Rapport Projet “Traitement d’images”

Raphaël PAVIEL – Alexandre VADET

Le projet consistait à implanter un traitement d’images grâce au GPU Nvidia sous CUDA.

En premier lieu, nous nous sommes basés sur le traitement d’image en séquentiel, pour avoir plus de facilité à l’exporter vers une version prenant en compte des Threads.

L’image sur laquelle nous avons effectué tous nos tests est celle-ci-dessus sous format JPEG.

Noir et Blanc

Nous nous sommes d’abord consacrés à un traitement “simple” de l’image qui ne nécessité pas la prise en compte des pixels voisins de celui sur lequel nous travaillons.
Beaucoup de filtre de la sorte pouvait être appliqués et nous avons choisis le passage de cette image en Noir et Blanc.

```
void noiretblanc( unsigned char * in, unsigned char * out, int colonnes, int lignes ) {
  for(int ligne=1; ligne<lignes-1; ligne++)
  {
    for(int colonne=1; colonne<colonnes-1; colonne++)
    {
      if( colonne < colonnes && ligne < lignes ) {
        int pos = ligne*colonnes+colonne;
        int posG = pos*RGBSIZE;

        unsigned char r = in[posG];
        unsigned char b = in[posG+1];
        unsigned char g = in[posG+2];

        out[posG]=0.21f*r+0.71f*g+0.07f*b;
        out[posG+1]=0.21f*r+0.71f*g+0.07f*b;
        out[posG+2]=0.21f*r+0.71f*g+0.07f*b;
      }
    }
  }
}
```

Cette fonction prend en paramètres, une liste ”in” représentant notre image d’entrée, une liste ”out” représentant notre image de retour, le nombre de ”colonnes” et le nombre de ”lignes”.
Elle parcourt ensuite chaque pixel de notre image grâce aux deux boucles for afin d’appliquer sur chacune des composantes RGB de chaque pixel, les opérations 0.21f sur R, 0.71f sur le G et 0.07f sur le B.
Étant donné que ce fut notre première manipulation sur une image, une des difficultés les plus marquantes fut la difficulté à trouver comment modifier à la fois la composante R,G et B.

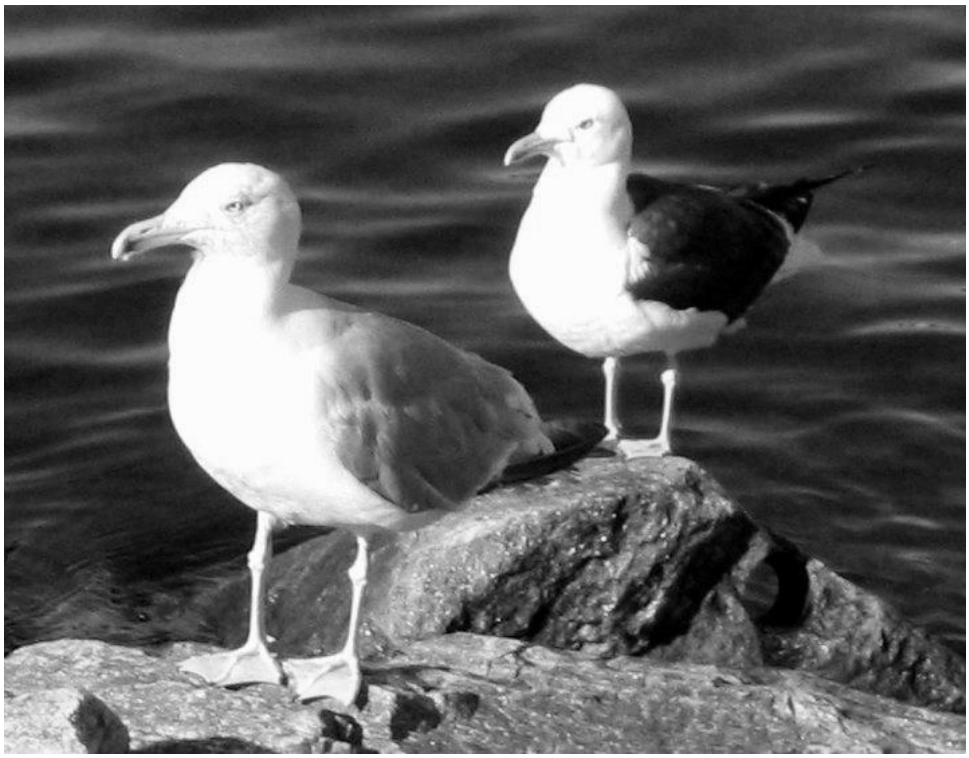

Image en noir et blanc

Retourner l'image

Nous avons ensuite fait un traitement faisant pivoter l'image à 180°.

```
void retourner( unsigned char * in, unsigned char * out, int colonnes, int lignes ) {
  for(int ligne=1; ligne<lignes-1; ligne++)
  {
    for(int colonne=1; colonne<colonnes-1; colonne++)
    {
      if( colonne < colonnes && ligne < lignes ) {

        int pos = RGBSIZE * ( ligne * colonnes + colonne );
        int oppose = colonnes*lignes*RGBSIZE - pos;

        auto rbis = in[oppose];
        auto gbis = in[oppose + 1];
        auto bbis = in[oppose + 2];

        out[pos] = rbis;
        out[pos + 1] = gbis;
        out[pos + 2] = bbis;
      }
    }
  }
}
```

