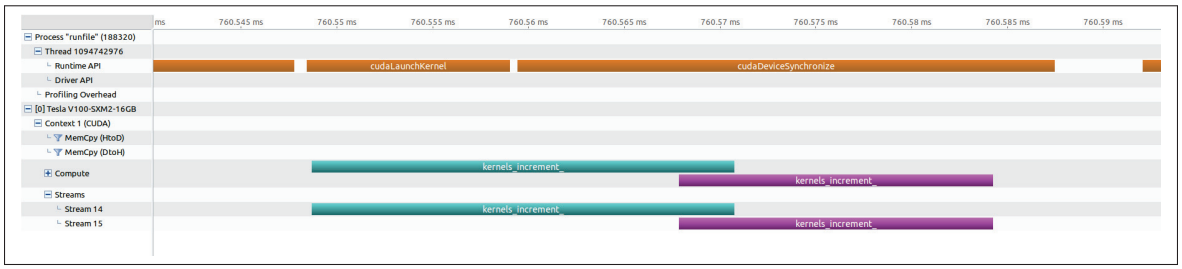


Figure 3.6 *Profiler* pour l'overlapping des *kernels* uniquement

Cela illustre le fait que souvent un *kernel* utilise au maximum les capacités du GPU. Il ne sera alors pas possible de superposer l'exécution de plusieurs *kernels* parce que tous les *threads* sur le GPU seront déjà utilisés. L'utilisation principale des *streams* réside donc dans la superposition des *kernels* avec un échange mémoire entre le CPU et le GPU.

3.7 Mémoire partagée et réductions sur le GPU : l'exemple du minimum

On présente dans cette partie comment faire une réduction pour trouver le minimum d'un vecteur sur GPU. On base ce travail sur la présentation de Harris (2007) dans laquelle on peut trouver différentes manières de faire des réductions sur le GPU.

La recherche du minimum dans un vecteur est une opération dont nous avons besoin dans notre code **CUTEFLOW** pour rechercher le pas de temps minimum sur chaque maille du domaine. Sur CPU il n'y a aucun problème car au fur et à mesure que l'on calcule les valeurs du pas de temps dans chaque maille on peut tenir à jour le pas de temps minimum. Sur GPU en revanche comme les calculs des pas de temps dans chaque maille se font *en même temps* et qu'un *thread* ne traite qu'une seule valeur on ne peut pas procéder de la même manière.

Une des premières façons de faire est, après avoir calculé les valeurs du vecteur de pas de temps sur le GPU, de copier le vecteur sur le CPU puis de rechercher le minimum sur le CPU et par la suite de re-copier cette valeur sur le GPU. C'était d'ailleurs fait de cette manière dans une des premières versions de **CUTEFLOW**. Cependant, comme on l'a montré dans les parties

précédantes, même en utilisant de la mémoire *pinned* sur le CPU les échanges mémoire vont être très lents. On va donc implémenter la recherche du minimum sur le GPU.

Faire une réduction sur le GPU n'est pas la chose la plus facile comme on ne peut pas synchroniser tous les *threads* d'un coup. La seule synchronisation possible est entre les *threads* d'un même *block*. On va donc d'abord rechercher le minimum à l'intérieur d'un *block* puis on utilisera des opérations *Atomic* pour que chaque *block* puisse comparer sa valeur minimum avec la valeur du minimum dans la mémoire globale du GPU sans interférence avec les autres *blocks*.

La recherche du minimum à l'intérieur d'un *block* doit se faire en utilisant de la mémoire partagée, déclarée avec l'attribut *shared* à l'intérieur des *kernels*. Tous les *threads* d'un même *block* auront accès à cette mémoire rapide et on pourra procéder à la réduction.

Plusieurs méthodes sont proposées dans Harris (2007), on présente ici celle qu'il appelle *Parallel Reduction : Sequential addressing* ce n'est pas la plus rapide mais comme on le verra par la suite ce ne sera pas le facteur limitant la vitesse de la réduction.

La première étape consiste à copier la partie du vecteur que le *block* va traiter en mémoire partagée. Ensuite la méthode consiste à n'utiliser que la première moitié des *threads* et de leur faire comparer la valeur en mémoire partagée qui leur correspond, d'indice ti, à la valeur d'indice $blockDim\%x/2 + ti$. De cette façon chaque *thread* compare deux valeurs du vecteur. On effectue ensuite la réduction voulue, dans notre cas, pour le minimum, si la valeur du vecteur en $ti + blockDim\%x/2$ est plus petite que la valeur en ti on copie cette valeur en ti. De cette manière toutes les valeurs minimales sont copiées dans la partie gauche du vecteur, on peut ensuite diviser le nombre de *threads* par deux et recommencer l'opération en comparant les valeurs entre ti et $ti + blockDim\%x/4$ jusqu'à ce que la valeur du minimum du vecteur soit dans la première case de la mémoire partagée. On reprend en figure 3.7 le processus de réduction pour trouver le minimum.

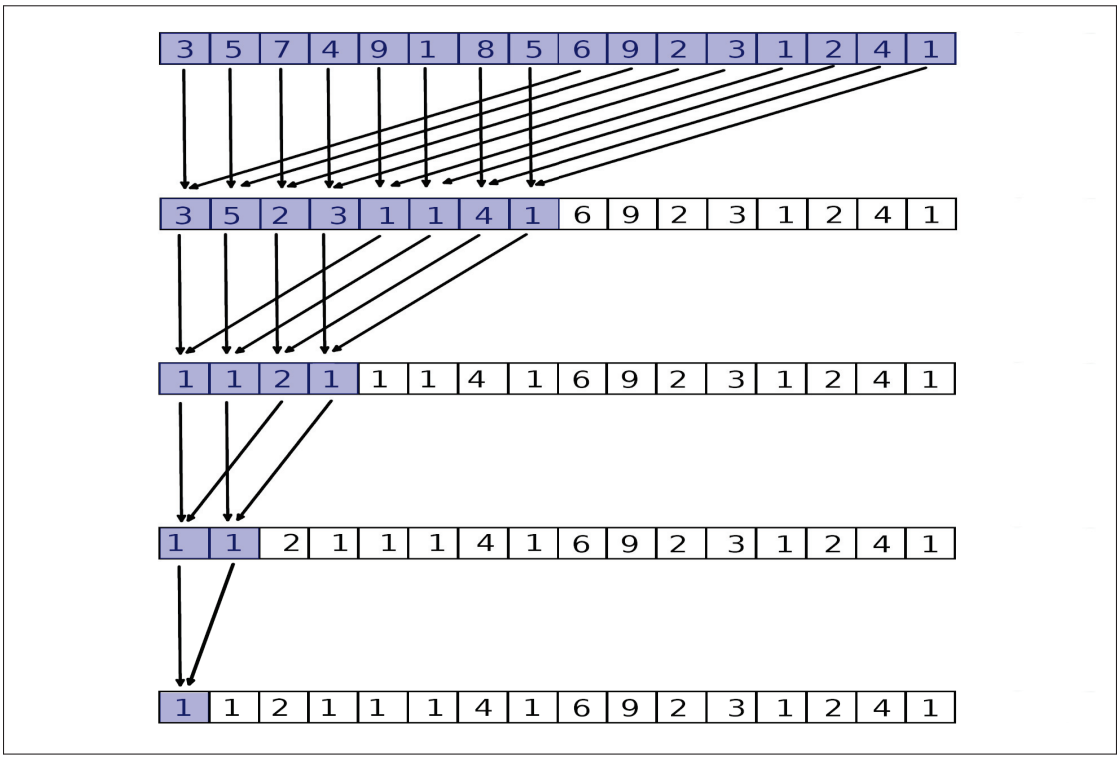


Figure 3.7 Processus de réduction pour trouver le minimum d'un vecteur sur GPU

À la fin de chaque étape il faut appeler $syncThreads()$ pour synchroniser les *threads* avant de passer à l'étape suivante. On note aussi qu'il faut que $blockDim$ soit une puissance de 2 si on veut pouvoir effectuer ce genre de réduction, comme on l'a dit plus haut, en pratique, on fera toujours ce choix.

Une fois la réduction effectuée à l'intérieur d'un *block* il faut que chacun écrive la valeur qu'il a trouvée dans la mémoire globale. Comme plusieurs *blocks* s'exécutent probablement sur le GPU en même temps, il faut faire attention à utiliser les opérations *Atomic* disponibles en CUDA Fortran qui nous permettent d'éviter que plusieurs *blocks* écrivent en même temps dans la mémoire globale. Au lieu de faire comme l'extrait 3.10 on fera comme dans 3.11 en utilisant l'opération *atomicmin* pour comparer la valeur du minimum global à la valeur minimum trouvée dans le *block* qui est dans la première case de la mémoire partagée.

Extrait 3.10 Ecriture du minimum dans la mémoire globale classique

```
if(min_global_d > shared(1)) then
    min_global_d = shared(1)
end if
```

Extrait 3.11 Ecriture du minimum dans la mémoire globale avec opération *Atomic*

```
ierr = atomicmin(min_global_d,shared(1))
```

On présente dans l'extrait 3.12 le *kernel* qui effectue la réduction qu'on vient de présenter,

Extrait 3.12 *kernel* de réduction pour trouver le minimum d'un vecteur sur le GPU

```
1  attributes(global) subroutine reduction_min(a)
2    use global_device
3    implicit none
4
5    real, dimension(n_d), intent(inout) :: deltatmin_d
6
7    integer :: i, ti, gi, ierr
8    real, shared       :: s(*)
9
10   ti = threadIdx%x
11   gi = (blockIdx%x - 1)*blockDim%x+ ti
12
13   if(gi<=n) then
14     s(ti) = deltatmin_d(gi)
15   else
16     s(ti) = 1000000.0
17   end if
18
19   call syncthreads()
20
21   i = blockdim%x/2
22   do while(i>=1)
23     if(ti <= i) then
24       s(ti) = min(s(ti),s(ti+i))
25     end if
26     i = i/2
27     call syncthreads()
28   end do
29
30   call syncthreads()
```

```
31    if(ti==1) then
32      ierr_d = atomicmin(dt_d,s(1))
33    end if
34    call syncthreads()
35  end subroutine reduction_min
```

On peut retrouver dans 3.12, aux lignes,

- 8, la déclaration dynamique de la mémoire partagée,
- 13-17, l'initialisation de la mémoire partagée,
- 22-28, la boucle qui divise par 2 le nombre de *threads* à chaque étape,
- 32, l'opération *atomicmin*.

On remarque qu'en ligne 8 on ne spécifie pas explicitement la taille du vecteur de mémoire partagée. Dans ce cas, cela ne crée pas de copie implicite. On peut spécifier la quantité de mémoire partagée dans la configuration de lancement du *kernel*. On a besoin d'avoir la place pour stoker un vecteur dont la taille correspond au nombre de *threads* dans un *block* et où chaque élément à la même taille que le vecteur sur lequel on veut faire la réduction. On procédera donc comme dans l'extrait 3.13 pour lancer la réduction avec la bonne quantité de mémoire partagée.

Extrait 3.13 Configuration du lancement de la réduction

```
size_shared = threads%x*sizeof(a_d(1))
call reduc_min<<<blocks, threads, size_shared, stream>>>(a_d)
```

Dans l'analyse de cette réduction, on a pu constater que c'est l'étape liée à l'opération *atomicmin* qui prend le plus de temps. La recherche à l'intérieur d'un *block* est bien plus rapide en comparaison. Pour pallier ce problème on peut augmenter le nombre de *threads* par *blocks* cela aura pour effet d'améliorer les performances. Dans ce cas on passera plus de temps dans la recherche du minimum à l'intérieur d'un *block*, étape qui est rapide, et le nombre réduit de *blocks* facilitera l'opération *atomicmin*.

C’est une réduction qu’on pourrait sûrement améliorer en implémentant les versions plus avancées présentées dans Harris (2007).

CHAPITRE 4

DÉCOMPOSITION DE DOMAINE

Dans ce chapitre on présente le pré-traitement mis en place pour effectuer la décomposition de domaine. L'objectif est, à partir du fichier de maillage d'un domaine, de générer autant de sous-fichiers de maillages que de sous-domaines voulus. On utilise donc une méthode assez classique décrite dans la figure 4.1, on présente dans la suite de ce chapitre les étapes importantes qui la constituent.

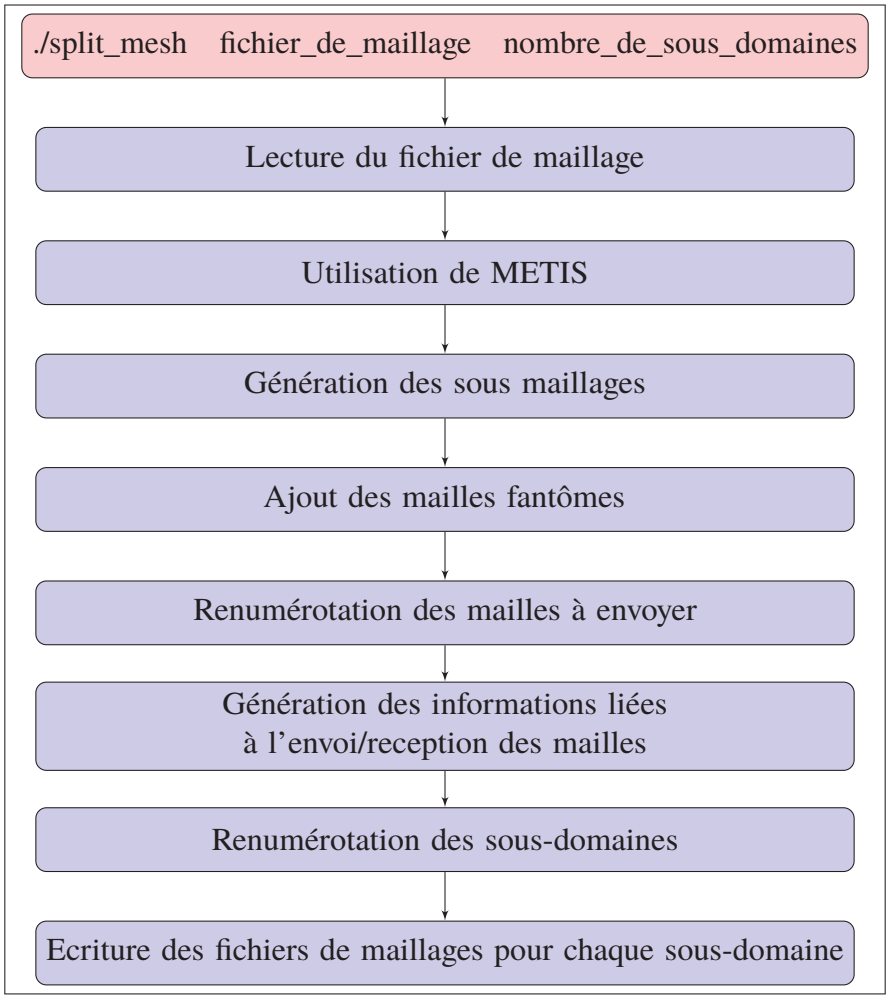


Figure 4.1 Routine split_mesh pour la décomposition de domaines

4.1 Utilisation de METIS

Pour effectuer la décomposition de domaine on utilise la bibliothèque METIS (Karypis & Kumar, 2009). C'est une bibliothèque très utilisée qui permet de décomposer un maillage en sous-domaines en utilisant le partitionnement du graphe du maillage.

C'est une étape très importante car il faut limiter au maximum les surfaces de contact entre les sous-domaines pour avoir dans la suite le minimum d'échanges mémoires entre les processeurs. METIS fait cela de manière très rapide et efficace cependant nous avons peu de contrôle sur la qualité de la décomposition. En effet on pourrait vouloir, en plus d'optimiser les surfaces de contact entre les sous-domaines, minimiser le nombre de sous-domaines qui se touchent, en particulier, éviter que des sous-domaines qui seront sur des nœuds de calcul différents se touchent. De cette façon on éviterait les échanges mémoires CPU vers GPU au maximum et les échanges se feraient au maximum en *Peer-to-Peer* (P2P) entre les GPU.

Comme nous avons peu de contrôle sur la décomposition en elle-même on va chercher à minimiser les échanges mémoire entre nœuds de calcul via une renumérotation des sous-domaines adaptée à nos problèmes.

4.2 Renumérotation des sous-domaines

On cherche ici à renuméroter les sous-domaines de la décomposition. On ne parle pas ici de la renumérotation des mailles dans chaque sous-domaine mais bien de la numérotation des sous-domaines eux-mêmes.

Sur la plupart des grappes de calcul il y a généralement 4 à 8 GPU par nœud de calcul. Si on envisage d'utiliser 4 GPU par nœud, cela implique que nous allons avoir 4 sous-domaines par nœud et donc il faudrait optimiser la renumérotation en conséquence. Cependant il n'y a souvent pas de solution miracle qui nous éviterait tous les échanges entre nœuds.

Cette renumérotation n'a pas eu d'impact significatif sur les performances du code : dans le meilleur des cas 8% de performances supplémentaires. On la présente tout de même ici comme c'est très simple à implémenter et que c'est une chose à laquelle il faut faire attention, particulièrement lorsque l'on veut traiter de nombreux sous-domaines.

Il semble que METIS traite déjà de manière interne la renumérotation des sous-domaines. Cependant, nous avons constaté que dans notre cas, certaines de ces numérotations pouvaient être facilement optimisées. On traite dans ce travail au maximum des décompositions en 32 sous-domaines, c'est donc ce qu'on va prendre comme exemple. En figure 4.2 le domaine de la rivière des Mille Îles décomposé en 32 sous-domaines avec la numérotation de METIS.

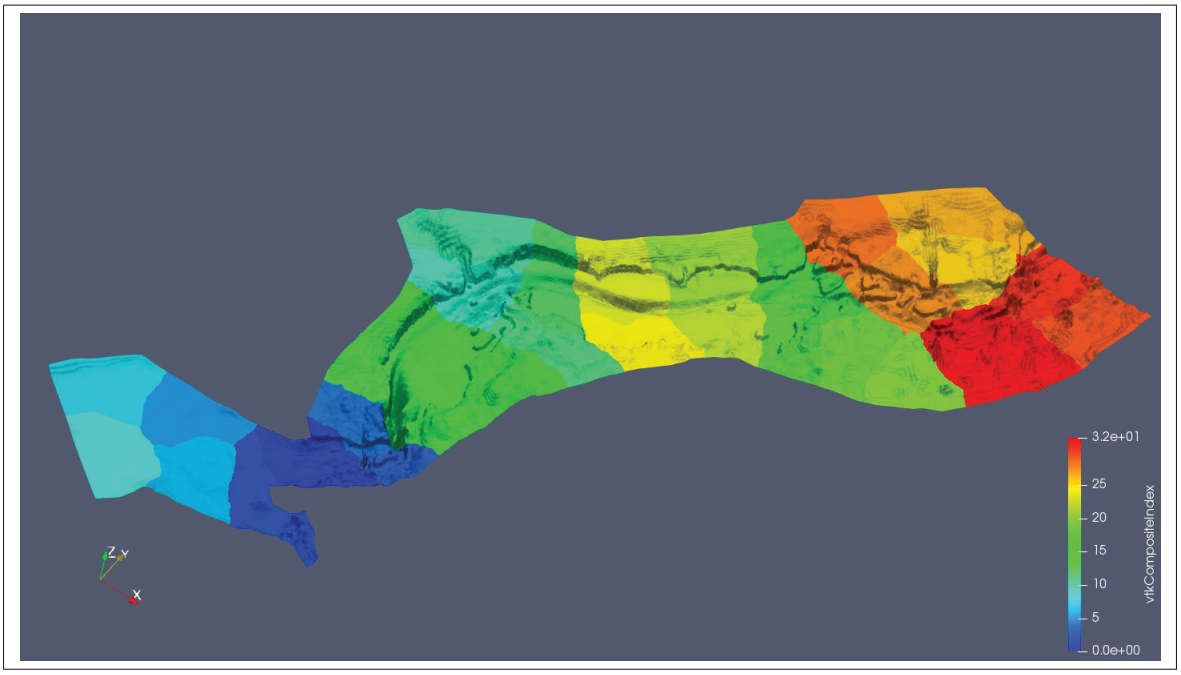


Figure 4.2 Domaine de la rivière des Mille Îles décomposé en 32 sous-domaines en utilisant la numérotation de METIS, chaque sous-domaine correspond à une couleur.

On constate sur la figure 4.2 que des sous-domaines verts touchent des sous-domaines rouges ce qui veut dire que ces sous-domaines vont devoir communiquer entre eux alors qu'il ne seront vraisemblablement pas sur le même nœud de calcul.

