technische universität
dortmund

Bachelorarbeit

Vergleich einer einheitlichen Implementierung von QuickScorer und RapidScorer mit OpenMP

Simon Koschel

Gutachter:
Prof. Dr. Katharina Morik
M.Sc. Sebastian Buschjäger

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS-8)
https://www-ai.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Hintergrund

Maschinelles Lernen ist in der heutigen Zeit allgegenwärtig und wird immer mehr Bestandteil des Alltags vieler Menschen. Seien es Empfehlungsdienste in sozialen Netzwerken und Online-Shops oder medizinische Diagnosen [37, S. 19]. Maschinelles Lernen findet immer häufiger und in den unterschiedlichsten Bereichen Anwendung.

Diese vermehrte Nutzung bringt somit auch einen höheren Energieverbrauch mit sich. Beispielsweise nutzen Suchmaschinen auf maschinellem Lernen basierende Algorithmen, um Suchanfragen zu verarbeiten. Allein für Google waren dies 2012 bereits mehr als eine Billion Suchanfragen pro Jahr. Mittlerweile sind es Schätzungen zufolge mehr als drei Billionen [15]. Da diese Anzahl mit jedem Jahr steigt, ist es eine wichtige Aufgabe, Methoden zu erforschen, die solche Prozesse effizienter machen und somit Energiekosten sparen. In Deutschland wird heutzutage noch immer etwa die Hälfte des Nettostroms aus nicht erneuerbaren Energien gewonnen [11]. Deshalb bedeutet eine Steigerung der Effizienz auch eine Einsparung an Energiekosten und hat somit einen direkten positiven Einfluss auf die Umwelt.

Ebenfalls in Endverbraucherprodukten wie Kameras und Smartphones findet man immer häufiger Anwendungen, die maschinelles Lernen nutzen. Diese können beispielsweise Objekte in Bildern erkennen oder Gesprochenes in Text übersetzen [17]. Jedoch ist die Nutzung von maschinellem Lernen auf ressourcenbeschränkter Hardware wie mobilen Endgeräten oft nicht möglich, da die Rechenleistung oder der verfügbare Speicher zur Nutzung nicht ausreicht. Deshalb ist es sinnvoll, Methoden zu erforschen, welche die Ausführung von Anwendungen des maschinellen Lernens auch für kleine Geräte effizienter gestalten.

Eine Möglichkeit dafür nennt sich Vektorisierung. Dabei wird dieselbe Operation in einem Prozessor gleichzeitig auf mehreren Datenelementen mit einer einzelnen Instruktion durchgeführt. Diese Methode wurde bereits im Kontext von Datenbanken [28] und einer State-Of-The-Art-Methode des maschinellen Lernens namens *Ensemble Learning* [22, 36]

genutzt, um die Effizienz zu steigern. Jedoch ist eine solche Anwendung stets hardwareabhängig und erfordert spezielles Wissen über die genutzte Ausführungsplattform.
Ziel dieser Arbeit ist es, eine plattformunabhängige vektorisierte Implementierung der Algorithmen QuickScorer und RapidScorer zu erarbeiten. Für die Vektorisierung soll die Programmierschnittstelle OpenMP verwendet werden. Mithilfe dieser Implementierungen wird daraufhin die Möglichkeit untersucht, diese Algorithmen auch auf ARM-Prozessoren zu verwenden.

1.2 Aufbau der Arbeit

Im Folgenden werden in Kapitel 2 die benötigten Grundlagen für diese Arbeit erläutert. Abschnitt 2.1 erklärt Begriffe und Definitionen rund um das maschinelle Lernen sowie Entscheidungsbäume und Ensembles. In Abschnitt 2.1.2 wird zudem kurz das Problem Learning to Rank vorgestellt. In Abschnitt 2.2 werden daraufhin das Konzept von SIMD erläutert und stellt in den Abschnitten 2.2.1 und 2.2.2 die SIMD-Erweiterungen AVX und NEON vor. Zuletzt wird die Programmierschnittstelle OpenMP vorgestellt und beschrieben, wie man damit Code vektorisieren kann.
In Kapitel 3 wird diese Arbeit in den Kontext der aktuellen Forschung rund um das Thema der Traversierung von Ensemblen von Entscheidungsbäumen gesetzt. Es werden verschiedene Algorithmen und Ansätze vorgestellt, in denen versucht wird, das Traversieren zu beschleunigen.
In Kapitel 4 werden die in dieser Arbeit untersuchten Algorithmen QuickScorer, V-QuickScorer und RapidScorer vorgestellt. Die Abschnitte 4.2 und 4.3 beschreiben die genauen Implementierungen mit den SIMD-Erweiterungen AVX und NEON, indem der Pseudocode auf die verwendeten SIMD-Instrinsic-Funktionen abgebildet wird. In Abschnitt 4.4 wird versucht, die skalaren Versionen der Algorithmen zu vektorisieren. Abschnitt 4.5 zeigt die Grenzen der hier genutzten Implementierungen auf.
Die Implementierungen aus Kapitel 4 werden daraufhin für die in Kapitel 5 dargestellten Experimente verwendet. In Abschnitt 5.1 folgt eine Aufstellung von verschiedenen Hypothesen. Abschnitt 5.2 beschreibt die Konfiguration der Experimente. Das Verhalten der Algorithmen auf verschiedenen Datensätzen wird in Abschnitt 5.3 analysiert und miteinander verglichen. Hinzu kommen Vergleiche über verschiedene Ausführungsplattformen.
Zuletzt werden in Kapitel 6 die Ergebnisse der Evaluation zusammengefasst. Zudem wird ein Ausblick auf mögliche weiterführende Forschung in diesem Bereich geliefert.

Kapitel 2

Grundlagen

In diesem Kapitel werden in Abschnitt 2.1 die für die vorliegende Arbeit notwendigen Grundlagen bezüglich des maschinellen Lernens erläutert. In Abschnitt 2.2 wird daraufhin das Konzept der Vektorisierung von Code mithilfe von SIMD und die SIMD-Erweiterungen AVX und NEON der Prozessorhersteller Intel und ARM erklärt. Abschnitt 2.2.3 stellt die Programmierschnittselle OpenMP vor und, wie damit gegebener Code plattformunabhängig vektorisiert werden kann.

2.1 Maschinelles Lernen

Maschinelles Lernen beschreibt ein Verfahren, das in gegebenen Daten Muster oder Strukturen erkennt und versucht, das Gelernte zu verallgemeinern. Damit soll erreicht werden, dass die Muster auch in anderen unbekannten Daten erkannt werden können. Es werden mit Algorithmen statistische Modelle aufgebaut, die Probleme lösen können, bei denen Menschen nicht mehr in der Lage sind, Muster zu erkennen. Im Folgenden werden einige Begriffe rund um das maschinelle Lernen definiert.
Die Menge von Daten, in denen Muster gefunden werden sollen, werden als $\mathcal{X}$ bezeichnet. Jedes Element $\mathbf{x} \in \mathcal{X}$ nennt man einen *Merkmalsvektor* oder *Featurevektor*. Ein Featurevektor $\mathbf{x} \in \mathcal{X}$ ist ein m-dimensionaler Vektor von reellen Zahlen $\mathbf{x} \in \mathbb{R}^m$. Jeder Wert $x^{(j)}, j = 1, ..., m$ beschreibt $\mathbf{x}$ auf eine bestimmte Weise. Ein solcher Wert heißt *Merkmal* oder *Feature* [35, S. 1-2]. Zu jedem $\mathbf{x}_i \in \mathcal{X}$ mit $i = 1, ..., N$ gibt es ein *Label* $y_i \in \mathcal{Y}$. Dieses entspricht der korrekten Vorhersage zu dem Featurevektor $\mathbf{x}_i$. Sind die Vorhersagen reelle Zahlenwerte $\mathcal{Y} = \mathbb{R}$, so redet man von Regression. Sind die Vorhersagen Elemente einer endlichen Menge von Klassen $\mathcal{Y} = \{0, ..., C\}$ mit $C \in \mathbb{N}, C \geq 1$, so redet man von Klassifikation. Ein Datensatz mit Labeln ist definiert als $\mathcal{D} = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_N, y_N) | (\mathbf{x}_i, y_i \in \mathcal{X} \times \mathcal{Y})\}$. Ein Modell repräsentiert eine Funktion $f_\theta : \mathcal{X} \to \mathcal{Y}$ mit den Parametern θ. Das Ziel ist, dass das Modell die zu klassifizierenden Daten $\mathbf{x}_i \in \mathcal{X}$ möglichst genau auf $y_i \in \mathcal{Y}$ abbildet. Um das zu erreichen, werden die Parameter θ mithilfe von Algorithmen angepasst [35,

S. 2]. Das Trainieren eines Modells mit einem gelabelten Trainingsdatensatz nennt man *Überwachtes Lernen* [31, S. 22]. Diese Art des maschinellen Lernens wird in dieser Arbeit verwendet.
Um zu beschreiben, wie gut ein Modell Eingaben klassifizieren kann, wird im Folgenden eine Metrik definiert. Im Fall der binären Klassifikation kann ein Modell für einen gegebenen Merkmalvektor $\mathbf{x}$ eine von zwei Klassen $\{0, 1\}$ vorhersagen. Dabei kann das Modell das Label y von $\mathbf{x}$ korrekt oder falsch vorhersagen. Insgesamt ergeben sich vier mögliche Szenarien:

- Richtig positiv: y ist 1 und das Modell sagt 1 vorher.
- Falsch negativ: y ist 1 und das Modell sagt 0 vorher.
- Falsch positiv: y ist 0 und das Modell sagt 1 vorher.
- Richtig negativ: y ist 0 und das Modell sagt 0 vorher.

Für M Wiederholungen sei TP die Anzahl der richtig positiven, FN die Anzahl der falsch negativen, FP die Anzahl der falsch positiven und TN die Anzahl der richtig negativen Ergebnisse. Die Genauigkeit Acc eines Modells über einem Datensatz ist definiert als:

$$Acc = \frac{TP + TN}{M} \tag{2.1}$$

Die Genauigkeit entspricht also dem Anteil der korrekt klassifizierten Daten.
Ein Modell soll nicht nur in der Lage sein, die für das Training verwendeten Daten zu klassifizieren. Es sollen generelle Regeln und Strukturen in den Daten gefunden werden, sodass diese auch auf nicht bekannte Daten angewendet werden können. Um das zu erreichen, teilt man einen Datensatz in drei disjunkte Teilmengen:

1. Trainingsdatensatz
2. Validierungsdatensatz
3. Testdatensatz

Das Modell wird ausschließlich mit dem Trainingsdatensatz trainiert. Daraufhin wird mit dem Validierungsdatensatz das beste Modell ausgewählt. Mit dem Testdatensatz wird dann anhand einer ausgewählten Metrik untersucht, wie gut das Modell in der Lage ist, die Daten zu generalisieren. Validierungsdatensatz und Testdatensatz sind meistens ähnlich groß und deutlich kleiner als der Trainingsdatensatz [35, S. 49].

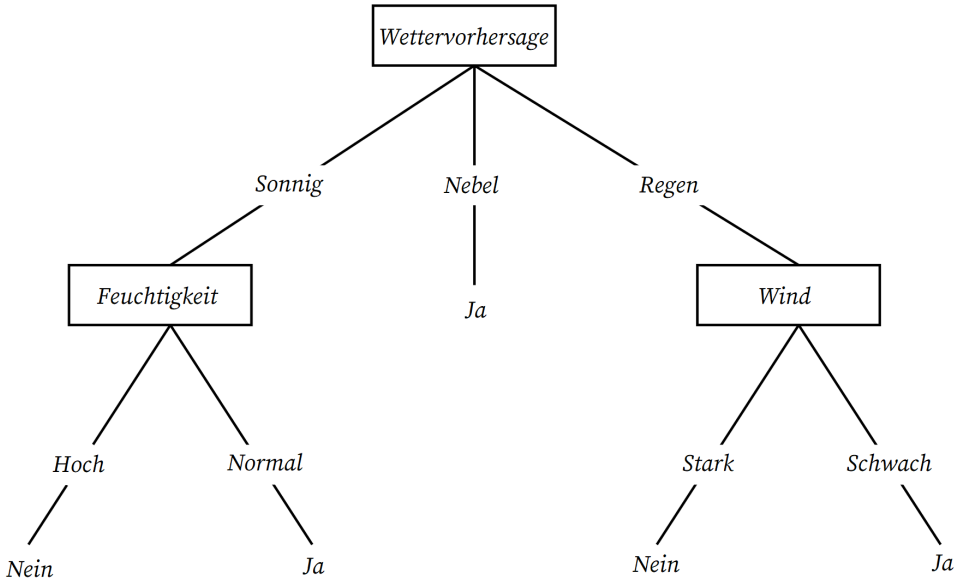

Abbildung 2.1: Ein Entscheidungsbaum zu der Frage "Werde ich heute Tennis spielen gehen?" [23]

2.1.1 Entscheidungsbäume

Entscheidungsbäume sind gerichtete Graphen, die zur Entscheidungsfindung genutzt werden. Ein Entscheidungsbaum $T = (N, L)$ besteht aus seinen Knoten $N = \{n_1, n_2, ...\}$ und Blättern $L = \{l_1, l_2, ...\}$. Zu jedem Knoten n gehört ein Test gegen ein bestimmtes Merkmal, definiert durch eine Feature-ID ϕ zu der Menge von Features $\mathcal{F} = \{f_1, f_2, ...\}$, einem Schwellwert $\gamma \in \mathbb{R}$ und Zeigern auf den linken und rechten Kindknoten. Um einen Baum zu durchlaufen, wird ein Featurevektor $\mathbf{x}$ in den Entscheidungsbaum hineingegeben. Gegen dessen Featurewerte $x^{(j)}$ wird in den Knoten getestet. Ist für einen Knoten n $x^{(\phi)} \leq \gamma$ erfüllt, so nennt man n einen *wahren* Knoten und folgt dem Pfad zum linken Kindknoten. Anderenfalls nennt man n einen *falschen* Knoten und folgt dem Pfad zum rechten Kindknoten. Letztlich landet man so bei einem der Blätter des Baumes, beim sogenannten *Austrittsblatt*. Jedes Blatt $l \in L$ enthält eine Vorhersage des Baumes. $f(\mathbf{x})$ bezeichnet die Vorhersage des Austrittsblatts für einen Featurevektor $\mathbf{x}$. Nutzt man Entscheidungsbäume für eine Klassifikation, so enthalten die Blätter eine der Klassen. Für Regression enthalten sie reelle Zahlenwerte. Zusätzlich wird Λ als die maximale Anzahl von Blättern in den Bäumen von $\mathcal{T}$ definiert. In Abbildung 2.1 ist ein einfaches Beispiel eines Entscheidungsbaums zur Klassifikation zu sehen. Die Struktur und die Werte in den Knoten und Blättern in einem Entscheidungsbaum können mithilfe von maschinellem Lernen antrainiert werden. Ein Algorithmus, der das macht, ist beispielsweise *ID3* [30].

Ensembles Anstatt ein sehr großes Modell mit sehr hoher Genauigkeit zu trainieren, werden beim *Ensemble Learning* viele kleine Modelle, sogenannte *schwache Lerner*, mit

niedrigerer Genauigkeit trainiert. Deren Ausgaben zusammengenommen können wiederum eine Ausgabe mit hoher Genauigkeit liefern. Ein Ensemble von Entscheidungsbäumen ist definiert als eine Menge von Entscheidungsbäumen $\mathcal{T} = \{T_1, T_2, ...\}$. Für Regressionsaufgaben wird die Vorhersage eines Ensembles durch eine gewichtete Addition der Vorhersagen der einzelnen Bäume $T_h = (N_h, L_h)$ in $\mathcal{T}$ berechnet:

$$F(\mathbf{x}) = \sum_{h=1}^{|\mathcal{T}|} w_h \cdot f_h(\mathbf{x}) \tag{2.2}$$

mit dem Gewicht $w_h \in \mathbb{R}$ von jedem Baum. Bei Klassifikationsaufgaben geschieht das häufig durch eine gewichtete Abstimmung der Bäume. Die zwei meistgenutzten und performantesten Ensemblemethoden sind *Gradient Boosting* [37, S. 23] und *Random Forest* [35, S. 93].

Bagging Zum Erstellen eines Random Forest [4] wird die Methode *Bagging* genutzt. Bagging (kurz für *Bootstrap Aggregating*) beschreibt eine Methode, aus zufällig ausgewählten Stichproben eines Datensatzes unterschiedliche Modelle zu generieren. Daraus werden durch einen Lernalgorithmus die kleinen Modelle erstellt. Oft kommt noch hinzu, dass jeder Baum des Ensembles nur durch einen Teil der gesamten Merkmale erstellt wird. Diesen Vorgang nennt man *Subspace Sampling* und fördert die Diversität innerhalb des Ensembles. Dies hat den Vorteil, dass die Trainingsdauer der Bäume reduziert wird. Bagging zusammen mit Subspace Sampling ergibt die Ensemblemethode Random Forest. [10, S. 331-333].

Boosting Bei gradientengestärkten Entscheidungsbäumen wird die Methode *Boosting* verwendet. Dabei startet man mit einem schwachen Lerner, der gerade häufiger als der Zufall richtige Voraussagen macht und fügt sequentiell weitere Lerner hinzu. Jedes hinzugefügte Modell versucht dabei, den Fehler in der Vorhersage des vorherigen Modells zu korrigieren. Zusammen können diese Bäume einen starken Lerner ergeben [37, S. 23-24].

2.1.2 Learning to Rank

Learning to Rank (LtR) bezeichnet ein Problem, wobei eine Menge von Dokumenten in eine Reihenfolge nach Relevanz gebracht werden muss. Die meistgenutzte Anwendung von LtR findet sich bei der Sortierung von Suchergebnissen bei Internetsuchmaschinen. Gegeben einer Menge von Anfrage-Dokument-Paaren (q, d_i), die durch einen Featurevektor $\mathbf{x}$ dargestellt werden, wird eine Relevanzpunktzahl $S(\mathbf{x})$ gefunden. Damit erfolgt daraufhin eine Sortierung der Dokumente d_i. Dieser Prozess wird auch als *Ranking* bezeichnet. Die effektivsten LtR-Algorithmen sind Gradient-Boosted Regression Trees (GBRT) und LambdaMART (λ-MART) [21] und erzeugen Ensembles von gewichteten Regressionsbäumen.

2.2 Vektorisierung mit SIMD

In diesem Abschnitt werden die Begriffe *SIMD* (Single Instruction, Multiple Data) und *Vektorisierung* erklärt, sowie verschiedene Methoden gezeigt, SIMD-Erweiterungen zu nutzen. Durch eine einzelne SIMD-Instruktion wird dem Prozessor vorgegeben, dass er dieselbe Operation auf mehreren Datenelementen derselben Größe und desselben Datentyps ausführen soll [24, S. 1108]. Im Gegensatz dazu nennt man einen Prozessor oder Algorithmus skalar, in denen pro Instruktion nur ein Datenelement verarbeitet wird. Um dies zu bewerkstelligen, werden spezielle SIMD-Register verwendet, die breiter als die gängigen 64-Bit-Register heutiger Hardware sind. Es gibt je nach Vektorerweiterung 128 Bit, 256 Bit oder sogar 512 Bit breite Register. Somit können in diesen Registern verschieden viele Werte gleichzeitig gespeichert werden. In einem 128-Bit-Register können beispielsweise acht 16-Bit-Zahlen oder zwei 64-Bit-Zahlen verarbeitet werden. Bei einem solchen Register mit acht 16-Bit-Zahlen sagt man, dass das Register in acht *Lanes* aufgeteilt wird. Auf den Werten in diesen Registern können gängige Operationen, wie Additionen, Multiplikationen etc. parallel ausgeführt werden, wobei sowohl Ganzzahl- als auch Fließkommaoperationen verschiedener Längen unterstützt werden. In den meisten Fällen ist es möglich, ein gesamtes SIMD-Register innerhalb eines Zyklus zu verarbeiten [13, S. 10]. Nimmt man nun acht 32-Bit-Gleitkommazahlen und möchte mit diesen nun eine Operation durchführen, so müsste man dem skalaren Ansatz nach achtmal dieselbe Instruktion ausführen. Mit SIMD würde man diese acht 32-Bit-Zahlen in ein 256-Bit-Register laden, dort eine Instruktion ausführen und daraufhin die Werte zurück in den Speicher laden. Somit ist eine theoretische Verbesserung der Laufzeit um das achtfache möglich [24, S. 1108].

