Korea Advanced Institute of Science and Technology

School of Electrical Engineering

EE817 GPU Programming and Its Applications Spring 2018

Team: 9

Student’s Name: Dinh Vu

Student’s ID: 20184187

**Term Project Final Report**

**Face Recognition using Principal Component Analysis**

1. **Problem**

Facial recognition is a technology capable of identifying a person from digital images. Principal Components Analysis (PCA) is one of the facial recognition techniques.

For the sake of convenience, the public data base Japanese Females Facial Expression (JFFE), including 213 gray images, is used and divided to training set, consisting of 163 images and test set, holding 50 images. The size of each image in JFFE is 64×64.

The implementation program, based on the training set, must recognize the every person in the test set by using PCA.

1. **Principal Component Analysis**

Because the size of each image is 64×64 and all images are grayscale, each image can be expressed to a row vector 1×4096. Therefore, the form of the training and test matrix respectively, is:

There are 7 steps of PCA. Each step will be explained as follow.

* 1. ***Find mean vector***

The mean vector is calculated by the below formula:

In this project, is the number of training images.

* 1. ***Subtract mean vector***

Each image vector subtracts to the mean vector.

* 1. ***Compute covariance matrix***
  2. ***Find eigenvectors and eigenvalues***

Because the size of the covariance matrix is very large, 4096×4096, it is impossible to calculate eigenvalues and eigenvector by the normal algebra method. However, Power Method, an iteration algorithm, can approximately determine eigenvalues and eigenvectors of the covariance matrix. The Power Method is presented below.

* + 1. Initialize vector subject to
    2. For , calculate
    3. Normalize:
    4. If is small enough, the process stop. If not, return to 2.4.2
    5. is the eigenvector corresponding with the highest eigenvalue

In order to find the next eigenvector corresponding to the second largest eigenvalue the matrix is computed as:

Then applying Power Method for matrix . We repeat the above procedure until k eigenvectors is found.

* 1. ***Project the training and test matrix in the new subspace***

After 2.4, the projection matrix is concatenated by k eigenvectors.

where and

Projection the training and test data in the new subspace created by k eigenvectors, is simply matrix multiplication with .

1. **C Version**

The training matrix and test matrix is stored in 2 text files train\_matrix.txt and test\_matrix.txt. The training matrix and test matrix will be read from these files to CPU.

* 1. ***Find mean vector***

In C programming, matrix operation always requires huge loop. All elements in each column of the training matrix are sum up and take the average to compute the correlated element in the mean vector. Due to store linear data by row-major, the memory address j × dimension + i must be calculated.

for (int i = 0; i < dimension; i++) {

tmp = 0;

for (int j = 0; j < Ntrain; j++) {

tmp = tmp + trainMat[j\*dimension + i];

}

mean[i] = tmp / Ntrain;

}

* 1. ***Subtract mean vector***

As mention before two for-loops are requirement, and the global memory address DestIndex = i × dimension + j is computed. Each row of the training matrix minus to the mean vector in order to move data to be near the center.

float \*X;

int Xbuffer = Ntrain \* dimension \* sizeof(float);

X = (float\*)malloc(Xbuffer);

for (int i = 0; i < Ntrain; i++) {

for (int j = 0; j < dimension; j++) {

DestIndex = i \* dimension + j;

X[DestIndex] = trainMat[DestIndex] - mean[j];

}

}

* 1. ***Compute covariance matrix***

The covariance matrix is compute by two function cpuMatTranpose() and cpuMatMul() as the mathematical background. In programming, it is not necessary to divide to Ntrain – 1 because matrix and matrix have the same eigenvalues and eigenvectors, for .

float \*covMat;

float \*X1;

int covMatBuffer = dimension \* dimension \* sizeof(float);

covMat = (float\*)malloc(covMatBuffer);

X1 = (float\*)malloc(Xbuffer);

cpuMatTranspose(X, Ntrain, dimension, X1);

cpuMatMul(X1, X, dimension, Ntrain, dimension, covMat);

