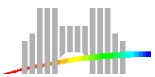
Compilation Avancée et Optimisation de Programmes

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr

16 octobre 2006



Pourquoi la Compilation

- Il est très fréquent d'avoir à écrire un petit compilateur spécialisé (Domain Specific Language, DSL). Exemples : XML, Interfaces homme-machine, DSL pour écrire des pilotes de périphériques, etc.
- Le domaine est très actif parce que le matériel et les applications évoluent très vite et que les compilateurs doivent suivre.

Evolution des processeurs

- Hiérachie de caches (parce que la vitesse des processeurs augmente plus vite que celle des mémoires).
- Mémoire "scratchpad", parce que les caches sont imprévisibles.
- Parallélisme, parce qu'il est plus facile d'augmenter la taille du chip que la vitesse d'horloge (ralentissement de la loi de Moore).
- Spéculation, parce qu'il faut bien utiliser tous ces transistors.
- Processeurs spécialisés, parce que les processeurs généralistes sont peu efficaces (exemple : processeurs pour le traitement du signal, pour la cryptographie).

Evolution des Applications

- Plus de puissance $\Rightarrow$ applications plus complexes.
- Course à la performance (simulation physique, mathématique, sociale).
- Généralisation des systèmes embarqués : ordinateurs pas ou peu programmables, cachés à l'intérieur d'un produit grand public :
  - minimiser la taille du programme et des données.
  - minimiser la consommation électrique (durée de vie des batteries).
  - algorithmes figés, donc possibilité d'utiliser des accélérateurs matériels.

Monoprocesseurs

- Un seul processeur, connecté à plusieurs caches en cascade et à une grande mémoire : problèmes d'amélioration de la localité.
- Processeurs spécialisés (DSP), mémoire scratchpad.
- Parallélisme caché : exploitation ?
  - Pipeline d'instructions, le problème des branchements.
  - Processeurs "superscalaires".
- Parallélisme apparent :
  - VLIW, parcels.
  - Prédication et spéculation.
  - Ordinateurs vectoriels.

Multiprocesseurs, I

Il y en a essentiellement quatres modèles qui peuvent s'hybrider.

- *Multithread* Il s'agit en fait d'un processeur unique utilisé en temps partagé rapide : il est possible de changer de processus en temps négligeable, par exemple à chaque défaut de cache.
- *Symetric Multiprocessor, SMP* Plusieurs processeurs connectés à une mémoire unique par l'intermédiaire d'un système de caches plus ou moins complexes.
  - Problème de la cohérence entre caches.
  - Programmation : diviser le problème en plusieurs tâches aussi indépendantes que possibles, utiliser des instructions de synchronisation.

Multiprocesseurs, II

- *Single Instruction, Multiple Data, SIMD* Plusieurs processeurs appliquent la même instruction à des données différentes.
  - Chaque processeur possède sa propre mémoire.
  - Le fonctionnement est synchrone.
  - Il faut prévoir un réseau de communication et un ordinateur "hôte".
  - Applications très spécifiques (traitement d'image).

Multiprocesseurs, III

- Ordinateurs à mémoire distribuée. Processeurs complets interconnectés par un réseau. Exemple : un ensemble de PC connectés par Ethernet.
  - Performances limitées par le débit du réseau : réalisation de réseaux plus performants qu'Ethernet : Myrinet, Quadrics, SCI ...
  - Une tâche par processeur échangeant des messages avec les autres tâches. Assure à la fois les échanges de données et les synchronisation. Les échanges doivent être aussi rare que possible.
  - Evolution vers les Grilles de Calcul.
  - Il existe des bibliothèques d'échange de messages (PVM, MPI, logiciels pour la grille).

Circuits spécialisés, I

- Il arrive que l'on puisse intégrer toute une application sur un seul chip (exemple : un modem). Comme le nombre de transistors augmente, il est même possible de réaliser un système multiprocesseur sur un chip (SoC) parfois muni d'un réseau (NoC).
- Il est plus fréquent que l'on isole le "noyau" d'une application et que l'on réalise un accélérateur couplé plus ou moins étroitement à un processeur généraliste.
- Exemple bien connu : la carte graphique.
- Autres exemples : transformateur de Fourier, analyseur de séquences génétiques, etc.
- Ces circuits se connectent en général au bus PCI, qui est souvent le goulot d'étranglement de la configuration. (Une carte graphique se connecte sur un bus plus proche du processeur).

