DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Implementing Streaming Parallel Decision Trees on Graphic Processing Units

DAVID SVANTESSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Implementing Streaming Parallel Decision Trees on Graphic Processing Units

DAVID SVANTESSON

Master in Computer Science
Date: June 18, 2018
Supervisor: Henrik Boström
Principal Supervisor: Theodore Vasiloudis
Examiner: Örjan Ekeberg
Swedish title: En implementering av Streaming Parallel Decision Trees på grafikkort
School of Electrical Engineering and Computer Science

Abstract

Decision trees have long been a prevalent area within machine learning. With streaming data environments as well as large datasets becoming increasingly common, researchers have developed decision tree algorithms adapted to streaming data. One such algorithm is SPDT, which approaches the streaming data problem by making use of workers on a network combined with a dynamic histogram approximation of the data. There exist several implementations for decision trees on GPU, but those are uncommon in a streaming data setting.

In this research, conducted at RISE SICS, the possibilities of accelerating the SPDT algorithm on GPU is investigated. An implementation is successfully created using the CUDA platform. The implementation uses a set number of data samples per layer to better fit the GPU platform. Experiments were conducted to investigate the impact on both accuracy and speed. It is found that the GPU implementation performs as well as the CPU implementation in terms of accuracy, suggesting that using small subsets of the data in each layer is sufficient for making accurate split decisions. The GPU implementation is found to be up to 113 times faster than the reference Scala CPU implementation for one of the tested datasets, and 13 times faster on average over all the tested datasets. Weak parts of the implementation are identified, and further improvements are suggested to increase both accuracy and runtime performance.

Sammanfattning

Beslutsträd har länge varit ett betydande område inom maskininlärning. Strömmande data och stora dataset har blivit allt vanligare, vilket har lett till att forskare utvecklat algoritmer för beslutssträd anpassade till dessa miljöer. En av dessa algoritmer är SPDT. Denna algoritm använder sig av flera arbetare i ett nätverk kombinerat med en dynamisk histogram-representation av data. Det existerar flera implementationer av beslutsträd på grafikkort, men inte många för strömmande data.

I detta forskningsarbete, utfört på RISE SICS, undersöks möjligheten att snabba upp SPDT genom att accelerera beräkningar med hjälp av grafikkort. En lyckad implementation skriven i CUDA beskrivs. Implementationen anpassar sig till grafikkortsplattformen genom att använda sig utav ett bestämt antal datapunkter per lager. Experiment som undersöker effekten på noggrannhet och hastighet har genomförsts. Resultaten visar att GPU-implementationen presterar lika väl som CPU-implementationen vad gäller noggrannhet, vilket påvisar att användandet av en mindre del av data i varje lager är tillräckligt för goda resultat. GPU-implementationen är upp till 113 gånger snabbare jämfört med en existerande CPU-implementation skriven i Scala, och är i medel 13 gånger snabbare. Svagaheter i implementationen identifieras, och vidare förbättringar till implementationen föreslås för att förbättra både noggrannhet och hastighetsprestanda.

Contents

1 Introduction

The area of machine learning has progressed from a research topic to popular commercial use in the last two decades [1]. Datasets today have become massive, which results in requiring new approaches to how the data is processed. Graphics processing units (GPUs) can be used to rapidly process large quantities of data within machine learning algorithms [2]. The use of GPUs in machine learning has gained popularity in the recent years, largely due to the introduction of GPGPU (General-purpose computing on graphics processing units) libraries such as CUDA [2]. Streaming data environments are also common, where data arrives in a consecutive manner [3]. This creates new challenges for both machine learning and GPU computing.

One of the popular areas within machine learning is decision trees, due to its comparatively good performance, as well as ease of interpretability [4]. It has a long history of development with many implementations now existing.

Within this thesis we aim to investigate how GPUs can be used to learn decision trees in a streaming data environment, which is conducted by creating an implementation of the algorithm Streaming Parallel Decision Trees (SPDT) by Ben-Haim and Tom-Tov [5].

In this section an overview of the thesis, an outline of relevant background, as well as the motivating factor behind this research is provided.

1.1 Background

Decision trees are a group of machine learning algorithms that can be used for both classification and regression tasks. They are predictive models which partition the input space into separate regions, creating approximations by either using a constant or a simple model within each sub-region [6].

In a streaming data environment, data is continually generated over time. Many use cases exist, such as continuously generated data logs, sensor data, buying behavior in an online market, user behavior in media streaming platforms etc. Streaming data introduces new limitations for decision tree algorithms. Most decision tree algorithms use batch learning methods for data partitioning, such as the algorithms introduced in chapter 2.1, and thus require a static dataset where the complete dataset is available. Since streams provide the data continuously over time, the algorithms must be adapted to be able to incrementally update the models. If data arrives rapidly, or if memory is limited, the algorithms may also have to be adapted to only perform one pass of the data. Online decision trees aim to use the underlying logic of decision trees in a streaming environment, often utilizing approximation strategies when learning from subsets of the dataset.

SPDT were introduced in 2010 by Ben-Haim and Tom-Tov [5]. The algorithm consists of using W number of distributed workers running on multiple systems; each worker obtaining an equal part of the data resulting in a data-parallel implementation. To increase the speed of the implementation, probabilistic approximations are utilized. Instead of using the standard approach for growing trees, histograms are utilized to make approximate decisions. The complete algorithm is described in section 4.2.

Few implementations exist for streaming decision tree algorithms on GPU. A Version of Very Fast Decision Trees (VFDT) was created by Marron, Bifet, and Morales [7]. They demonstrated that mining evolving high-speed data streams in a similar manner to VFDT was well suited for GPUs.

1.2 Problem

The SPDT algorithm requires several processors, cores, or computers on a network for its implementation. Within this thesis we investigate whether it is possible to leverage the massively-parallel architecture of GPU platforms instead, making use of COTS (Commercial off-the-shelf) GPU hardware. Porting the algorithm to GPU is not straight forward due to the GPUs hardware limitations, making investigations into how to parallelize SPDT for GPUs a significant part of the problem. The specific question to be answered in this thesis is:
Are GPUs a viable platform for implementing and accelerating the SPDT algorithm in comparison to a CPU implementation, using a single GPU to conduct all the calculations?

1.3 Purpose

Whilst there exists a host of streaming decision tree algorithms, very few have been considered for GPU platforms. The purpose of conducting this research is to explore the area of streaming decision tree algorithms, and investigate the usefulness of the GPU architecture within the area. GPUs are widely used within machine learning; the aim of this thesis is to broaden the research regarding streaming decision tree algorithms on the platform.

The motivation for implementing SPDT on GPU is due to the inherent characteristics of the algorithm possibly suiting the GPU architecture. SPDT takes a data-parallel approach, distributing the data to a set of workers (processors). The workers create histograms in parallel, with a merge step at the end. This is similar to how proven histogram methods on GPU such as research conducted by Milic et al. [8] behave. In addition, decision trees on GPU have been proven to demonstrate effective performance in research [9][7].

1.4 Objectives

Within this thesis the main objective of the research is to obtain a working implementation for SPDT on GPU. The goal is to demonstrate that a scalable implementation is possible on GPUs, through rigorous testing of the implementation. The aim is to create a first working prototype of the algorithm, to be investigated for further optimizations. This implementation is to be benchmarked and compared to an existing CPU implementation, ensuring that the GPU algorithm is faster, as well as correctly implemented.

1.5 Methodology/Methods

In order to answer the research question the following steps are conducted:

The implementation created is a first prototype, following the original SPDT paper [5] closely, while investigating which parts of the code can be parallelized. The code is written in C/C++ and CUDA. Following this, code profiling is conducted on the current version by benchmarking all parts of the algorithm, as well as using the profiler NVVP. The results from this part of the study is used to identify parts of the code performing poorly. This is used to identify possible optimizations for future implementations, as well as inherent problems in implementing the algorithm for the GPU platform.

In the results section an empirical study of the implementation is conducted, aiming to provide rigorous scientific validity to this research. Measurements are taken on algorithm performance, with regards to prediction accuracy. Datasets with diverse properties are selected for testing. These datasets are tested on the implementation, as well as an existing CPU implementation of SPDT for comparison.

The results are visualized and interpreted in graphical diagrams in the results section, and interpreted and discussed in the discussion section.

1.6 Delimitations

The original SPDT implementation utilizes the Minimum Description Length (MDL) pruning technique to achieve better predictive performance [5]. This is out of scope in this study; this study focuses only on the histogram and tree building aspects of the algorithm.

No C/C++ CPU implementation exist of SPDT. A Scala implementation is used for comparison, which limits the utility of time performance comparisons investigated due to it possibly being inherently slower.

1.7 Outline

A brief outline of each chapter in this thesis is provided below:

- Chapter 2 presents theory needed, such as decision trees and GPU computing.
- Chapter 3 presents an overview of related work regarding decision trees on GPU and CPU, as well as histogram algorithms on CPU and GPU.

- Chapter 4 explains the implementation of SPDT on GPU in detail.
- Chapter 5 presents the results, which includes all tests completed in table and graph format.
- Chapter 6 provides a discussion of the results in chapter 5, which includes discussions regarding the precision of the algorithm, as well as an analysis of performance in terms of speed. It also includes a short section on ethics and sustainability.
- Chapter 7 gives concluding remarks on the research.

2 Relevant Theory

In this chapter relevant background theory is presented, focusing on decision trees and GPU programming.

2.1 Decision Trees

Decision trees for both classification and regression have a rich history and have been around for many decades [10]. One of the now standard algorithms were introduced under the name CART (Classification and Regression Trees) by Breiman et al. [11]. CART, which is also used as an umbrella term for decision tree algorithms, has been expanded by many algorithms such as ID3 and C4.5 by Quinlan [12][13].

Most the information in this chapter has been gathered from two textbooks, "An Introduction to Statistical Learning" by James et al. [6] and "The Elements of Statistical Learning" by Friedman, Hastie, and Tibshirani [14].

2.1.1 Problem definition

As mentioned, decision trees are predictive models which partition the input space into separate regions. This study focuses on classification trees; therefore all properties regarding decision trees in this section assume the problem is referring to classification. Formally, given a tree $\mathbf{T}$ and an input vector $\mathbf{x} \in \mathbb{R}^d$ the task is to map $\mathbf{x}$ to a discrete label y from a set of possible labels $y \in \mathbb{N}$, given by a predictive constant or model found within each region R_i.

Given input the tree is traversed from the root by testing the input features $\mathbf{x}_i$ value against a scalar value within each node, until a leaf node is encountered. Each leaf can be labeled with a class such as the majority class of samples reaching the leaf, or they can return a probability $P(c_i|\mathbf{x})$ for each class using a distribution.

Depending on the feature parameters, the shape of the tree differs. If continuous values are considered the tree becomes a binary tree. At each internal node, referred to as a split node, a Boolean expression is tested. If the expression is true the left branch is taken and if false the right branch is taken. Nominal data, as well as non-binary splitting in general, is handled by many algorithms such as ID3. This can give rise to non-binary trees, which will not be considered in this study.

Figure 2.1 depicts a small classification tree with continuous input parameters. When a sample **x** arrives, the sample's second feature x_2 is tested first. If its value is below 4.6 the left branch is taken, otherwise the right branch is taken. This is conducted until a region R_i is encountered, where the sample is labeled.

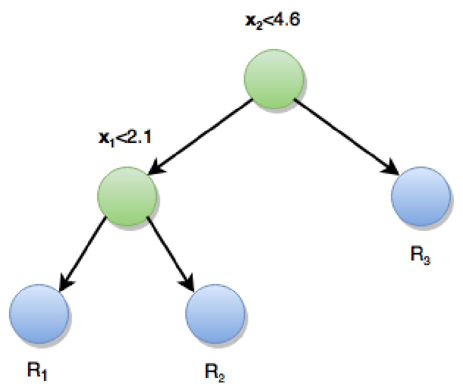

Figure 2.1: Example of a small binary classification tree

2.1.2 Impurity

Before describing how a tree is grown, some definitions must be introduced. Gini impurity for classification is a way to measure uncertainty in a dataset, also known as impurity. It is used in many decision tree algorithms, such as CART. It has the formal definition [6]:

$$Gini = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) \tag{2.1}$$

Given a set **S** = { $(\mathbf{x}_i, y_1)$,. . . , $(\mathbf{x}_N, y_N)$ } of data points and their associated label, Gini impurity measures the total variance between the C classes of the dataset.

In 2.1 each real number $\hat{p}_{mk}$, where $0 \leq \hat{p}_{mk} \leq 1$, is the fraction of the dataset belonging to class k in a partition m of the dataset. The Gini impurity has a small value when each $\hat{p}_{mk}$ are close to either one or zero. For example, if $\hat{p}_{m1} = 0.5$ and $\hat{p}_{m2} = 0.5$, the Gini impurity becomes 0.5, the maximum value the function can take for two classes. If $\hat{p}_{m1} = 0.9$ and $\hat{p}_{m2} = 0.1$ the Gini impurity is 0.18. In the case of the Gini impurity having small values the dataset is said to be pure, which suggests that it contains primarily data points from a single class.

