CS546 HOMEWORK 5

SERGIO PENAVADES SUAREZ
CWID: A20387929

1. Introduction

The purpose of this document is to show how I've improved the performance of the matrix normalization algorithm using CUDA. Additionally, I will explain how the code works and the code pieces added.

2. Process of implementation

In this homework I analyzed different ways of implementing a parallel program using CUDA, identifying the complexity of them and how extra instructions improve the applications performance. With the analysis I acquired the enough knowledge to transform the sequential coded provided in the new parallel code.

With a global vision of CUDA, I studied the provided code to understand everything inside and I developed the parallel code, making tests, measuring execution times and ensuring that the code worked well.

Due of the impossibility of launch the algorithms coded in CUDA in my personal computer, I tested all cases in Jarvis.

3. GPU vs CPU

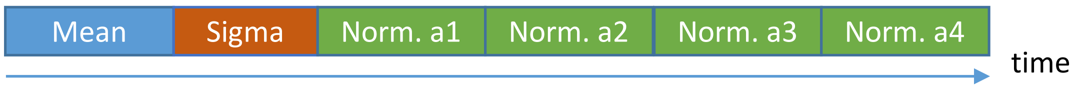
3.1. First approach

As first approach I divided the matrix of N x N as following:

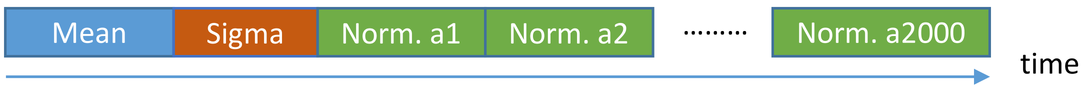
- I have one grid that contained the whole matrix
- The grid has N blocks, one per column
- Each block has just one thread

For this case, each block (one thread) has to calculate the column mean, the column standard deviation and each row normalization. Because of that, each thread has to execute a big amount of operations, so if the matrix has too many rows the number of operation increases. To illustrate it I provide the next example:

For a matrix (4 x 4) each thread (column) has to execute the next operations.



I discarded this option because as we can see in the diagram above, as much rows in the matrix as much time the execution was going to take. For example, if we imagine a matrix with (2000 x 2000), the number of normalizations will be 2000 per thread, so the increment of time will be significant.



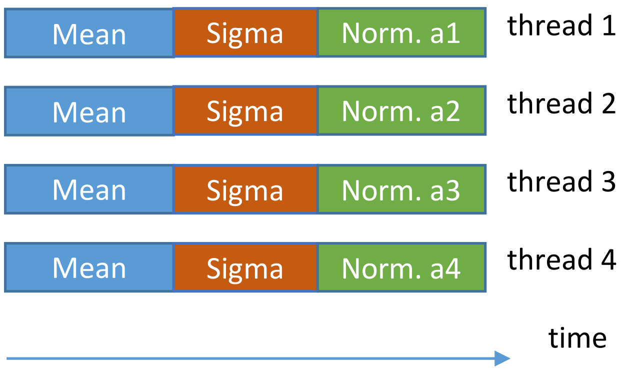
3.2. Final approach

In this final approach I divided the matrix of N x N as following:

- I have one grid that contained the whole matrix
- The grid has N blocks, one per column
- Each block has N threads, one per thread

Thanks of this new approach, each block is divided in N threads and each thread has to calculate the column mean, the column standard deviation and its row normalization. As result, each thread has to execute a lesser amount of operations which improve the first approach performance. To illustrate it I provide the next example:

For a matrix (4 x 4) we have one thread per matrix cell (N per column) that has to execute the next operations.

