OBHPC :

Programmation C et Mesures de Performance

Chadha SAKKA
chadha.sakka@ens.uvsq.fr

MASTER 1 CALCUL HAUTE PERFORMANCE ET SIMULATION
UNIVERSITE PARIS-SACLAY

14 Novembre 2022

Sommaire

1. Introduction & Présentation de la machine :

1.1. Objectifs

Ce TP a pour objectif d'effectuer un banc d'essai permettant de mesurer les performances de différentes versions d'algorithmes de calcul de produit matriciel, produit scalaire, et Baseline implémentées en Langage C tout en compilant les codes sur différents compilateurs et avec plusieurs flags d'optimisation.

1.2. Outils et machine

1.2.1. Caractéristiques de la machine

Les mesures de performance ont été réalisées sur une machine possédant une CPU de modèle Intel Core i5-8250 U CPU @ 1.60 GHz avec une architecture x86_64 basée sur 8 cœurs et 3 lignes de cache : L1 (32K), L2(256K) et L3 (6144K).

1.2.2. Compilateurs :

GCC, abréviation de GNU Compiler Collection, est le compilateur créé par le projet GNU. Il s'agit d'une collection de logiciels libres intégrés capables de compiler divers langages de programmation, dont C.

Clang est un compilateur pour les langages de programmation C, C++ et Objective-C C++. Son interface de bas niveau utilise les bibliothèques LLVM pour la compilation.

1.2.3. Flags d'optimisation :

O1 : réduire la taille du code et le temps d'exécution.

O2 : hérite les flags d'optimisation de O0 et O1 et optimise le code.

O3 : fournit plus d'outils pour l'optimisation du code.

Ofast : assure la meilleure optimisation possible du code en négligent la précision des résultats

1.2.4. Remarque :

Dans chaque partie de ce rapport on analysera dans un premier temps les mesures récoltées par version où on se concentrera sur la performance des compilateurs et la comparaison des flags d'optimisation. Ensuite, on passera à l'analyse globale des codes en mettant l'accent sur la performance des versions et l'efficacité de chaque algorithme.

2. Analyse des mesures de performance

Les mesures ont été effectué en exécutant le programme avec deux tailles de matrices différentes :

- n = 64, r = 60 pour l'analyse par version.
- n = 128, r = 60 pour l'analyse générale.

2.1. Produit matriciel

2.1.1. Algorithme naïf IJK

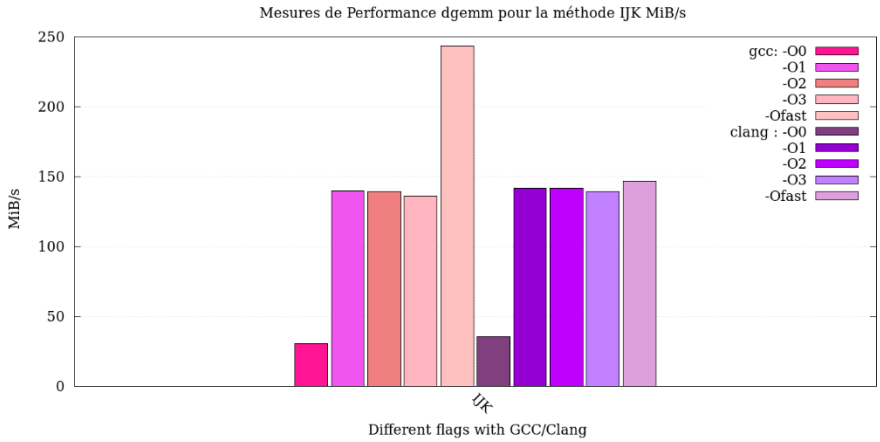

Figure 2.1.1 : Mesures de Performances IJK [n=64]

On remarque que la vitesse de transfert des données de l'algorithme naïf est assez lente (inférieure à 15 MiB/s) avec –O0. Par contre, on note que celle-ci est multiplié par cinq dès qu'on ajoute les flags d'optimisation –O1, -O2, -O3 et par 10 avec l'option –Ofast sur le compilateur gcc. Par ailleurs, l'écart de performance est quasiment le même pour les 2 compilateurs sauf avec le flag –Ofast.

