TP nbody3D

Alexandre MAHEUX

22 Janvier 2022

1 Introduction

Le nbody3D implémente le déplacement de particules dans un domaine en trois dimensions d'après la loi de Newton. Le code de base est fourni et l'objectif est d'améliorer ces performances pour diverses raisons. Ce n'est pas le cas ici mais nous pouvons imaginer que nous voulons un code pouvant éventuellement passer à une plus grande échelle avec des millions voire des milliards de particules. Ainsi un code plus performant est justifié (consommation électrique, temps d'exécution, ...). Pour ce TP, on se réduit à une amélioration séquentielle du code.

2 Conditions de travail

Les compilations, exécutions, analyses et profiling ont été réalisés sur la machine knl06, de type Knights Mill (KNM), en mode performance comme l'indique *cpupower* ($f = 1.3\,\text{GHz}$). Les compilateurs utilisés sont GCC de GNU et ICC d'Intel ; Clang LLVM n'étant pas présent sur le knl06. Vu que le travail se fait en séquentiel, on a besoin que d'un CPU (le 177ème ici, un peu choisi au hasard). Les outils d'analyses et de profiling utilisés sont Perf de Linux, MAQAO du LI-PaRAD, Advisor d'Intel. Les flags de base de compilation sont *-g* pour permettre le débugage, *-Ofast* pour une optimisation d'emblée au niveau des opérations mathématiques et de la vectorisation, *-march=native* ou *-xhost* pour indiquer l'architecture en question et *-fopt-info-all* ou *-qopt-report* pour générer un rapport d'optimisation.

3 Version 0

La version initiale est constituée d'un tableau de structure de taille 16384 (défaut) pour stocker les données (Array of Struct), ces derniers étant des float 32 bits. LA taille réelle de ce tableau est donnée en sortie. La fonction init initialise aléatoirement les données dans une boucle. La fonction *move_particles* calcule l'emplacement et la vitesse de chaque particule à l'étape d'après dans une double boucle. Enfin la fonction main simule le déplacement de 16384 particules sur 10 étapes, en mesurant le temps d'exécution de la fonction *move_particles*, le nombre d'intéractions/d'itérations par seconde et le nombre de GFLOP par seconde (nombre d'opérations flottantes par seconde) pour chaque étape. Une moyenne de ce dernier est calculée ainsi que son incertitude. La figure 1 montre la sortie de la version initiale :

```
Total memory size: 393216 B, 384 KiB, 0 MiB

 Step     Time, s  Interact/s  GFLOP/s
    0  1.082e+01  2.481e+07     0.57 *
    1  1.083e+01  2.479e+07     0.57 *
    2  1.082e+01  2.480e+07     0.57 *
    3  1.082e+01  2.481e+07     0.57
    4  1.083e+01  2.479e+07     0.57
    5  1.083e+01  2.479e+07     0.57
    6  1.083e+01  2.479e+07     0.57
    7  1.084e+01  2.477e+07     0.57
    8  1.084e+01  2.477e+07     0.57
    9  1.084e+01  2.477e+07     0.57
-----------------------------------------------------
Average performance:          0.57 +- 0.00 GFLOP/s
-----------------------------------------------------
```

```
Total memory size: 393216 B, 384 KiB, 0 MiB

 Step     Time, s  Interact/s  GFLOP/s
    0  8.497e-01  3.159e+08     7.27 *
    1  8.166e-01  3.287e+08     7.56 *
    2  8.165e-01  3.287e+08     7.56 *
    3  8.164e-01  3.288e+08     7.56
    4  8.167e-01  3.287e+08     7.56
    5  8.166e-01  3.287e+08     7.56
    6  8.168e-01  3.286e+08     7.56
    7  8.168e-01  3.286e+08     7.56
    8  8.164e-01  3.288e+08     7.56
    9  8.160e-01  3.290e+08     7.57
-----------------------------------------------------
Average performance:          7.56 +- 0.00 GFLOP/s
-----------------------------------------------------
```

FIGURE 1 – Résultats de la version initiale avec GCC (à gauche) et ICC (à droite)

