UVSQ université PARIS-SACLAY | ISTY Institut des Sciences et Techniques des Yvelines CAMPUS DE MANTES EN YVELINES CAMPUS DE SAINT-QUENTIN-EN-YVELINES

Rapport phase 1
étude de cas AOA sujet X

23/03/2025

Norman ALIE (gcc)
Souheila BENABID GUENDOUZI (clang)
Mathieu AKOUN (icx)

Table des matières

Introduction

```
# include < math .h > // sqrt

float baseline ( unsigned n ,
                 const double a [ n ][ n ] ,
                 const float b [ n ]) {
        unsigned i , j ;
        float s = 0.0;

        for ( j =0; j < n ; j ++)
                for ( i =0; i < n ; i ++)
                        s += a [ i ][ j ] / sqrt ( b [ i ]);

        return s ;
}
```

Le sujet n°5 est une somme pondérée des valeurs d'une matrice double a par la racine carrée des éléments du vecteur float b. On parcourt une double boucle *for* dans laquelle on va sommer le résultat de la division des éléments de a par la racine carrée des éléments de b.

Le travail à effectuer est le suivant:

- Mesurer, pour une multitudes de tailles n, le nombre de cycles RDTSC effectués lors de l'exécution du code compilé en O2 et tracer un graphe de ces valeurs en fonction de la taille choisie lors de la mesure.
- Mesurer pour une unique taille n, les différents nombres de cycles RDTSC selon l'optimisation choisie lors de la compilation (O2, O3, O3 + march=native…) et tracer un graphe comparant ces différentes valeurs.

1 Travail commun

1.1 Stratégies utilisées en commun pour améliorer la stabilité

Lors de la prise en main des outils mis à disposition, nous avons effectué une multitude de runs de calibrage afin de déterminer pour chacun quel était le nombre correct de répétitions de warm up, de mesure pour une taille n au départ fixée.

1.2 Autres travaux faits en commun

Nous avons effectué des calculs d'estimation de taille utile utilisée par le programme de par les types de données utilisés (matrice en double et vecteur en float) pour un nombre d'occurrences donné(taille n). Nous en sommes arrivés à estimer que le calcul suivant nous permettrait de déterminer les tailles de tableau n pour lesquelles on se situerait dans les caches L1, L2, L3 ou dans la RAM:

$T(n) = 8*n^2 + 4*n$. Soit le nombre de cases de la matrice (chaque case de type double occupe 8 octets en mémoire) et le nombre de cases du vecteur (chaque case de type float occupe 4 octets en mémoire).

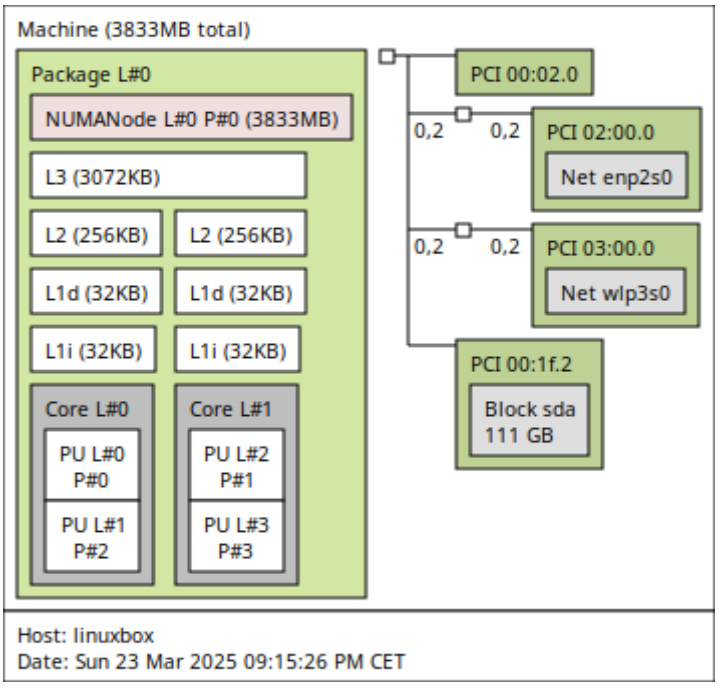
2 Analyse et performance avec gcc (Norman ALIE)

2.1 Environnement expérimental

Propriété	Valeur
Modèle de processeur	"Intel(R) Core(TM) i3-4160T CPU @ 3.10GHz"
Génération processeur / micro-architecture	Haswell
DVFS (pilote, gouverneur, réglages)	intel_cpufreq, performance, fmax=3.10 GHz
Linux	Linux Mint 22.1 based on Ubuntu 22.4
Virtualisation	None
Version noyau	6.8.0-51
Version GCC	13.3.0 (Ubuntu 13.3.0-6ubuntu2~24.04)
Version MAQAO	2.21.1 - a9bb74606c7e260f7f87782e64f16b00a6f20c1b::20250121-112452
Caches	64kio * 2 / 512kio * 2 / 3Mio
Autre	

L’architecture Haswell supporte les instructions AVX2 qui supportent la vectorisation sur 256bits plutôt que 128bits en AVX. Cependant, la taille des caches est relativement faible par rapport aux processeurs récents et il ne supporte pas l’AVX-512.

