

國立交通大學

資訊科學與工程研究所

碩士論文



CUDT: 以 CUDA 為基礎之決策樹演算法

CUDT: A CUDA Based Decision Tree Algorithm

研究生：邱俊傑

指導教授：袁賢銘 教授

中華民國一百年六月

CUDT: 以 CUDA 為基礎之決策樹演算法
CUDT: A CUDA Based Decision Tree Algorithm

研究生：邱俊傑

Student : Da-Ming Chang

指導教授：袁賢銘

Advisor : Shyan-Ming Yuan

國立交通大學
資訊科學與工程研究所
碩士論文



Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年六月

CUDT: 以 CUDA 為基礎之決策樹演算法

學生：邱俊傑

指導教授：袁賢銘

國立交通大學資訊科學與工程研究所

摘要

分類(classification)在機器學習(Machine Learning)和資料探勘(Data mining)中是一個很重要的議題。其中，決策樹被廣為運用在這個領域中，然而在現實生活中，資料多為高維度且資料量非常鉅量，在大量的資料下，整個決策樹建立的時間大多消耗在計算上，也就是這是一個運算密集的問題，也因此相當多的研究專注於加速分類模型的建立。

圖形處理器(GPU)是專門為處理圖形而設計的，因為影像的處理具高度平行化的特性，造就 GPGPU 的產生；許多研究專注於使用 GPU 來處理非圖形處理的大量運算，其加速的效果非常驚人，因此也有著極高的性價比。而 CUDA(Compute Unified Device Architecture)即為 NVIDIA 所提出的 GPGPU 的方案。

本論文基於 NVIDIA's CUDA 提出一個新的決策樹的演算法，在此架構中 CPU 負責流程處理，而 GPU 負責處理大量資料的運算。我們與著名的資料探勘軟體 Weka 和 SPRINT 來做比較，結果顯示我們的 CUDT 比起 Weka 在效能上有 6~5x 倍的加速，較大的資料上比起 SPRINT 我們在效能上有 18 倍的加速。

關鍵詞：GPGPU、CUDA、Decision Tree、Classification

CUDT: A CUDA Based Decision Tree Algorithm

Student: Chun-Chieh Chiu

Advisor: Shyan-Ming Yuan

Institute of Computer Science and Engineering

National Chiao Tung University

Abstract

Classification is an important issue both in Machine Learning and Data Mining. Decision tree is one of the famous classification models. In the reality case, the dimension of data is high and the data size is huge. Building a decision in large data base cost much time in computation. It is a computationally expensive problem.

GPU is a special design processor of graphic. The highly parallel features of graphic processing made today's GPU architecture. GPGPU means use GPU to solve non-graphic problems which need amounts of computation power. Since the high performance and capacity/price ratio, many researches use GPU to process lots computation. Compute Unified Device Architecture (CUDA) is a GPGPU solution provided by NVIDIA.

This paper provides a new parallel decision tree algorithm base on CUDA. The algorithm parallel computes building phase of decision tree. In our system, CPU is responsible for flow control and GPU is responsible for computation. We compare our system to the Weka-j48 algorithm. The result shows out system is 6~5x times faster than Weka-j48. Compare with SPRINT on large data set, our CUDT has about 18 times speedup.

關鍵詞：GPGPU、CUDA、Decision Tree、Classification

Acknowledgements

碩士班的生活對我而言是一段難忘的回憶。與文峰、智維和瑞廷在實驗室內熬夜寫作業趕論文，分享彼此的想法，讓我不只在專業技能上有所提升，也培養了與大家的革命情感。與國亨、家鋒討論論文的靈感和設計，他們的意見和指導讓我獲益匪淺。學弟妹聖凱、珮瑜、冠穎、紘維的加入，讓生活有更多的調劑和歡笑。感謝指導教授袁賢銘讓我在研究上能盡情發揮自己的想法，適時的給予我指導讓我能順利完成這篇論文。最後，我要感謝一直關心我的家人和朋友，因為你們才有今日的我，也才有這篇論文。

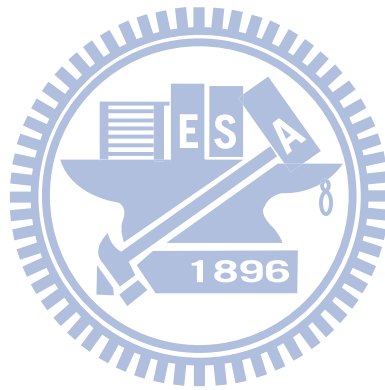


Table of Contents

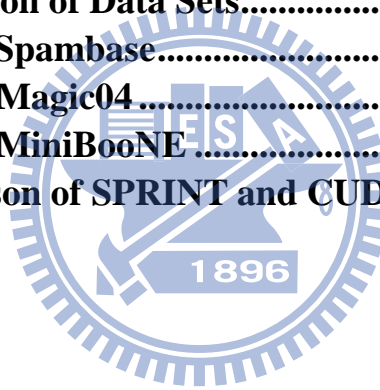
摘要.....	I
Abstract.....	II
Acknowledgements	IV
Table of Contents	V
List of Figures.....	VI
List of Tables.....	VII
Chapter 1 Introduction.....	1
1.1 Motivation	1
1.2 Objectives.....	2
1.3 Outline of the Thesis	2
Chapter 2 Background and Related Work.....	3
2.1 Decision Tree.....	3
2.2 GPU	4
2.3 Compute Unified Device Architecture (CUDA)	6
2.4 Prefix-sum	8
2.5 Related Work.....	8
Chapter 3 System Architecture.....	14
3.1 System Overview	14
3.2 System Flowchart	16
3.3 System Components	18
Chapter 4 Evaluation.....	30
4.1 Evaluation Environment.....	30
4.2 Data Sets.....	31
4.3 Evaluation of System.....	32
4.4 Evaluation of Each Level.....	36
Chapter 5 Conclusion	43
Chapter 6 Future Work.....	44
Reference.....	45

List of Figures

Figure 1 Example of Decision Tree.....	3
Figure 2 Floating-Point Operations per Second for the CPU and GPU [1].	5
Figure 3 The GPU Devotes More Transistors to Data Processing [1].	5
Figure 4 Architecture of CUDA	7
Figure 6 Example of Attribute Lists [6].	10
Figure 7 Example of Splitting Attribute List of SPRINT [6].	11
Figure 8 C_{above} and C_{below} of SPRINT [6].	12
Figure 9 Count Matrix of SPRINT [6].	12
Figure 10 Jobs Dispatch of CUDT.	15
Figure 11 Flowchart of CUDT	15
Figure 12 Flowchart of Building Phase.....	18
Figure 13 Example of Compact	23
Figure 14 Example of Reduce	23
Figure 15 Example of Split Attribute Lists.....	29
Figure 16 Speedup of each Component (1).....	35
Figure 17 Speedup of each Component (2).....	35
Figure 18 Speedup of building classifier	36
Figure 19 Execution Time of each Level on Spambase	38
Figure 20 Speedup of Level of Spambase	38
Figure 21 Execution Time of each Level on Magic04.....	39
Figure 22 Speedup of Level of Magic04.....	39
Figure 23 Execution Time of each Level on MiniBooNE	40
Figure 24 Speedup of Level of MiniBooNE	40
Figure 25 Active Data Size v.s. Node Size on Spambase	41
Figure 26 Active Data Size v.s. Node Size on Magic04	41
Figure 27 Active Data Size v.s. Node Size on MiniBooNE	42

List of Tables

Table 1 Traditional Algorithm of Decision Tree [5].	4
Table 2 Information of each Memory of CUDA 錯誤! 尚未定義書籤。	
Table 3 Definition of Prefix-sum (Scan) [8][13].	8
Table 4 Algorithm of Compact.	22
Table 5 Algorithm of Reduce	24
Table 6 Algorithm of Finding Split Points	25
Table 7 Algorithm of Split Attribute Lists	27
Table 8 Algorithm of Partition.	28
Table 9 Evaluation Environment.	31
Table 10 Information of Data Sets.	31
Table 11 Result of Spambase.	33
Table 12 Result of Magic04.	33
Table 13 Result of MiniBooNE	34
Table 14 Comparison of SPRINT and CUDT	34



Chapter 1 Introduction

1.1 Motivation

Machine learning is a theory which concerned with constructing computer systems with the ability to learn by either experience or by studying instructions. This capability to learn results in a system that can continuously self improve and thereby offer increased efficiency and effectiveness.

Machine learning algorithm is widely used in many domains. Usual tasks of machine learning algorithm includes classification, prediction, clustering and rule finding. They are used in computer graphic, image processing, intrusion detecting.

Many machine learning has a common characteristic. They need scan data bases repeated to find patterns of the data. The process of analyzing high dimension and lots of data requires huge computation power and storage. Many process needs several hours even several days in reality data bases. The performance of machine learning algorithm is endless.

Recently, the General-purpose computing on graphics processing units (GPGPU) has become popular since the its highly parallelization and powerful computing ability of float point.

Some documents shows the computing power of GPUs can now vastly exceed a CPU [1][2].

More and more non-graphic applications which needed amounts of computation are employed on GPU. Since GPGPU became a tendency, NVIDIA provides a platform of GPGPU which called Compute Unified Device Architecture. Many applications and researches of machine learning use CUDA as their GPGPU platform. Such as SVM, K-NN, K-means has CUDA version.

The decision tree learning algorithm is a very famous learning model in classification. Many researches are focus on improving performance of decision tree [5][6][7]. However, those algorithms are base on a distributed system. The cost of those devices is high.

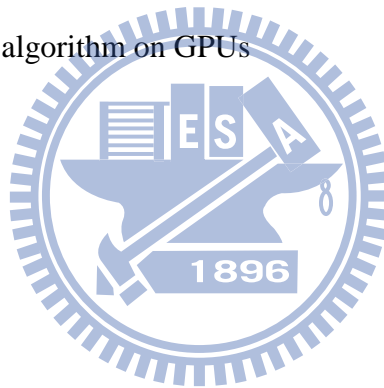
In this paper, we will present detail in a parallelized decision tree algorithm base on CUDA.

The goal is to evaluate the performance of the algorithm in terms of execution time, compared to CPU version decision tree.

1.2 Objectives

The following are our objectives:

1. A parallel decision tree algorithm on GPUs
2. High performance
3. Binary tree
4. Binary classification



In our experience of CUDA programming, binary classification and binary branch is suiting for CUDA architecture. Therefore, our decision tree is a binary tree with binary classification.

1.3 Outline of the Thesis

The rest of the paper is organized as follows. Chapter 2 is background and related works. We will present the CUDA architecture and some important parallel primitives. The related works shows the recently researches of decision tree on GPUs. Chapter 3 is our system architecture. We will describe our system in detail in this chapter. Chapter 4 is the evaluation of our algorithm. The last part of the paper is the conclusion and future work.

Chapter 2 Background and Related Work

2.1 Decision Tree

Decision tree is a supervised learning algorithm commonly used in machine learning and data mining. In classification, the goal of decision tree is evaluating a data which contains several values called attributes to predict the class of data. An example is shown in Figure 1.

