UNIVERSITÉ CÔTE D'AZUR

MASTER INFORMATIQUE

Rapport de TP1

Évaluation des performances d'un programme et optimisations de code par compilation

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 TP1 disponible ci-après : https://lms.univ-cotedazur.fr/2024/pluginfile.php/228842/mod_resource/content/10/Eval-Perf.pdf.

Architecture utilisé pour ce TP

CPU name: Intel(R) Core(TM) i7-10700F CPU @ 2.90GHz
CPU type: Intel Cometlake processor
CPU stepping: 5

Hardware Thread Topology

Sockets: 1
Cores per socket: 8
Threads per core: 2
Socket 0: (0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15)

NUMA domains: 1
Domain: 0
Processors: (0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15)
Distances: 10
Free memory: 21746.9 MB
Total memory: 31991.4 MB

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 deux fichiers C (disponibles dans l'archive fournisse) pour évaluer leur performances. Ces fichier «.c», nommés «dynamique.c» et «statique.c» sont composés de plusieurs fonctions décrient dans le sujet du TP :

- Addition de deux vecteurs de valeurs flottantes C = A + B
- Multiplication de deux matrices C = A × B
- Multiplication d'un vecteur par un scalaire A = A × s

Lors de ce tp nous utiliserons plusieurs outils comme la commande *time* ou *gprof* pour analyser notre code, ainsi que des options d'optimisations pour améliorer sa rapidité d'exécution.

Ces analyses sont nécessaires pour optimiser les bouts de code qui prennent beaucoup de temps. Nous pouvons à la compilation, réduire ce temps de calcul grâce à des options d'optimisation que nous développerons plus tard (voir man gcc).

Part II
Réalisation des benchmarks

Dans cette partie nous réaliserons des benchmarks pour avoir un ordre d'idée du temps de calcul de notre programme. Un benchmark est une série de tests réalisée pour analyser les performances d'un appareil informatique, composants (ex : carte graphique) ou dans notre cas d'un programme.

1 Détail des fonctions

Dans cette partie nous allons détaillé les fonctions décrient dans l'introduction. Nous utiliserons des matrices et vecteurs avec des données déclarées statiquement comme globales et des données allouées dynamiquement avec malloc. Les données sont des valeurs flottantes. Dans chaque fonctions, on print dans un fichier une valeur pour éviter les codes morts et faire en sorte que le compilateur exécute exactement ce que l'on veut.

1.1 Addition de deux vecteurs de valeurs flottantes C = A + B

Voici la fonction d'addition de deux vecteurs.

Algorithme 1 : Addition de deux vecteurs

```
void add() {
        freopen("error.log", "w", stderr);
    for (int i = 0; i < N; i++) {
        C1[i]=A1[i]+B1[i];
        fprintf(stderr,"%f",C1[i]);
    }
    fclose(stderr);
}
```

1.2 Multiplication de deux matrices C = A × B

Voici la fonction qui fait la multiplication de deux matrices.

Algorithme 2 Retourne la valeur maximale du tableau tab.

```
void multMatrice(){
    freopen("error.log", "w", stderr);
    for(int i=0; i< N; i++){
        for (int j=0; j< M; j++){
            for (int k=0; k< P; k++){
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
            fprintf(stderr,"%f",C[i][j]);
        }
    }
    fclose(stderr);
}
```

1.3 Multiplication d'un vecteur par un scalaire A = A × s

Voici la fonction de multiplication d'un vecteur.

Algorithme 3 Retourne la valeur maximale du tableau tab.

```
1 void mult_scalaire(){
2     freopen("error.log", "w", stderr);
3     for(int i=0;i<N;i++){
4         As[i]=As[i]*s;
5         fprintf(stderr,"%f",As[i]);
6     }
7     fclose(stderr);
8 }
```

2 Mesures des performances et profilage d'un programme

2.1 Temps d'exécution

Pour évaluer le temps d'exécution de notre programme, nous utiliserons la commande *time*. Pour plus d'efficacité nous exécuterons un makefile pour effectuer nos commandes.

