Projet de recherche (PRe)
Majeure : Mathématiques appliquées
Année scolaire : 2020

Plus de science pour moins de code : Portage de codes C sur GPU avec OpenACC

Non-confidentiel

Auteur : KREMPP Quentin **Promotion : 2021**

Tuteur ENSTA Paris : ALES Zacharie
Tuteur dans l'organisme d'accueil : MODAVE Axel

Stage effectué du 18/05/2020 au 31/07/2020
Organisation d'accueil : Equipe POEMS - Ensta Paris, Inria, CNRS
Adresse : 828 Boulevard des Maréchaux, 91120 Palaiseau

Table des matières

1 Note de non confidentialité

Ce document est non confidentiel. Il peut donc être consulté en ligne par tous.

2 Remerciements

Je remercie chaleureusement Axel MODAVE de m'avoir proposé ce sujet qui s'est avéré très enrichissant, et pour le suivi qu'ils m'ont offert avec Luis MALTEZ FARIA malgré les circonstances inhabituelles de cette crise sanitaire. Ils m'ont accompagné tout au long de ce projet, m'ont orienté et m'ont permis de produire un travail dont je suis fier. Je remercie aussi ceux qui m'ont entouré pendant la durée de mon télétravail quand il n'était pas tout les jours facile de se mettre au travail, Claire, Antoine et Théo.

3 Résumé

L'objectif de ce travail était d'étudier le standard *OpenACC*, qui vise à porter des codes en *C/C++* sur *GPU*, simplement en y ajoutant des directives indiquant au compilateur les portions que l'on souhaite paralléliser sur *GPU* et comment le faire. On appliquera cela à quelques codes simples et classiques pour établir les performances que l'on peut attendre d'une telle parallélisation, et établir qu'un gain de 40% de performances par rapport à un code parallélisé sur *CPU* à l'aide de *OpenMP* est possible avec un travail rudimentaire. Enfin, on parallélisera un code plus conséquent pour étudier la faisabilité en situation réelle d'un tel portage. On conclura qu'à condition d'apporter une attention particulière à la localisation des données, le gain de performances est considérable.

4 Mots clefs

Parallélisation, *GPU*, *CPU*, transferts de données, performances, niveaux de parallélisation, kernels, asynchronisme, FLOPS.

Glossaire

CPU Un processeur (ou unité centrale de traitement, *UCT* ou en anglais central processing unit, *CPU*) est un composant présent dans de nombreux dispositifs électroniques qui exécute les instructions machine des programmes informatiques.[1]. 6, 9, 10, 12–14, 16, 29, 30, 34–37

CUDA (initialement l'acronyme de Compute Unified Device Architecture) est une technologie de GPGPU (General-Purpose Computing on Graphics Processing Units), c'est-à-dire utilisant un processeur graphique pour exécuter des calculs généraux à la place du processeur. Ce langage de programmation permet de programmer des *GPU* en *C*.[2]. 9, 11, 13, 30, 31

cœur Un cœur physique est un ensemble de circuits capables d'exécuter des programmes de façon autonome. Toutes les fonctionnalités nécessaires à l'exécution d'un programme sont présentes dans ces cœurs.[3]. 9, 11, 13, 30, 35

FLOPS Le nombre d'opérations en virgule flottante par seconde (en anglais : floating-point operations per second ou *FLOPS*) est une unité de mesure de la performance d'un système informatique. Les opérations en virgule flottante (additions ou multiplications) incluent toutes les opérations qui impliquent des nombres réels.[4]. 6, 14, 16, 19, 20, 22, 23, 25, 26

GPU Un processeur graphique, ou *GPU* (de l'anglais Graphics Processing Unit), est un circuit intégré présent la plupart du temps sur une carte graphique et assurant les fonctions de calcul de l'affichage. Un processeur graphique a généralement une structure hautement parallèle qui le rend efficace pour une large palette de tâches graphiques.[5]. 2, 6, 7, 9–17, 22, 29–31, 33–38

parallélisation Le parallélisme consiste à mettre en œuvre des architectures d'électronique numérique permettant de traiter des informations de manière simultanée, ainsi que les algorithmes spécialisés pour celles-ci.[6]. 3, 6, 9, 10, 12, 13, 25, 27, 29, 30, 33–35, 38, 39

PCI L'interface *PCI* (de l'anglais Peripheral Component Interconnect) est un standard de bus local (interne) permettant

de connecter des cartes d'extension sur la carte mère d'un ordinateur.[7]. 10

5 Motivations et contexte

5.1 Enjeux du calcul sur *GPU*

L'objectif du High Performance Computing (*HPC*) est d'utiliser au mieux des ressources de calcul pour des tâches avec un grand nombre d'opérations. C'est l'un des sujets sur lesquels travaille l'équipe de recherche *POEMS*, collaboration entre l'*UMA* (*Unité de Mathématiques Appliquées*) de l'*ENSTA* et l'*INRIA*, au sein de laquelle j'ai effectué mon stage. Les optimisations de codes sur processeurs, avec exploitation du multithreading et des emplacements de données en mémoire sont un objet d'étude depuis les années 1980. Or, depuis les années 1990, le développement des environnements graphiques a amené progressivement à adjoindre des unités de calcul aux processeurs, dédiés aux calculs de rendu et d'affichage graphique. On a alors distingué le Central Processing Unit (*CPU*) du Graphical Processing Unit (*GPU*). Ces derniers effectuent des calculs bien plus lentement (les unité de calculs indépendantes d'un *GPU* sont cadencées généralement autour de 1 GHz, contre 3 à 4 GHz pour les *CPU* de haute gamme), mais avec beaucoup plus de parallélisme (Un *GPU* dispose de plusieurs milliers d'unités de calcul indépendantes, là où un processeur classique ne dépassera que rarement les 8 cœurs). Aussi, dès la fin des années 2000, les constructeurs de *GPU* ont commencé à proposer des moyens de les exploiter pour le *HPC*, comme le langage *CUDA*. Or si la parallélisation sur le processeur ne consiste "que" en du multithreading, et en l'exploitation maximale de la vectorisation, le portage sur *GPU* offre plusieurs niveaux de parallélisation du travail, et son absence d'autonomie demande de prêter une attention particulière aux transferts de mémoire, très coûteux en temps, ainsi qu'à la planification et la répartition des tâches.

5.2 Architecture d'un *GPU*

Conçus notamment pour calculer les géométries complexes formées par les millions de tétraèdres des rendus 3D, les *GPU* sont constitués de milliers d'unités de calcul indépendantes (environ 5000 sur un *GPU Tesla K80*), et hiérarchisés en trois échelles de groupements que l'on détaillera plus loin, et qui permettent un parallélisme bien plus élevé que sur la petite dizaine de cœurs qui sont la norme sur les

CPU actuels. Les *GPU* sont des éléments physiquement distincts des *CPU*. Ils ne partagent donc pas de mémoire, et communiquent par les ports internes des ordinateurs, c'est-à-dire généralement des ports *PCI*, de plus en plus *PCI-Express*, dont le débit maximal est autour de 16 GB/s. D'une part c'est un débit significativement plus faible que ceux que l'on relève entre le processeur et la mémoire vive qui sont autour de 20 GB/s en milieu de gamme, mais surtout, le *GPU* ne peut pas accéder de manière autonome à des données en *RAM*, ce qui oblige le *CPU* à copier les données nécessaire aux calculs dans la mémoire locale du *GPU*. Il est donc important d'adresser ces problèmes de localité des données lors du portage d'un code sur *GPU*.

5.3 Modèle de programmation d'un *GPU*

Sur un *GPU*, comme mentionné précédemment, il y a plus de niveaux de parallélisation possibles. En effet, il est constitué d'un très grand nombre d'unités de calcul indépendantes. Ces unités sont réparties en `gangs` indépendants, eux même divisés en `workers`, que l'on peut organiser au niveau logiciel en `vectors`, bien qu'ils ne possèdent a priori pas d'instructions de vectorisation à proprement parler, ou peu. On peut par exemple supposer que chaque niveau de parallélisation correspond essentiellement à de la mémoire partagée. Tout le problème de notre parallélisation va alors être de répartir correctement les différents niveaux de parallélisation au sein de boucles imbriquées, et de bien choisir le nombre de `gangs` et de `workers`, et la taille des `vectors`.

