technische universität
dortmund

Masterarbeit

Parallele, verteilte Implementierung der Junction-Tree-Inferenz

Lukas Heppe
Dezember 2019

Gutachter:
Prof. Dr. Katharina Morik
Dr. Nico Piatkowski

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

Inhaltsverzeichnis

Kapitel 1

Einleitung

Probabilistische graphische Modelle bieten die Möglichkeit, komplexe Wahrscheinlichkeitsverteilungen anhand von Graphen kompakt zu modellieren und finden in vielen unterschiedlichen Domänen, wie beispielsweise der Modellierung von natürlichsprachlichen Systemen [44], der Modellierung von Verkehrsflüssen [43], der Astrophysik [18] oder auch der *Computer Vision* [26,60] Anwendung. Eine Schlüsselaufgabe in graphischen Modellen ist die Inferenz, d.h. die Berechnung von bedingten- und Randwahrscheinlichkeiten anhand der gemeinsamen Wahrscheinlichkeitsverteilung. Diese kann entweder exakt oder approximativ sein. Die approximativen Ansätze haben allerdings das Problem, dass sie zu Fehlern in der Anwendung führen können [59]. Stattdessen wird sich diese Arbeit primär mit der exakten Inferenz beschäftigen. Die am weitesten verbreitete Technik zur exakten Inferenz ist die Junction-Tree-Inferenz [41]. Das Verfahren wird in zwei Schritten durchgeführt: Zunächst werden die Kreise des Eingabegraphen eliminiert, indem dieser in eine sekundäre Struktur, einen *Junction-Tree*, überführt wird. Anschließend wird auf diesem Baum der *Belief-Propagation*-Algorithmus [53] zur exakten Lösung der Inferenzaufgabe angewendet. Das Verfahren hat jedoch den Nachteil, dass die Komplexität exponentiell mit der Baumweite wächst. Da diskrete graphische Modelle oftmals zur Modellierung von kategorischen Verteilungen mit großem Zustandsraum genutzt werden [57], kann die Inferenz aufgrund dessen nicht in annehmbarer Zeit ausgeführt werden. Dasselbe Problem tritt auf, wenn diskrete graphische Modelle zur Modellierung von kontinuierlichen Zufallsvariablen genutzt werden. In diesem Fall müssen die Zufallsvariablen zunächst diskretisiert werden, wobei die Diskretisierung feingranular gewählt werden sollte, um möglichst wenig Informationen zu verlieren. Auch hier ergeben sich oftmals große Zustandsräume. Um trotzdem die Vorteile der exakten Inferenz genießen zu können, können parallele Programmiertechniken zur Reduzierung der Laufzeit eingesetzt werden. Hier existieren bereits mehrere Ansätze, welche im folgenden Abschnitt erläutert werden. Anschließend werden die Ziele und der Aufbau der Arbeit besprochen.

1.1 Verwandte Arbeiten

In diesem Abschnitt werden die verschiedenen bestehenden Techniken zur Beschleunigung der Junction-Tree-Inferenz vorgestellt. Der erste Ansatz von Namasivayam *et al.* beschäftigt sich mit der parallelen Umwandlung von Eingabegraphen zu Junction-Trees [48]. In der Arbeit wurde untersucht, welche Operationen sich wie parallelisieren lassen und welchen Einfluss diese auf die gesamte Laufzeit nehmen. Dominiert wurde die Laufzeit durch die Initialisierung der Potentialtabellen. Die anderen Arbeiten beschäftigen sich mit der Beschleunigung des Message-Passings. Auch diese Arbeiten lassen sich weiter aufteilen – die diversen Ansätze nutzen sowohl unterschiedliche Strategien zur Parallelisierung als auch verschiedene Zielplattformen. Als Zielplattformen dienten entweder Multiprozessorsysteme oder Rechencluster [39, 61, 62] oder einzelne *Graphics Processing Units* (GPUs) [32, 64, 65]. Zur Parallelisierung wurden zwei verschiedene Strategien genutzt. Die erste Strategie nutzt die Topologie des Graphen aus und führt die unabhängigen Pfade von den Blättern zur Wurzel parallel aus [39]. Die andere Strategie identifiziert unabhängige Berechnungen während der Marginalisierung innerhalb einer einzelnen Nachricht und führt diese parallel aus [65]. Im späteren Verlauf dieser Arbeit werden einige Techniken wiederverwendet und an entsprechender Stelle detaillierter erläutert.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist es, eine parallele und verteilte Implementierung der Junction-Tree-Inferenz zu entwerfen. Dabei sollen die verschiedenen Ansätze der vorherigen Arbeiten kombiniert werden. Die einzelnen Nachrichten sollen wie zuvor mit der Hilfe von Nvidia CUDA beschleunigt werden. Außerdem soll die topologiebasierte Parallelität ausgenutzt werden, um unabhängige Pfade verteilt auf unterschiedlicher Hardware parallel zu berechnen. Da die Initialisierung einen beachtlichen Anteil der Laufzeit einnimmt [48] und diese beispielsweise während des Trainings des Modells in jeder Iteration wiederholt durchgeführt werden muss, wird ebenfalls untersucht, wie diese Operationen weiter beschleunigt werden kann. Der Vollständigkeit halber wird außerdem untersucht, wie die dem Message-Passing folgende Operation der Normalisierung beschleunigt werden kann.

1.3 Aufbau der Arbeit

In Kapitel 2 werden probabilistische graphische Modelle im Detail erklärt. Zu Beginn des Kapitels werden die allgemeinen Grundlagen der Modelle besprochen. Anschließend werden verschiedene Techniken zur Inferenz, insbesondere die Junction-Tree-Inferenz, näher erläutert. Das darauffolgende Kapitel 3 wird die Möglichkeiten und Herausforderungen von paralleler Programmierung besprechen. Außerdem werden dort die verwendeten Fra-

meworks Nvidia CUDA und OpenMP vorgestellt. In Kapitel 4 wird die während der Arbeit entwickelte parallele und verteile Implementierung der Junction-Tree-Inferenz vorgestellt. Diese Techniken aus Kapitel 4 werden anschließend im darauffolgenden Kapitel 5 an verschiedenen Datensätzen getestet und evaluiert. Schließlich werden die Ergebnisse zusammengefasst und es wird ein Fazit gezogen.

Kapitel 2

Probabilistische Graphische Modelle

Probabilistische Graphische Modelle sind ein Teilgebiet des maschinellen Lernens und kombinieren Graphen- und Wahrscheinlichkeitstheorie [5]. Graphische Modelle werden zur kompakten Modellierung von komplexen Wahrscheinlichkeitsverteilungen genutzt. Anhand eines Graphen werden die Zufallsvariablen dieser Verteilung als Knoten dargestellt und deren Unabhängigkeiten über die Kanten. Mit jeder Zufallsvariable $\boldsymbol{X}_v$ ist ein Zustandsraum $\mathcal{X}_v$ verbunden, der sowohl reelle als auch diskrete Werte annehmen kann. Die Kanten können sowohl gerichtet als auch ungerichtet sein und ermöglichen die kompakte Repräsentation der Verteilung. Der Fokus dieser Arbeit wird auf ungerichteten Modellen, die sich *Markov Random Fields* (MRF) nennen, mit diskretem Zustandsraum liegen. Diese werden im folgenden Abschnitt 2.1 näher erläutert. Im weiteren Verlauf dieses Kapitels wird in Abschnitt 2.2 das Training und im darauffolgenden Abschnitt 2.3 die Inferenz an diesen Modellen erklärt. Der letzte Abschnitt 2.4 dieses Kapitels widmet sich dem speziellen Inferenzverfahren der *Junction-Trees*.

2.1 Markov Random Fields

Ein Markov Random Field ist ein probabilistisches graphisches Modell und repräsentiert die gemeinsame Wahrscheinlichkeitsverteilung eines Zufallsvektors $\boldsymbol{X} = (\boldsymbol{X_1}, \ldots, \boldsymbol{X_n})$ über die Menge der (maximalen) Cliquen $\mathcal{C}$ eines ungerichteten Graphen $G = (V, E)$. Damit ein Markov Random Field eine gültige Wahrscheinlichkeitsverteilung darstellt, müssen einige Annahmen getroffen werden, die auf der bedingten Unabhängigkeit beruhen. Deshalb wird diese zunächst definiert.

2.1.1 Definition (Bedingte Unabhängigkeit). Seien $\boldsymbol{X}$, $\boldsymbol{Y}$ und $\boldsymbol{Z}$ Zufallsvariablen. Die Zufallsvariablen $\boldsymbol{X}$ und $\boldsymbol{Y}$ sind bedingt unabhängig, gegeben $\boldsymbol{Z}$ [17]

$$\boldsymbol{X} \perp\!\!\!\perp \boldsymbol{Y} \mid \boldsymbol{Z}, \tag{2.1}$$

genau dann, wenn die folgende Gleichung gilt

$$\mathbb{P}(\boldsymbol{X}, \boldsymbol{Y} | \boldsymbol{Z}) = \mathbb{P}(\boldsymbol{X} | \boldsymbol{Z}) \cdot \mathbb{P}(\boldsymbol{Y} | \boldsymbol{Z}). \tag{2.2}$$

Auf Basis der bedingten Unabhängigkeiten werden die folgenden Annahmen getroffen [42], welche die kompakte Darstellung und effiziente Entwicklung von Algorithmen für diese Modelle erlauben.

Paarweise Markov-Eigenschaft Zwei nicht benachbarte Knoten $u, v \in V$ und deren Zufallsvariablen $\boldsymbol{X}_u$ und $\boldsymbol{X}_v$ sind bedingt unabhängig, gegeben die restlichen Variablen des Graphen:

$$\boldsymbol{X}_u \perp\!\!\!\perp \boldsymbol{X}_v \mid \boldsymbol{X}_{V \setminus \{u,v\}}$$

Lokale Markov-Eigenschaft Die Zufallsvariable $\boldsymbol{X}_u$ eines Knoten $u \in V$ ist bedingt unabhängig von allen Variablen des Graphen außer seiner Nachbarn $\mathcal{N}(u)$, gegeben seine Nachbarn:

$$\boldsymbol{X}_u \perp\!\!\!\perp \boldsymbol{X}_{V \setminus \{\mathcal{N}(u)\}} \mid \boldsymbol{X}_{\mathcal{N}(u)}$$

Globale Markov-Eigenschaft Seien $\boldsymbol{X}_A$, $\boldsymbol{X}_B$ und $\boldsymbol{X}_S$ disjunkte Teilmengen der Variablen des Graphen G. Falls $\boldsymbol{X}_S$ eine separierende Teilmenge ist, also alle Pfade von einem Knoten aus $\boldsymbol{X}_A$ zu $\boldsymbol{X}_B$ durch Knoten aus $\boldsymbol{X}_S$ verlaufen, sind $\boldsymbol{X}_A$ und $\boldsymbol{X}_B$ bedingt unabhängig, gegeben $\boldsymbol{X}_S$,

$$\boldsymbol{X}_A \perp\!\!\!\perp \boldsymbol{X}_B \mid \boldsymbol{X}_S.$$

Die Markov-Eigenschaften erlauben die Darstellung der gemeinsamen Wahrscheinlichkeitsverteilung als Produkt kleiner Verteilungen über die Cliquen des Graphen. Jeder Clique $C \in \mathcal{C}$ wird dazu eine Potentialfunktion $\psi_C : \mathcal{X}_C \mapsto \mathbb{R}_+$ zugeordnet, welche eine Realisierung der Zufallsvariable $\boldsymbol{X}_C$ auf eine positive reelle Zahl abbildet. Über die Potentialfunktionen wird die gemeinsame Wahrscheinlichkeitsverteilung eines Markov Random Fields wie folgt definiert [5]

$$\mathbb{P}(\boldsymbol{X} = \boldsymbol{x}) = \frac{1}{Z} \prod_{C \in \mathcal{C}} \psi_C(\boldsymbol{x}_C), \tag{2.3}$$

wobei Z die Partitionsfunktion ist und dafür sorgt, dass $\mathbb{P}(\boldsymbol{X})$ eine gültige Wahrscheinlichkeitsverteilung ist. Wenn die Variablen des Modells diskrete Zustände annehmen, wird die Partitionsfunktion wie folgt berechnet

$$Z = \sum_{\boldsymbol{x} \in \mathcal{X}} \prod_{C \in \mathcal{C}} \psi_C(\boldsymbol{x}). \tag{2.4}$$

Im Falle von stetigen Modellen wird die Summe durch ein Integral über alle möglichen Realisierungen ersetzt. Des Weiteren sind graphische Modelle immer durch einen Vektor

$\boldsymbol{\theta} \in \mathbb{R}^d$ parametrisiert, welcher gemeinsam mit der Graphstruktur die Wahrscheinlichkeitsverteilung bestimmt und aus gegebenen Daten während des Trainings geschätzt werden muss. Da der Fokus dieser Arbeit auf diskreten Modellen liegt, werden für diese zunächst die Potentialfunktionen eingeführt.

Diskrete Potentialfunktionen Sei $G = (V, E)$ ein graphisches Modell, in welchem jede Zufallsvariable $\boldsymbol{X}_v$ den Zustandsraum $\mathcal{X}_v = \{1, \ldots, m\}$ besitzt. Ferner sei C eine Clique des Graphen G mit $\mathcal{X}_C = \{1, \ldots, m^{|C|}\}$. Zur Definition der Potentialfunktionen ψ_C werden Indikatorfunktionen $\mathcal{I}[\cdot]$ genutzt,

$$\mathcal{I}[\boldsymbol{x}_C = j] = \begin{cases} 1 & \boldsymbol{x}_C = j \\ 0 & \text{sonst.} \end{cases} \tag{2.5}$$

Mit Hilfe eines Parametervektors $\boldsymbol{\theta}_C \in \mathbb{R}^{|\mathcal{X}_C|}$, welcher einen Eintrag für jeden Zustand der Clique C besitzt, und der Indikatorfunktion wird das Potential $\psi_C(\boldsymbol{x}_C)$ einer Clique C wie folgt definiert

$$\psi_C(\boldsymbol{x}_C) = \exp\Big(\sum_{x \in \mathcal{X}_C} \boldsymbol{\theta}_{C;x} \mathcal{I}[\boldsymbol{x}_C = \boldsymbol{x}] \Big). \tag{2.6}$$

Alternativ können die Potentialfunktionen über Featurefunktionen $\phi(\boldsymbol{x})$ definiert werden, welche einen Zustand x der Clique C auf einen Binärvektor der Länge $|\mathcal{X}_C|$ abbilden, welche genau eine eins für den jeweiligen Zustand gesetzt haben

$$\psi_C(\boldsymbol{x}_C) = \exp(\langle \boldsymbol{\theta}_C, \phi(\boldsymbol{x}_C) \rangle). \tag{2.7}$$

Das Einsetzen von Gleichung (2.7) in die gemeinsame Wahrscheinlichkeitsverteilung des Modells (2.3) liefert folgende neue Form, welche zur Klasse der Exponentialfamilie [59] gehört

$$\begin{aligned} \mathbb{P}_{\boldsymbol{\theta}}(\boldsymbol{X} = x) &= \frac{1}{Z(\boldsymbol{\theta})} \prod_{C \in \mathcal{C}} \exp(\langle \boldsymbol{\theta}_C, \phi(\boldsymbol{x}_C) \rangle) \\ &= \frac{1}{Z(\boldsymbol{\theta})} \exp(\sum_{C \in \mathcal{C}} (\langle \boldsymbol{\theta}_C, \phi(\boldsymbol{x}_C) \rangle) \\ &= \frac{1}{Z(\boldsymbol{\theta})} \exp(\langle \boldsymbol{\theta}, \phi(\boldsymbol{x}) \rangle) \\ &= \exp(\langle \boldsymbol{\theta}, \phi(\boldsymbol{x}) \rangle - \log Z(\boldsymbol{\theta})). \end{aligned}$$

Der Vektor $\boldsymbol{\theta}$ setzt sich aus der Konkatenation der jeweiligen Gewichtsvektoren $\boldsymbol{\theta}_C$ aller Cliquen zusammen und die Funktion $\phi(\boldsymbol{x})$ ist die Konkatenation aller Binärvektoren der Cliquen des Graphen.

Exponentialfamilie

Die Exponentialfamilie [59] ist eine Klasse von Wahrscheinlichkeitsverteilungen, welche in einer bestimmten Form vorliegen. Viele der oft gebräuchlichen Verteilungen, wie z.B. die

Normal-, Exponential-, Bernoulli- oder Poisson-Verteilung, gehören dieser Familie an. Die Wahrscheinlichkeitsfunktion (bzw. Dichtefunktion im stetigen Fall) der Exponentialfamilie wird wie folgt definiert [59]

$$\mathbb{P}_{\boldsymbol{\theta}}(\boldsymbol{X} = \boldsymbol{x}) = \exp\{\langle \boldsymbol{\theta}, \phi(\boldsymbol{x}) \rangle - A(\boldsymbol{\theta})\}, \tag{2.8}$$

wobei $\boldsymbol{\theta} \in \mathbb{R}^d$ der Parametervektor ist, die Funktion $\phi(\boldsymbol{X})$ eine suffiziente Statistik mit einer Featurefunktion $\phi : \mathbb{R}^p \mapsto \mathbb{R}^d$ und $A(\boldsymbol{\theta}) = \log Z(\boldsymbol{\theta})$ als Partitionsfunktion. Im Kontext graphischer Modelle kodiert ϕ die Graphstruktur über die Cliquen

$$\phi(\boldsymbol{x}) = (\phi_C(\boldsymbol{x}) : \forall\, C \in \mathcal{C}). \tag{2.9}$$

Der Gewichtsvektor $\boldsymbol{\theta}$ der Verteilung setzt sich aus der Konkatenation der jeweiligen Gewichtsvektoren $\boldsymbol{\theta}_C$ aller Cliquen zusammen. Hier wird deutlich, wo die Kompaktheit dieser Modelle ihren Ursprung hat. Anstatt für jeden Zustand der gemeinsamen Verteilung eine Wahrscheinlichkeit einen Wert speichern zu müssen ($\prod_{v \in V} \mathcal{X}_v$), wird nur für jede Clique und jeden ihrer annehmbaren Zustände ein Wert gespeichert ($\sum_{C \in \mathcal{C}(G)} \mathcal{X}_C$) .

2.2 Training

Beim Training eines graphischen Modells sollen die Parameter $\boldsymbol{\theta}$ aus einem gegebenen Datensatz $\mathcal{D} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)}\}$ mit $\boldsymbol{x}^{(i)} \in \mathcal{X}^p$ so geschätzt werden, dass die Verteilung des Modells der Verteilung der Daten entspricht [59]. Dabei wird die Annahme getroffen, dass die Daten einer fixen, unbekannten Verteilung folgen und die Beispiele der Stichprobe unabhängig und gleichverteilt (i.i.d) gezogen wurden. Ist die Struktur, also die Menge der Kanten E, bekannt, können die Parameter durch die Maximum-Likelihood-Methode geschätzt werden. Dazu wird die Likelihood-Funktion $\mathcal{L}$ genutzt, welche ihr Maximum annimmt, falls die Daten $\mathcal{D}$ wahrscheinlich aus der durch $\boldsymbol{\theta}$ parametrisierten Verteilung $\mathbb{P}_{\boldsymbol{\theta}}$ generiert wurden [27]. Die Likelihood-Funktion wird wie folgt definiert

$$\mathcal{L}(\boldsymbol{\theta}|\mathcal{D}) = \prod_{i=1}^{n} \mathbb{P}_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}). \tag{2.10}$$

Üblicherweise wird jedoch die Log-Likelihood ℓ verwendet, um numerische Instabilitäten zu vermeiden. Diese treten auf, da zur Berechnung der Funktion viele kleine Werte miteinander multipliziert werden. Außerdem kann der Gradient einfacher berechnet werden, denn es muss nicht wiederholt die Produktregel angewandt werden. Die Log-Likelihood ℓ ist wie folgt definiert

$$\ell(\boldsymbol{\theta}|\mathcal{D}) = \sum_{i=1}^{n} \log \mathbb{P}_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}). \tag{2.11}$$

Da angenommen wird, dass die Likelihood-Funktion ihr Maximum für einen Parameter, welche die Daten $\mathcal{D}$ am wahrscheinlichsten erzeugt hat, annimmt, wird folgendes Optimierungsproblem aufgestellt

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^d} -\ell(\boldsymbol{\theta};\mathcal{D}), \tag{2.12}$$

welches beispielsweise durch gradientenbasierte Verfahren [49] gelöst werden kann, die den Parametervektor iterativ mit einer Schrittweite $\alpha > 0$ nach folgender Gleichung aktualisieren

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{\theta}|\mathcal{D}).$$

Für Modelle der Exponentialfamilie lautet die Log-Likelihood wie folgt [59]

$$\ell(\boldsymbol{\theta}|\mathcal{D}) = \langle\boldsymbol{\theta}, \hat{\boldsymbol{\mu}}\rangle - A(\boldsymbol{\theta}), \tag{2.13}$$

wobei $\hat{\boldsymbol{\mu}} = \hat{\mathbb{E}}[\phi(\boldsymbol{X})] = \frac{1}{n}\sum_{i=1}^{n}\phi(\boldsymbol{x}^{(i)})$ der empirische Mittelwertparameter ist. Für diskrete paarweise Modelle ist $\hat{\boldsymbol{\mu}}$ die Konkatenation der beobachteten relativen Kantenwahrscheinlichkeiten des Datensatzes $\mathcal{D}$. Nach Wainwright und Jordan ist die Ableitung der Partitionsfunktion der Erwartungswert der Verteilung unter dem Parameter $\boldsymbol{\theta}$ [59]. Somit lässt sich der Gradient der Log-Likelihood als die Differenz zwischen dem Erwartungswert der empirischen Verteilung und der des Modells wie folgt aufschreiben

$$\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{\theta}|\mathcal{D}) = \hat{\mathbb{E}}[\phi(\boldsymbol{X})] - \mathbb{E}_{\boldsymbol{\theta}}[\phi(\boldsymbol{X})]. \tag{2.14}$$

Zur Berechnung des Gradienten werden also die erwarteten Kantenwahrscheinlichkeiten benötigt, welche durch verschiedene Inferenztechniken berechnet werden können. Diese werden im folgenden Abschnitt näher erläutert.

