Analysis Acceleration in TMVA for the ATLAS Experiment at CERN using GPU Computing

Hazel McKendrick

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2011

Abstract

ATLAS is one of two general purpose collision detectors within the Large Hadron Collider, detecting millions of events per second. One tool for the eventual analysis of this data is TMVA, the Toolkit for Multi-Variate Analysis. Comprising of a number of machine learning techniques, it supports physicists in classifying events.

This project forms a feasibility study into the use of GPU (Graphics Processing Unit) devices to parallelise TMVA to determine whether such techniques might lead to future performance gains for the framework.

In particular, the Multi-layer perceptron, a class of neural network, is ported to the GPU programming platform CUDA. Performance when training single networks is generally comparable to CPU performance but show promise for future improvement. However, this parallelism of the GPU also allows multiple networks to be trained simultaneously, and this is leveraged to show significant performance gains over undertaking such a task in serial. The challenges and potential for these results to be applied across the TMVA framework is then considered and discussed.

Acknowledgements

I would like to thank my supervisor Andrew Washbrook for his advice, support and assistance throughout this project.

Thanks also to my parents for providing me the opportunity to pursue this degree and supporting me throughout.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Hazel McKendrick*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 ATLAS

Located at CERN (the European Organisation for Nuclear Research) near Geneva in Swizerland, ATLAS (A Toroidal LHC Apparatus) is one of the two general purpose particle detectors within the LHC (Large Hadron Collider), used in the pursuit of answers to fundamental questions in high-energy physics. The experiment is concerned with investigating, improving upon and confirming the standard model of physics, as well as seeking evidence of new physics entirely. A particular aspect of this involves the search for the Higgs boson, the only particle in the standard model yet to be detected.

Protons are accelerated in bunches of 10^{11} which cross at four main collision points on the LHC ring at a rate of 40MHz, where detectors including ATLAS are situated. This leads to 1 billion collision events every second, however only a fraction of these are of interest. As such, ATLAS is composed of three levels through which the detected data is filtered, reducing it from the 40 million events detected per second down to 200 which are then stored and analysed, representing a typical reduction factor of 200,000 [1].

The first level of the trigger is formed of bespoke and massively parallel hardware, while levels two and three are software filters operating on large computing farms. Following this, the data is replicated globally across the LHC Computing Grid for offline analysis and simulation by roughly 100,000 processor cores, stored at national "Tier 1" centers with around 25PB of disk space [1].

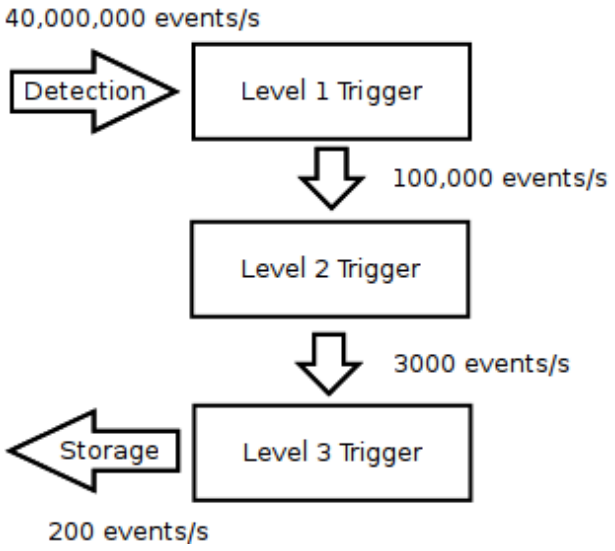

Figure 1.1: Event reduction from detection to storage in ATLAS.

As the quantities of data to be analysed will only increase as LHC operations increase in luminosity, investigating opportunities for potential performance improvements is of great importance. This project forms a feasibility study into parallelising a currently serial analysis tool TMVA (the Toolkit for Multivariate Analysis) using GPGPU (General-Purpose Graphics Processing Unit) computing in order to seek massive performance gains.

1.2 TMVA

ROOT [2] is a collection of frameworks designed to solve data analysis problems in HEP (high-energy physics). It provides various component packages with the aim of providing a common set of tools and techniques relevant to multiple LHC experiments, in order to aid physicists in performing data analysis and to visualise and interpret the results.

TMVA, the Toolkit for Multi-Variate Analysis [3], is a data analysis package which can operate as part of ROOT. It provides a number of distinct supervised learning techniques, used to separate events which can be classified as scientifically significant signal from those considered to be background. The background and operation of TMVA is discussed in more detail in Chapter 3.

While TMVA is designed with HEP and the associated quantities of data in mind, pro-

cessing requirements can be extensive, with large scale computing solutions needed to produce results in a timely manner [4]. For a number of years, the need to seek performance gains through increased parallelism rather than processor improvements correlated with Moore's Law has been apparent [5]. Alternative highly-parallel hardware architectures, in particular GPUs, have lead to performance gains in a number of areas and as such could provide an effective tool in data analysis for ATLAS [4].

1.3 Thesis Overview

This project forms a feasibility study into analysis acceleration in TMVA through the use of GPUs (Graphics Processing Units). Chapter 2 provides an introduction to GPU computing, and the CUDA platform which will be used. Chapter 3 gives further details of the TMVA framework, the techniques within it, and its operation and structure leading to the selection of the Artificial Neural Network as an appropriate area on which to focus for investigation. It will then give background information about the operation of Neural Networks in general, and TMVA's multi-layered perceptron in particular. Chapters 3 also introduces the theoretical and conceptual challenges of parallelising the TMVA framework and neural network implementation within it.

Chapter 4 details the practical implementation of a multi-layered perceptron on the GPU, including analysis of the CPU based methods, design and creation of GPU based functions and issues involving data storage and retrieval. Chapter 5 expands on the methods in chapter 4 by detailing how multiple neural networks can be trained in parallel on GPU.

Chapter 6 gives the results of training single and multiple neural networks using two relevant data sets. These are analysed and discussed, considering the reasons behind any unexpected results and the potential for improvement of the GPU based solution. Chapter 7 concludes the thesis, discussing the applicability of the feasibility study results to TMVA as a whole and suggesting areas for future investigation.

Chapter 2

GPU Computing

2.1 Background

Traditionally, the development of GPUs (Graphics Processing Units) has been driven by the computer games industry, where specialised and pipeline-able hardware is required to render thousands of polygons in a matter of milliseconds. As a result, GPUs are optimised to allow high-throughput data-parallel processing of floating point numbers, as shown in 2.1.

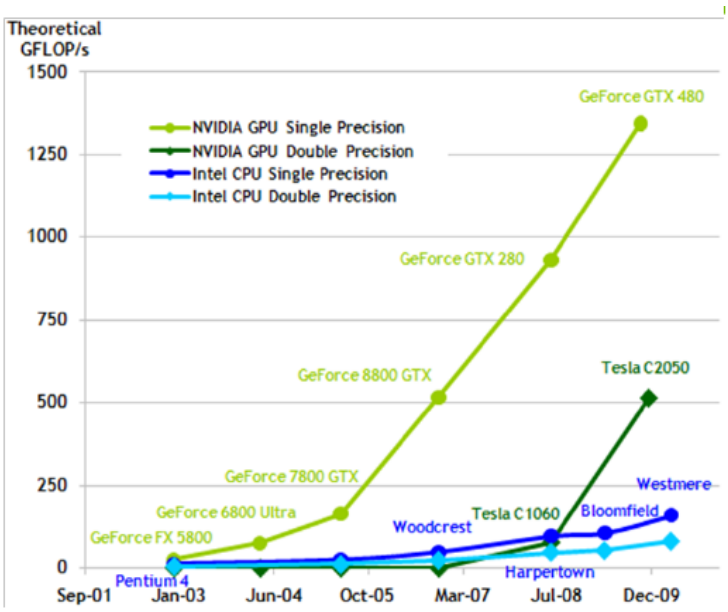

Figure 2.1: A comparison of the performance capabilities, in floating point operation per second, of GPUs and CPUs available over time. From [6]

In recent years this capability has also been applied to general purpose computing problems, first by adapting non-graphics data to be stored in textures and using parallel graphics "shader" program to process it, and later through general purpose technologies such as Nvidia's CUDA and the Khronos Group's OpenCL [6] [7]. These developments reflect the performance needs of numerous fields from finance to fluid simulation, as demonstrated by the breadth of interest shown at Nvidia's GPU Technology Conferences[8]. There is a strong possibility for heterogeneous systems to provide a cost-effective alternative to CPU based clusters, with numerous GPU based clusters now part of the top 500 supercomputers[9]. Figure 2.2 shows the performance improvements which Nvidia suggest might be expected from the world's most powerful supercomputers if graphics hardware was used, and the cost of producing clusters competitive with these.

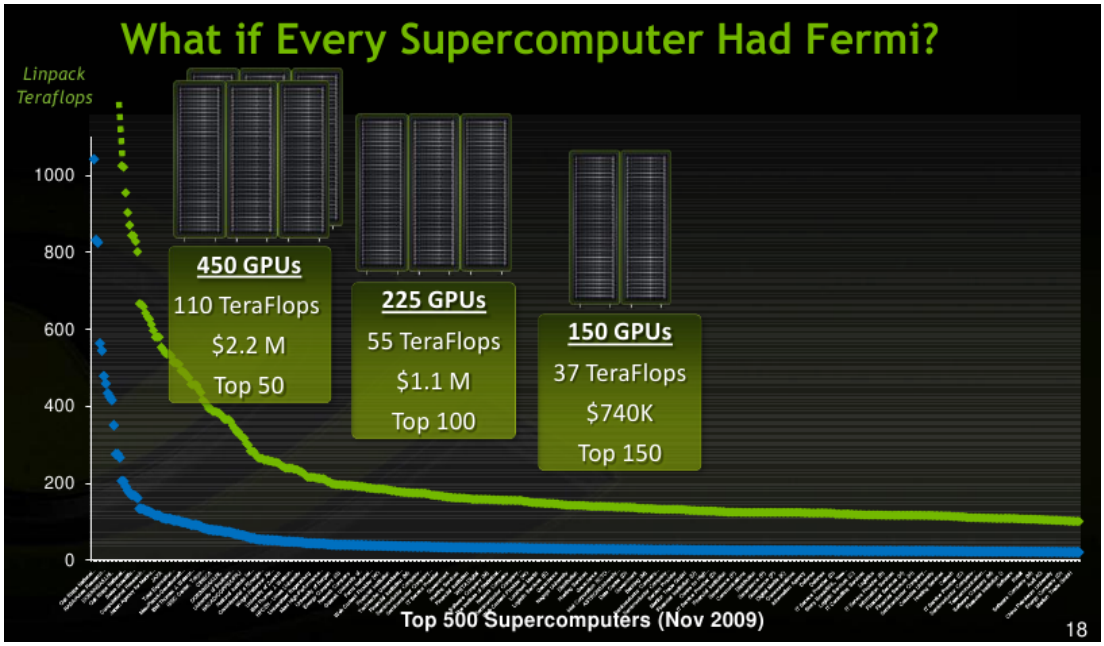

Figure 2.2: NVidia's suggested improvements in teraflops from supplementing existing supercomputers with GPU hardware. From [?]

However, the highly parallel nature of GPU hardware still presents a number of challenges for general purpose use. As the following sections will describe, careful consideration must be given to memory allocation, structure and transfers, the division of a problem for multi-threaded computation as well as branching and operation within each thread. As 2.3 shows, the benefits and many of the challenges stem from dedicating more transistors on a GPU to data processing at the cost of flow-control and caching.

Figure 2.3: A basic comparison of schematics for CPU and GPU architectures. While the CPU dedicates extensive resources to control and caching alongside a lower number of Arithmetic and Logic Unit (ALU) cores, the GPU maximises the number of lower individual performance cores available. From [?]

2.2 Nvidia's CUDA Platform

Nvidia's CUDA (Compute Unified Device Architecture) provides a platform on which programs can be developed in high level languages (such as C using Nvidia's "C for CUDA") with access to a GPU's memory and instruction set.

A number of alternative SDKs for GPU based development exist including StreamSDK, DirectCompute and OpenCL. However, CUDA was selected for this project due to its current popularity, performance and the quantity of documentation and resources available. While OpenCL might be expected to exceed CUDA in popularity in the long term due to its wider device support (where CUDA can only be used on Nvidia hardware) and open standards, its current maturity level in terms of performance and support was not yet considered to match that of CUDA.

While the CUDA manual should be referred to for detailed programming instructions, the following sections provide an overview of the platform, API and associated issues.

2.2.1 Kernels

A kernel represents a single module of computation which can be performed in parallel by a number of threads. With C for CUDA, these can be written in a variant of C with a number of CUDA specific and C++ derived extensions, for example support for templating and in recent versions of the framework C++ style memory management.

A basic kernel which increments event element in an array might take the following form:

Listing 2.1: A simple kernel example

```
1 __global__ void IncArray(float* A)
2 {
3   int id = threadIdx.x;
4   A[id] += 1.f;
5 }
```

Pointers passed to the function must refer to locations in the GPU's address space, as such the data must be copied to the device before the kernel is invoked and any results transferred back to host memory after completion. Memory can be managed in a traditional C-like way with the cudaMalloc, cudaFree, and cudaMemcpy functions.

The `__global__` qualifier specifies that a kernel will be executed on device, but is callable only from the host CPU. Alternatives are `__device__`, which signifies that the kernel resides on and can only be invoked from the device and `__host__`, which specifies functions existing and executed from the host (and will be assumed if no qualifier is used).

