GPU Acceleration of Logistic Regression with CUDA

P.K.K.Madhawa, M.S.Jeevananda, P.M.B.C. Malmi, U.R.V.Sandaruwan, K.Wimalawarne
Dept. of Computer Science and Engineering
University of Moratuwa
Moratuwa, Sri Lanka

***Abstract*-- Logistic regression (LR) is a widely used machine learning algorithm. It is regarded unsuitably slow for high dimensional problems compared to other machine learning algorithms such as SVM, decision trees and Bayes classifier. In this paper we utilize the data parallel nature of the algorithm to implement it on NVidia GPUs. We have implemented this GPU-based LR on the newest generation GPU with Compute Unified Device Architecture (CUDA). Our GPU implementation is based on BFGS optimization method. This implementation was extended to multiple GPU and cluster environment. This paper describes the performance gain while using GPU environment.**

Keywords: machine learning, classification, CUDA, logistic regression, GPGPU

I. INTRODUCTION

Classification is the process of assigning given input data in to one of the given number of categories for its most effective and efficient use. Logistic regression (LR) is one of the oldest classification algorithms found in machine learning literature which is used for prediction of the probability of occurrence of an event by fitting data to a logit function logistic curve [1]. The major drawback of the LR algorithm is its slow compared to other classification algorithms.

Several machine learning researchers have introduced several methods to improve the efficiency of the algorithm [2]. With the advent of multi-core machines and distributed computing the inherently parallel nature of the algorithm is utilized and being used in many production fraud detection and advertising quality and targeting products [1]. Apache Mahout has a LR implementation which runs on top of distributed computing framework Apache hadoop [3]. Researchers at Microsoft research have developed a multi-core LR algorithm which is included in Microsoft Sigma machine learning toolkit [4].

This paper presents an implementation of LR algorithm to be run on general purpose Graphical Processing Units (GPU). Nowadays, most desktop computers are equipped with programmable GPUs with plenty powerful Single Instruction Multiple Data (SIMD) processors that can support parallel data processing and high-precision computation.

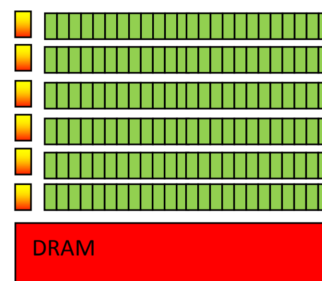

Figure 1: The GPU Devotes More Transistors to Data Processing

The massively parallel nature of the GPU architecture is shown in Fig. 1.

GPU has evolved into a highly parallel, multi-threaded, many core processor with tremendous computational horsepower and very high memory bandwidth. GPU threads are executed in SIMD (Single Instruction Multiple Data) and manage by the hardware.

In this architecture we keep the GPU as co-processor to the CPU such that we can keep CPU-based storage and transfer between CPU and GPU. CUDA is the language which is supported for the NVIDIA graphics card. There is no support for recursion and iterative functions which has conditional clauses. We have to merge the completed work run in CPU and GPU separately.

We implemented all the large scale matrix and vector operations to be run on the GPU.

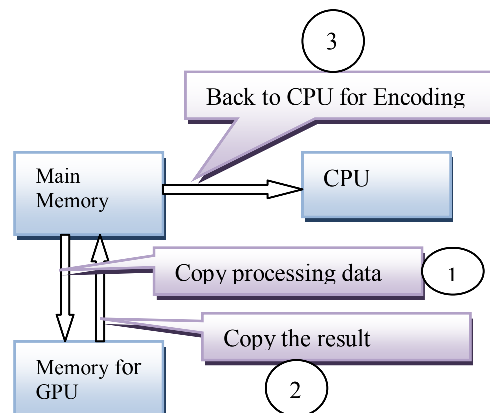

Figure 2: Transfer data between CPU and GPU

A. General purpose GPU Computing

The functionality of Graphics Processing Units (GPU) has, traditionally, been very limited. In fact, for many years the GPU was only used to accelerate certain parts of the graphics pipeline. Once specially designed for computer graphics and difficult to program, today's GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C. With the introduction of the CUDA architecture GPU established as a general purpose computing device which doesn't need any graphic APIs [5].
CUDA represents the co-processor as a device that can run a large number of threads. The threads are managed by representing parallel tasks as kernels (the sequence of work to be done in each thread) mapped over a domain (the set of threads to be invoked) [6]. Kernels are scalar and represent the work to be done at a single point in the domain. The kernel is then invoked as a thread at every point in the domain. The parallel threads share memory and synchronize using barriers. Parallelism can be obtained using GPUs through data parallelism, thread parallelism and task parallelism.