2.3 Inferenz

In probabilistischen Modellen wird unter dem Begriff Inferenz die Berechnung bestimmter Eigenschaften und Werte anhand einer gemeinsamen Wahrscheinlichkeitsverteilung verstanden [59]. Zunächst werden die typischen Problemstellungen der Inferenz definiert und anschließend erläutert, wie diese mit einem graphischen Modell gelöst werden können.

Marginalisierung Sei $\mathbb{P}$ eine gemeinsame Wahrscheinlichkeitsverteilung über zwei Zufallsvariablen $\boldsymbol{A}$ und $\boldsymbol{B}$. Die Verteilung einer einzelnen Variable wird als Randverteilung bezeichnet. Zur Berechnung der Randverteilung einer Variable müssen die Wahrscheinlichkeiten aller annehmbaren Zustände der restlichen Variablen aufsummiert werden, während der Wert der zu berechnenden Variable fixiert wird. Die Randverteilung $\mathbb{P}_{\boldsymbol{A}}$ der Zufallsvariable $\boldsymbol{A}$ kann wie folgt berechnet werden

$$\mathbb{P}_{\boldsymbol{A}}(\boldsymbol{A} = \boldsymbol{a}) = \sum_{b\in\mathcal{B}}\mathbb{P}(\boldsymbol{A} = \boldsymbol{a}, \boldsymbol{B} = \boldsymbol{b}).$$

Bedingte Wahrscheinlichkeiten Sei $(\boldsymbol{X}_Q, \boldsymbol{X}_E)$ eine Partitionierung der Variablen einer gemeinsamen Wahrscheinlichkeitsverteilung $\mathbb{P}$ in eine Menge von Anfragevariablen $\boldsymbol{X}_Q$ und eine Menge von Evidenzvariablen E. Ferner sei $\boldsymbol{q}$ eine Zuweisung der Variablen $\boldsymbol{X}_Q$ und $\boldsymbol{e}$ eine Zuweisung für $\boldsymbol{X}_E$. Gesucht wird die bedingte Wahrscheinlichkeit, dass sich die Anfragevariablen in Zustand q befinden, gegeben die beobachteten Daten $\boldsymbol{e}$

$$\mathbb{P}(\boldsymbol{X}_Q = \boldsymbol{q} | \boldsymbol{X}_E = \boldsymbol{e}). \tag{2.15}$$

Dieses Problem kann als Spezialfall der Marginalisierung betrachtet werden, denn sofern die gemeinsame Verteilung der Variablen sowie die Randverteilung der Evidenzvariablen $\boldsymbol{X}_E$ bekannt ist, lautet die bedingte Wahrscheinlichkeit

$$\mathbb{P}(\boldsymbol{X}_Q = \boldsymbol{q} | \boldsymbol{X}_E = \boldsymbol{e}) = \frac{\mathbb{P}(\boldsymbol{X}_Q = \boldsymbol{q}, \boldsymbol{X}_E = \boldsymbol{e})}{\mathbb{P}(\boldsymbol{X}_E = \boldsymbol{e})}. \tag{2.16}$$

Maximum-a-posteriori Zustand Des Weiteren ist die Berechnung des Maximum-a-posteriori Zustands (MAP) eine wichtige Aufgabe, welche zur Vorhersage genutzt werden kann. Der MAP-Zustand ist derjenige, welcher die höchste Wahrscheinlichkeit besitzt.

$$\boldsymbol{x}^* = \arg\max_{\boldsymbol{x} \in \mathcal{X}} \mathbb{P}(\boldsymbol{X} = \boldsymbol{x}) \tag{2.17}$$

In diese Berechnung können ebenfalls Beobachtungen einbezogen werden. Wie schon bei der Berechnung der bedingten Wahrscheinlichkeit gesehen, wird die Knotenmenge partitioniert. Anschließend muss folgendes Problem gelöst werden

$$\boldsymbol{x}^* = \arg\max_{\boldsymbol{x} \in \mathcal{X}_Q} \mathbb{P}(\boldsymbol{X}_Q = \boldsymbol{q} | \boldsymbol{X}_E = \boldsymbol{e}). \tag{2.18}$$

Partitionsfunktion Die komplexeste Inferenzaufgabe ist die Berechnung der Partitionsfunktion, denn um sie zu berechnen, müssen die Potentiale jedes Zustands des Zustandsraum aufsummiert werden

$$Z = \sum_{\boldsymbol{x} \in \mathcal{X}} \prod_{C \in \mathcal{C}} \psi_C(\boldsymbol{x}). \tag{2.19}$$

Exakte und approximative Inferenz

Für allgemeine Graphen ist die exakte Inferenz ein NP-schweres Problem [9]. Um die Marginalverteilung einer Variable zu berechnen, müssen die restlichen Variablen heraus marginalisiert werden. Dies ist insbesondere dann aufwendig, wenn die Verteilung viele Variablen enthält und diese viele Zustände annehmen können. In graphischen Modellen kann jedoch die Eigenschaft, dass sich die Verteilung aus dem Produkt kleinerer Verteilungen zusammensetzt, ausgenutzt werden. Da jeder Faktor nur von einer Teilmenge der Variablen abhängt, können aufgrund des Distributivgesetzes einzelne Summen weiter nach innen gezogen werden, wodurch die Anzahl der Berechnungen verringert wird. Eine

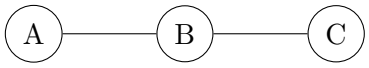


Abbildung 2.1: Diese Abbildung zeigt einen Kettengraph mit drei Knoten.

Eliminationsordnung der Variablen des Graphen bestimmt die Reihenfolge, in der die einzelnen Faktoren heraus marginalisiert werden sollen. Da die Variablen nacheinander herausmarginalisiert werden, wird dieses Vorgehen *Variablenelimination* genannt [37].

2.3.1 Beispiel. Sei $G = (V, E)$ ein Kettengraph, der aus den drei Knoten A, B und C besteht (siehe Abbildung 2.1). Die nicht normalisierte Wahrscheinlichkeitsverteilung des Graphen lautet

$$\mathbb{P}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}) \propto \psi(\boldsymbol{A}, \boldsymbol{B})\psi(\boldsymbol{B}, \boldsymbol{C}).$$

Zur Berechnung der Marginalverteilung der Variable $\boldsymbol{A}$, müssen alle Zustände der Variablen $\boldsymbol{B}$ und $\boldsymbol{C}$ heraus marginalisiert werden.

$$\mathbb{P}(\boldsymbol{A} = a) \propto \sum_{b \in \mathcal{B}} \sum_{c \in \mathcal{C}} \mathbb{P}(\boldsymbol{A} = a, \boldsymbol{B} = b, \boldsymbol{C} = c)$$
$$\mathbb{P}(\boldsymbol{A} = a) \propto \sum_{b \in \mathcal{B}} \sum_{c \in \mathcal{C}} \psi(a, b)\psi(b, c)$$

Im oberen Schritt wurde die Definition der faktorisierten Verteilung eingesetzt. Da das Potential $\psi(a, b)$ nicht von der Variable $\boldsymbol{C}$ abhängt, kann die Berechnung durch die Anwendung des Distributivgesetztes wie folgt vereinfacht werden.

$$\mathbb{P}(\boldsymbol{A} = a) \propto \sum_{b \in \mathcal{B}} \psi(a, b) \sum_{c \in \mathcal{C}} \psi(b, c)$$

Belief Propagation Auch wenn der Variableneliminationsalgorithmus dem naiven Ansatz überlegen ist, kann er nur die Randverteilung einer Variable errechnen. Soll die Verteilung zweier oder mehrerer Variablen berechnet werden, muss die aufwendige Marginalisierung wiederholt werden. Pearl hat dabei erkannt, dass zur Berechnung der Randverteilung zweier Variablen viele gemeinsame Zwischenergebnisse berechnet werden, welche weitergenutzt werden können. Diese Erkenntnis hat er in dem Message-Passing-Algorithmus *Belief Propagation* [53] ausgenutzt, um auf Baumstrukturen die exakte Randverteilung aller Variablen gleichzeitig berechnen zu können. Um die Zwischensummen speichern zu können, wird für jede Kante (u, v) pro Richtung eine Nachricht $m_{u \to v}(\boldsymbol{x_v})$ und $m_{v \to u}(\boldsymbol{x_u})$ angelegt. Die Nachrichten können für jeden Zustand der Zielvariable die Zwischensumme speichern und entsprechen den heraus marginalisierten Faktoren im Variableneliminationsalgorithmus.

Zu Beginn werden die Nachrichten mit dem Wert eins initialisiert. Darauf folgen die zwei Phasen des Message-Passing: der *Inside*- und der *Outside*-Pass. Während des Inside-Pass

wird ein Knoten des Graphen als Wurzel ausgewählt und alle Blätter beginnen damit, die Nachrichten in Richtung der Wurzel zu senden. Die Nachricht eines Knoten u an einen Knoten v ist dabei wie folgt definiert

$$m_{u\to v}(\boldsymbol{x}_v) = \sum_{\boldsymbol{x}_u \in \mathcal{X}_u} \psi_u(\boldsymbol{x}_v) \cdot \prod_{k \in \mathcal{N}(u)\setminus\{v\}} m_{k\to u}(\boldsymbol{x}_u). \tag{2.20}$$

Jeder Knoten muss sich dabei an das Protokoll halten, welches besagt, dass er seine Nachricht in Richtung der Wurzel erst absenden darf, sobald die Nachrichten der restlichen Nachbarn eingetroffen sind. Nachdem der Wurzelknoten die Nachrichten aller seiner Nachbarn erhalten hat, beginnt der Outside-Pass: Die Nachrichten müssen von der Wurzel zurück den zu Blättern propagiert werden. Es lässt sich zeigen, dass wenn der zugrunde liegende Graph ein Baum ist, immer ein Knoten existiert, der eine Nachricht weiterleiten kann [5]. Insgesamt müssen $2|E|$ Nachrichten berechnet werden. Sobald der Algorithmus terminiert, kann die Marginalverteilung eines Knoten v als Produkt der eintreffenden Nachrichten abgefragt werden

$$\mathbb{P}_{\boldsymbol{v}}(\boldsymbol{X}_v = \boldsymbol{x}) = \prod_{u \in \mathcal{N}(v)} m_{u\to v}(\boldsymbol{x}).$$

Die Normalisierungskonstante Z kann nach dem Message-Passing ebenfalls an jedem Knoten berechnet werden

$$Z = \sum_{\boldsymbol{x} \in \mathcal{X}_v} \prod_{u \in \mathcal{N}(v)} m_{u\to v}(\boldsymbol{x}).$$

Indem die Summationsoperation in der Gleichung 2.20 durch die `max`-Operation ersetzt wird, kann der Algorithmus ebenfalls zur Berechnung des Maximum-a-posteriori-Zustands der Verteilung genutzt werden. Der Belief-Propagation-Algorithmus liefert jedoch nur auf Bäumen das korrekte Ergebnis, da in Graphen, die Kreise enthalten, das Protokoll nicht eingehalten werden kann. Alternativ kann auf das *Junction-Tree*-Verfahren [41] oder eine approximative Technik zurück gegriffen werden. Bevor das Junction-Tree-Verfahren in Abschnitt 2.4 näher erläutert wird, gibt es einen kleine Übersicht über approximative Inferenzverfahren.

Loopy Belief Propagation Obwohl nicht garantiert ist, dass der Belief-Propagation-Algorithmus auf einem Graph mit Kreisen konvergiert, wird er oftmals zur Approximation der Randverteilung genutzt. Dazu wird der Belief-Propagation-Algorithmus in einer Schleife für eine maximale Anzahl an Iterationen oder bis die Nachrichten gegen einen Wert konvergieren, wiederholt ausgeführt. Das Verhalten dieses Algorithmus wurde in einer Arbeit von Murphy [47] untersucht. Dabei stellte sich heraus, dass falls der Algorithmus konvergiert, er oftmals eine gute Approximation liefert. Jedoch gibt es keine Konvergenzgarantien und die Lösung kann beliebig große Fehler enthalten [45].

Stichprobenbasierte Inferenz Eine weitere Methode zur approximativen Inferenz sind *Markov-Chain-Monte-Carlo*-Verfahren (MCMC) [3]. Mit Hilfe dieser Methoden können Stichproben aus einer Verteilung gezogen werden, welche wiederum dafür eingesetzt werden können, um Erwartungswerte und Wahrscheinlichkeiten über Monte-Carlo-Simulationen abzuschätzen

$$\mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}}[f(\boldsymbol{x})] \approx \frac{1}{N}\sum_{i=1}^{n} f(\boldsymbol{x}^{(i)}).$$

Zum Ziehen von Beispielen wird in graphischen Modellen oft der *Gibbs-Sampling*-Algorithmus [8] (Algorithmus 1) verwendet. Aufgrund der Markov-Eigenschaft eines Graphen kann effizient aus der bedingten Verteilung eines Knoten, gegeben seine Nachbarn, ein Wert gezogen werden. Jedoch haben MCMC-Verfahren einige Nachteile. Es ist oftmals unklar, wie die Anzahl der Iterationen gesetzt werden muss, um ein valides Samples der Verteilung zu erhalten. Ebenfalls ist nicht bekannt, wie viele Beispiele aus der Verteilung gezogen werden müssen, um gute Approximationen für die Erwartungswerte und Wahrscheinlichkeiten zu erhalten.

Algorithmus 1 Gibbs Sampling [8]

Eingabe: Graph $G = (V, E)$, Anzahl an Iterationen n, initialer Zustand $\boldsymbol{x}^{(0)}$
Ausgabe: Sample $\boldsymbol{x}$ der Verteilung

1: **for** i = 1 **to** n **do**
2: $\quad \boldsymbol{x}^{(i)} = \boldsymbol{x}^{(i-1)}$
3: $\quad$ **for** $v \in V$ **do**
4: $\quad\quad \boldsymbol{x}_v^{(i)} \sim \mathbb{P}(\boldsymbol{X}_v|\mathcal{N}(\boldsymbol{X}_v))$
5: **return** $\boldsymbol{x}^{(n)}$

Variational Inferenz Viele komplexe Wahrscheinlichkeitsverteilungen lassen sich in der Realität nicht ausrechnen, da beispielsweise der Zustandsraum zu groß ist, um die Partitionsfunktion berechnen zu können. Die Idee hinter der Technik namens *Variational Inference* [6,33] ist es, die gesuchte Verteilung durch eine einfachere Verteilung zu approximieren. Dazu wird zunächst eine Klasse von Ersatzverteilungen $\mathcal{Q}$ bestimmt, von der angenommen wird, dass sie der originalen Verteilung möglichst ähnlich ist. Wie unterschiedlich zwei Verteilungen sind, kann durch die Kullback-Leibler-Divergenz [12] gemessen werden. Seien $\mathbb{P}$ und $\mathbb{Q}$ Wahrscheinlichkeitsverteilungen über eine Zufallsvariable $\boldsymbol{X}$ mit Zustandsraum $\mathcal{X}$, dann ist die Kullback-Leibler-Divergenz (KL) $D_{KL}(\mathbb{P}|\mathbb{Q})$ für diskrete Verteilungen wie folgt definiert

$$D_{KL}(\mathbb{P}|\mathbb{Q}) = \sum_{\boldsymbol{x}\in\mathcal{X}} \mathbb{P}(\boldsymbol{x}) \log \frac{\mathbb{P}(\boldsymbol{x})}{\mathbb{Q}(\boldsymbol{x})}.$$

Anschließend wird ein Optimierungsproblem aufgestellt, welches die Verteilung der Ersatzverteilungen sucht, welche am nächsten an der wahren Verteilung liegt

$$\mathbb{Q}^* = \arg\min_{\mathbb{Q}\in\mathcal{Q}} D_{KL}(\mathbb{Q}|\mathbb{P}). \tag{2.21}$$

Mit dieser Formulierung besteht jedoch weiterhin das Problem, dass die Verteilung $\mathbb{P}$ ausgewertet werden muss, um die Divergenz zu berechnen. Zur Lösung wird $\mathbb{P}$ einfach durch die nicht normalisierte Verteilung $\widetilde{\mathbb{P}}$ ersetzt, woraus die *evidence lower bound* (ELBO) resultiert. Dies ist eine untere Schranke der Evidenzlikelihood, deren Maximierung äquivalent zur Minimierung der KL-Divergenz ist. Anschließend wird diese Schranke über eine Klasse von Verteilungen optimiert. Die einfachste Variante ist die *Mean-Field*-Approximation, in welcher die Ersatzverteilung nur aus einzelnen unabhängigen Faktoren besteht

$$\mathcal{Q}(\boldsymbol{x}) = \prod_{v\in V} \mathrm{q}(\boldsymbol{x}_v). \tag{2.22}$$

Im Kontext von graphischen Modellen bedeutet dies, dass alle Kanten des Graphen entfernt werden. Diese Technik ist oftmals schneller als Sampling-Verfahren, konvergiert jedoch selten gegen das globale Optimum [6].

2.4 Junction Tree

Eine Technik zur exakten Inferenz auf beliebigen Graphen ist das *Junction-Tree*-Verfahren [41]. Die Idee hinter dem Verfahren ist es, den Eingabegraphen in einen *Junction-Tree* zu überführen und somit die Kreise zu eliminieren, indem die maximalen Cliquen zu neuen Knoten zusammengeführt werden. Anschließend wird die Eigenschaft des Belief-Propagation-Algorithmus ausgenutzt, dass dieser auf Bäumen die exakte Lösung liefert. Ein Junction-Tree $\mathcal{T}$, auch Join- oder Cluster-Tree genannt, ist ein ungerichteter Baum, dessen Knoten den maximalen Cliquen eines Graphen G entsprechen. Verbunden werden die Knoten so, dass die *Running Intersection Property* erfüllt ist. Die Running Intersection Property besagt, dass wenn eine Zufallsvariable $\boldsymbol{X}$ in einer Clique C_i und einer Clique C_j vorkommt, sie ebenfalls in jeder Clique auf dem Pfad von C_i zu C_j enthalten ist [37]. Durch die Running Intersection Property wird garantiert, dass für jede Kante des Junction-Trees die Schnittmenge der anliegenden Cluster nicht leer ist. Neben den Clustern wird für jede Kante ein zusätzlicher Knoten, ein Separator, eingeführt. Ein Separator zwischen zwei Cliquen C_i und C_j ist immer mit der Schnittmenge der Variablen der anliegenden Cluster assoziiert $\boldsymbol{X}_S = \boldsymbol{X}_{C_i} \cap \boldsymbol{X}_{C_j}$. Des Weiteren wird für jeden Separator ein zusätzliches Potential $\psi_S(\boldsymbol{X}_S)$ eingeführt. Die Separatoren garantieren zusammen mit der Running Intersection Property, dass wenn eine Zufallsvariable in mehreren Clustern des Junction-Trees enthalten ist, sie nach dem Message-Passing in beiden Clustern die selbe Verteilung

annimmt. Mit den Separatoren wird die Wahrscheinlichkeitsverteilung des Baumes wie folgt definiert

$$\mathbb{P}(\boldsymbol{X} = \boldsymbol{x}) = \frac{\prod_{C \in \mathcal{C}} \psi_C(\boldsymbol{x})}{\prod_{S \in \mathcal{S}} \psi_S(\boldsymbol{x})}. \tag{2.23}$$

2.4.1 Kompilierungsphase

Bevor die eigentliche Inferenzroutine ausgeführt werden kann, muss zu einem Basisgraph G zunächst einmalig ein Junction-Tree $\mathcal{T}$ berechnet werden. Da die Knoten eines Junction-Trees den maximalen Cliquen des Basisgraphen entsprechen, müssen diese zunächst identifiziert werden. Jedoch ist in allgemeinen Graphen die Suche nach der Menge aller maximalen Cliquen ein NP-vollständiges Problem [35]. Als Lösung bietet sich die Triangulation des Graphen an, da für einen triangulierten Graphen die maximalen Cliquen in linearer Zeit bestimmt werden können. Es werden vier Schritte zur Berechnung eines Junction-Trees benötigt, welche in den folgenden Abschnitten näher erläutert werden. Die ersten drei Schritte werden genutzt, um die maximalen Cliquen des Basisgraphen zu identifizieren. Im letzten Schritt werden die Cliquen so verbunden, dass es sich bei dem resultierenden Graphen um einen Junction-Tree handelt. Es sei erwähnt, dass die Kompilierungsphase nicht deterministisch ist und für den selben Graphen viele unterschiedliche Junction-Trees existieren [31].

Moralisierung Der erste Schritt der Kompilierungsphase ist die Moralisierung des Graphen und wird nur für gerichtete azyklische Graphen (DAG) benötigt, da in diesem Schritt der Graph in einen Ungerichteten transformiert wird [13]. Dazu werden zunächst die gerichteten Kanten in Ungerichtete umgewandelt. Anschließend wird für jeden Knoten überprüft, ob dieser zwei Eltern besitzt. Ist dies der Fall, wird zwischen diesen eine zusätzliche Kante eingefügt.