2.2.2 Thread Hierarchy

Current computation on a CPU tends to involve a relatively low number of powerful cores, commonly between 8 and 12 in a modern server, and a low degree of parallelism across these. The aim GPU computing is the inverse: a large number of less powerful cores are leveraged to perform many operations in parallel. Parallel operations on the GPU with CUDA are organised through a hierarchy of threads, as shown in figure 2.4. This arrangement assists the programmer by providing a logical way to address data structures such as vectors, matrices and arrays of one or more dimensions based on thread and block coordinates, but also relates to the multi-processor and stream-processor based architecture of a GPU.

- Individual threads process kernels in parallel. The ID of each thread can be used for memory access to allow vectors or matrices of data to be operated on. One or more threads will be processed by a single stream processor (SP)
- Blocks consist of single or multi-dimensional arrays of threads, all processing

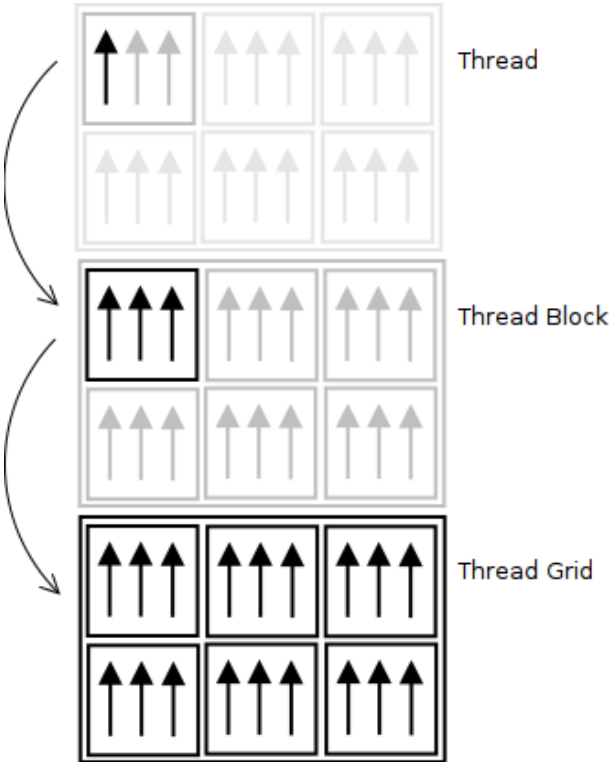

Figure 2.4: The CUDA thread hierarchy. Each thread, block and grid operates on the same kernel.

the same kernel in parallel, which can be coordinated using synchronisation primitives such as barriers ensuring every thread reaches a certain point before computation is allowed to continue. Threads within a block must reside on the same physical multiprocessor (MP) core and as such are limited in number, to 1024 on current devices such as Nvidia's Fermi architecture.

- A grid consists of an array of thread blocks, again all processing the same kernel and arranged in a single or multidimensional grid.

This model allows for scalability of parallelism between GPUs with varying capabilities, as demonstrated in 2.5. As such, no guarantees can be made about the ordering of blocks and they must be able to be processed independently.

Grid and block sizes are specified for a given kernel call using CUDA's triple angle-bracket syntax as follows:

Listing 2.2: Calling a kernel

```
1    dim3 blockNum(8,8);
2    dim3 threadNum(16,16);
```

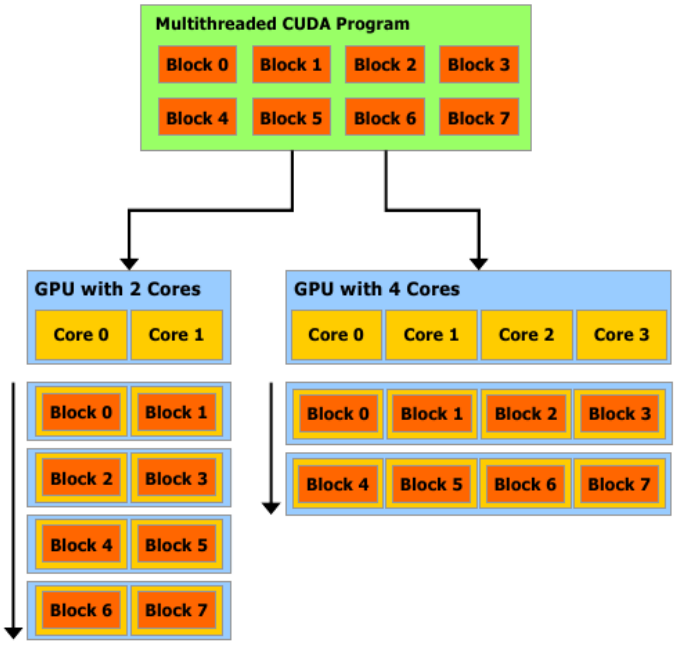

Figure 2.5: Block placement varies between different GPUs, allowing CUDA programs to scale to different hardware platforms. From [?]

```
3     BasicKernel<<<blockNum, threadNum>>>(gpuArray);
```

This creates a block size of 16x16 threads and a grid of 8x8 blocks, calling a basic kernel to operate on an array. Unique IDs for each thread can be calculated from block and thread ids as `blockID * blockSize + threadID`.

2.2.3 Heterogeneous Computing Model

CUDA allows the use of a GPU as a co-processor, with overall program flow control maintained by the CPU. Figure 2.6 shows how kernels can be used to support CPU operations, with data transfers and allocation also initiated from the host.

2.2.4 Memory Model

The CUDA memory model is hierarchical in nature, offering several alternatives in terms of allocation, performance and lifespan for access by threads, thread blocks and the host.

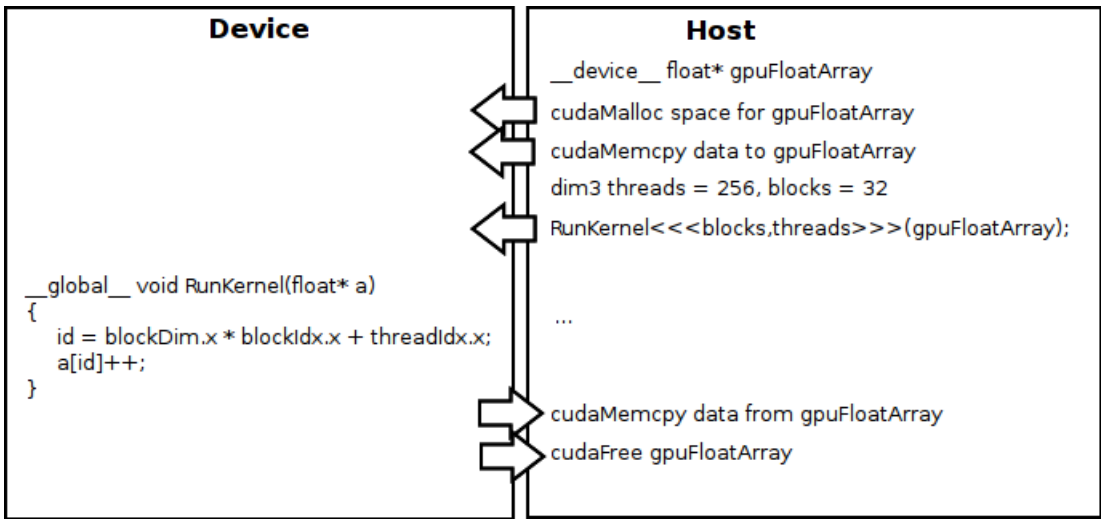

Figure 2.6: CUDA program flow is initiated and controlled by the host.

2.2.4.0.1 Global Device Memory Global memory can be allocated and written to and copied from by the host, then both read and written to by any thread in any block. The contents of global memory are retained between kernel operations, allowing results to be transferred between thread block and kernel calls, however access can be slow. The size can be up to 1GB or more on modern GPU hardware, but data transfer speeds can form a bottleneck in using all of this effectively.

An example use case for global memory includes any data which is transferred from CPU to be modified, or gathering results from across thread blocks to complete a calculation.

2.2.4.0.2 Shared Memory Each thread block has access to its own shared memory block for the duration of a kernel call, which can be allocated using the `__shared__` qualifier. This memory is divided into banks which can quickly be accessed in parallel so long as conflicts do not arise, but cannot be accessed by the host or other blocks, and additionally it is limited in size, to either 16k or 48k on modern Nvidia GPUs.

The speed of access makes this memory suitable for performing calculations within a kernel call, in some cases offering a benefit even when a number of results must be copied from and to global memory.

2.2.4.0.3 Local Memory and Registers Private memory allocated within a kernel is available in the form of local memory and registers. While registers offer almost

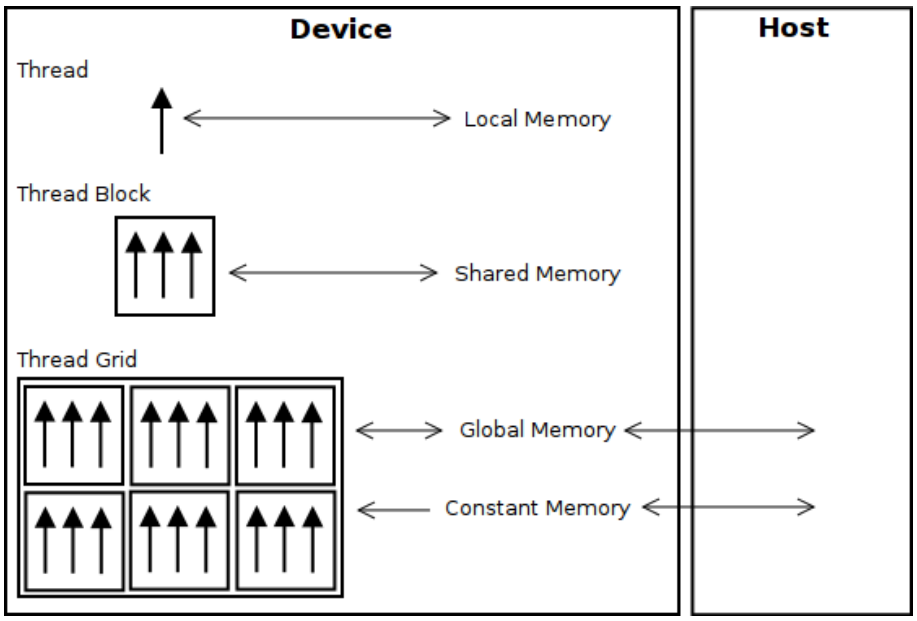

Figure 2.7: The CUDA memory hierarchy corresponds to its thread hierarchy, with host access only to globally shared memory.

immediate access, they are limited in number per multiprocessor and as such per thread block, and local memory access can be substantially slower. This is due to the fact that local memory is not physically local to the stream processor, its name derives from its limited local scope.

The compiler rather than the programmer is responsible for determining the use of registers versus local memory when space is allocated in a kernel. However this memory might be used when storing single temporary values such as each thread's ID.

2.2.4.0.4 Constant Memory

Constant memory, and the related constant cache, provides a limited quantity of global and persistent memory which can read by all threads. This memory is optimised for broadcast to every thread in a block rather than lookup based on individual thread IDs.

2.2.5 Warp Scheduling

A final intricacy of CUDA which is important to discuss comes in the form of warps and warp scheduling. A warp is a group of 32 threads which all must execute the same instruction simultaneously, controlled by a warp scheduler which will allow any available warps to continue executing. In the case of branches and conditional statements

within a warp, the options will be processed sequentially until paths converge again. As such, it is normally recommended to avoid branching within kernel calls.

2.3 Hardware Architectures

For the purpose of this project, two Nvidia devices are available: the C1060 “Tesla” device and the C2050 “Fermi” device. The basic specifications of these are shown in table 2.1.

	Tesla	Fermi
Number of processor cores	240	448
Processor core clock	1.296 GHz	1.15 GHz
Memory	800 MHz 4GB	1.5 GHz 3GB

Table 2.1: Basic comparison of Tesla and Fermi hardware architecture. From [6]

Figure 2.8 additionally shows the architecture of the Fermi device, demonstrating the configuration and vast number of stream processors (shown in green). Also notable is that the Fermi architecture provides a small amount of L1 and L2 cache (pale blue), where previously this was not available.

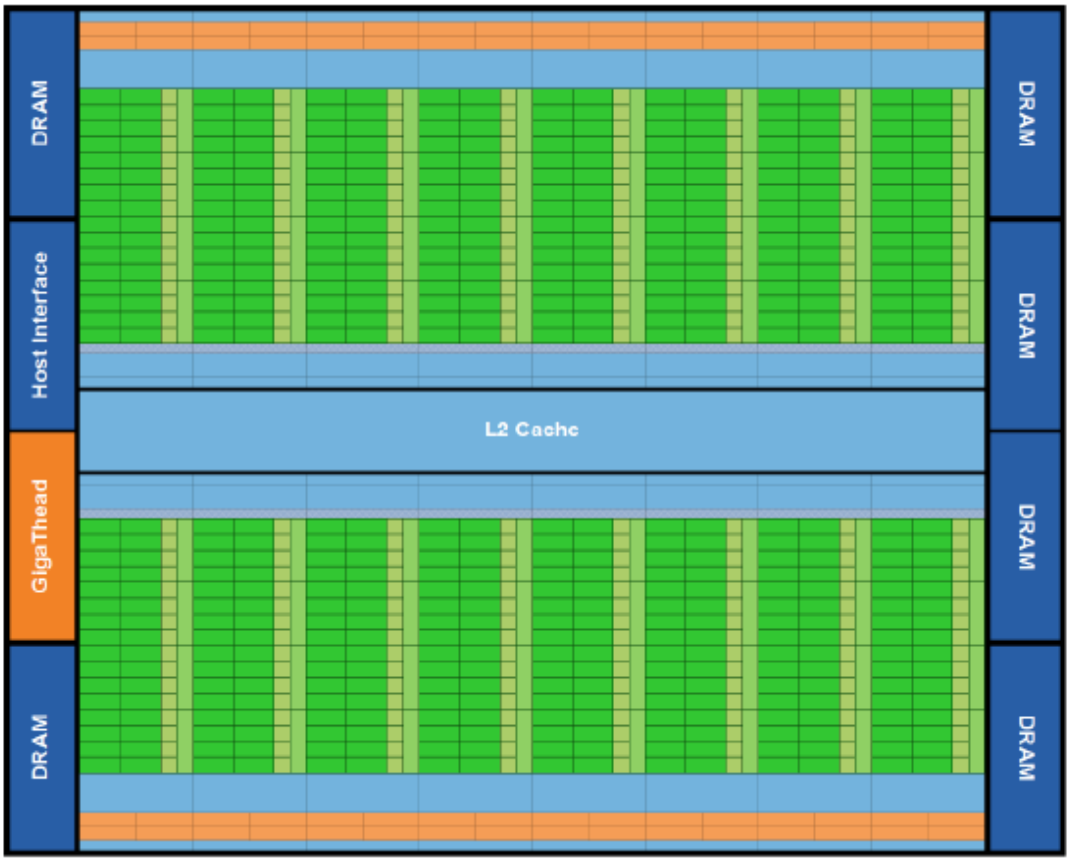

Figure 2.8: Architecture of Nvidia's Fermi GPU. From [6]

Chapter 3

TMVA

3.1 Framework Overview

3.1.1 TMVA Techniques

The Toolkit for Multivariate Analysis implements ten distinct analysis techniques:

- Rectangular cut optimisation
- Projective likelihood estimation
- Multidimensional probability density estimation
- Multidimensional k-nearest neighbour method
- Linear discriminant analysis
- Function discriminant analysis
- Artificial neural networks
- Boosted decision trees
- Predictive learning
- Support vector machines

Each of these involves varying benefits and drawbacks for data analysis, as detailed in table 3.1, and many offer a choice of implementations, specialisations or variants.

