Architecture parallèle

Optimisation d'un code de simulation : n-corps

Auteurs :
M. Sébastien Dubois

Professeur :

22 Janvier 2022

Table des matières

1 Introduction 2
2 Problème aux n-corps 2
2.1 Algorithme de la simulation et discussions 2
3 Stratégies d'optimisation de code 3
3.1 Utilisation de fonctions mathématiques alternatives 4
3.2 Compilation et flags 5
3.3 Structure de données et allocation mémoire 6
3.4 Utilisation de vectorisation par paradigme SIMD 8
4 Résultats finaux 9
4.1 Problème aux n-corps 9
4.2 Calcul de Pearson 10

1 Introduction

Beaucoup d'ajouts d'études sont à venir très prochainement.

2 Problème aux n-corps

2.1 Algorithme de la simulation et discussions

La résolution du problème aux n-corps s'articule autour de l'algorithme suivant :

Algorithm 1: Algorithme naif de simulation des n-corps

for $t = [1 : t_{max}]$ **do**
  for $i = [1 : n]$ **do**
    `// s is the sum` $\sum_j \mathbf{F_{j\to i}}$
    $\mathbf{s} = \mathbf{0}$;
    for $j = [1 : n]$ **do**
      `// compute the contribution of n-1 particles in i-th particle`
      $\mathbf{F}_{j\to i} = \frac{Gm_j m_i}{d_{ji}^2}\mathbf{e}_{ji}$;
      $\mathbf{s} = \mathbf{s} + \mathbf{F}_{j\to i}$;
    end
    `// explicit integration scheme`
    $\mathbf{v}_i^t = \mathbf{v}_i^{t-1} + dt \times \frac{\mathbf{s}}{m_i}$;
    $\mathbf{x}_i^t = \mathbf{x}_i^{t-1} + dt \times \mathbf{v}_i^t$;
  end
end

Pour chaque pas de temps $\mathbf{t}$, il est nécessaire de calculer l'effort résultant du champ gravitationnel des $n-1$ corps sur le i-ème corps. Ainsi, la complexité de l'algorithme naïf suivant un seul pas de temps est de $\mathcal{C}^{\text{naif}} = \mathcal{O}(n^2)$.

Le calcul de l'effort $\mathbf{F}_{j\to i}$ nécessite cependant une transformation permettant d'éviter la normalisation du vecteur directeur $\frac{\mathbf{d}_{ji}}{|\mathbf{d}_{ji}|} = \mathbf{e}_{ji}$. Ainsi, on utilisera plutôt le formalisme suivant $\mathbf{F}_{j\to i} = \frac{Gm_j m_i}{d_{ji}^2}\mathbf{e}_{ji} = \frac{Gm_j m_i}{d_{ji}^3}\mathbf{d}_{ji}$. Il est de fait nécessaire d'effectuer une unique mesure de la distance $\mathbf{d}_{ji} = \mathbf{X}_j - \mathbf{X}_i$, et une unique mesure de racine (racine troisième) de $\mathbf{d}_{ji}$.

De plus, afin de simplifier les équations et le nombre d'opérations à effectuer, le programme propose - de façon simplement arbitraire - d'utiliser des masses et une constante gravitationelle unitaire suivant $m_i = 1\ \forall i \in [1, n]$ et $G = 1$. Ainsi, le programme passe d'autant plus de temps à effectuer des calculs complexes suivant la racine carré, la puissance troisième et l'inverse. Il est ainsi important d'accorder beaucoup d'attention à ces phases de calcul.

Il existe une optimisation algorithmique intéressante issue de la troisième loi de Newton suivant le principe d'action et réaction. Cela se traduit par $\mathbf{F}_{j\to i} = -\mathbf{F}_{i\to j}$. Il est alors possible en théorie de diviser par deux le nombre de calcul d'efforts. Cependant, cette stratégie nécessiterait d'effectuer l'intégration explicite de $\mathbf{v}_i^t$ et $\mathbf{v}_j^t$ dans la boucle j, ce qui ne permet plus d'exploiter l'intérêt de l'accumulateur $\mathbf{s}$. Il serait potentiellement envisageable d'exploiter une liste $\mathbf{s}$ de taille

n afin de garder l'avantage de l'accumulateur. l'algorithme à accumulateur modifié deviendrait alors :