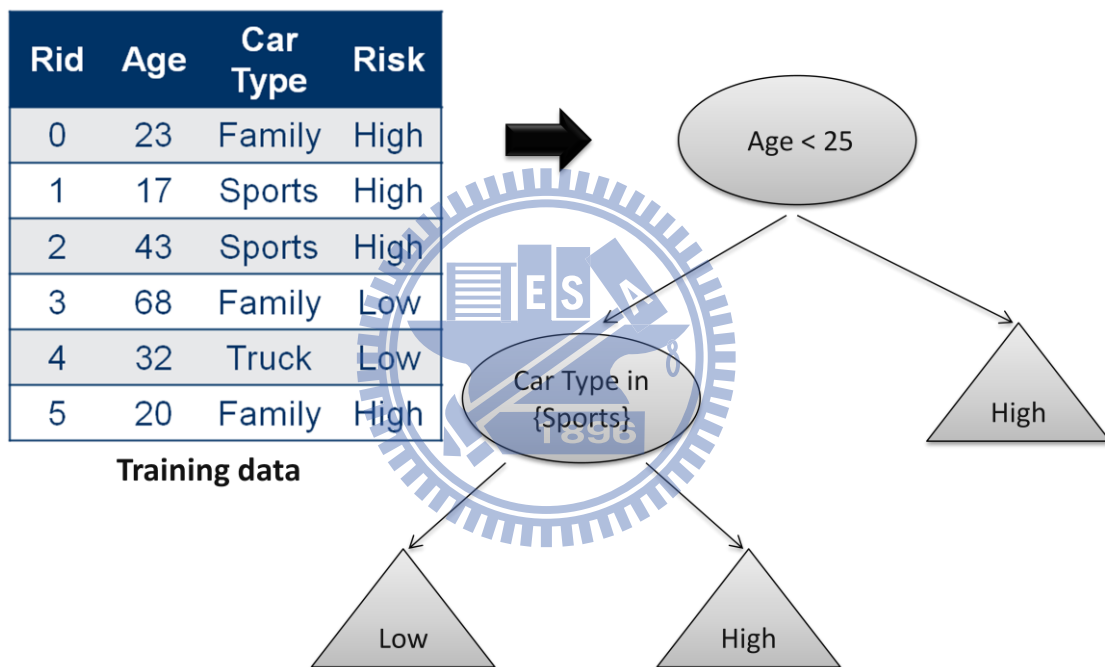


Figure 1 Example of Decision Tree

There are two kind of node in the tree. An internal node (oval-shaped) denotes a classified rule. The rule decides the path of tested data. A leaf node (triangle) denotes a result of classification. The leaf decides the target value of classification.

The learning process of tree is shown on Table 1 [5]. The algorithm needs a set of data called *Train Data*. Each record contains several values which called feature to describe the data. The

building process would partition the train data into several subspaces in each internal node. For each subspace, treat as a new training set and recursive called process. The partition will be terminated if all data on the node are of the same class and the node would become a leaf.

<pre>Partition (Data S) if (all points in S are of the same class) then return; For each attribute A do evaluate splits on attribute A; Use best split found to partition S into S_1 and S_2; Partition(S_1); Partition(S_2); Initial call: Partition(TrainData)</pre>

Table 1 Traditional Algorithm of Decision Tree [5].

2.2 GPU

A GPU is a processor that processes 3D graphics rendering. The GPUs deliver very high performance of float-points processing since is that it is a highly parallel machine comprised by many cores. GPUs keep these processors busy by juggling thousands of parallel computational threads. In theory, GPUs are capable of performing any computation that can be mapped to the stream computing model. This model has been exploited for many algorithms such as ray-tracing, K-means, matrix multiplication, K-NN, SVM and the other applications. The comparison of computation power of float-point between CPU and GPU was depicted as Figure 2.

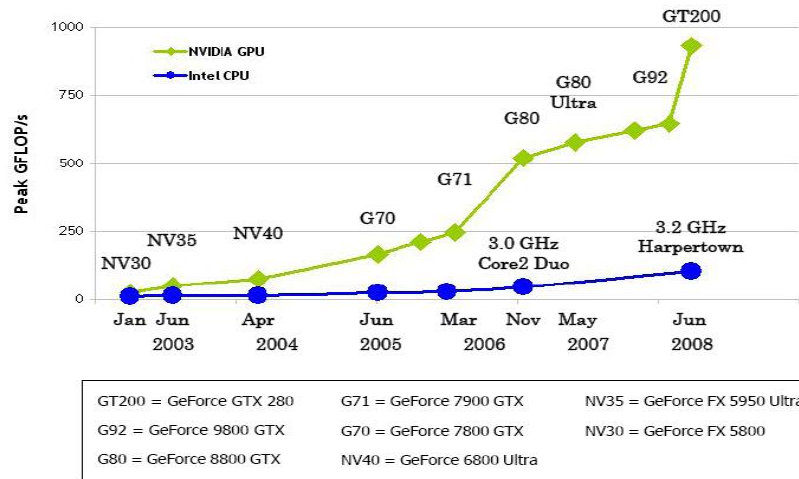


Figure 2 Floating-Point Operations per Second for the CPU and GPU [1].

The primary difference between CPU and GPU is that the GPU is specialized for rendering graphics, highly parallel computations. As illustrated by Figure 3, GPU devotes more transistors to data processing rather than data caching and flow control. GPU is more efficiently than CPU if a problem needs lots of computation and shit for parallel.

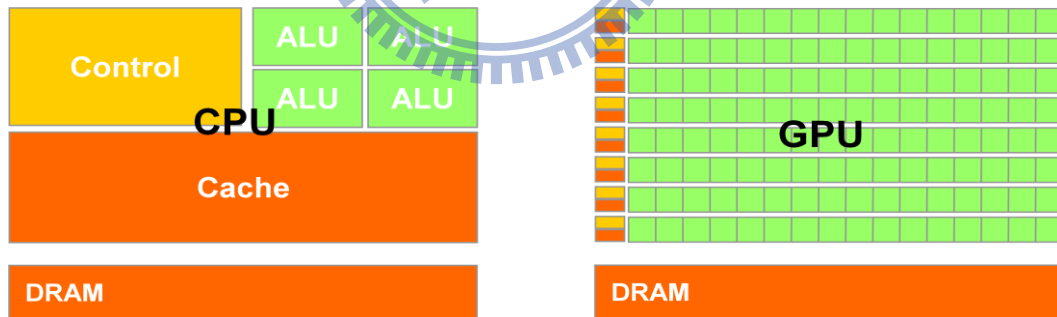


Figure 3 The GPU Devotes More Transistors to Data Processing [1].

2.3 Compute Unified Device Architecture (CUDA)

CUDA is a general purpose parallel computing architecture including a new parallel programming model and an instruction set architecture. The CUDA programming model provides a C extended language for developers. The program will then run in thousands or millions of parallel invocations, or threads on device. The source files include a mix of host code runs on CPU and device code which runs on GPUs. When running a CUDA program, developers simply run the program on the host CPU. The CUDA driver automatically loads and executes the device programs on the GPU.

In the CUDA programming model, the GPU is treated as a co-processor onto which an application running on a CPU can launch a massively parallel compute kernel. This is called massively threaded architecture. The CUDA function needs a configuration of the size of thread to be launch. The configuration set up a grid. A grid composes several blocks. The threads in the same block would have a serial ID called thread index. It can be used to help divide up work among the threads.

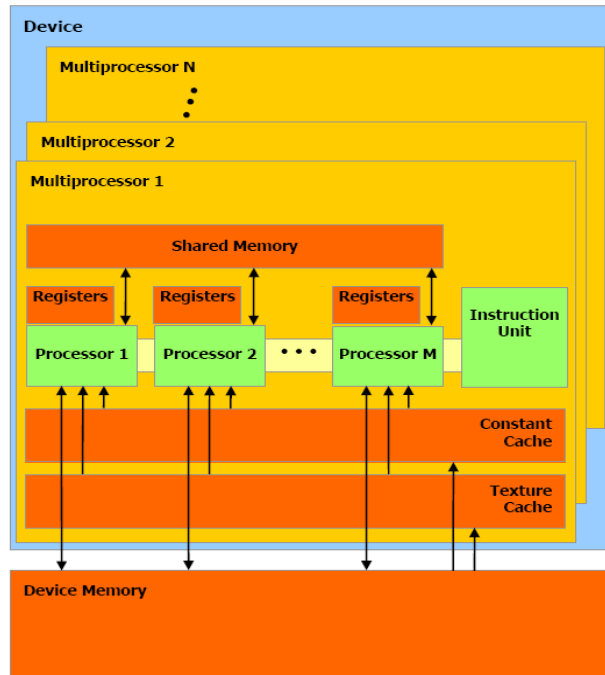


Figure 4 Architecture of CUDA

Threads running on the GPU in the CUDA programming model have access to several memory regions including on chip memory, registers, shared memory, constant cache, texture cache, and off chip memory, global memory constant memory and texture memory. Figure 4 shows each multi-processor can access on/off chip memory of the several types as following:

Device storages consist of constant memory, texture memory, global memory, local memory and registers. Constant memory and texture memory are read-only caches of device memory and must be allocated before calling device functions. They are rare on device. The on chip caches, constant cache and texture cache, can have significant performance increasing of optimization of CUDA program. Global memory is the main storage of CUDA. It is a slower and biggest memory on device. Threads can read/write the global memory directly, but there is no cache supported for global memory. Register is the fastest storage of CUDA. The number of registers of a multi-processor would restrict the maximum number of concurrent executed threads. Shared memory is visible to threads which are runs on the same block. It is

a highly speed storage. The local memory space is not cached, the usage of local memory should be avoided since it is very slow. The overhead of accessing local memory is large.

2.4 Prefix-sum

Prefix-sum, as known as scan, is a very important building block of parallel algorithm. Many applications such as sorting, lexically compare strings, evaluated polynomial can be implemented by scan [13]. The element which is prefix-sum will be the result of operated all elements before current element. The following is the definition of prefix-sum:

<p>Given an array A of n elements</p> $A = [a_0, a_1, \dots, a_{n-1}]$ <p>Given a binary operator \odot</p> <p>Given I as identify of \odot</p> $\text{Scan}(A) = [I, a_0, (a_0 + a_1), \dots, (a_0 + a_1 + \dots + a_{n-2})]$

Table 2 Definition of Prefix-sum (Scan) [8][13].

Prefix-sum makes no sense in sequence algorithm but it is very important in parallel algorithms. Both CUDA SDK and CUDPP implement scan as an important library. The CUDPP Sorting algorithm is a high performance CUDA radix sort. The scan is the backbone of CUDPP sort, each round of sorting is building on prefix-sum [4]. Our system also uses the parallel prefix-sum of CUDPP in many components.

2.5 Related Work

We introduce two researches related with our research. The first one is a classical parallel decision tree algorithm “SPRINT”. The second is a random forests base on CUDA.

2.5.1 SPRINT: A Scalable Parallel Classifier for Data Mining

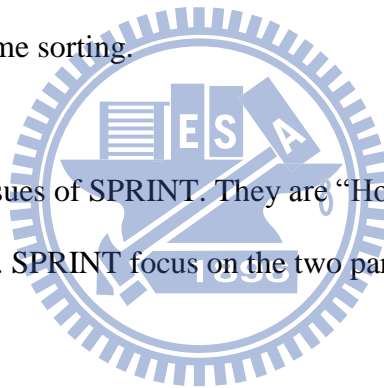
SPRINT is a classical algorithm of parallel decision tree. The algorithm has several targets as follow:

1. Reduced the time of building a decision tree
2. Eliminated the barrier of memory size

In order to achieve the two targets, they design a special data structure called attribute lists.

The function of attribute list is isolated each attributes. It makes independence of each attribute. There are two advantages of the data structure. First, only need one-time sort for each attribute since the relationship between data and each feature is eliminated. The attribute can keep the order by one-time sorting.

There are two importance issues of SPRINT. They are “How to finding a split point” and “How to splitting attributes”. SPRINT focus on the two parts of computation. The building process is in sequence.



