Kungliga Tekniska Högskolan

Introduction to GPU and accelerator programming for scientific computing

Project: Optimizing Stochastic Gradient Descent with cuBLAS.

Author: Theodoros Vasiloudis
School of Computer Science and Communication

February 23, 2016

Abstract

For this project we improved an existing implementation of a parallel version of the mini-batch Stochastic Gradient Descent optimization algorithm on a GPU, using CUDA and the Thrust and cuBLAS libraries. We examine the scale of improvements we are able to get by using optimized BLAS calls for the GPU compared to unoptimized CUDA calls, and perform some large scale experiments to examine the behavior of the two approaches under different scenarios.

1 Introduction

As our project for the PDC Summer School in High Performance Computing we implemented mini-batch Stochastic Gradient Descent (SGD) using CUDA and the Thrust library. In that work, we identified a few areas that we could specifically improve in our conclusions:

1. **Dynamic Parallelism - cuBLAS for vector operations**

   SGD involves a number of vector operations, mostly dot products. In the current implementation these are done sequentially through `for` loops within kernels. By using dynamic parallelism and specifically cuBLAS operations we can significantly speed up these computations.

2. **Sequential memory access**

   In SGD one commonly shuffles the dataset in memory before each iteration, as it has been shown to improve convergence performance. In the current implementation we emulate this shuffling by randomizing the order in which we access elements of the dataset, leading to non-sequential memory accesses. What we could do instead is to shuffle the data on the GPU memory and perform sequential memory accesses.

3. **Change parallelism unit**

   In the current implementation our unit of parallelization is the size of the batch; we run one batch at a time and parallelize the computations in terms of the data points within the batch. Common batch sizes range from 10 - 1000 making this way to parallelize highly inefficient on the GPU, even if dynamic parallelism is employed. We could change the parallelism scheme of the algorithm so we can run multiple batches in parallel, thereby greatly increasing the parallelism and achieving much higher occupancy on the GPU.

In this project we focused on improvements 1 and 2, and slightly extended the scope of improvement 1, to make use of matrix-vector BLAS operations instead of only vector-vector.

For improvement 1, we developed a completely new codepath for the iterations inside the SGD algorithm, (lines 3-9 in Algorithm 1) that uses matrix-vector operations through the cuBLAS library. We also made changes to the old implementation to allow us to use vector-vector cuBLAS calls from within kernels using dynamic parallelism.

Improvement 2 came as a pre-requisite for the new codepath that makes use of matrix-vector operations, as BLAS routines assume that matrices lie in contiguous memory areas. To achieve that efficiently and to avoid having to perform expensive host-device copies at each iteration, we copy the complete dataset to the GPU once, and then shuffle it on the GPU at the start of each iteration.

The changes discussed in improvement 3 would require radical changes to the codebase which were beyond the scope of this project, and as such were not implemented.

We investigated the performance consequences that each of the improvements brought, and extended our previous experiments to include larger datasets that better investigate the performance of the implementations at scale.

The rest of this report is laid out as follows: Section 2 provides some theoretical background on the algorithm used. Section 3 focuses on parallel versions of the algorithm, providing some relevant work, and a description of the algorithm we implemented. In Section 4 we describe the main contributions of this work, focusing on how we expect each to improve upon the original implementation. In Section 5 we show an experimental evaluation of the improvements made, and in Section 6 we discuss and analyze the results.

Readers already familiar with parallel SGD and our previous implementation are free to skip to Section 4 where we present the main contributions of this work.

2 Stochastic Gradient Descent

One of the most common problems in estimation, optimization theory and machine learning is minimizing an objective function Bottou [2010]. Consider the *loss* function $Q(z, \mathbf{w})$. This function depends both on a sample $z = (x, y)$ which is a pair made of an input x and an output y and on the vector $\mathbf{w}$ which parametrizes the function that connects the input to the output. In such a context then we would like to find the vector $\mathbf{w}$ that minimizes this error averaged over our data. We can then write the following:

$$E(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} Q(z_i, \mathbf{w}) \tag{1}$$

where i the sample index, as we go through the complete dataset $\mathbf{z}$.

The Gradient Descent method proposes moving in the direction that minimizes the loss function order to find $\mathbf{w}$. That is, we perform steps of a given size in the direction of the loss function gradient and update $\mathbf{w}$ at every step:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_w Q(z_i, \mathbf{w}) \tag{2}$$

where γ controls the step size. This quantity is usually called the learning rate and controls the rate by which we change $\mathbf{w}$. The parameter γ is of crucial importance for the convergence or divergence of the algorithm. Setting it too high means taking large steps on the error surface at each iteration, making it possible for the algorithm to diverge. If we set γ too small the algorithm will not perform enough exploration, leading to slow convergence.

Under sufficient regularity assumptions when the initial estimate of $\mathbf{w}$ is close enough to the minimum and the learning rate is small enough this algorithm can achieve linear convergence Dennis Jr and Schnabel [1996].

