Impact énergétique du compilateur
Compilateur Deca

Projet Génie Logiciel, groupe 15

January 27, 2023

Contents

1 Impact énergétique de la compilation

Impact énergétique de la compilation Un des objectifs principaux dans tous les programmes qu'on code est d'utiliser les structures de données et les algorithmes les plus optimales afin de réduire le temps d'exécution et donc réduire l'énergie utilisée. Lors de la création de ce compilateur, on a aussi suivi cette approche même si parfois le choix optimale peut prendre plus de temps dans la phase de dévelopement .

On a utilisé par exemple des tableaux de hachage dans la gestion des registres pour savoir si un registre est occupé ou non car cette structure de données était la plus optimale dans notre architecture. On a aussi utilisé un stack afin de stocker les registres à rendre après d'une opération(POP opération).

Afin de pouvoir comparer la vitesse de notre compilateur avec celle de java, on a créé un code deca et un code java qui sont syntaxiquement très proche et on a comparé le temps de compilation des deux fichiers.

Le graph dans la figure 1 compare le temps de compilation d'un code deca avec notre compilateur et un code java syntaxiquement très proche au code de deca. La barre blue considère un code avec 300 ligne de code/ rouge 500 ligne de code/ jaune 1000 ligne de code. Dans l'axe des y, on mesure le temps de compilation en ms.

Les lignes de code qu'on a placées dans les deux compilateurs sont les mêmes, par simple on veut dire que les lignes code représente un opération courte par exemple l'attribution d'une valeur à un registre ou une opération en très deux éléments. Ceux qui sont compliquées représente par exemple l'opération suivante:

```
u = (((((((5+4/3))*2)+6)*9)*7)+60)*(2*(4*(5*(6))));
```

Cette opération nécessite beaucoup de registre, ainsi il faut passer par les algorithmes qui associent les registre aux littéraux et il faut passer les structures de données utilisées. On peut remarquer à partir de

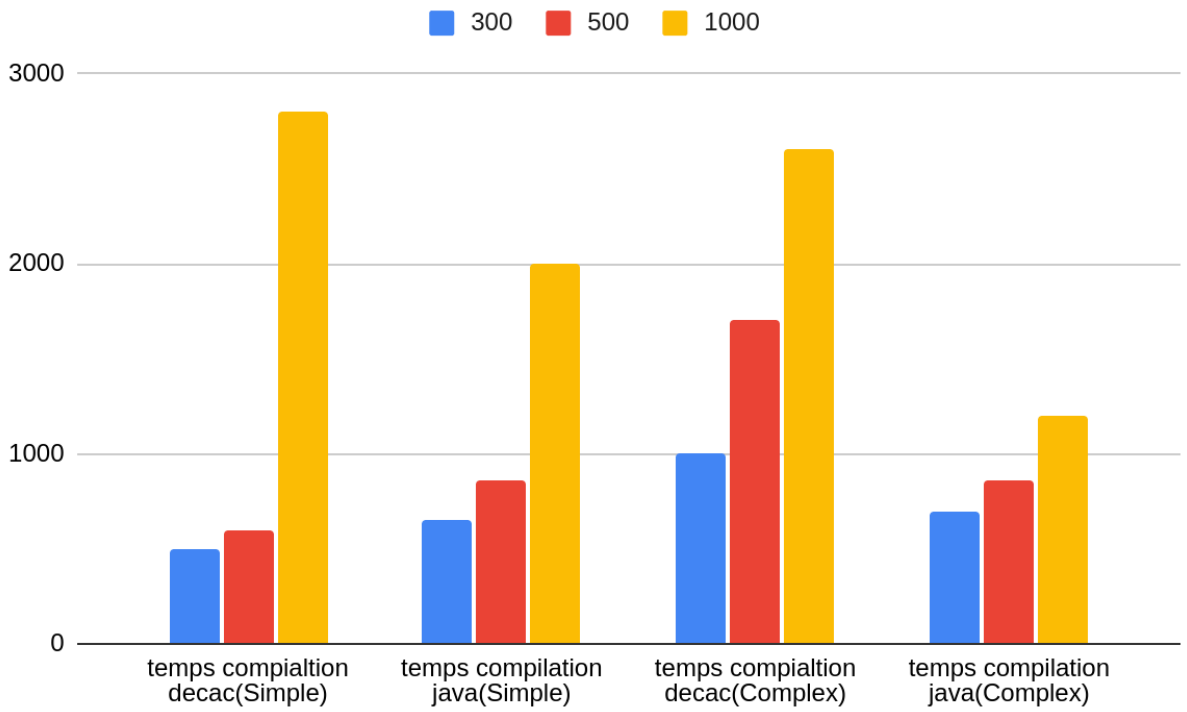

Figure 1: Graphe qui compare le temps d'exécution du compilateur de java et celui de deca pour un fichier de code java et deca similaire syntaxiquement. l'axe des ordonnées représente le temps de compilation et les bars represente des programmes de différentes lignes de code

