



UNIVERSITÉ
DE GENÈVE
FACULTÉ DES SCIENCES

THÈSE DE MASTER

Implémentation sur GPU de simulation de fluide avec
la méthode Lattice Boltzmann pour des exécutions
hybrides avec Palabos

Adrien PYTHON

Février 2018



Encadrant :

JONAS LATT

Co-encadrant :

Paul ALBUQUERQUE

Avant-propos

Contexte

Ce mémoire présente le travail réalisé pour l'obtention d'un Master en sciences informatiques à l'université de Genève.

Conventions typographiques

Ce document suit des règles typographiques afin d'en simplifier la lecture :

- *l'italique* est employé pour les termes anglais ou sur lesquels l'attention est attirée ;
- une police de caractère à **chasse fixe** indique un extrait de code, une commande, un chemin ou la sortie standard d'une console.

Remerciements

Je souhaite remercier chaleureusement Jonas Latt, pour son suivi régulier ainsi que sa patience tout au long de ce travail ; Paul Albuquerque, pour son soutien ; mes camarades et amis Thomas Bertrand et Jayro Aldaz, pour leur compagnie dans ces innombrables et interminables week-ends et soirées de travail à Battelle (puis chez Costa...) et sans qui je ne serais probablement pas arrivé jusqu'ici ; Stéphane et mes anciens collègues assistants, pour ces trois belles années à l'hepia et leurs constants rappels qu'il serait bien d'enfin finir ce Master ; Céline, pour son précieux soutien ; ma mère pour la relecture de ce document ; et finalement ma famille et mes proches.

Table des matières

1	Introduction	1
2	Méthode de Lattice Boltzmann	2
2.1	Algorithme	2
2.2	Notation DnQm	4
2.3	Unité de mesure de performance LUPS	4
3	État de l'art	5
3.1	Simulations sur processeurs Cell	5
3.2	Simulations sur CPU	6
3.3	Simulations sur GPU	7
3.4	Simulations hybrides (CPU et GPU)	13
3.5	Bilan	15
4	Implémentation	16
4.1	Phase préliminaire	16
4.1.1	Mécanisme de tests automatiques	16
4.1.2	Implémentations Python et C	16
4.2	Implémentations Cuda	19
4.2.1	Sources Cuda de Sailfish	19
4.2.2	Code « historique » de Sailfish	19
4.2.3	Définitions entre calculs sur CPU et GPU	25
4.2.4	Passage de la 2D à la 3D et bibliothèque <code>lbmcuda</code>	26
4.3	Intégration à Palabos	27
4.3.1	Co-processeurs	27
4.3.2	Transfert mémoire entre Palabos et un CP	29
4.3.3	Stratégies de réorganisation des données	31
4.3.4	Simulation d'écoulement dans une cavité	33
5	Mesures de performances	36
5.1	Configuration des <i>benchmarks</i>	36
5.1.1	Simulations effectuées	36
5.1.2	Matériel utilisé	36
5.1.3	Limitations mémoire pour les simulations LBM	36
5.2	<i>Benchmarks</i> des GPU	37
5.2.1	Bande passante et débits de transfert	37
5.2.2	<i>Overhead</i> de lancement d'un <i>kernel</i> et latence des transferts	37
5.3	<i>Benchmarks</i> de l'implémentation GPU	38
5.3.1	Ordre des indices	38

5.3.2	Taille du domaine	39
5.3.3	Taille des blocs	40
5.3.4	Nombre d’itérations	41
5.3.5	Profilage	41
5.4	<i>Benchmarks</i> de l’implémentation hybride	42
5.4.1	Profilage d’exécutions séquentielles	42
5.4.2	Exécutions parallèles	44
6	Modèle de performance	53
6.1	Modèle GPU	53
6.1.1	Identification des paramètres	53
6.1.2	Formulation du modèle de base	53
6.1.3	Affinage du modèle	54
6.1.4	Calcul et mesure des coefficients $\Psi\Gamma F$	56
6.1.5	Courbe théorique	57
6.2	Modèle hybride	58
6.2.1	Identification des paramètres	58
6.2.2	Formulation du modèle	58
7	Bug et améliorations	61
7.1	Erreur de déallocation de mémoire globale	61
7.2	Entrelacement du <i>kernel</i> de transfert et de calcul	61
7.3	Fusion des tableaux de population	61
8	Conclusion	63

Table des figures

2.1	Population D2Q9	2
2.2	Propagation pour la population $f^{in}(x, t)$ au centre ($x = [1, 1]$)	3
3.1	Population D2Q37 telle qu'illustrée par Biferale et al. [4]	6
3.2	Arrangements mémoire AoS et SoA tels qu'illustrés par [2]	9
3.3	Performances de Sailfish telles qu'illustrées par [13]	10
3.4	Algorithmes <i>Push</i> et <i>Pull</i>	10
3.5	Géométries de simulation telles qu'illustrées par [8]	11
3.6	Optimisations de la disposition des populations telles qu'illustrés par [31] .	12
3.7	Performances mesurées par [31] sur une Nvidia GTX285	12
3.8	Performances multi-GPU telles qu'illustrées par [9]	13
3.9	Performances multi-CPU telles qu'illustrées par [9]	14
4.1	Simulation Lattice Boltzmann Method (LBM) d'un fluide autour d'un cylindre	17
4.2	Simulation LBM simplifiée avec une perturbation au centre du domaine . .	18
4.3	Coupe au centre d'une simulation LBM simplifiée en 3D	18
4.4	<i>Streaming uncoalesced</i> d'un bloc de quatre <i>threads</i>	20
4.5	Arrangement en mémoire des populations du <i>streaming</i> de la figure 4.4 . .	21
4.6	Accès mémoire désaligné des <i>threads</i> lors du <i>streaming</i> de la figure 4.4 . .	21
4.7	Stratégie de <i>streaming coalesced</i> par réalignement des populations en mémoire partagée	22
4.8	<i>Streaming coalesced</i> d'un bloc de quatre <i>threads</i>	23
4.9	Accès mémoire aligné des <i>threads</i> lors du <i>streaming</i> de la figure 4.8 . . .	24
4.10	Differences moyennes (par itération) entre les calculs sur CPU et GPU . . .	25
4.11	Découpage d'un domaine 12x4 en trois sous-domaines par Palabos	28
4.12	Adressage d'une population sur Palabos	30
4.13	Enveloppe extérieure et intérieure	31
4.14	Enveloppe Palabos 3D	32
4.15	Taille des transferts entre Palabos et un Co-Processeur (CP)	33
4.17	Images générées par <code>cavity_benchmark</code> d'un écoulement dans une cavité .	34
4.18	Dimensionnement des sous-domaines de <code>cavity_benchmark</code>	34
4.16	Adressage des populations dans un sous-domaine 4×4	35
5.1	Bandé passante effective	38
5.3	Performances en fonction de l'ordre des indices	39
5.4	Performances en fonction de la taille du domaine	40
5.6	Performances en fonction du nombre d'itérations	41
5.9	Performances en fonction du nombre d'itérations sur Palabos	44
5.2	Débit des transferts entre GPU et CPU (<code>cudaMemcpy</code>)	45

5.5	Performances en fonction de la taille des blocs	46
5.7	Profil des temps d'exécution sur un GPU	47
5.8	Courbe des temps d'exécution sur un GPU	48
5.10	Profil des temps d'exécution sur un co-processeur CPU	49
5.11	Profil des temps d'exécution sur un co-processeur GPU	50
5.12	Performance des méthodes de transfert entre Palabos et le co-processeur	51
5.13	Performance d'exécutions parallèles sur 9 <i>threads</i>	52
6.1	Droite de régression linéaire sur les temps mesurés du <i>Collide & Stream</i> . .	55
6.2	Courbes théoriques du modèle de performance	60
7.1	Structure C pour l'arrangement SoA des populations sur le <i>kernel</i>	62

Acronymes

AoS	Array of Structures
ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
CP	Co-Processeur
CS	Collide & Stream
ECC	Error-correcting code
ELBM	Entropic LBM
FLUPS	Lattice Cell Update Per Second
FMA	Fused Multiply-Add
GPU	Graphics Processing Unit
IDE	Environnement de développement Intégré
LBM	Lattice Boltzmann Method
LUPS	Lattice Update Per Second
MPI	Message Passing Interface
SM	Streaming Multiprocessor
SoA	Structure of Arrays
SP	Stream Processor

Chapitre 1

Introduction

La simulation de fluide numérique a de nombreuses applications dans des domaines aussi variés que la géologie, pour l'étude des volcans [6], du jeux-vidéo, pour un rendu réaliste de l'eau, du médical, pour modéliser l'écoulement du sang [12], de l'aviation, pour étudier l'aérodynamisme d'un fuselage et en prédire les sources de bruit [18], et bien autres encore. C'est par conséquent un sujet qui intéresse de nombreux chercheurs.

La méthode de Lattice Boltzmann (Lattice Boltzmann Method (LBM) en anglais) est un algorithme de simulation de fluide attractive pour sa disposition à modéliser des comportements de fluides complexes et à être parallélisé.

Palabos est un solveur en mécanique des fluides numérique basé sur LBM qui permet de distribuer une simulation sur un *cluster* de CPU pour en accélérer les calculs. Ses performances pourraient être améliorée encore davantage avec l'utilisation de GPU pour certaines parties du domaine. En effet, cette technologie, conçue pour réaliser des calculs parallèles à haute vitesse, se prête à la résolution des calculs que demande LBM.

Pour y parvenir, ce travail propose une implémentation de LBM sur GPU, réalisée pour être intégré à Palabos sous la forme d'un module, nommé co-processeur ou accélérateur.

Ce document commence par présenter une vue d'ensemble des méthodes utilisées dans la simulation de fluide avec LBM. Il détaille ensuite l'implémentation réalisée, à travers ses deux principales phases de développement : la réalisation du code GPU puis son intégration à Palabos. Un chapitre analyse par la suite les mesures de performances réalisées pour estimer les capacités de l'implémentation et en dériver un modèle de performance, et ainsi prédire son comportement des configurations différentes. Ce modèle est présenté pour l'implémentation GPU et esquissé pour l'approche hybride avec Palabos. Cette thèse est finalement conclue avec un bilan des améliorations envisageables et du travail réalisé.

Chapitre 2

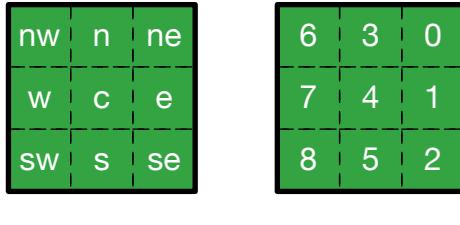
Méthode de Lattice Boltzmann

2.1 Algorithme

Pour simuler le comportement de fluides, LBM découpe l'espace physique à modéliser en un domaine discret composé de populations, organisées en une grille, et discrétise le temps en générations.

Une population représente une portion de l'espace physique. Une simulation consiste en l'application de deux étapes de calculs, effectuées de façon identique sur chaque population du domaine, pour calculer les populations de la génération suivante où le processus pourra être répété. Le résultat de la génération t ne dépend par conséquent que de l'état du domaine à la génération $t-1$. Il s'agit, à l'instar du bien connu jeu de la vie de J. CONWAY, d'un automate cellulaire.

Une population, illustrée par la figure 2.1, est composée de 9 directions (nord, sud, est ...), auxquelles sont attribuées une valeur qui représente la densité de probabilité qu'a la vitesse du fluide d'aller dans une direction donnée et un ordre arbitraire, à travers une numérotation (pour simplifier les formules).



(a) Directions (b) Numérotation

FIGURE 2.1 – Population D2Q9

Les populations du domaine sont notées $f^{in}(x, t) = \{f_0^{in}(x, t), f_1^{in}(x, t), \dots, f_8^{in}(x, t)\}$, avec x la position dans la grille.

La première étape de simulation d'une génération, dite de *collision*, introduit les vecteurs de vitesse v , tel que les voisins de $f^{in}(x, t)$ sont trouvés par $f^{in}(x + v_i, t)$:

$$v_0 = [1, 1]$$

$$v_3 = [0, 1]$$

$$v_6 = [-1, 1]$$

$$v_1 = [1, 0]$$

$$v_4 = [0, 0]$$

$$v_7 = [-1, 0]$$

$$v_2 = [1, -1]$$

$$v_5 = [0, -1]$$

$$v_8 = [-1, -1]$$

On commence par calculer les variables *macroscopiques* de la population, soit sa densité ρ et ses vélocités u :

$$\rho(x, t) = \sum_{i=0}^8 f_i^{in}(x, t) \quad (2.1)$$

$$u = \frac{1}{\rho(x, t)} \sum_{i=0}^8 v_i \cdot f_i^{in}(x, t) \quad (2.2)$$

puis on calcule les directions i de la population f_i^{out} qui résultent de la collision :

$$f_i^{out} = f_i^{in} - \omega \cdot (f_i^{in} - E(i, \rho, u)) \quad (2.3)$$

avec E la fonction d'*equilibrium* :

$$E(i, \rho, u) = \rho \cdot t_i \left(1 + \frac{v_i \cdot u}{c_s^2} + \frac{1}{2 c_s^4} (v_i \cdot u)^2 - \frac{1}{2 c_s^2} |u|^2 \right) \quad (2.4)$$

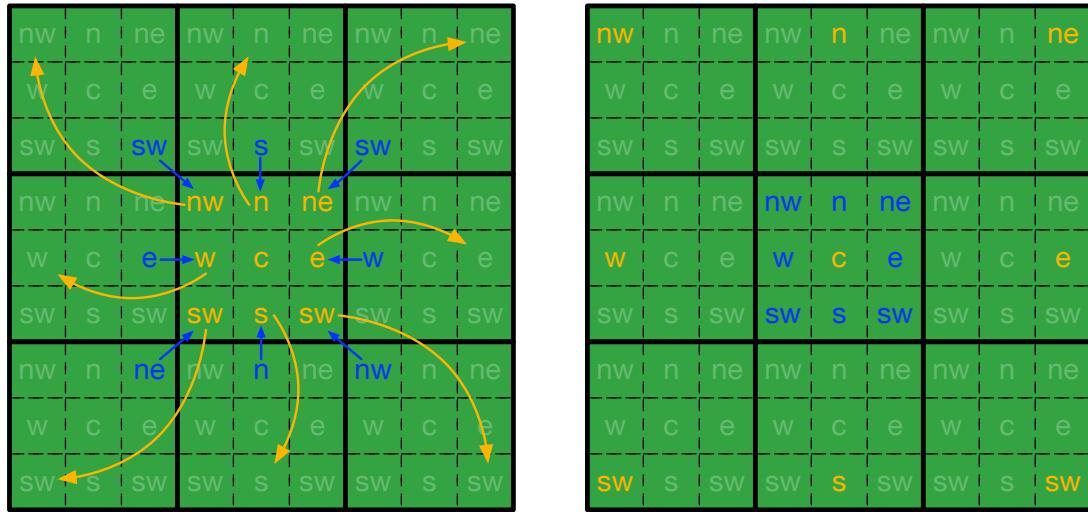
avec c_s la vitesse du son, ω le paramètre de relaxation, et les constantes t_i , qui compense les longueurs inégales des différentes vélocités, avec $t_i = 1/9$ pour les directions orthogonales, $t_i = 1/36$ pour les diagonales et $t_i = 4/9$ pour la centrale, soit :

$$t = \{1/36, 1/9, 1/36, 1/9, 4/9, 1/9, 1/36, 1/9, 1/36\} \quad (2.5)$$

La seconde étape, illustrée par la figure 2.2, propage les densités des populations f_i^{out} calculées dans cette génération t dans les nouvelles populations de la suivante $t + \delta t$, où δt correspond au pas de temps utilisé.

$$f_i^{in}(x, t) = f_i^{out}(x - v_i \delta t, t - \delta t) \quad (2.6)$$

On observe dans ce processus qu'il est hautement parallélisable. Chaque population est calculée indépendamment des autres, jusqu'à la propagation qui n'implique que les voisins directs de la cellule. C'est ce potentiel qu'exploitent les implémentations parallèles, pour accélérer leur simulation.



(a) Génération t

(b) Génération $t + \delta t$

FIGURE 2.2 – Propagation pour la population $f^{in}(x, t)$ au centre ($x = [1, 1]$)

2.2 Notation DnQm

L'algorithme présenté dans la section 2.1 est axé sur un domaine en 2 dimensions, et avec des populations à 9 directions. On note cette configuration D2Q9, avec D2 pour le nombre de dimensions et Q9 le nombre de directions.

En effet, l'algorithme est également applicable pour 3 dimensions, avec les configurations D3Q15, D3Q19 et D3Q27.

2.3 Unité de mesure de performance LUPS

La simulation de fluide avec LBM est un processus gourmand en mémoire et en temps de calcul. Un pan important de la recherche concernant LBM vise à trouver des techniques pour améliorer les performances de son implémentation.

Les Lattice Update Per Second (populations calculées par seconde) ou LUPS sont communément utilisés comme unité de mesure des performances.

Chapitre 3

État de l'art

LBM est un algorithme propice pour des simulations distribuées sur les nœuds d'un *cluster* de calcul, ou sur les cœurs d'un GPU. Il a par conséquent attiré l'attention de nombreux chercheurs.

Ce chapitre fait un tour d'horizon des technologies et des méthodes utilisées pour améliorer les performances des simulations basées sur la méthode de Lattice Boltzmann. Les sections suivantes présentent les travaux chronologiquement, en commençant par les simulations réalisées sur les processeurs Cell, puis les travaux qui utilisent les CPU comme puissance de calcul. Sont ensuite observées les recherches axées sur le portage de cet algorithme sur GPU et finalement les méthodes de simulation hybrides entre CPU et GPU.

Les articles discutés ici se concentrent, pour la plupart, sur les implémentations D3Q19, mais d'autres configurations sont abordées et sont alors spécifiées.

3.1 Simulations sur processeurs Cell

Cell est un processeur multi-core développé par Sony, Toshiba et IBM à partir de 2001. Ces processeurs, optimisés pour le calcul parallèle, équipent notamment la console de jeu PlayStation3, mais aussi certains super-ordinateurs.

Peng et al. [26] comparent en 2008 les performances d'un code LBM sur un *cluster* de neuf consoles PlayStation3 avec celles d'un *cluster* Xeon classique. Pour l'étape de collision, ils observent que les PlayStation3 dépassent les performances des Xeon pour les problèmes plus grands que 1024 et qu'elles s'améliorent plus le domaine est grand. L'étape de propagation marque de meilleures performances sur le *cluster* de PlayStation3 pour les domaines plus grands que 2048. En effet, cette étape est surtout intensive au niveau des communications. Or, la bande passante Ethernet (plutôt bas prix) de la PlayStation3 est moins bonne que celle d'un *cluster* Xeon.

Au final, les auteurs observent des performances jusqu'à 11.02 fois meilleures avec les processeurs Cell de la PlayStation3 que sur un cluster Xeon, avec le domaine de simulation le plus grand. À titre de comparaisons, une seconde implémentation Cuda est réalisée et affiche un *speed-up* de 8.76 par rapport au CPU.

Stürmer et al. [29] présentent en 2009 un solveur LBM sur processeur Cell pour l'étude des anévrismes. En effet, il est très difficile, si ce n'est impossible, de mesurer avec précision les paramètres hémodynamiques dans les artères intracrâniennes d'un patient vivant. Une simulation sur PlayStation3 offre une solution pour la compréhension du développement

des anévrismes, et de leur ruptures, qui se révèle abordable pour l'hôpital qui l'utilise (la console étant vendue à 400€ à l'époque).

En plus de son prix avantageux, de ses dimensions compactes qui permettent d'installer le dispositif n'importe où, et de sa faible consommation d'énergie, les auteurs affirment que la PlayStation3 dépasse par un facteur de 3.7 les performances d'un Intel Xeon 5160, avec la solution développée.

Biferale et al. [4] réalisent en 2010 une implémentation LBM D2Q37 (figure 3.1) pour QPACE, un super-ordinateur massivement parallèle basé sur des processeurs PowerX-Cell 8i. Les simulations réalisées étudient les propriétés de l'instabilité de Rayleigh–Taylor sur de grands domaines de 4096×6000 et 2048×3600 populations, sur respectivement $2.6 \cdot 10^5$ et 10^6 générations pour des temps d'exécution de 32 et 44 heures par simulation. Les auteurs notent vis-à-vis de l'implémentation que leur routine de collision est la plus gourmande en temps d'exécution, en raison des calculs de nombres flottants, et que la propagation constitue le goulot d'étranglement de plus important de l'implémentation, en raison de la trop faible bande passante de ~ 3.5 Go/s.

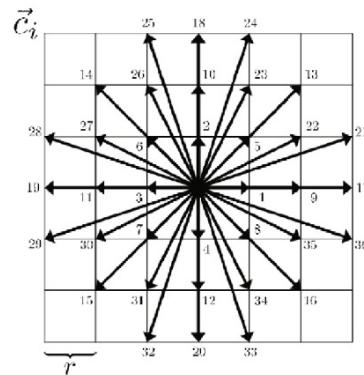


FIGURE 3.1 – Population D2Q37 telle qu'illustrée par Biferale et al. [4]

Williams et al. [32] développent en 2008 un ensemble d'optimisations pour LBMHD (une application LBM pour modéliser les turbulences magnétohydrodynamiques) appliquées automatiquement par un générateur de code, qui crée plusieurs versions du code (par optimisation appliquée) pour les architectures multi-core Intel Clovertown, AMD Opteron X2, Sun Niagara2, STI Cell et Intel Itanium2. Les codes sont ensuite auto-évalués pour déterminer les optimisations les plus intéressantes par plateforme.

Les auteurs observent que le processeur Cell offre très nettement les meilleures performances, mais déplorent la difficulté de développer pour ce matériel (éloigné des pratiques de programmation classiques). Ils soulignent également le succès de leur méthode d'*auto-tuning* du code qui parvient à des *speed-up* allant jusqu'à 14 fois les performances du code original.

3.2 Simulations sur CPU

Si les processeurs de type Cell ont un temps offert une alternative intéressante aux CPU pour accélérer les calculs parallèles, ces derniers restent pourtant toujours une unité de calcul très utilisée de nos jours.

Min et al. [23] exécutent en 2013 des *benchmarks* pour le programme de simulation LBM distribuée Palabos, sur le super-ordinateur chinois Sunway BlueLight MPP (le deuxième super-ordinateur le plus puissant de Chine en 2011). Ils décrivent dans leur article l'utilisation novatrice des *flags* (standard) d'optimisation pour la compilation les sources de Palabos, à savoir `optimize = true, MPIparallel=true, optimFlags=-O3 -OPT:Ofast` et la définition des compilateurs SW1600 `serialCC=swCC` et `parallelCXX=mpiswCC`.

Ils comparent ensuite dans leurs mesures les performances relevées pour différents nombres de *cores* utilisés et relèvent que leur stratégie d'exécution parallèle et d'optimisation réduit le temps d'exécution (par rapport à une exécution séquentielle et non-optimisée).

En 2016, Li et al. [19] optimisent des exécutions parallèles de l'implémentation D3Q19 du programme open-source *openlbfmflow* sur Tianhe-2, le super-ordinateur chinois le plus puissant de l'époque. Cette machine est composée de 16000 noeuds de calculs, chacun muni de deux processeurs Xeon E5-2692 v2 et trois Xeon Phi 31S1P MIC.

Les auteurs exploitent les capacités du *cluster* à travers des exécutions hybrides sur ces deux types de processeurs. Pour tirer les meilleures performances de cette collaboration, ils optimisent l'utilisation de la cache et de l'architecture SIMD, utilisent une implémentation hybride entre MPI et OpenMP pour la parallélisation et un système de *load-balancing* entre les deux types de processeurs, pour éviter des transferts de données inutiles.

Sur un seul noeud de calcul, l'implémentation affiche jusqu'à un *speed-up* de 3.2 avec l'approche collaborative, par rapport à une exécution n'utilisant que les deux CPU du noeud. Les mesures d'exécutions sur de nombreux noeuds montrent quant à eux une bonne capacité de montée en charge de l'approche hybride.

En 2017, Schmieschek et al. [28] présentent leur optimisation de l'implémentation *open-source* LB3D qu'ils ont ensuite testé sur le super-ordinateur ARCHER au Royaume-Uni. Afin de vérifier que le *refactoring* opéré n'a pas compromis le fonctionnement d'origine de LB3D, les auteurs ont réalisé des simulations de décomposition spinodale d'un mélange amphiphile, de détermination de la perméabilité d'un modèle en milieu poreux et enfin de la formation de mésophases amphiphiles.

Les simulations réalisées sur ARCHER montrent un *speedup* de 3 par rapport à la version non optimisée et permettent d'observer une excellente montée en charge jusqu'à 49152 *cores* vers lesquels les performances commencent à se détériorer.

3.3 Simulations sur GPU

L'évolution des capacités de calcul des GPU, qui dépassent désormais largement celles des CPU, ne passe pas inaperçue aux yeux des chercheurs, qui au début des années 2000 commencent à en explorer les capacités pour LBM [5, 7, 15]. En effet, cette technologie offre des performances grandissantes d'année en année, à un prix raisonnable. Une tendance soutenue en 2007 avec l'arrivée de CUDA, une technologie qui permet de programmer en C les GPU de Nvidia.

Li et al. [20] proposent ainsi en 2003 une implémentation qu'ils testent sur une GeForce4 Ti 4600 et comparent les performances avec une implémentation CPU sur un PC avec un processeur P4 1.6GHz et 512MB de mémoire DDR. Ils observent un *speed-up* de 50 par rapport à la version CPU, à partir d'un domaine de dimension 32^3 . Les per-

formances alors atteintes, d'environ 9.8 MLUPS pour un domaine de 256^3 , permettent de visualiser interactivement la simulation, notamment des domaines de 64^3 et 128^3 .

Toelke and Krafczyk [30] implémentent en 2008 un modèle D3Q13 avec la technologie CUDA, publiée un an plus tôt. Ils en explorent et expliquent brièvement le fonctionnement, avant de détailler celui de leur implémentation. Les mesures, réalisées avec une GeForce 8800, affichent des performances allant jusqu'à 592 MLUPS en simple précision.

Kaufman et al. [14] publient en 2009 leurs mesures de performances obtenues avec quatre implémentations interactives sur GPU qu'ils comparent à des simulations similaires sur CPU.

La première est programmée avec Cg (un langage développé par Nvidia et Microsoft) et OpenGL, en simple précision. Testée sur une GeForce FX 5900 Ultra (cadencé à 400 MHz avec 256MB de DDR SDRAM), elle atteint les 3.87 MLUPS sur un domaine de dimensions 128^3 , soit un *speed-up* de 17 par rapport à une implémentation CPU sur un Pentium IV (cadencé à 2.53 GHz avec 1 GB de PC800 RDRAM).

La seconde simule un domaine $480 \times 400 \times 80$ sur un *cluster* de GPU (GeForce FX 5900 Ultra) où chaque nœud calcule un sous-domaine 80^3 . Elle parvient à 51.68 MLUPS sur 32 nœuds, contre 11.38 MLUPS avec un *cluster* de CPU (Intel Pentium Xeo 2.4 GHz).

La troisième utilise Zippy, un *framework* qui simplifie le développement d'application pour *cluster* de GPU. Les simulations, réalisées avec la même configuration que celle du paragraphe précédent, atteignent les 80 MLUPS sur un *cluster* de GPU Nvidia Quadro FX 4500 et de CPU Intel Xeon 3.6 GHz.

Finalement, la troisième implémentation, réalisée avec CUDA, atteint les 144 MLUPS sur une Nvidia Quadra FX 4600 sur un domaine 64^3 (contre 94 MLUPS avec Zippy sur cette même carte) et 278 MLUPS sur un domaine 96^3 avec une Nvidia Tesla C1060. Avec cette carte, les performances de cette dernière simulation chutent à 121 MLUPS en double précision.

La même année, Bailey et al. [3] présentent une implémentation CUDA et discutent des optimisations réalisées (notamment vis-à-vis de l'accès à la mémoire globale du GPU) pour en améliorer les performances. Ils les mesurent ensuite sur une Nvidia 8800 GTX sur laquelle ils relèvent 300 MLUPS en simple précision. Ceci correspond à un *speed-up* de 28 par rapport à une simulation sur CPU Intel quadcore d'une implémentation utilisant OpenMP (une interface de programmation parallèle).

Les auteurs relèvent par ailleurs que CUDA permet une utilisation plus précise des ressources matérielles du GPU par rapport aux autres interfaces de programmation OpenGL, Rapidmind et BrookGPU.

Kuznik et al. [16] présentent en 2010 une implémentation D2Q9 en CUDA sur laquelle ils mesurent notamment l'influence sur les performances des nombres flottants en simple et double précision. Si leur implémentation atteint les 915 MLUPS avec un domaine 2048^2 sur une Nvidia GTX280, les performances tombent à 239 MLUPS en double précision, soit d'un facteur 3.8.

Rinaldi et al. [27] proposent en 2012 une implémentation CUDA sur laquelle ils relèvent l'importance pour les performances qu'occupent les accès *coalesced* à la mémoire globale, son schéma d'accès, l'utilisation de la mémoire partagée et la réduction du nombre de

branchements (`if/else`).

Ils mesurent ensuite les performances de leur implémentation qui atteignent 259 MLUPS sur une Geforce GTX 260. Des optimisations supplémentaires, comme l'utilisation de constante à la place de paramètres d'exécution et la réduction du nombre d'*output* entre les itérations, permettent d'atteindre 400 MLUPS, au prix d'une flexibilité et précision réduite.

