Accelerating Classification Models on GPUs with CUDA

EE 451 Final Project

Anisha Palaparthi and Sriya

Introduction

The classification task in Machine Learning involves predicting a class label from data, placing each data example into a class, or category, predicted by an ML algorithm. There are several applications of classification, including but not limited to image recognition, fraud detection, and medical diagnoses [src]. These examples highlight how classification is a very important task in Machine Learning as it has many motivating real-world applications. In this project we primarily focus on binary classification for multiple datasets, predicting either a 0 or a 1 for the data label to distinguish between the classes. We additionally explore multiclass classification on the real-world MNIST dataset, which sorts a handwritten digit input into one of ten categories - the numbers 0 through 9.

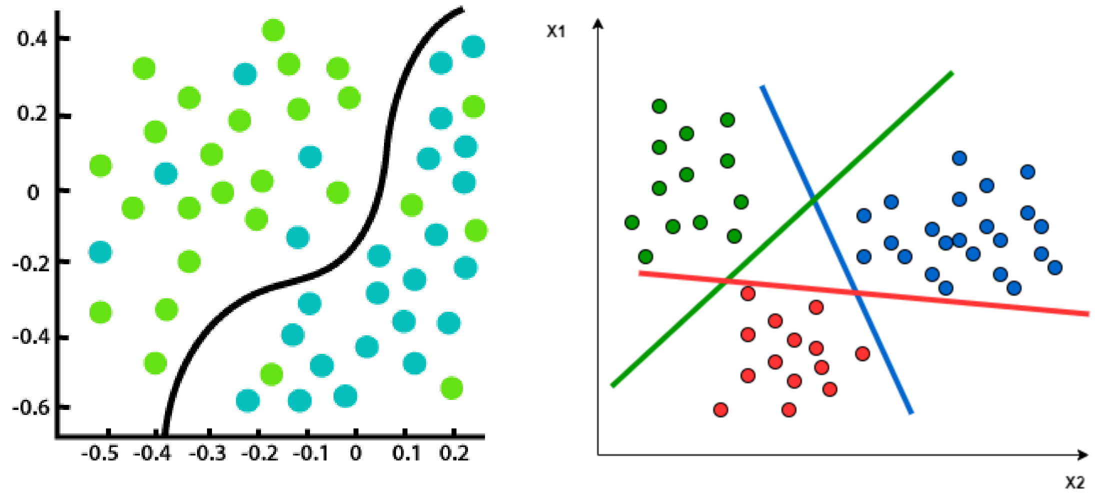


Fig. 0 - Visualizations of Binary (left) and Multi-Class (right) Classification [1] [2]

In specific, we will explore several aspects of classification:

- Implement a serial version of training Support Vector Machines, Logistic Regression, and Multi-Layer Perceptions as classification models.
- Develop and implement a parallelized version of training Support Vector Machines, Logistic Regression, and Multi-Layer Perceptions using CUDA to leverage GPU hardware.
- Compare execution times between the CPU-based and GPU-based implementations of the three algorithms to identify which models are best suited for GPU parallelization
- Implement a multi-class classification SVM model using serial training

- Develop and implement a CUDA implementation for training a multi-class classification SVM model.
- Evaluate all models on various datasets to achieve similar accuracy between CPU and GPU implementations and to evaluate speedup
- Discuss the scalability of using GPUs to accelerate the three models, particularly with respect to model size.

Objectives

ML Algorithms for Classification

Support Vector Machine

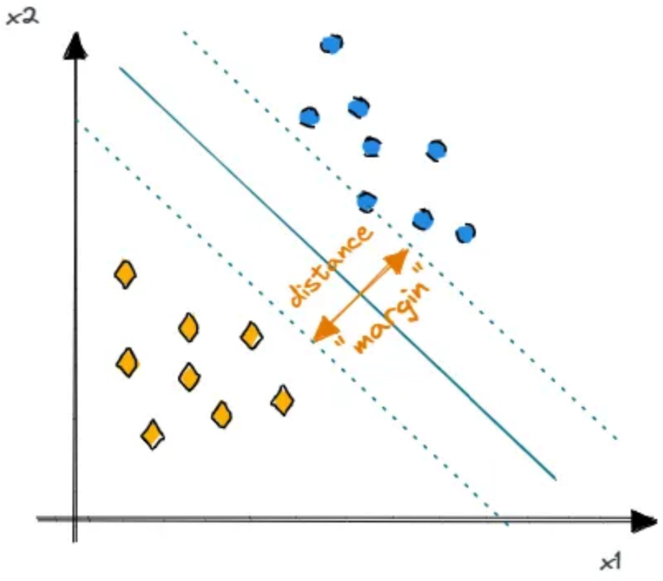


Fig 1: Visualizing margin calculations with SVM [3]

We begin by examining the effect of parallelization on the performance and execution speed of Support Vector Machine, or SVM, on the classification task. SVM is a highly performant “off-the-shelf” supervised learning algorithm that approaches classification with the objective of generating a maximal margin separating hyperplane. In other words, the Linear SVM aims to calculate a linear decision boundary in the space of input features that results in the greatest margin, a measure of distance from the closest data points in either class to the generated decision boundary. This results results in the objective function described below:

$$\mathcal{J}(w,b) = \lambda\frac{1}{2}||w||^2 + \frac{1}{n}\sum_{i=1}^{n} max(0, 1 - y_i(w \cdot x_i + b))$$

Fig 2: SVM Objective Function (SVM Medium)

Algorithm 1 describes the high-level pseudocode of the Support Vector Machine technique, with algorithm parameters eta (learning rate) and lambda. Note that using SVM requires us to predict either -1 or 1 as class labels rather than 0 or 1, denoted by the use of cls[i] in the pseudocode below.

```
SupportVectorMachine(x, y, eta) {
      initialize weights w, b = 0
      for (i = 0...numIterations) {
            for (j = 0...numSamples) {
                  linearModel = dotProduct(x, w) + b
                  if (linearModel * cls[j] >= 1):
                        w -= eta * w * lambda
                  else:
                        w -= eta * (lambda * w - (cls[j] * x))
            }
      }
}
```

Algorithm 1: Support Vector Machine

The primary operations in this pseudocode include vector dot product and vector addition.