Clairement ICC est meilleur que GCC pour la version initiale. Il y a 13.3 fois plus de FLOP/s avec ICC. Le rapport d'optimisation généré pour GCC indique à toute les boucles le message suivant "*missed : couldn't vectorize loop*" qui signifie qu'aucune vectorisation n'a pu être effectuée malgre -Ofast. Dans le cours, on a vu que c'est la forme AOS qui empêche vraiment à GCC de faire son travail. Le rapport d'ICC, quant à lui, indique "*LOOP WAS VECTORIZED*" à toutes les boucles sauf celle dans init. ICC arrive à s'adapter à l'AOS et par des manipulations arrive à vectoriser comme il peut, d'où cette différence entre GCC et ICC. En effet, le message "*vectorization support : non-unit strided load was generated for the variable* $< p.x[j] >$*, stride is 6*" indique qu'il a vu qu'il fallait une stride de 6 pour accéder consécutivement à une coordonnée spécifique dans la mémoire ($[x_0, y_0, z_0, v_{x0}, v_{y0}, v_{z0}, x_1, ...]$, entre x_0 et x_1 il faut faire +6 dans les adresses mémoires), d'où la vectorisation de ICC. Pour améliorer cela, nous avons le Struct of Array (SOA) qui est une structure de tableaux et permet une meilleure vectorisation du code. Il faut donc passer du AOS au SOA pour améliorer les performances.

4 Version 1

La version 1 a donc une structure de données en SOA et les structures en paramètres des fonctions sont passées par pointeur pour éviter les copies inutiles. Maintenant c'est la manière dont les appels sont fait qui est différente ($p[i].x \longrightarrow p.x[i]$). La différence se fait ressentir tout de suite à l'exécution (Figure 2).

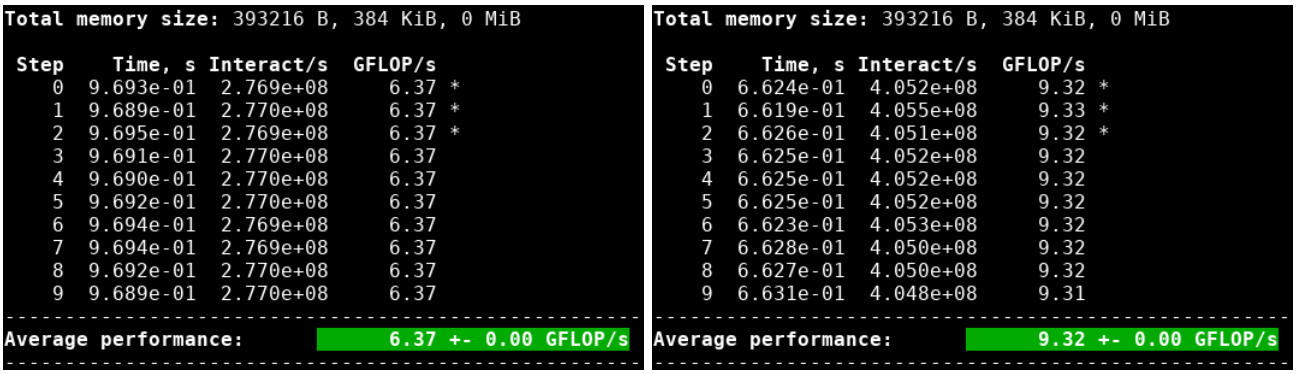

FIGURE 2 – Résultats de la version 1 avec GCC (à gauche) et ICC (à droite)

Avec GCC, la version 1 est 11.3 fois plus performant que la version 0. Avec ICC, c'est 1.2 fois plus performant. Entre GCC et ICC pour la version 1, c'est 1.5 fois plus performant. Le passage en SOA permet à GCC de vectoriser mais pas complètement malheureusement. Le message suivant du rapport de GCC "*nbody1.c :55 :9 : optimized : loop vectorized using 64 byte vectors*" est le seul qui indique une vectorisation d'une boucle. Cela signifie que, si on arrive à vectoriser les autres, on peut atteindre un très bon speed-up avec GCC. ICC, quant à lui, a vu disparaître la stride d'avant pour laisser place à des données non-alignées en mémoire : "*vectorization support : unaligned access used inside loop body*" qui, par rapport au stride, ralentit les lectures en mémoire il me semble. Cependant, il y a bien eu amélioration par rapport à avant et cela est dû à la réduction du nombre d'instructions et du nombre de cycles nécessaires à la réalisation de ces instructions (voir Table 1) mesurés avec l'outil Perf. La version 1 nécessite 1.2 fois moins de cycles à la réalisation des instructions, amélioration obtenu grâce au SOA.

