Analyse énergétique

Samy Ahjaou Hamza Bouihi Omar Guessous
Grégoire Sename Mamadou Thiongane

Groupe 9 – Équipe 42

26 janvier 2024

Table des matières

1 Introduction

Le projet de développement du compilateur Deca revêt une importance cruciale non seulement dans la conception d'un logiciel efficace mais également dans la minimisation de son empreinte énergétique. Dans cette perspective, cette documentation met en lumière deux dimensions fondamentales où les choix opérés tout au long du projet influent directement sur la consommation d'énergie des logiciels générés par le compilateur, le compilateur lui-même, et le processus de validation associé.

La première dimension examinée est l'efficience du code produit par le compilateur. Étant donné que les logiciels générés par le compilateur consommeront de l'énergie lors de leur exécution, il est impératif d'évaluer le coût énergétique des assemblages d'instructions que réalise le compilateur. Dans cette optique, il devient essentiel de privilégier des assemblages moins gourmands en énergie. Cette considération revêt une importance particulière dans le cadre de l'extension du projet, où les contraintes sont moins contraignantes, offrant ainsi une marge de manœuvre plus significative.

La deuxième dimension à prendre en compte est l'efficience du procédé de fabrication. La compilation et l'exécution des tests sont des étapes consommatrices d'énergie. Ainsi, il est nécessaire de réfléchir à l'optimisation des processus et des scripts de validation tout en maintenant la qualité du compilateur comme objectif principal.

Cette documentation s'attache à explorer les différentes facettes de l'efficience énergétique du projet, en mettant en lumière les moyens employés pour évaluer la consommation énergétique tout au long du processus de développement. Nous discuterons également de l'impact des choix de compilation de Deca vers ima, en soulignant les considérations énergétiques spécifiques à notre projet. Une attention particulière sera accordée à la discussion du processus de validation, avec une stratégie visant à minimiser son impact énergétique sans compromettre l'effort de validation et la qualité du compilateur. Enfin, nous aborderons la manière dont l'impact énergétique est pris en compte dans l'extension du projet.

Au-delà des directives fournies, cette documentation s'engage à explorer toute analyse pertinente qui pourrait enrichir la compréhension de l'efficience énergétique dans le cadre du projet du compilateur Deca.

2 Efficience du code produit

Notre première approche pour évaluer l'efficacité du projet a consisté à utiliser la commande `ima -s [fichier compilé]` afin d'obtenir des informations sur le temps d'exécution et le nombre d'instructions générées pour un fichier `.ass`. Cette démarche a été particulièrement appliquée aux fichiers `syracuse.deca` et `ln2.deca`, qui nous ont été fournis pour évaluer les performances du compilateur. Les résultats obtenus sont présentés ci-dessous.

Nom du test	**Nombre d'instructions**	**Temps d'execution**
Syracuse	43	1404
LN2	126	18280

Table 1 – Résultats des programmes de performance

Nous voyons donc que le code produit est relativement performant, que nous avons pu comparer

avec le vieux compte de Professeurs comme cela :

$$\text{Différence} = \frac{|\text{GL42} - \text{VieuxCompDeProf}|}{\text{GL42}}$$

Nous obtenons donc les écarts suivants :

Nom du test	**Écart temps d'execution**
Syracuse	4,6%
LN2	16,9%

TABLE 2 – Comparaison des programmes de performance avec VieuxCompDeProf

Nous obtenons un temps très proche pour le programme *syracuse.deca* mais plus éloigné pour *ln2.deca*. L'expression calculée dans *ln2.deca* étant plus compliquée, nous supposons que c'est ici que notre compilateur est moins efficace.

3 Efficience du procédé de fabrication

3.1 Gestion de la validation

Nous avons élaboré une batterie complète de 258 tests couvrant divers aspects fonctionnels de notre compilateur. L'objectif est d'évaluer la performance globale du compilateur en termes de conformité aux spécifications et d'identifier d'éventuels problèmes. Cependant, une attention particulière a été portée à éviter les tests doublons, garantissant ainsi l'efficacité de notre processus de validation.

Dans le but d'optimiser l'utilisation des ressources, nous avons adopté une approche visant à créer des fichiers de tests de taille minimale tout en conservant une couverture complète des fonctionnalités. Cette stratégie contribue à réduire le temps d'exécution des tests et, par conséquent, l'impact énergétique global de notre processus de validation.

Notre choix de mettre en place un script global pour vérifier l'ensemble des fonctionnalités, combiné à des scripts individuels pour chaque composant du compilateur (lexeur, parseur, context, codegen et decac), offre une flexibilité essentielle. Cette modularité nous permet d'exécuter des tests spécifiques en fonction des besoins, optimisant ainsi l'efficacité énergétique du processus de validation.

