Rapport du tp Nbody3D

January 22, 2022

Nom et Prénom:Diop Elhadji Fallou
Numéro Étudiant: 22101564

1 Sommaire

Introduction

Modification au niveau du code

Carateristiques de la machine knl03

compilateur

gcc

icc

clang

Performance avec perf

gcc

icc

Conclusion

Références

2 Introduction

L'optimisation en informatique est quelque de délicat puisqu'elle dépend de plusieurs paramètres:l'algorithme du code, la structure du code à optimiser, les compilateurs, les *flags* de compilation,les bibliothèques de calcul,les boucles, et pleins d'autres choses.

Dans notre cas sur le $nbody3D$, on cherche à le optimiser en modifiant certaines parties du code (les calculs mathématiques couteux,la parallélisation des boucles) en comparant aussi les compilateurs (gcc,clang,icc) et regarder aussi les mesures de performance de chacun en temps de calcul et en gigaflops par seconde.

3 Carateristiques de la machine knl03

Le nœud de calcul Intel Knights Landing (64 cores - 256 threads) a : DDR memory: 110GB HBM memory: 16GB (NUMA node 1)
fournisseurid : GenuineIntel modèle : 87 nom du modèle : Intel(R) Genuine Intel(R) CPU 0000 @ 1.30GHz pas : 1 microcode : 0x1ac processeur MHz : 1300 à 1400 taille du cache : 1024 Ko identifiant physique : 0 frères et soeurs : 256 ID de base : 63 cœurs de processeur : 64 apicide : 255 apicide initial : 255 fpu : oui fpuexception : oui niveau cpuid : 13 wp : oui taille clflush : 64 cachealignment : 64 tailles d'adresse : $46x(8octets)$ physiques et $48x(8octets)$ virtuels Mais pour notre cas seulement deux nœuds de calcul sont qui: 0 et 1.
Le nœud 0 contient 256 processeurs. taille du nœud 0 : 96521 Mo nœud 0 libre : 93982 Mo
Le nœud 1 a:
taille du nœud 1 : 16125 Mo nœud 1 libre : 16038 Mo

4 Modification au niveau du code Nbody3d

On a transformé les composants (x ,y,z) et leurs vitesses (vx,vy,vz) de notre particule p en pointeur (*x,*y,*z) et (*vx,*vy,*vz) pour gagner une performance.
On sait que:

$$a^{\frac{3}{2}} = (a^3)^{\frac{1}{2}}$$

.
Donc cette propriété mathématique nous permet de transformer $pow(d, \frac{3}{2})$ en $(d * d * d)^{\frac{1}{2}}$ pour gagner une performance car la fonction $pow()$ prend beaucoup de temps pour calculer le rapport d'un meme nombre avec des exposants différents.
On a utilisé aussi La racine carrée inverse rapide (en anglais fast inverse square root, parfois abrégé Fast InvSqrt()) nous permet de calculer $x^{\frac{-1}{2}}$ pour avoir plus de performance en temps.
On a utilisé OpenMP(Open Multi-Processing) API (Application Programming Interface) standard à titre illustratif pour la programmation d'applications parallèles sur les architectures à mémoire partagée.donc vous pouvez ne pas tenir compte de cela.

5 compilateur

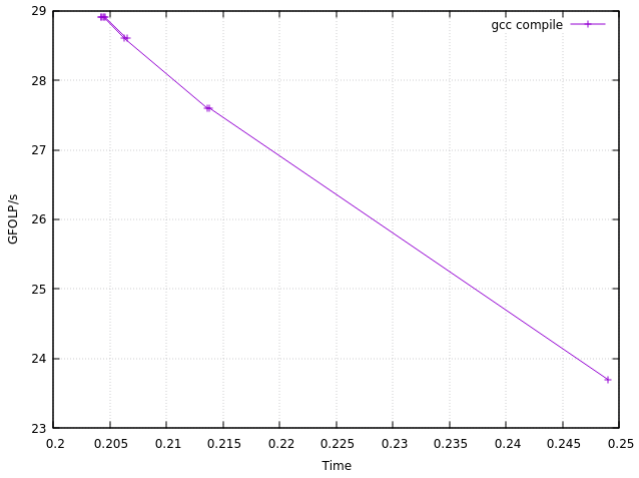
gcc

Le GNU Compiler Collection abrégé en GCC, est un logiciel libre capable de compiler plusieurs langage dont C. Pour le compilateur gcc sur la machine $kn03$avec $march = native - mavx2 - Ofast - fopt - info - all$ le temps de calcul diminue lorsque le gigaflops augmente.$0fast$ permet d'ignorer le strict respect des normes et permet aussi à tous les $-03optimisations$.
Donc elle nous a permis d'avoir une performance moyenne de :

$$29.1 \pm -0.0GFLOP/s$$

.
Et les flags $-march = native - mavx512f - Ofast - fopt - info - all$, on a une performance moyenne de:

$$30.0 \pm -0.0GFLOP/s$$



Mais lorsque je parallélise avec OpenMP,j'ai une performance de

$$1502GFLOPS/s$$

.
Car le OpenMP accélère le programme et donne des directives pour indiquer au compilateur quelles sont les boucles à paralléliser et la distribution des threads.

5.1 icc

Le compilateur *icc* de Intel est compilateur qui produit un code optimisé pour les processeurs.

