CHAPTER 4

PARALLEL COMPUTATION OF MMDBM ALGORITHM WITH CPU AND CUDA GPU MINING USING RADIX AND QUICK SORT ALGORITHMS

4.1 INTRODUCTION

NVIDIA Corporation, market leader in GPU market, introduced a general purpose parallel computing architecture in November 2006, to harness the computing capabilities of their high-end GPUs. Compute Unified Device Architecture (CUDA) is based on a new parallel programming model and instruction set architecture that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language. Other languages such as FORTRAN, C++, OpenGL, and DirectX will be supported in the future.

Compute Unified Device Architecture (CUDA) [ NVIDIA (2010) ] allows programmers to work together straight with the GPU and run programs on them, thus efficiently utilizing the advantage of parallelization. CUDA is not a new programming language, rather an extension to C with GPU- particular commands, options and operations. [ NVIDIA (2010) ] Programs written in CUDA are compiled by nvcc compiler and can be run only on NVIDIA’s GPU’s. A CUDA program can be run on several number of processors core and only the number of processor requirements are to be known to the runtime system.
A decision tree classifier called Mixed Mode Database Miner (MMDBM) which is used to classify large number of datasets with large number of attributes is implemented with different types of sorting techniques (Quick sort and Radix sort) in both CPU Computing (java) and GPGPU Computing (CUDA) and the results are discussed. This classifier is suitable for handling large number of both numerical and categorical attributes. The MMDBM Classifier has been implemented in CUDA GPUs and the code is provided. We used the parallelized algorithms of the two sorting techniques on GPU using Compute Unified Device Architecture (CUDA) parallel programming platform developed by NVIDIA Corporation. In this chapter, we have discussed an efficient parallel (Quick sort and Radix sort) sorting procedures on GPGPU Computing and compared the results of GPU to the CPU Computing. The main result of MMDBM is used to compared the classifier with an existing CPU Computing results and GPU Computing results. The GPU sorting algorithms provide quick and exact results with less handling time compare to other classification algorithms.

4.2 DESIGN OF MMDBM CLASSIFIER ON CPU AND GPU COMPUTING

The classification proceeds in two different phases: Attribute selection of algorithm and sorting attributes.

4.2.1 Attribute Selection

The first phase is the attribute selection. This is done by accessing the randomly generated database and detecting the attribute values and the type of every attribute [ Sundar (2006) ]. Once an attribute is detected, its information is stored in a list called “Attribu”, Table 4.1 is given below.
Table 4.1 Attribute Collection

<table>
<thead>
<tr>
<th>Attribute</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical</td>
<td>{integer}</td>
</tr>
<tr>
<td>Categorical</td>
<td>{string}</td>
</tr>
<tr>
<td>BP</td>
<td>{Low BP, High BP, Normal BP}</td>
</tr>
</tbody>
</table>

Then sorting of the random data from the database inside the GPU device takes place. This is done for every numerical attribute and the result is stored in an another array structure in the host. Once all the data is sorted, the split point can be found by accessing the middle element of the sorted array and it is stored in a variable called Mid* where * represents the name of every attribute, which is shown in Table 4.2.

Table 4.2 Sorting attributes

<table>
<thead>
<tr>
<th>Sorting attribute (Quick sort)</th>
<th>Attribute name {value}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Midage=result/2</td>
</tr>
</tbody>
</table>

Once presorting is complete, the arrays containing the corresponding attribute values are created. This array has been loaded into the memory for classification. The leaf entry of each class list is initialized to ‘0’, the root node of the tree.

4.3 MMDBM ALGORITHM

**Input:** A is the set of attributes containing n attributes $A=\{a_1, a_2, \ldots, a_n\}$ from data base

**Output:** Distribution of the node count and construction of the tree. Data value is generated randomly in database Transfer the data from GPU device to CPU host (cuda memory) to dispatch the value in arrays.
Copy to GPU device and Radix sort for sorting the random data value from the data sets inside the device GPU.

Get the mid-point value of each and every attributes

\( a_i \) is the attribute name and \( v_i \) is the midpoint values of each attributes, \( C=0 \) is the class value and \( M=0 \) is the missing value.

For \( i = 1 \) to \( n \) // \( n \) is the number of the attributes nodes scan the attribute of all the records

IF \( (a_1 \leq v_1) \) AND \( (a_2 \leq v_2) \) AND ... AND \( (a_n \leq v_n) \) THEN \( C \).

