Programmation Parallèle

Fiche 5 : Cache

Cette fiche ainsi que les fichiers à télécharger sont disponibles sur le site
https://gforgeron.gitlab.io/pap/

L'objectif est de mettre en œuvre des techniques d'optimisation pour éviter les défauts de cache inutiles. Pour simplifier les expériences nous allons supprimer les optimisations du compilateur en modifiant ainsi le makefile :

```
########## Config Section ##########

ENABLE_SDL          = 1
ENABLE_MONITORING   = 1
#ENABLE_VECTO        = 1

...

CFLAGS                  += -O0 -Wall -Wno-unused-function -march=native
```

Scrollup

Nous allons modifier le noyau `scrollup` du programme EasyPAP afin provoquer de nombreux défauts de cache et observer leurs impacts sur les performances.

1. Modifier le fichier `makefile` pour compiler en niveau d'optimisation `-O1`. Cela désactivera la vectorisation automatique et nous permettra de mieux distinguer l'influence du cache sur les performances du programme.
2. Produire une version `scrollup_compute_ji()` en dupliquant la fonction `compute_seq()` :

```
1 for (int i = 0; i < DIM; i++) {
2   int src = (i < DIM - 1) ? i + 1 : 0;
3   for (int j = 0; j < DIM; j++)
4     next_img (i, j) = cur_img (src, j);
5 }
```

et en inversant l'ordre des boucles en `i` et `j`, on pourra aussi éliminer le test en traitant à part le cas de la dernière ligne :

```
1 for (int j = 0; j < DIM; j++)
2      next_img (DIM - 1, j) = cur_img (0, j);
```

3. En utilisant l'image `shibuya.png`, vérifier visuellement le résultat. Comparer « visuellement » la fluidité de l'animation obtenues par les versions `seq` et `ji` du noyau `scrollup`.
4. Utiliser le script `plots/run-xp-scrollup.py` pour lancer une série d'expériences qui va faire varier la taille de l'image et le nombre d'itérations pour travailler à coût constant :

$$\text{itérations} = \frac{4096 \times 4096}{\text{DIM} \times \text{DIM}}$$

$$\text{volume de calcul} = \text{itérations} \times \text{DIM}^2 = \frac{4096 \times 4096}{\text{DIM}^2} \times \text{DIM}^2$$

5. Utiliser la commande suivante pour générer un élégant graphique :

```
./plots/easyplot.py  -y time -x size --yscale log --xscale log \
                     --delete iterations -if scrollup.csv
```

6. Interpréter ce graphique sachant qu'un pixel est codé sur 4 octets et que les tailles des caches de la machine peuvent être obtenues via la commande `lstopo`.

Multiplication de matrices

Le programme `mul_mat.c` effectue une multiplications de matrices de façon classique.

1. Modifier le code de `mulMat2` afin d'utiliser plus efficacement le cache du processeur. Le gain obtenu est-il décevant, correct ou plus que satisfaisant ?
2. Il est probable que quelques défauts de cache évitables subsistent. Les repérez-vous ? Quelle permutation des boucles sur i, j, k induit le plus petit nombre de défauts de cache ? Modifier votre code en conséquence.

Bonus Lorsque N est assez grand il est probable quelques défauts de cache évitables subsistent dans votre code. Supposons que le cache fasse 8 Mo pour quelle valeur de N apparaissent ces défauts de cache ? Quelle technique faudrait-il utiliser pour limiter ces défauts ?

Rotation d'une image

Nous allons étudier l'impact du cache sur les performances du noyau `rotation90` du programme EasyPAP.

1. Produire une version optimisée `rotation90_compute_tiled()` en utilisant la technique de pavage (tuilage). On s'inspirera du code de la fonction `mandel_compute_tiled` pour cela.
2. Comparer les performances obtenues.
3. Produire une version parallèle de chaque code (`omp` et `omp_tiled`), comparer les performances obtenues pour une image de taille `2048`.
4. On pourra optimiser la version parallèle en faisant en sorte qu'un pixel de l'image soit toujours traité par un même thread (en lecture et en écriture). Vous pouvez par exemple vous inspirer du tuilage illustré en Figure 1, page 3.

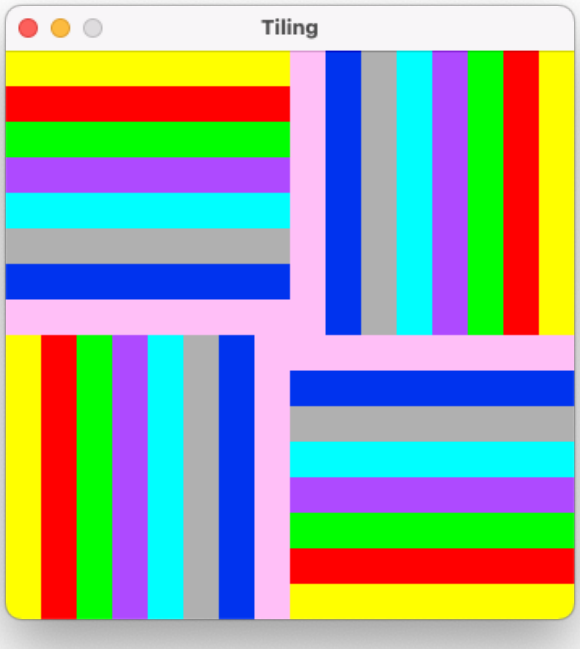

FIGURE 1 – Exemple de tuilage optimisant l'accès au cache pour la version parallèle du noyau `rotation90` avec `OMP_NUM_THREADS=8`.