Algorithm 2: Algorithme naif de simulation des n-corps avec accumulateur modifié

for $t = [1 : t_{max}]$ **do**
  `// s_i are the sum` $\sum_j \mathbf{F_{j \to i}}$
  $\mathbf{s}_i = \mathbf{0}\ \forall i \in [1, n]$;
  for $i = [1 : n]$ **do**
    for $j = [i + 1 : n]$ **do**
      `// compute the contribution of n-1 particles in i-th particle`
      $\mathbf{F}_{j \to i} = \frac{G m_j m_i}{d_{ji}^2} \mathbf{e}_{ji}$;
      $\mathbf{s}_i = \mathbf{s}_i + \mathbf{F}_{j \to i}$;
      $\mathbf{s}_j = \mathbf{s}_j - \mathbf{F}_{j \to i}$;
    end
    `// explicit integration scheme`
    $\mathbf{v}_i^t = \mathbf{v}_i^{t-1} + dt \times \frac{\mathbf{s}_i}{m_i}$;
    $\mathbf{x}_i^t = \mathbf{x}_i^{t-1} + dt \times \mathbf{v}_i^t$;
  end
end

Cependant, cet algorithme est complexe à optimiser, En effet, l'exploitation de la *vectorisation* est complexe du fait des accumulateurs $\mathbf{s}_j$.

Bien-sur, il existe des algorithmes optimisés tel que l'algorithme de Barnes-Hut *exploitant une structure en* ***octree*** *afin de rassembler le calcul de la distance d_{ji}. Cela permet notamment de réduire la complexité suivant $\mathcal{C}^{naif} = \mathcal{O}(n \log(n))$. Cependant, un tel schéma induit nécessairement de l'erreur et ne compose pas un algorithme exacte d'un problème aux n-corps. Il a alors été choisi de travailler uniquement sur la version naïve.*

3 Stratégies d'optimisation de code

Il est question d'optimiser l'exécution de ce code du point de vue du temps d'exécution. Il existe plusieurs pistes traditionnelles à envisager pour augmenter la vitesse d'exécution d'un programme :

- exploitation de `flags` de compilation permettant d'activer des routines d'amélioration du code ;
- modification de l'allocation mémoire et de mise en place des structures de données ;
- utilisation de fonctions mathématiques alternatives ;
- utilisation du principe de vectorisation - exécution de type `SIMD` ;
- utilisation du multiprocessing - à l'échelle du processeur ou de noeuds de calculs.

Chacune de ses pistes seront exploitées afin de comprendre l'effet de chacune de ces améliorations indépendamment des autres, afin de déterminer quel est le gain potentiel sur l'exemple concret des n-corps. Aussi, il est important de préciser que certaines des optimisations sont effectuées au déficit de la précision numérique. Ainsi, il sera aussi question de quantifier la précision numérique de ces méthodes afin, si nécessaire, d'être en mesure d'utiliser la variante la plus rapide pour

