CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

In the present era of intelligent data analysis, the high performance organizations face three challenges. They are data analysis, data growth, different data formats and less time for the decision-making. In any organization, data is an important asset. To store the data prevailing in any of the companies digitalization helps providing inexpensive data storage devices. It will help to store all related transaction data in the form of large information system. In today’s business world, the data bases can range in size into the tera bytes or peta bytes of data. This is an indication of tremendous data growth and this comes with a difficulty in terms of retrieving the hidden and useful information, which can be used for decision making. This opens gate for a unique technique, which will work effectively to retrieve the hidden and useful decision making information even in the midst of data growth in the databases.

Transaction database may have different types of data format. For example patient’s record in hospital management system may have the following different types of data fields.

Sex = Categorical Attributes (Male or Female).
Age = Numerical attribute ( years).
Weight = Numerical attribute ( 0 to 9).
Sports = Numerical attribute ( 1 to 10).
Sleep = Numerical attribute ( 0 to 24).
Drink = Numerical attribute (0 to 5).
Blood Pressure = Categorical Attributes (HP, LP, NP).

Mining of different data format records is not an easy task as they are not in a common format. Some of the machine learning techniques can not work on this different type of dataset.

Decision need to be made quickly and correctly, using all available data. The amount of data doubles every 24 months, which affects response time and the sheer ability to comprehend its content. Competition is getting tougher in the area of business intelligence and added information value. With the given time limit the policy makers have to make the effective decision or else there will be a huge damage in terms of customer and their business. As a result we need a unique technique, which will help us to retrieve the hidden and useful information as quick as possible with in the stipulated time period [Pushparaj (2008)].

In this thesis, a decision tree based, predictive classifier called Mixed Mode Database Miner refered as MMDBM here after, has been taken into consideration and is implemented in Java for CPU computation and CUDA for GPU computation. The main problem discussed in this thesis is implementing the algorithm presented in MMDBM with the help of GPU computing thereby reducing the time taken for classification and comparing it with the time taken for implementing the algorithm in CPU computing.

This is the first step of our research work where the goal is to generalize the idea of applying GPU computing in each and every area of computation wherever it is needed.
1.1.1 Problem Description

The main problem discussed in this thesis is implementing the algorithm presented in MMDBM. This is based on decision tree classifier with the help of GPU computing thereby reducing the time taken for classification and comparing it with the time taken for implementing the algorithm in CPU computing. To record the difference in its behavior when we increase the number of data.

1.1.2 Project Goal

The goal of this project can be divided as follows:

Concept: Understand the concept of all existing classification techniques and how these can be implemented in CUDA.

Method: After the literature study, find the technique which is more appropriate for implementing the discussed algorithm in CUDA.

Evaluate: Evaluate the discussed algorithm in CUDA and its behaviour when the number of data records are increased.

1.2 HISTORY OF GPU

The history of graphic processors is as long as the PC itself. The first graphic processor called CGA (Color Graphics Adapter) was invented by IBM in 1981. After one and half decades of development, graphic processor has become more powerful and was able to support 3D acceleration on PC desktop. Then in 1999 the term Graphic Processing Unit was born when Nvidia introduced “the world’s first GPU” - GeForce256. From then onwards, research
in the fields of physics, medical imaging and so on started to take advantage of GPU to accelerate their applications. This is considered as the beginning of GPGPU computing.

However, since the GPU was not first invented for computing, scientists had to use graphic programming language like OpenGL to program the GPU. Developers had to map their scientific applications to “graphic applications” [ NVIDIA (2009) ]. That was really a big challenge and limited the accessibility and performance of the GPU for scientific computing. Seeing the big market in GPU computing, Nvidia modified their GPU and made it more programmable. In addition, a new GPU programming model CUDA was introduced by Nvidia in 2007. After that, developers were able to easily program the GPU using CUDA at ease, which is just an extension of standard C.

