Parallelization of Standard Machine Learning Optimization Techniques using CUDA

Report submitted in fulfillment of the requirements
for the Exploratory Project of

Second Year B.Tech.

by

Naveen Punjabi [16075061]
Prakhar Sinha [16075062]

Under the guidance of

Prof K.K. Shukla

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY (BHU) VARANASI
Varanasi 221005, India

May 2018

Dedicated to

Our parents and teachers

Declaration

We certify that

1. The work contained in this report is original and has been done by ourselves and the general supervision of our supervisor.

2. The work has not been submitted for any project.

3. Whenever we have used materials (data, theoretical analysis, results) from other sources, we have given due credit to them by citing them in the text of the report and giving their details in the references.

4. Whenever we have quoted written materials from other sources, we have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Place: IIT (BHU) Varanasi
Date: May 2, 2018

Naveen Punjabi
B.Tech. Part II

Prakhar Sinha
B.Tech. Part II

Department of Computer Science and Engineering,
Indian Institute of Technology (BHU) Varanasi,
Varanasi, INDIA 221005.

Certificate

*This is to certify that the work contained in this report entitled "**Parallelization of standard machine learning optimization techniques using CUDA**" being submitted by **Naveen Punjabi** (**Roll No. 16075061**) and **Prakhar Sinha** (**Roll No. 16075062**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology (BHU) Varanasi, is a bona fide work of our supervision.*

Place: IIT (BHU) Varanasi
Date: May 2, 2018

Prof K.K. Shukla
Department of Computer Science and Engineering,
Indian Institute of Technology (BHU) Varanasi,
Varanasi, INDIA 221005.

Acknowledgments

We have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. We would like to extend our sincere gratitude to all of them.

We are highly indebted to Prof K.K. Shukla for his guidance and constant supervision as well as for providing necessary information regarding the project and also for his support throughout the duration of the project. We would like to express our gratitude towards our parents for their kind co-operation and encouragement which help us in completion of this project. Our thanks and appreciations also go to our colleagues in developing the project and people who have willingly helped us out with their abilities. We are also thankful to Nvidia for its helpful documentation on CUDA.

Place: IIT (BHU) Varanasi **Naveen Punjabi**

Date: May 2, 2018 **Prakhar Sinha**

Abstract

With increasing demand of machine learning tools in the recent times, it has become necessary to produce efficient and accurate machine learning models. The large amount of data available in today's world has made this goal quite achievable. Therefore, parallel machine learning has become an increasingly pressing problem.

In this project, the possibility of parallelizing the training phase of machine learning is explored. Parallelization is implemented using Nvidia's CUDA API. Linear Regression model is trained parallely using batch gradient and stochastic gradient descents. In batch gradient descent, the computational part involving matrix multiplication is parallelized. The dataset in divided into several batches and each batch is executed parallely for the stochastic gradient descent. The model is trained for the "Million Songs Dataset (UCI datasets)". The speed-up and the CPU and GPU utilizations are recorded and a comparative study is provided for the serial and parallel implementations.

Contents

List of Figures

List of Tables

List of Symbols

Symbol	Description
α	Learning Rate
γ	Step size in descent
∇	Gradient
θ	Parameters

Chapter 1

CUDA : Introduction

Compute Unified Device Architecture, popularly known as CUDA, is a parallel computing platform and programming model developed by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the Graphics Processing Unit (GPU).

CUDA was developed with several design goals in mind. It provides a small set of extensions to standard programming languages, like C, C++, Python that enable a straightforward implementation of parallel algorithms. It is capable of supporting heterogeneous computation where applications use both the CPU and GPU. Serial portions of applications are run on the CPU, and parallel portions are offloaded to the GPU. CUDA-capable GPUs have hundreds of cores that can collectively run thousands of computing threads concurrently.

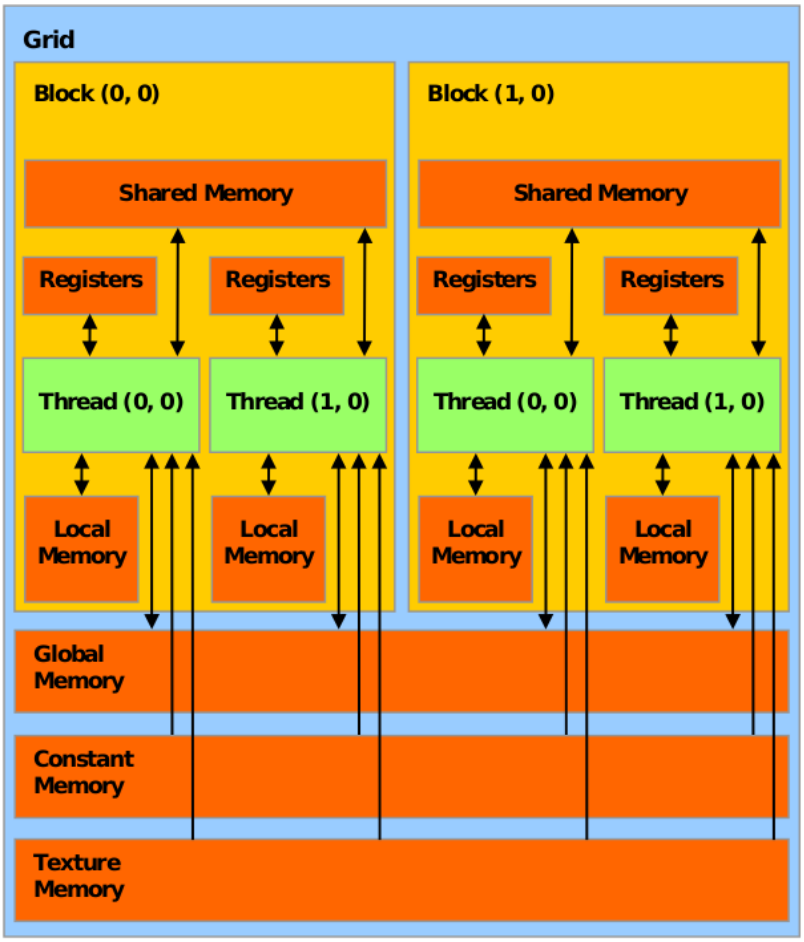
1.1 Scalable Programming Model

At its core, CUDA has three key abstractions a hierarchy of thread groups, shared memories, and barrier synchronization that are simply exposed to the programmer as a minimal set of language extensions.

