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, C++, Fortran ... . 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 plusieurs cas concrets sur l'ensemble de la base de tests proposée par LLVM.

Mots-clés 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 à Montréal).

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 plateformes d'exécution, notamment les machines de calcul parallèle : multi et *many-core*, 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

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

1. Projet d'Orientation en Master, Unité d'enseignement dispensée à l'Université Lyon 1, dans le cadre du Master 1 d'Informatique
2. Compilation, and Analysis, Software and Hardware, `http://www.ens-lyon.fr/LIP/CASH/`
3. Field Programmable Gate Arrays (circuits programmables)

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

```
int n = 55;
```

FIGURE 1 – *Deux programmes sémantiquement équivalents*

pour la variable i.

Les compilateurs comme LLVM et GCC sont capables d'analyser le premier programme, de le transformer pour avoir le second programme exposé, puis de compiler ce dernier.

Pour faire cette transformation, LLVM applique plusieurs transformations, nommées passes, de manière séquentielle. L'objectif de ce POM est de fournir des outils pour étudier comment les passes interagissent entre elles, et quel est l'impact sur les performances de programmes réels (sont-ils plus rapides, consomment-ils moins de mémoire ?).

Nous commençons par expliquer comment fonctionnent CLANG et LLVM, en particulier comment fonctionne le système de passes et quels sont les outils déjà disponibles. Puis à partir de la section 3, nous proposons 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 suite de tests 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. Pour pouvoir l'implémenter, il a fallu étudier le fonctionnement de la suite de tests. Cette étude a mené à l'écriture de notes, dont un exemple est disponible en annexe D.

2 LLVM et les compilateurs

2.1 Compilation d'un programme

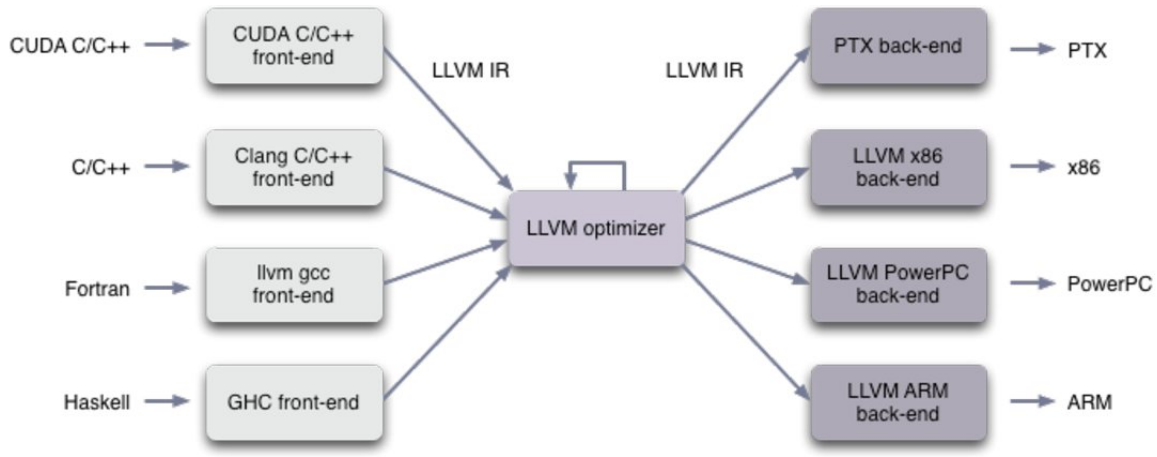

FIGURE 2 – *Fonctionnement de* CLANG *et de* LLVM *extraite de [1]*

La Figure 2 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 de désactiver certaines passes lors de l'exécution de ces combinaisons (si l'utilisateur sait qu'une passe n'améliorera pas son programme par exemple).

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.

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

(a)

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

(c)

```
; [#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
}
```

(b)

FIGURE 3 – *Un programmé écrit en C (a), converti en IR (b) puis optimisé (c)*

La figure 3 présente un exemple de code écrit en C en haut à gauche (a). Le code est transformée vers l'IR présentée à droite (b). Sans entrer dans les détails techniques, on peut voir que l'IR res-

semble à du code assembleur dans le sens où on a des allocations explicites de la mémoire (avec "alloca"), les instructions "load" et "store" et les structures de contrôle sont remplacées par des sauts. Les différentes passes de `O3` sont appliquées, et seul le code en bas à gauche (c) 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 (31506 fichiers), qui est livré avec une base de tests pour valider son développement ainsi que les performances des programmes générés.

2.4.1 LLVM-LIT

LLVM-LIT est l'outil principal permettant de tester et de mesurer les performances de LLVM. Il repose sur une exploration récursive d'un dossier et exécute tous les tests contenus dans celui-ci. 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 mesuré avec l'outil TIMEIT inclus avec LLVM ou avec PERF [4] 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 ne peut pas utiliser l'outil MASSIF de VALGRIND pour récupérer l'usage de mémoire. VALGRIND est appelé avec l'outil "memcheck" qui dédié à la vérification des accès mémoires (l'usage le plus commun de ce mode est de savoir si la mémoire allouée explicitement par l'utilisateur est libérée)

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

2.4.2 La suite de tests

La communauté LLVM propose une base de test. En plus de tests dédiés aux tests unitaires et de non régression, elle contient plus de 292 programmes sous la forme de fichiers *.c* ou *.cpp* dédiés à la mesure des performances en temps.