On utilise la machine à distance knl03 et on fait la compilation avec la flag $-xhost-Ofast-qopt-report$ car il prend en charge tous les $-03compilations$ et on voit avec notre graphe ci-dessous une perturbation et manque de stabilité malgré que sa performance moyenne est de :

$$27.8 \pm -0.0GFLOP/s$$

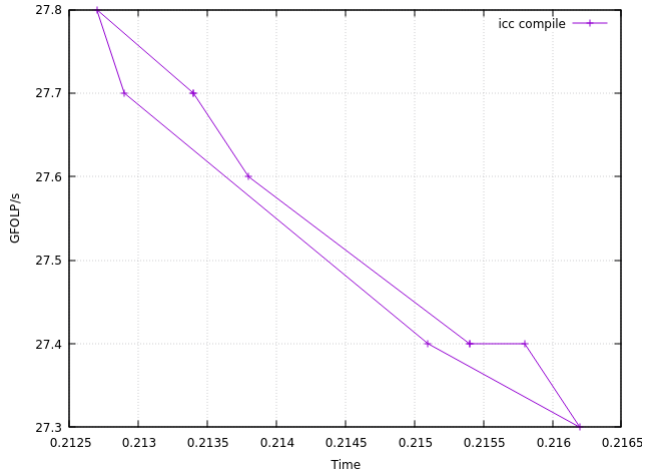
.Et la compilation avec la flag $-xhost-Ofast-qopt--zmm-usage=high$ qui permet d'activer totalement la vectorisation $AVX512$ et déclenche une analyse plus exhaustive des dépendances de données et on arrive avec performance moyenne de

$$27.9 \pm -0.0GFLOP/s$$

. Mais lorsqu'on parallélise avec OpenMP,on une performance de

$$1697.7 \pm -0.0GFLOP/$$

.
Car icc est un outil capable de vectoriser et de paralléliser.Donc avec icc,OpenMp accélère le programme et donne des directives pour indiquer icc quelles sont les boucles à paralléliser et la distribution des threads.C'est pourquoi on assiste à une parallélisation presque parfaite



5.2 clang

Le compilateur *clang* fait partie de la famille de C langages.
Avec les machines à distance *clang* n'a pas fonctionner par ce qu'il reconnait pas la librairie "omp.h",donc on a utilisé notre laptop à titre illustratif pour comprendre ce que cela fait mais vous pouvez ne pas tenir compte de cela. Et

on fait la compilation avec la flag $-march = native - mavx2 - Ofast$ qui tient en compte tous les $-03 compilations$ et on a une performance moyenne de :

$$36.4 \pm -0.0 GFLOP/s$$

.
Car *clang* s'appuie sur l'optimiseur et le générateur de code LLVM, qui lui permet d'avoir une optimisation de haute haute qualité et une prise en charge de la génération de code pour de nombreuses cibles.

Performance avec perf

La performance étant un outil qui énonce les indications permettant de mesurer les possibilités optimales d'un matériel, d'un logiciel, d'un système ou d'un procédé technique pour exécuter une tâche donnée.
Donc nous,on a choisi $perf$, un outil de mesure de performances *linux* pour voir quels sont les points chauds de notre programme pour chaque compilateur.

5.3 gcc perf

Pour la mesure de performances de notre code avec le compilateur gcc on a:
échantillons : 8k de cycles d'événements nombre d'événements (environ): 3005690216
On voit que 98.67 pour 100 du calcul passe sur la fonction moveparticles. Et dans la fonction moveparticules on deux points chauds qui sont :$vfmadd231ps$ (Multipliez les valeurs à virgule flottante simple précision emballées de zmm2 et zmm3/m512/m32bcst, ajoutez à zmm1 et placez le résultat dans zmm1) qui occupe les 25.83 pour 100 du calcul et $vrsqrt28ps$ (Calcule la racine carrée réciproque des valeurs float32 dans l'opérande source (le deuxième opérande) et stocke les résultats dans l'opérande de destination (le premier opérande)) dont les 15.58 du calcul se passent ici.les deux opérations se passent sur les registres zmm donc on peut supposer que y a une dépendance sur les deux opérations comme $waitandread$.Donc pour améliorer la performance on doit optimiser le calcul de la racine carré.

5.4 icc perf

Pour la mesure de performances de notre code avec le compilateur icc on a:
échantillons : 8k de cycles d'événements nombre d'événements (environ): 3005690216
On voit que 98.97 pour 100 du calcul passe dans le main. Et dans le main il y a 7 points chauds qui sont : $vrcp28ps$ (Calcule l'approximation réciproque de la valeur float32ues dans l'opérande source (le deuxième opéraet) et stocker les résultats à l'opérande de destination (le premier opérande)en utilisant le masque

d'écriture k) qui occupe les 21.28 pour 100 des points chauds, et $vfmadd231ps$ qui occupe les 14.21 pour 100 des points chauds et $vrsqrt28ps$ lui aussi occupe les 11.7 pour 100 des points chauds.Les 3 opérations se passent sur les registres aes et les zmm, donc le $waitandread$ prend plus de temps. Pour améliorer la performance il faut une meilleure precision et optimiser le calcul du racine carré.

6 Conclusion

L'optimisation d'un programme reste un problème majeur malgré que les compilateurs fournissent des $flags$ très optimisés.Mais certains sont plus efficaces sur la performance en temps et d'autres sur la précision comme c'est le cas avec gcc et icc.Gcc fait les calculs plus rapides mais icc cherche à trouver la meilleure précision.

7 Références

Cours de : Mr William Jalby et Sala Ibnamar alias Yaspr
https://www.felixcloutier.com/x86
http://www.idris.fr/jean-zay/
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
https://fr.wikipedia.org/wiki/Racinecarr
https://clang.llvm.org/docs/UsersManual.htm