Documentation de la performance énergétique
Projet GL Groupe 44

COLIN-SOHY Mathéo

DIMITRIOU Tristan

EL HIT Oumayma

GAUX Antoine

HARRAUD Paul-Louis

Table des matières

1 Introduction

Ce document a pour but de montrer que l'ensemble de notre travail, de l'efficacité des algorithmes utilisés, au code généré en passant par les méthodes de travail et de vérification du projet, a été fait en accord avec une démarche d'efficience tant dans l'aspect des performances que dans l'aspect de la consommation énergétique du compilateur et des fichiers qu'il crée.

2 Compilation des sources

Durant l'ensemble du projet, la compilation des sources du projet a été réalisée en utilisant l'outil *maven*, qui permet à l'instar d'un *Makefile*, de ne recompiler que les fichiers qui ont été modifiés, ou ne sont plus présents compilés.
Dès lors, durant toute la durée du projet, les appels à `mvn compile` ont permis de limiter la consommation énergétique lors de la compilation des sources. Les appels à `mvn clean`, nécessitant donc une nouvelle compilation de l'ensemble des sources ont également été le plus limités possible, se réduisant aux plus grandes vérifications de l'ensemble du projet, tels qu'avant un partage du travail d'un membre avec le reste de l'équipe afin de confirmer le bon fonctionnement global de l'ajout actuel.

3 Génération de code

Le principe de génération de code présenté comme exemple dans le sujet ne produit pas un code optimisé, et certains de nos choix ont encore alourdis le code généré.

Listing 1 – chainedList

```
class Cell{
    int content;
    Cell next;
    Cell endOfList;

    void init(int a, Cell b, Cell c){
        this.content = a;
        this.next = b;
        this.endOfList = c;
    }

    void printList(){
        println(content);
        if (this.next.equals(this.endOfList)){
        } else {
            next.printList();
        }
    }
}

{
    int a1 = 1;
    int a2 = 2;
    int a3 = 3;
    int a4 = 9;
    Cell c1 = new Cell();
    Cell c2 = new Cell();
    Cell c3 = new Cell();
    Cell endOfList = new Cell();
    c3.init(a3, endOfList, endOfList);
    c2.init(a2, c3, endOfList);
    c1.init(a1, c2, endOfList);
    c1.printList();
}
```

Sur cet exemple, en repassant sur le code généré pour raccourcir le code (enlever des `LOAD` inutiles entre autres), on économise 15 instructions et on réduit d'environ 6% le temps d'exécution, ce qui n'est pas énorme. Nos appels de méthodes et initialisations sont donc relativement optimisés.

Si on prend le code `syracuse42.deca` fourni et qu'on enlève toutes les instructions de manipulation mémoire inutiles, on divise presque par 2 le nombre d'instruction et on réduit d'environ 28% le temps d'exécution. On voit donc que le code généré pour les opérations arithmétiques peut être vraiment optimisé, sans pour autant rentrer dans le cadre d'une extension.

Lors du code de la partie génération de code, nous avons estimé que le plus important était la justesse plutôt que l'optimisation, ce qui explique sur certains points d'importants gâchis d'énergie.

4 Procédé de validation

Une politique d'économie énergétique a aussi été mise en œuvre lors du procédé de validation du compilateur.

En effet, si un grand nombre de tests a été généré afin de valider l'intégrité de l'ensemble du compilateur, ceux-ci ne sont en réalité que très rarement utilisés en totalité.
Dans les faits, lors du processus de développement, chacun ne lançait, pour vérifier la validité de son travail actuel, que les tests relatifs à la partie concernée par ses ajouts, puisque le reste du compilateur était donc inchangé sur sa branche. De plus, les scripts lancés lors des vérifications individuelles s'exécutaient sur l'ensemble des fichiers de test, quelque soit le nombre d'erreurs afin de connaître directement l'ensemble des problèmes à corriger, plutôt que les découvrir et les corriger un à un, ce qui permet de réduire le nombre de compilations de l'ensemble du projet, ainsi que le nombre de lancements des programmes de tests.
Ce n'est que lors de l'intégration et de la fusion de toutes les modifications du compilateur ensemble que la totalité des tests étaient effectués afin de confirmer que les modifications étaient compatibles et répondaient bien aux spécifications du langage deca.

Durée des tests en fonction de la partie à valider
— Tests syntaxiques : 51s
— Tests contextuels : 1 min 31s
— Tests de génération de code : 53s

— Totalité des tests : 2 min 56s

Par ces analyses de temps d'exécution du compilateur, et donc en parallèle de sa consommation énergétique, nous pouvons affirmer que l'approche décrite ci-dessus permet une économie énergétique de la validation d'un facteur au moins 2 pour chaque exécution, sans pour autant rendre la base de tests moins complète sur la partie en question.

De la même façon, les appels à l'outil *jacoco* ont été limités, et toujours effectué par un seul membre du groupe, qui partageait ensuite l'analyse aux autres membres afin d'éviter les redites qui auraient pu composer une part importante de la consommation énergétique du projet.

5 Extension

Lors de la conception de l'extension, nous nous sommes efforcé d'atteindre une bonne précision.
L'efficacité de notre algorithme, que l'on étudie a posteriori n'est pas idéale.
Par exemple, nous utilisons une boucle principale dans laquelle on effectue des comparaisons en fonction d'un entier qui sert de sélecteur.
Ce qui veut dire qu'à chaque fois que l'on calcule le sinus ou le cosinus d'un angle, des comparaisons sont effectuées 32 fois pour chaque autre fonction (atan et asin)
De plus nous n'avons pas utilisé d'instruction FMA, ce qui par ailleurs aurait pu augmenter la précision. On a donc plusieurs multiplications et additions et ce a chaque itération.
Le calcul de chaque fonction trigonométrique prend environ 127000 cycles IMA. Outre la complexité de l'algorithme mis en place, ce résultat est cohérent avec le fait que nos instructions se répètent, parfois inutilement. Et ça vient simplement de l'algorithme et de la manière dont il est implémenté.

6 Palmarès du Projet GL 2022

Finalement, nous obtenons les scores suivants sur les fichiers de performances fournis par les professeurs, afin d'établir un palmarès entre les différentes équipes :

NB : Ces tests ont été réalisés avec l'option -n du compilateur

— Syracuse : 97 instructions et un temps d'exécution de 2046 cycles (S)
— Ln2 : 238 instructions et un temps d'exécution de 24426 cycles (L0)
— Ln2 (avec fonction) : 269 instructions et un temps d'exécution de 26361 cycles (L1)

Au final, nous cummulons un score total de : (L0+L1+10*S) **71247**