Computer Vision + ML using GPU/CUDA

CuML [GitHub]

(Implementation of various ML/DL algorithms/models in CUDA)

CSN- 221: Computer Architecture and Microprocessor (CAM)

Jitesh Jain [19114039] - *CuCNN*

Anurag Bansal [19114010] - *CuSVD*

Anushka Singh [19114011] - *CuKNN*

Hardik Thami [19114035] - *CuSVM*

Anshika Mittal [19114009] - *CuDecisionTree*

Abstract

With the rise of the AI/ML/DL algorithms or models in many parts of the industry, efficient techniques for executing these are the need of the hour. That's where CUDA, an architecture developed by NVIDIA specifically for its GPU, comes into the picture. Training and testing the algorithms on GPUs decreases the time taken and helps gain better accuracy due to batch-wise training.

In this report, we present five algorithms implemented using CUDA along with their results.

Contents

CuCNN

(Cuda Convolutional Neural Network)

- Jitesh Jain

1. Problem Statement

Implement a simple ***3-layer CNN***[5][6] (Convolutional Neural Network) for Image Classification on ***MNIST***[1][2] using ***CUDA***[3][4] ***Programming*** and observe the *effect of different kernel settings on the performance of the model*.

2. Novelty

This work presents a study about the *effect of using different Grid Size and Block Size on a CNN* model's performance, which is supported by the experiments.

It also presents forward a *simple 3-layer* CNN architecture that attains a *test accuracy in access of 97%* along with a *training time of only 5* minutes on a **Tesla T4** (available for free on *GoogleColab*[7]) system.

3. Evaluation Parameters

I use the **MNIST**[1][2] dataset for training and testing of our CNN model. **MNIST**[1][2] is an extensive database of handwritten digits

(0-9) containing **60000 images of size *28x28***. Due to its ***wide usage*** for *testing and training of classification models in the field of Machine Learning*, we consider it as a **benchmark** for evaluating the model's performance.

I evaluate the model based on three parameters:

- ***Accuracy:*** The *ratio of the number of correctly classified images to the total number of images*.
- ***Train Error:*** The *total error* calculated during the training as the *sum of the euclidean norm* (implemented in CUDA using *cublasSnmr2* in the *cuBLAS*[8] library) of the vector containing the sum of the differences between the target and predicted probability for each digit of a given image *divided by the number of images*.
- ***Training Time:*** The *time taken to train* the CNN model on a Tesla T4 GPU for 50 epochs.

4. Methodology

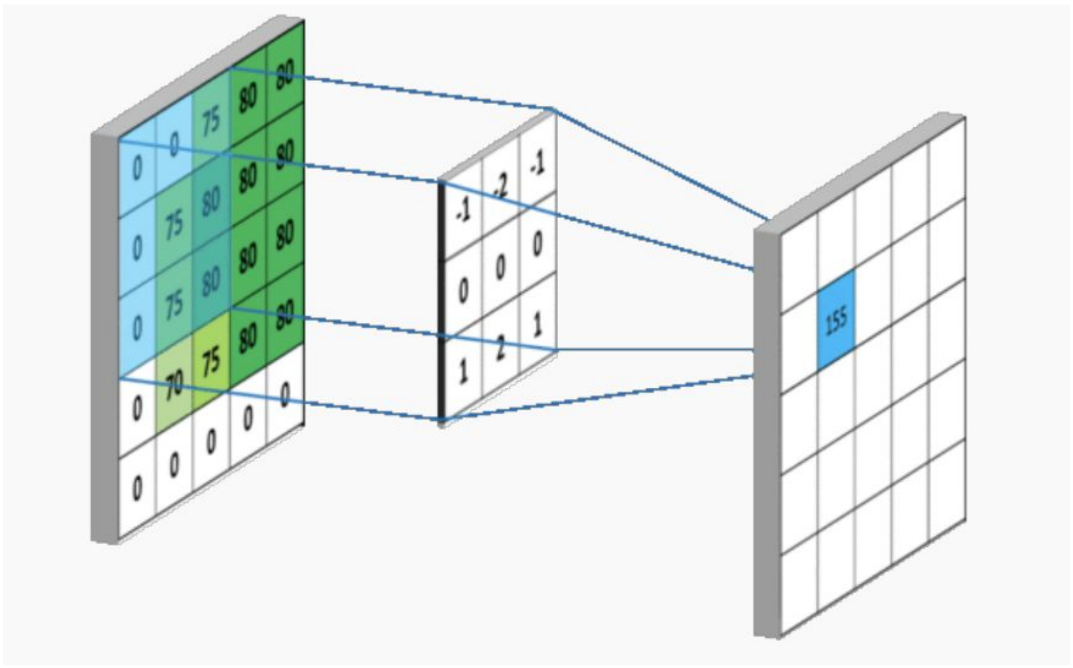
In this section, I first provide an overview of CNN in ***Section 4.1***. I briefly explain our 3-layer CNN architecture in ***Section 4.2***, followed by introducing the terms in CUDA programming in ***Section 4.3***. Finally, in ***Section 4.4***, I present the implementation details.

4.1 Convolutional Neural Network (CNN)

A **CNN**[5][6] is a Neural Network specifically designed for computer vision applications like image classification, semantic segmentation, etc.

The basic building blocks of a CNN are:

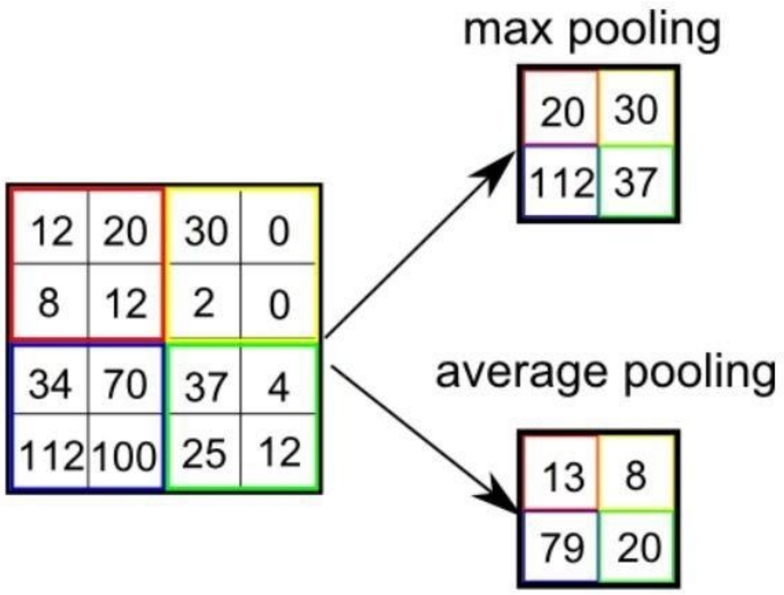
1. **Convolution layer**: It consists of *sliding window* operations where a ***filter*** (also called the ***kernel***, the *middle one in the image below*) *slides* over the ***input feature map*** (*left one in the image below*) and *outputs one value corresponding to* each of the overlaps. It works using *parameter sharing* (as only the kernels have the learnable parameters), thus reducing the parameters by a considerable margin compared to a *linear deep neural network*[9].



Convolution Operation

[**Source**]

2. **Pooling Layer:** It is used to reduce the spatial size of the feature map output by a convolution layer. There are two types of pooling: Average Pooling and Max Pooling (see the image below). I don't use a pooling layer in the architecture because the input image already has a small size (28 x 28).



