Computación Paralela y Distribuida
Departamento de Ciencia de la Computación
Universidad Católica San Pablo
13 de Junio de 2023

CUDA: Grey Scale y Blur

Rafael Alonso David Peñalva
rafael.david@ucsp.edu.pe

1. Introducción

Se utilizan `CUDA C/C++` y `stb_image.h` para implementar dos filtros (kernels):

- toGreyScale
- blur

Empezaremos detallando el input y el ouput de los kernels. Más adelante se explicará cada kernel por separado. Finalmente se verá la configuración de lanzamiento.
El código fuente se puede encontrar en: CUDA-Filters.

2. I/O

2.1. Input

El input para ambos kernels es una imagen, representada como un vector unidimensional de caracteres $(0 - 255)$ en el que irán los 3 canales de cada píxel. Si la imagen es de $20x20$, habrán $20x20x3$ elementos en el vector. A continuación la función `imageToRGBVector` que convierte una imagen a vector:

```
vector<unsigned char> imageToRGBVector(const string &filename,
int &width, int &height, int &channels){
  unsigned char *image =
  stbi_load(filename.c_str(), &width, &height, &channels, STBI_rgb);
  vector<unsigned char> rgbVector;

  if (image != nullptr){
    int imageSize = width * height * channels;
    rgbVector.assign(image, image + imageSize);
    stbi_image_free(image);
  }
  return rgbVector;
}
```

Listing 1: Implementación de imageToRGBVector

Luego, el vector resultante tendrá que ser copiado del host a la memoria del dispositivo.

```
// Se utiliza imateToRGBVector para obtener el vector de entrada
vector<unsigned char> rgbVectorIn = imageToRGBVector(inputPath, width,
    height, channels);
// Se crea un puntero a los datos del vector en el host
unsigned char *h_rgbVectorIn = rgbVectorIn.data();
```

```
// Se crea un puntero a los datos del vector en el dispositivo
unsigned char *d_rgbVectorIn;
// Se separa memoria para almacenar el vector en el dispositivo
cudaMalloc((void **)&d_rgbVectorIn, size);
// Se copia el vector del host al dispositivo
cudaMemcpy(d_rgbVectorIn, h_rgbVectorIn, size, cudaMemcpyHostToDevice);
```

Listing 2: Copia del vector en el host a la memoria del dispositivo

Las imágenes de entrada se encuentran en la carpeta `images/in`.

2.2. Output

El output de los kernels es un vector unidimensional de caracteres $(0 - 255)$ modificado por el kernel. Este vector tendrá que ser copiado a la memoria del host.

```
// Se crea un puntero a los datos del vector en el host
unsigned char *h_rgbVectorOut = rgbVectorOut.data();
// Se crea un puntero a los datos del vector en el dispositivo
unsigned char *d_rgbVectorOut;
// Se separa memoria para almacenar el vector en el dispositivo
cudaMalloc((void **)&d_rgbVectorOut, size);
// Se copia el vector del host al dispositivo
cudaMemcpy(d_rgbVectorOut, h_rgbVectorOut, size,
cudaMemcpyHostToDevice);

// ... procesamiento del kernel sobre d_rgbVectorOut

// Se copia el vector del dispositivo al host
cudaMemcpy(h_rgbVectorOut, d_rgbVectorOut, size,
cudaMemcpyDeviceToHost);
```

Listing 3: Copia del vector en el dispositivo a la memoria del host

Posteriormente se utiliza la función `RGBVectorToImage` para convertir el vector en una imagen. Las imágenes de salida se encuentran en la carpeta `images/out`.

