Random Forests for CUDA GPUs

Daniel Slat
Mikael Hellborg Lapajne

School of Computing
Blekinge Institute of Technology
Sweden
This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Daniel Slat
Address: Ågårdsgatan 5a, 372 35 Ronneby, Sweden
E-mail: daniel.slat@gmail.com

Mikael Hellborg Lapajne
Address: Vårgatan 2, 274 36 Skurup, Sweden
E-mail: mikelapajne@gmail.com

University advisor(s):
Prof. Håkan Grahn
School of Computing, BTH

Dr. Niklas Lavesson
School of Computing, BTH

School of Computing
Blekinge Institute of Technology
SE – 371 79 Karlskrona
Sweden

Internet : www.bth.se/com
Phone : +46 455 38 50 00
Fax : +46 455 38 50 53
ABSTRACT

Context. Machine Learning is a complex and resource consuming process that requires a lot of computing power. With the constant growth of information, the need for efficient algorithms with high performance is increasing. Today's commodity graphics cards are parallel multi processors with high computing capacity at an attractive price and are usually pre-installed in new PCs. The graphics cards provide an additional resource to be used in machine learning applications. The Random Forest learning algorithm which has been showed competitive within machine learning has a good potential for performance increase through parallelization of the algorithm.

Objectives. In this study we implement and review a revised Random Forest algorithm for GPU execution using CUDA.

Methods. A review of previous work in the area has been done by studying articles from several sources, including Compendex, Inspec, IEEE Xplore, ACM Digital Library and Springer Link. Additional information regarding GPU architecture and implementation specific details have been obtained mainly from documentation available from Nvidia and the Nvidia developer forums. The implemented algorithm has been benchmarked and compared with two state-of-the-art CPU implementations of the Random Forest algorithm, both regarding consumed time for training and classification and for classification accuracy.

Results. Measurements from benchmarks made on the three different algorithms are gathered showing the performance results of the algorithms for two publicly available data sets.

Conclusion. We conclude that our implementation under the right conditions is able to outperform its competitors. We also conclude that this is only true for certain data sets depending on the size of the data sets. Moreover we conclude that there is potential for further improvements of the algorithm both regarding performance as well as adaption towards a wider range of real world applications.

Keywords: CUDA, Random forests, Parallel computing, Graphics processing units
The thesis is structured according to the ‘Hybrid Master Thesis’ (HMT) format, which was proposed in the summer 2007 by members of BTH, and is still in the experimental phase. The idea of the HMT format is to have a hybrid form between an IEEE/ACM paper and a traditional master thesis. One of the reasons behind the HMT format is to increase the number of theses that can be published as papers. A further reason is to help students focus their writing and express themselves clearly.

The document is divided into two major parts. The former part (Part A) follows the IEEE/ACM structure and focuses on the most relevant areas of the thesis project. It is adapted towards publication with an intended audience in the Machine Learning community.

The latter part (Part B) is intended for a wider audience of computer scientists not necessarily familiar with Machine Learning. It consists of a series of appendixes which aim at giving the reader an introduction to the subject as well as additional details on relevant parts of the paper.
CONTENTS

PART A – PAPER

1. INTRODUCTION AND MOTIVATION .................................................................................. 3

2. BACKGROUND AND RELATED WORK .............................................................................. 4

2.1. RANDOM FOREST ........................................................................................................ 4

2.2. USING GPUs FOR ML .................................................................................................. 4

2.3. ML WITH CUDA ......................................................................................................... 5

2.4. PARALLELIZATION OF RF ON CPU ......................................................................... 5

2.5. RF ON GPU ................................................................................................................ 5

3. CUDA AND GPU ARCHITECTURE .................................................................................... 5

3.1. GPU ARCHITECTURE .................................................................................................. 5

3.2. COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA) ........................................... 6

3.3. NVIDIA GPU COMPUTE CAPABILITY ....................................................................... 7

4. IMPLEMENTATION .............................................................................................................. 7

4.1. IMPLEMENTATION OVERVIEW .................................................................................. 7

4.2. IMPLEMENTATION AND PARALLELIZATION APPROACH ...................................... 7

4.3. INFORMATION GAIN .................................................................................................. 8

4.4. ARFF READER .......................................................................................................... 8

4.5. GPU AND CUDA SPECIFIC OPTIMIZATIONS .............................................................. 8

4.5.2. Textures ................................................................................................................ 8

4.5.3. Page-locked memory ............................................................................................ 8

4.5.4. Global & Constant variables ............................................................................... 8

4.5.5. Fast Math library ................................................................................................ 8

4.5.6. Logarithmic functions ........................................................................................ 9

4.5.7. Random Number generator .................................................................................. 9

5. EXPERIMENTAL METHODOLOGY ................................................................................... 9

5.1. PLATFORM ................................................................................................................... 9

5.2. DATA SETS ................................................................................................................ 9

5.3. MEASUREMENTS AND PARAMETERS ...................................................................... 9

6. RESULTS ............................................................................................................................ 10

7. DISCUSSION ....................................................................................................................... 10

8. CONCLUSIONS AND FUTURE WORK ............................................................................ 11

9. REFERENCES ..................................................................................................................... 11

PART B - APPENDIXES

APPENDIX A: MACHINE LEARNING ....................................................................................... 20

1. TERMINOLOGY ............................................................................................................... 20

2. MACHINE LEARNING .................................................................................................... 20

2.1. ALGORITHM CLASSES .............................................................................................. 20

2.1.1. Supervised learning .............................................................................................. 20

2.1.2. Unsupervised learning ......................................................................................... 20
Random Forests for CUDA GPUs

Daniel Slat
Blekinge Institute of Technology
dasl05@student.bth.se

Mikael Hellborg Lapajne
Blekinge Institute of Technology
mihk05@student.bth.se

Abstract

Machine Learning is a complex and resource consuming process that requires a lot of computing power. With the constant growth of information, the need for efficient algorithms with high performance is increasing. Today's commodity graphics cards are parallel multi processors with high computing capacity at an attractive price and are usually pre-installed in new PCs. The graphics cards provide an additional resource to be used in machine learning applications. We present a revised Random Forests algorithm, which parallelizes the generation of decision trees and is executed on the graphics processing unit, which compared to Weka shows a speedup of up to 4.9 times for total computation time and up to 9.2 times compared to LibRF for the EULA data set.

1. Introduction and motivation

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

ML algorithms have been proven to be useful in a variety of application domains [29]. Usual ML tasks involve recognition, diagnosis, planning, prediction and classification. This can be used for a wide range of applications such as control systems in cars, face and image recognition or categorization of objects such as documents or books.

In the data mining field, machine learning algorithms are used routinely when searching for patterns in large data sets. The process of analyzing large amounts of data is complex and resource consuming and with the never-ending growth of databases; data amount doubling every three years [6], there is an increasing need for more performance.

Recent year’s performance trend for GPUs has been tremendous and it is increasing more rapidly than CPU performance [8][34][35]. The total computing power of a GPU can now vastly exceed that of a CPU. For example, the Intel Core i7 965 Desktop CPU has about 50 to 70 GFLOPS of computing power [3][12], but compared to the Nvidia GTX 295 that theoretically exceeds the TFLOPS limit [14][15], it is diminished.

Having previously only been available for graphics work the GPUs have been opened up for use in other applications. With General-purpose computing on graphics processing units (GPGPU), tasks that are normally carried out on the CPU, can be done on the GPU. With the introduction of the Nvidia Compute Unified Device Architecture (CUDA), the multi parallel processors in GPUs have become accessible in a new way. CUDA gives the opportunity to write programs for execution on GPUs much in the same way as for CPUs.

The Random Forest learning algorithm introduced by Leo Breiman [1] is a learning algorithm that has shown to perform well in many different applications. Studies have also shown that the algorithm is well suited for parallel execution. Combining a parallelized version of the Random Forest algorithm with the power of a multi parallel processor like the GPU has the potential to show good performance gains.

In this paper we will present details on a parallelized CUDA Random Forest for execution on GPU. The goal is to evaluate the performance of the algorithm in terms of execution time, compared to its CPU-based counterparts. The implementation was tested with two publicly available data sets and performance benchmarks for our implementation have been compared with two state-of-the-art CPU implementations. Here, our experimental results show that our implementation of the Random Forest algorithm, under the right conditions due to the size of the data set, is able to outperform the two CPU-based algorithms LibRF and Weka without any significant loss in classification accuracy.

The rest of the paper is organized as follows. In section 2 relevant related work done in GPU-based computing
and machine learning is presented. Following that in section 3, the GPU and CUDA architecture and important issues to consider in GPU programming are presented. Section 4 presents our implementation and details on the CUDA specific techniques used. The experimental procedure is described in section 5. In section 6 the test results presenting the performance of the algorithm are reviewed. In section 7 we discuss the results. This is then followed by a conclusion and suggestions for future work in section 8.

2. Background and related work

2.1. Random Forest

The concept of Random Forests (RF) was first introduced to the machine learning community by Leo Breiman [1]. It is an ensemble classifier consisting of decision trees. The idea behind Random Forest is to build many decision trees from the same data set using bootstrapping and randomly sampled variables to create trees with variation. The bootstrapping generates new data sets for each tree by sampling examples from the training data uniformly and with replacement. These bootstraps are then used for constructing the trees which are then combined in to a forest. This has proven to be effective for large data sets with missing attributes values [1].

Each tree is constructed by the principle of divide-and-conquer. Starting at the root node the problem is recursively broken down into sub-problems. The training instances are thus divided into subsets based on their attribute values. To decide which attribute to split upon in a node, $k$ attributes are sampled randomly for investigation. Among these candidates, the attribute that gives the best information gain is chosen as split attribute. The information gain property represents how good an attribute can separate the training instances according to their target attribute. As long as splitting gives a positive information gain, the process is repeated. If a node is not split it becomes a leaf node, and is given the class attribute that is the most common occurring among the instances that fall under this node. Each tree is grown to the largest extent possible, and there is no pruning. Classification is performed by traversing the input query instances through each tree and then each tree votes for a class for each instance. The RF considers the class with most votes among all trees as the answer to a classification query.

There are two main parameters that can be adjusted when training the RF algorithm. First, the number of trees can be set by the user, secondly there is the $k$ value, or number of attributes to consider in each split. These parameters can be tuned to optimize classification performance depending on the problem at hand. In the original paper on random forests [1], it was shown that the forest error rate depends on two things:

- The correlation between any two trees in the forest. Increasing the correlation increases the forest error rate.
- The strength of each individual tree in the forest. A tree with a low error rate is a strong classifier. Increasing the strength of the individual trees decreases the forest error rate.

