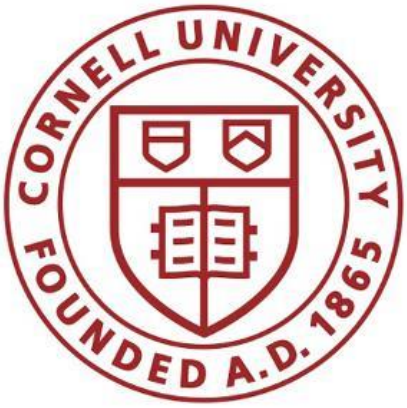
Hardware Accelerated Machine Learning Via Parallel Programming on GPUs

A Design Project Report



School of Electrical and Computer Engineering of Cornell University

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering, Electrical and Computer Engineering

Submitted by:
Justin S. Selig (jss459)

M. Eng Project Advisor:
Prof. Zhiru Zhang

Degree Date:
December 2017

Table of Contents

Project Title:
Hardware Accelerated Machine Learning via Parallel Programming on GPUs

Author:
Justin Selig

Abstract:
The aim of this project is to diagnose and remedy a computational bottleneck that exists within a machine learning model used by Prof. Zhiru Zhang's research group – an application of logistic regression for email spam-filtering. This project involves the use of parallel-programming techniques in CUDA to implement a mapping and reduction algorithm which speeds up the stochastic gradient-descent over objective functions which train the system. This project is an extension of the work of Gustavo Angarita, a former PhD student in the Computer Systems Laboratory. As a result of this work, the improved algorithm trains the model in less time without affecting test error-rate. This increases the multi-modal efficiency of the spam-filter.

Keywords:
GPU, Logistic Regression, CUDA, Map-Reduce, Parallel-Programming, Machine Learning

Report Approved by Project Advisor:
Prof. Zhiru Zhang

Executive Summary

This project began as a foray into quantitatively analyzing the behavior of a GPU for the execution of an optimization algorithm developed by a previous student to train a spam-filter. The application of spam-filtering described in this project is a prototypical example of the use of logistic regression (LR). The goal was specifically to gain an understanding of and characterize the behavior of a GPU to discover any potential bottlenecks that yielded poor performance on this Machine Learning (ML) model. The previous attempt to optimize a parallel-reduction Stochastic Gradient Descent (SGD) optimization resulted in indeterminate accuracy, interminable execution, and undefined behavior which was wrongly attributed to thread divergence.

This project therefore evolved from trying to gain an understanding of the GPU behavior to discovering a deeper flaw within the previous implementation of the algorithm itself. One main accomplishment of this project was therefore in diagnosing and resolving this algorithmic issue from a prior implementation of a parallel map-reduce algorithm in CUDA.

This document covers the background research and design exploration involved in the approach towards resolving a critical bug in the spam-filter and subsequently obtaining tangible results. This paper also explores the design considerations of parallel-programming models and suggests methods for profiling behavior on a complex system which abstracts much of its compute functionality.

1. Introduction

Training of machine learning (ML) algorithms is computationally intensive involving millions of linear arithmetic operations on large-scale datasets over many iterations in the training phase of a model. These computations are often simple and need to be redundantly applied across multidimensional vectors or tensors of data. SIMD (single-instruction multiple-data) processors have been integrated into modern architectures, including CPUs, and allow for fast computation of one operation across a data-point. These cores are fundamental to the operation of GPUs (Graphical Processing Units) built originally for compute-intensive graphics applications. GPUs are now being applied to training and inference of machine learning models because of their ability to parallelize vector computations across hundreds of SIMD cores. My goal for this project is to exploit these sophisticated microarchitectural units to accelerate the learning phase of an ML model by using a parallel-programming approach on a GPU.

Optimization over convex objective (loss) functions for linear ML models is highly suited for parallel computation on GPUs since the main compute step involves a basic vector dot product with map-reduce logic. An application suited for logistic regression is spam-filtering for emails – determining whether a Bag of Words (BoW) falls under one of two classes: spam or ham. Prof. Zhang's research group has developed a spam-filter using a logistic regression algorithm. A form of gradient-based stochastic optimization (SGD) is applied to this model using SIMD cores on an Intel CPU in addition to mini-batch stochastic gradient descent (MBGD) both using an open-source linear algebra API. In this project, I implement a parallel mapping and reduction algorithm capable of speeding up various SGD optimizations over objective functions to train the spam-filter.

As a result of my work, the spam-filter model is able to be trained with fewer user-defined computations in less wall-clock time without affecting test error-rate. By taking a low-level parallel programming approach to this and various other models that apply gradient-descent, we may gain a better understanding of the computational bottlenecks that exist on GPUs. The end result in applying such an optimization is that this approach may increase the multi-modal efficiency of the spam-filter and similar models alike.

2. Design Problem

In the following section, I provide a background for the problem addressed and outline the algorithmic methodology and systems used to address this problem.

2.1 Motivation

The machine learning (ML) application considered in this project is a spam-filter. Spam-filters are ubiquitous in any email platform since they help reject unsolicited mail that clutters our inboxes. These filters act as classifiers to determine, based on the content and sender of the email, whether or not the message belongs to two categories: spam or ham (not spam). The filter is able to accomplish this because it has foreknowledge of what emails in these categories look like based on a training process that considers vast amounts of prior spam/ham emails. Spam emails constantly shape-shift and are becoming more difficult to classify. A high-performance model is therefore necessary to adapt to the changing email landscape in which spammers attempt to outsmart users and infect host machines.

2.2 Background

ML algorithms come in many shapes and sizes, however, they may be generally divided into two sub-disciplines (supervised and unsupervised) and further divided into two sub-tasks (training and testing). Supervised ML is a highly-researched sub-discipline that involves intervention by an external authority to provide "labels" for training data that, through an optimization process, characterize a model via a set of learned parameters. This trained model serves as a representation of some underlying, God-given probability distribution. Therefore, with a trained model, one is able to automate the classification or regression of unseen data that are theoretically derived from the same distribution. Below, I outline this process for the model considered in this project.