2.5.1.1 Data Structure

The traditional decision tree algorithms need repeat sorting for continuous attributes. The sorting cause lots of unnecessary execution time. Many researches devoted in reducing the overhead of the sorting. SPRINT is a representative of such algorithms. In SPRINT, it only need a pass sorting for continuous attributes since its special data structures called“attribute list”. Since the attribute lists separate each attribute, SPRINT need other histograms to calculate the split criteria.

2.5.1.1.1 Attributes Lists

The Figure 6 shows an example of an attribute list. The left one is an attribute list of continuous attribute, and the right one is an example of categorical attribute. An attribute list is composed of three arrays. The First array is the attribute value, the second is the class label of the record and the third is the index of record. It is obvious that each attribute list is independent, so we can sort each continuous attribute one time and doesn't need extra sorting. In the split stage, each list will be split into two disjoint subspaces. Figure 7 shows an example of slitting. This mechanism reduces the overhead of sorting but increase the overhead in splitting the attribute lists. However, the new overhead is smaller compare to repeat sorting.

Age	Class	rid	Car Type	Class	rid
17	High	1	family	High	0
20	High	5	sports	High	1
23	High	0	sports	High	2
32	Low	4	family	Low	3
43	High	2	truck	Low	4
68	Low	3	family	High	5

Figure 5 Example of Attribute Lists [6].

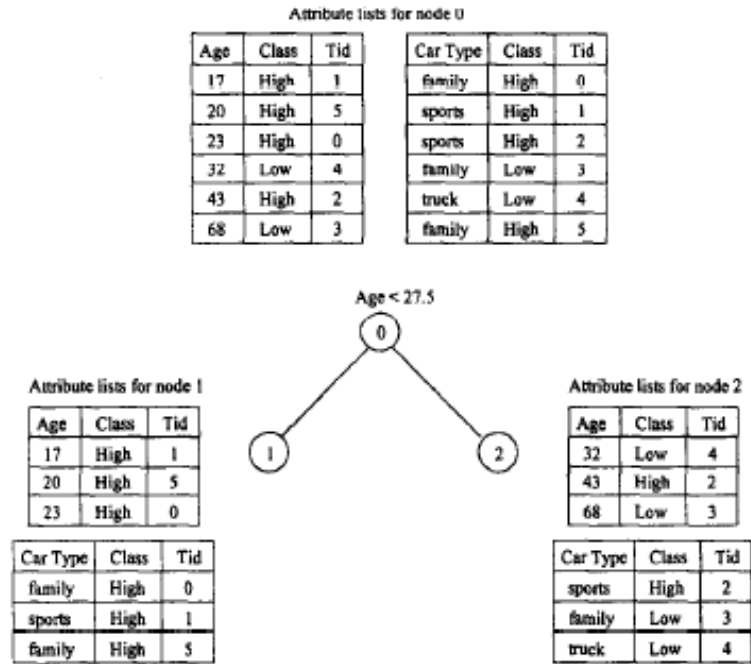


Figure 6 Example of Splitting Attribute List of SPRINT [6].

2.5.1.2 Finding Split point

In order to finding the best split point, the algorithm needs to calculate the criteria of splitting. The classes distributed information is used to calculating the split criteria. SPRINT has two different approaches for each attributes. For continuous attribute, SPRINT uses two histograms, denoted as C_{below} and C_{above} , to capture the class distribution of the attribute records at a given node. The Figure 8 shows an example of the two histograms. C_{below} records the sum of each class number before current data and C_{above} records the sum of each class number after current data.

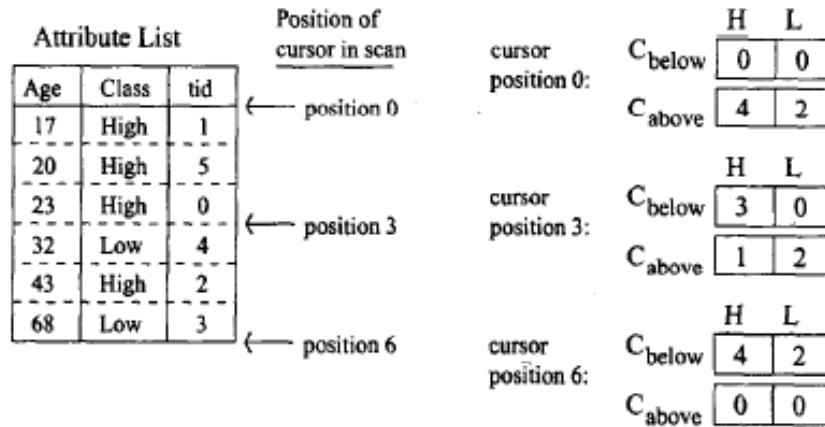


Figure 7 C_{above} and C_{below} of SPRINT [6].

For categorical attribute, SPRINT uses a histogram called “count matrix”. Figure 9 shows an example of count matrix. Each entry of counting matrix records a class distributed of a different value of the attribute. After finishing the calculation of class distribution, we have all information of calculating split criteria.

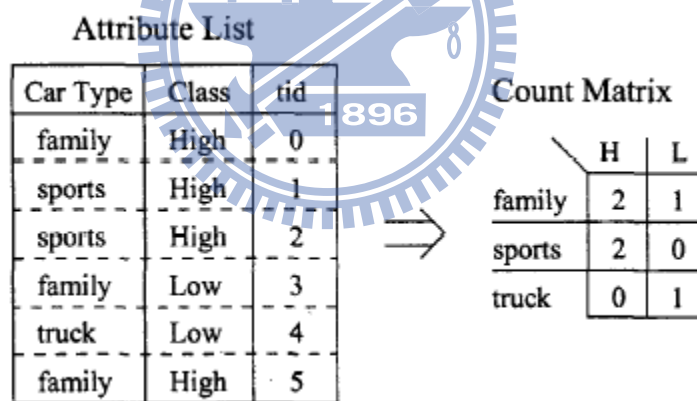


Figure 8 Count Matrix of SPRINT [6].

In parallel version SPRINT, it partitions the attribute lists into several subspace with the same size. Each processor calculates the local class distributed and exchange with each other to get the global class distributed. After getting the global class distributed, each processor calculates the local split criteria of each possible split points. For continuous attribute, the

possible split points of an attribute are each different value points. For categorical attribute, the number of possible split points is equal to the number of different value of the attribute.

After finishing the local split criteria, each processor finds the local best point. In order to get the global best split point, processors communicate with each other to find the best split points.

2.5.1.2 Splitting Attribute Lists

SPRINT uses attribute lists for each attributes so it needs special mechanism for partition data. Since each attribute lists are partition into several subspaces to processors, different attribute lists may has different data on the processor. In order to keep the sorted value in order, SPRINT uses a global hash table to records the location of data on child nodes. All processors would communicate with each other to getting the global hash table. After finishing the hash table, each processor can split the local attribute lists into the attribute of child nodes.

2.5.2 Random Forests for CUDA GPUs

Random forest was first introduced by Len Breiman [12]. It is comprised by many decision trees. Each tree uses a randomized features selected. The train data is used a sampling process. Each tree uses the same data set to generate the training data with replacement. This has proven to be effective for large data sets with missing attributes values [12].

The CUDA implemented random forests parallelize both building and classification. They use a CUDA thread to build a tree. All trees are built in parallel to each other but the trees themselves are built sequentially. However, this approach is not suit for large data sets if the data set is bigger than GPU memory size.

Chapter 3 System Architecture

In the traditional decision tree building algorithm, continuous attributes cost lots of the execution time since it need repeat sorting when calculated the split criteria. In order to reduce the sorting time, many researches are focus on reduce the times of sorting. SPRINT is one of the classical algorithms which parallelize builds a decision tree. SPRINT aim at reducing the building time and eliminating the restriction of memory size. Since above, SPRINT uses special data structure and mechanism to solve the problem. Though our algorithm aims at the in-memory problem, we learn from SPRINT because it provides some features are suit for CUDA's architecture. This chapter is organized as follows. Section 3.1 is an overview of our system. Section 3.2 is a flowchart of the system. We will describe in detail in section 3.3.

3.1 System Overview




Figure 10 shows an overview of our system. The blue part is the jobs of CPU and the green part is the jobs of GPU. The principle of our design is obvious that the CPU is responsible for the flow control, I/O handle and communication with graphic device, the GPU focuses on the computing intensive jobs. For instance, create attribute lists needs sorting each attribute lists, we put this part into device; the time of calculate split criteria and split attribute lists are in proportion to the data size and attribute number.

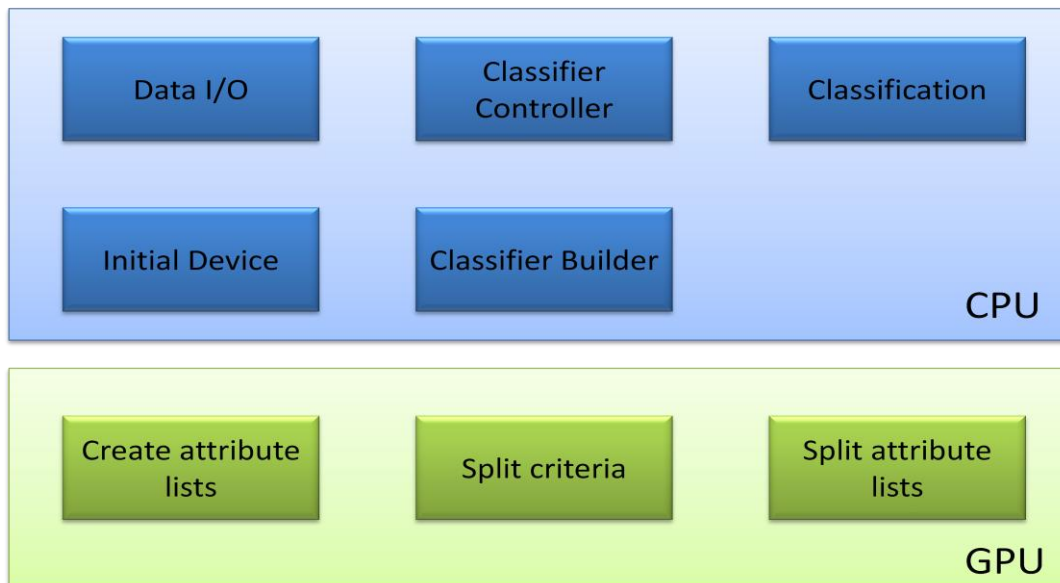


Figure 9 Jobs Dispatch of CUDT.

