Compte Rendu
MN-TP2 Blas

Fonctions BLAS:

En utilisant des boucles for, nous avons implémenté les fonctions de blas 1, blas 2 et blas 3. Ces fonctions nous permet de faire des calculs avec des vecteurs de nombres flottants, mais aussi avec des vecteurs de nombre complexes (avec leur partie réel et imaginaire) ainsi qu'avec des matrices (dans blas 2 et 3). Les différents type des fonctions sont représenté par une lettre.

S	Type flottant à précision simple
D	Type flottant à précision double
C	Type complexe à précision simple
Z	Type complexe à précision double

Type complexe

```
typedef struct c{
    float real;
    float imaginary;
} complexe_float_t;

typedef struct z{
    double real;
    double imaginary;
} complexe_double_t;
```

Figure 0: Implémentation des types complexes

Nous avons implémenté nos types complexes avec des structures possédant deux variables, la partie réelle et la partie imaginaire du nombre complexe. Nous avons ensuite créé 4 fonctions pour ces deux types permettant d'additionner deux nombres complexes, de multiplier deux nombres complexes, de conjuguer un nombre complexe et enfin de comparer deux nombres complexe.

Blas 1

Les fonctions **dot** calcule la somme des vecteurs X*Y.

Les fonctions **copy** réalisent une copie de X dans Y.

Les fonctions **axpy** permettent de calculer Y = Y + alpha*X, avec alpha étant une constante.

Les fonctions **iamin** renvoient l'indice où se situe l'élément qui a la plus petite valeur absolue de X.
Les fonctions **iamax** renvoient l'indice où se situe l'élément qui a la plus grande valeur absolue de X .
Les fonctions **swap** inversent les éléments de 2 vecteurs X et Y.

Les fonctions **asum** calculent la somme des valeures absolues des éléments d'un vecteur.
Les fonctions **nrm2** calculent la norme du vecteur X.

Blas 2

Les fonctions **gemv** prennent en paramètre 3 structures : deux vecteurs X et Y, et une matrice A. On a également besoin de deux constantes alpha et beta. La fonction réalise ensuite le calcul suivant :

$$Y = alpha*A*X + beta*Y$$

Pour réaliser facilement cette fonction on utilise la fonction dot de BLAS 1. Pour cette fonction on considère que la matrice est stocké par colonnes (donc on ne fait pas attention au paramètre layout). De plus on ne traite que le cas où la matrice est normal, soit non transposée.

Blas 3

Les fonctions **gemm** prennent en paramètre 3 matrices A, B et C ainsi que deux constantes alpha et beta. Avec ces paramètres, la fonction réalise le calcul suivant:

$$C = alpha*A*B + beta*C$$

Pour réaliser facilement cette fonction on utilise aussi la fonction dot de BLAS 1. Et on ne realise que les mêmes cas que gemv.

Performances

Nous avons testé les performances de toutes nos fonctions blas, en GFLOP(s) ou en GO/s. De plus, nous comparons nos performances calculées avec celle du blas réalisé par Intel. Tous nos programmes de performances se trouvent dans le dossier example du tp. Pour les utiliser il faut avoir la librairie cblas.h.
Pour la compilation de nos programmes BLAS, nous avons expérimenté avec les options de compilation de GCC -O1, -O2 et -O3, permettant d'optimiser le code. Nous avons décidé de garder l'option -O3 qui est censée optimiser le mieux possible nos programmes.
Lors de la réalisation de nos tests de performances, nous avons remarqué plusieures choses commune à tous les programmes.

