Data Regression with Normal Equation on GPU using CUDA

Vaibhav Mohan
School of Information Technology and Engineering
Vellore Institute of Technology
Vellore, India

Mayank Gupta
School of Information Technology and Engineering
Vellore Institute of Technology
Vellore, India

Abstract— **Demand in the consumer market for graphics hardware that accelerates rendering of 3D images has resulted in Graphic Cards that are capable of delivering astonishing levels of performance. These results were achieved by specifically tailoring the hardware for the target domain. As graphics accelerators become increasingly programmable however, this performance has made them an attractive target for other domains. Graphic processing units provide a low-cost parallel computing architecture. It is possible to achieve massive parallelism by SIMD (Single Instruction Multiple Data) on General Purpose Graphics Processing Unit (GPGPU) integrated with Central Processing Unit (CPU).**

In this implementation, Normal Equation Algorithm is used to achieve parallelism in data regression on a set of data given using a programming model, Compute Unified Device Architecture (CUDA) which uses multithreading technique. Normal Equation is one of the algorithms to predict, forecast, mine huge amount of data. Normal Equation using CUDA can achieve high performance. Here, Normal Equation is implemented on Graphics Processing Unit (GPU) and on CPU to process given datasets for prediction of patterns by finding weights of the Regression model. The time spent for computation is compared in both the cases.

Keywords- Central Processing Unit; Graphics Processing Unit; Normal Equations; CUDA; Multithreading.

I. INTRODUCTION

Files with huge amount of data sets consume enormous amount of time to process on CPU. It is time consuming to perform many operations on it thereby resulting in degradation of performance. CUDA is such a programming model by which performance in terms of time for any computation is improved [4].

Machine learning technique analyses the relationship between the two variables, *X* and *Y* [1]. For each subject, both *X* and *Y* are known and we want to find the straight line that fits best through the mass of data. We have a lot of applications of this model. In some scenario, the slope and intercept of the line might have scientific meaning. In other cases, the resulting fitted model can be used to summarize the data, to predict unobserved values from the same system, and to understand the mechanisms that may underlie the system. These algorithms produce the slope of a line that best fits a single set of data points [3]. Suppose if we have four data points (2,3), (1,5), (3,9), and (7,6), then it is desired to find a line $y = \Theta_1 + \Theta_2 x$ that fits best these four given points. In other words, we would like to find the numbers Θ_1 and Θ_2 that approximately solves the over determined linear system of four equations in two unknowns in some best sense.

$$\Theta_1 + 2\,\Theta_2 = 3$$
$$\Theta_1 + 1\,\Theta_2 = 5$$
$$\Theta_1 + 3\,\Theta_2 = 9$$
$$\Theta_1 + 7\,\Theta_2 = 6$$

The framework on machine learning concepts contains the following three components

1. Model class
2. Score function
3. Optimization/Search process.

The model class and score function are decided on the basis of rating different instances of selected model class. The aforementioned instances are provided by the optimization or search process. The optimization/search process using linear regression is an iterative algorithm that gradually descends down the gradient of the cost function, and generates parameters defining a new instance of our selected model class [6]. In the case of Linear regression the least square approach to solve this problem is trying to minimize the sum of squares of errors between X and Y, i.e. to find the minimum of the function

$$S(\Theta_1, \Theta_2) = [3-(\Theta_1 + 2\,\Theta_2)^2] + [5-(\Theta_1 + 1\,\Theta_2)^2] + [9-(\Theta_1 + 3\,\Theta_2)^2] + [3-(\Theta_1 + 2\,\Theta_2)^2]$$

The minimum of this function could be found by differentiating it partially with respect to Θ_1 and Θ_2 and setting them to zero. This results in the system of two equations in two unknowns where the value of theta is computed iteratively by minimizing the cost function $S(\Theta_1, \Theta_2)$.