2.1.2. Optimisation 1 : Inversion de boucle

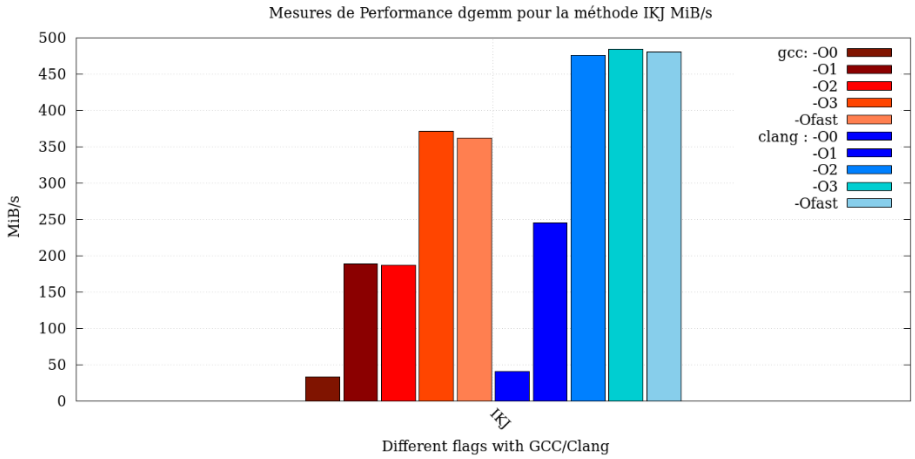

Figure 2.1.2 : Mesures de Performances IKJ [n=64]

On remarque ici qu'avec un simple échange de boucles la vitesse a pu atteindre facilement 500 MiB/s (le double) avec le compilateur clang en utilisant les flag d'optimisation -O2,-O3 et –Ofast et presque 400 MiB/s avec le compilateur gcc. Ceci s'expliquerait peut-être par le fait que la structure de l'algorithme est plus adaptée ou plus proche de l'architecture du CPU.

2.1.3. Optimisation 2 : Echange de boucles & extraction de l'élément invariant

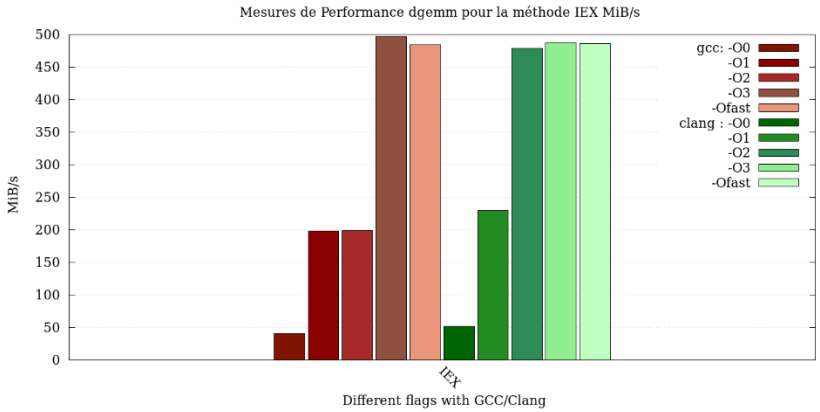

Figure 2.1.3 : Mesures de Performances IKJ [n=64]

On observe une nette amélioration de l'écart de performance entre les mesures de la version ikj et iex en compilant le code sur gcc et en utilisant les flags d'optimisation –O3 et –Ofast. En revanche, on obtient pratiquement les mêmes résultats pour les 2 versions ikj et iex compilées avec clang.