La même année, Astorino et al. [2] présentent leurs implémentations D3Q15 et D3Q19 avec lesquelles ils atteignent respectivement 490 MLUPS et 370 MLUPS sur une GeForce GTX 480.

Parmi les optimisations exposées, leur article analyse les arrangements mémoires AoS (Array of Structures) et SoA (Structure of Arrays), illustré par la figure 3.2.

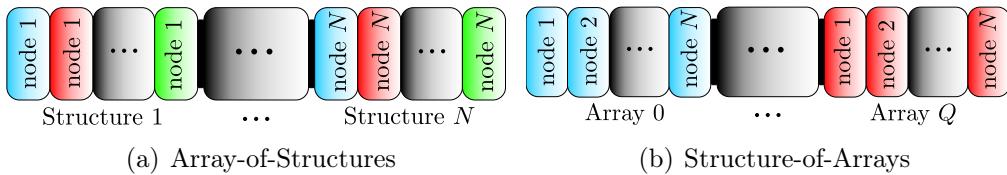


FIGURE 3.2 – Arrangements mémoire AoS et SoA tels qu’illustrés par [2]

AoS conserve les structures de populations $f_i(x)$ les unes à côté des autres, soit leurs directions $i = \{1 \dots Q\}$ avec Q le nombre de directions (9 en D2Q9, 19 en D3Q19, ...) et N la taille du domaine :

$$f_{\text{AoS}} = f_1(1), f_2(1), \dots, f_Q(1), f_1(2), f_2(2), \dots, f_Q(2), \dots, f_1(N), f_2(N), \dots, f_Q(N)$$

tandis qu’SoA conserve les « tableaux » de populations avec les cellules côte à côte :

$$f_{\text{SoA}} = f_1(1), f_1(2), \dots, f_1(N), f_2(1), f_2(2), \dots, f_2(N), \dots, f_Q(1), f_Q(2), \dots, f_Q(N)$$

Ces deux arrangements optimisent des étapes différentes. En effet, AoS est efficace lors de la collision, qui accède simultanément aux directions de la cellule x , tandis que SoA est optimisé pour la propagation, qui accède aux cellules voisines de x , une direction après l'autre. C'est cet arrangement qu'utilise l'implémentation que présentent les auteurs.

Habich et al. [11] publient en 2013 une comparaison entre les performances de leurs implémentations CUDA et OpenCL sur une carte Nvidia Tesla C2070 et AMD 6970. Ils commencent par mesurer l'effet de l'utilisation d'une mémoire ECC (Error-correcting code memory) et notent une baisse des performances de 10 à 18% lorsqu'elle est activée.

Les deux implémentations affichent les mêmes performances sur les deux cartes avec 650 MLUPS en simple précision et 290 MLUPS en double précision.

Januszewski and Kostur [13] présente Sailfish en 2014, une implémentation multi-GPU en Python de LBM. Cette dernière utilise une approche novatrice, dans le sens qu'elle génère au *run-time* le code CUDA ou OpenCL qu'elle exécute ensuite sur GPU. Sailfish profite ainsi de l'expressivité et de la flexibilité du langage Python pour la configuration des simulations et de la puissance de calcul des GPU offerte par l'utilisation de CUDA ou OpenCL.

La figure 3.3 illustre les mesures de performances réalisées par les auteurs, avec un code généré en CUDA sur un domaine $256 \times 100 \times 100$, en simple et double précision, en D3Q13, D3Q15, D3Q19 et D3Q27 sur 6 cartes graphiques différentes.

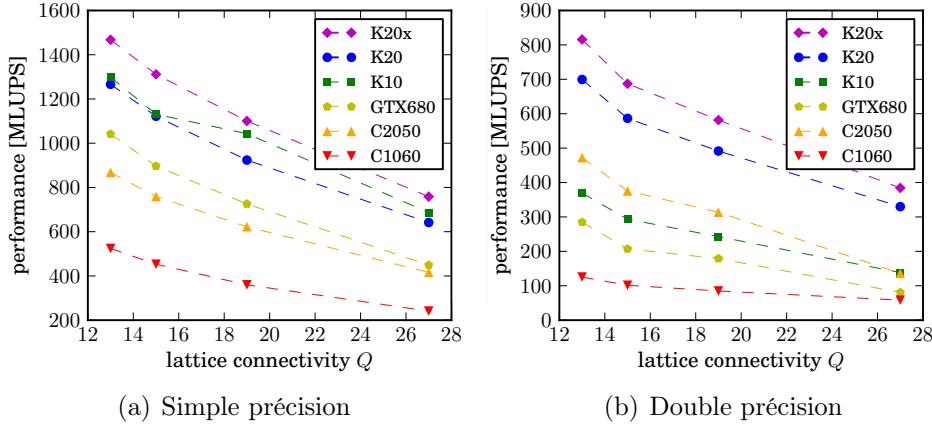


FIGURE 3.3 – Performances de Sailfish telles qu'illustrées par [13]

La même année, Mawson and Revell [22] proposent leur implémentation CUDA qui utilise l'arrangement mémoire SoA, pour favoriser les accès *coalesced* à la mémoire globale (les populations voisines étant proches les unes des autres pour la propagation). Leur travail discute également des algorithmes *Push* et *Pull* (figure 3.4).

L'utilisation de la méthode *Pull* s'avère avantageuse, car elle favorise les accès *uncoalesced* en lecture par rapport aux accès *uncoalesced* en écriture qui sont plus coûteux.

Les mesures réalisées avec les cartes graphiques K5000m et K20c révèlent respectivement des performances de 420 MLUPS et 1036 MLUPS en simple précision.

Algorithm 1: Méthode <i>Push</i>	Algorithm 2: Méthode <i>Pull</i>
<pre> for all $x \in f_i(x, t)$ do for all i do $f_i^{local} \leftarrow f_i(x, t)$ end for Conditions aux bords Calcul de ρ et u for all i do Calcul de collision dans f_i^{local} $f_i(x + v_i, t + 1) = f_i^{local}$ end for end for </pre>	<pre> for all $x \in f_i(x, t)$ do for all i do $f_i^{local} = f_i(x - v_i \delta t, t - \delta t)$ end for Conditions aux bords Calcul de ρ et u for all i do Calcul de collision dans f_i^{local} end for end for </pre>

FIGURE 3.4 – Algorithmes *Push* et *Pull*

Campos et al. [8] montrent en 2016 une utilisation de LBM sur GPU pour la simulation numérique des équations d'électrophysiologie cardiaque. L'implémentation CUDA utilise le modèle D3Q7 et un arrangement mémoire des populations SoA.

Plutôt qu'utiliser des variables temporaires pour conserver localement les valeurs f_i d'une population, leur implémentation utilise une autre méthode [17, 21] pour économiser la mémoire qui *swap* ces valeurs avec une autre variable.

Contrairement à une simulation ordinaire, dont le domaine est généralement cubique ou rectangulaire, la leur s'applique sur un domaine à la géométrie irrégulière (figure 3.5). L'implémentation utilise un tableau d'indexage indirect pour représenter efficacement la géométrie de ce type de domaine.

Les mesures réalisées affichent des performances allant jusqu'à 419 MLUPS en simple précision sur une Tesla M2050, soit un *speed-up* de 500 par rapport aux mesures réalisées sur un CPU Intel Xeon E5620 2.4 GHz. Les mesures relevées en double précision montrent quant à elles un *speed-up* allant jusqu'à 300.

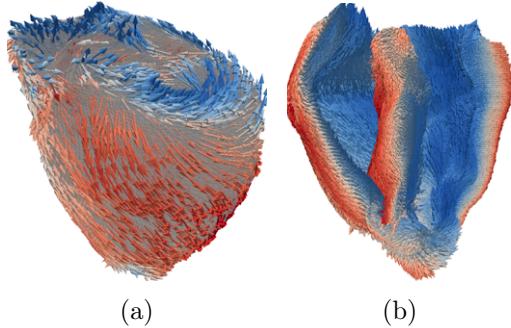


FIGURE 3.5 – Géométries de simulation telles qu'illustrées par [8]

Obrecht et al. [25] présentent en 2016 une implémentation CUDA performante du modèle en double population LW-ACM, pour laquelle ils réalisent des mesures de performances sur une Nvidia GeForce GTX Titan Black dotée d'un GPU GK110. Ces dernières atteignent les 1295 MLUPS sur une simulation de cavité de dimensions 256^3 en simple précision.

En 2017, Tran et al. [31] expérimentent plusieurs méthodes d'optimisation dans leur implémentation CUDA et mesurent les performances de leur implémentation. Du point de vue de la mémoire, celle-ci utilise l'arrangement SoA, pour réduire le coût des accès *uncoalesced*, ainsi qu'une disposition mémoire optimisée pour réduire l'écart en mémoire entre les populations (figures 3.6).

Les auteurs observent que le modèle D3Q19, bien que plus précis que les modèles avec moins de directions, utilise plus de variables et par conséquent plus de registres sur le GPU. Si le nombre de registres utilisés est supérieur aux nombres de registres disponibles, il en résulte une perte de performance en raison de l'utilisation de la mémoire locale par le GPU. Ils proposent ainsi un certain nombre de mesures visant à réduire au maximum l'utilisation inutile des registres, comme a. réduire le nombre de variables intermédiaires (pour le calcul des index notamment) b. utiliser, si possible, la mémoire partagée à la place de variables locales c. combiner les variables `char` d'un octet dans un entier de quatre octets et d. réutiliser les variables existantes dont le *scope* est plus grand que nécessaire.

Les mesures de performance sont réalisées sur une Nvidia Tesla K20 et une Nvidia GTX285. La figure 3.7 évalue, sur la Nvidia GTX285, l'effet des optimisations proposées ainsi que des méthodes *Push* et *Pull* également implémentées. Dans les meilleures conditions, l'implémentation atteint finalement les 1210.63 MLUPS sur la Nvidia Tesla K20.

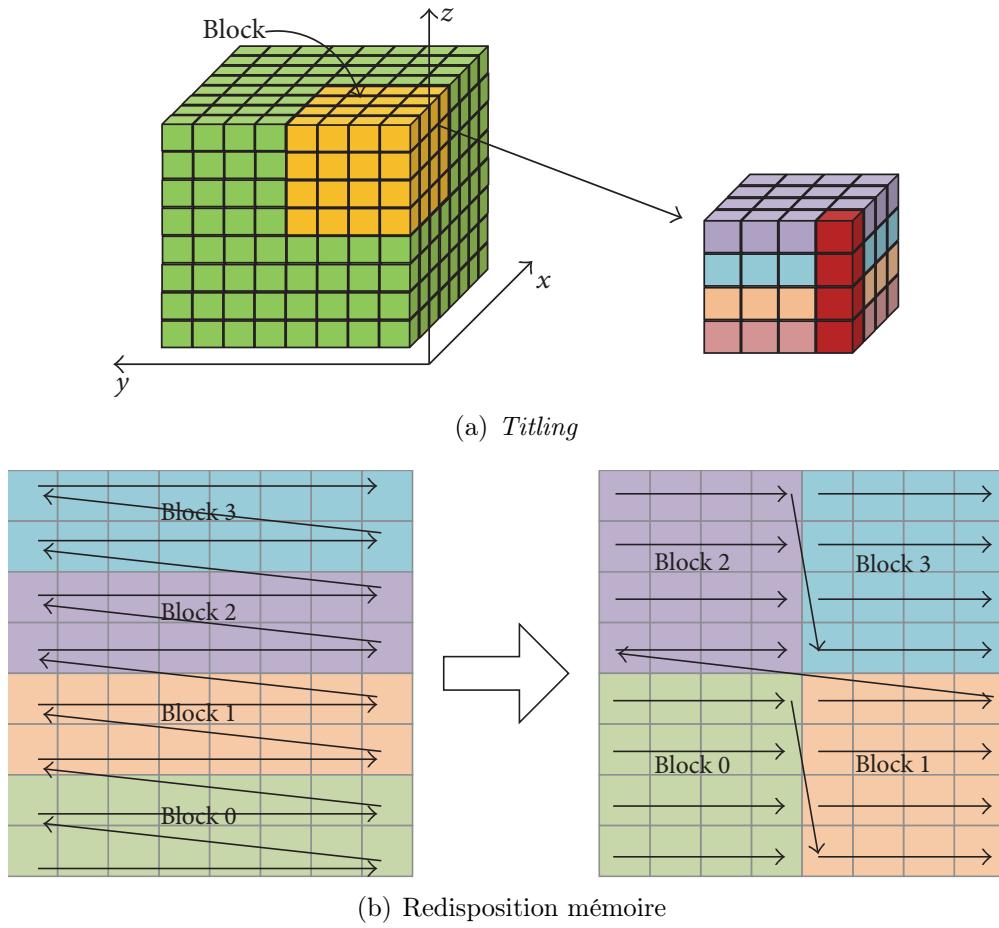


FIGURE 3.6 – Optimisations de la disposition des populations telles qu'illustrés par [31]

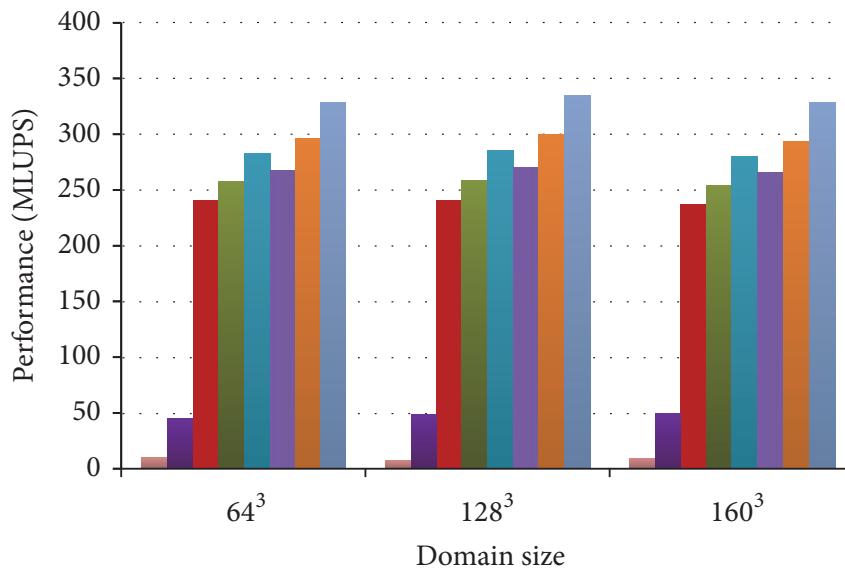


FIGURE 3.7 – Performances mesurées par [31] sur une Nvidia GTX285

3.4 Simulations hybrides (CPU et GPU)

Les CPU et les GPU ont tous les deux leurs avantages dans différentes configurations. Par ailleurs, l'utilisation d'un GPU requiert un CPU pour lancer son *kernel* et les *cluster* équipent en général de puissants processeurs multi-cœurs. Ignorer leur puissance de calcul sous prétexte que les GPU sont de façon générale plus rapides est une perte regrettable. Des travaux adressent cette question et essaient de faire collaborer les CPU et GPU.

Feichtinger et al. [9] présentent en 2011 leur approche hybride à l'aide de WaLBerla, un *framework* de simulation physique massivement parallèle. Il découpe une tâche de simulation en étapes nommées *Sweeps*, constituées d'une étape de communication de l'enveloppe d'un sous-domaine à ses voisins et d'une étape de travail, occupée par un appel au *kernel* dans le cas de l'utilisation de GPU.

Leurs implémentation CPU utilise l'arrangement mémoire AoS tandis que leur implémentation GPU utilise l'arrangement SoA et la méthode *Pull*.

Les performances mesurées pour leur implémentation GPU seul atteignent 500 et 250 MLUPS, respectivement en simple et double précision, sur une Tesla C1060. Sur un seul nœud, avec un GPU et un CPU, les performances plafonnent à partir d'un domaine de dimension 200^3 contre 70^3 lorsque seul le GPU est utilisé en raison de l'*overhead* occasionné par la copie des enveloppes. De ce fait, les performances diminuent avec le nombre de sous-domaines, à raison de 5% avec 8 sous-domaines et 25% avec 64. Les meilleures performances sont atteintes avec 340 et 167 MLUPS, respectivement en simple et double précision.

Pour plusieurs nœuds de calcul GPU, bien qu'on observe une augmentation des performances avec celle du nombre de nœuds, les performances relatives diminuent entre 1 et 30 nœuds de 500 à 235 MLUPS en simple précision et de 250 à 100 MLUPS en double précision (figure 3.8). Les auteurs observent que l'approche multi-GPU fait un usage moins efficace du matériel qu'en multi-CPU (figure 3.9), mais nécessite moins de nœuds de calcul.

Les simulations réalisées en configuration hybride répartissent équitablement la charge entre CPU et GPU. Les auteurs notent toutefois que, dans un objectif d'augmentation des performances, leur implémentation n'est pas adaptée, mais qu'elles peuvent être améliorées dans certaines conditions. Ils considèrent dans leurs futurs travaux de ne simuler que des domaines purement liquides avec les ressources GPU.

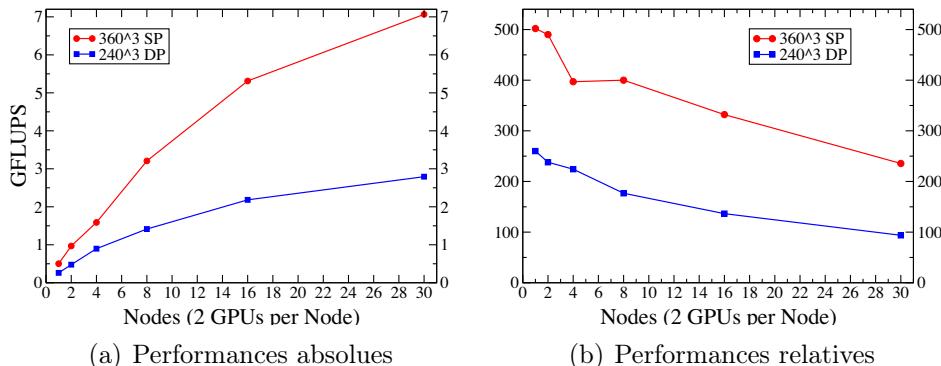


FIGURE 3.8 – Performances multi-GPU telles qu'illustrées par [9]

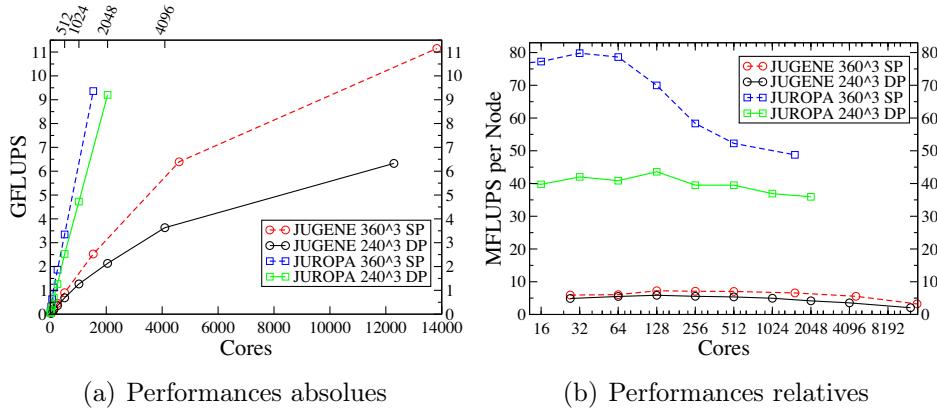


FIGURE 3.9 – Performances multi-CPU telles qu’illustrées par [9]

En 2015, Ye et al. [33] présentent leur travail basé sur Entropic LBM (ELBM). Le *load-balancing* pour la division du domaine est basé sur un modèle de performance et les arrangements mémoires utilisés sont les classiques SoA pour l’implémentation CPU et AoS pour le GPU.

Les mesures de performances sont réalisées sur deux AMD Opteron™ (2.2GHz) et une Tesla C2050. Tandis qu’une simulation sur CPU atteint les 1.75 MLUPS et 78.52 MLUPS sur GPU, une simulation hybride atteint les 99.72 MLUPS, soit un *speed-up* de 5.7 par rapport au CPU seul.

La même année, Feichtinger et al. [10] présentent la suite de leur travail avec le *framework* waLBerla. Leurs simulations hybrides sur le super-ordinateur Tsubame 2.0, composé de processeurs Intel Xeon X5670 et Tesla M2050, mettent à contribution plus de mille GPU et parviennent à atteindre les 80 GFLOPS.

Pour y parvenir, leur implémentation décompose le domaine non uniforme et masque le temps de transfert entre GPU et CPU, qui est supérieur au temps d’exécution du *kernel*, en employant une approche à deux *kernels* entrelacés. Un *kernel* extérieur, qui s’occupe de l’enveloppe du sous-domaine pour les communications, et un *kernel* intérieur qui s’occupe du reste du sous-domaine, qui n’a ainsi plus de dépendance pour les communications et peut être exécuté en parallèle. Les auteurs définissent le temps d’exécution t_s dans un modèle de performance, avec entrelacement :

$$t_s = t_k + t_b + t_{pci} + t_{mpi} \quad (3.1)$$

et sans entrelacement :

$$t'_s = t_{k,outer} + \max(t_{k,inner}, t_b + t_{pci} + t_{mpi}) \quad (3.2)$$

avec les temps t_k du *kernel* ($t_{k,inner}$ intérieur et $t_{k,outer}$ extérieur), t_b de copie de *buffer*, t_{pci} de transfert PCI-E et t_{mpi} de communication MPI.

Les simulations sur un seul noeud atteignent 93 MFLUPS pour l’implémentation CPU et 234 MFLUPS pour l’implémentation GPU. L’approche hybride, sur un CPU et un GPU, permet quant à elle d’obtenir un *speed-up* de 28% par rapport à une simulation sur un seul GPU. Les 640 MFLUPS sont atteints avec trois GPU et avec un *speed-up* de 10% si on y ajoute l’approche hybride. Les auteurs notent que ces deux scénarios sont correctement prédits par leur modèle de performance.

3.5 Bilan

Pour améliorer les performances des simulations utilisant le modèle de Lattice Boltzmann, de nombreuses approches ont été testées au fil du temps en mettant à profit les technologies les plus puissantes de leur époque. Une tendance marquée s'oriente ces dernières années vers les GPU, dont la puissance de calcul dépasse depuis celle des CPU pour un prix intéressant. Les recherches dans ce domaine ont permis de mettre à jour de nombreuses optimisations (évitement ou réduction du coût des accès *uncoalesced*, limitation du nombre de registres...) qui améliorent les performances des implémentations GPU de LBM.

Toutefois, l'utilisation de la puissance de calcul des CPU a encore de beaux jours devant elle. Comme l'observe Kaufman et al. [14], les CPU et GPU possèdent tous les deux leurs avantages propre. À ce titre, les implémentations hybrides ont un avenir prometteur, puisqu'elles mettent en commun les qualités de ces deux mondes pour en profiter. Une conclusion que confirment les dernières recherches réalisées dans ce domaine et encourage la poursuite d'un développement de module GPU pour Palabos.

Chapitre 4

Implémentation

4.1 Phase préliminaire

Le projet considère deux principales étapes dans son développement : l'implémentation d'un code Cuda performant de l'algorithme de LBM puis son intégration dans Palabos.

Toutefois, une phase de préparation du projet et de familiarisation à LBM eut lieu au préalable. Elle se base sur un code Python existant, nommé `lbm_py`, qui simule un écoulement en deux dimensions autour d'un cercle.

Cette implémentation sert à la fois de base aux implémentations suivantes et permet de vérifier la validité de leurs résultats. En effet, la première étape de cette phase préliminaire fut de ré-implémenter cette simulation en C. Par conséquent, un mécanisme de test automatique est nécessaire.

4.1.1 Mécanisme de tests automatiques

Le mécanisme adopté s'appuie sur un système de *makefile*. Chaque implémentation en possède un avec une cible `output`. Elle génère un fichier qui contient les valeurs des populations à la fin de la simulation. Le `makefile` principal, situé à la racine, possède lui une cible `test` qui appelle les cibles `output` de toutes les implémentations du projet puis compare leur résultat au fichier de référence (produit par exemple par `lbp_py`).

4.1.2 Implémentations Python et C

Le projet a débuté avec le code Python `lbp_py`, porté en C sous le nom de `lbp_py2c` (figure 4.1), et deux autres codes Python ont par la suite été implémentés sur cette base :

- `lbp_simple` : Simplifie `lbp_py` en retirant l'écoulement ainsi que la gestion des obstacles et initialise l'espace de recherche avec une perturbation au centre (figure 4.2).
- `lbp_simple_3d` : Portage en trois dimensions de `lbp_simple` et porté en C sous le même nom (figure 4.3).

Les codes Python servent de base de test (voir section 4.1.1) pour leur implémentation respective en C puis Cuda.

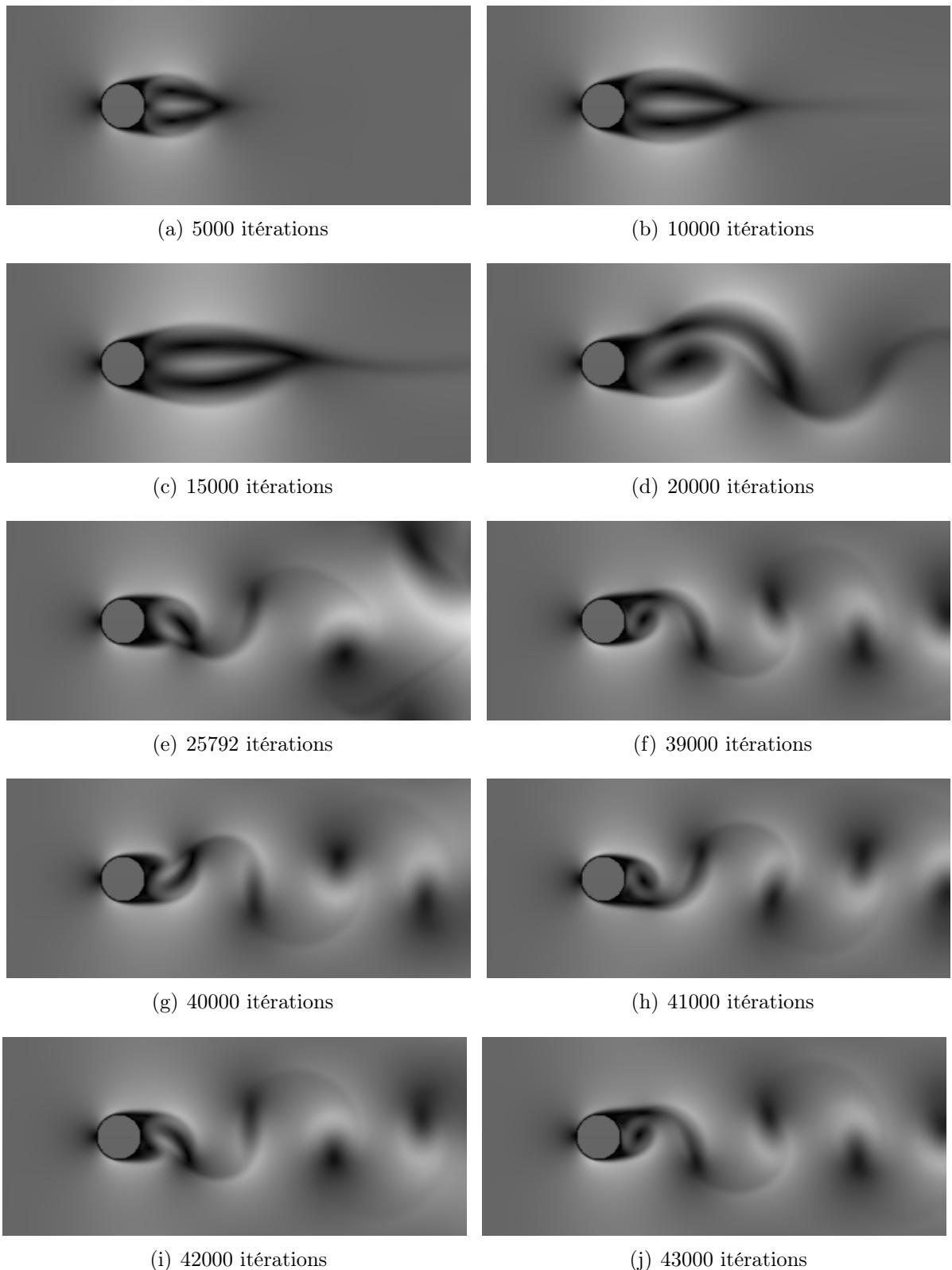


FIGURE 4.1 – Simulation LBM d'un fluide autour d'un cylindre

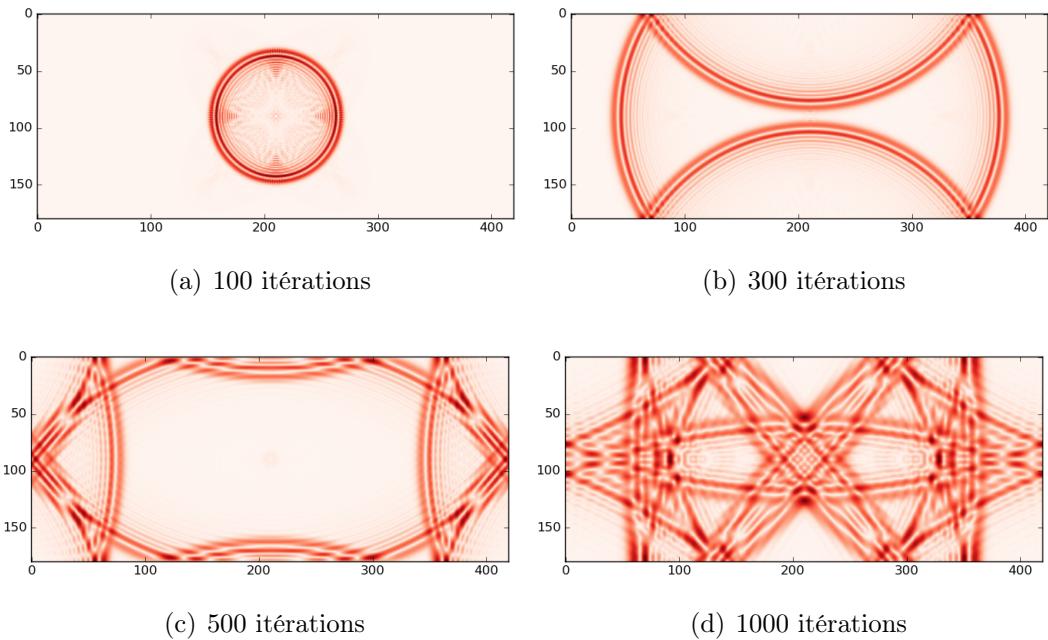


FIGURE 4.2 – Simulation LBM simplifiée avec une perturbation au centre du domaine