Lors de l'exécution d'un programme sur *GPU*, les ensembles de calculs successifs sont rassemblés en un noyau de calcul, que l'on appelle un *kernel*. Pour chaque opération, un *kernel* doit être mis en place par le *CPU*, c'est-à-dire qu'il détermine la répartition des calculs sur l'*accélérateur* pour une "itération". Une fois que cette préparation est faite, les instructions sont envoyées au *GPU* qui ne fait qu'exécuter les calculs. L'*hôte* attend alors qu'un *kernel* soit terminé avant d'en mettre en place un nouveau. Il est donc nécessaire de réfléchir à la planification des calculs en amont afin de ne pas perdre du temps dans de la synchronisation inutile, ou dans la préparation de kernels superflus.

5.4 Objet de l'étude

L'objectif de cette étude va être d'étudier les possibilités de portage de codes C existants sur *GPU* à l'aide du standard *OpenACC*, d'étudier la facilité d'un tel portage, et de s'intéresser aux gains de performance possibles, via une comparaison notamment avec des codes optimisés via *OpenMP*. Nous commencerons par présenter l'outil dont nous allons nous servir durant le stage : *OpenACC*, puis nous en étudierons en détail les performances que l'on peut obtenir sur des exemples académiques classiques, pour enfin se pencher sur un code plus conséquent de résolution d'équations différentielles sur des éléments finis par un solveur de Jacobi.

6 L'outil *OpenACC*

6.1 Philosophie OpenACC

OpenACC est un standard d'écriture de code *C*, *C++* et *Fortran*, ayant pour objectif d'exploiter la puissance de calcul des cartes graphiques. Contrairement à *Cuda* qui est un langage à part entière ayant la même visée, *OpenACC* n'est qu'un ensemble de directives à insérer dans un code pré-existant pour l'optimiser, assorti de fonctions et de variables d'environnement pour en affiner le contrôle, à la manière de *OpenMP*. Cependant, contrairement à *OpenMP*, l'efficacité des directives *OpenACC* dépendra largement du matériel, notamment graphique, de l'utilisateur. Les directives devront donc être pensées en fonction, et la compilation devra a priori se faire sur la machine sur laquelle le code sera utilisé. Ceci s'explique par le fait qu'*OpenMP* ne donne des directives concernant que des opérations exécutées par le processeur, et qui s'adapteront automatiquement aux spécifications du processeur, c'est-à-dire au nombre de cœurs, et aux instructions de vectorisation disponibles au moment de la compilation. En revanche, pour *OpenACC*, beaucoup d'autres facteurs entrent en ligne de compte, notamment le temps de communication entre le processeur et la carte graphique, qui peut faire beaucoup évoluer le temps d'exécution.

De plus, il serait illusoire d'imaginer pouvoir obtenir des performances égales à celles de langages spécifiques, comme *CUDA* avec *OpenACC*. Elles seront en général jusqu'à 20% moins bonnes [8]. Cependant, cette perte de performance se fait au profit d'une part

d'une simplicité d'écriture, et d'autre part d'une portabilité du code importante. En effet, d'une machine à l'autre, recompiler le code suffira à obtenir des performances intéressantes, et les modifications à faire pour tirer le maximum d'*OpenACC* seront minimes.

Toutes les directives seront écrites de la façon suivante :

```
#pragma acc directive argument(parametres)
```

6.2 Compilation

Non seulement la compilation ne pourra se faire réellement efficacement que sur la machine qui fera tourner le code, mais en plus tous les compilateurs ne prennent pas encore en compte, du moins pas totalement, toutes les spécifications et directives du standard *OpenACC*. Si *GCC*, le compilateur du projet *GNU*, affirme le faire en théorie, le développement n'est pas encore totalement abouti et on ne l'utilisera qu'à titre de comparaison. On choisira plutôt de se concentrer sur *PGI*, le compilateur développé par les équipes *Nvidia* pour du matériel *Nvidia*, car c'est la configuration implémentant le mieux les spécifications de *OpenACC*.

6.3 Profiling

Avant de commencer à baliser son code avec des directives OpenACC, il est préférable de le tester sur de petits exemples, et de l'exécuter via un outil que l'on appelle un *profiler*, qui va étudier les accès mémoire et les temps d'exécution des différentes sections de code, dans le but de déterminer les sections qui ralentissent l'exécution du programme, afin de déterminer si leure exécution sur le matériel graphique de la machine serait intéressant ou non. En effet, l'envoi de données du *CPU* vers le *GPU* est coûteux temporellement, et n'est donc pas toujours justifié. On utilisera pour cette étude *PGPROF*, le *profiler* du compilateur *PGI*, et *NVVP* (*Nvidia Visual Profiler*), qui sont sensiblement identiques. Le profiling est une étape importante de l'optimisation sous *OpenACC* car l'impact des temps de transfert de données est important et n'est pas forcément appréhendable sans des outils spécialisés tels que les *profiler*s. Il est donc nécessaire durant les étapes de parallélisation sous *OpenACC* d'étudier régulièrment le profil d'un code pour comprendre les étapes clefs des temps d'exécution.

En pratique, il n'aura pas été possible d'utiliser le profiler en graphique, seulement en ligne de commande, ce qui complique l'exploitation des données extraites, et demande une bonne connaissance de *CUDA*. C'est pourquoi les études de perfomances faites dans la suite n'auront pu être qu'empiriques et non appuyées sur les données du profiler.

6.4 Portabilité

On se concentrera dans cette étude sur l'exploitation du standard *OpenACC* pour le calcul sur *GPU*. Cependant, toutes les directives que l'on va étudier dans la suite peuvent être utilisées pour du calcul sur *CPU* (par exemple si la machine possède plusieurs *CPU*). En effet, *OpenACC* ne fait une distinction qu'entre l'*hôte*, qui est le composant sur lequel le programme est exécuté (En général un cœur du *CPU*), et les *accélérateurs*, qui sont tout les composants capable de calculer en parallèle de l'*hôte* (c'est-à-dire les autres cœurs du *CPU*, les *CPU* supplémentaires s'il y en a plus d'un sur la machine, ou encore les GPU). On peut préciser l'*accélérateur* sur lequel on veut faire tourner notre code (le *targeted accelerator*) à la compilation via le flag `-ta:`, suivi par exemple de l'argument `tesla` pour une carte graphique Nvidia Tesla, ou `multicore` pour un *CPU* à plusieurs cœurs.

6.5 Etapes de l'optimisation d'un code

L'optimisation d'un code via *OpenACC* passe généralement par trois étapes :

- La parallélisation, durant laquelle on va chercher à faire exécuter un maximum de boucles sur les différents accélérateurs, quitte à observer une perte de performances. L'important est de vérifier via un profiler que les calculs au sein de chaque itération ont bien connu un gain dans la vitesse d'exécution. En effet, c'est en général les transferts de données qui sont chronophages et dont on va s'occuper dans un second temps.
- L'*offloading*, c'est-à-dire s'assurer que les transferts de données sont effectués sans superflu, en limitant au maximum les changements de mémoire. C'est en général le point faible des compilateurs, il est donc important d'y accorder un soin particulier. Le flag `-Minfo=all` permet de vérifier les opérations en

mémoire, et le profiling permet d'étudier les transferts plus en détail. Le flag `-ta:tesla,time` permet quant à lui d'étudier de manière rudimentaire les temps passés à effectuer des calculs et des déplacements mémoire.

— L'affinage du code propre à la machine sur laquelle il est exécuté. C'est à cette étape-là que l'on va pouvoir prendre en compte la configuration de la machine sur laquelle est exécuté le code, en se penchant plus avant notamment sur le nombre de `gang`, de `workers`, et la taille des vecteurs.

Il est à noter que cette méthodologie n'est qu'une ligne directrice et n'est en aucun cas impérative, mais elle a cependant fait ses preuves.

7 Etude de cas académiques avec *OpenACC*

Pour tester les performances d'*OpenACC*, on s'est penché sur trois exemples de codes classiques : *DGEMM*, le calcul d'intégrales par l'approximation des trapèzes, et le calcul des différences finies.

Ces codes ont été compilés et exécutés sur une machine équipée d'un *GPU Nvidia Tesla K80*. Les temps d'exécution incluent les temps de transfert de mémoire, et interviennent donc dans les calculs de *FLOPS*, ce qui n'est probablement pas le cas dans les calculs de *FLOPS* faits par Nvidia pour estimer les performances de leurs cartes graphiques.

Les codes *OpenMP* sont compilés et exécutés sur une machine équipée d'un *CPU Intel XEON E5*, avec le *Intel C++ Compiler*.

