OS202 - Compte rendu du TP n°1

Arthur Bouvet

January 2024

Table des matières

1 Présentation de la machine

Voici la configuration de l'ordinateur sur lequel j'ai effectué mon TP :

```
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          39 bits physic
                          al, 48 bits vi
                          rtual
  Byte Order:             Little Endian
CPU(s):                   16
  On-line CPU(s) list:    0-15
Vendor ID:                GenuineIntel
  Model name:             13th Gen Intel
                          (R) Core(TM) i
                          7-1360P
    CPU family:           6
    Model:                186
    Thread(s) per core:   2
    Core(s) per socket:   8
    Socket(s):            1
    Stepping:             2
    BogoMIPS:             5222.36
```

2 Produit Matrice-Matrice

2.1 Taille de la matrice

Le premier paramètre que nous faisons varier est la taille de la matrice. L'augmentation de la taille de la matrice conduit naturellement à une très forte hausse du temps de calcul. Cette hausse suit assez fidèlement la complexité en $O(n^3)$ associée au nombre d'opérations réalisées : on passe d'un temps de calcul de $2 * 10^{-5}$ s pour une matrice de taille 10x10 à environ 20s pour une matrice de taille 1000x1000.

Nous nous concentrons ensuite sur les cas particuliers des matrices de tailles 1023, 1024 et 1025 pour mettre en évidence le rôle du cache dans le fonctionnement du CPU. Nous répétons 4 fois les exécutions pour chaque valeur :

— 1023 : temps de calcul compris entre 23.6s et 24.9s
— 1024 : temps de calcul compris entre 25.7s et 30.4s
— 1035 : temps de calcul compris entre 22.9s et 24.2s

Même si la différence de valeurs entre les matrices de tailles 1023 et 1025 d'un part et 1024 d'autres part est bien plus faible que celle attendue (en cours, certains ordinateurs affichaient des différences d'un facteur 5), nous observons tout de même que les temps de calcul pour matrices de tailles 1023 et 1025 sont semblables et légèrement plus faible que les temps pour les matrices 1024. Nous pouvons expliquer cela par le recours au cache par le processeur : à chaque fois que l'on accède à une valeur dans la mémoire, les valeurs voisines sont stockées dans le cache, dans l'hypothèse où elles seront utilisées par la suite. Les caches modernes ont généralement une taille équivalente à une puissance de 2 : 128, 256, ... octets. Le cas particulier d'une matrice de taille 1024 (puissance de 2) correspond donc très probablement au cas où les données de la matrice ne s'alignent pas de manière optimale avec la taille du cache, ce qui entraîne des ralentissements à cause d'accès mémoire moins efficaces.

2.2 Ordre des boucles

Une matrice est représentée par un nombre de lignes, un nombre de colonnes et un array contenant les valeurs. Il y a plusieurs possibilités de stocker les valeurs dans un array : mettre les éléments d'une même ligne à côté puis passer à la ligne suivante ou le même fonctionnement mais par colonne. L'accesseur (opérateur ())présent dans la classe matrix du fichier Matrix.hpp nous indique que le stockage se fait ici par colonne. Nous nous attendons donc à ce qu'un parcours de la matrice par colonnes soit plus efficace. En effet, cela permettrait d'optimiser l'utilisation du cache et de l'accès mémoire étant donné que les données lues à la suite (sur une colonne) sont stockées dans le cache (données voisines à la première donnée lue).

Nous intervertissons donc les boucles i, j et k de parcours des éléments de la matrice dans le calcul du produit matriciel afin d'observer cet effet.

— i : nombre de lignes des matrices A et C
— k : nombre de lignes de la matrice B et nombre de colonnes de la matrice A
— j : nombre de colonnes des matrices B et C

Nous nous attendions à ce que les boucles faisant intervenir k et j avant i soient plus efficaces étant donné que les indices k et j correspondent aux colonnes de A et B respectivement. Nous observons une tendance dans ce sens mais la différence entre les moyennes est trop faible pour affirmer que l'effet attendu est mis en évidence par ce test (de la même manière que la différence n'était pas non plus celle attendue dans la première partie).

