UNIVERSITÉ CÔTE D'AZUR

MASTER INFORMATIQUE

Rapport de TP

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

Réalisé par
Francois Flandin

Encadré par
Pr Sid Touati

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

Table des matières

1 Introduction

Dans le cadre de ce travail pratique, nous avons exploré les concepts liés à l'évaluation des performances des programmes informatiques et à leur optimisation par la compilation.
Ces compétences sont essentielles pour développer des applications efficaces, capables d'exploiter au mieux les ressources matérielles disponibles.
Ce TP se concentre sur plusieurs aspects clés, notamment la réalisation de benchmarks, le profilage des programmes, et l'utilisation des options d'optimisation des compilateurs.
L'objectif principal de ce rapport est de détailler les expériences menées, d'interpréter les résultats obtenus, et d'évaluer les bénéfices des diverses optimisations.

Environnement Expérimental

Micro-Architecture

Dans cette partie sera detaillée la micro-architecture de la machine de tests.

Nom de modèle : 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
Taille des adresses : 39 bits physical, 48 bits virtual
Coeurs physiques : 4
Taille de ligne de cache : 64 octets

Graphical Topology				
Coeurs	0 4	1 5	2 6	3 7
Cache L1	48 kB	48 kB	48 kB	48 kB
Cache L2	1MB	1MB	1MB	1MB
Cache L3	8 MB			

Table 1 – Topologie de la machine de tests

Environnement Logiciel

Dans cette partie sera détaillée la partie logiciel de la machine de test, les fichiers scripts pour changer entre machine allégée et machine classique sont fournis dans l'archive.

Distribution : Fedora v40 WorkStation
Compilateur utilisé : `gcc` version 14.2.1 20240912 avec option `-O0`
Outil de mesure des temps d'exécution : commande `time`.

Configuration expérimentale

Processus en activité

Les benchmarks n'ont pas été efféctués sur un environnement allégé, pour la curiosité on laissera un dossier `data_allegee` avec les résultats des benchmarks réalisés sur une configuration minimale.
De part la nature non-allégée de la machine de tests, il ya beaucoup de processus en activité, dont des éditeurs de texte, le terminal, un navigateur et autres applications de messagerie.

Méthodologie de récolte des données expérimentales

Pour mesurer le temps d'exécution des différents benchmarks `matrix_multiply`, `vec_add`, `sc_vec_mult`, on va utiliser plusieurs la commande `time`, executee un total de 4 fois. Le resultat de ces commandes est fourni dans l'archive. Les résultats des benchmarks sont ensuite envoyés dans fichier `data.txt`, que l'on pourra interpréter avec le programme python `export_time_data.py`. Le programme se charge de calculer la moyenne des temps globaux de chaque benchmark, ainsi que de calculer le taux d'accélération de chaque programme par rapport au programme sans niveau d'optimisation spécifié.

2 Mesures des performances et profilage d'un programme

2.1 Temps d'exécution des programmes

Dans cette partie, nous allons voir des benchmarks sur les trois programmes *matrix multiply addition de vecteurs* et *multiplication de vecteur par un scalaire*.

2.1.1 Multiplication de matrices

Version	Temps d'exécution global moyen	Temps d'exécution utilisateur moyen	Proportion utilisateur	Temps d'exécution système moyen	Proportion système
Statique	23.55s	23.48s	99.7%	0,0175s	0.00074%
Dynamique	27.65s	27.65s	99.3%	0,0275s	0.00099%

Analyse : On constate donc que pour la multiplication de matrices, la version statique est plus performante que la version dynamique, d'une part par la moyenne des temps d'exécution, en moyenne la version statique gagne 4.17 secondes.
D'autre part, la version statique a une plus grande proportion de son temps de calcul dans le code utilisateur, même si la différence est petite (0.4%) et que cela pourrait être dû à un changement de contexte, on remarque aussi que le temps d'exécution système est plus grand pour la version dynamique.