Circuits spécialisés, II

Deux grand types :

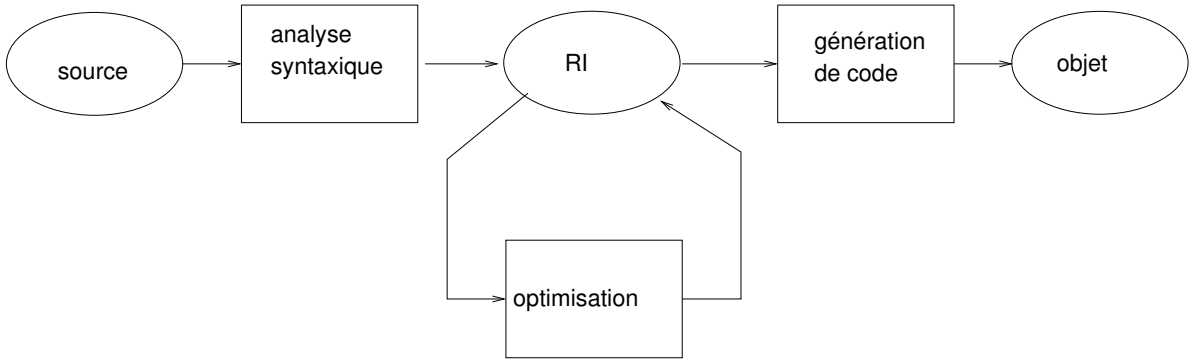
- Circuits classiques : structure analogue à celle d'un processeur :
  - Un *chemin de données* où les calculs sont effectués : registres, accès à la mémoire, opérateurs arithmétiques et logiques, et les connexions nécessaires.
  - Un contrôleur, qui est un automate fini, qui assure l'enchaînement des opérations, configure le chemin de données et interprète les signaux d'état.
- Circuits systoliques : un grand nombre d'opérateurs identiques interconnectés régulièrement par une grille 1D ou 2D.
  - Les opérateurs peuvent contenir des données propres et recevoir des signaux de contrôle.
  - Les applications doivent être très régulières.
  - Avantage : il existe des méthodes de synthèse automatique.
  - Exemple : le filtrage.

Circuits spécialisés, III

Quelques problèmes :

- La taille du circuit : le prix du circuit augmente avec sa taille. Ajouter de la mémoire ou augmenter le parallélisme fait augmenter la taille, mais améliore les performance. Il y a un compromis à trouver.
- La prédictabilité : beaucoup de circuits spécialisés travaille en temps réél – téléphonie, télévision, pilotage, etc. Certains dispositifs ont des comportements difficiles à prédire (caches, réseaux).
  - Remplacer les caches par des mémoires "scratchpad".
  - Utiliser des réseaux à garantie de service, ou ordonnancer les communications.
- la consommation électrique : parce que les système embarqués sont souvents autonomes. Le parallélisme réduit la consommation, la localité réduit la consommation, etc.

Organisation du Cours



- On s'intéressera essentiellement aux optimisations, et un peu à la génération de code.
- Pour pouvoir optimiser, il faut savoir ce que fait le programme (en gros), et donc l'analyser.
- On traitera les optimisations "non standard" suivantes :
  - Recherche et exploitation du parallélisme.
  - Localité.
  - Optimisation pour la synthèse de circuits.

Conclusion provisoire

- Le progrès en compilation est indispensable pour tirer parti des processeurs modernes.
- C'est un sujet de recherche très actif (journaux, conférences, projets de recherche européens, surtout dans le domaine de l'embarqué).
- Il existe des débouchés dans l'industrie :
  - fabricants de processeurs (Intel, IBM, HP, ST-Micro).
  - éditeurs de logiciels (Microsoft, Borland, etc).
  - éditeurs de logiciels de CAO (Synopsis, Mentor Graphics, Cadence, etc.)
  - Concepteurs de circuits embarqués (Thalès, Philips, Nokia).
  - Recherche développement : CEA.

Une étude de cas : le produit matrice vecteur

Il existe de multiples façons d'écrire le produit matrice * vecteur. Par exemple :

```
for(i=0; i<n; i++)
  c[i] = 0.0;
for(j=0; j<n; j++)
  for(i=0; i<n; i++)
    c[i] += a[i][j]*b[j];
```

n	t (ms)
1000	25,3
2000	145
4000	1602

Observer que le temps est multiplié par plus que 4 quand n est multiplié par 2.

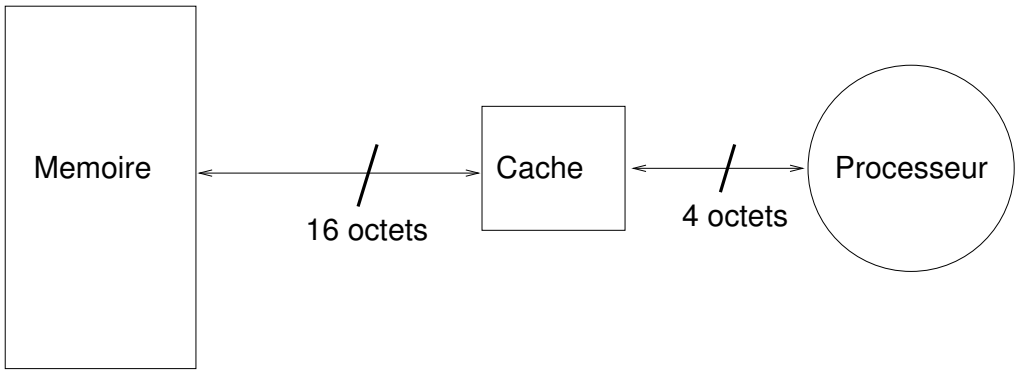
Une autre version

```
for(i=0; i<n; i++){
  s = 0.0;
  for(j=0; j<n; j++)
    s += a[i][j]*b[j];
  c[i] = s;
}
```