Stochastic Gradient Descent is a variation of the scheme proposed above that can be used when the complete dataset is not available at runtime, e.g. in a streaming setting, or when the data does not fit into memory. In this case instead of taking the average gradient over the complete data we only use

the local gradient taken from a single datapoint to calculate the direction of movement, and update the weights accordingly:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \gamma \nabla_w Q(z_i, \mathbf{w}) \tag{3}$$

It follows from the algorithm that this optimization can be implemented online due to the fact that very little memory is needed for the calculation. The convergence of SGD is granted in some mild conditions when the learning rate γ decreases over time in such a way that the series indexed by them does not diverge Bottou [2010].

In Wilson and Martinez [2003] it is shown, perhaps counter-intuitively, that batch gradient descent has worse run-times and convergence characteristics when compared to stochastic gradient descent.

Mini-batch stochastic gradient descent lies somewhere in the middle between these two algorithms. There, the optimization process is performed on small subsets of the complete dataset that are called *batches*. At each step in the algorithm we choose a random subset of the data, calculate the gradient, and update the weights, based on the information we are able to get from the batch. However, the as the batch size increases we get noisier estimates of the true gradient, leading to slower convergence rate of $O(1/\sqrt{bT} + 1/T)$, where T is the number of iterations and b is the size of the batch.

The main advantage in speed for mini-batch SGD comes from the fact that instead of the vector-vector multiplications that are necessary in plain SGD, we are able to batch samples together and perform vector-matrix multiplications, allowing us to use optimized implementations of such operations e.g. BLAS (Basic Linear Algebra Subprograms) Lawson et al. [1979].

3 Parallel Stochastic Gradient Descent

3.1 Related Work

There exist a number of approaches for parallelizing SGD Zinkevich et al. [2010], Recht et al. [2011], Dekel et al. [2012], and each comes with some advantages and disadvantages. Some of these algorithms are mostly intended to be used in a distributed setting, but can however be adapted to function well in a highly parallel setting as well.

The approach presented in Zinkevich et al. [2010] is an intuitive parallel extension of SGD, where each processor solves a local SGD problem using a random partition of the complete dataset, and the solutions from each processor are then communicated and averaged. This approach involves practically no communication between the processors during the optimization phase. While this is beneficial in terms of network/communication costs, it results in subpar performance when it comes to the optimality of the solution.

Recht et al. [2011] also propose an asynchronous algorithm which is aimed at problems with sparse parameters, i.e. we assume that the cost function we are optimizing can be decomposed (to a degree). This property makes it possible for individual processors to work on different parts of $\mathbf{w}$, and perform gradient descent using only a single parameter from the complete weight vector, making it possible to atomically update that single parameter, in parallel with the other processors, without the need for locking the weight vector while one processor makes its updates. While this approach allows for fast iterations since each processor can work and update his part of the solution independently from the others, it has the additional assumption of problem sparsity and the lack of communication between

processors can often lead to diverging solutions Dai et al. [2015].

Dekel et al. [2012] show a distributed mini-batch SGD approach which they show is optimal in terms of the *optimality gap* which they define as the difference between the loss given by the optimal weight vector $Q(z, w^*)$ and the approximation obtained by the algorithm, $Q(z, \bar{w})$. The algorithm proposed uses mini-batches to calculate approximate gradients for each processor, which are then summed and averaged across the different processors, and the averaged gradient is used to update the weights at the end of each iteration. While this algorithm obviously has higher communication costs when compared to the ones we described before, it can attain better convergence guarantees, and enjoys faster convergence in terms of number of iterations, as the different processors have access to more up-to-date and consistent sets of weights. One drawback of this approach when it comes to highly parallel architectures however is that its parallelization factor is limited by the minimum of number of batches and available processors.

3.2 Algorithm Used

For this work we chose a variant of the algorithm described by Dekel et al. [2012], which involves the following steps:

1. For each iteration:
   (a) Shuffle the dataset, and split into batches of size b.
   (b) For each batch in the dataset, do in parallel:
      i. Calculate the gradient for each data point in the batch.
      ii. Sum and average the gradients for all data points in the batch.
      iii. Update the weight vector according to the average gradient.

We can write the above more formally in algorithmic form:

Algorithm 1: Mini-batch parallel SGD on a GPU

Input: A dataset **D** of containing n $z = (x, y)$ items, initial learning rate α, number of epochs T, batch size b

Output: A vector of weights providing an approximate linear solution

Initialize w
for $i \leftarrow 1$ ***to*** T **do**
  Shuffle the dataset, and split into n/b batches
  for *batch* $\mathbf{z}_s$ *over* **D do**
    for $j \leftarrow 1$ ***to*** b **do parallel**
      $\hat{g_j} = \nabla_w Q(z_j, w_i)$, where $z_j \in \mathbf{z}_s$
    $\gamma = -(\alpha/\sqrt{i}))$
    Do parallel sum $\sum_j \hat{g_j}$ to compute the average gradient for the current batch, $\bar{g_i} = 1/j \sum_j \hat{g_j}$
    Do parallel scaling of the weight vector: $w_{i+1} = w_i - \gamma\, \bar{g_i}$
return w_T

This algorithm has two differences compared to the one by Dekel et al. [2012]. The first one is that where they sum the gradients over all batches in order to reduce communication costs –the algorithm is aimed at a distributed environment– we are able to sum the gradient for the sample *within* each batch. We are able to do this due to the low communication costs that we have once all the data is loaded to the GPU, and using this approach we tie the parallelization factor to the size of the mini-batch b instead of the number of batches c. Our assumptions were that depending on the data size and parameters, it may be true that $b \gg c$, allowing for better utilization of the GPU architecture. We discussed in Section 1 why this was a bad design choice for a GPU architecture.

