UVSQ
université PARIS-SACLAY

OBHPC Rapport TP2

YANG Yizhi

Nov 8, 2023

UVSQ
université PARIS-SACLAY

Introduction

Dans ce TP de es outils de base HPC, nous allons connaître les informations basiques sur les CPUs d'un système Linux, mesurer et analyser la performance d'un programme des différents algorithmes du produit matriciel (dgemm), du produit scalaire des vecteurs (dotrpod) et de la somme des éléments d'un vecteur (reduc), et comparer le débit moyen (en MiB/s) sous compilations (clang, gcc) et optimisations (-O0 — -Ofast) variées.

Informations sur les CPUs

Extrayons d'abord les informations de nos CPUs.* J'utilise le sytème Linux installé sur une clé USB, il possède 4 CPUs, en architecture x86_64, la fréquence de CPU est entre 0,5 et 3,2 GHz, celle actuelle est de 2,4 GHz. Après la commande « sudo cpupower frequency-set —governor performance », la fréquence est de 3,08 GHz.

Fixons la fréquence de CPU afin de stabiliser les mesures. La commande « cpupower frequency-set --governor performance » nous permet de maximiser la fréquence.

Ce tableau-ci représente les informations sur les trois caches du sytème Linux (Ubuntu) :

	Cache L1	Cache L2	Cache L3
cohency line size	64	64	64
id	0	0	0
level	1	2	3
number of sets	64	512	4096
physical line partition	1	1	1
shared cpu list	0,2	0,2	0-3
shared cpu map	5	5	f
size	32K	256K	4096K
type	Data	Unified	Unified
ways of associativity	8	8	16

*cf les fichiers cpuinfo et cpupowerinfo

Réalisation de mesures

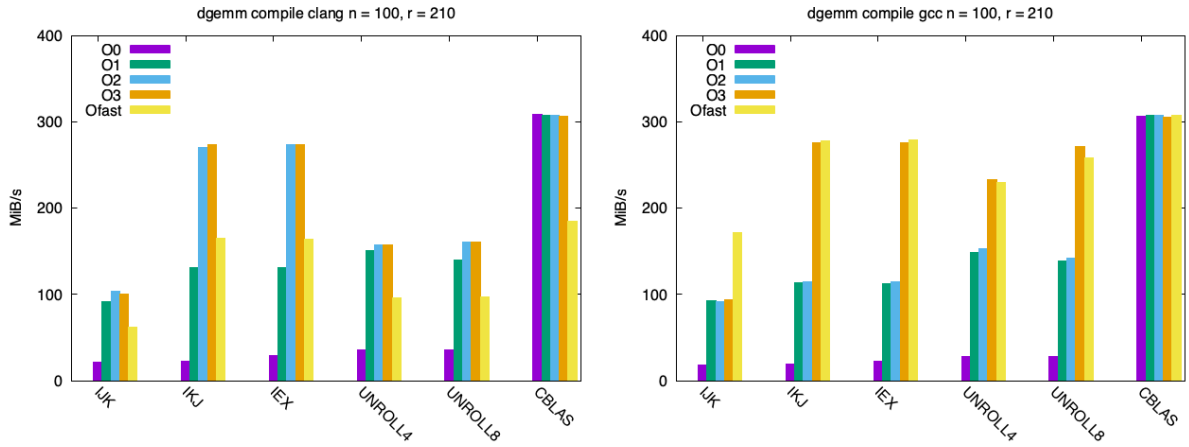
Rajoutons la méthode « unrolling by 8 » qui est similaire au déroulage 4 dans le fichier kernel.c. Lord de chaque exécution, appliquons la commande « taskset -c 1 » pour attacher le processus à l'ensemble des coeurs de CPU1. Cette étape nous permet de stabiliser les mesures de performance.

Vérifions bien les écart-types à la sortie de chaque lancement de mesure, ceux-ci doivent être inférieurs à 5% pour valider les résultats obtenus. (En général, les valeurs trop loins de la moyenne doivent être éliminées pendant les expériences répétitives, car elles sont très peu précises et introduisent des erreurs importantes.)

