CUDA Accelerated Singular Value Decomposition for PCA

E4040_2019Fall_PCA_report
Ananye Pandey ap3885, Dwiref Oza dso2119
Columbia University

Abstract

Principal Component Analysis is a useful dimensionality reduction tool used to essentially summarize high dimensional data. A key underlying step in PCA is Singular Value Decomposition. SVD can be computed using a variety of algorithms. Of the many available algorithms that can be used to iteratively converge to the singular decomposition of a matrix, we demonstrate that the Given's rotation calculation in the Jacobi method is parallelizable and can thus allow for faster computation of the eigenvalues and eigenvectors.

1. Overview

1.1 Problem in a Nutshell

PCA is used widely as a data preprocessing step in various Machine Learning (ML) algorithms. Most data gathered in real life can be shown to have many unnecessary parameters explaining little to no variance in the data, or we may find a heavy correlation between different parameters. In either case, it is helpful in terms of both computing resources and time to reduce the dimensionality of the data.

Our motive was to select an algorithm to calculate the eigenvalues and eigenvectors that can be parallelized. One such method is the Jacobi rotation method. Jacobi method iteratively converts the covariance matrix of the desired data to the diagonal matrix containing its singular values (eigenvalues). This is done by iteratively rotating each off-diagonal element to zero by multiplying it by a Given's rotation matrix. We show that it can be parallelized by being able to rotate all individual off-diagonal elements (not belonging to the same row or column) in parallel.

1.2 Prior Work

There are many sequential iterative algorithms to calculate the eigenvalues and eigenvectors of a real symmetric matrix such as QR decomposition, the Jacobi method and power method.

B. Butrylo et al. released a paper on solving eigenvalues of a matrix by using SMP clusters, called SSOR method [1]. Using clusters to solve for eigenvalues is quick for larger matrices, but also increases equipment costs and power consumption.

J. Xia et al. presented a General Purpose GPU (GPGPU) implementation for solving the maximum eigenvalues of a matrix, and QR method for solving all eigenvalues on OpenGL [2].

Due to the structural differences in CUDA and Python, and owing to the fact that GPGPUs provide good floating point calculation, parallelism and large memory bandwidth, we use CUDA for parallelizing the Jacobi method.

2. Description

We first give an overview and working of the serial Jacobi algorithm to calculate the eigenvalues and eigenvectors of the given data matrix. We then explain all the parallelizable modules and then implement them on CUDA.

2.1. Objectives and Technical Challenges

Our main objective was to implement a faster version of SVD by parallelizing major serial operations. This can be accomplished mainly by selecting the appropriate off-diagonal elements of the covariance matrix which are independent of each other and rotating them parallelly. Along with this parallelization, we parallelize matrix multiplication as well since it gives us faster results.

A major technical challenge is to efficiently manage GPU device memory. Since there are still some iterations to be performed, the GPU memory can run out.

2.2. Problem Formulation and Design

Given the data matrix X, we calculate its eigenvalues and eigenvectors for PCA as follows:

1. The Hermitian X^T (Transpose, since most real-life data is real) of the input matrix X.

2. The covariance matrix of the data is calculated. (X^TX)
3. The eigenvalues and eigenvectors are calculated using Jacobi method.
4. The eigenvalues are sorted in descending order and the eigenvalue columns are interchanged accordingly.
5. Square root the eigenvalues to give the singular values of the data matrix X.
6. Corresponding eigenvector matrix calculated is V. Calculate its transpose V^T.
7. U is calculated as $U = MV\Sigma^{-1}$
8. The first K columns of vector V^T are selected as vectors for the new reduced feature space.

The Jacobi method was introduced in 1846 by Carl Gustav Jacob Jacobi but gained popularity only after computers were invented in the 1950s. The method is briefly described:

1. Calculate the largest off diagonal element to reduce.
2. Calculate the rotation matrix to be used
3. Rotate the corresponding off-diagonal element to zero.
4. Repeat steps 1-3 until the input covariance matrix is completely diagonalized. Resulting matrix is the Σ^2 matrix.
5. Repeat steps 1-3 in exactly the same way as in step 4 for a unit diagonal matrix of the same size as the input size. The resulting matrix is eigenvector matrix V.