IF \( a_i \leq v_i \) is true goto left side node traverse up to \( N \) number of the Node THEN
count the class values
\[
C = C + 1;
\]
else

IF \( a_i \leq v_i \) is false goto right side node traverse up to \( N \) number of the node.
Count the class value or traversal node that already exist then update the appropriate class count value.
\[
C = C + 1;
\]
else
Count the missing values and update the count value.
\[
M = M + 1;
\]
class and missing count values, if the same data exists then update the appropriate class count values.

End IF
End IF
End For

Transfer the data from CPU host to GPU device, classify the data, Compute the node count and class count.
Copy the result to CPU host, generate the distribution of the node count and construction of the decision tree (cuda Memcpy GPU device to CPU host).
4.4 GPU RADIX SORT ALGORITHM

If Radix sort is one of the oldest and well-known sorting algorithms [Satish (2009)]. It is frequently used amongst the most efficient for sorting small keys. Radix sort procedures assume that the keys are represented as d-digit numbers in a radix-r notation. On binary computers, it is likely to assume that the radix r = 2b and that the keys are an integral multiple of b bits in length. The sorting algorithm itself consists of d passes which consider the i-th digits of the keys in order from least to most significant digit [Dusseau (1996), Grand (2007)]. In each pass, the input sequence is sorted with value to digit i of the keys, with the requirement that this sort be stable (i.e., it conserves the comparative ordering of keys with equal digits). The sorting algorithm used within each pass of radix sort is usually a counting sort or bucket sort [Coreman (2001), Satish (2009), Zagha (1991)]. In a single pass, each key can be placed into one of r buckets. To compute the output index at which the element should be written, which we refer to as the rank of the element, we must simply count the number of elements in the lower numbered buckets plus the number of elements already in the current bucket. Having computed the rank of each element, we complete the sorting step by spreading the elements into the output array in the location determined by their ranks. A parallel algorithm of radix sort is given below.

4.4.1 GPU Code for Radix sort Algorithm

```cpp
__global__ static void Radix_sort ( int* values, int* temp, int loop, int* split, int* e, int* f, int* t)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int remainder[N], quotient[N];
    int f_count, totalFalses;
    if (idx < N) {
```

Center For Research-Anna University Chennai
// split based on least significant bit
quotient[idx] = values[idx];
for (int x = 0; x < loop + 1; ++x) {
  remainder[idx] = quotient[idx] % 10;
  quotient[idx] = quotient[idx] / 10;
}
// set e[idx] = 0 in each 1 input and e[idx] = 1 in each 0 input
if (remainder[idx] == 1) {
  split[idx] = 1;
  e[idx] = 0;
} else {
  split[idx] = 0;
  e[idx] = 1;
}
__syncthreads();
if (idx < N) {
  // scan the 1s
  f_count = 0;
  for (int x = 0; x < N; ++x) {
    f[x] = f_count;
    if (e[x] == 1)
      f_count++;
  }
  // calculate totalFalses
  totalFalses = e[N-1] + f[N-1];
  if (split[idx] == 1) {
    // t = idx - f + totalFalses
    t[idx] = idx - f[idx] + totalFalses;
else if (split[idx] == 0) {
    // t = f[idx]
    t[idx] = f[idx];
}

// Scatter input using t as scatter address
temp[t[idx]] = values[idx];

