M1 CHPS
Architecture Parallèle
Projet : Optimisation d'un code NBODY 3D

Joël BOUGRON
Dépôt : https://github.com/Joel-CHP/M1_CHPS_AP_NBODY3D
Université Paris Saclay

21 janvier 2023

Table des matières

1 Présentation

1.1 Objet

Il s'agit de comparer des performances pour plusieurs versions d'un programme NBODY 3D. Chaque version intègre les améliorations de la version précédente.

La latence mémoire (i.e. la durée d'accès aux données) est un paramètre prépondérant pour la vitesse d'exécution d'un programme. Pour accélérer l'exécution, il convient d'avoir une gestion fine des emplacements où lire et écrire les données. Les caches sont les emplacements les plus rapidement accessibles. Mais ils sont d'une taille réduite. Par conséquent, il faut sélectionner les données qui sont dans les caches. Le but principal étant d'éviter les "cache miss", i.e. les moments où lesaccès échouent car les données ne sont pas dans les caches.
Il existe différentes stratégies afin de réduire le nombre d'accès aux caches :

- Algorithmique de type "cache oblivious", en considérant l'existence de la mémoire cache dans les calculs de complexité.
- Code assembleur par utilisation d'instructions lisant ou écrivant dans la mémoire sans passer par le cache, ainsi que des instructions qui permettent de précharger une donnée dans le cache (instructions de prefetching).
- Choix des structures de donnée.

C'est cette dernière stratégie (entre autres techniques d'optimisation) que nous appliquerons dans la présente étude.

1.2 Présentation physique

On considère N corps qui interagissent gravitationnellement selon les lois de Newton. On ne considère pas de champ potentiel externe. Il s'agit d'une intégration numérique avec un pas de temps constant.

1.3 Présentation du programme

La version de base du programme permet avec la commande ./launch.sh d'exécuter le programme et créer des fichiers de sortie .csv dans le répertoire sortie/. Puis, dans le répertoire sortie/ : la commande ./sorties.sh permet de créer des fichiers image .png intégrés dans le présent rapport.
Une version longue permet avec la commande ./launch.sh d'afficher des détails dans le terminal (sans création de fichiers .csv). Le programme affiche alors les performances de chaque itération puis imprime les performances moyennes (avec l'erreur relative).
Les 3 premières itérations sont considérées comme un échauffement et ne sont pas utilisées pour calculer la performance moyenne et l'erreur.

1.4 Présentation de la machine

La machine utilisée dispose d'un processeur Intel Core i7-8565U.
Selon "https ://ark.intel.com/content/www/fr/fr/ark/products/ 149091/intel-core-i78565u-processor-8m-cache-up-to-4-60-ghz.html" elle est compatible avec les extensions du jeu d'instructions Intel :

— SSE4.1
— SSE4.2
— AVX2

1.5 V0 : version initiale

La version initiale présente une structure des données de type AoS (Array of Structure). Cette structure, dans certaines conditions, induit une latence (durée d'accès aux données) plus importante. Ainsi, quand on cherchera une donnée dans un cache et que la donnée sera absente ("cache miss"), il sera incrémenté vers le cache le plus proche une nouvelle recherche relativement coûteuse.
C'est ce qui est attendu dans notre cas particulier du fait de l'algorithme utilisé. En effet, selon l'ordre des boucles, le parcours de chacune des composantes de positions et de vitesses est prioritaire par rapport au parcours des N corps en interaction.

2 Optimisations

2.1 Méthodologie

Dans le cadre de cette étude, nous faisons varier :

— 2 nombres de corps (1000 et 10000).
— 3 compilateurs (GCC, Clang et ICX)
— 8 versions d'optimisation (V00 à V07).

— 3 options d'optimisation du compilateur (-O0, -O2, -Ofast).
Ce qui totalise 144 mesures de performances.

Quand nous compilons avec le flag -Ofast, il faut avoir à l'esprit que ce flag d'otimisation en induit d'autres telles que le déroulage de boucle, la vectorisation. C'est pourquoi nous les omettons tel que décrit ci-dessous. C'est l'ordre des versions d'optimisation qui prime pour comprendre le cheminement.