3. Software Design

3.1 Serial Design

The algorithm for the serial algorithm is given in Figure 5. The basic operations in each iteration revolve mainly around 1) calculating the index of each off diagonal element, 2) rotating the matrices, 3) rotating eigenvectors and 4) updating values.

For these operations, we introduce three functions:

1. maxind():
   This function calculates the index of the largest element in each row after a given index k by comparing the absolute value of the corresponding index and updating its value accordingly.

```
m1 = k + 1
for i in range(k+2,size):
    if(abs(A[k][i])>abs(A[k][m1])):
        m1 = i
return m1
```

Figure 1. Function to find maximum column index

2. rotate():
   This function is introduced to calculate the given's rotation of a given row-pair elements.

$$\begin{matrix} eik \\ eil \end{matrix} = \begin{bmatrix} cos & -sin \\ sin & cos \end{bmatrix} \cdot \begin{matrix} eik \\ eil \end{matrix}$$

```
kl = c * A[k][l] - s * A[i][j]
ij = s * A[k][l] + c * A[i][j]
A[k][l] = kl
A[i][j] = ij
return A
```

Figure 2. Function to rotate row pair (i,j) and (k,l)

3. update():
   This function is used to update the changed status of the eigenvalues and reduce the state of the system accordingly.

```
y = e[k]
e[k] = y + t
if (changed[k]==True and y == e[k]):
    changed[k] = False
    state -= 1
elif (changed[k]==False and y!=e[k]):
    changed[k] = True
    state += 1
return changed, state
```

Figure 3. Function to update the state of the system and status of eigenvalues

The sine and cosine values for each iteration are calculated using Pythagoras theorem as follows:

```
p = As[k][l]
y = 0.5 * (e[l]-e[k])
d = abs(y) + np.sqrt(p*p + y*y)
r = np.sqrt(p*p + d*d)
c = d/r
s = p/r
```

Figure 4. Algorithm to calculate the sine (s) and cosine (c) values in each iteration

Algorithm 1: Serial PCA using Jacobi method

Result: Sigma, VT
initialization;
input matrix := D (NxP);
covariance matrix = As := D.T * D;
E := diag1xP;
iterations := 0;
for *i in range(P)* **do**
  index[i] := maxind(As, P, i);
  e[i] := As[i][i];
  changed[i] := True;
end
while *iterations* $< 1e7$ **do**
  m:=0;
  for *k in range(0,P)* **do**
    if $|As[k, index[k]]| < |As[m, index[m]]|$ **then**
      m:=k;
    end
  end
  k:=m; l:=index[k]; p:=As[k,l];
  $y := 0.5 \times e[k] - e[l]$; d:= $|y| + \sqrt{p^2 + y^2}$;
  $r := \sqrt{p^2 + d^2}$; c:=d/r; s:=p/r;
  if $y < 0$ **then**
    s:=-s; t:=-t;
  end
  As[k,l]:=0; update(k,-t); update(l,t);
  for *i in range(0,k-1)* **do**
    rotate(i,k,i,l)
  end
  for *i in range(k+1,l-1)* **do**
    rotate(k,i,i,l)
  end
  for *i in range(l+1,P)* **do**
    rotate(k,i,l,i)
  end
  for *i in range(0,P)* **do**
    $E[i,k] = c \times E[i,k] - s \times E[i,l]$; $E[i,l] = s \times E[i,k] + c \times E[i,l]$;
  end
  sigma=sort(e); indices:=argsort(e);
  for *i in range(0,P)* **do**
    for *j in range(0,P)* **do**
      U[i,j]:=E[i,indices[j]]
    end
  end
end

Figure 5. Serial Algorithm for SVD using Jacobi method

3.2 Parallel Design

We have recognized the main parallelable parts of the algorithm. These include transpose, matrix multiplication and parallel row pair reduction.