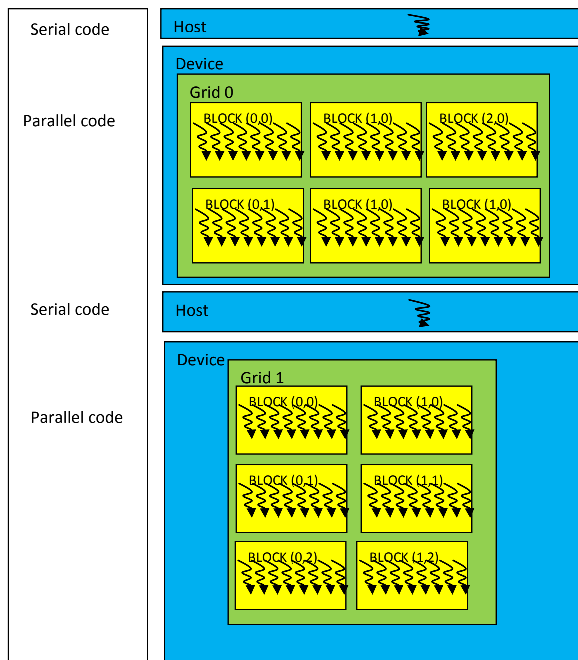

Figure 3: Serial code executes on the host while parallel code executes on the device.

B. Memory hierarchy of the GPU

GPU's memory system creates a branch in a modern computer's memory hierarchy. The GPU, just like a CPU, has its own caches and registers to accelerate data access during computation. GPUs, however, also have their own main memory with its own address space meaning that programmers must explicitly copy data into GPU memory before beginning program execution.

Programmers can partition the problem into sub problems that can be solved independently in parallel by blocks of threads. (such as a figure number). GPU has on board device memory, which has high bandwidth and a high latency. Threads within the same thread block are divided into SIMD groups, called warps. A warp of threads can combine accesses to consecutive data items in one device memory segment into a single memory access transaction, or called coalesced access.

Data is prepared for processing on the GPU by copying it to the graphics board's memory. Data transfer is performed using DMA and can take place concurrently with kernel processing. Once written, data on the GPU is persistent unless it is deallocated or overwritten, remaining available for subsequent kernels.
But all these massive parallelism comes with a price. The fact that they adopt the SIMD / MIMD parallel processing scheme means that algorithms only benefit from the architectural properties, if they can be adapted to that scheme. Moreover, important fundamental constructs such as integer data operands are missing and associated operations such as bit-shifts and bitwise logical operations. The GPU lacks 64-bit double precision number formats.

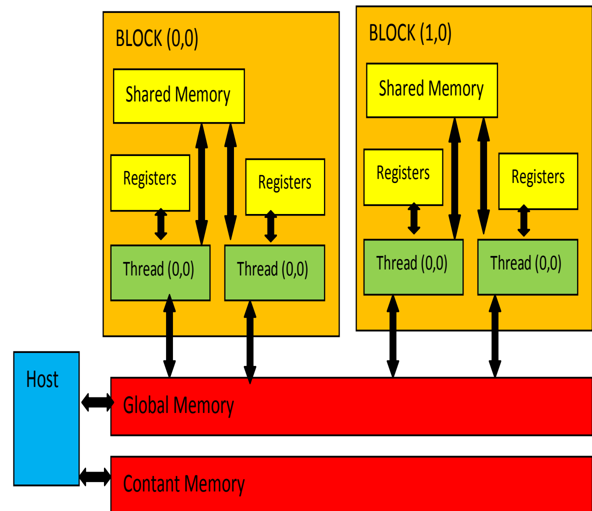

Figure 4: Hardware implementation of the CUDA memory

II. LOGISTIC REGRESSION

Logistic regression is a simple model for predicting the probability of event and is often used for binary classification. When the possible outcomes are coded as 0 and 1, we can train the logistic regression model that will predict the probability of the second event.

