Fast On-line Statistical Learning on a GPGPU

FangZhou Xiao[1] **Eric McCreath**[1] **Christfried Webers**[1,2]

[1] School of Computer Science, College of Engineering & Computer Science
The Australian National University,
Canberra, ACT 0200, Australia,
Email: `Shaw.Xiao@anu.edu.au`, `Eric.McCreath@anu.edu.au`

[2] NICTA,
Canberra, ACT 0200, Australia
Email: `christfried.webers@nicta.com.au`

Abstract

On-line Machine Learning using Stochastic Gradient Descent is an inherently sequential computation. This makes it difficult to improve performance by simply employing parallel architectures. Langford et al. made a modification to the standard stochastic gradient descent approach which opens up the possibility of parallel computation. They also proved that there is no significant loss in accuracy in their approach. They did empirically demonstrate the performance gain in speed for the case of a pipelined architecture with a few processing units. In this paper we report on applying the Langford et al. approach on a General Purpose Graphics Processing Unit (GPGPU) with a large number of processing units. We accelerate the learning speed by approximately 4.5 times compared to a standard single threaded approach with comparable accuracy. We also evaluate the GPU performance for the sequential variant of the algorithm, which has not previously been reported. Finally, we investigate how changes in the number of threads, number of blocks, and amount of delay, effects the overall performance and accuracy.

Keywords: GPGPU, Asynchronous Optimisation, Statistical Machine Learning, On-line Learning

1 Introduction

Parallel architectures, such as the GPU or multi-core systems, are set to take over traditional serialised architectures given they facilitate a path to continued performance improvement. Given the GPU's outstanding floating point performance, it is a low cost solution for high-performance computing. Furthermore given the widespread deployment of graphics cards capable of CUDA or OpenCL, writing HPC applications on a GPU has become a very attractive option.

This work was supported in part by the Australian Research Council grant DP0987773.
NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.
We would also like to thank Alistair Rendell and Alexander J. Smola for their guidance and encouragement with this project.

GPUs have shown to be outstanding for some scientific modelling applications (Stone et al. 2007, Collange et al. 2007) by achieving surprising acceleration. Steinkraus et al. (2005) claimed that GPUs are pleasant substitutions to dedicated machine learning hardware, such as analog chips and coarse-grained parallel computers. Compared with CPUs or even multi-core CPUs, GPUs still exhibit advantages in many applications (Raina et al. 2009). Previous work on machine learning using GPUs mainly obtained acceleration through intrinsic parallel structures in applications. For example Raina et al. (2009) experimented with large-scale deep unsupervised learning on GPUs and Steinkraus et al. (2005) applied GPUs to a two-layer fully connected neural network. Generally, GPUs (or even other parallelised architectures) support applications which consist of highly symmetric and loosely coupled calculations. These applications are naturally parallelisable. Unfortunately, many applications also involve critical sequential blocks, which can not easily be parallelised. Such difficulty limits the performance of modern parallelised architectures. This paper tentatively applies the GPU's computational capability to one such sequential algorithm: on-line statistical machine learning using gradient descent. Although GPUs might not be the most suitable platform available to address sequential issues, other parallelised architectures more or less face similar issues of efficiency when utilising parallel computation capabilities.

Machine learning is concerned with the design and development of models and algorithms that allow computers to improve their performance over time based on data. Depending on whether or not all training data are used at each iteration step of the algorithm, one can distinguish between batch and on-line learning.

Batch machine learning approaches evaluate candidate hypotheses against the entire set of training instances. This can be very slow when the training set is very large. Furthermore, as all the data must fit into the memory, batch learning utilising large datasets is not well suited for GPUs which have a small local memory.

On-line learning takes one instance of data at a time, and improves its performance solely based on this data item. This process iterates through all the data (possibly a number of times) until either some convergence criteria are achieved or the model predicts unseen test data sufficiently accurately. As only the model parameter and one data item at a time have to be kept in memory, on-line learning better suits the architecture of GPUs. This is a great advantage especially when the training set is very large.

The serial nature of on-line approaches means that they are difficult to parallelise. This is because the calculation of hypothesis x_{i+1} requires both the input of instance z_i, training label y_i as well as the previous hypothesis x_i. This entire process is depicted in Figure 1.

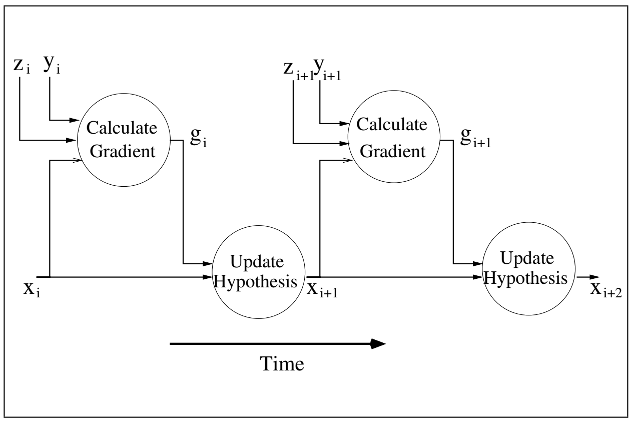

Figure 1: Data flow for the stochastic gradient descent algorithm. In order to calculate the hypothesis x_{i+1}, one data item z_i, training label y_i, as well as the previous hypothesis x_i are required. A stage can only start its calculations if the previous stage has finished all computations.

Langford's (Langford et al. 2009) paper "Slow Learners Are Fast" modifies the standard gradient descent algorithm permitting the calculation of the gradient to use a delayed version of the hypothesis. This opens up the possibility of concurrently executing the calculation of the gradient. This is depicted in Figure 2 which shows the dependencies when the gradient calculation uses a delay of 1.