2.1.2 Multiplication d'un vecteur par un scalaire

Version	Temps d'exécution global moyen	Temps d'exécution utilisateur moyen	Proportion utilisateur	Temps d'exécution système moyen	Proportion système
Statique	2.5185s	2.06s	81%	0.4575s	18%
Dynamique	2.354s	1.8875s	80%	0.4625s	19%

Analyse : On constate donc que pour la multiplication de vecteur par un scalaire, la version dynamique est cette fois plus performante, mais uniquement sur le plan du temps d'exécution, on constate une baisse de presque 0.2 secondes par rapport a l'exécution statique, cependant, du côté de la proportion temps utilisateur/temps système, la version statique l'emporte, avec un gain de 1% sur le temps utilisateur.

2.1.3 Addition de deux vecteurs

Version	Temps d'exécution global moyen	Temps d'exécution utilisateur moyen	Proportion utilisateur	Temps d'exécution système moyen	Proportion système
Statique	1.72875s	1.045s	60%	0.6825s	39%
Dynamique	1.8175s	1.135s	62%	0.6825s	37%

Analyse : On constate donc que pour l'addition de vecteurs, la version statique est la plus performante, mais uniquement sur le plan de la vitesse d'exécution, on constate une baisse de presque 0.1 seconde, à ce niveau-là c'est peut-être une erreur du benchmark, mais ce n'est pas non plus un gain de performance important.

2.2 Comment calculer l'IPC, le CPI et le GFLOPS

2.2.1 Calculer l'IPC

La formule pour calculer l'IPC est la suivante :

$$IPC = \frac{nombre\_d'instructions}{nombre\_de\_cycles}$$

2.2.2 Calculer le CPI

La formule pour calculer le CPI est la suivante :

$$CPI = \frac{nombre\_de\_cycles}{nombre\_d'instructions}$$

2.2.3 Calculer le GFLOPS

La formule pour calculer le GFLOPS est la suivante :

$$GFLOPS = \frac{Nombre\_de\_calculs\_flottants}{Temps\_d'execution}$$

3 Optimisations de code avec le compilateur gcc

3.1 Comparaison des niveaux d'optimisation

Dans cette partie, nous allons voir différents benchmarks sur les différents programmes, a plusieurs niveaux d'optimisation avec gcc.

3.1.1 Matrix Multiply

Version statique

Niveau optimisation	Moyenne temps globaux	Taux d'accélération
Aucun	**23.55**	**1**
O1	12.58	1.87
O2	4.4775	5.25
O3	4.325	5.44
Os	12.87	1.82

Après analyse des résultats, on peut en conclure que pour la version statique de la multiplication de matrices, le plus optimisé serait *-O3* avec un taux d'accélération de 5.44.
Cependant on remarque aussi *-O2* qui possède quand a lui un taux d'accélération de 5.25, ce qui reste relativement proche.

Version dynamique

Niveau d'optimisation	Moyenne temps globaux	Taux d'accélération
Aucun	**27.83**	**1**
O1	20.24	1.375
O2	13.29	2.09
O3	13.4325	2.07
Os	20.825	1.33

Après analyse des résultats, on peut en conclure que pour la version dynamique de la multiplication de matrices, le plus optimisé serait *-O2* avec un taux d'accélération de 2.09.
Cependant on remarque aussi *-O3* qui possède quand a lui un taux d'accélération de 2.07, qui est extrêmement proche, on ne peut donc pas dire lequel est le plus efficace, car cette très légère différence pourrait venir soit de la commande time, soit du contexte d'exécution.

3.1.2 Addition de vecteurs

Version statique

Niveau d'optimisation	Moyenne temps globaux	Taux d'accélération
Aucun	1.72875	1
O1	1.05625	1.63
O2	0.897	1.92
O3	0.894	1.93
Os	1.05125	1.64

Après analyse des résultats, on peut en conclure que pour la version statique de l'addition de vecteurs, le plus optimisé serait *-O3* avec un taux d'accélération de 1.93.
Cependant on remarque aussi *-O2* qui possède quand a lui un taux d'accélération de 1.92, qui est extrêmement proche, on ne peut donc pas dire lequel est le plus efficace, car cette très légère différence pourrait venir soit de la commande time, soit du contexte d'exécution.

Version dynamique

Niveau optimisation	moyenne temps global	taux d'accélération
Aucun	1.8175	1
O1	1.11425	1.63
O2	0.93525	1.94
O3	0.8975	2.02
Os	1.031	1.76

Après analyse des résultats, on peut en conclure que pour la version dynamique de l'addition de vecteurs, le plus optimisé serait *-O3* avec un taux d'accélération de 2.02.
Cependant on remarque aussi *-O2* qui possède quand a lui un taux d'accélération de 1.94, ce qui reste relativement proche.

