Calcul à hautes performances (HPC)

Ce chapitre est le chapitre « technologique » du mémoire, dans lequel nous allons introduire et décrire les concepts et les mécanismes liés aux architectures que nous allons utiliser pour accélérer le calcul d'algèbre linéaire. Dans un premier temps, nous allons répondre à la question : quelles sont les architectures pensées pour le calcul parallèle et qui répondent aux exigences des calculs que nous considérons ? Par la suite, nous allons décrire le modèle de programmation CUDA, qui permet d'utiliser les processeurs graphiques (GPU) NVIDIA pour accélérer une application. La dernière partie du chapitre sera consacrée à l'exécution du calcul sur un cluster. Plus spécifiquement, nous allons discuter les aspects liés aux communications, lorsque les nœuds de calcul sont des CPUs ou des GPU.

Sommaire

2.1 Calcul à hautes performances et algèbre linéaire

2.1.1 Architectures pensées pour le calcul parallèle

Avant l'apparition des premières plate-formes de calcul parallèle, un programme résolvant un problème correspondait à un calcul *séquentiel*. Dans ce cadre, le programme est composé d'une série d'instructions, exécutées sur une seule ressource, typiquement un processeur d'une machine mono-cœur. Une seule instruction est exécutée à la fois sur la ressource. Plus tard, les architectures ont permis la résolution *parallèle* du calcul, c'est-à-dire l'utilisation simultanée de plusieurs ressources. Le problème est alors divisé en différentes parties qui peuvent être résolues en parallèle. Chaque partie est transformée en une suite d'instructions. Chaque instruction est exécutée sur une ressource.

Aujourd'hui, nous trouvons un large ensemble d'architectures matérielles adaptées à la résolution parallèle, parmi lesquelles :

— Les processeurs multi-cœurs : ce sont les processeurs dont la puce contient au moins deux unités d'exécution (cœurs). Les configurations typiques sont 4, 6 ou 8 cœurs par puce.
— Les processeurs *many-cœurs* : ce sont des architectures multi-cœurs où le nombre de cœurs est de l'ordre de dizaines à quelques centaines, par exemple la famille des coprocesseurs Intel Xeon Phi.
— Les processeurs vectoriels : ce sont des processeurs pourvus de fonctionnalités de parallélisme de données ; c'est-à-dire qu'ils peuvent exécuter en parallèle une même instruction sur des données distinctes.
— Les processeurs graphiques (GPU) : les processeurs graphiques sont intégrés dans les cartes graphiques pour accélérer les traitements graphiques qui sont généralement hautement parallèles (*embarrassingly parallel*).
— Les circuits logiques configurables : ce sont des architectures reconfigurables, à l'image des FPGA, où le programmeur configure le circuit à l'aide de blocs logiques et de ressources de routage. Ces architectures permettent d'exploiter le parallélisme matériel.
— Les clusters de calcul : un cluster de calcul est un ensemble de machines interconnectées par un réseau de communication, généralement à haut débit.

Les architectures que nous avons décrites dans la liste précédente, qui n'est pas exhaustive, se distinguent de par le niveau du parallélisme qu'elles exploitent :

— instruction ;
— donnée ;
— mémoire ;
— tâche.

Les architectures parallèles sont généralement classées selon l'organisation des données, des processus et des instructions. Flynn a introduit dans sa taxonomie [Fly72] 4 modes d'exécution possibles :

— SISD (Single Instruction Single Data) : Ce mode correspond au fonctionnement traditionnel d'une machine mono-processeur qui n'exploite aucun parallélisme.
— SIMD (Single Instruction Multiple Data) : Une même instruction est effectuée en parallèle sur des données différentes. On trouve ce mode dans les processeurs vectoriels.
— MISD (Multiple Instructions Single Data) : Plusieurs instructions agissent en parallèle sur une même donnée. Il n'existe pas beaucoup d'architectures qui implémentent cette catégorie.

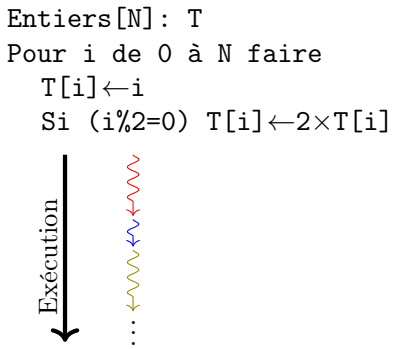
— MIMD (Multiple Instructions Multiple Data) : Ce mode est utilisé dans les machines multi-processeurs, où plusieurs processeurs exécutent différentes instructions sur différentes données.

Plus tard, cette classification a été complétée par les deux sous-catégories suivantes du mode MIMD :

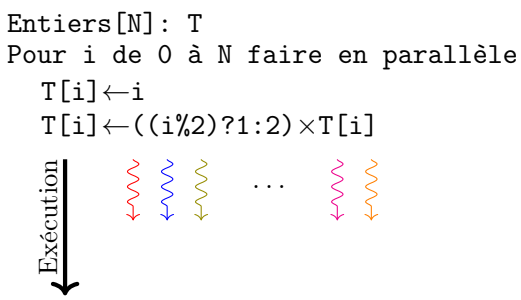
— SPMD (Single Program Multiple Data) : Un même programme (code) est exécuté par les processeurs, mais les instructions effectuées par chaque processeur peuvent varier. On trouve ce mode dans les processeurs graphiques.

— MPMD (Multiple Programs Multiple Data) : On rencontre ce mode par exemple sur un processeur bi-cœur, où on exécute un programme différent sur chaque cœur.

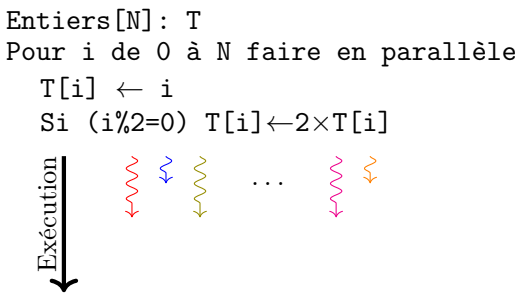
Dans la figure 2.1, nous illustrons le fonctionnement des modes SISD, SIMD et SPMD sur des programmes (écrits en pseudo-code), où on traite un tableau d'entiers. Dans la sous-figure 2.1a, il y a un traitement purement séquentiel (SISD). Dans la sous-figure 2.1b, les mêmes instructions sont effectuées en parallèle sur les éléments du tableau (SIMD). Dans la sous-figure 2.1c, le même programme n'exécute pas les mêmes instructions, parce que nous avons ajouté un test (SPMD).



(a) SISD



(b) SIMD



(c) SPMD

FIGURE 2.1 – Modes d'exécution : SISD, SIMD et SPMD.

2.1.2 Architectures appropriés à notre type d'algèbre linéaire

Dans cette section, nous allons réfléchir sur les caractéristiques que doit satisfaire une architecture appropriée à notre contexte applicatif d'algèbre linéaire. Nous avons mentionné à la sous-section 1.4.5 que notre problème est la résolution d'un système linéaire grand et creux sur un grand corps fini (voir section 3.2 pour plus de détails sur les caractéristiques des entrées). La

brique de base du calcul d'algèbre linéaire, si l'on utilise des algorithmes *boîte noire* comme ce sera le cas de ceux présentés au chapitre 3, est l'opération de multiplication de la matrice creuse par un vecteur ou par un bloc de vecteurs. Nous avons besoin d'une architecture qui exploite différents niveaux du parallélisme (voir section 3.3). L'architecture doit satisfaire les critères suivants :

1. La taille importante de la matrice et des vecteurs fait que nous avons besoin de pouvoir stocker et traiter d'une manière efficace des données dont les tailles peuvent atteindre quelques dizaines de gigaoctets.
2. La matrice est creuse, avec des zones très creuses. Ceci fait que les accès sur le ou les vecteurs d'entrée risquent d'être irréguliers. C'est pourquoi nous avons besoin de mémoires et de mécanismes de cache qui permettent de limiter les pénalités dues à l'irrégularité des accès.
3. Les opérations arithmétiques sont effectuées sur un grand corps fini. Il est important que l'architecture logicielle permette d'écrire des briques de calcul arithmétique spécifiques et efficaces. De plus, cette arithmétique peut être accélérée en utilisant une représentation qui exploite la vectorisation (voir chapitre 6).
4. Si nous envisageons de distribuer l'algèbre linéaire sur un cluster de machines, les calculs ne peuvent pas être complètement indépendants et sont donc amenés à partager des données de taille importante. C'est pourquoi nous avons besoin d'un réseau et de mécanismes de communication efficaces en terme de latence et de débit pour minimiser les coûts de communication. Nous pourrons aussi mettre en place un parallélisme de tâches, et dans ce cas, nous aurons besoin d'un modèle approprié.

À partir de cette description, nous concluons que les circuits configurables ne sont pas appropriés à cause de leur mémoire limitée (le critère 1 est non satisfait). Par contre, les CPU munis de plusieurs cœurs et d'unités vectorielles, ainsi que des accélérateurs de type GPU ou processeur *many-cœurs* se positionnent comme des architectures appropriées (les critères 1, 2 et 3 sont satisfaits ; le critère 1 peut être contraignant pour les GPU à partir de très grandes tailles).