Lors d'une exécution de mesure, on arrête tous les autres processus pour ne pas influencer les résultats obtenus.

Interprétations dgemm

Ici présentent les histogrammes de compilateurs clang et gcc pour les algorithmes de produit matriciel de taille 100 * 100, avec 210 itérations.



Comparons d'abord la performance selon les algorithmes différents. En effet, la fonction dans la bibliothèque CBLAS reste toujours la plus vite et la plus stable (très peu variée) d'après toutes les versions de compilations. L'implémentation « naïve » (loop ijk) est la moins efficace dans tous les cas.

Comparées avec la méthode naïve ijk, les implémentations ikj et iex ont beaucoup augmenté la performance pour des raisons différentes. En échangeant l'ordre d'itération, la méthode ikj accède les éléments de la matrice qui se trouvent près l'un et l'autre dans la mémoire; en stockant une valeur constante, la méthode iex réduit le nombre d'accès à la même adresse de mémoire.

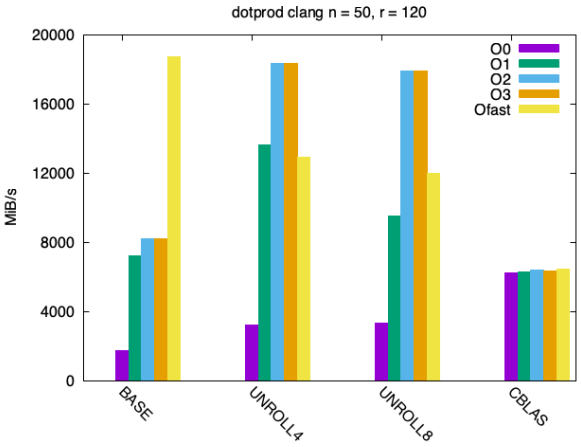
Les implémentations avec déroulage 4 et 8 nous permettent aussi d'améliorer la performance, cette méthode effectue à chaque itération plusieurs (ici 4 ou 8)

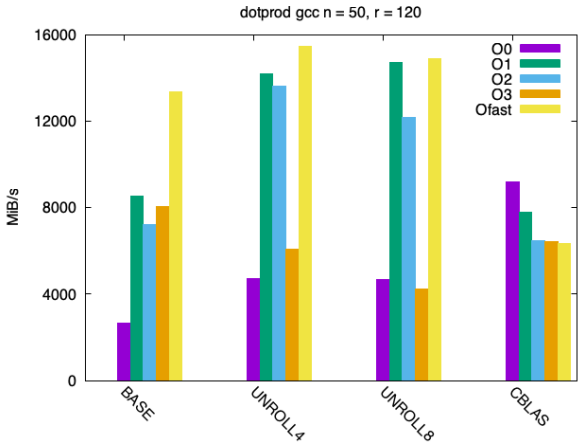
itérations en parallèle afin de diminuer le nombre de tests de condition, de calculs des indices etc.

Mais les méthodes ikj, iex, déroulage 4 et déroulage 8 ne sont pas toujours favorables pour la performance. L'efficacité des méthodes ikj et iex dépendra de l'architecture matérielle sous-jacente et de la taille des matrices en jeu. Le déroulage peut entraîner une augmentation de la taille du code et peut ne pas être avantageux pour toutes les boucles. Le gain de performances dépend de la nature des boucles, de l'architecture matérielle sous-jacente et des optimisations du compilateur.