1. Transpose():

```
__global__ void parTranspose(float *idata, float *odata, int cols, int rows) {
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;
    if ((ix < cols) && (iy < rows)) {
        odata[iy*cols + ix] = idata[ix*rows + iy];
    }
}
```

Figure 6. Transpose kernel

For transpose we use block size of [32*32] and grid size is calculated according to matrix_size/block_size.

2. Multiply():

```
#define BLOCK_SIZE 16
__global__ void kernel_MatMul(float *A, int rA, int cA, float *B, int rB, int cB, float *C) {
    int bIDx = blockIdx.x, bIDy = blockIdx.y, tIDx = threadIdx.x, tIDy = threadIdx.y;
    int row_ = bIDy * BLOCK_SIZE + tIDy;
    int col_ = bIDx * BLOCK_SIZE + tIDx;
    __shared__ float A_sub[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float B_sub[BLOCK_SIZE][BLOCK_SIZE];
    float C_sub = 0.0;
    for (int m = 0; m < (BLOCK_SIZE + cA - 1) / BLOCK_SIZE; m++) {
        if (m * BLOCK_SIZE + tIDx < cA && row_ < rA) {
            A_sub[tIDy][tIDx] = A[row_ * cA + m * BLOCK_SIZE + tIDx];
        }
        else {
            A_sub[tIDy][tIDx] = 0.0;
        }
        if (m * BLOCK_SIZE + tIDy < rB && col_ < cB) {
            B_sub[tIDy][tIDx] = B[(m * BLOCK_SIZE + tIDy) * cB + col_];
        }
        else {
            B_sub[tIDy][tIDx] = 0.0;
        }
        __syncthreads();
#pragma unroll
        for (int k = 0; k < BLOCK_SIZE; k++) {
            C_sub += A_sub[tIDy][k] * B_sub[k][tIDx];
        }
        __syncthreads();
    }
    if (row_ < rA && col_ < cB) {
        C[cB * BLOCK_SIZE * bIDy + BLOCK_SIZE * bIDx + cB * tIDy + tIDx] = C_sub;
    }
}
```

Figure 7. Tiled Matrix Multiplication Kernel

The matrix multiplication kernel is implemented using a block size of [16*16] and grid size similar to transpose.

3. Compute rowpairs:

The first step of the algorithm is to calculate the rowpairs which are independent and can be rotated parallely. This is done by the chess_compute kernels.

```
__device__ void chess_tourney_params(int P, int *row_pair, int iter) {
    //NOTE: here, row_pair is thread-local
    int localID = threadIdx.x;
    int index1, index2;
    index1 = (localID + iter) % (P - 1);
    if (localID != 0) {
        index2 = (P - localID + iter - 1) % (P - 1);
    }
    else {
        index2 = P - 1;
    }
    row_pair[0] = min(index1, index2);
    row_pair[1] = max(index1, index2);
}
__global__ void kernel_compute_all_chess_params(int P, int *device_IterBlockToElem) {
    int blockID = blockIdx.x;
    int index = blockID*P + threadIdx.x*2;
    int *row_pair = (int *) malloc(sizeof(int)*2);
    chess_tourney_params(P, row_pair, blockID);
    device_IterBlockToElem[index] = row_pair[0]; //|=(P-1)X(P/2*2)
    device_IterBlockToElem[index+1] = row_pair[1];
    free(row_pair);
}
```

Figure 8. Kernels to compute rowpairs

We calculated the block size to be used as[P-1 * P/2] and grid size calculated as in transpose.

