



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

1. Suma de Vectores	2
1.1. Código	2
1.2. Output	3
2. Suma de Matrices	3
2.1. Código	3
2.2. Output	5
3. Procesamiento de Imágenes	5
3.1. Escala de Grises	5
3.2. Blur	7

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

Se realiza la suma de vectores en GPU de 1000000 de hebras y con 256 hebras por bloque, En la funcion Kernel se especifica el número de bloques y hebras en cada uno de estos. La suma se realiza hebra a hebra.

1.1. Código

```
1 #include <iostream>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <vector>
5 #include <string>
6 #include <cuda.h>
7 using namespace std;
8 __global__ void suma_vectores(float *c ,float *a , float *b,int N)
9 {
10     int idx=blockIdx.x * blockDim.x+ threadIdx.x;
11     if(idx<N)
12     {
13         c[idx]=a[idx] + b[idx];
14     }
15 }
16
17 int main(void)
18 {
19     float *a_h,*b_h,*c_h;
20     float *a_d,*b_d,*c_d;
21     int N=1000000;
22     size_t size=N*sizeof(float);
23     a_h = (float *) malloc (size);
24     b_h = (float *) malloc (size);
25     c_h = (float *) malloc (size);
26     srand(1);
27
28     for (int i = 0; i < N; ++i) {
29         a_h[i] = rand();
30         b_h[i] = rand();
31     }
32
33 }
```

```

34  cudaMalloc((void**)& a_d, size);
35  cudaMalloc((void**)& b_d, size);
36  cudaMalloc((void**)& c_d, size);
37
38  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
39  cudaMemcpy(b_d, a_h, size, cudaMemcpyHostToDevice);
40  int block_size=256;
41  int n_blocks=N/block_size + (N%block_size ==0 ? 0:1);
42  suma_vectores <<< n_blocks, block_size >>> (c_d, a_d, b_d, N);
43  cudaMemcpy(c_h, c_d, size, cudaMemcpyDeviceToHost);
44  printf("Suma con  %d hebras con  %d hebras por bloque!\n", N,
    block_size);
45
46  /*for (int i=0;i<N;i++)
47  {
48      cout<<c_h[i]<<" "<<endl;
49  }*/
50
51  free(a_h);
52  free(b_h);
53  free(c_h);
54  return(0);
55 }

```

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 deprecated, and may be removed in a future release (Use -Wno-deprecated-gpu-targets to suppress warning).
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ ./a
Suma con  1000000 hebras con 256 hebras por bloque!

```

2. Suma de Matrices

Se realiza la suma de matrices en GPU de 16384 de hebras y con 512 hebras por bloque, En la funcion Kernel se especifica el número de bloques y hebras en cada uno de estos. La suma se realiza hebra a hebra.

2.1. Código

```
1 #include <cuda.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <stddef.h>
5
6 const int N = 16384;
7 const int THREADS_PER_BLOCK = 512;
8
9 __global__ void add_threads_blocks (int *a, int *b, int *c, int n) {
10
11     int index = threadIdx.x * blockDim.x * threadIdx.x;
12     if (index < n) {
13         c[index] = a[index] + b[index];
14     }
15 }
16
17 int main(void) {
18     int *a, *b, *c;
19     int *d_a, *d_b, *d_c;
20     size_t size = N * sizeof(int);
21
22     srand(1);
23
24     a = (int *) malloc(size);
25     b = (int *) malloc(size);
26     c = (int *) malloc(size);
27
28
29     for (int i = 0; i < N; ++i) {
30         a[i] = rand();
31         b[i] = rand();
32     }
33     //uint kernelTime;
34     //cutCreateTimer(&kernelTime);
35     //cutResetTimer(kernelTime);
36     cudaMalloc((void **) &d_a, size);
37     cudaMalloc((void **) &d_b, size);
38     cudaMalloc((void **) &d_c, size);
39     cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
40     cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
41     //cutStartTimer(kernelTime);
42     add_threads_blocks<<<(N + (THREADS_PER_BLOCK - 1)) /
43         THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c, N);
44     // cudaThreadSynchronize();
45     //cutStopTimer(kernelTime);
46     cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
47     printf("Suma con %d hebras con %d hebras por bloque!\n", N,
48         THREADS_PER_BLOCK);
```

```
47 //printf ("Time for the kernel: %f ms\n", cutGetTimerValue(kernelTime
   ));
48 free(a); free(b); free(c);
49 cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
50
51 return 0;
52 }
```

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 deprecate
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

El programa lee un archivo creado con el Cimg, lo almacena en una matriz para luego aplicarle la conversion a esclaa de grises, esto genera un archivo de salida,que despues es leído por el Cimg.

