High Performance Computing - CET4

Introduction to CUDA

Juan José Rodríguez Aldavero
May 28, 2020

Contents

1 Hello World in CUDA

1. **Describe how the code works** We start the practice by defining how the CUDA code provided by the course works. The following code writes the message "Hello World" on the screen:

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

//Definition of the kernel.
__global__
void hello(char *a, int *b) {
  a[threadIdx.x] += b[threadIdx.x];
}

//Definition of the main function
int main() {

  //Definition and initialization of variables and pointers
  char a[N] = "Hello \0\0\0\0\0\0";
  int b[N] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
  char *ad;
  int *bd;

  //Allocation of memory in the host
  const int csize = N*sizeof(char);
  const int isize = N*sizeof(int);

  printf("%s", a); //Prints "Hello "

  //Allocation of memory in the device
  cudaMalloc( (void**)&ad, csize );
  cudaMalloc( (void**)&bd, isize );

  //Copy from the host to the device
  cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );

  //Definition of the grid size
  dim3 dimBlock( blocksize, 1 );
  dim3 dimGrid( 1, 1 );
  hello<<<dimGrid, dimBlock>>>(ad, bd);

  //Copy from the device to the host
  cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );
  cudaFree( ad );
```

```
  printf("%s\n", a);

  return EXIT_SUCCESS;
}
```

Listing 1.1: *Example CUDA program hello.cu*

What this code does is mainly defined by the kernel hello. In essence, the operation performed by CUDA is reduced to the next line of code:

```
a[threadIdx.x] += b[threadIdx.x];
```

where *threadIdx.x* corresponds to the number of the thread within the block, which is size 16.

Since the block is the same size as the array, N = blocksize = 16, we only have a single block and the operation is reduced to adding the characters of the array b[N] to the characters of the array b[N] by bytes. The result of this addition doing the corresponding operation gives rise to the string "Hello World", which is copied from the device to the host in the address of the variable a and later printed on the screen.

2. **What memory transfer is produced between the host and the device (GPU)?**
   Memory transfers occur twice. Firstly, the process carried out on the *host* reserves an amount of memory on both the host and the GPU via the following lines of code:

```
//Host
const int csize = N*sizeof(char);
const int isize = N*sizeof(int);

//Device
cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice);
cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice);
```

and then copies the information reserved in memory in the host over the memory reserved in the device using the

```
cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice);
```

Finally, CUDA performs the operation on the device by running the kernel and copies the information back to the host using the code:

```
cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost);
```

2 Programming Exercise

1. **Provide your implementation using CUDA and provide an example use of your code**
   We continue the practice by doing a programming exercise on CUDA. This exercise will consist of two separate sections. The first will consist of, starting from a MN matrix, calculating the l element moving average for each element in the matrix. In other words, calculate the average of the element (i, j) with the previous l elements $(i, j-1), (i, j-2), dts$. Graphically, it can be seen as

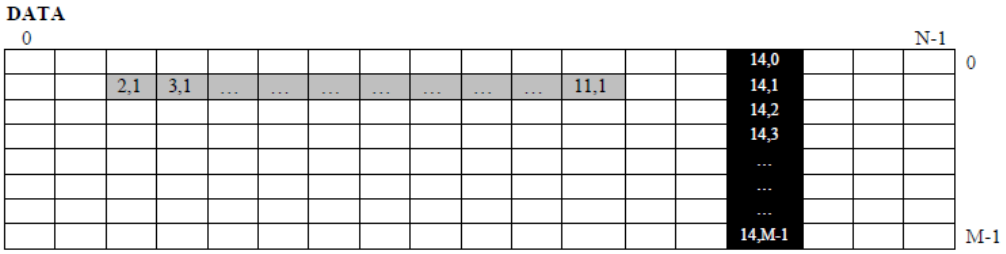

Figure 2.1: Graphical representation for the required CUDA exercise

The second exercise consists of calculating the average per row of the matrix to form a M element vector.

We start with the moving average exercise. We have tried to do the exercise with a two-dimensional representation, but this gives problems (in particular we get the error 'Segmentation Fault' due to problems with memory allocation). For this reason we have chosen to represent the matrix as a one-dimensional vector of $M * N$ elements and later process the data recovering the matrix form. We have chosen to work with a matrix of random numbers between 0 and 10, although other types of matrices can also be initialized with, for example, statistical distributions.