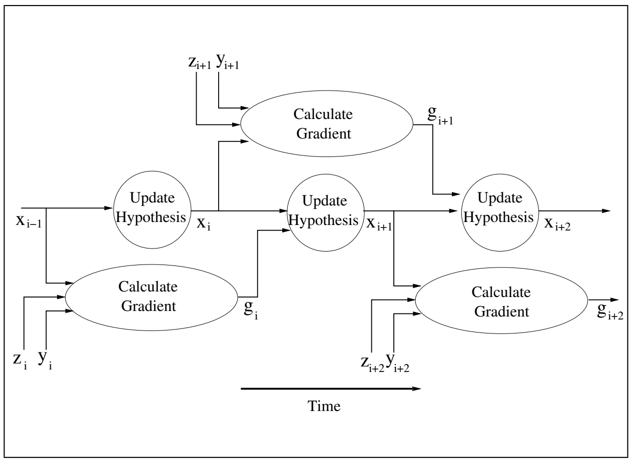

Figure 2: Data flow for delayed stochastic gradient descent algorithm. With a delay of 1, two threads can run concurrently. Thread A (the lower one) starts to calculate gradient g_i immediately after hypothesis x_{i-1} is available. Before it updates the hypothesis x_i with g_i, thread B (the upper one) uses the hypothesis x_i to calculate the gradient g_{i+1}. As can be seen from the data flow, gradient calculation in the threads and updating hypothesis can be run in parallel.

Clearly a new gradient calculation can be started after each hypothesis update. That means if the delay τ is larger than one, potentially τ threads can be run in parallel. On the other hand, if the delay is too large, the gradient updates become outdated because they are based on too old hypotheses. This limits the number of the delay τ from above.

Langford et al. provide the theoretical foundation for our series of experiments. They prove convergence properties of the convex minimisation problem. Yet they do not experimentally show the performance improvement on an actual parallel architecture. Our experiments implement the delayed mechanism on the TREC dataset (Cormack 2007), and verify the utility of this approach.

The stochastic gradient approach assumes that the training instances are independent and identical distributed (iid) data, this will generally require the training set to be randomly shuffled before being provided to the learner. This opens up the possibility of allowing the reordering of examples when they are used within the learner, which in turn enables us to arbitrarily allocate instances to threads without tightly enforcing an ordering via synchronisation.

The delayed approach trades accuracy with parallelism. Using this parallelism should make the program run faster, however, if you need to run more steps to gain the same accuracy then the speed gained via parallelism needs to outweigh the accuracy losses. Otherwise it is better to just run the standard serial code. We have explored and reported the effect on accuracy in this paper.

There are a number of challenges in implementing the delayed stochastic gradient descent algorithm using a GPU. They include:

1. synchronisation is difficult to implement across all the blocks,
2. synchronisation has the potential to be costly in terms of performance,
3. the GPUs memory size is relatively small, and
4. it is slow to transfer data between the host and the device memory.

2 Delayed Stochastic Gradient Descent

This problem is one of binary classification problems. Email t is denoted $\boldsymbol{z_t} \in Z$ and given the label $y_t \in \{\pm 1\}$, so if $y_t = -1$ then the message $\boldsymbol{z_t}$ is labelled as spam, whereas, if $y_t = 1$ then the message $\boldsymbol{z_t}$ is labelled as not spam. Z belongs to an n dimensional space and has a corresponding n dimensional feature space $X \subseteq R^n$. This feature space contains the hypotheses we intend learning. To determine the classification of a new message $\boldsymbol{z}$ we simply take the inner product between the message and that of our current hypothesis $\langle \boldsymbol{z}, \boldsymbol{x} \rangle$ if this is negative then the message is predicted to be spam and if the inner product is positive then the message is predicted to be not spam.

The loss associated with email t using hypothesis $\boldsymbol{x}$ is $l(y_t\langle \boldsymbol{z_t}, \boldsymbol{x} \rangle)$. The smoothed quadratic soft-margin loss function is used:

$$l(\chi) = \begin{cases} \frac{1}{2} - \chi & \text{if } \chi < 0 \\ \frac{1}{2}(\chi - 1)^2 & \text{if } \chi \in [0, 1] \\ 0 & \text{otherwise} \end{cases}$$

The aim is to find a hypothesis $\boldsymbol{x}$ that minimises the sum of the loses over all the training instances. The basic idea of gradient descent, also known as steepest descent, is using a single example move the hypothesis $\boldsymbol{x}$ in the direction of the negative gradient. This is repeated over all the instance of the training set over a number of repetitions. An annealing schedule

η controls the convergence speed. The amount of delay used is denoted $\tau \in N$. Langford et al. (Langford et al. 2009) proved that if the delay is within a tolerable range, delayed stochastic gradient descent would converge to an acceptable value. The convex function used for calculating the gradient is:

$$f_t(\boldsymbol{x_t}) = l(y_t\langle \boldsymbol{z_t}, \boldsymbol{x_t}\rangle)$$

The algorithm for stochastic gradient descent with the delayed mechanism is given in the following steps:

1. Initialise weight vectors $\boldsymbol{x_1}, ..., \boldsymbol{x_\tau} = \boldsymbol{0}$
2. Compute the gradient:

$$\begin{aligned}\boldsymbol{g_t} &= \bigtriangledown f_t(\boldsymbol{x_t}) \\ &= \boldsymbol{z_t}\frac{\partial\, l(y_t\langle \boldsymbol{z_t}, \boldsymbol{x_t}\rangle)}{\partial\, \langle \boldsymbol{z_t}, \boldsymbol{x_t}\rangle} \\ &= \begin{cases} -y_t\boldsymbol{z_t}, & \text{if } \langle \boldsymbol{z_t}, \boldsymbol{x_t}\rangle \le 0 \\ y_t(\langle \boldsymbol{z_t}, \boldsymbol{x_t}\rangle - 1)\boldsymbol{z_t}, & \text{if } \langle \boldsymbol{z_t}, \boldsymbol{x_t}\rangle \in [0,1] \\ \boldsymbol{0}, & \text{otherwise.} \end{cases}\end{aligned}$$

3. Update $\boldsymbol{x_{t+1}} = \boldsymbol{x_t} - \eta_t \boldsymbol{g_{t-\tau}}$.
4. Repeat Steps 2 and 3.

Step 3 in the above algorithm is based on the assumption that the weight vector has not been changed greatly after τ delays, so we can update the weight vector by using the delayed gradient. As the number of instances processed increases the annealing schedule, which is $\eta_t = \frac{1}{t-\tau}$, becomes small. This provides a guarantee that the weight vector will only change slowly.

Regarding to simplicity of implementation and consistency with Langford's experimental settings, we used dot product in Euclidean spaces to calculate inner product mentioned.

3 Implementation Issues

This section generally covers concerns or possible issues if researchers try to replicate our experiments. Regarding to GPU hardware restrictions, we have to make changes to the codes running on the GPU. Where possible we kept similar experimental settings to that of Langford et al. (2009), if any other issues haven't been mentioned in the paper.

3.1 Machine Learning Issues

Our training set, the TREC dataset (Cormack 2007), consists of 75419 labeled e-mail messages. This dataset has three subsets, "full", "partial" and "delay". The "delay" part was used because it was large enough to properly evaluate the implementation, yet, small enough to still be able to run tests in a reasonable amount of time. In order to transform raw messages into manipulable data, we conducted a separate program from training codes to extract relevant information, called pre-processing. Note that this part of program had not been evaluated, only the performance of training codes was concerned. Information was extracted from the email header fields 'From:','To:','Subject:', and 'Time:', and also from the body of the email. Symbols other than alphabetic letters and numbers, such as ASCII code less than 48, were removed from the text. Then rest text was consisted of words that were separated by space. All the word were capitalised and recorded into a word dictionary. Langford's experiment used both the bag of words and the bag of words pairs representation. For simplicity reason we only adopted the bag of words representation. The sparse format of the message representation was transformed to a condensed format which listed just the features appearing within the email.

However techniques in text classification like excluding most frequently occurring words, such as "a", "the" or words that only appear once from the word dictionary, had not been applied. Therefore less preknowledge or less artificial intervention was involved into the experiment. Without specific settings for text classification, our results might be meaningful for general machine learning problems. Furthermore our experiment results shown that weights of these features had very limited influence on classification results. Alternatives of these settings may affect accuracy. However, as our focus is on comparing the GPU implementation with that of a standard CPU approach, comparing both speed and accuracy. The relative performance, rather than absolute performance, of the two approaches is more informative.

To fulfil the assumption that input data are iid data, when messages were loaded, we randomised the data by repeatedly swapping randomly selected messages. This provided us with some confidence that the data provided to the learner was not correlated. For evaluation, we used the standard ten-fold cross validation to calculate average learning results.

3.2 Data Structure on the GPU

In order to fit the experimental data into the GPU memory, we re-arranged some data structures. The message matrix that records feature indices was transformed into a long vector, called "datalist". The start positions of each message was recorded in another vector called "positionlist", with which we can easily extract the message from the long vector "datalist". Target values of messages were put into a third vector called "targetlist". The GPU has fast but small constant memory. We stored "positionlist" and "targetlist" into the constant memory. Because the information of "positionlist" and "targetlist" is frequently required, by storing it in the constant memory will significantly increase efficiency. The vector of feature weights was stored in global memory. Although this memory is very slow, global memory was the only space where all threads can read and update data in our current understanding. Furthermore for the same reason two vectors of "gradient" and "messageid" in global memory were used to store implicit gradient information. The size of "gradient" depended on the delay parameter.

Since the GPU used a different memory system with that of the host CPU system, thus we needed to copy data from the host memory to the device memory and copy results from the device memory back to the host memory after computing. We had to minimise this cost by avoiding frequently transferring data between the host and the device.

3.3 Coding

In order to get benchmark results for evaluating the GPU's performance, we wrote equivalent programs for both the CPU and the GPU. Both codes are written in C, and the CUDA API was used for calling the kernel executed on the GPU. We used an asynchronous approach to parallelise most parts of the program.

In our program, we can change the parameters of iteration number, delay amount, grid size and block size (the total number of threads running is the product of the grid size and the block size) to measure the training time and the error rate. Repetitive learning was run over many iterations on the same dataset as indicated. Because every thread does the similar tasks, we could explain one thread as representative.

The thread loaded features of one message and the corresponding weights from the global memory. After calculating the gradient, this thread wrote the gradient and the message ID into "gradient" and "messageid" vectors. The index of the vectors was decided by the thread's unique id. Then the thread updated the delayed gradient stored in "gradient" and "messageid" vectors (delay τ times ago) to the global weight vector and cleared the outdated data.

Assuming every thread handles one parallel computation, then the thread size as Langford suggested is delay τ plus 1. For example if the delay is zero, then at least one thread should be running and thus the delay mechanism is disabled. Note that total thread size did not necessarily equal to delay τ plus 1. Total thread size is a number within the range of one and instances size. Delay size is between zero and the total thread size.