```
1 #include <stdio.h>
2 #include <fstream>
3 #include <iostream>
4 #define CHANNELS 3 // we have 3 channels corresponding to RGB
5 using namespace std;
6
7 #define CHANNELS 3 // we have 3 channels corresponding to RGB
8 // The input image is encoded as unsigned characters [0, 255]
9 --global-- void colorConvert(float * Pout, float * Pin, int width, int
   height)
10 {
11     int Col = threadIdx.x + blockIdx.x * blockDim.x;
12     int Row = threadIdx.y + blockIdx.y * blockDim.y;
```

```

13  if (Col < width && Row < height)
14  {
15      // get 1D coordinate for the grayscale image
16      int greyOffset = Row*width + Col;
17      // one can think of the RGB image having
18      // CHANNEL times columns than the grayscale image
19      int rgbOffset = greyOffset*CHANNELS;
20      float r = Pin[rgbOffset]; // red value for pixel
21      float g = Pin[rgbOffset + 1]; // green value for pixel
22      float b = Pin[rgbOffset + 2]; // blue value for pixel
23      // perform the rescaling and store it
24      // We multiply by floating point constants
25      Pout[greyOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
26  }
27 }
28
29
30 void save_data(float o[225][225])
31 {
32     ofstream archivo("gray.dat");
33     for (int i = 0; i < 225; ++i)
34     {
35         for (int j = 0; j < 225; ++j)
36         {
37             archivo<<o[i][j]<<" ";
38         }
39         archivo<<endl;
40     }
41 }
42
43 void GrayScale(float m[225][225*3], int width, int height)
44 {
45     float o[225][225];
46
47     int size_in = width * (height*3);
48     int size_out = width * height;
49     int memSize_in = size_in * sizeof(float);
50     int memSize_out = size_out * sizeof(float);
51
52     float *d_A, *d_B;
53
54     cudaMalloc((void **) &d_A, memSize_in);
55     cudaMalloc((void **) &d_B, memSize_out);
56
57     cudaMemcpy(d_A, m, memSize_in, cudaMemcpyHostToDevice);
58
59     dim3 DimGrid(floor((width-1)/16 + 1), floor((height-1)/16+1), 1);
60     dim3 DimBlock(16, 16, 1);
61     colorConvert<<<DimGrid,DimBlock>>>(d_B, d_A, width, height);

```

```
62
63     cudaMemcpy(o, d_B, memSize_out, cudaMemcpyDeviceToHost);
64
65     cudaFree(d_A);
66     cudaFree(d_B);
67     save_data(o);
68 }
69
70 void leer_data(const char *file, float m[225][225*3])
71 {
72     char buffer[100];
73     ifstream archivo2("image.dat");
74     for (int ii = 0; ii < 225; ++ii)
75     {
76         for (int jj = 0; jj < 225; ++jj)
77         {
78             archivo2 >> m[ii][jj*3] >> m[ii][jj*3+1] >> m[ii][jj*3+2];
79         }
80         archivo2.getline(buffer, 100);
81     }
82 }
83
84 int main()
85 {
86     int width=225, height=225;
87     float m[225][225*3];
88     leer_data("image.dat", m);
89     GrayScale(m, width, height);
90
91     return EXIT_SUCCESS;
92 }
```

Código 3: Grayscale

3.2. Blur

El programa lee un archivo creado con el Cimg, lo almacena en una matriz para luego aplicarle el Blur(halla el promedio de los n pixeles alrededor de cada punto), este se aplica a cada una de las capas REd, Green y Blue, esto genera un archivo de salida, que despues es leído por el Cimg.

