Projet multicoeurs IT390

Programmation des accélérateurs GPU (Cuda/OpenACC)
Sujet A : Problème des n-corps

MISTRAL Baptiste

LEDERER Victor

ENSEIRB-MATMECA - Filière Matméca : 3A CHP

Enseignant : AUGONNET Cédric

1 Temps en séquentiel

Sans optimisations et sur un seul processus, le temps de calcul est $T_0 = 243,91\ sec$. La figure 1 suivante en présente le détail.

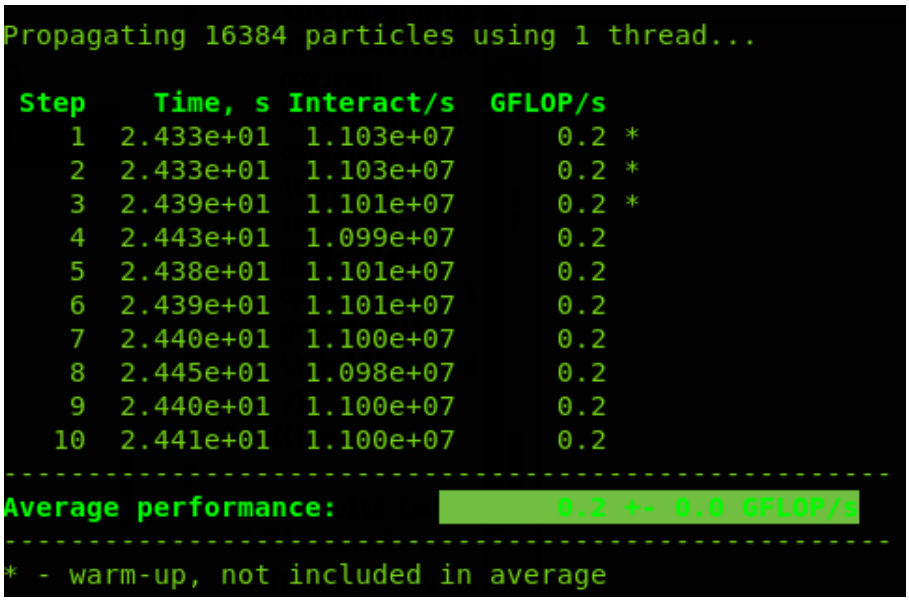

FIGURE 1 – Temps de calcul par itération en temps pour nbody.c(sans optimisation, 1 processus)

Par la suite nous réaliserons le portage sur GPU de la fonction *bodyForce*. Avec **OpenACC**, on réalise l'instruction *#pragma acc parallel loop* sur la boucle externe des particules qui subissent la force. La même boucle sera parallélisée en **cuda**, en utilisant $int\ i = threadIdx.x + blockIdx.x * blockDim.x$; avec

- blockIdx.x : index du block dans la grille
- blockDim.x : le nombre de threads dans le block
- threadIdx.x : index du threads dans le block

De cette façon un unique index est donné par thread, chaque thread itérant sur leur propre index i. Pour valider les différentes implémentations (**OpenACC**, **CUDA**), nous avons sauvegardé dans un fichier les résultats obtenu avec le code séquentiel, en remplaçant le random seed par une initialisation constante.

Important : Tous les temps qui seront comptés et les gains d'amélioration sont sur le code permettant d'enregistrer les positions et les vitesses des particules à chaque pas de temps (1 fichier par pas de temps).

2 Portage du code en OpenACC

On étudie le code séquentiel écrit en C. Le profiling du code indique que la partie du code qui doit être optimisée est la fonction *MoveParticle*. De ce fait, on décide de paralléliser la boucle de calcul des vitesses et des positions pour chacune des particules. On ajoute une directive devant la boucle en i : *#pragma acc parallel loop*.
Cependant, si on execute le *main* sans directives, l'éxecution est rapide mais ne fournit pas les bons résultats par rapport au code séquentiel. En effet, il faut d'abord effectuer une copie des données sur le GPU en utilisant la directive *#pragma acc copy(particle[0 :nParticles])* avant la boucle en temps pour pouvoir allouer la mémoire sur le device. Dans la boucle en temps, on ajoute aussi des directives *#pragma acc update host/device* avant et après l'appel du noyau (qui est en fait la fonction MoveParticles).