Logistic Regression

We additionally implement Binary Logistic Regression [4], a popular algorithm for binary classification of categorical variables. In contrast with standard linear regression techniques, logistic regression is a logit model, predicting a value between 0 and 1 representing the probability that an input example resides in a particular class. This technique first applies learned weights and bias to generate a linear combination of the input features. It then applies the sigmoid function to this linear combination to generate the probability of the positive output class conditioned on the input features. This equation is defined below for an output class y, an i-dimensional input feature vector **x** = [x1,..,xi], and weights **w** (including a bias term within **w**):

$$P(y = 1|x_1, ..., x_i) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}}$$

Fig 3: The Probabilistic Form of Logistic Regression, a sigmoid function applied with an input **w**^{T}**x** [Medium Logistic]

To train these weight and bias terms, we leverage the Gradient Descent algorithm (explained in detail in the Context section) on our cost function, the negative log likelihood. This is derived by taking the logarithm of the maximum likelihood, or the probability of observing our data given the values of our model parameters, **w**. This log likelihood equation simplifies to the following for $P = \frac{1}{1+e^{-w.x}}$:

$$\sum y_j log P_j + (1 - y_j) log[1 - P_j]$$

Fig 4: The Log Likelihood equation, the negative of our cost function [Medium Logistic]

Deriving the gradient of this cost function and plugging it into the gradient descent equation $w = w - \eta J(w)$ for our cost function $J(w)$ yields the following update rule calculated at each iteration of our algorithm:

$$\frac{dJ}{dw_i} = \sum_j [P_j - y_j] x_{ij}$$

Fig 5: The gradient of our cost function

The pseudocode we leverage for Logistic Regression is detailed in Algorithm 2.

```
LogisticRegression(X, y) {
      Initialize weights w <- Normal(0, 1)
      Initialize gradients gradW <- 0
      for (i = 0...numIterations) {
            yHat <- weightedSigmoid(X)
            for (j = 0...numSamples) {
                  for (k = 0...numFeatures) {
                        gradW[k] += (yHat[j] - y[j]) * X[j, k]
                  }
            }
      w <- w - eta * gradW
    }
}
WeightedSigmoid(X) {
      return 1 / (1 + exp(-1 * X @ w))
}
```

Algorithm 2: Logistic Regression

As indicated above, the primary operations are matrix-vector multiplication and vector transformation (leveraged in WeightedSigmoid) as well as vector dot product (used in gradient calculation).

Multi-Layer Perceptron

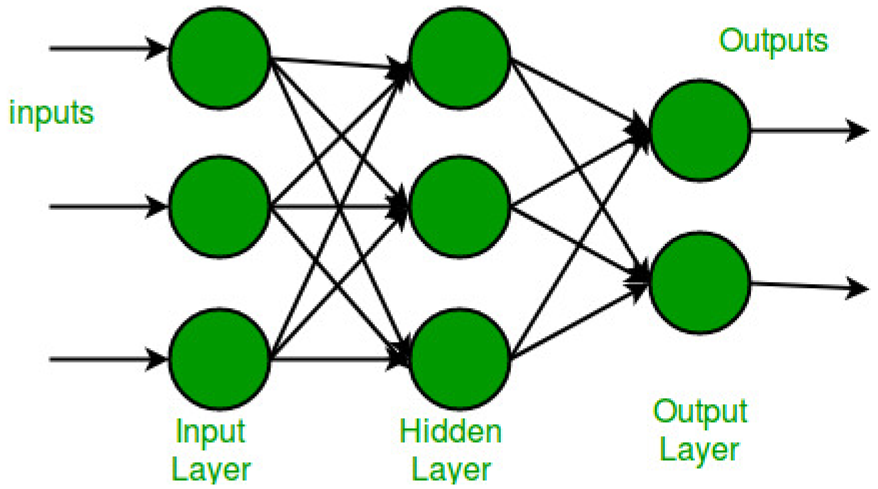


Fig 6: A visualization of a 2-layer MLP with 3 input nodes, 3 hidden nodes, and 2 output classes [5]

Finally, we evaluate the impact of GPU acceleration on a Multi-Layer Perceptron model [6], implementing a 2-layer neural network with sigmoid activation to predict class labels. This is the most computationally intensive of our three models, involving two key steps: forward propagation through the network to generate output probabilities and backward propagation to apply the gradient descent update rule to our model parameters.

Forward Propagation

Taking a matrix of input examples as input, we begin by passing the data through a linear layer, applying learned weights to generate a linear combination of features for each data example. We then apply a non-linearity, the sigmoid activation function (described in the Logistic Regression section above). This “hidden layer” generates outputs that we pass into our output layer, which applies another set of learned weights. This output layer reduces the dimensionality of our data to the number of output classes (2 in the case of binary classification). Finally, we pass these outputs through a sigmoid function to produce class probabilities. In short, the following equation describes the functions applied to our input data:

```
output = Sigmoid(Linear_output(Sigmoid(Linear_hidden(input))))
```

Backward Propagation

After each training iteration, we use the output probabilities predicted in forward propagation to update each of our weights (or model parameters) by propagating gradients backwards through the MLP. We calculate these gradients based on the following scaled L2 loss function, applying the gradient descent update rule with learning rate eta to minimize its value:

$$\frac{1}{2}\sum_{k=1}^{K}(d_j(k) - y_j(k))^2$$

Fig 7: Our MLP Loss Function [6]

We treat the number of training iterations, eta, and the size of our hidden layer as hyperparameters, which we then tuned to maximize performance on each dataset. We initially set the number of nodes in the hidden layer as ⅔(numFeatures + numClasses), learning rate at 0.001, and number of iterations at 100.

