HPC

FACE RECOGNITION

Using PCA, LDA, ICA, Fisherfaces

Rohan Vinkare

S20220010244

INTRODUCTION

- Face recognition is a widely adopted biometric technique used in surveillance, security, and identity verification systems.
- This project evaluates and compares the performance of four classical facial recognition algorithms: Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), Independent Component Analysis (ICA), and Fisherfaces.
- The focus is on preparing these algorithms for High Performance Computing (HPC) optimization using parallel computing platforms such as CUDA and OpenMP.
- As a baseline, sequential (serial) implementations of all four algorithms have been successfully completed and evaluated.

OBJECTIVE

- Implement and analyze four classical face recognition algorithms: PCA, LDA, ICA, and Fisherfaces, using a common dataset for consistent evaluation.
- Evaluate their performance in sequential form to establish a baseline for comparison with parallel implementations.
- The project aims to optimize computational performance using parallel processing techniques tailored to each algorithm.
- Key performance metrics such as execution time, speedup, and scalability will be measured to assess the effectiveness of each parallel approach.

DATASET AND TOOLS

- Dataset: AT&T Face Dataset on Kaggle
- 40 subjects, 10 grayscale images each (size: 112x92).

Tools :

- C++ for algorithm logic
- openCV
- CUDA/OpenMP (planned)

Environment :

- Multi-core CPU (for OpenMP)
- CUDA-compatible GPU (for parallel acceleration)

WORK DONE SO FAR

- PCA (Principal Component Analysis):

  Successfully implemented for face recognition.

  Used to reduce dimensionality and extract key features (eigenfaces).

- LDA (Linear Discriminant Analysis):

  Implemented in sequential form to enhance class separability.

- ICA (Independent Component Analysis):

  For feature extraction based on statistical independence.

- Fisherfaces (PCA + LDA):

  Combined approach to improve recognition accuracy over PCA alone.

METHODOLOGY OVERVIEW

Algorithm	Parallel Platform (Planned)	Reason
PCA	CUDA	Matrix operations & eigen decomposition benefit from GPU
Fisherfaces	CUDA	Combines PCA & LDA → GPU ideal for high-dimensional ops
LDA	OpenMP	Loop-level parallelism in scatter matrix computation
ICA	OpenMP	Independent component stages map well to shared memory threads

WHY THESE PLATFORMS?

- **CUDA (for PCA & Fisherfaces)**:
- High throughput for matrix ops
- Significant speedup on GPUs
- **OpenMP (for LDA & ICA)**:
- Easy integration with C++
- Best for loop-heavy or memory-shared tasks

WHY NOT OTHERS?

- **MPI:** More suitable for distributed clusters
- **Pthreads:** Lower-level, harder to manage than OpenMP

PCA – SUMMARY

- Purpose: Reduce dimensionality via eigenfaces.
- **Steps:**

  Normalize faces

  Compute covariance matrix

  Perform eigen decomposition
- Current: Implemented in C++ (sequential)
- Planned: CUDA for speedup via parallel projection & eigenspace computation

LDA – SUMMARY

- Purpose: Maximize class separability.
- Requires: Supervised learning (label information).
- **Steps:**

  Compute between-class & within-class scatter matrices

  Solve generalized eigenproblem
- Planned: OpenMP for parallel scatter matrix computation

ICA – SUMMARY

- Purpose: Extract independent statistical features.
- **Steps:**

  Whitening

  Iterative estimation of independent components
- **Planned: OpenMP**

  Each component estimation as parallel loop

  Shared memory efficiency

FISHERFACES – SUMMARY

- urpose: Combine PCA + LDA for optimal class separation.
- **Steps:**

  PCA for dimensionality reduction

  LDA for maximizing discriminability
- **Planned:** CUDA (both PCA and LDA stages are compute-heavy)

SAMPLE RESULTS :– OPENMP

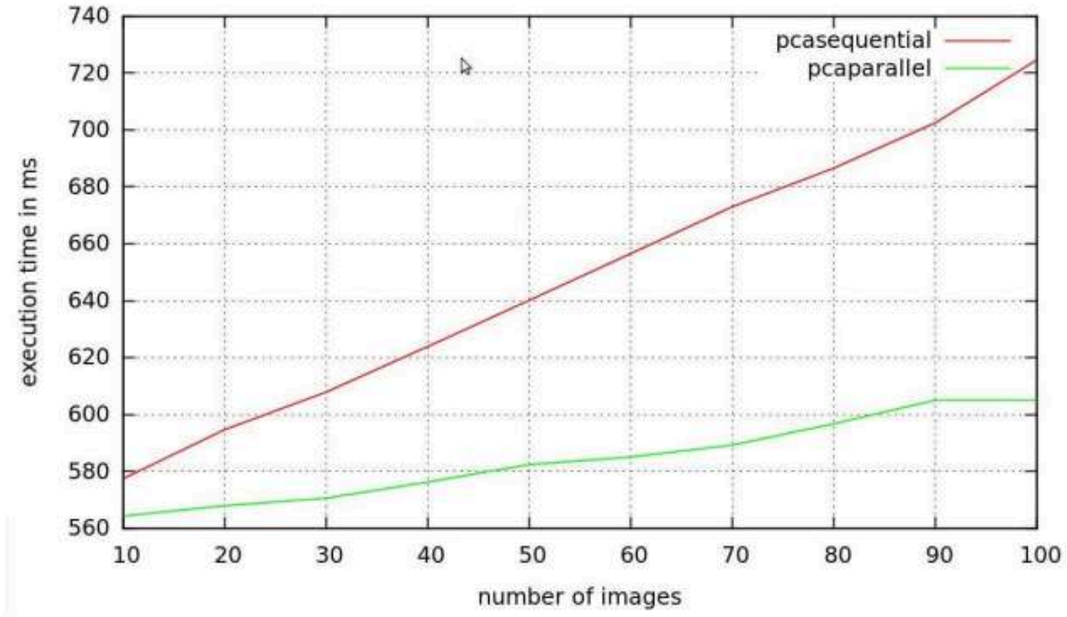


Fig. 2. Performance comparison of sequential and parallel PCA Algorithm.

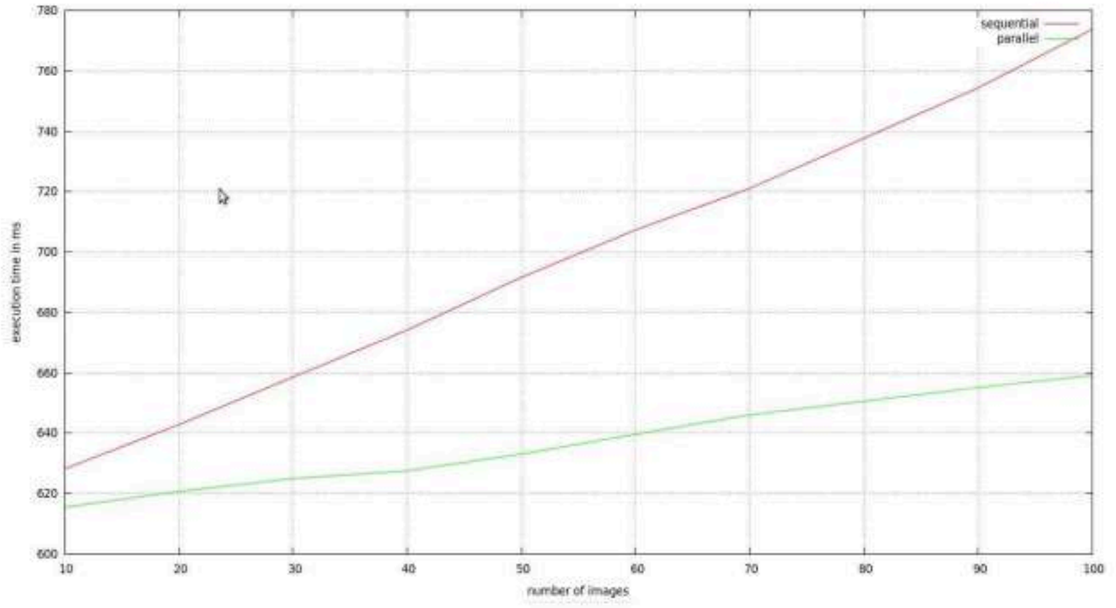


Fig. 3. Performance comparison of sequential and parallel ICA.

SAMPLE RESULTS :– OPENMP

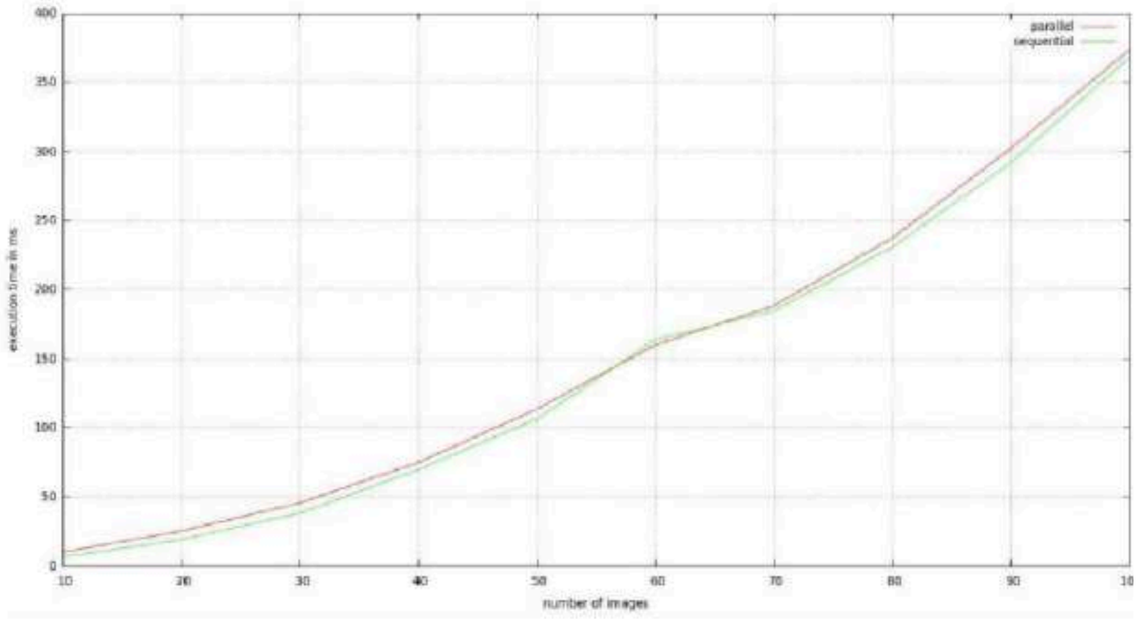


Fig.4. Performance Comparison of LDA sequential and parallel algorithms.

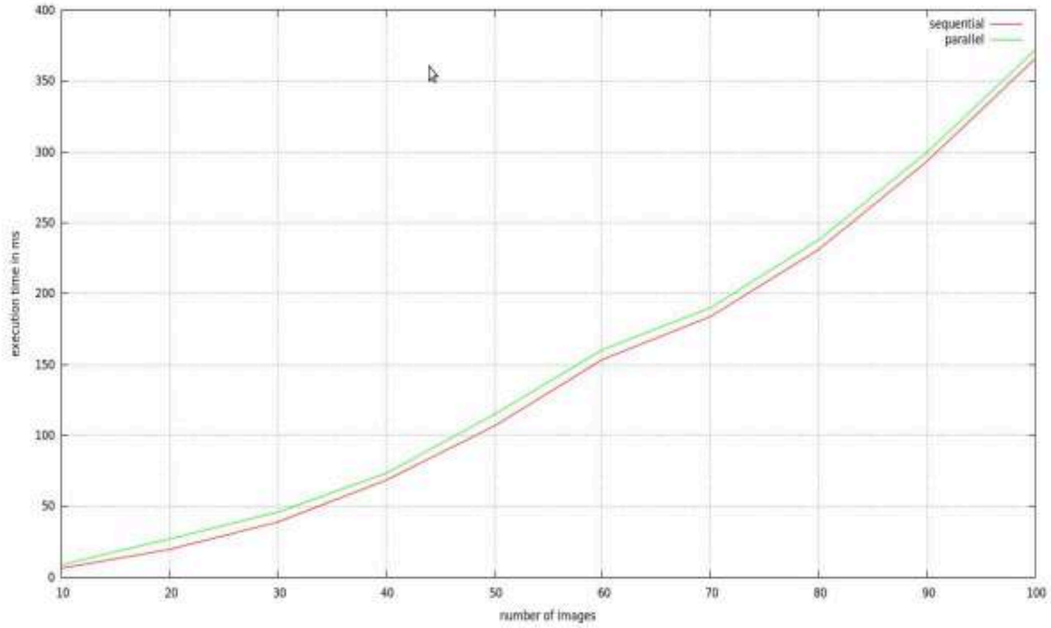
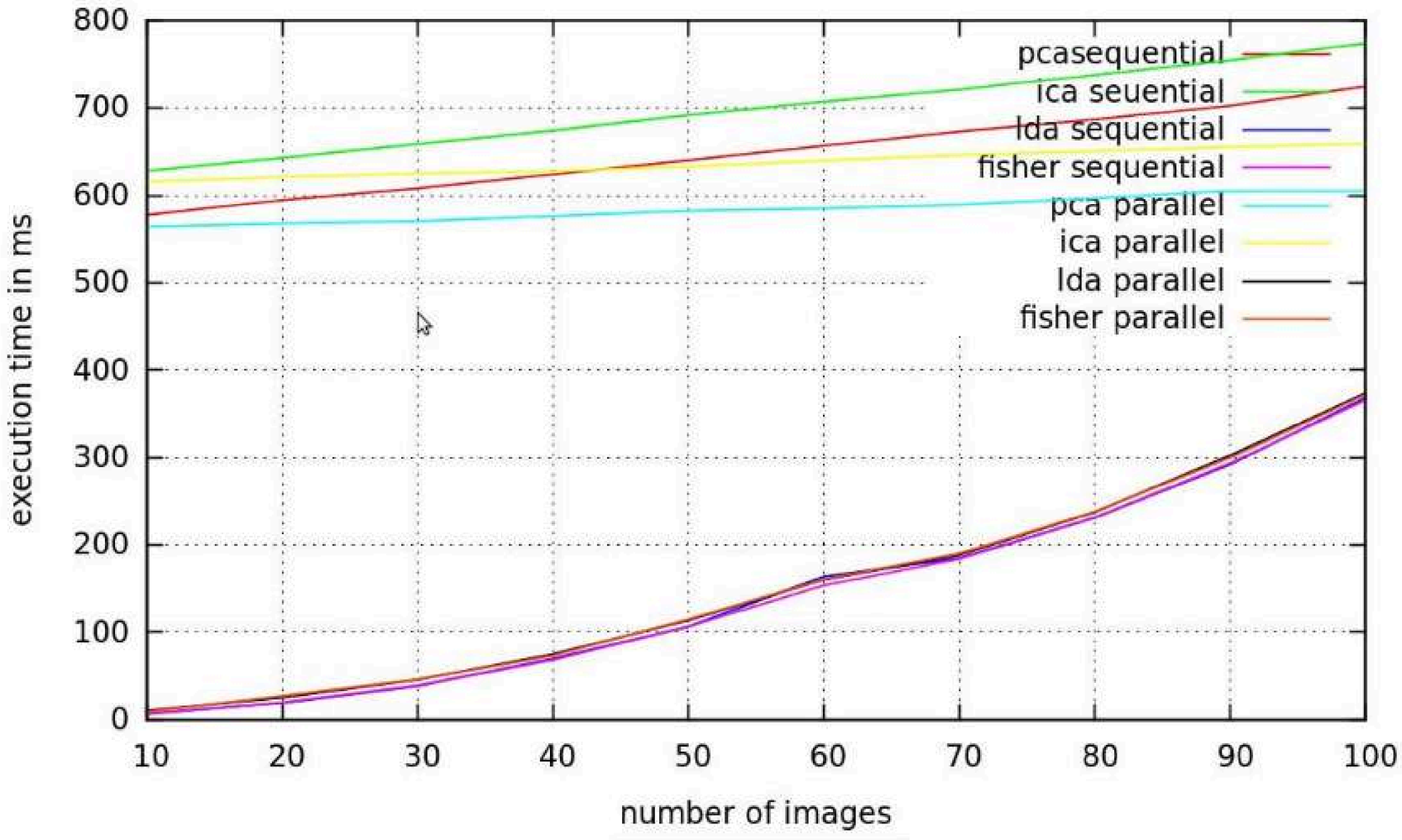


Fig. 5. Performance Comparison of sequential and parallel Fisher Face.



SAMPLE RESULTS OPENMP

FUTURE WORK

- **PCA** will be parallelized using CUDA to leverage GPU acceleration and improve performance for large-scale datasets.
- **LDA** will be parallelized using OpenMP, enabling multi-threaded execution on multi-core CPUs for improved efficiency.
- **Benchmark Performance**
- Measure execution time, speedup, and scalability for each parallel implementation.
- Compare against sequential baselines to evaluate improvement.

Evaluation - 02

HPC

FACE RECOGNITION

Using PCA, LDA, Fisherfaces

Guide :– Dr. Bheemappa Halavar

Sujal Awargand
S20220010213

Rohan Vinkare
S20220010244

Jathin Reddy
S20220010241

PROJECT OVERVIEW

PROBLEM STATEMENT

- Facial recognition systems require significant computational resources due to high-dimensional data processing
- Traditional sequential implementations of PCA and LDA algorithms are computationally intensive
- Need for faster processing for real-time facial recognition applications
- Challenge of optimizing performance while maintaining accuracy

OBJECTIVES

1. Implement and optimize PCA and LDA algorithms for facial recognition
2. Develop parallel implementations using:
   - OpenMP (CPU parallel processing)
   - CUDA (GPU acceleration)
3. Compare performance metrics between:
   - Sequential implementation
   - OpenMP parallel implementation
   - CUDA parallel implementation
4. Achieve significant speedup while maintaining recognition accuracy

DATASET AND TOOLS

- Dataset: AT&T Face Dataset on Kaggle
- 40 subjects, 10 grayscale images each (size: 112x92).