The second difference is that where they take a random subset of the data at each iteration, we have the assumption that the dataset will fit into the memory of the GPU, which allows us to use the more traditional *epochs* to train. An epoch is one pass over the complete dataset, and by shuffling the dataset at the beginning of each epoch we are able to get better convergence behavior as shown in Bottou [2010].

The heaviest process computationally is the gradient calculation, and it is this that we specifically targeted for our improvement through the use of Level 2 (matrix-vector) BLAS operations.

4 Improvements to the original implementation

In this section we describe the implementation tools and libraries that were used in the project, and provide an overview of the specific improvements that we made to the original implementation.

4.1 Tools and libraries

The tools and libraries used for this project are the following:

4.1.1 CUDA

The CUDA platform was first released in 2008 Nickolls et al. [2008] and since then it has allowed developers and researchers to utilize the parallel computation capabilities of GPUs for scientific computing and numerical calculations without requiring expertise on graphical programming. The CUDA API is an extension of ANSI C, where a single operation is performed on multiple cores on the GPU in parallel, a model that NVIDA calls SIMT for *Single-Instruction-Multiple-Threads*.

The CUDA API is a low-level API, and while it provides many opportunities for optimizations, it also requires programming in a very different paradigm than most programmers are used to, which can lead to many errors and un-optimized implementations. For that reason we chose to use two higher-level libraries that ease the development of programs that make use of GPUs, while sacrificing as little performance as possible.

4.1.2 Thrust

Thrust Bell and Hoberock [2012] is a parallel algorithms library aimed at increasing the productivity of programmers by offering a model of programming similar to the one provided by the standard library

on C++. In other words this switches the attention from *how* to compute something to simply *what* to compute.

One of Thrust's most important features is a vector container with both host and device implementations that allow us to use common STL operations like addition and removal of elements, as well as Thrust-specific parallel operations like `reduce`, and provides memory safety within the Thrust library.

We make use of Thrust vectors in order to store all the matrices and vectors we use in our implementation, and use operations like `transform`, `reduce` and `reduce_by_key` in order to apply SAXPY operations, vector scaling, and sums.

4.1.3 cuBLAS

The cuBLAS library provides optimized implementations of the BLAS routines aimed at Nvidia cards.

BLAS routines provide programmers with basic building blocks for linear algebra problems, like matrix-vector and matrix-matrix multiplication. There exist open-source implementations of these routines such as OpenBLAS, as well as closed-source versions optimized for specific hardware such as MKL provided by Intel, and ACML provided by AMD. cuBLAS is one such implementation provided by Nvidia that allows developers to easily tap the processing power of GPUs in an intuitive way, without having to write CUDA code.

4.2 Implementation of the improvements

As mentioned in the introduction, we took a step-wise approach to the improvements made, in order to see how each improvement affected the performance of the implementation.

Our first improvement was making very small changes to two hot-spots of the original implementation that performed dot products between two vectors and an addition plus scaling operation, commonly known as the AXPY operation in BLAS. This should allow us to achieve greater utilization of the GPU, as before the kernels were using simple `for` loops over the vectors to perform these operations. However, since these are loops that happen over the columns (features) of the data matrix, we can only realistically expect improvements for wide matrices with many features.

These vector-vector operations are happening within the GPU code itself, in the context of a kernel call, so we had to use dynamic parallelism to achieve that. Dynamic parallelism refers to the ability to launch kernels from within other kernels and is available since CUDA version 5.0 on devices with Compute Capability ≥ 3.5. It's also possible to make cuBLAS calls from within kernels which is what we did. However as we will see in Section 5, the scheduling overhead introduced by dynamic parallelism seems to have a huge negative effect on the overall performance.

Even without the overhead introduced by dynamic parallelism, the improvement gained from vector-vector operations (Level 1 BLAS routines) is limited. In order to better utilize the hardware, matrix-vector (Level 2) or matrix-matrix (Level 3) operations should be used if possible. In our case since we are dealing with a simple linear optimization problem our model is a one-dimensional vector, which limits us to Level 2 operations. For this we created a completely new codepath for the SGD iterations, that makes use of cuBLAS' GEMV (generalized matrix-vector) operations during the gradient calculation, which is the main computational bottleneck in SGD.

Using Level 2 operations however assumes that the matrix involved is laid out in contiguous areas of the memory. So in order to implement this, we needed to first implement what we have listed as improvement 2 in Section 1, namely making sure that the batches of data are laid out contiguously in memory, and accessed in order at each iteration.

The main challenge with this improvement was that we have to shuffle the dataset at the beginning of each iteration in order to improve convergence guarantees. This creates then a small tradeoff; compared to the original implementation where we access data points randomly, we now have to spend extra time at the beginning of each iteration to shuffle the dataset in the GPU memory, so that we are able to access the data sequentially and use the Level 2 BLAS routines. Our assumption was that the extra overhead of shuffling at each iteration would be counteracted by the greatly increased speed in computation due to the Level 2 BLAS routines and sequential access of the data, which was confirmed by our experiments shown in Section 5.

