

# Cloud computing Project: Principal Component Analysis

TOUZI Mohamed, MOKHTARI Wissam

May 20, 2023

## 1 Introduction

Principal Component Analysis (PCA) [2] is a widely used statistical technique for dimensionality reduction and data analysis. It is particularly useful when dealing with high-dimensional data by identifying patterns and extracting the most important features or components.

The main goal of PCA is to transform a set of potentially correlated variables into a new set of uncorrelated variables called principal components. These components are ordered in such a way that the first component captures the maximum amount of variance in the data, the second component captures the maximum remaining variance orthogonal to the first component, and so on.

In this project, our goal is to explore and compare different methods of PCA execution, specifically focusing on the implementations using Central Processing Units (CPU) and Graphics Processing Units (GPU).

The choice between CPU and GPU implementations of PCA is crucial, especially when dealing with large datasets or computationally intensive tasks. GPUs excel at parallel processing, making them well-suited for handling complex computations in a highly efficient manner. By leveraging the power of GPUs, we can potentially accelerate the PCA process and achieve faster results compared to CPU implementations.

## 2 PCA Methods

### 2.1 Classic PCA

It's often used to visualize genetic distance and relatedness between populations [1]. Here are the steps for PCA:

- **Standardize Data:** Normalize your data so all features have a mean of 0 and variance of 1.
- **Calculate Covariance Matrix:** Determine how different features in your data vary with respect to each other, capturing any relationships between them.
- **Compute Eigenvectors and Eigenvalues:** Using linear algebra, compute these from the covariance matrix to find the principal components, or the major patterns in your data.
- **Select Principal Components:** Rank the principal components in order of their significance or explanatory power, and choose the top ones to keep, depending on your needs.
- **Form Projection Matrix:** Concatenate the top eigenvectors into a matrix, which will be used to transform the original dataset to a reduced dimension space.
- **Transform Original Dataset:** Multiply the original dataset with the projection matrix to get a lower-dimension version of your data.

## 2.2 NIPALS Algorithm

The NIPALS algorithm [3] concludes with two key tasks. First, the assembly of the matrix  $T$  and the matrix  $P$ . Each column of  $T$  is constructed from the individual score vectors  $t_h$ , and each column of  $P$  is constructed from the individual loading vectors  $p_h$ :

$$T = [t_1, t_2, \dots, t_h] \quad (1)$$

$$P = [p_1, p_2, \dots, p_h] \quad (2)$$

The NIPALS algorithm for Principal Component Analysis begins by setting the iteration count, denoted as  $h$ , to 1, and initializing  $X_h$  as  $X$ , the original data matrix. The algorithm then follows these steps:

1. Select any column from  $X_h$  as  $t_h$ .
2. Calculate the loadings  $p_h$  for the  $h^{th}$  component, which can be done using this formula:

$$p_h = \frac{X_h^T t_h}{t_h^T t_h}$$

3. Normalize  $p_h$  to unit length:

$$p_h = \frac{p_h}{\|p_h\|}$$

4. Compute the scores  $t_h$  for the  $h^{th}$  component:

$$t_h = \frac{X_h p_h}{p_h^T p_h}$$

5. Repeat steps 3 and 4 until convergence is achieved for the  $h^{th}$  principal component.

6. Update the residuals matrix  $X_{h+1}$ :

$$X_{h+1} = X_h - t_h p_h^T$$

7. Compute the eigenvalue  $\lambda_h$  for the  $h^{th}$  component:

$$\lambda_h = t_h^T t_h$$

8. Increment  $h$  by 1 and repeat steps 1 to 7 for the next principal component.

Finally, the columns of  $T$  are formed by the  $t_h$  vectors (scores), and the columns of  $P$  are formed by the  $p_h$  vectors (loadings).

The resulting Principal Components (PCs) may need to be scaled for certain applications. A common way to scale the PCA solution is to define the loadings  $P$  as  $V$  and scores  $T$  as  $U^T S$ , where  $U$ ,  $S$ , and  $V$  are obtained from the singular value decomposition (SVD) of the original data matrix  $X$ .

## 3 PCA execution under CPU

The classic PCA usually involves a singular value decomposition (SVD) of the data matrix. The NIPALS algorithm, on the other hand, is iterative and does not require calculation of the covariance matrix. It can

also handle missing data, which classic PCA can't do without some form of data imputation. The purpose of executing both methods using Numpy module is likely to compare their performances, in terms of speed and their ability to handle large datasets, under CPU as our first computational resources available .