These abstractions provide fine-grained data parallelism and thread parallelism,

nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors, and only the runtime system needs to know the physical multiprocessor count.

Figure 1.1 Automatic Scalability



1.2 Multi-threaded Co-processor

When programmed through CUDA, the GPU is viewed as a compute device capable of executing a very high number of threads in parallel. It operates as a co-processor to the main CPU, or host: In other words, data-parallel, compute-intensive portions of applications running on the host are off-loaded onto the device.

More precisely, a portion of an application that is executed many times, but independently on different data, can be isolated into a function that is executed on the device as many different threads. To that effect, such a function is compiled to the instruction set of the device and the resulting program, called a kernel, is downloaded to the device.

1.3 Thread Batching

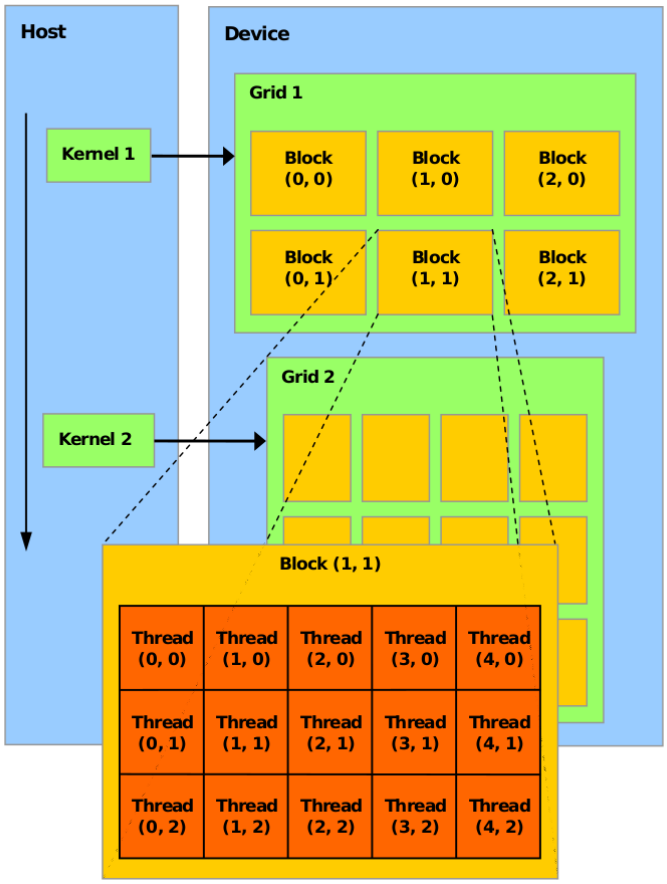
The batch of threads that executes a kernel is organized as a grid of thread blocks as described in further sections.

1.3.1 Thread Block

A thread block is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel, where threads in a block are suspended until they all reach the synchronization point.

Each thread is identified by its thread ID, which is the thread number within the block. To help with complex addressing based on the thread ID, an application can also specify a block as a two-l or three-dimensional array of arbitrary size and identify each thread using a 2- or 3-component index instead. For a two-dimensional block of size (D_x, D_y), the thread ID of a thread of index (x, y) is $(x + yD_x)$ and

Figure 1.2 Thread Batching



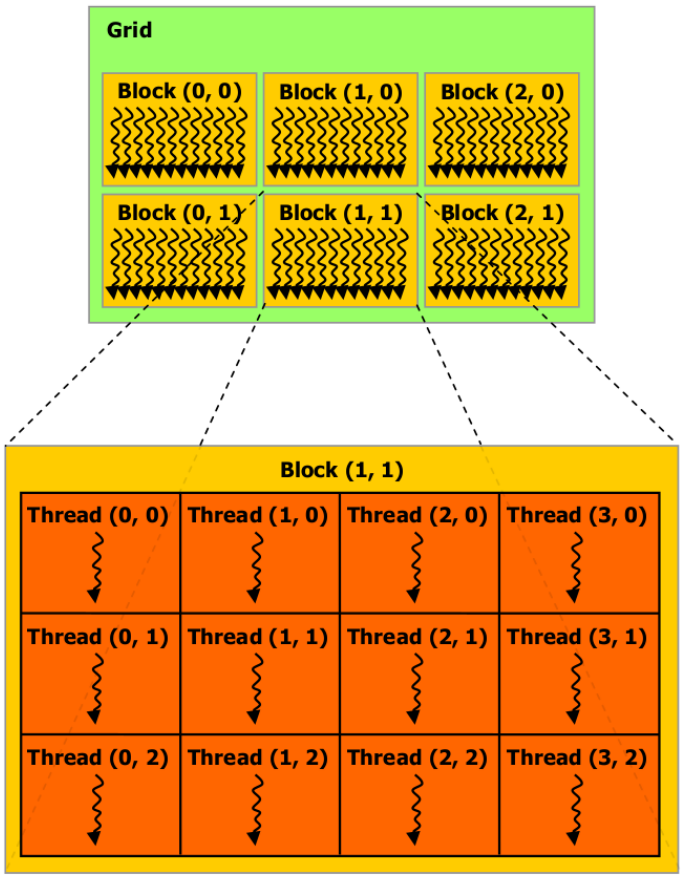
for a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is $(x + yD_x + zD_xD_y)$.

1.3.2 Grid of Thread Blocks

There is a limited maximum number of threads that a block can contain. However, blocks of same dimensionality and size that execute the same kernel can be batched together into a grid of blocks, so that the total number of threads that can be launched in a single kernel invocation is much larger. This comes at the expense of reduced thread cooperation, because threads in different thread blocks from the same grid cannot communicate and synchronize with each other. A device may run all the blocks of a grid sequentially if it has very few parallel capabilities, or in parallel if it

has a lot of parallel capabilities, or usually a combination of both.

Figure 1.3 Grid of Thread Blocks



Each block is identified by its block ID, which is the block number within the grid. To help with complex addressing based on the block ID, an application can also specify a grid as a two-dimensional array of arbitrary size and identify each block using a 2-component index instead. For a two-dimensional block of size (D_x, D_y), the block ID of a block of index (x, y) is $(x + yD_x)$.