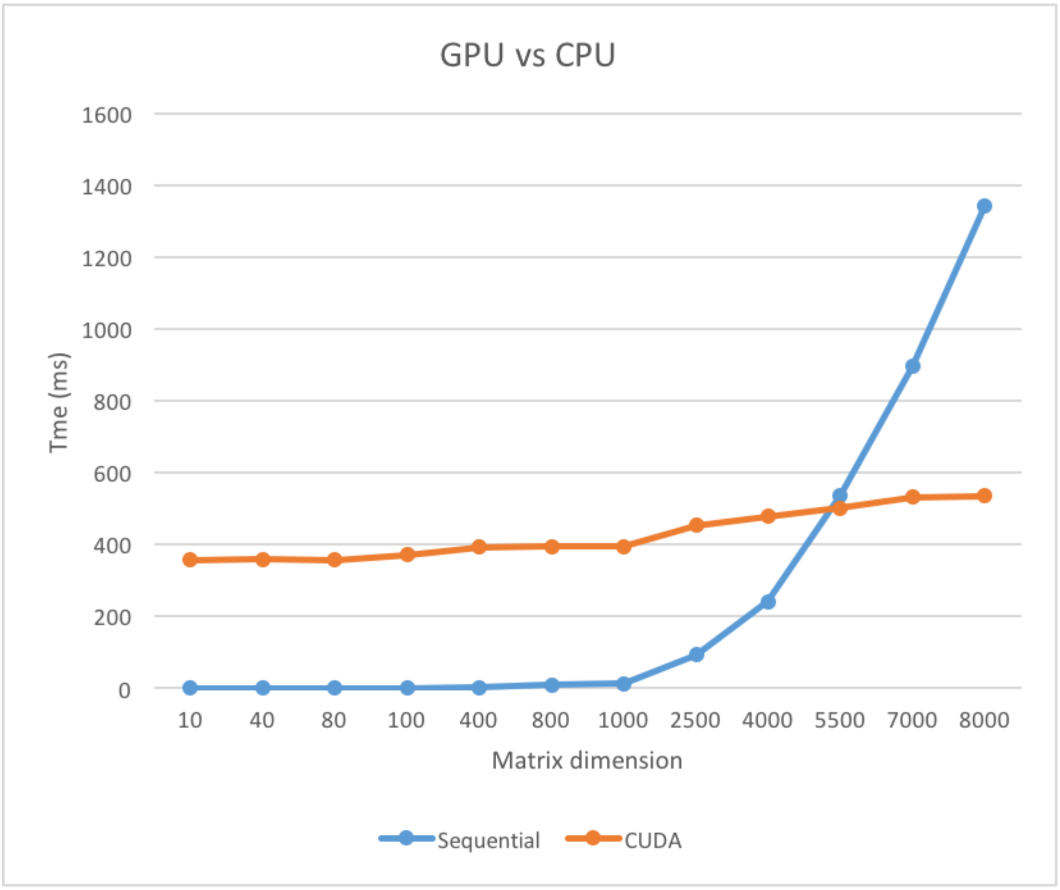


As we can see in the diagram above for a 4 x 4 matrix, if we compare it with the first approach diagram, each thread has to execute 3 normalizations less which derives in saving time. If we Imagine a matrix with (2000 x 2000), the number of normalizations will be 1 per thread, in stead of 2000 so the improvement of the normalization time will be 2000/1, so two thousand times better. Because of these reasons, I chose this second approach as my final approach.

The next table includes time execution with different matrix dimensions in both codifications, sequential an CUDA:

	Sequential time (ms)	**CUDA time (ms)**	**Speed Up**
10	0.023	357.879	-
40	0.076	359.765	-
80	0.302	357.468	-
100	0.458	372.219	-
400	2.285	393.973	0.005
800	9.043	394.549	0.022
1000	14.115	394.879	0.035
2500	92.682	454.657	0.206
4000	241.23	479.864	0.503
5500	538.2	502.646	1.071
7000	896.11	532.45	1.684
8000	1345.59	536.681	2.510

In the graph below I show the behavior of this implementation and how it improves the original code:



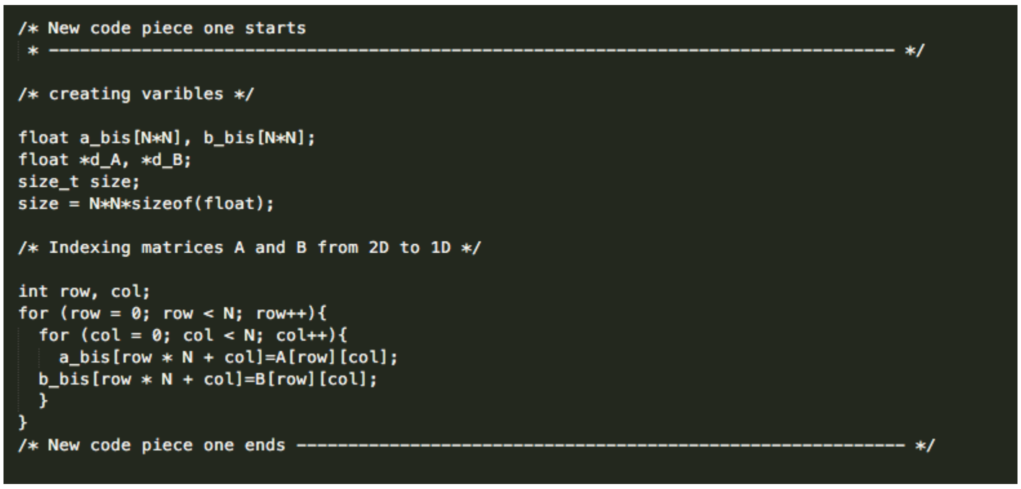
In the graph it is clear that CUDA improves the performance once the matrix dimension reaches the value 5500 approximately. As we can see, the elapsed time for the CUDA code is more or less constant for the matrix dimensions provided. Due to that I can assert that when the amount of calculations increases the CUDA is more efficient because all the calculations are distributed parallel among all the GPU cores. As consequence the speedup is improved from N=5500 approximately too.