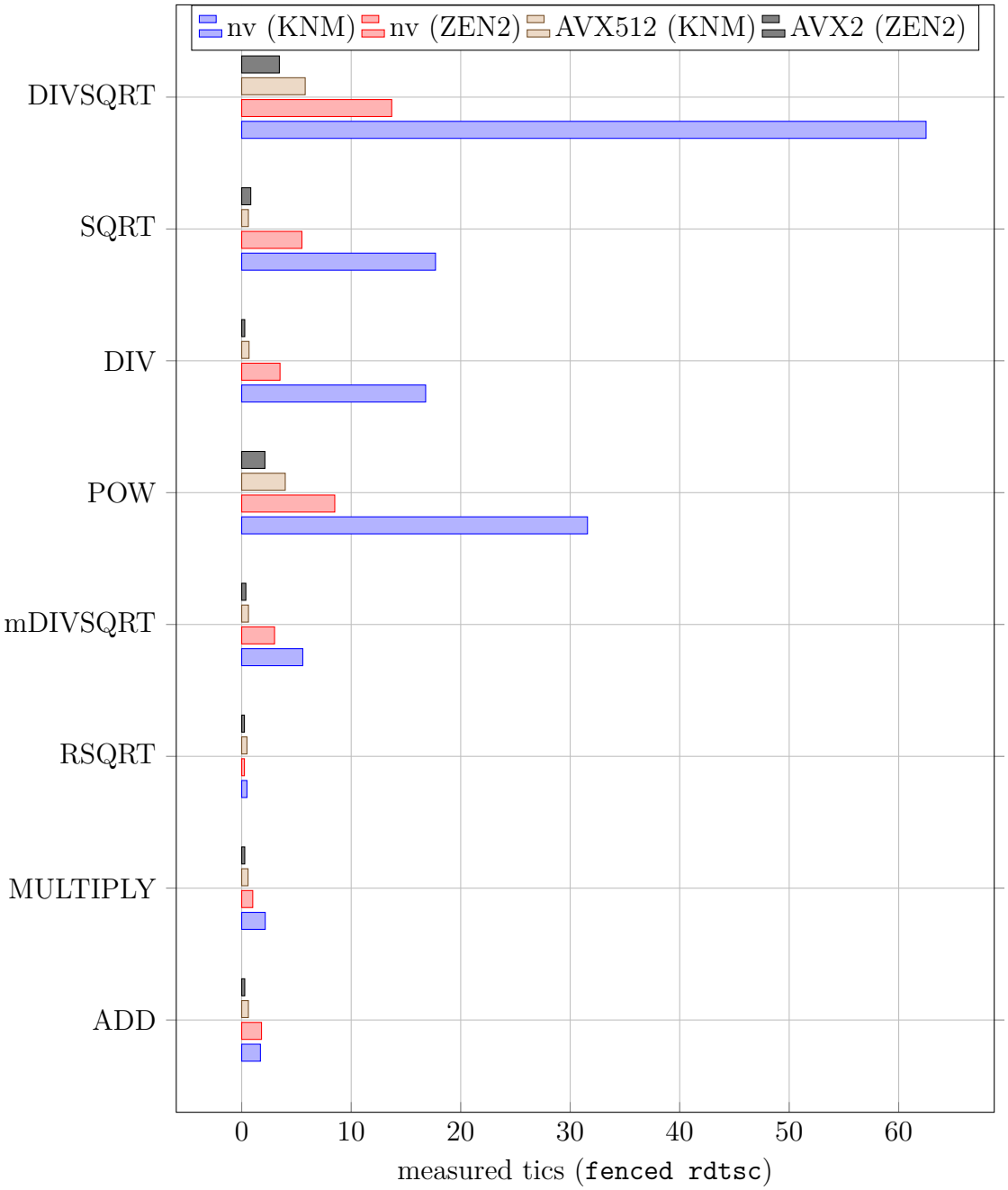
une précision donnée. Un code référence utilisant une précision `80 bits` par l'intermédiaire des instructions `x87` sera mis en place pour obtenir une solution de référence.

3.1 Utilisation de fonctions mathématiques alternatives

L'algorithme initial propose le calcul des termes $\frac{1}{\left(\sqrt{(x_j-x_i)^2+(y_j-y_i)^2+(z_j-z_i)^2}\right)^3}$ par l'utilisation de la fonction `pow` permettant de calculer selon des puissances non entières, et selon la fonction inverse `/`. Cependant, l'exécution de ces fonctions est lente et peut être optimisée. Voici les différents axes d'études permettant de sélectionner la meilleure solution :

- utilisation de la fonction `sqrt` suivie d'une double multiplication $\bullet \times \bullet \times \bullet$;
- utilisation de la fonction `sqrtf` dédiée au calcul sur flottant simple précision - `sqrt` étant normalement dédiée au calcul sur double précision ;
- calcul d'une série approximant $\sqrt{()}$ par des sommes et des produits ;
- calcul de l'inverse de la racine carré suivant $\frac{1}{\sqrt{()}}$ avec `rsqrtps`.

Les différentes stratégies sont implémentées afin de mesurer la précision et la rapidité d'exécution. Les mesures ont été effectués sur des `array` rentrants tous dans le cache `L1` des processeurs. Les mesures de temps ont été effectues avec un compteur de temps spécial de type `fenced rdtsc`. Ce compteur est vérifié comme étant précis sur plusieurs processeurs (cf. projet MPI) en bloquant l'exécution out-of-order. La mesure est effectuée selon plusieurs répétitions afin d'augmenter la précision de la mesure (en prenant soin de ne pas rencontrer de phénomène de *dead code élimination*. Voici les résultats obtenus sur le processeur knl et sur un processeur d'architecture ZEN2 :



Il est très clair que les fonctions `pow` et `div` suivie de `sqrt` sont très longues à l'exécution. En revanche, línstruction vectorisée `rsqrtps` est aussi rapide qu'une addition par exemple. Or, nous avons besoin non pas de la racine mais de l'inverse de la racine. Il est alors absolument souhaitable d'utiliser cette dernière instruction. D'après certains tests - source : internet - la fonction `rsqrtps` possède une erreur de l'ordre de $10^{-}1$. Nous utiliserons alors cette instruction pour la simulation.

