GPU-accelerated Learning of Gradient Boosted Multi-label Classification Rules

GPU-beschleunigtes Lernen von Gradient Boosted Multi-label Klassifikationsregeln
Master thesis by Dennis Dröß ler
Date of submission: March 29, 2021

1. Review: Dr. Eneldo Loza Mencía
2. Review: M.Sc. Michael Rapp
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Knowledge Engineering Group

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Dennis Drößler, die vorliegende Masterarbeit gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 29. März 2021

Dennis Drößler

Abstract

Multi-label classification is the task of predicting a set of labels for an entity. For example, an image can contain multiple objects that are all relevant for the image. One way to deal with multi-label classification problems are multi-label rules. In this master thesis, an existing algorithm for learning multi-label classification rules – the BOOMER algorithm – is analysed regarding its potential to utilize the massively parallel execution capabilities of Graphics Processing Units (GPUs). This work presents a parallel CUDA-based implementation of the rule learning process consisting of three massively parallel compute steps compared to the nested loops of the reference CPU implementation. The GPU implementation is evaluated against the reference implementation on synthetic and real-world datasets and achieves an up to 47 times higher performance regarding execution time, while maintaining the same model quality. The potential speedup depends on the characteristic of the dataset, with the number of examples in the dataset being the most important factor. The GPU implementation is best suited for datasets with dense numeric features and a high number of examples, but also performs well on datasets with nominal features and a sufficient amount of examples. Switching to single precision, the GPU implementation reaches speedups of up to 336 compared to the reference implementation with reduced quality on some models.

Keywords: GPU, CUDA, Multi-label classification, Rule learning

Zusammenfassung

Bei der Multi-label-Klassifizierung geht es darum, einen Satz von Labels für eine Entität vorherzusagen. Zum Beispiel kann ein Bild mehrere Objekte enthalten, die alle für das Bild relevant sind. Eine Möglichkeit, mit Multi-label-Klassifikationsproblemen umzugehen, sind Multi-label-Regeln. In dieser Masterarbeit wird ein bestehender Algorithmus zum Lernen von Multi-Label-Klassifikationsregeln – der BOOMER-Algorithmus – auf sein Potential zur Nutzung von massiv-parallelen Berechnungen auf Grafikkarten (GPUs) analysiert. Eine parallele CUDA-basierte Implementierung des Regellernprozesses wird vorgestellt, die aus drei massiv parallelen Rechenschritten im Vergleich zu den geschachtelten Schleifen der CPU-basierten Referenzimplementierung besteht. Die GPU Implementierung wird mit der Referenzimplementierung auf synthetischen und realen Datensätzen evaluiert. Dabei erreicht die GPU-Implementierung eine bis zu 47-fach höhere Leistung hinsichtlich der Ausführungszeit bei gleichbleibender Qualität der gelernten Modelle. Die mögliche Beschleunigung hängt von der Charakteristik des Datensatzes ab, wobei die Anzahl der Beispiele im Datensatz der wichtigste Faktor ist. Die GPU-Implementierung ist am besten für Datensätze mit dichten numerischen Merkmalen und einer hohen Anzahl von Beispielen geeignet, zeigt aber auch bei Datensätzen mit nominalen Merkmalen und einer ausreichenden Anzahl von Beispielen eine gute Leistung. Mit einfacher Genauigkeit bei Gleitkommaberechnungen steigt der Beschleunigungsfaktor der GPU-Implementierung auf bis zu 336, wenn auch mit reduzierter Qualität bei einigen Modellen.

Schlagwörter: GPU, CUDA, Multi-label Klassifikation, Regellernen

Contents

1 Introduction

In the last ten years, Graphics Processing Units (GPUs) have been widely established to train machine learning models. Neural Networks have greatly profited from the processing power of modern GPUs with highly parallel implementations of learning algorithms. Parallel GPU implementations have been proposed also for other classes of machine learning algorithms. One example is the popular gradient boosting library XGBoost, which was extended to utilize GPUs for decision tree training.

In the meantime, the field of interpretable artificial intelligence has gained increased attention. One example for algorithms with interpretable models are rule learning algorithms which infer decision rules from training data. Based on the success of GPU-acceleration for gradient boosted decision trees, rule learning algorithms that are also based on gradient boosting, could benefit in a similar way. The goal of this thesis is to examine an existing rule learning algorithm, the BOOMER algorithm, and to identify the potential to utilize GPUs to accelerate the rule learning process. The BOOMER algorithm was published by Rapp et al. (2021) and is a stagewise algorithm to learn ensembles of gradient boosted multi-label classification rules.

A GPU-based implementation of the BOOMER algorithm using NVIDIA Compute Unified Device Architecture (CUDA) and C++ is developed in this thesis. Using a three step process consisting of a parallel prefix sum, a custom CUDA kernel and a parallel reduction, the GPU implementation is able to evaluate all condition candidates for a rule in parallel. The parallel implementation is compared to the CPU-based reference implementation on synthetic and real-world datasets and achieves a solid speedup on most datasets and a very high speedup on a few large datasets.

Chapter 2 briefly introduces the BOOMER algorithm and provides an introduction to multi-label classification and the GPU programming model as well as some notation used throughout the thesis. Chapter 3 references related work in accelerating machine learning algorithms with GPUs and the achieved speedups. Chapter 4 outlines the relevant segments of the BOOMER algorithm and the approaches to parallelize these segments.

Chapter 5 discusses the options for a parallel GPU implementation and highlights important parts of the parallel implementation. The parallel GPU implementation is evaluated and compared to the original CPU implementation in Chapter 6 on both real-world and synthetic datasets. Chapter 7 provides directions for future work to improve the parallel algorithm implemented in this thesis and possible optimizations for special cases. The thesis is concluded with a brief recap in Chapter 8.

2 Foundations

This chapter first briefly introduces the BOOMER algorithm, which is the subject of analysis and parallelization for this thesis. A short section on multi-label classification follows, as well as some definitions of frequently used terms and notations. The chapter also provides a short introduction to GPU-programming with CUDA C/C++ and closes with a discussion of possible differences between numerical results calculated on a CPU compared to a GPU.

2.1 The BOOMER Algorithm

The BOOMER algorithm was first presented by Rapp et al. in Rapp et al. (2021). It is described as a "stagewise algorithm for learning an ensemble of gradient boosted single- or multi-label rules $F = \{f_1, \ldots, f_T\}$ that minimizes a given loss function in expectation" (Rapp et al. 2021, Chapter 4).

The algorithm iteratively adds rules to an ensemble and recalculates the gradients and Hessians each time a rule has been added successfully (gradient-boosting). The individual rules make a prediction for either a single label or for all labels, depending on the specified hyper-parameter. The algorithm starts by adding the default rule, which covers all examples. All following rules are learned using a greedy top-down approach. At each greedy-iteration, the condition that improves the quality score of the current rule the most is added to the body of the rule and the rule head is adjusted accordingly. New conditions are added until no further improvement of the quality score is possible.

The possible conditions that may be added to a rule body are obtained by averaging adjacent feature values. Nominal attributes are converted to numerical ones using one-hot encoding before the learning procedure starts. When using single-label rules, the label a rule predicts for is determined when searching for the first condition and remains the same when new conditions are added.

To predict for an example x_n, the individual scores provided by all covering rules are summed up into a vector of scores $\boldsymbol{p}_n$. The vector $\boldsymbol{p}_n$ is then transformed into a binary label vector using the sign-function $\boldsymbol{y}_n = (\text{sgn}(\boldsymbol{p}_{n,1}), ..., \text{sgn}(\boldsymbol{p}_{n,K})$ in case of the label-wise logistic loss. This binary label vector indicates which labels are predicted as relevant (1) or irrelevant (0) by the model.

2.2 Multi-label Classification

The rules that are learned by the BOOMER algorithm are multi-label classification rules. In general, a classification rule is of the form $f : b \rightarrow \boldsymbol{p}$ where b is the *body* of the rule and $\boldsymbol{p}$ is the *head*. The body b is a conjunction of conditions $c_1 \wedge c_2 \wedge \ldots \wedge c_m$, where each condition c_i is a comparison $c_i : a_j \leq v$ that compares the value of an attribute a_j of an example to a constant v using a relational operator like $=$, $\neq$, $\leq$, or $\geq$. If all conditions in the body of a rule are satisfied for a given example, that example is *covered* by the rule and the head $\boldsymbol{p}$ is predicted. In single-label classification, p is a scalar numerical score. In multi-label classification, the prediction $\boldsymbol{p} = (p_1, p_2, ..., p_k)$ is a vector that contains a numerical score for each label. If an example is not covered by a rule, the null vector is predicted. For further information on multi-label learning and an overview of relevant metrics and algorithmic approaches, see for example M. Zhang and Zhou (2014), Zheng et al. (2020), and Tsoumakas et al. (2010)

2.3 Terms and Notations

This chapter introduces terms and notations that are frequently used in this work. In formulas, m is used for the number of features or attributes of a dataset. The number of examples is referred to as n, while k is used for the number of labels. A c stands for the number of split points for which conditions are tested.

Feature matrix: The *x-array* is a two-dimensional array of size $m \times n$ and contains the feature values for each example in the dataset.

Sorted_indices: The array *sorted_indices* is also of size $m \times n$ and contains the index permutation of the x-array such that the feature values are sorted in ascending order.

Gradients: A single gradient calculates as the first derivative of the loss function with respect to the model prediction for a certain example and label. The gradient vector of an

example is an n-dimensional vector where n is the number of labels. *Gradients* refers to a two-dimensional array of size $n \times k$ that contains the gradient vector for each example and is updated every time a new rule has been learned.

Hessians: A single Hessian calculates as the second derivative of the loss function with respect to the model prediction for a certain example and label. Considering only decomposable loss functions, the Hessian matrix contains non-zero elements only on the diagonal. Therefore the Hessian of an example can also be represented by an n-dimensional vector where n is the number of labels. Similar to gradients, *Hessians* refers to a two-dimensional array of size $n \times k$ that contains the Hessian vector for each example and is updated together with the gradients every time a new rule has been learned.

Condition: A condition is a comparison of an attribute value of an example and a fixed value. For example, temperature could be an attribute and 21 might be the attribute value of an example for that particular attribute. A condition could be temperature <= 18, which would not be satisfied by the given example. For numeric attributes like temperature, the comparison operators <= and > are used. Nominal attributes (e.g., male, female, diverse) are one-hot encoded so that again the operator <= and > can be used. A condition is stored as a 3-tuple of feature index, whether the condition uses the <= or the > operator and the threshold.

Split Point: A split point between examples is calculated by averaging the feature values of the current example and the previous example (after the examples have been sorted by that feature). A condition based on a split point always covers all examples up to the previous one in case the operator <= is used or all examples from the current one onwards in case the operator > is used. The number of split points depends on the number of different feature values present in each feature of a dataset. A binary feature results in only one split point, namely $(0 + 1)/2 = 0.5$, while a feature with n distinct values results in $n - 1$ split points. For each split point, two condition candidates are evaluated. The first condition candidate that is tested is *attribute* <= *threshold*, then *attribute* > *threshold* is tested.

2.4 GPU-Programming Model with CUDA C/C++

CUDA is an Application Programming Interface (API) to communicate with and instruct CUDA-enabled NVIDIA GPUs as well as a parallel computing platform. CUDA programs can be created using the programming languages C, C++ or Fortran. Language specific extensions and specialized compilers exist for these languages.

CUDA code is executed on the GPU by launching a so-called *kernel*. A kernel is launched as a special kind of function call using a syntax with triple angle brackets between the function name and the function parameters.

Listing 2.1: Exemplary CUDA kernel definition and kernel call in CUDA C++

```
1  __global__ void vecAdd(float* a, float* b, float* c, int n) {
2   tid = threadIdx.x + blockIdx.x * blockDim.x;
3   c[tid] = a[tid] + b[tid];
4  }
5  int main() {
6   ...
7   dim3 blockSize(256);
8   dim3 gridSize(n / blockSize.x);
9   vecAdd_kernel<<<blockSize,gridSize>>>(a,b,c,n);
10  ...
11 }
```

Listing 2.1 shows an example of a simple CUDA kernel with corresponding kernel call. The kernel is defined using the keyword _ _global_ _ before the return type of the function in line 1. The kernel adds two vectors *a* and *b* of size *n* element-wise and returns the result in the vector *c*. Inside the kernel, the built-in variables *threadIdx*, *blockIdx* and *blockDim* are available that uniquely identify each CUDA thread. CUDA threads are grouped into *blocks* of up to a total of 1024 threads per block. The kernel is then executed by a *grid* of multiple thread blocks of the same shape and size. Both blocks and grids can have up to three dimensions. The kernel parameters specify the number of threads with which the kernel will be executed and how they are distributed (e.g., 1024x1x1 or 16x8x4). Usually, the dimension of the thread blocks is fixed and the number of blocks is calculated before the kernel launch depending on the problem size. In this example, the number of threads per block is 256 and the kernel uses only one dimension. The size of the grid is calculated

by dividing the parameter n by the block size. This example assumes that the parameter n is divisible by 256 without remainder. If this is not the case, additional measures have to be taken to ensure correct program behaviour. For further information see NVIDIA Corporation (2021a) or NVIDIA Corporation (2020a).

The GPU used for the experiments in this work, a NVIDIA RTX 2080 Ti, features 4352 CUDA cores (NVIDIA Corporation 2021b) for parallel execution and can have up to 139 264 threads ready for execution (68 Streaming Multiprocessors (SMs) $\times$2048 resident threads / SM)(NVIDIA Corporation 2021a, Appendix I). The RTX 2080 Ti has a theoretical performance of 13.45 TFLOPS for 32 bit floating point calculations. For comparison, the installed CPU, an AMD Ryzen 7 3800X, features 8 cores with Hyper-Threading and has a 32 bit floating point performance of 1.728 TFLOPS.

The Thrust library (NVIDIA Corporation 2020b) is a C++ template library that exposes a C++ standard library compatible API and provides highly parallel implementations for common algorithms. Similar to the *std::vector*, Thrust provides a *thrust::device_vector*. The device vector typically resides in GPU memory and Thrust functions (e.g., *thrust::sort* or *thrust::transform*) automatically execute on the GPU via a CUDA kernel if they are called with a device vector.

The Thrust library also provides so-called *fancy iterators*, that enable the fusion of multiple operations into one function call. For example, the calculation of the euclidian norm of a vector requires three steps:

1. Squaring each element of the vector
2. Computing the sum over all squared values
3. Applying the square root to the sum

Implementing this without fancy iterators in Thrust would require a *thrust::transform* to square the elements, a *thrust::reduce* to compute the sum and then the application of *sqrt()*. Using fancy iterator, one can define a *thrust::transform_iterator* that computes the square of an element as input to the reduction. This fuses the square operation with the reduction and eliminates one memory read and write operation compared to the non-fused version. The full list of functions and iterators can be found in NVIDIA Corporation (2021c).

2.5 Floating Point Accuracy on GPUs

The BOOMER algorithm is a numerical algorithm, therefore the floating point accuracy of the calculation may heavily influence the results of the learning process. The current implementation uses the data type *float64* (IEEE-754 64 bit floating point type) to store the gradients and Hessians as well as the predicted and quality scores. The computations are therefore carried out in double precision.

An implementation of a numerical algorithm for GPUs is likely to produce at least slightly different results when compared to the original implementation. These numerical differences originate from multiple reasons which are described in the following and explained thoroughly in Whitehead and Fit-Florea (2020).

This section introduces the underlying reasons for these differences and why they may appear. Beforehand, a short introduction to floating point numbers in IEEE-754 format is given. A floating point number $x = s \cdot m \cdot 2^e$ consists of a sign bit $s \in \{0, 1\}$, a mantissa $m \in \{0, 1\}^{l_m}$ and an exponent $e \in \{0, 1\}^{l_e}$. The bit length of m and e depends on the representation. Most common are 32 and 64 bits total length which correspond to 23 and 8 bits for mantissa and exponent or 52 and 11 bits respectively.

Due to the fixed number of bits for the mantissa and exponent, floating point numbers only have a limited maximum precision. Adding two floating points numbers may lead to a situation where the exact result cannot be represented in the same floating point format as the summands. A rounding procedure towards the closest number that can be represented is required. Therefore, floating point arithmetic is not associative, as different operation order might lead to different intermediate results that are rounded differently, resulting in slightly different final results.

The example in Table 2.1 demonstrates this behaviour. The binary values for a, b and c are taken from Whitehead and Fit-Florea (2020, Chapter 2.2). All calculations for this example have been executed with Python 3.7.6 using the numpy data type *float32*. This example shows that for computations in 32 bit floating point, $(a + b) + c$ does not always equal $a + (b + c)$. Therefore, associativity does not hold for floating point computations.

Any variation in the sequence of floating point operations may lead to a different result. The parallelization of the BOOMER algorithm with GPUs will lead to significant changes in the execution order of floating point operations. The results of the parallel GPU implementation are therefore likely to differ slightly from the results of the existing CPU implementation.

Table 2.1: Example of the non-associativity of floating point addition when using 32 bit IEEE 754 floating point representation

Variable	Decimal value	Binary value
a	2.00000023841857910156250	01000000000000000000000000000001
b	1.00000011920928955078125	00111111100000000000000000000001
c	8.00000095367431640625000	01000001000000000000000000000001
a+b	3.00000047683715820312500	01000000010000000000000000000010
b+c	9.00000095367431640625000	01000001000100000000000000000001
(a+b)+c	11.00000190734863281250000	01000001001100000000000000000010
a+(b+c)	11.00000095367431640625000	01000001001100000000000000000001

First of all, a different compiler with a very different target architecture (GPU vs. CPU) is used. The CUDA compiler could choose other optimizations and therefore generate a different sequence of floating point operations.

The CPU implementation of the algorithm is currently executed sequentially. A parallel execution usually requires work to be divided among the available computing resources and may use a specialized parallel implementation with a vastly different execution order (e.g., sequential sum of n elements compared to parallel reduction). Additionally, the number of threads used by a parallel algorithm may influence the order of execution. The thread scheduler may also change the order in which the threads are executed arbitrarily and is therefore another possible source of changes in the execution order.

A second source is the presence of the Fused Multiply-Add (FMA) operation and the corresponding specialized execution units on the GPU. The FMA operation combines a multiplication and an addition into one operation and eliminates the intermediate rounding step, resulting in a higher precision compared to the non-fused operations. Instead of $r = rd(a + rd(b \cdot c))$, the FMA execution unit calculates $r = rd(a + b \cdot c)$.

The last source of different floating point results is the availability of x87 extended precision registers and calculations on x86 CPUs. These feature the capability to calculate floating point operations in an extended 80 bit format, keep them in a specialized 80 bit floating point register and only round to the lower precision of 64 or 32 bit when storing the result to memory.

2.6 Parallel Algorithms

The last section in this chapter defines some common parallel algorithms that are used throughout this work and for the parallel implementation of the BOOMER algorithm. These algorithms are implemented within the Thrust library (NVIDIA Corporation 2020b) for parallel execution on GPUs.

Reduction A reduction takes a binary associative operator $\oplus$ and a sequence of n elements $a_0, a_1, \ldots, a_{n-1}$ and returns $a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}$.

Prefix Sum The all-prefix sums or scan operation takes a binary associative operator $\oplus$ and a sequence of n elements $a_0, a_1, \ldots, a_{n-1}$ and returns the sequence $a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}$

Transformation The transform operation takes an unary operator $f(\cdot)$ and a sequence of n elements $a_0, a_1, \ldots, a_{n-1}$ and returns a sequence $f(a_0), f(a_1), \ldots, f(a_{n-1})$

Unique The unique operation takes a sequence of n elements $a_0, a_1, \ldots, a_{n-1}$ and returns the sequence $a_0, \ldots, a_i, \ldots$ of $k \leq n$ elements for which a_i was not equal to a_{i-1} in the original sequence.

3 Related Work

This chapter reviews literature and research related to parallelization techniques in machine learning and GPU-acceleration of classification algorithms. To the best knowledge of the author, there is currently no published research on the topic of accelerating algorithms for the task of learning classification rules using gradient boosting on GPUs. There is however research on GPU-accelerated decision tree learning, random forests and evolutionary rule learning.

Fonseca et al. (2005) group possible parallelization techniques or strategies of Inductive Logic Programming (ILP) algorithms in four categories. The most general strategy is called data parallelism and describes the execution of a part of or a full ILP algorithm on a portion of the training data. This form of parallel execution requires a mechanism to broadcast locally optimal solutions to all other processors to identify globally optimal solutions. A more specific strategy is the parallel exploration of independent hypotheses, which requires each processor to have a local copy of the training data. The amount of parallelism is dependent on the specifics of the learning task. A related strategy explores the search space of a single hypotheses in parallel. The search space can be partitioned and each processor explores one of the partitions. The amount of parallelism exposed by this strategy is dependant on how the search space is divided. The last and most specific parallelization strategy is called parallel coverage test. For each example, a coverage test determines whether the example is covered by the current hypotheses or not. This coverage test can be executed in parallel for all examples. The amount of parallelism therefore depends on the number of examples in the training data and how many are processed by each processor. A parallel implementation is not restricted to utilize only a single one of the four strategies. Each of the more specific strategies can be used in conjunction with the more general ones.

Besides the definition of these parallelization strategies, Fonseca et al. also summarize speedups achieved in other works on shared and distributed memory machines. They found speedups of 4x and 5x on 6 and 8 processors respectively using the parallel coverage

test and linear speedups up to 16 processors for the parallel exploration of the search space. For parallel exploration of independent hypotheses, they report a speedup of 2x on 6 processors, while data parallelism achieved speedups of 5x on 8 processors in one case and linear to super-linear speedup in another.

Harris et al. used GPUs for an accelerated hyper-parameter search in a setting where the optimal hyper-parameters to learn classification rules via beam search are identified using grid search and cross-validation (Harris et al. 2016). They parallelized their search algorithm so that each GPU-thread evaluates one rule candidate on all examples. After all rule candidates have been evaluated on all examples, the results are copied back to the CPU. The CPU then chooses the best rules and prepares new rule candidates for the next iteration. With additional optimizations they achieved a total speedup of 28x compared to a multi-threaded CPU implementation.

On the topic of GPU-accelerated rule-based classifiers, Cano and Krawczyk used GPUs to apply evolutionary algorithms to high-speed data stream mining. They implemented GPU algorithms that learn classification rules from data streams using differential evolution (Cano and Krawczyk 2018) and genetic programming (Cano and Krawczyk 2019) that outperformed other state-of-the-art rule-based classifiers.