2.1.4. Stratégie de déroulage de boucles :

a) Déroulage par 4 :

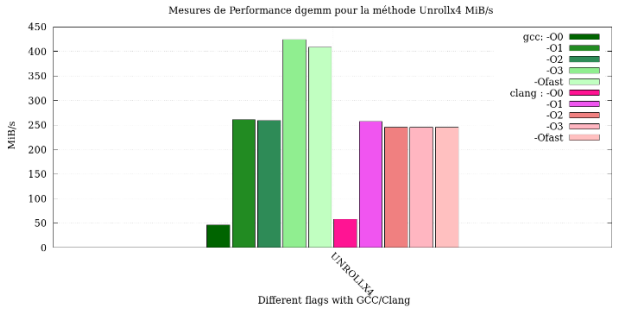

Figure 2.1.4.b : Déroulage x4 [n=64]

b) Déroulage par 8 :

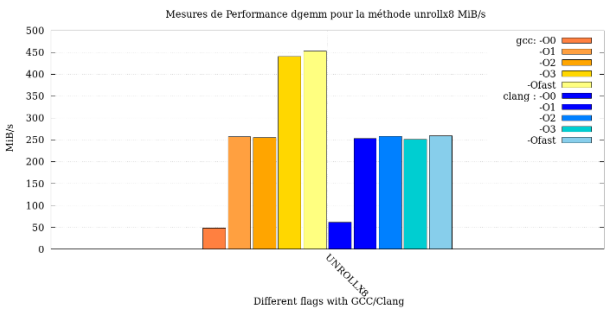

Figure 2.1.4.a : Déroulage x8 [n=64]

D'abord on remarque que nous obtenons quasiment les mêmes résultats pour les deux versions de déroulage. On observe par ailleurs que la vitesse maximale enregistrée sur le compilateur gcc atteint 450 MiB avec un écart de performance croissant en passant d'un niveau de flag d'optimisation à un niveau supérieur. Autrement dit la vitesse du programme augmente lorsque les flags sont plus performants. Par contre, on note que la vitesse est quasi constante pour tous les flags d'optimisation sur le compilateur clang.

2.1.5. Utilisation de BLAS (Basic Linear Algebra Subprograms)

*BLAS : **Basic Linear Algebra Subprograms (BLAS)** est un ensemble de fonctions standardisées réalisant des opérations de base de l'**algèbre linéaire** telles que l'addition de vecteurs, le produit scalaire ou la multiplication de matrices.*

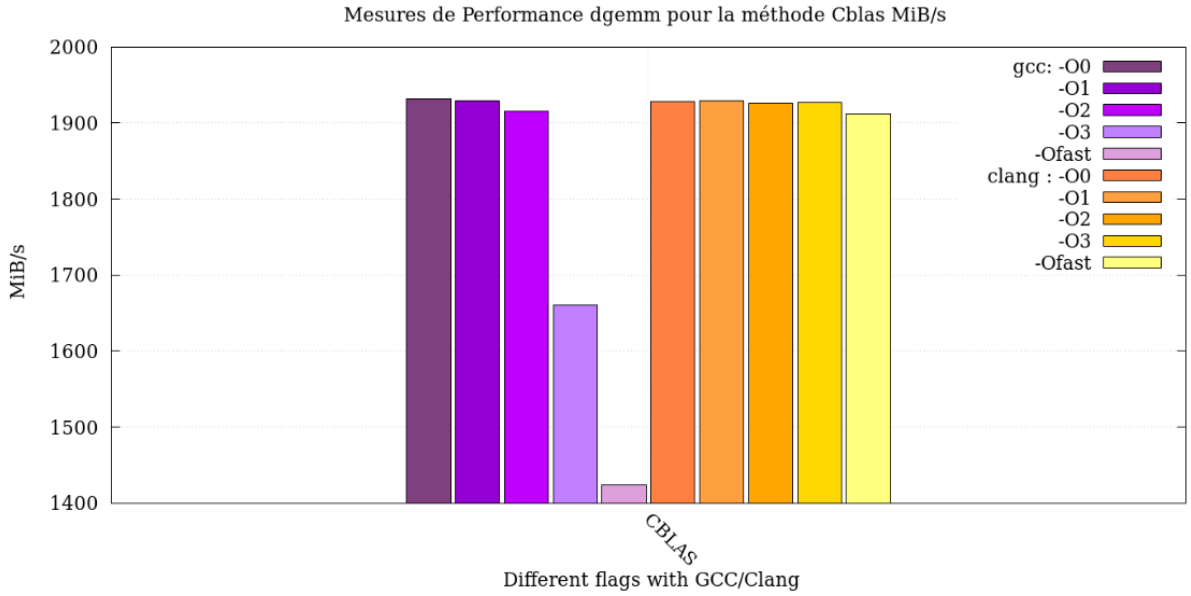

Figure 2.1.5 : Cblas [n=64]

