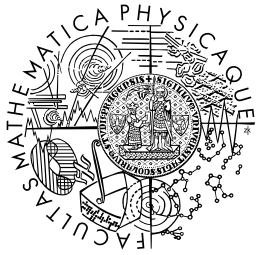


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

František Haas

Parallel Processing of Huge Astronomical Data

Department of Software Engineering

Supervisor of the master thesis: RNDr. Filip Zavoral, Ph.D.
Study programme: Software Systems
Study branch: Dependable Systems

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Parallel Processing of Huge Astronomical Data

Author: František Haas

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D., Department of Software Engineering

Abstract: This master thesis focuses on the Random Forests algorithm analysis and implementation. The Random Forests is a machine learning algorithm targeting data classification. The goal of the thesis is an implementation of the Random Forests algorithm using techniques and technologies of parallel programming for CPU and GPGPU and also a reference serial implementation for CPU. A comparison and evaluation of functional and performance attributes of these implementations will be performed. For the comparison of these implementations various data sets will be used but an emphasis will be given to real world data obtained from astronomical observations of stellar spectra. Usefulness of these implementations for stellar spectra classification from the functional and performance view will be performed.

Keywords: machine learning, random forests, data mining, parallel programming

I would like to sincerely thank to RNDr. Filip Zavoral, Ph.D. for his time, guidance, ideas and uttermost patience.

I also thank to RNDr. Martin Kruliš, Ph.D. for his valuable help and assistance.

Contents

1 Introduction **3**

2 Random Forests **5**
2.1 Supervised learning 5
2.2 Decision tree 5
2.3 Ensemble learning 5
2.4 Algorithm 6
2.5 Entropy and Information Gain 7
2.5.1 Sample set entropy and information gain example 7

3 Parallel programming **10**
3.1 OpenCL 10
3.1.1 Kernel and work item 10
3.1.2 Work group 12
3.1.3 Memory 13
3.2 CUDA 13
3.2.1 Kernel 13
3.2.2 Compute capability 13
3.2.3 Block 14
3.2.4 Warp 14
3.2.5 Memory 14
3.2.6 Coalesced memory access 14
3.2.7 Stream 14
3.3 OpenCL or CUDA 15

4 OpenCL implementation **16**
4.1 Complexity 16
4.1.1 Bootstrap draw 16
4.1.2 Tree node training 16
4.1.3 Complete tree training 17
4.2 Input arguments 17
4.3 Implementation overview 18
4.4 Data structures 18
4.5 Split point evaluation 19
4.6 Implementation 19
4.7 Bootstrap size 20
4.8 Evaluation 20
4.8.1 Synthetic data generator 20
4.8.2 Comparison 22
4.9 Summary 23

5 Fine-grained CUDA implementation **24**
5.1 Input arguments 24
5.2 Data structures 25
5.3 Implementation 27

5.4 Label count frequency . . . 27
5.5 Data load . . . 28
5.6 Sort . . . 28
5.6.1 CUDA Thrust . . . 28
5.6.2 CUB . . . 29
5.6.3 Bitonic sort . . . 29
5.6.4 Sort algorithms comparison . . . 30
5.7 Best information gain . . . 32
5.8 Summary . . . 35

6 Monolithic CUDA implementation 37
6.1 Input arguments . . . 37
6.2 Data structures . . . 37
6.3 Implementation . . . 38
6.4 Summary . . . 40

7 Classification 42
7.1 Training set size . . . 42
7.2 Tree count and bootstrap size . . . 42

8 Related work 45
8.1 CudaTree . . . 45
8.2 CUDA is not meant for training Random Forests . . . 45
8.3 Analytical programming and Random Forests . . . 45
8.4 Summary . . . 46

9 Conclusion 47
9.1 Future work . . . 47
9.1.1 In-place bitonic sort for arbitrary sized arrays . . . 47
9.1.2 Algorithm restructuralization . . . 48
9.1.3 Hybrid solution . . . 48

Bibliography 49

Appendices 51

A Contents of the enclosed CD 52

B Programming documentation 53

C User documentation 56

1. Introduction

Motivation for this diploma thesis is a problem of stellar spectra classification. There are different types of stars with various properties. Each star type produces a particular spectral signature. The task of stellar spectra classification is to distinguish these signatures and link them to correct star types. Astronomers acquire and collect extensive amounts of stellar spectra observations. As the number of observations is growing, manual data processing and star classification is not feasible anymore. Therefore, it is necessary to find effective methods for automation of the process as stellar spectra classification is an important part of astronomical research.

Generally, tasks in information technology are solved using finely tailored algorithms. These algorithms systematically process input data and converge them to an expected result. Designing such an algorithm requires deep knowledge of the input data, the desired output and all required steps in between. However, complications arise when it is not effectively possible to express all steps in an explicit algorithm. Let us take as an example a problem of optical recognition of printed characters captured by a digital camera. It can be hardly imagined how to efficiently tailor a well performing algorithm to recognize all characters when quality, skew or light conditions may vary a lot. It would be even harder to modify such an algorithm for a new font or symbol if needed.

In such a case, a better way to solve this task is to leave the recognition of important characteristics and relations in the data to a more general algorithm. Machine learning algorithms are capable of that. There is a lot of different machine learning algorithms and even more various implementations of each of them. The fundamental idea of machine learning algorithm is to learn from training data. Machine learning algorithms learn by finding characteristics and relations in input data important with respect to desired output values.

The quantity and quality of data used to train machine learning algorithms is very important for the performance of a trained algorithm on the real world data. However, the evaluation of data quality is done by a machine learning algorithm itself and it often needs to filter and process huge quantities of data to find out what is and what is not important for good the performance of the resulting model. Therefore, it would be very useful to have in hand an implementation that can cope with input data efficiently and produce results fast.

Recently, it has been possible in information technology to utilize various heterogeneous devices for high performance computing. For instance, current generations of GPUs can be used for general purpose computations. It is therefore appealing to implement machine learning algorithms on such devices and utilize large amounts of simple processing units capable of logical and arithmetic operations.

This diploma thesis explores possibilities of effective machine learning implementations for various hardware and software platforms.

Specifically, it deals with the Random Forests machine learning algorithm. Essentially, this algorithm selects and decides what actions to perform by calculating

and evaluating the information gain these actions would provide. It is an interesting algorithm and it bears certain properties making it an intriguing candidate for parallelization. Namely, a major section of its execution can be divided into completely independent parts, which is a very appealing property, and the performance and behaviour of the algorithm is given, apart from some other, by two input properties, whose both nominal increase can potentially improve algorithm's performance, while technically one of them influences the number of executed parts and the other determines how complex these parts are.

This thesis evaluates implementations of the algorithm on multiple platforms. It deals with CPU, OpenMP CPU, OpenCL GPU and CUDA GPU implementations. Open Multi-Processing (OpenMP) is a multi-platform API for parallel shared-memory programming [9]. Open Computing Language (OpenCL) is a framework for writing parallel applications executing across heterogeneous platforms [3]. Compute Unified Device Architecture (CUDA) is a parallel programming platform and API framework tailored for Nvida GPUs [10].

The second section describes Random Forests algorithm in detail. The third section depicts technical aspects and essentials of OpenCL and CUDA frameworks and pinpoints fundamental differences compared with general CPU programming. The fourth section outlines complexity of Random Forests and describes a concept OpenCL implementation and evaluates its characteristics. The fifth section describes and assesses aspects of a fine-grained CUDA implementation. The sixth section provides details and evaluation of a monolithic CUDA implementation. The seventh section measures algorithm's classification performance on various data. The last section summarizes the thesis and discusses future and related work.

2. Random Forests

Random Forests [1] is a machine learning algorithm. Such algorithm is usually not designed to solve only a single specific problem but it is crafted to be able to solve a broader group of problems. For instance, it can be utilized to perform classification or regression tasks. Classification covers a complex class of concrete problems. It contains, already mentioned optical character recognition, recognition of speech, people and also stars. Regression carries out value predictions. It may be used for stock price prognosis, weather forecasts and so on.

2.1 Supervised learning

There are supervised and unsupervised machine learning algorithms. In classification scope, supervised learning requires a set of data labelled with desired classes. For instance, a set of character images labelled with corresponding letters can be considered a data set suitable for optical recognition by supervised learning. In contrast, unsupervised learning does not require any labelled data. Such an algorithm is left to recognize important differences in input data and distinct different classes on its own. Random Forests is a supervised machine learning algorithm.

As mentioned, the goal of the supervised machine learning algorithm is to recognize important relations and patterns in input data samples' characteristics with respect to labelling and create a model that describes these associations. Characteristic, in case of an image of a letter, is for instance a colour of pixel on specific coordinates, a height of the image, an average tone of the image and so forth. Machine learning algorithms usually require sample data to share a common form. For example, images must be scaled to a certain size. Often, input data for machine learning algorithms is preprocessed by domain specific tools capable of data enrichment and addition of more useful attributes than originally present.

2.2 Decision tree

A decision tree is a structure made of binary tests (split nodes). The binary test is a function that accepts the value of a certain required attribute (characteristic). The binary test can result in another binary test or a final decision (final leaf node).

2.3 Ensemble learning

This algorithm utilizes principles of ensemble learning. The core of this method is to train multiple predictor models and combine them into a final one. These

models can be, in fact, fairly primitive as they do not have to solve the problem across whole data sets but just over a specific subset of it. When used, the aggregated model evaluates every partial model to retrieve its result and then uses an aggregate function to produce final results.

For the Random Forests algorithm, the partial predictor model is called a tree, or a decision tree. These trees are completely independent on each other. The subset of data the tree model handles is random. In order to combine partial results from all trees the Random Forests algorithm chooses the most frequent result as the final result.

2.4 Algorithm

Random Forests definition follows [1, p. 588]

Algorithm 2.4.1 (Random Forests)

- *N number of trees to create.*
- *K bootstrap size to train every tree on.*
- *L minimum bootstrap size to stop training at*

1. *For $n = 1$ to N:*
   (a) *Draw a random bootstrap B^* of size K from the training data.*
   (b) *Grow a random forests tree T_n to the bootstrapped data, by recursively repeating following steps for each terminal node of the tree, until the minimum node size L is reached.*
      i. *Select M variables at random from the P variables.*
      ii. *Pick the best variable/split-point among the M.*
      iii. *Split the node into two daughter nodes.*
2. *Output the ensemble of trees T_{n1}^{N}*

To make a prediction at a new point x:

Let every tree predict a class on its own and the result class is based on majority vote of all trees.

Bootstrap is a collection of samples generated uniformly with repetition from a training data set. Every sample has an equal probability to be selected on every pick and might appear in the bootstrap multiple times. Bootstrap size is configured on run-time.

During the decision tree training every time a split is made the bootstrap is split in two. It means that the quantity of samples a node is trained on is decreasing while the tree is growing to its leafs. It is obvious to stop the training process when all remaining samples belong to the same class. However, sometimes the training process continues until the sample collection becomes very small but no consensus is found yet. It is questionable how long the process should go on

and what can be gained if splitting happens for just few samples. In such case, the process is more likely a memorization of presented samples than a training. Model growths but the gain is doubtful. Thus, it might be preferable to stop the process and instead of extending the process until only a single class is present among remaining samples, the most dominant class present is used.

Another important property of the algorithm is the number M of attributes to be tested and searched for the split-point. Best split point search is the core of the Random Forests algorithm.

2.5 Entropy and Information Gain

In order to identify the best possible split information gain metric is used to evaluate and rank tested split points. Information gain is based on entropy comparison between the original integral bootstrap and weighted entropies of divided bootstrap parts.

Entropy [1] describes the measure of disorder in a system. For discrete magnitudes, entropy can be expressed as a logarithmic measure of all possible states and the probability of their occupation.

$$E = -\sum_i (p_i * \log_2(p_i))$$

Algorithm 2.5.1 (entropy)

1. *S is list of samples to compute entropy for*
2. *let N be the number of classes represented in S, class is represented in S if exists at least one sample in S that belongs to the class*
3. *let $< c_i >_1^N$ be vector representing the number of occurrences for every class present*
4. *then entropy $E = -\sum p_i * log_2(p_i)$, where $p_i = \frac{c_i}{|S|}$*

Algorithm 2.5.2 (information gain)

1. *S is list of samples to be split, SP is the split point*
2. *compute entropy E of S*
3. *split S according to SP into S_L and S_R*
4. *compute entropy E_L and E_R of S_L and S_R*
5. *$G = E - (E_L * (|S_L|/|S|) + E_R * (|S_R|/|S|))$*

2.5.1 Sample set entropy and information gain example

Let there be 12 points belonging to two distinct groups, blue and red, in two dimensional space in the form of a square. (see Figure 2.1).

