Université de Versailles - Saint-Quentin-en-Yvelines 2022-2023

Outils de Base du HPC

TD2 programmation C et mesures de performances

Corentin BEAULIEU

Sommaire

1 Présentation de la machine **1**
1.1 Processeur . . . 1
1.2 Cache . . . 1
1.2.1 Cache L1D . . . 1
1.2.2 Cache L2 . . . 1
1.2.3 Cache L3 . . . 1
1.3 Mémoire principale . . . 1
1.4 Logiciels . . . 1
1.4.1 Système d'exploitation . . . 1
1.4.2 Compilateurs . . . 1
1.4.3 Bibliothèque . . . 1

2 Mesures du produit matriciel **2**
2.1 Comparaison des compilations . . . 2
2.2 Comparaison des versions . . . 3
2.3 Conclusion . . . 4

3 Mesures du produit scalaire **4**
3.1 Comparaison des compilations . . . 4
3.2 Comparaison des versions . . . 6
3.3 Conclusion . . . 6

4 Mesures de la réduction **7**
4.1 Comparaison des compilations . . . 7
4.2 Comparaison des versions . . . 8
4.3 Conclusion . . . 8

5 Conclusion **9**

1 Présentation de la machine

Tous les résultats qui seront présentés ont été obtenu sur une machine avec les caractéristiques suivantes

1.1 Processeur

Modèle	µarch	f min	f nominale	f max	driver	nb cores	boost
Intel Core i5-4690	Haswell	800 MHz	3500 MHZ	3500 MHz	intel_cpufreq	4	OFF

1.2 Cache

Ce processeur a des caches qui ont les caractéristiques suivantes

1.2.1 Cache L1D

Taille total	Taille Ligne	Partage	Associativité	Type
32 Kio	64 o	par core	8 chemins	données

1.2.2 Cache L2

Taille total	Taille Ligne	Partage	Associativité	Type
256 Kio	64 o	par core	8 chemins	unifié

1.2.3 Cache L3

Taille total	Taille Ligne	Partage	Associativité	Type
6144 Kio	64 o	partagée entre tous les cores	12 chemins	unifié

1.3 Mémoire principale

Taille	Nombre	Taille total	Type	Vitesse	Largeur	Form factor
8 Gio	2	16 Gio	DDR3 synchronous	1600 MT/s	64 o	DIMM

1.4 Logiciels

1.4.1 Système d'exploitation

Le système utilisé est une distribution *Linux* basée sur *ArchLinux*. Le noyau est dans la version *6.0.7-arch1-1*.

1.4.2 Compilateurs

Compilateur	version	état
gcc	12.2.0	pleinement fonctionnel
clang	14.0.6	pleinement fonctionnel
icc	non installé	non installé
icx	2022.2.0.20220730	non fonctionnel

Étant donné que *icx* ne fonctionne pas bien qu'il soit installé, il ne sera pas utilisé pour les mesures de performance qui seront présentées par la suite.

1.4.3 Bibliothèque

Bibliothèque	version
cblas	3.10.1-1
mkl	2022.3.0.8767-1

Étant donné que *icx* n'est pas fonctionnel, la *mkl* ne sera pas présentée dans les résultats.

Toutes ces informations ainsi que d'autres ont été obtenues en exécutant le script *arch.sh* contenu dans le dossier *Scripts*. Le résultat de ce script est un fichier contenant toutes les informations nécessaires à la présentation de la machine. Vous pouvez retrouver celui généré au moment de la mesure dans le dossier *Rapport/Resultats* sous le nom *arch-info.txt*.

Pour assurer plus de stabilité de toutes nos mesures, la fréquence du processeur a été fixée à 3.5 GHz via la commande `cpupower frequency-set` et le gouverneur sélectionné est *userspace*. De plus, le boost a été désactivé. Enfin, l'exécution du programme mesuré a été confinée au core 1 grâce à la commande `taskset`.

Toutes les implémentations accomplissent des calculs flottants en double précision. Les structures de données représentant des matrices et des vecteurs ont été remplies de façon aléatoire. Les mesures ont été faites en utilisant la `clock_monotonic_raw` du processeur. Nos mesures se concentrent exclusivement sur les calculs d'algèbre linéaire, à cet effet, les allocations, remplissages et libérations de nos structures de données ainsi que nos écriture des résultats sont faits en dehors de toute mesure.

2 Mesures du produit matriciel

Les mesures ont été obtenu en lançant le programme avec $n = 128$ et $r = 33$. C'est-à-dire que les matrices multipliées sont des matrices 128×128 flottants double précision. Cette taille a été choisie car elle permet un temps de calcul assez long pour avoir une mesure précise tout en étant une puissance de 2 ce qui permet aux versions déroulées d'être dans les conditions les plus favorables.