Note : le temps d'exécution des fonctions `sqrt` *et* `pow` *dépendent de la valeur de l'argument !*

3.2 Compilation et flags

Il existe sur le marché un large panel de compilateurs sur l'architecture d'étude `x86` :

— `gcc`, qui est le compilateur `C` de la collection GCC. Il s'agit probablement du compilateur le plus populaire ;
— `clang`, qui utilise LLVM (Low Level Virtual Machine) conçue pour l'optimisation du code à la compilation (source : wikipedia). Il existe aussi `AOCC` et `icx` aussi basés sur LLVM ;
— `icc`, qui est le compilateur développé par Intel.

Les flags de compilation permettent d'activer ou non des routines d'optimisation lors de la compilation. Par exemple, sur `gcc`, il est possible de compiler sans optimisation - par argument `-O0`, ou avec de nombreuses optimisations - avec `-O3`. Voici un tableau récapitulatif des flags intéressants à utiliser dans le projet :

	`gcc`
optimisation pour la durée de compilation	`-O0`
optimisation pour la dure d'exécution	`-O3`
optimisation avec précision calcul mathématiques non assurée	`-Ofast`
optimisations spécifiques à l'architecture d'exécution	`-mtune=native`
optimisations spécifiques uniquement à l'architecture d'exécution	`-march=native`
inline functions	`-finline-functions`
déroullage de boucles	`-funroll-loops`
simplification de boucles	`-fpeel-loops`
Activation de la vectorisation automatique	`-ftree-vectorize`
Activation de la vectorisation uniquement sur boucle	`-ftree-loop-vectorize`
Activation de la vectorisation de block basique	`-ftree-slp-vectorize`
Utilisation de l'AVX2	`-mavx2`
Utilisation des instructions de type `fma`	`-mfma`
pas de test d'erreur de nombre `-errno`	`-no-math-errno`

Il est à noter que l'utilisation d'un flag peut en activer d'autres. Notamment, sous `gcc`, l'utilisation du flag `O3` active `-funroll-loops` et `-ftree-vectorize` et `Ofast` active `-ffast-math` et `-fno-math-errno`.

Il est intéressant de tester notamment la différence de temps d'exécution et de précision entre les différents `-On`, et avec ou sans les redondances de flags telles que `-funroll-loops`.

3.3 Structure de données et allocation mémoire

Le problème des n-corps tri-dimensionnel nécessite le stockage de la position, de la vitesse et de l'accélération selon les trois coordonnées pour chaque particule. Il y a alors 9 flottants par particule à stocker. La façon dont sont stockées et accédées ces données influence la possibilité de vectorisation et vitesse de lecture et écriture au sein de la mémoire. Il existe deux principaux paradigmes de stockage :

— Les stockages en `SOA` (Structure Of Array) consistant au stockage en blocks des variables d'un même type de données (exemple : stockage continu des coordonnées selon x pour l'ensemble des particules) ;
— Les stockages en `AOS` (Array Of Structure) consistant au stockage en blocks de structures (exemple : stockage continu des coordonnées selon x, y et z).

Il est possible, lorsque le choix de la `AOS` est faite, d'exploiter soit trois structures de trois coordonnées - ce qui a été utilisé) soit une unique structure comportant l'ensemble des neuf flottants. *A priori*, l'utilisation d'une structure de neuf flottants induit la plus faible performance des trois déclinaisons. Nous verrons plus tard l'effet du choix de la structure sur les calculs.

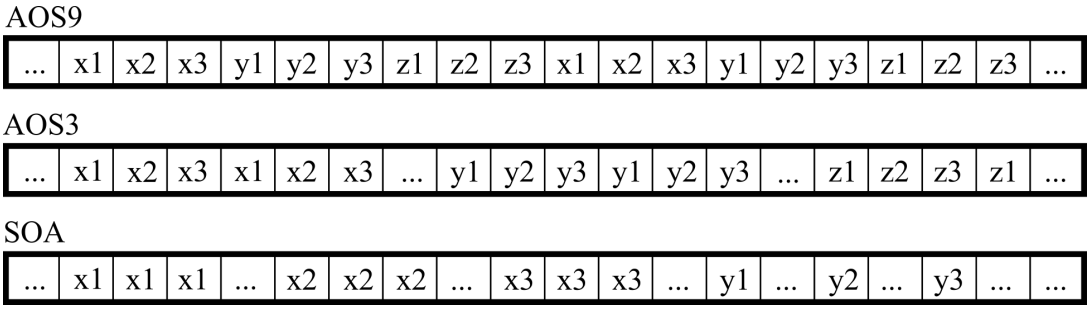


FIGURE 1 – Représentation des différentes structures de données

On distingue dans l'algorithme deux phases differentes de calcul :

