UNIVERSITÉ CÔTE D'AZUR

MASTER INFORMATIQUE

Rapport de TP2

Transformations de boucles

Réalisé par
Mehdi Mansour

Encadré par
Professeur Sid Touati

Master Informatique
à Université Côte d'Azur
Premier semestre de l'année 2024-2025

Abstract

Ce rapport a pour objectif de détailler les différents résultats obtenus lors de ce TP2 disponible ci-après : https://lms.univ-cotedazur.fr/2024/pluginfile.php/228844/mod_resource/content/2/TransformationsBoucles.pdf.

Architecture utilisée pour ce TP

Architecture : x86_64
nom CPU : Intel(R) Core(TM) i7-10700F CPU @ 2.90GHz
Type de CPU : Intel Cometlake processor
CPU stepping: 5
Hyperthreading activé
Sockets: 1
Cœurs par socket: 8
Threads par cœurs: 2
Socket 0: (0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15)

NUMA :
    Nœud(s) NUMA : 1
    Nœud NUMA 0 de processeur(s) : 0-15
Processeurs: (0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15)
Distances: 10
RAM disponible: 21746.9 MB
RAM Total : 31991.4 MB

Environnement logiciel

Version et distribution Linux : Debian 12

Topologie du PC

Topologie graphique								
Core	0 8	1 9	2 10	3 11	4 12	5 13	6 14	7 15
Cache L1	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB
Cache L2	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB
Cache L3	16 MB							

Contents

Part I
Introduction

L'objectif de ce TP est d'analyser les résultats d'exécution de quatre fichiers C (disponibles dans l'archive fournie) pour évaluer leur performances. Ces fichiers «.c», nommés «tp2_unrolling.c», «tp2_fusion.c», «tp2_interchange.c» et «tp2_tiling.c» sont composés de plusieurs fonctions. Chacun de ces fichiers utilise des techniques d'optimisation de code pour optimiser une fonction multiplication de matrices à valeurs flottantes avec matrices déclarées en statique :

Algorithme 1 : Multiplication de deux matrices C = A × B.

```
1     for(int i=0; i< N; i++){
2         for (int j=0; j< M; j++){
3             for (int k=0; k< P; k++){
4                 C[i][j] = C[i][j] + A[i][k] * B[k][j];
5             }
6         }
7     }
```

Lors de ce TP tous les codes sources seront compilés, avec au plus, l'option d'optimisation -O2, au-delà le compilateur risque d'effectuer des transformations de boucles qui perturberont nos résultats. Nous analyserons l'efficacité de ces techniques d'optimisation de code avec l'aide de la commande *time*.

Part II
Déroulage de boucle (loop unrolling)

Dans cette partie nous réaliserons des benchmarks pour tester l'efficacité de la technique du déroulage de boucle pour avoir un ordre d'idée du temps de calcul de notre programme avec ou sans cette optimisation du code. Tous nos calculs seront faits avec le fichier «tp2_unrolling.c», avec l'option d'optimisation -O2, grâce à la commande *make unrolling*.

1 Déroulage de la boucle interne 7 fois

L'objectif de cet exercice est de dérouler la boucle interne de notre fonction 7 fois (ici la boucle interne est k). Le déroulage de boucle n fois consiste à dupliquer le corps de boucle n-1 fois (ici n-1=7), on obtient donc n copies (ici n=8).
On obtient donc la boucle k suivante :

```
for (k; k< P; k+=8){
        C[i][j] += A[i][k] * B[k][j];
        C[i][j] += A[i][k+1] * B[k+1][j];
        C[i][j] += A[i][k+2] * B[k+2][j];
        C[i][j] += A[i][k+3] * B[k+3][j];
        C[i][j] += A[i][k+4] * B[k+4][j];
        C[i][j] += A[i][k+5] * B[k+5][j];
        C[i][j] += A[i][k+6] * B[k+6][j];
        C[i][j] += A[i][k+7] * B[k+7][j];
    }
```

2 Benchmark déroulage de boucle

Les benchmarks sont réalisés grâce à la commande *make unrolling*, on obtient les résultats dans un fichier nommé «unrolling.txt» disponible dans le dossier *res/*. En voici un extrait rangé dans un tableau :

2.1 Code avec et sans déroulage de boucle

Version	Temps Total	Temps Utilisateur	Temps Système
sans déroulage	3,769s	3,765s	0,004s
avec déroulage	3,683s	3,679s	0,004s

Table 1: Comparaison entre avec déroulage de boucle et sans

On a un facteur d'accélération de 1,02, soit une légère amélioration.

