INFO3 Polyteh Grenoble
Bibliothèque d'Algèbre Linéeaire (BLAS)
TD-TP 2

Jean-François Méhaut

12 février 2022

L'objectif de ce TP2 est de développer une bibliothèque d'algèbre lineaire. Vous reprendrez une partie des fonctions de la bibliothèque BLAS (Basic Linear Algebre Suprograms). Cette bibliothèque BLAS sera structuée en 3 ensembles de fonctions.

— Les fonctions **BLAS1** effectuent des opérations de type vecteur-vecteur.
— Les fonctions **BLAS2** effectuent des opérations de type vecteur-matrice.
— Les fonctions **BLAS3** effectuent des opérations de type vecteur-vecteur.

Ce TP2 de MN est à rendre avant le **25 avril 2022**. Vous déposerez sur le moodle votre compte-rendu avec le lien git vers le code source de votre bibliothèque BLAS Vous donnerez des éléments d'analyse de performance (GFLOP/s, GB/s) de certaines fonctions BLAS1, BLAS2 et BLAS3. Vous indiquerez également l'enpreinte mémoire (en GB) des programmes de tests que vous fournirez.

Les fonctions de votre bibliothèque se feront avec 4 types de données :

— `float`, flottant, simple précision (**s**)
— `double`, double, double précision (**d**)
— `complexe_float_t`, complexe simple précision (**c**)
— `complexe_double_t`, complexe double précision (**z**)

La documentation de l'ensemble des fonctions BLAS développées par la société Intel se trouve sur le site suivant : `http://cali2.unilim.fr/intel-xe/mkl/mklman/index.htm`.

Pour les tests et évaluations de performance, vous créerez un programme de test par type de fonction. Chaque programme de test devra tester les fonctions avec les 4 types de données (`float`, `double`, `complexe simple` et `complexe double`).

1 Préparation du TP

Vous devez d'abord récupérer l'archive `TPBLAS.tar.gz` sur le moodle.
Vous allez ensuite exécuter la commande `tar` pour extraire le code source qui est fourni.

```
$ tar xvfz TPBLAS.tar.gz
TPBLAS/
TPBLAS/examples/
TPBLAS/examples/test_complexe4.c
TPBLAS/examples/flop.c
TPBLAS/examples/test_complexe3.c
```

```
TPBLAS/examples/Makefile
TPBLAS/examples/flop.h
TPBLAS/examples/test_dot.c
TPBLAS/examples/test_complexe.c
TPBLAS/examples/test_complexe2.c
TPBLAS/doc/
TPBLAS/include/
TPBLAS/include/complexe2.h
TPBLAS/include/mnblas.h
TPBLAS/include/complexe.h
TPBLAS/lib/
TPBLAS/src/
TPBLAS/src/complexe.c
TPBLAS/src/Makefile
TPBLAS/src/dot.c
TPBLAS/src/copy.c
TPBLAS/src/swap.c
$
```

Cette archive contient les répertoires suivants :

- Le répertoire `src` contient les fichiers source des fonctions BLAS. Un fichier `Makefile` est fourni pour compiler les premières fonctions BLAS1. Vous compléterez ce fichier `Makefile` avec les nouvelles fonctions BLAS1, BLAS2 et BLAS3 que vous implémenterez.
- Le répertoire `include` contient le fichier `mnblas.h`. Les premières fonctions BLAS1 y sont déclaréées, d'autres déclarations sont commentées. Vous devrez les décommenter au fur et à mesure de l'avancement de votre travail.
- Le répertoire `lib` contient la librairie statiqe (`libmnblas.a`) et la librairie dynamique (`libmnblasdyn.so`). Ces deux bibliothèqies sont générées après compilation et édition de liens des fichiers du répertoire *tt*.
- Le répertoire *examples* contiendra les examples de programmes utilisant votre bibliothèque BLAS. Il est conseillé d'y placer plusieurs programmes d'examples (un par fonction ou type de fonction). Les versions en statique et en dynamique doivent être générées. Ce répertoire contient également un fichier `flop.c` qui permet de calculer le nombre de flop (opération flottante) par seconde de la fonction.

