Accélération de calculs 2D avec des Threads

On s'intéresse à des traitements qui manipulent des images existantes, ou qui en créent des nouvelles. On dispose d'une application qui permet d'afficher des images, de lancer des calculs dessus, et de visualiser les évolutions de l'image à chaque itération. Téléchargez

```
git clone https://gitlab.com/gforgeron/easypap-se.git
cd easypap-se/
ln -s /net/cremi/rnamyst/etudiants/easypap/images
make
```

Pour lancer l'application avec une image d'exemple (un appui sur la touche `ESC` permet de quitter l'application) :

```
./run -l images/shibuya.png
```

Dans le cas présent, le traitement (que l'on appellera également *noyau*) par défaut s'appelle `none` et consiste à ne rien faire. Par défaut, c'est la variante séquentielle (`seq`) qui est invoquée.
On peut spécifier le noyau à invoquer à l'aide de l'option `-k`, et la variante à l'aide de l'option `-v`. Ainsi, on aurait pu lancer cette première exécution comme ceci :

```
./run -l images/shibuya.png -k none -v seq
```

Les fonctions implémentant les différentes variantes d'un noyau se trouvent généralement dans le fichier `kernel/c/<noyau>.c`. Regardez le contenu du fichier `kernel/c/none.c`, et remarquez que la fonction implémentant la variante séquentielle s'appelle `"none_compute_seq"`.

Exercice 6.1 On s'intéresse au noyau `invert` qui calcule le négatif d'une image couleur, c'est-à-dire qui remplace chaque pixel par un pixel résultant du complément binaire de chacune de ses composantes couleur Rouge, Vert et Bleu.
Une version séquentielle du code vous est fournie dans le fichier `kernel/c/invert.c`. Regardez comment chaque pixel de l'image est parcouru, à l'aide des deux boucles imbriquées. La variable globale `DIM` indique à la fois la largeur et la hauteur de l'image (qui est toujours carrée). L'accès à un pixel à la i^{eme} ligne et à la j^{eme} colonne s'effectue au travers de l'expression `cur_img(i, j)`.
Si on lance le noyau sans option particulière, l'image va clignoter de manière très rapide puisqu'à chaque itération paire l'image retrouve son état d'origine :

```
./run -l images/shibuya.png -k invert
```

Pour avoir le temps d'observer chaque image, il est possible de demander une pause à chaque itération :

```
./run -l images/shibuya.png -k invert -p
```

Enfin, pour mesurer précisément les performances du noyau `invert` sans le ralentissement dû à l'affichage, il faut utiliser l'option `-n` (*no display*) et `-i <nb>` (nombre d'itérations souhaitées) :

```
./run -l images/shibuya.png -k invert -n -i 1000
```

Notez le temps d'exécution total affiché.
Il s'agit désormais d'élaborer une version parallèle du noyau `invert` dans laquelle plusieurs threads calculeront chacun une bande horizontale différente de l'image. Pour déterminer le nombre de threads à lancer on utilisera la fonction `easypap_requested_number_of_threads()`.

1. Écrivez une fonction `invert_compute_thread(unsigned nb_iter)` qui pour chaque itération crée des threads, chaque thread exécutant une fonction `thread_fun(void *id)` (à écrire également) qui affiche le numéro du thread l'exécutant. N'oubliez pas d'attendre la terminaison des threads à la fin de la boucle traitant les itérations au moyen de `pthread_join()`. Après avoir compilé, testez ainsi votre code :

   ```
   ./run -l images/shibuya.png -k invert -v thread -i 1

   ./run -l images/shibuya.png -k invert -v thread -i 3
   ```

   Sachant que le nombre de threads à employer peut être précisé par la variable `OMP_NUM_THREADS`, vérifiez que le programme crée bien le nombre de threads demandés ainsi :

   ```
   OMP_NUM_THREADS=8 ./run -l images/shibuya.png -k invert -v thread -i 1
   ```

2. Il faut maintenant que chaque thread calcule le numéro de la première et de la dernière ligne de l'image qui délimitent la bande sur laquelle il va travailler. Chaque thread travaillera sur une bande d'épaisseur $\frac{DIM}{\#threads}$, sauf le dernier qui aura une bande un peu plus épaisse si la division ne tombe pas juste. Faites en sorte que chaque thread affiche sa ligne de départ et le nombre de lignes qui lui revient.
3. Lorsque tout est correct, utilisez la fonction `do_tile()` pour implémenter la version définitive. Enlevez les `printf`, et vérifiez visuellement :

   ```
   ./run -l images/shibuya.png -k invert -v thread -i 1
   ```

4. Enfin, comparez les performances avec la version séquentielle :

   ```
   ./run -l images/shibuya.png -k invert -v thread -n -i 1000
   ```

   Que pensez vous de l'accélération obtenue ($\frac{temps\ sequentiel}{temps\ parallele}$) ?