We can explain all of above by following that CUDA as much symmetry the data has as much performance is given by the GPU because of its computational features.

4. Code added and algorithm modified

For each piece of code added I include a screenshot to show the code.

4.1. Piece of code 1



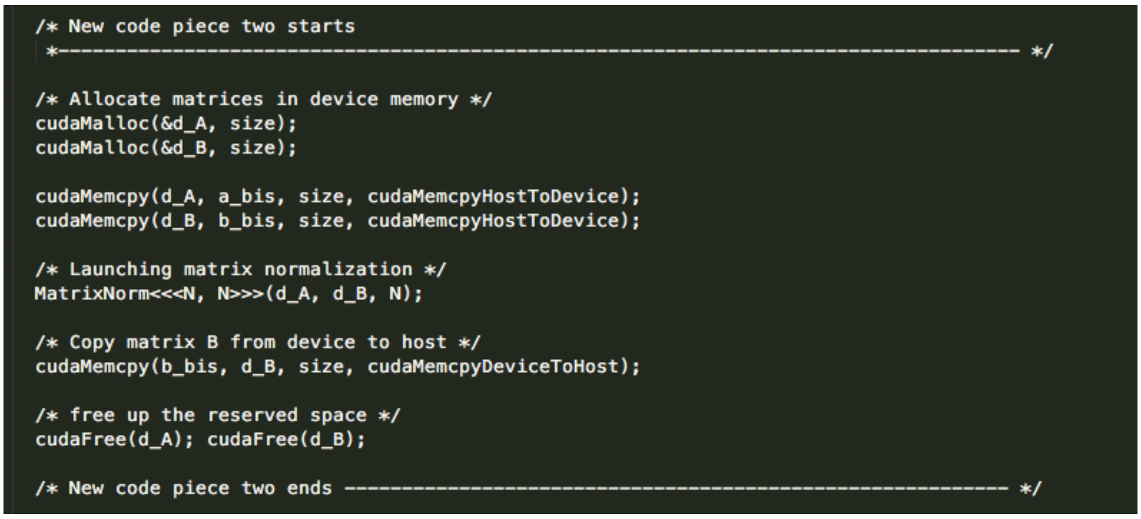
```
/* New code piece one starts
 * ---------------------------------------------------------------------------------- */

/* creating varibles */

float a_bis[N*N], b_bis[N*N];
float *d_A, *d_B;
size_t size;
size = N*N*sizeof(float);

/* Indexing matrices A and B from 2D to 1D */

int row, col;
for (row = 0; row < N; row++){
  for (col = 0; col < N; col++){
    a_bis[row * N + col]=A[row][col];
  b_bis[row * N + col]=B[row][col];
  }
}
/* New code piece one ends ------------------------------------------------------------ */
```

In the piece of code above, due to the impossibility of using 2D arrays in CUDA I created two new 1D arrays called “a_bis” and “b_bis”. So, the purpose of this piece of code is to index the provide matrixes in the 1D arrays.

