High-Performance Implementation of Robust PCA Using CUDA

Nazife Esra Çiğdem
504231339
cigdemn19@itu.edu.tr

***Özet*—This project explores the application of Robust Principal Component Analysis (RPCA) and demonstrates how GPU acceleration via CUDA can significantly enhance its computational performance. Starting from a mathematically intensive baseline RPCA implementation in Python, the algorithm was translated into C++ and further optimized using CUDA with cuBLAS and cuSOLVER libraries. Performance evaluations on images of varying sizes show that the CUDA implementation drastically reduces processing times compared to the CPU version, enabling efficient decomposition of high-resolution images into low-rank and sparse components.**

I. Literature and Baseline Overview

The main study on which I based this project is robust principal component analysis (RPCA) [1]. This work presents a mathematically rigorous and novel approach to decomposing data into low-rank and sparse components, which has broad applications in areas such as computer vision and signal processing. I chose this paper because it is mathematically intensive, with minimal emphasis on implementation details or extensive code. Its results are foundational and often serve as building blocks for more complex systems, such as video surveillance and anomaly detection applications.

The mathematical model described in the paper has an available Python implementation, which I found on the "Papers with Code" website. [2]

II. Project Contributions and Implementation

The following key tasks were accomplished as part of this project. Each step will be explained in detail, accompanied by pseudocode, in the subsequent subsections:

- The original Python implementation of RPCA was translated into C++.
- The C++ code was then adapted into CUDA to leverage GPU acceleration.
- A test image setup was developed, resizing images to four different scales. Both the C++ and CUDA implementations were executed with varying iteration counts to evaluate performance.

A. C++ Code

The Python implementation of the RPCA algorithm is available at the following GitHub repository [3]. Below is the pseudocode representing the core algorithm which is RPCA initialization and iterative procedure on given data:

Algorithm 1: RPCA initialization and iterative procedure: - Pseudocode

Input : Data matrix D, tolerance tol, maximum iterations max_iter
Output: Low-rank matrix L, sparse matrix S
Initialize sparse matrix S, dual variable Y, parameters μ, λ;
Set $iter = 0$, $err = \infty$;
while $err > tol$ *and* $iter < max\_iter$ **do**
  $L \leftarrow \text{SVD\_Threshold}(D - S + \frac{1}{\mu}Y, \frac{1}{\mu})$;
  $S \leftarrow \text{Shrink}(D - L + \frac{1}{\mu}Y, \frac{\lambda}{\mu})$;
  $Y \leftarrow Y + \mu(D - L - S)$;
  $err \leftarrow \|D - L - S\|_F$;
  $iter \leftarrow iter + 1$;
  if $iter$ *mod print_interval = 0* ***or*** $err \leq tol$ **then**
    Print iteration number and error;

The RPCA class is then applied to images resized to different dimensions using OpenCV. The output images, which resemble those in the baseline paper, are generated accordingly. The algorithm for this process is outlined below.

Algorithm 2: Image Resizing, RPCA Application, and Result Image Saving

Input : Original grayscale image `img`, list of target image sizes $\{s_1, s_2, \ldots, s_n\}$, RPCA parameters
Output: Foreground and background images for each size
foreach *size* s *in* $\{s_1, s_2, \ldots, s_n\}$ **do**
  Resize image `img` to $s \times s$ using OpenCV;
  Convert `img_resized` to data matrix D;
  Initialize RPCA with matrix D;
  Run RPCA algorithm on D to obtain sparse matrix S and low-rank matrix L;
  Convert S and L back to images `foreground` and `background`;
  Save `foreground` and `background` images with filenames indicating size s;

The resulting images and performance metrics will be

discussed in the following section.