7.1 *DGEMM*

7.1.1 Présentation de la routine

La librairie *Basic Linear Algebra Subprograms* est un ensemble de routines parmi les plus couramment utilisées en *HPC*, la routine la plus emblématique étant *DGEMM* (la multiplication matricielle généralisée), soit l'opération $\alpha \times AB + \beta \times C$, avec A, B et C des matrices, et α et β des réels. Elle est particulièrement intéressante du fait de son utilisation fréquente pour mesurer les *FLOPS* d'une machine, c'est pourquoi on y a apporté un travail plus approfondi que sur les codes suivants. On a recodé la routine *DGEMM* de la

façon suivante (en supposant que la matrice B ait été transposée au préalable) :

7.1.2 Code

```
1  #pragma acc data pcopyin(A[0:A_l * A_c], B[0:B_l * B_c]) pcopy
       (C[0:C_l * C_c])
2  {
3  #pragma acc parallel loop gang num_workers(32) vector_length
       (32)
4   for(int i = 0; i < C_l; i++){
5  #pragma acc loop worker
6     for(int j = 0; j < C_c; j++){
7       C[i * C_c + j] *= beta;
8       float c = 0;
9  #pragma acc loop vector reduction(+:c)
10      for(int k = 0; k < A_c; k++){
11        c += alpha * A[i * A_c + k] * B[j * B_l + k];
12      }
13      C[i * C_c + j] += c;
14    }}}
```

Listing 1 – Implémentation de *DGEMM* avec *OpenACC*

7.1.3 Détail des clauses

- `# pragma acc data` : on commence par créer une zone de données englobant toute notre fonction, qui indique les déplacements dans la mémoire souhaitée et à l'intérieur de laquelle aucune autre donnée ne sera transférée entre l'*hôte* et l'accélérateur.
- `pcopyin(A[:], B[:]) pcopy(C[:]) pcreate(B_t[:])` : on transfère les matrices A et B et C dans la mémoire du *GPU* au début du calcul, en précisant que l'on souhaite récupérer la matrice C en fin de calcul.
- `# pragma acc parallel loop gang` : sur les deux boucles principales de la fonction, on déclare une zone parallèle, et on ajoute l'argument `loop` qui signale au compilateur que cela concerne la boucle qui suit directement. L'argument `gang` indique à quelle échelle on souhaite paralléliser la boucle.
- `# pragma acc loop worker` : sur les boucles intermédiaires,

on parallélise à l'échelle des `workers`. Sur la première boucle, on pourrait vouloir utiliser plutôt la clause `vector`, mais en étudiant les indices, on se rend compte que les calculs ne sont pas vectorisables.

— `# pragma acc loop vector reduction(+:c)` : sur cette boucle la plus intérieure, les calculs sont vectorisables, et c'est ce que l'on spécifie au compilateur. De plus, on souhaite calculer la somme des résultats des calculs effectués, on déclare donc une réduction.

— `num_gangs(256) num_workers(32) vector_length(32)` : on déclare ici, au niveau de la directive `parallel`, le nombre de `gangs`, de `workers` et la taille des `vectors` que l'on souhaite à l'intérieur de la zone. Ces valeurs ont été déterminées empiriquement et peuvent varier d'un code à l'autre.

Le code finalement obtenu permet d'atteindre un nombre de *FLOPS* théoriques de l'ordre de 10^{11}, soit environ 10% du nombre de *FLOPS* annoncé par le constructeur, et ce en comptant les temps de transfert mémoire.

7.1.4 Etude du paramétrage

Pour déterminer les nombres de `gangs`, `workers` et les tailles de `vecteurs`, on procède à une étude avec différentes valeurs. On commence par fixer le nombre de `gangs` à 512, puis on effectue des tests sur différents nombres de `workers` et différentes tailles de `vecteurs`. On remarque tout d'abord que pour un même nombre de `gangs`, le produit du nombre de `workers` et de la taille des `vecteurs` est plafonné. C'est dû à la structure du *GPU* et à son fonctionnement interne. En effet, les opérations de vectorisation ne sont a priori pas effectuées de la même manière que sur un *CPU*, c'est-à-dire pas par une unique unité de calcul, mais par plusieurs simultanées. Au sein d'un gang, ces unités de calcul sont réparties entre les `workers` qui peuvent alors utiliser les unités de calcul à disposition pour faire des simili calculs vectoriels. La taille des `vecteurs`, en bits, est minorée par 32, soit la taille d'un float. Le maximum de `workers` mobilisable au sein d'un gang avec des `vecteurs` de taille 32, est 32. On en déduit donc qu'un gang contient 32 unités de calcul indépendantes, capable chacune d'effectuer des calculs sur un float, soit 32 bits. On peut

alors faire varier la taille des `vecteurs`, en faisant varier le nombre de `workers` en conséquence. On parlera de couple "nombre de `workers`, taille de `vecteurs`" (W/V). Pour chaque taille de vecteur, on essaiera tous les nombres de `workers` possibles à partir de 2. Cependant, s'il est compréhensible que l'on ne puisse excéder un certain nombre de `workers` et une certaine taille de vecteur, du fait du lien entre les deux et des limitations physique du *GPU*, certaines combinaisons ne semblent pas fonctionner. C'est le cas pour 16W/64V, mais aussi pour 4W/64V, alors que toutes les combinaisons "adjacentes" fonctionnent (i.e : 4W/32V, 4W/128V, 2W/64V, 8W/64V). On compare donc les différentes performances sur des matrices carrées de tailles variant de 1000 à 10000. Pour chaque couple W/V, on fait le rapport avec les temps de référence, que l'on prend pour le programme compilé sans imposer de nombre de `workers` ni de taille de vecteur, et on fait la moyenne de ces rapports pour les différentes tailles de matrices. On obtient alors la figure suivante :

W/V	32	64	128	256	512
2	42.37	47.97	62.08	111.6	200.4
4	37.38	x	66.29	120.8	
8	31.87	44.97	73.95		
16	32.37	x			
32	33.99				

TABLE 1 – Mesure des temps pour une matrice carrée de taille 1000, en millisecondes (x = Erreur d'origine inconnue lors de l'exécution)

W/V	32	64	128	256	512
2	38.09	38.08	38.91	37.95	41.65
4	33.00	x	32.20	32.71	
8	27.99	30.40	29.63		
16	26.35	x			
32	24.18				

TABLE 2 – Mesure des temps pour une matrice carrée de taille 10000, en secondes (x = Erreur d'origine inconnue lors de l'exécution)

On constate ici que la taille des vecteurs est un facteur de ralentissement, alors que le nombre de workers est un facteur d'accélération significatif. Au point qu'il est plus intéressant, si possible, de diminuer la taille des vecteurs plutôt que d'augmenter le nombre de workers (c.f : comparaison 2W/256V, 4W/256V, 2W/128V). On conclut donc

que le plus intéressant est de minimiser la taille des vecteurs, et de maximiser le nombre de workers, le couple 32W/32V étant alors la meilleure combinaison possible pour les situations étudiées ici, notamment dans le cas de grandes matrices. On peut alors s'interroger sur la pertinence de la vectorisation sur la boucle intérieure, puisqu'elle semble être contre-productive. En revanche, la répartition sur les workers permet des performances significatives. On peut alors se demander si l'on ne pourrait pas retirer la vectorisation de la boucle intérieure.

Pour ce qui est du nombre de gangs, qui semble totalement décorrélé du couple W/V, et avec un plafond beaucoup plus élevé, on se propose de tester toutes les puissances de 2, de 2 à 65536, avec un couple 32W/32V fixé.

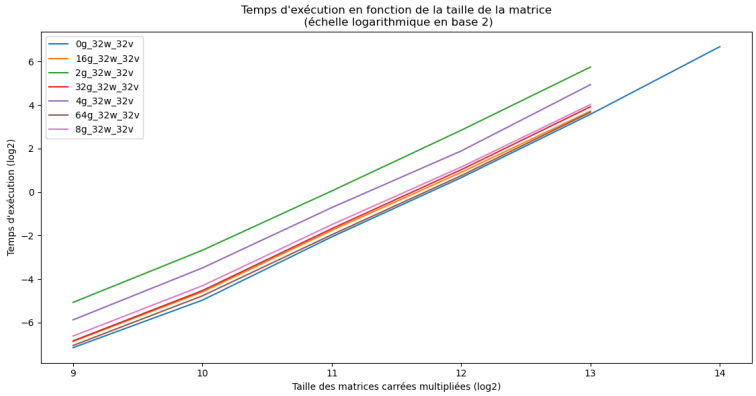


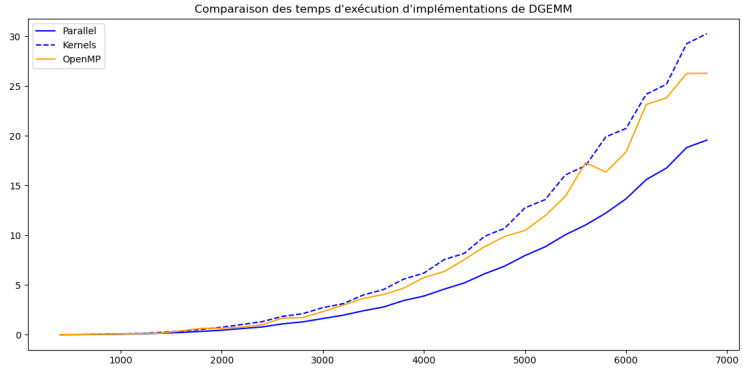
FIGURE 1 – Evolution des temps de calcul en fonction du nombre de gangs

On constate ici que l'augmentation des gangs produit des résultats significatifs entre 2 et 128 gangs, et qu'au-delà il est plus difficile de quantifier les augmentations de performances, les courbes étant quasiment confondues (elles ont été retirées par souci de lisibilité). On note cependant que les temps de référence, c'est-à-dire les temps mesurés sans précision de nombre de gangs à la compilation, sont du même ordre que ceux mesurés pour le maximum de gangs spécifiés à la compilation.