```
1 #include <stdio.h>
2 #include <fstream>
3 #include <iostream>
4 #define BLUR_SIZE 3
5 using namespace std;
6
```

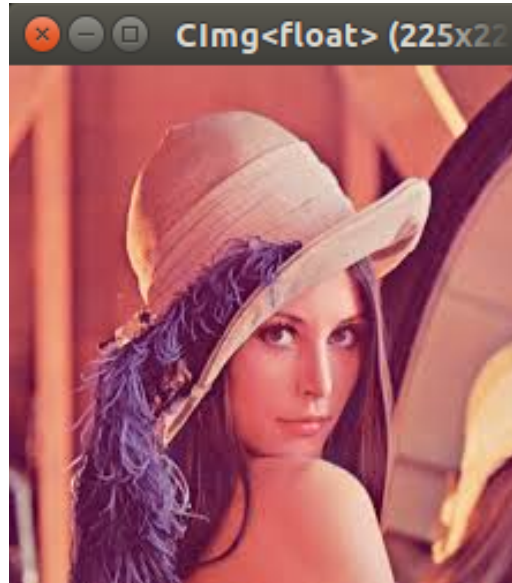



Figura 1: Lena original

```
7  __global__
8  void blurKernel(float * in, float * out, int w, int h)
9  {
10     int Col = blockIdx.x * blockDim.x + threadIdx.x;
11     int Row = blockIdx.y * blockDim.y + threadIdx.y;
12     if (Col < w && Row < h)
13     {
14         int pixVal = 0;
15         int pixels = 0;
16         // Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
17         for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow)
18         {
19             for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol)
20             {
21                 int curRow = Row + blurRow;
22                 int curCol = Col + blurCol;
23                 // Verify we have a valid image pixel
24                 if(curRow > -1 && curRow < h && curCol > -1 && curCol < w)
25                 {
26                     pixVal += in[curRow * w + curCol];
27                     pixels++; // Keep track of number of pixels in the
28                     // accumulated total
29                 }
30             }
31             // Write our new pixel value out
32             out[Row * w + Col] = (float)(pixVal / pixels);
```

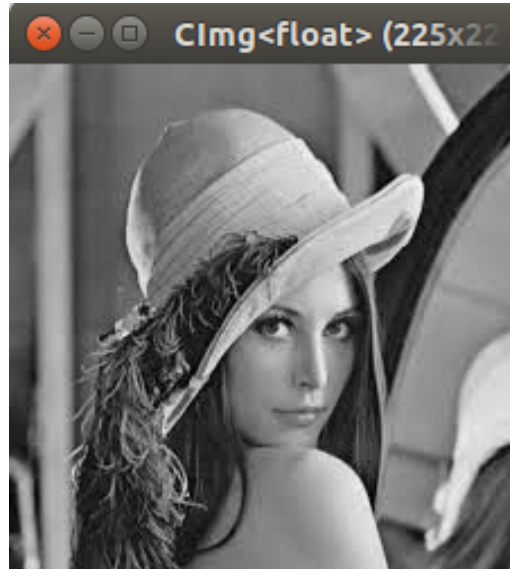


Figura 2: Lena a escala de grises

```
33 }  
34 }  
35  
36 void save_data(float r[225][225], float g[225][225], float b[225][225])  
37 {  
38     ofstream archivo("bluur.dat");  
39     for (int i = 0; i < 225; ++i)  
40     {  
41         for (int j = 0; j < 225; ++j)  
42         {  
43             archivo<<r[i][j]<<" "<<g[i][j]<<" "<<b[i][j]<<" ";  
44         }  
45         archivo<<endl;  
46     }  
47 }  
48  
49 void Blur(float r[225][225], float g[225][225], float b[225][225], int  
    width, int height)  
50 {  
51     float o_r[225][225];  
52     float o_g[225][225];  
53     float o_b[225][225];  
54  
55     int size = width * height;  
56     int memSize = size * sizeof(float);  
57  
58     float *d_A, *d_B;
```