On propose d'améliorer cette numérotation en utilisant l'algorithme de Cuthill-McKee (Cuthill & McKee, 1969). Dans l'algorithme classique, on choisit de commencer la numérotation par un domaine de bord en recherchant le domaine avec le degré le moins important, c'est-a-dire le domaine ayant le moins de voisins. Cependant dans notre cas avec peu de sous-domaines, beaucoup n'ont que 2 voisins et cela n'assure pas forcément de commencer sur un domaine du bord. Pour résoudre simplement ce problème, on propose de commencer la numérotation par un domaine qui contient des nœuds d'entrée du domaine original. Ces nœuds sont donnés dans nos maillages et nous servent ensuite à définir les conditions aux limites, il est donc facile d'identifier les sous-domaines contenant des nœuds d'entrée. Une fois ce premier domaine choisi, on utilise l'algorithme de Cuthill-McKee classique. En exemple, sur la figure 4.3 le même domaine de la rivière des Mille Îles après la renumérotation,

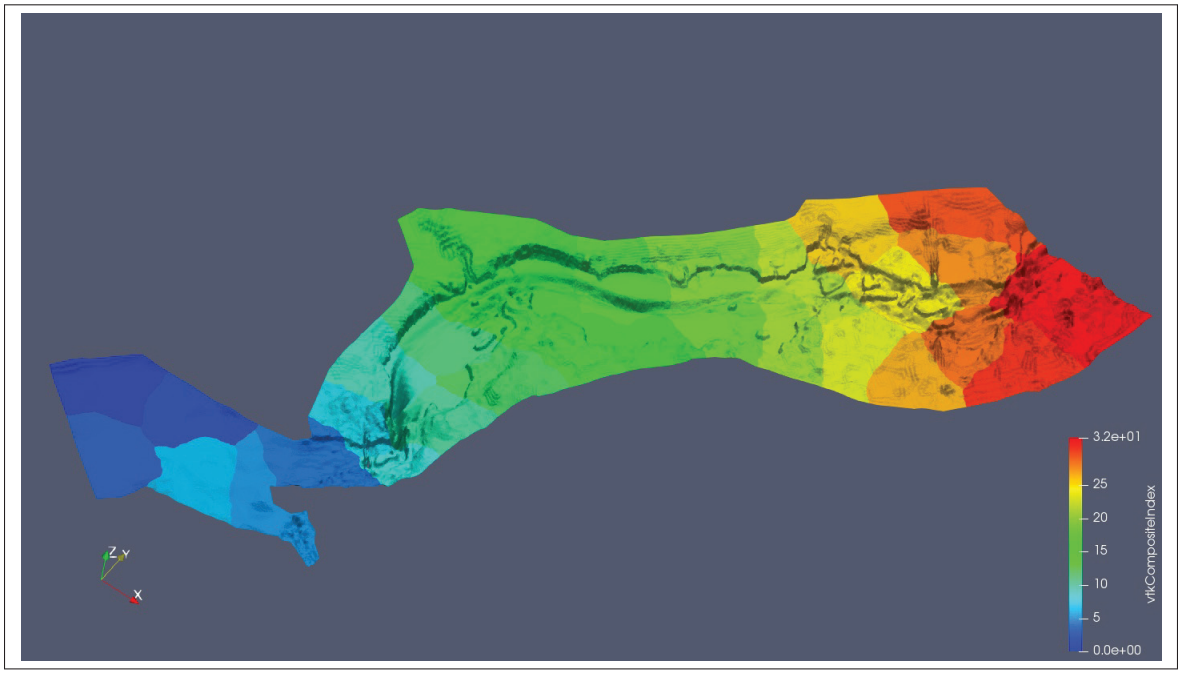


Figure 4.3 Domaine de la rivière des Mille Îles décomposé en 32 sous-domaines en utilisant notre renumérotation, chaque sous-domaine correspond à une couleur.

Comme on s'y attendait avec l'algorithme de Cuthill-McKee, les sous-domaines proches ont des numéros proches ce qui implique qu'ils seront probablement sur le même nœud de calcul.

Néanmoins, comme on l'a précisé, il y aura forcément des échanges entre des domaines qui seront sur des nœuds distincts.

Même si on constate une renumérotation plutôt satisfaisante sur le domaine des Mille Îles présenté précédemment, cette méthode atteint vite ses limites. Par exemple avec des domaines contenant plusieurs entrées comme le domaine de l'archipel de Montréal. On peut voir la numérotation de METIS sur la figure 4.4. De la même façon que précédemment on a des sous-domaines proches qui ont des numéros éloignés.

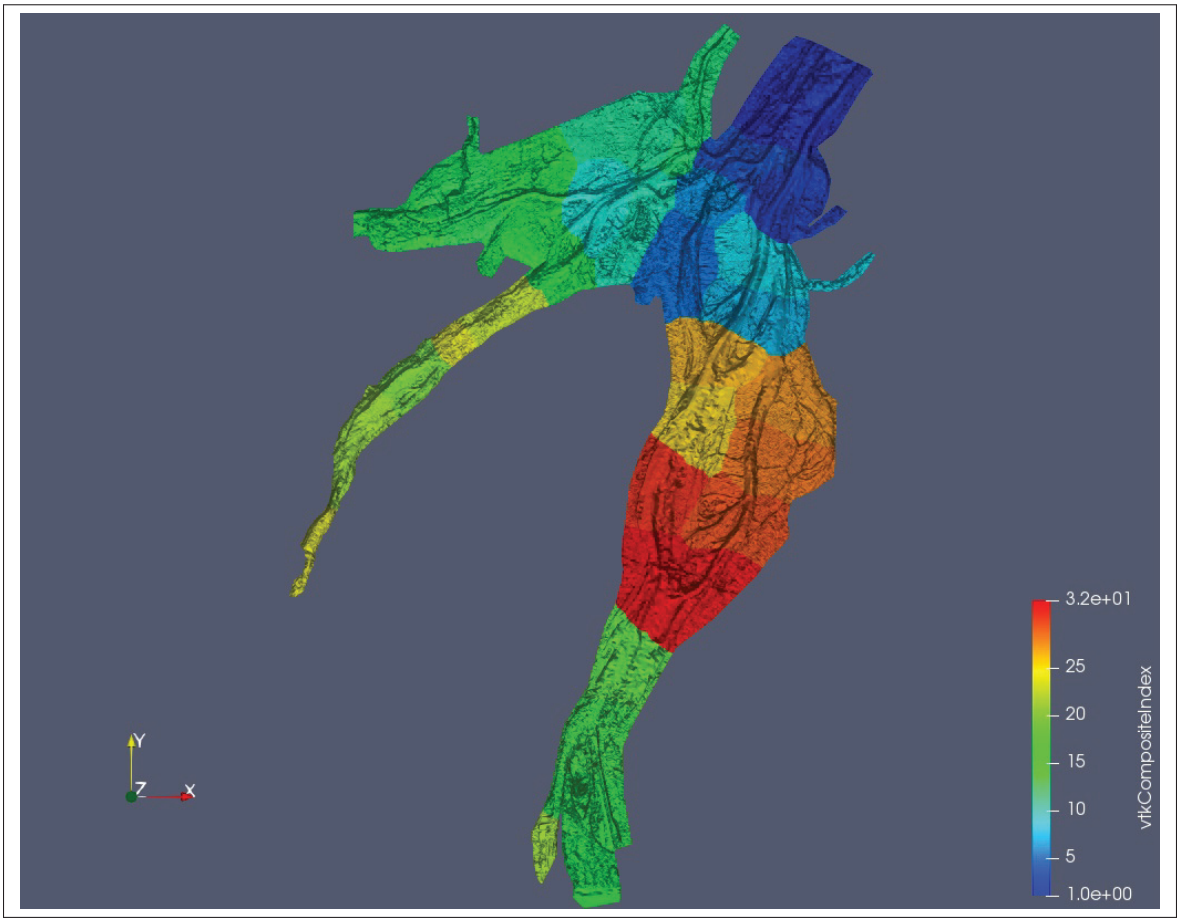


Figure 4.4 Domaine de l'archipel de Montréal décomposé en 32 sous-domaines en utilisant la numérotation de METIS, chaque sous-domaine correspond à une couleur.

Notre renumérotation donne la figure 4.5. Il est dans ce cas difficile de dire si notre numérotation est vraiment meilleure. Si on souhaite traiter de plus en plus de domaines contenant plusieurs

entrées et sorties et qu'on veut les décomposer en encore plus de sous-domaines il faudra alors proposer une meilleure méthode pour faire cette renumérotation. Pour l'utilisation qu'on en fait dans ce travail cette simple renumérotation est suffisante.

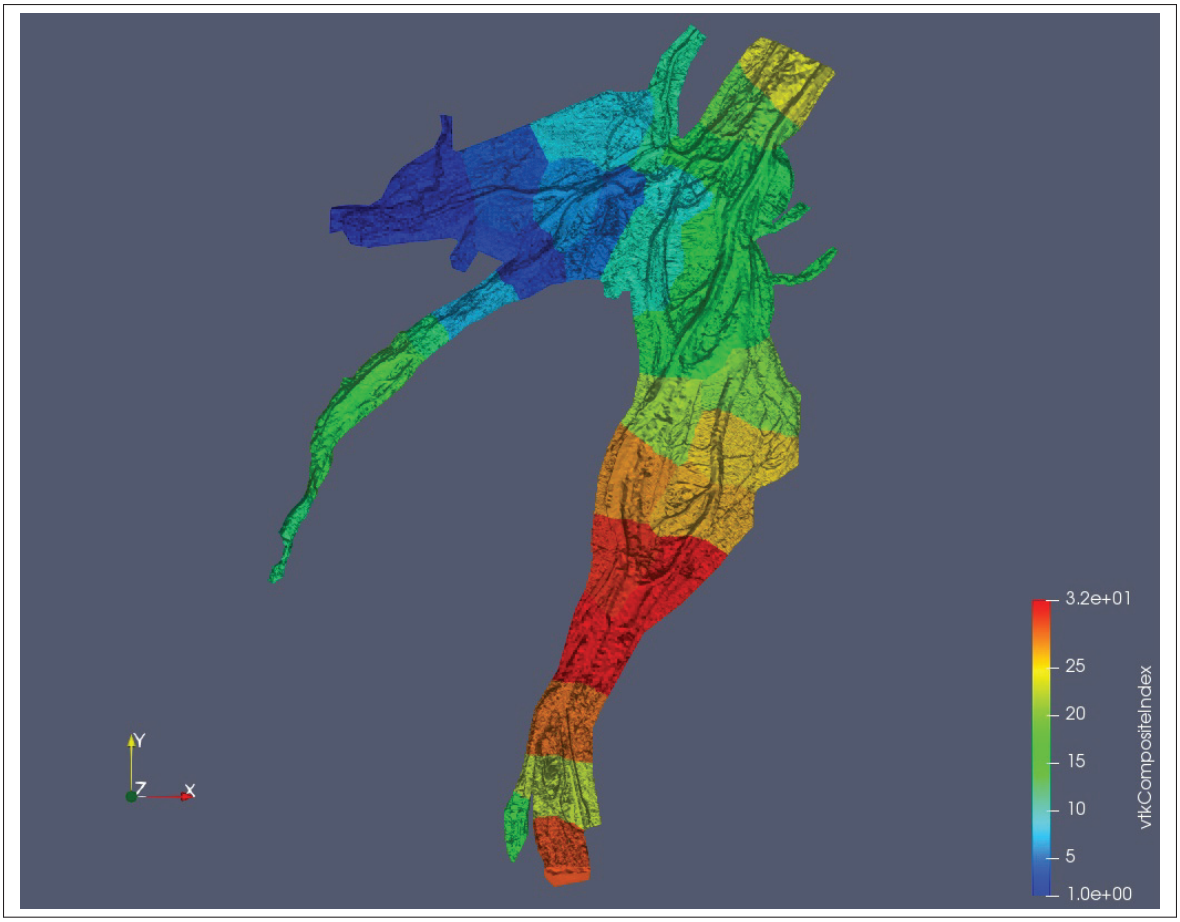


Figure 4.5 Domaine de l'archipel de Montréal décomposé en 32 sous-domaines en utilisant notre re-numérotation, chaque sous-domaine correspond à une couleur.

4.3 Ajout de mailles fantômes

A la suite de la décomposition du maillage initial, nous obtenons le nombre de sous-domaines voulus. Il va maintenant nous falloir traiter ces sous-domaines en anticipant les besoins qu'on aura lors de l'échange mémoire.

Nous utilisons dans nos simulations une méthode volumes finis comme présentée dans le chapitre 2. Le schéma numérique de cette méthode requiert de connaître les inconnues dans les mailles voisines, c'est-à-dire, qui ont une arête commune à la maille que l'on cherche à calculer. Cela implique un traitement particulier aux bords des sous-domaines.

En ce qui concerne les bords du domaine original, ils sont traités avec les conditions aux limites présentées au chapitre 2. Pour les bords des sous-domaines qui sont en contact avec d'autres sous-domaines il nous faut faire un traitement particulier. En effet, pour calculer la nouvelle valeur d'une maille sur un bord de ce type il nous faut aller chercher la valeur des mailles voisines qui se trouvent potentiellement dans un autre sous-domaine, attribué à un autre processeur. C'est à ce moment là qu'il va falloir utiliser la bibliothèque MPI pour effectuer un échange mémoire entre les processeurs.

Pour éviter d'avoir un échange mémoire à faire à chaque fois que l'on tombe sur une maille frontière une façon commune de faire est d'ajouter une couche de mailles dites fantômes à chaque sous-domaine. Ainsi on peut récupérer les valeurs de toutes ces mailles d'un seul coup puis ne plus se soucier d'échanges mémoire lors de l'exécution du schéma numérique que l'on utilisera alors pour calculer les nouvelles valeurs des mailles non-fantômes.

Pour trouver les mailles fantômes on va d'abord rechercher les nœuds communs entre deux sous-domaines adjacents. La figure 4.6 montre l'attribution des mailles faites par METIS, un domaine est coloré en bleu l'autre en blanc. On montre la ligne de séparation entre les domaines en rouge. Les nœuds communs aux deux sous-domaines sont sur cette ligne.

Une fois qu'on a ces nœuds communs on va rechercher dans chaque sous-domaine les mailles qui ont deux nœuds successifs appartenant à la liste des nœuds communs. Ces mailles seront des mailles fantômes pour l'autre sous-domaine. On donne en exemple la figure 4.7. Les mailles vertes sont les mailles fantômes à ajouter au domaine bleu. Elles seront donc à recevoir par ce domaine et à envoyer depuis le domaine blanc.

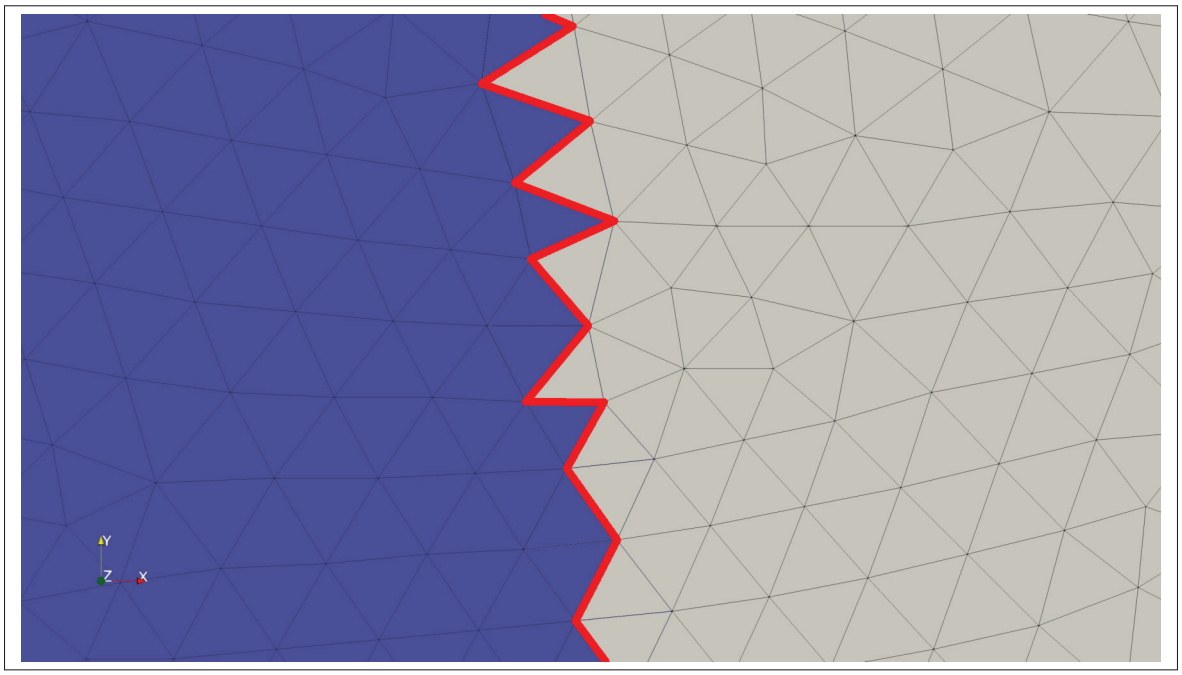


Figure 4.6 Attribution des mailles faites par METIS à l'interface entre deux sous-domaines. Les mailles bleues correspondent à un domaine, les mailles blanches à l'autre domaine. En rouge la ligne des nœuds communs aux deux sous-domaines est représentée.

De même la figure 4.8 montre en violet les mailles fantômes à ajouter au domaine blanc qui seront donc reçues par ce domaine et envoyées par le domaine bleu.

Cette manière de faire est relativement rapide mais ce n'est pas forcément la meilleure en particulier parce qu'il est difficile d'étendre proprement cette méthode à une seconde couche de mailles fantômes.

Une meilleure méthode qui peut s'étendre à plusieurs couches de mailles fantômes aurait pu être de calculer une matrice de voisinage pour chaque sous-domaine puis de comparer cette matrice à la matrice de voisinage du domaine global. On pourrait ainsi facilement détecter quelles mailles n'ont pas de voisines dans le sous-domaine alors qu'elles devraient en avoir et réagir en conséquence.

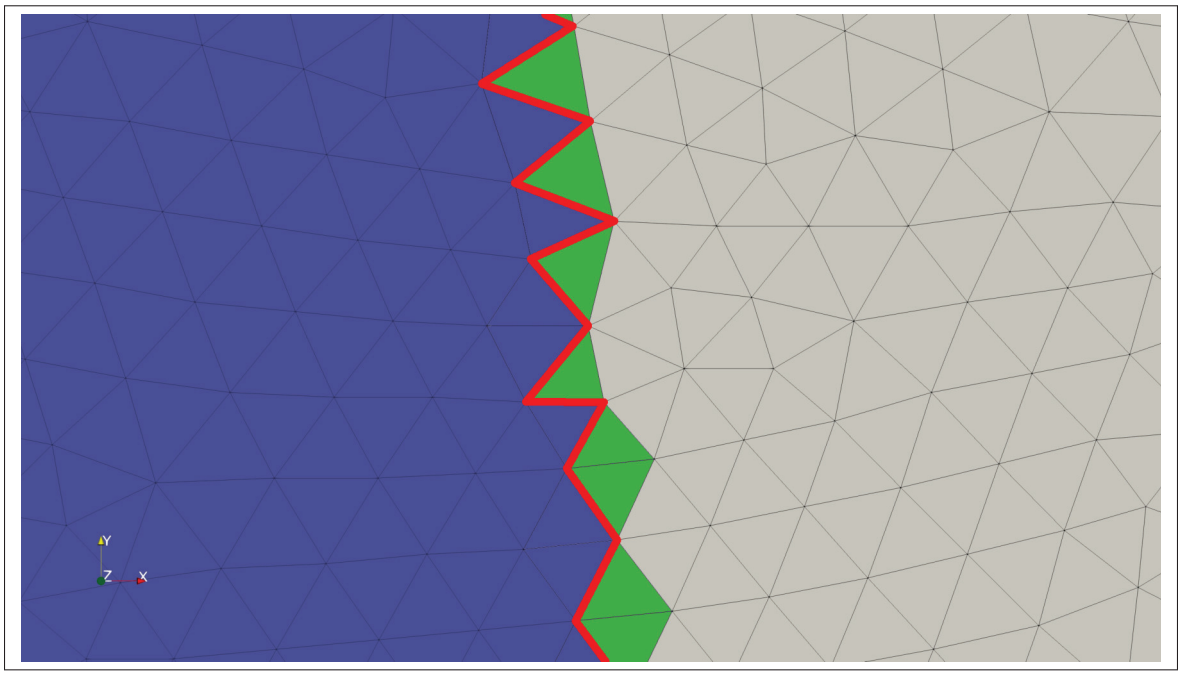


Figure 4.7 Attribution des mailles faites par METIS à l'interface entre deux sous-domaines. Les mailles bleues correspondent à un domaine, les mailles blanches à l'autre domaine. En rouge la ligne des nœuds communs aux deux sous-domaines est représentée. En vert les mailles fantômes à ajouter au domaine bleu.

L'avantage est qu'on peut facilement étendre cette méthode à une seconde couche de mailles fantômes comme il suffit de répéter la même opération. L'inconvénient est que la création de ces matrices n'est pas rapide et il faudrait donc porter cette routine sur GPU pour avoir de bonnes performances. On choisit ici de ne pas le faire pour ce pre-traitement parce qu'il ne sera effectué qu'une seule fois pour chaque maillage et n'est donc pas une priorité pour l'optimisation générale de notre code.

4.4 Renumérotation des mailles fantômes et génération des informations d'envoi/réception

La numérotation des mailles que l'on va envoyer/recevoir est très importante pour les performances des échanges mémoires. En effet lorsque l'on fait un échange mémoire avec MPI il y a un certain temps de latence avant l'envoi des messages. Cela implique qu'il faut les regrouper au maximum pour en envoyer un gros plutôt que beaucoup de petits.