- Les premiers tests sont souvent les plus lents (et donc possèdent les GFLOPS ou GO/s les plus faibles), nous pensons que cela vient du faites que les données ne sont pas encore en cache et donc que l'accès mémoire fait perdre de la vitesse.
- Nos programmes sont majoritairement tous moins performant que ceux réalisés par Intel. Nous avons des hypothèses sur cela, soit cela doit venir du faites que leur programmes sont mieux implémentés que les nôtres (ce qui est fort probable), soit que leur programmes utilise du multi-threading permettant de grandement accélérer l'exécution (ce qui est pratiquement sûr) ou enfin que leurs programmes utilisent directement le hardware du processeur pour certaines fonctions.
- Les résultats des performances obtenues ne sont pas les mêmes que les résultats que l'on pensait recevoir. Fréquemment les performances de nos fonctions non complexe était meilleur que nos performance sur les fonctions utilisant des complexes. Nous pensions que les fonctions utilisant des nombres complexe seraient plus performantes car réalisant le plus d'opération simple et donc rapide à réaliser, ou alors que les performances entre non complexes et complexes soient relativement similaires.

Performance de dot

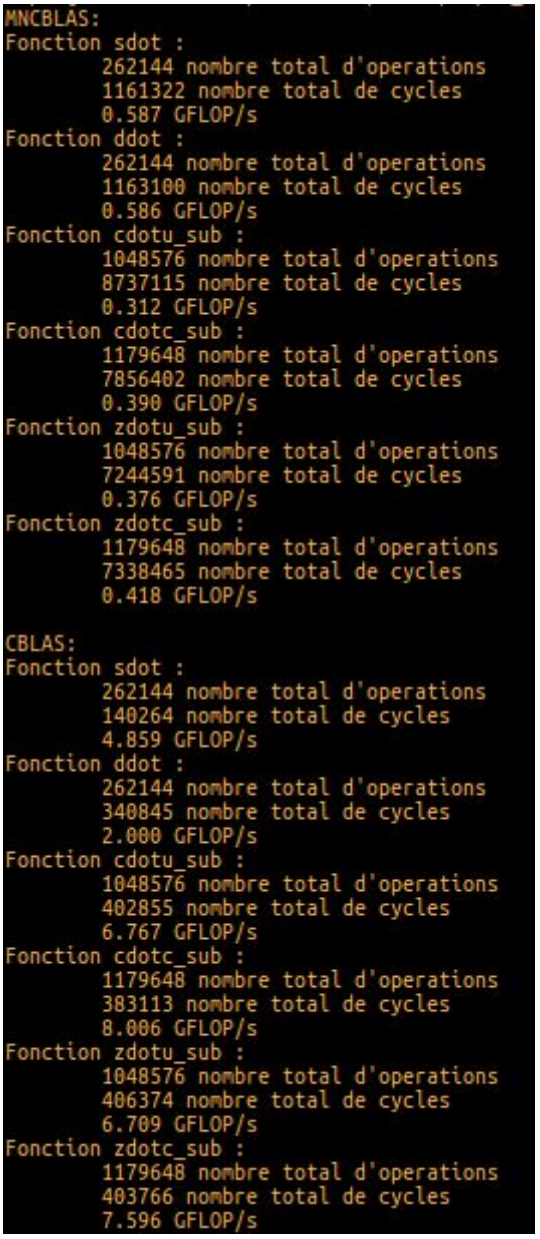

Figure 1: Performances des fonctions dot

On remarque dans un premier temps que nos fonctions blas sont moins performantes que celles écrites par Intel. Nos fonctions sont moins optimisés, donc plus lentes. Cette tendance se retrouve dans toutes les autres fonctions de ce TP. Concernant les fonctions en elle même, on voit que les fonctions dans les nombre réels sont globalement plus performante que les fonctions traitant les nombres complexes. Dans celles ci, calculer le conjugué d'un nombre complexe ralentit encore plus le programme. On remarque que nos fonctions on tendance a être plus rapide en travaillant avec des doubles plutôt qu'avec des flottants.