— La phase selon les boucles imbriquées (i, j) de complexité $\mathcal{O}(n^2)$. Cette phase est la plus coûteuse lorsque le nombre de particules est grand. Il est laors important d'optimiser au mieux selon cette phase de calcul ;

— La phase d'intégration explicite de complexité $\mathcal{O}(n)$.

Or, dans la double boucle imbriquée, les données successivement accédées sont liées au calcul de la distance $\mathbf{d}_{ji} = \mathbf{X}_j - \mathbf{X}_i$ et au calcul de la racine carrée. On rappelle que $\mathbf{F}_{j\to i} = \frac{1}{\left(\sqrt{(x_j-x_i)^2+(y_j-y_i)^2+(z_j-z_i)^2}\right)^3}(X_j - X_i)$. Ainsi, une étape de calcul (i, j) nécessite l'accès aux neuf coordonnées de la particule. Or, une requête mémoire s'opère par *cache-line* entière de 64 bytes - soit 16 flottants simple précision simultanément. Ainsi, si le choix de la structure `AOS9` est faite, une seule requête de cache-line suffit pour charger dans les registres toutes les coordonnées de 6 particules différentes. Cependant, si la structure `SOA` est utilisée ; il faudra charger 3 cache-lines différentes pour une itération. Il y a ainsi une différence de *localité temporelle* et de *localité spatiale* à l'avantage de la méthode `AOS9`. Pour le vérifier, on propose d'effectuer des mesures de temps sur les deux codes, en utilisant ou non les flags d'optimisation précédemment explicités : Cependant, on prendra garde de désactiver la vectorisation. Voici les mesures de temps moyens selon les deux déclinaisons :

	`AOS9`	`SOA`
time on Ryzen (s)	8.55×10^{-1}	9.42×10^{-1}
time on KNL (s)	9.41	9.358

FIGURE 2 – Mesure de performance selon les flags `gcc -march=native -funroll-loops -ffast-math -finline-functions -O3 -fno-tree-vectorize -mno-avx2 nbody_.c -lp`

Ainsi, on observe expérimentalement que , sans vectorisation, la version en *Array Of Structure* est plus rapide. Cependant, le paradigme de vectorisation est difficilement applicable dans le cas présent avec une structure de données de type `AOS9`. En effet, la vectorisation - paradigme de type SIMD - permet d'effectuer sur plusieurs lots données continus des instructions identiques. Or, il est clair qu'en l'état, les instructions propres à un couple (i, j) nécessitent d'opérer des

instructions dépendantes des coordonnées - et donc dépendantes du lot de données. En revanche, dans le cas de l'utilisation dstructure de type `SOA`, les valeurs chargées sont de mêmes natures - et sont donc directement compatibles avec le paradigme `SIMD`. Ainsi, il est fort probable que la structure `SOA` prend son sens lorsqu'est utilisé la vectorisation.

3.4 Utilisation de vectorisation par paradigme SIMD

Le paradigme SIMD consiste en l'exécution d'une unique instruction sur une séquence de données. Dans le processeur, cette séquence de données est continue dans l'adressage mémoire - C'est d'ailleurs la raison pour laquelle la structure AOS ne se vectorise pas bien facilement dans ce cas -. En fait, lorsque des instructions élémentaires concernant des données flottantes sont exécutées, c'est la plupart du temps soit dans les unités d'exécution `x87` - permettant une meilleure précision - soit par l'intermédiaire d'autres unités. Ces unités sont composées d'ajout au fil du temps, en commençant par les unités `SSE`, `AVX2` puis `AVX512`. Ainsi, les instructions arithmétiques sont la plupart du temps exécutées au sein de ces unités. L'avantage de ces unités est qu'elles peuvent être exploitées selon le paradigme SISD (sur un scalaire) ou SIMD (sur un vecteur) sans latence supplémentaire. Par exemple dans le cas de donnes flottantes simples précisions, il est possible d'opérer en AVX512 16 opérations selon la même instruction. Ainsi, il est possible d'effectuer plusieurs opérations scalaires équivalentes en un seul tic d'horloge.