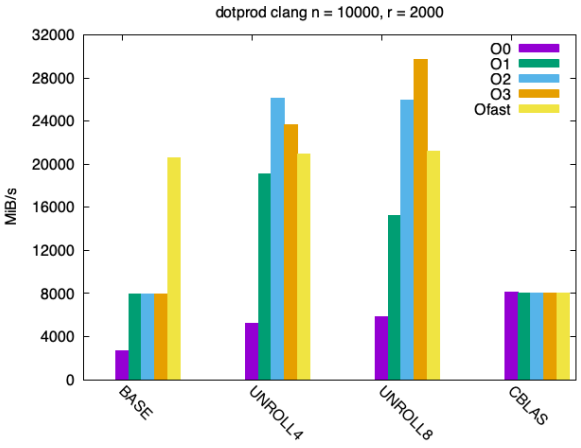
Interprétation dotprod

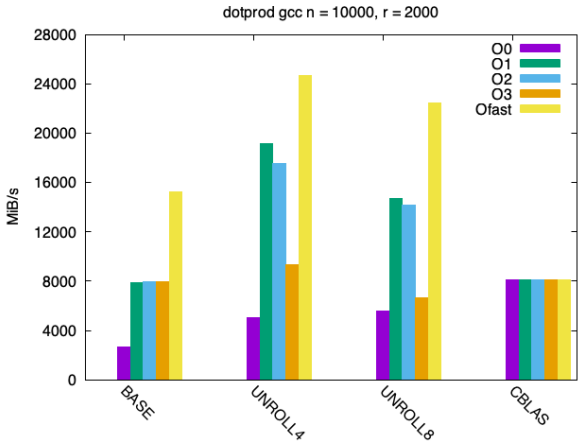
Ici présentent les histogrammes de compilateurs clang et gcc pour les algorithmes de produit scalaire des deux vecteurs de dimension 50, avec 120 itérations.



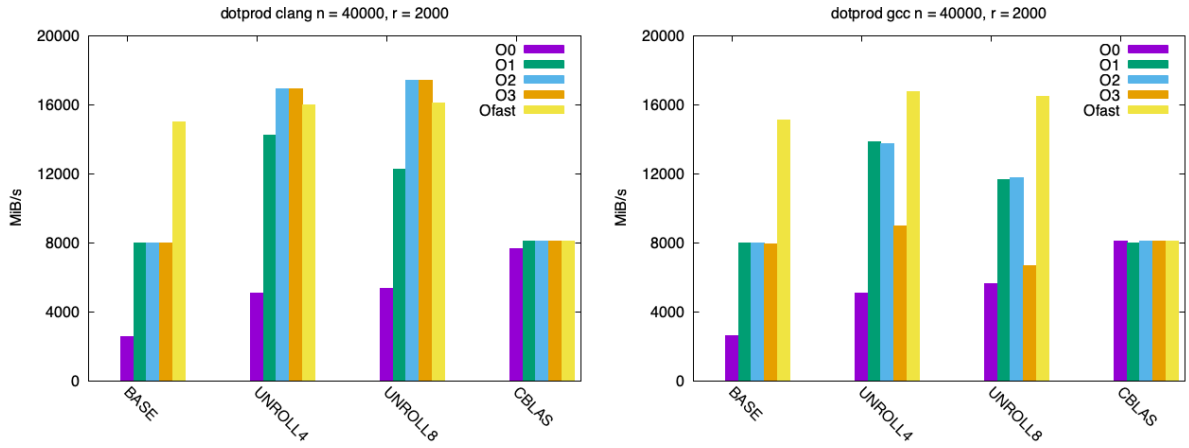


Ici présentent les histogrammes de compilateurs clang et gcc pour les algorithmes de produit scalaire des deux vecteurs de dimension 10000, avec 2000 itérations. (Cache L2)





Ici présentent les histogrammes de compilateurs clang et gcc pour les algorithmes de produit scalaire des deux vecteurs de dimension 40000, avec 2000 itérations. (Cache L3)