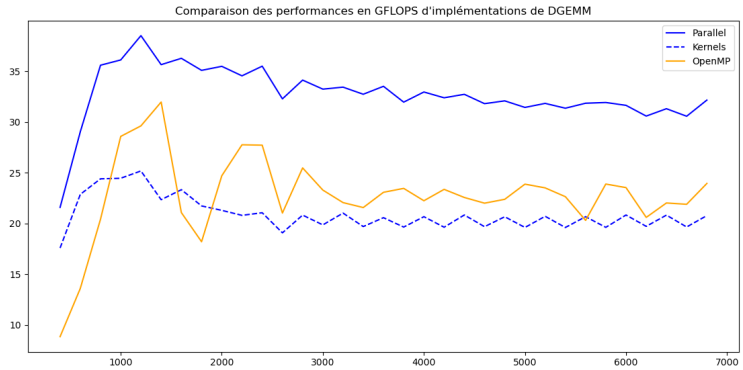
7.1.5 Mesures de performances

On calcule de plus le nombre de FLOPS théoriques avec la formule suivante :

$$\text{flops} = \frac{(\text{taille des matrices})^2 \times (2 \times \text{taille des matrices} + 3)}{\text{temps d'exécution}}$$



(a) Comparaison des temps d'exécution en fonction de la taille des matrices carrées

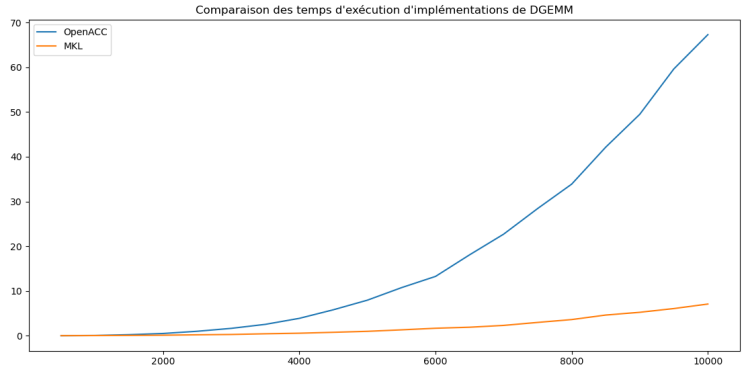


(b) Comparaison des GFLOPS en fonction de la taille des matrices carrées

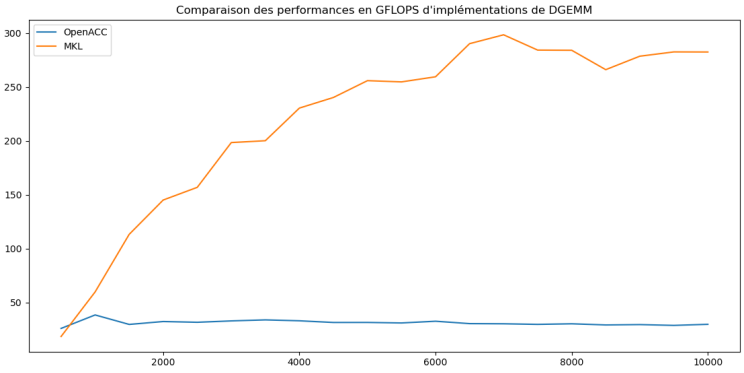
FIGURE 2 – Etude de la multiplication généralisée de matrices carrées

On compare alors en figure 2 les résultats obtenus pour le codes compilé avec la directive `parallel` et les paramètres étudiés ci-dessus, pour le code compilé avec seulement la directive `kernels`, et pour le

code compilé avec des directives *OpenMP*. On observe sur la figure 2b que la directive `kernels` de *OpenACC* suffit à obtenir des résultats comparables à ceux de *OpenMP*, bien qu'un peu moins bons. En revanche, la directive `parallel` et une étude approfondie du nombre de gangs, de workers, et de la taille des vecteurs permettent une augmentation de performances de près de 40%.



(a) Comparaison des temps d'exécution en fonction de la taille des matrices carrées



(b) Comparaison des GFLOPS en fonction de la taille des matrices carrées

FIGURE 3 – Etude de la multiplication généralisée de matrices carrées avec une implémentation optimisée (*MKL*)

On peut aussi comparer en figure 3 l'implémentation *OpenACC* avec la directive `parallel` à un *DGEMM* classique comme celui développé par *Intel*, contenu dans la librairie *MKL*. On constate ce qui était attendu : les performances de *MKL* sont bien supérieures

à celles obtenues par *OpenACC*. On remarque même que malgré l'augmentation de la taille des matrices, les performances de ce dernier n'augmentent pas, contrairement à ce que l'on aurait pu penser.

7.2 Intégration par la méthode des trapèzes

7.2.1 Présentation

L'intégration d'une fonction par la méthode des trapèzes consiste à découper l'aire sous la courbe étudiée en trapèzes successifs dont on sait aisément calculer puis sommer l'aire pour approximer l'intégrale recherchée.

7.2.2 Code

```
1  double approx = 0.5 * (f(a) + f(b));
2  double h = (b - a) / N;
3  #pragma acc parallel loop reduction(+:approx)
4  for(int i = 0; i < N; i++){
5    approx += f(a + i * h);
6  }
7  approx *= h;
8
9  #pragma acc routine
10 double f(double x){
11   return 1/x;
12 }
```

Listing 2 – Implémentation de l'intégration par la méthode des trapèzes *OpenACC*

7.2.3 Détail des clauses

- `#pragma acc parallel loop` On indique que la boucle qui suit est parallélisable.
- `#reduction(+:approx)` On effectue ici une réduction sur la variable `approx` qui est la somme des aires de tous les trapèzes calculés.
- `#pragma acc routine` On déclare ici que la fonction qui suit va être appelée et peut être parallélisée, voire vectorisée lors de ces appels.

7.2.4 Mesure des performances

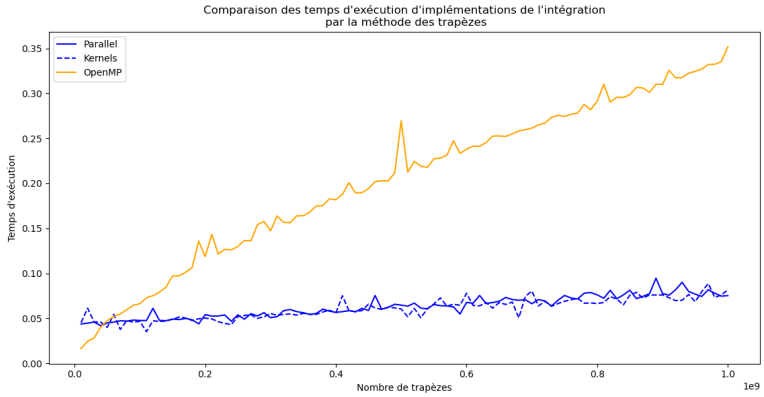
Pour le calcul de l'aire de chaque trapèze, on considèrera que l'appel à une fonction extérieure ne correspond qu'à une opération à virgule flottante, soit 4 opérations par itération en comptant la multiplication par h et le calcul de $a + i * h$. On a donc la formule suivante :

$$\text{flops} = 4 \times \frac{\text{nombre de trapèzes}}{\text{temps d'exécution}}$$