Another common measure of impurity is Entropy. Entropy is used in many algorithms such as ID3 and C4.5, and has the definition [6]:

$$Entropy = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk} \tag{2.2}$$

Entropy has the same characteristics as Gini impurity explained above for the values of $\hat{p}_{mk}$. Given a set **S** of data points, an estimate of how much information a certain feature contains, Gain can be used [12]:

$$\mathit{Gain} = Score(\mathbf{S}) - \sum_{v \in Values(A)} \frac{|\mathbf{S}_v|}{|\mathbf{S}|} Score(\mathbf{S}_v) \tag{2.3}$$

Gain uses either Gini or Entropy as the Score function to calculate how much information is gained by choosing to split a dataset on one of its features. Gain represents the difference between current entropy and the weighted entropy in the child set.

2.1.3 Growing trees

Starting out, a tree consists of just one node, the root node. Decision trees are recursively grown, using either a depth-first or breadth-first approach. In each iteration when growing a tree an unlabeled leaf node is picked. A decision is made as to whether to label the node with some class, or if it is to be split into further nodes. This is normally decided by a stopping criterion, such as only splitting if a minimum number of samples remain at a node. Another common stopping criterion is to consider the node's purity, and stop if it is high enough.

Algorithm 1 below presents a high-level description of a breadth-first approach to growing binary decision trees. Given the set **S** = { $(\mathbf{x}_i, y_1)$,. . . , $(\mathbf{x}_N, y_N)$ } of input-label pairs, the algorithm starts by adding the root node to the tree **T** and the tentative set **TempN**. In each iteration, all leaf nodes in **TempN** are considered. If the node satisfies the stopping criterion the node is labeled by the majority class of the labels in the set $\mathbf{D_{Leaf}}$, where $\mathbf{D_{Leaf}}$ is the data points available in the leaf. Otherwise, all possible splits $\boldsymbol{\Delta}$ are calculated using some function EvaluateSplits(). There are many ways of finding the best split; in the following sections two common approaches are explained. In line 9 a list $\boldsymbol{\Delta}$ over all possible splits are often returned. Line 10 then entails sorting the list $\boldsymbol{\Delta}$ by Gain, and picking the highest value. After a split feature and its corresponding split value has been found, the new child nodes are added to the next iterations tentative set **TempNNext** as well as to the tree **T**. When no unlabeled leaves exist in the tentative set **TempN**, the final tree **T** is returned.

Algorithm 1 Growing decision tree, breadth-first approach

```
Input: A training set D: {$(\mathbf{x}_i, y_1)$,. . . , $(\mathbf{x}_N, y_N)$}
 1: T ← $RootNode$                                   ▷ Tree consisting of root node
 2: TempN ← $RootNode$                       ▷ Set of leaf nodes under consideration
 3: while TempN is not empty do
 4:     TempNNext ← {}           ▷ Initialize empty set for next layer to consider
 5:     for Each $Leaf$ in TempN do
 6:         if $Leaf$ satisfies stopping criterion then
 7:             Label $Leaf$ by majority class $C_{majority}$ in $\mathbf{D_{Leaf}}$
 8:         else
 9:             Calculate all possible splits $\boldsymbol{\Delta}$ for Leaf using EvaluateSplits($\mathbf{D_{Leaf}}$)
10:             Create a split using highest $\boldsymbol{\Delta}$
11:             Add new nodes $Split_{Left}$ and $Split_{Right}$ to TempNNext and T
12:     TempN ← TempNNext
Output: Fully grown tree T
```

Finding split points by exhaustive search

A common way of finding the best split point is by performing an exhaustive search over every possible split value, for every feature. Algorithm 2 details this approach, using Gini Gain as a scoring function to determine how good the split is. First the Gini of the entire set at the current node is calculated, labeled $Gini_D$. Next, each feature value a of each feature $\mathbf{A}$ in the dataset is considered. The data $\mathbf{D}$ is split into $\mathbf{D_{Left}}$ and $\mathbf{D_{Right}}$, using a as a threshold. Gini impurity is calculated on both datasets, and a weighted average is taken of the calculations. Finally, the Gain is calculated. The Gain, the used feature, and the used feature value are added to $\mathbf{\Delta}$. $\mathbf{\Delta}$ is returned when the exhaustive search is completed.

Algorithm 2 Calculate best split point by exhaustive search

```
Input: A dataset D: $\{(\mathbf{x}_i, y_1),\ldots,(\mathbf{x}_N, y_N)\}$
 1: $\mathbf{\Delta} \leftarrow \{\}$                                    ▷ Initialize empty set of split points
 2: $Gini_D \leftarrow \sum_{k=1}^{K} \hat{p}_k(1-\hat{p}_k)$               ▷ Where $\hat{p}_k$ proportion of class $k$ in D
 3: for Each feature A in D do
 4:     for Each feature value $a$ in A do
 5:         Split D into $\mathbf{D_{Left}}$ and $\mathbf{D_{Right}}$ using $a$ as threshold
 6:         $Gini_{D_{Left}} \leftarrow \sum_{k=1}^{K} \hat{p}_{kl}(1-\hat{p}_{kl})$        ▷ $\hat{p}_{kl}$ proportion of class k in $\mathbf{D_{Left}}$
 7:         $Gini_{D_{Right}} \leftarrow \sum_{k=1}^{K} \hat{p}_{kr}(1-\hat{p}_{kr})$       ▷ $\hat{p}_{kr}$ proportion of class k in $\mathbf{D_{Right}}$
 8:         $WeightedGini \leftarrow \hat{p}_r \cdot Gini_{\mathbf{D_{Left}}} + \hat{p}_l \cdot Gini_{\mathbf{D_{Right}}}$
 9:         $Gain \leftarrow (Gini_D - WeightedGini)$
10:         Add Gain, feature, feature value triple to $\mathbf{\Delta}$
Output: List $\mathbf{\Delta}$ of all possible split values, feature, and feature value
```

Finding split points by histograms

Another approach to calculating split points is found in Algorithm 3. This approach creates histograms over the data which is conducted in order to create a smaller, but hopefully still accurate, representation over the feature space. There exist numerous variations using histograms; this example is based on an implementation by Kamath, Cantú-Paz, and Littau [15]. As can be seen from Algorithm 3, it is very similar to Algorithm 2 from line 7 onward. Starting out, the Gini impurity for the entire set **D** is calculated. Next, for each feature a histogram of B bins is created, where B is defined by the user. Split points can then be calculated for example at weighted bin boundaries, as in [15]. This creates a smaller set of possible split points to consider compared to Algorithm 2. The rest of the procedure is analogous to Algorithm 2.

Algorithm 3 Calculate best split point by histogram

```
Input: A dataset D: $\{(\mathbf{x}_i, y_1),\ldots, (\mathbf{x}_N, y_N)\}, integer B$
 1: $\boldsymbol{\Delta} \leftarrow \{\}$                                   ▷ Initialize empty set of split points
 2: $Gini_D \leftarrow \sum_{k=1}^{K} \hat{p}_k(1-\hat{p}_k)$               ▷ Where $\hat{p}_k$ proportion of class k in D
 3: for Each feature A in D do
 4:     $h_A \leftarrow$ Make histogram over feature A
 5:     $SplitPoints \leftarrow$ Create uniform distribution from histogram $h_A$
 6:     for Each feature value $u$ in SplitPoints do
 7:         Split D into $\mathbf{D_{Left}}$ and $\mathbf{D_{Right}}$ using $u$ as threshold
 8:         $Gini_{D_{Left}} \leftarrow \sum_{k=1}^{K} \hat{p}_{kl}(1-\hat{p}_{kl})$        ▷ $\hat{p}_{kl}$ proportion of class $k$ in $\mathbf{D_{Left}}$
 9:         $Gini_{D_{Right}} \leftarrow \sum_{k=1}^{K} \hat{p}_{kr}(1-\hat{p}_{kr})$       ▷ $\hat{p}_{kr}$ proportion of class $k$ in $\mathbf{D_{Right}}$
10:         $WeightedGini \leftarrow \hat{p}_r \cdot Gini_{\mathbf{D_{Left}}} + \hat{p}_l \cdot Gini_{\mathbf{D_{Right}}}$
11:         $Gain \leftarrow (Gini_D - WeightedGini)$
12:         Add Gain, feature, feature value triple to $\boldsymbol{\Delta}$
Output: List $\boldsymbol{\Delta}$ of all possible split values, feature, and feature value
```

2.1.4 Accuracy

Accuracy is the only form of error estimation used in this report. Given N samples, if $f(x)$ is the predicted class of a sample x and y its true value, accuracy is given by:

$$Accuracy = \frac{1}{N} \cdot \sum^{N} [1 | f(x_i) == y_i] \tag{2.4}$$

2.1.5 Pruning

Growing a tree as described has possible drawbacks. The accuracy on the training set may be good, but large decision trees tend to not generalize well and perform poorly on unseen test data due to overfitting the further the tree is grown. This is due to complex trees resulting in higher variance. A way to reduce variance, at a cost of bias, is by making the tree less complex. Pruning is a common way to achieve this. Many ways exist, but the general idea is to test the error rate of the tree before and after removing parts of the tree. If the increase of error is small, or the error rate is decreased, the parts can be removed resulting in a simpler tree.

2.2 Graphic processing units

In this section the most fundamental parts of GPU architecture and GPU programming are introduced. How it differs from serial CPU code is also explained. The *CUDA Programming Guide* [16] is used as a reference throughout the section.

2.2.1 GPU architecture

Code execution on a single CPU is executed in a Single Instruction Single Data (SISD) manner. Instructions are executed serially one after the other, with one instruction acting on a single data in memory. This architecture is specialized for low latency, as is required by most general computing devices. CPUs generally have a large cache, sophisticated flow control to allow for example pipelining and branch prediction, and a small number of arithmetic logic units (ALU).

GPUs are hardware components capable of large-scale parallelism. In contrast to CPUs, GPUs specialize in data-parallel problems, sacrificing latency and advanced flow control in favor of high data throughput. This can be seen in figure 2.2, in which the GPU has a far greater number of ALUs but much smaller space for control and cache in comparison to the CPU. GPUs execute code in a Single Instruction Multiple Data (SIMD) manner, each instruction acting on multiple parts of the memory simultaneously. GPUs consist of several streaming multiprocessors (SMs), which each contain several hundred threads. Each SM runs at a lower clock speed than CPUs. The acceleration comes from being able to process large sets of elements, each requiring several arithmetic operations, simultaneously by data-parallel processing.

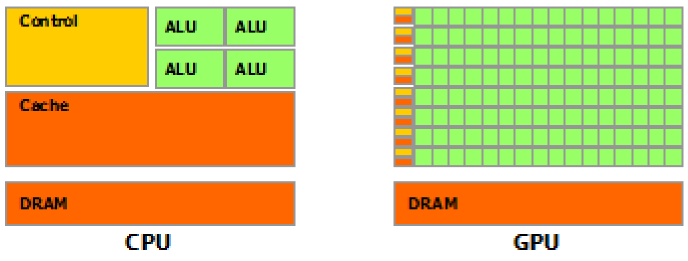

Figure 2.2: CPU and GPU hardware architecture

2.2.2 CUDA Programming

CUDA was introduced in 2006 by NVIDIA, and is the most common framework used for conducting general purpose GPU (GPGPU) programming on GPUs. When using CUDA, NVIDIA separates between CPU and GPU code by using the terms "host" and "device"; both terms are used in this section of the thesis. Languages supported in the CUDA framework are C and C++, while wrappers for many other languages exist as well.

CUDA code consists of kernels, which is the general name for user created functions running on the device, and host code which runs on the host itself. Kernels are launched from the host side, which is performed similarly to calling functions in C++. The difference is that the user must also specify a grid with corresponding blocks.

When launching a kernel, many threads are created on the device. The threads are divided into a hierarchy, with warps and blocks on the lowest level. Each warp consists of 32 threads. The code is executed in lock-step, where each thread in a warp executes the same instruction, and each thread must wait for all other threads in the warp to finish their instruction. A block is a collection of threads defined by the user. Blocks can be defined in up to three dimensions, depending on which structure is needed by the user when solving a problem. Each thread within a block has a thread ID, and is addressable within device code. In the same way as threads are ordered in blocks, blocks are ordered in a grid pattern of up to three dimensions. Within device code, each block has an addressable block ID. The hierarchy of threads, blocks and grids is illustrated in figure 2.3. The execution pattern of host and device code is illustrated in figure 2.4.

Launching kernels with correct block and thread settings are important for performance. An SM runs one block at a time. For maximum performance, it is important that all threads in an SM are utilized. Each block should therefore strive to use all threads available in an SM, which puts limitations on how code should be parallelized for performance. This is known as thread occupancy, the more threads occupied at any moment the better.

When running a kernel, the GPU schedules which blocks to run and when. Synchronization is available at different levels. Threads can be synchronized within each block, forcing all threads to reach the same instruction before continuing. This is used for example to avoid race conditions in reading and writing to memory. When launching a kernel from the host, the host does not wait until the device is ready, but continues with the next instruction. Synchronization exists on kernel level as well, to make sure a kernel execution is conducted before continuing on the host side. Atomic operations also exist.