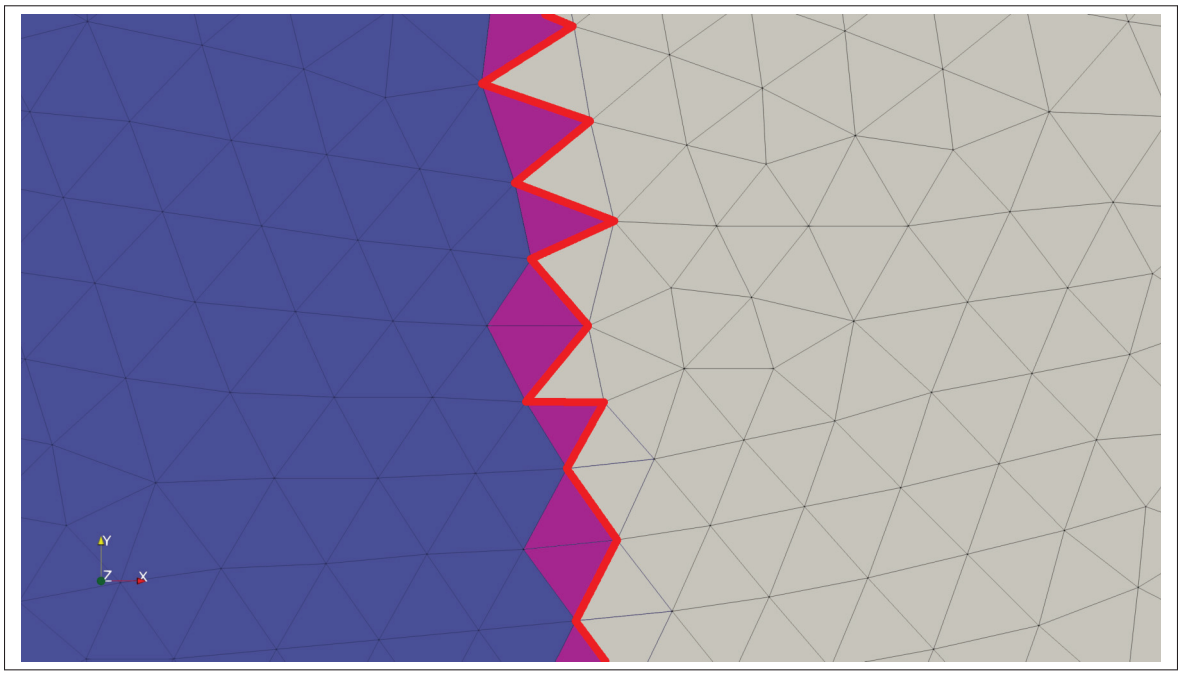


Figure 4.8 Attribution des mailles faites par METIS à l'interface entre deux sous-domaines. Les mailles bleues correspondent à un domaine, les mailles blanches à l'autre domaine. En rouge la ligne des nœuds communs aux deux sous-domaines est représentée. En violet les mailles fantômes à ajouter au domaine bleu.

Notre but est donc d'envoyer toutes les mailles dont un sous-domaine adjacent à besoin d'un seul coup en envoyant un bloc mémoire. Pour cela il faut que les mailles à envoyer à un sous-domaine en particulier soient regroupées en un seul bloc dans notre numérotation. On va donc renuméroter les mailles de façon à avoir : en premier les mailles qui ne seront ni envoyées ni reçues, ensuite les mailles à envoyer à d'autres sous-domaines, regroupées en autant de blocs que de sous-domaines adjacents, puis finalement les mailles fantômes que l'on doit recevoir, regroupées ici aussi en autant de blocs que de sous-domaines adjacents.

Vu la manière dont on ajoute les mailles fantômes à chaque sous-domaine, il est naturel que ces mailles se situent en fin de numérotation et qu'elles forment déjà un bloc pour chaque sous-domaine adjacent. D'un autre côté, les mailles que chaque sous-domaine doit envoyer sont elles issues de la numérotation initiale du maillage et ne sont donc généralement pas côte à côte dans la numérotation. Il va donc nous falloir faire un traitement particulier pour s'assurer que

les mailles que chaque sous-domaine doit envoyer soient bien regroupées en blocs dans notre numérotation. L'objectif est d'avoir un fichier de maillage comme le présente la figure 4.9.

De cette façon on va pouvoir générer directement dans les fichiers de maillages les informations qui seront nécessaires dans la simulation pour envoyer et recevoir les messages directement par bloc. En particulier comme le présente la figure 4.9, chaque fichier de maillage va contenir les indices de départ de chaque bloc à envoyer, leur taille ainsi que l'indice du sous-domaine à qui le bloc doit être envoyé. On procède de la même façon pour les mailles à recevoir, on stocke l'indice de départ du bloc de mailles fantômes à recevoir, la taille de ce bloc ainsi que le numéro du sous-domaine depuis lequel on doit recevoir ce bloc [1].

Sur la figure 4.9 on associe chaque catégorie de maille à une couleur :

- Les mailles non colorées ne sont ni à envoyer ni à recevoir ;
- Les mailles rouges sont à envoyer au sous domaine 3 ;
- Les mailles orange sont à envoyer au sous domaine 5 ;
- Les mailles cyan sont à recevoir du domaine 3 ;
- Les mailles bleues sont à recevoir du domaine 3.

Générer ces données en pré-traitement simplifie ensuite grandement la tâche dans le code de la simulation. On aura directement lu dans les fichiers de maillages toutes les informations nécessaires pour effectuer correctement l'échange des mailles fantômes.

4.5 Écriture des différents fichiers de sortie

Plusieurs fichiers de sorties sont générés à la fin de cette routine de décomposition de domaine. En premier, les fichiers de maillages de chacun des sous-domaines comme présenté en figure 4.9. Comme on l'a dit ces fichiers vont contenir : les coordonnées des nœuds, la connectivité des

[1] Il est possible comme on le constate sur la figure 4.9 qu'il y ait une différence entre le nombre de mailles à envoyer/recevoir avec un domaine. C'est dû au fait que parfois une maille fantôme à deux arêtes sur la ligne des nœuds communs dans un domaine alors que de l'autre côté il a une maille pour chaque arrête

```
Table des coordonnées
30783
    1 275003.0816 5047356.1232 36.3931
    2 275009.0435 5047346.7332 36.4766
...
30782 277426.1926 5047279.5925 33.9046
30783 277426.1926 5047279.5925 33.9046

Table des connectivités
60529
    1       4       3       8   0.0300
    2       2       7      12   0.0300
...
...
59771   30498   30500   30497   0.0300
59772   30497   30500   30501   0.0300
59773       1       2       3   0.0300
59774       1       3       4   0.0300
...
...
59963   18279   18488   18489   0.0300
59964   15536   15656   15657   0.0500
59965   15656   15683   15847   0.0500
...
...
60150   30500   30502   30501   0.0300
60151   30503       1       4   0.0300
60152   30504       5       2   0.0300
...
...
60341   18488   30647   18489   0.0300
60342   15657   15656   30648   0.0500
60343   15656   15847   30649   0.0500
...
...
60528   30499   30501   30782   0.0300
60529   30501   30502   30783   0.0300

Mailles fantômes à envoyer par bloc
2
59773   191     3
59964   187     5

Mailles fantômes à réceptionner par bloc
2
60151   191     3
60342   188     5
```

Figure 4.9 Fichier de maillage pour le sous-domaine 4.

éléments, les indices de départ des blocs de mailles à envoyer/recevoir ainsi que leur taille et l'indice des sous-domaines qui doivent recevoir/envoyer ces blocs. On ne le montre pas sur la figure 4.9 mais le fichier de maillage contient aussi les indices des nœuds d'entrée et sortie du domaine pour l'application des conditions aux limites.

Ensuite on va générer pour chaque sous-domaine une table de correspondance entre la numérotation locale du sous-domaine et la numérotation globale du domaine original. Ces tables vont ensuite servir à regrouper des fichiers solution générés sur chaque sous-domaine sur le domaine original, ou à l'inverse à découper une solution initiale sur chacun des sous-domaines. Ces fichiers sont très importants car ils permettent d'écrire des routines de découpe de n'importe quel type de fichiers avec la même décomposition et numérotation que la découpe du maillage original. Cela permet une grande versatilité, par exemple si on a généré une solution sur 2 sous-domaines et qu'on veut relancer le code avec 4 sous-domaines, il nous suffit de recombiner les 2 premiers fichiers solution en un seul fichier sur le domaine global puis de découper ce dernier sur les 4 sous-domaines voulus pour la suite.

Finalement on pourra aussi générer sur demande des fichiers contenant les nœuds communs à deux sous-domaines adjacents ou encore afficher les nœuds d'entrée, sortie et mur de chaque sous-domaine. Ces sorties ne sont pas obligatoires mais permettent de vérifier le bon fonctionnement de la décomposition que l'on vient de faire.

CHAPITRE 5

PROGRAMMATION MULTI-GPU AVEC MPI ET CUDA FORTRAN

MPI (Message Passing Interface) est un standard qui définit des fonctions de communications entre plusieurs processeurs ou ordinateurs distants. OpenMPI (Gabriel *et al.*, 2004) en est une implémentation, c'est ce que nous utilisons ici.

L'utilisation de OpenMPI nous permet de lancer différentes copies du code sur des processeurs différents. Chaque *process* MPI va alors avoir sa mémoire propre qu'il sera le seul à pouvoir modifier. Dans notre approche chaque *process* MPI va traiter un des sous-domaine en utilisant un GPU.

On présente d'abord les fonctions classiques de MPI que nous utilisons dans notre code, on présente ensuite plusieurs façons réaliser l'échange mémoire des mailles fantômes.

5.1 Présentation de MPI

On présente rapidement le fonctionnement de MPI et quelques fonctions de communication basiques sans trop rentrer dans le détail. L'objectif est ici de donner un aperçu global de l'utilisation de MPI.

5.1.1 Initialisation, rang et type de variables

Pour commencer on peut initialiser l'environnement MPI avec,

```
integer :: mpi_ierr
call MPI_INIT(mpi_ierr)
```

Cette initialisation va automatiquement créer le communicateur global *MPI_COMM_WORLD* sur lequel tous les *process* MPI vont pouvoir communiquer. On peut ensuite récupérer le rang

du *process* MPI avec,

```
integer :: mpi_ierr, mpi_process_rank
call MPI_COMM_RANK(MPI_COMM_WORLD, mpi_process_rank, mpi_ierr)
```

et le nombre total de *process* avec

```
integer :: mpi_ierr, num_mpi_process
call MPI_COMM_SIZE(MPI_COMM_WORLD, num_mpi_process, mpi_ierr)
```

Le *rang* du *process* MPI correspond à son numéro, si on lance 4 *process* MPI on aura des *process* de rang, 0, 1, 2 et 3. Lors de l'initialisation on va aussi créer un type de variables pour MPI à utiliser lors des échanges mémoire. Dans **CUTEFLOW** on peut choisir la *simple* ou *double* précision pour le calcul en définissant la variable *fp_kind*. On définira dans le même temps, *fp_kind_mpi*, le type de variables pour MPI. Pour la *simple* précision,

```
integer , parameter :: fp_kind = kind(0.0)
integer , parameter :: fp_kind_mpi = MPI_REAL
```

et pour la *double* précision,

```
integer , parameter :: fp_kind = kind(0.0D0)
integer , parameter :: fp_kind_mpi = MPI_DOUBLE_PRECISION
```

5.1.2 Communications globales : Exemple du *REDUCE* et *ALLREDUCE*

Certaines fonctions de MPI permettent de faire des communications globales entre tous les processeurs d'un même communicateur. Parmi ces fonctions, la fonction *REDUCE* permet d'effectuer une réduction sur des valeurs qui sont sur différents processeurs. Une réduction peut être de plusieurs types, souvent on cherche le minimum, le maximum ou la somme entre tous

les éléments d'un vecteur dont chaque processeur possède un élément. La liste complète des réductions possibles peut être trouvée dans la documentation de MPI.

La réduction qui nous intéresse est celle qui retourne le minimum entre plusieurs valeurs d'un vecteur. C'est une des réductions que l'on utilise dans **CUTEFLOW** pour trouver le pas de temps minimum sur le domaine global à partir des pas de temps minimum de chaque sous-domaine. Dans la documentation on peut trouver,

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM,
        IERROR)
    <type>    SENDBUF(*), RECVBUF(*)
    INTEGER    COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

avec,

- *SENDBUF* le *buffer* à envoyer,
- *RECVBUF* le *buffer* à envoyer,
- *COUNT* la taille du *buffer*,
- *DATATYPE* le type de variables,
- *OP* l'opération de réduction,
- *ROOT* le rang du *process* qui doit recevoir le résultat,
- *COMM* le communicateur sur lequel procéder à la réduction,
- *IERR* le statut d'erreur.

Cette fonction permet de faire une réduction et d'avoir le résultat sur le *process* de rang *ROOT*. La plupart du temps, ce que l'on veut vraiment, c'est que tous les *process* aient le résultat de la réduction. On va donc utiliser la fonction *ALLREDUCE*. Cette fonction est identique à la fonction *REDUCE* hormis qu'il n'est pas nécessaire de spécifier un *process* comme *ROOT*.

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM,
        IERROR)
    <type>    SENDBUF(*), RECVBUF(*)
    INTEGER    COUNT, DATATYPE, OP, COMM, IERROR
```

avec,

- *SENDBUF* le *buffer* à envoyer
- *RECVBUF* le *buffer* à envoyer
- *COUNT* la taille du *buffer*
- *DATATYPE* le type de variables
- *OP* l'opération de réduction
- *COMM* le communicateur sur lequel procéder à la réduction
- *IERR* le statut d'erreur

Certaines autres fonctions de communication globales fonctionnent de la même manière avec une variante *ALL* qui distribue le résultat sur tous les *process*, par exemple *GATHER* et *ALLGATHER* qui permettent de récupérer les valeurs d'un vecteur distribué sur plusieurs *process*.

La dernière chose que nous allons utiliser pour effectuer la réduction est la variable *MPI_IN_PLACE* qui, quand on l'utilise comme *SENDBUF*, permet à la valeur *RECVBUF* de servir à la fois de variable d'arrivée et de départ pour la réduction. Cela nous permet, pour trouver le minimum de la variable *dt* sur tous les *process*, de simplement faire,

```
call MPI_ALLREDUCE(MPI_IN_PLACE, dt, 1,
        fp_kind_mpi, MPI_MIN, MPI_COMM_WORLD, mpi_ierr)
```

De cette façon, on effectue une réduction qui recherche le minimum, grâce à *MPI_MIN*, sur tous les *process* du communicateur *MPI_COMM_WORLD* et qui met le résultat directement dans la variable *dt*. Autrement dit en une ligne chaque *process* a récupéré la valeur du pas de temps minimum dans sa variable *dt*.

Ces fonctions de communication globale sont très puissantes et il faut les utiliser au maximum. Cependant dans certains cas ce n'est pas facile, voire impossible. Dans ce cas on se tourne vers des communications point à point qu'on présente dans la partie suivante.

5.1.3 Communications point à point : *SEND* et *RECV*

Les communications point à point permettent d'envoyer un message depuis un *process* MPI vers un autre très simplement. On peut trouver dans la documentation de MPI,

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type>     BUF(*)
    INTEGER     COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

avec,

- *BUF* l'adresse initiale du *buffer* à envoyer
- *COUNT* la taille du *buffer*
- *DATATYPE* le type de variables
- *DEST* le rang du *process* destinataire
- *TAG* un identifiant pour le message
- *COMM* le communicateur sur lequel la communication doit se passer
- *IERR* le statut d'erreur

La fonction *RECV* est identique à l'exception qu'il faut spécifier le rang du *process* émetteur du message au lieu du *process* destinataire.

On comprend mieux ici le travail du pré-traitement dans la décomposition en sous-domaines et le choix du format du fichier de maillage de la figure4.9. Dans ce fichier, on avait spécifié l'indice de départ des mailles fantômes à envoyer, le nombre de mailles à envoyer ainsi que le numéro du sous domaine destinataire. Le numéro de la première maille à envoyer servira de *BUF*, le nombre de mailles de *COUNT* (à une multiplication près), et le numéro du sous-domaine destinataire servira de *DEST*. Finalement, si besoin, on pourra générer un *TAG* spécifique à chaque message.

5.1.4 Communications non-bloquantes

La plupart des fonctions définies dans MPI ont un équivalent avec le préfixe *I*, comme *ISEND*, *IRECV* ou encore *IALLREDUCE*. Ces fonctions agissent de manière non bloquante c'est-à-dire

que l'exécution va continuer sans attendre que ces fonctions aient retourné. Ces fonctions vont prendre un argument supplémentaire, *REQUEST*, qui nous permettra d'attendre la fin de la communication avec une fonction du type *MPI_WAIT*.

Ce genre de fonction est utile pour superposer des calculs avec des échanges mémoire. On peut par exemple lancer un *ISEND* juste avant de faire un calcul, de cette façon l'échange mémoire se fera en même temps que le calcul. Il faudra cependant faire attention à ce que l'on n'échange pas des valeurs dont à besoin pour faire le calcul. Il faudra donc faire un usage adéquat de la fonction *MPI_WAIT* pour attendre la fin des communications.

5.2 Échange de mémoire sur le GPU de manière classique

Dans la partie précédante on a vu comment échanger des variables entre les différents *process* MPI. Normalement cette façon de faire ne fonctionne qu'avec des variables définies sur le CPU. Lorsque l'on couple MPI avec du CUDA-Fortran chaque *process* MPI possède des variables propres sur le CPU et sur le GPU qui lui est associé. Pour échanger les variables du GPU il faut alors, dans le cas classique, copier ces variables sur le CPU puis faire l'échange mémoire MPI et ensuite copier les variables sur le GPU en retour. Par exemple pour faire la réduction sur le pas de temps *dt*,

```
ierr = cudaMemcpy(dt,dt_d,1)
call MPI_ALLREDUCE(MPI_IN_PLACE, dt, 1,
        fp_kind_mpi, MPI_MIN, MPI_COMM_WORLD, mpi_ierr)
ierr = cudaMemcpy(dt_d,dt,1)
```

On peut évidemment faire plus complexe pour superposer les échanges mémoires avec des calculs en utilisant des *streams* pour faire une copie asynchrone entre le CPU et le GPU et/ou en utilisant la version non-bloquante de *ALLREDUCE*, *IALLREDUCE*.

Bien que cette méthode fonctionne, ce n'est généralement pas la plus efficace surtout si on ne superpose pas les échanges avec des calculs. Une meilleure méthode, qui est aussi plus simple à

programmer, consiste à utiliser du CUDA-*Aware* OpenMPI c'est ce qu'on présente dans la partie suivante.

5.3 Échange mémoire sur GPU en utilisant du CUDA-*Aware* OpenMPI

Utiliser du CUDA-*Aware* OpenMPI signifie que la librairie OpenMPI a été compilée avec le support pour CUDA et peut échanger la mémoire du GPU directement. C'est une possibilité qui est présente depuis la version 1.7 de OpenMPI.

Dans la documentation on peut lire, "Now, the Open MPI library will automatically detect that the pointer being passed in is a CUDA device memory pointer and do the right thing." [1](ope). On ne va pas entrer en détail dans ce que *the right thing* veut dire mais on peut donner quelques indices.

Si deux GPU sont sur le même noeud de calcul il est possible qu'ils puissent communiquer en *Peer-To-Peer*, c'est-à-dire qu'ils vont communiquer directement entre eux sans repasser pas la mémoire de l'hôte. On peut alors écrire une fonction qui vérifie si les GPU que l'on veut faire communiquer peuvent communiquer en *Peer-To-Peer*. Si c'est le cas on peut alors faire un échange mémoire directement entre les deux GPU en utilisant la fonction *cudaMemcpy2D()* du CUDA Fortran. Si les GPU ne peuvent pas communiquer en *Peer-To-Peer* on sera alors obligé de copier les variables du GPU vers le CPU puis d'utiliser MPI pour faire l'échange mémoire. C'est le maximum que l'on peut faire au niveau de la programmation et l'utilisation de CUDA-*Aware* OpenMPI nous permet de ne pas se soucier de ce genre de choses et utilisera l'échange en *Peer-To-Peer* lorsque cela sera possible.

On pourrait alors voir le CUDA-*Aware* OpenMPI comme une facilité de programmation qui nous permettrait de ne pas se soucier de comment l'échange mémoire est fait. Cependant, comme certaines fonctionnalités sont trop bas niveau pour être accessibles depuis le CUDA-Fortran,

[1] traduction libre : "Maintenant, la librairie OpenMPI va automatiquement détecter que le pointeur pointe sur la mémoire du GPU et va faire la bonne chose.

l'utilisation de CUDA-*Aware* OpenMPI est vraiment nécessaire pour avoir les meilleures performances. Une telle fonctionnalité est par exemple le GPU Direct RDMA (Remote Direct Memory Access) qui permet l'échange mémoire direct entre des GPU sur des noeuds de calculs différents. Cette fonctionnalité n'est pas accessible depuis la programmation en MPI et CUDA-Fortran, la seule façon de l'utiliser est au travers d'une version CUDA-*Aware* d'OpenMPI. Ce n'est évidement pas le seul avantage à utiliser du CUDA-*Aware* OpenMPI, comme tout est géré par la librairie, tous les échanges mémoires inutiles sont évités.

Utiliser du CUDA-*Aware* OpenMPI présente aussi certains désavantages. Il est plus difficile de savoir ce qui est utilisé par la librairie pour faire l'échange mémoire et il peut être difficile de savoir si l'on utilise vraiment un système au maximum de ses performances. De plus, pour un bon fonctionnement, il faut que la librairie OpenMPI soit particulièrement compilée pour accommoder le système. Dans notre cas, elle doit être compilée au minimum avec le support pour CUDA et pour le compilateur *pgf90*. On a pu constater que ce n'est pas forcément évident de trouver une telle version de OpenMPI déjà présente sur les grappes de calculs et il a souvent fallu de longues discussions avec le support des grappes de calculs pour avoir quelque-chose de fonctionnel. Des discussions sont en ce moment toujours en cours quant à l'activation du support pour le GPU Direct RDMA sur les clusters que nous utilisons.

On peut avec du CUDA-Aware OpenMPI faire la réduction directement sur la variable *dt_d* déclarée sur le GPU de la manière suivante,