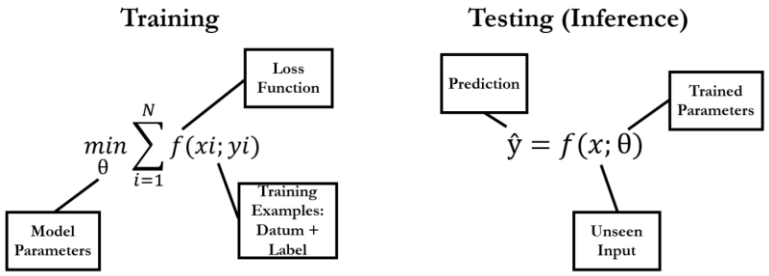


Figure 1: Generalization of the supervised ML approach. Left: The convex (or non-convex) optimization objective which seeks to minimize some function *f* over n-dimensional vectors *x* (input data) and *y* (labels). Right: Making prediction with learned model θ, function *f,* and unseen data *x*.

2.2.1 Logistic Regression as an ML Model

Logistic Regression (LR) is an example of a learning technique that models a sigmoidal distribution usually for binary decision-making. More specifically, the model consists of a logistic hypersurface fit to a set of points with binary labels: positive (1) or negative (0), in this case, spam or ham, respectively. This hypersurface is learned through a routine known as 'training.' Once this model is obtained after training is complete, the filter can be applied to subsequent emails via the the following procedure (testing):

1. The input to the filter is a raw email message. This message is known as a Bag of Words (BoW) since it is an unsorted vector of textual data.
2. Raw messages are transformed into feature vectors which are simplified representations of the email. This feature extraction from the BoW is intended to reduce the dimensionality of the input to the filter. This occurs through a process known as 'hashing' where each word is passed through a function that maps it to some numerical value. The hash function is a frequency counter for this application (histogram generator), however, it could represent any function that groups similar (or dissimilar) words to buckets. The number of buckets is similarly customizable.

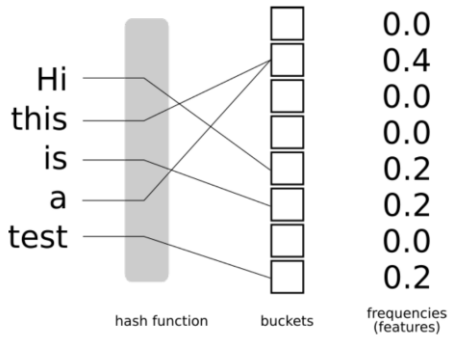


Figure 2: Visualizing feature extraction. Raw data in the form of a BoW is sorted into buckets representing the frequency of occurrence of any similar words[1].

3. The features for this particular email are provided as input to the LR model which outputs a value between 0 and 1. This regressed value can be interpreted as the probability that the input email is spam. This classification occurs based on a threshold value which is usually set to 0.5. Values below this are not considered spam.

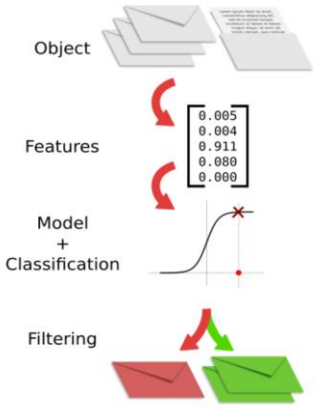


Figure 3: Visualizing the inference (classification) phase of a spam-filter[1].

2.2.2 Optimizing the Model

The described procedure is made possible through a training phase of the LR model. The purpose of this training phase is to learn the parameters that fit a multidimensional sigmoidal distribution defined by the following equation:

$$\pi_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(\boldsymbol{x} \cdot \boldsymbol{\theta}) = \frac{1}{1 + e^{-(\boldsymbol{x} \cdot \boldsymbol{\theta})}}$$

This verbally translates to: "The probability that data point x belongs to the positive class given parameter vector θ." The positive class is spam, the data point is the n-dimensional input feature vector, and θ is the n-dimensional parameter vector learned during training, described below. It is important to note that this operation includes a vector dot-product between x and θ. Such an operation would involve performing an element-wise multiplication (map) and sum across all entries (reduce) to yield a single numerical result:

$$\boldsymbol{x} \cdot \boldsymbol{\theta} = x_0 * \theta_0 + x_1 * \theta_1 + x_2 * \theta_2 + x_3 * \theta_3 + \ldots + x_{n-1} * \theta_{n-1}$$

Generating this model, specifically the parameter vector θ, involves minimizing a loss function that describes the distance between the model prediction and reality. The loss function chosen previously for this application is the negative log-likelihood for the Bernoulli distribution of the labels Y given X, and parameterized by θ [1]:

$$L(\theta) = -\ln\left(\prod_{i=1}^{m}(\pi_{\boldsymbol{\theta}}(\boldsymbol{x_i}))^{y_i}(1 - \pi_{\boldsymbol{\theta}}(\boldsymbol{x_i}))^{1-y_i}\right) = -\sum_{i=1}^{m} y_i \pi_{\boldsymbol{\theta}}(x_i) + (1 - y_i)(1 - \pi_{\boldsymbol{\theta}}(x_i))$$

Throughout the training process, this loss is used to continually update the model weights, or the entries of θ, until some convergence condition is satisfied. This process is known as Gradient Descent.

2.2.3 Gradient Descent (GD)

Gradient Descent is used to update the model weights of the parameter vector little-by-little through stepping in the direction of steepest descent for the loss function. The size of this step is determined the 'learning-rate', a meta-parameter η used in the update step: $\theta_{t+1} = \theta_t - \eta g(\theta)$. The steepest direction is determined by taking the gradient of $L(\theta)$ with respect to the parameter vector to obtain:

$$\mathbf{g}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{Y}) = \sum_{i=1}^{m} (\pi_{\boldsymbol{\theta}}(\boldsymbol{x_i}) - y_i)\boldsymbol{x_i}$$

It is also important to note that this operation includes making a prediction π_θ which involves the same vector dot product described in section 2.2.2. If these are high-dimensional vectors which serve as operands to a single multiply-accumulate (MAC) module on an unoptimized CPU, this computation alone could be extremely expensive in terms of wall-clock runtime. Luckily, most modern processors are optimized to account for these types of operations through the use of SIMD (single-instruction, multiple-data) cores.