1.3 ARCHITECTURE GPU

Nvidia is one of the biggest GPU provider in the world and its CUDA-Capable GPUs are of great performance in the field of GPU computing. Figure 1.1 shows the basic architecture of a CUDA-capable GPU. As Figure 1.1 illustrates, a CUDA-capable GPU is organized into a set of blocks and size of the set can be different from one generation to another. Each block consists of two streaming multiprocessors(SMs) and the number of streaming multiprocessors generation is dependent. Looking into one streaming multiprocessor, there are eight streaming processors(SPs), which share the same instruction cache and control logic. GPU used for computing always comes with a big (several giga bytes) Graphic Double Data Rate(GDDR) DRAM. The GDDR DRAM is of very high bandwidth and can be used as global memory for GPU [ NVIDIA (2010) ].
1.3.1 Need for GPUs

To execute the parallel computing, GPU is required. A diagrammatic comparison between GPU and CPU architecture is given in the Figure 1.2. CPU architecture has few computing cores optimised for sequential serial processing, while a GPU architecture consist of thousands of smaller computing cores to handle multi-tasks parallely. GPU uses these massive parallel cores to provide high memory bandwidth for high performance computing. GPU’s architecture is similar to that of a CPU, despite having a lot of computing cores. There are many architectures available namely Tesla, Fermi and Kepler. Out of these NVIDIA based GPU architecture is Kepler [ NVIDIA (2014) ].
1.4 GPU MULTIPROCESSOR MODEL

GPUs rely on a set of multiprocessors. In order to process a great amount of vertices and fragments at the same time, it’s clear from (NVIDIA, 2007d) that the device is implemented as a set of multiprocessors as illustrated in Figure 1.3. Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD): At any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data. This way, GPUs can be seen as a set of SIMD Multiprocessors with on-chip shared memory, which is rather different from PRAM (Parallel Random Access Machine) models (MILLER; BOXER, 2000). The latter models were created in order to be widely used as a parallel model of computation, so that they would do for parallel computing what the RAM2 model did for sequential computing [Nguyen (2007)].

The PRAM model consists of a set of processors \((P_1, P_2, \ldots, P_n)\) each of which is identical to a RAM processor, a memory element. This is a com-
mon global memory, which is seen by all processors and a memory access unit, which is supposed to access every memory location in $\Theta(1)$ time, for every processor. It is important to note that processors will need to communicate to each other through commonly shared memory if they wish to cooperate [NVIDIA (2014)].

![Figure 1.3 GPU Multiprocessor Structure](image)

Several facts refrain GPUs from being classified as general PRAMs. One of them is that, different types of classifications can be applied based on the instruction issued. From (NVIDIA, 2007), if the instruction is executed by a warp3, it writes to the same location in global or shared memory for more than one of the threads of the warp, how many writes occur to that location and the order in which they occur is undefined, but one of the writes is guaranteed to succeed. This would classify G80 architecture as an arbitrary concurrent-
read, concurrent-write PRAM. But atomic instructions can be used, if an atomic instruction executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, each read, modify, write to that location occurs and they are all serialized, but the order in which they occur is undefined, so all processors are granted to read and write. Atomic instructions do not fit in any of PRAM classifications. Another reason for not modeling GPUs as PRAMS is that it considers all memory access times equal to every processor. Since memory accesses are important parts of algorithm implementations on GPUs, this modeling must be very careful.

1.5 COMPUTER UNIFIED DEVICE ARCHITECTURE (CUDA)

CUDA is a general purpose parallel computing architecture introduced by NVIDIA in November 2006 that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a Central Processing Unit [NVIDIA (2014)]. It transparently scales its parallelism to adopt to the continue increase in number of processor cores in today’s graphics hardware which is chiefly driven by the computational demands of high-end games.

A CUDA program can usually be divided into two part: host code and device code. Host code, as the name indicated, is the code that is run on the host, which is usually a traditional CPU. It is the serial part of the program which is written in straight ANSI C. Device code, on the other hand, is the parallel part of the program which is written in ANSI C extended with CUDA keywords. A complete CUDA program is mixed source code with both host code and device code.