Pooling Operation
[Source]

3. **Activation Function:** It is generally used to induce non-linearity in the network to make the model more robust to variations in the data. I use ***sigmoid*** as the activation function.

$$S(x) = 1 / (1 + e^{x})$$

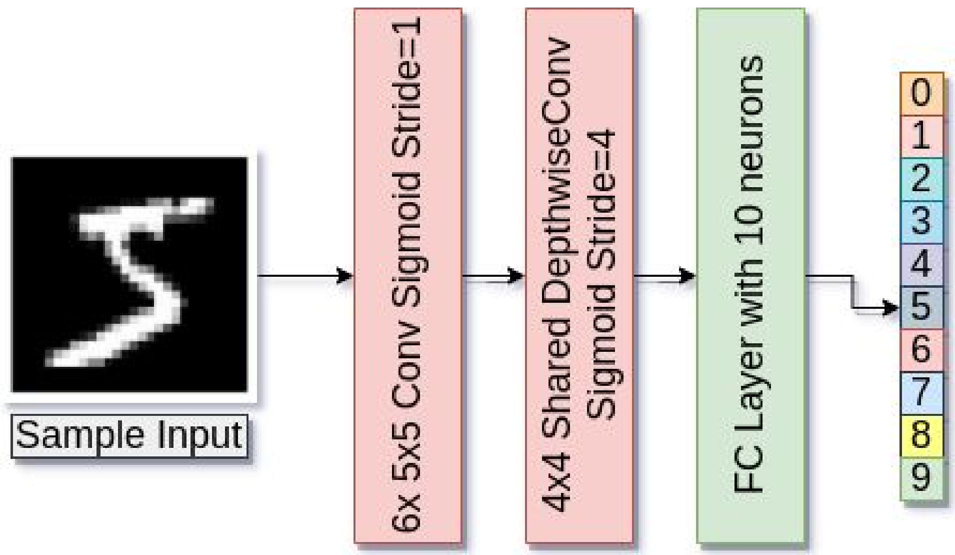
Sigmoid Function

4. **Fully-Connected Layer:** It is a linear layer with multiple nodes at the end stages of the network to predict the probability of each category. I use an FC layer with *10* nodes in the architecture.

4.2 The CNN Architecture

The 3-Layer CNN consists of:

- ***Convolution Layer:*** Applies a **Convolution Operation** with *6* kernels of size *5 x 5* with *stride=1* on the input image (size= *28 x 28*) to output a map of shape=*24 x 24 x 6.*
- ***Shared Depthwise Convolution Layer:*** Applies a *shared Depthwise* (the same kernel applied to different channels of the input from the previous layer) *Convolution Operation* with a *4 x 4* kernel with *stride = 4* on each channel of the previous Convolution layer's output feature map to output a map of shape = *6 x 6 x 6.*
- **Fully Connected Layer:** *Flattens* the output from the previous layer to a layer with *10 nodes,* with each node's value representing the *probability* of a *digit from 0-9.*


Sample Input
6x 5x5 Conv Sigmoid Stride=1
4x4 Shared DepthwiseConv Sigmoid Stride=4
FC Layer with 10 neurons
0
1
2
3
4
5
6
7
8
9

3-layer CNN Architecture (made using draw.io)

4.3 CUDA Programming

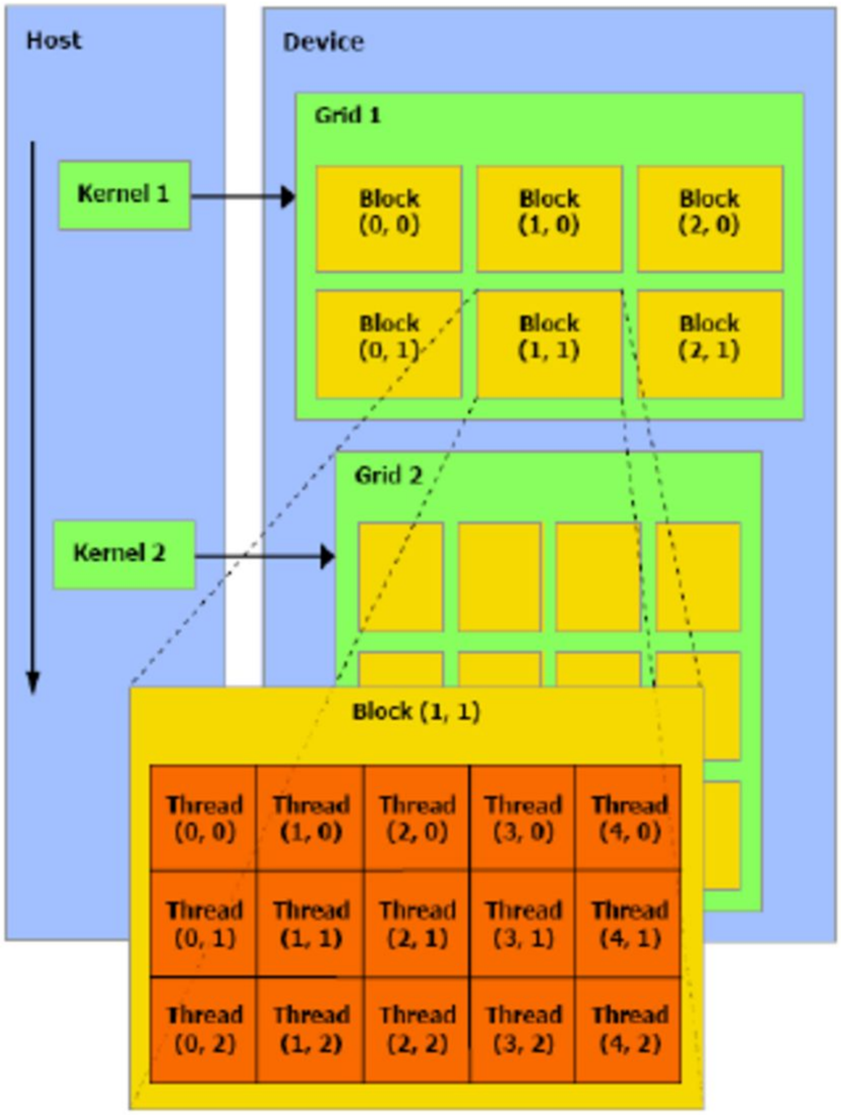
CUDA[3][4] is a parallel computing platform and programming model developed by Nvidia for general computing on its GPUs. CUDA enables developers to speed up compute-intensive applications by harnessing GPUs' power for the computation's parallelizable part.

Some technical terms:

- **Host:** The *CPU* that makes calls to the kernels.
- **Device:** The *GPU*.
- **__global__ Function:** Called by the *host* and executed on the *device*.
- **__device__ function:** Called by the *device* and executed on the *device*.
- **Block Size:** Number of threads inside a block.
- **Grid Size:** Number of blocks inside a grid.

4.4 Implementation Details

I implemented specific feed-forward *and backpropagation* functions for our CNN model along with activations and error functions, all using the **CUDA** framework. Some important details are:



High-level Overview of CUDA

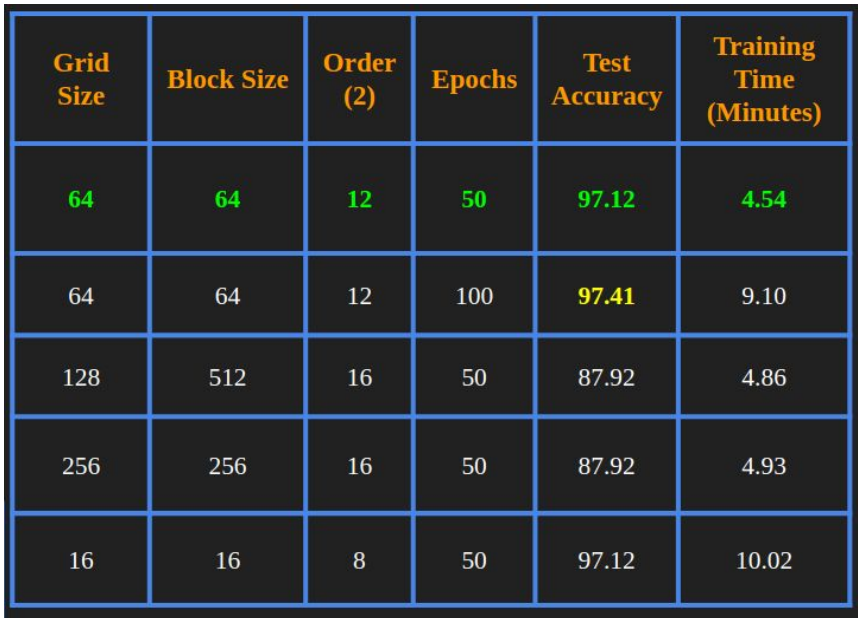
[Source]

- I used simple ***float multi-dimensional arrays*** to implement the *parameter structures* (weights and biases) and *outputs* for the CNN.