perf stat	version 0	version 1
cycles	10,773,191,667	8,673,714,984
instructions	6,767,140,021	6,093,263,179
insn per cycle	0.63	0.70
user time (s)	8.297	6.682

TABLE 1 – Perf stat avec ICC de la version 0 et 1

Pour avoir une idée de la version 2, nous utilisons *perf record* pour situer les points chauds dans les instructions assembleur générées par GCC et ICC de la version 1. Déjà, comme on pouvait le pressentir, la quasi-totalité du temps de calcul est situé dans la fonction *move_particles* pour les deux compilateurs : 99.78% pour GCC, 98.88% pour ICC. Les améliorations futures seront dans cette fonction. Dans le détail (voir Figure 3), le code généré par GCC passe 43.59% de son temps à calculer une racine carrée et 13.44% dans les divisions (l'instruction *vrcp28ps*,

de ce que j'ai compris, remplace la division 1/x par une version optimisée qui calcule une approximation de la réciproque d'une fonction soit 1/f (fonction déterminée par les valeurs du vecteur/registre ZMM)). Le code généré par ICC passe 19.4% de son temps dans les divisions et 12.36% à convertir un registre ZMM en YMM (instruction *vcvtpd2ps*). Par contre pour la racine carrée, par rapport à GCC, ICC a vu que la ligne "*pow(d$_2$,3/2)*", combinée avec les lignes d'après où on divise par ce dernier, se résume en gros par des calculs d'inverse de racine carrée (instruction *vrsqrt28pd*). ICC y passe très peu de temps (non montré sur la figure 3).

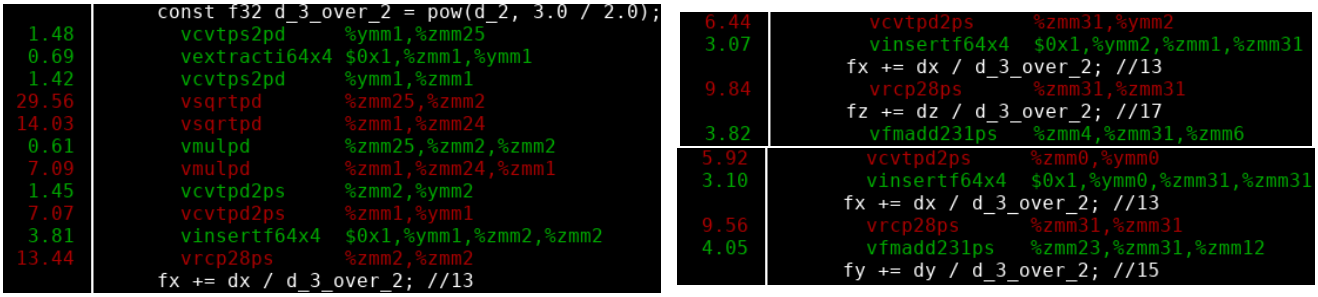

FIGURE 3 – Points chauds dans le code assembleur pour GCC (à gauche) et ICC (à droite)

Il est clair qu'il faut regarder plus en détail les lignes de code correspondant à ces calculs. Si on peut arriver à les transformer pour éviter certaines instructions, ce serait parfait.