CRITERIA		MVA METHOD									
		Cuts	Likeli-hood	PDE-RS / k-NN	PDE-Foam	H-Matrix	Fisher / LD	MLP	BDT	Rule-Fit	SVM
Perfor-mance	No or linear correlations	⋆	⋆⋆	⋆	⋆	⋆	⋆⋆	⋆⋆	⋆	⋆⋆	⋆
	Nonlinear correlations	○	○	⋆⋆	⋆⋆	○	○	⋆⋆	⋆⋆	⋆⋆	⋆⋆
Speed	Training	○	⋆⋆	⋆⋆	⋆⋆	⋆⋆	⋆⋆	⋆	○	⋆	○
	Response	⋆⋆	⋆⋆	○	⋆	⋆⋆	⋆⋆	⋆⋆	⋆	⋆⋆	⋆
Robust-ness	Overtraining	⋆⋆	⋆	⋆	⋆	⋆⋆	⋆⋆	⋆	○	⋆	⋆⋆
	Weak variables	⋆⋆	⋆	○	○	⋆⋆	⋆⋆	⋆	⋆⋆	⋆	⋆
Curse of dimensionality		○	⋆⋆	○	○	⋆⋆	⋆⋆	⋆	⋆	⋆	
Transparency		⋆⋆	⋆⋆	⋆	⋆	⋆⋆	⋆⋆	○	○	○	○

Table 3.1: Comparison of techniques available in TMVA. ⋆ represents good performance and ○ poor performance. From [3]

Based on the criteria in table 3.1, one or more techniques should be selected and customised through a series of optional parameters for use.

3.1.2 Phases of Operation

While the operation of each method in TMVA varies widely, as machine learning techniques they all follow a similar two-phase pattern for analysis:

- Training. Datasets where the composition of signal and background is already known are used to produce a mapping function for analysis with each technique.
- Application. The trained technique(s) are applied to a yet-to-be classified data set.

Timing results for training and applying each technique using a sample data set are shown in figure 3.1. These timings represent a small sample of events using the default training settings for TMVA and as such may not fully demonstrate the comparative strengths and weaknesses of each technique. However, it is clear to see that there is a great variety in timings between each method, and that on average training times are substantially longer than the testing times which follow. As such, the focus of the investigation will be on the potential for reduction of training rather than application times.

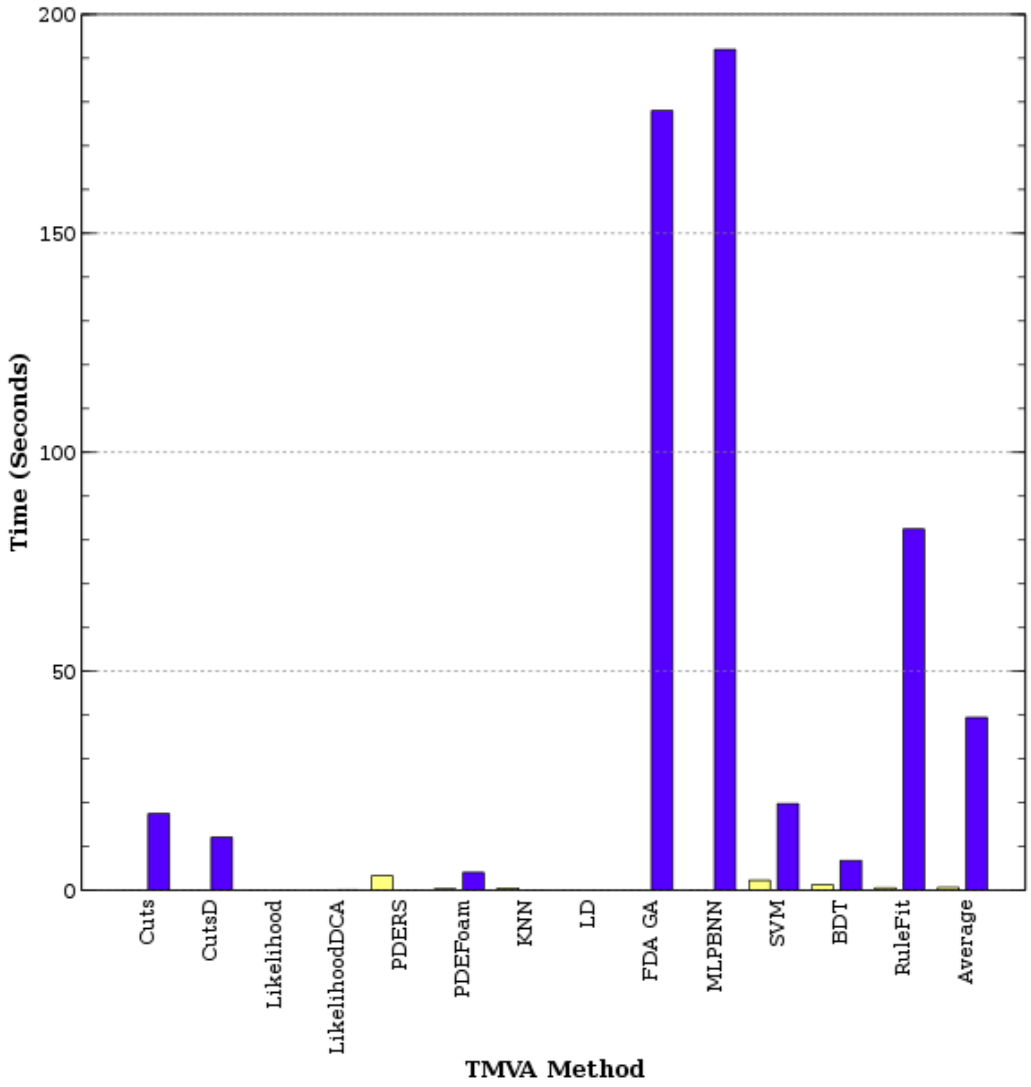

Figure 3.1: Graph of example training and testing times on a small sample dataset for techniques in TMVA. Left bars represent testing times (often too small to be visible), while right bars represent training time.

3.1.3 Software Structure

TMVA is a widely object-oriented piece of software, with data and techniques encapsulated in individual objects wherever possible. This allows for flexibility for the codebase to expand and offer alternative methods, configurability by composing methods of objects based on parameters based at run-time, and minimal redundancy as code can be shared between methods.

Using an abstract factory pattern allows TMVA to remain agnostic about which of its methods are being used for analysis as methods are created and handled in terms of their abstract base class. This ensures that each method can be applied to the training and test data sets in the same way, ensuring results are not biased by the framework overall as it can make use of them without knowledge of their internal functionality.

However, while a widely object-oriented and modular codebase can aid maintainability and understanding, it can also lead to reduction in performance by encouraging multiple levels of indirection in order to call methods and access data, and by using available cache inefficiently. The pipelined nature of a GPU discourages this style of programming, as such alternative code structures may need to be considered in order to successfully port the techniques.

3.2 General Challenges

In evaluating the potential for TMVA to be parallelised at a conceptual level, a number of challenges must be considered.

3.2.0.0.5 Abstract Object Oriented Structure As discussed above, the structure expected for pipelined and parallel GPU code differs greatly from the object-oriented nature of TMVA. This presents a challenge in that simply locating all code and data-dependencies relating to a specific technique is a non-trivial task. Additionally, code which is shared between methods in TMVA might need to be implemented separately when CUDA is used.

3.2.0.0.6 Integration of CUDA TMVA has a complex build process, and relies on a number of pre-processor macros for compile-time code generation. A method must

be found for adding CUDA capabilities to an established codebase which minimises alterations to unrelated CPU code.

3.2.0.0.7 Varied Techniques Techniques in TMVA do at times share code and support classes, however in general there are large differences between them. It should be evaluated whether parallelisation methods applied to one technique are relevant to others in the framework, or whether each must be considered independently.

3.2.0.0.8 Flexibility and Options TMVA supports a large variety of options and alternative implementations which are selected at run time. While branching anywhere in CPU code in order to support these is generally considered acceptable, this is not necessarily possible in GPU code. Transferring extraneous data and branching to support various user-given parameters might be detrimental to a CUDA implementation of TMVA techniques, however failing to support such options would limit the utility of the parallelised code.

3.3 Related Work

There has been various work relating to techniques and functionality contained in TMVA and its implementation on GPU, which might influence the choice of areas to attempt to parallelise. Monte Carlo simulation, for example, relies on randomly sampling values and selecting the best solution from these. The lack of dependence between samples makes part of such a technique ideal for parallelisation, as suggested by Farbin when discussing how GPUs might be applied to HEP problems [4]. Després et al used Monte Carlo simulation within a GPU based ray tracer to investigate the applicability of this combination to radiation therapy [10]. Overall, they presented a performance improvement over a CPU based version of the same experiment, however they noted that the lower precision of GPU calculations can be problematic.

Simulated annealing and genetic algorithms are both iterative methods for approximating solutions to optimisation problems, with simulated annealing being probabilistic and inspired by metallurgy while genetic algorithms derive from natural evolution. Choong et al detail the implementation of simulated annealing on GPUs, and found significant performance gains over CPU versions [11], although constrained memory

access and the SIMD architecture did prove to be limiting. Yu et al reached similar conclusions in their work with genetic algorithms, but additionally found memory bandwidth to be a serious bottleneck in cases where it was not possible to implement all parts of the algorithm on the GPU [12].

An alternative technique, Support Vector Machines, partition data into classes by constructing hyperplanes which divide them. Catanzaro et al found the performance gains from implementing SVMs on GPU to be of a magnitude great enough to fundamentally alter the applicability of such techniques [13]. While processing 600000 data points and 50 dimensions took 18 hours on CPU, this was reduced to a mere 34 minutes with a consumer GPU. Using the map-reduce pattern, careful consideration was given to data transfer and cache access; naïve ports of CPU algorithms are unlikely to achieve similar results.

More complicated techniques from TMVA have also successfully been implemented on GPU. Aritifical Neural Networks comprise a system of connected artificial neurons which is adapted to represent a relationship between inputs and outputs. Guzvha et al explained how GPUs can be a cost-effective and high performing alternative to CPUs and specialised neuro-processors [14] while Luo et al found similar gains from training Multi-Layer Perceptrons on GPUs and used this as the basis for a number of implementation recommendations [15]. Their advice include working to decrease data transfer between CPU and GPU, staying aware of the differences in GPU hardware, and seeking bottlenecks which in this case were caused by using multiple passes to find minimum data values. Based on the success of this previous research and the extensive training time for the technique, neural networks will form the focal point of this feasibility study.

3.4 The Neural Network

3.4.1 Background

Artificial Neural Networks (ANNs) are a biologically inspired machine learning technique, using layers of interconnected arificial neurons to model relationships between input and output data. Through training of the network, weights and values within it can be adjusted to allow input data to be passed between connected neurons and

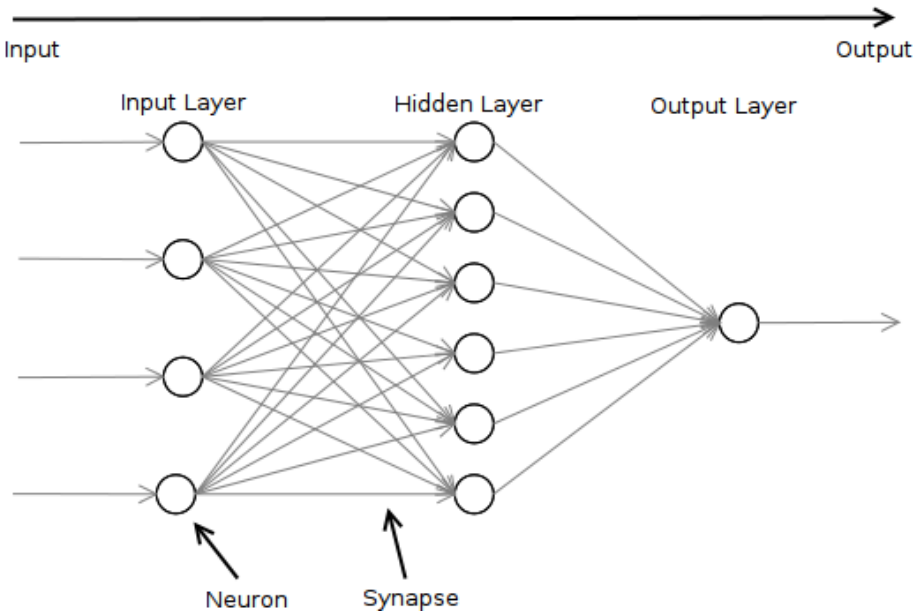

Figure 3.2: Example structure of a basic Multi-layer perceptron.

eventually classified.

