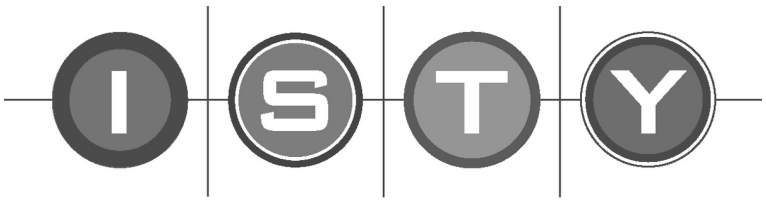


INSTITUT DES SCIENCES ET TECHNIQUES DES YVELINES

Projet AOA - Sujet 13

Hugo Henrotte
Olivier Benaben
Maxime Vincent
Maxence Bringuier

11 avril 2021 ⋈ LaTeX

Table des matières

1 Introduction

Nous avons travaillé sur le sujet 13.

Le projet est séparé en 2 phases majeures :

- une phase d'optimisation statique ; nous ne modifions pas le code et cherchons à l'optimiser lors de la compilation
- une phase d'optimisation dynamique ; ici nous optimisons le code avant de le compiler

Nous cherchons à obtenir les meilleures performances possibles en modifiant différents paramètres.
Pour ce faire nous allons donc jouer sur les paramètres suivants :

- compilation sous différents compilateurs tels que gcc, icx ou icc
- choix des flags judicieux en fonction de notre code
- choix des warmups et répétitions
- choix de la taille des tableaux en fonction de différents caches et RAM.

Fonction d'étude

```
void s13 (unsigned n, const float a[n], const float b[n],
   float c[n][n], int offset, double radius) {
   int i, j;

   for ( i = 0; i < n ; i ++) {
     for ( j = offset; j < n; j ++) {

          if ( a[ j ] < radius ) {
            c [ i ][ j ] = a [ j ] / b [ i ];
          }
          else {
              c [ i ][ j ] = 0.0;
          }

       }
   }
}
```

Nous nous sommes répartis les analyses comme suit :

- le cache L1 sera analysé par Olivier Benaben
- le cache L2 sera analysé par Hugo Henrotte
- le cache L3 sera analysé par Maxence Bringuier
- la mémoire RAM sera analysée par Maxime Vincent

2 Méthodologie

2.1 Justification des 5 points clés du driver (warmups, repets etc.)

Le driver tente de résoudre des problèmes de stabilité du système qui pourraient fausser les mesures de performance du kernel.

Il prend donc différentes mesures pour résoudre ces problèmes :

Répétitions de "warmup"

Ce sont des répétitions qui viennent en amont des mesures afin de préparer le système.

Ces warmups sont des passages "à vide" du code que l'on veut benchmarker. Cela permet de remplir les caches avec les valeurs des tableaux de données sur lesquelles le kernel fait ses opérations.
Cela permet de "chauffer" la machine pour effectuer les tests dans des conditions les plus stables possibles.
On verra l'utilité des warmups dans les figures suivantes, qui montrent que la première itération du kernel s'effectue dans un laps de temps beaucoup plus grand que les suivantes.
Elles permettent d'exclure le régime transitoire. Elles sont d'autant plus importantes lorsque la machine est froide.

Répétitions de l'expérience

Cette mesure prise par le driver pour réduire les erreurs de mesures est simple à comprendre : on effectue plusieurs fois les mesures pour en faire la médiane et atténuer les éventuelles perturbations qui pourraient fausser les mesures.

Méta-répétitions pour mesurer la stabilité

Comme le point ci-dessus, cette répétition va exécuter N fois tout le processus de test : les warmups + les répétitions Cela permet encore une fois d'obtenir :

- Répétitions du corps du driver
- Via un script ou une boucle dans le driver
- En toute rigueur, 31 méta-répétitions nécessaires
- Souhaité : $(médiane - minimum)/minimum < 5\%$
  on prendre NB_META = 31

Sonde utilisée

On utilisera la sonde `RDTSC` pour les mesures de temps pour sa précision.

Valeurs des tableaux

Les valeurs des tableaux sur lesquels portent les opérations du kernel sont randomisées à chaque méta-répétition. Les valeurs sont pseudo aléatoires et permettent donc de comparer fidèlement les exécutions (sans risquer que des optimisations du processeurs interfèrent sur l'effectivité des instructions).
Cela permet également d'éviter des exceptions de flottants.

2.2 Stratégies utilisées en commun pour améliorer la stabilité

Optimisation pré-exécution

Afin d'optimiser notre compilation nous faisons en sorte qu'elle soit le moins perturbée possible par des éléments extérieurs :

- *cpupower -c $CORE_ID frequency-set –governor performance*
  Cette commande permet de définir le coeur sur lequel se fera la compilation
- *taskset*
  Cette commande fait en sorte qu'aucun élément étranger ne viennent s'exécuter sur le coeur en question lors de la compilation de notre code
- Brancher l'ordinateur au secteur afin que le processeur ne bride pas les calculs pour économiser de la batterie
- avoir le moins de processus lancé sur la machine. il est même recommandé de lancer la machine sans interface graphique afin d'obtenir les meilleurs résultats

choix des flags

-ffast-math

Sous gcc un flag intéressant est $-mrecip$. En effet celui-ci permet d'utiliser les instructions RCPSS et RSQRTSS (et leurs variantes vectorisées RCCPS et RSQRTPS). ces instructions permettent nottamment d'augmenter la précision des calculs par rapport aux instrucstions DIVSS et SQRTSS. Des options a ce flags sont également disponible afin d'approximer certains calculs tels que la division avec $-mrecip = div$ ou encore $-mrecip = vec - div$.

2.3 Méthodologie de détermination du nombre de répétitions de warmup et de mesure

2.3.1 warmups

Pour déterminer le nombre de warmup nous avons utilisé la fonction *rdtsc*() afin de connaître le temps d'exécution des warmups. Nous avons donc supprimé les warmups dans notre code pour savoir à partir de combien de répétitions la machine devenait stable. Nous avons donc définit le nombre de répétitions à 100 avec 31 méta répétitions pour avoir un échantillon d'étude correct.

A partir de là, il nous sera possible d'étudier la médiane des valeurs de ces méta-répétitions afin de trouver le nombre de warmup nécessaire à la machine.

Méthode de calcul des warmups

```
for (i=0; i<repm; i++)
{
    uint64_t t1 = rdtsc();
    s13 (size, a, b, c, offset, radius);
    uint64_t t2 = rdtsc();
    rep[m][i]=(t2 - t1);
}
```

Une courbe de ces résultats (# repet / Répétition), devrait montrer une courbe décroissante avec une asymptote minorant le nombre de cycles par itération.

On prendra alors *repw* tel que la suite de la courbe soit stabilisée (*ie.* le régime transitoire est passé).

2.3.2 répétitions

Pour fixer un nombre de répétitions, on fera tourner le programme en mesurant à chaque méta-répétition le nombre de cycles ; on cherche à atteindre 3000 cyles par itération afin que la compilation dure quelques secondes. Cela nous permettra de l'étudier dans le futur avec *maqao*.

Re-exécuter l'expérience permet d'amortir l'erreur du timer (rdtsc).

Méthode de calcul des répétitions

```
uint64_t t1 = rdtsc();
for (i=0; i<repm; i++){
        s13 (size, a, b, c, offset, radius);
    }
uint64_t t2 = rdtsc();
```

2.3.3 méta-répétitions

On répétera l'expérience un certain nombre de fois pour améliorer la stabilité des mesures. On répétera donc le driver 31 fois pour des raison statistiques :

$$\frac{médiane - minimum}{minimum} < 5\%$$

2.3.4 Détermination de la taille des tableaux

Nous devons déterminer la taille des tableaux tels que les valeurs traitées tiennent dans le cache souhaité. Nous traitons 3 tableaux, 2 de taille n et 1 de taille n^2.

De ce fait il faut que

$$2n + n^2 < cache\ ciblé$$

.

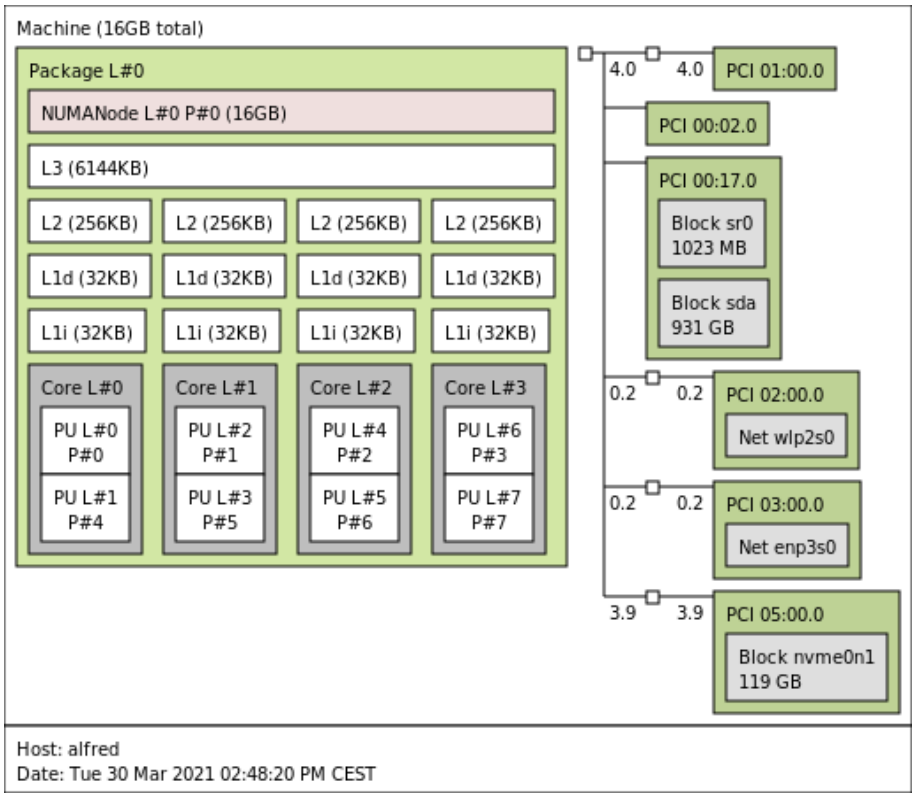
2.4 Cache L1 (Olivier BENABEN)

2.4.1 Environnement expérimental

Les mesures pour l'optimisation de notre kernel sur le L1 ont été effectuées sur la machine suivante :

Type	**Informations**
processeur	Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
Génération du processeur	Skylake
Linux	Pop!_OS 20.10 x86_64
Virtualisation	Simple
Version noyau	Linux version 5.8.0-7642-generic
Version GCC	gcc 10.2.0
Version oneAPI	Intel(R) oneAPI DPC++ Compiler 2021.1 (2020.10.0.1113)
Version MAQAO	2.13.0

Voici les informations des différents caches de la machine.



2.4.2 Justification de la taille des tableaux pour que ça tienne dans le cache

Nous devons déterminer la taille des tableaux tels que les valeurs traitées tiennent dans le cache souhaité. Nous traitons 3 tableaux : 2 de taille n et 1 de taille n^2.

De ce fait il faut que

$$2n + n^2 < sizeof(L1)$$

.

Mon L1 fait 32 kB. Nous allons donc faire en sorte de le remplir à 80%.
On note

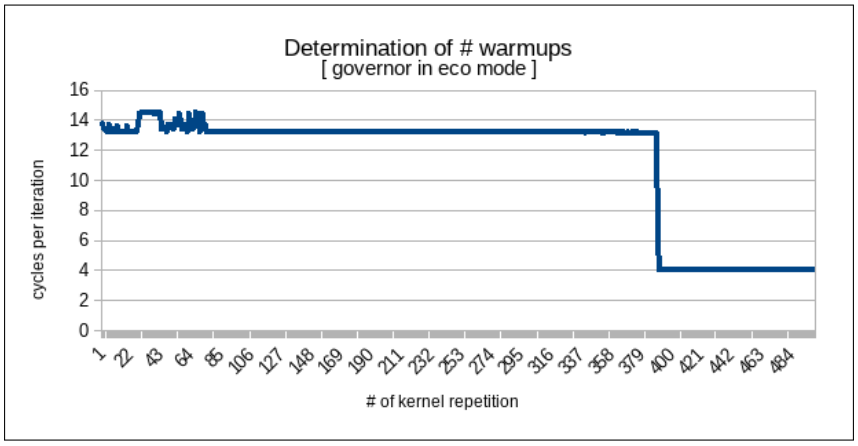
$$L1 = \frac{0,8 * sizeof(L1)}{sizeof(float)}$$

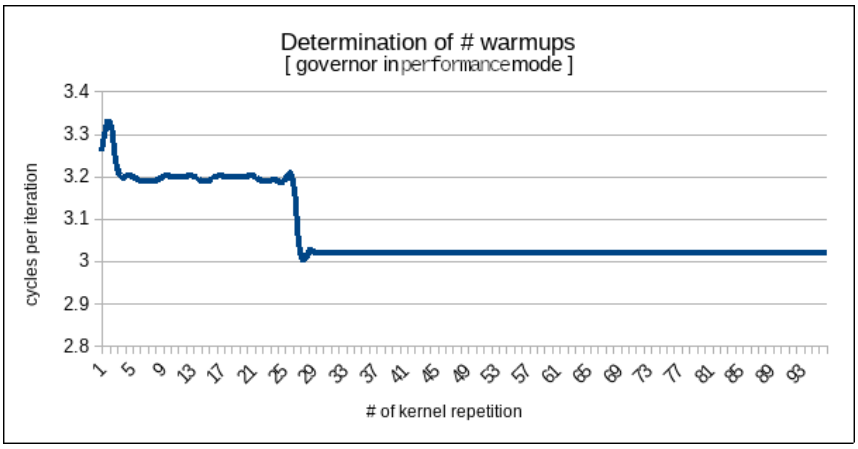
On cherche donc n tel que $2n + n^2 = L1$.
On trouve donc $n = \sqrt{2801} - 1$
Ainsi on prendra : $n = 51$

2.4.3 Détermination du nombre de répétitions de warmups et de mesures

Warmups

En suivant la méthodologie décrite précédemment pour la détermination du nombre de warmups, on construit le graphique suivant (`governor eco`, puis `performance`) :





Les deux graphes présentent l'evolution des médianes des des cycles/itération de chaque méta-répétition en fonction de leur répétition.

En `governor eco`, le nombre de warmups requis pour une expérience stable est donc d'environ 410 répétitions, et seulement 30 pour le `governor performance`.

Par la suite nous utiliserons le `governor performance`.

Répétitions

En suivant la méthode : on cherche ici a trouver un nombre de cycles/itération stable proche de 3000. On trouve empiriquement que 750 répétitions permettent de respecter cette règle.

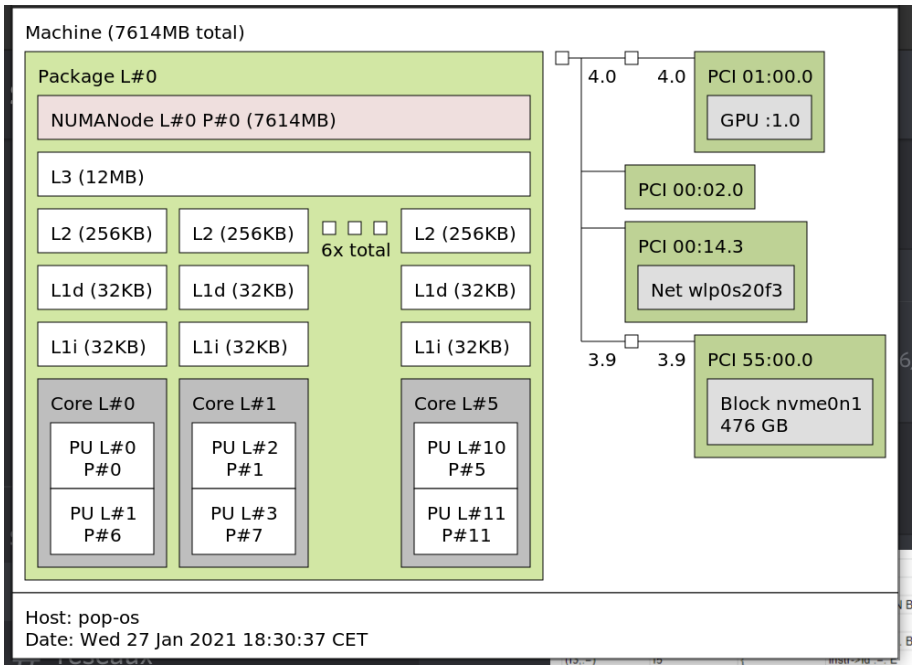
2.5 Cache L2

2.5.1 Environnement expérimental

Toutes les mesures du ont été effectuées sur la machine suivante :

Type	Informations
processeur	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
Génération du processeur	Comet Lake
Linux	Pop!_OS 20.10 x86_64
Virtualisation	Simple
Version noyau	Linux version 5.8.0-7642-generic
Version GCC	gcc 10.2.0
Version oneAPI	Intel(R) oneAPI DPC++ Compiler 2021.1 (2020.10.0.1113)
Version MAQAO	2.13.0
Techno RAM	2x8 Go DDR4 3200MHz

Voici les informations des différents caches de la machine.



2.5.2 Justification de la taille des tableaux pour que ça tienne dans le cache L2

Nous devons déterminer la taille des tableaux tels que les valeurs traitées tiennent dans le cache souhaité. Nous traitons 3 tableaux, 2 de taille n et 1 de taille n^2.

