IMAGE COMPRESSION USING CUDA

Udit Singh Parihar
2018701024
Team Id: 24

PCA

1. Image = $[c_1, c_2, ..., c_n]$
2. mean = $(c_1+c_2+ ... + c_n)/n$
3. Image = $[c_1\text{-mean}, c_2\text{-mean}, ..., c_n\text{-mean}]$
4. Cov = Image’ * Image
5. [V, D] = eigen_decomposition(Cov)
6. [V, D] = sort(V, D)
7. V^k = V[1:k]
8. Red_image = I*V^k

EIGEN DECOMPOSITION

Eigenvalues, Eigenvectors

$\downarrow$

QR Algorithm

$\downarrow$

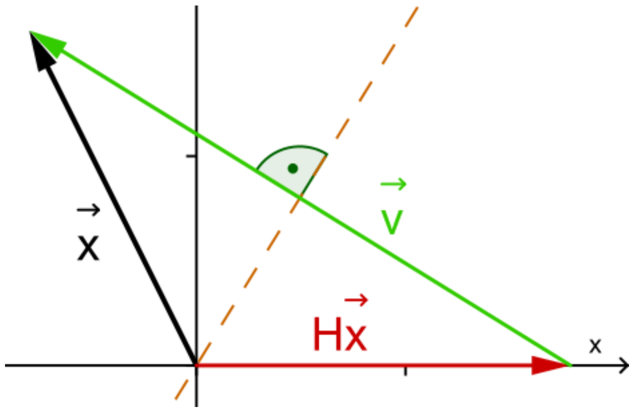
QR Decomposition

$\downarrow$

Householder

HOUSEHOLDER MATRIX

1. $x = [x_1, x_2, \ldots, x_n]^T$ -> $y = [\|x\|_2, 0, \ldots, 0]^T$
2. y = H*x ; where H = I – y*u*u'
3. H is symmetric and preserve length
4. Householder Algorithm



$\beta \leftarrow \max_{1 \le i \le n} |x_i|$
if $(\beta = 0)$ **then**
  $\gamma \leftarrow 0$
else
  $x_{1:n} \leftarrow x_{1:n}/\beta$
  $\tau \leftarrow \sqrt{x_1^2 + \cdots + x_n^2}$
  if $(x_1 < 0)$ **then**
    $\tau = -\tau$
  end if
  $x_1 \leftarrow \tau + x_1$
  $\gamma \leftarrow x_1/\tau$
  $x_{2:n} \leftarrow x_{2:n}/x_1$
  $x_1 \leftarrow 1$

QR DECOMPOSITION

1. $R_1 = H_1 * X$
2. $R_2 = H_2 * R_1$

   .

   .

   .
3. $R = H_n R_{n-1}$
4. $Q = H_1 * H_2 * ... * H_n$

5. Q = Orthogonal, R = Upper Triangular

$$\begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & q_2^{11} & q_2^{12} & \cdots & q_2^{1,n-1} \\ \vdots & \cdots & \ddots & \cdots & \vdots \\ 0 & q_2^{n-1,1} & \cdots & \cdots & q_2^{n-1,n-1} \end{bmatrix}$$

QR ALGORITHM FOR EIGENVALUES

1. for i = 1 : n
2. $Q_i * R_i = qr(A_i)$
3. $A_{i+1} = R_i * Q_i$
4. end

5. Eigen values, $D = A_{inf}$
6. Eigen vectors, $V = Q_1*Q_2*...*Q_n$
7. return [V, D]

MATLAB IMPLEMENTATION

```matlab
function [Q, R] = my_qr(A)
    [m, n] = size(A);
    Q = eye(m);
    R = A;

    for j = 1:n
        u = R(j:end,j);
        normx = norm(u);

        if u(1) < 0
            normx = -normx;
        end

        u(1) = u(1) + normx;
        tau = u(1) / normx;
        u = u/u(1);
        H = eye(size(u,1)) - tau*u*u';
        R(j:end,j:end) = R(j:end,j:end) - tau * (u * u') * R(j:end,j:end);
        Q(:,j:end) = Q(:,j:end) - Q(:,j:end)*(tau * u * u');
    end

    Q*R;
```

1. QR

```matlab
function [V,D] = my_eig(A)
    eps = 1e-15;
    is_converged = 0;
    sum_prev = -10;
    change =0;
    steps=0;

    D = A;
    V = eye(size(D));

    while ~is_converged
        [q, r] = my_qr(D);
        D = r*q;
        V = V*q;

        sum = trace(D);
        change = sum - sum_prev;

        if abs(change) < eps
            is_converged = 1;
        end

        sum_prev = sum;
        steps = steps + 1;
    end

    [V,D] =  sortem(V,D);

    change;
    steps

function [P2,D2]=sortem(P,D)
    D2=diag(sort(diag(D),'descend'));
    [c, ind]=sort(diag(D),'descend');
    P2=P(:,ind);
```