- The functions called during feedforward propagation and backpropagation were specified as **__global__** as they are called during training from the **host (CPU).**
- I train our model for ***50 epochs (or iterations***, in different settings) and 100 epochs (or iterations, in one setting).
- The **best performing** *kernel sizes* are: **Grid Size = 64**, **Block Size = 64**, i.e, ***kernel <<<64,64>>>.***

5. Results/Experiments

I perform experiments on *different Block Sizes* and *Grid Sizes* settings for the kernel and make *useful observations*. All experiments were performed on a **Tesla T4 GPU.**

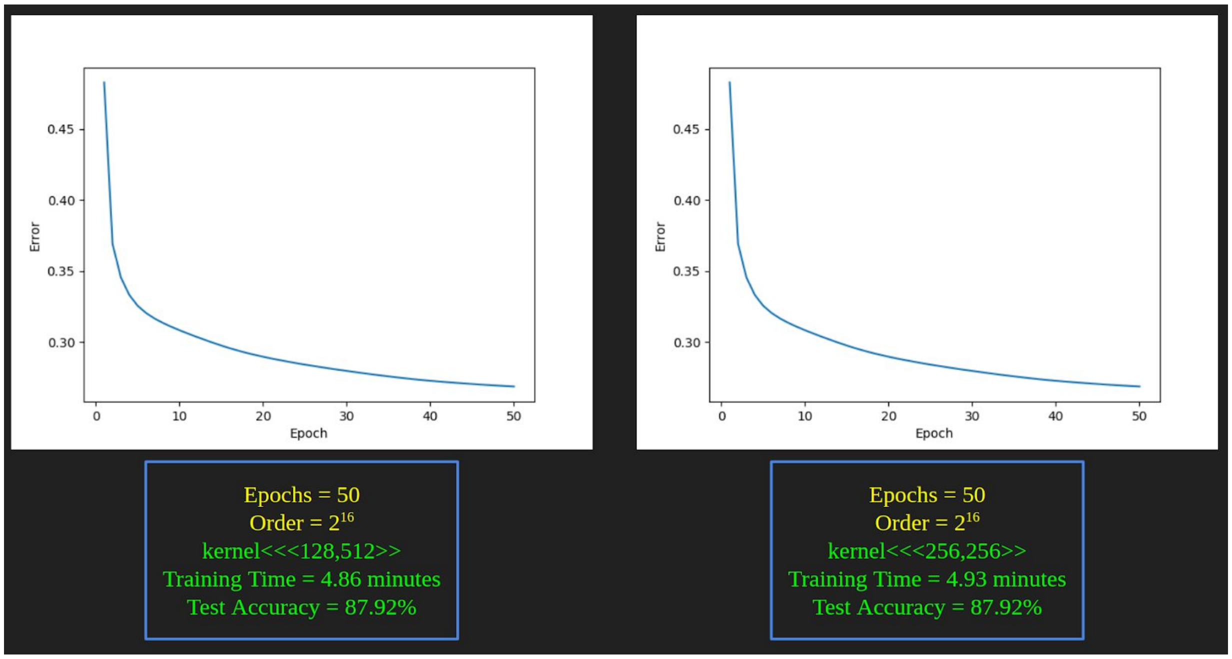


Grid Size	Block Size	Order (2)	Epochs	Test Accuracy	Training Time (Minutes)
64	**64**	**12**	**50**	**97.12**	**4.54**
64	64	12	100	**97.41**	9.10
128	512	16	50	87.92	4.86
256	256	16	50	87.92	4.93
16	16	8	50	97.12	10.02

Results

Observations

- The model's performance depends on the ***Grid Size*** (number of Blocks) and ***Block Size*** (number of Threads).
- Products of the same order give almost the same results (in terms of time and accuracy).

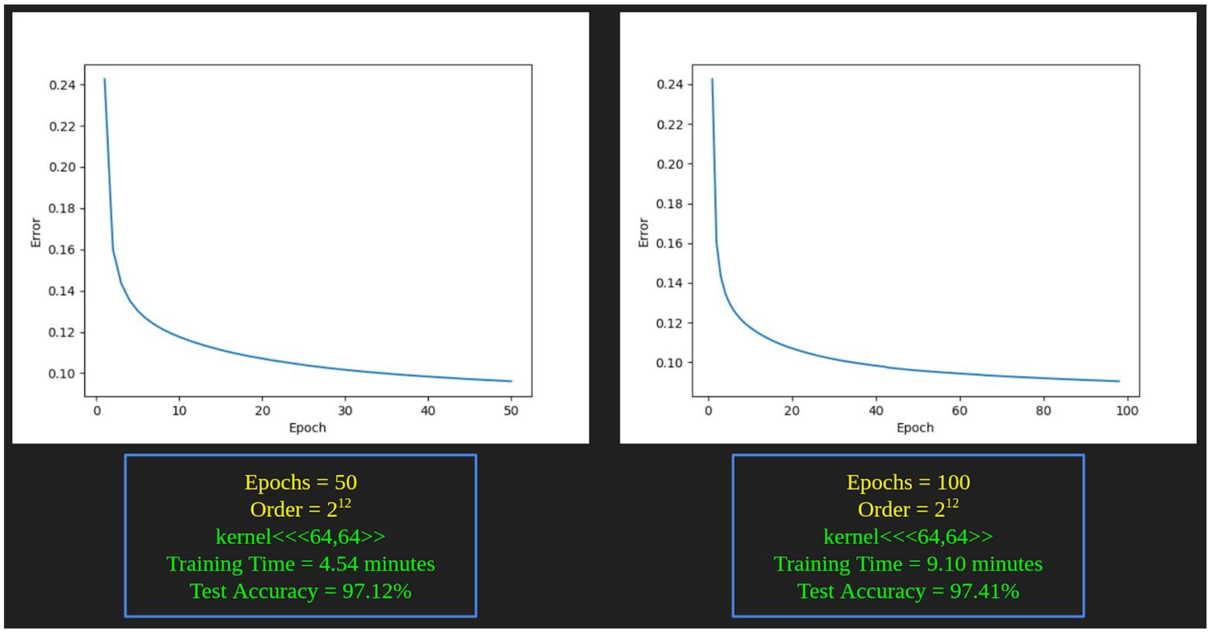


Epochs = 50
Order = 2^{16}
kernel<<<128,512>>
Training Time = 4.86 minutes
Test Accuracy = 87.92%

Epochs = 50
Order = 2^{16}
kernel<<<256,256>>
Training Time = 4.93 minutes
Test Accuracy = 87.92%

Similar Order (product) give similar results

- Higher-order (product, 2^{16}) gives *inaccurate results* compared to those of lower-order (2^{12}) *of 2*.