SIMD parallelisiert Code auf einer anderen Ebene als es beispielsweise bei *Multithreading* der Fall ist. Beim Multithreading können mehrere Kerne dieselbe Operation parallel durch verschiedene Threads ausführen lassen. Der Kontrollfluss und die genutzten Instruktionen können somit bei jedem Thread unterschiedlich sein. SIMD arbeitet nur auf einem Thread, sodass alle Datenelemente demselben Kontrollfluss folgen müssen und auch immer dieselben Operationen ausgeführt werden. Multithreading parallelisiert somit auf Threadebene, während SIMD auf Datenebene parallelisiert. Die beiden Techniken können auch kombiniert werden. Für den Fall, dass für ein Datenelement bereits alle Operationen abgeschlossen sind, für ein anderes Element jedoch noch weitere Operationen auszuführen sind, ist es auch möglich, die Operationen bedingt auf den Elementen ausführen zu lassen. Der Prozessor würde weiterhin auch für die abgeschlossenen Datenelemente die Instruktionen ausführen, wobei die Ergebnisse dann für diese gegebenenfalls verworfen werden [13, S. 12]. Dieses Verhalten nennt man *Control Divergence* und kann die Ausführung von Code mit SIMD-Instruktionen verlangsamen.

Es gibt drei Möglichkeiten, SIMD-Instruktionen zu verwenden:

1. Direkter Einsatz von Assembly-Instruktionen
2. Einsatz von vom Compiler unterstützten *Intrinsic*-Funktionen
3. Auto-Vektorisierung durch den Compiler

Direktes Schreiben von Assemblycode mit SIMD-Instruktionen wird oft als die beste Methode für die höchste Geschwindigkeit bei Vektorisierung angesehen. Jedoch ist sie auch im hohen Maße fehleranfällig und kostet viel Zeit, da der Programmierer die SIMD-Instruktionen und Registerzugriffe selber schedulen, die Register per Hand allozieren und den Call-Stack eigenständig verwalten muss [24, S. 1108]. Solche Dinge werden dem Programmierer bei der Nutzung von Intrinsic-Funktionen abgenommen. SIMD-Intrinsic-Funktionen kapseln die SIMD-Assembly-Instruktionen in normale Funktionsaufrufe für C/C++ und bieten dem Programmierer eine höhere Abstraktionsebene als Assembly-Code. Diese Intrinsic-Funktionen werden *inline* aufgerufen. Das bedeutet, dass der Rumpf einer Funktion an der Stelle eingefügt wird, wodurch der zusätzliche Aufwand eines Funktionsaufrufs nicht auftritt. Dadurch bietet es dieselben Vorteile wie SIMD-Assembly-Code und zusätzlich eine bessere Lesbarkeit und weniger Fehler während der Programmierung. Ein Nachteil ist jedoch, dass der Geschwindigkeitsschub nun abhängig davon ist, wie gut der Compiler auf Registerebene das Schedulen der Aufrufe der Intrinsic-Funktionen optimieren kann [24, S. 1108]. Durch eigene Datentypen wird zudem die Typenüberprüfung deutlich vereinfacht.
Bei der Auto-Vektorisierung bleibt es dem Compiler vollständig überlassen, eigenständig passende Schleifen im Code zu finden, welche dann vektorisiert werden können. Dabei ist die Fähigkeit des Programmierers, auto-vektorisierbaren Code zu schreiben, wichtig. Der Compiler muss jedoch auch in der Lage sein, vektorisierbaren Code zu erkennen. Unterschiedliche Compiler können unterschiedlich gut sein, Code automatisch zu vektorisieren.

2.2.1 Advanced Vector Extensions (AVX)

Intels *Advanced Vector Extensions* (AVX) bezeichnet eine Menge von SIMD-Instruktionen für Intel-Prozessoren. Diese erweitert vorherige SIMD-Instruktionen (MMX (Multi Media Extension) und SSE (Streaming SIMD Extensions)) durch Neuerungen an der Hardware und neue Instruktionen [19, S. 1]:

- 128-Bit-Register von SSE werden auf 256 Bit vergrößert.
- Instruktionen mit drei Operanden sind nun möglich. Zuvor war stets der erste Operand auch das Zielregister der Operation. Somit kann Code geschrieben werden, der nicht stets einen der Parameter überschreibt.

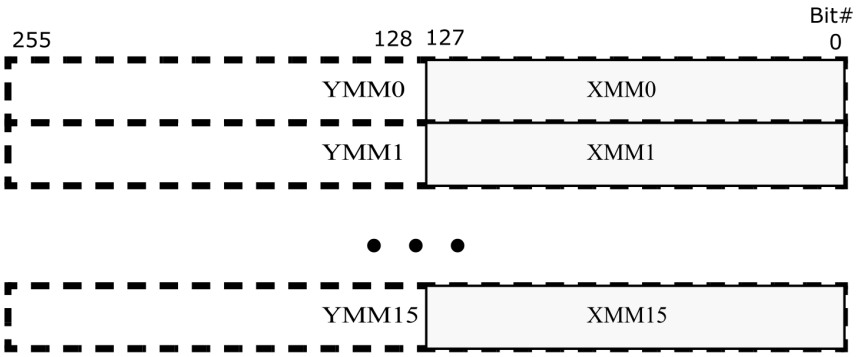

Abbildung 2.2: Die YMM-Register von AVX überdecken die XMM-Register von SSE [14].

- vier Operanden sind für einige wenige Instruktionen auch möglich und können so denselben Code in weniger Instruktionen umsetzen.
- Die Ansprüche an die Speicherausrichtung sind weniger streng als bei SSE.

AVX2 ist eine Erweiterung zu AVX, in der viele 128-Bit-Instruktionen für Ganzzahlen auch für 256 Bit breite Register unterstützt werden. Zusätzlich unterstützt werden beispielsweise das Laden von nicht im Speicher zusammenhängenden Datenelementen für Ganz- und Fließkommazahlen von 32 und 64 Bit Größe oder Vektor-Shift-Instruktionen für 32 und 64 Bit. Auch einige Funktionalitäten, die über Lanes eines einzelnen SIMD-Registers operieren, werden durch neue Instruktionen unterstützt. Außerdem wird für jede AVX-Instruktion auf Fließkommazahlen das Gegenstück einer AVX2-Instruktion auf 32 oder 64 Bit großen Ganzzahlelementen ergänzt [14, S. 21-22].
SSE verwendet 16 128 Bit breite XMM-Register XMM0 - XMM15. Diese wurden mit AVX durch die 256 Bit breiten YMM-Register YMM0 - YMM15 ersetzt. SSE-Instruktionen können jedoch auf dieser Hardware weiterhin genutzt werden, da, wie in Abbildung 2.2 zu sehen, die XMM-Register von SSE den unteren 128 Bit der YMM-Register entsprechen [14, S. 34].
Möchte man in C/C++ vektorisiert programmieren, so kann man AVX-Intrinsic-Funktionen dazu verwenden.[1] Dazu muss zunächst die Headerdatei `immintrin.h` inkludiert werden. Zum Kompilieren muss die Option `-march=native` angegeben werden, damit alle Instruktionen, die auf der aktuellen Maschine verfügbar sind, genutzt werden können. Wie bereits in Abschnitt 2.2 erklärt, werden SIMD-Intrinsic-Funktionen oft direkt auf die Assembly-Instruktionen abgebildet, bieten jedoch dabei Typsicherheit. In Tabelle 2.1 sind die genutzten Datentypen aufgelistet. Die meisten Bezeichnungen der Intel AVX-Intrinsic-Funktionen entsprechen folgendem Format:

```
_mm256_op_suffix(data_type param1, data_type param2, data_type param3)
```

[1] Eine vollständige Auflistung aller Intel Intrinsic-Funktionen findet man unter `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

Datentyp	Bedeutung
`__m256`	8 × 32-Bit-Fließkommazahlen
`__m256d`	4 × 64-Bit-Fließkommazahlen
`__m256i`	256 Bit an Ganzzahlen (ohne expliziter Unterscheidung der Größe der Datenelemente)
`__m128`	4 × 32-Bit-Fließkommazahlen
`__m128d`	2 × 64-Bit-Fließkommazahlen
`__m128i`	128 Bit an Ganzzahlen (ohne expliziter Unterscheidung der Größe der Datenelemente)

Tabelle 2.1: Datentypen bei Verwendung von AVX-Intrinsic-Funktionen

Datentyp	Bedeutung
`p[s/d]`	Gleitkommazahl mit einfacher oder doppelter Genauigkeit
`ep[i/u]NNN`	Ganzzahl mit oder ohne Vorzeichen mit `NNN` Bits, mit `NNN` = 128, 64, 32, 16 oder 8
`siNNN`	Skalare `NNN`-Bit-Ganzzahl, mit `NNN` = 256, 128 oder 32 (genutzt bei Operationen, wo Größe der Datenelemente irrelevant)

Tabelle 2.2: Intel AVX Suffixkennzeichen

Das Präfix `_mm256` steht dafür, dass auf den neuen 256-Bit-Registern operiert wird. `_op` zeigt die auszuführende Operation an (beispielsweise `add` für eine Addition oder `and` für ein logisches UND). `_suffix` bezeichnet den Datentyp der Elemente, auf denen operiert wird. Die ersten Buchstaben des Suffix stehen für packed (`p`), extended packed (`ep`) oder scalar (`s`) [19, S. 8]. Die Möglichkeiten für die restlichen Buchstaben sind in Tabelle 2.2 aufgelistet. Die Parameter entsprechen bei AVX meist den in Tabelle 2.1 aufgezählten Datentypen.

2.2.2 ARM NEON

Mit der ARMv6-Architektur wurden erstmalig SIMD-Instruktionen auf ARM-Prozessoren eingeführt. Diese operieren auf den Standard-32-Bit-Registern dieser Architektur und erlaubten mehrere 16-Bit- oder 8-Bit-Werte in eines dieser Register zu befüllen und darauf verschiedene Operationen auszuführen [1].

Erst mit der ARMv7-Architektur wurden dann auch breitere Register eingeführt, wodurch dann auch Operationen auf 32-Bit-Datenelementen vektorisiert werden konnten. SIMD-Instruktionen konnten so auf 64-Bit-Registern (D, doubleword) und auf 128-Bit-Registern (Q, quadword) ausgeführt werden. Diese Vektorerweiterung für ARM-Prozessoren wurde NEON genannt [1]. NEON unterstützt Ganzzahlen mit und ohne Vorzeichen von 64-Bit,

Suffix	Bedeutung
`floatMMxN_t`	Gleitkommazahl mit doppelter oder einfacher Genauigkeit mit `MM` Bits (`MM` = 64 oder 32)
`[u]intMMxNN_t`	Ganzzahl mit oder ohne Vorzeichen mit `NN` Bits (`NN` = 64, 32, 16 oder 8)

Tabelle 2.3: ARM NEON Datentypen

Suffix	Bedeutung
`f`*mm*	Gleitkommazahl mit doppelter oder einfacher Genauigkeit mit *mm* Bits (*mm* = 64 oder 32)
`[i/u]`*nn*	Ganzzahl mit oder ohne Vorzeichen mit *nnn* Bits (*nnn* = 64, 32, 16 oder 8)

Tabelle 2.4: ARM NEON Suffixkennzeichen

32-Bit, 16-Bit und 8-Bit Größe. Zudem kommen noch 32-Bit-Fließkommazahlen und 16-Bit- und 8-Bit-Polynome hinzu. Unterstützung für 64-Bit-Fließkommazahlen wurden erst mit der 2012 erschienenen ARMv8-A-Architektur eingeführt [2].

NEON kann 32 64-Bit-Register verwenden. Diese können einmal als 16 128-Bit-Register angesehen werden (Q0 - Q15) oder als 32 64-Bit-Register (D0 - D31). Während Intels AVX viele Instruktionen lediglich für Datenelemente von 32 Bit oder 64 Bit Breite anbietet, so hat NEON für die meisten Funktionen Varianten für alle möglichen Breiten von 8 Bit bis 64 Bit.

Die bei der Programmierung mit Intrinsics verfügbaren Datentypen entsprechen der Form `data_typeNNxMM_t` und sind in Tabelle 2.3 abgebildet. `data_type` steht für `int`, `uint` oder `float`(Die vorliegende Arbeit beschränkt sich auf diese drei Datentypen), `NN` für die Anzahl an Bits der Datenelemente (64 Bit, 32 Bit, 16 Bit, 8 Bit) und `MM` für die Anzahl von Datenelementen in dem entsprechenden Vektorregister.

```
voperation[q]_suffix(data_type param)
```

Jede NEON-Intrinsic-Funktion beginnt mit `v`. Daraufhin folgt die Operation mit `operation`. Das Zeichen `q` ist optional und zeigt an, ob auf 128-Bit-Registern oder auf 64-Bit-Registern gearbeitet wird. Ist dieses vorhanden, so arbeitet man mit einem *Quadword*, also einem 128-Bit-Register. Zuletzt wird `suffix`, wie in Tabelle 2.4 angezeigt, durch den entsprechenden Datentyp mit der Breite der Datenelemente ersetzt.

Um mit NEON-Intrinsics programmieren zu können, muss die Headerdatei `arm_neon.h` inkludiert werden. Die einzelnen NEON-Funktionen sind unter `https://developer.arm.com/architectures/instruction-sets/intrinsics/` zu finden. Um C/C++ Code mit

NEON-Intrinsics zu kompilieren, muss mit GCC die Option `-mfpu=neon` angegeben werden [1].

2.2.3 OpenMP

OpenMP ist ein weit verbreiteter Industriestandard, um Code zu parallelisieren. Dabei wird eine Programmierschnittselle angeboten, welche hilft, existierenden C/C++ oder Fortran-Code ohne großen Aufwand zu parallelisieren. Dieser wird von vielen Architekturen verschiedener Hersteller unterstützt und erlaubt so das Schreiben von plattformunabhängigem Code [26]. Die OpenMP-Spezifikation umfasst Compilerdirektiven, Bibliotheksroutinen und Umgebungsvariablen. Mit `gcc` kann man durch die Option `-fopenmp` dem Compiler das Interpretieren der Compilerdirektiven erlauben. Durch Compilerdirektiven kann man den Compiler anweisen, bestimmte Bereiche des Codes zu parallelisieren.

OpenMP SIMD

Anfangs war eine Parallelisierung des Codes mit OpenMP nur auf Threadebene möglich. Seit Version 4.0 ist auch eine Parallelisierung auf Datenebene mit SIMD möglich [25]. Dazu fügt man die Direktive `#pragma omp simd` vor eine zu vektorisierende `for`-Schleife ein. Zusätzlich können OpenMP-Klauseln definiert werden, wodurch dem Compiler Informationen gegeben werden, die dieser durch statisches Analysieren des Codes nicht herausfinden kann. Beispielsweise kann der Programmierer angeben, unter welchen Umständen keine Abhängigkeiten unter den Schleifendurchläufen existieren [16, S. 61]. OpenMP unterstützt den Compiler im Grunde bei der Auto-Vektorisierung, indem der Programmierer Annahmen beispielsweise über das Datenlayout garantiert, die der Compiler selber nicht treffen kann.

Nimmt man beispielsweise an, man hat drei Arrays `A`, `B` und `C` der Länge `N`. Möchte man die Elemente aus `B`, zu denen aus `C` addieren und in `A` speichern, so kann man das mit AVX und NEON folgendermaßen umsetzen:

```
// C++ code mit AVX2
for (int i = 0; i < n; i+=8) {
  __m256 _B = _mm256_loadu_ps(&B[i]);
  __m256 _C = _mm256_loadu_ps(&C[i]);
  __m256 _A = _mm256_add_ps(_B, _C);
  _mm256_storeu_ps(&A[i], _A);
}
```

```
// C++ code mit NEON
for (int i = 0; i < n; i+=4) {
  float32x4_t _B = vld1q_f32(&B[i]);
  float32x4_t _C = vld1q_f32(&C[i]);
  float32x4_t _A = vaddq_f32(_B, _C);
  vst1q_u32(&A[i], _A);
}
```

Im Vergleich dazu nun die Umsetzung mit OpenMP SIMD:

```
// C++ code mit OpenMP
#pragma omp simd
for (int i = 0; i < n; i++) {
        A[i] = B[i] + C[i];
}
```

Wie man sieht, ist das Programmieren mit Intrinsic-Funktionen selbst bei einem Minimalbeispiel aufwändig. Zusätzlich ist Code mit Intrinsic-Funktionen nicht portabel und erfordert Wissen des Programmierers über die spezifische Plattform und deren SIMD-Erweiterung.
Damit OpenMP eine `for`-Schleife vektorisieren kann, muss sich diese in *kanonischer Form* befinden [26, S. 53]. Das bedeutet, dass die Anzahl der Iterationen schon vor Eintritt in die Schleife bekannt sein muss.
Weitere Informationen und Annahmen über die `for`-Schleife können über die OpenMP-Klauseln an den Compiler übergeben werden.
`safelen` beschreibt die sichere Distanz zwischen zwei Iterationen der assoziierten `for`-Schleife. Keine zwei Iterationen, die gleichzeitig ausgeführt werden, dürfen dabei einen höheren Abstand als `safelen` haben.
`simdlen` beschreibt die bevorzugte Anzahl von Iterationen, die gleichzeitig ablaufen sollen. Jede dieser gleichzeitigen Iterationen wird durch eine andere SIMD-Lane durchgeführt.
Die `aligned`-Klausel versichert dem Compiler, dass die Adresse jedes Arrayelements im Speicher an der angegebenen Anzahl an Bytes ausgerichtet ist.
Ist ein Array als `private` markiert, dann bekommt jede SIMD-Lane, die ein Element des Arrays referenziert, ein eigenes Listenelement.
`linear` führt eine lineare Inkrementierung auf einem gegebenen Array aus. Dabei wird der Index zwischen den Iterationen immer um den Wert *linear-step* inkrementiert. Ist *linear-step* nicht definiert, so ist der Wert standardmäßig 1.
Bei Nutzung der `reduction`-Klausel wird ebenso für jedes angegebene Array eine private Kopie für jede SIMD-Lane erstellt. Am Ende der assoziierten `for`-Schleife werden die originalen Arrays mit der durch einen *reduction-identifier* angegebenen Operation aktualisiert.
Durch `collapse` versucht der Compiler n verschachtelte Schleifen in eine einzige Schleife mit einem globalen Adressraum umzuwandeln [12, S. 20].
Zusätzlich gibt es noch weitere Vorgaben zur Nutzung der OpenMP-SIMD-Compilerdirektiven. Alle mit `omp simd` assoziierten Schleifen müssen *perfectly nested* sein. Das bedeutet, dass zwischen zwei verschachtelten Schleifen kein intervenierender Code oder OpenMP-Direktive stehen darf. Funktionen, wie `return`, `break` oder `continue`, die den Kontrollfluss der Funktion verändern, verhindern ebenfalls die Vektorisierung der assoziierten Schleifen [26, S. 75].

Kapitel 3

Verwandte Arbeiten

Es gibt verschiedene Ansätze, um das Traversieren von Ensembles von Entscheidungsbäumen zu beschleunigen. Zunächst werden untersuchte Möglichkeiten beschrieben, die die Geschwindigkeit des Traversierens mithilfe neuer Algorithmen erhöhen möchten.