Early works on accelerating the learning of decision trees and random forests with GPUs include Grahn et al. (2011) and Van Essen et al. (2012). Grahn et al. implemented a CUDA-based random forest algorithm for GPUs. Using one CUDA thread per tree in the forest for the CUDA kernels, they achieved a speedup of 30x over a multi-core CPU implementation (FastRF) and 50x over a sequential CPU implementation (libRF).

Van Essen et al. implemented a random forest classifier for multi-core CPU, GPU as well as Field Programmable Gate Array (FPGA) and compared their performance. The FPGA clearly outperformed both in total classification performance and performance per Watt. The GPU had by far the best performance per cost, while still outperforming the multi-core CPU by about 2x.

Jansson et al. also proposed a GPU implementation of a random forest algorithm called *gpuRF* (Jansson et al. 2014). They achieved a much better performance on more recent GPUs with a very high number of cores than Grahn et al. They used multiple CUDA threads for each tree instead of just one CUDA thread per tree. Their GPU implementation outperformed both its multi-threaded CPU-based competitors, WekaRF and FastRF.

Mitchell and Frank (2017) implemented a GPU-based decision tree construction algorithm for the gradient boosting library XGBoost. Their implementation is based on the libraries CUB and Thrust and utilizes parallel algorithms like the prefix sum also used in this work.

They achieve speedups of 2x to 6x compared to an existing CPU-algorithm in XGBoost by adaptively switching between two approaches based on the tree depth.

H. Zhang et al. (2017) followed a different approach to implement gradient boosted decision trees for the GPU. Instead of finding the best split point exactly, they approximate the best split using histograms. Their algorithm is implemented in OpenCL in contrast to the Mitchell and Frank (2017) and this work. It outperformed the existing histogram-based CPU implementation in XGBoost by about 7 to 8 and was about 25 times faster than the CPU-based exact-split algorithm in XGBoost, while maintaining similar accuracy.

Wen et al. (2018) used run-length-encoding and dynamic workload allocation to outperform the CPU-based exact-split algorithm in XGBoost by 2 to 3 in terms of price to performance ratio. Their GPU implementation is also about 2 times faster than XGBoost on a 20 core CPU.

An improved version with additional features is presented in (Wen et al. 2019). Their algorithm now generates partial histograms on CUDA-block level and then aggregates the partial histograms to global histograms. These histograms are used to find approximate split points during decision tree training. They achieved a speedup of up to 60x for approximate split finding compared to XGBoost and also outperform existing GPU implementations.

Based on the GPU implementation for XGBoost by Mitchell and Frank (2017) and an experimental support in XGBoost for datasets that are larger than the main memory, Ou (2020) developed a new out-of-core GPU implementation. Previous GPU implementations could not process arbitrarily large datasets due to the limited GPU memory. Using a special sampling technique, the new implementation is able to process datasets with 500 features and 85 million examples on a GPU with 16 GiB memory, without a significant decrease in model accuracy or training performance.

Outside of neural networks, only Skryjomski et al. (2019) have used GPUs for multi-label classification by implementing the multi-label k-nearest-neighbour (MLkNN) algorithm for the GPU. They evaluated their implementation against a single-core CPU implementation on both real-world and synthetic datasets and achieved speedups of up to 216x on the real-world datasets with the same predictive accuracy. With the synthetic datasets, they identified the minimum number of features or examples required for maximum occupancy of the GPU and therefore the optimal speedup. Their implementation produced higher speedups on datasets with nominal features compared to datasets with numeric features.

4 Approach for Parallelization

In this chapter, the BOOMER algorithm is analysed regarding options for parallelization. Before discussing the BOOMER algorithm, the applicability of the parallelization strategies described by Fonseca et al. (2005) is reviewed with respect to the BOOMER algorithm. The main aspects of the algorithm, gradient boosting and top down greedy rule induction, should remain conceptually the same to allow for a comparison with the original implementation.

Fonseca et al. name four parallelization strategies on different levels of an ILP learning algorithm. In the following, the four parallelization strategies as well as their counterpart in the BOOMER algorithm are described. A profiling of the current implementation of the BOOMER algorithm is used to determine the parts of the algorithm where most of the execution time is spent.

The most high-level parallelization strategy involves the parallel execution of an algorithm or major parts of an algorithm on parts of the training data and combining the partial results. The current implementation of the BOOMER algorithm is capable of directly executing the i-th fold of an n-fold cross-validation with the option *--current-fold i*. This enables the execution of all n folds of an n-fold cross-validation in parallel. Executing the BOOMER algorithm on parts of the data and combining the partial results is not possible without major changes as gradient boosting is inherently sequential. This would require a form of parallel boosting (see e.g., Lozano and Rangel 2005) and therefore change the nature of the algorithm.

The next strategy described by Fonseca et al. is the parallel exploration of independent hypotheses. This would correspond to the parallel induction of multiple rules in case of the BOOMER algorithm. Inducing multiple rules in parallel would again violate the inherently sequential boosting nature of the algorithm and is therefore also not considered.

The third strategy is named parallel exploration of the search space of a single hypotheses. This corresponds to finding the best condition to add to a rule in parallel in case of the

BOOMER algorithm. In the current implementation, the three-dimensional search space of size $m \times n \times k$ is traversed sequentially over labels, examples and features. Finding the optimal condition in this search space can be parallelized with varying effort for the three dimensions and is discussed in Sections 4.3.2 and 4.3.3 as well as 4.5.2 and 4.5.3.

The fourth strategy is the parallel coverage test on all examples of a dataset. A coverage test is executed in the BOOMER algorithm each time a refinement has been completed. A parallelization as proposed by Fonseca et al. is possible, but of lower priority than the previous strategy, as the coverage test only takes about 8 % of the total runtime, while the part targeted by the parallel exploration of the search space takes about 90 % of the total execution time (compare Section 4.3).

Section 4.1 defines the configuration in which the BOOMER algorithm is used and analysed. A more detailed description of the BOOMER algorithm in this configuration is provided in Section 4.2. The major parts of the algorithm are explained and their importance based on a runtime profiling is assessed in Section 4.3. In Section 4.4, the asymptotic complexity of the BOOMER algorithm is analysed, while Section 4.5 describes the intended parallelization on an abstract level.

4.1 Focus of this Work

This thesis concentrates on one specific configuration of the BOOMER algorithm. This configuration aims to minimize the logistic loss function for each label independently using single-label rules. As the loss function is label-wise decomposable, the predicted scores and quality scores for each label of an example can be calculated independently. The use of single-label rules restricts the search space for new conditions to a single label after the first condition has been added to the body of a rule. Non-decomposable loss functions are not part of the scope of this work, but are considered during design decisions so that they can be supported in the future.

4.2 Structure of the BOOMER-Algorithm

This section describes the general structure of the BOOMER algorithm as presented in Rapp et al. (2021). Algorithm 4.1 provides an overview of the BOOMER algorithm using a

non-decomposable loss function and single label rules. Optional features like sub-sampling of features, examples or labels, pruning of rules and shrinkage are omitted.

After reading the specified training data from the file system, the algorithm pre-processes the data. One-hot encoding is used to convert nominal features into numerical ones. At the start of the learning process, the algorithm generates a default rule that covers all examples (line 2). Then the process to induce a single rule is repeated until the specified number of rules has been reached.

Every rule starts with an empty body. In each iteration of the refinement loop in line 6, all possible conditions are evaluated for a potential refinement of the rule (lines 12 - 15). The calculations required for lines 14 and 15 are referred to as *find head*. After all conditions have been tested for a refinement, the best refinement is used as the starting point for a new refinement iteration (line 7). When no further refinement is possible, the rule is added to the ensemble (line 21). During the refinement of a rule, the sums of gradients and Hessians ($\boldsymbol{g}$ and H in Algorithm 4.1) are updated each time an example has been processed (lines 18 and 19), regardless of whether a condition was tested for a refinement or not. The sums of gradients and Hessians calculate as the sum of the gradient vectors and Hessian matrices of the processed examples. They are required to find the best head that corresponds to a potential refinement and to assess the quality of the refinement.

After a rule has been added to the ensemble, the gradients and Hessians of the examples that are covered by the new rule are updated with respect to the scores that are predicted by the new rule. After the specified number of rules has been reached, the final ensemble is returned and evaluated on the test dataset.

Algorithm 4.1 : Pseudo code for the BOOMER algorithm when configured to use a decomposable loss function and single-label rules. Adapted from Algorithms 1, 2 and 3 from Rapp et al. (2021).

Data : Training examples $\{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}_{n=1}^{N}$, first and second derivative ℓ' and ℓ'' of the loss function, number of rules T, ℓ_2 regularization weight λ

Result : Ensemble of rules F

$\mathcal{G} = \{\boldsymbol{g}_n\}_{n=1}^{N}$, $\mathcal{H} = \{H_n\}_{n=1}^{N} \leftarrow$ calculate gradient and Hessians w.r.t. ℓ' and ℓ''

$f_1 : b_1 \rightarrow \boldsymbol{p}^{(1)}$ with $b_1(\boldsymbol{x}) = 1, \forall x$ and $\boldsymbol{p}_i^{(1)} = -\frac{g_i}{h_{ii}+\lambda}$ with $\boldsymbol{g} = \sum_{n=1}^{N} b(\boldsymbol{x}_n)\boldsymbol{g}_n$ and $H = \sum_{n=1}^{N} b(\boldsymbol{x}_n)H_n$

$\mathcal{G}, \mathcal{H} \leftarrow$ update gradients and Hessians of examples covered by f_1

for $t = 2$ **to** T **do**

  $f^* : b \rightarrow \boldsymbol{p}$ = new rule

  do

    $f = f^*$

    for $j = 1$ **to** L **do**

      $\boldsymbol{g}, H = \mathbf{0}$

      foreach *example* e *in ascending order of feature values* **do**

        foreach *possible condition* c *on attribute* A_j *and example* e **do**

          $f' : b' \rightarrow \boldsymbol{p}'$ = copy of f

          add condition c to body b'

          $p_i = -\frac{g_i}{h_{ii}+\lambda}\ \forall i$

          $\boldsymbol{p}$ = best single label prediction $p_i \in \boldsymbol{p}$

          if f' *is better than* f^* **then**

            $f^* = f'$

        $\boldsymbol{g} = \boldsymbol{g} + \mathcal{G}_e$

        $H = H + \mathcal{H}_e$

  while $f^* \neq f$

  $f_t = f^*$

  $\mathcal{G}, \mathcal{H} \leftarrow$ update gradients and Hessians of examples covered by f_t

return Ensemble of rules $F = \{f_1, \ldots, f_T\}$

4.3 Computationally Relevant Parts of the BOOMER Algorithm

In this section, insights acquired by profiling the reference CPU implementation of the BOOMER algorithm are discussed. The reference implementation is written in Python and Cython. Python is used for data pre-processing and interfacing, while Cython is used for the computations of the rule induction process. Table 4.1 shows how the overall execution time is distributed across different code sections from the rule induction of the reference implementation. The dataset *corel5k* is an image dataset that consists only of binary features. Therefore, there is at most a single split point for each feature, regardless of the number of examples. This reduces the number of calls to the function *find_head* drastically compared to the total number of iterations of the enclosing loop. As a result, most of the execution time is spent on updating the gradients and Hessians (*update_search*) as well as determining which conditions need to be tested.

Table 4.1: Aggregated profiling results of the reference CPU implementation on dataset *corel5k*, using 10 rules and 1 fold

Code Section	#Executions	Total Time	Percent Total
while loop	127	155.5 s	99.6 %
for loop features	126746	143.2 s	91.7 %
for loop examples	50943908	142.8 s	91.7 %
determine conditions	50943908	82.1 s	52.8 %
find_head <=	49520	0.7 s	0.0 %
find_head >	49520	0.6 s	0.0 %
update best head	329	0.2 s	0.0 %
update_search	50943908	60.6 s	38.9 %
coverage test	118	12.3 s	7.9 %

To assess the importance of *find_head* for datasets with numeric features, the profiling was extended to another dataset. The dataset *scene* consists exclusively of numeric features, meaning that *find_head* and the score calculation are executed in almost every loop iteration. The results are shown in Table 4.2. The overall distribution of the execution time changed significantly. The function *find_head* and the update of the best head are now responsible for over 35 % of the execution time. In exchange, the relative importance of update_search decreased.

The following sections focus on the three parts of the algorithm where a parallelization can improve the overall execution time of the algorithm the most. The first part is the

Table 4.2: Aggregated profiling results of the reference CPU implementation on dataset *scene*, using 10 rules and 1 fold

Code Section	#Executions	Total Time	Percent Total
while loop	91	22.7 s	99.9 %
for loop features	26754	20.8 s	91.7 %
for loop examples	6287190	20.7 s	91.5 %
determine conditions	6287190	10.3 s	45.3 %
find head <=	6101879	2.8 s	12.5 %
find head >	6101879	3.1 s	13.5 %
update best head	2306	2.7 s	11.8 %
update search	6287190	1.9 s	8.4 %
coverage test	82	1.9 s	8.2 %

code that determines the conditions for which the function *find_head* is executed. Then, within the function *find_head*, a score calculation is executed inside a loop. This part is more relevant for datasets with numeric feature, as *find_head* is executed much more often compared to datasets with nominal features. The third part of interest is the update of gradients and Hessians (*update_search*). The fourth relevant code section identified during profiling, the coverage test, could also be parallelized in a similar way as Algahtani and Kazakov (2018) proposed for ILP algorithms. The coverage test is not considered in the following as it is responsible for less than 10 % of the execution time in both profiling scenarios.

4.3.1 Determining Conditions to Test

Before the rule induction starts, the reference implementation creates a view of the feature values called *sorted_indices* where the entries for each feature are sorted in ascending order. In this view, all examples that have the same feature value for an attribute are in a sequence. For each split point between two consecutive examples with different feature values of two consecutive examples, two conditions are tested for a refinement. One uses the comparison operator $\leq$, the other uses $>$. If two examples have the same feature value, no split point exists and no conditions can be tested.

Listing 4.1 shows the relevant code that determines which conditions should be tested to find a refinement of the current rule. The variable x contains the feature values and the variable *sorted_indices* holds the sorted view of x. In line 17, the feature value of

Listing 4.1: Cython code to determine the split points for which *find_head* is executed

```
for c in range(num_features):
 f = get_index(c, feature_indices)

 # Reset the loss function when processing a new feature...
 loss.begin_search(label_indices)
 i = sorted_indices[0, f]
 loss.update_search(i)
 previous_threshold = x[i, f]
 ...
 for r in range(r + 1, num_examples):
  i = sorted_indices[r, f]
  ...
  current_threshold = x[i, f]

  # Split points between examples with the same feature value
  # must not be considered...
  if previous_threshold != current_threshold:
   # Find and evaluate the best head for the current refinement
   current_head = find_head(head, label_indices, loss)
  ...
  previous_threshold = current_threshold
  loss.update_search(i,...)
```

the example from the previous iteration is compared to the feature value of the current example. If these two feature values are different, a split point exists between the previous and the current example and a possible refinement could be found using *find_head*.

This procedure is implemented sequentially, but can be parallelized as follows. Using one thread for each example except the first one, each thread can calculate in parallel its index $i = \text{sorted\_indices}[r, f]$ and the corresponding threshold $x[i, f]$, as well as the index $i' = \text{sorted\_indices}[r - 1, f]$ of its predecessor and the corresponding threshold $x[i', f]$. Then the threads can all compare their two thresholds in parallel and those with a split point can calculate their refinement.

After all threads have finished calculating their refinement, the best refinement has to be determined. In the sequential implementation, this is done by comparing the best result of the current iteration with the best result from the previous iteration and then keeping the better one. This can also be done in parallel with a reduction operation (see for example Blelloch 1990, Chapter 1.2) using the same comparison operator.

A short analysis of the asymptotic complexity of the described part of the algorithm follows. The number of features is denoted by m, the number of examples by n and the number of processors for a parallel implementation by p.

For each feature, the full loop over n examples is executed. Each of these loop iterations consists of one equality test, two executions of *find_head* (one for $\leq$ and one for $>$) and two comparisons of the previous best head with the calculated head. The complexity analysis assumes the worst case, where all examples have different feature values. Dropping constant terms, the asymptotic complexity of the sequential implementation can be determined as $O(m \cdot n)$ equality tests + $O(m \cdot n)$ applications of *find_head* + $O(m \cdot n)$ comparisons. The equality test as well as the comparison are in $O(1)$, while *find_head* is in $O(k)$ (see asymptotic analysis in Section 4.3.2). As *find_head* is the dominant term, the asymptotic complexity can be simplified to $O(n)$ applications of *find_head*.

Considering a potential parallel implementation as mentioned above, two asymptotic measures can be calculated. One is the amount of work that the parallel algorithm performs and the other is the time the parallel algorithm requires. A parallel algorithm is called *work-efficient*, "if it performs the same amount of work, to within a constant factor, as the fastest known sequential algorithm" (Blelloch and Maggs 1996).

For the parallel algorithm, only the work for the comparisons changes. If a parallel reduction as proposed by Blelloch (1990, Chapter 1.2) is used to compute the best refinement, additional $O(log(p))$ comparisons are required. This increases the number of comparisons to $O(n + log(p))$. If the number of processors is assumed as constant, as it is

the case for a single CPU or GPU, the number of comparisons can be simplified to $O(n)$ and the amount of work a parallel implementation must perform is asymptotically in the same order of magnitude. Therefore, such a parallel implementation can be work-efficient.

Considering the asymptotic execution time, the time for the sequential algorithm is the same as its work, $O(n)$ executions of *find_head*. The parallel algorithm can execute *find_head* p times in parallel. This results in an asymptotic execution time of $O(n/p)$ times the execution time for *find_head*.

4.3.2 Score Calculation

The algorithm to calculate the scores required by *find_head* depends on the type of the loss function. For non-decomposable losses, the scores are obtained by solving a system of linear equations. In the case of decomposable losses, the algorithm shown in Listing 4.2 is used to calculate the scores. The calculation of a single score for a combination of feature, example and label is done in lines 14 to 23 and is completely independent from results for other combinations of features, examples and labels. It only depends on the sums of gradients and Hessians as well as the L2 regularization weight. In the case that the condition to be added uses the operator $>$, the condition would cover all examples from the current one onward instead of all examples up to but not including the current one. The sums of gradients and Hessians would therefore need to be calculated over all examples that are currently not included in the sum. This is achieved by subtracting the current sums of gradients and Hessians from the total sums over all examples of the current feature and label in lines 8 to 11.

The calculation is carried out in a loop over the labels. Each iteration depends on the sums of gradients and Hessians, on the total sums of gradients and Hessians (if the condition uses the $>$ operator) and the L2 regularization weight. Therefore the loop can be parallelized by having one thread per iteration execute the full iteration using thread-local storage for intermediate variables. The two result arrays *predicted_scores* and *quality_scores* are then written to in parallel.

The asymptotic complexity can be calculated as $O(k)$ arithmetic operations, where k denotes the number of labels. A parallel implementation does not involve additional work. The amount of work is therefore $O(k)$ for both the sequential and the parallel implementation. Considering the asymptotic execution time, the sequential algorithm requires $O(k)$ time, while the parallel algorithm requires only $O(k/p)$ where p is the number of processors.

Listing 4.2: Cython code for the calculation of predicted scores and quality scores, which are required for *find_head*. The calculation of the scores accounts for most of the execution time of *find_head*

```
# For each label, calculate the score to be predicted,
as well as a quality score...
for c in range(num_labels):
 sum_of_gradients = sums_of_gradients[c]
 sum_of_hessians = sums_of_hessians[c]

 if uncovered:
  l = get_index(c, label_indices)
  sum_of_gradients = total_sums_of_gradients[l] \\
                     - sum_of_gradients
  sum_of_hessians = total_sums_of_hessians[l] - sum_of_hessians

 # Calculate score to be predicted for the current label...
 score = -sum_of_gradients / (sum_of_hessians
                              + l2_regularization_weight)
 predicted_scores[c] = score

 # Calculate the quality score for the current label...
 score_pow = pow(score, 2)
 score = (sum_of_gradients * score)
                     + (0.5 * score_pow * sum_of_hessians)
 quality_scores[c] = score
               + (0.5 * l2_regularization_weight * score_pow)
```

Listing 4.3: Cython code for the update of the sums of gradients and Hessians

```
1 # For each label, add the gradient and hessian of the example
2 # at the given index to the current sum
3 for c in range(num_labels):
4  l = get_index(c, label_indices)
5  sums_of_gradients[c] += gradients[example_index, l]
6  sums_of_hessians[c] += hessians[example_index, l]
```

4.3.3 Update of Gradients and Hessians

The code to update the sums of gradients and Hessians after an example has been processed is shown in Listing 4.3. The values of the gradients and the Hessians for the processed example are added to the current sums of gradients and Hessians for each label individually. This loop over the labels can be parallelized in the same way as the loop in Section 4.3.2 as each iteration is independent from the others.

One difficulty remains as the implementation of the loss function is stateful. The state is reset on calling *reset_search()* before the first example of a feature is processed. In each iteration the state is accumulated with a call to *update_search()* before the next example is processed. The update is independent of whether a refinement was calculated in this iteration or not. The state of the loss function is required during the calculation of the refinement and is held in the two fields *sums_of_gradients* and *sums_of_hessians*. These two arrays accumulate the sum of the gradients and Hessians of each example in the order in which they are processed. As the state is reset before processing the first example of a feature, the state accumulates only while processing examples of the same feature. This allows for a parallelization over the features by having one thread per feature execute the loop over the examples. Each thread then executes the inner loop iterations sequentially, while the outer loop is executed in parallel.

Further parallelization is possible, as the order in which the examples are processed is known upfront. Therefore, the values of the arrays *sums_of_gradients* and *sums_of_hessians* can be computed for all iterations of a feature before the refinements are calculated. This requires additional memory to store the values of the two arrays for each iteration and breaks the data dependency between the iterations. The refinements are no longer dependant on the updates of the sums in the previous iterations. Therefore, all refinements of a feature can be calculated in parallel, in addition to the refinements for different

features. The amount of additional memory per feature is $O(n \cdot k)$ in the worst case, as for each of the up to n examples, for which a refinement has to be calculated, $2k$ entries of the sums of gradients and Hessians must be stored. The worst case here is the case where all examples have unique feature values because then the sums of gradients and Hessians need to be stored for all examples.