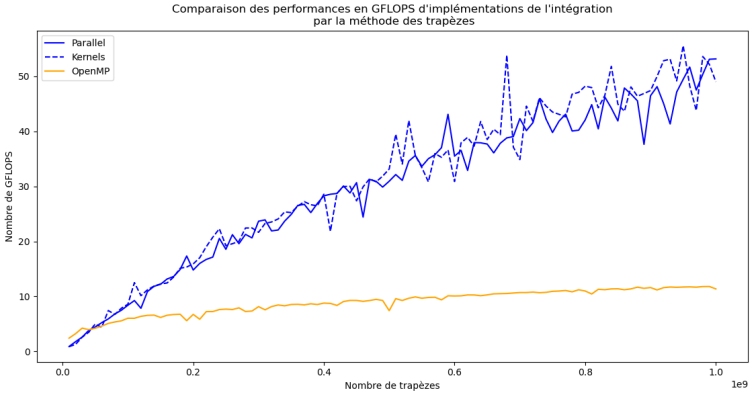
On compare en figure 4 le calcul d'intégrales par la méthode des trapèzes implémenté avec *OpenACC* en utilisant les directives `parallel` et `kernels`, et avec *OpenMP*. On constate sur la figure 4a que là où le temps d'exécution de l'implémentation *OpenMP* augmente linéairement, le temps d'exécution de l'implémentation *OpenACC* est quasiment constant, que l'on utilise la directive `parallel` ou `kernels`, ce qui correspond à l'augmentation linéaire du nombre de *GFLOPS* en figure 4b, qui finit par dépasser celui de l'implémentation *OpenMP*. Le *GPU* n'étant, comme le montre la figure 4a, pas poussé à son maximum, le nombre maximum de *GFLOPS* n'est pas représentatif de ses capacités. Les temps d'exécution sont trop courts pour faire une différence entre les deux implémentations *OpenACC*, mais les calculs sont ici limités par les contraintes liées à la taille des variables. En effet, un `int` ne peut stocker des entiers au-dessus de 4×10^9, soit l'ordre auquel on s'est arrêté dans les mesures de performances, or les temps d'exécution restent en-dessous de la demie-seconde. C'est cependant suffisant pour déterminer que le *GPU* permet ici aussi de meilleures performances lorsque le nombre de trapèzes devient conséquent.

Pour obtenir une comparaison des temps d'exécution plus intéressante, il pourrait par exemple être judicieux d'étudier une fonctions plus complexe, comme un développement limité de fonction trigonométriques.

Il est important de noter que l'augmentation du nombre de trapèzes ici ne signifie pas nécessairement une augmentation de la précision du résultat, on constate en effet plutôt un cumul d'erreurs



(a) Comparaison des temps d'exécution en fonction du nombre de trapèzes



(b) Comparaison des GFLOPS en fonction du nombre de trapèzes

FIGURE 4 – Etude du calcul de $\int_1^{10^6} \frac{1}{x} dx$ par la méthode des trapèzes

d'arrondis qui diminuent cette précision. Cependant on s'intéresse ici aux temps de calculs plus qu'au résultat.

7.3 Différences finies

7.3.1 Présentation

Le calcul de différences finies est une méthode de recherche de solutions approchées à des équations aux dérivées partielles, qui consiste en une discrétisation du problème par un schéma numérique sur une grille structurée. On s'intéresse ici à l'exemple classique de l'équation de la chaleur.

7.3.2 Code

```
1  double dt = 1./(double)(T-1);
2  double dx = 1./(double)(N-1);
3  #pragma acc data pcopy(C[0:N*N]) pcreate(Cnew[0:N*N])
4  {
5  for(int n = 0; n < T; n++){
6  #pragma acc parallel loop gang num_gangs(4096) vector_length
       (512)
7    for(int i = 1; i < (N-1); i++){
8  #pragma acc loop vector
9      for(int j = 1; j < (N-1); j++){
10       Cnew[N * i + j] = (1 - 4 * dt / (dx * dx)) * C[N * i + j
             ] + dt / (dx * dx) * (C[N * (i + 1) + j] + C[N * (i
             - 1) + j] + C[N * i + (j + 1)] + C[N * i + (j - 1)])
             ;
11     }
12   }
13   double* temp = Cnew;
14   Cnew = C;
15   C = temp;
16 }}
```

Listing 3 – Implémentation des différences finies

7.3.3 Détail des clauses

— `#pragma acc data` On déclare ici les déplacements de données que l'on souhaite effectuer, et l'on encadre par des accolades la zone concernée.

— `pcopyin(C[:]) pcopyout(Cnew[:])` On signale au compilateur que l'on souhaite copier en entrée le tableau `C`, et que l'on ne souhaite pas le récupérer à la fin. En revanche, on déclare

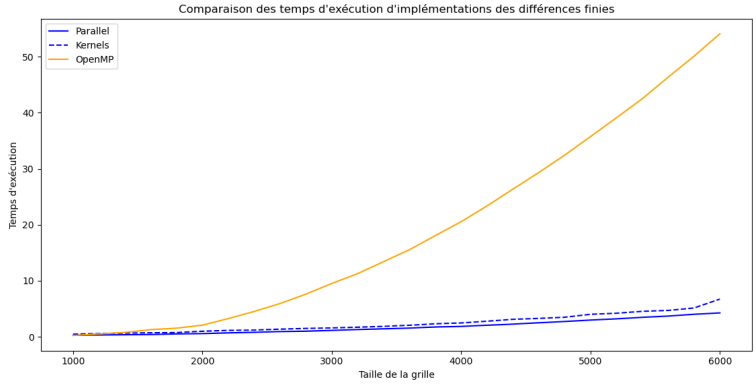
vouloir récupérer les données calculées pour `Cnew`, mais sans prendre en compte les données qu'il contient déjà.
— Les directives `parallel`, `loop`, `gang`, `worker` et `vector`, ont les mêmes rôles de balisage et structuration de la parallélisationdu code que précédemment pour *DGEMM*.

7.3.4 Mesure des performances

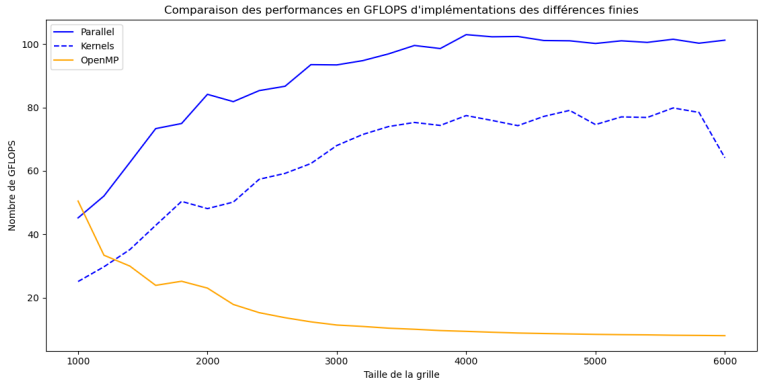
Pour chaque case, on effectue 12 opérations sur des flottants, on calcule donc les FLOPS théoriques par la formule suivante :

$$\text{flops} = 12 \times \frac{\text{itérations} \times (\text{taille de la grille})^2}{\text{temps d'exécution}}$$

On compare en figure 5 le calcul des différences finies implémenté avec *OpenACC* en utilisant les directives `parallel` et `kernels`, et avec *OpenMP*. On choisit ici de se limiter à 1000 itérations temporelles, cependant il est à noter que les itérations temporelles ne pouvant être parallélisées, chacune demande la création d'un nouveau *kernel*, ce qui est coûteux en temps. Sur de petites grilles et un grand nombre d'itérations, l'implémentation *OpenMP* donnera donc de meilleurs résultats, comme le montre la figure 5b. En revanche, pour peu d'itérations mais une grande grille, l'implémentation *OpenACC* sera plus efficace. On remarque de plus que la directive `kernels` donne des résultats intéressants par rapport à *OpenMP*, mais qu'il est possible d'obtenir des performances autour de 25% meilleures avec la directive `parallel` et une étude rapide des performances en fonction du nombre de gangs et de la taille des vecteurs. On remarque de plus que l'on arrive à atteindre les 100 *GFLOPS*, soit 5% de la puissance nominale de la *Tesla K80* en double précision, et ce avec très peu de directives.



(a) Comparaison des temps d'exécution en fonction de la taille de la grille



(b) Comparaison des GFLOPS en fonction de la taille de la grille

FIGURE 5 – Etude du calcul de différences finies pour les équations de la chaleur en deux dimensions pour 1000 itérations temporelles

8 Le cas des éléments finis

8.1 Exposé du problème

On choisit ici d'utiliser *OpenACC* sur un code plus conséquent, en l'occurence sur une implémentation de la méthode des éléments finis avec solveur de Jacobi. On se penchera plus particulièrement sur ce dernier.

