Universidad Nacional de San Agustin

Escuela Profesional de
Ciencia de la Computación

Algoritmos Paralelos

CUDA

Alumna :
Rosa Yuliana Gabriela
Paccotacya Yanque
Profesor:
Mg.Alvaro Henry Mamani
Aliaga

Índice

Todas las pruebas y ejecuciones de código se realizaron en una PC con NVIDIA Corporation GK208M GeForce GT 740M con capability de 3.0 y Cuda 8.0.

1. Suma de Vectores

1.1. Código

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <string>
#include <cuda.h>
using namespace std;
__global__ void suma_vectores(float *c ,float *a , float *b,int N)
{
    int idx=blockIdx.x * blockDim.x+ threadIdx.x;
    if(idx<N)
    {
            c[idx]=a[idx] + b[idx];
    }
}

int main(void)
{
    float *a_h,*b_h,*c_h;
    float *a_d,*b_d,*c_d;
    int N=1000000;
    size_t size=N*sizeof(float);
    a_h = (float *) malloc (size);
    b_h = (float *) malloc (size);
    c_h = (float *) malloc (size);
    srand(1);

    for (int i = 0; i < N; ++i) {
     a_h[i] = rand();
     b_h[i] = rand();
    }


    cudaMalloc((void**)& a_d,size);
    cudaMalloc((void**)& b_d,size);
    cudaMalloc((void**)& c_d,size);

    cudaMemcpy(a_d,a_h,size,cudaMemcpyHostToDevice);
```

```
  cudaMemcpy(b_d,a_h,size,cudaMemcpyHostToDevice);
  int block_size=256;
  int n_blocks=N/block_size + (N%block_size ==0 ? 0:1);
  suma_vectores <<< n_blocks,block_size >>> (c_d,a_d,b_d,N);
  cudaMemcpy (c_h,c_d,size,cudaMemcpyDeviceToHost);
  printf("Suma con  %d hebras con %d hebras por bloque!\n", N,
   block_size);

  /*for (int i=0;i<N;i++)
  {
     cout<<c_h[i]<<" "<<endl;
  }*/

  free(a_h);
  free(b_h);
  free(c_h);
  return(0);
}
```

Código 1: Suma de vectores

1.2. Output

```
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ nvcc -o a sum_vect.cu
nvcc warning : The 'compute_20', 'sm_20', and 'sm_21' architectures are deprec
ed, and may be removed in a future release (Use -Wno-deprecated-gpu-targets to
uppress warning).
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ ./a
Suma con  1000000 hebras con 256 hebras por bloque!
```

2. Suma de Matrices

2.1. Código

```
#include <cuda.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

const int N = 16384;
const int THREADS_PER_BLOCK = 512;

__global__ void add_threads_blocks (int *a, int *b, int *c, int n) {

  int index = threadIdx.x * blockIdx.x * threadIdx.x;
```

```
  if (index < n) {
    c[index] = a[index] + b[index];
  }
}

int main(void) {
  int *a, *b, *c;
  int *d_a, *d_b, *d_c;
  size_t size = N * sizeof(int);

  srand(1);

  a = (int *) malloc(size);
  b = (int *) malloc(size);
  c = (int *) malloc(size);


  for (int i = 0; i < N; ++i) {
    a[i] = rand();
    b[i] = rand();
  }
  //uint kernelTime;
  //cutCreateTimer(&kernelTime);
  //cutResetTimer(kernelTime);
  cudaMalloc((void **) &d_a, size);
  cudaMalloc((void **) &d_b, size);
  cudaMalloc((void **) &d_c, size);
  cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
  //cutStartTimer(kernelTime);
  add_threads_blocks<<<(N + (THREADS_PER_BLOCK - 1)) /
    THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c, N);
  // cudaThreadSynchronize();
  //cutStopTimer(kernelTime);
  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
  printf("Suma con %d hebras con %d hebras por bloque!\n", N,
    THREADS_PER_BLOCK);
  //printf ("Time for the kernel: %f ms\n", cutGetTimerValue(kernelTime
    ));
  free(a); free(b); free(c);
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

  return 0;
}
```

Código 2: Suma de matrices

2.2. Output

```
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ nvcc -o a sum_matrix.cu
nvcc warning : The 'compute_20', 'sm_20', and 'sm_21' architectures are deprecat
ed, and may be removed in a future release (Use -Wno-deprecated-gpu-targets to s
uppress warning).
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ ./a
Suma con  16384 hebras con 512 hebras por bloque!
```

3. Procesamiento de Imágenes

Para obtener las matrices de las imagenes y tambien poder mostrarlas luego de su procesamiento se usó Cimg y archivos .dat.

3.1. Escala de Grises

```
#include <stdio.h>
#include <fstream>
#include <iostream>
#define CHANNELS 3 // we have 3 channels corresponding to RGB
using namespace std;

#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(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)
  {
    // get 1D coordinate for the grayscale image
    int greyOffset = Row*width + Col;
    // one can think of the RGB image having
    // CHANNEL times columns than the grayscale image
    int rgbOffset = greyOffset*CHANNELS;
    float r = Pin[rgbOffset]; // red value for pixel
    float g = Pin[rgbOffset + 1]; // green value for pixel
    float b = Pin[rgbOffset + 2]; // blue value for pixel
    // perform the rescaling and store it
    // We multiply by floating point constants
    Pout[greyOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
  }
```

