Optimization of Eigenface Algorithm for Facial Recognition Using AVX2

A Discussion Paper Presented to
the Faculty of the College of Computer Studies
De La Salle University

In Partial Fulfillment
of the Requirements for the Degree of
Bachelor of Science in Computer Science

by
Ibaoc, Christian Gabriel P.
Sia, Chantal Hyacynth S.
Valero, Nigel Kristoffer C.
Veloso, Antonio Luiz G.

Submitted to:
Professor Roger Luis Uy

July 30, 2024

Table of Contents

I. Discussion of AVX2 Implementation

The `eigen.asm` file contains the implementation of the parallel Eigenface algorithm using AVX2, generally employing vector operations on packed/ymm registers containing matrix pointers to be able to operate on more than one matrix element at a time.

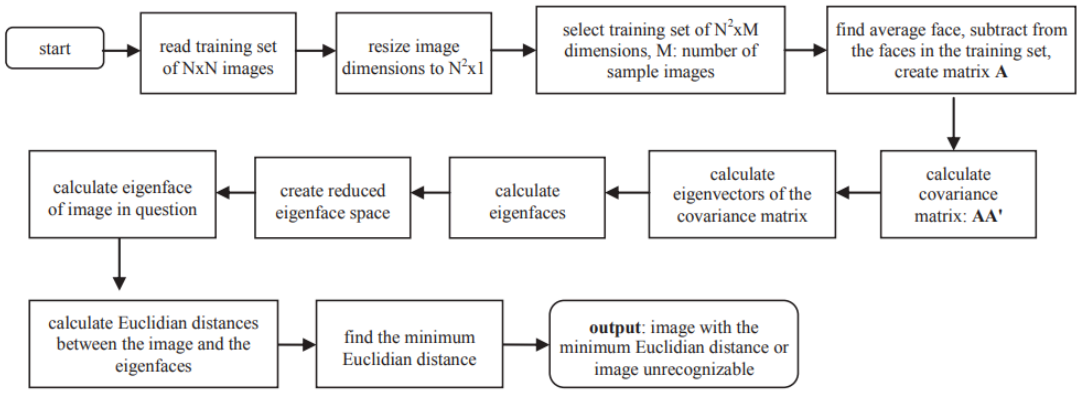

Figure 1.1. Flowchart of the Eigenface Algorithm

The following functions and explanations are used to implement parallelism to the Eigenface algorithm or to make specific steps of the algorithm more efficient:

A. asmNorm

asmNorm takes place in the "find average face" part of the algorithm, and it basically adds the pixel values of the training set array using ymm registers and packed vector addition until there is only one column left containing the sum of the training images, which will be divided to find the average image.

B. asmNormDiv

asmNormDiv takes place after asmNorm in the training phase, and it outputs the "average face" by dividing the sum of training images by the number of training images. Despite being a simple step at first glance, the AVX2 implementation is complicated by AVX2's limited ability to divide integers. This

causes this function to require multiple steps of conversion (mostly to single-precision floating-point) to allow for packed vector division and to undo previous conversions. Due to the complications caused by these conversions, the height and width of the training images are limited to integers divisible by 32.

C. subtNorm

In this function, the "average face" given by asmNormDiv is simply subtracted from the values in the training set array. The packed vector subtraction doesn't require prior conversions, so it is not as complicated as the previous function.

D. covMat

The covariance matrix is found by multiplying the normalized training set array with its transverse. This function takes in the normalized training set array and its transverse and performs matrix multiplication between them. By using vector packed multiply and add on packed registers containing the values of the training set array and its transverse, adding the results, and dividing it by the number of elements to process, the function outputs the value of a single normalized element of the covariance matrix at a row and column specified in the C++ file. As such, this function is called for each element of the covariance matrix.

E. jacobify

This function implements the Jacobian Rotation algorithm [1] to basically arrive at the eigenvalues and the eigenvectors from the covariance matrix. Specifically, the (estimated) diagonal of the output Jacobian matrix contains the eigenvalues and the matrix of eigenvectors (with column indices corresponding to the column index of the eigenvalues) is produced as an excess output.