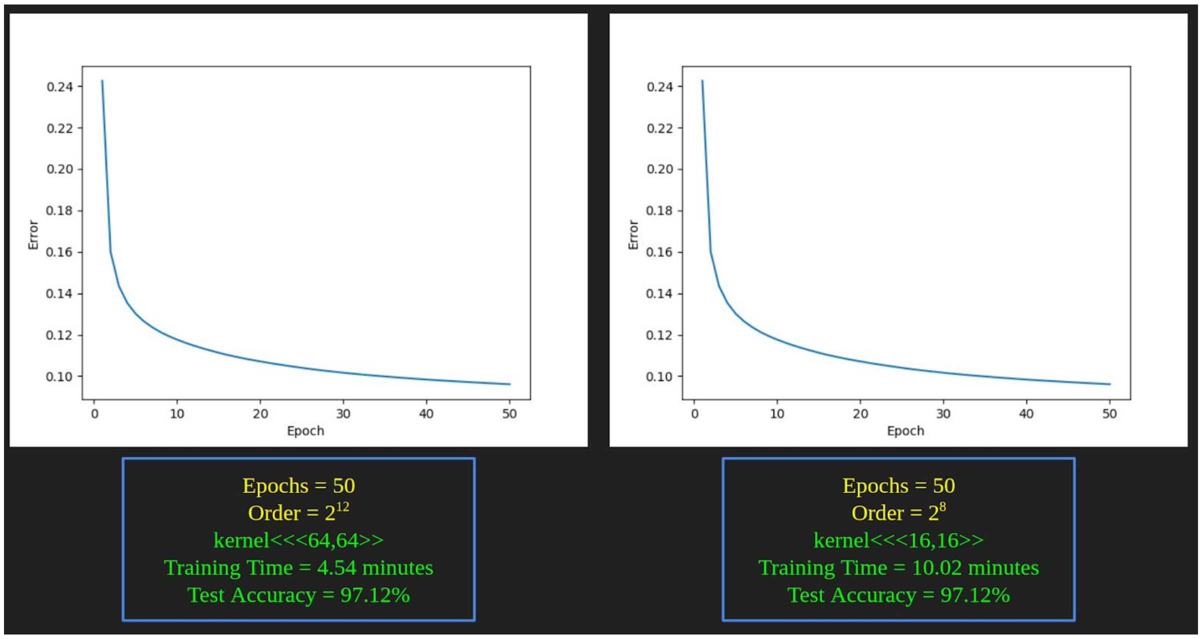


Epochs = 50
Order = 2^{12}
kernel<<<64,64>>
Training Time = 4.54 minutes
Test Accuracy = 97.12%

Epochs = 100
Order = 2^{12}
kernel<<<64,64>>
Training Time = 9.10 minutes
Test Accuracy = 97.41%

The training doesn't improve accuracy a lot after 50 epochs

- As the product's order *becomes smaller*, training time increases even if accuracy remains the same (*2^8 order* v/s *2^{12} order*).



Epochs = 50
Order = 2^{12}
kernel<<<64,64>>
Training Time = 4.54 minutes
Test Accuracy = 97.12%

Epochs = 50
Order = 2^{8}
kernel<<<16,16>>
Training Time = 10.02 minutes
Test Accuracy = 97.12%

Training time increases although the accuracy is the same

6. Conclusion

In this report, I present a simple 3-layer *CNN*[5][6] implemented using *CUDA*[3][4]. We call the resulting implementation ***CuCNN.*** Along with successful training and inference on the *MNIST*[1][2] dataset, I also present some useful insights into ML algorithms' performance on CUDA.

CuCNN by:

Jitesh Jain (Enrolment Number: **19114039**)

7. References

1. MNIST handwritten digit database
2. https://github.com/projectgalateia/mnist#mnist
3. Getting Started with CUDA
4. About CUDA
5. ImageNet Classification with Deep Convolutional Neural Networks
6. https://towardsdatascience.com/an-introduction-to-convolutional-neural-networks-eb0b60b58fd7?gi=31c7f978cb3b
7. Google Colab Introduction
8. cuBLAS Library
9. Neural networks and deep learning
10. CS334 on Udacity

CuDecisionTree

(Cuda Decision Tree)

- Anshika Mittal

---

1. Problem Statement

Implement a new parallelized decision tree algorithm on a CUDA (compute unified device architecture), which is a GPGPU solution provided by NVIDIA.

2. Novelty

In order to improve data processing latency in huge data mining, in this project I design and implement a new parallelized decision tree algorithm on a CUDA. By leveraging the existing CUDA components, such as prefix-sum and parallel sorting, my proposed CuDecisionTree system performs well and gets major performance improvement than sequential decision tree algorithms.

3. Evaluation Parameters/ Dataset used

Some of the evaluation parameters in my model are total cost time, accuracy , and the size of the system.

Data Points are generated randomly in the code.

Input Format is: {num_of_samples,attr_count}

Sample Input: 200000 784

Sample Output : Total nodes in the tree: 63583

4. Methodology

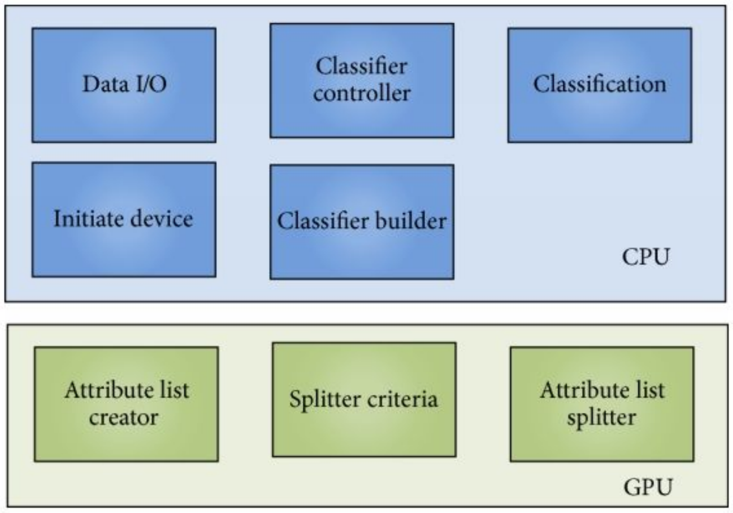


Figure: The CuDecisionTree system components.

The principle of CuDecisionTree is dispatching flow control, handling, and communication tasks to CPU and on the other hand assigning computing intensive jobs to GPU. The blue parts in the figure are running on the CPU while the green parts are running on a GPU.

There are 7 major steps in the CuDecisionTree system.

(1) Training and testing data are loaded to host memory from disks.

(2) Initialization of the device includes query device information, allocation memory space, and copy of training data into the device.

(3) The system sets up some parameters for the user. For instance, the minimum numbers of data of a leaf, the maximum depth of the classifier etc.

(4) Creating attribute lists in device by moving each attribute to corresponding position and then sorting all attribute lists in devices.

(5) This is the most important step of the system where an iterative breadth first scheme is used instead of the recursive model of decision tree building algorithm. Host is in charge of the working flow of the whole system.

(6) Now the classification is performed on the host sequentially.

(7) The results are presented on hosts.

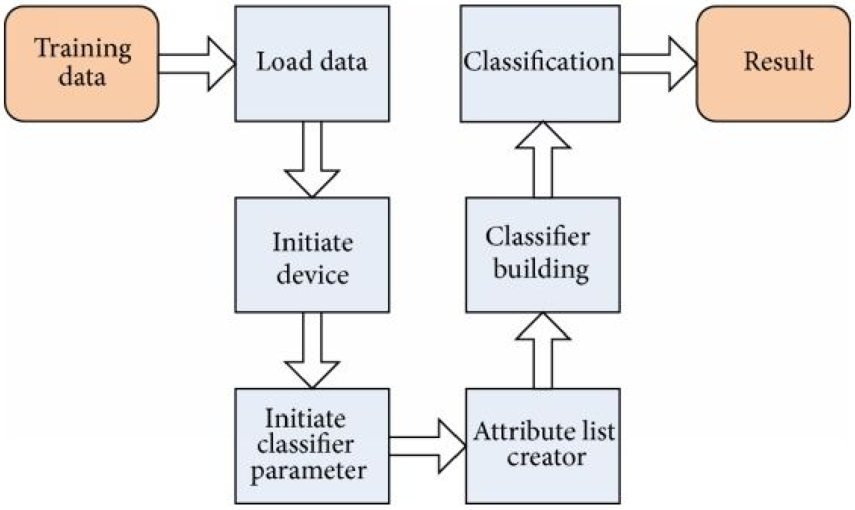


Figure: Flowchart of the proposed CuDecisionTree algorithm.

