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

1.1- 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.

1.2- `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.

1.3- `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.

1.4- 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.

1.5- 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

2.1- 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.

```
import torch
print(torch.cuda.is_available())
cpu = torch.device("cpu")
cuda = torch.device("cuda:0")

torch.cuda.empty_cache()
device = cuda #cpu
```

This code was changed to test CPU and GPU processing for each data set size. The devices is the location where all Tensors are created and processed.

2.2- 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

2.3- 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

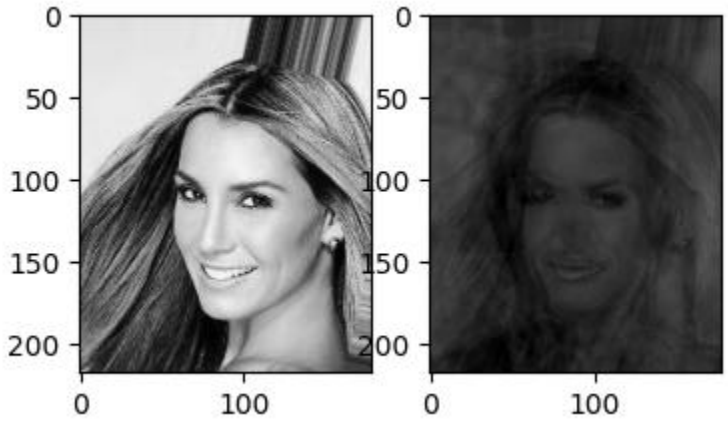
Code:

```
%%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
```

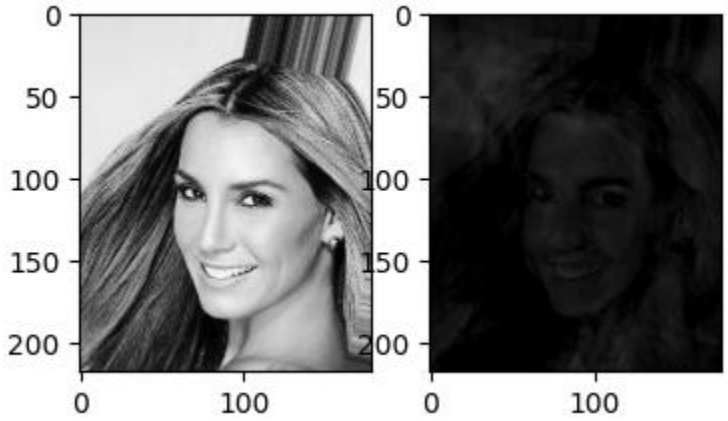
Timing(prin_comp=500):

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



Examples with k=500, original on left



2.4- 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

2.5- Subtracting the mean face

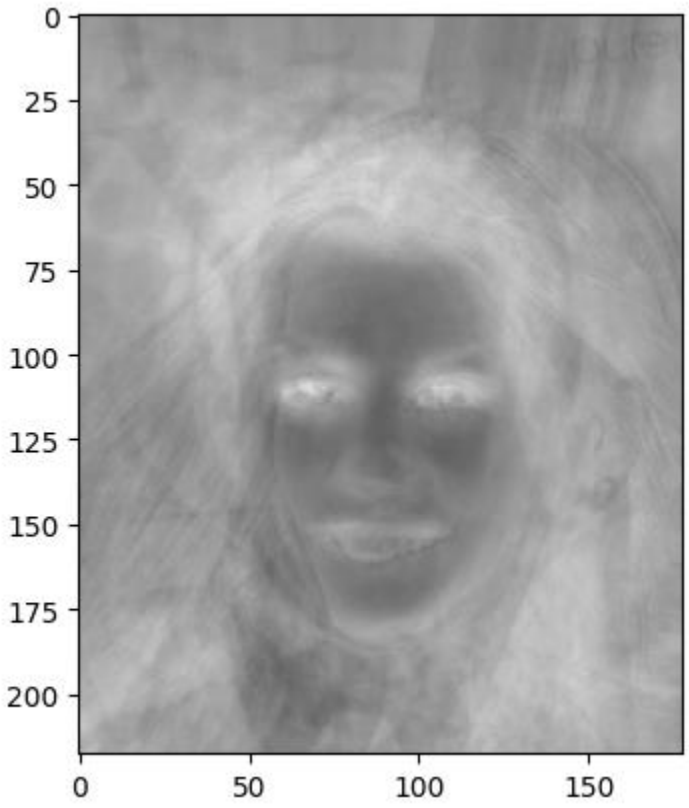
Code:

```
%%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



2.6- 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

2.7- 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

2.8- 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

2.9- Compute Eigen Faces

Each eigen face is the result of multiplying the mean subtracted face times the best eigen vectors. The number of eigen faces is equal to the number of chosen eigen vectors. Each face represent some core component of information about the original image. The eigen face is then multiplied by a constant; the value of this constant is the eigen value associated with that face divided by the sum total of all the eigen values. This essentially gives the weight of the eigen face towards the composite face it is used to build.

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)
```