2.2 Présentation des versions

— V00 : Version initiale ;
— V01 : Version - SoA ;
— V02 : Version - SoA - Alignement ;
— V03 : Version - SoA - Alignement - Sans pow avec moins de divisions ;
— V04 : Version - SoA - Alignement - Sans pow avec moins de divisions - Sqrtf ;
— V05 : Version - SoA - Alignement - Sans pow avec moins de divisions - Sqrtf - Déroulage de boucle par le compilateur ;
— V06 : Version - Ajout de la vectorisation automatique par le compilateur ;
— V07 : Version - Intrinsics ;

2.3 VersionSoA

La structure de données est un facteur important afin de limiter les "cache miss". Dans le cas de SoA (Structure of Array), les variables de même type sont stockées de manière contigüe. Ainsi on regroupe 6 types de données :
— composante x des positions,
— composante y des positions,
— composante z des positions,
— composante vx des vitesses,
— composante vy des vitesses,
— composante vz des vitesses,

Par ailleurs cette organisation des données rend le code plus facilement vectorisable.

On constate une amélioration moyenne de l'ordre de 5% à 10%.

2.4 Alignement mémoire

Le SoA prend tout son intérêt quand on travaille en même temps à l'alignement mémoire.

Nous utilisions la fonction 'malloc' pour les allocations mémoire. Nous remplaçons

par la fonction 'aligned_alloc' qui permet d'allouer un emplacement en imposant une contrainte d'alignement fournie en argument.
Il s'agit de trouver les données dans les caches quand elles sont recherchées. Ainsi, on souhaite réduire le nombre d'accès mémoire en regroupant les données. Cela permet en particulier de stocker un plus grand nombre de valeurs dans les registres et donc accélère théoriquement les calculs.
Pour ce faire, on étudie les caractéristiques de notre machine et on définit l'allocation mémoire en conséquence. La commande $ cat /proc/cpuinfo nous indique : "cache_alignment : 64 bits".

On constate également une amélioration moyenne de l'ordre de 5% à 10%.

2.5 Suppression d'instructions coûteuses

2.5.1 Suppressions de pow et suppression de divisions

Nous effectuons les économies de cycles CPU suivantes, le tout étant équivalent mathématiquement selon le niveau de précision du programme :

- Les puissances 3/2 correspond à la multiplication de 3 racines 1/2 à calculer 3 fois, nous les remplaçons par 1 seul calcul en amont de puissance 1/2.
- Nous remplaçons les pow de puissance 3 (de racines 1/2) par une multiplication à 3 facteurs.
- Nous remplaçons 3 divisions (calcul de fx, fy, fz) par 1 division puis 3 multiplications de l'inverse (calcul de fx, fy, fz).

On constate une amélioration :

- Très forte (de 500% à 1000%) pour Clang.
- Forte (environ 50%) pour GCC.
- Forte (environ 50%) pour ICX.

2.5.2 Suppression de sqrt

Nous remplaçons le sqrt (précision : double, trop élevé par rapport à la précision du programme) par sqrtf (précision : float, suffisant pour le programme).

On constate une amélioration moyenne de l'ordre de 5% à 10%.

2.6 Loop unrolling

Pour les optimisations de compilation supérieures ou égales à -O2, le déroulage de boucle est activé.
Afin de désactiver le déroulage de boucle, dans les versions précédentes nous avions ajouté le flag "-fno-unroll-loops".
Pour activer le déroulage de boucle dans la présente version, nous retirons ce flag.

On constate une amélioration moyenne de l'ordre de 25% mais quasi-nulle dans certains cas.

2.7 Vectorization (SSE, AVX, or AVX512 for x86_64 architectures)

2.7.1 Compiler auto-vectorization

Pour les optimisations de compilation supérieures ou égales à -O3, la vectorisation est activée.
Afin de désactiver la vectorisation, dans les versions précédentes nous avions ajouté le flag "-fno-tree-vectorize".
Pour activer le déroulage de boucle dans la présente version, nous retirons ce flag et nous ajoutons le flag "-ftree-vectorize".

On constate une forte amélioration moyenne.

2.7.2 Intrinsics en AVX2

Nous intégrons des instrinsics, c'est-à-dire que nous convertissons une partie du code avec des instructions de code assembleur afin de le vectoriser manuellement. Ainsi, selon le principe SIMD (Single Instruction Multiple Data), quand c'est possible, on opère sur des vecteurs et non des scalaires, ce qui accélère le code.
Ces instructions à l'intérieur du code ne sont pas modifiées à la compilation.
Dans notre cas, les opérations scalaires (resp. vectorielles) sont de simple précision (float) dont présentent le suffixe par "SS" pour Scalar Single precision (resp PS pour Packed Single precision).
Ces jeux d'instructions peuvent être des familles présentées ci-dessous, objet d'évolutions dans le temps qui ont pour principe d'augmenter le nombre de registres disponibles ainsi que la capacité de chaque registre.