Asadi *et al.* stellen in [3] gleich mehrere Ansätze vor. Der Algorithmus STRUCT+ optimiert eine klassische Traversierung von Entscheidungsbäumen von der Wurzel bis zum Blatt, indem er das Datenlayout effizient aufbaut. IF-ELSE übersetzt jeden Entscheidungsbaum statisch in eine Struktur von if-then-else-Codeblöcken. Dadurch müssen die Schwellwerte der Knoten nicht mehr aus dem Speicher geladen werden, sondern werden direkt aus dem Code gelesen. Das Problem beider Ansätze ist, dass durch die vielen Verzweigungen im Code viele Kontrollflussabhängigkeiten existieren. Eine Kontrollflussabhängigkeit besteht, wenn eine Instruktion von der Auswertung einer Bedingung abhängig ist, was die Ausführung des Codes verlangsamt. PRED [3] ist ein Algorithmus, der durch Verzweigungen erzeugte Kontrollflussabhängigkeiten in Datenabhängigkeiten umwandelt. Der Code wartet dabei nicht auf die Auswertung einer Bedingung einer Verzweigung, sondern auf das Ergebnis einer Berechnung, um einen Datenzugriff korrekt zu tätigen. VPRED ist die vektorisierte Version von PRED und führt erstmalig eine Parallelisierung auf Datenebene im Kontext der Traversierung von Entscheidungsbäumen ein. In einer weiteren Arbeit [32] untersucht Sharp eine GPU-Implementierung eines Algorithmus zur Traversierung von Random Forests, der ähnlich wie PRED funktioniert.

In einer aktuelleren Arbeit wird der Algorithmus QuickScorer [21, 9] vorgestellt, welcher versucht, die Traversierung von Entscheidungsbäumen cachefreundlicher zu gestalten und weniger Kontrollflussabhängigkeiten bei der Auswertung zu erzeugen. Darauf aufbauend wird in [22] die vektorisierte Variante dieses Algorithmus namens V-QuickScorer vorgestellt. In [36] stellen Ting *et al.* eine weitere vektorisierte Variante namens RapidScorer vor, welche besonders für tiefere Ensembles von Entscheidungsbäumen die Schwächen von V-QuickScorer behebt. Diese Algorithmen werten die Bäume eines Ensembles nicht ein-

zeln aus, sondern traversieren featureweise über das gesamte Ensemble. Da die vorliegende Arbeit auf diesen drei Algorithmen basiert, werden sie in Kapitel 4 genauer beschrieben.
Lettich *et al.* untersuchen in einer weiteren Arbeit [18] die Möglichkeit einer GPU-Implementierung von QuickScorer und einer Variante mit Multithreading.
Die Traversierung von Random Forests auf FPGAs wird von Buschjäger *et al.* in [6] und von Van Essen *et al.* in [34] untersucht. Letztere Arbeit implementiert zudem eine Traversierung von Random Forests mit Multithreading auf CPUs und eine Umsetzung auf GPUs.
Buschjäger *et al.* [5] und Tang *et al.* [33] untersuchen Strategien, bei denen sie das Traversieren von Entscheidungsbäumen an den Cache eines Prozessors anpassen, um die *Locality* während der Ausführung zu maximieren. Locality beschreibt ein Prinzip, nach dem auf kürzlich verwendete Elemente des Speichers oder Elemente, die sich im Speicher in dessen Nähe befinden, wahrscheinlich zeitnah wieder zugegriffen wird. Ein Maximieren der Locality beschleunigt die Ausführung von Code.
Neben diesen Strategien gibt es Ansätze, welche auf Kosten der Genauigkeit der Ergebnisse das Traversieren von Ensembles beschleunigen. Lucchese *et al.* untersuchen in [20] verschiedene Pruningmethoden, mit denen Bäume im Ensemble entfernt und die Gewichte der übrigen angepasst werden. Cambazoglu *et al.* erforschen in [7] eine Möglichkeit, das Traversieren von Dokumenten mit niedrigen Punktzahlen frühzeitig abzubrechen, bevor sie das gesamte Ensemble durchlaufen haben.
Die vorliegende Arbeit knüpft an die Ergebnisse von QuickScorer, V-QuickScorer und RapidScorer an und fügt den Experimenten mit einem ARM-Prozessor und der SIMD-Erweiterung NEON eine neue Ausführungsplattform hinzu. Während die vorherigen Arbeiten stets nur Gradient-Boosted Decision Trees untersuchen, wird in dieser Arbeit auch die Performanz dieser Algorithmen mit der Ensemblemethode Random Forest erforscht.

Kapitel 4

Umsetzung

In diesem Kapitel werden die hier verwendeten Algorithmen vorgestellt und gezeigt, wie diese vektorisiert werden können. In Abschnitt 4.1 wird der QuickScorer-Algorithmus erklärt. In Abschnitt 4.2 und Abschnitt 4.3 werden die Algorithmen V-QuickScorer und RapidScorer erläutert und wie diese mit den SIMD-Erweiterungen AVX und NEON implementiert werden können. In Abschnitt 4.4 wird versucht, V-QuickScorer und RapidScorer plattformunabhängig zu implementieren.

4.1 QuickScorer

Beim QuickScorer-Algorithmus (QS) wird im Gegensatz zum klassischen Ansatz der Traversierung von Ensembles nicht über jeden Baum einzeln, sondern featureweise über das gesamte Ensemble traversiert.
Die erste Änderung ist das Einführen einer neuen Datenstruktur. Für jeden Knoten n_i^h in jedem Baum T_h wird statisch ein Bitvektor erstellt. Dieser hat genau die Länge der Anzahl der Blätter von T_h, wobei jedes Bit repräsentativ für ein Blatt von T_h steht. Die Bits des Bitvektors, welche für die Blätter stehen, wenn man dem linken Pfad von n_i^h folgt, sind Nullen. Die Blätter des rechten Pfades sind Einsen. Zusätzlich sind die Blätter, die von n_i^h nicht mehr erreichbar sind, auch Einsen. Der Knoten n_2 in Abbildung 4.1 bekommt somit den Bitvektor 110011, da die beiden linken Blätter nicht mehr zu erreichen sind und somit durch zwei Einsen dargestellt werden. Zu den mittleren beiden Blättern gelangt man, indem man dem linken Pfad folgt, welche durch zwei Nullen repräsentiert werden. Durch das Folgen des rechten Pfades kommt man bei den letzten beiden Blättern an, die dann wieder durch zwei Einsen dargestellt werden.
Die Nullen in dieser Darstellung stehen in der Bitvektordarstellung für die Blätter, die in einer Traversierung nicht erreicht werden können, sollte der aktuelle Knoten ein falscher Knoten sein. Für das Beispiel bedeutet das: Unabhängig davon, wie die anderen Knoten ausgewertet werden, wird Knoten n_2 als falsch ausgewertet, wodurch man die Blätter l_2

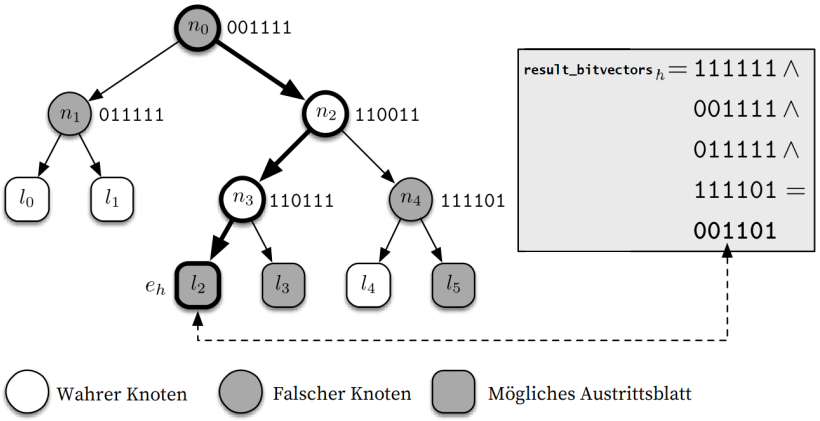

Abbildung 4.1: Beispiel der Traversierung eines Entscheidungsbaumes mit QuickScorer [21]

und l_3 nicht mehr erreichen kann. Die Bitvektoren aller Knoten von T_h werden im Array `bitvectors`$[T_h]$ gespeichert. Durch das Einhalten einer festen Länge der Bitvektoren der Knoten wird verhindert, dass Bits an die richtige Position für die Verundung verschoben werden müssen.

Genau wie für alle Knoten wird nun auch für alle Bäume T_h des Ensembles $\mathcal{T}$ jeweils ein Ergebnisbitvektor erstellt. Die Ergebnisbitvektoren werden in dem Array `result_bitvectors` gespeichert. Ein Ergebnisbitvektor steht repräsentativ für alle Blätter von T_h, die aktuell als Ausgabe des Baumes in Frage kommen. Am Anfang des Algorithmus besteht dieser nur aus Einsen und wird im Verlauf des Algorithmus durch logische Verundungen mit den Bitvektoren der falschen Knoten mehr und mehr mit Nullen gefüllt. Eine solche Verundung der Bitvektoren muss nun für alle falschen Knoten mit den Ergebnisbitvektoren der entsprechenden Bäume geschehen. Die Reihenfolge, in der dies geschieht, ist wegen der Assoziativität der logischen Verundung irrelevant. Als Ergebnis dieses Schrittes bleiben die Ergebnisbitvektoren der einzelnen Bäume über. In dem Beispiel sind n_0, n_1 und n_4 falsche Knoten, weshalb diese mit dem Ergebnisbitvektor des Baumes verundet werden müssen. Diese Berechnung ist rechts in Abbildung 4.1 zu sehen. Dieser Schritt des Algorithmus heißt *Bitvektorberechnungsschritt*.

Die Ergebnisse dieser Berechnungen werden im nächsten Schritt zur Bestimmung der Austrittsblätter von jedem Baum verwendet. Lucchese *et al.* beweisen in [21], dass genau das ganz linke der noch möglichen Austrittsblätter das Austrittsblatt für eine Eingabe ist. Somit muss in den sich ergebenden Ergebnisbitvektoren das ganz linke auf 1 gesetzte Bit gefunden werden, dass dann die Position der korrekten Vorhersage für diesen Baum enthält. Das ganz linke auf 1 gesetzte Bit im Ergebnisbitvektor des Beispiels ist das dritte von links, welches l_2 entspricht. Somit enthält l_2 die korrekte Vorhersage für diesen Baum. Dieser Schritt wird im Algorithmus *Punktzahlberechnungsschritt* genannt.

Um diesen Prozess in einen Algorithmus zu formen, muss sowohl ein Prozess gefunden werden, der alle falschen Knoten des Ensembles möglichst effizient identifiziert als auch ein Datenlayout, dass eine solche Traversierung zulässt. Das Finden der falschen Knoten wird bewerkstelligt, indem alle Knoten nach Feature in aufsteigender Reihenfolge der Schwellwerte sortiert werden. Für jedes Feature f_k wird dann der Test $x_k \leq \gamma_i^h$ ausgewertet. Ist dieser falsch, so ist n_i^h ein falscher Knoten und der Bitvektor des aktuellen Knotens wird mit dem Ergebnisbitvektor des entsprechenden Baums verundet. Daraufhin geht man zum nächsthöheren Schwellwert und testet gegen diesen. Ist der Test hingegen erfolgreich, so müssen alle weiteren Knoten dieses Features nicht weiter getestet werden, da alle weiteren Tests wegen der aufsteigenden Sortierung auch erfolgreich sein werden. Somit kann man direkt zum Testen der Knoten des nächsten Features übergehen.
Da man die Bäume nicht von oben nach unten traversiert, müssen die Knoten des Ensembles auch keine Zeiger zu ihren Kindknoten speichern. Stattdessen speichern Knoten den Schwellwert des Featuretests, die ID des Baumes, zu dem der Knoten gehört, und den Bitvektor des Knotens. Diese Werte werden jedoch nicht in einem `struct` gespeichert, sondern in drei Arrays `thresholds`, `treeids` und `bitvectors`. Dabei gehören für jedes i jeweils `thresholds`$[i]$, `treeids`$[i]$ und `bitvectors`$[i]$ zum selben Knoten. Um nicht das zu testende Feature in jedem Knoten speichern zu müssen, wird ein Hilfsarray `offsets` erstellt, das den Index des ersten Knotens eines Features speichert. Das Array `leafvalues` enthält die Vorhersagen der einzelnen Blätter von jedem Baum im Ensemble.
Somit ergibt sich das in Abbildung 4.2 beschriebene Speicherlayout. Ein Ensemble wird für den QuickScorer-Algorithmus also mit den folgenden Arrays kodiert:

- `thresholds`: Array von Schwellwerten der Knoten, sortiert nach Feature-ID und aufsteigend nach Schwellwert
- `tree_ids`: Array von IDs der den Knoten zugehörigen Bäume, sortiert nach Feature-ID und aufsteigend nach Schwellwert
- `bitvectors`: Array von Bitvektoren der Knoten, sortiert nach Feature-ID und aufsteigend nach Schwellwert
- `offsets`: Hilfsarray, dass die Startindizes der Knoten mit gleicher Feature-ID in den Arrays `thresholds`, `tree_ids` und `bitvectors` markiert
- `result_bitvectors`: Array von Ergebnisbitvektoren der Bäume des Ensembles
- `leafvalues`: Array der aneinandergereihten Vorhersagen der Bäume des Ensembles
- `weights`: Array von Gewichten der Bäume des Ensembles

Algorithmus 1 zeigt den QuickScorer-Algorithmus. Im ersten Schritt wird für jeden Baum jedes Bit der Ergebnisbitvektoren in `result_bitvectors` mit Einsen befüllt. Daraufhin wird über die Schwellwerte jedes Features iteriert (Zeile 4 - 10), bis die Bedingung

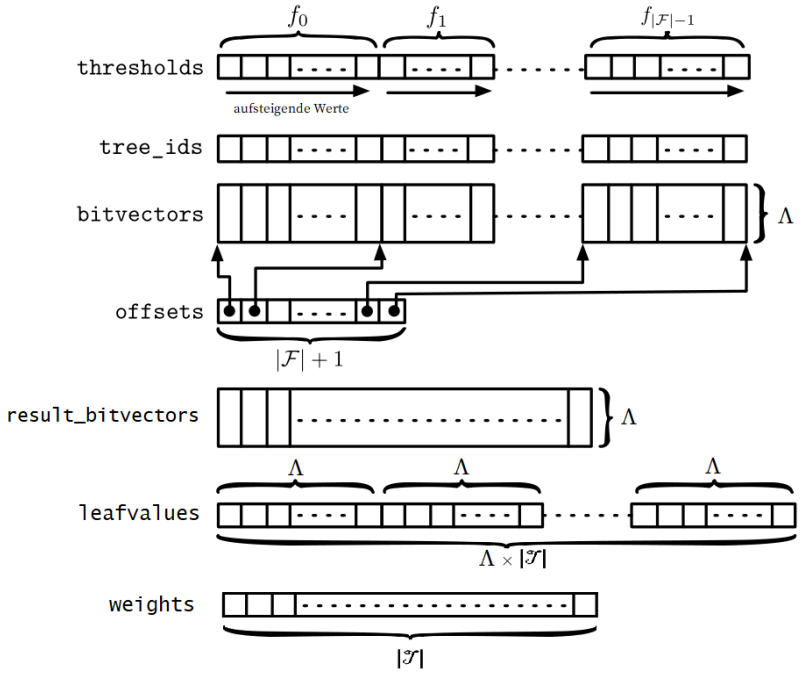

Abbildung 4.2: Datenlayout des QUICKSCORER-Algorithmus [21]

$\texttt{x}[k] \leq \texttt{thresholds}[i]$ erfüllt ist. Bis das der Fall ist, werden die Ergebnisbitvektoren stets mit dem Bitvektor des aktuellen Knoten verundet (Zeile 9). Im letzten Schritt (Zeile 11 - 16) werden dann die Positionen der ganz linken auf 1 gesetzten Bits von jedem Baum bestimmt (Zeile 15), um damit die Vorhersagen von jedem Baum aus dem Array `leafvalues` auszulesen und gewichtet aufzusummieren (Zeile 16 - 17). In der Implementierung werden die Gewichte bereits im Vorhinein mit den Werten des Arrays `leafvalues` multipliziert. Dadurch muss die Multiplikation nicht bei jedem Durchlauf des Algorithmus geschehen. Zum Schluss wird die aufsummierte Punktzahl von jedem Baum zurückgegeben (Zeile 18).

Algorithmus 1 QuickScorer-Algorithmus

```
1:  function QuickScorer(x, 𝒯)
2:      for T_h ∈ 𝒯 do
3:          result_bitvectors[h] ← 11...11
4:      for f_k ∈ ℱ do                                  // Bitvektorberechnungsschritt
5:          i ← offsets[k]
6:          end ← offsets[k + 1]
7:          while x[k] > thresholds[i] do
8:              h ← tree_ids[i]
9:              result_bitvectors[h] ← result_bitvectors[h] ∧ bitvectors[i]
10:             i++
11:             if i ≥ end then
12:                 break
13:     score ← 0                                       // Punktzahlberechnungsschritt
14:     for T_h ∈ 𝒯 do
15:         j ← index of leftmost bit set to 1 of result_bitvectors[h]
16:         l ← h · Λ + j
17:         score ← score + weights[h]· leafvalues[l]
18:     return score
```

4.2 V-QuickScorer

Lucchese *et al.* erweitern QuickScorer um SIMD-Erweiterungen und nennen den daraus entstehenden Algorithmus V-QuickScorer (vQS) [22]. Mit diesem ist es möglich, eine Parallelisierung auf Datenebene zu realisieren. Umgesetzt haben sie den Algorithmus einmal mit SSE 4.2 und einmal mit AVX2. Im Bitvektorberechnungsschritt kann diese Parallelisierung umgesetzt werden, sodass Feature von bis zu acht Dokumenten gleichzeitig gegen einen Schwellwert eines Knotens geprüft werden. Auch die Aktualisierung der Ergebnisbitvektoren ist so vektorisierbar. Ebenfalls kann die Vorhersage für die Dokumente im Punktzahlberechnungsschritt parallel berechnet werden. Um dies umzusetzen, sind nur geringe Änderungen am Datenlayout nötig. Das Array `result_bitvectors` muss für jedes Dokument dupliziert werden. Der Kontrollfluss innerhalb des Algorithmus bleibt somit identisch zu dem von QuickScorer.

Die genaue Implementierung des Algorithmus ist abhängig von der maximalen Anzahl von Blättern im Ensemble und von der genutzten SIMD-Erweiterung, da sie jeweils unterschiedlich breite Register und unterschiedliche Intrinsic-Funktionen zur Verfügung stellen. Ist $\Lambda = 32$ so kann man für die Bitvektoren 32-Bit-Ganzzahlen verwenden. Für $\Lambda = 64$ müssen die Ergebnisbitvektoren und die jeweiligen Bitvektoren der Knoten in zwei Vek-

torregistern verwaltet werden, wodurch ein größerer Verwaltungsaufwand entsteht und die Verundungen in zwei Instruktionen ausgeführt werden müssen. Für V-QuickScorer wird $v = \frac{r}{\lambda}$ als die Anzahl von parallel traversierbaren Dokumenten definiert, wobei r der maximalen Registerbreite und λ der Breite der Featurewerte in Bits (Datentyp `float`) entspricht.