```
call MPI_ALLREDUCE(MPI_IN_PLACE, dt_d, 1,
        fp_kind_mpi, MPI_MIN, MPI_COMM_WORLD, mpi_ierr)
```

CHAPITRE 6

CUTEFLOW EN MPI/CUDA-FORTRAN

On présente dans cette partie le fonctionnement de **CUTEFLOW**. C'est un programme développé dans le groupe de recherche du GRANIT de l'ETS pour résoudre les équations de St-Venant.

La première version séquentielle sur CPU a été développée par Azzeddine Soulaïmani et Youssef Loukili en suivant le travail dans Loukili & Soulaïmani (2007), la version séquentielle a ensuite été modifiée par Jean-Marie Zokagoa en suivant le travail effectué dans Zokagoa & Soulaïmani (2010). Le code a ensuite été porté sur GPU en CUDA Fortran par Arun Kumar Suthar on peut trouver le travail effectué dans Suthar & Soulaïmani (2018). Finalement dans ce travail on a utilisé MPI pour porter le code sur une architecture multi-GPU.

6.1 Compilation

CUTEFLOW utilise MPI et du CUDA-Fortran, on utilise alors le compilateur de PGI, *pgf90* pour compiler le code avec le *wrapper mpif90* d'OpenMPI.

Sur les clusters de calculs on utilise les versions suivantes de différentes modules :

- pgi/19.4 ;
- cuda/10.0.130 ;
- openmpi/3.1.2 .

Pour compiler le code il suffit de savoir la *Compute Capability* des GPU sur lesquels le code sera exécuté. Par exemple les GPU Tesla SMXV100 présents sur BELUGA ont une *Compute Capibility* de 7.0 alors que les GPU P100 de CEDAR et GRAHAM ont une *Compute Capability* de 6.0. La plupart du temps on compilera avec les options d'optimisation au maximum avec l'option $-O3$.

Extrait 6.1 Makefile pour compilation de **CUTEFLOW** sur BELUGA

```
CFLAGS = -Mcuda=cc70 -O3
CC = mpif90

OBJS = general_cudaf.cuf ...

runfile : ${OBJS}
  ${CC} ${CFLAGS} ${OBJS} -c
  ${CC} ${CFLAGS} ${OBJS} -o $@
```

Même si le *makefile* est très simple, il faudra souvent prendre le temps de vérifier que tous les modules que l'on utilise soient compatibles entre eux pour ne pas avoir d'erreur lors de la compilation. Faire fonctionner la compilation de ce code a été une tâche longue parce qu'il faut souvent plusieurs interactions avec le support des grappes de calcul pour avoir des modules compatibles entre eux pour l'utilisation qu'on veut en faire.

Il faudra par exemple vérifier que les commandes suivantes retournent la valeur $True$. Pour le support du CUDA par la librairie OpenMPI,

```
shell$ ompi_info --parsable --all |
        grep mpi_built_with_cuda_support:value
```

ou encore si on veut le support du GPU Direct RDMA,

```
shell$ ompi_info --all | grep btl_openib_have_cuda_gdr
```

et

```
shell$ ompi_info --all | grep btl_openib_have_cuda_gdr
```

On peut retrouver toutes ces informations sur le site de OpenMPI (https://www.open-mpi.org/faq/?category=runcuda)

6.2 Maillages

Comme on l'a montré dans la revue de littérature, beaucoup de codes qui utilisent des GPU utilisent des maillages structurés. C'est en grande partie dû à deux choses :

- La décomposition de domaine sur des maillages structurés est souvent très simple,
- Les accès mémoires sont plus optimisés avec des maillages structurés.

Le premier point est traité grâce à la bibliothèque METIS et a été présenté dans le chapitre 4

On ne pourra jamais avoir des accès mémoires aussi bons qu'avec un maillage structuré. On peut néanmoins les optimiser en renumérotant les mailles avec par exemple un algorithme de Cuthill-McKee (Cuthill & McKee, 1969), c'est généralement une étape qui est effectuée lors de la création du maillage.

CUTEFLOW est écrit pour prendre en charge uniquement des maillages 2D triangulaires non structurés. C'est un type de maillage qui permet de mailler facilement des géométries complexes, par exemple les zones autour des piles des ponts qui traversent les rivières.

6.3 Initialisation

Comme on l'a fait remarquer en introduction, on a deux problèmes distincts lors de l'exécution des calculs. Le premier, est de créer une condition initiale stabilisée pour s'en servir dans un second temps comme condition initiale pour les études suivantes.

On a différents choix pour initialiser le domaine :

- Initialiser un problème de Riemann de chaque côté d'une droite du plan,
- Initialiser avec un plan qui décrit la surface libre,
- Initialiser à partir d'un fichier de solution initiale.

Pour initialiser le domaine quand on n'a pas encore de solution, on va utiliser l'initialisation avec un plan. On choisira alors un point du domaine et un vecteur normal au plan pour décrire la surface libre. Les vitesses seront nulles au départ. Ce genre d'initialisation demande parfois

plusieurs essais parce qu'il est courant qu'il y ait des dénivelés importants dans certaines zones. On aura alors, soit trop d'eau dans le domaine et on devra alors attendre qu'il se vide, soit trop peu d'eau et il faudra attendre que le domaine se remplisse avec le désavantage dans les deux cas qu'il y aura beaucoup de bancs couvrant-découvrant à gérer ce qui peut provoquer des problèmes de stabilité.

Une fois une première solution stable générée on initialisera les simulations suivantes en partant de cette solution.

6.3.1 Pas de temps local

Pour faciliter la première initialisation on peut utiliser un pas de temps local dans chaque cellule. Dans ce cas on va utiliser le pas de temps calculé dans chacune des cellules pour mettre à jour la valeur des inconnues dans cette cellule. La solution alors générée pendant le régime transitoire ne sera pas physique car les mailles vont avoir des temps différents, cependant la solution au régime permanent le sera. Une telle méthode est présentée dans Elkadri E, Soulaïmani & Deschênes (2000).

Cette méthode permet en théorie d'arriver plus rapidement au régime permanent car les cellules très petites ne vont pas ralentir le calcul des grandes. C'est une méthode que nous sommes en train de tester et de valider dans le code comme cela peut provoquer certains problèmes de stabilité.

6.3.2 Mailles sèches comme murs

Une autre façon de faciliter la première initialisation, est de remarquer que souvent ce qui pose problème est la présence de grandes zones de couvrant-découvrant. Il est en effet courant avec une initialisation avec un plan d'avoir de grandes zones inondées qui doivent se vider, ou au contraire des zones sèches qui doivent se remplir.

On peut éviter quelques problèmes en considérant que les mailles sèches lors de l'initialisation sont comme des murs et qu'aucun flux ne peut y rentrer. Aucune zone ne pourra alors se remplir d'eau. Il faut alors coupler l'utilisation des mailles sèches comme murs avec une initialisation où on surestime la hauteur d'eau dans le domaine. Ce genre de technique peut parfois éviter certains problèmes pour certains domaines.

6.4 Traitement des conditions aux limites

Dans le traitement des équations de St-Venant, on a en pratique à gérer uniquement des conditions d'entrée, de sortie et de murs. Les informations sur les nœuds d'entrée et sortie doivent être présentes dans les fichiers de maillages. Les mailles qui n'ont pas de voisines à travers une arête seront automatiquement attribuées une condition de mur sur cette arête.

Une des contributions de ce travail a été l'ajout du traitement de plusieurs entrées et plusieurs sorties pour un domaine. Précédemment, on pouvait uniquement gérer une seule entrée et une seule sortie, mais avec la possibilité d'utiliser des domaines de plus en plus grands il était nécessaire de pouvoir en traiter plusieurs.

La plupart du temps, en régime fluvial, on impose des conditions de débit entrant aux entrées, en spécifiant un certain débit pour chaque entrée, et on impose une certaine hauteur d'eau pour chaque sortie du domaine. On a explicité le calcul de ces conditions dans le chapitre 2.

6.5 Fichiers de sortie

Il est possible de générer différents fichiers de sortie pour le post-traitement :

- Solution aux nœuds,
- Solution aux éléments,
- Solution au format *vtk* pour Paraview,
- Solution au format *T3S* pour Bluekenue,
- Solution selon des coupes,

- Solution en certains nœuds seulement (jauges).

On pourra reprendre les fichiers de solutions aux éléments comme fichier de condition initiale pour faire repartir la simulation.

Lors de ce travail, on a ajouté la sortie de fichiers au format *vtk* pour faire une visualisation avec Paraview Ayachit (2015). Paraview est un logiciel très utilisé pour la visualisation scientifique. Il est présent sur toutes les grappes de Compute Canada et est utilisable en mode serveur-client. Cela signifie que l'on peut lancer un serveur Paraview sur une grappe de calcul et s'y connecter depuis son ordinateur personnel. Cela simplifie grandement les choses, on n'a plus à télécharger les résultats de simulations, ce qui, avec des domaines de plus en plus grands, devenait dans tous les cas de moins en moins pratique. Paraview est en plus utilisable avec plusieurs GPU ou CPU sur les grappes de calcul ce qui accélère grandement le post-traitement. Finalement, il permet de regrouper des fichiers de simulations ce qui nous permet de facilement regrouper les fichiers résultats générés par chaque *process* MPI de nos simulations.

6.6 Description du fonctionnement de CUTEFLOW

La figure 6.1 représente le fonctionnement simplifié du code CUTEFLOW, on discute ici de quelques fonctions importantes.

6.6.1 Recherche des voisins

Une des premières fonctions lancée sur le GPU est la recherche de mailles voisines à chacune des mailles. On construit lors de cette étape une table de connectivité entre les mailles. Cette table nous indique quelle maille est adjacente à la maille i via le côté k et est particulièrement utile lors du calcul des flux.

Cette fonction écrit la table de connectivité des mailles dans un fichier de telle façon que lors des futures simulations on n'aura pas besoin de la recalculer. Dans le futur il serait utile d'effectuer cette étape dans le code qui fait la décomposition en sous domaines.

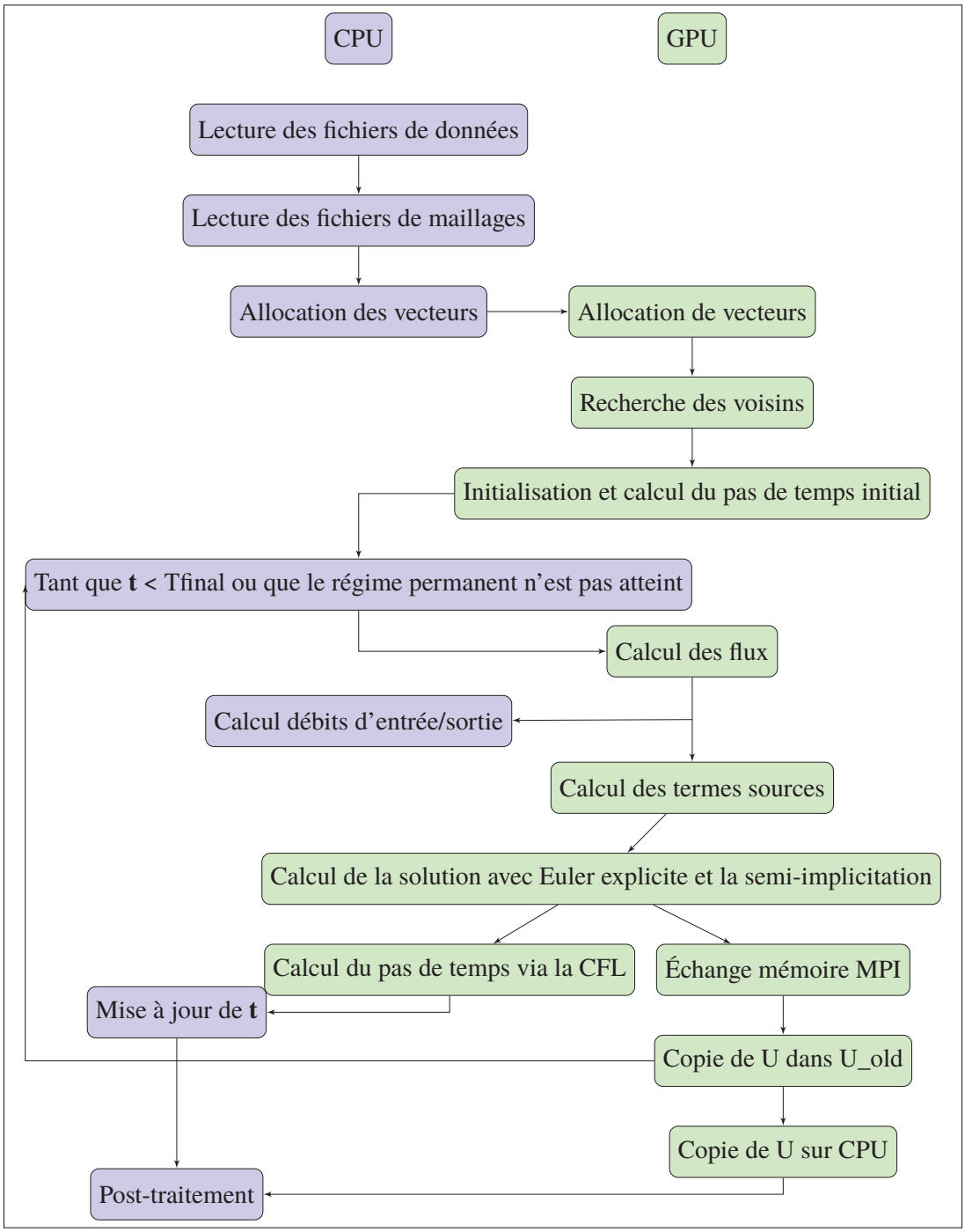


Figure 6.1 Graphe du fonctionnement de **CUTEFLOW**

6.6.2 Calcul des flux

Le calcul des flux pour chaque maille est fait sur le GPU. On associe à chaque côté de chaque maille un *thread* en utilisant la configuration des *threads* et *block* suivante,

Extrait 6.2 Configuration de lancement pour le calcul des flux