Device functions are special functions that are similar to common C++ functions. A device function can be called from any thread. No parallelization is defined in device functions; they are simply a way to create reusable code that can be defined outside a kernel.

CUDA code is compiled with the NVCC compiler into PTX, the CUDA instruction set architecture. Users can also write code directly in PTX, which has an assembly like interface. When running CUDA code, the PTX code is compiled into machine code at runtime by a just-in-time (JIT) compiler in the device driver.

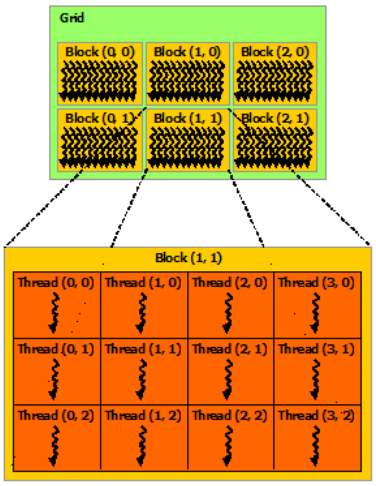

Figure 2.3: CUDA thread hierarchy; grids, blocks and threads

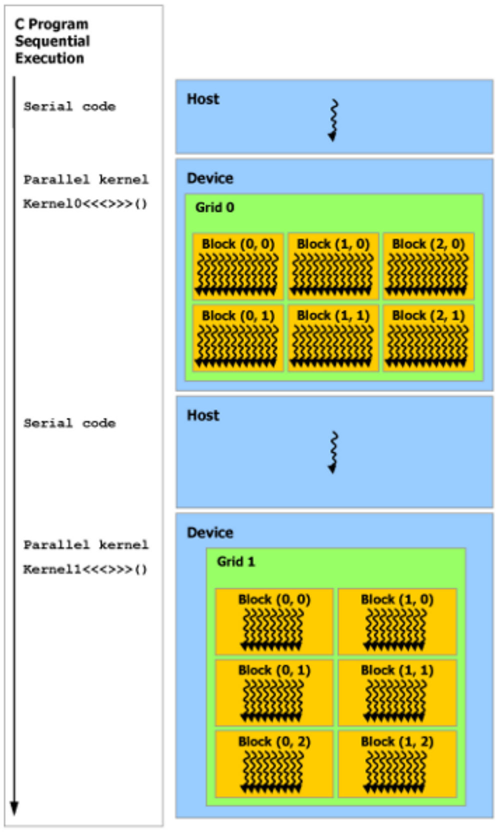

Figure 2.4: CUDA Code Execution

2.2.3 GPU memory layout

Device memory is separated into three distinct layers; global memory, shared memory, and thread local memory. This is illustrated in figure 2.5.

Global Memory

Global memory is explicitly allocated by the user using the cudaMalloc(*) command, which functions similarly to the malloc(*) command in C. This memory is available from anywhere inside device code, i.e. it has a global access scope. Global memory is the slowest device memory to access. Data can be transferred back and forth between host and device through global memory. This is performed by host side function calls such as cudaMemCpy(*). This is also slow; repeated use should be avoided to ensure optimal performance.

Care must be taken when accessing global memory. Memory in the device is represented using segments consisting of up to 256 bytes. Each time global memory is accessed, a transfer is initiated from global memory to an SMs L1 cache, which is accessible by all threads within a block. Users should take care to ensure that threads access memory lying close, which is referred to as coalesced memory access. This minimises memory transfers, which has a considerable impact on performance.

Shared Memory

Shared memory is allocated and initialized in device code and is only accessible from threads within the same block. Shared memory is faster to access than global memory. It is often used when repeated access to memory is necessary. Users can avoid repeated access to global memory by using shared memory for memory intensive operations, to later write shared memory to global memory to improve performance. It can also be used to share data between threads in a block.

Thread Local Memory

Each running thread in a kernel has access to thread local memory, which is private and only accessible within each thread. This memory contains information such as instruction pointers and the threads' stack.

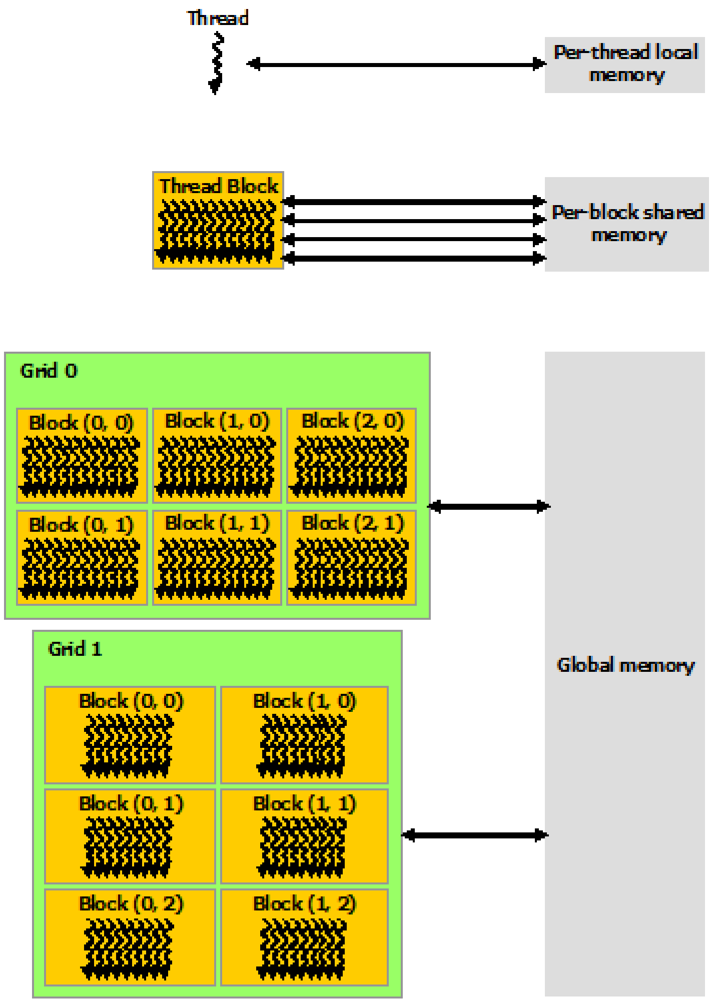

Figure 2.5: GPU Memory Hierarchy

3 Related Work

In this section an overview of research in fields related to this thesis is given. General online decision trees are investigated, demonstrating different solutions considered for this thesis. The original research behind SPDT is explained in an extended section. Both static data and online data GPU implementations of decision trees are investigated. Most of the duration of the training process is spent calculating the best possible split for each leaf node. One of the aims of SPDT is to use algorithmic approximations to gain speedup, the main one being the use of histograms in split finding. Therefore, it is necessary to present some relevant background research on parallel histogram implementations on both CPU and GPU, which is presented in the last section.

3.1 Online Learning Decision Trees

There exists a host of online decision tree algorithms based on ID3 [17]. ID4 by Schlimmer and Fisher [18], is a direct extension of the ID3 algorithm, based on E-scores. E-scores are used to compute the expected information in a node for a feature of any given sample, in a similar way that information gain is used. Information about each feature's E-score is kept in each node. Whenever a new feature obtains a better E-score than the previous best score in the node, the entire sub-tree below the node is discarded. An extension of ID4 by the name ID5R was introduced by Utgoff [19]. The algorithm takes a similar approach to ID4, using E-scores to determine node splits. The major change from ID4 is when a better E-score is found in another feature, the tree is restructured, instead of dropping the sub-tree. A major drawback of both ID4 and ID5R is that it only manages binary data, and is unable to handle numeric data [17]. The algorithm was extended again by Utgoff under the name ITI, focusing on the handling of numerical data [20].

Another branch of incremental decision trees was introduced in 2000 in the form of Very Fast Decision Trees (VFDT) by Domingos and Hulten [21]. VFDT uses the Hoeffding bound for node splitting, creating what they refer to as Hoeffding trees. The Hoeffding bound is used as a measure of how many samples are needed to give a correct representation of the data, within a specified error margin. Samples are collected in each leaf of the tree, and a conventional approach to finding the optimal split is used on the collected subset when a large enough number of samples have been gathered. The required amount in each node is decided by the Hoeffding bound. It is highlighted that this approximation approach to building decision trees are asymptotically close to trees produced by a conventional batch algorithm, and therefore does not have a large negative impact on predictions made by the final model if the error margin for the Hoeffding bound is kept small enough. This was demonstrated empirically by comparing the algorithm to C4.5. Further, VFDT learns in small constant time per sample, which is an important feature in a streaming environment. An extension to VFDT called Concept Adapting Very Fast Decision Trees (CVFDT) was implemented in 2001 by Hulten, Spencer, and Domingos [22]. VFDT assumes that the underlying generation of data is drawn from a stationary distribution, CVFDT extends VFDT to handle concept drift [17].

Further approaches to general online decision trees include PLANET [23], ScalParC [24], and Mondrian forests [25].

3.2 Streaming Parallel Decision Trees

The use of histograms for approximating split value decisions are a popular alternative to explicit evaluation. The classical method of searching for good values to split a feature on is by conducting an exhaustive search, as explained in algorithm 3. By using histograms, the possible number of split points to consider can be reduced considerably.

There are two approaches to using histograms for split finding. The first one performs a pre-sorting of the data in order to handle large volumes like in algorithm 3. The second method uses incremental histograms to create approximate representations of the data [5], such as the algorithm presented in this section. Pre-sorting is conducted in for example SPRINT [26] and SLIQ [27]. Examples of approximate representations of the data can be found in for example CLOUDS [28] and SPIES [29].

A first attempt to use histograms in a parallel streaming environment for node splitting was developed by Ben-Haim and Tom-Tov [5]. Their SPDT algorithm consisted of using W number of distributed workers running on multiple systems; each worker obtaining an equal part of the training data resulting in a data-parallel implementation.

To increase the speed of the implementation, probabilistic approximations are utilized. Instead of using the standard approach for node splitting, online histograms are used to make approximate split decisions. Each worker has a full view of the current tree being built. The workers accumulate samples in the leaf nodes of the tree, building a separate histogram h for each class in each leaf, and for each feature within each class. The histograms all have B bins, where each bin contains a point p_i symbolizing the bin mean, as well as a counter m_i which counts the number of samples in the bin. With every arriving sample the closest bins in corresponding histogram are merged. At certain intervals, the workers send their accumulated histograms back to the master process which merges all histograms. The merged histograms are used to for split decisions. The splits are performed by creating uniform distributions from each accumulated histogram. A parameter $\tilde{B}$ defines how many points each uniform distribution should consist of. The $\tilde{B}$ points of the histograms are the considered split values for each feature. Speedup was shown to scale well on number of samples, as well as on number of workers used.

There are four important procedures in the algorithm regarding the use of histograms that should be explained further. The four procedures are referred to as UPDATE, MERGE, SUM, and UNIFORM. All pseudo code is taken directly from the original paper, they were included here to ensure that the reader fully understands their algorithm.

The UPDATE procedure takes a histogram h and a sample point p, and adds p to histogram h. The algorithm is explained in figure 3.1. In line 3 a new bin $(p, 1)$ is created, resulting in $B + 1$ bins in the histogram. The histogram bins are sorted on the points $p_1 \dots p_{B+1}$. The sorted points are used to find the two points p_i and p_{i+1} closest to each other. These two bins are then merged, as can be seen in line 7.