3.4 Scheduling

The delayed update mechanism required a well organised read and write schedule. However it was difficult for the GPU to keep such an organised schedule, because direct communications between threads is not easily available (although global memory could be used, it would be very slow). One thread that starts processing messages earlier cannot promise to finish earlier. The thread does not care about other threads' status (especially threads in different blocks). In most cases, such a schedule was chaos and uncontrollable, especially when the disabled delay mechanism was used. Variances in the delay built up when the program had been run for some time. The delay mechanism would help to keep the schedule organised, because the more delay we assign, less threads would try to access the same shared data. Also each thread cleared its own "gradient" and "messageid" spaces after updates. For every thread, it required the results from $t-\tau$ thread but it will not wait on previous threads. By clearing "gradient" and "messageid" spaces after every update, we can at least make sure that if a thread updated results earlier than previous threads finishing computing, it would at least make no changes to the final result.

4 Results and Evaluation

This section illustrates our experimental results regarding to the CPU and the GPU respectively. We evaluate the performance of the algorithm running on the CPU as benchmarks and investigate how changes in the number of threads, number of blocks, and amount of delay, effects the overall performance and accuracy. We also discuss the limitations of our implementation on the GPU.

The graphics card used is the NVIDIA Geforce GTX 295 and it consists of two identical GeForce GTX 200 GPUs. Each of these GPUs has 30 Streaming Multiprocessors and each of the Streaming Multiprocessors has 8 cores. We introduce SM as an abbreviation for Streaming Multiprocessors. Each of the GPUs has 895M global memory and 64K bytes constant memory. Execution blocks have 16K of shared memory and 16K of registers. The GPU runs at 1.24 GHz and has a CUDA capability of v1.3. The host machine uses a AMD Phenom^{TM} II X4 945 processor with 4GB of main memory. For simplicity we describe the NVIDIA GPU card as the 'GPU' and the AMD host as the 'CPU'.

4.1 Experiments on the CPU

Figure 3 shows the results of Langford's experiment based on "full" dataset. Our results based on both "full" and "delay" dataset were shown in Figures 4 and 5. In order to show curves clearly, we used different scales of x-axis from Langford's figure.

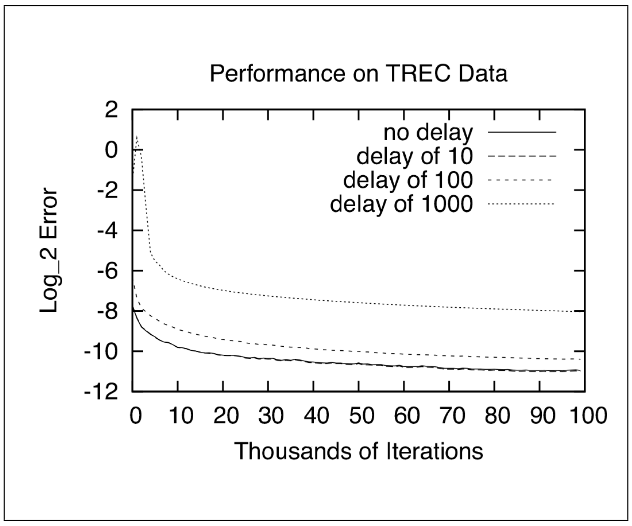

Figure 3: Results of Langford's experiment on the full dataset. The curves of relatively small delay (compared with intance size and iteration times) are close to the curve of zero delay. The error rate increases when delay size increases.

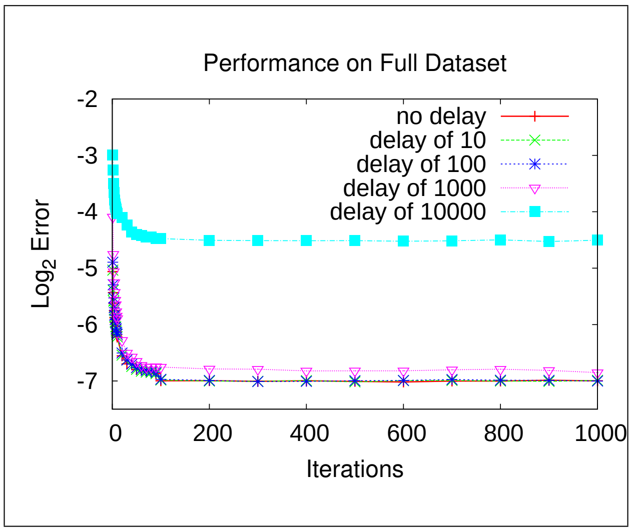

Figure 4: Performance of our experiment on the full dataset. Our results exhibit similar trends as Langford's results. For a relatively large delay, the accuracy will be significantly affected.

In order to test that the learner was working properly, we created an artificial dataset. This artificial dataset includes some known positive and negative keywords. Through learning, the induced weight vector showed that these keywords had a much larger value than that of other less important words. Fur-

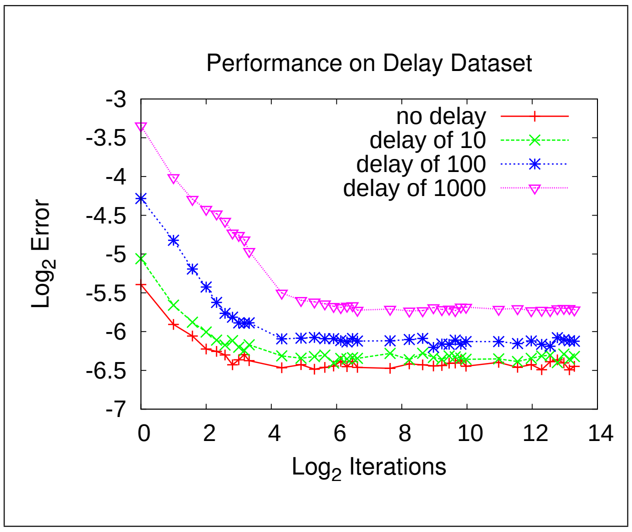

Figure 5: Performance of our experiment on delay dataset. This a similar experiment on a smaller dataset.