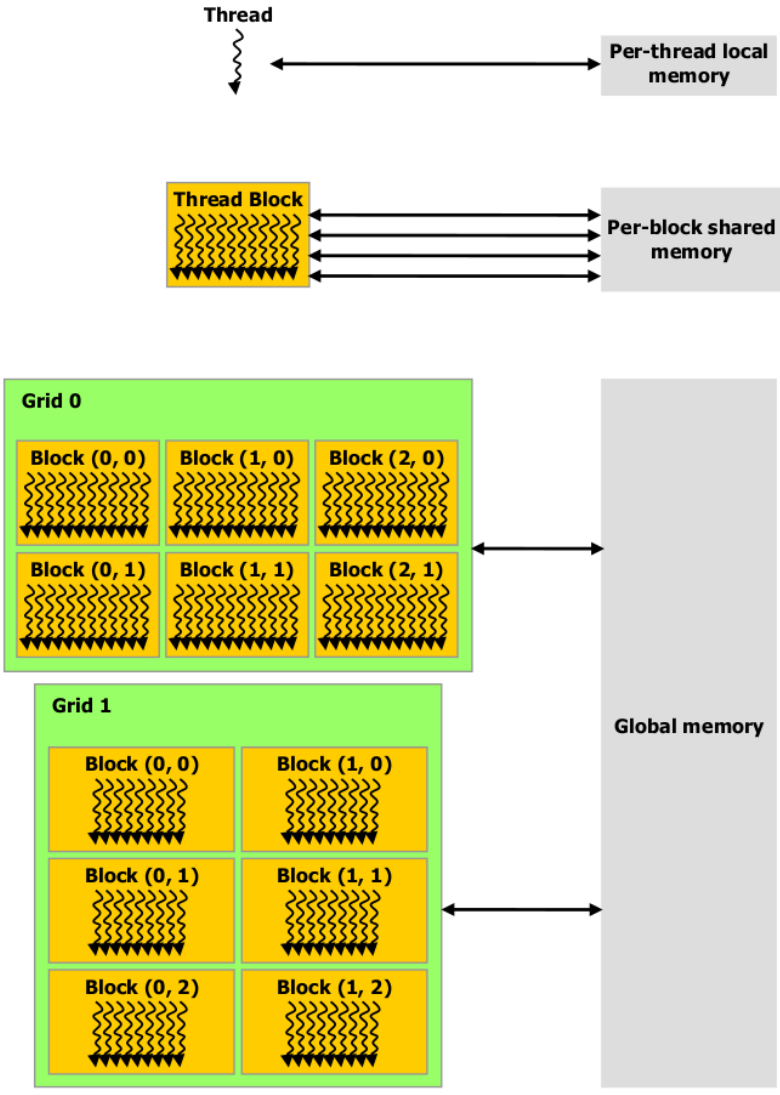
1.4 Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have

access to the same global memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces.

The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats. The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

Figure 1.4 Memory Hierarchy



Chapter 2

Introduction to Numba

Numba supports CUDA GPU programming by directly compiling a restricted subset of Python code into CUDA kernels and device functions following the CUDA execution model.

One feature that significantly simplifies writing GPU kernels is that Numba makes it appear that the kernel has direct access to NumPy arrays. NumPy arrays that are supplied as arguments to the kernel are transferred between the CPU and the GPU automatically (although this can also be an issue).

Numba does not yet implement the full CUDA API, so some features are not available. However the features that are provided are enough to begin experimenting with writing GPU enable kernels. CUDA support in Numba is being actively developed, so eventually most of the features should be available.

2.1 Writing CUDA Kernels

CUDA has an execution model unlike the traditional sequential model used for programming CPUs. In CUDA, the code will be executed by multiple threads at once (often hundreds or thousands) and it will be modeled by defining a thread hierarchy of grid, blocks, and threads. Numba also exposes three kinds of GPU memory:

- global device memory
- shared memory
- local memory

2.1.1 Kernel Declaration

A kernel function is a GPU function that is meant to be called from CPU code. It has two fundamental characteristics:

- Kernels cannot explicitly return a value; all result data must be written to an array passed to the function (if computing a scalar, we will probably pass a one-element array).
- Kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block (note that while a kernel is compiled once, it can be called multiple times with different block sizes or grid sizes).

Listing 2.1 Kernel Declaration

```
from numba import cuda


@cuda.jit
def cuda_kernel(input_array):
    """
    Code for kernel.
    """
```

2.1.2 Kernel Invocation

A kernel is typically launched in the following way:

Listing 2.2 Kernel Invocation

```
import numpy
import math

# Create the data array
data = numpy.ones(256)
# Set the number of threads in a block
threadsperblock = 32
# Calculate the number of thread blocks in the grid
blockspergrid = math.ceil((data.size)/threadsperblock)

# Now start the kernel
cuda_kernel[blockspergrid, threadsperblock](data)
```

There are two main steps:

1. Instantiate the kernel by specifying a number of blocks per grid and the number of threads per block. The product of the two will give the total number of threads launched.

2. Running the kernel, by passing it the input array (and any separate output arrays if necessary). By default, running a kernel is synchronous: the function returns when the kernel has finished executing and the data is synchronized back.

2.2 Thread Positioning

When running a kernel, the kernel functions code is executed by every thread once. It therefore has to know which thread it is in, in order to know which array element(s) it is responsible for. More complex algorithms may define more complex responsibilities, but the underlying principle is the same.

To help deal with multi-dimensional arrays, CUDA allows us to specify multi-dimensional blocks and grids. In the example above, we could make blockspergrid and threadsperblock tuples of one, two or three integers. Compared to 1-dimensional declarations of equivalent sizes, this doesnt change anything to the efficiency or behaviour of generated code, but can help us write our algorithms in a more natural way.

One way is for the thread to determines its position in the grid and block and manually compute the corresponding array position:

Listing 2.3 Thread Positioning

```
@cuda.jit
def cuda_kernel(input_array):
    # Thread id in a 1D block
    tx = cuda.threadIdx.x
    # Block id in a 1D grid
    ty = cuda.blockIdx.x
    # Block width, i.e. number of threads per block
    bw = cuda.blockDim.x
    # Compute flattened index inside the array
    pos = tx + ty * bw
    if pos < io_array.size: # Check array boundaries
        input_array[pos] *= 2 # do the computation
```

The following special objects are provided by the CUDA backend for the sole purpose of knowing the geometry of the thread hierarchy and the position of the current thread within that geometry:

- numba.cuda.threadIdx - The thread indices in the current thread block. For 1-dimensional blocks, the index (given by the x attribute) is an integer spanning the range from 0 to numba.cuda.blockDim - 1. A similar rule exists for each

dimension when more than one dimension is used.