Dans le cadre de ce travail, nous allons étudier l'utilisation des CPU multi-cœurs et vectoriels et des GPU pour le problème d'algèbre linéaire.

2.2 CPU multi-cœurs et vectoriels

Un CPU multi-cœurs est un processeur qui contient deux cœurs ou plus gravés sur la même puce. Le premier processeur multi-cœur est apparu au milieu des années 80 avec le CPU bi-cœur de Rockwell International. Dès le début des années 2000, différents constructeurs tels que IBM, AMD, Intel ont commencé à commercialiser des processeurs multi-cœurs. L'idée d'introduire plusieurs unités de calcul dans un même processeur permet d'implémenter physiquement le *multi-processing* et répond à la problématique du besoin continu d'augmenter la fréquence du CPU.

Un processeur vectoriel est un processeur pourvu d'instructions vectorielles, c'est-à-dire d'instructions appropriées au traitement parallèle de données d'un bloc de taillé fixée, qu'on appelle vecteur. Par le passé, les architectures vectorielles ont été développées pour les supercalculateurs, par exemple pour les machines Cray et ILLIAC. De nos jours, la vectorisation est exploitée dans les processeurs incorporant des instructions SIMD, par exemple dans tous les processeurs Intel à partir du Pentium III, dans les processeurs POWER de IBM et également dans le processeur CELL de la PlayStation 3.

Dans le cadre de ce travail, nous allons considérer les microprocesseurs de type x86 et les jeux d'instructions SIMD suivants :

— SSE (Streaming SIMD Extensions) qui fournit des registres 128 bits ;
— AVX (Advanced Vector Extensions) qui fournit des registres 256 bits.

Un registre SSE peut contenir 2 composantes 64 bits et exécuter deux instructions en parallèle, il peut aussi être configuré de sorte à contenir 4 composantes, sur 32 bits chacune. Un registre AVX peut contenir 4 composantes 64 bits ou 8 composantes 32 bits.

2.3 Programmation sur les GPU : Modèle CUDA

Dans ce travail, nous allons nous restreindre aux plate-formes NVIDIA et nous allons utiliser le modèle de programmation CUDA.

Le nom CUDA (Compute Unified Device Architecture) désigne l'architecture matérielle et logicielle spécifique aux GPU NVIDIA et qui permet d'exploiter leur capacité de traitement massivement parallèle pour des tâches plus génériques que les traitements graphiques et qui vont s'étendre à des applications de calcul scientifique, de simulation, etc. Cette orientation vers l'utilisation du processeur graphique pour des applications pas nécessairement graphiques est communément désignée par l'acronyme GPGPU (General-purpose Processing on Graphics Processing Units), qu'on traduirait par le calcul généraliste sur processeurs graphiques.

CUDA met à disposition de programmeurs grand public une plate-forme de calcul parallèle qui supporte conjointement les langages C, C++ et Fortran. Il existe aussi des interfaces pour d'autres langages telles que `PyCUDA` pour Python, `jCUDA` pour Java, etc. La description qui suit, ainsi que le travail présenté dans ce mémoire, sont basés sur le modèle de programmation de CUDA, parce que nous allons nous limiter aux plate-formes NVIDIA. Néanmoins, il est possible d'utiliser d'autres modèles de programmation, en l'occurrence celui de OpenCL [OpenCL]. OpenCL propose un modèle de programmation multi-plateforme qui cible plusieurs systèmes parallèles hétérogènes tels que les GPU (NVIDIA, AMD, Intel, etc), les CPU multi-cœurs et les processeurs CELL de la PlayStation 3.

Le modèle CUDA est assez largement décrit dans la littérature [SK10, CUDAa] et des plate-formes de documentation en ligne lui sont consacrées [2, 3].

2.3.1 Architectures considérées

Les cartes graphiques NVIDIA sont classées selon leur architecture. L'architecture correspond à des évolutions chronologiques de la structure et de l'organisation matérielles du GPU et du modèle de programmation qui permet de l'utiliser. Dans le cadre de ce travail, nous avons eu accès à différents modèles de cartes graphiques qui appartiennent à deux architectures :

— Fermi [Fermi] ;
— Kepler [Kepler].

Pour distinguer les révisions d'une architecture, NVIDIA utilise un identifiant numérique appelé compute capability, de la forme $x.y$, où x désigne l'architecture (par exemple 2 pour Fermi et 3 pour Kepler) et y désigne la révision [CUDAa]. À titre d'exemples, nous trouvons pour l'architecture Fermi les compute capabilities 2.0 et 2.1 et pour l'architecture Kepler les compute capabilities 3.0, 3.5 et 3.7. Les cartes que nous utilisons sont soit de compute capability

2. NVIDIA CUDA ZONE : https://developer.nvidia.com/cuda-zone
3. GTC On-Demand : http://on-demand-gtc.gputechconf.com/gtcnew/on-demand-gtc.php

2.0 soit de compute capability 3.0. Les mécanismes et les spécifications que nous allons décrire dans la suite référent à ces deux compute capability.

Nous utilisons les cartes suivantes :

— GeForce GTX 580 (Fermi, compute capability 2.0) ;
— Tesla M2050 (Fermi, compute capability 2.0) ;
— GeForce GTX 680 (Kepler, compute capability 3.0).

Il existe une autre classification des cartes graphiques NVIDIA selon la gamme : GeForce, Quadro et Tesla. La gamme GeForce est une gamme « grand public » destinée principalement au marché des jeux vidéo. La gamme Quadro est destinée aux « professionnels » et cible des applications de type CAO (Conception Assistée par Ordinateur). La gamme Tesla a été conçue pour des applications de calcul scientifique, en fournissant des fonctionnalités spécifiques telles que les codes ECC (Error Correction Codes) qui certifient l'exactitude des calculs et la fonctionnalité Direct GPU qui améliore les communications. Contrairement à la classification selon l'architecture, la classification selon la gamme est peu pertinente dans notre contexte.

2.3.2 Modèle d'exécution

Le CPU qui est connecté au GPU et qui le contrôle est appelé *hôte* (*host*). Le GPU est désigné par le terme *périphérique* (*device*). Un programme CUDA est composé d'un code exécuté sur l'*hôte* et d'un code exécuté sur le *périphérique*. Le code exécuté sur le périphérique est composé de fonctions, appelés *kernels*, qui lorsqu'elles sont appelées sont exécutées plusieurs fois en parallèle par plusieurs *threads* CUDA en suivant le mode SPMD (Single Program, Multiple Data).

L'exécution typique d'un programme CUDA qui lance un seul *kernel* est composée des actions suivantes, qui sont spécifiées explicitement par le programmeur dans le code, qui exploite à ces fins des appels de l'interface CUDA (voir figure 2.2) :

1. créer des données sur la mémoire de l'*hôte* ;
2. transférer les données vers la mémoire du *périphérique* ;
3. lancer un *kernel* depuis le CPU ;
4. exécuter le *kernel* en parallèle sur le GPU ;
5. synchroniser le GPU et le CPU ;
6. copier les résultats vers la mémoire du CPU.

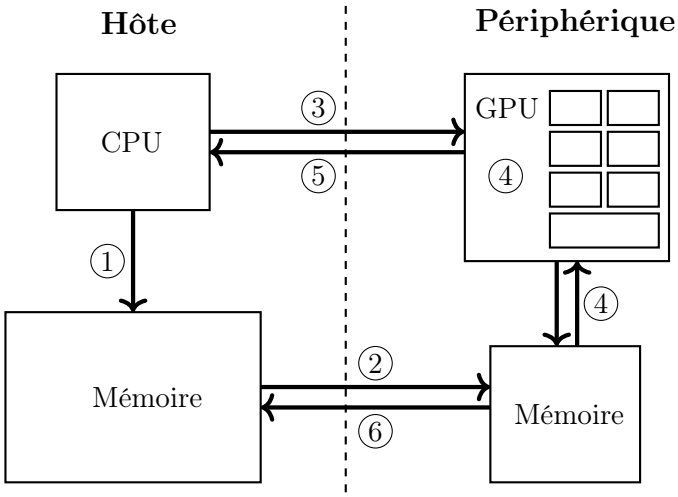


FIGURE 2.2 – Représentation chronologique de l'exécution d'un programme CUDA.

2.3.3 Hiérarchie GBT

Les *threads* sont organisés en *blocks* (blocs en français). Un ensemble de *blocks* forme un *grid* (grille en français). Cette hiérarchie est appelée la hiérarchie GBT (*Grid* de *Blocks* de *Threads*) (voir figure 2.3) :

— le *thread* est une instance du *kernel*. Il possède un identifiant unique dans son *block* ;
— le *block* est constitué de *threads* qui peuvent communiquer et se synchroniser. Un *block* a un identifiant unique dans son *grid* ;
— le *grid* est l'ensemble des *blocks* qui exécutent le même *kernel*.