Triangulation Während der Triangulation eines ungerichteten Graphen wird dieser in einen triangulierten Graphen überführt. Ein Graph wird trianguliert oder auch chordal genannt, wenn jeder Kreis der Länge vier oder größer eine *Sehne* (engl. chord) besitzt. Als Sehne wird eine Kante bezeichnet, welche zwei nicht benachbarte Knoten eines Kreises verbindet. Die Triangulation ist essentiell, denn nach Lauritzen existieren Junction-Trees nur für triangulierte Graphen [42]. Außerdem kann die Menge der maximalen Cliquen eines chordalen Graphen in linearer Zeit bestimmt werden [22]. Beliebige ungerichtete Graphen können durch das Ergänzen von zusätzlichen Kanten trianguliert werden. In Abbildung 2.2 ist ein Graph zu sehen, der durch die Hinzunahme der blau markierten Kanten zu einem chordalen Graphen wird. Idealerweise wird die minimale Triangulierung gefunden, was jedoch nicht garantiert ist, da die Suche nach dieser ein NP-hartes Problem ist [63] und die verwendeten Algorithmen nur Approximationen liefern. Eine Triangulierung ist

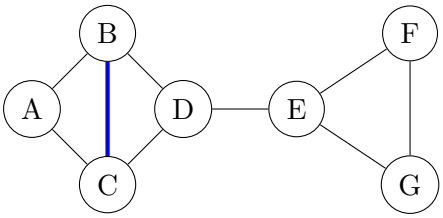


Abbildung 2.2: Diese Abbildung zeigt einen Graphen, der durch die Hinzunahme der blau markierten Kante chordal (trianguliert) wird.

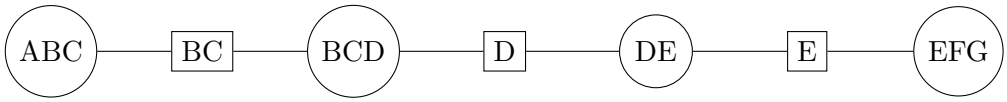


Abbildung 2.3: Diese Abbildung zeigt einen Junction-Tree zu dem Graphen aus Abb. 2.2. Die Cluster werden durch runde Knoten dargestellt, die Separatoren durch Rechtecke.

minimal genau dann, wenn die Entfernung einer der hinzugefügten Kanten in einem nicht chordalen Graphen resultiert [28]. Nicht-minimale Triangulierungen haben den Nachteil, dass zusätzliche Kanten eingefügt werden, wodurch die Cliquengrößen wachsen und somit die Laufzeit zunimmt. Von der Triangulierung wird letztendlich die Komplexität der Inferenz bestimmt. Als Baumweite r ist die Kardinalität der maximalen Cliquen des chordalen Graphen minus eins definiert. Die Komplexität während des Message-Passing beläuft bei m Kanten und einer Zustandsmenge s pro Knoten auf $\mathcal{O}(m \cdot s^r)$ [46].

Algorithmus 2 Maximum Cardinality Search [4]

Eingabe: Graph $G = (V, E)$

Ausgabe: Eliminationsordung π

Initialisiere alle Knoten als unmarkiert.
for i $= |V|$ **down to** 1 **do**
  $X \leftarrow$ Unmarkierte Variable in V mit der größten Anzahl an markierten Nachbarn.
  $\pi(X) = i$
  Markiere X.
return π

Suche der maximalen Cliquen Da der Graph jedoch im vorherigen Schritt trianguliert wurde, kann die Menge der maximalen Cliquen in linearer Zeit bestimmt werden [4]. Dazu wird eine perfekte Eliminationsordnung π mit dem *Maximum Cardinality Search*-Algorithmus (MCS, siehe Algorithmus 2) bestimmt, welche anschließend genutzt wird, um die maximalen Cliquen des Graphen zu bestimmen. In einer perfekten Eliminationsordnung bildet jeder Knoten mit seiner Nachbarschaft eine Clique $\mathcal{C} = \{\{v \cup \mathcal{N}(v)\} \mid x \in \pi\}$.

Erstellung des Baums Im vorherigen Schritt wurden die maximalen Cliquen des Basisgraphen identifiziert, welche als Knoten des Junction-Trees dienen. Um die Kompilierungsphase abzuschließen, müssen die Knoten so verbunden werden, dass die Running Intersection Property für jede Clique erfüllt ist. Jensen hat in seiner Arbeit bewiesen [31], dass es genügt, wenn zwischen allen Cliquen paarweise die Schnittmenge der Variablen berechnet wird und ihre Kardinalität als Gewichte für einen maximalen Spannbaum genutzt wird. Formal: Sei $\mathcal{C}$ die Menge aller maximalen Cliquen des Basisgraphen. Die Kantengewichte w werden wie folgt errechnet

$$\forall\, C_i \in \mathcal{C}.\, \forall C_j \in \mathcal{C}.\, w_{ij} = |C_i \cap C_j|.$$

Anschließend werden diese Gewichte genutzt, um mit Hilfe des Algorithmus von Kruskal [40] einen maximalen Spannbaum zu errechnen. Der berechnete Spannbaum ist der gesuchte Junction-Tree.

2.4.2 Message-Passing-Phase

Nachdem in der Kompilierungsphase der Junction-Tree berechnet wurde, soll dieser in der Message-Passing-Phase genutzt werden, um die zuvor erwähnten Inferenzaufgaben zu lösen. Der Algorithmus besteht aus den folgenden drei Schritten: Initialisierung, Message-Passing sowie der Normalisierung. Während der Initialisierung werden zunächst neue Potentialtabellen für die Cliquen und Separatoren des Junction-Trees angelegt und mit den Potentialen des Basisgraphen aufgefüllt. Während des darauffolgenden Message-Passing werden die Potentiale so manipuliert, dass sie anschließend proportional zur gemeinsamen Wahrscheinlichkeitsdichte sind

$$\psi_C(\boldsymbol{x}) \propto \mathbb{P}_C(\boldsymbol{x}).$$

Im letzten Schritt werden die Potentialtabellen normalisiert, so dass die Potentiale der Wahrscheinlichkeitsverteilung entsprechen

$$\psi_C(\boldsymbol{x}) = \mathbb{P}_C(\boldsymbol{x}).$$

Zunächst wird die Standardvariante zur Berechnung der Randwahrscheinlichkeiten vorgestellt. Anschließend wird erläutert, wie die anderen Inferenzaufgaben mit diesem Algorithmus gelöst werden können.

Initialisierung der Potentiale

Im ersten Schritt der Message-Passing-Phase müssen neue Potentialtabellen für die Cliquen des Junction-Trees angelegt und initialisiert werden. Für eine Clique C besitzt die neue Potentialtabelle ψ_C die Größe $\prod_{v \in \mathcal{C}} \mathcal{X}_v$. Hier wird deutlich, wo die hohe Laufzeit des Verfahrens herkommt. Um eine oder mehrere Variablen zu marginalisieren, muss über eine exponentiell wachsende Menge an Potentialen aufsummiert werden. Während der

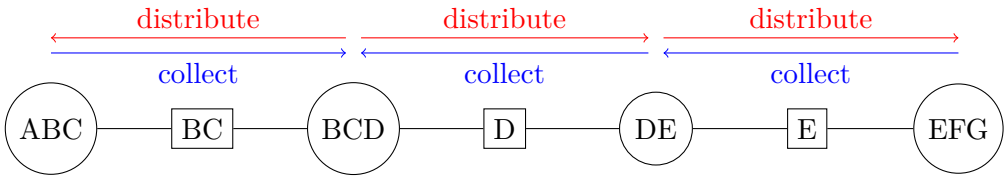


Abbildung 2.4: Diese Abbildung zeigt den Ablauf des Message-Passing in einem Junction-Tree. Dabei wurde die Clique BCD als Wurzel gewählt. Zunächst werden die Nachrichten von den Blättern während der *Collect-Evidence*-Phase in Richtung der Wurzel propagiert. Sobald die Nachrichten der Cliquen ABC und DE an der Wurzel eingetroffen sind, beginnt die *Distribute-Evidence*-Phase, in welcher die Nachrichten zurück zu den Blättern gesendet werden.

Initialisierung werden alle Potentiale der Separatoren mit dem Wert eins initialisiert. Zur Initialisierung der Cliquenpotentiale, wird zu jedem Potential des Basisgraphen eine Clique gesucht, welche alle Variablen des Potentials enthält. Falls einer Clique mehrere Potentiale des Basisgraphen zugeordnet werden, werden diese zusammen multipliziert.

Message Passing

Da es sich bei dem Junction-Tree um einen Baum handelt, liefert der Belief-Propagation-Algorithmus auf einem Junction-Tree die exakte Marginalverteilung aller Knoten. Wie bereits zuvor wird ein beliebiger Knoten als Wurzel ausgewählt, woraufhin die Nachrichten von den Blättern hin zur Wurzel und anschließend in umgekehrter Reihenfolge zurück zu den Blättern propagiert wird (siehe Abbildung 2.4). Im Kontext von Junction-Trees wird die Phase, in der die Nachrichten von den Blättern zur Wurzel propagiert werden, *Collect-Evidence* (siehe Algorithmus 4) genannt. Die Rückrichtung wird als *Distribute-Evidence* bezeichnet (siehe Algorithmus 5). Die Nachrichten müssen leicht abgeändert werden, um den Fall zu behandeln, dass eine Variable in mehreren Cliquen vorkommen kann. Diese Variante des Belief-Propagation-Algorithmus nennt sich Shafer-Shenoy [56], in welcher die Nachrichten wie folgt definiert sind

$$m_{C_i \to C_j}(\boldsymbol{x}_j) = \sum_{C_i \setminus S_{ij}} \psi_{C_i}(\boldsymbol{x}_j) \prod_{n \in \mathcal{N}(C_i \setminus C_j)} m_{n \to C_i}(\boldsymbol{x}_j). \tag{2.24}$$

Andersen *et al.* haben parallel eine zeiteffizientere Variante des Message-Passing entwickelt, den Hugin-Algorithmus [2] (siehe Algorithmus 3). Die Nachricht einer Clique C_i an die Clique C_j über einen Separator S_{ij} besteht hier aus mehreren Schritten: Zunächst wird die alte Potentialabelle ψ_S des Separators gesichert, woraufhin die neue Tabelle des Separators ψ_S^* durch die Marginalisierung der Clique C_i berechnet wird

$$\psi^*_{S_{ij}}(\boldsymbol{x}) = \sum_{\boldsymbol{c} \in C_i} \psi_{C_i}(\boldsymbol{c}, \boldsymbol{x}). \tag{2.25}$$

Letztendlich wird die Potentialtabelle der Clique C_j aktualisiert, indem ein Element dieser mit dem Verhältnis zwischen der alten und neuen Potentialtabelle des Separators multipliziert wird

$$\psi_{C_j}(\boldsymbol{x}) = \frac{\psi^*_{S_{ij}}(\boldsymbol{x})}{\psi_{S_{ij}}(\boldsymbol{x})}\psi_{C_j}(\boldsymbol{x}). \tag{2.26}$$

Dies ist äquivalent zu den Shafer-Shenoy-Nachrichten, da die Produkte der Formel 2.24 in den Potentialtabellen der Separatoren zwischengespeichert werden. Der Shafer-Shenoy-Algorithmus ist besser dazu geeignet, viele ähnliche Anfragen zu beantworten, wohingegen der Hugin-Algorithmus zur schnelleren Berechnung der Randverteilung genutzt werden kann [52].

Algorithmus 3 Message

Eingabe: Clique B, Clique C

1: $S \leftarrow$ `Separator`(B, C)
2: **for** $s \in \mathcal{X}_S$ **do**
3: $\quad\psi^*_S(s) \leftarrow 0$
4: $\quad$**for** $x \in \mathcal{X}_B \setminus \mathcal{X}_S$ **do**
5: $\quad\quad\psi^*_S(s) \leftarrow \psi^*_S(s) + \psi_B(s, x)$
6: $\quad\psi^*_C(s) = \psi_C(s) \cdot \dfrac{\psi^*_S(s)}{\psi_S(s)}$

Algorithmus 4 Collect

Eingabe: Knoten v

1: **for** each child u of v **do**
2: $\quad$`Collect`(u)
3: $\quad$`Message`(v, u)

Algorithmus 5 Distribute

Eingabe: Knoten v

1: **for** each child u of v **do**
2: $\quad$`Message`(v, u)
3: $\quad$`Distribute`(u)

Normalisierung

Nachdem das Message-Passing abgeschlossen wurde, sind die Werte in den Potentialtabellen proportional zur gemeinsamen Verteilung der Variablen in den jeweiligen Cliquen. Mit diesen Potentialen können jedoch nur Anfragen, wie z.B. welches Event wahrscheinlicher ist, beantwortet werden. Um exakte Auskunft über bedingte und Randwahrscheinlichkeiten beantworten zu können, müssen die Potentiale normalisiert werden. Dazu wird für jede Clique C des Junction-Trees die Summe über alle Potentiale errechnet und jedes Potential der Clique anschließend durch die Summe dividiert (siehe Algorithmus 6).

Bedingte Wahrscheinlichkeiten Der Junction-Tree-Algorithmus kann ebenfalls zur Berechnung von bedingten Wahrscheinlichkeiten genutzt werden. Sei E eine Teilmenge der Variablen des Graphen und e eine Belegung dieser. Die einzige Änderung, die getroffen

a	b	c	ψ_{abc}
0	0	0	10
0	0	1	11
0	1	0	3
0	1	1	8
1	0	0	13
1	0	1	2
1	1	0	1
1	1	1	2

b	c	ψ^*_{bc}
0	0	23
0	1	13
1	0	4
1	1	10

Abbildung 2.5: Diese Abbildung demonstriert das Update der Potentialtabelle ψ_{BC} des Separators BC während der Nachricht des Knoten ABC an sein benachbartes Cluster BCD. Die linke Tabelle zeigt die Potentialtabelle der Clique ABC, wohingegen rechts die aktualisierte Tabelle ψ^*_{BC} des Separators zu sehen ist. Zur Berechnung des neuen Wertes der Separatortabelle wird über alle passenden Zustände marginalisiert. Die jeweiligen zusammen summierten Elemente sind in dieser Abbildung in der selben Farbe markiert.

Algorithmus 6 Normalisierung

Eingabe: Junction-Tree $\mathcal{T}$

1: **for** $C \in \mathcal{C}(\mathcal{T})$ **do**
2: $\quad Z_C \leftarrow \sum_{\boldsymbol{x} \in \mathcal{X}_C} \psi_C(\boldsymbol{x})$
3: $\quad$ **for** $\boldsymbol{x} \in \mathcal{X}_C$ **do**
4: $\quad\quad \psi_C(\boldsymbol{x}) \leftarrow \dfrac{\psi_C(\boldsymbol{x})}{Z_C}$

werden muss, ist, dass während der Initialisierung nur die Potentiale des Basisgraphen in den Junction-Tree geladen werden, welche konsistent mit den Beobachtungen e sind.

Maximum-a-posteriori Zustand Mit dem Junction-Tree-Algorithmus lässt sich genauso wie mit dem Belief-Propagation-Algorithmus der Maximum-a-posteriori Zustand berechnen, indem die Summation in der Gleichung 2.25 durch die `max`-Operation ersetzt wird. Beobachtungen können ebenfalls in Betracht gezogen werden, wenn während der Initialisierung, wie im vorherigen Abschnitt, nur die Potentiale geladen werden, welche den Beobachtungen entsprechen.

Normalisierungskonstante Nachdem die Message-Passing-Phase abgeschlossen wurde, können die berechneten Potentiale dazu genutzt werden, die Normalisierungskonstante effizient zu berechnen, ohne die Summation über den gesamten Zustandsraum durchzuführen. Dazu werden die Verteilungen des Markov Random Fields und des Junction-Trees gleichgesetzt und nach Z aufgelöst. In der folgenden Gleichung werden zur Vermeidung von Notationsüberlagerungen die Potentiale des Junction-Tree mit ϕ notiert

$$
\begin{aligned}
\log(\mathbb{P}_{MRF}(\boldsymbol{x})) &= \log(\mathbb{P}_{JT}(\boldsymbol{x})) \\
\log\left(\frac{1}{Z}\prod_{C\in\mathcal{C}(G)}\psi_C(\boldsymbol{x})\right) &= \log\left(\frac{\prod_{C\in\mathcal{C}(\mathcal{T})}\phi_C(\boldsymbol{x})}{\prod_{S\in\mathcal{S}(\mathcal{T})}\phi_S(\boldsymbol{x})}\right) \\
-\log(Z) + \sum_{C\in\mathcal{C}(G)}\log(\psi_C(\boldsymbol{x})) &= \sum_{C\in\mathcal{C}(\mathcal{T})}\log(\phi_C(\boldsymbol{x})) - \sum_{S\in\mathcal{S}(\mathcal{T})}\log(\phi_S(\boldsymbol{x})) \\
A = \log(Z) &= \sum_{C\in\mathcal{C}(G)}\log(\psi_C(\boldsymbol{x})) - \left(\sum_{C\in\mathcal{C}(\mathcal{T})}\log(\phi_C(\boldsymbol{x})) - \sum_{S\in\mathcal{S}(\mathcal{T})}\log(\phi_S(\boldsymbol{x}))\right)
\end{aligned}
$$

Die Logpartitionskonstante lässt sich also als die Differenz zwischen der Logwahrscheinlichkeit des Junction-Trees sowie der nicht normalisierten Logwahrscheinlichkeit des Markov Random Fields errechnen.

Kapitel 3

Parallele Programmierung

Von 1986 bis 2002 ist die Taktrate von Prozessoren durchschnittlich um 50% pro Jahr gestiegen [51], wohingegen die Performanz seit 2002 jährlich nur noch um 20% ansteigt. Die Ursache für den Einbruch im jährlichen Anstieg der Performanz ist die steigende Anzahl an Transistoren pro Fläche auf dem Chip [29]. Diese können bei steigender Taktrate nicht ausreichend gekühlt werden. Um diesem Problem entgegenzuwirken, geht der Trend seit einigen Jahren dahin, mehrere Kerne pro Prozessor oder sogar mehrere Prozessoren pro Rechner zu nutzen. Seit geraumer Zeit werden außerdem hochparallele Koprozessoren, wie *Field Programmable Gateway Arrays* (FPGAs) oder Grafikkarten (GPUs) eingesetzt.

Die parallele Programmierung findet in vielen Bereichen, wie beispielsweise dem wissenschaftlichen Rechnen, Simulationen, aber auch der Industrie, Anwendung [24]. Durch die Parallelität ist es möglich, große Probleminstanzen zu lösen, welche auf Einzelkernprozessoren nicht in annehmbarer Zeit terminieren würden. Ebenfalls können geeignete kleinere Probleme deutlich schneller gelöst werden, was insbesondere im Kontext von Echtzeitanwendungen oder den riesigen Datenmengen, die heutzutage anfallen, von Nöten ist.

Allerdings bringt die Programmierung von parallelen Systemen viele neue Herausforderungen mit sich [29]. Zunächst stellt sich die Frage, ob ein gegebenes Problem überhaupt von parallelen Berechnungen profitieren kann. Dazu müssen unabhängige Arbeitsschritte und Datenabhängigkeiten identifiziert werden. Eine Datenabhängigkeit liegt genau dann vor, wenn das Ergebnis einer vorherigen Berechnung benötigt wird. Anschließend muss das Problem in einem der parallelen Programmiermodelle modelliert werden. Zwei weit gebräuchliche Modelle sind der Daten- und der Taskparallelismus. Weist ein gegebenes Problem Datenabhängigkeiten auf, bietet sich der Taskparallelismus an. Dort werden die einzelnen Arbeitspakete als Knoten und die Abhängigkeiten als Kanten eines Graphen modelliert und nacheinander abgearbeitet (siehe Abbildung 3.1). Beim Datenparallelismus hingegen soll die selbe Operation auf jedem Element einer Datenstruktur, wie z.B. eines Arrays, ausgeführt werden. Dazu wird das Array zunächst partitioniert und jeder Prozessor

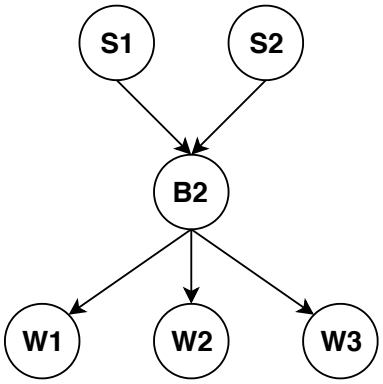


Abbildung 3.1: In dieser Abbildung ist ein Abhängigkeitsgraph mit sechs Arbeitspaketen zu sehen. Zu Beginn der Ausführung können die beiden Pakete S1 und S2 parallel bearbeitet werden, welche abgearbeitet sein müssen, bevor mit der Aufgabe B2 fortgeführt werden kann. Die Pakete W1, W2 und W3 dürfen parallel bearbeitet werden, jedoch erst nachdem die Aufgabe B2 erfolgreich abgeschlossen ist.

erhält eine Partition, die dieser abarbeiten muss (siehe Abbildung 3.2). Neben den zwei genannten Programmiermodellen gibt es noch weitere Modelle, wie z.B. dem Master-Slave- oder Server-Client-Modell, die in dieser Arbeit jedoch nicht verwendet und deshalb nicht näher vorgestellt werden. Weitere Informationen über diese Modelle sind in dem Buch *Introduction to Parallel Computing* [24] zu finden. In der Praxis werden oftmals hybride Modelle, also eine Kombination aus verschiedenen Ansätzen genutzt.

Eine weitere Herausforderung sind die sogenannten *Race Conditions*. Eine Race Condition tritt auf, falls zwei verschiedene Prozesse oder Threads auf die selbe Ressource zugreifen und das Ergebnis abhängig von der Reihenfolge der Ausführung ist. Um Race Conditions zu vermeiden, müssen die Zugriffe koordiniert und synchronisiert werden. Dabei ist es wichtig, dass so selten wie möglich aber so oft wie nötig synchronisiert wird, da das Programm den geschützten Bereich sequentiell ausführt. Außerdem muss die Synchronisation so implementiert werden, dass kein Deadlock auftreten kann. Im Kontext von Systemen mit verteiltem Speicher muss außerdem geklärt werden, welche Daten wo gespeichert werden sollen und wie die Kommunikation möglichst effizient bewerkstelligt werden kann.