B. CUDA Code

The CUDA implementation builds upon the C++ code by parallelizing key operations to leverage GPU acceleration. This includes using CUDA kernels for element-wise operations such as the shrinkage function, and employing cuBLAS and cuSOLVER libraries for efficient linear algebra routines like vector norms and singular value decompositions (SVD). These optimizations significantly speed up the RPCA algorithm compared to the CPU version.

cuBLAS is NVIDIA's GPU-accelerated library for basic linear algebra subprograms (BLAS). It provides optimized routines for vector and matrix operations such as multiplication, addition, dot products, and norms. Offloading these operations to the GPU significantly accelerates computation-intensive linear algebra tasks fundamental to algorithms like RPCA.

cuSOLVER is a CUDA library designed for advanced linear algebra computations including matrix factorizations, eigenvalue decompositions, and singular value decomposition (SVD). It offers efficient GPU-accelerated solvers essential for RPCA's SVD step, enabling faster and more scalable matrix decompositions compared to CPU implementations.

Together, these libraries enable the CUDA implementation of RPCA to perform complex linear algebra operations efficiently on the GPU, resulting in faster processing and improved scalability.

The pseudocode for image resizing and iteration over different sizes is omitted, as it is identical to the C++ implementation.

Algorithm 3: CUDA-Accelerated Robust Principal Component Analysis (RPCA) Initialization

Input: Grayscale image file path img_path, list of target image sizes $\{s_i\}$, tolerance tol, max iterations max_iter
Output: Foreground and background matrices, performance metrics
foreach *size* $s \in \{s_i\}$ **do**
  Load grayscale image of size s;
  Initialize float array h_D of length $s \times s$;
  Allocate device arrays $d\_D, d\_S, d\_Y, d\_L, d\_tmp$ of size $s \times s$;
  Copy $h\_D \rightarrow d\_D$ (host to device);
  Initialize d_S, d_Y, d_L to zero on GPU;
  Compute Frobenius norm of d_D using cuBLAS: $\|D\|_F \rightarrow$ norm;
  Set $\mu \leftarrow \frac{s^2}{4 \times \text{norm}}$, $\mu^{-1} \leftarrow \frac{1}{\mu}$;
  Set $\lambda \leftarrow \frac{1}{\sqrt{s}}$;
  Initialize cuBLAS and cuSOLVER handles;
  Set err $\leftarrow \infty$, iter $\leftarrow 0$;
  while *err* $>$ *tol* ***and*** *iter* $<$ *max_iter* **do**
    Copy $d\_D \rightarrow d\_tmp$;
    Compute $d\_tmp \leftarrow d\_tmp - d\_S + \mu^{-1} d\_Y$ using cuBLAS saxpy operations;
    Copy d_tmp back to host h_D;
    Reshape h_D to $s \times s$ matrix M;
    Perform SVD on M using OpenCV: $M = U \Sigma V^T$;
    Threshold singular values: $\Sigma_{ii} \leftarrow \max(\Sigma_{ii} - \mu^{-1}, 0)$;
    Reconstruct $L_k = U \Sigma V^T$;
    Copy L_k to d_L on device;
    Copy $d\_D \rightarrow d\_tmp$;
    Compute $d\_tmp \leftarrow d\_tmp - d\_L + \mu^{-1} d\_Y$ using cuBLAS saxpy;
    Launch CUDA kernel `shrink_kernel` on d_tmp with threshold $\lambda \mu^{-1}$;
    Copy $d\_tmp \rightarrow d\_S$ (device to device);
    Copy $d\_D \rightarrow d\_tmp$;
    Compute $d\_tmp \leftarrow d\_tmp - d\_L - d\_S$ using cuBLAS saxpy operations;
    Compute $d\_Y \leftarrow d\_Y + \mu \times d\_tmp$ using cuBLAS saxpy;
    Copy d_tmp to host h_err;
    Compute sum of squares of h_err elements, then $err = \sqrt{\text{sum}}$;
    Increment iter $\leftarrow$ iter $+ 1$;
  end
  Copy $d\_S \rightarrow h\_S$, $d\_L \rightarrow h\_L$;
  Destroy cuBLAS and cuSOLVER handles;
  Free all allocated GPU memory;
  Return foreground h_S, background h_L, number of iterations, and final error;
end

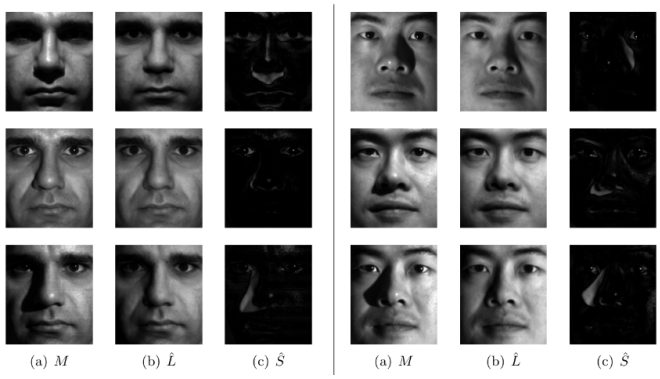
The resulting images and performance metrics will be

discussed in the following section.