The problem of computing the values of *sums_of_gradients* and *sums_of_hessians* for all iterations of a feature is of the following form:

$\boldsymbol{y}_0 = \boldsymbol{x}_0$
$\boldsymbol{y}_1 = \boldsymbol{x}_0 + \boldsymbol{x}_1$
$\boldsymbol{y}_2 = \boldsymbol{x}_0 + \boldsymbol{x}_1 + \boldsymbol{x}_2$
...

The x_i are the individual gradient or Hessian vectors of an example and the y_i are the sums of gradients or Hessians corresponding to iteration i. This problem structure matches the one of the general prefix sum where the x_i and y_i are vectors of length n that are summed up element wise by the prefix sum. Blelloch describes an efficient parallel algorithm for the calculation of prefix sums (Blelloch 1990). The described algorithm produces the prefix sum of a single vector of length n using p processors. It has a time complexity of $O(n/p + log(p))$. Extending this formulation to a second dimension, the individual elements become vectors of length k and the $+$ operator becomes the element-wise vector addition. This can be interpreted as the k-times application of the basic prefix sum on the respective elements of the vectors. The time complexity is then $O(k \cdot (n/p + log(n)))$.

4.4 Asymptotic Complexity

This section summarizes the asymptotic complexity of the individual steps from the previous three sections. These three parts are the dominating parts of the algorithm in terms of execution time. The parts of the implementation that are associated with reading input files and writing output files, as well as statistics and logging are not considered here. There are datasets where the set-up process takes a few minutes, but it is constant per dataset and negligible compared to the total execution time if the number of rules to be learned is large. Another part that is not considered is the sorting of the feature values at the start of the learning process, as the required execution time is also negligible compared to the time required to learn the rules.

The following abbreviations are used: number of features m, number of examples n, number of labels k, number of rules r. For the number of split points, the worst case n is assumed. The number of conditions per rule c_r is dependant on the characteristics of the dataset but usually limited to a certain value to maintain interpretability of the learned rules. It is therefore treated as constant and does not appear in the asymptotic complexity.

Under these assumptions, the asymptotic work and time complexity of the rule induction process of the BOOMER algorithm is:

$$\begin{aligned} & O(r) \text{ rule inductions} \\ = \ & O(r \cdot m \cdot n) \cdot \text{find\_head} \\ = \ & O(r \cdot m \cdot n \cdot k) \end{aligned}$$

The cost for one evaluation of *find_head* is $O(k)$ and the cost for one rule induction is one *find_head* for each combination of feature and example, so $O(m \cdot n) \cdot$ *find_head*.

The parallelization of the main loop as described in Section 4.3.3 requires additional $O(n \cdot k)$ memory per feature. The sequential implementation currently requires $O(m \cdot n)$ memory for the feature matrix *x* and the *sorted_indices*, $O(n \cdot k)$ for the gradients and Hessians and $O(k)$ for the sums of gradients and Hessians during an iteration. In total, the sequential rule induction process requires $O(n \cdot (m + k))$ memory.

The parallel algorithm also requires $O(n \cdot (m + k))$ to store the feature values, sorted indices, gradient and Hessians. Depending on the degree of parallelism, additional $O(n \cdot k)$ to $O(m \cdot n \cdot k)$ memory is required to store the prefix sums of gradients and Hessians. The memory requirement depends on how many times *find_head* shall be executed in parallel. If all refinements for all features, examples and labels of a rule shall be calculated in parallel, $O(m \cdot n \cdot k)$ memory is required as the sums of gradients and Hessians need to be stored for each label, example and feature. If only the refinements of a single feature (or any fixed number of features) shall be calculated in parallel, only $O(n \cdot k)$ memory is required to store the sums of gradients and Hessians for each label and example.

The time complexity of the parallel algorithm is dependant on the available number of processors p. If a sufficiently large number of processors is assumed ($p \geq m \cdot n \cdot k$), all refinements can be calculated in parallel and the prefix sum over gradients and Hessians can also be calculated fully in parallel. The time complexity for one rule would reduce to $O(log(n) + log(p))$. The $log(n)$ term comes from the prefix sum of Section 4.3.3 and the $log(p)$ term from the reduction of Section 4.3.1. Assuming only a constant number of processors $p \ll m \cdot n \cdot k$, the complexity can be given as $O(m \cdot n \cdot k/p)$ (the logarithmic terms are left out as they are dominated by the linear terms).

4.5 Parallel Transformation

The reference implementation of the BOOMER algorithm is sequential and uses only a single thread on a single CPU core. As explained in Section 4.3.3, there are no data dependencies between iterations of different features. A parallelization solely over the features could be implemented. This approach is probably suited for a parallel CPU implementation, but not for a GPU implementation. During the execution of a CUDA kernel, all 32 threads of each *warp* in the kernel execute the same instructions or are *predicated off*. Threads can be *predicated off* when the execution branches have diverged, e.g., after an *if* statement. Only some of the threads execute one branch while the other threads execute another branch. For example, a warp executes an *if* statement and 24 threads execute the *then* branch and 8 threads the *else* branch. This is called *branch-divergence* (NVIDIA Corporation 2020a, Chapter 12). All threads of the warp must execute the same instructions, so the *then* and the *else* branch are both executed after each other. The 8 threads of the *else* branch are predicated off during the execution of the *then* branch and vice versa.

The *if* statements to determine which conditions need to be tested combined with the surrounding loop could produce significant divergence inside a warp. For a GPU implementation, high parallelism and low control flow are well suited. This would be the case if the calculation of all scores for each feature, example and label could be executed in parallel.

The calculations of the scores for *find_head* themselves have no dependencies on other iterations, meaning that they can be executed in parallel if all inputs can be provided. Providing the inputs to all iterations is the main problem for a parallelization over examples and labels. The loss function holds a state via the sums of gradients and Hessians which are updated at the end of each iteration. In order to calculate multiple examples in parallel, the corresponding sums of gradients and Hessians are required for each example. As the order in which the examples are processed is known upfront, the solution used in this work is to compute all sums of gradients and Hessians before computing the scores. This results in a trade-off between exploitable parallelism and required memory to store the intermediate sums of gradients and Hessians.

4.5.1 Determining Conditions to Test

Before any split point can be tested for a refinement, a parallel implementation must first determine all split-points that will eventually be tested for the current refinement. The sequential implementation executes a check at each iteration before executing *find_head* to determine whether the current example provides a split point or not. Depending on the type of the dataset, any number from one example to all examples of a feature can provide a split point. For example, a binary feature yields only a single split point between 0 and 1, as all examples have either 0 or 1 as their feature value.

Algorithm 4.2 : Algorithm to determine all conditions to test in parallel

Data : Feature values $X \in F^{m \times n}$, Permutation $\pi^X \in \{0, \ldots, n\}^{m \times n}$
Result : $\mathbf{r} = (r_1, \ldots, r_m)^\top$, with $r_i \in \{0, n\}^c$

```
1 parallel for f in 0, ..., m − 1 do
2     g ← starting indices of consecutive groups of identical elements in π_f(X[f][:])
3     parallel for i in 0, ...,length(g) do
4         r_f = π_f^X(g[i])
```

Algorithm 4.2 shows pseudocode that, given a matrix of feature values X and a permutation π^X that sorts the rows of X in ascending order, returns for each feature the indices of the examples with a different feature value than their predecessor. In an implementation, the starting indices of consecutive groups of identical elements can be found using functions like *std::unique* from the C++ Standard Library or *numpy.unique* from the NumPy library for Python. The Thrust library also provides a parallel implementation with *thrust::unique*.

Table 4.3: Example for Algorithm 4.2 to determine the conditions to test using $m = 1$ feature and $n = 4$ examples.

X	=	[1,3,2,2]
π^X	=	[0,2,3,1]
$\pi^X(X)$	=	[1,2,2,3]
g	$\leftarrow$	[0,1,3]
r	=	[π^X(0),π^X(1),π^X(3)] = [0,2,1]

The example in Table 4.3 shows which values the variables and sequences would hold for the given X and π^X. For simplicity, only a single feature and four examples are used.

4.5.2 Parallel Score Calculation

The scores required for *find_head* are calculated in a loop over the labels as described in Section 4.3.2. Only the case of a decomposable loss function is considered here. The calculations that are performed at each iteration are independent of the other iterations. The calculations can therefore be parallelized by executing each iteration in a separate thread. Algorithm 4.3 shows the corresponding pseudo code. The individual steps of the computation are replaced by a call to a function *scores*, which takes as arguments the sums of gradients and Hessians for a label as well as the ℓ_2 regularization weight. The function *scores* returns a predicted score and a quality score.

Algorithm 4.3 : Parallel calculation of the predicted and quality scores over labels

Data : $\boldsymbol{g}, H, \Sigma_g, \Sigma_H$ – prefix sums of the gradients and Hessians up to an example and the total sums of gradients and Hessians over all examples, $\lambda \in \mathbb{R}$ – The L2 regularization weight and $d \in \{\leq, >\}$ – The comparison operator of the condition for which the scores are calculated

Result : Predicted and quality scores $p, q \in \mathbb{R}^k$

```
if d == > then
    g = Σ_g − g
    H = Σ_h − H
parallel for i in 0, ..., k − 1 do
    p_i, q_i = scores(g_i, H_ii, l_2)
```

With this formulation of the loss function, it is also possible to parallelize the score calculation over the examples, if the sums of gradients and Hessians are known for each example. This is shown in Algorithm 4.4. The sums of gradients and Hessians are now required for each example and label. The returned scores are two-dimensional and contain one score for each combination of example and label. This can be extended in the same way for the features. This would result in three-dimensional gradient and Hessian inputs as well as score outputs. Depending on the size of the dataset, this extension to the third dimension may exceed the available memory. Therefore, the amount of parallelism on the feature level could be a tunable parameter that depends on the number of execution units and the available memory. The efficient calculation of the required sums of gradients and Hessians is discussed in the following section.

Algorithm 4.4 : Parallel calculation of the predicted and quality scores over examples and labels

Data : $\boldsymbol{g}_n, H_n, \Sigma_g, \Sigma_H$ – prefix sums of the gradients and Hessians over all examples and the total sums of gradients and Hessians over all examples, $l_2 \in \mathbb{R}$ – The l_2 regularization weight and $d \in \{\leq, >\}$ – The comparison operator of the condition for which the scores are calculated

Result : Predicted and quality scores $p, q \in \mathbb{R}^{n \times k}$

```
parallel for j in 1, ..., n − 1 do
    g = g_j
    H = H_j
    if d == > then
        g = Σ_g − g
        H = Σ_h − H
    parallel for i in 0, ..., k − 1 do
        p_i, q_i = scores(g_i, H_ii, l_2)
```

4.5.3 Parallel Update of Gradients and Hessians with Prefix Sums

To parallelize the score calculation over the examples, the respective sums of gradients and Hessians are required. This section provides two approaches to compute the sums of gradients and Hessians for all examples and labels of a single feature. The algorithms in this section are described for a single feature only for readability. As explained in the previous section, the calculations for each feature can be executed in parallel, so the algorithms in this section could be executed inside a parallel loop over the features.

A first approach to calculate the required elements could be to execute the original loop without executing *find_head* and store the intermediate values during each iteration. Considering only the update of gradients and Hessians in each step of the rule induction, the implementation looks like Algorithm 4.5. The notation $G[i][:]$ selects all elements of the i-th row of the matrix G.

All operations that are not associated with the computations of the sums of gradients and Hessians are hidden behind the *compute(…)* function call. During each iteration of the inner *for* loop, the value of *g* and *h* could be stored to a data structure. The outer loop as well as the vector additions in the last two lines could be parallelized. After a value for *g* and *h* has been calculated, the remaining computations based on these values can also

Algorithm 4.5 : Schematic description of the rule induction. All computations that are not relevant for the update of the sums of gradients and Hessians have been summarized as a compute function call.

Data : $G, H \in \mathbb{R}^{n \times k}$ – gradients and Hessians, $l_2 \in \mathbb{R}$ - The l_2 regularization weight
Result : Predicted and quality scores $p, q \in \mathbb{R}^{n \times k}$

```
for f in 0, ..., m − 1 do
    i =sorted_indices[f][0]
    g = G[i][:]
    h = H[i][:]
    for e in 1, ..., n − 1 do
        i =sorted_indices[f][e]
        compute(...)
        g = g + G[i][:]
        h = h + H[i][:]
```

be executed in parallel. Another approach is to parallelize the calculation also over the examples, which is described in the following.

With i^k =sorted_indices[f][k], the variable g holds the following values during the inner loop for a feature f:
Before the first iteration: $\boldsymbol{g} = G[i^0][:]$
Iteration 1: $\boldsymbol{g}^1 = G[i^0][:] + G[i^1][:]$
Iteration 2: $\boldsymbol{g}^2 = G[i^0][:] + G[i^1][:] + G[i^2][:]$
Iteration 3: $\boldsymbol{g}^3 = G[i^0][:] + G[i^1][:] + G[i^2][:] + G[i^3][:]$
...
Iteration n-1: $g^{n-1} = \sum_{j=0}^{n-1} G[i^j][:]$
Iteration n: $g^n = \sum_{j=0}^{n} G[i^j][:]$

If the $G[i^k][:]$ are defined as the input sequence, then the g^k are the output sequence of the all-prefix sums (also called *scan*) operation on the input sequence $G[i^k][:]$. This output sequence g^k at index k contains exactly the values that the gradient vector contains at the end of the k-th iteration. The same can be applied to the Hessians. Applying the scan operation on the input sequence $H[i^k][:]$ returns the output sequence h^k. Using the two sequences g^k and h^k, the loss function can be executed for each example independently. The sequences were specific for feature f, but as the features are independent, sequences g_j^k and h_j^k can be generated for each feature j. The search space for the best condition can

therefore be searched fully in parallel, if sufficient resources (processing cores, memory) are available.

Algorithm 4.6 : Parallel calculation of the prefix sum over gradients and Hessians.

Data : $G, H \in \mathbb{R}^{n \times k}$ – gradient and Hessian, π_X - permutation of the feature values so that they are sorted in ascending order (per feature)

Result : Prefix sums of the gradients and Hessians $\Sigma^G, \Sigma^H \in \mathbb{R}^{m \times n \times k}$

```
1 parallel for i in 0, ..., m − 1 do
2     parallel for j in 0, ..., k − 1 do
3         p^G_{i,j} = π_i(G[:][j])
4         p^H_{i,j} = π_i(H[:][j])
5         Σ^G[i][:][j] = parallel prefix_sum(p^G_{i,j})
6         Σ^H[i][:][j] = parallel prefix_sum(p^H_{i,j})
```

Algorithm 4.6 shows a formalization of the above, where the vector of gradient entries for all examples of a specific label j is selected with $G[:][j]$. This vector is then reordered by the permutation so that the examples are sorted in ascending order by their corresponding feature values. A parallel prefix sum algorithm is applied and the result is stored in the corresponding slice of the three-dimensional result data structure. The computation of the two prefix sums can also be done in parallel.

5 Implementation

After the previous chapter covered the theoretical possibilities how the BOOMER algorithm can be parallelized, this chapter describes the implementation of the parallel GPU version. Table 5.1 provides an overview of the process to calculate the best refinement for a feature. The reference implementation uses a loop over the examples and executes *find_head* and *update* (sums of gradients and Hessians) alternately. After each application of *find_head*, the new head is compared with the current best head and the better one is stored. After all examples have been processed, the best head found during the loop is returned. The intended parallelization as elaborated in Chapter 4 only executes a single *prepare* step, then executes *find_head* in parallel on all examples and applies a reduction to find the best head afterwards. The *prepare* step calculates all results of the original *update* step using a parallel prefix sum.

Table 5.1: Abstract view on the sequential and parallel version of the refinement process for one feature in the BOOMER algorithm

Sequential implementation	Intended parallelization
for each example	prepare()
find_head()	find_heads()
update()	reduce() $\rightarrow$ Optimum
find_head()	
update()	
$\vdots$	
find_head()	
update()	
$\rightarrow$ Optimum	

Depending on the memory constraints, there are three levels of parallelism available to handle multiple features. If all sums of gradients and Hessians fit into GPU memory, the

prepare step can calculate the sums of gradients and Hessians for all features in parallel. The *find_heads* and *reduce* steps can then also be executed for all features.

A less memory intensive level executes the prepare step only for a fixed number of features (at least one) in a loop. Each time the prefix sums for a feature have been calculated, only a subset of the calculated elements is required (not all examples provide a split point). For datasets with a low number of distinct feature values per feature, this can result in a significant reduction in memory usage. By calculating the prefix sums only for a fixed number of features, this reduction can be applied each time a block of features has been processed instead of after all features have been processed. The steps *find_head* and *reduce* can still be executed fully in parallel.

The third level would be to execute *prepare*, *find_heads* and *reduce* in a loop over the features. This way, the prefix sums of gradients and Hessians only need to be stored for a single feature.

Before the actual implementation, Section 5.1 explains why CUDA was chosen for the GPU implementation and which other options to utilize GPUs exist. Section 5.2 provides an overview of the internal data structure of the CUDA implementation. After the data structure has been explained, Section 5.3 discusses the topic of *slicing* to enable the GPU implementation to process larger datasets. Section 5.4 follows the same structure as Sections 4.3 and 4.5 and showcases the implementation of the three major parts of the BOOMER algorithm using CUDA C/C++ and the Thrust library. Each of the three algorithms is followed by a short example to illustrate the parallel implementation. After the implementation, Section 5.5 describes two tunable aspects of the GPU implementation and Section 5.6 describes the limitations of the GPU implementation regarding processable datasets.

5.1 Evaluation of Options for Implementation

There are many options to write code for GPUs. One category are language extensions and libraries like CUDA C/C++ and the Thrust library from NVIDIA as well as their AMD counterparts HIP/ROCm[1] and rocThrust[2]. There are also bindings for other languages, for example PyCUDA (Klöckner et al. 2012b; Klöckner et al. 2012a) for Python, which are also relevant to this work. Finally there are pragma or directive based extensions like

[1] Advanced Micro Devices, Inc 2021a.
[2] Advanced Micro Devices, Inc 2021b.

OpenACC[3] and OpenMP (Dagum and Menon 1998) which both use C/C++ or FORTRAN as base language and can offload parts of code onto a GPU.

The reference implementation of the BOOMER algorithm is written in Python and Cython, so interfacing with other Python or C/C++ code is already possible and straightforward. The compute cluster at the Knowledge Engineering Group provides only NVIDIA GPUs and the NVIDIA CUDA Toolkit. The AMD equivalents of CUDA and Thrust are therefore not considered in this work.

The Thrust library offers parallel (CPU and GPU) implementations of algorithms like the prefix sum. Availability of highly optimized implementations of such algorithms like with the Thrust library is therefore preferred for the implementation. This excludes OpenACC and OpenMP, as they focus mostly on accelerating loops via directives, but do not provide implementations of more complex algorithms.

PyCUDA provides Python wrappers for the CUDA API. It is also possible to write custom CUDA kernels as Python strings, which are then compiled Just-In-Time during execution. Using the Boost[4] C++ library and CodePy[5] there is also the option to use Thrust algorithms with PyCUDA by specifying the function calls as Python strings. But this was not tested during evaluation. The high dependence on other libraries and the fact that CUDA kernel code and Thrust function calls need to be specified as Python strings are considered as disadvantages considering readability and maintenance.

As functions written in C++ can be called directly from within Cython code, CUDA C/C++ with Thrust is chosen for the GPU implementation in this work. The accelerated parts of the code are compiled as a static library and linked together with the Cython code. They only depend on the NVIDIA CUDA Toolkit and the NVIDIA driver.

In case the algorithm should later be used on AMD GPUs, the tool *hipify* which is part of AMD ROCm, can be used to translate the CUDA C/C++ code to HIP C++. The translated HIP C++ code can then be run on both AMD and NVIDIA GPUs.

[3] OpenACC Organization 2021.
[4] `https://www.boost.org/`
[5] `https://documen.tician.de/codepy/`

5.2 Memory Layout

The algorithms in Chapter 4 use vectors and lists of vectors to store the gradients and Hessians as well as all intermediate results. An implementation in CUDA C/C++ should have all data reside in plain one-dimensional arrays for maximal performance. This requires a 2D-interpretation of the memory layout and a corresponding indexing schema. To access the element at the i-th row and j-th column of a two-dimensional matrix of size $m \times n$, the index is calculated as $idx(i, j) = i * n + j$. As C/C++ uses row-major order, the rows of a matrix are stored contiguously. The following arrays are used in the CUDA implementation to store the gradients, Hessians, sums of gradients and Hessians, as well as intermediate results:

d_gradient_matrix Array of size $(n \times k)$ that holds the gradient vector for each example. All label entries for one example are contiguous in memory.

d_hessian_matrix Array of size $(n \times k)$ that holds the diagonal of the Hessian matrix for each example. All label entries for one example are contiguous in memory.

d_x_matrix Array of size $(m \times n)$ that holds the feature values for each example. The features values for one feature are contiguous in memory.

d_sorted_indices Array of size $(m \times n)$ that holds the permutation that sorts each row of *d_x_matrix* in ascending order. Same memory layout as *d_x_matrix*.

d_features Array of size c that holds the index of the feature to which a split point belongs to.

d_examples Array of size c that holds the index of the example to which a split point belongs to.

d_labels Array that holds the indices of the labels. Either of size k in case the condition that is computed is the first condition for the current rule, or of size 1 in case the condition is not the first condition for the current rule (due to the use of single-label rules).

d_sums_of_gradients Array of size $(c \times k)$ that holds the element of the prefix sum of the gradient matrix that corresponds to a split point. The entries for all labels of an example are stored contiguously in memory.

d_sums_of_hessians Array of size $(c \times k)$ that holds the element of the prefix sum of the Hessian matrix that corresponds to a split point. The entries for all labels of an example are stored contiguously in memory.

d_total_sums_of_gradients Array of size k that holds the sum over the gradient vectors for all examples.

d_total_sums_of_hessians Array of size k that holds the sum over the diagonals of the Hessians matrix for all examples.

d_predicted_scores Array of size $(c \times k)$ that holds the predicted scores of the evaluated conditions.

d_quality_scores Array of size $(c \times k)$ that holds the quality scores of the evaluated conditions.

Additional vectors are used to store intermediate results and are mentioned in the respective chapters in which they are used. Some of these vectors are used in more than one part of the algorithm to decrease the memory footprint. This is omitted in the following chapters for clarity.