C'est précisément l'utilisation de ces opérations SIMD que nous allons exploiter afin d'accélérer l'exécution de la simulation. Il existe plusieurs façons d'obtenir un code exécutant des instructions de type AVX :

- par compilation, en ajoutant les flags explicités précédemment. Il est parfois difficile de forcer l'utilisation d'instructions vectorisées par compilation - notamment sans le cas de la racine carrée ;
- par code assembleur pur, en écrivant manuellement l'accès aux registres et aux instructions. C'est une méthode très fastidieuse et pratiquement pas utilisée ;
- par code intrinsic - qui génèrent des appels a du code assembleur pur. Cette stratégie permet une implémentation aisée de l'AVX - mais pas évidente et sujette a erreurs d'implémentations.

Les différents codes développées ont utilises des stratégies de vectorisation par compilation, et par utilisation de fonctions intrinsics, mais pas d'assembleur pur.

Voici un exemple de code implémenté en C pur et en intrinsic :

```
for (int j = 0 ; j < repetition ; j++)
    {
  for (int i = 0 ; i < n ; i++){
    sum += v_x[i] * v_y[i] ;
  }
}
```

Listing 1 – `fma` sur deux arrays en C pur

```
__m256 rx, ry,rp, rs, rx2, ry2, rp2,
    rs2;
for (int j = 0 ; j < repetition ; j++)
    {
rs = _mm256_setzero_ps() ;
rs2 = _mm256_setzero_ps() ;
  for ( int i = 0 ; i < n ; i+=16){
    rx = _mm256_load_ps(&v_x_al[i]);
    ry = _mm256_load_ps(&v_y_al[i] );
    rp = _mm256_mul_ps(rx, ry) ;
    rs = _mm256_add_ps(rp, rs);

    rx2 = _mm256_load_ps(&v_x_al[i+8])
    ;
    ry2 = _mm256_load_ps(&v_y_al[i+8]
    );
    rp2 = _mm256_mul_ps(rx2, ry2) ;
    rs2 = _mm256_add_ps(rp2, rs2);
  }
  _mm256_store_ps(&result[0], rs) ;
  _mm256_store_ps(&result[8], rs2) ;

}
```

Listing 2 – `fma` sur deux arrays en intrinsic

L'écriture est plus fastidieuse et nécessite beaucoup de prudence. Ainsi, il est absolument nécessaire de valider le code en mesurant la précision du calcul par rapport à une solution de référence (obtenue par flag `precise` par exemple sur `icc`.

4 Résultats finaux

4.1 Problème aux n-corps

Plusieurs variantes ont été développées :
- deux versions en `AOS` suivant :
  - AOS1 est la version basique ;
  - AOS2 est la version optimisee.
- 5 versions en SOA suivant :
  - SOA1 est la version SOA sans optimisation et sans vectorisation ;
  - SOA2 est la version non optimisée mais vectorisee par compilation ;
  - SOA3 est la version optimisée et vectrisee par compilation ;
  - SOA4 est la version en intrinsics en AVX2 ;
  - SOA5 est la version en intrinsic en AVX512.

Il a donc été possible d'augmenter la performance par un facteur de $g = 64$ - au profil d'une légère perte de précision.

Bien-sûr, l'utilisation des fonctions intrinsics augmente le risque probable d'effectuer des erreurs d'implémentation. Il est donc nécessaire de vérifier et de valider le résultat. n propose