4.2. Piece of code 2

In this second piece of code is added inside the “main function”. This code has the next steps:

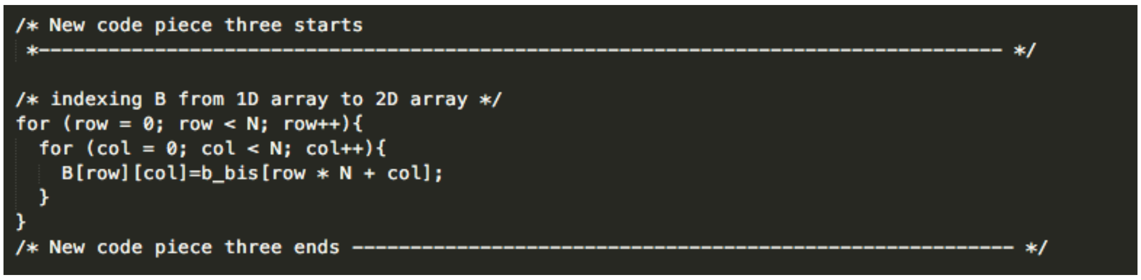
1. Reserve the needed memory space to allocate the result
2. Send the matrixes from de host to the GPU devices
3. Execute the matrix normalization function
4. Send the matrix from the GPU devices to the host
5. Free up the reserved space



```
/* New code piece two starts
 *---------------------------------------------------------------------------------- */

/* Allocate matrices in device memory */
cudaMalloc(&d_A, size);
cudaMalloc(&d_B, size);

cudaMemcpy(d_A, a_bis, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, b_bis, size, cudaMemcpyHostToDevice);

/* Launching matrix normalization */
MatrixNorm<<<N, N>>>(d_A, d_B, N);

/* Copy matrix B from device to host */
cudaMemcpy(b_bis, d_B, size, cudaMemcpyDeviceToHost);

/* free up the reserved space */
cudaFree(d_A); cudaFree(d_B);

/* New code piece two ends --------------------------------------------------------- */
```

4.3. Piece of code 3

This third piece of code is executed after the matrix normalization and it purpose is to index the B matrix from the device to the B matrix provided by the formulation.

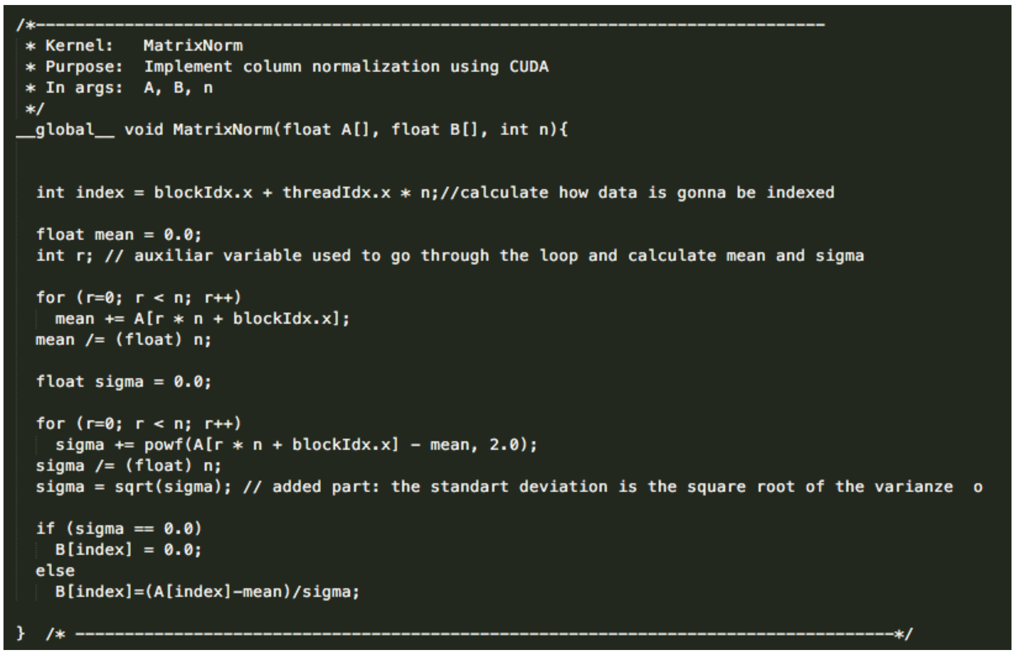


```
/* New code piece three starts
 *---------------------------------------------------------------------------------- */

/* indexing B from 1D array to 2D array */
for (row = 0; row < N; row++){
  for (col = 0; col < N; col++){
    B[row][col]=b_bis[row * N + col];
  }
}
/* New code piece three ends -------------------------------------------------------- */
```

4.4. Parallel algorithm

The code below is the algorithm parallelized. It is important to highlight that first of of all, the algorithm calculates the mean. Second, is calculates the standard deviation which is dependent of the mean. And finally, it calculates the normalization which is dependent of the mean and de standard deviation.