- SSE (Streaming SIMD Extension).
- AVX (Advanced Vector eXtension).
- AVX2 qui élargit la plupart des commandes SSE et AVX 128 bits en 256 bits, ce qui correspond également à la largeur des registres.
- AVX512.

Selon le processeur de notre machine, c'est AVX2 qui sera utilisé.
Dans notre cas, les opérations scalaires (resp. vectorielles) sont de simple précision (float) dont présentent le suffixe par "SS" pour Scalar Single precision (resp PS pour Packed Single precision).
Les tailles des données sont alors de 256 bits, soit 32 bytes, ce qui permet de charger 8 flottants x 4 bytes (tailles des variables x, y, z, vx, vy, vz du présent programme).
A la compilation, on utilise l'option -mavx2 : "these switches enable the use of instructions

in the [...] AVX2."

Nous recodons la fonction move_particles en intrinsics AVX2 avec application des principes suivants :

- _m256 : type de variable spécifique AVX2 alternatif pour les variables de la fonction.
- 8 affectations de valeurs à softening et dt car 8 flottants sont chargés.
- Increments de la boucle for externe par des multiples de 8.
- _mm256_setzero_ps : pour l'initialisation à zéro des composantes des forces (registres de flottants YMM).
- _mm256_loadu_ps : pour déplacements des valeurs dans des vecteurs.
- _mm256_sub_ps : pour les soustractions.
- _mm256_rsqrt_ps : pour les inverses des racines carrées de flottants avec une erreur relative maximale limitée à 1,5*2^-12.
- _mm256_fmadd_ps : pour multiplier 2 paramètres et stocker dans le 3ème paramètre (boucle for sur fx, fy, fz ; boucle for sur vx, vy, vz).
- _mm256_storeu_ps : pour stocker les 8 flottants (256 bits) (en 2nd paramètre) dans la mémoire (1er paramètre).
- #include immintrin.h : pour avoir à disposition les instructions ci-dessus.

On constate une forte amélioration moyenne.

2.7.3 Inline assembly

Le principe de l'inline assembly consiste à intégrer (selon les compilateurs) un code de bas niveau (langage assembleur) dans le code compilé à partir d'un langage de plus haut niveau (ici : C).
Il s'agit de rechercher les "points chauds" c'est-à-dire les plus sensibles algorithmiquement aux performances, quand on peut être plus efficace que ce qui pourrait être généré par le compilateur. L'intérêt principal est de réduire le nombre de jumps.

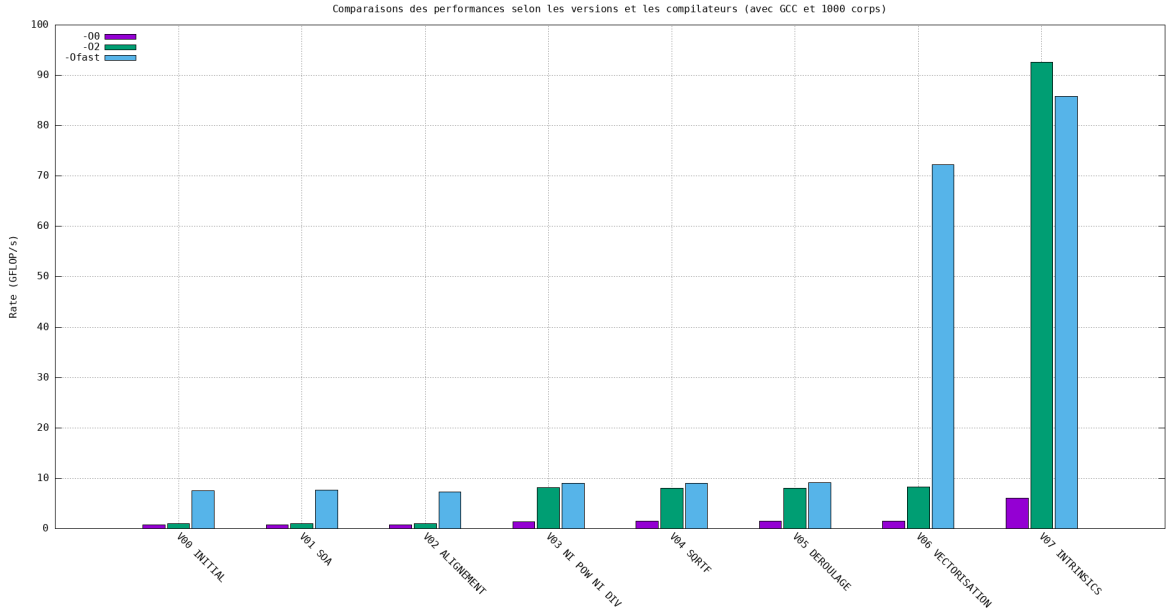
Ce sujet fera l'objet d'un développement ultérieur.

