Outillage pour l'étude de l'impact de l'ordre des passes de LLVM

Julian BRUYAT

Juin 2019

Résumé LLVM est un compilateur de programmes C et C++. L'infrastructure du compilateur transforme les programmes vers une représentation intermédiaire, sur laquelle il va travailler en appliquant des *passes* ("phases") d'analyse et d'optimisation pour améliorer les performances. On s'intéresse ici à l'ordonnancement de ces passes, et à l'impact de cet ordonnancement sur les performances dans le cas général. Dans ce projet, nous proposons des méthodes et des outils pour évaluer l'impact des combinaisons de passes de compilation sur les performances des programmes compilés. Nous appliquons notre méthode à quelques cas concrets sur l'ensemble de la base de tests proposée par LLVM.

Mot-clefs Compilation, Clang, LLVM, Optimisation, Passes

1 Introduction

Ce POM [1] a été proposé et encadré par Matthieu Moy et Laure Gonnord (CASH, LIP/ENS Lyon, Université Lyon 1) ainsi que Sébastien Mosser (Université du Québec).

1.1 L'équipe CASH

CASH [2] est une équipe de recherche commune à l'Inria Grenoble et au Laboratoire de l'Informatique du Parallélisme (LIP) localisée à l'École Normale Supérieure de Lyon (ENS Lyon).

L'équipe CASH travaille sur des thématiques autour de la compilation et l'ordonnancement de programmes, l'extraction de programmes flots de données parallèles depuis des programmes séquentiels, ou encore l'analyse statique de programmes. L'objectif est de fournir aux développeurs des solutions permettant d'écrire au mieux des programmes tirant parti des différentes plate-formes d'exécution, notamment les machines de calcul parallèle : multi et many-coeurs, GPU (cartes graphiques), FPGA [3].

1.2 LLVM

Un compilateur est un logiciel dont le but est de transformer un programme écrit par un humain (en langage C par exemple) en du code exécutable par une machine.

LLVM[6] est un compilateur développé par l'Université de l'Illinois dont la première version est sortie en novembre 2003. Son objectif est d'être un compilateur modulaire avec des points d'entrée clairs, en particulier pour ajouter de nouvelles passes et pouvoir les tester.

1.3 Sujet

Nous nous intéressons ici à comment sont organisées les optimisations faites par le compilateur sur un programme. En effet, le programme compilé ne correspond pas exactement à ce que le développeur a écrit, en général il a été transformé et restructuré.

1. Projet d'Orientation en Master
2. Compilation, and Analysis, Software and Hardware
3. Field Programmable Gate Arrays, des circuits programmables

```
int n = 0;
for (int i = 0 ; i != 10 ; i++)
    n += (i + 1);
```

```
int n = 55;
```

FIGURE 1 – Deux programmes équivalents

Les deux codes présentés à la figure 1 sont équivalents en terme de sémantique, mais en terme de temps d'exécution et de mémoire utilisée, le premier sera plus long et utilisera une case mémoire pour la variable i.

Pour optimiser, deux types de passes sont appliquées : des passes d'analyse et des passes d'optimisation. L'objectif de ce POM est de fournir des outils pour étudier comment interagissent les passes entre elles, et quel est le résultat sur des programmes réels (sont-ils plus rapides, consomment-ils moins de mémoire ?).

On propose ici deux méthodes afin d'étudier l'ordre des passes : une première méthode d'étude de programmes individuels, permettant de faire des mesures variées mais ne passant pas à l'échelle. La seconde modifie les règles de génération de la *test-suite* de LLVM afin d'appliquer l'ordonnancement voulu de passes. Cette méthode a l'avantage de pouvoir s'appliquer sur un grand nombre de programmes, facilitant les études de cas générales.

