Matrix Normalization with CUDA

Source code for this homework can be find in the archive directory, along with a README file.

Introduction

The goal of this project is to study the parallelization with CUDA of the normalization of a matrix of size `N`. equation (where $A \in M_N(\mathbb{R})$). Several solutions have been implemented and tested on Jarvis cluster. Those solutions have been compared to a given serial algorithm.

About Normalization

The given algorithm hands in three steps, for each column:
(1) Computation of μ;
(2) Computation of σ;
(3) Normalization using μ and σ.

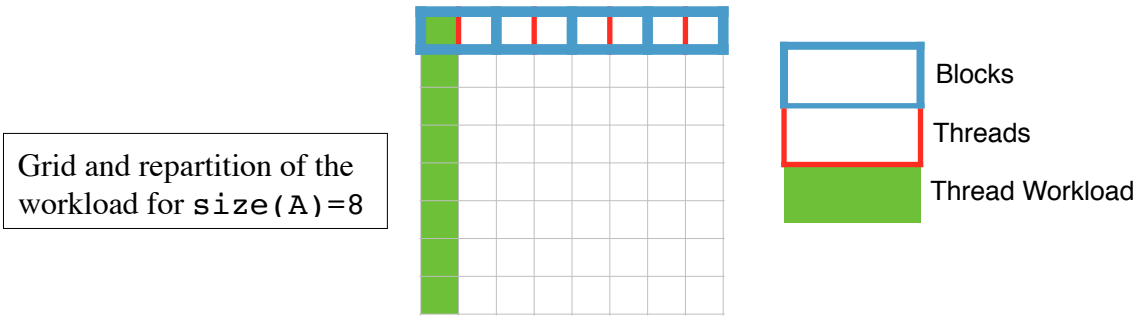
Each step involves `O(N)` operations per column so the global algorithm involves $O(N^2)$ operations. Besides, we can notice a few things about this algorithm, which will have implications on the parallelizing methods and will limit them:

- There is no dependance between columns workload;
- The steps of the algorithm need to be processes in this exact order, because they all depend on previous ones.

Solutions

1. One thread per column

In this implementation I decided to divide the matrix per columns, with one thread computing the workload for one column. In order to do that, the grid had to cover the width of the matrix, according to the following figure:



Thus, each thread only has `O(N)` assigned operations.

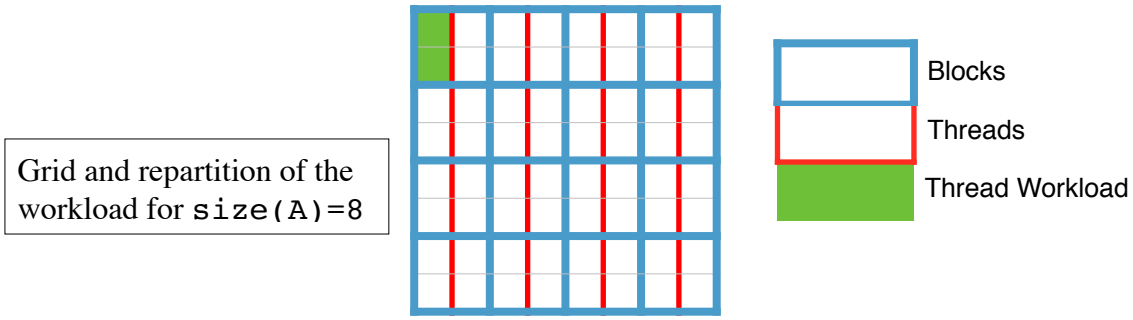
2. Splitting into multiple kernels

The second method aims at sharing the workload for each step. The repartition of the threads is the same than for method 1, but instead of computing in once μ, σ and then final matrix, the main process will call 1 kernel function for each of these steps.

Theoretically, this method should be slower than method 1, because the workload per thread is the same but the memory traffic is higher (more kernel functions are called). This method is mostly a step toward method 3.

3. Covering the entire matrix with blocks

For this last method, I wanted to maximize the number of threads so the GPU could be used more effectively. However, as the computation of μ and σ is specific to each column, it is more interesting to keep each thread on a single column. Thus, the repartition of the workload will be as in the next figure:



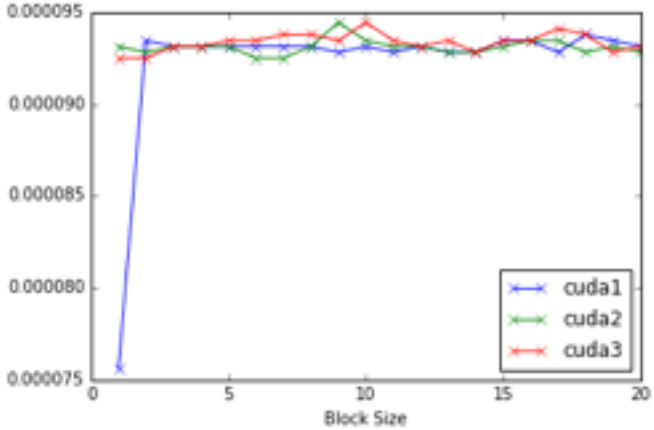
Grid and repartition of the workload for `size(A)=8`

Theoretically, we create this time gridDim more threads and then divide the workload per the same number. Nevertheless, doing that force us to compute the local μ and σ for each thread in two more arrays `S` and `M`, and then sum all μ and σ for each column. However, this cost is negligible compared to gain obtained with the grid covering.

Tests and results

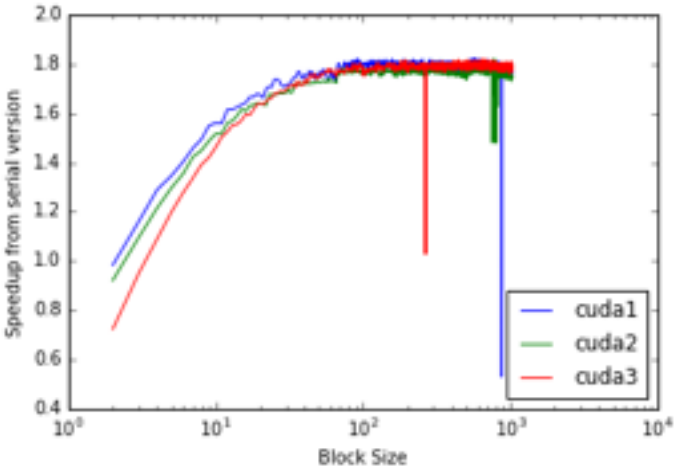
Each of these methods have been tried on a small matrix (`size=20`) and a big one (`size=8000`) for every possible `blockDim` from `1` to `1024`.

On the small matrix, with no surprise, the serial method is faster (we have a very low speedup), according to the next figure (y axe corresponds to the speedup compared to the serial method):



We can see that all three methods seem to have the same (low) speed. We can understand that because there is a lot of time lost is memory traffic and thread creation/disparition.

On the big matrix, we have a maximum speedup around 1.8, as shown in the following figure:



Unfortunately, none of the three methods seems to be significantly better than another, that could be explained thanks to memory traffic, but I still do not understand why the third method does not show any significant speedup.

However, we can see that the blocks size can be optimized. A good value seems to be around 100.

Conclusion

In conclusion, although we have a nice speedup compared to a full CPU processing, I expected a bigger one, especially with the third method.

I did not have the time to test these methods on several GPUs but still a lot of time seems to be lost in memory traffic, although it is quite minimal in the implementation of the third method.