A. Binary logistic model

$$\Delta w = -H^{-1} * \nabla f(w)$$

The logistic function is defined as

$$f(z)=\frac{e^{z}}{1+e^{z}}=\frac{1}{1+e^{-z}}$$

The logistic function is useful because it can take any value as an input from negative infinity to positive infinity, whereas the output is confined to values between 0 and 1. The output f(z) represents the probability of a particular outcome, given the set of explanatory variables.
The variable *z* is a measure of the total contribution of all the independent variables used in the model and is known as the logit.

The variable z is defined as

$$z = w_0 + w_1 * x_1 + w_2 * x_2 + ...w_k * x_k$$

Where w_0 is called intercept and $w_{1,} w_{2,}.., w_k$ are called regression coefficients.
Methods such as Maximum likelihood estimation and iteratively re-weighted squares are used for parameter estimation.

B. Newton's method

Newton method updates the weight vector (w) in the following way [7]

$$w^{k+1} = w^k + s^k$$

Where k is the iteration index and s^k , the Newton direction, is the solution of the following linear system:

$$\nabla^2 f\left(w^k\right)s^k = -\nabla f\left(w^k\right)$$

Where $\nabla^2 f(w)$ is the Hessian of f(w) and $\nabla f(w)$ is the gradient vector of f(w).

Two types of methods are used for solving the above linear system: direct methods (e.g., Gaussian elimination), and iterative methods (e.g., Jacobi and conjugate gradient).

Due to the low number of iterations, conjugate gradient method is the most widely used method to solve this linear system.

The formulas used to find the hessian matrix and the gradient vector of the function f(w) are shown below.

$$\nabla f(w) = w + C\sum_{i=1}^{l}(\sigma(y_i w^T x_i) - 1)y_i x_i,$$

$$\nabla^2 f(w) = I + CX^T DX$$

Where I is an identity matrix and D is a diagonal matrix.

$$D_{ii} = \sigma(y_i \mathbf{w}^T \mathbf{x}_i)(1 - \sigma(y_i \mathbf{w}^T \mathbf{x}_i)), \text{ and } X = \begin{bmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_l^T \end{bmatrix}$$

The disadvantages of Newton's method are the added cost of finding the Hessian matrix and the cost of solving a linear system to find the inverse Hessian matrix. Finding the diagonal matrix D on the GPU doesn't utilize the hardware, since only the diagonal elements are calculated.

Quasi-Newton methods are used to overcome these problems.

.

C. BFGS method

Instead of finding the inverse of the Hessian matrix inverse Hessian is approximated inside each iteration. The algorithm starts with $H^{-1} = I$. At each step w is updated as [8]

$$\Delta w = -H^{-1} * \nabla f(w)$$

$$w(k+1) = w(k) + \Delta w$$

In the BFGS algorithm, the H^{-1} is updated as below.

$$b = 1 + \frac{\Delta g^T H^{-1} * \nabla f(w)}{\Delta w^T \Delta g}$$

$$H_{new}^{-1} = H^{-1} + \frac{1}{\Delta w^T \Delta g}(b\Delta w \Delta w^T - \Delta w \Delta g^T H^{-1} - H^{-1}\Delta g \Delta w^T)$$

Advantages of this method are, this doesn't need calculation of the Hessian matrix and its inverse in each iteration and most of the matrix and vector operations can be easily parallelized using GPU architecture.

III. IMPLEMENTATION

At the implementation our objective was to minimize the number of kernel methods in order to reduce the time wasted on thread initialization and synchronization. Nevertheless the number of kernel methods cannot be reduced to one, since synchronization can be done only among threads in a single block. Independent tasks in kernel methods were identified and task parallelism approach was employed in order to improve the performance further.