We use Thrust in order to perform this permutation, with the caveat that we maintain a copy of the original data in memory and shuffle the data by performing a permuted copy to another destination matrix/vector on the GPU. This means that we have to use twice the amount of memory to store our data, a limitation that could be overcome with an alternative that shuffles the data in place, but did not have the time to implement for this project.

Once we had improvement 2 in place we could proceed with creating the new codepath that utilizes Level 2 BLAS operations to calculate the gradients. We should note that for this codepath we are not using any CUDA device code, but rather only use either cuBLAS or Thrust calls for all the calculations, which we believe greatly improves the readability and maintainability of the code.

5 Results

In this section we will demonstrate and briefly analyze the results of our experiments. Since this project is about performance improvements compared to an existing implementation, we will focus on the time it takes to run the same problem using the different implementations. The differences between the implementations in terms of correctness and accuracy are virtually non-existent, accounting for numerical precision and the randomness introduced by the data shuffling.

We will show how the different implementations scale in term of the number of samples and features in the dataset, and the size of the we set. For easy reference we present these parameters of interest in Table 1.

Parameter	Description	Symbol
Batch Size	The size of the mini batch	b
Number of Samples	The number of training examples	N_s
Number of Features	The number of features for each example	N_f

Table 1: Main parameters for scalability testing

Benchmarking environment

The experiments described below where performed on a desktop PC, running Ubuntu 14.04 64bit. The code was compiled with CUDA version 7.5 which includes Thrust 1.8.2 and cuBLAS 7.5.18.

The hardware characteristics were: 4GB of RAM, 4-core Intel i7-870 @ 2.93Ghz (Nehalem architec-

ture), and the GPU used was an Nvidia GTX 750Ti, a Maxwell chip (GM107) which has a Compute Capability 5.0 and comes with 4GB of GDDR5 VRAM installed.

Description of The Experiments

In order to test our implementation of SGD we use it to estimate the weights for a linear regression problem which can be described as follows:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} ||\mathbf{y} - \mathbf{w}\mathbf{X}||_2^2 \tag{4}$$

that is, we try to find $\hat{\mathbf{w}}$ such that it minimizes the squared loss between the dependent variable $\mathbf{y}$ and $\mathbf{w} \cdot \mathbf{X}$, where $\mathbf{X}$ is our data matrix, with dimensions $N_s \times N_f$.

In order to test the performance of our program in a reliable way we generate our own artificial data sets. This ensures reproducibility with enough variability and furthermore consistent access to the ground truth and we ensure control over N_s and N_f to test how our algorithm scales.

In order to generate experiments that scale consistently with different data set sizes we used subsets from a bigger data set. That is, we first build a data set with the maximum data size that we want to test and then reduce it by taking pieces of arbitrary data sizes. This ensures that the convergence characteristics of the dataset –or "difficulty" to learn the weights– should be consistent across the differently sized datasets, since they originate from the same generating process.

Experiment implementation

In order to ease the set up and running of the experiments we used a couple of different languages and libraries. First create the test datasets using Python and the `skcikit-learn` library. This allows us to easily vary the size of the generated data in terms of number of samples and features, and we are able to quickly create datasets with ranges of values. We then use a Python script that launches the CUDA program which creates a JSON output file for each parameter setting, reads the results from the files, and creates the plots for the experiments.

This automated way to perform experiments allowed us to easily run our experiments, and spend more time exploring the performance of the implementations.

5.1 Comparison between plain CUDA and using dynamic parallelism

For our first set of experiments we used relatively small datasets to perform comparisons between all three implementations of the algorithm. We tested the scaling characteristics of each in terms of number of samples, number of features and the batch size used.

For the sample number experiments we created datasets with a number of samples ranging from 50,000 to 100,000 with 4 columns/features and used a batch size of 1000, learning rate of 0.01, and 10 training epochs.

For the feature number experiments we created datasets with a 10,000 samples and number of features ranging from 200 to 1,000 with and used a batch size of 1000, learning rate of 0.1, and 20 training

epochs.

For the batch size experiments we created a dataset with 100,000 samples and 1000 and used a batch size ranging from 200 to 1200, with a learning rate of 0.1, and 20 training epochs.

Scaling in terms of number of samples

We will first investigate the runtime performance of the original, plain CUDA implementation and the one using Level 1 cuBLAS calls through dynamic parallelism.