Tools :

- C++ for algorithm logic
- openCV
- CUDA/OpenMP (planned)

Environment :

- Multi-core CPU (for OpenMP)
- CUDA-compatible GPU (for parallel acceleration)

PRINCIPAL COMPONENT ANALYSIS (PCA)

- – Dimensionality reduction technique
- – Transforms high-dimensional data into lower-dimensional space
- – Preserves maximum variance in the data

Eigenfaces Approach

– Represents faces as linear combinations of base faces

– Reduces facial images to principal components

– Maintains essential features while reducing data size

Working Process

– Compute mean face from training images

– Calculate covariance matrix

– Extract eigenvectors (eigenfaces)

– Project faces onto eigenface space

LINEAR DISCRIMINANT ANALYSIS (LDA)

- Supervised learning algorithm ,Focus on maximizing class separability ,Finds optimal projection for classification

Fisherfaces Approach

- Maximizes between-class scatter
- Minimizes within-class scatter
- Better handling of lighting and expression variations

Working Process

- Calculate within-class and between-class scatter matrices
- Compute eigenvectors of scatter matrices
- Project data onto discriminant space
- Optimize class separation

APPLICATION IN FACIAL RECOGNITION – PCA – 1

- Converts high-dimensional data into a lower-dimensional space while retaining maximum variance.

1. **Data Preprocessing**
   - Load PGM images and normalize pixel values.
   - Compute the mean face vector.
   - Center the data by subtracting the mean face vector.

2. Covariance Matrix Computation:

- Calculate the covariance matrix using the centered data.

3. Eigenvector and Eigenvalue Computation

- Use power iteration to compute eigenvalues and eigenvectors.
- Select top 'k' eigenvectors as eigenfaces.

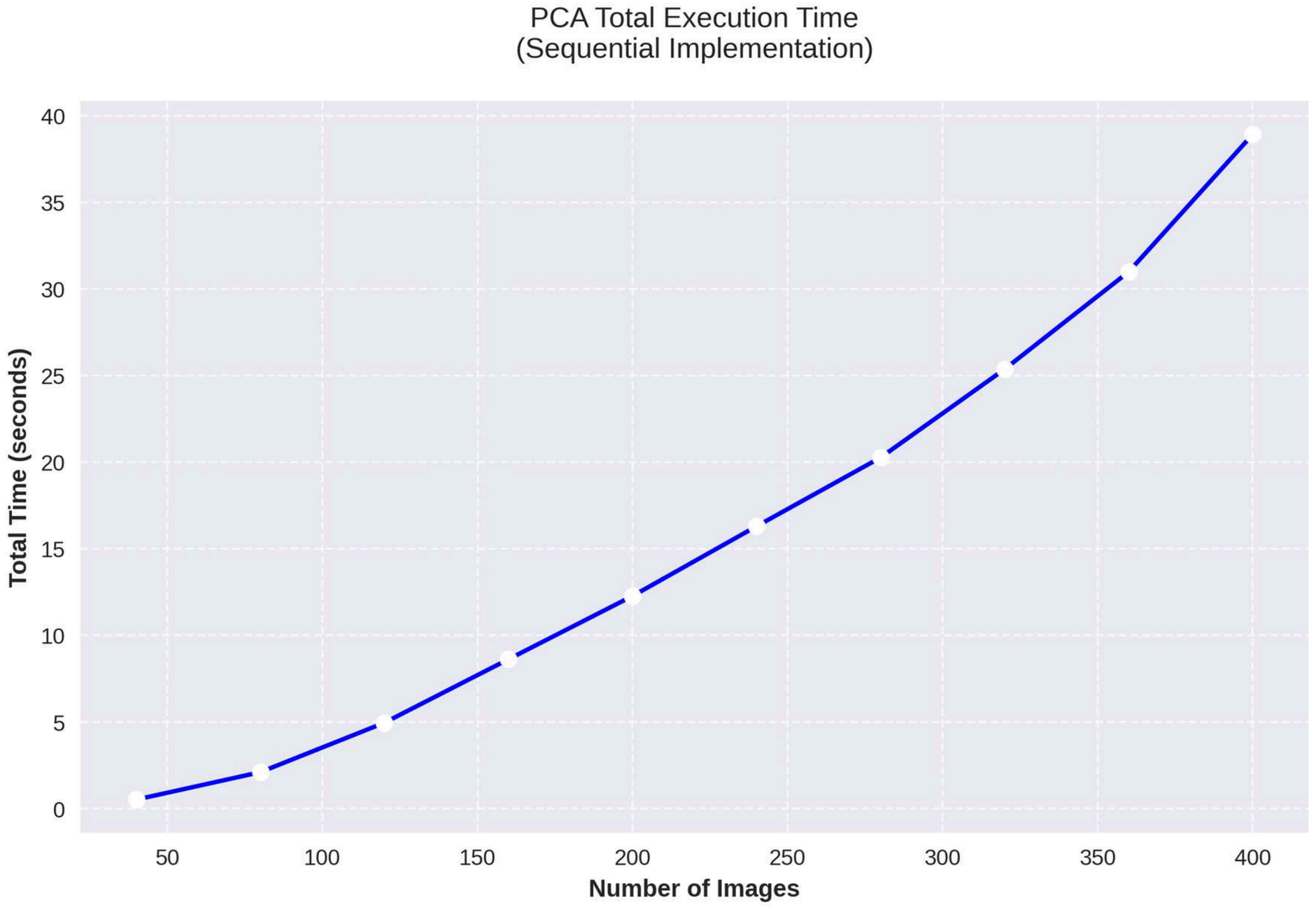
APPLICATION IN FACIAL RECOGNITION – PCA – 2

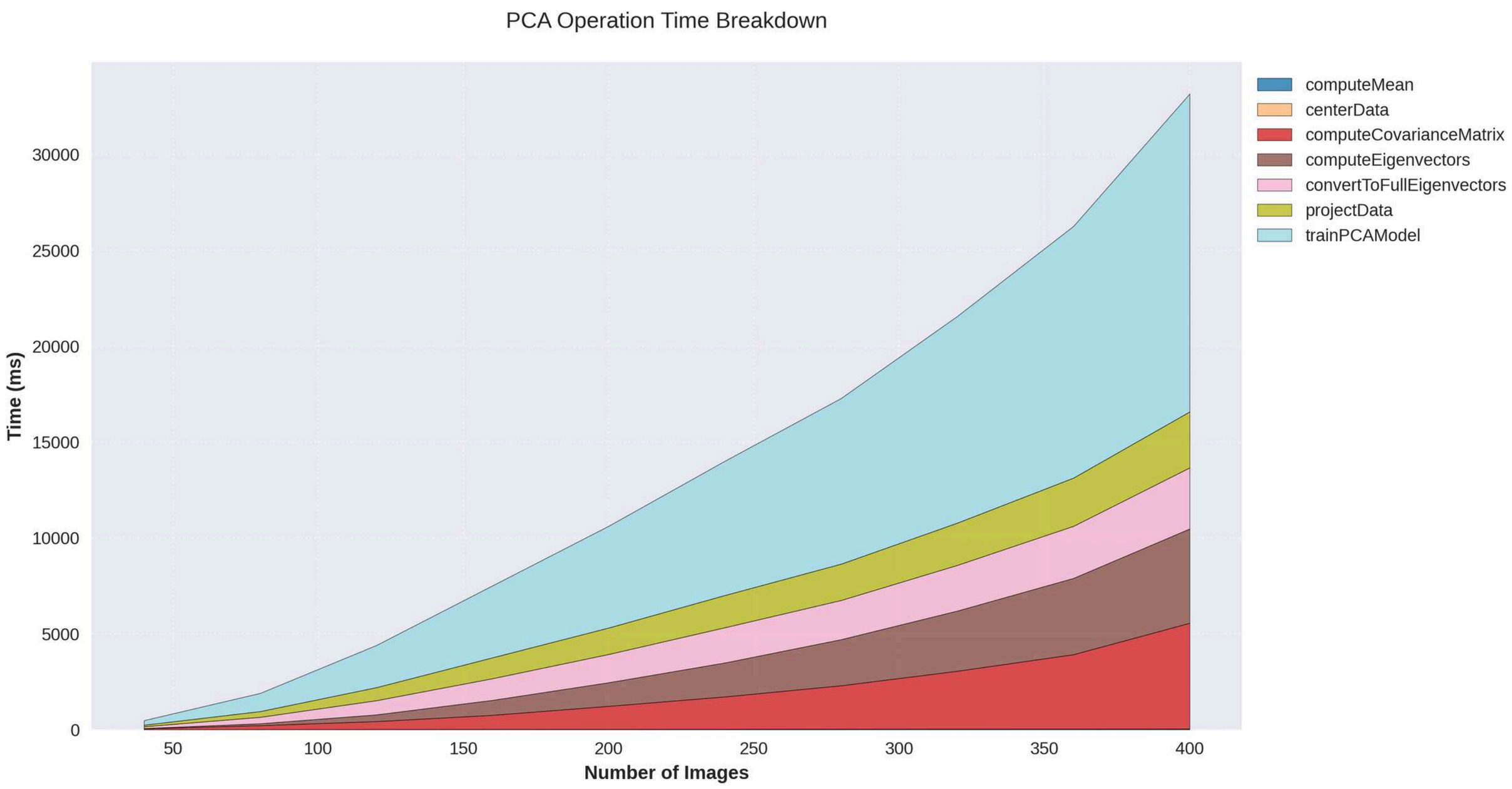
4. Projection and Recognition

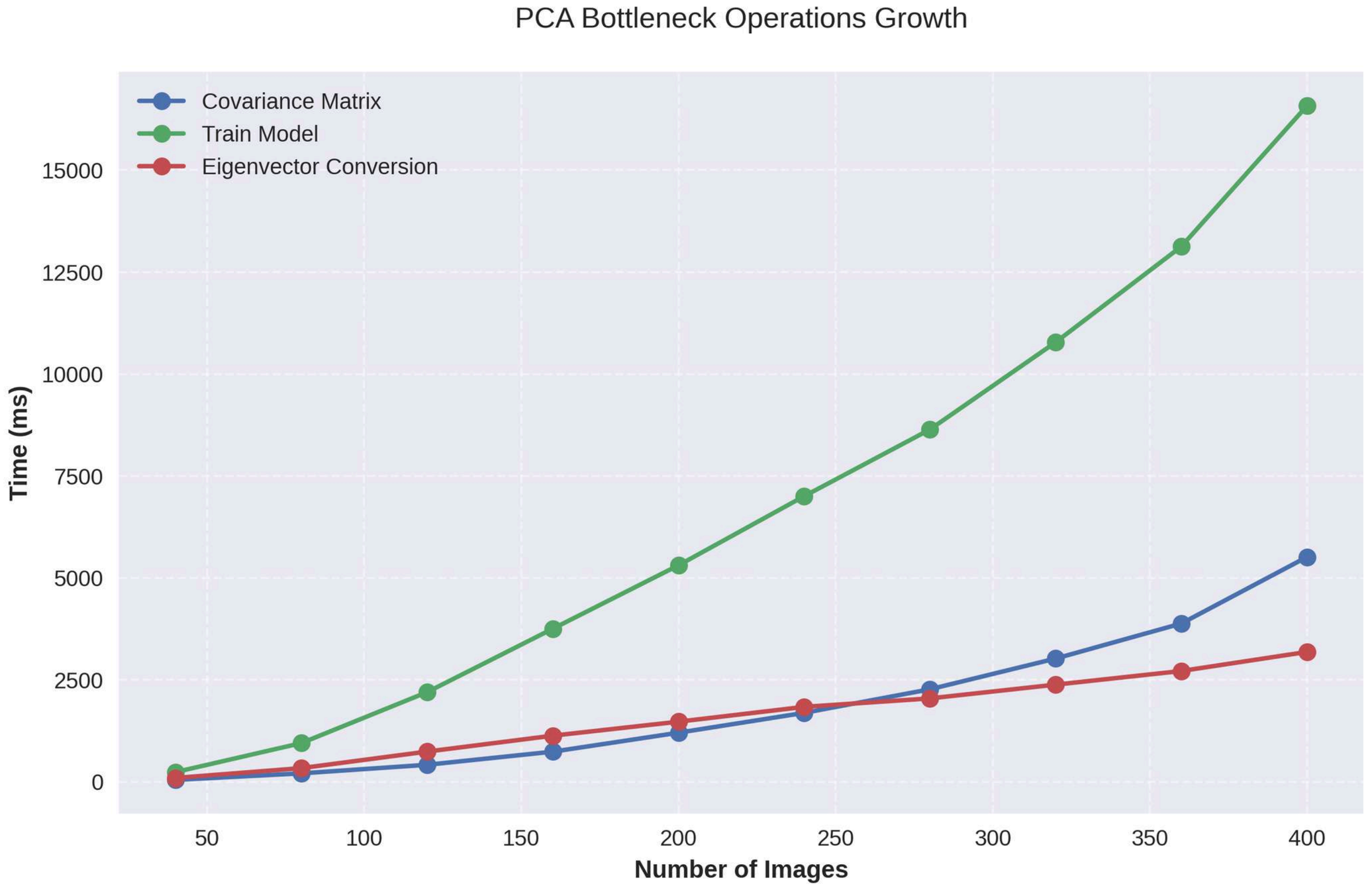
- Project training and test data onto the eigenfaces.
- Use Euclidean distance to find the nearest match in the feature space.

5. Output

- Save the mean face and eigenfaces as PGM files.
- Display recognition accuracy.


PCA Total Execution Time
(Sequential Implementation)
Total Time (seconds)
0
5
10
15
20
25
30
35
40
50
100
150
200
250
300
350
400
Number of Images


PCA Operation Time Breakdown
Time (ms)
0
5000
10000
15000
20000
25000
30000
50
100
150
200
250
300
350
400
Number of Images
computeMean
centerData
computeCovarianceMatrix
computeEigenvectors
convertToFullEigenvectors
projectData
trainPCAModel


PCA Bottleneck Operations Growth
Covariance Matrix
Train Model
Eigenvector Conversion
Time (ms)
0
2500
5000
7500
10000
12500
15000
50
100
150
200
250
300
350
400
Number of Images

AREAS FOR PARALLELIZATION – PCA 1

1. Normalization of Pixel Values

- Normalize images by dividing each pixel by the maximum value.
- Parallelization: Distribute the computation of pixel normalization across threads (OpenMP) or CUDA blocks.

2. Mean Vector Computation

- Compute the mean of all training images pixel-wise.
- Parallelization: Sum the pixel values across images in parallel.

3. Centering the Data

- Subtract the mean vector from each image.
- Parallelization: Process each image independently.

AREAS FOR PARALLELIZATION – PCA 2

4. Covariance Matrix Computation

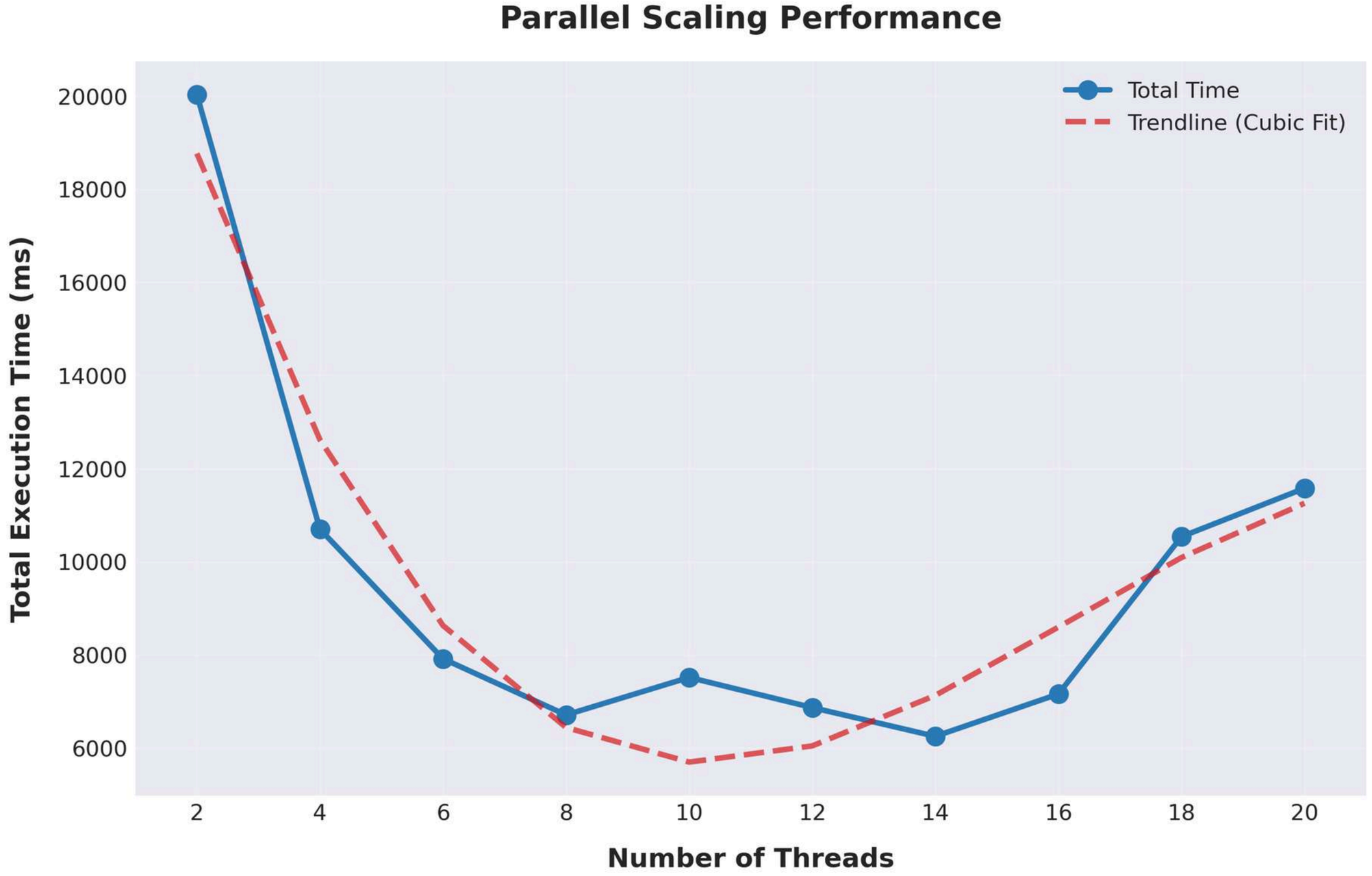
- Calculate the covariance matrix using matrix multiplication.
- Parallelization: Perform matrix multiplications in parallel using OpenMP or CUDA.