2. Eigens

```matlab
function [] = my_pca(img_name)
    img = imread(img_name);
    img = im2double(rgb2gray(img));
    img = imresize(img, 0.8);

    m = mean(img);
    [row, col] = size(img);
    m = repmat(m, row, 1);
    img = img - m;

    c = img'*img;
    [v,d] = eig(c);
    [v,d] =  sortem(v,d);

    [sz, sz] = size(v);
    v = v(:,1:10);
    size(v)

    red_img = img*v*v' + m;


    imshow(red_img);
    drawnow;

function [P2,D2]=sortem(P,D)
    D2=diag(sort(diag(D),'descend'));
    [c, ind]=sort(diag(D),'descend');
    P2=P(:,ind);
```

3. PCA

C++/Cuda IMPLEMENTATION

```cpp
#ifndef LINEAR_ALGEBRA_H
#define LINEAR_ALGEBRA_H

#include <vector>

namespace la{

    typedef std::vector<float> Vector;
    typedef std::vector<Vector> Matrix;

    void fill_matrix(Matrix& mat, int range);

    void print_matrix(const Matrix& mat);

    void fill_vector(Vector& col, int range);

    void print_vector(const Vector& vec);

    void check_householder(const Matrix& P, const Vector& x);

    void householder(Vector u, Matrix& P);

    Matrix mat_mul(const Matrix& A, const Matrix& B);

    void mat_mul(const Matrix& A, const Matrix& B, Matrix& C,
        int , int , int, int);

    void qr(const Matrix& A, Matrix& Q, Matrix& R);

    float trace(const Matrix& A);

    Matrix trans(const Matrix& mat);

    void sort_index(Vector& v, Vector& idx);

    void sort_eigens(Matrix& V, Matrix& D);

    void eig(const Matrix& A, Matrix& V, Matrix& D);

    void mean_col(const Matrix& A, Matrix& m);

    void mat_sub(const Matrix& A, const Matrix& B, Matrix& C);

    void mat_add(const Matrix& A, const Matrix& B, Matrix& C);

    void pca(const Matrix& A, Matrix& A_red, const int k_col);

    void matrixMul(float *A, float *B, float *C, int N);
}

#endif
```

1. Matrix operations

```cpp
__global__ void matrixMulKernel(float* A, float* B, float* C, int N) {
    int ROW = blockIdx.y*blockDim.y+threadIdx.y;
    int COL = blockIdx.x*blockDim.x+threadIdx.x;

    float tmpSum = 0;

    if(ROW < N && COL < N){
        for (int i = 0; i < N; i++){
            tmpSum += A[ROW * N + i] * B[i * N + COL];
        }
    }

    C[ROW * N + COL] = tmpSum;
}


void la::matrixMul(float *A, float *B, float *C, int N){
    dim3 threadsPerBlock(N, N);
    dim3 blocksPerGrid(1, 1);
        if (N*N > 512){
            threadsPerBlock.x = 512;
            threadsPerBlock.y = 512;
            blocksPerGrid.x = ceil(double(N)/double(threadsPerBlock.x));
            blocksPerGrid.y = ceil(double(N)/double(threadsPerBlock.y));
        }

    matrixMulKernel<<<blocksPerGrid,threadsPerBlock>>>(A, B, C, N);
}
```

2. Parallel matrix multiplication

K-APPROXIMATION USING PCA

1. Red_Mat = Mat * V * V’ + Mean
2. V = EigenMatrix(Cov)

4.0000	7.0000	8.0000	6.0000	4.0000	6.0000	7.0000	3.0000
10.0000	2.0000	3.0000	8.0000	1.0000	10.0000	4.0000	7.0000
1.0000	7.0000	3.0000	7.0000	2.0000	9.0000	8.0000	10.0000
3.0000	1.0000	3.0000	4.0000	8.0000	6.0000	10.0000	3.0000
3.0000	9.0000	10.0000	8.0000	4.0000	7.0000	2.0000	3.0000
10.0000	4.0000	2.0000	10.0000	5.0000	8.0000	9.0000	5.0000
6.0000	1.0000	4.0000	7.0000	2.0000	1.0000	7.0000	4.0000
3.0000	1.0000	7.0000	2.0000	6.0000	6.0000	5.0000	8.0000
7.0000	6.0000	7.0000	10.0000	4.0000	8.0000	5.0000	6.0000
3.0000	6.0000	5.0000	8.0000	5.0000	5.0000	4.0000	1.0000