* 1. ***Determine eigenvectors and eigenvalues***

The Power Method implemented by C language follows as section 2.4. A do-while loop is used with threshold esp = 0.00001.

void PowerMethod(float \*A, int n, float eps, float \*eigVec, float \*lambda) {

float \*Q;

float \*prevQ;

float \*Z;

float \*StepVec;

float \*QtA;

float norm2z = 0;

float dist = 1;

int Buffer = n \* sizeof(float);

Q = (float\*)malloc(Buffer);

prevQ = (float\*)malloc(Buffer);

Z = (float\*)malloc(Buffer);

StepVec = (float\*)malloc(Buffer);

QtA = (float\*)malloc(Buffer);

Q[0] = 1;

for (int i = 1; i < n; i++) Q[i] = 0;

do {

for (int i = 0; i < n; i++) prevQ[i] = Q[i];

cpuMatMul(A, Q, n, n, 1, Z);

norm2z = cpuNorm2(Z, n, 1);

for (int i = 0; i < n; i++) Q[i] = Z[i] / norm2z;

cpuMatSub(Q, prevQ, n, 1, StepVec);

dist = cpuNorm2(StepVec, n, 1);

} while (dist > eps);

for (int i = 0; i < n; i++) eigVec[i] = Q[i];

cpuMatMul(Q, A, 1, n, n, QtA);

cpuMatMul(QtA, Q, 1, n, 1, lambda);

free(Q);

free(prevQ);

free(Z);

free(StepVec);

free(QtA);

}

In order to calculate the next eigenvector, after each for-loop, the eigenvalues is load into the array eigVal[] and the eigenvector is stored in the array k\_eig\_vec[].

for (int i = 0; i < k; i++) {

PowerMethod(B, dimension, 0.00001, eigVec, lambda);

for (int p = 0; p < dimension; p++) {

DestIndex = p \* k + i;

k\_eig\_vec[DestIndex] = eigVec[p];

}

eigVal[i] = lambda[0];

cpuMatMul(eigVec, eigVec, dimension, 1, dimension, VVt);

cpuMatMulScalar(VVt, dimension, dimension, eigVal[i], VVt);

cpuMatSub(B, VVt, dimension, dimension, B);

}

* 1. ***Project the training and test data***

It is straightforward to project the training data, only function cpuMatMul() is called.

cpuMatMul(X, k\_eig\_vec, Ntrain, dimension, k, project\_train\_img);

However, the test data requires to move to near the center that means subtracting to the mean vector. Therefore, function testPCA() is built to get the projection of the test images.

void testPCA(int \*testMat, float \*k\_eig\_vec, float \*mean, int Ntest, int imgSize, int k, float \*project\_test\_img) {

int DestIndex;

int dimension = imgSize \* imgSize;

float \*X;

int Xbuffer = Ntest \* dimension \* sizeof(float);

X = (float\*)malloc(Xbuffer);

for (int i = 0; i < Ntest; i++) {

for (int j = 0; j < dimension; j++) {

DestIndex = i \* dimension + j;

X[DestIndex] = testMat[DestIndex] - mean[j];

}

}

cpuMatMul(X, k\_eig\_vec, Ntest, dimension, k, project\_test\_img);

free(X);

}

* 1. ***Identify the test images***

After we get the projection of the training images and the test images, the Euclide distance between projection of each test image and the projection of the training images are computed. Then, the training image with the minimum distance is chosen.

distance = norm2(project\_train\_img[j] - project\_test\_img[i]);

min{d1, d2, d3, …, d163}



**Figure 3.1.** Distance between the projection of test and train

void identify(float \*project\_train\_img, float \*project\_test\_img, int Ntrain, int Ntest, int dimension, int \*recognized\_img) {

int DestIndex;

float \*test;

float \*train;

float \*D;

float distance = 0;

float min;

int buffer = dimension \* sizeof(float);

test = (float\*)malloc(buffer);

train = (float\*)malloc(buffer);