De ce fait il faut que

$$2n + n^2 < cache\ ciblé$$

.

Notre cache L2 fait 256 kB. Nous allons donc faire en sorte de le remplir à 80%.

On note

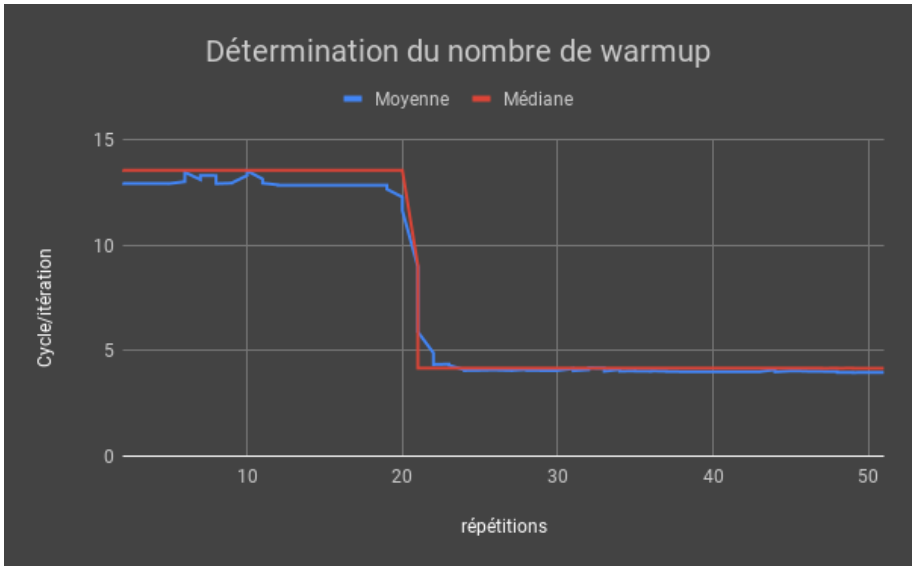
$$L2 = \frac{0,8 * sizeof(L2)}{sizeof(float)}$$

On cherche donc n tel que $2n + n^2 = L2$.
On trouve donc $n = \sqrt{25601} - 1$
$n = 159$

2.5.3 Détermination du nombre de répétitions de warmups et de mesures

Warmups

En suivant la méthodologie décrite précédemment nous pouvons tracer le graphique suivant.



D'après le graphique ci-dessus nous pouvons donc affirmer que 25 warmups sont nécessaires à la machine afin d'être stable.

La chute non progressive est sûrement dû à mon processeur I7 10 génération, qui a des bonnes performances.
Répétitions

Nous souhaitons obtenir un nombre de cycle proche de 3000 pour chaque méta-répétition.
Après différents essais nous choisissons 1000 répétitions afin de respecter les 3000 cycles par itération.

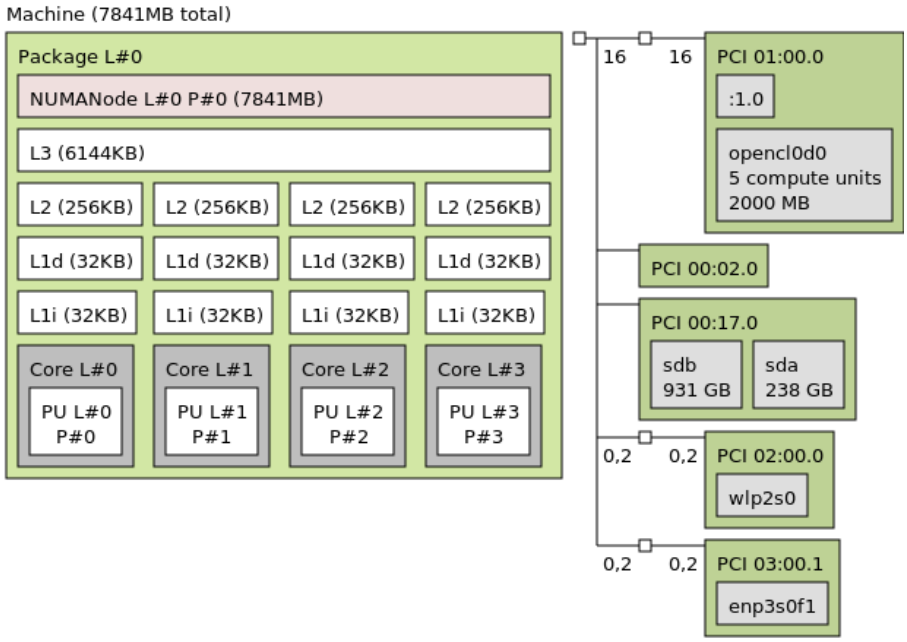
2.6 Cache L3

2.6.1 Environnement expérimental

Toutes les mesures ont été effectuées sur la machine suivante :

Type	Informations
processeur	Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz
Génération du processeur	Kaby Lake
Linux	Pop!_OS 20.10 x86_64
Virtualisation	dual Boot with windows
Version noyau	Linux version 5.8.0-7642-generic
Version GCC	gcc 9.3.0
Version oneAPI	Intel(R) oneAPI DPC++ Compiler 2021.1 (2020.10.0.1113)
Version MAQAO	2.13.0
Techno RAM	8GB DDR4 2400MHz

Voici les informations des différents caches de la machine.



2.6.2 Justification de la taille des tableaux pour que ça tienne dans le cache L3

Nous devons déterminer la taille des tableaux tels que les valeurs traitées tiennent dans le cache souhaité. Nous traitons 3 tableaux, 2 de taille n et 1 de taille n^2.

De ce fait il faut que

$$2n + n^2 < sizeof(L3)$$

.

Notre cache L3 fait 6144 KB. Nous allons donc faire en sorte de le remplir à 80%. On note

$$L3 = \frac{0,8 * sizeof(L3)}{sizeof(float)}$$

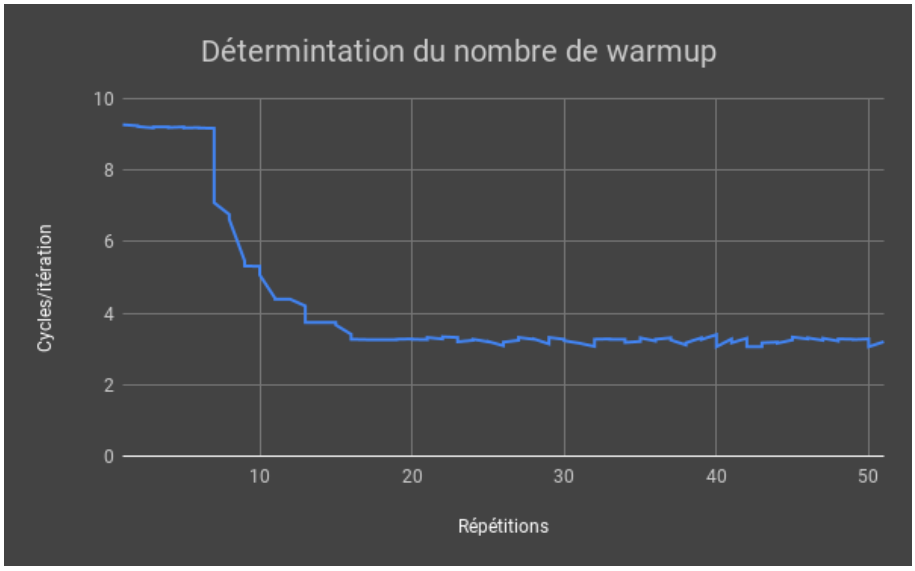
On cherche donc n tel que $2n + n^2 = L3$.
On trouve donc $n = \sqrt{614400} - 1$
$n = 782$

Nos tableaux auront donc une taille de 782 éléments afin de passer dans le cache L3.

2.6.3 Détermination du nombre de répétitions de warmups et de mesures

Warmups

En suivant la méthodologie décrite précédemment nous pouvons tracer le graphique suivant.



D'après le graphique ci-dessus nous pouvons donc affirmer que 20 warmups sont nécessaires à la machine afin d'être stable.

Répétitions

Nous souhaitons obtenir un nombre de cycle proche de 3000 pour chaque méta-répétition.
Après différents essais nous choisissons 800 répétitions afin de respecter cette contrainte.

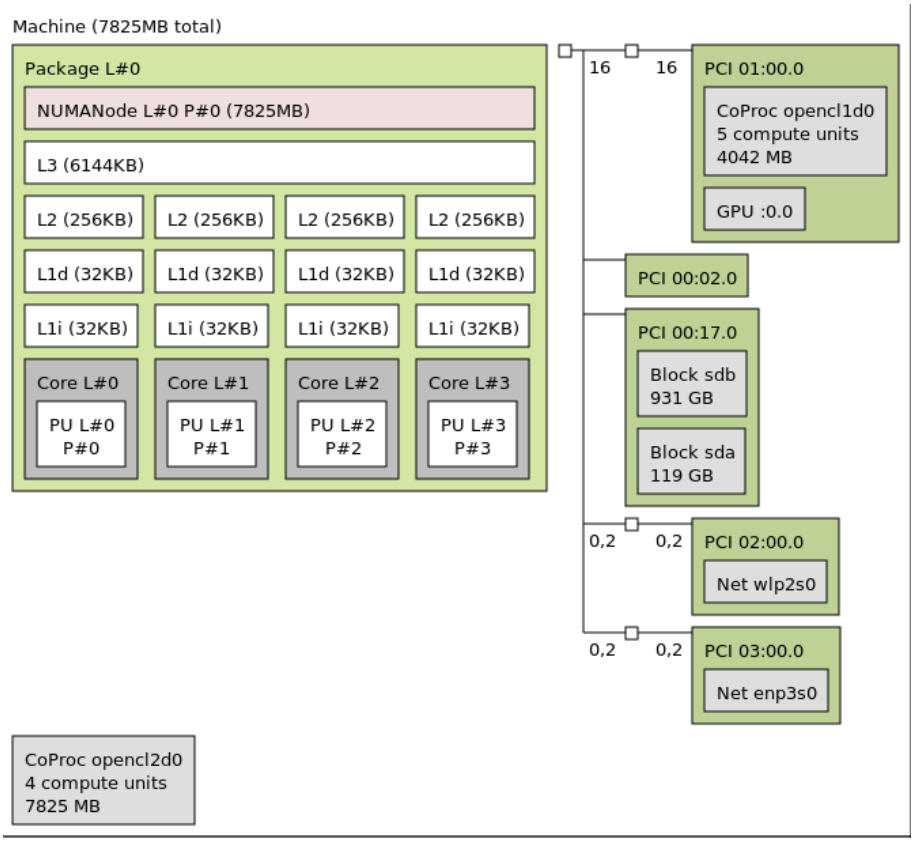
2.7 RAM

2.7.1 Environnement expérimental

Toutes les mesures du ont été effectuées sur la machine suivante :

Type	**Informations**
processeur	Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz
Génération du processeur	Kaby Lake
Linux	Pop!_OS 20.10 x86_64
Virtualisation	dual Boot with windows
Version noyau	Linux version 5.8.0-7625-generic
Version GCC	gcc 10.2.0
Version oneAPI	Intel(R) oneAPI DPC++ Compiler 2021.1 (2020.10.0.1113)
Version MAQAO	2.13.0
Techno RAM	8GB DDR4 3200MHz

Voici les informations des différents caches de la machine.



2.7.2 Justification de la taille des tableaux pour que ça tienne dans la RAM

Nous devons déterminer la taille des tableaux tels que les valeurs traitées tiennent dans le cache souhaité. Nous traitons 3 tableaux, 2 de taille n et 1 de taille n^2.

De ce fait il faut que

$$2n + n^2 < RAM$$

.

Nous voulons avoir un tableau passant dans la RAM. De ce fait nous allons prendre un tableau faisant 3 fois la taille du cache L3.

On note

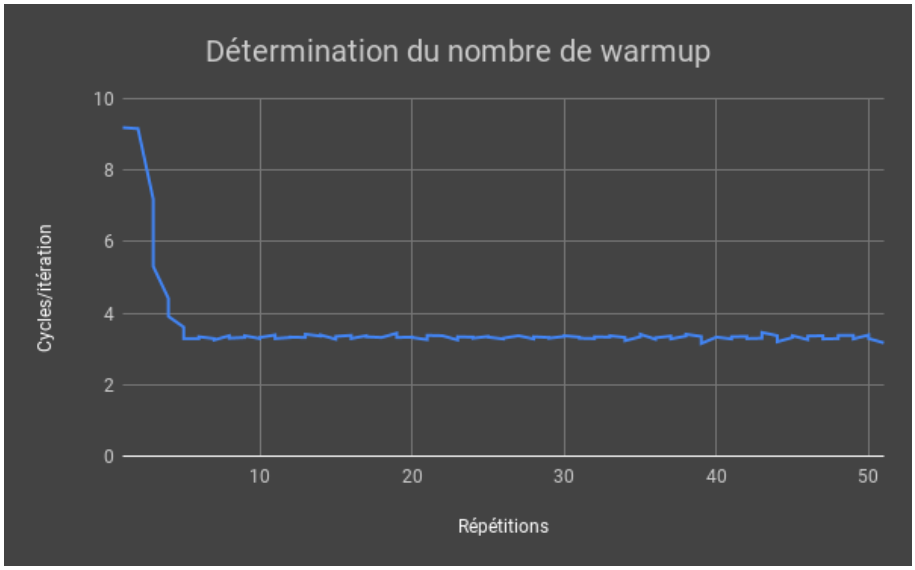
$$RAM = \frac{3 * sizeof(L3)}{sizeof(float)}$$

On cherche donc n tel que $2n + n^2 = RAM$.
On trouve donc $n = \sqrt{2304001} - 1$
$n = 1516$

2.7.3 Détermination du nombre de répétitions de warmups et de mesures

Warmups

En suivant la méthodologie décrite précédemment nous pouvons tracer le graphique suivant.



D'après le graphique ci-dessus nous pouvons donc affirmer que 10 warmups sont nécessaires à la machine afin d'être stable.

Répétitions

Nous souhaitons obtenir un nombre de cycle proche de 3000 pour chaque méta-répétition.Après différents essais nous choisissons 1000 répétitions afin de respecter cette contrainte.

3 Situation à l'issue de la phase 1

3.1 Analyses et mesures en L1 (Olivier BENABEN)

3.1.1 Warmups & Répétitions

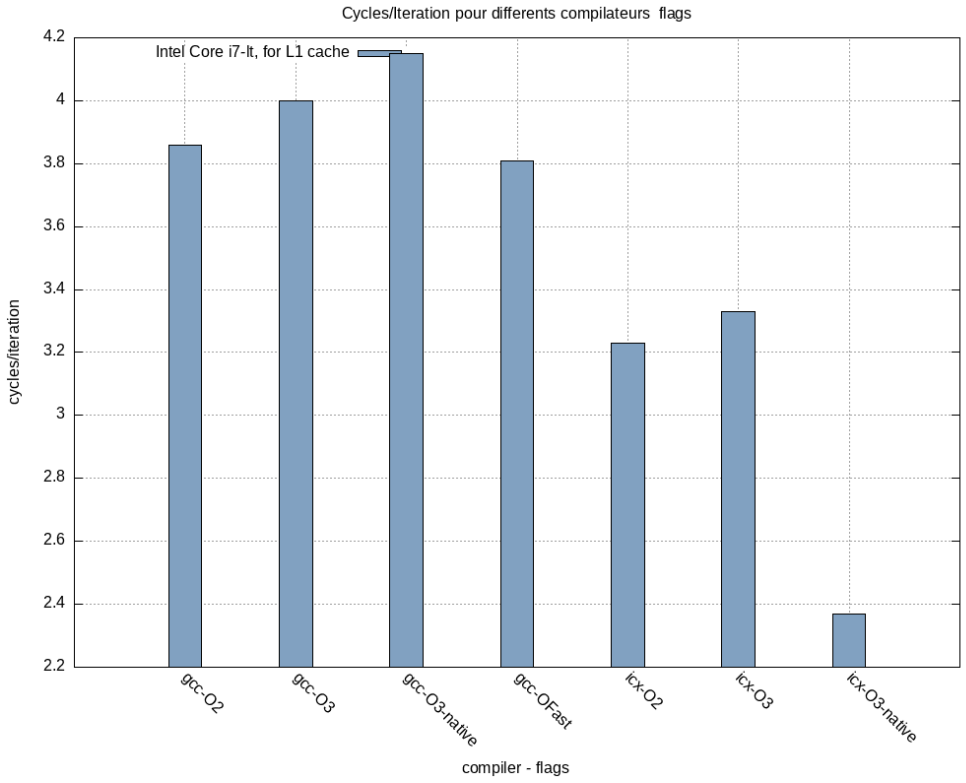
Le nombre de warmups et de répétitions déterminés dans la phase 1 sur ma machine étaient :

— **warmups :** `30`

— **repetitions :** `750`

3.1.2 Flags de compilations

Ci dessous la comparaison de la performance du programme en fonction des flags de compilation utilises (avec `gcc`, `icc` et `icx`) :



3.2 Analyses et mesures en L2 (Hugo HENROTTE)

Toutes les mesures ont été effectuées avec la fonction ORIGINAL.