- numba.cuda.blockDim - The shape of the block of threads, as declared when instantiating the kernel. This value is the same for all threads in a given kernel, even if they belong to different blocks (i.e. each block is full).

- numba.cuda.blockIdx - The block indices in the grid of threads launched a kernel. For a 1-dimensional grid, the index (given by the x attribute) is an integer spanning the range from 0 to numba.cuda.gridDim - 1. A similar rule exists for each dimension when more than one dimension is used.

- numba.cuda.gridDim - The shape of the grid of blocks, i.e. the total number of blocks launched by this kernel invocation, as declared when instantiating the kernel.

These objects can be 1-, 2- or 3-dimensional, depending on how the kernel was invoked. To access the value at each dimension, use the x, y and z attributes of these objects, respectively.

2.3 Absolute Positions

Simple algorithms will tend to always use thread indices in the same way as shown in the example above. Numba provides additional facilities to automate such calculations:

- numba.cuda.grid(ndim) - Return the absolute position of the current thread in the entire grid of blocks. ndim should correspond to the number of dimensions declared when instantiating the kernel. If ndim is 1, a single integer is returned. If ndim is 2 or 3, a tuple of the given number of integers is returned.

- numba.cuda.gridsize(ndim) - Return the absolute size (or shape) in threads of the entire grid of blocks. ndim has the same meaning as in grid() above.

Using these functions, the our example can become:

Listing 2.4 Absolute Positions

```
@cuda.jit
def cuda_kernel(input_array):
    pos = cuda.grid(1)
    if pos < input_array.size:
        input_array[pos] *= 2 # do the computation
```

2.4 Memory Management

Numba has been automatically transferring the NumPy arrays to the device when us invoke the kernel. However, it can only do so conservatively by always transferring the device memory back to the host when a kernel finishes. To avoid the unnecessary transfer for read-only arrays, it is possible to manually control the transfer.

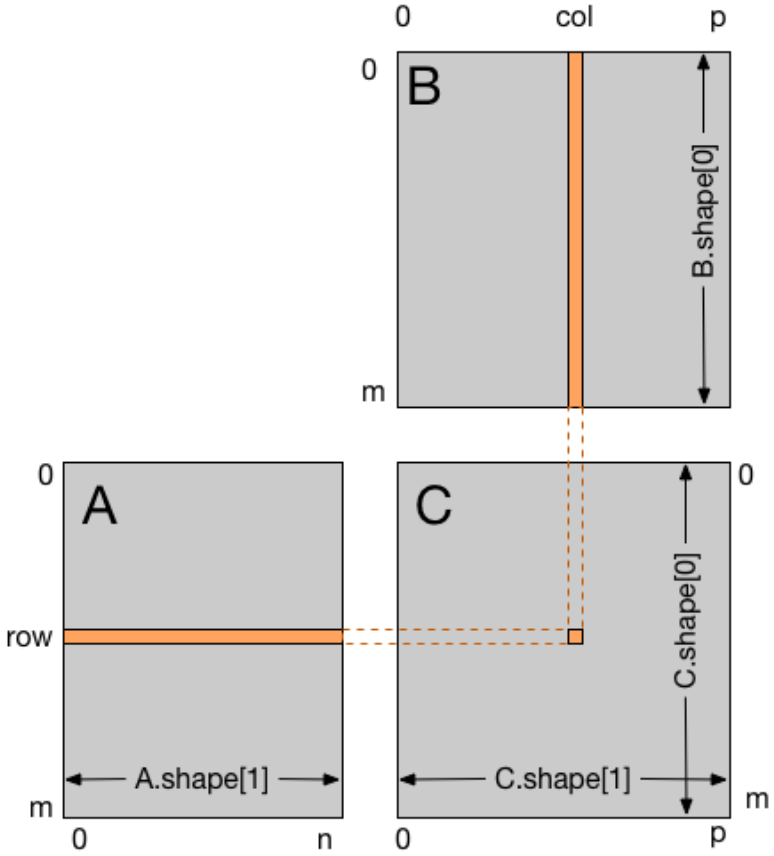
Listing 2.5 Device Memory Allocation

```
# Allocate an empty device ndarray
device_array = cuda.device_array(shape)
# Allocate and transfer a NumPy ndarray to the device
device_array = cuda.to_device(array)
"""
Start the kernel to get result
"""
# Copy the Device array to Host
host_array = result.copy_to_host()
```

2.5 Matrix Multiplication

A straightforward implementation of matrix multiplication for matrices A and B is where each thread reads one row of A and one column of B and computes the corresponding element of C. For input arrays where $A.shape == (m, n)$ and $B.shape == (n, p)$ then the result shape will be $C.shape = (m, p)$. The host code must create and

Figure 2.1 Matrix Multiplication



initiliaze the A and B arrays, then move them to the device. Next, it must allocate space on the device for the result array. Once the kernel has completed, the result array must be copied back to the host so that it can be displayed.

2.5.1 Shared Memory and Thread Synchronization

A limited amount of shared memory can be allocated on the device to speed up access to data, when necessary. That memory will be shared (i.e. both readable and writable) amongst all threads belonging to a given block and has faster access times than regular device memory. The function to create a shared memory array is:

Listing 2.6 Shared Memory Allocation

```
shared_array = cuda.shared.array(shape,type)
```

shape is either an integer or a tuple of integers representing the arrays dimensions. *type* is a Numba type of the elements needing to be stored in the array.

Because the shared memory is a limited resource, it is often necessary to preload a small block at a time from the input arrays. All the threads then need to wait until everyone has finished preloading before doing the computation on the shared memory.