5 Version 2

Les versions précédentes ont les lignes suivantes dans la boucle la plus interne dans la fonction *move_particles*. Ceci divise dx, dy et dz par d_2 puissance 1.5.

```
1 const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0);
2 fx += dx / d_3_over_2;
3 fy += dy / d_3_over_2;
4 fz += dz / d_3_over_2;
```

La fonction *pow* et les divisions coûtent vraiment très cher comme nous avons vu précédemment. Mais on peut remarquer que $y = x^{1.5} = \sqrt{x}^3$ et donc $\frac{1}{y} = (\frac{1}{\sqrt{x}})^3$. Il s'en déduit les lignes de code équivalents suivantes :

```
1 f32 d_3_over_2 = 1/sqrtf(d_2);
2 d_3_over_2 = d_3_over_2 * d_3_over_2 * d_3_over_2;
3 fx += dx * d_3_over_2;
4 fy += dy * d_3_over_2;
5 fz += dz * d_3_over_2;
```

On passe de 8 opérations flottantes à 9 mais surtout on passe de 4 divisions à 5 multiplications et de la fonction *pow* à la fonction *sqrt* inverse (*rsqrt* en une seule instruction). La multiplication étant bien moins coûteuse que la division, il est clair que il y aura une nette amélioration ici. La Figure 4 montre la sortie de la version 2. La différence est considérable par rapport à la version 1. Pour GCC, il y a 4.9 fois plus de FLOP qu'avant, et pour ICC 4.4 fois plus. Pour cette version, ICC est 1.3 fois mieux que GCC contre 1.5 fois pour la version d'avant. L'écart entre les deux compilateurs reste assez uniforme pour la version 1 et 2. Dans le rapport de GCC, de ce que je peux comprendre, il n'y a pas de grand changement. Dans le rapport de ICC, le message "*vectorization support : number of FP up/down converts : single/double precision to double/single precision 2/1*" apparaissant dans le rapport de la version n'apparaît plus dans le rapport de la version 2. Cela devait être lié à l'instruction *vcvtpd2ps*, qui convertit un registre ZMM en YMM, pour lequel le code passait le plus de temps dans la version 1.

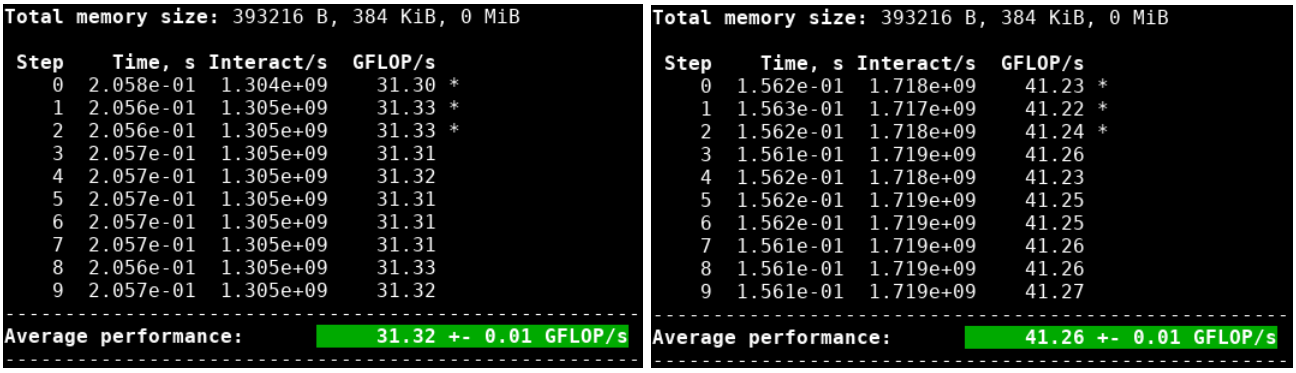

FIGURE 4 – Résultats de la version 2 avec GCC (à gauche) et ICC (à droite)

L'outil Perf nous permet de compléter la Table 1 avec la version 2 (voir Table 2). La manipulation des opérations mathématiques a permis une grosse diminution du nombre d'instructions et du nombre de cycles nécessaires à les réaliser, soit 4.1 fois moins de cycle total que la version 1.

perf stat	version 0	version 1	version 2
cycles	10,773,191,667	8,673,714,984	2,092,905,797
instructions	6,767,140,021	6,093,263,179	2,730,412,340
insn per cycle	0.63	0.70	1.30
user time (s)	8.297	6.682	1.584

TABLE 2 – Perf stat avec ICC de la version 0, 1 et 2

Un visuel sur les codes assembleurs de la version 2 nous indique que les points chauds se sont déplacés sur les multiplications, ce qui est normal vu qu'on a enlevé les divisions. Le calcul de la racine carrée inverse coûte toujours un peu (normal) mais bien moins qu'avant. Il n'y a plus ces conversions ZMM en YMM bizarres et à noter que les registres utilisés sont tous des ZMM ce qui témoigne d'une très bonne vectorisation de la boucle interne en AVX512. Je ne pense pas qu'on puisse faire mieux pour les multiplications.