On remarque que la vitesse est quasiment constante pour les deux compilateurs gcc et clang sauf pour les flags –O3 et –Ofast avec gcc qui sembleraient peut-être moins « compatibles » avec CBLAS ? Ou bien s'agirait-il peut-être d'une erreur dans les mesures effectuées. Mais globalement les mesures restent stables en faisant appel à Cblas.

2.1.6. Analyse générale :

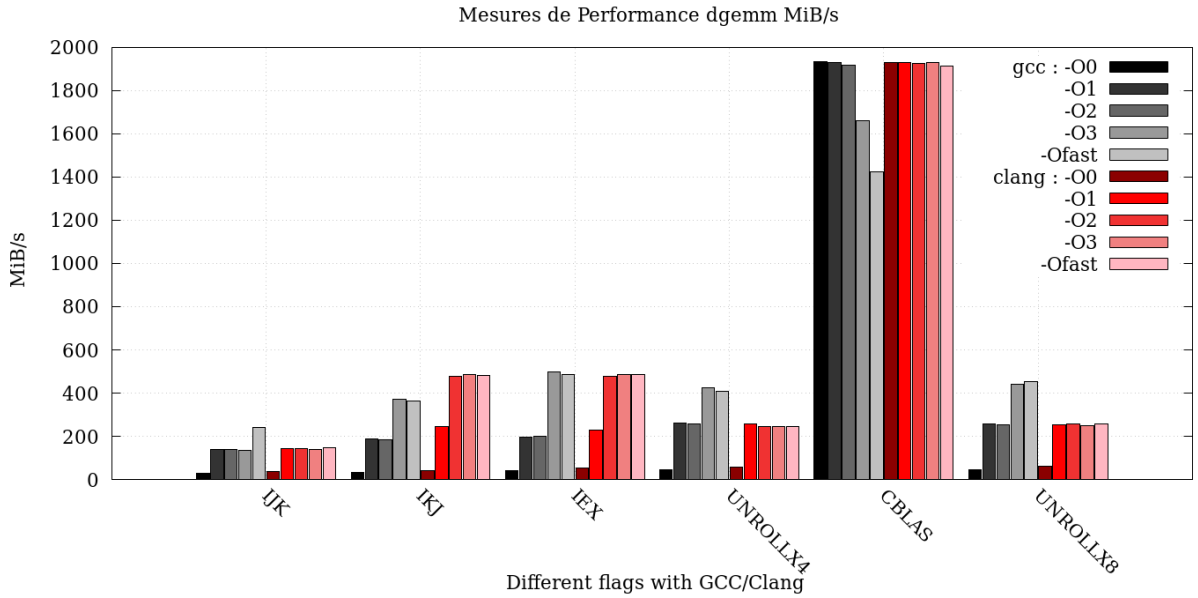

Figure 2.1.6.a : Mesures de Performances de toutes les versions dgemm [n=64]