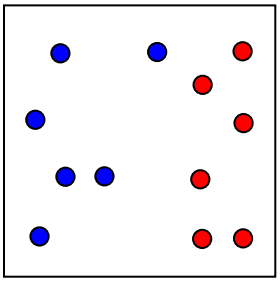
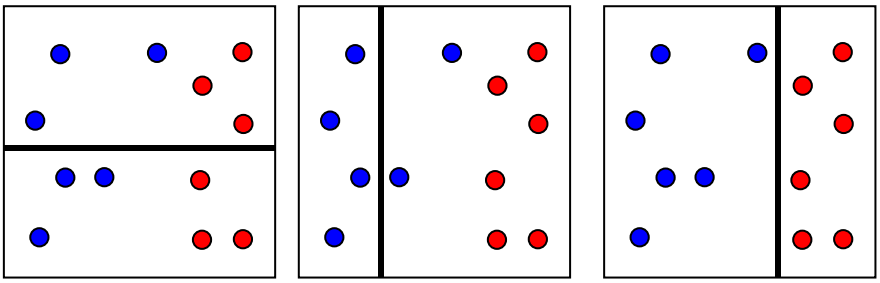


Figure 2.1: Square S of all samples



(a) $S_a^1 up, S_a^2 down$ (b) $S_b^1 left, S_b^2 right$ (c) $S_c^1 left, S_c^2 right$

Figure 2.2: Possible divisions of square S

Entropy (see Algorithm 2.5.1) for point distribution among various groups in the square is equal to 1 (see Equation 2.1).

$$\begin{aligned} p_0 &= 6/12, \text{ 6 red} \\ p_1 &= 6/12, \text{ 6 blue} \\ E_S &= -((0.5 * \log 2(0.5) + (0.5 * \log_2(0.5)) \\ E_S &= -((0.5 * -1) + (0.5 * -1)) \\ E_S &= 1 \end{aligned} \tag{2.1}$$

There are many possible ways to split this square into two parts.

(a) It can be divided into two identical rectangles (S_a^1 and S_a^2) both composed of 3 blue and 3 red elements. Entropy is the same for both rectangles, specifically $E_{S_a^1} = 1, E_{S_a^2} = 1$ (see Figure 2.2(a)).

(b) Another option is to divide the original set S into two distinct parts, one (S_b^1) consisting of 4 blue elements, the other (S_b^2) of 2 blue and 6 red elements. Entropy for each of them is different, namely $E_{S_b^1} = 0, E_{S_b^2} = 0.81$ (see Figure 2.2(b)).

(c) Yet another option is to split the original set S into one part (S_c^1) made only of all blue elements and the other (S_c^2) of all red elements. The entropy of both rectangles is equal to $E_{S_c^1} = 0, E_{S_c^2} = 0$ (see Figure 2.2(c)).

$$
\begin{aligned}
E_S &= 1 \\
E_{S_a^1} &= 1 \\
E_{S_a^2} &= 1 \\
|S_a^1|/|S| &= 6/12 \\
|S_a^2|/|S| &= 6/12 \\
G_a &= E_S - (E_{S_a^1} * (|S_a^1|/|S|) + E_{S_a^2} * (|S_a^2|/|S|)) \\
G_a &= E_S - (1 * 0.5 + 1 * 0.5) \\
G_a &= 0
\end{aligned}
\tag{2.2}
$$

Information gain of the first demonstrated division S_a is equal to 0 (see Equation 2.2). Therefore, such a split is not in any way interesting for a machine learning algorithm as it would not likely add any value to the model.

$$
\begin{aligned}
G_b &= 1 - (0 * 4/12 + 0.81 * 8/12) \\
G_b &= 0.54
\end{aligned}
\tag{2.3}
$$

The division S_b has a positive information gain (see Equation 2.3) and therefore seems to be a better candidate for a split point in a decision tree.

$$
\begin{aligned}
G_c &= 1 - (0 * 6/12 + 0 * 6/12) \\
G_c &= 1
\end{aligned}
\tag{2.4}
$$

Clearly, the last example of illustrated possible divisions has the highest information gain (see Equation 2.4) of all. It is clear from the figure that this division splits the original square S into a rectangle of only blue points and another rectangle of just red points. This is the best split point to use for a decision tree node.

This example demonstrates one of the core ideas of Random Forests algorithm. The Random Forests is a classification algorithm, the data samples it works with possess certain attributes and belong to various classes. In the presented example, the samples are represented using elements in two-dimensional space, the classes are represented using distinct colours and the attributes, two in this case, are outlined using the coordinate system. There were 3 possible split points investigated, using both available attributes, and the information gain algorithm identified the best one of them.

3. Parallel programming

There is a broad range of various hardware platforms offering different features. There are central processing units (CPU), graphical processing units (GPU), configurable field-programmable gate arrays (FPGA) and digital signal processors (DSP). Each of them has a different philosophy and suites best a different set of tasks.

For instance, general control intensive applications can benefit from CPUs provided with data and code cache together with intelligent branch prediction support. Computation intensive applications targeting high throughput can profit from vector or many-core hardware that enables execution of many numerical instructions at once. Ultra low latency applications effectively utilize FPGAs.

3.1 OpenCL

Open Computing Language (OpenCL) [3] is a technology standard framework for writing parallel programs across heterogeneous computing platforms, possibly consisting of CPUs, GPUs, FPGAs and DSPs. OpenCL framework is logically defined in four parts, called models.

The platform model specifies there is one processor (host) in control of the execution and one or more co-processors (device) to distribute work on. It also defines an abstract hardware model for OpenCL representing the work, kernel (see Section 3.1.1).

The execution model defines how an OpenCL context is set up on the host and how are OpenCL kernels executed on the devices. It includes kernel concurrency model and host-device interactions.

The memory model describes an abstract memory hierarchy used on all execution devices. The abstract memory model is in general common for all device types. However, the actual mapping of memory model layers to physical memory architectures differs highly and so do various related performance properties. Size of distinct memory model layers might differ considerably across various target hardware platforms.

Programming model outlines how kernel code concurrency is mapped to physical devices.

3.1.1 Kernel and work item

Kernel is the central point of the OpenCL execution model. In detail, it is a C99 function prefixed with a kernel marker (see Figure 3.1) and its syntax is enhanced with special purpose OpenCL functions and keywords [3].

Kernel is implicitly expected to be executed by multiple threads. This is in contrast to standard programming practice (see Figure 3.2). The particular number

```
__kernel
 void kernel_increment( int[] values, int length) {
 values[get_global_id(0)]+= 1;
}
```

Figure 3.1: Increment array kernel

```
void increment( int[] values, int length) {
 for (int i = 0; i < n; i++) {
  values[i]+= 1;
 }
}
```

Figure 3.2: Increment array function

of threads is configured by the host. Each thread must be able to identify itself to be capable of distinguishing the part of work allocated for it - this is similar to the standard C fork function where a parent and a child process split paths by evaluating the return value of fork function. In an OpenCL kernel this purpose is served by `get_global_id(0)` function, which returns consequent integral numbers, starting from 0, unique for each thread executing an instance of the kernel (see Figure 3.1).

In the OpenCL terminology, kernel is not called but launched. Every kernel launch must specify a number of threads it will be processed by. Work item is the minimal unit of work in OpenCL, it is a term interchangeable with thread. Standard approach for specifying the number of work items, called range, is to mimic proportions of input or output data sets. Let there be a kernel (see Figure 3.1) whose function is to increment all elements in an array by one, and an array, whose values should be incremented. One possibility is to specify the range so that it fits the to-be-incremented array size (see Figure 3.3). Offset makes the work items identification start from an artificial number instead of the default zero. Group parameter specifies the size of work group. It is described in more detail in the following section. Data sets are not always limited to one dimensional arrays and to fit the purpose the range can spawn up to three dimensions.

Matrix summation (see Figure 3.4), for instance, can be expressed by two dimensional thread indexing. The parameter of the `get_global_id` function states the thread dimension in question. This solution is probably more expressive and readable compared to a custom one to two dimensional addressing.

```
cl::NDRange offset       = cl::NullRange;
cl::NDRange group        = cl::NullRange;
cl::NDRange range        = cl::NDRange( array_size);
commandQueue.enqueueNDRangeKernel(
        kernel_increment, offset, range, group);
```

Figure 3.3: Increment array kernel launch

```
__kernel
void kernel_sum( int[][] a, int[][] b, int[][] c) {
 int x = get_global_id(0);
 int y = get_global_id(1);
 c[x][y] = a[x][y] + b[x][y];
}
```

```
cl::NDRange range = cl::NDRange( matrix_height, matrix_width);
commandQueue.enqueueNDRangeKernel(
        kernel_sum, cl::NullRange, range, cl::NullRange);
```

Figure 3.4: Matrix summation

3.1.2 Work group

Work group [3] is an abstraction that groups possibly related work items where thread cooperation can be required and made more effective. It is tightly linked with architecture of underlying hardware platforms. There are more strict constraints for in work group item execution than for unrelated items. For instance, one item's result might be dependent on another item's result in the same work group while this does not necessary hold for items from different work groups.

An insight into a sample device architecture can provide the reasons for these constraints. Otherwise, one could wonder why not all items are executed in one big work group. NVIDIA Fermi [12] architecture Tesla M2090 [11] GPUs are used for the evaluation of implementations in this thesis and for this reason they are used for the explanation as well.

The Fermi architecture GPUs consist of 512 CUDA cores. The CUDA core is a processor with a fully pipelined logic unit (ALU), a floating point unit (FPU) and a set of private registers. These cores are organized in groups of 32 and together form a Streaming Multiprocessor (SMP). There are 16 SMPs on any Fermi GPU. All cores of a single SMP share 64KB of memory used either for in work group communication or as an L1 cache, 16 load/store units for cache and memory access and 4 Special Function Units (SFU). The ratio of explicitly controlled memory to L1 cache is configurable. The Fermi architecture supports up to 6GB of GDDR5 DRAM shared by all SMPs.

The Fermi architecture is capable of executing 2048 threads per SMP. In fact, there can be 32 threads running at once on a SMP, one on each CUDA core. Kernel code is being executed in a single instruction multiple thread (SIMT) manner. All threads executing on all 32 cores at the moment compute the same instruction concurrently. CUDA cores are capable of context switch under 25 microseconds. Therefore, the execution of a substantial number of threads frequently switching context on a multiprocessor may hide latencies of memory loads/stores or arithmetic operations.

A kernel launch requires a local and global range to be specified. The global range determines the total number of threads to be spawned, while the local range determines the number of threads allocated on a certain SMP in work group cooperation.

Work group cooperation allows threads to communicate more effectively using local memory instead of global memory and also gives them the possibility to synchronize. In contrast, communication between work groups is slower since only global memory might be used and synchronization is very tricky or even impossible in certain scenarios. Generally, there is no guarantee on the order and timing of work group scheduling to SMPs and as a result when all SMPs are fully utilized, active threads in all groups are waiting to synchronize with a group that has not been started yet, but then, the awaited group will not be scheduled at all.

3.1.3 Memory

There are three types of memory available in the OpenCL kernel [3]. Private memory is separate for every thread. Referring still to the Fermi architecture, it is allocated in device registers if provision is sufficient. Local memory is shared by all threads in a work group. Global memory is shared by all SMPs on the device. It can offer up to 6GB of available space.

Global memory is usually employed to store most of data structures needed for computation similarly to CPU main memory. Local memory is often used as an explicit cache for hot parts of global memory or for thread communication in work groups. Access to local memory is faster but it is usually much smaller in size compared with global memory.

3.2 CUDA

Compute Unified Device Architecture (CUDA) is a parallel programming platform and an API framework tailored for Nvida GPUs [10]. It resembles many OpenCL aspects closely.

Given the level of detail in Section 3.1, most mentioned terms can be considered generally equal in context of OpenCL and CUDA frameworks, with only certain differences in terminology.

3.2.1 Kernel

The CUDA kernel [10] is very similar to the OpenCL kernel. The philosophy holds but the syntax of CUDA API calls differ.

3.2.2 Compute capability

The compute capability is essentially a version of CUDA framework and it states what CUDA features and constraints are provided by a GPU given the compute capability it supports.

3.2.3 Block

A block is the equivalent of the OpenCL work group. It groups individual threads together and provides means of memory communication and certain execution guarantees. The Fermi architecture limits block size to 1024 threads [12].

3.2.4 Warp

There are 32 CUDA cores on Fermi architecture GPUs. When a block executes there are 32 threads of the block running at the same time on an SMP, one on each CUDA core [12]. These 32 threads are considered a warp. This aspect can be generally ignored as it does not affect functional correctness but it can be exploited for performance optimizations. Namely, it can be used to improve memory access throughput, atomic operation scheduling or there are special intrinsic instructions for in-warp thread communication [13].

3.2.5 Memory

Fundamentally, there are three memory layers similar to the OpenCL memory model. There is a private memory dedicated for thread-only operations. There is a shared memory (OpenCL local memory) common for all threads in block. There is a global memory shared across all blocks and the host.

3.2.6 Coalesced memory access

Coalesced memory access is a very important performance consideration of a CUDA capable GPU architecture [10]. Stores and loads of threads within a single warp accessing the global memory can be coalesced into only single memory transaction by the device if certain conditions are met.

In the simplest scenario, if all threads in a warp access adjacent 4-byte words there is only single memory transaction necessary. However, if consecutive threads access sequential 4-byte words but the memory is not aligned with cache lines there are two memory transactions performed.

