Rapport

Maxime LECHEVALLIER

Jules HABLOT

Rémi GATTAZ

Table des Matières

Configuration notre machine de test

Processeur :

- Fréquence d'Horloge : 3.40GHz
- Cache L3 : 6 Mo
- Mémoire centrale : 8 Go 1600 MHz DDR3
- Support SSE: sse4_2 et inférieur

Système :

CENTOS 7

Compilateur :

gcc (GCC) 4.8.3 20140911 (Red Hat 4.8.3-9)

Taille des primitifs :

- int : 4 octets
- float : 4 octets
- double : 8 octets
- char : 1 octet

La bibliothèque Vecteur

Les constantes :

- SMALLN : entier définissant le nombre d'éléments d'un petit vecteur
- BIGN : entier définissant le nombre d'éléments d'un grand vecteur

Le choix des valeurs pour SMALLN et BIGN est expliqué dans le point "Tailles des Vecteurs et Matrices" dans "Choix d'implémentation".

Les types :

- t_sdVect : petit vecteur de doubles
- t_sfVect : petit vecteur de floats
- t_bdVect : grand vecteur de doubles
- t_bfVect : grand vecteur de floats

Les Fonctions

Pour chacun des types définis, les fonctions suivantes ont été implémentés :
- add : fonction d'addition de deux vecteurs
- mScalaire : multiplication d'un vecteur par un scalaire
- pScalaire : produit scalaire de deux vecteurs.
- read : lecture d'un vecteur depuis stdin
- write : écriture d'un vecteur dans stdout

Puisqu'il a fallu définir ces fonctions pour chacun des types de la librairies mais que la notion de surcharge n'existe pas en C, ces fonctions ont du être nommé différemment pour chaque type. Leurs nommage a été fait de la façon suivantes : `nomFonction_type` .

On trouve par exemple la fonction `add_bdVect` comme étant la fonction d'addition de deux grands vecteurs de doubles.

La bibliothèque Matrice

Les constantes :

- SMALLN_MATR : entier définissant la taille des petites matrices carrées
- BIGN_MATR : entier définissant la taille des grandes matrices carrées

Les types :

- t_sdMatr : petite matrice de double
- t_sfMatr : petite matrice de float
- t_sdMatrVect : ligne ou colone d'une petite matrice de double
- t_sfMatrVect : ligne ou colone d'une petite matrice de floats
- t_bdMatr : grande matrice de double
- t_bfMatr : grande matrice de floats
- t_bdMatrVect : ligne ou colone d'une grande matrice de double
- t_bfMatrVect : ligne ou colone d'une grande matrice de floats

C'est pour les fonctions gaxpy que nous avons du définir les types t_*MatrVect. Notre bibliothèque de vecteur ne permet pas de choisir la taille des vecteurs, nous avons donc du définir des types nous permettant de manipuler des lignes ou colonnes des matrices.

Les Fonctions

Pour chaque type matrices, nous avons défini les fonctions suivantes :
- add : addition de deux matrices
- addVect : version vectorisé de l'addition
- factLU : factorisation LU d'une matrice
- gaxpy : gaxpy
- mScalaire : multiplication par un scalaire d'une matrice
- mult : multiplication de deux matrices
- multVect : version vectorisé de la multiplication de deux matrices
- transpo : tranposition d'une matrice
- read : lecture d'une matrice depuis stdin
- write : écriture d'une matrice dans stdout

Pour chacun des types lignes ou colonne, les fonctions suivantes ont été définies :
- read : lecture d'une ligne ou colonne depuis stdin
- write : écriture d'une ligne ou colonne dans stdout

La convention de nommage des fonction a été la même que pour la librairie vecteur. Nous avons suffixé chaque fonction par le type des arguments qu'elle reçoit et retourne. Nous obtenons donc par exemple `add_bdMatr` .comme étant la fonction d'addition de deux matrice de type t_bdMatr.