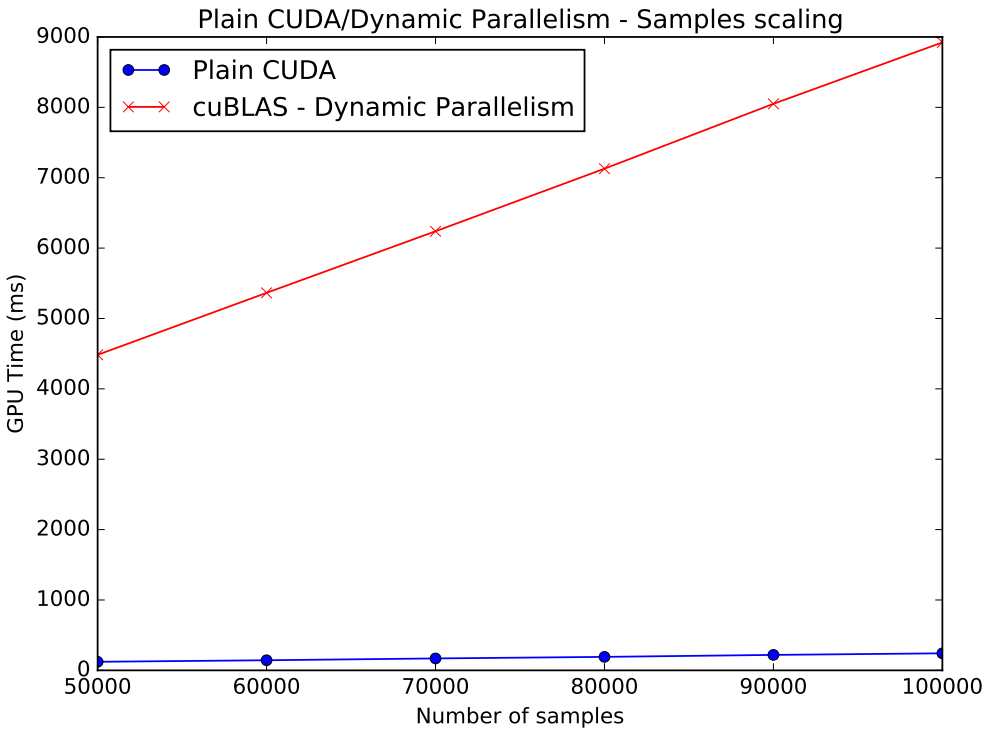

Figure 1: GPU time as a function of number of samples for plain CUDA vs. cuBLAS with dynamic parallelism.

It becomes immediately evident that our use of dynamic parallelism introduces a significant bottleneck causing the runtime to increase by two orders of magnitude. We haven't been able to trace the source of this performance degradation, but we have verified that our use dynamic parallelism is according to the recommendations set by Nvidia. In our implementation as mentioned in Section 4 all we did was change two for loops within a kernel, one to use the cuBLAS `DOT` operation and one to use the `AXPY` operation. Our current assumption is that the scheduling of the threads is ruined due to constant synchronization within the calling kernels.

Also, as we mentioned in Section 4, for a small number of features like the 4 used in this experiments we don't expect any improvement as there is very little to do in parallel. We investigate what happens when many features are used in the following experiments.

Scaling in terms of number of features

For larger number of features, the difference becomes less pronounced, but remains largely similar as before.

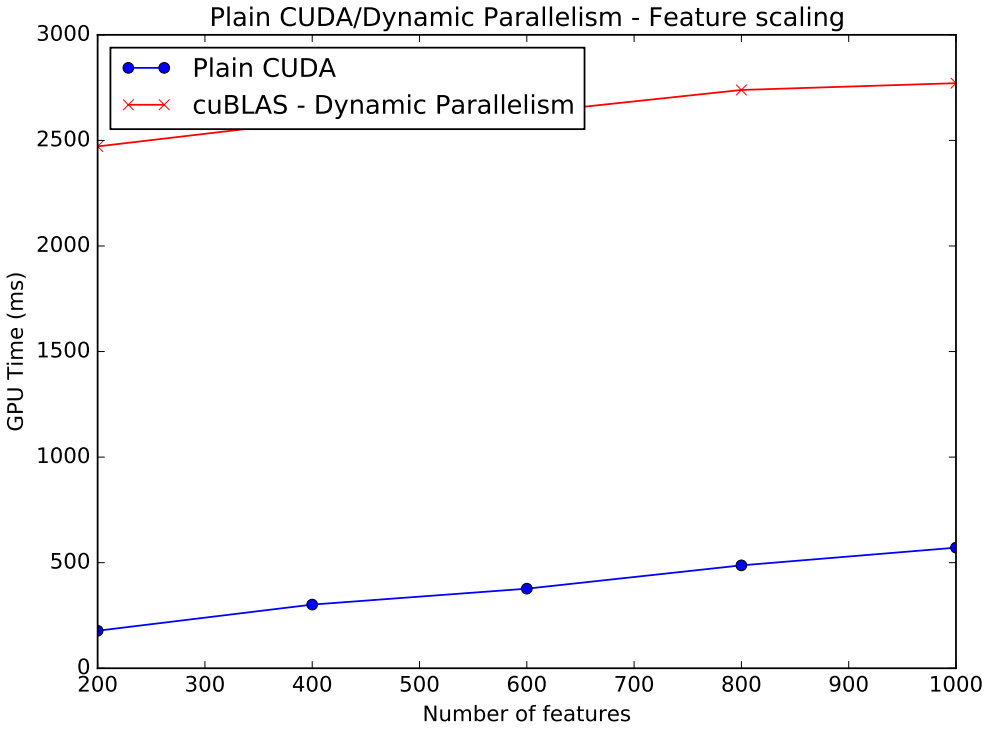

Figure 2: GPU time as a function of number of features for plain CUDA vs. cuBLAS with dynamic parallelism.

With this in mind, we cannot claim that the performance degradation comes from a lack of parallelism potential. The last potential cause we wanted to investigate was having a lack of parallelism per block, which we do next.