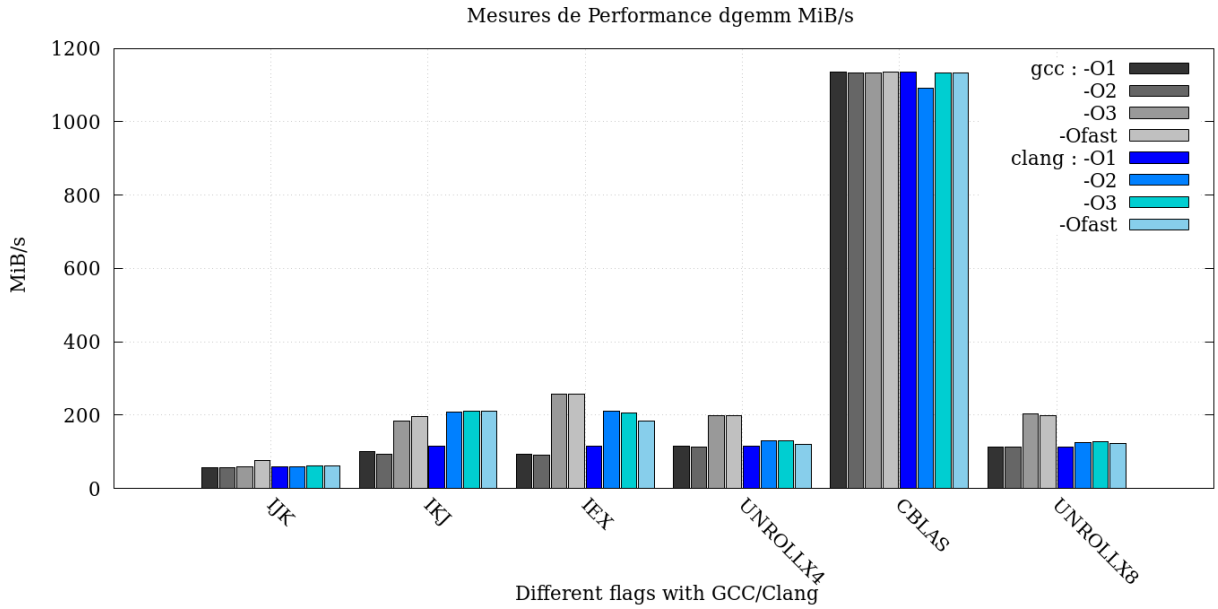

Figure 2.1.6.b : Mesures de Performances de toutes les versions dgemm [n=128]

On observe sur les 2 derniers histogrammes que la version cblas est nettement la plus rapide que toutes les autres versions. On en déduit que l'utilisation des bibliothèques optimisées telle que BLAS assure une meilleure performance et une excellente stabilité des mesures. Par contre, on remarque que la vitesse diminue avec l'augmentation de la taille des matrices : en effet, en doublant la taille des matrices on divise la vitesse par 2.

2.2. Produit Scalaire

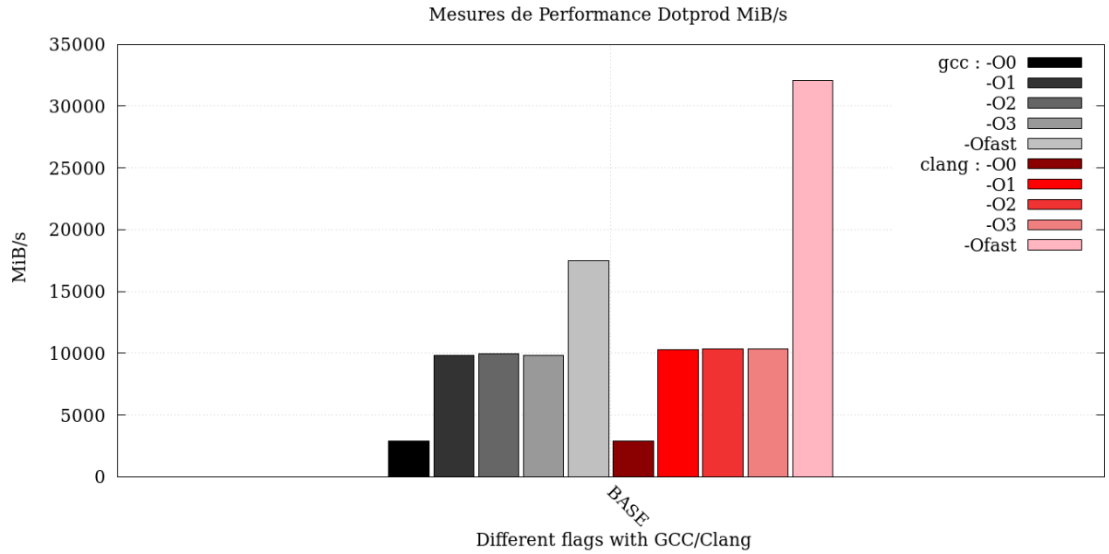

Figure 2.2.1 : Mesures de Performances Dotprod [n=64]