Reducing $k$ reduces both the correlation and the strength. Increasing it increases both. Somewhere in between is an optimal range of $k$. By watching the classification accuracy for different settings a good value of $k$ can be found.

Creating a large number of decision trees sequentially is ineffective when they are built independently of each other. This is also true for the classification (voting) part where each tree votes sequentially. Since the trees in the forest are independently built, both the training and the voting part of the RF algorithm can be implemented for parallel execution. An RF implementation working in this way would have potential for great performance gains when the number of trees in the forest is large. Of course the same goes the other way; if the number of trees in the forest is small it would be an ineffective approach. This will become clearer when looking at our architecture and implementation in section 4 of the document.

2.2. Using GPUs for ML

Previous work in using GPUs for machine learning has shown promising results. In 2005 a neural network (NN) was ported for GPU execution which showed a promising threefold increase in performance for classification time [4]. In that study it was also concluded that algorithms such as Support Vector Machines (SVM), Convolutional Networks and Decision Trees in general were not suited for GPU execution due to their high demands on memory access.

Today the memory modules used are both faster and larger than in 2005. As the GPU architectures have been improved they have also got equipped with larger and faster shared memories and more registers. This reduces the performance penalties for memory access and makes decision trees and other algorithms interesting to evaluate. The lack of support for 32 bit floating point operations was also mentioned as a possible problem [4] when using GPUs for scientific calculations where better precision might be needed. Double float precision is now available in new GPUs and precision is no longer a problem in that
sense, even though it can generate a negative impact on performance under certain circumstances.

2.3. ML with CUDA

Some efforts have been done in studying CUDA-based ML algorithms. The SVM learning and classification algorithm has been evaluated [10] and a very large performance increase was seen. Performance for training time increased 9-35 times and 81-138 times for classification compared to LLBSVM (A Library for Support Vector Machines) running on a standard processor. In the same study another CUDA implementation of the SVM algorithm was evaluated as well which also showed large performance increases.

Another ML algorithm that has been tested in CUDA is Spectral Clustering [7]. Performance of the algorithm was compared to a CPU version and another GPU version. About ten times speedup was achieved for the CUDA version compared to their parallelized CPU version. Compared to the other GPU setup, CUDA showed a three times speedup. The results in the study also showed that memory speed and limits in the bus transfer speed between main memory and device memory can have a big impact on performance. Memory handling is an important part as this affects the way to implement an algorithm optimally depending on how large data sets will be used. Also as the bus speed is asynchronous with different speeds in transfers to and from the device, performance can vary significantly depending on the algorithm design.

When it comes to parallelization of algorithms and the large number of simultaneous threads in CUDA, management can become an obstacle. As seen in [8] a request for automatic thread spawning was stated. Currently the optimal number of threads has to be decided by the programmer and memory management has to be carefully considered both depending on the hardware and the algorithm used.

2.4. Parallelization of RF on CPU

The original implementation of the RF algorithm [16] based on the theoretical work by Breiman [1] was written in the Fortran 77 programming language and made for sequential CPU execution. However, an MPI enabled parallelized reimplementation in Fortran 90 has been done [11]. The parallelization in the Fortran 90 implementation was mainly about building many trees in parallel where the trees themselves are still built sequentially. Compared to the original implementation a speedup of three to four times was estimated. The code offers more room for optimization though and the main contribution to the original code is related to coding and design changes. The possibility of building many random forests in parallel is also mentioned. The purpose would then be to generate several random forests with different parameter settings and then choose the forest that yields the best classification performance. That way time could be saved when figuring out the optimal training settings.

The open source machine learning project Weka has made a parallelized version of their random forest implementation called FastRandomForest [17]. The FastRandomForest project aims at doing as much speed optimizations to the code as possible and compared to their ordinary sequential version it shows a speedup of 2.4 times running in single thread mode and tests presented on the project homepage [18] show a total speedup of 7 times when running four threads. The performance results of the FastRandomForest are similar to those of the Fortran90 implementation [11].

2.5. RF on GPU

As far as we know, a GPU version of the Random Forest algorithm has only been evaluated in one previous study [13]. In that study the authors used Microsoft’s Direct3D SDK and High Level Shader Language (HLSL) to implement their solution. The main contribution was in terms of classification speed, which yielded about one hundred times speedup. The results show impressive numbers; however training time was still linear and dependent on the number of training examples, the number of trees, the depth of the trees and the number of features evaluated.

We have found no previous studies where a parallelized version of the RF algorithm using CUDA has been presented. The results of previous work have shown however that the random forest algorithm is suitable for a parallelized implementation approach. GPGPU techniques have been used before to successfully speed up machine learning algorithms, and the CUDA architecture with its improvements for GPGPU programming has shown promising results.

These facts together raise the question on how well a parallelized random forest implemented in CUDA would perform. That is why we have implemented and evaluated our CUDA Random Forest (CudaRF) algorithm.

3. CUDA and GPU Architecture

3.1. GPU architecture

GPUs are parallel multi processors and the key to speed improvements is parallelization. The massive capacity for simultaneous calculations in the GPU is also growing fast with new generations of GPUs. For example,
the Nvidia GeForce GTX 200 series are equipped with up to 240 processor cores and can spawn up to 30720 concurrent threads [3]. Compared to the two generations older GeForce 8800 GTX, that could only spawn 12288 threads, it is almost a threefold increase, and we expect it to continue increasing in a rapid pace. Therefore, if ML algorithms can be parallelized and mapped to the GPU architecture effectively, there is great potential for performance increases. Furthermore, recent years improvements in GPU architectures have made 64-bit double float precision and memory scatter available; features valuable in scientific applications.

In Fig. 1 the architecture for an Nvidia CUDA GPU is shown. The GPU consists of a number of streaming multi processors (SM). Inside each of these SMs are eight CUDA cores. Each of the SMs also has a specific memory region that is shared among the cores in the same SM. This small but fast memory can be used as a cache to provide higher bandwidth than when using texture lookups [24]. It can also be used for synchronization and passing of data between threads running on the same SM.

![Fig. 1: Overview of CUDA enabled GPU architecture](image)

Not only parallelization is important when adopting an algorithm for GPU execution. It is easy to believe that parallelization will solve all speed problems and that a speedup nearly linear to the number of cores can be achieved. However, the GPU architecture differs from that of a CPU in several ways.

Before the GPU can be put to work data must be transferred from the host (CPU) to the device (GPU) via the graphics bus. In order to not get a performance loss, the bandwidth must be very large and unnecessary transfers kept to a minimum. This has been a common problem before and can still be a major bottleneck. The PCI Express x16 v2.0 theoretically supports a bandwidth of max 8GB/s [20] and was first introduced to Nvidia cards along with the G92 core (8800GT/GTS 512MB).

After taking the bandwidth between Host and Device into account, it may in some cases be more efficient to execute functions on the device even when they could be executed faster on the host, or vice versa, just to avoid transferring data between the two [21].

3.2. Compute Unified Device Architecture (CUDA)

The Compute Unified Device Architecture (CUDA) was developed by NVIDIA and is in part an API extension to C. CUDA contains the Parallel Thread Execution (PTX) Instruction Set Architecture (ISA) and the parallel compute engine inside the GPU, and brings these powerful tools closer to the developer. CUDA provides access to the SMs on the GPU so they can be used for other tasks than graphics calculations. With C-like code it is possible to program the SMs on the GPU in a similar way to writing threaded code for CPUs.

CUDA is available in all NVIDIA GPUs since the G80 (8000-series). With CUDA developers can use high level languages such as C and C++ to make use of the highly parallel NVIDIA GPUs for various computing tasks. Other languages such as Python, Java, Fortran, Matlab and Microsoft .NET Framework are available through native method bindings [19][32][33]. CUDA has some advantages over other available GPGPU techniques. Among other things, it supports scattered reads, which means that code can be read from arbitrary addresses in memory. Communication between threads has also been improved with the introduction of the shared memory region in the SM.

CUDA provides a simplified way for non-graphic tasks to be executed on the GPU, and by this hopefully offloading the CPU and speedup application execution. However, a GPU is designed differently than a CPU and not all programs can benefit from being executed on GPUs.

A CUDA enabled GPU create, manage, schedule and execute threads in groups of 32, called warps [19]. 32 is the size of a warp on an Nvidia GPU of compute capability 1.2 [21] and to achieve maximum performance, the number of threads running should be a multiple of 32. But equally important, to utilize the full potential of the GPU, the total number of threads running should be in the thousands. This differs a lot from programming on a regular CPU with only a few processing cores and a low amount of concurrent threads. In CUDA the threads must be handled manually by the programmer.
3.3. Nvidia GPU Compute capability

Nvidia GPUs are constantly improving and new features are added on a regular basis. This makes it important to take into account the used GPUs compute capability.

The latest compute capability is version 1.3 which was introduced with the Nvidia GTX200-series. Every time a new compute capability version is introduced newer features are added and old ones are improved. For example, compute capability 1.2 doubled the amount of registers per multiprocessor, also the number of active warps per multiprocessor was increased with 50% and so was the number of active threads per SM [19]. With compute capability 1.3 the support for double-precision floating-point numbers were introduced. These changes between different versions are important to consider when designing programs for CUDA. You must carefully consider which attributes to store in registers and caches for the program to perform well and this can be confusing for a developer new to GPGPU development. In Table 1 the characteristics for compute capability 1.2 which is used in this study can be seen.

Table 1: Relevant characteristics and constraints for compute capability 1.2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers per multiprocessor</td>
<td>16384</td>
</tr>
<tr>
<td>Threads per warp</td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Shared memory per multiprocessor</td>
<td>16KB</td>
</tr>
<tr>
<td>Amount of constant memory</td>
<td>64KB</td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16KB</td>
</tr>
<tr>
<td>Cache working set per SM for constant memory</td>
<td>8KB</td>
</tr>
<tr>
<td>Cache working set per SM for texture memory</td>
<td>6-8KB</td>
</tr>
<tr>
<td>Maximum number of active blocks per SM</td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of active threads per SM</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum texture width (one dimensional texture)</td>
<td>$2^{13}$</td>
</tr>
<tr>
<td>Support for atomic functions operating in shared memory and atomic functions</td>
<td></td>
</tr>
</tbody>
</table>

4. Implementation

4.1. Implementation overview

In Fig. 2 an overview of the execution flow of CudaRF is presented. The execution starts with the host reading the input data stored in ARFF files from disk. The data is formatted by the host and fills are added for missing attribute values. All data is then transferred from the host to the device memory. The device runs a parallel kernel for the bagging process where each tree gets a list of which instances to use. Instances not used by a tree are considered as the out of bag (oob) instances for that tree. The actual growing of the forest is done in the second kernel where all nodes and branches for every tree are created. The trees are built in parallel and each tree is built by its own thread. When the forest is completely built, each tree performs a classification run on its oob instances. The results of the oob run are transferred back to the host for calculation of the oob error rate.