Scaling in terms of number of batch size

We wanted to investigate whether the large batch size (1000) that determines the unit of parallelism in our implementation interfered with the scheduler's ability to schedule new threads to perform the dynamic parallelism operations. Since each block can run at most 1024 threads, it could be possible that we were "filling" each block with the parallel threads for the batch, leaving no threads free to run the vector operations in parallel. By using smaller batch sizes we might be able to launch more parallel threads for the vector operations. We can see the results of these experiments in Figure 3.

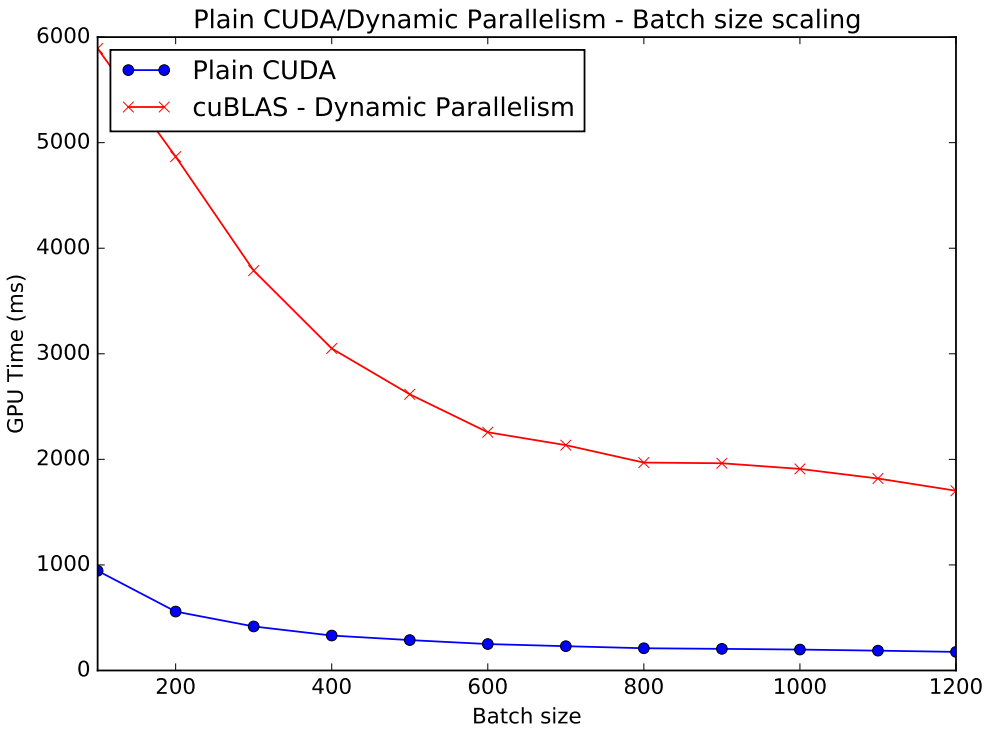

Figure 3: GPU time as a function of batch size for plain CUDA vs. cuBLAS with dynamic parallelism.

We can see in Figure 3 that as the batch size increases the difference becomes less pronounced, but it still remains at around an order of magnitude. So with something obviously going wrong with our implementation of dynamic parallelism, in the rest of the experiments we will focus on the differences between our original implementation and the new one using Level 2 cuBLAS routines.

5.2 Comparison between plain CUDA and using cuBLAS Level 2 routines

We start our comparison of the implementations using the relatively small scale datasets we used in the previous experiments.

Scaling in terms of number of samples

We take a look at the scaling in terms of number of samples first. Our expectation is that there shouldn't be a large difference between the two, because given the very small number of features (4), there is not a lot of room for the Level 2 routine to operate.