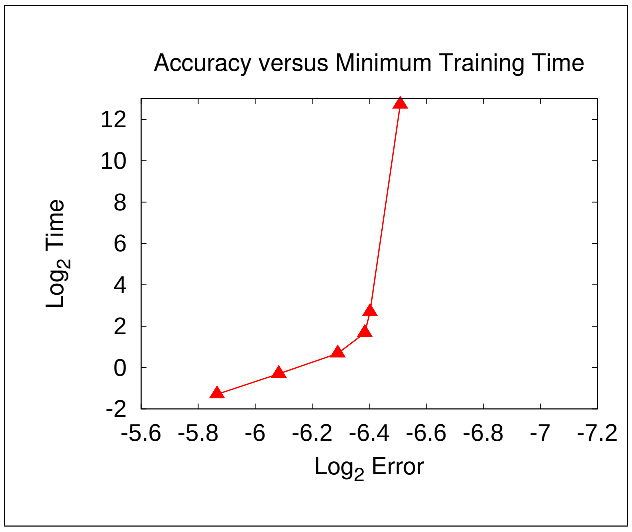

Figure 6: Example of minimum CPU sequential execution training time required to achieve different accuracy levels based on delay dataset. To achieve higher accuracy will require much more computations.

thermore, positive and negative keywords had opposite signs. Frequently occurring features ended up with small weights, which barely contributed to classification results.

Then we changed some target values of messages (adding some outliers) or exchanged position of messages pairs, the results showed that accuracy was affected in the first few iterations, but gradually converged to a similar accuracy level. Therefore we are confident that our algorithm, was not only able to distinguish important features from trivial ones, but also ignored outliers to some extend. Also we found that if the input data was large, the effect of delay could be minimised if we ran many repetitions in the learning.

Nevertheless our results did not exactly follow that of Langford's results, since there were a few different settings used. Langford used the bag of word pairs as well as the bag of words to represent features, whereas, we only used bag of words for simplicity reason. We believe that if we adopted alternative settings, the error rate of classification could be much lower and more steady. Nevertheless all these results show the same trends: if the number of iterations increases, the error rate presents more steady and lower; if the delay amount increases, the less accuracy of converged results would achieve. Under our assumption that this experiment was a relative comparison, if we used our results of the same settings on the CPU as a benchmark, we could still explore the character of the GPU.

We tested both results that were calculated by single precision and double precision under our settings. The results showed that there were no obvious changes over the classification results within one hundred thousand iterations. The weight of each feature could be affected on the 10^{-5} scale, but overall classification results remained the same. In this case, if the data clusters were distinctive, then such accuracy loss was tolerable. However in other applications, the precision of parameters may be more significant.

Figure 6 shows relationship between accuracy and minimum training time required. As we see, to gain higher accuracy will incur much more computations. It is beneficiary if GPUs can accelerate this procedure while they still be able to keep comparable accuracy.

We ran codes both on the CPU and the GPU using one thread on the "delay" subset without setting delay. The results showed that the CPU's efficiency was roughly 35 times higher than that of the GPU (without considering a constant memory transfer time). Regardless of other facets, in order to gain acceleration from current settings, we ought to parallelise more than 35 threads (actually much more) on the GPU. If the CPU running time is less than the memory transfer time, then there is no point in using the GPU for acceleration. Only if the CPU's running is much greater than the memory transfer time, could we possibly gain acceleration. Longer CPU running time is related to more repetitive learning or a larger dataset.

4.2 Experiments on the GPU

We outlined the performance of the GPU through changing parameters of delay, iteration, grid size and block size. Based on the knowledge of these parameters, we drew cures of acceleration that we could achieve in our understanding. More elaborate optimisation is still achievable.

4.2.1 Delay

According to Figure 4 and Figure 5, with iteration of eight hundred times, all curves of delay have converged to steady states. Therefore most of our experiments are tested under 800 iterations. Here we test results according to changing delay given total running threads as shown in Figure 7 and Figure 8. We set the grid size to 30 (equals to number of Streaming Multiprocessors, thus each SM execute one block) and the block size to 320 (10-fold of warp size). Note that memory transfer time is not considered. Figure 7 shows that accuracy increases when the delay goes up to 32. After that accuracy drops gradually when delay keeps increasing. Because when delay is smaller than 32, one warp of execution will definitely have two or more threads trying to access the same delay space. We cannot assure that former threads will finish computing and write back before later threads started to read. Yet if the delay is over the warp size, during one warp execution, in most cases, one thread accessed one delay space and all threads will finish computing before next warp execution. However in terms of Langford's delay hypothesis, the proper delay should be much higher than 32 in this case. Using delay of 32 actually updated results earlier than expected. Figure 8 shows how execution time is affected by changing the delay. If the delay is smaller, then less processing time would be consumed. A big delay space will result in more cache misses, whereas small delay seemed to be more efficient. This indicates that with a delay of 32 we will help to achieve the fastest processing speed without affecting accuracy, yet this does not strictly follows Langford delay hypothesis.