Elle s'utilise en trois phases :

— La phase de génération utilisant CMAKE. Durant cette phase, CMAKE va parcourir des fichiers nommés CMakeList.txt. Cette étape génère des fichiers contenant les règles de compilation (par exemple MAKEFILE ou NINJA selon le choix de l'utilisateur) permettant de compiler la *test-suite* ;
— À partir des fichiers de configurations et des règles de compilation qu'ils contiennent, MAKEFILE ou NINJA va compiler les différents programmes contenus par la suite de tests. Cette compilation se fait en utilisant le CLANG et les options de compilations passées lors de la phase de génération ;
— La phase d'exécution des tests avec LLVM-LIT. Lors de cette phase, chaque programme compilé est exécuté, puis sa sortie (sortie standard et code de sortie [5]) est vérifiée.

L'utilisation principale de la suite de tests est de tester si les programmes compilent avec le CLANG en train d'être testé, et si ils ont la sortie attendue. Elle permet également de mesurer le temps de compilation et d'exécution des programmes en utilisant les différentes options de CLANG sur différentes configurations de machines.

4. `https://perf.wiki.kernel.org/index.php/Main_Page`
5. Valeur retournée par le programme au système d'exploitation lorsqu'il est terminé

2.4.3 Travaux précédents

L'idée de tester différents ordres de passes sur un grand nombre de programmes s'approche en 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 de passes jusqu'à obtenir 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], dans [3], Guelton et Varrette proposent de générer des configurations avec un algorithme génétique, et testent 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.

3 Nouvelles solutions de tests de performances

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. La littérature parle de *Phase Ordering* pour ce problème. Pour le traiter, deux obstacles se posent à nous.

Nous sommes tout d'abord confrontés au fait que CLANG ne permette 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 résoudre notre problème nous avons besoin de faire 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 déterministes, 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 qui sont explicitement demandées), le poids de l'exécutable généré, le nombre de fois où une passe agit ...

Les deux solutions proposées dans ce document reposent sur deux axes indépendants. Le premier est l'élaboration d'un script indépendant qui s'appuie uniquement sur les outils de base permettant de décomposer CLANG. Le second modifie la manière de compiler la base de tests existante (à savoir la suite de tests 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 sur un unique fichier.

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

Le fichier est 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. Comme nous l'avons vu dans la section 2.4.1 dédiée à LLVM-LIT, ce dernier ne permettait pas de choisir l'outil de VALGRIND utilisé. Ici, nous pouvons accéder à toutes ses fonctionnalités ;
— 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 de l'enrobage de la fonction "main" telle que présentée dans l'annexe A (cela permet de réaliser des mesures plus précises qu'avec un outil extérieur).

3.1.1 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 rapidement été écartée ;

— Intégrer la suite de tests à notre outil. Cette seconde option est complexe à mettre en place car elle implique de réimplémenter certaines fonctionnalités de CMAKE.

Cette première méthode de mesure est donc efficace sur un programme cible déterminé mais ne passe pas à l'échelle. Nous nous proposons donc d'étudier la suite de tests et son fonctionnement.

3.2 Mesures sur la suite de tests de LLVM

Utiliser la *test-suite* telle quelle ne permet pas de répondre à nos besoins. Les programmes qui la composent sont compilés avec CLANG ce qui ne permet pas de choisir l'ordre des passes. La *test-suite* n'a pas été conçue avec une telle modularité à l'esprit mais pour valider le développement du compilateur dans son ensemble et en mesurer les performances.

Nous avons besoin de pouvoir choisir les passes et compiler de 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*. Cette étude a mené à la rédaction de notes, dont un exemple est produit dans l'annexe D.

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 utilisées 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 différences 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 à Clang 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* [7] 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 spécialistes ;

— 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 auraient dû être envoyés 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

6. *Integrated Development Environment* ou Environnement de Développement
7. Le *Pass Manager* est la partie du code de LLMV chargée de stocker la liste des passes et de les appliquer

transformer l'IR en l'exécutable souhaité. Le problème de cette méthode est qu'elle requiert de coder comment répartir les différents arguments reçus entre les programmes appelés.

3.2.2 Réécriture de règles de construction de la suite de test

La solution que nous avons implémentée 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 tests 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).