The MERGE procedure takes several histograms $h_1 \dots h_n$ as input, and returns a new histogram consisting of B bins resulting from merging histograms $h_1 \dots h_n$. This is conducted similarly to UPDATE, but the bin merging is conducted repeatedly until the resulting histogram consists of B bins. Figure 3.2 shows the MERGE algorithm. In line 2 all bins in all histograms $h_1 \dots h_n$ are sorted, giving

```
input A histogram $h = \{(p_1, m_1), \ldots, (p_B, m_B)\}$, a point $p$.
output A histogram with $B$ bins that represents the set $S \cup \{p\}$, where $S$ is the set represented by $h$.
1: if $p = p_i$ for some $i$ then
2:     $m_i = m_i + 1$
3: else
4:     Add the bin $(p, 1)$ to the histogram, resulting in a histogram of $B+1$ bins $h \cup \{(p, 1)\}$. Denote
       $p_{B+1} = p$ and $m_{B+1} = 1$.
5:     Sort the sequence $p_1, \ldots, p_{B+1}$. Denote by $q_1, \ldots, q_{B+1}$ the sorted sequence, and let $\pi$ be a
       permutation on $1, \ldots, B+1$ such that $q_i = p_{\pi(i)}$ for all $i = 1, \ldots, B+1$. Denote $k_i = m_{\pi(i)}$,
       namely, the histogram $h \cup (p, 1)$ is equivalent to $(q_1, k_1), \ldots, (q_{B+1}, k_{B+1})$, $q_1 < \ldots < q_{B+1}$.
6:     Find a point $q_i$ that minimizes $q_{i+1} - q_i$.
7:     Replace the bins $(q_i, k_i)$, $(q_{i+1}, k_{i+1})$ by the bin
           $\left(\frac{q_i k_i + q_{i+1} k_{i+1}}{k_i + k_{i+1}}, k_i + k_{i+1}\right).$
8: end if
```

Figure 3.1: UPDATE procedure [5]

a sorted sequence of length $\sum_{i=1}^{N} B_i$. In line 3 to 6 the closest bins are found and merged; this is repeated until the resulting histogram consists of B bins.

```
input Histograms $h_1 = \{(p_1^{(1)}, m_1^{(1)}), \ldots, (p_{B_1}^{(1)}, m_{B_1}^{(1)})\}$, $h_2 = \{(p_1^{(2)}, m_1^{(2)}), \ldots, (p_{B_2}^{(2)}, m_{B_2}^{(2)})\}$, an inte-
    ger $B$ .
output A histogram with $B$ bins that represents the set $S_1 \cup S_2$, where $S_1$ and $S_2$ are the sets repre-
    sented by $h_1$ and $h_2$, respectively.
1: For $i = 1, \ldots, B_1$, denote $p_i = p_i^{(1)}$ and $m_i = m_i^{(1)}$. For $i = 1, \ldots, B_2$, denote $p_{B_1+i} = p_i^{(2)}$ and
   $m_{B_1+i} = m_i^{(2)}$.
2: Sort the sequence $p_1, \ldots, p_{B_1+B_2}$. Denote by $q_1, \ldots, q_{B_1+B_2}$ the sorted sequence, and let $\pi$ be a
   permutation on $1, \ldots, B_1+B_2$ such that $q_i = p_{\pi(i)}$ for all $i = 1, \ldots, B_1+B_2$. Denote $k_i = m_{\pi(i)}$,
   namely, the histogram $h_1 \cup h_2$ is equivalent to $(q_1, k_1), \ldots, (q_{B_1+B_2}, k_{B_1+B_2})$, $q_1 < \ldots < q_{B_1+B_2}$.
3: repeat
4:     Find a point $q_i$ that minimizes $q_{i+1} - q_i$.
5:     Replace the bins $(q_i, k_i)$, $(q_{i+1}, k_{i+1})$ by the bin
           $\left(\frac{q_i k_i + q_{i+1} k_{i+1}}{k_i + k_{i+1}}, k_i + k_{i+1}\right).$
6: until The histogram has $B$ bins
```

Figure 3.2: MERGE procedure [5]

The SUM procedure takes a histogram h together with a point b as input, and returns the estimated number of points in the interval $[-\infty, b]$. In other words, the procedure returns the sum of the estimated bin counters smaller than the point b. The algorithm is illustrated in figure 3.3. In line 1 a bin is found such that $p_i \leq b < p_{i+1}$, the point b lying between two bin means p_i and p_{i+1}. Calculating the sum of points smaller than i is trivial, which is done by summing bin counts smaller than m_i in line 3. The number of counted points between p_i and b can be estimated by calculating the trapezoid spanned by the two points and then calculating its area. This is done in line 2. In line 6 the left side of bin i is added. This is necessary because the trapezoid used to calculate the area starts from the middle of bin i, which is represented by point p_i.

```
input  A histogram $\{(p_1,m_1),\dots,(p_B,m_B)\}$, a point $b$ such that $p_1 < b < p_B$.
output  Estimated number of points in the interval $[-\infty, b]$.
1: Find $i$ such that $p_i \leq b < p_{i+1}$.
2: Set
       $s = \frac{m_i + m_b}{2} \cdot \frac{b - p_i}{p_{i+1} - p_i}$
   where
       $m_b = m_i + \frac{m_{i+1} - m_i}{p_{i+1} - p_i}(b - p_i).$
3: for all $j < i$ do
4:     $s = s + m_j$
5: end for
6: $s = s + m_i/2$
```

Figure 3.3: SUM procedure [5]

The UNIFORM procedure takes a histogram h as well as an integer $\tilde{B}$ as input, and returns a set of numbers $u_1 < \cdots < u_{\tilde{B}}$. The returned set of numbers is a uniform distribution created from the histogram h, taking all bin means and counters into consideration. Figure 3.4 describes how the uniform points are calculated. The procedure is described as a reverse SUM function, where bin coordinates are calculated given the trapezoid created by the histogram bins.

With the histogram procedures explained, the tree growing algorithm can now be discussed. The algorithm is illustrated in figure 3.5. The algorithm differs only in how candidate splits are estimated by using online histograms compared to regular decision tree implementations. Each iteration of the tree growing starts at line 3, where all current data samples are navigated to each current leaf. How many samples are used is not stated in the algorithm. The algorithm can use either Gini or information gain to calculate δ in line 8.

As mentioned, the algorithm uses W workers to do some of the computation.

```
input  A histogram $\{(p_1, m_1), \ldots, (p_B, m_B)\}$, an integer $\tilde{B}$.
output  A set of real numbers $u_1 < \ldots < u_{\tilde{B}}$, with the property that the number of points between
    two consecutive numbers $u_j, u_{j+1}$, as well as the number of data points to the left of $u_1$ and to
    the right of $u_{\tilde{B}}$, is $\frac{1}{\tilde{B}} \sum_{i=1}^{B} m_i$.
1: for all $j = 1, \ldots, \tilde{B} - 1$ do
2:     Set $s = \frac{j}{\tilde{B}} \sum_{i=1}^{B} m_i$
3:     Find $i$ such that $\texttt{sum}([-\infty, p_i]) < s < \texttt{sum}([-\infty, p_{i+1}])$.
4:     Set $d$ to be the difference between $s$ and $\texttt{sum}([-\infty, p_i])$.
5:     Search for $u_j$ such that
           $$d = \frac{m_i + m_{u_j}}{2} \cdot \frac{u_j - p_i}{p_{i+1} - p_i},$$
       where
           $$m_{u_j} = m_i + \frac{m_{i+1} - m_i}{p_{i+1} - p_i}(u_j - p_i).$$
       Substituting
           $$z = \frac{u_j - p_i}{p_{i+1} - p_i},$$
       we obtain a quadratic equation $az^2 + bz + c = 0$ with $a = m_{i+1} - m_i$, $b = 2m_i$, and $c = -2d$.
       Hence set $u_j = p_i + (p_{i+1} - p_i)z$, where
           $$z = \frac{-b + \sqrt{b^2 - 4ac}}{2a}.$$
6: end for
```

Figure 3.4: UNIFORM procedure [5]

```
input  Training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
1: Initialize $T$ to be a single unlabeled node.
2: while there are unlabeled leaves in $T$ do
3:     Navigate data samples to their corresponding leaves.
4:     for all unlabeled leaves $v$ in $T$ do
5:         if $v$ satisfies the stopping criterion or there are no samples reaching $v$ then
6:             Label $v$ with the most frequent label among the samples reaching $v$
7:         else
8:             Choose candidate splits for $v$ and estimate $\Delta$ for each of them.
9:             Split $v$ with the highest estimated $\Delta$ among all possible candidate splits.
10:        end if
11:    end for
12: end while
```

Figure 3.5: Tree building procedure [5]

This work is explained in figure 3.6. Each worker receives $\frac{N}{W}$ of the data, where N is the total number of samples. Each worker navigates it's given samples to a leaf node. If the leaf node is unlabeled, for all features i the corresponding histograms $h(v,i,c)$ are updated where v is the leaf index, i feature index and c the corresponding class index. The only part conducted in the workers is the UPDATE procedure.

When the workers have completed updating all its histograms, the histograms are sent to the master process. First, the master merges the histograms $h(v,i,j)$ from all workers using the MERGE function. For all leaves that do not fulfill the stopping criterion, the histograms $h(v,i,c)$ are used to calculate the candidate splits for each leaf. To do this the MERGE function is again used, but now in order to merge on sample class. This creates a histogram $h(v,i)$ for each leaf. Histogram $h(v,i)$ is used as input to the UNIFORM procedure together with $\tilde{B}$, which gives the candidate split locations $u_1 < \cdots < u_{\tilde{B}}$ for the leaf. Impurity δ is now calculated for each split location utilizing the SUM procedure and the histograms $h(v,i,j)$. The best split can now be determined as per usual when building a tree, as in line 9 in figure 3.5.

```
input  $1/W$ of the training set, where $W$ is the number of processors
output  histograms to be transmitted to the master processor
1: Initialize an empty histogram $h(v,i,j)$ for every unlabeled leaf $v$, attribute $i$, and class $j$.
2: for all observed training samples $(\mathbf{x}_k, y_k)$, where $\mathbf{x}_k = (\mathbf{x}_k^{(1)}, \ldots \mathbf{x}_k^{(d)})$ do
3:    if the sample is directed to an unlabeled leaf $v$ then
4:       for all attributes $i$ do
5:          Update the histogram $h(v,i,y_k)$ with the point $\mathbf{x}_k^{(i)}$, using the update procedure.
6:       end for
7:    end if
8: end for
```

Figure 3.6: COMPRESS procedure [5]

3.3 GPU implementations on static data

There exists research regarding the implementation of decision trees and its ensemble counterparts on GPUs for static data. In a static data environment, where all data is available in either memory or on disk, there exists possibilities for parallelization. Since the release of CUDA in 2007, which is the dominant environment for GPGPU programming, researchers and developers have released a host of implementations for the platform. Below is a selection of recent papers, that are either significant within the field or specifically relevant to this study. In general, forests are often considered, as the problem lends itself well to the GPU architecture.

A random forest implementation in CUDA under the name CudaRF was introduced by Grahn et al. [30]. The implementation was approximately 30 times faster than FastRF and 50 times faster than LibRF, both Weka implementations of random forests. The paper identifies basic optimizations for random forests on GPU such as memory management using page-locked memory, global and shared memory usage, as well as the potential increase in efficiency of using information gain instead of Gini impurity on GPUs.

Jansson, Sundell, and Boström [9] introduced two algorithms in 2014 for random forests and extremely random forests, gpuRF and gpuERT. Their study suggested that prior implementations were made considering a few number of cores on older GPU architectures where parallelization was made on a tree-level. It therefore focused on optimizing for newer architectures with a larger number of cores. This is accomplished by building every tree in the entire forest concurrently, taking a task-parallel approach while considering such things as GPU thread scheduling and cache efficiency. Parallelization is performed both on a node level, as well as inter-node level for calculating information gain. The execution scales with tree depth, due to the higher number of leaf nodes further down in the tree. It performs better than contemporary algorithms such as CudaRF.

Lo et al. created CUDT, an implementation based on SPRINT [26] in CUDA [31]. It takes a different approach than CudaRF and gpuRF, focusing solely on parallelizing node splitting on GPU and implements the actual tree building algorithms on CPU. It therefore does not gain any speedup when adding more trees to the forest, but sees larger performance gains when the dataset is large.

An adaption of ID3 was implemented by Nasridinov, Lee, and Park [32]. The approach parallelizes the tree building on both an outer level by distributing nodes over blocks, and within nodes for the sorting of data points for the node splitting procedure. The approach achieves a performance 1.5 faster than CudaRF, and about 5 times faster than Weka.

In a recent study by Zhang, Si, and Hsieh [33], using histograms for split decisions in Gradient Boosted Decision Trees (GBDT) was investigated. The histograms were built completely on the GPU, taking care to minimize atomic conflicts by using data structures fit to the problem. They demonstrated that their histogram implementation on GPU was up to eight times faster than the histogram implementation on CPU, and up to 25 times faster than using non-approximate split finding on CPU. They emphasize that the CPU algorithms run on very powerful 28-core platforms, while the GPU implementations use Commercial of-the-shelf (COTS) GPUs, such as GTX 1080.

3.4 GPU implementations for online data

A version of VFDT optimized for GPUs by the name GVFDT was created by Marron, Bifet, and Morales [7]. They demonstrated that mining evolving high-speed data streams in a similar manner to VFDT was well suited for GPUs. By using a data structure that separate tree traversal and leaf information, and if each tree was considered a balanced binary tree, they could keep the entire forest in a 1D array and were able to efficiently traverse the tree using simple offsets. The leaf information was stored outside of the tree traversal in separate arrays. Information gain and node splitting calculations were both parallelized, as well as calculating the entire forest simultaneously. Due to allocating memory for completely balanced trees, memory space is sacrificed in favor of easy tree traversal for better performance. Comparing to MOAs VFDT implementation they achieved a speedup of 25x. It should be noted to MOAs VFDT runs on a single thread on CPU, thus not sufficiently optimized for a fair comparison. According to their tests GVFDT performs at least 300x better than the VFDT implementation in VFML, a serial CPU implementation written in C.

The study of streaming decision trees on GPUs are very limited, though GPUs do appear to be a viable choice according to the above results.