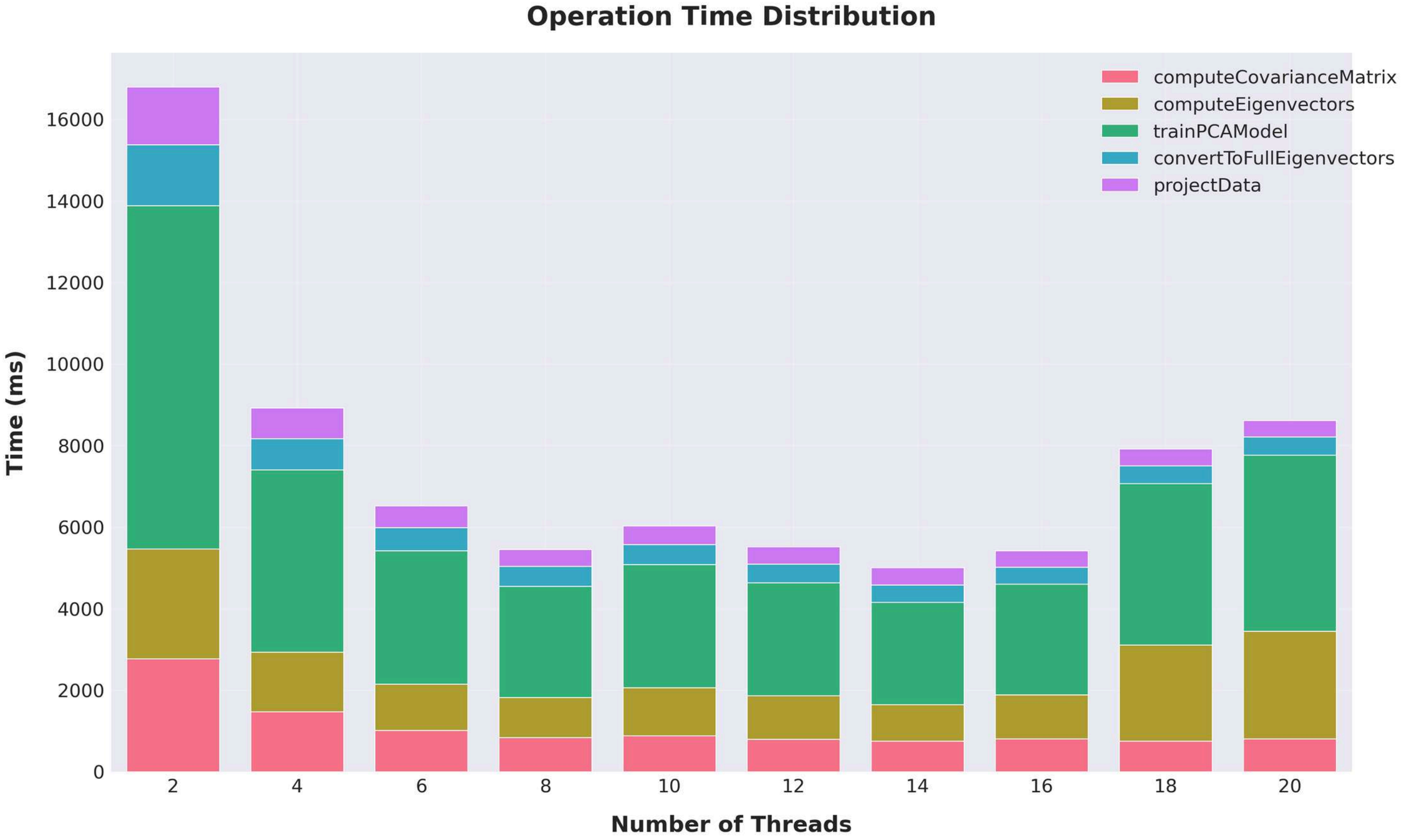
5. Eigenvalue and Eigenvector Computation

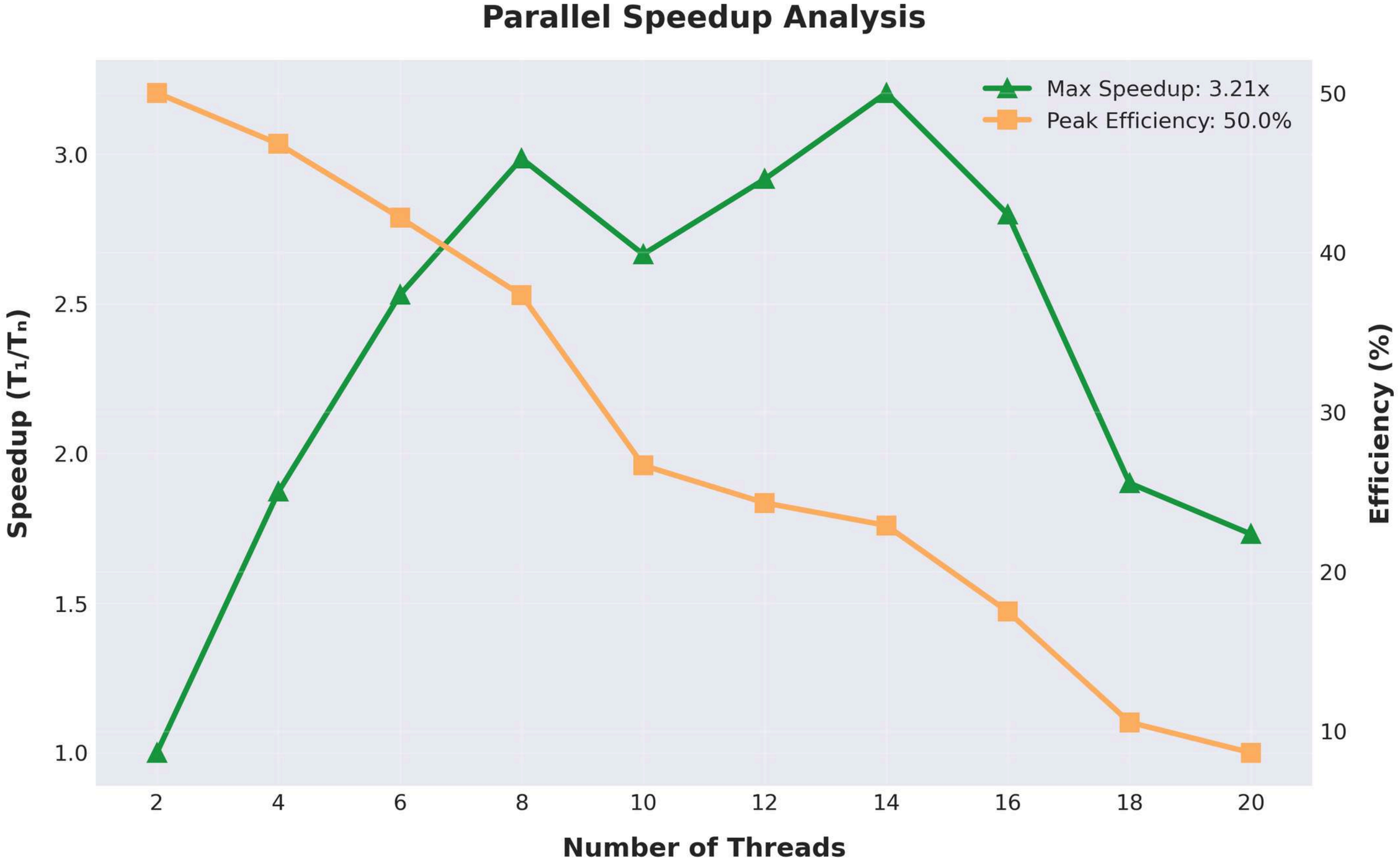
- Use power iteration to compute eigenvalues and eigenvectors.
- Parallelization: Parallelize the matrix-vector multiplications within each iteration.

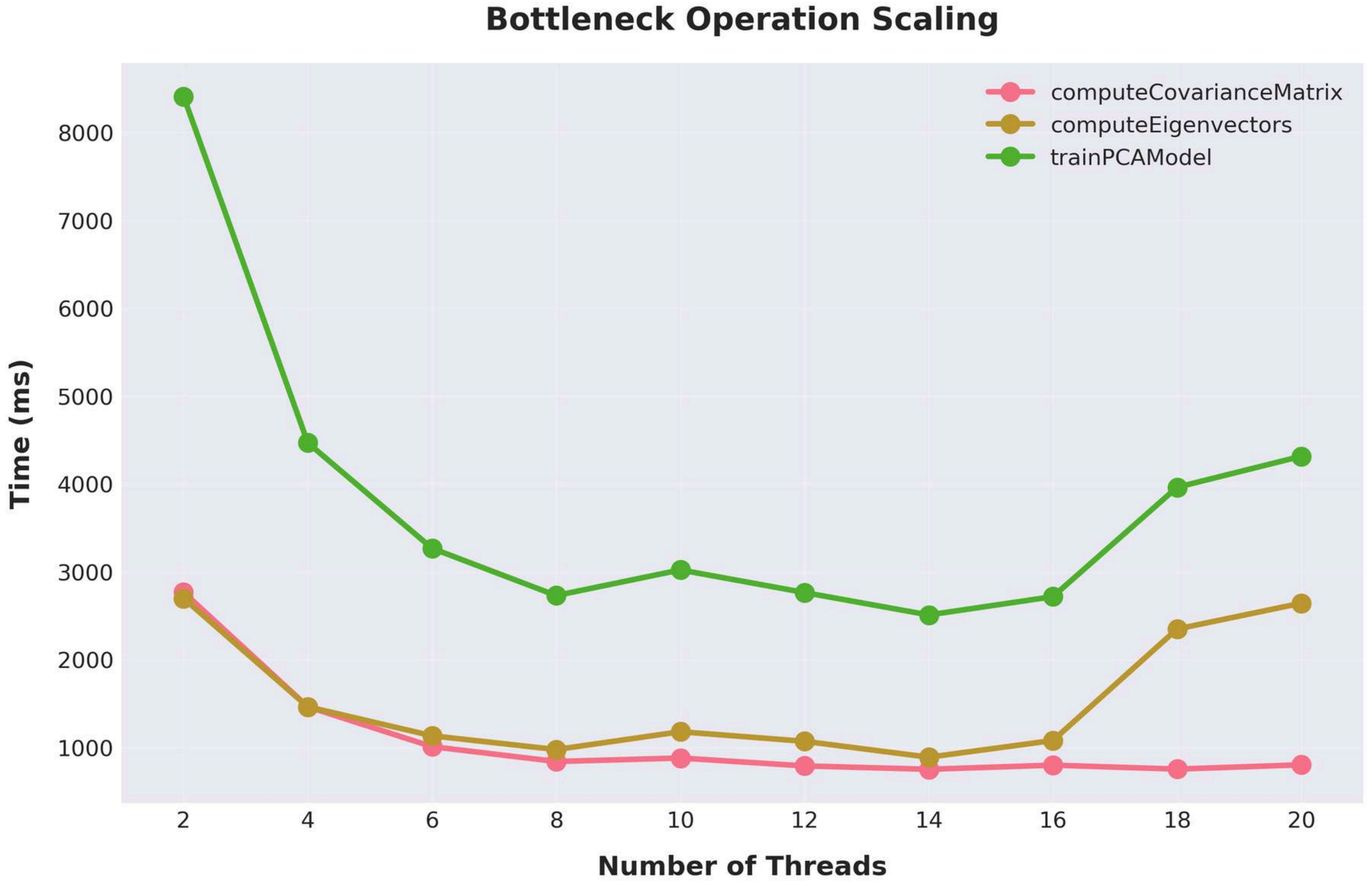
6. Projection onto Eigenfaces

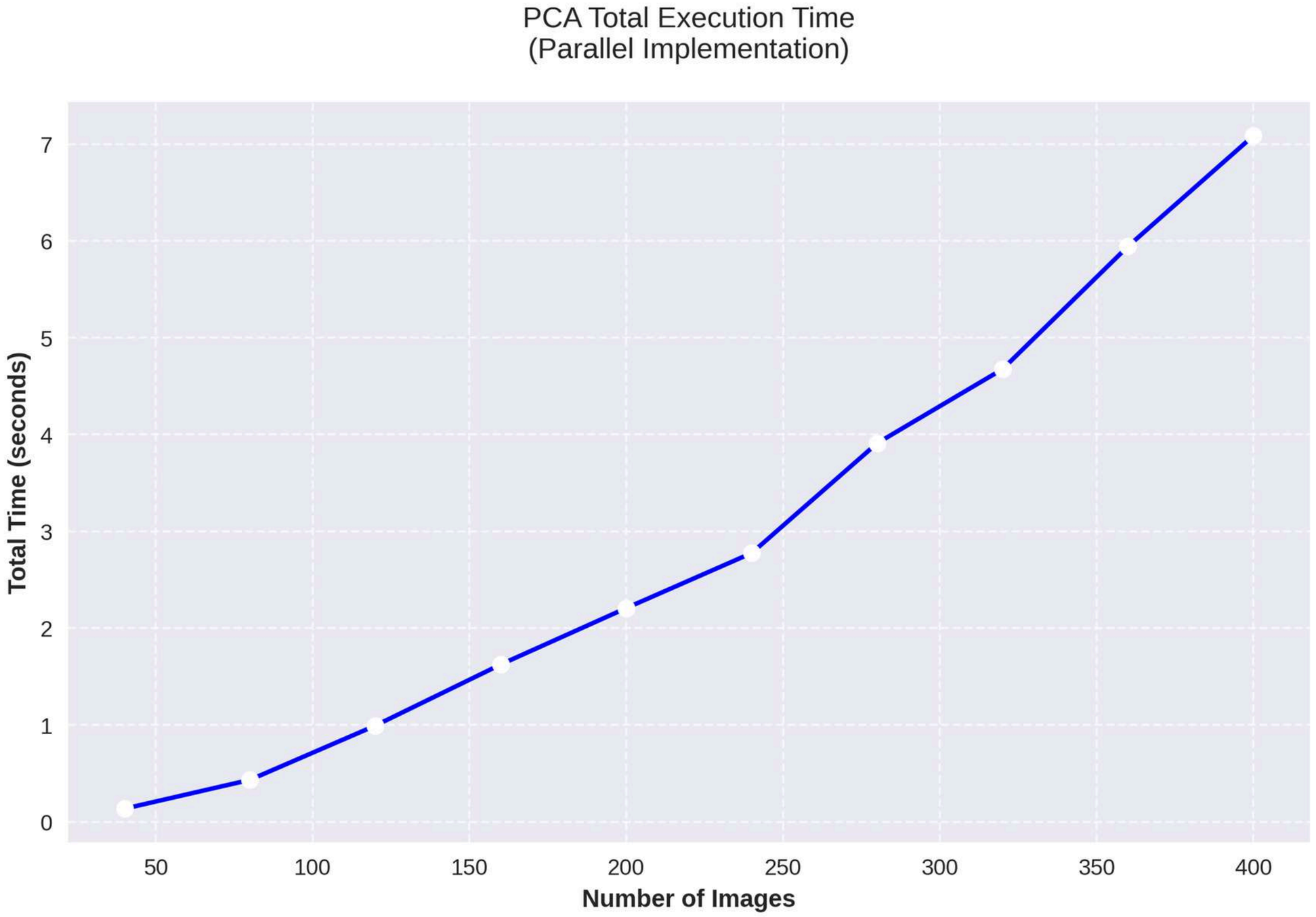
- Project the training and test data onto the top 'k' eigenvectors.
- Parallelization: Process each image independently during projection.