Synchronization is then required again after the computation to ensure all threads have finished with the data in shared memory before overwriting it in the next loop iteration. The function to synchronized threads is:

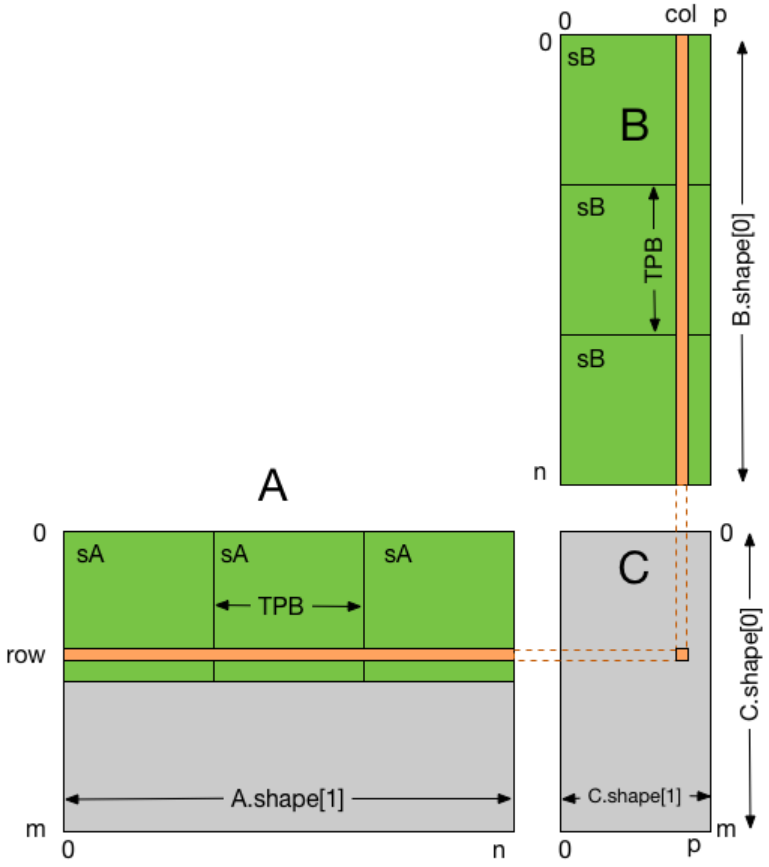
Listing 2.7 Shared Memory Allocation

```
cuda.syncthreads()
```

2.5.2 Improved Matrix Multiplication

The following example shows how shared memory can be used when performing matrix multiplication. In this example, each thread block is responsible for computing a square sub-matrix of C and each thread for computing an element of the sub-matrix. The sub-matrix is equal to the product of a square sub-matrix of A (sA) and a square sub-matrix of B (sB). In order to fit into the device resources, the two input matrices are divided into as many square sub-matrices of dimension TPB as necessary, and

Figure 2.2 Improved Matrix Multiplication



the result computed as the sum of the products of these square sub-matrices.

Each product is performed by first loading sA and sB from global memory to shared memory, with one thread loading each element of each sub-matrix. Once sA and sB have been loaded, each thread accumulates the result into a register (tmp). Once all the products have been calculated, the results are written to the matrix C in global memory.

By blocking the computation this way, we can reduce the number of global memory accesses since A is now only read B.shape[1] / TPB times and B is read A.shape[0] / TPB times.

Chapter 3

Gradient Descent

3.1 Overview

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point. In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to the coefficients in linear regression or weights in neural network. Gradient descent is based on the observation that if the multi-variable function $F(x)$ is defined and differentiable in a neighborhood of a point a, then $F(x)$ decreases fastest if one goes from a in the direction of the negative gradient of F at a, $-\nabla F(a)$. It follows that, if

$$a_{n+1} = a_n - \gamma \nabla F(a_n)$$

for γ small enough, then $F(a_n) \geq F(a_{n+1})$. In other words, the term $\gamma \nabla F(a)$ is subtracted from a because we want to move against the gradient, toward the minimum. With this observation in mind, one starts with a guess x_0 for a local minimum of F,

and considers the sequence $x_0, x_1, x_2, ...$ such that

$$x_{n+1} = x_n - \gamma \nabla F(x_n), n \geq 0$$

We have

$$F(x_0) \geq F(x_1) \geq F(x_2) \geq ...,$$

so hopefully the sequence (x_n) converges to the desired local minimum. In machine learning, gradient descent is used to fit the model and increase its accuracy. In this report, we are present the gradient descent method for linear regression.

3.2 Cost Function

A cost function is a wrapper around our model function that tells us "how good" our model is at making predictions for a given set of parameters. The cost function has its own curve and its own gradients. The slope of this curve tells us how to change our parameters to make the model more accurate. We use the model to make predictions. We use the cost function to update our parameters. Our cost function can take a variety of forms as there are many different cost functions available. The cost function that we will use here (for linear regression) is the "Squared Mean Loss"

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

where $J(\theta)$ represents the cost function, m is the number of training instances, $h_\theta(x^{(i)})$ is the predicted label for the i^{th} instance and $y^{(i)}$ is the actual label for the i^{th} instance.

3.3 Updating parameters

Supposing the hypothesis is

$$h_\theta(x_1^{(i)}, x_2^{(i)}, .., x_n^{(i)}) = \theta_0 + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + ... + \theta_n x_n^{(i)}$$

where $\theta_0, \theta_1, .., \theta_n$ are the parameters and $x_1, x_2, .., x_n$ are the features. The gradient descent update equation is given by

$$\theta_j := \theta_j - \alpha \frac{\partial J}{\partial \theta_j} \qquad for j = 0, 1, 2..., n$$

where α is the learning rate.

Chapter 4

Batch Gradient Descent

4.1 Overview

In Batch Gradient Descent, at every iteration to update the parameters $\theta_0, \theta_1, \ldots, \theta_n$, the value of $\frac{\partial J}{\partial \theta}$ is calculated by taking into consideration all the instances in the training dataset. For every update, the algorithm traverses through all the instances in order to calculate the gradient.