```
__global__ void smooth(float *in, float *out, int n, int l) {

  //Identify the thread ID as the column number
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  //Impose some constraints and calculate the loop
  if (col < n) {
    float tmpAvg = 0.0;
    //Sum the i-th and the 9 previous elements
    for (int i = 0; i < l; i++) {
      //If out of range, sum zeros
```

```
        if (col - i >= 0) {
          tmpAvg += *(in + col-i) ;
        } else {
          tmpAvg += 0;
        }
      }

      *(out + col) = tmpAvg / (float)l;
    }
}
```

Listing 2.1: *Kernel for the moving average of the matrix*

Later, in the body of the function we perform a process very similar to that seen in the previous exercise, only applied to the new kernel.

```
int main(int argc, char *argv[]) {

  //Initialize matrix and window dimensions and allocate pointers
  int n = 50000;
  int l = 9;
  float *d_A;
  float *d_B;
  float *h_A;
  float *h_B;

  //Allocate host and device arrays
  size_t bytes = n*sizeof(float);
  h_A = (float*)malloc(bytes);
  h_B = (float*)malloc(bytes);
  cudaMalloc(&d_A, bytes);
  cudaMalloc(&d_B, bytes);

  //Initialize content of input array
  int i;
  for(i=0; i<n; i++) {
  *(h_A + i) = rand() % 10 + 1;
  }

  //Copy host vector to device
  cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);

  //Define grid dimensions
  dim3 blockSize(32,1,1);
  int grid = (int)ceil((float)n / blockSize.x);
  dim3 gridSize(grid,1,1);

  //Execute the kernel
  smooth<<<gridSize,blockSize>>>(d_A, d_B, n, l);

  //Copy device vector to host
  cudaMemcpy(h_B, d_B, bytes, cudaMemcpyDeviceToHost);

  //Write array to txt file
  ofstream myfile ("moving_average.txt");
```

```
  if (myfile.is_open()) {
    for(int i = 0; i < n; i ++){
      myfile << *(h_B + i) << " " ;
    }
    myfile.close();
  } else cout << "Unable to open file";

  //Release device and host memory
  cudaFree(d_A);
  cudaFree(d_B);
  free(h_A);
  free(h_B);

  return 0;
}
```

Listing 2.2: *Main function for the program*

Compiling this CUDA code as indicated in the course you can obtain a moving_average_l.txt file with 50000 elements in the cluster login node, where l is the window chosen for moving average. The processing of this data has been done in the local machine through the following Python script:

```
import matplotlib.pyplot as plt

def importData(data):
from numpy import genfromtxt
array = genfromtxt(data, delimiter=' ')
return array

def chunks(lst, n):
"""Yield successive n-sized chunks from lst."""
for i in range(0, len(lst), n):
yield lst[i:i + n]