After the training phase is complete, the query instances are sent to the device for classification. Each tree then makes a prediction for every query instance. Predictions are done in parallel by the trees and each tree is executed by a separate thread. When all predictions have been made, they are transferred back to the host where the votes are summarized. The results are then presented to the user and saved to disk.

Fig. 2: Execution flow and communication between Host and Device for CudaRF.

4.2. Implementation and parallelization approach

When looking at other RF implementations such as the Weka Random Forest [22], the trees in the forest are constructed recursively. This is a logical and straight on approach, and in our initial implementation design we decided to use recursive methods as well. Unfortunately CUDA did not agree with this, since there is no support...
for recursion in device kernels [19], thus we had to redesign our algorithm for iterative tree generation.

We decided that each tree is built by only one CUDA thread. This is similar to the approach used by Topic et al [11]. All trees are built in parallel to each other but the trees themselves are built sequentially. The decision to construct each tree sequentially comes from the fact that each node created in a tree is dependent on subsequent nodes. Since CUDA works well with a large number of threads, our implementations full potential is only reached when building a large forest.

4.3. Information gain

In our implementation, we did not use the Gini importance calculation proposed in [1] for node splitting. Instead entropy calculations are done to determine the information gain. This has the advantage of moving computation time from training to classification. Also instead of searching among all possible values for an optimal split value, the middle value of those instances affected by the split is used. This way a suitable split value can be found even faster. As shown in the results section, this method does not result in degraded classification accuracy.

4.4. ARFF reader

Testing data is read from ARFF files which is an open source database format specifically designed for machine learning data sets [25], and a custom ARFF file reader has been implemented. This is advantageous since we now have the ability to read data sets similar to those used in other RF implementations. Although the ARFF reader does not support all ARFF data sets, it does supply us with the functions we need to successfully compare our implementation to other CPU RF implementations. The ARFF reader can easily be improved with more features if needed.

4.5. GPU and CUDA specific optimizations

Since memory and bus speed can affect the total performance of the program, we made use of several techniques to improve memory transfers between host and device and also to minimize the use of the rather slow global device memory.

4.5.2. Textures

In our implementation we use texture arrays to store the training data for the forest. The data is copied to the device into a one dimensional read-only texture array localized in the global device memory. These texture arrays are read only, but since they are cached (which the rest of the global memory is not) this improves the speed of fetching data. A possible way of increasing performance further might be to use a two-dimensional texture array instead of a one-dimensional because CUDA is better optimized for 2D arrays and the size limit for the array would increase substantially [19].

4.5.3. Page-locked memory

We also use page-locked or pinned memory, on the host where it is possible. For example when reading the ARFF file data, the cudaHostAlloc() call is used instead of a regular memory allocation (malloc). This is to allocate memory as page locked on the host, which prevents the operating system from paging out memory if needed. When page locked memory is used the PCI-E bandwidth has a theoretical limit of 8GB/s instead of the regular 4GB/s.

A possible disadvantage of using page locked memory is that if the host needs to page out memory because of other processes running on the machine it cannot do so and system performance may degrade. This is not relevant in our case since all tests are run on a dedicated machine.

4.5.4. Global & Constant variables

For storing commonly accessed variables we make use of the device constant memory as much as possible. This is primarily to preserve registers, but since the constant memory is cached it is also faster than the global memory [19]. The size of the constant memory is very limited though, only certain variables can be put there and larger things like the training data does not fit.

To further preserve registers and shared memory, the number of attributes passed to each method/kernel is kept to a minimum or stored on constant memory as recommended in the best practices guide from Nvidia [21].

4.5.5. Fast Math library

To increase performance for mathematical functions we made use of the fast math library available in CUDA. The fast math library works by doing a fast approximation of values instead of a regular calculation. This gives a less precise result in some calculations but is faster than the ordinary math libraries. For instance, the throughput of a single-precision floating-point division on CUDA is 0.88 operations per clock cycle. But the faster __fdividef(x, y) has a higher throughput of 1.6 operations per clock cycle [19]. The precision of the fast math library is more than enough for the calculations done in our implementation.
4.5.6. Logarithmic functions

Logarithmic functions are used in the calculation of the information gain when performing node splits. These calculations are done several times in each node that is investigated, and instead of using the regular log function (logf()), we use the faster less precise __logf(). We expect that the small loss in precision is not significant for our classification accuracy and instead focus on achieving higher speed. This decision was based on that RF is built on sampled variables and chance, and therefore a less precise sampling is not to impact the outcome of the classifier. Among the logarithmic calculations used, log2 is commonly occurring and by replacing this computation with a pre defined value, it could be removed completely.

4.5.7. Random Number generator

One major part of the RF algorithm is the large amount of random number samples. In CUDA this became a problem since there is no random number generator present in the CUDA math library (or any other standard CUDA library for that matter). Therefore, we had to implement our own random number generator, for this we used the Mersenne Twister [23] example from the CUDA SDK which we then adopted to work together with our program. This also enabled us to generate random numbers in parallel.

5. Experimental methodology

The main aim of the experiment is to compare the execution time of CudaRF with two CPU-based RF implementations. The classification accuracy is also measured to ensure that the speed specific optimizations used in CudaRF did not degrade the classification accuracy of the algorithm. The algorithms used in addition to CudaRF are the C++ Random Forest library [30] and the Weka Random Forest [31].

5.1. Platform

Two different testing systems were used for the experiment tests. System 1 was used for CPU tests and System 2 for GPU tests. Since our CUDA algorithm executes on the GPU and the other two algorithms on the CPU, using two different systems did not pose a problem. The components in the two systems are quite similar to each other and we do not expect them to give an advantage for any of the algorithms.

**System 1:**
OS: Windows 7 64-bit with Nvidia driver 197.45
CPU: Intel Core2Quad Q6600
RAM: 4GB DDR2
GPU: Gigabyte GT220 1GB GDDR3
CUDA version: 2.3

The specifications for the GPU are as follows:
- Compute capability 1.2
- 48 CUDA cores (6 SMs)
- Core processor clocked at 625MHz
- Shader processor clocked at 1360MHz
- Memory clocked at 790MHz
- Total amount of memory is 1GB GDDR3.
- Memory interface is 128-bit.
- Memory bandwidth is 25.3GB/s.

**System 2:**
OS: Windows 7 64-bit
CPU: Intel Core i7 920
RAM: 6GB DDR3

5.2. Data sets

For the experiment tests two publicly available data sets with different characteristics were used, the end user license agreement (EULA) [26] and Spambase [27].

The EULA data set consists of 996 instances defined by 1265 numerical attributes and one nominal target attribute. The EULA data set is used for distinguishing spyware from legitimate software by finding patterns in the EULAs.

Spambase is available in the UCI Machine Learning repository [27] and is used for classifying e-mail as spam or non-spam based on the content of the e-mails. Spambase consists of 4601 instances with 57 numerical attributes and one nominal target attribute.

5.3. Measurements and parameters

The tests were divided into two main parts:

- Performance tests for training and classification time.
- Tests for determining the classification accuracy.

The tests were performed for several different parameter configurations. The two parameters that were changed for the random forest are the number of trees ($t$) and the number of attributes to consider in each split ($k$). These two parameters are typically tuned for optimal classification accuracy depending on the data set used. The parameters were independently changed to observe their impact on execution time. Measurements were collected for $t=1.....256$ with an exponential step size and...
Performance tests were run for the three algorithms; CudaRF, Weka and LibRF. Both data sets, EULA and Spambase, were tested. The training and classification tests measuring computation time were run on the complete data sets with all instances and each test was iterated ten times to get stable values. The results presented in section 6 are the average values of the ten iterations made for each parameter configuration. Measurements for CudaRF were performed on system 1, whereas the measurements for LibRF and Weka were performed on system 2.

Included within the time measurements are allocation of memory, transferring of data between host and device, complete training and classification, compilation of results and finally freeing of memory.

In order to measure the classification accuracy of CudaRF, we used 10 fold cross validation which has showed useful for determining classification performance [28]. The same tests were also run on Weka and each test was run with ten iterations for every parameter configuration. LibRF has no function for 10 fold cross validation and were used only for comparison of the execution time.

Furthermore, to analyze how the number of instances in a data set impacts the execution time of CudaRF, three additional measurements were performed. These were carried out on three different stripped down versions of the Spambase dataset, containing 25%, 50% and 75% of the instances. The instances removed from the original set were randomly chosen, and the tests were run with \( t=1,\ldots,128 \) and \( k=6 \) (default value).

6. Results

The experiment results show that CudaRF outperforms both LibRF and Weka regarding total execution time for the EULA data set. With \( t \) values greater than 10 we see that CudaRF is much more effective than the other two algorithms and the parallelization approach for the tree generation works well in minimizing computation time. A maximum speed up of 9.2 times for CudaRF compared to LibRF is seen with \( t=128 \) and \( k=6 \). Compared to Weka, CudaRF shows a maximum speed up of 4.9 times with \( t=256 \) and \( k=21 \).

The three-dimensional diagrams in Fig. 3 show how execution time depends on the \( t \) and \( k \) values for all three algorithms and both data sets tested. For EULA the lowest execution time was found for \( k=6 \) for CudaRF and \( k=11 \) for LibRF. For Weka the lowest \( k \) value gives the lowest execution time. We also see that greater \( t \) values benefit our algorithm compared to the other two tested where a larger increase in execution time is observed when \( t \) is growing.

When looking at the results in Fig. 5 and Table 3, it is clear however that CudaRF performs worse than both its CPU-based counterparts for the Spambase data set. Spambase contains roughly about four times as many instances as EULA (4601 compared to 996) and in Fig. 6 we can see how the number of instances in the set impacts the CudaRF algorithm with an increasing execution time. For the full data set the average execution time is almost 7.5 times higher than when running with 25% of the instances. The reasons behind this are discussed in section 7.