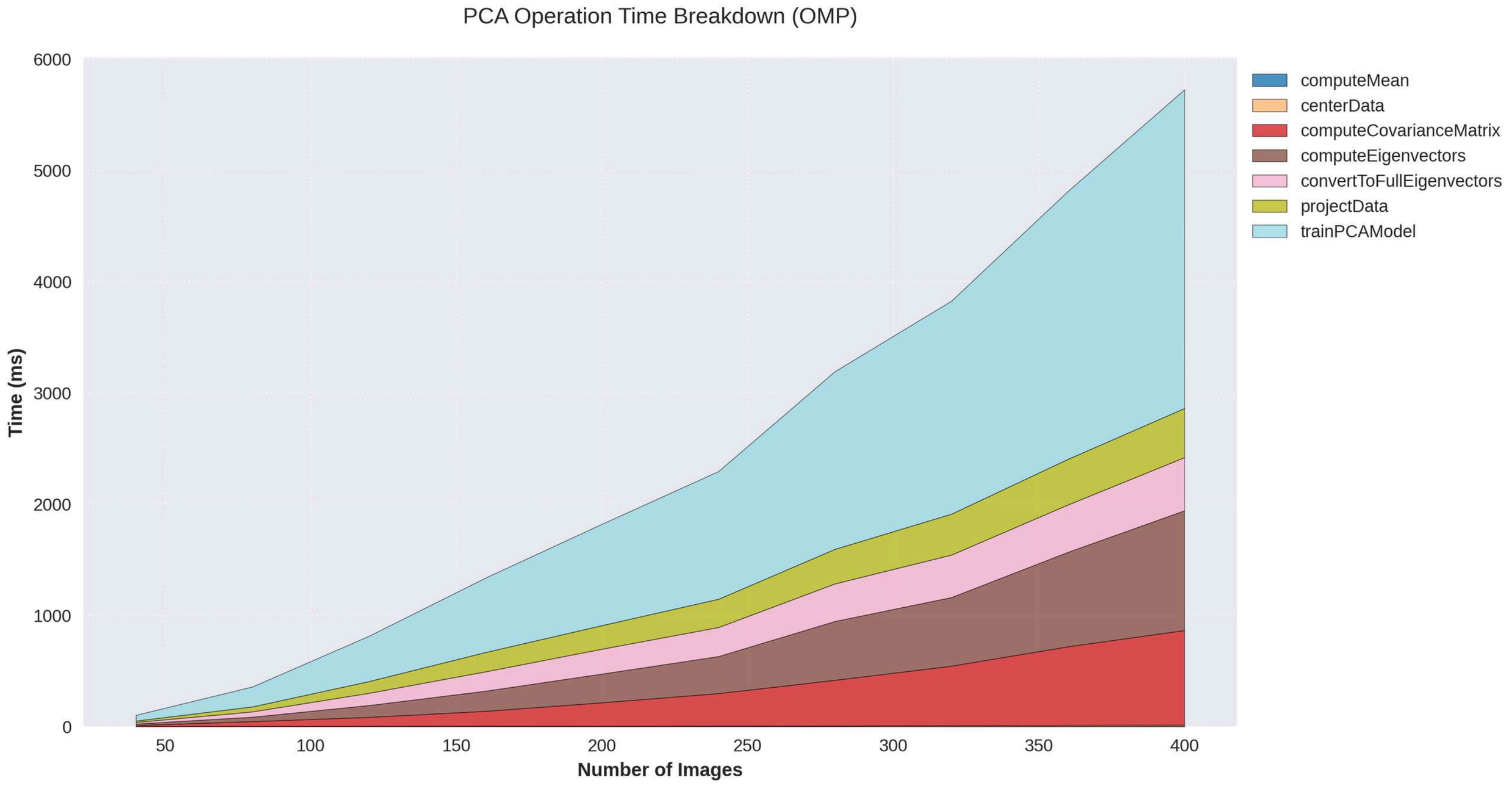

Parallel Scaling Performance
Total Time
Trendline (Cubic Fit)
Total Execution Time (ms)
20000
18000
16000
14000
12000
10000
8000
6000
2
4
6
8
10
12
14
16
18
20
Number of Threads


Operation Time Distribution
computeCovarianceMatrix
computeEigenvectors
trainPCAModel
convertToFullEigenvectors
projectData
Time (ms)
0
2000
4000
6000
8000
10000
12000
14000
16000
2
4
6
8
10
12
14
16
18
20
Number of Threads

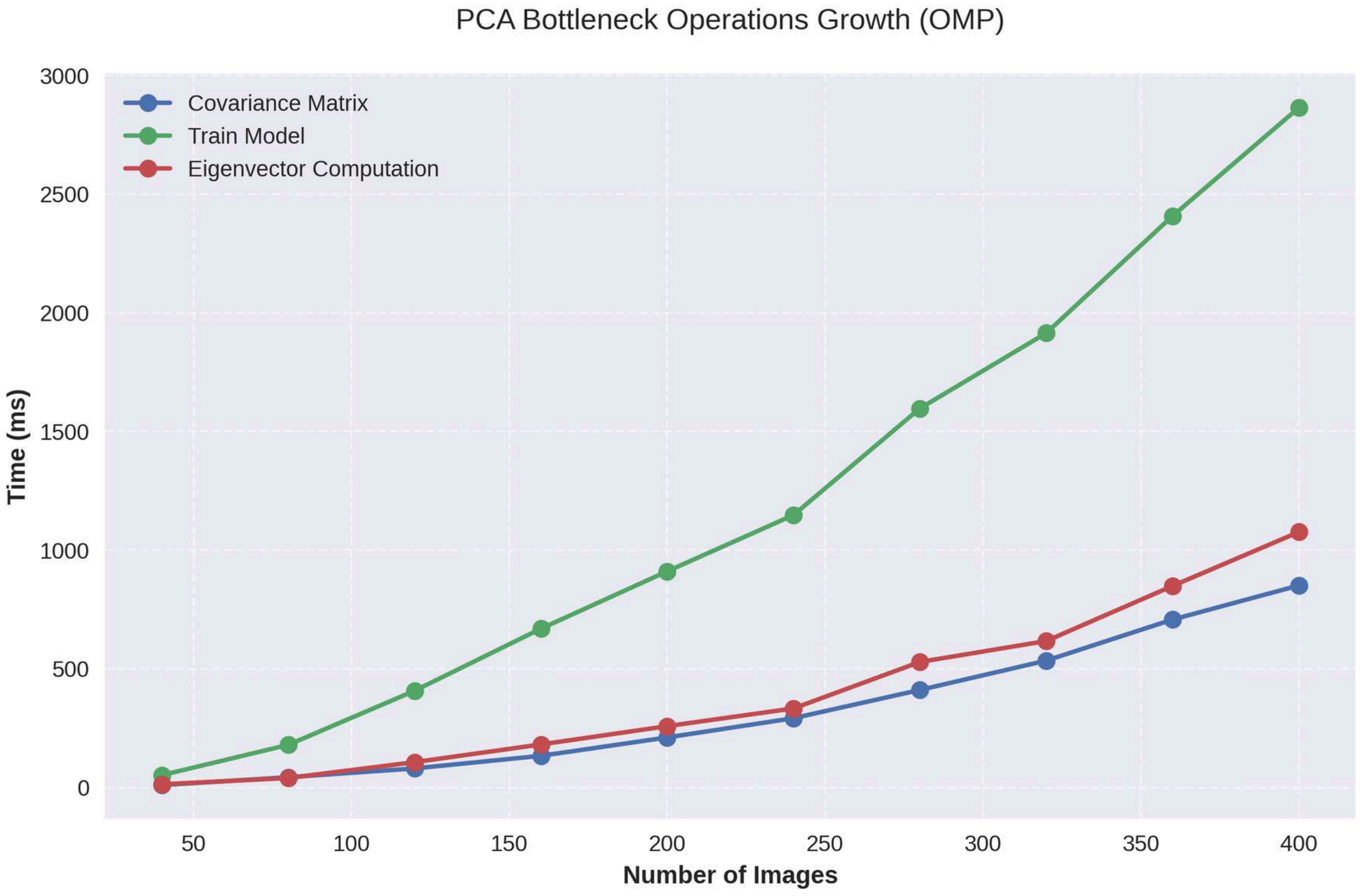

Parallel Speedup Analysis
Max Speedup: 3.21x
Peak Efficiency: 50.0%
Speedup (T₁/Tₙ)
Efficiency (%)
Number of Threads
1.0
1.5
2.0
2.5
3.0
10
20
30
40
50
2
4
6
8
10
12
14
16
18
20

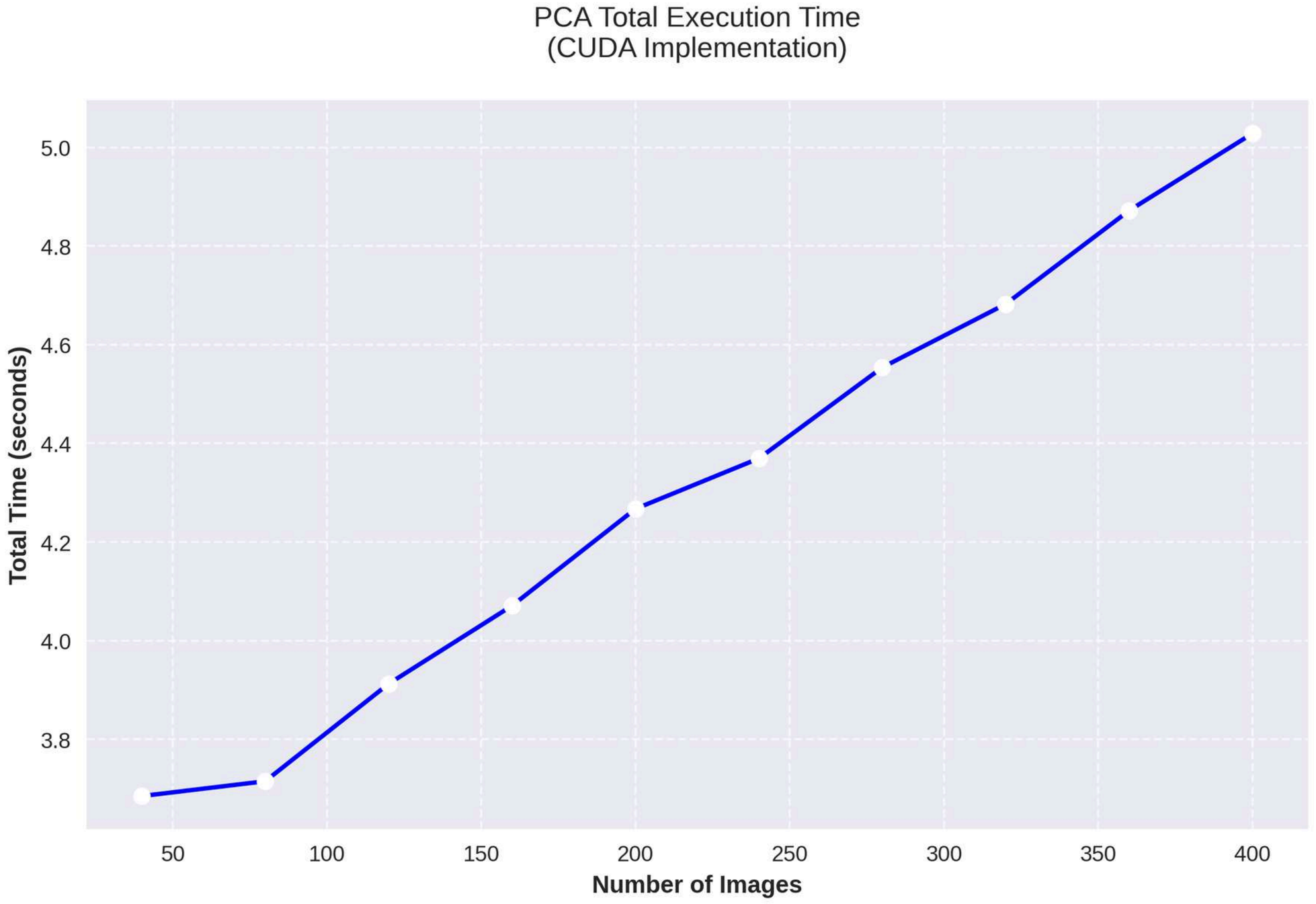

Bottleneck Operation Scaling
computeCovarianceMatrix
computeEigenvectors
trainPCAModel
Time (ms)
8000
7000
6000
5000
4000
3000
2000
1000
2
4
6
8
10
12
14
16
18
20
Number of Threads

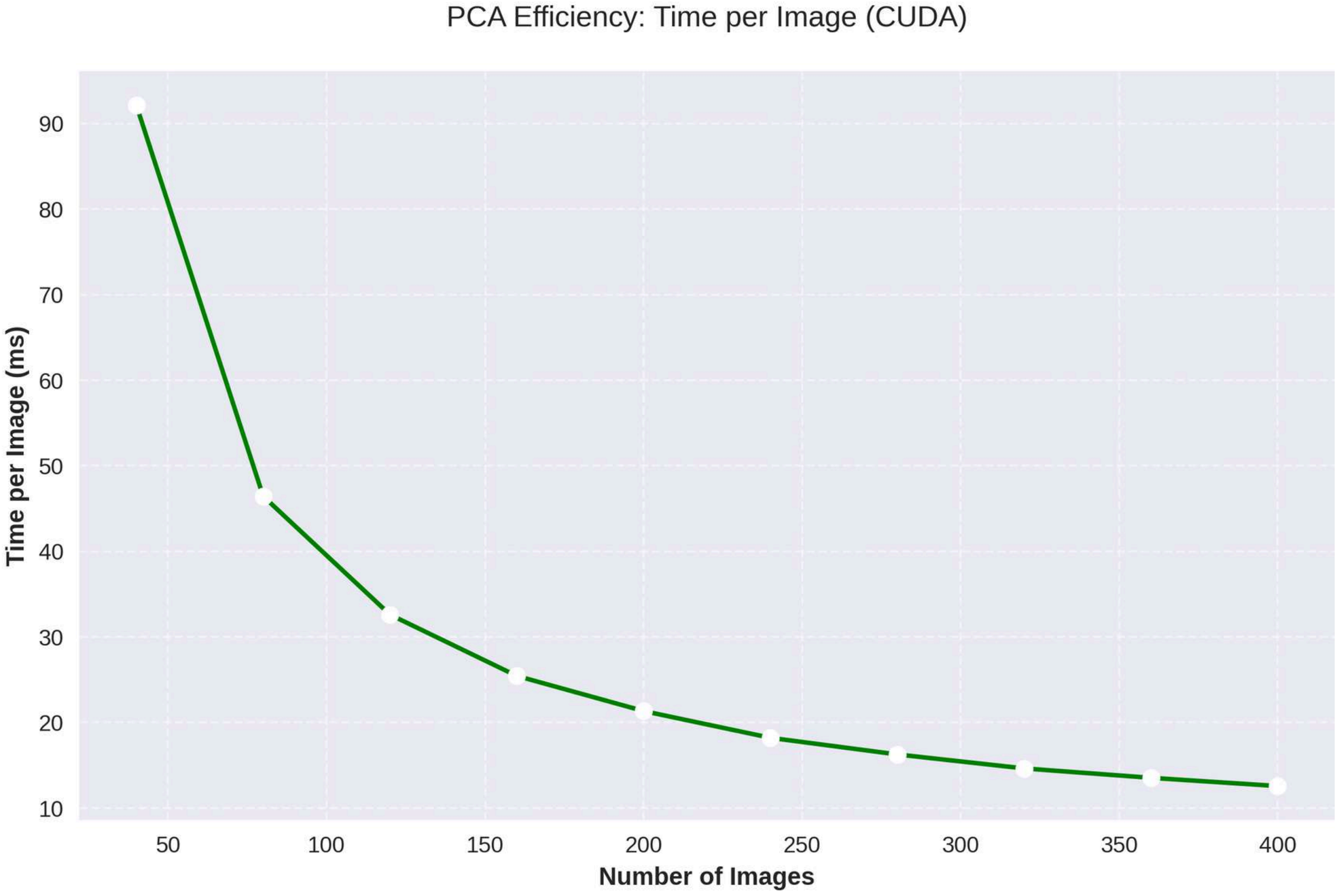

PCA Total Execution Time
(Parallel Implementation)
Total Time (seconds)
0
1
2
3
4
5
6
7
50
100
150
200
250
300
350
400
Number of Images

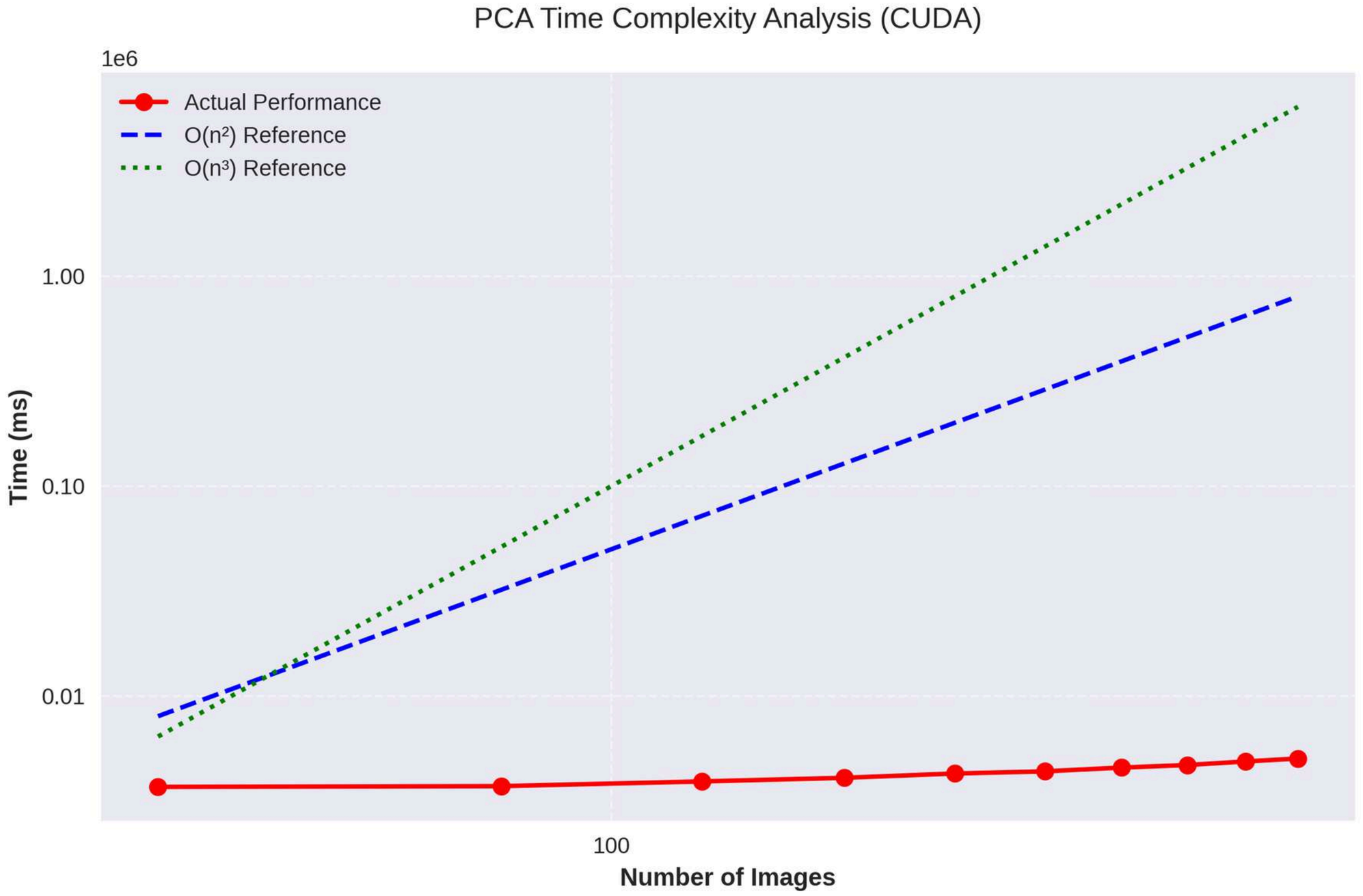

PCA Operation Time Breakdown (OMP)
Time (ms)
0
1000
2000
3000
4000
5000
6000
50
100
150
200
250
300
350
400
Number of Images
computeMean
centerData
computeCovarianceMatrix
computeEigenvectors
convertToFullEigenvectors
projectData
trainPCAModel