Using time complexity analysis, tasks which had a higher time complexity (ex: - matrix multiplications) were identified. Shared memory architecture was employed to mitigate the time overruns in the GPU. These time overruns occurred due to the heavy data transfers of the matrix multiplication between the global memory and the registers of the GPU. By using the shared memory, these time overruns could be mitigated significantly.

Even though shared memory provides such a performance gain, it is a limited. Therefore it is not possible to use this approach as it is for any data set. In order to overcome this we used an approach, which divides the matrices in to sub matrices and do the operations for sub matrices independently and combine the solutions at the end.

IV. EXPERIMENTAL RESULTS

In this section, we present our experimental results on classification and training of the Logistic Regression on GPU.

A. Experimental Setup

Our experiments were performed on a PC with an NVIDIA GTX480 GPU and an Intel Core i7 CPU, running on Ubuntu 10.04 Lucid. The GPU consists of 15 SIMD multi-processors, each of which has 32 processors running at 1.45 GHZ. The GPU memory is 1.5GB with the peak bandwidth of 141.7 GB/sec. The CPU has 8 cores running at 2.4 GHZ. The main memory is 8 GB with the peak bandwidth of 5.6 GB/sec. The GPU uses a PCI-E bus to transfer data between the GPU memory and the main memory with a theoretical bandwidth of 4 GB/sec. The PC has a 160 GB SATA magnetic hard disk. All source code was written in C and compiled using gcc. The version for CUDA was 4.0.

Comparison: We compared the GPU based implementation with a CPU based implementation

Data sets: For testing purposes we used two large data sets which have different number of feature parameters.

We obtained handwritten digits dataset from MNIST[10] which has 784 features and training set of 60,000 examples. Other than that we used internet advertisement data set obtained from the Machine Learning Repository of University of California, Irvine which has 1584 features [11]

Metric: We measured the elapsed time for the completion of the training process. Since comparison experiments were conducted on the same input file we excluded the initial file input from the total time measurement.

B. Results

figure 5: shows the total running time of training process of the algorithm for MNIST dataset to all implementation

.

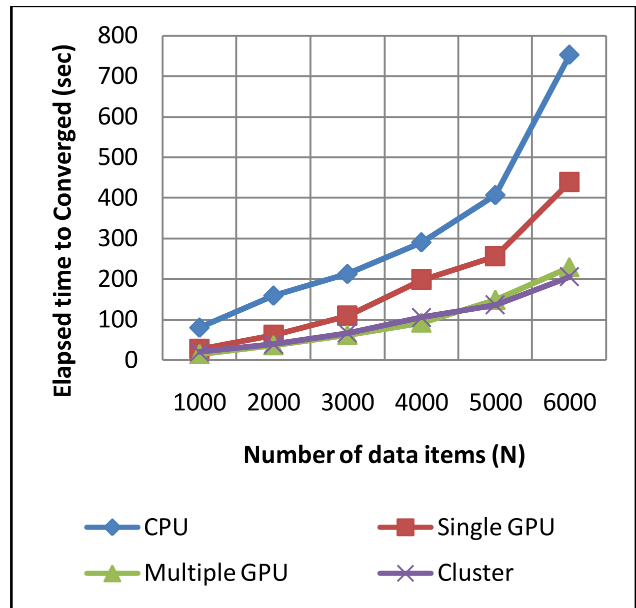

Figure 5: Elapsed Time with varying number of data objects

Figure6: shows the total running time of training process of the algorithm for internet advertisement dataset to all implementation