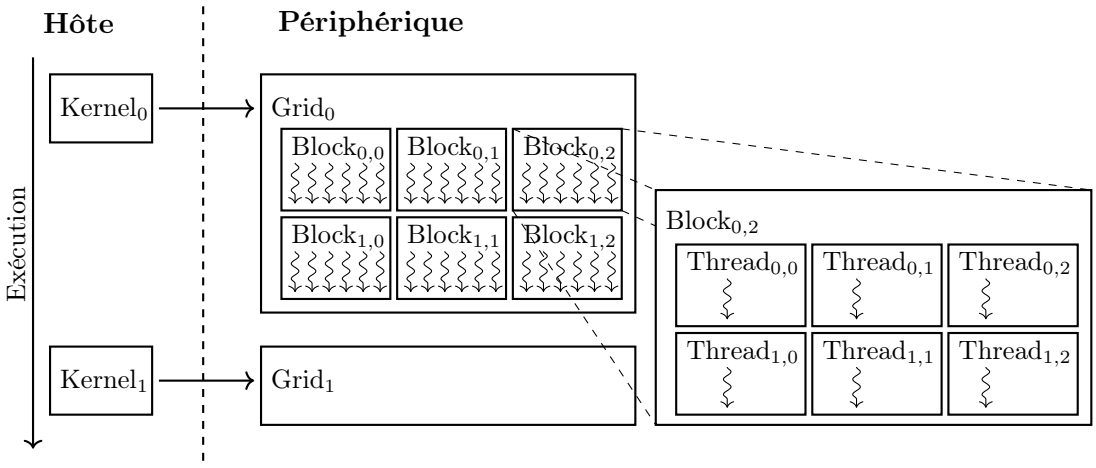


FIGURE 2.3 – Organisation du calcul selon la hiérarchie GBT.

Le programmeur est amené à définir les paramètres de dimension et de taille de la hiérarchie GBT en accord avec les spécifications de son application :

— la dimension du *grid* de *blocks* ? 1D, 2D ou 3D ;
— les longueur, largeur et hauteur du *grid* de *blocks* : x ?, y ? et z ? ($\max(x) = 65535$ pour Fermi et $\max(x) = 2^{31} - 1$ pour Kepler, $\max(y) = \max(z) = 65535$) ;
— la dimension du *block* de *threads* ? 1D, 2D ou 3D ;

— les longueur, largeur et hauteur du *block* de *threads* : x ?, y ? et z ? ($\max(x) = \max(y) = 1024, \max(z) = 64$ et $\max(threads) = 1024$).

L'ensemble de ces caractéristiques sont fondamentalement attachées au *kernel*.

2.3.4 Hiérarchie des cœurs

Le GPU comporte une grille composée d'un grand nombre de cœurs CUDA. Cette grille est appelée *Streaming Processor Array* (SPA) en vocabulaire CUDA. Les cœurs CUDA, aussi désignés par l'appellation *Streaming Processor* (SP), sont organisés en multiprocesseurs, qu'on appelle aussi *Streaming Multiprocessors* (SM pour Fermi et SMX pour Kepler). Le nombre de cœurs est de l'ordre du millier par carte, par exemple, la GeForce GTX 680 possède 1536 cœurs. Un SM contient 32 cœurs CUDA.

Lorsque le calcul d'un *kernel* est effectué sur la carte, les *blocks* sont distribués sur les SM disponibles. Un SM peut contenir simultanément jusqu'à 16 *blocks*. Dans une même génération de cartes graphiques, la puissance de calcul de la carte va être proportionnelle au nombre de SM. Un GPU avec deux fois plus de SM va mettre deux fois moins de temps pour exécuter un *kernel*.

Chaque SM groupe les *threads* appartenant à un même *block* en des ensembles de 32 *threads*, appelés *warps*. Le SM crée, ordonnance et exécute les *warps* sur ses différents cœurs. Chaque instruction du *kernel* est exécutée par les 32 *threads* du *warp* simultanément. Le mode d'exécution des *threads* du même *warp* est de type SIMD. Toutefois, il est possible que deux *threads* d'un même *warp* prennent deux branches différentes dans le code, on parle alors d'une divergence de code ou de divergence de branche. Dans ce cas, le paradigme SIMD n'est plus maintenu ; les deux branches ne sont pas exécutées en parallèle, mais l'une après l'autre, ce qui affecte négativement les performances.

Si nous cherchons à établir un parallèle entre la hiérarchie GBT et la hiérarchie des cœurs, le *grid* est exécuté sur le SPA ; il y a une analogie entre le *block* et le SM, même si un SM exécute plusieurs *blocks* ; et les *thread* correspondraient aux cœurs CUDA.

2.3.5 Hiérarchie de la mémoire

CUDA fournit plusieurs espaces mémoire qui se différencient de par leur taille, la vitesse de leurs accès, leur usage et les mécanismes de cache qui leur sont dédiés. Il existe des mémoires sur puce (*on-chip*), dont la taille est relativement petite et dont les accès sont rapides, et des mémoires hors-puce (*off-chip*), qui composent la mémoire DRAM (Dynamic Random-Access Memory) et dont les tailles sont plus importantes et les accès plus lents.

La hiérarchie de la mémoire est composée des mémoires suivantes :
— les registres dans chaque SM ;
— la mémoire locale dans la DRAM ;
— la mémoire partagée dans chaque SM ;
— la mémoire globale dans la DRAM ;
— la mémoire texture dans la DRAM ;
— la mémoire constante dans la DRAM.

Il existe plusieurs niveaux de cache d'accès :
— un cache de premier niveau (L1) dans chaque SM ;
— un cache de second niveau (L2) partagé par tous les SM ;
— un cache texture dans chaque SM.

Les caches L1 et L2 accélèrent les accès aux mémoires locale et globale, alors que le cache texture accélère les lectures de la mémoire texture. Le cache L1 partage un espace commun avec la mémoire partagée. Sa taille est configurable ; elle peut être 16 ou 48 ko pour les GPU de compute capability 2.0 et 16 , 32 ou 48 ko pour les GPU de compute capability 3.0 dans chaque SM. La taille du cache L2 est 768 ko pour les GPU de compute capability 2.0 et 1536 ko pour les GPU de compute capability 3.0. La taille du cache texture est 12 ko par SM.

Dans la figure 2.4, nous schématisons une vue d'ensemble des différents mémoires et caches, ainsi que les entités de la hiérarchie GBT qui leur correspondent. Nous donnons ensuite des détails sur chaque mémoire.

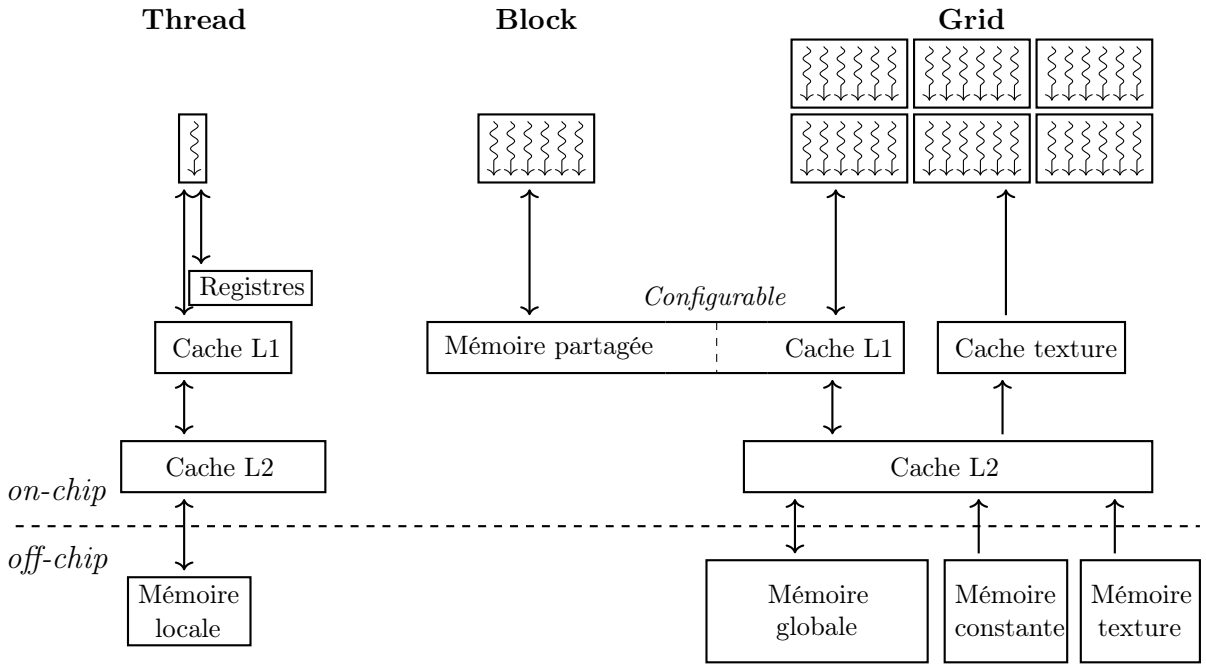


FIGURE 2.4 – Vue d'ensemble des espaces mémoire et des caches et les correspondances avec les entités de la hiérarchie GBT.