The high-level pseudocode of the MLP approach is detailed in Algorithm 3.

```
MLP(X, y) {
      # forward propagation
      for (epoch = 0...numEpochs) {
            outputLayer1 = X @ wHidden + bHidden
            outputLayer1 = Sigmoid(outputLayer1)

            outputLayer2 = outputLayer1 @ wOutput + bOutput
            output = Sigmoid(outputLayer2)

            prediction = argmax(outputLayer2)
      }
      # backpropagation
      deltaOutput = output - outputLayer2
      deltaOutput = -1 * (deltaOutput - SigmoidDerivative(outputLayer2))
      wOutput -= eta * deltaOutput * outputLayer1 (appropriately indexed)
      bOutput -= eta * deltaOutput (appropriately indexed)

      deltaHidden = wOutput @ deltaOutput * SigmoidDerivative(outputLayer1)
      wHidden -= eta * deltaHidden * X (appropriately indexed)
      bHidden -= eta * deltaHidden (appropriately indexed)
}
```

Algorithm 3: 2-Layer Multi-layer Perceptron

Context

An important point is that the number of model parameters in all three models scales with the number of input features in each dataset. For SVM and Logistic Regression, the size of the weights and biases we employ must match the dimensionality of our input, resulting in linear scaling with feature size. In our MLP, we also perform this scaling of hidden layer weight sizes with the number of features; however, this model requires additional computation complexity to accommodate higher-dimensional datasets. Based on our hyperparameter tuning, we realized that we additionally needed to scale the number of hidden nodes used in our model with feature size in order to achieve comparable performance. The calculation we use to initially set the hidden layer size is detailed above in the Multi-Layer perceptron section. The biasHidden (layer 1) vector's size is the number of nodes in the hidden layer (which is a scaled transformation of the feature size), the weightHidden (layer 1) matrix is of size featureSize x numHidden, and the weightOutput (layer 2) matrix is of size numHidden x numClasses. Thus, the size of each of the models is determined by the feature size, so we can estimate model size using feature size (which we will use during our discussion of scalability below).

Additionally, we tackle the multiclass problem using a one-vs-all approach. In this approach, we train n binary classification models for a dataset with n output classes. For a class i in {0,...,n-1}, these models each predict the probability that a data example does or does not belong in class i. We then take the class with the maximal class confidence as our final prediction. We chose this approach over the alternative one-vs-one strategy where each class is compared to every other class, as that would involve training n^2 rather than n classifiers with little to no boost in performance.

Parallelization Strategy

For each of the 3 classification algorithms mentioned above, we provide parallelization techniques to leverage GPU hardware during training using CUDA. The general approach taken for each of the algorithm is to define CUDA kernel functions that update a particular element of the weights and biases that parameterize the model; however, the parallelization strategy used for each models differs significantly in order to realize maximum parallelization, which we will explain in further detail below. While the serial implementations for the models were written based on prior work on classification, the parallel algorithms as explained below were developed independently without any reference to existing material.

Parallelization Strategy for Support Vector Machine

As mentioned in the Algorithms section, the fundamental operations for SVM are vector dot product and vector addition/subtraction. However, we notice that in each of these operations, every individual vector element of the weight vector is computed independently; furthermore, there is no data dependency between elements of the vector for gradient calculations. Thus, we can avoid having to transfer large weight vectors between the CPU and GPU for every vector operation, and we instead define our kernel function to encompass the entire training for loop, placing synchronization commands upon completion of the dot product calculation and for each iteration to maintain correctness of the training. As follows, the number of threads is the feature size of the data.

We also make use of shared memory in order to limit latency of global memory accesses. In particular, we define 2 vectors (the size of which are dependent on the number of features of a dataset) – the weight vector and an input sample vector to store the current sample being using to update gradients – as well as 2 values for constraint and bias that is shared across all threads.
Each thread iterates for a set number of epochs and sees all the samples in the training data in each epoch (the same as the serial version). Within each iteration, the thread first loads its corresponding feature into shared memory, where it then computes the product between this feature and its corresponding weight value. Next, the kernel computes the sum of these products and computes the final result in thread 0. Thread 0 then calculates whether the constraint (as defined above) has been satisfied and updates the bias. A barrier after this computation ensures that all threads wait until the constraint and bias has been calculated before proceeding to the gradient update stage, where each thread updates its weight value and synchronizes before beginning the next iteration.

Parallelization Strategy for Logistic Regression

Recall from the Algorithms section, we illustrate the 3 fundamental operations for the logistic regression classification model: matrix-vector multiplication, vector transformations (for sigmoid calculations and weight updates), and vector dot products (for gradient calculations).

Since these operations involve matrices of various shapes, we cannot employ the same singular kernel framework as used for the SVM. Instead, we must split the parallelization into two kernel functions. The first kernel executes the weighted sigmoid calculations and the second kernel updates the weight vector, as described in further detail below. (Note here that these kernels do not contain many repeated accesses to the same data, so we do not invest into moving data to shared memory).

The first kernel employs *numSamples* number of threads and executes the matrix-vector multiplication between the input matrix of training data and weight vector and the subsequent sigmoid calculation. Each thread is responsible for updating the probability that its corresponding data sample belongs to class 1; to do so, the kernel computes the dot product between its row of the input data matrix and the weight matrix (the result of the dot products across all the threads is the product vector from the matrix-vector multiplication). It then updates its corresponding probability by computing the sigmoid of the dot product and loading it into the global vector.

The second kernel employs *featureSize* number of threads and is responsible for computing the gradients as well as updating its corresponding element of the weight vector. To calculate its corresponding gradient, each thread computes the vector subtraction between the probability and output vectors, and computes the dot product of the difference with the input sample. This dot product estimates the gradient, which is used to update the thread's corresponding weight element.