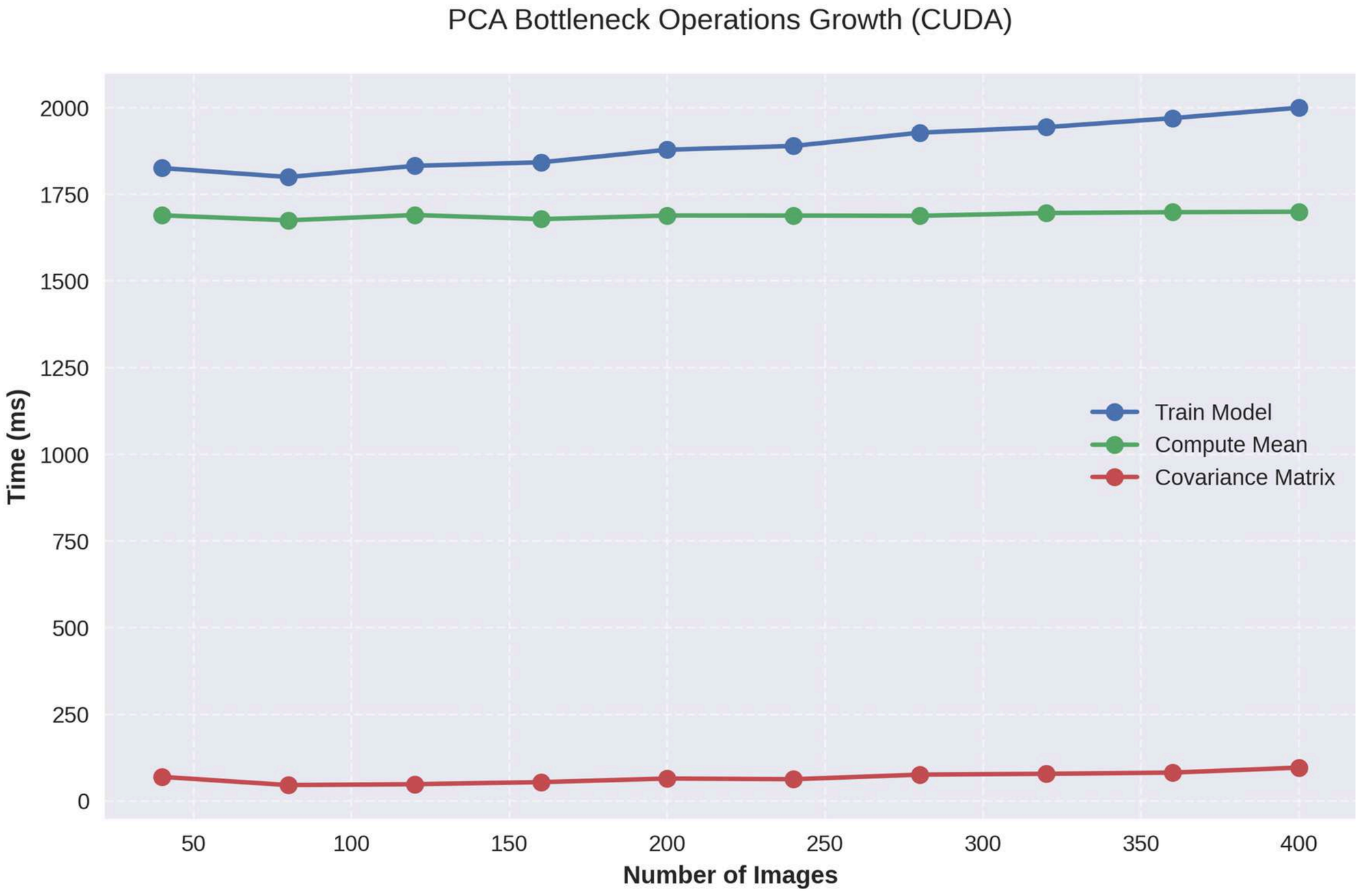

PCA Efficiency: Time per Image (OMP)
Time per Image (ms)
18
16
14
12
10
8
6
4
50
100
150
200
250
300
350
400
Number of Images

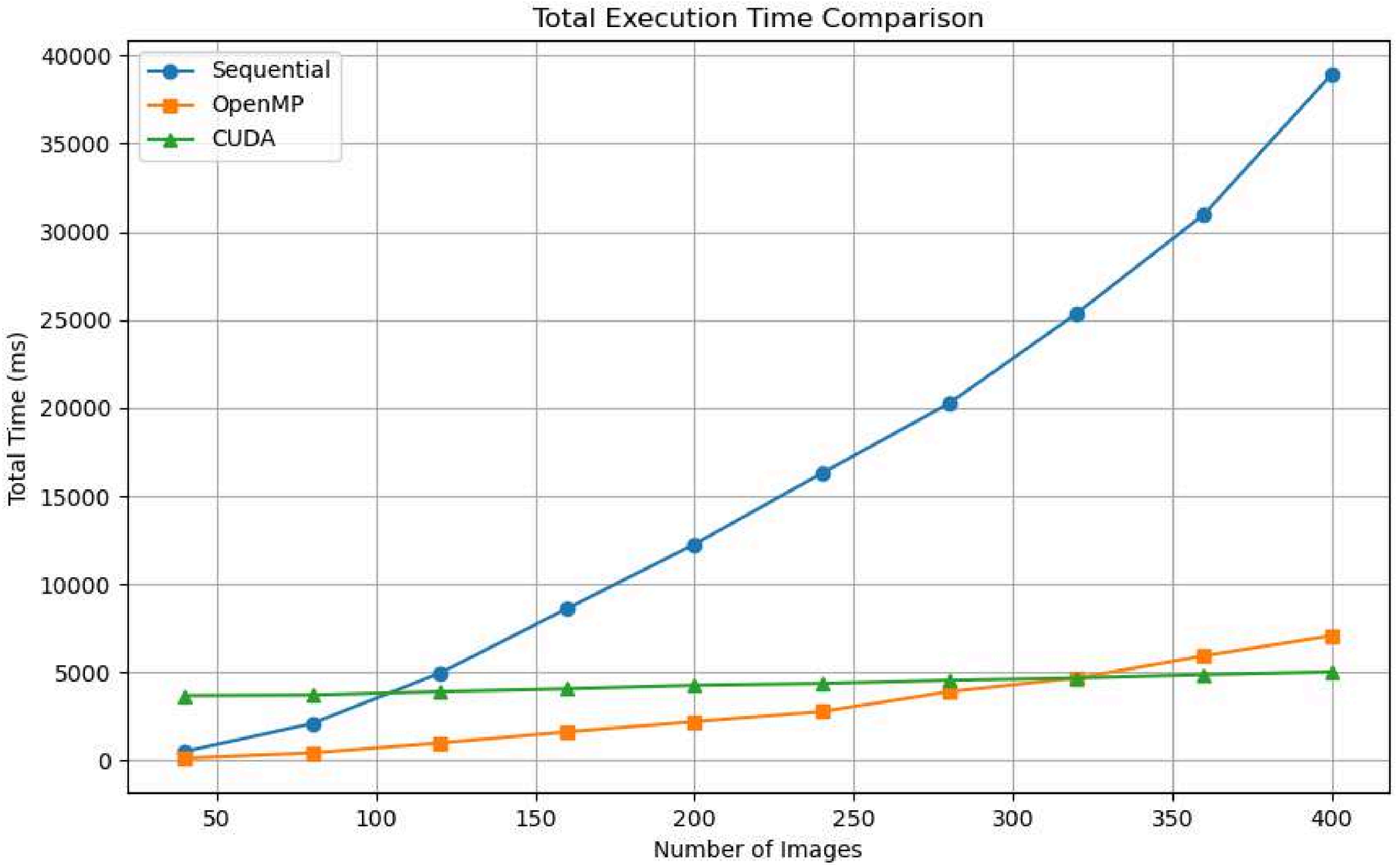

PCA Bottleneck Operations Growth (OMP)
Covariance Matrix
Train Model
Eigenvector Computation
Time (ms)
3000
2500
2000
1500
1000
500
0
50
100
150
200
250
300
350
400
Number of Images

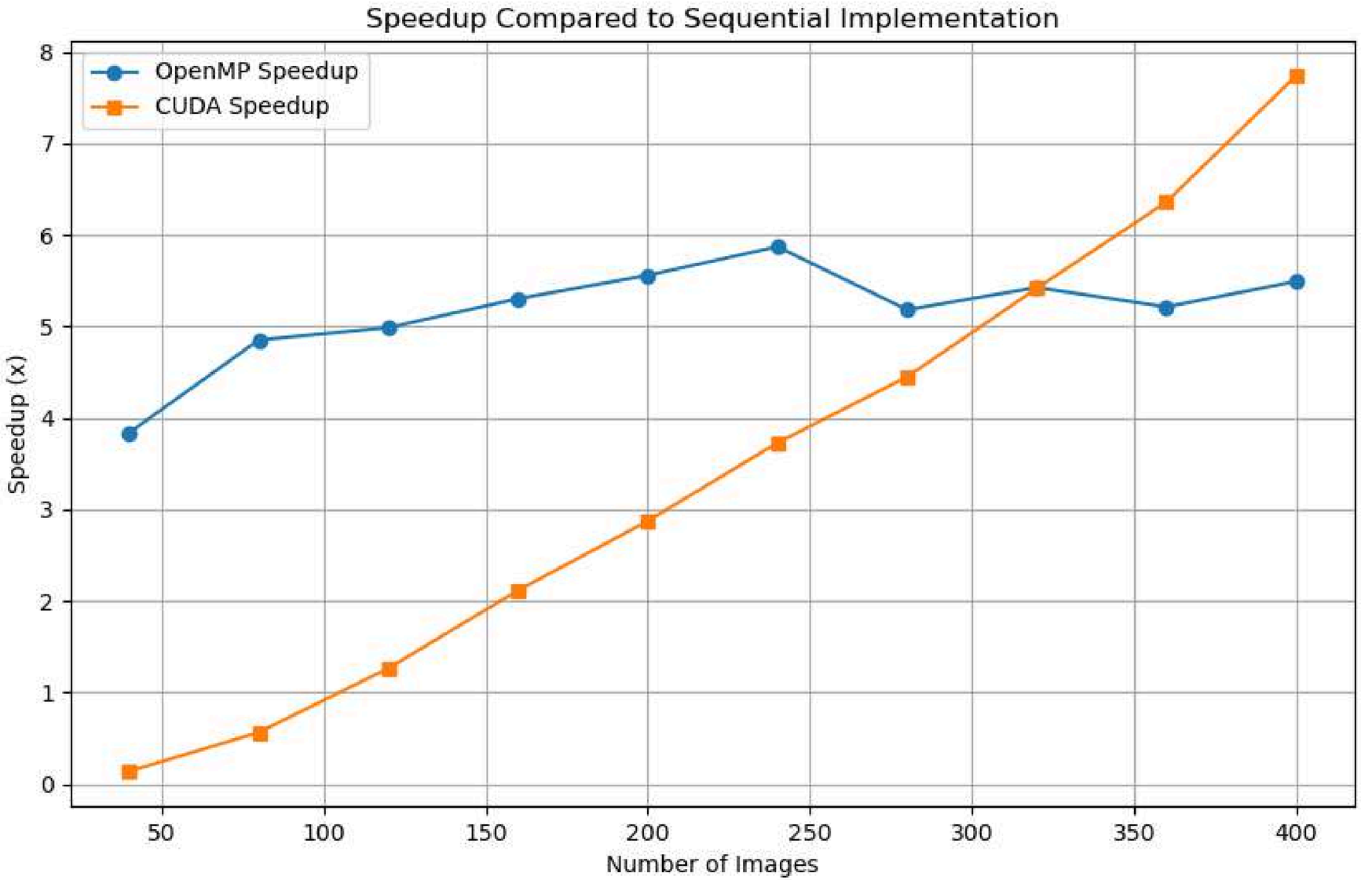

PCA Total Execution Time
(CUDA Implementation)
Total Time (seconds)
5.0
4.8
4.6
4.4
4.2
4.0
3.8
50
100
150
200
250
300
350
400
Number of Images

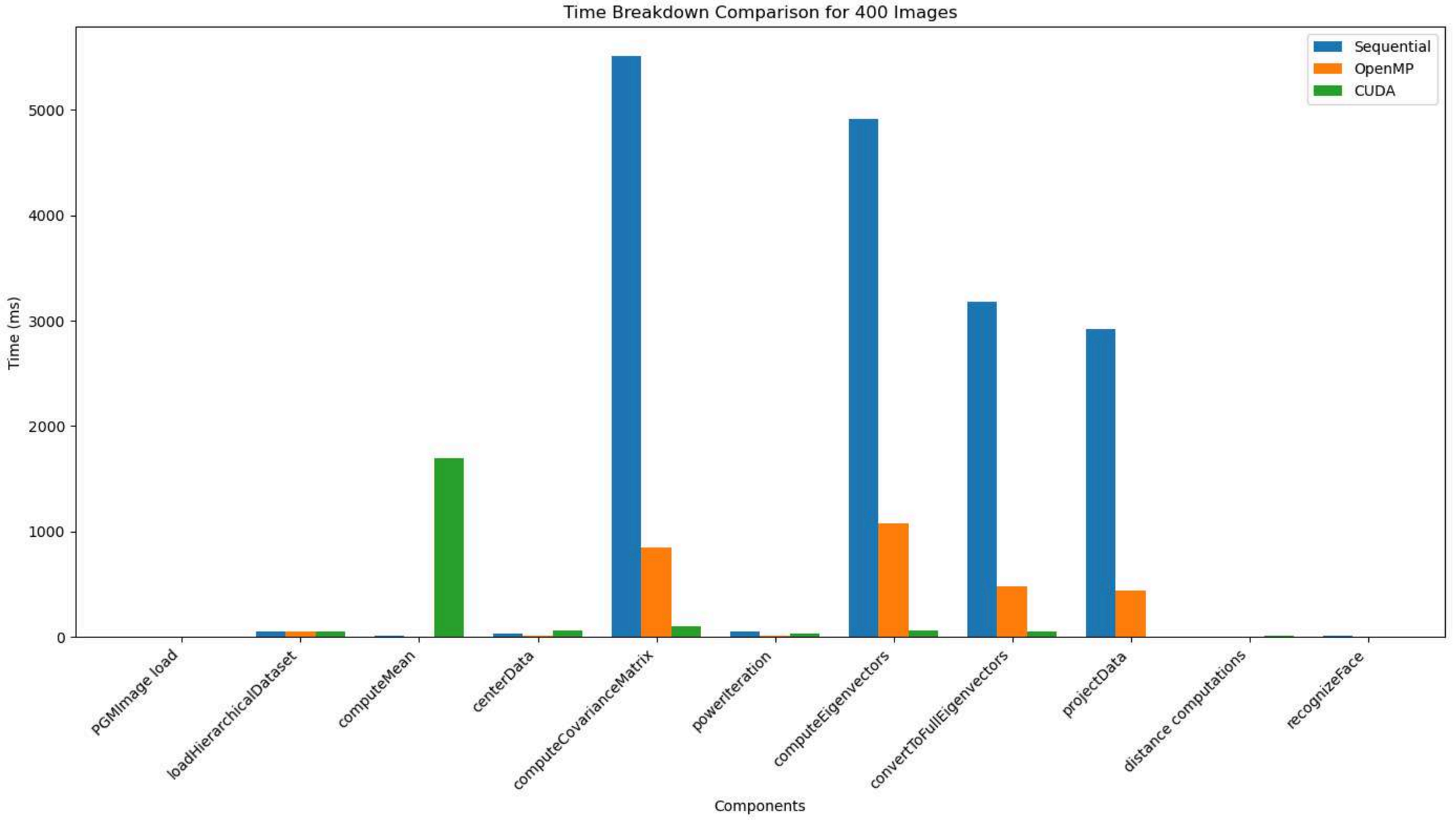

PCA Efficiency: Time per Image (CUDA)
Time per Image (ms)
90
80
70
60
50
40
30
20
10
50
100
150
200
250
300
350
400
Number of Images


PCA Time Complexity Analysis (CUDA)
1e6
Actual Performance
O(n²) Reference
O(n³) Reference
1.00
0.10
0.01
Time (ms)
100
Number of Images


PCA Bottleneck Operations Growth (CUDA)
Time (ms)
2000
1750
1500
1250
1000
750
500
250
0
50
100
150
200
250
300
350
400
Number of Images
Train Model
Compute Mean
Covariance Matrix


Total Execution Time Comparison
Sequential
OpenMP
CUDA
Total Time (ms)
Number of Images
0
5000
10000
15000
20000
25000
30000
35000
40000
50
100
150
200
250
300
350
400


Speedup Compared to Sequential Implementation
OpenMP Speedup
CUDA Speedup
Speedup (x)
Number of Images
0
1
2
3
4
5
6
7
8
50
100
150
200
250
300
350
400


Time Breakdown Comparison for 400 Images
Sequential
OpenMP
CUDA
Time (ms)
0
1000
2000
3000
4000
5000
PGMImage load
loadHierarchicalDataset
computeMean
centerData
computeCovarianceMatrix
powerIteration
computeEigenvectors
convertToFullEigenvectors
projectData
distance computations
recognizeFace
Components

APPLICATION IN FACIAL RECOGNITION – LDA – 1

- Supervised dimensionality reduction technique.
- Maximizes class separability while minimizing within-class variance.

1. **Data Preprocessing**
   - Load PGM images and normalize pixel values.
   - Compute the mean face vector.
   - Center the data by subtracting the mean face vector.

2. **Class-Specific Data Handling**
   - Group samples by class (person ID).
   - Compute within-class and between-class scatter matrices.

3. **PCA as Preprocessing**
   - Apply PCA to reduce dimensionality before applying LDA.