Normal Equations, using linear algebra, could have our optimization/search process generate the optimal instance of our model class in one try. The Θ is computed by the following equation:

$$\Theta = (X^T X)^{-1} X^T Y$$

There are essentially two problems in linear algebra:

1. $Ax = y$
2. $Ax = \lambda x$

$Ax = y$ is a compact notation for describing a system of linear equations, which in general is either consistent (it has at least one solution) or not (it has no solution). There are several methods for solving systems of linear equation that are consistent, but for that aren't, we attempt to find the best approximate solution x; the one in which Euclidean length of the so called residual vector ($r = y-Ax$) is as close to zero as possible:

$$x = \min_x | |y-Ax| |$$

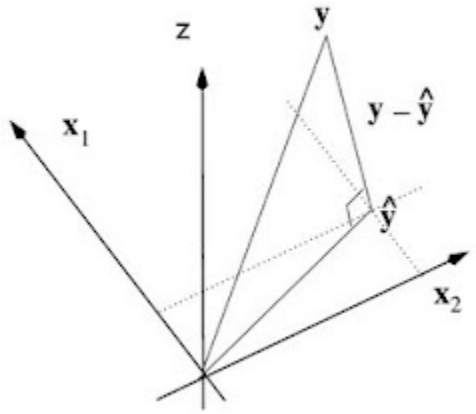

Fig 1. System of Linear Equations

Imagine we have a system of linear equations (as shown in Fig. 1) with parameter vector x=[x1; x2], such that the space spanned by Sp{x:[x1;x2;z=0], for all x1 and x2} is the range of our matrix A for all x, R(A).

Vector 'y' in Fig. (1), which is our observed target vector in $Ax = y$, is clearly not in R(A). Nevertheless, there is among all the vectors in R(A), one that is closest to y, which we denote to be $\hat{y}$. So, rather than finding the parameter vector x which satisfies $Ax = y$ (something we obviously cannot do), we will find the parameter vector x such that $Ax = \hat{y}$, which is the closest we can get to y.

Examining the diagram, our geometric intuition suggests that the vector $y-\hat{y}$ is orthogonal to every vector in the range of A, and since the columns of A form a spanning set for the range of A, Sp{A1,A2}=R(A), the dot product of each of those columns with $y-\hat{y}$ should equal zero:

$$A1^T(y- \hat{y}) = 0 \text{ ; and ;}$$
$$A2^T(y- \hat{y}) = 0$$

In matrix-vector notation:

$$A^T(y- \hat{y}) = 0$$

Now, since $\hat{y} = Aw$ for some w in our space Sp{x:[x1;x2;z=0], for all x1 and x2}, we have that:

$$A^T(y- Aw) = 0$$

Or, in more familiar form or the **Normal Equation**:

$$\mathbf{A^T Aw = A^T y}$$

And so we can solve for w in closed form:

$$w = (A^T A)^{-1} A^T y$$

The constraint of normal equations is that $(X^T X)^{-1}$be an invertible matrix [6]. This minimization problem has a unique solution, provided that the *n* columns of the matrix *X* are linearly independent, given by solving the **normal equations.** The main contributions of this paper are as follows:

1. We have analyzed the characteristics of Normal Equation. We found that it is better to implement it on GPU to improve performance.

2. We have computed the Θ value for the given X and Y values using Normal Equation Algorithm.

3. We have selected Normal Equation algorithm for parallel implementation on GPU and performance is compared on CPU.

4. We have analyzed that the Normal equation method is less costly as compared to Linear Regression technique because we don't need to choose learning rate 'α' in case of Normal Equations technique as we don't have to iterate and find the best fitting values as in case of linear regression method.

5. We have also analyzed that Normal Equations implementation using GPU can effectively be used for a greater number of features in the dataset.

II. RELATED WORK

GPUs that are available now-a-days provide high computation power at low costs and have been described as desktop supercomputers [5]. They are capable of performing massively parallel operations using CUDA threads. The GPUs have been used for many general purpose computations due to their low cost, high computing power, and high availability. The latest GPUs, for instance, can deliver close to 1 Tera Flops (TFLOPs) of compute power at a higher cost. The stages of were exploited for parallelism with the flow of execution handled serially using the pipeline in the earlier, GPGPU model. The GPUs expose a general, data-parallel programming model today in the form of CUDA. The recently adopted OpenCL standard will provide a common computing model to not only all GPUs, but also to other platforms like multi-core, many-core, and Cell/B.E. CUDA from NVIDIA

presents a heterogeneous programming model where the parallel hardware can be used in conjunction with the CPU [2],[4]. In conjunction with a CPU, it can be used as Bulk Synchronous Parallel (BSP) hardware with the CPU deciding the barrier for synchronization. GPU programming models are constrained in such a way that the compiler and runtime can reason about the application and extract the parallelism automatically. Examples of this include DirectX, CUDA, and Cg. Intel architecture is more general purpose than GPU and other coprocessor architecture. Unlike GPUs, Intel architectures have:

1) Inter-core communication through substantial, coherent cache hierarchies.