Parallelization Strategy for Multi-Layer Perceptron

The Multi-Layer Perceptron neural network involves the greatest number of computations out of all three models, and can be broken into two sections for each sample seen in an epoch: Forward Propagation and Backward Propagation. Forward Propagation includes 2 matrix vector multiplications and vector additions to compute the Layer 1 output as sigmoid(x^T * weightHidden + biasHidden) and Layer 2 output as sigmoid(outputL1^T * weightOutput + biasOutput). Backward propagation, as described in the Algorithms section, requires a much more involved approach to calculate gradients and update the weight matrices and bias values, including element-wise Hadamard products, vector transformations using the sigmoid derivative, and broadcasted matrix/vector subtraction.

Because of the several immediate vectors of large size (due to the high number of features and large amount of hidden layers) required for computation, we also utilize shared memory to store these vectors to reduce latency for global memory accesses.

We employ a kernel function that employs *numHiddenLayers* threads. To calculate the layer 1 output during forward propagation, each thread is responsible for one element for the output vector through a dot product between the data sample and a column of the weightHidden matrix, incrementing by the biasHidden value, and compute the sigmoid (which amounts to a matrix-vector multiplication and vector addition across all threads). For calculating the layer 2 output during forward propagation, only *numOutputLayers* threads are active to conduct a similar series of operations using the weightOutput matrix and biasOutput vector.

Then, for the first stage of Backward Propagation, the kernel also activates *numOutputLayers* threads. Each of the threads is responsible for updating a column of the weightOutput matrix and an element of the biasOutput vector through a series of vector transformations and dot products that compute the Layer 2 gradients. The final stage of Backward Propagation again activates the total *numHiddenLayers* number of threads. Each thread is responsible for updating a column of the weightHidden matrix and an element of the biasHidden vector by computing one element of the Hadamard product and conducting similar vector transformations and dot products that compute the Layer 1 gradients.

Parallelization Strategy for Multi-Class Classification

As we will further discuss in the "Experimental Setup" section, we will explore how parallelization impacts the execution time for the multi-class classification problem on the MNIST dataset (specifically for the Support Vector Machine). We will employ the one-vs-all method, meaning we will train 10 SVM classification models – one for each of the digits 0-9. As a result, the parallelization strategy utilizes the same kernel as described for the SVM above; however, instead of 1 kernel, we will launch 10 streams, each of which is responsible for training one of the classifiers asynchronously.

Hypothesis

We hypothesize that all three implementations – logistic regression, SVMs, and MLPs – will have increased performance (ie faster execution time) when trained on the GPU by exploiting parallel floating operations.

We also hypothesize that the speedup will increase when feature size is larger and that MLPs will see better speedup due to the large number of computations required during training compared to the other two models.

Experimental Setup

The platform used to run the serial implementations was my personal computer, which is an HP Spectre that has an Intel Core i7-1070H Processor with 6 cores. The platform used to run the CUDA implementations was an NVIDIA Tesla V100 with 32 GB RAM and PCIe (USC's HPC Cluster Discovery).

Datasets

We have selected the following datasets that best showcase the comparison between the serial and parallel execution strategies for the various classification models.

First, we select 3 types of datasets to verify how all three of the presented classification perform with respect to correctness (accuracy) and performance (execution time) in both serial and parallelized GPU contexts. The first dataset is "Blobs-2D" (created from the "Blobs" dataset provided by Scikit-Learn to generate points to 2D space, 2 clusters, and a standard deviation of 1.05 for each cluster), as visualized below [7]. Blobs-2D represents a dataset with small feature size and relatively easier to model, and the classification task is to decide which cluster a given datapoint belongs to (we delve further into "Blobs" below). The second dataset is the "Titanic" dataset from Kaggle [8], which provides information about passengers on board the Titanic, and the classification task is to predict whether a given passenger survived on the Titanic or not. After preprocessing data to account for missing and unnormalized values, this dataset has 8 features, a mix of integers and floating point values. The third dataset we select is the MNIST Dataset of handwritten digits [9], where each data point is a 784-dimensional vector that contains a value 0-255 for each of the 784 pixels in the image, and the classification task is to determine what digit a given image corresponds to. This dataset presents an opportunity for multi-class classification as well as for a larger feature size with a more complicated decision process.

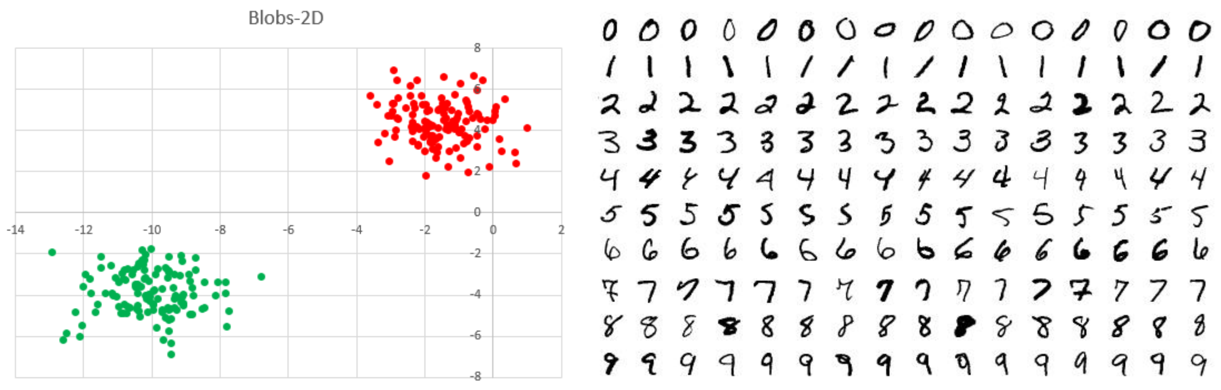


Figure 7 - Visualization of the Blobs-2D Dataset (left) and MNIST Images [10] (Right)

