# Rapport du TP : Programmation C et mesures de performances :

## Introduction :

Le but de ce TP était de tester les performances de différents type d'algorithme ainsi que différents type de flags d'optimisation sur les opérations de multiplication entre deux matrices, de multiplication matrice-vecteur ansi que de réduction de matrice. Pour cela nous avons du utiliser des methodes de benchmarking et faire attention a la stabilité de notre machine (pour tout les tests de performance sur Debian en tout cas, Mac étant plus compliqué a gérer dès que l'on essaie de toucher au système).

## Informations sur l'architecture cible :

Tout d'abord nous avons du extraire les informations de l'architecture de notre ordinateur. Pour ce TP j'ai travaillé sur deux machines différentes.

Tout d'abord un Mac sous l'architecture x86 possédant un processeur Intel i7 de 6 coeurs et 16 Go de Ram DDR4. Pour le cache de données nous avons : 32Ko pour le cache L1 d'instruction et de meme pour celui de datas, 262Ko pour le cache L2 et 9437Ko pour le cache L3 (les unitées sont supposées d'après les informations que l'on trouve sur un Linux classique). Sur ce dernier je n'ai pas trouvé le moyen de fixer la fréquence du CPU ou encore de pin un processus sur un coeur de calcul spécifique, j'ai donc utilisé les résultats obtenus comme base de comparaison en sachant qu'ils n'avaient aucune valeure pour de la mesure de performances.

Par la suite j'ai travaillé sur un Dell Inspiron fonctionnant sur l'architecture x86_64, le processeur est un AMD Ryzen7 5700U de 16 coeurs et 16 Go de Ram DDR4. Pour le cache de données nous avons, pour le cache L1 : partagé entre les coeurs 0 et 1, de taille 32Ko et de type Data (on ommetera la partie Instruction du cache L1). Pour le cache L2:partagé entre les coeurs 0 et 1 de meme, de taille 512 Ko et de type unified et enfin le cache L3 de taille 4096Ko partagé entre les coeurs de 0 à 7 lui aussi de type Unified. Cette machine ci fonctionnant sous Debian il n'y a eu aucun problème pour pin un processus sur un CPU ou encore pour vérifier que le CPU tourne a une fréquence (un maximum) stable. Les mesures faites sont donc plus exact que celles récupérées sur le Mac.

## Programmation C et autre :

Pour la partie programmation en C il n'y a pas eu de difficulté particulière, nous avons du implémenter pour chaque type d'opération un déroulage x8, il s'agissait donc simplement de traiter 8 calculs à chaque itération de boucle puis de traiter ceux qui n'avaient pas été faits dans une seconde boucle (simplement car tout nos calculs de sont pas divisibles en bloc de 8).

Par la suite nous avons été ammené a travailler avec GNUPlot ce qui en revanche a été un peu plus complexe, la documentation étant plutot pauvre/peu compréhensible. Les plots fournit ne sont donc pas forcément beau mais sont fonctionnels et permettent d'avoir l'information souhaitée. Le script utilisé est fournit dans les fichiers « plot.p » de chaque typé d'opération (dgemm, dotprod et reduction).

Nous avons aussi été ammené a modifier le makefile fournit pour tester différent type de flag d'optimisation ainsi que différents compilateurs. Pour cela on change directement les arguments en lancant le make, « make CC=gcc OFLAGS=-O3 » par exemple pour lancer la compilation avec gcc et O3. Pour ce TP j'ai travaillé avec clang (version 11.0.3) sur Mac OS (toutes les versions nommées Mac par la suite on été générées avec ce compilateur) et clang (version 14.0.6) ainsi que gcc (version 12.2.0) sur Debian.

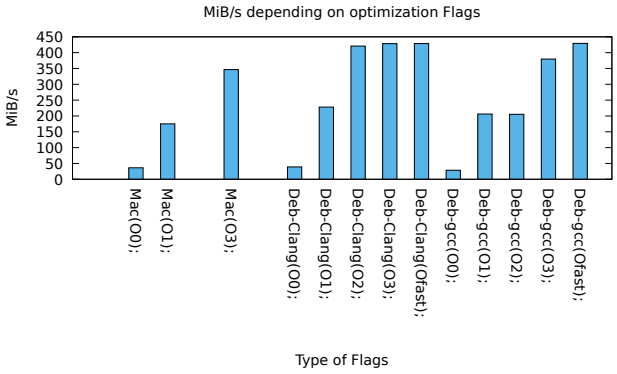
## Mesures de Performances et Résultats :

Dans cette partie, nous avons du récolter des mesures de performances et les analyser, pour cela la machine utilisée doit être stable. Pour les tests effectués sur Mac cela n'a pas pu être vérifié, l'ordinateur était en revanche bien branché au secteur et aucune application ouverte pour avoir le maximum de stabilité possible.