Un solveur de Jacobi calcule itérativement la solution d'une équation de la forme $A.x = b$, en construisant la suite définie par la

relation suivante : $u_{k+1} = (b - N.u_k).D^{-1}$, avec D diagonale, N de diagonale nulle telle que $A = D + N$.

La matrice N étant essentiellement constituée de zéros (*matrice creuse*), la question du stockage de données s'est posée. En effet, un tableau complet plein de zéros est une grosse perte en termes de mémoire. On a donc commencé à la stocker sous la forme de trois listes, une contenant les valeurs non nulles, les deux autres contenant les colonnes et les lignes correspondant aux valeurs (format *Coordinate list*). Cependant, ce format se prête mal à la parallélisation car le parcours simultané des valeurs peut entraîner du *data race*. On a alors choisi de changer de format, en y préférant le format de *Yale*, où le tableau de valeurs est ordonné par l'indice de la ligne des valeurs, la première liste contenant à l'indice n l'emplacement, dans la liste des valeurs, où commencent les valeurs de la n^e ligne, et la deuxième liste contenant la colonne correspondant à chaque valeur. Ce format permet un parcours de la matrice par lignes sans risquer de *data race*.

8.2 Abord du code

La première chose que l'on remarque est que les itérations successives du solveur ne pourront pas êtres exécutées en parallèle, et que l'on pourra alors faire appel à la directive `async` qui permettra de ne pas perdre de temps lors de la préparation des *kernels*.

Dans une première tentative de mettre en place *OpenACC*, après la création de la clause `data` autour de la boucle principale du solveur, on se rend compte que sur un code important, la gestion des données est d'autant plus délicate qu'il y a beaucoup de variables, et qu'au sein d'une clause `data`, il est important de se demander, pour chaque opération, quelle est la variable sur laquelle on est en train de travailler, c'est-à-dire la variable en mémoire de l'*hôte*, ou la copie (implicite ou non) dans la mémoire de l'*accélérateur*.

De plus, s'il est recommandé d'effectuer d'abord la parallélisation, puis de s'occuper des transferts de mémoire, il s'avère, d'expérience, qu'il peut aussi être une bonne pratique de faire l'inverse, surtout lorsque la zone à paralléliser contient de nombreux *kernels*. En effet, la construction progressive de ceux-ci permet de vérifier empiriquement que l'on effectue bien les calculs voulus sur les bonnes données, et que l'on n'a fait aucune omission dans les allocations et copies mémoires.

De plus, avec les spécifications 2.7 d'*OpenACC*, les *deep copy* ne sont pas encore implémentées, et les structures ne sont pas correctement copiées sur l'*accélérateur*. On a donc renommé tous les pointeurs vers des tableaux dans des structures pour éviter les erreurs de segmentation.

8.3 Codes

```
1  #pragma acc data copyin(N2_val[:N_nnz], diagA_val[:numDof],
       b_val[:numDof], N2_idzR[:numDof + 1], N2_idC[:N_nnz]) copy
       (u_val[:numDof]) create(Nu_val[:numDof])
2  {
3    while (it < maxit && residual > tol){
4  #pragma acc parallel loop async
5      for(int i=0; i<numDof; ++i){
6        Nu_val[i] = 0.;
7      }
8  #pragma acc parallel loop independent async
9      for(int i = 0; i < numDof; i++){
10       for(int j = N2_idzR[i]; j < N2_idzR[i + 1]; j ++){
11         Nu_val[i] += N2_val[j] * u_val[N2_idC[j]];
12       }
13     }
14     if((it % 1000) == 0){
15       residual = 0;
16 #pragma acc parallel loop copy(residual) reduction(+:residual)
       async
17       for(int n=0; n<numDof; n++){
18         double tmp = diagA_val[n] * u_val[n] - Nu_val[n] -
               b_val[n];
19         residual += tmp*tmp;
20       }
21 #pragma acc wait
22       residual = sqrt(residual);
23     }
24 #pragma acc parallel loop async
25     for(int i=0; i<numDof; ++i){
26       u_val[i] = (Nu_val[i] + b_val[i]) / diagA_val[i];
27     }
28     it++;
29   }
30 #pragma acc wait
31 }
```

Listing 4 – "Optimisation du solveur de Jacobi"

8.4 Détail des clauses

— `pragma acc data` Afin de limiter les transferts mémoire, on ne fait que copier les valeurs dont on a besoin dans la mémoire sans récupérer les éventuelles modifications à la fin de la zone via `copyin`. La seule valeur que l'on copie et que l'on récupère est la valeur de `u_val` avec `copy`. Enfin, la variable `Nu_val` ne contenant que des calculs temporaires, on ne fait que lui allouer un espace mémoire, sans copie de données.

— `pragma acc parallel loop async` A chacune des boucles, on se contente de signaler la parallélisation au compilateur, ce qui fait un nouveau *kernel* à chaque boucle. En effet, on est obligé d'effectuer ces *kernels* les uns à la suite des autres et non en concurrence pour éviter le *data race*. Cependant, le paramètre `async` permet de laisser le *CPU* continuer à travailler sans attendre la fin des *kernels*, et donc notamment de préparer les *kernels* suivants sans attendre et de gagner du temps.

— `pragma acc wait` Si le paramètre `async` permet à l'*hôte* de continuer ses calculs et donc de préparer les *kernels* suivants, il continue à compter les itérations sans attendre la fin des *kernels* de l'*accélérateur*. Pour éviter que l'on arrive au maximum d'itérations sans avoir laissé l'*accélérateur* terminer ses calculs, on ajoute cette directive pour que toutes les 1000 itérations, on resynchronise l'*hôte* et l'*accélérateur* après le calcul de la condition d'arrêt, avant de continuer les itérations si besoin. On ajoute aussi cette directive à la fin de la zone pour récupérer les valeurs exactes des variables, et éviter ainsi le *data race*.

On note que le code obtenu est de complexité linéaire par rapport au nombre de valeurs non nulles dans la matrice N et par rapport à la taille des matrices.

8.5 Mesure des performances

On obtient alors, après parallélisation, les résultats en figure 6. On constate que la parallélisation sur le *GPU* montre son efficacité lorsque le nombre d'éléments augmente, et ce d'autant plus que le

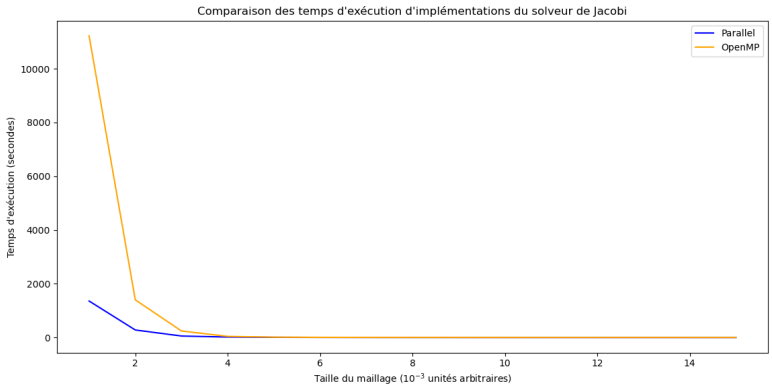


FIGURE 6 – Etude du temps d'exécution du solveur de Jacobi pour différentes tailles de maillage

code est de complexité linéaire. En effet, ce résultat est cohérent avec le fait que le cadençage du *GPU* soit plus faible que celui du *CPU*, mais avec un nombre d'unités de calcul indépendantes bien plus élevé. En effet, on voit que ces deux aspects se compensent sur un petit nombre d'éléments, mais que là où les itérations à effectuer sur chaque cœur du *CPU* augmentent avec le nombre d'éléments à traiter, le *GPU* répartit d'autant plus le calcul et démontre d'autant plus son efficacité, étant jusqu'à 10 fois plus performant.

9 Conclusion

9.1 Retour sur les résultats obtenus

On a constaté durant cette étude que le standard *OpenACC* permettait de porter facilement des codes sur *GPU*, via l'insertion de quelques directives de compilation dans le code.