3.2.7 Stream

Fermi architecture GPUs support up to 2048 threads scheduled on an SMP at the same time. Nevertheless, a block supports up to 1024 threads only. Actually, there can be multiple blocks scheduled on an SMP at once. The block to SMP schedule is performed solely by CUDA framework and cannot be explicitly controlled using CUDA API [10].

Scheduling as many threads as possible can potentially hide or at least lower memory access latencies. Nonetheless, a kernel might need only a single block of threads to perform all the work. When launched synchronously, such a kernel

essentially blocks the device and all other subsequent kernel launches have to wait for this kernel to finish.

However, in some scenarios various kernel launches can be functionally independent, therefore do not require sequential behavior and, in contrast, could profit from parallel execution by utilizing device's SMPs more effectively. CUDA provides streams to launch and execute concurrently multiple kernels on a single device at once. The Fermi architecture supports up to 16 streams.

Default stream is used for kernel launches that do not specify any stream explicitly. In contract to explicit streams, the default stream does not just ensure on sequential kernel execution but blocks the host program until the kernel execution is finished.

3.3 OpenCL or CUDA

OpenCL and CUDA frameworks offer two different interfaces for programming GPUs. OpenCL is a more generic framework with capability to work with a broad range of different hardware platforms while CUDA is specifically tailored for NVIDIA GPUs. Naturally, it might be possible for the more specific CUDA interface to provide more efficient means to operate the hardware and therefore possibly offer better performance. Measurements [21] show that CUDA, in certain scenarios, provides better performance for similar kernel execution as well as for data transfers. However, the tool support, documentation and user friendliness are other factors for consideration. It is not in scope of this thesis to provide a comparison of these frameworks but it uses both of them and provides certain practical experience gained during their usage.

4. OpenCL implementation

This section describes the first Random Forests algorithm (see Section 2) parallel implementation built on top of OpenCL framework (see Section 3.1). In the beginning, it deals with a complexity analysis of the algorithm. It describes algorithm's input arguments, internal data structure and design details. It evaluates performance of the implementation compared with reference CPU serial and parallel implementations and discusses results.

4.1 Complexity

The basis for effective algorithm implementation is a complexity analysis. Its purpose is to identify time and memory constraints with respect to input arguments. The evaluation of algorithm can be performed using divide and conquer strategy. At first, step granularity is assessed. Then, every step is evaluated separately and its parameters are determined. In the end, all steps' constraints are combined together.

The following analysis is limited to a single decision tree training. Generalization for the entire algorithm is straightforward.

4.1.1 Bootstrap draw

The first step (see Algorithm 2.4.1(1.a)) of the decision tree training algorithm (see Algorithm 2.4.1) creates the bootstrap list of train samples drawn from the input data set. The size of the bootstrap is configured as a parameter K. Samples are selected using pseudo-random uniform distribution from all available samples. Let us assume the selection is $\mathrm{O}(1)$ operation then the complexity of the whole step is $\mathrm{O}(K)$.

4.1.2 Tree node training

The second step (see Algorithm 2.4.1(1.b)) is more complex.

Its first part (see Algorithm 2.4.1(1.b,i)) is linear with respect to the number of inspected attributes m.

Its second part (see Algorithm 2.4.1(1.b,ii)) depends on the size of the bootstrap the specific node is trained on. Let the node bootstrap size be equal to Z.

The best split point selection is performed using the information gain algorithm (see Algorithm 2.5.2).

It begins with the evaluation of the entropy (see Algorithm 2.5.1) of the whole node's bootstrap. Entropy calculation requires summed occurrences of labels of all samples. Thus, the calculation's complexity is linear to the bootstrap size and the number of distinct labels present in the whole input set. Therefore, the

complexity is bound by $\mathrm{O}\big(C + Z\big)$. The size of bootstrap tends to dominate number of labels.

Then, for all inspected attributes, every sample's value present in the bootstrap is considered a potential split point and its quality is measured. It requires the computation of entropy of all samples with the value equal or lower to the assessed value and entropy of all samples with the value higher than the assessed value. In case of categorical attributes, the entropy of samples with an equal value is calculated and separately entropy of samples with unequal value is calculated as well. Finally, information gain is given as the entropy of the whole bootstrap's entropy minus the sum of just described entropies weighted by respective sample counts.

The evaluation of a single potential split point is again bound by $\mathrm{O}\big(C + Z\big)$. For all inspected attributes and all bootstrap samples it is therefore bound by $\mathrm{O}\big(mZ(C + Z)\big)$. Thus, the bound for the tree's root node training is $\mathrm{O}\big(mZ(C + Z)\big)$.

4.1.3 Complete tree training

The tree node training complexity needs to be generalized for the complete tree training. When the algorithm trains a node, it splits its bootstrap into two, possibly uneven, bootstraps it needs to perform training on again. So, while node's training complexity is bound by $\mathrm{O}\big(mZ(C + Z)\big)$, its children complexity is bound by $\mathrm{O}\big(m(Z - Y)(C + (Z - Y)) + mY(C + Y)\big)$, where Y is the size of one of the children's bootstraps.

Generally, the algorithm that searches for the best split point, in the simplest case, evaluates the performance of all possible split points and the evaluation of a single split point is linear to the size of a bootstrap as well. Therefore, the most straightforward scenario yields quadratic complexity for each bootstrap division.

The maximum depth of the tree is in the worst case linear to the root's bootstrap size. In practice, the depth depends on the training data quality. Plenty of useful attributes can result in shallow trees and quick training. On the other hand, attributes lacking relevant information can result in very deep trees and slower training.

4.2 Input arguments

The algorithm's input is a set of training samples (see Figure 4.1), a number of trees to be trained, a bootstrap size and a minimal bootstrap size, where the training stops. It is an algorithm of supervised machine learning (see Section 2.1) and therefore a sample represents a set of attribute values and a label the sample belongs to. The set of attributes is the same for all samples in the data set. Numerical and categorical attributes are supported.

Sample	Attribute 1	Attribute 2	Attribute 3	Attribute 3	Attribute 4	...
#1	-595330.06	672084.2	3	-111428.57	686722.75	...
#2	-539251.06	94618.77	0	-509989.47	102257.64	...
#3	-634118.44	101074.62	1	-321004.7	473251.28	...
#4	-408535.5	628082.8	2	-526286.6	544952.5	...
#5	-516505.97	342787.0	1	-797219.94	480743.34	...
#6	-465080.62	96763.48	1	-702295.75	279796.88	...
#7	-304500.94	773571.06	0	-548080.0	181174.23	...
#8	-194303.6	647671.25	3	-212528.5	280061.62	...
...	...	...	...	...	...	...

Figure 4.1: Sample input data

4.3 Implementation overview

The idea of the implementation is to exploit the fact that trees can be, by the algorithm's nature, trained completely independent of each other. Also, the implementation minimizes host to device interactions to avoid frequent latencies of host to device data transfers.

Specifically, every tree is trained by a separate work group (see Section 3.1.2) and all trees are trained in a single kernel launch. The single kernel launch is used to minimize mentioned data transfer latencies and it also provides OpenCL runtime with flexibility for efficient distribution of work groups across devices.

The implementation parallelizes the algorithm not only on the tree level but it utilizes work group threads effectively for a single tree training.

This leaves mostly only initialization and finalization functions for the host code while all the rest is a sole responsibility of the device kernel.

4.4 Data structures

There is a global array of all input samples and their attribute values in the global memory.

Every tree is trained on a distinct bootstrap. Therefore, every work group has an array of indexes of its bootstrap's samples. It is allocated in the local memory.

To transfer trained trees to the host, there is a global result buffer used by all work groups where all trained trees are stored.

Since the whole tree training is performed entirely on the device and the extent of training is not known a priori the execution, the number of nodes trained depends on the data quality and pseudo-random selections, work groups must manage a list of nodes to be trained. In principle, this could be addressed using recursion. Nonetheless, OpenCL does not currently support recursion on NVIDIA devices. Therefore, every work group keeps a stack of nodes to train. The stack is an array

of entries, where each entry contains a start and an end index into the bootstrap and a pointer to the result buffer to store a trained node in. The stack is allocated in the local memory.

There are two types of tree nodes in the decision tree. There is a split node and a final node (see Section 2.2). The final node contains label information. The split node contains attribute identification, attribute's split value and two children node references.

4.5 Split point evaluation

The performance critical part of this implementation is the selection of the best split value. All threads in the work group are utilized for this operation. The information gain algorithm requires label occurrences of samples left to and right to the investigated split point summed up separately. The bootstrap is evenly divided across available threads in the work group and each thread sums label occurrences into its private memory (see Section 4.1.2). It is more effective for threads to save partial results in private memory and reduce final numbers into a shared array in the end than concurrently update a shared array throughout the iteration. For instance, if there were only few distinct labels present in the data, then multiple threads would most probably update the sum of the same label occurrence at the same time. Therefore, at the same time multiple threads would access the same part of memory. In order to maintain consistency threads, would use atomic operation. However, too many atomic update operations in the same part of memory degrade performance.

4.6 Implementation

Generally, the host allocates and initializes device buffers for input data and kernel computation. Then, it launches a kernel that trains all trees in parallel. For a single tree, the kernel generates the bootstrap, initializes stack and trains nodes until there are no more bootstrap parts to split. When all nodes are trained, the host reads all created nodes and constructs trees.

Algorithm 4.6.1 (OpenCL device kernel)

1. *Generate bootstrap*
2. *Initialize stack*
3. *While stack is not empty*
   (a) *Retrieve top entry from stack*
   (b) *If bootstrap's size is below limit or all samples share common label, add final node into the result buffer, continue*
   (c) *Select random attribute*
   (d) *Find the best split point*
   (e) *Arrange bootstrap into two parts, separated using best split point' value*

(f) Add split node into the result buffer

(g) Add two entries onto stack, on for each bootstrap part

4. Retrieve device result buffer and construct trees

Algorithm 4.6.2 (OpenCL host program)

1. Allocate and initialize device buffers

2. Launch tree train kernel 4.6.1, one work group per tree

3. Retrieve device result buffer and construct trees

4.7 Bootstrap size

The bootstrap size each tree is trained on is limited by the size of device's local memory the kernel will run on. The available local memory size on Tesla M2090 is by default 49KB, hence the maximum bootstrap size is roughly 12000 samples. Some parts of the local memory space is allocated for other purposes. The current implementation could be enhanced to support bigger bootstraps. Possible solution is to let the bootstrap reside in the global memory while using the local memory for temporary caching only or avoid the local memory at all.

4.8 Evaluation

4.8.1 Synthetic data generator

In order to properly test efficiency, correctness and performance of a machine learning algorithm, there is a need for good testing data sets. It is important to know the characteristics of the data for proper result interpretation.

It is challenging to find data sets meeting these conditions and hence a simple data generator for machine learning algorithms has been created. The main requirements laid follow

- Number of samples.
- Number of attributes.
- Number of output labels.
- Standard deviation of output label count distribution.
- Mean of an attribute quality.
- Standard deviation of attributes quality distribution.

The correlation between the output label and an attribute value is quite simple. Based on the quality determined for an attribute, the output labels are shuffled as much as possible for values starting from 0% and almost completely separated labels for quality closing to 100%. Then, attribute values are generated in an increasing or decreasing manner to span across all the samples. Therefore, good (see Figure 4.2) or bad quality (see Figure 4.3) data can be generated on demand.

Sample	Attribute	Class
#1	10.0	0
#2	15.0	0
#3	20.0	0
#4	25.0	2
#5	30.0	2
#6	35.0	1
#7	40.0	1
#8	45.0	1

Figure 4.2: Good quality attributes.

Sample	Attribute	Class
#1	10.0	0
#2	15.0	1
#3	20.0	2
#4	25.0	0
#5	30.0	1
#6	35.0	2
#7	40.0	0
#8	45.0	1

Figure 4.3: Bad quality attributes.

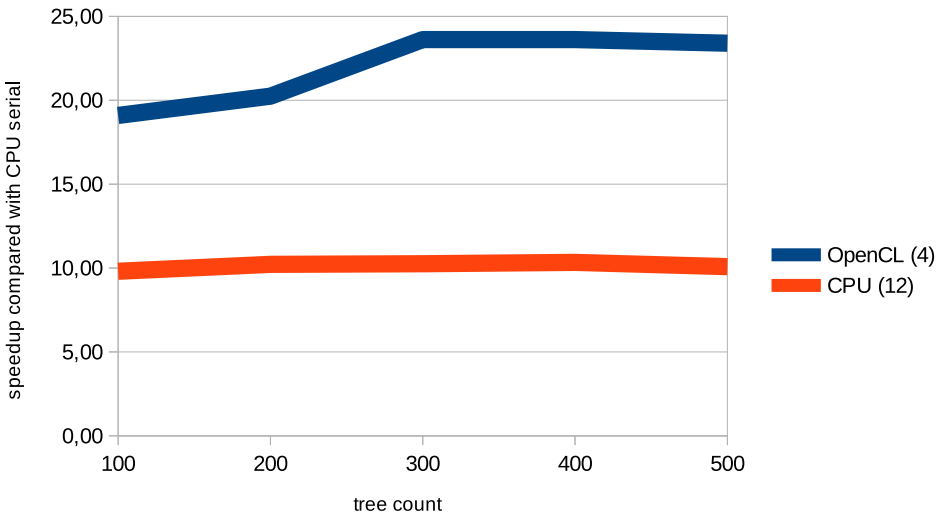


Figure 4.4: OpenCL and CPU parallel speedup comparison

4.8.2 Comparison

In order to estimate the quality of the implementation, there are reference serial and parallel CPU implementations created. The reference serial CPU implementation (see Algorithm 4.8.1) is in principle very similar to the OpenCL implementation (see Algorithm 4.6.2). The only difference is that there is no notion of the host and device program, everything is executed solely on CPU. The reference parallel CPU implementation is build on top of the serial CPU implementation. It uses OpenMP [9] framework to train trees in parallel.