Algorithmus 2 beschreibt den V-QuickScorer-Algorithmus mit Pseudocode. Im ersten Schritt (Zeilen 2 - 3) werden die Ergebnisbitvektoren von jedem Baum mit Einsen befüllt. Da jedes Dokument eigene Ergebnisbitvektoren verwalten muss, geschieht dies für jedes der acht Dokumente. Im Bitvektorberechnungsschritt werden dann die Knoten des Ensembles wie in QuickScorer featureweise durchlaufen. Die `for`-Schleifen in Zeile 4 - 5 stellen dieselbe Struktur dar, wie sie im QuickScorer-Algorithmus genauer beschrieben ist. Jeder Featurewert der Dokumente wird gegen den aktuellen Schwellwert getestet und falls die Tests für jedes Dokument fehlschlagen, geht man zum nächsten Feature über (Zeile 6 - 10). Zeile 6 setzt dabei den Schwellwert, gegen den getestet wird. Zeile 7 lädt die Featurewerte für jedes Dokument in den Vektor $\overrightarrow{x}$[2]. In Zeile 8 wird dann mit diesem Test eine Maske erstellt. Jedes Datenelement des Vektors $\overrightarrow{mask}$, für das der Test erfolgreich war, enthält nur Einsen, alle anderen Datenelemente enthalten nur Nullen. Ist für mindestens eines dieser Dokumente der Test erfolgreich, so enthält $\overrightarrow{mask}$ noch Einsen. Für den Fall, dass $\overrightarrow{mask} = \overrightarrow{0}$ gilt, so ist der Test aus Zeile 8 für alle Dokumente fehlgeschlagen. In diesem Fall wurden alle falschen Knoten für dieses Feature gefunden und man kann zum nächsten Feature übergehen (Zeile 9 - 10). Ist der Test für mindestens ein Dokument erfolgreich, wird in Zeile 11 der Bitvektor des aktuellen Knoten mit dem Index n in $\overrightarrow{m}$ und die Ergebnisbitvektoren der Dokumente in $\overrightarrow{b}$ geladen. Zeile 13 verundet genau diese Vektoren und Zeile 14 speichert diese Werte bedingt durch $\overrightarrow{mask}$ zurück in die Ergebnisbitvektoren im Speicher für die Datenelemente, bei denen $\overrightarrow{mask}$ nur mit Einsen befüllt ist.

Im Punktzahlberechnungsschritt wird zunächst ein Vektor zur Speicherung der Scores mit Nullen initialisiert (Zeile 15).[3] Daraufhin wird über alle Bäume des Ensembles iteriert (Zeile 16) und die Scores von jedem Baum werden für jedes Dokument aufaddiert. Dafür wird skalar der Index des ganz linken auf 1 gesetzten Bits in den Ergebnisbitvektoren der Dokumente gefunden (Zeile 17), womit dann der Index des Austrittsblatts im Array `leafvalues` gefunden wird (Zeile 18). In Zeile 19 wird dann ein Vektor $\overrightarrow{v}$ verwendet, um die Punktzahl des aktuellen Baumes zwischenzuspeichern. In Zeile 20 wird dann dieses Ergebnis mit dem Gewicht `weights`$[\overrightarrow{h}]$ gewichtet zu $\overrightarrow{s}$ addiert. Am Ende wird dieser Vektor $\overrightarrow{s}$ dann zurückgegeben.

[2]Befindet sich über einer Variable ein Pfeil nach rechts, dann stellt diese einen SIMD-Vektor dar.

[3]Nutzt man für die Speicherung der Scores `double`, werden zwei Vektoren benötigt. Für bessere Lesbarkeit wird hier nur ein Vektor initialisiert

Algorithmus 2 V-QuickScorer-Algorithmus

1: **function** V-QuickScorer($\{x_i\}_{i=0,...,v-1}, \mathcal{T}$)
2: **for** $T_h \in \mathcal{T}$ **do**
3: $\forall i = (v-1){:}0$: `result_bitvectors`$[h][i] \leftarrow 11...11$
4: **for** $f_k \in \mathcal{F}$ **do** `// Bitvektorberechnungsschritt`
5: **for** $(\gamma, h, n) \in N_k$ in ascending order **do**
6: $\overrightarrow{\gamma} \leftarrow (\gamma, ..., \gamma)$
7: $\overrightarrow{x} \leftarrow (x_{v-1}[k], ..., x_0[k])$
8: $\overrightarrow{mask} \leftarrow \overrightarrow{x} > \overrightarrow{\gamma}$
9: **if** $\overrightarrow{mask} = 0$ **then**
10: `break`
11: $\overrightarrow{m} \leftarrow$ (`bitvectors`$[n]$,...,`bitvectors`$[n]$)
12: $\overrightarrow{b} \leftarrow$ (`result_bitvectors`$[h][v-1]$,...,`result_bitvectors`$[h][0]$)
13: $\overrightarrow{y} \leftarrow \overrightarrow{m} \wedge \overrightarrow{b}$
14: `result_bitvectors`$[h][(v-1){:}0] \xleftarrow{\overrightarrow{mask}} \overrightarrow{y}$
15: $\overrightarrow{s} \leftarrow (0, ..., 0)$ `// Punktzahlberechnungsschritt`
16: **for** $T_h \in \mathcal{T}$ **do**
17: $\forall i = (v-1){:}0 : j_i \leftarrow$ index of leftmost 1 bit of `result_bitvectors`$[h][i]$
18: $\forall i = (v-1){:}0 : l_i \leftarrow h \cdot \Lambda + j_i$
19: $\overrightarrow{v} \leftarrow$ (`leafvalues`$[l_7]$,...,`leafvalues`$[l_0]$)
20: $\overrightarrow{s} \leftarrow \overrightarrow{s} +$ `weights`$[\overrightarrow{h}] \cdot \overrightarrow{v}$
21: **return** $\overrightarrow{s}$

4.2.1 Vektorisierung mit AVX2

Im Folgenden wird beschrieben, wie der Pseudocode aus Algorithmus 2 mit AVX/AVX2-Intrinsic-Funktionen umgesetzt werden kann. Da AVX 256 Bit breite Register verwendet, können hier acht Dokumente parallel verarbeitet werden. Um in Zeile 6 den Schwellwert γ in ein Vektorregister zu laden, wird die Intrinsic-Funktion `_mm256_set1_ps` benutzt. Um daraufhin in Zeile 7 die Featurewerte für jedes Dokument in $\overrightarrow{x}$ zu laden, kann man die Intrinsic-Funktion `_mm256_set_ps` verwenden. Um die Werte in den Vektoren zu vergleichen (Zeile 8), wird `_mm256_cmp_ps` mit `_CMP_GT_OQ` als dritter Parameter verwendet. Um aus dem Vektor $\overrightarrow{mask}$ einen mit 0 vergleichbaren Wert herauszuziehen (Zeile 9), wird `_mm256_movemask_ps` verwendet. Zum Laden der Bitvektoren des Knoten mit Index n (Zeile 11) wird `_mm256_set1_epi32` verwendet. Da sich die Ergebnisbitvektoren `result_bitvectors` nacheinander im Speicher befinden, können diese mit `_mm256_loadu_si256` in den Vektor geladen werden (Zeile 12). Zum Verunden der zwei Vektoren (Zeile 13) wird `_mm256_and_si256` verwendet. Um die Ergebnisbitvektoren bedingt zu speichern (Zeile 14), wird `_mm256_maskstore_epi32` verwendet. Nutzt man im Punktzahlberechnungsschritt den Datentyp `double` zur Speicherung des Scores, dann benötigt man zwei Vektoren. Diese werden mit `_mm256_setzero_pd` auf 0 initialisiert (Zeile 15). Um dann die Scores aus den Austrittsblättern der einzelnen Bäume zu erhalten (Zeile 19), wird `_mm256_set_pd` verwendet. Zum Aufaddieren von $\overrightarrow{v}$ auf $\overrightarrow{s}$ wird `_mm256_add_pd` genutzt. Zuletzt werden die Werte im Vektor $\overrightarrow{s}$ dann noch in mithilfe von `_mm256_storeu_pd` in den Speicher geschrieben (Zeile 21).

4.2.2 Vektorisierung mit NEON

Algorithmus 2 kann wie folgt mit NEON-Intrinsic-Funktionen umgesetzt werden. Hierbei können wegen der 128 Bit breiten NEON-Register nur vier Dokumente parallel verarbeitet werden. Zeile 6 wird mit der Intrinsic-Funktion `vdupq_n_f32` umgesetzt. Um die Werte der Dokumente für ein bestimmtes Feature zu setzen (Zeile 7), muss man die Werte in ein Array laden und dann `vld1q_f32` aufrufen. GCC erlaubt einem auch die Werte zu laden, indem man sie mit geschweiften Klammern umschlossen auflistet: `{ float e3, float e2, float e1, float e0 }`. Zum Vergleichen dieser beiden Vektoren wird `vcgtq_f32` verwendet. Da die Experimente auf einem Raspberry Pi 3 mit 32-Bit-Architektur ausgeführt werden, kann `vaddvq_u32` nicht verwendet werden, was die einzelnen Werte eines Vektorregisters aufaddieren würde. Um zu erkennen, ob mindestens eines der Register noch eine 1 enthält, extrahiert man stattdessen zwei 64-Bit-Lanes des 128-Bit-Registers mit `vgetq_lane_u64`. Mit `vdupq_n_u32` wird dann in Zeile 11 der Bitvektor des Knoten mit Index n geladen. Da die Ergebnisbitvektoren angrenzend im Speicher liegen, kann für das Laden von `result_bitvectors` in Zeile 12 `vld1q_u32` genutzt werden und zum Verunden der beiden Vektoren (Zeile 13) nutzt man `vandq_u32`. Ein bedingtes Speichern von

Zeile	AVX2 Intrinsic-Funktion
6	`__m256 _mm256_set1_ps(float a)`
7	`__m256 _mm256_set_ps(float e7, ..., float e0)`
8	`__m256 _mm256_cmp_ps(__m256 a, __m256 b, const int imm8)` mit `imm8 = _CMP_GT_OQ`
9	`int _mm256_movemask_ps (__m256 a)`
11	`__m256i _mm256_set1_epi32 (int a)`
12	`__m256i _mm256_loadu_si256 (__m256i const * mem_addr)`
13	`__m256i _mm256_and_si256 (__m256i a, __m256i b)`
14	`void _mm256_maskstore_epi32 (int* mem_addr, __m256i mask, __m256i a)`
15	`__m256d _mm256_setzero_pd (void)`
19	`__m256d _mm256_set_pd(double e3, ..., double e0)`
20	`__m256d _mm256_add_pd(__m256d a, __m256d b)`
21	`void _mm256_storeu_pd(double * mem_addr, __m256d a)`

Tabelle 4.1: Abbildung des Pseudocodes von V-QuickScorer (Algorithmus 2) auf die genutzten AVX2-Instrinsic-Funktionen

den Vektorregistern zu den skalaren Registern wird nicht unterstützt. Deshalb werden mit `vbslq_u32` anhand von $\overrightarrow{mask}$ die Ergebnisse der Verundung bedingt in den Vektor $\overrightarrow{b}$ geladen und dann mit `vst1q_u32` in den Speicher geschrieben (Zeile 14). Da wie erwähnt auf einer 32-Bit-Architektur gearbeitet wird, stehen Vektoren mit 64-Bit-Fließkommazahlen als Datenelementen nicht zur Verfügung. Deshalb wird der Datentyp `float` genutzt, wodurch man zur Speicherung der Punktzahlen der Dokumente auch nur ein Vektorregister benötigt. Zur Initialisierung der Punktzahlen im Punktzahlberechnungsschritt wird `vdupq_n_f32` genutzt. Bei der Programmierung mit NEON kann das ganz linke auf 1 gesetzte Bit in den Ergebnisbitvektoren mit `vclzq_u32` vektorisiert gefunden werden (Zeile 17). Auch Zeile 18 kann mit `vaddq_u32` vektorisiert werden. Diese Indizes müssen dann jedoch mithilfe von `vst1q_u32` wieder in ein Array extrahiert werden, damit man diese Werte als Indizes für das Array `leafvalues` verwenden kann. Für das Laden der Punktzahlen für einen Baum (Zeile 19) wird wieder die in GCC verfügbare Syntax `{ float e3, float e2, float e1, float e0 }` genutzt. Mit `vaddq_f32` wird die Punktzahl aufaddiert (Zeile 20). Um die Punktzahl wieder in ein Array zu speichern, wird letztlich `vst1q_f32` genutzt.

[4]Diese Syntax wird von GCC unterstützt.

Zeile	NEON Intrinsic-Funktion
6	`float32x4_t vdupq_n_f32(float32_t a)`
7	`{ float e3, float e2, float e1, float e0 }`^{4}
8	`uint32x4_t vcgtq_f32(float32x4_t a, float32x4_t b)`
9	`vgetq_lane_u64(vreinterpretq_u64_u32(_mask), 0) > 0` `|| vgetq_lane_u64(vreinterpretq_u64_u32(_mask), 1) > 0`
11	`uint32x4_t vdupq_n_u32(uint32_t value)`
12	`uint32x4_t vld1q_u32(uint32_t const * ptr)`
13	`uint32x4_t vandq_u32(uint32x4_t a, uint32x4_t b)`
14	`uint32x4_t vbslq_u32(uint32x4_t a, uint32x4_t b, uint32x4_t mask)` `void vst1q_u32(uint32_t * ptr, uint32x4_t val)`
15	`float64x2_t vdupq_n_f64(float64_t a)`
17	`uint32x4_t vclzq_u32(uint32x4_t a)`
18	`uint32x4_t vaddq_u32(uint32x4_t a, uint32x4_t b)`
19	`{ float e3, ..., float e0 }`^{4}
20	`float32x4_t vaddq_f32(float32x4_t a, float32x4_t b)`
21	`void vst1q_f32(float32_t * ptr, float32x4_t val)`

Tabelle 4.2: Abbildung des Pseudocodes von V-QuickScorer (Algorithmus 2) auf die genutzten NEON-Instrinsic-Funktionen

4.3 RapidScorer

Der RapidScorer-Algorithmus (RS) [36] passt QuickScorer an, indem er das Datenlayout verändert, die nötigen Operationen anpasst und den Code mit SIMD-Erweiterungen vektorisiert. Insgesamt werden drei große Änderungen gemacht:

1. Einführung einer neuen Datenstruktur für die Bitvektoren der Knoten `epitome`
2. Verschmelzung von äquivalenten Knoten mithilfe der neuen Datenstruktur `eqnode`
3. Speicherung der Ergebnisbitvektoren in einer Struktur namens `ByteTransposition`

Insgesamt können so $v = \frac{r}{\lambda}$ Dokumente gleichzeitig das Ensemble traversieren, wobei r die maximale Registerbreite und λ die minimale Breite eines Datenelements ist. In dieser Implementierung entspricht r 256 (AVX2) oder 128 (NEON) und $\lambda = 8$, also der Länge eines Bytes. Das bedeutet, dass mit AVX2 32 und mit NEON 16 Dokumente gleichzeitig das Ensemble durchlaufen können. Zusätzlich werden die Beispiele x für Blöcke von v Beispielen transponiert, sodass die Werte jedes Features k hintereinander im Speicher liegen, also $x_v^{(k)}, ..., x_v^{(k+1)}$.
Ein Problem von QuickScorer und V-QuickScorer ist, dass sie für jeden Knoten und jeden Baum einen Bitvektor der Länge Λ speichern müssen. Je höher Λ wird, desto ineffizienter werden diese Algorithmen. Besonders ab $\Lambda > 64$ trifft dies zu, da dann mehr Instruktionen zum Verunden der Ergebnisbitvektoren nötig sind. Im RapidScorer-Algorithmus wird zum Speichern der Bitvektoren der Knoten mit `epitome` eine neue Datenstruktur eingeführt. Wie in Abschnitt 4.1 erläutert entspricht der Bitvektor von Knoten n einer Menge von Einsen, (die unerreichbaren Blätter links von n), einer Menge von Nullen (die Blätter des linken Pfades von n), einer Menge von Einsen (die Blätter des rechten Pfades von n) und nochmals einer Menge von Einsen (die unerreichbaren Blätter rechts von n). Die Bitvektoren können somit durch den regulären Ausdruck $E = 1^*0^*1^*$ beschrieben werden. Um diesen zu speichern, muss man sich ausschließlich die Position der ersten und letzten Null im Bitvektor merken. RapidScorer kodiert die Bitvektoren der Knoten in einem `epitome`, welches das erste Byte, dass eine Null enthält (`fb`), die Position dieses Bytes im Bitvektor (`fbp`), das letzte Byte, dass eine Null enthält (`eb`), und die Position dieses Bytes im Bitvektor (`ebp`) speichert. Zusammen ergibt das Quadrupel `ep = {fb,fbp,eb,ebp}` ein `epitome`. So kann der Bitvektor eines Knoten kompakt gespeichert werden, ohne dass der benötigte Speicherplatz linear mit Λ wächst. Algorithmus 3 stellt dar, wie ein Ergebnisbitvektor mit einem Bitvektor eines Knotens in Form eines `epitome`s aktualisiert werden kann. Für den Fall, dass die erste Null und die letzte Null des Bitvektors in dasselbe Byte fallen, gibt es die optimierte Datenstruktur `epitome_short`, bestehend aus dem Tupel `epS = {fb,fbp}`. Der Vorteil ist, dass in diesem Fall nur Zeile 2 von Algorithmus 3 nötig ist, um `result_bitvectors`[h] zu aktualisieren.

Algorithmus 3 Epitome AND

```
1: function EPITOME_AND((ep, result_bitvectors[h]))
2:     result_bitvectors[h][ep.fbp] ← result_bitvectors[h][ep.fbp] ∧ ep.fb
3:     result_bitvectors[h][ep.fbp + 1:ep.ebp] ← 00...00 bytes
4:     result_bitvectors[h][ep.ebp] ← result_bitvectors[h][ep.ebp] ∧ ep.eb
```

Die zweite Änderung ist das Zusammenfassen von Knoten. Knoten im Ensemble werden äquivalent genannt, genau dann, wenn sie denselben Schwellwert und dieselbe Feature-ID enthalten. Bei den vorherigen Algorithmen in der Erkennung der falschen Knoten werden die äquivalenten Knoten redundant öfter getestet. Durch das Zusammenführen der Knoten werden sie nur noch einmal getestet. Dies geschieht mit dem Einführen einer neuen Datenstruktur namens `eqnode = (`γ, u`,treeids,epitomes)`. γ entspricht dem zu testenden Schwellwert, u der Anzahl der in dieser `eqnode` zusammengefassten Knoten, `treeids` einem Array von Baum-IDs der u zusammengefassten Knoten und `epitomes` einem Array von `epitome`s der u zusammengefassten Knoten.
Die dritte Änderung ist die Speicherung der Ergebnisbitvektoren von jedem Baum in dem neuen `ByteTransposition`-Layout. Anstatt wie zuvor bei V-QuickScorer die Ergebnisbitvektoren nacheinander in einem Vektorregister zu speichern, werden diese hier byteweise transponiert und in mehreren Vektorregistern gespeichert. Für 256 Bit breite Register können also 32 Dokumente gleichzeitig verarbeitet werden. Für $\Lambda = 32$ sind $\frac{32}{8} = 4$ Vektorregister nötig. Abbildung 4.4 beschreibt das `ByteTransposition`-Layout. Somit arbeitet man dabei auf Datenelementen von acht Bit Breite.
Insgesamt ergibt sich das in Abbildung 4.3 gezeigte Datenlayout. Ein Ensemble von Entscheidungsbäumen wird beim RapidScorer-Algorithmus also folgendermaßen kodiert:

- `nodes`: Array von `eqnode`s, wobei jedes Element Knoten mit gleicher Feature-ID und Schwellwert zusammenfasst. Die sich ergebenden `eqnode`s werden nach Feature-ID und dann in aufsteigender Reihenfolge der Schwellwerte sortiert.
- `offsets`: Hilfsarray, das Startindizes der Feature in `nodes` markiert
- `result_bitvectors`: Array von Ergebnisbitvektoren der Bäume des Ensembles
- `leafvalues`: Array der aneinandergereihten Vorhersagen der Bäume des Ensembles
- `weights`: Array von Gewichten der Bäume des Ensembles

Der gesamte Ablauf von RapidScorer ist in Algorithmus 4 dargestellt. Zuerst wird jedes Vektorregister von `result_bitvectors` mit Einsen initalisiert (Zeile 2 - 3). Daraufhin wird über alle Feature und alle `eqnode`s dieses Features iteriert (Zeile 4 - 5), um die falschen Knoten zu identifizieren. Zeile 6 lädt wie bei V-QuickScorer den Schwellwert γ des aktuellen Knoten in das Vektorregister $\overrightarrow{\gamma}$ und die Werte des aktuellen Features f_k der zu

