Ejercicios - Algoritmos Paralelos

Lorena Xiomara Castillo Galdos

Junio 2017

- **Ejercicios cuda**

  - Suma de matrices

```
#include <iostream>
#include <cuda.h>
using namespace std;

__global__ void MatrizAddKernel_A(float *A, float *B, float *C,
        int n)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if(i < n*n)
  {
    C[i] = A[i] + B[i];
  }
}


__global__ MatrizAddKernel_B(float *A, float* B, float* C,
        int n)
{
  int i=threadIdx.x+blockDim.x*blockIdx.x;
  if(i < n*n)
  {
    C[i]=A[i]+B[i];
  }
}


__global__ MatrizAddKernel_C(float *A, float* B, float* C,
        int n)
{
  int i= n*blockIdx.x;
  if(i < n*n)
  {
    for(int j = 0 ; j < n ; j++)
    {
      C[i + j]=A[i + j]+B[i + j];
    }
  }
}

__global__ MatrizAddKernel_D(float *A, float* B, float* C,
        int n)
```

```
{
  int i= blockIdx.x;
  if(i < n*n)
  {
    for(int j = 0 ; j < n ; j++)
    {
      C[i+n *j]=A[i+n *j]+B[i+n *j];
    }
  }
}


void MatrizAdd(float *A, float *B, float *C, int n)
{
  float *d_A, *d_B, *d_C;
  size_t size = n*n * sizeof(float);

  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);

  //MatrizAddKernel_A<<< 5 , 2 >>>(d_A, d_B, d_C, n);
  MatrizAddKernel_B<<< 1 , n*n >>>(d_A, d_B, d_C n);
  //MatrizAddKernel_C<<< n , n >>>(d_A, d_B, d_C, n);
  //MatrizAddKernel_D<<< n , n >>>(d_A, d_B, d_C, n);

  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}

int main(void)
{
    float *A, *B , *C;
    int n = 10 ;
    A = new float[n*n];
    B = new float[n*n];
    C = new float[n*n];
    for (int i = 0; i < n*n; ++i)
    {
        A[i] = i;
        B[i] = i;
    }

    MatrizAdd(A, B, C ,n);

    for(int i = 0; i < n*n ; ++i)
    {
       cout << C[i] << " ";

    }
  return 0;
}
```

- Multiplicación de matriz vector

```
#include <iostream>
#include <cuda.h>
using namespace std;

__global__ void MultMatrizVectKernel(float *A, float *B, float *C,
        int n)
{
  int i = n * blockIdx.x;
  if(i < n*n)
  {
    for(int j = 0; j < n ; ++j)
    {
      C[blockIdx.x] += A[i + j] * B[j];
    }
  }
}

void MultMatrizVector(float *A, float *B, float *C, int n)
{
  float *d_A, *d_B, *d_C;
  size_t size_A = n*n * sizeof(float);
  size_t size_B = n * sizeof(float);
  size_t size_C = n * sizeof(float);

  cudaMalloc((void **) &d_A, size_A);
  cudaMalloc((void **) &d_B, size_B);
  cudaMalloc((void **) &d_C, size_C);

  cudaMemcpy(d_A, A, size_A, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, size_B, cudaMemcpyHostToDevice);

  MultMatrizVectKernel<<< n, 1 >>>(d_A, d_B, d_C, n);

  cudaMemcpy(C, d_C, size_C, cudaMemcpyDeviceToHost);

  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}

int main(void)
{
    float *A, *B , *C;
    int n = 10 ;
    A = new float[n*n];
    B = new float[n];
    C = new float[n];
    for (int i = 0; i < n*n; ++i)
    {
        A[i] = i;
    }

    for (int i = 0; i < n; ++i)
    {
        B[i] = i;
    }

    MultMatrizVector(A, B, C ,n);
```

```
    for(int i = 0; i < n ; ++i)
    {
        cout << C[i] << " ";

    }
  return 0;
}
```