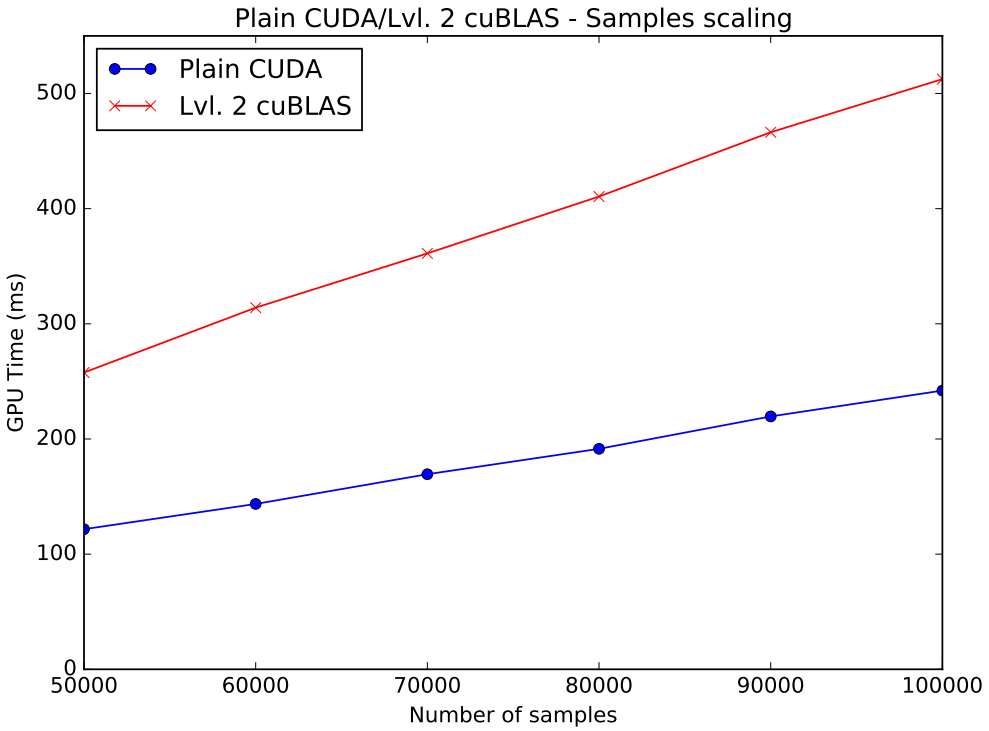

Figure 4: GPU time as a function of number of samples for plain CUDA vs. Lvl. 2 cuBLAS.

Our assumptions were confirmed from the experiments as shown in Figure 4. The overhead introduced by the shuffling of the data and the use of the cuBLAS routine results in longer runtimes compared to the original implementation.

We investigate what happens when large numbers of features are used in the following experiments.

Scaling in terms of number of features

For larger number of features, there should be enough level of parallelism so that the optimized Level 2 routines can make up for the overhead introduced by the shuffling.

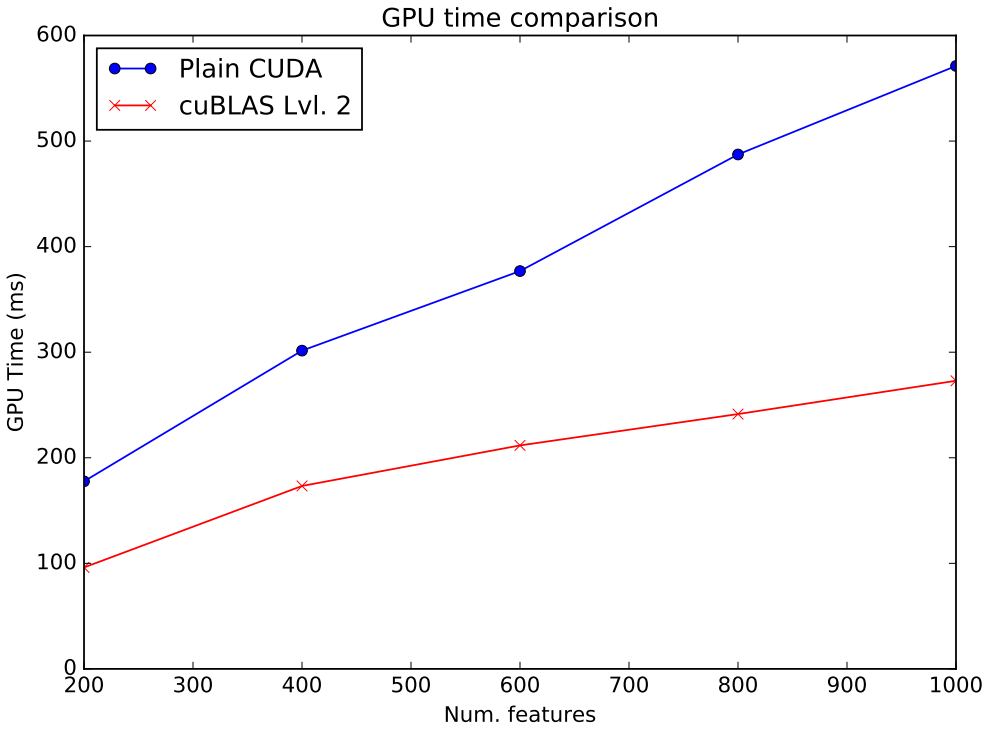

Figure 5: GPU time as a function of number of features for plain CUDA vs. Lvl. 2 cuBLAS.

Figure 5 illustrates exactly that. The performance gain from the added parallelism in terms of features allows the cuBLAS code to run approximately twice as fast, and maintain better scaling characteristics as well (runtime increases ∼2x for 5x features, vs. an increase of ∼3x for the original implementation for the same increase in features).

Scaling in terms of number of batch size

For the batch size experiments we mentioned we only used 256 features to allow for parallelism "room" for the dynamic parallelism implementation. As a result, the differences between the original implementation and the Level 2 cuBLAS code are less pronounced, and become smaller as the batch size increases, as shown in Figure 6.

