Team Buffalo : Rapport projet BLASter

I. Introduction

Le but du projet était de créer un compilateur source à source qui remplace automatiquement des bouts de code effectuant des calculs BLAS (*Basic Linear Algebra Subprograms*, une spécification de fonctions faisant des opérations d'algèbre linéaire) implémentés de façon non-optimale dans le code source par un appel de fonction venant d'une bibliothèque qui implémente le même calcul mais de façon optimisée. L'intérêt de se référer à BLAS était que beaucoup de constructeurs font des bibliothèques optimisées pour cette spécification et automatiser l'optimisation était une motivation pour gagner du temps.

Team Buffalo

- BEFOLE Benjamin
- ROSTAQI Yossef
- ELHADDAD Hamza
- CHERGUI MALIH Ilyes

Promo M1 SIL 2019-2020, Université de Strasbourg

II. Fonctionnement

Le programme (sans option) prend en entrée deux paramètres, le fichier du code source à parcourir et le fichier de spécification. Le code source doit être écrit dans un langage qui est un sous-ensemble du langage C pour qu'il soit supporté par notre grammaire, le fichier de spécification est une liste de fonctions qui ont leur signature qui proviennent d'une bibliothèque optimisée, et un corps de fonction qui correspond au type d'implémentation qu'on s'attend à retrouver dans le programme source (une implémentation assez naïve dans le cas de nos fichiers de test).

Une fois le programme lancé, on effectue une analyse lexicale (via Lex) et syntaxique (via Yacc) pour créer un arbre de syntaxe abstraite (*abstract syntax tree,* ou AST) au fur et à mesure que l'analyse syntaxique se déroule sur le code source en entrée. Une table des symboles est créée à partir de cet AST. On crée un second AST mais cette fois qui correspond à la liste des fonctions spécifiées dans le second fichier en entrée.

Nous fournissons un jeu de tests pour chaque niveau de BLAS (niveau 1, 2 et 3) ainsi que les fichiers de spécification correspondantes (lib_1.c, lib_2.c et lib_3.c).

Déroulement de l'optimisation

L'optimisation se fait à partir de l'AST du code source, l'AST du fichier de spécification et de la table des symboles du code source.
Pour chaque fonction dans l'AST du fichier de spécification, on compare le corps de la fonction spécifiée, à l'AST du code source que l'on parcourt, si les deux sont similaires dans leur construction alors on effectue une analyse plus poussée afin de vérifier si l'on peut vraiment remplacer le bout du code source par l'appel de fonction.

Pour effectuer cette analyse plus profonde on a besoin de savoir si les variables de la boucle créent la même sémantique pour ce bout de code (par exemple un *gemm* implémenté naïvement peut être similaire à une double boucle *for* effectuant des opérations matricielles mais si cette double boucle for itère sur des variables sémantiquement différents alors il n'y a rien à remplacer).

Pour chaque variable rencontré on regarde s'il peut être remplacé par la variable correspondante dans l'AST du code source, on parcourt le même cheminement sur l'AST du code source et l'AST de la fonction (puisqu'ils ont une structure similaire).
Ensuite :

- Si on tombe sur l'identifiant de la variable dont on cherche à déterminer sa remplaçabilité, on stocke le pointeur vers l'identifiant correspondant dans l'AST du code source (grâce à la table des symboles)

- Et on regarde si, la prochaine fois que l'on rencontre cet identifiant dans l'AST de la fonction, le pointeur stocké est toujours le même

- Si c'est le cas alors cette variable est remplaçable, sinon elle ne l'est pas et on peut tout de suite arrêter la procédure et ne pas remplacer ce bout de code

À la fin si toutes les variables sont remplaçables alors on peut enfin supprimer ce sous-arbre dans l'AST du code source et le remplacer par un appel de fonction ayant la signature de la fonction spécifiée dans le second fichier en entrée.

Finalisation

Enfin, après que toutes les AST de fonctions spécifiées ont été comparées avec les sous-arbres de l'AST du code source, on peut enfin prendre ce dernier et (qu'il ait été transformé pour remplacer des bouts de code par des appels de fonctions ou non) le traduire en un code C indenté.

Le code C généré n’est pas forcément le plus élégant visuellement, il y a des artefacts au niveau des parenthésages très protecteurs mais il est fonctionnel si on spécifie la bibliothèque optimisée dans le fichier de spécification des fonctions (sinon le code ne pourra pas compiler vu qu’il n’intègre pas la bibliothèque optimisée).