As I told in previous parts of this report, the CUDA coda has N blocks and N threads per block, so in the index variable I recalculate where the data has to be indexed in the matrix.

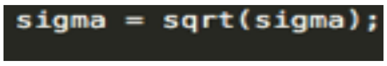


```
/*-------------------------------------------------------------------------------
 * Kernel:   MatrixNorm
 * Purpose:  Implement column normalization using CUDA
 * In args:  A, B, n
 */
__global__ void MatrixNorm(float A[], float B[], int n){


  int index = blockIdx.x + threadIdx.x * n;//calculate how data is gonna be indexed

  float mean = 0.0;
  int r; // auxiliar variable used to go through the loop and calculate mean and sigma

  for (r=0; r < n; r++)
    mean += A[r * n + blockIdx.x];
  mean /= (float) n;

  float sigma = 0.0;

  for (r=0; r < n; r++)
    sigma += powf(A[r * n + blockIdx.x] - mean, 2.0);
  sigma /= (float) n;
  sigma = sqrt(sigma); // added part: the standart deviation is the square root of the varianze  o

  if (sigma == 0.0)
    B[index] = 0.0;
  else
    B[index]=(A[index]-mean)/sigma;

}  /* ---------------------------------------------------------------------------------*/
```

5. Algorithm improvement

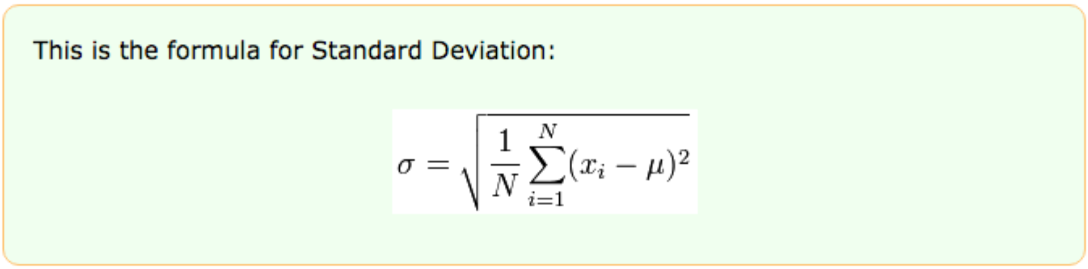
Following the instructions provided for this homework, in the second step the algorithm calculates the standard deviation. But in the sequential code provided, it in not calculating the standard deviation, it is just that calculates the variance.

Therefore, I added the next line to the code:



```
sigma = sqrt(sigma);
```

Because the standard deviation is the square root of the variance.



This is the formula for Standard Deviation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

6. Conclusion

Thanks to CUDA it is possible to improve the performance over codes that present parallelism symmetry. Because in codes which it is possible to distinguish among blocks data that are independent in the algorithm executions, if we separate these blocks and execute them at the same time, there is a clear improvement in implementing parallel solutions with respect to other studied methods.

Likewise, it is important to highlight the importance that how GPU works. A GPU has the advantage to execute separately several threads at the same time but it is necessary to think about what piece of data is going to be executed by each GPU thread. Therefore, it is necessary to make a deep study of the code to parallelize and be aware of its features (loops, exchange of data, data synchronization, etc..) to implement an adequate parallelization.

7. References

[1] Matrix multiplication with CUDA
https://www.shodor.org/media/content/petascale/materials/UPModules/matrixMultiplication/moduleDocument.pdf

[2] Implement matrix addition on a gpu using cuda
http://www.cs.usfca.edu/~peter/cs220/code/cuda/mat_add.cu

[3]Easy introduction to CUDA in C
https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/

[4] Programando la GPU con CUDA
http://dmi.uib.es/~josemaria/files/CUDA-v36-ESP-para-UIB.pdf

[5] Allocating 2D arrays in CUDA
http://www.stevenmarkford.com/allocating-2d-arrays-in-cuda/

[6] An Efficient Matrix Transpose in CUDA C/C++
https://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/

[7] CUDA Thread basics
http://users.wfu.edu/choss/CUDA/docs/Lecture%205.pdf

[8] Difference Between CUDA and OpenCL
https://streamcomputing.eu/blog/2010-04-22/difference-between-cuda-and-opencl/