n	t (ms)	précédent
1000	14,6	25,3
2000	58,8	145
4000	232	1602

- D'où vient la différence ?
- Les deux programmes sont-ils équivalents ?

Influence du cache



- Dans la première version :

```
for(i=0; i<n; i++)
  c[i] += a[i][j]*b[j];
```

  les accès à la mémoire ne sont pas consécutifs ; on gaspille 3/4 des données lues.
- La deuxième version a de la *localité spatiale* :

```
for(j=0; j<n; j++)
  s += a[i][j]*b[j];
```

De quel droit ?

- On passe d'un code à l'autre par transformations successives :

```
for(j=0; j<n; j++)
  for(i=0; i<n; i++)
    c[i] = c[i] + a[i][j]*b[j];
```

```
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    c[i] = c[i] + a[i][j]*b[j];
```

- Dans les deux codes, la valeur de `c[i]` utilisée pour l'addition provient de l'itération $(i, j-1)$ et l'autre terme est invariant. La suite des valeurs de `c[i]` est donc la même. Il y a eu *échange de boucles*.
- Pour justifier l'introduction de `s`, on raisonne en sens inverse : on passe du deuxième code au code ci-dessus par une autre transformation, *l'expansion de scalaire*.

Et si le processeur est parallèle ?

- La première version a du parallélisme : les itérations de la boucle sur `i` son indépendantes. Mais il faut beaucoup de processeurs pour compenser son inefficacité.
- La deuxième n'en a pas, à cause du scalaire `s`.
- On peut rendre la première version efficace en transposant la matrice `a` :

```
for(i=0; i<n; i++)
  c[i] = 0.0;
for(j=0; j<n; j++)
  for(i=0; i<n; i++)
    c[i] += a[j][i]*b[j];
```

n	t (ms)	précédent
1000	14,5	14,6
2000	52,4	52,8
4000	205	232

- La transformation est légale à condition de la faire partout dans le code.
- Le résultat est un peu meilleur, sans doute à cause d'une meilleure utilisation du processeur.

Elimination de Gauss

Version naïve :

```
for(i=0; i<n; i++)
  for(j=i+1; j<n; j++)
    for(k=i+1; k<n; k++)
      a[j][k] -= a[j][i]*a[i][k]/a[i][i];
```

Il y a $O(n^3)$ additions, multiplications et divisions.

Optimisation I/II

- La quantité `a[i][i]` (le *pivot*) est invariante dans le boucle sur `j` et `k`.

  En effet, i est constant dans la boucle, mais surtout il est impossible que l'écriture de `a[j][k]` modifie `a[i][i]` car d'après les bornes des boucles, $j \leq i+1$ et $k \geq i+1$.

- Il en est de même pour `a[j][i]`.
- On peut donc hisser (*hoisting*) certains calculs hors des boucles :

Optimisation II/II

```
for(i=0; i<n; i++){
  p = 1.0/a[i][i];
  for(j=i+1; j<n; j++){
    q = a[j][i]*p;
    for(k=i+1; k<n; k++)
       a[j][k] -= q * a[i][k];
  }
}
```

- Il y a seulement $O(n)$ divisions.
- On reconnait dans la dernière ligne une opération vectorielle :

$$\vec{v} := \vec{v} - q\vec{w}.$$

- Si on prend la précaution d'expanser le scalaire `q` en `q[j]` on se rend compte que la boucle sur `j` est parallèle.
- Par une succession de transformations, on a adapté le code original pour un multiprocesseur vectoriel.

Conclusion provisoire II

- L'optimisation d'un programme se fait par succession de transformations.
- Avant d'appliquer une transformation, il faut savoir si elle est légale ; c'est l'analyse de programme qui permet de répondre à la question.
- L'information essentielle est la description du *flot des données* : quelle est la source de chaque valeur utilisée dans les calculs.
- Du flot des données on déduit les contraintes d'ordre d'exécution.
- Pour les architectures modernes, l'optimisation de la localité l'emporte sur tout le reste. On l'obtient par changement de l'ordre d'exécution, sous réserve des contraintes du flot de données.
- La reconnaissance de forme (opérations vectorielles, détection de récurrences) joue un role important.