APPLICATION IN FACIAL RECOGNITION – LDA – 2

4. LDA Projection Matrix

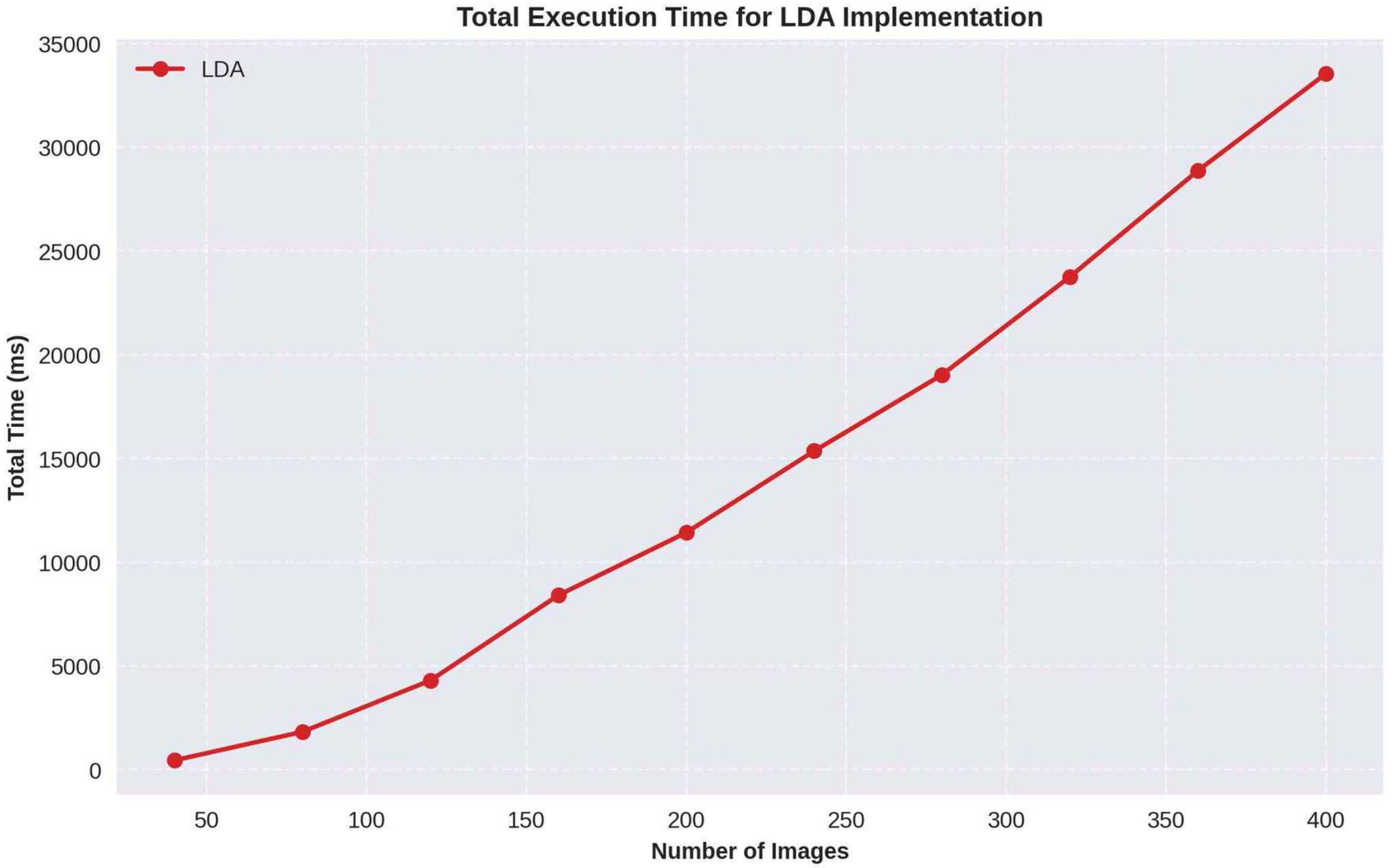
- Solve the generalized eigenvalue problem for SB and SW.
- Compute Fisherfaces in the PCA-reduced space.

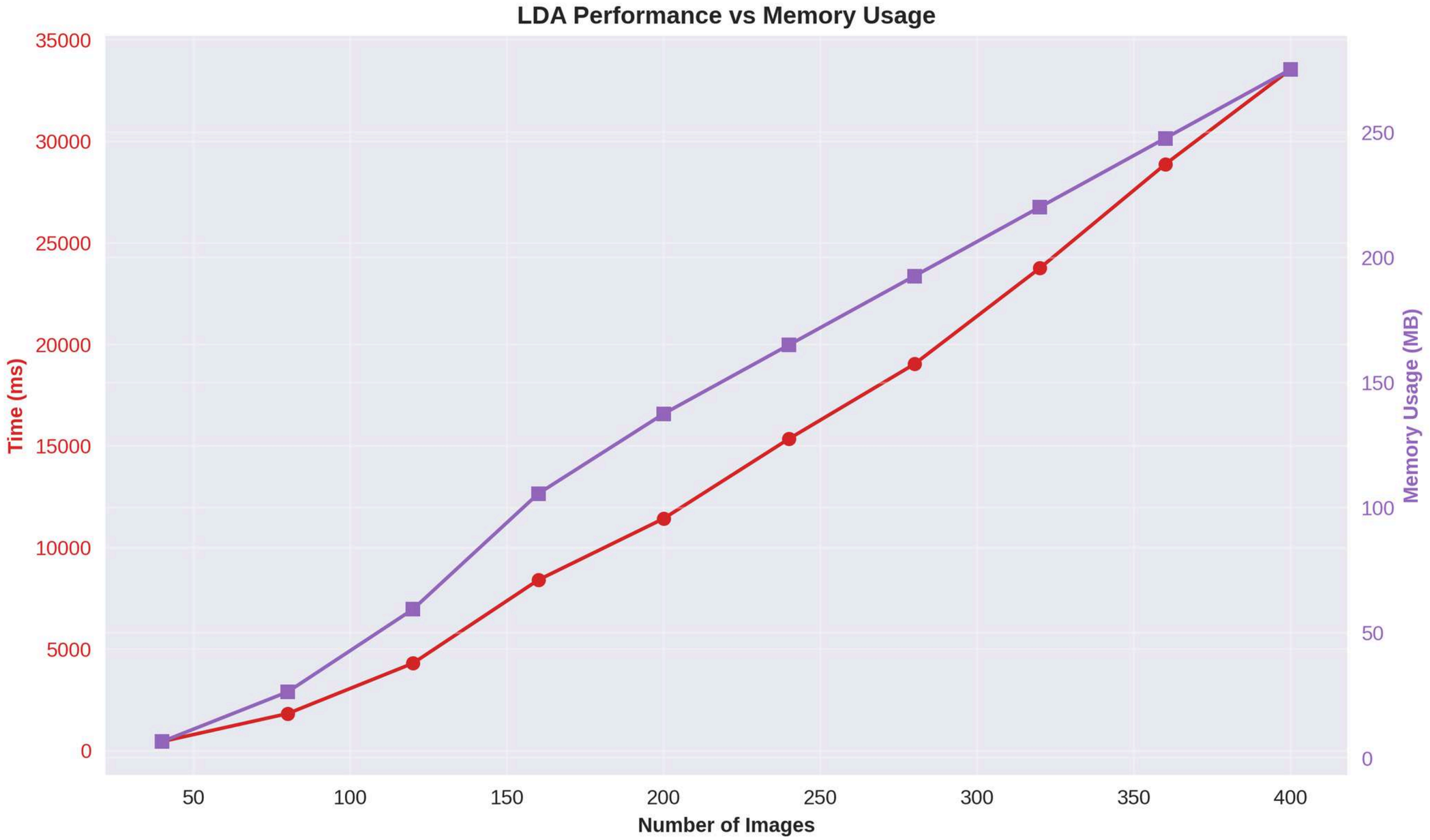
5. Projection and Recognition

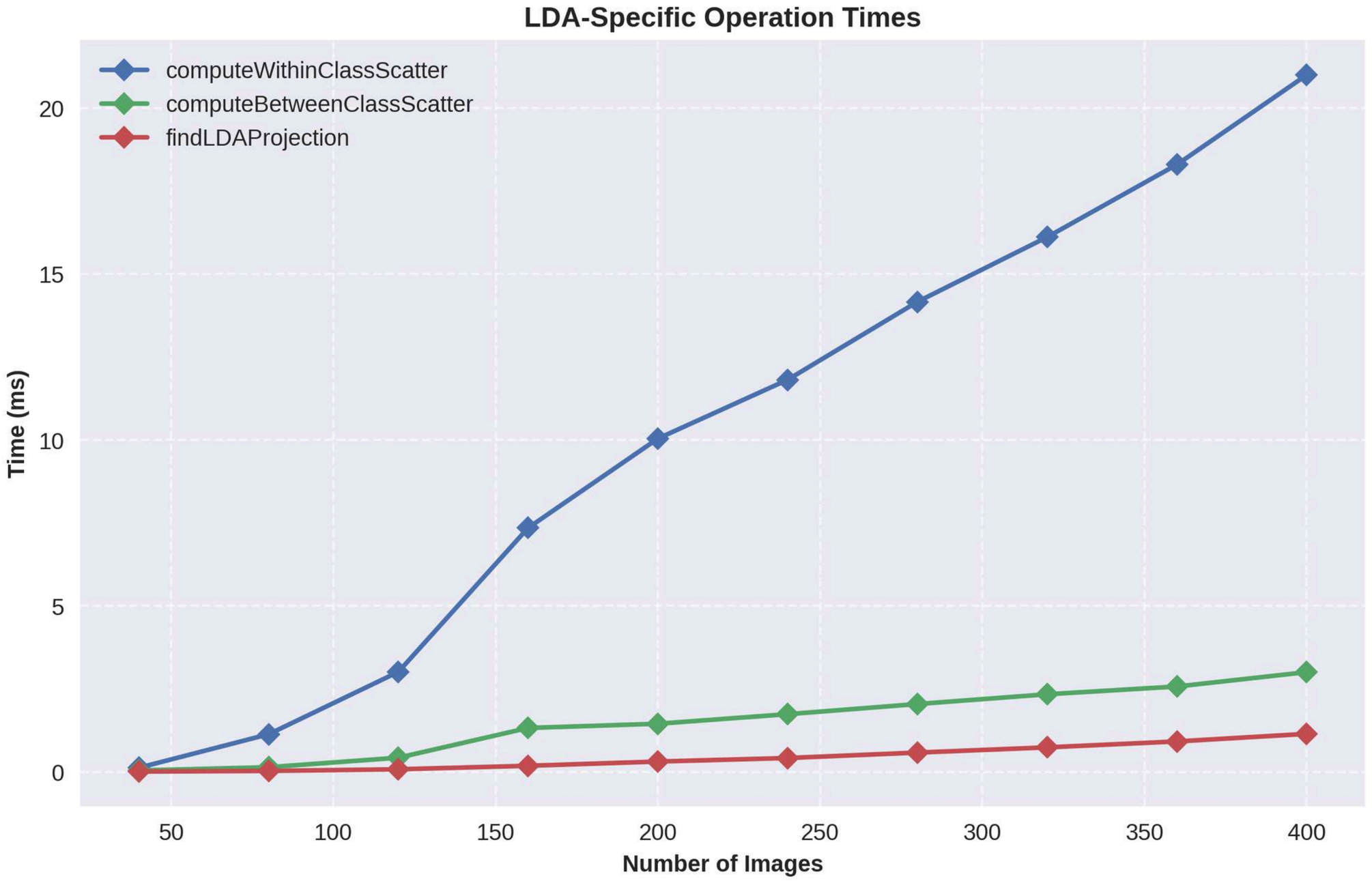
- Project training and test data onto Fisherfaces.
- Use Euclidean distance to find the nearest match.

6. Output

- Save the global mean face and Fisherfaces as PGM files.
- Display recognition accuracy and confusion matrix.


Total Execution Time for LDA Implementation
LDA
Total Time (ms)
0
5000
10000
15000
20000
25000
30000
35000
50
100
150
200
250
300
350
400
Number of Images


LDA Performance vs Memory Usage
Time (ms)
35000
30000
25000
20000
15000
10000
5000
0
Memory Usage (MB)
250
200
150
100
50
0
50
100
150
200
250
300
350
400
Number of Images


LDA-Specific Operation Times
computeWithinClassScatter
computeBetweenClassScatter
findLDAProjection
Time (ms)
0
5
10
15
20
50
100
150
200
250
300
350
400
Number of Images

AREAS FOR PARALLELIZATION – LDA 1

1. Normalization of Pixel Values

- Normalize images by dividing each pixel by the maximum value.
- Parallelization: Similar to PCA, distribute computation across threads or CUDA blocks.

2. Global Mean Computation

- Compute the global mean vector for all training images.
- Parallelization: Sum the pixel values across images in parallel.

3. Within-Class Scatter Matrix (SW) Computation

- Compute scatter matrices for each class and sum them.
- Parallelization: Process each class independently, and parallelize within-class computations.

AREAS FOR PARALLELIZATION – LDA 2

4. Between-Class Scatter Matrix (SB) Computation

- Compute the scatter matrix for the difference between class means and the global mean.
- Parallelization: Each class's computation can run independently.

5. PCA Preprocessing

- Reduce dimensionality using PCA before LDA.
- Parallelization: Apply the same parallelization techniques as in PCA.

6. LDA Projection Matrix Computation

- Solve the generalized eigenvalue problem for SW and SB.
- Parallelization: Parallelize the eigenvalue and eigenvector computation.