By comparing Fig. 4 and Fig. 5 we can observe that LibRF is more sensitive to the number of attributes in the data set than Weka and CudaRF. For the Spambase data set LibRF is almost equal in performance to Weka but for EULA which has over 1200 attributes LibRF is by far the slowest algorithm. We also note that for Spambase the lowest \( k \) value results in the lowest execution time for all three algorithms.

Table 4 and 5 shows the classification accuracy for CudaRF and Weka for both data sets and we conclude that no significant difference between Weka and CudaRF regarding classification accuracy was found.

7. Discussion

The global device memory can be considered as very slow compared to the other memory regions on the GPU. According to Nvidia, global memory fetches has a latency of about 400-600 clock cycles [21]. It is clearly visible in the results that it slows down our algorithm when the number of instances in the dataset is large. This problem is difficult to get around. Due to the limited amount of shared memory these memories cannot be used for storing the training data. All trees in the forest must also have access to the full data set, which means that the data set cannot be split into smaller parts. The result is that each tree must go to the global memory to retrieve and write values.

Nvidia is aware of the limitations of the global memory and has equipped their GPUs with two caches; constant- and texture cache. By making use of the texture cache for the training data we managed to get a performance increase for the global memory reads. Our tests indicated an approximate performance increase of
30% when textures were used. This may seem modest when considering that the texture cache is several times faster than the global memory. This can likely be explained by the fact that different threads retrieve different data in global memory leading to frequent cache misses. Admittedly, we did have an advantage of the texture cache, this is most certainly due to the effect of the spatial locality that occurs among the trees but cache misses will probably be even more frequent with larger data sets and more trees.

Compared to CudaRF, Weka and LibRF show a much larger increase in execution time with a growing number of trees. This is not surprising as the many cores in the GPU give CudaRF an advantage for parallelized tasks. In the case of CudaRF running the EULA data set we see that it handles the number of trees well.

However, the size of the data set used has a big impact on the execution time for CudaRF. The high memory latency on the GPU is likely the cause for this and the CPU implementations are not affected in the same way thanks to their fast memory access. The biggest performance increase for CudaRF can therefore be seen with smaller data sets than Spambase and with many trees, preferably over 100 and a large value for \( k \). We believed it would perform better for larger data sets like Spambase as well, and that the memory issues would be compensated with the parallelism among the trees, unfortunately this was not the case.

Another thing that the GPU is sensitive to is when threads diverge too much. Normally the GPU performs best when the same execution path is taken by all threads and the use of conditional statements like if/else can drag down performance. The kernel used for the tree building in our program is relatively large and in several places the threads may diverge. The actual design of RF makes every tree behave differently and this can be hard to overcome.

8. Conclusions and future work

We have presented a new parallelized implementation of the Random Forest algorithm using the Compute Unified Device Architecture. Our experimental tests show that it is able to outperform both the Weka and LibRF algorithms for the EULA data set containing 996 instances. CudaRF is also better suited than LibRF and Weka for forests containing many trees and with the use of relatively large \( k \) values (number of attributes to be tested at each split). The results also show that the computation time for CudaRF heavily depends on the number of instances in the data set and for the Spambase data set containing 4601 instances our algorithm does not perform better than LibRF or Weka. No significant difference in classification accuracy between CudaRF and Weka was found.

CudaRF can be improved further and for future work we suggest to further parallelize the tree generation. Though several techniques have been used to address limitations in the CUDA memory model, CudaRF can be further optimized by using asynchronous and concurrent calls for memory operations and bus transfers. We also aim at extending the ARFF reader with more features. For example, the file reader currently only supports numeric input values, and to extend it with more options would make CudaRF applicable on a wider range of data sets and real world applications.

9. References


Fig. 3: Total execution time (training and testing) for CudaRF, LiRF and Weka for the Eula and Spambase data sets.
<table>
<thead>
<tr>
<th>Librf</th>
<th>CudaRF</th>
<th>Weka</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>1 6 11 16 21</td>
<td>1 6 11 16 21</td>
</tr>
<tr>
<td>t</td>
<td>Total time</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>---------------</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>389 463 479 458 464</td>
<td>1360 971 1354 1520 1403</td>
</tr>
<tr>
<td>2</td>
<td>782 992 874 859 866</td>
<td>2903 1923 2049 2579 2589</td>
</tr>
<tr>
<td>4</td>
<td>1501 1759 1768 1753 1730</td>
<td>2708 1628 2153 2309 2533</td>
</tr>
<tr>
<td>8</td>
<td>3059 3490 3317 3207 3353</td>
<td>2917 1780 2031 2349 2713</td>
</tr>
<tr>
<td>16</td>
<td>6210 6922 6752 6632 6377</td>
<td>3397 1842 2124 2401 2669</td>
</tr>
<tr>
<td>32</td>
<td>12299 13751 13611 14027 12871</td>
<td>4155 1947 2275 2521 2880</td>
</tr>
<tr>
<td>64</td>
<td>23344 27690 27408 26336 26217</td>
<td>6260 3565 4106 4623 5249</td>
</tr>
<tr>
<td>128</td>
<td>50497 51650 52541 56629 52596</td>
<td>9850 5602 6386 7359 8303</td>
</tr>
<tr>
<td>256</td>
<td>98678 94378 85070 96199 103678</td>
<td>17404 10679 12110 13883 15840</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Librf</th>
<th>CudaRF</th>
<th>Weka</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>1 6 11 16 21</td>
<td>1 6 11 16 21</td>
</tr>
<tr>
<td>t</td>
<td>Training time</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>---------------</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>368 446 449 442 443</td>
<td>1274 919 1309 1478 1371</td>
</tr>
<tr>
<td>2</td>
<td>748 969 856 844 850</td>
<td>2788 1851 1992 2520 2535</td>
</tr>
<tr>
<td>4</td>
<td>1481 1738 1748 1730 1708</td>
<td>2581 1570 2091 2258 2491</td>
</tr>
<tr>
<td>8</td>
<td>3033 3461 3293 3186 3332</td>
<td>2786 1716 1973 2297 2660</td>
</tr>
<tr>
<td>16</td>
<td>6179 6881 6710 6596 6335</td>
<td>3260 1774 2065 2346 2619</td>
</tr>
<tr>
<td>32</td>
<td>12234 13678 13544 13957 12809</td>
<td>4009 1876 2213 2463 2825</td>
</tr>
<tr>
<td>64</td>
<td>23236 27561 27291 26225 26107</td>
<td>6113 3488 4040 4564 5191</td>
</tr>
<tr>
<td>128</td>
<td>50304 51407 52326 56422 52393</td>
<td>9580 5470 6269 7252 8200</td>
</tr>
<tr>
<td>256</td>
<td>98317 93914 84650 95797 103289</td>
<td>16995 10469 11933 13717 15682</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Librf</th>
<th>CudaRF</th>
<th>Weka</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>1 6 11 16 21</td>
<td>1 6 11 16 21</td>
</tr>
<tr>
<td>t</td>
<td>Testing time</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>---------------</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>21 18 30 16 21</td>
<td>86 52 46 42 32</td>
</tr>
<tr>
<td>2</td>
<td>33 23 18 16 16</td>
<td>115 71 57 59 54</td>
</tr>
<tr>
<td>4</td>
<td>20 21 20 23 22</td>
<td>127 58 62 52 43</td>
</tr>
<tr>
<td>8</td>
<td>26 29 24 21 20</td>
<td>131 64 58 51 53</td>
</tr>
<tr>
<td>16</td>
<td>31 41 42 36 42</td>
<td>137 68 59 55 50</td>
</tr>
<tr>
<td>32</td>
<td>66 73 67 70 62</td>
<td>146 72 62 57 55</td>
</tr>
<tr>
<td>64</td>
<td>108 130 117 111 111</td>
<td>147 76 66 59 58</td>
</tr>
<tr>
<td>128</td>
<td>193 243 215 208 203</td>
<td>270 133 117 107 103</td>
</tr>
<tr>
<td>256</td>
<td>361 465 420 402 388</td>
<td>409 209 178 165 158</td>
</tr>
</tbody>
</table>
Table 3: Execution time (ms) with Spambase for $t=1,...,128$ and $k=1,...,16$

| k | Librf 1 1775 1734 1762 1789 | 1775 1734 1762 1789 |
|   | CudaRF 6159 22574 34706 43344 | 6159 22574 34706 43344 |
|   | Weka 9249 26229 33032 46549 | 9249 26229 33032 46549 |
|   | Total time 1724 1680 1708 1739 | 1724 1680 1708 1739 |
| t | 1 1724 1680 1708 1739 | 1724 1680 1708 1739 |
|   | 2 1911 1367 1309 1189 | 1911 1367 1309 1189 |
|   | 4 1995 1003 1067 1401 | 1995 1003 1067 1401 |
|   | 8 1543 1499 1754 2114 | 1543 1499 1754 2114 |
|   | 16 1828 3150 3001 3494 | 1828 3150 3001 3494 |
|   | 32 3121 5204 6384 7644 | 3121 5204 6384 7644 |
|   | 64 5608 10124 12265 14766 | 5608 10124 12265 14766 |
|   | 128 10981 20118 24416 29272 | 10981 20118 24416 29272 |
|   | Training time 4058 9912 19456 29466 | 4058 9912 19456 29466 |
|   | 2 1911 1367 1309 1189 | 1911 1367 1309 1189 |
|   | 4 1995 1003 1067 1401 | 1995 1003 1067 1401 |
|   | 8 1543 1499 1754 2114 | 1543 1499 1754 2114 |
|   | 16 1828 3150 3001 3494 | 1828 3150 3001 3494 |
|   | 32 3121 5204 6384 7644 | 3121 5204 6384 7644 |
|   | 64 5608 10124 12265 14766 | 5608 10124 12265 14766 |
|   | 128 10981 20118 24416 29272 | 10981 20118 24416 29272 |
|   | Testing time 435 401 368 313 | 435 401 368 313 |
|   | 2 62 58 64 56 | 62 58 64 56 |
|   | 4 73 73 72 69 | 73 73 72 69 |
|   | 8 100 114 103 103 | 100 114 103 103 |
|   | 16 162 178 164 164 | 162 178 164 164 |
|   | 32 293 311 300 281 | 293 311 300 281 |
|   | 64 535 563 527 504 | 535 563 527 504 |
|   | 128 1023 1117 1045 1006 | 1023 1117 1045 1006 |
|   | Total time 1775 1734 1762 1789 | 1775 1734 1762 1789 |
|   | 2 1973 1424 1373 1245 | 1973 1424 1373 1245 |
|   | 4 2068 1076 1139 1470 | 2068 1076 1139 1470 |
|   | 8 1643 1613 1856 2217 | 1643 1613 1856 2217 |
|   | 16 1991 3327 3565 4107 | 1991 3327 3565 4107 |
|   | 32 3415 5515 6683 7925 | 3415 5515 6683 7925 |
|   | 64 6143 10687 12792 15269 | 6143 10687 12792 15269 |
|   | 128 12004 21235 25461 30278 | 12004 21235 25461 30278 |
|   | Training time 1724 1680 1708 1739 | 1724 1680 1708 1739 |
|   | 2 1911 1367 1309 1189 | 1911 1367 1309 1189 |
|   | 4 1995 1003 1067 1401 | 1995 1003 1067 1401 |
|   | 8 1543 1499 1754 2114 | 1543 1499 1754 2114 |
|   | 16 1828 3150 3001 3494 | 1828 3150 3001 3494 |
|   | 32 3121 5204 6384 7644 | 3121 5204 6384 7644 |
|   | 64 5608 10124 12265 14766 | 5608 10124 12265 14766 |
|   | 128 10981 20118 24416 29272 | 10981 20118 24416 29272 |
Fig. 4: Average execution time for LibRF, Weka and CudaRF for the EULA data set.