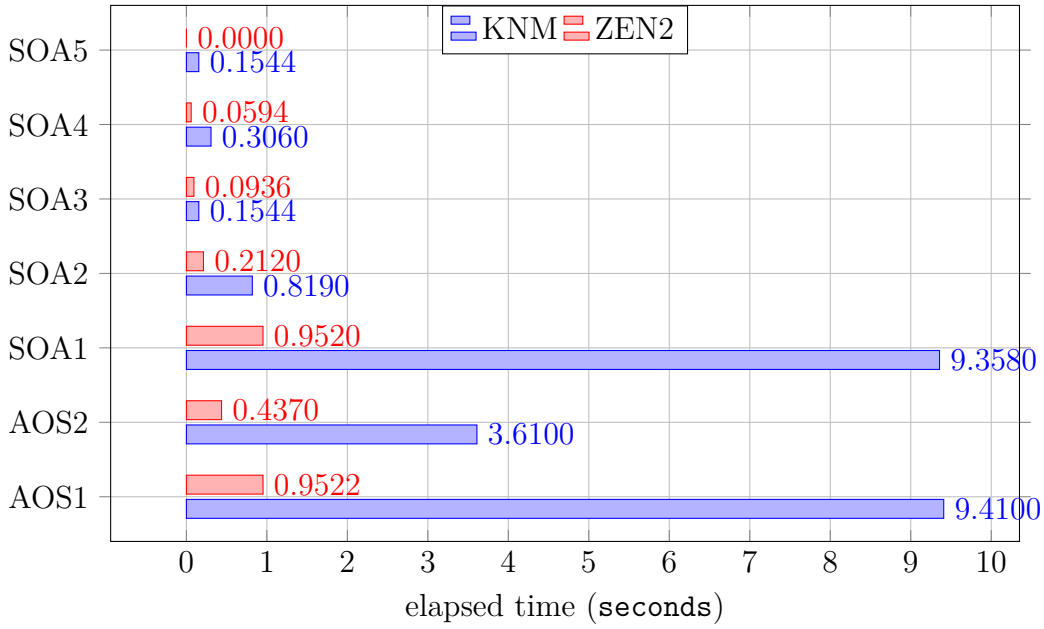


FIGURE 3 – Temps moyen d'exécution d'une itération du problème des n-corps sur deux architectures différentes

de vérifier l'implementation en mesurant la trajectoire de la première particule du tableau. Bien-sur, afin d'observer une erreur éloquente vis-a-vis de la solution de référence, il est nécessaire de mettre la vitesse initialement nulle. Ainsi, le déplacement et la vitesse ne dépend que de l'implémentation du calcul des efforts et non pas de l'état de la vitesse initiale. Le point difficile dans l'implementation en intrinsic est le respect des calculs lies aux premières et dernières particules . En effet, puisque la lecture des positions selon la boucle `i` doit se faire selon toutes les combinaisons glissantes de lecture continue, il est nécessaire de rajouter un offset avant et après les valeurs de position des particules. Il est aussi possible de s'affranchir de cette difficulté en utilisant plutôt une stratégie utilisant le registre de positions selon `i` composé de 16 valeurs identiques. Cependant, cela nécessite une somme horizontale `hadd` supplémentaire.

4.2 Calcul de Pearson

Voici les résultats du nombre de cycles nécessaires au calcul de l'équation de PEARSON suivant des versions en AOS, SOA, intrinsic, puis leurs versions déroulées manuellement :

Et voici l'implémentation des deux variantes (C pur et intrinsic) :