```
threads = dim3(256, 3, 1)
blocks = dim3(ceiling(nelt/real(threads%x)), 1, 1)
```

où *nelt* désigne le nombre d'éléments du maillage. (voir chapitre 3 pour plus de précisions)

De cette façon on a trois *threads* par maille, un pour chaque côté de la maille. Chaque *thread* va ensuite traiter la résolution du problème de Riemann sur le côté qui lui correspond comme indiqué dans le chapitre 2.

6.6.3 Calcul des termes sources

Les calculs des termes sources, que ce soit ceux de friction ou de bathymétrie sont faits très simplement en associant un *thread* à chaque maille et en faisant les calculs explicités dans le chapitre 2.

Comme le calcul des termes sources ne dépend pas du calcul des flux, ces deux *kernel* peuvent être lancés sur des *stream* différents sur le GPU pour qu'ils se superposent. Même si c'est la bonne idée, ces *kernels* utilisent déjà au maximum le GPU et on a pu constater qu'il ne se superposent que très peu à la manière de la figure 3.6.

6.6.4 Calcul de la solution avec Euler explicite et la semi-implicitation

Ici aussi c'est le même processus, un *thread* est associé à chaque maille, on calcule l'inverse de la matrice locale à cause de la semi-implicitation puis on met à jour la solution comme décrit dans l'équation 2.72 du chapitre 2 que l'on rappelle ici,

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = \left[I - \frac{\Delta t}{2} J_f \right]^{-1} \left[-\frac{1}{|\Omega_i|} \sum_{j=1}^{3} L_{ij} T_{n_{ij}}^{-1} \tilde{G}^{HLLC}(U_L^n, U_R^n).n_i + S_f(U_i^n) + S_{O_i}^n(U_i^n, U_{ij}^{**^n}, n_{ij}) \right]$$

Comme on a déjà calculé le flux du schéma HLLC et les termes sources avec les fonctions précédentes c'est une étape très rapide.

6.7 Superposer le calcul du pas de temps via la CFL et les échanges MPI

Dès qu'on a mis à jour la nouvelle solution on peut commencer le calcul du nouveau pas de temps via la condition CFL. Ce n'est pas une étape qui est censée être très longue en soit parce que chaque *thread* va calculer un pas de temps local pour chaque maille. Cependant lorsque l'on ajoute la réduction sur GPU, pour trouver le pas de temps minimum dans le sous domaine traité, et ensuite la réduction via un *ALLGATHER* sur le pas de temps (voir la fin du chapitre 5), pour avoir le pas de temps minium de tous les sous domaines, c'est une étape qui devient assez longue.

On en profite alors pour faire l'échange des mailles fantômes entre les sous-domaines en même temps. Cela ne pose pas de problème parce-que le calcul du nouveau pas de temps ne concerne que les mailles intérieures du sous domaine. On va alors lancer un échange non bloquant des mailles fantômes juste avant de faire toutes les opérations sur le pas de temps pour superposer au maximum ces deux étapes.

6.8 Fin de simulation et post-traitement

On a deux conditions d'arrêt dans le code,

- si le temps de la simulation a atteint le temps maximal,
- si le débit de sortie est égal au débit d'entrée pendant un certain temps.

Quand une des deux conditions est vérifiée, la solution est copiée sur le CPU, les différents fichiers de sortie peuvent être écrits et le post-traitement sera effectué avec d'autres codes.

CHAPITRE 7

RÉSULTATS

Dans ce chapitre on commence par présenter les différentes grappes de calculs utilisées, puis, après avoir défini le *speed-up* et l'efficacité, on présente différents résultats pour un cas de bris de barrage (*Dam-Break*) classique puis pour le cas de la rivière des Mille Îles et finalement pour le cas de l'archipel de Montréal.

7.1 Grappes de calculs utilisées

La plupart des résultats présentés ici ont été générés sur BELUGA. C'est une grappe de calcul qui a été inaugurée durant l'année 2019 et qui est physiquement présente à l'ETS. Sur ce cluster il y a 172 noeuds comprenant chacun 4 GPU NVIDIA V100SXM2 ainsi que 40 cœurs CPU répartis sur 2 CPU Intel Gold 6148 Skylake @ 2.4 GHz. Ce sont parmi les GPU les plus puissants auxquels on peut avoir accès de nos jours.

Durant le travail on a aussi utilisé CEDAR et GRAHAM qui sont deux autres grappes de calculs de *Compute Canada* qui utilisent les GPU NVIDIA P100. Ces deux grappes de calculs ont une configuration presque identique ce qui rend le port du code de l'une à l'autre très simple.

On a pu constater une accélération de 20% lors du passage des GPU NVIDIA P100 de CEDAR et GRAHAM aux GPU NVIDIA V100SXM2 de BELUGA.

Le fait d'utiliser BELUGA pour calculer les *speed-up* et efficacités plutôt que CEDAR ou GRAHAM est à notre désavantage. En effet plus les GPU sont performants moins il est facile de superposer les calculs aux échanges mémoires et plus les différentes latences vont ressortir. On a pu constater de meilleurs *speed-up* sur CEDAR ET GRAHAM bien que les temps finaux soient meilleurs sur BELUGA. L'utilisation de BELUGA étant plus représentative des performances futures du code, c'est sur cette grappe de calcul que nous avons calculé les *speed-up* et efficacités présentés dans la suite.

7.2 Définition du *Speed-Up* et de l'efficacité

Avant de présenter les résultats on rappelle la définition du *Speed-Up* et de l'efficacité. Dans la suite on comparera toujours les temps sur n GPU au temps sur 1 GPU. On aura alors pour le *speed-up*,

$$\text{Speed-Up} = \frac{\text{Temps sur 1 GPU}}{\text{Temps sur } n \text{ GPU}}$$

et pour l'efficacité,

$$\text{Efficacité} = \frac{\text{Temps sur 1 GPU}}{\text{n * Temps sur } n \text{ GPU}}$$

7.3 Cas d'un bris de barrage unidimensionnel

Le cas test qu'on présente ici correspond à la résolution d'un problème de Riemann unidimensionnel. Pour simuler ce problème on prend un domaine rectangulaire $\Omega = [-10, 10] * [0, 100]$. On choisit l'initialisation suivante,

$$\begin{cases} h & = \begin{cases} 10 \text{ si } y < 50, \\ 1 \text{ si } y > 50. \end{cases} \\ u & = 0 \\ v & = 0 \end{cases}$$

On utilise un maillage basique qu'on génère à partir de points placés sur une grille dans le plan xy. La figure 7.1 montre un maillage grossier du domaine avec 400 éléments.

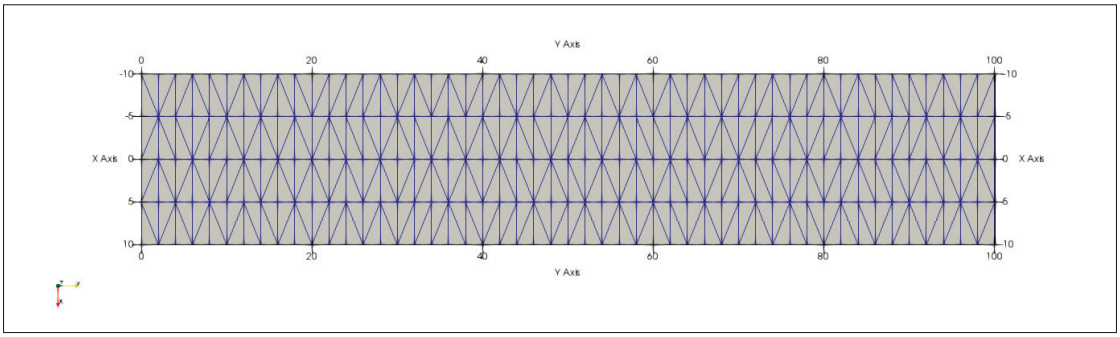


Figure 7.1 Maillage de 400 éléments pour le cas du *Dam break*

7.3.1 Solutions

On présente les solutions obtenues à différents instants. Ce sont des résultats très classiques que l'on peut retrouver entre autres dans Ata *et al.* (2013) ; Toro (2001) ; Zokagoa & Soulaïmani (2010).

Tableau 7.1 Solution pour le cas du *Dam break* sur un maillage de 400 000 éléments.

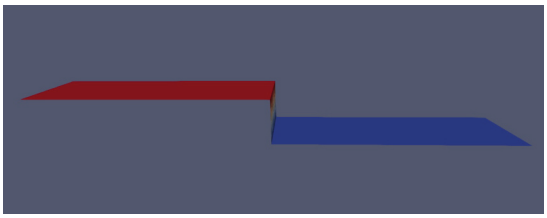
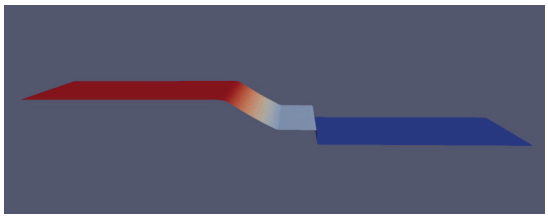
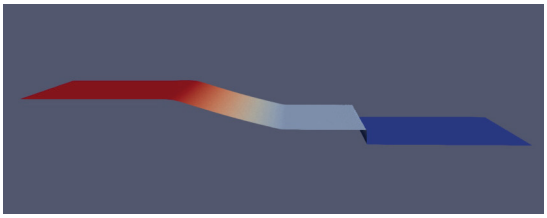
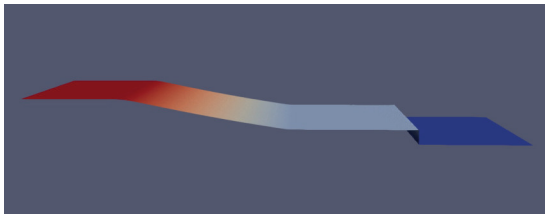
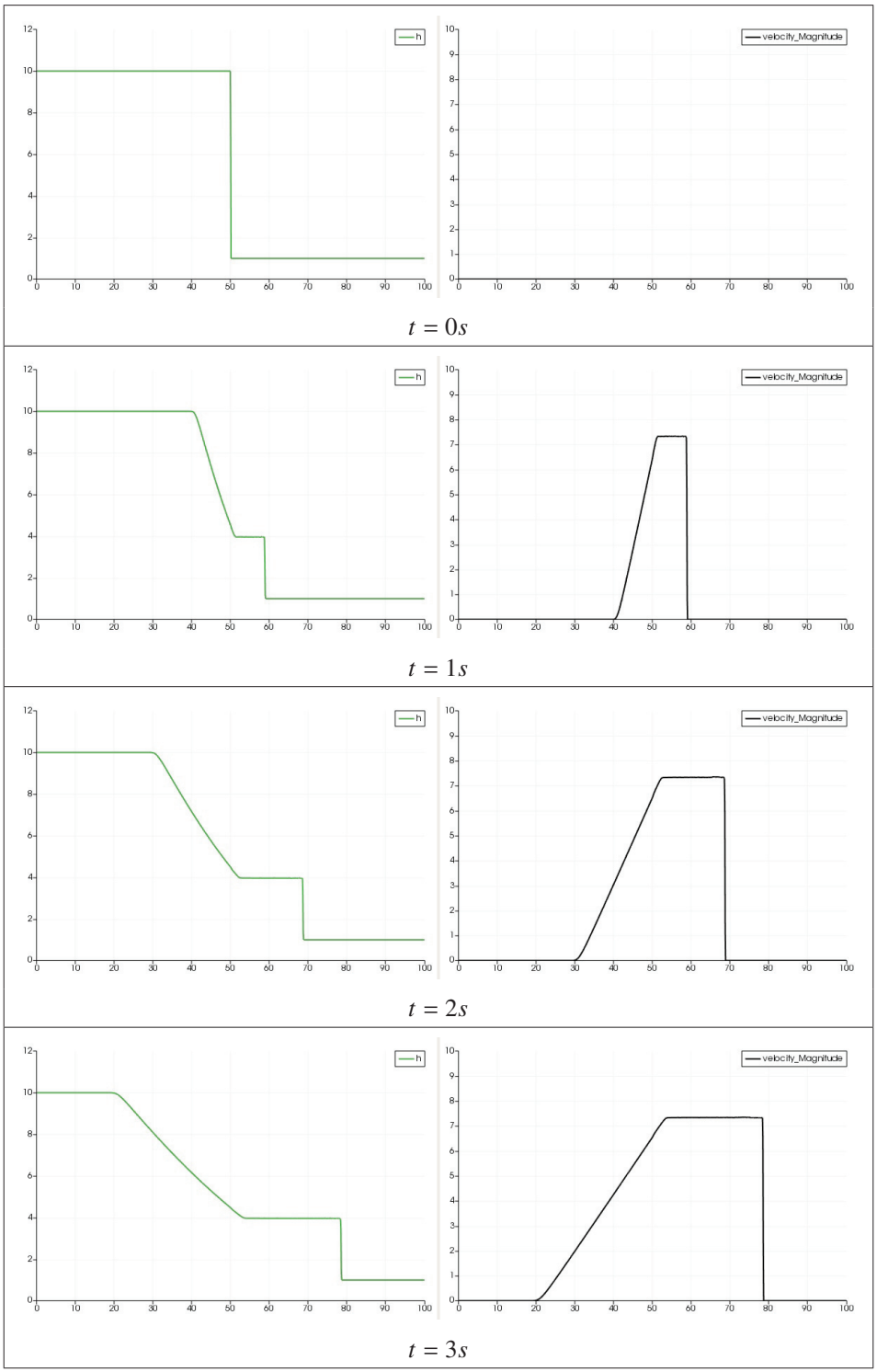
 $t = 0s$	 $t = 1s$
 $t = 2s$	 $t = 3s$

Tableau 7.2 Solution projetée sur l'axe y pour le cas du *Dam break* sur un maillage de 400 000 éléments.



7.3.2 *Speed-Up* et efficacité pour différents maillages

On présente dans le tableau 7.3 les *speed-up* et efficacités obtenus pour différents maillages de 400 000, 1 600 000, 6 300 000 et 13 000 000 éléments.

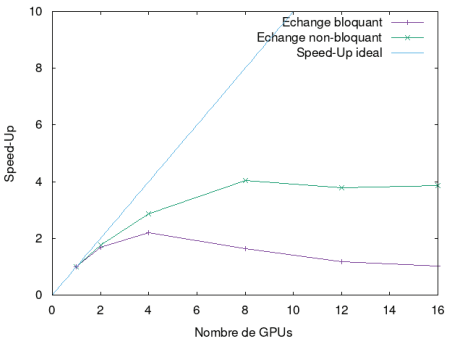
Dans un premier temps, on constate que les performances de l'échange non bloquant sont toujours bien meilleures que celles de l'échange bloquant. C'est un résultat auquel on s'attendait parce que l'échange non bloquant nous permet de lancer tous les échanges mémoires en même temps en plus de les superposer à des calculs comme le calcul de la condition CFL.

Ensuite, on constate que sur le plus petit maillage de 400 000 éléments, on est loin du *speed-up* idéal avec au maximum un facteur d'accélération de 4, même en utilisant 16 GPU. On explique cela assez simplement : le maillage est trop petit pour permettre de superposer correctement les calculs aux échanges mémoires. Dans ce cas on est strictement limité par des latences, qu'elles viennent du lancement des *kernels* sur les GPU ou des échanges mémoires MPI. On constate ensuite que plus le maillage devient gros plus on a un *speed-up* proche de l'idéal.

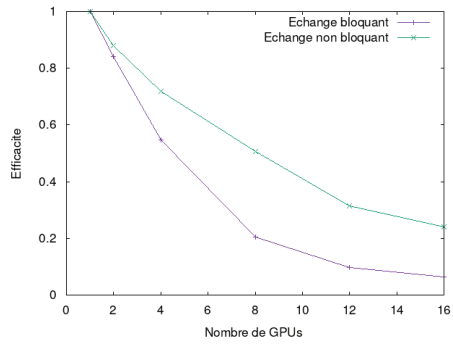
D'après ces résultats on peut déterminer le nombre d'éléments optimal par GPU. On choisit ici de considérer optimale une efficacité supérieure à 80%. Avec ce choix, le nombre optimal d'éléments par GPU semble être entre 300 000 et 500 000 éléments. Cela veut dire que, pour avoir des résultats que l'on considère optimaux, il faut utiliser 1 GPU pour le cas de 400 000 éléments, 4 GPU pour le cas de 1 600 000 éléments, 12 GPU pour le cas de 6 300 000 éléments et finalement 20 GPU pour le cas de 13 000 000 éléments.

Évidement ce choix d'une efficacité supérieur à 80% est arbitraire. On peut vouloir durant la première phase des simulations, au moment où on cherche à avoir la première solution stabilisée, aller le plus vite possible sans se soucier de l'efficacité. Ce n'est pas un problème d'avoir une efficacité faible durant la première phase parce qu'on lance uniquement une seule simulation et on utilise donc peu de ressources sur les grappes de calcul. Par contre durant la deuxième phase, où on construit la base de données des simulations, on va plutôt vouloir une efficacité

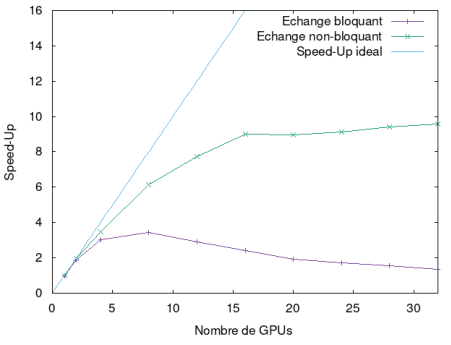
Tableau 7.3 *Speed-Up* et efficacités pour différents types de maillages pour la cas test du *Dam-Break*



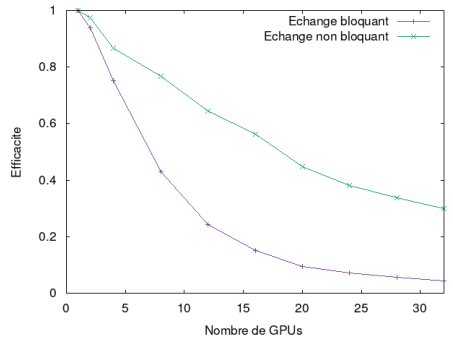
Speed-up, maillage de 400 000 éléments



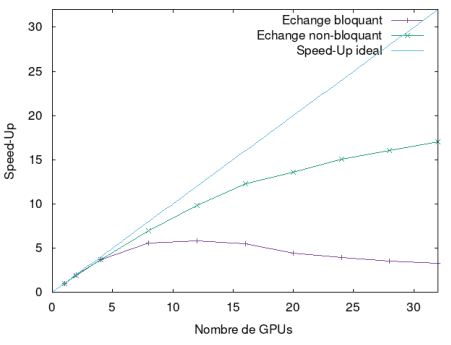
Efficacité, maillage de 400 000 éléments



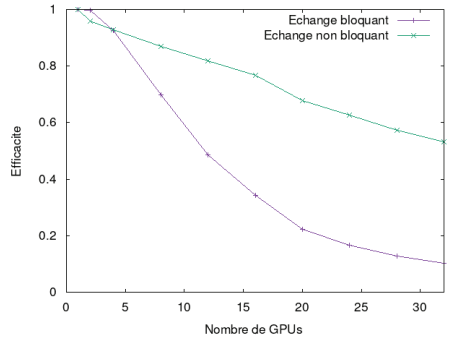
Speed-up, maillage de 1 600 000 éléments



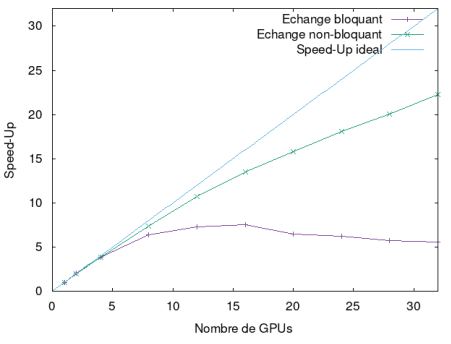
Efficacité, maillage de 1 600 000 éléments



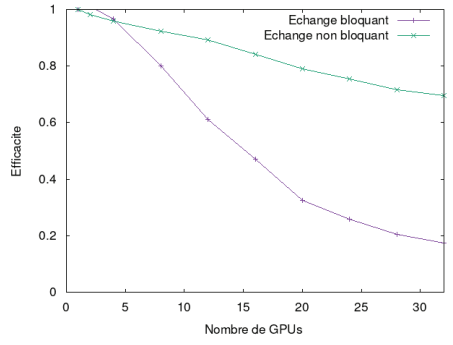
Speed-up, maillage de 6 300 000 éléments



Efficacité, maillage de 6 300 000 éléments



Speed-up, maillage de 13 000 000 éléments



Efficacité, maillage de 13 000 000 éléments

supérieure à 80% étant donné qu'on va lancer des centaines de simulations. Ce sera la phase la plus coûteuse en ressources sur les grappes de calculs.

7.4 Cas de la rivière des Mille Îles

On présente ici des résultats obtenus sur le domaine de la rivière de Mille Îles. Sur la figure 7.2 on peut voir le domaine de simulation superposé sur une image satellite récupérée sur Google Earth Pro. L'image est orientée avec l'axe y vers le nord. L'entrée du domaine est en bas à gauche de l'image et la sortie est en haut à droite.

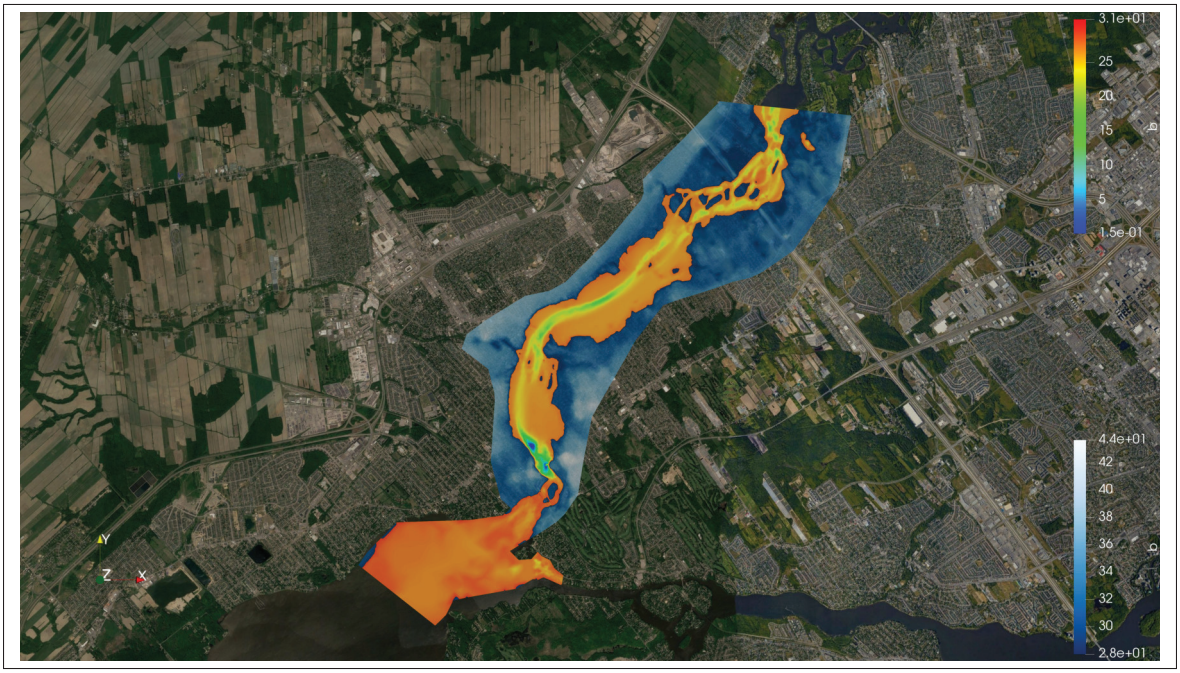


Figure 7.2 Domaine de la Rivière des Mille Îles superposé sur un image satellite générée avec Google Earth Pro.

On peut voir sur la figure 7.3 uniquement la partie du domaine qui est mouillée lorsque le débit d'entrée de la rivière est de 800 m^3/s, ce qui correspond à un débit moyen pour la rivière.

On a plusieurs versions de maillages pour ce domaine qui varient entre 200 000 et 11 millions d'éléments. Le maillage est dans tous les cas raffiné dans les zones critiques, en particulier

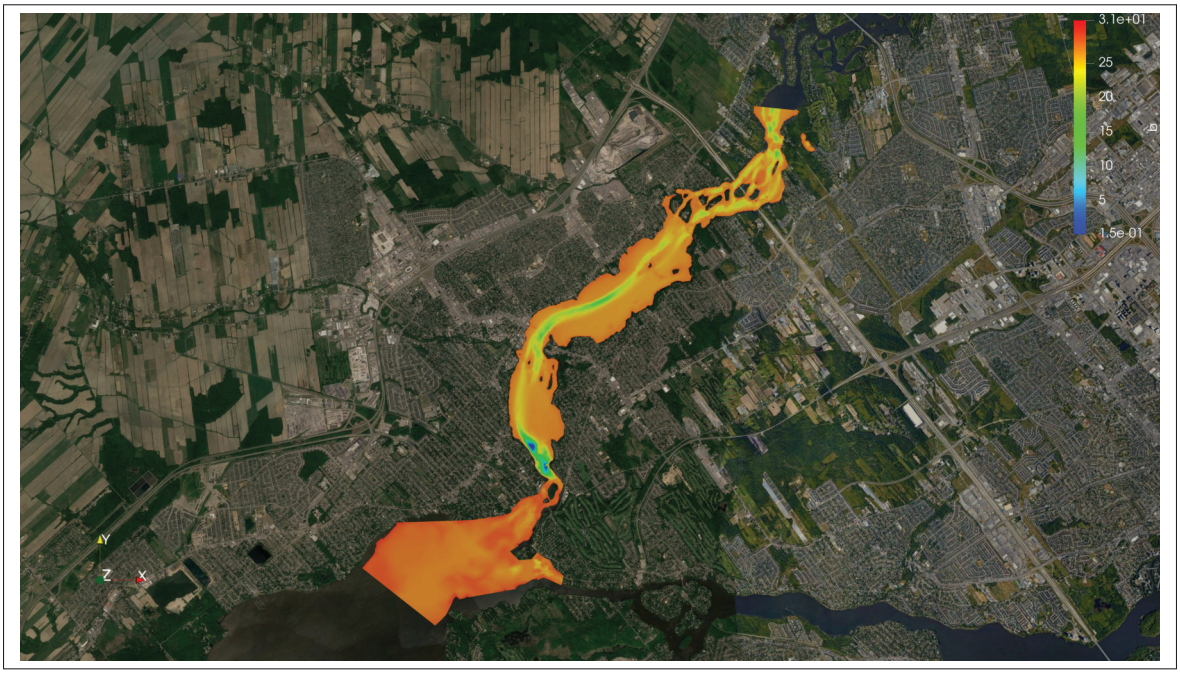


Figure 7.3 Domaine de la Rivière des Mille Îles superposé sur un image satellite générée avec Google Earth Pro., Seules les zones mouillées sont affichées