2.1.1 fonction addition

Pour la fonction addition on obtient ces résultats (exécuté avec la ligne de commande *make test_add*, voir résultat dans *resultat/res_test_add.txt*) :

Version	Temps Total	Temps Utilisateur	Temps Système
Statique	0,548s	0,400s	0,148s
Dynamique	0,454s	0,281s	0,173s

Table 1: Comparaison entre programme statique et dynamique (Exécution 1)

Version	Temps Total	Temps Utilisateur	Temps Système
Statique	0,562s	0,426s	0,136s
Dynamique	0,441s	0,297s	0,144s

Table 2: Comparaison entre programme statique et dynamique (Exécution 2)

On voit une légère différence entre les données statiques et celles allouées dynamiquement avec malloc.

- le temps real représente le temps total effectué (en seconde).
- le temps user représente le temps CPU utilisateur (exécution des instructions du programme)
- le temps sys représente le temps CPU système (appel système, opération entrées/sorties ...)

On a pour la première exécution, avec données statiques, un temps total de 0,548s secondes. Le temps utilisateur représente ici environ 73% du temps total ((0,400/0,548)*100=72.9). Le temps système quant à lui représente environ 27%. 62% environ pour le temps utilisateur avec données allouées dynamiquement.

On remarque également que pour chaque exécution, on a des résultats différents et pour une raison simple : il y'a d'autres processus qui s'exécutent en même temps que notre programme, ce qui explique ces différences. Par exemple pour la deuxième exécution, le temps utilisateur de la version statique représente 75% du temps total, soit 2 points de différence

2.1.2 fonction multiplication matrice

Pour la fonction multiplication on obtient ces résultats (exécuté avec la ligne de commande *make test_mult*, voir résultat dans *resultat/res_test_mult.txt*):

Version	Temps Total	Temps Utilisateur	Temps Système
Statique	18,018s	17,749s	0,240s
Dynamique	33,561s	33,334s	0,204s

Table 3: Comparaison entre programme statique et dynamique (Exécution 1)

Version	Temps Total	Temps Utilisateur	Temps Système
Statique	17,303s	17,007s	0,192s
Dynamique	28,881s	28,570s	0,232s

Table 4: Comparaison entre programme statique et dynamique (Exécution 2)

Pour la version statique, on a un temps utilisateur qui représente 98% du temps total contre 2% pour le temps système. Ces résultats nous montrent que l'exécution de la fonction multiplication de matrices est très longue. On remarque également que le temps d'exécution est différent d'une exécution à une autre.

2.1.3 fonction scalaire

Pour la fonction multiplication d'un vecteur par un scalaire on obtient ce résultat (exécuté avec la ligne de commande *make test_scal*):

Version	Temps Total	Temps Utilisateur	Temps Système
Statique	0,471s	0,327s	0,144s
Dynamique	0,425s	0,284s	0,140s

Table 5: Comparaison entre programme statique et dynamique (Exécution 1)

Version	Temps Total	Temps Utilisateur	Temps Système
Statique	0,480s	0,344s	0,136s
Dynamique	0,418s	0,287s	0,131s

Table 6: Comparaison entre programme statique et dynamique (Exécution 2)

Pour la première exécution version statique, le temps utilisateur représente 69%, le temps système 31% du temps total. Grâce à ces benchmarks on sait que notre programme va prendre beaucoup de temps à faire la multiplication de matrices. On sait donc quelle partie du code optimiser.

2.2 Calcul des métriques de performances

Voici ci-dessous les formules des métriques suivantes : CPI, IPC et GFLOPS

$$\text{CPI(P,I,A)} = \frac{\text{TempsExecution}(P, I, A) \times \text{FrequenceHorloge}(A)}{\text{NombreInstructions}(P, I, A)}$$

$$\text{IPC(P,I,A)} = \frac{1}{\text{CPI(P,I,A)}}$$

$$\text{GFLOP} = \frac{\text{Nombre total d'opérations flottantes}}{\text{Temps d'exécution (en secondes)} \times 10^9}$$

2.3 Profilage d'un programme

Pour cet exercice, nous allons profiler avec l'outil *gprof*, le fichier «dynamique.c» avec l'option *-pg* pour permettre à *gprof* de récupérer les données qu'il souhaite. Voici la suite de commande que l'on exécute dans le terminal (*make gprof*, résultat dans *resultat/res.txt*):

```
gcc -pg -O0 dynamique.c -o dynamique_gprof
./dynamique_gprof
gprof -b dynamique\_gprof gmon.out > resultat/res.txt
```