Si l'on peut alors, sur des modifications comparables, obtenir des performances significativement meilleures que celles d'*OpenMP* du fait de la puissance du *GPU*, on reste cependant loin derrière les performances théoriques annoncées par le constructeur du *GPU*. Il est donc raisonnable de penser que dans le cadre de tests de parallélisation sur *GPU*, *OpenACC* puisse fournir une première approximation des possibilités, mais dans une perspective par exemple de simulation commerciale, une implémentation complète en *CUDA* donnera proba-

blement des résultats significativement meilleurs. On peut cependant envisager, du fait de la possibilité d'échanges de données entre des langages comme *CUDA* ou *OpenCL* avec *OpenACC*, non abordée ici, que les portions critiques d'un code soient implémentées sur *GPU* dans un langage spécialisé, et que les portions annexes soient, elles, portées sur *GPU* via *OpenACC* par souci de simplicité.

9.2 Compétences acquises

Ce stage m'a permis d'approfondir ma compréhension des *GPUs*, et même du fonctionnement d'un ordinateur dans son ensemble. J'ai pu m'initier au calcul distribué, ce qui est un atout majeur dans ma formation car applicable à un panel de domaines immense. Je pense, pour compléter ma formation, que je devrais m'initier à des langages spécialisés, comme *CUDA* qui m'aurait été utile durant ce stage. Je souhaiterais aussi, à l'avenir, me pencher plus en détail sur le calcul scientifique à hautes performances, car c'est là aussi une compétence aussi intéressante que polyvalente.

L'expérience de travail à distance fut très intéressante mais à double tranchant. En effet, si j'ai été beaucoup plus libre sur mes horaires et plus généralement mon cadre de travail, il était parfois difficile de se motiver à travailler, surtout lorsque ma progression semblait dans une impasse. Heureusement, j'ai été bien entouré, tant par mes encadrants de stage qui ont su me guider efficacement malgré la distance, que par les camarades avec qui j'ai passé mon stage et qui m'ont aidé à maintenir une ambiance de travail sérieuse et productive.

L'expérience de recherche fut enrichissante, car c'est un domaine qui m'était jusqu'alors inconnu. La première difficulté fut de se trouver confronté à l'étendue infinie des possibles, de laquelle il m'a fallu tirer une structure à mon projet de recherche. Heureusement mes encadrants ont su me proposer un schéma à la fois suffisamment précis pour que mon stage ait un sens et un but, mais aussi suffisamment dense pour que je puisse faire moi-même mes choix sur ce que je souhaitais étudier en détail, et sur la méthodologie à employer. J'ai dû ensuite me résoudre à la frustration de la recherche peu couronnée de réussite, ou alors de réussites bien plus modestes qu'attendues, et du sentiment de lenteur dans la progression qui en résulte. Enfin, il a fallu se faire à l'idée que je n'aurais ni le temps, ni la volonté

d'explorer à fond tout ce qu'il était possible d'explorer, ce qui ne fut pas facile à admettre. J'ai cependant apprécié la liberté que procure ce type de projet, la satisfaction d'une part d'avoir à exploiter les compétences acquises lors de mon cursus scolaire, et de me pousser intellectuellement, d'autre part de pouvoir choisir la méthodologie, les résultats à présenter et la façon de les obtenir.

Les annexes suivantes sont pour l'essentiel un résumé des directives, arguments et paramètres proposés par le standard *OpenACC* 3.0 [9].

A Directives de parallélisation *OpenACC*

A.1 La directive `parallel`

Cette directive sert à déclarer une zone du code comme parallèle, à la manière de *OpenMP*. Cependant, dans la mesure où l'on cherche ici à utiliser la puissance de calcul du *GPU*, il faut ajouter en argument les données que l'on a besoin de copier vers la mémoire du GPU, avec le paramètre `copyin()`, et de récupérer depuis la mémoire du *GPU* en fin de calcul, avec le paramètre `copyout()`, car ce transfert est coûteux et donc à limiter au maximum. On appelle cela l'*offloading*.

A.2 La directive `kernels`

Cette directive sert à déclarer au compilateur qu'une zone est optimisable, tout en lui laissant la liberté de l'optimiser. On peut ensuite ajouter d'autres directives pour affiner cette parallélisation automatique, mais globalement, la directive `parallel` impose au compilateur une région parallèle, et l'efficacité de la parallélisation repose uniquement sur le programmeur. A l'inverse, `kernels` n'est qu'une incitation pour le compilateur à optimiser une zone, dans les limites de ses capacités d'analyse, et de ce qu'il identifiera comme sans risque pour l'intégrité du résultat. C'est donc une façon simple d'optimiser son code, et éventuellement de se faire une idée de ce qu'il est possible d'atteindre comme performances, mais il n'est pas garanti de tirer le maximum de performances atteignables à l'aide d'*OpenACC*.

A.3 La directive `seq`

A l'inverse de `parallel` et `kernels`, cette directive sert à déclarer une portion de code comme séquentielle.

A.4 La directive `loop`

La directive `loop` donne une indication au compilateur sur le fait que la boucle qui suit est parallélisable. Elle peut s'employer seule, ou comme argument de `parallel` ou `kernels`. Elle n'est jamais superflue car elle donne une indication supplémentaire au compilateur sur la démarche à suivre.

La différence entre le multithreading sur *CPU* et sur *GPU* est que le *GPU* est constitué d'un nombre bien plus important d'unités de calcul indépendantes, organisées en groupes eux-mêmes indépendants. Il est donc possible d'exécuter en parallèle des portions de boucles sur différents groupes d'unités de calcul appelés `gang` à l'aide de la directive éponyme, à la manière de la directive `parallel for` de *OpenMP*. Cependant, au sein même de cette parallélisation, il est possible d'effectuer une "sous parallélisation" sur les unités de calcul indépendantes, appelées `worker` à l'aide de la directive éponyme, qui peuvent alors se partager une autre boucle, au sein de laquelle il est possible d'effectuer une vectorisation des calculs à l'aide de la directive `vector`.

A.5 La directive `routine`

Elle permet de déclarer au code une fonction qui sera appelée dans une boucle à paralléliser. Il est important de déclarer ses fonctions de cette façon car elles risqueraient sinon d'empêcher des parallélisations.

A.6 La directive `atomic`

Identique à son équivalent *OpenMP*, cet argument permet de préciser le comportement des threads vis-à-vis d'une variable partagée lorsque l'argument `reduction` est trop restrictif. Il est évidemment moins efficace car plus permissif.

B Les arguments *OpenACC*

Aux directives s'ajoutent des arguments qui précisent le comportement à adopter pour le compilateur.

B.1 L'argument `reduction`

Identique à son équivalent *OpenMP*, cet argument permet de préciser le comportement à avoir vis-à-vis d'une variable dont chaque thread a une copie locale. Il peut s'agir d'une somme, d'une addition, d'un calcul de minimum, de maximum, ou un certain nombre d'opérations bits à bits.

B.2 L'argument `private`

Cet argument permet de préciser les variables dont chaque thread doit avoir une copie locale. Il est à noter que les variables déclarées dans une boucle seront locales, les itérateurs seront locaux, les variables consultées seront locales et initialisées à leurs valeurs avant la boucle à chaque itération.

B.3 Les arguments `gang`, `workers` et `vector`

Lors de la parallélisation de calculs sur *GPU*, il existe plusieurs niveaux de parallélisme. En effet, une carte graphique est composée de nombreuses unités de calcul indépendantes, appelées `workers`, qui sont organisés en groupes appelés `gangs`, et qui, comme les cœurs d'un *CPU*, peuvent effectuer des calculs vectorisés. Il y a donc trois niveaux de parallélisme à considérer, et il est intéressant, au sein de plusieurs boucles imbriquées, d'étudier les différentes répartitions des tâches pour en déterminer la plus efficace. En effet, après chaque directive `loop`, il est possible de préciser sur quel niveau on souhaite décomposer la boucle, avec les arguments `gang`, `worker` et `vector`. Ils sont a priori toujours dans cet ordre, et l'argument `vector` est a priori toujours sur la boucle la plus intérieure (en effet il n'est possible de vectoriser que des calculs sur des tableaux).

B.4 L'argument `collapse()`

Cet argument permet, lors d'une imbrication importante de boucles, de ne plus itérer que sur un seul indice pour faciliter la vectorisation. L'entier en paramètre précise combien de boucles on souhaite combiner.

C Gestion de la mémoire

La gestion de la mémoire est l'un des points clefs du calcul sur *GPU*. En effet, le transit de données entre la mémoire du *CPU*, la mémoire vive, et la mémoire du *GPU* est très coûteux en temps, il est donc essentiel de savoir bien organiser ses données et ne pas faire de déplacements superflus. On appelle ces tranfserts de mémoire l'*offloading*. Il est possible d'ajouter localement aux directives déjà présentes des arguments de déplacement de données spécifiques, mais le plus simple est de créer autour de la zone à paralléliser une clause `data`.