3.1.3 Multiplication de vecteur par un scalaire

Version statique

Niveau d'optimisation	Moyenne temps globaux	Taux d'accélération
Aucun	2.5185	1
O1	0.82225	3.06
O2	0.666	3.78
O3	0.675	3.73
Os	0.827	3.04

Après analyse des résultats, on peut en conclure que pour la version statique de la multiplication de vecteur par un scalaire, le plus optimisé serait *-O2* avec un taux d'accélération de 3.78. Cependant on remarque aussi *-O3* qui possède quand a lui un taux d'accélération de 3.73, ce qui reste relativement proche.

Version dynamique

Niveau optimisation	Moyenne temps global	Taux d'accélération
Aucun	2.354	1
O1	0.9365	2.51
O2	0.7335	3.20
O3	0.67775	3.47
Os	0.8365	2.81

Après analyse des résultats, on peut en conclure que pour la version dynamique de la multiplication de vecteur par un scalaire, le plus optimisé serait *-O3* avec un taux d'accélération de 3.47. Cependant on remarque aussi *-O2* qui possède quand a lui un taux d'accélération de 3.20, ce qui reste relativement proche.

3.2 -fprofile-generate

Voici ce qu'en dit le manuel de gcc :

```
1 Enable options usually used for instrumenting application to produce
    profile useful for later recompilation with profile feedback based
    optimization. You must use -fprofile-generate both when compiling and
    when linking your program.
```

3.3 -fprofile-use

Voici ce qu'en dit le manuel de gcc :

```
1 Enable profile feedback-directed optimizations[...] Before you can
    use this option, you must first generate profiling information.
```

Voici un exemple, le temps d'exécution de matrix multiply en statique avec profile use :

Niveau optimisation	Moyenne temps globaux	Taux d'accélération
Aucun	**23.55**	**1**
-fprofile-use	23,70	0,99

TABLE 2 – Comparaison des temps d'execution entre `gcc` et `gcc -fprofile-use`.

On ne note donc pas de gain de performances, on perd meme un peu de temps.

4 Optimisations de code avec le compilateur d'Intel icc ou icx

4.1 Comparaison des niveaux d'optimisation

4.1.1 Multiplication de matrices

Version statique

Niveau d'optimisation	Moyenne temps globaux	Taux d'accélération
Aucun	**22.06**	**1**
O1	12.195	1.80
O2	13.3025	1.65
O3	15.465	1.42
Os	11.59	1.90

Après analyse des résultats, on peut en conclure que pour la version statique de multiplication de matrices, le plus optimisé serait *-Os* avec un taux d'accélération de 1.90.
Cependant on remarque aussi *-O1* qui possède quand a lui un taux d'accélération de 1.80, ce qui est relativement proche.

Version dynamique

Taille de bloc	Moyenne temps globaux	Taux d'accélération
Aucun	**26.9825**	**1**
O1	12.48	2.16
O2	14.385	1.87
O3	13.645	1.97
Os	12.7675	2.11

Après analyse des résultats, on peut en conclure que pour la version statique de multiplication de matrices, le plus optimisé serait *-O1* avec un taux d'accélération de 1.16.
Cependant on remarque aussi *-Os* qui possède quand a lui un taux d'accélération de 2.11, ce qui est très proche, on peut donc aussi s'en servir si on veut avoir un code plus court.

4.1.2 Addition de vecteurs

Version statique

Niveau optimisation	Moyenne temps globaux	Taux d'accélération
Aucun	**1.45**	**1**
O1	1.1	1.31
O2	0.96	1.51
O3	1.0175	1.42
Os	1.11	1.30

Après analyse des résultats, on peut en conclure que pour la version statique de l'addition de vecteurs, le plus optimisé serait *-O2* avec un taux d'accélération de 1.51.
Cependant on remarque aussi *-O3* qui possède quand a lui un taux d'accélération de 1.42, ce qui est relativement proche.