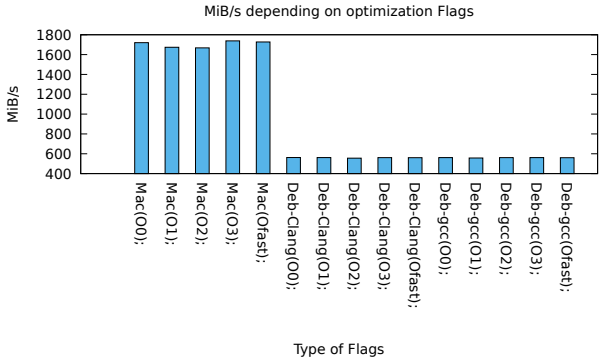
Pour l'ordinateur sous Debian en revanche j'ai pu m'assurer de cela grace a la commande « cpupower frequency-info ». J'ai ensuite fixé le gouverneur du cpu 1 en mode performance (set la fréquence du CPU à 1.80GHz, le maximum possible sur ma machine). Puis pour chaque Benchmark, le processus était pin sur le coeur 1.

J'ai donc obtenus les résultats suivant avec ces méthodes pour les différents algorithmes de dgemm (les graphiques sont aussi disponibles dans le dossier Results de chaque opération):

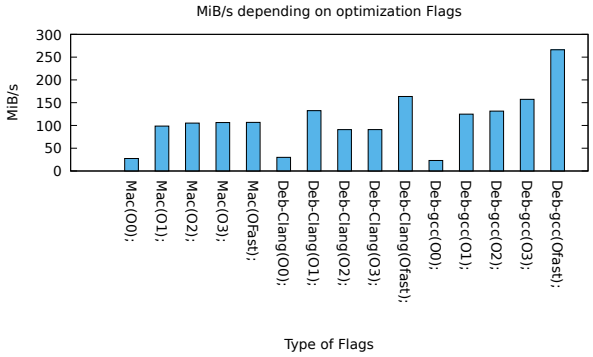
IEX Algorithme :



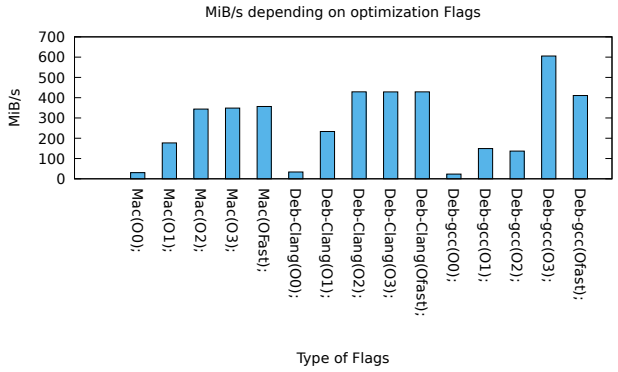

CBLAS DGEMM :



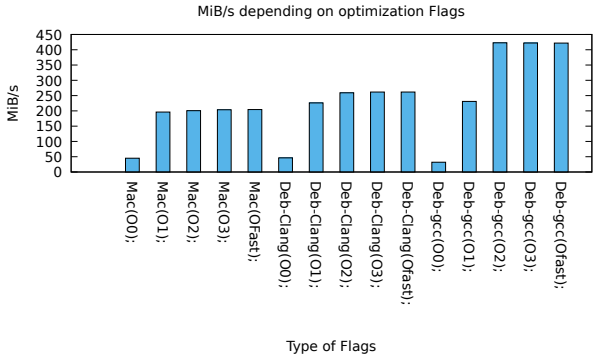

IJK Algorithme :



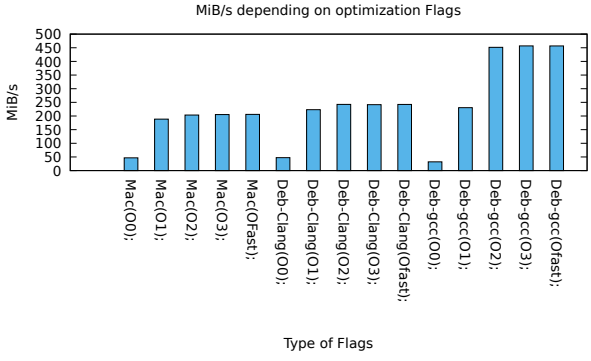

IKJ Algorithme :




Unroll 4 Algorithme :



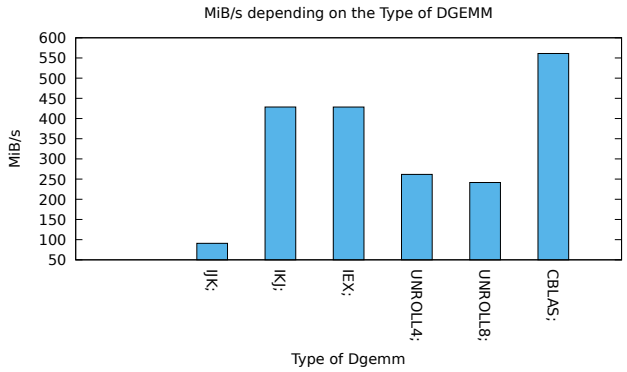

Unroll 8 Algorithme :