2 LLVM et les compilateurs

2.1 Compilation d'un programme

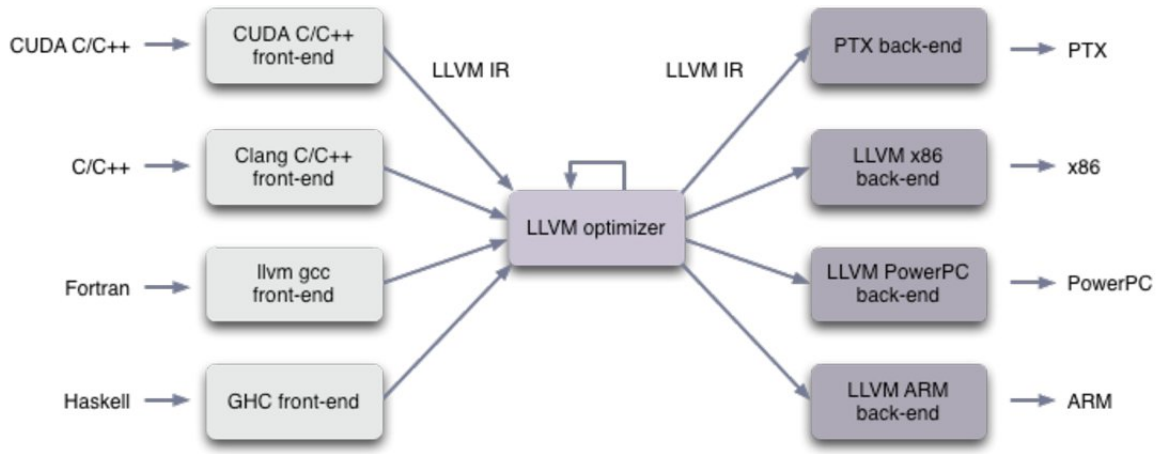

FIGURE 2 – Fonctionnement de CLANG et de LLVM

La Figure 2, extraite de [1] montre que la lecture du code source et la compilation vers du code machine sont des parties bien distinctes au sein du compilateur LLVM. La représentation intermédiaire (IR pour *Intermediate Representation*) sert de langage commun entre les différents *front-end* (partie de gauche) et les différents *back-end* (à droite, avec les étapes de génération de code et l'application des passes dépendant de la machine cible).

Dans Clang-LLVM (le *front-end* pour C/C++, mais aussi le nom du compilateur entier pour C/C++), CLANG lit le code source, vérifie les symboles, construit un arbre syntaxique et converti le code dans la représentation intermédiaire de LLVM. Celle-ci est ensuite, après de nombreuses passes d'analyses et d'optimisation, transformée en du langage machine par LLVM (par exemple en du code exécutable par un processeur Intel).

L'étape qui nous intéresse ici est notée "LLVM Optimizer". Celle-ci transforme une IR en une autre IR afin que le programme généré à la sortie soit plus performant.

2.2 Ordonnancement des passes

"LLVM Optimizer" est le gestionnaire de passes. Son rôle est de recevoir la liste des passes à appliquer et de les appliquer séquentiellement.

Comme dans GCC, CLANG, propose des combinaisons de passes standard sous la forme des options `-O0`, `-O1`, `O2`, `-O3`. Ces options ont pour but d'être simples à utiliser. L'outil CLANG propose uniquement de désactiver certaines passes lors de l'exécution de ces combinaisons, ce qui rend son usage peu extensible.

Pour réaliser une combinaison quelconque de passes, il faut donc utiliser une combinaison d'outils de LLVM de la façon suivante : tout d'abord, on utilise uniquement le *front-end* CLANG pour générer l'IR, ensuite l'outil OPT permet de choisir explicitement les passes à appliquer, dans l'ordre voulu. Ensuite nous générons le code à l'aide du *back-end* adapté à notre machine.

2.3 La représentation intermédiaire de LLVM

La représentation intermédiaire LLVM est une variante de SSA (*Static Single Assignment*, c'est-à-dire que chaque variable utilisée est affectée une seule fois. Les passes d'analyse et d'optimisation à l'intérieur de LLVM OPTIMIZER sont lancées par un outil spécialisé, OPT, qui permet à un utilisateur de lancer les passes dans l'ordre de son choix.

Le but est de transformer un programme afin de le rendre meilleur (le plus souvent en un code plus rapide, mais d'autres métriques que nous évoquons plus tard peuvent être utilisées.).