2) Efficient, low latency thread synchronizations across the entire processor array.

3) Narrower effective SIMD width.

At a high level, the goal is to define a constrained programming model that efficiently and portably targets highly parallel general purpose cores, such as Intel multi-core and Tera-scale systems. There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy. Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of Instruction and Data. Each of these dimensions can have only one of two possible states: Single or Multiple. GPU based processors can efficiently perform floating point operations and use parallelism at massive levels due to which they can be a suitable choice for processing large amounts of data.

III. Analysis of Normal Equation

In the paradigm of mathematics, the normal equation technique is an approach to fitting a mathematical model to data in cases where the idealized value provided by the model for any data point is expressed linearly in terms of the unknown parameters of the models.

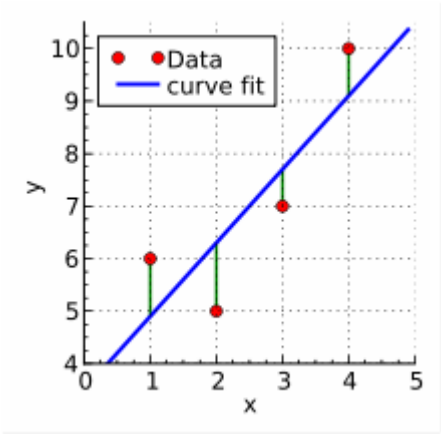

Fig. 2. A plot of the data points (in red), the line of best fit (in blue), and the residuals (in green).

Figure (2) shows the line which best fits the four points that are shown in the graph (red dots). This line is determined by using Normal Equation technique. The *x* and *y* co-ordinate values of points shown in the figure are used to construct the *'X' and 'Y'* matrices that are used to calculate Θ.

This concept can also be applied even if we have more number of variables. If there are *'m'* number of training sets and *'n'* number of features (multiple variables) involved with each sets, then the is computed as follows:

$$X=[\ x1^T\ x_2^T\ x_3^T\ x_4^T\ \ldots.\ X_n^T]^T$$

$$Y=[\ y^{(1)}\ y^{(2)}\ y^{(3)}\ y^{(4)}\ \ldots.\ y^{(m)}]^T$$

$$\Theta = (X^TX)^{-1}X^TY$$

For example, consider the following tables:

TABLE I : values of x_0, x_1 and x_2.

x_0	**Size (feet^2)** x_1	**Number of Bedrooms** x_2
1	2104	5
1	1416	3
1	1534	3
1	852	2

TABLE II : values of x_3, x_4 and y.

Number of Floors x_3	**Age of home (years)** x_4	**Price($1000)** y
1	45	460
2	40	232
2	30	315
1	36	178

Here

$x_0^T = [\ 1\ \ 1\ \ 1\ \ 1\ ]$

$x_1^T = [\ 2104\ \ 1416\ \ 1534\ \ 852\ ]$

$x_2^T = [\ 5\ \ 3\ \ 3\ \ 2\ ]$

$x_3^T = [\ 1\ \ 2\ \ 2\ \ 1\ ]$

$x_4^T = [\ 45\ \ 40\ \ 30\ \ 36\ ]$

Therefore, $X = [\ x_0^T\ \ x_1^T\ x_2^T\ x_3^T\ x_4^T\ ]$

and $Y = [\ 460 \quad 232 \quad 315 \quad 178\ ]^T$

The Θ is then calculated using the above mentioned formula.

The advantages of using Normal Equation technique over linear regression technique in calculating Θ are as follows:

1. We don't need to choose learning rate (α).
2. We don't need to perform too much iteration as it is done in the case of linear regression [1].