Des Weiteren ist die Skalierbarkeit der Implementierung wichtig. Ein theoretisches Modell zur Bestimmung des maximalen Speedups ist das Amdahlsche Gesetz [1], welches besagt, dass der maximal möglich erreichbare Speedup S durch den sequentiellen Anteil der Arbeit $(1 - P)$ beschränkt ist. Die Formel zur Berechnung des maximalen Speedups bei N Prozessoren lautet

$$S = \frac{1}{(1 - P) + \frac{P}{N}}.$$

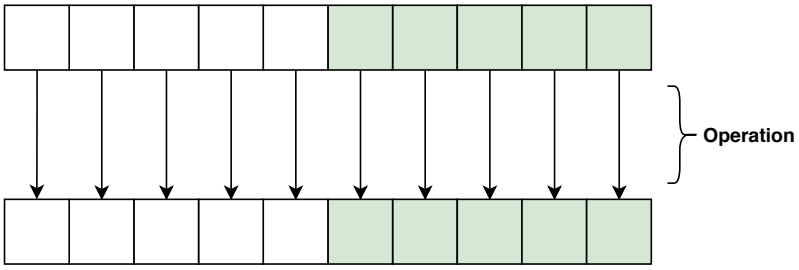


Abbildung 3.2: Schematische Darstellung des Datenparallelismus. In der oberen Zeile ist das Eingabearray zu sehen. Auf jedem Element des Arrays soll die selbe Operation ausgeführt und das Ergebnis zurückgeschrieben werden. Um die Berechnung zu beschleunigen, wird das Eingabearray in zwei Subarrays partitioniert (farblich angedeutet). Diese werden anschließend von verschiedenen Kernen abgearbeitet.

3.0.1 Beispiel. Sei P = 80% der parallelisierbare Anteil eines Programms, dann ist der maximal erreichbare Speedup durch das Amdahlsche Gesetz – selbst bei einer unbegrenzten Menge an Prozessoren – begrenzt auf

$$S = \lim_{N\to\infty} \frac{1}{(1-0.8) + \frac{0.8}{N}} = \lim_{N\to\infty} \frac{1}{(0.2) + \frac{0.8}{N}} = \frac{1}{0.2+0} = 5.$$

In dieser Arbeit soll eine verteilte, parallele Implementierung der Junction-Tree-Inferenz implementiert werden. Die einzelnen Operationen und Nachrichten sollen, wie in vorangegangenen Arbeiten, mit Hilfe von Nvidia CUDA beschleunigt und dabei verteilt auf mehreren GPUs berechnet werden. Zur Verwaltung der Arbeitspakete wurde die *OpenMP*-API verwendet. In den folgenden Abschnitten werden diese Frameworks näher erläutert.

3.1 Nvidia CUDA

Seit den späten 1980er Jahren nahm die Popularität der Betriebssysteme mit graphischer Oberfläche rasant zu [54]. Zeitgleich wurden große Fortschritte in der 3D-Computergrafik gemacht, was zur Veröffentlichung von beliebten Spielen wie Doom oder Quake führte. Jedoch hatten diese Spiele für damalige Verhältnisse starke Hardwareanforderungen und die Grafikberechnungen wurden immer anspruchsvoller. Dies hatte zur Folge, dass die Berechnungen beschleunigt werden mussten. Deshalb wurden dedizierte *Graphics Processing Units* (GPUs) entwickelt, welche zur Beschleunigung von Berechnungen auf 3D-Bildern genutzt wurden. Die Chips der GPUs wurden hoch parallel designt, so dass sie in Realzeit alle Pixel eines Bildes parallel manipulieren können. In der Abbildung 3.3 ist ein Vergleich der *Giga-Flops* zwischen GPUs und CPUs zu sehen. Dabei ist deutlich zu erkennen, dass die GPU mehr Rechenoperationen im Vergleich zur CPU in der selben Zeit ausführen kann.

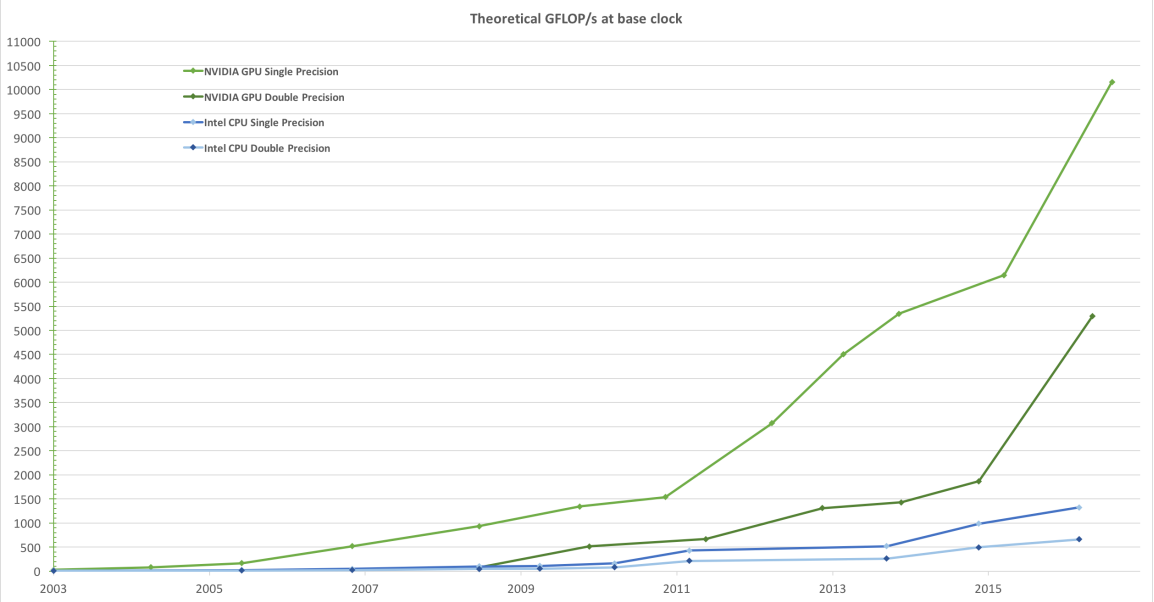


Abbildung 3.3: Diese Abbildung zeigt die theoretischen *Giga-Flops* der CUDA-Hardware in Abhängigkeit des Jahres. Quelle: [50].

Diese Fortschritte der Rechenleistung boten ebenfalls großes Potential zur Beschleunigung von wissenschaftlichen Berechnungen und Simulationen. Jedoch war die Programmierung der GPUs zu dem damaligen Zeitpunkt sehr aufwendig, da sie nur über Grafik-APIs angesprochen werden konnten. Daraufhin hat der GPU-Hersteller NVIDIA im Jahre 2006 [1] die *Compute Unified Device Architecture* [2] (CUDA) vorgestellt, welche eine Erweiterung der *C*-Programmiersprache ist und die *General-purpose computing on graphics processing units* (GP-GPU) Programmierung auf NVIDIA Grafikkarten ermöglicht [11]. Seitdem wird CUDA erfolgreich in vielen Domänen, wie beispielsweise der Medizin [34] und dem wissenschaftlichen Rechnen eingesetzt [54]. Aufgrund ihres speziellen Programmierparadigmas, welches im folgenden Abschnitt näher erläutert wird, können GPUs jedoch nicht für alle Probleme verwendet werden. Außerdem haben sie den Nachteil, dass sie mit einem hohen Energieverbrauch einhergehen.

Eine Alternative zu CUDA wäre die *Open Computing Language* (OpenCL) [58] gewesen, welche nicht nur die Programmierung für GPUs, sondern auch für andere Koprozessoren ermöglicht. Allerdings wurde sich gegen die Nutzung von OpenCL entschieden, da die Zielplattform NVIDIA GPUs sind und CUDA außerdem einen größeren Umfang an Bibliotheken und Tools besitzt.

[1] `https://www.nvidia.com/en-us/about-nvidia/corporate-timeline/` (Zuletzt besucht am 21.11.2019.)

[2] `https://developer.nvidia.com/cuda-toolkit` (Zuletzt besucht am 21.11.2019.)

3.1.1 Programmiermodell

CUDA stellt ein Programmiermodell bereit, welches sich insbesondere für Probleme mit einem hohen Anteil an Datenparallelismus anbietet. Zur Parallelisierung eines Problems, müssen zunächst die Stellen des Programmcodes identifiziert werden, die hierzu gut geeignet sind. Im Falle von Datenparallelismus sind dies vorwiegend Schleifen, deren Ausführungen unabhängig sind. Zur Beschleunigung wird anschließend ein *Kernel*, so nennt sich eine Funktion in der CUDA-API, implementiert. Kernel unterscheiden sich dabei von herkömmlichen Funktionen, denn ein Kernel wird so designt, dass er eine Aufgabe auf genau einem Element ausführt. Im Vergleich zu CPU-Funktionen wird beim Aufruf eines Kernels dieser nicht einmal, sondern n Instanzen von diesem gestartet. Jeder Instanz wird dabei ein eigener Thread zugewiesen, dem ein Index zugeordnet ist. Zur Laufzeit fragt die Instanz ihren Index ab und führt die spezielle Aufgabe auf dem ihm über den Index zugewiesenen Element aus.

Die Threads werden in sogenannten Threadblöcken organisiert. Ein Threadblock ist eine logische Einheit und kann bis zu 1024 Threads enthalten. Threads innerhalb eines Threadblocks können effizient untereinander über schnellen, geteilten Speicher kommunizieren. Außerdem können Threads blockintern effizient synchronisiert werden. Da die Anzahl an Threads pro Block auf 1024 limitiert ist, können für größere Probleminstanzen mehrere Blöcke gestartet werden. Die Konfiguration der Problemgröße auf Threads und Blöcke nennt sich *Grid* und ist sowohl vom Algorithmus als auch der Problemgröße abhängig. Bestenfalls besteht die Gridstruktur aus so vielen Threads, dass mehr Threads als Kerne gestartet werden. Nur so kann die GPU maximal ausgelastet werden, indem die Latenz der Speicherzugriffe durch andere rechenbereite Threads versteckt wird.

CUDA-Programme werden in Host- und Devicecode aufgeteilt. Der Devicecode besteht dabei aus einem oder mehreren Kerneln, die vom Host aus aufgerufen werden. Bevor der Kernel gestartet werden kann, muss sich ein CPU-Thread auf dem Host um die Verwaltung der Daten kümmern. Zunächst muss der Host auf dem Device Speicher allokieren und die Daten, auf denen die Operationen ausgeführt werden sollen, in diesen kopieren. Anschließend muss eine zum Algorithmus und zur Eingabegröße passende Gridstruktur bestimmt werden und der Kernel aufgerufen werden. Ab diesem Zeitpunkt beginnt die GPU das Rechnen. Da die Kernelaufrufe asynchron sind, kann die CPU parallel zur GPU Funktionen ausführen. Alternativ kann der Hostthread auch blockieren und auf den Abschluss der Berechnungen warten. Sobald die Berechnungen auf der GPU abgeschlossen sind, können die Ergebnisse zurück in den Arbeitsspeicher der CPU kopiert werden.

3.1.2 Hardware

Die spezielle CUDA-Hardware unterscheidet sich stark von der herkömmlichen CPU-Architektur (siehe Abbildung 3.4). Während eine CPU aus einigen wenigen Kernen besteht,

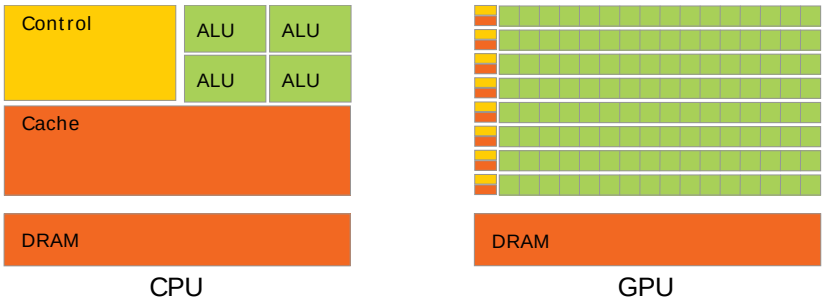


Abbildung 3.4: Diese Abbildung verdeutlicht die Unterschiede zwischen einer GPU und einer CPU. Eine CPU besteht aus wenigen Recheneinheiten (ALU), besitzt dafür jedoch mehr Kontrolllogik und einen riesigen Cache. Die GPU hingegen besteht aus einem Array an Stream-Multiprozessoren, von denen jeder aus einer großen Menge an Recheneinheiten besteht. Quelle: [50].

ist eine GPU als Array von unabhängigen Streaming-Multiprozessoren aufgebaut, von denen jeder aus einer Vielzahl aus Cudakernen besteht. Die NVIDIA GeForce GTX 1080 besitzt insgesamt 2560 Kerne, welche auf 20 Streaming-Multiprozessoren aufgeteilt sind, die jeweils wiederum aus 128 Kernen bestehen [3]. Verglichen mit CPU-Kernen weisen die Cudakerne eine geringere Taktrate auf und stellen weniger Funktionalität bereit, was jedoch durch die schiere Menge an Kernen ausgeglichen wird. Die Streaming-Multiprozessoren führen die Threads immer in sogenannten *Warps*, Gruppen von 32 Threads, aus. Dabei wird das *Single-Instruction-Multiple-Data* (SIMD) Modell angewandt, jeder Thread des Warps führt dieselbe Instruktion aus.

Ihre Daten bekommt eine GPU über den PCI-E-Bus. Diese werden zunächst in den globalen Speicher kopiert, welcher nicht direkt auf den Streaming-Multiprozessoren liegt, wie in Abbildung 3.4 zu sehen ist. Das Lesen und Schreiben aus dem globalen Speicher ist deshalb vergleichsweise langsam, jedoch bietet dieser im Vergleich zu den schnelleren Speicherarten ein Vielfaches der Kapazität. Die GTX 1080 besitzt beispielsweise 8 Gigabyte. Neben dem globalen Speicher gibt es noch zwei Arten von *Read-Only*-Speichern, den Textur- und Konstatenspeicher, welche wie der globale Speicher von allen Threads gelesen werden können. Neben dem globalen Speicher können Threadblöcke *Shared-Memory* anfordern. Shared-Memory ist ein schneller Speicher mit sehr geringer Latenz, welcher von allen Threads eines Blocks les- und schreibbar ist. Außerdem besitzt jeder Thread Zugriff auf die schnellen Register eines Chips. Sollten die Register nicht ausreichen, um die privaten Daten eines Threads zu speichern, kann der Thread auf lokalen Speicher zurückgreifen. Jedoch ist dieser nicht wirklich "lokal", da es sich dabei um den globalen Speicher handelt. Lokal bedeutet in diesem Falle, dass nur der Thread Zugriff auf diesen Speicher hat. Neuere

[3] `https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf` (Zuletzt besucht am 21.11.2019.)

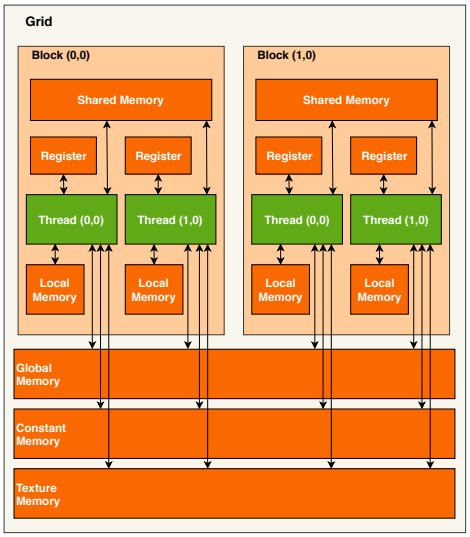


Abbildung 3.5: Schematische Darstellung der CUDA Speicherhierarchie an einem Grid aus zwei Blöcken mit je zwei Threads. Im unteren Teil der Grafik sind die verschiedenen Arten des globalen Speicher zu sehen, auf die jeder Thread Zugriff hat. Weiterhin ist zu sehen, dass jeder Thread Zugriff auf seine Register und lokalen Speicher besitzt. Die letzte Speicherart ist der Shared-Memory, welcher blockweise verfügbar ist. Alle Threads eines Blocks haben Lese- und Schreibezugriff auf denselben Speicher. In Anlehnung an den CUDA Programming Guide V2 [4].

Generationen der NVIDIA GPUs besitzen außerdem L1- und L2-Caches, welche jedoch verglichen mit einer modernen CPU weniger Kapazität anbieten.

3.2 OpenMP

OpenMP ist eine API zur *Shared-Memory*-Programmierung von Mehrkernprozessoren [16]. Die API bietet ein einfaches und flexibles Interface zur Entwicklung von parallelen Anwendungen an. Dazu werden einzelne Codepfade, wie beispielsweise Schleifen, die parallel ausgeführt werden sollen, mit sogenannten Compilerdirektiven annotiert. Während der Programmübersetzung generiert der Compiler den entsprechenden Code zu den Direktiven, um die Pfade zu parallelisieren. Außerdem können einfach parallele Regionen erzeugt werden, welche zur Orchestrierung von mehreren GPUs in einer Maschine genutzt werden können. Zur Orchestrierung von mehreren GPUs, die auf verschiedene Hosts verteilt sind, bietet sich die Nutzung einer Implementierung des *Message-Passing-Interfaces* (MPI) an [19].

[4] `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf` (Zuletzt besucht am 22.11.2019.)

Kapitel 4

Implementierung

Das Ziel dieser Arbeit ist die Entwicklung einer parallelen und verteilten GPU-Implementierung des Hugin-Algorithmus zur Junction-Tree-Inferenz. Diese soll dabei helfen, größere Problemstellungen verteilt über mehrere GPUs lösen zu können und die Inferenz insgesamt zu beschleunigen. In diesem Kapitel wird eine solche Implementierung vorgestellt, welche auf Basis der PX-Bibliothek[1] entwickelt wurde. Die Bibliothek ist in der Programmiersprache C++ geschrieben und bietet die Modellierung diskreter, ungerichteter graphischer Modelle an. In der Software werden die Modelle als Mitglied der Exponentialfamilie implementiert und sind paarweise über die Kanten parametrisiert. Die Software beinhaltet bereits einen Junction-Tree-Algorithmus, welcher als Grundlage für diese Arbeit genutzt wird. Das Kapitel ist wie folgt aufgebaut: Zunächst werden in Abschnitt 4.1 allgemeine Implementierungsdetails besprochen, während der darauffolgende Abschnitt 4.2 die Beschleunigung der einzelnen Operationen des Message-Passing erläutert. Anschließend wird näher dargelegt, wie die einzelnen Operationen verteilt auf unterschiedlichen GPUs berechnet werden. Zuletzt wird ein hybrider Ansatz zum modellbasierten Scheduling vorgestellt.

4.1 Allgemeine Details

Die folgenden Abschnitte legen allgemeine Details der Implementierung näher dar. In den zwei ersten Teilabschnitten wird näher erläutert, wie ein Junction-Tree als flache Datenstruktur gespeichert werden kann (4.1.1) und wie das Message-Passing durch die Nutzung von Indizes beschleunigt werden kann (4.1.2) . Die beiden letzten Abschnitte widmen sich den Präzisionsproblemen (4.1.3) und der Datenhaltung bzw. Optimierung der Kopieroperationen (4.1.4).

[1] `https://randomfields.org/px` (Zuletzt besucht am 21.11.2019.)

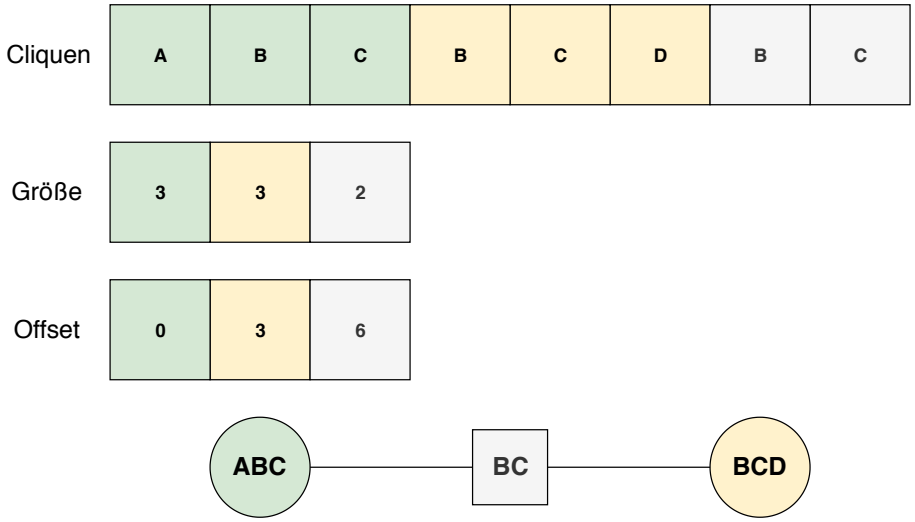


Abbildung 4.1: Diese Abbildung zeigt, wie der abgebildete Junction-Tree mit Hilfe dreier Arrays gespeichert werden. Das erste Array speichert pro Clique, welche Knoten in ihr enthalten sind. Um die Cliquen voneinander trennen zu können, wird ebenfalls ein zweites Array benötigt, welches die jeweilige Größe der Clique speichert. Das Offset-Array wird benötigt, um direkt zu den Knoten einer Clique springen zu können und wird nach dem folgenden Schema gebildet: Offset[0] = 0, Offset[i] = $\sum_{j=0}^{i}$ Größe[j] + Offset[$i-1$].