D = (float\*)malloc(buffer);

for (int i = 0; i < Ntest; i++) {

min = 100000;

for (int j = 0; j < dimension; j++) {

DestIndex = i \* dimension + j;

test[j] = project\_test\_img[DestIndex];

}

for (int m = 0; m < Ntrain; m++) {

for (int n = 0; n < dimension; n++) {

DestIndex = m \* dimension + n;

train[n] = project\_train\_img[DestIndex];

}

cpuMatSub(train, test, 1, dimension, D);

distance = cpuNorm2(D, 1, dimension);

if (distance < min) {

min = distance;

recognized\_img[i] = m + 1;

}

}

}

free(test);

free(train);

free(D);

}

1. **CUDA C Version**

The training matrix and test matrix is stored in 2 text files train\_matrix.txt and test\_matrix.txt. The data of training matrix and test matrix is read from these files to CPU, then, copy them to GPU by using API function cudaMemcpy().

* 1. ***Find mean vector***

Each thread calculated the corresponding element of the mean vector. It means that each thread read data of each column of the training matrix then take average.



**Figure 4.1.** Threads assign to calculate mean vector

\_\_global\_\_ void gpuMeanVec(int \*trainMat, int Ntrain, int dimension, float \*mean) {

int tid = threadIdx.x + blockIdx.x \* blockDim.x;

float tmp = 0;

int DestIndex;

for (int i = 0; i < Ntrain; i++) {

DestIndex = i \* dimension + tid;

tmp = tmp + trainMat[DestIndex];

}

mean[tid] = tmp / Ntrain;

}

There are totally 4096 threads, so the configuration can be 32 thread blocks and 128 threads each block

gpuMeanVec<<<32, 128>>>(d\_trainMat, Ntrain, dimension, d\_mean);

* 1. ***Subtract mean vector***

Each thread read data of an element of training matrix and minus to the correlated element of the mean vector. Hence, there are 4096×163 threads.



**Figure 4.2.** Subtract mean vector

\_\_global\_\_ void gpuSubMean(int \*trainMat, float \*mean, int Ntrain, int dimension, float \*X) {

int tid, tx, ty;

tx = threadIdx.x + blockIdx.x \* blockDim.x;

ty = threadIdx.y + blockIdx.y \* blockDim.y;

if ((tx < dimension) && (ty < Ntrain)) {

tid = dimension \* ty + tx;

X[tid] = trainMat[tid] - mean[tx];

}

}

If block 128×1 is chosen, grid will be 32×163 in order to take full of threads

dim3 block1(128, 1);

dim3 grid1(dimension/128, Ntrain);

float \*d\_X;

int Xbuffer = Ntrain \* dimension \* sizeof(float);

cudaMalloc((void\*\*)&d\_X, Xbuffer);

gpuSubMean<<<grid1, block1>>>(d\_trainMat, d\_mean, Ntrain, dimension, d\_X);

* 1. ***Compute covariance matrix***

int covMatBuffer = dimension \* dimension \* sizeof(float);

float \*d\_Xt, \*d\_covMat;

cudaMalloc((void\*\*)&d\_Xt, Xbuffer);

cudaMalloc((void\*\*)&d\_covMat, covMatBuffer);

gpuMatTranspose<<<grid1, block1>>>(d\_X, d\_Xt, Ntrain, dimension);

dim3 block2(32, 16);

dim3 grid2(dimension/32, dimension/16);

gpuMatMul<<<grid2, block2>>>(d\_Xt, d\_X, d\_covMat, dimension, Ntrain, dimension)

* 1. ***Determine eigenvectors and eigenvalues***

Unlike C program, CUDA C program cannot use do-while loop for iteration algorithm. Fortunately, the Power Method convergence very fast, so instead using do-while, the 1000 for-loop is used.



