Università degli Studi di Siena

Project title: A way to compute PCA analysis on CUDA

Students: Giacomo Nunziati (matr. 103246)
Christian Di Maio (matr. 104985)

Professor: Roberto Giorgi

Introduction

1 Overview

The objective of the project is to compare the performance of different systems equipped with one or more Nvidia GPU accelerator (for more details, refer to the appendix). The chosen benchmark is the algorithm to perform principal component analysis (PCA) of a data set.
The algorithm has been developed in the CUDA environment and implemented in various different ways, in order to highlight the different performance outcomes that can be obtained when the various architectural elements of the GPU are used.
Once the algorithm have been implemented they have been executed on the selected systems and the performance results have been measured and compared.

2 PCA

The principal component analysis is looking for a way to reduce the dimensionality of a given feature space. [JC16]
The goal of the project is to find, given a data set represented by an $N \times M$[1] matrix, all the eigenvectors of the co-variance matrix of the input data, sorted from the most important to the less one. The method that we apply to perform the PCA is basically divided in 3 steps:

- Co-variance matrix calculation.
- Eigenvalues associated to the co-variance matrix.
- Given each eigenvalue find the associated eigenvector.

Assuming that we have Input as input matrix, with a form like this:

$$Input = \begin{bmatrix} i_{11} & i_{12} & \dots & i_{1M} \\ \vdots & \ddots & & \\ & & & \\ i_{N1} & & & i_{NM} \end{bmatrix}$$

[1] N-rows,M-columns

2.1 Co-variance matrix calculation

Given the maximum and the mean of each column, respectively max_j and $mean_j$, of the matrix Input, the calculation of the co-variance matrix is done by evaluating the normalized input matrix, accordingly with the following formula:

$$i'_{ij} = \frac{i_{ij} - mean_j}{|max_j|}$$

$$NormalizedInput = \begin{bmatrix} i'_{11} & i'_{12} & \dots & i'_{1M} \\ \vdots & \ddots & & \\ & & & \\ i'_{N1} & & & i'_{NM} \end{bmatrix}$$

After normalizing the columns of the input matrix, the co-variance matrix can be obtained multiplying NormalizedInput by itself:

$$CovMat = NormalizedInput^T \times NormalizedInput$$

So at this point we have the Co-variance Matrix with dimension $M \times M$[2].

$$CovMat = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1M} \\ \vdots & \ddots & & \\ & & & \\ c_{M1} & & & c_{MM} \end{bmatrix}$$

[2] M columns of input matrix

2.2 Find Eigenvalues with Jacobi method

Starting from the co-variance matrix we start a loop on which we iterate the following steps [Qua+14b]:

- Step 0:
$$A^{(1)} = CovMat$$

- Step 1: Find over the upper triangle of the matrix $A^{(k)}$ the maximum value (absolute) $max_{rs}^{(k)}$ on which r and s are respectively the row and the column index.
  Given r and s calculate:

$$m^{(k)} = \frac{a_{rr}^{(k)} - a_{ss}^{(k)}}{2a_{rs}^{(k)}}$$

$$t = min\{-m^{(k)} + \sqrt{1 + (m^{(k)})^2}, -m^{(k)} - \sqrt{1 + (m^{(k)})^2}\}$$

- Step 2: Calculate the rotation matrix

$$G_{rs}^{(k)} = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & \cos\phi & \sin\phi & & \\ & & -\sin\phi & \cos\phi & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix} \begin{matrix} \\ \\ \boldsymbol{r} \\ \boldsymbol{s} \\ \\ \\ \end{matrix}$$

(columns $\boldsymbol{r}$ and $\boldsymbol{s}$ marked above the $\cos\phi$ / $\sin\phi$ columns)

  Where

$$\cos\phi = \frac{1}{\sqrt{1-t^2}}$$
$$\sin\phi = t\cos\phi$$

- Step 3: Compute the new Matrix $A^{(k+1)}$,

$$A^{(k+1)} = G_{rs}^{(k)^T} \times A^{(k)} \times G_{rs}^{(k)}$$

- Step 4: Go To *Step 1* ***until***

$$\max_{\forall a_{ij} \in A^{(k+1)}, i>j} |a_{ij}^{(k+1)}| \leq \varepsilon$$

At this point we have on the diagonal of A all the eigenvalues of the co-variance matrix.
To be faster and save some computational resources, we collect only the eigenvalues which store the 99% of information, otherwise we have a very low valued eigenvalues which is useless with respect of our main goal.

