CSE 625 Term Project Report

Eigen Face Analysis with CUDA and PyTorch

Caleb Klenda

12-07-2022

# Project statement and objective

The purpose of this project is to compare the effectiveness of different parallelized computing techniques. A portion of the project is dedicated to doing this using OpenMP, an open source library for C++.

For the second part of the project, the celebA dataset is used to perform principal component analysis and to compute eigen faces of the images. The timing of each step of this process is done with the CPU and then the GPU and timings compared. The objective is to analyze the improvement provided by GPU processing, to evaluate the effectiveness of the eigenface computation method and to discuss the uses of the computed eigen faces.

# Approach

The overall approach to this project was two-fold. For the first part, the provided all\_pairs Codeblocks project was used as a reference to create OpenMP versions of all the functions within the project. These functions were then measured for performance on various sizes of the overall matrix that they would compute. The methodologies provided in the project were compared against their OpenMP counterparts.

The second part of the project leans heavily on the use of the PyTorch library to perform computations. PyTorch allows access to matrix operations that utilizes CUDA cores in the GPU used for this project. Even still, the size of the data is so large that only a subset of the total images is used. Images are first modified using Principal Componant Analysis to reduce their dimensionality. Images are also set to grayscale as part of the preprocessing step. The reason for choosing grayscale is that preliminary testing showed better results and the overall process was simplified by ignoring the color aspect of the images. Then, the average face was computed and subtracted from each image to form and image with only the most unique aspects remaining. The covariance of this batch of tensors was computed. From the covariance, the eigen vectors were exptracted and the best k eigen vectors (based on magnitude of their respective eigen values) kept. Each of these eigenvectors, when multiplied with the mean subtracted image, forms one of the eigen faces. The sum of these eigen faces returns the original mean subtracted image.

**Hardware Used:**

The project was all performed on my home computer with the following specifications:

* **CPU**

Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz

AVX2 (256-bit MM registers)

10 cores / 20 threads

20 MB Intel Smart Cache (L3-cache)

* **RAM**

32 GB DDR4 RAM

* **GPU**

TUF RTX3080 (Ampere GPU)

8704 CUDA cores  
 5 MB of L2-Cache

10GB GDDR6X

* **OS**

Windows

**Software Used:**

* Python 3.10
* Cuda Compilation tools 11.8

# Implementation

**Section 1: OpenMP All\_Pair\_Distance Implementation**

* 1. ***Code Overview***

The following protoypes were rewritten using OpenMP:

1. block\_all\_pairs (block work distribution)
2. block\_ cyclic\_all\_pairs (block cyclic work distribution)
3. dynamic\_all\_pairs (dynamic work distribution)

The three functions were re-written to utilize OpenMP to compute the pair-wise distance matrix of MNIST train images. All algorithms were tested using 12 threads and a chunk size of 2 (if the algorithm used a chunk size). A couple of modifications were made to the signatures of these functions for ease of use.

First, the following function type was declared:

typedef void (\*AllPairsWorker\_t)(

    std::vector<float> &,

    std::vector<float> &,

    uint64\_t,

    uint64\_t,

    uint64\_t);

This allows for simplistic printing and testing of functions, as the new OpenMP functions share the same type as the C++ ones. The the following helper:

void printAndTimePairs(

    std::string name,

    AllPairsWorker\_t Worker,

    AllPairWorkerData\_t \*data)

{

    std::cout << name << "...\n";

    StartTimer();

    Worker(data->mnist, data->allPairs, data->rows, data->threads, data->chunksize);

    std::cout << "\t " << name << " time = " << StopTimer() << "\n";

    std::cout << "\tall\_pair[1000] = " << data->allPairs[1000] << "\n\n";

}

The data is in the following struct, which gathers all relevant information:

typedef struct

{

    std::vector<float> mnist;

    std::vector<float> allPairs;

    uint64\_t rows;

    uint64\_t threads;

    uint64\_t chunksize;

} AllPairWorkerData\_t;