2.8 Bande passante mémoire

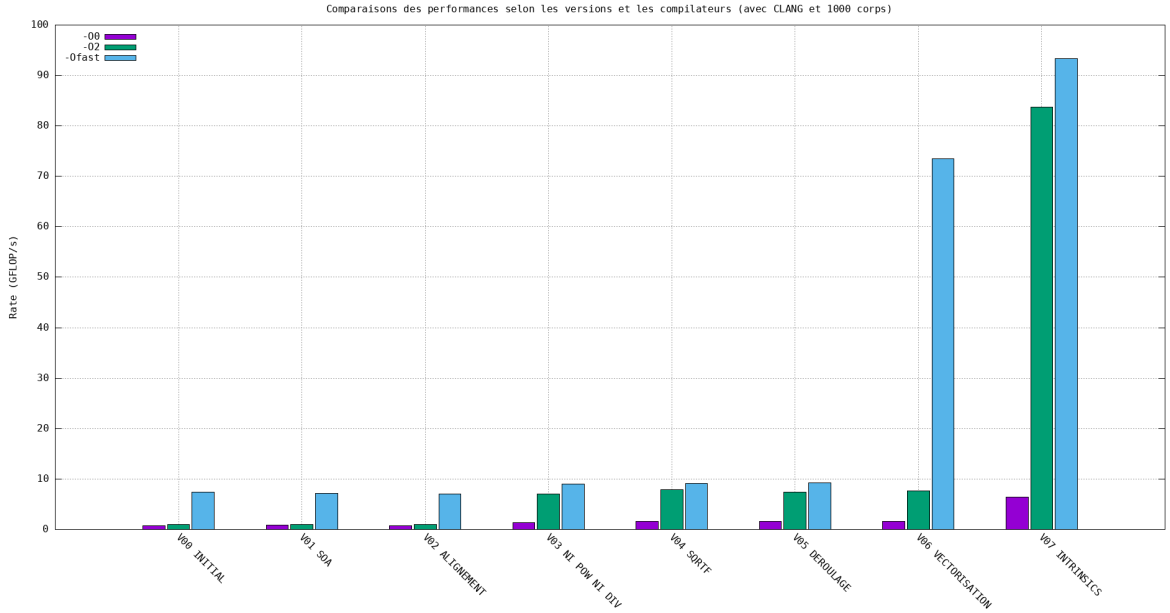
Nous avons ajouté une mesure de bandwidth (KiB/s), affichable dans la version longue.