Une bonne compilation et un programme bien écrit devraient déjà permettre de profiter de la vectorisation cependant la taille des caches risque d'être un facteur limitant et il pourrait être intéressant d’utiliser du tiling pour subdiviser les matrices et ainsi limiter les accès à la RAM.



2.2 Choix taille des tableaux

Notre programme s'exécute en mono-cœur. On ne tiendra pas compte de la répartition des caches.

En analysant le code source de notre kernel, on a pu déduire que la taille du jeu de donnée à stocker en fonction de n est tel que:

$$T(n) = n^2 \times 8 + n \times 4 \text{ octets.}$$

En effet, on manipule une matrice de double (8 octets) et un tableau de float (4 octets). La matrice est de taille n*n et le tableau de taille n.

Cela nous permet de calculer les valeurs de n suivantes:

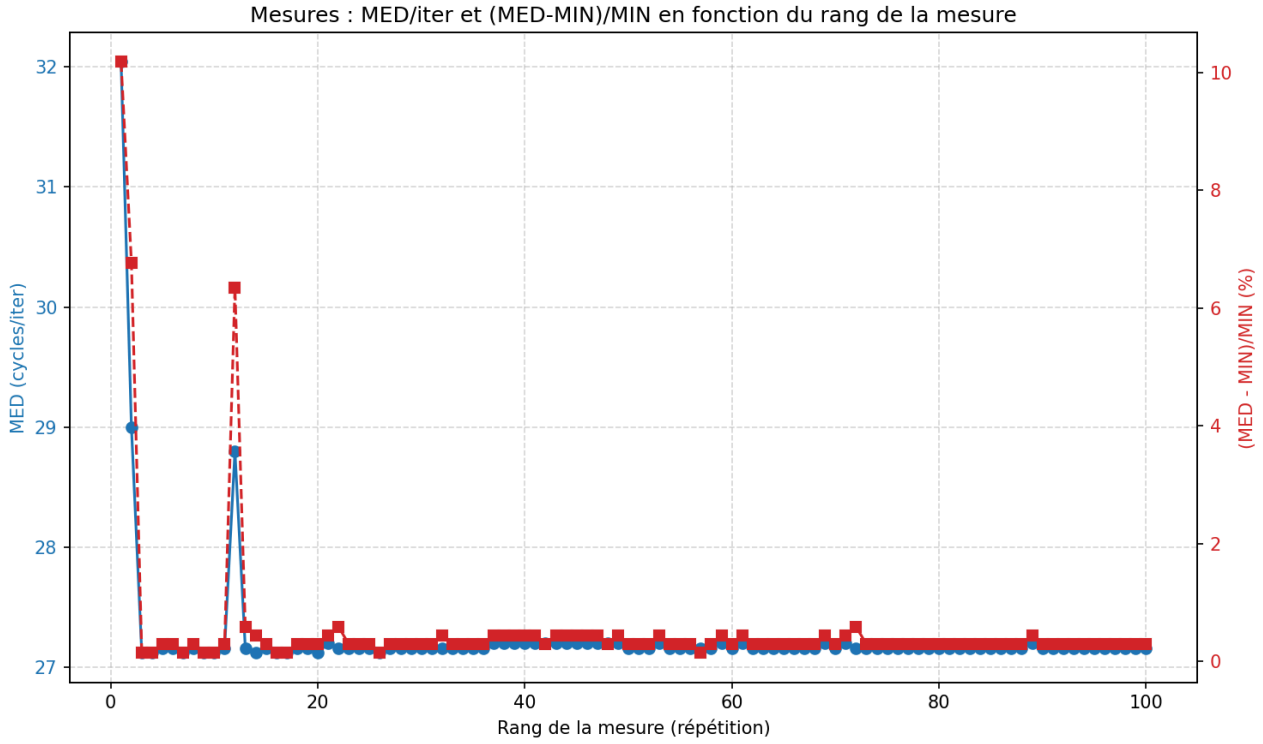
- $n > 90 \Rightarrow T(n) > L1$
- $n > 255 \Rightarrow T(n) > L2$
- $n > 626 \Rightarrow T(n) > L3$

On commencera donc nos mesure un peu en dessous de L1, ici à n=30 pour finir nos mesure à n=3535 afin de s’aligner sur les mesures des autres machines.