The following code defined the all\_pairs function, so that it could easily be called in a loop to test all the various sizes in a single run.

int all\_pairs(uint64\_t nRows = 60000)

{

    std::cout << "Load MNIST train-image dataset ......\n";

    StartTimer();

    std::vector<float> mnist(ROWS \* COLS, 5); // values initialized to 5

    load\_binary(mnist.data(), ROWS \* COLS,

                "./data/train-images.bin");

    StopTimer();

    // validate data

    if ((int)(mnist[156] \* 10000) != 4941) // the value should be 0.494118

        return 0;

    if (nRows < 1 || nRows > ROWS)

        return -1;

    std::vector<float> all\_pair(nRows \* nRows);

    AllPairWorkerData\_t data;

    data.allPairs = all\_pair;

    data.mnist = mnist;

    data.rows = nRows;

    data.threads = 12;

    data.chunksize = 2;

    std::cout << "\n\nCompute pair\_wise\_distance for first " << nRows << " MNIST train images (gcc) using " << data.threads << " threads \n\n";

    printAndTimePairs("block\_all\_pairs", block\_all\_pairs, &data);

    printAndTimePairs("block\_cyclic\_all\_pairs", block\_cyclic\_all\_pairs, &data);

    printAndTimePairs("dynamic\_all\_pairs", dynamic\_all\_pairs, &data);

    printAndTimePairs("OpenMP\_block\_all\_pairs", OpenMP\_block\_all\_pairs, &data);

    printAndTimePairs("OpenMP\_block\_cyclic\_all\_pairs", OpenMP\_block\_cyclic\_all\_pairs, &data);

    printAndTimePairs("OpenMP\_dynamic\_all\_pairs", OpenMP\_dynamic\_all\_pairs, &data);

    return 0;

}

Lastly, all three implementations shared the same following code to run the function. The difference between the function was in the OpenMP configuration that was performed.

void OpenMP\_CalculatePairWiseDistance(

    std::vector<float> &mnist,

    std::vector<float> &all\_pair,

    uint64\_t rows,

    uint64\_t unused = 0)

{

#pragma omp parallel for schedule(runtime)

    for (uint64\_t i = 0; i < rows; i++)

    {

        for (uint64\_t I = 0; I <= i; I++)

        {

            float accum = float(0);

            for (uint64\_t j = 0; j < COLS; j++)

            {

                float residue = mnist[i \* COLS + j] - mnist[I \* COLS + j];

                accum += residue \* residue;

            }

            all\_pair[i \* rows + I] = all\_pair[I \* rows + i] = accum;

        }

    }

}

In this block of code, the OpenMP preprocessor command *schedule* performs an optimization based on which OpenMP schedule type was selected. The configuration for this is called with omp\_set\_schedule[1] in each respective implementation. The loop code is the same as the sequential all\_pair implementation; the OpenMP library handles all the parallelization and threads.

* 1. **OpenMP*\_block\_all\_pairs***

**Timing Results**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Matrix Size | 400 | 800 | 10,000 | 20,000 | 30,000 | 60,000 |
| C++ Block | 0.0062947 | 0.0207421 | 4.75428 | 24.9631 | 76.2645 | 351.505 |
| OpenMP block | 0.0023593 | 0.0034301 | 2.59424 | 18.5267 | 56.6752 | 326.576 |

**Implementation**

// block work distribution

void OpenMP\_block\_all\_pairs(

    std::vector<float> &mnist,

    std::vector<float> &all\_pair,

    uint64\_t rows,

    uint64\_t num\_threads = 64,

    uint64\_t unused = 0)

{

    uint64\_t chunkSize = rows / num\_threads;

    omp\_set\_dynamic(0);

    omp\_set\_num\_threads(num\_threads);

    omp\_set\_schedule(omp\_sched\_static, chunkSize);

    OpenMP\_CalculatePairWiseDistance(mnist, all\_pair, rows);

}

}

Here we set the dynamic threading of OpenMP to 0 as we want to run each test with 12 threads (This is the same across all three implementations). Second, the number of threads is set. Third, the schedule is chosen as static and chunk size set to the rows / threads which is analogous to the original block\_all\_pairs implementation. Lastly, the helper function is called which computes the distance using a block work distribution.