3.2.1 Warmups & Répétitions

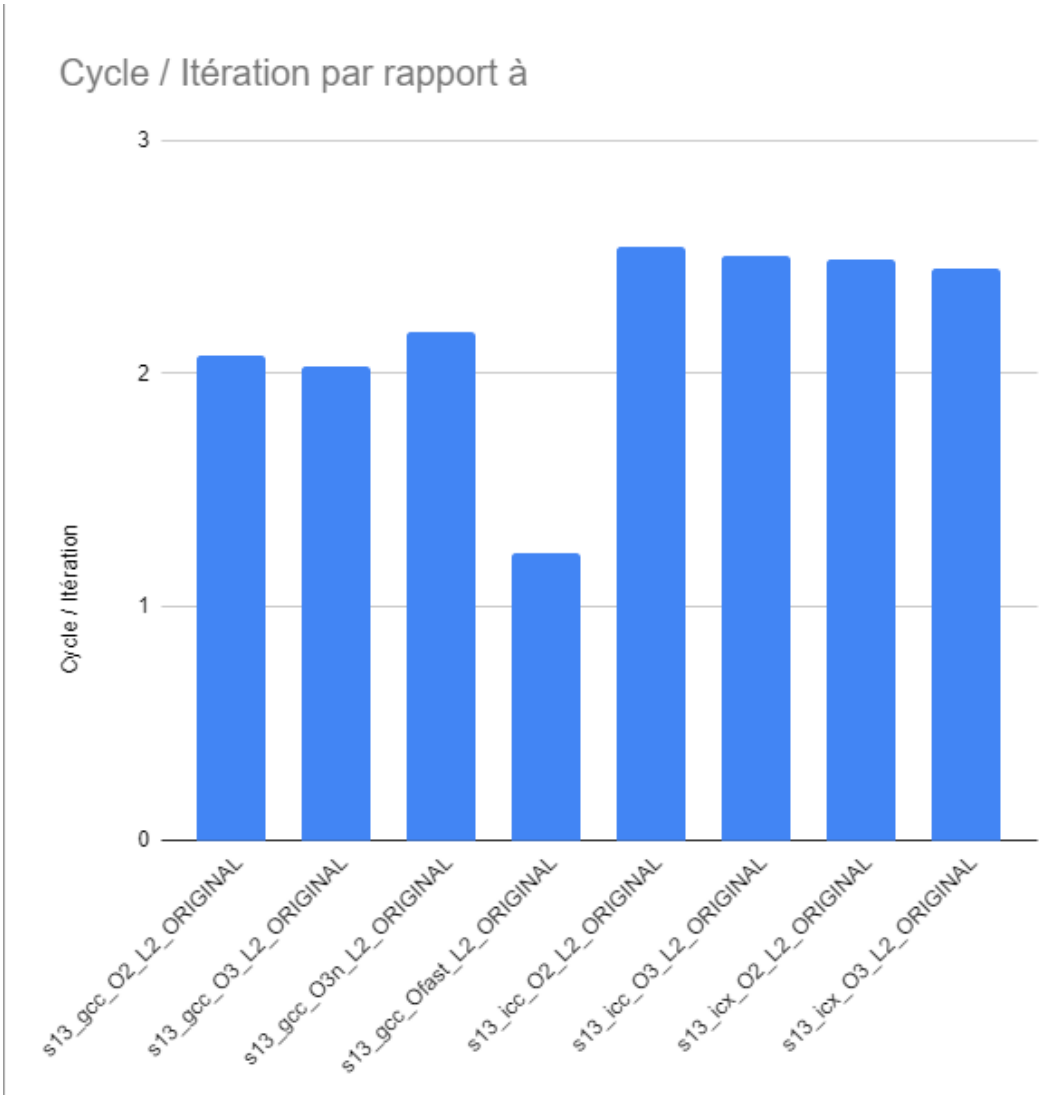
Le nombre de warmups et de répétitions déterminés dans la phase 1 sur ma machine étaient :

— **warmups :** `25`

— **repetitions :** `1000`

3.2.2 Flags de compilations

Ci dessous la comparaison de la performance du programme en fonction des flags de compilation utilises (avec `gcc`, `icc` et `icx`) :



On remarque que avec le compilateur gcc et particulièrement avec $-OFast$ on obtient le minimum de cycle par itération.

3.3 Analyses et mesures en L3 (Maxence BRINGUIER)

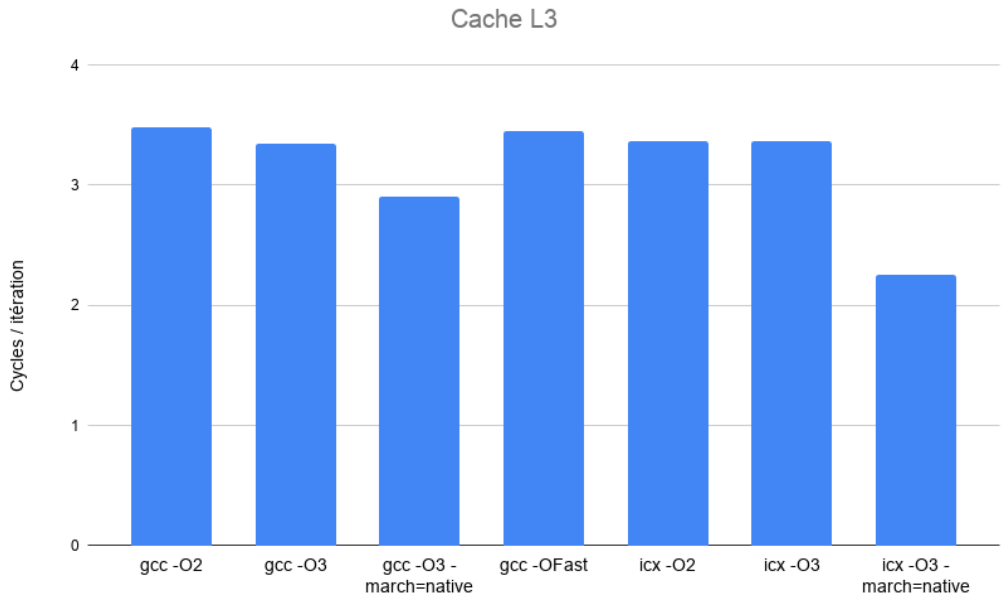
Toutes les mesures ont été effectuées avec la fonction ORIGINAL.

Le nombre de warmups et de répétitions déterminés dans la phase 1 sur ma machine étaient :

— **warmups :** `20`

— **repetitions :** `800`

Nous allons donc comparer le temps médian de chaque exécution en fonction de différents compilateur et flags d'optimisation.



On remarque que *gcc* obtient des cyles par itération légèrement plus court qu'avec *icx*. Les meilleures performances étant toutes deux atteintes avec le flag $-O3 march = native$.

3.4 Analyses et mesures en RAM (Maxime VINCENT)

3.4.1 Warmups & Répétitions

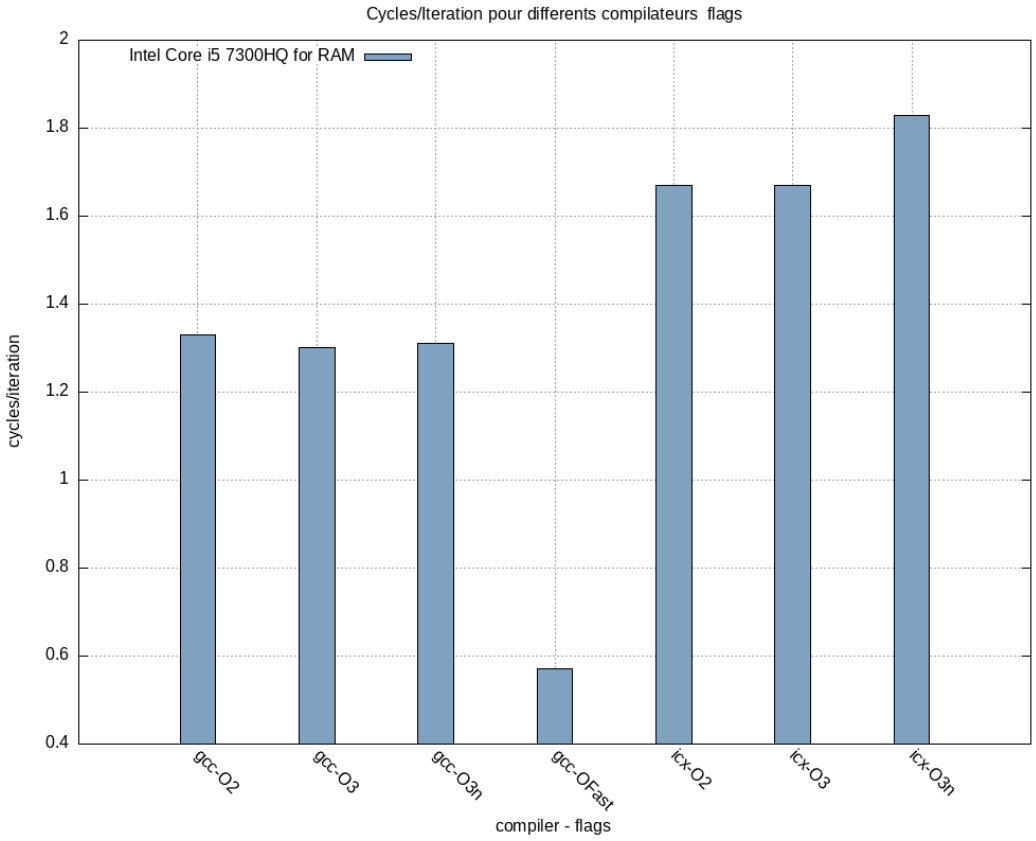
Le nombre de warmups et de répétitions déterminés dans la phase 1 sur ma machine étaient :

— **warmups :** `10`

— **repetitions :** `1000`

3.4.2 Flags de compilations

Ci dessous la comparaison de la performance du programme en fonction des flags de compilation utilises (avec `gcc`, `icc` et `icx`) :



3.5 Conclusion

Malgré les différents flags et les différents compilateurs utilisés, nous avons pu remarquer à travers les différents rapports maqao que le code ne produit pas un binaire vectorisé. Nous observons donc la limite de l'étude statique de ce code, car le code original de notre fonction à optimiser ne permet pas aux compilateurs de l'induire vers une vectorisation des instructions.

Le fait d'utiliser des flags plus optimisés ne rend pas forcément le code plus performant. Les codes générés sont parfois identiques. Par contre la gestion des niveaux de caches que nous avons calculés au préalable permet d'observer que le nombre de cycles par répétitions est plus important dans le cache L1 que dans la Ram (donc plus le n est petit plus le nombre de cycles par répétition est important).

Lors de la phase 2, il sera donc important d'optimiser le code avant la compilation afin de permettre de meilleurs résultats et analyses, typiquement le résultat attendu sera l'utilisation d'instructions vectorielles. Le fait d'optimiser le code au préalable permet des gains d'exécution importants.

Notre étude s' est focalisée dans un premier temps sur gcc. Les comparaisons faites entre icx et gcc peuvent donc ne pas être représentatives de la réalité. C'est un axe majeur à améliorer lors de la phase 2.

4 Optimisation source en se contraignant à `gcc -O2`

4.1 Analyses faites en commun (et présentation des optims)

Tout d'abord On trouve une forme plus "agréable" en supprimant des conditions inutiles. Puis nous avons fait le choix de ne charger qu'une seule fois les éléments d'une ligne.

Listing 1 – Loop-invariant code motion

```
for ( i = 0; i < n ; i ++) {
    bi = b [ i ];
    for ( j = offset; j < n; j ++) {
      c [ i ][ j ] = a[j] < radius ? 0.0 : a [ j ] / bi;
    }
  }
```

Une fois cette opération faite nous avons donc dérouler notre boucle afin de la vectoriser. Nous l'avons donc dérouler plusieurs fois pour l'étudier.

Listing 2 – Loop unrolling - Loop-invariant

```
for ( i = 0; i < n ; i ++) {
    bi = b [ i ];

    for ( j = offset; j < (n-3); j +=4) {
      c [ i ][ j ] = a[j] < r ? 0.0 : a [ j ] / bi;
      c [ i ][ j+1 ] = a[j+1] < r ? 0.0 : a [ j+1 ] / bi;
      c [ i ][ j+2 ] = a[j+2] < r ? 0.0 : a [ j+2 ] / bi;
      c [ i ][ j+3 ] = a[j+3] < r ? 0.0 : a [ j+3 ] / bi;
    }

    for(j = (n-3); j<n;j++){
      c [ i ][ j ] = a[j] < r ? 0.0 : a [ j ] / bi;
    }
  }
```

Après différents déroulages et analyse du code on remarque que les registres XMM sont codés sur 128 bits et un float est codé sur 4 bytes.
On peut donc peut mettre 4 floats par registre xmm.

Nous devons donc dérouler 4 fois la boucle pour maximiser la vectorisation. On déroule donc 4 fois les deux boucles, voir le code ci-dessous :

Listing 3 – Loop unrolling 4 times - Loop-invariant

```
for ( i = 0; i < (n-3) ; i+= 4) {
    bi[0][0] = b[i];bi[0][1] = b[i];bi[0][2] = b[i];bi[0][3]
        = b[i];
    bi[1][0] = b[i+1];bi[1][1] = b[i+1];bi[1][2] = b[i+1];bi
       [1][3] = b[i+1];
    bi[2][0] = b[i+2];bi[2][1] = b[i+2];bi[2][2] = b[i+2];bi
       [2][3] = b[i+2];
    bi[3][0] = b[i+3];bi[3][1] = b[i+3];bi[3][2] = b[i+3];bi
       [3][3] = b[i+3];
    for ( j = offset; j < (n-3); j +=4) {
```

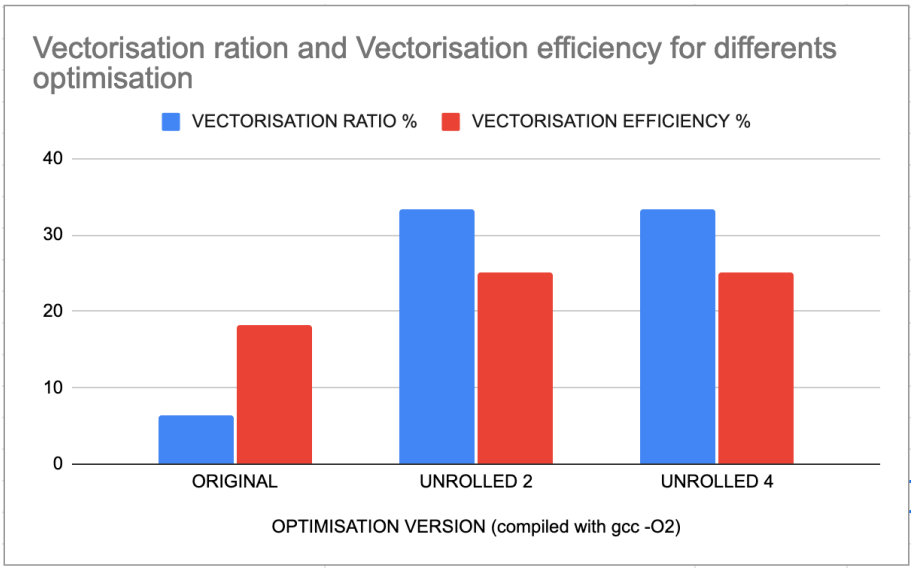
```
    c [ i ][ j ] =   a[j] < r ? 0.0 :   a[j]  / bi[0][0];
    c [ i ][ j+1 ] = a[j+1] < r ? 0.0 : a[j+1] / bi[0][1];
    c [ i ][ j+2 ] = a[j+2] < r ? 0.0 : a[j+2] / bi[0][2];
    c [ i ][ j+3 ] = a[j+3] < r ? 0.0 : a[j+3] / bi[0][3];
    c [ i+1 ][ j ]   = a[j]  <  r ? 0.0 : a[j]  / bi
      [1][0];
    c [ i+1 ][ j+1 ] = a[j+1] < r ? 0.0 : a[j+1] / bi
      [1][1];
    c [ i+1 ][ j+2 ] = a[j+2] < r ? 0.0 : a[j+2] / bi
      [1][2];
    c [ i+1 ][ j+3 ] = a[j+3] < r ? 0.0 : a[j+3] / bi
      [1][3];
    c [ i+2 ][ j ] =   a[j]  <  r ? 0.0 : a[j]  / bi
      [2][0];
    c [ i+2 ][ j+1 ] = a[j+1] < r ? 0.0 : a[j+1] / bi
      [2][1];
    c [ i+2 ][ j+2 ] = a[j+2] < r ? 0.0 : a[j+2] / bi
      [2][2];
    c [ i+2 ][ j+3 ] = a[j+3] < r ? 0.0 : a[j+3] / bi
      [2][3];
    c [ i+3 ][ j ] =   a[j]  <  r ? 0.0 : a[j]  / bi
      [3][0];
    c [ i+3 ][ j+1 ] = a[j+1] < r ? 0.0 : a[j+1] / bi
      [3][1];
    c [ i+3 ][ j+2 ] = a[j+2] < r ? 0.0 : a[j+2] / bi
      [3][2];
    c [ i+3 ][ j+3 ] = a[j+3] < r ? 0.0 : a[j+3] / bi
      [3][3];
  }

  for(j = (n-3); j<n;j++){
    c [ i ][ j ]   = a[j] < r ? 0.0 : a[j]  / bi[0][0];
    c [ i+1 ][ j ] = a[j] < r ? 0.0 : a[j] / bi[1][0];
    c [ i+2 ][ j ] = a[j] < r ? 0.0 : a[j] / bi[2][0];
    c [ i+3 ][ j ] = a[j] < r ? 0.0 : a[j] / bi[3][0];
  }

}
for(i = (n-3); i<n;i++){
  bi[0][0] = b[i];bi[0][1] = b[i];bi[0][2] = b[i];bi[0][3]
      = b[i];
  for ( j = offset; j < (n-3); j +=4) {
    c [ i ][ j ] =   a[j]  <  r ? 0.0 :  a[j]  / bi[0][0];
    c [ i ][ j+1 ] = a[j+1] < r ? 0.0 : a[j+1] / bi[0][1];
    c [ i ][ j+2 ] = a[j+2] < r ? 0.0 : a[j+2] / bi[0][2];
    c [ i ][ j+3 ] = a[j+3] < r ? 0.0 : a[j+3] / bi[0][3];
  }
  for(j = (n-3); j<n;j++){
    c [ i ][ j ]   = a[j] < r ? 0.0 : a[j] / bi[0][0];
  }
}
```

4.2 Analyses et mesures en L1 (Olivier BENABEN)

Pour augmenter les performances de notre kernel, nous essayons donc dans un premier temps de "forcer" le compilateur a utiliser de instructions vectorielles.