Performance de copy

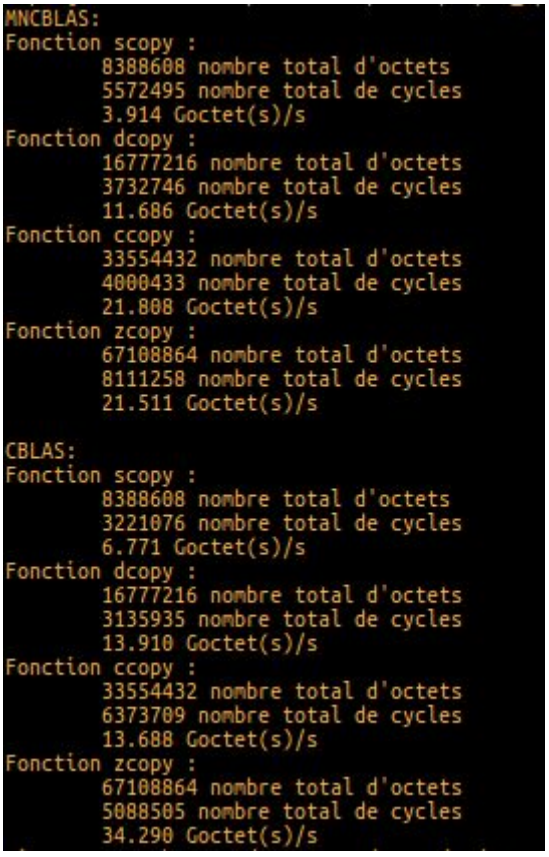

Figure 2: Performances des fonctions copy

Les fonctions copy sont un peu particulières. On ne peut pas calculer de performance en GFLOP/s car il n'y a pas d'opérations flottantes dans ces fonctions. On calcule donc les performances autrement, avec le nombre d'octets par secondes. Ici on a une tendance inverse. Les opérations avec les doubles sont encore plus rapides, mais les complexes sont plus rapide que les nombres non-complexes. Cela est dû au fait que des informations sont stockées dans le cache. L'accès en mémoire est donc plus rapide. On remarque également que notre fonction ccopy est plus rapide que celle écrite par Intel, cela vient peut être de l'optimisation à la compilation (-O3) qui peut être fait des approximations ou garde des valeurs en cache ou alors juste du cache du processeur étant plus utilisé dans un cas que dans l'autre.

Performance de axpy

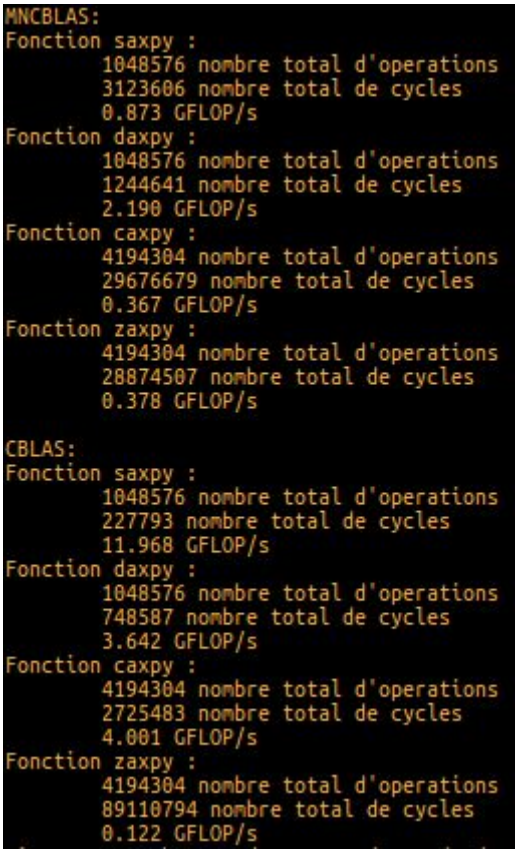

Figure 3: Performances des fonctions axpy

Pour axpy, on voit que d'abord que les nombres complexes sont plus lent dans nos fonctions. Les doubles sont plus rapides ques les flottants. La fonction daxpy est plus rapide que les autres. En comparant avec les fonctions d'Intel, on remarque quelques différences. Leurs fonctions utilisant les doubles sont plus lentes que leurs fonctions utilisant les flottants, tandis que nos fonctions suivent la tendance inverse. Cela peut s'expliquer par une différence d'implémentation, on un manque d'optimisation.