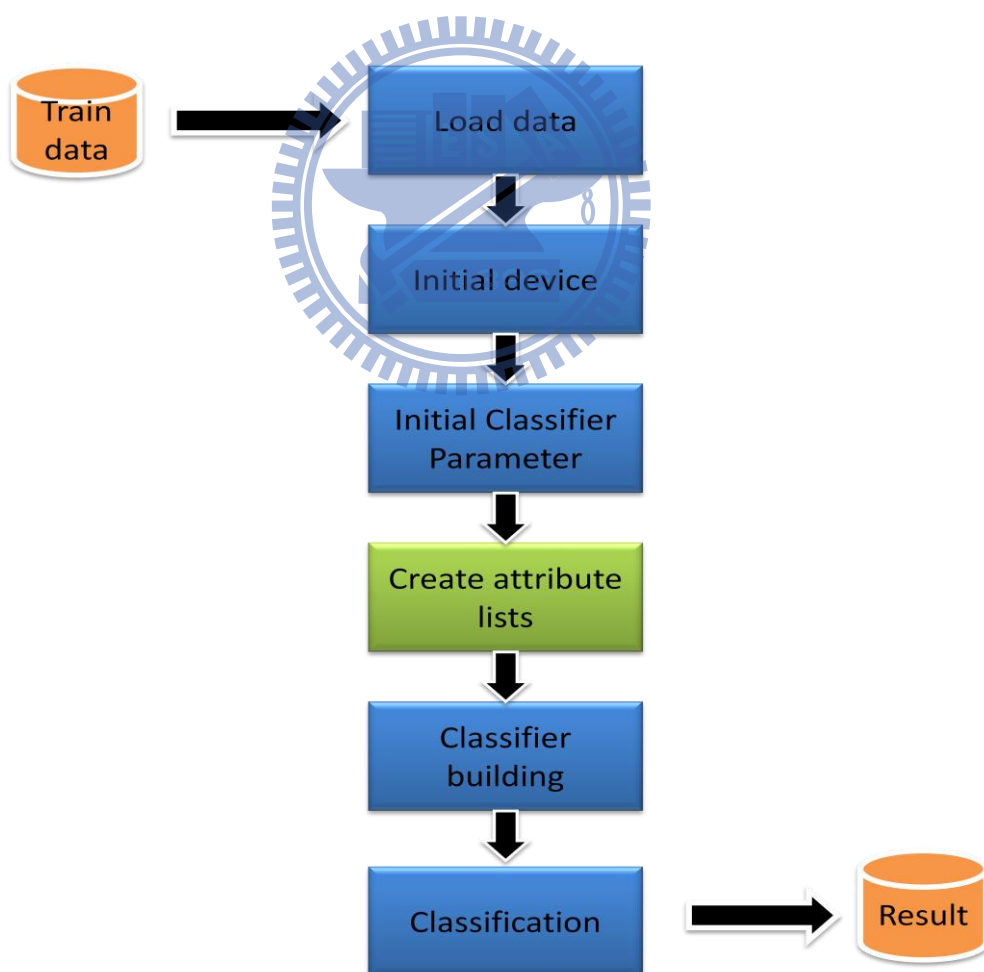


Figure 10 Flowchart of CUDT

3.2 System Flowchart

Comparing to the CudaRF, both training phase and the classification phase are parallelized in CudaRF. This approach shows a nice performance on smaller data sets. However, there is no guarantee that the device memory can store all trees. The algorithm needs backup the trees in host memory if the trees sizes are over the device memory. This will increase the overhead in memory between host and device.

In our CUDT, we only focus on the phase of building trees. It's more accurate to say that we focus on parallel processing the computing of splitting nodes. Although reducing the parallelism in building multiple trees, this policy increases the scalability and performance in huge data sets.

The Figure 11 shows a flowchart of system. The following steps illustrate the main execution steps in our system.

1. Training and testing data are loaded to host memory from disk.
2. Initialization of the device includes query device information, allocation memory space and copy training data into device.
3. In this step, the system will setup some parameters from user. For instance, the minimum numbers of data of a leaf, the maximum depth of the classifier.
4. Create attribute lists in device. We will move each attribute to correspond position. After finishing the data movement, we would sort each attribute lists in device.
5. The most important step of the system. Instead of using the recursive model of decision tree building algorithm, we use an iterative breadth first scheme for our system. Host plays a role of a manager. It is in charge of working flow of system. The computation intensive problem is send to device. Figure 12 shows a flowchart of building classifier.

6. The classification is performed on host. In other words, the process of classification is in sequence.
7. The results are presented on host.

We will now describe in more detail of the flowchart of building classifier. The system will loop until all data has belonged to leaf. For a segment of data, the system would check if all data of this segment has same class label, positive and negative in our system. Make a leaf node if all class of data is the same or processing the finding a split point of the segment. A leaf node denotes a result of classification. The data would be classified as the class of leaf node if it stops at this node in classifying. After find a candidate split point, we need to split the attribute list and make an internal node. An internal node could be thought as a rule which decides the path to classify the data.

In the next section, we will describe in more detail of each system components, for instance, how the attribute list is created, i.e., step4, and how finding a candidate split point and splitting attribute lists in setp5. We also shows how apply those parallel primitives and how we employ the computation power of GPU to our system.

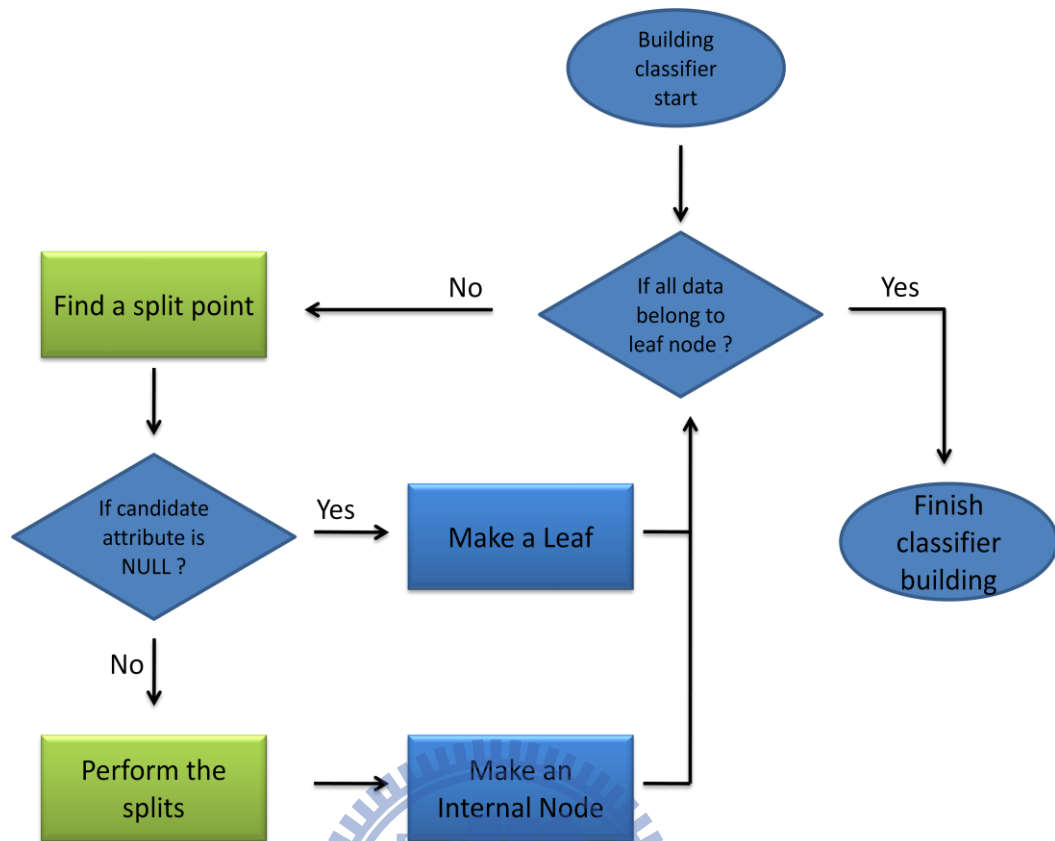


Figure 11 Flowchart of Building Phase

3.3 System Components

3.3.1 Load Data & Initial Device

The execution starts with the host reading input data from disk. After loading data from disk, the system will allocated the space of the device memory to store data. The allocation includes entire training data, the space of attribute lists and some internal buffer inside of the device.

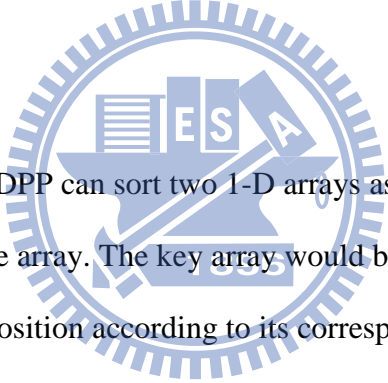
3.3.2 Initial Classifier Parameters

After finishing the allocation of device memory, we set the parameters of our CUDT. The parameters are user defined. For example, the minimum size of data of a leaf, the maximum

depth of the decision tree, the type of classification evaluation. (ex: cross validation, sampling...etc)

3.3.3 Create Attribute Lists

There are two parts of creating attribute lists. The first is moving the data to its corresponding list. After finishing the data movement, we need sorting each attribute lists. There is a well-known CUDA library which called CUDPP (CUDA Data Parallel Primitives Library). CUDPP offers a serial efficient library to developers. Several important algorithms are implemented in those libraries, for instance parallel prefix sum and parallel sorting. However the CUDPP has a wonderful parallel radix sort algorithm, the sorting algorithm is not suit for our system.



The sorting algorithm 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 be sorted and the element of value array would be changed its position according to its corresponding key element. It's called a key value pair sorting. The sorting algorithm of CUDPP only supports a key value pair sorting, but we have two values to one key (key is the *attribute value* field, *rid* and *class label* are values).

According to above, we modify the CUDPP sorting algorithm into one key to two values. In order to get the best performance, we modified the sorting algorithm from the CTA level to public interface level [4].

3.3.4 Classifier Building

There are two important functions of building classifier. The first is “Finding Split point” which performs finding the candidate split point and attribute. The second is “Split Attribute Lists” which would be performed after finding a valid split point.

3.3.4.1 Finding a split point

While growing the tree, the goal at each node is finding the “best” attribute and split point that “best” divided the train data. The value of a split point depends on how well it separates the classes distribution. There are many split criteria has been proposed in the pasts. We use *gini index* [14] as splitting criteria of our CUDT.

At first, let’s consider how the sequential algorithm works. In the sequential version, the process need scan an attribute list to finish a class distributed table. After finishing the table, the process has all information to calculate the *gini index* and find the best split point of this attribute. However, it only processes one attribute. We need calculate all attribute lists and find the best among them.

In our CUDT, we find the best split point in one pass; we process all attribute lists in one pass. The Table 6 shows the algorithm of finding a best split point. First, the system needs to record the class distribution into below table and save the number of total positive class. For continuous attribute, the candidate split points are mid-points between every two consecutive attribute values. It is obvious that there are many redundant elements of the below table, so the system need to remove unnecessary data from the histogram. The processing is called compact.

Table 4 shows the algorithm of compact. The compact needs a flag array and other arrays (payloads) as input. The value of flag element is “0” and “1”. “0” means the correlative payloads are true elements. True elements should be reserved in final output. The algorithm first scans the flag array to get the positions of true elements. After getting the position, the threads with true elements would put the elements into their positions. Each thread loops several times to put all payloads into correct address. The Figure 13 shows an example of compact.

After getting valid split points of all attributes, the system will calculate the splitting criteria of each possible split points. Since the class distributed table has all class information of the data segment, we can calculate the *gini index* of each possible split points.

The final step of this algorithm is finding the best point from the possible splitting points. There are a parallel primitive called “Reduction”. A brief description of reduction is that many parallel threads generate a single result. The Figure 14 shows how reducing an array to finding a minimum value. We use the CUDPP prefix-sum library of CTA level to implement the reduction. The algorithm of reduce is described in detail in Table 4.

After finding the best split point, device will upload the information to host. After getting the data, host can setup the information of children of this node and splits the attribute lists.