Feed-forward neural networks (FFNNs) are artificial networks are a simple and popular form of neural network, in which data passes only in one direction between input and output, with no loops or cycles. Multi-layer perceptrons (MLPs) are a sub-class of FFNN formed of multiple layers of neurons, with data passed between them.

3.4.2 Network Structure

Figure 3.2 shows the structure of a basic MLP. Data is fed in to a number of input neurons. This layer of neurons is connected by a number of weighted synapses to the next "hidden" layer, which can then be connected in the same way to additional hidden layers. Eventually, they data exits the network through the output layer.

3.4.3 The Neuron

Individual neurons take a number of weighted inputs through their synapses, and from them form a single output value. This is achieved through an activation threshold function applied to the combined input to the neuron, and a neuron response function which combines these inputs. Figure 3.3 demonstrated how the sum of weighted inputs

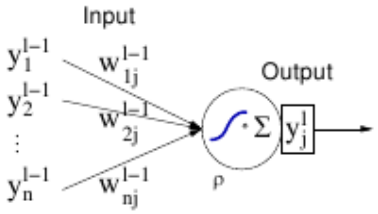

Figure 3.3: Input combination in an individual neuron. From [3].

$$\kappa: \ (y_1^{(\ell)}, .., y_n^{(\ell)} | w_{0j}^{(\ell)}, .., w_{nj}^{(\ell)}) \rightarrow \begin{cases} w_{0j}^{(\ell)} + \sum_{i=1}^{n} y_i^{(\ell)} w_{ij}^{(\ell)} & \textit{Sum}, \\ w_{0j}^{(\ell)} + \sum_{i=1}^{n} \left(y_i^{(\ell)} w_{ij}^{(\ell)} \right)^2 & \textit{Sum of squares}, \\ w_{0j}^{(\ell)} + \sum_{i=1}^{n} |y_i^{(\ell)} w_{ij}^{(\ell)}| & \textit{Sum of absolutes}, \end{cases}$$

$$\alpha: \ x \rightarrow \begin{cases} x & \textit{Linear}, \\ \frac{1}{1+e^{-kx}} & \textit{Sigmoid}, \\ \frac{e^x - e^{-x}}{e^x + e^{-x}} & \textit{Tanh}, \\ e^{-x^2/2} & \textit{Radial}. \end{cases}$$

Figure 3.4: Input functions for neurons in TMVA's MLP. y and w values refer to the initial input or input synapse weights to the neuron respectively. From [3].

can be used, along with a sigmoid function, to produce an output value.

TMVA's MLP class contains a choice of common activation and response functions, as detailed in figure 3.4. Differing activation functions allow data to be classified according to differing divisions. It is the choice of these activation functions which allows multi-layer perceptrons to classify data which is not linearly separable.

3.4.4 Error Gradients and Backpropagation

Once an output value from the final layer has been obtained, it must be used to train and improve the network. This is achieved by a gradient-descent method. First, the difference between the desired or expected output value for a given event and the actual value obtained is calculated. Using this, we must move along the output (activation) function by multiplying by its derivative, towards the expected value by this calculated

delta value and bounded by a learning rate which is pre-set between 0 and 1. Using this calculated error, we proceed to step backwards through each layer of the network, propagating it to adjust and improve synapse weights. This takes a similar form to forward propagation, with the weights of preceding synapses affecting neurons in the previous layer.

Chapter 4

Development of A CUDA Based MLP

4.1 Approaches

As a feasibility study in applying GPU based computing techniques to TMVA, the practical work involved not just creating an independent neural network, but porting existing TMVA functionality to CUDA.

Based on this, a number of approaches were considered:

1. Following the object oriented structure of TMVA as closely as possible, to produce code matching the original in structure as closely as possible. This method might not fully exploit the available GPU parallelism, but if successful would allow ease of maintainability and understanding for future programmers.
2. Using a data-oriented programming pattern, but maintaining the use of a number of seperate kernel functions matching those in the CPU MLP. This would allow data parallelism in a way supported by GPU computing, while maintaining some similarity between the GPU and CPU codebases.
3. Creating an entirely seperate larger kernel function which might be structured somewhat differently to that in TMVA. This would allow the GPU solution to fully exploit available parallelism and memory on the GPU, but could cause challenges in creating a GPU based MLP which matches the existing CPU solution and features.
4. Adapting an existing GPU based MLP solution to use data through TMVA. This

would result in a GPU solution entirely independent of TMVA's existing MLPs, but might allow the quick evaluation of whether and how GPU computing could be applied to TMVA.

4.2 Preparing TMVA

A number of preparatory steps had to be completed before any GPU functionality could be added to TMVA.

4.2.1 Analysis

Initially, the CPU based MLP class provided by TMVA was profiled in terms of processing times and memory use, using igProf[**?**]. This reinforced the previous assertion that training times were far more significant than testing times, and that the MLP back-propagation algorithm comprised the majority of the work completed. These results provide direction when seeking parallelism in TMVA.

Table 4.1 shows the cumulative results of profiling a training and testing run on the MLP class in TMVA, covering 70% of the total running time. Table 4.2 however gives the timing results for individual functions involved in the processing of MLPs in TMVA. Note that these correspond mainly to functions outwith the MethodMLP class; together the tables demonstrate that the main MethodMLP class takes an organisational role in allowing the hierarchical traversal of training functions, however the functional actually taking most time to process individually are external to it. Additionally, functions involving the traversal of array classes comprise a significant proportion (8.1%) of the total time. This is something which might be mitigated by carefully planning the data positioning on GPU.

4.2.2 TMVA Expansion

In order to add any additional functionality to TMVA, it was necessary to create a new method, labelled "CUDA_MLP" which can be selected for use along with other techniques when analysis is performed. This was achieved by modifying a version of the MethodMLP class, deriving from a Neural Network Method base class to allow for

% of total time	Function
100.0	main
97.5	TMVA::Factory::TrainAllMethods()
96.4	TMVA::MethodBase::TrainMethod()
96.1	TMVA::MethodMLP::Train()
96.1	TMVA::MethodMLP::Train(int)
96.0	TMVA::MethodMLP::BackPropagationMinimize(int)
83.7	TMVA::MethodMLP::TrainOneEpoch()
83.1	TMVA::MethodMLP::TrainOneEvent(int)
57.4	TMVA::MethodMLP::UpdateNetwork(double, double)
30.8	TMVA::MethodANNBase::ForceNetworkCalculations()

Table 4.1: MLP functions in TMVA, ordered by the cumulative percentage of processing time they require.

% of total time	Function
8.10	TObjArrayIter::Next()
6.07	TMVA::TSynapse::CalculateDelta()
5.07	TObjArray::At(int) const
4.53	tanh
3.80	TMVA::TSynapse::AdjustWeight()
3.36	TMVA::TSynapse::GetWeightedValue()
3.28	TFormula::EvalParFast(double const*, double const*)
2.92	TMVA::TNeuronInputSum::GetInput(TMVA::TNeuron const*) const
2.34	malloc
2.33	TMVA::TNeuron::CalculateDelta()

Table 4.2: MLP functions in TMVA, ordered by the percentage of total time spent in that individual method.

CUDA related functionality to be provided externally to the class, and for this to be setup and initialised. In this way, CUDA methods can be selected and used alongside CPU based methods at run time and data in the TMVA class hierarchy can be accessed.

4.2.3 CUDA Integration

An initial concern with the feasibility of supporting an extensive codebase with a non-trivial build process like TMVA with GPUs was the practical issue of allowing CUDA access from existing classes. However, with alterations to the Makefile and build process it became possible to build CUDA *.cu* files alongside the TMVA's original C++ source and header files.

The pattern for using C for CUDA and C++ files together in a method then is to create a C++ class, conformant with any of requirements for working correctly with TMVA's pre-processor macros, and to create kernel functions in a corresponding .cu file. C wrapper functions can then also exist in the .cu file, and only these should call kernel functions directly. The C++ code can then include and reference a header file defining these, thus creating a uni-directional access pattern for the CUDA functions.

4.3 Initial Attempt

The initial attempt to parallelise the neural network functionality involved attempting to follow the first approach described about; closely following the structure of the object oriented codebase and transferring required data to GPU for each function. This was intended to be an intentionally simplistic approach to the problem which might aid ease of understanding a future work at the cost of performance. However, such an approach proved unworkable as private data contained in supporting classes was not always readily available, and functions such as random number generation provided blocks which could not readily be overcome.

It was clear from this that a more drastic restructuring and rewriting of existing code would be necessary to produce a functioning network.

4.4 Data Oriented Approach

4.4.1 Data Oriented Design Background

Data oriented design focusses specifically on the location and efficient access of data relating to a specific problem. While an object oriented approach might store all the variables relating to an object in a single class, a data oriented approach would consider the collection of objects as a whole. Rather than storing arrays-of-structs (or classes) defining each individual object, structs-of-arrays would be created. In essence, the same data would be stored, however objects would exist only implicitly as a collection of array entries while the same data item would exist in continguous memory. This can greatly improve the speed of iteration over an array and cache use on a CPU; on the GPU it allows us to reduce class data to the minimum required, transfer it as a block and access it using thread IDs.

4.4.2 Neural Network Data

While the original CPU based TMVA code encapsulated data in a number of classes, the ported GPU code relies on separate array for each data item. The size of host memory and the need for flexibility means that a large amount of supporting data is stored by the original code, however given the limited memory and transfer overheads of using the GPU it was necessary to reduce this to the minimal quantities. This flexible structure allows for networks of varying sizes and structures to be created at runtime.

This was determined to be the following data items:

- float array neurons, containing neurons for each layer
- float array neuron deltas values
- float array synapses weights
- int array neuronCounts detailing the number of neurons in each layers
- float learning rate

These arrays are allocated before training each network, and the first two initialised directly on GPU while the latter two and the floating point data value are initialised and copied from the CPU.

It should be noted that this minimal use of data on GPU restricts the opportunity to apply user-set parameters which is an important aspect of the CPU version of TMVA, however the MLP class alone contains numerous parameter values and incorporating all of these would not be feasible.

4.4.3 Algorithm

The first phase in designing the CUDA based MLP was reducing TMVA's MLP functionality to the basic neural network algorithm to be implemented. Because TMVA is object-oriented in nature, particular consideration was given to which classes contained each piece of data or functionality. The algorithm can be summarised as follows, however on CPU this leads to the use of a number of supporting classes and functions.

For each event passed through the network in order to train it:

- Initialise the input layer
- Feed the values forward through neuron layers
- Calculate the error
- Backpropagate the error

As explained above, wrapper functions are written in C to allow host control of CUDA kernels, and to provide access from C++ code. Listing 4.1 provides psuedo-code for the MLP wrapper function created. This closely corresponds to the above algorithm, using CUDA kernels to perform each individual step, but also ensures that memory is correctly allocated and filled in order for training to take place. The following sections provide details of the component steps of the algorithm.

Listing 4.1: Psuedocode for the MLP training wrapper function

```
1 CUDA_TrainMLP(...){
2   Allocate memory for the network
3   Allocate and transfer events values
4   Create thread grid
5
6   For each epoch
7     For each event
8       Initialise the input layer
9       For each layer from front to back
```

```
10   Calculate Neuron activation values
11   Propogate values forward
12       Calculate output errors
13       For each layer from back to front
14   Back propograte error value
15       Update synapse weights
16     Reduce the learning rate
17
18   Retrieve the network
19 }
```

As the above pseudocode suggests, parallelisation of the TMVA code focusses on being able to process multiple neurons simultaneously, as events must be fed sequentially through the network over a number of epochs in order for the network to train correctly. This means the success of parallelisation will depend on the number of input neurons, which corresponds directly with the number of input variables in the dataset.

4.4.4 Initialisation

Before the parallel algorithm can begin, weights are initialised to random values. This means that at first, events being fed forward through the network will not be classified in any meaningful way but will produce error values used to refine the network.

This work is performed only once and takes place on the CPU because there is no native psuedo-random number generator available on GPU, however this means that synapse values must be copied over to GPU, whereas neuron values for example can be initialised in place.

Listing 4.2: Initial weight value calculation

```
1     cpuMLP->synapses[i] = 4.0f * (float)(rand())/(float)(RAND_MAX
          ) - 2.0f;
```

4.4.5 Forward Propagation

A single event is fed through the network at one time, using each of its variables as an input to a separate neuron on the input layer, as demonstrated by the following simple kernel:

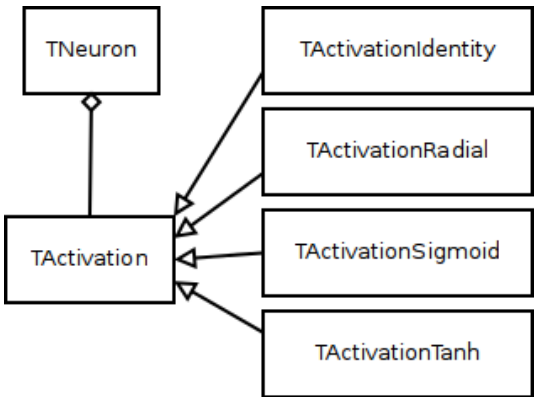

Figure 4.1: Class structure for neuron activation functions in TMVA.