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 configuration NINJA [8] puis une seconde étape où l'on construit la suite de tests. Dans notre cas, nous allons nous intéresser aux fichiers de configuration 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 4 – *Règle de base telle que générée par* CMAKE *dans "rules.ninja".*

Principe La figure 4 présente une règle ninja telle que générée par défaut pour le test nommé *filter*. La valeur des variables comme *$DEFINES* sont déterminées dans le fichier "build.ninja". La ligne de commande exécutée telle qu'elle est écrite dans le fichier de règles n'a donc aucune connaissance sur le programme cible en dehors du langage utilisé. On peut donc lire la ligne *commands* et la 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 5. 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 5 – *Réécriture automatique de la ligne commands*

8. NINJA (ou NINJA-BUILD), tout comme MAKE est un outil de construction de fichiers. NINJA est favorisé par la communauté LLVM car il est plus léger et plus rapide que MAKE

Faiblesses L'implémentation proposée ne fonctionne pas sur toutes les machines car elle 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. Nous sommes limités à la mesure du temps de compilation, du temps d'exécution et la taille de l'exécutable produit.

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. Nous mesurons uniquement le temps d'exécution des programmes.

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 cohérents 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 *speed-up*[9] 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, on calcule le *speed-up* puis on dessine la moyenne géométrique de ces valeurs pour limiter l'impact des *speed-up* éloignés de la moyenne.

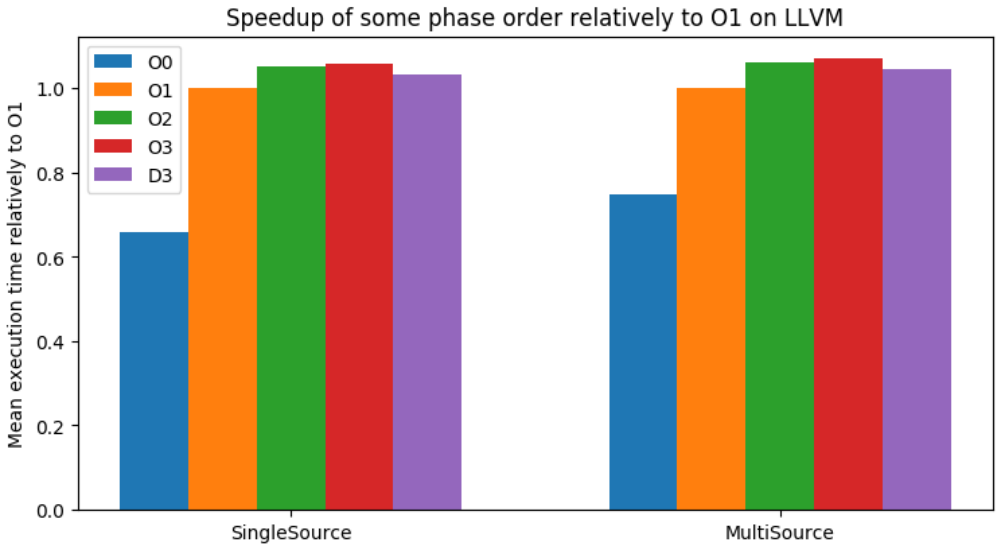

FIGURE 6 – *Speed-up des différents ordres de passes de base et la simulation de O3*

9. Accélération du temps d'exécution

La figure 6 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 passes, nous souhaitons voir si nous arrivons à reproduire le comportement de `O3` avec OPT.

Sur la figure 6, on a noté `D3` (pour *Decomposed* `O3`) les speed-up des programmes compilés en utilisant l'ordre des passes de `O3` que l'on invoque directement avec OPT au lieu de passer l'option `O3` à CLANG.

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 sont également inférieurs à ceux de `O2`.

Plusieurs pistes ont été explorées pour tenter d'expliquer cette différence (option permettant l'optimisation des appels à la STL [10], appels à des stratégies d'optimisation de code plus agressives avec l'option *codegen hook*…). Mais elles ont été infructueuses.

4.2 Étude de l'interaction entre *loop-unswitch* et *licm*

La documentation de LLVM recommande d'utiliser la passe *licm* avant l'utilisation de la passe *loop-unswitch*. Ces deux passes ont pour effet de transformer le code. L'effet de *licm* est décrit dans l'annexe B tandis que l'effet de la passe *loop-unswitch* est décrit dans l'annexe C.

Nous nous proposons de vérifier si l'ordre de ces deux passes a une importance ? en prenant la liste des passes de `O3`, et en n'exécutant qu'une seule fois *loop-unswitch* et *licm* dans cette liste. *loop-unswitch* n'apparaît qu'une fois dans la liste des passes exécutées par `O3`, alors que *licm* apparaît 3 fois. Nous supprimons donc les trois occurrences de *licm* de `O3`, et nous rajoutons une exécution de *licm* dans un premier temps avant la passe *loop-unswitch*, dans un autre temps après.