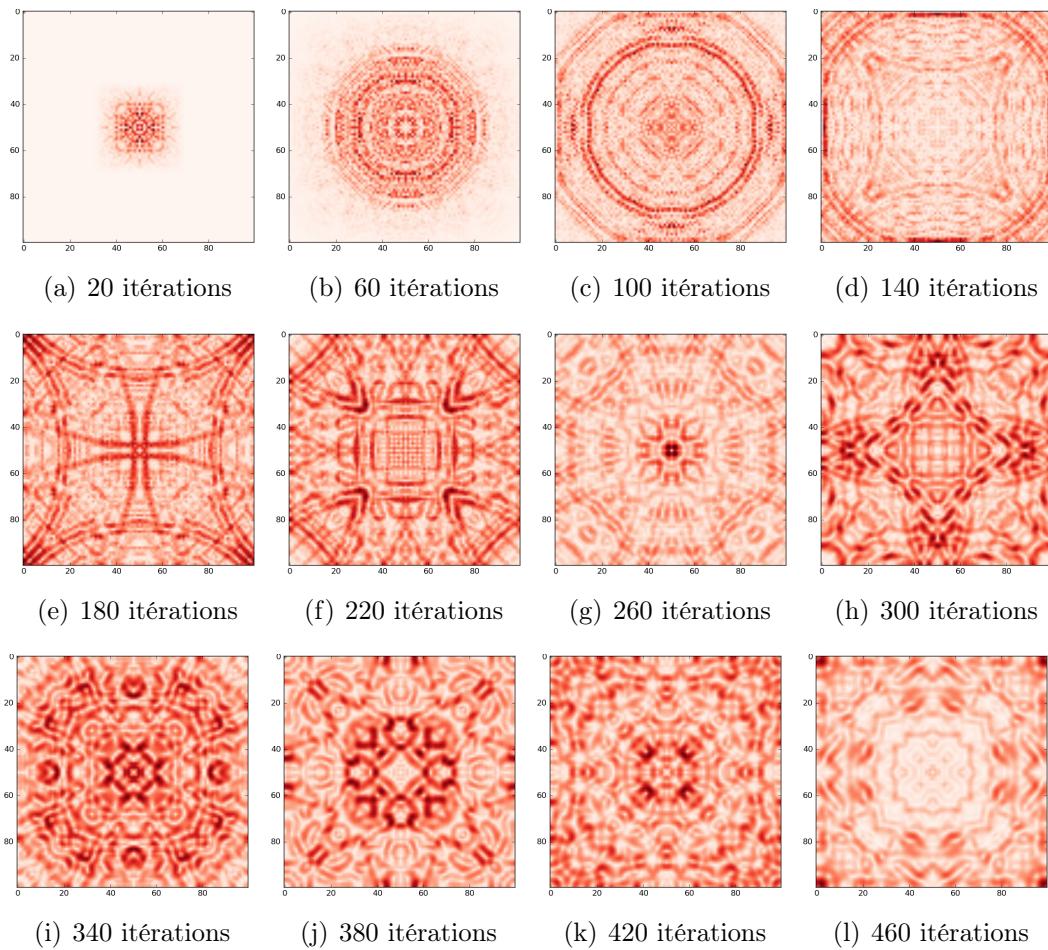


FIGURE 4.3 – Coupe au centre d'une simulation LBM simplifiée en 3D

4.2 Implémentations Cuda

4.2.1 Sources Cuda de Sailfish

Sailfish est un outil de simulation de dynamiques des fluides basés sur LBM . Bien qu'il soit écrit en Python, Sailfish génère, compile et exécute un code Cuda pour optimiser l'exécution de ses simulations sur GPU.

Dans leur article, Januszewski and Kostur [13] décrivent son fonctionnement général, mais afin d'avoir un aperçu plus concret, une analyse du code est nécessaire.

Une brève analyse du code Python permet d'identifier que la fonction `get_code` génère le code Cuda. Celle-ci est appelée dans le corps de la fonction `_update_compute_code` du fichier `sailfish/subdomain_runner.py` du projet, qui compile ensuite le code généré. À l'aide d'un IDE comme PyCharm et d'un point d'arrêt, il est alors possible d'extraire le code Cuda au format ASCII avant qu'il soit compilé en langage machine.

Le code Cuda, plus long et plus compliqué qu'escompté, déclare de nombreuses fonctions dont plusieurs semblent avoir une utilité identique. Il est difficile d'identifier quelle portion du code est utilisé, et de quelle façon sans une analyse approfondie du fonctionnement de Sailfish.

Il n'a pas conséquent pas été particulièrement exploité.

4.2.2 Code « historique » de Sailfish

Le projet disponible sur le dépôt Github de Sailfish offre une intéressante implémentation, dans le dossier `/historical`. Ces sources ne font pas partie du reste du projet, mais implémentent de façon simple une simulation D2Q9 avec LBM sur GPU et produit des performances respectables. On y observe les pratiques détaillées dans les sous-section qui suivent.

Utilisation des registres

Les populations sont conservées dans la mémoire globale, à disposition de tous les cœurs du GPU ainsi que du CPU à travers un `cudaMemcpy`. Toutefois, le temps d'accès à ce type de mémoire est très lent. Plutôt que d'accéder aux populations sur la mémoire globale lors de chaque calcul, celles-ci sont copiées dans des variables locales du *kernel*. Ces variables, qui se situent dans les registres du GPU, le type de mémoire le plus rapide, sont alors utilisées dans les calculs et sont finalement copiées dans la mémoire globale à la fin du *kernel*. Cette méthode réduit sensiblement le nombre d'accès à la mémoire globale et augmente ainsi nettement les performances.

Structure de donnée des populations

Les populations sont conservées dans 9 tableaux différent, soit un par direction, de la taille du domaine. L'arrangement mémoire suit par conséquent le modèle SoA.

Calcul des populations

Plutôt qu'utiliser une boucle pour itérer sur les différentes directions et en calculer de façon générique leur état (comme l'`equilibrium`), celles-ci sont dépliées pour effectuer un à un le calcul adéquat.

Streaming coalesced

Le *streaming* copie les directions de la population calculée par un cœur GPU dans les populations adjacentes qui résident en mémoire globale. L'accès à cette mémoire est lent, mais peut être optimisé par des accès contigus et aligné sur 16 octets. On dit de ce schéma d'accès à la mémoire qu'il est « *coalesced* ».

Les *threads* au sein d'un bloc sont activés par *warp* de 32 sur l'axe des *x*. Ces *threads*, qui accèdent à la mémoire globale lors du *streaming*, doivent chacun y écrire une à une les valeurs des directions calculées qui résident dans leurs registres, soit 32 doubles par direction. La mémoire globale est accédée par transaction de 32, 64, 128 ou 256 octets. Par conséquent, si les directions des 32 populations (qui occupent 256 octets) accédées par les *threads* sont contiguës en mémoires, seule une transaction en mémoire globale est nécessaire.

Les figures 4.4 à 4.9 illustrent le *streaming* d'un bloc de 4 *threads* sur un domaine 12×3 . Sur la figure 4.4, les *threads* utilisent un schéma d'accès où les directions sont propagées aux populations adjacentes directement en mémoire globale. L'arrangement en mémoire des directions sur les populations accédées est illustré par la figure 4.5. On observe sur la figure 4.6 que l'écriture des valeurs pour les directions nord et sud est naturellement alignée. Ce n'est toutefois pas le cas pour les autres directions puisqu'elles impliquent un décalage en *x* de -1 à l'est et $+1$ à l'ouest. Leur accès n'est par conséquent pas *coalesced*.

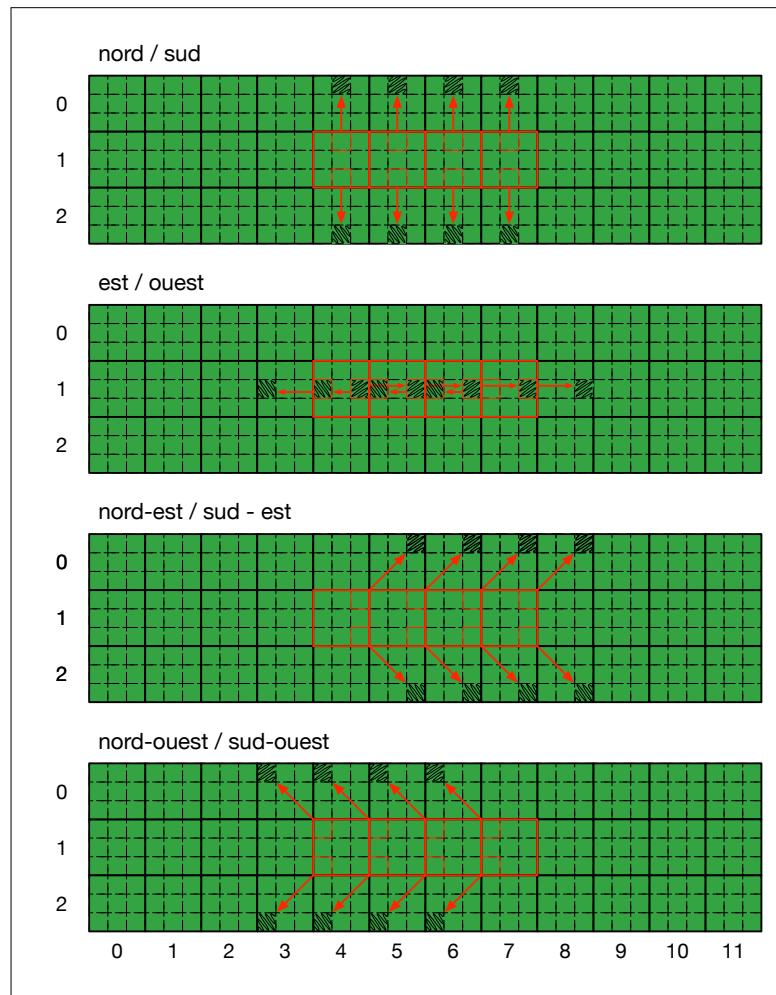
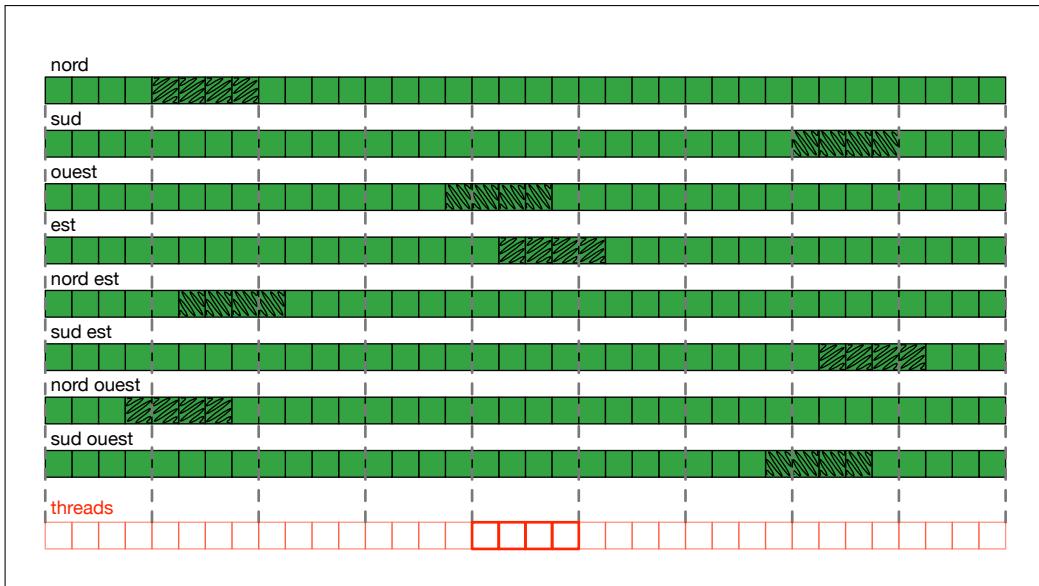
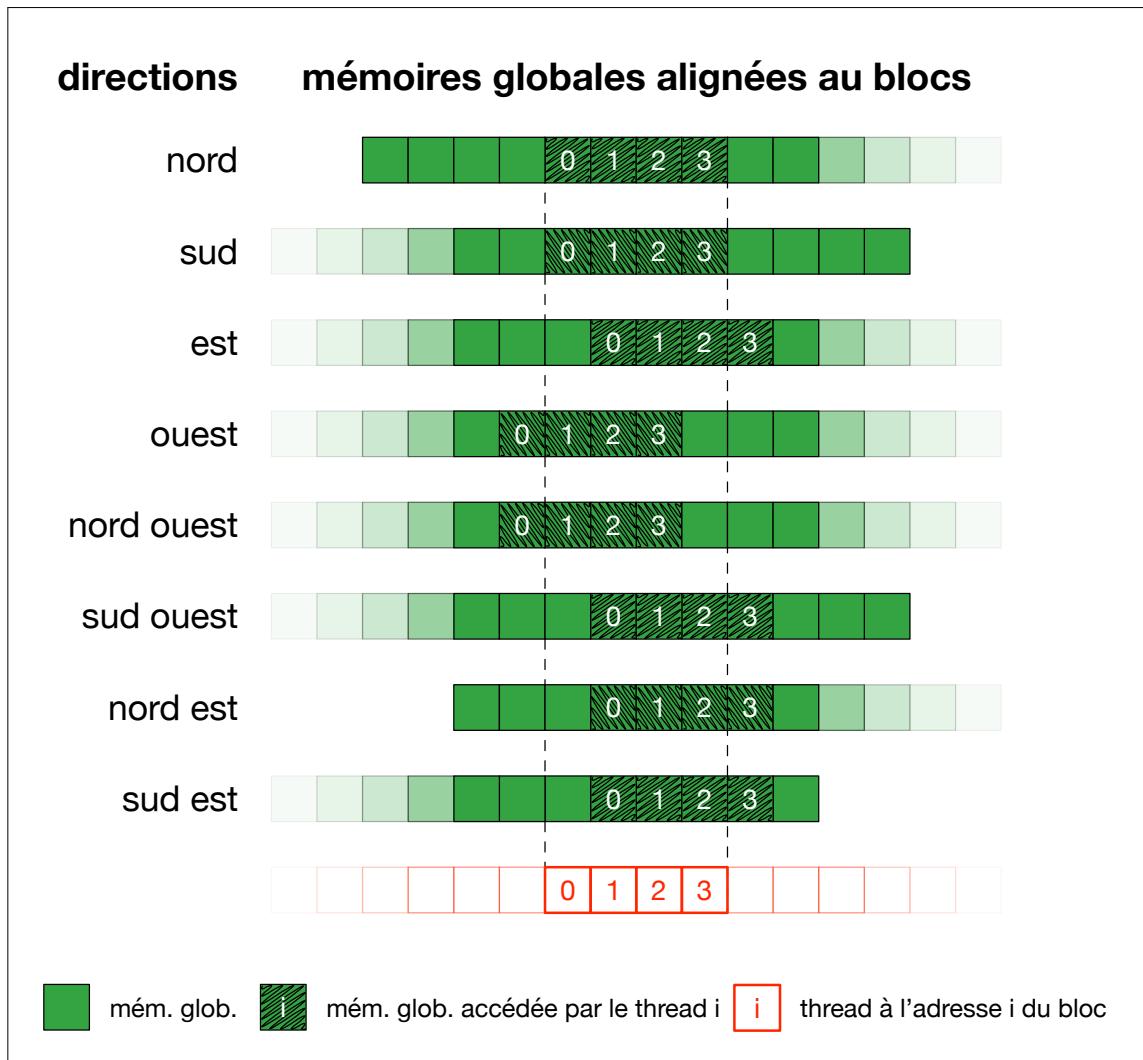


FIGURE 4.4 – *Streaming uncoalesced* d'un bloc de quatre *threads*

FIGURE 4.5 – Arrangement en mémoire des populations du *streaming* de la figure 4.4FIGURE 4.6 – Accès mémoire désaligné des *threads* lors du *streaming* de la figure 4.4

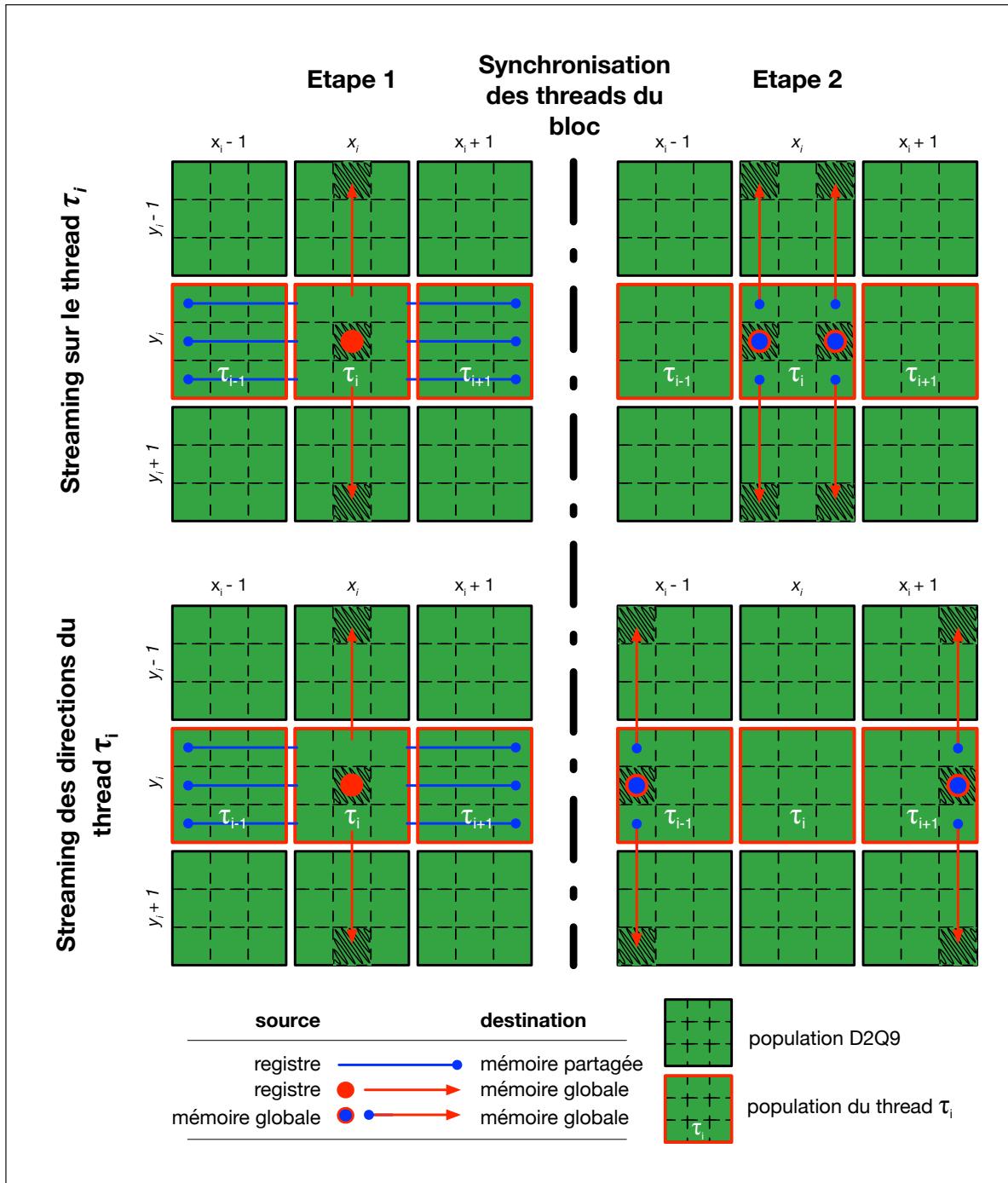


FIGURE 4.7 – Stratégie de *streaming coalesced* par réalignement des populations en mémoire partagée

Pour remédier au désalignement inhérent à la position de la population calculée par chaque *thread*, une stratégie en deux étapes est utilisée pour écrire les directions en mémoire globale (figure 4.7).

1. Dans un premier temps, au lieu de les propager directement en mémoire globale, les directions nord-ouest, ouest et sud-ouest de chaque *thread* au sein d'un bloc sont écrites en mémoire partagée sur l'indice du *thread* de gauche ($x - 1$) et les directions nord-est, est et sud-est sur l'indice du *thread* de droite ($x + 1$).
2. Dans un second temps, chaque *thread* cherche en mémoire partagée les directions inscrites à son indice et les propage en mémoire globale aux populations en dessus ($x - N_x$) et en dessous ($x + N_z$).

Une barrière de synchronisation entre les deux étapes assure que chaque *thread* a bien calculé et inscrit les directions en mémoire partagée, à l'attention de ses voisins, avant de procéder à la seconde étape.

La figure 4.8 illustre ce mécanisme exécuté simultanément sur chaque *thread* d'un bloc et la figure 4.9 l'alignement des indices des accès en mémoire globale. On constate avec cette stratégie des accès *coalesced*, à l'exception de ceux des *thread* au bord du bloc. En effet, ces derniers n'ont pas accès à la mémoire partagée du *thread* adjacent qui se trouve dans un bloc différent. Ils doivent par conséquent propager certaines directions directement en mémoire globale.

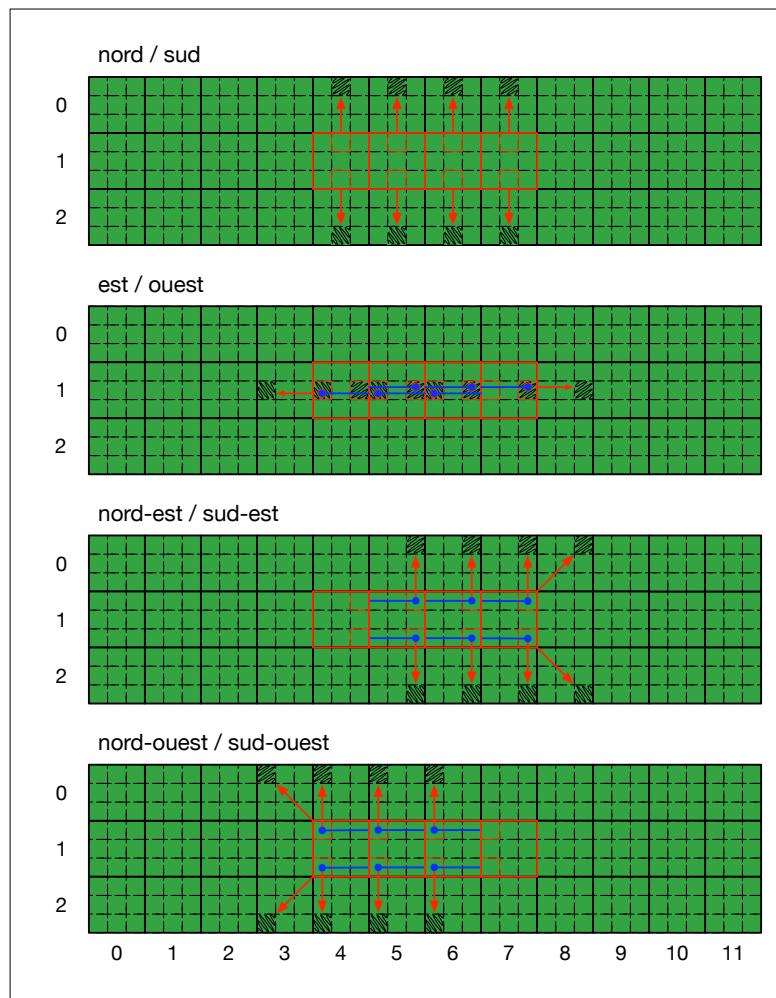


FIGURE 4.8 – *Streaming coalesced* d'un bloc de quatre *threads*

Les directions nord et sud, déjà alignées, sont directement propagées en mémoire globale, indépendamment lors de la première ou seconde étape.

L'exemple utilisé illustre des blocs de 4 *thread* par souci de lisibilité. En pratique, ceux-ci sont plus grands (32 ou 64) afin de réduire ainsi le nombre d'accès *uncoalesced*.

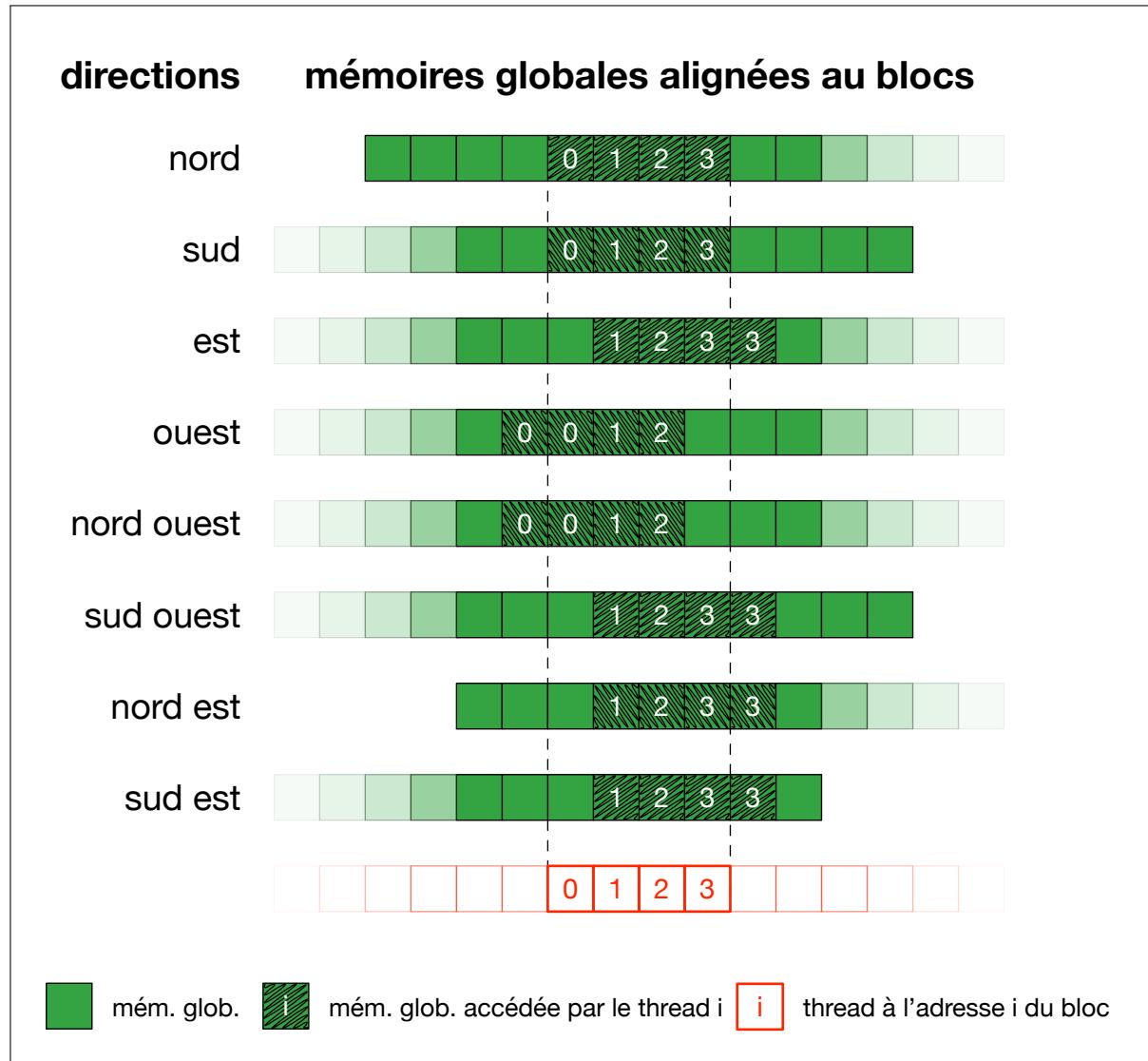


FIGURE 4.9 – Accès mémoire aligné des *threads* lors du *streaming* de la figure 4.8

Toutefois, Obrecht et al. [24] cherchent à éviter l'usage de la mémoire partagée plutôt qu'éviter à tout prix les accès *uncoalesced*. En effet, bien que cette stratégie se montre efficace sur les anciennes générations de GPU, le gain en performance semble nettement moindre sur les plus GPU, qui réduisent le coût des accès *uncoalesced* à l'aide de la mémoire cache.