Fig. 5: Average execution time for LibRF, Weka and CudaRF for the Spambase data set.

Fig. 6: Average execution time for CudaRF depending on the size of the data set.
Table 4: Classification accuracy for CudaRF and Weka for the EULA data set.

<table>
<thead>
<tr>
<th>Trees</th>
<th>K</th>
<th>Average accuracy (%)</th>
<th>Standard deviation</th>
<th>Trees</th>
<th>K</th>
<th>Average accuracy (%)</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>88,74</td>
<td>2,79</td>
<td>1</td>
<td>1</td>
<td>88,40</td>
<td>0,54</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>91,78</td>
<td>1,82</td>
<td>2</td>
<td>1</td>
<td>85,56</td>
<td>0,68</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>92,65</td>
<td>1,51</td>
<td>4</td>
<td>1</td>
<td>91,44</td>
<td>0,60</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>93,24</td>
<td>1,54</td>
<td>8</td>
<td>1</td>
<td>92,06</td>
<td>0,37</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>93,32</td>
<td>1,79</td>
<td>16</td>
<td>1</td>
<td>91,65</td>
<td>0,31</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>93,45</td>
<td>1,68</td>
<td>32</td>
<td>1</td>
<td>91,29</td>
<td>0,28</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>93,48</td>
<td>1,74</td>
<td>64</td>
<td>1</td>
<td>91,13</td>
<td>0,23</td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>93,40</td>
<td>1,63</td>
<td>128</td>
<td>1</td>
<td>90,94</td>
<td>0,21</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>93,39</td>
<td>1,70</td>
<td>256</td>
<td>1</td>
<td>90,73</td>
<td>0,10</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>90,25</td>
<td>2,49</td>
<td>1</td>
<td>6</td>
<td>89,71</td>
<td>1,14</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>92,43</td>
<td>1,83</td>
<td>2</td>
<td>6</td>
<td>87,67</td>
<td>1,20</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>93,17</td>
<td>1,94</td>
<td>4</td>
<td>6</td>
<td>91,70</td>
<td>0,62</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>93,88</td>
<td>2,00</td>
<td>8</td>
<td>6</td>
<td>93,45</td>
<td>0,57</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>94,03</td>
<td>1,83</td>
<td>16</td>
<td>6</td>
<td>93,41</td>
<td>0,34</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>94,21</td>
<td>1,93</td>
<td>32</td>
<td>6</td>
<td>93,62</td>
<td>0,37</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>94,34</td>
<td>1,75</td>
<td>64</td>
<td>6</td>
<td>93,71</td>
<td>0,19</td>
</tr>
<tr>
<td>128</td>
<td>6</td>
<td>94,45</td>
<td>1,78</td>
<td>128</td>
<td>6</td>
<td>93,69</td>
<td>0,14</td>
</tr>
<tr>
<td>256</td>
<td>6</td>
<td>94,47</td>
<td>1,68</td>
<td>256</td>
<td>6</td>
<td>93,64</td>
<td>0,14</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>90,09</td>
<td>2,62</td>
<td>1</td>
<td>11</td>
<td>89,95</td>
<td>0,95</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>92,49</td>
<td>1,93</td>
<td>2</td>
<td>11</td>
<td>87,86</td>
<td>0,98</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>93,16</td>
<td>1,77</td>
<td>4</td>
<td>11</td>
<td>92,51</td>
<td>0,55</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>93,79</td>
<td>1,85</td>
<td>8</td>
<td>11</td>
<td>93,42</td>
<td>0,42</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>94,24</td>
<td>1,74</td>
<td>16</td>
<td>11</td>
<td>93,82</td>
<td>0,37</td>
</tr>
<tr>
<td>32</td>
<td>11</td>
<td>94,46</td>
<td>1,81</td>
<td>32</td>
<td>11</td>
<td>93,96</td>
<td>0,22</td>
</tr>
<tr>
<td>64</td>
<td>11</td>
<td>94,59</td>
<td>1,71</td>
<td>64</td>
<td>11</td>
<td>94,07</td>
<td>0,21</td>
</tr>
<tr>
<td>128</td>
<td>11</td>
<td>94,76</td>
<td>1,68</td>
<td>128</td>
<td>11</td>
<td>93,87</td>
<td>0,20</td>
</tr>
<tr>
<td>256</td>
<td>11</td>
<td>94,63</td>
<td>1,70</td>
<td>256</td>
<td>11</td>
<td>93,95</td>
<td>0,12</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>90,43</td>
<td>2,59</td>
<td>1</td>
<td>16</td>
<td>90,02</td>
<td>0,79</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>92,32</td>
<td>1,91</td>
<td>2</td>
<td>16</td>
<td>88,05</td>
<td>0,83</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>93,36</td>
<td>1,95</td>
<td>4</td>
<td>16</td>
<td>92,14</td>
<td>0,64</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>93,93</td>
<td>1,85</td>
<td>8</td>
<td>16</td>
<td>93,38</td>
<td>0,38</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>94,40</td>
<td>1,78</td>
<td>16</td>
<td>16</td>
<td>93,70</td>
<td>0,49</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>94,76</td>
<td>1,79</td>
<td>32</td>
<td>16</td>
<td>93,89</td>
<td>0,22</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>94,75</td>
<td>1,71</td>
<td>64</td>
<td>16</td>
<td>94,05</td>
<td>0,20</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
<td>94,75</td>
<td>1,76</td>
<td>128</td>
<td>16</td>
<td>94,11</td>
<td>0,24</td>
</tr>
<tr>
<td>256</td>
<td>16</td>
<td>94,89</td>
<td>1,79</td>
<td>256</td>
<td>16</td>
<td>94,21</td>
<td>0,14</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>90,26</td>
<td>2,66</td>
<td>1</td>
<td>21</td>
<td>90,23</td>
<td>0,89</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>92,73</td>
<td>1,92</td>
<td>2</td>
<td>21</td>
<td>88,58</td>
<td>0,57</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>93,67</td>
<td>1,80</td>
<td>4</td>
<td>21</td>
<td>92,13</td>
<td>0,49</td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>94,43</td>
<td>1,91</td>
<td>8</td>
<td>21</td>
<td>93,77</td>
<td>0,35</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
<td>94,54</td>
<td>1,94</td>
<td>16</td>
<td>21</td>
<td>94,12</td>
<td>0,35</td>
</tr>
<tr>
<td>32</td>
<td>21</td>
<td>94,85</td>
<td>1,84</td>
<td>32</td>
<td>21</td>
<td>93,99</td>
<td>0,30</td>
</tr>
<tr>
<td>64</td>
<td>21</td>
<td>94,88</td>
<td>1,74</td>
<td>64</td>
<td>21</td>
<td>94,27</td>
<td>0,26</td>
</tr>
<tr>
<td>128</td>
<td>21</td>
<td>94,94</td>
<td>1,68</td>
<td>128</td>
<td>21</td>
<td>94,33</td>
<td>0,14</td>
</tr>
<tr>
<td>256</td>
<td>21</td>
<td>95,00</td>
<td>1,63</td>
<td>256</td>
<td>21</td>
<td>94,32</td>
<td>0,18</td>
</tr>
<tr>
<td>Trees</td>
<td>K</td>
<td>Average accuracy (%)</td>
<td>Standard deviation</td>
<td>Trees</td>
<td>K</td>
<td>Average accuracy (%)</td>
<td>Standard deviation</td>
</tr>
<tr>
<td>-------</td>
<td>---</td>
<td>-----------------------</td>
<td>--------------------</td>
<td>-------</td>
<td>---</td>
<td>-----------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>74.25</td>
<td>3.75</td>
<td>1</td>
<td>1</td>
<td>88.44</td>
<td>1.67</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>79.01</td>
<td>1.57</td>
<td>2</td>
<td>1</td>
<td>87.66</td>
<td>1.64</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>83.73</td>
<td>0.82</td>
<td>4</td>
<td>1</td>
<td>92.15</td>
<td>1.20</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>84.87</td>
<td>1.43</td>
<td>8</td>
<td>1</td>
<td>94.27</td>
<td>1.01</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>86.41</td>
<td>0.85</td>
<td>16</td>
<td>1</td>
<td>95.20</td>
<td>0.91</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>87.50</td>
<td>0.60</td>
<td>32</td>
<td>1</td>
<td>95.55</td>
<td>0.85</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>87.69</td>
<td>0.37</td>
<td>64</td>
<td>1</td>
<td>95.72</td>
<td>0.87</td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>87.73</td>
<td>0.59</td>
<td>128</td>
<td>1</td>
<td>95.77</td>
<td>0.85</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>88.22</td>
<td>0.59</td>
<td>1</td>
<td>6</td>
<td>90.27</td>
<td>1.27</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>87.56</td>
<td>0.24</td>
<td>2</td>
<td>6</td>
<td>89.46</td>
<td>1.08</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>91.44</td>
<td>0.36</td>
<td>4</td>
<td>6</td>
<td>93.17</td>
<td>1.15</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>93.55</td>
<td>0.19</td>
<td>8</td>
<td>6</td>
<td>94.64</td>
<td>1.03</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>94.21</td>
<td>0.27</td>
<td>16</td>
<td>6</td>
<td>95.26</td>
<td>0.92</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>94.41</td>
<td>0.10</td>
<td>32</td>
<td>6</td>
<td>95.48</td>
<td>0.87</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>94.62</td>
<td>0.21</td>
<td>64</td>
<td>6</td>
<td>95.67</td>
<td>0.85</td>
</tr>
<tr>
<td>128</td>
<td>6</td>
<td>94.66</td>
<td>0.14</td>
<td>128</td>
<td>6</td>
<td>95.75</td>
<td>0.83</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>88.67</td>
<td>0.52</td>
<td>1</td>
<td>11</td>
<td>90.80</td>
<td>1.27</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>87.92</td>
<td>0.41</td>
<td>2</td>
<td>11</td>
<td>90.17</td>
<td>1.36</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>92.02</td>
<td>0.42</td>
<td>4</td>
<td>11</td>
<td>93.32</td>
<td>1.09</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>93.56</td>
<td>0.45</td>
<td>8</td>
<td>11</td>
<td>94.88</td>
<td>1.06</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>94.37</td>
<td>0.17</td>
<td>16</td>
<td>11</td>
<td>95.27</td>
<td>0.86</td>
</tr>
<tr>
<td>32</td>
<td>11</td>
<td>94.75</td>
<td>0.17</td>
<td>32</td>
<td>11</td>
<td>95.47</td>
<td>0.89</td>
</tr>
<tr>
<td>64</td>
<td>11</td>
<td>94.77</td>
<td>0.13</td>
<td>64</td>
<td>11</td>
<td>95.55</td>
<td>0.90</td>
</tr>
<tr>
<td>128</td>
<td>11</td>
<td>94.82</td>
<td>0.12</td>
<td>128</td>
<td>11</td>
<td>95.63</td>
<td>0.86</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>88.90</td>
<td>0.48</td>
<td>1</td>
<td>16</td>
<td>91.14</td>
<td>1.42</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>87.91</td>
<td>0.68</td>
<td>2</td>
<td>16</td>
<td>90.50</td>
<td>1.51</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>91.94</td>
<td>0.37</td>
<td>4</td>
<td>16</td>
<td>93.33</td>
<td>1.14</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>93.57</td>
<td>0.14</td>
<td>8</td>
<td>16</td>
<td>94.50</td>
<td>1.06</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>94.43</td>
<td>0.20</td>
<td>16</td>
<td>16</td>
<td>95.00</td>
<td>0.96</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>94.67</td>
<td>0.17</td>
<td>32</td>
<td>16</td>
<td>95.29</td>
<td>0.95</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>94.83</td>
<td>0.10</td>
<td>64</td>
<td>16</td>
<td>95.43</td>
<td>0.93</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
<td>94.89</td>
<td>0.13</td>
<td>128</td>
<td>16</td>
<td>95.48</td>
<td>0.96</td>
</tr>
</tbody>
</table>
Appendix A: Machine Learning