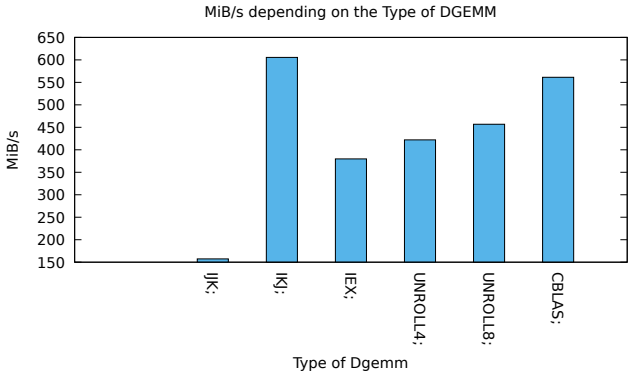
D’après les résultats, on observe que CBLAS est extrèmement stable, que ce soit en O0 comme en Ofast ce qui n’est le cas pour aucune autre implémentation. On peut observer aussi que pour gcc dans l’ensemble les performances sont meilleures que pour clang avec par exemple quasiment un facteur 2 entre clang avec O3 et gcc avec O3 sur l’implémentation Unroll 8. On observe aussi que dans la plus part des cas quelque soit le compilateur on améliore les performances petit a petit entree O0 et Ofast (pour Ofast en regardant dans le detail on se rend compte en revanche qu’il y a plus d’erreurs numérique qu’avec O3). On remarque aussi que les performances mesurées sur Mac ne sont pas totalement « à l’ouest » comparé au reste, en revanche pour CBLAS la différence est flagrante. On observe les mêmes phénomènes pour le dotprod et la réduction.

Un autre point intéressant à étudier est la différence d’éfficacité au sein d’un meme type d’opération des différents algorithme. Par exemple pour le flag 03 sur une DGEMM nous avons les résultats suivants :

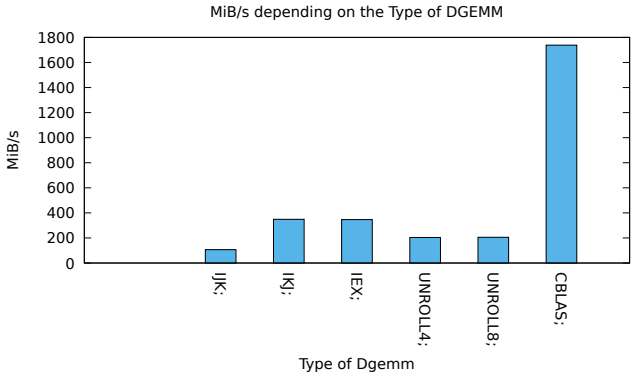
Clang avec O3 :




gcc avec O3




MacOS Clang avecc O3




On observe ici que en fonction du compilateur les version « optimisées » ne sont pas les mêmes, en effet Clang a l’air de mieux gérer l’algorithme IEX que gcc ansi que IJK, gcc tant qu’a lui est plus performant sur les algorithme IKJ et Unroll (4 et 8). En revanche avec O2 on observera que clang est plus performant sur tout ce qui n’est pas du unroll. On peut donc en conclure que gcc est plus adapté aux algorithme de déroulage (pour O0/O1 que ce soit gcc ou clang les performances sont très similaires, j’ai donc jugé que ce n’était pas pertinent de les ajouter). Par la suite pour ce qui est du dotprod et de la réduction on observera seulement que l’algorithme de base est moins performant qu’un déroulage x8 quelque soit le compilateur et le flag d’optimisation.

## Conclusion :

D’après les résultats ci dessus, on peut observer que CBLAS est bien plus optimisé que nos algorithme, quelque soit le compilateur ou le flag d’optimisation, on observe aussi qu’en fonction du compilateur (clang ou gcc) les optimisation ne sont pas faites aux mêmes endroits, gcc sera plus performant par exemple sur du déroulage. Un résultats étonnant en revanche est que malgré que sur le Mac le processus ne pouvait pas etre pin sur un seul coeur et que l’on ne pouvait pas s’assurer de la fréquence du CPU, les performances restent cohérentes avec celles mesurées sur une machine stable.

## Annexe :

Résultats des benchmarks et plots pour DGEMM :
TD2/dgemm/Results/…

Résultats des benchmarks et plots pour dotprod :
TD2/dotprod/Results/…

Résultats des benchmarks et plots pour reduc :
TD2/reduc/Results/...