Un certain nombre de travaux montrent que les accès *uncoalesced* en lecture sont légèrement moins couteux que les accès en écriture et qu'il est plus intéressant de profiter de cette propriété que d'utiliser la mémoire partagée. En effet, les accès à cette mémoire, pour y copier puis y lire les populations, bien qu'ils soient rapides, ne sont pas gratuits pour autant.

4.2.3 Différences entre calculs sur CPU et GPU

Cuda supporte la norme IEEE 754 pour les calculs avec nombre flottants [35]. Toutefois, dans la première implémentation Cuda de LBM , le protocole de test (décrit en section 4.1.1) a montré des différences sur certains calculs réalisés par les codes CPU.

Pour les mesurer, un code LBM 2D CPU et un code GPU ont été implémentés et leurs résultats comparés. La figure 4.10 illustre la différence moyenne entre les valeurs calculées sur CPU et GPU pour l'ensemble des populations f_{in} de chaque itération.

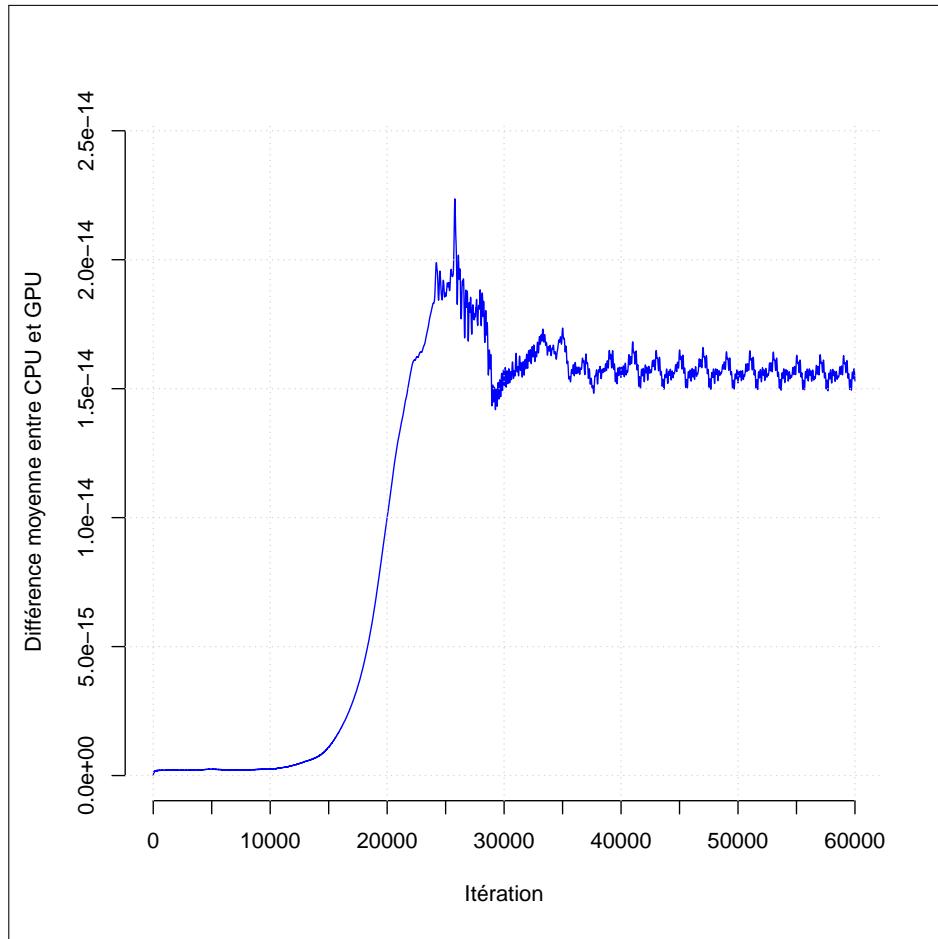


FIGURE 4.10 – Différences moyennes (par itération) entre les calculs sur CPU et GPU

L'écart entre les résultats observés sur CPU et GPU suit les étapes de la simulation réalisée (fluide autour d'un cylindre) illustrée par les figures 4.1 :

- entre 0 et environ 12000 itérations, une trace se forme derrière le cylindre et l'écart moyen reste faible et stable (quinze zéros après la virgule) ;
- entre 15000 et 20000 itérations, la trace commence à osciller et l'écart augmente brutalement (treize zéros après la virgule) ;
- à la 25792^{ième} itération, l'oscillation commence à se stabiliser et l'écart atteint son maximum (toujours treize chiffres après la virgule) ;
- dès environ 36000 itérations, l'oscillation est stabilisée (comme le montrent les figures 4.1(f) et 4.1(j), le même motif réapparaît toutes les 4000 itérations environ) et les écarts avec.

Cette différence entre certains résultats sur CPU et GPU est le fruit d'une optimisation du compilateur Cuda. Sur CPU, lorsqu'une opération du type $r = x \times y + z$ est rencontrée, le processeur calcule d'abord $r_{xy} = x \times y$ puis $r = r_{xy} + z$. Deux opérations sont donc réalisées, ce qui implique deux potentiels arrondis dans le cas des nombres flottants. Sur GPU, le compilateur optimise ce type de calculs [36] avec la fonction Fused Multiply-Add (FMA) qui réalise $x \times y + z$ en une seule opération. Le calcul est ainsi plus rapide et précis (puisque un seul arrondi est effectué).

Comme le souligne la documentation de Cuda, cette optimisation n'est pas forcément souhaitable dans certaines circonstances et peut être évitée avec l'utilisation de fonctions intrinsèques [36, 37].

Cette précaution force le GPU à effectuer les multiplications et additions indépendamment et permet de s'assurer que les calculs réalisés sur CPU et GPU produisent les mêmes résultats.

4.2.4 Passage de la 2D à la 3D et bibliothèque `lbmcuda`

De nombreuses implémentations 2D ont été réalisées. Elles ont d'abord été adaptées des portages en C, puis inspirées par le code historique de Sailfish, jusqu'à la dernière (nommée `lbm_opt2`) qui a fini par atteindre des performances acceptables (450 MLUPS sur une Nvidia GeForce GT 750M).

Pour passer du modèle D2Q9 à D3Q19, les 10 populations manquantes sont ajoutées en mémoire globale et sont intégrées aux calculs de la collision ainsi qu'au *streaming*. À ce stade, l'implémentation utilisait encore la mémoire partagée pour le *streaming*. Suite à l'ajout des populations, l'implémentation est testée avec et sans ce mécanisme. Mais comme l'évoque la section 4.2.2, il s'avère que les *GPU* récents ne pâtissent pas autant des accès *uncoalesced* en raison d'optimisation automatique à travers l'utilisation de la mémoire cache.

D'autres optimisations, proposées par Januszewski and Kostur [13] et Tran et al. [31] améliorent les performances :

- **Registres** : Les registres sont utilisés par certaines variables locales du *kernel*. C'est le type de mémoire le plus rapide, mais leur nombre est restreint. En dépasser la limite entraîne un débordement en mémoire local, moins rapide, et par conséquent une perte de performance. Hélas, le passage de D2Q9 à D3Q19 augmente significativement le nombre de variables locales et par conséquent l'utilisation des registres. Ne pas précalculer des valeurs utilisées plusieurs fois, et qui devraient être ainsi stockées dans des variables intermédiaires, mais les recalculer à chaque fois semble être une « optimisation » contre-intuitive, mais réduit le nombre de registres et par conséquent contribue à améliorer les performances.
- **Cache L1** : Januszewski and Kostur [13] relèvent que les accès à la mémoire locale dus aux dépassements du nombre de registres passent par la cache L1 et que plus sa part inutilisée est grande et moins la mémoire globale est mise à contribution. Ils proposent ainsi d'augmenter la taille de la cache et de désactiver la cache L1 pour les accès à la mémoire globale. La première opération est accomplie par `cudaDeviceSetCacheConfig` :

```
if ( cudaDeviceSetCacheConfig (cudaFuncCachePreferL1) != cudaSuccess)
    fprintf(stderr, "cudaFuncSetCacheConfig\u201cfailed\n");
```

et la seconde en compilant le code Cuda avec les arguments `-Xptxas -dlcm=cg`.

- **Divisions par des multiplications** : Les multiplications sont un peu plus rapides que les divisions. Remplacer une division par une multiplication, lorsque cela est possible, peut ainsi améliorer les performances. Si plusieurs calculs requièrent une division par la même valeur (comme avec *rho* dans la fonction *macroscopic*), l'inverse de sa valeur peut être calculé initialement puis être multiplié par la suite pour éviter plusieurs divisions. Cette méthode a toutefois l'inconvénient de modifier parfois très légèrement le résultat du calcul en raison d'un arrondi différent avec les valeurs flottantes et ajoute une variable locale intermédiaire.

Finalement, en vue de son intégration à Palabos et afin d'en simplifier l'utilisation, l'implémentation Cuda est portée dans une bibliothèque dynamique nommée **lbmcuda** que le projet **lbm_simple_lbmcuda** utilise pour implémenter la même configuration que **lbm_simple** en C et Python. La bibliothèque implémente les principales fonctions suivantes :

- **lbm_simulation_create** : Création d'un domaine de simulation en spécifiant ses dimensions ainsi que la valeur d'omega.
- **lbm_simulation_update** : Mets à jour les populations du domaine sur une itération.
- **lbm_lattices_read** et **lbm_lattices_write** : Transfert de l'intégralité des populations du domaine respectivement depuis et vers le GPU.
- **lbm_lattices_read_subdomain** et **lbm_lattices_write_subdomain** : Transfert partiel du domaine.
- **lbm_read_palabos_subdomain** et **lbm_write_palabos_subdomain** : Transfert partiel du domaine avec réorganisation mémoire selon l'ordre utilisé par Palabos (voir section 4.3.3).

4.3 Intégration à Palabos

4.3.1 Co-processeurs

Fonctionnement

Il est possible de demander à Palabos de découper un domaine de recherche en plusieurs sous-domaines afin de les calculer indépendamment les uns des autres. Comme l'illustre la figure 4.11, un sous-domaine possède une portion unique du domaine de recherche (vert) ainsi qu'une marge extérieure (bleu) qui s'entrelace avec les zones centrales des sous-domaines adjacents. Son rôle est de propager les valeurs calculées par les sous-domaines voisins pour calculer la zone centrale (verte).

Ce découpage permet 1. de distribuer la simulation sur différents nœuds (à l'aide d'MPI) et 2. de déléguer les calculs à un CP spécialisé.

Un CP (ou accélérateur) est un module capable de calculer l'état d'un sous-domaine, normalement pris en charge par Palabos sur CPU, mais cette fois confié par celui-ci. Son objectif est d'accélérer la vitesse des calculs en se spécialisant sur un certain type de sous-domaine. L'utilisation d'un CP est illustrée dans le programme d'exemple **coProcessor**¹ fourni dans Palabos où l'on distingue cinq étapes importantes.

1. **Initialisation des CP** : On commence par assigner un CP aux sous-domaines désignés d'un espace de recherche découpé au préalable.

1. dossier : <palabos root>/examples/codesByTopic/coProcessor

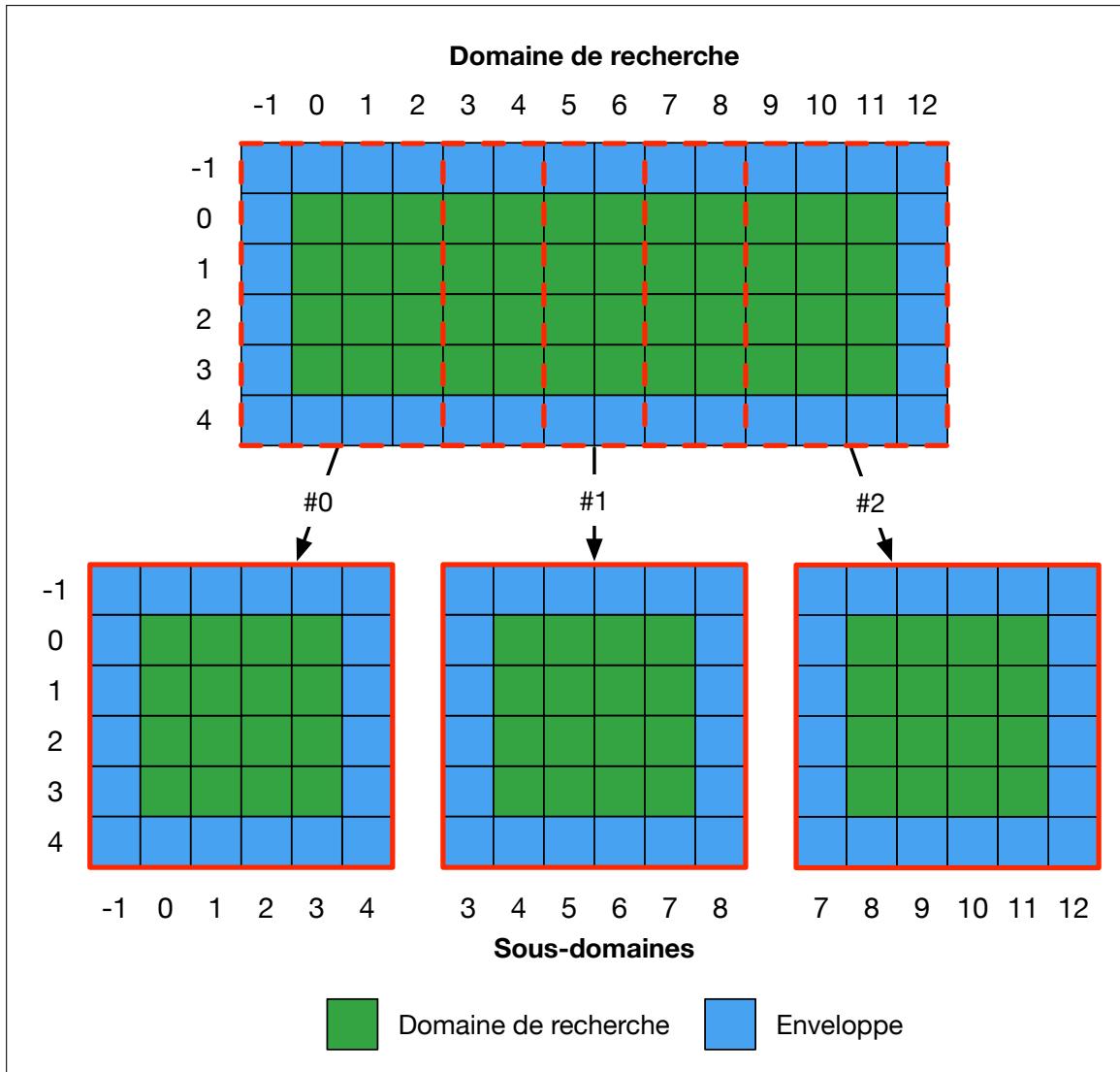


FIGURE 4.11 – Découpage d'un domaine 12x4 en trois sous-domaines par Palabos

2. **Transfert CPU à CP** : À ce stade, seul Palabos connaît l'état de l'espace de recherche. Par conséquent, on transfère les valeurs des populations dans les sous-domaines délégués aux CP.
3. **Calcul du Collide & Stream (CS)** : On calcule *une* itération avec LBM sur l'ensemble des sous-domaines (avec Palabos et les CP).
4. **Transfert CP à CPU** : À ce stade, contrairement à la seconde étape, Palabos ignore l'état des sous-domaines assignés au CP, que seuls eux connaissent. On transfère cette fois ces valeurs dans le sens opposé.
5. **Duplication des chevauchements** : Les enveloppes des sous-domaines se chevauchent. Il faut par conséquent dupliquer les valeurs d'un sous-domaine, calculé à l'étape précédente, dans les enveloppes des sous-domaines voisins.

Au terme de ce processus, une itération complète est ainsi effectuée. Pour en calculer une autre, on recommencera depuis la seconde étape.

Implémentation

Un co-processeur n'est autre qu'une classe qui hérite de `CoProcessor3D`. Pour fonctionner, elle doit implémenter les principales méthodes suivantes. Elles sont respectivement appelées lors des quatre premières étapes mentionnées précédemment.

- `addDomain` : Associe un ID à un sous-domaine attribué au CP et en définit les dimensions.
- `send` : Envoie l'état du sous-domaine, de la mémoire de Palabos vers celle du CP.
- `collideAndStream` : Calcule une itération sur un sous-domaine du CP.
- `receive` : Envoie l'état du sous-domaine, de la mémoire du CP vers celle de Palabos.

C'est ce mécanisme de CP qui est utilisé pour donner à Palabos la capacité de calculer un sous-domaine sur GPU.

4.3.2 Transfert mémoire entre Palabos et un CP

À chaque itération, il est nécessaire de transférer l'état des populations des sous-domaines deux fois :

1. **Avant de calculer les populations des CP** : depuis Palabos vers les CP, pour propager les populations calculées par les sous-domaines adjacents.
2. **Après les avoir calculés sur les CP** : depuis les CP vers Palabos, pour propager leurs populations à l'itération suivante.

Lors de l'appel à méthode `send` et `receive`, Palabos transmet un espace mémoire contigu où les populations sont respectivement écrites et lues. Il est adressé en calculant l'index suivant :

$$i_{pal} = (x * N_z * N_y + y * N_z + z) * 19 + dir \quad (4.1)$$

avec la dimension N_y et N_z du sous-domaine en y et z et i_{pal} l'index de la population en $\{x, y, z\}$ avec pour direction dir (figure 4.12) :

dir_{center}	= 0	$dir_{northwest}$	= 5	dir_{east}	= 10	dir_{east}^{top}	= 15
dir_{west}	= 1	dir_{west}^{bottom}	= 6	dir_{north}	= 11	dir_{east}^{bottom}	= 16
dir_{south}	= 2	dir_{west}^{top}	= 7	dir_{center}^{top}	= 12	dir_{north}^{top}	= 17
dir_{center}^{bottom}	= 3	dir_{south}^{bottom}	= 8	$dir_{northeast}$	= 13	dir_{north}^{bottom}	= 18
$dir_{southwest}$	= 4	dir_{south}^{top}	= 9	$dir_{southeast}$	= 14		

Si le CP utilise la même stratégie d'adressage, une simple copie de la mémoire suffit. Dans le cas inverse, il faut considérer une étape de réorganisation de la mémoire.

La communication des populations entre le CPU et le GPU passe par la mémoire globale du GPU à l'aide de la fonction `cudaMemcpy`. Ce type de transactions doit être réduit au minimum, car il s'agit du type d'accès mémoire le plus lent dans le contexte des GPU. Hélas, Palabos requiert impérativement ces transactions à chaque itération, ce qui réduit les performances.

Toutefois, il n'est pas nécessaire de transférer l'intégralité du sous-domaine à chaque fois, à l'exception des cas suivants :

1. lors de la première itération, où il faut initialiser le sous-domaine avec les populations initiales depuis Palabos ;

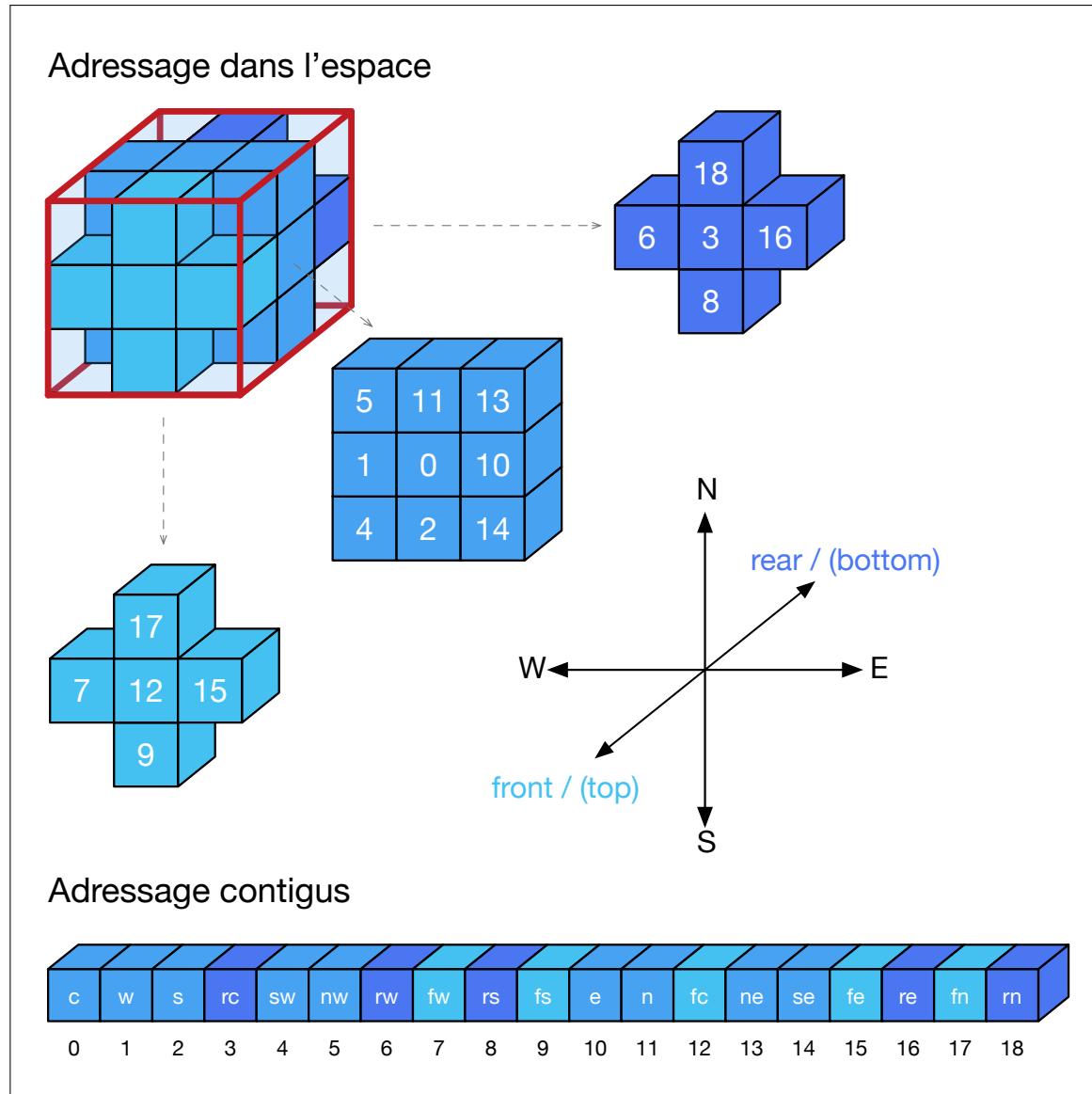


FIGURE 4.12 – Adressage d'une population sur Palabos

2. après une itération où l'on désire connaître l'état du domaine de recherche, où il faut rapatrier le sous-domaine.

En effet, comme l'illustre la figure 4.13, seules les communications de l'enveloppe extérieure et intérieure sont nécessaires. L'enveloppe extérieure lors de l'envoi (de Palabos au CP), pour permettre la propagation vers l'intérieur du sous-domaine lors du CS et l'enveloppe intérieure lors de la réception (du CP vers Palabos) pour mettre à jour les enveloppes extérieures des sous-domaines adjacents lors de la duplication des chevauchements (*Duplicate overlaps*) en préparation de l'itération suivante.

Un mécanisme est prévu par Palabos pour ne demander que le transfert de cette enveloppe. Lorsqu'il est utilisé, Palabos procède à un appel à `send` ou `receive` par face de l'enveloppe (figure 4.14), soit six appels. Bien que cela représente une multiplication par 6 du nombre d'appels (et par conséquent de l'overhead lié aux `cudaMemcpy`), cette méthode permet de réduire significativement l'espace mémoire à copier (figure 4.15).

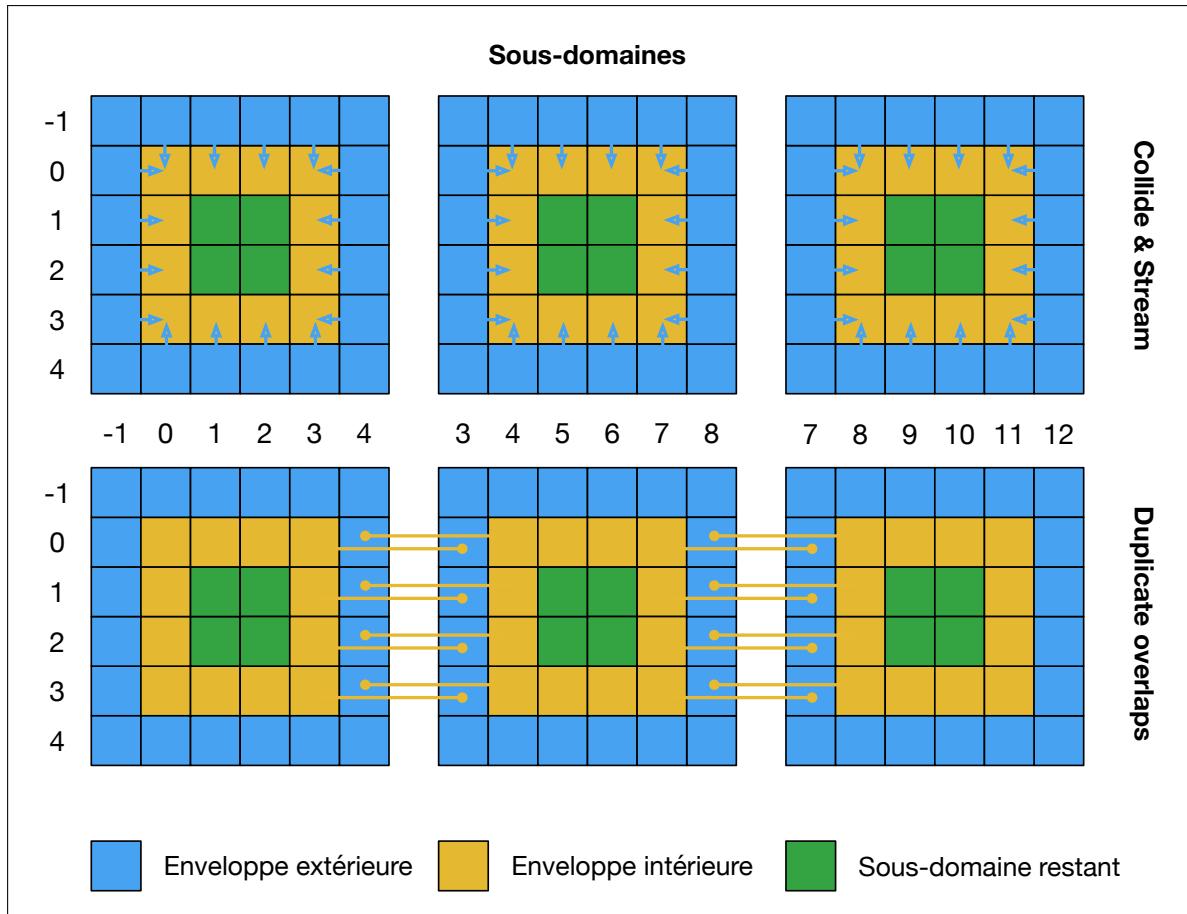


FIGURE 4.13 – Enveloppe extérieure et intérieure

4.3.3 Stratégies de réorganisation des données

Comme le souligne la fin de la section 4.3.2, lorsque l’adressage des données est différent entre Palabos et le CP, une étape de réorganisation doit intervenir lors du processus de transfert. Dans le cas présent, Palabos utilise l’arrangement mémoire AoS, tandis que l’implémentation GPU utilise l’arrangement SoA.

Deux stratégies de réorganisation ont été explorées :

1. **Réorganisation sur CPU et transfert** : Cette stratégie réarrange les données reçues ou à transférer sur le CPU dans l’ordre qu’utilise le GPU en mémoire globale et tente de profiter de la forme du sous-domaine transmis pour réduire le nombre d’appels à `cudaMemcpy` lorsque la mémoire est contiguë.

L’ordre dans lequel la réorganisation et le transfert ont lieu dépend de l’opération :

- **send** :
 - réorganisation mémoire sur CPU ;
 - transfert mémoire du CPU au GPU
- **receive** :
 - transfert mémoire du GPU au CPU
 - réorganisation mémoire sur CPU ;

Un transfert de l’ensemble du sous-domaine est simplement réalisé par 19 `cudaMemcpy` (un par tableau). Toutefois, il est impossible d’utiliser cette méthode telle quelle lors

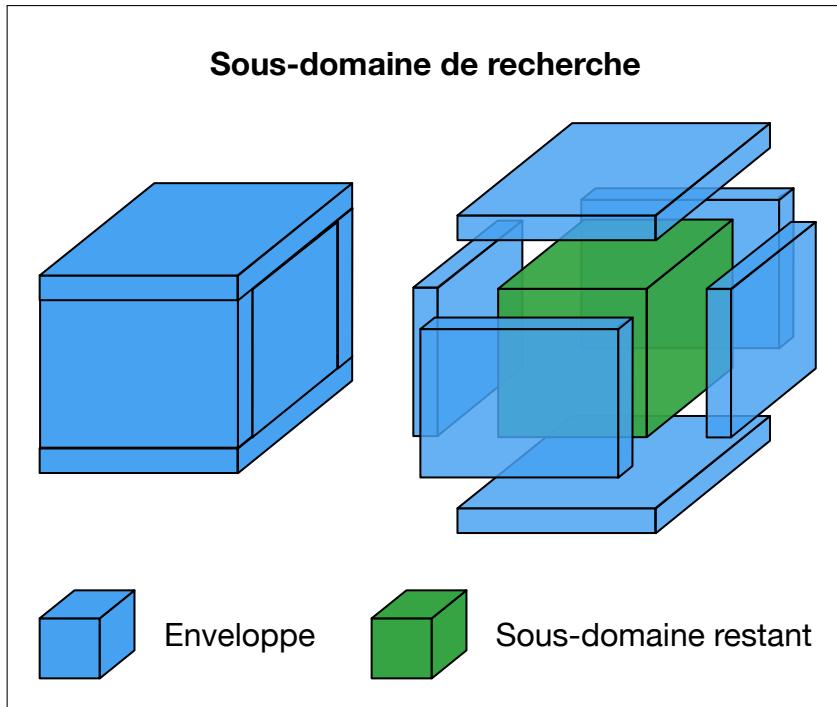


FIGURE 4.14 – Enveloppe Palabos 3D

d'un transfert partiel (pour une partie de l'enveloppe par exemple). En effet, comme l'illustre la figure 4.16, certaines faces ont un adressage contigu, tandis que d'autres non.

