

# Compte Rendu

## MN-TP2 Blas

<b>Fonctions BLAS:</b>	<b>2</b>
Type complexe	2
Blas 1	2
Blas 2	3
Blas 3	3
<b>Performances</b>	<b>4</b>
Performance de dot	5
Performance de copy	6
Performance de axpy	7
Performance de gemv	8
Performance de gemm	9

## 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.

<b>S</b>	Type flottant à précision simple
<b>D</b>	Type flottant à précision double
<b>C</b>	Type complexe à précision simple
<b>Z</b>	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 complexes.

## 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 valeurs 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 réalise 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 exemple 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 plusieurs 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 fait 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 fait 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

```

MNCBLAS:
Fonction sdot :
    262144 nombre total d'operations
    1161322 nombre total de cycles
    0.587 GFLOP/s
Fonction ddot :
    262144 nombre total d'operations
    1163100 nombre total de cycles
    0.586 GFLOP/s
Fonction cdotu_sub :
    1048576 nombre total d'operations
    8737115 nombre total de cycles
    0.312 GFLOP/s
Fonction cdotc_sub :
    1179648 nombre total d'operations
    7856402 nombre total de cycles
    0.390 GFLOP/s
Fonction zdotu_sub :
    1048576 nombre total d'operations
    7244591 nombre total de cycles
    0.376 GFLOP/s
Fonction zdotc_sub :
    1179648 nombre total d'operations
    7338465 nombre total de cycles
    0.418 GFLOP/s

CBLAS:
Fonction sdot :
    262144 nombre total d'operations
    140264 nombre total de cycles
    4.859 GFLOP/s
Fonction ddot :
    262144 nombre total d'operations
    340845 nombre total de cycles
    2.000 GFLOP/s
Fonction cdotu_sub :
    1048576 nombre total d'operations
    402855 nombre total de cycles
    6.767 GFLOP/s
Fonction cdotc_sub :
    1179648 nombre total d'operations
    383113 nombre total de cycles
    8.006 GFLOP/s
Fonction zdotu_sub :
    1048576 nombre total d'operations
    406374 nombre total de cycles
    6.709 GFLOP/s
Fonction zdotc_sub :
    1179648 nombre total d'operations
    403766 nombre total de cycles
    7.596 GFLOP/s

```

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 optimisées, 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

```

MNCBLAS:
Fonction scopy :
  8388608 nombre total d'octets
  5572495 nombre total de cycles
  3.914 Goctet(s)/s
Fonction dcopy :
  16777216 nombre total d'octets
  3732746 nombre total de cycles
  11.686 Goctet(s)/s
Fonction ccopy :
  33554432 nombre total d'octets
  4000433 nombre total de cycles
  21.808 Goctet(s)/s
Fonction zcopy :
  67108864 nombre total d'octets
  8111258 nombre total de cycles
  21.511 Goctet(s)/s

CBLAS:
Fonction scopy :
  8388608 nombre total d'octets
  3221076 nombre total de cycles
  6.771 Goctet(s)/s
Fonction dcopy :
  16777216 nombre total d'octets
  3135935 nombre total de cycles
  13.910 Goctet(s)/s
Fonction ccopy :
  33554432 nombre total d'octets
  6373709 nombre total de cycles
  13.688 Goctet(s)/s
Fonction zcopy :
  67108864 nombre total d'octets
  5088505 nombre total de cycles
  34.290 Goctet(s)/s

```

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

```
MNCBLAS:
Fonction saxpy :
1048576 nombre total d'operations
3123606 nombre total de cycles
0.873 GFLOP/s
Fonction daxpy :
1048576 nombre total d'operations
1244641 nombre total de cycles
2.190 GFLOP/s
Fonction caxpy :
4194304 nombre total d'operations
29676679 nombre total de cycles
0.367 GFLOP/s
Fonction zaxpy :
4194304 nombre total d'operations
28874507 nombre total de cycles
0.378 GFLOP/s

CBLAS:
Fonction saxpy :
1048576 nombre total d'operations
227793 nombre total de cycles
11.968 GFLOP/s
Fonction daxpy :
1048576 nombre total d'operations
748587 nombre total de cycles
3.642 GFLOP/s
Fonction caxpy :
4194304 nombre total d'operations
2725483 nombre total de cycles
4.001 GFLOP/s
Fonction zaxpy :
4194304 nombre total d'operations
89110794 nombre total de cycles
0.122 GFLOP/s
```

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 que 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, ou un manque d'optimisation.

## Performance de gemv

```
MNCBLAS:
Fonction sgemv :
    4292608 nombre total d'operations
    15780213 nombre total de cycles
    0.707 GFLOP/s
Fonction dgemv :
    4292608 nombre total d'operations
    17636534 nombre total de cycles
    0.633 GFLOP/s
Fonction cgemv :
    17039360 nombre total d'operations
    117768653 nombre total de cycles
    0.376 GFLOP/s
Fonction zgemv :
    17039360 nombre total d'operations
    93528250 nombre total de cycles
    0.474 GFLOP/s

CBLAS:
Fonction sgemv :
    4292608 nombre total d'operations
    1437479 nombre total de cycles
    7.764 GFLOP/s
Fonction dgemv :
    4292608 nombre total d'operations
    1441717 nombre total de cycles
    7.741 GFLOP/s
Fonction cgemv :
    17039360 nombre total d'operations
    5123217 nombre total de cycles
    8.647 GFLOP/s
Fonction zgemv :
    17039360 nombre total d'operations
    14866622 nombre total de cycles
    2.980 GFLOP/s
```

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 la plus lente.



## Performance de gemm

```
MNCBLAS:
Fonction sgemm :
    2252800 nombre total d'operations
    2681980 nombre total de cycles
    2.184 GFLOP/s
Fonction dgemm :
    2252800 nombre total d'operations
    1068196 nombre total de cycles
    5.483 GFLOP/s
Fonction cgemm :
    2867200 nombre total d'operations
    17014192 nombre total de cycles
    0.438 GFLOP/s
Fonction zgemm :
    2867200 nombre total d'operations
    16400458 nombre total de cycles
    0.455 GFLOP/s

CBLAS:
Fonction sgemm :
    2252800 nombre total d'operations
    1753196 nombre total de cycles
    3.341 GFLOP/s
Fonction dgemm :
    2252800 nombre total d'operations
    2231673 nombre total de cycles
    2.625 GFLOP/s
Fonction cgemm :
    2867200 nombre total d'operations
    9738166 nombre total de cycles
    0.766 GFLOP/s
Fonction zgemm :
    2867200 nombre total d'operations
    9699896 nombre total de cycles
    0.769 GFLOP/s
```

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 fonctionnalités proposées dans la fonction par Intel.