Choix d'implémentation

Tailles des Vecteurs et Matrices

SMALLN et BIGN

SMALLN et BIGN sont deux valeurs qui ont été choisis afin que les petits vecteurs tiennent dans le cache L3 du processeur mais que les grands vecteur ne s'y trouve pas.

La fonction add utilise 3 vecteurs en même temps. Les autres fonctions n'utlisent qu'un ou deux vecteurs. De plus, un double utilise plus de place en mémoire qu'un float. Pour le choix de SMALLN, il a donc fallu choisir une valeurs telle que 3 vecteurs de SMALLN doubles puissent contenir dans L3.

La cache L3 de notre machine de tests était de 6Mo. Il était donc nécessaire de choisir SMALLN tel que 3*8*SMALLN < 6 000 000.

Nous avons donc choisi SMALLN = 1 000

La fonction utilisant le moins de vecteurs en simulatané est la fonction mScalaire qui n'en utlise qu'un seul. Et un float utilise moins d'espace en mémoire qu'un double. Pour le choix de BIGN il a donc fallu choisir une valeur telle qu'un vecteur de BIGN floats ne puisse pas être contenu dans L3.

Il était donc nécessaire de choisir BIGN tel que 1*4*BIGN > 6000000

Nous avons donc choisi BIGN = 2 000 0000

En utilisant la commande htop, et en observant la colonne RES, nous avons pu observer que les petits vecteur était effectivement stocké sur L3 puisque non présent dans la mémoire. Au contraire, on a pu observer qu'il était présent en mémoire lors de la manipulation de grands vecteurs.

SMALLN_MATR et BIGN_MATR

Le choix des valeurs pour SMALLN_MATR et BIGN_MATR a été fait avec les mêmes motivations. Nous avons obtenu SMALLN_MATR = 200 et BIGN_MATR = 1250.

Time

Pour calculer les temps d'executions user de chaque fonctions, nous avons définis dans perfs/src/myTime.c des fonctions qui nous permettent d'utiliser les méthodes utilisant les structures timeval.

Ceci nous permets de calculer le temps user plus précisemment qu'avec la commande time car nous pouvons ignorer tous le temps nécéssaire à l'initialisation des données.

read & write

Les fonctions de lectures et d'écriture de vecteurs et de matrices utilisent respectivement les flux stdin et stdout. Il a été envisagé d'utiliser des fichiers à la place. Mais finalement notre choix s'est porté sur cette solution pour qu'il n'y ai pas de gestion de paramètres.

Cette solution n'a pas de désavantage par rapport à l'autre envisagé puisqu'il est possible de remplacer temporairement stdin ou stdout par un fichier.

Tests effectués

Pour tester nos bibliothèques, deux types de programmes de types ont été crées. Les premiers nous ont permis de déterminer que les fonctions et procédures que nous avons faites fonctionnent correctement. Les secondes nous ont permis tester les performances des fonctions.

Tests de fonctionalités

Pour tester les fonctionalités de la bibliothèque, nous avons créé les programmes de tests contenus dans le dossier ./tests/src. Ces programmes sont tous appelés par le fichier bash tests.sh.

Pour compiler les programmes et effectuer les tests de fonctionalités, il faut lancer les commandes suivantes :

```
$ ./tests.sh
```

Le résultat des tests sont stockés dans le fichier ./tests/resultat.txt

Il est possible de générer les programmes de tests avec la target tests dans le makefile.

```
$ make tests
```

Récemment, nous avons pu constater que la compilation des programmes de tests en -O3 engendrait une erreur interne à gcc faisant crasher le compilateur. Ce bug n'a pas encore été rapporté à l'écriture de ce rapport.

Tests de performances

Pour évaluer les performances de nos fonctions, nous avons créé le fichier perf.sh. Avec ce fichier bash, on effectue 1 000 fois chaque opérations. Avant de lancer la commande, il faut construire les fichiers utilisés en utilisant la target perfs du makefile.

```
$ ./perf.sh
```

Le résultat des tests est stocké dans le fichier ./perfs/resultats.csv

Il est possible de générer les programmes de tests performances avec la target test dans le makefile.

```
$ make perfs
```

En modifiant le contenu de la variable CFLAGS dans le makefile, nous avons compilé les programmes de tests sans paramètres d'optimisations puis avec -O1, -O2 et -O3.