1. Original Matrix
2. Mat = 10X8
3. V = 8X8

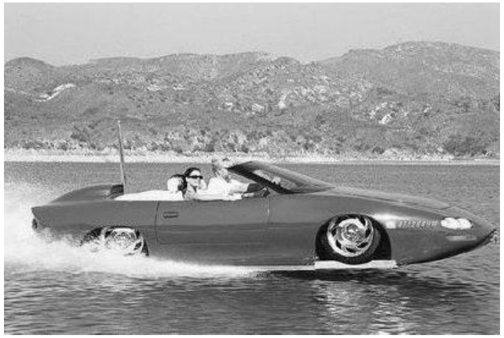
3.1105	6.4575	6.8752	7.0500	4.9582	5.9849	5.5702	3.1765
10.4180	2.1742	3.8540	8.5842	1.7501	9.1353	4.5702	8.0431
1.1208	7.0549	3.2116	6.9609	1.9632	8.9302	8.2112	10.0749
3.3601	1.3330	3.2130	3.9268	8.2808	5.5711	10.2071	3.4129
3.0731	9.0545	10.0724	7.9543	3.9921	6.9545	2.0823	3.0387
9.5754	3.6353	1.6652	9.9644	4.5703	8.5841	8.7218	4.4014
5.7929	0.9617	3.4721	6.8420	1.8893	1.2599	6.5649	3.6687
2.7940	0.7989	6.8781	1.8095	5.5249	6.4635	4.9716	7.4975
6.9389	6.2503	6.0877	9.0502	3.3453	8.5988	4.4870	5.1204
3.8162	6.2798	6.6706	7.8579	4.7257	4.5177	5.6137	1.5657

1. k = 5 approx
2. V_red = 8X5

2.7666	5.6096	6.9523	6.1331	4.7402	5.8295	5.3367	4.0619
9.1495	2.1526	1.9442	8.6106	2.9105	8.0315	7.5182	6.7429
5.1948	4.2945	5.0472	7.0756	4.0442	6.6672	6.1666	5.0818
4.8321	4.4910	5.3318	6.9348	4.1481	6.5421	6.0426	4.9295
1.1065	6.5087	8.2549	5.4888	5.2161	5.2568	4.7694	3.3646
8.9577	2.2565	2.0947	8.5362	2.9655	7.9653	7.4526	6.6623
5.9164	3.9037	4.4809	7.3557	3.8373	6.9162	6.4132	5.3849
3.6770	5.1166	6.2381	6.4865	4.4792	6.1436	5.6478	4.4443
5.5643	4.0944	4.7572	7.2190	3.9382	6.7947	6.2929	5.2370
2.8351	5.5725	6.8986	6.1597	4.7206	5.8531	5.3601	4.0907

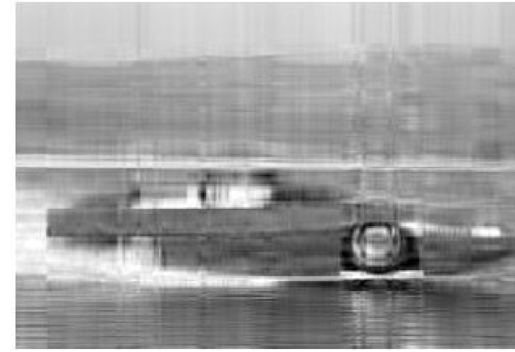
1. k = 1 approx
2. V_red = 8X1

K-APPROXIMATION IMAGES



Original Image

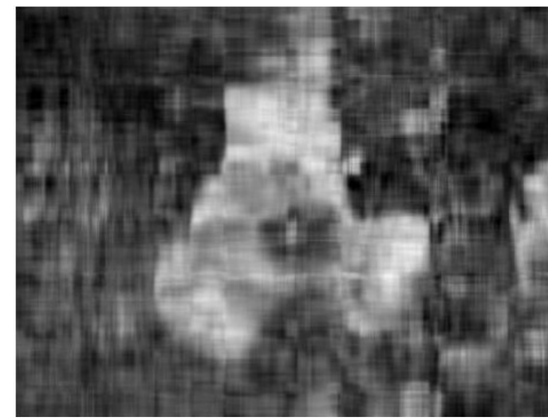
k = size/2 approx



K = 10 approx

Original Image

k = size/2 approx



K = 10 approx

Original Image

k = size/2 approx

K = 20 approx

CONCLUSION

1. Speed using Hetergenous parallel programming
2. Using OpenMP to avoid copying back and forth from device to host.