Afin de démontrer cela, nous avons mesuré les temps d'exécution de tous ces scripts de test.

Sur un MacBook Air M2 équipé de 16 Go de mémoire, les performances des scripts shell ont été évaluées comme suit :

Il est notable que le test le plus chronophage est largement test-decac.sh, suivi de test-synt.sh. Les parties du projet qui ont exigé un développement plus complexe et, par conséquent, d'avantage de tests sont test-context.sh et test-gencode.sh. L'exécution séparée de ces tests permet d'économiser plus de 75% du temps système et utilisateur et ainsi d'être bien plus efficace énergétiquement.

Les temps totaux renforcent cette observation, avec seulement 7 secondes pour test-gencode.sh au lieu des 1'21 du temps total.

Pour les scripts python, les performances sont les suivantes :

Script	Temps utilisateur (s)	Temps système (s)	Temps total (m's)
test-all.sh	111,99	13,41	1'21,32
test-lex.sh	1,93	0,26	1,543
test-synt.sh	29,50	3,39	20,433
test-context.sh	24,37	2,79	16,807
test-gencode.sh	8,99	1,20	6,920
test-decac.sh	47,22	5,80	35,668

TABLE 3 – Performances des Scripts Shell

Script	Temps utilisateur (s)	Temps système (s)	Temps total (m's)
test-all.sh	111,99	13,41	1'21,32
test-lex.sh	1,93	0,26	1,543
test-synt.sh	29,50	3,39	20,433
test-context.sh	24,37	2,79	16,807
test-gencode.sh	8,99	1,20	6,920
test-decac.sh	47,22	5,80	35,668

TABLE 4 – Performances des Scripts Shell

Bien que les scripts python soient plus rapides, ils étaient légèrement moins fiables que les scripts shell. Par conséquent, nous avons privilégié l'utilisation des scripts shell à la fin du projet.

3.2 Gestion du répertoire Git

Nous avons décidé d'introduire une branche de développement, nommée `dev`, positionnée entre la branche `master` et les branches `feature` dédiées au développement des fonctionnalités et des tests spécifiques. Cette branche `dev` agit comme un intermédiaire pour l'envoi de nos mises à jour vers la branche `master`. L'objectif est d'éviter de créer une `merge request` distincte pour chaque branche `feature`, en optant plutôt pour une approche globale. Cette démarche significativement réduit le temps nécessaire pour l'exécution du `pipeline`.

Nous pouvons voir ici une partie de l'historique Git, avec en vert la branche `dev` intermédiaire à `master` en rouge.

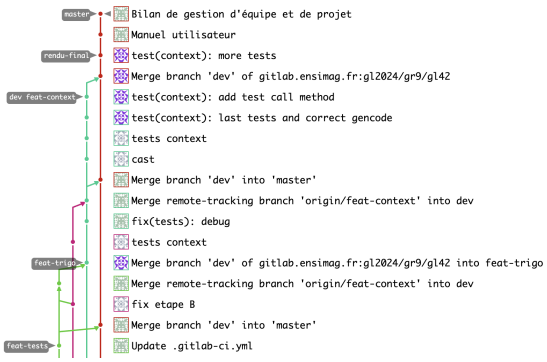

FIGURE 1 – Historique Git

Au total, nous avons déclenché le `pipeline` à 31 reprises, dont 10 occurrences pour résoudre les ajustements initiaux et corriger les anomalies dans le fichier `yml`. Ces actions ont été réalisées

pour 260 commits au total. Le nombre maximal de branches `feature` actives simultanément a été limité à 5 (codegen, syntax, context, trigo, tests). Nous estimons ainsi avoir réduit de trois fois le nombre de jobs dans le `pipeline`, ce qui a non seulement permis d'économiser considérablement de temps et d'énergie sur les machines de l'ENSIMAG gérées par le runner, mais aussi de libérer le runner partagé pour d'autres étudiants de l'ENSIMAG. Ces ajustements ont eu un impact positif. Le temps d'exécution moyen du `pipeline` étant de 2 minutes 30 cela a réduit significativement le temps total d'utilisation des `runners` , contribuant ainsi à une utilisation plus efficace des ressources partagées.