Considering the vector representation

$$\begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} := \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{bmatrix}$$

Now, we know the fact that if $f = f(x, y)$ is a function, then

$$\frac{\partial f}{\partial \theta} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial \theta} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial \theta}$$

In a similar way, if we view the loss function as $J = J(h_\theta(x^{(1)}), h_\theta(x^{(2)}), \ldots, h_\theta(x^{(m)}))$,

then

$$\frac{\partial J}{\partial \theta_j} = \sum_{i=1}^{m} \frac{\partial J}{\partial h_\theta(x^{(i)})} \frac{\partial h_\theta(x^{(i)})}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \frac{\partial h_\theta(x^{(i)})}{\partial \theta_j}$$

Now, we have $h_\theta(x^{(i)}) = \sum_{i=0}^{n} x_j^{(i)} \theta_j$; using the above equation we get

$$\frac{\partial J}{\partial \theta_j} = \sum_{i=1}^{m} \frac{1}{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} = \frac{1}{m} \begin{bmatrix} x_j^{(1)} x_j^{(2)} \dots x_j^{(m)} \end{bmatrix} \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}$$

Thus,

$$\begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{bmatrix} := \begin{bmatrix} x_0^{(1)} & x_0^{(2)} & x_0^{(3)} & \dots & x_0^{(m)} \\ x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & \dots & x_1^{(m)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_n^{(1)} & x_n^{(2)} & x_n^{(3)} & \dots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}$$

which is equivalent to

$$\frac{\partial J}{\partial \theta} = \frac{1}{m} x^T (h_\theta(x) - y)$$

So finally, the parameter update equation becomes

$$\theta := \theta - \alpha \frac{1}{m} x^T (h_\theta(x) - y)$$

Algorithm 1 Algorithm for serial implementation of BGD

1: **procedure** BatchGradient($\theta, X, Y, m, n, \alpha$):
2: *Repeat until convergence*
3: $reduction \leftarrow h_\theta(X) - Y$
4: $gradient \leftarrow X^T * reduction$
5: $\theta \leftarrow \theta - \frac{1}{m} * \alpha * gradient$

4.2 What to parallelize ?

Our main aim is to speed up the training of the machine learning model (here, linear regression). The approach that we have used to speed up batch gradient descent is:

4.2.1 Parallelizing matrix multiplication

Updating the parameters while training the model requires calculating the product of the matrices x^T and $(h_\theta(x) - y)$. So, we can parallelize this multiplication by running the threads parallely on the GPU cores where each thread calculates the value at a unique index in the resulting matrix.

4.3 Parallelized BGD

For parallelizing batch gradient descent, we implement a CUDA kernel which performs parallel matrix multiplication at two steps.

4.3.1 Calculating $h_\theta(X)$

We calculate $h_\theta(X)$ as

$$h_\theta(X) = X \times \theta$$

X and θ are as defined in Algorithm 1. The matrix X has dimensions $m \times n$ where m represents the number of training instances and n represents the number of features. The matrix θ has dimensions $n \times 1$. Therefore, the result matrix $h_\theta(X)$ has dimensions $m \times 1$.

4.3.2 Calculating gradient

X^T and *reduction* are as defined in Algorithm 1. The matrix x^T has dimensions $n \times m$. The matrix *reduction* has dimensions $m \times 1$. There the resulting matrix that we get is of dimensions $n \times 1$.

Chapter 5

Stochastic Gradient Descent

5.1 Overview

Stochastic gradient descent is a stochastic approximation of the gradient descent optimization and iterative method for minimizing a cost function that is written as a sum of differentiable functions. In stochastic gradient descent, the true gradient of $J(\theta)$ is approximated by a gradient at a single example:

$$\theta := \theta - \alpha \nabla J(\theta)^{(i)}$$

We have considered

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

$$= \frac{1}{m} \sum_{i=1}^{m} J(\theta)^{(i)}$$

$$where \qquad J(\theta)^{(i)} = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Therefore, to update the parameter θ, any one sample is randomly chosen from the training dataset and θ is then updated by taking into consideration the gradient of the cost function for that particular sample rather than the whole training dataset. This

is mostly used in the case where the training dataset is quite large (usually greater than 10^7). This speeds up then training phase of the machine learning model as the CPU then does not have to compute the gradient of each individual instance at each update step.

Although this method may cause the steps to be taken in directions which do not lead to the minimum of the objective function, but ultimately it converges due to a large number of instances present in the training dataset.

5.2 Serial Implementation

Given a dataset having m number of samples and n number of features. In each iteration, the gradient of the cost function for a single sample is calculated. θ is updated based on the gradient of this sample. This process is repeated for the whole dataset. The serial implementation for Stochastic Gradient Descent is demonstrated in Algorithm 2.

Algorithm 2 Algorithm for serial implementation of SGD

1: **procedure** StochasticGradient($\theta, X, Y, m, n, \alpha$):
2: *Repeat until convergence* :
3: Draw j from 1,2,...,m randomly
4: $reduction \leftarrow h_\theta(x^{(j)}) - y^{(j)}$
5: $gradient = x^{(j)} * reduction$
6: $\theta \leftarrow \theta - \alpha * gradient$

5.3 What to parallelize?

Since the training dataset is quite large, so one way of parallelizing the algorithm would be to divide the dataset into smaller k batches of equal size where each batch contains randomly chosen instances from the training dataset. Each batch is used to compute an independent set of parameters θ. Therefore, every instance from each batch is run parallely on the GPU cores. After convergence of parameters for each

of the k batches, the final parameters is calculated by taking the mean over all the parameters calculated by the individual batches. The parallelized implementation for Stochastic Gradient Descent is demonstrated in Algorithm 3.

Algorithm 3 Algorithm for parallel implementation of SGD

procedure ParallelStochasticGradient($\theta, X, Y, m, n, \alpha$):
    Define $T = m/k$
    Randomly partition the samples giving T samples to each batch
    for *all* $i = 1$ *to* k *parallelly* **do**
        Randomly shuffle the data in batch i
        Initialize $\theta_{(i,0)} = 0$
        for *all* $t = 1$ *to* T **do**
            Get the t^{th} sample in the i^{th} batch : $x^{(i,t)}$
            $reduction \leftarrow h_\theta(x^{(i,t)}) - y^{(i,t)}$
            $gradient = x^{(i,t)} * reduction$
            $\theta_{(i,t)} \leftarrow \theta_{(i,t-1)} - \alpha * gradient$
    Aggregate from all batches $\theta = \frac{1}{k} \sum_{i=1}^{k} \theta_{(i,t)}$

Chapter 6

Working Model

6.1 GPU Specifications

The machine learning model is trained on CUDA-enabled GPU - NVIDIA GeForce 940x. Its specifications are as follows:

Graphics Processor:	GeForce 940MX
CUDA Cores:	384
Total Memory:	2048 MB
Memory Interface:	64-bit
Bus Type:	PCI Express x4 Gen3

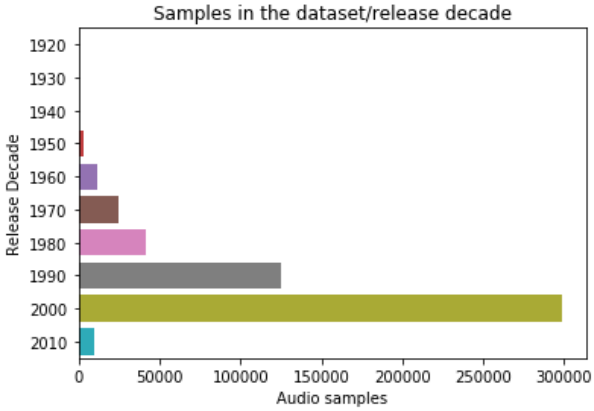
Table 6.1 GPU Specifications

6.2 Dataset Used

- **Name of the Dataset** : Million Songs Dataset (from Standard UCI Datasets)
- **Description** : Prediction of the release year of a song from audio features. Songs are mostly western, commercial tracks ranging from 1922 to 2011, with a peak in the year 2000s.

- **Dataset Information** : 90 attributes, 12 = timbre average, 78 = timbre covariance

Figure 6.1 Million Songs Dataset



The first value is the year (target), ranging from 1922 to 2011. Features extracted from the 'timbre' features from The Echo Nest API. We take the average and covariance over all 'segments', each segment being described by a 12-dimensional timbre vector.

6.3 Implementation

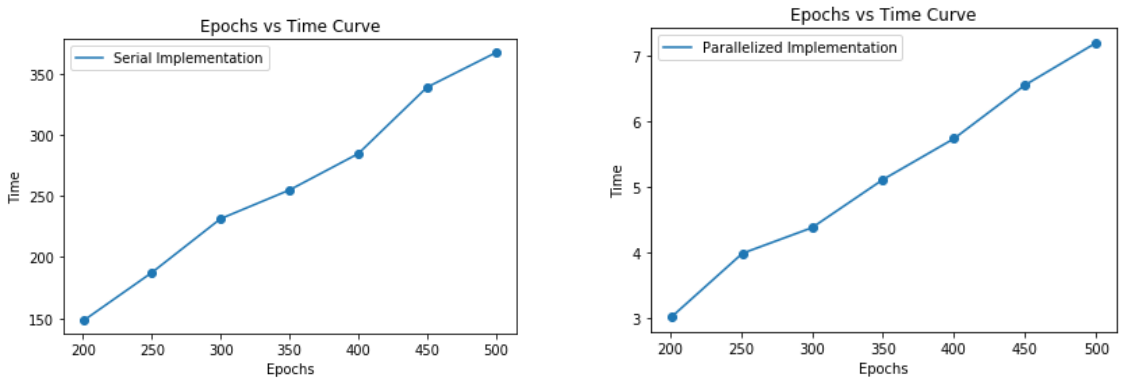
The model is trained for linear regression using Batch Gradient Descent and Stochastic Gradent Descent respectively. The results obtained are discussed in the following sections.

6.3.1 Batch Gradient Descent

- Number of samples in training set : 5000
- Number of samples in validation set : 1000
- Number of samples in test set : 1000
- Learning rate α : 0.001

Serial Implementation

Figure 6.2 Batch Gradient Descent

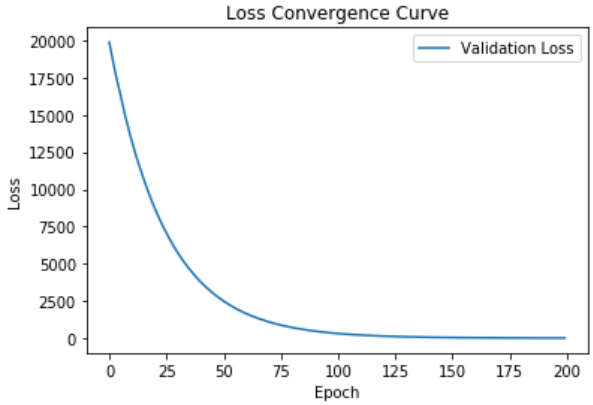


Final Validation Loss:	6.1783
Test Loss:	3.5512

Table 6.2 Serial BGD Results

Parallelized Implementation

Figure 6.3 Parallel BGD Loss Convergence



Final Validation Loss:	3.1783
Test Loss:	4.5512

Table 6.3 Parallel BGD Results

6.3.2 Stochastic Gradient Descent

- Number of samples in training set : 500000
- Number of samples in validation set : 100000
- Number of samples in test set : 1000000
- Learning rate α : 0.001

Serial Implementation

For a dataset of 50000 samples, serial Implementation of Stochastic Gradient Descent took about 23000 for 10 epochs.