L'étape de transfert tire parti des faces dont l'adressage est contigu pour réduire le nombre d'appels à `cudaMemcpy`. Par conséquent, voilà comment seraient transférées les faces suivantes :

- XZ : un `cudaMemcpy` pour l'ensemble de la face et par direction (soit 19) ;
- XY : un `cudaMemcpy` par ligne et par direction (soit $4 \times 19 = 76$) ;
- YZ : un `cudaMemcpy` par population et par direction (soit $4 \times 4 \times 19 = 304$) ;

On constate que le nombre de `cudaMemcpy` devient très important sur les faces dont les indices ne sont pas consécutifs. De plus, ils ne transfèrent qu'une seule valeur à la fois. Une telle face, d'un domaine 100×100 , nécessite 190000 appels à `cudaMemcpy` pour être transférée. Cette stratégie offre par conséquent de très mauvaises performances.

2. **Transfert et réorganisation sur GPU** : Plutôt que réorganiser les données sur le CPU, cette stratégie utilise à cette fin le GPU et transfère d'un seul bloc les données entre le CPU et le GPU avec un unique `cudaMemcpy`. Là encore, l'opération dicte l'ordre dans lequel la réorganisation et le transfert à lieu ont lieu :

- **send** :
- (a) transfert mémoire du CPU au GPU
- (b) réorganisation mémoire sur GPU ;
- **receive** :
- (a) réorganisation mémoire sur GPU ;
- (b) transfert mémoire du GPU au CPU

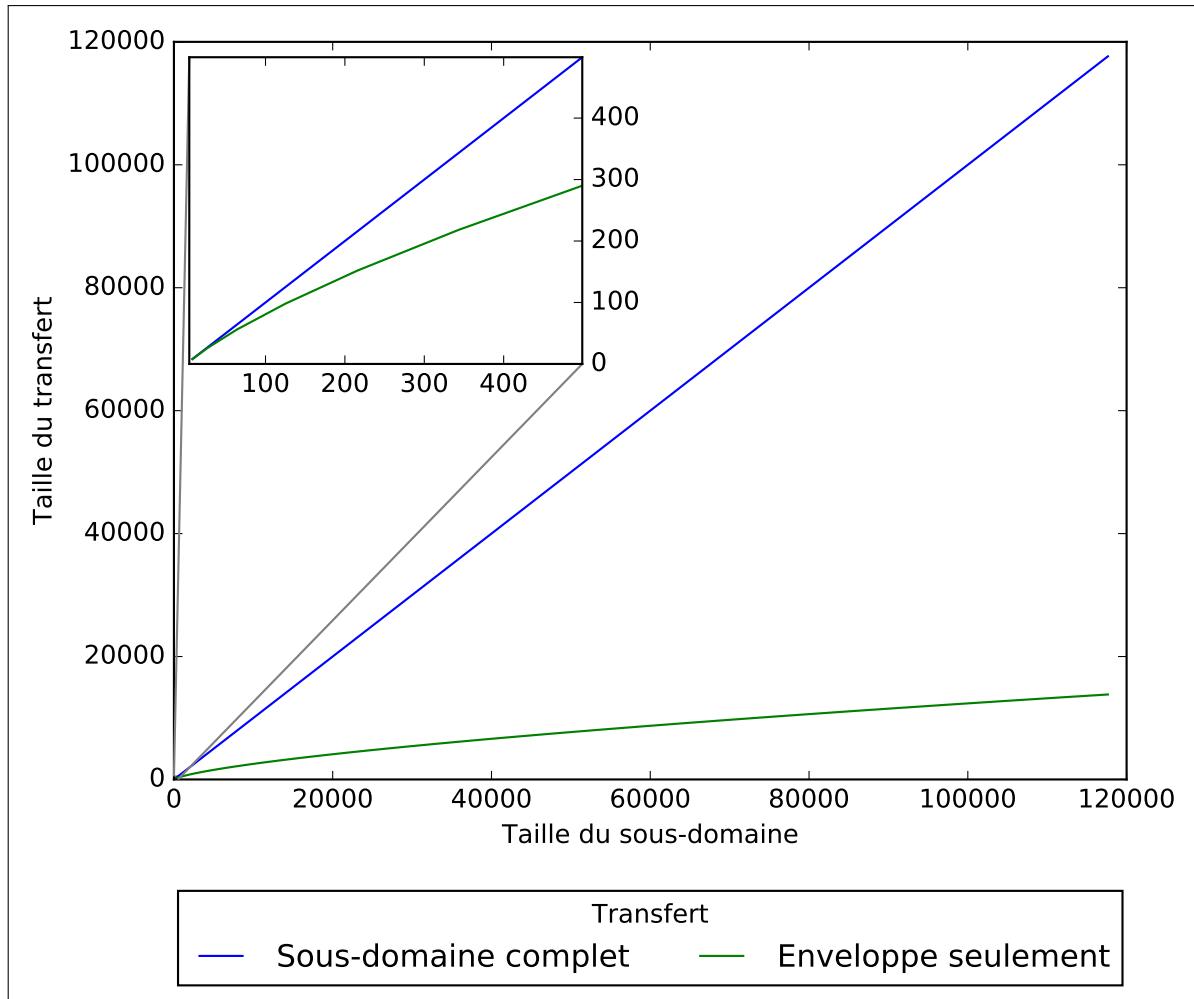


FIGURE 4.15 – Taille des transferts entre Palabos et un CP

On observe une inversion de l’ordre des opérations, par rapport à la stratégie précédente. Dans le cas du *send*, les données sont directement transférées au GPU, puis exécutent un *kernel* dédié à la réorganisation et la copie des données vers la structure de donnée utilisées par le *kernel* de calcul.

Dans le cas du *receive*, un autre *kernel*, dédié cette fois à la réorganisation et la copie des données depuis la structure de donnée de calcul, est exécuté avant que les données soient finalement transférés du GPU au CPU.

Cette méthode réduit à la fois considérablement le nombre d’appels à `cudaMemcpy` et profite de capacité de parallélisation du GPU pour accélérer la réorganisation des données. Elle présente ainsi de bien meilleures performances.

4.3.4 Simulation d’écoulement dans une cavité

Pour tester l’intégration à Palabos et en mesurer les performances, l’implémentation `cavity_benchmark` a été réalisée. À l’exception de quelques modifications (destinées à rendre les exécutions configurables notamment), leur code est essentiellement copié de l’exemple `coProcessor` fournit par Palabos.

Cette implémentation simule l’écoulement d’un fluide dans une cavité. Le domaine est cubique et découpé en vingt-sept sous-domaines. Il est possible de choisir les dimen-

sions du domaine central. Les autres domaines ajustent alors les leurs automatiquement. La figure 4.18 illustre le découpage et dimensionnement des sous-domaines pour deux exemples. Le premier pour un sous-domaine central dont les côtés mesurent le $1/3$ de ceux du domaine et $2/3$ pour le second.

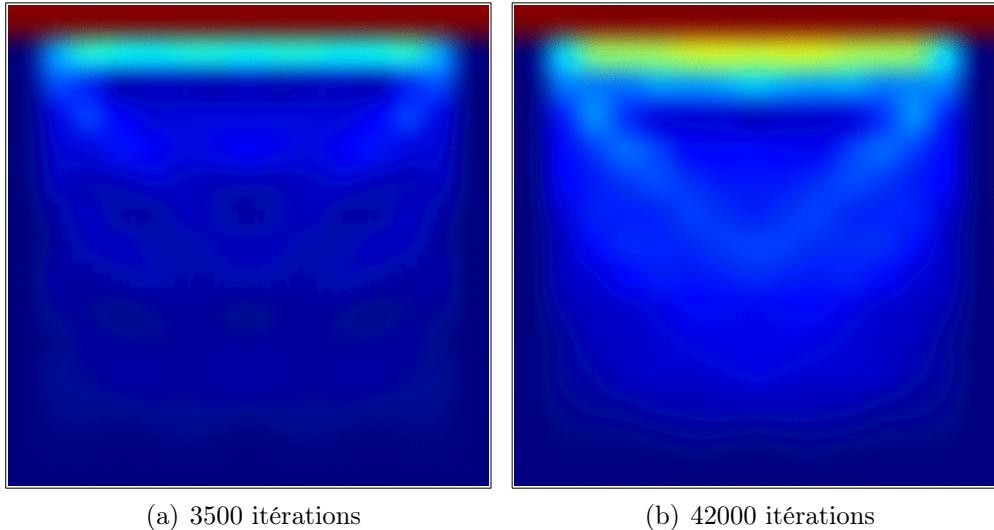


FIGURE 4.17 – Images générées par `cavity_benchmark` d'un écoulement dans une cavité

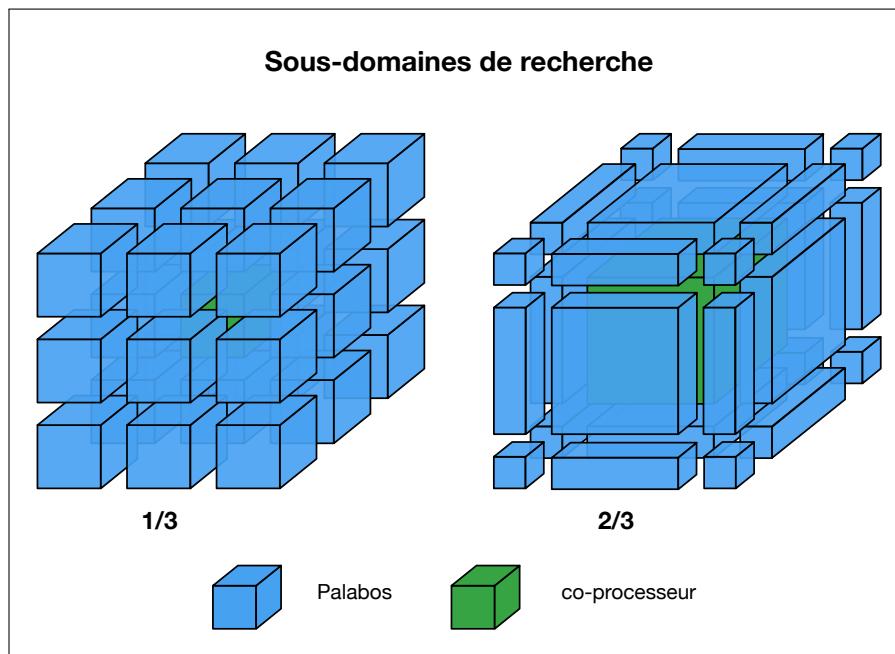
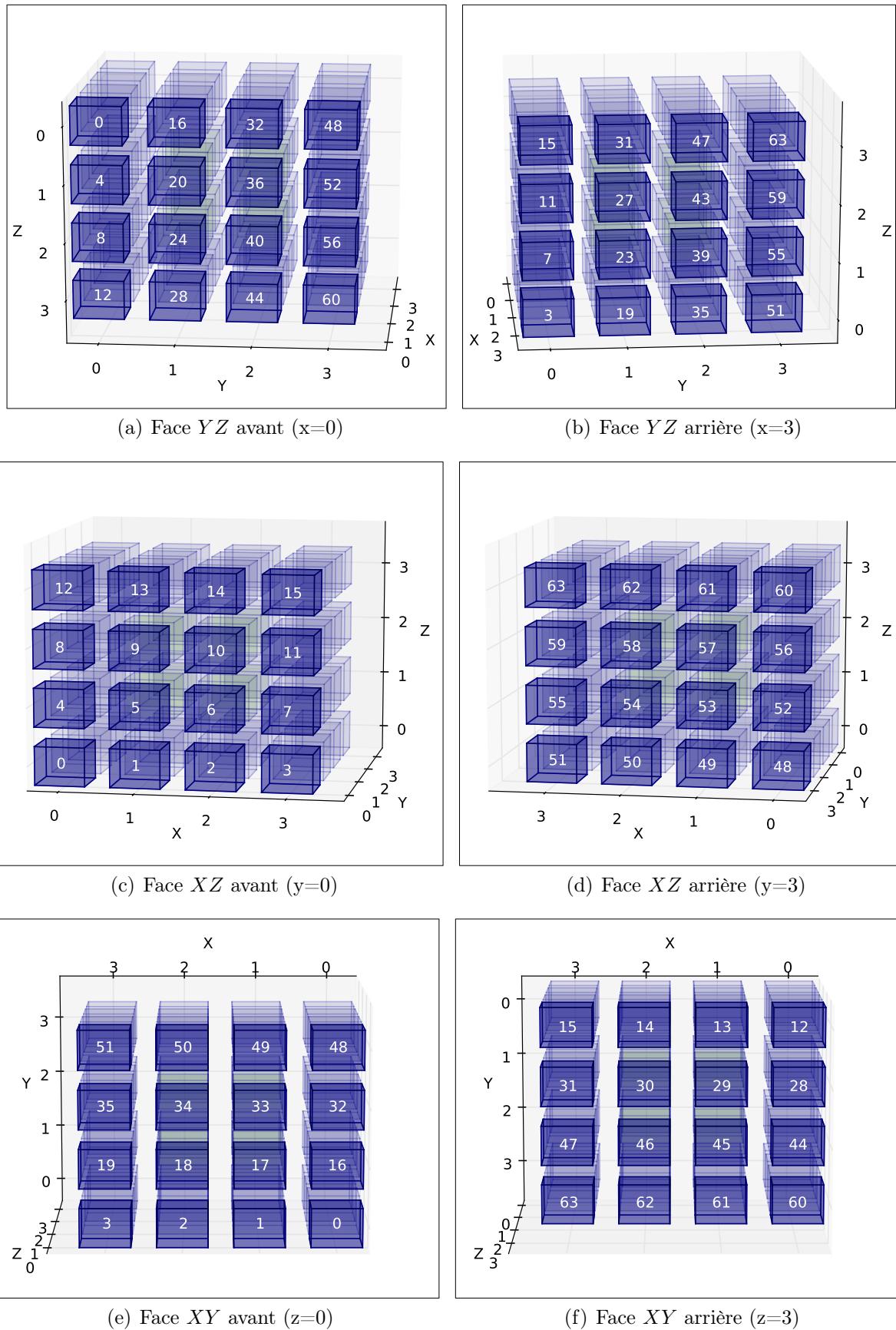


FIGURE 4.18 – Dimensionnement des sous-domaines de `cavity_benchmark`

Le sous-domaine central est confié au co-processeur GPU (ou à un co-processeur CPU à des fins de test si on le désire) tandis que les vingt-six autres qui l'entourent sont calculés sur le CPU par Palabos.


 FIGURE 4.16 – Adressage des populations dans un sous-domaine 4×4

Chapitre 5

Mesures de performances

5.1 Configuration des *benchmarks*

5.1.1 Simulations effectuées

Ce chapitre analyse les performances obtenues par l'implémentation GPU et hybride (Palabos) et les paramètres qui les affectent. Les *benchmarks* sont réalisés avec deux programmes différents qui utilisent la bibliothèque `lbmcuda` (l'implémentation Cuda) : `lbum_simple_lbmcuda` et `cavity_benchmark`.

Le premier simule le même type de problème que `lbum_simple` (voir section 4.1.2) avec ou sans transfert des populations à chaque itération afin de mesurer les performances sur GPU. Le second, décrit en section 4.3.4, simule un écoulement dans une cavité pour évaluer les performances d'une configuration hybride CPU/GPU avec Palabos.

5.1.2 Matériel utilisé

Les mesures sont effectuées sur les GPU et CPU listés dans la table 5.1 pour évaluer l'impact sur les performances de plusieurs types de matériel. Le GPU utilisé est indiqué sur les mesures ou dans la section qui les présentent. Si rien n'est indiqué, c'est le GPU Pascal qui est utilisé.

5.1.3 Limitations mémoire pour les simulations LBM

Les dimensions du domaine simulé sont limitées par la mémoire du CPU et du GPU, les simulations étant très gourmandes en mémoire. En effet, les populations du domaine sont allouées au moins une fois sur le CPU pour les transferts et surtout trois fois en mémoire globales sur le GPU : une fois avec f_{palabos} pour les transferts et réorganisations mémoire entre CPU et GPU (section 4.3.3), puis deux pour la simulation avec f^{in} et f^{out} . Chaque population occupe f_{size} octets en mémoire :

$$f_{\text{size}} = 8 \times Nx \times Ny \times Nz \times 19 \quad (5.1)$$

avec 8 étant le nombre d'octets d'un nombre flottant en double précision, Nx , Ny et Nz les dimensions du domaine et 19 le nombre de directions d'une population D3Q19. Ainsi, les trois populations d'une simulation d'un domaine cubique de dimensions 256^3 occupent $8 \times 256^3 \times 19 \times 3$ octets soit environ 7.65 Go en mémoire globale.

	Pascal	Titan	Tesla
Carte NVIDIA	TESLA P100	TITAN X	TESLA M2090
Architecture	Pascal	Pascal	Tesla
Fréquence maximum (MHz)	1480	1531	1300
Type de mémoire	HBM2	GDDR5	GDDR5
Bandé passante mémoire (Go/s)	549	480.3	177.6
Perf. simple précision (Tflops)	10609	10157	1331.2
Perf. double précision (Tflops)	5304	317	665.6
Mémoire globale (MBytes)	12194	12190	5301
Nombre de SM	56	28	16
SP par Streaming Multiprocessor (SM)	64	128	32
CPU Intel® Xeon®	E5-2630 v4	E5-2643 v3	X5650
Fréquence du CPU (MHz)	2200	3400	2670

TABLE 5.1 – Spécifications du matériel [40] utilisé pour les *benchmarks*

Si l'on y ajoute les vélocités, dont la transmission est également prévue, on peut leur additionner ν_{size} octets :

$$\nu_{\text{size}} = 8 \times Nx \times Ny \times Nz \times 3 \quad (5.2)$$

avec 3 le nombre de dimensions en D3Q19 ; soit encore environ 402.6 Mo de plus, pour arriver à un total d'environ 8.05 Go pour un domaine de dimensions 256³.

5.2 *Benchmarks* des GPU

5.2.1 Bande passante et débits de transfert

Cette section présente les mesures des débits d'accès à la mémoire globale et des transferts entre CPU et GPU. Elles ont été collectées avec la méthode proposée par [39].

Les premières, illustrées sur la figure 5.1, montrent les bandes passantes effectives mesurées (l'accès à la mémoire globale sur le GPU). La progression en fonction de la quantité des données copiées est due au fait que le nombre de *threads* capables de copier les données en parallèle évolue jusqu'à ce que la limite de *threads* soit atteinte. Les mesures confirment les bandes passantes théoriques de la table 5.1, les valeurs effectives étant usuellement entre 75% et 85% des bandes passantes théoriques.

Les secondes mesures, illustrées par la figure 5.2, montrent les débits de transfert entre le GPU et le CPU avec `cudaMemcpy` dans un sens et dans l'autre. On observe la même tendance avec Pascal et Titan et un débit nettement inférieur et plus instable avec Tesla.

5.2.2 Overhead de lancement d'un *kernel* et latence des transferts

Comme l'illustre la table 5.2, l'exécution d'un *kernel* sur le GPU ou d'un transfert de données avec `cudaMemcpy` n'est pas immédiate. Pour mesurer la latence du démarrage d'un

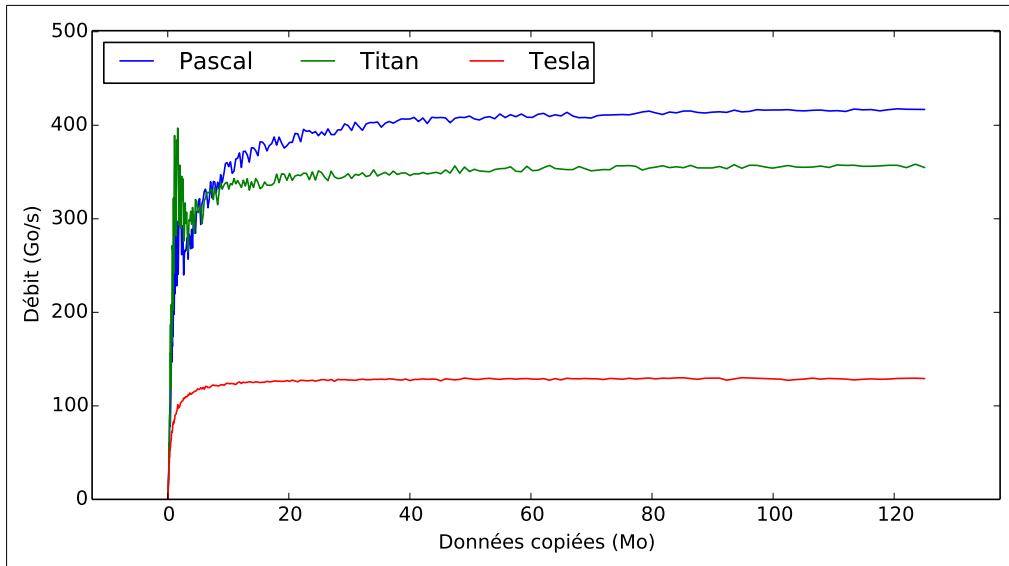


FIGURE 5.1 – Bande passante effective

kernel, on mesure simplement le temps d'exécution d'un *kernel* vide (sans aucune instruction). Pour la latence d'un transfert opéré avec `cudaMemcpy`, on ne peut pas simplement faire une copie de 0 octet.

Si on considère le temps de transfert $T_{trans} = T_{lat} + \frac{N}{bw}$, avec T_{lat} la latence, N la quantité de données à transférer et bw la bande passante, comme le propose Albuquerque et al. [1], il est possible de déduire T_{lat} . On mesure le temps d'exécution T_{trans} pour un transfert avec N données puis un second temps de transfert T'_{trans} mesuré pour $m \cdot N$ données et on calcule ensuite T_{lat} ainsi :

$$T_{lat} = \frac{m \cdot T_{trans} - T'_{trans}}{m - 1} = \frac{\left(m \left(T_{lat} + \frac{N}{bw} \right) - T_{lat} + \frac{m \cdot N}{bw} \right)}{m - 1} = \frac{(m - 1)T_{lat}}{m - 1} \quad (5.3)$$

	Pascal	Titan	Tesla
Lancement d'un Kernel [μs]	4.4	3.552	5.07
Lancement d'un transfert [μs]	30 – 50	46 – 75	580 – 640

TABLE 5.2 – Latences du *kernel* et des transferts avec `cudaMemcpy`

5.3 *Benchmarks* de l'implémentation GPU

5.3.1 Ordre des indices

Les populations sont conservées dans 19 tableaux en mémoire globale (un par direction). Comme sur CPU, l'ordre des indices x, y et z a un impact important sur les performances pour des questions de cache. En effet, les populations sont accédées à une adresse x différente par chaque *thread*, c'est donc l'indice qui change le plus rapidement. Pour profiter au mieux des mécanismes de cache, il faut que les zones mémoire accédées consécutivement soient très proches les unes des autres.

La formule suivante calcule un indice à une dimension, avec i , j et k étant chacun l'un des indices x , y ou z et I , J et K la taille de la dimension de l'indice.

$$\text{indice} = (i + I) \bmod I + \left((j + J) \bmod J + (k + K) \bmod K \times J \right) \times I \quad (5.4)$$

Par conséquent, $J \times I$ individus séparent k et $k + 1$, J individus séparent j et $j + 1$ mais i et $i + 1$ eux sont voisins en mémoire. Il faut par conséquent attribuer i à l'indice qui change le plus rapidement, soit x . Cette conclusion est confirmée par les mesures illustrées par la figure 5.3. Plus les individus indexés par x sont distants, plus les performances se dégradent.

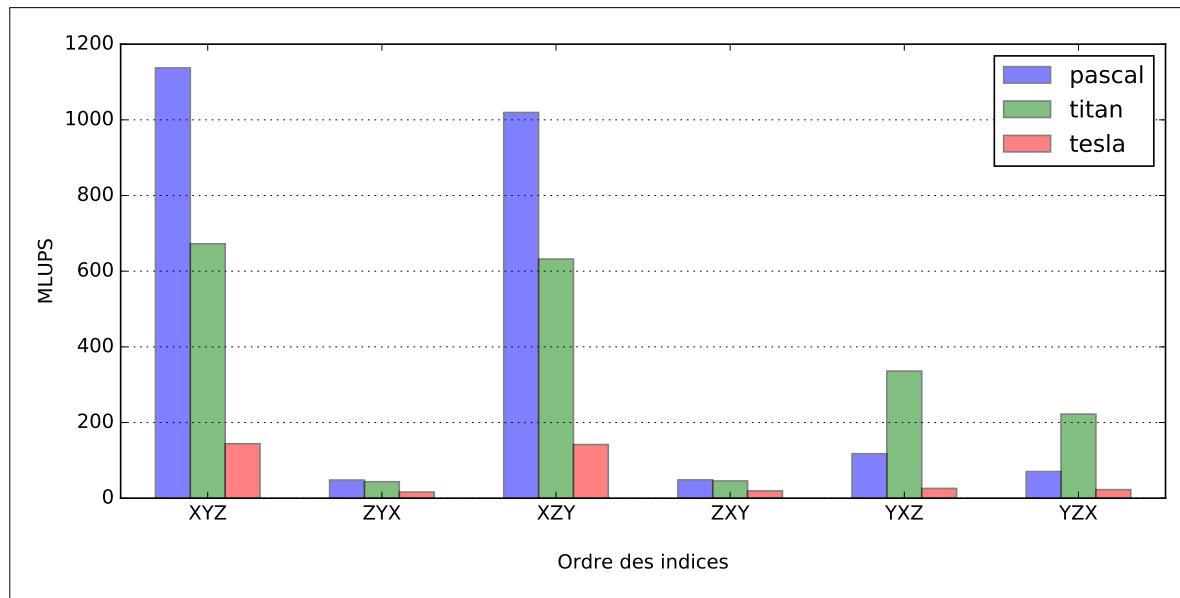


FIGURE 5.3 – Performances en fonction de l'ordre des indices

Les meilleures performances sont atteintes avec $i = x$, $j = y$ et $k = z$. Cependant, les *benchmarks* des sections suivantes utilisent $i = x$, $j = z$ et $k = y$. Ce choix est dû à une mesure, effectuée précédemment sur une carte Titan, qui suggérait à tort un très léger gain de performance avec cette configuration.

5.3.2 Taille du domaine

Les figures 5.4(b) et 5.4(a) montrent que les dimensions du domaine de recherche ont un impact plus ou moins important en fonction du GPU utilisé. Sur la figure 5.4(a), à l'exception de 192³, des pics de performances sont généralement atteints avec les domaines de dimensions multiples de 32 et surtout 64, puis immédiatement suivis d'une chute importante des performances. On observe ainsi une corrélation avec la taille des *warp* (32) et des blocs (64) qui explique probablement en partie cette tendance. Cette supposition est confirmée par la figure 5.4(b), qui utilise des blocs de 32 *threads*. On observe dans ce cas des pics identiques sur les domaines multiples de 32.

L'impact de la taille du domaine par rapport à celle des *warp* et surtout des blocs est certainement dû à une moindre utilisation des capacités du GPU lors du calcul des derniers blocs. En effet, si les dimensions du domaine de recherche ne sont pas multiples des critères mentionnés, certains *threads* n'ont plus de données à calculer. Leur puissance de calcul est par conséquent perdue jusqu'à l'itération suivante.