Measurements are performed on hardware available at the Department of Software Engineering of the Faculty of Mathematics and Physics. The machine used for this thesis contains 12 Intel Xeon E5645 processor and 4 NVIDIA Tesla M2090 GPUs.

Algorithm 4.8.1 (CPU serial)

1. *For all trees*
   - (a) *Generate bootstrap*
   - (b) *If bootstrap's is size is below limit or all samples share common label, create final node, return*
   - (c) *Select random attribute*
   - (d) *Find the best split point*
   - (e) *Arrange bootstrap into two parts, separated using best split point' value*
   - (f) *Create split node*
   - (g) *For both bootstrap parts recursively call 1.(b)*

The OpenCL implementation performs quite well compared with reference CPU serial and parallel implementations and provides noticeable performance speedup (see Figure 4.4). The OpenCL implementation is more than 20 times faster

than the reference serial CPU implementation and the reference parallel CPU implementation is 10 times faster than the serial version.

4.9 Summary

This implementation finishes training of hundreds of trees with thousands of samples in bootstraps in a matter of minutes. Therefore, the application of this implementation for bigger sets would last for hours or days at worst. Hence, this implementation cannot be considered ready for processing of big data sets.

Since almost all the functionality and logic is built entirely in a tightly coupled device kernel, it makes identification of possibly slow parts of the program on critical paths complicated. Moreover, there seems to be an issue with performance profiling with certain OpenCL driver versions on some NVIDIA GPUs. There is even an open online petition addressing this issue [22].

This implementation omits one possible optimization. The selection of the best split point among bootstrap samples can be performed more effectively. Bootstrap samples can be sorted before the best split point search, therefore label occurrences of the whole bootstrap can be summed just once and for each possible split point evaluation they need to be updated only once, in constant time.

To summarize, this implementation provides an insight into the algorithm's performance but still leaves certain questions unanswered and possibilities for improvement open.

5. Fine-grained CUDA implementation

This section describes an improved Random Forests implementation. First, it deals with general aspects of this implementation, differences compared to the original OpenCL implementation (see Section 4), and motivation behind decisions made. Then, it describes the implementation in more detail. In the end, it evaluates the implementation.

This implementation is more fine-grained than the original OpenCL implementation. It addresses issues experienced with bottleneck identification in the original implementation built monolithically in a single device kernel. A less coupled implementation should be simpler to test and the performance evaluation of individual parts should be possible.

In contrast to the original OpenCL implementation, this one is build on top of the CUDA framework. Since the hardware available for testing are NVIDIA GPUs, it makes sense to explore possibilities of the CUDA framework as well. Also, as mentioned in the OpenCL implementation, there has been experienced issues with OpenCL profiling on NVIDIA GPUs on recent versions of CUDA drivers. Therefore, an implementation on top of the framework tailored specifically for the NVIDIA hardware might give some more insight into the issue and generally tools available for the CUDA framework might be easier to use.

This implementation also uses the possible optimization of the best split point search and therefore it is expected to perform significantly better.

Technically, this implementation creates trees and manages the execution mainly on the host but offloads all computation demanding parts in separately launched kernels.

5.1 Input arguments

Input (see Figure 5.1) is almost identical to the OpenCL implementation. The only difference is that this implementation supports only numerical attributes. This greatly simplifies the implementation as handling of distinct attribute types

```
newrf::rf_c*
 train(
 const int seed,
 const std::vector< std::vector< float > >& input,
 const std::vector< int >& output,
 const int tree_count,
 const int tree_bootstrap_size,
 const int tree_stop_size);
```

Figure 5.1: Fine-grained CUDA implementation parameters

```
// input attributes
// [device][...]
std::vector< float* > input_d;
```

Figure 5.2: Input data set

```
typedef struct sample_t {
  // id of the sample in input attribute array
  int id;
  // the clazz this sample belongs to
  int clazz;
} sample_t;

// [device][tree][...]
// sample_t holds bootstrap sample ids and labels
// size = bootstrap size (per tree)
std::vector< std::vector< sample_t* > > sample_s_d;
```

Figure 5.3: Bootstraps for all trees

complicates device kernel code considerably. In fact, this constraint does not limit the algorithm's functionality. Categorical attributes can be handled by input data preprocessing. It removes categorical attributes but adds artificial numerical attributes that mimic the expected semantics.

5.2 Data structures

Input samples and their attribute values are stored in arrays in the global memory (see Figure 5.2).

Bootstraps are allocated separately for every tree. Bootstraps do not contain samples with their attributes but only indexes into the global input array and labels (see Figure 5.3).

Best split point computation requires arrays to hold values of a single attribute in size of the bootstrap allocated for every tree in the global memory. The investigated attribute's values of the bootstrap's samples are copied from the global input array before the best split point search (see Figure 5.4).

Kernel computation results are stored in a structure allocated in the global memory. Namely, it contains details of the best split point found and the prevailing label found in the bootstrap (see Figure 5.5).

```
// holds values of a single attribute for best split point search
// [device][tree][...]
// size = bootstrap size (per tree)
std::vector< std::vector< float* > > value_s_d;
```

Figure 5.4: Temporary attribute array

```
// this struct is used to gather computed values from multiple
    kernels
// therefore the updates are not always in sync and valid even
    though some are
// in question and not present in this struct is the remaining
    range of samples evaluated
typedef struct split_t {
// position of samples array that provides the best possible
    among remaining samples
int position;
// value of the sample's attribute
// splits the attribute according to
//       if (x <= value)
//           -> left
//       else
//           -> right
// attribute not specified here
float value;
// the gain of the best split
float gain;

// the most prevalent label among all remaining samples
prevalent_t all;
// the most prevalent label among all remaining samples left to
    the best split (inclusive)
prevalent_t left;
// the most prevalent label among all remaining samples left to
    the best split (exclusive)
prevalent_t right;
} split_t;

// device to host communication, contains device kernel results
// [device][stream][...]
// size = 1 (per stream)
std::vector< std::vector< split_t* > > split_d;
```

Figure 5.5: Split result structure

```
struct node_t {
  int attribute;
  float value;
  node_t *left;
  node_t *right;
};

struct interval_t {
  int begin;
  int end;
  int retry;
  int tree_id;
  newrf::node_t* node;
  bool valid;
};
```

Figure 5.6: Interval structure

The host maintains a list of trees that must be trained yet. More precisely, it stores actual pieces of trees to be trained. There is a list of intervals (see Figure 5.6) into the bootstrap that must be processed and a node must be trained for.

5.3 Implementation

Essentially, the host initializes a list of intervals for all devices. These lists contain complete bootstraps for tree roots training. Then, the program takes an interval after another interval, finds the best split point, creates a node and adds new intervals bounded by the original interval range and the split point. If required, it removes the original interval and repeats this process until there is no interval left on any device.

Algorithm 5.3.1 (Fine-grained CUDA)

1. *Allocate all device structures (see Section 5.2)*
2. *Initialize device structures with input samples and generate bootstraps for all trees*
3. *Initialize interval list for all trees*
4. *While there is an interval any device's list to process:*
   - (a) *For all devices, for all streams do:*
     - i. *Retrieve interval on top of the device's interval list*
     - ii. *Asynchronously launch kernels:*
       - A. *Prepare attribute values of bootstrap samples into the temporary value array*
       - B. *Sort the bootstrap samples according to the temporary value array*
       - C. *Find the best split point among bootstrap samples*
       - D. *Detect the most prevailing label among bootstrap samples*
   - (b) *For all devices, for all streams do:*
     - i. *Wait for all streams to finish*
     - ii. *Copy device result buffers*
     - iii. *Create nodes and conditionally create new intervals*

5.4 Label count frequency

The label count frequency kernel iterates over bootstrap samples in the interval from its beginning till its end and sums occurrences of each label.

There is a global buffer allocated for label counts. Bootstrap samples are divided across all threads and each thread summarizes label counts in its part. In the first

```
__host__ __device__
void sort_by_key(
 const thrust::detail::execution_policy_base<DerivedPolicy> &exec
     ,
 RandomAccessIterator1 keys_first,
 RandomAccessIterator1 keys_last,
 RandomAccessIterator2 values_first);
```

Figure 5.7: Thrust sort by key signature

iteration the first thread inspects the first element, the second thread inspects the second element and so on. In the second iteration the first thread starts just after the last thread of the first iteration.

Finally, when all threads process its part of the bootstrap, the most prevailing label is identified and together with a flag whether there are other labels present or not, the results are written to the split result structure.

5.5 Data load

The kernel that loads data iterates over bootstrap samples in the interval from its beginning till its end and every thread copies a single attribute value of a particular sample into the temporary buffer of values.

5.6 Sort

Sorting is a common algorithm used internally in many other algorithms. There are some available CUDA libraries implementing sorting algorithm, more specifically a sort by key algorithm.

Two CUDA libraries fulfilling the requirement have been identified. CUDA Thrust and CUB library have been examined and their performance have been evaluated. Apart from these, three versions of the bitonic sort algorithm have been implemented and compared.

5.6.1 CUDA Thrust

CUDA Thrust is an official NVIDIA C++ template utility library based on Standard Template Library (STL) [14]. It is a header only library with detailed documentation and comprehensive examples. It was the first library of choice to use for sorting.

Thrust provides standalone host functions for the radix sort algorithm. Mainly, it offers means for the sort by key algorithm, where two independent arrays are sorted. Only one array is used as a source for keys to compare and both arrays are sorted (see Figure 5.7). Also, it seems to fit the requirement of asynchronous per stream execution utilized by the implementation (see Algorithm 5.3.1).

However, measurements revealed a significant drawback. There seems to be a compelling inefficiency in small data set processing, it is more than 100 times slower than the STL CPU sort. In contrast, large data sets are processed very effectively and the speedup is more than 30 times higher compared to the STL CPU sort. However, such large data sets are infrequently processed in tree training. Moreover, as the training progresses the data sets are getting smaller but are growing in numbers. Therefore, the CUDA Thrust's radix sort could be utilized in the beginning of the training where bootstraps are still considerably large.

Even though the sort by key function accepts a stream to execute on as an argument, it does not work asynchronously as expected. Generally, when a CUDA kernel is launched and a stream is specified, the launching call does not block. Actually, a kernel launch can block but only due to a technical limitation in form of a limited kernel launch queue. The CUDA launch queue can hold up to 1024 prepared kernels [10]. Therefore, first 1024 CUDA kernel launches do not block while any subsequent kernel launches do block, unless enough queued kernels have finished already. Measurements showed that calls to the sort by key function make up almost all of the test run time while synchronization on streams in the end is practically instant. In contrast, other measurements of asynchronous kernels over streams usually spend just a fraction of time in the kernel launch phase but wait considerable amount of time in the final stream synchronization phase.

5.6.2 CUB

CUB from NVlabs is a library of parallel primitives and utilities. It is designed as a collection of rather small and isolated functions usable as building blocks for the maintainable CUDA code without a need to reimplement common functionality over and over again [15].

The idea was to use device implementation of the radix sort in the CUB library and tailor a sorting kernel to meet the needs. Unfortunately, even though officially supported, sorting of floating point data does not behave as expected. Radix sort of floating point (IEEE 754) values require a lexicographical transformation to be performed before sorting to yield correct results. But this transformation is not performed and therefore data is not correctly sorted. The issue has been reported including a test evidence and a source code to reproduce the issue but there has been no reply from CUB authors yet.

5.6.3 Bitonic sort

The bitonic mergesort is a parallel algorithm for sorting. It has the worst case performance of $\mathrm{O}\left(\log n^2\right)$ of parallel time [16]. The algorithm runs in multiple phases, in each phase every value is accessed and compared to another value and swapped if needed. This implies that phases cannot be interleaved but a new phase can be started only when a previous phase has been completely finished. In principle, there are two ways to implement this in CUDA.

First, a kernel is implemented as a single phase only. This allows a phase to be launched with an arbitrary number of threads and blocks to execute on and therefore achieves great parallelism. All phases are asynchronously launched at once in a single stream which ensures correct synchronization between phases.