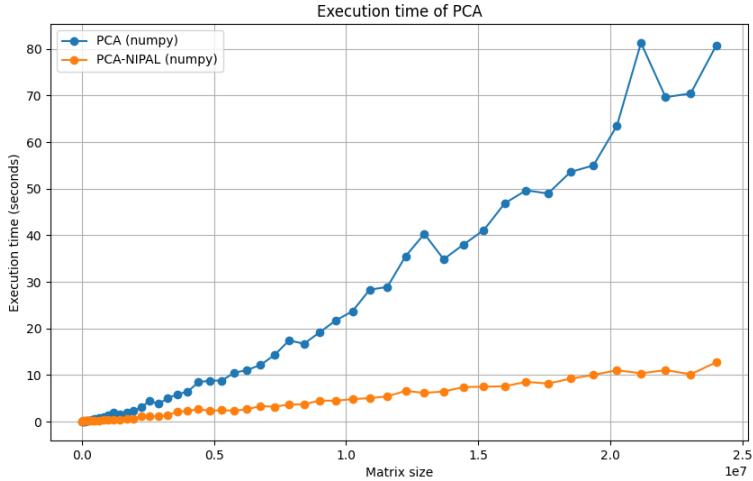


Figure 1: Execution time function of matrix X size

Notably, NIPALS exhibits significantly faster execution times, showcasing its efficiency and scalability with increasing data size.

Note that for the sake of simplification, we are operating under the assumption that the matrices we are working with are square matrices.

## 4 NIPALS under CPU vs NIPALS under GPU

Since NIPALS is the survivor of the last section, we will be assessing the performance of the NIPALS algorithm on a GPU, and contrasting this with the CPU execution times you have already observed.

Given that GPUs are specifically designed for high computational throughput, this comparison will provide valuable insight into whether PCA implementations can be further optimized by leveraging the power of GPU computing.

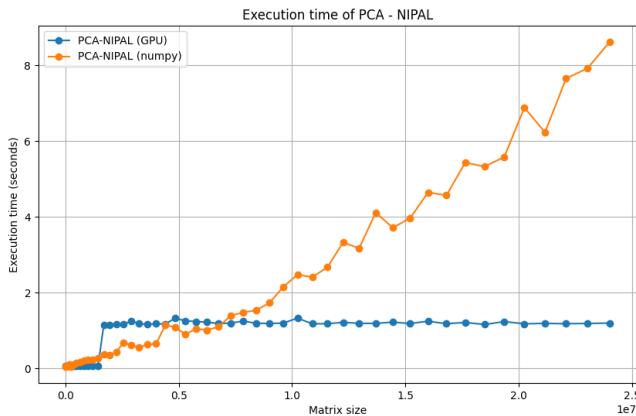


Figure 2: Execution time function of matrix X size Under CPU VS GPU

The graph demonstrates the speed of NIPALS implementation as a function of matrix size under CPU and

GPU. It is evident that the GPU implementation outperforms the CPU implementation in terms of execution time, especially for larger matrix sizes. Also the numpy method outperforms the GPU method for low matrix sizes this is explained by the fact that transferring variables from the CPU to the GPU during the execution process may takes some time. This significant improvement in speed can be attributed to the parallel processing capabilities of GPUs. By harnessing the computational power of the GPU, we can distribute the workload across multiple cores, enabling simultaneous execution of tasks and accelerating the overall computation. As a result, the GPU implementation demonstrates enhanced efficiency and performance compared to the CPU implementation.

However, it is important to note that certain variables must be transferred from the GPU to the CPU during the execution process. This transfer incurs additional overhead and introduces a potential bottleneck in the overall speed of execution. While the GPU implementation showcases faster execution, the time taken for this data transfer can impact the overall efficiency.

Note that the GPU used is a Tesla T4, to run the code and get the plots we just excute the main.py file , also we used SourceModule from pycuda.compiler inorder to write the cuda code.

## Bibliography

- [1] Hervé Abdi and Lynne J Williams. “Principal component analysis”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 2.4 (2010), pp. 433–459. DOI: 10.1002/wics.101.
- [2] Hervé Abdi and Lynne J. Williams. “Principal component analysis”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 2.4 (2010), pp. 433–459.
- [3] Kevin Wright. “The NIPALS algorithm”. In: (2017). URL: [https://cran.r-project.org/web/packages/nipals/vignettes/nipals\\_algorithm.html](https://cran.r-project.org/web/packages/nipals/vignettes/nipals_algorithm.html) (visited on 10/27/2017).