1. Terminology

- **Instance**: An instance is a data sample. For example, an instance might be a flower sample.

- **Attribute**: An attribute is a characteristic of a sample, for example, a possible attribute is the flower length.

- **Prediction**: Prediction is defined as the predicted output of an algorithm, for example, when data from different flowers have been analyzed, the algorithm should be able to make an accurate prediction for a flower where only a few attributes are known.

- **Classification**: Classification is similar to prediction; it is the final part of the prediction when the algorithm couples a class to the attributes, and for example, when it couples a specific sort of flower with a set of attributes.

2. Machine learning

Machine learning (ML) is a theory concerned with constructing computer systems with the ability to learn by either experience or by studying instructions. This self-improving capability can in a system result in increased efficiency and effectiveness.

ML algorithms have been proven to be useful in a variety of application domains [1]. Usual ML tasks involve recognition, diagnosis, planning, prediction, and classification. This can be used for a wide range of applications such as control systems in cars, face and image recognition or categorization of objects such as documents or books.

2.1. Algorithm classes

Formally the process of learning is about choosing a hypothesis \( h \) of a space of hypotheses \( H \) about the phenomenon at hand. There are several kinds of machine learning algorithms doing this in different ways. Which one is suited for a certain task depends on the problem domain. Learning algorithms are generally organized into different categories based on the desired outcomes of the algorithms. Three common types of learning approaches and examples are described in the following subsections.

2.1.1. Supervised learning

Supervised learning is a method for creating a function from input training data. Training data consists of a number of input objects and correct outputs for these. The problem is then to predict the value of the function for any given input object after seeing only the training examples. To achieve this, the program must generalize the interpretation of situations not yet encountered, based on previously presented data. During the training phase the program learns this by comparing its output with the correct outputs in the training data, and modifies the function to minimize the error.

**Decision trees**: A decision tree is a kind of classifier in which each non-leaf node is labeled with an attribute or a question of some sort, and in which the branches at that node correspond to the possible values of the attribute, or answers to the question. For example, if the attribute was color, then there would be branches below that node for the possible values of color, say red, green, and blue. Leaf nodes are labeled with a class. The classification of an instance starts at the top with the root node. Each question is then answered and the appropriate branch is chosen until we end up in a leaf node. The label on that leaf node is the class for that instance.

2.1.2. Unsupervised learning

The unsupervised learning does not have any correct answers to compare its function against like the supervised learning. Instead the program is given the input patterns and then has to find interesting patterns, regularities, or clustering among them itself. A completely non-assisted agent, however, cannot learn what to do because of lack of information concerning the proper action and the desired state.
**Clustering:** One example of unsupervised learning is clustering. Here the aim is to find groupings in input data. If we for example take a company with data of previous costumers, clustering can be used to find similarities between customers and organize them into groups. For example customers with similar attributes can be grouped together and customer profile distribution can be analyzed to see trends. This can help the company with future customers to see for example what services or products different groups of customers are interested in as well as identifying outliers.

### 2.1.3. Reinforcement learning

Reinforcement learning is when the program takes action by itself in different situations. It explores its environment and makes decisions based on which situation and state it is in. The environment can then either give positive or negative rewards to the program based on what it does. The reinforcement learning algorithm tries to find a policy in order to maximize the rewards during its problem solving life. In reinforcement learning you also have to consider how the environment works.

**Game playing:** Playing a game is a solid example of reinforcement learning. Here, single moves are not particularly significant, but a series of correct moves can make all the difference. In for example chess a small number of correct moves can result in checkmate. Gaming is because of its simplicity to describe but difficulty to master a very interesting field in AI & ML [1].

**Robots:** Another well suited example in the area of reinforcement learning is robotics. For instance if a robot is to reach a goal by navigating in an environment it will through trying and failing several times learn the correct path and identify possible obstacles in its way.

### 2.2. Classification

One common task for learning algorithms is classification. In classification the task is to take each input instance to the algorithm and assign it to a particular class. For example, in a spamfilter for e-mail, the task might involve analyzing e-mails and classifying them as spam or not spam. In an optical character recognition task, the task would involve taking instances representing images of characters, and classifying them according to which character they are. One common application where classifiers are used with great success is document classification [2].

Classifier learning algorithms improve their performance by training by examples. The training process involves that the algorithm is presented with a number of examples that represents something and then the algorithm builds its own set of rules that can describe what it has been presented with. The model that has been built during the training can be seen as containing the patterns inherent in the data set. This data set can be very large and to find these patterns manually would in many cases be impossible or extremely time consuming. When the training phase is complete the model can be used to predict things about new unknown data. For example when data of a new patient is input to a medical system the system could tell whether the patient suffers from a certain disease or not. The result of this classification depends on a number of factors though. The training set is just a subset of all possible information that can be gathered for a situation, and we cannot expect a model that is always correct. Depending on the problem we may come close though. The relation between correctly and incorrectly classified predictions is used as a measure on how good the model is. It is important to know the accuracy of the model before putting it out to real use, since decisions can have a direct impact on people’s lives.

### 2.3. Data mining

Data mining is about finding valuable patterns in existing data [3]. With the fast technical evolution the computer storage capacity is increasing and the exchange and availability of data as well. Large systems such as medical systems or bank systems often use databases containing vast amounts of data. Searching through this data manually would be impossible. With data mining the stored data is searched automatically by computers. This has been used in statistics and by economics a long time, but what is new is the possibilities available to find patterns in this data. Machine learning techniques can be used to thoroughly search through these large data sets to find valuable patterns and relationships.

A lot of different algorithms can be used for data mining. Most ML algorithms provided by the Weka workbench can be utilized for data mining [2]. Usually data mining involves a few basic tasks [4], these are:
Classification, Clustering, Regression and Association rule learning. Classification and clustering have already been discussed in previous chapters, Regression involves finding the mathematical function that best describes the data and Association rule learning or Market basket analysis, as it is sometimes called, is about grouping associated data.

2.4. Common issues

When using ML algorithms there are some common issues that are important to consider, such as the selection and interpretation of input data. Which data is relevant and what happens if the data is incorrect? The problem situation and how much prior knowledge that is available will affect the accuracy of the model. There can be missing values in the set, and even if all values are present, they can be interpreted in different ways. It can be difficult to know which attributes or characteristics are important or how equal things are. For example, how equal is a blue ball to a red ball? It often takes a lot of experimentation before the algorithm gets good. Another common problem is overfitting. Overfitting is when a model seems to fit the training data well, but when used for predicting values for new data it does not perform as well as expected. This can for example occur if a too small training set is given to the algorithm. The algorithm can then have difficulties generalizing it enough to work on new unknown examples. Instead it tends to introduce patterns, which only applies to the current set. This is a fundamental problem in ML and can sometimes be difficult to overcome.

3. References


Appendix B: Random Forest

1. Introduction

Random Forest (RF) is a learning algorithm introduced to the machine learning community by Leo Breiman [1]. It is an ensemble classifier, meaning it consists of several classifiers. As the name implies it is a forest consisting of several decision trees. These trees are built independently of each other in a randomized way to be unique. Just like many other classifiers, a random forest learns by training on existing data to find patterns, which can then be used to make predictions for new data.