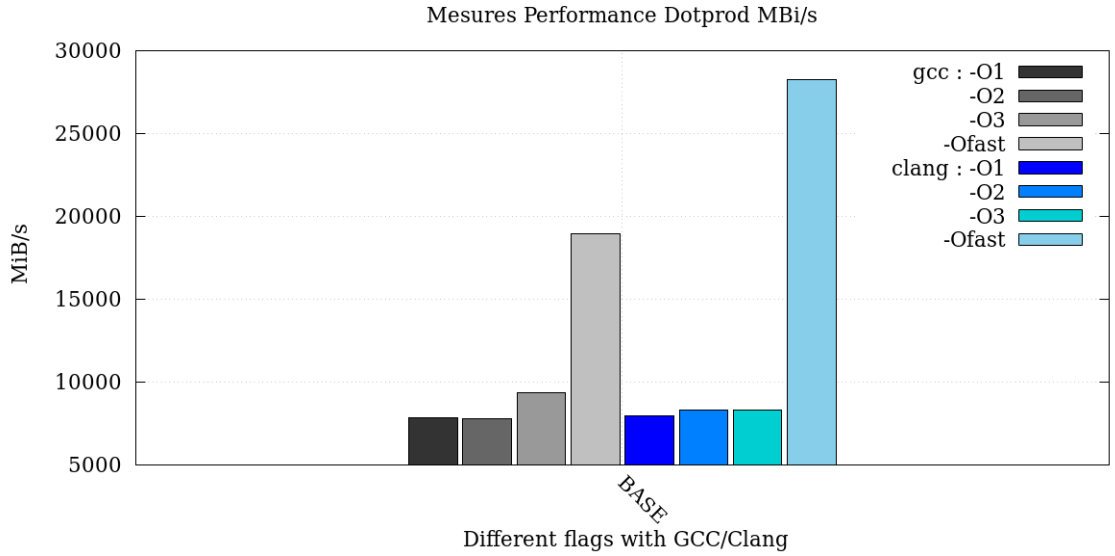

Figure 2.2.2 : Mesures de Performances Dotprod [n=128]

Les mesures obtenues avec gcc et clang sont assez similaires et contrairement aux mesures de la dgemm la vitesse diminue légèrement lorsqu'on augmente la taille des matrices ce qui est tout à fait logique vu que le produit scalaire est beaucoup moins coûteux que la multiplication de 2 matrices. Donc on ne verra pas la différence de vitesse en variant « légèrement » la taille des vecteurs. On remarque également que (clang/-Ofast) est la meilleure combinaison (compilateur/flag) pour optimiser la vitesse du code.