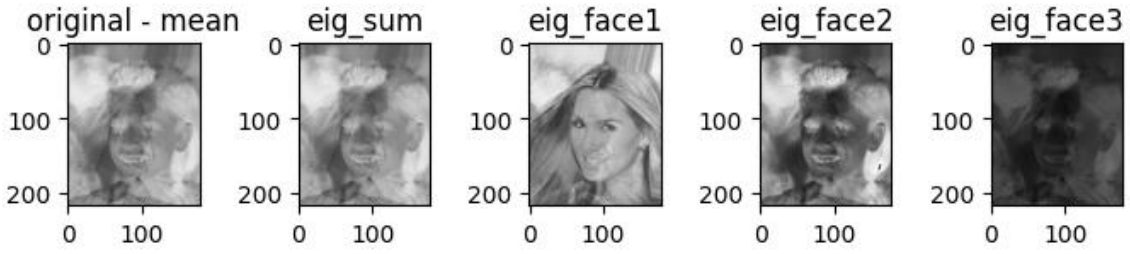
Timing:

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**

2.10- Results

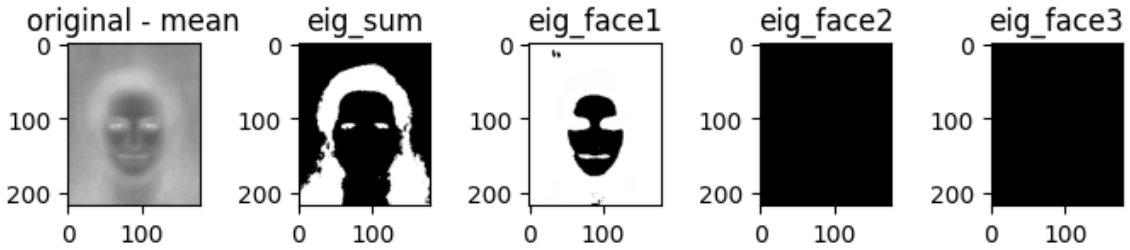
The best eigen faces were calculated for the images by varying the number of best vectors and the number of principal components used in the preprocessing step. An increase in the principal component amount resulted in a better overall result. However, this is a trade off as it increased the dimensionality 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)



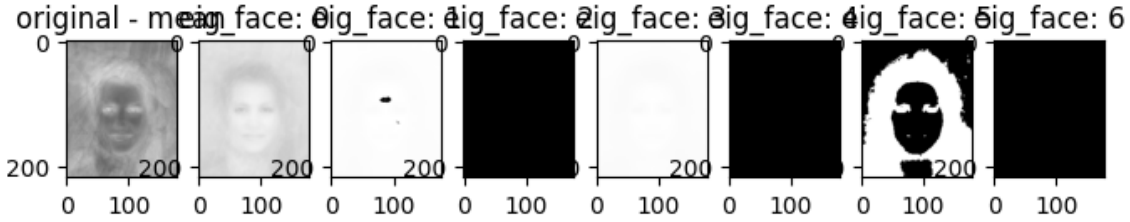
In the above image, the composite face is not an excellent representation of the first image; eig_face1 actually captures a more unique and clear version. This is in part due to the variety of background in the images. Though gray scaling helps, the information in the backgrounds is kept as unique due 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)



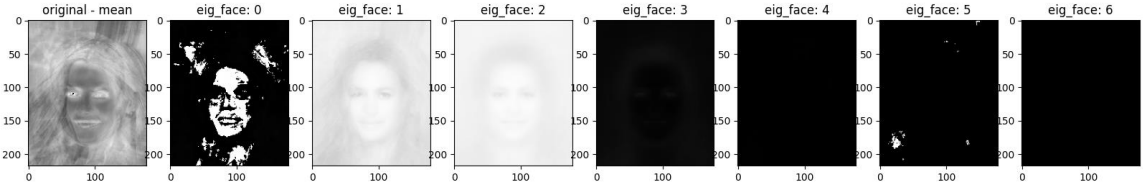
In this image, the only difference is a halving of principal components. This leaves less overall data and there is not enough information to reconstruct the original image.

Final Result 3 (size of data 100, principal_comp = 5000, k_vectors = 6)

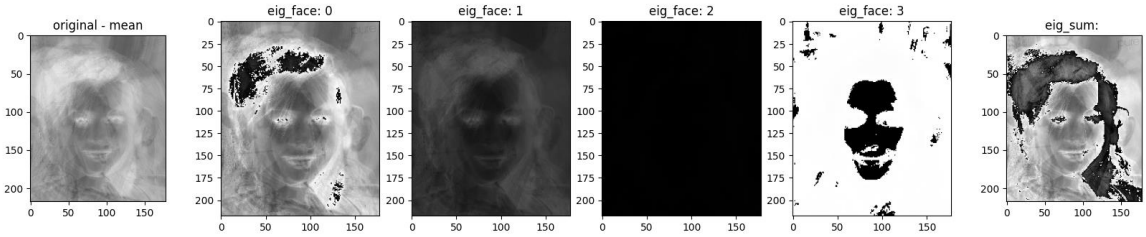


This result is interesting as it uses a smaller set of data with higher component count and eigen faces. However, most of the resulting eigen faces appear to carry minimal data, but the summation of all of them is closer to the mean.

Final Result 4 (size of data 1000, principal_comp = 10000, k_vectors = 6)



Final Result 5 (size of data 5000, principal_comp = 500, k_vectors = 3)



Overall, these results are imperfect. There could be some refinement and modifications to increase accuracy for better results. The handling of tensor data types 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 completely black eigen faces could never be fully eliminated; even though they have low eigen values and do not effect the sum very much, they should not be generated if the process is done 100% correct.

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