Due to the use of functions like *thrust::transform* and *thrust::exclusive_scan* from the Thrust library, *thrust::device_vectors* are used as containers for raw GPU memory. A *thrust::device_vector* contains iterators to the beginning and the end of the contiguous memory location as well as the information for the Thrust library that the function shall be executed on the device (which is the GPU in this case).

5.3 Slicing

The maximum amount of GPU memory is typically much less than the possible main memory of a CPU (compare 11 GiB of the NVIDIA RTX 2080 Ti with 128 GiB main memory on the cluster node in Table 6.1). Together with the increased memory footprint of the parallel algorithm to store the prefix sums of gradients and Hessians, additional measures are required to process larger datasets. For example, using the values of the dataset *bookmarks*, which has 4300 features after one-hot encoding, 87856 examples and 208 labels, the prefix sums for gradients and Hessians would require $4300 \cdot 87856 \cdot 208 \approx$ 585 GiB memory each when stored in *double* precision. Such a dataset could therefore not be processed without further measures. A solution to this problem is to split the required data into chunks of a maximal size that fits in GPU memory and process all chunks

sequentially. For example, chunks of 10 features could be processed ($10 \cdot 87856 \cdot 208 \approx$ 1.36 GiB for the prefix sum of the gradients). The search space can theoretically be split on each of the three dimensions (features, examples or labels) or any combination of them. This process is also called *slicing*.

When splitting the search space into chunks, it becomes necessary to compute the optimal results for each chunk and compare these partial results to choose the optimal result of the whole dataset. This adds a small computational overhead, as there are only $O(k)$ comparisons required, where k is the number of chunks the data is split in.

A short discussion is held in the following to decide on which dimensions to slice. Besides the three basic slicing options over features, examples and labels, also all combinations of the three are possible.

Slicing over labels is theoretically possible regarding the configuration considered in this work (label-wise loss function with single-label rules). The computations for each label are independent of the other labels, which would allow for a slicing over labels. But considering possible future extensions with example-wise loss function and multi-label rules, slicing over labels is not applicable. For example-wise losses with multi-label rules, all labels of a single example are required to solve a system of linear equations. Slicing over labels is therefore discarded both as a single measure as well as in combination with other slicing options as it would complicate extending this work.

Slicing over examples would require bookkeeping of the last values of the prefix sums of gradients and Hessians ($O(n)$ memory with n being the number of labels) of the last example of each chunk to start computation of the next chunk based on the stored values. As slicing over examples is also possible considering other loss functions and multi-label rules, it is a candidate for implementation.

The third option is slicing over features which does not require to store additional information like slicing over examples. As the implementation of the prefix sum over gradients and Hessians is sequential in features (the prefix sum is calculated in parallel over examples and labels, but not over features), slicing over the features is very straightforward to implement.

For this work, only a slicing over features is implemented as it suffices for all datasets on which the implementation is evaluated. For future work, an additional slicing over examples could be implemented to process datasets with a higher amount of examples and labels for which the required memory for a single feature exceeds the available GPU memory.

5.4 Parallel Implementation with CUDA C/C++ and Thrust

This section presents an implementation of the parallel algorithms developed in Chapter 4. It follows the same structure as Sections 4.3 and 4.5. Due to the limited amount of GPU memory (11 GiB on the cluster node and 8 GiB on the development PC), this implementation computes the prefix sums over gradients and Hessian only for one feature in parallel. The sums of gradients and Hessians are stored in the corresponding arrays of the data structure (see Section 5.2) for as many features as there is GPU memory available (slicing over features). The score calculation is then executed with a CUDA kernel in parallel for all entries of these arrays. If feature slicing is required, the prefix sum calculation and the kernel execution are executed sequentially for each slice.

5.4.1 Determining Conditions to Test

The Thrust library provides powerful algorithms for stream compaction (removing elements from a data stream). Determining the conditions to test comes down to finding and removing sequences of identical values for each feature. The function *thrust::unique* removes all but the first element of groups of consecutive elements with identical value. Considering only a single feature, applying thrust::unique once on the sorted feature values would remove all duplicate feature values as required. There remain two problems to solve: The BOOMER-algorithm requires the array of example indices and not the array of feature values, so the elements must be removed from the *sorted_indices* array. Another problem concerns the bounds between features. As all example indices for all features are stored contiguously in a single array, applying thrust::unique could identify a group of consecutive elements with identical values that span across two features and therefore belong to two separate groups, even though they are contiguous in memory.

Visualization:
feature values: 0 1 3 5 | 5 6 12 14 | 14 14 15 ...

Above visualization shows two possibly problematic situations. The third situation, where the last element of the previous feature is different from the first element of the current feature, is not shown, as the unique operation does not apply a modification here. The visualization shows the array of feature values, where the symbol "|" denotes the boundary between two features.

The first situation is the one, where the element 5 is present as the last element of the first feature and also as the first element of the second feature. In this case the unique operation removes the second 5 instance, as it is part of a consecutive group and not the first element of that group. Comparing to the original CPU implementation, this is not a wrong behaviour. The CPU implementation only updates the loss function for the first example of each feature and does not execute *find_head* (the gradients and Hessians are zero at this point, because the loss function is updated after an example has been processed). Therefore, this 5 instance has to be removed later regardless and is not an error.

The second situation is the one, where the element 14 is present as the last element of second feature and also as the first and second element of the third feature. The unique operation removes both 14 instances in the third feature. With the same argument as for the first situation, the removal of the first element of the third feature is not an error. The removal of the last 14 instance is also not an error, as it would have been removed regardless of whether the unique operation restarts at a feature boundary or not.

In summary, applying the unique operation on the array where all features are stored contiguously is not a problem, as at most elements around the feature boundaries are removed which would have to be removed later regardless. The other problem is concerned with the removal of elements from another array during the application of the unique operation on an array.

This is solved by using the generalized version, *thrust::unique_by_key*. The generalized function takes an array of keys and an array of values as arguments and searches the key-array for groups of consecutive elements with identical value and then removes all except the first element of each group in the key *and* the value array. The usage of the function *thrust::unique_by_key_copy* is shown in Listing 5.1. The *_copy* version acts similar to the normal *thrust::unique_by_key*, but instead of modifying the source arrays, it *copies* the unique values to a specified array. The array *d_x_values* contains the entries of the feature matrix *x*, reordered according to the *sorted_indices* (sorted in ascending order per feature). The first two arguments to *thrust::unique_by_key_copy* are the beginning and the end of the source key array. This array determines which elements are copied and which are not. The third argument is the source value array, from which the determined elements are also copied. The listing uses a *thrust::counting_iterator* to generate the example indices with unique feature values. The fourth argument is the target key array, where the unique elements are copied to. As the actual feature values are not of interest, the listing uses a *thrust::discard_iterator*. The discard_iterator simply discards all values that are written to it. This reduces the number of memory writes and therefore saves

Listing 5.1: Thrust implementation to determine in parallel which conditions to test

```
thrust::unique_by_key_copy(
  d_x_values.begin(),
  d_x_values.end(),
  thrust::make_counting_iterator(0),
  thrust::make_discard_iterator(),
  d_examples.begin());
thrust::transform(
  d_examples.begin(),
  d_examples.end(),
  d_features.begin(),
  get_row_index(num_examples));
thrust::transform(
  d_examples.begin(),
  d_examples.end(),
  d_examples.begin(),
  get_remainder(num_examples));
```

memory bandwidth compared to the non-copy version of *thrust::unique_by_key*, as the entries of *d_x_values* are not modified. The fifth argument is the target value array, called *d_examples*. The indices of examples with unique feature values are stored in this array after the function has been executed. It is important to note that the indices do not represent the actual examples in their original order, as the array *d_x_values* contained the *sorted* feature values. Furthermore, the examples indices are strictly increasing and do not reset for a new feature. This is corrected in the following steps.

First, the example indices are mapped to their corresponding feature index by dividing by the number of examples. This is done by applying *thrust::transform* with *d_examples* as source and *d_features* as target array. The struct *get_row_index* applies the division by *num_examples* to each element of *d_examples*. After that, each example index can be replaced by the remainder of its value modulo the number of examples. This transformation is applied in-place, with *d_examples* as source and target array.

After this procedure, the array *d_features* contains the information, to which feature the examples in *d_examples* belong. The array *d_examples* contains the indices of the examples with unique feature values when sorted by feature value. The array *d_examples* therefore

contains the information, for which examples the sums of gradients and Hessians are required for to calculate a refinement with *find_head*.

Example

Figure 5.1 shows a short example of the algorithm to determine the conditions to test in parallel (compare Listing 5.1). The inputs are the feature matrix x (1) and the sorted indices (2). Both are displayed as matrices of size $(m \times n)$ in the figure for clarity. The implementation uses linear arrays where the rows are stored consecutively. This example uses the dataset *weather-numerical* as input data. Only the feature with index 3 (corresponds to the attribute *temperature*) is considered for this example. (1) shows the feature values of the 14 examples in the datset. (2) shows the content of the sorted indices for the feature with index 3. Using the sorted indices, the array *d_x_values* (3) receives the feature values in ascending order. Now the function *thrust::unique_by_key_copy* is applied to copy the indices of the unique feature values to the array *d_examples* (4). In this example, only the second occurrence of the feature value *72* is removed. After that, the two transformations from Listing 5.1 are applied to convert the entries of *d_examples* into feature-example index combinations. The entries for *d_features* are calculated by dividing (without remainder) the entries of *d_examples* by the number of examples, which is 14 in this case. Finally, the entries of *d_examples* are transformed by applying the modulo operation, so that the entries are now the indices of the examples of the feature in *d_features* at the same index.

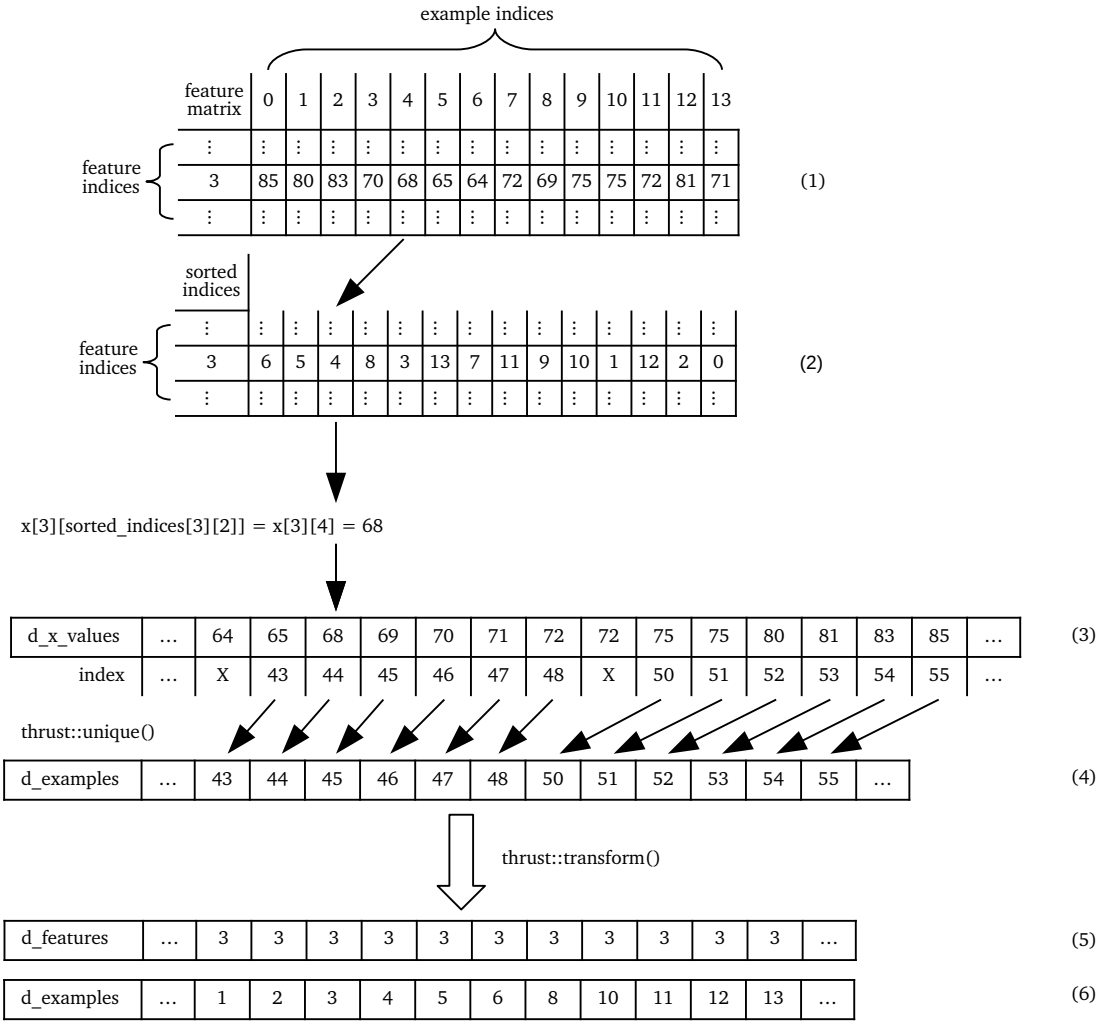


Figure 5.1: Visualization of the values in memory for the algorithm to determine the conditions to test in parallel. The input data of the fourth feature (index 3) of the dataset *weather-numerical* during the induction of the first condition of the first non-default rule are used.

5.4.2 Parallel Score Calculation in CUDA

The custom CUDA kernel to calculate the scores required for *find_head* in CUDA is fairly straight-forward as the data structure has been set up for Single Instruction Multiple Data (SIMD) processing. The examples for which the scores need to be computed are determined with the algorithm described inSection 5.4.1. The data structure uses two arrays, *sums_of_gradients* and *sums_of_hessians* to provide the input data for the score calculation in this kernel. The two arrays have the same layout. For each example, for which the scores need to be calculated, the sum of gradients and Hessians is stored label-consecutive in the respective array. With this layout, the CUDA kernel can access the elements via a 2D indexing scheme. The dimension *x* is used to index the example and the dimension *y* indexes the label of that example. The output arrays *predicted_scores* and *quality_scores* have the same layout. Therefore, the same index can be used to index all four arrays.

Listing 5.2 shows the implemented CUDA kernel. The function parameters are omitted for readability. The keyword `__global__` in line 1 tells the compiler to compile the function for the GPU. Lines 3 and 4 contain the usual index calculation that each CUDA thread executes at the beginning of a kernel. This kernel is a two-dimensional kernel, therefore the index calculation is done for both x- and y-dimensions but not for the z-dimension. CUDA kernels can be one-, two-, or three-dimensional using the built-ins threadIdx, blockIdx and blockDim. Algorithms that operate on higher-dimensional data can either organize the data linearly and use a one-dimensional kernel or calculate all required indices inside the kernel for each thread.

In the case that, e.g, the number of labels is not a multiple of the number of threads in the y-dimension, the index calculation in lines 3 and 4 would produce an index that is not valid or out of bounds. Lines 6 to 8 therefore ensure that a thread that calculated an invalid index immediately returns and cannot produce an illegal memory access or overwrite data of another thread. In line 9, the final index to access the arrays of the data structure is calculated.

The remaining lines from 11 to 27 correspond to the implementation in Cython and calculate the predicted and quality scores. First, the sum of gradients and Hessians up to the current example are read and stored thread-locally (lines 11 and 12). In case the comparison operator ”>” is used in the condition that corresponds to the current example, the sum of gradients and Hessians must be calculated from this example until the end and therefore the current sums are subtracted from the total sums over all gradients and

Listing 5.2: CUDA code of to calculate the score for *find_head*

```
__global__ void calculate_scores_kernel(...) {

  size_t tidx = threadIdx.x + blockIdx.x * blockDim.x;
  size_t tidy = threadIdx.y + blockIdx.y * blockDim.y;

  if (tidx >= num_conditions || tidy >= num_labels) {
    return;
  }
  size_t index = (tidx * num_labels + tidy);

  double sum_of_gradients = sums_of_gradients[index];
  double sum_of_hessians = sums_of_hessians[index];

  if (uncovered) {
    size_t l = label_indices[tidy];
    sum_of_gradients = total_sums_of_gradients[l] -
       sum_of_gradients;
    sum_of_hessians = total_sums_of_hessians[l] -
       sum_of_hessians;
  }

  // Calculate score to be predicted for the current label...
  double score = divide_or_zero(-sum_of_gradients,
     sum_of_hessians + l2_regularization_weight);
  predicted_scores[index] = score;

  // Calculate the quality score for the current label...
  double score_pow = pow(score, 2.0);
  score = (sum_of_gradients * score) + (0.5 * score_pow *
     sum_of_hessians);
  quality_scores[index] = score + (0.5 *
     l2_regularization_weight * score_pow);
}
```

Listing 5.3: Definition of thread and block dimensions and the CUDA kernel call

```
1 size_t dimX = 16;
2 size_t dimY = 8;
3 dim3 threadsPerBlock(dimX,dimY);
4 size_t blockX = num_conditions / threadsPerBlock.x + 1;
5 size_t blockY = num_labels / threadsPerBlock.y + 1;
6 dim3 numBlocks(blockX, blockY);
7 calculate_scores_kernel<<<numBlocks,threadsPerBlock>>>();
```

Hessians (lines 14 to 18). From line 20 until 27, the predicted and quality scores are calculated and stored at their index in the corresponding arrays.

The kernel call as well as the specification of the kernel dimensions are shown in Listing 5.3. Lines 1 to 3 define the dimensions of a block of threads. The type *dim3* is CUDA-specific and is used to specify kernel launch parameters of two or three dimensions. Line 3 specifies that each block shall use 16 threads in x-dimension and 8 threads in y-dimension. In order to launch enough blocks so that all elements are processed, the number of blocks in x- and y-dimension is calculated by dividing the number of conditions and labels by the corresponding block-dimension. The kernel launch itself is specified with triple angle brackets <<< ... >>> in line 7. This is a CUDA-specific syntax only used for kernel launches. The kernel launch parameters are specified between the triple angle brackets.

Example

Figure 5.2 shows an example of the parallel score calculation. This example continues the previous example in Figure 5.1. The arrays *d_features* and *d_examples* were calculated in the previous example. Furthermore, this example assumes that the sums of gradients and Hessians already exist (they are calculated in Figure 5.3). The dataset *weather-numerical* contains 3 labels and the algorithm is configured to use an L2 regularization weight of 1.0. For each pair of entries from the sums of gradients and Hessians, two scores are calculated for two conditions each. The *predicted_scores* p are calculated as $p_i = \frac{-g_i}{h_i+\lambda}$, where g_i is the i-th entry of *sums_of_gradients*, h_i is the i-th entry of *sums_of_hessians* and λ is the L2 regularization weight. The *quality_scores* q are calculated as $q_i = g_i p_i + \frac{1}{2} p^2 (h_i + \lambda)$. These formulas are specific to the configuration using a decomposable loss function and single-label rules. The CUDA kernel computes the predicted and quality scores for all

indices in parallel. Then, the best quality score is selected with *thrust::min_element*, which returns an iterator to the best quality score (index 11 for the values shown in the example). Dereferencing this iterator yields the corresponding quality score. Finally, the arrays *d_features* and *d_examples* are used to identify the feature and example, to which the best quality score belongs to. The sums of gradients and Hessians, as well as the predicted and quality scores, contain one entry for each label of an example that provides a split point. The arrays *d_features* and *d_examples* contain only one entry for each example that provides a split point. The index of the example with the best quality score is therefore calculated by dividing the index for the *quality_scores* (11) by the number of labels (3). This yields index 3 for the arrays *d_features* and *d_examples* and a remainder of 2, which is the index of the label. Therefore the best condition for this selection is the condition with operator $\leq$ on feature 3, example 1 predicting for label 2. The same calculations are also executed for the operator $>$. In this case, the entries of the sums of gradients and Hessians are subtracted from the total sums of gradients and Hessians first.

num_labels	3
L2 regularization weight	1.0

index	…	9	10	11	12	13	14	…
sums_of_gradients	…	-0.361	0.361	-0.702	0.278	-0.278	-0.404	…
sums_of_hessians	…	0.231	0.231	0.209	0.461	0.461	0.418	…

index	…	9	10	11	12	13	14	…
predicted scores ≤	…	0.293	-0.293	0.581	-0.190	0.190	0.285	…
quality scores ≤	…	-0.053	-0.053	-0.204	-0.026	-0.026	-0.058	…

Best quality score for this selection

thrust::min_element returns

index of min element	11
value of min element	-0.204

11 / 3 = 3 with remainder 2

condition “≤” for feature 3, example 1, label 2 is optimal

index	…	3	4	5	6	7	8	9	10	11	12	13	…
d_features	…	3	3	3	3	3	3	3	3	3	3	3	…
d_examples	…	1	2	3	4	5	6	8	10	11	12	13	...

Figure 5.2: Visualization of the values in memory for the algorithm to calculate the predicted and quality scores in parallel. This example is a continuation of the examples in Figures 5.1 and 5.3 and uses the same entries of the feature with index 3 from the dataset *weather-numerical* for the computed arrays *d_features* and *d_examples*. The sums of gradients and Hessians are assumed to have already been calculated (e.g., with Figure 5.3).

5.4.3 Prefix Sums over Gradients and Hessians with Thrust

The calculation of the prefix sums over gradients and Hessians is implemented fully with Thrust routines. The main routine is *thrust::exclusive_scan*, which executes the prefix sum. After the prefix sum has been computed, all elements that are not required for the score calculation can be discarded. This is done by using the function *thrust::gather*, which copies elements specified by a map from a source array to a destination array. The array *d_examples* from Section 5.4.1 is used as the map. The gather operation selects the elements of the prefix sums specified in *d_examples*, which are exactly the sums of gradients and Hessians required for the score calculation.

The implementation of the prefix sum and the selection of the required elements of the prefix sum itself are rather short and concise, as each of them corresponds to a single Thrust function call. In contrast, preparing the input data requires complicated permutation and transpose operations. There exists a special case where the transpose operations are not required. This section starts with this special case to showcase the parallel implementation and to introduce the general idea. The differences for the general case are then shown separately.

The special case is the situation when the prefix sum has to be calculated for one label only. This is the case for single-label datasets and when using single-label rules after the first condition has been added to a rule (see Section 2.1).

