

Portage de codes C sur GPU avec OpenACC

Quentin KREMP

3 juillet 2020

Table des matières

1 Motivations et contexte

1.1 Enjeux du calcul sur GPU

L'objectif du High Performance Computing (*HPC*) est de réaliser un maximum de calculs en un minimum de temps. Les optimisations de codes sur processeurs, avec exploitation du multithreading et des emplacements de données en mémoire ont été un objet d'étude depuis les années 80. Or, depuis les années 90, le développement des environnements graphiques a amené progressivement à adjoindre des unités de calculs 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, mais avec beaucoup plus de parallélisme. Aussi, dès la fin des années 2000, les constructeurs de *GPU* ont vu la possibilité de les exploiter pour le *HPC*. Or si la parallélisation sur le processeur ne consiste "que" en du multithreading, et exploiter au maximum la vectorisation, la parallélisation sur *GPU* offre plusieurs niveaux de parallélisation du travail, et son absence d'autonomie demande d'apporter une attention particulière aux transferts de mémoire, très coûteux en temps, ainsi qu'à la planification et la répartition du travail.

1.2 Gestion des gangs, workers et vecteurs

Sur un *GPU*, il y a plus de niveaux de parallélisation possible. 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 en **vectors**, bien qu'ils ne possèdent à priori pas d'instructions de vectorisation à proprement parler. On peut supposer que chaque niveau de parallélisation à correspond essentiellement à de la mémoire partagée par exemple. Tout le problème de notre parallélisation va alors être de correctement répartir les différents niveaux de parallélisation au sein de boucles imbriquées, et de bien choisir les nombres de **gangs** et de **workers**, et de bien choisir la taille des **vectors**

1.3 Gestion de la mémoire

Aujourd'hui, 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 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 nécessaires 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*.

1.4 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 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 la synchronisation inutile, ou dans la préparation de *kernels* superflus.

2 L'outil *OpenACC*

2.1 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, 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 à priori se faire sur la machine sur laquelle le code sera utilisé. Ceci s'explique par le fait qu'*OpenMP* ne donne des directives ne concernant que des opérations exécutées par le processeur, et qui s'adapteront automatiquement aux spécifications du processeur, c'est à dire au nombre de coeurs, 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

générale jusqu'à 20% moins bonnes. Cependant, cette perte de performance se fait au gain d'une part d'une simplicité d'écriture, et surtout 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 minimales.

Toutes les directives seront écrites de la façon suivante :

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

2.2 Compilation

Non seulement la compilation ne pourra se faire réellement efficacement que sur la machine qui fera tourner le code, mais en plus tout 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* annonce 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*, que l'on utilisera par conséquent pour du matériel *Nvidia* car c'est la configuration implémentant le mieux les spécifications de *OpenACC*.

2.3 Profiling

Avant de commencer à baliser son code avec des directives *OpenACC*, il est préférable de la tester sur de petits exemples, et de l'exécuter via un outil que l'on appelle un profiler, qui va étudier les appels mémoire et les temps d'exécution des différentes sections de code, dans le but d'analyser quelles sont les sections qui ralentissent l'exécution du programme, afin de déterminer si leur exécution sur le matériel graphique de la machine serait intéressant ou non. En effet, l'envoi de données du CPU (Central Processing Unit, ou processeur) vers le GPU (Graphical Processing Unit, ou carte graphique) 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 tel les profilers. Il est donc nécessaire durant les étapes de parallélisation sous *OpenACC* d'étudier régulièrement le profil d'un code pour comprendre les étapes clefs des temps d'exécution.

2.4 Remarque

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 coeur du CPU), et les *accélérateurs*, qui sont tout les composants capable de calculer en parallèle de l'*hôte* (C'est à dire les autres coeurs 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 coeurs.

2.5 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 itérations 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.
- 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 différentes 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.

3 Etude de cas académiques avec *OpenACC*

Pour tester les performances d'*OpenACC*, on s'est penché sur trois exemples de codes classiques : le *DGEMM*, soit l'opération $\alpha \times AB + \beta \times C$, avec A , B et C des matrices, le calcul d'intégrales par l'approximation des trapèzes, le calcul des différences finies, et une fonction de transposition qui permettra de montrer les possibilités de travail asynchrone.

Ces codes ont été compilés et exécutés sur une machine équipée d'une carte graphique (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.