**Figure 4.3.** Finding k eigenvalues and k eigenvectors using Power Method

float prev\_lambda = 0;

const float eps = 0.000001;

float \*Q, \*normZ, \*k\_eig\_vec;

float \*d\_Q, \*d\_Z, \*d\_W, \*d\_normZ;

int normBuffer = sizeof(float);

int eigVecBuffer = k \* imgBuffer;

Q = (float\*)malloc(imgBuffer);

normZ = (float\*)malloc(normBuffer);

k\_eig\_vec = (float\*)malloc(eigVecBuffer);

cudaMalloc((void\*\*)&d\_Q, imgBuffer);

cudaMalloc((void\*\*)&d\_Z, imgBuffer);

cudaMalloc((void\*\*)&d\_W, covMatBuffer);

cudaMalloc((void\*\*)&d\_normZ, normBuffer);

int block3 = 32;

int grid3 = dimension / block3;

int sharedMemSize = block3 \* sizeof(float);

for (int i = 0; i < k; i++) {

Q[0] = 1;

for (int m = 1; m < dimension; m++) Q[m] = 0;

cudaMemcpy(d\_Q, Q, imgBuffer, cudaMemcpyHostToDevice);

gpuMatMul<<<16, 256>>>(d\_covMat, d\_Q, d\_Z, dimension, dimension, 1);

// Power Method iteration

for (int j = 0; j < 1000; j++) {

normZ[0] = 0;

cudaMemcpy(d\_normZ, normZ, normBuffer, cudaMemcpyHostToDevice);

gpuNorm2<<<grid3, block3, sharedMemSize>>>(d\_Z, d\_normZ, dimension);

cudaThreadSynchronize();

cudaMemcpy(normZ, d\_normZ, normBuffer, cudaMemcpyDeviceToHost);

normZ[0] = sqrt(normZ[0]);

cudaMemcpy(d\_normZ, normZ, normBuffer, cudaMemcpyHostToDevice);

gpuNormalize<<<grid3, block3, sharedMemSize>>>(d\_Z, d\_normZ, d\_Q, dimension);

cudaThreadSynchronize();

gpuMatMul<<<16, 256>>>(d\_Q, d\_covMat, d\_Z, 1, dimension, dimension);

cudaThreadSynchronize();

normZ[0] = 0;

cudaMemcpy(d\_normZ, normZ, normBuffer, cudaMemcpyHostToDevice);

gpuEigVal<<<grid3, block3, sharedMemSize>>>(d\_Q, d\_Z, d\_normZ, dimension);

cudaThreadSynchronize();

cudaMemcpy(normZ, d\_normZ, normBuffer, cudaMemcpyDeviceToHost);

if (abs(prev\_lambda-normZ[0]) < eps) {

cudaMemcpy(Q, d\_Q, imgBuffer, cudaMemcpyDeviceToHost);

break;

}

prev\_lambda = normZ[0];

}

// The new subspace created by k eigenvectors

for (int p = 0; p < dimension; p++) {

DestIndex = p \* k + i;

k\_eig\_vec[DestIndex] = Q[p];

}

* 1. ***Project the training and test matrix***



**Figure 4.4.** Projection training and test data

// Project the training images in the new subspace

float \*d\_kEigVec, \*d\_proj\_train;

int projTrainBuffer = Ntrain \* k \* sizeof(float);

cudaMalloc((void\*\*)&d\_kEigVec, eigVecBuffer);

cudaMalloc((void\*\*)&d\_proj\_train, projTrainBuffer);

cudaMemcpy(d\_kEigVec, k\_eig\_vec, eigVecBuffer, cudaMemcpyHostToDevice);

dim3 grid4((k+block2.x-1)/block2.x, (Ntrain+block2.y-1)/block2.y);

gpuMatMul<<<grid4, block2>>>(d\_X, d\_kEigVec, d\_proj\_train, Ntrain, dimension, k);

// Project the test images in the new subspace

float \*d\_X1;

cudaMalloc((void\*\*)&d\_X1, testMatBuffer);

dim3 grid5(dimension/128, Ntest);

gpuSubMean<<<grid5, block1>>>(d\_testMat, d\_mean, Ntest, dimension, d\_X1);

* 1. ***Identify the test images***

This step I using identify() function same as in C program.