Listing 5.4: Prefix sum over gradients and gathering of required elements in the special case where the number of labels is 1

```
1  thrust::exclusive_scan(
2   d_gradient_buffer.begin(),
3   d_gradient_buffer.end(),
4   d_gradient_buffer.begin()
5  );
6
7  thrust::gather(
8   d_examples.begin()+i,
9   d_examples.begin()+i+end,
10  d_gradient_buffer.begin(),
11  d_sums_of_gradients.begin()+i
12 );
```

Listing 5.4 shows the prefix sum over the gradients and the selection of the required sums of gradients with the gather operation. This listing assumes that the array *d_gradient_buffer* contains the entries of a single label of the gradient matrix for all examples, reordered so that the examples are in ascending order of feature value. The arguments to the two function calls are placed on a new line each for better readability. Lines 1–5 show the prefix sum operation as an exclusive scan on the gradient buffer. The scan is executed in-place, meaning that the source and target arrays are the same. After the scan, the gather operation is executed in lines 7–11. The first two arguments specify the beginning and end of the map, which is the part of the array *d_examples* from Section 5.4.1 that corresponds to the current feature. The offset *i* is the offset to the start of the current feature, while *end* is the number of elements to select for this feature. The third argument is the source array, the gradient buffer after the scan operation has been applied. The fourth element is the target array, which contains the sums of gradients for each selected example. It is also indexed with the offset *i*.

The code to prepare the gradient buffer is shown in Listing 5.5. On a high-level view, the code selects all entries of a column of the gradient matrix (all examples for 1 label) and reorders the entries according to *sorted_indices*. These reordered entries are then stored in the gradient buffer for the prefix sum.

On a detailed level, an iterator that contains the indices of the entries of a column of the gradient matrix is created first. This iterator uses a counting iterator beginning at 0. The counting iterator is modified with a transform iterator by multiplying with the number of labels and adding an offset corresponding to the required label. The resulting iterator returns the indices of the gradient entries of the label with number *offset* for all examples in their natural order (lines 14–17). Then, this iterator is transformed further to reorder its elements according to the array *sorted_indices*. The reordering is done with the struct *copy_idx_func* (The code for the struct is adapted from Robert Crovella[6]). The function *operator()* is executed on each element of the iterator, calculates the corresponding row and column from the linear index (lines 8 and 10), computes the new row according to *sorted_indices* (line 9) and returns the corresponding linear index. The offset for *sorted_indices*, *offset2* consists of the index of the current feature multiplied with the number of examples. This indexes the sorted indices corresponding to the example of the current feature. Finally, a permutation iterator is created on the gradient matrix with the previously created iterator (lines 25–28). The permutation iterator returns the gradient entries of the column specified by *offset* reordered according to *sorted_indices*. The content of this permutation iterator is then copied to the gradient buffer (lines 30–34).

[6] `https://stackoverflow.com/a/35121077`

Listing 5.5: Preparation of the gradient buffer for Listing 5.4 for a single label

```
struct copy_idx_func
{
  size_t c;
  size_t *p;
  copy_idx_func(const size_t _c, size_t *_p) : c(_c),p(_p) {};
  __host__ __device__
  size_t operator()(size_t idx){
    size_t myrow = idx/c;
    size_t newrow = p[myrow];
    size_t mycol = idx%c;
    return newrow*c+mycol;
  }
};
auto index = thrust::make_transform_iterator(
  thrust::make_counting_iterator<size_t>(0),
  _1 * num_labels + offset
);
auto permutation = thrust::make_transform_iterator(
  index,
  copy_idx_func(
    num_labels,
    thrust::raw_pointer_cast(&d_sorted_indices[offset2])
  )
);
auto p_gradients = thrust::make_permutation_iterator(
  d_gradient_matrix.begin(),
  permutation
);

thrust::copy_n(
  p_gradients,
  num_examples,
  d_gradient_buffer.begin()
);
```

Listing 5.6: Transposing the gradient buffer with cuBLAS<t>geam()

```
1  double* A = thrust::raw_pointer_cast(d_gradient_buffer.data());
2  double* C = thrust::raw_pointer_cast(d_gradients_T.data());
3  double alpha = 1.0;
4  double beta  = 0.0;
5  size_t m = num_examples;
6  size_t n = num_labels;
7
8  cublasDgeam(
9    cublasHandle,
10   CUBLAS_OP_T, CUBLAS_OP_T,
11   m, n,
12   &alpha,
13   A, n,
14   &beta,
15   A, n,
16   C, m
17 );
```

In the case that the number of labels is not equal to 1, the full rows of the gradient matrix are required. In this case, the iterator described in lines 14–17 is not required any more. But after the row permutation step (lines 18–28), an additional step has to be executed due to the memory layout of the gradient matrix. The gradient matrix is stored label-contiguous, but the prefix sum requires the input data example-contiguous. Therefore, the permuted gradient matrix is transposed before applying the prefix sum.

Listing 5.6 shows the usage of the function *Dgeam* from the *cuBLAS* library to transpose the gradient buffer. The *cuBLAS* library does not support transposing a matrix in-place, so a second buffer called *d_gradients_T* is used to store the transposed matrix. The function *geam()* performs the matrix addition $C = \alpha * op(A) + \beta * op(B)$ where $op(A) = A$ if `CUBLAS_OP_N` is used and $op(A) = A^\top$ if `CUBLAS_OP_T` is used. Specifying $\alpha = 1$ and $\beta = 0$ transposes the matrix A. Further information can be found in the CUDA Toolkit Documentation[7].

[7] https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-geam

Listing 5.7: Prefix sum over the permuted and transposed gradients in the general case with more than one label

```
1  auto start = thrust::make_transform_iterator(
2    thrust::make_counting_iterator(size_t(0)),
3    _1 / num_examples
4  );
5  auto end = thrust::make_transform_iterator(
6    thrust::make_counting_iterator(num_labels * num_examples),
7    _1 / num_examples
8  );
9  thrust::exclusive_scan_by_key(
10   start,
11   end,
12   d_gradients_T.begin(),
13   d_gradients_T.begin()
14 );
```

Now the gradients are in the right order to calculate the prefix sum for each row. To apply the exclusive scan to each row in parallel, the generalized version *thrust::exclusive_scan_by_key* is used. It calculates a *segmented* prefix sum[8], where all elements that belong to the same segment participate in the same prefix sum. The segments are determined by the sequence specified with the first two arguments. For the indices of all elements of this sequence that are equal, the prefix sum is applied to the array specified as the third argument. In other words, every time the value of the key sequence changes, the prefix sum for the value sequence restarts. The result of the segmented prefix sum is written to the fourth argument. Listing 5.7 shows the implementation of the prefix sum in the general case. Instead of the gradient buffer, two iterators named *start* and *end* now determine the number of elements in the prefix sum (lines 10 and 11). The two iterators generate the key sequence that determines the segments of the prefix sum. A counting iterator starting at 0 is divided by the number of examples (lines 2 and 3), which is exactly the length of the rows of the transposed gradient matrix. The iterator *start* therefore contains *num_examples* long sequences of identical values that increase by 1 each sequence. This sequence instructs the segmented prefix sum to compute the prefix

[8] https://thrust.github.io/doc/group__segmentedprefixsums.html

sum over each row of the transposed gradient matrix in parallel. The iterator *end* marks the end of the sequence at *num_labels* $\times$ *num_examples*.

One more change is required for the gather operation. As more than one gradient entry must be extracted for each example, an indexing scheme is necessary to translate example j, label k to the memory location of the (num_labels x num_examples) gradient prefix sum matrix. The code for the general gather operation is shown in Listing 5.8. The iterator *it_start* is created by transforming a counting iterator beginning at 0 with the structure *calc_index* (lines 14–22). The structure is initialized with four values. The example indices for the current feature (line 17, similar to the single label case, except that *i* is now a multiple of *num_labels* and therefore divided to get the same index). The label indices are now also required (line 18). The last two parameters are the number of examples and labels (lines 19 and 20). The *operator()* function of the structure then calculates the required index to the transposed gradient matrix. The iterator *it_end* again marks the end of the sequence, the variable *end* contains the number of gradient entries to gather multiplied with the number of labels. The gather operation copies the required entries to the sums of gradients array with the corresponding offset *i*.

This whole procedure is then also applied to the Hessian matrix. This is a point where a second GPU could be used very naturally without major changes to the code. As the cluster machines only have a single GPU, the prefix sums over gradients and Hessians are calculated one after the other.

In summary, this step takes the gradient and Hessian arrays, calculates the prefix sums over these arrays in a specified order and returns a specified selection or subset of the results. The implementation only processes one feature at a time, as calculating the gradient prefix sums over all features in parallel would increase the GPU memory footprint by $O(m)$ where m is the number of features in the dataset. Furthermore, the complexity of the index calculation for the permutation and gather operation would increase significantly.

Example

One execution of the prefix sum calculation for the feature with index 3 of the dataset *weather-numerical* is shown in Figure 5.3. The array *sorted_indices* from Figure 5.1 is reused, as well as the array *d_examples*. The first step is the permutation of the row of the input matrix (2) according to the sorted indices (1). The input matrix can be either the gradient or the Hessians matrix. This example uses the gradient matrix of the first execution calculated according to the label-wise logistic loss function. The rows of the

Listing 5.8: Generalized gather for more than one label

```
struct calc_index
{
  size_t* e;
  size_t* l;
  size_t m;
  size_t n;
  calc_index(size_t* _e, size_t* _l, size_t _m, size_t _n) :
        e(_e), l(_l), m(_m), n(_n) {};
  __host__ __device__
  size_t operator()(size_t idx){
    return l[idx%n] * m + e[idx/n];
  }
};
auto it_start = thrust::make_transform_iterator(
  thrust::make_counting_iterator(size_t(0)),
    calc_index(
      thrust::raw_pointer_cast(&d_examples[0]+i/num_labels),
      thrust::raw_pointer_cast(&d_labels[0]+i%num_labels),
      num_examples,
      num_labels
    )
);
auto it_end = thrust::make_transform_iterator(
  thrust::make_counting_iterator(end),
    calc_index(
      thrust::raw_pointer_cast(&d_examples[0]+i/num_labels),
      thrust::raw_pointer_cast(&d_labels[0]+i%num_labels),
      num_examples,
      num_labels
    )
);
thrust::gather(
  it_start,
  it_end,
  d_gradients_T.begin(),
  d_sums_of_gradients.begin()+i
);
```

gradient matrix are reordered so that the corresponding example would be in ascending order of feature value for feature 3. Then, the permuted gradient matrix (3) is transposed. The transposed matrix (4) is then scanned (or prefix summed) row-wise (5). An exclusive scan is used to avoid static offsets. The columns of the prefix summed matrix are then gathered via the indices of *d_examples* for the current feature and stored label-consecutive in *d_sums_of_gradients*. The same operations are then also applied to the Hessian matrix. The results are stored in *d_sums_of_hessians*. After that, the scores are calculated as shown in Figure 5.2.

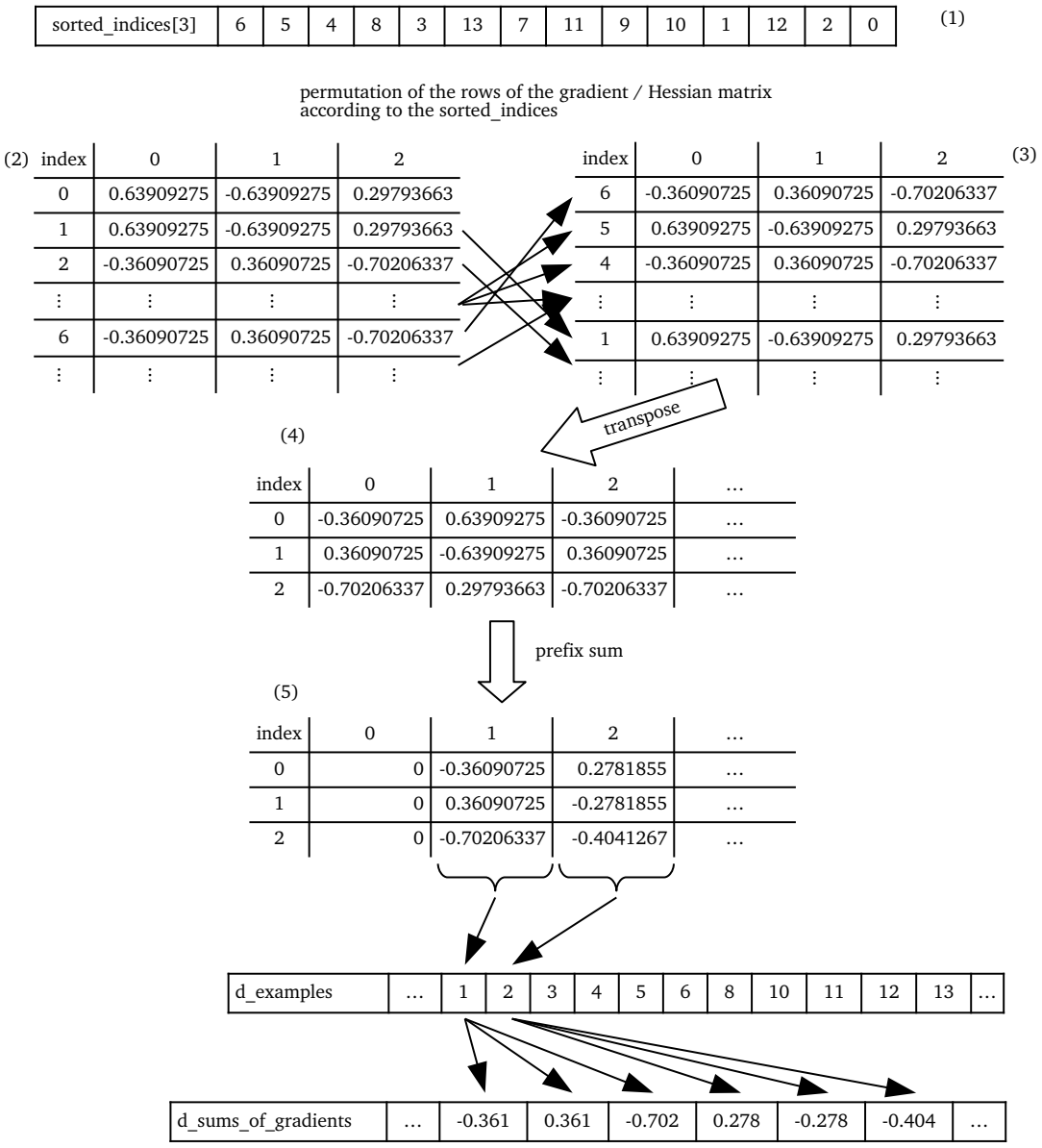


Figure 5.3: Visualization of the values in memory for the algorithm to compute the prefix sum of gradients and Hessians in parallel. This example continues the example in Figure 5.1 and produces the sums of gradients required in Figure 5.2. The arrays *sorted_indices* and *d_examples* are the same as in Figure 5.1.

5.5 CUDA-Parameter and Thrust Tuning

This section revisits two parts of the implementation in CUDA C/C++ and Thrust. The first part in Section 5.5.1 is concerned with a tunable parameter, the CUDA kernel dimensions. The second part in Section 5.5.2 showcases the usage of the NVIDIA Visual Profiler to determine which of two possible implementations in Thrust is faster.

5.5.1 CUDA Kernel Launch Parameters

The CUDA kernel implemented in Section 5.4.2 is configured to use 16 threads in dimension x and 8 threads in dimension y as default launch parameters. Depending on the number of conditions that are tested for a refinement and the number of labels, this default configuration might not be optimal.

For example, a dataset with only 2 labels is expected to perform worse when the kernel is configured with 8 threads in dimension y (label dimension) than when configured with 2 threads in dimension y. Assuming a total of 128 threads per block, the configurations would be 16x8 and 64x2. In case the dataset has more than 64 condition candidates (which is very likely), the first configuration (16x8) would waste $6 \cdot 16 = 96$ threads per block, requiring a total of 4 blocks with 128 threads each to cover all condition candidates. The second configuration (64x2) would require only a single block of 128 threads.

Table 5.2: Number of conditions and labels in selected datasets

Dataset	Conditions	Labels	Ratio
scene	344452	6	57408.67
yeast	153992	14	10999.43
emotions	26865	6	4477.50
birds	47361	19	2492.68
bibtex	3672	159	23.09
corel5k	998	374	2.67

It is therefore expected that the execution time of the kernel is dependant on the configuration of the kernel dimension and that the same kernel configuration performs differently on datasets with varying ratios of the number of conditions to the number of labels. Table 5.2 lists the number of condition candidates that are tested during the calculation of the first condition, the number of labels and the ratio of the two for a selection of the real-world

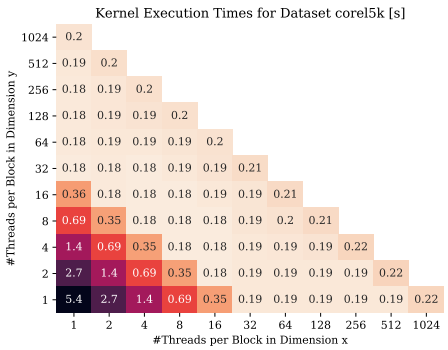
datasets. The datasets are described in Section 6.2.2. The selection contains the datasets with the highest and the lowest ratio as well as some intermediate datasets.

Figure 5.4 shows the execution times for the CUDA kernel for multiple kernel dimensions. The times contain the time for the kernel execution and an immediate *CUDADeviceSynchronize()*. The device synchronize is required as kernel launches are asynchronous in CUDA and return control immediately back to the CPU. A device synchronize blocks the CPU until all kernels have finished. The kernel execution times are logged during an execution of the GPU implementation with a total of 10 rules and a single fold. Then the median over the execution times of the kernel executions that correspond to the first condition of each rule is calculated and displayed in the heatmap. As the maximum number of threads per block is 1024 in CUDA, the heatmaps have a triangular form where the number of threads is constant on the diagonals.

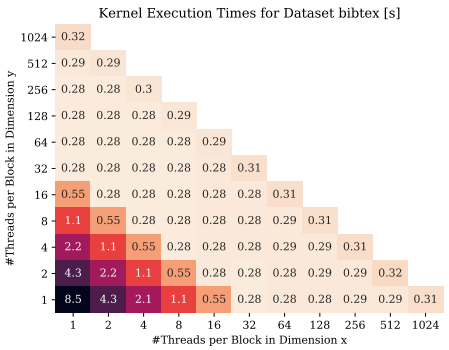
The heatmaps are sorted in ascending order by the ratio of condition candidates to labels. For the first two datasets, *corel5k* (a) and *bibtex* (b), the ratio of threads in dimension x to threads in dimension y does not influence the execution time, as long as the total number of threads per block is between 32 and 512. The low performance for less than 32 threads per block is expected, as the size of a *warp* is always 32. This means that the warps are not utilized properly with less than 32 threads per block and the performance drops. The CUDA C++ Best Practices Guide (NVIDIA Corporation 2020a, Chapter 10.3) recommends at least 64 threads per block and suggests that 128 to 256 is usually a good starting point for the number of threads per block.

The other four datasets (c) to (f) exhibit better performance when the ratio of x/y is greater than 1 compared to when the ratio is less than 1. The best performance is achieved with a total number of threads per block of 32 to 256 and at least 16 threads in dimension x. Dataset *birds* (c) shows a special behaviour in the sense that using only 1 thread in dimension y results in less performance than using 2 or more. This is not the case for the other datasets.

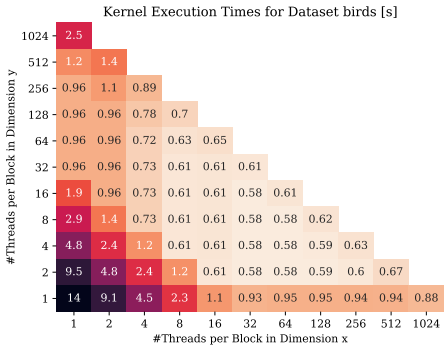
In the case of single-label rules, one special case has to be addressed separately. After the first condition has been added to a rule, the following condition candidates are only evaluated on the single label that yielded the best value for the first condition. This changes the ratio of the number of condition candidates to labels, as the denominator is now equal to 1. Moreover, the number of condition candidates also changes because the number of covered examples strictly decreases as more conditions are added to a rule. The heatmaps for kernel executions with only one label are shown in Figure 5.5. As expected, their characteristic is similar to the ones with a high ratio from Figure 5.4, but with an even stronger focus on the lower middle to lower right part of the triangle.