__syncthreads();
// copy new arrangement back to values
if (idx < N) {
    values[idx] = temp[idx];
}

4.5 GPU QUICK SORT ALGORITHM

4.5.1 GPU Code for Quick sort Algorithm

__global __ static void quicksort (int*values)
{
    int pivot, L, R;
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int start[MAX_LEVELS];
    int end[MAX_LEVELS];
    start[idx] = idx;
    end[idx] = N - 1;
    while (idx >= 0) {
        int split = (start[idx] + end[idx]) / 2;
        if (split == 0) {
            // t = f[idx]
            t[idx] = f[idx];
        }
        // Scatter input using t as scatter address
        temp[t[idx]] = values[idx];
        __syncthreads();
        // copy new arrangement back to values
        if (idx < N) {
            values[idx] = temp[idx];
        }
    }
}
L = start[idx]; R = end[idx];
if (L < R) {
    pivot = values[L];
    while (L < R) {
        while (values[R] ≥ pivot && L < R)
            R–;
        if (L < R)
            values[L++] = values[R];
        while (values[L] < pivot && L < R)
            L++;
        if (L < R)
            values[R–] = values[L];
    }
    values[L] = pivot;
    start[idx + 1] = L + 1;
    end[idx + 1] = end[idx];
    end[idx++] = L;
    if (end[idx] - start[idx] > end[idx - 1] - start[idx - 1])
        { swap start[idx] and start[idx-1]
          int tmp = start[idx];
          start[idx] = start[idx - 1];
          start[idx - 1] = tmp;
          // swap end[idx] and end[idx-1]
          tmp = end[idx];
          end[idx] = end[idx - 1];
          end[idx - 1] = tmp;
        }
    }
else idx–;
}
4.6 PROPOSED METHOD

Today the GPU mining can reduce the classification and processing time of parallel code by many times. It comes at the expense of significant power and energy consumption [ Sundar (2006) ]. In this section, we propose parallel computing approach for construction of decision tree on GPU mining. The reasons why we have selected MMDBM decision tree algorithm to use split point to apply every node is that implementation of the GPU mining using radix sort algorithm. Finally, count the class values.

There are two parts of creating attribute lists. The first is moving the data to its matching list. After completing the data movement, we need to sort every attribute lists. There is a well-known CUDA library, which is called as CUDPP. The sorting procedure of CUDPP can sort two 1-D arrays as input, the first is called key array and the second is called value array. The key array would sort and the element of value array would change its position according to its corresponding key element. It is called key value pair sorting. The sorting algorithm of CUDPP only supports a key value pair sorting, but we have two values for one key (key is the attribute value field, rid and class label are values).

In order to get the greatest performance, we have changed the CUDPP sorting algorithm into one key for two values. The kernel comprised of a grid of scalar threads. Every thread has given a unique identifier (thread ID) which can be used to help divide up work among the threads. Within a grid, threads are grouped into blocks, which are also referred to as cooperative thread arrays (CTAs), where a single CTA thread has access to a frequent high-speed memory called the shared memory. We modified the sorting algorithm from the CTA level to public interface level [ Cederman (2009), Telbany (2006) ].

The following steps illustrate the main execution step in our imple-
mentation. Further, Figure 4.1 shows which parts of the execution are done on the host CPU and on the device GPU, respectively, as well as the data transfers that take place between the host and the device (GPU).

This algorithm is a six step process. The data has generated randomly in GPU device, it is transferred to the CPU host and data values are stored in an array (step 1 and step 2). The array values are copied to GPU and all the numeric attributes values are sorted using radix sort algorithm (step 3). The sorting data values are transferred to CPU host to get mid-point value of all the numeric attributes (step 4). The mid-point value checks the condition of a record in each attribute value, the condition is true goto left side node and the condition is false goto right side node which is based on the MMDBM algorithm. Finally
the class values are counted and the histogram of node level is calculated and the condition which is checked from top to bottom is called distributes (step5). (refer the tree). The final results are transferred from GPU to CPU and the acceleration ratio time of GPU mining is calculated.

4.6.1 Implementation of Algorithm for GPU

The pre-sorting process is completed and to the mid-point value of each attributes is found

IF \( \text{age} \leq \text{midpoint} \) Then C.

Age is the attribute variable name and values are in data sets table. If the condition is either true or false, the corresponding node data is passed to another node pointed as leaf from the corresponding parent node, which is called split point [ Sundar (2006), Ömer (2013) ].

![Figure 4.2 Splitting point for node](image)

4.6.2 Split points code for Quick and Radix sort Algorithms

By our algorithm, we find the mid-point by sorting the array elements and storing the middle element of the sorted array. There by we have made only
one pass in finding the midpoint. Once the midpoint is calculated, the arrays are passed to the GPU classification function along with the midpoints to classify the records. The GPU quick and radix sort code for finding the mid-points is given below.