```
}


void save_data(float o[225][225])
{
  ofstream archivo("gray.dat");
  for (int i = 0; i < 225; ++i)
  {
    for (int j = 0; j < 225; ++j)
    {
          archivo<<o[i][j]<<" ";
    }
    archivo<<endl;
  }
}

void GrayScale(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);
  colorConvert<<<DimGrid,DimBlock>>>(d_B, d_A, width, height);

  cudaMemcpy(o, d_B, memSize_out, cudaMemcpyDeviceToHost);

  cudaFree(d_A);
  cudaFree(d_B);
  save_data(o);
}

void leer_data(const char *file, float m[225][225*3])
{
  char buffer[100];
  ifstream archivo2("image.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);
  }
}

int main()
{
  int width=225, height=225;
  float m[225][225*3];
  leer_data("image.dat",m);
  GrayScale(m,width,height);

  return EXIT_SUCCESS;
}
```

Código 3: Grayscale

Figura 1: Lena original

3.2. Blur

```
#include <stdio.h>
#include <fstream>
#include <iostream>
```

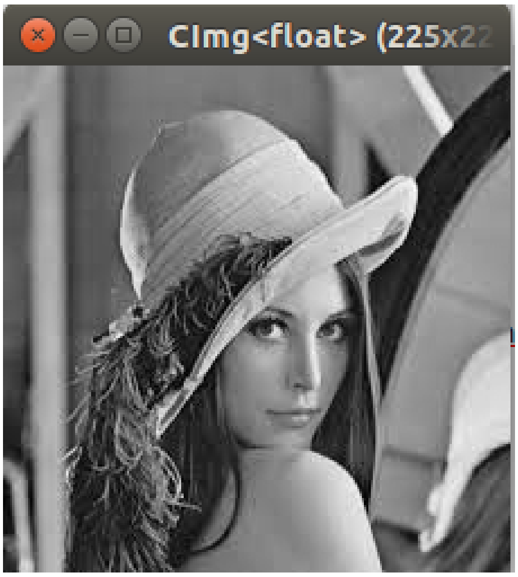

Figura 2: Lena a escala de grises

```
#define BLUR_SIZE 3
using namespace std;

__global__
void blurKernel(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;
    // Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
    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;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w)
        {
          pixVal += in[curRow * w + curCol];
          pixels++; // Keep track of number of pixels in the
  accumulated total
        }
      }
```

```
    }
    // Write our new pixel value out
    out[Row * w + Col] = (float)(pixVal / pixels);
  }
}

void save_data(float r[225][225], float g[225][225], float b[225][225])
{
  ofstream archivo("bluur.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 Blur(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);
  blurKernel<<<DimGrid,DimBlock>>>(d_A, d_B, width, height);
    cudaMemcpy(o_r, d_B, memSize, cudaMemcpyDeviceToHost);

  cudaMemcpy(d_A, g, memSize, cudaMemcpyHostToDevice);


  blurKernel<<<DimGrid,DimBlock>>>(d_A, d_B, width, height);
  cudaMemcpy(o_g, d_B, memSize, cudaMemcpyDeviceToHost);


  cudaMemcpy(d_A, b, memSize, cudaMemcpyHostToDevice);

```

```
  blurKernel<<<DimGrid,DimBlock>>>(d_A, d_B, width, height);
  cudaMemcpy(o_b, d_B, memSize, cudaMemcpyDeviceToHost);

  cudaFree(d_A);
  cudaFree(d_B);
  save_data(o_r,o_g,o_b);
}

void leer_data(const char *file, float r[225][225], float g[225][225],
    float b[225][225])
{
  char buffer[100];
  ifstream archivo2("lena.dat");
  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];
  leer_data("lena.dat",r,g,b);
  Blur(r,g,b,width,height);
  return EXIT_SUCCESS;
}
```

Código 4: Blur

Figura 3: Lena original

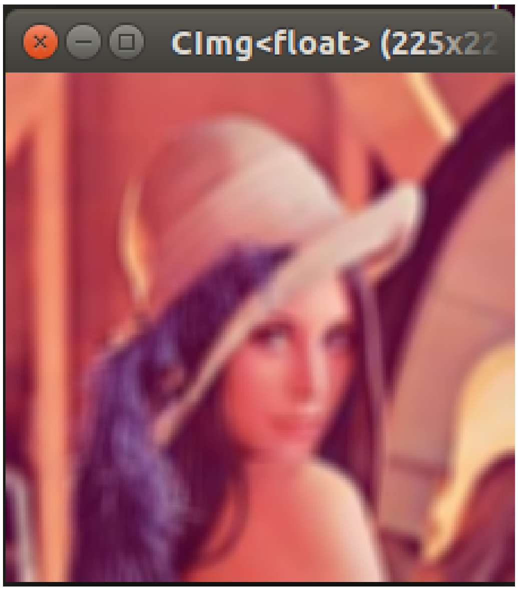

Figura 4: Lena con BlurSize de 3