4.1.1 Flache Datenstrukturen

Der in der PX-Bibliothek implementierte Junction-Tree-Algorithmus nutzt einige Datenstrukturen der Standardbibliothek, welche nicht für die CUDA-Programmierung geeignet sind. Für die Programmierung von Grafikkarten eignen sich flache Strukturen wie Arrays am besten. Deshalb musste die grundlegende Datenstruktur zunächst so angepasst werden, dass diese nur noch auf Arrays arbeitet. Junction-Trees lassen sich gut durch drei Arrays beschreiben: Das erste Array speichert für jede Clique, welche Knoten in ihr enthalten sind, das zweite Array speichert die jeweilige Cliquengröße und das letzte Array speichert für jede Clique den Index des ersten Knoten im ersten Array (siehe Abbildung 4.1).

4.1.2 Index-Mapping-Vektoren

Eine gängige Strategie zur Verkürzung der Laufzeit der Inferenzroutine ist die Nutzung eines *Speicher-Laufzeit Tradeoffs* [30]. Der Speicher-Laufzeit Tradeoff ist eines der klassischen Probleme in der Informatik und bezeichnet den Fall, dass die Laufzeit eines Programms im Austausch gegen einen höheren Speicherverbrauch verkürzt werden kann. Um während einer Nachricht den neuen Wert eines Eintrages des Separatorknoten zu berechnen, müssen alle Werte aus der Tabelle ψ_B, die konsistent mit der Konfiguration des Eintrages der Tabelle ψ_S sind, aufsummiert werden (siehe Abbildung 2.5). Allerdings werden die Potentialtabellen aus Effizienzgründen nur als Array abgespeichert, weshalb diese nicht durch

Konfigurationen indizierbar sind und nur eine implizite Abbildung zwischen den Indizes und den Konfigurationen besteht. Deshalb muss zur Aktualisierung des j-ten Separatoreintrages der Index j zunächst als Zustandsstring dekodiert werden. Ein Zustandsstring S ist ein Array, in dem jedes Element einem Knoten der zugehörigen Clique zugeordnet ist und einen der Werte des Zustandraumes des Knoten annehmen kann. Zu einem Index j kann der i-te Eintrag S_i des Zustandsstrings S mit der folgenden Formel berechnet werden [62]

$$S_i = \left\lfloor \frac{t-1}{\prod_{k=1}^{i-1} \mathcal{X}_k} \right\rfloor \% \mathcal{X}_i$$

Falls alle Variablen einer Clique den selben Zustandsraum k besitzen, wäre der Zustandsstring die Darstellung des Index im Zahlensystem der Basis k. Nachdem der j-te Index des Separators in den zugehörigen Zustandsstring überführt wurde, müssen alle Konfigurationen der Clique B, die konsistent mit dem Separator sind, gebildet und als Index kodiert werden. Um die konsistenten Zustände zu erhalten, werden die Variablen des Separators fix gehalten, während über die restlichen Variablen und alle annehmbaren Zustände iteriert wird. Ein Zustandsstring S einer Clique C kann über die folgende Formel als Index j der Potentialtabelle ψ_C kodiert werden [62]

$$j = 1 + \sum_{i=1}^{|C|} S_i \prod_{j=1}^{i-1} \mathcal{X}_j$$

Letztendlich ergibt sich das neue Element der Separatortabelle ψ_S als die Summe der Potentiale konsistenten Zustände der Potentialtabelle ψ_C.

Diese Berechnungen nehmen insbesondere bei Cliquen mit einer großen Anzahl an Variablen oder riesigen Zustandsräumen einen hohen Anteil der Laufzeit in Anspruch. Außerdem werden die Zuweisungen von Zustand zu Index pro Inferenzroutine einmal in der Collect-Evidence- und ein weiteres Mal in der Distribute-Evidence-Phase benötigt. Um die doppelte Berechnung zu vermeiden, werden zu Beginn der Inferenzroutine Index-Mapping-Vektoren $\mu_{B,S}$ angelegt [30], welche zu einem Index des Separators eine Liste mit den Indizes der konsistenten Potentiale enthalten (siehe Abbildung 4.2). Für jeden Separator und jeden von seinen annehmbaren Zuständen werden zwei Index-Mapping-Vektoren angelegt, einer für jede anliegende Clique. Diese Vektoren werden anschließend cliquenweise gruppiert und in einer Lookup-Tabelle gespeichert. Die Vektoren haben einen großen Einfluss auf die Laufzeit (siehe Abschnitt 5.2), gehen allerdings auch mit einem erheblichen Speicherverbrauch einher. Sei $\mathcal{S}$ die Menge aller Separatoren eines Junction-Trees ,`index_t` der Datentyp, mit welchem die Vektoren abgespeichert werden sollen und `sizeof` eine Funktion, welche den Speicherbedarf eines Datentypen in Bytes zurückgibt, dann benötigt das gesamte Mapping im Hauptspeicher die folgende Anzahl an Bytes

$$\texttt{sizeof}(\text{index\_t}) \cdot \sum_{s \in \mathcal{S}} \sum_{C \in \mathcal{N}(s)} |\psi_C|.$$

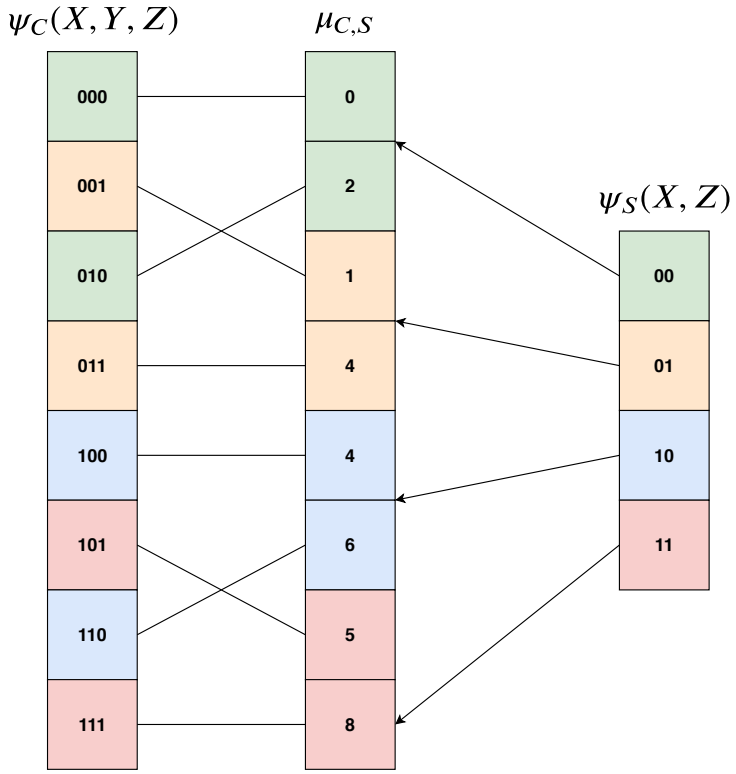


Abbildung 4.2: Diese Abbildung zeigt ein Index-Mapping $\mu_{C,S}$ zwischen der Clique C und dem Separator S. Für jeden Zustand des Separators wird ein Index-Mapping-Vektor angelegt, welcher die Indizes der Potentiale, welche konsistent mit dem Separatorzustand sind, speichert. Konsistente Zustände und deren Einträge im Mapping-Vektor sind mit der selben Farbe hinterlegt.

Die Index-Mapping-Vektoren wachsen also sowohl mit der Anzahl an Knoten als auch der Größe der Potentialtabellen des Junction-Trees, wodurch der Speicherverbrauch sehr schnell ansteigt. Des Weiteren kann die Berechnung der Index-Mapping-Vektoren einige Zeit in Anspruch annehmen, welche jedoch beinahe im ersten Aufruf der Inferenzroutine wieder ausgeglichen wird. Außerdem kann die Berechnung der Indizes sehr einfach parallelisiert werden: für jeden Separator können die Mappings zu jedem seiner anliegenden Cliquen parallel ausgeführt werden. In dieser Arbeit wird OpenMP zur Parallelisierung der Vorberechnung genutzt. Die Berechnung der Vektoren lohnt sich insbesondere dann, wenn die Junction-Tree-Inferenz öfter ausgeführt wird, wie beispielsweise als Subroutine zur Parameterschätzung.

4.1.3 Präzisionsungenauigkeiten

Aufgrund der limitierten Präzision der Hardware kann es bei der Multiplikation vieler kleiner Werte zu Unterläufen kommen [37]. Insbesondere bei der Initialisierung werden oft viele kleine Werte miteinander multipliziert. Um diese Probleme zu vermeiden, werden die

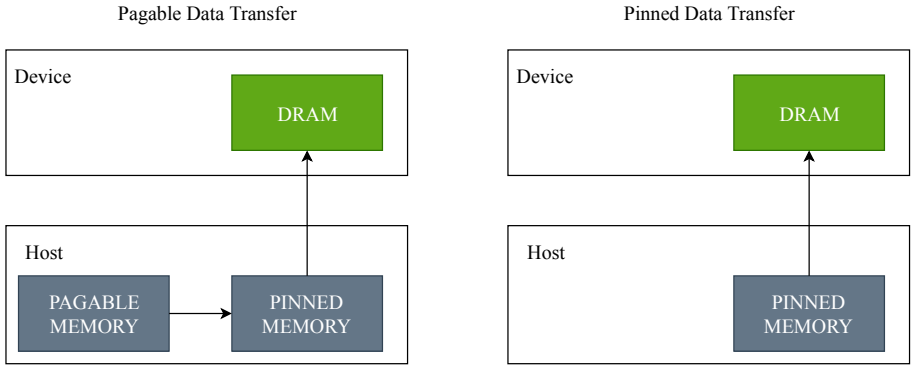


Abbildung 4.3: Wenn ein Speicherbereich nicht als *pinned* markiert ist, muss dieser, bevor er in den Speicher der GPU kopiert wird, in einen Zwischenpuffer kopiert werden. Anschließend kann die *Direct-Memory-Access-Engine* die Daten über den PCI-Bus übertragen. Ist der Speicher direkt als *pinned* markiert, können die zusätzlichen Kopien vermieden werden. Quelle: Eigene Darstellung in Anlehnung an `https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/` (Zuletzt besucht am 21.11.2019.)

Berechnungen stattdessen im Lograum durchgeführt, wodurch die Multiplikationen durch Additionen ersetzt werden. Dabei muss jedoch beachtet werden, dass die Marginalisierung nicht im Lograum durchgeführt werden kann und die gespeicherten Potentiale wieder zurückgerechnet werden müssen. Im Lograum sehen die Aktualisierungen der Potentialtabellen während einer Nachricht wie folgt aus:

$$\psi_S^* = \log\Big(\sum_V \exp(\psi_B)\Big)$$
$$\psi_C^* = \psi_S^* - \psi_S + \psi_C$$

4.1.4 Datenhaltung und Double Buffering

Während der Implementierung musste die Entscheidung getroffen werden, welche Daten wann wo gespeichert werden. Der erste Ansatz war es, alle Daten dauerhaft im GPU-Speicher zu halten, um Kopieroperationen zwischen dem Host und dem Device möglichst zu vermeiden. Da die Potentialtabellen jedoch exponentiell anwachsen, war der GPU-Speicher bereits für Probleme mittlerer Größe vollständig ausgenutzt. Außerdem ist dieser Ansatz in Hinblick auf die Multi-GPU-Implementierung nicht hilfreich, da, sofern die GPUs nicht auf einem PCI-E-Bus liegen, immer alle Potentiale über die CPU zwischen den Devices hin und her kopiert werden müssen. Stattdessen wurde eine Strategie gewählt, in der die Daten in einen statischen und dynamischen Anteil partitioniert wurden, dessen aktueller Stand immer im Hauptspeicher der CPU liegt. Der statische Teil der Daten ist vergleichsweise klein und konstant und liegt deshalb dauerhaft im GPU-Speicher. Er beinhaltet allgemeine Informationen über den Junction-Tree, wie beispielsweise die

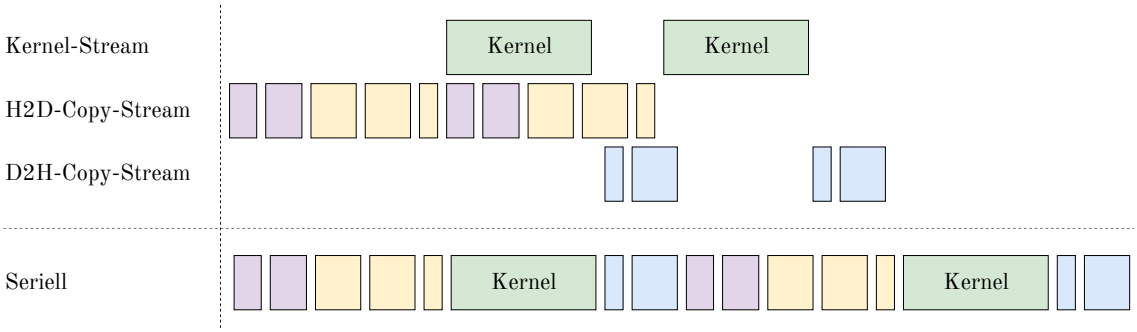


Abbildung 4.4: Die Grafik verdeutlicht die Latenzverringerung zwischen den Kernelaufrufen mit dem Doppelpufferungsansatz. Dieser ist im oberen Teil der Abbildung zu sehen, während unten die serielle Ausführung abgebildet ist. Es ist zu erkennen, dass durch die gleichzeitige Ausführung von Berechnungen und Kopieroperation ein beachtlicher Teil der Laufzeit gegenüber der seriellen Variante eingespart werden konnte.

Zustandsräume und Cliquenzuordnung. Die dynamischen Daten bestehen aus fünf Puffern, drei für die Potentialtabellen der Cliquen, welche zur Berechnung einer Nachricht benötigt werden, sowie zwei weiteren Puffern für die jeweiligen Indizes. Allokiert werden die Puffer für die Vektoren und zwei der drei Potentialtabellen jeweils mit genügend Speicher, um einmal die größte Potentialtabelle speichern zu können. Der dritte Puffer wird so initialisiert, dass er ausreichende Kapazitäten für die größte Separatortabelle enthält. Die Puffer werden immer wiederverwendet, denn die Allokation von GPU-Speicher ist aufwendig und sollte, sofern sie nicht benötigt wird, vermieden werden [50]. Sei m $= \max_{c \in \mathcal{C}} |\psi_C|$ und s $= \max_{s \in \mathcal{S}} |\psi_S|$, dann beläuft sich der Speicherverbrauch des dynamischen Anteil der Daten auf

$$2 \cdot m \cdot \texttt{sizeof}(\text{index\_t}) \; + \; (2 \cdot m + n) \cdot \texttt{sizeof}(\text{value\_t}).$$

Zur Optimierung der Kopiervorgänge werden die Potentialtabellen und Indizes auf der CPU-Seite als sogenanntes *Pinned-Memory* allokiert, wodurch es dem Betriebssystem verboten wird, die Speicherbereiche auszulagern. Dies ermöglicht dem CUDA-Treiber die Nutzung von *Direct Memory Access*, ohne die Daten zuvor in einen Speicherbereich zu kopieren, der *pinned* ist (siehe Abbildung 4.3). Des Weiteren bietet moderne CUDA-Hardware mit Feature *Concurrent copy and kernel execution* die Möglichkeit, parallel zu gestarteten Berechnung Kopieroperationen von und zu der GPU durchzuführen. Dieses Feature wird genutzt, um einen Doppelpufferungsansatz wie in der Arbeit von Jeon *et al.* [32] zu implementieren. Bei einem Doppelpufferungsansatz werden zwei Puffer anstatt einem angelegt, um während der Berechnungen in der Lage zu sein, die Daten für den nächsten Kernel in den Gerätespeicher zu kopieren. Außerdem können die Ergebnisse parallel zur daraffolgenden Berechnung zurückkopiert werden. Diese Technik reduziert die Latenz zwischen den Kernelaufrufen (siehe Abbildung 4.4).

4.2 GPU-Beschleunigung

Die Inferenz besteht aus den drei Operationen Initialisierung, Message-Passing und der Normalisierung. Jede der drei Operationen besteht wiederum aus einzelnen Funktionsaufrufen, die Möglichkeiten zur Parallelisierung bieten. In den folgenden drei Abschnitten wird für jede der Operationen erläutert, wie diese einzelnen Aufrufe auf einer GPU beschleunigt werden können. Wie jeweils die gesamten Operationen durch eine Verteilung der Berechnungen beschleunigt werden können, wird im darauffolgenden Abschnitt 4.3 näher dargestellt.

4.2.1 Initialisierung

Während der Initialisierung werden die Gewichte des zugrunde liegenden Markov Random Fields in die Potentiale des Junction-Trees übertragen. Da die Modelle in der Basisimplementierung über die Kanten parametrisiert sind, wird zu jeder Kante $e = (s, t)$ des Markov Random Fields eine Clique C des Junction-Trees gesucht, welche die beiden Knoten s und t enthält. Anschließend wird über die Menge der annehmbaren Zustände der Clique C iteriert und es wird für jeden der Zustände überprüft, welchen Wert die Knoten s und t annehmen. Das Gewicht, dass sich die Kante im Zustand e im Zustand (s, t) befindet, wird anschließend auf das Potential für den entsprechenden Zustand der Clique aufaddiert. Da die Cliquen des Junction-Trees oftmals einen großen Zustandsraum besitzen, bietet die Parallelisierung der Schleife großes Potential. Für jeden Index wird ein CUDA-Thread gestartet, welcher diesen in einen Zustandsstring überführt, anschließend die Zustände der Knoten s und t aus diesem extrahiert und zuletzt das Gewicht für den Kantenzustand (s, t) auf das entsprechende Potential der Clique C addiert. Da möglicherweise mehrere Threads den selben Wert der Potentialtabelle aktualisieren, wird die CUDA-API-Funktion `atomicAdd(ptr, val)` verwendet, welche garantiert, dass die Addition atomar ausgeführt wird und keine Race-Conditions auftreten.

4.2.2 Message-Passing

Die Beschleunigung einer Nachricht der Clique B über den Separator S an die Clique C basiert auf den vorangegangen Arbeiten von Zheng *et al.* [64, 65]. Zur Berechnung der neuen Werte des Separatorknotens S haben die Autoren die Möglichkeit zur Nutzung von Datenparallelismus erkannt: Die Marginalisierungen zur Berechnung der neuen Einträge der Separatortabelle sind voneinander unabhängig und können parallel ausgeführt werden. Jedoch hängt die Parallelisierbarkeit nur von der Zustandsgröße des Separators ab, wobei weitere Untersuchungen der Autoren zeigen, dass die Separatoren vieler Junction-Trees nur kleine Zustandsräume besitzen. Dieser Fall tritt ein, wenn die Cliquen B und C wenig gemeinsame Variablen besitzen und resultieren in wenig parallelen Berechnungen. Da

GPUs aber *Manycore*-Prozessoren sind, werden diese nicht vollständig ausgenutzt. Um dieses Problem zu umgehen, wurde eine zusätzliche Art der Parallelisierung eingeführt: Pro Separatorzustand werden die zu berechnenden Summen in Teilsummen zerlegt, welche jeweils von einer Menge an Threads berechnet werden. Diese Teilsummen werden daraufhin im Shared-Memory zusammengeführt und ergeben den neuen Wert des Separators für den jeweiligen Zustand. Diese können dann wiederum genutzt werden, um die entsprechenden Zustände der Clique C zu aktualisieren. In dieser Implementierung wird pro Zustand des Separators ein Block mit t Threads gestartet, welche jeweils die Teilsummen errechnen. Möglicherweise kann noch mehr Parallelität genutzt werden, indem die Teilsummen nicht nur pro Thread, sondern auch blockweise berechnet werden. Ab einer gewissen Größe der Separatoren werden jedoch mehr Threads als Kerne gestartet und das Device ist komplett belegt.

4.2.3 Normalisierung

Zur Normalisierung der Potentiale einer Clique müssen zunächst alle Potentiale aufsummiert werden. Anschließend muss jedes Potential durch diese Summe dividiert werden. Beide Operationen können mit Hilfe einer GPU beschleunigt werden. Die Summation kann in die Berechnung von Teilsummen zerlegt werden, welche anschließend auf einen Wert reduziert werden. In dieser Arbeit werden, sofern die Clique groß genug ist, zwei Blöcke pro Streaming-Multiprozessor gestartet, welche jeweils eine Teilsumme berechnen. Wenn jeder Block aus t Threads besteht, muss jeder Thread bei einer Cliquengröße n $\frac{n}{2 \cdot \#\mathrm{SM} \cdot t}$ Elemente aufsummieren. Diese Aufteilung wurde gewählt, da so kein zusätzlicher Puffer notwendig ist. Nachdem alle Zwischensummen berechnet wurden, wird ein weiterer Kernel zur Reduktion der Teilsummen auf den Wert Z_C gestartet. Schließlich werden die Potentiale von einem darauffolgendem Kernel durch den Wert Z_C dividiert.

4.3 Verteilung der Berechnungen

In den vorherigen Abschnitten wurde näher erläutert, wie ein einzelner Funktionsaufruf der jeweiligen Phase durch Parallelisierung beschleunigt werden kann. Diese Funktionen müssen pro Operationen jedoch öfters ausgeführt werden, beispielsweise muss für jede Kante die Initialisierungfunktion aufgerufen und zwei Nachrichten über diese gesendet werden. Dieser Abschnitt wird sich der Frage widmen, welche weiteren Möglichkeiten zur Parallelisierung in den einzelnen Phasen vorhanden sind und falls ja, wie diese verteilt auf unterschiedlichen Geräten berechnet werden können. Wie bereits zuvor werden die Operationen nacheinander in der zeitlichen Reihenfolge ihrer Aufrufe betrachtet.