3 Résultats

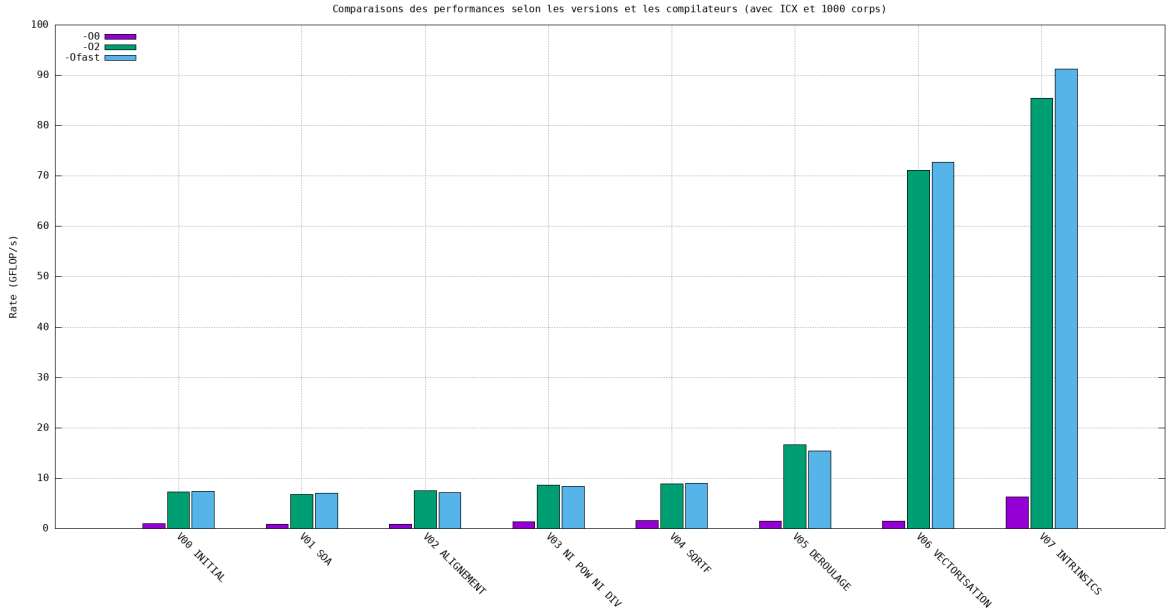
3.1 GCC et 1000 corps



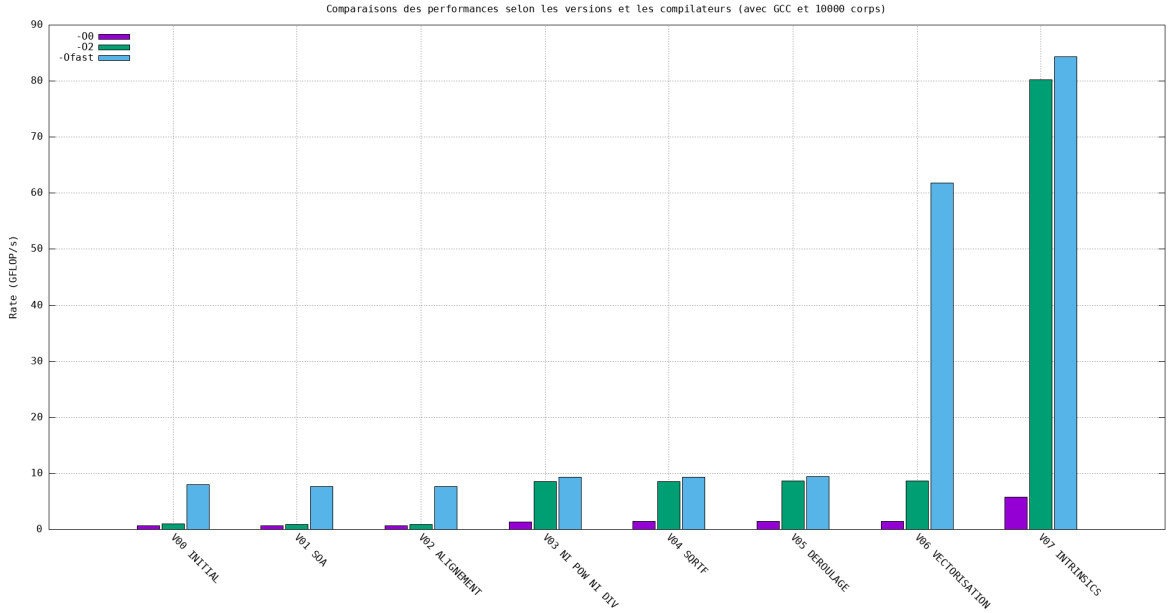
3.2 CLANG et 1000 corps



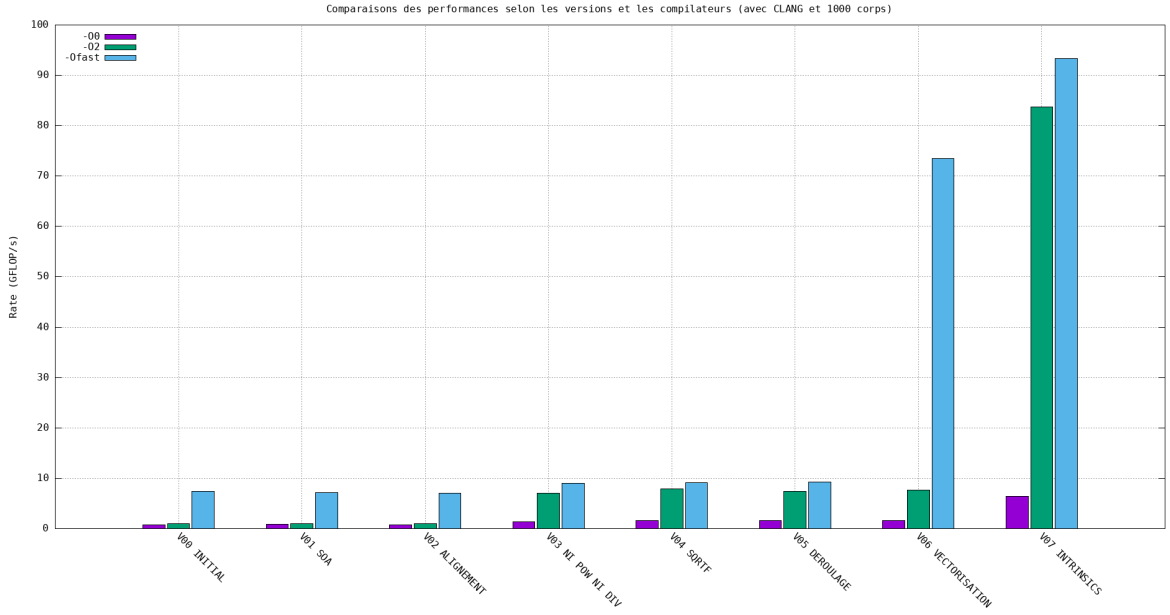
3.3 ICX et 1000 corps



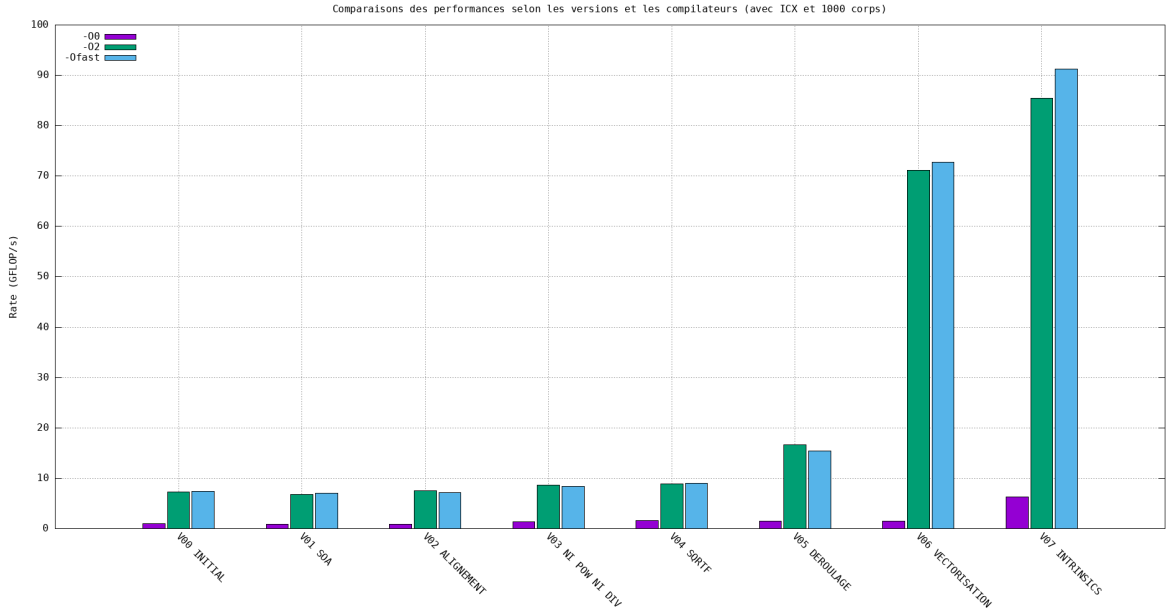
3.4 GCC et 10000 corps



3.5 CLANG et 10000 corps



3.6 ICX et 10000 corps



4 Objectifs ultérieurs

On pourrait prolonger l'étude et comparant les performances avec :

— Utilisation du jeu d'instructions de Intel AVX512. Les tailles des données seraient alors de 512 bits, soit 64 bytes, ce qui permettrait de charger 16 flottants x 4 bytes (tailles des variables x, y, z, vx, vy, vz du présent programme).
— Inline assembly.
— OpenMP directives.

5 Bibliographie

Dans le cadre de cette étude, nous avons consulté :

— https ://gcc.gnu.org/onlinedocs/gcc/ : pour le compilateur GCC (options x86, etc).
— https ://clang.llvm.org/docs/ : pour le compilateur Clang.
— https ://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
— https ://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html : pour l'inline assembly.
— https ://fr.wikipedia.org/wiki/Advanced_Vector_Extensions : concernant les généralités relatives aux jeux d'instructions.
— Zeste de savoir - Mémoire cache et optimisation de code