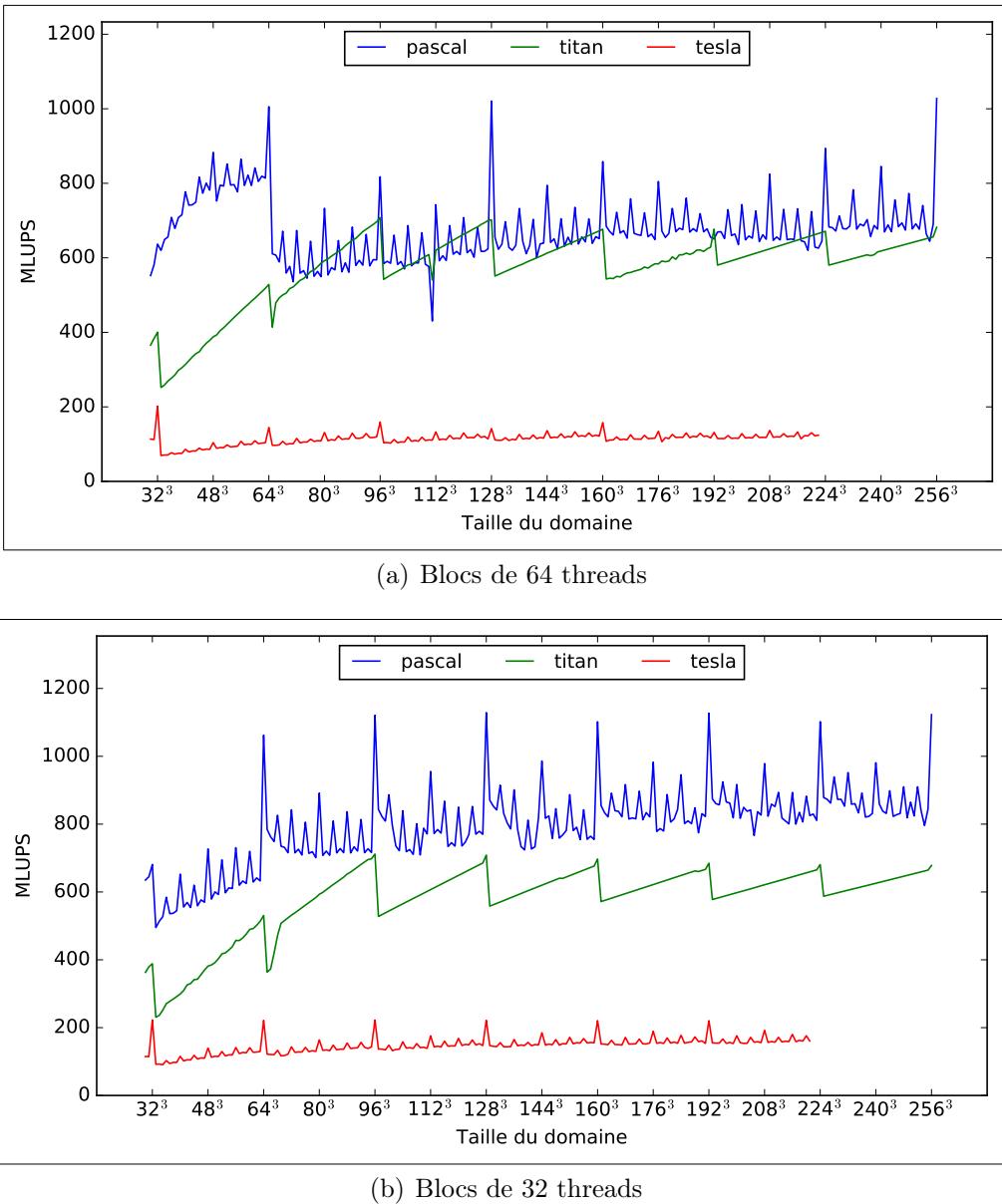


FIGURE 5.4 – Performances en fonction de la taille du domaine

Les mesures illustrées par figure 5.6 révèlent une tendance relative à la taille du domaine. Plus le domaine est grand, plus les performances sont stables. En effet, on observe des différences de performances allant jusqu'à plus de 90 MLUPS pour un domaine de 128^3 lattices, plus de 60 MLUPS pour un domaine de 64^3 lattices mais presque aucune après 50 itérations sur un domaine de 256^3 lattices.

5.3.3 Taille des blocs

La section précédente met déjà en lumière l'importance de la taille des blocs. Les mesures illustrées par la figure 5.5 cherchent à déterminer une taille optimale en fonction de dimension de domaine optimale (multiple de 32). Pour s'approcher du cas hybride, les performances sont désormais relevées sur des mesures qui simulent les transferts CPU/GPU de Palabos.

On observe des pics de performances pour les tailles de bloc multiples de 32 sur tous les types de GPU et, à moindre mesure, les multiples de 16. Toutefois, les meilleures

performances sont invariablement observées pour les blocs de 32 *threads*, suivis d’assez près par 64. Cette tendance s’explique probablement par la taille des *warp* qui est identique (32) et que le GPU ne peut pas exploiter ses capacités au mieux et atteindre la meilleure *occupancy* [34] si ces valeurs ne coïncident pas.

5.3.4 Nombre d’itérations

La figure 5.6 illustre l’évolution des performances en fonction du nombre d’itérations (ou générations) exécutées, sur trois tailles de domaine différentes (256^3 , 128^3 et 64^3). On observe une brève progression sur les toutes premières itérations, sur la plupart des mesures, certainement due aux initialisations et phénomènes de cache. Par la suite, les performances restent sans surprise relativement stables (sans tendance à la hausse ni à la baisse). Elles sont toutefois bien plus stables lorsque le domaine est grand.

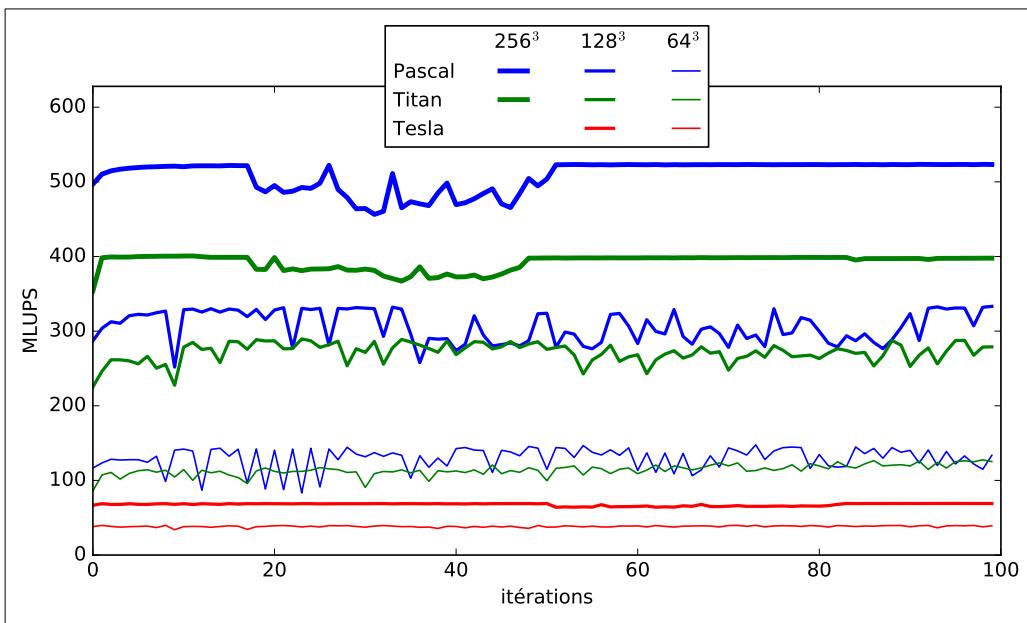


FIGURE 5.6 – Performances en fonction du nombre d’itérations

5.3.5 Profilage

Les *benchmarks* des sections précédentes se basent déjà sur des mesures qui simulent les transferts des populations entre le GPU et le CPU à chaque itération. Cette section s’intéresse à l’impact des trois principales étapes d’une exécution sur *GPU* soit les transferts *Send* (à destination du GPU) et *Receive* (à destination du CPU) ainsi que les calculs de l’itération *Collide & Stream*.

Les temps d’exécution sont extraits en deux étapes. Une première série de mesures, pour chaque dimension du domaine, est exécutée afin d’extraire un temps d’exécution de référence T_0 . Une seconde série est ensuite exécutée et extrait un temps T_1 pour l’étape dont on souhaite caractériser. Cette fois cependant, l’étape est artificiellement exécutée n fois (avec $n = 10$ dans le cas présent).

Les temps d’exécution T_0 et T_1 sont composé de T_{send} , $T_{receive}$ et $T_{collide\&stream}$ (les temps T_{step} possibles) ainsi que d’un temps restant T_ϵ non qualifié :

$$T_0 = T_{send} + T_{receive} + T_{col\&stream} + T_\epsilon \quad (5.5)$$

$$T_1 = T_0 + T_{step} \times (n - 1) \quad (5.6)$$

On peut ainsi calculer le temps d'exécution de l'étape T_{step} avec la formule suivante :

$$T_{step} = \frac{T_0 - T_1}{n - 1} \quad (5.7)$$

Cette méthode permet d'extraire les temps illustrés par les figures 5.7 et 5.8 où l'on observe que le *Collide & Stream* prend environ autant de temps que les transferts *Send* et *Receive* cumulés. La figure 5.8 illustre l'évolution des temps d'exécution en fonction de la taille du domaine sur une échelle linéaire.

5.4 *Benchmarks* de l'implémentation hybride

5.4.1 Profilage d'exécutions séquentielles

Maintenant que le profil des temps d'exécution est connu sur GPU, cette section s'intéresse à celui d'une exécution hybride à l'aide de `cavity_benchmark`. Les temps d'exécution sont calculés sur le même principe que ceux de la section 5.3.5 mais considèrent en plus les *Collide & Stream* effectués par les sous-domaines de Palabos ainsi que la duplication des chevauchements (détaillé en section 4.3.2). Les mesures sont effectuées sur le GPU Pascal, qui montrent jusqu'ici les meilleures performances.

Le domaine de 132^3 *lattices* est découpé en 27 sous-domaines (voir section 4.3.4). Le sous-domaine central est calculé par le co-processeur et les autres par le CPU. Deux séries de mesures, où sont comparé l'impact sur les performances des transferts complets ou partiels (détaillé en section 4.3.2), sont effectuées ; La première série utilise un co-processeur CPU (figures 5.10) à des fins de référence et la seconde un co-processeur GPU (figures 5.11).

Méthode de transfert (complet/partiel)

Cet aspect a l'effet le plus flagrant sur les performances. Avec un co-processeur CPU, le temps d'exécution général augmente avec la taille du domaine lors de transfert complet tandis qu'il reste stable avec des transferts partiels. Non seulement les temps de transfert complets sont nettement supérieurs, mais on observe aussi qu'une part inconnue du temps d'exécution de Palabos augmente ; probablement pour gérer ce flux de données nettement supérieur. On retrouve ce comportement avec un co-processeur GPU. Toutefois, on observe qu'avec la méthode de transfert partiel le temps de calcul diminue avec l'augmentation de la taille du sous-domaine central. La méthode de transfert complet est par conséquent à bannir, exception faite pour les très petits sous-domaines comme l'illustre la figure 5.12. Mais comme l'explique la sous-section suivante, ces sous-domaines de petite taille sont à éviter de toute manière.

Dimension des sous-domaines

Comme l'illustre la figure 4.15, la taille de l'enveloppe d'un transfert partiel n'augmente que très peu, proportionnellement à la taille du sous-domaine. Le temps gagné par le GPU à calculer une large partie du domaine n'est ainsi plus gâché par le temps nécessaire à

son transfert. Pour se convaincre du gain, il suffit de comparer le temps que met le co-processeur CPU (figure 5.10(b)) et le co-processeur GPU (figure 5.11(b)) pour calculer le plus grand sous-domaine : la moitié du temps total contre une fraction du temps à peine visible respectivement.

Pourtant, si le temps de calcul est proportionnel à la taille des sous-domaines confiés aux CPU de Palabos, celui-ci devrait être inférieur au temps observé par les mesures. En effet, si l'on calcule les proportions du domaine p_{pa} , et p_{gpu} attribuées au CPU et GPU, avec D la taille du domaine, d_{gpu} la taille du sous-domaine attribué au GPU et d_{pal} la taille cumulée des sous-domaines attribués à Palabos :

$$D = 132^3 = 2299968 \quad (5.8)$$

$$d_{gpu} = 128^3 = 2097152 \quad (5.9)$$

$$d_{pal} = D - d_{gpu} = 202816 \quad (5.10)$$

$$p_{gpu} = \frac{d_{gpu}}{D} \approx 0.912 \approx 91\% \quad (5.11)$$

$$p_{pal} = \frac{d_{pal}}{D} \approx 0.088 \approx 9\% \quad (5.12)$$

on trouve que seul 9% du domaine est attribué au CPU. Le temps de calcul devrait être près de 9% de celui mesuré pour $d_{gpu} = 8^3$. Or, il n'a diminué que d'un tiers environ.

Ce comportement s'explique probablement par les dimensions des sous-domaines attribués à Palabos. En effet, les sous-domaines avec $d_{gpu} = 128^3$ ressemblent à ceux illustrés sur la droite de la figure 4.18, mais nettement plus fins encore ! Dans cette configuration, l'accès mémoire à ces sous-domaines n'est probablement pas très optimal en raison de leurs dimensions.

Duplication des chevauchements

Les mesures montrent la part non négligeable qu'occupe la duplication des chevauchements. Toutefois, elle reste stable, qu'importe la configuration des sous-domaines.

Nombre d'itération

La section 5.3.4 montre que les performances de l'implémentation sur GPU ne sont pas influencées par le nombre d'itérations et celles de Palabos ne devraient pas l'être non plus. Toutefois, on ne peut pas écarter, a priori, la possibilité d'un mécanisme, caché dans l'architecture de Palabos, qui influencerait les performances au fil des générations.

Les mesures qu'illustre la figure 5.9 montrent cependant que les performances restent stables avec les itérations et écartent ainsi le doute sur la question.

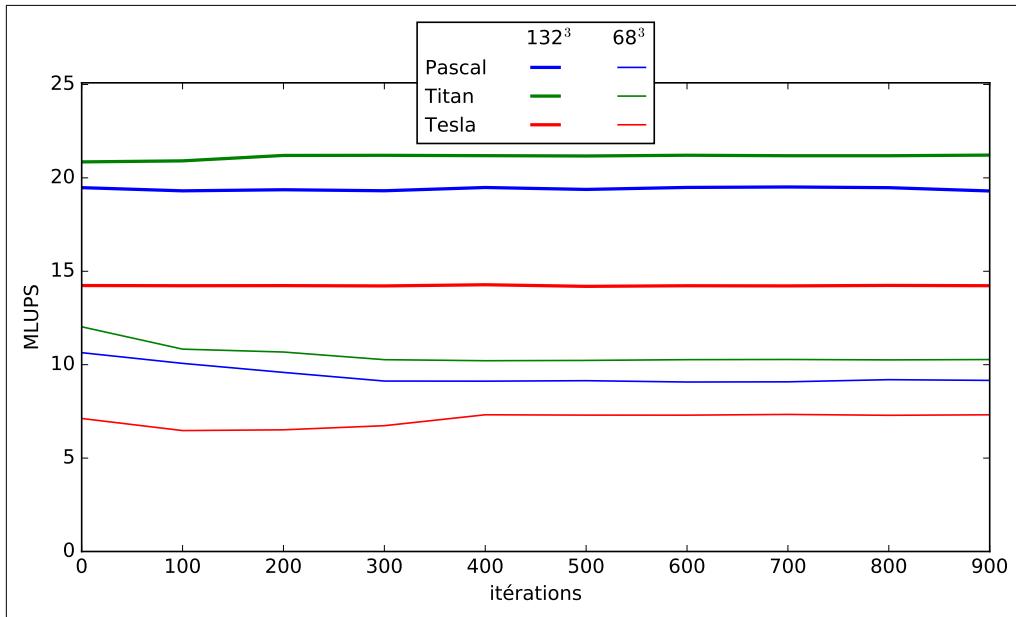


FIGURE 5.9 – Performances en fonction du nombre d’itérations sur Palabos

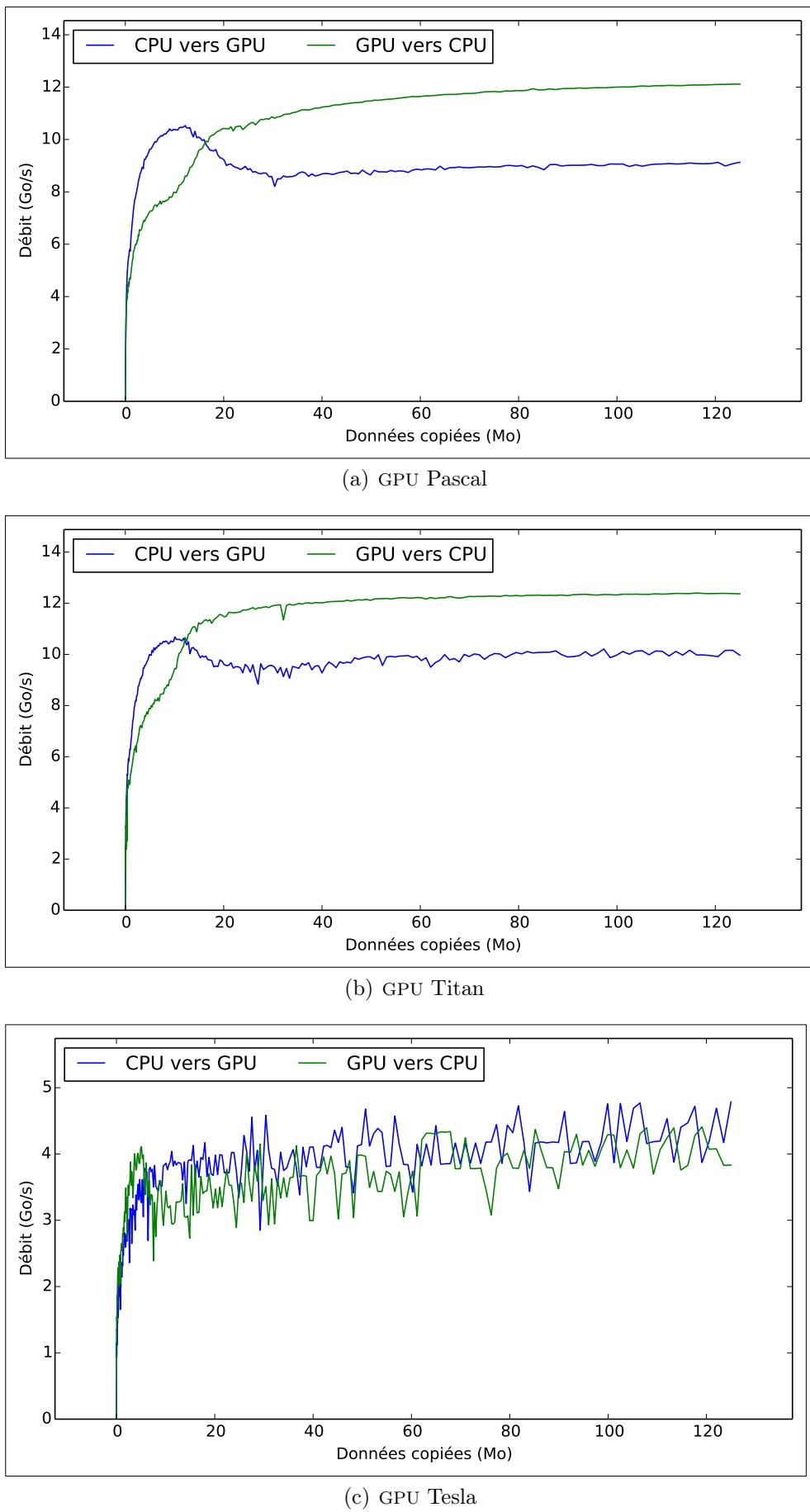
5.4.2 Exécutions parallèles

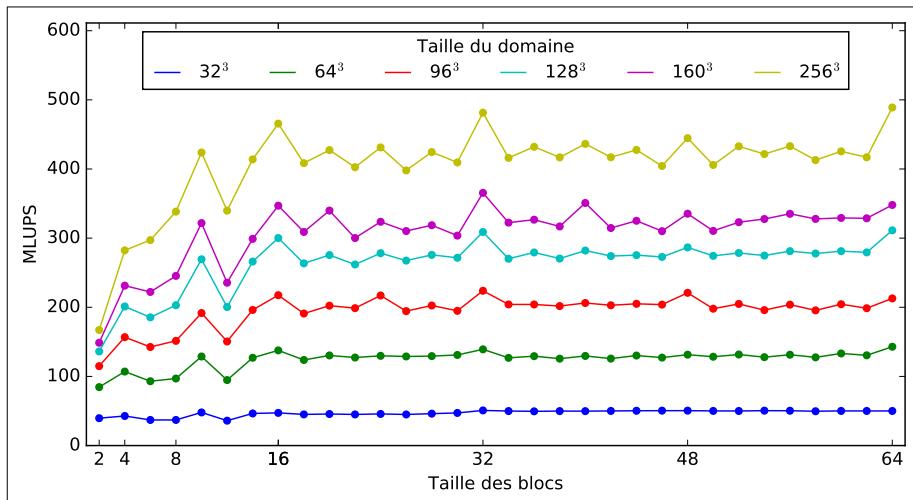
Pour obtenir des performances intéressantes, Palabos permet de paralléliser les calculs des différents sous-domaines. Cette section analyse les performances d’exécutions parallélisées sur 9 *threads*. Bien que l’idéale aurait été 27 *threads* (soit un par sous-domaine), le *cluster* de calcul ne permet pas d’en attribuer autant. Par conséquent, chaque *thread* calcule trois sous-domaine séquentiellement sur le CPU, à l’exception de l’un d’entre eux qui utilise le co-processeur sur le sous-domaine central. La figure 5.13 illustre les performances mesurées pour les exécutions hybrides avec les trois types de GPU disponibles.

Bien que le GPU le plus puissant soit Pascal, les meilleures performances parallèles sont obtenues sur Titan qui est doté de CPU plus rapides (voir la table 5.1).

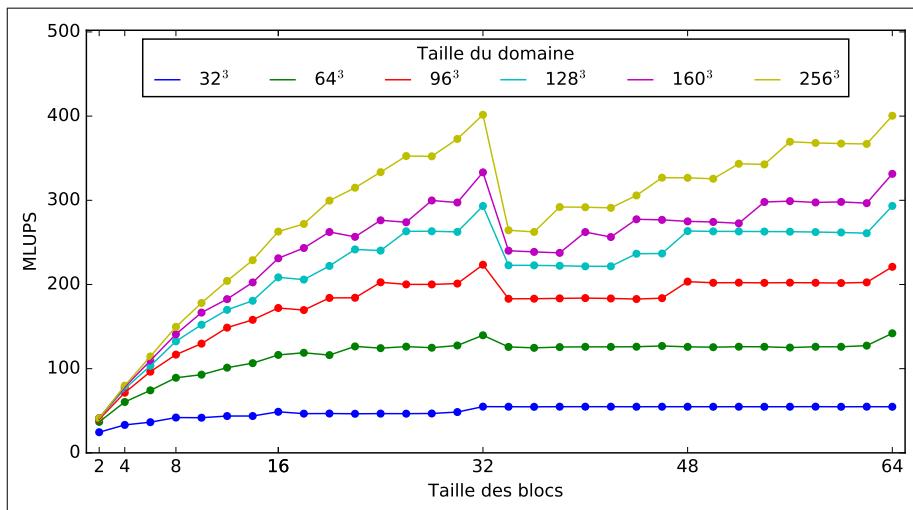
On observe sinon que les trois simulations suivent la même tendance. Les performances augmentent pour atteindre leur pic entre les dimensions 32^3 et 48^3 du sous-domaine du co-processeur puis diminuent ensuite avec une très légère augmentation à 128^3 .

Le pic de performance semble coïncider avec la dimension à laquelle tous les sous-domaines ont la même taille, soit $(132/3)^3 = 44^3$. Cette dimension assure une distribution équitable du travail entre tous les *threads* (à l’exception de celui du GPU qui serait plus rapide) et évite le problème évoqué en section 5.4.1 sur les dimensions non optimales pour les CPU. Cette tendance apparaît tant avec un co-processeur CPU que GPU, ce qui indique qu’elle est principalement due aux CPU. La seule différence (discrète) avec le GPU est la légère augmentation de performance que l’on devine sur 128^3 , qui parviendrait probablement à tirer les performances vers le haut à partir de ce point, s’il pouvait s’occuper d’un sous-domaine plus grand.

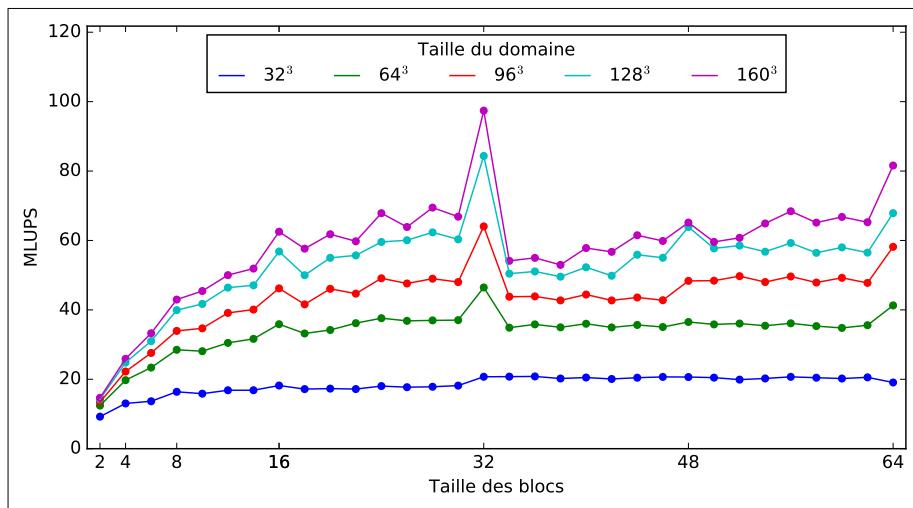
FIGURE 5.2 – Débit des transferts entre GPU et CPU (`cudaMemcpy`)