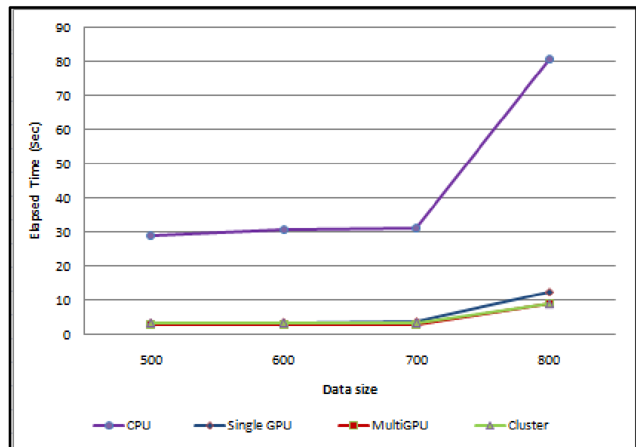

Figure 6: Elapsed Time with varying number of data objects for a dataset with large number of features

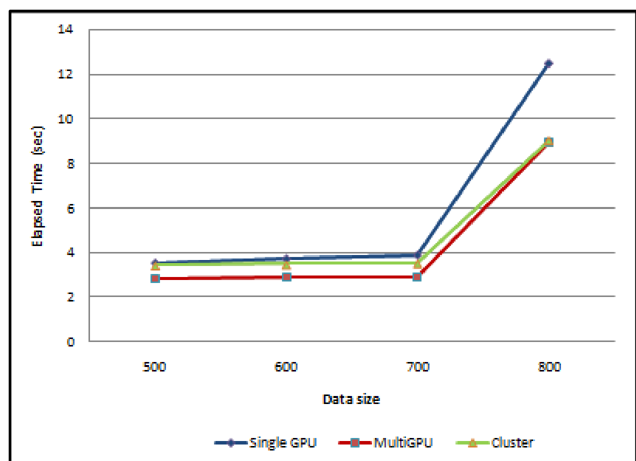

Figure 7: Highlighted GPU performances

V CONCLUTION AND FUTURE WORK

In this research we showed that, by parallelizing the time consuming matrix and vector operations on the GPU can reduce the running time by a significant amount. Our primary purpose in this research is to prove that developing GPU-aware data mining software is possible and useful.

We have studied detailed performance of our algorithm. In particular, we have examined the time breakdown of our GPU-accelerated Logistic Regression algorithm and have come to the conclusion that the Logistic Regression algorithm cannot be improved further using GPU accelerations since to be accelerated by the GPU by a huge factor it is required that the paralleled tasks need to have complex operations.

ACKNOWLEDGEMENTS

The GPU cluster was sponsored by the LK-Domain.

REFERENCES

[1] Paul R. Komarek, Andrew Moore, "*Logistic Regression for Data Mining and High-Dimensional Classification*", Carnegie Mellon Univ., May, 2004

[2] Paul R. Komarek , AndrewW. Moore ,"*Fast Robust Logistic Regression for Large Sparse Datasets with Binary Outputs"* Dept. of Mathematical Sciences. Carnegie Mellon University October 22, 2003 (revised Mar 26, 2007)

[3] Cheng T. Tau *et al* "Map-Reduce for Machine Learning on Multicore", *Advances in Neural Information Processing Systems* , Massachusetts Institute of Technology, 2007

[4] Thomas P. Minka, "*A comparison of numerical optimizers for logistic regression*"

[5] NVIDIA CUDA C Programming Guide, 3.2version, NVIDIA Corporation, USA, 2010.

[6] NVIDIA (2011, July 30) [online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[7] CJ Lin et al, "Trust Region Newton Method for Large-Scale Logistic Regression", Journal of Machine Learning Research, 9 (2008) 627-650 , 2009

[8] Malik Hj. Abu Hassan, Mansor B. Monsi & Leong Wah June , "Limited Modified BFGS Method for Large-Scale Optimization"

[9] G. Holmes; A. Donkin and I.H. Witten (1994). "Weka: A machine learning workbench". *Proc Second Australia and New Zealand Conference on Intelligent Information Systems, Brisbane, Australia*. Retrieved 2007-06-25.

[10] professor ,Yann LeCun, THE MNIST DATABASE[online]. Avaliable *:http://yann.lecun.com/exdb/mnist/*

[11] UCI, machine learning repository, internet data advertisement,[online] . Available*:http://archive.ics.uci.edu/ml/datasets/Internet+Advertisements [online]*