Pour cela nous utilisons la methode de *unrolling* comme vue précédemment, d'abord deux fois, puis quatre.

En mesurant les performances sur le cache L1, et en analysant avec Maqao le taux de véctorisation et son efficacité sur les différentes versions, on remarque que la méthode fonctionne bien : on a réussi a doubler le taux de véctorisation du code.



Regardons en détail les effets de l'unrolling dans le code assembleur produit par `gcc`.

Version originale

Source Function	Level	Vectorization Ratio (%)	Vectorization Efficiency (%)
s13	Innermost	6.35	18.28

```
0x17e8 TEST     %EAX,%EAX
0x17ea JS       1815
0x17ec CMP      %EAX,%EDI
0x17ee JBE      1815
0x17f0 MOVL     $0,(%RCX,%RDX,1)
0x17f7 MOVSS    (%R9,%RDX,1),%XMM0
0x17fd PXOR     %XMM1,%XMM1
0x1801 CVTSS2SD %XMM0,%XMM1
0x1805 COMISD   %XMM1,%XMM2
0x1809 JBE      1815
0x180b DIVSS    (%R8),%XMM0
0x1810 MOVSS    %XMM0,(%RCX,%RDX,1)
0x1815 ADD      $0x1,%EAX
0x1818 ADD      $0x4,%RDX
0x181c CMP      %ESI,%EAX
0x181e JNE      17e8
```

Les instructions dans la boucle principale sont toutes des `scalar single` : les opérations sur les tableaux se font case par case, aucune vectorisation n'est présente.

Version unrolled 2

Source Function	Vectorization Ratio (%)	Vectorization Efficiency (%)
s13	33.33	25

```
0x1800 MOVSS    (%RSI,%RAX,4),%XMM4
0x1805 MOVAPS   %XMM1,%XMM3
0x1808 COMISS   %XMM4,%XMM0
0x180b JA       1814
0x180d MOVAPS   %XMM4,%XMM3
0x1810 DIVSS    %XMM2,%XMM3
0x1814 MOVSS    %XMM3,(%RCX,%RAX,4)
0x1819 MOVSS    0x4(%RSI,%RAX,4),%XMM4
0x181f MOVAPS   %XMM1,%XMM3
0x1822 COMISS   %XMM4,%XMM0
0x1825 JA       182e
0x1827 DIVSS    %XMM2,%XMM4
0x182b MOVAPS   %XMM4,%XMM3
0x182e MOVSS    %XMM3,0x4(%RCX,%RAX,4)
0x1834 MOVSS    0x8(%RSI,%RAX,4),%XMM4
0x183a MOVAPS   %XMM1,%XMM3
0x183d COMISS   %XMM4,%XMM0
0x1840 JA       1849
0x1842 MOVAPS   %XMM4,%XMM3
0x1845 DIVSS    %XMM2,%XMM3
0x1849 MOVSS    %XMM3,0x8(%RCX,%RAX,4)
0x184f MOVSS    0xc(%RSI,%RAX,4),%XMM4
0x1855 MOVAPS   %XMM1,%XMM3
0x1858 COMISS   %XMM4,%XMM0
0x185b JA       1864
0x185d DIVSS    %XMM2,%XMM4
0x1861 MOVAPS   %XMM4,%XMM3
0x1864 MOVSS    %XMM3,0xc(%RCX,%RAX,4)
0x186a ADD      $0x4,%RAX
0x186e CMP      %EAX,%EDX
0x1870 JA       1800
```

Une grosse partie des instructions sont encore une fois des `scalar`, cependant certaines instructions sont passées en `packed` (les instructions `movaps` : Move Aligned Packed Single-Precision Floating-Point Values)

Version unrolled 4

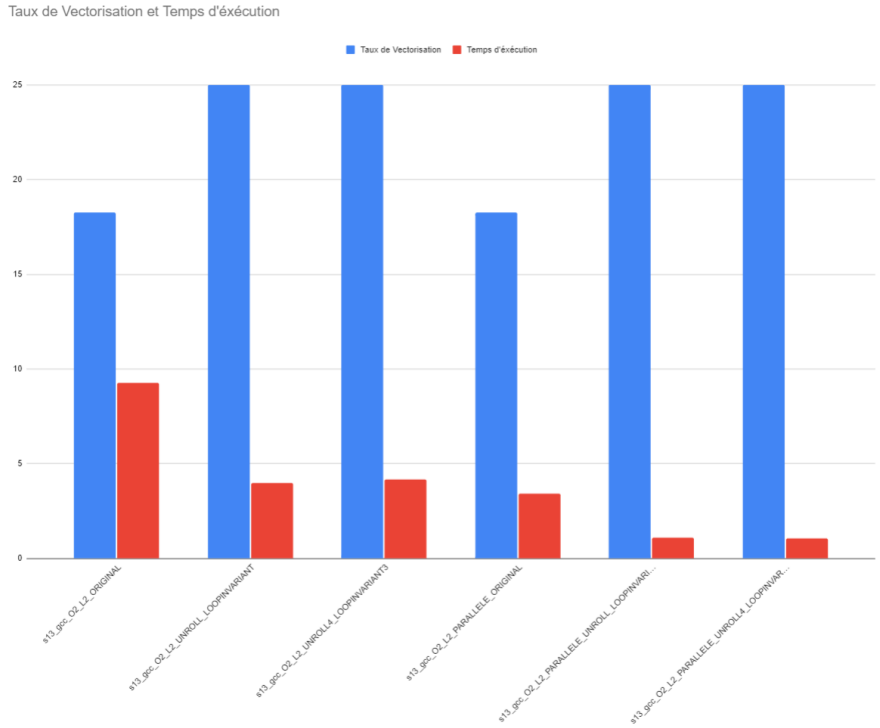
Source Function	Coverage (%)	Vectorization Ratio (%)	Vectorization Efficiency (%)
s13	66.67	33.33	25
s13	33.33	33.33	25

Dans cette version nous avons essayé de pousser la vectorisation a la main jusqu'au bout en espérant maximiser le taux de vectorisation du code. Cependant cet effort n'a pas été très concluant : le code assembleur obtenu pour la boucle principale est le même que pour l'unroll 2 précédant (la seconde boucle qui apparait dans le rapport de Maqao est en fait le bout de code qui sert a ramasser les miettes de l'unrolling).

4.3 Analyses et mesures en L2 (Hugo HENROTTE)

Nous avons chercher a optimiser la boucle et a l'analyser en comparant avec différents stades d'optimisation et différents flags d'exécutions permettant de meilleur optimisation.

	Taux de Vectorisation	Temps d'éxécution
s13_gcc_O2_L2_ORIGINAL	18,28	9,27
s13_gcc_O2_L2_UNROLL_LOOPINVARIANT	25	3,99
s13_gcc_O2_L2_UNROLL4_LOOPINVARIANT3	25	4,19
s13_gcc_O2_L2_PARALLELE_ORIGINAL	18,28	3,43
s13_gcc_O2_L2_PARALLELE_UNROLL_LOOPINVARIANT	25	1,11
s13_gcc_O2_L2_PARALLELE_UNROLL4_LOOPINVARIANT3	25	1,07



Les temps d'exécution des fonction déroulées avec un invariant de boucle sont beaucoup plus performantes.
Puis lorsque on déroule on s'aperçoit que le pourcentage de véctorisation du code assembleur est beaucoup plus grand et plus on déroule la boucle plus la véctorisation est importante.

Puis lorsque on parallélise les boucles on obtient des temps d'exécution encore plus court de l'ordre de 3 fois plus rapide que la version sans parallélisation.

4.4 Analyses et mesures en L3 (Maxence BRINGUIER)

Nous allons donc dérouler les deux boucles puis comparer les résultats obtenus.

Source Function	Level	Coverage (%)	Time (s)	Vectorization Ratio (%)	Vectorization Efficiency (%)
s13	Innermost	99.88	24.56	6.35	18.28

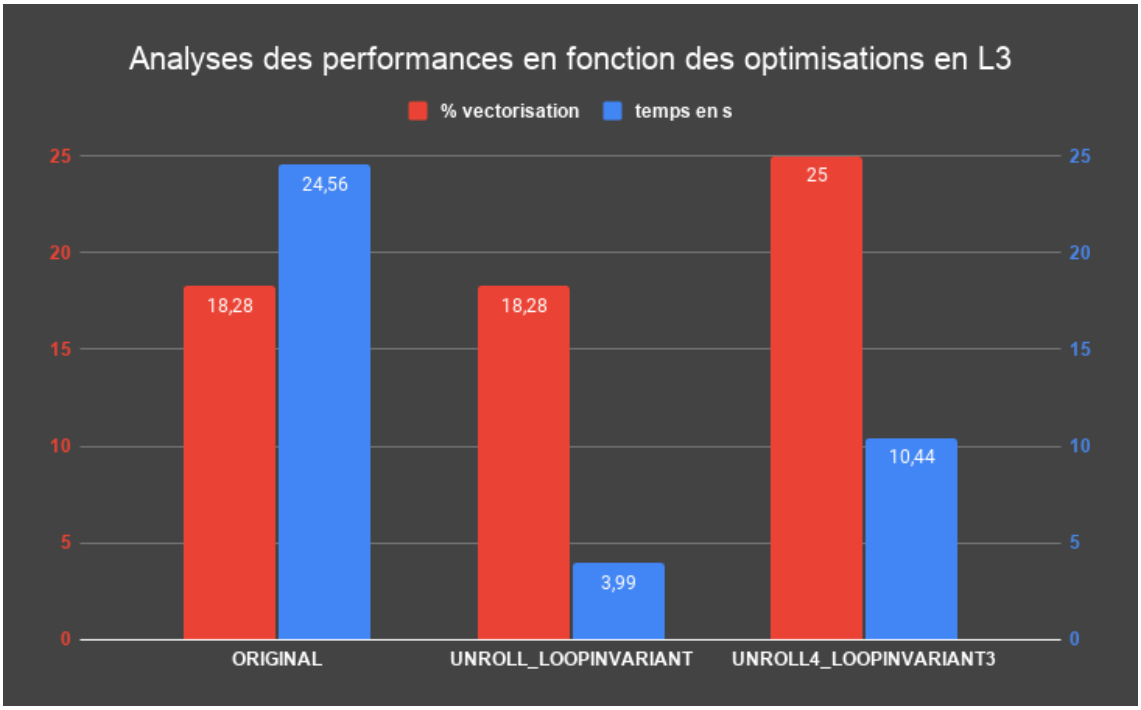
Fonction originale

Source Function	Level	Coverage (%)	Time (s)	Vectorization Ratio (%)	Vectorization Efficiency (%)
s13	Innermost	100	3.99	6.35	18.28

Fonction déroulée avec seulement la boucle intérieure

Source Function	Level	Coverage (%)	Time (s)	Vectorization Ratio (%)	Vectorization Efficiency (%)
s13	Innermost	98.77	10.44	33.33	25

Fonction déroulée avec les 2 boucles



Ici nous nous rendons compte que les temps d'exécution des fonction déroulées avec un invariant de boucle sont beaucoup plus performantes en terme de temps d'exécution.

On remarque également que le compilateur vectorise beaucoup mieux la fonction avec les deux boucles déroulées malgré comparé à la version déroulée une seule fois. La vectorisation s'effectue au détriment du temps d'exécution.

Coté code assembleur peu de différences sont notables entre les versions déroulées si ce n'est la duplication de registres dû au déroulage des boucles.
On note cependant une différence avec la fonction originale. En effet des XOR sont

présents dans la version *ORIGINAL*. Ils sont remplacés par des *MOVE* dans les versions déroulées.

```
0x183f MOVSS    (%RSI,%RDX,1),%XMM1
0x1844 PXOR     %XMM2,%XMM2
0x1848 CVTSS2SD %XMM1,%XMM2
0x184c COMISD   %XMM2,%XMM0
0x1850 JBE      185c
0x1852 DIVSS    (%R10),%XMM1    [1]
```

Code assembleur d'une itération de la fonction originale

```
0x1859 MOVSS    0x4(%RSI,%RAX,4),%XMM3    [1]
0x185f MOVAPS   %XMM8,%XMM2
0x1863 COMISS   %XMM3,%XMM0
0x1866 JA       186f
0x1868 DIVSS    %XMM1,%XMM3
0x186c MOVAPS   %XMM3,%XMM2
0x186f MOVSS    %XMM2,0x4(%RCX,%RAX,4)    [2]
```

Code assembleur d'une itération des fonctions déroulées

4.5 Analyses et mesures en RAM (Maxime Vincent)

L'objectif de cette analyse est de déterminer la meilleur optimisation possible du kernel en se contraignant à la compilation avec gcc et le flag -O2. Précédemment nous avons vue les différentes optimisation possible pour notre kernel :

— Unroll

— Loop invariant

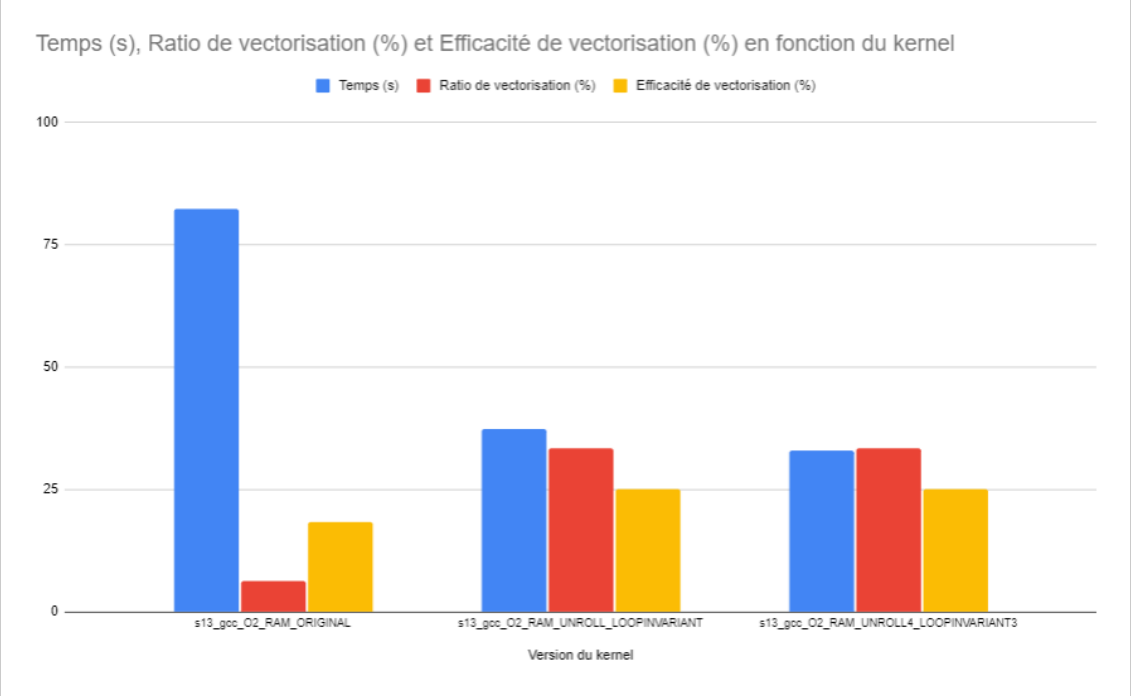
Je vais donc analysé le temps d'exécution, le ratio de vectorisation et l'efficacité de la vectorisation pour les trois optimisation suivante :

— ORIGINAL (kernel original)

— UNROLL_LOOPINVARIANT (boucle interne dérouler 4 fois et loop invariant)

— UNROLL4_LOOPINVARIANT3 (boucle externe et interne dérouler 4 fois et loop invariant)

Version du kernel	Temps (s)	Ratio de vectorisation (%)	Efficacité de vectorisation (%)
s13_gcc_O2_RAM_ORIGINAL	82,43	6,35	18,28
s13_gcc_O2_RAM_UNROLL_LOOPINVARIANT	37,3	33,33	25
s13_gcc_O2_RAM_UNROLL4_LOOPINVARIANT3	32,89	33,33	25



On constate que plus les boucles sont déroulé plus le temps d'exécution est rapide.
On remarque également que le compilateur vectorise beaucoup mieux le code assembleur quand les boucles sont dérouler.
On constate aussi que même si on continue a dérouler les boucles le ratio de vectorisation n'augmentera pas plus.