Pour le produit matriciel, nos expériences nous donnent les résultats suivants :

2.1 Comparaison des compilations

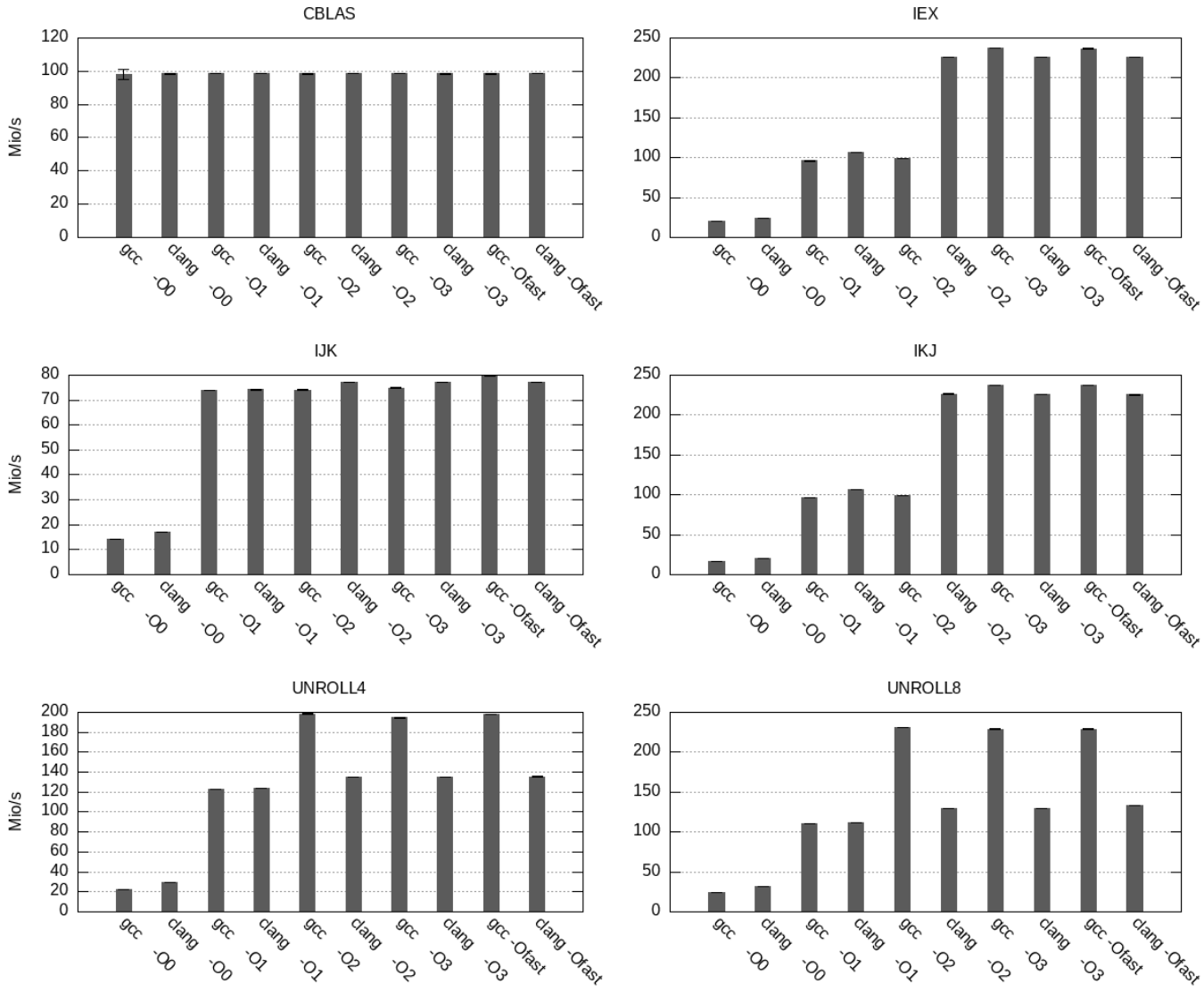

Fig. 1 – Performance de différentes version d'une dgemm en fonction du compilateur

CBLAS Cette version appelle directement la bibllioth`eque *cblas*. Pour la version *cblas*, on observe une très grande stabilité de la performance peu importe le compilateur ou le niveau d'optimisation choisi. On peut expliquer cela par le fait que cette version appelle une fonction d'une bibliothèque externe déjà compilée. Par conséquent, le traitement du calcul n'est pas influencé par le compilateur ou les options d'optimisation choisies.

IEX Cette version comporte une inversion des boucles internes pour obtenir un accès prioritaire en colonne et la déclaration constante dans celles-ci de l'élément non modifié de la première matrice. Pour la version *iex*, on observe une amélioration de performance de cette version en fonction de l'option de compilation choisie. Plus précisément, pour *O0* les deux compilateurs (gcc et clang) donnent des performances équivalentes. On retrouve le même constat pour *O1* bien qu'on observe que la version de clang est légèrement plus performante. On a une accélération de $\times 4$ entre *O0* et *O1*. On observe ensuite que les performances ne changent pas en passant de *O1* à *O2* pour gcc mais qu'il y a une accélération de plus de $\times 2$ pour clang. Enfin, pour *03* et *0fast*, on a des performances assez proche pour gcc et clang de l'ordre de celles obtenues avec clang en *O2*. gcc donne néanmoins des performances un peu meilleures. On peut donc penser que clang applique des optimisations en *O2* que gcc n'applique pas avant *O3*.

IJK Cette version est une implémentation naïve suivant la définition mathématique du produit matriciel. Pour la version *ijk*, on observe que les deux compilateurs restent toujours assez proches l'un de l'autre en terme de performances. La version compilée en *O0* est environ $7\times$ moins bonne que les autres. Toutes les autres configurations ont des performances assez proches même si *Ofast* est légèrement plus performante que les autres. On peut toutefois noter que les performances des versions de clang sont les mêmes à partir de *O2*, là où gcc donne des versions légèrement améliorées jusqu'à *Ofast*.