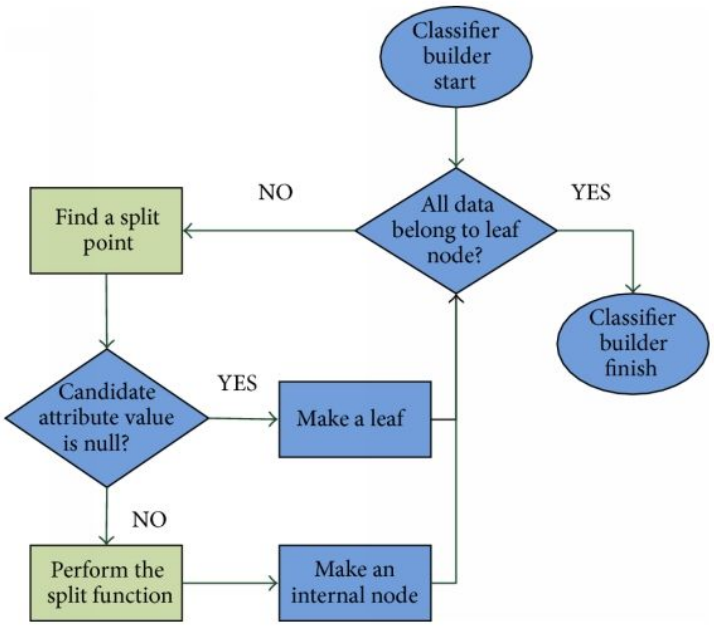
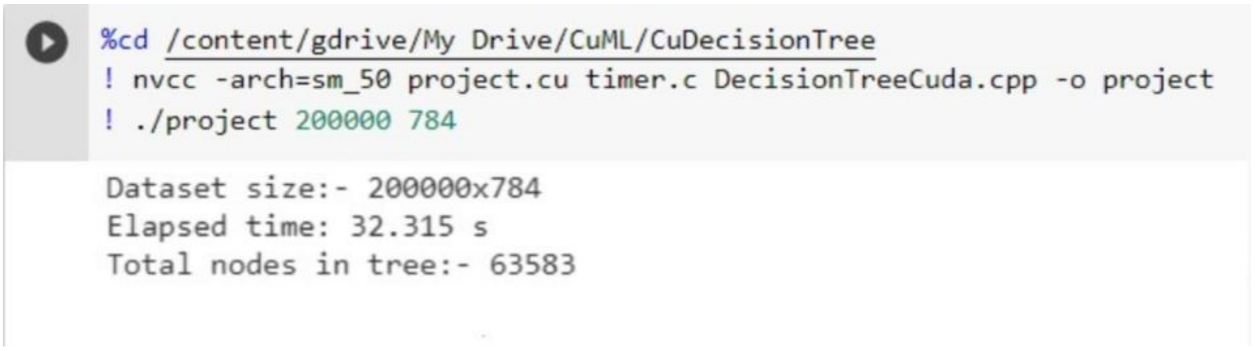

Classifier builder start
All data belong to leaf node?
NO
YES
Find a split point
Classifier builder finish
Candidate attribute value is null?
YES
Make a leaf
NO
Perform the split function
Make an internal node

Figure: Flowchart of the classifier building phase

5. Results



```
%cd /content/gdrive/My Drive/CuML/CuDecisionTree
! nvcc -arch=sm_50 project.cu timer.c DecisionTreeCuda.cpp -o project
! ./project 200000 784
```

```
Dataset size:- 200000x784
Elapsed time: 32.315 s
Total nodes in tree:- 63583
```

6. Conclusion

For solving problems that are able to be solved in parallel and with high density computation,using GPUs normally brings remarkable improvement of performance. Many machine learning algorithms have been developed on CUDA GPUs as they show performance improvement compared to CPU. This project shows that by leveraging the existing CUDA components, such as prefix-sum and parallel sorting, our proposed CuDecisionTree system performs well and gets major performance improvement than sequential decision tree algorithms.

CuDecisionTree by:

Anshika Mittal (Enrolment Number: **19114009**)

7. References

1. Getting Started with CUDA
2. About CUDA
3. Parallel Implementation of Decision Tree Research paper
4. Google Colab Introduction
5. Neural networks and deep learning

https://www.hindawi.com/journals/tswj/2014/745640/

CuKNN

(Cuda k-Nearest Neighbours)

- Anushka Singh

---

1. Problem Statement

Implement KNN algorithm using CUDA.

2.Novelty of work done

The k-nearest neighbours (KNN) algorithm is a simple, supervised machine learning algorithm that can be used to solve both classification and regression problems.
In this project, implementation of KNN algorithm is done for both CPU and GPU using CUDA.
It is observed that we get much better performance on GPU than CPU for this algorithm.

3.Evaluation parameters/ Data Set Used

The input consists of an array comprising of 3-dimensional data elements distributed in

3-dimensional space.It is generated within the program itself.Each element of the array is of the format : (x,y,z) , where x represents the x coordinate , y represents the y coordinate and z represents the z coordinate and here I have taken x = y = z .

Evaluation is done on the basis of the time taken for the algorithm to execute on CPU and GPU.

The output for each element would be its consecutive element , for example ,for element (1,1,1) output would be either (0,0,0) or (2,2,2) since each element in the input array is of the format: (x,y,z) where x=y=z.

4. Methodology

Theory :

KNN is a model that classifies data points based on the points that are most similar to it.
Its advantages are that it is easy to use, has quick calculation time and does not make any assumptions about the data.

However , its disadvantages are that its accuracy depends on the quality of data, and that we must find an optimal value of k.
The kNN algorithm mainly consists of the following steps:

1.Load the data

2.Initialize K to your chosen number of neighbors(here k=1).

3. For each example in the data

3.1 Calculate the distance between the query example and the current example from the data.The most common way to find this distance is the Euclidean distance.

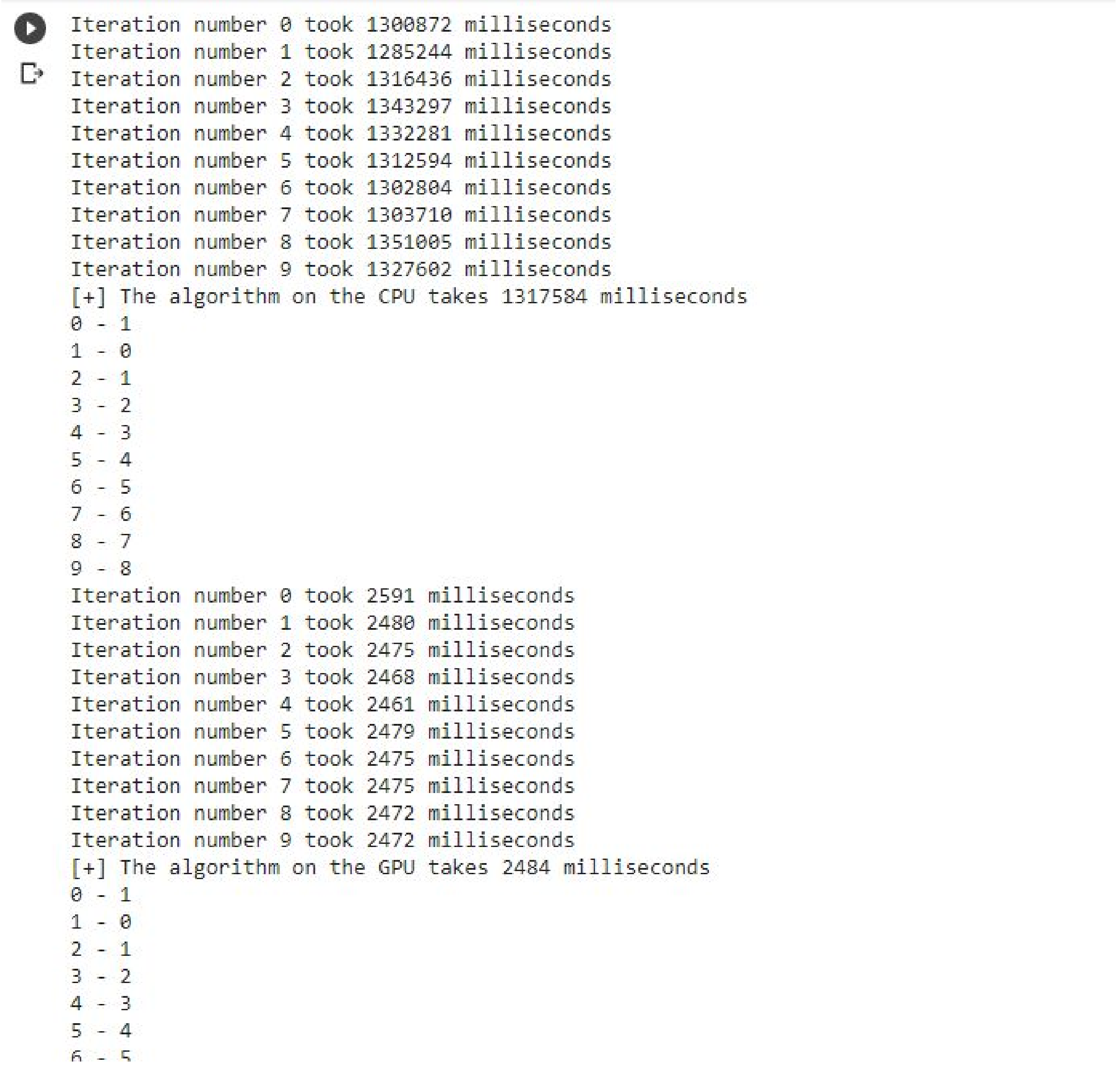
3.2 Add the distance and the index of the example to an ordered collection

