
Universidad Nacional de San Agustin

Ciencia de la Computación

# Informe de los ejercicios de CUDA (imágenes)

***Alumnas :***
*Judith Escalante Calcina*
***Profesor:***
*Mg.Alvaro Henry Mamani Aliaga*

# Índice

# 1. Convertir una imagen a escala de grises

La conversión de un imagen a escala de grises es un proceso muy común , este puede agilizarse con la ayuda de CUDA. En el siguiente código vemos el proceso :

## 1.1. Código fuente

```
#include <stdio.h>
#include <fstream>
#include <iostream>
#define CHANNELS 3
using namespace std;

__global__
void convertir_grises(float * Pout, float * Pin, int width, int height)
{
  int Col = threadIdx.x + blockIdx.x * blockDim.x;
  int Row = threadIdx.y + blockIdx.y * blockDim.y;
  if (Col < width && Row < height)
  {

    int greyOffset = Row*width + Col;
    int rgbOffset = greyOffset*CHANNELS;
    float r = Pin[rgbOffset];
    float g = Pin[rgbOffset + 1];
    float b = Pin[rgbOffset + 2];
    Pout[greyOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
  }
}

void guardar(float o[225][225])
{
  ofstream archivo("toy_gray.dat");
  for (int i = 0; i < 225; ++i)
  {
    for (int j = 0; j < 225; ++j)
    {
          archivo<<o[i][j]<<" ";
    }
    archivo<<endl;
  }
}
void llenar(const char *file, float m[225][225*3])
{
  char buffer[100];
```

```
  ifstream archivo2("toy_gray.dat");
  for (int ii = 0; ii < 225; ++ii)
  {
    for (int jj = 0; jj < 225; ++jj)
    {
          archivo2>>m[ii][jj*3]>>m[ii][jj*3+1]>>m[ii][jj*3+2];
    }
    archivo2.getline(buffer,100);
  }
}

void Escala(float m[225][225*3],int width, int height)
{
  float o[225][225];

  int size_in = width * (height*3);
  int size_out = width * height;
  int memSize_in = size_in * sizeof(float);
  int memSize_out = size_out * sizeof(float);

  float *d_A, *d_B;

  cudaMalloc((void **) &d_A, memSize_in);
  cudaMalloc((void **) &d_B, memSize_out);
  cudaMemcpy(d_A, m, memSize_in, cudaMemcpyHostToDevice);


  dim3 DimGrid(floor((width-1)/16 + 1), floor((height-1)/16+1), 1);
  dim3 DimBlock(16, 16, 1);
  convertir_grises<<<DimGrid,DimBlock>>>(d_B, d_A, width, height);
  cudaMemcpy(o, d_B, memSize_out, cudaMemcpyDeviceToHost);
  cudaFree(d_A);
  cudaFree(d_B);
  guardar(o);
}




int main()
{
  int width=225, height=225;
  float m[225][225*3];
  llenar("toy.dat",m);
  Escala(m,width,height);
  return EXIT_SUCCESS;
}
```

- Función llenar:
  - Esta función lee el archivo toy.dat el cual contiene la matriz de la imagen en valores númericos por cada pixel , cada fila del archivo pertenece a una fila del de la matriz de la imagen que se utilizará.Estos valores van desde el 0 hasta el 225.
- Escala:
  - Esta función recibe una matriz el ancho y largo de la figura original.
  - Define el tamaño de la matriz de entrada y el tamaño de la matriz de salida; támbien define el tamaño de las matrices que almacenarán la información en GPU , usa el *cudaMalloc* para reservar memoria en GPU del tamaño ya asignado anteriormente.
  - Asignamos el tamano de cada Grid , y por ende el tamaño de cada bloque.Por último le pasamos a la función Convertir grises el tamaño de GRID y el tamaño de bloques con las dos matrices en GPU y el ancho y largo de la figura original; finalmente copiamos a la matriz de tamaño estático $o$ la matriz resultante; es decir la matriz en escala de grises $d_B$ y liberamos memoria en la GPU.
- Función Convertir grises:
  - Esta es la conversión que hace que la figura se vuelva a escala de grises , primero lo que hace es asignar cuantos *thread* habrá en cada bloque.
  - Para cada pixel de la figura tenemos que saber su valor en *red* , *green* and *blue*; para asi tener un perfecto ajuste en todos los pixel de la figura.
  - Para obtener la intensidada de cada color en el pixel debemos procesar estos valores con esta fórmula $0{,}21 * redpixel + 0{,}71 * greenpixel + 0{,}07 * blue_pixel$ y almacenar este valor en la matriz de salida.
- Guardar:
  - Esta ultima función guarad la matriz de la figura en escala de grises en un archivo .dat para luego ser procesado con la matriz *Cimg*.