autour des piles d'un pont qui traverse la rivière. On présente sur la figure 7.4 la zone du barrage sous le premier pont du domaine qui présente un grand raffinement.

7.4.1 Initialisation du domaine et première solution

Les résultats présentés ici sont calculés avec 4 GPU. On présente la décomposition et la bathymétrie du domaine dans le tableau 7.4. On présente ensuite dans le tableau 7.5 les solutions pour différents temps.

Les résultats du tableau 7.5 ont été générés en partant d'une initialisation avec un plan orthogonal à l'axe z. On peut voir sur la première image à $t = 0s$ que l'entrée du domaine n'est pas complètement mouillée et que plus la simulation avance plus le domaine se remplit et plus on voit clairement le courant de la rivière se former.

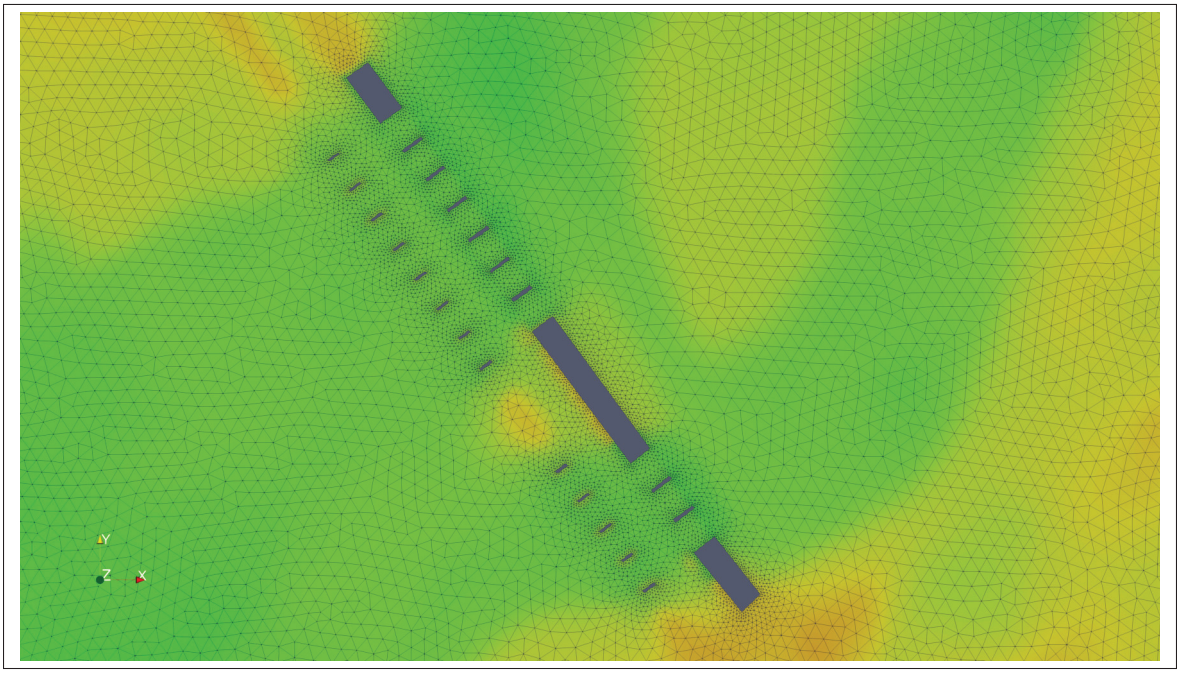


Figure 7.4 Zone de raffinement autour des piles d'un pont dans le domaine de la Rivière des Mille Îles, Maillage de 200 000 éléments.

Tableau 7.4 Domaine des Mille Îles décomposé en 4 sous-domaines à gauche et la bathymétrie à droite

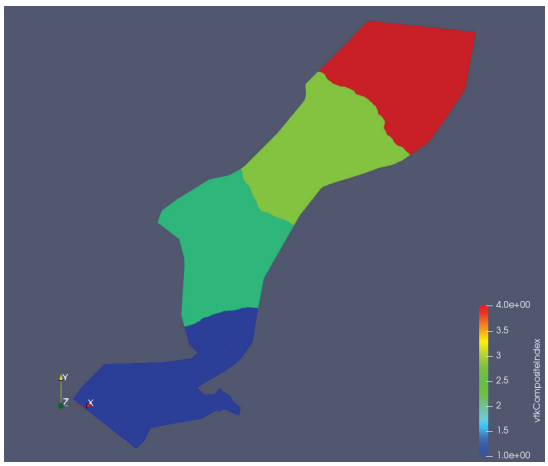
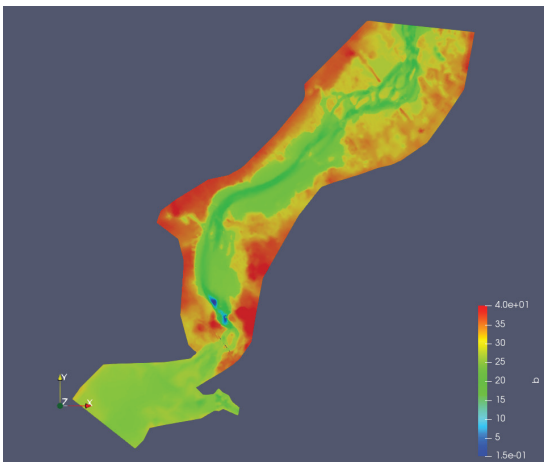
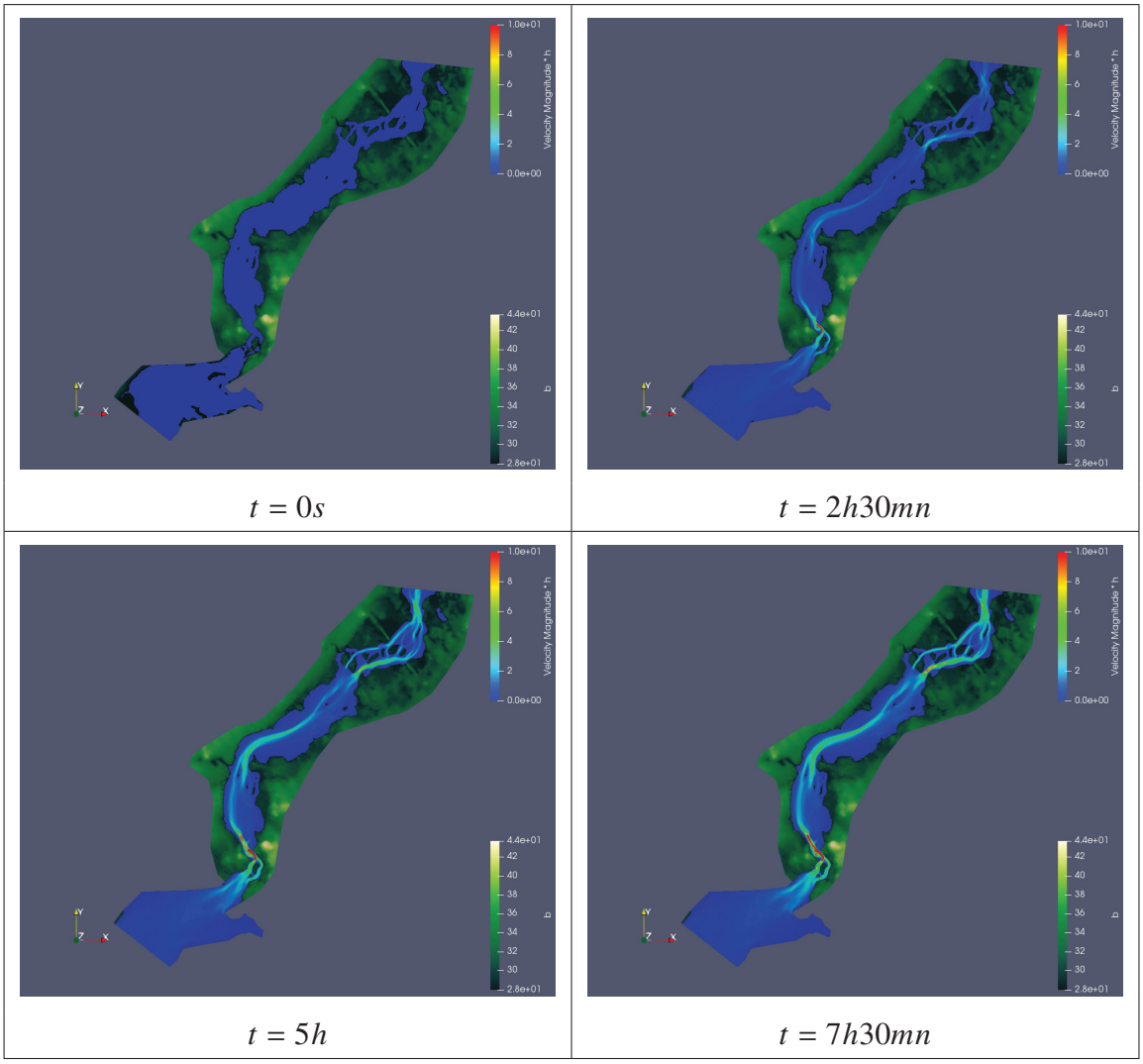
	
Sous-Domaines du maillage	Bathymétrie du domaine

Tableau 7.5 Solutions pour le cas de la rivière des Mille Îles sur un maillage de 740 000 éléments en utilisant 4GPU, domaine sec coloré en fonction de la bathymétrie, domaine mouillé coloré en fonction de $||h\mathbf{V}||_2$



$t = 0s$ $t = 2h30mn$

$t = 5h$ $t = 7h30mn$

En utilisant 4 GPU sur le maillage de 740 000 éléments, on a pu calculer les 7 h 30 min de simulation en 10 min. Évidement ce temps dépend très fortement du raffinement du maillage, plus le maillage est raffiné, plus la distance caractéristique dans le calcul de la condition CFL sera petite et plus il faudra faire d'itérations pour arriver au temps final.

7.4.2 Initialisation en mode bris de barrage fictif

On initialise ici la solution comme un problème de Riemann unidimensionnel. A gauche d'un droite du plan on définit une hauteur d'eau de 30 m et à droite 29 m. Les vitesses initiales sont nulles.

On présente ici les solutions pour différents instants dans le but de montrer l'utilité d'un maillage plus précis. Dans le tableau 7.6, à gauche on a la solution sur un maillage de 740 000 éléments et à droite la solution sur un maillage de 11 millions d'éléments.

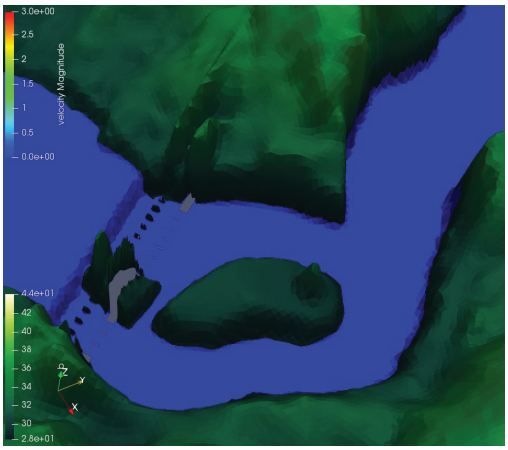
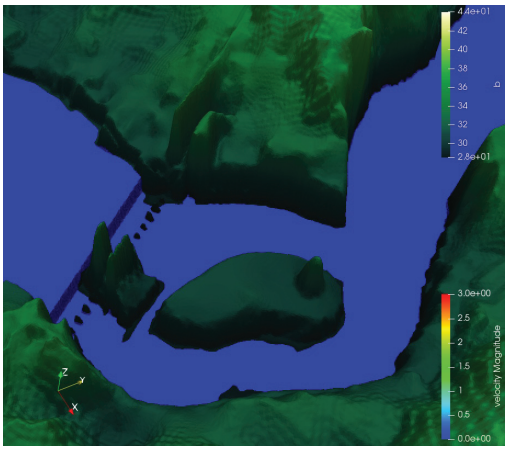
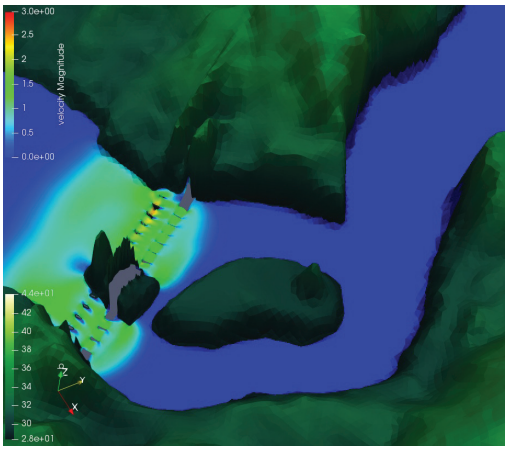
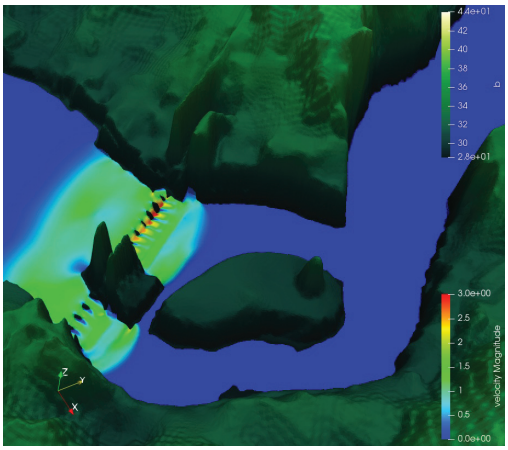
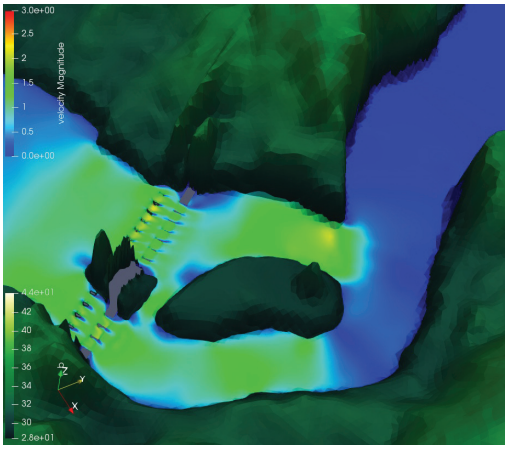
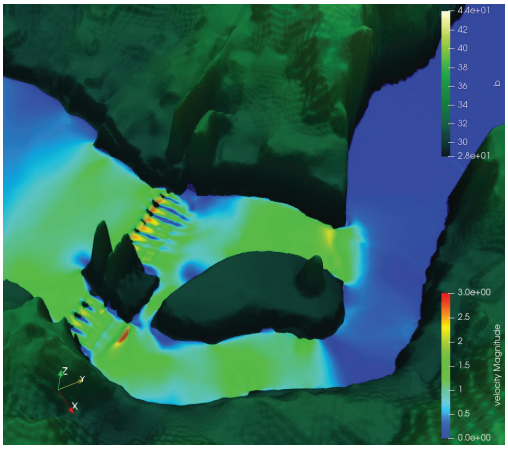
Sur la tableau 7.6 on constate que les solutions sont très proches. Néanmoins, on peut remarquer que la solution est définie plus finement dès l'initialisation où la discontinuité est plus marquée sur le maillage de 11 millions d'éléments. Au bout de 30 s on remarque ici aussi que la solution est plus fine en particulier sur le front de la vague. Finalement au bout de 200 s on constate l'utilité d'un maillage aussi fin comme on peut voir très clairement l'impact des piles du pont sur le cours d'eau alors qu'avec le maillage de 400 000 éléments l'impact est pratiquement invisible.

7.4.3 Visualisation de la ligne d'inondation

Une fois qu'on a généré une solution stable pour un débit moyen, on peut passer à la phase 2 des simulations. On présente sur la figure 7.5 les lignes d'inondation pour un débit de 800 m^3/s et 1100 m^3/s sur le domaine de la rivière des Mille Îles.

En réalité on lance plusieurs centaines de simulations en faisant varier plusieurs paramètres comme le débit ou le nombre de Manning. L'objectif est de construire une base de données pour ensuite faire des études statistiques. On peut trouver une étude de ces analyses dans Abdedou & Soulaïmani (2018) ; Fahsi *et al.* (2010) ; Zokagoa & Soulaïmani (2012) ou encore avec des algorithmes de *machine learning* dans Jacquier *et al.* (2020).

Tableau 7.6 Solutions à différents instants pour un problème de bris de barrage fictif sur la rivière des Mille Îles.

	
$t = 0s$, maillage de 700 000 éléments	$t = 0s$ maillage de 11 000 000 éléments
	
$t = 30s$, maillage de 700 000 éléments	$t = 30s$ maillage de 11 000 000 éléments
	
$t = 120s$, maillage de 700 000 éléments	$t = 120s$ maillage de 11 000 000 éléments

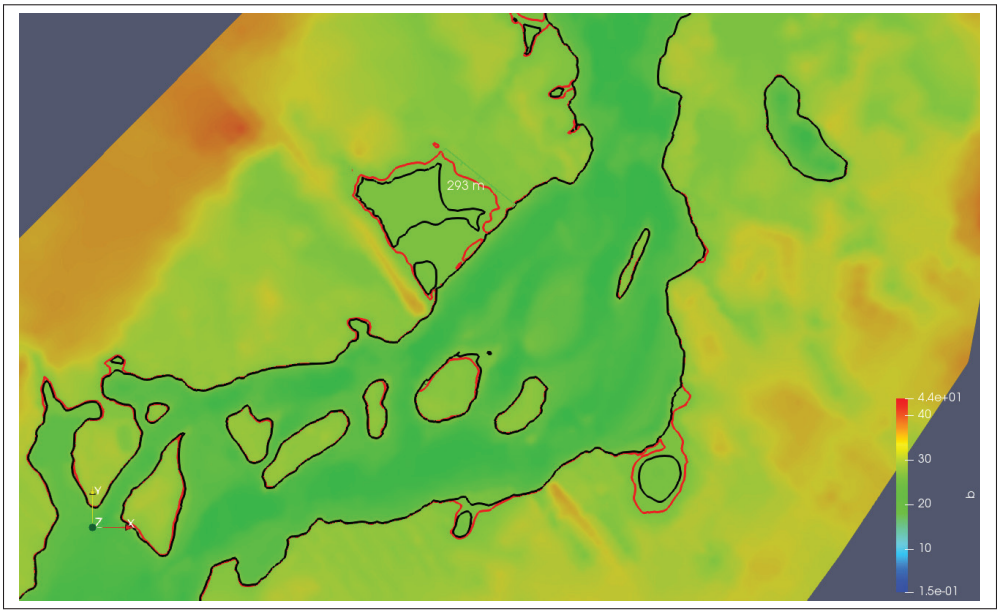


Figure 7.5 Lignes d'inondation pour un débit de 800 m^3/s en noir et 1100 m^3/s en rouge, superposées à la bathymétrie du domaine aval de la rivière des Mille Îles

7.4.4 *Speed-up* et efficacité

On présente dans le tableau 7.7 les *speed-up* et efficacités pour deux versions de maillages du domaine, une version de 740 000 éléments et une version de 11 700 000 éléments.

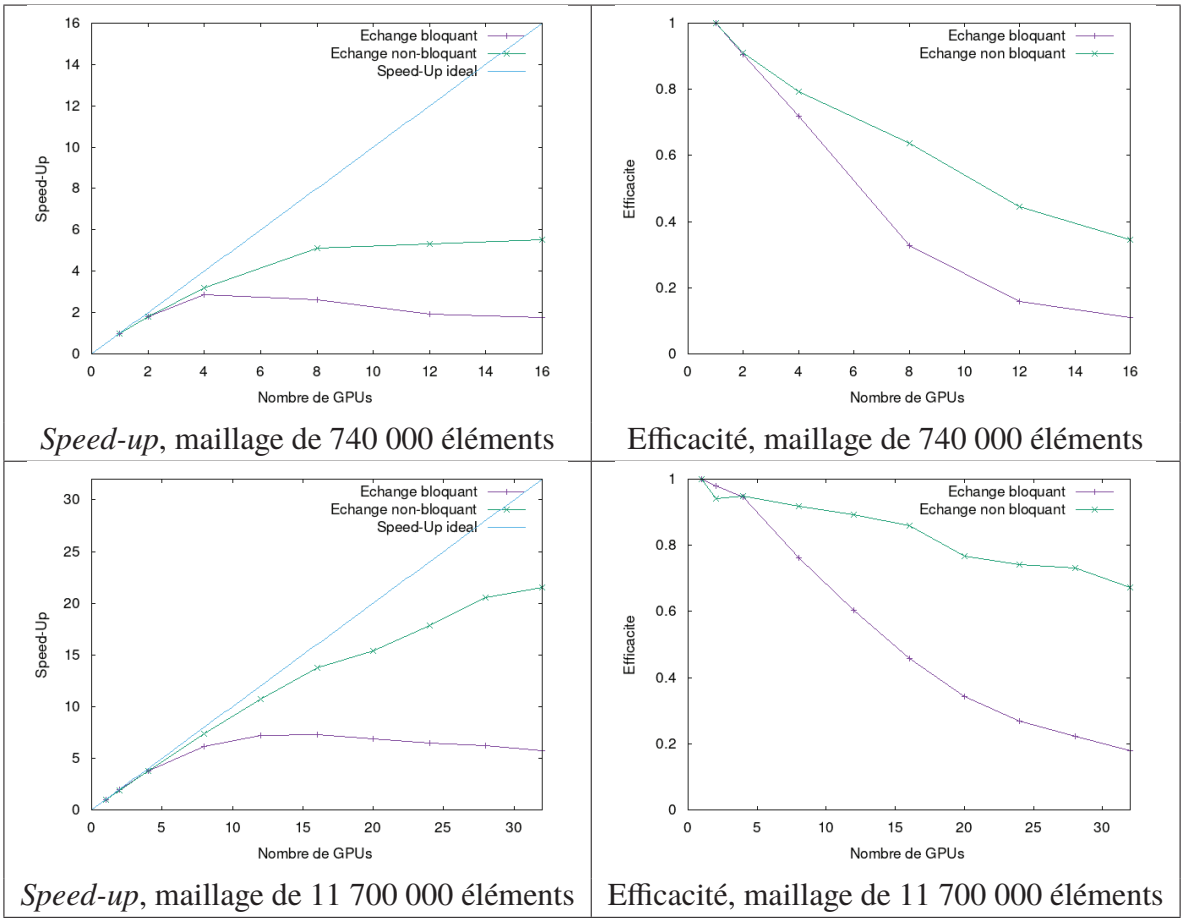
Les résultats qu'on obtient sont assez identiques à ceux de la section 7.3.2. Il semble ici aussi que 300 000 à 500 000 éléments soit un bon compromis pour avoir une bonne efficacité.

7.5 Cas de l'archipel de Montréal

On présente ici un maillage de l'archipel de Montréal qui nous a été gracieusement fourni par la Communauté Métropolitaine de Montréal pour les fins de nos travaux.

Le domaine comprend principalement, la rivière des Mille Îles, la rivière des Prairies et le St-Laurent. Le domaine a donc plusieurs entrées, 7 au total en comptant les petits affluents, et une seule sortie qui correspond au St-Laurent en haut du domaine.

Tableau 7.7 *Speed-Up* et Efficacité pour différents types de maillages pour le cas de la rivière des Milles-Iles



Speed-up, maillage de 740 000 éléments | Efficacité, maillage de 740 000 éléments

Speed-up, maillage de 11 700 000 éléments | Efficacité, maillage de 11 700 000 éléments

On présente sur la figure 7.6 le domaine complet, puis sur la figure 7.6 le domaine mouillé pour une solution stabilisée avec un débit entrant total de $15090 m^3/s$ qui correspond à un débit moyen pour ce domaine.

7.5.1 Solutions

Les résultats présentés ici sont calculés avec 4 GPU, on présente la décomposition et la bathymétrie du domaine dans le tableau 7.8. On présente ensuite dans le tableau 7.9 les solutions pour différents temps.

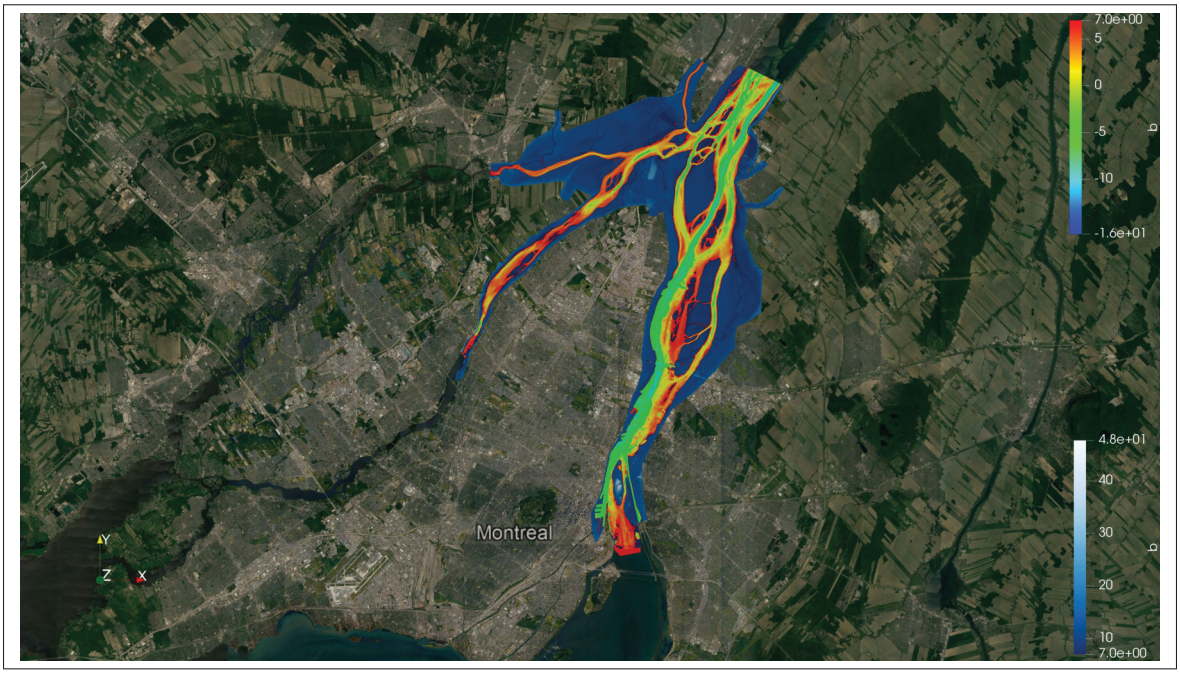
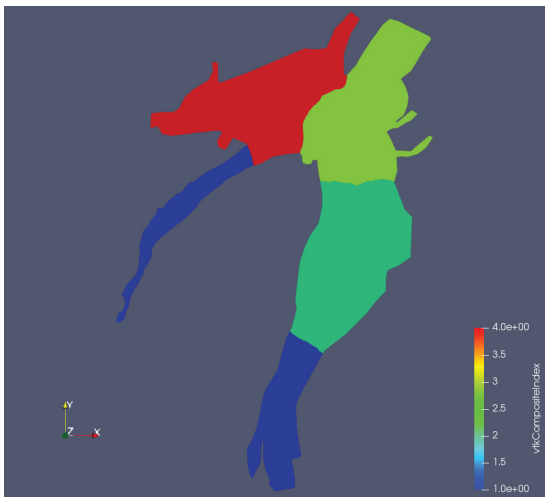
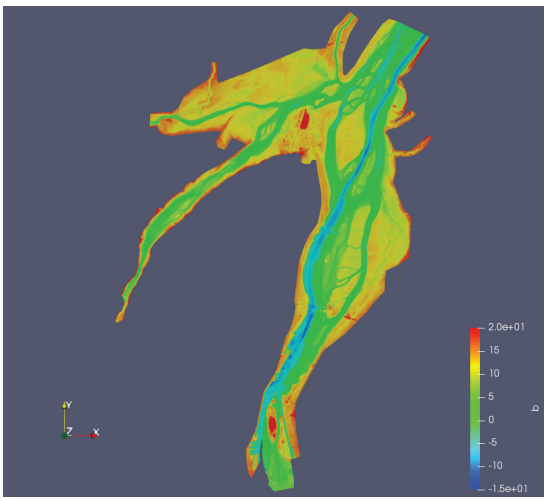


Figure 7.6 Domaine de l'archipel de Montréal superposé sur un image satellite générée avec Google Earth Pro.

Tableau 7.8 Domaine de l'archipel de Montréal décomposé en 4 sous-domaines à gauche et la bathymétrie à droite

	
Sous-Domaines du maillage	Bathymétrie du domaine

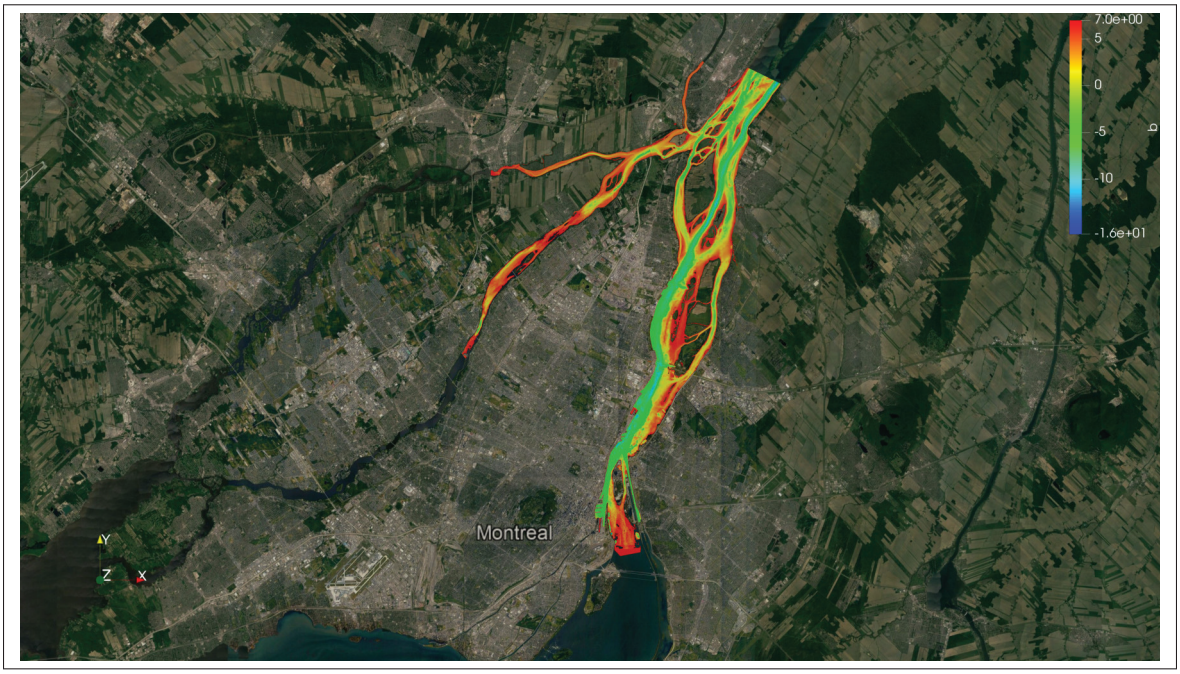


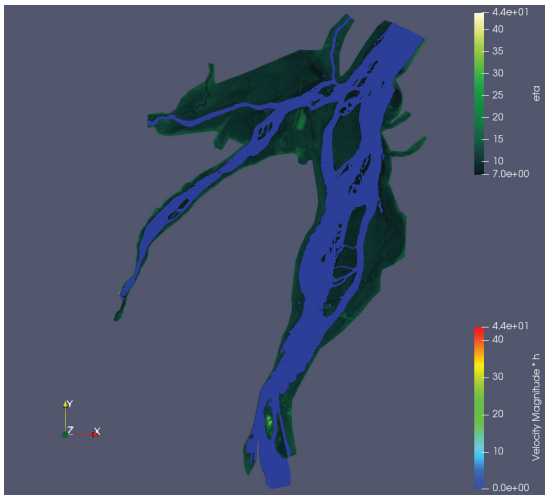
Figure 7.7 Domaine de la Rivière des Mille Îles superposé sur un image satellite générée avec Google Earth Pro., Seules les zones mouillées sont affichées

Les résultats du tableau 7.5 ont été générés en partant d'une initialisation avec un plan incliné. Prendre un plan incliné plutôt qu'un plan orthogonal à l'axe z permet d'avoir une initialisation plus rapide comme il est alors plus facile pour le fluide de prendre de la vitesse dans direction aval. On ne montre sur le tableau 7.9 que les solutions pour des temps faibles, il faut attendre $11h$ de simulation pour avoir une solution au régime permanent. Il a fallu $1h$ de calcul sur 4 GPU pour calculer ces $11h$ de simulation.

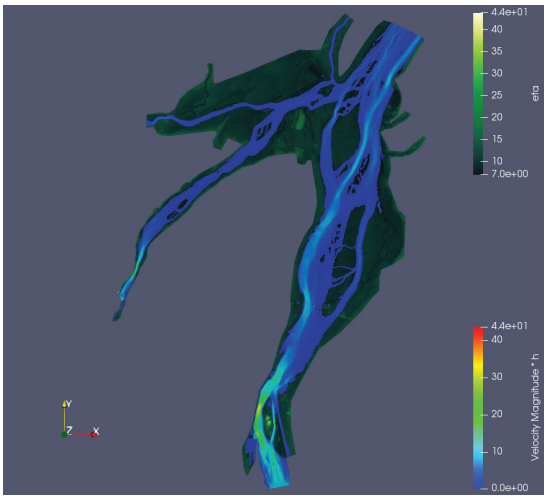
Pour l'instant nous sommes toujours dans la phase de validation du maillage en lançant des simulations sur des cas connus et en comparant les résultats à des relevés réels. L'objectif est de modifier la bathymétrie du maillage ainsi que de calibrer des paramètres tels que le nombre de Manning pour avoir les meilleurs résultats possibles.

On a pour l'instant une seule version de ce maillage avec 690 000 éléments. On imagine que lorsque la phase de calibration sera finie, un maillage bien plus fin sera créé.

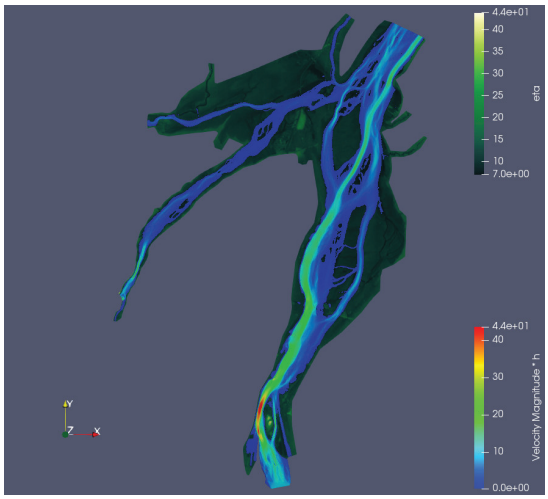
Tableau 7.9 Solutions pour le cas de la rivière de l'archipel de Montréal sur un maillage de 690 000 éléments en utilisant 4 GPU, domaine sec coloré en fonction de la bathymétrie, domaine mouillé coloré en fonction de $||h\mathbf{V}||_2$



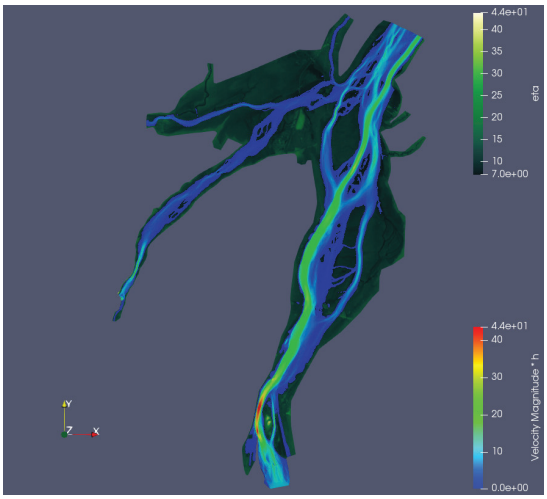
$t = 0s$



$t = 15mn$



$t = 30mn$



$t = 45mn$

Dans le futur on voudrait avoir un maillage de ce domaine avec bien plus d'éléments, par exemple de 10 à 100 millions, pour pouvoir tester les performances du code multi-GPU.

CONCLUSION ET RECOMMANDATIONS

Nous avons présenté dans ce mémoire les différentes étapes qui nous ont permis de porter le code **CUTEFLOW** pour la résolution des équations de St-Venant, sur une architecture multi-GPU.

Nous avons commencé par rappeler la définition mathématique des équations de St-Venant et leur résolution par des méthodes volumes finis. Une attention particulière a été portée aux solveurs de Riemann utilisés lors de cette étape. Plusieurs travaux ont été combinés pour obtenir les meilleurs résultats possibles.

Nous avons ensuite expliqué les principes fondamentaux de la programmation GPU. Nous avons détaillé le fonctionnement d'une telle programmation en CUDA-Fortran et nous l'avons appliquée sur un cas simple.

Une fois la présentation de la programmation GPU faite, nous sommes entrés dans le processus du portage sur une architecture multi-GPU en utilisant MPI et CUDA-Fortran. Nous avons donc commencé par expliquer le traitement de la décomposition de domaine en utilisant la bibliothèque METIS. Lors de cette étape une attention particulière a dû être portée sur la numérotation des mailles à envoyer et recevoir par chaque sous-domaine en anticipation des échanges mémoires MPI.

L'utilisation de MPI et en particulier d'une version CUDA-*Aware* d'OpenMPI a ensuite été faite. Une discussion sur l'importance d'une telle version, mais aussi sur la difficulté d'en trouver une compatible avec nos compilateurs a été présentée.

Finalement, après avoir présenté le fonctionnement général du code **CUTEFLOW** dans sa version multi-GPU, des résultats pour différents cas ont été présentés.

Lors de ce travail, nous avons constaté que porter un code sur une architecture multi-GPU, en particulier, en utilisant MPI et CUDA Fortran, demande beaucoup de temps et d'optimisation.

Non seulement, nous avons été confrontés aux problèmes classiques de la programmation en MPI mais en plus des problèmes dus au couplage avec CUDA-Fortran sont apparus. La simple utilisation des compilateurs a nécessité une attention particulière et a souvent demandé de longs échanges avec le support des grappes de calculs pour pouvoir compiler le code.

Malgré ces problèmes, nous avons pu noter des efficacités supérieures à 80% pour des maillages de 300 000 à 500 000 éléments par GPU. Cela signifie que la taille du maillage que l'on veut traiter ne dépend maintenant que du nombre de GPU disponibles. Malgré cela, jusqu'à présent, des maillages de 500 000 à 1 000 000 d'éléments étaient classiquement utilisés pour faire les études d'inondations dans les rivières. Il faut donc attendre que les possibilités de codes multi-GPU tels que **CUTEFLOW** ouvrent la porte à la création de maillages bien plus grands et avec un nombre d'éléments bien plus importants. Dans ce travail nous avons au maximum traité un maillage de 11 millions d'éléments pour le domaine de la rivière des Mille Îles.