Second, a kernel is implemented to perform all phases in a single invocation. This requires a single invocation only but limits the number of threads involved to the technical limit of the GPU. The Fermi architecture limits the block size to 1024 threads [12]. So, while the first option can process larger arrays (more than 1024 elements) using as many threads as there are elements. This option can utilize 1024 at most. Nonetheless, there is less overheard since there are fewer CUDA API calls and it also allows sorting of data small enough to fit in shared memory to be possibly more effective.

The bitonic sort requires data of particular lengths to work correctly. Only arrays in the size of power of two are supported. There is a modified arbitrary bitonic sorter capable of sorting arrays of any length [17]. The original algorithm is used in this implementation and the functionality of sorting arbitrary width arrays is ensured by copying values into a temporary array of the correct size, filled with infinity values at the end since infinity is not expected to appear in data. Technically, infinity can be replaced by NAN. The allocation and data copy cause noticeable performance penalty only for small data sets and in fact it is just theoretical since small data sets are sorted in shared memory.

5.6.4 Sort algorithms comparison

There are four algorithms measured and compared to provide a comprehensive insight into the performance of sorting algorithms on GPUs - Single phase bitonic in global memory, single phase bitonic in shared memory, Thrust radix sort and GNU GCC parallel sort.

The concept implementation of bitonic arbitrary sort performed considerably worse than other bitonic sort implementations. It is unclear whether the performance degradation is caused by ineffective implementation or the base bitonic sort algorithm fits CUDA framework better. A detailed analysis is out of scope of this thesis and is described in more detail in the future work section.

The single phase bitonic sort algorithm for shared memory (see Section 3.2.5) is limited to 64KB on the Fermi architecture [12]. This algorithm can be applied to smaller arrays only. It is the best performing algorithm for arrays of size anywhere between 32 to 8192 elements included (see Figure 5.8). At its peak it outperforms the C++ STL sort algorithm more than 13 times.

The single phase bitonic sort algorithm for global memory is not limited to arrays that small. However, its performance in the same range is slightly worse (see Figure 5.8). It is the best performing algorithm for arrays of size between 8192 and almost 65536 elements. It does not reach the quality of the shared memory version but it still outperforms the C++ STL sort algorithm at last 5 times in the mentioned range (see Figure 5.9).

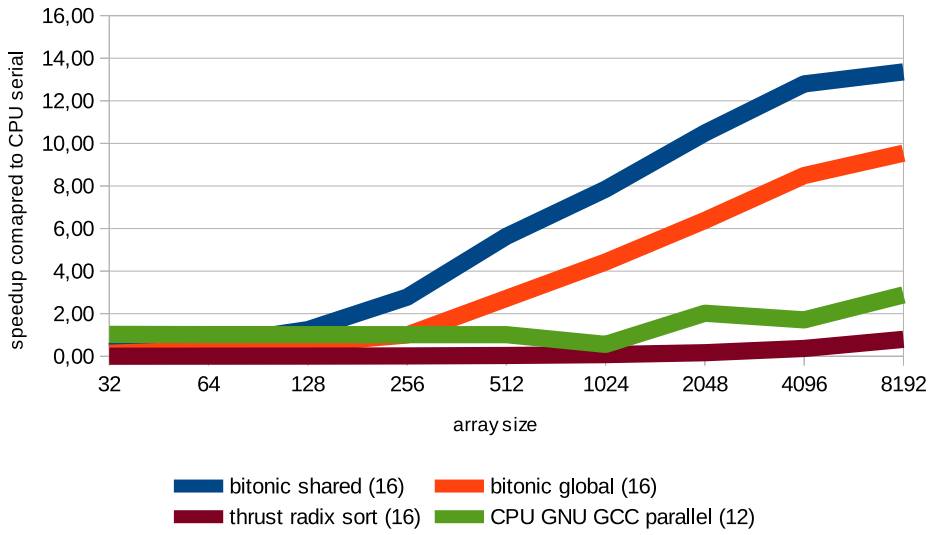


Figure 5.8: Sort algorithm comparison for small arrays

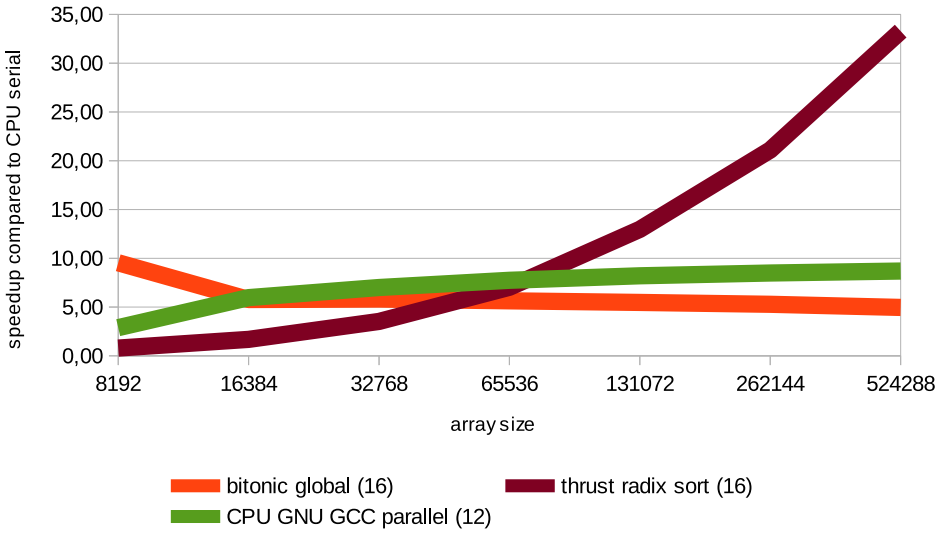


Figure 5.9: Sort algorithm comparison for big arrays

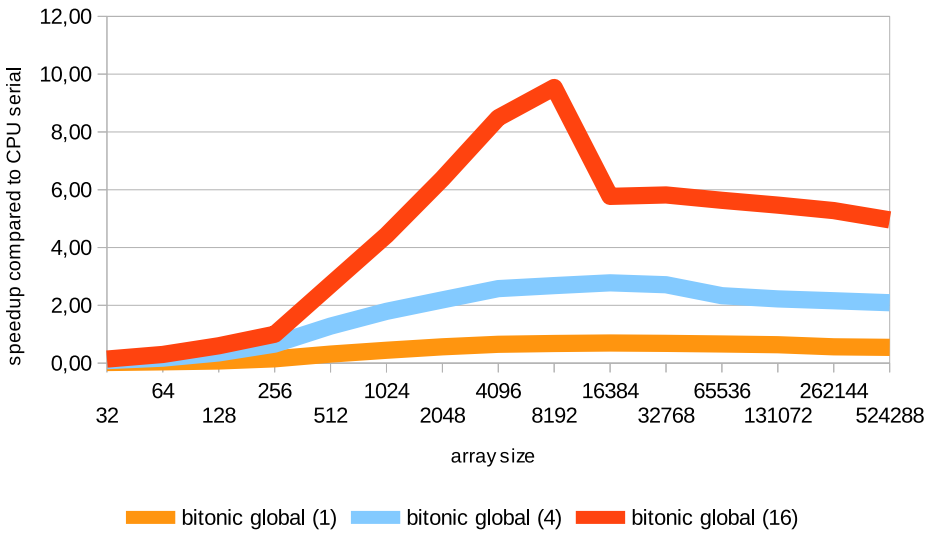


Figure 5.10: Bitonic algorithm comparison for various number of streams used

However, bitonic sort algorithms perform better than the C++ STL sort only when multiple streams are used and the more streams are used, the better (see Figure 5.10).

The Thrust radix sort algorithm outperforms all other algorithms for arrays bigger than 65536 elements. The maximum size of an array the performance has been measured for is 524288 elements and the algorithm outperforms the C++ STL sort algorithm for this array size more than 33 times (see Figure 5.9).

The bitonic sort algorithm for global memory, launched in 16 parallel streams, degrades noticeably in performance in the range between 8192 and 16384 elements (see Figure 5.10).

The fine-grained implementation uses the bitonic sort in shared memory for small arrays and in global memory for large arrays. The Thrust radix is not used because it outperforms others only for very large arrays and it does not outweight the limitation of blocked streams.

5.7 Best information gain

The best information gain kernel (see Algorithm 5.7.1) internally splits the interval into parts a particular thread will process. Every thread is assigned with an appropriate continuous part of the interval that is to be evaluated. The samples in the interval must be sorted according to the investigated attribute. There are two private arrays allocated in size of the total distinct labels present in the whole input data set. One array for all samples left to the start of thread's assigned part and one for the rest of samples.

For example, if there are 16 samples, 4 distinct input labels and 4 threads, then the first thread evaluates the first 4 samples, the second thread next 4 samples and so on (see Figure 5.11). Every thread needs separate variables to store

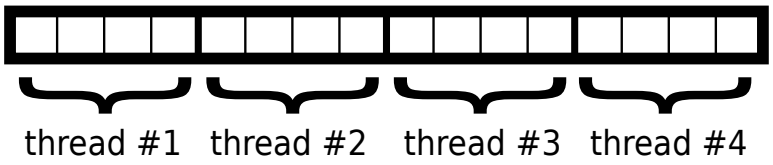


Figure 5.11: Sample distribution across 4 threads

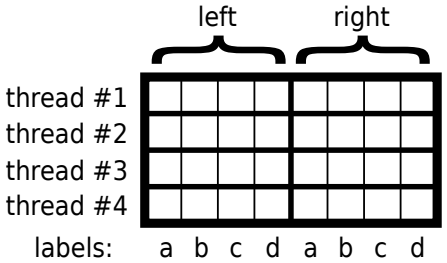


Figure 5.12: Thread label occurrence arrays for 4 threads and 4 labels

information about every label on both sides of the point it evaluates (see Figure 5.12).

Algorithm 5.7.1 (Best information gain)

1. *Compute entropy (see Algorithm 2.5.1) of the whole bootstrap*
2. *ThreadRange = BootstrapSetSize / ThreadCount*
3. *ThreadPosition = ThreadRange * ThreadId*
4. *Initialize Left[ClassCount] and Right[ClassCount] arrays*
5. *For i = 0 to BootstrapSetSize*
   (a) *If i < ThreadPosition*
      i. *Left[BoostrapSet[i].Class] += 1*
   (b) *Else*
      i. *Right[BoostrapSet[i].Class] += 1*
6. *For i = 0 to ThreadRange*
   (a) *Left[BoostrapSet[ThreadPosition+i].Class] += 1*
   (b) *Right[BoostrapSet[ThreadPosition+i].Class] -= 1*
   (c) *Compute entropy of Left*
   (d) *Compute entropy of Right*
   (e) *Compute information gain (see Algorithm 2.5.2) using Left and Right entropy*
   (f) *If this information gain exceeds this thread's previous best information gain, store it*
7. *Reduce all thread's stored best information gains and select the best of them*