2.2 Déroulage de boucle avec option à la compilation

Version	Temps Total	Temps Utilisateur	Temps Système
sans déroulage	3,769s	3,765s	0,004s
avec déroulage	3,683s	3,679s	0,004s
-funroll-loops	3,789s	3,785s	0,004s

Table 2: Exécution 1

Pour l'Exécution 1, l'option ne donne pas de bon résultat mais pour la suivante :

Version	Temps Total	Temps Utilisateur	Temps Système
sans déroulage	4,439s	4,435s	0,004s
avec déroulage	4,137s	4,129s	0,008s
-funroll-loops	4,063s	4,056s	0,008s

Table 3: Exécution 2

On a un facteur d'accélération de 1,1 avec l'option. Avec déroulage on a un facteur de 1,07. La différence ici est légère.

Part III
Fusion de boucles (loop fusion)

Dans cette partie nous réaliserons des benchmarks pour tester l'efficacité de la technique de fusion de boucles. Cette technique consiste à fusionner plusieurs boucles, qui parcourent les mêmes plages d'itération, en une. Tous nos calculs seront faits avec le fichier «tp2_fusion.c», sans option d'optimisation, grâce à la commande *make fusion*.

1 Déroulage boucle externe 2 fois

A partir de la fonction de multiplication de matrices on obtient, ci-dessous, ce code en déroulant deux fois la boucle externe i :

```
for (i=0; i< N; i+=3){
    for (j=0; j< M; j++){
        for (k=0; k<P; k++){
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
    if (i+1 < N){
        for (j=0; j< M; j++){
            for (k=0; k<P; k++){
                C[i+1][j] = C[i+1][j] + A[i+1][k] * B[k][j];
            }
        }
    }
    if (i+2 < N) {
        for (j=0; j< M; j++){
            for (k=0; k<P; k++){
                C[i+2][j] = C[i+2][j] + A[i+2][k] * B[k][j];
            }
        }
    }
}
```

2 Fusion des boucles j

A partir du code précédent, on obtient ce code en fusionnant les trois boucles j en une:

```
for (i = 0; i < N; i += 3) {
    for (j = 0; j < M; j++) {
        for (k = 0; k < P; k++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
        if (i + 1 < N) {
            for (k = 0; k < P; k++) {
                C[i+1][j] = C[i+1][j] + A[i+1][k] * B[k][j];
            }
        }
        if (i + 2 < N) {
```

```
12              for (k = 0; k < P; k++) {
13                  C[i+2][j] = C[i+2][j] + A[i+2][k] * B[k][j];
14              }
15          }
16      }
17  }
```

3 Benchmark fusion de boucle

Les benchmarks sont réalisés grâce à la commande *make fusion*, on obtient les résultats dans un fichier nommé «fusion.txt» disponible dans le dossier *res/*. En voici un extrait rangé dans un tableau :

Version	Temps Total	Temps Utilisateur	Temps Système
sans déroulage	38,203	38,198	0,004s
avec déroulage	40,240	40,158	0,080s
avec fusion	36,484	36,475	0,008s

Table 4: Comparaison du temps total entre trois versions de codes différentes

On remarque que l'exécution du code avec fusion est la plus rapide des trois. On remarque aussi que le code avec déroulage de boucle est plus long que le code sans technique d'optimisation, sûrement parce que c'est la boucle externe qui est déroulée.

Part IV
Permutation de boucles (loop interchange)

Dans cette partie nous réaliserons des benchmarks pour tester l'efficacité de la technique de permutation de boucles. Cette technique consiste à permuter deux boucles entre elles. Dans notre cas on peut faire 3! permutations soit 6 permutations toutes correctes. Tous nos calculs seront faits avec le fichier «tp2_interchange.c», avec option d'optimisation -O2, grâce à la commande *make interchange*.

1 Le meilleur ordre

L'ordre (i,k,j) est à priori l'ordre qui apporterait les meilleurs performances. Cet ordre permet de lire les matrices A[i][k], C[i][j] et B[k][j] plus rapidement car la mémoire est mieux utilisée, on a moins d'aller-retour vers la mémoire.