1. **Results**

The prediction results of two versions are written to identify.txt file. Both of versions give the same prediction shown in Table 5.1. There are total 5 error cases.

fp\_id = fopen("identify.txt", "wb");

if (fp\_id == NULL) {

printf("Error open file\n");

return 1;

}

fprintf(fp\_id, "Test Image\t\tPrediction\n");

for (int i = 0; i < Ntest; i++) {

fprintf(fp\_id, "%d \t\t\t\t %d\n", i + 1, recognized\_img[i]);

}

fclose(fp\_id);

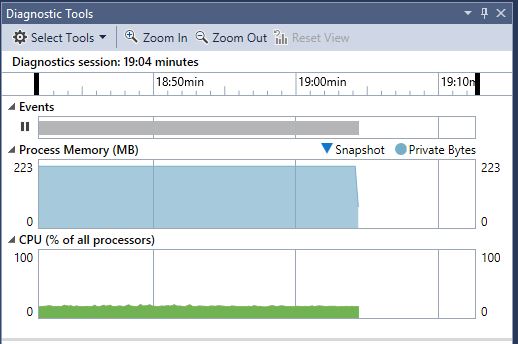
**Table 5.1.** Prediction results

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **ID of test image** | **k = 50** | **Human eyes** | **Incorrect** | **ID of test image** | **k = 50** | **Human eyes** | **Incorrect** |
| 1 | 18 | Not exist | x | 26 | 85 |  |  |
| 2 | 3 |  |  | 27 | 94 | 92 | x |
| 3 | 5 |  |  | 28 | 92 |  |  |
| 4 | 9 |  |  | 29 | 94 |  |  |
| 5 | 14 |  |  | 30 | 96 |  |  |
| 6 | 19 |  |  | 31 | 100 |  |  |
| 7 | 21 |  |  | 32 | 100 |  |  |
| 8 | 27 |  |  | 33 | 101 |  |  |
| 9 | 30 |  |  | 34 | 113 |  |  |
| 10 | 35 |  |  | 35 | 108 |  |  |
| 11 | 36 |  |  | 36 | 116 | 115 | x |
| 12 | 44 |  |  | 37 | 118 |  |  |
| 13 | 46 |  |  | 38 | 119 |  |  |
| 14 | 49 |  |  | 39 | 125 |  |  |
| 15 | 52 |  |  | 40 | 129 |  |  |
| 16 | 55 |  |  | 41 | 134 |  |  |
| 17 | 55 |  |  | 42 | 140 |  |  |
| 18 | 57 |  |  | 43 | 144 |  |  |
| 19 | 67 |  |  | 44 | 144 |  |  |
| 20 | 67 |  |  | 45 | 145 |  |  |
| 21 | 68 |  |  | 46 | 148 |  |  |
| 22 | 68 |  |  | 47 | 149 |  |  |
| 23 | 79 |  |  | 48 | 154 | 150 | x |
| 24 | 81 |  |  | 49 | 151 | 152 | x |
| 25 | 83 |  |  | 50 | 154 |  |  |