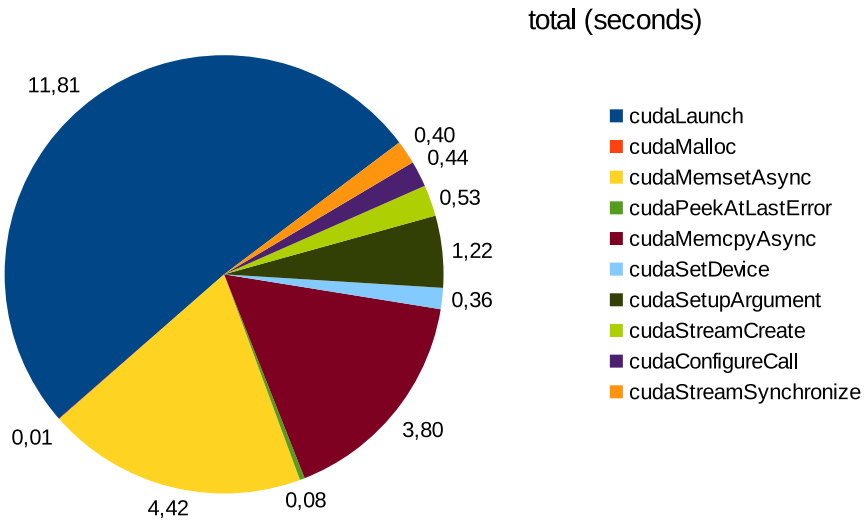


Figure 5.13: CUDA API calls total execution time

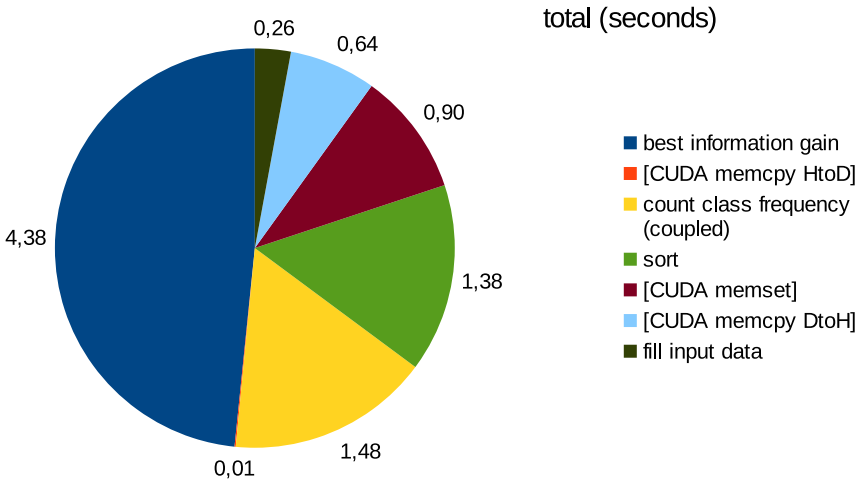


Figure 5.14: Kernels total execution time

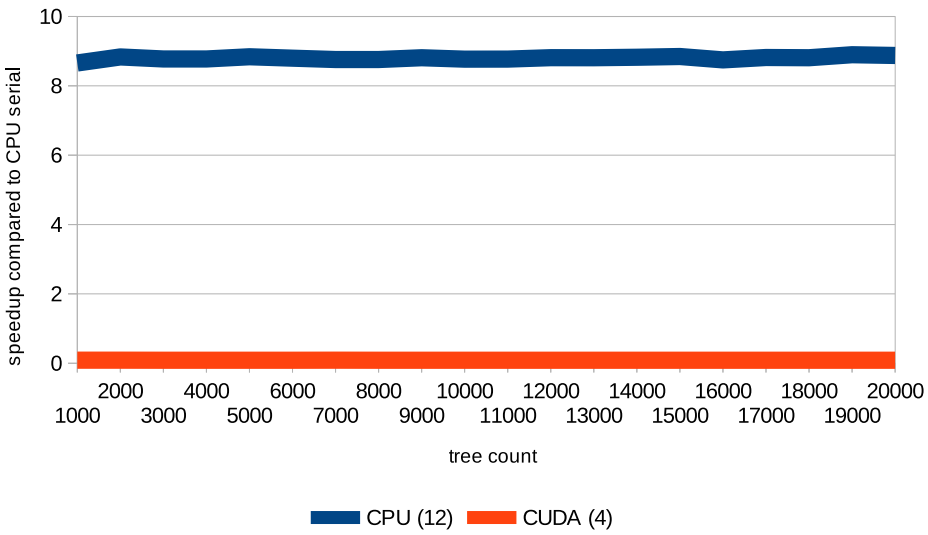


Figure 5.15: CUDA and CPU comparison with various tree count

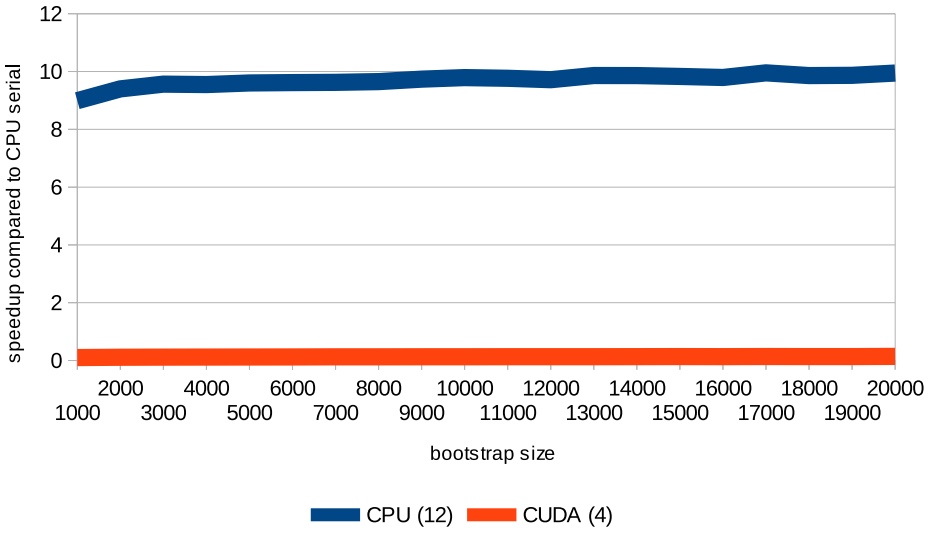


Figure 5.16: CUDA and CPU comparison with various bootstrap size

5.8 Summary

The program execution has been profiled using the Visual Profiler [10]. The program execution profile pinpoints certain performance weak points of this implementation. There are 1421892 CUDA launch calls performed in the sample instance. The program spent almost 12 seconds (see Figure 5.13) performing kernel launch calls while the whole program execution lasted for 26 seconds totally. In total, CUDA API calls took more time than all kernel executions lasted for (see Figure 5.14).

The reference CPU serial and parallel implementation has been updated to use the applied optimization to more precisely render the performance differences.

This implementation, in contrast to the OpenCL implementation, does not outperform the CPU parallel implementation nor does it even perform better than

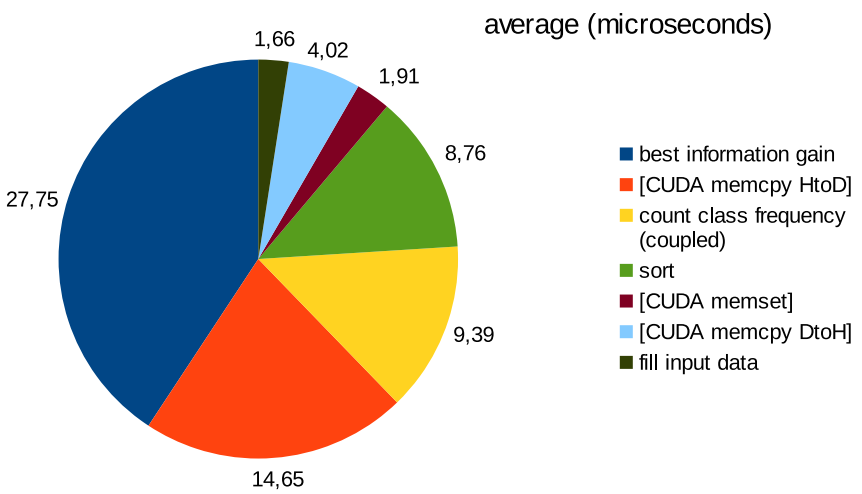


Figure 5.17: Kernels average execution time

the serial CPU implementation. There is no sign of an optimistic performance outlook neither for various values of the tree count parameters (see Figure 5.15) nor the bootstrap size parameter (see Figure 5.16).

There seems to be a fundamental complication of this fine-grained implementation. There are too many kernel launches performed for the implementation to be efficient at all. The root cause is the number of intervals the implementation has to process in total. In the beginning, there are exactly as many intervals as there are trees but the bigger the interval, the more probably its processing will queue two more, smaller intervals for processing. And the smaller the processed intervals are, the higher the ratio of maintenance work instead of kernel work is performed.

The label count frequency kernel that sums label occurrences over the bootstrap takes considerable amount of time, taking into account it consists of only a single linear iteration over the bootstrap. In total (see Figure 5.14) and on average (see Figure 5.17), it computes for as long as the sort kernel. Supposedly, it might be caused by atomic operations performed over only few variables by many threads. This can be probably fixed by replacing atomic operations over global variables with private variables for the iteration, and reduction into the global memory at the end.

This implementation provides performance information in more detail but it requires a serious refactoring to provide competitive efficiency compared to CPU implementations.

6. Monolithic CUDA implementation

This section describes an implementation based on the fine-grained CUDA implementation (see Section 5). Many details in this implementation are identical to the fine-grained implementation and therefore this section focuses mainly on describing differences made. In the end, it evaluates effects of these changes.

Main motivation for this monolithic implementation is to diminish the performance degradation caused by frequently executed CUDA API calls. In fact, this implementation is generally a combination of the OpenCL implementation philosophy and the fine-grained implementation internals. This implementation performs the whole tree training solely in device kernels but it controls the training process scheduling more precisely.

6.1 Input arguments

Input arguments are completely identical to the find-grained implementation's arguments (see Section 5.1).

6.2 Data structures

There are slightly different data structures compared to the fine-grained implementation. Since this implementation relies more heavily on device kernels even for tree building there are more structures needed. This increases memory requirements for a single tree training. Therefore, trees are trained in batches.

Input samples and their attribute values are stored in the global memory.

Bootstraps and temporary arrays for values of an attribute are located in the global memory. However, this time, it is not allocated for every tree but it is allocated in size of the batch.

Similarly to the fine-grained implementation there is a list of intervals to process. However, this time it is not located in the host memory but in the global device memory. It serves two purposes. It is again a list of intervals that must be processed and a list of nodes that have already been created in the tree. After a training is finished, the list is copied to the host memory and trees are constructed. It is allocated in size of the batch.

There is an array in the global memory with prepared random attribute selection. It is allocated in size of the batch.

The selection of attributes for tree's node split points is an important part of the Random Forests algorithm. Attributes must be selected uniformly from the whole attribute set. In the OpenCL implementation (see Section 4), there has

been an issue experienced during development. Attributes have been selected using a slightly bugged Lehmer random number generator function. The issue was not obvious to detect and it exhibited itself by only slim but noticeable classification precision degradation compared to results obtained with a proper uniform random data source.

To avoid further complications the C++ STL Mersenne Twister 19937 generator [20] is used as a random data source and the C++ STL uniform integer distribution to span intervals as necessary. Since the whole tree training happens in device code, the selection of attributes to split on is precomputed on the host and stored in the device buffer for device kernels to use.

CUDA framework provides similar API for uniform data generation. However, until version 7.5 [19] released in September 2015 there have been no proper means to generate random numbers on devices but only on the host which would require buffers to hold generated numbers anyway.

6.3 Implementation

Generally, the host allocates and initializes device structures to fit trees in size of the batch. Then, the host launches asynchronous kernels across available streams to train trees in the batch and waits for all kernels to finish. The host gathers results of tree training and continues with kernel launches until there are no more trees to be trained.

Algorithm 6.3.1 (Monolithic CUDA)

1. *Chose BatchSize*
2. *Allocate all device data structures (see Section 6.2)*
3. *Initialize input data device buffers*
4. *Let TrainedTrees = 0*
5. *While TrainedTrees < TreeCount do:*
   (a) *TrainedTrees += BatchSize*
   (b) *Initialize all batch device structures*
   (c) *For all devices, for all trees in the batch do:*
      i. *Asynchronously launch tree train kernel evenly across available streams*
   (d) *Wait for all streams to finish*
   (e) *Copy device result buffers to the host*
   (f) *Create trees*