* 1. **OpenMP*\_block\_cyclic\_all\_pairs***

**Timing Results**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Matrix Size | 400 | 800 | 10,000 | 20,000 | 30,000 | 60,000 |
| C++ block-cyclic | 0.0040413 | 0.0178878 | 1.86041 | 8.58314 | 24.1807 | 153.532 |
| OpenMP block-  cyclic | 0.0009989 | 0.0026154 | 1.19309 | 5.77042 | 13.2321 | 162.987 |

**Implementation**

// block cyclic work distribution

void OpenMP\_block\_cyclic\_all\_pairs(

    std::vector<float> &mnist,

    std::vector<float> &all\_pair,

    uint64\_t rows,

    uint64\_t num\_threads = 64,

    uint64\_t unused = 0)

{

    uint64\_t chunkSize = 2;

    omp\_set\_dynamic(0);

    omp\_set\_num\_threads(num\_threads);

    omp\_set\_schedule(omp\_sched\_static, chunkSize);

    OpenMP\_CalculatePairWiseDistance(mnist, all\_pair, rows);

}

First, the number of threads is set. Second, the schedule is chosen as static and chunk size set to the 2 which is analogous to the original block\_cyclic\_all\_pairs implementation. Lastly, the helper function is called which computes the distance using a block cyclic work distribution.

* 1. **OpenMP*\_dynamic\_all\_pairs***

**Timing Results**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Matrix Size | 400 | 800 | 10,000 | 20,000 | 30,000 | 60,000 |
| C++ dynamic | 0.0038365 | 0.0123817 | 2.77072 | 9.13156 | 21.7323 | 124.304 |
| OpenMP dynamic | 0.0007513 | 0.0024572 | 0.67911 | 5.24815 | 11.8631 | 143.983 |

**Implementation**

void OpenMP\_dynamic\_all\_pairs(

    std::vector<float> &mnist,

    std::vector<float> &all\_pair,

    uint64\_t rows,

    uint64\_t num\_threads = 64,

    uint64\_t chunk\_size = 64 / sizeof(float))

{

    uint64\_t chunkSize = 2;

    omp\_set\_dynamic(0);

    omp\_set\_num\_threads(num\_threads);

    omp\_set\_schedule(omp\_sched\_dynamic, chunkSize);

    OpenMP\_CalculatePairWiseDistance(mnist, all\_pair, rows);

}

First, the number of threads is set. Second, the schedule is chosen as dynamic and chunk size set to the 2 which is analogous to the original dynamic\_all\_pairs implementation. Lastly, the helper function is called which computes the distance using a dynamic work distribution.

* 1. **Conclusion**

The OpenMP library provides optimization that outperforms manual C++ implementation of the same work distribution schemes. Though the improvement is marginal for large N, smaller work cases see significant performance hikes.

**Section 2: Celebrity Face Analysis using PyTorch and CUDA**

* 1. ***Code Overview***

**This project used the celebA[3] dataset, stored locally in the folder img\_align\_celeba. The image data from these files was read in to a list called files. This list is used to read the image data into a PyTorch tensor format. Then, following the process to compute eigen faces, each step of the process is computed and timed using both CPU and GPU methods. Timing with very low values ( <1ms), due to the inaccuracy of %%time, may not be the most reliable.**

* 1. ***Reading in Tensor Data***

**The following code reads the data in which returns an array of N x M x C where N is the height, M is the width and C is the color channels. In the celebA dataset, N=218, M=178, and C=3. This image data is converted to grayscale. The composite data of all images is reshaped into a tensor of size B x Q, where B = total number of images being processed, and Q = N x M x C. %%Time was used over %%Timeit because only comparative analysis is being performed and the runtimes are quite long.**