```

59
60     cudaMalloc((void **) &d_A, memSize);
61     cudaMalloc((void **) &d_B, memSize);
62
63     cudaMemcpy(d_A, r, memSize, cudaMemcpyHostToDevice);
64     dim3 DimGrid(floor((width-1)/16 + 1), floor((height-1)/16+1), 1);
65     dim3 DimBlock(16, 16, 1);
66     blurKernel<<<DimGrid,DimBlock>>>(d_A, d_B, width, height);
67     cudaMemcpy(o_r, d_B, memSize, cudaMemcpyDeviceToHost);
68
69     cudaMemcpy(d_A, g, memSize, cudaMemcpyHostToDevice);
70
71
72     blurKernel<<<DimGrid,DimBlock>>>(d_A, d_B, width, height);
73     cudaMemcpy(o_g, d_B, memSize, cudaMemcpyDeviceToHost);
74
75
76     cudaMemcpy(d_A, b, memSize, cudaMemcpyHostToDevice);
77
78     blurKernel<<<DimGrid,DimBlock>>>(d_A, d_B, width, height);
79     cudaMemcpy(o_b, d_B, memSize, cudaMemcpyDeviceToHost);
80
81     cudaFree(d_A);
82     cudaFree(d_B);
83     save_data(o_r, o_g, o_b);
84 }
85
86 void leer_data(const char *file, float r[225][225], float g[225][225],
87               float b[225][225])
88 {
89     char buffer[100];
90     ifstream archivo2("lena.dat");
91     for (int ii = 0; ii < 225; ++ii)
92     {
93         for (int jj = 0; jj < 225; ++jj)
94         {
95             archivo2>>r[ii][jj]>>g[ii][jj]>>b[ii][jj];
96         }
97         archivo2.getline(buffer, 100);
98     }
99 }
100 int main()
101 {
102     int width=225, height=225;
103     float r[225][225];
104     float g[225][225];
105     float b[225][225];
106     leer_data("lena.dat", r, g, b);

```

```
107 Blur(r,g,b,width,height);  
108 return EXIT_SUCCESS;  
109 }
```

Código 4: Blur

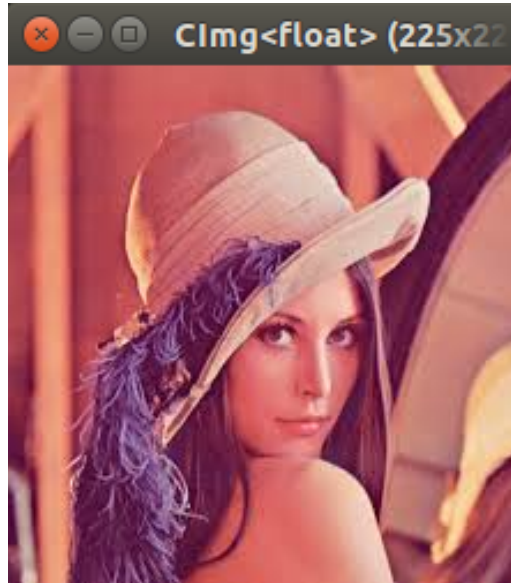


Figura 3: Lena original

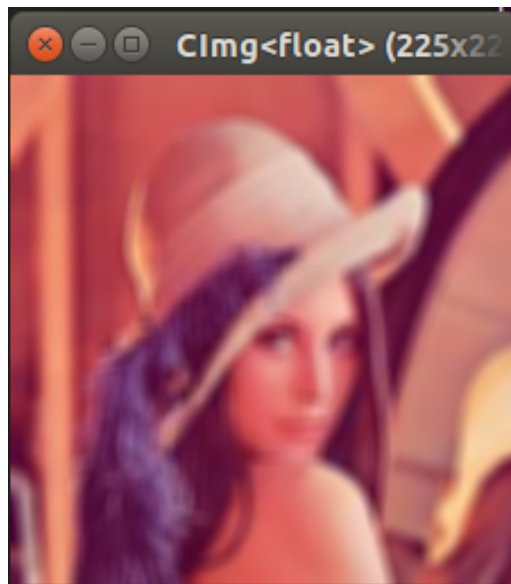


Figura 4: Lena con BlurSize de 3