4. Sort the ordered collection of distances and indices from smallest to largest (in ascending order) by the distances

5. Pick the first K entries from the sorted collection

6. Get the labels of the selected K entries

5.Results/Screenshots



```
Iteration number 0 took 1300872 milliseconds
Iteration number 1 took 1285244 milliseconds
Iteration number 2 took 1316436 milliseconds
Iteration number 3 took 1343297 milliseconds
Iteration number 4 took 1332281 milliseconds
Iteration number 5 took 1312594 milliseconds
Iteration number 6 took 1302804 milliseconds
Iteration number 7 took 1303710 milliseconds
Iteration number 8 took 1351005 milliseconds
Iteration number 9 took 1327602 milliseconds
[+] The algorithm on the CPU takes 1317584 milliseconds
0 - 1
1 - 0
2 - 1
3 - 2
4 - 3
5 - 4
6 - 5
7 - 6
8 - 7
9 - 8
Iteration number 0 took 2591 milliseconds
Iteration number 1 took 2480 milliseconds
Iteration number 2 took 2475 milliseconds
Iteration number 3 took 2468 milliseconds
Iteration number 4 took 2461 milliseconds
Iteration number 5 took 2479 milliseconds
Iteration number 6 took 2475 milliseconds
Iteration number 7 took 2475 milliseconds
Iteration number 8 took 2472 milliseconds
Iteration number 9 took 2472 milliseconds
[+] The algorithm on the GPU takes 2484 milliseconds
0 - 1
1 - 0
2 - 1
3 - 2
4 - 3
5 - 4
6 - 5
```

6. Conclusions

I observe that GPU implementation for this algorithm is much faster than the CPU implementation.Hence it can be concluded that GPU offers much better performance as it is much more efficient when we need to perform a large number of computations parallely.

CuKNN by:

Anushka Singh (Enrolment Number: **19114011**)

7.References

- For CUDA Programming:

  https://www.olcf.ornl.gov/wp-content/uploads/2013/02/Intro_to_CUDA_C-TS.pdf

- For KNN:

  https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

CuSVD

(Cuda Singular Value Decomposition)

- Anurag Bansal

1. Problem Statement

Implement ML Algorithm SVD(Singular Value Decomposition) and PCA(Principal Component Analysis) using CUDA.

2. Novelty of Work Done

The SVD Algorithm[1] is a technique which enables decomposition of a matrix A into 3 matrices U, Σ , V^T such that:

$$A = U\Sigma V^T$$

Here U, V are orthogonal matrix and Σ is a diagonal matrix such that the diagonal elements are in non increasing order i.e. $\Sigma[i][i] >= \Sigma[j][j]$ for $i > j$. The advantage of this decomposition is that if we imagine the columns of a 2D matrix as features corresponding to certain data represented as rows, then this decomposition helps to prioritize features among them and helps in data reduction by choosing only the most significant features. The data reduction is carried out with PCA (Principal Component Analysis) which based on certain retention_percentage generates a matrix with reduced dimension which contains information equivalent to retention_percentage*A (the original matrix), and thus helps in carrying out faster computation and also enables us to concentrate only on the data features which are useful.

The aim of this work is to make effective use of the computational power of GPU to carry out the decomposition parallelly.

3. Evaluation parameters/ Data Set Used

The input matrix is being generated randomly with the help of a python program. The input is of the format:

$$M \ N$$
$$a[0], a[1], a[2], \ \ldots\ldots\ldots\ldots\ldots, \ a[M * N - 1]$$

where M is the number of rows in the matrix and N is the number of columns in the matrix.
The output contains the matrix U(M* M), $\Sigma(M * N)$, V^T(N*N), A^{hat} (M*K), where A^{hat} is the matrix obtained by doing the Principal Component Analysis on the Singular Value Decomposition with certain *retention_percentage*. Here ((M*M),(M*N),(N*N),(M*K)) are dimensions of (U, Σ ,V^T, A^{hat}) respectively.

4. Methodology

Theory:

1) SVD calculation is carried with the help of Jacobi EigenValue Algorithm, which is a convenient way for obtaining the eigenvalues of symmetric matrices A[2]:

$$A = GDG^T$$

$$GG^T = I$$

where D is a Diagonal Matrix, G is a matrix which is orthogonal. Here D contains the eigenvalues of A and G is a matrix containing the eigenvectors of A as columns.

Combining the equation of SVD and the fact that U is orthogonal i.e.

$$A = U\Sigma V^T$$

$$UU^T = I$$

where I is the Identity Matrix. We Get:

$$A^T A = V\Sigma^2 V^T$$

which is similar to the Jacobi Eigenvalue Problem.

So by incorporating the Jacobi Eigenvalue Algorithm we can obtain the SVD of a Matrix A.

Pseudo Code for jacobi algorithm

```
procedure jacobi($S \in R^{n\times n}$; out $e \in R^n$; out $E \in R^{n\times n}$)

        var
          i, k, l, m, state $\in$ N
          s, c, t, p, y, d, r $\in$ R
          ind $\in N^n$
          changed $\in L^n$

        function maxind(k $\in$ N) $\in$ N ! index of largest off-diagonal element in row k
          m := k+1
          for i := k+2 to n do
            if $|S_{ki}| > |S_{km}|$ then m := i endif
          endfor
          return m
        endfunc

        procedure update(k $\in$ N; t $\in$ R) ! update $e_k$ and its status
          y := $e_k$; $e_k$ := y+t
          if $changed_k$ and ($y=e_k$) then $changed_k$ := false; state := state−1
          elsif (not $changed_k$) and ($y\neq e_k$) then $changed_k$ := true; state := state+1
          endif
        endproc




        procedure rotate(k,l,i,j $\in$ N) ! perform rotation of $S_{ij}$, $S_{kl}$
          ⎡   ⎤   ⎡      ⎤ ⎡    ⎤
          |$S_{kl}$|   |c  −s | |$S_{kl}$ |
    $\begin{bmatrix} \\ S_{ij} \end{bmatrix} := \begin{bmatrix} & \\ s & c \end{bmatrix} \begin{bmatrix} \\ S_{ij} \end{bmatrix}$
  endproc

  ! init e, E, and arrays ind, changed
  E := I; state := n
  for k := 1 to n do $ind_k$ := maxind(k); $e_k$ := $S_{kk}$; $changed_k$ := true endfor
  while state≠0 do ! next rotation
    m := 1 ! find index (k,l) of pivot p
    for k := 2 to n−1 do
      if $|S_{k\ indk}| > |S_{m\ indm}|$ then m := k endif
    endfor
    k := m; l := $ind_m$; p := $S_{kl}$
    ! calculate c = cos φ, s = sin φ
    y := $(e_l - e_k)/2$; d := $|y| + \sqrt{(p^2+y^2)}$
    r := $\sqrt{(p^2+d^2)}$; c := d/r; s := p/r; t := $p^2/d$
    if y<0 then s := −s; t := −t endif
    $S_{kl}$ := 0.0; update(k,−t); update(l,t)
    ! rotate rows and columns k and l
    for i := 1 to k−1 do rotate(i,k,i,l) endfor
    for i := k+1 to l−1 do rotate(k,i,i,l) endfor
    for i := l+1 to n do rotate(k,i,l,i) endfor
    ! rotate eigenvectors
    for i := 1 to n do
      $\begin{bmatrix} E_{ik} \\ \\ E_{il} \end{bmatrix} := \begin{bmatrix} c & -s \\ & \\ s & c \end{bmatrix} \begin{bmatrix} E_{ik} \\ \\ E_{il} \end{bmatrix}$
    endfor
    ! rows k, l have changed, update rows $ind_k$, $ind_l$
    $ind_k$ := maxind(k); $ind_l$ := maxind(l)
  loop
endproc
```