**Radix sort Algorithm**

Radix sort <<< 1, SO >>> (d_values, t_values, 1, _split, d_e,d_f,d_t);
cudaMemcpy(sag,d_values, size, cudaMemcpyDeviceToHost);
printf("MidPoint of the AGE is:");
MA=age[SO/2]; // mid point for Age
printf("Midpoint value of AGE ", MA);
cudaMemcpy(ad_values, wt, size, cudaMemcpyHostToDevice);
Radix sort <<< 1, SO >>> (ad_values, at_values, 1, ad_split ,ad_e, ad_f, ad_t);
cudaMemcpy(swt,ar_values, size, cudaMemcpyDeviceToHost) ;
printf("MidPoint of the WEIGHT is:");
MW=swt[SO/2]; // mid point for Weight
printf("Midpoint value of WEIGHT ", MW);
Radix sort <<< 1, SO >>> (sld_values, slt_values, 1, sld_split ,sld_e, sld_f, sld_t);
cudaMemcpy(slt,slr_values, size, cudaMemcpyDeviceToHost);
printf("MidPoint of the SLEEP is:");
MSLP=slt[SO/2]; // mid point for Sleep
printf("Mid point value of SLEEP", MSLP);
Radix sort <<< 1, SO >>> (spd_values, spt_values, 1, spd_split,spd_e, spd_f, spd_t);
cudaMemcpy(spt, spr_values, size, cudaMemcpyDeviceToHost) ;
printf("MidPoint of the SPORTS is: ");
MSP=spt[SO/2]; // mid point for Sports
printf("Midpoint value of SPORTS",MSP);

Radix_sort <<< 1, SO >>>(drd_values,drt_values, 1, drd_split,drd_e, drd_f, 
drd_t);
cudaMemcpy(drt,drr_values,size, cudaMemcpyDeviceToHost) ;
printf("MidPoint of the DRINKING HABBIT is:");
MDR=swt[SO/2]; //mid point for Drinks
printf("Midpoint value of DRINKS",MDR);

Quick sort Algorithm

quick_sort <<< MT / cThreadsPerBlock,MT/c ThreadsPerBlock, N >>> (d_values);
cudaMemcpy(sag,d_values,size, cudaMemcpyDeviceToHost) ;
printf("MidPoint of the AGE is: ");
MA=sag[SO/2]; //Mid-point of the Age
printf("Mid point value of AGE”, MA);

quick_sort <<< MT / cThreadsPerBlock, MT/c ThreadsPerBlock,N >>> (r_values);
cudaMemcpy(swt,r_values,size, cudaMemcpyDeviceToHost) ;
printf("MidPoint of the WEIGHT is ");
MW=swt[SO/2]; //Midpoint of the Weight
printf("Mid point value of WEIGHT”, MW);

quick_sort <<< MT / cThreadsPerBlock, MT/c ThreadsPerBlock,N >>> (a_values);
cudaMemcpy(sslp,a_values,size, cudaMemcpyDeviceToHost) ;
printf("Mid Point of the SLEEP is");
MSLP=sslp[SO/2]; // Midpoint of the Sleep
printf("Mid point value of SLEEP”, MSLP);
quick_sort <<< MT/cThreadsPerBlock, MT/cThreadsPerBlock, N >>> (s_values);
cudaMemcpy(sdr,s_values,size,cudaMemcpyDeviceToHost);
printf(“MidPoint of the DRINK is:”);
MDR=sdr[SO/2]; //Midpoint of the Drinks
printf(“Mid point value of DRINKS”,MDR);

quick_sort <<< MT/cThreadsPerBlock, MT/cThreadsPerBlock, N >>> (v_values);
cudaMemcpy(ssp,v_values,size,cudaMemcpyDeviceToHost);
printf(“Mid Point of the SPROTS is”);
MSP=ssp[SO/2]; //Mid-point of the Sports
printf(“Mid point value of SORTS”,MSP);

After getting the midpoint values and scanning the attributes of all
the records from connected data sets, we classify the node using IF (x_1 \leq v_1)
AND (x_2 \leq v_2) AND ... AND (x_n \leq v_n) (class value) rule [ Sundar (2006) ].
IF this rule is true goto left child node and travel up to N number of nodes and
finally count the class value, if the same data exists then count the value and up-
date the appropriate class count. IF condition is false goto right child node and
travel up to N number of nodes and finally count the class value, if the same data
exists then count the value and update the appropriate class count and else up-
date the missing count. Finally the distribution of the node counts are evaluated
based on the Predicted Rules Table 3.3 and the histogram of the classified nodes
are calculated with the construction of decision tree Figure 4.5. The design of
MMDBM algorithm is presented in Figure 4.1.