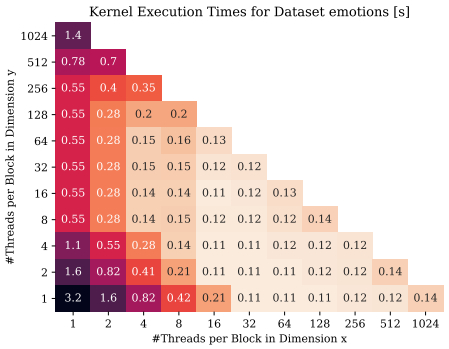
(a) Dataset corel5k, ratio = 2.67



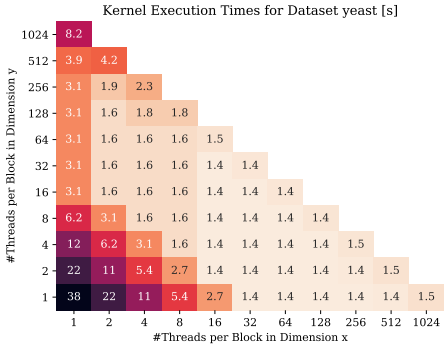
(b) Dataset bibtex, ratio = 23.09



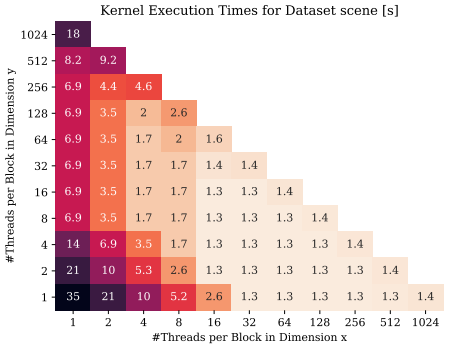
(c) Dataset birds, ratio = 2492.68



(d) Dataset emotions, ratio = 4477.50

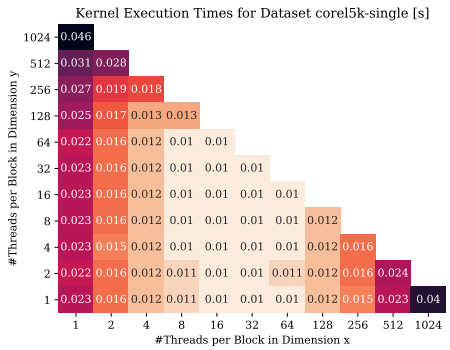


(e) Dataset yeast, ratio = 10999.43

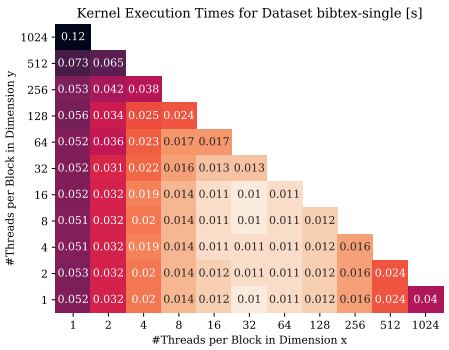


(f) Dataset scene, ratio = 57408.67

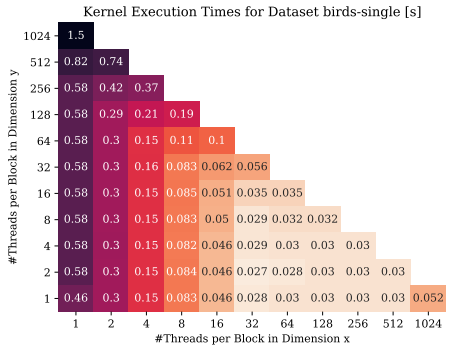
Figure 5.4: Kernel execution times for various kernel launch parameters and selected real-world datasets



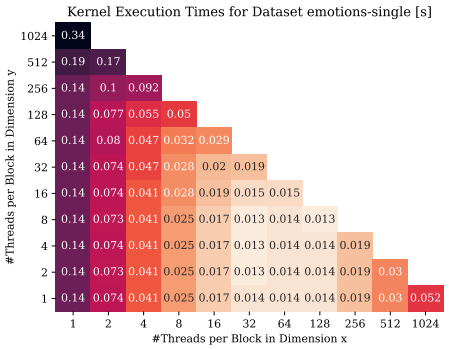
(a) Dataset corel5k



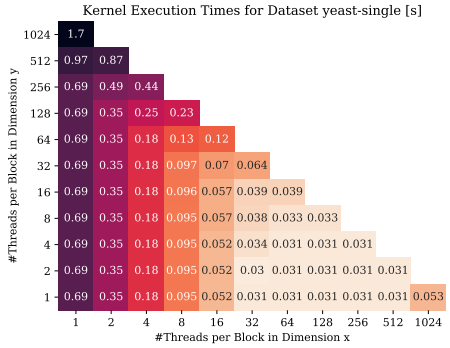
(b) Dataset bibtex



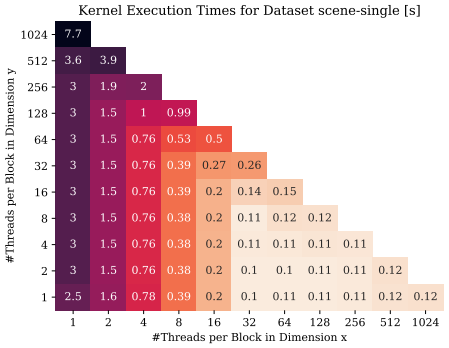
(c) Dataset birds



(d) Dataset emotions



(e) Dataset yeast



(f) Dataset scene

Figure 5.5: Kernel execution times for the same parameters and datasets as Figure 5.4 but for conditions where the number of labels equals 1

The importance of the kernel dimension parameter for the case where only a single label is processed is difficult to asses. The kernel execution time for a single label is usually much smaller than the kernel execution time for all labels of the dataset. The reduction in execution time depends on the number of labels in the dataset and how many examples are covered by the rule after the first condition has been added. The third component is the number of average number of conditions for a rule. If only very few conditions are added to each rule, the kernel execution time for single label executions (conditions after the first one) is probably much less relevant than the kernel execution time for full label executions. On the other hand, if many conditions (e.g., more than 20) are added to each rule, the kernel execution time for the single label case might even be more important than the kernel execution time for the full label case.

Based on the tested configurations, the default kernel parameters were set to 16x8. The execution time for this configuration is near the lowest or the lowest on almost all examined datasets. The parameters for x- and y- dimension of the CUDA kernel can be configured via environment variables. For long running learning task, a short evaluation of various configurations with a total of 64–256 threads using the desired dataset is recommended for optimal performance. The Best Practices Guide recommends to choose a multiple of 32 for the total number of threads per block to fully utilize all allocated warps (NVIDIA Corporation 2020a, Chapter 10.3).

5.5.2 Thrust ZIP-Iterator

The calculation of the prefix-sum over gradients and Hessians consists of the operations *permute*, *transpose*, *exclusive_scan* and *gather*. These operations are applied to both gradients and Hessians.

The *permute* operation is implemented using a *thrust:permutation_iterator* and a *copy* operation. The operations *exclusive scan* and *gather* are implemented with the respective Thrust functions, while the *transpose* operation is implemented using cublasDgeam (compare Sections 5.4.2 and 5.4.3). The Thrust library provides so-called *zip-iterators*, which allow a function to operate on multiple iterators or arrays at once. With such a zip-iterator, it is possible to execute the operations *copy*, *exclusive scan* and *gather* on both gradients and Hessians with one function call for each operation instead of two. This would half the number of required kernel launches and index calculations, but approximately double the number of memory accesses per call. A zip-iterator is created with *thrust::make_zip_iterator* from a tuple of two or more regular iterators, in this case the pointers to the gradient and Hessian arrays. This zip-iterator is then given as an argument to the respective Thrust

Table 5.3: Execution times of the kernels to compute the prefix sums over gradients and Hessians for regular implementation and using zip-iterators. Measured using the NVIDIA Visual Profiler on the dataset *emotions*.

Operation	Execution time separate	Execution time zip
permute	11.0 µs	1.5 µs
scan	38.9 µs	11.0 µs
gather	10.6 µs	1.4 µs

function call. The results are written to another zip-iterator of two pointers to the output arrays.

This section provides profiling results of the Thrust operations mentioned above both with and without zip-iterators executed on the development PC with a NVIDIA GeForce GTX 1070 (see Section 6.1 for full specification). The dataset *emotions* was used as a small dataset and *mediamill* as a large dataset to identify potential differences regarding the size of the dataset. For the execution times with the zip-iterator, the kernel execution time reported by the NVIDIA Visual Profiler is used. For the regular version, the execution time from the start of the first kernel until the end of the second kernel is measured with the Visual Profiler.

The profile timings for the small dataset are listed in Table 5.3. For all three operations, the version with the zip-iterator is 3 to 8 times faster. From the visual representation given by the Visual Profiler, most of the difference in execution time comes from the additional launch overhead for the second kernel launch in the regular case. For small datasets, using the zip-iterator versions for the three Thrust operations is therefore beneficial.

The profiling results for the large dataset are provided in Table 5.4. As the dataset is much larger, the execution times for the three operations are up to 3 orders of magnitude higher. As a result, the kernel launch overhead is negligible compared to the execution time of the kernels. For the permute operation, the zip-iterator version is still faster, but only by a few percent. The exclusive scan is now about 30 % slower using the zip-iterators compared to the regular version with two separate kernel launches. The gather operation is about 11 % slower with zip-iterators.

The permute operation is faster using zip-iterators for both the small and the large dataset. Therefore, the zip-iterator version is used for the permute operation. The scan operation is significantly slower in the zip-iterator version on the large dataset. As the GPU primarily targets larger datasets, the regular version with two separate function calls is used for the

Table 5.4: Execution times of the kernels to compute the prefix sums over gradients and Hessians for regular implementation and using zip-iterators. Measured using the NVIDIA Visual Profiler on the dataset *mediamill*.

Operation	Execution time separate	Execution time zip
permute	751.3 µs	731.8 µs
scan	830.4 µs	1179.9 µs
gather	1027.9 µs	1144.4 µs

scan. For the gather operations, the execution times on the large datasets do not differ significantly, but the zip-iterator version is much faster on the smaller dataset. The gather operation is therefore implemented with a zip-iterator.

5.6 Limitations

This section briefly describes the limits of the GPU implementation described in Section 5.4 regarding processable datasets and available GPU memory. An example for a dataset that is too large for the GPU implementation is the dataset *tmc2007*. It contains 97990 features after one-hot encoding, 28596 examples and 22 labels. This dataset cannot be processed with the GPU implementation on the NVIDIA RTX 2080 Ti installed in the cluster nodes (11 GiB of GPU memory). The algorithm would require over 30 GiB of GPU memory to execute a 5 fold run.

A formula to calculate the required amount of GPU memory in Byte depending on the dataset configuration can be derived from the size of all arrays allocated in GPU memory during execution. The formula for the expected memory requirements depending on the number of features m, the number of example n and the number of labels k is as follows:

$$mem(m, n, k, c) = 8 \cdot ((6 \cdot n \cdot k + m \cdot n + 2 \cdot k) + (m \cdot n + k + 2 \cdot (c + m))) \\ + 56 \cdot c \cdot k + 16 \cdot n + 8 \cdot (c \cdot k + n)$$

The variable c is the number of split points that are tested during execution and can be approximated by the number of different feature values. For the dataset *tmc2007*, using 22876 examples (28596 total, with 4/5 participating in the training set each fold) and assuming c as 97990, the above formula yields $mem(m, n, k, c) \approx 33.15$ GiB.

6 Evaluation

In this chapter, the implemented GPU version of the BOOMER algorithm is compared with the reference CPU implementation. The main focus of this evaluation is on the performance of the two implementations regarding execution time. Section 2.5 provided several reasons why the two implementations are likely to return different models with different classification performances. These differences are reported and interpreted where applicable, but are not the main focus of the evaluation. For the comparison of execution times, two different types of datasets are used. The first type are synthetic datasets that are generated to compare the scaling of the two implementations along the three dimensions features, examples and labels. The second type are real-world datasets from multiple online repositories described in Section 6.2.2.

6.1 Experimental Setup

This section provides a brief overview of the experimental setup and other hardware and software used in this work. A PC system with the specification given in the column *Development PC* in Table 6.1 was used for development, as well as debugging and profiling with the NVIDIA Visual Profiler. The experimental results in this chapter were obtained on cluster nodes of a compute cluster at the Knowledge Engineering Group at Technische Universität Darmstadt. The used cluster nodes are all configured identically, the specification is given in the column *Cluster Node* in Table 6.1.

To estimate the speedup that can be expected from the GPU implementation of the BOOMER algorithm, two properties of a CPU and a GPU can be compared. First, the total number of Floating Point Operations Per Second (FLOPS) is used to compare the computing power that is available to process data. Second, the memory bandwidth between CPU and main memory as well as between GPU and GPU memory defines how fast data can be moved to and from the processor.

Table 6.1: Hardware and software configuration of the development PC and the cluster node

	Development PC	Cluster Node
CPU	Intel(R) Core(TM) i7-6700K CPU @ 4.20GHz	AMD Ryzen 7 3800X 8-Core Processor @ 3.9GHz
GPU	NVIDIA Corporation GP104 [GeForce GTX 1070]	NVIDIA Corporation TU102 [GeForce RTX 2080 Ti]
RAM	16 GiB DDR4-2133	128 GiB
GPU RAM	8 GiB GDDR5X	11 GiB GDDR6
OS	openSUSE Leap 15.1	Debian Buster
C/C++ Compiler	GCC 7.3.0 by Anaconda 4.8.4 (Cython) & GCC 7.5.0 (CUDA)	GCC 8.3.0
NVIDIA Toolkit Version	Driver Version: 455.45.01 CUDA Version: 11.1 Cuda compilation tools, release 11.1, V11.1.105	Driver Version: 418.152.00 CUDA Version: 10.1 Cuda compilation tools, release 9.2, V9.2.148
Python Version	Python 3.7.6	Python 3.7.3

Assuming the highest memory bandwidth supported by the AMD Ryzen 7 3800X (DDR4-3200, PC4-25600), the memory bandwidth per memory channel is 25.6 GB/s. With two memory channels, the peak memory bandwidth is therefore 51.2 GB/s. The GPU, a NVIDIA RTX 2080 Ti is specified with a peak bandwidth of 616 GB/s[1]. From the memory bandwidth perspective, a speedup of up to 12x can be expected, if the algorithm is memory bound.

The number of FLOPS for the CPU is calculated with the formula #cores $\times$ clock speed $\times$ 16 (Hirsch 2020) and was verified by benchmarking. The reference implementation uses only a single core, the double-precision FLOPS for a single core are therefore calculated with the maximum single-core clock speed of 4.5 GHz[2] as 1 core$\cdot$4.5 GHz$\cdot 16 = 72$ GFLOPS. During a short single-core floating point benchmark, 71.232 GFLOPS were measured for the CPU, which supports the calculated number. The GPU is listed with a double precision performance of 420.2 GFLOPS[3], which is about 6 times higher. Considering single precision, the GPU is listed with 13.45 TFLOPS, compared to the doubled 144 GFLOPS of the CPU. The GPU therefore has about 95 times more single precision computing power than the CPU. The reason for this is that the floating point execution units of the CPU can do two single precision operations or one double precision operation. The GPU has 32 times more execution units for single precision than for double precision (4352 compared to 136 for the RTX 2080 Ti) (NVIDIA Corporation 2018). It is therefore interesting to also evaluate the GPU implementation when using single instead of double precision.

[1] `https://www.nvidia.com/en-gb/geforce/graphics-cards/rtx-2080-ti/`
[2] `https://www.amd.com/en/products/cpu/amd-ryzen-7-3800x#product-specs`
[3] `https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305`
(each accessed on 13.03.2021)

Table 6.2: Hyper-Parameter configuration of the BOOMER algorithm used for both synthetic and real-world experiments

Hyper-Parameter	Algorithm Parameter	Setting
Loss function	--loss	macro-logistic-loss
Head refinement	--head-refinement	single-label
Feature sub-sampling	--feature-sub-sampling	none
Instance sub-sampling	--instance-sub-sampling	none
Label sub-sampling	--label-sub-sampling	none
Shrinkage	--shrinkage	0.3
L2 regularization	--l2-regularization-weight	1.0
Cross-validation	--folds	5
Number of rules	--num-rules	100

6.2 Experiments

To compare the GPU implementation with the original CPU-implementation, the experiments described in this section were carried out on the compute cluster specified in Table 6.1. For all experiments, the same algorithm configuration as listed in Table 6.2 was used. The setting *macro-logistic-loss* stands for the label-wise logistic loss function. To ensure that the GPU is initialized and ready for execution, the whole algorithm is executed once on a very small dataset before each test run. After a longer time of inactivity, the first algorithm execution on the GPU can take an extra one or two seconds. To prevent this from influencing the measurements, the initial *warm-up* run is executed before the measurement starts. The following two sections describe the types of experiments with synthetic and real-world datasets.

6.2.1 Synthetic Datasets

The experiments with synthetic datasets are used to measure the scaling of both the CPU and the GPU implementation in the three dimensions features, examples and labels. The models that are learned by the algorithms on these synthetic datasets are part of the scope of this work. Therefore, the synthetic datasets are generated using random values drawn from a standard normal distribution for the feature values and from a binomial distribution for the labels.

The synthetic datasets are grouped in three *scenarios* with increasing amounts of features and labels. Each of the scenarios contains a base dataset with 10, 100 and 1000 features and labels respectively, as well as 1000 examples. For each of the base datasets, a set of variants with increased features, a set with increased examples and a set with increased labels is created. The scaling steps are $10, 100, 1000, 2000, \ldots, 10000, 20000, \ldots 50000$ for features and labels. For the examples, the scaling step 10 has been left out. In total, 129 distinct synthetic datasets are used (some configurations like 1000/1000/1000 appear in all three scaling variants). The datasets are not scaled to more than 50000 features, examples or labels to allow for a comparison to the reference CPU implementation, as the CPU implementation could not finish all five folds on synthetic datasets with 50000 features, examples or labels.

6.2.2 Real-World Datasets

The motivation for the experiments with real-world datasets is twofold. First, the real-world datasets provide a meaningful statement on the performance improvements that are achieved by the GPU implementation on commonly used datasets. They also serve as a comparison for new datasets, so that one can get a rough idea of which implementation to choose. Second, using real-world datasets allows for a comparison of the predictive performance of the models learned by the CPU and GPU implementation. Table 6.3 lists all 20 datasets that were used for the experiments with real-world datasets. The datasets are from multiple repositories, namely Mulan[4], WEKA[5] and MEKA[6]. The datasets *reuters-21578*[7] and *RCV1* (Lewis et al. 2004) are used in the word-embedded representation used by Nam et al. (2017). The dataset *genbase-without-useless-features* is a version of the dataset *genbase*, where all constant features have been removed (the filters *weka.filters.unsupervised.attribute.Remove-R1* and *weka.filters.unsupervised.attribute.RemoveUseless-M100.0* were applied). This dataset will be called *genbase-w/o* from now on for brevity.

[4] `http://mulan.sourceforge.net/datasets-mlc.html`
[5] `https://github.com/Waikato/weka-trunk/tree/master/wekadocs/data`
[6] `https://sourceforge.net/projects/meka/files/Datasets/`
[7] `http://www.daviddlewis.com/resources/testcollections/reuters21578/`
(each accessed on 17.03.2021)

Table 6.3: Used real-world datasets and their number of nominal and numerical features, examples and labels

Dataset	Features			Examples	Labels
	Total	Nominal	Numerical		
bibtex	1836	1836	0	7395	159
birds	260	2	258	645	19
bookmarks	2150	2150	0	87856	208
cal500	68	0	68	502	174
corel5k	499	499	0	5000	374
emotions	72	0	72	593	6
enron	1001	1001	0	1702	53
flags	19	9	10	194	7
genbase	1186	1186	0	662	27
genbase-without-useless-features	112	112	0	662	27
llog	1004	0	1004	1460	75
mediamill	120	0	120	43907	101
medical	1909	1909	0	1954	45
reuters-21578	512	0	512	10789	90
RCV1	512	0	512	804410	103
scene	294	0	294	2407	6
slashdot	3125	0	3125	3782	20
weather	4	2	2	14	3
weather-numerical	6	0	6	14	3
yeast	103	0	103	2417	14

6.3 Results

The previous sections introduced the algorithm configuration, the underlying hardware and the datasets. This section discusses the scaling and the achieved performance of the GPU implementation compared to the reference CPU implementation. The scaling of the two implementations on the synthetic datsets is analysed in Section 6.3.1. In Section 6.3.2, the scaling on the synthetic datasets is compared to asymptotic complexity determined in Section 4.4. The results using real-world datasets are discussed in Section 6.3.3. Finally, this chapter briefly describes some of the differences between the models learned by the CPU and the GPU implementation on the real-world datasets.

6.3.1 Performance - Synthetic Datasets

In this section, the scaling of the CPU and GPU implementation are analysed with respect to the three dimensions features, examples and labels. For each dimension, both implementations are executed with the series of datasets described in Section 6.2.1. From the asymptotic analysis of the two algorithms (compare Section 4.4), a linear scaling in all three dimensions is expected.

Figure 6.1 shows the execution time of the CPU implementation on 100 to 50 000 features for the three scenarios on the left. The execution times in this section are measured for each of the five folds and then averaged. This removes the initial overhead of loading and converting the datasets. As expected, all three scenarios show a linear increase in execution time with the number of features. The slopes of the three scenarios show that increasing the number of labels does not increase the execution time in the same way. The difference between the first and the second scenario is a factor of 10 in the labels (10 to 100). The execution times for the second scenario are higher by a factor of only 2.7x. The difference between the second and the third scenario is again factor 10 in the labels (100 to 1000). This time, the execution times are on average 11.1x higher. This indicates, that the CPU implementation is not able to utilize the hardware properly for a very low number of labels, resulting in a sub-linear scaling for very small numbers of labels.

The right part of Figure 6.1 shows the same plot for the execution times of the GPU implementation. The graphs are very similar, all three scenarios show the same linear scaling with the number of features, but on a much smaller scale (6000 compared to 100 000). Also, the execution times of the second scenario only differ by a factor of 1.1. The third scenario shows a 3.9 times higher execution time than the second scenario. This

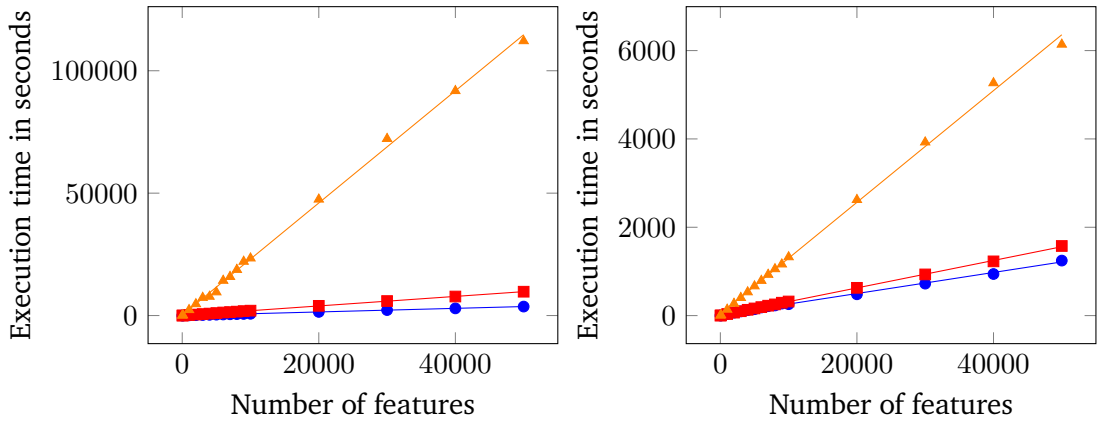


Figure 6.1: Execution time for three different scenarios and increasing number of features. The results for the CPU implementation are on the left plot, those for the GPU are on the right. The blue graph shows the scenario with 10 labels, the red graph the one with 100 labels and the orange graph shows the scenario with 1000 labels.

indicates that the GPU implementation has a scaling factor for the number of labels of less than 1 for at least up to 1000 labels. Calculating the average speedup of the GPU implementation compared to the CPU implementation yields 2.4x for the smallest scenario, 5.9x for the medium scenario and 16.5x for the large scenario. For each of the three scenarios, the speedup is not dependant on the number of features except for less than 1000 features, where the speedup drops significantly.

From the theoretical difference in memory bandwidth and FLOPS, a speedup of up to 12 and 6 was expected. This expectation is surpassed significantly by the large scenario. Possible reasons for this higher speedup are different compiler optimizations introduced by the use of the NVIDIA CUDA compiler and improved cache efficiency due to the different memory layout.