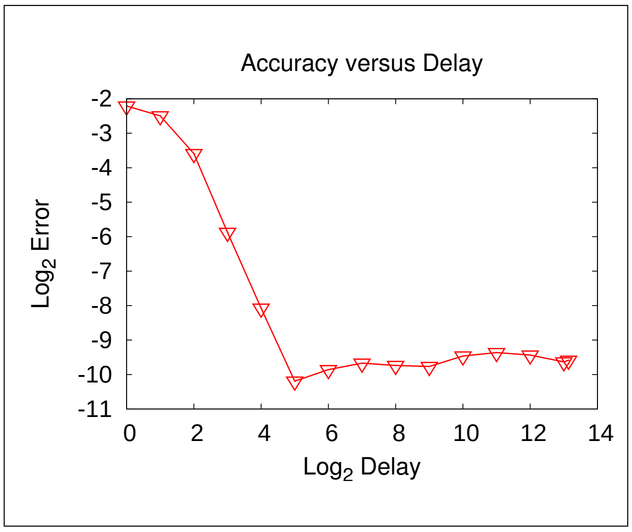

Figure 7: Example of accuracy versus delay curve tested given thread size and iterations. Accuracy increases when the delay goes up to 32. After that accuracy drops gradually when delay keeps increasing.

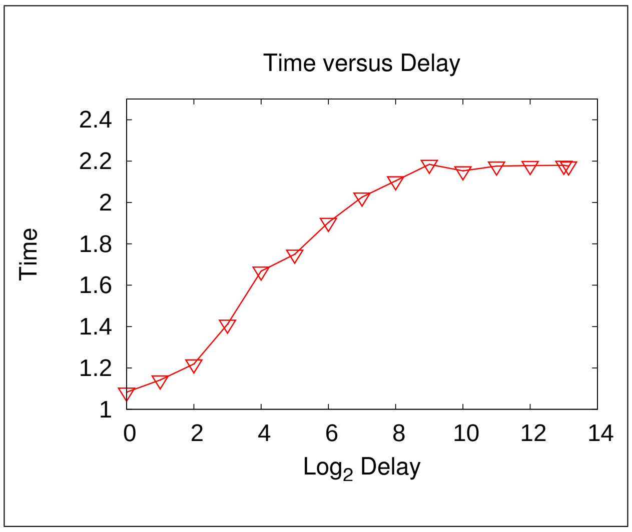

Figure 8: Example of time versus delay curve tested given thread size and iterations. Training time increases if the delay increases.

4.2.2 Iteration

We tested the relationship between changing repetitive learning iterations and performance as shown in Figures 9 and 10. In these experiments: the grid size is 30; the block size is 320; and the delay is 32. Note that memory transfer time is not considered. In Figure 10 we find that the processing time scales very well based on the number of iterations, which means computation and memory costs have a steady proportion. According to the results in Figure 9, it followed our simulation expectation that the error rate drops when the number of iteration increases. Even if we change the delay, it still shows a similar pattern. Surprisingly our classification results seems to be even better than the simulation code. A possibly reason might be that using the delay mechanism could initiate a better starting point for the gradient descent problem or it could avoid a local minimum trap. We would like to further experiment with this to understand what is happening.

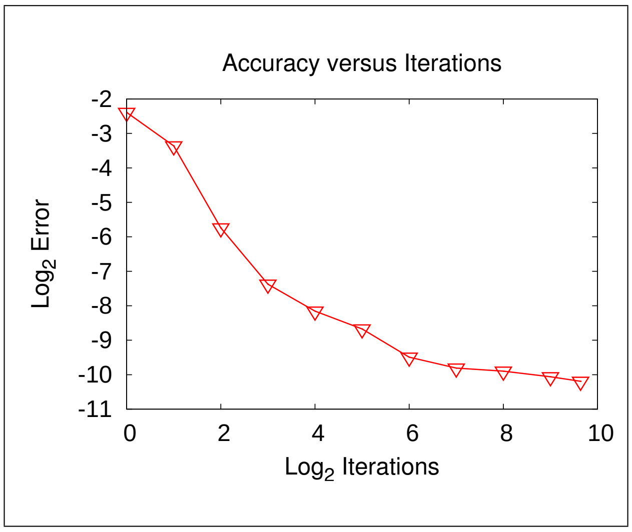

Figure 9: Example of accuracy versus iteration curve tested given threads and delay. The figure shows that error rate drops when iterations increase.

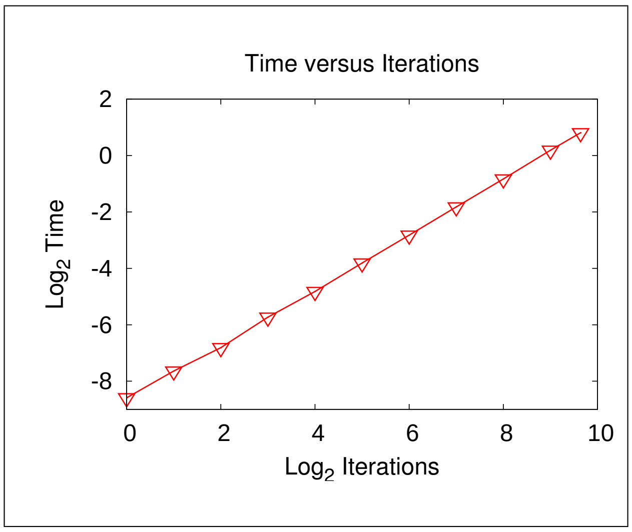

Figure 10: Example of time versus iteration curve tested given threads and delay. Training time is proportional to number of iterations.