Perspectives pour de futures recherches

À partir du travail réalisé, on peut faire les suggestions futures suivantes :

- Optimiser l'échange mémoire MPI dans **CUTEFLOW**, par exemple, en activant des fonctionnalités comme le GPU-Direct RDMA sur les grappes de calculs.
- Tester la robustesse de **CUTEFLOW** sur des maillages bien plus importants qui ont une utilité pratique comme le domaine de l'archipel de Montréal.
- Développer des méthodes de post-traitements capables de traiter les quantités de données générées par des codes tels que **CUTEFLOW**, en particulier pour les plus gros maillages.

ANNEXE I

CALCUL DE L'INTÉGRATION SUR LA HAUTEUR D'EAU DES ÉQUATIONS DU MOUVEMENT

On développe ici l'intégration des équations de conservation de la quantité de mouvement sur la hauteur d'eau. La démonstration qu'on présente s'appuie en partie sur Dawson & Mirabito (2008).

On rappelle qu'on a les deux équations suivantes,

$$u_t + uu_x + vu_y + wu_z = -gs_x + \frac{1}{\rho}\sigma_{11x} + \frac{1}{\rho}\sigma_{12y} + \frac{1}{\rho}\sigma_{13z} \tag{A I-1}$$

et,

$$v_t + uv_x + vv_y + wv_z = -gs_y + \frac{1}{\rho}\sigma_{21x} + \frac{1}{\rho}\sigma_{22y} + \frac{1}{\rho}\sigma_{23z} \tag{A I-2}$$

En utilisant l'équation de continuité, $\nabla.\mathbf{V} = u_x + v_y + w_z = 0$, on peut réécrire ces équations sous la forme,

$$u_t + (u^2)_x + (vu)_y + (wu)_z = -gs_x + \frac{1}{\rho}\sigma_{11x} + \frac{1}{\rho}\sigma_{12y} + \frac{1}{\rho}\sigma_{13z} \tag{A I-3}$$

et,

$$v_t + (uv)_x + (v^2)_y + (wv)_z = -gs_y + \frac{1}{\rho}\sigma_{21x} + \frac{1}{\rho}\sigma_{22y} + \frac{1}{\rho}\sigma_{23z} \tag{A I-4}$$

On rappelle aussi la formule de Leibniz,

$$\frac{d}{d\alpha}\int_{\xi_1(\alpha)}^{\xi_2(\alpha)} f(\xi,\alpha)d\xi = \int_{\xi_1(\alpha)}^{\xi_2(\alpha)} \frac{\partial f}{\partial \alpha}d\xi + \frac{d\xi_2}{d\alpha}f(\xi_2,\alpha) - \frac{d\xi_1}{d\alpha}f(\xi_1,\alpha) \tag{A I-5}$$

On utilisera la notation $\bar{u}$ pour décrire la variable u moyennée sur la hauteur d'eau, soit, $\bar{u} = \frac{1}{h}\int_b^s u dz$.

1. Terme de gauche des équations du mouvement

On intègre le terme de gauche de A I-3 sur la hauteur d'eau,

$$\text{LHS} = \int_b^s \left[u_t + (u^2)_x + (vu)_y + (wu)_z\right] dz \tag{A I-6}$$

$$\text{LHS} = \int_b^s u_t dz + \int_b^s (u^2)_x dz + \int_b^s (vu)_y dz + \int_b^s (wu)_z dz \tag{A I-7}$$

en utilisant le théorème de Leibniz, on peut calculer chacun des termes de A I-7,

$$\int_b^s u_t dz = \frac{\partial}{\partial t}\int_b^s u dz - u|_s s_t \tag{A I-8}$$

$$\int_b^s (u^2)_x dz = \frac{\partial}{\partial x}\int_b^s u^2 dz - u^2|_{z=s} s_x + u^2|_{z=b} b_x \tag{A I-9}$$

$$\int_b^s (uv)_y dz = \frac{\partial}{\partial y}\int_b^s uv dz - (uv)|_{z=s} s_y + (uv)|_{z=b} b_y \tag{A I-10}$$

$$\int_b^s (uw)_z dz = (uw)|_{z=s} - (uw)|_{z=b} \tag{A I-11}$$

On rappelle les expressions de $w_{z=s}$ et $w_{z=b}$ 2.23 et 2.24 déterminées en utilisant les conditions aux limites,

$$w_{z=s} = s_t + u_{z=s} s_x + v_{z=s} s_y \tag{A I-12}$$

et

$$w_{z=b} = u_{z=b} b_x + v_{z=b} b_y \tag{A I-13}$$

En remplaçant les expressions de $w_{z=s}$ et $w_{z=b}$ et en regroupant tous les termes calculés, on obtient pour A I-7,

$$\text{LHS} = \frac{\partial}{\partial t}\int_b^s u dz + \frac{\partial}{\partial x}\int_b^s u^2 dz + \frac{\partial}{\partial y}\int_b^s vu dz \qquad \text{(A I-14)}$$

On peut ensuite utiliser les valeurs moyennées pour avoir,

$$\text{LHS} = (h\bar{u})_t + (h\bar{u}^2)_x + (h\bar{u}\bar{v})_y + \{\text{termes disp.}\} \qquad \text{(A I-15)}$$

Où les termes de dispersion compensent le fait que la moyenne du produit n'est pas le produit des moyennes.

On obtient un résultat similaire pour la seconde équation du mouvement A I-4, ou en prenant,

$$\text{LHS} = \int_b^s \left[v_t + (uv)_x + (v^2)_y + (wv)_z\right] dz \qquad \text{(A I-16)}$$

on obtient,

$$\text{LHS} = (h\bar{v})_t + (h\bar{u}\bar{v})_x + (h\bar{v}^2)_y + \{\text{termes disp.}\} \qquad \text{(A I-17)}$$

2. Terme de droite des équations du mouvement

On intègre le terme de droite de A I-3 sur la hauteur d'eau,

$$\text{RHS} = \int_b^s \left[-gs_x + \frac{1}{\rho}\sigma_{11x} + \frac{1}{\rho}\sigma_{12y} + \frac{1}{\rho}\sigma_{13z}\right] dz \qquad \text{(A I-18)}$$

On explicite le calcul de chacun des termes,

$$\int_b^s -gs_x dz = -ghs_x \qquad \text{(A I-19)}$$

$$\int_b^s \sigma_{11x} dz = \frac{\partial}{\partial x} \int_b^s \sigma_{11} dz - \sigma_{11}|_{z=s} s_x + \sigma_{11}|_{z=b} b_x \tag{A I-20}$$

$$\int_b^s \sigma_{12y} dz = \frac{\partial}{\partial y} \int_b^s \sigma_{12} dz - \sigma_{12}|_{z=s} s_y + \sigma_{12}|_{z=b} b_y \tag{A I-21}$$

$$\int_b^s \sigma_{13z} dz = \sigma_{13}|_{z=s} - \sigma_{13}|_{z=b} \tag{A I-22}$$

On rappelle la définition de σ_b et σ_s données en 2.10 et 2.11,

$$\begin{aligned} \sigma_{b1} &= \sigma_{11}|_{z=b} b_x + \sigma_{12}|_{z=b} b_y - \sigma_{13}|_{z=b} \\ \sigma_{b2} &= \sigma_{21}|_{z=b} b_x + \sigma_{22}|_{z=b} b_y - \sigma_{23}|_{z=b} \end{aligned} \tag{A I-23}$$

et à la surface,

$$\begin{aligned} \sigma_{s1} &= -\sigma_{11}|_{z=s} s_x - \sigma_{12}|_{z=s} s_y + \sigma_{13}|_{z=s} \\ \sigma_{s2} &= -\sigma_{21}|_{z=s} s_x - \sigma_{22}|_{z=s} s_y + \sigma_{23}|_{z=s} \end{aligned} \tag{A I-24}$$

En utilisant ces définitions et en remplaçant tous les termes dans A I-18, on obtient,

$$\text{RHS} = -ghs_x + \frac{1}{\rho} \left[\frac{\partial}{\partial x} \int_b^s \sigma_{11} dz + \frac{\partial}{\partial y} \int_b^s \sigma_{12} dz + \sigma_{s1} - \sigma_{b1} \right] \tag{A I-25}$$

en utilisant les valeurs moyennées on obtient,

$$\text{RHS} = -ghs_x + \frac{1}{\rho} \left[(h\bar{\sigma}_{11})_x + (h\bar{\sigma}_{12})_y + \sigma_{s1} - \sigma_{b1} \right] \tag{A I-26}$$

On obtient un résultat similaire pour la seconde équation du mouvement, avec

$$\text{RHS} = \int_b^s \left[-g s_y + \frac{1}{\rho}\sigma_{21x} + \frac{1}{\rho}\sigma_{22y} + \frac{1}{\rho}\sigma_{23z} \right] dz \tag{A I-27}$$

on a,

$$\text{RHS} = -ghs_y + \frac{1}{\rho}\left[(h\bar{\sigma}_{21})_x + (h\bar{\sigma}_{22})_y + \sigma_{s2} - \sigma_{b2} \right] \tag{A I-28}$$

Finalement on obtient la forme finale des deux équations du mouvement intégrées sur la hauteur d'eau,

$$(h\bar{u})_t + (h\bar{u}^2)_x + (h\bar{u}\bar{v})_y + \{\text{termes dips.}\} = -ghs_x + \frac{1}{\rho}\left[(h\bar{\sigma}_{11})_x + (h\bar{\sigma}_{12})_y + \sigma_{s1} - \sigma_{b1} \right] \tag{A I-29}$$

et,

$$(h\bar{v})_t + (h\bar{v}\bar{v})_x + (h\bar{v}^2)_y + \{\text{termes disp.}\} = -ghs_y + \frac{1}{\rho}\left[(h\bar{\sigma}_{21})_x + (h\bar{\sigma}_{22})_y + \sigma_{s2} - \sigma_{b2} \right] \tag{A I-30}$$

Dans ce travail on négligera les termes de dispersion ainsi que les contraintes σ_{11}, σ_{12} et σ_{22} ce qui donne,

$$(h\bar{u})_t + (h\bar{u}^2)_x + (h\bar{u}\bar{v})_y = -ghs_x + \frac{1}{\rho}\left[\sigma_{s1} - \sigma_{b1} \right] \tag{A I-31}$$

et,

$$(h\bar{v})_t + (h\bar{v}\bar{v})_x + (h\bar{v}^2)_y = -ghs_y + \frac{1}{\rho}\left[\sigma_{s2} - \sigma_{b2} \right] \tag{A I-32}$$

En réarrangeant les équations on obtient bien le résultat annoncé en 2.32 et 2.33,

$$(h\bar{u})_t + (h\bar{u}^2 + \frac{1}{2}gh^2)_x + (h\bar{u}\bar{v})_y = -ghb_x + \frac{1}{\rho}\left[\sigma_{s1} - \sigma_{b1}\right], \qquad \text{(A I-33)}$$

et,

$$(h\bar{v})_t + (h\bar{u}\bar{v})_x + (h\bar{v}^2 + \frac{1}{2}gh^2)_y = -ghb_y + \frac{1}{\rho}\left[\sigma_{s2} - \sigma_{b2}\right]. \qquad \text{(A I-34)}$$

ANNEXE II

CALCUL DE LA MATRICE JACOBIENNE DU TERME DE FRICTION

On développe ici le calcul de 2.70. On rappelle qu'on a,

$$S_f = (0, -gh\frac{n^2 u\sqrt{u^2+v^2}}{h^{4/3}}, -gh\frac{n^2 v\sqrt{u^2+v^2}}{h^{4/3}})$$

en posant $q_x = hu, q_y = hv$ on obtient,

$$S_f = (0, -g\frac{n^2 q_x\sqrt{q_x^2+q_y^2}}{h^{7/3}}, -g\frac{n^2 q_y\sqrt{q_x^2+q_y^2}}{h^{7/3}})$$

On peut ensuite reprendre 2.70 avec ces nouvelles variables,

$$J_f = \frac{\partial S_f}{\partial U} = \begin{pmatrix} 0 & 0 & 0 \\ \partial S_{fx}/\partial h & \partial S_{fx}/\partial q_x & \partial S_{fx}/\partial q_y \\ \partial S_{fy}/\partial h & \partial S_{fy}/\partial q_x & \partial S_{fy}/\partial q_y \end{pmatrix} \qquad \text{(A II-1)}$$

et calculer les différents termes,

$$\begin{aligned} \partial S_{fx}/\partial h &= \frac{7}{3}gn^2\frac{q_x\sqrt{q_x^2+q_y^2}}{h^{10/3}} = \frac{7}{3}gn^2\frac{u\sqrt{u^2+v^2}}{h^{4/3}} = \frac{7}{3}Cu\sqrt{u^2+v^2} \\ \partial S_{fy}/\partial h &= \frac{7}{3}gn^2\frac{q_y\sqrt{q_x^2+q_y^2}}{h^{10/3}} = \frac{7}{3}gn^2\frac{v\sqrt{u^2+v^2}}{h^{4/3}} = \frac{7}{3}Cv\sqrt{u^2+v^2} \end{aligned}$$

avec $C = gn^2/h^{4/3}$.

ensuite,

$$\partial S_{fx}/\partial q_x = \frac{C}{h^2}\left[\sqrt{q_x^2+q_y^2}+\frac{q_x^2}{\sqrt{q_x^2+q_y^2}}\right] = \frac{C}{h^2}\frac{2q_x^2+q_y^2}{\sqrt{q_x^2+q_y^2}} = \frac{C}{h}\frac{2u^2+v^2}{\sqrt{u^2+v^2}}$$
$$\partial S_{fy}/\partial q_x = \frac{C}{h^2}\frac{q_x q_y}{\sqrt{q_x^2+q_y^2}} = \frac{C}{h}\frac{uv}{\sqrt{u^2+v^2}}$$

et finalement,

$$\partial S_{fx}/\partial q_y = \frac{C}{h^2}\frac{q_x q_y}{\sqrt{q_x^2+q_y^2}} = \frac{C}{h}\frac{uv}{\sqrt{u^2+v^2}}$$
$$\partial S_{fy}/\partial q_y = \frac{C}{h^2}\left[\sqrt{q_x^2+q_y^2}+\frac{q_y^2}{\sqrt{q_x^2+q_y^2}}\right] = \frac{C}{h^2}\frac{q_x^2+2q_y^2}{\sqrt{q_x^2+q_y^2}} = \frac{C}{h}\frac{u^2+2v^2}{\sqrt{u^2+v^2}}$$

On peut enfin réécrire A II-1,

$$J_f = \frac{\partial S_f}{\partial U} = \begin{pmatrix} 0 & 0 & 0 \\ \frac{7}{3}Cu\sqrt{u^2+v^2} & \frac{C}{h}\frac{2u^2+v^2}{\sqrt{u^2+v^2}} & \frac{C}{h}\frac{uv}{\sqrt{u^2+v^2}} \\ \frac{7}{3}Cv\sqrt{u^2+v^2} & \frac{C}{h}\frac{uv}{\sqrt{u^2+v^2}} & \frac{C}{h}\frac{u^2+2v^2}{\sqrt{u^2+v^2}} \end{pmatrix} \qquad \text{(A II-2)}$$

avec $C = gn^2/h^{4/3}$.

RÉFÉRENCES

OpenMPI Website : Running CUDA-Aware OpenMPI. Repéré à https://www.open-mpi.org/faq/?category=runcuda.

Abdedou, A. & Soulaïmani, A. (2018). A non-intrusive B-splines Bézier elements-based method for uncertainty propagation. *Computer Methods in Applied Mechanics and Engineering*, 345. doi : 10.1016/j.cma.2018.10.047.

Aissa, M., Verstraete, T. & Vuik, C. (2017). Toward a GPU-aware comparison of explicit and implicit CFD simulations on structured meshes. *Computers & Mathematics with Applications*, 74(1), 201 - 217. doi : https://doi.org/10.1016/j.camwa.2017.03.003. 5th European Seminar on Computing ESCO 2016.

Ata, R. & Soulaïmani, A. (2005a). A stabilized SPH method for inviscid shallow water flows. *International Journal for Numerical Methods in Fluids*, 47(2), 139-159. doi : 10.1002/fld.801.

Ata, R. & Soulaïmani, A. (2005b). A stabilized SPH method for inviscid shallow water flows. *International Journal for Numerical Methods in Fluids*, 47(2), 139-159. doi : 10.1002/fld.801.

Ata, R., Pavan, S., Khelladi, S. & Toro, E. F. (2013). A Weighted Average Flux (WAF) scheme applied to shallow water equations for real-life applications. *Advances in Water Resources*, 62, 155 - 172. doi : https://doi.org/10.1016/j.advwatres.2013.09.019.