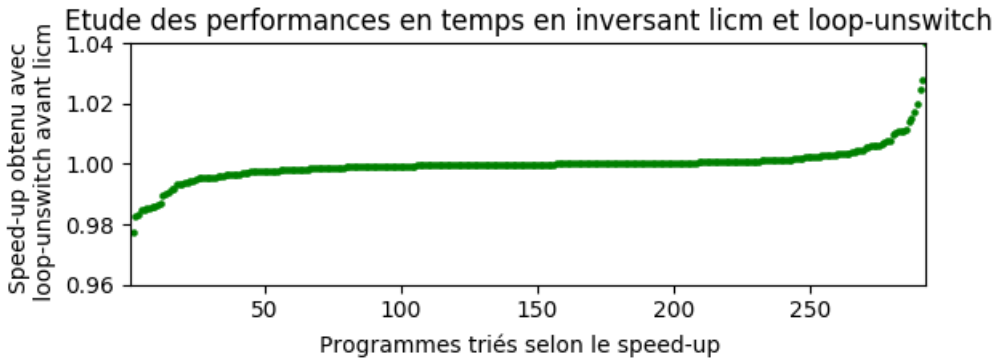

FIGURE 7 – *Speed-up en plaçant loop-unswitch avant licm*

La figure 7 montre que les temps d'exécution sont équivalents selon l'ordre choisi. Sur 292 programmes, 267 ont un temps d'exécution ayant une différence de temps de moins de 1% par rapport à l'autre ordre étudié. Les 25 autres programmes sont répartis équitablement et ne permettent pas de dégager une tendance pour la gestion des cas extrêmes.

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

10. *Standard Template Library* de C++

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 :

— La première approche est une approche technique consistant à améliorer l'outillage proposé :

1. L'utilisation de la suite de tests ne permet de récupérer que le temps d'exécution et la taille des exécutables. La possibilité d'exploiter d'autres métriques, comme une comparaison de l'*IR* permettrait d'avoir des résultats plus fins ;
2. Mener une étude du fonctionnement interne de CLANG, qui mènerait à l'intégration du choix des passes directement dans CLANG, ce qui permettrait de se passer de 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 codées en dur.

— La majorité du travail a porté sur l'étude de LLVM, la faisabilité de la mise en place d'une architecture de tests permettant de choisir les passes et sa mise en place. On peut donc commencer à l'exploiter afin de répondre au sujet d'origine, à savoir l'étude de l'impact des passes sur les autres :

1. Sur la recherche d'un ordre idéal de passes, 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 ;
2. Pour une étude plus en détail des passes existantes, pour poursuivre les travaux initiés dans la section 4.2, on peut se poser la question de la commutativité des passes. Pour dépasser une approche binaire, on peut imaginer associer à la notion de commutativité le pourcentage de programmes produisant des mesures similaires (par exemple dans le cas de *licm* et *loop-unswitch*, on aurait une commutativité de ces passes de 91%).

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.

Annexes

A Enrobage de la fonction *main* d'un programme pour mesurer plus précisément son temps d'exécution

Dans le code de gauche de la figure 8, 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 qui n'a que peu d'importance.

```
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 vrai_main() {
    // Code identique au main
    // précédent
}

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

    for (int i = 0 ;
         i != NOMBRE_DE_MESURES ;
         i++) {
        lancer_chrono();
        if(vrai_main()) return 1,
        arreter_chrono();
    }
    sortie_min_des_temps_chrono()
    return 0;
}
```

FIGURE *8 – 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 $main$. À droite de la figure 8, nous renommons la fonction "main". Nous appelons celle-ci dans la nouvelle fonction main après avoir lancé un chronomètre. A ce stade, la nouvelle fonction $main$ est écrite sans aucune connaissance sur l'ancien $main$, qui a été simplement renommé. Ce procédé est automatisable.

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 de plusieurs fichiers).

B Effet de la passe *licm*

Le rôle de *licm* [11] est de déplacer le code invariant situé dans une boucle (comme par exemple l'affectation d'une variable avec une constante) en dehors de la boucle.

```
int ok = 0:
for (int i = 0 ; i != 10 ; i++) {
  ok = 1;
  foo();
}
```

```
int ok = 0:
ok = 1;
for (int i = 0 ; i != 10 ; i++) {
  foo();
}
```

FIGURE 9 – *Un exemple de code simplifié par la passe licm*

La figure 9 montre un exemple trivial où la variable ok est toujours affectée à la même valeur dans la boucle.

La sémantique du programme ne change pas car ok aura dans les deux cas la valeur 1 en sortie de boucle, mais on économise 9 affectations. Cette optimisation est applicable parce que l'affectation n'a pas d'effet de bord, et que l'on sait que la boucle sera dans notre cas faite au moins une fois (car $i = 0$ vérifie $i \neq 50$).

C Effet de la passe *loop-unswitch*

La passe *loop-unswitch* a pour but d'améliorer le temps d'exécution de boucles ayant des conditions portant sur des variables constantes. Pour cela, elle duplique la boucle pour créer une version où la condition est vérifiée et une version où la condition ne l'est pas.

```
bool b = getABooleanValue();

for (int i = 0 ; i != 50 ; i++) {
  foo();
  if (b)
    bar();
  baz();
}
```

```
bool b = getABooleanValue();

if (b) {
  for (int i = 0 ; i != 50 ; i++) {
    foo(); bar(); baz();
  }
} else {
  for (int i = 0 ; i != 50 ; i++) {
    foo();        baz();
  }
}
```

FIGURE 10 – *Une boucle transformée par la passe loop-unswitch*

La figure 10 montre l'effet de la passe *loop-unswitch* sur un exemple. La variable b a une valeur constante : elle n'est pas amenée à changer lors de l'exécution de la boucle. Au lieu de vérifier 50 fois la valeur de b, on se contente de la vérifier une unique fois.

