Résumé des interventions de St-Nabor numéro A/311/CRI

Fabien Coelho
CRI, École des mines de Paris
35, rue Saint-Honoré,
77305 Fontaibleau cedex, FRANCE.
mél : **coelho@cri.ensmp.fr**

18 novembre 1999

1 Génération de code faible consommation pour *stencils*

1.1 Contexte

Le travail présenté a été effectué pendant que l'auteur était visiteur dans l'équipe de Francky Catthoor à l'IMEC (Leuven, Belgique). Voir entre autre le projet Acropolis :

`http://www.imec.be/vsdm/projects/mm_comp/`

L'IMEC a développé une méthodologie de minimisation de la consommation et/ou de la surface de silicium des applications multimédia manipulant beaucoup de données. Cette méthodologie, la DTSE (*Data Transfer and Storage Exploration*) guide l'ingénieur pour identifier et transformer les codes sources en réduisant la consommation liée aux transferts entre les différents niveaux de mémoire. Elle décompose le problème en sous-problèmes, et propose des algorithmes ou des heuristiques pour les résoudre.

1.2 Transformations de boucles pour les stencils

Nous nous sommes particulièrement intéressé à la méthodologie DTSE dans le cas particulier et simplifié des séquences de stencils, qui calculent une donnée dans un tableau en fonction d'un motif pris dans un tableau précédent.

Nous avons étudié pour différentes transformations de boucles l'impact sur les quantités de mémoires nécessaires, et le nombre de données transférées entre les différents niveaux de la hiérarchie mémoire.

Nous nous sommes principalement intéressés à l'impact des transformations sur le nombre de registres, en distinguant les registres vivants d'une itérations sur l'autre et ceux qui sont temporaires, sur la quantité de mémoire cache et principale, et enfin sur la taille du code, en prenant un exemple de *stencil* très simple.

fusion de boucle : elle dispense du stockage intermédiare d'un tableau de donnée, d'ou un gain important en quantité de mémoire principale et en nombre de données transférées. Par contre les besoins en mémoire cache et en registres pour le calcul d'une itération sont cummulés.

déroulage de boucle interne : le nombre de registre temporaire nécessaire est augmenté. Il y a peu d'impact sur les accès. Le longueur du code est multipliée par le facteur de déroulage. En définitive, cette transformation ne présente pas d'intérêt pour la DTSE. Elle peut en présenter pour d'autres transformations ou optimisations, comme par exemple dériver un pipeline logiciel.

déroulage de boucle externe (*unroll-and-jam*) : le nombre de registres temporaires et vivants, ainsi que la taille du code, sont augmentés de manière proportionnelle au déroulage. Par contre, les accès au cache sont réduits d'autant plus que la boucle est déroulée. Cette transformation est donc profitable du point de vue des transferts mémoires, et donc de la consommation.

tuilage interne : cette transformation permet de réduire la quantité de mémoire cache de premier niveau nécessaire, en ajoutant une mémoire cache de second niveau et des accès à celle-ci. Une telle transformation est utile si les besoins en cache de premier niveau sont trop important. La taille de code est légèrement augmentée par la transformation.

Partant de ces constatations pour la plupart classiques, nous avons commencé le développement d'un outil de génération automatique de code optimisé (OSAG : Optimized Stencil Application Generator) pour appliquer ces transformations. En effet, leur application manuelle est difficile et longue. Cette outil écrit en JAVA part de la géométrie des *stencils* et de la spécification des transformations souhaitées, pour en inférer les nombres de registres, volumes de caches et de mémoire ainsi que les transferts nécessaires, et pour produire un code optimisé.

1.3 Conclusion

Il est à noter que le génerateur de code aborde le problème de manière opposée à une compilation classique à base de transformations élémentaires et incrémentales : les décisions sont prises en descendant les niveaux mémoire (registres, caches, mémoire principale), et non en les remontant.

L'implémentation de ces transformations dans un compilateur est certes possible, mais n'est pas très commode : il faut beaucoup de phases et d'analyses pour *nettoyer* le code après chaque transformation (élimination de code mort, analyse de redondance, ré-ordonnancement des instructions...), qui doivent être implémentées dans un cas assez général. Pour le cas particulier et simple qui nous intéresse, il semble donc plus facile de faire un outil spécifique et indépendant, qui vise à générer un code directement optimisé en suivant un patron (*template*).

2 Optimisations de code à l'aide de propriétés algébriques

Nous avons présenté les travaux de thèse de Julien Zory. Dans sa thèse, qui doit être soutenue le **17 décembre 1999 à l'École des mines de Paris**, les propriétés algébriques (commutativité, associativité...) des opérateurs sont utilisées pour transformer les expressions de manière à augmenter la vitesse de leurs évaluations. Ces techniques peuvent s'appliquer à l'optimisation de codes scientifiques, au codes de contrôle (typiquement des codes d'enumération de points entiers dans un polyèdre), ou encore aux calculs d'adresses. Les travaux présentés concernent le niveau interne du processeur (ILP) et les boucles.

Voir la page de Julien Zory : `http://www.cri.ensmp.fr/~zory`

2.1 Parallélisme interne des processeurs

Les processeurs modernes disposent d'un parallélisme interne, qui permettent à plusieurs opérations d'avoir lieu en même temps. Certaines transformations algébriques permettent d'augmenter le parallélisme offert par l'application. Ces problèmes sont très combinatoires et une approche en plusieurs phases a été proposée pour traiter les expressions :

- normalisation, avec des techniques de réécriture ;
- heuristique gloutonne de factorisation ;
- heuristique gloutonne d'extraction des multiply-add ;
- algorithme d'équilibrage *à la Huffman*.

Pour les deux heuristiques, il est proposé d'optimiser la gravité de l'expression, *i.e.* la profondeur moyenne des opérations dans l'arbre d'expression. Ce critère est un bon compromis entre le coût des opérations et le chemin critique de l'expression.

2.2 Code de contrôle

Ces mêmes propriétés peuvent être exploitées pour optimiser le code de contrôle et le coût d'exécution des boucles. En particulier, de nouvelles opportunités de déplacement de calculs invariants et de détection des expressions communes sont dégagées. Nous présentons également une technique de déplacement d'instructions invariantes basées sur une réduction du graphe de dépendances.

2.3 Conclusion

Les résultats obtenus sont très prometteurs, avec des gains allant typiquement de 1,5 à 3. Les critères optimisés changent selon les processeurs ciblés, ce qui rend nécessaire l'application de ces transformations au niveau des compilateurs.