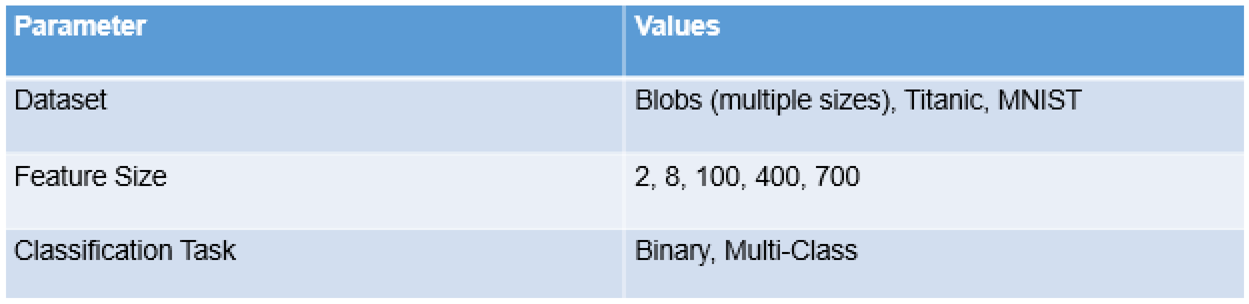
While the above three datasets allow us to benchmark the three models and the serial and CUDA implementations on common classification tasks, we also seek to investigate the scalability of using CUDA to parallelize these models on the GPU. As a result, we select “Blobs” and vary the feature size to explore this aspect of the classification models. In particular, we investigate 4 datasets which we will call: “Blobs-2D”, “Blobs-100D”, “Blobs-400”, and “Blobs-700D.” Each of the 4 datasets contain data samples originating from 2 clusters with feature sizes of 2, 100, 400, and 700 respectively, which will allow us to compare how these classification models scale with respect to the input size.

Parameters

In our experiments, we aim to compare the execution time between serial CPU-based implementations and the CUDA GPU-based implementations of all three algorithms. To evaluate our implementations, we will use two metrics. The first and main metric is execution time, which will allow us to analyze how the parallelization strategies implemented on the GPU will impact the model’s ability to perform computations during training. The second metric we use is accuracy, to ensure correctness of both the serial and CUDA implementations for each of the models; the idea behind the accuracy metric is that both serial and CUDA implementations should be close to each other for each model, which would indicate that the computations being done are the same in order to achieve the same level of accuracy.

There are three parameters that will affect execution time that we wish to explore. The first parameter varied in the experiments is the class of dataset which, as we explained above in the “Datasets” section, varies the type of training data that the model receives during computation. The second parameter is the number of features. The number of features directly impacts the size of the weight and bias vectors/matrices that parameterize the models; thus, we can use feature size as measure of the “size” of the model, and we wish to see how the speedup between the serial and CUDA implementations changes with respect to the model size. The third, and final, parameter is the type of classification – specifically binary and multi-class classification. We explore multi-class classification in the context of the Support Vector Machine, and wish to explore how much speedup the CUDA implementation on the GPU can achieve over the serial SVM in the multi-class classification task.

Other factors that can influence the performance and execution time are model specific parameters (such as lambda, learning rate, number of epochs, etc.). We maintain a constant value for these model-specific parameters across all experiments.



Parameter	Values
Dataset	Blobs (multiple sizes), Titanic, MNIST
Feature Size	2, 8, 100, 400, 700
Classification Task	Binary, Multi-Class

Figure 8 - Parameters for Experimentation

Results and Analysis

Comparison of Serial and Parallelized Training for 3 Models of Binary Classification