Final Validation Loss:	17.1321
Test Loss:	16.1093

Table 6.4 Serial SGD Results

Parallelized Implementation

Figure 6.4 Stochastic Gradient Descent

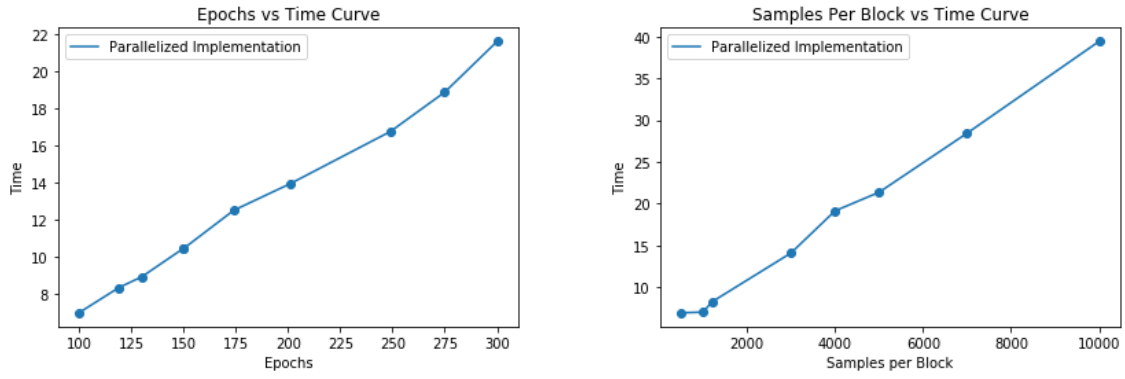
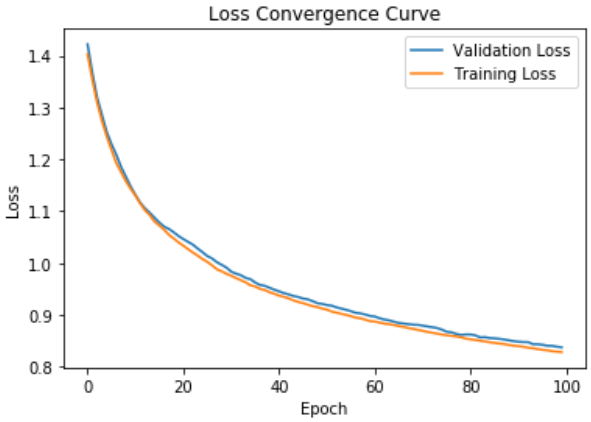


Figure 6.5 Parallel SGD Loss Convergence



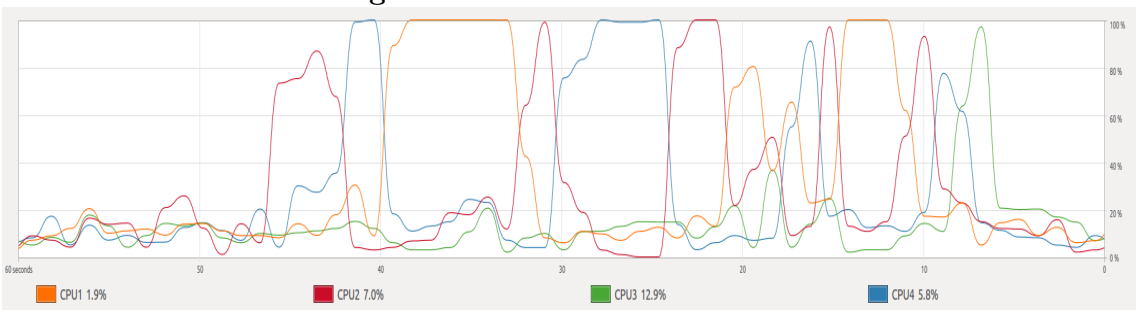
Final Validation Loss:	3.1783
Test Loss:	4.5512

Table 6.5 Serial SGD Results

6.3.3 CPU Utilization Graphs

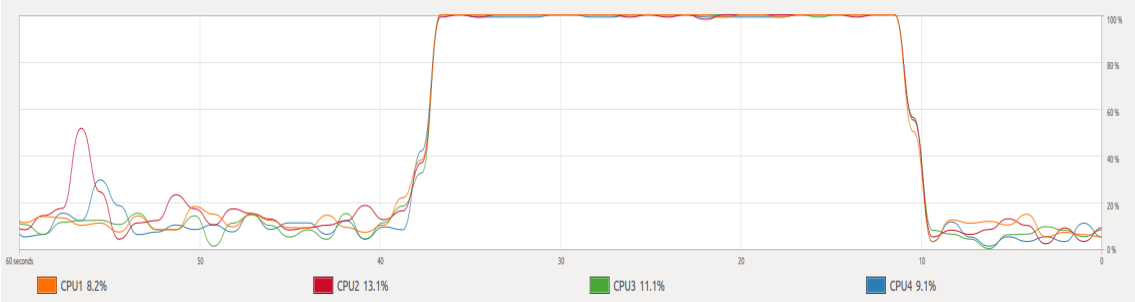
Parallelized Batch Gradient Descent

Figure 6.6 CPU Utilization BGD



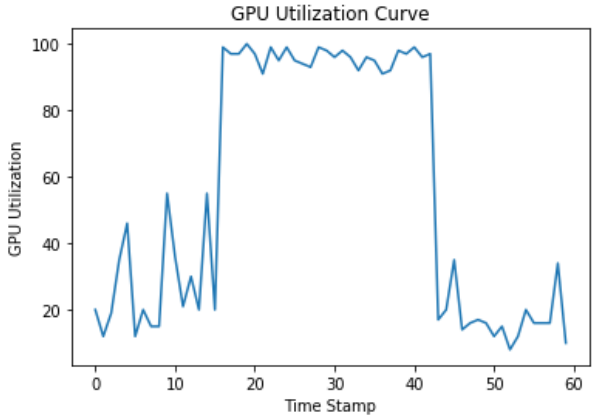
Parallelized Stochastic Gradient Descent

Figure 6.7 CPU Utilization SGD



6.3.4 GPU Utilization Graph

Figure 6.8 GPU Utilization



Chapter 7

Conclusions and Discussion

In this project, we have explored the parallelization techniques that can be used to speed up the optimization algorithms that are used for training a machine learning model. The data-parallel stochastic gradient algorithm implemented enjoys a number of key properties that makes it highly suitable for parallel, large-scale machine learning. A comparative study betwen the parallelized and non-parallelized versions has also been presented in this report. Our experiments on a large scale real-world dataset show that the parallelization reduces the wall-clock time needed to obtain an accurately trained model.

Future Directions

This report gives rise to a number of important problems that need to be looked into further :

- During parallelization, device to host and host to device transfers overheads are quite high. More than 70% of the running time is utilized in I/O. Algorithms may be implemented in other ways so as to reduce the I/O involved.
- CUDA provides inter-thread synchronization within a block but it does not pro-

vide inter-block synchronization. This leads to limitations while using CUDA. Synchronization among the blocks can be implemented further, which may increase the overhead on the part of programmer but achieve appreciable results in terms of performance.

Bibliography

[1] Martin A. Zinkevich, Markus Weimer, Alex Smola, and Lihong Li. *Parallelized Stochastic Gradient Descent.*

[2] Nvidia CUDA Guide

`https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

[3] Wikipedia : CUDA

`https://en.wikipedia.org/wiki/CUDA`

[4] Andrew NG *Machine Learning Course, Coursera.*