Cette optimisation améliore les performances en temps au détriment de la taille du code. Elle n'est donc appliquée que pour des boucles dont la taille est en-dessous d'un seuil.

D Exemple de notes : Fonctionnement de la test-suite (en anglais)

11. *Loop Invariant Code Motion*

Using the LLVM test suite

Summary: * Why the stable release of LLVM 7.0.1 * Getting Started guide without SVN * About the compilation of the test suite * LLVM-LIT * Unit tests * Files produced by the compilation of the test suite * Measuring memory usage * Miscellaneous * Adding a single-source test to the test suite

Test suite practical guide: * Configuring the test suite * Compiling the test suite * Benchmarking the compiled programs

In the rest of this document, we consider that this repository is in `/`, so this file is in `/notes`, llvm source code is in `/llvm/` and compiling llvm will have the executable in `/build/bin/`.

The first time, you can almost copy the instructions in `/script/installation.sh`. This script will fail due to having a lot of dependencies (ninja for example) that are not checked and installed by the script.

Why the stable release of LLVM 7.0.1

SVN sources are not reliable (it's the dev branch) and using them force us to ensure that we are using the same revision for every part (llvm, clang, ...).

Using SVN is even less reliable because LLVM is currently migrating from a self hosted SVN versionning system to Github:

https://llvm.org/docs/Proposals/GitHubMove.html http://lists.llvm.org/pipermail/llvm-dev/2019-January/128935.html

We relied to the LLVM Quick start guide to build the script: http://releases.llvm.org/7.0.1/docs/GettingStarted.html

LLVM is currently changing the organization of the code, so documentation that is not archived from the state of LLVM during 7.0.1 release may be obsolete. Since January 2019, LLVM is also changing its source code organization (gets rid of projects and tools folders to directly put the clang folder in the base llvm folder).

We resort to the release sources of LLVM 7.0.1 because it is a stable release: http://releases.llvm.org/download.html

Getting Started guide without SVN

To use the getting started guide, we just need to replace the commands with the pattern `svn co link folder_name` to "unpack the content of the related archive in the folder named folder_name".

If unsure : every archive is a sub-folder of the base llvm archive. We can check which folders can be added in project or tools by reading `CMakeLists.txt` file.

Compilation is slow (it can take several hours). Check `./tips-on-compiling-llvm.txt` to improve the performance of the further compilations (there are some cmake generation options to reuse the files of a previous compilation)

If LLVM is compiled as a shared library (`BUILD_SHARED_LIBS` on), we have to update `LD_LIBRARY_PATH` to include `llvm-build-folder/lib`:

```
% export LD_LIBRARY_PATH="$LD_LIBRATY_PATH:/folder/to/llvm/build/lib"
```

About the compilation of the test suite

LLVM Test Suite is a compilation of test programs and some configurations files for `llvm-lit`.

We can find this quickstart guide about the test suite: https://llvm.org/docs/TestSuiteGuide.html

Unfortunately, there are no archived version of this guide so it may be irrelevant for LLVM 7.0.1 at some point, and the installations steps are confusing.

What we need to do to install the test-suite is :

- Unpack the test-suite in `/llvm/projects/test-suite`
- Have compiler-rt unpacked in `/llvm/projects/compiler-rt`, clang in `/llvm/tools/clang`
- Compile LLVM
- Run the following commands (`/test-suite-build` can be anywhere)

```
% mkdir /test-suite-build
% cd /test-suite-build
% cmake -DCMAKE_C_COMPILER=/build/bin/clang -GNinja \
  -D TEST_SUITE_BENCHMARKING_ONLY=ON -D TEST_SUITE_USE_PERF=ON \
  -C/script/O3.cmake /llvm/projects/test-suite/
% ninja
```

Quick review of the options:

-GNinja

Use Ninja instead of Make (faster in theory; not really in practice).

-D TEST_SUITE_BENCHMARKING_ONLY=ON

Disable tests that are unsuitable for performance measurements. Thery either run for a very short time or are dominated by I/O performance, making them unsuitable as compiler performance tests.

-D TEST_SUITE_USE_PERF=ON

Use `perf` to measure the link time. This is not the tool used to benchmark the programs later on. The default is to use `timeit`, a C program that forks and execs, then uses `waitpid()`. `perf` is more advanced Linux profiling tool, and is recommended by the LLVM community to measure small execution times.

It might need root privilege but often doesn't. We study further the different measuring tools available in comparison-of-measuring-tools.md.

This will compile every program found in `/llvm/projects/test-suite/` using the compiler `/build/bin/clang` with the given cmake file (here is will compile the programs with -O3) and put the executable in `/test-suite-build` following the same structure as in `/llvm/projects/test-suite/`.