2.3 Find Eigenvectors with inverse power method

At this point we have all the *most important eigenvalues*, the way on which we retrieve the associated eigenvector is the following [Qua+14a]:
Given,

$$\lambda_{i \in [1,N]}$$

$$C \text{ defined as a } M \times M \text{ Matrix,}$$

where λ_i is the i-th eigenvalue and C is the co-variance matrix, *iterate* over each λ and do the following:

- Step 0: Initialize a RHS vector v, calculate the matrix

$$M_i = C - \lambda_i I$$

- Step 1: Solve the linear system

$$M_i x^{(k)} = v$$

- Step 2: Re-calculate the RHS vector v as

$$v = \frac{x^{(k)}}{||x^{(k)}||}$$

- Step 3: Calculate the approximate value of $\tilde{\lambda_i}^{(k)}$,

$$\tilde{\lambda_i}^{(k)} = v^T \times A \times v$$

- Step 4: Go to *Step 1* ***until***:

$$|\tilde{\lambda}_i^{(k)} - \tilde{\lambda}_i^{(k-1)}| < \varepsilon \;\; \& \;\; |\tilde{\lambda}_i^{(k)} - \lambda_i| < \varepsilon$$

When we complete this cycle we store the eigenvector into a matrix and move on to the next eigenvalue.

2.4 Find Eigenvectors with Jacobi method

In the third version of the program, the rotation matrices computed while approximating the eigenvalues are also used to simultaneously compute the eigenvectors of the co-variance matrix. In fact the transformations applied in the Jacobi methods are consecutive similarity transformations and so, the final matrix, that is diagonal, can also be obtained by applying an equivalent similarity transformation, by mean of the product matrix of all the transformation matrices. Since that matrix transforms the input matrix in a similar, diagonal matrix, its rows/columns are the eigenvectors of the initial matrix.

2.5 Comments on the two methods

Comparing the two methods for computing the eigenvalues, it comes out that the first method is slower but more accurate, while the second method is faster, but unstable and affected by numerical error. In fact, for big data sets, the numerical error is propagated on each matrix product, and the result is that the computed eigenvectors are completely wrong. This behavior is particularly evident with ill-conditioned input co-variance matrix.

3 The method used for taking measures

The method that we use to measure the time to execute a task is based on the *getTimeOfDay*[IG04] function, which allow us to retrieve the system's clock time. So, for counting the elapsed time for doing a task, we take a starting and an ending time, we obtain the effective execution time by making the difference of them. We measure 4 different parts of the code, suggested by the structure of the algorithm:

- Time for evaluating the normalization matrix.
- Time for calculating co-variance matrix.
- Time for retrieving the eigenvalues [3].
- Time for evaluating the eigenvectors.

We take also into account the fact that one single execution may not not sufficient for making a good report, so for each experiment we do the measurement 10 times[4] and collect:

- The mean execution time.
- The standard deviation.
- The minimum time of execution.
- The maximum time of execution.

The data set that has been used is composed of 561 columns and 7767 rows of real numbers.
For each version of the program and for each machine, the experiment has been repeated 7 times, considering different subsets of columns.

[3]Only for CPU and GPU1 version, because in GPU2 the eigenvalues calculation is indeed calculated inside the eigenvectors finder method.

[4]The only exception is for the CPU version: for an high dimensionality feature space we decide to do just one iteration, because the CPU version is far slower compared to the gpu's one.

4 Result

The first result that we want to emphasize is the execution time compared on the 3 different versions of PCA, executed on our best and worst machine (fig.1).
We can see that with the CPU version the execution time is very similar, the things start to be different when we compare the GPU based versions, on which we can see that the acceleration offered by GPU decrease the overall execution time.
Another observation is that whenever we encounter small feature spaces dimension data set, the overhead generated by making the execution heterogeneous[5] hides the speed-up offered by the GPU. The last consideration is that looking asymptotically the graph, we can see that the steepness of the CPU version is higher, compared to the GPU ones, so the speed-up is asymptotically increasing.