2.2.4 Stochastic Gradient Descent (SGD)

The above method describes a form of GD called Batch Gradient Descent (BGD) which uses the whole training set to determine the gradient of the loss function before the parameter vector is updated. An alternative form of gradient descent, SGD, was applied in this application to add stochasticity to the update step as well as speed up gradient computation. As opposed to GD, SGD draws a single training data point randomly from the available training set to compute the gradient of loss on each update of the parameter vector. This stochastic update is known to help the model converge faster to a desired minimum loss[2].

Figure 4: Visualizing the convergence pattern of BGD (left) and SGD (right) to a single minimum in dark blue[1].

Alternatively, there is the option to execute a mini-batch SGD (MBGD) which, rather than use the entire training set or single data point, uses a few random data points to calculate gradient. While

this is an optimal medium between the above two described approaches, this paper focuses on the employment of SGD for algorithmic simplicity.

2.2.5 Utility of a GPU

With a reasonable-size dataset, the training of LR can be performed on a conventional CPU with few drawbacks in efficiency. However, most ML practitioners have begun to realize that more data equals better test accuracy. It has therefore become imperative that the ML industry move towards hardware capable of accommodating larger and larger datasets in order to reduce the overhead of applying compute-intensive training on ML models at scale.

A Central Processing Unit (CPU) is the term coined to describe a general-purpose processor typically found in most personal computers and embedded systems today. CPUs are optimized for complex control flow as they must be able to execute a wide array of programs that abstract away details about the CPU microarchitecture itself. These devices typically have very few cores (1-8) with complex cache hierarchies that work well for non-uniform memory accesses. Such devices may be well-suited for ML algorithms with small or medium-sized datasets, however, as a dataset expands in size and the number of parameters required in a model increases, both training and inference (testing) start to take a hit in efficiency.

As an alternative, ML practitioners have looked to Graphical Processing Units (GPUs) to train and test their models. As opposed to CPUs, GPUs have many more cores - simpler in design - grouped together in a Streaming Multiprocessor (SM) model. GPUs were built originally for compute-intensive graphics applications requiring massive amounts of matrix operations. Because of its immense number of cores, a GPU is capable of performing many simple arithmetic operations in parallel per cycle. As it became apparent that ML models required similar levels of large-scale operations, practitioners began looking to GPUs to accelerate their algorithms.

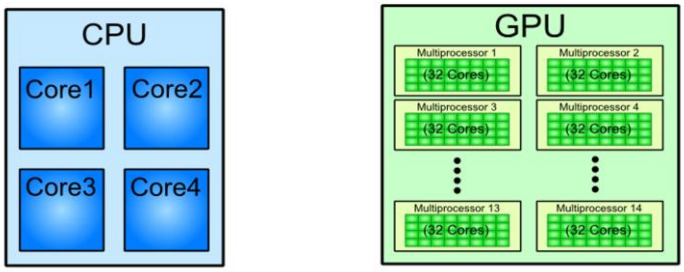


Figure 5: Visualizing a CPU (left) and GPU (right)[3].

One should note that the most computationally expensive operation performed in the GD optimization for this application, a vector dot-product, is highly parallelizable[2]. Each entry of x and θ for any single step is independent of one another. Because of this, each dot-product entry may be computed simultaneously - a fitting task for a GPU. Such a distributed computation is known as a

'map-reduce' where individual vector elements are distributed (mapped) to worker threads that each perform one or more multiplication operations. The final dot-product is computed when the thread workers' results are summed (reduced) to a single numerical answer. This reduction step may also be performed in parallel by several but fewer threads depending on the size of the original vector operands.

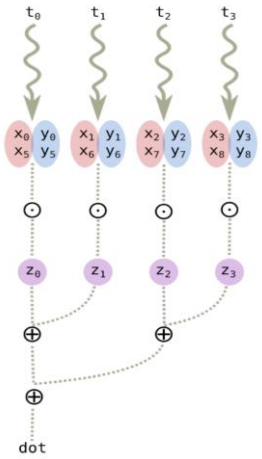


Figure 6: Visualizing a map-reduce operation for vectors **x** · **y**. Each thread computes and sums two dot-product entries. Then, the resulting z's are summed by another thread, then once more to compute a final result[1].

3. Approach

In the following section, I detail the system used in this project, software design, and the various approaches toward obtaining tangible results.

3.1 System Overview

A typical GPU is setup as a co-processor to a CPU. The CPU begins execution of a program and dispatches data to the GPU which performs some computations and returns a result back to the CPU. For the purpose of the spam-filter, training occurs when data points stored on the CPU are sent to the GPU which executes a prediction and parameter vector update. This process is outlined in a block diagram below.

