Université de Strasbourg

Master Informatique
Compilation
2019 / 2020

Projet BLASter : Optimisation par appel de bibliothèques

Le but du projet est de réaliser un compilateur source-à-source qui remplace certaines parties de code par des appels à des bibliothèques optimisées.

1 Introduction

Les architectures matérielles sont devenues tellement complexes que réussir à approcher leur performance crête est réservé à une forme d'élite de programmeurs qui connaissent intimement à la fois le problème à résoudre et l'architecture cible. Il est cependant possible de mettre à disposition l'expertise de ces programmeurs au travers de bibliothèques hautement optimisées auxquelles les utilisateurs peuvent faire appel. Pour certains calculs particulièrement utiles, les constructeurs de matériel fournissent bien souvent de telles bibliothèques dès la sortie de leurs nouveaux processeurs. Ainsi Intel fournit la bibliothèque propriétaire MKL (Math Kernel Library) qui propose des versions optimisées des principaux noyaux de calculs mathématiques (par exemple somme des éléments d'un vecteur, multiplication matrice-vecteur, multiplication de matrices, etc.) pour ses architectures. Nvidia fournit cuBLAS (CUDA Basic Linear Algebra Subroutine library) de la même manière pour ses processeurs graphiques.

Les programmeurs ne font pas toujours appel à ces bibliothèques malgré le gain de performance qu'elles peuvent apporter. Les raisons sont multiples : coût de leur licence, ignorance de leur existance, volonté de maîtriser l'ensemble du code, ou simplement le fait que la bibliothèque n'existait pas au moment de la création du logiciel. Dans ce projet, nous vous proposons de réaliser une solution automatique à ce problème. Vous allez construire un compilateur capable de reconnaître certaines parties d'un code correspondant à une fonction de bibliothèque connue, et vous allez substituer à ce code un appel à cette fonction.

2 BLAS : Basic Linear Algebra Subprograms

L'ensemble de fonctions importantes à optimiser le plus connu est appelé « BLAS » (Basic Linear Algebra Subprograms). Il s'agit de fonctions simples exécutant des calculs classiques d'algèbre linéaire sur des vecteurs ou des matrices, telles que l'addition de vecteurs, la multiplication d'un vecteur par un scalaire, le produit scalaire, la multiplication matrice-vecteur ou la multiplication de matrices. Ces routines ont été identifiées et standardisées à la fin des années 1970. Elles sont si importantes que les constructeurs de processeurs ainsi que d'autres sociétés en fournissent des implémentations libres ou payantes extrêmement optimisées.

Les BLAS sont découpées en sous-ensembles appelés « *levels* » qui correspondent à la fois à l'ordre chronologique d'ajout des fonctionnalités et à leur niveau de complexité algorithmique. On dénombre trois niveaux :

— **Level 1 –** Le premier niveau consiste uniquement en des opérations sur des vecteurs et produisant un nouveau vecteur ou un scalaire (addition de vecteurs, produit scalaire, norme d'un vecteur, etc.). La routine emblématique de ce niveau est l'addition générale de vecteurs appelée « axpy » (pour « a (fois) x plus y ») :

$$\vec{y} \leftarrow \alpha\vec{x} + \vec{y}$$

— **Level 2 –** Le second niveau contient les opérations entre matrices et vecteurs. Sa routine emblématique est la multiplication matrice-vecteur généralisée appelée « gemv » (*generalized matrix (times) vector*) :

$$\vec{y} \leftarrow \alpha \mathbf{A}\vec{x} + \beta\vec{y}$$

— **Level 3 –** Le troisième niveau contient les opérations entre matrices. Sa routine emblématique est la multiplication de matrices généralisée appelée « gemm » (*generalized matrix (times) matrix*) :

$$\mathbf{C} \leftarrow \alpha \mathbf{A}\mathbf{B} + \beta\mathbf{C}$$

Des informations complémentaires et la spécification complète seront trouvées sur la page suivante :

`http://www.netlib.org/blas/`

3 But du projet

L'objectif du projet est de développer un compilateur source à source (c'est à dire lisant un code source et produisant un nouveau code source) permettant d'identifier dans un code les sous parties correspondant à des fonctions connues disponibles dans des bibliothèques, et de les remplacer par le bon appel à bibliothèque. Votre compilateur prendra en entrée deux fichiers : d'une part le code source à modifier, et d'autre part un fichier contenant les spécifications des fonctions de bibliothèques connues. Il produira en sortie le nouveau fichier C.