Figure 6.2 shows two similar plots for the labels. The general situation is the same, as all three scenarios exhibit a linear scaling with the number of labels. The difference in execution time between the first and the second scenario is on average 9.3. Therefore, no inefficiency for a low number of features like for a low number of labels can be observed. Between the second and the third scenario, the difference is a factor of 11.7.

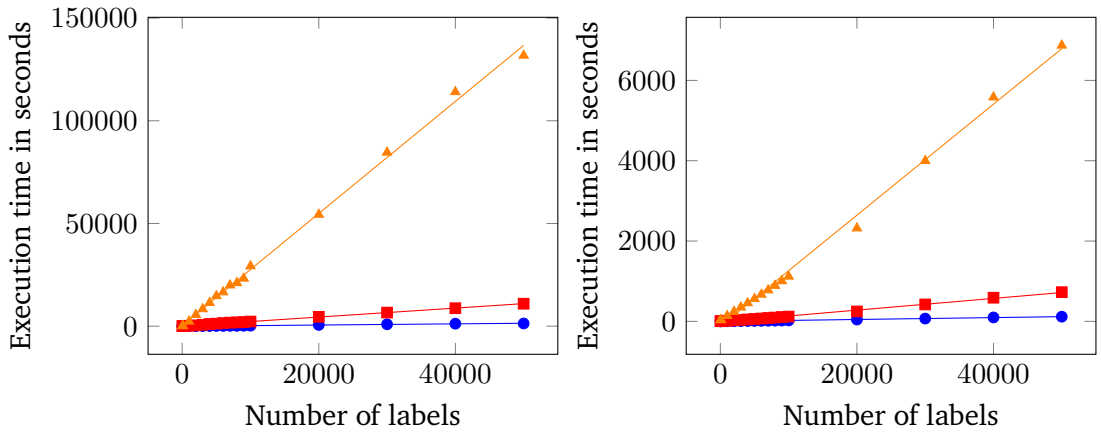


Figure 6.2: Execution time for three different scenarios and increasing number of labels. The results for the CPU implementation are on the left plot, those for the GPU are on the right. The blue graph shows the scenario with 10 features, the red graph the one with 100 features and the orange graph shows the scenario with 1000 features.

The plot for the GPU implementation on the right shows again a similar qualitative picture. All three scenarios scale linearly with the number of labels. The average ratio between the execution times of the second and the first scenario is 5.3, while the ratio between the third and the second scenario is 9.1 on average. This indicates that the first scenario is not able to utilize the GPU properly, while the second does. This is supported by the GPU utilization logged during execution with *nvidia-smi*. For the first scenario, the highest GPU utilization is on average 66.3 %. For the second scenario, 89.8 % was observed, increasing to 92.4 % for the third scenario. Both larger scenarios were able to reach 100 % utilization for 20 000 to 50 000 labels. The average speedup over the CPU implementation is 9.5 for the first scenario, 16.2 for the second and 20.6 for the third.

Figure 6.3 shows the execution time of the CPU and GPU implementation for the smallest example-scenario with 10 features and 10 labels. The blue graph in the CPU plot follows an $n \cdot log(n)$ curve, which is unexpected. A linear correlation between number of examples and execution time was expected from the asymptotic analysis. In contrast to the scaling of features and labels, the number of conditions per rule increased significantly when scaling the number of examples. This could explain the non-linear scaling. The average number of conditions per rule increases from less than 3 to more than 11 when increasing the number of examples from 100 to 50 000. To identify whether the increase in conditions

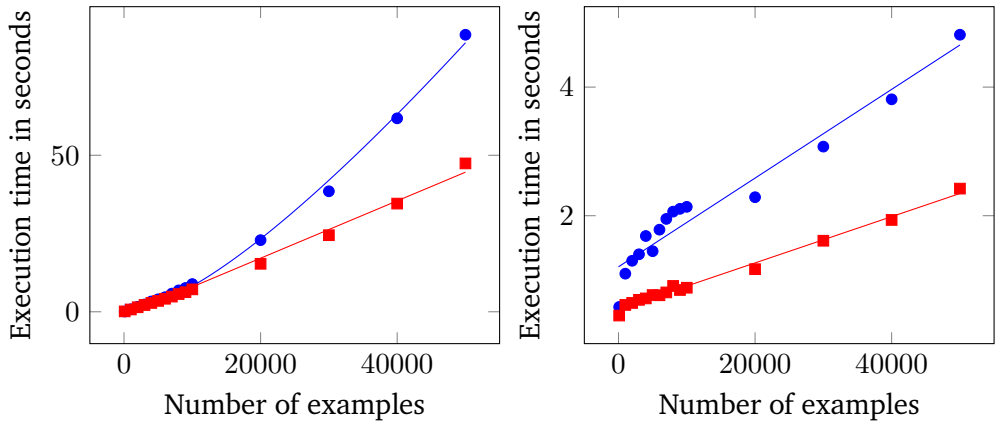


Figure 6.3: Execution time for the first scenario with 10 features and labels and increasing number of examples. The results for the CPU implementation are on the left plot, those for the GPU are on the right. The plots in blue show the regular scaling, the plots in red show the scaling when limiting the number of conditions per rule to 5.

per rule alone explains the non-linear scaling, a second set of tests was executed with the same datasets, but with an artificial limit of 5 on the number of conditions. The algorithm now stops adding conditions to a rule after the rule has 5 conditions, even if a further refinement was possible. The results of these tests are shown in red. The execution times with a limit on the number of conditions exhibit a linear scaling with the number of examples. Therefore, the increase in the number of conditions per rule for the unlimited case explains the non-linear scaling.

The second plot in Figure 6.3 shows the execution times of the GPU implementation. The variance in the blue graph is very high compared to the other plots. The execution time of the GPU implementation is below 5 seconds. Both a linear as well as a $n \cdot log(n)$ scaling could be possible. For the condition-limited graph in red, the same linear scaling as for the CPU implementation can be observed.

The graphs for the second scenario (100 features and labels) are shown in Figure 6.4. The graph for the CPU is very similar to the first scenario. The unlimited version shows an $n \cdot log(n)$ scaling, while the condition-limited version shows a linear scaling. The increase in conditions per rule now goes up to over 100 conditions pre rule for 50 000 examples. This time, the GPU graph shows the same qualitative properties as the CPU graph. Without

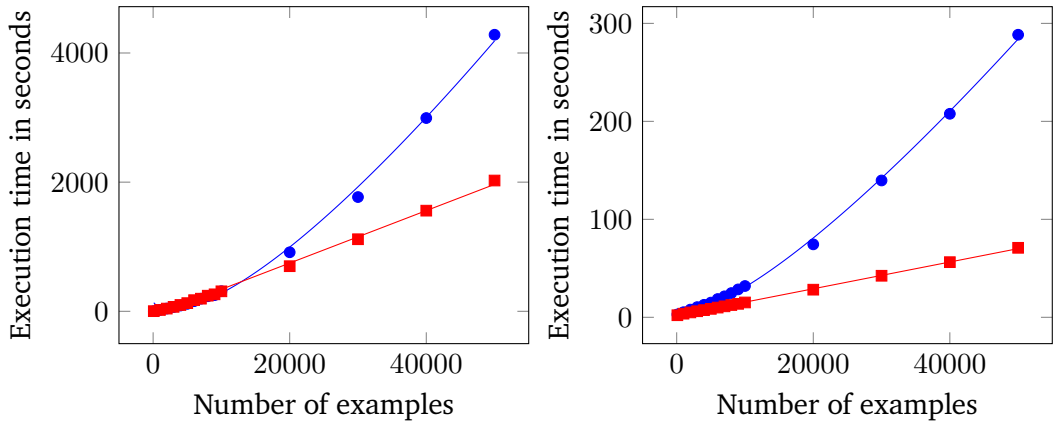


Figure 6.4: Execution time for the first scenario with 100 features and labels and increasing number of examples. The results for the CPU implementation are on the left plot, those for the GPU are on the right. The plots in blue show the regular scaling, the plots in red show the scaling when limiting the number of conditions per rule to 5.

the limit on the number of conditions per rule, the execution time scales with $n \cdot log(n)$. Limiting the number of conditions to 5 per rule results in a linear scaling similar to the CPU implementation.

The results for the third and largest example-scenario (1000 features and labels) are shown in Figure 6.5. For the CPU results, the graphs for the unlimited and the limited executions are nearly identical. The non-linear scaling of the unlimited version is much weaker compared to the second scenario. A reason for this could be the lower impact of the additional conditions on the total execution time of the algorithm. For the largest dataset (1000 features and labels, 50 000 examples), finding the first condition of each rule takes on average 1526.1 s, which is 875 times higher than the time to find each of the following conditions with an average of 1.74 s. For the largest medium dataset (100 features and labels, 50 000 examples), finding the first condition takes on average 14.6 s, while finding the following conditions requires only 0.2 s. The ratio between the two is only 73. For the largest scenario, the execution time is much more dominated by the time required to find the first condition of each rule compared to the smaller scenarios. This explains the reduced effect of limiting the number of conditions per rule.

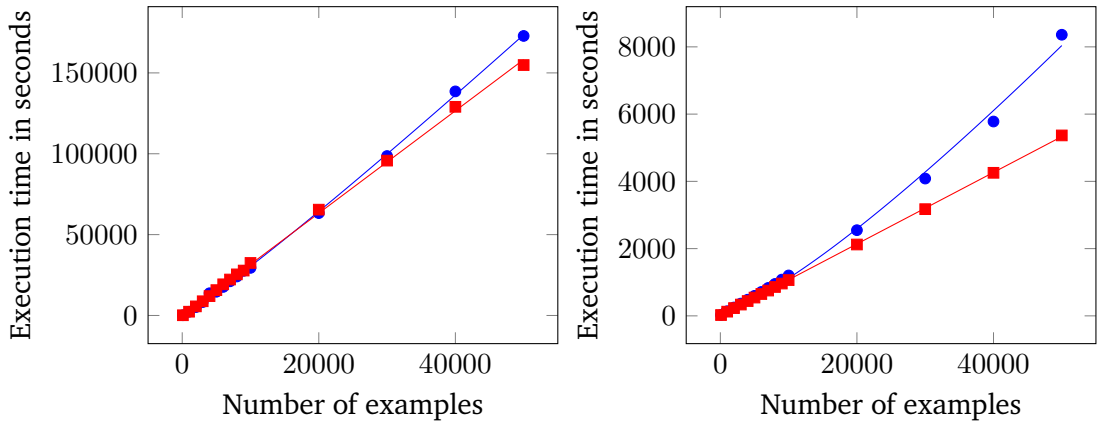


Figure 6.5: Execution time for the first scenario with 1000 features and labels and increasing number of examples. The results for the CPU implementation are on the left plot, those for the GPU are on the right. The plots in blue show the regular scaling, the plots in red show the scaling when limiting the number of conditions per rule to 5.

For the GPU plot, the difference between the unlimited and the limited graph are larger than for the CPU plot. Also, the non-linear scaling of the unlimited version is more visible. The ratio between the average time required to find the first condition per rule and the average time to find the following conditions are smaller for GPU implementation. For the medium scenario, the time for the first condition is 0.52 s, while the following conditions require an average of 0.01 s (ratio = 52). For the largest scenario, the times are 52.03 s for the first condition and 0.25 s for the following conditions (ratio = 208). While still 4 times higher, the difference is much lower than that of the CPU implementation. This explains the stronger non-linear behaviour of the GPU implementation, as the number of conditions per rule is more important than for the CPU implementation.

The 32-bit floating point version of the GPU implementation shows the same qualitative behaviour as the 64-bit GPU version for the scaling with labels. For the scaling with examples, the 32-bit version shows a linear scaling for the small and medium scenario and a sub-linear scaling for the large scenario (compared to $n \cdot log(n)$ for the 64-bit GPU version). The explanation for this behaviour is again the number of conditions per rule, which now stays around the same or decreases slightly. Calculating the execution time per condition yields a linear increase with the number of examples for all three scenarios. Considering the scaling with features, the small and medium scenario show

the same expected linear scaling as the 64-bit GPU implementation. The large scenario scales sub-linearly with the number of features similar to the scaling with the number of examples. The sub-linear scaling can not be explained by a change in the number of conditions per rule. The time required to learn the conditions after the first one increases linearly, but the time to learn the first condition increases sub-linearly. This behaviour is not explainable, but not investigated further, as the 32-bit GPU implementation is not the main focus of this work.

6.3.2 Comparison to Asymptotic Complexity

In this section, the scaling of the GPU algorithm is compared to the asymptotic complexity as determined in Section 4.4. The theoretical analysis suggests a linear increase in execution time when a dataset contains more features. This expected linear scaling is supported by the experiments on synthetic datasets with varying numbers of features. All three scenarios of different dataset sizes have exhibited a linear scaling when increasing the number of features in the dataset. The same can be said about the scaling when increasing the number of labels. From the theoretical analysis, a linear scaling is expected and this is supported by the synthetic experiments, as all three scenarios exhibited linear increase in execution time with increased number of labels in the dataset.

Concerning the number of examples in the dataset, the asymptotic complexity is more complicated. First of all, the algorithm sorts the example values for each feature, which has a lower bound of $O(n \cdot log(n))$. Then, the parallel algorithm to compute the prefix sum has at least a time complexity of $O(n + log(p))$. As the number of processors p is constant in this case (although it may vary between GPUs), the $log(p)$ term can be dropped. As the examples are sorted only once at the start of a fold, the sorting is not relevant compared to the time required to learn the rules of a fold. The experiments on synthetic datasets support this, as all three scenarios showed a linear scaling with the number of examples when limiting the number of conditions per rule.

6.3.3 Performance - Real-World Datasets

The previous sections analysed the performance of the GPU implementation on synthetic datasets. In this section, the performance of the GPU implementation is compared with the CPU implementation on the real-world datasets shown in Section 6.2.2. The GPU implementation will likely not be faster on very small datasets like *weather* due to the

Table 6.4: Execution times and calculated speedup of the CPU and the GPU implementation. All real-world datasets were executed with 5 folds and 100 rules per fold, except for *RCV1*, which finished only 1 of 2 folds due to time and memory limits.

Dataset	Execution time CPU [s]	Execution time GPU [s]	Speedup
bibtex	10585.19	777.61	13.6
birds	86.2	29.98	2.9
bookmarks	274910.1	7740.65	35.5
cal500	78.56	13.77	5.7
corel5k	2720.48	217.87	12.5
emotions	22.84	14.57	1.6
enron	596.34	305.2	2.0
flags	0.88	6.08	0.1
genbase	84.71	39.7	2.1
genbase-w/o	9.58	10.99	0.9
llog	153.61	67.12	2.3
mediamill	12944.19	422.75	30.6
medical	418.77	151.43	2.8
rcv1	260962.72	5483.76	47.6
reuters21578	12794.82	423.97	30.2
scene	491.73	83.66	5.9
slashdot	6566.14	1805.53	3.6
weather	0.13	1.37	0.1
weather-numerical	0.12	1.3	0.1
yeast	254.88	44.37	5.7

inherent overhead for CUDA kernel launches and added latency for copying data from host to device and back. The general expectation is that larger datasets result in a higher speedup, as the GPU can be utilized more by larger datasets.

Table 6.4 shows the execution time for both CPU and GPU algorithm as well as the speedup calculated by dividing the execution time of the CPU algorithm by the execution time of the GPU algorithm. For the dataset *RCV1*, only the execution time for 1 of 2 folds is available, as the GPU implementation ran out of memory for more than 2 folds, while the CPU implementation could not finish more than 1 fold within the time limit of 100 hours for jobs on the cluster nodes.

From the achieved speedups on the synthetic datasets, a speedup of up to 20 was expected. This expectation is surpassed significantly by some datasets. A possible reason for these even higher speedups is that the configuration of these datasets concerning the number of features, examples and labels is better suited for the GPU implementation than the configuration of the three synthetic scenarios.

The speedup column is very heterogeneous with large speedups of over 30x for the large datasets *RCV1*, *mediamill*, *bookmarks* and *reuters-21578*. The datsets with less than 10000 examples (e.g., *bibtex* and *corel5k*) show medium speedups up to 14x. In case of the four datasets *flags*, *genbase-w/o*, *weather* and *weather-numerical*, the GPU implementation is slower than the reference CPU implementation. Two of the largest speedups are achieved on the datsets *RCV1* and *reuters-21578*, which are used in the word-embedding representation used by Nam et al. (2017). Their high speedup is likely due to the dense numeric features values produced by the embedding, which results in a significantly higher number of conditions compared to other datasets.

In general, the GPU implementation is faster on larger datasets with more examples and labels. Datasets with a very high number of features compared to the number of examples (e.g., *enron*, *genbase* or *medical*) have a medium speedup of 2x to 4x. Figure 6.6 shows the speedup column from Table 6.4 plotted against the number of features, examples and labels. The plot shows a linear increase in speedup with a higher number of examples. For the number of labels, the speedup increases in a conical form. The minimum observed speedup increases linearly and the maximum observed speedup also increases approximately linear. The variance of the speedup therefore increases with the number of labels, together with the average speedup. For the number of features, no clear correlation is apparent. High speedups were observed both for low and high numbers of features. A very low increase of the minimum speedup with the number of features can be observed, similar to the plot against the number of labels.

Considering how the GPU version is implemented and which parts are parallelized, these results meet the expectation. One of the main computational parts, the calculation of the prefix sum, is parallel for the examples and labels, but sequential for the features (due to too high GPU memory requirements). The other computationally intensive part is the computation of predicted and quality scores. This part scales with the number of condition candidates and the number of labels. The number of conditions candidates depends on the number of features and the number of examples with unique features values for each feature. Assuming a fixed number of condition candidates, the implementation can exploit more parallelism when there are only few features and a high number of examples with unique feature values in the dataset compared to a high number of features and

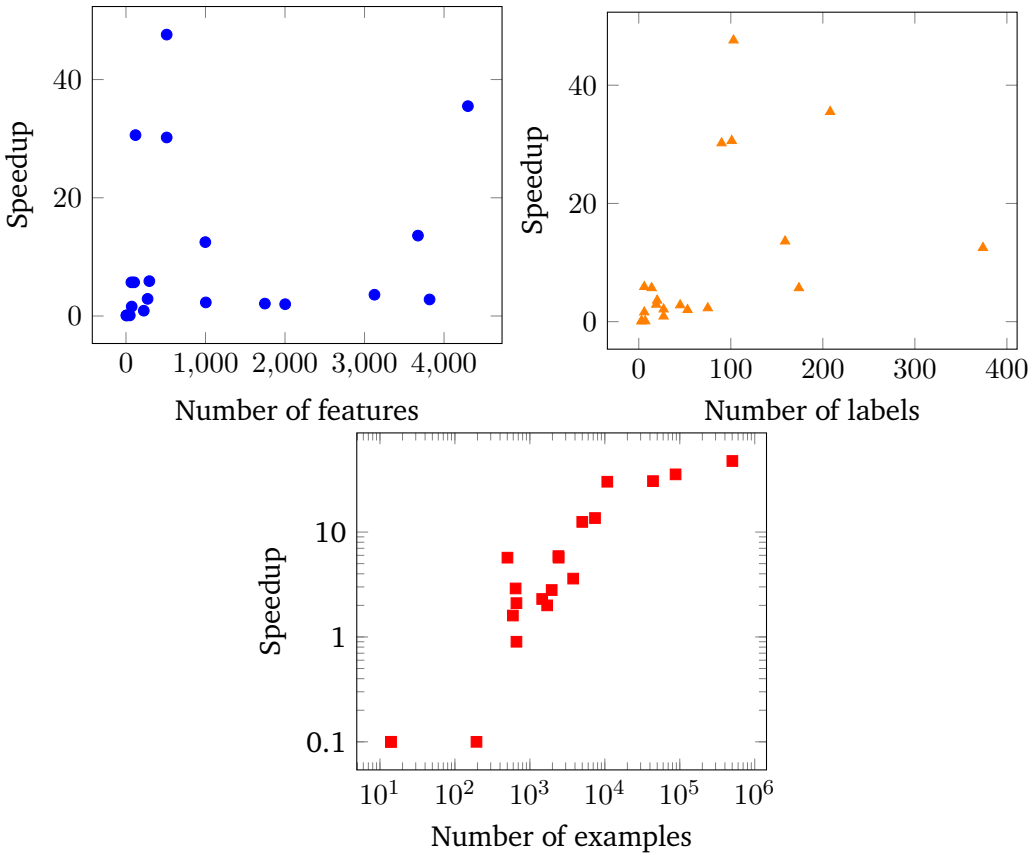


Figure 6.6: Speedup of the GPU over the CPU implementation on real-world datasets, plotted against the number of features, examples and labels.

few examples with unique feature values. It is therefore expected that the speedup over the CPU implementation is not dependent on the number of features in the dataset. A higher number of examples or labels correlates with a higher speedup as expected, as more examples and labels provide more exploitable parallelism. The speedup is expected to reach its maximum when the GPU is fully utilized, at which point the speedup should not increase further. This behaviour could be observed with the synthetic datasets at about 1000 features and labels as well as 10000 examples, where the speedup increased no further when increasing the number of examples up to 50000. For the real-world datasets, no such behaviour could be observed for the experiments that were made.

When compiling the GPU implementation with single precision floating point accuracy, the execution times listed in Table 6.5 were measured. First of all, for those datasets where the double precision GPU implementation could not provide a speedup, the situation does not change for the single precision implementation. The datasets *flags*, *genbase*, *weather* and *weather-numerical* are still faster on the CPU. For some other datasets, like *enron* and *medical*, no further speedup compared to the double precision implementation could be observed. Some other datasets with an already medium speedup, like *scene* and *yeast*, gain an additional speedup of 3 to 5 over the double precision implementation and reach the range of a high speedup of over 20x compared to the CPU implementation. Considering the column listing the speedup compared to the double precision implementation, it is apparent that no dataset yields a speedup near the theoretical difference of 32 in FLOPS. It is therefore likely that at least parts of the implementation (e.g., the prefix sum calculation) are not bound by the floating point compute capability of the GPU. Instead, the limitation could be the memory bandwidth (datasets with a speedup of about 2x) or the latency for memory reads (datasets with a speedup of about 1x) .