Algorithm Compact

Input: A class distribution table C
 A flag array Flag (Records each possible splits point)
 An address array Addr (Records address of valid elements)

1. Declare buffer[]
2. **For each** element C[i] **do in parallel**
3. **If** (Flag[i] == 1)
4. buffer[Addr[i]] = C[i]
5. **For each** element buffer[i] **do in parallel**
6. C[i] = buffer[i]

Table 3 Algorithm of Compact



Figure 12 Example of Compact

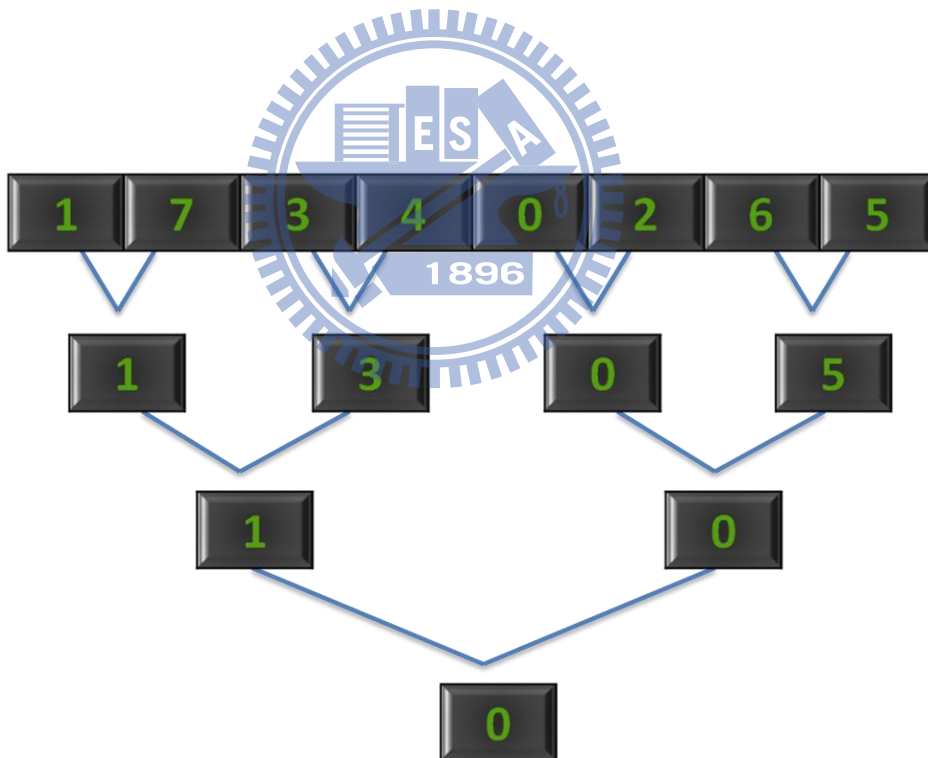


Figure 13 Example of Reduce

Algorithm Reduce

Input: An evaluated array E

Output: The minimum value of E

1. Declare $n = \text{sizeof}(E)$
2. Declare `buffer[]`
3. Declare `Min`
4. **While**($n > 1$)
5. **For each** segment E_i of E **do in parallel**
6. `buffer[i] = FindMinOf(E_i)`
7. $n = \text{sizeof}(\text{buffer})$
8. `E = buffer`
9. `Min = E[0]`
10. **Return** `Min`



Table 4 Algorithm of Reduce

Algorithm Finding Split Points

Input: A Set of attribute lists A which comprised by rid , $value$, $label$

Output: A winning attribute W
Index of split point X

1. **For each** attribute list A_i **do in parallel**
2. $C_i \leftarrow \text{Scan}(A_i.\text{label})$
3. **For each** data of A_i **do in parallel**
4. $\text{IsSplitPointFlag}_i[j] = (A_i.\text{value}_j \neq A_i.\text{value}_{j+1}) ? 1 : 0$
5. $\text{Addr}_i \leftarrow \text{Scan}(\text{IsSplitPointFlag}_i)$
6. $\text{Compact}(C_i, \text{IsSplitPointFlag}_i, \text{Addr}_i)$
7. $\text{value}_i \leftarrow \text{SplitCriteria}(C_i)$
8. **Reduce**(value)
9. **Return** W, X

Table 5 Algorithm of Finding Split Points

3.3.4.2 Splitting Attribute Lists

In traditional algorithms of building decision tree, the split attribute lists doesn't need extra work since all data are stored in order. It labels the split points of the data segment of this node. However, it's not working in our system since we partition an attribute as a single list. The different lists may have different data in the same position. Since above, we need a extra operation of splitting lists. Although the system need some extra executing time in splitting attribute lists, CUDA architecture is suit for binary split. It reduces the overhead caused by splitting.

The table 7 shows the algorithm of splitting the attribute lists. Partitioning the attribute list of the wining attribute is trivial. It just set the split index of wining split point of this attribute to node. Handling the winning attribute is very easy, however, we need a mapping between *rid* and sub-trees. The system uses a map table to store this mapping. A record is assigned to left partition if its value is smaller than the split point, or it will be assigned to right partition. After finishing split the winning attribute, the algorithm will keep work in the other attributes by the map table. As same with finding a split point, the system splits all attributes in one pass. A thread handles a record of an attribute list, finding the location of the record and store the result into a side array. We call a CUDPP parallel prefix-sum to calculate the side array. Moving all data of the segment into a buffer and performing a *partition* function. Figure 15 shows an example of splitting attribute lists.

The basic ideal of *partition* is partition an array into two disjoint subspaces. The Table 8 shows Partition algorithm in detail. The algorithm will partition the input data into two subspaces according to the flag array. The element will be assigned to left group if its flag is 0, or it would be assigned to right group. The algorithm first complements flag array and prefix sum it to get a false array. The total number of the false elements is recorded. The next step of

the algorithm is calculating the index of each element after partitioning. The index of an element is calculated above if it is a false element. If it is a true element, the index will equal to “the original index – above index + total number of false elements”. The final step is moving the elements to his position of the partition.

Algorithm Split attribute lists

Input: Wining attribute W

 An index of split point X

 A Set of attribute lists A

1. **For each** data d_j in W **do in parallel**
2. $\text{Flag}[j] = (\text{index}(d_j) > X) ? 0 : 1$
3. **For each** attribute list A_i **do in parallel**
4. **if** ($A_i \neq W$)
5. $\text{Partition}(A_i, \text{Flag})$
6. **Return**

Table 6 Algorithm of Split Attribute Lists

Algorithm Partition

Input: Target array A

A flag array with 0, 1 Flag

A Set of attribute lists A

1. **Declare** max = sizeof(A)
2. **Declare** buffer[max]
3. **Declare** FalseArray[max]
4. **Declare** TotalFalse
5. **Declare** Address[max]
6. **For each** element i in Flag[] **do in parallel**
7. buffer[i] = !Flag[i]
8. FalseArray[] <- Scan(buffer[])
9. TotalFalse = InverFlag[max] + FalseArray[max]
10. **For each** element i in buffer[] **do in parallel**
11. buffer[i] = i - FalseArray[i] + TotalFalse
12. Address[i] = (Flag[i] == 0) ? FalseArray[i] : buffer[i]
13. **For each** element i in A[] **do in parallel**
14. buffer[i] = A[i]
15. A[Address[i]] = buffer[i]

Table 7 Algorithm of Partition



Figure 14 Example of Split Attribute Lists

3.4.5 Classification

The tree is stored in host memory. The reason of constructing the classifier in host is the consideration of scalability. If the size of a tree is greater than the memory size of device, there are no ideal to maintain the tree in device memory. Our algorithm is designed for general cases. The system should be ease to scale in bigger data set. That is why we choice the policy. Since the tree is stored in host memory, the classification is processed in host side. Each data is tested in sequence.

Chapter 4 Evaluation

In our experiment evaluation, we focus on performance issue. Our goal is to design a high performance decision tree algorithm. Although accuracy is not our focal point, our algorithm also shows the accuracy of our system is acceptable. There are two objects which we compare with. First is a well-know open source data mining tool which called Weka [15]. Weka includes many data mining algorithms both classification and clustering, for example, decision tree, support vector machine, k-means...etc. We choice the Weka-j48 which is a C4.5 algorithm implemented in Java. Second one is our best optimized sequential SPRINT. Since our CUDT is one kind of SPRINT on GPU, we also compare each components of CUDT with SPRINT. This chapter is organized as following. The section 4.1 and 4.2 shows the environment and data set. The section 4.3 shows the result of evaluations.

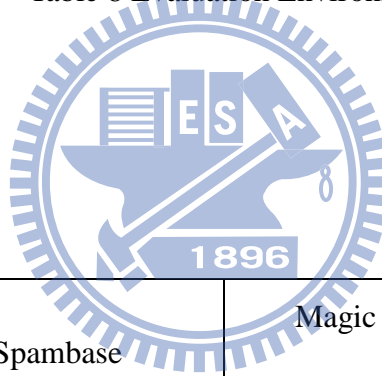
4.1 Evaluation Environment

We adopt Intel Core 2 Quad Q6600 and Geforce 9800GT for our computation platform. The configuration information is described as following. Our host is Intel Core 2 Quad Q6600 which has 4 cores. Each core has a clock rate with 2.4GHz. Our device is GeForce 9800GT which has 14 multiprocessors which called MPs. A MP has 8 CUDA Cores. There are 112 CUDA cores in total. The CUDA version is the version of the device driver. There are many new features in the newer version. The newest version of CUDA is 4.0RC. However, the features of CUDA 4.0 only impacts the recent generation of the GPU. There is no influence of our device. The compute capability means difference generation of CUDA GPUs [1][2]. Although our device is not as good as CPU, our system shows a good speed up on 9800GT.

	CPU	GPU
Device	Intel Core 2 Quad Q6600	GeForce 9800 GT (G92)
Number of cores	4	14 x 8
Clocks	2.4GHz	700 MHz
Memory	DDRII-800	DDRIII-900
Memory Size	2 GB	512 MB
OS	Ubuntu 9.04(Linux 32 bit)	
Compute Capability	--	1.1
CUDA Version	--	3.2

Table 8 Evaluation Environment

4.2 Data Sets



	Spambase	Magic Gamma Telescope	MiniBooNE particle identification
Number of Attribute	58	11	51
Number of Data	4601	19020	130065
Attribute Type	Continuous	Continuous	Continuous
Number of Class	2	2	2
Source	UCI	UCI	UCI

Table 9 Information of Data Sets

Table 10 shows the information of the data sets. There are three data sets in our evaluation.

The Spambase is a collection of mail data which has 57 continuous attributes of usually

features of spam mail. A categorical attribute denotes spam and non-spam. The number of data is 4601. The second data set is Magic Gamma Telescope. It is a physical data of high energy gamma particles. There are 19020 data of this data set. Ten continuous attributes of each record. A categorical attribute denotes the data into two classes. The final data is also a physical data. It is used to distinguish electron neutrinos (signal) from muon neutrinos (background). It has 51 attributes and 130065 data.

4.3 Evaluation of System

Table 11, 12, 13 shows the result of three algorithms. The table includes total cost time of building classifier, the accuracy of the classifier and the size of classifier. We use cross validation to evaluating the accuracy of our system. It means that we use all train data as test data. It shows the accuracy of our system is very close to Weka-j48 and the execution time is short than the both Weka and SPRINT. The tree sizes are the same of SPRINT and CUDT since we use the same criteria of splitting. Figure 18 is the speedup of building classifier.

In order to evaluating the speedup of each components of CUDT, we compare with SPRINT in detail. Table 14 shows the execution time of each component of CUDT and SPRINT.

Figure 16 shows the speedup of each component. The time of building a tree is sum of finding split point and splitting the attribute lists. Since the speedup of creating attribute lists is too higher than other components, we put it on Figure 17. Total time is the sum of building phase and creating attribute lists. We can see that our system is good for large data set.

The speedup of creating attribute lists is very good. It can achieve 14x times faster than SPRINT. However, the other components of our system are not as good as creating attribute

lists. The first reason is the size of tree. Not only the execution time but also the times of communication between CPU and GPU are proportional to the size of tree. Our system generates too many nodes that increase the execution time of building. The second reason is the size of data which really need calculated. The computation power of GPU is restricted by decreasing the size of active. The third reason is the increasing of nodes per level. More nodes makes more times of changing between CPU and GPU on each level. We will discuss those issues in the following sections.