IKJ Cette version comporte une inversion des deux boucles plus internes du produit matriciel. Pour la version *ikj*, on peut faire les mêmes observations que pour la version *iex*. Ceci est assez logique étant donné que ces deux versions ne diffèrent que par le l'utilisation d'une constante dans la boucle intermédiaire du calcul. Ainsi, on peut penser que les compilateurs font les mêmes transformations au même flags d'optimisation.

UNROLL4 Cette version reprends les transformation de *iex* et y ajoute un déroulage de la boucle la plus interne avec un facteur de quatre. Pour la version *unroll4*, on voit que les versions de gcc et clang sont très proches pour *O0* et *O1*. On voit une accélération de $\times 6$ environ entre ces deux options de compilation. clang a une légère amélioration de *O1* à *O2* puis les performances restent identiques pour *O3* et *Ofast*. Pour gcc, on observe la même chose mais l'amélioration est beaucoup plus prononcée avec environ 60% d'augmentation. On peut penser que pour le déroulage de boucle avec un facteur 4, gcc arrive à appliquer de meilleures transformations que clang.

UNROLL8 Cette version reprends les transformation de *iex* et y ajoute un déroulage de la boucle la plus interne avec un facteur de huit. Pour la version *unroll8*, on peut faire à peu près les mêmes observations que pour la version *unroll4*. Ceci nous amène à penser que sans options précises, clang n'arrive pas à faire des transformations aussi efficaces que gcc lors d'un déroulage de la boucle la plus interne d'un produit matriciel.

2.2 Comparaison des versions

Tout d'abord, on observe que *cblas* a de meilleures performances que toutes les autres versions lorsque l'optimisation est au niveau zéro.

Ensuite, au niveau d'optimisation un, on voit que le déroulage avec un facteur quatre devient la version la plus performante avec les deux compilateurs. Toutes les autres versions se rapprochent de la version de *cblas* qui reste constante comme on a pu le voir précédemment.

Le passage du niveau un au niveau deux d'optimisation double quasiment la performance de la meilleure version qui est le déroulage en facteur huit pour gcc et les versions *iex* et *ikj* pour clang.

Pour les deux compilateurs, le niveau trois et *fast* amènent peu de changements en performance pour les versions déroulées, *cblas* et *ijk*. De plus, on voit que clang n'a plus aucune réelle augmentation de la performance des toutes les versions du niveau deux au niveau *fast*. Enfin, pour gcc, les versions *iex* et *ikj* deviennent les versions les plus performantes de toutes au niveau trois.

Pour conclure, on peut noter la mauvaise performance de *cblas* et *ijk* peu importe le compilateur. On observe que pour les deux compilateurs et toutes les versions, sauf *cblas*, le passage du niveau zéro à un apporte une réelle amélioration. Finalement, on voit que pour le plus haut niveau d'optimisation, les version *iex* et *ikj* sont les meilleures.