Registres

Le premier espace de mémoire est celui des registres qui sont utilisés par les *threads* pour effectuer les instructions et stocker les variables locales. Les registres sont des registres 32 bits. Leur nombre est de 32k par SM pour les GPU de compute capability 2.0 et de 64k par SM pour les GPU de compute capability 3.0. Ces registres sont partagés par tous les *threads* exécutés sur le SM. Comme le nombre de *threads* qui sont exécutés simultanément sur un SM est de l'ordre de quelques centaines voire du millier, le nombre des registres alloués à un *thread* est de l'ordre de quelques dizaines (au plus 63).

Mémoire locale

La mémoire locale complète l'espace mémoire réservé au *thread*. Elle est utilisée pour stocker des structures qui consommeraient beaucoup de registres. Comme c'est une mémoire *off-chip* (une partie de la DRAM), ses accès sont lents, mais bénéficient des niveaux de cache L1 et L2.

Mémoire partagée

La mémoire partagée est une mémoire dédiée à l'échange entre les *threads* d'un même *block*. Typiquement, elle est utilisée quand on a besoin de combiner des résultats partiels de plusieurs *threads*. La mémoire partagée se trouve dans chaque SM et elle est commune à tous les *blocks* qui sont exécutés sur le SM. La mémoire partagée est une mémoire on-chip, c'est l'espace mémoire qui a le plus grand débit et la plus petite latence, si nous ne considérons pas les registres. Par exemple, l'accès à la mémoire partagée est environ 100 fois plus rapide que l'accès à la mémoire globale [Har13a].

Dans chaque SM, l'espace mémoire dédié à la mémoire partagée est commun avec le cache L1. Le programmeur peut partitionner cet espace avec 2 configurations possibles : (48ko de cache L1 et 16ko de mémoire partagée) ou (16ko de cache L1 et 48ko de mémoire partagée). La configuration d'équipartition (32ko de cache L1 et 32ko de mémoire partagée) est uniquement supportée par les GPU de compute capability 3.0.

La mémoire partagée est organisée en 32 bancs (ou *banks*) de sorte à ce que des mots de 32 bits successifs se trouvent dans des bancs successifs [4]. Lorsque les *threads* d'un *warp* accèdent aux bancs, les requêtes ne doivent pas générer de conflit de banc. Un conflit de banc se produit lorsque les *threads* accèdent à des mots distincts appartenant à un même *banc*. Lorsque les *threads* accèdent au même mot (multicast), il n'y a pas de conflit de banc. Dans la figure 2.5, nous donnons quelques exemples d'accès à la mémoire partagée et nous précisons si le schéma d'accès génère ou non un conflit de banc.

Un schéma (*pattern*) d'accès mémoire qui ne génère pas de conflit permet de paralléliser les requêtes, autrement dit d'avoir une seule requête pour tous les *threads* du *warp*. Dans le cas contraire, un conflit de banc se traduit par une séquentialisation des requêtes. Évidemment, minimiser voire éviter les conflits de bancs permet de maximiser les performances d'accès à la mémoire partagée.

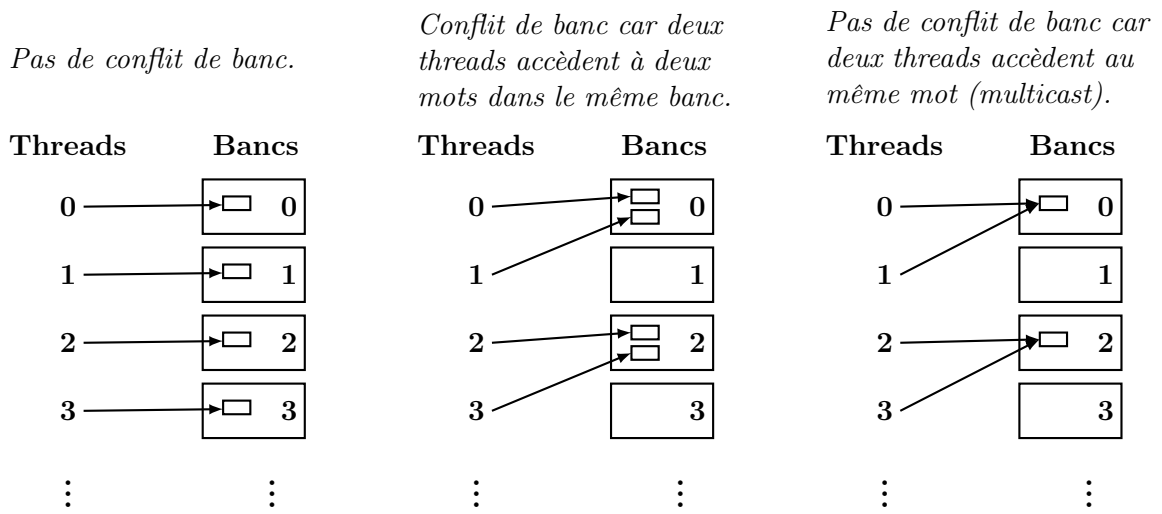


FIGURE 2.5 – Exemples de schémas d'accès à la mémoire partagée.

4. Pour les GPU de compute capability 3.0, il existe un mode 64 bits, où des mots de 64 bits se trouvent dans des bancs successifs.

Mémoire globale

La mémoire globale sert d'une part pour les transferts entre l'hôte et le périphérique, mais permet aussi de récupérer les entrées du *kernel* et d'écrire ses résultats finaux. Tous les *threads* du *grid* peuvent lire et écrire des données dans la mémoire globale.

La taille de la mémoire globale n'est pas une spécification de NVIDIA, pour un modèle de GPU donné. Sur le marché, nous pouvons trouvé deux à trois tailles possibles. Par exemple, pour le modèle GTX 680, nous trouvons sur le marché des cartes avec 2 Go ou 4 Go de DRAM.

Les accès à la mémoire globale sont « cachés » dans les caches L1 et L2 pour les GPU de compute capability 2.0 et seulement dans le cache L2 pour les GPU de compute capability 3.0. Une ligne du cache est de taille 128 octets. Lorsqu'un *warp* accède à la mémoire globale, sa requête est décomposée en plusieurs requêtes de 128 octets. En l'occurrence, si la taille d'un mot accédé par un *thread* est 4 octets, une seule requête est effectuée ; si la taille des mots est 8 octets pour un *thread*, deux requêtes sont effectuées.

Il existe un mécanisme de fusion d'accès (ou *coalescing*) du *warp* qui tend à fusionner les requêtes des 32 *threads*, c'est-à-dire permet de les effectuer simultanément. L'efficacité de ce mécanisme dépend de la localité des données. Lorsque nous observons les modèles des accès mémoire, il existe un schéma idéal, lorsque les accès sont contigus et alignés. Dans ce cas, la fusion est totale et on atteint les performances d'accès optimales. Deux autres schémas sont possibles. Le premier schéma correspond au cas où les accès des 32 *threads* restent contigus mais ne sont plus alignés, on parle alors d'un *offset* non nul. Le second schéma correspond au cas où les 32 *threads* accèdent à des zones espacées, on parle alors d'un *stride* différent de 1. Nous avons schématisé dans la figure 2.6 ces différents cas en considérant un accès d'un mot de 4 octets pour chaque *thread*.

Harris détaille dans [Har13b] le mécanisme de fusion d'accès et présente les résultats de mesure du débit d'accès à la mémoire globale lorsqu'on varie l'*offset* et le *stride*. Dans ses benchmarks, il a utilisé des cartes de compute capability 1.0, 1.3 et 2.0. Nous avons refait l'expérience de Harris sur une carte GeForce GTX 680 (compute capability 3.0) [5]. Les résultats sont visualisés dans la figure 2.7 et sont assez proches des résultats obtenus par Harris sur les GPU de compute capability 2.0. Pour le cas de l'offset non nul, le mécanisme de fusion d'accès n'est pas affecté et nous continuons à avoir de bonnes performances avec les GPU dont la compute capability est supérieure à 2.0 [6]. Pour le cas du *stride* différent de 1, nous avons une baisse des performances proportionnelle à la largeur du *stride*. Il est aussi envisageable de combiner les deux cas précédents, les performances sont alors proches de ce que nous observons pour le cas du *stride* différent de 1.