Performance de gemv

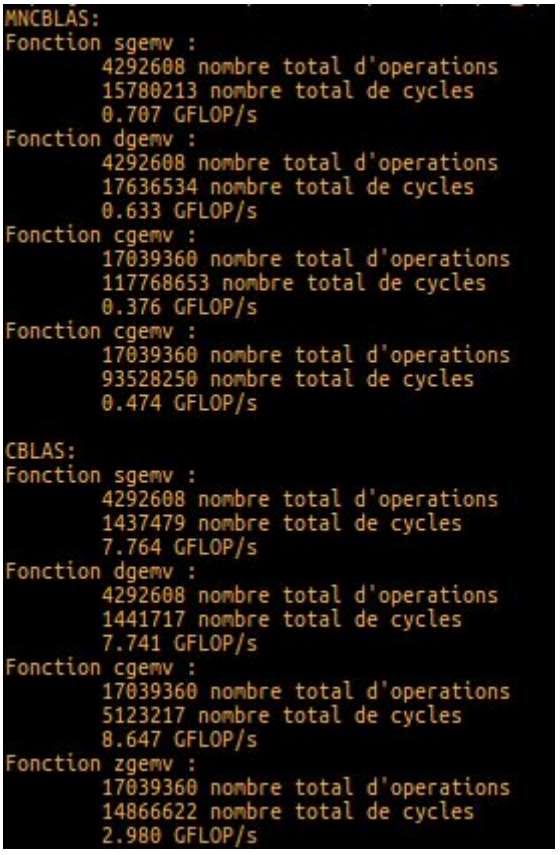

Figure 4: Performances des fonctions gemv

On remarque dans un premier temps que sgemv est plus rapide que dgemv, ce qui est logique. Par contre, zgemv est plus rapide que cgemv, alors que cela devrait normalement être l'inverse. Pour les fonctions d'Intel, on voit que cgemv est la fonction la plus rapide, contrairement à nous où elle est le plus lente.

Performance de gemm

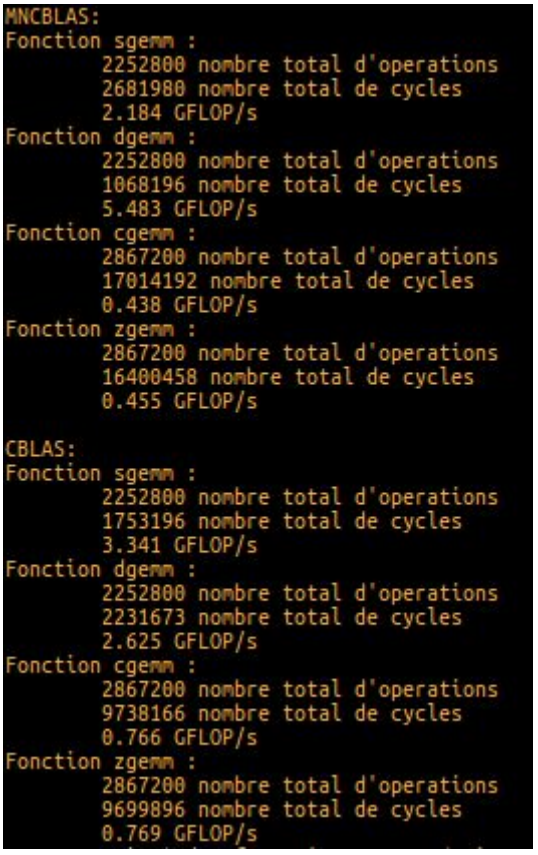

Figure 5: Performances des fonctions gemm

Nos fonctions gemm sont plus rapide en calculant avec des doubles, et les fonctions complexes sont bien plus lentes. Le seul élément remarquable est que notre fonction dgemm est anormalement rapide comparée à la fonction d'Intel. Cela est peut être dû au fait que l'on n'a pas implémenté toutes les fonctionnalitées proposées dans la fonction par Intel.