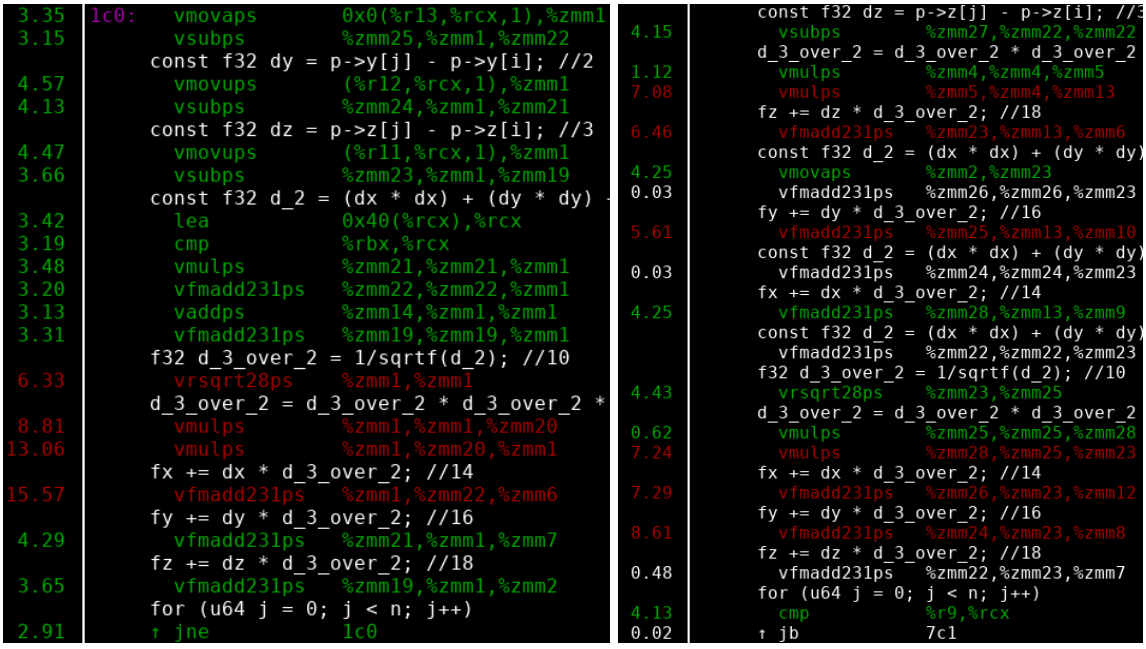

FIGURE 5 – Points chauds dans le code assembleur pour GCC (à gauche) et ICC (à droite)

En prévision d'une version 3, nous nous rappelons que dans le rapport de ICC il y a le message suivant qui revient

dans chaque boucle "*vectorization support : unaligned access used inside loop body*". Cela signifie que toutes les données ne sont pas alignées en mémoire et donc l'accès en mémoire se fait un peu dans le désordre. Nous pouvons tenter, pour améliorer cela, d'aligner ces données et voir si il y a une amélioration.