4.6 Conclusion

Pour tous les niveaux de caches nous avons constaté que plus le ratio de vectorisation est élevé plus le temps d'exécution est rapide. Nous avons aussi constaté que pour les kernel $UNROLL$, le ratio de vectorisation est plus élévé que le kernel $ORIGINAL$.

Au niveau du code assembleur nous avons remarquer que pour le kernel $ORIGINAL$ les instructions dans la boucle principale sont toutes des `scalar single` : les opérations sur les tableaux se font case par case, aucune vectorisation n'est présente.
Pour le kernel $UNROLL$ nous avons constaté que quelque instruction étais passé en `packed`.

5 Optimisation source en débridant la compilation

5.1 Analyses faites en commun (et présentation des optims)

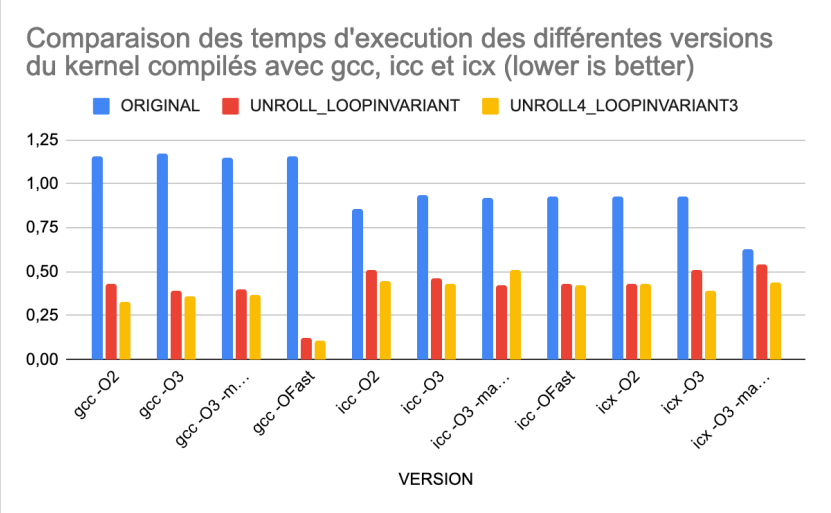
Afin d'améliorer la compilation nous avons décidé d'utilisé différent flags d'optimisation, O2, O3 O3 march=native et Ofast.

L'objectif de l'utilisation de ces flags est de "débridé" la compilation afin que le code assembleur résultant soit le plus optimisé possible.
Nous avons chercher a faire en sorte que le code assembleur résultant utilise des instructions vectorielle afin que le code soit vectorisé.
Nous avons compilé le kernel avec les deux boucle dérouler 4 fois avec gcc et l'option Ofast, on trouve le code assembleur suivant :

```
0x1960 MOVUPS   (%RAX,%R8,1),%XMM12
0x1965 MOVUPS   (%RAX,%R8,1),%XMM13
0x196a MOVAPS   %XMM1,%XMM2
0x196d ADD      $0x1,%R9D
0x1971 MULPS    %XMM4,%XMM12
0x1975 CMPPS    $0x2,%XMM13,%XMM2
0x197a ANDPS    %XMM12,%XMM2
0x197e MOVUPS   %XMM2,(%RDX,%R8,1)
0x1983 MOVUPS   (%RAX,%R8,1),%XMM12
0x1988 MOVUPS   (%RAX,%R8,1),%XMM14
0x198d MOVAPS   %XMM1,%XMM2
0x1990 MULPS    %XMM6,%XMM12
0x1994 CMPPS    $0x2,%XMM14,%XMM2
0x1999 ANDPS    %XMM12,%XMM2
0x199d MOVUPS   %XMM2,(%RDI,%R8,1)
0x19a2 MOVUPS   (%RAX,%R8,1),%XMM12
0x19a7 MOVUPS   (%RAX,%R8,1),%XMM15
0x19ac MOVAPS   %XMM1,%XMM2
0x19af MULPS    %XMM5,%XMM12
0x19b3 CMPPS    $0x2,%XMM15,%XMM2
0x19b8 ANDPS    %XMM12,%XMM2
0x19bc MOVUPS   %XMM2,(%RCX,%R8,1)
0x19c1 MOVUPS   (%RAX,%R8,1),%XMM12
0x19c6 MOVUPS   (%RAX,%R8,1),%XMM13
0x19cb MOVAPS   %XMM1,%XMM2
0x19ce MULPS    %XMM3,%XMM12
0x19d2 CMPPS    $0x2,%XMM13,%XMM2
0x19d7 ANDPS    %XMM12,%XMM2
0x19db MOVUPS   %XMM2,(%RSI,%R8,1)
0x19e0 ADD      $0x10,%R8
0x19e4 CMP      %EBX,%R9D
0x19e7 JB       1960
```

On constate ci dessus que les instructions utilisé sont des instructions `Packed Single-Precision Floating-Point Values` par exemple MOVUPS.
Nous avons donc réussi a générer un code assembleur utilisant des instructions vectorielles.

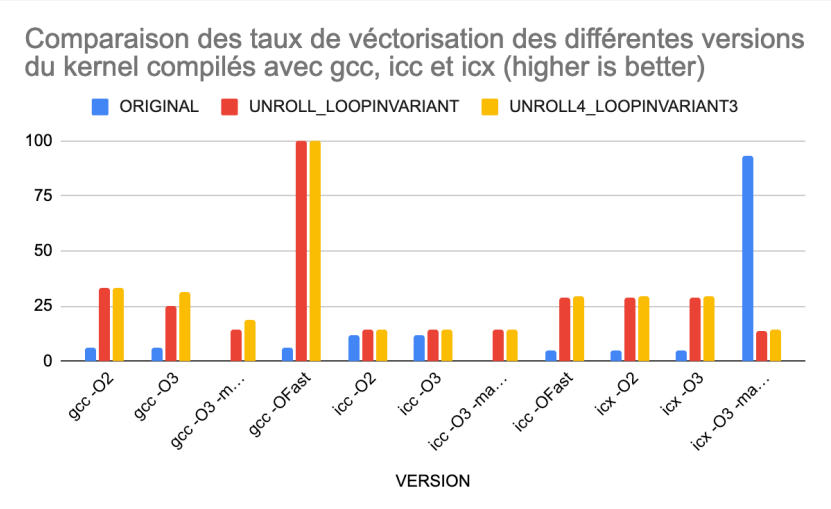
5.2 Analyses et mesures en L1 (Olivier Benaben)



On remarque que en général le binaire produit par gcc est un peu plus lent que ceux que produisent icc et icx. Cependant les améliorations de véctorisation qu'apportent nos déroulages de boucles sont bien plus effectifs avec gcc : on gagne d'un facteur 4 en temps d'execution, alors que le gain n'est que d'un facteur 2 avec icc et icx.

Il est intéressant de noter que le gain de temps est assez faible pour chaque version juste en changeant de compilateur/flags de compilation, comparé à ce qu'apporte comme performance l'unrolling "à la main". Aussi, on remarque que le gain entre l'unrolling 2 et 4 n'est pas significatif, même si il apporte une petite amélioration en général.

Regardons au niveau des taux de vectorisation des différentes versions si l'on retrouve des résulats qui expliqueraient notre permier graphe :



Comme prévu pour gcc : le taux de vectorisation est bien plus haut pour les versions déroulées. Cela peut donc expliquer le speedup obtenu pour gcc : les instructions vectorielles opèrent sur plusieurs cases des tableaux à la fois, et on peut alors réduire le nombre de fois ou l'on boucle dans la fonction principale.

On trouve le même résultat pour les binaires que produisent icc et icx ; ce qui n'explique donc pas le speedup moindre par rapport à gcc. Cette différence vient surement du fait que icc et icx a un niveau d'optimisation de base plus élevé que gcc, et que les binaires sont particulièrement efficaces sur un processeur intel, ce qui vient "lisser" la courbe de speedup obtenue avec notre exemple.

Il est d'ailleurs intéressant d'aller chercher dans l'assembleur produit les techniques qu'utilisent ces compilateurs pour avoir une telle avance par rapport à gcc, même pour la version original du kernel. On trouve donc que quand gcc utilise des instructions "simples" comme `COMP`, `MOVSS/PS`, `MOVSS/PS`, `ADDSS/PS`, icc et icx utilisent des instruction très spécifiques aux processeurs intel comme `VUCOMISS`, `VCVTSS2SS/SD`... C'est surement de là que vient cette différence de performances initiales.

Enfin, un point intéressant est le dernier point avec `icx -O3 -march=native` : on remarque que le taux de vectorisation est bien plus élevé avec le kernel original que les versions déroulées.

En regardant le code assembleur de la boucle on trouve en effet qu'avec la version originale du kernel, les instructions utilisées sont d'une part très précises à mon architecture, mais toutes les instructions sont vectorielles. On peut alors se demander pourquoi ce haut ratio de vectorisation disparaît pour les versions du code déroulées ?

Cela vient sûrement du fait que l'on a "dérangé" le compilateur en déroulant, vu qu'il est conçu pour reconnaître les patterns "courant" de code (ie. il comprendra mieux comment véctoriser une boucle for simple qu'une boucle étant à moitié déroulée comme nous l'avons fait).

Regardons au passage l'assembleur produit par GCC -Ofast pour les différentes versions du kernel, maqao indique que la boucle est vectorisée à 100% :

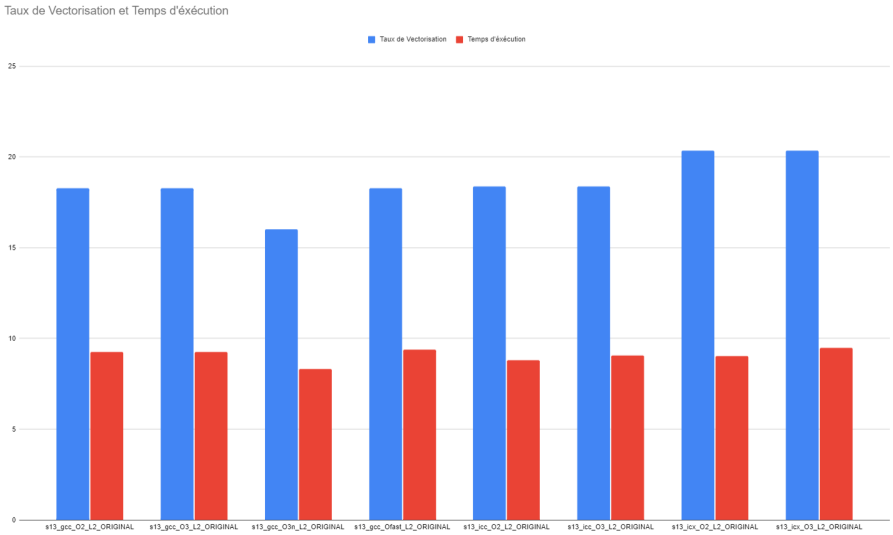
```
1 GCC -Ofast FULLY VERCTORIZED
2 ----------------------------
3 0x18e0 MOVUPS   (%RDX,%RCX,1),%XMM1
4 0x18e4 MOVUPS   (%RDX,%RCX,1),%XMM8
5 0x18e9 MOVAPS   %XMM4,%XMM0
6 0x18ec ADD  $0x1,%R11D
7 0x18f0 MULPS    %XMM2,%XMM1
8 0x18f3 CMPPS    $0x2,%XMM8,%XMM
9 0x18f8 ANDPS    %XMM1,%XMM0
10 0x18fb MOVUPS   %XMM0,(%RAX,%RCX,1)
11 0x18ff ADD  $0x10,%RCX
12 0x1903 CMP  %EDI,%R11D
13 0x1906 JB   18e0
~
~
~
~
~
```

```
1 GCC -Ofast ORIGINAL
2 -------------------
3 0x17e8 TEST %EAX,%EAX
4 0x17ea JS   1815
5 0x17ec CMP  %EAX,%EDI
6 0x17ee JBE  1815
7 0x17f0 MOVL $0,(%RCX,%RDX,1)
8 0x17f7 MOVSS    (%R9,%RDX,1),%XMM0
9 0x17fd PXOR %XMM1,%XMM1
10 0x1801 CVTSS2SD %XMM0,%XMM1
11 0x1805 COMISD   %XMM1,%XMM2
12 0x1809 JBE  1815
13 0x180b DIVSS    (%R8),%XMM0
14 0x1810 MOVSS    %XMM0,(%RCX,%RDX,1)
15 0x1815 ADD  $0x1,%EAX
16 0x1818 ADD  $0x4,%RDX
17 0x181c CMP  %EAX,%ESI
18 0x181e JNE  17e8
```

En effet on n'utilise plus que des informations en `...PS`, indiquant une instruction vectorielle.

5.3 Analyses et mesures en L2 (Hugo HENROTTE)

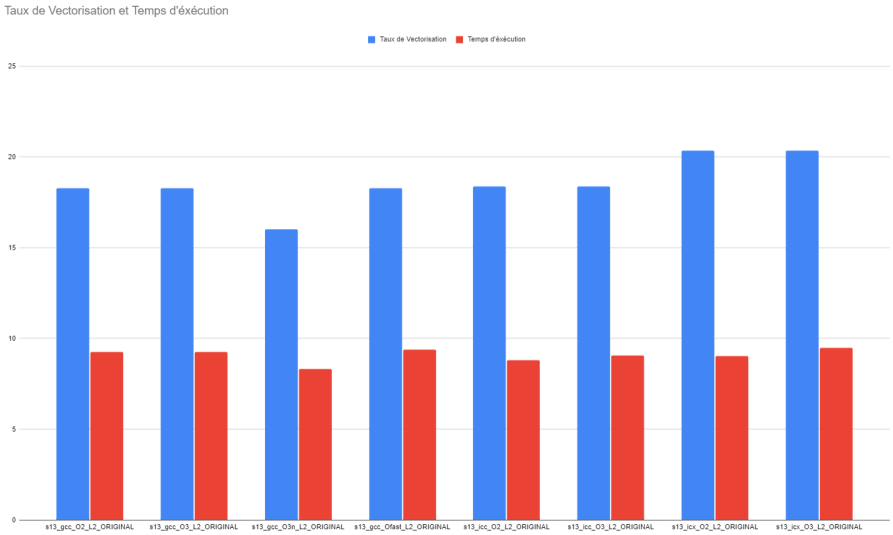
5.3.1 Comparaison de l'efficacité de vectorisation sans dérouler les boucles sous gcc selon les flags -O2 -O3 -OFast, icc selon -O2 -O3 et icx selon -O2 -O3



D'après le graphique ci-dessus nous nous rendons compte que les flags $-O3$ et $-O2$ pour icx on le taux de vectorisation le plus important.

Le flag $-O3 march = native$ pour gcc a énormement de mal à vectoriser les boucles.

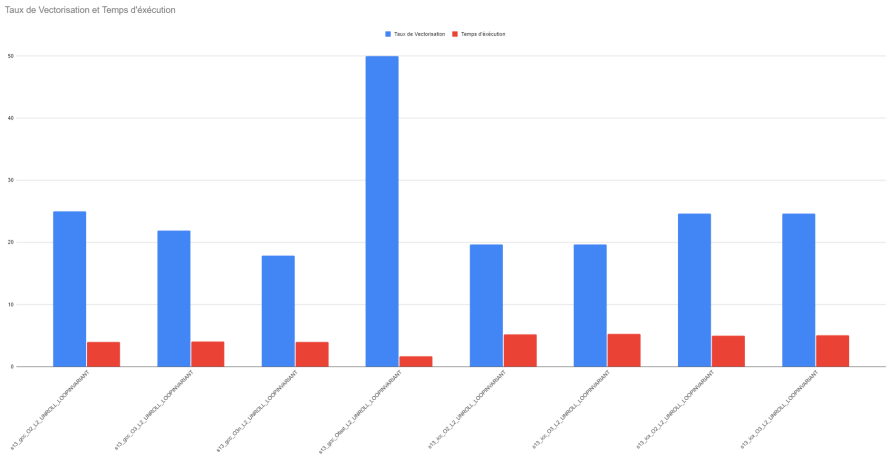
5.3.2 Comparaison des temps d'exécution sans dérouler les boucles sous gcc selon les flags -O2 -O3 -OFast, icc selon -O2 -O3 et icx selon -O2 -O3



D'après le graphique ci-dessus nous nous rendons compte que le flag $-O3 march = native$ pour gcc a le temps d'éxécution le plus rapids.

D'après le graphique ci-dessus nous nous rendons compte que les flags $-O3$ et $-O2$ pour icx on le taux de vectorisation le plus important.

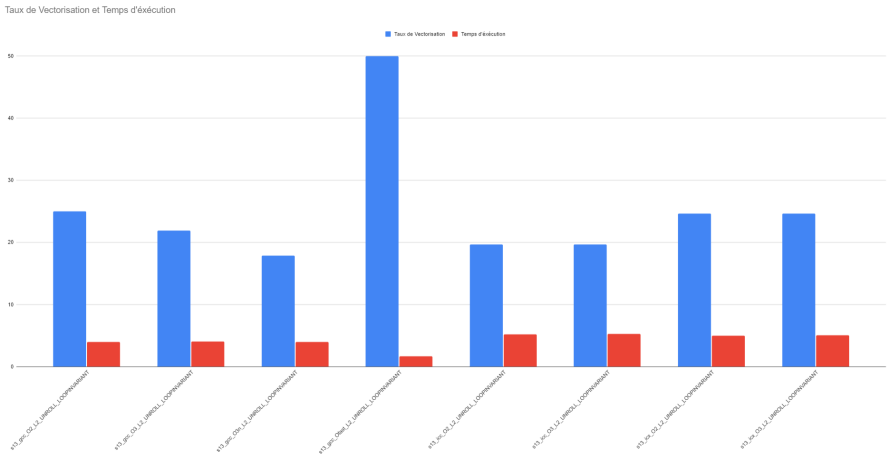
5.3.3 Comparaison de l'efficacité de vectorisation en déroulant que la boucle intérieur sous gcc selon les flags -O2 -O3 -OFast, icc selon -O2 -O3 et icx selon -O2 -O3



D'après le graphique ci-dessus nous nous rendons compte que les flags $-Ofast$ pour gcc on le taux de vectorisation le plus important lorsque la boucle intérieure est déroulé.