Example:

$$\begin{bmatrix}1&0&0&0&2\\0&0&3&0&0\\0&0&0&0&0\\0&4&0&0&0\end{bmatrix}=\begin{bmatrix}0&0&1&0\\0&1&0&0\\0&0&0&-1\\1&0&0&0\end{bmatrix}\cdot\begin{bmatrix}4&0&0&0&0\\0&3&0&0&0\\0&0&\sqrt{5}&0&0\\0&0&0&0&0\end{bmatrix}\cdot\begin{bmatrix}0&1&0&0&0\\0&0&1&0&0\\\sqrt{0.2}&0&0&0&\sqrt{0.8}\\0&0&0&1&0\\-\sqrt{0.8}&0&0&0&\sqrt{0.2}\end{bmatrix}$$

2) PCA Calculation:

The PCA is calculated by computing W by selecting first k columns of matrix U where k is calculated based on retention_percentage:

$$W = U[:,0:k] \ \# \ Syntax\ in\ Python$$

Then the matrix A^{hat} is calculated by:

$$A^{hat} = AW$$

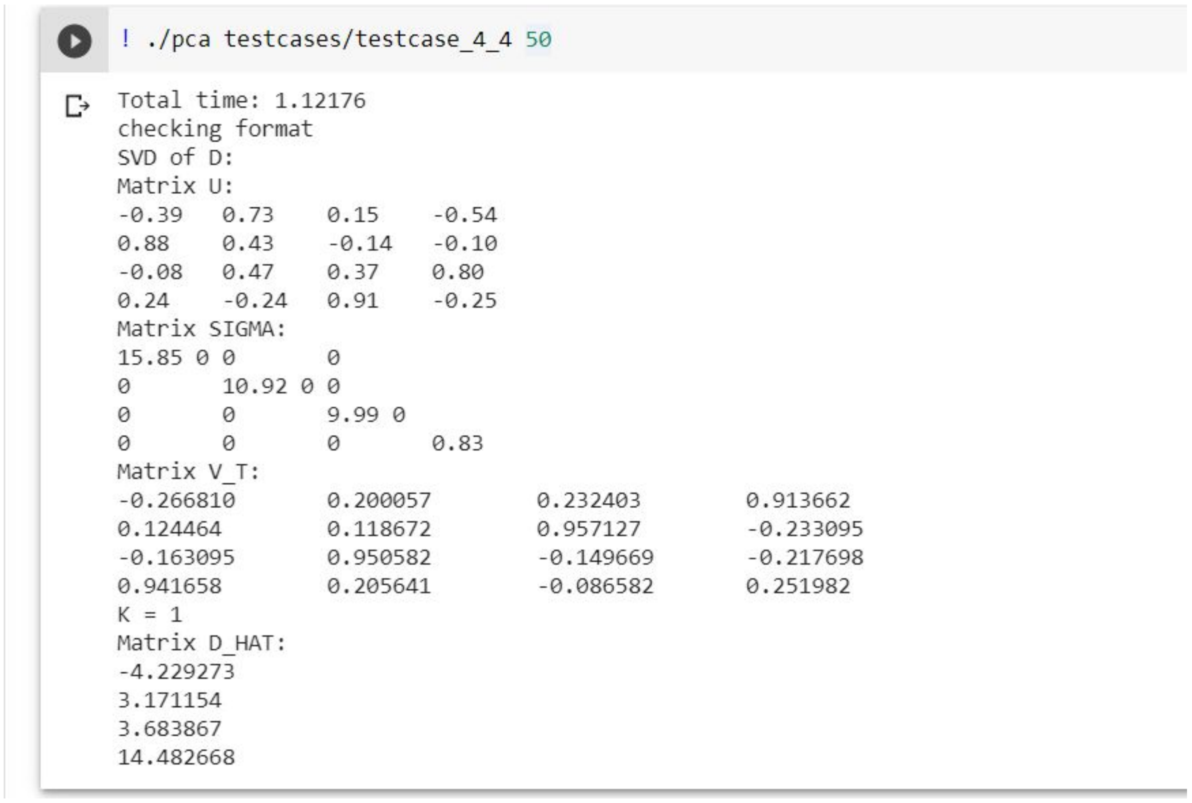
5. Results

Input:

4 4

2 -3 1 -3 1 2 4 9 6 8 4 -3 -8 12 -3 2

Output:



```
! ./pca testcases/testcase_4_4 50
```

```
Total time: 1.12176
checking format
SVD of D:
Matrix U:
-0.39   0.73    0.15    -0.54
0.88    0.43    -0.14   -0.10
-0.08   0.47    0.37    0.80
0.24    -0.24   0.91    -0.25
Matrix SIGMA:
15.85 0 0       0
0       10.92 0 0
0       0       9.99 0
0       0       0       0.83
Matrix V_T:
-0.266810       0.200057        0.232403        0.913662
0.124464        0.118672        0.957127        -0.233095
-0.163095       0.950582        -0.149669       -0.217698
0.941658        0.205641        -0.086582       0.251982
K = 1
Matrix D_HAT:
-4.229273
3.171154
3.683867
14.482668
```

6. Conclusion

GPU computation is much faster as compared to CPU, which is the reason for its popularity among Machine Learning enthusiasts. GPUs help in carrying out extensive computations and enhance the learning ability of ML Algorithms. This project provides an implementation of SVD and PCA, using CUDA which are important algorithms enabling extraction of significant features from data and correspondingly reducing the data thereby enabling faster computation.

CuSVD by:

Anurag Bansal (Enrolment Number: **19114010**)

7. References

[1] SVD - Explanation

https://en.wikipedia.org/wiki/Singular_value_decomposition

[2] WikiPedia - Jacobi Eigenvalue Algo

https://en.wikipedia.org/wiki/Jacobi_eigenvalue_algorithm

[3] CUDA Programming Basics

CS334 on Udacity

CuSVM

(Cuda Support Vector Machine)

- Hardik Thami

---

1. Problem Statement

Implement ML algorithm SVM (Support Vector Machines) in CUDA.

2. Novelty of work done

Support Vector Machines are supervised learning models for classification and regression problems. They can solve linear and non-linear problems and work well for many practical problems [1]. The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space (N — the number of features) that distinctly classifies the data points [2].

Through this project, I plan to implement the SVM in CPU and GPU using CUDA. To implement it, the focus will be to calculate a hyperplane that separates sample data set into classes to find the least parameters. Using those parameters, we will be able to find classes for new data. By implementing in GPU, we can obtain higher performance when compared to CPU implementation.

3. Evaluation parameters / Data set used

To make the implementation and testing faster, the size of the input and prediction data set has been limited. Throughout the implementation the following data set was used:

- Sample Training Data Points: {(1, 7), (2, 8), (3, 8), (5, 1), (7, 3), (6, -1)}
- Sample Output class: {-1, -1, -1, 1, 1, 1}
- Sample data to be predicted: {(0, 10), (1, 3), (3, 4), (3, 5), (5, 5), (5, 6), (6, -5), (5, 8)}

While adding more data points to the training set will help, to make it easier to understand and implement the data set was not expanded.