Gain en terme de nombre d'opérations : +3.2 Gflop/s

2.1 Portage du code en CUDA (méthode naive)

Dans une première approche du portage CUDA du code, on implémente une première méthode naive qui consiste à paralléliser le calcul des positions et des vitesses dans la fonction *MoveParticles*. De ce fait, on remplace cette fonction par un kernel CUDA et en supprimant la boucle de remplissage : cette boucle est remplacée par une itération sur i = i = threadIdx.x + blockIdx.x * blockDim.x, et ce tant que i ¡ nParticles. La boucle en j n'est donc pas modifiée. On néglige aussi l'impacte de la mise à jour des vitesses car elle ne représente qu'un O(n). Pour les modifications du *main*, on effectue des copies CUDA du vecteur *particle* de type *ParticleType*. On effectue les opérations dessus et on renvoie la copie dans le device pour chaque itération en temps, car on souhaite enregistrer l'ensemble des fichiers de résultats pour vérifier les calculs.

Gain en terme de nombre d'opérations : +3.7 Gflop/s
Après avoir vérifié que les fichiers de résultats correspondent bien à ceux obtenus dans le cas séquentiel, on se rend compte que cette première version permet d'améliorer le temps de calcul ainsi que le nombre d'opérations par secondes : on effectue bien les calcul dans le GPU.

2.2 Analyse des performances du code (profiling)

Après avoir implémenté une version naive du code en CUDA, il s'agit maintenant d'effectuer un profiling permettant de cibler les prochaines améliorations du code. On utilise alors **nvprof** qui permet de traduire toutes les opérations effectuées lors de l'éxecution du code :

```
==128940== Profiling application: ./nbody
==128940== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  2.22489s        10  222.49ms  219.28ms  250.21ms  body_kernel(int, pos_ParticleType*, vit_ParticleType*)
                    0.00%  36.321us         2  18.160us  17.985us  18.336us  [CUDA memcpy HtoD]
                    0.00%  31.520us         2  15.760us  15.648us  15.872us  [CUDA memcpy DtoH]
      API calls:   90.57%  2.18177s         4  545.44ms  43.135us  2.18162s  cudaMemcpy
                    9.37%  225.84ms         1  225.84ms  225.84ms  225.84ms  cudaProfilerStart
                    0.03%  608.15us        97  6.2690us     152ns  279.94us  cuDeviceGetAttribute
                    0.01%  300.39us         1  300.39us  300.39us  300.39us  cuDeviceTotalMem
                    0.01%  151.03us         2  75.515us  5.5670us  145.46us  cudaMalloc
                    0.01%  149.47us         1  149.47us  149.47us  149.47us  cuDeviceGetName
                    0.01%  140.92us         2  70.458us  16.610us  124.31us  cudaFree
                    0.00%  88.751us        10  8.8750us  6.0980us  27.601us  cudaLaunchKernel
                    0.00%  3.2070us         1  3.2070us  3.2070us  3.2070us  cuDeviceGetPCIBusId
                    0.00%  1.6950us         3     565ns     184ns  1.2470us  cuDeviceGetCount
                    0.00%     909ns         2     454ns     186ns     723ns  cuDeviceGet
                    0.00%     282ns         1  _  282ns     282ns     282ns  cuDeviceGetUuid
```

FIGURE 2 – Affichage du profiling de la version **CUDA** naive avec **nvprof**, simulation par défaut.