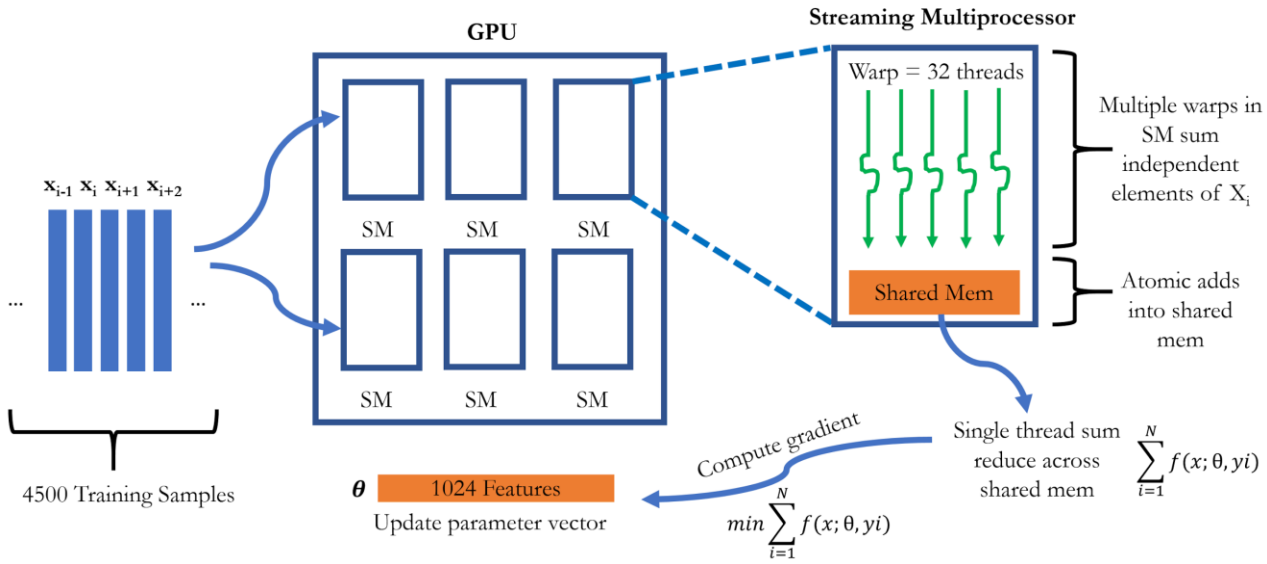


Figure 7: Block diagram of the spam-filter training process.

Training set feature vectors are distributed across GPU streaming multiprocessors (SM). Each SM is allocated a block of shared memory and is capable of executing 32 worker threads in parallel, known as a 'warp'. In this application, these worker threads are responsible for performing element-wise multiplications for one or more vector dot-products depending on how many data points x_i are chosen to be dispatched at-once. Next, a gradient is calculated to perform the update step of SGD. In order to update the parameter vector, each thread that computes a dot-product has to perform a read-modify-write operation on the feature vector θ. The feature vector is therefore a shared resource and must only be acted upon in an atomic fashion by any one thread.

3.1.1 Dataset

The full dataset contains 5000 data points of 1024 elements (dimensions) each. This is split among a training and test set. The training data set consists of 4500 feature vectors and the test set consists of 500.

3.2 Software Design

The software written for this project was in CUDA, a language developed by NVIDIA for use on its GPUs. CUDA is similar in syntax to C++, however, it includes additional functionality for interfacing with the GPU device.

3.2.1 The Parallel-Programming Model

A CUDA program is written in a manner that logically divides up work between a CPU and GPU. In order to invoke GPU functionality, a CPU would perform a 'kernel call' - not to be confused with the kernel of an Operating System. The kernel is provided several arguments including a grid-size, block-size, and optional shared-memory size akin to calling a 'function' as follows:

kernel_name <<<grid_size, block_size, shared_mem_size >>

The GPU divides up threads into a set of multidimensional 'blocks' which have a shared-memory. These blocks are organized in a multidimensional 'grid' which may access a global memory accessible across blocks. In this way, the user does not have to worry about the distribution of work within the GPU hardware and may refer to any one thread by its cartesian coordinate by the user-defined grid and block location.

3.2.2 Existing Framework

Prior to implementing the optimized parallel map-reduce algorithm I spent time learning the existing framework created by Gustavo Angarita, a former PhD student in the Computer Systems Laboratory. Gustavo had implemented a C++ version of various GD algorithms for a CPU using an open-source linear algebra API known as OpenBLAS (basic linear algebra sub-program). Where possible, this library exploits the capabilities of a modern CPU to perform SIMD operations which make large-scale vector computations highly efficient. The overall training process worked as follows:

```
// Ignore the first run to discard initialization overhead
training_function(training_set, training_options);

// Shuffle vectors and train num_runs times, then average results
for (size_t k = 0; k < benchmark_options.num_runs; k++) {

        // reset parameter vector to forget previous training
        resetParameters(training_set, training_options);

        // shuffleKeyValue(data_points, labels, num_points_total, num_features);
```

```
            training_timer.start();
            training_function(training_set, training_options);
            training_time += training_timer.stop();

            // Get Training error
            computeErrorRate(training_set, &train_errors);

            // Get Testing error
            computeErrorRate(testing_set, &test_errors);

    }
    // reset configuration parameters
    *training_options.step_size = Training_options.config_params["initial_step_size"];

    training_time /= benchmark_options.num_runs;
    scaleErrorDict(&train_errors, 100.0 / benchmark_options.num_runs);
    scaleErrorDict(&test_errors, 100.0 / benchmark_options.num_runs);
```

This top-level code initializes a wall-clock timer, and iterates through a set of runs (30) where on each run the particular training function (SGD, MBGD) is a called via a function pointer passed in as an argument to the method wrapping the above code. The average test error and training time is computed and averaged across all the runs to yield a result to the user. This training function is defined by the user for any type of training - whether it be a BLAS implementation of GD for a CPU or a CUDA implementation of GD that launches a kernel call to operate on a GPU. However, in each training function, multiple epochs are executed until a convergence condition, specified by the user, is satisfied. One epoch represents a single use of the entire training set to do gradient calculations. The convergence condition is a limit on the maximum allowable test accuracy computed after each run.

3.3 Debugging a GPU

The three-word title of this section may cause the novice GPU-programmer to cringe. Perhaps what makes parallel-programming such a valuable skill is the learning curve that exists when trying to wrap your head around the behavior of a GPU. When a GPU programmer invokes a kernel to distribute computations across thousands of threads allocated to hundreds of cores in a non-deterministic system whose proprietary compiler abstracts the behavior of hardware, the toolset to understand system-behavior is limited. While there is no way around the migraines that result from attempting to debug such programs, frameworks such as NVIDIA's Visual Profiler have been built to make this process slightly less overwhelming. Below, I describe the process of discovering the issue that existed in the prior CUDA implementation of a parallel map-reduce algorithm and the tools employed to remedy it.