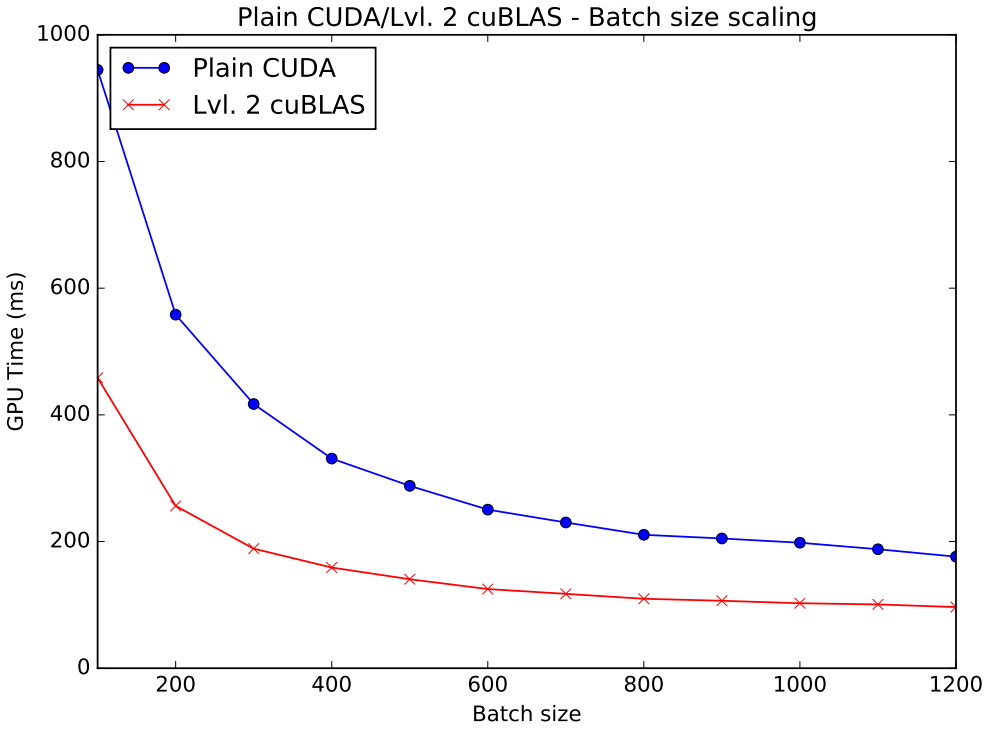

Figure 6: GPU time as a function of batch size for plain CUDA vs. Lvl. 2 cuBLAS.

5.3 Larger scale experiments

After observing these performance differences, we finally wanted to investigate two things: how do the two implementations differ when it comes to large datasets, and how does the performance change when we change the shape of the data matrix, but keep the number of elements the same. We summarize these results in Table 2:

	$100k \times 1k$	$10k \times 10k$	$200k \times 2k$
Level 2 cuBLAS	1488.73	855.02	4417.53
Plain CUDA	2832.07	2286.13	20819.90
Relative difference	$\times$**1.9**	$\times$**2.7**	$\times$**4.7**

Table 2: Comparing runtimes in ms for plain CUDA and Lvl. 2 cuBLAS for large scale data.

We can see that for the same number of datapoints, as the table becomes wider the Level 2 cuBLAS implementation has bigger gains, which is expected since the `for` loops in the original code are performed over the columns in the dataset. The final column of the table shows that as the overall size of the table increases, the gains for the cuBLAS implementation become more pronounced. We should note that at $200k \times 2k$ the size of the data table in memory becomes $\sim$1.5GB, and with the doubling of the data size mentioned in Section 4 the memory usage for the GPU in the cuBLAS implementation goes to $\sim$3.0GB, out of 4.0GB available in the GPU we used. If we want our implementation to scale to bigger data sizes then an in place data shuffling method becomes necessary.

6 Discussion

The results from using dynamic parallelism were unexpected and further investigation is needed to determine if it is our implementation that is suboptimal, or if there is an inherent performance penalty to using cuBLAS calls in such a manner. Using Level 2 cuBLAS calls however brought the performance benefits we expected, especially as we scale up to larger problem sizes.

We would like to note again that by using the Thrust and cuBLAS libraries, we were able to implement the same algorithm without writing any CUDA code. Our experiments show then that using high-level BLAS routines has the potential to improve performance while at the same time simplify the code and make it more readable and maintainable.

We believe that when the hotspots of a program are basic linear algebra routines, achieving better performance by hand-crafted CUDA kernels is very hard without specific assumptions about the data (like the width of the matrix), and the performance gain vs. maintainability of the code is a tradeoff every developer has to consider but is often overlooked. For large projects that span multiple years and developers, giving up some potential performance for a dramatic increase in code readability and maintainability is often the better choice.

In terms of future work we would like to continue improving this implementation as a way to learn event more about GPU programming. As mentioned in the introduction, it suffers from the bad design choice of using the batch size as the unit of parallelism, which is a major drawback, and will take significant changes to the codebase to fix. Also, in our cuBLAS code we are needlessly using twice the amount of memory needed to hold the data, which could also be addressed without major changes to the codebase. Another potential improvement is overlapping file IO with computation. Currently the most time consuming part of an experiment is reading the file from disk. If we instead overlap the reading of file chunks with running the computation we could achieve some more time savings.

References

Nathan Bell and Jared Hoberock. Thrust: Productivity-oriented library for cuda. *Astrophysics Source Code Library*, 1:12014, 2012.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P. Xing. Analysis of high-performance distributed ml at scale through parameter server consistency models. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, 2015.

Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *The Journal of Machine Learning Research*, 13(1):165–202, 2012.

John E Dennis Jr and Robert B Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*, volume 16. Siam, 1996.

Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

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.

D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003.

Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.