4. Compute parameters:

This function is used to calculate the sine and cosine rotation parameters to perform Given's rotation on any row pair.

```
__global__ void kernel_compute_params(float *device_A, int P, int iter,
    /*1 Block, P/2 threads: threadID t handles params for*/
   /* its alloted pair (for a particular device_iter)*/
    # define EPSILON 1e-4
    int localID = threadIdx.x;
    int k, l;
    float elem, y, d, r, c, s; //,t
    k = device_IterBlockToElem[iter*P+localID*2]; //row
    l = device_IterBlockToElem[iter*P+localID*2+1]; //col
    elem = device_A[k * P + l];
    __syncthreads();
    y = (device_A[l * P + l] - device_A[k * P + k]) * 0.5;
    __syncthreads();
    d = fabs(y) + sqrt(elem * elem + y * y);
    r = sqrt(elem * elem + d * d);
    if (r < EPSILON) {
        c = 1.0;
        s = 0.0;
    }
    else {
        c = d / r;
        s = y / fabs(y) * elem / r; //t=y/fabs(y)*p*p/d;
    }
    __syncthreads();
    if (k<P && l<P){
    device_cosine[k * P + l] = c;
    device_sine[k * P + l] = s;
    }
}
```

Figure 9. Compute parameters kernel

Grid size is set as [P*P] for this kernel.

5. Row Update Kernel:

This kernel is written to perform given's rotation off all independent off-diagonal elements calculated using rowpair on the covariance matrix.

```
__global__ void kernel_row_update(int iter, float *device_A, float *device_X,
    int localID = threadIdx.x;
    int blockID = blockIdx.x;
    /*Based on blockID [total blocks=P/2],
    compute the corresponding two rows: p,q for device_iter*/
    __shared__ int row_pair[2];
    __shared__ float params[2]; //[sin_, cos_]
    if (localID == 0)
    {
        row_pair[0] = device_IterBlockToElem[iter*P+blockID * 2];
        row_pair[1] = device_IterBlockToElem[iter*P+blockID * 2 + 1];
        params[0] = device_sine[row_pair[0] * P + row_pair[1]];
        params[1] = device_cosine[row_pair[0] * P + row_pair[1]];
    }
    __syncthreads();
    //CHECKPOINT: Can you reduce shared-memory bank conflicts here?
    int k = row_pair[0], l = row_pair[1];
    float sin_ = params[0], cos_ = params[1], elem_k=device_A[k*P+localID];
    float elem_l=device_A[l * P + localID];

    device_X[localID * P + k] = elem_k * cos_ - elem_l * sin_;
    device_X[localID * P + l] = elem_k * sin_ + elem_l * cos_;
}
```

Figure 10. Kernel to perform rotation on covariance matrix

Block size was set as [P*P] and grid size was calculated same as in transpose.

6. Column update kernel:

This kernel operates on an initial matrix E which is a diagonal [P*P] matrix with diagonal elements equal to one and all off-diagonal elements equal to zero. The columns of the matrix E are rotated using the same Given's rotation applied to the covariance matrix to converge to the eigenvector matrix V.

```
__global__ void kernel_col_update(int iter, float *device_A, float *device_X, int P,
    int localID = threadIdx.x;
    int blockID = blockIdx.x;
    __shared__ int col_pair[2];
    __shared__ float params[2]; //[sin_, cos_]
    if (localID == 0)
    {
        col_pair[0] = device_IterBlockToElem[iter*P+blockID * 2];
        col_pair[1] = device_IterBlockToElem[iter*P+blockID * 2 + 1];
        params[0] = device_sine[col_pair[0] * P + col_pair[1]];
        params[1] = device_cosine[col_pair[0] * P + col_pair[1]];
    }
    __syncthreads();

    int k = col_pair[0], l = col_pair[1];
    float sin_ = params[0], cos_ = params[1];

    float new_eigen_k, new_eigen_l;

    int kp = k*P + localID, lp = l*P+localID;
    device_A[kp] = device_X[kp] * cos_ - device_X[lp] * sin_;
    __syncthreads();
    device_A[lp] = device_X[kp] * sin_ + device_X[lp] * cos_;
    __syncthreads();
    new_eigen_k = device_eigenvectors[kp]*cos_ - device_eigenvectors[lp]*sin_;
    __syncthreads();
    new_eigen_l = device_eigenvectors[kp]*sin_ + device_eigenvectors[lp]*cos_;
    __syncthreads();
    device_eigenvectors[kp] = new_eigen_k;
    device_eigenvectors[lp] = new_eigen_l;
}
```

Figure 11. Kernel to update columns of eigenvector matrix

Block size was set as [P/2 * P/2] and grid size calculated as in transpose.