For example `/llvm/projects/test-suite/SingleSource/Benchmarks/BenchmarkGame/recursive.c` will be compiled as `/test-suite-build/SingleSource/Benchmarks/BenchmarkGame/recursive` (this means we can run `/test-suite-build/SingleSource/Benchmarks/BenchmarkGame/recursive`).

Compiling the test-suite will register the compilation time and display it when running `llvm-lit`.

ninja compiles every program found in the test-suite. We can compile only certains programs by typing `ninja name_of_the_program`. For example, `ninja Quicksort` will only compile the Quicksort test (found in `SingleSource/Benchmarks/Standford/Quicksort`).

Trivia : > http://llvm.org/docs/lnt/quickstart.html#running-tests > "You should always keep the test-suite directory itself clean (that is, never > do a configure inside your test suite). > Make sure not to check it out into the LLVM projects directory, as LLVM's > configure/make build will then want to automatically configure it for you."

LLVM-LIT

`llvm-lit` is the LLVM Integrated Tester used to run the tests. It's implemented in `/llvm/utils/lit/lit`; in theory it can use different measuring tools including `perf`, but in practice `strace` does not say so. It might fall back to an implementation at line 188 of `run.py` that uses `time.time()`.

https://llvm.org/docs/CommandGuide/lit.html

`llvm-lit` reads files and looks for RUN / CHECK instructions.

Unit tests

They are found in `llvm/test`, can't be compiled and are only used for non regression.

```
bruju@bruju-portable:~/pom/llvm/test/Integer$ cat BitPacked.ll
; RUN: llvm-as %s -o - | llvm-dis > %t1.ll
; RUN: llvm-as %t1.ll -o - | llvm-dis > %t2.ll
; RUN: diff %t1.ll %t2.ll

@foo1 = external global <4 x float>
@foo2 = external global <2 x i10>
```

```
define void @main()
{
        store <4 x float> <float 1.0, float 2.0, float 3.0, float 4.0>, <4 x float>* @foo1
        store <2 x i10> <i10 4, i10 4>, <2 x i10>* @foo2
    %l1 = load <4 x float>, <4 x float>* @foo1
        %l2 = load <2 x i10>, <2 x i10>* @foo2
        %r1 = extractelement <2 x i10> %l2, i32 1
        %r2 = extractelement <2 x i10> %l2, i32 0
        %t = mul i10 %r1, %r2
        %r3 = insertelement <2 x i10> %l2, i10 %t, i32 0
        store <2 x i10> %r3, <2 x i10>* @foo2
        ret void
}
bruju@bruju-portable:~/pom/llvm/test/Integer$ llvm-lit BitPacked.ll -a
-- Testing: 1 tests, 1 threads --
PASS: LLVM :: Integer/BitPacked.ll (1 of 1)
Script:
--
: 'RUN: at line 1';   /home/bruju/pom/build/bin/llvm-as /home/bruju/pom/llvm/test/Integer/Bi
: 'RUN: at line 2';   /home/bruju/pom/build/bin/llvm-as /home/bruju/pom/build/test/Integer/O
: 'RUN: at line 3';   diff /home/bruju/pom/build/test/Integer/Output/BitPacked.ll.tmp1.ll /h
--
Exit Code: 0


********************
Testing Time: 0.10s
  Expected Passes    : 1
```

This test checks that using `llvm-as` and `llvm-dis` once or twice gives the same result. CHECK instructions can also be used to check if an optimization has the intended effect on a small snippet.

As we are benchmarking optimizations, small units tests are not relevant for us.

Files produced by compilation of the test suite

Here we study the program `recursive.c`. It's a small C program that computes the results of some mathematical sequences (like Fibonnaci) and displays the result.

When compiled, here are the generated files

```
bruju@bruju-portable:~/pom/test-suite-build/SingleSource/Benchmarks/BenchmarkGame$ ls -al |
-rwxr-xr-x  1 root  root   8312 Feb  3 09:40 recursive
-rw-r--r--  1 root  root     38 Feb  3 09:40 recursive.link.time
```

```
-rw-r--r--  1 root  root    965 Feb  3 09:40 recursive.link.time.perfstats
-rw-r--r--  1 bruju bruju   249 Feb  3 08:59 recursive.test
```

- `recursive` is the executable;
- `recursive.link.time` is a summary of the link time; `.perfstats` is the suffix when `perf` is used. This is not the execution time of the compiled program, but that of the compiler (linker)!
- `recursive.test` stores the commands to run the test and check if its output is the intended one.