2.3 Détermination du nombre de répétitions de warmup et de mesure

Etalonnage du warmup L1

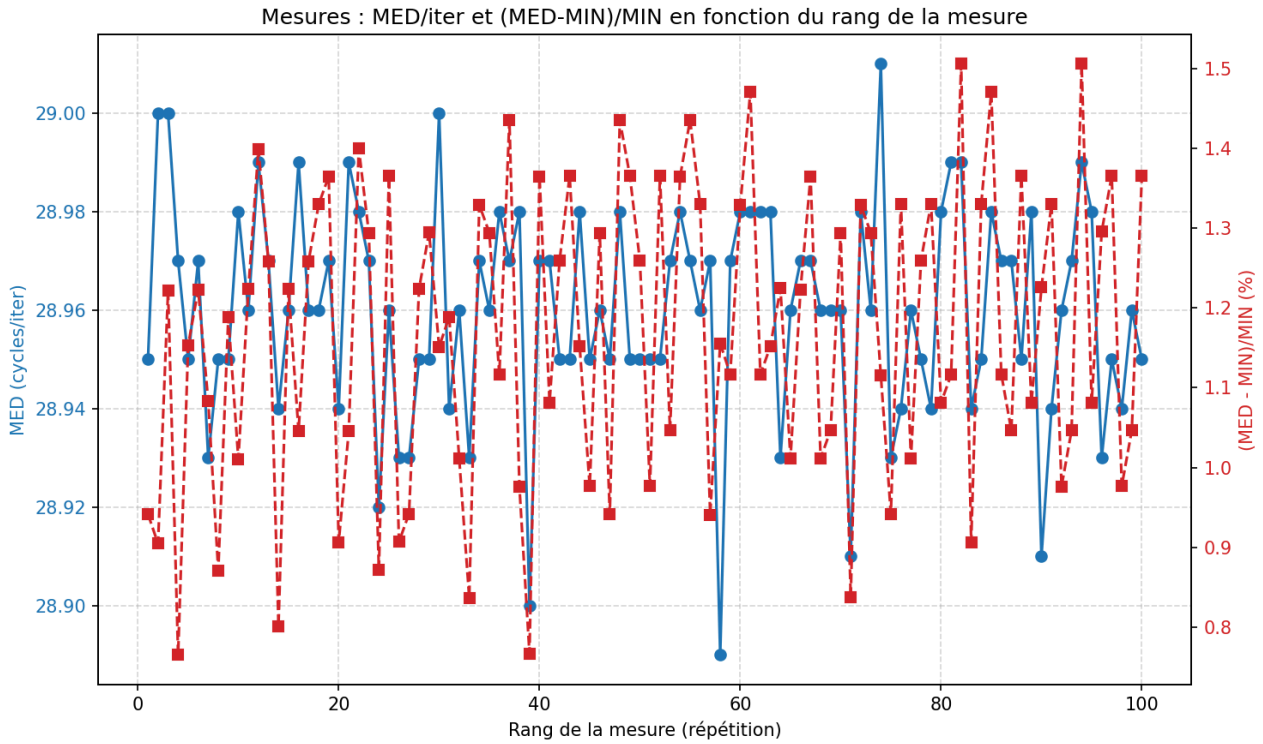
Pour l'étalonnage en L1 on utilisera un n de 10.



Selon Figure 2, on peut constater que la mesure devient stable à partir d'un warmup de 3 avec du bruit lors de la mesure 12 et décider de fixer le nombre de warmups à 3.

Etalonnage du warmup L3

Pour tenir en L3 sans déborder dans la RAM on choisit un n=300.



Selon Figure 3, on peut constater que malgré une apparence plus erratique que la mesure en L1 en réalité la stabilité est bonne dès le départ et on observe uniquement un bruit négligeable de l'ordre d'un demi pourcent. et décider de fixer le nombre de warmups à 1.

