Programmation Parallèle

Fiche 6 : Optimisation de code

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

Cette fiche porte sur l'optimisation de code séquentiel et la vectorisation. Nous allons pour cela travailler sur plusieurs noyaux fournis dans EASYPAP. Récupérez les dernières mises à jour à l'aide d'un discret `git pull`.

1 Optimisation du noyau `blur`

Le noyau de calcul `blur` floutte une image en calculant, pour chaque pixel, la valeur moyenne des pixels situés dans son voisinage immédiat. Cependant deux pixels différents peuvent avoir un nombre de voisins différent selon leur position sur l'image. Voici un code permettant de traiter tous les cas de façon uniforme.

```
int blur_do_tile_default (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++) {
      unsigned r = 0, g = 0, b = 0, a = 0, n = 0;

      int i_d = (i > 0) ? i - 1 : i;
      int i_f = (i < DIM - 1) ? i + 1 : i;
      int j_d = (j > 0) ? j - 1 : j;
      int j_f = (j < DIM - 1) ? j + 1 : j;

      for (int yloc = i_d; yloc <= i_f; yloc++)
        for (int xloc = j_d; xloc <= j_f; xloc++) {
          unsigned c = cur_img (yloc, xloc);
          ...
}
```

1. Un certain nombre de tests sont inutiles lorsque la tuile est éloignée du bord. Écrivez une variante de la fonction de tuilage qui ferait un traitement différencié pour les tuiles du bord et pour les tuiles du centre, afin de simplifier *au maximum* le calcul des tuiles internes en supprimant tous les tests inutiles.

   Voici à quoi pourrait ressembler le début de cette fonction :

   ```
   // Optimized implementation of tiles
   int blur_do_tile_opt (int x, int y, int width, int height)
   {
     // Outer tiles are computed the usual way
   ```

```
if (x == 0 || x == DIM - width || y == 0 || y == DIM - height)
  return blur_do_tile_default (x, y, width, height);

// Inner tiles involve no border test
...
```

2. Vérifiez le bon fonctionnement de votre programme en sauvegardant via l'option `--dump` les images calculées (on pourra travailler avec l'image `./images/shibuya.png`) et en les comparant (commande `diff`).
3. Observez le comportement de l'application en utilisant le *monitoring* et en appuyant sur la touche `h` pour activer le *heat mode* où l'intensité lumineuse de la tuile est proportionnelle à sa durée (relativement aux autres). Faites varier le grain.
4. Pour comparer les performances relatives de ces deux variables de tuilage, il est possible de générer des traces d'exécution et de les comparer. Par exemple :

```
# run regular version
./run -l images/shibuya.png -k blur -v tiled -wt default -nt 32 -i 10 -t -n \
      -lb "Regular 32x32"
# run optimized version
./run -l images/shibuya.png -k blur -v tiled -wt opt -nt 32 -i 10 -t -n \
      -lb "Optimized 32x32"
# compare last two traces
./view -c
```

5. Déterminer expérimentalement la meilleure taille de tuile. Cela nous donne-t-il pas des idées pour optimiser encore le programme ?
6. Programmez une version `blur_compute_omp_tiled`. Essayez d'équilibrer la charge au mieux, utilisez les traces pour vérifier la répartition des calculs.
7. On cherche maintenant à optimiser la réutilisation du cache. Par exemple deux images de taille 1920×1920 tiennent facilement dans l'ensemble des caches L3. Proposez une politique de distribution favorisant la réutilisation du cache.

2 Vectorisation du noyau `spin`

Le noyau spin, qui effectue quelques calculs trigonométriques simples, donne du fil à retordre au compilateur qui ne parvient pas à le vectoriser automatiquement. Nous allons donc effectuer manuellement la vectorisation des calculs en utilisant les *intrinsics*.

The interactive Intel Intrinsics Guide will undoubtly become your best friend during this assignment !

2.1 Début de vectorisation

Le fichier `spin-codelet.c` (fournit conjointement à cette fiche), contient des morceaux de code à copier-coller vers votre fichier `kernel/c/spin.c`. Ne déplacez pas `spin-codelet.c` dans l'arborescence EASYPAP.

Après avoir recompilé l'application, vous disposez dorénavant d'une variante de tuilage `avx` qui offre des performances sensiblement meilleures que la version séquentielle. Pour l'essayer :

```
./run -k spin -v tiled -wt avx
```

Examinez le code pour remarquer que la vectorisation est partielle à ce stade (cherchez notamment les `// FIXME`).

2.2 Arc tangente

Commençons par la fonction `_mm256_atan_ps` qui ne pose pas de souci particulier puisqu'elle ne contient pas d'instruction conditionnelle.

Remplacer le code séquentiel itératif par des instructions vectorielles pour calculer `AVX_VEC_SIZE_FLOAT` valeurs (c'est-à-dire `8`) d'arc tangente simultanément. Notez que vous aurez probablement besoin d'utiliser la fonction `_mm256_abs_ps` qui calcule les valeurs absolues des éléments d'un vecteur : regardez comment elle procède !

Vérifiez que votre code fonctionne en mode interactif, puis mesurez le temps gagné par rapport à l'ancienne version (option `-i 100 -n`).

2.3 Fonction `atan2`

Il reste maintenant à achever la vectorisation de la fonction `_mm256_atan2_ps`. Remarquez comment les premières instructions séquentielles ont été transformées en opérations vectorielles...

Remarquez que le test

```
if (y[i] < 0) th[i] = -th[i];
```

peut se lire « propager le bit de signe de `y` à `th`. »

Vérifiez visuellement les calculs sont toujours corrects, et appréciez le gain de performance obtenu !

3 Vectorisation du calcul de l'ensemble de Mandelbrot

Comme pour `spin`, un fichier `mandel-codelet.c` vous est fourni pour que vous puissiez copier-coller un début de solution vers votre fichier `kernel/c/mandel.c`.

3.1 Noir c'est noir

Bien que le code fournit compile, rien ne semble s'afficher à l'écran. Pour corriger cela, il faut insérer du code (`// FIXME 1`) pour sortir de la boucle lorsque les `8` suites ont toutes divergé. Cherchez dans le guide intrinsics une instruction permettant de vérifier qu'un masque ne contient que des zéros, et servez-vous en pour sortir de la boucle !

Désormais, la variante vectorielle devrait afficher des images :

```
./run -k mandel -v tiled -wt avx
```

3.2 Pixel Art

Toutefois, les images semblent pixellizées… En réalité, pour chaque vecteur horizontal de 8 pixels, les couleurs sont toujours identiques… et reflètent l'itération à laquelle la plus longue suite (parmi les 8 suites suivies en parallèle) a divergé.

Pour corriger ce problème (`// FIXME 2`), il faut faire en sorte que seuls les numéros d'itération correspondant aux suites n'ayant pas encore divergé soient incrémentés.

Une fois que le code marche, vous pouvez utiliser cette implémentation vectorielle au sein de votre variante OpenMP pour cumuler parallélisme intra- et extra-CPU. Quelle est l'accélération finale obtenue par rapport à l'implémentation séquentielle ?