Version dynamique

Niveau optimisation	Moyenne temps globaux	Taux d'accélération
Aucun	**1.655**	**1**
O1	1.162	1.42
O2	1.0175	1.62
O3	1.015	1.63
Os	1.1975	1.38

Après analyse des résultats, on peut en conclure que pour la version statique de l'addition de vecteurs, le plus optimisé serait *-O3* avec un taux d'accélération de 1.63.
Cependant on remarque aussi *-O2* qui possède quand a lui un taux d'accélération de 1.62, ce qui est donc extrêmement proche, on ne peut donc pas faire de choix sur lequel est le plus optimisé, car leur différence de 0.1 peut etre due a des facteurs exterieurs au code.

4.1.3 Multiplication de vecteur par un scalaire

Version statique

Niveau optimisation	Moyenne temps globaux	Taux d'accélération
Aucun	**1.5875**	**1**
O1	0.905	1.75
O2	0.5425	2.92
O3	0.5375	2.95
Os	0.66	2.40

Après analyse des résultats, on peut en conclure que pour la version statique de la multiplication de vecteur par un scalaire, le plus optimisé serait *-O3* avec un taux d'accélération de 2.95.
Cependant on remarque aussi *-O2* qui possède quand a lui un taux d'accélération de 2.92, ce qui est relativement proche.

Version dynamique

Niveau optimisation	Moyenne temps globaux	Taux d'accélération
Aucun	**1.702**	**1**
O1	0.905	1.88
O2	0.54	3.15
O3	0.5425	3.13
Os	0.665	2.56

Après analyse des résultats, on peut en conclure que pour la version statique de la multiplication de vecteur par un scalaire, le plus optimisé serait *-O2* avec un taux d'accélération de 3.15.
Cependant on remarque aussi *-O3* qui possède quand a lui un taux d'accélération de 3.13, ce qui est relativement proche.

5 Librairie MKL d'intel

Pour compiler le programme, icx ne détectait pas les librairies MKL, j'ai donc dû entrer cette commande :

```
icx -I${MKLROOT}/include mat_mult_mkl.c \
    -L${MKLROOT}/lib/intel64 \
    -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lm -ldl \
    -o mat_mult_mkl
```

Niveau optimisation	Moyenne temps globaux	Taux d'accélération
O3	**15.465**	**1**
MKL	0.26	59.48

6 Compteurs matériels de performances (sur Linux/x86)

6.1 Resultats Matrix Multiply

Version	Nombre d'instructions	Nombre d'instructions memoires
statique	16,015,251,894	3,013,603
dynamique	64,041,264,521	8,002,013,971

Version	Defauts de cache L1	Defauts de cache L2	Branchements
statique	2 499 899 792	510 391 935	2 002 072 500
dynamique	9 942 510 853	545 097 850	8 005 082 741

6.2 Addition de vecteurs

Version	Nombre d'instructions	Nombre d'instructions memoires
statique	715,936,708	195,012,882
dynamique	1,885,939,617	390,013,242

Version	Defauts de cache L1	Defauts de cache L2	Branchements
statique	33 760 327	57 774 512	130 803 218
dynamique	34 977 315	60 405 651	325 803 765

6.3 Multiplication d'un vecteur par un scalaire

Version	Nombre d'instructions	Nombre d'instructions memoires
statique	1,063,166,651	250,013,369
dynamique	3,313,168,754	625,013,613

Version	Defauts de cache L1	Defauts de cache L2	Branchements
statique	31 466 405	55 315 321	188 030 546
dynamique	31 390 423	55 278 017	563 030 941

7 Conclusion

Au cours de ces expériences, nous avons exploré différents outils de compilation et les nombreuses options qu'ils proposent pour optimiser nos programmes en C. Cela nous a permis de comparer les performances et les optimisations offertes par `gcc` et `icx`, avec un avantage notable

pour `gcc` dans ce domaine. Par ailleurs, nous avons analysé les différences entre la programmation statique et dynamique, mettant en évidence que la programmation dynamique entraîne généralement des performances inférieures en raison de sa nature plus flexible mais plus coûteuse en ressources.
Enfin, l'étude des compteurs matériels de performances a enrichi notre compréhension du fonctionnement interne du processeur. Ces outils fournissent des données précises au fil de l'exécution d'un programme, telles que le nombre d'instructions exécutées, les accès mémoire, ou encore les défauts de cache. Ces informations sont essentielles pour identifier les problèmes et affiner davantage l'optimisation des programmes.