IV. Result and Discussions

A performance analysis of matrix product timing, matrix transpose and matrix inversion timing is done over both CPU as well as GPU and the results are shown in the following graphs.

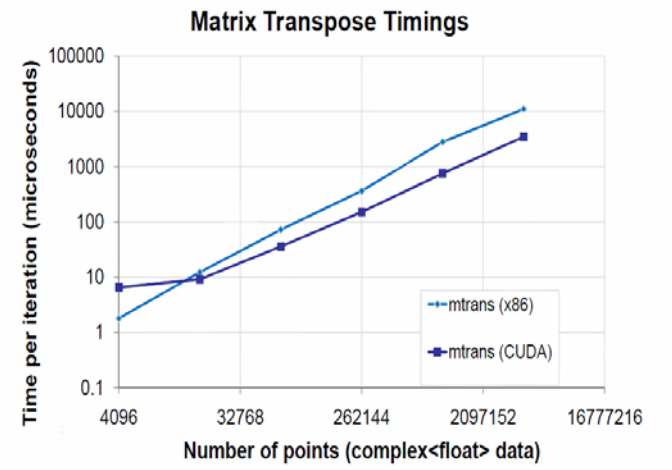

Fig. 3. The performance analysis of time taken by CPU vs time taken by GPU for performing Matrix Transpose operation.

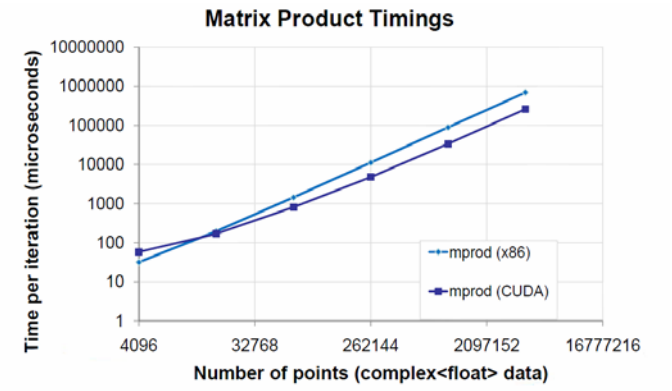

Fig. 4. The performance analysis of time taken by CPU vs time taken by GPU for performing Matrix Multiplication operation.

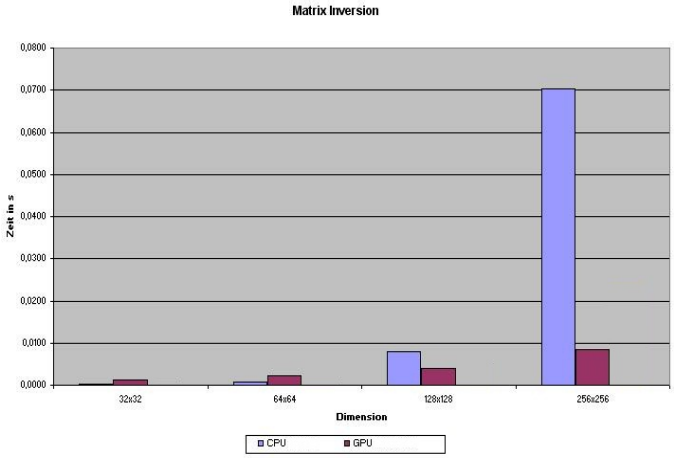

Fig. 5. The performance analysis of time taken by CPU vs time taken by GPU for performing Matrix Inverse operation (small dimension matrices).

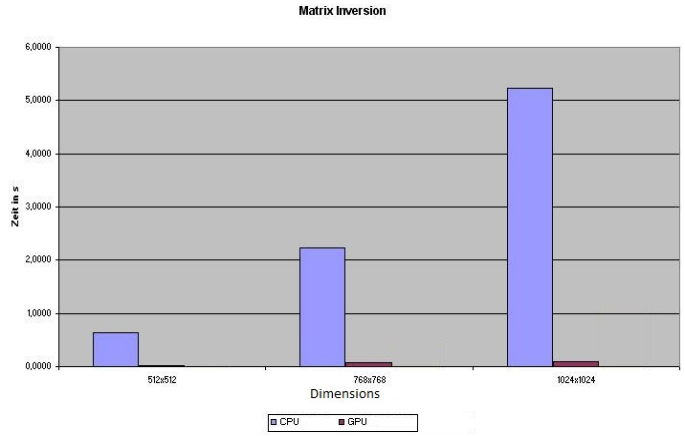

Fig. 6. The performance analysis of time taken by CPU vs time taken by GPU for performing Matrix Inverse operation (large dimension matrices).