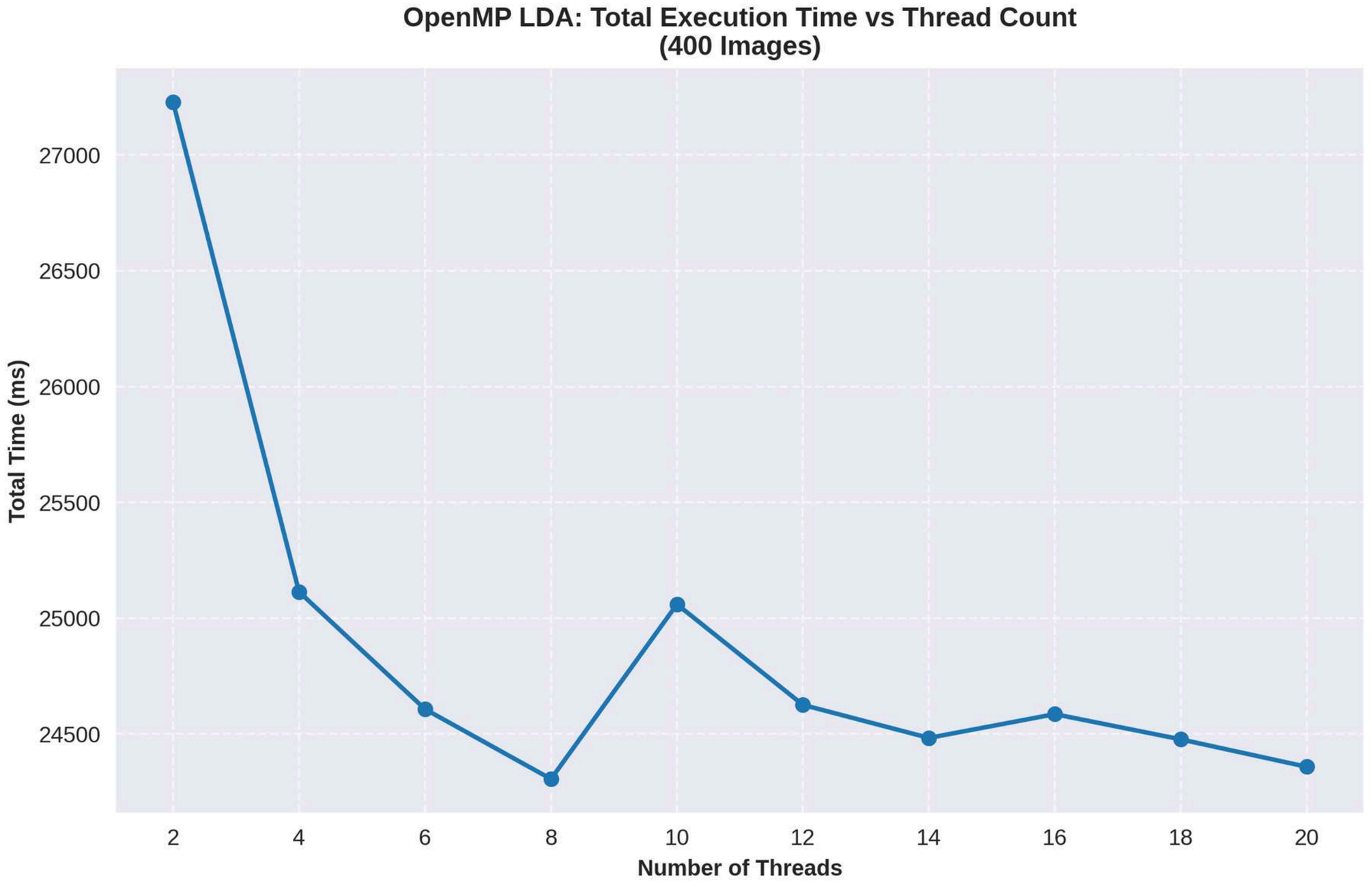
AREAS FOR PARALLELIZATION – LDA 3

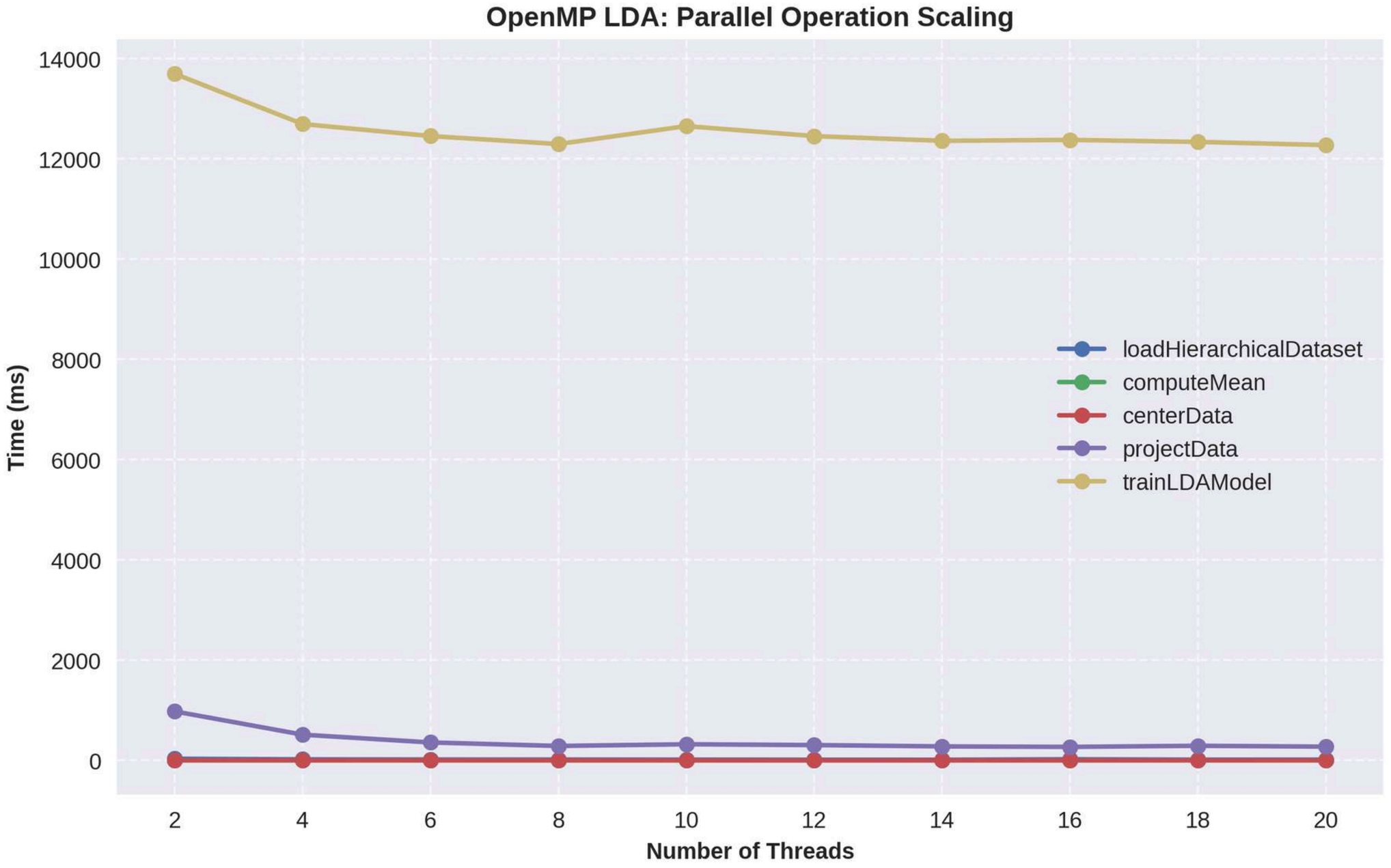
7. Projection onto Fisherfaces

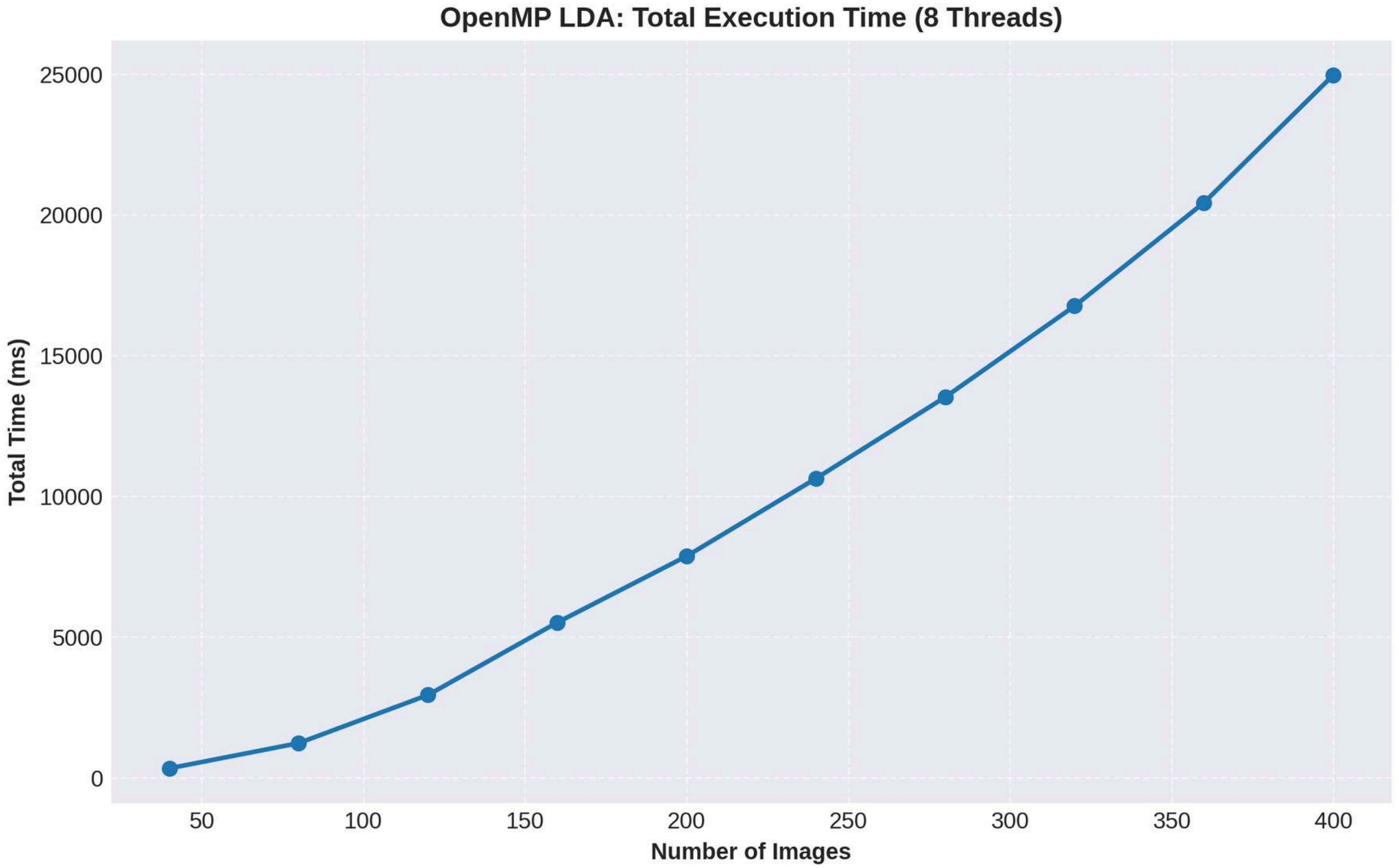
- Project the training and test data onto the Fisherfaces.
- Parallelization: Process each image independently during projection.

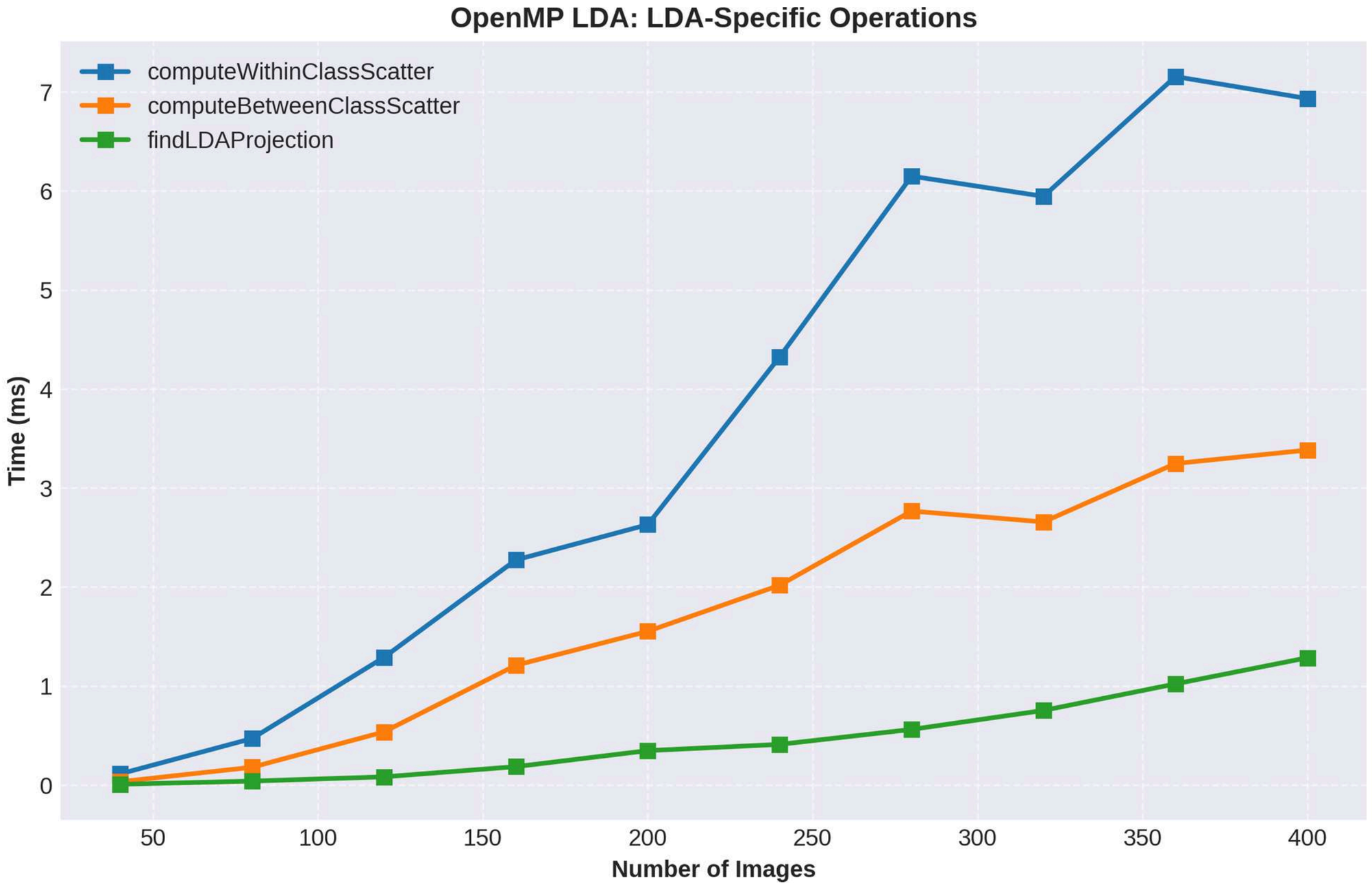
8. Reconstruction

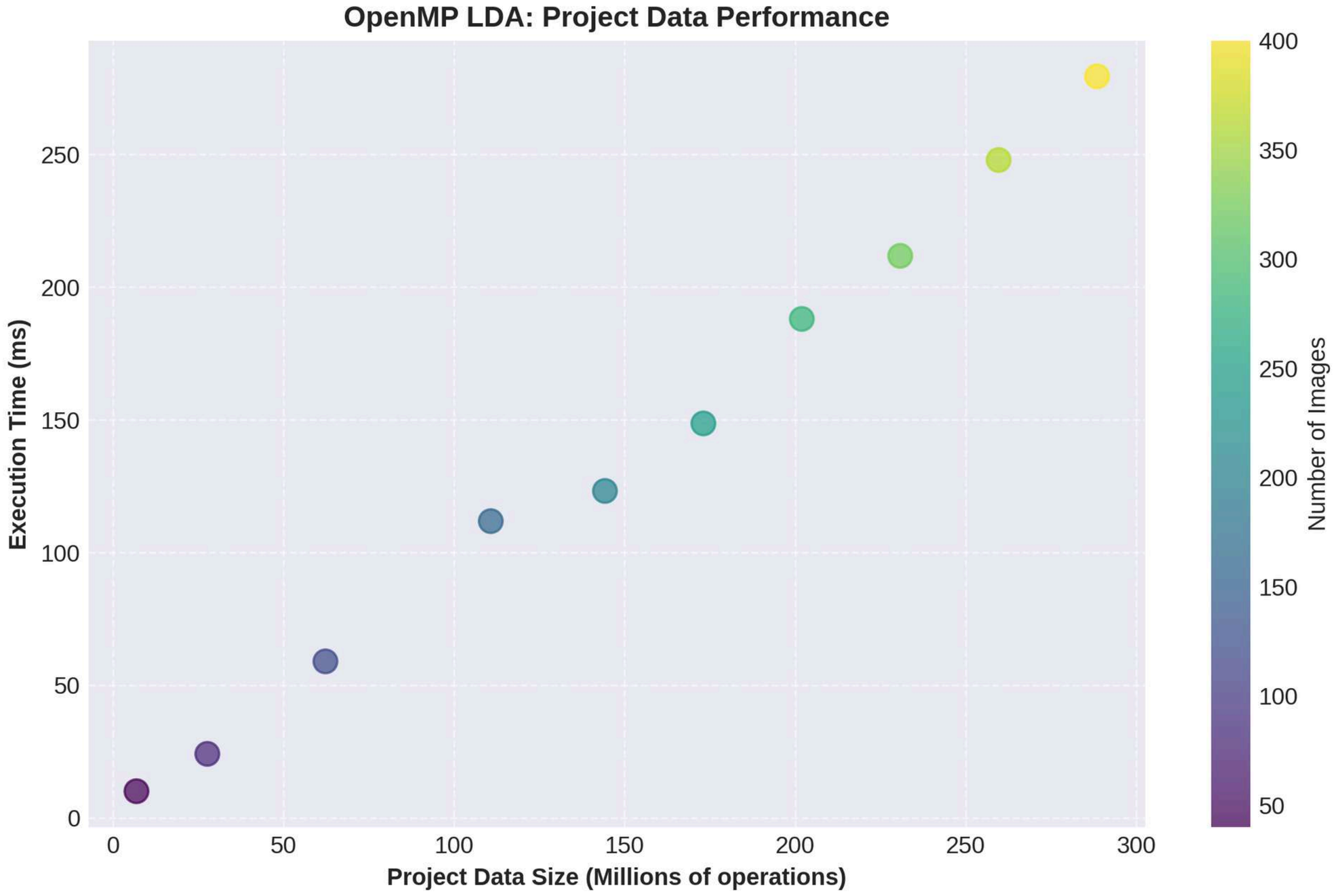
- Reconstruct images from their projections.
- Parallelization: Distribute reconstruction computations across threads or CUDA blocks.

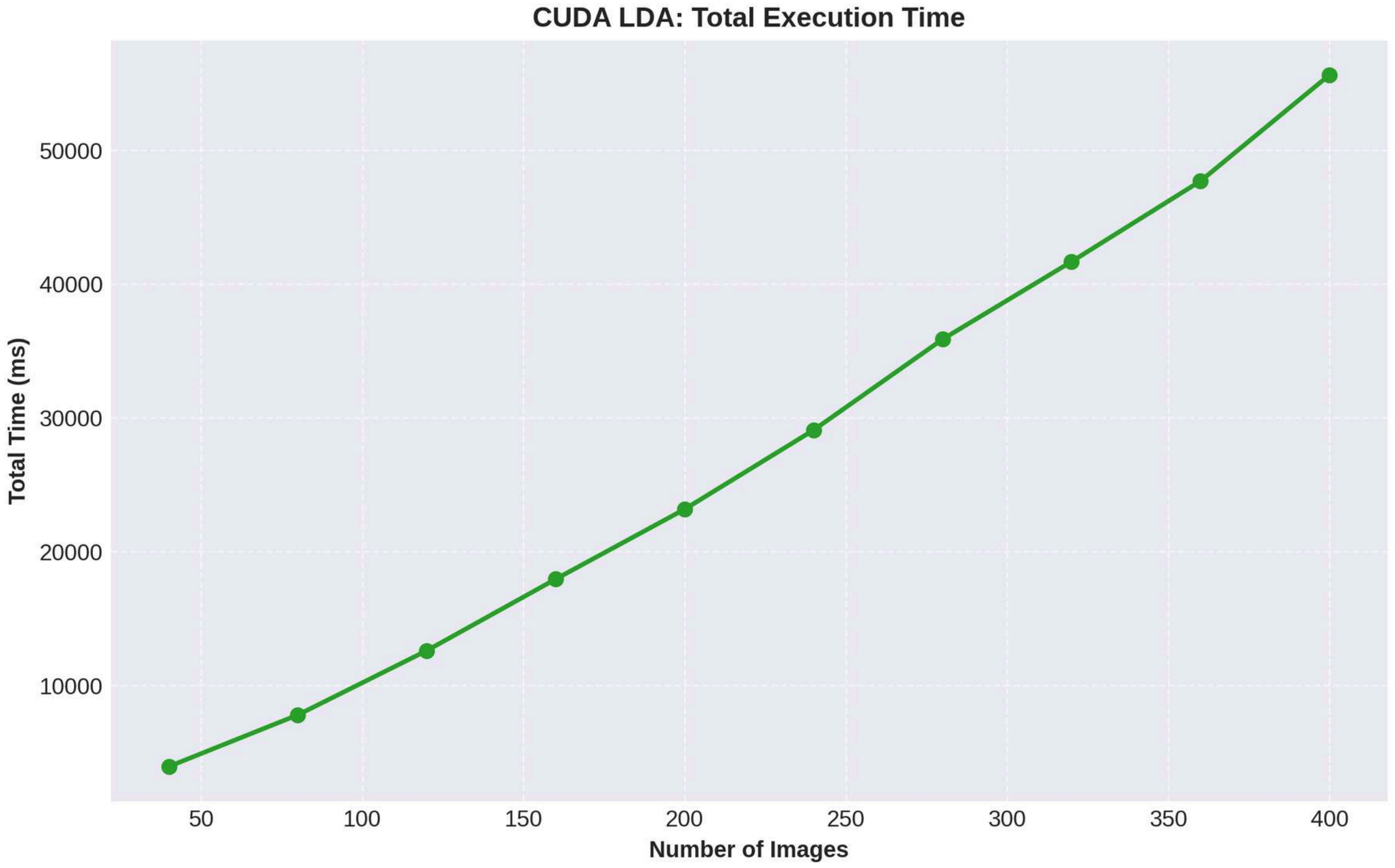

OpenMP LDA: Total Execution Time vs Thread Count
(400 Images)
Total Time (ms)
27000
26500
26000
25500
25000
24500
2
4
6
8
10
12
14
16
18
20
Number of Threads