5. Code disponible sur GitHub : https://github.com/parallel-forall/code-samples/blob/master/series/cuda-cpp/coalescing-global

6. Pour les GPU dont la compute capability est inférieure à 2.0, Harris a observé une baisse des performances.

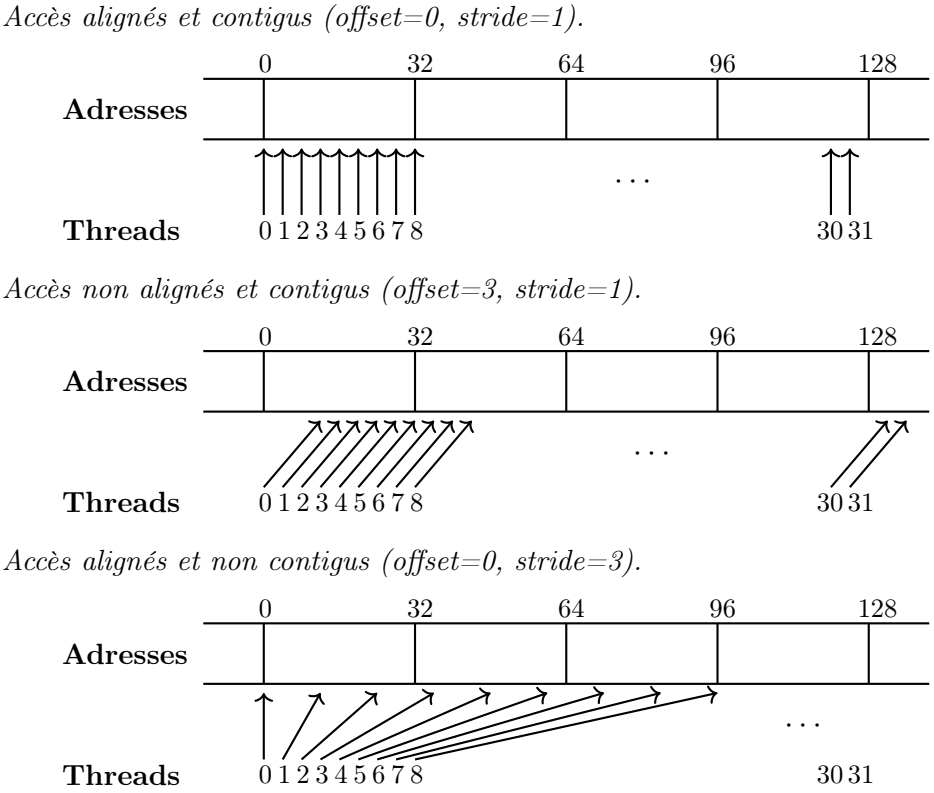


FIGURE 2.6 – Différents schémas d'accès à la mémoire globale par un *warp*. Chaque *thread* accède à un mot de 4 octets.

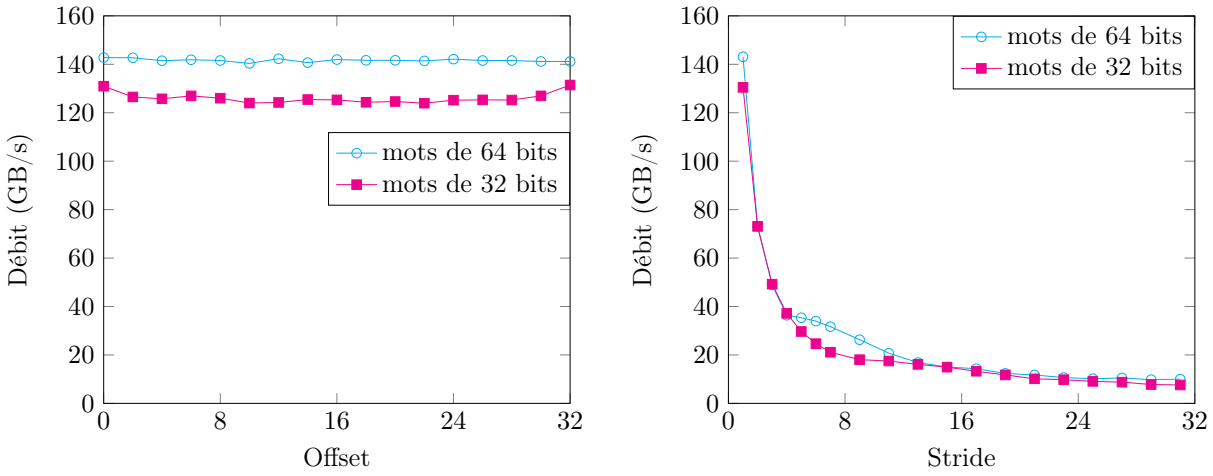


FIGURE 2.7 – Variation du débit effectif d'accès à la mémoire globale en fonction de l'*offset* (à gauche) et du *stride* (à droite) en précision simple et double sur une GTX 680.

Dans certains cas, nous pouvons être amenés à avoir nécessairement des schémas d'accès mémoire avec un *stride* plus grand que 1. La mémoire partagée est alors utilisée comme intermédiaire pour fusionner les accès. Par exemple, pour une lecture, on procède en deux temps ; on effectue une première copie des données de la mémoire globale à la mémoire partagée avec

des accès contigus, ensuite les *threads* lisent les données de la mémoire partagée avec des accès espacés. Ceci fonctionne parce que les accès à la mémoire partagée ne sont pas pénalisés par un espacement.

Mémoire texture

Les textures sont des concepts hérités de l'univers du traitement de l'image et de l'infographie qui représentent le cadre d'application traditionnel des GPU. Dans ces contextes, l'utilisation des textures permet d'exploiter la localité spatiale et de traiter d'une manière efficace des pixels adjacents. Dans un contexte de calcul générique, la localité spatiale correspond à un *thread* qui accède à une adresse proche des adresses accédées par des *threads* voisins.

La mémoire texture est une partie allouée de la mémoire globale ; c'est une zone mémoire off-chip en lecture seule. Elle est cachée dans le cache texture qui est localisé sur la puce. Lorsque l'application possède des schémas d'accès mémoire qui présentent une localité de données, utiliser la mémoire texture plutôt que la mémoire globale accélère les opérations de lecture.

Mémoire constante

La mémoire constante est une autre partie allouée de la mémoire globale. La mémoire constante est en lecture seule ; son mécanisme d'accès diffère de celui de la mémoire globale. La latence des accès en lecture à la mémoire constante est significativement plus petite que celle des accès à la mémoire globale. Toutefois, sa taille est limitée à 64 ko.

Type	Entité	Pénalité sur les temps d'accès
Registres	*thread*	1×
Locale	*thread*	100×
Partagée	*Block*	1×
Globale	*Grid*	100×
Texture	*Grid*	Dépend de l'application
Constante	*Grid*	1×

TABLE 2.1 – Récapitulatif des caractéristiques des mémoires.

Nous avons identifié dans cette hiérarchie de mémoire 9 niveaux de mémoire et de cache. L'ensemble de ces niveaux vont intervenir lors de l'implémentation des briques de l'algèbre linéaire. Nous allons dans le chapitre 5 détailler la façon avec laquelle les différents niveaux sont pris en compte et l'impact de ces choix.

2.3.6 Synchronisation des threads

Nous avons déjà vu que le *block* correspond à l'ensemble des *threads* qui sont amenés à collaborer entre eux pour effectuer une tâche. Ces *threads* peuvent échanger des données via la mémoire partagée et se synchroniser. Seuls les *threads* d'un même *warp* sont exécutés physiquement en même temps. Dans le cas où on a un intérêt à synchroniser l'ensemble des *threads* du *block*, il faut appeler une primitive de barrière de synchronisation, qui va garantir que tous les *threads* atteignent la même instruction avant de continuer. Une synchronisation de *threads* appartenant à des *warps* distincts est par conséquent plus coûteuse qu'une synchronisation de *threads* du même *warp*.

Dans certaines applications, on peut avoir un intérêt à synchroniser et à faire communiquer des *threads* appartenant à des *blocks* distincts. Dans ce cas, l'échange de données est réalisé à travers la mémoire globale. La synchronisation s'effectue en invoquant séparément deux *kernels* ; un premier *kernel* à la fin duquel tous les *threads* du *grid* écrivent à la mémoire globale et un second *kernel* au début duquel tous les *threads* du *grid* lisent les données de la mémoire globale. Ce schéma de synchronisation est évidemment alourdi par les invocations des *kernels* et les transferts de données.

Le jeu d'instructions CUDA fournit des instructions atomiques en mémoire partagée et en mémoire globale qui garantissent qu'une opération est effectuée par un *thread* sans l'interférence d'autres *threads*. Le jeu d'instructions fournit également pour les *threads* d'un *warp* des opérations d'échange de variable (*shuffle*), qui sont utiles dans des calculs de collecte de résultats [CUDAa].

2.3.7 Programmation plus bas niveau

L'environnement de programmation de NVIDIA propose en complément au language haut-niveau CUDA, un langage pseudo-assembleur, PTX (Parallel Thread Execution) [PTXa]. PTX fournit un jeu d'instructions, qu'un programmeur utilise pour écrire un code complet en PTX ou pour insérer des fonctions PTX dans un code CUDA [PTXb].

2.3.8 Bonnes pratiques pour l'optimisation du code CUDA

Le programmeur d'une application CUDA doit appliquer un ensemble de « bonnes pratiques » lors du design d'un *kernel* qui vont permettre de tirer avantage d'une manière efficace de la puissance de calcul du GPU. Nous allons décrire une liste non exhaustive de pratiques. Certaines pratiques sont spécifiques aux architectures Fermi et Kepler. Ces idées sont détaillées dans [CUDAa, chap. 5] et [CUDAb].