Listing 4.3: Event input to the network.

```
1 // Force the values of the first layer of the network
2 __global__ void CUDAk_InitInputLayer(int eventID, float *neurons,
      float *events, int max)
3 {
4   int id = CUDAk_GetId();
5   if(id < max)
6   {
7     neurons[id] = events[eventID + id];
8   }
9 }
```

Following this, the event values are propagated layer-by-layer through the network based on the activation function and concatenation of weighted synapse values to each neuron.

4.4.5.1 Activation Functions

TMVA's MLP offers a number of functions for neuron activation, contained within a separate TActivation class as detailed in figure 4.1. Each of these is represented by a single kernel in the CUDA codebase.

By default, TMVA uses the Tanh function, listed below. This determines a threshold activation value for each neuron in parallel.

Listing 4.4: Tanh function kernel code

```
1 __global__ void CUDAk_ActivateTanh(float *neurons, int max, int
      numNeurons)
2 {
3   int id = CUDAk_GetId();
4   if(id < max)
5   {
6     neurons[id] = (__expf(neurons[id]) - __expf(neurons[id] * -1.
          f)) / (__expf(neurons[id]) + __expf(neurons[id] * -1.f));
7   }
8 }
```

4.4.5.2 Neuron Input

We consider the neural network neuron and weight data in terms of a matrix multiplication in order to propagate the results of our activation-function transformed input through the network. The following kernel demonstrates how the input values to our vector of neurons in a later layer are formed from the multiplication of the vector of output values from the previous layer, and a matrix of the weighted values of the synapses connecting them. Alternative response functions to the sum of inputs, such as the absolute or square sums, are also provided in the GPU codebase.

Listing 4.5: Forward propagation based on the sum of inputs to a neuron.

```
1   // Use the sum of previous connected neurons to determine the
        value of the next layer
2 __global__ void CUDAk_NeuronInputSum(float *syanpses, float *
      neurons, float *nextNeurons, int x, int y, int max)
3 {
4   int id = CUDAk_GetId();
5   if(id < max)
6   {
7     nextNeurons[id] = 0;
8     for(int j = 0; j < y; j++)
9     {
10      nextNeurons[id] += neurons[j] * syanpses[j + (id * x)];
11    }
12  }
13 }
```

4.4.6 Error Calculation

The error in the output value must be calculated once the event data has propagate to the final layer in the network. While this only occurs on one neuron, the processing work is still completed on the GPU in this case to prevent unnecessary data transfer.

Notice how this follows the theory detailed at the end of 3, using a gradient descent method along the output neuron's activation function, in this case tanh, to determine a required delta value.

Listing 4.6: Calculation of the error in the output value.

```
1 __global__ void CUDAk_CalculateError(int eventID, int *
      eventSignal, float* eventWeights, int minval, int maxval,
      float* neurons, float* neuronDeltas, int max, int numNeurons)
2 {
3   int id = CUDAk_GetId();
4   if(id < max)
5   {
6     float desired = eventSignal[eventID] > 0 ? maxval : minval;
7     float error = neurons[numNeurons-1] - desired * eventWeights[
          eventID];
8     neuronDeltas[numNeurons-1] = error * CUDAk_DerivativeTanh(
          neurons[numNeurons-1]);
9   }
10 }
```

4.4.7 Back Propagation

Given the delta value from the output neuron, backpropagation follows the inverse scheme from forward propagation, with delta values calculated at each neuron based on the synapses coming after rather than before them.

Listing 4.7: Back propagation of values between neuron layers.

```
1 __global__ void CUDAk_Backpropogate(float* prevNeuronDeltas,
      float* prevNeuronValues, float* neuronDeltas, float* synapses,
       int x, int y, int max, int numNeurons)
2 {
3   int id = CUDAk_GetId();
4   if(id < max)
```

```
5   {
6     float error = 0;
7     for(int j = 0; j < y; j++)
8     {
9       error += (neuronDeltas[j] * synapses[j + id * x]);
10    }
11    prevNeuronDeltas[id] = error * CUDAk_DerivativeTanh(
          prevNeuronValues[id]);
12  }
13 }
```

Following this, synapse values are all updated to reflect the improved network based on the learning rate of the network and the delta values of neurons following them.

Listing 4.8: Updating synapse weights based on training.

```
1 __global__ void CUDAk_UpdateSynapses(float learnRate, float*
    neuronDeltas, float* neurons, float* synapses, int x, int y,
    int max, int numNeurons)
2 {
3   int id = CUDAk_GetId();
4   if(id < max)
5   {
6     for(int j = 0; j < y; j++)
7     {
8       synapses[j+id *x] += -learnRate * neuronDeltas[id + x ] *
            neurons[j + x];
9     }
10  }
11 }
```

4.4.8 Network Retrieval

The training scheme is run for a fixed number of "epochs" in order for it to converge on an appropriate configuration. Once this is complete, the network must be transferred back into the TMVA structure from the linear arrays in which it is currently contained. This is problematic due to the slightly differing operation of the CPU and GPU implementations of the network, as the CPU implementation contains additional "bias" neurons to allow the activation functions to be shifted by fixed values, thus aiding convergence of the network.

At present, the network neurons and synapses are copied back into the CPU based data structure by getting hold of each component neuron and synapse in turn, and replacing its value with the one from the GPU. However, when used to classify event data, this leads to results which appear to be incorrect, or at least significantly different from the CPU implementation. As such, while it appears that the difference in network configurations and the difficulty of retrieving data in this way are the cause of errors in the network produced, additional faults in the GPU implementation cannot be ruled out.

4.4.9 Event Data

Event data is originally stored in tree structures in TMVA, and must be packed into linear arrays in order to be transferred and used on the GPU. This is achieved within the CUDA MLP method class, where access to the event data is possible, and again only a fraction of the available data is required.

Listing 4.9: Packing event data into a linear array.

```
1 void TMVA::MethodMLPCUDA::CUDA_FillEventBatch(CUDA_EventBatch*
      eventbatch)
2 {
3   // Find out number of event values
4   const TMVA::Event *eventForData = GetEvent((Long64_t)0);
5   int nValues = eventForData->GetNVariables();
6   int nEvents = Data()->GetNEvents();
7
8   // Fill array with event data
9   for (int i = 0; i < nEvents; i++)
10  {
11    int index = i;
12
13    const TMVA::Event *e = GetEvent(index);
14    for (int j=0; j < nValues; j++)
15    {
16      eventbatch->eventValues[i*nValues+j] = e->GetValue(j);
17    }
18    eventbatch->eventWeights[i] = e->GetWeight();
19    eventbatch->eventIsSignal[i] = DataInfo().IsSignal(e);
20  }
21 }
```

All events are packed into an array in host memory, but these may not all be transferred to GPU memory in a single batch. If the number of events is large enough, they will be grouped into a number of smaller batches which are transferred and replaced as required in GPU memory. This allows the GPU implementation to scale even to large data sets.

Chapter 5

Training Multiple Networks

5.1 Motivation

The size and extent to which a neural network can be parallelised corresponds to the number of neurons within it. With a low number of input variables (and corresponding input neurons), the overall size of the Neural Network may not be large enough for parallelisation to be worthwhile.

However, the developers of TMVA suggested that a future improvement to the framework could involve allowing users to automatically select parameters for training, and the use hill climbing algorithms to tune these (personal communication, May 2011). GPU computing offers a potentially beneficial platform for training multiple networks simultaneously in order to investigate the feasibility of this goal.

5.2 Parallel Design

Modifying the existing implementation to allow multiple networks to be trained simultaneously first involved allowing for the allocation of additional space for neural network data. It was decided that continuing the data oriented approach applied in the initial GPU porting work would be suitable for this, so the original linear arrays of data were retained but extended to contain multiple networks. This means that arrays can now be indexed using an offset based both on thread id and on network id.

Each network is made to reside in a single thread block, allowing them to be processed

identically in parallel. While the current GPU implementation exists through a number of small kernels, this additionally offers scope for a larger kernel function to be developed in future, within which the neural network data could be copied from global to shared memory, potentially leading to further performance gains.

Events are still transferred to the GPU either as a complete set or in batches, however now each event is used to train a number of networks in parallel. In this way, the impossibility of event parallelism is compensated for by network parallelism.

5.2.1 Kernel Alterations

The chosen memory scheme allowed kernels to be used by multiple networks with minimal issues. Simply re-addressing the network data according to the network, and as such the thread block ID allowed the appropriate data to be accessed, as follows:

Listing 5.1: Activation function modified for use by multiple networks.

```
1 __global__ void CUDAk_ActivateSigmoid(float *neurons, int max,
      int numNeurons)
2 {
3   int id = CUDAk_GetId();
4   int block = CUDAk_GetOffsetNeuron(numNeurons);
5   if(id < max)
6   {
7     neurons[id + block] = 1.f/(1.f + __expf(-1.f * neurons[id +
          block]) );
8   }
9 }
```

Note that the block address is now retrieved, based on the total number of neurons in each network, as `blockIdx.x * numNeurons;` and then used as an offset when accessing arrays.

Chapter 6

Results and Analysis

6.1 Data Sets

For analysis and testing, two data sets were used:

- The TMVA Sample Data Set. This data consists of 6000 sample events, with 4 input variables.
- The ATLAS Data Set. This set consists of 32000 recorded events with 35 input variables.

6.2 Data Transfer Results

Figure 6.1 shows the results of transferring increasing numbers of events from the sample data set to the GPU in batches of 100, demonstrating a linear scaling as might be expected given the repeated work. Figure 6.2 demonstrates that increasing the batch size to match the total events can reduce transfer times, suggesting that as many events as possible should be transferred to and reside on the GPU at any one time as possible.

Tests were also performed with larger numbers of events, the ATLAS events rather than the TMVA Samples and using the Fermi rather than Tesla architecture, however the resulting values corresponded very closely with the above graphs so they have not been replicated here.

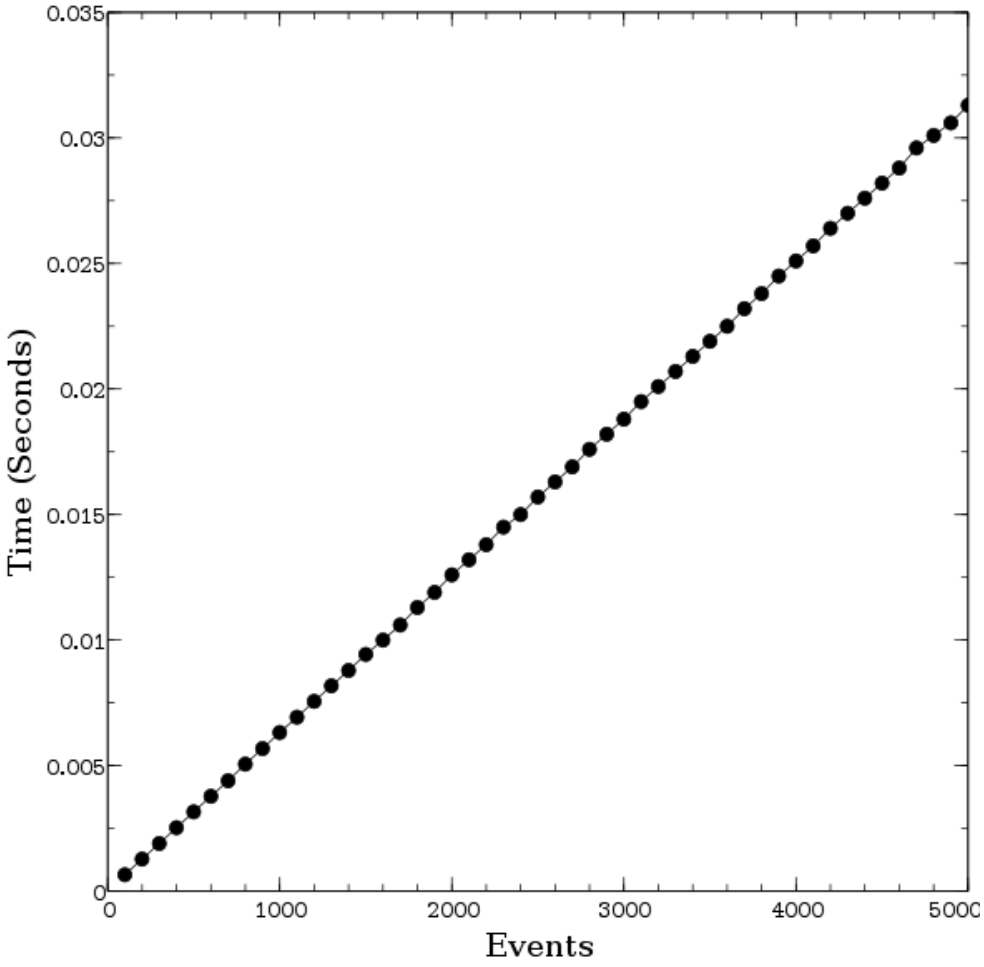

Figure 6.1: Timings for transferring increasing numbers of events to the Tesla GPU in batches of 100.