Ordre	Moyenne des 5 mesures de temps de calcul
i / k / j	11.6s
k / i / j	11.7s
k / j / i	10.6s
j / i / k	12.5s
j / k / i	11.1s

TABLE 1 – Performances des différents ordres d'exécution pour des matrices de taille 1023

NB : le temps de calcul a été divisé par 2 par rapport à la première partie, probablement car j'ai branché mon ordinateur entre temps et donc que les performances de mon PC par défaut n'étaient pas les mêmes.

2.3 Parallélisation des boucles avec OpenMP

Nous considérons l'ordre des boucles k / j / i. Nous ajoutons la ligne suivante afin de paralléliser les boucles k et j en une seule grande boucle :

```
#pragma omp parallel for collapse(2)
```

Ainsi, k traverse toujours les colonnes de A avec un accès optimisé mais cette fois, l'accès aux colonnes de B par j est mieux contrôlé car il se fait en parallèle. Nous pouvons ensuite tester l'influence du nombre de threads en exécutant notre fichier avec la ligne suivante :

```
OMP_NUM_THREADS=4 ./a.out 1023
```

Nous relevons ensuite l'accélération pour les différents nombre de threads : $a_p = \frac{V_{1thread}}{V_{pthreads}}$

Nombre de threads	Temps de calcul	Accélération
1	12.4s	1
4	10.3s	1.2
16	8.6s	1.4

TABLE 2 – Accélération du calcul matriciel en fonction du nombre de threads de la parallélisation

Nous constatons que le temps de calcul diminue lorsque le nombre de threads augmente, une tendance attendue puisque une parallélisation plus importante est censée améliorer les performances du calcul. Néanmoins, nous aurions pu attendre une accélération plus proche du nombre de threads que celle obtenue : lorsque 4 threads sont utilisés pour la parallélisation, le calcul aurait

pû être près de 4 fois plus rapides. En pratique, l'égalité n'est pas parfaite car il y a des retards lorsque plusieurs threads cherchent à accéder aux mêmes données simultanément mais cela n'explique pas les valeurs faibles obtenues pour l'accélération.

Plusieurs éléments nous font penser que, malgré la parallélisation, le calcul matriciel ne soit toujours pas optimal. D'abord, il pourrait être intéressant d'utiliser des blocs de matrice afin de diminuer la distance entre les éléments accédés, améliorant ainsi la localité du cache. Par ailleurs, il faut optimiser le nombre de niveaux de boucles parallélisés ainsi que que le nombre de threads, en particulier lorsque le nombre de threads dépasse le nombre de coeurs physiques disponibles. Enfin, le partitionnement des boucles doit aussi pêtre optimisé afin que certains threads ne soient pas surchargés lorsque d'autres sont en attente.

2.4 Calcul du produit matriciel par blocs

Dans cette sous-partie, nous implémentons le premier élément d'amélioration mentionné dans le paragraphe précédent. Une première fonction calcul le produit matriciel scalaire d'un bloc, elle est équivalent à la fonction scalaire précédemment utilisée mais prend en paramètres les variables qui identifient les blocs concernés dans les matrices d'origine.

Une seconde fonction itère sur les blocs des matrices en faisant appel à la première afin de calculer la produit matriciel global.

```
void prodSubBlocks(int iRowBlkA, int iColBlkB, int iColBlkA, int szBlock,
                   const Matrix& A, const Matrix& B, Matrix& C) {
    // Calcul du produit des sous-blocs
    for (int k = 0; k < A.nbCols / szBlock; ++k) {
        for (int i = iRowBlkA; i < iRowBlkA + szBlock; ++i) {
            for (int j = iColBlkB; j < iColBlkB + szBlock; ++j) {
                for (int l = 0; l < szBlock; ++l) {
                    C(i, j) += A(i, k * szBlock + l) * B(k * szBlock + l, j);
                }}}}}

void prodMatrixMatrixBlocking(const Matrix& A, const Matrix& B, Matrix& C, int szBlock) {
    int n = A.nbRows;
    // Boucles sur les blocs de A et B
    for (int iRowBlkA = 0; iRowBlkA < n; iRowBlkA += szBlock) {
```

```
for (int iColBlkB = 0; iColBlkB < n; iColBlkB += szBlock) {
    for (int iColBlkA = 0; iColBlkA < n; iColBlkA += szBlock) {
        prodSubBlocks(iRowBlkA, iColBlkB, iColBlkA, szBlock, A, B, C);
    }}}}
```

Nous cherchons ensuite à optimiser la taille des blocs : nous calculons donc le temps de calcul pour des blocs de tailles 32, 64, 128, 256.