- Imágenes blurr y gray

```
#include <cv.h>
#include <highgui.h>
#include <iostream>
#include <math.h>

using namespace std;

#define CHANNELS 3


__global__ void colorConvertKernel(unsigned char * greyImage,
    unsigned char * rgbImage,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;
            unsigned char r =  rgbImage[rgbOffset    ];
            unsigned char g = rgbImage[rgbOffset + 2];
            unsigned char b = rgbImage[rgbOffset + 3];
            greyImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
         }
}

//1px
__global__  void blurrKernel(unsigned char * in,
        unsigned char * 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;

//promedio de blur_size x blur_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;
//verifica pixel
                    if(curRow > -1 && curRow < h && curCol > -1
                      && curCol < w)
                    {
                        pixVal += in[curRow * w + curCol];
                        pixels++;
                    }
                }
            }
            out[Row * w + Col] = (unsigned char)(pixVal / pixels);
        }
  }


int main(int argc, char** argv)
{
   IplImage* input_image = NULL; //datos imagen
   input_image = cvLoadImage(argv[1], CV_LOAD_IMAGE_UNCHANGED);

    int width = input_image->width;
    int height = input_image->height;

    float* imagen_cpu = new float[width * height * 4];
    float* imagen_gpu = new float[width * height * 4];

    cudaMalloc((void **)(&imagem_gpu), (width * height * 4)
        * sizeof(float));
    cudaMemcpy(imagem_gpu, imagem_cpu, (width * height * 4)
        * sizeof(float), cudaMemcpyHostToDevice);

//16Thrd
    dim3 dimGrid(ceil(width/16.0), ceil(height/16.0), 1);
    dim3 dimBlock(16, 16, 1);

    colorConvertKernel<<<dimGrid,dimBlock>>>(imagen_gpu, input_image,
        width,  height);

    blurrkernel<<<dimGrid,dimBlock>>>(imagem_gpu, input_image,
        width,  height)


    cudaMemcpy(imagem_cpu, imagem_gpu, (width * height * 4)
        * sizeof(float), cudaMemcpyDeviceToHost);
    cvReleaseImage(&input_image);
    cvReleaseImage(&out_image);
    return 0;
}
```

- **Openmp**

  - Odd even sort

```
 #include <iostream>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <chrono>
using namespace std;

void odd_even(int *V, int n, int thread_count)
{
  int fase,i,tmp;
  for (fase = 0; fase < n; fase++)
  {
   if (fase % 2 == 0)
   #pragma omp parallel for num_threads(thread_count) \
     default(none) shared(a, n) private(i, tmp)
   for (i = 1; i < n; i += 2)
   {
        if (V[i - 1] > V[i])
        {
          tmp = V[i - 1];
          V[i - 1] = V[i];
          V[i] = tmp;
        }
    }
    else
    #pragma omp parallel for num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp)
    for (i = 1; i < n - 1; i += 2)
    {
        if (V[i] > V[i+1])
        {
          tmp = V[i+1];
          V[i+1] = V[i];
          V[i] = tmp;
        }
    }
   }
}

void odd_even_(int *V, int n, int thread_count)
{
   int fase,i,tmp;
   #pragma omp parallel num_threads(thread_count) \
        default(none) shared(a, n) private(i, tmp, fase)
   for (fase = 0; fase < n; fase++)
   {
     if (fase % 2 == 0)
     #pragma omp for
     for (i = 1; i < n; i += 2)
     {
        if (V[i - 1] > V[i])
        {
          tmp = V[i - 1];
          V[i - 1] = V[i];
          V[i] = tmp;
        }
     }
     else
     #pragma omp for
     for (i = 1; i < n - 1; i += 2)
     {
```

```
        if (a[i] > a[i+1])
        {
            tmp = a[i+1];
            a[i+1] = a[i];
            a[i] = tmp;
        }
    }
  }
}

void print(int *V,int n)
{
        for (int i = 0; i < n; ++i)
        {
            cout<<V[i]<<" ";
        }
        cout<<endl;
}

int main(int argc, char const *argv[])
{
        int thread_count=4;
        int n=20000;
        int *V=new int[n];
        srand (time(NULL));
        for (int i = 0; i < n; ++i)
        {
                int iSecret = rand() % n;
                V[i]=iSecret;
        }
        auto start = chrono::high_resolution_clock::now();
        odd_even(V,n,thread_count);

        auto finish = chrono::high_resolution_clock::now();
        chrono::duration<double> elapsed = finish - start;
        cout << "Time taken First Odd-Even: " << elapsed.count() <<endl;

        auto start2 = chrono::high_resolution_clock::now();
        odd_even_(V,n,thread_count);

        auto finish2 = chrono::high_resolution_clock::now();
        chrono::duration<double> elapsed2 = finish2 - start2;
        cout << "Time taken Second Odd-Even: " << elapsed2.count() <<endl;
        return 0;
}
```