4.7 ACCELERATION RATIO FOR GPU

GPU Performance To test the acceleration performance, an accelera-
tion ratio (speed-up) γ is defined as γ = \frac{t_{CPU}}{t_{GPU}} where the total processing time on
the CPU, $t_{CPU}$, comprises only the time of main loop executed while the total processing time on the GPU, $t_{GPU}$, includes additional time of transferring data between Host and Device in the interest of fairness. $\gamma$ first rises as the number of threads increases. There are two reasons for the changes in $\gamma$. First, if the number of threads are less, the time spent on data transferring between Host and Device takes up a considerable proportion of the total processing time of the GPU. As the number of threads increases, the proportion decreases rapidly. Second, only after all blocks in a kernel is executed, the next kernel can be launched in the GPU. The time consumed on kernel launching can be roughly considered as a fixed cost. Using more blocks means a reduction in the percentage of kernel launching time [ Panchatcharam (2013) ].

**CPU Time** = Generate the random Values + Sorting Time + Classification Time.

**GPU Time** = Data transfer from Host to Device and Device to Host.

**Acceleration Ratio** = CPU computation Time / GPU computation Time.

we have calculated CPU, GPU and Acceleration ratio time and the sorting time for all the records is 0.00, because the number thread has been created.

### Table 4.3 Acceleration Ratio time for Quick sort

<table>
<thead>
<tr>
<th>Quick sort with MMDBM Algorithm</th>
<th>Number of Records in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200 X 100</td>
</tr>
<tr>
<td>Generate random</td>
<td>0.100</td>
</tr>
<tr>
<td>Sorting</td>
<td>0.000</td>
</tr>
<tr>
<td>Classification time</td>
<td>0.510</td>
</tr>
<tr>
<td>CPU Time</td>
<td>0.610</td>
</tr>
<tr>
<td>GPU Time</td>
<td>0.510</td>
</tr>
<tr>
<td>Acceleration</td>
<td>1.196</td>
</tr>
</tbody>
</table>
Table 4.4 Acceleration Ratio time for Radix sort

<table>
<thead>
<tr>
<th>Radix-sort with MMDBM Algorithm</th>
<th>Number of Records in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200 X 100</td>
</tr>
<tr>
<td>Generate random</td>
<td>0.090</td>
</tr>
<tr>
<td>Sorting</td>
<td>0.000</td>
</tr>
<tr>
<td>Classification time</td>
<td>0.510</td>
</tr>
<tr>
<td>CPU Time</td>
<td>0.610</td>
</tr>
<tr>
<td>GPU Time</td>
<td>0.510</td>
</tr>
<tr>
<td>Acceleration</td>
<td>1.196</td>
</tr>
</tbody>
</table>

Comparison of acceleration ratio time for GPU Quick sort with Radix sort algorithm, is shown in Figure 4.3.

![Figure 4.3 Design of Fast Classifier Mining Algorithm](image)

4.8 MAIN RESULT

We have tested the efficiency of our MMDBM classification algorithm to implement in it different types of sorting algorithms (Quick sort and Radix sort) and the results of CPU Computing are compared with GPU Computing. We now consider Medical database for Blood Pressure (BP) where data mining techniques are applied. Test has been carried out to estimate the perfection of classification and the classification handling time. The medical database for Blood pressure where both the algorithms Quick sort and Radix sort are
given the task to predict the risk of persons for having high BP, low BP, normal BP based on seven different types of attributes.

4.8.1 Medical database for BP

The medical database contains data from reviews conducted among patients. The database holds records of the following Attributes.

1. Sex: Categorical [M/F];
2. Age: representing the age of the person; Numeric [years];
3. Weight: The weight of the person; Numeric [Kilo grams];
4. Sports: The extent of exercise of a person; Numeric [1-10]
5. Sleep: The number of hours a person sleeps on an average; numeric [0, 24];
6. Drink: The extent of drinking of a person; Numeric [1-5];
7. BP: Categorical [HP, LP, NP], this is class values;

We have classified all the attributes and got 24 distributions of the node and travelled path from 30,000 records and counted the number of patients with low, high and normal BP. Each distribution has been generated by multiple IF < condition > Then rule. This rule is generated dynamically based on mid-point value along with the predicted rules where all the mid-point value compared with all the database attribute value. Finally get the total class count value, distribution of the node count and travelled path which is given below.
4.8.2 Rules and Histogram of all the distributions

1. If( Sex == F & Age ≤ 47 & Weight > 35 & Sports > 5 & Sleep > 8 Drink > 3) count the BP value of LP-394, HP-450 and NP-391 and travelled path: 1-3-6-13-27-55-111.