6 Version 3

La version 3 consiste donc à remplacer les *malloc* par des *aligned_alloc* et de préciser le mot-clé *restrict* pour les variables dans la structure. De plus, des recherches sur Internet amènent à ajouter la fonction *__assume_aligned(array, ALIGN)* juste avant une boucle pour préciser l'alignement. La Figure 6 montre la sortie de la version 3.

```
Total memory size: 393216 B, 384 KiB, 0 MiB

 Step     Time, s Interact/s  GFLOP/s
    0  1.889e-01  1.421e+09    34.09 *
    1  1.888e-01  1.422e+09    34.12 *
    2  1.889e-01  1.421e+09    34.10 *
    3  1.888e-01  1.422e+09    34.12
    4  1.889e-01  1.421e+09    34.10
    5  1.889e-01  1.421e+09    34.10
    6  1.889e-01  1.421e+09    34.11
    7  1.889e-01  1.421e+09    34.11
    8  1.888e-01  1.422e+09    34.12
    9  1.888e-01  1.421e+09    34.11
-----------------------------------------------------
Average performance:          34.11 +- 0.01 GFLOP/s
-----------------------------------------------------
```

```
Total memory size: 393216 B, 384 KiB, 0 MiB

 Step     Time, s Interact/s  GFLOP/s
    0  1.553e-01  1.728e+09    41.47 *
    1  1.554e-01  1.727e+09    41.44 *
    2  1.551e-01  1.731e+09    41.54 *
    3  1.551e-01  1.730e+09    41.53
    4  1.551e-01  1.730e+09    41.52
    5  1.552e-01  1.730e+09    41.52
    6  1.551e-01  1.731e+09    41.54
    7  1.551e-01  1.731e+09    41.53
    8  1.552e-01  1.730e+09    41.51
    9  1.552e-01  1.729e+09    41.51
-----------------------------------------------------
Average performance:          41.52 +- 0.01 GFLOP/s
-----------------------------------------------------
```

FIGURE 6 – Résultats de la version 3 avec GCC (à gauche) et ICC (à droite)

L'amélioration n'est pas aussi fulgurante que précédemment, et aussi je m'attendais à mieux quand même. Pour GCC, on a 1.09 fois plus de FLOP qu'avant, soit une augmentation de 9%. Pour ICC, c'est 1.006 fois plus, soit une augmentation de 0.6% par rapport à la version 2. Pour la version 3, ICC est 1.2 fois mieux que GCC, même si les changements ont permis à GCC de mieux s'améliorer que ICC. Le rapport de GCC ne change pas par rapport à avant et celui de ICC ne montre pas de changement non plus. Les messages sur les données non alignées sont toujours là. Nous pouvons regarder la différence avec perf stat sur la Table 3 suivante. IL y a une légère diminution du nombre de cycles et une légère augmentation du nombre d'instructions. Le gain est vraiment infime.

perf stat	version 0	version 1	version 2	version 3
cycles	10,773,191,667	8,673,714,984	2,092,905,797	2,084,031,453
instructions	6,767,140,021	6,093,263,179	2,730,412,340	2,732,072,872
insn per cycle	0.63	0.70	1.30	1.31
user time (s)	8.297	6.682	1.584	1.562

TABLE 3 – Perf stat avec ICC de la version 0, 1, 2 et 3

Regarder les codes assembleurs correspondants ne sert plus à rien pour l'instant. Le code généré par GCC s'est un peu amélioré mais reste proche de la version 2. Celui généré par ICC est quasiment inchangé et le rapport dit que ce n'est pas aligné alors que c'est spécifié dans le code.
On va essayer d'utiliser l'outil MAQAO pour tenter de mieux analyser la version 3. La Figure 7 et 8 sont les profiles MAQAO de base de la version 3 respectivement avec GCC et ICC. Tout d'abord, pour GCC, MAQAO nous conseille de rajouter le flag *-funroll-loops* pour indiquer au compilateur de dérouler les boucles si possible. Il est vrai que ce flag permet une amélioration des performances en général. De plus, l'accès aux tableaux n'est efficace qu'à 50% donc doit être amélioré si possible. Pour ICC, apparemment pas de manque au niveau des flags même si j'essaierai bien de mettre *-funroll-loops* aussi. L'accès aux tableaux est efficace à 100%, donc c'est bien. Pour les deux compilateurs, il y aurait un potentiel speedup si seules les instructions flottantes sont conservées. Dans le détail des boucles, la boucle interne est 100% vectoriser pour les deux compilateurs et représente 98-99% de la couverture de l'exécution. Précédemment on savait que c'était dans la fonction *move_particles* que l'exécution se déroulait principalement. Maintenant, nous savons que c'est exactement dans la boucle interne de cette fonction que ça passe 98-99% du temps. Même avec une meilleure vectorisation des autres boucles, cela n'améliorera pas de beaucoup le nombre de FLOP et le temps d'exécution.