Many of the steps for the Jacobian Rotation happen in the C++ file, but to keep the discussion related to AVX2 and parallelism, the jacobify function basically performs matrix multiplication on two given matrices. It does this

through three levels of loops, the outermost of which (greaterloopy) iterates over each row, the second of which (jacobyloop) iterates over each column, and the innermost of which (innerloop) finds the value of a single element in the product matrix. Of course, this makes use of vector multiplication and addition.

F. asmweights

The asmweights function uses vector operations to perform matrix multiplication between the training images and the eigenvectors, similar to previous functions that use matrix multiplication. The output is stored in a general weight matrix.

G. eigPart1 and eigPart2

Both eig functions use matrix multiplication like before through vector operations, with a difference in inputs and outputs. For part 1, the inputs used are the general weight matrix and the image vector of a specific image are used to get a weight vector that will "represent" the specific image. Part 2 uses this weight vector and performs matrix multiplication with the general weight matrix to produce the final eigenface of the image, a matrix with equal size to a normal image vector but weighted based on the general weight matrix

These functions are used not just in the training phase but also in the recognition phase, which occurs right after the training phase ends.

II. Comparison of C++ and AVX2 Implementation

To compare the execution times (for the training phase) of the C++ and AVX2 implementations of the Eigenface algorithm, the `#define setting` line in the Eigenface.cpp file must be set to true for the AVX2 implementation and false for the C++ implementation. Furthermore, since the execution time of the Eigenface algorithm is dependent on the number of training images, the execution time of both implementations can also be compared with a different number of training images. This can be done by setting the value of `#define trsize` to the desired number of training images.

# of Training Images	Sequential ExecutionTime (ms)	Parallel Execution Time (ms)
4	3272	1340
8	16320	2495
12	65014	18257
16	174213	44077

Table 1.1. Execution Time Comparison Table

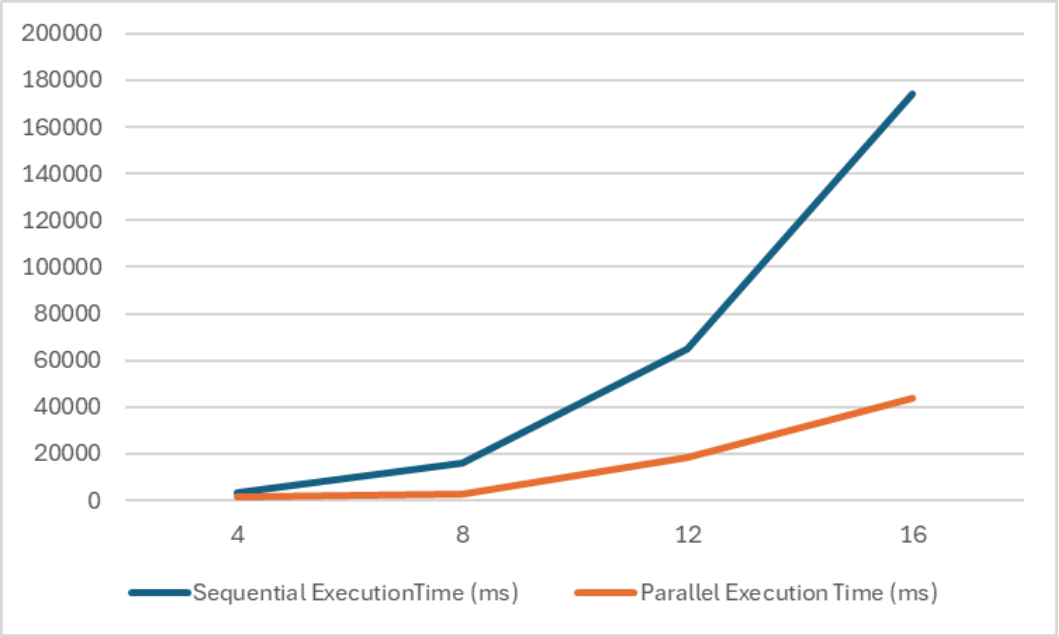

Figure 2.1. Execution Time Comparison Graph

Figure 2.2. Execution Time Captures

From what can be observed from the table and the line graph, the execution time for the C++ implementation is both much higher than the execution time for the AVX2 implementation and grows at a much faster rate. This shows that implementing parallelism to the Eigenface drastically speeds up the program setup.

References

[1] https://phys.au.dk/~fedorov/Numeric/now/Book/eigen.pdf