3.5 CPU implementations of parallel histograms

Calculating histograms in parallel on CPU is a challenging problem, which often requires complicated synchronization patterns as shown by Ahn, Erez, and Dally [34]. This limits parallelism, the higher the probability of two bins being updated concurrently, the more the algorithm becomes serialized. This also highlights the importance of the number of buckets used, as having more buckets available should lessen the probability of memory conflicts. A common way to handle synchronization in parallel implementations is by using memory locks. Every time a value is added to a histogram bin, a lock is used by the process to ensure no memory conflicts occur.

A comprehensive study of implementing histograms on CPU was conducted by Jung, Park, and Lee [35]. Different scalable approaches were implemented for fixed-width and variable-width histograms, as well as histograms with an undetermined number of bins. For a fixed number of bins, multi-core processor thread-level parallelization was performed through two different means. The shared histogram method uses atomic updates as mentioned above, which achieves better performance when the number of bins is large or when the data is not too skewed. This is due to memory access serialization when bins are updated concurrently using atomic operations. The private histogram method uses a distributed approach, maintaining one private histogram in each thread with a final reduction step into a global histogram at the end. The second method has the advantage of avoiding atomic operations, making this approach ideal when the number of bins is small or the data is skewed. The fixed-width implementation performance is comparable to their benchmark comparison in Intel Integrated Performance Primitives (IPP). Comparisons to GPU implementation CUDA Histogram [36] were made, achieving a slightly worse performance than the GPU implementation. It should be highlighted that this GPU implementation is for general histograms, not optimized for any special case, and implemented for older generation GPUs.

3.6 GPU implementations of parallel histograms

In early work researching histograms on GPU, Shams et al. achieved a speedup of up to 30 times compared to their own CPU implementation [37]. Earlier implementations were only able to handle up to 64 bins, which is argued to be too small for many problems. Two implementations were presented, one fit to well behaved data such as a Gaussian distribution, and the other for degenerate distributions or otherwise sparse data. Limitations at the time included no hardware synchronization between threads, as well as very limited shared memory. Both limitations have since been removed by newer NVIDIA hardware; there should therefore be room for further performance gains using an optimized approach.

Milic et al. investigated two approaches to histograms in CUDA [8]. The first used private histogram bins for each block with shared memory, followed by a merge step calculating a global histogram at the end. The second method was written using the Thrust library to first sort the input data, then search for a bin fitting an upper bound. It is highlighted that performance may be impacted due to Thrust not having asynchronous kernel calls, which has been remedied since the paper was released. It was shown that implementing histograms is sensitive to the characteristics of the data. The first method performed better when few data elements were considered, number of bins were large, or the data was evenly distributed. This was identified to be due to the large number of atomic updates necessary, effectively serializing the execution. In the case of many atomic conflicts, a pre-sorting method of the data performed better.

In a developer blog post by NVIDIA, it was demonstrated that modern NVIDIA GPUs are well suited to handle histograms [38]. This is due to the hardware now coming equipped with shared memory atomics; something that studies highlighted above can clearly benefit from.

4 The Implementation

In this chapter the GPU implementation of SPDT is detailed. Software used is given, an overview of the program flow is delineated, the memory layout is explained, and all GPU kernels are described.

4.1 Additional software and libraries used

Apart from CUDA and C/C++, several software is utilized in the implementation. This section describes the software used, as well as why the software is selected.

4.1.1 Python

Python is used for creating an automated testing pipeline. The testing is automated by executing the implementation from python, testing the implementation using different parameters. Python is also used to create all graphs in this paper. The python script creates one JSON file for each execution, all aggregated in a second script creating the final results and graphs.

4.1.2 NVIDIA Visual Profiler

NVIDIA Visual Profiler (NVVP) [39] is a graphical tool developed by NVIDIA used for profiling programs written in CUDA C/C++. The tool is used for identifying bottlenecks in code that can be used as feedback for optimizations. It keeps track of memory transfers, kernel code, and CPU code, giving a unified timeline view over the entire execution of the program. The program also provides application analysis, giving the user hints for optimization by pointing out where and why in the code performance is slow. It is used in this research in order to analyze how parts of SPDT performs on GPUs, and how it should be optimized.

4.1.3 CUB

CUB [40] is a framework by NVIDIA that supplies collective software primitives for CUDA. It contains implementations for problems such as scanning, reduction, and sorting. The functions can be used from host code with devicewide primitives, or from inside device kernels with both block-wide and warp-wide primitives. The implementation within this research makes extensive use of the device-wide sorting functions detailed in chapter 5.

4.1.4 SPDT CPU implementation

This is a Scala implementation of SPDT on CPU, implemented as described in the original paper. The implementation differs from the one in this report in that it uses the entire dataset in every layer when building the tree, removing every correctly classified sample each layer and only training on incorrectly classified samples. Evaluation has only been implemented for two classes, therefore comparisons in accuracy can only be conducted for datasets of this kind. The code is fully available online [41]. In the following chapters this implementation is referred to as CPU-SPDT.

4.1.5 Software versions

Table 4.1 details the software versions used.

Software	Version
Operating System	Ubuntu 16.04.4 LTS
CUDA	9.1
C++	11
CUB	1.8.0
Python	3.6.0

Table 4.1: Software versions used.

4.2 Program overview

The tree building process is illustrated in the flowchart in figure 4.1. Each device kernel is explained in following sections. The entire tree building process is conducted in a breadth-first manner, building one layer of the tree at the time. This is similar to Jansson, Sundell, and Boström [9], who argues that breadth-first fits the GPU architecture better than available depth-first approaches. The entire tree building process is conducted on GPU, avoiding memory copying between each kernel launch. A decision was made to limit the number of samples used per node of the tree. This is inspired by the Hoeffding bound in [21] and [7] introduced in chapter 3.1. The idea is that using only a small subset of the entire dataset in each layer should be sufficient for making split decisions. This implementation does not use a statistical bound, and adds samples per node as a parameter to the algorithm. A further reason for using a subset of the dataset is in each layer is that the implementation is intended to be used in streaming environments, where only parts of the data is available at any given time.

Each function in the SPDT algorithm exposes different levels of possible parallelization, making the use of only one parallelization scheme difficult. Therefore each function has been investigated, and a fitting parallelization scheme has been applied, justified for each kernel in the following sections.

The tree building starts by navigating all samples to the corresponding leaf node as in line 3 in figure 3.5, the tree building, by launching the $navigate\_samples$ kernel. Next, each worker updates it's histograms using the $histogram\_update$ kernel. This corresponds to the entire procedures in COMPRESS and UPDATE, figure 3.6 and figure 3.1. Next is the merging process, which is conducted similarly to the original MERGE in figure 3.2, but has been divided into two processes. Line 2 in figure 3.2, the sorting of $h(v, i, c)$ for all workers, is conducted using CUB's $deviceSegmentedRadixSort$ [42]. This is a kernel that is called from the host side, able to sort a given array in a specified number of segments. Line 3 to 6 is then conducted by the kernel $histogram\_merge\_on\_workers$. Similarly in the next step, the merging of histograms $h(v, i, c)$ into $h(v, i)$ is achieved by first using $deviceSegmentedRadixSort$, followed by the kernel $histogram\_merge\_on\_classes$. Now uniform points used for split finding can be calculated using $h(v, i)$ as in UNIFORM figure 3.4, which is performed by the $histogram\_uniform$ kernel. The $histogram\_uniform$ kernel makes use of the sum procedure, which has been implemented as an inlined device function. The uniform points are now used in the

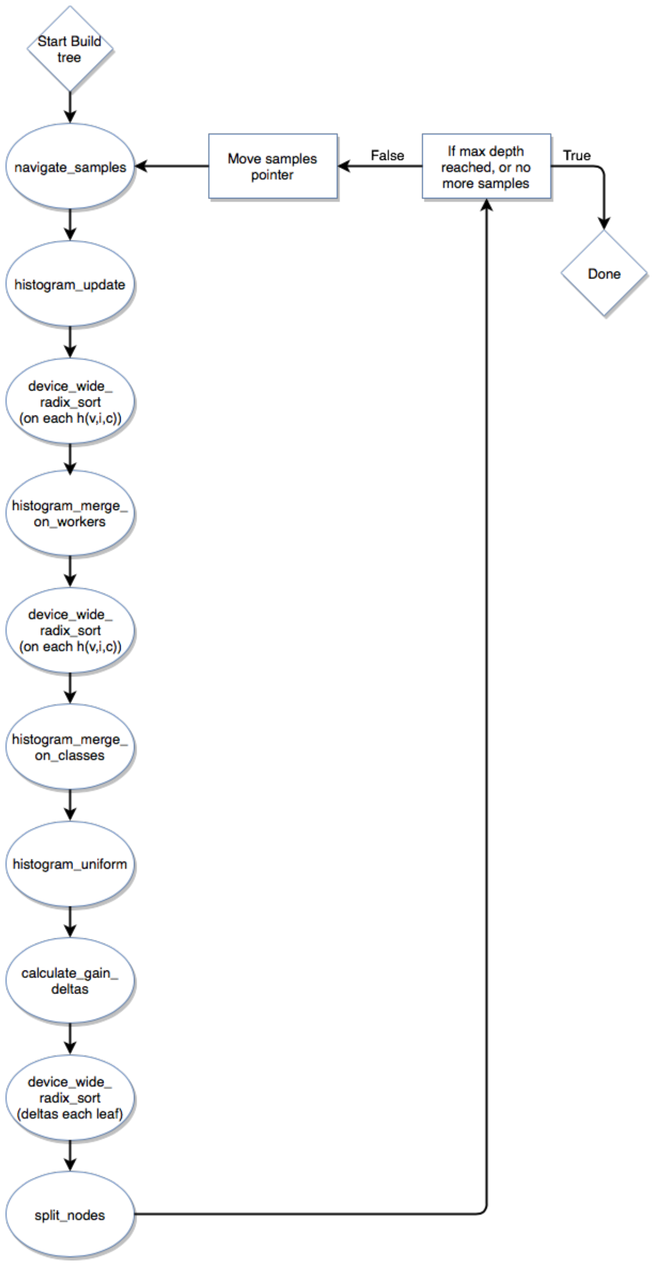
Start Build tree
navigate_samples
Move samples pointer
False
If max depth reached, or no more samples
True
Done
histogram_update
device_wide_ radix_sort (on each h(v,i,c))
histogram_merge_ on_workers
device_wide_ radix_sort (on each h(v,i,c))
histogram_merge_ on_classes
histogram_uniform
calculate_gain_ deltas
device_wide_ radix_sort (deltas each leaf)
split_nodes

Figure 4.1: Program flow of tree building program on GPU

$calculate\_gain\_deltas$ kernel, which calculates the Gini gain of all possible split point for every leaf. Another sorting is conducted using $deviceSegmentedRadixSort$, which is conducted on Gini gain for each leaf in order to easily find highest delta as in line 9 in figure 3.5, the tree building procedure. Finally the tree is extended in the $split\_nodes$ kernel, where stop conditions are checked and split decisions made. This concludes one iteration of the build tree function, which extends the tree one level. A check is made host-side if maximum depth has been reached, or if all samples has been used. If not, memory pointers for samples are moved to point to unseen samples and the tree building continues.

In testing of the implementation it was noted that the accuracy of the algorithm falls considerably when using a higher depth than eight in the algorithm. This is likely a bug in the implementation, but it is the reason why a depth larger than eight is never used when testing accuracy and performance.

4.3 Memory structure

The implementation is conducted with a simple memory structure. This is chosen in order to keep the program logic at a manageable level, and make the tree easy to traverse. The memory is pre-allocated for the entire tree for a maximum tree depth at the beginning of the program, similar to the GVFDT algorithm by Marron, Bifet, and Morales [7]. The reason is that it is more efficient to allocate memory for the entire tree one time, rather than allocating more memory for each layer in the tree required during the building process. The memory layout for the tree is taken from the same paper, although a non-compact approach is taken here. Since the tree is binary the number of nodes required in the tree can be calculated with the simple formula $2^{maxDepth} - 1$, which only depends on the maximum depth set by the user. This means that memory for leaves are allocated for the entire tree in the beginning, not taking into consideration if they will ever be used in the tree building process.

The tree makes use of four 1D arrays, all of the same size. The four arrays are $nodeInfo$, $leafClass$, $features$, and $featureValues$. The $nodeInfo$ array describes if a node is an internal node, or a leaf node. It is a boolean array, where 0 means a leaf node and 1 means an internal node. The $leafClass$ array contains the class for each node in the entire tree, both internal nodes and leaf nodes. The $features$ and $featureValues$ arrays describes which feature each inner node has been split on, and its corresponding split value, and is used for tree navigation. Each array is easy to traverse. To find the child nodes of a node i, simple indexing is performed. The left child node is found by $2i + 1$, and the right child node is found by $2i + 2$. The array structure is exemplified in 4.2, where the $nodeInfo$ array structure is illustrated. The memory structure is the same for all four arrays.

The samples are in this version stored in two 1D arrays, one for features and one for labels. The samples are stored consecutively one after the other. There are several temporary arrays used for intermittent calculations in each kernel. All temporary arrays are allocated at the beginning of the program; they do not impact program performance as they are never transferred to or from the device during tree building.