Vous allez ensuite compiler la compilation de la bibliothèse et la compilation des examples. Vous devez d'abord vous positionner dans le répertoire `src` pour compiler la bibliithèque BLAS. Vous devez ensuite vous positionner dans le répertoire `examples` pour compiler les programmes de test et examples.

```
$ cd TPBLAS/src
$ make
gcc -O2 -Wall -fPIC -O2 -fopenmp -I../include -c copy.c
gcc  -O2 -Wall -fPIC -O2 -fopenmp -I../include -c swap.c
gcc  -O2 -Wall -fPIC -O2 -fopenmp -I../include -c dot.c
gcc -O2 -Wall -fPIC -O2 -fopenmp -I../include -c complexe.c
rm -f libmnblas.a ../lib/libmnblas.a
ar -r libmnblas.a copy.o swap.o dot.o complexe.o
ar: creation de libmnblas.a
cp libmnblas.a ../lib
```

```
rm -f libmnblasdyn.so ../lib/libmnblasdyn.so
gcc -shared -o libmnblasdyn.so copy.o swap.o dot.o complexe.o
cp libmnblasdyn.so ../lib
$ cd ../examples
$ make
gcc -Wall -O2 -fPIC -I../include -c test_complexe4.c
gcc -Wall -O2 -fPIC -I../include -c flop.c
gcc -o test_complexe4 test_complexe4.o flop.o -fopenmp
                                        -L../lib -lmnblas
gcc -Wall -O2 -fPIC -I../include -c test_complexe3.c
gcc -o test_complexe3 test_complexe3.o flop.o -fopenmp
                                         -L../lib -lmnblas
gcc -Wall -O2 -fPIC -I../include -c test_complexe2.c
gcc -o test_complexe2 test_complexe2.o flop.o -fopenmp
                                          -L../lib -lmnblas
gcc -Wall -O2 -fPIC -I../include -c test_complexe.c
gcc -o test_complexe test_complexe.o flop.o -fopenmp
                                           -L../lib -lmnblas
gcc -Wall -O2 -fPIC -I../include -c test_dot.c
gcc -o test_dot test_dot.o flop.o -fopenmp -L../lib -lmnblas
gcc -o test_dot_dyn flop.o test_dot.o -fopenmp
                                           -L../lib -lmnblasdyn
$
```

Vous lancerez ensuite l'éxécution des programmes de tests qui sont dans le répertoire `examples`.

```
$ test_complexe
residu_micro = 1.000000e-06
c1.r 4.000000 c&.i 8.000000
apres boucle cd1.real 12810.000000 cd1.imaginary 16391.000000
                                                 duree 0.000003
calcul complexe  1024 operations duree 0.0000030 seconde
                                Performance 0.341 GFLOP/s
$ test_complexe2
residu_micro = 0.000000e+00
c1.r 4.000000 c1.i 8.000000
apres boucle cd1.real 12810.000000 cd1.imaginary 16391.000000 0.000002
calcul complexe  1024 operations duree 0.0000020 seconde
                                  Performance 0.512 GFLOP/s
$ test_complexe3
residu_tsc = 16 cycles
c1.r 4.000000 c1.i 8.000000
apres boucle cd1.real 12810.000000 cd1.imaginary 16391.000000
                                     1330 cycles
calcul complexe nano  1024 operations 1330 cycles
                                    Performance 2.002 GFLOP/s
$ test_complexe4
residu_nano = 2.930000e-07
c1.r 4.000000 c1.i 8.000000
apres boucle cd1.real 12810.000000 cd1.imaginary 16391.000000 0.000003
calcul complexe  1024 operations duree 0.0000026 seconde
```

```
                                        Performance 0.399 GFLOP/s

$
```

2 Calculs sur les nombres complexes

Deux types de nombres complexes sot définis dans le fichier `include/complexe.h`. Ce type et défini avec une structure contenant la partie réelle et la partie imaginaire du nombre complexe.