Exercice 6.2 Le noyau `invert` introduit très peu de calcul par pixel, et ses performances sont davantage liées à la rapidité de la mémoire qu'à celle du processeur. Il est donc difficile d'obtenir des accélérations flatteuses.
On devrait avoir davantage de chances avec le noyau `blur`, qui calcul un flou sur l'image en appliquant à un pixel la moyenne des pixels voisins. Ouvrez le fichier `kernel/c/blur.c` et regardez comment procède la fonction `blur_compute_seq` pour calculer la nouvelle valeur de chaque pixel. Notez que le programme utilise cette fois deux images : on lit les valeurs des pixels (provenant de l'itération précédente) avec `cur_img(i,j)` et on écrit la nouvelle valeur dans `next_img(i,j)`. Une fois que tous les pixels sont calculés, on inverse le rôle des deux images en appelant `swap_images()`, puis on peut passer à l'itération suivante...
Essayez :

```
./run -l images/shibuya.png -k blur
```

Comme pour le noyau `invert`, il s'agit de développer une version multi-thread.

1. Copiez-collez votre fonction `invert_compute_thread` ainsi que les fonctions annexes et variables globales dans le fichier `kernel/c/blur.c`. Renommez la fonction principale `blur_compute_thread`, adapter cette fonction introduisant l'appel à `swap_images()`.
2. Faites en sorte que chaque thread calcule le flou sur sa bande d'image.
3. Vérifiez que votre version fonctionne, puis évaluez les performances :

```
./run -l images/shibuya.png -k blur -v thread -n -i 1000
```

Calculez l'accélération obtenue par rapport à la version séquentielle.

4. Pour les étudiants bien avancés, il s'agit de créer les threads une seule fois par appel à la fonction `blur_compute_thread`. Pour ce faire, dupliquer les fonctions `thread_fun` et `blur_compute_thread` et les renommer `bar_fun` et `blur_compute_bar`. Déplacez la boucle portant sur les itérations de `blur_compute_bar` dans `bar_fun` puis adaptez le code des deux fonctions. Attention, il faut cette fois une barrière de synchronisation à la fin de chaque itération **avant** l'appel à `swap_images()`. Un thread et un seul doit appeler `swap_images()`. Ensuite, les threads doivent à nouveau se resynchroniser avant d'attaquer l'itération suivante.

Exercice 6.3 Le noyau `mandel` affiche une représentation graphique de l'ensemble de Mandelbrot (`https://fr.wikipedia.org/wiki/Ensemble_de_Mandelbrot`) qui est une fractale définie comme l'ensemble des points du plan complexe pour lesquels les termes d'une suite ont un module borné par 2.
À chaque itération, le programme affiche une image dont les pixels ont une couleur qui dépend de la convergence de lq suite associée au point complexe du plan. Entre chaque itération, un zoom est appliqué afin de changer légèrement de point de vue.
Vous pouvez lancer le programme de la façon suivante :

```
./run -s 512 -k mandel
```

L'option `-s 512` spécifie que l'on souhaite obtenir une image de 512×512 pixels.
Sans surprise, la version séquentielle `mandel_compute_seq` se trouve dans le fichier `kernel/c/mandel.c`.

1. En vous inspirant de votre travail sur le noyau `blur`, implémentez une version « `thread` » du noyau `mandel` où chaque thread créé travaillera sur une bande horizontale de l'image.
   Mesurez l'accélération obtenue sur quelques itérations :

   ```
   ./run -s 512 -k mandel -v thread -n -i 50
   ```

   Qu'en concluez-vous ?
2. Définissez maintenant une version « `thread_cyclic` » dans laquelle les lignes de l'image sont attribuées aux threads de manière cyclique.
   Mesurez l'accélération obtenue.
3. Définissez une version « `thread_dyn` » où les lignes sont dynamiquement attribuées aux threads en fonction de leur disponibilité. Utilisez une fonction `get_next_line()` qui sera chargée de renvoyer le numéro de la prochaine ligne pas encore traitée.