graph que pour un code similaire, le compilateur de java est plus rapide et surtout dans le cas où on travaille avec une opération complexe, ainsi on peut conclure de cela que le compilateur de java contient des algorithmes très rapide pour associer au littéraux des registres. On peut conclure alors qu'il existe des choses à améliorer dans notre compilateur afin qu'il puisse fonctionner rapidement et donc consommer moins d'énergie.

2 Impact énergétique du code assembleur généré

Lors de la génération du code assembleur, un effort a été fait afin de réduire les instructions qu'on généré lors de la compilation. Il y avait des choses simples comme par exemple quand on fait des opérations et on a un identificateur à droite de l'opération binaire on utilise directement l'adresse que cet identificateur est stockée et on le met pas dans un registre. Le code suivant illustre cela, si on fait cette opération. On utilise directement l'adresse de y au lieu de la mettre dans un registre et exploité ce registre pour faire le calcul

```
{
 h = x*y;
}
```

Dans les opération booléenne aussi des optimisation ont été faites par exemple si on a l'opération booléenne suivante on va juste évaluer la partie gauche si c'est false on quitte directement au lieu de passer par toutes les instructions, dont il faut passer par lorsqu'on compile

```
{
```

```
        false && (opbool);
}
```

Cette optimisation simple qui réduit le nombre d'opérations à exécuter ont un impact sur le temps d'exécution et ainsi un effet sur l'énergie consommée.

Dans plusieurs opérations on a utilisé le registre R0 et R1 comme des variables intermédiaires pour faire un calcul spécifique par exemple dans le module, il faut vérifier que le quotient est non nul ainsi dans ce cas on utilise le registre R0. On a aussi utilisé ces registre dans la méthode de instance of.

L'utilisation de ces registres peut réduire le nombre d'opérations de registre considérablement car dans le cas où les registres sont occupés on a besoin de passer par push et pop et load au lieu de un load uniquement.

3 Impact énergétique de phase de validation

Dans le processus de validation, l'objectif est de vérifier que le compilateur fonctionne correctement et lance les bonnes erreurs et retourne les bons résultats.

Ainsi, pour bien tester le compilateur il faut au minimum avoir une bonne couverture des éléments à tester et s'assurer que le compilateur réagisse correctement quand il détecte un problème avec le code ou quand il donne une sortie dans la partie qui test la génération et l'exécution du code.

L'approche qu'on a pris pour minimiser les tests et tout tester en même temps c'est de pas tester la même chose plusieurs fois. Dans chaque partie de testing, on divise les fonctions et erreurs à tester afin qu'on n'a pas deux fichiers de test qui font la même chose.

L'appel de la méthode vérifie prend du temps, pour ceux qui utilise ubuntu directement sans passer par docker cette opération prend 4-5 minutes à la fin du projet, mais pour ceux qui utiliser docker le mémoire et cpu attribuée à ce container été très limité ainsi mvn verify prend 30 minutes pour s'exécuter qui n'est pas pratique du tout si on veut vérifier que le compilateur fonctionne correctement, ainsi la réduction de nombre de tests n'était pas uniquement pour optimiser de l'énergie mais pour réduire le temps de vérification.

3.1 impact énergétique dans votre extension tab

Dans l'extension tab il fallait créer des tableaux et des matrices, une chose importante à prendre en compte est d'écrire du code assembleur rapide qui permet d'accéder aux éléments du tableau efficacement. Afin d'optimiser cette opération, on génère du code assembleur qui fait les moins d'opération possible.

Par exemple, si on veut accéder à un élément d'une matrice, on prend des indices par l'utilisateur ensuite on détermine l'endroit dans la mémoire ou ces indices sont stockés.