Pour cela, on a deux types de passes : les passes d'analyse qui lisent le code et l'annotent, et les passes d'optimisation qui en utilisant les annotations, transforment le code afin de le rendre meilleur.

```
int main() {
  int k = 777;
  if (k == 777) {
    k = 0;
  }
  return k;
}
```

```
; [#uses=0]
; Function Attrs: noinline norecurse
;   nounwind readnone uwtable
define dso_local i32 @main()
  local_unnamed_addr #0 {
entry:
  ret i32 0
}
```

```
; [#uses=0]
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %k = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 777, i32* %k, align 4
  %0 = load i32, i32* %k, align 4
  %cmp = icmp eq i32 %0, 777
  br i1 %cmp, label %if.then, label %if.end

if.then:
  store i32 0, i32* %k, align 4
  br label %if.end

if.end:
  %1 = load i32, i32* %k, align 4
  ret i32 %1
}
```

FIGURE 3 – Un programmé écrit en C, converti en IR puis optimisé

La figure 3 présente un exemple de code écrit en C en haut à gauche. Le code est transformée vers l'IR présentée à droite. Sans entrer dans les détails techniques, on peut voir que l'IR ressemble à du code assembleur dans le sens où on a des allocations explicites de la mémoire, les instructions "load" et "store" et les structures de contrôle sont remplacé par des sauts. Les différentes passes de

`O3` sont appliquées, et seul le code en bas à gauche reste, qui est plus court, et renvoie un entier sur 32 bits valant 0.

2.4 Outils déjà disponibles

Le compilateur LLVM est une base de code très grande (chiffre quel chiffre ? les 900 tests de la tsuite dont 300 pour benchmark + les tests unitaires ? le nombre de lignes de code de llvm ?), qui est livré avec une base de tests pour valider son développement ainsi que les performances des programmes générés. L'infrastructure fournit aussi des outils pour que les développeurs puissent tester leurs nouvelles passes (relativement) facilement.

2.4.1 LLVM-LIT

LLVM-LIT est l'outil principal permettant de tester et de *benchmarker*[4] LLVM.

Il se repose sur une exploration récursive d'un dossier. LLVM-LIT recherche des fichiers contenant des directives comme `RUN` (exécution d'un programme) ou `CHECK` (par exemple pour tester si la sortie d'un programme correspond à ce qui est attendu).

Son rôle est principalement de coordonner l'utilisation d'autres outils et d'agréger leurs résultats. Parmi les métriques récupérées par LLVM-LIT, nous pouvons citer :

— Le temps d'exécution est mesuré avec l'outil TIMEIT inclus avec LLVM ou avec PERF[5]. selon le choix de l'utilisateur. PERF donne des mesures plus précises mais requiert des privilèges super utilisateur.
— Le temps de compilation avec TIMEIT
— La taille des exécutables générés.
— En revanche, bien que LLVM-LIT utilise VALGRIND, on peut pas utiliser l'outil MASSIF pour récupérer l'usage de mémoire. VALGRIND étant utilisé uniquement en mode "memcheck", mode dédié à la vérification de la bonne gestion de la mémoire (l'usage le plus commun de ce mode est de savoir si la mémoire allouée explicitement par l'utilisateur est libérée).

Ces mesures peuvent être écrites dans un fichier au format JSON, ce qui facilite les traitements automatiques des résultats.

2.4.2 La test suite

La communauté LLVM propose une base de test nommée "LLVM Test Suite". En plus de test dédiés aux tests unitaires et de non régression, elle contient plus de 300 programmes dédiés au *benchmarking*.

Elle s'utilise en deux phases :

— La phase de compilation utilisant CMAKE. Elle se repose sur une structure récursive de fichiers CMake, indiquant où trouver les tests du dossier courant et si il faut exploiter les sous-dossiers. La génération avec cmake génère des fichiers contenant les règles permettant de compiler la test-suite.
— La phase d'exécution des tests avec LLVM-LIT. Chaque programme compilé par la test-suite génère un fichier .test tel que celui-de la figure 4 avec la manière d'exécuter le test et comment de vérifier sa sortie.

```
RUN: /path/to/built/tsuite/SingleSource/Benchmarks/BenchmarkGame/recursive
VERIFY: /path/to/built/tsuite/tools/fpcmp %o \
/path/to/built/tsuite/SingleSource/Benchmarks/BenchmarkGame/recursive.reference_output
```

FIGURE 4 – Un exemple de fichier .test pour LLVM-LIT

4. Mesurer les performances, généralement en temps
5. `https://perf.wiki.kernel.org/index.php/Main_Page`

L'utilisation principale de la test-suite est de tester si les programmes compilent, et si ils ont la sortie attendue. Elle permet également de *benchmarker* les programmes en utilisant les différentes options de CLANG combinées à d'autres configurations.