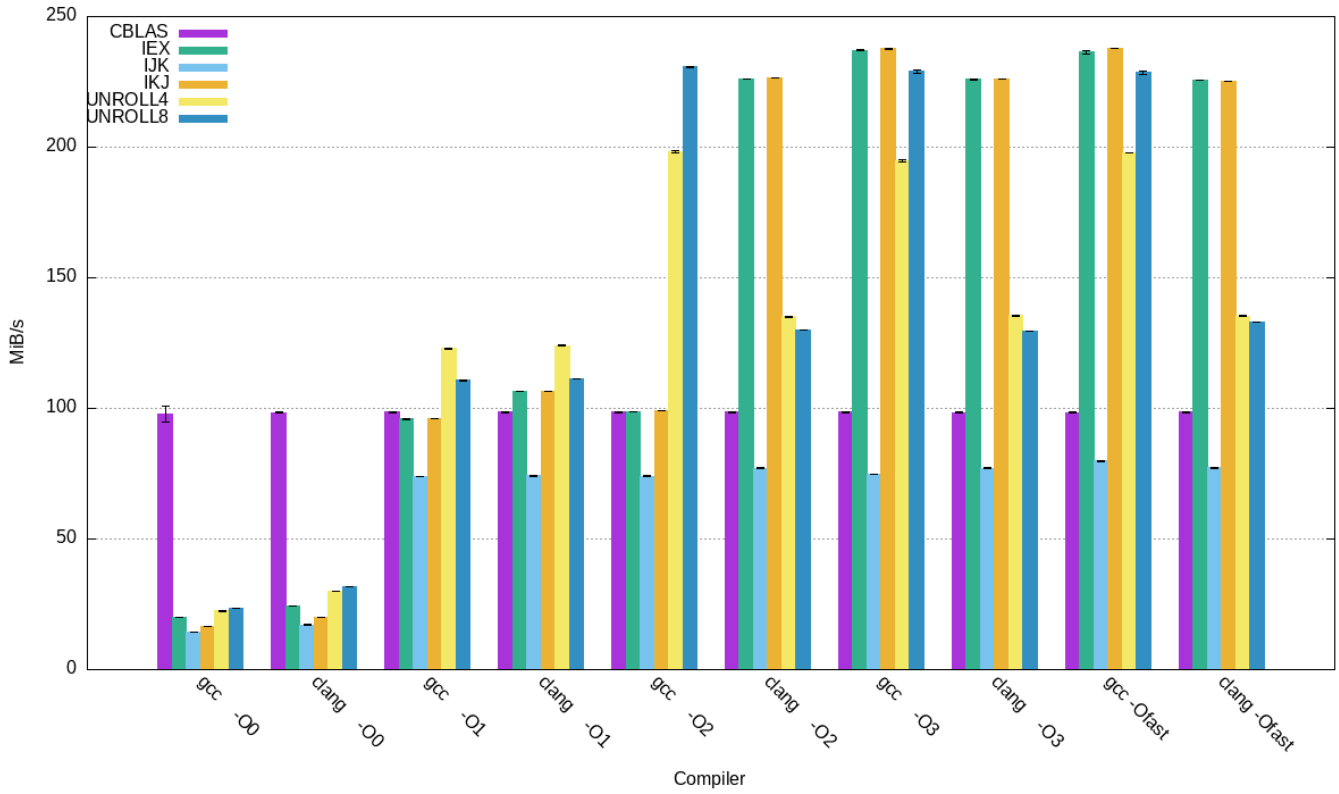

Fig. 2 – Performance d'une dgemm en fonction de la version et du compilateur

2.3 Conclusion

Pour le produit matriciel, on peut noter que *cblas* est très stable entre les configurations mais reste assez mauvaise par rapport à d'autres implémentations.

On peut dire que pour notre configuration, on obtient le plus de performances pour un calcul matriciel pour les versions *iex* et *ikj* compilés grâce à gcc en lui passant l'option *-O3* ou *-0fast* même si cette dernière peut avoir un effet négatif sur la stabilité numérique.

Fait étonnant, le déroulage de boucle (implémentés à partir de *iex*) aurait dans le cas présent, dans notre configuration, un effet négatif sur qualité des optimisations réalisées par les compilateurs.

3 Mesures du produit scalaire

Les mesures ont été obtenues en lançant le programme avec $n = 1048576$ et $r = 33$. C'est-à-dire que les vecteurs contiennent 1048576 nombres flottants double précision. Cette taille a été choisie car elle permet un temps de calcul assez long pour avoir une mesure précise tout en étant une puissance de 2 (2^{20}) ce qui permet aux versions déroulées d'être dans les conditions les plus favorables.

Ces expériences nous donnent les résultats suivants sur notre ordinateur :

3.1 Comparaison des compilations

BASE Cette version est l'implémentation naïve d'un produit scalaire. Pour la version *base*, on observe une augmentation de la performance de *O0* à *O1* de plus de $\times 2$. On voit ensuite une lente augmentation pour clang avec l'augmentation des *O*. Enfin, on voit que pour gcc, on a une légère réduction des performances de *O2* à *O3* avant de remonter au niveau *Ofast*.

CBLAS Cette version appelle directement une fonction de la bibliothèque *cblas*. Pour la version *cblas*, on voit ici que les performances sont assez stables peu importe l'option lorsque gcc est utilisé pour la compilation. Cependant, on remarque que lorsque clang est utilisé à la place la performance est beaucoup moins stable avec un baisse en *O1* avant d'avoir une augmentation. On voit que clang donne une meilleure version *cblas* que gcc sauf en *O1*.

UNROLL4 Cette version contient un déroulage de la boucle avec un facteur quatre. Pour la version *unroll4*, on observe pour les deux compilateurs une amélioration des performances de quasiment $\times 3$ de *O0* à *O1*. Elles restent