2.3. Reduc

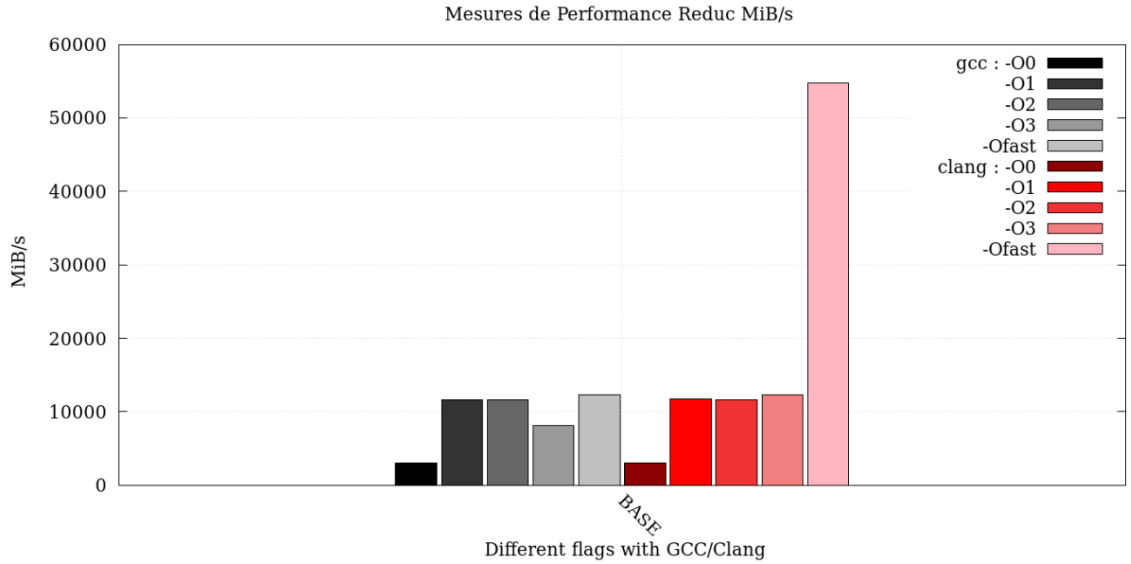

Figure 2.3.a : Mesures de Performances Reduc [n=64]

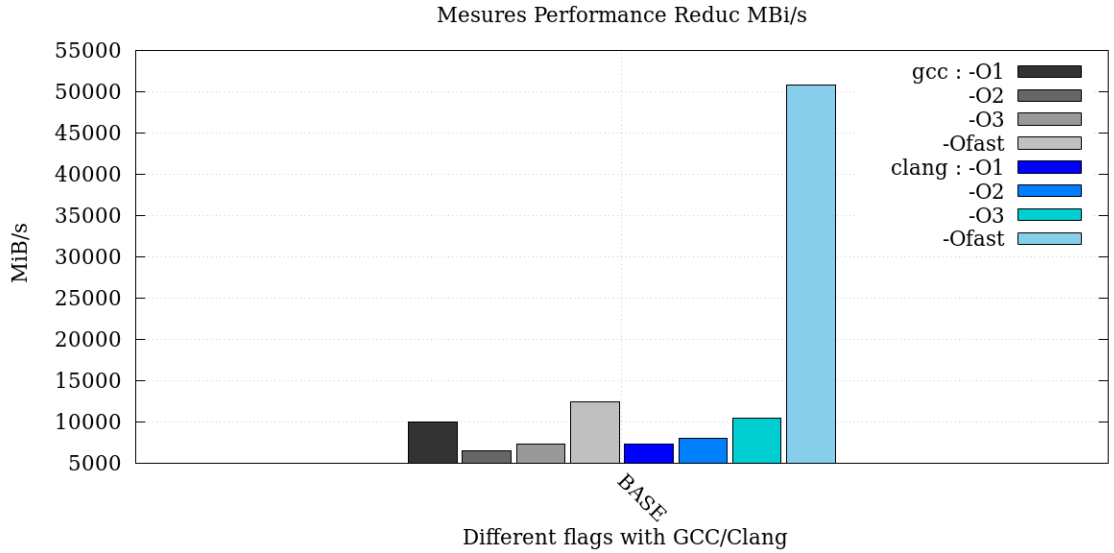

Figure 2.3.b : Mesures de Performances Reduc [n=128]

Pour n=64 on observe que la vitesse est multiplié par 4 en passant de –O0 aux autres flags d'optimisation est reste constante. En revanche, pour n=128 elle chute en passant de –O1 à –O2 et –O3 et reprend une valeur cohérente avec –Ofast.

Remarque : Avec le flag -O0 le temps d'exécution est très long avec les 2 compilateurs gcc ou clang c'est la raison pour laquelle que je n'ai pas réussi à récupérer les mesures de ce flag.

3. Conclusion :

En conclusion, la stratégie de déroulage est plus performante que l'algorithme naïf pour la plupart des flags d'optimisation, l'appel d'une bibliothèque externe déjà compilée telle que cblas assure des mesures de performance meilleures et plus stables et la variation de la taille des matrices affecte les résultats obtenus ce qui pourrait affaiblir la qualité d'interprétation des résultats.