Efficacité d'un programme séquentiel

L'objectif est de mettre en œuvre des techniques d'optimisation pour les programmes séquentiels.

1 Fusion de boucle

Observer le programme `boucles.c`. Quelle optimisation auriez-vous naturellement tendance à faire ? Le gain obtenu est-il décevant, correct, ou plus que satisfaisant ? Comment l'expliquer ?

Reproduire l'expérience en ajoutant une troisième puis une quatrième boucle, obtenez-vous une accélération encore meilleure ? Pourquoi ?

2 Déroulement de boucle

Par défaut on compilera avec l'option `-O2` qui optimise le code (placement de variables dans des registres,...) sans pour autant bouleverser le code (déroulement de boucles, suppression de code inutilie, utilisation d'instructions vectorielles).

Tout d'abord, observons le coût d'une boucle : le programme `deroulement.c` effectue un calcul tout bête au sein d'une boucle qui a un nombre d'itérations connu. Tel quel, chaque mesure prend quelques secondes.

1. Le compilateur `gcc` permet de glisser des directives de compilation au sein du code. La directive suivante demande de dérouler deux fois la boucle :

   `#pragma GCC unroll 2`

   Comparer les performances du programme en déroulant 2, 3 et 4 fois la boucle.
2. Compiler votre programme avec `-O3` et comparer le temps d'exécution.
3. Dérouler *à la main* la boucle, c'est-à-dire par exemple définir `D` à 2 pour faire 2 fois moins d'itérations, mais en répliquant le contenu de la boucle pour lui faire faire deux fois le calcul par itération. Attention aux indices de tableau : *vérifiez* que le résultat obtenu est bien le même. Observer les performances obtenues en déroulant 2 et 4 itérations.
4. Optimiser le cœur de la boucle en utilisant des variables supplémentaires :

   `sum0 += c[i] ; sum1 += c[i+1];`

   ou ne faisant qu'une seule affectation par tour :

   `sum += c[i] + c[i+1]` .

   Vérifier le résultat obtenu. Comparer les performances obtenues en déroulant plusieurs fois la boucle.
5. Compiler votre meilleure version avec l'option `-O3` puis `-O3 -march=native` et comparer les performances obtenues.

3 Prédiction de branchement

L'objectif du programme `prediction.c` est de mesurer la qualité du prédicteur de branchement d'un cœur. On sait que les prédicteurs employés dans nos ordinateurs sont capables de reconnaître si un branchement n'est jamais pris, ou bien s'il est pris une fois sur deux par exemple.

On cherche ici à déterminer expérimentalement la capacité du prédicteur à mémoriser une séquence de k décisions de branchement. Pour obtenir ces k décisions on s'appuie sur une séquence aléaoire de k tests en comptant combien de caractères sont supérieurs à 127 dans un tableau de k caractères générés pseudo-aléatoirement.

Une fois fixée la séquence de k tests on mesure ensuite la durée d'exécution de 10 000 tests et incrémentations. Comme il y a des séquences plus faciles à apprendre que d'autres, on itère ce procédé en changeant de séquence alélatoire.

1. Lire le code de la boucle la plus interne ;
2. Compiler et lancer le code qui affiche la taille de la séquence et le nombre de cycles pris par le cœur pour traiter 10 000 tests ;
3. Faire passer le nombre de mesures à 15 pour faire une courbe et l'analyser :

   ```
   ./prediction > /dev/null 2>> data
   Rscript tracer-prediction.r data
   ```

4. Reproduire l'expérience sur des machines d'autres salles ou sur des serveurs.

4 Branchements vs calculs

Voici un extrait du programme `evitons-les-sauts.c` :

```
for (i=0; i<N; i++) {
  if (c[i] == 0)
   sum0++;
  else if (c[i] == 1)
   sum1++;
}
sum2 = N-sum0-sum1;
```

Ce code compte le nombre de fois qu'apparaissent 0, 1 et 2 dans un tableau contenant exclusivement des 0, 1 et 2. Son exécution est en fait relativement lente, on peut aller bien plus vite. Comment ? En évitant les branchements qui cassent le pipeline !

1. Chercher différents moyens d'effectuer le même calcul sans utiliser de branchement (i.e. la construction `if` notamment). Penser à exploiter les propriétés des nombres 0, 1 et 2.
2. Comparer vos solutions à celles proposées dans le fichier source. Ajouter vos solutions.
3. Compiler et exécuter le programme aux différents niveaux d'optimisation (`-O1 -O2 -O3`).