`fpcmp` is a tool to check the difference between two files (output) with som tolerance (for example ignoring whitespace).

```
bruju@bruju-portable:~/pom/test-suite-build/SingleSource/Benchmarks/BenchmarkGame$ cat recur
RUN: /home/bruju/pom/test-suite-build/SingleSource/Benchmarks/BenchmarkGame/recursive
VERIFY: /home/bruju/pom/test-suite-build/tools/fpcmp %o /home/bruju/pom/llvm/projects/test-s

bruju@bruju-portable:~/pom/test-suite-build/SingleSource/Benchmarks/BenchmarkGame$ cat recur
# started on Sun Feb  3 09:40:19 2019


 Performance counter stats for '/home/bruju/pom/build/bin/clang -O3 SingleSource/Benchmarks/

        133.329486      task-clock (msec)         #    0.513 CPUs utilized
                24      context-switches          #    0.180 K/sec
                 2      cpu-migrations            #    0.015 K/sec
             6,733      page-faults               #    0.050 M/sec
       345,668,653      cycles                    #    2.593 GHz
       249,822,974      instructions              #    0.72  insn per cycle
        50,037,482      branches                  #  375.292 M/sec
         1,115,546      branch-misses             #    2.23% of all branches

       0.260005240 seconds time elapsed
```

Let's run llvm-lit on `recursive.test`:

```
bruju@bruju-portable:~/pom/test-suite-build/SingleSource/Benchmarks/BenchmarkGame$ sudo llvm
-- Testing: 1 tests, 1 threads --
PASS: test-suite :: SingleSource/Benchmarks/BenchmarkGame/recursive.test (1 of 1)
********** TEST 'test-suite :: SingleSource/Benchmarks/BenchmarkGame/recursive.test' RESULTS
compile_time: 0.8584
exec_time: 0.7465
hash: "77ab3049ec64d21bc56c887eca29c9e9"
link_time: 0.1333
size: 8312
size..bss: 8
size..comment: 87
size..data: 16
```

```
size..dynamic: 480
size..dynsym: 96
size..eh_frame: 424
size..eh_frame_hdr: 100
size..fini: 9
size..fini_array: 8
size..gnu.hash: 28
size..gnu.version: 8
size..gnu.version_r: 32
size..got: 16
size..got.plt: 32
size..init: 23
size..init_array: 8
size..interp: 28
size..note.ABI-tag: 32
size..plt: 32
size..rodata: 143
size..text: 1090
**********
Testing Time: 0.92s
  Expected Passes    : 1
```

Everything is static but `exec_time` and `Testing time`. The given `exec_time` is the same as the one given by `time ./recursive` and `perf stat ./recursive`.

`llvm-lit` has these options:

- `-a` displays run commands;
- `-j <number>` controls the number of parrallel jobs;
- `-o <file>` outputs the results in the given file.

It also has some configuration files.

TODO : `.cfg` structure

Measuring memory usage

When benchmarking, `llvm-lit` will only run one executable.

```
bruju@bruju-portable:~/pom/test-suite-build/SingleSource/Benchmarks/BenchmarkGame$ cat recur
RUN: /home/bruju/pom/test-suite-build/SingleSource/Benchmarks/BenchmarkGame/recursive
RUN: valgrind --tool=massif --stacks=yes /home/bruju/pom/test-suite-build/SingleSource/Bench
VERIFY: /home/bruju/pom/test-suite-build/tools/fpcmp %o /home/bruju/pom/llvm/projects/test-s
bruju@bruju-portable:~/pom/test-suite-build/SingleSource/Benchmarks/BenchmarkGame$ llvm-lit
-- Testing: 1 tests, 1 threads --
More than one executable used in test test-suite :: SingleSource/Benchmarks/BenchmarkGame/re
NOEXE: test-suite :: SingleSource/Benchmarks/BenchmarkGame/recursive2.test (1 of 1)
Testing Time: 0.10s
```

The `--vg` option doesn't seem to let us use massif (a valgrind tool to profile memory usage) because `--leak-check=no` option is automatically added (if we specify –vg-check, `--leak-check-full` is added instead). There is no workaround without modifying the source code (`utils/lit/lit/LitConfig.py`, lines 60 to 64).

Note: just rewrite the PATH and remove the argument from within a script.

Miscellaneous

Tests on a remote machine is possible using ssh (might be useful to compare performances on a "standard" computer and on the ENS servers)

LNT: http://llvm.org/docs/lnt/index.html http://llvm.org/docs/lnt/quickstart.html#quickstart

Client - Server based infrastructure to store and visualize performance data generated by llvm test suite.

It is the recommanded way to run tests by LLVM community.

Adding a single-source test to the test suite

An example of a diff where a single source test named `revertBits.c` is added.

https://reviews.llvm.org/file/data/62jzeoji3i7fnqdmabrr/PHID-FILE-e4wncsf7pdbtkccz4cbc/D35188.diff

When adding a new test in an existing repertory :

- Edit CMakeLists.txt. If Source is specified, add the `.c` file. If it's not specified, the test suite will compile every `.c` file in the repertory into an individual executable if `llvm_singlesource()` is in the `CMakeLists.txt`.
- Put the source (`new_test.c`) and the reference output (`new_test.reference_output`)

Test suite practical guide

Running the test-suite requires three steps:

- Configuring the test suite with `CMake`
- Compiling the test-suite with `ninja`
- Benchmarking the compiled programs with `llvm-lit` (or something else)

When we rewrite the rules, we are adding a step between the first and the second original steps to modify the build step.