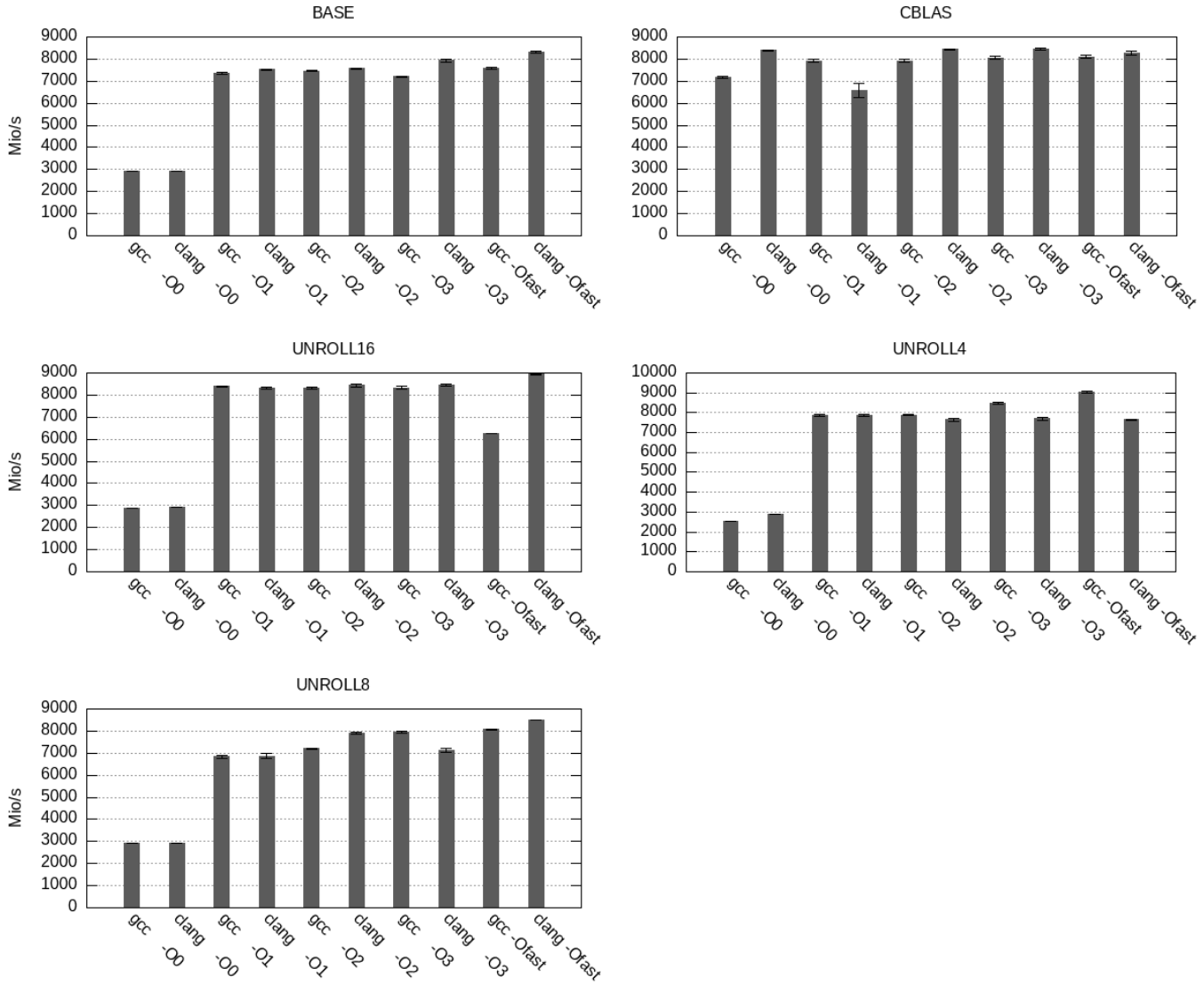

Fig. 3 – Performance de différentes versions d'un dotprod en fonction du compilateur

ensuite constantes de *O1* à *Ofast* pour clang alors que gcc aura d'autres améliorations en *O3* et en *Ofast*. Finalement la meilleure version est celle de gcc en -*Ofast*.

UNROLL8 Cette version contient un déroulage de la boucle avec un facteur huit. Pour la version *unroll8*, on voit une augmentation de *O0* à *O1* pour les deux compilateurs d'environ $\times 2$ de la performance. Pour gcc, on a une augmentation légère de la performance à chaque nouveau flag d'optimisation. Pour clang, on a une augmentation de *O1* à *O2* puis une baisse de performance en *O3*. Enfin, clang obtient les meilleures performances dans la version *Ofast*.

UNROLL16 Cette version contient un déroulage de la boucle avec un facteur seize. Pour la version *unroll16*, on remarque une augmentation de *O0* à *O1* de plus de $\times 2$. Par la suite les options d'optimisation n'affectent plus les performances pour les deux compilateurs (dont les performances sont équivalentes) jusqu'à *Ofast*. A ce niveau d'optimisation, on voit que clang a une légère augmentation alors que gcc a une baisse d'environ 25%. On peut interpréter cette baisse par une transformation appliquée par gcc qui a l'effet inverse que celui escompté.