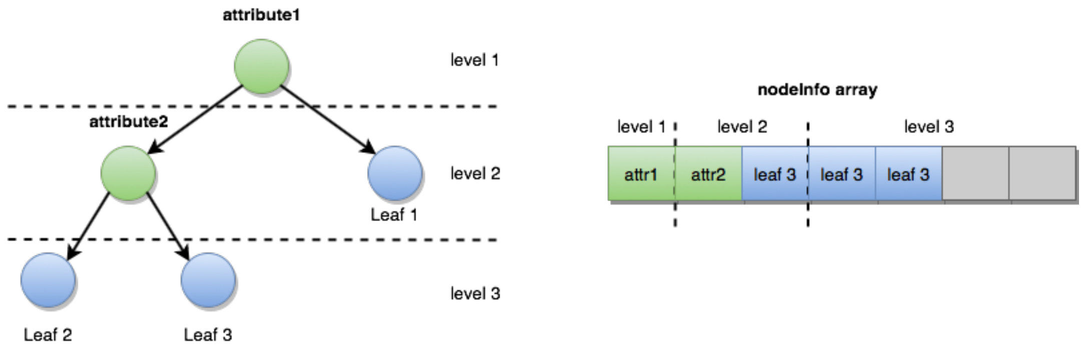

Figure 4.2: Memory layout, illustrates an example tree on the left and corresponding nodeInfo array on the right

4.4 Kernels

The kernels are where all of the tree building is conducted. Each kernel is analyzed in the following chapters in terms of execution time, making it important to explain how each function is parallelized and why those decisions are made.

4.4.1 navigate_samples kernel

This kernel is responsible for navigating samples to the current leaf nodes of the tree, corresponding to line 3 in figure 3.5. The number of threads is set to 128. This number is used in several kernels. The code was developed on a GPU using 128 threads per SM, if running on GPUs with higher threads per SM the maximum value should be used for best performance. The block size is set to $\frac{numberOfSamples}{128} + 1$. Each thread navigates one sample down to a leaf node. Very little work is needed due to the simple tree layout. No synchronization is needed, and parallelization is trivial and scales on the number of samples processed.

4.4.2 histogram_update kernel

This kernel creates and updates each histogram $h(v, i, c)$ within each worker tree. The kernel corresponds to the COMPRESS and UPDATE procedures in figures 3.6 and 3.1. The kernel is launched using W (worker) number of blocks, and each block contains one thread for each feature in the sample set. The kernel therefore scales with characteristics from the dataset. Each worker handles one sample at a time, where each worker thread updates each corresponding feature histogram. The reason for each worker only updating one leaf histogram at a time is due to race conditions that are difficult to solve. A sample can be navigated to any leaf; having more leaves being updated in a single worker at once is difficult due to the possibility of the same histograms being updated simultaneously by different threads.

4.4.3 histogram_merge_on_workers kernel

This kernel is responsible for merging all histograms $h(v, i, c)$ from all workers, which works similarly to the master process in SDPT using the MERGE process in figure 3.2. Line 1 and 2 in figure 3.2 is performed by using CUB's *deviceSegmentedRadixSort* before launching this kernel, only line 3 to 6 is implemented inside the kernel. The blocks use a 3D structure of leaves v, features i, and classes c. The block structure is due to the merging of each $h(v, i, c)$ being independent, thus they can be performed simultaneously in different blocks. The number of threads used is equal to $bins \cdot workers$. This is again due to synchronization issues when merging and replacing bins. All bins from all workers must be merged iteratively within one block. Thread parallelization is conducted by each thread calculating line 4 in figure 3.2 in parallel. Bin merging is only performed by thread 0 in each block, again due to synchronization issues that are difficult to solve. In each iteration the two closest bins have to be found, the bins merged, and all bin indexes higher than the merged bins shifted down one step in order to keep the array ordered. In the hardware used the maximum block size is 1024, which results in the value $bins \cdot workers$ being a limitation of how many bins and workers can be used.

4.4.4 histogram_merge_on_classes kernel

This kernel is similar to *histogram_merge_on_workers*, the difference being that it merges all class histograms into $h(v, i)$ for each leaf and feature. Number of blocks used is a 2D structure of leaves v and features i. The number of threads is equal to the $bins \cdot classes$. Similarly to worker merge, the value $bins \cdot classes$ is a limitation but here limits which datasets can be chosen. Parallelization is performed exactly as in *histogram_merge_on_workers*.

4.4.5 histogram_uniform kernel

This kernel creates one uniform distribution for each leaf and feature. The kernel is an implementation of the UNIFORM procedure in figure 3.4. The blocks are a 2D structure of leaves v and features i. The number of threads is equal to $\tilde{B}$, the number of uniform points to be generated for each feature. Each thread can calculate each uniform point completely asynchronously without any need for synchronization, therefore the entirety of 3.4 can be performed in parallel for each uniform point. This kernel makes use of the SUM procedure in figure 3.3; it has been implemented as a simple device function called by each thread without further parallelization. This was decided due to the kernel already exposing enough parallelization without further parallelizing the SUM procedure.

4.4.6 calculate_gain_deltas kernel

Gain deltas are calculated using Gini gain, it is a parallelization of function 2.3. The number of blocks is equal to the number of leaves; all leaves can be processed in parallel. The number of threads are set to a constant value of 128. Each gain is independent of all other gain calculations, parallelization is therefore embarrassingly parallel and easy to implement. Each thread calculates a gain, jumps 128 steps ahead in the available split points array, and calculates the next gain.

4.4.7 split_nodes kernel

This kernel checks whether a leaf node should be split, or if some stopping criterion has been reached. The four arrays making up all tree information are updated in here. The number of blocks used is equal to $\frac{(numberofleaves)}{128} + 1$, and the number of threads is kept at a constant value of 128. Checking if a leaf should be split or not is completely independent, making this kernel embarrassingly parallel and easy to implement. Each thread checks one leaf, and updates the four tree arrays accordingly.

4.4.8 Kernel settings table

In table 4.2 all kernel launch settings are provided, it delineates the parameters of the algorithm which is useful in the result chapter.

Kernel	Blocks	Threads
$navigate\_samples$	$\frac{samples}{128} + 1$	128
$histogram\_update$	$workers$	$features$
$histogram\_merge\_on\_workers$	$(leaves, features, classes)$	$(workers \cdot bins)$
$histogram\_merge\_on\_classes$	$(leaves, features)$	$(classes \cdot bins)$
$histogram\_uniform$	$(leaves, features)$	*uniform points*
$calculate\_gain\_deltas$	$leaves$	128
$split\_nodes$	$\frac{(leaves)}{128} + 1$	128

Table 4.2: Kernel launch settings

5 Results

This chapter is divided into two sections. The first section presents the achieved accuracy of the implementation, abbreviated as GSPDT, as well as a comparison to CPU-SPDT. It also demonstrates how the implementation behaves during learning. The second section presents how the algorithm performs in terms of speed. This section also goes on to analyze how much time is spent in each kernel, and finally investigates the worst performing kernels.

5.1 Experimental data

The datasets used are all in the LIBSVM format [43]. The data is taken from the LIBSVM website [44], as well as from OpenML [45]. Care has been taken to select datasets with varying properties in order to investigate how the implementation in this thesis behaves depending on the data used. It should be noted that several of the datasets chosen contains two classes, this is to be able to compare GSPDT to CPU-SPDT due to the latter only containing evaluation for such datasets. Table 5.1 contains all datasets used for training and testing.

Dataset	Classes	Features	Samples
Click prediction	2	12	399482
Cod rna	2	8	245057
Eeg eye state	2	15	14980
Letter	26	16	15000
Letter Binarized	2	16	15000
Page Blocks	5	10	5473
Shuttle	9	7	43500
Skin Segmentation	2	3	245057
SVM Guide	2	4	3089

Table 5.1: Datasets used.

The datasets have been divided into train and test sets, and only parts of datasets are used. For datasets without a dedicated test set, a part of the original dataset not used for training was used for testing purposes. The maximum depth investigated is limited to 8 as noted in chapter 5.1, and the number of samples per node are set to 100. These value result in GSPDT needing a maximum of 25600 samples for training. Using only parts of the datasets is important to do in order to be able to properly compare GSPDT to the CPU-SPDT implementation. Table 5.2 shows the size of the train and test sets used for all tests in this chapter.

Dataset	Train Size	Test Size	Classes	Features
Click prediction small	25600	10000	2	12
Cod-rna	25600	10000	2	8
Eeg-eye-state	12800	2180	2	15
Letter	12800	5000	26	16
Letter Binarized	12800	5000	2	16
Page Blocks	3200	2273	5	10
Shuttle	25600	10000	9	7
Skin Segmentation	25600	10000	2	3
SVM Guide	3200	4000	2	4

Table 5.2: Datasets used.

5.2 Hardware

Table 5.3 shows the specifications of the hardware used for all experiments.

Hardware	Specification
CPU	Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz
GPU	NVIDIA GeForce GTX 980 Ti 6GB, Compute Capability 5.2
Memory	64 GB 1600 MHz DDR3

Table 5.3: Hardware used.

5.3 Accuracy tests

The tests in table 5.4 were all completed using 50 bins, 50 uniform points and 2 workers. This parameter choice does impact accuracy; the main purpose here is to demonstrate that the two implementations perform similarly. The exception is Letter, where the number of bins is set to 30, due to the maximum block value noted in section 4.4.4. 100 samples per node is used for GSPDT. This value impacts the performance of GSPDT. A new dataset must be made as in table 5.5 in order to test different values, which is not conducted here due to time constraints. The tests are conducted in order to investigate if performance is similar between the two implementations when using similar settings, on the exact same samples. Evaluation such as accuracy tests have not been implemented for more than two classes in CPU-SPDT, which explains the missing results. The CPU-SPDT results are the results pre-pruning, used for fair comparison. The last row of table 5.4 represents the mean accuracy for all datasets containing two classes. This was included to be able to make a straight forward comparison between GSPDT and CPU-SPDT.

Dataset	Depth	Accuracy GSPDT	Accuracy CPU-SPDT
Click-prediction-small	8	0.8334	0.8318
Cod-rna	8	0.8779	0.9069
Eeg-eye-state	7	0.7183	0.7371
Letter	7	0.3068	n/a
Letter Binarized	7	0.9648	0.9838
Page-blocks	5	0.8720	n/a
Shuttle	8	0.9946	n/a
Skin Segmentation	8	0.9301	0.9929
SVM Guide	5	0.95125	0.69425
Mean 2 class datasets	-	0.8793	0.8578

Table 5.4: Accuracies on test sets, GSPDT uses 100 samples per node. The mean for all datasets containing two classes is provided in the final row.

Another test is conducted with one less depth for each dataset, while increasing the number of samples per node used to 200. The depth decreases due to each node using twice as many samples, which results in needing to lessen the depth by one layer. This test is performed in order to highlight the importance of samples per node used in regard to accuracy in GSPDT. The results are illustrated in table 5.5.

Further tests are conducted to investigate how the implementation behaves in terms of learning in each level of the tree. Figure 5.1 and 5.2 illustrates the accuracy on all training samples seen so far, plotted on depth in the learning procedure.

Dataset	Depth	Accuracy GSPDT	Accuracy CPU-SPDT
Click-prediction-small	7	0.8334	0.8301
Cod-rna	7	0.8838	0.8943
Eeg-eye-state	6	0.5656	0.7211
Letter	6	0.3124	n/a
Letter Binarized	6	0.9662	0.9792
Page-blocks	4	0.8720	n/a
Shuttle	7	0.9956	n/a
Skin Segmentation	7	0.9802	0.9899
SVM Guide	4	0.9462	0.6917
Mean 2 class datasets	-	0.8626	0.8511

Table 5.5: Accuracies on test sets, GSPDT uses 200 samples per node with one less depth per model. The mean for all datasets containing two classes is provided in the final row.

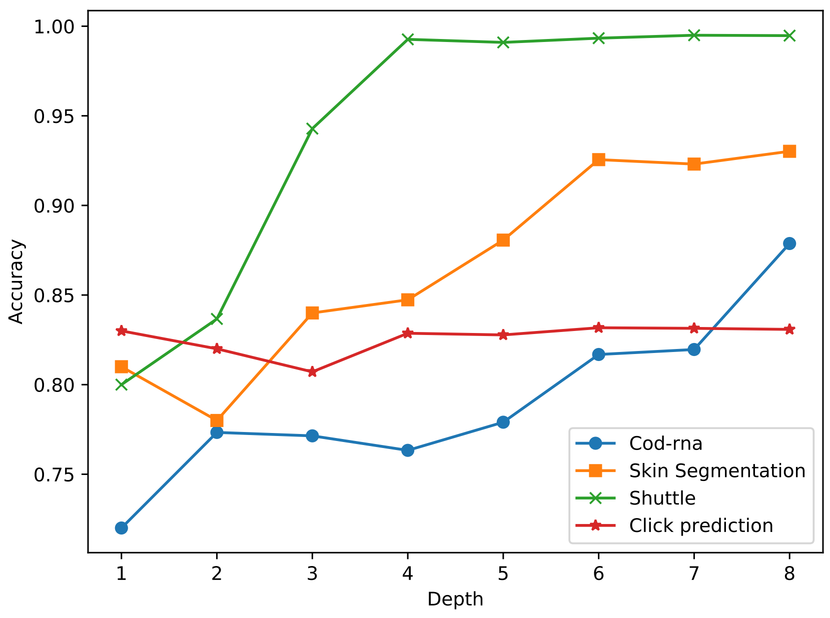

Figure 5.1: Accuracy by depth reached on training set.