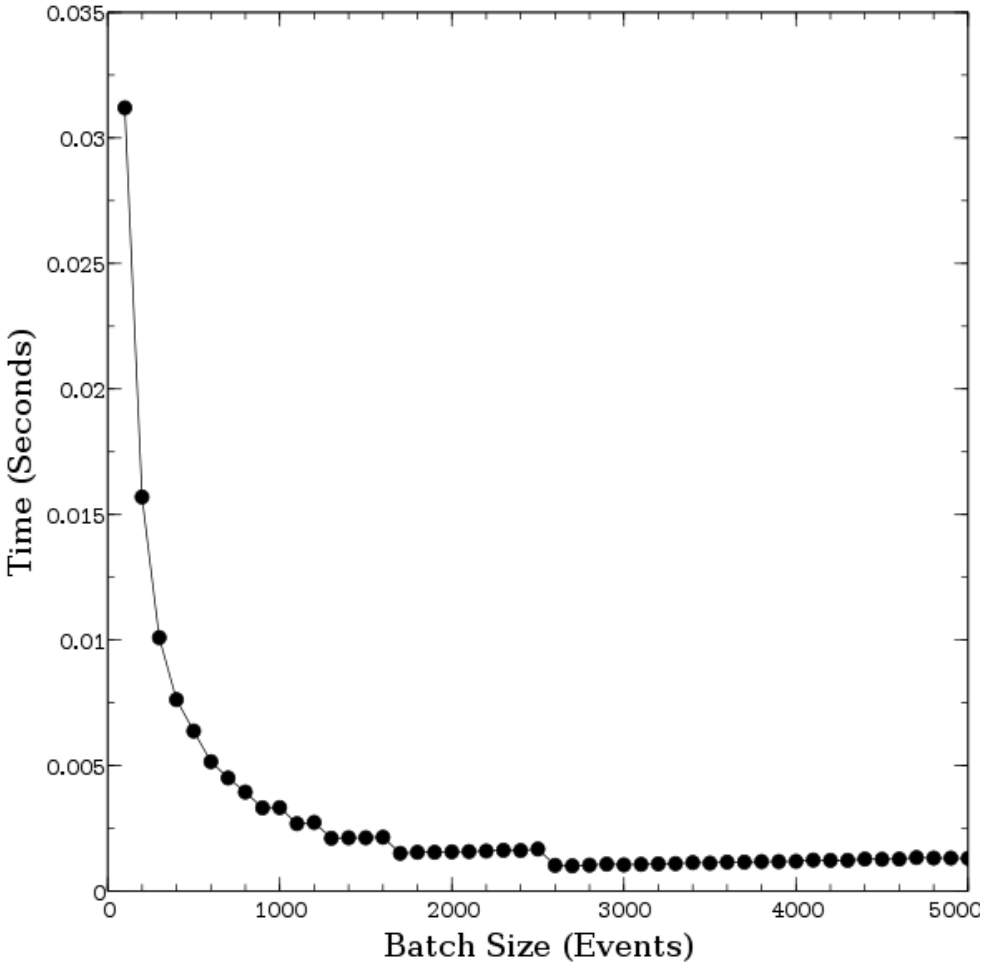

Figure 6.2: Timings for transferring 5000 events in batches of increasing sizes to the Tesla GPU.

6.3 Single Network Results

6.3.1 TMVA Sample Data Set

6.3.1.1 Tesla Architecture

Initial results using the Tesla architecture suggest the GPU implementation of a single MLP with four input variables giving 14 neurons takes around 119 seconds on GPU, as opposed to 18 seconds on CPU. While discouraging, this result is not altogether surprising given the amount of additional work which must be performed on both CPU and GPU in order to allocate memory, transfer data and network data and invoke kernels. With only four input variable the extent to which the network is parallelised is greatly limited, so these is still potential for gains with a larger problem size.

Figure 6.3 shows the results of passing events to the GPU throughout the execution of the algorithm in batches on 1000, whereas 6.4 shows the results of transferring all results in a single batch. The differences between these were clearly minimal. Section 6.2 clarifies why this is the case, showing that while transferring data in larger batches has performance benefits over smaller the times are not significant enough to make a difference given the extended processing time the MLP requires.

6.3.1.2 Fermi Architecture

Surprisingly, as shown in figure 6.5, the results of training a single network on the Fermi architecture is slower than both the Tesla architecture and training on CPU. There are a number of possible reasons for this such as the limited size of the network compared with overhead costs which might be increased on this hardware or the limited number of active threads; any future work on this project should perform more in-depth analysis of optimisation for this platform.

6.3.2 ATLAS Data Set

6.3.2.1 Tesla Architecture

Figures 6.6 and 6.7 demonstrate the results of training an MLP on the Tesla hardware using the larger ATLAS data set. In this case, training with the full data set on CPU

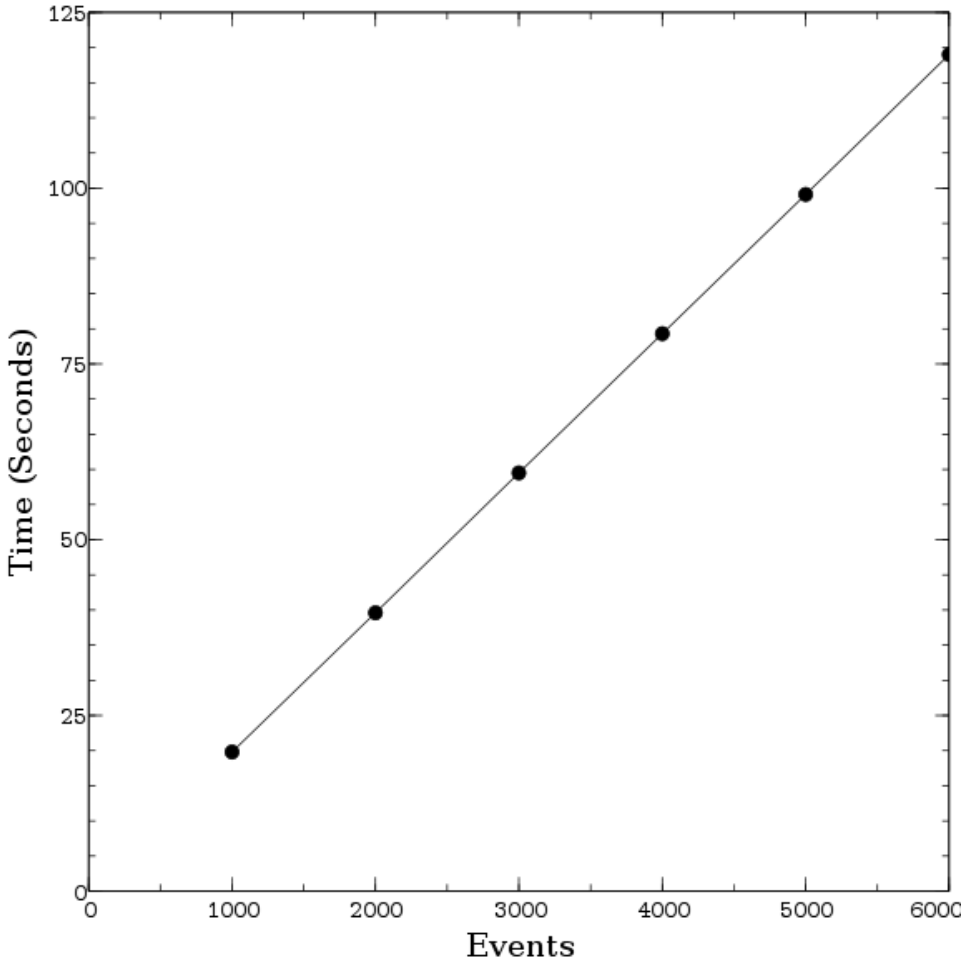

Figure 6.3: Results of training a single network with the TMVA data set, passing events to the GPU in batches, using the Tesla architecture.

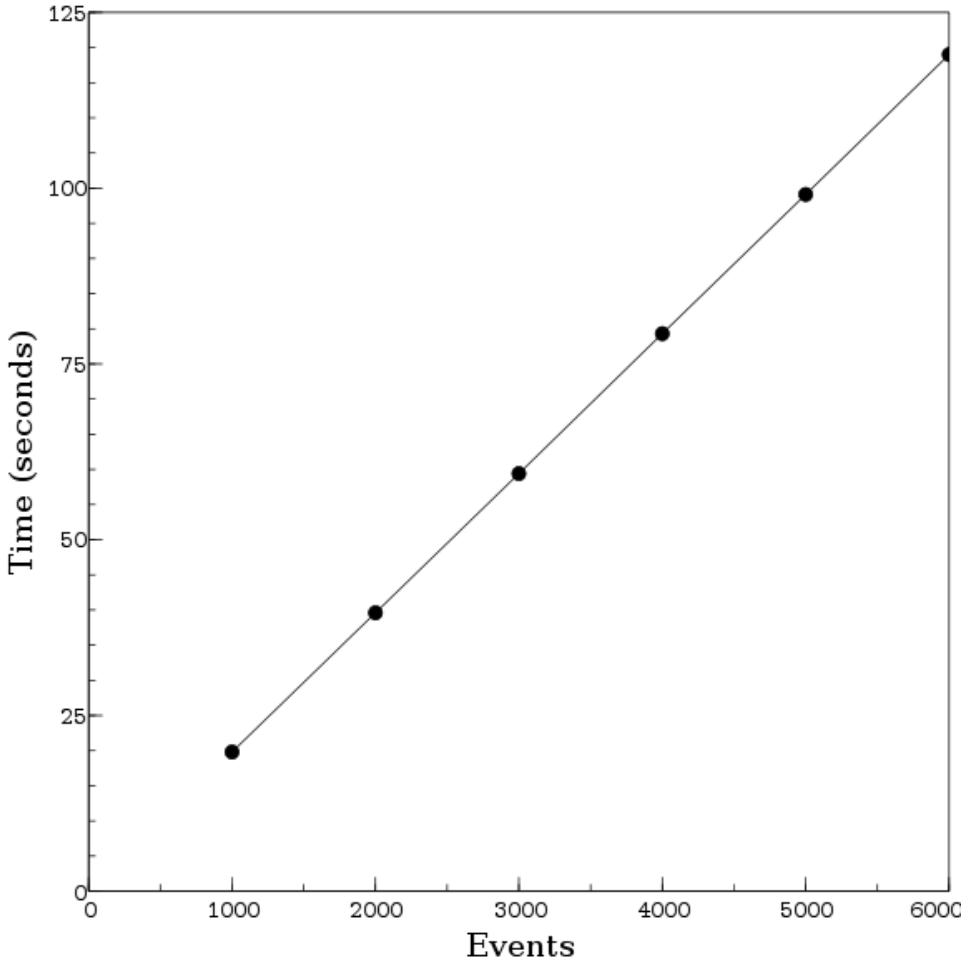

Figure 6.4: Results of training a single network with the TMVA data set, passing all events as a single batch, using the Tesla architecture.

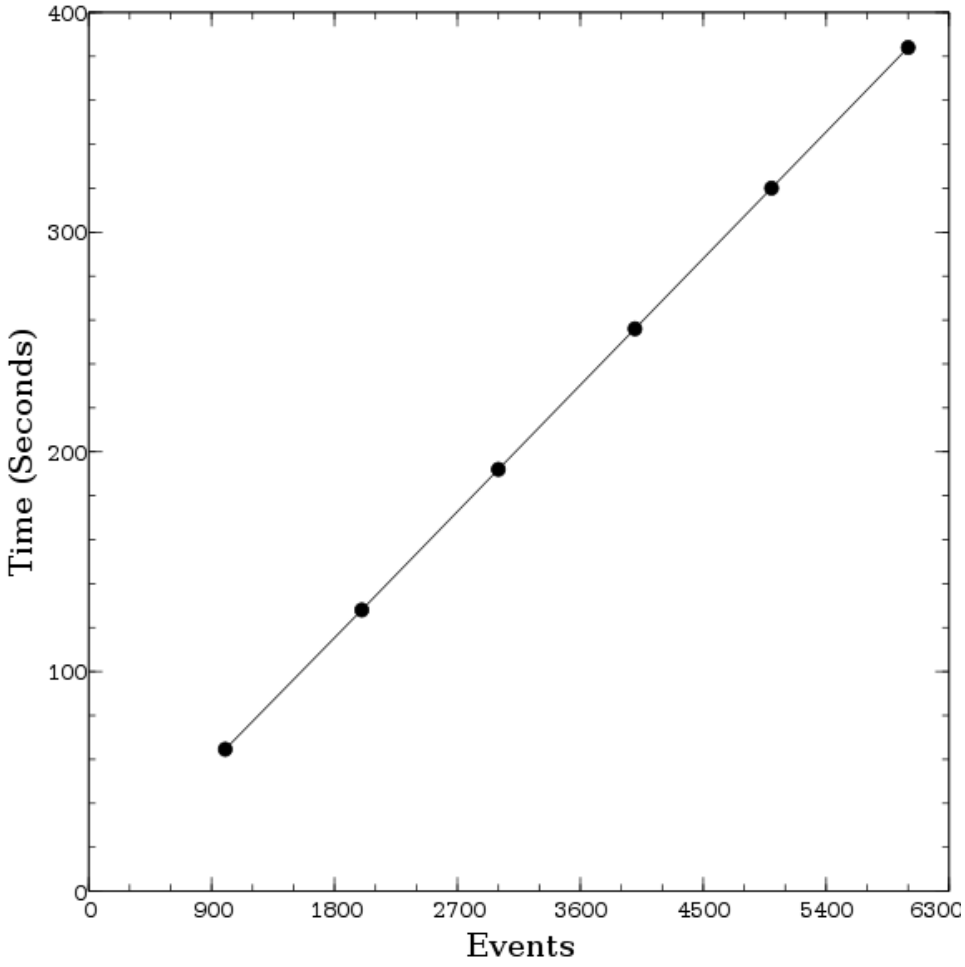

Figure 6.5: Results of training a single network with the TMVA data set, passing all events as a single batch, using the Fermi architecture.

took an average of 1060 seconds, whereas the GPU based algorithm is expected (based on the results up to 28,000 events) to perform almost 40% faster with an estimated time of 650 seconds. This is likely due to the larger number of input variables in the ATLAS data set; these lead to higher numbers of neurons which means the GPU based network can be parallelised to a greater extent.