**Code:**

%%time

import torch

import imageio.v3 as iio

import numpy as np

total\_number\_of\_faces = 100

dimx, dimy = int(218), int(178)

def rgb2gray(image):

    img\_gray = np.zeros((218, 178, 3), dtype = np.uint8)

    return np.stack([np.dot(image, [.333,.333,.333])]\*3, axis=-1)

tensor\_data = torch.zeros(total\_number\_of\_faces, dimx\*dimy\*3, dtype=torch.uint8, device=device)

for index, filename in enumerate(files):

    if index == total\_number\_of\_faces:

        break

    tensor\_data[index] = torch.tensor(((rgb2gray(iio.imread(dirname+"/"+filename)).flatten().ravel())),  device=device)

**Timing:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tensor Device** | **100** | **200** | **500** | **1000** | **5000** |
| **CPU** | **570 ms** | **807 ms** | **2.58 s** | **3.59 s** | **27.3 s** |
| **CUDA** | **218 ms** | **532 ms** | **935 ms** | **1.79 s** | **7.74 s** |

* 1. ***Preprocessing with Principal Component Analysis***

This step helps center the data and reduce dimensionality. It also is the most costly operation both in time and in size. The data must be in float32 format, which makes the tensor 4 times larger than it was as a ubyte. **The GPU for this project is limited to 10 GB of VRAM, so 1,000 is the largest number of image worked with at a time as any more exceeds its memory capabilities**

**Code:**

%%time

U,S,V = torch.pca\_lowrank(tensor\_data.double(), q=total\_number\_of\_faces, center=True, niter=2)

prin\_comp = 500

preprocessed\_data\_temp = torch.matmul(tensor\_data.double().T, V.T[:, :prin\_comp])

preprocessed\_data\_temp.shape

**Timing(prin\_comp=500):**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tensor Device** | **100** | **200** | **500** | **1000** | **5000** |
| **CPU** | **2.47 s** | **4.95 s** | **17.3 s** | **51.2 s** | **11m 28s** |
| **CUDA** | **334 ms** | **1.68 s** | **2.9 s** | **9.93 s** | **N/A** |

***Examples with k=1000, original on left***

***A picture containing application

Description automatically generated***

***Examples with k=500, original on left***

***Graphical user interface, application

Description automatically generated with medium confidence***

* 1. ***Computing the Mean Face***

Interestingly there is a difference in growth factor. The CPU-based algorithm grows as N does, looking from the limited data to be O(N). The CUDA-based algorithm runs on some much smaller linear factor of N, only gaining a 5 ms of processing time from an increase of 4900 to N.

**Code:**

%%time

import pandas as pd

face\_loader = torch.utils.data.DataLoader(tensor\_data, batch\_size=10000, shuffle=False)

mean\_face = torch.zeros(celeb\_height\*celeb\_width\*celeb\_depth, dtype=torch.float32, device=device)

for faces in face\_loader:

        faces = faces.type(torch.float32)

        mean\_face += (faces.sum(0)) / (1.0 \* total\_number\_of\_faces)

if tensor\_data.is\_cuda:

        plt.imshow((mean\_face.type(torch.uint8)).cpu().reshape(218, 178, 3))

else:

        plt.imshow((mean\_face.type(torch.uint8)).reshape(218, 178, 3))

**Timing:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tensor Device** | **100** | **200** | **500** | **1000** | **5000** |
| **CPU** | **42.7 ms** | **52.1 ms** | **114 ms** | **230 ms** | **1.26 s** |
| **CUDA** | **11.4 ms** | **12.3 ms** | **14.9 ms** | **15.2 ms** | **16.3 ms** |

* 1. ***Subtracting the mean face***

**Code:**

%%time

mean\_face\_removed = torch.zeros(total\_number\_of\_faces, dimx\*dimy\*3, dtype=torch.uint8, device=device)

for index, filename in enumerate(files):

    if index == total\_number\_of\_faces:

        break

    mean\_face\_removed[index] = torch.subtract(preprocessed\_data[index], mean\_face, alpha=1)

plt.imshow((mean\_face\_removed[0].cpu().type(torch.uint8)).reshape(218, 178, 3))

mean\_face\_removed.shape

**Timing:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tensor Device** | **100** | **200** | **500** | **1000** | **5000** |
| **CPU** | **864 ns** | **65 ms** | **143 ms** | **261 ms** | **572 ms** |
| **CUDA** | **10.4 ms** | **91.3 ms** | **99.7 ms** | **128 ms** | **102.2 ms** |