3. Kernels

3.1. toGreyScale

Este kernel realiza una conversión de color a escala de grises en paralelo.

```
__global__
void colorToGreyScaleConversion(unsigned char *Pout, unsigned char
    *Pin, int width, int height)
{
  // Se calculan las coordenadas del pixel correspondiente al hilo en
    el bloque de hilos
  int Col = threadIdx.x + blockIdx.x * blockDim.x;
  int Row = threadIdx.y + blockIdx.y * blockDim.y;
  // Se verifica que el pixel se encuentre dentro de los limites de la
    imagen
  if (Col < width && Row < height)
  {
    // Se calculan los desplazamientos necesarios para acceder al pixel
    en sus 3 canales
```

```
11      int greyOffset = Row * width + Col;
12      int rgbOffset = greyOffset * CHANNELS;
13
14      // Se obtienen los 3 canales del pixel del vector de entrada
15      unsigned char r = Pin[rgbOffset];
16      unsigned char g = Pin[rgbOffset + 1];
17      unsigned char b = Pin[rgbOffset + 2];
18
19      // Se calcula el valor en la escala de grises del pixel
20      unsigned char tmp = 0.21f * r + 0.71f * g + 0.07f * b;
21
22      // Se actualiza el valor del vector de salida
23      Pout[rgbOffset] = tmp;
24      Pout[rgbOffset + 1] = tmp;
25      Pout[rgbOffset + 2] = tmp;
26    }
27 }
```

Listing 4: Implementación de colorToGreyScaleConversion

Ejemplo del escalado aplicado:

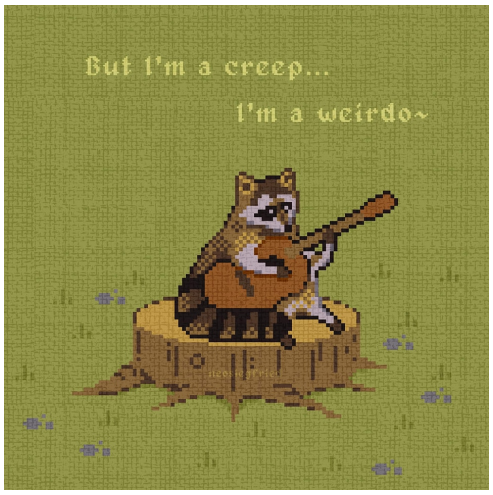

Figura 1: Original

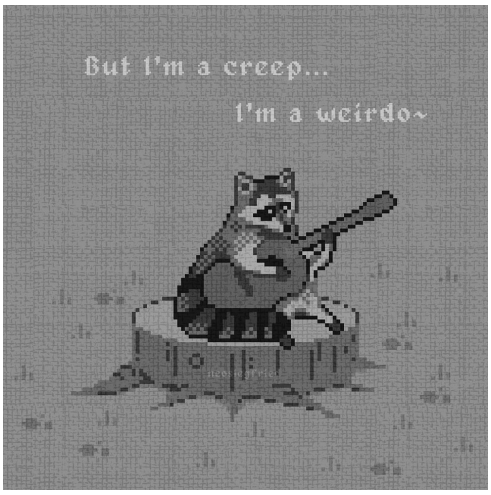

Figura 2: Filtro aplicado

3.2. blur

Este kernel aplica un efecto de desenfoque a una imagen en paralelo. El nivel de desenfoque aplicado dependerá del tamaño de la ventana (píxeles vecinos). El tamaño se configura con la variable `BLUR_SIZE`. Para la implementación se consideró un tamaño de 10 con el objetivo de que el desenfoque sea notable.

```
1 __global__
2 void blurKernel(unsigned char *Pout, unsigned char *Pin, int width, int
      height)
3 {
4   // Se calculan las coordenadas del pixel correspondiente al hilo en
      el bloque de hilos
5   int Col = threadIdx.x + blockIdx.x * blockDim.x;
6   int Row = threadIdx.y + blockIdx.y * blockDim.y;
7
```

```
  // Se verifica que el pixel se encuentre dentro de los limites de la imagen
  if (Col < width && Row < height)
  {
    // Se inicializan las variables que almacenaran los valores acumulados de los canales de color de los pixeles vecinos
    int pixVal_r = 0;
    int pixVal_g = 0;
    int pixVal_b = 0;
    int pixels = 0;

    // Se utiliza un bucle anidado para recorrer los pixeles vecinos dentro de una ventana de finida por BLUR_SIZE
    for (int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE + 1; ++blurRow)
    {
      for (int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE + 1; ++blurCol)
      {
        // Se calcula el desplazamiento para el pixel en la ventana
        int currRow = Row + blurRow;
        int currCol = Col + blurCol;

        int rgbOffset = (currRow * width + currCol) * CHANNELS;

        // Se verifica si el pixel se encuentra dentro de los limites de la imagen
        if (currRow > -1 && currRow < height && currCol > -1 && currCol < width)
        {
          // Se acumulan los valores de los canales del pixel dentro de la ventana y se incrementa el contador de pixeles
          pixVal_r += Pin[rgbOffset];
          pixVal_g += Pin[rgbOffset + 1];
          pixVal_b += Pin[rgbOffset + 2];
          pixels++;
        }
      }
    }
    // Se calcula el desplazamiento del pixel del vector de salida
    int blurrOffset = (Row * width + Col) * CHANNELS;

    // Los valores promedio se calculan dividiendo las sumas cumulativas por el contador. Los resultados se asignan al vector de salida.
    Pout[blurrOffset] = (unsigned char)(pixVal_r / pixels);
    Pout[blurrOffset + 1] = (unsigned char)(pixVal_g / pixels);
    Pout[blurrOffset + 2] = (unsigned char)(pixVal_b / pixels);
  }
}
```

Listing 5: Implementación de blurKernel

Ejemplo del escalado aplicado:

4. Configuración de lanzamiento

La configuración de lanzamiento de los kernels consistirá en 2 variables: `dimGrid` y `dimBloc`.

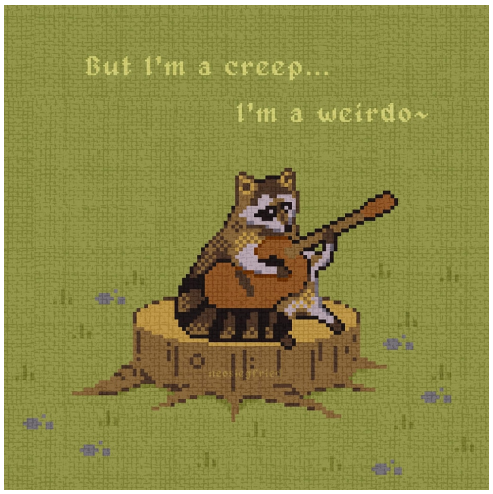

Figura 3: Original

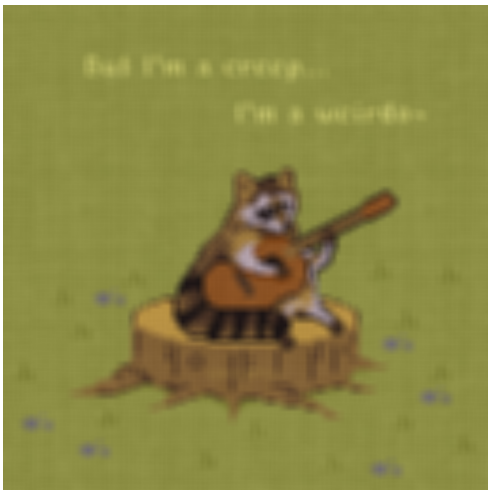

Figura 4: Filtro aplicado

```
dim3 dimGrid(ceil(width / 16.0), ceil(height / 16.0), 1);
dim3 dimBlock(16, 16, 1);
Kernel<<<dimGrid, dimBlock>>>(d_rgbVectorOut, d_rgbVectorIn, width,
    height);
```

Listing 6: Configuración de lanzamiento

La variable `dimGrid` define las dimensiones del grid. Se utiliza la función `ceil` para redondear hacia arriba la división del ancho y alto de la imagen entre 16.0 (tamaño del bloque). Esto asegura que hayan suficientes bloques para cubrir todos los píxeles de la imagen.
La variable `dimBloc` define las dimensiones del bloque de hilos. Se establece un bloque de $16x16x1$, lo que significa que habrán 256 hilos por bloque.