Optimisation algorithmique

Tout d'abord, il est important de n'employer le GPU que pour des algorithmes adaptés à la programmation parallèle. Il faut cibler des calculs (ou des parties de calcul) coûteux pour pouvoir amortir le temps des transferts *hôte-périphérique* qui sont importants. Par la suite, on est amené à adapter le *kernel* aux caractéristiques architecturales du GPU pour profiter de l'accélération fournie par le périphérique. En particulier, nous précisons trois actions principales :

— Il faut partager le travail sur les *threads* et les *blocks* de sorte à ce que les charges des multiprocesseurs soient équilibrés.

— Il faut privilégier le calcul par rapport au transfert. En effet, le GPU dispose d'une grande puissance arithmétique, alors que ses accès mémoire peuvent être très lents. Dans certains cas, recalculer une donnée est plus efficace que de la chercher en mémoire. La latence d'une opération arithmétique est de l'ordre de 20 cycles, celle d'un accès mémoire peut aller jusqu'à plus de 400 cycles [Vol10].

— Il faut exploiter la synchronisation physique du *warp*, réduire la synchronisation entre les *threads* du *block* qui engendre un surcoût et éviter le plus possible la synchronisation entre différents *blocks*.

Limitation de la divergence de code

Nous avons déjà mentionné la divergence de code, qui se produit lorsqu'au moins deux *threads* d'un même *warp* exécutent des instructions différentes. Typiquement lorsque nous avons unique-

ment deux instructions différentes, le SM séquentialise l'exécution des deux branches.

C'est pourquoi lorsqu'on réfléchit à comment distribuer le travail sur la hiérarchie GBT, il est plus intéressant de faire en sorte qu'un *warp* ne contienne que des *threads* qui exécutent un même code.

Dans certains cas, on peut être obligé d'avoir un comportement différent pour un ou plusieurs *threads* du *warp*. Une solution pour maintenir le paradigme SIMD est d'avoir la divergence dans les données plutôt que dans les instructions. Cette approche n'est réalisable que dans certains cas.

Amélioration de la bande passante de la mémoire

Les accès mémoire représentent le plus souvent le goulot d'étranglement d'une application CUDA. Il est important d'exploiter efficacement les différents niveaux de mémoires et de caches :

- Il faut privilégier l'utilisation des mémoires qui résident sur la puce (la mémoire partagée) et les mémoires qui ont une meilleure bande passante (la mémoire texture et la mémoire constante). En particulier, il faut utiliser la mémoire partagée plutôt que la mémoire globale tant que les tailles des données le permettent.
- Il faut mettre en place des schémas d'accès à la mémoire globale qui maximisent la fusion des accès, ainsi que des schémas d'accès à la mémoire partagée qui minimisent les conflits de bancs.
- Dans certains cas, il est intéressant d'utiliser la mémoire partagée comme intermédiaire pour fusionner des accès espacés à la mémoire globale (voir la sous-section 2.3.5).
- Il faut aussi mettre en place des schémas d'accès qui optimisent les caches (*cache-friendly*), en particulier le cache L1 et le cache texture.

Augmentation du remplissage

Le taux de remplissage (*occupancy* en terminologie NVIDIA) représente une mesure de l'utilisation des SM ; il est calculé comme le ratio moyen du nombre de *warps* exécutés sur un SM par le nombre maximal de *warps* [CUDAa]. Un taux de remplissage élevé indique que beaucoup de *threads* sont exécutés simultanément, ce qui permet de dissimuler la latence arithmétique, c'est-à-dire le temps nécessaire à un *thread* pour effectuer une opération. Pour un même ensemble de calculs menés, un taux de remplissage élevé améliore les performances ; alors que l'inverse, c'est-à-dire un taux de remplissage faible se traduit généralement par un impact négatif sur les performances.

L'utilisation des ressources est le facteur principal qui agit sur le remplissage. En effet, le nombre des registres et la quantité de mémoire partagée sont limités sur un SM. Si un *thread* consomme « beaucoup » de registres ou si un *block* alloue « beaucoup » de mémoire partagée, l'ordonnanceur va maintenir moins de *warps* par SM [LR11]. Le nombre de *threads* par *block* a aussi un impact sur le remplissage. En effet, un SM peut exécuter au plus 8 *blocks*. Par conséquent, si la taille des *blocks* est petite, nous aurons peu de *threads* qui sont exécutés en parallèle et nous ne pouvons plus masquer la latence des opérations. On masque la latence d'une opération en recouvrant avec d'autres opérations. Généralement, le nombre de *threads* par *block* devrait être entre 128 et 512 pour avoir des performances correctes.

Néanmoins, augmenter le remplissage n'est pas nécessairement l'unique stratégie pour masquer la latence. Volokov décrit dans [Vol10] une autre approche qui, au lieu d'augmenter le remplissage, va tendre à le réduire en rajoutant du parallélisme d'instructions. En d'autres termes, ceci consiste à diminuer le nombre de *threads* qui sont exécutés simultanément, en ayant plus

d'instructions par *thread*. Le parallélisme d'instructions va pouvoir masquer la latence. Dans certaines applications, cette stratégie améliore les performances plus que la stratégie d'augmenter le remplissage.

Les *kernels* que nous allons rencontrer ici, sont des *kernels* dont les performances sont limitées par les accès mémoire. Pour ces *kernels*, la stratégie d'augmenter le remplissage est plus rentable. C'est pourquoi dans ce travail, en particulier au chapitre 5, nous serons amenés à augmenter le remplissage, parce qu'il fournit une amélioration des performances des *kernels*.

2.3.9 Mesure de performance

NVIDIA offre plusieurs outils d'analyse et de profilage qui permettent de mesurer les performances d'une application et d'identifier les facteurs qui limitent les performances. Nous allons principalement utiliser deux outils :

— CUDA Occupancy Calculator : outil de calcul du remplissage, qui se présente sous la forme d'un tableur [7], où le programmeur indique les ressources consommées (registres et mémoire partagée) et le nombre de *threads* par *block* ;

— `nvvp` : outil de profilage graphique configurable (son équivalent non graphique est `nvprof`) qui effectue des mesures de métriques sur la consommation des ressources, l'efficacité des accès mémoire, la divergence de code, etc.

2.4 Passage à l'échelle d'un cluster de calcul

Maintenant, nous nous intéressons à l'exécution d'une application sur un cluster de calcul. Le cluster est composé d'un ensemble de nœuds. Les nœuds sont interconnectés par un réseau de communication à haut débit, typiquement de l'InfiniBand (IB). Nous considérons les applications où les différents nœuds de calcul partagent des données de taille importante, ce qui correspond à notre contexte d'algèbre linéaire. Ce sont donc les aspects liés aux communications que nous allons détailler. Nous allons nous limiter aux architectures qui nous intéressent : les CPU multi-cœurs et les GPU.

2.4.1 Le réseau InfiniBand (IB)

Le réseau InfiniBand est un réseau de communication à haute performance qui fournit une latence de l'ordre de la microseconde et un débit de plusieurs dizaines de Gbit/s (voir dans la table 2.2 la latence et le débit des différents réseaux InfiniBand). Il est possible d'utiliser la couche native IB verbs ou de passer par une bibliothèque de communication (tel que MPI) qui utilise directement la couche IB verbs.

	Latence	Débit théorique
InfiniBand DDR	2.5 µs	16 Gb/s
InfiniBand QDR	1.3 µs	32 Gb/s
InfiniBand FDR-10	0.7 µs	41.25 Gb/s
InfiniBand FDR	0.7 µs	54.54 Gb/s
InfiniBand EDR	0.5 µs	100 Gb/s

TABLE 2.2 – Latence et débit des différents réseaux InfiniBand.

7. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Dans le cadre de ce travail, nous utiliserons des clusters équipés du réseau InfiniBand QDR ou du réseau InfiniBand FDR.

2.4.2 Communications CPU

Nous supposons que le cluster est composé de machines multi-cœurs. Supposons que nous avons des processus ou des threads [8] qui sont exécutés sur les cœurs des machines. Nous avons alors deux types de communication :

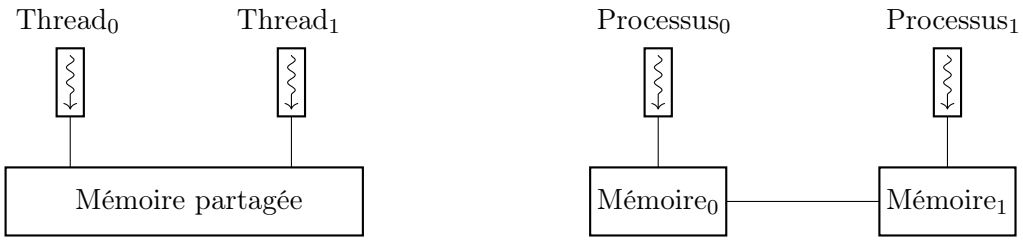
— les communications entre les processus exécutés sur le même nœud ;
— les communications entre les processus exécutés sur des nœuds distincts.

Ces deux types de communication sont désignés respectivement par les communications *intra-nœud* et les communications *inter-nœuds*.

Il existe deux modèles de communication. À chacun de ces modèles correspond un certain nombre d'outils et d'intergiciels disponibles. La question à ce niveau est de comparer ces deux modèles, en particulier en terme du coût lié à l'utilisation de leurs outils respectifs et à quel point ces outils correspondent aux contraintes de notre application.