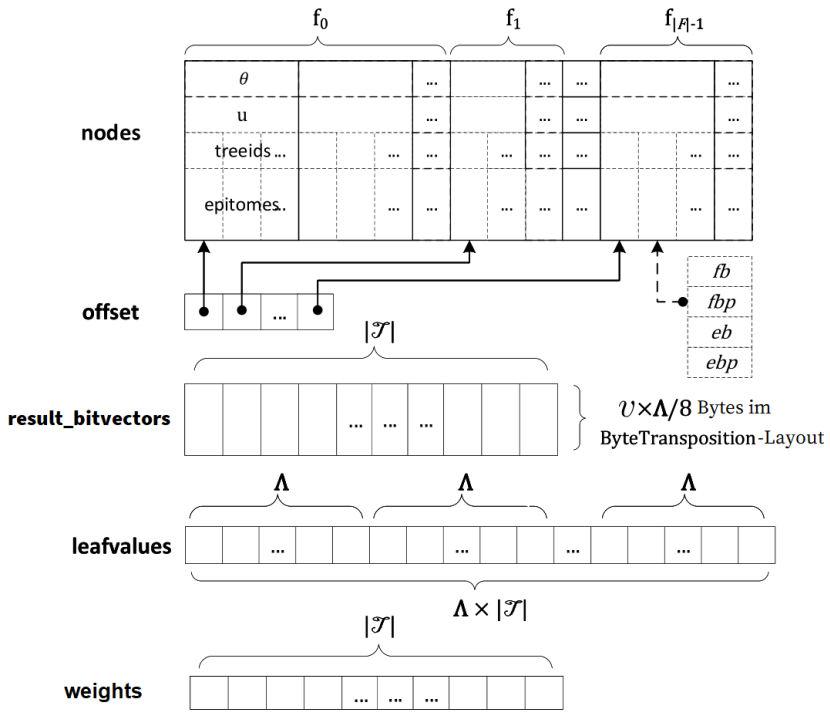

Abbildung 4.3: Datenlayout des RapidScorer-Algorithmus [36]

testenden Dokumente in $\overrightarrow{x}$ (Zeile 7). Der Einfachheit halber stellt $\overrightarrow{x}$ im Pseudocode nur ein einzelnes Vektorregister dar. Jedoch werden für die Speicherung vier dieser Register benötigt, um alle Featurewerte zu speichern, da jedes Vektorregister nur acht `float`-Werte speichern kann. In Zeile 8 wird dann jeweils γ gegen die Featurewerte von jedem x_i getestet und das Ergebnis als Maske in $\overrightarrow{mask}$ gespeichert. Die vier resultierenden $\overrightarrow{mask}$-Vektoren werden dann jeweils auf 8-Bit-Datenelemente verlustfrei reduziert, damit sie in ein Vektorregister gespeichert werden können. Besteht ein Datenelement dann nur aus Einsen, so war der Test $\overrightarrow{x} \leq \overrightarrow{\gamma}$ erfolgreich und für das entsprechende Datenelement ist der aktuelle Knoten ein wahrer Knoten. Besteht ein Datenelement nur aus Nullen, so ist der Test fehlgeschlagen, der aktuelle Knoten ist falsch und für dieses Datenelement muss der Ergebnisbitvektor aktualisiert werden. Sind in jedem Datenelement von $\overrightarrow{mask}$ nur Einsen enthalten, so ist der aktuelle Knoten für jedes Dokument ein wahrer Knoten, der Test in Zeile 9 wird erfolgreich sein und man kann zum Testen des nächsten Feature übergehen (Zeile 10). Ist $\overrightarrow{mask}$ noch für mindestens ein Datenelement 0, dann wird über alle Knoten in der aktuellen `eqnode` iteriert (Zeile 11), die ID des zum Knoten zugehörigen Baum h und das Epitome p des aktuellen Knotens geladen. Mit p, $\overrightarrow{\texttt{result\_bitvectors}[h]}$ und $\overrightarrow{mask}$ wird dann die Funktion Vectorized_AND aufgerufen, die in Algorithmus 5 dargestellt ist und später genauer erklärt wird. Sobald über alle Feature iteriert wurde, werden im Punktzahlberechnungsschritt die Punktzahlen auf 0 initialisiert (Zeile 15). Auch hier

werden mehrere Vektorregister zur Speicherung der Scores benötigt, weil sie nicht in ein Vektorregister passen. Daraufhin wird über alle Bäume des Ensembles iteriert (Zeile 16), sodass die in Algorithmus 6 dargestellte Funktion VECTORIZED_FINDLEAFINDEX dann jeweils den Index des Austrittsblatts für jedes Dokument herausfindet (Zeile 17). Die Funktion wird nachfolgend genauer erläutert. In Zeile 18 wird dann der jeweilige Wert aus dem Array `leafvalues` ausgelesen und auf die Scores in $\overrightarrow{s}$ aufaddiert. Im letzten Schritt wird der Vektor $\overrightarrow{s}$ mit den Punktzahlen zurückgegeben.

Algorithmus 4 RAPIDSCORER-Algorithmus

1: **function** RAPIDSCORER($\{x_i\}_{i=0,...,v-1}, \mathcal{T}$)
2: **for** $T_h \in \mathcal{T}$ **do**
3: $\forall i = (v-1):0$: `result_bitvectors`$[h][i] \leftarrow 11...11$
4: **for** $f_k \in \mathcal{F}$ **do** `// Bitvektorberechnungsschritt`
5: **for** $(\gamma, h, n) \in N_\phi$ in ascending order **do**
6: $\overrightarrow{\gamma} \leftarrow (\gamma, ..., \gamma)$
7: $\overrightarrow{x} \leftarrow (x_{v-1}[k], ..., x_0[k])$
8: $\overrightarrow{mask} \leftarrow \overrightarrow{x} \leq \overrightarrow{\gamma}$
9: **if** $mask \neq \overrightarrow{FF_{hex}}$ **then** $// \; FF_{hex} = 11111111$
10: `break`
11: **for** $q \in 0, 1, ...,$ `nodes`$[n].u - 1$ **do**
12: $h \leftarrow$ `nodes`$[n]$.`treeids`$[q]$
13: $p \leftarrow$ `nodes`$[n]$.`epitomes`$[q]$
14: VECTORIZED_AND($p, \overrightarrow{\texttt{result\_bitvectors}[h]}, \overrightarrow{mask}$)
15: $\overrightarrow{s} \leftarrow (0, ..., 0)$ `// Punktzahlberechnungsschritt`
16: **for** $T_h \in \mathcal{T}$ **do**
17: $\overrightarrow{c} \leftarrow$ VECTORIZED_FINDLEAFINDEX($\overrightarrow{\texttt{result\_bitvectors}[h]}$)
18: $\overrightarrow{s} \leftarrow \overrightarrow{s} +$ `weights`$[\overrightarrow{h}] \cdot$ `leafvalues`$[h][\overrightarrow{c}]$
19: **return** $\overrightarrow{s}$

Algorithmus 5 beschreibt die Funktion VECTORIZED_AND, mit dem die Ergebnisbitvektoren der Dokumente parallel aktualisiert werden können. Die Eingabe sind das `epitome` p des aktuellen Knotens, das Array von Vektorregistern des zum Knoten gehörenden Baum in `ByteTransposition`-Form `result_bitvectors`[h] und das Ergebnis des Featuretests $\overrightarrow{mask}$. Die Attribute `fbp` und `ebp` von p helfen als Index für `result_bitvectors`[h]. Das Byte bei Index `fbp` wird hierbei durch $\overrightarrow{mask}$ bedingt mit `fb` verundet (Zeile 2). Ist die Bedingung p.`fbp` $\neq p$.`ebp` nicht erfüllt, dann ist p ein `epitome_short` und der Algorithmus terminiert (Zeile 3). Anderenfalls wird für alle Indizes zwischen p.`fbp` und p.`ebp` das Array `result_bitvectors`[h] durch $\overrightarrow{mask}$ bedingt auf 0 gesetzt (Zeile 4 - 5). Zum

Schluss wird noch das letzte Byte, das eine Null enthält $p.\texttt{eb}$ durch $\overrightarrow{mask}$ bedingt mit `result_bitvectors`[h] am Index $p.\texttt{ebp}$ verundet (Zeile 6).

Algorithmus 5 VECTORIZED_AND

```
1: function VECTORIZED_AND((p, result_bitvectors[h]⃗, mask⃗))
2:     result_bitvectors[h][p.fbp]⃗ ← result_bitvectors[h][p.fbp]⃗ ∧ mask⃗ ∨ p.fb⃗
3:     if p.fbp ≠ p.ebp then
4:         for n = p.fbp + 1 to p.ebp − 1 do
5:             result_bitvectors[h][n]⃗ ← mask⃗ ∧ result_bitvectors[h][n]⃗
6:         result_bitvectors[h][p.ebp]⃗ ← result_bitvectors[h][p.ebp]⃗ ∧ mask⃗ ∨ p.eb⃗
```

Algorithmus 6 stellt die Funktion VECTORIZED_FINDLEAFINDEX dar. Als Parameter werden die Ergebnisbitvektoren $\overrightarrow{\texttt{result\_bitvectors}[h]}$ des Baumes mit ID h im `ByteTransposition`-Layout übergeben. Zeile 2-3 beschreibt Schritt 3.a, indem zunächst das ganz linke *wahre* Byte (das ganz linke Byte, das eine 1 enthält) jeweils in $\overrightarrow{b}$ und dessen Index in $\overrightarrow{c_1}$ gespeichert wird. Dabei wird durch das Array von `__m256i`-Werten im `ByteTransposition`-Layout iteriert. Falls ein Byte eine 1 enthält, dann wird der Index in $\overrightarrow{c_1}$ und das Byte in $\overrightarrow{b}$ gespeichert. Ähnlich zu Zeile 9 bei RAPIDSCORER wird dieser Schritt in einer Maske festgehalten, damit die Werte nicht nachfolgend überschrieben werden. In Zeile 4 wird dann der Index des ganz linken auf 1 gesetzten Bits in $\overrightarrow{b}$ gefunden, was Schritt 3.b entspricht. In Schritt 3.c werden dann die Indizes in $\overrightarrow{c_1}$ mit 8 multipliziert und die Indizes aus $\overrightarrow{c_2}$ werden darauf addiert und in $\overrightarrow{c}$ gespeichert. Insgesamt bekommt man so die Indizes der ganz linken auf 1 gesetzten Bits in den Ergebnisbitvektoren heraus, welcher in Zeile 6 dann zurückgegeben wird.

Algorithmus 6 VECTORIZED_FINDLEAFINDEX

```
1: function VECTORIZED_FINDLEAFINDEX(result_bitvectors[h]⃗)
2:     b⃗ ← ganz linke wahre Bytes in result_bitvectors[h]⃗          // Schritt 3.a
3:     c1⃗ ← Index des ganz linken wahren Bytes in result_bitvectors[h]⃗
4:     c2⃗ ← Index des ganz linken auf 1 gesetzten Bits in b⃗        // Schritt 3.b
5:     c⃗ ← c2⃗ + 8 · c1⃗                                           // Schritt 3.c
6:     return c⃗
```

4.3.1 Vektorisierung mit AVX2

Die Implementierung des RAPIDSCORER-Algorithmus mit AVX2 gestaltet sich insgesamt aufwendig, da viele Operationen keine 8-Bit-Datenelemente unterstützen. Diese müssen häufig durch eine Kombination von Intrinsic-Funktionen eigens implementiert werden. Da AVX 256 breite SIMD-Register nutzt, können 32 Dokumente gleichzeitig verarbei-

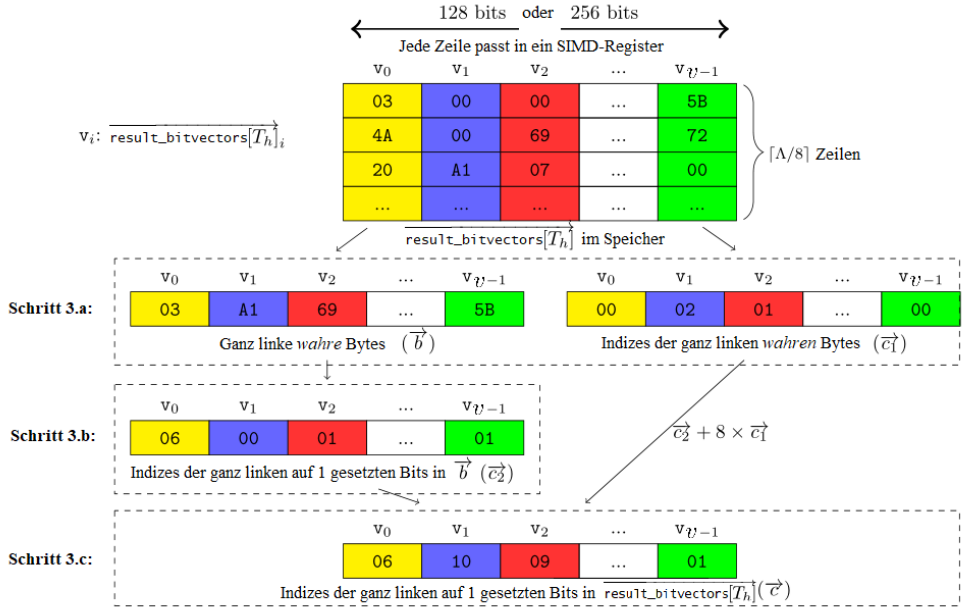

Abbildung 4.4: Vektorisierung des Findens der ganz linken auf 1 gesetzten Bits im `ByteTransposition`-Layout [36]

tet werden. Um den Schwellwert des aktuellen Knotens in $\overrightarrow{\gamma}$ zu laden (Zeile 6), wird `_mm256_set1_ps` genutzt. Für die Werte der 32 Dokumente für Feature f_k lädt man diese mit `_mm256_set_ps` in $\overrightarrow{x}$ (Zeile 7). Den Vergleich dieser Werte (Zeile 8), setzt man mit `_mm256_cmp_ps` um. Der Vergleichsoperator wurde in dieser Implementierung umgedreht, da es die weiteren Operationen einfacher macht. Um $\overrightarrow{mask}$ in nur ein AVX-Register zu speichern, müssen die Register mit 32-Bit-Datenelementen zu einem Register mit 8-Bit-Datenelementen umgewandelt werden. Dazu werden `_mm256_packs_epi32` und `_mm256_packs_epi16` genutzt und zusätzlich `_mm256_permutevar8x32_epi32` um die Datenelemente in der korrekten Reihenfolge abzuspeichern. Ob der Test für alle x_i erfolgreich war, kann dann mit `_mm256_movemask_epi8` ausgelesen werden (Zeile 9). Da die Bedingung des Tests in Zeile 9 invertiert ist, muss $\overrightarrow{mask}$ ebenfalls invertiert werden, damit die Funktion VECTORIZED_AND korrekt die Ergebnisbitvektoren berechnen kann. Das kann mit `_mm256_xor_si256(`$\overrightarrow{mask}$`, _mm256_set1_epi8(-1))` realisiert werden. Die Zeilen 2, 5 und 6 können dann mit `_mm256_and_si256` und `_mm256_or_si256` realisiert werden. Die Initialisierung der Punktzahlen (Zeile 15) wird mit `_mm256_setzero_pd` umgesetzt. Für Schritt 3.a des VECTORIZED_FINDLEAFINDEX-Algorithmus (Algorithmus 6) wird durch das Array von `__m256i`-Werten im `ByteTransposition`-Layout iteriert und jeweils mit `_mm256_cmpeq_epi8` überprüft, ob das Datenelement ungleich 0 ist. Ist das der Fall, wird das Byte und der Index durch eine Maske bedingt gespeichert. Die Maske wird stets mit `_mm256_and_si256` aktualisiert. Das Speichern von Index und Byte geschieht mithilfe von

<table>
<tr><th>Zeile</th><th>AVX2 Intrinsic-Funktion</th></tr>
<tr><td>6</td><td><code>__m256 _mm256_set1_ps(float a)</code></td></tr>
<tr><td>7</td><td><code>__m256 _mm256_set_ps(float e7, ..., float e0)</code></td></tr>
<tr><td>8</td><td><code>__m256 _mm256_cmp_ps(__m256 a, __m256 b, const int imm8)</code>
mit <code>imm8 = _CMP_GT_OQ</code>
<code>__m256i _mm256_packs_epi32 (__m256i a, __m256i b)</code>
<code>__m256i _mm256_packs_epi16 (__m256i a, __m256i b)</code>
<code>__m256i _mm256_permutevar8x32_epi32 (__m256i a, __m256i idx)</code></td></tr>
<tr><td>9</td><td><code>int _mm256_movemask_epi8 (__m256i a)</code></td></tr>
<tr><td>14</td><td>Vectorized_AND:</td></tr>
<tr><td></td><td><code>__m256i _mm256_and_si256 (__m256i a, __m256i b)</code>
<code>__m256i _mm256_or_si256 (__m256i a, __m256i b)</code></td></tr>
<tr><td>15</td><td><code>__m256d _mm256_setzero_pd (void)</code></td></tr>
<tr><td>17</td><td>Vectorized_FindLeafindex:
<table>
<tr><th>Zeile</th><th>Intrinsic-Funktion</th></tr>
<tr><td>2-3</td><td><code>__m256i _mm256_cmpeq_epi8 (__m256i a, __m256i b)</code></td></tr>
<tr><td></td><td><code>__m256i _mm256_and_si256 (__m256i a, __m256i b)</code></td></tr>
<tr><td></td><td><code>__m256i _mm256_blendv_epi8 (__m256i a, __m256i b, __m256i mask)</code></td></tr>
<tr><td>4</td><td><code>__m256i _mm256_shuffle_epi8 (__m256i a, __m256i b)</code></td></tr>
<tr><td>5</td><td><code>__m256i _mm256_slli_epi32 (__m256i a, int imm8)</code></td></tr>
<tr><td></td><td><code>__m256i _mm256_add_epi8 (__m256i a, __m256i b)</code></td></tr>
</table></td></tr>
<tr><td>18</td><td><code>__m256d _mm256_set_pd(double e3, ..., double e0)</code>
<code>__m256d _mm256_add_pd(__m256d a, __m256d b)</code></td></tr>
<tr><td>19</td><td><code>void _mm256_storeu_pd(double * mem_addr, __m256d a)</code></td></tr>
</table>

Tabelle 4.3: Abbildung des Pseudocodes von RapidScorer (Algorithmus 4), Vectorized_AND (Algorithmus 5) und Vectorized_FindLeafindex (Algorithmus 6) auf die genutzten AVX2-Instrinsic-Funktionen

`_mm256_blendv_epi8`. Schritt 3.b kann mit geschicktem Einsatz von `_mm256_shuffle_epi8` und einer Lookup-Tabelle umgesetzt werden. Für die Multiplikation mit 8 in Schritt 3.c wird `_mm256_slli_epi32` genutzt, um die Werte in $\overrightarrow{c_1}$ um 3 Stellen nach links zu verschieben und für die Addition wird `_mm256_add_epi8` verwendet. Um die Punktzahlen aufzuaddieren (Zeile 18) wird `_mm256_set_pd` verwendet, um die Vorhersagen der Blätter in ein AVX-Register zu laden. Mit `_mm256_add_pd` können diese dann zu $\overrightarrow{s}$ addiert werden. Mit `_mm256_storeu_pd` werden dann die Punktzahlen aus den Registern in ein Array gespeichert (Zeile 19).