3.1 Langage supporté

Le langage supporté est un sous-ensemble du langage C. Les restrictions par rapport à C sont les suivantes :

— Les seuls types possibles sont l'entier `int` et le flottant `double`, ainsi que les tableaux multidimensionnels de ces types. Toutes les variables sont locales et statiques. Il est également possible de définir des constantes de ces types. Le support du type entier `int` est prioritaire : si vous n'avez pas le temps de supporter le type `double`, ne supportez que `int` en le faisant bien.
— Les opérateurs possibles sont `+`, `-` (unaire et binaire) `*`, `/`, `++` et `--`.
— Les structures de contrôle possibles sont :
  — les conditionnelles `if` avec ou sans `else`,
  — les boucles `while`,
  — les boucles `for` qui fonctionnent comme des boucles itératives (celles du C sont en fait beaucoup plus générales) : la première partie correspond à l'initialisation d'un itérateur, la seconde à la condition d'arrêt de la boucle et la troisième à la mise à jour de l'itérateur.
— Les appels de fonctions, y compris récursivement, sont possibles. Cependant il vous est demandé de vous concentrer sur le reste avant de tenter d'ajouter le support des fonctions. Dans un premier temps, seule la fonction `main()` sans argument sera présente.
— La bibliothèque standard ne fournit que la fonction *printf()* après inclusion de *stdio.h*. On suppose que cette fonction ne sera jamais appelée dans les fonctions de bibliothèques connues. Votre compilateur pourra se servir de cette propriété pour lui réserver un traitement simplifié (par exemple avec un simple copier-coller textuel des arguments depuis le fichier d'origine vers le fichier de destination).

3.2 Fonctions de bibliothèques connues

La spécification des fonctions de bibliothèques connues est laissée libre : c'est à vous de définir la forme de cette spécification. Les seules contraintes sont que cette spécification doit être textuelle et lisible par un humain. La spécification devra fournir les éléments utiles à votre compilateur pour repérer des parties de code qui correspondent à des fonctions connues et remplacer ces parties par les appels aux fonctions correspondantes. Il n'est pas nécessaire d'avoir l'implémentation de la bibliothèque pour la spécifier. Cependant une forme d'implémentation peut bien sûr être la spécification elle-même : cela est laissé à votre appréciation.

Les BLAS sont donnés comme exemple, mais votre spécification de fonctions connues pourra ne contenir qu'un sous-ensemble des BLAS et d'autres fonctions qui n'appartiennent pas aux BLAS mais que vous jugez pertinentes. Au minimum, votre spécification devra être capable de reconnaître les fonctions `axpy` pour les fonctions de type « *level 1* », `gemv` pour les fonctions de type « *level 2* » et `gemm` pour les fonctions de type « *level 3* », toutes sur des nombres entiers, dans cet ordre de priorité : proposez d'abord la reconnaissance de fonctions de niveau 1 et ainsi de suite.

Voici deux pistes possibles pour aborder ce problème, sans aucune obligation de les suivre. Piste 1 : spécifiez une grammaire pour chaque fonction à reconnaître et utilisez Lex et Yacc pour les localiser dans le code source d'origine. Piste 2 : spécifiez les fonctions à reconnaître sous forme de fonctions naïves, utilisez votre parseur pour les mettre sous forme d'arbres de syntaxe abstraite, et recherchez ces arbres dans l'arbre de syntaxe abstraite du programme original.