2.4.3 Travaux précédents

Notre démarche s'approche dans un sens de la compilation itérative [5], c'est-à-dire le fait de compiler un programme, mesurer ses performances, et recompiler avec une autre séquence d'optimisations jusqu'à avoir le programme qu'on estime le plus optimisé possible.

Dans une démarche de compilation source à source (on compile des fichiers .c en d'autres fichiers .c) utilisant PIPS[4], [3] propose de générer des configurations avec un algorithme génétique, et teste les performances en compilant avec gcc et icc.

Dans notre cas, les améliorations de la chaîne d'optimisation ne visent pas à améliorer un programme spécifique mais à découvrir une chaîne meilleure que la chaîne actuelle pour le plus de programmes possibles. Pour cela, on utilise un échantillon que l'on espère représentatif des programmes, qui sert de base pour trouver une meilleure chaîne qui sera ensuite utilisée pour d'autres programmes.

La littérature parle de *Phase Ordering* pour ce problème.

3 Nouvelles solutions de *benchmarking*

Nous cherchons à étudier l'impact des passes sur les performances d'une multitude de programmes afin de trouver expérimentalement un meilleur ordre que celui actuellement implémenté dans LLVM. Pour traiter ce problème, deux obstacles se posent à nous.

Nous sommes tout d'abord confronté au fait que CLANG ne permet pas de choisir explicitement l'ordre de passes que l'on souhaite. En revanche, des outils sont proposés pour décomposer la compilation (OPT, LLC, LLVM-DIS ...). Pour notre problème nous avons besoin de faire un appel à OPT pour modifier la représentation intermédiaire que doit générer CLANG puis transformer cette IR en un exécutable. Or aucun outil existant n'implémente ce procédé automatiquement.

Une fois l'architecture en place, nous devons étudier comment évaluer un programme et trouver des métriques à la fois intéressantes et implémentables. Le temps d'exécution est le premier facteur évident, en particulier le temps d'exécution minimum. En effet, sur des programmes séquentiels déterministe, celui-ci est fixe sur une machine donnée. D'autres métriques peuvent également être utilisées comme l'usage mémoire (en particulier l'étude de la pile, sachant qu'il est peu probable de réussir à optimiser les allocations de mémoires dans le tas, donc explicitement demandés), le poids de l'exécutable général, le nombre de fois où une passe agit ...

Les deux solutions proposées dans ce document se reposent sur deux axes indépendants. Le premier est l'élaboration d'un script indépendant qui se repose uniquement sur les outils de base permettant de décomposer CLANG. Le second se repose sur la modification de la base de tests existante (à savoir la "test-suite" de LLVM).

3.1 Mesures sur des programmes simples

La première méthode proposée est l'utilisation d'un script qui automatise la compilation et les mesures sur un programme écrit en C dans un unique fichier.

En entrée, ce programme prend le fichier voulu ainsi que les options de compilation (que ce soit une option comme `-O3` ou un ordre de passes.)

Le fichier est ensuite compilé (soit directement avec CLANG, soit en utilisant OPT comme intermédiaire) et les mesures sont faites.

Comme nous maîtrisons l'ensemble de la chaîne de compilation et l'exécution, nous pouvons par exemple :

— Mesurer l'usage mémoire avec VALGRIND.

— Mesurer la taille de l'exécutable compilé.

— Utiliser l'outil de notre choix pour la mesure de temps. Cela est intéressant par exemple pour appliquer la méthode du *wrapping* de la fonction main tel que présenté dans la section 3.1.1.