RF has shown to give good prediction accuracy on large data sets with missing attributes [1], and it is also claimed that it is not prone to overfitting [1]. But as shown by Segal [2], random forests do not perform as well as expected for all data sets. This is especially true for noisy regression data. By tuning the algorithm parameters carefully, random forest can often be used with good results in such data sets as well, but precise guidelines are lacking. It has also been shown that RF does not perform very well with data containing lots of irrelevant attributes [3]. This can be a problem in applications where lots of data is gathered without interpretation at one stage, and the analysis of it is done at a later stage. Random forest provides its own functions for evaluating attribute importance, to make it easier to remove irrelevant attributes. It also has internal error estimation that shows how well it performs. These techniques can be used together to fine-tune the algorithm to bring accuracy levels up, but it can also be a complicated and lengthy process to find the optimal parameter values. Our experience with random forest shows that it is fast for prediction but the training process can be a much more time consuming process, sometimes taking more than hundred times the prediction time in our tests.

2. The learning algorithm

2.1. Training

Training the random forest means growing several decision trees independently and adding them to the forest. The trees are grown in the following way:

Each tree picks $N$ cases from the training set containing $N$ cases, but every tree picks its own samples with replacement. Sampling with replacement means that when a sample is selected, it is put back to the data set again, and it can therefore be selected by the same tree again. This way each tree gets its own unique training set, sampled from the total set of training data, resulting in a forest with many different trees.

When the training sets have been prepared the trees can be built. First a root node is created and then branches are added to it. Every node is investigated to see if it should be split and new branches and nodes added to it, or if it should be considered as a leaf node. Each tree is grown to the largest extent possible, and there is no pruning. When the forest is completely grown it can be saved and used at a later stage with new data sets.

Every tree starts at the root node. This node will split the data set based on an attribute value and thus separate the instances into different subtrees. To decide which attribute is the best to split upon, $K$ attributes are selected randomly for investigation. The benefit of splitting on a certain attribute is decided by the statistical property called information gain. Information gain represents how good an attribute can separate the classes in the set. The value to use for split point for this attribute can be found by investigating all possible values and choosing the one that results in the highest information gain, or simply by picking a value in the middle between the minimum and maximum value. As long as splitting a node and adding branches to it gives a positive information gain, the process is repeated. If a node is not split it becomes a leaf node, and is given the class attribute that is the most common occurring among the instances that fall under this node.

2.2. Prediction

The prediction process is fairly straightforward and is performed on a completely grown forest. When new data containing instances to be classified are input to the forest, each tree starts predicting every instance in the set. The process of predicting an instance goes as follows:
Start at the root node and check which attribute and value this node splits on. Compare this to the attribute value of the current instance and take the appropriate branch to the next level in the tree. Repeat the process and ultimately we end up in a leaf node. The class attribute assigned to this leaf node is the answer for the current instance in the current tree. This way every tree makes its own decision and casts a vote for a class. The votes are then gathered and the class with the most votes is considered the answer for the classification of that instance.

3. Additional characteristics

3.1. Out of bag (oob) error estimates

The test error rate is important to know because it shows us how good our model is. Usually the accuracy of a model is evaluated using tenfold cross validation. In addition to using tenfold cross validation, RF has its own internal estimate of the error rate, the oob error estimate. This is calculated internally during the training run and, cross validation is not necessary to get an unbiased estimate of the error rate [4]. The oob error estimation is done as follows:

Each tree is constructed using a different bootstrap sample from the original data. From these $N$ samples $N$ cases are sampled with replacement. This results in about one third of the cases being left out of the bootstrap sample and not used in the construction of the tree. When the tree is fully built, these oob samples are put back to the tree to get a classification. This gives us a test set classification for each case in about one third of the trees. At the end of the run, take $j$ to be the class that got most of the votes every time case $n$ was oob. The proportion of times that $j$ is not equal to the true class of $n$ averaged over all cases is the oob error estimate.

3.2. Variable importance

RF can be used for deciding which variables in a data set that are the most important to use for classification. This gives the opportunity to remove attributes that do not contribute enough to the classification task. This is one additional benefit of using a random forest instead of a single decision tree. The importance score for a variable can be found in the following way:

Every tree in the forest performs a classification on its oob instances and the correct votes are counted. Amongst these oob instances the values for variable $k$ are then permuted, and classified again. The number of correctly classified instances in the permuted run is then subtracted from the previous run. The average value of this subtraction among all trees is the raw importance score for variable $k$. If the data set contains a very large number of attributes, the forest can be run once with all variables and then run again using only the most important variables from the first run.

3.3. Missing values

Data sets are not always complete, and for one reason or another some attributes values might be missing. This must be handled in an appropriate way, since we do not want to discard instances that are missing attribute values. Just because a value for an attribute is missing, the other attributes values might still be useful. RF provides two different approaches in handling this:

The first way is the fastest. If the variable is numeric, the missing value is simply replaced by the median value of all instances belonging to class $j$. If the variable is categorical it is replaced by the most frequent non-missing value in class $j$. These replacement values are called fills.

The second way is more complex but has given better classification performance than the first, even with large amounts of missing data. It replaces missing values only in the training set. First a rough and inaccurate filling is done on missing attribute values. Then a forest run is done and proximities are computed. If a value is missing for attribute $k$, it is replaced by the average of the non-missing values of the $k$th attribute weighted by the proximities between the $n$th case and the non-missing value case. When fills have been added, the process is repeated again to find new and better fills.

4. Configuration parameters

There are two main parameters you can adjust when using the random forest algorithm. These parameters can be tuned for optimal classification performance depending on the problem at hand. First, the number of trees can
be set by the user; secondly there is the $K$ value, or number of attributes to consider when splitting nodes. In the original paper on random forests [1], it was shown that the forest error rate depends on two things:

- The correlation between any two trees in the forest. Increasing the correlation increases the forest error rate.
- The strength of each individual tree in the forest. A tree with a low error rate is a strong classifier. Increasing the strength of the individual trees decreases the forest error rate.

Reducing $k$ reduces both the correlation and the strength. Increasing it increases both. Somewhere in between is an optimal range of $k$ - usually quite wide. Using the oob error rate a good value of $k$ can be found. These are the main adjustable parameters to which random forests are sensitive.

5. References


Appendix C: Waikato environment for knowledge analysis (Weka)

1. Introduction

Weka is a Java implemented suite of Machine Learning (ML) algorithms [1], developed at the University of Waikato in New Zealand. It is distributed under the GNU General Public License and is completely free and open source. Weka contains both a command-line and a graphical user interface, and thus is easy to set up and use, in addition to this Weka is an API that can be utilized when developing new algorithms and other fields of application where ML algorithms are used. The suite can be used for many different ML goals, e.g. algorithm benchmarking and development of new algorithms. The Weka tool supports two main graphical interfaces, the explorer and experimenter views.

The explorer view is an easily navigated Weka exploration tool. This component enables the user with various data management/manipulation tools. Abilities like exploring database attributes, convert between different formats, etc. is possible through this tool. Different dataset aspects can be viewed such as missing values and attribute distribution. The user can also run the data through a classifier and see simple results displayed on screen, although the explorer component is not recommended when running batch runs.

With the Weka experimenter component, the user can customize algorithms with different attribute configurations (parameter settings) and datasets. This tool is useful when running experiments, and it allows for more advanced settings with tailored algorithm execution, it enables the user with automation of different algorithm/data setups and several runs (batch runs) can be performed automatically. In addition, the experiment results can be analyzed in the built in experiment analyzer, here the user can tailor how the results are to be displayed with many different options and if needed, also export them to a file. When running tests on a dataset, the user is able to choose between three different experiment types, these are 10-fold cross validation, train/test percentage split with randomized data and train/test percentage split preserved order. Iteration control is also available to the user.

1.1. Random Forest (RF) with Weka

Following is an explanation of thesis-relevant features of the RF algorithm included in the Weka workbench, a full list can be found on the Weka web page [1]. Weka has a complete java implementation of Random Forest (RF) that is available both through the explorer and experimenter graphical interfaces. Through the experimenter interface the user is enabled with a lot of customization, apart from the above mentioned Weka experimenter abilities the user is enabled with the following customizations when building a RF:

- numTrees - Number of trees to grow in the forest.
- numFeatures - Number of features(attributes to consider (k-value).
- seed - Random number seed.
- maxDepth – Maximum depth of the generated trees.
- Debug – Additional info will be posted in the console.

The Random Forest algorithm uses the RandomTree classifier as a building block, this enables further customization of the algorithm execution to the user in form of the minNum variable which specifies the minimum total weight of the instances in a leaf.

1.2. Attribute-Relation File Format (ARFF)

ARFF files are used extensively by Weka, both for experiment input and output. This file format was developed at the same university as WEKA and is stored as an ASCII text file [2]. Several well-known machine learning datasets are distributed together with Weka as ARFF files and if the dataset in question is not yet ported to an ARFF file, the Weka suite allows the user to convert files between ARFF files and other common file formats.
ARFF files consist of two distinct parts, the header and the data section. The header section contains a data set identifier, a list of the attributes and their type. Following this, comes the data section which contains the actual instances (data samples) line by line. The ARFF document type is fairly uncomplicated and can easily be read and edited manually by hand if needed.

1.2.1. Example of ARFF Header section structure

In File Example 1 the header section of an ARFF file is presented, the fairly simple structure is presented with example data. As can be seen in the example the relation parameter that can be seen as a dataset identifier, is proceeded by ‘@relation’, in our case the relation is ‘weather’. Different attributes are identified by ‘@attribute’ and the values they may have are specified between the curly-brackets, the different values are separated by a comma character. Lines preceded with a ‘%’ are treated as comments and will be ignored by the file reader.

% The weather data
% This example was copied from Witten and Frank's data mining book
@relation weather
@attribute outlook { sunny, overcast, rainy }
@attribute temperature numeric
@attribute humidity numeric
@attribute windy { TRUE, FALSE }
@attribute play { yes, no }

File Ex. 1: Header section of an ARFF file.

1.2.2. Example of ARFF Data section structure

The data section of the ARFF file can easily be identified by the preceding ‘@data’ tag. Instances in the data section are separated by a ‘newline’. Each instance is represented on one row and the attribute order is structured after the header section attribute order. In this case the “Class” attribute is the first attribute; in this case it can hold both ‘sunny’, ‘overcast’ and ‘rainy’. All other attributes can, as the class attribute, hold one of the values specified for them in the header section. Missing values are represented by question marks (no missing values present in example). The attribute values (columns) are separated by a comma character.