7. Convergence Algorithm:

The convergence of the algorithm requires P-1 steps for a covariance matrix of size [P*P].

```
while(itr < P - 1):
    # Compute rotation parameters: sine and cosine
    # for all (p, q), q>p
    sin, cos = cP.compute_params(A, np.int32(P), np.int32(itr), iterBlock)

    # row update
    X = dU.row_update(np.int32(itr), np.float32(A), np.float32(X),
                      np.int32(P), np.float32(sin), np.float32(cos), iterBlock)

    # col update
    eigenvectors = dU.col_update(np.int32(itr), np.float32(A), np.float32(X),
                                 np.int32(P), np.float32(sin),
                                 np.float32(cos), iterBlock)

    itr += 1
```

Figure 12. Convergence Algorithm

4. Results

4.1 Tools Used:

The serial code was written on Python 3.7 and run on an Intel i5 8th gen CPU.

The parallel code was written on pycuda on Python 3.7 and run locally on an Nvidia GeForce RTX2070 using CUDA 10.0.

4.2 Speedup in implementation of multiplication on GPU:

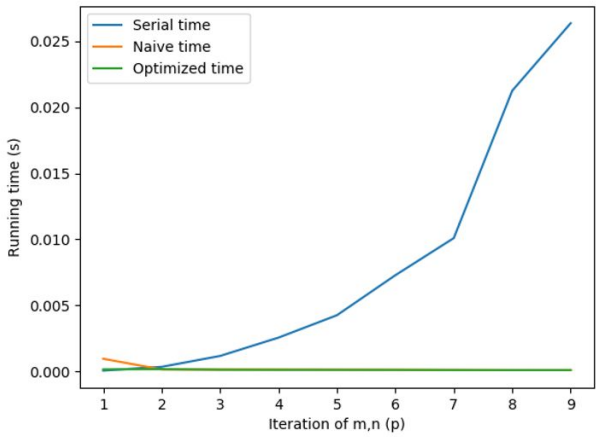

Figure 13. Speedup of implementing naive and tiled matrix multiplication on a GPU as compared to numpy dot

It is evident that a trivial operation like multiplication when parallelized works much faster than multiplication implemented on the serial algorithm.

4.3 Evaluation of serial and parallel code on small matrices:

The following graph shows the running time of the parallel algorithm for the given array sizes.

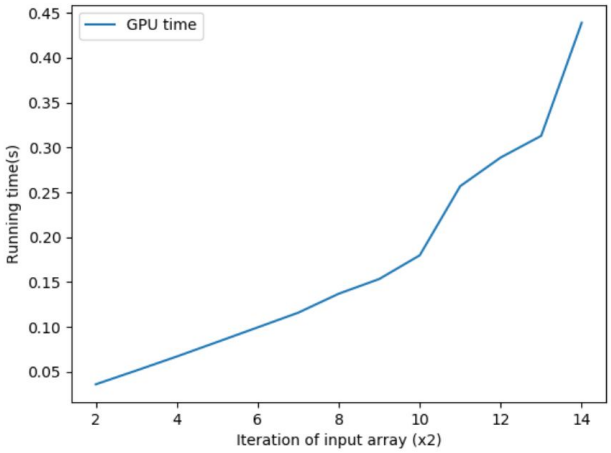

Figure 14. Running time of Parallel SVD

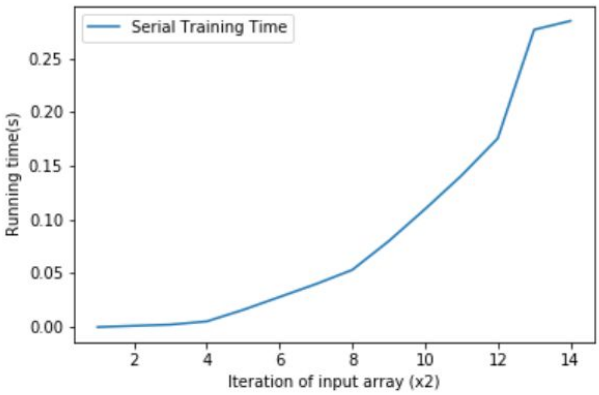

Figure 15: Runtime of serial SVD

The runtime of the parallel SVD is highly affected by the time taken to copy arrays from device to host and vice versa, increasing with each iteration.