Audusse, E. & Bristeau, M.-O. (2005). A well-balanced positivity preserving "second-order" scheme for shallow water flows on unstructured meshes. *Journal of Computational Physics*, 206(1), 311 - 333. doi : https://doi.org/10.1016/j.jcp.2004.12.016.

Audusse, E., Bouchut, F., Bristeau, M.-O., Klein, R. & Perthame, B. (2004a). A Fast and Stable Well-Balanced Scheme with Hydrostatic Reconstruction for Shallow Water Flows. *SIAM Journal on Scientific Computing*, 25(6), 2050-2065. doi : 10.1137/S1064827503431090.

Audusse, E., Bouchut, F., Bristeau, M.-O., Klein, R. & Perthame, B. (2004b). A Fast and Stable Well-Balanced Scheme with Hydrostatic Reconstruction for Shallow Water Flows. *SIAM Journal on Scientific Computing*, 25(6), 2050-2065. doi : 10.1137/S1064827503431090.

Ayachit, U. (2015). *The ParaView Guide : A Parallel Visualization Application*. Clifton Park, NY, USA : Kitware, Inc.

Bradford, S. F. & Sanders, B. F. (2002). Finite-Volume Model for Shallow-Water Flooding of Arbitrary Topography. *Journal of Hydraulic Engineering*, 128(3), 289-298. doi : 10.1061/(ASCE)0733-9429(2002)128:3(289).

Brodtkorb, A. R., Hagen, T. R., Lie, K.-A. & Natvig, J. R. (2010). Simulation and visualization of the Saint-Venant system using GPUs. *Computing and Visualization in Science*, 13(7), 341-353. doi : 10.1007/s00791-010-0149-x.

Brodtkorb, A., Sætra, M. & Altinakar, M. (2011). Efficient shallow water simulations on GPUs : Implementation, visualization, verification, and validation. *Computers & Fluids - COMPUT FLUIDS*, 55. doi : 10.1016/j.compfluid.2011.10.012.

Brodtkorb, A. R., Sætra, M. L. & Altinakar, M. (2012). Efficient shallow water simulations on GPUs : Implementation, visualization, verification, and validation. *Computers & Fluids*, 55, 1 - 12. doi : https://doi.org/10.1016/j.compfluid.2011.10.012.

Brufau, P., García-Navarro, P. & Vázquez-Cendón, M. E. (2004). Zero mass error using unsteady wetting–drying conditions in shallow flows over dry irregular topography. *International Journal for Numerical Methods in Fluids*, 45(10), 1047-1082. doi : 10.1002/fld.729.

García-Rodríguez, González-Vida, Macías, Parés & Vázquez-Cendón]]CASTRO Castro, M., Ferreiro], A. F., García-Rodríguez, J., González-Vida, J., Macías, J., Parés, C. & Vázquez-Cendón], M. E. (2005). The numerical treatment of wet/dry fronts in shallow flows : application to one-layer and two-layer systems. *Mathematical and Computer Modelling*, 42(3), 419 - 439. doi : https://doi.org/10.1016/j.mcm.2004.01.016.

Cosnard, M., Nivat, M. & Robert, Y. (1992). *Algorithmique parallèle*.

Cuthill, E. & McKee, J. (1969). Reducing the Bandwidth of Sparse Symmetric Matrices. *Proceedings of the 1969 24th National Conference*, (ACM '69), 157–172. doi : 10.1145/800195.805928.

Dawson, C. & Mirabito, C. M. (2008). The shallow water equations.

De la Asunción, M., Mantas, J. M. & Castro, M. J. (2010). Programming CUDA-Based GPUs to Simulate Two-Layer Shallow Water Flows. *Euro-Par 2010 - Parallel Processing*, pp. 353–364.

De la Asunción, M., Castro, M. J., Fernández-Nieto, E., Mantas, J. M., Acosta, S. O. & González-Vida, J. M. (2013). Efficient GPU implementation of a two waves TVD-WAF method for the two-dimensional one layer shallow water system on structured meshes. *Computers & Fluids*, 80, 441 - 452. doi : https://doi.org/10.1016/j.compfluid.2012.01.012. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

Dhatt, G., Soulaimani, A., Ouellet, Y. & Fortin, M. (1986). Development of new triangular elements for free surface flows. *International Journal for Numerical Methods in Fluids*, 6, 895 - 911. doi : 10.1002/fld.1650061204.

Elkadri E, N., Soulaïmani, A. & Deschênes, C. (2000). A finite element formulation of compressible flows using various sets of independent variables. *Computer Methods in Applied Mechanics and Engineering*, 181(1), 161 - 189. doi : https://doi.org/10.1016/S0045-7825(99)00069-9.

Elsen, E., LeGresley, P. & Darve, E. (2008). Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics*, 227(24), 10148 - 10161. doi : https://doi.org/10.1016/j.jcp.2008.08.023.

Escalante, C., Morales de Luna, T. & Castro, M. (2018). Non-hydrostatic pressure shallow flows : GPU implementation using finite volume and finite difference scheme. *Applied Mathematics and Computation*, 338, 631 - 659. doi : https://doi.org/10.1016/j.amc.2018.06.035.

Fahsi, A., Soulaïmani, A. & Tchamen, G. (2010). Application of reliability techniques for the estimation of uncertainties in fluvial hydraulics simulations. *Canadian Journal of Civil Engineering*, 37, 991-1002. doi : 10.1139/L10-011.

Fatica, M. & Ruetsch, G. (Éds.). (2014). *CUDA Fortran for Scientists and Engineers*. Boston : Morgan Kaufmann. doi : https://doi.org/10.1016/B978-0-12-416970-8.00008-0.

Fortin, M., Manouzi, H. & Soulaïmani, A. (1993). On finite element approximation and stabilization methods for compressible viscous flow. *International Journal for Numerical Methods in Fluids*, 17, 477 - 499. doi : 10.1002/fld.1650170604.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L. & Woodall, T. S. (2004, September). Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation. *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pp. 97–104.

Harris, M. (2007). Optimizing parallel reduction in CUDA. Repéré à http://developer.download .nvidia.com/assets/cuda/files/reduction.pdf.

Harten, A. (1983). High resolution schemes for hyperbolic conservation laws. *Journal of Computational Physics*, 49(3), 357 - 393. doi : https://doi.org/10.1016/0021-9991(83)90136-5.

Harten, A., Lax, P. D. & Leer, B. v. (1983). On Upstream Differencing and Godunov-Type Schemes for Hyperbolic Conservation Laws. *SIAM Review*, 25(1), 35-61. doi : 10.1137/1025002.

Hemker, P. W. & Spekreijse, S. P. (1985). Multigrid Solution of the Steady Euler Equations. Dans Braess, D., Hackbusch, W. & Trottenberg, U. (Éds.), *Advances in Multi-Grid Methods : Proceedings of the conference held in Oberwolfach, December 8 to 13, 1984* (pp. 33–44). Wiesbaden : Vieweg+Teubner Verlag. doi : 10.1007/978-3-663-14245-4_4.

Jacobsen, D., Thibault, J. & Senocak, I. (2010). An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters.

Jacquier, P., Abdedou, A., Delmas, V. & Soulaïmani, A. (2020). Non-Intrusive Reduced-Order Modeling Using Uncertainty-Aware Deep Neural Networks and Proper Orthogonal Decomposition : Application to Flood Modeling. Soumis à la revue Journal Of Computational Physics.

Karypis, G. & Kumar, V. (2009). MeTis : Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0.

Khorassani, K. S., Chu, C.-H., Subramoni, H. & Panda, D. K. (2019). Performance Evaluation of MPI Libraries on GPU-Enabled OpenPOWER Architectures : Early Experiences. *High Performance Computing*, pp. 361–378.

Kirk, D. B. & mei W. Hwu, W. (Éds.). (2017). *Programming Massively Parallel Processors (Third Edition)* (éd. Third Edition). Morgan Kaufmann. doi : https://doi.org/10.1016/B978-0-12-811986-0.00022-4.

Komatitsch, D., Erlebacher, G., Göddeke, D. & Michéa, D. (2010). High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics*, 229(20), 7692 - 7714. doi : https://doi.org/10.1016/j.jcp.2010.06.024.

Lai, J., Li, H., Tian, Z. & Zhang, Y. (2019). A Multi-GPU Parallel Algorithm in Hypersonic Flow Computations. *Mathematical Problems in Engineering*, 2019, 2053156. doi : 10.1155/2019/2053156.

Loukili, Y. & Soulaïmani, A. (2007). Numerical Tracking of Shallow Water Waves by the Unstructured Finite Volume WAF Approximation. *International Journal for Computational Methods in Engineering Science and Mechanics*, 8. doi : 10.1080/15502280601149577.

Niksiar, P., Ashrafizadeh, A., Shams, M. & Madani, A. H. (2014). Implementation of a GPU-based CFD Code. *2014 International Conference on Computational Science and Computational Intelligence*, 1, 84-89.

NVIDIA. (2020). CUDA Community Showcase. Repéré à https://www.nvidia.com/object/cuda_showcase_html.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E. & Purcell, T. J. (2007). A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1), 80-113. doi : 10.1111/j.1467-8659.2007.01012.x.

Sanders, J. & Kandrot, E. (2010). *CUDA by Example : An Introduction to General-Purpose GPU Programming* (éd. 1st). Addison-Wesley Professional.

Skevik, G. U. (2014). *Load-balancing multi-GPU shallow water simulations on small clusters.* (Mémoire de maîtrise, University of Oslo).

Smith, L. S. & Liang, Q. (2013). Towards a generalised GPU/CPU shallow-flow modelling tool. *Computers & Fluids*, 88, 334 - 343. doi : https://doi.org/10.1016/j.compfluid.2013.09.018.

Soulaïmani, A. (1983). *Nouveaux aspects de l'application de la méthodes des éléments finis pour la résolution des écoulements à surface libre*. (Mémoire de maîtrise).

Soulaïmani, A. & Fortin, M. (1994). Finite element solution of compressible viscous flows using conservative variables. *Computer Methods in Applied Mechanics and Engineering*, 118(3), 319 - 350. doi : https://doi.org/10.1016/0045-7825(94)90006-X.

Suthar, A. & Soulaïmani, A. (2018). Internship report parallelization of shallow water equations solver : CuteFlow.

Toro, E. (2001). *Shock-Capturing Methods for Free-Surface Shallow Flows*.

Turchetto, M., Dal Palù, A. & Vacondio, R. (2020). A General Design for a Scalable MPI-GPU Multi-Resolution 2D Numerical Solver. *IEEE Transactions on Parallel and Distributed Systems*, 31(5), 1036-1047.

Vacondio, R., Dal Palù, A. & Mignosa, P. (2014). GPU-enhanced Finite Volume Shallow Water solver for fast flood simulations. *Environmental Modelling & Software*, 57, 60 - 75. doi : https://doi.org/10.1016/j.envsoft.2014.02.003.

Viñas, M., Lobeiras, J., Fraguela, B., Arenaz, M., Amor, M., García, J., Castro, M. & Doallo, R. (2013). A multi-GPU shallow-water simulation with transport of contaminants. *Concurrency and Computation : Practice and Experience*, 25(8), 1153-1169. doi : 10.1002/cpe.2917.

Xu, C., Zhang, L., Deng, X., Fang, J., Wang, G., Cao, W., Che, Y., Wang, Y. & Liu, W. (2014). Balancing CPU-GPU Collaborative High-Order CFD Simulations on the Tianhe-1A Supercomputer. *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 725-734.

Zaoui, F. (2018). A GPU-enabled finite volume solver for large shallow water simulations.

Zokagoa, J.-M. & Soulaïmani, A. (2010). Modeling of wetting–drying transitions in free surface flows over complex topographies. *Computer Methods in Applied Mechanics and*

Engineering, 199(33), 2281 - 2304. doi : https://doi.org/10.1016/j.cma.2010.03.023.

Zokagoa, J. M. & Soulaïmani, A. (2012). A POD-based reduced-order model for free surface shallow water flows over real bathymetries for Monte-Carlo-type applications. *Computer Methods in Applied Mechanics and Engineering*, s 221–222, 1–23. doi : 10.1016/j.cma.2011.11.012.

Zokagoa, J. M. & Soulaïmani, A. (2018). A POD-based reduced-order model for uncertainty analyses in shallow water flows. *International Journal of Computational Fluid Dynamics*, 1-15. doi : 10.1080/10618562.2018.1513496.