@sunny 85, 85, FALSE, no
@sunny 80, 90, TRUE, no
@overcast 83, 86, FALSE, yes
@rainy 70, 96, FALSE, yes
@rainy 68, 80, FALSE, yes
@rainy 65, 70, TRUE, no
@overcast 64, 65, TRUE, yes
@sunny 72, 95, FALSE, no
@sunny 69, 70, FALSE, yes
@rainy 75, 80, FALSE, yes
@sunny 75, 70, TRUE, yes
@overcast 72, 90, TRUE, yes
@overcast 81, 75, FALSE, yes
@rainy 71, 91, TRUE, no

File Ex. 2: Data section of an ARFF file.

2. References


Appendix D: Compute Unified Device Architecture (CUDA)

1. Introduction

Developed by NVIDIA, CUDA is a parallel computing architecture that enables developers to utilize NVIDIA’s graphics processing units (GPUs) for general-purpose computing. Although general-purpose computing on graphics processing units (GPGPU) existed before CUDA, NVIDIA revolutionized the area with some sought after improvements. With CUDA, developers can use high level languages such as C and C++ to make use of the highly parallel NVIDIA GPUs for various computing tasks. Other languages such as Python, Java, Fortran, Matlab and Microsoft .NET Framework are available through native method bindings [2][15][16]. Before CUDA was introduced, some commonly used languages to program GPUs were: C for Graphics (Cg), High Level Shading Language (HLSL) and OpenCL Shading Language (GLSL) [12][13][14]. All of these languages offered very limited GPU programmability and required prior graphics Application Programming Interface (API) knowledge. With CUDA, it was made possible to access the processing power of the GPU in an easier way, much like when developing regular C/C++ applications. The first GPUs supporting CUDA were released in 2006 (NVIDIA GeForce 8800 series). These GPUs had up to 128 CUDA cores [7], a number steadily increasing with each new generation of GPUs. As can be seen in studies [1][9][10] GPU performance has been increasing beyond that of CPUs, and early reports on NVIDIA’s upcoming Fermi architecture [11] indicates that this trend continues.

2. NVIDIA GPU Hardware

When CUDA enabled GPUs were introduced it brought several new functions and improvements to GPGPU computing. Some of the main improvements were:

- C support, which removed the need to learn a special graphics programming language.
- Vertex and pixel pipelines were replaced with a single processor that executed all the vertex, geometry, pixel and computing code.
- Scalar thread processor, this removed the need for the developer to manually manage vector registers.
- Single-instruction multiple-thread (SIMT) execution model where multiple threads concurrently execute a single instruction.
- Thread communication and synchronization improvements.

As seen in the hardware overview in Fig. 1, a CUDA GPU consists of a number of Streaming Multiprocessors (SMs). Each SM consists of eight CUDA cores (or Stream Processors), registers and a shared memory. The registers on the SM are shared by all eight cores and are equivalent to CPU registers. To improve synchronization and communication between cores, a shared memory that all cores on the SM can write to and read from was added. Each SM is also equipped with a constant- & texture cache. These two cached memories are readable by all SMs but can only be written to from the CPU (host). Since device memory (global memory) latency can become a serious bottleneck these caches can prove very useful in many cases.
2.1. Compute Capability

With each new generation of GPUs changes to hardware, such as memory and registers, occur. To keep track of these changes in an smooth and easy way NVIDIA has introduced a compute capability that each GPU is assigned. This makes it easier for developers to keep track of different limitations in the hardware, for example the number of active threads per SM is lower with compute capability 1.0 than in 1.2, 768 respectively 1024. A CUDA application can easily be compiled for different hardware by changing the compiler compute capability flag. Currently four different compute capabilities exist: 1.0, 1.1, 1.2 and 1.3. The first number is the major revision number and the second number is the minor revision number, currently no major platform changes have occurred.

A list of relevant features of compute capability 1.2 [2]:

- Registers per multiprocessor is 16384
- 32 threads per warp
- Maximum number of threads per block is 512
- Shared memory per multiprocessor is 16 KB organized into 16 banks
- The amount of constant memory is 64 KB
- The amount of local memory per thread is 16 KB
- The cache working set for constant memory is 8 KB per multiprocessor
- The cache working set for texture memory varies between 6 and 8 KB per multiprocessor
- The maximum number of active blocks per multiprocessor is 8
- The maximum number of active warps per multiprocessor is 32
- The maximum number of active threads per multiprocessor is 1024
- For a one-dimensional texture reference bound to a CUDA array, the maximum width is $2^{13}$
- Support for atomic functions operating in shared memory and atomic functions

3. CUDA Architecture & Development environment

To successfully be able to develop and execute CUDA code, in addition to the CUDA enabled GPU, several software components are required. As can be seen in Fig. 2, the following components are the core of the CUDA architecture:

1. CUDA enabled GPU – Discussed in previous section.
2. NVIDIA CUDA Hardware support in OS kernel – Currently supported: Windows, Linux and Mac OS X.
3. CUDA enabled drivers including device-level API for developers.
4. Part of the CUDA driver, the Parallel thread execution (PTX) defines an instruction set architecture (ISA) and a low level virtual machine.

Fig. 2: CUDA architecture overview showing different approaches for programming NVIDIA GPUs [4].

As Fig. 2 shows, the CUDA software development environment supports two different programming interfaces [4]:

1. Device-level interface: CUDA Kernels are written in separate files using a supported API of choice. Currently the device-level interface supports:
2. **Language integration interface**: C Runtime for CUDA enables the developer to execute standard C functions on the GPU, additional support for Java, Python, Matlab, Fortran and Microsoft .NET is also available through native method bindings. These high level language instructions are generated into PTX instructions which by the PTX virtual machine then are optimized and translated into target GPU instructions.

4. **CUDA Programming**

   With the thesis subject in mind, focus will be on the language integration programming interface in this section since this is the most relevant one of the two. CUDA is built upon the SIMD execution model. In other words, CUDA can compute a lot of instructions in parallel as long as they are all the same type of instruction. Things like multiplication of large matrices, where the same operation is performed on many different data points, are ideal tasks for CUDA. Comparing the raw number of cores between the high end CUDA GPU GTX 480 with 480 CUDA cores [6], and an Intel Core i7 CPU with 4 cores [5], reveals that not all applications can be adopted to perform well on CUDA and only largely parallel problems will harness the full potential of the GPU.

4.1. **Host-Device communication & Kernels**

   As can be seen in Fig. 3, parallel CUDA functions, or ‘Kernels’, are executed from the host. Before the kernel is launched, the developer has to make sure that the required data has been transferred to the device. Data transfers between the host and device is one of the major bottlenecks in GPGPU programming. The transfer rate to and from the GPU is limited by the PCI-E buss speed and this has to be taken into account when developing and designing solutions for execution on the GPU. Even though special techniques such as pinned memory and concurrent bus transfers exist, transfers between CPU and GPU can still be a performance bottleneck for CUDA applications. Therefore, as mentioned in [3], traffic between host and device should be minimized, and for some problems, even though they might execute faster on the CPU, the time it would take to transfer the data back and forth might make the gain in speed obsolete.
Fig. 3: Possible execution flow of a CUDA program containing one kernel [2].

When memory has been allocated and the data has been transferred, the kernel can be launched much the same way as a regular C function. As seen in Code Ex. 1 the main difference in the code is the ‘<<<1, N>>>’ where 1 stands for the number of blocks and N the number of threads per block. Blocks and threads will be discussed in the next section. When the kernel has been invoked the host continues execution and will only wait for kernel completion if a sync threads command is encountered. Because of this feature some CUDA functions can be launched asynchronously, these are:

- Kernel invocations.
- Memory copies between device and host functions that are prefixed with Async.
- Device to device memory copies.
- Set memory functions.

```c
// Kernel definition
__global__ void testKernel(int* variableA, int* variableB) {
  ...
}
int main() {
  ...
  // Kernel invocation
  testKernel<<<1, N>>>(variableA, variableB);
  ...
}
```

**Code Ex. 1:** The kernel ‘testKernel’ is defined and invoked.
4.2. Thread- & Memory hierarchy

As can be seen below in Fig. 4, CUDA threads (called only ‘threads’ for simplicity) are organized into blocks. Blocks can have up to three dimensions, and each thread is identified by a unique ID (tid). In a similar way as for threads, blocks are organized into grids, where a grid has up to three dimensions and each block has a block ID. The maximum number of threads per block and blocks per grid is hardware dependent and differs with different compute capability. Each kernel invoked has one grid assigned to it, and each thread executes an instance of that kernel.

Each thread has a private local memory. This local memory is mainly used for such as function calls and register overflow but can also be used manually by the developer. Each block is assigned a shared memory for intra block communication and passing of data between threads. The global memory is accessible by all threads, blocks and grids. In Fig. 5 an illustration of this memory hierarchy can be observed.
5. Software to Hardware Mapping

With CUDA, largely parallel portions of applications could easily be offloaded from the host to the device, although, to achieve maximum performance the number of threads running should be in the thousands, the latest NVIDIA GPU can run almost 25k threads simultaneously [6]. The developer has to take this into account when determining what code can be executed with CUDA. Running a low number of threads will probably result in a bottleneck [3] for the GPU where cores idle.

The local thread memory and global memory spaces are read-write regions of device memory and are not cached. The shared memory is an on-chip fast memory located on the SM and is preferred over the larger slower global memory. As can be seen in Fig. 1, a CUDA GPU also has two cached memories, constant and textures. These memories can be utilized to reduce latency when fetching data from the device memory. If the shared and cached memory regions are not utilized correctly, and a lot of global or local memory fetches occur, the result might be in CUDA cores, or even SMs, idling [3].

The CUDA thread hierarchy is not just for easier thread management, but it is also directly mapped to the hierarchy of GPU processors:

- Kernels that are executed by a grid of threads run on the device (GPU).
- Each SM runs one or more blocks of threads. The SM executes threads in groups of 32 called warps.
- Each of the eight cores on the SM executes threads one at a time.

The CUDA memory hierarchy is also directly related to the different on-chip memories:

- Global memory is located on the device memory.
- Shared memory is located on each SM.
- Local memory is located in the device memory. When a thread is running, it is assigned a number of registers to speed up execution.
6. References