Figure (3) shows the time taken by CPU in performing Matrix Transpose operation with respect to that of time taken by GPU in doing the same. The number of data points used is shown on x-axis and time taken per iteration is shown on y-axis. From the figure, we can infer that for small number of points, CPU takes less time as compared to GPU. But as the number of data point increases, the GPU takes less amount of time in performing matrix transpose as compared to CPU.

Figure (4) shows the time taken by CPU in performing Matrix Product operation with respect to that of time taken by GPU in doing the same. The number of data points used is shown on x-axis and time taken per iteration is shown on y-axis. It can be inferred from the graph that initially the performance of CPU is better as compared to that of GPU when there are less number of data points. As the number of data points increases, the GPU takes less time in comparison to that with CPU in performing the Matrix Multiplication operation.

Figure (5) shows the time taken by CPU in performing Matrix Product operation with respect to that of time taken by GPU in doing the same for smaller dimension matrices (dimensions of matrix varies from 32x32 to 256x256). The dimension of matrix used is shown on x-axis and time taken is shown on y-axis. It can be inferred from the graph that performance of GPU is better as compared to that of CPU.

Figure (6) shows the time taken by CPU in performing Matrix Product operation with respect to that of time taken by GPU in doing the same for larger dimension matrices (dimensions of matrix varies from 512x512 to 1024x1024). The dimension of matrix used is shown on x-axis and time taken is shown on y-axis. It can be inferred from the graph that performance of GPU is better as compared to that of CPU.

From Figure (3), (4), (5) and (6), it can be inferred that Matrix Multiplication, Matrix Transpose operation and Matrix Inverse operation take less time on GPU when compared with CPU. Therefore the Normal Equation algorithm will take less time in computing the result if it is implemented over GPU instead of CPU.

Further this algorithm can be improved if we use CUDA Basic Linear Algebra Subroutines (CUBLAS) while performing Matrix Transpose and Matrix Multiplication.

V. CONCLUSION

Data regression by Normal Equation on GPU using CUDA is implemented on NVIDIA Graphics Processing Unit integrated with Central Processing Unit. It is observed that the multithreading architecture and SIMD approach of CUDA helps for performance improvement in a great sense. There is tremendous difference in the results obtained on CPU and on GPU. So, CUDA programming is one of the best approaches to optimize the time for various algorithms which require huge amount of data, and is further suitable for operations which require floating point arithmetic. Therefore, GPUs can be used to run Machine Learning algorithms efficiently, with their capabilities to handle floating point arithmetic and big data as well.

REFERENCES

[1] Andrew, "Machine learning, Linear Regression with multiple variables".

[2] Brooks Moses, Don McCoy, Justin Voo, Stefan Seefeld CodeSourcery, Incorporation, "Comparison of Multicore Processors using Sourcery VSIPL++" .

[3] Jyoti B. Kulkarni, A. A. Sawant, Vandana S. Inamdar, "Database Processing by Linear Regression on GPU using CUDA", 2011, Proceedings of the ICSCCN 2011, IEEE International Conference, Pg 20-23.

[4] "Getting started with CUDA", www.nvidia.com.

[5] "Programming with CUDA", www.nvidia.com.

[6] "Linear Regression- Normal Equations", http://mechanistician.blogspot.in

[7] David B. Kirk, Wen-mei W. Hwu, " Programming Massively Parallel Processors".

AUTHORS PROFILE

Vaibhav Mohan is currently pursuing his undergraduate studies from Vellore Institute of Technology and is currently in final year of his bachelor of Technology course. His area of interest is algorithms, code optimizations, soft computing, parallel computing and parallel algorithms.

Mayank Gupta is a final year student at Vellore Institute of Technology in Bachelor of technology program. His area of interest is algorithms, code optimizations, soft computing, and parallel algorithms.