3.3 Exemple

La figure 1 présente un exemple possible (il sera adapté en fonction de votre spécification de fonctions connues). Les entrées sont présentées en figure 1(a) et figure 1(b) et correspondent respectivement au fichier source d'entrée et à une spécification des fonctions connues. Ici la spécification est une implémentation C naïve mais cette forme n'est absolument pas imposée : vous êtes libres de la choisir de la manière qui vous semblera la plus pertinente. On peut repérer que les trois premières boucles du code original sont des formes de la première fonction connue, de même la quatrième boucle est une forme de la troisième fonction connue. Le compilateur doit donc les détecter et remplacer ces parties de code par un appel adapté à la fonction de la bibliothèque qui convient, ce qui mène au code final en figure 1(c).

Comme on peut le voir sur l'exemple, les morceaux de code qui n'ont pas pu être identifiés restent inchangés, et de même il peut y avoir des fonctions connues qui sont inutiles.

4 Aspects pratiques et techniques

Le compilateur BLASter devra être écrit en C à l'aide des outils Lex et Yacc. Le compilateur devra réaliser l'analyse lexicale et syntaxique du sous ensemble de C (et donc indiquer les erreurs de syntaxe par exemple). Le code fourni en entrée devra être traduit sous la forme d'un **arbre de syntaxe abstraite** (AST). L'implémentation de cette structure de données vous est laissée libre, cependant vous devrez fournir une fonction d'affichage textuel de cet arbre, ainsi qu'une fonction de traduction de cet arbre en un code C. Ainsi votre compilateur source-à-source devra analyser le code en entrée pour le traduire sous la forme d'un AST, éventuellement effectuer les opérations d'identification de fonctions connues et de remplacement sur l'AST, puis enfin traduire cet AST dans le code cible. Il pourra y avoir des différences entre le code d'entrée et celui de sortie même sans identification de fonctions connues (perte des commentaires, changement d'indentation, etc.) sans pénalité tant que le code est sémantiquement équivalent à l'original.

```c
#include <stdio.h>
#define N 16

int main() {
  int i;
  int ty[N], tx[N];

  for (i = 0; i < N/2; i++) {
    tx[i] = 1;
  }
  for (i = N/2; i < N; i++) {
    tx[i] = 2;
  }

  for (i = 0; i < N; i++) {
    ty[i] = 2 * N + 1;
  }

  for (i = N/4; i < 3*N/4; i++) {
    ty[i] = (N + 5) * tx[i] + ty[i];
  }

  for (i = 0; i < N; i++) {
    printf("%d ", ty[i]);
  }
  printf("\n");
  return 0;
}
```

(a) Exemple de code source

```c
#include "spec.h"

void int_vec_set(int lo, int hi, int* v, int val) {
  for (int i = lo; i < hi; i++) {
    v[i] = val;
  }
}

void int_vec_swap(int lo, int hi, int* v1, int* v2) {
  for (int i = lo; i < hi; i++) {
    int tmp = v2[i];
    v2[i] = v1[i];
    v1[i] = tmp;
  }
}

void int_axpy(int lo, int hi, int a, int* x, int* y) {
  for (int i = lo; i < hi; i++) {
    y[i] = a * x[i] + y[i];
  }
}
```

(b) Spécification des fonctions connues (ici sous la forme d'une implémentation naïve)

```c
#include "spec.h"
#include <stdio.h>
#define N 16

int main() {
  int i;
  int ty[N], tx[N];

  int_vec_set(0, N / 2, tx, 1);
  int_vec_set(N / 2, N, tx, 2);
  int_vec_set(0, N, ty, 2 * N + 1);
  int_axpy(N / 4, 3 * N / 4, N + 5, tx, ty);

  for (i = 0; i < N; i++) {
    printf("%d ", ty[i]);
  }
  printf("\n");
  return 0;
}
```

(c) Code généré final

FIGURE 1 – Exemple d'entrées et de sortie d'un compilateur BLASter

Concernant l'identification et le remplacement de parties de code, évidemment vous ne pourrez pas toujours reconnaître les fonctions si elles sont trop déformées dans le code d'origine. Commencez par des identifications strictes et simples et tentez de rendre votre reconnaissance de plus en plus puissante. Parfois plusieurs reconnaissances seront possibles : choisissez celle la plus intéressante (un level 2 au lieu d'un level 1 par exemple). À vous de nous montrer votre créativité et de nous l'indiquer dans le compte rendu et lors de la démo.