Taille des blocs	Moyenne des temps de calcul sur 4 mesures
32	10.6s
64	10.9s
128	12.4s
256	12.3s

TABLE 3 – Temps de calcul du produit matriciel par blocs en fonction de la taille des blocs

Nous remarquons (sans forçément pouvoir généraliser les résultats étant donné leur caractère approximatif depuis le début du TP) que les performances sont plus élevées pour des tailles de blocs faibles. Nous pouvons faire l'hypothèse que ces tailles permettent une meilleure utilisation des caches L1 (les plus proches des unités de calcul) étant donné que tout le bloc peut à chaque fois être compris dans le cache, facilitant ainsi les accès mémoires à chaque itération. Il faut tout de même rester vigilants quant à ces résultats car nous pouvons imaginer qu'une faible taille de blocs puisse entraîner une surcharge de calcul due à une augmentation du nombre de blocs à traiter.

Au contaire, les grandes tailles de blocs peuvent entraîner un dépassement de la taille du cache, n'optimisant ainsi pas les accès mémoires et ralentissant le calcul.

2.5 Parallélisation du calcul matriciel par blocs avec OpenMP

Il y a 2 parallélisations possibles pour le calcul par blocs : paralléliser au niveau du calcul scalaire d'un bloc ou au niveau de l'itération sur les différents blocs des matrices. C'est le 2e cas qui nous intéresse car il va en particulier permettre de palier un difficulté pointée dans le paragraphe précédent : des petites tailles de blocs surchargent le calcul en augmentant le nombre de blocs. Traiter plusieurs blocs en même temps avec la parallélisation permettrait donc d'augmenter les performances.

Nous pouvons donc implémenter la parallélisation en ajoutant la même ligne de code qu'à la page 3 avant l'itération sur les blocs dans la fonction prodMatrixMatrixBlocking. La modification du nombre de threads se fait à l'exécution comme précédemment.

NB : Ayant traité cette partie à un autre moment que la parallélisation des blocs, les temps de calculs n'étaient plus comparables. Je me suis donc concentré ici sur l'interprétation des évolutions attendues.

Les temps de calcul pour la parallélisation des blocs font apparaître une plus grande diminution lorsqu'on augment le nombre de threads (24.8s pour 1 threads, 12.3s pour 4 threads). Nous pouvons donc imaginer que l'augmentation du nombre de threads a plus d'influence sur la parallélisation des blocs que la parallélisation scalaire (conclusion à nuancer toujours, du fait de l'inhomogénéité des résultats). En effet, à première vue, lorsqu'on parallélise par blocs, chaque bloc traité séparément est associé à des zones de la matrices différentes donc à des espaces mémoires eux-aussi différents, stockés dans des caches L1 différents, il y aurait donc moins de conflits d'accès mémoire que dans le cas de la parallélisation scalaire.

2.6 Comparaison avec Blas

Nous pouvons imaginer que le produit matriciel blas ne se contente pas d'optimisation ponctuelle comme nous l'avons fait jusqu'à présent mais optimise la gestion du cache à chaque niveau L1, L2, L3...

Le rapport de performances était différent selon le nombre de threads utilisé : notre implémentation par blocs était en moyenne (sur 6 mesures) 1.4 fois plus lente que l'implémentation pour 1 thread contre 1.1 fois plus lente pour 4 threads.

J'ai également cherché à montrer que l'optimisation blas était d'autant plus efficace pour des matrices de grandes tailles (pour lesquelles les performances de notre implémentation diminuaient fortement) mais sans succès.