C.1 La clause `data`

Comme toutes les directives d'*OpenACC*, elle se déclare de la façon suivante :

```
#pragma acc data
```

On y ajoute ensuite des arguments de déplacement de données pour préciser ce que l'on veut faire. Comme précisé précédemment, ces arguments peuvent être ajoutés aux différentes directives, notamment `parallel`, `loop`, ou `kernels`, au risque cependant de créer des déplacements inutiles.

On précisera entre parenthèses les tableaux de données que l'on souhaite transférer, suivi entre crochets de l'indice de l'entrée à laquelle on veut commencer le transfert, suivi du nombre d'entrées à transférer. Par exemple, `A[0:N]` permettra de transférer les données de l'indice `0` à l'indice `N`. On notera qu'il est en théorie possible d'écrire `A[:]` pour transférer toutes les données contenues dans `A`, mais il est possible que l'*hôte* n'arrive pas à trouver de lui-même le début et la fin du tableau.

C.2 L'argument `copyin()`

Cet argument sert à copier les données spécifiées dans la mémoire de l'*accélérateur* pour y faire des opérations. Ces données ne seront pas actualisées dans la mémoire du *CPU* ni dans la RAM et les potentielles modifications effectuées dessus seront perdues à la fin de la portion de code concernée par la directive.

C.3 L'argument `copyout()`

Cet argument sert à créer une zone de données dans la mémoire de l'*accélérateur* de la taille des éléments passés en paramètre, zone dans laquelle on pourra stocker les résultats des opérations effectuées dans la portion de code concernée par la directive, et qui ensuite sera exportée dans la RAM, remplaçant les données présentes s'il y en avait.

C.4 L'argument `copy()`

Cet argument cumule les fonctions de `copyin()` et `copyout()`. Il est donc utile pour les données qui ont déjà été initialisées, sur lesquelles on souhaite faire des modifications qui seront conservées pour la suite de l'exécution.

C.5 L'argument `present()`

Cet argument sert à spécifier que les éléments passés en paramètre sont déjà présents dans la mémoire du *GPU* (suite à une directive antérieure) et que l'on cherche à les réutiliser (attention, il n'est pas certain que les modifications effectués hors *GPU* soient actualisées).

C.6 L'argument `update()`

Cet argument permet de demander à l'hôte de simplement résoudre les deltas entre un jeu de données déjà en mémoire de l'*accélérateur* qui a été actualisé, par un autre *accélérateur* ou par l'*hôte*, au lieu de le copier totalement.

C.7 Le préfixe `p-`

Les arguments ci-dessus ont vocation à disparaître au profit des mêmes précédés de "`p`" (*ex :* `pcopyin`, `pcreate`...), qui signifie "*Present or ...*", et qui a pour effet que le *CPU* vérifie si les données ne sont pas déjà présentes avant de les copier, ce qui peut faire gagner un temps précieux si la région `data` est à l'intérieur d'une boucle par exemple.

C.8 Les structures en C++

On notera qu'il est possible d'utiliser des classes en C++ avec des accélérateurs, il suffit d'ajouter les directives `OpenACC` nécessaires aux différents constructeurs et destructeurs, mais cela ne sera pas détaillé ici.

D Affinage du code

Une fois que le code a été parallélisé et que les transferts de mémoire ont été gérés, il est toujours possible de gagner en performances en ajustant le code à l'*accélérateur* sur lequel on veut le faire tourner. Le code perd alors de sa portabilité, mais il devient possible d'obtenir de bien meilleures performances.

D.1 Gestion des gangs, workers et vecteurs

Par défaut, une fois la structure de la parallélisation indiquée au compilateur, celui-ci va déterminer les nombres de `gangs`, de `workers` et la taille des `vectors` automatiquement. Il est cependant possible de lui préciser ces valeurs dans la limite de ce que l'*accélérateur* peut supporter. Dans le cas d'une région de type `kernels`, on précisera directement les nombres de `gangs`, `workers`, et la taille des `vecteurs` en paramètres des arguments correspondants. Dans le cas d'une région de type `parallel`, on précisera ces valeurs après la directive `parallel` avec les arguments `num_gangs()`, `num_workers()` et `vector_length()`. Il est recommandé de faire différents essais afin de trouver les paramètres optimaux. Attention cependant, des valeurs trop élevées ne produiront pas nécessairement d'erreurs à la compilation, mais à l'exécution.

D.2 Gestion des *kernels* et de la synchronicité

Lors de l'exécution d'un programme sur *GPU*, les ensembles de calculs successifs sont rassemblés en un noyau de calcul, que l'on appelle un *kernel*. Pour chaque portion de code exécutée sur *GPU*, un *kernel* doit être mis en place par l'*hôte*. C'est-à-dire qu'il détermine la répartition des calculs sur l'*accélérateur* pour chaque appel. Une fois que cette préparation est faite, les instructions sont envoyées à l'*accélérateur* qui ne fait qu'exécuter les calculs. L'*hôte* attend alors

qu'un *kernel* soit terminé avant d'en mettre en place un nouveau. Or, le temps d'attente de l'*hôte* pourrait parfois être employé à effectuer des opérations indépendantes, comme par exemple la mise en place d'un nouveau *kernel*. Il est alors possible de dire à l'hôte de ne pas attendre la fin d'un *kernel* pour effectuer des opérations en ajoutant l'argument `async` aux directives de parallélisation de boucles. Ensuite, lorsqu'il est nécessaire d'attendre que tous les calculs sur l'*accélérateur* aient été exécutés, on peut insérer la directive `wait`, qui attendra la fin de l'exécution des *kernels* pour lancer les opérations suivantes.

Il est à noter que l'argument `async` ne compromet pas l'ordre d'exécution des opérations, il permet juste d'exécuter des opérations sur l'*hôte* pendant que des opérations sont effectuées sur l'*accélérateur*. Il faudra alors porter une attention toute particulière aux données traitées, car chacun des deux agira sur ses copies locales des variables, ce qui peut compromettre par exemple la terminaison d'une boucle si l'*accélérateur* effectue des opérations sur la variable de terminaison alors que c'est l'*hôte* qui est en charge de vérifier la terminaison, car sa variable ne sera jamais modifiée.

Il est aussi possible de créer plusieurs files d'exécution, et de préciser dans quelle file on souhaite voir un ensemble d'instruction exécuté en ajoutant en paramètre de l'argument `async()` le numéro de la file d'exécution. Les éléments d'une même file d'exécution sont exécutés séquentiellement, les éléments de files différentes sont exécutés concurrentiellement. Il devient alors possible d'effectuer des calculs sur une partie des données, et de transférer les résultats pendant que l'on lance d'autres calculs simultanément.

D.3 Gestion de plusieurs accélérateurs

Il est possible qu'il y ait plusieurs accélérateurs disponibles au sein d'une même machine. Il est alors intéressant d'exploiter cette opportunité. On peut récupérer le nombre d'accélérateurs de type `device_type` via la fonction `acc_get_num_device(device_type)`, et le numéro de l'*accélérateur* sur lequel on travaille via `acc_get_device_num(device_type)`. On peut changer l'accélérateur sur lequel on travaille via l'instruction `acc_set_device_num(n, device_type)`. Le travail par bloc devient alors d'autant plus pertinent.

Table des figures

Références

[1] "Processeur." https://fr.wikipedia.org/wiki/Processeur, 2020. [Online - accesssed on August 2020].

[2] "Compute unified device architecture." https://fr.wikipedia.org/wiki/Compute_Unified_Device_Architecture, 2020. [Online - accessed on August 2020].

[3] "Microprocesseur multi-cœur." https://fr.wikipedia.org/wiki/Microprocesseur_multi-c\T1\oeur, 2020. [Online - Accessed on August 2020].

[4] "Flops." https://fr.wikipedia.org/wiki/FLOPS, 2020. [Online - accessed on August 2020].

[5] "Processeur graphique." https://fr.wikipedia.org/wiki/Processeur_graphique, 2020. [Online - accessed on August 2020].

[6] "Parallélisme." https://fr.wikipedia.org/wiki/Parall%C3%A9lisme_(informatique), 2020. [Online - Accessed on August 2020].

[7] "Pci." https://fr.wikipedia.org/wiki/PCI_(informatique), 2020. [Online - Accessed on August 2020].

[8] "Openacc programming and best practice guides." https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf, 2015. [Online - accessed on August 2020].

[9] "The openacc programming interface." https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf, 2019. [Online - accessed on August 2020].