3.1 *DGEMM*

On a recodé la librairie *DGEMM* de la façon suivante :

```
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_gangs(512) 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_t[j * B_l + k];
12    }
13    C[i * C_c + j] += c;
14  }}}
```

Implémentation de *DGEMM* avec *OpenACC*

Détail des clauses :

- **# pragma acc data** : On commence par créer une zone de données englobant toute notre fonction, qui indique les déplacements en 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, en précisant que l'on souhaite récupérer la matrice *C* en fin de calcul, et on crée une zone mémoire *B_t* dédiée pour calculer la transposée de *B*, que l'on effectue afin de permettre la vectorisation des calculs.
- **# 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.

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 de son fonctionnement interne. En effet, les opérations de vectorisations ne sont à 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 tout les nombres de workers possible à partir de 2. Cependant, si 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 cependant. 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

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

TABLE 2 – Mesure des temps pour une matrice carrée de taille 10000, en secondes

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. 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.

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. 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.

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

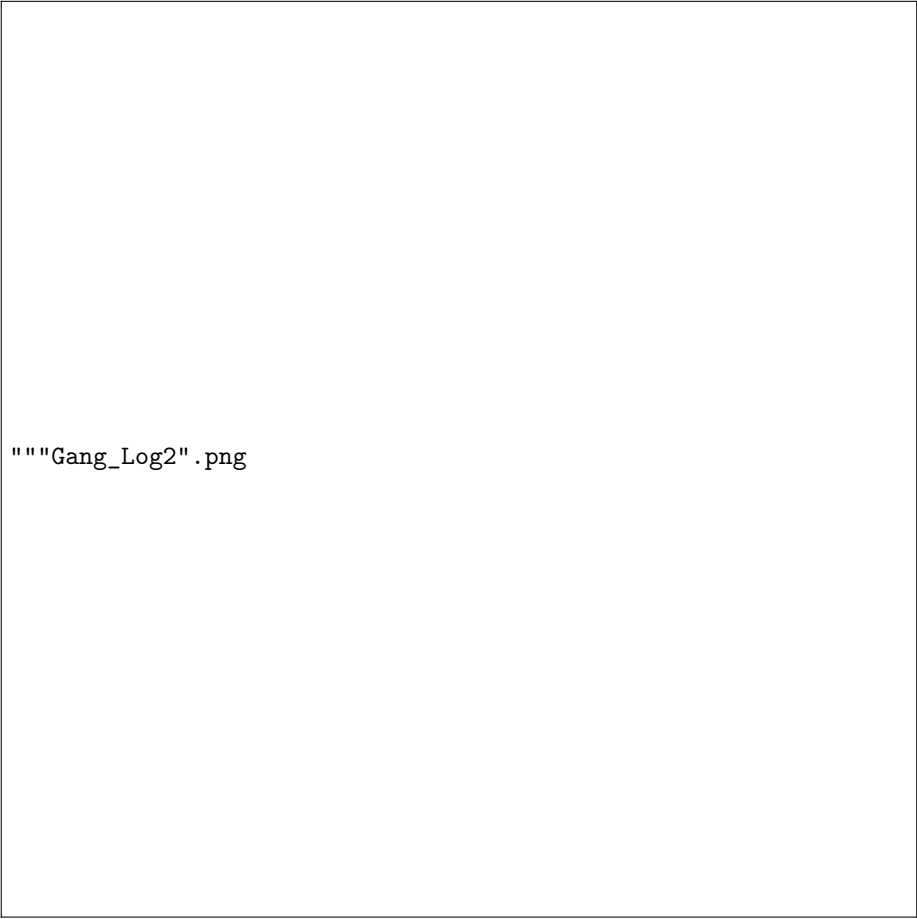
$$\text{flops} = \frac{(\text{taille des matrices})^2 \times (2 \times \text{taille des matrices} + 3)}{\text{temps d'exécution}}$$

On calcule alors les FLOPS théoriques en fonction des temps d'exécution. On prendra comme paramétrage 32W/32V, et on ne précisera pas de nombre de gangs. La courbe suivante est elle aussi en échelle logarithmique :

On peut alors comparer le programme ainsi obtenu à un *DGEMM* classique comme celui de *Intel*, contenu dans la librairie *MKL*.