2 Benchmark permutation de boucles

Les benchmarks sont réalisés grâce à la commande *make interchange*, on obtient les résultats dans un fichier nommé «interchange.txt» disponible dans le dossier *res/*. En voici un extrait rangé dans un tableau :

Version	Temps Total	facteur d'accélération
ijk	3,767s	\
ikj	1,910s	1,97
kij	2,059s	1,83
kji	5,078s	0,74
jik	11,957s	0,32
jki	18,544s	0,20
-floop-interchange	0,010s	377

Table 5: Comparaison entre les différentes permutations

Comme dit précédemment, l'ordre (i,k,j) est le meilleur en terme de performance avec un facteur d'accélération de 1,97. On remarque également que certaines permutations n'améliorent pas le temps total mais au contraire l'augmentent. C'est le cas de (k,j,i), (j,i,k) et (j, k, i). On remarque surtout que l'option *-floop-interchange* est beaucoup plus performantes que n'importe quelle permutation, avec un facteur d'accélération de 377, ce qui est conséquent.

Part V
Tuilage ou blocage de boucle (loop tiling, loop blocking)

Dans cette partie nous réaliserons des benchmarks pour tester l'efficacité de la technique de tuilage de boucles. Cette technique consiste à transformer le code pour que la boucle la plus interne puisse faire des calculs sur un ensemble réduit de données. Tous nos calculs seront fait avec le fichier «tp2_tiling.c», avec option d'optimisation -O2, grâce à la commande *make tiling*.

1 Pseudo-code

Voici un exemple de code utilisant cette technique :

```
1 for (i0 = 0; i0 < N; i0 += B)
2     for (j0 = 0; j0 < M; j0 += B)
3         for (k0 = 0; k0 < P; k0 += B)
4             for (i = i0; i < min(i0 + B, N); i++)
5                 for (j = j0; j < min(j0 + B, M); j++)
6                     for (k = k0; k < min(k0 + B, P); k++)
7                         C[i][j] += A[i][k] * B[k][j];
```

Où B est le bloc. On le calcul grâce à la formule suivante :

$$3 \times B^2 \times \text{sizeof(float)} \leq C$$

avec C le capacité du cache de données en octets, sizeof(float) = 4 octets. Comme on veut B on va faire le calcul suivant :

$$B \leq sqrt(\frac{C}{12})$$

Dans notre cas :

- B= 52 pour le cache L1
- B= 147 pour le cache L2
- B= 1182 pour le cache L3

2 Benchmark tuilage de boucle

Dans cet exercice nous allons également utiliser les permutations de boucles pour améliorer les performances de notre code. Les benchmarks sont réalisés grâce à la commande *make tiling*, on obtient les résultats dans un fichier nommé «tiling.txt» disponible dans le dossier *res/*. En voici un extrait rangé dans un tableau :

Version	Temps Total L1	Temps Total L2	Temps Total L3
sans tuilage	1,188s	1,179s	1,168s
ijk	3,046s	3,968s	4,559s
ikj	2,652s	2,731s	2,421s
kij	2,397s	2,627s	2,517s
kji	3,529s	4,293s	4,711s
jik	2,285s	3,047s	5,699s
jki	2,599s	2,982s	5,418s
-floop-block	4,623s	4,989s	4,756s
-floop-block et -floop-interchange	5,050s	4,806s	5,026s

Table 6: Comparaison entre les différentes permutations et options

On remarque qu'avec tuilage de boucle c'est plus long mais c'est sûrement parce que le jeu de données est trop faible pour être efficace. On remarque aussi qu'un tuilage de boucle source est plus efficace que l'option -floop-block dédiée pour ça. On remarque qu'en général les temps les plus élevés sont de ceux avec un bloc de niveau L3.

Part VI
Conclusion

Ces techniques d'optimisation de code sont utiles pour améliorer les performances générales d'un code. Certaines, dans le cadre de ce TP sont plus efficace que d'autres, c'est notamment le cas de la permutation et de la fusion de boucle. Dans notre cas, notre code est petit, avec un jeu de données faible, cela peut donc expliquer les résultats peu concluants pour les techniques de tuilage de boucles (temps plus long que sans tuilage) ou encore le déroulage de boucle (amélioration minime du temps de calcul).
Ces techniques peuvent, combinées entre elles, améliorer les performances de nos codes. Il faut pour cela étudier en amont notre code, réaliser plusieurs benchmark sur plusieurs versions du code (si on le peut en terme de temps et de coût financier).