4.3.2 Vektorisierung mit NEON

Da NEON 128 Bit breite Register zur Verfügung stellt, können hier 16 Dokumente gleichzeitig verarbeitet werden. NEON unterstützt auch 8-Bit-Datenelemente für die meisten Operationen, weshalb sich die Implementierung des RapidScorer-Algorithmus damit einfacher gestaltet, als mit AVX2. Das Setzen des Schwellwerts γ für den aktuellen Knoten (Zeile 6) ist mit `vdupq_n_f32` umgesetzt. Das Laden der Featurewerte für die 16 Dokumente nutzt die in GCC erlaubte Syntax, die Werte mit geschweiften Klammern zu umschließen (Zeile 7). Für den Test in Zeile 8 wird `vcgtq_f32` verwendet. Auch hier ist der Test in der Implementierung invertiert, da sich die Implementierung einfacher gestaltet. Das Ergebnis wird dann mit `vmovn_u32`, `vcombine_u16`, `vmovn_u16` und `vcombine_u8` auf 8-Bit-Datenelemente komprimiert, damit es in ein Vektorregister passt. Die Bedingung in Zeile 9 kann mit zwei Aufrufen von `vgetq_lane_u64` umgesetzt werden. Um die Ergebnisbitvektoren korrekt zu aktualisieren, muss vor dem Aufruf von Vectorized_AND auch hier die Inverse von $\overrightarrow{mask}$ berechnet werden, wozu `veorq_u8` verwendet wird. Vectorized_AND selbst wird mit `vandq_u8` und `vorrq_u8` implementiert. Die Register für die Punktzahlberechnung werden mit `vdupq_n_f32` initialisiert (Zeile 15). In Schritt 3.a von Vectorized_FindLeafindex wird mithilfe von `vtstq_u8` geprüft, ob das aktuelle Register `result_bitvectors`$[m]$ Einsen enthält. Mit dem Testergebnis wird eine Maske aktualisiert, anhand derer die Indizes und Bytes in $\overrightarrow{b}$ und $\overrightarrow{c_1}$ gespeichert werden. Dafür wird `vbslq_u8` verwendet. Zum Finden der ersten Bit 1 in $\overrightarrow{b}$ (Schritt 3.b) wird `vclzq_u8` und für das Multiplizieren und Addieren in Schritt 3.c `vmlaq_u8` genutzt. Zeile 18 von RapidScorer wird mit `vaddq_f32` umgesetzt und das Speichern der Punktzahlen von den Vektorregistern zum Speicher (Zeile 19) mit `vst1q_f32`.

4.4 Vektorisierung mit OpenMP

In Abschnitt 2.2.3 wurde beschrieben, wie man C/C++ Code mit OpenMP vektorisieren kann. Im Folgenden wird versucht, dies auf QuickScorer und RapidScorer anzuwenden. Um den RapidScorer-Algorithmus zu vektorisieren, muss zunächst eine nichtvektorisierte Version des Algorithmus implementiert werden. Dabei ist es sinnvoll, stets Datentypen der gewünschten Breite der Datenelemente im Zielvektor zu verwenden. Das Array `result_bitvectors` und die Variablen `b`, `c1`, `c2` und `c` haben deshalb bei RapidScorer den Datentyp `uint8_t`, damit der Compiler auch Datenelemente von acht Bit Breite für die Vektoren verwendet.

Dazu wird die zu vektorisierende `for`-Schleife, die den Aufruf von QuickScorer umschließt, mit dem Pragma `#pragma omp simd` versehen. Zusätzlich wird die `private`-Klausel zu dem Array `result_bitvectors` angewendet, damit jede SIMD-Lane eine eigene Kopie des Arrays bekommt.

<table>
<tr><td>Zeile</td><td>NEON Intrinsic-Funktion</td></tr>
<tr><td>6</td><td><code>float32x4_t vdupq_n_f32(float32_t a)</code></td></tr>
<tr><td>7</td><td><code>float32x4_t vld1q_f32(float32_t const * ptr)</code></td></tr>
<tr><td>8</td><td><code>uint32x4_t vcgtq_f32(float32x4_t a, float32x4_t b)</code></td></tr>
<tr><td>9</td><td><code>uint64_t vgetq_lane_u64(uint64x2_t v, const int lane)</code></td></tr>
<tr><td>13</td><td>VECTORIZED_AND
<code>uint8x16_t vandq_u8(uint8x16_t a, uint8x16_t b)</code>
<code>uint8x16_t vorrq_u32(uint8x16_t a, uint8x16_t b)</code></td></tr>
<tr><td>14</td><td><code>uint32x4_t vbslq_u32(uint32x4_t a, uint32x4_t b, uint32x4_t mask)</code>
<code>void vst1q_u32(uint32_t * ptr, uint32x4_t val)</code></td></tr>
<tr><td>15</td><td><code>{ float e3, float e2, float e1, float e0 }</code>[4]</td></tr>
<tr><td>17</td><td>VECTORIZED_FINDLEAFINDEX
<table>
<tr><td>Zeile</td><td>Intrinsic-Funktion</td></tr>
<tr><td>2-3</td><td><code>uint8x16_t vtstq_u8(uint8x16_t a, uint8x16_t b)</code></td></tr>
<tr><td></td><td><code>uint8x16_t vandq_u8(uint8x16_t a, uint8x16_t b)</code></td></tr>
<tr><td></td><td><code>uint8x16_t vbslq_u8(uint8x16_t a, uint8x16_t b, uint8x16_t c)</code></td></tr>
<tr><td>4</td><td><code>uint8x16_t vclzq_u8(uint8x16_t a)</code></td></tr>
<tr><td>5</td><td><code>uint8x16_t vmlaq_u8(uint8x16_t a, uint8x16_t b, uint8x16_t c)</code></td></tr>
</table></td></tr>
<tr><td>18</td><td><code>float32x4_t vaddq_f32(float32x4_t a, float32x4_t b)</code></td></tr>
<tr><td>19</td><td><code>void vst1q_f32(float32x4_t * ptr, float32x4_t val)</code></td></tr>
</table>

Tabelle 4.4: Abbildung des Pseudocodes von RAPIDSCORER (Algorithmus 4), VECTORIZED_AND (Algorithmus 5) und VECTORIZED_FINDLEAFINDEX (Algorithmus 6) auf die genutzten NEON-Instrinsic-Funktionen

Jedoch scheint OpenMP nicht in der Lage zu sein, die äußerste Schleife zu vektorisieren. Das liegt daran, dass sich innerhalb des Algorithmus mehrere Schleifen auf gleicher Ebene befinden und die mit OpenMP SIMD assoziierten Schleifen dadurch nicht perfectly nested sind. Somit ist es nicht möglich, mehrere Dokumente gleichzeitig das Ensemble traversieren zu lassen, wie es bei V-QuickScorer der Fall ist.
Stattdessen kann man versuchen, die Schleifen innerhalb der Traversierung eines einzelnen Dokuments parallel ablaufen zu lassen. Dazu fügt man vor die Schleifen des Bitvektorberechnungsschritts und des Punktzahlberechnungsschritts die Compilerdirekte `pragma omp simd` ein. In der zweiten Schleife muss zunächst das `break` aus Zeile 12 in Algorithmus 1 entfernen, da der Code keine Änderungen des Kontrollflusses enthalten darf. Dies geschieht, indem die negierte Bedingung von Zeile 11 mit in die Bedingung der Schleife in Zeile 7 aufgenommen wird ($i < end$). Daraufhin fügt man die Klausel `reduction(& : result_bitvectors)` hinzu, damit die Ergebnisse konkurrierender Iterationen zusammengefügt werden können. Hier offenbart sich ein Problem. Nutzt man die Klausel `reduction` mit dem Operator `&`, dann dupliziert der Compiler das Array für jede Lane. Durch den Operator wird jedoch auch der Initialisierungswert definiert, welcher für `&` 0 ist. Das Problem ist, dass dadurch jeder kopierte Ergebnisvektor ein falsches Ergebnis trägt, da er nicht mehr wie vorgesehen mit nur Einsen initialisiert ist. Auch ohne diese Schwierigkeit scheint OpenMP nicht in der Lage zu sein, die innere Schleife in Zeile 7 zu vektorisieren. Die Schleifenausführung soll weiterhin abgebrochen werden, sobald der Featurewert den Schwellwert eines Knoten überschreitet. Somit enthält der Algorithmus weiterhin eine Änderung des Kontrollflusses, was diese Schleife nicht vektorisierbar macht.
Bei der Auto-Vektorisierung von RapidScorer kommt noch eine weitere Schwierigkeit hinzu. Der Algorithmus wechselt oft zwischen Datenelementen von 32 und 8 Bit Breite, was es dem Compiler nochmals erschwert, den Code zu vektorisieren. Weil RapidScorer grundlegend die gleiche Kontrollflussstruktur wie QuickScorer besitzt, scheitert die Auto-Vektorisierung auch bei dem Algorithmus aus denselben genannten Gründen.
Sowohl für die Intel als auch für ARM-Plattform scheint OpenMP nicht in der Lage zu sein, QuickScorer oder die nicht-vektorisierte Variante von RapidScorer auto-vektorisieren zu können.

4.5 Grenzen der Umsetzung

Da die hier implementierten Algorithmen nicht öffentlich verfügbar sind, wurden sie bestmöglich anhand der verfügbaren wissenschaftlichen Arbeiten nachprogrammiert. Für den Fall, dass die Anzahl der Blätter Λ nicht vollständig in ein Register passt, füllen Lucchese *et al.* in [21] das Register rechts mit Nullen auf. In meiner Implementierung habe ich mich dazu entschieden, links mit Nullen aufzufüllen, was die Implementierung im Nachhinein komplizierter gemacht hat. Für den Fall, dass kein Knoten in einem Baum falsch

ist und der Baum weniger als Λ Blätter enthält, speichert Zeile 16 den falschen Index für das Austrittsblatt. Um das auszugleichen, muss das Array `leafvalues` für alle Bäume mit weniger als Λ Blättern mit dem Wert der letzten Vorhersage aufgefüllt werden. Zudem erzeugt es einen Mehraufwand beim Finden des Index der ganz linken auf 1 gesetzten Bits in den Ergebnisbitvektoren. Besonders die Initialisierung der `result_bitvectors` beim RapidScorer-Algorithmus gestaltet sich als Mehraufwand, da diese transponiert im `ByteTransposition`-Layout gespeichert sind.
Die Implementierungen unterstützen Bäume mit maximal 32 Blättern. Deshalb ist es nicht möglich, die Experimente auf Bäumen mit mehr Blättern auszuführen. Die Experimente mit einer höheren Anzahl von Blättern haben das Potenzial, sehr interessante Ergebnisse zu liefern. Denn erst ab Ensembles mit mehr als 64 Blättern wird QuickScorer ineffizienter, da mehr Register zur Speicherung der Ergebnisbitvektoren genutzt werden müssen. Somit müssen diese auch mit mehreren Instruktionen aktualisiert werden. Der RapidScorer-Algorithmus hat dabei den Vorteil, dass seine Effizienz nicht abhängig von der Anzahl der Blätter der Bäume ist. Das würde zukünftige Experimente mit Random Forests in diesem Bereich interessant machen.
Die verwendete Hardware für die ARM-Plattform unterstützt aufgrund ihres 32-Bit-Betriebssystems keine 64-Bit-Fließkommazahlen. Deshalb verwenden die Arrays `weights` und `leafvalues`, die eigentlich vom Typ `double` sind, den Typ `float`.
Bei der Implementierung der `eqnode`s des RapidScorer-Algorithmus wurde `std::vector` für die Listen `treeids` und `epitomes` verwendet. Der Speicher für diese Vektoren wird dabei nicht auf dem Stack sondern auf dem Heap gespeichert und die `eqnode` erhält einen Zeiger an diese Speicherstelle. Das bedeutet, dass die Zugriffe auf die Listen `treeids` und `epitomes` jedes Mal zusätzlich eine Dereferenzierung des Zeigers benötigen. Alternativ wäre es möglich, C-Arrays fester Länge zu verwenden. Dadurch wird die Dereferenzierung des Zeigers umgangen, jedoch benötigt eine solche Implementierung redundant Speicher. Das liegt daran, dass jedes Array dann die Anzahl der maximal verschmolzenen Knoten in einem Ensemble speichern können muss, da die Länge dieser Arrays statisch sein muss.

Kapitel 5

Experimente

In dieser Arbeit werden die Experimente der Arbeit von Ting *et al.* [36] reproduziert, in der RapidScorer vorgestellt wurde. Um die Hypothesen, die in Abschnitt 5.1 formuliert werden, zu überprüfen, werden Laufzeitmessungen auf verschiedenen Datensätzen durchgeführt. Verglichen werden die Implementierungen der Algorithmen QuickScorer (QS), V-QuickScorer (vQS) und RapidScorer (RS). Da sich im Laufe der Arbeit herausgestellt hat, dass OpenMP nicht in der Lage ist, die Algorithmen zu vektorisieren, wurden die Implementierungen mit OpenMP im Folgenden nicht berücksichtigt. In den Experimenten werden QS und die AVX-Implementierungen von vQS und RS auf den Intel-Prozessoren verglichen. Um QS mit den NEON-Implementierungen von vQS und RS zu vergleichen, wird ein ARM-Prozessor verwendet. Die Konfiguration der Experimente wird in Abschnitt 5.2 erläutert. In Abschnitt 5.3 werden die Ergebnisse der Experimente daraufhin vorgestellt und interpretiert.

5.1 Hypothesen

Es gilt zu testen, ob vQS und RS mit der SIMD-Erweiterung NEON denselben Leistungsschub erhalten kann, wie es bei vQS mit AVX2 der Fall ist. Im Punktzahlberechnungsschritt konnte das Finden des ganz linken auf 1 gesetzten Bits im Ergebnisbitvektor vektorisiert werden, was die Ausführung beschleunigen könnte. Die Vermutung ist dennoch, dass der Leistungsaufschwung aufgrund der niedrigeren SIMD-Registerbreite nicht so hoch sein wird wie bei AVX2.

Die Experimente für die SIMD-Erweiterung AVX werden jeweils auf einer Server- und einer Desktop-CPU von Intel ausgeführt. Durch diesen Vergleich soll untersucht werden, inwiefern auch geringe Unterschiede in der Hardware einen Einfluss auf die Performanz der Algorithmen haben können. Die größten Unterschiede sind verschiedene Größen des L2- und L3-Caches sowie die Prozessorgeneration, aus der sie stammen. Vergleicht man die Arbeiten [22] und [36], dann sieht man, dass sich die Ergebnisse der Experimente auf

dem Datensatz MSN leicht unterscheiden. Für Ensembles mit 1000 Bäumen und $\Lambda = 32$ beispielsweise ist vQS(AVX) in [22] 2.4x schneller als QS, während es in [36] bloß das 1.6-fache ist. Ein solcher Unterschied kann durch die unterschiedlichen verwendeten Lernalgorithmen für die Modelle, durch Unterschiede in den Implementierungen oder durch die verschiedene Hardware bedingt sein. In den Experimenten wird dasselbe Modell mit derselben Implementierung genutzt, sodass überprüft werden kann, ob Unterschiede in der Laufzeit durch die Verwendung verschiedener Hardware entstehen. Es wird vermutet, dass die Wahl des Prozessors keinen großen Einfluss auf die Laufzeiten der Algorithmen hat.
Zusätzlich wird in den Experimenten die Leistung der genannten Algorithmen auf Random Forests untersucht. Anstelle von Punktzahlen geben die Bäume hier Klassenwahrscheinlichkeiten zurück. Das führt dazu, dass deutlich mehr Werte im Punktzahlberechnungsschritt zusammengerechnet werden müssen und dadurch dieser Schritt insgesamt mehr Zeit in Anspruch nehmen wird. Das bedeutet auch, dass die Vektorisierung die Algorithmen nochmals schneller machen kann und zwischen den vektorisierten und nicht-vektorisierten Varianten des Algorithmus ein größerer Leistungsschub zu sehen sein könnte. Jedoch ist anzunehmen, dass der Bitvektorberechnungsschritt mehr Zeit in Anspruch nehmen wird als der Punktzahlberechnungsschritt. In letzterem wird für die Klassifikation einmal über alle Bäume iteriert und daraufhin über die verschiedenen Klassen, um die einzelnen Wahrscheinlichkeiten aufzusummieren. Im Bitvektorberechnungsschritt hingegen wird möglicherweise über alle Knoten des gesamten Ensembles iteriert. Zwischen dem Leistungszuwachs der Algorithmen fürs Ranking mit GBDT oder für die Klassifikation mit Random Forests werden deshalb ähnliche Ergebnisse erwartet.

5.2 Konfiguration

Die Experimente bezüglich des Rankings wurden auf dem LtR-Datensatz MSN [29] ausgeführt. Dieser besteht aus Anfrage-Dokument-Paaren mit je 136 Features. Darin enthalten sind 723412 Trainingsbeispiele, 235259 Beispiele zum Validieren und 241512 Beispiele zum Testen. Damit wurden GBDT-Modelle mit 8, 16 und 32 Blättern mit dem Open-Source-Tool XGBoost [8] und der Methode LambdaMART trainiert. Es wurden Ensembles mit 1000, 2000, 5000 und 10000 Bäumen trainiert.
Für die Experimente zur Klassifikation mit Random Forests wurden die Datensätze Magic[5] mit 10 Features, Adult[6] mit 108 Features, EEG[7] mit 14 Features, MNIST[8] mit 784 Features und Fashion[9] mit ebenfalls 784 Features verwendet. Für das Trainieren der Random Forests wurde die freie Software-Bibliothek Scikit-learn von Python verwendet. Es

[5] `https://archive.ics.uci.edu/ml/datasets/magic+gamma+telescope`
[6] `https://archive.ics.uci.edu/ml/datasets/adult`
[7] `https://archive.ics.uci.edu/ml/datasets/eeg+database`
[8] `http://yann.lecun.com/exdb/mnist/`
[9] `https://github.com/zalandoresearch/fashion-mnist`

wird eine 80/20-Aufteilung zwischen Trainings- und Testdatensatz genutzt. Auch hier haben die Modelle 8, 16 und 32 Blätter. Baranauskas *et al.* beweisen empirisch in [27], dass die Performanz von Random Forests durch mehr Bäume gesteigert werden kann. Bei 128 - 256 Bäumen scheint es jedoch keine signifikante Steigerung der Performanz zu geben. Es wurden für die vorliegenden Datensätze keine weiteren Parameterstudien bezüglich der Klassifikationsperformanz durchgeführt, da die Geschwindigkeit der Traversierung das primäre Thema dieser Arbeit ist. Um dennoch realistische Ensembles zu erzeugen, wurden die Random Forests mit 256, 512 und 1024 Bäumen trainiert und getestet.
Alle Implementierungen wurden mit höchster Optimierungsstufe (`-O3`) kompiliert. Die OpenMP-Implementierungen wurden zusätzlich mit `-fopenmp` und `-fopenmp-simd` kompiliert.
Die Experimente wurden auf einem Server mit einem auf 3,30 GHz getakteten Intel Xeon W-2155, einem auf 3,60 GHz getakteten Intel Core i9 9900K und einem Raspberry Pi 3 Mod. B+ mit einem auf 1,4 GHz getakteten ARM Cortex A53 durchgeführt. Beide Intel-Maschinen besitzen 64 GiB RAM und drei Cache-Level. Beim Xeon W-2155 ist der L1-Cache 32 KB, der L2-Cache 1MB und der L3-Cache 13,75 MB groß. Beim i9 9900K ist der L1-Cache ebenfalls 32 KB, der L2-Cache 256kB und der L3-Cache 16 MB groß. Die Kerne haben jeweils einen eigenen L1- und L2-Cache. Der L3-Cache wird sich von den Kernen jeweils geteilt. Der Raspberry Pi 3 besitzt 1 GiB RAM.
Da der verwendete Raspberry Pi eine 32-Bit-Architektur besitzt, sind die Algorithmen lediglich mit 32-Bit-Fließkommazahlen implementiert.