2. If( Sex == F & Age ≤ 47 & Weight > 35 & Sports ≤ 5 & Sleep ≤ 8 Drink ≤ 3) count the BP value of LP-390, HP-421, NP-381 and travelled path: 1-3-6-13-26-52-104.

3. If( Sex == M & Age > 47 & Weight > 35 & Sports ≤ 5 & Sleep ≤ 8 & Drink > 3) count the BP value of LP-373, HP-392 and NP-353 and travelled path: 1-2-5-11-22-44-89.

4. If( Sex == M & Age ≤ 47 & Weight > 35 & Sports ≤ 5 & Sleep ≤ 8 & Drink ≤ 3) count the BP value of LP-393, HP-472, NP-372 and travelled path: 1-2-4-9-18-36-72.

5. If( Sex == M & Age ≤ 47 & Weight ≤ 35 & Sports ≤ 5 & Sleep ≤ 8 & Drink ≤ 3) count the BP value of LP-318, HP-456, NP-302 and travelled path: 1-2-4-8-16-32-64.

6. If( Sex == F & Age > 47 & Weight > 35 & Sports ≤ 5 & Sleep ≤ 8 & Drink ≤ 3) count the BP value of LP-411, HP-435, NP-417 and travelled path: 1-3-7-15-30-60-120.


8. If( Sex == M & Age ≤ 47 & Weight ≤ 35 & Sports > 5 & Sleep > 8 Drink ≤ 3) count the BP value of LP-447, HP-493, NP-427 and travelled path: 1-2-4-8-17-35-70.
9. If( Sex == M & Age > 47 & Weight > 35 & Sports > 5 & Sleep > 8 & Drink ≤ 3) count the BP value of LP-487, HP-448, NP-450 and travelled path: 1-2-5-11-23-47-94.

10. If( Sex == F & Age > 47 & Weight > 35 & Sports > 5 & Sleep ≤ 8 & Drink ≤ 3) count the BP value of LP-421, HP-493, NP-401 and travelled path: 1-3-7-15-31-62-124.

11. If( Sex == F & Age ≤ 47 & Weight ≤ 35 & Sports > 5 & Sleep ≤ 8 & Drink ≤ 3) count the BP value of LP-452, HP-418, NP-432 and travelled path: 1-3-6-12-25-50-100.

12. If( Sex == M & Age ≤ 47 & Weight > 35 & Sports > 5 & Sleep > 8 & Drink ≤ 3) count the BP value of LP-302, HP-482, NP-352 and travelled path: 1-2-4-9-19-39-78.

13. If( Sex == F & Age > 47 & Weight > 35 & Sports > 5 & Sleep > 8 Drink ≤ 3) count the BP value of LP-415, HP-470, NP-405 and travelled path: 1-3-7-15-31-63-126.


15. If( Sex == M & Age ≤ 47 & Weight ≤ 35 & Sports ≤ 5 & Sleep > 8 & Drink > 3) count the BP value of LP-427, HP-396, NP-444 and travelled path: 1-2-4-8-16-33-67.


18. If(Sex == F & Age ≤ 47 & Weight ≤ 35 & Sports ≤ 5 & Sleep ≤ 8 & Drink ≤ 3) count the BP value of LP-417, HP-481, NP-407 and travelled path: 1-3-6-12-24-48-96.

19. If(Sex == M & Age > 47 & Weight ≤ 35 & Sports ≤ 5 & Sleep ≤ 8 & Drink ≤ 3) count the BP value of LP-418, HP-423, NP-427 and travelled path: 1-2-5-10-20-40-80.

20. If(Sex == M & Age > 47 & Weight ≤ 35 & Sports > 5 & Sleep > 8 & Drink ≤ 3) count the BP value of LP-391, HP-401, NP-391 and travelled path: 1-2-5-10-21-43-86.

21. If(Sex == F & Age > 47 & Weight ≤ 35 & Sports ≤ 5 & Sleep ≤ 8 & Drink ≤ 3) count the BP value of LP-305, HP-482, NP-305 and travelled path: 1-3-7-14-28-56-112.