3.2 Comparaison des versions

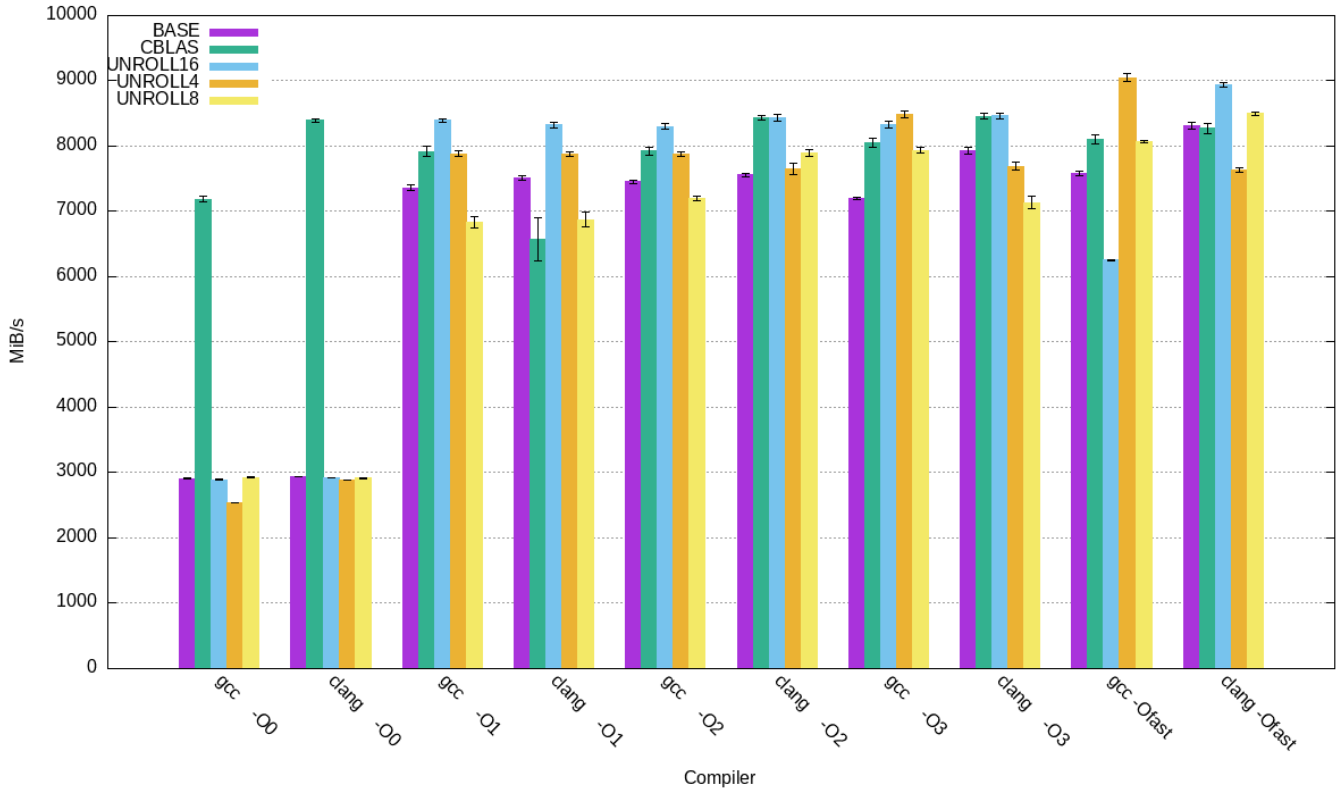

Fig. 4 – Performance d'un dotprod en fonction de la version et du compilateur

Tout d'abord, on observe que pour le produit scalaire, *cblas* a de bonnes performances tout du long.

Néanmoins, à partir de *01* jusqu'à *O3*, la meilleure version est le déroulage de boucle avec un facteur seize peu importe le compilateur.

Pour clang, la version *unroll16* reste la meilleure sur les derniers flags d'optimisation. On peut voir aussi que la version *unroll4* est la moins performante avec ce compilateur pour *O3* et *Ofast*.

Enfin, gcc quant à lui, obtient des meilleures performances avec la version *unroll4* pour *O3* et *Ofast* qui est la plus performante de toutes. On peut remarquer que le déroulage avec un facteur seize perd en performance avec le flag d'optimisation le plus agressif avec ce dernier compilateur.

3.3 Conclusion

On a vu que dans cette configuration la version la plus performante était le déroulage de boucle avec un facteur quatre compilé avec gcc et l'option *Ofast*. Néanmoins, on peut douter de la précision numériques des transformations appliquées par le compilateur avec ce flag.

On a aussi observé des performances assez proches pour toutes les versions (sauf *base*) et peu importe le compilateur pour les options *O2* et *O3*.

Enfin, on a noté de mauvaises transformations appliquées par gcc dans le passage de *O3* à *Ofast* pour la version *unroll16* qui amènent à baisse des performances de cette version.

4 Mesures de la réduction

Les mesures ont été obtenues en lançant le programme avec $n = 1048576$ et $r = 33$. C'est-à-dire que le vecteurs contient1048576 flottants. Cette taille a été choisie car elle permet un temps de calcul assez long pour avoir une mesure précise tout en étant une puissance de 2 (2^{20}) ce qui permet aux versions déroulées d'être dans les conditions les plus favorables.

Suite à ces expériences, nous avons obtenu les résultats suivants :