2.4 Mesures en O2, O3, Ofast : L1 jusqu'à RAM

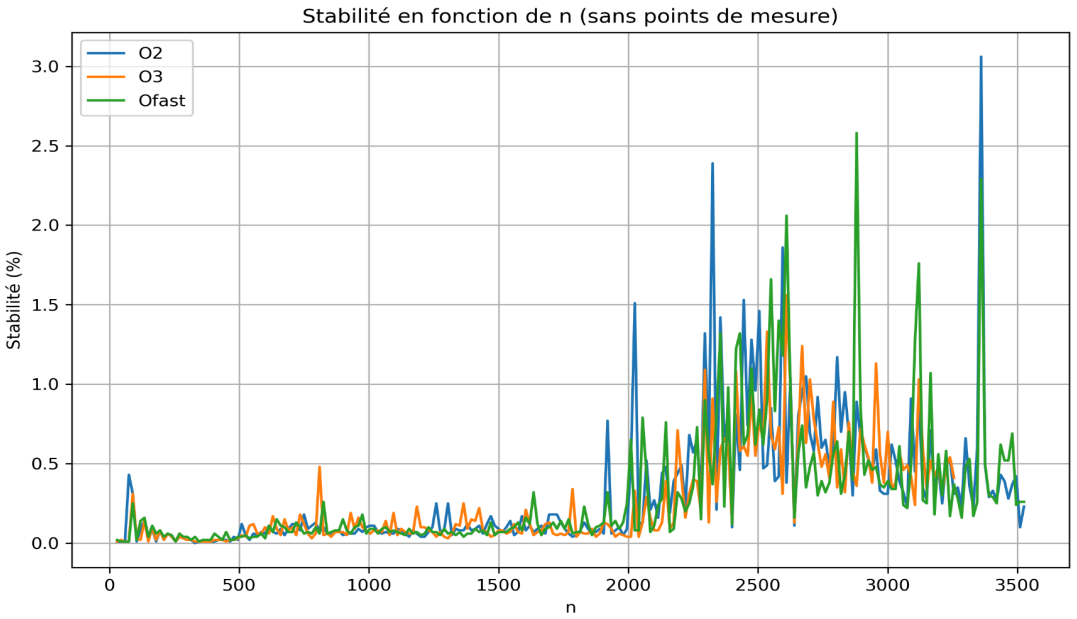


Figure 4: on remarque une diminution de la stabilité quand n dépasse 2000 (32Mo)

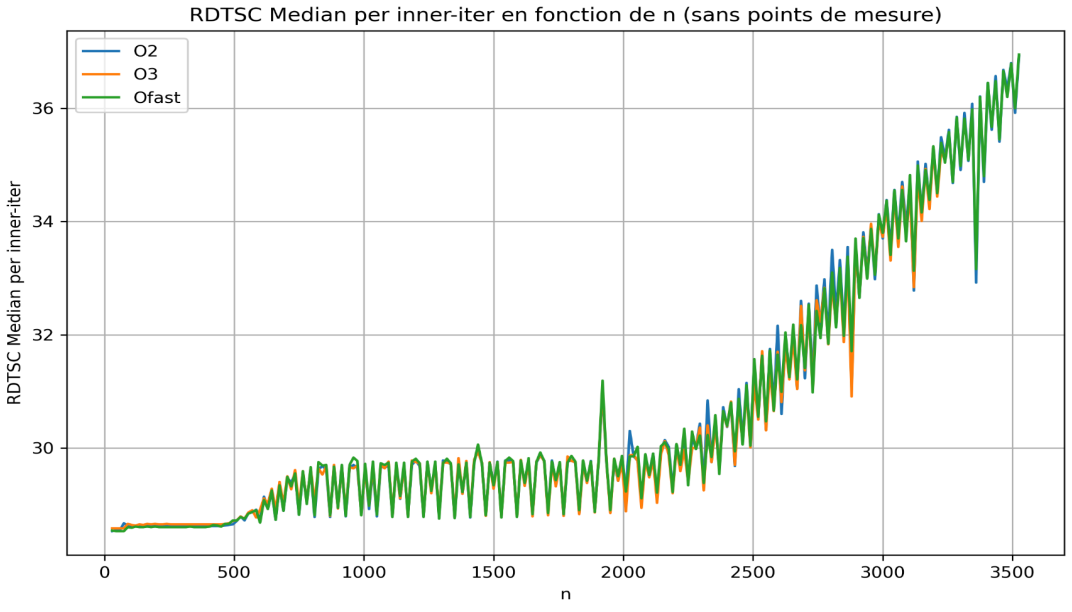


Figure 5: De même, le RDTSC augmente à partir de n>2000 environ.

Selon Figure 4 et 5 on peut constater que les mesures en O2, O3 et Ofast sont très similaires et que le fonctionnement avec le jeu de données en cache L1, L2 ou L3 n'implique pas de grandes différences.
Le passage en RAM est très marqué, surtout sur la figure 5 où on remarque un plateau pour n>600. Ce plateau, moins marqué, est aussi présent sur la stabilité (figure 4).

Un phénomène intéressant est cette courbe croissante pour n>2000 (environ 32 Mo de données). Cela pourrait s'expliquer par la faible quantité de RAM installée (4 Go) et un système qui commence à swapper.

Plus n augmente, plus la pression mémoire devient importante et plus le système swap, ce qui expliquerait cette croissance. De plus, en lançant les mesures avec le swap désactivé, le système devient instable et ne peut pas exécuter toutes les mesures. Il pourrait être intéressant de relancer les mesures avec plus de RAM.

2.5 Explorations des options de compilations

On a testé la compilation avec Ofast et march=native mais la différence était mineure par rapport à Ofast comme on peut le voir sur la figure suivante:

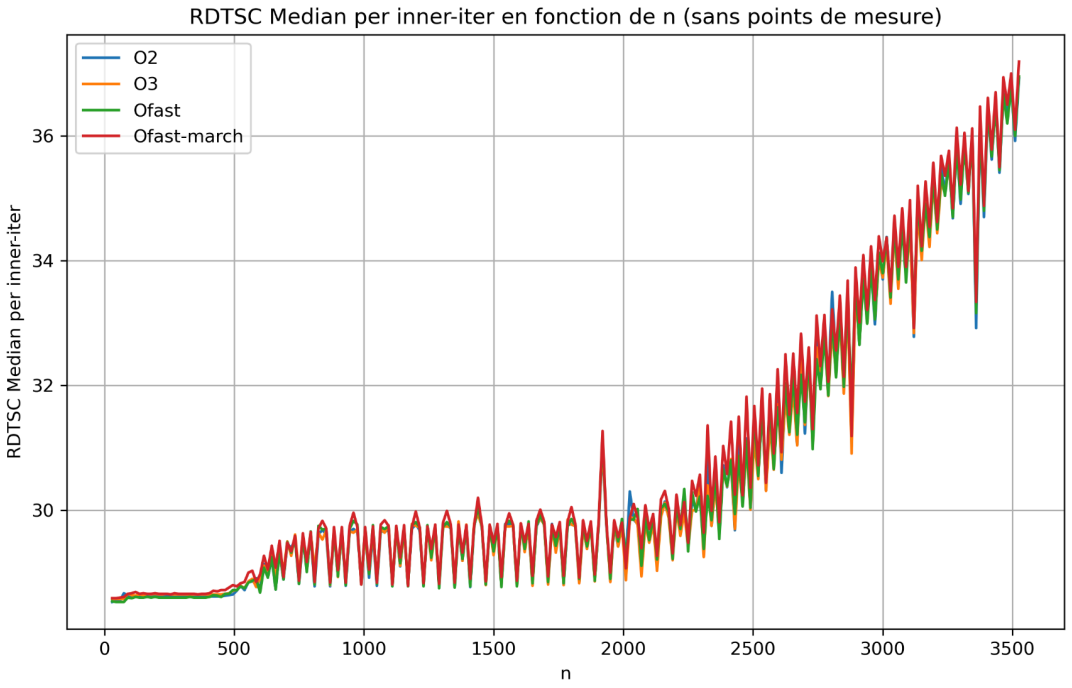
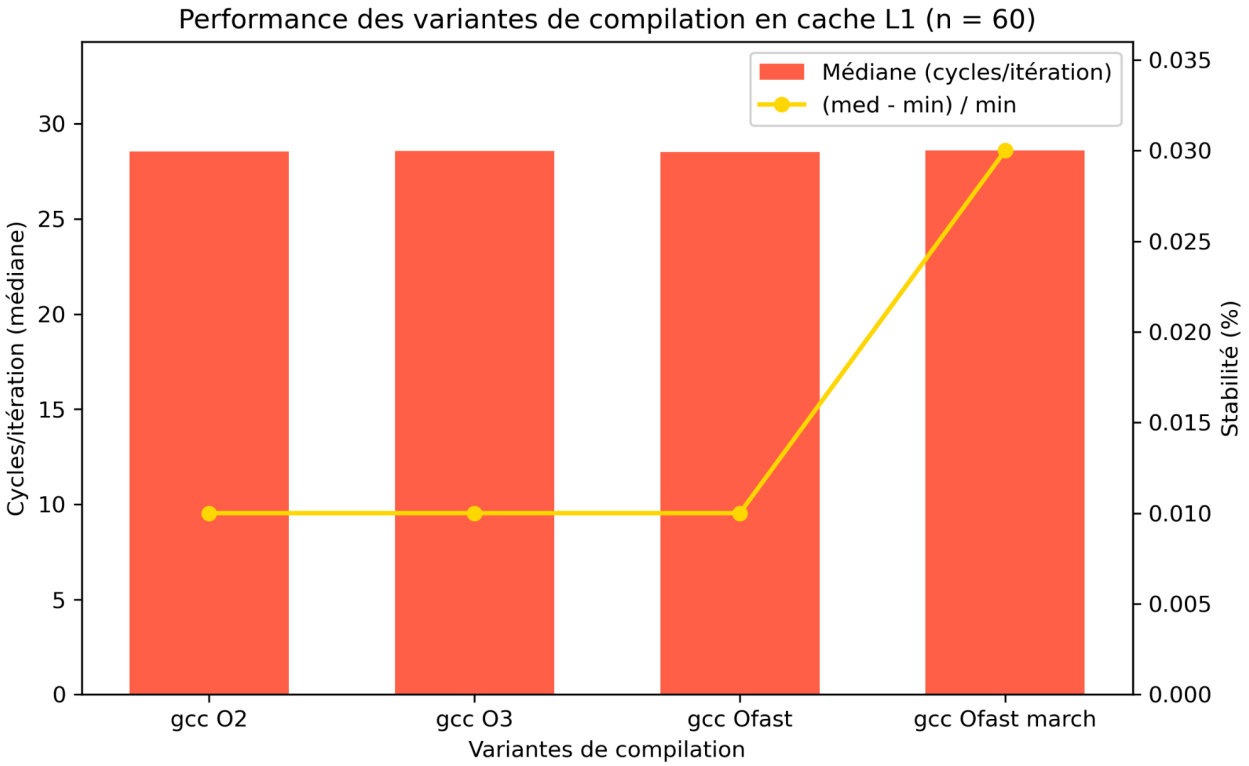


Figure 6: Comparaison O2, O3, Ofast et Ofast + march=native

2.6 Mesures en L1



Selon Figure 7, on peut remarquer que les options de compilation n’affectent pas les performances lorsque le jeu de données est stocké en L1.