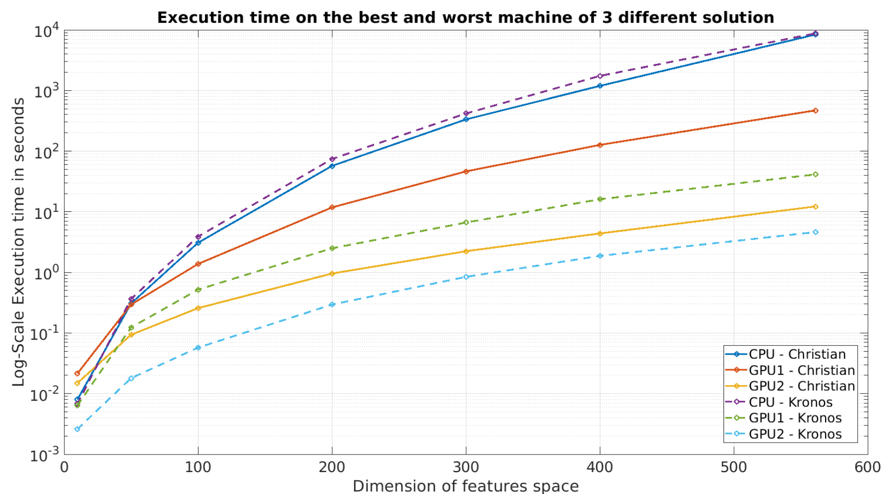

Figure 1: Comparison among GPU and CPU versions (on the best and worst machine

[5]e.g. data transfer from/to CPU-GPU

A slightly different result is obtained by looking at the overall execution time measured on all the available machines (3). As we can see the GPU version 2 is globally faster than GPU version 1, moreover performance of version 1 is less sensitive with respect to the number of CUDA cores (see AXM4 and Kronos lines).
Although the speed-up gained on bigger feature spaces is much more evident on GPU1 than GPU2.

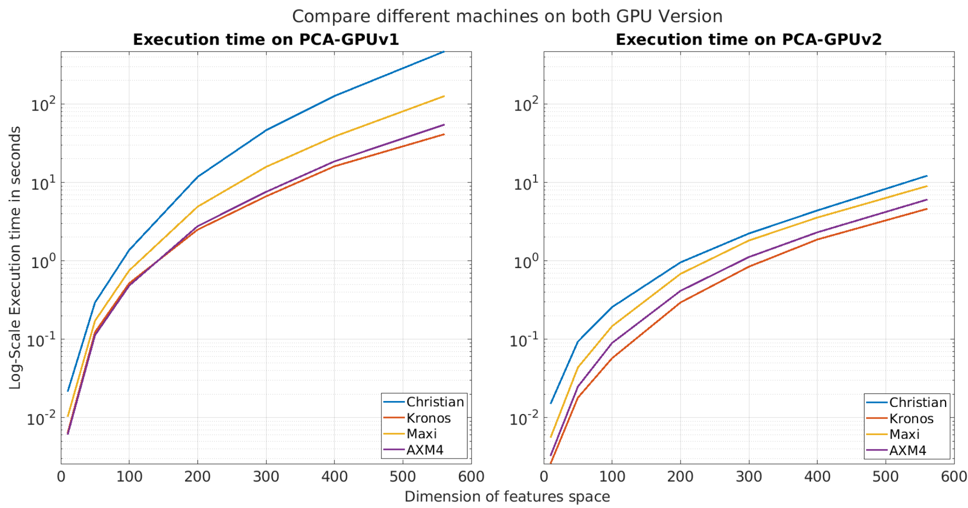

Figure 2: Comparison between different GPU versions on all the machine.

To validate the results that we find, we calculate (as reference) all the eigenvectors on MATLAB. For seeing if our program works well we calculate the norm of the absolute value given by the difference of the result coming from the 2 GPU versions and MATLAB (fig.2).
As we can see, with a small dimensional feature spaces both GPU1 and GPU2 give us good results, but as the spaces of the data set increase, we lose a lot of precision on the version 2. In specific, the figure shows us the mean of the error norm as a line over the number of features, we also show for each experiment the maximum error that we collect (horizontal line connected to the associated mean).