3.1.1 Mesure du temps d'exécution

Dans le code de gauche de la figure 5, la boucle de fonction $main$ peut être optimisée pour directement affecter 55 à s. Cette optimisation permet de passer d'une complexité en $\theta(n)$ à une complexité en $\theta(1)$. La majorité du temps d'exécution du programme est alors le chargement en mémoire et le lancement du programme, et non l'exécution du code. La plupart des outils de mesure de temps mesurent le temps total, et non le temps d'exécution effective du code, qui dans ce cas sera composé majoritairement d'un facteur dont nous n'avons que peu d'influence.

```
int main() {
    s = 0;
    n = 10;
    for (int i = 0 ; i != n ; i++) {
        s += (i + 1);
    }
    printf("La somme des nombres ");
    printf("de 1 à %d est %d\n",
           n, s);
    return 0;
}
```

```
int actual_main() {
    // Code identique au main précédent
}

int main() {
    if (actual_main()) return 1;
// Force une première mise en cache

    for (int i = 0 ; i != NOMBRE_DE_MESURES)
        lancer_chrono();
        if(actual_main()) return 1,
        arreter_chrono();
    }
    output_min_des_temps_chronometres()
    return 0;
}
```

FIGURE 5 – Principe du *wrapping*

Pour mesurer de manière plus précise le temps d'exécution d'un programme rapide, nous devons mesurer uniquement le temps d'exécution de sa fonction main. A droite de la figure 5, nous renommons la fonction "main". Nous appelons celle-ci dans la nouvelle fonction main après avoir lancé un chronomètre.

Cette méthode est néanmoins sensible aux effets de cache, et ne mesure pas efficacement les programmes utilisant par exemple des variables globales. De plus, elle ne fonctionne pas avec les programmes dont la fonction main termine avec un appel à exit(0) au lieu de return 0;. De plus, le programme d'enrobage doit idéalement prévoir tous les cas (programmes utilisant des arguments, programmes écrits en C ou en C++ ou encore programmes pouvant être composés plusieurs fichiers).

3.1.2 Passage à l'échelle

A ce stade, on souhaite automatiser le lancement des mesures sur un grand nombre de programmes.

Deux options sont possibles pour étendre la méthode proposée précédemment :

— Constituer notre propre base de tests. Cette approche ne se basant pas sur l'existant (la base conséquente de tests déjà présents), elle a été rapidement écartée.

— Intégrer la TEST-SUITE à notre outil. Cette second option est néanmoins compliquée à implémenter car elle implique de réinventer CMAKE.

Cette première méthode est donc efficace sur un programme cible déterminé, mais ne répond pas à notre objectif de pouvoir passer à l'échelle. Nous proposons donc de réutiliser la base de tests présente dans la *test-suite*.

3.2 Mesures sur la test suite

Utiliser la *test-suite* telle quelle ne permet pas de répondre à nos besoins. Lorsque les programmes qui la compose sont compilés, ils le sont en utilisant l'exécutable CLANG. Or nous avons vu que CLANG ne permettait pas de choisir l'ordre des passes. La test-suite a été conçue pour valider le développement du compilateur et en mesurer les performances.

Nous avons besoin de pouvoir choisir les passes et compiler la test suite un grand nombre de fois avec des options différentes. Nous cherchons donc à adapter de façon simple les outils autour de la *test-suite*.

3.2.1 Pistes envisagées

Avant de parler de la solution mise en place, nous allons évoquer rapidement les pistes qui ont été écartées.

— Exploiter l'option "export compile commands" de CMAKE. Cette option permet de lister dans un fichier json toutes les unités de compilation qui ont été générées. L'avantage théorique de cette méthode est qu'elle permettrait de concilier la test-suite avec notre premier script. Cette option est néanmoins inexploitable car elle est prévue pour les IDE [6], et donc les règles de compilation générées ne contiennent que les premières étapes de compilation vers des fichiers objets (extension .o). Or nous avons besoin des étapes de compilation amenant jusqu'à un exécutable, et en particulier des différentes options utilisés qui ne sont pas présentes.

— Modifier le code de CLANG pour intégrer directement l'ordre des passes que l'on veut étudier. Cette solution a l'avantage de permettre des mesures précises (par exemple, dans la section 4.1.2, on observe des différentes de temps d'exécution entre un programme compilé avec "opt -O3" et "opt" avec les passes de O3). Mais elle impose de modifier le code puis de recompiler CLANG, ce qui pose des problèmes à la fois en terme de saisie (il faut trouver un moyen pratique de modifier l'ordre des passes dans le code source) et de temps (elle impose de compiler CLANG à chaque fois). Elle ne se prête donc pas à une démarche d'exploration.