III. Results

A. Baseline Paper

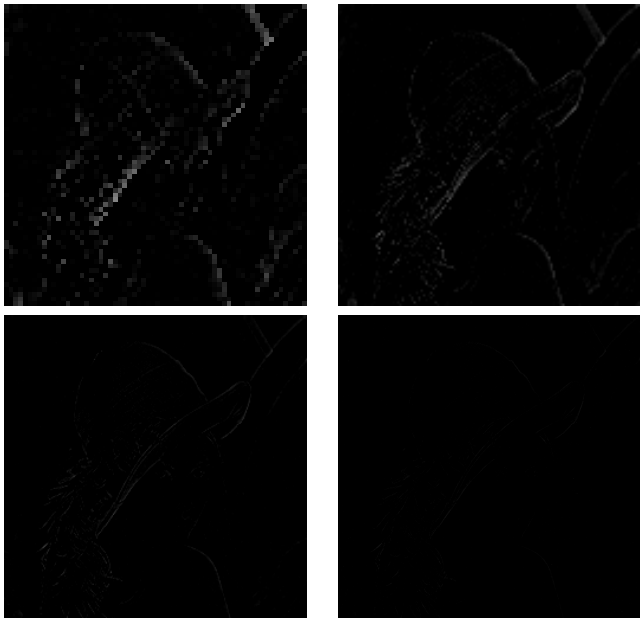
According to the original paper, the proposed RPCA algorithm has two key applications: video surveillance and face recognition. The authors state that this method can effectively handle challenges such as shadows and other corruptions in face images. Thet produced the following images to showcase it:



Şekil 1. Result images from [1]. The size of each image is 192 × 168 pixels.

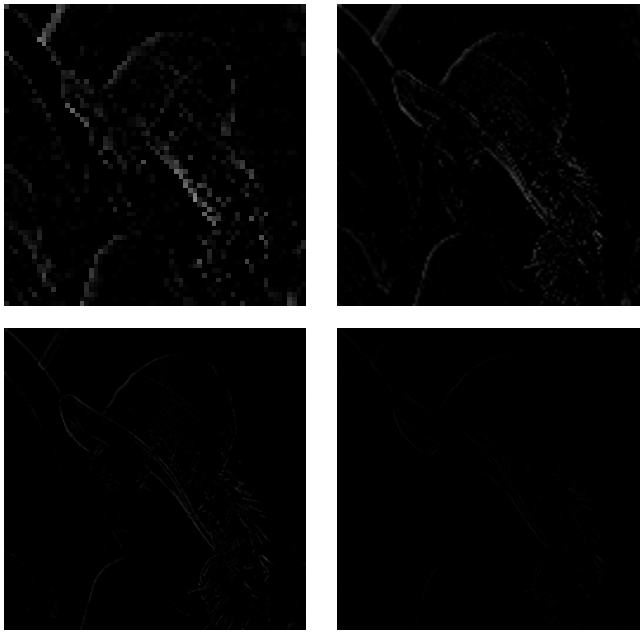
As part of this project, I generated similar face processing results using the well-known Lenna image [3]

B. C++ and CUDA Image Results



Şekil 2. Grayscale images produced using sparse matrix (foreground) approximations with images of size 64, 128, 256, and 512, respectively using C++ code applied to [3]

Şekil 3. Grayscale images produced using low rank matrix (background) approximations with images of size 64, 128, 256, and 512, respectively using C++ code applied to [3]



Şekil 4. Grayscale images produced using sparse matrix (foreground) approximations with images of size 64, 128, 256, and 512, respectively using CUDA code applied to [3]