The NVCC compiler will separate them during compilation. Then the ANSI C code will be compiled with host’s standard C compiler and later it
is run on the CPU. The GPU code, also know as kernel, will be further compiled by NVCC and mapped to the GPU device.

### 1.5.1 CUDA Thread Organisation

Threads - basic processing units on GPU - are organised into a 3-stage hierarchy. At the bottom level threads are grouped into warps - there are 32 threads in one warp, and each warp is divided into two half-warps. The characteristic of a half-warp is similar to the SIMD architecture in which all threads in a warp execute the same set of instructions simultaneously - any differences in the execution must be serialised. The advantage of using warps is that whenever a warp has to wait for some long-latency operation another free warp may be executed, so that long-latency operations are efficiently hidden [NVIDIA (2010, NVIDIA (2014)]. Since warps are not part of CUDA language definition (programmer cannot explicitly influence warp behaviour), from programming point of view the threads are organised into thread blocks.

The size of a thread block is determined by programmer and is arbitrary, although it is recommended to use sizes which are multiple of warp size. The requirement is that each block must run independent of each other so that they can be scheduled across any number of cores. This allows aforementioned scalability of CUDA programs. Independence of each block comes at the expense of inter-block synchronisation and fast memory sharing when none of which is feasible. Within a block, however, threads cooperate with each other using shared memory and a barrier function — `syncthreads()` may be used to synchronise threads. It is guaranteed that a thread block resides on one SM.

Blocks of threads are in turn grouped into one- or two-dimensional grids. Whereas the size of thread blocks is dictated by the optimisation process, the number of blocks in a grid is determined by the size of data dealt with. Only
one grid exists on one graphics card and there is a separate grid for each device in a system. Which is given Figure 1.4.

Figure 1.4 Grid Block in CUDA (NVIDIA)

1.5.2 GF108 Architecture

The MMDBM Classifier with the help of GPU, we have compared different types of sorting procedure about CPU Computing, GPU Computing and CUDA. We start NVIDIA GeForce GT 525M with Fermi based GPU architecture [12, 13]. The NVIDIA GeForce GT 525M is a relatively fast mid-range laptop graphics card and the inheritor to the GeForce GT 425M. It is based on
the GF108 core as measure of the Fermi architecture. Consequently, it supports DirectX 11 and OpenGL 4.0. Likened to the GT 425M, core clock rates of the GT 525M have been increased by about 7 percent.

The GF108 core of the GT 525M is connected to the GF100 core in the GeFore GTX 480M and offers 96 shaders and a 128 Bit memory bus for DDR3 VRAM [Nguyen (2007)]. Except for the memory controllers, the GF108 can basically be measured by a halved GF106. Hence, the architecture is not directly equivalent to the old GT215 (e.g., GeForce GTS 350M) or GT216 (e.g., GeForce GT 330M) cores. Unlike the GF100, the smaller GF104, GF106, and GF108 cores were not only summarized, but also considerably adjusted. In dissimilarity to the GF100, which was measured for qualified applications, these final chips target the consumer market. They feature more shaders (3x16 instead of 2x16), more texture units (8 instead of 4) and more SFUs per streaming multi-processor (SM). As there are still only 2 warp schedulers (versus 3 shader groups), Nvidia now uses superscalar execution in order to utilize the higher amount of shaders per SM more efficiently. In theory, the performance per core should be greatly improved over previous generations [Nickolls (2008), NVIDIA (2010), NVIDIA (2014)].