— Plutôt que de modifier directement l'ordre des passes de Clang, on pourrait intégrer une nouvelle option qui lirait l'ordre des passes à exécuter plutôt que de se reposer sur un ordre de passes prédéfini comme `O3`. A la manière de OPT, CLANG serait alors capable de remplir son *Pass Manager* avec les passes passées en argument par l'utilisateur.
Mais cette méthode, qui a été trouvée après l'implémentation de la réécriture de règles 3.2.2, a été estimée trop coûteuse en temps d'implémentation. Bien que CLANG soit *Open Source*, la complexité du code empêche d'ajouter facilement une option à des programmeurs non expérimentés.

— On peut imaginer créer un script que l'on utilisera à la place de CLANG. Ce script aurait pour rôle de recevoir les différents arguments qui aurait dû être envoyé à CLANG, et de créer l'enchaînement d'appels souhaité, à savoir un premier appel à CLANG pour obtenir une IR, un appel à OPT pour appliquer les passes souhaitées puis un appel à CLANG pour transformer l'IR en l'exécutable souhaité. Le problème de cette méthode est qu'elle requiert de coder comment répartir les différents arguments reçus dans les programmes appelés.

3.2.2 Réécriture de règles

La solution que nous avons implémenté est une solution se reposant sur la modification des fichiers de construction de la test-suite. L'idée est de modifier les règles de génération des test pour remplacer l'appel de CLANG par une succession d'appels visant à produire le comportement que l'on souhaite (c'est-à-dire compiler en choisissant l'ordre des passes grâce à OPT)

6. *Integrated Development Environment* ou Environnement de Développement

Fichiers générés par CMake La compilation de la test-suite requiert deux étapes : une première étape où CMAKE génère le fichier MAKEFILE ou les fichiers de configurations NINJA puis une seconde étape où l'on *build* la test-suite. Dans notre cas, nous allons nous intéresser aux fichiers de configurations ninja. Deux fichiers sont générés par CMAKE : un fichier "build.ninja" contenant les parties variantes pour chaque programme de la test-suite et un fichier "rules.ninja" contenant des règles génériques.

```
rule CXX_COMPILER__filter_test
depfile = $DEP_FILE
deps = gcc
command = /path/to/built/testsuite/tools/timeit --summary $out.time
          /path/to/clang/clang++ $DEFINES $INCLUDES $FLAGS -MD -MT
          $out -MF $DEP_FILE -o $out -c $in
description = Building CXX object $out
```

FIGURE 6 – Règle de base telle que générée par CMAKE dans "rules.ninja"

Principe La figure 6 présente une règle ninja telle que générée par défaut pour le test nommé *filter*. La ligne de commande exécutée n'utilise aucune connaissance sur le programme cible en dehors du langage utilisé. On peut donc reconnaître le motif et le remplacer pour effectuer les modifications que l'on souhaite, en particulier décomposer l'appel de CLANG en plusieurs appels. Cela nous permet de maîtriser totalement la chaîne de compilation comme sur la figure 7. En outre, l'utilisation de OPT nous donne la possibilité d'en extraire des informations issues de l'application des passes (par exemple [7] fait l'étude de l'analyse de pointeurs, et notamment l'impact des passes `basicaa`, et une nouvelle proposition `sraa` sur des passes ultérieures).

```
command = /path/to/built/testsuite/tools/timeit --summary $out.time
          /path/to/clang/clang++  $DEFINES $INCLUDES $FLAGS -MD -MT
          $out -MF $DEP_FILE -s -o $out.bc -emit-llvm -Xclang
          -disable-O0-optnone -c $in
       && /path/to/built/testsuite/tools/timeit --summary $outopt.time
          /path/to/clang/opt {la liste des opt qu'on veut} $out.bc -o $out.bc
       && /path/to/built/testsuite/tools/timeit --summary $outclang2.time
          /path/to/clang/clang++ -c $out.bc -o $out $FLAGS
```

FIGURE 7 – Réécriture de la ligne commands

Faiblesses L'implémentation proposée ne fonctionne pas sur toutes les machines car il dépend de la bonne reconnaissance de la règle d'origine. Une meilleure implémentation serait de modifier directement les fichiers CMAKE de base de la test-suite pour qu'il génère directement les bonnes règles.