The datasets *bookmarks*, *mediamill*, *reuters-21578* and *RCV1* also increase in speedup, reaching a performance up to 336 higher than the CPU implementation. These very high speedup numbers should be interpreted carefully, as they are higher than the theoretical difference in floating point computing power between the CPU and GPU. For the dataset *RCV1*, the total number of conditions learned for all 100 rules of the first fold, decreased by 30 %. This reduces the total amount of work and therefore also the execution time. It is possible that the lower condition count is correlated to the reduced floating point precision. An analysis of the influence of the floating point accuracy on the algorithm behaviour and the model performance is left for future work. The differences between the models learned by the CPU and GPU implementations are briefly reviewed in Section 6.3.4. All datasets with a speedup over the double precision GPU implementation that is significantly larger than 2x, also show a significant reduction model quality, which could be a reason for the higher speedup.

Table 6.5: Execution times of the single precision GPU implementation as well as calculated speedup over CPU and double precision GPU implementation. All real-world datasets were executed with 5 folds and 100 rules per fold except for *RCV1*, which finished only 1 of 2 folds due to time and memory limits.

Dataset	Runtime GPU-32 [s]	Speedup over CPU	Speedup over GPU
bibtex	589.34	18.0	1.3
birds	8.22	10.5	3.6
bookmarks	4543.45	60.5	1.7
cal500	7.19	10.9	1.9
corel5k	144.59	18.8	1.5
emotions	6.22	3.7	2.3
enron	305.36	2.0	1.0
flags	6.06	0.1	1.0
genbase	42.15	2.0	0.9
genbase-w/o	11.76	0.8	0.9
llog	69.48	2.2	1.0
mediamill	201.39	64.3	2.1
medical	158.58	2.6	1.0
rcv1	776.08	336.3	7.1
reuters21578	85.96	148.9	4.9
scene	23.3	21.1	3.6
slashdot	1770.34	3.7	1.0
weather	1.38	0.1	1.0
weather-numerical	1.33	0.1	1.0
yeast	9.52	26.8	4.7

6.3.4 Differences in Learned Models

This section discusses differences in the models that are learned by the CPU and GPU implementation. As explained in Section 2.5, the parallel GPU implementation is very likely to produce at least slightly different numerical results when computing for example the sums of gradients or the quality scores. This chapter looks at two metrics reported by the BOOMER algorithm after a model has been learned. The first metric is called *Subset 0/1 Accuracy* and reports the percentage of examples in the test set that are correctly classified. The second metric is the *Hamming Accuracy* which reports the percentage of correctly predicted labels. For further information on the two metrics and a formal definition of the corresponding loss function, see e.g., Tsoumakas et al. (2010), M. Zhang and Zhou (2014), and Hüllermeier et al. (2020).

With both implementations, five folds with 100 rules each were learned using the hyper-parameters given in Table 6.2. No experiments were made to find the optimal hyper-parameters for the datasets. Therefore, the focus of this section is not on the actual values for subset 0/1 and Hamming accuracy, but on the respective difference between the GPU and the CPU implementation. Table 6.6 lists the differences in subset 0/1 and Hamming accuracy for the real-world datasets. For each dataset and both implementations, the number of correctly classified examples has been calculated by multiplying the subset 0/1 accuracy with the number of examples for which a prediction was made. Similarly, the number of correct labels has been calculated by multiplying the number of examples for which a prediction was made with the number of labels in the dataset and the Hamming accuracy. Then, the number of examples that are correctly classified by the CPU implementation is subtracted from the number of examples that are correctly classified by the GPU implementation. This value is the *Difference Subset 0/1*. The *Difference Hamming* is calculated in the same way using the number of correct labels.

The table shows that there are no differences in model performance for 12 of the 20 datasets and some differences for the other 8 datasets. For the 8 datasets with differences, neither implementation is always better than the other. There are 5 datasets where the GPU implementation produced a better model with respect to subset 0/1 accuracy (*emotions*), Hamming accuracy (*birds* and *slashdot*) or both (*reuters21578* and *scene*). The CPU implementation performed better on the datasets *yeast* and *medical*. The dataset *RCV1* shows an interesting result, as the model of the CPU implementation correctly classified 1 example more, but in total 14 labels less.

Considering the same metrics for the 32-bit floating point GPU implementation, the same 12 datasets as before show no differences and the datasets *slashdot* and *medical* have the

Table 6.6: Differences in subset 0/1 and Hamming accuracy between GPU and CPU implementation on real-world datasets, multiplied with the number of examples (subset) and the number of examples times the number of labels (Hamming).

Dataset	Difference Subset 0/1	Difference Hamming
bibtex	0	0
birds	0	1
bookmarks	0	0
cal500	0	0
corel5k	0	0
emotions	1	0
enron	0	0
flags	0	0
genbase	0	0
genbase-w/o	0	0
llog	0	0
mediamill	0	0
medical	-1	-1
RCV1	1	-14
reuters21578	6	3
scene	8	7
slashdot	0	1
weather	0	0
weather-numerical	0	0
yeast	0	-4

Table 6.7: Differences in subset 0/1 and Hamming accuracy between GPU-fp32 and CPU implementation on real-world datasets, multiplied with the number of examples (subset) and the number of examples times the number of labels (Hamming).

Dataset	Difference Subset 0/1	Difference Hamming
bibtex	0	0
birds	-4	-7
bookmarks	0	0
cal500	0	0
corel5k	0	0
emotions	-21	-53
enron	0	0
flags	0	0
genbase	0	0
genbase-w/o	0	0
llog	0	0
mediamill	0	0
medical	-1	-1
RCV1	-9705	-116751
reuters21578	-809	-952
scene	-183	-219
slashdot	0	1
weather	0	0
weather-numerical	0	0
yeast	-23	-150

same differences as the 64-bit GPU implementation (compare Table 6.7). The other 6 datasets are influenced heavily by the decreased floating point precision. For example, on the dataset *RCV1*, the model of the GPU-fp32 implementation does not classify a single example correctly, but only 0.3 % less labels (116751 / 40253253 of the CPU implementation).

Another metric to assess the differences in the learned models is the difference between the rule sets induced by the CPU and GPU implementation. For the five folds executed for the dataset *birds*, a total of 500 rules were induced, 100 for each fold. In total, 94 rules are different. The number of different rules per fold is 21, 15, 27, 17, 14 for folds 1 to 5.

On average, about 40 % of the rules in the induced rule sets of the 64-bit GPU implementation and about 66 % of the rules of the 32-bit GPU implementation differ from the rule sets induced by the CPU implementation. For the 32-bit GPU implementation, there are multiple datasets where all rules except for the default rule differ. This comparison does not treat the rule sets as actual sets, but instead as lists. Each rule is compared for equality with the rule that was learned at the same index in the other rule set. So if equal rules were learned in a different order, they would count as different. Considering the rule sets as actual sets and ignoring the order in which the rules were learned, on average only 24 % of the rules between the rule sets of CPU and 64-bit GPU implementation differ. For the 32-bit GPU implementation, about 45 % of the rules differ from the CPU rule set.

7 Future Work

In this chapter, multiple directions for future work based on the achievements and findings of this work are discussed. Based on the results from Section 6.3.4, an interesting topic is the influence of floating point accuracy during calculation on the quality and predictive performance of the models. This work found a significant drop in predictive performance on some datasets with numeric features when using only 32-bit floating point values instead of 64-bit. But some minor differences between the predictive performance of models learned with the CPU implementation and model learned with the GPU implementation could already be observed when using 64-bit accuracy on the GPU. A future work could extend the reference CPU implementation to work with arbitrary floating point accuracy and identify the optimal hyper-parameter for various floating point accuracies, e.g., 32- and 64-bit. With respect to GPU computing, 16-bit may also be of interest. On the other side, it may also be of interest to look at 80- or even 128-bit floating point accuracy, as it may improve the quality of learned models due to the higher floating point precision. Another aspect could be the influence of different floating point precisions on the individual parts of the rule learning process. If parts of the algorithm are less sensitive to a change in floating point accuracy, a mixed precision implementation could yield improved model performance (if higher precision is used in critical sections) or better execution time (if lower precision can be used in non-critical sections).

Another topic for future work is the extension of this GPU implementation to multiple GPUs and additional loss functions. There are multiple ways to use additional GPUs without major changes to the implementation. First of all, the prefix sum over gradients and Hessians is currently executed sequentially, as they both utilize the GPU up to 100 %. These two prefix sums could very naturally be computed in parallel on two separate GPUs. For an arbitrary number of GPUs, the same mechanism that is currently used for the feature slicing on large datasets could distribute the individual slices to different GPUs. Both extensions require the input data to be distributed over the available GPUs. The partial results then need to be gathered after the GPU-parallel part has finished.

The current GPU implementation supports only label-wise decomposable losses. For the non-decomposable example-wise losses, systems of linear equations need to be solved for each example in *find_head*. The reference CPU implementation uses Basic Linear Algebra Subprograms (BLAS) routines to solve these linear equation systems. For NVIDIA GPUs, an implementation of the BLAS routines is available with cuBLAS[1] and cuSOLVER[2]. Both libraries support applying the same operation to an array of operands with the so-called *batched* versions. Similar to the current GPU implementation, the prefix sum of gradients and Hessians could be calculated. The linear equation systems could then potentially be solved in parallel on a single or on multiple GPUs.

Another direction for future work is the optimization for sparse datasets. The current GPU implementation stores the feature matrix as a plain array, even when most entries are zero. In this scenario, using a format like Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) could reduce the memory footprint of the application significantly. In addition, due to the nature of a sparse feature matrix containing mostly zeros, an algorithmic optimization is possible for the calculation of the prefix sum. In its current implementation, the prefix sum operation sums up the gradients vectors in ascending order of feature values of their corresponding examples (similar for the Hessians). In case of a sparse dataset, most gradients correspond to an example with feature value zero. All those elements with feature value zero (except the first one) do not provide a split point. The prefix sum therefore does not need to be calculated for these entries, only for those examples with non-zero feature values before and after the zero-block. The prefix sum can be executed up to and including the gradient vector corresponding to the first examples with feature values zero. Then, as the total sum over all gradient vectors is already known, an inclusive scan can then be applied to the reverse of the last non-zero block using subtraction instead of addition and the total sums as initial value.

[1] `https://docs.nvidia.com/cuda/cublas/index.html`
[2] `https://docs.nvidia.com/cuda/cusolver/index.html`

8 Conclusion

In this work, an existing algorithm for learning gradient boosted multi-label classification rules – the BOOMER algorithm – was analysed with respect to its parallelizability and accelerated using GPUs. Three major parts – determining conditions to test, score calculation and update of gradients and Hessians – of the single-core reference implementation were identified for a parallel implementation using CUDA C/C++ and the Thrust library. The proposed parallelization of the algorithm uses additional memory to remove a data dependency in the main rule learning loop of the reference implementation. This algorithmic transformation enables a massively parallel score calculation with a custom CUDA kernel and the use of a parallel prefix sum function from the Thrust library to compute the data required inside the kernel.

Randomly generated synthetic datasets were used to analyse the scaling of the new GPU implementation in comparison to the reference CPU implementation and the expected asymptotic complexity. The expected linear scaling was verified for all three dimensions – features, examples and labels.

The GPU implementation is aimed at larger datasets and achieved speedups of up to 47x on real-world datasets during the experimental evaluation. Only minor differences in the learned models were found on some datasets. The learned rule sets differed between CPU and GPU implementation due to changes in the floating point execution order introduced by the parallel execution. More than half of the tested datasets showed equal model quality with respect to subset 0/1 accuracy and Hamming accuracy, while the other datasets showed that neither CPU or GPU consistently outperformed the other. When using single precision floating point accuracy on the GPU, the performance increased up to a speedup of 336x compared to the double precision CPU implementation, although at the cost of reduced model quality on some datasets.

Based on the presented results, the goal of this work – to accelerate the existing rule learning algorithm using GPUs – was achieved successfully. It can be concluded that the use of GPUs for the learning of gradient boosted multi-label classification rules is a viable

way forward to significantly reduce training times. With the possible future support of multi-GPU computing and further optimizations, even larger datasets can be used to train models within short amounts of time.

List of Acronyms

API Application Programming Interface

BLAS Basic Linear Algebra Subprograms

CPU Central Processing Unit

CSC Compressed Sparse Column

CSR Compressed Sparse Row

CUDA Compute Unified Device Architecture

FLOPS Floating Point Operations Per Second

FMA Fused Multiply-Add

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

ILP Inductive Logic Programming

SIMD Single Instruction Multiple Data

SM Streaming Multiprocessor

List of Figures

List of Tables

List of Algorithms

List of Listings

Bibliography

Advanced Micro Devices, Inc (Feb. 2021a). *AMD ROCm Open Software Platform*. URL: https://www.amd.com/en/graphics/servers-solutions-rocm (visited on 12. 02. 2021).

– (Feb. 2021b). *ROCm Thrust - run Thrust dependent software on AMD GPUs*. URL: https://github.com/ROCmSoftwarePlatform/rocThrust (visited on 12. 02. 2021).

Algahtani, Eyad and Dimitar Kazakov (2018). "GPU-Accelerated Hypothesis Cover Set Testing for Learning in Logic". In: *Up-and-Coming and Short Papers of the 28th International Conference on Inductive Logic Programming (ILP 2018), Ferrara, Italy, September 2-4, 2018*. Ed. by Fabrizio Riguzzi, Elena Bellodi, and Riccardo Zese. Vol. 2206. CEUR Workshop Proceedings. CEUR-WS.org, pp. 6–20. URL: http://ceur-ws.org/Vol-2206/paper1.pdf.

Blelloch, Guy E. (Nov. 1990). *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

Blelloch, Guy E. and Bruce M. Maggs (Mar. 1996). "Parallel Algorithms". In: *ACM Comput. Surv.* 28.1, pp. 51–54. ISSN: 0360-0300. DOI: 10.1145/234313.234339. URL: https://doi.org/10.1145/234313.234339.

Cano, Alberto and Bartosz Krawczyk (2018). "Learning Classification Rules with Differential Evolution for High-Speed Data Stream Mining on GPU s". In: *2018 IEEE Congress on Evolutionary Computation, CEC 2018, Rio de Janeiro, Brazil, July 8-13, 2018*. IEEE, pp. 1–8. DOI: 10.1109/CEC.2018.8477961. URL: https://doi.org/10.1109/CEC.2018.8477961.

– (Mar. 2019). "Evolving Rule-Based Classifiers with Genetic Programming on GPUs for Drifting Data Streams". In: *Pattern Recognition* 87, pp. 248–268. DOI: 10.1016/j.patcog.2018.10.024.

Dagum, L. and R. Menon (1998). "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1, pp. 46–55. DOI: 10.1109/99.660313.

Fonseca, Nuno A., Fernando Silva, and Rui Camacho (2005). “Strategies to Parallelize ILP Systems”. In: *Inductive Logic Programming*. Ed. by Stefan Kramer and Bernhard Pfahringer. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 136–153. ISBN: 978-3-540-31851-4.

Grahn, H., N. Lavesson, M. H. Lapajne, and D. Slat (2011). “CudaRF: A CUDA-based implementation of Random Forests”. In: *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, pp. 95–101. DOI: 10.1109/AICCSA.2011.6126612.

Harris, Greg, Anand V. Panangadan, and Viktor K. Prasanna (2016). “GPU-Accelerated Parameter Optimization for Classification Rule Learning”. In: *Proceedings of the Twenty-Ninth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2016, Key Largo, Florida, USA, May 16-18, 2016*. Ed. by Zdravko Markov and Ingrid Russell. AAAI Press, pp. 436–441. URL: http://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS16/paper/view/12891.

Hirsch, Christian (June 2020). *GFlops-Leistung von Prozessoren ausrechnen*. URL: https://www.heise.de/ct/artikel/GFlops-Leistung-von-Prozessoren-ausrechnen-4774315.html (visited on 03. 03. 2021).

Hüllermeier, Eyke, Johannes Fürnkranz, Eneldo Loza Mencia, Vu-Linh Nguyen, and Michael Rapp (2020). “Rule-Based Multi-label Classification: Challenges and Opportunities”. In: *Rules and Reasoning*. Ed. by Víctor Gutiérrez-Basulto, Tomáš Kliegr, Ahmet Soylu, Martin Giese, and Dumitru Roman. Cham: Springer International Publishing, pp. 3–19. ISBN: 978-3-030-57977-7.

Jansson, K., H. Sundell, and H. Boström (2014). “gpuRF and gpuERT: Efficient and Scalable GPU Algorithms for Decision Tree Ensembles”. In: *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pp. 1612–1621. DOI: 10.1109/IPDPSW.2014.180.

Klöckner, Andreas, Nicolas Pinto, Bryan Catanzaro, Yunsup Lee, Paul Ivanov, and Ahmed Fasih (2012a). “Chapter 27 - GPU Scripting and Code Generation with PyCUDA”. In: *GPU Computing Gems Jade Edition*. Ed. by Wen-mei W. Hwu. Applications of GPU Computing Series. Boston: Morgan Kaufmann, pp. 373–385. ISBN: 978-0-12-385963-1. DOI: 10.1016/B978-0-12-385963-1.00027-7. URL: https://www.sciencedirect.com/science/article/pii/B9780123859631000277.

Klöckner, Andreas, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih (2012b). “PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation”. In: *Parallel Computing* 38.3, pp. 157–174. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001. URL: https://www.sciencedirect.com/science/article/pii/S0167819111001281.

Lewis, David D., Yiming Yang, Tony G. Rose, and Fan Li (Dec. 2004). “RCV1: A New Benchmark Collection for Text Categorization Research”. In: *J. Mach. Learn. Res.* 5, pp. 361–397. ISSN: 1532-4435.

Lozano, Fernando and Pedro Rangel (2005). “Algorithms for Parallel Boosting”. In: *Proceedings of the Fourth International Conference on Machine Learning and Applications*. ICMLA ’05. USA: IEEE Computer Society, pp. 368–373. ISBN: 0769524958. DOI: 10.1109/ICMLA.2005.8. URL: https://doi.org/10.1109/ICMLA.2005.8.

Mitchell, Rory and Eibe Frank (July 2017). “Accelerating the XGBoost algorithm using GPU computing”. In: *PeerJ Computer Science* 3, e127. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.127. URL: https://doi.org/10.7717/peerj-cs.127.

Nam, Jinseok, Eneldo Loza Mencía, Hyunwoo J. Kim, and Johannes Fürnkranz (2017). “Maximizing Subset Accuracy with Recurrent Neural Networks in Multi-label Classification”. In: *Advances in Neural Information Processing Systems 30 (NIPS-17)*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett. Long Beach, CA, pp. 5419–5429. URL: http://papers.nips.cc/paper/7125-maximizing-subset-accuracy-with-recurrent-neural-networks-in-multi-label-classification.

NVIDIA Corporation (Sept. 2018). *NVIDIA Turing GPU Architecture*. Tech. rep. WP-09183-001_v01. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (visited on 18. 03. 2021).

– (Dec. 2020a). *CUDA C++ Best Practices Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (visited on 18. 01. 2021).

– (Dec. 2020b). *Thrust Quick Start Guide*. URL: https://docs.nvidia.com/cuda/thrust/index.html (visited on 18. 01. 2021).

– (Jan. 2021a). *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 18. 01. 2021).

– (Jan. 2021b). *GeForce RTX 2080 Ti*. URL: https://www.nvidia.com/de-de/geforce/graphics-cards/rtx-2080-ti/ (visited on 19. 01. 2021).

– (Jan. 2021c). *thrust: Modules*. URL: https://thrust.github.io/doc/modules.html (visited on 18. 01. 2021).

OpenACC Organization (Feb. 2021). *OpenACC*. URL: https://www.openacc.org (visited on 12. 02. 2021).

Ou, Rong (2020). *Out-of-Core GPU Gradient Boosting*. arXiv: 2005.09148 [cs.LG].

Rapp, Michael, Eneldo Loza Mencía, Johannes Fürnkranz, Vu-Linh Nguyen, and Eyke Hüllermeier (2021). “Learning Gradient Boosted Multi-label Classification Rules”. In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Frank Hutter, Kristian

Kersting, Jefrey Lijffijt, and Isabel Valera. Cham: Springer International Publishing, pp. 124–140. ISBN: 978-3-030-67664-3.

Skryjomski, Przemysław, Bartosz Krawczyk, and Alberto Cano (2019). "Speeding up k-Nearest Neighbors classifier for large-scale multi-label learning on GPUs". In: *Neurocomputing* 354. Recent Advancements in Hybrid Artificial Intelligence Systems, pp. 10–19. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2018.06.095. URL: https://www.sciencedirect.com/science/article/pii/S0925231219304588.

Tsoumakas, Grigorios, Ioannis Katakis, and Ioannis P. Vlahavas (2010). "Mining Multi-label Data". In: *Data Mining and Knowledge Discovery Handbook, 2nd ed*. Ed. by Oded Maimon and Lior Rokach. Springer, pp. 667–685. DOI: 10.1007/978-0-387-09823-4_34. URL: https://doi.org/10.1007/978-0-387-09823-4%5C_34.

Van Essen, B., C. Macaraeg, M. Gokhale, and R. Prenger (2012). "Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?" In: *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 232–239. DOI: 10.1109/FCCM.2012.47.

Wen, Z., B. He, R. Kotagiri, S. Lu, and J. Shi (2018). "Efficient Gradient Boosted Decision Tree Training on GPUs". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 234–243. DOI: 10.1109/IPDPS.2018.00033.

Wen, Z., J. Shi, B. He, J. Chen, K. Ramamohanarao, and Q. Li (2019). "Exploiting GPUs for Efficient Gradient Boosting Decision Tree Training". In: *IEEE Transactions on Parallel and Distributed Systems* 30.12, pp. 2706–2717. DOI: 10.1109/TPDS.2019.2920131.

Whitehead, Nathan and Alex Fit-Florea (Dec. 2020). *Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. Tech. rep. TB-06711-001_v11.2. NVIDIA Corporation. URL: https://docs.nvidia.com/cuda/pdf/Floating_Point_on_NVIDIA_GPU.pdf (visited on 18. 01. 2021).

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

Zhang, M. and Z. Zhou (2014). "A Review on Multi-Label Learning Algorithms". In: *IEEE Transactions on Knowledge and Data Engineering* 26.8, pp. 1819–1837. DOI: 10.1109/TKDE.2013.39.

Zheng, X., P. Li, Z. Chu, and X. Hu (2020). "A Survey on Multi-Label Data Stream Classification". In: *IEEE Access* 8, pp. 1249–1275. DOI: 10.1109/ACCESS.2019.2962059.