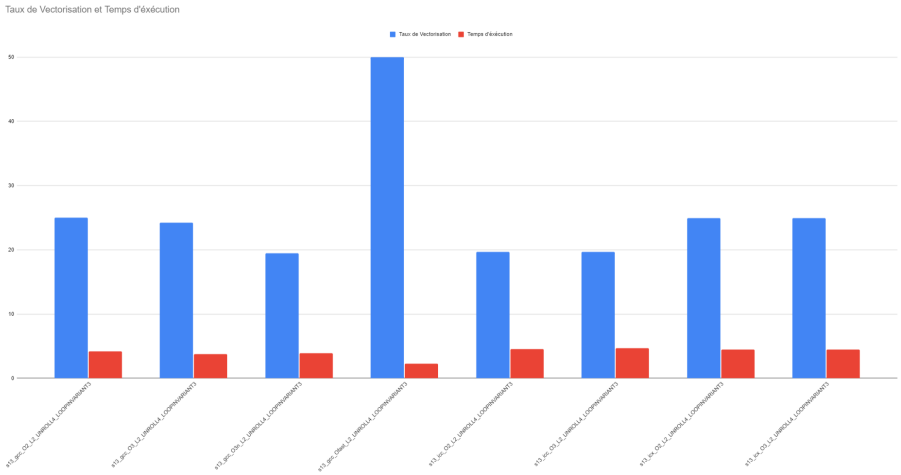
Le flag $-O3march = native$ pour gcc et $-O2$ $-O3$ pour icc ont énormement de mal à vectoriser les boucles.

5.3.4 Comparaison des temps d'exécution en déroulant que la boucle intérieur sous gcc selon les flags -O2 -O3 -OFast, icc selon -O2 -O3 et icx selon -O2 -O3



D'après le graphique ci-dessus nous nous rendons compte que le flag $-Ofast$ pour gcc a le temps d'éxécution le plus rapide. C'est icc $-O3$ avec le temps d'exécution le plus important.

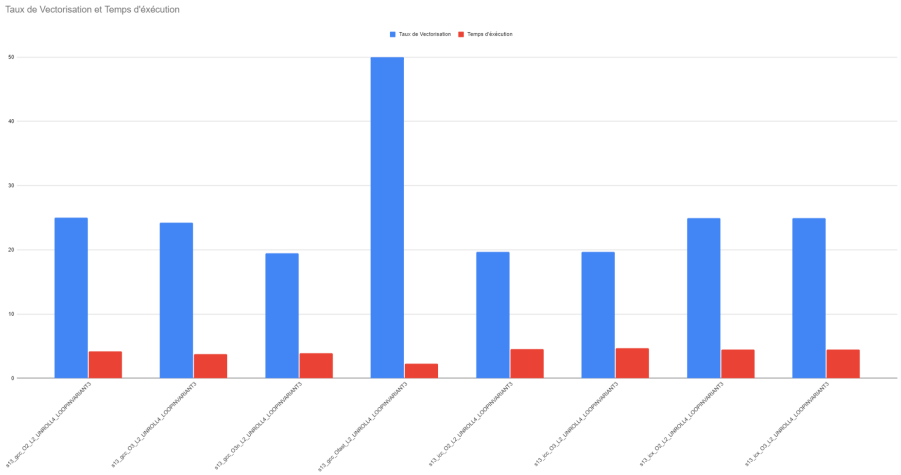
5.3.5 Comparaison de l'efficacité de vectorisation en déroulant les deux boucles par 4 sous gcc selon les flags -O2 -O3 -OFast, icc selon -O2 -O3 et icx selon -O2 -O3



D'après le graphique ci-dessus nous nous rendons compte que les flags $-Ofast$ pour gcc ont le taux de vectorisation le plus important lorsque la boucle intérieure est déroulée.

Le flag $-O3 march = native$ pour gcc et $-O2$ $-O3$ pour icc ont énormément de mal à vectoriser les boucles.

5.3.6 Comparaison des temps d'exécution en déroulant les deux boucles par 4 sous gcc selon les flags -O2 -O3 -OFast, icc selon -O2 -O3 et icx selon -O2 -O3



D'après le graphique ci-dessus nous nous rendons compte que le flag $-Ofast$ pour gcc on le temps d'exécution les plus rapides. C'est icc $-O3$ avec le temps d'exécution le plus important.

5.3.7 Résumé des analyses

	Taux de Vectorisation	Temps d'éxécution
s13_gcc_O2_L2_ORIGINAL	18,28	9,27
s13_gcc_O3_L2_ORIGINAL	18,28	9,24
s13_gcc_O3n_L2_ORIGINAL	16,02	8,31
s13_gcc_Ofast_L2_ORIGINAL	18,28	9,39
s13_icc_O2_L2_ORIGINAL	18,38	8,8
s13_icc_O3_L2_ORIGINAL	18,38	9,07
s13_icx_O2_L2_ORIGINAL	20,34	9,04
s13_icx_O3_L2_ORIGINAL	20,34	9,48

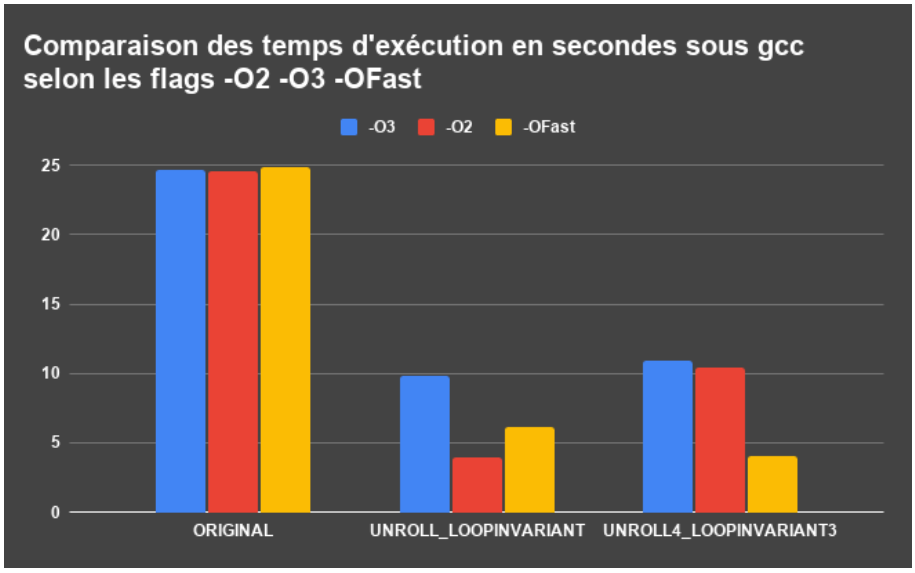
	Taux de Vectorisation	Temps d'éxécution
s13_gcc_O2_L2_UNROLL_LOOPINVARIANT	25	3,99
s13_gcc_O3_L2_UNROLL_LOOPINVARIANT	21,88	4,05
s13_gcc_O3n_L2_UNROLL_LOOPINVARIANT	17,86	4,02
s13_gcc_Ofast_L2_UNROLL_LOOPINVARIANT	50	1,69
s13_icc_O2_L2_UNROLL_LOOPINVARIANT	19,64	5,23
s13_icc_O3_L2_UNROLL_LOOPINVARIANT	19,64	5,26
s13_icx_O2_L2_UNROLL_LOOPINVARIANT	24,64	5,02
s13_icx_O3_L2_UNROLL_LOOPINVARIANT	24,64	5,07

	Taux de Vectorisation	Temps d'éxécution
s13_gcc_O2_L2_UNROLL4_LOOPINVARIANT3	25	4,19
s13_gcc_O3_L2_UNROLL4_LOOPINVARIANT3	24,22	3,78
s13_gcc_O3n_L2_UNROLL4_LOOPINVARIANT3	19,44	3,93
s13_gcc_Ofast_L2_UNROLL4_LOOPINVARIANT3	50	2,28
s13_icc_O2_L2_UNROLL4_LOOPINVARIANT3	19,64	4,54
s13_icc_O3_L2_UNROLL4_LOOPINVARIANT3	19,64	4,66
s13_icx_O2_L2_UNROLL4_LOOPINVARIANT3	24,91	4,47
s13_icx_O3_L2_UNROLL4_LOOPINVARIANT3	24,91	4,44

D'après ces photos les temps d'exécution de la fonction $ORIGINAL$ n'est pas du tout optimisé, on remarque lorsque que l'on déroule la boucle on gagne 50% du temps que l'on avait en $ORIGINAL$, et encore si on regarde particulierement $gcc -Ofast$ alors le temps est diviser par 5 pour $UNROLL\_UNVARIANT$ et de 10 pour $UNROLL4\_UNVARIANT3$. A temps égal icx arrive à mieux vectoriser les boucles quelque soit les flags utilisés et icc est mieux que gcc en $-O2$ et $-O3$ mais pas pour icx.

5.4 Analyses et mesures en L3 (Maxence BRINGUIER)

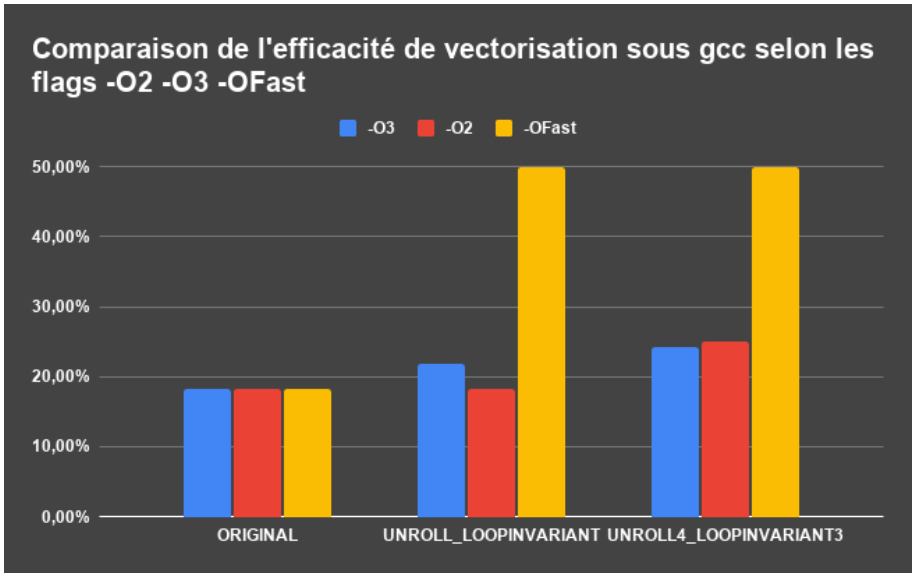
5.4.1 Comparaison des temps d'exécution sous gcc selon les flags -O2 -O3 -OFast



D'après le graphique ci-dessus nous nous rendons compte que les flags $-O3$ et $-OFast$ rendent les temps d'exécution au minimum deux fois plus rapide.

On se rend également compte que $-Ofast$ a plus de facilité d'exécution avec un double déroulage des boucles. Contrairement à $-OFast$, $-O3$ a plus de facilité avec un simple déroulage de boucle.

5.4.2 Comparaison de l'efficacité de vectorisation sous gcc selon les flags -O2 -O3 -OFast

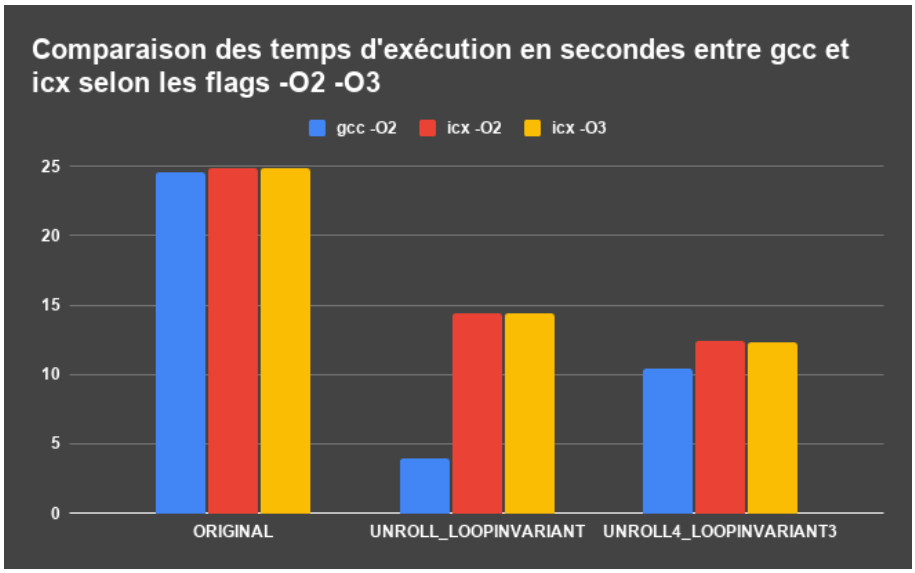


Sous $-OFast$ on se rend compte que 100% des boucles sont vectorisés quelque soit

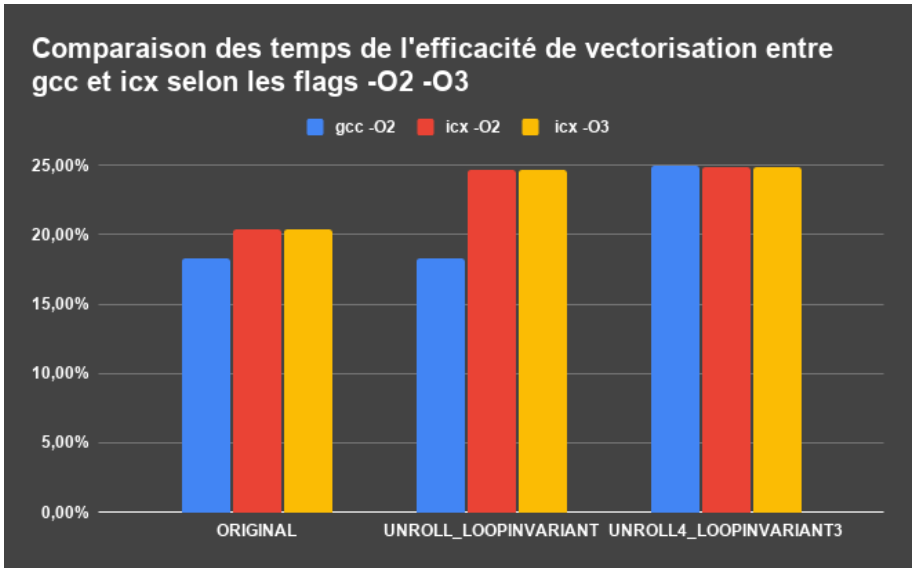
le déroulage donnée. On atteint donc une efficacité de vectorisation de 50%.

Les flags $-O2$ et $-O3$ ont énormement de mal à vectoriser les boucles, qu'elles soient déroulées ou non.

5.4.3 Comparaison des temps d'exécution entre icx et gcc avec les flags d'optimisation -O2 -O3



5.4.4 Comparaison de l'efficacité de vectorisation entre icx et gcc avec les flags d'optimisation -O2 -O3

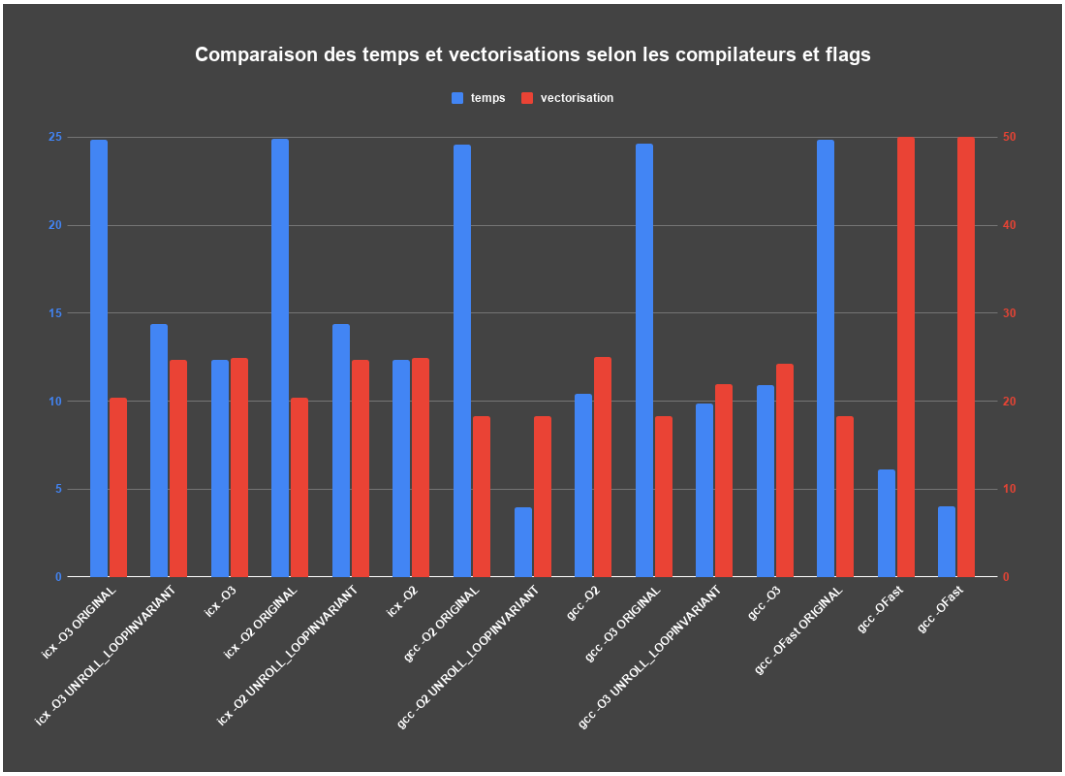


Les deux compilateurs génèrent un code assembleur similaire. Tout deux utilisent des instructions assembleurs XOR lors d'une exécution de la version $ORIGINAL$. *icx* continue à utiliser ces instructions contrairement à *gcc* qui les remplacent avec des

$MOVE$ comme vu précédemment.