3.2 Intégration par la méthode des trapèzes

On a implémenté l'intégration par la méthode des trapèzes de la façon suivante :



""Gang_Log2".png

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

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



```

14  return exp(x);
15  }

```

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

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 tout les trapèzes calculés.
- **#pragma acc routine** On déclare ici que la fonction qui suit va être appelée et peut être parallélisée, voir vectorisée lors de ces appels.

3.3 Différences finies

On a implémenté les différences finies de la façon suivante :

```

1  double* iteration(int T, int N, double* C, double* Cnew){
2      double dt = 1./((double)(T-1));
3      double dx = 1./((double)(N-1));
4      #pragma acc data pcopy(C[0:N*N]) pcreate(Cnew[0:N*N])
5      {
6          for(int n = 0; n < T; n++){
7              #pragma acc parallel loop gang num_gangs(4096)
8                  vector_length(512)
9                  for(int i = 1; i < (N-1); i++){
10                     #pragma acc loop vector
11                     for(int j = 1; j < (N-1); j++){
12                         Cnew[N * i + j] = (1 - 4 * dt / (dx * dx)) * C[N
13                             * i + j] + dt / (dx * dx) * (C[N * (i + 1) + j
14                             ] + C[N * (i - 1) + j] + C[N * i + (j + 1)] +
15                             C[N * i + (j - 1)]);
16                     }
17                 }
18             }
19             double* temp = Cnew;
20             Cnew = C;
21             C = temp;
22         }
23     }
24     return C;
25 }

```

Implémentation des différences finies

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 que précédemment de balisage et structuration de la parallélisation du code que pour *DGEMM*.

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

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

Ce qui produit la courbe suivante :

3.4 Transposition

On a implémenté la transposition de la façon suivante :

```

1  double* transpose(double* A, int A_l, int A_c, int
    threads){
2
3      int num_blocks = threads;
4      int block_c_size = (A_c / num_blocks);
5      double* A_t = malloc(A_l * A_c * sizeof(double));
6
7      #pragma acc data pcopyin(A[:A_l * A_c]) pcreate(A_t[:A_c
    * A_l])
8      {
9
10         for(int i = 0; i < num_blocks; i++){
11             #pragma acc parallel loop gang num_gangs(256) num_workers
    (32) async(i)
12                 for(int k = 0; k < block_c_size; k++){
13                     #pragma acc loop worker
14                         for(int l = 0; l < A_l; l++){
15                             A_t[i * block_c_size * A_c + k * A_l + l] = A[l *
    A_c + i * block_c_size + k];
16                         }
17                     }
18             #pragma acc update self(A_t[i * block_c_size * A_c:A_l *
    block_c_size]) async(i)
19         }

```

```

20 #pragma acc wait
21 }
22     return A_t;
23 }

```

Implémentation de la transposition

Détail des clauses :

- **#pragma acc data** Comme vu précédemment, on décrit les transferts de donnée souhaités
- **pcopyin(A[:]) pcreate(A_t[:])** Ici on importe les données de **A** dont on a besoin pour transposer, en revanche au lieu d'un **copyout(A_t[:])**, on utilise **create(A_t[:])** qui va nous permettre de moreceller les transferts de donnée et prendre avantage de la possibilité de travailler simultanément sur l'hôte et sur l'accélérateur.
- **#pragma acc parallel ... async(i)** Les directives et paramètres sont ici identiques à ceux de *DGEMM*, mais il y a ici en plus l'argument **async(i)** qui permet de placer la transposition dans le "thread" **i**
- **#pragma acc update self(A_t[:]) async(i)** Cette directive permet de mettre à jour les données de **A_t** dans la mémoire de l'hôte (ce qui est précisé par le paramètre **self**). On remarquera que l'on ne met ici à jour que la plage de donnée d'un bloc en mémoire et non toute la matrice, et que l'on a mis cette actualisation dans le "thread" **i**. Cela permet que les calculs de transposition et de transfert de mémoire d'un même bloc se fassent séquentiellement, mais que ceux de blocs séparés se fassent parallèlement pour gagner du temps
- **#pragma acc wait** Enfin, à la fin de la région parallèle, on s'assure que tout les threads sont bien synchronisés avant de reprendre l'exécution.

4 Le cas des éléments finis

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, comme ici on cherche 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 parager 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 sinon elles risqueraient 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**

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 coeurs 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 à priori toujours dans cet ordre, et l'argument **vector** est à 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.