***Example with k=500***

***A picture containing calendar

Description automatically generated***

* 1. ***Computing Covariance Matrix***

This is a simple one-liner thanks to PyTorch. This value is actually not used as there is a faster covariance that can be used instead. However, this was too simple not to include, so here it is.

**Code:**

cov\_data = torch.cov(mean\_face\_removed, correction=1)

**Timing:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tensor Device** | **100** | **200** | **500** | **1000** | **5000** |
| **CPU** | **74.4 ms** | **146 ms** | **444 ms** | **1.86 s** | **18.2 s** |
| **CUDA** | **71.4 ms** | **140 ms** | **542 ms** | **1.22 s** | **5.43 s** |

* 1. ***Compute Eigen Vectors***

**Code:**

%%time

# automatically normalizes to 1

small\_cov = torch.matmul(mean\_face\_removed.double(), mean\_face\_removed.double().T)

print("shape:",small\_cov.shape)

eigen\_val, eigen\_vec = torch.linalg.eig(small\_cov)

eigen\_vec

**Timing:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tensor Device** | **100** | **200** | **500** | **1000** | **5000** |
| **CPU** | **73.9 ms** | **164 ms** | **646 ms** | **2 s** | **32.8 s** |
| **CUDA** | **14.4 ms** | **52. 6 ms** | **317 ms** | **1.14 s** | **15.7 s** |

* 1. ***Compute K-Best vectors***

Considering this algorithm is selecting the top k best vectors from an array, there is little surprise that minimal difference is present in computation types. No PyTorch methods were used here to leverage the CUDA cores, so the algorithm is essentially the same in both cases.

**Code:**

%%time

#number of k -largest to keep

best\_k\_vectors = 4

import numpy as np

np\_eigen\_vals = np.array(eigen\_val.cpu())

np\_eigen\_vals\_abs = np.zeros(len(np\_eigen\_vals))

#find the largest magnitude of the eigen values for each vector

for i in range(len(np\_eigen\_vals)):

    np\_eigen\_vals\_abs[i] = np.absolute(np\_eigen\_vals[i])

#find the top k indicies of each eigen vector with maximal value

indicies = np.argpartition(np\_eigen\_vals\_abs, -best\_k\_vectors)[-best\_k\_vectors:]

best\_vectors = [eigen\_vec[index].double() for index in indicies]

# #convert A^TA vectors to A^TA vectors

for j in range(len(best\_vectors)):

    best\_vectors[j] = torch.matmul(mean\_face\_removed.double().T, best\_vectors[j])

eigen\_sum = np.array([np\_eigen\_vals\_abs[index] for index in indicies]).sum()

**Timing(k=4):**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tensor Device** | **100** | **200** | **500** | **1000** | **5000** |
| **CPU** | **129 ms** | **259 ms** | **642 ms** | **1.43 s** | **7.43 s** |
| **CUDA** | **137 ms** | **264 ms** | **646 ms** | **1.21 s** | **6.2 s** |

* 1. ***Compute Eigen Faces***

**Code:**

%%time

eigen\_faces\_celebA = torch.zeros(total\_number\_of\_faces, best\_k\_vectors, dimx\*dimy\*3, dtype=torch.float32, device=device)

for index, filename in enumerate(files):

    if index == total\_number\_of\_faces:

        break

    eigen\_faces\_index = torch.zeros(best\_k\_vectors, dimx\*dimy\*3, dtype=torch.float32, device=device)

    for i in range(0, best\_k\_vectors-1):

        eigen\_faces\_index[i] = torch.mul(mean\_face\_removed[i], best\_vectors[i+1])\*(np\_eigen\_vals\_abs[i+1]/eigen\_sum)

    eigen\_faces\_celebA[index] = eigen\_faces\_index

print(eigen\_faces\_celebA.shape)