De surcroît, l'utilisation d'un fichier .gitignore a été mise en œuvre afin d'éviter d'archiver sur le serveur GitLab des fichiers superflus tels que les fichiers .ass, .class, etc. Au total, plus de 1000 fichiers ont été évités d'être stockés inutilement. Lors de l'évaluation des données, le constat a été le suivant : 24 Mo ont évité d'être stockés inutilement. Bien que cela puisse sembler insignifiant à première vue, cette économie de taille, à l'échelle de l'ensemble de l'ENSIMAG, pourrait représenter une économie potentielle de près de 1,5 Go.

La mise en place d'une stratégie proactive pour réduire l'impact énergétique de notre processus de validation reflète notre engagement envers une approche durable du développement logiciel. Des optimisations ont été intégrées dans la gestion des ressources, la maîtrise de la consommation de mémoire et l'exploitation judicieuse de l'exécution parallèle de certains tests. Ces efforts visent à minimiser la consommation d'énergie tout en préservant l'intégrité de l'effort de validation et la qualité du compilateur.

4 Impact énergétique de notre extension

Notre extension, axée sur les fonctions trigonométriques (Trigo), repose sur la conception d'algorithmes à la fois efficaces et fiables. L'objectif principal est d'optimiser le calcul des fonctions trigonométriques, en veillant à minimiser le temps d'exécution et l'utilisation de la mémoire. En effet, dans le contexte de l'utilisation de notre bibliothèque, il est impératif que le calcul d'une fonction trigonométrique, tel que le sinus, s'effectue de manière rapide, sans compromettre la fiabilité du résultat.

Pour atteindre cet objectif, nous avons opté pour une approche basée sur l'estimation des fonctions trigonométriques à l'aide de séries entières et de l'algorithme Cordic. L'approximation des séries entières jusqu'à la précision maximale, en utilisant l'ulp (unit in the last place), s'est révélée essentielle pour rendre nos fonctions trigonométriques efficaces. Cette méthode garantit une convergence rapide vers une valeur précise. De plus, l'exploitation du FMA (fused multiply-add) pour les opérations d'addition et de multiplication a permis de réduire le nombre d'instructions en langage assembleur, contribuant ainsi à minimiser les erreurs et à optimiser la taille du code.

À l'heure actuelle, nous sommes convaincus de la performance de notre bibliothèque Trigo. Elle se distingue par sa rapidité, sa précision et son utilisation minimale d'instructions, répondant ainsi aux exigences d'efficacité cruciales pour tout logiciel l'implémentant.

5 Conclusion

En conclusion, ce document a mis en lumière l'importance cruciale de considérer l'efficience énergétique tout au long du développement du compilateur Deca. En intégrant des réflexions sur deux dimensions clés – l'efficience du code produit et l'efficience du procédé de fabrication

– notre équipe a cherché à optimiser non seulement les performances du compilateur mais également à minimiser son impact sur la consommation d'énergie.

L'évaluation de l'efficience énergétique du projet a été abordée de manière approfondie, en mettant en évidence les méthodes utilisées pour mesurer la consommation énergétique à différentes étapes du processus de développement. Les résultats obtenus ont démontré une performance relative du code généré, comparée de manière critique avec des références antérieures.

Les choix de compilation de Deca vers ima ont été examinés sous l'angle des considérations énergétiques spécifiques à notre projet. Une comparaison rigoureuse avec des compilateurs antérieurs a souligné les écarts, mettant en lumière les domaines où des améliorations pourraient être envisagées pour optimiser davantage l'efficacité énergétique.

La gestion du processus de validation a été un point focal, avec une stratégie détaillée visant à minimiser l'impact énergétique sans sacrifier la qualité du compilateur. L'utilisation de scripts de test modulaires, la création de tests de taille minimale, et l'optimisation des ressources ont contribué à une validation efficace et écoénergétique.

L'extension du projet, axée sur les fonctions trigonométriques, a été abordée avec un accent particulier sur la rapidité, la précision et l'utilisation minimale d'instructions. Les choix algorithmiques, tels que l'estimation par séries entières et l'utilisation de l'algorithme Cordic, ont été justifiés en fonction de leur contribution à l'efficience énergétique de la bibliothèque Trigo.

Enfin, la gestion du répertoire Git, avec l'introduction d'une branche de développement et l'utilisation judicieuse de .gitignore, a démontré notre engagement envers une approche responsable et efficace du développement logiciel.

En somme, ce projet a été guidé par une conscience aiguë de l'importance de l'efficience énergétique dans la conception de logiciels. Les efforts déployés pour mesurer, analyser et optimiser ont abouti à un compilateur Deca et à une extension Trigo qui non seulement répondent aux exigences fonctionnelles mais le font de manière économe en énergie. Cela témoigne de notre engagement envers une approche durable du développement logiciel, reflétant les préoccupations contemporaines liées à la consommation énergétique.