Ce traitement est très similaire à celui du noir et blanc, car, lui aussi ne prend pas en compte les voisins du pixel sur lequel nous travaillons pour calculer la nouvelle valeur.

Notons qu'ici, nous avons instancier rbis, gbis et bbis qui correspond au pixel opposé à celui que nous traitons à l'instant t.

Il n'y avait pas de difficulté particulière à faire cette fonction étant donné que la méthodologie pour la faire était presque identique au noir et blanc.

Image retournée à 180°

Détection de contour (edge detection)

Nous nous sommes ensuite attaqués, aux traitements qui nécessités une considération du voisinage de chaque pixel.

La convolution est un processus permettant d'ajouter à chaque pixel d'une image tous ses voisins selon une certaine matrice.

Voici la matrice correspondante à la détection de contour d'une image.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

La plus grosse difficulté que nous avons rencontrée ici, était la compréhension de ces différentes opérations qui menées vers la valeur du pixel sur lequel nous nous trouvons.

```
void detectionContours(unsigned char * in, unsigned char * out, int colonnes, int lignes) {
  for(int ligne=1; ligne<lignes-1; ligne++)
  {
    for(int colonne=1; colonne<colonnes-1; colonne++)
    {
      if (ligne >= 1 && ligne < lignes - 1 && colonne >= 1 && colonne < colonnes - 1)
      {
          for (int i = 0; i < RGBSIZE; ++i)
          {
              unsigned char p_h = in[RGBSIZE * ((ligne - 1) * colonnes + colonne) + i];
              unsigned char p_g = in[RGBSIZE * (ligne * colonnes + colonne - 1) + i];
              unsigned char pixel = in[RGBSIZE * (ligne * colonnes + colonne) + i];
              unsigned char p_d = in[RGBSIZE * (ligne * colonnes + colonne + 1) + i];
              unsigned char p_b = in[RGBSIZE * ((ligne + 1) * colonnes + colonne) + i];

              int resultat = p_h + p_g + (-4*pixel) + p_d + p_b ;
              if (resultat > 255)
              {
                resultat = 255;
              }
              if (resultat < 0)
              {
                resultat = 0;
              }
              out[RGBSIZE * (ligne * colonnes + colonne) + i] = resultat;
          }
      }
    }
  }
}
```

Ici, tout comme les deux traitements précédents, nous parcourons la liste des pixels de l'image. Étant donné le fait que nous devons connaitre la valeur des voisins de chaque pixel, nous les avons sauvegardées dans des variables prévues à cet effet (p_h, p_g,…).

Grâce à ces variables, nous avons pu calculer un résultat, nous permettant de calculer dans les "if" en dessous, si la couleur de notre pixel serait blanche (255) ou noire (0).

Notons que ce résultat reflète directement les différentes opérations que nous devons faire sur les pixels voisins pour obtenir le résultat. Il est donc normal que nous ne prenions pas le pixel "p_hg" ou "p_bd" car ils sont multipliés par 0.

Image avec détection des contours

Amélioration de la netteté et flou

Tout comme, l'edge detection, ces traitements utilisent la convolution. Nous nous sommes donc contentés de réutiliser notre méthode detectionContours() en changeant les opérations faites pour obtenir le résultat.

Amélioration de la netteté	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

Image avec netteté améliorée

Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

Image avec floutage

CUDA

La version séquentielle faite, nous nous sommes attaqués à la version. Les méthodes ne sont pas différentes, mises à part l'ajout du Threading dans celles-ci.

En effet, le parcourt de l'image se fait ici par bloc de Threads.

```
auto colonne = blockIdx.x * blockDim.x + threadIdx.x;
auto ligne = blockIdx.y * blockDim.y + threadIdx.y;
```

Entrainant la suppression des doubles boucles "for" vues dans la version séquentiel.

TABLE DE COMPARAISON DE TEMPS

Tableau de temps d'exécutions des algorithmes en secondes.

	Séquentiel	Threads
Noir et Blanc	0.0156339	0.00642506
Retourner 180°	0.0119511	0.00778669
Détection de contour	0.03588	0.00737094
Amélioration de netteté	0.0242688	0.00795091
Flouter	0.0398813	0.00769638