Portage de codes C sur GPU avec OpenACC

Quentin KREMPP

24 juin 2020

dans cette section, il faut explique quel est l'objectif du PRE.

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 sont 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 planifaction 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

⚠ Ce serait plus clair d'expliquer séparément
1) Architecture
2) modèle de programmation

de bien choisir les nombres de `gangs` et de `workers`, et de bien choisir la taille des `vectors`

1.3 Gestion de la mémoire

Archi.

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 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*.

1.4 Gestion des *kernels* et de la synchronicité

modèle prog.

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.

2 L'outil *OpenACC*

2.1 Philosophie OpenACC

OpenACC : il y a aussi des fonctions, var. d'environn. etc. (comme OpenMP)

OpenACC est un standard d'écriture de code *C*, *C++* et *Fortran*, ayant pour objectif d'exploiter la puissance de calcule 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 êtres 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 minimes.

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'envoie 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 régulièrment 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 *BLAS3*, 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*

3.1 BLAS 3

On a recodé la librairie BLAS de la façon suivante :

```
1  float* custom_gcblas_t(float* A, int A_l, int A_c, float*
       B, int B_l, int B_c, float* C, int C_l, int C_c,
     float alpha, float beta){
2
3    int A_s = A_l * A_c;
4    int B_s = B_l * B_c;
5    int C_s = C_l * C_c;
6
7    float* restrict B_t = malloc(B_s * sizeof(float));
8  #pragma acc data pcopyin(A[0:A_s], B[0:B_s]) pcopy(C[0:
       C_s]) pcreate(B_t[0:B_s])
9  {
10 #pragma acc parallel loop gang num_gangs(512) num_workers
       (32)
11   for(int i = 0; i < B_c; i++){
12 #pragma acc loop worker
13     for(int j = 0; j < B_l; j++){
14       B_t[j * B_c + i] = B[i * B_l + j];
15     }
16   }
17 #pragma acc parallel loop gang num_gangs(512) num_workers
       (32) vector_length(32)
18   for(int i = 0; i < C_l; i++){
19 #pragma acc loop worker
20     for(int j = 0; j < C_c; j++){
21       C[i * C_c + j] *= beta;
22       float c = 0;
23 #pragma acc loop vector reduction(+:c)
24       for(int k = 0; k < A_c; k++){
25         c += alpha * A[i * A_c + k] * B_t[j * B_l + k];
26       }
27       C[i * C_c + j] += c;
28     }
29   }
30 }
31   return C;
32 }
```

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 calcules 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 différent 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 cenpendant. 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 signifie ...)

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 signifie ...)

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 pretinence 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 décorellé 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.

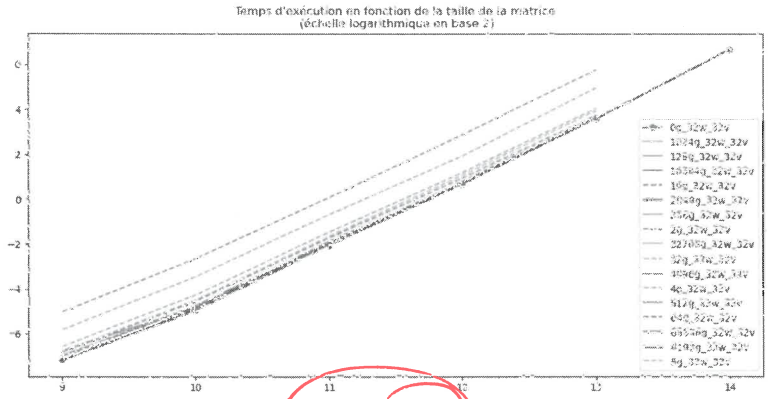

FIGURE 1 – Evolution des temps de calcul en fonction du nombre de gangs

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 preformances. 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.

3.2 Intégration par la méthode des trapèzes

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

```
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)
       vector_length(512)
8      for(int i = 1; i < (N-1); i++){
9  #pragma acc loop vector
10       for(int j = 1; j < (N-1); j++){
11         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)]);
12       }
13     }
14     double* temp = Cnew;
15     Cnew = C;
16     C = temp;
17   }
18   }
19   return C;
20 }
```

Implémentation des différences finies

Détail des clauses :