Algorithmus 7 Sorted-Balance [36]

Eingabe: O – Liste der Arbeitspakete, w – Gewichte, k – Anzahl an Partitionen

Ausgabe: $\mathcal{P}$ – Makespan-Approximation der Liste O

```
1: work ← array(k, 0)
2: sort(O, key = work, descending=True)
3: P ← [ ]
4: for i = 1 to |O| do
5:    min ← 0
6:    for k = 1 to k do
7:       if work[k] < work[min] then
8:          min = k
9:    P[min].append(O[i])
10:   work[min] ← work[min] + work[i]
11: return P
```

4.3.1 Initialisierung

Zur Initialisierung müssen die Kantengewichte des Markov Random Fields in die Potentialtabellen der Cliquen des Junction-Trees geladen werden. Da dies für jede Kante geschehen muss, bietet sich die Verteilung der Berechnungen auf verschiedenen Geräten an. Zur Vermeidung von Race Conditions und Dateninkonsistenz muss beachtet werden, dass die Potentialtabelle einer Clique nicht von mehreren Geräten gleichzeitig manipuliert wird. Um dies zu garantieren, sollen Jobs, welche die selben Cliquen initialisieren, immer sequentiell auf einem Device laufen. Dies soll durch ein Schedule sichergestellt werden, welches pro Device eine Liste von Jobs verwaltet. Zur Berechnung des Schedules werden zum Programmstart einmalig alle Kanten des Graphen gescannt und zu jeder Kante e wird eine entsprechende Clique C gesucht und in einem Tupel $t = (e, C)$ in einer Liste $\mathcal{L}$ abgespeichert. Daraufhin wird diese Liste nach den Cliquen sortiert, so dass Jobs, welche dieselbe Clique manipulieren, nebeneinander liegen. Anschließend wird eine Liste von Listen angelegt, in der jede der inneren Listen alle Jobs beinhaltet, die dieselbe Clique manipulieren. Im Folgenden werden die inneren Listen als ein Arbeitspaket betrachtet, da sie sequentiell auf einem Device berechnet werden müssen. Um die Pakete möglichst optimal auf die verschiedenen Devices zu verteilen, müssen Kosten beziehungsweise Gewichte mit den Berechnungen assoziiert werden. In dieser Arbeit wird dazu jedem Paket als Gewicht das Produkt aus dem Zustandsraum der Clique sowie der Anzahl an Cliquen der inneren Liste zugewiesen. Die möglichst gleichmäßige Verteilung der Arbeitspakete auf k Geräte kann als Instanz des klassischen Makespan-Problems [23] betrachtet werden. Sei $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ eine Menge von n Items und $w_i \geq 0$ das Gewicht des i-ten Items. Dann

Algorithmus 8 Schedule Initialisierung

Eingabe: $G = (V, E)$ – Graph des MRF, $\mathcal{T}$ – Junction-Tree, k – Anzahl der Devices

Ausgabe: Partitionierung der Jobs auf k Devices

1: $\mathcal{L} \leftarrow [\,]$
2: **for** $e \in G.E$ **do**
3: $\quad \mathcal{L}$.append((e, matchingClique($\mathcal{T}$, e)))
4: sort($\mathcal{L}$, key=Clique)
5: $\mathcal{J} \leftarrow$ groupByClique($\mathcal{L}$)
6: $\boldsymbol{w} \leftarrow \{|\mathcal{J}^{(i)}| \cdot |\mathcal{X}_{\mathcal{J}^{(i)}.C}| \mid i \in \{1, \ldots, |\mathcal{J}|\}\}$
7: $\mathcal{S} \leftarrow$ SortedBalance($\mathcal{J}$, $\boldsymbol{w}$, k)
8: **return** $\mathcal{S}$

ist das Problem wie folgt definiert: Finde eine Partition $\mathcal{P} = \{S_1, S_2, \ldots, S_k\}$, so dass die maximale Summe der Gewichte einer Teilmenge S_j minimal ist

$$\mathcal{P}^* = \arg\min_{\mathcal{P}} \max_{S_j \in \mathcal{P}} \sum_{i=1}^{|S_j|} w[S_i].$$

Als Makespan der Partition $\mathcal{P}^*$ wird die maximale Summe der Gewichte einer ihrer Teilmengen bezeichnet. Die exakte Lösung dieses Problems ist NP-schwer, weshalb in dieser Arbeit zur Approximation der gierige *Sorted-Balance*-Algorithmus (siehe Algorithmus 7) genutzt wird [36]. Dieser garantiert, dass der Makespan der Zuweisung maximal $\frac{3}{2}$ schlechter ist, als die der optimalen Lösung $\mathcal{P}^*$. Schließlich werden die Listen pro Device nach den zu aktualisierenden Cliquen sortiert, wodurch unnötige Kopieroperationen zwischen Host und Device vermieden werden können. Beim Aufruf der Initialisierungsfunktion wird pro Device ein OpenMP-Thread gestartet, welcher die Jobs in der vorberechneten Liste $\mathcal{S}$ abarbeitet. Wie gut die Initialisierung parallelisiert werden kann, hängt von der Anzahl an Cliquen und der Verteilung von den Variablen zu den Cliquen ab.

4.3.2 Message-Passing

In Abschnitt 4.2.2 wurde die Berechnung einer einzelnen Nachricht durch die Nutzung von Datenparallelismus beschleunigt. Das Message-Passing-Protokoll bietet basierend auf der Topologie eines Junction-Trees eine weitere Möglichkeit zur Parallelisierung. Während der Collect-Evidence-Phase können alle Blätter parallel beginnen, ihre Nachrichten in Richtung der Wurzel zu senden. Die einzige Bedingung ist, dass Knoten, die mehr als eine eintreffende Nachricht erwarten, auf alle Nachrichten warten, bevor diese ihre Nachricht weiterleiten dürfen. Im Folgenden werden diese als Synchronisationsknoten bezeichnet. Nachdem die Collect-Evidence-Phase abgeschlossen ist, können alle Pfade von der Wurzel aus zu den Blättern ebenfalls parallel berechnet werden. In den folgenden Abschnitten wird ein Algorithmus zur Berechnung eines Schedule vorgestellt. Der erste Schritt besteht daraus,

Algorithmus 9 Zerlegung des Baums

Eingabe: $G = (V, E)$ – Baum, r – Wurzel, $\mathcal{S}$ – Synchronisationsknoten

Ausgabe: Liste mit Mengen $\mathcal{W}$

```
1: P ← DFS(r)
2: i ← 0
3: Q ← PriorityQueue(S, dist(S, r))
4: while not empty(Q) do
5:   s ← pop(x)
6:   for path in P do
7:     if intersection(path, s) then
8:       outer, inner ← split(path,s)
9:       replace(P, path, inner)
10:      W_i ← W_i ∪ outer
```

für einen gegebenen Wurzelknoten Mengen von unabhängigen Arbeitspaketen zu berechnen. Anschließend wird ein hierarchisches Schedule aus den Paketen berechnet. Zuletzt wird die Frage behandelt, welcher Knoten sich als Wurzel eignen könnte, um möglichst viele unabhängige Pfade zu erhalten.

Berechnung der Arbeitspakete

Die Arbeit von Kozlov *et al.* [39] nutzt einen dynamischen Ansatz zur Berechnung der Arbeitspakete. Während der Traversierung des Graphen werden die Nachrichten nach dem *Producer–Consumer*-Schema in eine geteilte Queue gelegt und nacheinander von den verschiedenen CPU-Arbeitsprozessen aus der Queue gelesen. Im Gegensatz zu diesem Ansatz wurde sich für die Vorberechnung eines statischen Schedules entschieden, denn so ist garantiert, dass die Kopieroperationen von dem Doppelpufferungsansatz profitieren und Ergebnisse effizient weiterverwendet werden können.

Die Idee des Algorithmus zur Berechnung der Arbeitspakete (Algorithmus 9) ist die stufenweise Zerlegung des Baumes an den Synchronisationsknoten. Zunächst wird ein Knoten r des Junction-Tree als Wurzel bestimmt, woraufhin eine Tiefensuche genutzt wird, um die Pfade zu den Blättern zu bestimmen. Anschließend werden die Synchronisationsknoten und deren Distanz zur Wurzel bestimmt und in einer Prioritätswarteschlange der Distanz nach gespeichert. Der Algorithmus geht iterativ vor und wählt in jeder Iteration i den von der Wurzel am weitesten entfernten Knoten aus. Von jedem Blatt des Baumes wird überprüft, ob der Pfad von diesem zur Wurzel durch den Schnittpunkt verläuft. Falls der Pfad den Punkt schneidet, wird der Pfad an dieser Stelle durchtrennt und der Teil von dem Blatt bis zum Schnittpunkt in einer Liste $\mathcal{W}^{(i)}$ gespeichert. Nachdem alle Pfade überprüft wurden, wird der Synchronisationsknoten zu einem Blatt des Graphen und die Liste enthält alle

Algorithmus 10 Merge

Eingabe: Liste mit Mengen unabhängigen Pfaden $\mathcal{W}$

Ausgabe: Reduzierte Liste $\mathcal{W}$

```
1: while change do
2:   for i = 1 to |W| do
3:     for path ∈ W_i do
4:       if ∄ p_{a→b} ∈ W_{i-1} : path.a == p.b then
5:         W_{i-1} ← W_{i-1} ∪ path
6:         W_i ← W_i \ path
7:         change ← True
```

Pfade von den Blättern zu diesem. Der Algorithmus terminiert, sobald der Graph nur noch aus einem Knoten, der Wurzel, besteht.

Aufgrund der Prioritätswarteschlange sind immer nur die Pfade bis zu einem Synchronisationsknoten auf einer Ebene der Liste $\mathcal{W}$, jedoch können alle Pfade zu den äußersten Synchronisationsknoten parallel berechnet werden. Um das Schedule so anzupassen, dass alle parallelen Pfade auf einer Ebene sind, wird jeweils pro Ebene und jedem ihrer Pfade überprüft, ob dieser unabhängig von allen Pfaden der vorherigen Ebene ist. Diese Überprüfung findet statt, indem der letzte Knoten des Pfades der vorherigen Ebene mit dem aktuellen Startknoten verglichen wird. Sind diese nicht gleich, ist der Knoten von dem spezifischen Pfad unabhängig. Ist ein Pfad von allen Pfaden der vorherigen unabhängig, kann er in die vorherige Ebene verschoben werden. Da Pfade auch mehrere Ebenen nach vorne gezogen werden können, wird dieses Vorgehen solange wiederholt, wie Pfade die Ebenen wechseln. Das Vorgehen ist im Merge-Algorithmus (Algorithmus 10) beschrieben. Sobald keine Änderungen an den einzelnen Ebenen mehr durchgeführt werden, sind jeweils alle Pfade pro Ebene voneinander unabhängig und die jeweiligen Ebenen enthalten die Arbeitspakete für die Collect-Evidence-Phase. Da während der Distribute-Evidence-Phase die Nachrichten über dieselben Pfade nur in umgekehrter Reihenfolge gesendet werden, wird eine Kopie der Liste $\mathcal{W}$ angelegt und diese wird umgekehrt. Außerdem werden alle Pfade innerhalb der Ebenen umgedreht. Daraufhin wird die Kopie noch einmal dem Merge-Algorithmus unterzogen und letztendlich an die Liste $\mathcal{W}$ angehängt. Die Pfade einer Ebene der Liste sollen anschließend verteilt auf mehreren GPUs berechnet werden.

Schedule

Um die zuvor berechneten Arbeitspakete pro Ebene auf k Geräte zu verteilen, kann erneut der Sorting-Balance-Algorithmus verwendet werden. Für jede Ebene wird dieser mit den entsprechenden Jobs der Ebene aufgerufen. Jeder Job ist in diesem Fall ein Pfad, der parallel ausgeführt werden kann. Als Gewicht erhält ein Job die Summe der

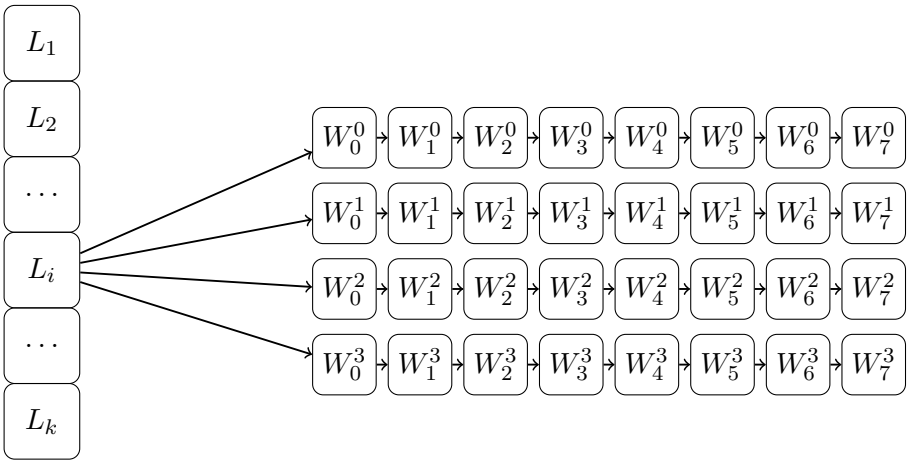


Abbildung 4.5: Das Schedule für das Message-Passing existiert aus k Ebenen. Jede Ebene besteht aus $k = 4$ Queues, eine pro GPU.

Algorithmus 11 Generate Schedule

Eingabe: $\mathcal{W}$ – Liste der Ebenen mit unabhängigen Paketen, k – Anzahl an Streams k

Ausgabe: L – Liste mit einem Schedule für jede Ebene

1: **for** i = 1 **to** $|\mathcal{W}|$ **do**
2: $\boldsymbol{w} \leftarrow \texttt{array}(|\mathcal{W}^{(i)}|, 0)$
3: **for** j = 1 **to** $|\mathcal{W}^{(i)}|$ **do**
4: $\boldsymbol{w}_i \leftarrow \sum_{\text{msg} \in \mathcal{W}_j^{(i)}} |\mathcal{X}_{msg.B}| + |\mathcal{X}_{msg.C}|$
5: $L[i] \leftarrow \texttt{SortedBalance}(\mathcal{W}^{(i)}, \boldsymbol{w}, k)$
6: **return** L

Operationen, die entlang des Pfades während der Nachrichten ausgeführt werden müssen. Sobald die Message-Passing-Operation ausgeführt werden soll, werden k OpenMP-Threads gestartet, welche die Arbeitspakete ebenenweise auf unterschiedlichen Devices abarbeiten. Nach jeder Ebene wird einmal synchronisiert, so dass alle Abhängigkeiten für die Jobs der darauffolgenden Ebene erfüllt sind. Zusätzlich muss während der Aktualisierung einer Clique, die mehrere Nachbarn besitzt, synchronisiert und gelockt werden, da die Manipulation der selben Potentialtabelle auf unterschiedlichen Geräten zu einem inkonsistenten Zustand der Tabellen führen würde [55].

Wahl des Wurzelknoten

Der Algorithmus ist für jede Wahl des Wurzelknoten korrekt, sofern sich an das Nachrichtenprotokoll gehalten wird. Jedoch kann die Wahl der Wurzel das Schedule beeinflussen, da abhängig von der Wahl dieser sich die Pfade von den Blättern zur Wurzel ergeben, über welche die Nachrichten fließen. Wird beispielsweise ein Blatt als Wurzel genommen, geht Parallelität verloren, da wenn ein Knoten, welcher zentraler liegt, als Wurzel genommen

worden wäre, es den zusätzlichen parallelen Pfad von diesem Blatt zur Wurzel gäbe. Um die Nachrichten parallel berechnen zu können, sollte das Schedule aus möglichst vielen unabhängigen Pfaden bestehen, in denen so selten wie möglich synchronisiert werden muss. Die grundlegende Implementierung der PX-Bibliothek führt die Pfade von den Blättern zu der Wurzel nicht parallel aus, weshalb die Wahl der Wurzel keinen Einfluss auf die Laufzeit hat und einfach der erste Knoten des Junction-Tree gewählt wird. In dieser Arbeit wird ein Ansatz vorgestellt, der garantieren sollen, dass kein Blatt und ein möglichst zentraler Knoten als Wurzel gewählt wird. Dies soll mehr Parallelität und ebenfalls möglichst ausgeglichene Pfade von den Blättern zur Wurzel liefern.

Der Ansatz basiert auf der *Betweenness Centrality* [21], welche für einen Knoten v misst, wie viele der kürzesten Pfade von allen Knoten zueinander durch diesen fließen. Formal ist die Betweenness Centrality eines Knoten v wie folgt definiert

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{4.1}$$

wobei σ_{st} die Anzahl aller kürzesten Pfade von s nach t ist und $\sigma_{st}(v)$ die Anzahl der Pfade, die den Knoten v passieren. Die Idee dieses Ansatzes ist es, den Knoten mit der höchsten Betweenness Centrality als Wurzel zu wählen

$$Wurzel = \arg\max_{v \in V} C_B(v),$$

da Knoten mit höheren Werten der Betweenness zentraler im Graphen liegen und so die Pfade zu den Blättern möglichst kurz sind. Zur effizienten Berechnung der Betweenness Centrality für jeden Knoten des Graphen kann Brandes Algorithmus genutzt werden [7].

4.3.3 Verteilte Normalisierung

Nachdem das Message-Passing ausgeführt wurde, müssen die Potentiale jeder Clique normalisiert werden. Da die Potentiale der Cliquen unabhängig voneinander sind, können sie in der Theorie alle parallel ausgeführt werden, der Algorithmus 6 wird also so abgeändert, dass die äußere Schleife parallel ausgeführt wird. Da zur komplett parallelen Ausführung jedoch Speicher für alle Cliquen auf den Devices bereitstehen müsste und nicht alle Daten parallel zwischen Host und Device kopiert werden können, wird pro Device und pro Puffer ein OpenMP-Thread gestartet, welcher das Kopieren der Daten und Aufrufen der Kernel übernimmt. Diese Funktion sollte direkt mit der Anzahl an GPUs skalieren.

4.4 Modellbasiertes Scheduling

Während der Implementierung wurde beobachtet, dass insbesondere für kleinere Modelle die Laufzeit der GPU-Inferenz größer ist als jene der CPU. Eine mögliche Ursache könnte sein, dass die Parallelität der GPU nicht ausgenutzt wird und die überlegene Taktrate der

CPU die fehlende Parallelität ausgleicht. Außerdem können kleine Kopieroperationen nicht die volle Bandbreite ausschöpfen und auf der CPU komplett vermieden werden. Ebenfalls gehen Kernelaufrufe immer mit einem gewissen Overhead einher. In einer Arbeit von Zheng *et al.* [64] wird zur Behebung dieses Problems eine Technik namens *Clique-Merging* verwendet, welche für jeweils zwei benachbarte Cliquen überprüft, ob diese nicht in einer Clique zusammengefasst werden sollten. Indem kleine Cliquen eliminiert werden, erhoffen die Autoren sich mehr Möglichkeiten für Parallelität und geringeren Overhead. Jedoch kommt dieses Verfahren auch mit einigen Nachteilen daher. Auf der einen Seite muss ein Schwellwert bestimmt werden, ab und bis zu welcher Größe Cliquen zusammengefasst werden sollen, denn zu kleine oder zu große Cliquen haben wiederum andere Nachteile. Auf der anderen Seiten ist es nicht immer möglich, alle kleinen Cliquen zu eliminieren. In dieser Arbeit soll stattdessen ein alternativer Ansatz verfolgt werden – ein Laufzeitmodell könnte entscheiden, welches Problem auf welcher Plattform am schnellsten gelöst werden kann.

Zunächst muss jedoch untersucht werden, welche Operationen die Laufzeit beeinflussen und ob sich diese überhaupt modellieren lassen. Zur Datensammlung wurde die Inferenzroutine mehrfach mit verschiedenen graphischen Modellen sowohl auf der CPU als auch auf der GPU durchgeführt und die Laufzeit der Operationen Initialisierung, Message-Passing und der Normalisierung geloggt. Genutzt wurden dabei verschiedene künstliche Graphen mit einer variierenden Anzahl an Knoten und Zuständen sowie Graphen aus dem *Bayesian-Network-Repository* [2]. Anschließend wurden die Daten pro Operation nach den Eingabeparametern gruppiert und deren mittlere Laufzeit errechnet. In den folgenden Teilabschnitten wird für die jeweiligen Operation untersucht, von welchen Eingabeparametern die Laufzeit abhängt und ob diese sich modellieren lässt.

4.4.1 Initialisierung

In der Abbildung 4.6 ist die Visualisierung der Korrelationsmatrix zwischen den Eingabeparametern des Algorithmus und der Laufzeit zu sehen. Die Eingabeparameter sind die Anzahl der annehmbaren Zustände der Clique, die Anzahl an Knoten der Clique sowie die Anzahl der Parameter der zugehörigen Kante. Berechnet wurde die Korrelation mit dem Pearson-Korrelationskoeffizienten, welcher den linearen Zusammenhang zwischen zwei Variablen misst [20]. Die Grafik zeigt deutlich, dass es sowohl auf der CPU als auch auf der GPU einen starken linearen Zusammenhang zwischen der Anzahl an Zuständen der Clique sowie der Laufzeit gibt. Diese Beobachtung motiviert die Modellierung der Laufzeit durch eine lineare Regression, welche den linearen Zusammenhang einer Zielvariable $y \in \mathbb{R}$ und einem Merkmalsvektor $\boldsymbol{x} \in \mathbb{R}^n$ über einen Parametervektor $\boldsymbol{\theta} \in \mathbb{R}^n$ und einem Biasterm $b \in \mathbb{R}$ wie folgt modelliert [27]

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \langle \boldsymbol{x}, \boldsymbol{\theta} \rangle + b. \tag{4.2}$$

[2] `https://www.cse.huji.ac.il/~galel/Repository/` (Zuletzt besucht am 21.11.2019.)