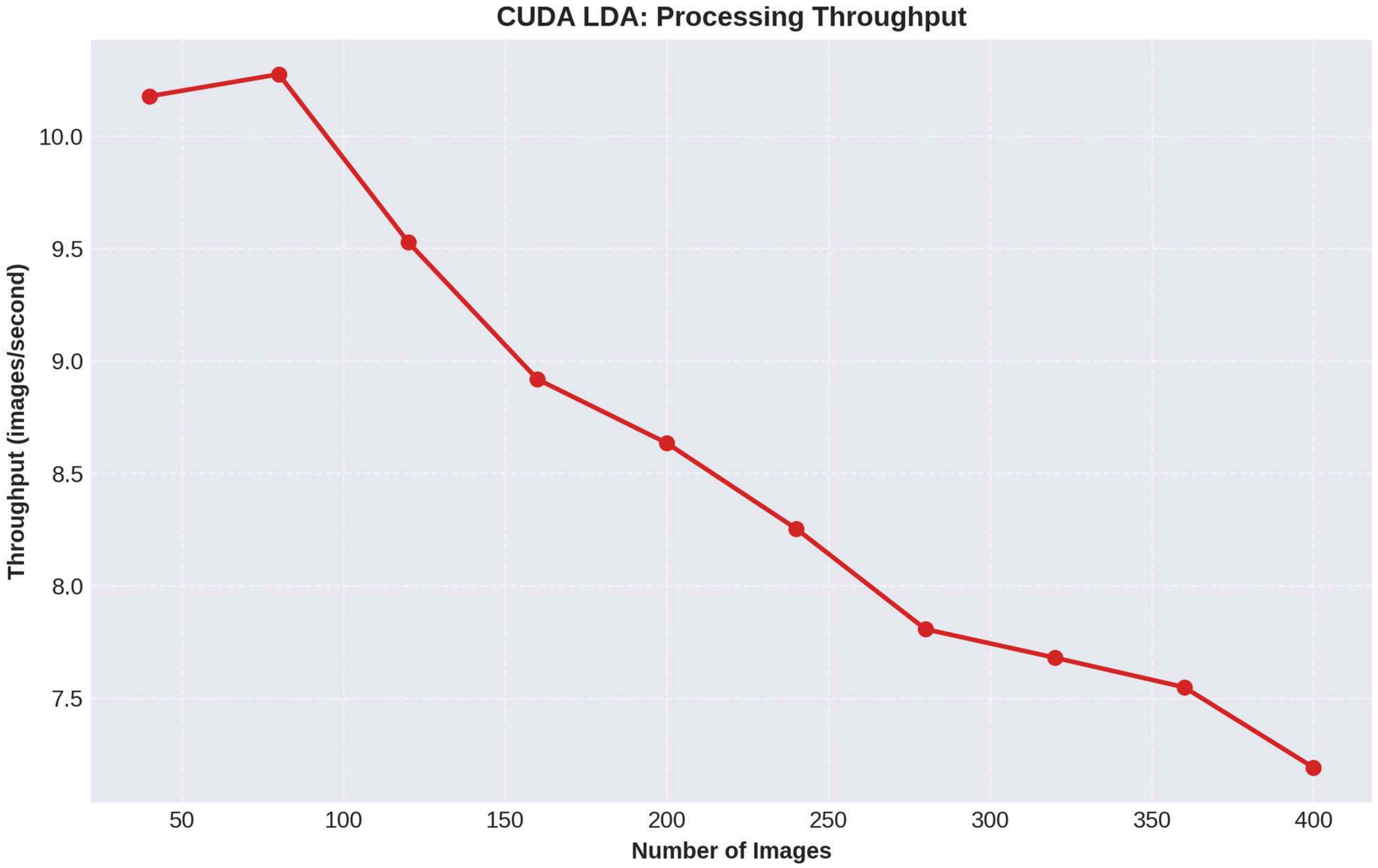

OpenMP LDA: Parallel Operation Scaling
Time (ms)
14000
12000
10000
8000
6000
4000
2000
0
2
4
6
8
10
12
14
16
18
20
Number of Threads
loadHierarchicalDataset
computeMean
centerData
projectData
trainLDAModel

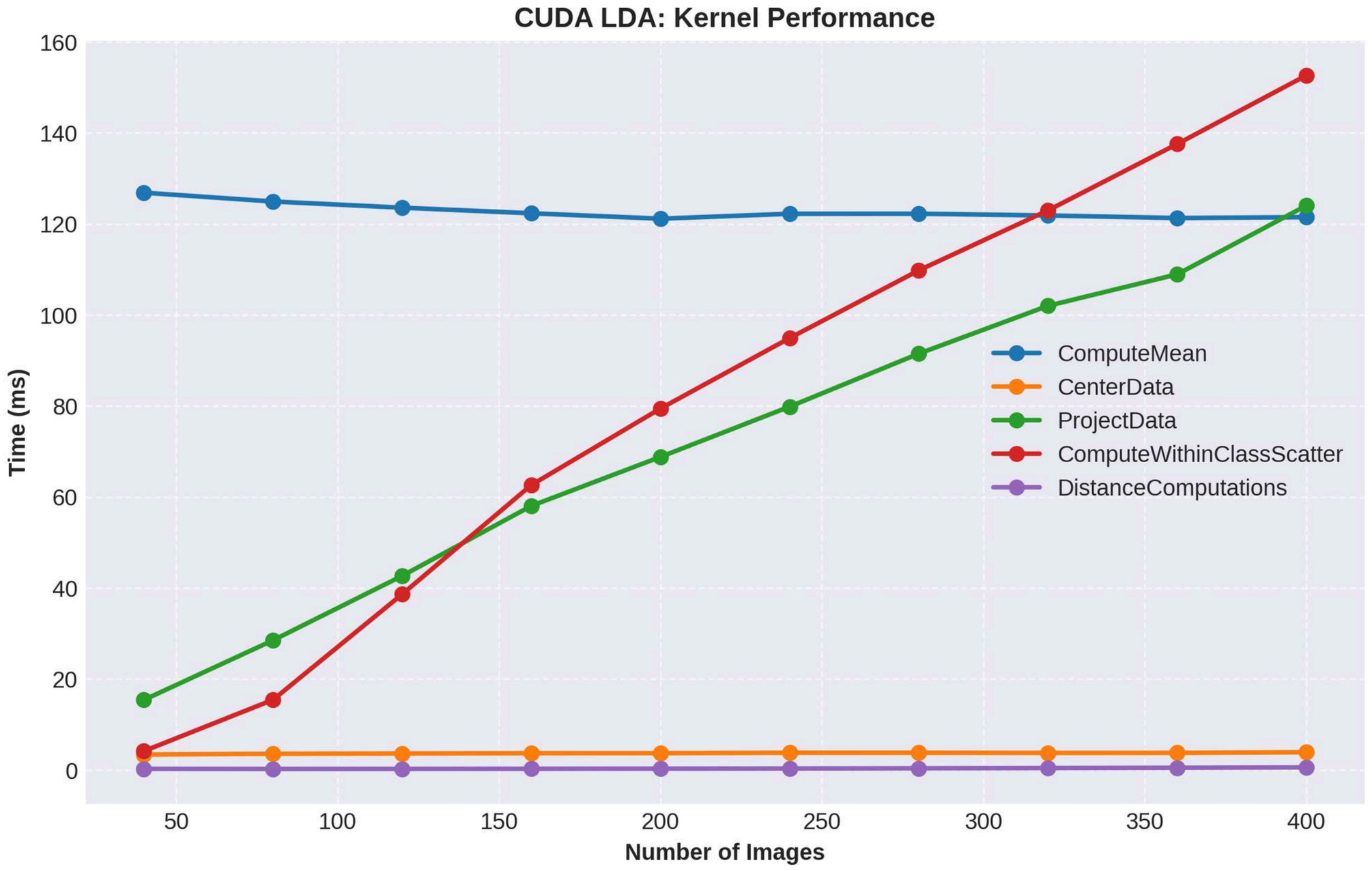

OpenMP LDA: Total Execution Time (8 Threads)
Total Time (ms)
25000
20000
15000
10000
5000
0
50
100
150
200
250
300
350
400
Number of Images

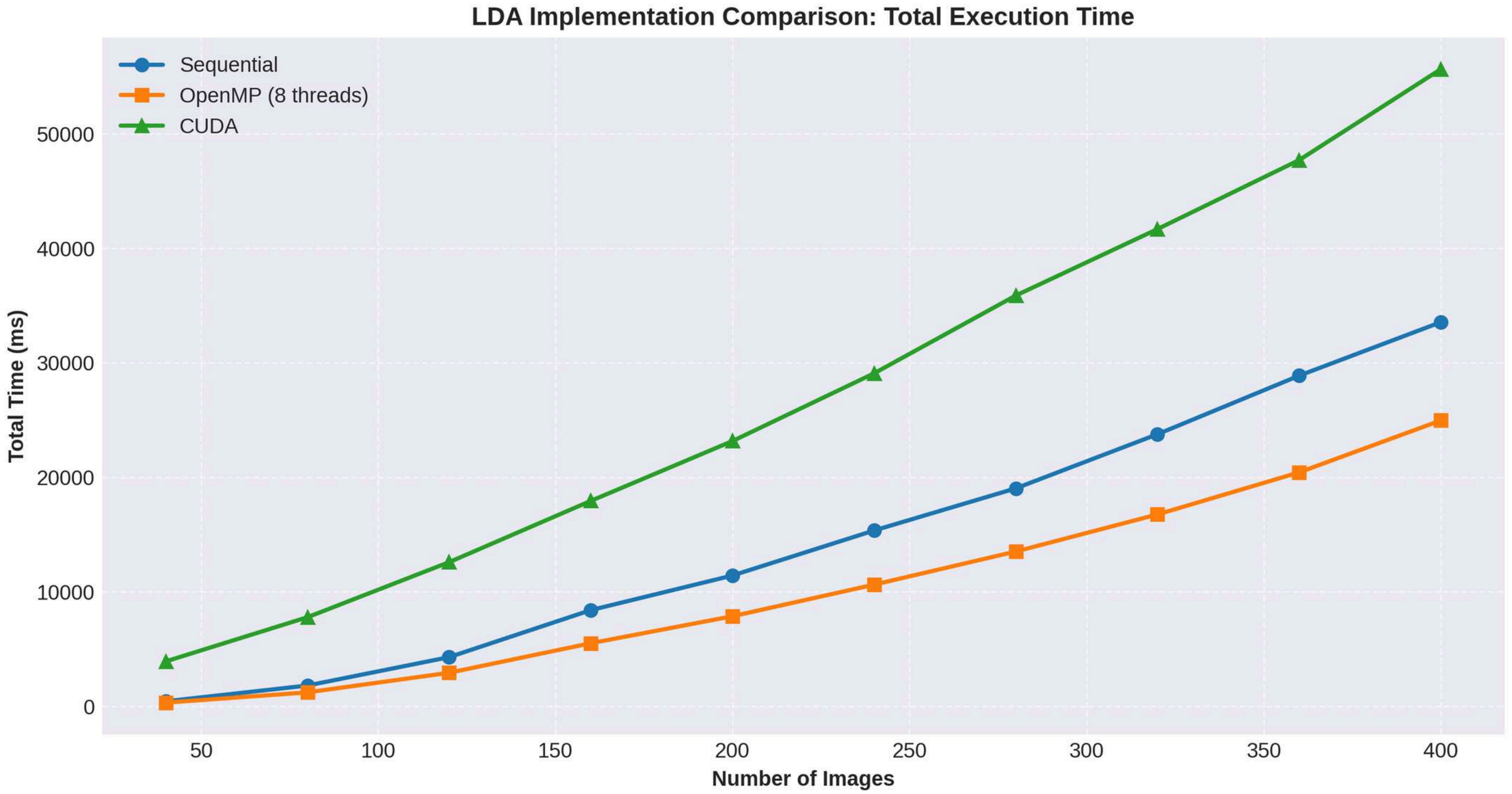

OpenMP LDA: LDA-Specific Operations
computeWithinClassScatter
computeBetweenClassScatter
findLDAProjection
Time (ms)
0
1
2
3
4
5
6
7
50
100
150
200
250
300
350
400
Number of Images

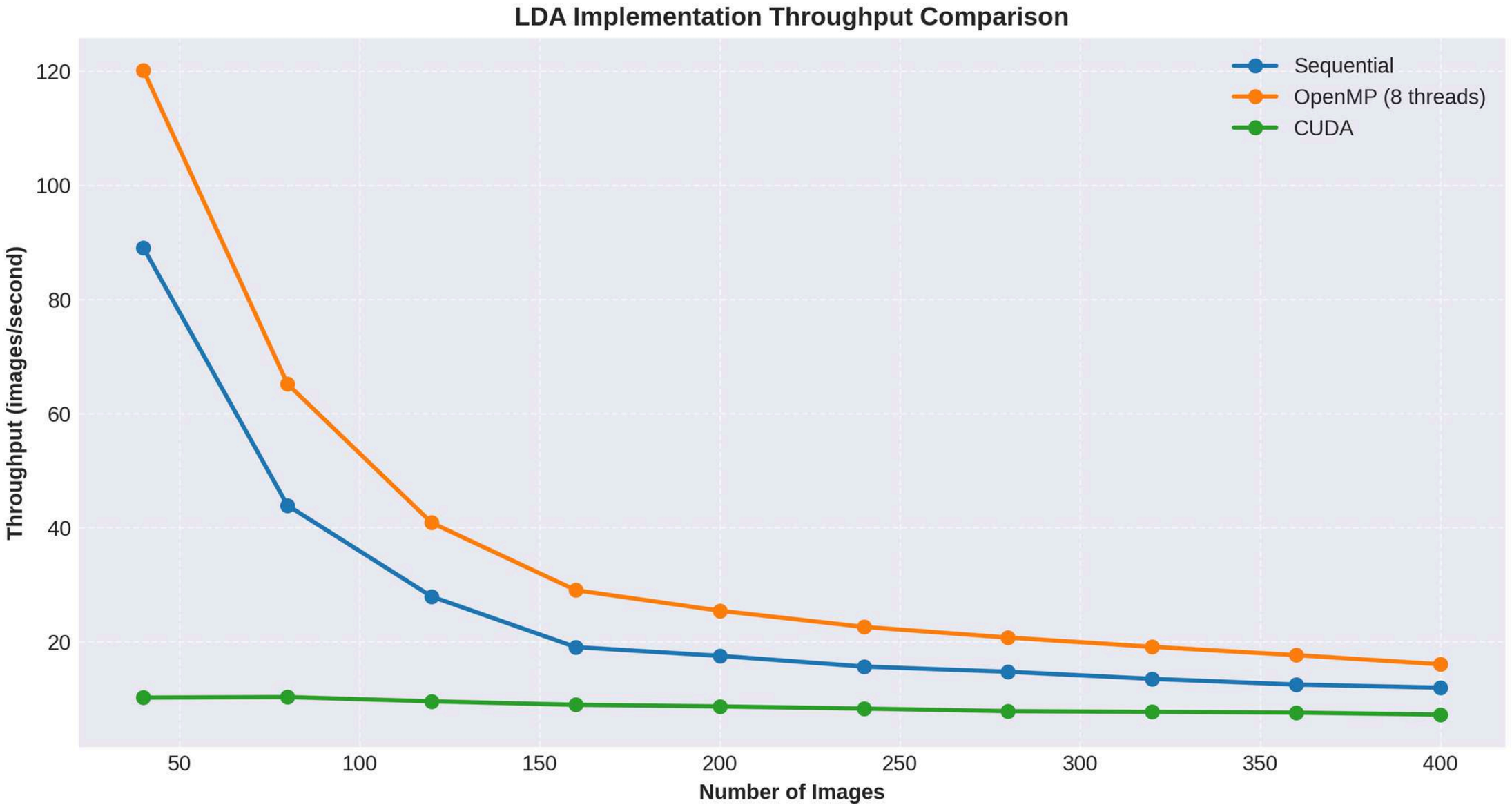

OpenMP LDA: Project Data Performance
Execution Time (ms)
0
50
100
150
200
250
Project Data Size (Millions of operations)
0
50
100
150
200
250
300
Number of Images
50
100
150
200
250
300
350
400


CUDA LDA: Total Execution Time
Total Time (ms)
50000
40000
30000
20000
10000
50
100
150
200
250
300
350
400
Number of Images


CUDA LDA: Processing Throughput
Throughput (images/second)
10.0
9.5
9.0
8.5
8.0
7.5
50
100
150
200
250
300
350
400
Number of Images


CUDA LDA: Kernel Performance
Time (ms)
Number of Images
0
20
40
60
80
100
120
140
160
50
100
150
200
250
300
350
400
ComputeMean
CenterData
ProjectData
ComputeWithinClassScatter
DistanceComputations


LDA Implementation Comparison: Total Execution Time
Sequential
OpenMP (8 threads)
CUDA
Total Time (ms)
0
10000
20000
30000
40000
50000
50
100
150
200
250
300
350
400
Number of Images


LDA Implementation Throughput Comparison
Sequential
OpenMP (8 threads)
CUDA
Throughput (images/second)
120
100
80
60
40
20
50
100
150
200
250
300
350
400
Number of Images


LDA Implementation Speedup Comparison
OpenMP (8 threads)
CUDA
Speedup (x) vs Sequential
0.2
0.4
0.6
0.8
1.0
1.2
1.4
50
100
150
200
250
300
350
400
Number of Images