Voici ci-dessous un extrait de *res.txt* :

```
time   seconds   seconds    calls   s/call   s/call  name
 98.33     23.68    23.68        1    23.68    23.68  multMatrice
  1.66     24.08     0.40        1     0.40     0.40  initialiserMatrices
  0.00     24.08     0.00        1     0.00     0.00  add
  0.00     24.08     0.00        1     0.00     0.00  mult_scalaire

  granularity: each sample hit covers 2 byte(s) for 0.04% of 24.08
      seconds
```

En regardant la colonne «calls», on remarque que 4 fonctions ont été exécuté, celle de la colonne «name» classé dans l'ordre décroissant du temps d'exécution. Ce qui veut dire que «multMatrice» met 23.68s sur 24.08s pour s'exécuter soit 98,33% du temps. Les fonctions «add» et «mult_scalaire» quant à elles, sont à 0 car imperceptible par l'outil. Leur temps d'exécution est très faible.

Part III
Optimisations de code

3 Optimisations de code avec le compilateur gcc et icx

Pour cet exercice, nous allons exécuter un script (ici makefile) pour effectuer nos calculs efficacement, l'objectif étant de faire une comparaison entre les 4 options d'optimisation (-O0, -O1, -O2, -O3, -Os) et les compilateurs gcc et icx. Les résultats obtenus sont redirigés dans des fichiers disponibles dans le dossier *resultat/*.

3.1 res_time_statique.txt et res_time_dynamique.txt

Ces deux fichiers se présentent comme suit : en premier, les 4 exécutions, pour les 4 options d'optimisation différentes, avec le compilateur gcc puis 4 exécutions avec le compilateur icx et enfin une exécution avec l'option *-mkl* d'icx. Les 3 fonctions sont exécutées ensemble.

3.1.1 res_time_statique.txt

Voici un exemple des résultats obtenus décrit dans res_time_statique.txt:

Temps total / compilateur	-O0	-O1	-O2	-O3	-Os	-mkl
gcc	18,981s	6,953s	6,346s	6,407s	6,430s	\
icx	14,873s	6,800s	3,627s	3,515s	6,043s	3,417s
taux accélération gcc	1	2,73	2.99	2,96	2,95	\
taux accélération icx	1	2,19	4,1	4,23	2,46	4,35

Table 7: Comparaison entre gcc et icx pour différentes options d'optimisation, version statique

3.1.2 res_time_dynamique.txt

Voici un exemple des résultats obtenus décrit dans res_time_dynamique.txt:

Temps total / compilateur	-O0	-O1	-O2	-O3	-Os	-mkl
gcc	28,914s	14,813s	8,845s	8,925s	11,941s	\
icx	27,305s	9,197s	9,924s	10,012s	9,614s	9,330s
taux accélération gcc	1	1,95	3,27	3,24	2,42	\
taux accélération icx	1	2,97	2,75	2,72	2,84	2,93

Table 8: Comparaison entre gcc et icx pour différentes options d'optimisation, version dynamique

3.2 res_time_add.txt, res_time_mult.txt et res_time_scal.txt

Les fichiers se présentent comme suit : dans un premier temps, Les exécution statiques avec, les 4 exécutions, pour les 4 options d'optimisation différentes, avec le compilateur gcc puis 4 exécutions avec le compilateur icx. Dans un second temps les exécutions dynamiques dans le même ordre que les statiques.