### 1.2. Resultados

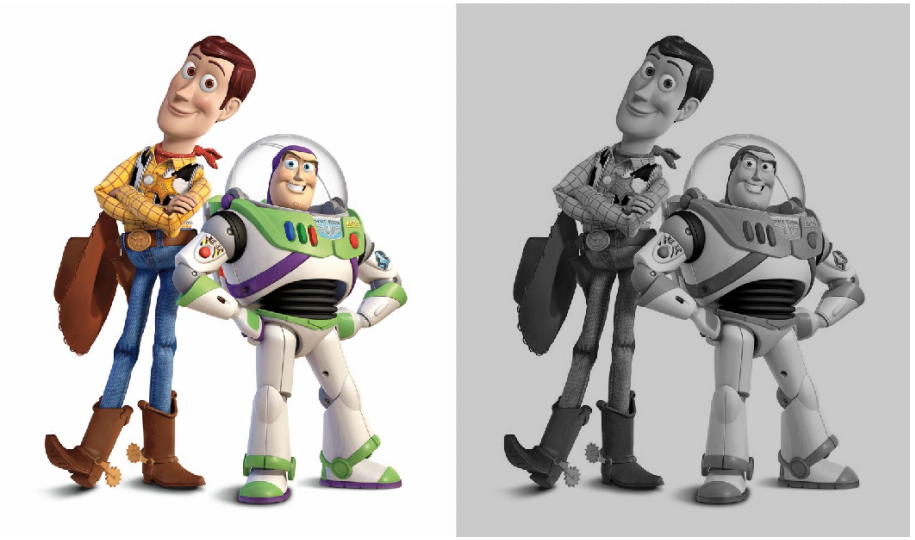

Figura 1: Imagen real y en escala de grises

## 2. Imagen borrosa

### 2.1. Código fuente

```
#include <stdio.h>
#include <fstream>
#include <iostream>
#define BLUR_SIZE 1
using namespace std;

__global__
void Kernel_bor(float * in, float * out, int w, int h)
{
  int Col = blockIdx.x * blockDim.x + threadIdx.x;
  int Row = blockIdx.y * blockDim.y + threadIdx.y;
  if (Col < w && Row < h)
  {
    int pixVal = 0;
    int pixels = 0;

    for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow)
    {
      for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol)
```

```
      {
        int curRow = Row + blurRow;
        int curCol = Col + blurCol;
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w)
        {
          pixVal += in[curRow * w + curCol];
          pixels++;
        }
      }
    }

    out[Row * w + Col] = (float)(pixVal / pixels);
  }
}

void guardar(float r[225][225], float g[225][225], float b[225][225])
{
  ofstream archivo("toy_bor.dat");
  for (int i = 0; i < 225; ++i)
  {
    for (int j = 0; j < 225; ++j)
    {
          archivo<<r[i][j]<<" "<<g[i][j]<<" "<<b[i][j]<<" ";
    }
    archivo<<endl;
  }
}

void borroso(float r[225][225], float g[225][225], float b[225][225],
    int width, int height)
{
  float o_r[225][225];
  float o_g[225][225];
  float o_b[225][225];

  int size = width * height;
  int memSize = size * sizeof(float);

  float *d_A, *d_B;

  cudaMalloc((void **) &d_A, memSize);
  cudaMalloc((void **) &d_B, memSize);
  cudaMemcpy(d_A, r, memSize, cudaMemcpyHostToDevice);


  dim3 DimGrid(floor((width-1)/16 + 1), floor((height-1)/16+1), 1);
  dim3 DimBlock(16, 16, 1);
  Kernel_bor<<<DimGrid,DimBlock>>>(d_A, d_B, width, height);
```

```
  cudaMemcpy(o_r, d_B, memSize, cudaMemcpyDeviceToHost);
  Kernel_bor<<<DimGrid,DimBlock>>>(d_A, d_B, width, height);
  cudaMemcpy(o_g, d_B, memSize, cudaMemcpyDeviceToHost);
  cudaMemcpy(d_A, b, memSize, cudaMemcpyHostToDevice);

  Kernel_bor<<<DimGrid,DimBlock>>>(d_A, d_B, width, height);

  cudaMemcpy(o_b, d_B, memSize, cudaMemcpyDeviceToHost);
  cudaFree(d_A);
  cudaFree(d_B);
  guardar(o_r,o_g,o_b);
}

void llenar(const char *file, float r[225][225], float g[225][225],
    float b[225][225])
{
  char buffer[100];
  ifstream archivo2(file);
  for (int ii = 0; ii < 225; ++ii)
  {
    for (int jj = 0; jj < 225; ++jj)
    {
          archivo2>>r[ii][jj]>>g[ii][jj]>>b[ii][jj];
    }
    archivo2.getline(buffer,100);
  }
}

int main()
{
  int width=225, height=225;
  float r[225][225];
  float g[225][225];
  float b[225][225];
  llenar("toy.dat",r,g,b);
  borroso(r,g,b,width,height);
  return EXIT_SUCCESS;
}
```

- Función llenar:
  - Esta función lee el archivo toy.dat el cual contiene la matriz de la imagen en valores númericos por cada pixel , cada fila del archivo pertenece a una fila del de la matriz de la imagen que se utilizará.Estos valores van desde el 0 hasta el 225.
- Borroso:

- Esta función recibe tres matrices una de cad color del mismo tamaño que la imagen original.
- Define el tamaño de las matrices que almacenarán la información en GPU , usa el *cudaMalloc* para reservar memoria en GPU del tamaño ya asignado anteriormente.
- Asignamos el tamano de cada Grid , y por ende el tamaño de cada bloque.Por último le pasamos a la blor Kernel el tamaño de GRID y el tamaño de bloques con las dos matrices en GPU y el ancho y largo de la figura original para cada matriz, para asi tener el difuminado de la figura de forma precisa.

- Función kernel bor:
  - En esta función compara el cada pixel con los de su alrededor y asi decide como modificara el valor para su difuminado.
- Guardar:
  - Esta ultima función guarad la matriz de la figura en escala de grises en un archivo .dat para luego ser procesado con la matriz *Cimg*.

## 2.2. Resultados


Figura 2: Imagen real e imagen borrosa

# 3. Conclusión

Aunque no se pueda apreciar , el procesamiento de imagenes en CUDA es más eficiente que el hecho por el CPU , como Cimg ; actualmente los ejercicios del informe son realizados con imagenes pequeñas pero si se tiene una imagen enorme , almacenar los datos de esta en la tarjeta gráfica resulta provechoso para su procesamiento veloz y eficiente.