4. Methodology

Theory

The idea of SVM is to implement a hyper plane that lies at maximum distance from all data points. The distance of a data point from hyperplane is the vector projection. This projection needs to maximize for all data points. This is calculated as parameter, w for the model. However, some data points may fall on the other side of the hyperplane even if it belongs to other class. To adjust this, we need to add another factor called bias, b. The best model will have low w and low b which means that all data points have large margin from hyper plane and lowest bias to include all data points. Hyper plane is defined as:

$$y\,(x \cdot w + b) - 1 = 0$$

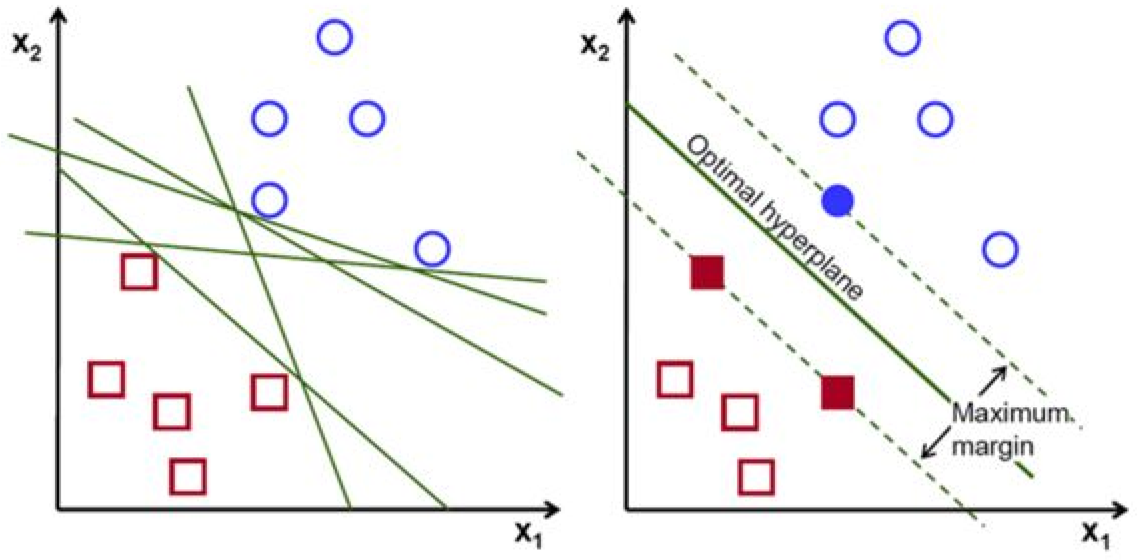
Where y - Class of a data point

x - Input data point (features)

w - Weights

b – Bias

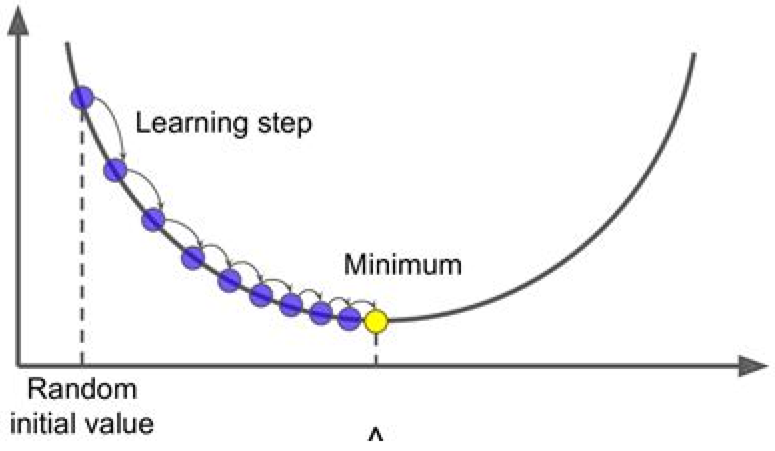
There are a lot of possible hyperplanes that satisfy the above equation as shown below. Out of all these hyperplanes, one of them will have the least magnitude for w and b.



Possible hyperplanes

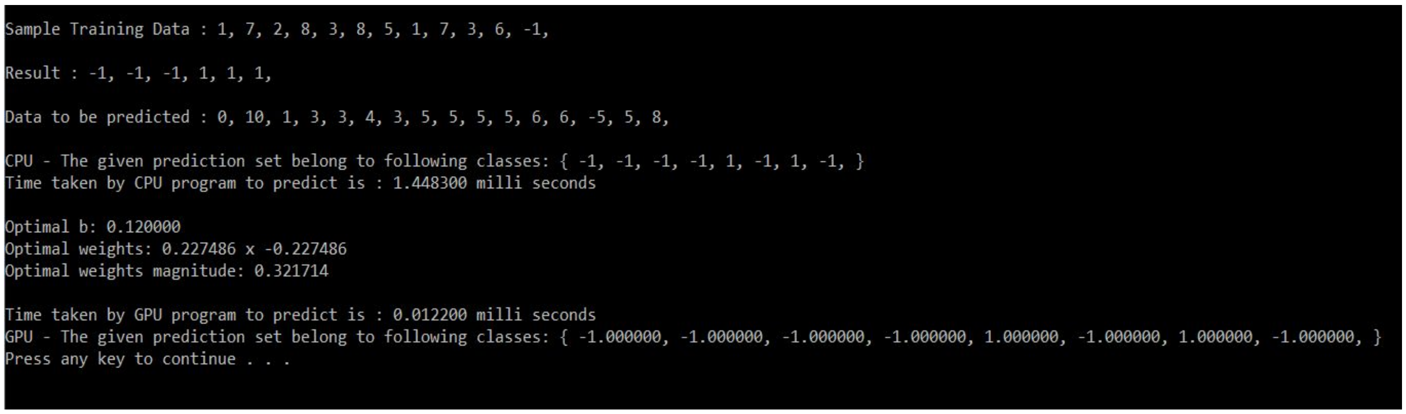
The idea for finding optimal w and b is an optimization problem which is implemented through this project in both CPU and GPU.

The idea of optimization is as shown below where x axis shows difference of $y\,(x \cdot w + b) - 1$ from 0 and y axis shows magnitude of w. As shown below the first jump will be using one learning step value and then step value will be decreased as it gets closer and closer to the median value.



Once the value is calculated we can classify other data points using the value of $y\,(x \cdot w + b) - 1$. If this is greater than 0 then it belongs to class 1. Otherwise it belongs to class -1.

5. Results



```
Sample Training Data : 1, 7, 2, 8, 3, 8, 5, 1, 7, 3, 6, -1,

Result : -1, -1, -1, 1, 1, 1,

Data to be predicted : 0, 10, 1, 3, 3, 4, 3, 5, 5, 5, 5, 6, 6, -5, 5, 8,

CPU - The given prediction set belong to following classes: { -1, -1, -1, -1, 1, -1, 1, -1, }
Time taken by CPU program to predict is : 1.448300 milli seconds

Optimal b: 0.120000
Optimal weights: 0.227486 x -0.227486
Optimal weights magnitude: 0.321714

Time taken by GPU program to predict is : 0.012200 milli seconds
GPU - The given prediction set belong to following classes: { -1.000000, -1.000000, -1.000000, -1.000000, 1.000000, -1.000000, 1.000000, -1.000000, }
Press any key to continue . . .
```

6. Conclusions

GPU implementation is so much faster than CPU. GPU completes simulation in micro seconds while CPU completes in milliseconds. This project also gave an overview of how fast this algorithm works on GPU and how it can be used for regression as well as multi class classification problems.

CuSVM by:

Hardik Thami (Enrolment Number: **19114035**)

7. References

[1] Medium - https://medium.com/@LSchultebraucks/introduction-to-support-vector-machines-9f8161ae2fcb

[2] towardsdatascience.com -https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47