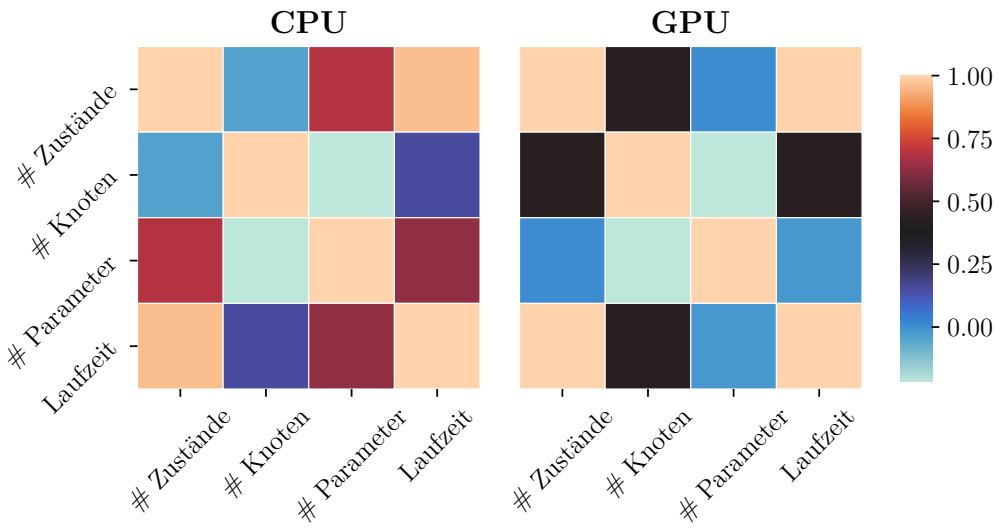


Abbildung 4.6: Diese Abbildung zeigt die Korrelationsmatrix zwischen den Eingabeparametern und der Laufzeit für die Initialisierung einer Clique. Es ist zu erkennen, dass ein starker linearer Zusammenhang zwischen der Anzahl der Zustände einer Clique und der Laufzeit besteht.

Die Idee ist es nun, ein Modell der Laufzeit sowohl für die CPU als auch eins für die GPU zu erstellen. Bevor eine Clique initialisiert wird, werden die Modelle zur Vorhersage der Laufzeit genutzt und die Operation wird auf dem Device ausgeführt, für welches die geringere Laufzeit vorhergesagt wird. Alternativ zum Regressionsansatz könnte das Problem auch in ein Klassifikationsproblem überführt werden, indem ein Datenpunkt das Label $+1$ bekommt, falls die Laufzeit der GPU größer ist als jene der CPU und ansonsten das Label -1. Zur Modellierung würde sich in diesem Fall eine logistische Regression anbieten, jedoch wurde sich dagegen entschieden, da die Vorhersage der Laufzeit ebenfalls zur besseren Berechnung eines Schedules eingesetzt werden könnte.

4.4.2 Message-Passing

Um die Modellierbarkeit der Laufzeit einer Nachricht zu überprüfen, wurde der Einfluss der Eingabeparameter des Algorithmus wie im vorherigen Abschnitt über eine Korrelationsmatrix analysiert. In diesem Fall sind die Eingabeparameter während einer Nachricht der Clique B an seine benachbarte Clique C über den Separator S die jeweiligen Größen der Potentialtabellen. Die entsprechende Matrix ist in Abbildung 4.7 zu sehen. Auf Seiten der CPU ist ein starker linearer Zusammenhang zwischen der Potentialtabellengröße der Cliquen B und C sowie der Laufzeit zu erkennen. Im Falle der GPU sieht es diesmal anders

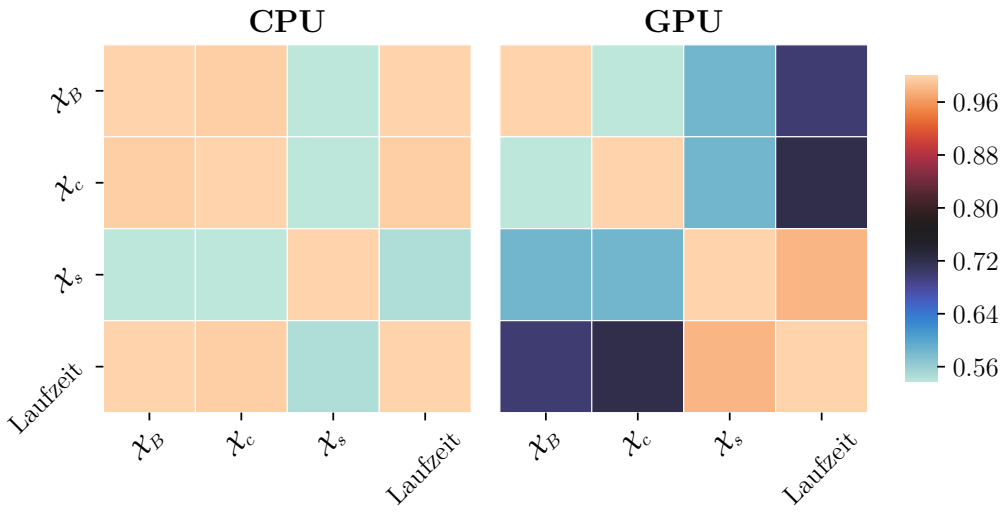


Abbildung 4.7: Diese Abbildung zeigt die Korrelationsmatrix zwischen den Eingabeparametern und der Laufzeit für die Nachricht einer Clique B an seine benachbarte Clique C über den Separator S.

aus: Am stärksten korreliert die Größe des Separators S mit der Laufzeit. Dies ist dadurch zu erklären, dass die parallelen Berechnungen über die Anzahl der Separatorzustände kontrolliert werden und die Elemente der anliegenden Potentialtabellen parallel bearbeitet werden. Auch hier scheint ein linearer Zusammenhang gegeben zu sein, welcher erneut durch ein lineares Modell modelliert werden kann.

4.4.3 Normalisierung

Im Vergleich zu den vorherigen Operationen hängt die Laufzeit der Normalisierung nur von einem Eingabeparameter, der Cliquengröße, ab. Daher bietet sich die Visualisierung der Laufzeit als Funktion in Abhängigkeit der Cliquengröße in einem eindimensionalen Plot an. Die Zustandsgröße wurde auf 10000 limitiert, um einen besseren Einblick in die interessante Region zu erhalten. Größere Cliquen konnten auf der GPU immer schneller normalisiert werden. In der Abbildung 4.8 ist deutlich erkennbar, dass die Laufzeit der Normalisierung auf der CPU einem linearen Trend folgt. Die Laufzeit auf der GPU folgt ebenfalls einem linearen Trend, jedoch gibt es bei einer Cliquengröße von ca. 1000 einen Sprung in der Laufzeit. Dieser lässt sich dadurch erklären, dass aufgrund der Blockgrößenbeschränkung von 1024 Threads zwei Kernel gestartet werden müssen – der erste berechnet die jeweiligen Teilsummen und der Zweite reduziert diese auf einen Wert. Nach dem Sprung folgt die

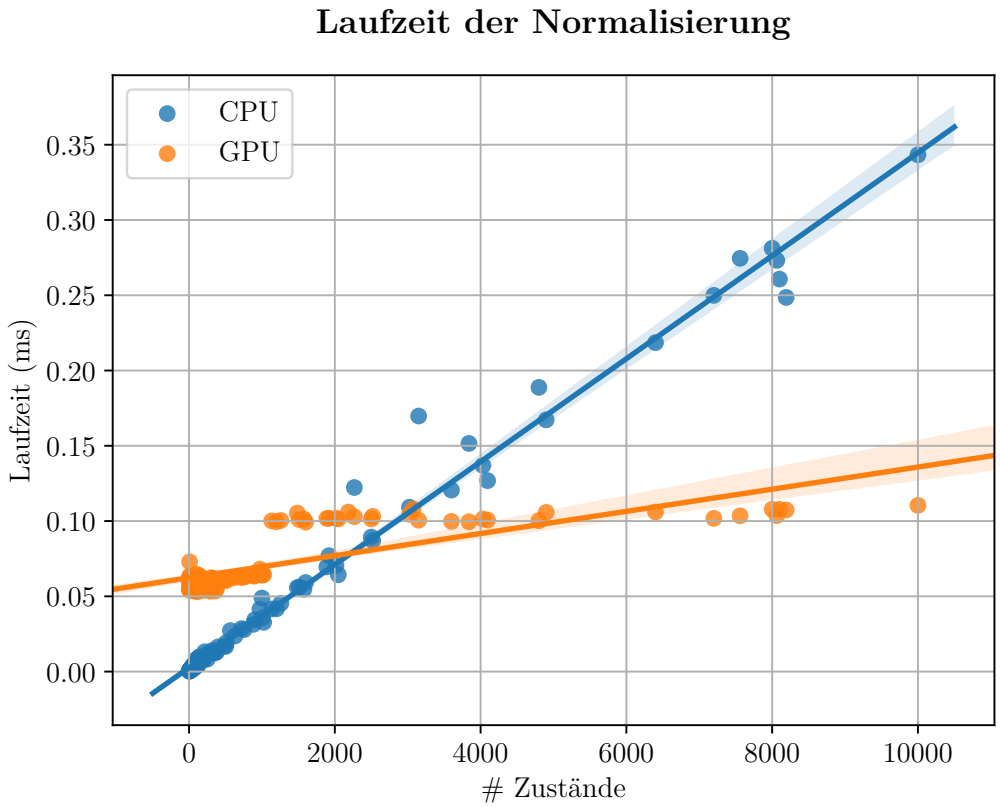


Abbildung 4.8: Diese Abbildung stellt die Laufzeit der Normalisierung in Millisekunden in Abhängigkeit der Cliquengröße dar.

Laufzeit der GPU ebenfalls einem linearen Trend. Sie steigt jedoch sehr langsam, da von der Operation die Parallelität der Hardware exzellent ausgenutzt werden kann.

Für alle Operationen wurden lineare Zusammenhänge zwischen den Eingabeparametern und der Laufzeit erkannt. Diese sollen nun zur Optimierung des Schedules genutzt werden, indem ein Modell entscheidet, welches Problem am besten auf welcher Plattform gelöst wird. Die Effektivität dieser Technik wird im folgenden Kapitel evaluiert.

Zusammenfassung

In diesem Kapitel wurde ein Algorithmus zur Beschleunigung der Junction-Tree-Inferenz vorgestellt. Beschleunigt wurden die einzelnen Funktionsaufrufe durch die Nutzung von Datenparallelismus auf einer CUDA-fähigen GPU. Durch die Hinzunahme von Taskparalellismus konnten weitere unabhängige Arbeitspakete identifiziert und Schedules erstellt werden, welche wiederum durch ein Laufzeitmodell optimiert wurden. Die Schedules können ebenfalls so berechnet werden, dass die Jobs auf mehrere GPUs eines Hosts verteilt werden. Um die Jobs über mehrere Hosts zu verteilen, könnte die OpenMP-Orchestrierung durch eine MPI-Implementierung ersetzt werden. Jedoch sind heutzutage nicht selten bis zu

vier oder gar mehr GPUs pro Rechner verbaut, welche bereits eine enorme Rechenkraft bereitstellen. Des Weiteren wurden die Ideen von anderen Arbeiten, wie beispielsweise die Index-Mapping-Vektoren [30] und der Doppelpufferungsansatz [32], eingebaut.

Kapitel 5

Experimente

In den Experimenten soll evaluiert werden, wie gut die Implementierung auf künstlichen und echten Datensätzen funktioniert. Um Vergleichbarkeit zu gewährleisten, wurden alle Experimente auf demselben System ausgeführt. Das verwendete System nutzte als Betriebssystem Ubuntu 16.04 und besteht aus einer $Intel(R)Xeon(R)E5-2690$ CPU, 500GB Hauptspeicher und vier NVIDIA GTX 1080. Zur Übersetzung der Software wurde der C++-Compiler aus der *GNU Compiler Collection* [1] (GCC) in der Version 5.4 verwendet. Dabei wurde die höchste Optimierungsstufe `O3` genutzt. Der GPU-Code wurde mit CUDA 10.1 [2] entwickelt und übersetzt. Zur Triangulation der Basisgraphen wurde der MCS-Algorithmus genutzt. Ein direkter Vergleich mit den vorherigen Arbeiten ist nicht möglich, da die Autoren weder Quellcode noch die exakten Junction-Trees beziehungsweise ihre Methoden zur Triangulierung bereitgestellt haben. Außerdem basieren Teile der Implementierung auf deren Techniken [30,32,65]. Stattdessen wird als Baseline die CPU-Implementierung der PX-Bibliothek verwendet. Am Ende des Kapitels werden die Ergebnisse bewertet. Aufgebaut ist dieses Kapitel wie folgt: Zunächst werden die verwendeten Graphen und Datensätze in 5.1 näher beschrieben. Anschließend wird die Effektivität von jeder der verwendeten Techniken nacheinander in den Abschnitten 5.2 bis 5.5 evaluiert. Daraufhin wird in Abschnitt 5.6 die aus den verschiedenen Techniken zusammengesetzte Multi-GPU-Implementierung mit den CPU- und der Single-GPU-Implementierung verglichen. Schließlich wird auf die Limitierungen eingegangen und es wird ein Fazit gezogen.

Es sei erwähnt, dass die Laufzeiten in allen Tabellen und Grafiken in Millisekunden angegeben sind und über 50 Läufe der Junction-Tree-Inferenz gemittelt wurden. In den Abbildungen sind die Standardabweichungen über Fehlerbalken eingezeichnet. Da die Tabellen bereits ohne Angabe der Standardabweichung sehr viel Platz einnehmen, sind deren Standardabweichungen in Anhang A.3 gelistet.

[1] `https://gcc.gnu.org/gcc-5/` (Zuletzt besucht am 21.11.2019.)

[2] `https://developer.nvidia.com/cuda-10.1-download-archive-base` (Zuletzt besucht am 21.11.2019.)

5.1 Datensätze

Um zu verifizieren, dass die Implementierungen die Laufzeit des Algorithmus verkürzen, wurden sie auf künstlichen und echten Datensätzen angewendet. Die echten Datensätze wurden aus dem *Bayesian-Network-Repository* ausgewählt [3]. Da es sich bei den Modellen um Bayessche Netze handelt, wurden diese zunächst moralisiert. Die Eigenschaften der verschiedenen Graphen sowie die der zugehörigen Junction-Trees sind in Tabelle A.1 aufgeführt. In Abschnitt A.2 des Anhangs sind außerdem Visualisierungen der verschiedenen Junction-Trees und die Größe der Zustandsräume zu sehen. Ebenfalls soll das Sensornetzwerk des ICE-CUBE-Hochenergie-Neutrino-Observatoriums [25] modelliert werden. Das Ziel des Observatoriums ist die Erkennung von Neutrinos. Es befindet sich am Südpol und besteht aus 5160 digitalen optischen Sensoren (DOM). Treffen Neutrinos auf die DOMs, interagieren sie mit den Sensoren und geben Energie ab. Da es sich bei der Implementierung um ein diskretes graphisches Modell handelt, mussten die gemessenen Energiewerte zunächst diskretisiert werden. Um eine einigermaßen feine Auflösung beizubehalten, wurden die Messwerte auf 20 Zustände reduziert. Die Abhängigkeitsstruktur wurde mit der Chordalysis-Software [4] geschätzt. Diese Vorarbeit wurde von Mirko Bunse [5] geleistet. Als künstliche Datensätze wurden verschiedene Basisgraphen (siehe Abbildung 5.1) mit einer variierenden Anzahl an Knoten und Zuständen genutzt. So konnte untersucht werden, wie sich die Implementierungen mit einer steigenden Anzahl an Cliquen oder Zuständen verhalten.

5.2 Index-Mapping-Vektoren

In der ersten Reihe von Experimenten wurde untersucht, welchen Einfluss die Index-Mapping-Vektoren auf die Laufzeit der Inferenzroutine nehmen. Dabei wurde nur die Laufzeit des Message-Passing zwischen der CPU und der indexbasierten CPU-Implementierung verglichen, da die Index-Mapping-Vektoren keinen Einfluss auf die anderen Operationen haben. Für künstliche Graphen wurde untersucht, wie sich die Laufzeit sowohl in Abhängigkeit der Anzahl an Knoten als auch der Anzahl an Zuständen pro Knoten verhält. Während der eine Parameter variiert wurde, wurde der fixe Parameter auf den Wert 50 gesetzt. Ausgenommen ist der Gittergraph, für welchen nur eine Zustandsmenge von zwei gewählt wurde. Durch die hochgradig verbundene Struktur ergeben sich während der Triangulation viele große Cliquen. Für einen Gittergraphen mit $n \times n$ Knoten besteht der Junction-Tree größtenteils aus Cliquen der Größe $(n + 1)$, was in Potentialtabellen der Größe 2^{n+1} resultiert. Die Ergebnisse der Experimente mit der variierenden Anzahl an Knoten sind in Abbildung 5.2 zu sehen. Dabei ist zu erkennen, dass die Nutzung

[3] `https://www.cse.huji.ac.il/~galel/Repository/` (Zuletzt besucht am 21.11.2019.)

[4] `https://github.com/fpetitjean/Chordalysis` (Zuletzt besucht am .)

[5] mirko.bunse@tu-dortmund.de

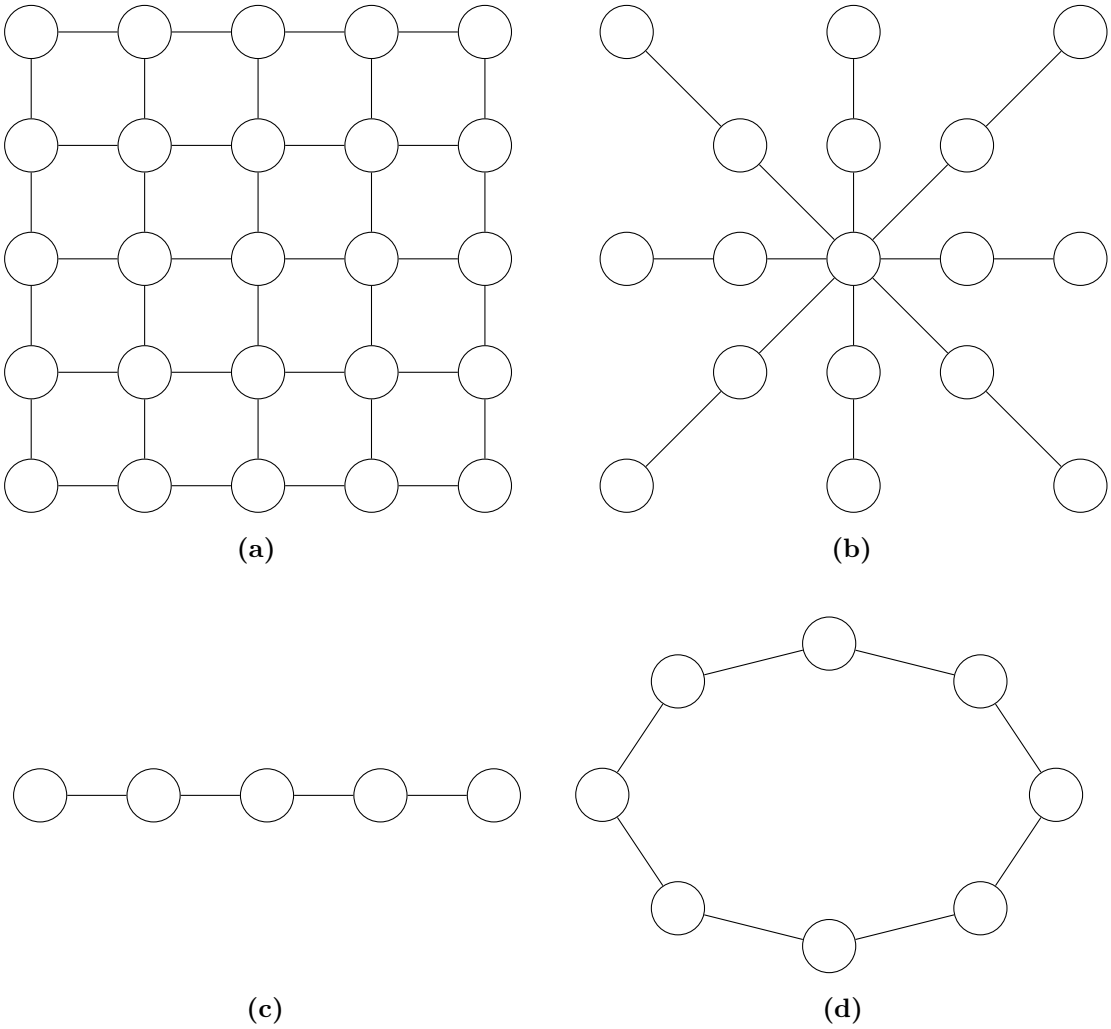


Abbildung 5.1: Diese Abbildung zeigt die Struktur der verschiedenen künstlichen Graphen, die in den Experimenten verwendet wurden. Oben links ist ein Gittergraph der Größe 5×5 zu sehen. Rechts von diesem ist ein Sterngraph mit acht Armen der Länge zwei zu sehen. In der unteren Reihe ist links ein Kettengraph mit fünf Knoten und rechts ein Kreisgraph mit acht Knoten zu sehen.

der Mapping-Vektoren immer einen Vorteil in der Laufzeit mit sich bringt. Insbesondere bei dem Gittergraphen ist ein großer Speedup zu beobachten, welcher durch die rasant wachsende Größe der Potentialtabellen zu erklären ist. Die Vorberechnung der Index-Mapping-Vektoren spart einen großen Anteil der eigentlichen Laufzeit des Message-Passing ein. In Abbildung 5.3 sind die Ergebnisse für eine variierende Anzahl an Zuständen zu sehen. Hier ist ein ähnlicher Trend zu erkennen: Je mehr Zustände der Graph besitzt, umso mehr Laufzeit wird durch die Vorberechnung eingespart. Ebenfalls wurden Experimente auf echten Datensätzen durchgeführt. Die Ergebnisse sind in Tabelle 5.5 zu sehen. Auch hier sind beachtliche Speedups von 7.8 (pathfinder) bis 14.9 (win95pts) zu erkennen. In den Experimenten hat sich herausgestellt, dass sich die Nutzung der Index-Mapping-Vektoren