From a visual standpoint, the only noticeable difference is that the output images produced by the CUDA implementation are rotated and translated compared to the original. This discrepancy arises from how image data is stored and accessed: OpenCV stores images in row-major order (i.e., rows first, then columns), and the CPU implementation using Eigen preserves this order by iterating over rows and then

Şekil 5. Grayscale images produced using low rank matrix (background) approximations with images of size 64, 128, 256, and 512, respectively using CUDA code applied to [3]

columns. The images shown here are rotated 90 degrees clockwise relative to the original images produced by the CUDA code.

C. Code Performance Results of Implemented Algorithms

Tablo I
TOTAL PROCESSING TIMES FOR DIFFERENT IMAGE SIZES COMPARING CPU AND CUDA IMPLEMENTATIONS.

Size	CPU Total Time (ms)	CUDA Total Time (ms)
64x64	6,504	454
128x128	44,559	183
256x256	311,332	771
512x512	11,442,851	3,923

Table I presents a comparison of the total processing times between the CUDA and C++ implementations. The results clearly demonstrate that the CUDA implementation significantly outperforms the CPU-based version in terms of execution time.

Tablo II
CPU TIMING BREAKDOWN PER IMAGE SIZES

Size	Load	Convert	RPCA	Total	Error
64×64	5	0	6,502	6,504	0.311
128×128	5	0	44,559	44,563	0.452
256×256	5	3	311,316	311,332	0.326
512×512	5	16	9,907,216	11,442,851	0.269

The results clearly show that the C++ RPCA implementation exhibits super-linear scaling in execution time with respect to image size. While the data loading and format conversion times remain negligible, the RPCA step dominates total runtime and grows disproportionately. For instance, increasing the image size from 256×256256×256 to 512×512512×512 results in a more than 30× increase in total time, suggesting the algorithm's cubic complexity due to repeated Singular Value Decompositions (SVD). This confirms that the Eigen-based CPU implementation becomes computationally infeasible for high-resolution images without optimization or hardware acceleration.

Tablo III
CUDA TIMING BREAKDOWN PER IMAGE SIZES

Size	Load (ms)	ToDev (ms)	RPCA (ms)	Total (ms)	Final Err
64×64	5	0	58	454	0.311221
128×128	3	0	178	183	0.452154
256×256	3	0	765	771	0.327283
512×512	4	2	3910	3923	0.269419

The CUDA implementation of RPCA is broken down into several key stages. First, the **Load** step reads and resizes the grayscale image using OpenCV to match the target dimensions. Next, in the **To device** phase, the image data is flattened in column-major order and transferred to GPU memory. The main computation happens during the **RPCA** step, where the algorithm iteratively performs low-rank and sparse matrix decomposition using cuBLAS for vector operations, shrinkage via a custom CUDA kernel, and SVD thresholding on the host. This process continues for up to 100 iterations or until convergence. Once decomposition is complete, the **Save** step converts the resulting foreground and background matrices back into images and writes them to disk. Finally, the **Total** time encompasses all steps from image loading to result saving, representing the complete runtime of the CUDA-based RPCA pipeline.

IV. CONCLUSION

The CUDA implementation significantly outperforms the CPU version across all image sizes, delivering much faster total processing times. While the CPU runtime grows super-linearly—primarily due to the expensive Singular Value Decomposition steps—the CUDA version maintains efficient performance by leveraging GPU acceleration, making it far more suitable for high-resolution images and large-scale RPCA computations.

KAYNAKLAR

[1] E. J. Candès, X. Li, Y. Ma, and J. Wright, "Robust principal component analysis?" arXiv preprint arXiv:0912.3599, 2009, [Online; accessed 24-May-2025]. [Online]. Available: https://arxiv.org/abs/0912.3599

[2] D. Ganguli, "robust-pca: Python implementation of robust principal component analysis," https://github.com/dganguli/robust-pca/blob/master/r$_pca.py$, 2015, [$Online; accessed 24-May-2025$].

[3] "Lenna test image," originally published in Playboy magazine, November 1972. Widely used in image processing research.