data = "moving_average_100.txt"
lists = list(chunks(importData(data), 1000))
for i in range(len(lists)):
plt.plot(lists[i])
plt.show()
```

Listing 2.3: *Python script to plot the moving averages*

The program has been executed for the values $l = 9$ and $l = 100$. The results obtained are as follows:

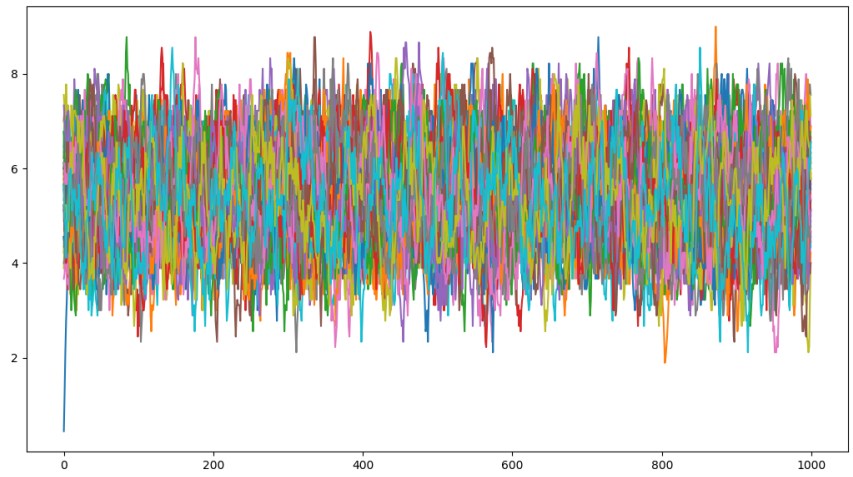

Figure 2.2: *Result for the moving average with a window of 9*

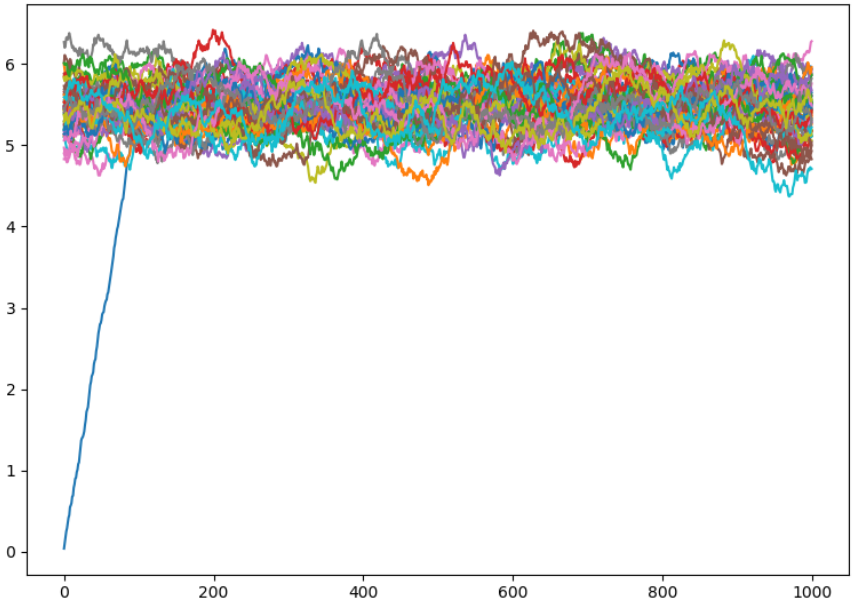

Figure 2.3: *Result for the moving average with a window of 9*

We obtain the representation of $M = 50$ lines of different colors, where each one presents $N = 1000$ random elements between 0 and 10. We see how increasing the size of the

window gives rise to a greater moving averaging that can be seen in both graphics. The origin of the deviation on the left side for the blue data is unknown.

We continue with the deliverable by creating a program that calculates the vector of averages for the k-th element. To do this, we will reuse the previous code by implementing some changes. We start with the kernel:

```
__global__ void avg(float *in, float *out, int n, int m) {
  //Identify the thread ID as the column number
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  //Calculate the loop iterations for the corresponding vector
      elements
  if (col < n) {
    float tmpAvg = 0.0;
    //Calculate the sum of all the elements in the column
    for (int i = 0; i < m; i++) {
      int k = n * i;
      tmpAvg += *(in + col + k) ;
    }
    *(out + col) = tmpAvg;
  }
}
```

Listing 2.4: *Kernel for the vector average of the matrix*