2.3 Étude du type *ParticleType*

D'après la version fournie du code, le type défini *ParticleType* est une type permettant des faire des calculs en *ArrayOfStructure*(AOS). Il existe cependant un type plus efficace pour le stockage et le traitement des données : le type *StructureOfArray*(SOA). Ce type consiste à stocker les positions ensemble ainsi que les vitesses pour chacune des projections sur les axes. On doit alors créer un type contenant 6 vecteurs de taille *nParticles*. De ce fait, il s'agit simplement de changer les affectations lors des boucles pour le pointeur *particle-¿x[i]*. Pour le calcul et la procédure de copie das le GPU, le code reste similaire. Cette structure de stockage pour les valeurs offre une meilleure proximité en mémoire et permet un gain de calcul non négligeable :

Gain en terme de nombre d'opérations : +67.2 Gflop/s.
On peut donc voir qu'adapter en conséquences la structure du type *ParticleType* permet d'obtenir un gain non négligeable en terme de performances. Il s'agit maintenant d'améliorer la réutilisation des données dans le code.

3 Amélioration du taux de réutilisation des données

Nous nous sommes inspirées du produit matrice/matrice, cela consiste à former une gille de D_1/D_2 $nparticles/p \times nparticles$, avec p le nombre de threads découpant la structure de donnée du type $ParticleType$, comprenant la position et la vitesse. La taille des blocks de cette grille est p. La dimension D_1 s'execute en parallèle tant dis que la dimension D_2 est en parallèle sur chaque threads. La première étape consiste à charger un premier block depuis la mémoire partager, puis à synchroniser les unités de calcul et réaliser les calculs. On synchronise alors les threads et recommence l'opération pour un nouveau block. Pour notre travail, nous nous sommes basé sur :
`https://www.cc.gatech.edu/~hyesoon/spr10/lec_cuda5.pdf?fbclid=IwAR0fAN6Pq983kVJsyedHx2E97-uAlv`
`https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body`
`https://github.com/harrism/mini-nbody`

Gain en terme de nombre d'opérations : 3083.2 ± 50.4 **Gflop/s**
Nous avons aussi vérifié que les résultats sur la positions des particules et vitesse étaient les mêmes au cours du temps que la version séquentielle (en choisissant une initialisation fixe). Pour obtenir un tel gain, nous avons aussi utilisé les *SoA* données par les types *float3* et *float4* de cuda (voir :`http://www.icl.utk.edu/~mgates3/docs/cuda.html?fbclid=IwAR07nIPHE56L1QLMgH6Pk_C4Ew9j02oPmtfKTkpnHAw203fd_yGBxadjUIo`). En effet, pour eviter de réaliser de multiple transaction mémoire il est préférable de performer une coalesced memory access (alignement des données, type contigue). Pour ce faire la taille des structures lues doit être 4,8,16. D'où l'utilisation de *float4*.

4 Bonus : Parallélisme sur CPU avec OpenMP

Après avoir vu les améliorations que peuvent apporter le portage sur GPU du code, on peut se demander s'il est possible d'améliorer le code séquentiel en utilisant du parallélisme seulement sur CPU. De la même manière que pour l'OpenACC, on choisit de paralléliser sur CPU en utilisant des directives avec OpenMP. La directive principale du code est celle qui permet de paralléliser la boucle de remplissage en i dans la fonction *MoveParticles* en considérant le j de la boucle imbriquée en tant que variable privée.

5 Conclusion

Lors de ce projet, nous avons étudié le problème des n-corps à partir d'un code C à optimiser. Nous avons vu que le portage sur GPU avec OpenACC permettait un gain de performances en utilisant les bonnes directives. Cependant, il a été vu que transformer la structure du code en utilisant **CUDA** permettait des améliorations bien plus importantes. Même si l'implémentation naive permet d'obtenir de bons résultats, l'étude des structures à utiliser se révèle être très pertinente pour permettre une meilleure accessibilité des données dans la carte graphique. De plus, lors des appels de noyau et des calculs de solutions, améliorer le taux de réutilisation des données se révèle essentiel pour obtenir des gains de performance encore plus significatifs.