5.3 Evaluation

Im Folgenden werden die Resultate der Experimente beschrieben und analysiert. Die Ergebnisse für das Ranking sind in Tabelle 5.1 aufgelistet. Die Ergebnisse der Experimente bezüglich der Klassifikationsdatensätze sind in den Tabellen A.2 - A.6 im Anhang zu finden. Die Tabellen beschreiben die durchschnittliche Zeit (in μs), die die Implementierungen für das Durchlaufen des jeweiligen Ensembles pro Eingabe benötigen und sind nach Ausführungsplattform, Algorithmus, Ensemblegröße und Anzahl von Blättern (Λ) gruppiert. Der relative Geschwindigkeitszuwachs der schnellsten Implementierung gegenüber den anderen steht zusätzlich in Klammern hinter dem Ergebnis.
Die Genauigkeiten der Klassifikationsmodelle sind in Tabelle A.1 aufgelistet.

5.3.1 Ranking

Für den Intel i9-9900K schneidet RS für $\Lambda = 8$ besser ab als die anderen Algorithmen. Mit steigendem Λ sinkt die Performanz jedoch, sodass er im Fall von 5000 Bäumen und $\Lambda = 32$ genauso schnell wie QS ist. vQS hingegen ist für $\Lambda = 8$ langsamer als RS, für

Plattform	Algorithmus	Λ	Anzahl Bäume			
			1000	2000	5000	10000
Intel i9-9900K	RS(AVX)	8	1.5 (-)	3.4 (-)	8.1 (-)	16.3 (-)
	vQS(AVX)		2.2 (1.5x)	4.4 (1.3x)	11.6 (1.4x)	23.8 (1.5x)
	QS		3.3 (2.2x)	5.3 (1.6x)	14.2 (1.8x)	26.8 (1.6x)
	RS (AVX)	16	3.0 (1.1x)	5.8 (-)	14.7 (-)	28.3 (-)
	vQS(AVX)		2.8 (-)	5.8 (-)	14.7 (-)	30.7 (1.1x)
	QS		4.2 (1.5x)	10.3 (1.8x)	23.6 (1.6x)	46.1 (1.6x)
	RS(AVX)	32	6.7 (1.6x)	13.8 (1.6x)	37.8 (1.6x)	69.6 (1.4x)
	vQS(AVX)		4.1 (-)	8.4 (-)	23.1 (-)	49.2 (-)
	QS		7.9 (1.9x)	16.0 (1.9x)	37.3 (1.6x)	75.9 (1.5x)
Intel Xeon W-2155	RS(AVX)	8	1.6 (1.3x)	3.6 (1.6x)	8.5 (1.4x)	16.5 (1.4x)
	vQS(AVX)		1.2 (-)	2.3 (-)	6.0 (-)	12.2 (-)
	QS		4.1 (3.4x)	6.5 (2.8x)	15.5 (2.6x)	33.9 (2.8x)
	RS(AVX)	16	3.3 (1.9x)	6.3 (1.7x)	14.8 (1.3x)	30.4 (1.5x)
	vQS(AVX)		1.7 (-)	3.7 (-)	11.0 (-)	20.9 (-)
	QS		5.1 (3.0x)	10.3 (2.8x)	26.5 (2.4x)	62.7 (3.0x)
	RS(AVX)	32	7.1 (2.4x)	13.9 (2.0x)	39.4 (2.0x)	88.1 (2.3x)
	vQS(AVX)		3.0 (-)	6.8 (-)	19.6 (-)	37.7 (-)
	QS		8.6 (2.9x)	17.1 (2.5x)	47.0 (2.4x)	95.8 (2.5x)
ARM Cortex A53	RS(NEON)	8	26.4 (-)	62.1 (-)	201.4 (-)	445.0 (-)
	vQS(NEON)		29.7 (1.1x)	67.1 (1.1x)	234.9 (1.2x)	792.4 (1.8x)
	QS		48.3 (1.8x)	84.5 (1.4x)	258.8 (1.3x)	1009.5 (2.3x)
	RS(NEON)	16	67.2 (1.3x)	171.0 (1.4x)	484.3 (-)	1033.2 (-)
	vQS(NEON)		53.6 (-)	126.4 (-)	725.2 (1.5x)	1601.6 (1.6x)
	QS		68.5 (1.3x)	146.4 (1.2x)	773.3 (1.6x)	2512.1 (2.4x)
	RS(NEON)	32	178.7 (1.9x)	425.1 (1.5x)	1157.8 (-)	–
	vQS(NEON)		93.8 (-)	291.8 (-)	1521.4 (1.3x)	–
	QS		122.9 (1.3x)	324.0 (1.1x)	1943.0 (1.7x)	–

Tabelle 5.1: Laufzeit pro Dokument in μs von RapidScorer(RS), V-QuickScorer(vQS) und QuickScorer(QS) auf dem LtR-Datensatz MSN.

Λ	Anzahl Bäume			
	1000	2000	5000	10000
8	27.26%	22.76%	17.47%	13.55%
16	28.28%	23.31%	16.35%	11.1%
32	26.29%	20.22%	11.77%	6.7%

Tabelle 5.2: Anzahl von Knoten vor und nach Verschmelzen äquivalenter Knoten im Datensatz MSN-1

$\Lambda = 16$ gleich schnell und für $\Lambda = 32$ schneller. Für den Intel Xeon W-2155 erhält man für RS und QS ähnliche Ergebnisse. V-QuickScorer scheint hier jedoch schneller zu sein als auf dem Intel i9-9900K. In jedem Experiment ist V-QuickScorer 1.3x - 2.4x schneller als RS und 2.4x - 3.4x schneller als QS. Auf dem ARM Cortex A53 des Raspberry Pi ist RS(NEON) für $\Lambda = 8$ 1.1x - 1.8x schneller als vQS(NEON). Für 1000 und 2000 Bäume ist RS(NEON) sogar langsamer oder so schnell wie QS. Für die Modelle mit 5000 und 10000 Bäumen ist RS(NEON) für jedes Λ schneller als die anderen Algorithmen. Gegenüber vQS(NEON) ist RS(NEON) 1.2x - 1.8x schneller und gegenüber QS sind es 1.3x - 2.4x. Das Modell mit 10000 Bäumen und $\Lambda = 32$ konnte der Raspberry Pi vermutlich wegen der hohen Menge an Daten nicht verarbeiten.

5.3.2 Klassifikation

Für den Datensatz Magic und $\Lambda = 32$ verhalten sich die Intel-Maschinen ähnlich. vQS ist 2.1x - 2.6 schneller als QS, wobei RS ein wenig schneller als QS ist. Auf dem i9-9900K mit $\Lambda = 8$ ist RS ein wenig schneller als vQS und 1.7 - 1.8x schneller als QS. Bei $\Lambda = 16$ ist vQS bereits 1.2x - 1.4x schneller als RS. Für $\Lambda = 32$ ist vQS bereits 1.9x - 2.3x so schnell wie RS und 2.1x - 2.6x so schnell wie QS. Auf dem Xeon-Prozessor ist für $\Lambda = 8$ RS 1.1x - 2.1x so schnell wie vQS und 1.4x - 4.9x mal so schnell wie QS. Für $\Lambda = 16$ und 256 Bäume ist RS 1.3x mal so schnell wie vQS und 1.7x mal so schnell wie QS. Für die Modelle mit $\Lambda = 16$ und 512 und 1024 Bäumen hingegen ist vQS 1.3 - 1.6x schneller als RS und 3.1x - 3.6x schneller als QS. Für $\Lambda = 32$ ist vQS 2.2x - 2.4x schneller als RS und 2.4x - 2.6 mal schneller als QS. Auf dem ARM Cortex A53 ist vQS für alle Λ am schnellsten, 1.1x - 2.4x schneller als RS und 1.3x - 1.5x schneller als QS. Für $\Lambda = 32$ ist QS immer schneller als RS.

Auf dem Datensatz Adult ist RS insgesamt schneller als auf dem Datensatz Magic. Fast für alle Modelle mit $\Lambda = 8$ und $\Lambda = 16$ ist RS schneller oder gleich auf mit den anderen Algorithmen. Für $\Lambda = 32$ ist wiederum vQS in fast allen Fällen am schnellsten. RS ist teilweise langsamer oder so schnell wie QS.

Die Ergebnisse für den Datensatz EEG zeichnen ein ähnliches Bild wie es bei Magic der Fall ist. Für $\Lambda = 8$ ist RS meist ähnlich schnell wie vQS und ein wenig schneller als QS.

Insgesamt ist eine Korrelation zu beobachten, in der die Performanz von RS im Vergleich zu den anderen Algorithmen mit steigendem Λ abnimmt. vQS ist auf dem i9-Prozessor 1.5x - 2.3x mal schneller als QS. Auf dem Xeon-Prozessor sind es 1.4x - 3.8x mal und auf dem ARM-Prozessor 1.4x - 1.5x schneller.

Auf dem Datensatz MNIST verhalten sich die Algorithmen je nach Modell sehr unterschiedlich. Auf dem i9-Prozessor ist RS für $\Lambda = 8$ und $\Lambda = 16$ 1x - 1.2x schneller als vQS. vQS ist dabei nochmals 1.4x - 2.1x schneller als QS. Auf dem Xeon-Prozessor sind sehr unterschiedliche Laufzeiten zu beobachten, bei denen QS teilweise schneller als RS oder vQS ist. Für den ARM-Prozessor ist vQS stets der schnellste Algorithmus mit einem Leistungszuwachs von 1x - 1.5x gegenüber RS und 1.3x - 1.5x gegen über QS.

Auf dem i9-Prozessor für $\Lambda = 8$ und $\Lambda = 16$ ist RS auf dem Datensatz Fashion nur auf einem der Modelle 1.2x schneller als vQS und ansonsten gleichauf und 2x - 2.8x schneller als QS. vQS ist in diesem Bereich 2x - 2.5x schneller als QS. Für die Modelle mit $\Lambda = 32$ ist vQS 1.4x - 1.5x schneller als RS und 2x - 2.5x schneller als QS. Wie für MNIST erhält man hier sehr unterschiedliche Ergebnisse für den Intel Xeon W-2155. vQS für sieben der neun Modelle 1x - 2.3x schneller als RS und über alle Modelle 1.1x - 3.9x schneller als QS. In zwei Fällen ist RS 1.2x - 1.3x schneller als vQS. Auf dem ARM Cortex A53 ist vQS stets der schnellste der Algorithmen und ist 1.1x - 1.6x schneller als RS und 1.3x - 1.6x schneller als QS.

Insgesamt kann man auf den Klassifikationsdatensätzen feststellen, dass RS für $\Lambda = 8$ meistens schneller oder genauso schnell wie vQS und schneller als QS ist. Mit steigender Blätteranzahl sinkt die Performanz von RS jedoch, sodass vQS für $\Lambda = 16$ und $\Lambda = 32$ oft schneller ist. Zwischen der Größe des Ensembles und der Laufzeit der Algorithmen pro Dokument scheint keine erkennbare Korrelation zu bestehen. Für einzelne Modelle kann zudem beobachtet werden, dass QS schneller als einer der beiden vektorisierten Algorithmen ist.

5.3.3 Analyse

Die Hypothese, dass NEON verglichen mit den AVX-Implementierungen nicht effizienter ist, hat sich bewahrheitet. Sowohl vQS hat auf beiden Intel-Prozessoren meist einen stärkeren Leistungszuwachs als die NEON-Implementierung. Dasselbe gilt für die AVX-Implementierung des RapidScorer-Algorithmus. Besonders auf den Klassifikationsdatensätzen ist zu sehen, dass RS mit NEON oft sogar langsamer ist als QS.

Anders als in der Hypothese vermutet kann beobachtet werden, dass der Prozessor einen Leistungsunterschied hervorrufen kann. Erkennbar ist das im Vergleich der Laufzeiten von vQS auf den beiden Intel-Maschinen, besonders für den Rankingdatensatz MSN. Dabei ist vQS auf dem Intel i9-9900K 1.1x - 1.9x mal schneller als QS, während vQS auf dem Intel Xeon W-2155 2.4x - 3.4x schneller als QS ist. Ein möglicher Grund dafür ist die unter-

Datensatz	Λ	Anzahl Bäume		
		256	512	1024
Magic	8	80.05%	70.65%	58.25%
	16	81.54%	71.14%	61.19%
	32	80.94%	73.65%	62.99%
Adult	8	11.71%	7.19%	5.21%
	16	10.36%	7.83%	5.55%
	32	10.18%	7.37%	5.84%
EEG	8	43.72%	33.01%	25.31%
	16	45.3%	35.26%	24.66%
	32	43.65%	32.29%	22.74%
MNIST	8	44.98%	36.22%	28.01%
	16	48.41%	39.65%	32.06%
	32	52.46%	44.11%	36.01%
Fashion	8	71.82%	59.57%	47.98%
	16	74.11%	62.92%	51.73%
	32	75.68%	65.78%	54.53%

Tabelle 5.3: Anzahl von Knoten vor und nach Verschmelzen äquivalenter Knoten in Klassifikationsdatensätzen

schiedliche Größe des L2-Caches der Prozessoren. Auf den Klassifikationsdatensätzen kann dieser Leistungsunterschied von vQS zwischen den Intel-Maschinen jedoch nicht konsistent gefunden werden. Für RS scheint es diesen Unterschied zwischen den Prozessoren nicht zu geben.
Da die Experimente auf den Klassifikationsdatensätzen sehr unterschiedlich ausfallen, ist es schwierig, in den Ergebnissen Strukturen und Muster zu erkennen. Auch die Ergebnisse zwischen den Experimenten mit den Rankingdatensatz und den Klassifikationsdatensätzen sind sehr unterschiedlich ausgefallen und widersprechen der zuvor verfassten Hypothese. Die Ergebnisse zwischen Datensätzen hängen von vielen Eigenschaften der Datensätze ab. Die Tabellen 5.2 und 5.3 stellen die prozentuale Anzahl von Knoten im Ensemble vor und nach Verschmelzen der äquivalenten Knoten dar. Darin ist zu sehen, dass dieser prozentuale Wert für alle Datensätze mit der Anzahl der Bäume sinkt. Mit der Tiefe der Bäume sinkt der Anteil auf dem Rankingdatensatz zudem erst ab 5000 Bäumen signifikant. Auf den Klassifikationsdatensätzen scheint dieser Wert je nach Datensatz eher zu steigen. Das bedeutet, dass dabei eher neue Knoten hinzukommen, als wiederverwendet werden. Dieser Anteil reduziert die Anzahl der benötigten Vergleiche für RS. Vermutlich ist dies jedoch nicht der einzige entscheidende Faktor für die Performanz von RS.
In den Test- und Trainingsdatensätzen von Adult, Fashion und MNIST sieht man, dass einige Featurewerte für viele Beispiele den niedrigsten Wert (0) haben und andere Beispiele höhere Featurewerte besitzen. Durchlaufen solche Beispiele mit einem der vektorisierten Algorithmen parallel das Ensemble, so kann dies zu einer hohen Control Divergence führen, was das Ausführen dieser verlangsamt. Dieser Effekt ist durch die höhere Parallelität für RS nochmals höher als für vQS.
In der Arbeit [36], in der der Algorithmus vorgestellt wurde, konnte RapidScorer in jedem der Experimente bessere Ergebnisse erzielen als die anderen verwendeten Algorithmen. In dieser Arbeit konnten diese Ergebnisse auf demselben Datensatz jedoch nicht reproduziert werden. Das könnte daran liegen, dass die hier verwendete Implementierung wie in Abschnitt 4.5 beschrieben noch optimiert werden kann. Durch die Nutzung von Arrays anstatt `std::vector` könnte die Leistung von RS steigen und somit näher an den Ergebnissen von Ting *et al.* [36] liegen.

Kapitel 6

Fazit

Wegen der in Abschnitt 4.4 genannten Gründe kann OpenMP die Algorithmen weder über mehrere Dokumente noch über mehrere Feature vektorisieren. Somit konnte das in Kapitel 1 formulierte Ziel einer plattformunabhängigen, vektorisierten Implementierung der Algorithmen QuickScorer und RapidScorer nicht umgesetzt werden.

V-QuickScorer und RapidScorer konnten in dieser Arbeit dennoch erfolgreich für ARM-Prozessoren mithilfe der SIMD-Erweiterung NEON implementiert werden. Somit konnte das Teilziel erreicht werden, die Traversierung von Ensembles auf einer Plattform für kleine Geräte zu beschleunigen. Für diese Implementierungen mit Vektorisierung wurde die SIMD-Erweiterung NEON von ARM-Prozessoren genutzt.

Zusätzlich konnten die genannten Algorithmen für Intel-Prozessoren implementiert werden. Die Implementierungen mit AVX und NEON wurden zudem erweitert, sodass auch Random Forests mit Klassenwahrscheinlichkeiten traversiert werden können. Mit den Implementierungen konnten Experimente auf einem Ranking-Datensatz und auf verschiedenen Klassifikationsdatensätzen ausgeführt werden. Die Ergebnisse der Laufzeitmessungen wurden zwischen den Algorithmen und den verschiedenen Ausführungsplattformen verglichen. Die Ergebnisse vorheriger Arbeiten konnten dabei nur teilweise reproduziert werden. V-QuickScorer war in den Experimenten für Ensembles mit 32 Blättern oft schneller als RapidScorer. Gründe für diese Unterschiede liegen möglicherweise in den genannten Implementierungsdetails des RapidScorer-Algorithmus. Es konnte zudem anhand des V-QuickScorer-Algorithmus auf den genutzten Intel-Prozessoren gezeigt werden, dass derselbe Algorithmus auf verschiedener Hardware sehr unterschiedlich performant sein kann.

Diese Arbeit kann in Zukunft um mehrere Aspekte erweitert werden. Eine Implementierung der `eqnodes` mithilfe von Arrays anstelle des aktuell genutzten `std::vector` kann die Ausführung des RapidScorer-Algorithmus nochmals beschleunigen. Die Implementierungen mit NEON können auf einer anderen ARM-Maschine getestet, auf der auch 64-Bit-Fließkommazahlen und weitere SIMD-Instruktionen unterstützt werden.

Die Implementierungen auf der Intel-Plattform können in Zukunft die neue SIMD-Erweiterung AVX-512 nutzen. Mit nochmals breiteren Registern kann der Grad der Parallelisierung gesteigert und mithilfe neuer SIMD-Instruktionen kann die Ausführung effizienter gestaltet werden.