**Timing:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tensor Device** | **100** | **200** | **500** | **1000** | **5000** |
| **CPU** | **173 ms** | **329 ms** | **825 ms** | **19.1 s** | **1 m 12 s** |
| **CUDA** | **153 ms** | **48.6 ms** | **120 ms** | **1.82 s** | **18.4 s** |

* 1. ***Results***

The best eigen faces were calculated for the images by varying the number of best vectors and the number of principal components used in the preprocessing step. An increase in the principal component amount resulted in a better overall result. However, this is a trade off as it increased the dimensionality and computation time significantly and more than any other variable. Therefore, a balance had to be struck between computation time an accuracy. Another aspect to note is that some information may have been lost in the varies casting from ubyte->float->imaginary->ubyte for the tensors. Each result shows a single face from the dataset along with its eigen faces. Here are the results:

Final Result 1. (size of data 1000, principal\_comp = 500, k\_vectors = 3)

Chart

Description automatically generated

In the above image, the composite face is not an excellent representation of the first image; eig\_face1 actually captures a more unique and clear version. This is in part due to the variety of background in the images. Though gray scaling helps, the information in the backgrounds is kept as unique due to the variety despite not actually being valuable. An additional preprocessing step to remove backgrounds could eliminate this. Another consideration would be to drastically increase the number of eigen faces.

Final Result 2. (size of data 1000, principal\_comp = 250, k\_vectors = 3)

Chart

Description automatically generated

In this image, the only difference is a halving of principal components. This leaves less overall data and there is not enough information to reconstruct the original image.

Final Result 3 (size of data 100, principal\_comp = 5000, k\_vectors = 6)

A picture containing graphical user interface

Description automatically generated

This result is interesting as it uses a smaller set of data with higher component count and eigen faces. However, most of the resulting eigen faces appear to carry minimal data, but the summation of all of them is closer to the mean.

Final Result 4 (size of data 1000, principal\_comp = 10000, k\_vectors = 6)

Chart

Description automatically generated

Final Result 5 (size of data 5000, principal\_comp = 500, k\_vectors = 3)

Chart

Description automatically generated

Overall, these results are imperfect. There could be some refinement and accuracy increase to further perfect results. The handling of tensor data types in particular, which were limited by GPU memory constraints and time, resulted in some lost information that produces a somewhat flawed final result. Other issues could stem from an imperfect understanding of the formulas applied.

The eigen faces could be used to train a neural network to perform face recognition as detailed in the case study[2]. The results from this project would likely produce a somewhat flawed neural network, but the performance of the data in that sense is outside the scope of this project. There are libraries that perform many of these steps behind the scenes in one go, and would allow a jump straight to neural network training.[4]

In terms of computation power though, the GPU reigns supreme. CUDA core processing is much quicker than CPU processing in nearly every step of the computation process. PyTorch allows for simpler manipulation of data as well. Many complex matrix operations are reduced to a simple PyTorch library call. Tensors combined with CUDA cores process much faster than NumPy arrays. Especially in image processing, the limiting factor seems to be memory on a GPU rather than time. However, this could be a quirk of inefficient memory management unique to this project or simply a limitation of the hardware available to this project. Still, when possible, the GPU can parallelize matrix operations CUDA core with greater efficiency than the CPU can in RAM.

# **Contributions**

History of the project and full set of code for class can be found here:

https://github.com/cmklen/cse625-parallel-programming/tree/main/FinalProject

# **References**

[1] OPENMP API Specification: Version 5.0 November 2018

https://www.openmp.org/

[2] <http://www.vision.jhu.edu/teaching/vision08/Handouts/case_study_pca1.pdf>

[3] <https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>

[4] <https://machinelearningmastery.com/face-recognition-using-principal-component-analysis/>

[5] <https://pytorch.org/docs/stable/index.html>