— Le modèle mémoire partagée, utilisé dans OpenMP et POSIX Threads, où les *threads* communiquent en partageant leurs données sur une mémoire commune (voir figure 2.8a).
— Le modèle de passage de messages, où la mémoire est distribuée sur tous les processus qui, pour échanger des données, sont amenés à effectuer des transferts (voir figure 2.8b). Les modèles de passage de messagess, typiquement MPI, utilisent généralement des procédés de type RDMA (Remote Direct Memory Access), qui ne font pas intervenir les CPU, plutôt que des procédés de type *polling* qui font intervenir les CPU. Par exemple, les implémentations MPI utilisent le RDMA sur les réseaux InfiniBand et le RoCE (RDMA over Converged Ethernet) sur les réseaux Ethernet. Ces techniques sont plus efficaces en terme de latence et de débit.

Le modèle de passage de messages de MPI se positionne logiquement pour des communications *inter-nœuds*, alors que celui de OpenMP est plus approprié aux communications *intra-nœud*.



(a) Modèle à mémoire partagée (b) Modèle de passage de messages

FIGURE 2.8 – Modèles de communication.

Pour des applications qui regroupent les deux types de communication, comme celles que nous considérons ici, trois stratégies peuvent s'appliquer [RHJ13] :

— utiliser un même modèle, typiquement MPI, dans tous les communications, en assignant un ou deux processus par cœur ;

8. Différence entre procesus et *thread* : un processus est une instance du programme, un *thread* est un bout de code qui est exécuté en parallèle. Dans le modèle MPI, on parle de processus. Dans le modèle OpenMPI, on parle plutôt de threads.

— utiliser à la fois MPI et OpenMP (ou POSIX) où MPI gère les communications *inter-nœuds* et OpenMP gère les communications *intra-nœud*, en assignant à chaque nœud un processus MPI, qui exécute sur les cœurs du nœud plusieurs *threads* OpenMP ;
— utiliser MPI pour les communications *inter-nœuds* et les directives MPI-3 d'utilisation de la mémoire partagée pour les communications *intra-nœud*.

La première stratégie est la plus simple à mettre en œuvre vu qu'un seul modèle est utilisé. Toutefois, les communications *intra-nœud* ne sont potentiellement pas optimales, même si elles seront plus rapides que les communications *inter-nœuds* parce qu'elles ne sont pas effectuées à travers le réseau. La seconde stratégie permet de diminuer la mémoire consommée, vu que les *threads* OpenMP partagent de la mémoire, mais peut présenter certaines difficultés liées à l'inadéquation des deux modèles, typiquement les implémentations MPI actuelles ne supportent pas beaucoup le *multi-threading*. La troisième stratégie représente théoriquement l'alternative la plus prometteuse à l'utilisation MPI+OpenMP.

2.4.3 Communications GPU

Nous supposons maintenant que les nœuds du cluster hébergent des GPU et nous discutons de comment transférer les données d'un GPU à un autre. De même que pour les communications CPU, nous devons traiter deux cas possibles (et non exclusifs) : le premier quand un seul nœud héberge deux GPU ou plus, et le second quand les GPU sont sur des nœuds distincts. Par souci de simplification, nous considérons uniquement deux GPU et le scénario d'envoyer des données de l'un à l'autre. On peut facilement étendre ceci vers le cas d'un plus grand nombre de GPU et pour construire des routines de communication plus compliquées.

Communications *intra-nœud*

CUDA offre trois schémas de communication possibles :
— Passer par le CPU : la communication doit impliquer l'*hôte*. Elle est composée de deux transferts, une copie du *périphérique* vers l'*hôte* (D2H : Device to Host), puis une copie de l'*hôte* vers le *périphérique* (H2D : Host to Device) (voir figure 2.9).

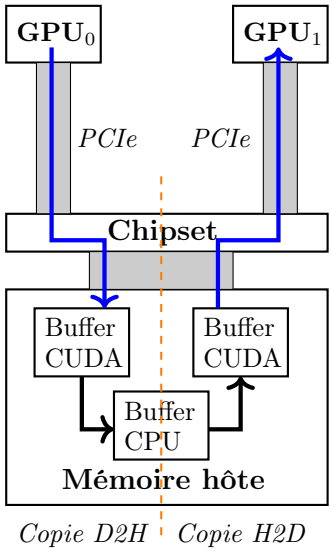


FIGURE 2.9 – Communication GPU *intra-nœud* en passant par le CPU.

— Copie du *périphérique* vers le *périphérique* (D2D pour Device to Device) : du point de vue du programmeur, il s'agit une copie directe des *buffers* GPU. Toutefois, le transfert continue de passer par la mémoire de l'*hôte*. Par rapport au cas précédent, la copie est cependant complètement *pipelinée*, alors que dans le premier cas, les deux transferts sont séparément *pipelinés* (voir figure 2.10).

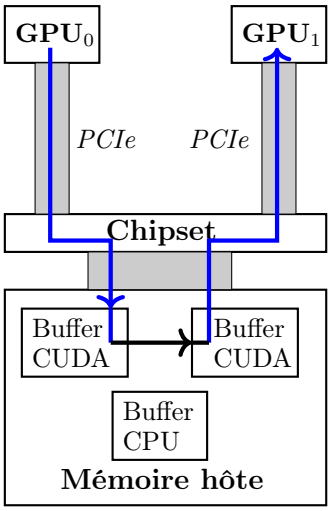


FIGURE 2.10 – Communication GPU *intra-nœud* avec copie du *périphérique* vers le *périphérique*.

— Accès Direct Pair à Pair (P2P DMA pour Peer-to-Peer Direct Memory Access) : la fonctionnalité permet aux GPU de partager des données indépendamment du CPU [Sch11] (voir figure 2.11). Cette fonctionnalité requiert d'activer l'accès pair à pair dans chaque GPU. La fonctionnalité P2P DMA réduit le surcoût du passage par la mémoire de l'*hôte* et ainsi accélère les transferts.

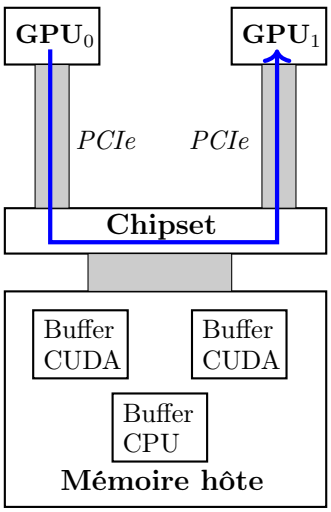


FIGURE 2.11 – Communication GPU *intra-nœud* avec un Accès Direct Pair à Pair.

Pour vérifier l'accélération fournie par le P2P DMA, nous avons mesuré les débits et les latences relatives à chaque approche. L'expérience a été réalisée avec deux cartes NVIDIA GeForce GTX 680. Nous avons mesuré le temps d'une copie d'un GPU vers un GPU pour des tailles crois-

santes de données. L'observation de la figure 2.12 montre que l'utilisation de la fonctionnalité P2P DMA accélère les communications en terme de latence et de débit.

	Latence en µs
Passer par CPU	24
Copie D2D	22
Copie P2P DMA	18

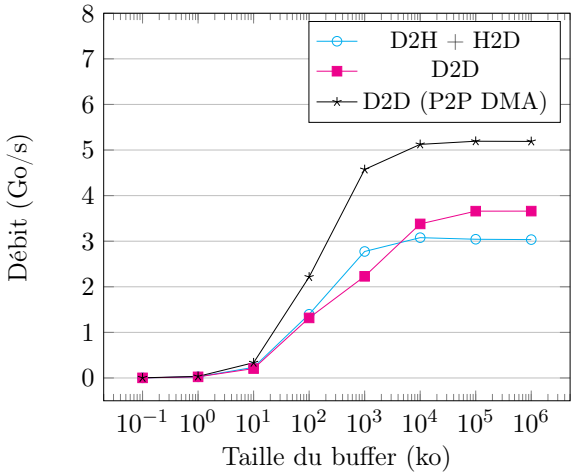


FIGURE 2.12 – Comparaison des communications GPU *intra-nœud* en terme de latence et de débit.

Communications *inter-nœuds*

L'option triviale est d'effectuer le transfert en trois étapes : effectuer une copie du GPU vers le CPU en utilisant les routines CUDA (copie D2H), enuite utiliser MPI pour faire une copie entre les CPU (copie H2H), et finalement une copie CUDA du CPU vers le GPU destinataire (copie H2D) (voir figure 2.13).