De ce fait il n'y a que très peu de différences dans les résultats obtenus entre *gcc* et *icx* quelque soit le flag entre $-O2$ et $-O3$.

5.4.5 Résumé des analyses



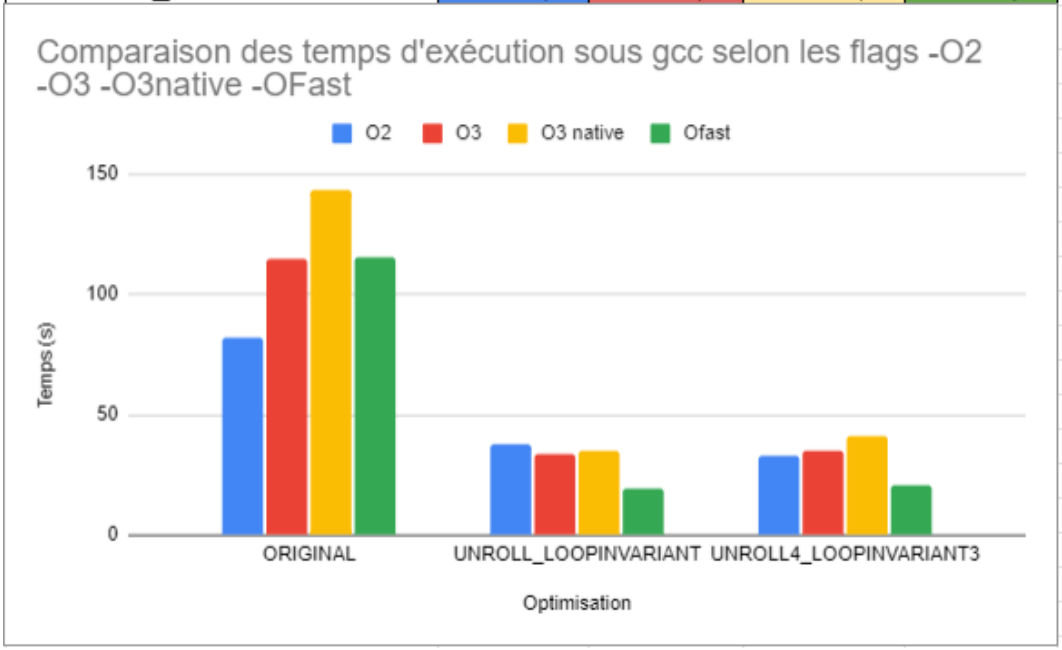
D'après ce graphique nous nous rendons bien compte à travers les temps d'exécution que le code source de la fonction $ORIGINAL$ n'est pas du tout optimisé.
A temps égal *icx* arrive à mieux vectoriser les boucles quelque soit les flags utilisés.

Le fait d'utiliser des flags plus "optimisés" tel que $-O3, -O3 march = native$ n'améliore pas forcément les performances. Certains flags inclus dans ceux cités précédemment doivent ralentir l'exécution car non utiles ou hors contexte dans notre cas.

5.5 Analyses et mesures en RAM (Maxime Vincent)

5.5.1 Comparaison des temps d'exécution sous gcc selon les flags -O2 -O3 -O3native -OFast

Optimisation	O2	O3	O3 native	Ofast
ORIGINAL	82,43	115,18	143,93	115,41
UNROLL_LOOPINVARIANT	37,71	33,74	35,36	19,58
UNROLL4_LOOPINVARIANT3	33,17	34,85	41,38	20,88

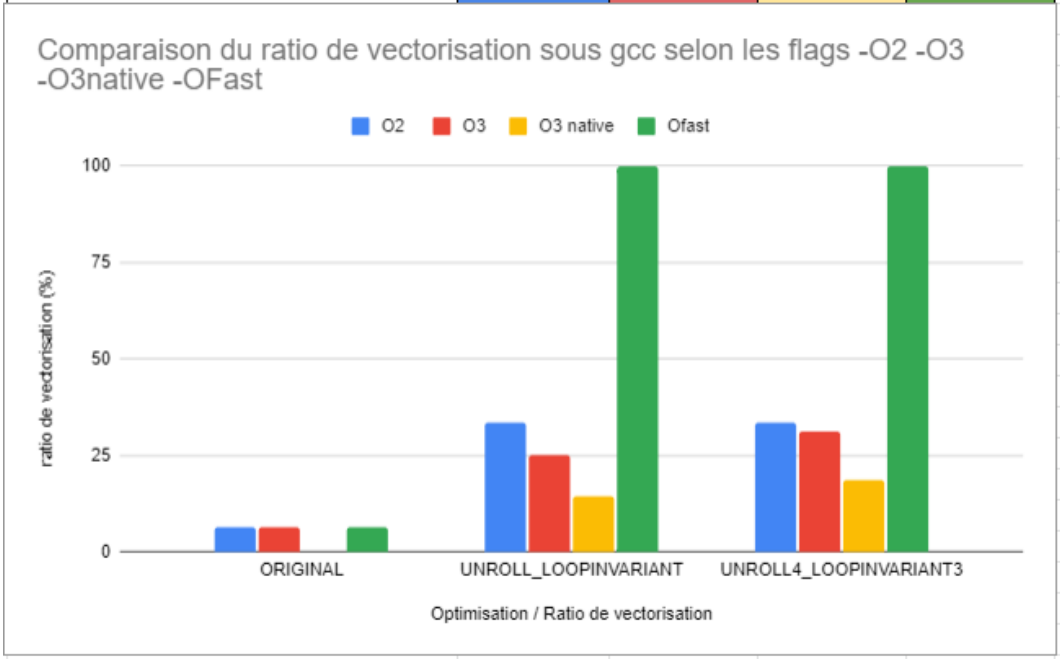


On observe sur le graphique ci-dessus que pour le kernel $ORIGINAL$ le temps d'exécution est plus rapide en O2.
Pour les kernels avec les boucles dérouler ($UNROLL\_LOOPINVARIANT$ et $UNROLL4\_LOOPINVARIANT3$) on remarque que avec l'option de compilation Ofast le temps d'exécution 1,5 fois plus rapide qu'avec O2, O3 et O3 native.

On constate aussi que pour tous les flags d'optimisation le temps d'exécution est bien plus rapide lorsqu'on déroule les boucles du kernel.

5.5.2 Comparaison du ratio et de l'efficacité de vectorisation sous gcc selon les flags -O2 -O3 -O3native -OFast

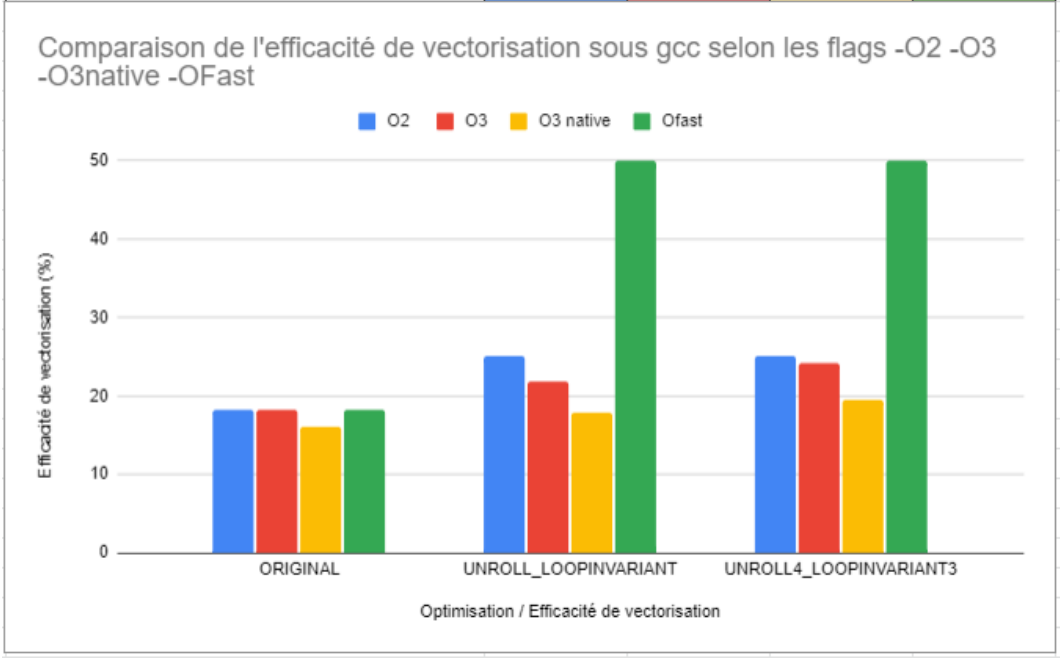
Optimisation / Ratio de vectorisation	O2	O3	O3 native	Ofast
ORIGINAL	6,35	6,35	0	6,35
UNROLL_LOOPINVARIANT	33,33	25	14,29	100
UNROLL4_LOOPINVARIANT3	33,33	31,25	18,52	100



On remarque que le ratio de vectorisation pour le kernel *ORIGINAL* est très bas peu import le flags de compilation, on constate même que pour le flag O3 native le ratio de vectorisation est de zero.

Pour les boucle dérouler on constate que le flags d'optimisation Ofast vectorise a 100% la boucle. On peut aussi remarquer que O2 vectorise un peu mieux que O3 et O3 native.s

Optimisation / Efficacité de vectorisation	O2	O3	O3 native	Ofast
ORIGINAL	18,28	18,28	16,02	18,28
UNROLL_LOOPINVARIANT	25	21,88	17,86	50
UNROLL4_LOOPINVARIANT3	25	24,22	19,44	50

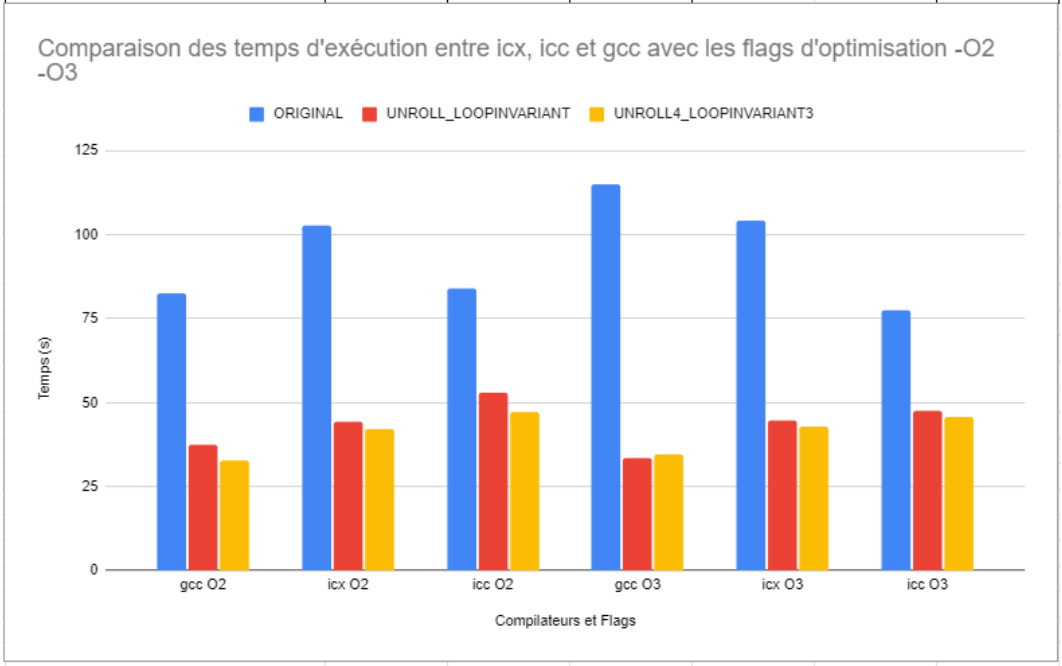


On remarque que pour le kernel *ORIGINAL* l'efficité de vectorisation est la meme peu importe le flag d'optimisation.

Pour les boucle vectorisé on constate que le flag Ofast permet d'avoir un efficacité de vectorisation de 50%

5.5.3 Comparaison des temps d'exécution entre icx, icc et gcc avec les flags d'optimisation -O2 -O3

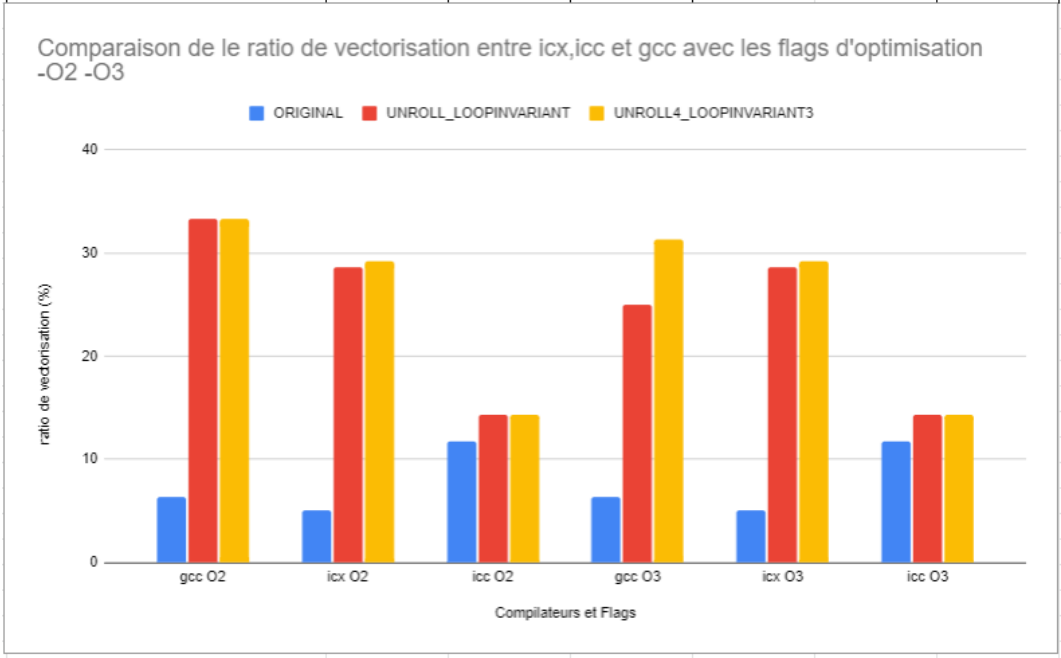
Optimisation	gcc O2	icx O2	icc O2	gcc O3	icx O3	icc O3
ORIGINAL	82,43	102,84	84,03	115,18	104,23	77,51
UNROLL_LOOPINVARIANT	37,3	44,34	53,05	33,41	44,77	47,62
UNROLL4_LOOPINVARIANT3	32,89	42,12	47,36	34,46	42,89	45,85



On observe pour le kernel $ORIGINAL$ qu'avec le flag d'optimisation $O2$ le temps d'execution est le plus rapide avec gcc et icc. Pour le flag $O3$ on observe que icc est plus rapide.
Pour les version dérouler du kernel on observe dans un premier temps que la version avec les deux boucle dérouler est plus rapide que la version avec seulement une boucle dérouler. Au niveau des compilateur on observe que gcc est toujours plus rapide que icc et icx avec les version $UNROLL$ du kernel

5.5.4 Comparaison de le ratio de vectorisation entre icx,icc et gcc avec les flags d'optimisation -O2 -O3

Optimisation	gcc O2	icx O2	icc O2	gcc O3	icx O3	icc O3
ORIGINAL	6,35	5,05	11,76	6,35	5,05	11,76
UNROLL_LOOPINVARIANT	33,33	28,57	14,29	25	28,57	14,29
UNROLL4_LOOPINVARIANT3	33,33	29,2	14,29	31,25	29,2	14,29



Pour la version $ORIGINAL$, on observe que icc vectorise plus que gcc et icx. Pur les autres version on constate au contraire que gcc et icx vectorise mieux que icc.
On peut donc en déduire que gcc et icx gère mieux le déroulage de boucle que icc.

5.5.5 Résumé des analyses

Nous avons pu observer que peut importe le compilateur ou le flags d'optimisation utilisé la version $ORIGINAL$ et tojours moins rapide et moins vectorisé que les version $UNROLL$.