(a) GPU Pascal



(b) GPU Titan



(c) GPU Tesla

FIGURE 5.5 – Performances en fonction de la taille des blocs

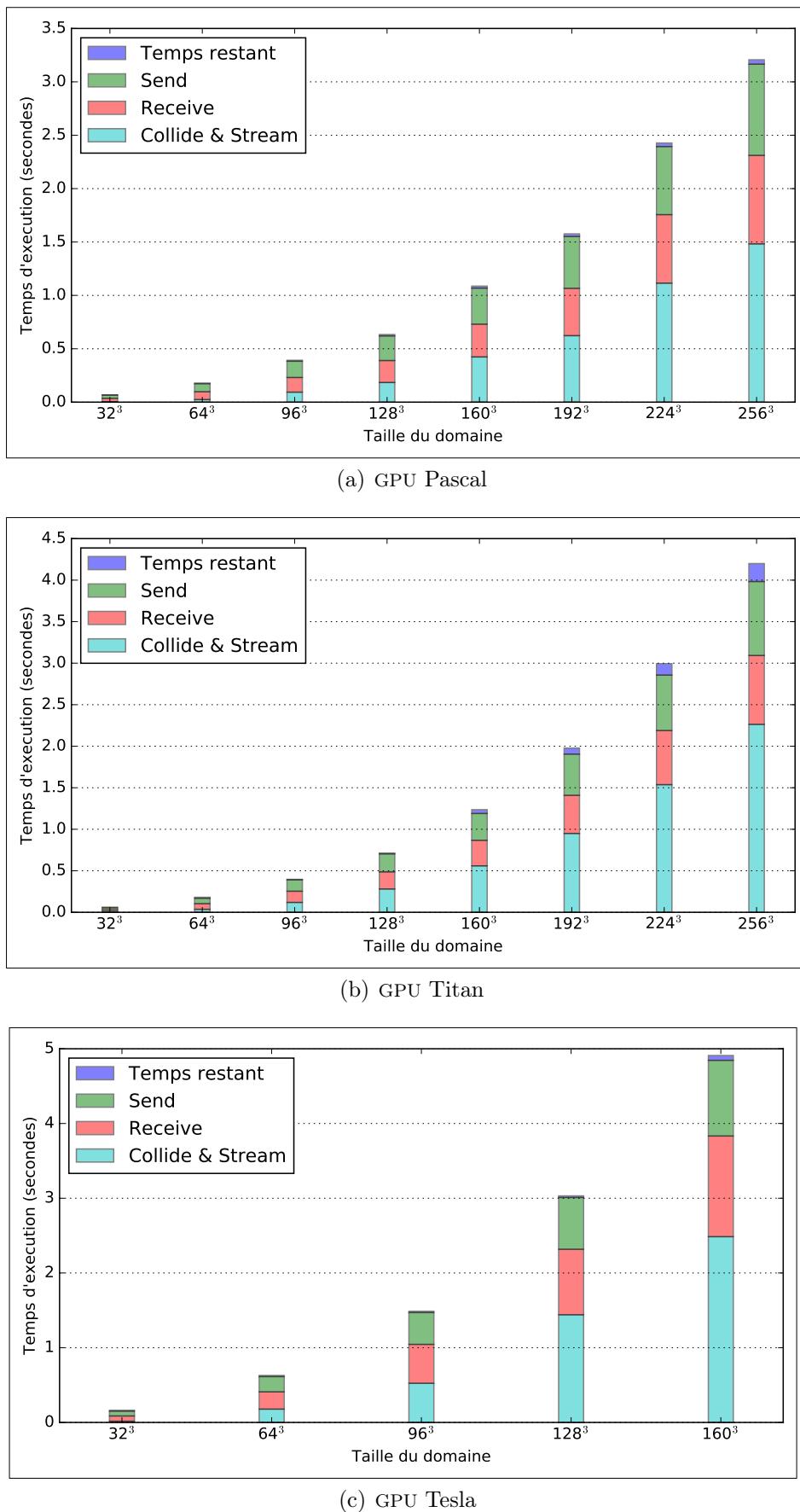


FIGURE 5.7 – Profil des temps d'exécution sur un GPU

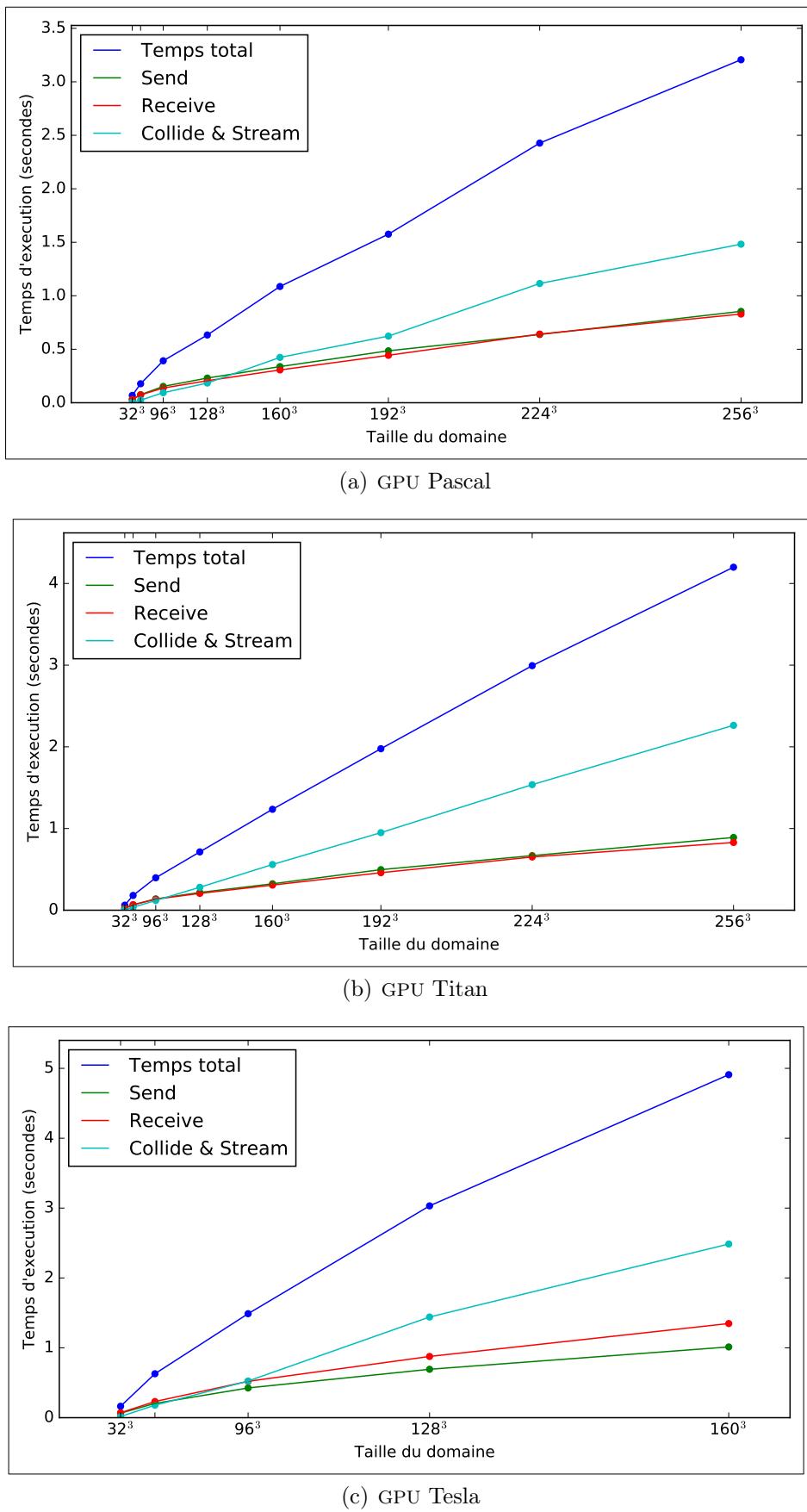


FIGURE 5.8 – Courbe des temps d'exécution sur un GPU

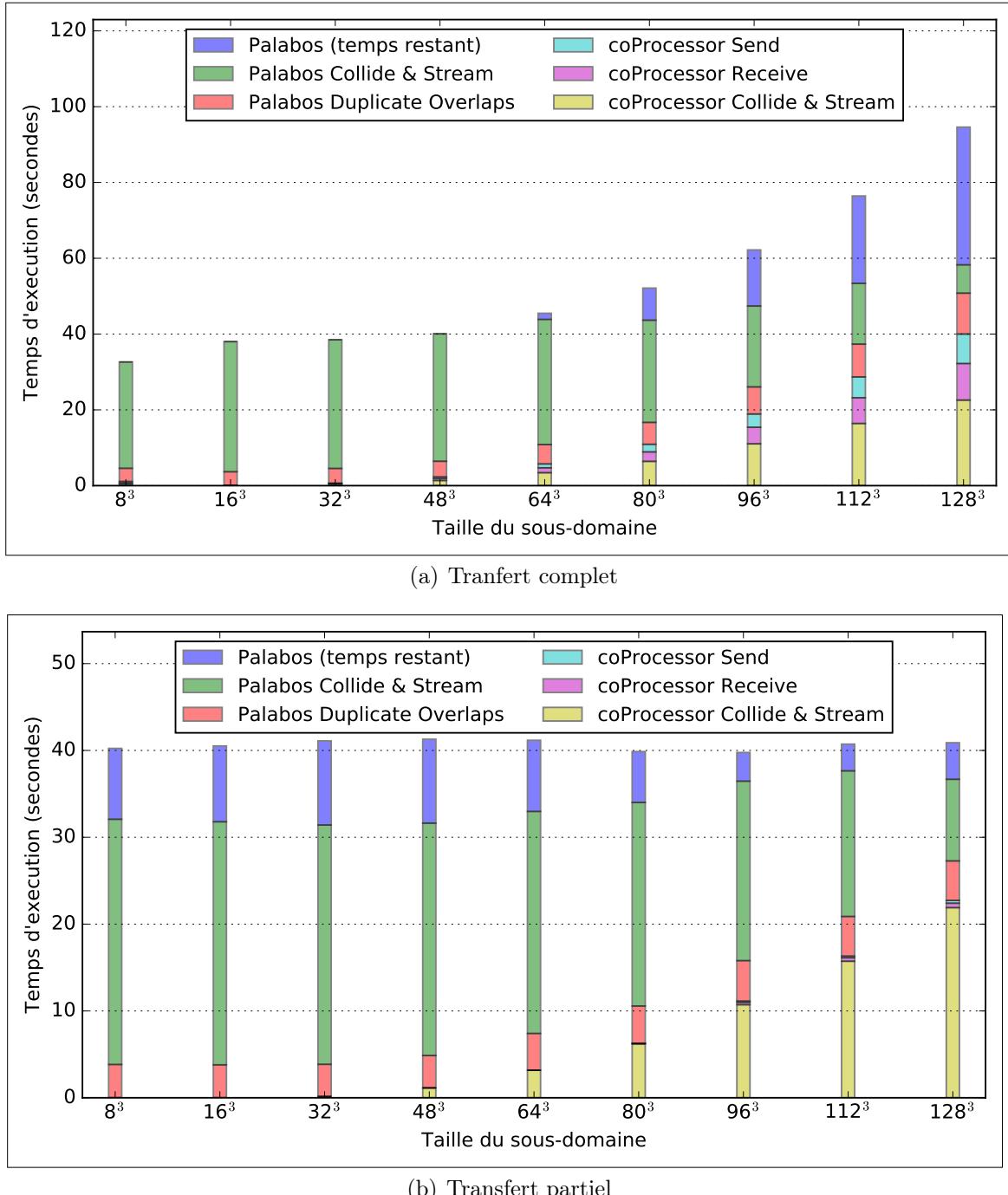


FIGURE 5.10 – Profil des temps d'exécution sur un co-processeur CPU

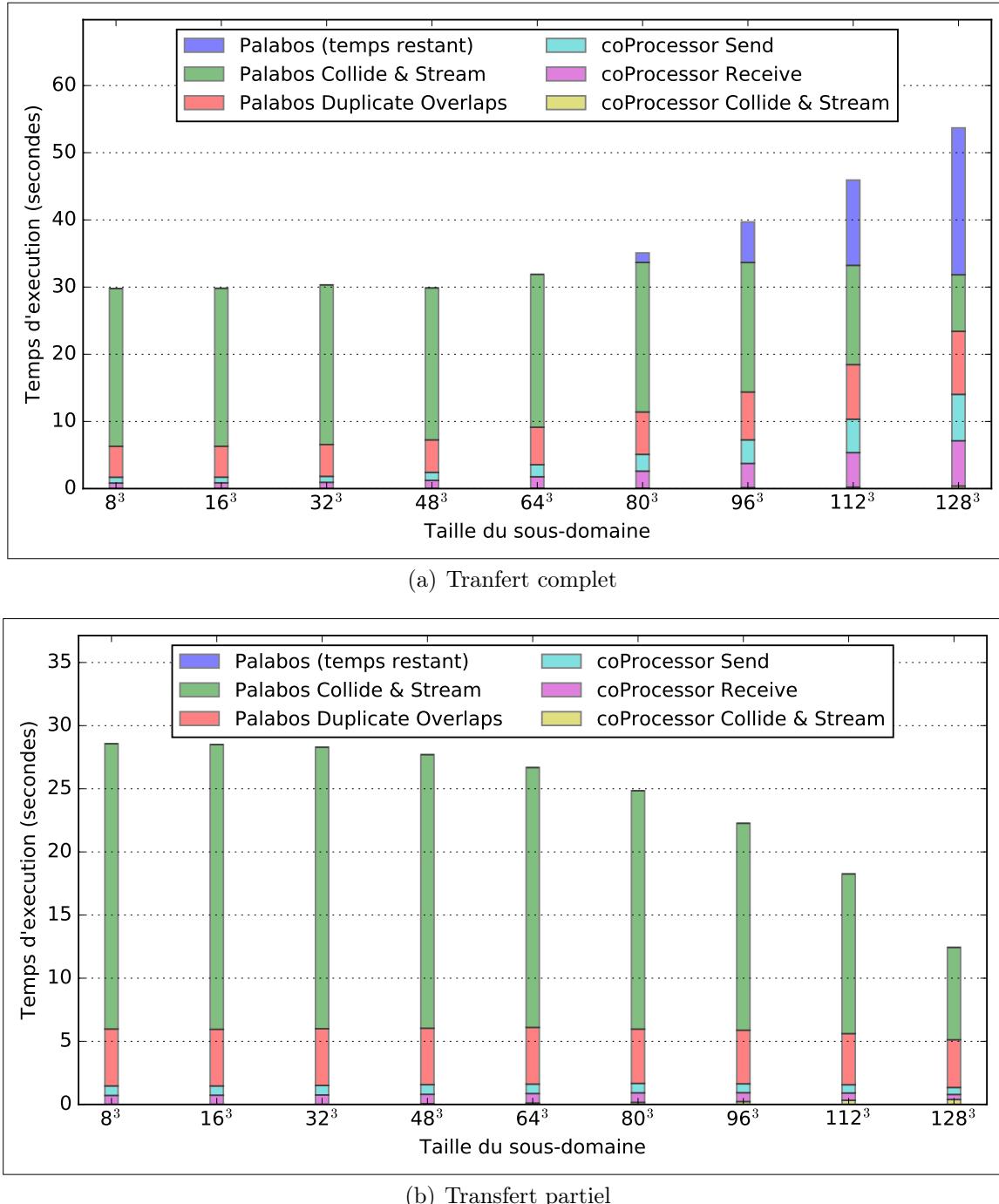


FIGURE 5.11 – Profil des temps d'exécution sur un co-processeur GPU

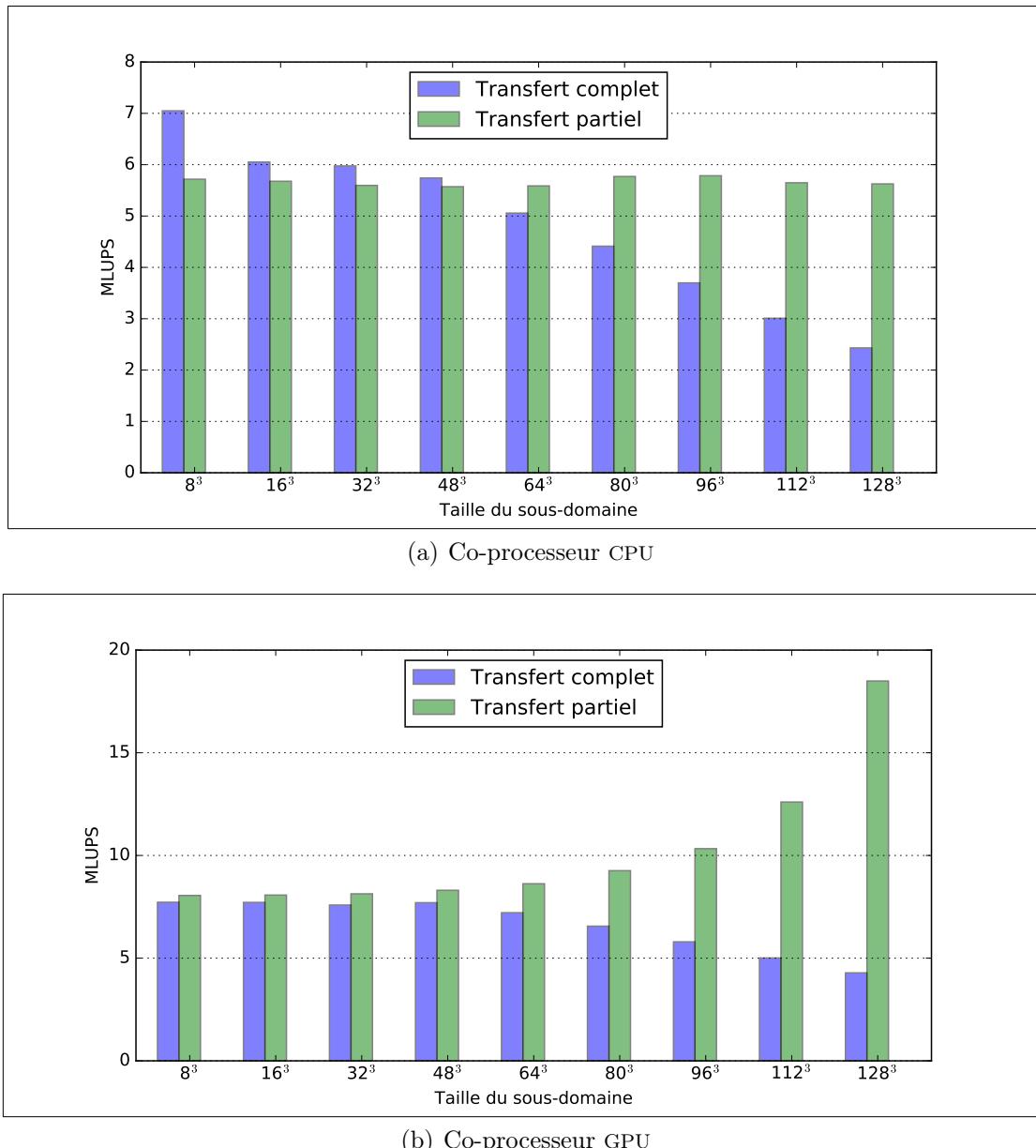


FIGURE 5.12 – Performance des méthodes de transfert entre Palabos et le co-processeur

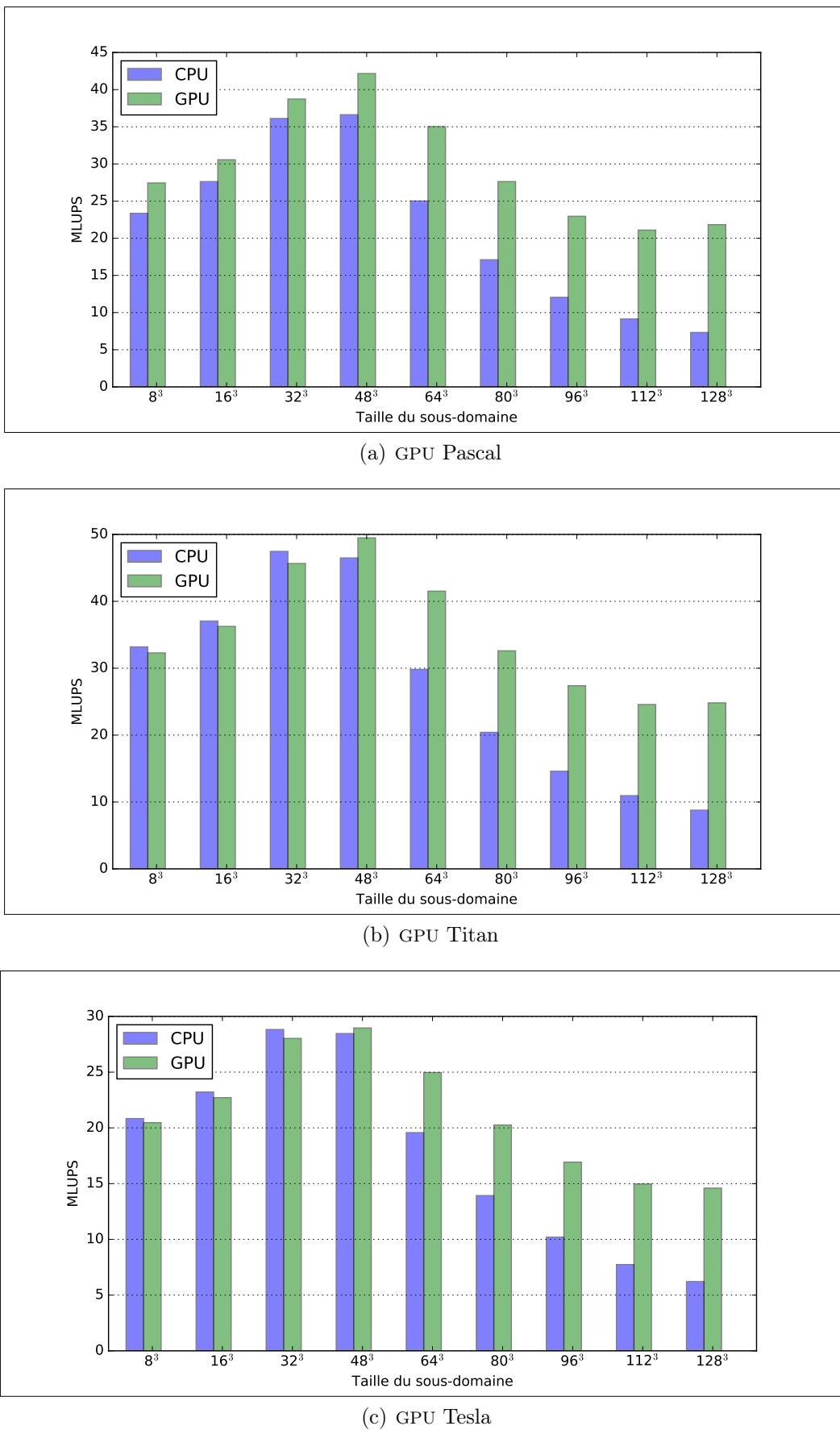


FIGURE 5.13 – Performance d'exécutions parallèles sur 9 threads

Chapitre 6

Modèle de performance

6.1 Modèle GPU

6.1.1 Identification des paramètres

Les performances de l'implémentation sur GPU sont affectées par les paramètres suivants :

- le GPU ;
- la taille du domaine ;
- la taille des blocs ;
- la méthode de transfert ;
- et l'ordre des indices.

À l'exception des deux premiers paramètres, les mesures ont montré des configurations optimales pour chacun d'entre eux ; à savoir l'utilisation d'une taille de blocs de 32 *threads*, de transferts partiels du domaine et d'adresser la mémoire des populations avec les indices organisés dans l'ordre XYZ.

Ces paramètres n'ont pas besoin d'être inclus dans le modèle de performance, puisqu'ils sont fixés à leur configuration optimale. Seuls le GPU utilisé et la taille du domaine simulé sont pris en compte. On considérera également des dimensions optimales pour la taille du domaine (multiples de 32).

6.1.2 Formulation du modèle de base

Les profilages réalisés en section 5.3.5 montrent que le temps d'exécution d'une simulation se résume essentiellement à la somme des périodes du *Send*, *Collide & Stream* et *Receive*. Les mesures suggèrent aussi les tendances, reconnaissables sur chaque type de GPU, que suivent leurs courbes en fonction de la taille du domaine.

Pour commencer, la courbe du temps d'exécution de *Collide & Stream* semble être linéaire. Cette supposition est confirmée par les droites de régression tracées à l'aide des mesures effectuées sur la figure 6.1. On peut ainsi exprimer son temps d'exécution T_{cs} , pour une génération, par la formule suivante, avec N_x , N_y et N_z les dimensions du domaine ou N_{total} , le nombre total de populations et un coefficient Ψ déterminé en fonction du GPU.

$$T_{cs} = \Psi \cdot N_x \cdot N_y \cdot N_z \quad (6.1a)$$

$$= \Psi \cdot N_{total} \quad (6.1b)$$

Ensuite, les courbes du temps d'exécution de *Send* et *Receive* semblent suivre l'évolution de la taille de l'enveloppe transférée (figure 4.14). La formule suivante exprime ainsi, pour une génération, le temps de transfert T_{send} de l'enveloppe extérieure, du CPU au GPU, réalisé par le *Send*, avec N_{out} la taille de l'enveloppe extérieure et un coefficient Γ déterminé en fonction du GPU.

$$T_{send} = \Gamma \cdot \left(N_x \cdot N_y \cdot N_z - (N_x - 2)(N_y - 2)(N_z - 2) \right) \quad (6.2a)$$

$$= \Gamma \cdot N_{out} \quad (6.2b)$$

De la même façon, la formule qui suit exprime le temps de transfert $T_{receive}$ de l'enveloppe intérieure, du GPU au CPU, réalisé par le *Receive*, avec N_{in} la taille de l'enveloppe intérieure et un coefficient F déterminé en fonction du GPU.

$$T_{receive} = F \cdot \left((N_x - 2)(N_y - 2)(N_z - 2) - (N_x - 4)(N_y - 4)(N_z - 4) \right) \quad (6.3a)$$

$$= F \cdot N_{in} \quad (6.3b)$$

Les coefficients $\Psi\Gamma F$ du modèle de performance, dans sa formulation complète, permettent ainsi d'exprimer le temps d'exécution sur GPU pour g générations (itérations) :

$$T_{GPU} = g(T_{cs} + T_{send} + T_{receive}) \quad (6.4a)$$

$$T_{GPU} = g(\Psi \cdot N_{total} + \Gamma \cdot N_{out} + F \cdot N_{in}) \quad (6.4b)$$

ou les performances en LUPS :

$$\text{LUPS} = \frac{N_{total} \cdot g}{T_{total}} \quad (6.5a)$$

$$\boxed{\text{LUPS} = \frac{N_{total}}{\Psi \cdot N_{total} + \Gamma \cdot N_{out} + F \cdot N_{in}}} \quad (6.5b)$$

6.1.3 Affinage du modèle

Description du modèle de performance pour automate cellulaire

Albuquerque et al. [1] proposent un modèle de performance pour évaluer le temps d'exécution d'un automate cellulaire sur un GPU qu'ils formulent ainsi :

$$T_{GPU} = g \cdot T_{kernel} = g \left(\frac{N_{total} \cdot C_T}{N_{SM} \cdot N_P \cdot D \cdot freq} + T_{launch} \right) \quad (6.6)$$

avec N_{SM} le nombre de SM, N_P le nombre de processeurs sur chaque SM, D la profondeur du *pipeline* des processeurs, $freq$ sa fréquence, C_T le travail dont un *thread* est capable et T_{launch} le temps de démarrage d'un *kernel*.

La formule est aussi étendue pour un *cluster* de GPU :

$$T_{cluster} = g \left(\frac{N_{total} \cdot C_T}{N_{GPU} \cdot N_{SM} \cdot N_P \cdot D \cdot freq} + T_{launch} + T_{trans} \right) \quad (6.7)$$

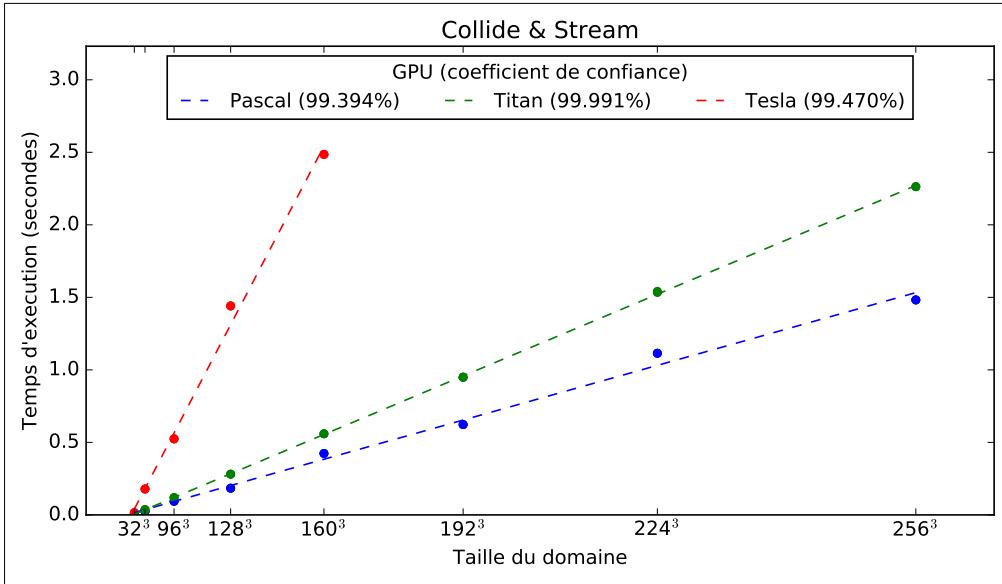


FIGURE 6.1 – Droite de régression linéaire sur les temps mesurés du *Collide & Stream*

avec N_{GPU} le nombre de GPU et le temps de transfert $T_{trans} = T_{lat} + N_{boundary}/bw$ où T_{lat} est la somme des latences, $N_{boundary}$ les données aux bords à transférer et bw la bande passante.

Parmi tous ces termes, N_{SM} , N_P , D et $freq$ sont des propriétés du GPU et C_T , T_{launch} , T_{trans} sont inférés à partir des simulations.