4.1 Comparaison des compilations

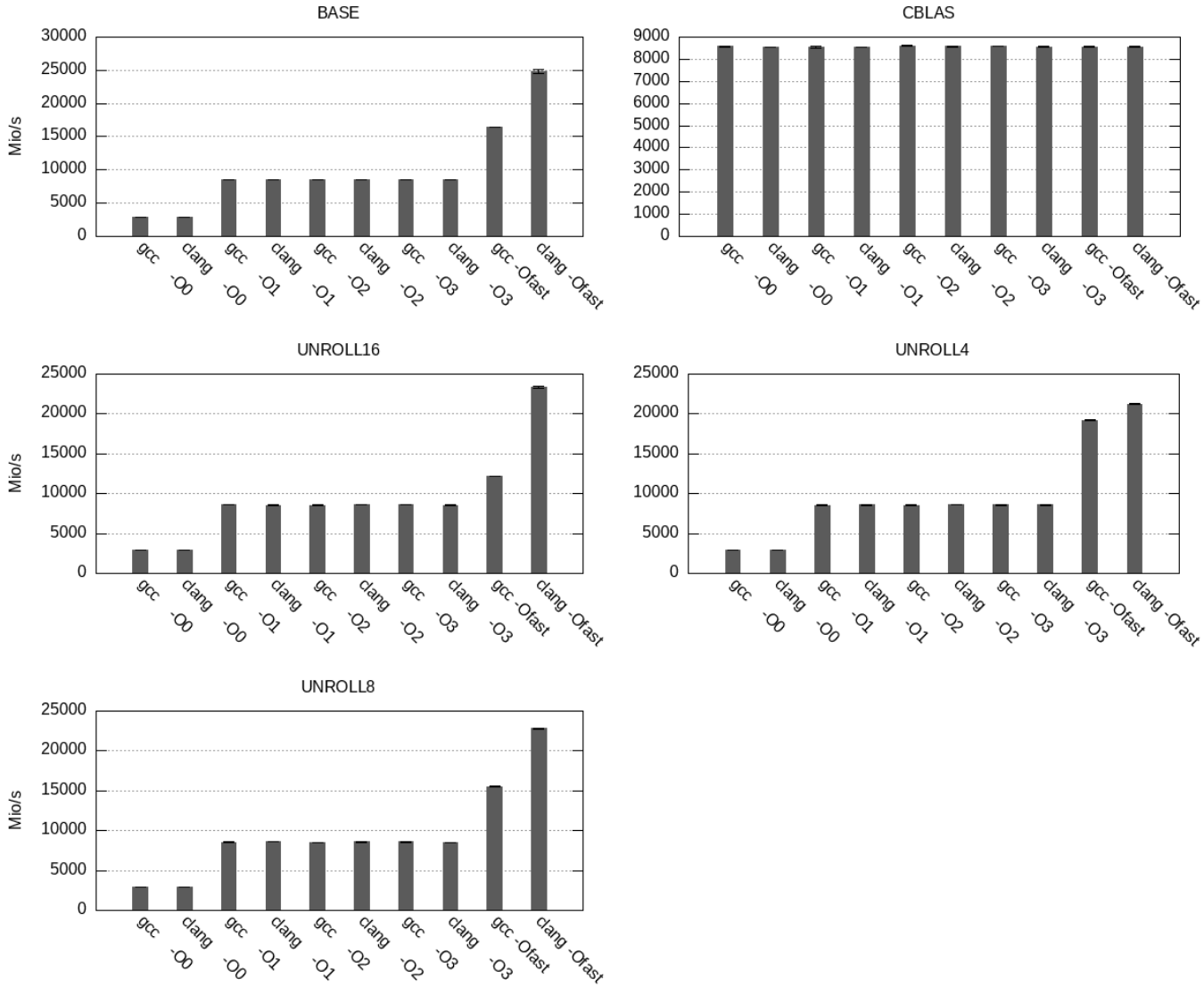

Fig. 5 – Performance de différentes versions d'une réduction en fonction du compilateur

On peut remarquer en préambule que l'évolution des différentes versions se ressemble beaucoup (sauf pour *cblas*).

BASE Cette version est une implémentation naïve de l'algorithme de réduction. Pour la version *base*, on observe une augmentation de *O0* à *01* d'environ ×3. Puis, sur les niveaux d'optimisation *O1*, *O2* et *O3*, les performances restent constantes et équivalentes entre les deux compilateurs. Enfin, pour *Ofast*, on observe une augmentation d'un peu plus de ×2.5 pour clang et d'environ ×1.6 pour gcc.

CBLAS Cette version appelle directement l'implémentation de la norme 1 (la réduction n'existant et les nombres générés étant tous positifs, les résultats sont identiques) contenue dans la bibliothèque *cblas*. Pour la version *cblas*, on observe une très grande stabilité des performances entre les compilateurs et les options d'optimisation. On peut expliquer cela par le fait que cette version appelle une bibliothèque pré-compilée.

UNROLL4 Cette fonction contient un déroulage de la boucle principale avec un facteur quatre. Pour la version *unroll4*, on observe une augmentation de *O0* à *01* d'environ ×3. Puis, sur les niveaux d'optimisation *O1*, *O2* et *O3*, les performances restent constantes et équivalentes entre les deux compilateurs. Enfin, pour *Ofast*, on observe une augmentation d'un peu plus de ×2.5 pour clang et d'environ ×1.6 pour gcc.

UNROLL8 Cette fonction contient un déroulage de la boucle principale avec un facteur huit. Pour la version *unroll8*, on observe une augmentation de *O0* à *01* d'environ ×3. Puis, sur les niveaux d'optimisation *O1*, *O2* et *O3*, les performances restent constantes et équivalentes entre les deux compilateurs. Enfin, pour *Ofast*, on observe une augmentation d'environ ×2.5 pour clang et d'un peu plus de ×1.5 pour gcc.

UNROLL16 Cette fonction contient un déroulage de la boucle principale avec un facteur seize. Pour la version *unroll16*, on observe une augmentation de *O0* à *01* d'environ ×3. Puis, sur les niveaux d'optimisation *O1*, *O2* et *O3*, les performances restent constantes et équivalentes entre les deux compilateurs. Enfin, pour *Ofast*, on observe une augmentation d'environ ×2.5 pour clang et ×1.5 pour gcc.