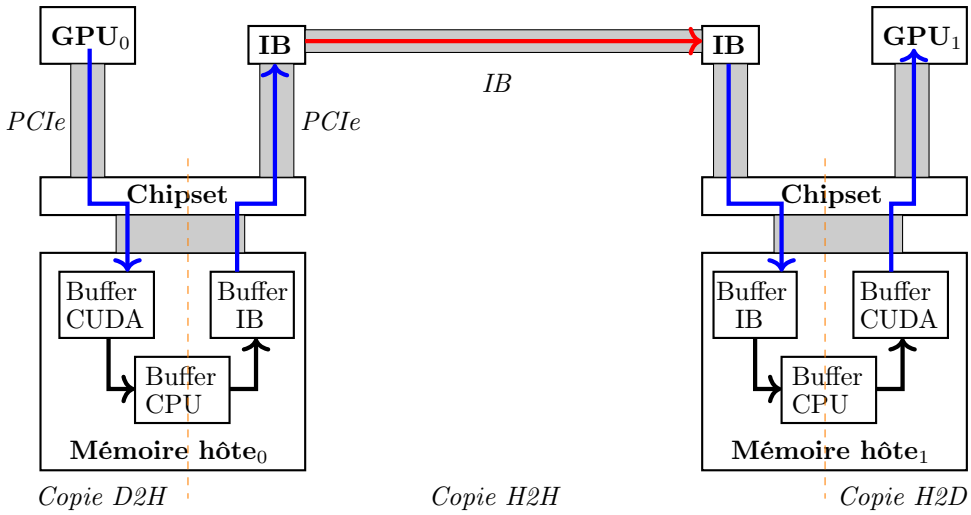


FIGURE 2.13 – Communication GPU *inter-nœuds* en 3 étapes.

Toutefois, il est possible d'éviter de passer par le CPU en utilisant la fonctionnalité *Cuda-aware MPI* qui combine MPI et CUDA [Kra13a, Kra13b]. Cette fonctionnalité permet d'utiliser les buffers GPU directement dans les routines MPI. Du point de vue du programmeur, le transfert de données se réduit à un seul appel à une routine MPI. Avec la fonctionnalité *Cuda-aware MPI*,

les transferts sont complètement *pipelinés*, alors qu'avec la première approche, chaque copie est séparément *pipelinée*. La fonctionnalité est intégrée dans plusieurs bibliothèques MPI, telles que MVAPICH2 (à partir de la version 1.8) et OpenMPI (à partir de la version 1.7).

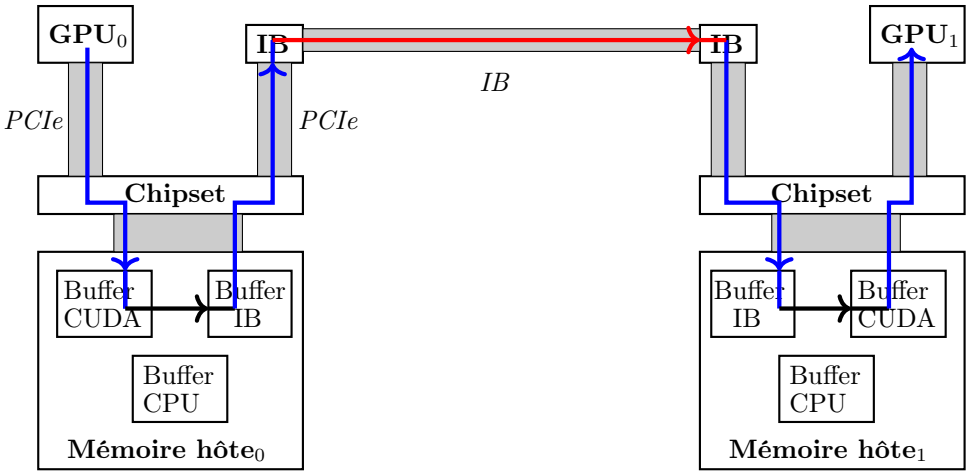


FIGURE 2.14 – Communication GPU *inter-nœuds* avec *Cuda-aware MPI*.

Pour estimer l'accélération qu'apporte la fonctionnalité *Cuda-aware MPI*, nous avons réalisé des mesures des communications GPU *inter-nœuds* entre deux cartes GeForce GTX 680 installées sur deux nœuds connectés par de l'InfiniBand QDR. Nous avons utilisé CUDA 5.0 et Open MPI 1.7.3. Nous avons ajouté aux mesures des communications GPU des mesures relatives à une copie d'un buffer CPU à un autre buffer CPU en utilisant les routines MPI, pour avoir une référence.

	Latence en µs
Copie D2D sans Cuda-aware MPI	15
Copie D2D avec Cuda-aware MPI	13
Copie H2H avec MPI	1

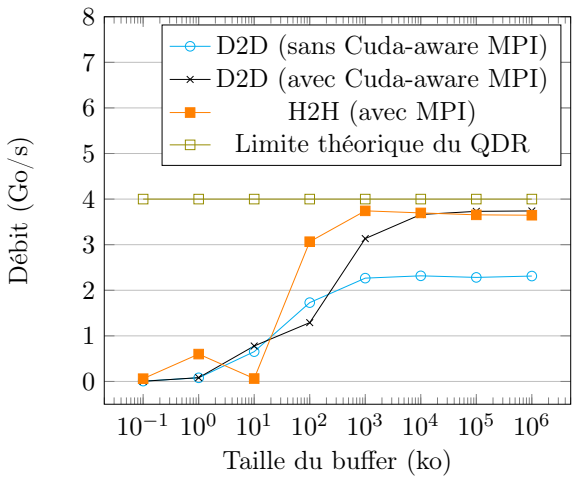


FIGURE 2.15 – Comparaison en terme de latence et de débit des communications GPU *inter-nœuds*, avec et sans *Cuda-aware MPI*, ainsi qu'une comparaison avec la communication entre *hôtes*.

La figure 2.15 montre que la fonctionnalité *Cuda-aware MPI* permet d'atteindre à partir d'une grande taille des données envoyées le débit d'un transfert MPI d'un buffer CPU à un autre buffer CPU. Ces deux débits sont proches de la limite théorique du QDR (4 Go/s), qui correspond au débit maximal qu'une communication *inter-nœuds* peut atteindre. Le gain fourni par l'accé-

lération *Cuda-aware MPI* est semblable à celui fourni par le P2P DMA pour les communications *intra-nœud* (voir figure 2.12).

Il existe une autre fonctionnalité, *GPUDirect*, qui optimise les communications [GPUDirect] en enlevant le passage par la mémoire de l'*hôte* (voir figure 2.16). En utilisant cette fonctionnalité, la latence d'un transfert de donnée est réduite par rapport à un transfert qui passe par le CPU. Toutefois, le *GPUDirect* ne devrait pas amener une amélioration au niveau du débit [PHV+13]. Nous n'avons pas pu déployer cette fonctionnalité dans notre application, car elle n'est supportée que par des cartes Tesla et Quadro récentes, auxquelles nous n'avions pas accès. Le lecteur intéressé par des mesures des performances de l'utilisation de *GPUDirect* peut se référer au travail de [PHV+13] qui rapporte des mesures en terme de latence et de débit.

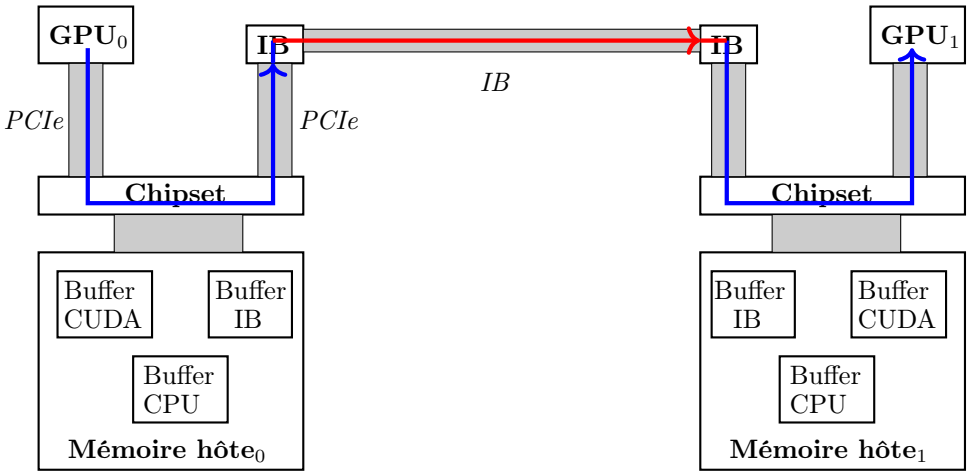


FIGURE 2.16 – Communication GPU *inter-nœuds* avec *Cuda-aware MPI* et *GPUDirect*.

Les mesures de performance effectuées pour les communications *intra-* et *inter-nœuds* ont été obtenues avec des communications bloquantes de type `MPI_Send` et `MPI_Recv`. Ces résultats sont très proches de ceux que nous avons obtenus avec des communications non-bloquantes de type `MPI_Isend` et `MPI_Irecv`.

2.5 Conclusion

Ce chapitre « technologique » a principalement servi pour la familiarisation du lecteur avec la programmation sur les GPU NVIDIA, aussi bien pour le cas d'un seul GPU, que pour le cas multi-GPU. Nous avons introduit la terminologie NVIDIA et le modèle de programmation CUDA. Nous avons décrit l'organisation des *threads*, des cœurs et des mémoires et discuté les mécanismes de communication pour les cas multi-CPU et multi-GPU. Nous aurons recours dans les chapitres suivants à la plupart de ces concepts et de ces mécanismes pour décrire les approches et les solutions proposées.