Ce travail est à réaliser en équipe dans le cadre du cours de *Compilation* et du cours de *Conduite de Projets*, et à rendre à la date indiquée par vos enseignants en cours et sur Moodle.

Une démonstration finale de votre compilateur sera faite durant la dernière séance de TP. Vous devrez rendre sur Moodle dans une archive :

— Le code source de votre projet complet dont la compilation devra se faire simplement par la commande « make ». Le nom de l'exécutable produit doit être « blaster »
— Un document détaillant les capacités de votre compilateur, c'est-à-dire ce qu'il sait faire ou non, ainsi que l'explication de votre spécification de fonctions connues. Soyez honnêtes, indiquez bien les points intéressants que vous souhaitez que le correcteur prenne en compte car il ne pourra sans doute pas tout voir dans le code.
— Un jeu de tests.

Votre compilateur devra fournir les options suivantes :

— `-version` devra indiquer les membres du projet.
— `-tos` devra afficher la table des symboles.
— `-ast` devra afficher l'AST (avant et après identification des fonctions connues le cas échéant).
— `-o <name>` devra écrire le code résultat dans le fichier `name`.

5 Recommandations importantes

Écrire un compilateur est un projet conséquent, il doit donc impérativement être construit incrémentalement en validant chaque étape sur un plus petit langage et en ajoutant progressivement des fonctionnalités ou optimisations. Une démarche extrême et totalement contre-productive consiste à écrire la totalité du code du compilateur en une fois, puis de passer au débogage ! Le résultat de cette démarche serait très probablement nul, c'est-à-dire un compilateur qui ne fonctionne pas du tout ou alors qui reste très bogué.

Par conséquent, nous vous conseillons de développer tout d'abord un compilateur *fonctionnel* mais *limité* à la traduction d'expressions arithmétiques simples, sans structures de contrôle. À partir d'une telle version fonctionnelle, il vous sera plus aisé de la faire évoluer en intégrant telle ou telle fonctionnalité, ou en considérant des expressions plus complexes, ou en intégrant telle ou telle structure de contrôle. De plus, même si votre compilateur ne remplira finalement pas tous les objectifs, il sera néanmoins capable de générer des programmes corrects et qui «marchent» !

Concernant l'aspect reconnaissance de fonctions connues, nous vous suggérons de commencer par un reconnaître des formes très simples, par exemple ce niveau n'appartenant pas aux BLAS (mais pas délirant pour autant) :

— **Level 0 –** Le niveau 0 consiste en des opérations sur des scalaires. La routine emblématique de ce niveau est la multiplication-accumulation « mulacc » (pour « *multiply-accumulate* ») pour laquelle il existe souvent une instruction machine dédiée :

$$a \leftarrow a + b \times c$$

6 Précisions concernant la notation

— Si votre projet ne compile pas ou plante directement, la note 0 (zéro) sera appliquée : l'évaluateur n'a absolument pas vocation à aller chercher ce qui pourrait éventuellement ressembler à quelque chose de correct dans votre code. Il faut que votre compilateur s'exécute et qu'il fasse quelque chose de correct, même si c'est peu.
— Si vous manquez de temps, préférez faire moins de choses mais en le faisant bien et de bout en bout : on préférera un compilateur incomplet mais qui génère un programme C compilable et exécutable en passant par un AST minimal à une analyse syntaxique seule.

— Élaborez des tests car cela fait partie de votre travail et ce sera donc évalué.
— Faites les choses dans l'ordre et focalisez sur ce qui est demandé. L'évaluateur pourra tenir compte du travail fait en plus (support des fonctions par exemple) seulement si ce qui a été demandé a été fait et bien fait.
— Une conception modulaire et lisible sera fortement appréciée (et inversement).

Bon travail :-) !