6.3.2.2 Fermi Architecture

Again, the Fermi architecture demonstrates lower performance than the Tesla architecture as demonstrated in figure 6.8.

6.4 Multiple Networks

6.4.1 TMVA Sample Data Set

Increasing numbers of neural networks were trained simultaneously with the same event data, and time taken was found not to increase despite the additional GPU processing work. To train the entire set on CPU would take an average of 18 seconds, so as discussed training a single network on GPU resulted in a significant loss of performance on this set. However, as the results in figures 6.9 and 6.10 show, it is possible to train a number of neural networks in parallel with no increase is time taken. Training any more than 10 networks on the Tesla architecture offers a performance gain over training multiple networks on CPU, and this increase is only eventually limited by memory or the number of thread blocks available. With only 4 input variables giving around 19 neurons and 45 synapses, each of these networks will require less than a kilobyte of global device memory to process, so the gains are potentially very large.

6.4.2 ATLAS Data Set

The ATLAS data set results are consistent with those above, with the GPU based networks taking an average of 750 seconds to train, compared to the 1060 seconds taken on CPU. This result demonstrated the applicability of GPU computing to problems which can be scaled appropriately; we are able to train numerous networks on GPU in less time than it would previously have taken to train only one.

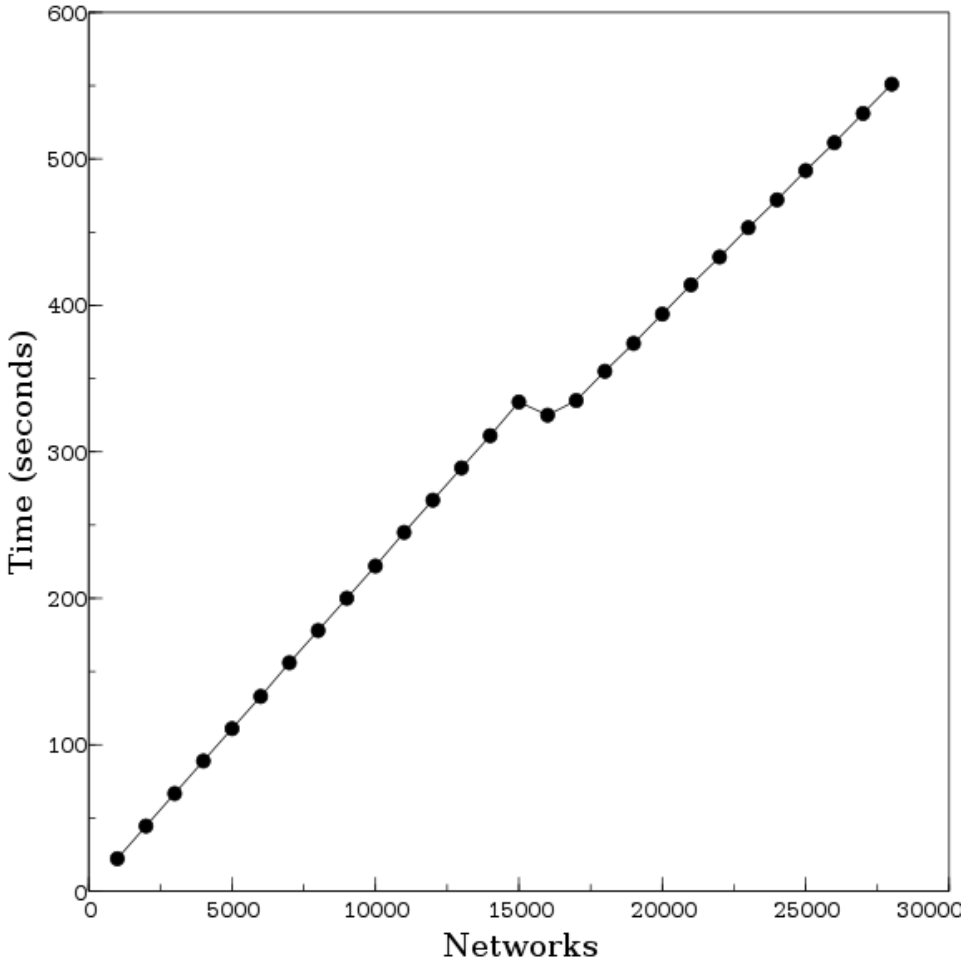

Figure 6.6: Results of training a single network with the ATLAS data set, passing all events as a single batch, using the Tesla architecture.

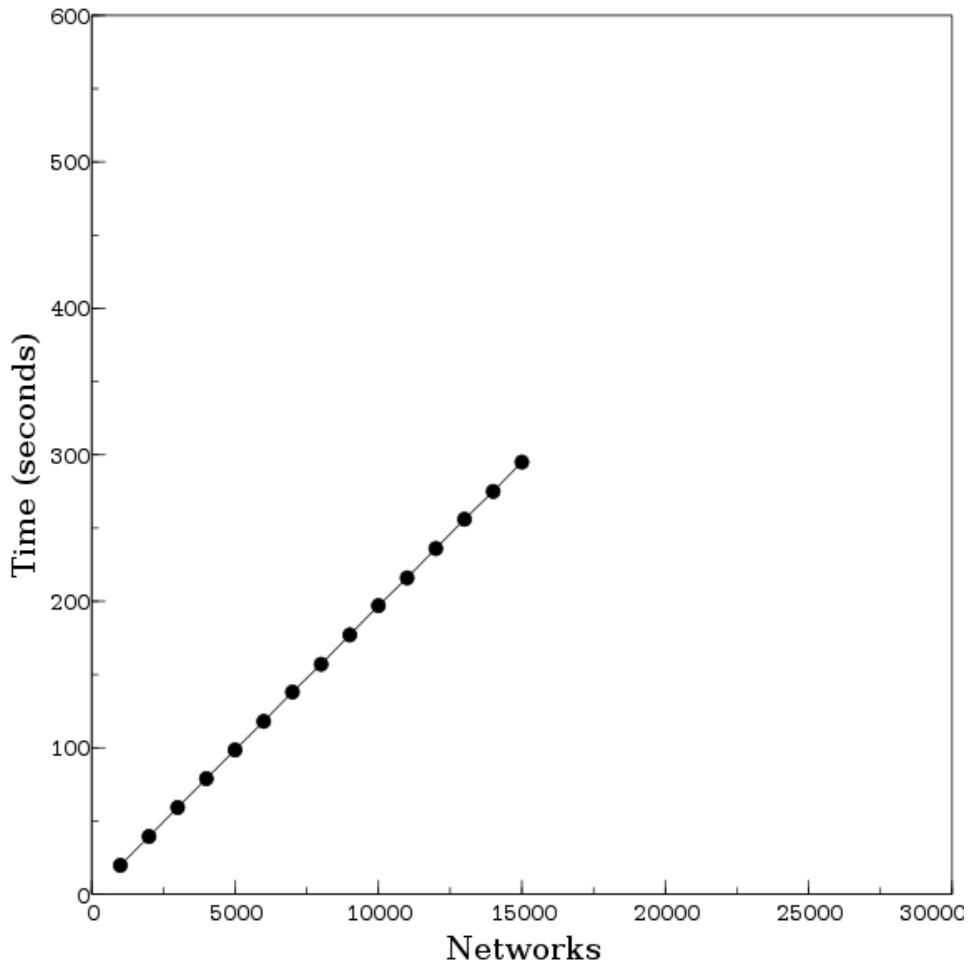

Figure 6.7: Results of training a single network with the ATLAS data set, passing events in a number of batches, using the Tesla architecture. Due to the extended training time for this data set, only up to 15,000 of the 32,000 events were tested however the results closely correspond to those from transferring events as a single batch.

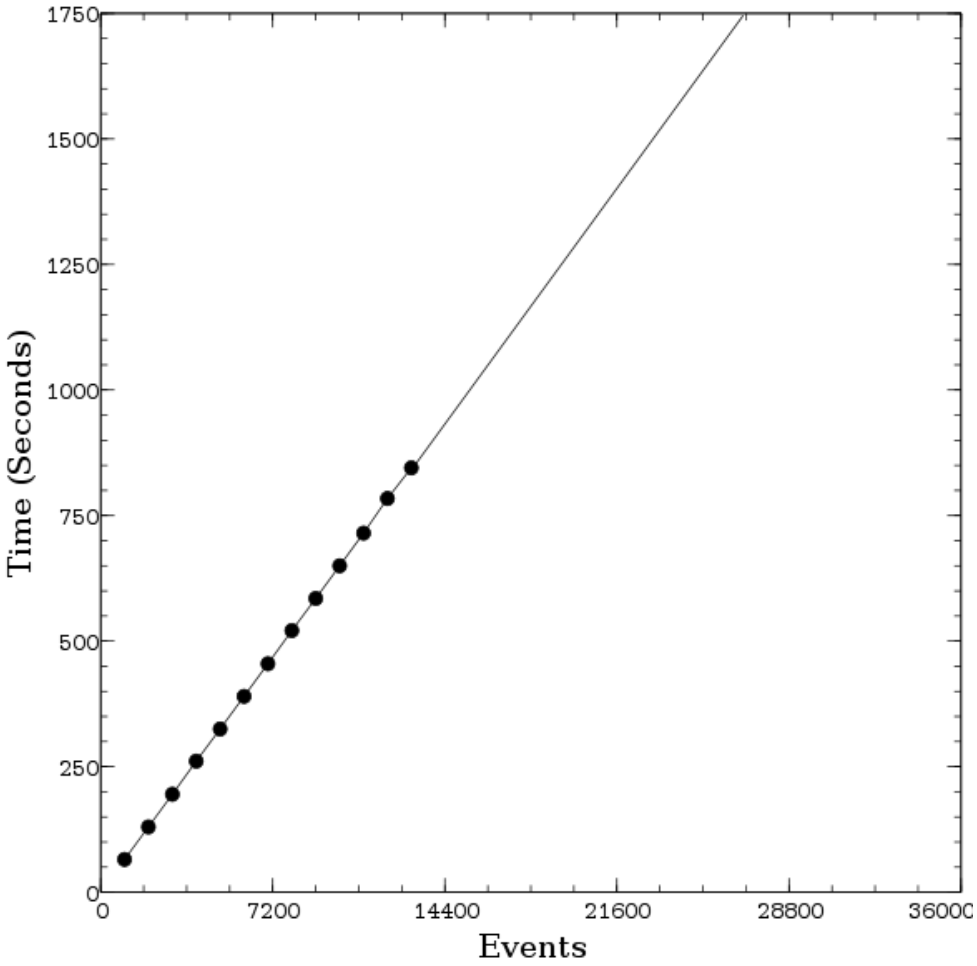

Figure 6.8: Results of training a single network with the ATLAS data set, passing events in a number of batches, using the Fermi architecture. Due to the extended training time for this data set, only 13,000 of the 32,000 events were tested.

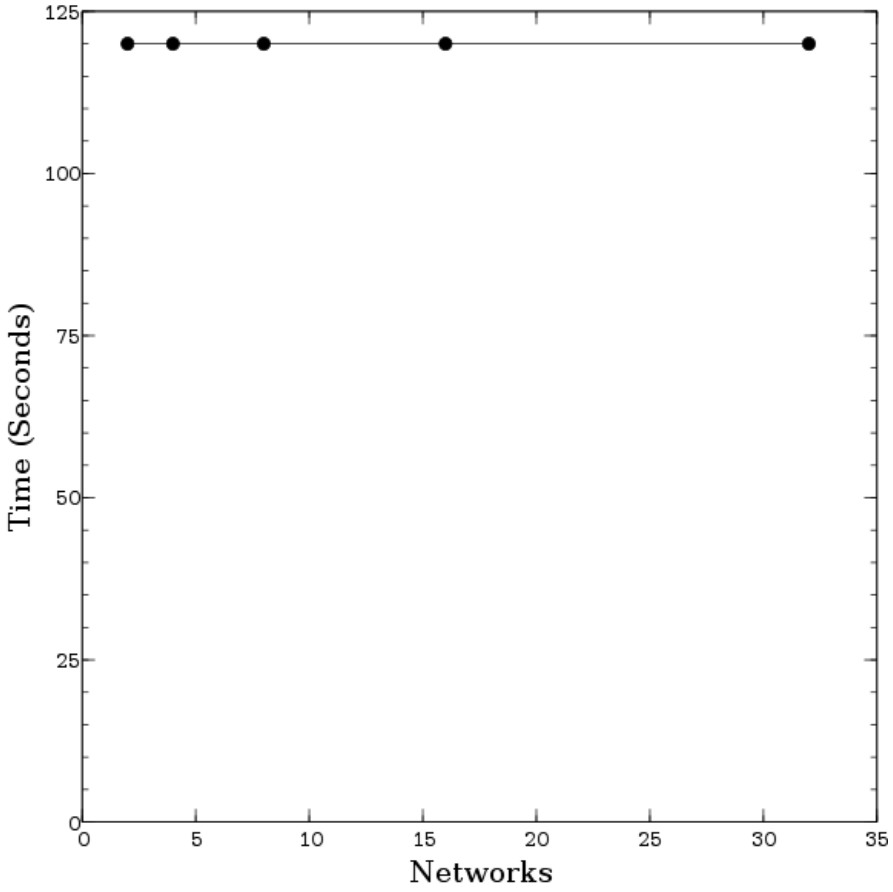

Figure 6.9: Results of training increasing numbers of neural networks in parallel with the TMVA data set, using the Tesla architecture.

