**Équipe ICPS** **Sujet de TER**
**Sujet Compilation - Génération de code de haut niveau** **2014–2015**

# Génération de « capsules » pour noyaux de calcul

| | |
|---|---|
| **Lieu** | Équipe ICPS, ICube (UMR CNRS 7357) |
| **Encadrant** | Cédric Bastoul (cedric.bastoul@unistra.fr) |

## Équipe d'accueil

L'équipe Informatique et Calcul Parallèle Scientifique (ICPS) appartient au laboratoire ICube de l'université de Strasbourg. La préoccupation principale de cette équipe est de fournir les moyens théoriques et logiciels aux développeurs d'applications pour optimiser leurs programmes ou permettre leur déploiement sur des architectures à large échelle ou dans le cloud. Formée de 9 membres permanents, d'une dizaine de doctorants, post-doctorants ou ingénieurs, ses membres sont fortement impliqués dans des collaborations européennes et internationales sur des projets scientifiques, des projets de transfert technologique et des plateformes logicielles libres, incluant les compilateurs GCC et LLVM, la bibliothèque de calcul PolyLib ou le générateur de code de haut niveau CLooG. Durant ce TER, vous ferez partie de cette équipe et vous travaillerez avec les chercheurs impliqués dans ces problématiques.

## Sujet

L'équipe ICPS participe activement au développement de compilateurs source-à-source tels que PoCC [*] ou Pluto [†]. Ces compilateurs sont capables de transformer de manière très agressive certaines parties du code appelées « noyaux de calcul » pour les paralléliser et les optimiser automatiquement. Les noyaux de calcul sont des morceaux critiques de code en général assez courts où une application passe beaucoup de temps et qu'il est donc intéressant d'optimiser. Les noyaux de calcul sont souvent extraits des applications pour être optimisés de manière indépendante (par exemple lorsqu'un industriel souhaite sous-traiter leur optimisation sans fournir l'application complète). Il faut alors construire un programme autour du noyau de calcul pour simuler son exécution.

Il s'agit dans ce TER d'automatiser la construction de code autour des noyaux de calcul isolés, on appellera cela le code « capsule ». Un exemple de noyau isolé est présenté en Figure 1. Il faudra dans ce travail analyser les noyaux de calcul (quelles sont les données accédées, quelle est leur taille, sont-elles des entrées au noyau, des sorties, etc.) et générer automatiquement un code capable de les exécuter tel que celui présenté en Figure 2 pour le noyau en Figure 1. Le travail d'analyse sera largement assisté par un ensemble de fonctions existantes qui seront présentées à l'étudiant qui choisira ce TER. Il s'agit en fait exactement des mêmes techniques et analyses qui composent les moteurs d'optimisation des outils de compilation avancée. Ce travail a pour but de faire partie intégrante d'un compilateur source-à-source.

```
#pragma scop
for (i = 0; i < 2*N - 1; i++)
  z[i] = 0;

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    z[i+j] += x[i] * y[j];
#pragma endscop
```

Figure 1 – Noyau de calcul d'une multiplication de polynômes

L'étudiant choisissant ce sujet devra être intéressé par les techniques de compilation avancées. Des compétences raisonnables en programmation C et en environnement Unix/Linux sont requises. Si le TER débouche sur un travail de qualité, celui-ci sera intégré au compilateur PoCC, ce qui devra être une source supplémentaire de motivation !

∗. http://pocc.sf.net
†. http://pluto-compiler.sf.net

```c
#include <stdio.h>
#include <stdlib.h>
#define PARAM1 4

/* Computation kernel */
void kernel(float* z, float* x, float* y, int N) {
  int i, j;
#pragma scop
  for (i = 0; i < 2*N - 1; i++)
    z[i] = 0;

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      z[i+j] += x[i] * y[j];
#pragma endscop
}

/* Kernel array allocation functions */
void alloc_z(float** z, int N) {
  *z = (float*)malloc((2*N) * sizeof(float));
}

void alloc_x(float** x, int N) {
  *x = (float*)malloc((N) * sizeof(float));
}

void alloc_y(float** y, int N) {
  *y = (float*)malloc((N) * sizeof(float));
}

/* Initialization of kernel input functions */
void input_x(float* x, int N) {
  int i;
  for (i = 0; i < N; i++)
    x[i] = rand()%5;
}

void input_y(float* y, int N) {
  int i;
  for (i = 0; i < N; i++)
    y[i] = rand()%5;
}

/* Print kernel output functions */
void output_z(float* z, int N) {
  int i;
  for (i = 0; i < 2*N - 2; i++)
    printf("%6.3f ", z[i]);
  printf("\n");
}

int main() {
  float* z, *x, *y;
  srand(42);

  // Memory space allocation for the arrays
  alloc_z(&z, PARAM1);
  alloc_x(&x, PARAM1);
  alloc_y(&y, PARAM1);

  // Initialization of input data for the kernel
  input_x(x, PARAM1);
  input_y(y, PARAM1);

  // Launch the kernel
  kernel(z, x, y, PARAM1);

  // Print the output values
  output_z(z, PARAM1);
}
```

FIGURE 2 – Code complet (capsule + noyau) pour la multiplication de polynômes

## Référence clé (synthèse et critique de l'UE Initiation Recherche)

La référence clé à étudier et à présenter en UE Initiation Recherche est l'article [1].

## Références

[1] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In Proc. of the 2008 ACM Conf. on Programming language design and implementation (PLDI'08), Tucson, AZ, USA, June 2008.