3.3.1 Discovering the Issue

The process of discovering a bug in the existing CUDA-optimized algorithm began once I started profiling the behavior of the supposedly optimized parallel reduction. The spam filter was designed to terminate training once a desired accuracy was reached (<5%). However, it had gone unnoticed

that the spam-filter also terminates training under the condition that 100 epochs had passed. I call this condition an 'epoch time-out.' A single training epoch is when the entire training set has been used to update the parameter vector - the training set is repeatedly applied until the convergence condition holds. I discovered in the process of running the optimized implementation that the GPU program took much longer to converge than the baseline CPU implementation. In fact, any time the program was executed and a timeout occurred, the total training time was non-deterministically distributed across a range of wall-clock times.

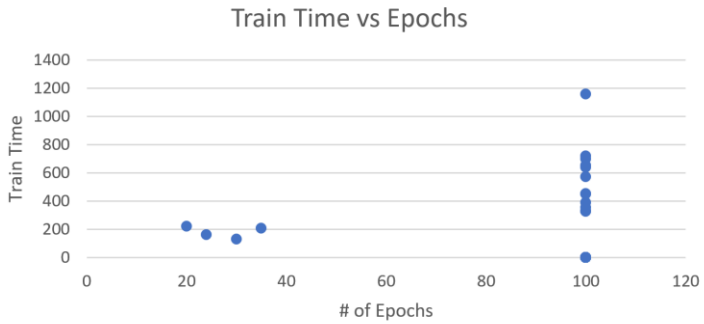


Figure 8: Profiling the CUDA SGD based on runtime.

I began tinkering with other another optimization algorithm developed based on the prior one which employed MBGD instead of a SGD for training. This also demonstrated odd behavior as any batch sizes that used anything more than a trivial number of data points experienced a 100-epoch timeout.

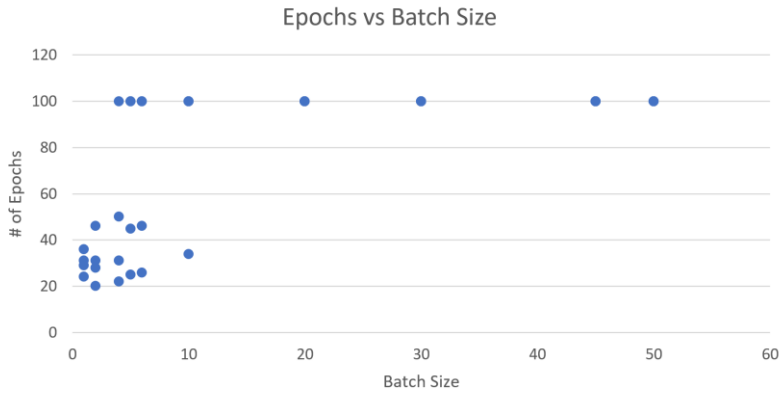


Figure 9: Profiling the timeout condition for various CUDA MBGD batch sizes.

Because of the repeated appearance of this timeout anomaly, training accuracy was unusually high at around 30% while the baseline CPU implementation continued to satisfy the 5% accuracy threshold. Speaking with a former student who had worked on this implementation, it was surmised that error rates remained high because of a condition known as 'thread divergence.' Thread divergence occurs when a multithreaded system containing shared memory has non-blocking operations that execute

in an undefined manner because of poorly-managed control flow. However, when I removed the timeout data points from my analysis, average test error remained relatively constant across CPU SGD, MBGD, and CUDA SGD/MBDG algorithms.

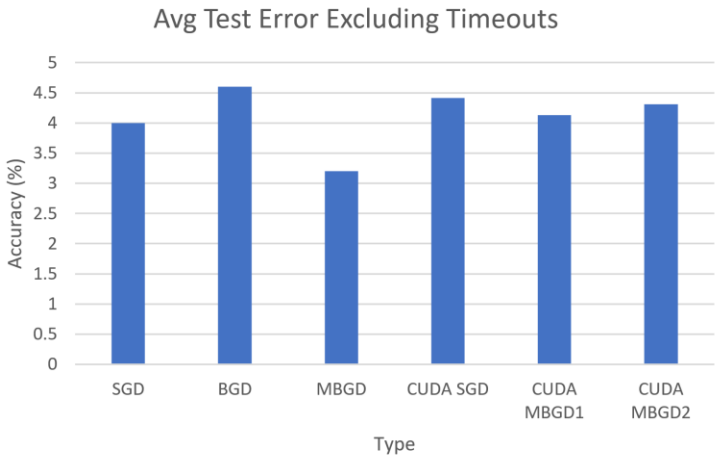


Figure 10: Average test error for various optimizations excluding timeouts.

Because of this non-deterministic functionality, I refused to believe that the GPU controlling thread behavior would operate correctly in some scenarios but fail in others. It was clear that the issue lie not in the hardware but in the software implementation itself. Based on the framework described in section 3.2.2, I developed a custom training function similar to the baseline BLAS implementation and the previous CUDA SGD algorithm.

3.3.2 Implementing the CUDA Software

I began digging into the previous CUDA implementation itself. Since the code surrounding kernel calls in this program was relatively generic, the problem must've arose from either the mapping or reduction step of the distributed (partial) dot product. After reformulating the thread distribution methodology, I came up with a mapping implementation that accounted for a number of variables in the program:

```
size_t tidx = threadIdx.x;
size_t points_per_block = (blockDim.x / threads_per_datapoint); //datapoints per block
size_t point_idx = (blockIdx.x * points_per_block) + (tidx / threads_per_datapoint);

// index relative to the datapoint instead of the block
size_t relative_tidx = tidx % threads_per_datapoint;
size_t num_thread_computations = num_features/threads_per_datapoint; //# computes/thread
size_t thread_start_idx = relative_tidx * num_thread_computations;
```

Ultimately, this block of code describes the proper distribution of GPU threads across the input datapoints. This accounts for a variable number of datapoints, datapoints per block, and threads per datapoint - all different configurations that must be met for the spam-filter application. The main

purpose of this block is to compute mutually exclusive thread bounds (*thread_start_idx, num_thread_computations*) for a kernel execution in one particular block (*blockIdx.x*) that are passed into a reduction function.