```
int main(int argc, char *argv[]) {

  //Define matrix dimensions and pointers
  int n = 1000; //columns
  int m = 50; //rows
  float *d_A;
  float *d_B;
  float *h_A;
  float *h_B;

  size_t bytes = n*m*sizeof(float);
  size_t bytes_vec = n*sizeof(float);

  //Allocate host arrays
  h_A = (float*)malloc(bytes);
  h_B = (float*)malloc(bytes_vec);

  //Allocate device arrays
  cudaMalloc(&d_A, bytes);
  cudaMalloc(&d_B, bytes_vec);

  //Initialize content of input array
  int i;
  for(i=0; i<(n*m); i++) {
    *(h_A + i) = rand() % 10 + 1;
  }

  //Copy host vector to device
```

```
  cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);

  //Define grid dimensions
  dim3 blockSize(500,1,1);
  int grid = (int)ceil((float)(n*m) / blockSize.x);
  dim3 gridSize(grid,1,1);

  //Execute the kernel
  avg<<<gridSize,blockSize>>>(d_A, d_B, n, m);

  //Copy device vector to host
  cudaMemcpy(h_B, d_B, bytes_vec, cudaMemcpyDeviceToHost);

  //Write array to txt file
  ofstream myfile ("avg_vector.txt");
  if (myfile.is_open()) {
    for(int i = 0; i < n; i ++){
      myfile << *(h_B + i) << " " ;
    }
    myfile.close();
  } else cout << "Unable to open file";

  //Release device and host memory
  cudaFree(d_A);
  cudaFree(d_B);
  free(h_A);
  free(h_B);

  return 0;
}
```

Listing 2.5: *Main function*

The result is vector with 1000 elements between 0 and 10. If we plot their histogram we see that they obey the central limit theorem, that is, their distribution follows a Gaussian distribution.

5,16	5,2	5,08	5,38	4,88	5,66	5,38	5,82	5,8	5	4,94	5,2	5,2
4	5,78	5,26	5,28	5,32	5,46	5,16	4,92	5,86	5,42	5,48	4,76	6,1
6	5,18	4,84	5,34	5,66	5,74	5,62	5,08	5,54	6,04	6,02	6,06	6,06
32	5,68	6,08	6,36	4,88	5,76	5,36	6,14	5,68	5,26	5,46	5,68	6,64
6,2	5,96	5,08	6,06	5,14	4,86	5,94	5,56	5,22	6,04	6	5,54	6,36

Figure 2.4: *Sample of the average vector*

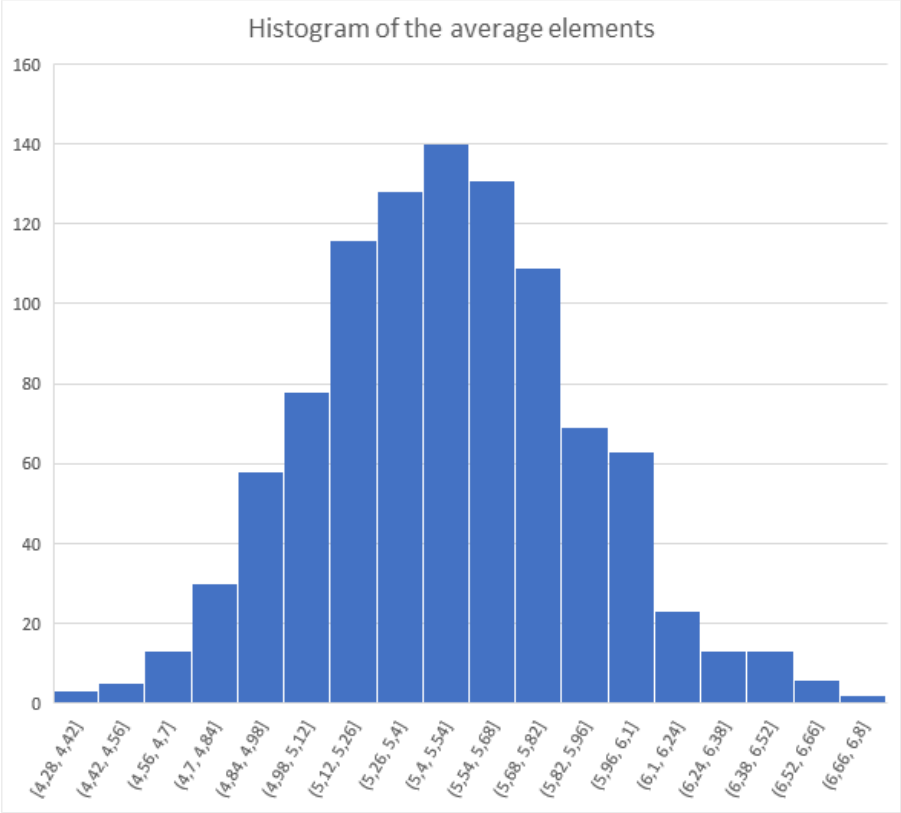

Figure 2.5: *Gaussian distribution for the elements of the average vector*

3 Using CUDA with Data from Cyberinfrastructure

We continue the exercise by applying the above functions to data sets from an ocean observatory cyber-infrastructure. In particular, using data sets from temperature and pressure sensors at different heights of the ocean floor inside an underwater volcano, we want to see the effects of the moving average and the vector of averages defined in the previous exercise.

Essentially, the functions will be the same with the only difference that you will have to read the txt files containing the data and copy the information into the input matrix. Instead of copying all the code again, the relevant part that includes these changes is given by:

```
int i;
tream file("botpt_pres.txt");
file.is_open())

(i=0; i<n; i++) {
e >> *(h_A + i);
```

Listing 3.1: *Necessary change at the code to read the txt files*

We proceed to calculate the moving average of the Cybertext data. In particular, we calculate the smoothings for the pressure and temperature measurements of the CTD sensor at 1542 meters depth, for a window of $l = 1$ (no smoothing) and $l = 100$ (strong smoothing) to see extreme cases. The results are:

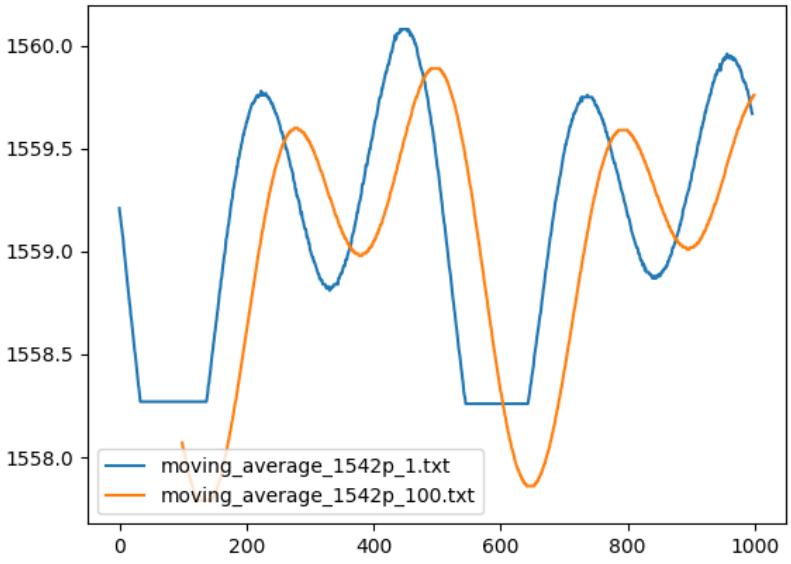

Figure 3.1: *Smoothing for pressure sensor measurements at 1542 meters depth*

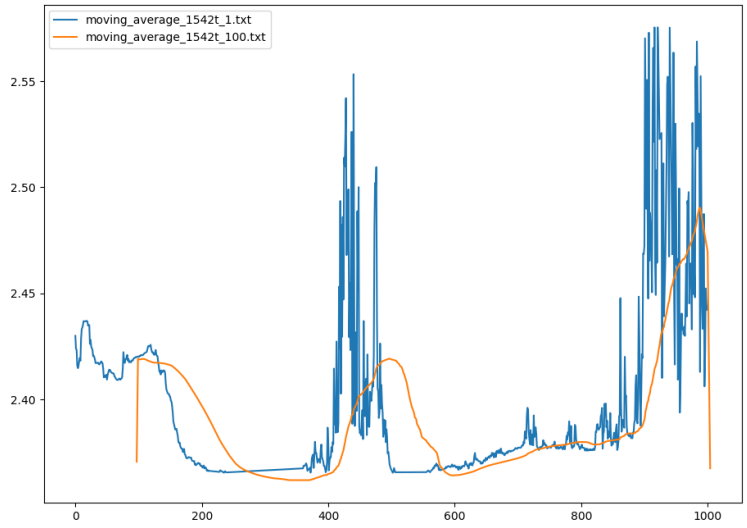