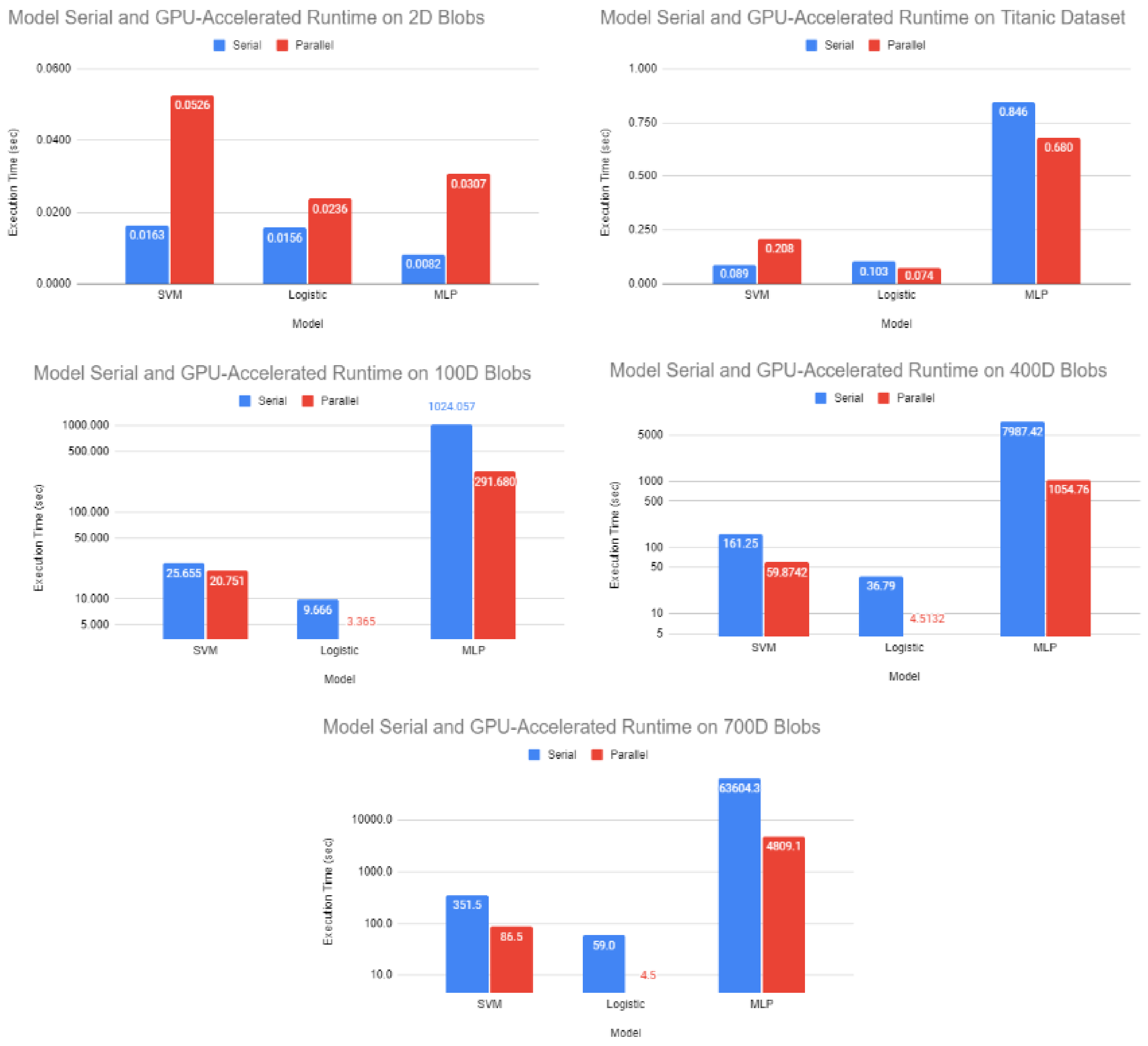


Figure 9 - Comparison of Serial and Parallel Execution Times

The graphs above in Figure 9 showcase how CUDA can harness the GPU to achieve faster training times than the serial implementations. Each chart showcases the training time for the serial execution on the CPU and the parallel execution on the GPU for each of the three models for the various datasets. Note that the GPU-Accelerated execution times include both the time taken to copy data from the CPU to GPU and GPU to CPU as well as for the computation itself.

We notice that all the models when accelerated perform worse on the Blobs-2D dataset than their serial counterparts. This is because the Blobs-2D dataset has a very small number of features, which means parallelizing computation does not provide much speedup; however, the GPU based implementations incur substantial cost for moving data between the CPU and GPU. Thus, for simple datasets like Blobs-2D, it is not as helpful to accelerate training on the GPU.

One aspect to notice is that the difference between serial and parallel execution times is lower for SVM than for the other two models; and even in the case of the Titanic dataset, it is interesting that the parallel execution time is actually slower than the serial execution time. This is likely also caused by the communication cost of moving data between the CPU and GPU outweighing the speedup gained from parallelizing the gradient descent computations. We suspect this is because the main operation for SVMs is vector dot product compared to more complex matrix operations in logistic regression and MLPs, which allow for greater speedup that outweighs their communication costs.

Impact of Scaling Feature Size

The three figures below show the scalability of GPU-Acceleration as a parallelization strategy for each of the three models on binary classification. The speedup is calculated as (Serial Execution Time) / (Parallel Execution Time)