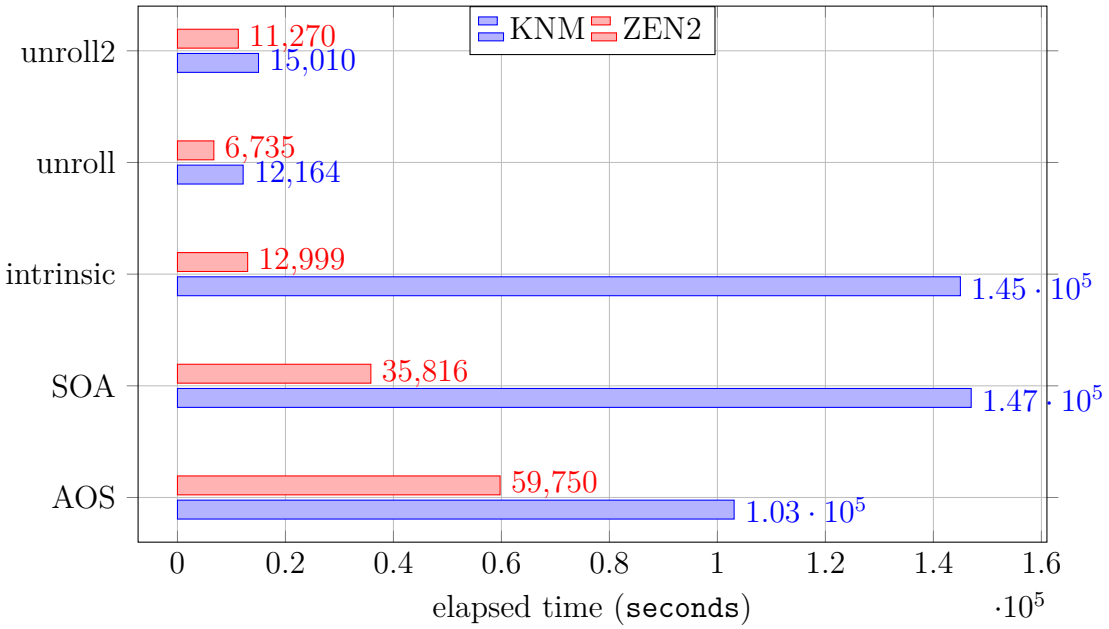


FIGURE 4 – nombre de cycles moyens d'exécution du calcul de PEARSON

```
rho = 0.0 ;
sum_x = 0.0 ;
sum_y = 0.0 ;
sum_x_2 = 0.0 ;
sum_y_2 = 0.0 ;
sum_xy = 0.0 ;
num = 0.0 ;
den = 0.0 ;
for (int i = 0 ; i < n ; i++){
  sum_xy += v_x[i] * v_y[i] ;
  sum_x += v_x[i] ;
  sum_x_2 += v_x[i] * v_x[i] ;
  sum_y += v_y[i] ;
  sum_y_2 += v_y[i] * v_y[i] ;
}
num = n * sum_xy - sum_x * sum_y ;
den = sqrt(n * sum_x_2 - sum_x * sum_x
    ) * sqrt(n * sum_y_2 - sum_y *
    sum_y) ;
rho = num / den  ;
```

Listing 3 – Calcul de Pearson en C pur

```
rho = 0.0 ;  sum_x = 0.0 ; sum_y = 0.0
    ; sum_x_2 = 0.0 ; sum_y_2 = 0.0 ;
   sum_xy = 0.0 ; num = 0.0 ;den = 0.0
    ;
sx = _mm256_setzero_pd() ;
sy = _mm256_setzero_pd() ;
sxy = _mm256_setzero_pd() ;
sx2 = _mm256_setzero_pd() ;
sy2 = _mm256_setzero_pd() ;
sxu = _mm256_setzero_pd() ;
syu = _mm256_setzero_pd() ;
sxyu = _mm256_setzero_pd() ;
sx2u = _mm256_setzero_pd() ;
sy2u = _mm256_setzero_pd() ;
for (int i = 0 ; i < n ; i+=8){
  rx = _mm256_load_pd(&v_x_al[i]);
  ry = _mm256_load_pd(&v_y_al[i]);
  rx2 = _mm256_mul_pd(rx, rx) ;
  ry2 = _mm256_mul_pd(ry, ry) ;
  rxy = _mm256_mul_pd(rx, ry) ;
  sx = _mm256_add_pd(sx, rx);
  sy = _mm256_add_pd(sy, ry);
  sx2 = _mm256_add_pd(sx2, rx2);
  sy2 = _mm256_add_pd(sy2, ry2);
  sxy = _mm256_add_pd(sxy, rxy);
  rxu = _mm256_load_pd(&v_x_al[i+4]);
  ryu = _mm256_load_pd(&v_y_al[i+4]);
  rx2u = _mm256_mul_pd(rxu, rxu) ;
  ry2u = _mm256_mul_pd(ryu, ryu) ;
  rxyu = _mm256_mul_pd(rxu, ryu) ;
  sxu = _mm256_add_pd(sxu, rxu);
  syu = _mm256_add_pd(syu, ryu);
  sx2u = _mm256_add_pd(sx2u, rx2u);
  sy2u = _mm256_add_pd(sy2u, ry2u);
  sxyu = _mm256_add_pd(sxyu, rxyu);
}
_mm256_store_pd(&asx[0], sx) ;
_mm256_store_pd(&asy[0], sy) ;
_mm256_store_pd(&asxy[0], sxy) ;
_mm256_store_pd(&asx2[0], sx2) ;
_mm256_store_pd(&asy2[0], sy2) ;

_mm256_store_pd(&asx[4], sxu) ;
_mm256_store_pd(&asy[4], syu) ;
_mm256_store_pd(&asxy[4], sxyu) ;
_mm256_store_pd(&asx2[4], sx2u) ;
_mm256_store_pd(&asy2[4], sy2u) ;
for (int i = 0 ; i < 8 ; i++){
  sum_x += asx[i] ;
  sum_y += asy[i] ;
  sum_xy += asxy[i] ;
  sum_x_2 += asx2[i] ;
  sum_y_2 += asy2[i] ;
}
num = n * sum_xy - sum_x * sum_y ;
den = sqrt(n * sum_x_2 - sum_x * sum_x
    ) * sqrt(n * sum_y_2 - sum_y *
    sum_y) ;
rho = num / den  ;
```

Listing 4 – Calcul de Pearson en intrinsic déroulé