Figure 3.2: *Smoothing for temperature sensor measurements at 1542 meters depth*

We see how the smoothing eliminates the extreme values and the peaks but follows the trend of the original values. The flat parts at the bottom are because due to the presence of outliers a cut has been made for the 15% percentile, so it is not a natural behavior of the data. The values for the BOTPT sensor are also calculated, this time for a wider range of windows to observe the

evolution of the smoothing in more detail. The result is:

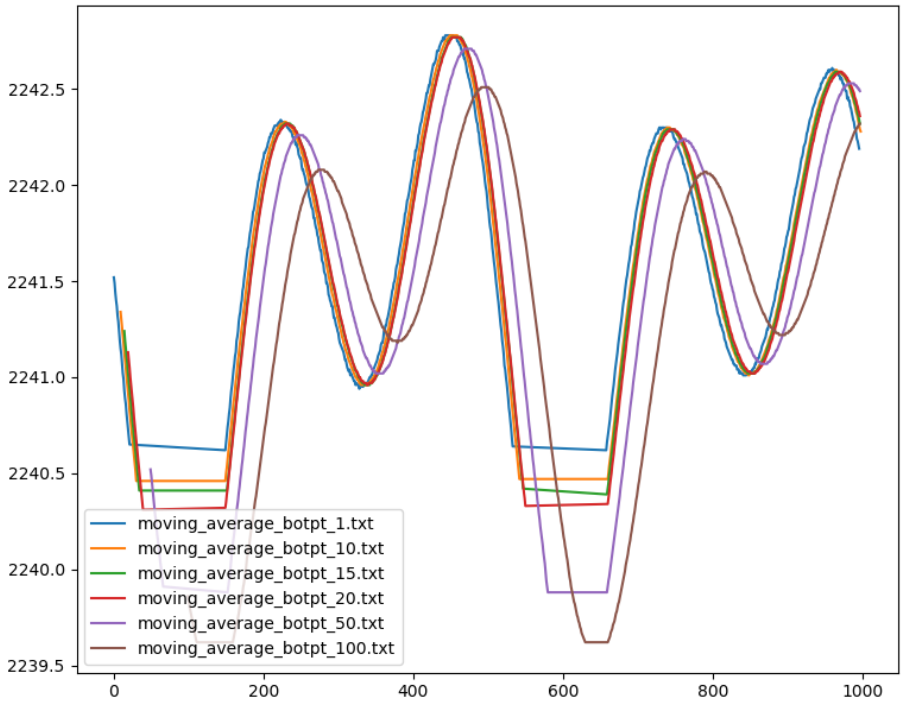

Figure 3.3: *Smoothing for BOTPT sensor measurements*