Total Time (s)		1.90
Profiled Time (s)		1.90
Time in analyzed loops (%)		100.0
Time in analyzed innermost loops (%)		99.5
Time in user code (%)		99.5
Compilation Options		**gcc_nbody3**: -funroll-loops is missing.
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		50.2
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.00
	Nb Loops to get 80%	2
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	2
Fully Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	3
FP Arithmetic Only	Potential Speedup	1.26
	Nb Loops to get 80%	1

FIGURE 7 – Profile MAQAO de la version 3 avec GCC

Total Time (s)		1.58
Profiled Time (s)		1.58
Time in analyzed loops (%)		98.7
Time in analyzed innermost loops (%)		98.1
Time in user code (%)		98.7
Compilation Options		OK
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		100
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.15
	Nb Loops to get 80%	1

FIGURE 8 – Profile MAQAO de la version 3 avec ICC

Loop id	Source Location	Source Function	Level	Coverage run_0 (%)	Max Time Over Threads run_0 (s)	Time w.r.t. Wall Time run_0 (s)	Nb Threads run_0	Vectorization Ratio (%)	Vectorization Efficiency (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized	Speedup If Perfect Load Balancing run_0
5	gcc_nbody3 - nbody3.c:60-73	move_particles	Innermost	98.68	1.88	1.88	1	100	100	1	1	1	1
1	gcc_nbody3 - nbody3.c:30-42	init	Single	0.53	0.01	0.01	1	0	7.07	1.96	1.74	15.26	1
3	gcc_nbody3 - nbody3.c:82-86	move_particles	Single	0.26	0	0	1	0	6.25	1.17	1.38	16	0

Loop id	Source Location	Source Function	Level	Coverage run_0 (%)	Max Time Over Threads run_0 (s)	Time w.r.t. Wall Time run_0 (s)	Nb Threads run_0	Vectorization Ratio (%)	Vectorization Efficiency (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized	Speedup If Perfect Load Balancing run_0
9	icc_nbody3 - nbody3.c:60-73	main	Innermost	98.1	1.55	1.55	1	100	100	1	1	1	1

FIGURE 9 – Profile MAQAO des boucles de la version 3 avec GCC (au-dessus) et ICC (en-dessous)

Donc ce qu'on peut déjà faire, c'est d'ajouter le flag en question à la compilation vu qu'au niveau vectorisation il n'y a plus rien à faire apparemment.

7 Version 4

Cette version 4 a juste le makefile modifié avec le flag de compilation *-funroll-loops* en plus pour GCC comme MAQAO le dit et pour ICC par curiosité. La Figure 10 montre le résultats de cette version. Comme prédit par MAQAO, il y a une amélioration pour GCC. Le nombre de FLOP est 1.06 fois plus grand dans cette version, soit une augmentation de 6%. Dans le rapport de GCC, il y est indiqué que des boucles ont été déroulées avec des message du type "*optimized : loop unrolled 7 times*", d'où une meilleure performance. Par contre, pour ICC aucun changement, aucune réaction. Le flag sera enlevé dans la version suivante.

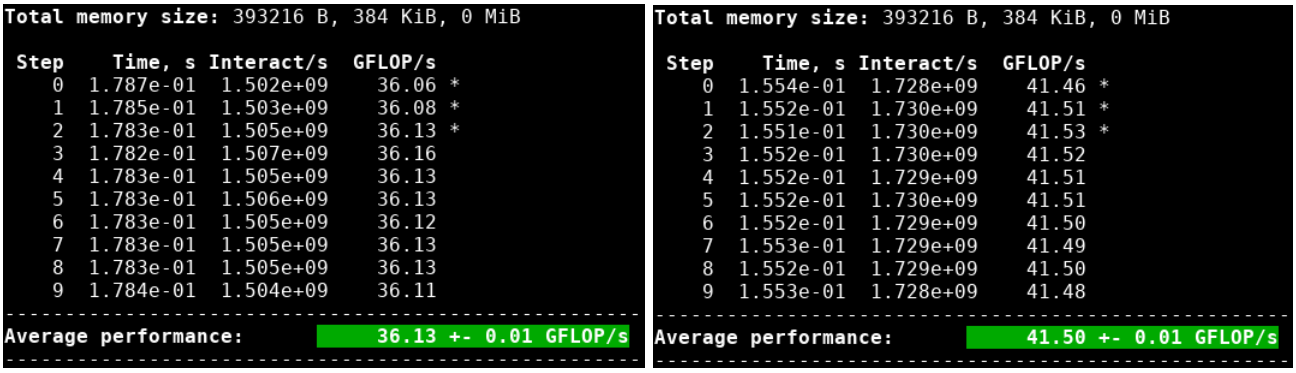

FIGURE 10 – Résultats de la version 4 avec GCC (à gauche) et ICC (à droite)