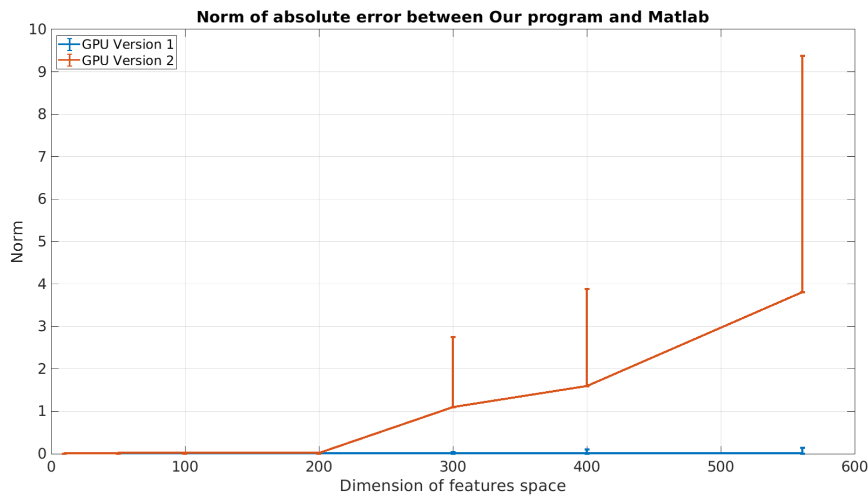

Figure 3: Comparison among GPU1 and GPU2 versions on the obtained eigenvectors

For fig. 4 it is clear that most of the execution time is spent on computing the eigenvalues and eigenvectors of the matrix, while the normalization and the computation of the co-variance matrix take a small portion of the total execution time. It is also clear that the speedup of the GPU version 2 is totally due to the eigenvalues and eigenvectors computation task, since the first two steps are identical in the two programs. For the right graph is evident (but it is true also for GPU version 1) that the steps of normalization and computation of the co-variance matrix scale better that the other tasks, with increasing the number of CUDA cores.

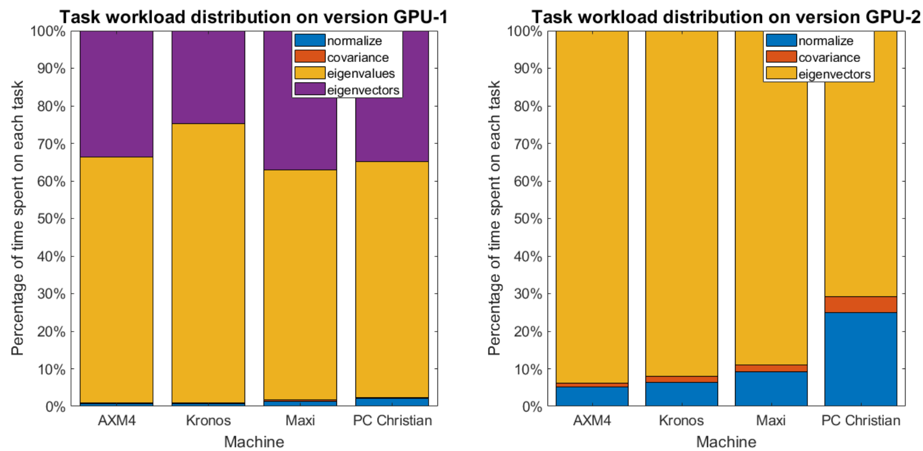

Figure 4: Time spent in each tasks, measured on the best and worst machine, using 200 columns as dimension of the feature space

Fig 5 shows how the time spent in each task by different machines changes with the dimension of the feature space. For the two graphs of GPU version 2, we can deduce that the task of computation of eigenvectors scales worse, not just increasing the number of available cores, but also increasing the dimension of the problem. In the graphs of GPU version 1 it is shown that the two tasks for computing the eigenvalues and eigenvectors have an atypical behaviour. They both scale worse than the other tasks. However, while for small values of dimension of the problem the last step seems to scale better, for high values of it, the computation of the eigenvectors requires more and more of the total execution time. This is particularly evident on the results of the Kronos machine, and this suggests that the computation of the eigenvectors scales worse than the computation of the eigenvalues, with increasing the number of available cores. Moreover, the process seems not to be about to stop: we could hypothesize that for even higher dimension of the feature space, most of the time is spent on computing the eigenvectors.