4.2 Comparaison des versions

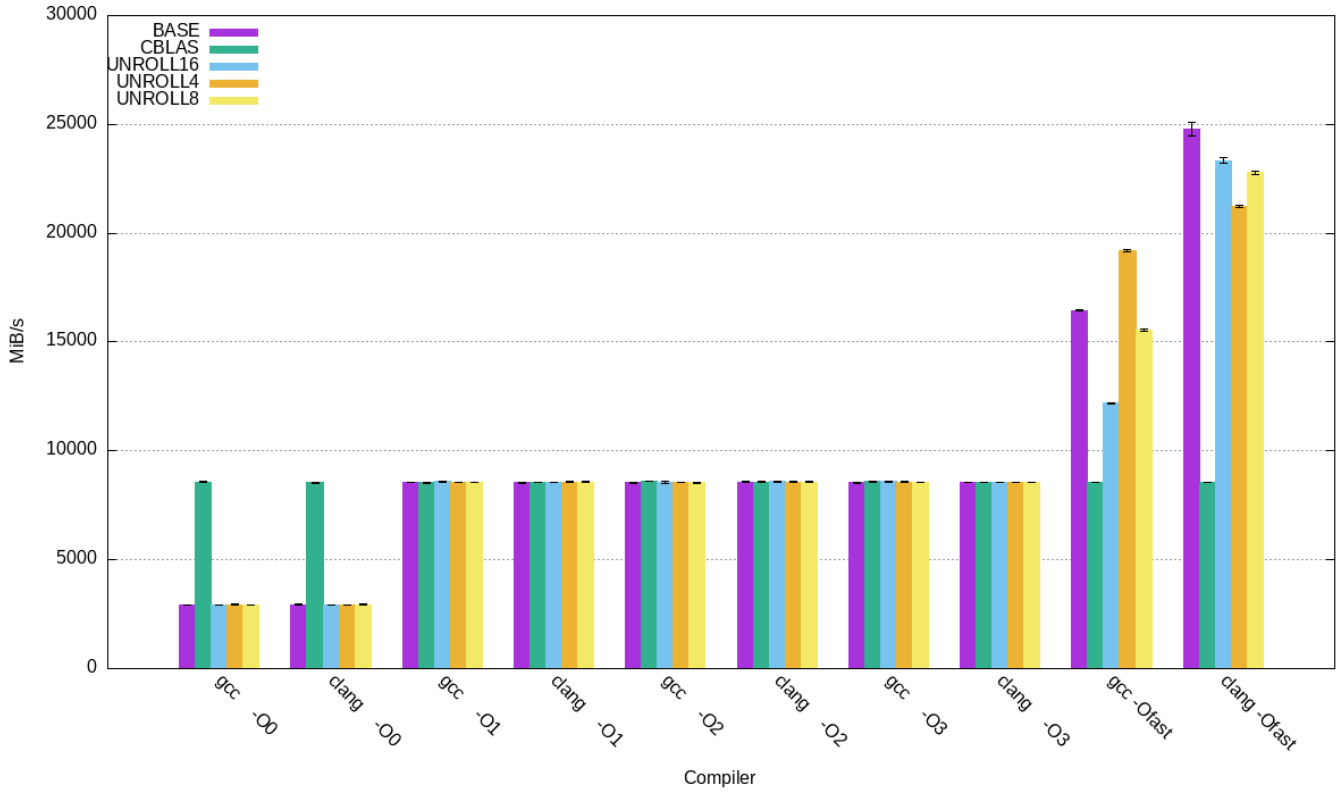

Fig. 6 – Performance d'une réduction en fonction de la version et du compilateur

On voit un équivalence entre les versions pour les flags d'optimisation *O1*, *O2* et *O3*.

On obverse aussi une grosse augmentation des performances en *Ofast* de toutes les versions peu importe le compilateur sauf pour *cblas*.

Les meilleures versions (en terme de performances) sont celles données par clang avec l'option d'optimisation la plus agressive.

4.3 Conclusion

On a observé que *cblas* est très stable et la meilleure version pour une compilation sans optimisation.

Pour les flags *O1*, *O2* et *O3*, les performances de toutes les versions peu importe le compilateur sont équivalentes. On peut penser que les compilateurs n'appliquent pas plus de transformations ou que celle-ci n'offrent pas de gain de performance. Cela met aussi en lumière le traitement sûrement similaire des différentes versions par les compilateurs.

Enfin, on a pu noter une grosse augmentation des performance de toutes les versions sauf *cblas* lorsque l'option *Ofast* est passée au compilateur. On voit donc au final que la version la plus performante est *base* compilée par clang.

On peut, néanmoins, se poser la question de la justesse du calcul fait avec le dernier flag d'optimisation.

5 Conclusion

On observe de façon générale, et de façon attendue, une augmentation des performances toutes les versions (sauf *cblas* qui est pré-compilée) avec l'augmentation du flag d'optimisation à la compilation. Néanmoins, on a vu des exceptions à cette règle surtout pour le calcul du produit scalaire avec gcc montrant une baisse de performances pour le déroulage avec facteur seize en *Ofast* et clang pour le déroulage avec facteur quatre en *O3*.

On note ensuite que la version de *cblas* a beau être toujours la meilleure en *O0*, elle est par la suite au même niveau que les autres implémentations (réduction, produit scalaire), si ce n'est totalement dépassé par la plupart des autres. Pour le produit matriciel, les meilleures versions sont plus de deux fois plus performantes que *cblas*. On voit surtout que sur toutes nos implémentations, *cblas* n'est jamais la plus performante au plus hauts niveaux d'optimisation par les compilateurs. On peut toutefois noter la très grande stabilité de cette bibliothèque pré-compilée à part pour le produit scalaire où les performances semblent être influencées par la compilation.

On a vu des particularités qu'il est intéressant de noter. Tout d'abord, clang a beaucoup plus de mal à optimiser les déroulages de boucles que gcc sur le produit matriciel mais pas sur les autres noyaux de calcul donnant appliquant même de meilleures transformation que gcc dans la plupart des cas.

Ensuite, *Ofast* produit des exécutables dont les performances sont proches de ceux compilés avec *O3* sauf dans le cas de la réduction où on voit une très nette augmentation des performances.

On a pu noter aussi à plusieurs reprises, une augmentation de la performance en *02* pour clang qui n'apparaît qu'en *O3* pour gcc, ce qui nous fait penser que clang n'applique pas les mêmes transformations que gcc en *O2*.

Enfin, bien que les cache lines sur le processeur utilisé sont de 64o, le déroulage de boucle n'est pas forcément la version la plus performante avec un facteur de huit.