Exceptés pour quelques programmes de tests de performances, les programmes de tests effectuent tous 10 000 000 000 d'opérations. Pour chaque programme, pour calculer le nombre de Flops, il a donc fallu diviser le nombre d'opérations effectuées sur des flottants par le temps d'éxécution des programmes.

Etant donnée que les tests de performances sont effectués automatiquement et très facilement, toutes les fonctions de la librairies ont été testés.

Les résultats d'éxécution dans des tableaux et accompagné de graphiques se trouvent dans le fichier /perfs/result.ods.

Interprétation des résultats

Générales :

Le temps d'éxecution d'un programme sans optimisations gcc est toujours bien supérieur au temps d'éxécution avec. Cela se voit particulièrement avec l'opération de multiplication d'un vecteur par un scalaire avec des doubles (mScalaire_sdVect et mScalaire_bdVect). On remarque cependant que la différence des résulats avec les optimisations -O2 et -O3 est très faible.

Les verions openmp de nos fonctions ne semblent pas augmenter les performances comme nous l'attendions. En effet, on observe pour pratiquement toutes les fonctions des baisses de performances nettes. Nous pouvons d'ailleurs faire la même constatations avec la version vectorisés de certaines fonctions.

Vecteur :

- produit sclaire : Les verions vectorisés de cette fonction sont très lent en comparaison de la version non vectorisé. L'optimisation openmp n'est pas non plus correcte.
- addition : Que ce soit avec des floats ou des doubles, la version vectorisé de cette fonction améliore les résultats. Cependant, les optimisations openmp, même alliés à la vectorisation, sont très lentes en comparaison

Sans optimisation, les temps d'éxécutions avec des petits ou grands vecteurs ne sont pas très différents. En revanche, la différence est très nette avec une optimisation. On peut donc penser que sans optimisation, le compilateur n'utilise pas le cache pour stocker les vecteurs qui pourrait y tenir.

On voit que sans optimisations, quelque soit le type de vecteur utilisés, on est entre 300 et 360 Mega flops pour toutes les opérations. En revanche, la multiplication par un scalaire d'un vecteur en utilisant des doubles est beaucoup plus lente. Nous n'avons pas trouvé d'explications à ceci. De plus, avec un paramètre d'optimisation, cette différence disparait.

Pour les petits vecteurs, plus l'optimisation est haute, plus le temps d'éxecution est faible et donc le nombre de flops augmente.

Pour les grands doubles, l'optimisation O3 est toujours moins performante que l'optimisation O2.

Tous les résultats avec openmp que nous avons pu trouvé sont très inférieurs à ceux que nous avons pu observé sans. Ils restent cependant cohérent entre eux. Nous n'avons pas testé sur une autre ordinateur, nous ne pouvons donc pas affirmer qu'il s'agit d'une problème d'implémentation ou de la version d'openmp que nous utilisons.

Les versions vectorisés des fonctions sont un peu plus rapide que les version non vectorisées. Cependant, les différences ne sont pas flagrantes.

Matrice :

- Addition : L'optimisation gcc permet bien d'augmenter le nombre de flops pour cette opération. La version vectorisé de cette fonction est la encore un peu plus rapide que la version non vectorisé.
- Multiplication : Sans optimisation, la multiplication de matrice est très lente. Avec la première optimisation, le nombre de flops a subit une nette amélioration. Les optimisations suivantes n'ont en revanche pas beaucoup affecté ce nouveau résultat.
- Factorisation LU: Nous n'avons pas eu le temps de faire la fonction factorisation_lu avec openmp. De plus, les résultats que nous obtenons sans openMp semblent très loin de la réalité. Nous obtenons des résulats avec plus de 20Giga flops. De plus, la fonction termine toujours presque instantanément alors que, comme pour la multiplication, c'est une opération qui a une complité n^3. Notre implémentation est donc surement incorrecte.