5. Discussion and Further Work

The parallel version works comparatively faster compared to the serial iterative algorithm. However, we noticed that on the off chance when an eigenvector value remains untouched during rotation, the algorithm rarely returned nan values.

The column, row and parameter update step as not as efficient as they can be since there is a measurably low overlap between host to device memory copies as shown in the figure.

Furthermore, due to kernels with an if-else structure, concurrency is affected. This is the reason why Nvidia Visual Profiler logs the fact that there is low kernel concurrency when the same kernel is executed in parallel threads.

Figure 16. Profiling section showing time taken by each of the kernels and the nearly overlapping memory transfers

The following tables provide a summary of the memory transfer operations of memory from device to host and vice versa.

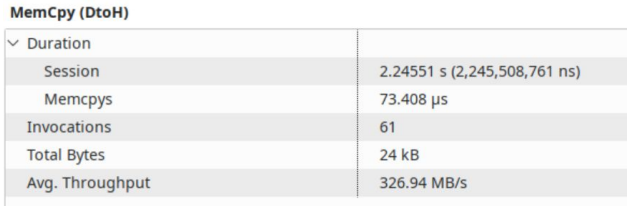

Figure 17. CUDA Memory copy from device to host properties

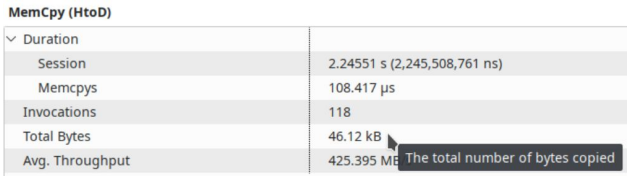

Figure 18. CUDA Memory copy from host to device properties

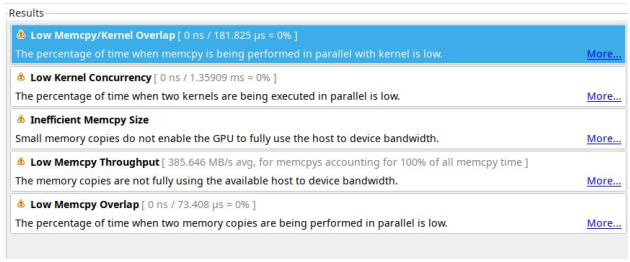

Figure 19. Nvidia Visual Profiler's Analysis tab provides a report of the code's memory access efficiency and concurrency

6. Conclusion

One of the major issues of efficient implementation of CUDA code is efficient device memory management. The total amount of memory on the GPU device is limited and hence it is not uncommon to run into memory errors for larger data matrices.

Although our code works for data matrices upto size 17, we can further strive to improve the memory efficiency by introducing memory pooling or deallocating memory more regularly while interfacing with pycuda.

A parallel implementation of Jacobi SVD algorithm was thus implemented by us.

7. Acknowledgements

We would like to thank Dr. Zoran Kostic for his consistent guidance and support.

8. References

[1] B. Boguslaw, J. Erricos, "*Handbook of Parallel Computing*"

[2] M. Berry, A. Sameh, *"An Overview of Parallel Algorithms for the singular value and symmetric eigenvalues problems,"* Journal of Computational and Applied Mathematics, 2012

[3] S. Labahar, P. J. Narayanan, *"Singular Value Decomposition on GPU using CUDA,"* 2009

[4] Z. Liang, *"Jacobi SVD,"* A matlab implementation, https://github.com/zlliang/jacobi-svd

9. Appendices

Individual Student Contributions (in %)

Task	% ap3885	% dso2119
Overall	**50**	**50**
Serial	**75**	**25**
Parallel	**25**	**75**
Presentation Report	**50**	**50**