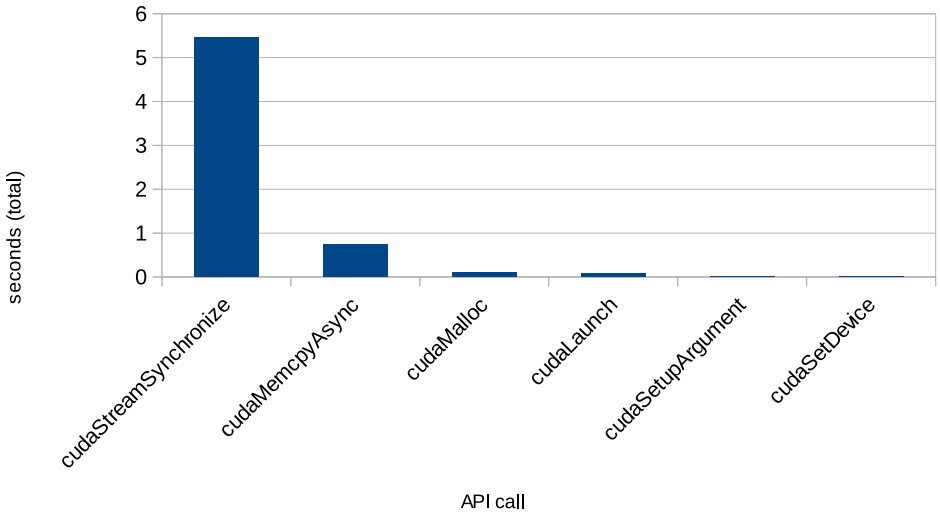


Figure 6.1: CUDA API calls total execution time

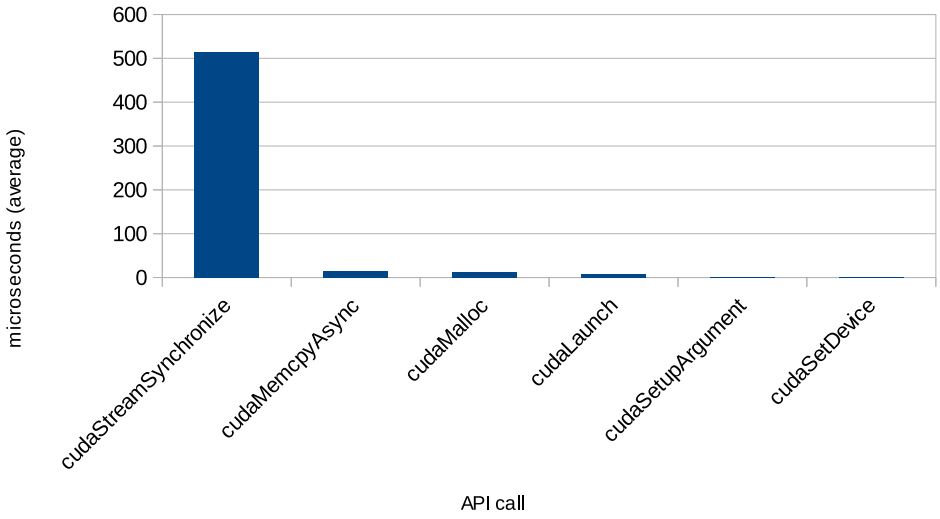


Figure 6.2: CUDA API calls average execution time

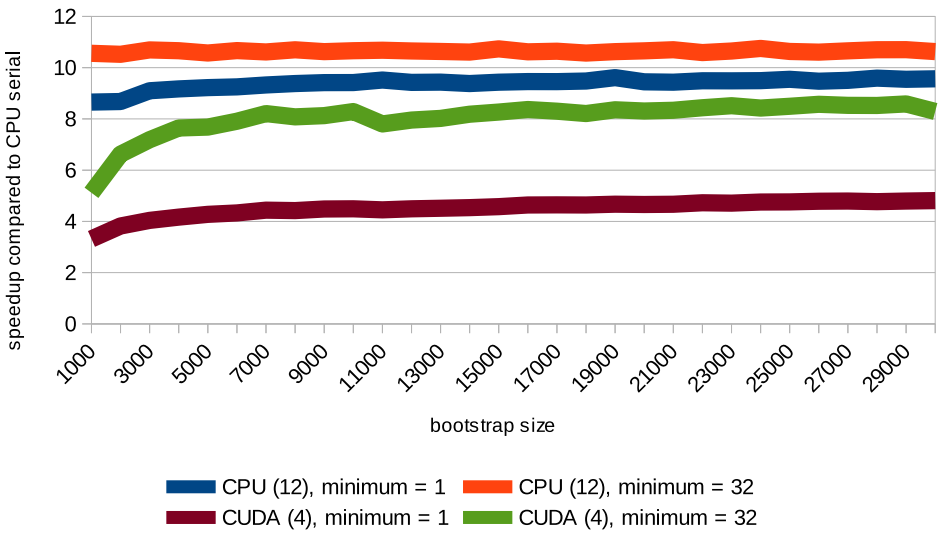


Figure 6.3: Performance comparison with various bootstrap size and train stop size

6.4 Summary

This implementation resolves the problem of considerable performance degradation caused by frequent CUDA API calls (see Section 5.8). The only outstanding function is the stream synchronization (see Figure 6.1, 6.2). This is expected since this call essentially just waits for all launched kernels in a stream to finish.

In contrast to the fine-grained implementation, this implementation runs 8 times faster compared to a reference serial CPU version (see Figure 6.3). Nevertheless, there have been more ambitious performance expectations.

Generally, there might be certain fundamental aspects in the approach chosen for the implementation that practically resulted in technical details that are not in line with best practices for effective CUDA programming.

First, an input data set is in fact a matrix of attributes and samples where a row represents values of a single attribute for all samples and column values of every attribute for a single sample. Tree training requires a list of samples, the bootstrap. Node training needs the bootstrap and values of samples for a randomly selected attribute. Therefore, every node training accesses data scattered across the input data matrix in the global memory. There is, however, a considerable performance difference between certain memory access patterns. It is usually much more effective to operate with data located close together than to access data spread unpredictably across a vast range as in this case (see Section 3.2.6).

This is given by the Random Forests algorithm nature. There can be many attributes and many samples in an input data set and every bootstrap uses a random selection of these samples. Theoretically, every bootstrap could be kept together with copy of attributes. However, this solution requires considerably more memory and more memory transfers.

Second, the split point search algorithm expects sample labels and selected attribute's values prepared in two arrays and it needs both these arrays to be sorted

by these values. The entropy computation requires sums of all label occurrences calculated. Therefore, to compute the information gain of a certain point, the algorithm needs sums of label occurrences left to the investigated point and also sums of label occurrences right to the point. When label occurrences of some point are known it is a constant operation to determine summed label occurrences of its neighbour. Therefore, threads in a block are allocated with a separate continuous part of the bootstrap to process and they independently determine the partial best split point with the highest information gain located in individual intervals (see Section 5.7). Unfortunately, this again results in a situation where threads do not access closely located parts of the memory in a critical part of the implementation, but are scattered across the whole bootstrap.

Third, the implementation launches kernels to train whole trees. In the beginning, tree's nodes are trained using configured bootstrap size. However, the bootstrap for nodes deeper in the tree gets smaller and smaller as it continuously gets split. Since the kernel is launched once for a tree, the thread count for the whole tree training is constant. So, at first, the work for thread can be substantial but as training progresses, the bootstrap size gets smaller and eventually the bootstrap might be even smaller than the number of threads, which effectively wastes thread's instruction cycles. The minimal bootstrap size can be configured but it puts produced model's classification performance in question.

This issue could be addressed by using an adaptive solution, which would process bootstraps of different sizes in different ways. For example, small bootstraps could be handled in groups by single kernels or on CPU. This is described in more detail in the conclusion (see Section 9.1).

To summarize, the implementation is getting into acceptable performance bounds but does not outstand. It is practically usable though and possibly there can be found ways to improve the performance as well. Few ideas for future work and improvements are described in the conclusion.

7. Classification

Goal of this thesis was to explore and implement tools for classification of stellar spectra. The implementation is provided and this section evaluates a practical application on real world data.

The classification performance of the algorithm is tested on a preprocessed and labelled data set measured in the Ondrejov observatory. The data contains 1565 samples of manually labelled B[e] stars of 5 different types. The data set is a preprocessed set of raw wavelength intensity refined using Cohen-Daubechies-Feauveau 5/3 transformation [8].

Bigger data sets are available but their use is not simple and straightforward. The data must be pulled from remote databases, normalized and converted to formats suitable for the implementation. This work is beyond the scope of this thesis. We try to estimate the classification performance on bigger data sets using measurements performed on the limited data set.

7.1 Training set size

This measurement gives an insight into what difference in the classification performance is provided by ratio of training data set size to verification data set size.

Measurements have been performed using sets in size of 2% to 80% of all samples for training, the rest for verification tests. Even for a very small number of samples the resulting model performs surprisingly well on the verification data set. For 2% of samples, only 31 in total, used for training the success rate is above 81% and for 10% of samples, 156 in total, used for training the success rate is above 93%. The classification performance steadily grows for increasing training data set and success rate reaches 98% for training set in size of 80% (see Figure 7.1).

Apparently, even very small data sets can be used for training and provide feasible results for recognition of unseen samples. Nevertheless, the richer and bigger the training data set are, the better recognition results are provided by trained models.

7.2 Tree count and bootstrap size

The classification performance of a trained model is highly dependent on the number of trees and bootstrap size a classifier is trained on. One of the benefits of the Random Forests algorithm is the fact that it is resistant to overfitting [7]. It means it is not easily possible to train a model that performs well on training data but its performance for verification data sets degrades. This is an undesired property of some machine learning algorithms. Such algorithms must

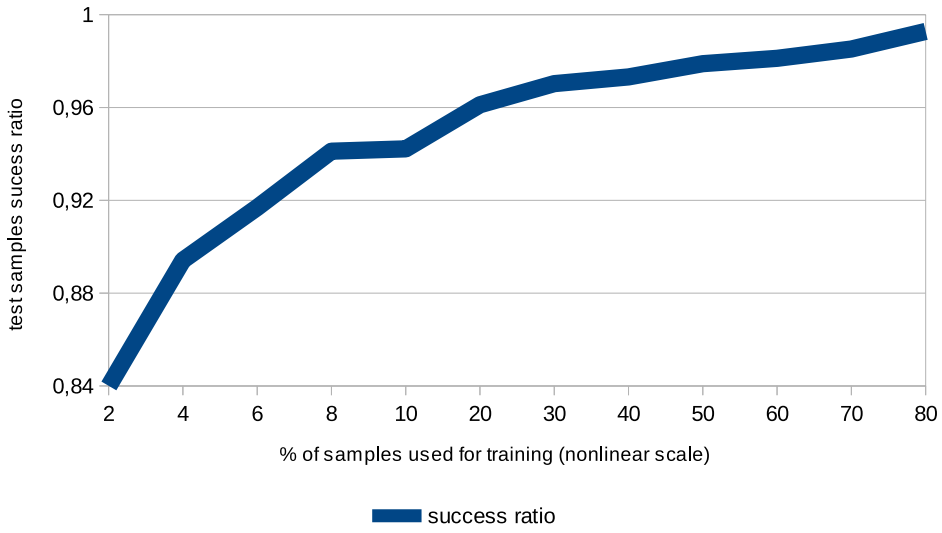


Figure 7.1: Classification performance for various training set size

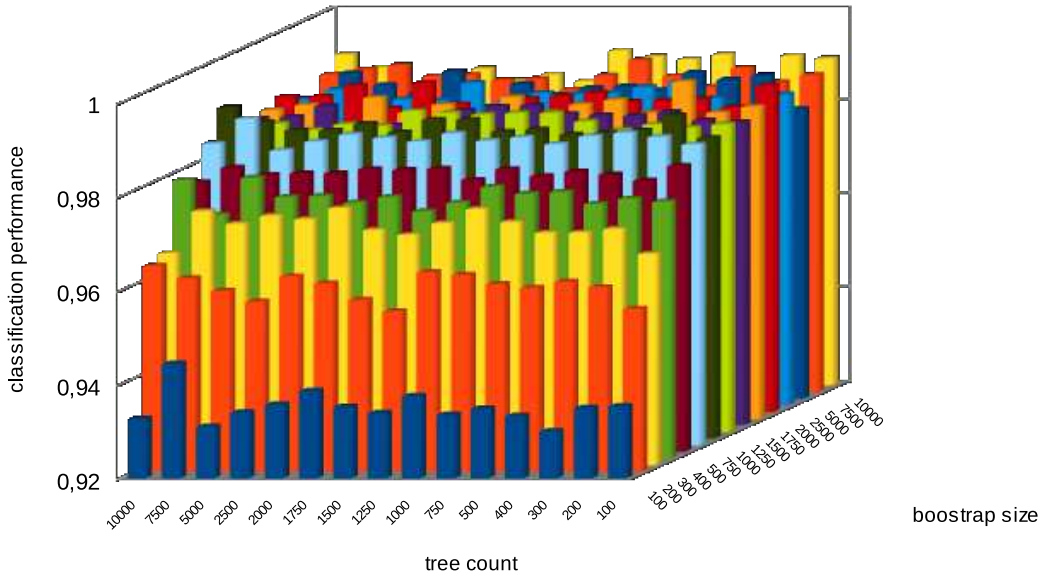


Figure 7.2: Classification performance for various tree count and bootstrap size

be carefully tuned and their parameters must be tweaked thoroughly to avoid creation of models that memorize presented test samples but cannot successfully recognize only slightly different samples.

Since Random Forests algorithm is resistant to overfitting, it is safe to increase input parameters while the classification performance is growing. It means that unlimited increase of input parameters will provide better performance though.

For instance, training on 80% of 1565 samples provides the best classification performance for bootstrap size of 750 samples and more. Further increase in bootstrap or tree count does not provide any relevant gain in classification performance. Interestingly, an increase in a tree count does not seem to improve recognition of created models (see Figure 7.2).

8. Related work

Several projects implement a classifier based on Random Forests algorithm and utilize GPUs for execution. They apply various methods to obtain an effective classification tool. The projects described and referred to below utilized CUDA framework for their implementations.

8.1 CudaTree

CudaTree is a comprehensive project [4]. It explores multiple possible implementation approaches. The tuned implementation available publicly, called CudaTree, combines multiple approaches to deliver the maximum performance. Initially, it uses the same method used in the fine-grained implementation (see Section 5). There is a kernel launched for every tree's node while the bootstrap is still large enough. Then, when the bootstrap gets smaller, the CudaTree processes the rest of the bootstrap in another kernel in a single launch, which is a contrast to the fine-grained implementation. Also, it uses a different method to determine the best bootstrap division, it computes label histograms, Gini impurity scores over feature tresholds and it evaluates what brings in the highest information gain [1]. The combination of different execution approaches for distinct bootstrap size is an interesting idea and probably one of the reasons this project provides an effective tool for classification.

8.2 CUDA is not meant for training Random Forests

Another related project from an unknown author provides experience gained from an implementation of Random Forests in a brief report [5]. The project approached the problem in a similar fashion the fine-grained implementation did. The main loop of the program on CPU keeps track of nodes to be trained and schedules GPU kernels to train these nodes and transfers the results back to host until there are no more nodes to be trained. The report does not make it entirely clear, how many kernels are launched at once and how streams are used to schedule particular kernels. This project didn't succeed in the creation of a GPU implementation more efficient than a serial reference CPU implementation. The report suggests the complexity of the kernel and overutilization of GPU memory as ideal candidates for performance improvement.