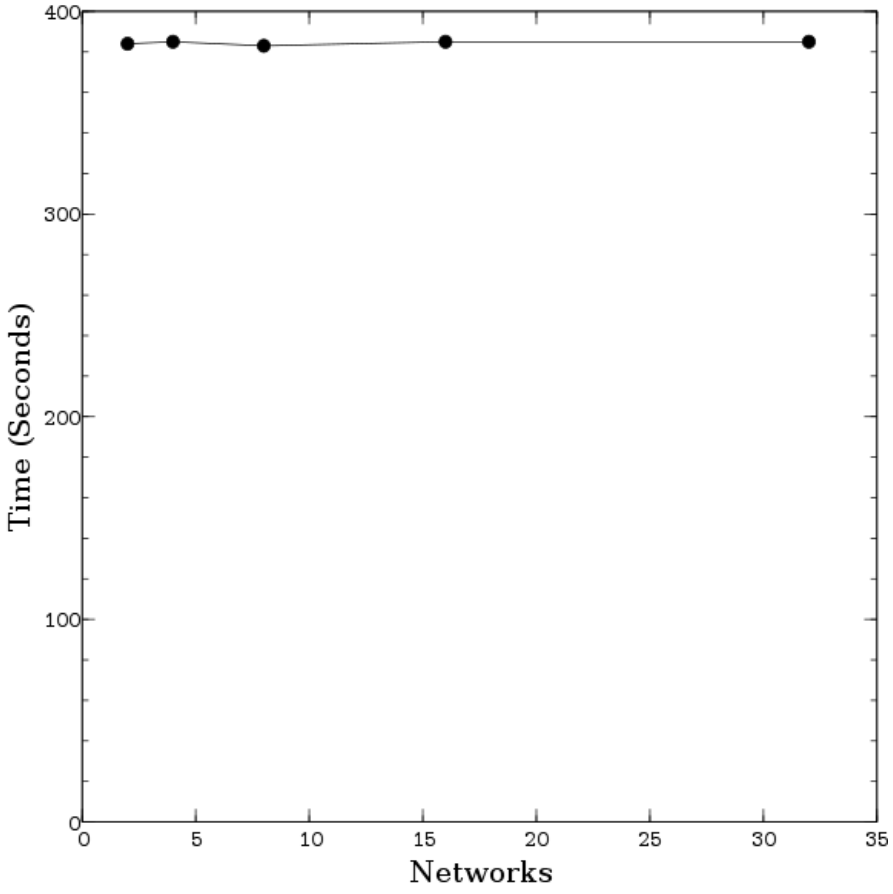

Figure 6.10: Results of training increasing numbers of neural networks in parallel with the TMVA data set, using the Fermi architecture.

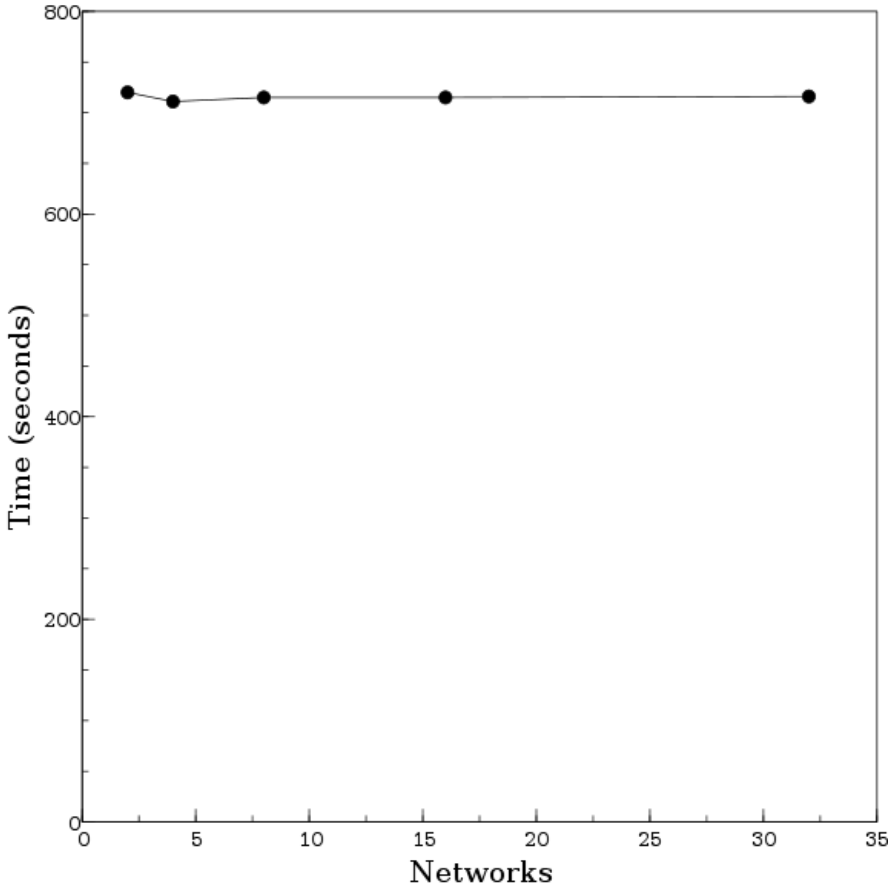

Figure 6.11: Results of training increasing numbers of neural networks in parallel with the ATLAS data set, using the Tesla architecture.

6.4.3 Further Discussion

Overall it was shown that performance benefits can be found by processing larger individual MLPs on GPU, and in particular when processing a number of networks simultaneously.

Amdahl's law states that the extent to which you can parallelise a problem is limited by the fraction of this problem which must be executed in serial. In terms of GPU computing, control from the host can provide this limiting factor. However, Gustafson's law considers the fact that a problem size can be scaled to match the available hardware and computing power, and it is this which we exploit by training a number of networks in parallel. While there does seem to be potential for performance gains when processing larger MLPs with CUDA, the most beneficial use of the hardware might be to apply it to the separate problem of training networks with differing learning rates or parameters in order to determine which are more suitable.

The unexpectedly slow results from the Fermi architecture however reinforce the fact that porting software to the GPU is not a straightforward task, and requires calculation, testing and iteration in order to be successful. Producing a framework which performs well on all available hardware is unlikely to be successful.

Chapter 7

Conclusion

7.1 Future Work

Given the work completed and the results obtained, there is considerable scope for continued work in this area. Suggestions for work to further the MLP implementation on GPU and in relation to TMVA as a whole are provided.

7.1.1 Bias Neuron

While TMVA's MLP implementation offers a vast number of features and configuration options, the most significant one not to have been included in the GPU implementation is the bias neuron. This is an additional neuron in each of the non-output layers of the network which can be used to shift the activation function, leading to faster or superior convergence. It was omitted from the original GPU implementation as its inclusion would likely necessitate minor branching in the kernel code, however in order for the GPU implementation to match the original codebase it is a necessary addition.

It could be implemented by increasing the size of network layers, and also storing data about whether the value of each neuron has been naturally propagated or "forced" to a certain values, which is the case with input and bias neurons and is the method of implementation found in TMVA.

7.1.2 TMVA Options

TMVA's MLP class is just one of several neural network implementations provided by TMVA, yet even it provides a number of options for user configuration, such as sampling events rather than using a complete dataset. As Listing 7.1 shows, this tends to be catered for by simply branching within individual functions.

Listing 7.1: Branching code in TMVA based on user options

```
1 void TMVA::MethodMLP::Train(Int_t nEpochs)
2 {
3 ...
4   if (fTrainingMethod == kGA) GeneticMinimize();
5   else if (fTrainingMethod == kBFGS) BFGSMinimize(nEpochs);
6   else BackPropagationMinimize(nEpochs);
7 ...
8 }
```

While branching does not always necessitate a major performance hit on the GPU (so long as all threads can take the same branch), this flexibility does cause difficulties for porting the codebase to CUDA. This is in part due to the quantity of data which must be made available, and also due to the complex code flow which this entails within an object-oriented structure to adhere to.

However, in order for TMVA to successfully make use of GPU based techniques the design ideals of the framework must be kept in-tact; flexibility must be afforded to the user. Options for successfully implementing this include making use or pre-processor macros and forcing decisions about parameters to be made a compile time rather than run-time, allowing for a greater amount of code to be replicated by providing separate kernel functions for different options, or sacrificing performance and space by creating a general-purpose kernel.

7.1.3 Hill Climbing with MLP Parameters

The project successfully demonstrated that multiple MLPs can be run in parallel without increasing time requirements. The intention was to use this to allow options to automatically be chosen or tuned using hill-climbing techniques; determining the most suitable network configuration for a given dataset by iteratively adjusting parameters.

Given the timing results for training multiple networks, there is clearly scope for this to be achieved effectively on GPU.

To pursue this further, appropriate parameters for tuning must be selected, and a method of selecting the most suitable network must be developed. Three alternative parameters which would be suited to this task would be:

- The configuration of the network in terms of the number of neurons on each hidden layer, as the linear array structure implemented easily allows this to be altered for each network trained.
- The learning rate, as this is represented by a single floating point value but alterations to it can greatly affect the convergence of a network.
- The activation function for neurons, as a number have been implemented on GPU as part of this project and it is not normally possible to determine simply by looking at a data set which is most appropriate for classification.

Determining the most suitable network could be done by testing each with a small population of events in parallel on GPU, and determining which classifies the highest percentage correctly. This could also be performed after fewer training epochs to test for convergence, ceasing training if it is no longer necessary.

7.1.4 Improved Memory Use

At present, all network and event data is stored in global memory, while shared memory offers a faster alternative and is currently unused. The current network implementation consists of a number of small kernel functions controlled from the host. However, it would be possible to combine these into a single larger kernel, thus reducing the amount of work required in invoking kernels. Additionally, this might lengthen the lifespan of the kernel to the point where it becomes worthwhile to copy data for each network into it's own block's shared memory, leading to improved access time.

7.2 Applicability to TMVA

As a feasibility into the use of GPUs to accelerate analysis in TMVA, this project has obtained promising performance results, but also discovered a number of issues and

challenges.

The most significant barrier to porting TMVA to GPU is its modular and object-oriented structure, which necessitates that a great deal of code be completely rewritten and restructured for GPU computing. This requires the developer to have a full understanding of each technique to be ported in order to determine where parallelism can safely be found, as well as understanding the intricacies of GPU programming. As such, it is also difficult to determine whether the success of porting MLPs to GPU indicates that this will also be possible for other techniques in TMVA.

An important point to draw from this study is that porting an existing software base leads to additional challenges which are not present when writing new GPU code. There is an expectation that GPU methods will produce results which correspond exactly with the original CPU based output, but in many cases that is difficult and may not even be possible. TMVA provides a number of alternative approaches to each machine-learning technique it implements along with a great number of parameters for tuning, and providing all of these or even an exact match for a set of these when re-writing code is far more difficult than simply providing a working implementation of some method.

However, training large MLPs on GPU did offer a speed gain over CPU based training on the Tesla hardware. As such, there is certainly potential for GPU computing to be used to accelerate analysis in TMVA, and as detailed above there is scope for performance improvement within this individual technique.

The most significant result of this study was determining that regardless of the size of performance gain found for individual networks, the additional computing power offered by GPUs can be leveraged for other useful work. In this case, we trained multiple neural networks simultaneously, which in future could be used as a way to automatically select and tune parameters to find the optimal configuration of a given technique.

In conclusion, GPU computing can be a challenging area in which to seek performance gains, but the pervasiveness, cost and continued improvement of graphics card hardware means it is an area which cannot be ignored. The structure and flexibility of TMVA cause some difficulties in porting it to CUDA, however the initial results from this study indicate that continuing this work could see further performance gains or alternative uses for GPU parallelism in the framework.

Bibliography

[1] "ATLAS experiment factsheets." http://www.atlas.ch/fact-sheets-view.html.

[2] "ROOT." http://root.cern.ch/.

[3] A. Hoecker, P. Speckmayer, J. Stelzer, J. Therhaag, E. von Toerne, and H. Voss, "TMVA: toolkit for multivariate data analysis," *PoS*, vol. ACAT, p. 040, 2007.

[4] A. Farbin, "Emerging computing technologies in high energy physics," *Arxiv preprint arXiv:0910.3440*, 2009.

[5] H. Sutter, "The free lunch is over: a fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, 2005.

[6] "CUDA." http://www.nvidia.com/object/cuda_home_new.html.

[7] "OpenCL." http://www.khronos.org/opencl/.

[8] "GPU technology conference - NVIDIA." http://www.gputechconf.com/page/home.html.

[9] "TOP500 list - june 2011 (1-100) | TOP500 supercomputing sites." http://www.top500.org/list/2011/06/100.

[10] P. Desprs, J. Rinkel, B. H. Hasegawa, and S. Prevrhal, "Stream processors: a new platform for monte carlo calculations," in *Journal of Physics: Conference Series*, vol. 102, p. 012007, 2008.

[11] A. Choong, R. Beidas, and J. Zhu, "Parallelizing simulated Annealing-Based placement using GPGPU," in *2010 International Conference on Field Programmable Logic and Applications*, (Milan, Italy), pp. 31–34, Aug. 2010.

[12] Q. Yu, C. Chen, and Z. Pan, "Parallel genetic algorithms on programmable graphics hardware," *Advances in Natural Computation*, p. 10511059, 2005.

[13] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *Proceedings of the 25th international conference on Machine learning*, ICML '08, (New York, NY, USA), p. 104111, ACM, 2008. ACM ID: 1390170.

[14] A. Guzhva, S. Dolenko, and I. Persiantsev, "Multifold acceleration of neural network computations using GPU," in *Artificial Neural Networks ICANN 2009* (C. Alippi, M. Polycarpou, C. Panayiotou, and G. Ellinas, eds.), vol. 5768 of *Lecture Notes in Computer Science*, pp. 373–380, Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-04274-4_39.

[15] Z. Luo, H. Liu, and X. Wu, "Artificial neural network computation on graphic process unit," in *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, vol. 1, p. 622626, 2005.