On peut résumer dans la Table 4 les mesures que nous donne perf stat pour les deux compilateurs.

perf stat	version 0	version 1	version 2	version 3	version 4
cycles	140,491,310,462	12,617,636,859	2,695,951,492	2,476,350,243	2,338,197,141
instructions	59,216,067,575	4,621,440,098	3,277,521,998	3,214,677,562	2,921,373,311
insn per cycle	0.42	0.37	1.22	1.30	1.25
user time (s)	108.6	9.751	2.083	1.913	1.806

TABLE 4 – Perf stat avec GCC des versions 0, 1, 2 ,3 et 4

Les rapports MAQAO de l'un et l'autre ne change pas par rapport à la version précédente, à part que les flags de compilation sont maintenant OK pour GCC. Vu que Perf et MAQAO ne nous indique plus comment évoluer le programme, j'ai essayé d'utiliser l'outil Advisor d'Intel par curiosité. Mais par manque de temps, je ne le mets pas dans le compte-rendu.
Donc, je suis sûr qu'il y a d'autre possibilité mais je ne vois pas lesquelles. Juste des tentatives hasardeuses avec tout ce qu'on peut trouver sur Internet peut permettre une amélioration supplémentaire. Par exemple ajouter ligne "*#pragma omp simd*" juste avant la première boucle dans *move_particles* permet une moyenne de 37.51 ± 0.01 GFLOPs pour GCC, soit une augmentation de 4%. Mais pour ICC, cela descend à 40.57 ± 0.01 GFLOPs , soit une réduction de 2%. En plus de ne pas vraiment savoir tout ce que cette ligne permet, elle réduit le maximum de GFLOP atteint. Donc elle ne nous sert pas.

8 Résumé

En conclusion, le nombre de FLOP entre la première version et la dernière version varie d'un facteur de 63.4 pour GCC et d'un facteur de 5.5 pour ICC. Malgré l'écart de base entre les deux compilateurs, les améliorations ont fait que GCC a pu se rapprocher des performances de ICC. Comme quoi, il faut mâcher tout le travail de GCC pour qu'il fonctionne bien. Enfin l'histogramme suivant (Figure 11) résume les données et l'évolution des performances de chaque version.

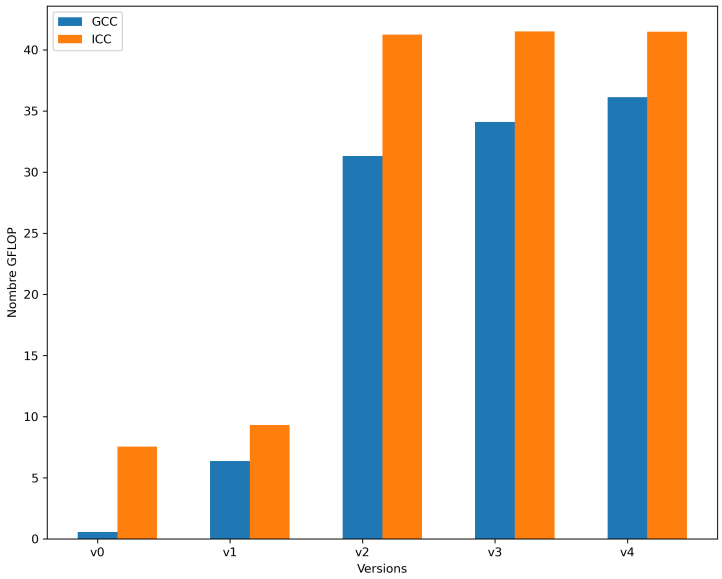

FIGURE 11 – Histogramme représentant le nombre de GFLOP en fonction de la version et du compilateur