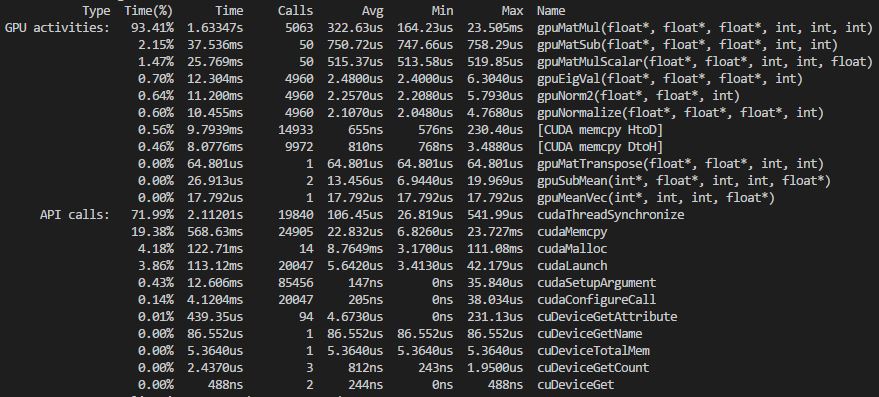
Table 5.2 compare the execution time between two programs. C program is simulated on AMD Phenom X6 110T and finish after 19 min 04 s. While, CUDA C program is executed on Intel Core i7 and GeForce GTX 980 Ti and only consumes 4.68 s, faster than C program more than 200 times.

**Table 5.2.** Compare C and CUDA C program

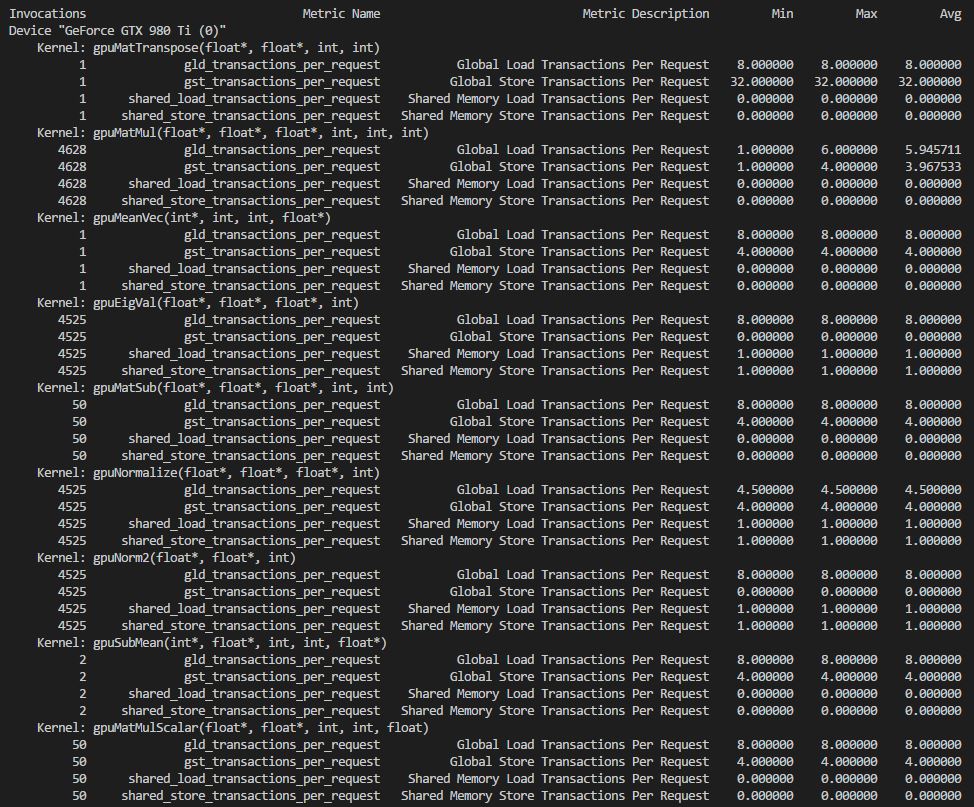
|  |  |  |
| --- | --- | --- |
|  | **C Version** | **CUDA C Version** |
| **Hardware** | AMD Phenom X6 110T | Intel Core i7-7700K & GeForce GTX 980 Ti |
| **Execution time** | 19 min 04 s | 4.68 s |



**Figure 5.1.** Simulation result of C program



**Figure 5.2.** Execution time of each function in CUDA C program



**Figure 5.3.** Load and store transactions in global and shared memory