Sorbonne Université

Polytech – EI-SE5

HPC – TP1 : Optimisations mono-cœur sur CPU

Dans ce TP nous allons nous focaliser sur l'écriture d'un code mono-cœur efficace. Pour y parvenir, nous allons appliquer des transformations de code source permettant de tirer parti de l'architecture des CPU modernes. L'optimisation sera focalisée sur un algorithme classique en traitement de l'image (application d'une matrice de convolution, filtrage 2D ou stencil 2D). Gardez à l'esprit que ces mêmes optimisations peuvent être transposées à d'autres domaines applicatifs.

1 Mise en bouche

1.1 Rappel sur les options d'optimisation du compilateur

Le compilateur permet de transformer le code source (que vous écrivez) en binaire « compréhensible » par la machine. Cette étape est nécessaire pour pouvoir exécuter votre code sur un CPU. En plus de traduire le code source en binaire, **le compilateur peut aussi optimiser le code pour réduire son temps d'exécution et sa consommation énergétique**. Pour cela le compilateur propose **des options d'optimisation**. Les plus connues commencent par `-O` et définissent un niveau d'optimisation global :

- `-O0` : pas d'optimisation,
- `-O1` : active une série d'optimisation dans le but de réduire la taille du binaire et son temps d'exécution tout en conservant un temps de compilation relativement faible,
- `-O2` : active toutes les optimisations possibles (exceptées les optimisations qui demandent un compromis entre efficacité et taille du binaire), cette option demande un temps de compilation plus long que `-O1`,
- `-O3` : actives toutes les optimisations possibles qui ne compromettent pas la précision des calculs, le temps de compilation est accru par rapport à `-O2`.

Ces différents niveaux d'optimisation sont valables pour les compilateurs GNU (`gcc`) et Clang. Cela peut varier en fonction du compilateur que vous utilisez, il est donc à la charge du développeur de bien connaître son compilateur. Pour plus de détails sur les différentes optimisations du compilateur vous pouvez vous rendre ici : `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

1.2 Le noyau de calcul `blur`

Dans ce TP nous allons utiliser EasyPAP et plus particulièrement nous allons travailler sur le noyau de calcul (ou *kernel*) `blur`. Ce noyau de calcul a pour effet de flouter une image. C'est un algorithme que l'on retrouve souvent dans les logiciels de traitement de l'image comme Photoshop. La Figure 1 illustre le traitement que l'on va appliquer dans la suite du TP.
Vous pouvez télécharger une banque d'images compatible avec EasyPAP ici : `http://raymond.namyst.emi.u-bordeaux.fr/no_link/EasyPAP/images.zip`. Vous retrouverez alors vous aussi les joies du canoë-kayak nous rappelant à tous à quel point les vacances sont douces.
Le Code 1 présente l'implémentation actuelle de l'algorithme `blur`. Comme vous pouvez le voir, la fonction `blur_compute_seq` est appelée en premier (c'est le point d'entrée du *kernel* EasyPAP), puis la fonction `blur_do_tile_default` est appelée à chaque itération.
Vous pouvez dès à présent lancer le noyau de calcul dans EasyPAP :

```
./bin/easypap -k blur --load-image images/1024.png -i 100 -v seq -wt default
```

Cela présuppose que vous ayez extrait le ficher `images.zip` à la racine d'EasyPAP. La commande précédente permet de lancer 100 itérations du *kernel* `blur`. De plus, nous avons spécifié le type de tuile avec l'option `-wt` (ou `--with-tile` en version longue). Cela permet de remplacer automatique l'appel de fonction `do_tile` ligne 3 dans le Code 1 par la fonction nommée `blur_do_tile_default` ligne 9 dans le même Code 1. Notez que c'est EasyPAP qui fait cela pour vous.

(a) Image originale.

(b) Image floue après 20 itérations.

FIGURE 1 – Illustration du noyau de calcul `blur` sur une image.

```
unsigned blur_compute_seq (unsigned nb_iter) {
  for (unsigned it = 1; it <= nb_iter; it++) {
    do_tile (0, 0, DIM, DIM, 0);
    swap_images ();
  }
  return 0;
}

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);
          r += extract_red (c);
          g += extract_green (c);
          b += extract_blue (c);
          a += extract_alpha (c);
          n += 1;
        }
      r /= n;
      g /= n;
      b /= n;
      a /= n;
      next_img (i, j) = rgba (r, g, b, a);
    }
  return 0;
}
```

Code source 1 – Code du noyau de calcul `blur` dans EASYPAP.

Dans ce TP nous allons écrire plusieurs variantes de la fonction `blur_do_tile_default`. Pour cela vous serez invités à copier coller cette même fonction et à remplacer la fin du nom (ici `default`) par le nouveau nom. Vous pourrez ensuite comparer les temps d'exécution des différentes versions en utilisant le paramètre `-wt` d'EasyPAP.
Le traitement effectué par le *kernel* `blur` est une moyenne des 8 pixels qui entourent un pixel (le pixel entouré est compris dans la moyenne). Cela a pour effet de diffuser la couleur avec un voisinage 3x3 et rend un effet de flou. Ce même traitement est appliqué à chaque pixel de l'image et le nouveau pixel calculé est stocké dans l'image suivante. Plus on itère, plus l'image devient floue.
Les différentes composantes d'un pixel sont le rouge, le vert, le bleu et le canal alpha (= la transparence). Chacune de ces composantes est stockée sur 8 bits. Toutes ces composantes sont regroupées dans un même entier 32-bit. Les fonctions `extract_red`, `extract_green`, `extract_blue`, `extract_alpha` et `rgba` vous permettent de manipuler simplement les différentes composantes d'un pixel.