Adaptation du modèle

L’implémentation sur GPU de LBM, qui est un automate cellulaire, peut profiter de ce modèle pour calculer les coefficients $\Psi\Gamma F$ formulés en section 6.1.2. Elle fonctionne comme un *cluster*, en raison des transferts à chaque itération, mais avec $N_{GPU} = 1$. On adapte ainsi légèrement le modèle en conséquence :

$$T_{GPU} = g \left(\frac{N_{total} \cdot C_T}{N_{SM} \cdot N_P \cdot D \cdot freq} + T_{launch} + T_{trans} \right) \quad (6.8)$$

Le coefficient Ψ qui équivaut au temps requis pour calculer une population par génération correspond selon ce modèle à l’expression suivante :

$$\Psi = \frac{C_T}{N_{SM} \cdot N_P \cdot D \cdot freq} \quad (6.9)$$

et s’intègre ainsi à la formule générale :

$$T_{GPU} = g \left(\Psi \cdot N_{total} + T_{trans} + T_{launch} \right) \quad (6.10)$$

Les coefficients Γ et F correspondent au temps requis pour transférer et réorganiser une population, respectivement vers et depuis le GPU. Le modèle définit les temps de transfert avec T_{trans} .

$$T_{trans} = T_{lat} + \frac{N_{boundary}}{bw} \quad (6.11)$$

où $N_{boundary} = N_{out} + N_{in}$. Les mesures montrent que cette formulation n’est toutefois pas suffisamment précise, car la bande passante est différente d’un sens à l’autre. Il faut

par conséquent ajuster le modèle pour qu'il utilise un T_{trans} différent par type de transfert :

$$T_{GPU} = g \left(\Psi \cdot N_{total} + T_{s,trans} + T_{r,trans} + T_{launch} \right) \quad (6.12)$$

L'implémentation réalisée ne se contente toutefois pas de transférer des données lors du *send* et du *receive*, mais les réorganise sur le GPU, dont le *kernel* a un C'_T différent du premier. On compte par ailleurs deux T_{launch} supplémentaires, par *kernel* de réorganisation.

$$T_{GPU} = g \left(\Psi \cdot N_{total} + T_{s,trans} + T_{r,trans} + 2 \cdot \frac{N_{boundary} \cdot C'_T}{N_{SM} \cdot N_P \cdot D \cdot freq} + 3 \cdot T_{launch} \right) \quad (6.13)$$

Pour simplifier la lecture, on pose ψ , le temps nécessaire à ce *kernel* pour réorganiser une population.

$$T_{GPU} = g \left(\Psi \cdot N_{total} + T_{s,trans} + T_{r,trans} + 2 \cdot \psi \cdot N_{boundary} + 3 \cdot T_{launch} \right) \quad (6.14)$$

On développe les T_{trans} puis distingue N_{in} et N_{out} des $N_{boundary}$ utilisés jusqu'à présent.

$$T_{GPU} = g \left(\Psi \cdot N_{total} + \frac{N_{out}}{bw_{send}} + \frac{N_{in}}{bw_{receive}} + \psi \cdot (N_{out} + N_{in}) + 3 \cdot T_{launch} + 2 \cdot T_{lat} \right) \quad (6.15)$$

On peut ainsi considérer que les coefficients Γ et F correspondent aux expressions suivantes, que l'on ajoute au modèle.

$$\Gamma = \frac{1}{bw_{send}} + \psi \quad (6.16) \qquad F = \frac{1}{bw_{receive}} + \psi \quad (6.17)$$

$$T_{GPU} = g \left(\Psi \cdot N_{total} + \Gamma \cdot N_{out} + F \cdot N_{in} + 3 \cdot T_{launch} + 2 \cdot T_{lat} \right) \quad (6.18)$$

En dehors de la formulation détaillée des coefficients $\Psi\Gamma F$, cette formule est très proche du modèle de base proposé en 6.1.2, à cela qu'il considère T_{launch} et T_{lat} , qu'il faut ajouter à la définition des temps intermédiaires :

$$T_{cs} = \Psi \cdot N_{total} + T_{launch} \quad (6.19)$$

$$T_{send} = \Gamma \cdot N_{out} + T_{launch} + T_{lat} \quad (6.20)$$

$$T_{receive} = F \cdot N_{in} + T_{launch} + T_{lat} \quad (6.21)$$

6.1.4 Calcul et mesure des coefficients $\Psi\Gamma F$

Le modèle adapté d'Albuquerque et al. [1] en section 6.1.3 nous donne des formules pour calculer les coefficients, à commencer par Ψ .

$$\Psi = \frac{C_T}{N_{SM} \cdot N_P \cdot D \cdot freq}$$

Les valeurs de N_{SM} , $freq$ et N_P se trouvent dans la documentation du GPU (table 5.1) ou s'obtiennent avec la fonction `cudaGetDeviceProperties` de Cuda, qui renseigne respectivement les champs suivants de la structure `cudaDeviceProp` : `multiProcessorCount`, `clockRate` et finalement le couple `major` et `minor` pour déterminer la *compute capability* du GPU et déduire ses N_P (table 6.1) :

Compute Capability	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5	3.7	5.0	5.2	6.0	6.1	6.2
Nombre de SP		8		32	48		192			128		64		128

TABLE 6.1 – Nombre de Stream Processor (SP) par SM [38]

En l'absence d'informations concernant la profondeur D des *pipelines*, le calcul se base sur des valeurs de C_T/D , inférées par les mesures.

$$\left(\frac{C_T}{D}\right)_{Pascal} = 4773.888 \quad \left(\frac{C_T}{D}\right)_{Titan} = 7407.5904 \quad \left(\frac{C_T}{D}\right)_{Tesla} = 4126.72$$

On trouve ainsi les valeurs du coefficient Ψ pour chaque GPU :

$$\Psi_{Pascal} = 0.9 \cdot 10^9 \quad \Psi_{Titan} = 1.35 \cdot 10^9 \quad \Psi_{Tesla} = 6.2 \cdot 10^9$$

Les coefficients Γ et F , dont les valeurs dépendent de toutes les propriétés du modèle, sont plus compliqués à calculer et sont inférés par les mesures :

$$\begin{array}{lll} \Gamma_{Pascal} = 2.1 \cdot 10^{-6} & \Gamma_{Titan} = 2.1 \cdot 10^{-6} & \Gamma_{Tesla} = 9 \cdot 10^{-6} \\ F_{Pascal} = 2.25 \cdot 10^{-6} & F_{Titan} = 2.3 \cdot 10^{-6} & F_{Tesla} = 7 \cdot 10^{-6} \end{array}$$

6.1.5 Courbe théorique

Par rapport au modèle de base, le modèle affiné introduit les latences T_{lat} et T_{launch} . Les mesures de la section 5.2.2 montrent que ces valeurs sont de l'ordre de la [μs]. Elles ont par conséquent une incidence négligeable sur les mesures effectuées où $g = 100$.

La figure 6.2 illustre les courbes théoriques du modèle de base, tracées avec les coefficients $\Psi\Gamma F$ définis en section 6.1.4 et synthétisés dans la table 6.2. On constate un léger écart entre la courbe du temps total cumulé et les mesures sur Titan. Cet écart correspond au temps restant non qualifié, observé lors des mesures (figure 5.7), et que le modèle ne prend évidemment pas en compte.

À cette exception près, les courbes suivent fidèlement les temps mesurés. On peut affirmer par conséquent que le modèle de performance fonctionne correctement.

	Ψ	Γ	F
Pascal	$0.9 \cdot 10^{-9}$	$2.1 \cdot 10^{-6}$	$2.25 \cdot 10^{-6}$
Titan	$1.35 \cdot 10^{-9}$	$2.1 \cdot 10^{-6}$	$2.3 \cdot 10^{-6}$
Tesla	$6.2 \cdot 10^{-9}$	$9 \cdot 10^{-6}$	$7 \cdot 10^{-6}$

TABLE 6.2 – Coefficients $\Psi\Gamma F$ du modèle de performance

6.2 Modèle hybride

6.2.1 Identification des paramètres

Les contraintes mémoire, dues à l’implémentation GPU, à Palabos et aux machines de test, n’ont pas permis d’obtenir les performances que le système pourrait atteindre dans une configuration optimale. Cette section n’entre par conséquent pas trop en détail dans le modèle de performance hybride, mais esquisse ce à quoi il devrait ressembler.

L’utilisation de Palabos introduit les paramètres supplémentaires suivants :

- les dimensions N_{sx} , N_{sy} et N_{sz} des sous-domaines (et leur taille $N_s = N_{sx} \cdot N_{sy} \cdot N_{sz}$) ;
- le nombre N_{sd} de sous-domaines ;
- les paramètres de l’hôte (CPU, mémoire, etc.)

Les mesures montrent deux schémas de performances en fonction de la taille d’un sous-domaine. Tandis qu’elles augmentent continuellement sur GPU, les performances sur CPU restent stables entre 8.5 et 10 MLUPS mais baissent jusqu’à moins de 2 MLUPS lorsque N_s est petit et que certaines de ses dimensions sont petites (forme aplatie).

Ainsi, la stratégie de découpage du domaine doit trouver un compromis pour la division en sous-domaines pour :

- maximiser la taille des sous-domaines sur le co-processeur GPU ;
- en maximiser le nombre pour ceux qui doivent être exécutés sur CPU ;
- et minimiser le nombre de sous-domaines CPU aux dimensions trop petites.

Cette question est toutefois le sujet d’un travail différent. Par ailleurs, le modèle esquissé ici ne détaillera pas l’impact des paramètres physiques de la machine hôte (fréquence du CPU, bande passante, etc.).

6.2.2 Formulation du modèle

Palabos peut s’exécuter sur N_T threads. Dans une configuration de calculs séquentiels, avec $N_T = 1$, le temps d’exécution total $T_{Palabos}$ peut s’exprimer ainsi :

$$T_{Palabos} = g \left(T_{do} + \sum_i^{N_{sd}} T_{sd_i} \right) \quad (6.22)$$

avec T_{do} le temps d’exécution du *duplicate overlaps* et T_{sd_i} le temps calcul du sous-domaine i .

Le fonctionnement exact sous-jacent à T_{do} est caché dans l’implémentation de Palabos. Mais comme les mesures montrent que son temps d’exécution ne varie qu’en fonction de

la taille du domaine, on peut alors l'exprimer ainsi :

$$T_{do} = DO \cdot N_{Total} \quad (6.23)$$

avec DO un coefficient qui correspond au temps nécessaire au processus de duplication des chevauchements pour s'exécuter sur une population.

Le temps T_{sd_i} dépend du moyen employé pour calculer le sous-domaine. Dans le cas présent, $T_{sd_i} = T_{GPU_i}$ ¹ du modèle de performance du GPU, présenté en section 6.1.2 ou 6.1.3, si c'est le sous-domaine du co-processeur GPU, $T_{sd_i} = T_{CPU_i}$, le temps de calcul sur CPU si c'est un sous-domaine de Palabos.

Lors d'une simulation parallélisée, Palabos distribue à chaque *thread* t les sous-domaines $s(t) \in S$ à calculer. Son temps d'exécution T_{thread_t} , avec T_{launch} son temps de démarrage, s'exprime ainsi :

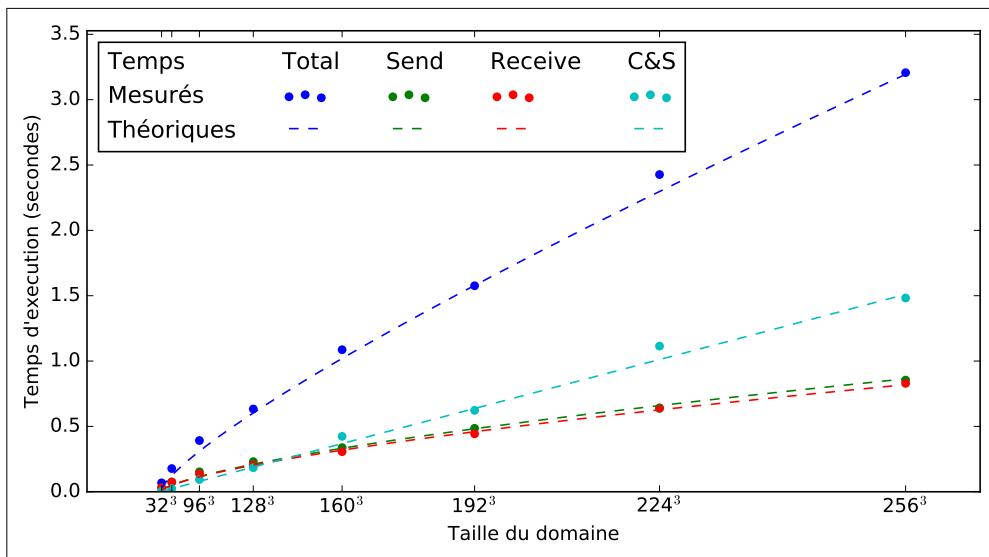
$$T_{thread_t} = T_{launch} + \sum_{s(t) \in S} T_{sd_{s(t)}} \quad (6.24)$$

Finalement, le temps d'exécution $T_{Palabos}$ s'exprime ainsi :

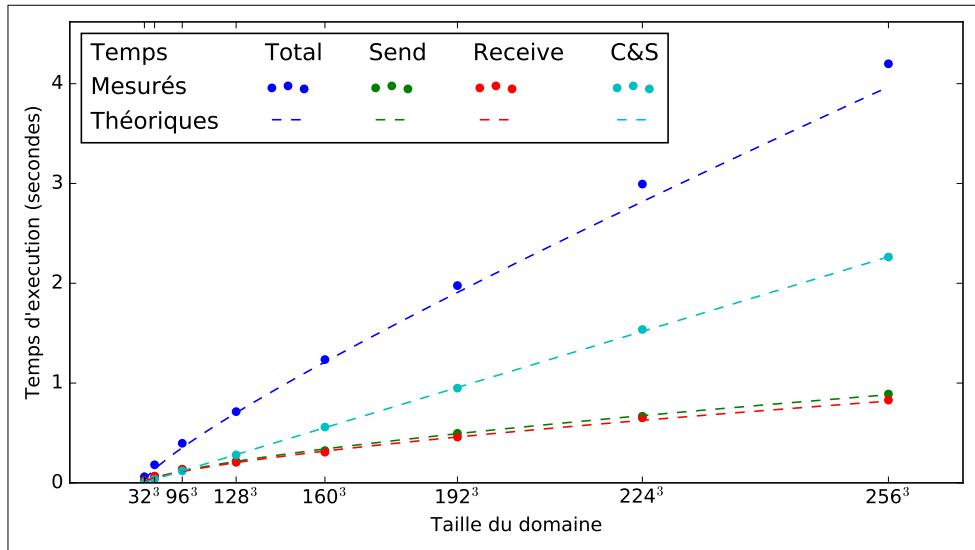
$$T_{Palabos} = g\left(T_{do} + \max(T_{thread})\right) \quad (6.25)$$

avec $\forall t, \max(T_{thread}) \geq T_{thread_t}$. En effet, les *threads* dont l'exécution est terminée doivent attendre les autres pour recevoir les bordures de leur voisin afin de calculer la génération suivante. Il est par conséquent primordial de diviser le domaine correctement pour réduire au maximum l'attente entre chaque *thread* pour ne pas gaspiller de temps de calcul potentiel.

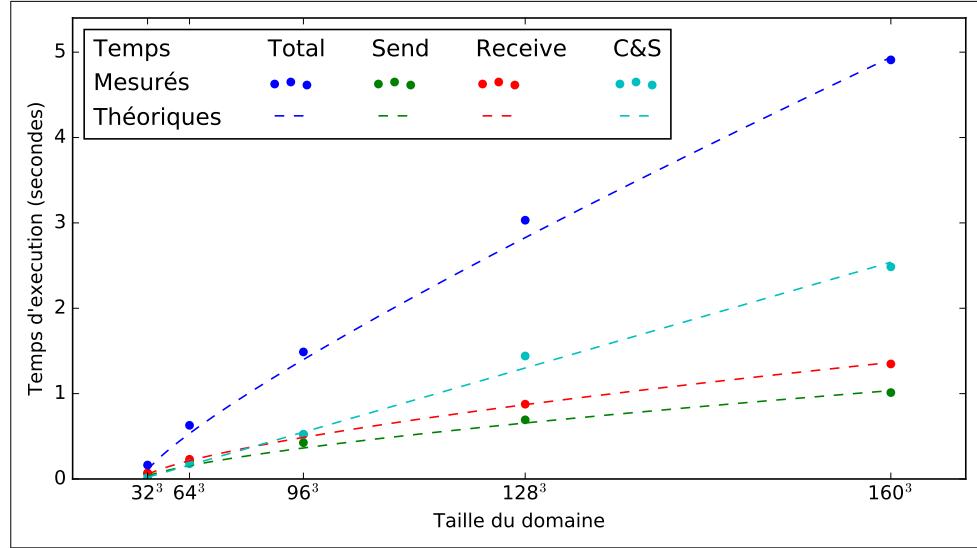
1. avec $g_{GPU_i} = 1$



(a) GPU Pascal



(b) GPU Titan



(c) GPU Tesla

FIGURE 6.2 – Courbes théoriques du modèle de performance

Chapitre 7

Bug et améliorations

7.1 Erreur de déallocation de mémoire globale

La bibliothèque `lbmcuda` implémente la fonction `lbt_simulation_destroy` qu'elle prévoit pour désallouer des ressources utilisées en mémoire globale du GPU à l'aide d'appel à `cudaFree`. Cette fonction est par conséquent appelée dans le destructeur de la classe `D3Q19CudaCoProcessor3D` de Palabos, pour libérer la mémoire lorsque le co-processeur GPU est détruit. Toutefois, lors de l'appel au destructeur, les `assert` placé autour des `cudaFree` dans `lbt_simulation_destroy` soulèvent que ceux-ci ont échoué avec l'erreur suivante :

```
driver shutting down
```

Ce problème est connu avec l'utilisation de `cudaFree` dans un destructeur [41, 42] et il est difficile de le corriger. Il ne pose toutefois pas vraiment problème, puisqu'il apparaît lorsque la simulation est terminée, et que l'exécution de Palabos se termine. Par conséquent, la mémoire est, normalement, libérée automatiquement lorsque le processus prend fin.

7.2 Entrelacement du *kernel* de transfert et de calcul

Une optimisation de l'implémentation GPU envisageable consiste à utiliser la méthode, proposée par Feichtinger et al. [10], qui consiste à entrelacer l'exécution d'un *kernel* de transfert et d'un *kernel* de calcul. Le premier à la charge du transfert ainsi que du calcul de la bordure du sous-domaine, tandis que le second calcule le reste du sous-domaine, qui n'a alors plus de dépendance avec la bordure.

Cette méthode permet ainsi de masquer au moins une partie du temps de transfert, l'étape la plus gourmande en temps d'exécution.

7.3 Fusion des tableaux de population

L'implémentation souffre encore du nombre important de registres qu'elle requiert, et par conséquent des accès à la mémoire local du GPU qui en résulte. Plusieurs méthodes pour en réduire leur nombre sont proposées par Tran et al. [31]. L'une d'elles pourrait encore être appliquée par la fusion des tableaux qui conservent les directions des populations.

```

1 typedef struct {
2     // Middle plane
3     double* ne;    // [ 1, 1, 0]  1./36   (1./36)
4     double* e;     // [ 1, 0, 0]  1./18   (1./9 )
5     double* se;    // [ 1,-1, 0]  1./36   (1./36)
6     double* n;     // [ 0, 1, 0]  1./18   (1./9 )
7     double* c;     // [ 0, 0, 0]  1./3    (4./9 )
8     double* s;     // [ 0,-1, 0]  1./18   (1./9 )
9     double* nw;    // [-1, 1, 0]  1./36   (1./36)
10    double* w;    // [-1, 0, 0]  1./18   (1./9 )
11    double* sw;    // [-1,-1, 0]  1./36   (1./36)
12    // Top plane
13    double* te;    // [ 1, 0, 1]  1./36
14    double* tn;    // [ 0, 1, 1]  1./36
15    double* tc;    // [ 0, 0, 1]  1./18
16    double* ts;    // [ 0,-1, 1]  1./36
17    double* tw;    // [-1, 0, 1]  1./36
18    // Bottom plane
19    double* be;    // [ 1, 0, -1] 1./36
20    double* bn;    // [ 0, 1, -1] 1./36
21    double* bc;    // [ 0, 0, -1] 1./18
22    double* bs;    // [ 0,-1, -1] 1./36
23    double* bw;    // [-1, 0, -1] 1./36
24 } lbm_lattices;

```

FIGURE 7.1 – Structure C pour l’arrangement SoA des populations sur le *kernel*

En effet, une structure (figure 7.1) accueille les tableaux de chaque direction, qui sont alloué chacun indépendamment.

Moyennant une adaptation du calcul de l’index d’une population, cette structure pourrait être remplacée par l’utilisation d’un seul tableau contenant toutes les directions, toujours avec un arrangement SoA. Cette méthode pourrait économiser quelques registres, puisque seul un pointeur serait conservé, à la place de dix-neuf.

Chapitre 8

Conclusion

Ce travail, dont le but est la réalisation d'une implémentation Cuda de LBM pour Palabos, a débuté avec de nombreux prototypes en C puis Cuda, avant d'aborder son intégration à Palabos.

L'objectif final de cette implémentation est évidemment d'en améliorer les performances. Les mesures effectuées montrent que l'objectif est atteint, mais soulignent aussi les limites d'une telle approche dans le cadre d'exécutions hybrides, à commencer par l'impact qu'ont les transferts de populations entre le GPU et le CPU à chaque itération. La division du domaine doit également être le sujet d'une attention particulière pour s'assurer d'obtenir les meilleures performances possibles et surtout éviter de les dégrader.

À ce titre, il s'avère que tous les problèmes ne se prêtent pas forcément bien à cet exercice. La simulation d'écoulement dans une cavité par exemple, telle qu'elle est implantée le découpe du domaine, n'est pas optimale pour profiter des capacités qu'offre l'implémentation GPU réalisée. En effet, les sous-domaines créés sur les bords sont trop petits et leur calcul sur CPU affiche de mauvaises performances. Une meilleure configuration serait composée d'un sous-domaine plus grand pour le GPU et de sous-domaines aux dimensions plus adéquates pour les CPU.

La taille du domaine est hélas également limitée par la mémoire disponible. En effet, l'implémentation réalisée se révèle gourmande, surtout en conjonction avec Palabos, ce qui limite sa capacité à offrir les meilleures performances qu'elle pourrait atteindre avec de plus grands domaines. Trouver un moyen de réduire son empreinte mémoire serait une piste à explorer dans le cadre d'un futur travail.

Enfin, les mesures obtenues dans les *benchmarks* valident le modèle de performance proposé par ce travail. En effet, il parvient à simuler avec précision les performances de l'implémentation GPU, à l'aide des coefficients $\Psi\Gamma F$. Le modèle proposé par Albuquerque et al. [1] a ensuite permis d'affiner le modèle et caractériser les paramètres de ces coefficients, pour en permettre le calcul. Un second modèle pour l'implémentation hybride est également esquisssé pour les exécutions séquentielles et parallèles. En l'affinant davantage et en l'utilisant en conjonction avec le modèle de l'implémentation GPU, il pourrait se révéler très utile pour estimer la meilleure stratégie de division du domaine, afin de maximiser les performances d'une simulation distribuée.

Bibliographie

- [1] Paul Albuquerque, Pierre Künzli, and Xavier Meyer. *Performance model for a cellular automata implementation on a GPU cluster*, PARCO, vol. 22. January 2012.
- [2] M. Astorino, J. Becerra Sagredo, and A. Quarteroni. A modular lattice boltzmann solver for GPU computing processors. *SeMA Journal*, vol. 59(1), pages 53–78, July 2012. ISSN 1575-9822, 2254-3902.
- [3] P. Bailey, J. Myre, S. D. C. Walsh, D. J. Lilja, and M. O. Saar. Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors. In *2009 International Conference on Parallel Processing*, pages 550–557, September 2009.
- [4] L. Biferale, F. Mantovani, M. Pivanti, M. Sbragaglia, A. Scagliarini, S. F. Schifano, F. Toschi, and R. Tripiccione. Lattice Boltzmann fluid-dynamics on the QPACE supercomputer. *Procedia Computer Science*, vol. 1(1), pages 1075–1082, May 2010. ISSN 1877-0509.
- [5] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse Matrix Solvers on the GPU : Conjugate Gradients and Multigrid. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 917–924, New York, NY, USA, 2003. ACM. ISBN 978-1-58113-709-5.
- [6] Federico Brogi. *The Lattice Boltzmann Method for the study of volcano aeroacoustic source processes*. PhD thesis, University of Geneva, 2017.
- [7] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs : Stream Computing on Graphics Hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.
- [8] J. O. Campos, R. S. Oliveira, R. W. dos Santos, and B. M. Rocha. Lattice Boltzmann method for parallel simulations of cardiac electrophysiology using GPUs. *Journal of Computational and Applied Mathematics*, vol. 295, pages 70–82, March 2016. ISSN 0377-0427.
- [9] Christian Feichtinger, Johannes Habich, Harald Köstler, Georg Hager, Ulrich Rüde, and Gerhard Wellein. A flexible Patch-based lattice Boltzmann parallelization approach for heterogeneous GPU–CPU clusters. *Parallel Computing*, vol. 37(9), pages 536–549, September 2011. ISSN 0167-8191.
- [10] Christian Feichtinger, Johannes Habich, Harald Köstler, Ulrich Rüde, and Takayuki Aoki. Performance modeling and analysis of heterogeneous lattice Boltzmann simulations on CPU–GPU clusters. *Parallel Computing*, vol. 46, pages 1–13, July 2015. ISSN 0167-8191.

- [11] J. Habich, C. Feichtinger, H. Koestler, G. Hager, and G. Wellein. Performance engineering for the lattice Boltzmann method on GPGPUs : Architectural requirements and performance results. *Computers & Fluids*, vol. 80, pages 276–282, July 2013. ISSN 0045-7930.
- [12] Miki Hirabayashi, Makoto Ohta, Daniel A. Rüfenacht, and Bastien Chopard. A lattice Boltzmann study of blood flow in stented aneurism. *Future Generation Computer Systems*, vol. 20(6), pages 925–934, August 2004. ISSN 0167-739X.
- [13] M. Januszewski and M. Kostur. Sailfish : A flexible multi-GPU implementation of the lattice Boltzmann method. *Computer Physics Communications*, vol. 185(9), pages 2350–2368, September 2014. ISSN 0010-4655.
- [14] Arie Kaufman, Zhe Fan, and Kaloian Petkov. Implementing the lattice Boltzmann model on commodity graphics hardware. *Journal of Statistical Mechanics-Theory and Experiment*, , pages P06016, June 2009. ISSN 1742-5468.
- [15] Jens Krüger and Rüdiger Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 908–916, New York, NY, USA, 2003. ACM. ISBN 978-1-58113-709-5.
- [16] Frederic Kuznik, Christian Obrecht, Gilles Rusaouen, and Jean-Jacques Roux. LBM based flow simulation using GPU computing processor. *Computers & Mathematics with Applications*, vol. 59(7), pages 2380–2392, April 2010. ISSN 0898-1221.
- [17] Jonas Latt. Technical report : How to implement your DdQq dynamics with only q variables per node (instead of 2q). 2007.
- [18] Phoi-Tack Lew, Luc Mongeau, and Anastasios Lyrintzis. Noise prediction of a subsonic turbulent round jet using the lattice-Boltzmann method. *The Journal of the Acoustical Society of America*, vol. 128(3), pages 1118–1127, September 2010. ISSN 0001-4966.
- [19] Dali Li, Chuanfu Xu, Yongxian Wang, Zhifang Song, Min Xiong, Xiang Gao, and Xiaogang Deng. Parallelizing and optimizing large-scale 3d multi-phase flow simulations on the Tianhe-2 supercomputer. *Concurrency and Computation-Practice & Experience*, vol. 28(5), pages 1678–1692, April 2016. ISSN 1532-0626.
- [20] W. Li, X. M. Wei, and A. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. *Visual Computer*, vol. 19(7-8), pages 444–456, December 2003. ISSN 0178-2789.
- [21] Keijo Mattila, Jari Hyväläluoma, Tuomo Rossi, Mats Aspnäs, and Jan Westerholm. An efficient swap algorithm for the lattice Boltzmann method. *Computer Physics Communications*, vol. 176(3), pages 200–210, February 2007. ISSN 0010-4655.
- [22] Mark J. Mawson and Alistair J. Revell. Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs. *Computer Physics Communications*, vol. 185(10), pages 2566–2574, October 2014. ISSN 0010-4655.
- [23] Tian Min, Gu Weidong, Pan Jingshan, and Guo Meng. Performance Analysis and Optimization of PalaBos on Petascale Sunway BlueLight MPP Supercomputer. *Procedia Engineering*, vol. 61, pages 241–245, January 2013. ISSN 1877-7058.

- [24] Christian Obrecht, Frederic Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. Global Memory Access Modelling for Efficient Implementation of the LBM on GPUs. *Lecture notes in computer science*, vol. 6449, pages 151–161, 2011.
- [25] Christian Obrecht, Pietro Asinari, Frederic Kuznik, and Jean-Jacques Roux. Thermal link-wise artificial compressibility method : GPU implementation and validation of a double-population model. *Computers & Mathematics with Applications*, vol. 72(2), pages 375–385, July 2016. ISSN 0898-1221.
- [26] Liu Peng, Ken-ichi Nomura, Takehiro Oyakawa, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. Parallel Lattice Boltzmann Flow Simulation on Emerging Multi-core Platforms. In Emilio Luque, Tomàs Margalef, and Domingo Benítez, editors, *Euro-Par 2008 – Parallel Processing : 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008. Proceedings*, pages 763–777. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-85451-7.
- [27] P. R. Rinaldi, E. A. Dari, M. J. Venere, and A. Clausse. A Lattice-Boltzmann solver for 3d fluid simulation on GPU. *Simulation Modelling Practice and Theory*, vol. 25, pages 163–171, June 2012. ISSN 1569-190X.
- [28] S. Schmieschek, L. Shamardin, S. Frijters, T. Kruger, U. D. Schiller, J. Harting, and P. V. Coveney. LB3d : A parallel implementation of the Lattice-Boltzmann method for simulation of interacting amphiphilic fluids. *Computer Physics Communications*, vol. 217, pages 149–161, August 2017. ISSN 0010-4655.
- [29] Markus Stürmer, Jan Götz, Gregor Richter, Arnd Dörfler, and Ulrich Rüde. Fluid flow simulation on the Cell Broadband Engine using the lattice Boltzmann method. *Computers & Mathematics with Applications*, vol. 58(5), pages 1062–1070, September 2009. ISSN 0898-1221.
- [30] J. Toelke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3d CFD. *International Journal of Computational Fluid Dynamics*, vol. 22(7), pages 443–456, 2008. ISSN 1061-8562.
- [31] Nhat-Phuong Tran, Myungho Lee, and Sugwon Hong. Performance Optimization of 3d Lattice Boltzmann Flow Solver on a GPU. *Scientific Programming*, vol. 2017, 2017. ISSN 1058-9244.
- [32] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann simulation optimization on leading multicore platforms. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–14, April 2008.
- [33] Yu Ye, Kenli Li, Yan Wang, and Tan Deng. Parallel computation of Entropic Lattice Boltzmann method on hybrid CPU-GPU accelerated system. *Computers & Fluids*, vol. 110, pages 114–121, March 2015. ISSN 0045-7930.
- [34] Achieved Occupancy, January 2018. URL <https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm>.
- [35] Cuda Toolkit Documentation, Cuda and Floating Point, November 2017. URL <http://docs.nvidia.com/cuda/floating-point/#cuda-and-floating-point>.

- [36] Cuda Toolkit Documentation, Controlling Fused Multiply-add, November 2017. URL <http://docs.nvidia.com/cuda/floating-point/#controlling-fused-multiply-add>.
- [37] Cuda Toolkit Documentation, Double Precision Intrinsics, November 2017. URL http://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__DOUBLE.html.
- [38] CUDA, January 2018. URL <https://en.wikipedia.org/w/index.php?title=CUDA&oldid=820132007>. Page Version ID : 820132007.
- [39] How to Implement Performance Metrics in CUDA C/C++, November 2012. URL <https://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/>.
- [40] List of Nvidia graphics processing units, January 2018. URL https://en.wikipedia.org/w/index.php?title=List_of_Nvidia_graphics_processing_units. Page Version ID : 819127188.
- [41] Stack Overflow : Trouble launching CUDA kernels from static initialization code, January 2018. URL <https://stackoverflow.com/a/24883665>.
- [42] Stack Overflow : cuda call fails in destructor, January 2018. URL <https://stackoverflow.com/a/35828076>.