2.7 Mesures en L3

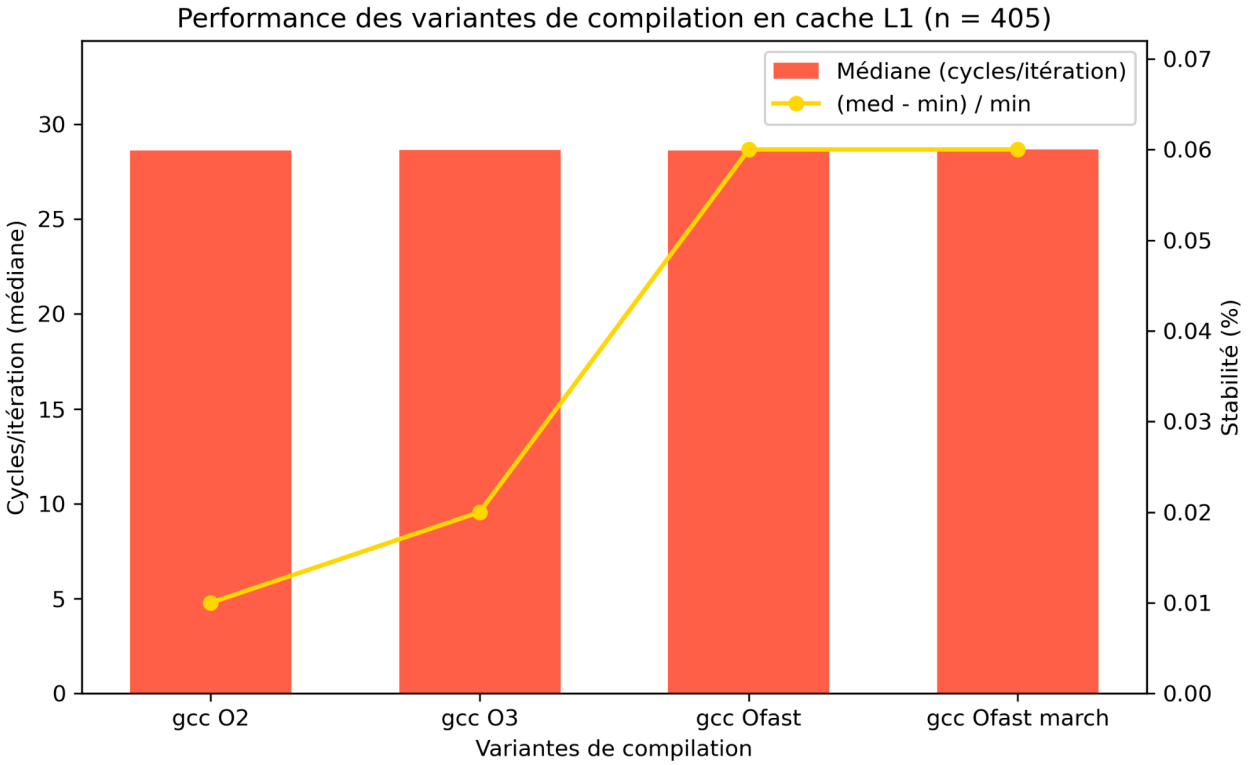


Figure 8: De même, lorsque le problème est en L3, aucune différence notable entre les options de compilation.

2.8 Différences entre variantes, analyse...

Cette section sert à expliquer les différences (ou ce qui est identique…) entre les variantes de compilation, et donc ce qui a pu s'observer sur les mesures en L1 et en L3. On veut voir ici des comparaisons entre codes assembleurs et rapports/métriques MAQAO.

(ci-dessous exemple à remplacer par votre propre boucle)

flag	gcc -O2	gcc -O3	gcc -Ofast -march=native
chargement	movsd, cvtss2sd, cvtsd2ss	movsd, cvtss2sd, cvtsd2ss	vmovsd, vcvtss2sd, vcvtsd2ss
calcul	sqrtsd, divsd, addsd	sqrtsd, divsd, addsd	vsqrtsd, vdivsd, vaddsd

On peut constater que les instructions scalaires (avec le suffixe SS) ont été remplacées par des instructions vectorielles (préfixées v) dans la version -Ofast -march, permettant ainsi d'exploiter le jeu d'instructions AVX pour un meilleur débit.

Autres comparaisons à partir de rapports MAQAO:

	gcc -O2	gcc -O3	gcc -Ofast -march=native	gcc -O3 -march=native
Vector len use	25%	25%	23%	23.44%
Vectorization ratio	11.44%	11.44%	0%	5.88%
Active time in kernel	13.43s	13.43s	13.43s	13.40s

On observe que le ratio de vectorisation est plus faible avec O3 et Ofast mais les performances sont tellements proches entre les différents flags de compilation que ces valeurs n'ont pas trop d'interet.