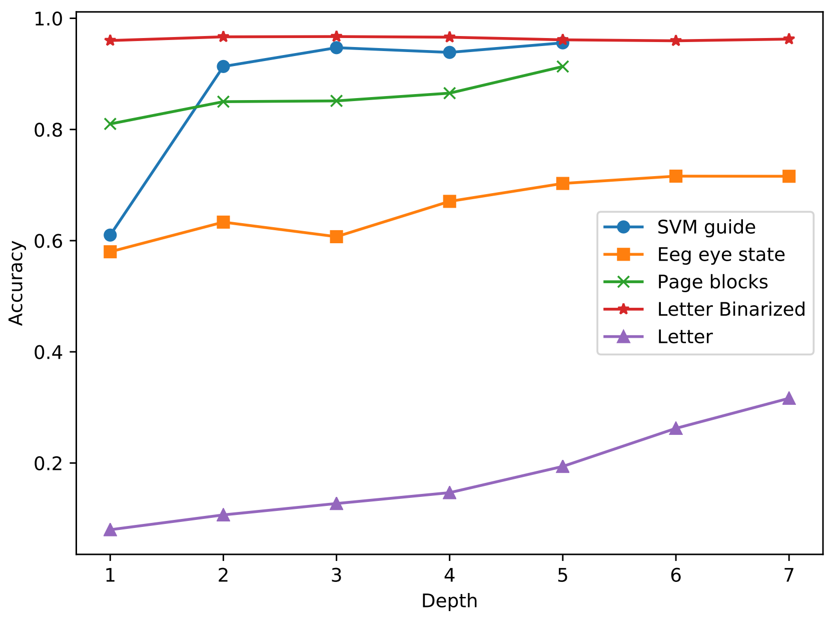

Figure 5.2: Accuracy by depth reached on training set.

5.4 Time tests

Table 5.6 presents the runtime for tests conducted in the tests corresponding to table 5.4. The table shows runtime for GSPDT for the building phase without memory handling in the first column, with memory handling in the second, and the time for executing CPU-SPDT in the third. Comparing the two last columns, GSPDT is up to 113 times faster as can be seen for the Eeg eye state dataset, with a mean speed increase of 13.

Dataset	GSPDT (Train)	GSPDT (Program)	CPU-SPDT
Click-prediction-small	1.5985	2.0052	18.565
Cod-rna	1.7096	2.1936	89.017
Eeg-eye-state	0.47445	0.91115	102.979
Letter	12.386	12.890	20.607
Letter Binarized	1.0373	1.3669	15.116
Page-blocks	0.33094	0.59793	15.444
Shuttle	3.0313	3.4871	33.545
Skin Segmentation	1.8844	2.2405	21.826
SVM Guide	0.14735	0.39894	5.055
Mean Time	2.5189	2.8990	35.795

Table 5.6: Time for training, 100 samples per node. Time in seconds.

In order to investigate where bottlenecks lie in the implementation, how much time is spent in each kernel is investigated. The runtime is separated into time spent in each kernel during tree building. Three datasets with diverse statistics are chosen to illustrate the behavior. The results are presented in table 5.7.

Kernel	Cod-rna	Page Blocks	Shuttle
navigate_samples	0.0613	0.0181	0.0035
histogram_update	94.0529	68.651	67.273
histogram_merge_on_workers	2.2314	4.4102	4.1509
histogram_merge_on_classes	1.1585	22.896	25.441
histogram_uniform	0.6451	0.4936	0.2499
calculate_gain_deltas	0.0613	0.2252	0.0667
split_nodes	0.0043	0.01353	0.0026
cubSegmentedRadixSort	1.8399	3.2911	2.8115

Table 5.7: Percentage of time spent in each kernel.

From table 5.7 it is clear that most time is spent updating and merging the histograms, as well as time spent sorting. The kernels doing this are *histogram_update*, *histogram_merge_on_workers*, and *histogram_merge_on_classes*, as well as sorting with CUB. Only the three first kernels are investigated, since the last is library dependent. These kernels are investigated in regard to how they scale with their dependent parameters. *histogram_merge_on_workers* and *histogram_merge_on_classes* are implemented identically, the former scales on number of workers used and the latter on number of classes used. Only *histogram_merge_on_workers* is shown below, the user has free control over the number of workers to use and it is thus easier to analyze how the kernels scales on workers rather than classes. *histogram_merge_on_workers* scales on $bins \cdot workers$, it is therefore enough to investigate how changing the amount of workers impact the algorithm, while keeping the number of bins constant. The result is illustrated in figure 5.3. *histogram_update* is tested against workers seen in figure 5.4, as well as samples used per node as seen in figure 5.5. A final test is conducted to investigate how the total runtime of the implementation behaves when increasing the number of workers, seen in figure 5.6

The kernels *histogram_update* and *histogram_merge_on_workers* are further investigated using the CUDA profiler NVVP. Analyzing the *histogram_update* kernel demonstrates that the number of threads used per block is small, with a thread occupancy of 1.6% when testing the Cod-rna dataset. The *histogram_merge_on_workers* kernel has an occupancy of 99%, but the thread execution efficiency is around 10%, which represents how much the threads must wait for other threads due to synchronization or branching computations.

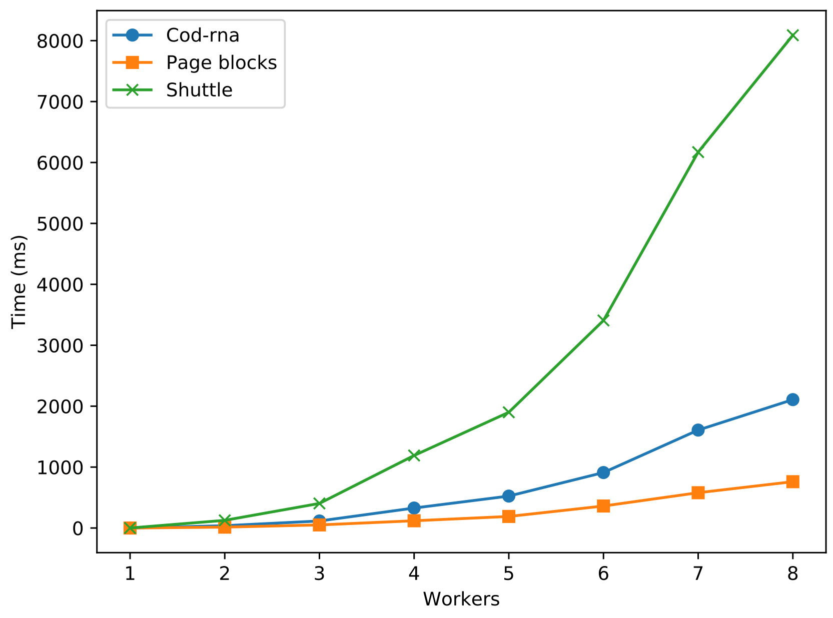

Figure 5.3: Merging on workers, time in milliseconds.

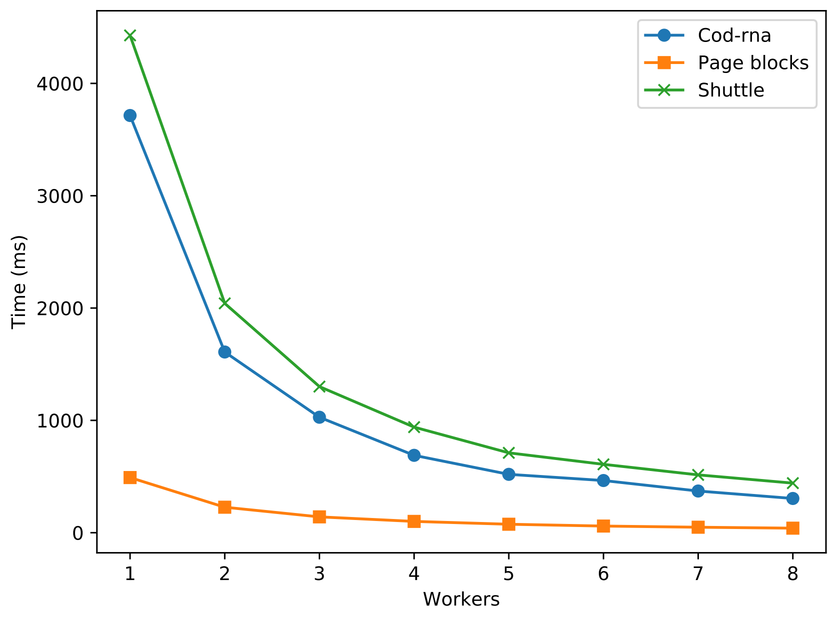

Figure 5.4: Update time scaling on workers, time in milliseconds.

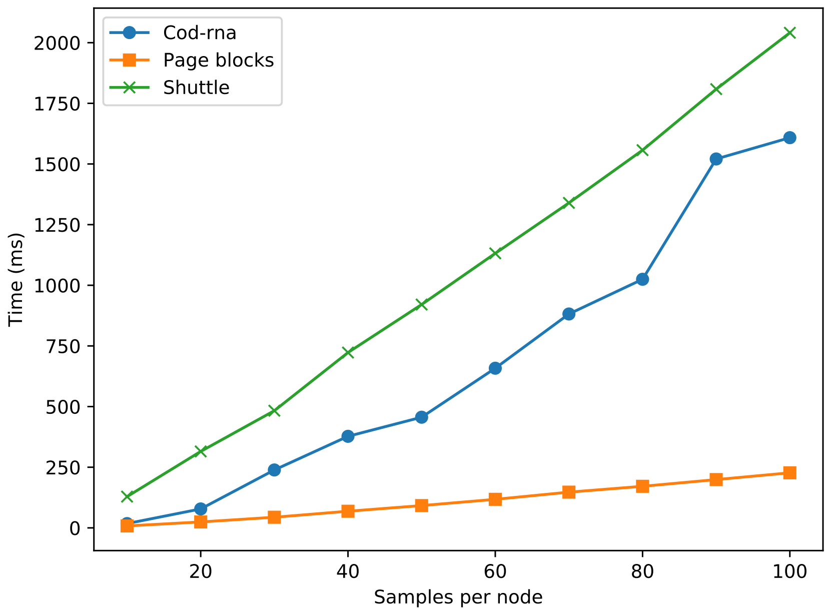

Figure 5.5: Update time scaling on samples per node, time in milliseconds.

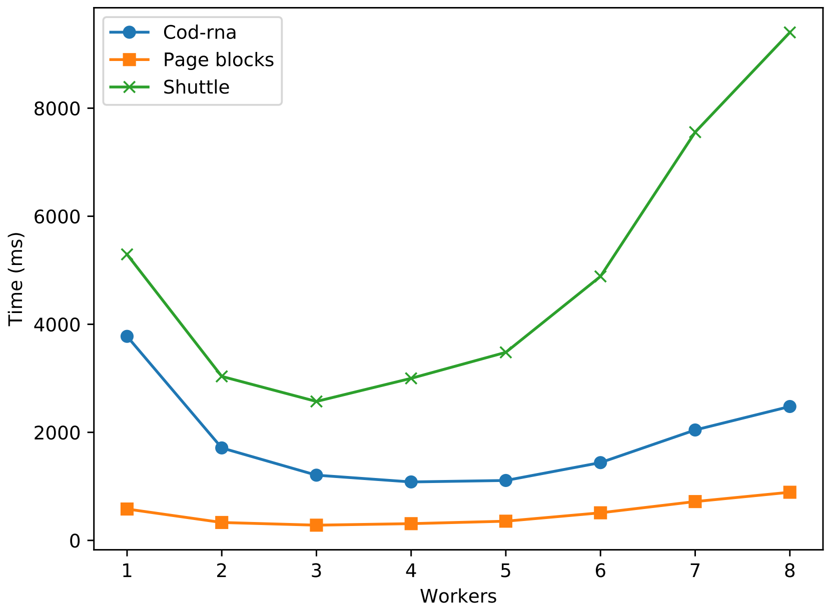

Figure 5.6: Total time of building tree scaling on workers, time in milliseconds.

6 Discussion and future work

In this chapter the achieved accuracy and runtime is discussed. This thesis also aims to find improvements to the current implementation, which is included in this discussion section.