CUDA is a general purpose parallel computing architecture containing a new parallel programming model and an instruction set architectures [Sanders (2011)]. CUDA is an extension of the C language. A CUDA program mostly contains of CPU code and at least one kernel, i.e., void returning function to be implemented by the GPU. The key words —— global —— qualifier to the kernel function which is called by CPU, we have executed the function on our GPU. The —— device —— keyword lets us mark functions as callable from threads executing on the device by GPU. The —— host —— keyword is for function which can only be called by CPU. Both —— host —— and —— device —— keyword is for function as qualifiers can be combined. Note that the function
--- global --- and --- device --- functions have access to these automatically defined variables [Nguyen (2007), Nickolls (2008)]. A variable is given by dim3 gridDim - Dimensions of the grid in blocks (at most 2D), dim3 blockDim - Dimensions of the block in threads, dim3 blockIdx - Block index within the grid, dim3 threadIdx - Thread index within the block with the keyword --- shared --- indicates that it will be stored in the shared memory of SM. The number of blocks in a grid and threads in a block should be declared by using dim3 announce, while CUDA kernel. Refer the Figure 1.5 is given below. The variable dim3 should be incorporated as a parameter as follows:

```c
    dim3 gridDim(i,j,k);
    dim3 blockDim(p,q,r);
    kernel function <<<gridDim, blockDim >>> (a,b,c);
```

Where $i$, $j$ and $k$ are the number of blocks in $x$, $y$ and $z$ directions in grid. $p$, $q$ and $r$ are the number of threads in $x$, $y$ and $z$ directions in a block. $a$, $b$ and $c$ are the parameters of the kernel. The CUDA function calls differ from C function call only by the part <<<gridDim, blockDim >>>. This kernel is executed on GPU and called from CPU [NVIDIA (2014)]. This kernel function should be declared with the Keyword --- global ---. The CUDA API essentially comprises functions for memory manipulation in VRAM: cudaMalloc to allocate memory, cudaFree to free it and cudaMemcpy to copy data between RAM and VRAM and vice-versa. We will end this section by explaining how a CUDA program is compiled. Compiling is done in several levels.

In the first level, the code dedicated to CPU is extracted from the file and passed to the standard compiler. In the next level, the code dedicated to the GPU is converted into an intermediate language PTX which is like an assembler. Finally, the last level translates this intermediate language into commands that are specific to the GPU and encapsulates them in binary form which is executable [Pospichal (2010), Sander (2011)].
1.5.3 Programming in CUDA

Before starting to write programs in CUDA, we need a basic understanding of C or C++ programming. Few things which we need to keep in mind while programming in CUDA are listed below [Chun-chieh (2011) ].

1. CUDA provides function type qualifiers that are not in C/C++ to enable the programmer to define where a function should run.

2. $\text{-- host --} :$ if the function declaration contains this qualifier then it specifies that code should run on the host CPU (it is the default) [ Vennusatuluri (2007) ].
3. **device**: if the function declaration contains this qualifier then it specifies that code should run on the GPU and the function can only be called by code running on the GPU [Shifu (2009)].

4. **global**: if the function declaration contains this qualifier then it specifies that code should run on the GPU but have to be called from the host - this is the access point to start multi-threaded codes running on the GPU [Greg (2010)].

5. GPU device can’t execute code on the host.

6. CUDA imposes some restrictions, such as device code is C-only (host code can be C++), device code can’t be called recursively.

7. All calls to a global function must specify how many threaded copies are to be launched and in what configuration.

8. Call for any global or device function is defined by a specific syntax `<<< >>>` [NVIDIA (2009), NVIDIA (2010)].

9. Inside the `<<< >>>` syntax, we need at least two arguments to be present for calling any global or device function, one for blockgrid and another for number of thread blocks.

10. A typical function call looks like function name `<<< bg; tb >>>`.

11. `bg` specifies the dimensions of the block grid and `tb` specifies the dimensions of each thread block.

12. A new datatype called `dim3` has been defined by CUDA in which three unsigned integer components defaults to 1.

13. `dim3` has struct-like access - members are `x`, `y` and `z`.

14. For the code running on the GPU (device and global), some variables are predefined, which allow threads to be located inside their `blocks` and `grids` [NVIDIA (2010)].
15. *dim3 gridDim* which specifies the dimensions of the grid [Hards (2007)].

16. *uint3 blockIdx* an unsigned integer which refers to the location of the block in the grid.

17. *dim3 blockDim* which specifies the dimensions of the block.

18. *uint3 threadIdx* an unsigned integer which refers to the location of the thread in the block [Nickolls (2008)].