With this modification to the mapping scheme, I was able to obtain correct results by performing a linear reduction across the described bounds. This involved simply looping across the thread bounds, summing up individual partial dot-products, writing the result to shared memory, then looping over and summing the results in shared memory for a single data point. This, however, was an inefficient use of the GPU since it did not take advantage of the GPU's parallel compute capability. Instead, I designed the reduction as follows:

```
FeatureType partial_dot = 0;
FeatureType temp_data_i, temp_param_i;
for (size_t i = thread_start_idx;
   ((i<thread_start_idx + num_thread_computations) && (i<num_features)); i++)
{
    temp_data_i = __ldg(data_point_i + i);
    temp_param_i = __ldg(parameter_vector + i);
    partial_dot += temp_data_i * temp_param_i;
}

__syncthreads();
FeatureType sum = warpReduceSum(partial_dot);
shared_memory[threadIdx.x/threads_per_datapoint] = 0;

__syncthreads();
if (relative_tidx % 32 == 0)
{
   atomicAdd(&shared_memory[threadIdx.x/threads_per_datapoint], sum);
}
```

This block of code, although short in size, is the most critical portion of the algorithm. In this section, each thread computes a partial dot-product by performing a Multiply Accumulate (MAC) on a section of the input data point x_i and parameter vector θ. Rather than index these arrays via standard C/C++ syntax, I invoke a load operation which is optimized for streaming (sequential) memory accesses. Since this partial reduction may take longer for some threads, I insert a *__syncthreads* instruction which acts as a fence barring any thread from continuing computation until all threads in the associated block have reached the same point. Each variable in a CUDA program is considered a vector which stores the same variable's value in entries of the vector for all threads (32) in a warp. By design, *threads_per_datapoint* is a multiple of 32 and each element of a data point is guaranteed to exist in the same block. By invoking *warpReduceSum* on the *partial_dot* vector, I am able to quickly sum up elements in a warp where these elements represent 32 entries of a single datapoint. After syncing once more, the goal is to sum up the rest of the partial dot-products for a data point. In order to communicate across datapoint threads, each thread that computes a warp sum atomically adds into the same region of shared memory. Each data point is reserved its own portion in this memory structure.

3.3.3 Debugging CUDA Software

This reduction technique yielded incorrect results at first and required deeper intervention since the logic was sound. To do this, I simplified the problem by altering the input data points to 1024-element vectors with entries of all 1's. The result of such a dot product should be 1024. It turned out, however, that the dot-product value computed was non-deterministic and arbitrarily large. This indicated that I was either summing up the address pointers rather than the content, or there were memory integrity issues in which I was not actually reading/writing properly to the shared memory locations.

It is impossible to directly print to a console the values computed within a kernel without obtaining what amounts to garbage. The GPU acts as a co-processor to the console-controlling CPU and therefore must copy back data to the CPU in order to give the user any idea of what happens during runtime. This copy, however, can only occur after a kernel call. I therefore came up with a grey-box technique to probe the contents of GPU variables by allocating static memory for a 'probe' variable on the CPU, passing this to the GPU via global memory in a kernel call, assigning the probe a value during runtime execution of the GPU kernel, and then copying the value back to the CPU at which point it could be printed to the console. As a result, I discovered that the shared memory within the thread blocks had remnant data from previous kernel executions. It is therefore imperative that, before results of a dot-product can be passed to shared memory, the memory contents must be zeroed.

4. Performance Analysis

There are several ways to evaluate the performance of a ML model. From the practitioner's perspective, what matters most is the algorithm's performance on unseen data. However, from a pragmatist's point-of-view, slightly improved test results are only worth the trouble if runtime is reasonable. Luckily, a GPU aids us in both these regards.

The result of applying the optimized CUDA implementation of a parallel map-reduce algorithm was about a best-case 15% speedup over the baseline CPU implementation and 26% speedup over an unoptimized CUDA linear reduction with no significant hit to accuracy. Average test error remained around 6.4% excluding cases where the GPU could not execute because the requested number of threads exceeded the available GPU resources.

Below is a table of the results from a single train and test for various configurations of the LR parameters. This was generated without taking into account the convergence scenario as a testament to the algorithm's raw performance without excessive overfitting from performing too many epochs.

Type	*Datapoints Per Block*	*Threads Per Datapoint*	*Total Runs*	*Num Epochs in Last Run*	*Train Time*	*Train TPR*	*Train FPR*	*Train Error*	*Test TPR*	*Test FPR*	*Test Error*
CPU SGD Baseline	--	--	30	30	85.8341	99.99999	0	0	94.56522	1.898734	3.2
CUDA SGD Optimized	1	128	30	30	73.59577	98.93245	1.482524	1.32963	94.14855	3.860759	4.593333
CUDA SGD Optimized	2	128	30	30	73.20963	98.48613	0.777621	1.048889	93.36956	2.415612	3.966667
CUDA SGD Optimized	4	128	30	30	78.10077	99.38279	0.442177	0.506667	93.87681	3.291139	4.333333
CUDA SGD Optimized	8	128	30	30	103.434	97.59549	1.258503	1.680741	92.08333	3.765823	5.293333
CUDA SGD Optimized	1	256	30	30	119.6902	95.01809	4.074595	4.408889	92.08333	6.381856	6.946667
CUDA SGD Optimized	2	256	30	30	122.9877	99.81102	0.234577	0.217778	94.34782	3.175105	4.086667

CUDA SGD Optimized	4	256	30	30	173.4247	98.77362	1.035656	1.105926	93.24275	3.829114	4.906667
CUDA SGD Optimized	8	256	30	30	5.7368	0	0	36.84444	0	0	36.8 Exceeds Resources
CUDA SGD Optimized	1	512	30	30	220.2363	99.92762	0.112597	0.097778	94.38406	3.364979	4.193333
CUDA SGD Optimized	2	512	30	30	318.1911	62.89706	10.7225	20.44222	65.77898	11.23418	19.69333
CUDA SGD Optimized	3	512	30	30	5.739667	0	0	36.84444	0	0	36.8 Exceeds Resources
CUDA SGD Optimized	4	512	30	30	5.7507	0	0	36.84444	0	0	36.8 Exceeds Resources

Figure 11: Performance of Optimized CUDA SGD under various thread configurations.