6.1 Accuracy

The results in table 5.4 demonstrate that the mean accuracy of the GSPDT is similar to CPU-SPDT, and illustrates that the implementation is working as intended. Some datasets perform worse on GSPDT such as Cod-rna and Skin Segmentation, while SVM Guide performs well in comparison to CPU-SPDT. It is important to consider how the algorithm parameters can affect the performance. As investigated in the original paper of SDPT, performance can vary depending on number of workers used. Variables such as number of bins used, and uniform points used, matter as well. This is due to the unpredictability of the histograms created; a different order of merging the histograms from different workers for example can create substantially differing histograms and end results.

This implementation adds another parameter, samples used per node. Table 5.5 illustrates the importance of this parameter. Many datasets perform worse, largely due to the decrease in depth of the tree when doubling the number of samples used per layer. The result of the dataset Skin Segmentation illustrates how accuracy can be improved for certain datasets when using a higher number of samples per layer, even when decreasing depth of the tree learned. GPSDT uses only a subset of the entire dataset in each layer compared to the CPU-SPDT implementation which uses the entire dataset each iteration. Using a statistically rigorous framework to select samples used per layer should be investigated in future iterations of the implementation, such as utilizing the Hoeffding bound as in VFDT [21].

Figure 5.1 and 5.2 illustrates that the implementation is generally improving in accuracy for each depth in the tree, up to a certain point. The Worst performing dataset is Letter. In general datasets containing more classes perform worse. It should be mentioned that a depth of 7 is only used for training the dataset, which contains 26 classes. Figure 5.2 illustrates that accuracy is indeed increasing for each iteration. Investigations into using deeper trees for complicated datasets should be conducted in future work.

6.2 Time analysis and kernel improvements

General runtime for each tested dataset is presented in table 5.6. Comparisons to the CPU-SPDT implementation are conducted in order to ensure GSPDT is in line with expected runtime, and is used as a baseline. CPU-SPDT is implemented in Scala, and uses a slightly different approach to tree building. The comparison does demonstrate that GSPDT is faster in each test, and up to 113 faster in the case of the Eeg eye state dataset.

The kernel runtime investigations demonstrate that a clear majority of the time in building the tree is spent in updating the histograms, as well as in the merging procedures. The problem stems from inherent problems in the original algorithm; the bins in both the update and merge procedures are merged in an iterative manner. The worker merge runtime increases heavily when increasing the number of workers as seen in figure 5.3, while the class merging runtime increases heavily when increasing the number of classes in the dataset as can be seen by comparing Cod-rna (2 classes) to Shuttle (9 classes) in table 5.7. The update procedure decreases greatly in runtime with added workers; as can be seen in 5.4. Update also increases linearly in runtime with samples per node used. Figure 5.6 demonstrates how adding more workers decreases runtime of the entire algorithm up until a certain point. This demonstrates that in the current implementation workers added decreases runtime of update more than it increases runtime of the merging of worker histograms if the number of workers is not too high. This problem is important to solve in future implementations, as the algorithm should perform better when more workers are added.

Analyzing *histogram_update* kernel with NVVP demonstrates that the thread occupancy is small, at 1.6%. The kernel scales on number of features for the dataset used. Currently one block is used per worker, and each worker updates all feature histograms for one sample at a time. This is poor for datasets with a small number of features. To increase occupancy a similar approach as to the *split_nodes* kernel can be taken. A static number of threads for each block can be set, and the number of blocks spawned dependent on the number of total features for all workers combined, setting blocks to $\frac{(workers \cdot features)}{128} + 1$ if max threads per SM is 128.

Analyzing the *histogram_merge_on_workers* kernel in NVVP demonstrates an occupancy of 99%. The slow execution speed is evident from the thread execution efficiency being below 10%. This wait is due to only thread 0 in each block being responsible for finding the closest bins, and shuffling the array elements. The work being performed by thread 0 can be optimized by letting all threads work together. Finding the closest bins could be performed using CUB's block reduction features, which lets users define block or warp reductions inside a kernel [46]. Using this, each block can perform a reduction by allowing all threads do the bin comparison for finding the two closest bins together. The shuffling procedure can be conducted using CUB's warp shuffle modules [47]. These modules enable threads to shuffle down data; each thread writes to a thread of lower thread id and reads data from a thread with a higher thread id. Implementing the above should increase kernel performance. The same solution can be used to increase performance of the *histogram_merge_on_classes* kernel.

6.3 Ethics and Sustainability

It is important to highlight issues of ethics and sustainability in regard to any topic within machine learning. With regards to ethics, very technical aspects of implementing decision trees in a streaming data environment is investigated. As such, nothing in the thesis itself touches upon topics where ethics are valid to discuss. However, as decision trees are widely used within machine learning, care must be taken when using this or similar algorithms. The range of uses are endless, for example from using it within the field of medicine to using it to analyze individuals' private data [1]. With regards to sustainability, the topic of this thesis is to the increase performance of an existing algorithm by using dedicated hardware. If the speed of the algorithm can be increased, it will hopefully reduce the energy consumption needed for calculations. Future research of this algorithm should investigate power consumption of the GPU, and compare it to existing algorithms on CPU.

7 Conclusions

This study details a first implementation of SPDT on GPUs. A choice is made to build the tree layer by layer, using a specified number of samples per layer. It is demonstrated that the accuracy of the implementation was close to that of the CPU version, which uses the entire dataset in each layer. This is an interesting finding which opens up possibilities for parallelizing decision trees in a streaming data environment. Future research should be conducted to construct a statistically rigorous approach to choosing correct number of nodes to use per layer, investigating if it is competitive with other approaches.

Performance in terms of speed is shown to be beneficial compared to the CPU implementation. The GPU version is up to 113 times faster than the reference CPU version for one dataset, with an average increase of 13 over all the tested datasets. A weakness of the implementation was found to be the parallelization of updating and merging histograms for workers and classes. Techniques to solve the problem in future iterations are found, which could lead to a better performing algorithm.

Bibliography

[1] Michael I Jordan and Tom M Mitchell. “Machine learning: Trends, perspectives, and prospects”. In: *Science* 349.6245 (2015), pp. 255–260.

[2] Dilpreet Singh and Chandan K Reddy. “A survey on platforms for big data analytics”. In: *Journal of Big Data* 2.1 (2015), p. 8.

[3] Albert Bifet and Richard Kirkby. “Data stream mining a practical approach”. In: (2009).

[4] Lior Rokach. “Decision forest: Twenty years of research”. In: *Information Fusion* 27 (2016), pp. 111–125.

[5] Yael Ben-Haim and Elad Tom-Tov. “A streaming parallel decision tree algorithm”. In: *Journal of Machine Learning Research* 11.Feb (2010), pp. 849–872.

[6] Gareth James et al. *An introduction to statistical learning*. Vol. 112. Springer, 2013.

[7] Diego Marron, Albert Bifet, and Gianmarco De Francisci Morales. “Random Forests of Very Fast Decision Trees on GPU for Mining Evolving Big Data Streams.” In: *ECAI*. Vol. 14. 2014, pp. 615–620.

[8] Ugljesa Milic et al. “Parallelizing general histogram application for cuda architectures”. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*. IEEE. 2013, pp. 11–18.

[9] Karl Jansson, Håkan Sundell, and Henrik Boström. “gpuRF and gpuERT: efficient and scalable GPU algorithms for decision tree ensembles”. In: *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE. 2014, pp. 1612–1621.

[10] Wei-Yin Loh. “Fifty years of classification and regression trees”. In: *International Statistical Review* 82.3 (2014), pp. 329–348.

[11] Leo Breiman et al. *Classification and regression trees*. CRC press, 1984.

[12] J. Ross Quinlan. “Induction of decision trees”. In: *Machine learning* 1.1 (1986), pp. 81–106.

[13] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[14] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Vol. 1. Springer series in statistics New York, 2001.

[15] Chandrika Kamath, Erick Cantú-Paz, and David Littau. "Approximate splitting for ensembles of trees using histograms". In: *Proceedings of the 2002 SIAM International Conference on Data Mining*. SIAM. 2002, pp. 370–383.

[16] *CUDA Programming Guide*. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Accessed: 2018-05-23.

[17] Ponkiya Parita and Purnima Singh. "A Review on Tree Based Incremental Classification". In: *Ponkiya Parita et al, International Journal of Computer Science and Information Technologies* 5.1 (2014), pp. 306–309.

[18] Jeffrey C Schlimmer and Douglas Fisher. "A case study of incremental concept induction". In: *AAAI*. Vol. 86. 1986, pp. 496–501.

[19] Paul E Utgoff. "Incremental induction of decision trees". In: *Machine learning* 4.2 (1989), pp. 161–186.

[20] Paul E Utgoff. "An improved algorithm for incremental induction of decision trees". In: *Machine Learning Proceedings 1994*. Elsevier, 1994, pp. 318–325.

[21] Pedro Domingos and Geoff Hulten. "Mining high-speed data streams". In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2000, pp. 71–80.

[22] Geoff Hulten, Laurie Spencer, and Pedro Domingos. "Mining time-changing data streams". In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2001, pp. 97–106.

[23] Biswanath Panda et al. "Planet: massively parallel learning of tree ensembles with mapreduce". In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1426–1437.

[24] Mahesh V Joshi, George Karypis, and Vipin Kumar. "ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets". In: *Parallel processing symposium, 1998. IPPS/SPDP 1998. proceedings of the first merged international... and symposium on parallel and distributed processing 1998*. IEEE. 1998, pp. 573–579.

[25] Balaji Lakshminarayanan, Daniel M Roy, and Yee Whye Teh. "Mondrian forests: Efficient online random forests". In: *Advances in neural information processing systems*. 2014, pp. 3140–3148.

[26] John Shafer, Rakesh Agrawal, and Manish Mehta. "SPRINT: A scalable parallel classi er for data mining". In: *Proc. 1996 Int. Conf. Very Large Data Bases*. Citeseer. 1996, pp. 544–555.

[27] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. "SLIQ: A fast scalable classifier for data mining". In: *International Conference on Extending Database Technology*. Springer. 1996, pp. 18–32.

[28] Khaled AlSabti, Sanjay Ranka, and Vineet Singh. "Clouds: Classification for large or out-of-core datasets". In: *Conference on Knowledge Discovery and Data Mining*. 1998.

[29] Ruoming Jin and Gagan Agrawal. "Communication and memory efficient parallel decision tree construction". In: *Proceedings of the 2003 SIAM International Conference on Data Mining*. SIAM. 2003, pp. 119–129.

[30] Håkan Grahn et al. "CudaRF: a CUDA-based implementation of random forests". In: *Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on*. IEEE. 2011, pp. 95–101.

[31] Win-Tsung Lo et al. "CUDT: a CUDA based decision tree algorithm". In: *The Scientific World Journal* 2014 (2014).

[32] Aziz Nasridinov, Yangsun Lee, and Young-Ho Park. "Decision tree construction on GPU: ubiquitous parallel computing approach". In: *Computing* 96.5 (2014), pp. 403–413.

[33] Huan Zhang, Si Si, and Cho-Jui Hsieh. "GPU-acceleration for Large-scale Tree Boosting". In: *arXiv preprint arXiv:1706.08359* (2017).

[34] Jung Ho Ahn, Mattan Erez, and William J Dally. "Scatter-add in data parallel architectures". In: *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE. 2005, pp. 132–142.

[35] Wookeun Jung, Jongsoo Park, and Jaejin Lee. "Versatile and scalable parallel histogram construction". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM. 2014, pp. 127–138.

[36] T. Rantalaiho. *Generalized Histograms for CUDA-capable GPUs*. `https://github.com/trantalaiho/Cuda-Histogram/`. Accessed: 2018-03-06. 2011.

[37] Ramtin Shams, RA Kennedy, et al. "Efficient histogram algorithms for NVIDIA CUDA compatible devices". In: *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*. 2007, pp. 418–422.

[38] *GPU Pro Tip: Fast Histograms Using Shared Atomics on Maxwell*. `https://devblogs.nvidia.com/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/`. Accessed: 2018-02-26.

[39] *NVIDIA Visual Profiler website*. `https://developer.nvidia.com/nvidia-visual-profiler/`. Accessed: 2018-04-23.

[40] *CUB*. `https://nvlabs.github.io/cub/`. Accessed: 2018-04-23.

[41] *Soundcloud SPDT*. `https://github.com/soundcloud/spdt`. Accessed: 2018-05-23.

[42] *CUB Segmented modules*. `https://nvlabs.github.io/cub/group___segmented_module.html`. Accessed: 2018-05-23.

[43] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: a library for support vector machines". In: *ACM transactions on intelligent systems and technology (TIST)* 2.3 (2011), p. 27.

[44] *LIBSVM Data*. `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`. Accessed: 2018-04-23.

[45] Joaquin Vanschoren et al. "OpenML: Networked Science in Machine Learning". In: *SIGKDD Explorations* 15.2 (2013), pp. 49–60. DOI: `10.1145/2641190.2641198`. URL: `http://doi.acm.org/10.1145/2641190.2641198`.

[46] *CUB BlockReduce*. `https://nvlabs.github.io/cub/classcub_1_1_block_reduce.html`. Accessed: 2018-05-23.

[47] *CUB Warp-wide modules*. `https://nvlabs.github.io/cub/group___warp_module.html`. Accessed: 2018-05-23.