Travail à faire Lancer le *kernel* `blur` sur votre machine, comprendre le code.

1.3 Temps d'exécution en fonction des options d'optimisation

La compilation du code EasyPAP est définie dans le ficher `Makefile` à la racine. Les options de compilations sont définies à la ligne 57 :

```
CFLAGS += -O3 -march=native -Wall -Wno-unused-function
```

Comme vous pouvez le voir l'option `-O3` est présente, c'est l'option d'optimisation de niveau 3.

Travail à faire En relisant la Section 1.1, compiler le code avec les différents niveaux d'optimisation disponibles puis comparer et noter les différents temps d'exécution pour 100 itérations. Pour voir le temps d'exécution il faut utiliser l'option `--no-display` d'EasyPAP.

Pour la suite du TP, il faut utiliser le niveau d'optimisation « zéro » (`-O0` = pas d'optimisation du compilateur), cela permet de mieux voir l'impact des différentes transformations de code que nous allons effectuer. À la fin, ET SEULEMENT À LA FIN vous pourrez comparer les temps d'exécution avec des niveaux d'optimisation plus élevés.

2 Modifier le code pour diminuer le temps d'exécution

2.1 Version du *kernel* `blur` sans gestion des bords (`default_nb`)

Dans le Code 1, les pixels aux bords de l'image font l'objet d'un traitement différent. En effet, aux bords de l'image il n'est pas possible de faire une moyenne sur 9 pixels car certains pixels n'existent tout simplement pas. Ce traitement spécifique des pixels aux bords est assuré par les lignes 13-16 ou l'on calcule des limites.

Travail à faire Afin de simplifier les transformations à venir, vous devez écrire une version `default_nb` (`nb` pour *no border*) dans laquelle il y aura un traitement plus simple pour les bordures : une simple copie du pixel précédent. Cela va simplifier le corps de la boucle de calcul (entre L10 et L31) et les lignes 13-16 doivent disparaître. Testez votre code avec EasyPAP, vous devez clairement voir que les bords ne changent pas et il ne doit pas y avoir de clignotement.

2.2 Dérouler sur l'axe des abscisses (`optim1`)

Travail à faire En partant de la version `default_nb` écrite précédemment, écrire une nouvelle version du code (que vous nommerez `optim1`) qui déroule manuellement la boucle ligne 18. En d'autres termes, la boucle L18 doit disparaître. Comparez les temps d'exécution entre les versions `default_nb` et `optim1`. Que constatez-vous ? Comment l'expliquez-vous ?

2.3 Dérouler sur l'axe des ordonnées (`optim2`)

Travail à faire En partant de la version `optim1` écrite précédemment, écrire une nouvelle version du code (que vous nommerez `optim2`) qui déroule manuellement la boucle ligne 17. En d'autres termes, la boucle L17 doit disparaître. Comparez les temps d'exécution entre les versions `optim1` et `optim2`. Que constatez-vous ? Comment l'expliquez-vous ?

2.4 Inliner les appels de fonction (`optim3`)

Vous devez maintenant avoir un code avec :

- 9 appels à la fonction `extract_red`,
- 9 appels à la fonction `extract_green`,
- 9 appels à la fonction `extract_blue`,
- 9 appels à la fonction `extract_alpha`,
- 1 appel à la fonction `rgba`.

Soit 37 appels de fonction dans le corps de la boucle la plus profonde. Ces fonctions sont définies dans le fichier `inlude/img_data.h`. Vous remarquerez que ces fonctions sont définies dans un ficher d'en-têtes et qu' elles sont précédées du mot clef `inline`. En temps normal, lors d'un appel à une de ces fonctions, le compilateur est capable de remplacer l'appel par le code de la fonction (*inlining*). Cependant nous sommes en `-O0` et donc le compilateur n'applique aucune optimisation.

Travail à faire En partant de la version `optim2` écrite précédemment, écrire une nouvelle version du code (que vous nommerez `optim3`) qui remplace les appels des fonctions énoncées précédemment par leur code. Comparez les temps d'exécution entre les versions `optim2` et `optim3`. Que constatez-vous ? Comment l'expliquez-vous ?

2.5 Rotation de variables (`optim4`)

Certains accès mémoires sont redondants d'un pixel à l'autre. Lors du parcours des pixels on avance en suivant l'axe des abscisses d'abord. Dans ce cas, on refait les accès à la colonne de droite et à la colonne du milieu (dans la matrice 3x3 qui représente notre stencil).

Travail à faire En partant de la version `optim3` écrite précédemment, écrire une nouvelle version du code (que vous nommerez `optim4`) où vous effectuerez une rotation de variables sur l'axe des abscisses. Comparez les temps d'exécution entre les versions `optim3` et `optim4`. Que constatez-vous ? Comment l'expliquez-vous ?

2.6 Gestion des bords (`optim5`)

Travail à faire En partant de la version `optim4` écrite précédemment, écrire une nouvelle version du code (que vous nommerez `optim5`) où vous remettrez la gestion des bords comme dans la version initiale du code (fonction `blur_do_tile_default`). Pour cela, ne modifiez pas votre boucle principale, remplacez juste la copie de pixels des bords par le code initial de gestion des bords. Vous pouvez copier coller du code. Comparez les temps d'exécution entre les versions `optim4` et `optim5`. Que constatez-vous ? Comment l'expliquez-vous ?

2.7 Compiler et comparer avec des options d'optimisation

Travail à faire Pour terminer le TP, vous devez maintenant activer les différents niveaux d'optimisation du compilateur (`-O1`, `-O2` et `-O3`). Comparez les temps d'exécution des différentes versions optimisées en fonction du niveau d'optimisation. Est-ce que le code optimisé est plus rapide ? Si oui, de combien par rapport à la version initiale du code ?