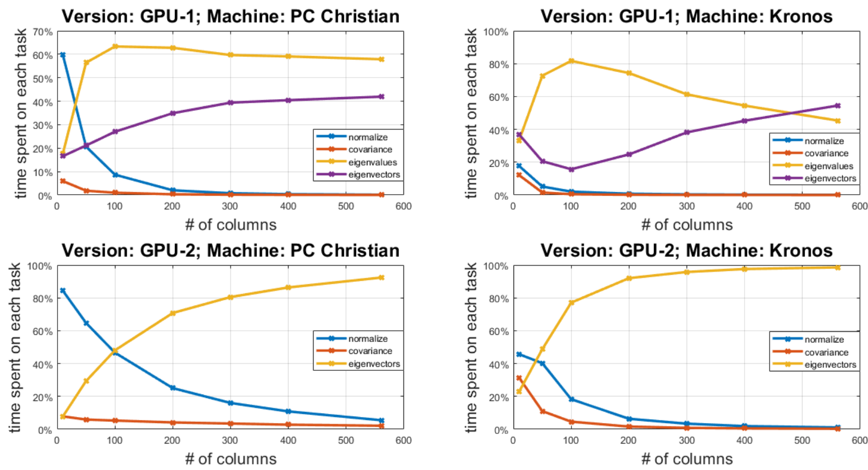

Figure 5: time spent in the different tasks over of columns.

5 Conclusion

The main goal of the project has been reached: two GPU based solutions has been proposed and tested on different machines, showing that they are more performing than the simple scalar solution, although one of them is numerically unstable for high dimensional feature spaces.
As expected, machines provided with more CUDA cores in their GPU are more performing, so the programs are scalable.
Nevertheless, using a GPU could not always be the best choice, since for small dimensional feature spaces, CPU and GPU based programs have very similar performance.

A Specifications of the machines used in the experiments

				GPU				
Name	Operating System	CPU	RAM(GB)	Model	Cuda Ver.	Cuda Cores	Architecture	Device RAM (GB)
Pc Christian	Ubuntu 20.10	Intel i7-5500U (4) @ 3.000GHz	12	GeForce 940M	11.2	384 @ 1.18 GHz	Maxwell	2
Kronos	Ubuntu 18.04	Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz	48	2x Tesla V100-SXM2	11.0	5120 @ 1.37 GHz	Volta	32
Maxi	Debian 10	Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz	64	GTX970	8.0	1664 @ 1.05 GHz	Kepler	4
AXM4	Ubuntu 20.04	AMD Ryzen Threadripper 1950X @ 3.4 GHz	128	TITAN-Xp	11.2	3840 @ 1.41 GHz	Pascal	12

Figure 6: Table of hardware specs.

References

[IG04] The IEEE and The Open Group0. *The Open Group Base Specifications Issue 6*. publisher: IEEE Std 1003.1, 2004 Edition. 2004. URL: https://pubs.opengroup.org/onlinepubs/009604599/functions/gettimeofday.html (visited on 04/13/2021).

[JC16] Ian T. Jolliffe and Jorge Cadima. "Principal component analysis: a review and recent developments". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374.2065 (2016), p. 20150202. DOI: 10.1098/rsta.2015.0202. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2015.0202. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2015.0202.

[Qua+14a] Alfio Quarteroni et al. "Il metodo delle potenze". In: *Alfio Quartieroni - Matematica Numerica*. 4th ed. Springer, 2014, pp. 174–179.

[Qua+14b] Alfio Quarteroni et al. "Metodi per il calcolo di autovalori di matrici simmetriche". In: *Alfio Quartieroni - Matematica Numerica*. 4th ed. Springer, 2014, pp. 203–207.