3 Performance avec clang (Souheila BENABID)

3.1 Environnement expérimental

Propriété	Valeur (exemple)
Modèle de processeur	AMD Ryzen 5 PRO 5650U with Radeon Graphics
Génération processeur / micro-architecture	Zen 3 (Architecture "Cezanne")
DVFS (pilote, gouverneur, réglages)	`amd_pstate`, `performance`, `fmin=1.6 GHz`, `fmax=2.3 GHz` *(d'après lscpu, la fréquence max est 2.3 GHz, mais elle peut monter avec le Boost)*
Linux	Linux Mint 21.2
Virtualisation	Linux Natif
Version noyau	5.15.0-131-generic
Version Clang utilisée	17.0.6
Version MAQAO	2.21.1
Autre	
Autre	

- **Mon processeur** est un **AMD Ryzen 5 PRO 5650U**, basé sur **l'architecture Zen 3**. Il supporte plusieurs **jeux d'instructions avancés**, notamment **AVX2**, **FMA**, et **SSE4.2**, qui sont utilisés pour la vectorisation et l'optimisation des performances.

Côté mémoire cache : *figure 1*

- Chaque cœur a **32 Ko de cache L1 Data (L1d)** et **32 Ko de cache L1 Instructions (L1i)**.
- Chaque cœur a **512 Ko de cache L2** (au total **3 Mo pour les 6 cœurs**).
- Tous les cœurs partagent **16 Mo de cache L3**, ce qui est assez grand et favorise les accès rapides aux données.

Contrairement aux architectures **Intel Haswell** par exemple, **Zen 3 a un cache L3 beaucoup plus grand (16 Mo)**, ce qui peut impacter les performances des optimisations mémoire.

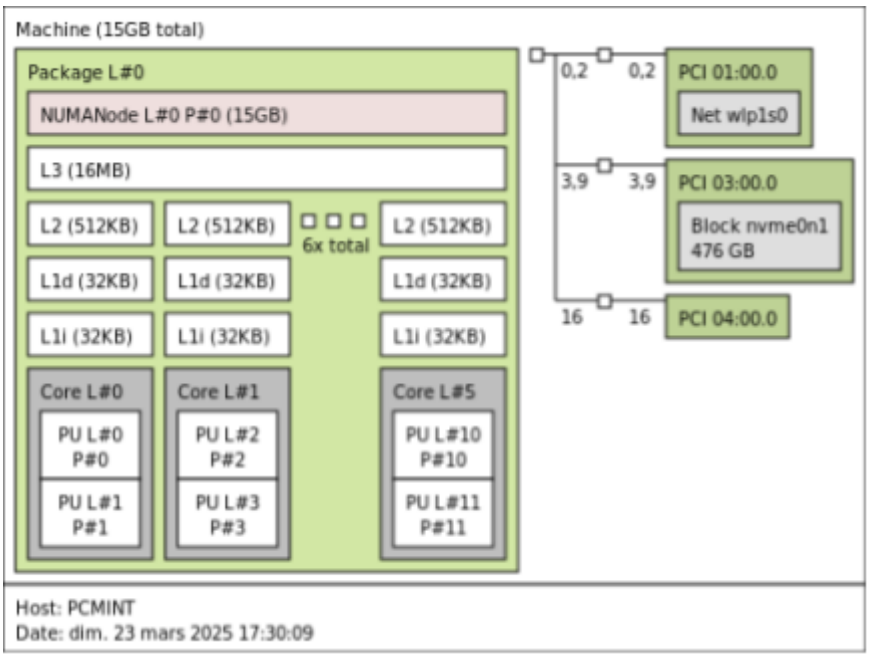


figure 1: topologie CPU

3.2 Choix taille des tableaux

Comme dit précédemment, notre programme s'exécute en mono-cœur. On ne tiendra donc pas compte de la répartition des caches entre les différents cœurs.
En analysant le code source de notre kernel, nous avons déterminé que la taille du jeu de données à stocker en fonction de n est donnée par :

T(n)=n^2×8+n×4 octets

En effet :
La matrice a[n][n] est stockée en double (8 octets par élément).
Le vecteur b[n] est stocké en float (4 octets par élément).

En fonction des tailles de caches de mon AMD Ryzen 5 PRO 5650U, nous avons calculé les seuils suivants :

Cache	Taille (Kio)	Valeur n correspondante
L1	192 Kio	n > 120
L2	3072 Kio (3 MiB)	n > 392
L3	16384 Kio (16 MiB)	n > 1448

Nous commencerons donc nos mesures un peu en dessous de L1, ici à n = 100, pour terminer au-delà de L3, ici à n = 1500.