19. For code running on the GPU, both device and global code, the memory used to hold their variable can be specified.

20. **−− device −−**: if the variable declaration contains this qualifier then it specifies that the variable resides in the GPUs global memory and is defined while the code runs.

21. **−− constant −−**: if the variable declaration contains this qualifier then it specifies that the variable resides in the constant memory space of the GPU and is defined while the code runs.

22. **−− shared −−**: if the variable declaration contains this qualifier then it specifies that the variable resides in the shared memory of the thread block and has the same lifespan as the block.

### 1.5.4 CUDA Code Execution

Figure 1.6 shows the typical execution of a CUDA program. Basically there are two kinds of execution: synchronize and asynchronous. In the synchronize execution, program starts the host code, which is serially executed. When a kernel is invoked, the GPU device will take charge and a large number threads will be generated to take advantage of the parallelism. The CPU will be suspend until
the GPU finish its work, and then continues executing until the end of the code. However, in the asynchronous execution, when a kernel is invoked, the CPU will continue while the kernel code is launched on the GPU.

![CUDA code execution diagram](image1)

**Figure 1.6 CUDA code execution**

![CUDA Runtime API diagram](image2)

**Figure 1.7 CUDA Runtime API**
1.5.5 CUDA Runtime API

CUDA runtime API is a set of API functions that provide some certain kinds of services, such as data transfer and so on. As shown in Figure 1.7, it is an intermediate software layer between application and CUDA drivers. Some CUDA libraries also call some CUDA runtime API. In contrast to the low level CUDA driver API, CUDA runtime API act as high level API. Each function call of the CUDA runtime API will break down to some more basic instructions that manage the low level CUDA driver API [NVIDIA (2010), NVIDIA (2014)].

1.6 ORGANISATION OF THE THESIS

This thesis comprises of five chapters and discusses the proposed algorithm using different types of sorting procedures and a decision tree based, predictive classifier called Mixed Mode Database Miner referred as MMDBM. It has been taken into consideration and is implemented in Java and CUDA programming for graphics processor unit (GPU) computation.

Chapter 1, deals with the introduction of parallel processing and a detailed literature review of GPU, CUDA programming model, Comparison of experimental results as well as background for the remaining chapters of the thesis.

Chapter 2, Algorithm and techniques. Data mining is used to extract hidden information from large datasets by using different types of techniques namely, classification, association rule and clustering. Classification is one of the most popular techniques in data mining, where billions of business users and industries are using the fast classifier mining algorithm for classifying the data.
Chapter 3, A comparative study on MMDBM classifier incorporating various sorting procedure is implemented in Java programming. Classification is one of the most important method in data mining. The classifier based on decision tree is called Mixed Mode Data Base Miner (MMDBM), which is tested with different types of sorting techniques (merge sort, Quick sort and Radix sort algorithms) to compare the processing time with SLIQ classifier.

Chapter 4, Parallel computation of MMDBM algorithm with CUDA programming. NVIDIA Corporation, a leader in GPU market, introduced a general purpose parallel computing architecture in November 2006, to harness the computing capabilities of their high-end GPUs. A decision tree classifier called Mixed Mode Database Miner (MMDBM) used to classify large number of data sets with large number of attributes is implemented with different types of sorting techniques (Quick sort and Radix sort) both in Java programming and GPGPU Computing (CUDA) and the results are discussed extensively in this chapter.

Chapter 5, An Empirical Study of Supervised Learning Methods for Breast Cancer Diseases. Cancer is a leading disease in the world that causes an increase in death rate, both in developing and developed nations alike. Of all types of cancers, Breast cancer is almost an incurable form of cancer and incurs high expenditure for treatment. Among the database of cancer patients, breast cancer is on the increase year after year. To classify these type of diseases, ANN and Decision tree techniques were applied in most of the studies. This chapter presents the results of an empirical study and comparative analysis of Supervised Learning methods that use four different types of datasets that culminates in the proposition of a classifier, called MMDBM (Mixed Mode Database Miner) as one of the best classifiers among 19 Supervised Learning methods.