4.2.3 Grid size versus Block size

We tested codes with zero delay to find out how changes in grid size and block size will affect processing speed. In ideal case, if we double the number of threads, the training time will be half the original. There are three ways to increase the total number of threads running on a GPU: increase block number per grid, increase thread number per block, and combined. The results through changing grid size and block size under 64 iterations is shown as Figure 11. If we only scale the grid size, we could see a repetitive pattern of 210 thread size (7-fold of SM number). SM can fit several blocks if the resources are available (Hong and Kim 2009). So in this case seven blocks could be executed at one time, therefore we have this repetitive cure. Therefore it was ideal to have the block size equal to SM's fold to achieve better acceleration. Note that we cannot fit as many as seven blocks into one SM as thread number increases. If we only increase the block size (biggest number of threads was 512 for our GPU), processing time drops steadily, although, it is not as steep as the ideal curve. Therefore we conclude that to efficiently use the GPU resources, it is advisable to have more threads running in the blocks. While at the same time we keep all SMs working. So the grid size of 30 seems to be the optimum setting.

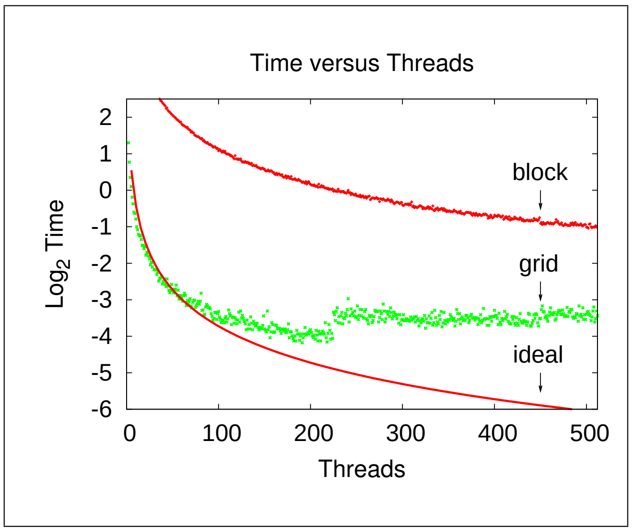

Figure 11: Example of time versus threads curve. The grid curve follows the ideal curve in the begining. The block curve drops steadily.

4.2.4 Acceleration

Figure 12 illustrates the acceleration achieved by comparing processing time between CPU and GPU through both running 800 iterations. There are three curves shown in the figure: "extreme", "standard" and "tuned". In order to increase total thread number, we first increased the grid size until it reached 30 (the number of SMs) and then increased the block size. "Extreme" speed-up was the ratio of CPU processing time to memory transfer time by assuming GPU processing time was zero. "Standard" curve was plotted that thread number equalled to delay plus one and "tuned" curve was tested under a fixed delay of thirty-two. Note that all curves take memory transfer time into account.

4.3 Discussion

One big problem we found in the parallelism of asynchronous optimisation is that a direct and fast communication between threads is not available. Similar findings revealed by Xiao and Feng (2009) claimed

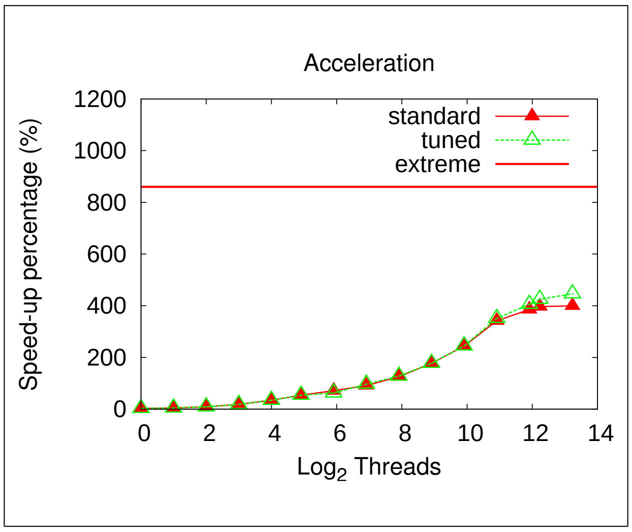

Figure 12: Acceleration curve that we could possibly achieve.

that inter-block GPU communication is a main contributor to total processing time. Furthermore, our control over thread scheduling is limited. There are a few possible approaches we could try to address these synchronisation issues:

1. Stop parallelism after some time, synchronise results and restart the parallelism. This approach would help to ensure correctness stage by stage. However this would introduce more overhead and also the efficiency of parallelism would be undermined.
2. Apply parallelism of pipelined optimisation to decompose loss function as Langford suggested (Langford et al. 2009). Assuming $f_i(\boldsymbol{x}) = g(\langle\phi(\boldsymbol{z_i}), \boldsymbol{x}\rangle)$ The issue is to find out appropriate $\phi(\boldsymbol{z_i})$ and feed data $\boldsymbol{z_i}$ to partial functions. The advantage of this approach is that it dramatically reduces synchronisation costs and updates partial values locally. Nevertheless when combining partial values we still need to make sure all threads are working on the same data.