Spambase	Weka – j48	SPRINT	CUDT
Cost Time	715 ms	1861.55 ms	124.78 ms
Accuracy	98.32%	97.82%	97.82%
Tree size	379	385	385
Leave Size	190	193	193

Table 10 Result of Spambase

Magic04	Weka – j48	SPRINT	CUDT
Cost Time	1350 ms	409.78 ms	257.72 ms
Accuracy	90.6%	93.54%	93.54%
Tree size	707	1579	1579
Leave Size	354	790	790

Table 11 Result of Magic04

MiniBooNE	Weka – j48	SPRINT	CUDT
Cost Time	141000 ms	47391 ms	2541.86 ms
Accuracy	98.52%	98.31%	98.31%
Tree size	6441	8127	8127
Leave Size	3221	4064	4064

Table 12 Result of MiniBooNE

Data Sets	Spambase		Magic04		MiniBooNE	
	SPRINT	CUDT	SPRINT	CUDT	SPRINT	CUDT
1. InitialDevice	--	1761.4	--	1686.25	--	2106.58
2. CreateAttributeLists	1719.71	37.27	57.33	9.4	29270.56	192.45
3. FindingSplitPoint	56.75	58.91	275.06	191.40	11022.46	1479.01
4. SplitAttributeLists	93.02	29.02	69.89	63.23	2973.29	726.78
5. Building phase (3+4)	141.83	87.51	352.45	248.31	18120.47	2349.40
6. Total Time (2+3+4)	1861.55	124.78	409.78	257.72	47391.04	2541.86

Table 13 Comparison of SPRINT and CUDT

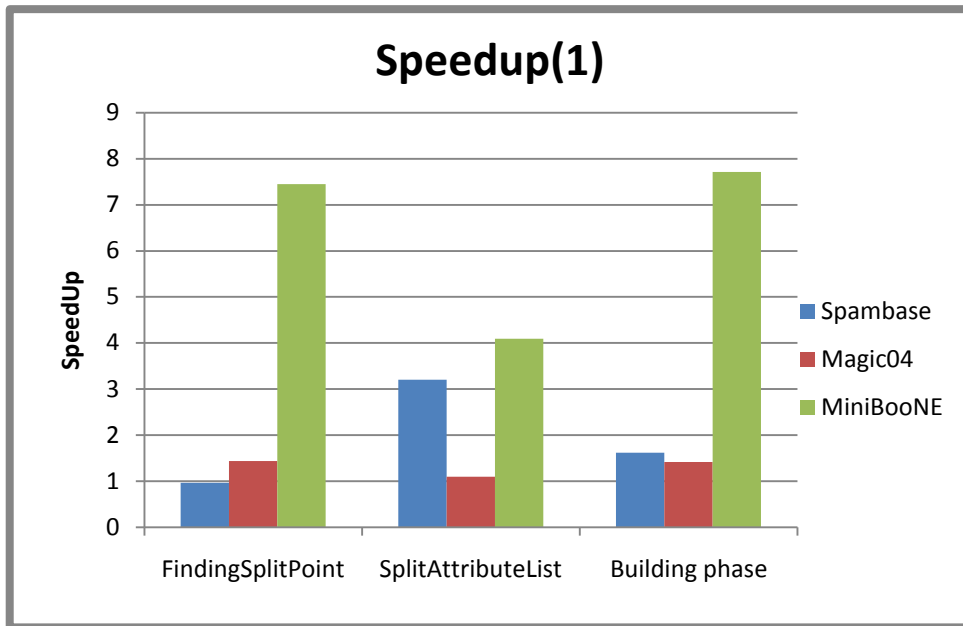


Figure 15 Speedup of each Component (1)

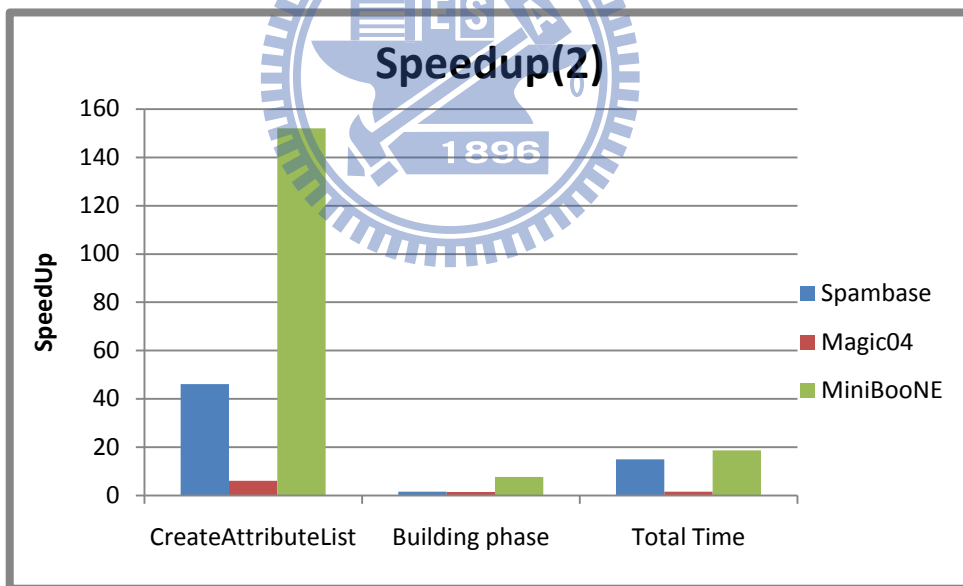


Figure 16 Speedup of each Component (2)

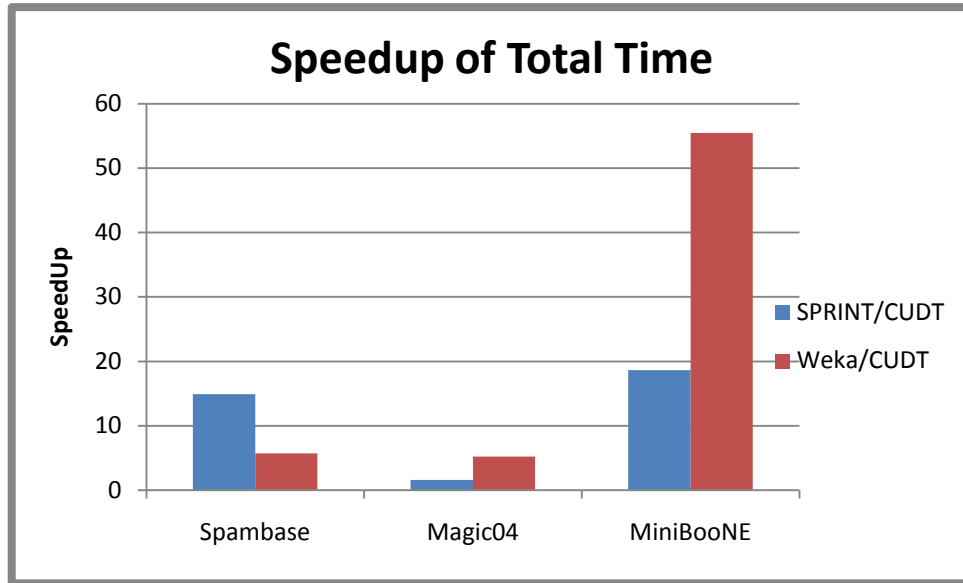


Figure 17 Speedup of building classifier

4.4 Evaluation of Each Level

In this section, we will evaluate each level of building tree. Since create attribute lists performs very nice speedup, we focus on building phase. Figure 19, 21, 23 show the execution time of each level of CUDT and SPRINT. Figure 20, 22, 24 show the speed up of each level. The best speedup is always on the first level. There are two reasons for the result. First, the active data size is always big on the first level. A GPU device is composed with many weak cores. Large data can exert the computation power of GPU. It is why CUDT is always better than SPRINT on first level.

Second, increased nodes on each level increases the times of communication between CPU and GPU. Since we only parallel the computation of creating a single node, the building phase makes a tree iteratively. We need upload some data from GPU to CPU after finding the split points. More nodes increase the data move between CPU and GPU. The following level's speedup is not as good as first level since the increased node size.

Since above reasons, the upper levels have more good speed up than lower levels. There is a performance turning point between CPU and GPU. Figure 25, 26, 27 shows the relationship between node size and active data size. The blue line presents active data size. The red one is the size of nodes on the level. We can see that the trend of execution time of CUDT is very close to the size of node on each level. It shows that our system is sensitive with node sizes instead data size.