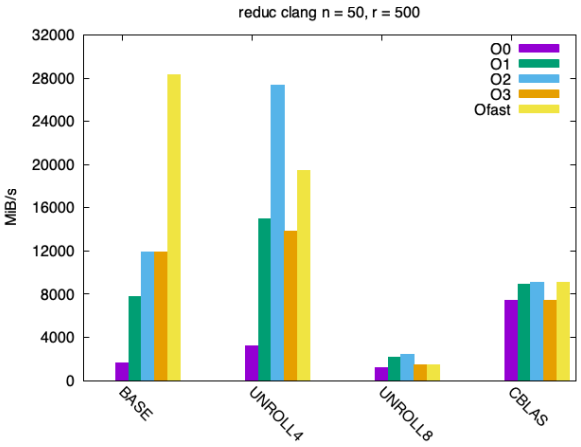
On constate une stabilité de la fonction CBLAS dans toutes les options d'optimisation (-O0 — -Ofast) quelque soit compilateur (clang ou gcc), car elle est déjà très bien optimisée, mais elle n'améliore la performance que sous l'option -O0. Les déroulages par 4 et par 8 possède à peu près la même allure dans les histogrammes, le déroulage par 8 n'est pas plus avantageux par rapport au celui par 4, car cela réduit les coûts de la vérification des conditions mais augmente la taille de boucle.

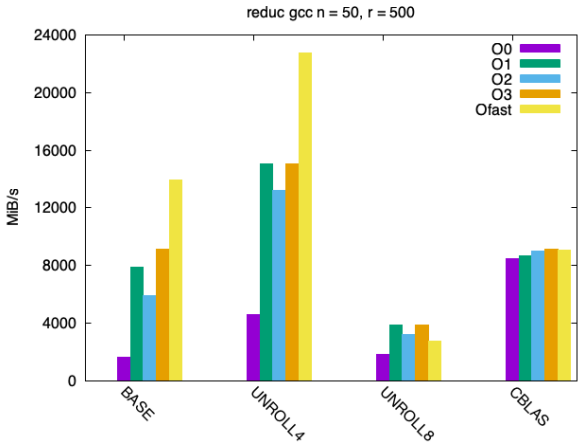
On observe aussi une meilleure efficacité de compilateur clang par rapport à celle de gcc. Clang a été conçu avec une architecture modulaire, ce qui signifie que le compilateur est divisé en composants indépendants qui peuvent être réutilisés. Cette modularité peut faciliter la mise en cache des résultats de certaines étapes de compilation, améliorant potentiellement la performance, en particulier pour des projets de petite à moyenne taille. Clang utilise un algorithme de parsing plus moderne et efficace appelé Recursive Descent with backtracking, ce qui peut contribuer à une analyse syntaxique plus rapide comparée à celle de gcc.

La meilleure performance est obtenue dans la cache L2 avec la compilation clang -O3, le déroulage par 8.

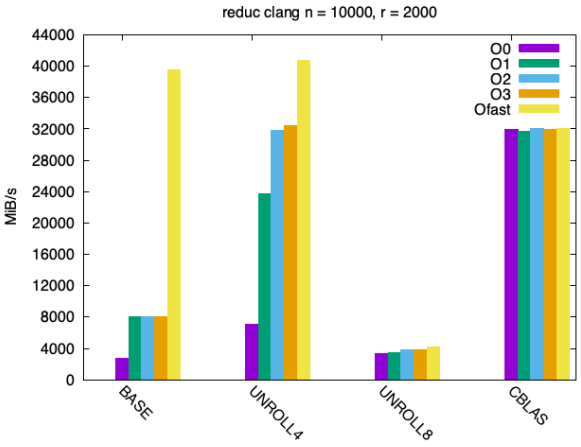
Interprétation reduc

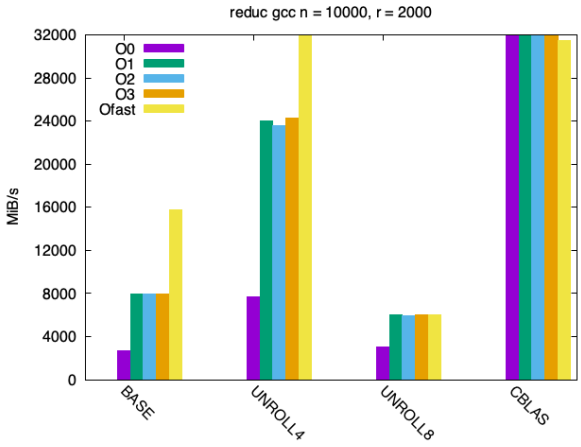
Ici présentent les histogrammes de compilateurs clang et gcc pour les algorithmes de somme des éléments d'un vecteur de dimension 50, avec 500 itérations.