Pour résumer, on construit une parallélisation sur GPU de la façon suivante :

C Gestion de la mémoire

La gestion de la mémoire est l'un des points clef 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 transferts

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 ajouté 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 crochet 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 fonction 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 donnée 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 sauver un temps précieux si la région `data` est à l'intérieure 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 alors 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, il va déterminer le nombre 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 correspondant. 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 opération, 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 une "itération". 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. Cette attente est coûteuse et non nécessaire dans les cas où les jeux de données ne sont pas interdépendants. Il est alors possible de dire à l'hôte de ne pas attendre la fin d'un *kernel* pour en mettre en place un nouveau en ajoutant l'argument **async** aux directives de parallélisation de boucles. De plus, si la synchronicité est requise pour certaines opérations, il est possible de la récupérer en ajoutant la directive **wait** devant ces opérations.

Il est possible aussi de créer plusieurs files d'exécution, et de préciser dans quelle file d'exécution 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 d'exécutions différentes sont exécutés concurrentiellement. Il devient alors possible de séparer le travail en blocs indépendants, ce qui permet que les kernels soient préparés de manière asynchrone et permet potentiellement un gain de temps important si un grand nombre de kernels sont exécutés successivement.

On peut aussi se servir de cet argument pour effectuer des transferts de données via l'hôte pendant que l'accélérateur effectue des calculs.

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.

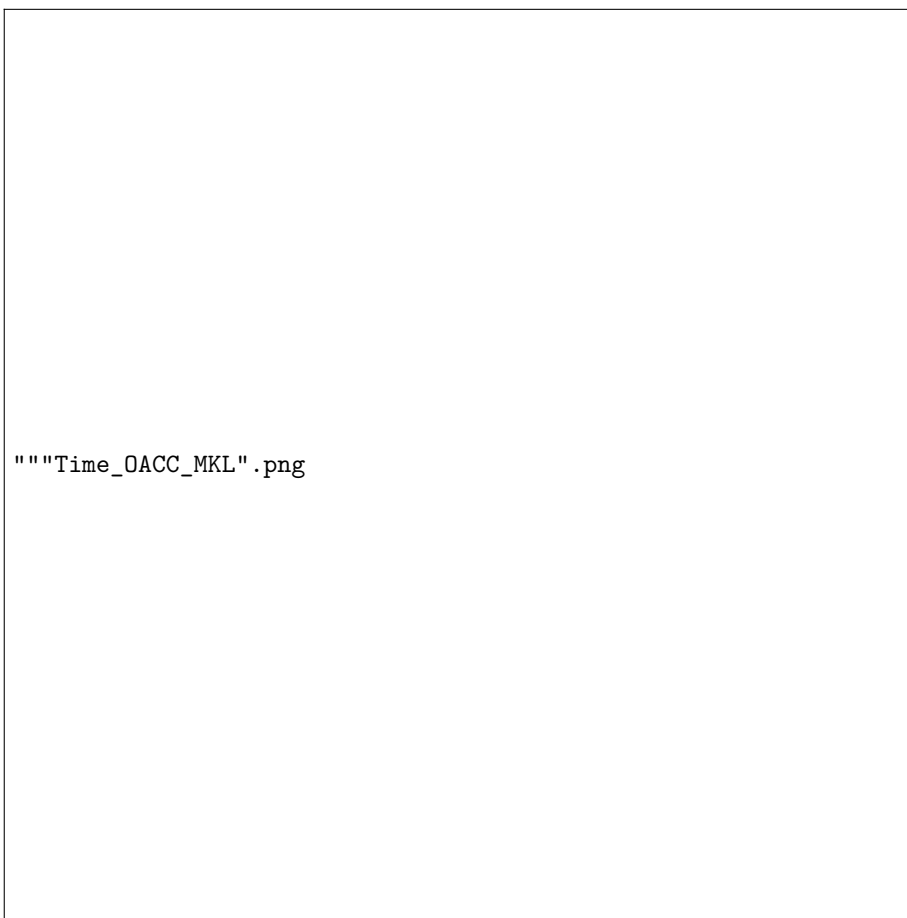


FIGURE 2 – Comparaison des temps d'exécution en fonction de la taille des matrices carrées





FIGURE 4 – GFLOPS en fonction de la taille de la grille