4 Performance avec icx (Mathieu AKOUN)

4.1 Environnement expérimental

Propriété	Valeur (exemple)
Modèle de processeur	11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
Génération processeur / micro-architecture	11th Gen (Tiger_Lake) / Willow Cove
DVFS (pilote, gouverneur, réglages)	`intel_pstate`, `performance`, `fmin=400MHz`, `fmax=4.2GHz`
Linux	Linux Mint 22
Virtualisation	Dual-boot
Version noyau	6.8.0-53-generic
Version ICX utilisée	2025.0.4
Version MAQAO	2.21.1
Caches	192 Kib/ 5 Mib/ 8 Mib
Autre	

Cet environnement est constitué de 4 coeurs physiques et de 8 threads avec:

L1D: **48 KB** par coeur

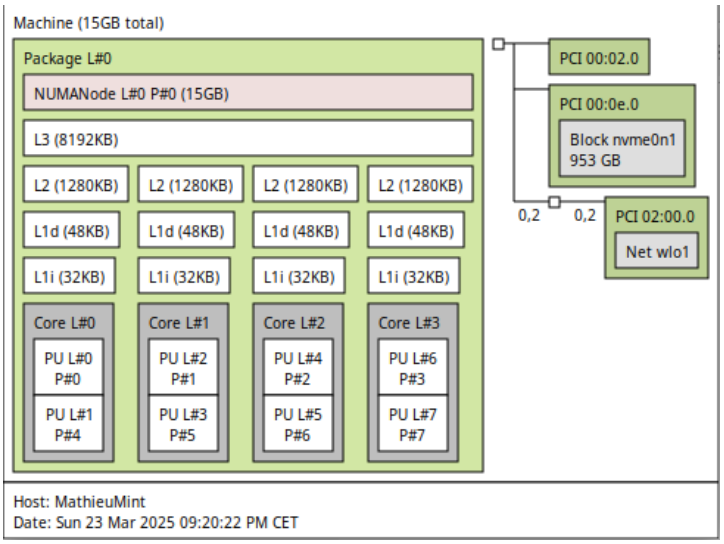
L2: **1,25 MB** par coeur

L3: **8 MB** partagés (améliore le multithreading en stockant les données communes à tous les cœurs).

Tiger-Lake possède plusieurs extensions SIMD avancées telles que:

- AVX2: Vectorisation sur 256 bits pour les entiers et les flottants (comparé à 128 sur AVX)
- FMA: Accélération des opérations de multiplication et d'addition en une seule instruction
- AVX2-512: Permet (partiellement) la vectorisation sur 512 bits.

Sur Figure 1 on peut voir que notre processeur a 4 coeurs, chacun avec un cache L1D de 32 KB et L2 de 256KB. Un cache L3 de 6 MB est partagé entre ces coeurs.



4.2 Choix des tailles du tableau

Notre programme s'exécute en mono-cœur. On ne tiendra pas compte de la répartition des caches.

En analysant le code source de notre kernel, on a pu déduire que la taille du jeu de donnée à stocker en fonction de n est tel que:

$$T(n) = n^2 \times 8 + n \times 4 \text{ octets.}$$

En effet, on manipule une matrice de double (8 octets) et un tableau de float (4 octets). La matrice est de taille n*n et le tableau de taille n.

Cela nous permet de calculer les valeurs de n suivantes:

- $n > 156 \Rightarrow T(n) > L1$
- $n > 810 \Rightarrow T(n) > L2$
- $n > 1024 \Rightarrow T(n) > L3$

On commencera donc nos mesure un peu en dessous de L1, ici à n=148 pour finir nos mesure à n=3535 afin de s’aligner sur les mesures des autres machines.

5 Conclusion

Pour cette étude de cas, différents environnements de compilation ont été exploités afin d'analyser les performances d'un programme. Des ensembles de données de tailles variés et différents degrés d'optimisation ont été employés pour identifier les changements de performance. L'objectif était de déterminer l'impact des cycles RDTSC sur diverses dimensions de tableaux et d'évaluer les variations de résultats obtenus en fonction des niveaux d'optimisation (O2, O3, Ofast, march=native,..).

Les relevés effectués au niveau L1 ont démontré que les choix de compilation n'ont pas engendré de variations notables en termes de performance, alors qu'au niveau L3, des variations minimes ont été constatées. Néanmoins, une augmentation des temps de cycle RDTSC a été observée lorsque les volumes de données dépassent 2000 éléments (n > 2000), phénomène pouvant être attribué à une insuffisance de mémoire RAM disponible, conduisant ainsi à des accès au disque (swap).

Une découverte importante concerne l'utilisation de la vectorisation. Malgré le remplacement des instructions scalaires par des instructions vectorielles dans la version d'optimisation "march=native", l'efficacité de la vectorisation n'a pas atteint les attentes, notamment avec des paramètres comme "Ofast" qui ont révélé des performances proches de celles obtenues avec l'option O2, tant en matière de vectorisation qu'en termes de vitesse d'exécution. Cela souligne l’importance d’une optimisation bien ciblée, prenant en compte les spécificités matérielles et les caches.

En conclusion, ces tests ont permis de mieux comprendre l’impact des options de compilation sur les performances, mais aussi les limitations des architectures, notamment en termes de capacité mémoire. Ce projet a renforcé la compréhension des optimisations à appliquer dans des contextes spécifiques et l'importance d’ajuster les tests et les compilations en fonction du matériel utilisé pour maximiser les gains de performance.