3.2.1 res_time_add.txt

Voici un exemple des résultats obtenus décrit dans «res_time_add.txt»:

Temps total / compilateur	-O0	-O1	-O2	-O3	-Os
gcc	0,553s	0,397s	0,272s	0,301s	0,332s
icx	0,486s	0,330s	0,267s	0,264s	0,342s
taux accélération gcc	1	1,4	2,034	1,840	1,667
taux accélération icx	1	1,473	1,818	1,838	1,419

Table 9: Comparaison entre gcc et icx pour différentes options d'optimisation, version statique

Temps total / compilateur	-O0	-O1	-O2	-O3	-Os
gcc	0,467s	0,406s	0,361s	0,235s	0,358s
icx	0,479s	0,412s	0,220s	0,223s	0,357s
taux accélération gcc	1	1,150	1,294	1,987	1,305
taux accélération icx	1	1,163	2,173	2,148	1,344

Table 10: Comparaison entre gcc et icx pour différentes options d'optimisation, version dynamique

3.2.2 res_time_mult.txt

Voici un exemple des résultats obtenus décrit dans «res_time_mult.txt»:

Temps total / compilateur	-O0	-O1	-O2	-O3	-Os
gcc	24,695s	7,677s	7,206s	8,028s	7,575s
icx	19,513s	6,747s	4,068s	4,207s	7,302s
taux accélération gcc	1	3,22	3,42	3,08	3,26
taux accélération icx	1	2,89	4,79	4,64	2,67

Table 11: Comparaison entre gcc et icx pour différentes options d'optimisation, version statique

Temps total / compilateur	-O0	-O1	-O2	-O3	-Os
gcc	37,611s	21,004s	10,452s	10,214s	13,864s
icx	34,989s	10,889s	10,294s	10,831s	10,744s
taux accélération gcc	1	1,79	3,60	3,68	2,71
taux accélération icx	1	3,22	3,40	3,23	3,26

Table 12: Comparaison entre gcc et icx pour différentes options d'optimisation, version dynamique

3.2.3 res_time_scal.txt

Voici un exemple des résultats obtenus décrit dans «res_time_scal.txt»:

Temps total / compilateur	-O0	-O1	-O2	-O3	-Os
gcc	0,515s	0,411s	0,270s	0,299s	0,328s
icx	0,485s	0,339s	0,254s	0,298s	0,324s
taux accélération gcc	1	1,25	1,91	1,72	1,57
taux accélération icx	1	1,43	1,91	1,63	1,50

Table 13: Comparaison entre gcc et icx pour différentes options d'optimisation, version statique

Temps total / compilateur	-O0	-O1	-O2	-O3	-Os
gcc	0,469s	0,380s	0,377s	0,247s	0,413s
icx	0,476s	0,363s	0,357s	0,239s	0,369s
taux accélération gcc	1	1,24	1,24	1,90	1,14
taux accélération icx	1	1,31	1,33	1,99	1,29

Table 14: Comparaison entre gcc et icx pour différentes options d'optimisation, version dynamique

Part IV
Man gcc

Pour cette partie nous utilisons le script «option.sh» pour exéctuer la commande

```
gcc -Q --help=optimizers -O<N> | grep enable | awk '{print $1}'
avec N ={1, 2, 3, s}
```

pour les options O1, O2, O3 et Os. La liste pour chaque option ce trouve dans le dossier *resultat/*. Les fichiers sont sous la forme «liste_option_ON.txt».

Part V
Librairie MKL d'Intel

Dans cette section nous allons comparer les résultats obtenus avec le compilateur icx avec les options -O3 et -mkl. Ces résultats sont déjà présenté dans la section 3.1. On remarque que les temps totaux avec l'options -mkl sont meilleurs que ceux avec l'option -O3.

Part VI
Compteurs matériels de performances

Dans cette section nous allons analyser les événements matériels provoqués par le processus statique_gcc_O0. Avec la commande perf suivante :

```
perf stat -e cycles,cache-misses,instructions,L1-dcache-load-misses,L1-
    icache-load-misses,branch-instructions,branch-loads,node-loads ./
    statique_gcc_O0
```

on obtient le résultat illustré sur la photo «perfStat.png» disponible dans le dossier *resultat/* où l'on voit le détail des événements matériels enregistrés.

Avec la commande

```
perf record -e cycles,cache-misses,instructions,L1-dcache-load-misses,L1-
    icache-load-misses,branch-instructions,branch-loads,node-loads ./
    statique_gcc_O0
```

on obtient un fichier «perf.data_statique_O0» qui grâce à la commande

```
perf report perf.data\_statique\_O0
```

permet de visualisé également des événements matériels. On peut voir le résultat de cette action sur la photo «perfReport.png» disponible dans le dossier *resultat/*.