La formule utilisée pour trouver cette position est de faire la multiplication de l'indice du nombre de la ligne donnée par l'utilisateur par le nombre de colonnes de la matrice et ensuite ajoutée l'indice de colonne données par l'utilisateur.

Comme les opérations de PUSH et de POP combinées consomment 6 cycles de processeur. On a choisi dans ce cas d'utiliser les registre R0 et R1 et faire l'opération de Load qui cout uniquement 2 cycle de processeur. Ainsi avec cette simple exploitation de registre R0 et R1, on a pu économiser 12 cycles de processeurs (2 PUSH et POP et 2 LOAD contre 2 LOAD)

Lors de l'initialisation d'une matrice plusieurs bibliothéque comme numpy choisi de mettre un zéro dans tous les éléments de la matrice or que cette opération sera complètement inutile si cette matrice est n'est pas exploitée par l'utilisateur. Dans notre implémentation on n'initialise pas la matrice à zéro mais on donne à l'utilisateur les méthodes nécessaires afin de faire cette opération.

Dans le cas où l'utilisateur définit une matrice très large mais n'exploite uniquement que quelques éléments de cette matrice, notre implémentation fera une réduction non négligeable du nombre d'instructions à réaliser.

4 Impact énergétique de la bibliothèque de calcul matricielle

Il faut développer des méthodes pour faire des opérations avec les matrices, notre objectif initial était d'avoir des algorithmes optimaux avec la plus faible complexité temporelle afin de réduire le nombre d'opérations par exemple la multiplication entre les matrices.

Initialement, notre objectif était de développer l'algorithme de strassen de la multiplication matricielle afin d'avoir de réduire le temps d'exécution mais cela était très compliqué à faire avec ce compilateur car on ne pouvait libérer la mémoire et lors du développement on a pris compte que l'algorithme de multiplication naïf sera plus efficace due à l'utilisation gigantesque de mémoire qu'une opération de multiplication prendra.

Pour évaluer la rapidité de notre programme dans la multiplication matricielle, on a écrit un code en python qui définit deux matrices et met des valeurs dedans et fait la multiplication entre les matrices définies avec le même algorithme.

Le code qu'on exécute par ima passe un processeur virtuel, ainsi c'est peu difficile de comparer le temps d'exécution d'un programme qui s'exécute directement sur notre cpu. Le résultat qu'on a eu avec cette expérience ne permet de rien conclure car le code python ne passe pas par un processeur virtuel. Ainsi, on se trouve dans la position ou le code python est 20 plus rapide que celui généré par le compilateur comme on peut le voir dans la figure 2.

Afin de pouvoir comparer les deux programmes, il fallait que j'utilise une bibliothèque de python “hwcounter” qui calcule le nombre de cycles de processeur qu'on a utilisés pour exécuter le code python.

Pour avoir le nombre de cycle de ima, on a utilisé l'option -d qui nous permet d'accéder à cette valeur.

Le nombre de cycles nécessaire pour exécuter le code python est 100 fois plus grands que celui du code ima ainsi dans le graph de la figure 3 j'ai uniquement mis les cycles associé au program ima.

Pour calculer l'énergie utilisées pour cette opération de multiplication on a fait le produit du nombre de cycle par la fréquence du processeur(Motorola 68000) d'après la page wikipedia https://en.wikipedia.org/wiki/Motorola_68000 et on a aussi pris la puissance à partie de cette page. La formule qu'on a utilisée est E = P*T, P : puissance du processeur et T : temps associé au cycles qui est calculé par le produit entre le nombre de cycles et P est la puissance nécessaire pour le processeur.

La puissance prise est 2W et la fréquence 4 Mhz. La figure 4 représente l'énergie utilisée par le processeur pour faire le produit matricielle pour un nombre d'éléments dans la matrice.

Par une simple régression linéaire, on a pu conclure que la multiplication de deux matrices avec 100 millions d'éléments donc matrice 10000x10000 coûtera 849998 joule qui est égal à 0.23kwh qui est assez conséquent pour une opération de ce type d'où l'importance d'algorithme plus rapide pour faire ce calcul.

5 Conclusion

Le compilateur qu'on a réalisé n'est pas terrible en terme de consommation d'énergie mais on a pris conscience à partir de cette étude de notre empreinte énergétique et aussi des prochaines étapes à prendre afin d'optimiser notre code dans la génération du code: -amélioration des algorithmes de compilation par exemple la gestion des registres. -amélioration des algorithmes dans la bibliothèque de calcul matricielle.