De plus, comme nous utilisons l'infrastructure de la *test-suite* nous sommes limités par les outils proposés par LLVM-LIT. Cette architecture ne permet donc pas par exemple de mesurer l'usage en mémoire des programmes.

Toutefois, malgré les faiblesses de cette méthode, sa simplicité à être mise en place et à être comprise font qu'elle a été conservée pour le reste du projet.

4 Résultats

L'architecture ayant été mise en place, nous l'exécutons sur quelques cas concrets.

4.1 Cohérence sur des cas connus

Nous commençons par tester si notre architecture répond au besoin initial, à savoir mesurer en temps l'exécution des programmes de la "test-suite". Pour cela, on regarde d'abord si les résultats obtenus sur des séquences connues (`O3`, `O0` ...) sont cohérent avec ce qui est attendu, puis nous testons la possibilité de choisir un ordre de passes. En effet, si `O0`, `O3` et un ordre de passe donné quelconque donnent tous les mêmes résultats expérimentaux, alors ceux-ci sont faux.

4.1.1 Séquences implémentées par LLVM

Nous commençons par comparer les *speedup*[7] de `O0`, `O2` et `O3`, par rapport à `O1`. On exécute la "test-suite" sur ses programmes dans les catégories *SingleSource* et *MultiSource* (programmes écrits en C ou en C++, en un seul fichier pour le premier, en plusieurs pour le second). Après avoir effectué plusieurs mesures du temps d'exécutions des programmes, on prend le temps minimum, calcule le *speed-up* puis on dessine la moyenne géométrique de ces valeurs pour limiter l'impact des *speed-up* éloignés de la moyenne.

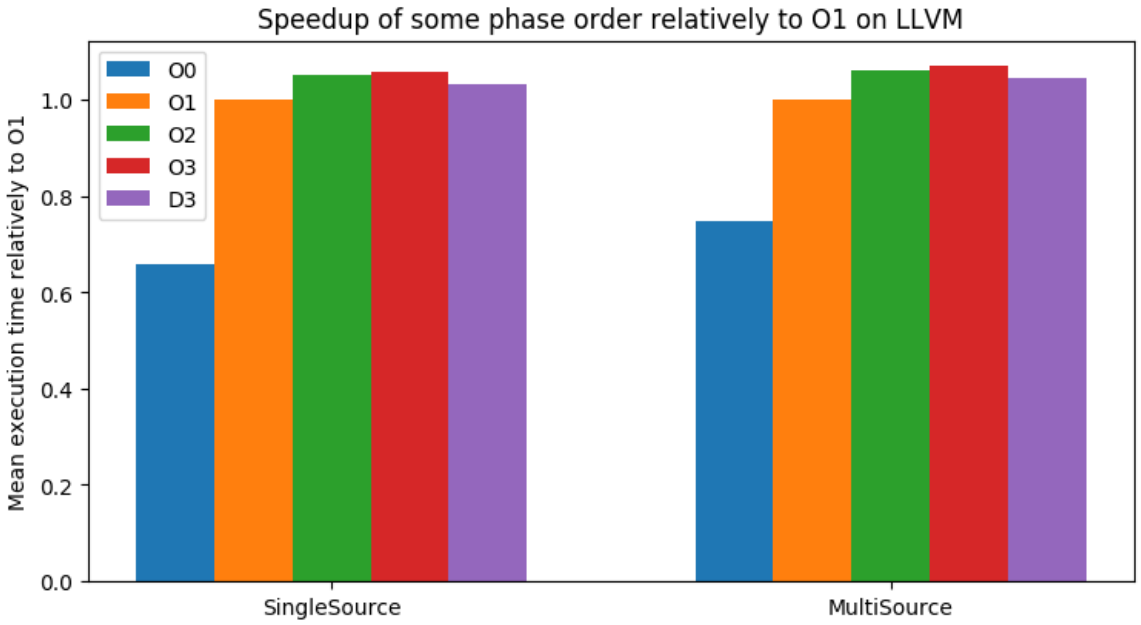

FIGURE 8 – Speedup des différents ordres de passes de base

La figure 8 montre que les résultats obtenus sont bien ceux qu'on attend, à savoir que `O3` est en moyenne plus rapide que `O2`, et que plus la valeur accompagnant `O` est élevée, plus les programmes générés sont rapides.