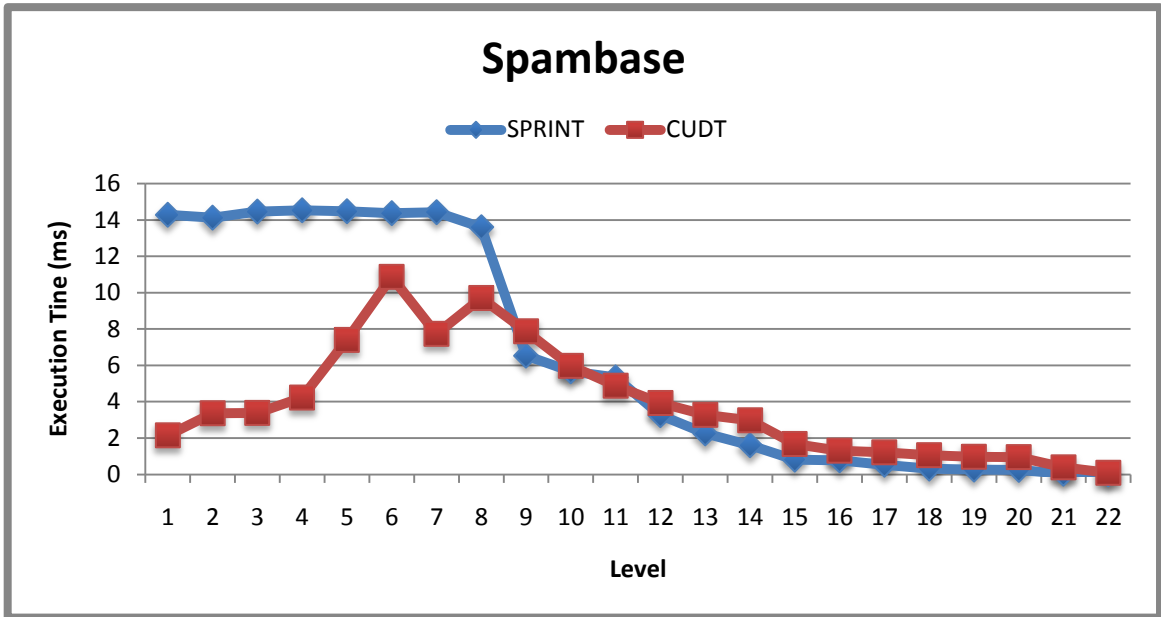


Figure 18 Execution Time of each Level on Spambase

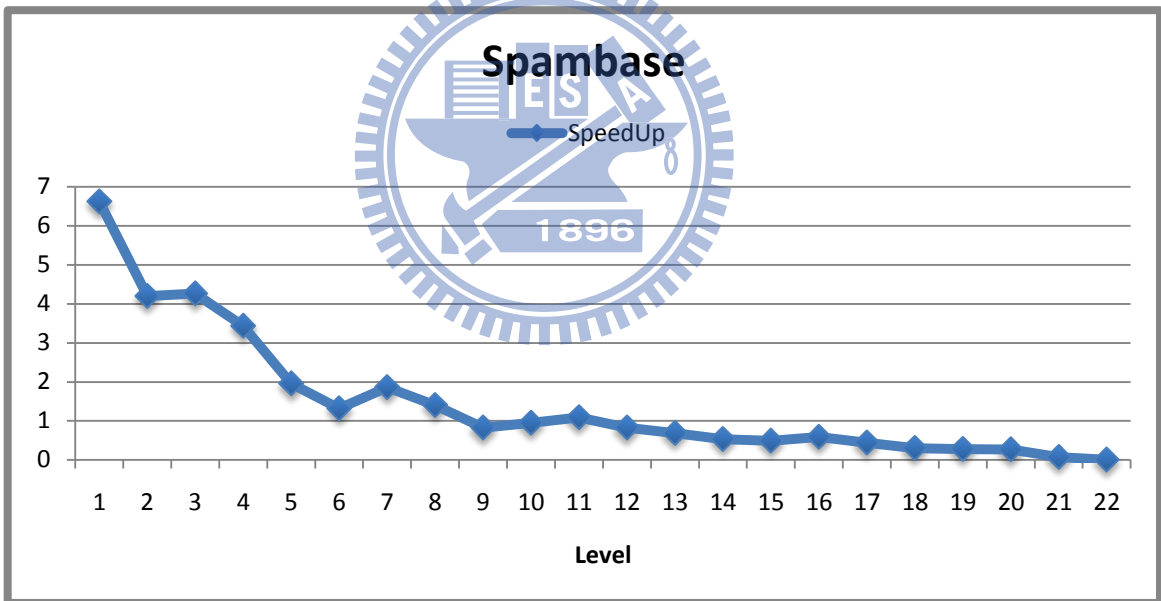


Figure 19 Speedup of Level of Spambase

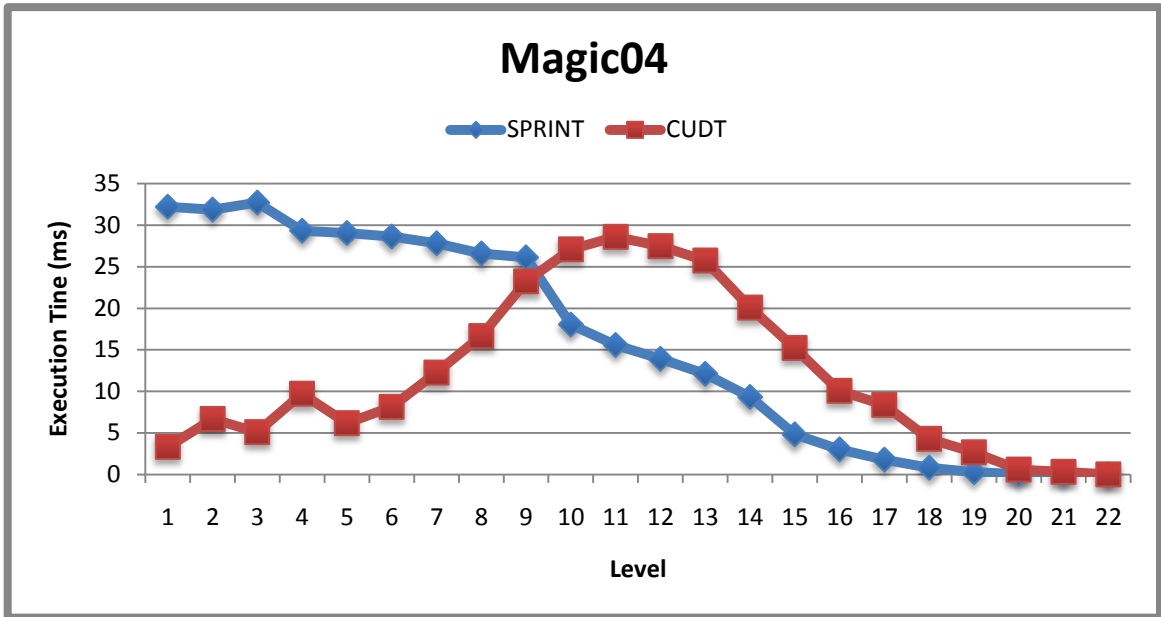


Figure 20 Execution Time of each Level on Magic04

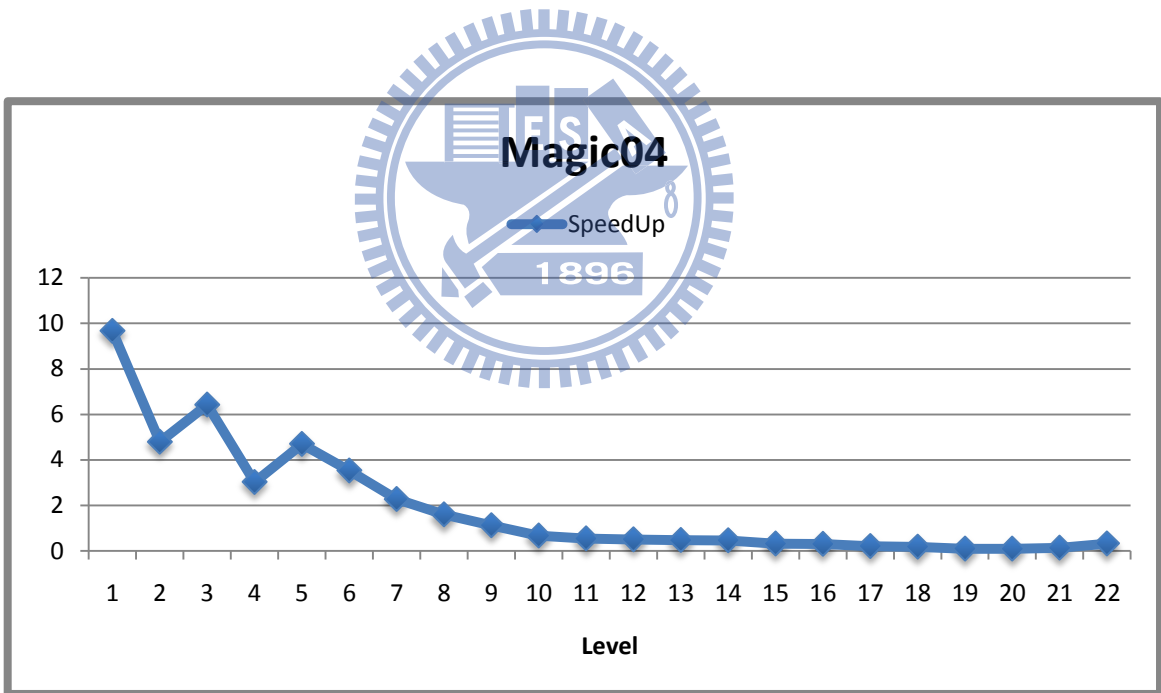


Figure 21 Speedup of Level of Magic04

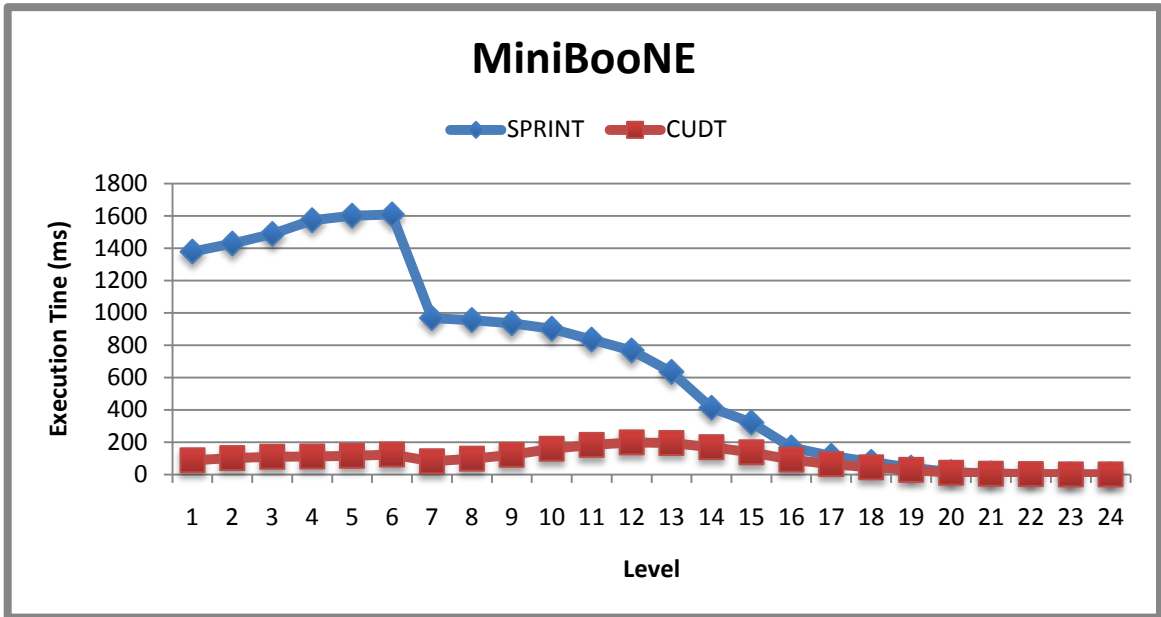


Figure 22 Execution Time of each Level on MiniBooNE

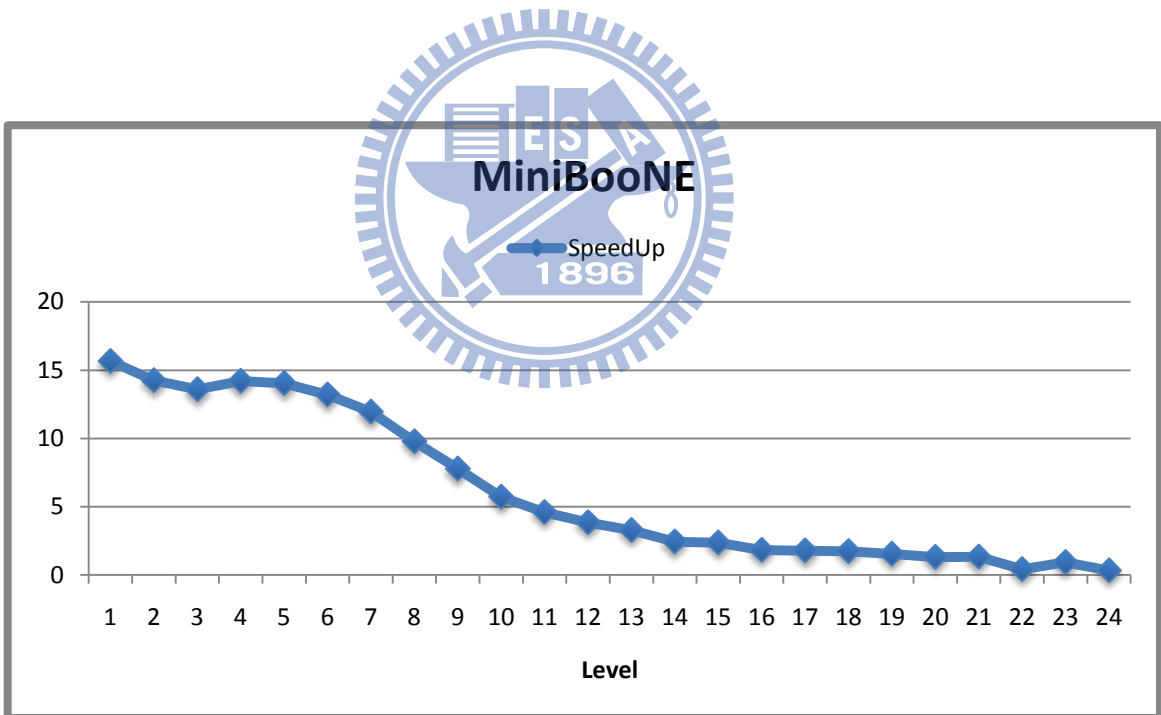


Figure 23 Speedup of Level of MiniBooNE

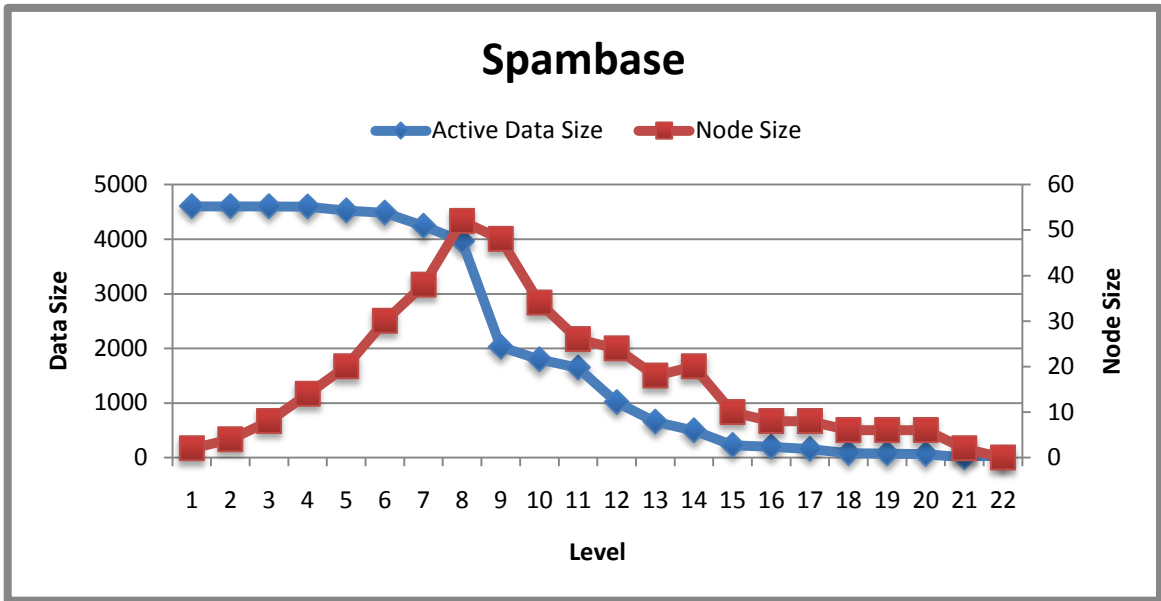


Figure 24 Active Data Size v.s. Node Size on Spambase

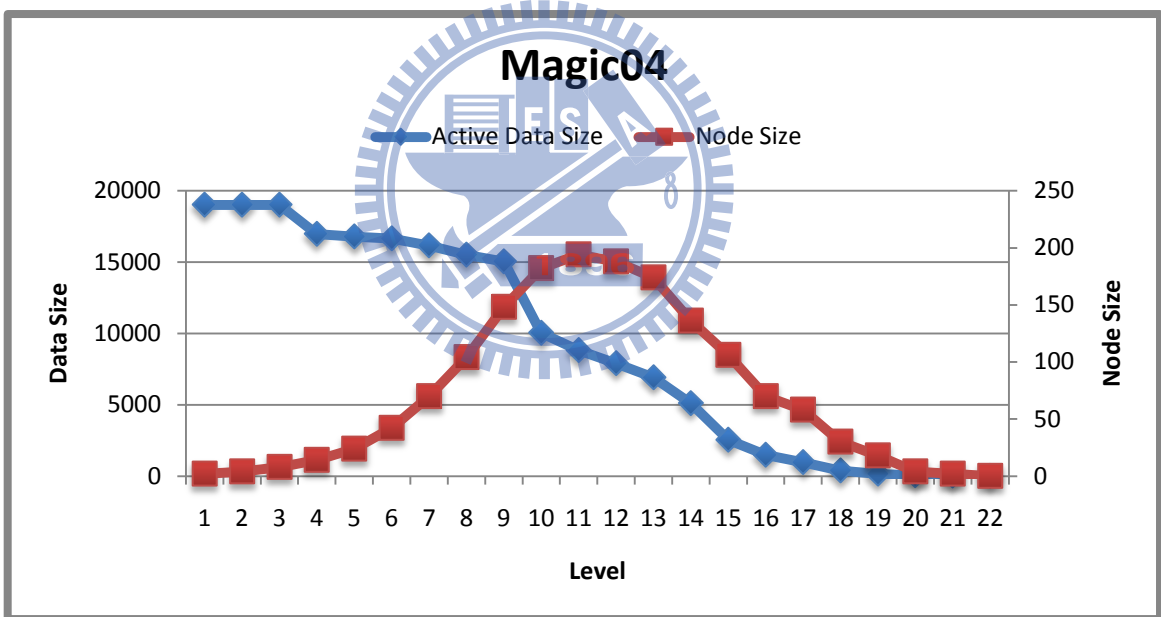


Figure 25 Active Data Size v.s. Node Size on Magic04

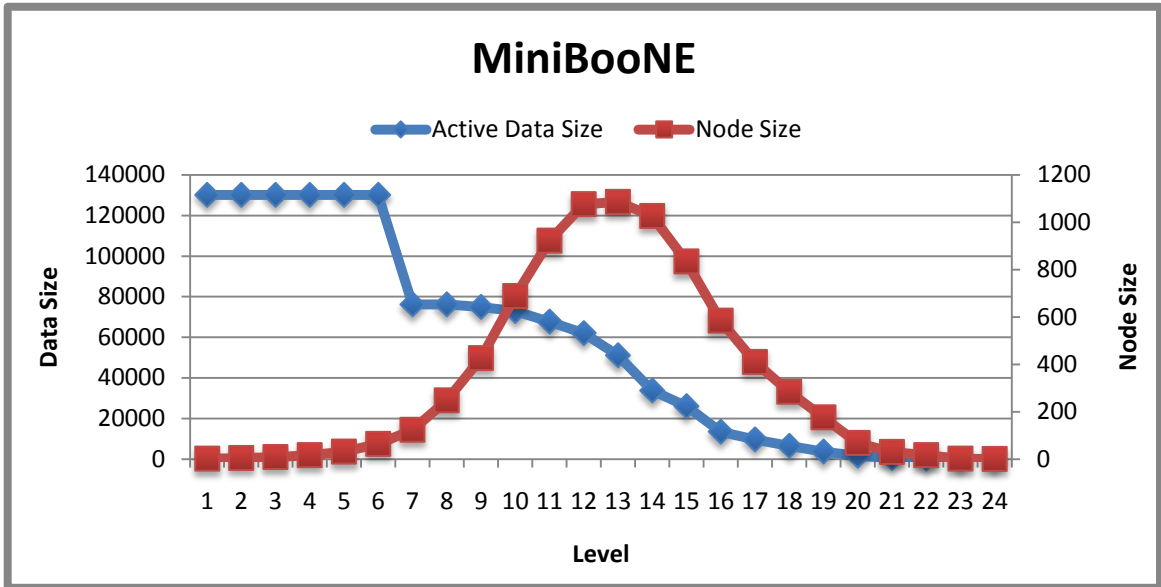



Figure 26 Active Data Size v.s. Node Size on MiniBooNE



Chapter 5 Conclusion

Using GPU for problems with high density computation normally brings remarkable improving of performance. Of course, these problems should be able to be parallelized. Many machine learning algorithms has been developed on CUDA GPUs. They also show performance improvement comparing to the implementation of CPU. Using CUDA for high performance computation is more and more popular since it has high capacity/price and great computing power. In this paper, we survey the background of existing decision tree algorithm and CUDA programming model. We proposed a new parallel algorithm base on CUDA. There are many parallel primitives, for instance the prefix-sum and parallel sorting, are used in our algorithm. We parallel the computation and build tree in sequence. The basic ideal of our choice is scalability.

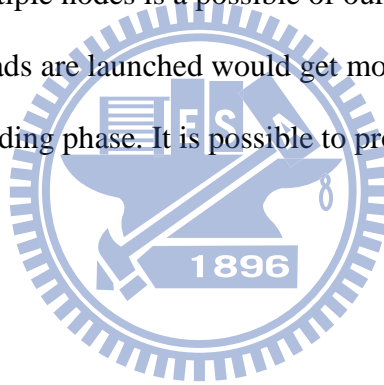


The experiment result shows our performance improvement. Comparing to the famous java open source project Weka. Our CUDT has 5~55 time faster than Weka without significant difference of accuracy. Comparing to our best optimized SPRINT, our CUDT has maximum 18 times faster than SPRINT. However, the tree size of CUDT is greater than Weka. The worst case of our evaluation is about 50% redundant nodes in our system on Magic04 data set. Since the executed time is sensitive with tree size. Reducing tree size of our algorithm is necessary.

Chapter 6 Future Work

The following are our future work:

1. Reducing the tree size is necessary since the redundant nodes not only hurt performance of building but also reduce the accuracy.
2. Handling miss value is very important issue. A miss value is a value of data is unable to identify. Since, increase we aim at performance of building phase, we doesn't solve problem of miss value. The problem of miss value would be taken care in next step of this project.
3. Implement more split criteria.
4. Parallel processing multiple nodes is a possible of our system since CUDA is a SIMT architecture. More threads are launched would get more benefit from GPU. Since we use a BFS algorithm in building phase. It is possible to process multiple nodes concurrently.



Reference

- [1]. “NVIDIA CUDA Programming Guild, 3.2 edition”, NVIDIA Corporation 2010.
- [2]. “NVIDIA CUDA Best Practices Guild, 3.2 edition”, NVIDIA Corporation 2010.
- [3]. Mark Harris, “Optimizing Parallel Reduction in CUDA”, NVIDIA Corporation
<http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf>.
- [4]. Mark Harris, “CUDPP: CUDA Data-Parallel Primitives Library 1.1.1”, NVIDIA, UCDAVIS, 29 April 2010 <<http://code.google.com/p/cudpp/>>.
- [5]. Manish Mehta, Rakesh Agrawal, and Jorma Rissanen, “SLIQ: A fast scalable classifier for data mining”. In *Proc. of the fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.
- [6]. J. C. Shafer, R. Agrawal, and M. Mehta. “SPRINT: A scalable parallel classifier for data mining”. In *Proc. 22nd Int. Conf. Very Large Databases, VLDB*, pages 544–555, 1996.
- [7]. Venu Satuluri, ”A survey of parallel algorithms for classification”, 15 March 2007.
- [8]. Mark Harris, “Parallel Prefix Sum (Scan) with CUDA”, April 2007.
- [9]. J. R. Quinlan, “C4.5: Programs for Machine Learning”, Morgan Kaufman, 1993.
- [10]. Toby Sharp, “Implementing Decision Trees and Forests on a GPU”, Microsoft Research, Cambridge UK.
- [11]. Daniel Slat, Mikael Hellborg Lapajne, “Random Forests for CUDA GPUs”, 2010.
- [12]. L. Breiman, Random forests. *Machine Learning*, 45:5–32, 2001.
- [13]. Guy E. Blelloch, “Prefix Sums and Their Applications”, CMU-CS-90-190, November 1990.
- [14]. L. Breiman, J.H. Friedman, R. A. Olshen, and C.J. Stone, “Classification and Regression Trees.” Wadsworth, Belmont, 1984.
- [15]. Weka 3, <<http://www.cs.waikato.ac.nz/ml/weka/>>.