Theoretical bandwidth of the NVIDIA GeForce GTX 295 is

$$1.24 \times 10^9 \times (512/8) \times 2 = 158.7 \text{ GB/sec}$$

We used *cudaprof* as profiler to test performance by setting the grid size to 30, the block size to 320 with delay 32. Overall global memory throughput was 29.9972 GB/sec. Occupancy for major function was 0.625. Only about 20% the GPU's theoretical capability had been achieved. This results were foreseeable if we implemented a sequential logical program onto parallel architectures. Bridges et al. (2007) experimented sequential codes of C benchmarks in SPEC CINT2000 on multi-core achieving speedup of 454% using 32 threads. Also some applications can hardly gain speedup. The bottleneck of the program was excessive access to global memory during synchronisation. The latency drag down the utilisation of computations. Xiao and Feng (2009) claimed that inter-block synchronisation is the main contributor to total processing time when computation is highly parallelised.

5 Conclusions and Future Work

Nowadays increasing parallel cores have become the stimulus for fast continuing growth in transistor count. However sequential applications have not taken the advantages of increasing computability. Therefore it is worthwhile to explore how to make use of tomorrow's processors. In this paper, we implemented Delayed Stochastic Gradient Descent Algorithm on a GPU platform. With the delay mechanism, we could parallelise an essentially sequential problem that has hardly been handled by general parallel architectures. Based on our experiment, we show that this alternative delay algorithm could achieve comparable accuracy to that of a sequential algorithm through parallel computations.

We also estimated GPU's performance on a strong dependency case, which had not previously been revealed. Our results showed that the GPU's high computability was not fully exerted compared to other GPU implementations. Because the application excessively attempted to access global memory, when parallelism became high, computation time contributed much less than memory access time. However global memory is the only all threads accessible memory, thus it was difficult to avoid such costs. Note that acceleration could still be increased if the dataset size or iterations increased, but synchronisation would still be the bottleneck. GPUs have very light weight threads which are not specially designed for complex operations. Because of its low scheduling design, GPUs gain benefits of fast growing computations but lose complex logic control over parallelism.

In order to solve the sequential problem proposed, there are two possible solutions: to find or to design an architecture that supports fast and complex logic over parallelism, or to revise current algorithms that minimise synchronisation cost however achieving applicable accuracy.

Experimenting on GPUs' simple cores will reveal its incapability of current design when solving some general problems. However further study of combining algorithms with architecture would possibly indicate what are the most important features to be necessarily included in future designs. For example, if synchronisation plays an important role in future parallelism program model, it assures that Fermi architecture (new generation of NVIDIA GPUs) with fast global cache is a smart choice. Furthermore it could be a good idea to have a central coordinate processor in parallel architecture such as PS3, which can sufficiently communicate with other parallel cores. This coordinator can gather information from other threads, but at the same time manage the scheduling and usage of memory. Besides we are also interested to experiment on other existing architectures. As Vuduc et al. (2010) suggested, a hybrid CPU/GPU architecture may perform better overall. This would also eliminate memory transfer time between the host system and the GPU's memory, which is currently one of limitations with current graphics card configurations.

To minimise the synchronisation cost, we can experiment with pipelined approaches of parallelism that require less synchronisation. Furthermore we could improve machine learning algorithms from frequently synchronisation required to only occasional synchronisation required. For example our application requires synchronisation after processing each instance. If an algorithm only requires synchronisation after processing one hundred instances, this would make a big difference.

In general our conclusions are limited in two main ways. Firstly, our application was specifically on one machine learning approach, thus, limiting the generalisation of our claims. Secondly we have only tested the approach on the NVIDIA GTX295 using CUDA programming language. It would be interesting to explore the approach on other GPU models and other programming APIs.

In the next stage in this research we will experiment with other machine learning algorithms on various GPU models or other architectures. We would like to explore optimisation techniques, e.g. pipelined optimisation, orthogonal feature spaces. We could experiment on other stochastic algorithm to evaluate hypothesis of delayed update. Another direction is to explore other algorithms which are more suited to existing parallel architectures.

References

Bridges, M., Vachharajani, N., Zhang, Y., Jablin, T. & August, D. (2007), Revisiting the Sequential Programming Model for Multi-Core, *in* 'International Symposium on Microarchitecture - MICRO 2007'.

Collange, S., Daumas, M. & Defour, D. (2007), Graphic processors to speed-up simulations for the design of high performance solar receptors, *in* 'IEEE 18th International Conference on Application-specific Systems'.

Cormack, G. (2007), TREC 2007 spam track overview, *in* 'proceeding of the Sixteenth Text REtrieval Conference (TREC 2007)'.

Hong, S. and Kim, H. (2009), An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, *in* 'Proceedings of the 36th annual international symposium on Computer architecture', Austin, TX, USA.

Langford, J., Smola, A. & Zinkevich, M. (2009), Slow Learners are Fast, *Journal of Machine Learning Research*, Vo1. 1, 1–23.

Raina, R., Madhavan, A. & Ng, A. (2009), Large-scale Deep Unsupervised Learning using Graphics Processors, *in* 'Proceedings of the 26th Annual International Conference on Machine Learning'.

Steinkraus, D., Buck, J. & Simard, P. (2005), Using GPUs for Machine Learning Algorithms, *in* 'Proceedings of the 2005 Eight International Conference on Document Analysis and Recognition (ICDAR'05)'.

Stone, J., Phillips, J., Freddolino, P., Hardy, D., Trabuco, L. & Schulten, K. (2007), Accelerating molecular modeling applications with graphics processors, *Journal of Computational Chemistry*, Vol. 28, 2618–2640.

Vuduc, R., Chandramowlishwaran, A., Choi, J., Guney, M. & Shringarpure, A. (2010), On the Limits of GPU Acceleration, *in* '2nd USENIX Workshop on Hot Topics in Parallelism'.

Xiao, S., and Feng, W. (2009), Inter-block GPU communication via fast barrier synchronization, *Technical Report TR-09-19*, Dept. of Computer Science, Virginia Tech.