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

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

...
...
```

Des fonctions d'addition, multiplication et division sur les nombres complexes doivent être réalisées.

1. Ecrivez les fonctions de calcul sur les nombres complexes qui sont dans le fichier `complexe.c`.

2. Complétez le fichier `test_complexe.c` pour tester et évaluer les opérations opérant sur les complexes. Le programme `test_complexe.c` utilise la fonction `calcul_flop` définie dans le fichier `flop.c`. La mesure de temps est basée sur la fonction `gettimeofday` qui mesure le temps au niveau de la milliseconde.

Le fonction `complexe2.h` contient également une déclaration des types `complexe`. Les déclarations de type sont identiques à celles qui se trouvent dans le fichier `complexe.h`. La principale différence se situe au niveau de la signature des fonctions qui sont *inlinées* dans le fichier `complexe2.h`. L'extension `inline` est prise en compte par le compilateur C qui va effectuer le changement suivant : Lorsqu'une fonction inlinée est appelée, le compilateur place le code de la fonction à l'endroit de l'appel de la fonction *inlinée*. Cela va éviter au programme d'empiler les paramètres de la fonction. C'est partculièrement efficace pour des fonctions où le nombre d'instruction est faible.

3. Complétez le fichier `complexe2.h` avec le code associé à l'ensemble des fonctions opérant sur les complexes.
4. Vous allez lancer l'exécution du programme `test_complexe2`. Commentez le résultat obtenu.
5. Vous allez créer un programme `test_complexe3.c` qui va utiliser les outils et fonctions de mesure du temps au niveau des cycles processeur et de la nanoseconde.
6. Comparez les résultats obtenus avec les programmes `test_complexe` et `test_complexe3`.

3 Fonctions BLAS1

Plusieurs fonctions BLAS1 doivent être implémentées. Vous trouverez sur la documentation Intel la description de ces fonctions.

Les fonctions `swap` et `copy` sont un peu particulières car elles n'effectuent pas de calcul. Ces deux fonctions effectuent prncipalement des copies en mémoire. La fonction `copy` copie le vecteur `x` dans le vecteur `y`. La fonction `swap` permute le contenu des vecteurs `x` et `y`. Vous calculerez la performance de ces fonctions en utilisant comme métrique des giga-octets par seconde (GB/s).

Pour les autres fonctions BLAS1, vous calculerez la performance avec comme métrique le nombre d'opérations flottantes par seconde. Vous pourrez utiliser les fonctions qui sont définies dans les fichiers `flop.h` et `flop.c`. Vous pourrez faire les mesures en utilisant des fonctions de timing au niveau microseconde ou nanoseconde (cycles processeurs). L'exécution des tests sera fixée sur un des cœurs de votre processeur en utilisant la commande `taskset`.

7. Implémentez les fonctions `swap` opérant sur les 4 types de données. Écrivez un programme de test et d'évaluation de ces fonctions.
8. Implémentez les fonctions `copy` opérant sur les 4 types de données. Écrivez un programme de test et d'évaluation de ces fonctions.
9. Implémentez les fonctions `dot` opérant sur les 4 types de données. Écrivez un programme de test et d'évaluation de ces fonctions.
10. Implémentez les fonctions `axpy` opérant sur les 4 types de données. Écrivez un programme de test et d'évaluation de ces fonctions.
11. Implémentez les fonctions `asum` opérant sur les 4 types de données. Écrivez un programme de test et d'évaluation de ces fonctions.
12. Implémentez les fonctions `iamin` opérant sur les 4 types de données. Écrivez un programme de test et d'évaluation de ces fonctions.
13. Implémentez les fonctions `iamax` opérant sur les 4 types de données. Écrivez un programme de test et d'évaluation de ces fonctions.
14. Implémentez les fonctions `nrm2` opérant sur les 4 types de données. Écrivez un programme de test et d'évaluation de ces fonctions.

4 Fonctions BLAS2

15. Vous implémenterez les fonctions de *gemv* opérant sur les 4 types de données. Vous écrirez un programme de test et d'évauation de performance de ces 4 fonctions.

5 Fonctions BLAS3

16. Vous implémenterez les fonctions de *gemm* opérant sur les 4 types de données. Vous écrirez un programme de test et d'évauation de performance de ces fonctions.

6 OpenMP

Dans cette dernière partie du TP2, vous allez insérer des directives OpenMP dans quelques unes des fonctions BLAS que vous avez développées.

17. Ajoutez des pragmas OpenMP dans les fonctions BLAS1 suivantes : *copy*, *dot* et *axpy*. Vous mesurerez les performances avec OpenMP.

18. Ajoutez des pragmas OpenMP dans la fonction BLAS2 suivante : *gemv*. Vous mesurerez les performances avec OpenMP.
19. Ajoutez des pragmas OpenMP dans la fonction BLAS3 suivante : *gemm*. Vous mesurerez les performances avec OpenMP.
20. Pour conclure, comparez les performances des fonctions sur les flottants, double, complexe float et complexe double. Comparez les performances des fonctions BLAS1, BLAS2 et BLAS3, avec et sans OpenMP.