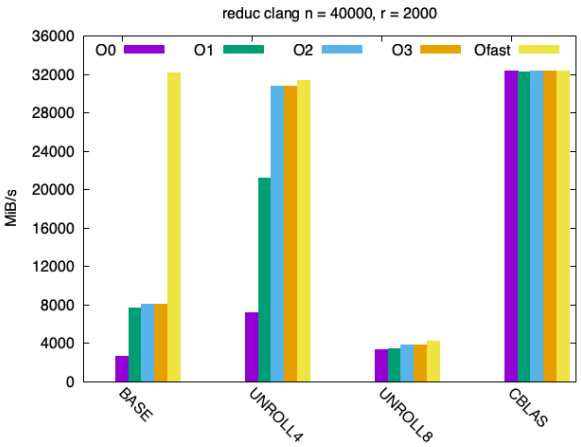


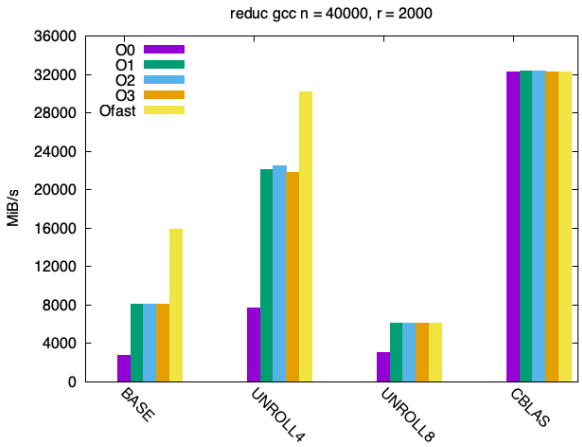
Ici présentent les histogrammes de compilateurs clang et gcc pour les algorithmes de somme des éléments d'un vecteur de dimension 10000, avec 2000 itérations. (Cache L2)





Ici présentent les histogrammes de compilateurs clang et gcc pour les algorithmes de somme des éléments d'un vecteur de dimension 40000, avec 2000 itérations. (Cache L3)





On constate au premier coup d'oeil que le déroulage par 8 est une très mauvaise optimisation d'algorithme car il n'est même pas plus efficace que la méthode « naïve » . En effet, si la boucle est déjà relativement petite, le déroulage peut augmenter la taille du code sans offrir suffisamment d'opportunités pour améliorer le parallélisme ou réduire les coûts de saut conditionnel. Le déroulage peut entraîner une saturation du pipeline du processeur, surtout si le processeur n'est pas capable d'exécuter suffisamment d'instructions en parallèle pour justifier un déroulage important.

La performance de la fonction CBLAS en caches L2 et L3 est stable, mais comparée avec les résultats obtenus en cache L1, elle est presque quatre fois plus puissante lorsqu'on augmente la taille du vecteur. C'est peut-être parce que les processeurs modernes sont capables d'effectuer plusieurs instructions en parallèle, notamment à l'aide de jeux d'instructions SIMD (Single Instruction, Multiple Data). Lorsque la taille du vecteur augmente, il devient possible d'exploiter davantage le parallélisme des instructions, ce qui peut améliorer les performances.

La meilleure performance est obtenue dans la cache L2 avec la compilation clang -Ofast, le déroulage par 4.

Conclusion

Les performances de compilation peuvent dépendre du contexte spécifique du code source, des options de compilation utilisées, et des versions spécifiques des compilateurs. Les expériences effectuées dans ce TP ne sont plus qu'une introduction de l'analyse de la performance, pour aller plus loin, il nous faut considérer plus de facteurs, comme les bruits du système, la bande passante, les hardwares etc; ainsi, des tests plus précis et mieux ciblés sont attendus.