As seen in Figure 11, it is interesting to note how performance degrades as the total number of requested threads increases. Figure 12 below demonstrates that total training time increases monotonically with an increase in the total number of threads.

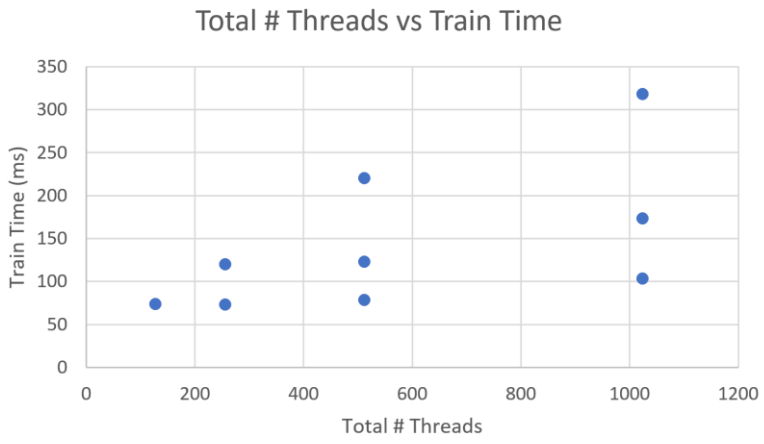


Figure 12: Training time vs total number of requested threads.

This figure is particularly notable because, even with an increase in thread count, there is no change in the outcome of computation. However, the time it takes for training to occur increases even with a training set of the same size. This phenomenon likely occurs due to issues in mutual exclusion. With the increased number of atomic memory accesses that must occur after each warp reduction, threads must experience a spin-lock condition in which they must be scheduled to write their result

to the shared memory resource. This result is also surprising considering the initial partial dot product calculated via an explicit for-loop is smaller for each worker thread when there are more threads overall. This demonstrates that, as far as runtime is concerned, many streaming memory reads for computing partial dot-products are preferred over fewer read-modify-writes to shared memory.

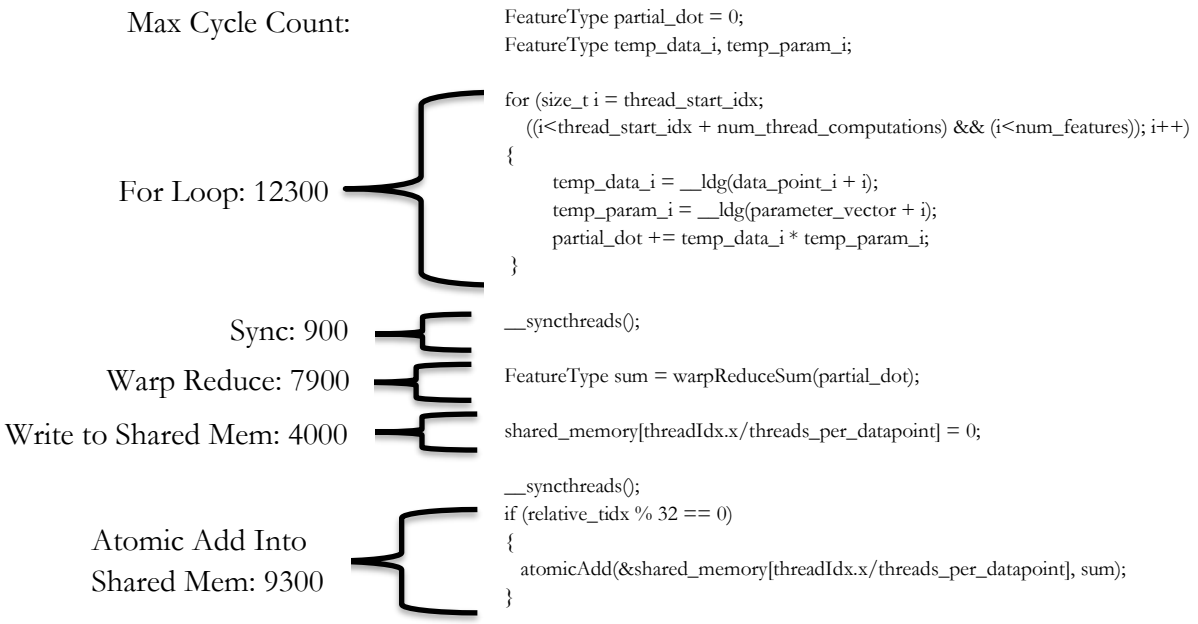
4.1 NVIDIA NSight GPU Visual Profiler

One tool provided by NVIDIA for profiling the behavior of CUDA programs is their Visual Profiler. This tool turned out to be less helpful than anticipated since it only provided total execution times for kernel calls and CUDA API calls (memcpy) which could be done with any ordinary software timer. The output below shows multiple runs of the train-and-test script for the optimized CUDA SGD. Gold represents memory-copy routines either from host-to-device (CPU to GPU) or device-to-host (GPU to CPU). These are around 9.5ms in length. Each blue block represents a single kernel call, about 3.5ms in length. There are multiple blocks that occur after a single memory-copy since the algorithm executes multiple epochs which each involve a kernel execution.