Anhang A

Weitere Informationen

Datensatz	Λ	Anzahl Bäume		
		256	512	1024
Magic	8	80.42%	80.49%	80.49%
	16	83.18%	83.25%	83.46%
	32	84.23%	84.04%	83.99%
Adult	8	79.87%	78.95%	79.41%
	16	83%	83.33%	82.6%
	32	84.17%	84.15%	84.14%
EEG	8	68.86%	68.79%	68.86%
	16	72.63%	72.5%	73.13%
	32	76.03%	76.1%	76.1%
MNIST	8	75.26%	75.29%	75.65%
	16	82.12%	82.32%	81.96%
	32	86.35%	86.26%	86.5%
Fashion	8	68.72%	69.14%	68.99%
	16	74.94%	75.13%	75.09%
	32	77.14%	76.73%	76.58%

Tabelle A.1: Genauigkeit der Klassifikationsdatensätze

Plattform	Algorithmus	Λ	Anzahl Bäume		
			256	512	1024
Intel i9-9900K	RS(AVX)	8	0.7 (-)	1.3 (-)	2.6 (-)
	vQS(AVX)		0.8 (1.1x)	1.5 (1.2x)	2.9 (1.1x)
	QS		1.2 (1.7x)	2.4 (1.8x)	4.7 (1.8x)
	RS(AVX)	16	1.3 (1.4x)	2.5 (1.4x)	4.7 (1.2x)
	vQS(AVX)		0.9 (-)	1.8 (-)	3.9 (-)
	QS		1.9 (2.1x)	3.5 (1.9x)	7.8 (2.0x)
	RS(AVX)	32	2.8 (2.3x)	6.0 (2.1x)	11.5 (1.9x)
	vQS(AVX)		1.2 (-)	2.9 (-)	5.9 (-)
	QS		3.1 (2.6x)	6.4 (2.2x)	12.3 (2.1x)
Intel Xeon W-2155	RS(AVX)	8	1.3 (-)	1.4 (-)	2.7 (-)
	vQS(AVX)		1.8 (1.4x)	1.6 (1.1x)	5.7 (2.1x)
	QS		1.8 (1.4x)	6.8 (4.9x)	8.0 (3.0x)
	RS(AVX)	16	1.3 (-)	2.7 (1.3x)	5.2 (1.6x)
	vQS(AVX)		1.7 (1.3x)	2.1 (-)	3.3 (-)
	QS		2.2 (1.7x)	6.6 (3.1x)	11.8 (3.6x)
	RS(AVX)	32	3.1 (2.2x)	6.5 (2.4x)	11.5 (2.3x)
	vQS(AVX)		1.4 (-)	2.7 (-)	5.0 (-)
	QS		3.4 (2.4x)	6.9 (2.6x)	13.0 (2.6x)
ARM Cortex A53	RS(NEON)	8	12.1 (1.3x)	23.4 (1.2x)	50.0 (1.1x)
	vQS(NEON)		9.6 (-)	19.2 (-)	46.4 (-)
	QS		14.3 (1.5x)	28.1 (1.5x)	65.5 (1.4x)
	RS(NEON)	16	22.1 (1.4x)	60.9 (1.9x)	132.5 (1.7x)
	vQS(NEON)		15.5 (-)	32.2 (-)	76.3 (-)
	QS		23.6 (1.5x)	47.8 (1.5x)	104.1 (1.4x)
	RS(NEON)	32	61.4 (2.3x)	166.3 (2.7x)	352.6 (2.4x)
	vQS(NEON)		27.1 (-)	61.0 (-)	148.2 (-)
	QS		40.4 (1.5x)	80.4 (1.3x)	206.8 (1.4x)

Tabelle A.2: Laufzeit pro Dokument in μs von RapidScorer(RS), V-QuickScorer(vQS) und QuickScorer(QS) auf dem Klassifikationsdatensatz Magic

Plattform	Algorithmus	Λ	Anzahl Bäume		
			256	512	1024
Intel i9-9900K	RS(AVX)	8	0.5 (-)	0.9 (-)	1.9 (-)
	vQS(AVX)		0.8 (1.6x)	1.6 (1.8x)	3.0 (1.6x)
	QS		1.1 (2.2x)	2.0 (2.2x)	3.9 (2.1x)
	RS(AVX)	16	0.8 (-)	1.7 (-)	3.6 (-)
	vQS(AVX)		0.9 (1.1x)	1.8 (1.1x)	3.7 (-)
	QS		1.4 (1.7x)	2.6 (1.5x)	5.8 (1.6x)
	RS(AVX)	32	1.8 (1.6x)	3.7 (1.5x)	7.5 (1.4x)
	vQS(AVX)		1.1 (-)	2.4 (-)	5.2 (-)
	QS		1.9 (1.7x)	3.9 (1.6x)	8.1 (1.6x)
Intel Xeon W-2155	RS(AVX)	8	0.7 (-)	1.1 (-)	1.9 (-)
	vQS(AVX)		2.0 (2.9x)	1.3 (1.2x)	2.0 (1.1x)
	QS		1.5 (2.1x)	2.7 (2.5x)	4.5 (2.4x)
	RS(AVX)	16	0.9 (-)	2.0 (1.3x)	3.5 (1.1x)
	vQS(AVX)		0.9 (-)	1.5 (-)	3.1 (-)
	QS		1.6 (1.8x)	2.9 (1.9x)	5.5 (1.8x)
	RS(AVX)	32	1.8 (1.8x)	3.7 (1.9x)	7.8 (-)
	vQS(AVX)		1.0 (-)	2.0 (-)	9.9 (1.3x)
	QS		2.2 (2.2x)	4.0 (2.0x)	7.7 (-)
ARM Cortex A53	RS(NEON)	8	11.3 (-)	21.7 (-)	42.6 (-)
	vQS(NEON)		11.2 (-)	21.0 (-)	40.8 (-)
	QS		14.3 (1.3x)	26.9 (1.3x)	50.6 (1.2x)
	RS(NEON)	16	18.0 (1.1x)	36.1 (1.2x)	74.9 (1.2x)
	vQS(NEON)		15.9 (-)	30.3 (-)	61.4 (-)
	QS		20.6 (1.3x)	36.1 (1.2x)	75.3 (1.2x)
	RS(NEON)	32	33.0 (1.4x)	66.9 (1.5x)	171.0 (1.7x)
	vQS(NEON)		23.3 (-)	45.3 (-)	100.3 (-)
	QS		29.3 (1.3x)	55.6 (1.2x)	117.5 (1.2x)

Tabelle A.3: Laufzeit pro Dokument in μs von RapidScorer(RS), V-QuickScorer(vQS) und QuickScorer(QS) auf dem Klassifikationsdatensatz Adult

Plattform	Algorithmus	Λ	Anzahl Bäume		
			256	512	1024
Intel i9-9900K	RS(AVX)	8	0.6 (-)	1.1 (-)	2.1 (-)
	vQS(AVX)		0.8 (1.3x)	1.4 (1.3x)	3.0 (1.4x)
	QS		1.2 (2.0x)	2.2 (2.0x)	4.4 (2.1x)
	RS(AVX)	16	1.1 (1.2x)	2.2 (1.2x)	4.2 (1.1x)
	vQS(AVX)		0.9 (-)	1.8 (-)	3.7 (-)
	QS		1.6 (1.8x)	3.3 (1.8x)	7.0 (1.9x)
	RS(AVX)	32	2.9 (2.6x)	5.1 (1.9x)	9.4 (1.8x)
	vQS(AVX)		1.1 (-)	2.7 (-)	5.3 (-)
	QS		2.5 (2.3x)	5.5 (2.0x)	11.2 (2.1x)
Intel Xeon W-2155	RS(AVX)	8	0.6 (-)	2.2 (-)	2.3 (-)
	vQS(AVX)		0.6 (-)	2.2 (-)	2.4 (-)
	QS		1.7 (2.8x)	3.3 (1.5x)	8.7 (3.8x)
	RS(AVX)	16	1.2 (-)	2.4 (-)	4.2 (1.3x)
	vQS(AVX)		2.2 (1.8x)	2.4 (-)	3.2 (-)
	QS		4.0 (3.3x)	3.7 (1.5x)	7.1 (2.2x)
	RS(AVX)	32	2.5 (2.1x)	5.1 (2.2x)	9.4 (-)
	vQS(AVX)		1.2 (-)	2.3 (-)	9.8 (-)
	QS		3.0 (2.5x)	5.8 (2.5x)	12.7 (1.4x)
ARM Cortex A53	RS(NEON)	8	12.5 (1.2x)	24.1 (1.1x)	47.6 (1.1x)
	vQS(NEON)		10.6 (-)	21.4 (-)	43.1 (-)
	QS		15.4 (1.5x)	30.5 (1.4x)	59.6 (1.4x)
	RS(NEON)	16	22.4 (1.3x)	43.5 (1.3x)	100.9 (1.4x)
	vQS(NEON)		16.8 (-)	34.0 (-)	69.7 (-)
	QS		25.2 (1.5x)	48.9 (1.4x)	100.5 (1.4x)
	RS(NEON)	32	44.3 (1.6x)	121.8 (2.1x)	276.8 (2.1x)
	vQS(NEON)		28.3 (-)	58.4 (-)	133.5 (-)
	QS		41.7 (1.5x)	85.1 (1.5x)	181.2 (1.4x)

Tabelle A.4: Laufzeit pro Dokument in μs von RapidScorer(RS), V-QuickScorer(vQS) und QuickScorer(QS) auf dem Klassifikationsdatensatz EEG

Plattform	Algorithmus	Λ	Anzahl Bäume		
			256	512	1024
Intel i9-9900K	RS(AVX)	8	1.3 (-)	2.5 (-)	4.8 (-)
	vQS(AVX)		1.6 (1.2x)	2.7 (1.1x)	5.2 (1.1x)
	QS		3.2 (2.5x)	5.3 (2.1x)	10.4 (2.2x)
	RS(AVX)	16	1.9 (-)	3.5 (-)	6.7 (-)
	vQS(AVX)		2.0 (1.1x)	3.6 (-)	7.5 (1.1x)
	QS		4.2 (2.2x)	6.7 (1.9x)	12.0 (1.8x)
	RS(AVX)	32	3.3 (1.2x)	6.5 (1.2x)	12.7 (1.2x)
	vQS(AVX)		2.7 (-)	5.5 (-)	11.0 (-)
	QS		5.4 (2.0x)	9.4 (1.7x)	15.6 (1.4x)
Intel Xeon W-2155	RS(AVX)	8	1.5 (-)	6.3 (2.4x)	4.9 (1.1x)
	vQS(AVX)		1.7 (1.1x)	2.6 (-)	4.6 (-)
	QS		8.2 (5.5x)	4.8 (1.8x)	12.3 (2.7x)
	RS(AVX)	16	3.5 (-)	4.5 (-)	7.1 (-)
	vQS(AVX)		5.9 (1.7x)	6.6 (1.5x)	8.0 (1.1x)
	QS		4.6 (1.3x)	6.5 (1.4x)	12.6 (1.8x)
	RS(AVX)	32	5.1 (-)	7.7 (1.4x)	13.9 (1.1x)
	vQS(AVX)		6.6 (1.3x)	5.6 (-)	12.1 (-)
	QS		5.5 (1.1x)	8.8 (1.6x)	15.6 (1.3x)
ARM Cortex A53	RS(NEON)	8	37.6 (1.1x)	63.6 (-)	154.7 (1.3x)
	vQS(NEON)		35.0 (-)	61.0 (-)	123.0 (-)
	QS		52.4 (1.5x)	91.2 (1.5x)	182.8 (1.5x)
	RS(NEON)	16	49.4 (1.1x)	111.4 (1.3x)	246.8 (-)
	vQS(NEON)		44.4 (-)	88.8 (-)	250.1 (-)
	QS		66.3 (1.5x)	130.3 (1.5x)	345.5 (1.4x)
	RS(NEON)	32	103.9 (1.5x)	228.2 (1.4x)	464.9 (1.1x)
	vQS(NEON)		68.8 (-)	161.0 (-)	409.2 (-)
	QS		92.8 (1.3x)	212.3 (1.3x)	513.6 (1.3x)

Tabelle A.5: Laufzeit pro Dokument in μs von RapidScorer(RS), V-QuickScorer(vQS) und QuickScorer(QS) auf dem Klassifikationsdatensatz MNIST

Plattform	Algorithmus	Λ	Anzahl Bäume		
			256	512	1024
Intel i9-9900K	RS(AVX)	8	1.4 (-)	2.8 (-)	5.4 (-)
	vQS(AVX)		1.7 (1.2x)	2.8 (-)	5.6 (-)
	QS		3.9 (2.8x)	6.3 (2.2x)	11.4 (2.1x)
	RS(AVX)	16	2.1 (-)	4.0 (-)	7.5 (-)
	vQS(AVX)		2.2 (-)	4.1 (-)	7.2 (-)
	QS		5.3 (2.5x)	8.5 (2.1x)	14.1 (2.0x)
	RS(AVX)	32	3.8 (1.4x)	7.5 (1.5x)	14.1 (1.4x)
	vQS(AVX)		2.8 (-)	5.1 (-)	10.1 (-)
	QS		7.0 (2.5x)	11.9 (2.3x)	19.7 (2.0x)
Intel Xeon W-2155	RS(AVX)	8	2.9 (-)	5.0 (2.1x)	5.9 (1.3x)
	vQS(AVX)		3.9 (1.3x)	2.4 (-)	4.7 (-)
	QS		4.2 (1.4x)	8.1 (3.4x)	9.9 (2.1x)
	RS(AVX)	16	2.4 (1.3x)	4.8 (1.4x)	8.0 (-)
	vQS(AVX)		1.8 (-)	3.5 (-)	8.0 (-)
	QS		7.0 (3.9x)	10.2 (2.9x)	15.2 (1.9x)
	RS(AVX)	32	6.5 (2.3x)	8.6 (-)	15.5 (1.3x)
	vQS(AVX)		2.8 (-)	10.1 (1.2x)	12.3 (-)
	QS		8.1 (2.9x)	11.7 (1.4x)	20.1 (1.6x)
ARM Cortex A53	RS(NEON)	8	38.7 (1.1x)	74.0 (1.2x)	167.6 (1.3x)
	vQS(NEON)		35.8 (-)	63.4 (-)	126.8 (-)
	QS		55.4 (1.5x)	98.8 (1.6x)	195.9 (1.5x)
	RS(NEON)	16	51.9 (1.1x)	127.8 (1.4x)	264.7 (1.1x)
	vQS(NEON)		47.1 (-)	93.1 (-)	249.3 (-)
	QS		73.6 (1.6x)	138.9 (1.5x)	348.4 (1.4x)
	RS(NEON)	32	109.7 (1.6x)	242.4 (1.5x)	494.5 (1.2x)
	vQS(NEON)		68.4 (-)	158.6 (-)	422.2 (-)
	QS		102.9 (1.5x)	216.4 (1.4x)	551.6 (1.3x)

Tabelle A.6: Laufzeit pro Dokument in μs von RapidScorer(RS), V-QuickScorer(vQS) und QuickScorer(QS) auf dem Klassifikationsdatensatz Fashion

Abbildungsverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] ARM Limited: *Introducing NEON - Development Article*. `https://developer.arm.com/documentation/dht0002/a/`, 2009.

[2] ARM Limited: *ARM Cortex-A Series Programmer's Guide for ARMv8-A (Version 1.0)*. `https://developer.arm.com/documentation/den0024/a/`, 2015.

[3] Asadi, Nima, Jimmy Lin und Arjen P De Vries: *Runtime Optimizations for Tree-Based Machine Learning Models*. IEEE Transactions on Knowledge and Data Engineering, 26(9):2281–2292, 2014.

[4] Breiman, Leo: *Random forests*. Machine learning, 45(1):5–32, 2001.

[5] Buschjäger, Sebastian, Kuan-Hsun Chen, Jian-Jia Chen und Katharina Morik: *Realization of Random Forest for Real-Time Evaluation through Tree Framing*. In: *2018 IEEE International Conference on Data Mining (ICDM)*, Seiten 19–28, 2018.

[6] Buschjäger, Sebastian und Katharina Morik: *Decision Tree and Random Forest Implementations for Fast Filtering of Sensor Data*. IEEE Transactions on Circuits and Systems I: Regular Papers, 65(1):209–222, 2018.

[7] Cambazoglu, B Barla, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng und Jon Degenhardt: *Early exit optimizations for additive machine learned ranking systems*. In: *Proceedings of the third ACM international conference on Web search and data mining*, Seiten 411–420, 2010.

[8] Chen, Tianqi und Carlos Guestrin: *Xgboost: A scalable tree boosting system*. In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, Seiten 785–794, 2016.

[9] Dato, Domenico, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto und Rossano Venturini: *Fast Ranking with Additive Ensembles of Oblivious and Non-Oblivious Regression Trees*. ACM Transactions on Information Systems, 35(2), 2016.

[10] Flach, Peter: *Machine learning: the art and science of algorithms that make sense of data*. Cambridge University Press, 2012.

[11] Frauenhofer ISE: *Öffentliche Nettostromerzeugung in Deutschland in 2021*. `https://www.energy-charts.info/charts/energy_pie/chart.htm?l=de&c=DE&interval=year`. Aufgerufen am 01.11.2021.

[12] Huber, Joseph N, Oscar R Hernandez und Matthew Graham Lopez: *Effective vectorization with OpenMP 4.5*. ORNL/TM-2016/391. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States). Oak Ridge Leadership Computing Facility (OLCF), 2017.

[13] Hughes, Christopher J: *Single-instruction multiple-data execution*. Synthesis Lectures on Computer Architecture, 10(1):1–121, 2015.

[14] Intel: *Advanced vector extensions programming reference*. Intel Corporation, 2011.

[15] Internet Live Stats: *Google Search Statistics*. `https://www.internetlivestats.com/google-search-statistics/`. Aufgerufen am 01.11.2021.

[16] Klemm, Michael, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero und Xavier Martorell: *Extending OpenMP* with vector constructs for modern multicore SIMD architectures*. In: *International Workshop on OpenMP*, Seiten 59–72. Springer, 2012.

[17] LeCun, Yann, Yoshua Bengio und Geoffrey Hinton: *Deep learning*. nature, 521(7553):436–444, 2015.

[18] Lettich, Francesco, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto und Rossano Venturini: *Parallel Traversal of Large Ensembles of Decision Trees*. IEEE Transactions on Parallel and Distributed Systems, 30(9):2075–2089, 2019.

[19] Lomont, Chris: *Introduction to intel advanced vector extensions*. Intel white paper, 23, 2011.

[20] Lucchese, Claudio, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Fabrizio Silvestri und Salvatore Trani: *Post-learning optimization of tree ensembles for efficient ranking*. In: *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, Seiten 949–952, 2016.

[21] Lucchese, Claudio, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto und Rossano Venturini: *QuickScorer: A Fast*

Algorithm to Rank Documents with Additive Ensembles of Regression Trees. In: *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, Seite 73–82. Association for Computing Machinery, 2015.

[22] Lucchese, Claudio, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto und Rossano Venturini: *Exploiting CPU SIMD Extensions to Speed-up Document Scoring with Tree Ensembles.* In: *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '16, Seite 833–836. Association for Computing Machinery, 2016.

[23] Mitchell, Tom M: *Machine learning.* 1997.

[24] Mitra, Gaurav, Beau Johnston, Alistair P Rendell, Eric McCreath und Jun Zhou: *Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms.* In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, Seiten 1107–1116. IEEE, 2013.

[25] OpenMP Architecture Review Board: *OpenMP Application Program Interface (Version 4.0).* 2012.

[26] OpenMP Architecture Review Board: *OpenMP Application Program Interface (Version 4.5).* 2015.

[27] Oshiro, Thais Mayumi, Pedro Santoro Perez und José Augusto Baranauskas: *How many trees in a random forest?* In: *International workshop on machine learning and data mining in pattern recognition*, Seiten 154–168. Springer, 2012.

[28] Polychroniou, Orestis, Arun Raghavan und Kenneth A Ross: *Rethinking SIMD Vectorization for In-Memory Databases.* In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, Seite 1493–1508. Association for Computing Machinery, 2015.

[29] Qin, Tao und Tie-Yan Liu: *Introducing LETOR 4.0 Datasets.* CoRR, abs/1306.2597, 2013.

[30] Quinlan, J. Ross: *Induction of decision trees.* Machine learning, 1(1):81–106, 1986.

[31] Shalev-Shwartz, Shai und Shai Ben-David: *Understanding machine learning: From theory to algorithms.* Cambridge university press, 2014.

[32] Sharp, Toby: *Implementing decision trees and forests on a GPU.* In: *European conference on computer vision*, Seiten 595–608. Springer, 2008.

[33] Tang, Xun, Xin Jin und Tao Yang: *Cache-conscious runtime optimization for ranking ensembles.* In: *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval.* ACM, 2014.

[34] Van Essen, Brian, Chris Macaraeg, Maya Gokhale und Ryan Prenger: *Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?* In: *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, Seiten 232–239, 2012.

[35] Vemuri, Vijay K: *The Hundred-Page Machine Learning Book: by Andriy Burkov, Quebec City, Canada, 2019, 160 pp., 49.99(Hardcover); 29.00 (paperback); 25.43(KindleEdition),(Alternatively,canpurchaseatleanpub.comataminimumpriceof 20.00), ISBN 978-1999579517*, 2020.

[36] Ye, Ting, Hucheng Zhou, Will Y. Zou, Bin Gao und Ruofei Zhang: *RapidScorer: Fast Tree Ensemble Evaluation by Maximizing Compactness in Data Level Parallelization.* In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, Seite 941–950. Association for Computing Machinery, 2018.

[37] Zhou, Zhi-Hua: *Ensemble methods: foundations and algorithms.* Chapman and Hall/CRC, 2019.