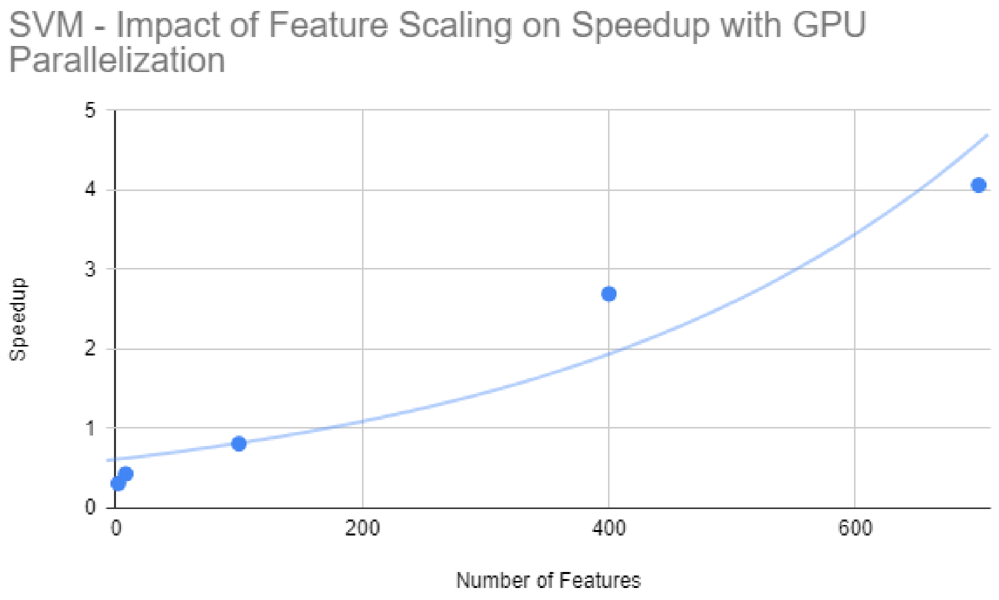


Figure 10 - Feature Scaling for Support Vector Machine

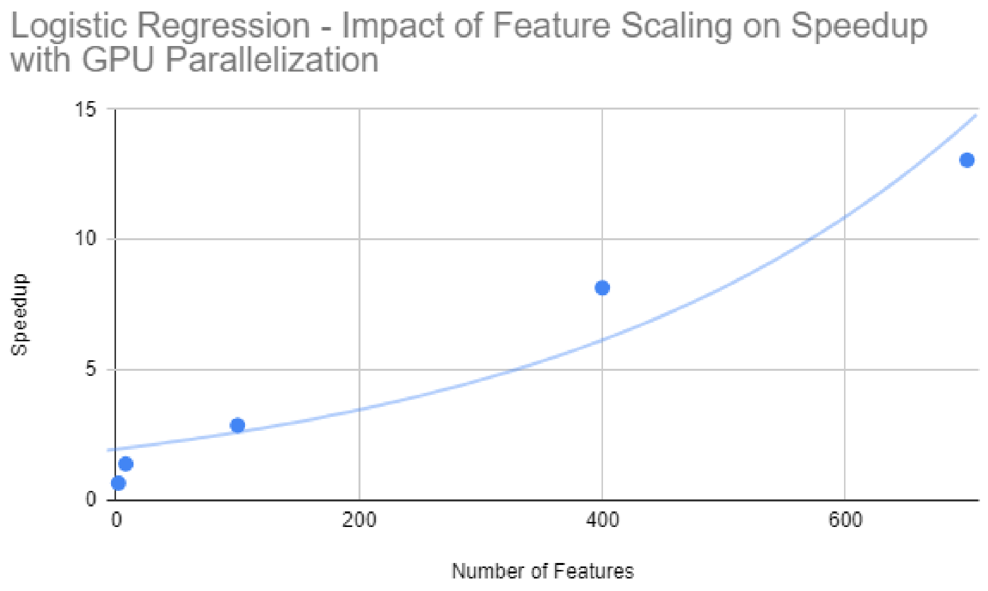


Figure 11 - Feature Scaling for Logistic Regression

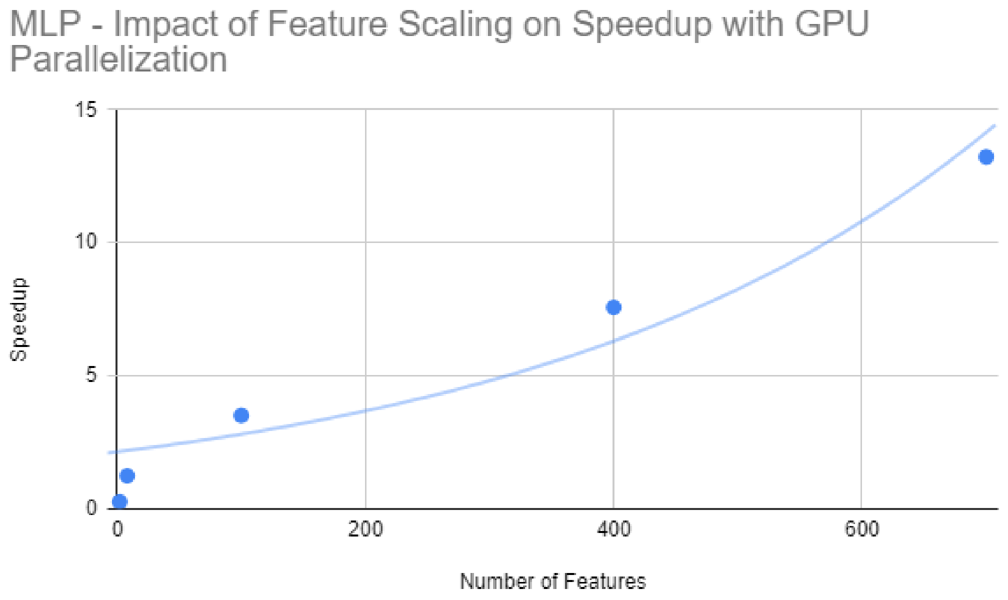


Figure 12 - Feature Scaling for Multi-Layer Perception

For each of the three models, as the feature size increases (and therefore the size of the model and the number of threads used increases), the speedup increases as well. This allows us to demonstrate the weak-scalability of all three models when using CUDA to parallelize their training algorithms on GPU hardware.

While both Logistic Regression and MLP achieve a speedup of 13x for Blobs-7D, we notice that SVM only achieves a speedup of about 4x. While still scalable, SVM does not benefit as well from parallelization, likely due to the relative simplicity of the training computations. Also, the computation of the dot product requires more synchronization between the threads. The dot product eventually moves to thread 0, which computes whether the constraint is satisfied and places it into shared memory for other threads to use during computation. This necessity for greater synchronization between the threads also can explain why the speedup is not as pronounced in the CUDA implementation of SVM training compared to the other two models.