8.3 Analytical programming and Random Forests

A different project investigates the use of analytical programming, evolutionary algorithms together with Random Forests for classification of stellar spectra [6].

It utilizes analytical programming and evolutionary algorithms to enhance and reduce the feature set of input data to improve the classification performance. The article focuses mainly on the description of specifics of analytical programming used. The Random Forests implementation is not described in much detail. According to the report, it provides relevant results although there is still room for improvement in matter of execution performance.

8.4 Summary

Random Forests algorithm can be implemented even more efficiently for GPUs than this thesis achieved. Nevertheless, slightly different and possibly more complex methods would probably have to be used. Moreover, they must be cleverly combined together into a comprehensive solution to create an effective tool.

9. Conclusion

The goal of this diploma thesis was to implement a high throughput Random Forests implementation for GPUs devices and evaluate its performance on real world data sets. The algorithm was implemented in three different ways using OpenCL and CUDA frameworks.

The nature of the Random Forests algorithm allows for an obvious parallelism on the tree level allowing for an easy and efficient parallel CPU implementation. Essentially, just few lines of OpenMP declarations on top of the serial CPU version can effectively parallelize the algorithm for an arbitrary number of CPUs without any performance degradation since the algorithm requires no form of synchronization, dependency or reduction across any two distinct trees to be performed.

In contrast, GPU implementations are fairly more complex compared with CPU implementations while the performance gain is measurable even though not as high as initially expected. Technically, there are certain aspects (see Section 6.4) of these implementations caused by the chosen approach that can partly explain the reasons for the, possibly inadequate performance that cannot be easily addressed without a massive restructuralization and redesign of implementations that is out of scope of this thesis.

9.1 Future work

There are outlined some ideas for a possible future work that could be generally beneficial for GPU programming or could help with the provision of a better Random Forests implementation.

9.1.1 In-place bitonic sort for arbitrary sized arrays

One of the challenging aspects of this thesis was the search for an effective sorting algorithm for device kernels. There are implementations to use available. Usually, they are based on the radix sort algorithm that is a good choice for GPU programming in general as it offers interesting performance for considerably big data sets. However, the implementation tested was outperformed on smaller data sets by the bitonic sort algorithm (see Section 5.6.4). The bitonic sort algorithm is an in-place sorting algorithm but it generally supports only sequences of a certain size. There is a generalization of the algorithm for arbitrary big sequences [17]. A concept implementation of this, a more flexible version, was created and evaluated but it, most possibly due to the implementation's limitations, performed worse than the general version. Nonetheless, more research in this topic and a fine tuned arbitrary in-place bitonic sort implementation could be generally a very useful tool.

9.1.2 Algorithm restructuralization

Described implementations are based on the abstract view of the Random Forests algorithm and they internally work with tree and node structures. Therefore, they reach a state where the amount of work associated with training a node becomes very small. Another approach would be to focus solely on the repetitive division of the bootstrap. Hence, the size of the processed data by kernels would be constant. The question is, how the logical division of the data affects the performance and what are the requirements to implement this and what complications and constraints can possibly emerge.

9.1.3 Hybrid solution

It might be possible to derive a hybrid implementation by using the parallel CPU implementation for computation of small bootstraps and the GPU implementation for bigger ones. This could theoretically yield a better performance than present solutions.

Bibliography

[1] Hastie, Trevor. Tibshirani, Robert. Friedman, Jerome. *The Elements of Statistical Learning*. Second Edition. Springer: xxx. ISBN xxx : xxx

[2] Witten, Ian H. Frank, Eibe. *Data Mining - Practical Machine Learning Tools and Techniques*. Second Edition. Elsevier: 2005. ISBN 0-12-088407-0

[3] Gaster, Benedict R. Howes, Lee. Kaeli, David R. Mistry, Perhaad. Schaa, Dana. *Heterogeneous Computing with OpenCL*. Revised OpenCL 1.2 Edition. ISBN 978-0-12-405894-1

[4] Liao, Yisheng. Rubinsteyn, Alex. Power, Russell. Li, Jinyang. *Learning Random Forests on the GPU*.
`http://news.cs.nyu.edu/~jinyang/pub/biglearning13.pdf`
`https://github.com/EasonLiao/CudaTree`

[5] *CUDA is not meant for training random forests*.
`https://rstudio-pubs-static.s3.amazonaws.com/15192_5965f6c170994ebb972deaf18f1ddf34.html`

[6] Šaloun, Petr. Drábik, Peter. Zelinka, Ivan. Bucko, Jaroslav. *Big Data Spectra Analysis Using Analytical Programming and Random Decision Forests*

[7] Kleinberg, E. M. *An overtraining-resistant stochastic modeling method for pattern recognition*. Ann. Statist. 24 (1996), no. 6, 2319–2349. doi:10.1214/aos/1032181157.
`http://projecteuclid.org/euclid.aos/1032181157`

[8] Bromová, Pavla. Bařina, David. Škoda, Petr. Vážný, Jaroslav. Zendulka, Jaroslav. *Classification of Spectra of Emission-line Stars Using Feature Extraction Based on Wavelet Transform*

[9] *OpenMP*
`http://openmp.org/wp/`

[10] *CUDA*
`https://developer.nvidia.com/cuda-zone`

[11] *Tesla M-Class GPU Computing Module*.
`http://www.nvidia.com/docs/IO/105880/DS_Tesla-M2090_LR.pdf`

[12] *Whitepaper, NVIDIA's Next Generation, CUDA Compute Architecture: Fermi*.
`http://www.nvidia.com/object/IO_89570.html`

[13] *Whitepaper, NVIDIA's Next Generation, CUDA Compute Architecture: Kepler TM GK110*.
`https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`

[14] *CUDA Thrust documentation*.
`http://docs.nvidia.com/cuda/thrust/`

[15] *NVLabs CUB documentation.*
https://nvlabs.github.io/cub/

[16] *Bitonic sort wikipedia.*
https://en.wikipedia.org/wiki/Bitonic_sorter

[17] Lang, Hans Werner. *Bitonic sorting network for n not a power of 2.*
http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm

[18] *CUDA Toolkit 6.0 release notes.* April 2014.
http://developer.download.nvidia.com/compute/cuda/6_0/rel/docs/CUDA_Toolkit_Release_Notes.pdf

[19] *CUDA Toolkit 7.5 release notes.* September 2015.
http://developer.download.nvidia.com/compute/cuda/7.5/Prod/docs/sidebar/CUDA_Toolkit_Release_Notes.pdf

[20] *ISO International Standard ISO/IEC 14882:2014(E) – Programming Language C++.*

[21] Karimi, Kamran. Dickson, Neil G. Hamze, Firas. *A Performance Comparison of CUDA and OpenCL*

[22] *Petititon for Better OpenCL support in NVIDIA's CUDA SDK*
http://www.ipetitions.com/petition/opencl-examples-in-cuda-5-sdk

Appendices

A. Contents of the enclosed CD

```
project
├── code (thesis source codes)
│   ├── cmake (CMake utilities)
│   ├── CxxTinyTest (test framework)
│   ├── CxxTools (common C++ utilities)
│   └── NewRF (Random Forests implementation and resources)
├── run
│   └── release-unix-newrf.sh
└── thesis
    └── resources (raw measurements, spreadsheets, graphics, ...)
```

B. Programming documentation

The fine-grained and the monolithic CUDA implementations (called as NewRF project) are created in C++. The C++ toolchain used for development and testing is GCC, version 4.8.5. The build system used is CMake, version 3.2.1. The CUDA device compute capability required is 2.0 and the CUDA toolkit version used is 7.5.

The NewRF project source code is divided into multiple modules (see Figure B.1).

Common

Common module contains the Random Forests data structures, input data structures and utilities for data loading. The shared Random Forests data structures used by CPU and CUDA implementations are:

```
struct node_c {
 int attribute;
 float value;
 node_c *left;
 node_c *right;
};

struct rf_c {
 std::vector< node_c* > root_s;
};
```

CPU

CPU module consists of reference CPU serial and parallel implementations. The parallel implementation reuses most of the serial implementation internal and utilizes OpenMP macros for parallelization of the algorithm.

CUDA

CUDA module contains both the fine-grained and the monolithic CUDA implementation. CUB and Thrust header libraries are included with the source code.

Where possible, implementations share common structures and functions. The distinct parts of the monolithic implementation are marked with "2nd" as a suffix of header and source files. The fine-grained implementation has no suffix.

Benchmark

Benchmark module contains code for measurements of sorting algorithms and a comparison of different Random Forests implementations and various parameters.

```
NewRF
├── source
│   ├── benchmark
│   ├── common (code common for CPU and CUDA implementations)
│   │   └── include
│   ├── cpu
│   │   └── include
│   ├── cpu-test
│   ├── cuda
│   │   ├── cub (CUB library source code)
│   │   ├── include
│   │   ├── internal (internal headers for use in CUDA tests only)
│   │   └── thrust (Thrust library source code)
│   ├── cuda-test
│   ├── data (test data)
│   ├── run (CLI interface for CPU and CUDA implementations)
│   └── test
└── CMakeLists.txt
```

Figure B.1: CPU and CUDA implementations source code structure

Build

In source

The project is described in CMake build file. It can be compiled and built using standard CMake commands in the NewRF directory. Binaries are installed inside the "install" directory.

```
$ cmake . \
 -DCMAKE_BUILD_TYPE=RelWithDebInfo \
 -DCMAKE_INSTALL_PREFIX:PATH=install
$ cmake --build . --target install
```

Script

Another option is to use an included build script in the run directory. The "release-unix-newrf.sh" shell script compiles the project out of source and installs binaries into a separate "../build/NewRF-unix" directory. Moreover, it triggers the test suite execution.

```
$ cd project/run
$ ./release-unix-newrf.sh
...
-- The C compiler identification is GNU 4.8.5
-- The CXX compiler identification is GNU 4.8.5
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
...
[==========] Tests started.
...
```

```
[==========] Tests ended. Summary: successful ...
```

Provided libraries are located in a new directory.

```
$ cd ../build/NewRF-unix/install/bin/
$ ls -lta
total 24860
drwxrwxr-x 4 haas haas      39 Jun 28 22:35 ..
drwxrwxr-x 2 haas haas     128 Jun 28 22:35 .
-rwxr-xr-x 1 haas haas 2252392 Jun 28 22:35 NewRF-run
-rwxr-xr-x 1 haas haas 2818090 Jun 28 22:35 NewRF-test
-rwxr-xr-x 1 haas haas 8933439 Jun 28 22:35 NewRF-cuda-test
-rwxr-xr-x 1 haas haas 3112346 Jun 28 22:35 NewRF-cpu-test
-rwxr-xr-x 1 haas haas 8329366 Jun 28 22:35 NewRF-benchmark
```

C. User documentation

The implementation accepts following arguments:

`./NewRf-run ...`

1. backend - Execution backend, either "cpu" or "cuda".
2. seed - Seed to use for bootstrap generation and attribute selection.
3. tree count - Tree count to train. For "cuda" backend the tree count is expected to be divisible by the amount of available devices.
4. bootstrap size - Bootstrap size.
5. split minimum - Minimum bootstrap size to stop training at.
6. data format - Input data format, either "gen" or "fv".
   (a) gen - The file is expected to start with three meta data lines. Columns are comma separated, the first column is expected to contain numerical integral labels, the rest are numerical real attributes.

```
# type:  OUT,           IN,          IN,          IN
# data: CU32,         NF32,        NF32,        NF32
# name:     ,            ,            ,
           4,  -595330.06,   672084.2,  -111428.57
           0,  -539251.06,   94618.77,  -509989.47
           3,  -634118.44, 101074.625,   -321004.7
           ...
```

   (b) fv - The attribute file contains comma separated numerical real attributes. The label file contains numerical integral labels.
7. train samples - Path to train attributes, in the specified data format.
8. train samples - Path to train labels, in the specified data format (same as path for attributes in "gen" format).
9. test samples - Path to test attributes, in the specified data format.
10. test samples - Path to test labels, in the specified data format (same as path for attributes in "gen" format).

Run

```
$ ./NewRF-run cuda 42 80 1000 1 fv \
 ../data/fv_dwt_cdf53_10_mean.mat \
 ../data/fv_titles.txt \
 ../data/fv_dwt_cdf53_10_mean.mat \
 ../data/fv_titles.txt
train: attributes=9 size=1565 in=... out=...
test : attributes=9 size=1565 in=... out=...
train duration: 509ms
test  duration: 4ms
rate          : 0.996805
```

```
$ ./NewRF-run cpu 42 80 1000 1 gen \
 ../data/data5-0-0.txt \
 ../data/data5-0-0.txt \
 ../data/data5-0-1.txt \
 ../data/data5-0-1.txt
train: attributes=30 size=80000 in=... out=...
test : attributes=30 size=10000 in=... out=...
train duration: 48ms
test  duration: 36ms
rate          : 0.2857
```