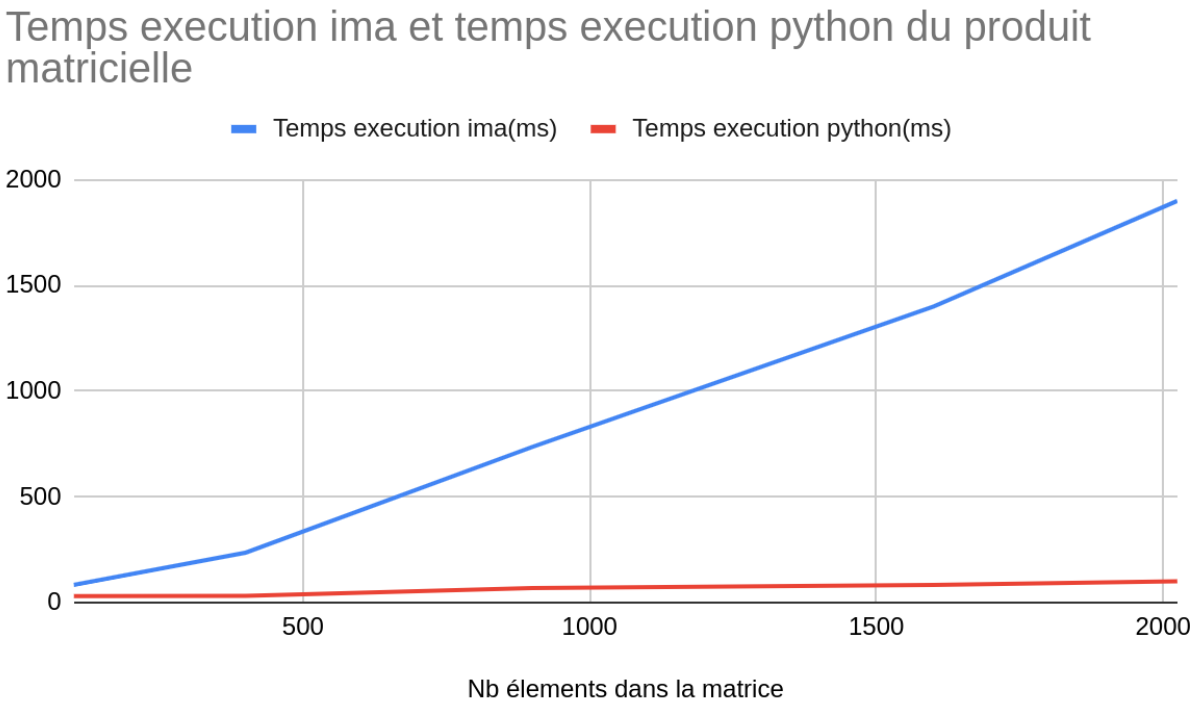

Figure 2: Ce graph calcule le temps d'exécution du produit matriciel pour un program deca et d'un program python