Figure 13: Timeline output of NSight: 10ms memcpy, 3.5ms per kernel call

4.2 White Box Testing Kernel Code

The output of the Visual Profiler is helpful in understanding system bottlenecks, however, it is not so helpful when it comes to mapping these bottlenecks back to code. I scoured the internet for a formal tool to help profile my software implementation to no avail. However, using the software 'probe' methodology described in section 3.3.2, I was able to make leeway on this task. As it happens, CUDA includes a clock API that allows a programmer to measure clock cycles passed during and within a kernel call[4]. Measuring the number of clock cycles for sections of code helped map out the critical sections within my SGD implementation as such:



The timing results above are quite intuitive. The longest section of code that runs is the for-loop, followed by atomic additions into shared memory, followed by the warp reduction. Thread synchronization also consumes some wall-clock time, but is trivial compared to the other sections. It is clear that software-defined iterations within a GPU kernel are unideal from a performance perspective. This profile also reinforces the fact that atomic accesses (read-modify-writes) to shared memory consume a large portion of program runtime.

5. Discussion

The previous attempt to optimize the LR parallel compute resulted in indeterminate accuracy, interminable execution, and undefined behavior. By examining the behavior of this program, I was able to diagnose and remedy an algorithmic flaw. As a result of applying an optimized parallel map-reduce in CUDA for a GPU, I was able to achieve a slight speedup over the baseline CPU. I was also able to profile the wall-clock time for sections of my optimized code to identify potential bottlenecks.

5.1 Future Considerations

While these results are significant and reproducible for any similar GPU platform, they come at a cost to the developer. Although NVIDIA makes managing threads a logical task through its multidimensional thread/block paradigm, working with such a structure is cumbersome when adding any significant level of modularity or abstraction to the GPU software. This calls for the use of higher-level libraries that reduce the cognitive load of managing thread indices and free a programmer to think about the algorithm design itself. Libraries such as TensorFlow, although written in Python, are GPU-compatible and even contain automated functionality to perform gradient descent. Other libraries such as Torch are built directly on-top of CUDA and use custom high-level syntax with Lua.

As for profiling the behavior of a GPU, there is not much in the way of tools that help do this. Most GPUs abstract the Intellectual Property of the manufacturer, which includes compiler behavior and microarchitectural details. For an NVIDIA GPU, the only acknowledged profiler available is the NSight tool. Nonetheless, measuring bottlenecks in code is still possible through the technique described in section 4.2.

Based on the runtime results compared to CPU, one might wonder why using an optimized algorithm on a GPU results in only marginal speedup over baseline. For one, the 5000-sample dataset employed is actually not that large. The total number of entries in the whole dataset is 5,120,000 (5000*1024). On a single-cycle CPU with no threading capability running on a low-power embedded system with a low clock-speed, perhaps it might take a lot longer to run the LR algorithm on this dataset. However, the baseline system uses an Intel Xeon Processor with 8-cores running at 2.6 GHz clock speed and employs SIMD cores for vector computations. There is not much room for improvement unless the dataset is scaled up by an order of magnitude.

5.2 Further Optimizations

Based on the output of the NSight Visual Profiler, it is evident that repeated calls to the CUDA API for memory copy functionality take up a small but noticeable portion of the overall runtime. Since the same feature-vector data are used between epochs, it would be interesting to explore the

possibility of storing weights on the GPU device in global memory for a collection of epochs. If the GPU permits, it might also be possible to store the entire 4500-sample training set on the GPU to eliminate the need for communication across runs. Such an optimization would remove about 10ms from each training run.

The spam-filter may also benefit from the conversion to fixed-point representations for features and the use of fixed-point arithmetic. This is a form of quantization that has been known to dramatically accelerate the speed of both training and inference without significantly impacting test error rate[5].

Another approach would be to modify the algorithm itself by substituting the atomic additions into shared memory with normal addition. This lock-free approach to parameter-vector updates is known as 'Hogwild' and has been shown to significantly decrease runtime without affecting test error rate given there is some level of stochasticity to memory accesses[6].

It might also be worthwhile to explore the feasibility of developing an OpenCL implementation of LR for the purpose of synthesizing hardware modules on an FPGA. Using the Vivado High-Level Synthesis Toolkit, it may be possible, with few headaches, to port the existing CUDA/C++ implementation to an even more specialized device that would vastly accelerate the speed of computation with smaller power requirements.

6. References

[1] G. Angarita, "Parallel Logistic Regression for High-Performance Spam Email Filtering," 2016

[2] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li, "Parallelized Stochastic Gradient Descent," Adv. Neural Inf. Process. Syst., no. 23, 2011.

[3] https://sqream.com/wp-content/uploads/2017/03/cpu_vs_gpu-11.png

[4] https://stackoverflow.com/questions/10585990/how-to-measure-the-inner-kernel-time-in-nvidia-cuda

[5] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. CoRR, abs/1502.02551, 2015. URL http://arxiv.org/abs/1502.02551.

[6] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In Advances in Neural Information Processing Systems, pages 693–701, 2011.

7. Acknowledgements

I would like to acknowledge Prof. Zhiru Zhang for his guidance during the development of this project. I would also like to acknowledge Gustavo Angarita for providing a large code base which served as the framework for this project.

8. Appendix

Running the Source Code

To run the CUDA source code, follow the below procedure:

1. Login to *en-openmpi04.ece.cornell.edu*. This server has a GPU installed and is therefore capable of executing CUDA code.
2. Clone the repository at https://github.com/cornell-zhang/parallel-programming
3. Checkout branch *jss_opt* which contains functionality of *master* but includes the ability to run the optimized CUDA code.
4. The source file I have added is located in *parallel-programming/Spam-Filter/cuda* and is named *sgd_single_point_optimized_reduce.cu*. The training function that calls this code is located in *main_cuda.cpp*.

Running the NSight Visual Profiler

To obtain the visual profiler results of section 4.1, follow the below procedure:

1. Open a terminal with x-forwarding capability (eg. MobaXTerm).
2. Run the executable located at */usr/local/cuda-7.5/bin/nsight*
3. Click on Run → Profile in the nav bar.
4. The profiler acts as a remote client, so you must login with your SSH credentials.
5. The profiler will execute the file located at *./bin/cuda* and output to the window a visual timeline after running the program.