22. If(Sex == F & Age > 47 & Weight ≤ 35 & Sports > 5 & Sleep > 8 & Drink ≤ 3) count the BP value of LP-490, HP-491, NP-421 and travelled path: 1-3-7-14-29-59-118.

23. If(Sex == F & Age ≤ 47 & Weight ≤ 35 & Sports > 5 & Sleep > 8 & Drink ≤ 3) count the BP value of LP-293, HP-487, NP-388 and travelled path: 1-3-6-12-25-51-102.

24. If(Sex == M & Age ≤ 47 & Weight > 35 & Sports ≤ 5 & Sleep > 8 & Drink > 3) count the BP value of LP-292, HP-441, NP-225 and travelled path: 1-2-4-9-18-37-75.
The classification tree is constructed to classify all the attributes to get the distribution of the 24 node values of the travelling path from 30,000 records and the classification tree is given below.

Figure 4.4 Distribution of the node count values

Figure 4.5 Classification tree
To the given algorithm, a large set of data was generated to test the accuracy of least processing time by this method. Figure 4.5 depicts the classification tree for the medical database considered. The classification test were carried out with different amount of data supplied to the program ranging from 20,000 to 1,00,000 and classification is done.

Figure 4.6 Scalability of Processing time in MMDBM and SLIQ using Java

It is observed that with the increase in the number of records, the prediction of accuracy is improved. Table 3.3 shows the prediction rules obtained from the database. Figure 4.4 shows the class distribution at each of the nodes count values. The results found in this paper is compared with the processing time of (SLIQ and MMDBM) using implementation of java programming is given below the Figure 4.6.

Both the algorithms (Quick sort and Radix sort) are a large set of dataset which have been generated to test the precision minimum processing time by CPU and GPU Computing. Figure 4.7 and Figure 4.8 demonstrates the classification tree of the medical database which measures the node count of the values. The classification test were accepted with different amounts of data provided to the program extending from 10,000 to 1,00,000 and the classification
is completed. It is perceived that with the rise in the number of records, the estimate of accuracy is upgraded. Comparative analysis of the processing time for Quick sort (CPU and GPU) with Radix sort (CPU and GPU) algorithms, is shown in Figure 4.7 and 4.8.

![Figure 4.7 Scalability of CPU and GPU time with Quick sort Algorithm](image1)

![Figure 4.8 Scalability of CPU and GPU time with Radix sort Algorithm](image2)
This test was carried out on a Microsoft windows 7 together with CUDA version 5.0. The hardware platform consists of an Intel core i5-245M CPU 2.50 GHz and 6 GB RAM. Table 4.5 summarized GPU characteristic used in the experiments. The GPU is used in NVIDIA GT525M card with 4095M.

Table 4.5 Characteristics of Geforce GT525M card

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Core</td>
<td>96</td>
</tr>
<tr>
<td>Graphics Clock</td>
<td>475 MH(_z)</td>
</tr>
<tr>
<td>Process Clock</td>
<td>950 MH(_z)</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>900 MH(_z)</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>128-it</td>
</tr>
<tr>
<td>Total available graphics</td>
<td>4095 MB</td>
</tr>
<tr>
<td>Dedicated video memory</td>
<td>2048 MB DDR3</td>
</tr>
<tr>
<td>Shared system memory</td>
<td>2047 MB</td>
</tr>
</tbody>
</table>

4.9 CONCLUSION

Classification is one of the major tasks in data mining. A new classifier called Mixed Mode Database Miner (MMDBM) has been programmed in Java with Quick sort and Radix sort algorithms and has been tested using Medical database and also the same algorithms have been programmed in GPU Computing (CUDA). The algorithm can handle huge amount of datasets with large amount of Attributes. GPUs Quick sort and Radix sort algorithm provides exceptional scalability with the Medical data sets that has been taken for analysis and testing. The main Results have been taken into consideration and verified for accuracy and the program code is provided for CPU and GPU Computing. We have discussed an efficient parallel quick sort and Radix sort algorithms in GPU and results from the Comparison of computational acceleration ratio (speed-up) and efficiency of processing time of CPU and GPU Computing in MMDBM Classifier. The main results are used to compare the classifier with an
existing CPU-quick sort and radix sort for the MMDBM classifier and GPU-quick sort and radix sort algorithms provide rapid and exact results with minimum execution time and supports real time applications.