4.1.2 Reproduction de O3

Notre objectif étant de tester de nouveaux ordres de base, nous souhaitons voir si nous arrivons à reproduire le comportement de `O3` avec OPT.

Sur la figure 8, on a noté `D3` (pour *Decomposed* `O3`) les speedup des programmes compilés en utilisant l'ordre des passes de `O3` que l'on invoque directement avec OPT au lieu de passer l'option.

On remarque que non seulement, les résultats sont inférieurs à ceux de `O3` alors que l'on s'attend à ce qu'ils soient identiques (en dehors des erreurs de mesure), mais ils osnt également inférieurs à ceux de `O2`.

7. Accélération du temps d'exécution

Plusieurs pistes ont été explorées pour tenter d'expliquer cette différence (option permettant l'optimisation des appels à la STL [8], appels à des stratégies d'optimisation de code plus agressives avec l'option *codegen hook*...). Mais elles ont été infructueuses.

4.2 Étude de nouvelles séquences

Conclusion

Le travail a ici porté essentiellement sur l'étude de Clang/LLVM et comment intégrer un choix dynamique de l'ordre des passes. Sur le papier, la démarche est simple car LLVM a une structure très modulaire, permettant de réaliser ce choix pour un programme donné. Le passage à l'échelle est plus problématique car il a fallu déterminer un moyen pratique de choisir les passes et de faire des mesures sur un grand nombre de programmes. L'approche proposée est la réécriture des règles de compilation des programmes de la *test-suite*. Cette approche, bien que simple à mettre en œuvre, a cependant pour inconvénient de dépendre de l'implémentation de la compilation de la TEST-SUITE, en particulier de sa manière de générer ces fichiers de génération.

Deux approches sont envisageables pour poursuivre l'étude du sujet :

— Intégrer le passage de passes à CLANG au lieu d'utiliser OPT. L'implémentation basique consisterait à créer une nouvelle option qui permet de dire à CLANG de remplir le *pass manager* en lisant une variable globale et non plus en utilisant des passes *hard codées*.

— La majorité du travail a porté sur l'étude de LLVM, comment mettre en place l'architecture qu'on veut, et quelle serait la viabilité de le mettre en place. Une architecture ayant été mise en place, on peut commencer à l'exploiter afin de répondre au sujet d'origine, à savoir l'étude de l'impact des passes sur les autres. Au-delà d'une approche "force brute" consistant à tester à l'intuition des ordres de passes jusqu'à en trouver un "meilleur" (selon des métriques qui restent à définir), on peut également imaginer des approches plus "intelligentes" comme un algorithme génétique, ou exploiter les propositions faites par [2] traitant du même sujet.

Références

[1] Sadaf Alam, Benjamin Cumming, and Ugo Varetto. Extending the capabilities of the cray programming environment with clang-llvm framework integration. 2014.

[2] Y. B. Asher, G. Haber, and E. Stein. A study of conflicting pairs of compiler optimizations. In *2017 IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 52–58, Sep. 2017.

[3] Serge Guelton and Sébastien Varrette. Une approche génétique et source à source de l'optimisation de code. In *Rencontres francophones du Parallélisme (RenPar'19), Symposium en Architecture de machines (SympA'13) et Conférence Française sur les Systèmes d'Exploitation (CFSE'7)*, Toulouse, France, September 2009.

[4] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization : An overview of the pips project. In *Proceedings of the 5th International Conference on Supercomputing*, ICS '91, pages 244–251, New York, NY, USA, 1991. ACM.

[5] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Embedded processor design challenges. pages 171–187, New York, NY, USA, 2002. Springer-Verlag New York, Inc.

[6] Chris Lattner and Vikram Adve. LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[7] Maroua Maalej Kammoun. *Low-cost memory analyses for efficient compilers*. Theses, Université de Lyon, September 2017.

8. *Standard Template Library* de C++