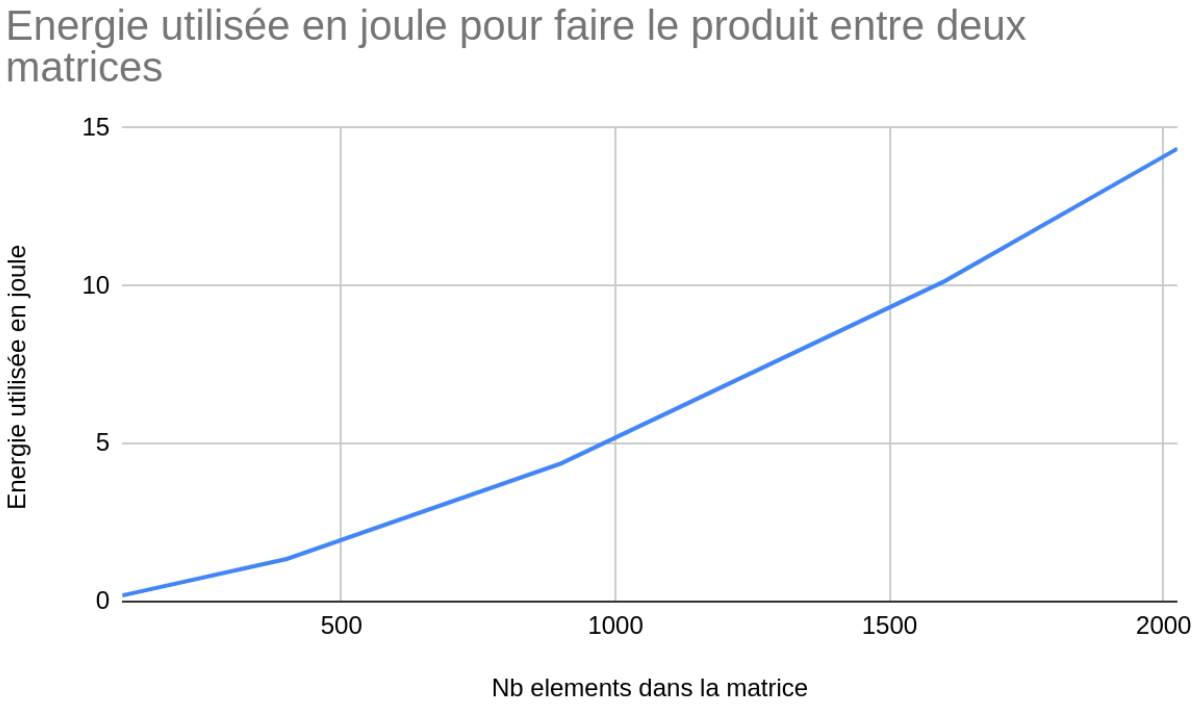

Figure 3: Le graph montre l'énergie utilisée pour le produit matricielle

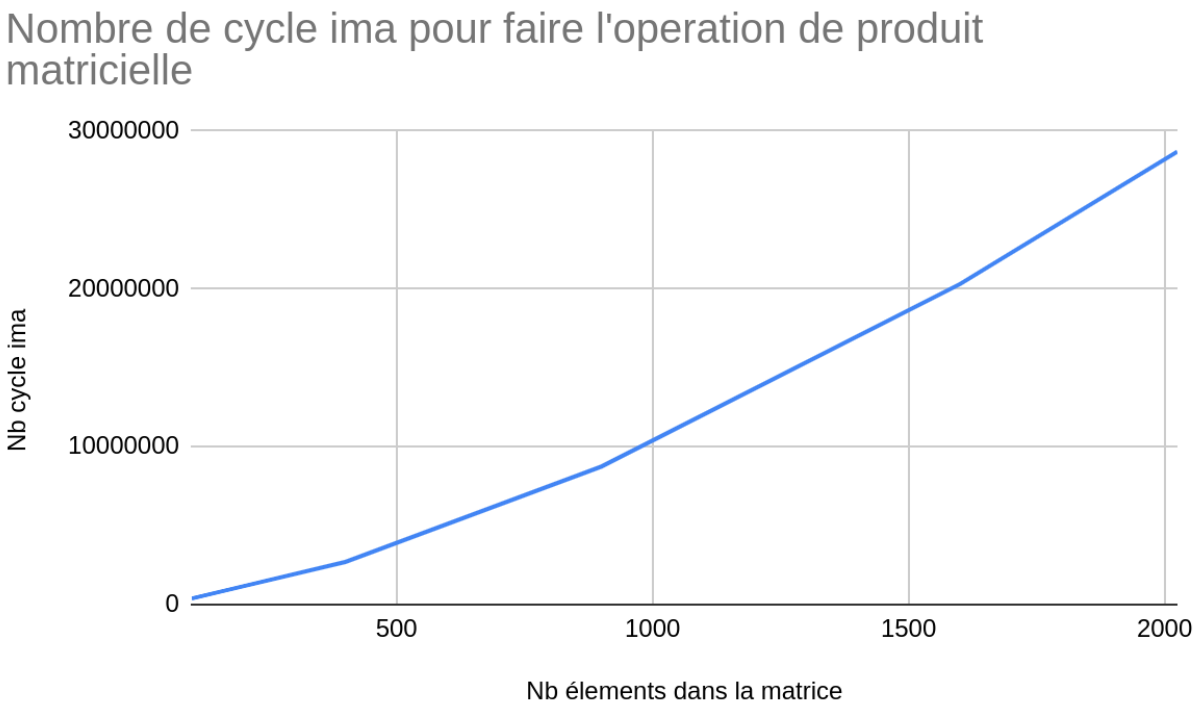

Figure 4: Le graph suivant calcule le nombre de cycle cpu nécessaires pour faire la multiplication de deux matrices