Multi-Classification on MNIST Using SVM

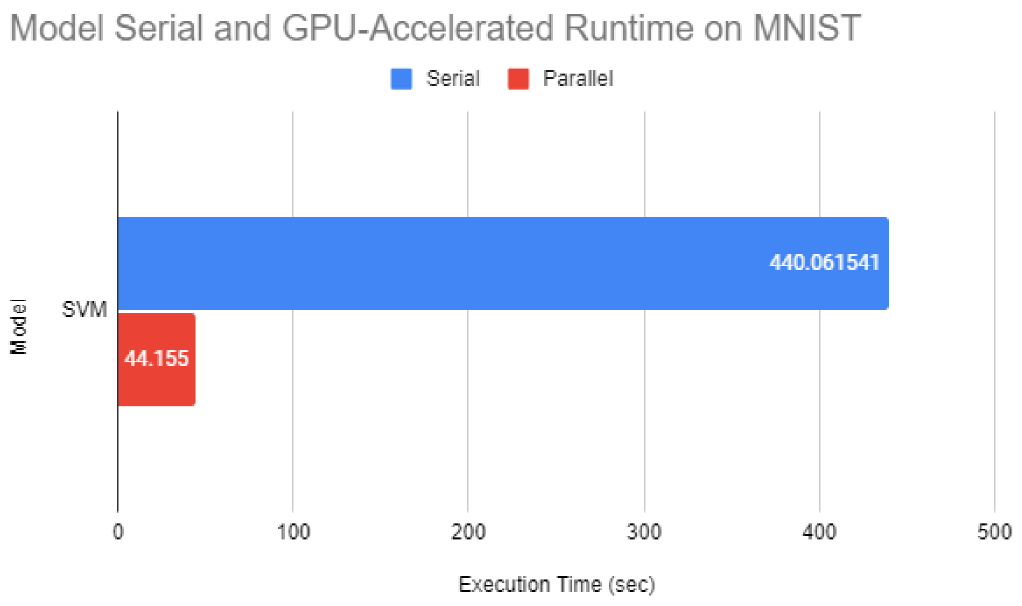


Figure 13 - Serial and Parallelized Execution Time for SVM on MNIST

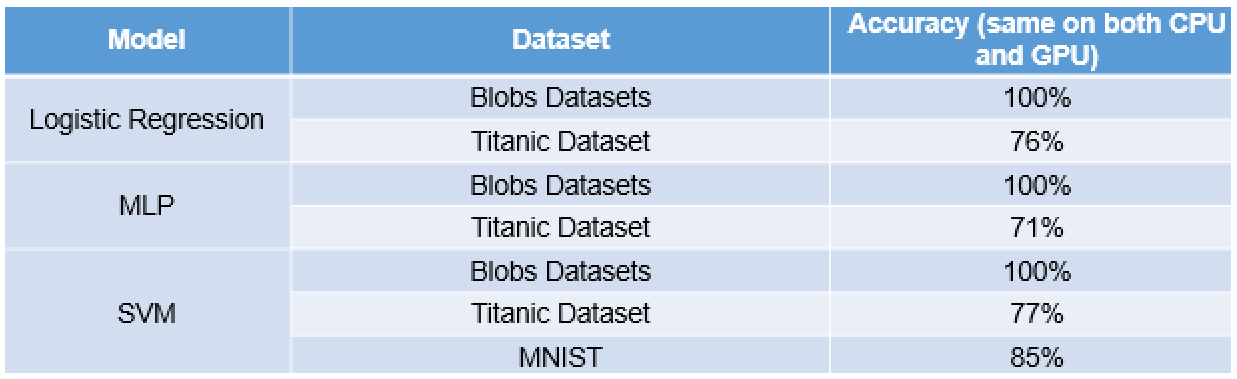
We decided to explore how parallelization would impact training time on not just binary classification, but also the more complicated task of multi-class classification. In particular, we focus on the 10-class classification task for the MNIST dataset of handwritten digits for the Support Vector Machine model. This is more complicated as it requires training not one, but 10 models. Figure 13 above shows how the SVM achieved approximately a 10x speedup when training was accelerated on the GPU compared to the serial implementation on the CPU.

It is interesting to note that while SVM did not achieve a very large speedup on the binary classification task with a similar number of features (Blobs-700D achieved about a 4x speedup while MNIST has 784 features), there is a substantial speedup on the multi-class classification task. This is because the CPU-based implementation must conduct training on each digit's classifier serially, meaning the execution time for training increases by a factor of the number of classes. However, since the CUDA implementation on the GPU trains each of these classifiers in parallel using 10 asynchronous streams, it is possible to achieve much greater speedup.

Accuracy Values For Correctness of Parallelization Strategies

Finally, we evaluate the accuracy of these models on the datasets when trained on the CPU and GPU, to ensure that both the serial and parallel implementations are, in fact, performing the same computations and achieving the same accuracy when evaluated on the training data.

Figure 14 below shows that we achieved the same accuracy between the CPU and GPU versions of the models, verifying our implementations.



Model	Dataset	Accuracy (same on both CPU and GPU)
Logistic Regression	Blobs Datasets	100%
	Titanic Dataset	76%
MLP	Blobs Datasets	100%
	Titanic Dataset	71%
SVM	Blobs Datasets	100%
	Titanic Dataset	77%
	MNIST	85%

Figure 14 - Accuracy of the Models on Various Datasets

Conclusion

In this project, we implement 3 models for binary classification: Support Vector Machine, Logistic Regression, and Multi-Layer Perceptron. For each of these models, we identify computations within training that can be parallelized, and we develop from scratch a CUDA implementation for these models that leverages GPU-hardware. We test these models on various datasets and realize the execution time for accelerated training on the GPU almost always outperforms the CPU-based implementations. Furthermore, we explore how model size (determined by the feature size) impacts speedup, and we are able to therefore show that our CUDA implementations achieve weak scalability. Finally, we explore how we can extend the SVM binary classification model to do multi-class classification, and develop a CUDA based implementation as well. We use this to showcase how the multi-class classification model of SVM benefits from GPU parallelization even more so than the binary classifier.

References

[1] S. Das, “CURSE OF DIMENSIONALITY,” *Medium*, Jul. 01, 2020. https://medium.com/@soumiksanku08/curse-of-dimensionality-293d0d16fe2a (accessed Dec. 06, 2023).

[2] baeldung, “Multiclass Classification Using Support Vector Machines | Baeldung on Computer Science,” *www.baeldung.com*, Oct. 07, 2020. https://www.baeldung.com/cs/svm-multiclass-classification

[3] M. Lanhenke, “Implementing Support Vector Machine From Scratch,” *Medium*, May 01, 2022. https://towardsdatascience.com/implementing-svm-from-scratch-784e4ad0bc6a

[4] D. Egorov, “Logistic Regression From Scratch,” *Geek Culture*, May 07, 2021.

https://medium.com/geekculture/logistic-regression-from-scratch-59e88bea2ba2?fbclid=IwAR0XUBBcqnu43pqbsCOODUk5GrwlMLsoIQRyF9u5af_VhMKLMNoTkP41Ia0 (accessed Dec. 06, 2023).

[5] GeeksForGeeks, “Multi-Layer Perceptron Learning in Tensorflow,” *GeeksforGeeks*, Nov. 03, 2021. https://www.geeksforgeeks.org/multi-layer-perceptron-learning-in-tensorflow/

[6] I. Species, “Multilayer Perceptron from scratch,” *kaggle.com*, 2021. https://www.kaggle.com/code/vitorgamalemos/multilayer-perceptron-from-scratch?fbclid=IwAR3o9FSYRiqCGFIIo34No4C6ZHvG0Ksbu_kImgldC3rL6neoKlHBzloHafk (accessed Dec. 06, 2023).

[7] “sklearn.datasets.make_blobs,” *scikit-learn*. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

[8] “Titanic - Machine Learning from Disaster,” *kaggle.com*. https://www.kaggle.com/competitions/titanic

[9] H. KHODABAKHSH, “MNIST Dataset,” *www.kaggle.com*, 2018. https://www.kaggle.com/datasets/hojjatk/mnist-dataset

[10] “Papers with Code - MNIST Dataset,” *paperswithcode.com*. https://paperswithcode.com/dataset/mnist