Nous avons aussi constaté que le flags $Ofast$ permet d'avoir le meilleur ratio de vectorisation (100%) avec gcc.

Concernant les compilateurs nous avons pu remarqué que icc vectorise mieux nativement que gcc et icx mais gère moins bien le déroulage de boucle que gcc et icx.
En conclusion la meilleur optimisation est le kernel $UNROLL4\_LOOPINVARIANT3$ avec le compilateur *gcc* et le flags $Ofast$.

Source Location	Source Function	Level	Coverage (%)	Time (s)	Vectorization Ratio (%)	Vectorization Efficiency (%)
s13_gcc_OFast -	s13	Innermost	97.92	20.44	100	50

5.6 Conclusion

Après analyses et comparaisons des différentes exécutions nous nous rendons compte que *icx* a plus de facilités à vectoriser les boucles en contre partie d'une perte de temps

à l'exécution.
Ce phénomène parait logique car *icx* a été conçu pour des architectures intel © et est donc optimisé pour.

Cependant les meilleurs résultats obtenus ont été avec $gcc - Ofast$. Le compilateur arrive a vectorisé l'ensemble des boucles tout en gardant un temps d'exécution généralement meilleur qu'avec d'autres options.

6 Parallélisation multi-threads

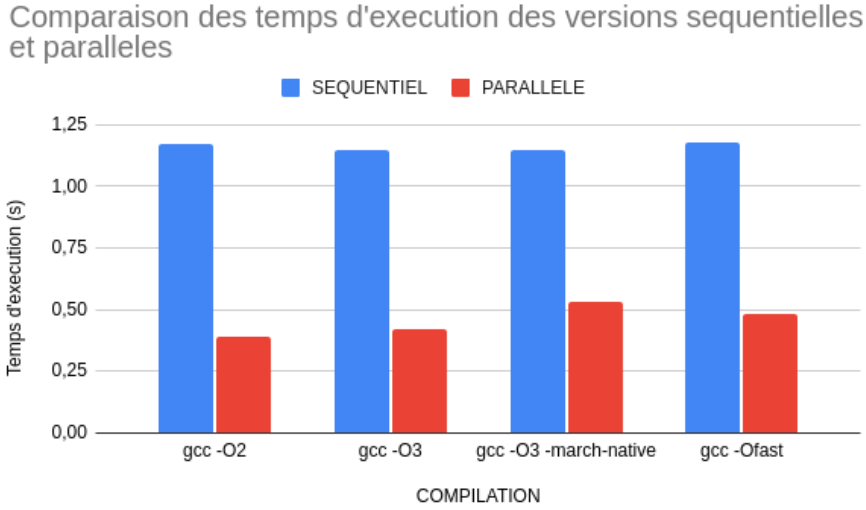
6.1 Analyses faites en commun (et présentation des optims)

Nous avons fait le choix de travailler avec openMP.
Une optimisation évidente du code consiste a paralléliser les itérations de boucle. De plus une itération ne dépend d'aucune autre donc il est très simple de mettre en place la parallélisation.

Nous avons donc, pour chaque version, parallélisé la boucle interne comme suit :

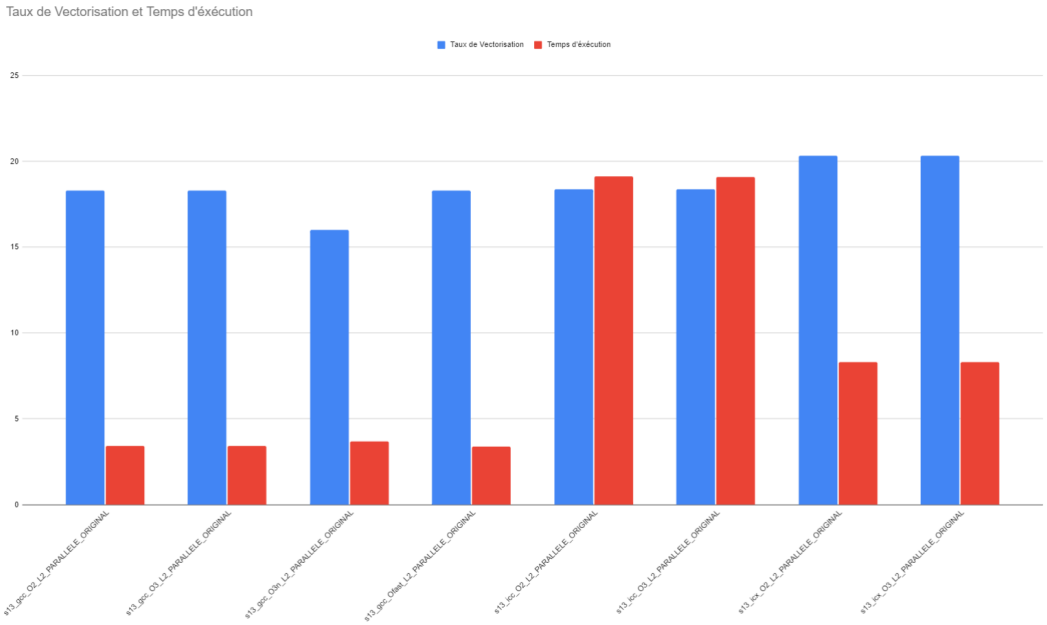
```
#pragma omp parallel for num_threads(4) collapse(2)
for(){
    for(){
        /* contenu de la boucle */
    }
}
```

6.2 Analyses et mesures en L1 (Olivier BENABEN)

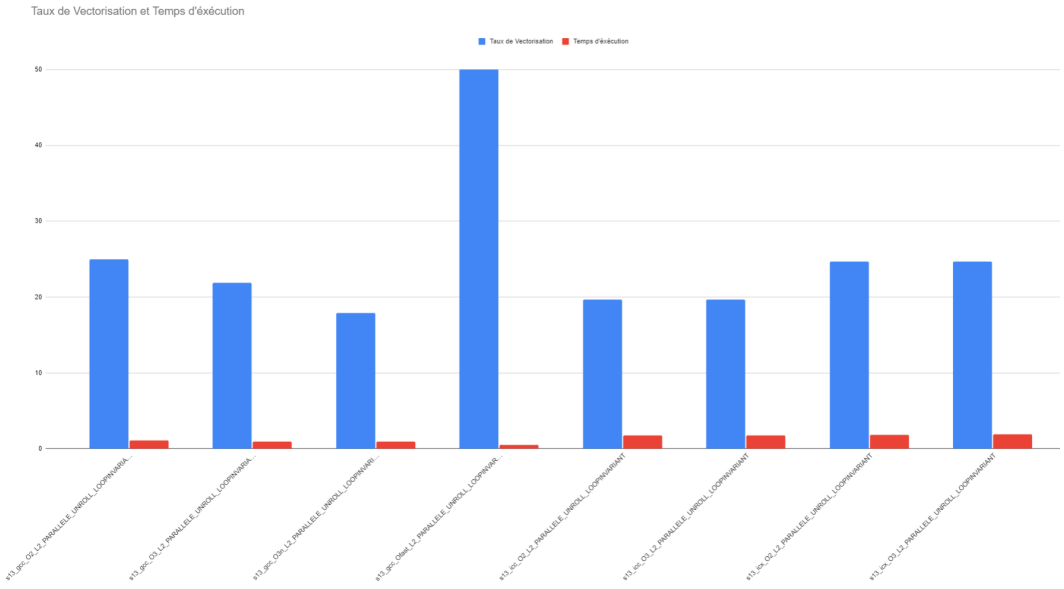


La parallélisation nous propose un gain de temps d'exécution très agréable. Ici on parallélise sur 4 threads, le programme est donc théoriquement quatre fois plus rapide étant donne qu'il traite 4 opération en même temps, résultat qui se vérifie sur l'histogramme ci dessus.

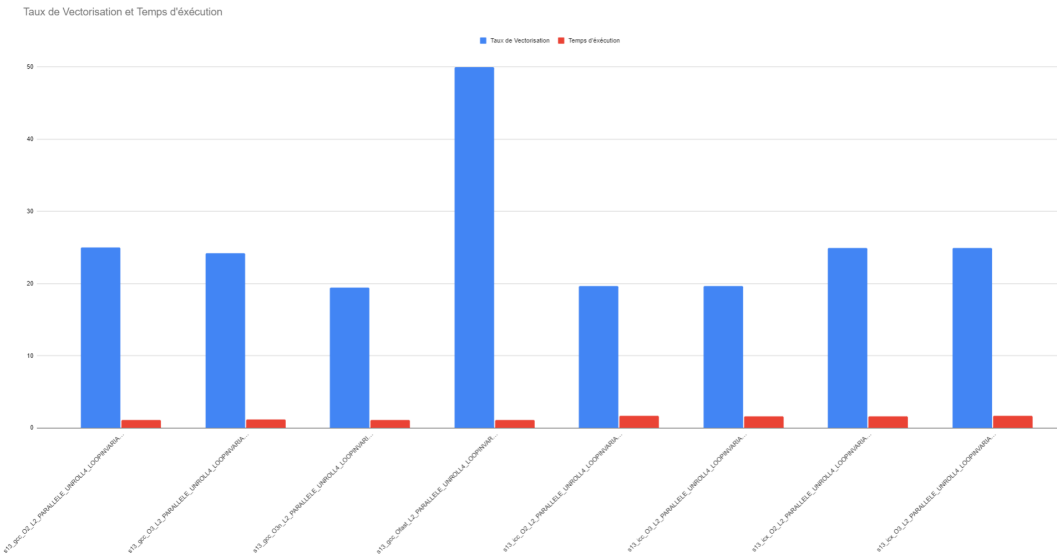
6.3 Analyses et mesures en L2 (Hugo HENROTTE)



On remarque que lorsque on essaie de paralléliser les boucles mais qu'on ne les déroule pas alors le compilateur icc n'apprécie pas et on obtient des temps d'exécution qui font le double de ceux obtenu sans la parallélisation.



D'après le graphique ci-dessus nous nous rendons compte que le flag $-Ofast$ pour gcc on le temps d'exécution les plus rapides..

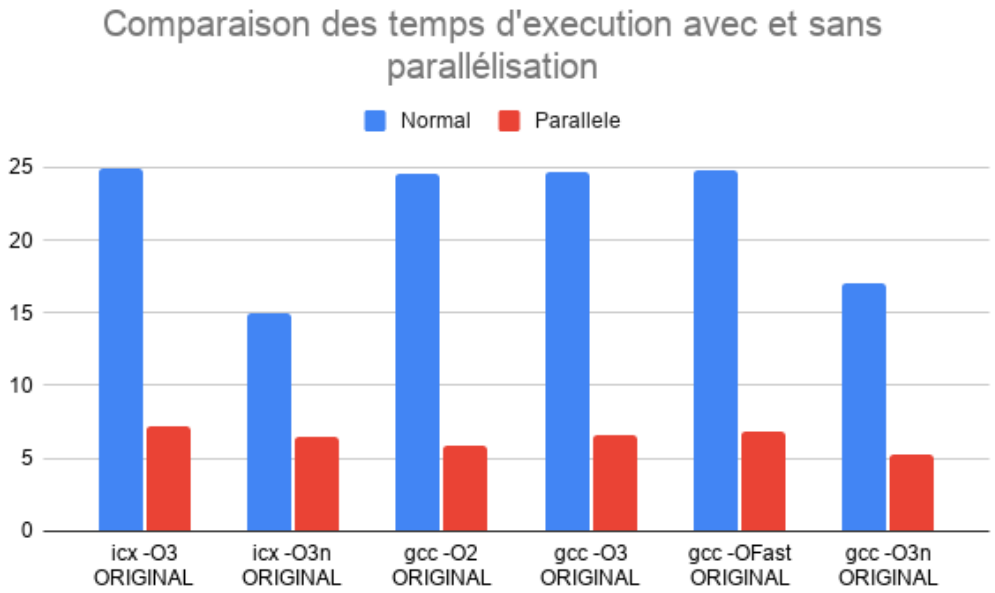


D'après le graphique ci-dessus nous nous rendons compte que le flag $-Ofast$ pour gcc a le temps d'exécution le plus rapide.

Les performances sont bien meilleurs que les résultats non parallélisés sauf pour icc quand les boucles ne sont pas déroulées, on peut noter que les résultats sont meilleurs avec la parallélisation (en moyen le temps d'exécution est 3 fois plus rapide).

6.4 Analyses et mesures en L3 (Maxence BRINGUIER)

Seuls les temps d'exécution sont censés changer car nous parallélisons les itérations.

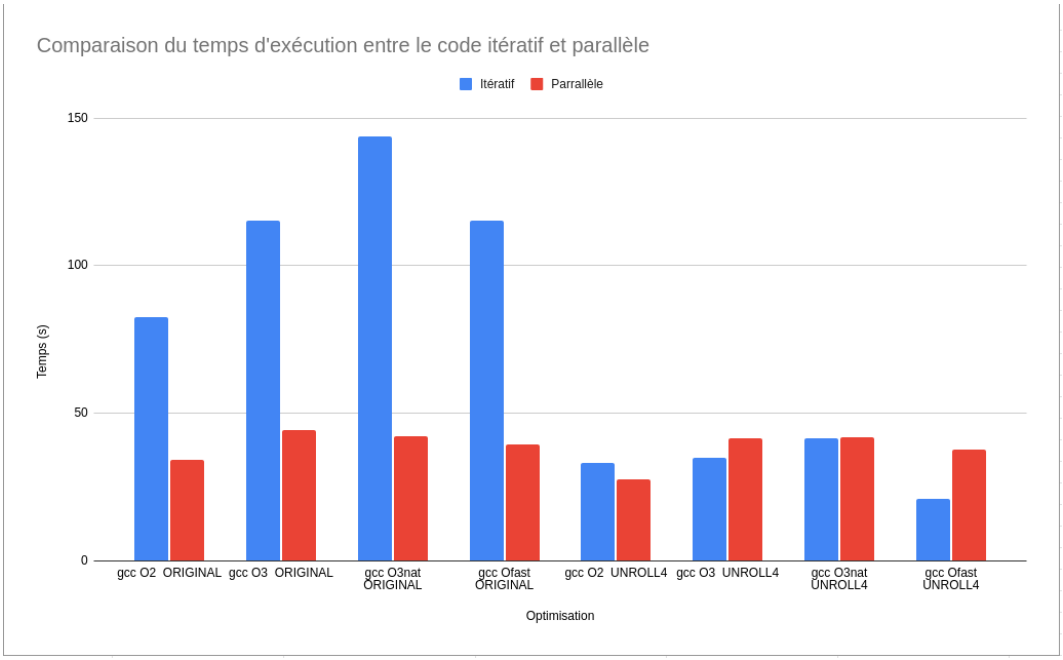


Nous nous rendons compte que le fait de paralléliser le code entre différents threads permets un gain de temps considérable. Cela paraît logique car l'ensemble des calculs sont exécutés simultanément.

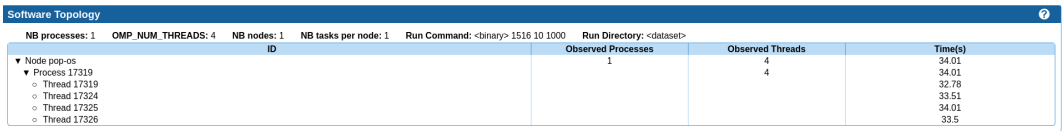
Dans les versions déroulées, l'écart de temps est moins notable entre exécution parallèle et séquentiel, cependant les temps d'exécution sont tout de même meilleurs.

6.5 Analyses et mesures en RAM (Maxime VINCENT)

6.5.1 Comparaison du temps d'exécution entre le code itératif et parallèle



Dans un premier temps on observe que le kernel $ORIGINAL$ avec OMP_PARALLELE s'exécute plus rapidement avec que celui sans. Au contraire on remarque que le temps d'exécution sur le kernel $UNROLL$ est a peu prêt équivalant avec ou sans OMP_PARALLELE.



On peut voir dans maqao (image ci-dessus) que le temps d'exécution est répartie sur les différents threads lors de l'exécution .

6.6 Conclusion

Dans cette partie nous avons pu observer que la parallélisation permet de répartir le temps d'exécution du processus sur différents threads.
Nous avons observé que sur le kernel $ORIGINAL$, la parallélisation permet d'améliorer grandement le temps d'exécution (environ 2 à 3 fois plus rapide). Mais nous avons aussi constaté que le parallélisation sur le kernel déroulé n'améliore pas le temps d'exécution. Cela s'explique car le ratio de vectorisation du code augmente, et donc un processus sans thread est aussi rapide voir plus qu'un processus avec 4 threads.

7 Conclusion

Ce projet nous a permis de nous rendre compte que l'implémentation des boucles principales d'un programme est très important pour sa performance ; spécifiquement en utilisant GCC, qui semble très réactif aux optimisations tels que le loop unrolling, if hoisting, etc.. comparé aux compilateurs plus axés sur la performance par défaut (où une optimisation de la part du programmeur peut déboussoler sa compréhension du code).

En effet ceux-ci arrivent à optimiser le code sans ces procédés car ils s'adaptent automatiquement au processeur afin de fournir les meilleurs résultats possibles ; ils utilisent notamment de nombreuses instructions spécifiques.

Le fait de paralléliser notre code est également un aspect non négligeable d'optimisation. Les temps d'exécution sont largement meilleurs dans ce cas là.

Nous nous sommes également rendu compte que quelque soit le cache ou RAM utilisé les optimisations restaient relativement similaires.