Configuring the test suite

```
% cmake -DCMAKE_C_COMPILER=/build/bin/clang -GNinja \
  -D TEST_SUITE_BENCHMARKING_ONLY=ON -D TEST_SUITE_USE_PERF=ON \
```

```
  -C/script/O3.cmake /llvm/projects/test-suite/
```

This command reads the `CMakeLists.txt` files to builds the different generation rules and also the `lit.cfg` and `lit.site.cfg.in` to configure `llvm-lit`. All these files are found in `/llvm/projects/test-suite/`.

`CMakeLists.txt` includes other files, and its code leads to the reading of every other `CMakeLists.txt` files in the subfolders of `/llvm/projects/test-suite/`.

Several files are generated, in particular to `build.ninja` and `rules.ninja`.

`/python/test-suite-builder.py` and `/testsuite/configure.py` (same script) are provided to run this step automatically.

Compiling the test suite

```
% ninja
```

This command reads `build.ninja` and `rules.ninja`. The second one contains all the commands used for compilation and linking, and is transformed through rule rewriting to customize compilation options.

Here's an example of rule:

```
#############################################
# Rule for compiling C files.

rule C_COMPILER__recursive
  depfile = $DEP_FILE
  deps = gcc
  command = /test-suite-build/tools/timeit --summary $out.time //build/bin/clang $DEFINES $I
  description = Building C object $out


#############################################
# Rule for linking C executable.

rule C_EXECUTABLE_LINKER__recursive
  command = $PRE_LINK && /test-suite-build/tools/timeit --summary $TARGET_FILE.link.time /bu
  description = Linking C executable $TARGET_FILE
  restat = $RESTAT
```

We can see that the provided rules for every compilation uses no knowledge about the program. Instead, there are several variables (like `$FLAGS`). Their value is specified in the `build.ninja` file. Note that the `build.ninja` includes other sections than the ones to specify variables.

For example for the `recursive` test, a section of the `build.ninja` file:

```
# Object build statements for EXECUTABLE target recursive
```

```
############################################
# Order-only phony target for recursive

build cmake_object_order_depends_target_recursive: phony || tools/fpcmp-host tools/timeit-ho
build SingleSource/Benchmarks/BenchmarkGame/CMakeFiles/recursive.dir/recursive.c.o: C_COMPIL
  DEFINES = -DNDEBUG
  DEP_FILE = SingleSource/Benchmarks/BenchmarkGame/CMakeFiles/recursive.dir/recursive.c.o.d
  FLAGS = -O3   -w -Werror=date-time
  OBJECT_DIR = SingleSource/Benchmarks/BenchmarkGame/CMakeFiles/recursive.dir
  OBJECT_FILE_DIR = SingleSource/Benchmarks/BenchmarkGame/CMakeFiles/recursive.dir
  TARGET_COMPILE_PDB = SingleSource/Benchmarks/BenchmarkGame/CMakeFiles/recursive.dir/
  TARGET_PDB = SingleSource/Benchmarks/BenchmarkGame/recursive.pdb

# =============================================================================
# Link build statements for EXECUTABLE target recursive


############################################
# Link the executable SingleSource/Benchmarks/BenchmarkGame/recursive

build SingleSource/Benchmarks/BenchmarkGame/recursive: C_EXECUTABLE_LINKER__recursive Single
  FLAGS = -O3
  LINK_LIBRARIES = -lm
  OBJECT_DIR = SingleSource/Benchmarks/BenchmarkGame/CMakeFiles/recursive.dir
  POST_BUILD = :
  PRE_LINK = :
  TARGET_COMPILE_PDB = SingleSource/Benchmarks/BenchmarkGame/CMakeFiles/recursive.dir/
  TARGET_FILE = SingleSource/Benchmarks/BenchmarkGame/recursive
  TARGET_PDB = SingleSource/Benchmarks/BenchmarkGame/recursive.pdb
# =============================================================================
```

Running `ninja` compiles every program of the test suite, and registers its compilation (linking) time and reference output.

For example, the `recursive` program, here are the built files:

```
# Executable
SingleSource/Benchmarks/BenchmarkGame/recursive
# Instructions for llvm-lit to run this test
SingleSource/Benchmarks/BenchmarkGame/recursive.test
# Compilation time
SingleSource/Benchmarks/BenchmarkGame/recursive.link.time
# Extra information about the compilation time (provided by perf if used)
SingleSource/Benchmarks/BenchmarkGame/recursive.link.time.perfstats
```

Benchmarking the compiled programs

```
% llvm-lit . -j <number> -s -o results.json
```

Look for every `.test file` in the current folder and its subfolder and runs their instructions, checking that the output is correct.

Measures done (executable size, compilation time, execution time, ...) are registered in the `results.json` file.

An alternative is to use `/testsuite/configure-perf.py` to make measures with `perf`. Passing `DTEST_SUITE_USE_PERF=ON` at configure time is apparently not enough (see `strace` of the previous command), plus `llvm-lit` only makes one repetition.