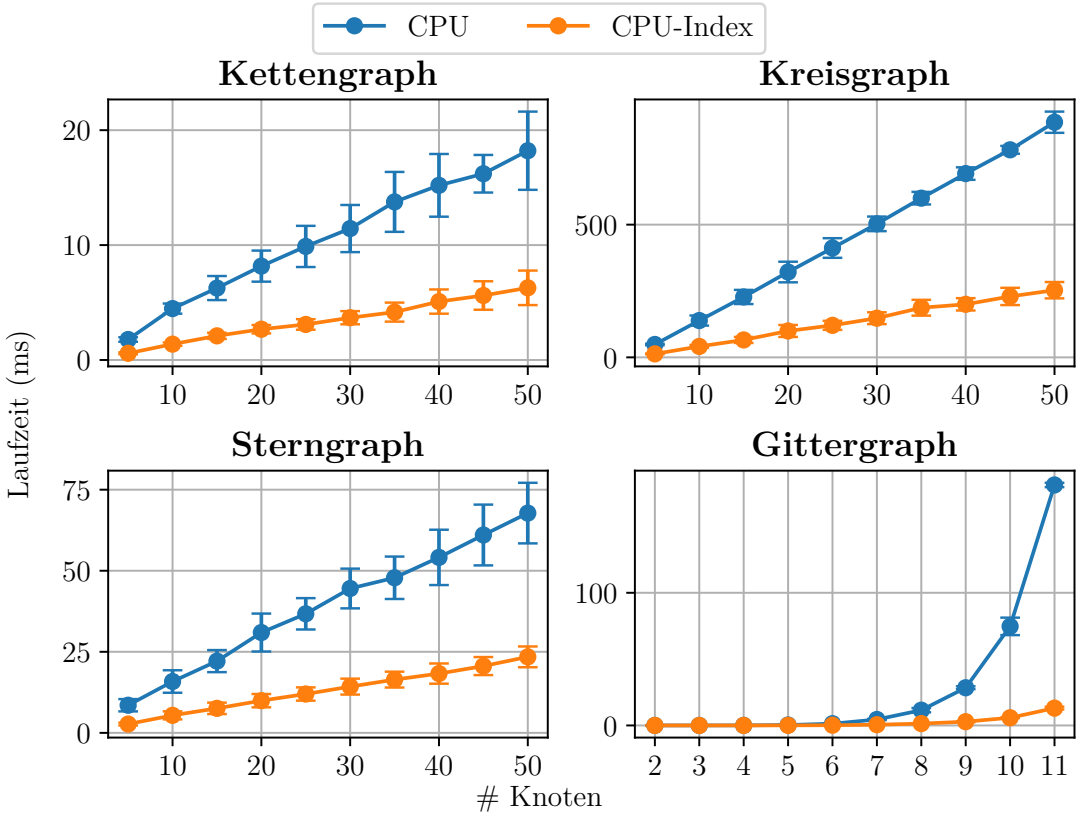


Abbildung 5.2: In dieser Abbildung ist die mittlere Laufzeit des Inferenzalgorithmus über 50 Läufe für verschiedene Graphtypen im Vergleich von CPU zu der CPU-Implementierung mit den Indizes zu sehen.

in jedem Fall lohnt. Insbesondere wenn ein Junction-Tree öfter zur Inferenz verwendet werden soll, wie beispielsweise als Subroutine beim Training, können enorme Vorteile in der Laufzeit gewonnen werden.

5.3 Doppelpufferungsansatz

In den folgenden Experimenten wurde untersucht, ob die Nutzung der Doppelpufferungstechnik einen Einfluss auf die Laufzeit hat. Verglichen wurde während dieser Experimente die GPU-Implementierung, wie sie in Abschnitt 4.2.2 beschrieben ist, mit aktivierter und deaktivierter Doppelpufferungstechnik. In Tabelle 5.1 sind die Ergebnisse für echten Datensätze zu sehen. Es ist zu erkennen, dass die Laufzeit mit Nutzung der Technik für alle außer dem *child*-Datensatz geringer ausfallen. Beim *alarm*-Datensatz ist jedoch nur ein marginaler Vorteil zu erkennen. Dies lässt sich damit erklären, dass die Cliquen dieser Graphen nur aus kleinen Potentialtabellen bestehen und kaum Laufzeit neben den Berechnungen versteckt werden kann. Für mittelgroße Datensätze (*hailfinder*, *pathfinder*) funktioniert dieser Ansatz gut, da bis zu 25% der Laufzeit gegenüber der nicht doppelt

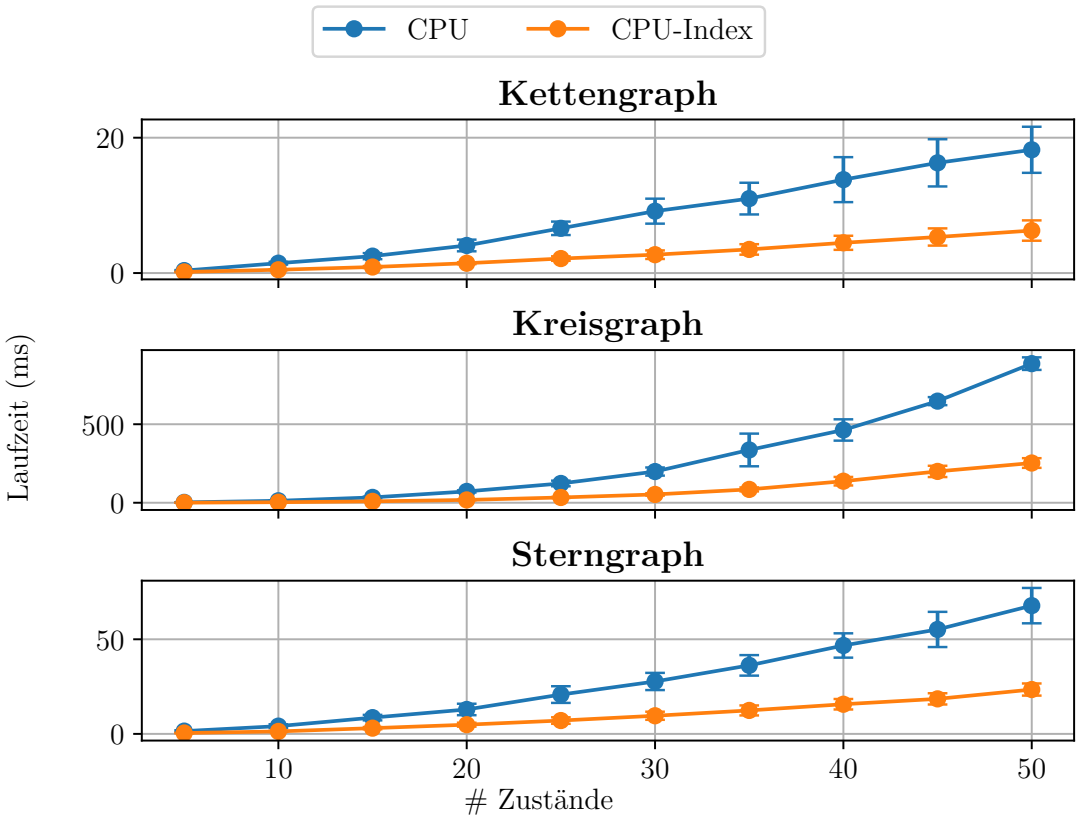


Abbildung 5.3: Diese Abbildung zeigt die mittlere Laufzeit mit Nutzung der Index-Mapping-Vektoren in Abhängigkeit der Zustandsgröße im Vergleich zur standardmäßigen CPU-Implementierung.

gepufferten Laufzeit eingespart werden kann. Datensätze, die aus vielen großen Cliquen bestehen (*grid15*, *water*), können ebenfalls von dieser Technik profitieren, jedoch nicht so stark wie die kleineren Datensätze. Zum einen lässt sich dies durch die Nutzung von *Pinned-Memory* erklären, da größere Kopieroperationen die komplette Bandbreite des PCI-E-Bus ausnutzen können. Zum anderen fällt die Kopierdauer neben den längeren Berechnungen nicht so stark ins Gewicht. Des Weiteren wurden die Versuche für künstliche Graphen mit einer variablen Menge an Knoten und Zuständen durchgeführt. Die Ergebnisse der Versuche mit einer variablen Menge an Knoten sind in Abbildung 5.4 zu sehen. Für den Ketten- und Sterngraph sind nur marginale Verbesserungen zu sehen. Wird jedoch der Plot mit der variablen Anzahl an Zuständen betrachtet (siehe Abbildung 5.5), ist erkennbar, dass alle Graphtypen mit zunehmender Zustandsgröße von dieser Technik profitieren können. Ebenfalls können der Kreis- und Gittergraph von dieser Technik mit einer wachsenden Menge an Knoten profitieren. Diese Beobachtungen lassen sich durch die wachsende Cliquengröße der Junction-Trees erklären. Wenn der Junction-Tree aus vielen Cliquen mit kleinem Zustandsraum besteht, kann kaum Latenz versteckt werden. Sobald

Tabelle 5.1: Diese Tabelle stellt die mittlere Laufzeit in Millisekunden jeweils mit und ohne aktivierter Doppelpufferungstechnik dar.

		alarm	child	insurance	win95pts	hailfinder	pathfinder	grid15	water
Initialisierung	GPU	3.070	1.376	5.560	13.699	4.936	11.093	207.391	1040.394
	GPU-Double-Buffer	**2.106**	**1.038**	**4.034**	**9.116**	**3.612**	**7.787**	**158.703**	**994.703**
MP	GPU	**1.781**	**1.201**	2.653	**3.003**	3.478	9.489	639.110	601.263
	GPU-Double-Buffer	2.653	1.802	**2.079**	3.759	**3.283**	**8.142**	**554.570**	**463.843**
Normalisierung	GPU	3.124	2.020	2.429	4.813	4.311	11.117	77.543	65.269
	GPU-Double-Buffer	**3.116**	2.020	**1.482**	**3.141**	**2.806**	**6.601**	**77.506**	**65.105**
Summe	GPU	7.974	**4.597**	10.642	21.515	12.725	31.699	924.044	1706.927
	GPU-Double-Buffer	**7.875**	4.860	**7.595**	**16.016**	**9.701**	**22.530**	**790.779**	**1523.651**

Tabelle 5.2: Diese Tabelle stellt die zehnfach kreuzvalidierten Metriken der Laufzeitmodelle für die einzelnen Operationen im Vergleich zwischen der CPU und GPU dar. Die letzte Spalte zeigt die Accuracy des CPU- vs. GPU-Entscheidungsproblems.

		MSE	MAE	Median Error	Accuracy
Initialisierung	CPU	0.00646	0.04049	0.01751	0.92474
	GPU	0.00012	0.00812	0.00633	
Message-Passing	CPU	0.00553	0.02084	0.00762	0.96982
	GPU	0.12692	0.05887	0.01146	
Normalisierung	CPU	0.00278	0.02694	0.01423	0.98091
	GPU	0.00065	0.02239	0.02006	

die Cliquen jedoch größer werden, kann ein größerer Anteil der Kopierdauer neben den länger rechnenden Kernelaufrufen versteckt werden. Zusammenfassend lässt sich sagen, dass alle der hier präsentierten Graphen von der Doppelpufferungstechnik profitieren können. Der einzige Nachteil dieser Technik ist, dass die doppelte Menge an GPU-Speicher benötigt wird. Im Folgenden wird diese Technik für alle weiteren GPU-Experimente genutzt.

5.4 Modellbasiertes Scheduling

Als Nächstes soll die Effektivität des modellbasierten Schedulings evaluiert werden. Zur Generierung von Trainingsdaten für die Laufzeitmodelle wurde die Inferenzroutine wiederholt mit verschiedenen Datensätzen auf der CPU und GPU ausgeführt. Dabei wurde die Laufzeit der jeweiligen Operationen gemessen. Anschließend wurden die Ergebnisse nach den Eingabeparametern gruppiert und deren Mittelwert berechnet. Außerdem wurden alle Datenpunkte aus dem Datensatz entfernt, deren Eingabeparameter für die Cliquengröße größer als 15.000 ist. Ab diesem Wert waren für alle Operationen die GPU-Implementierungen schneller und es ermöglicht dem Modell sich besser auf den interessanten Abschnitt zu

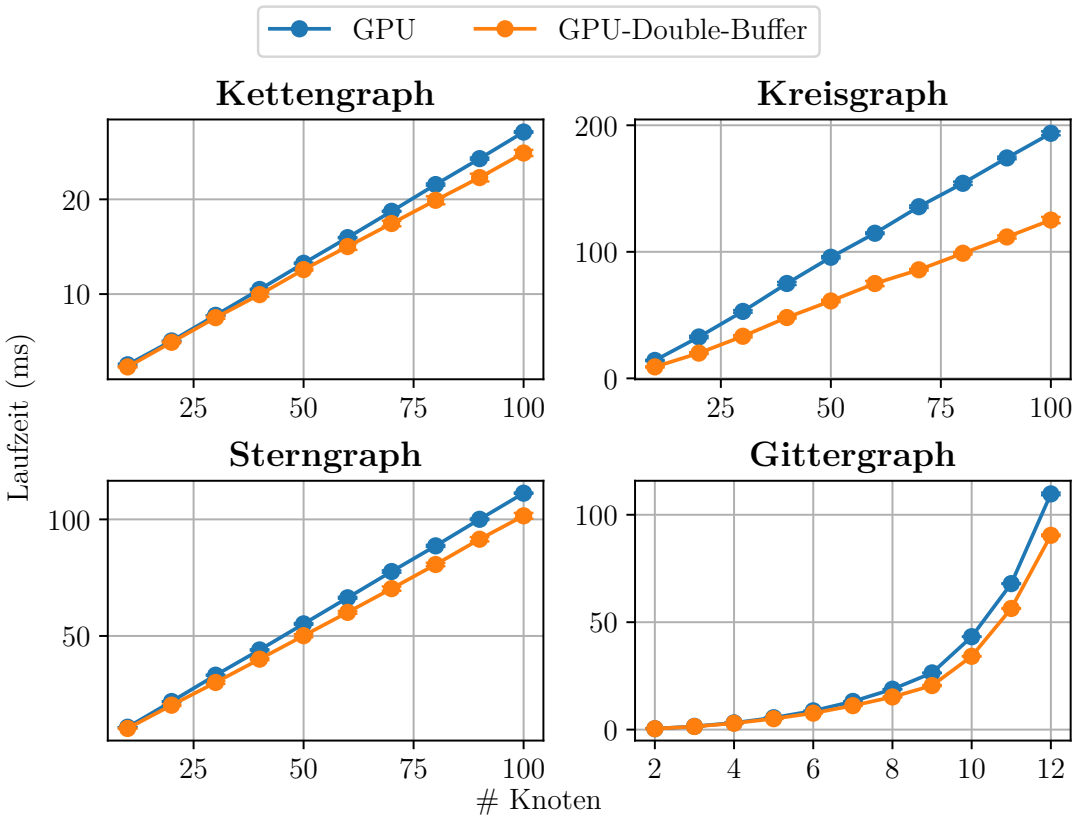


Abbildung 5.4: In dieser Abbildung ist die mittlere Laufzeit mit und ohne Nutzung des Doppelpufferungsansatzes für die künstlichen Graphen zu sehen.

fokussieren. Die Untersuchungen in Abschnitt 4.4 legten den Verdacht nahe, dass ein linearer Zusammenhang zwischen den Eingabeparametern und der Laufzeit besteht. Dies motivierte die Nutzung einer linearen Regression, welche im Folgenden evaluiert wird. Zur Evaluation eines Modells wird immer eine Metrik benötigt, welche die Abweichungen der Vorhersagen des Modells von den Beobachtungen misst. In dieser Arbeit werden der mittlere quadratische (MSE engl. *mean squared error*), der mittlere absolute (MAE) und der Median-Fehler betrachtet und zur Evaluation herangezogen. Da es sich eigentlich um ein Entscheidungsproblem handelt, wurde außerdem die Klassifikationsgenauigkeit betrachtet. Um diese zu berechnen, wurden die beiden linearen Modelle auf einem Testdatensatz zur Vorhersage der Laufzeit angewandt und falls die Laufzeit der GPU kleiner als jene der CPU war, hat der jeweilige Datenpunkt die Klasse +1 erhalten. Anschließend wurde die *Accuracy* berechnet. Für jede Operation der Inferenz wurde das Vorgehen 10-fach kreuzvalidiert. Die entsprechenden Metriken sind in Tabelle 5.2 dargestellt. Dabei ist zu erkennen, dass die Laufzeitmodelle im Mittel geringe Fehler aufweisen. Auch das Lösen der Entscheidungsaufgabe liefert vielversprechende Ergebnisse. Die Accuracy der Modelle für das Message-Passing und die Normalisierung liefern mit 96.7% und 98% zufriedenstel-

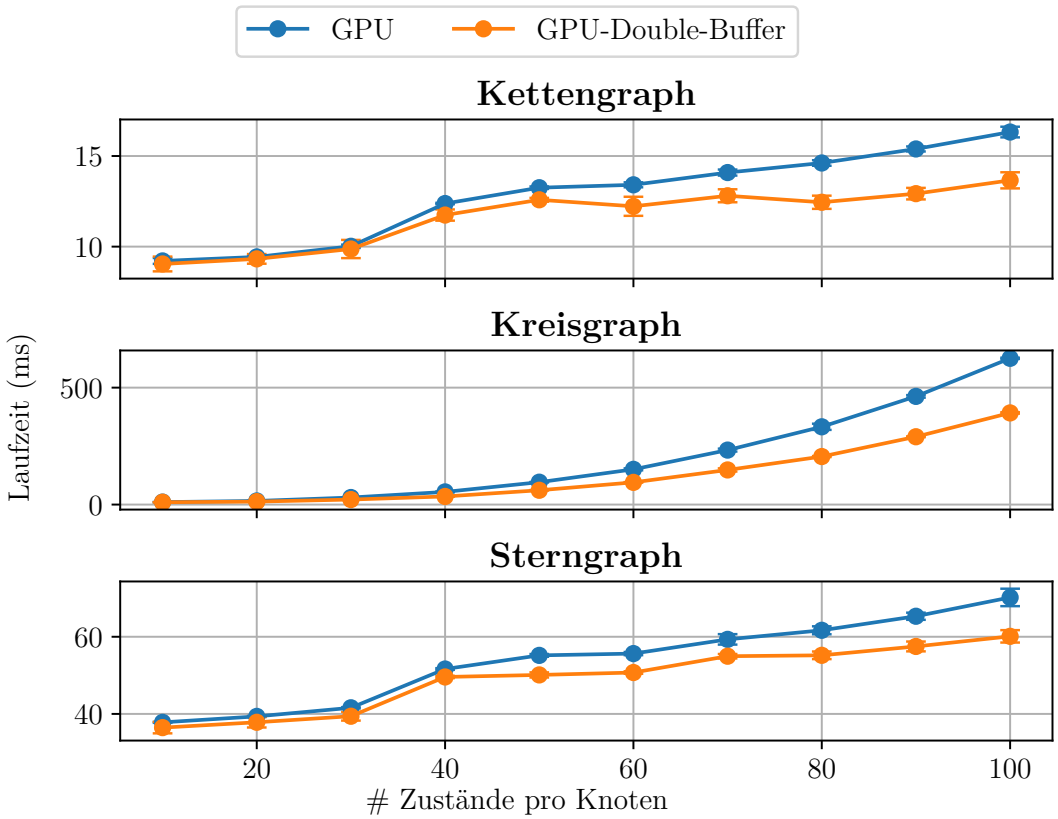


Abbildung 5.5: Diese Abbildung zeigt die mittlere Laufzeit der Doppelpufferungstechnik in Abhängigkeit von der Zustandsgröße im Vergleich zur normalen GPU-Implementierung.

lende Werte. Lediglich beim Modell der Initialisierung besteht mit 92.4% Accuracy etwas Verbesserungsbedarf.

Neben den Metriken wurde ebenfalls evaluiert, wie sich die Laufzeit der Inferenzroutine mit dem hybriden Schedulingansatz verhält. Die Ergebnisse der echten Datensätze sind in Tabelle 5.3 dargestellt. Der hybride Ansatz liefert für die Initialisierung, trotz der geringeren Accuracy, das beste Ergebnis für jeden Datensatz. Auf kleineren Datensätzen neigt das Modell dazu, keine GPU-Unterstützung zu nutzen. Sobald die Graphen allerdings aus mehr und gemischteren Cliquengrößen bestehen, kann das Modell die Vorteile beider Plattformen kombinieren – die Kopieroperationen werden für kleine Cliquen vermieden und große Cliquen können von der starken Parallelität der GPU profitieren. Graphen, die aus vielen riesigen Cliquen bestehen, profitieren kaum von dem Modell, da nur sehr wenige Cliquen auf die CPU geschedult werden. Jedoch bringt das Modell keine Nachteile mit sich. Beim Message-Passing verhält sich das Modell ähnlich, allerdings schneidet es im Vergleich zur Initialisierung nicht so gut ab. Auf drei Datensätzen gewinnt die CPU-Implementierung, auf den restlichen der hybride Ansatz. Dies lässt sich dadurch erklären, dass der Ablauf der Doppelpufferungstechnik unterbrochen werden kann, denn um eine Clique zu aktualisieren,

Tabelle 5.3: Diese Tabelle stellt die mittlere Laufzeit in Millisekunden über zwischen dem CPU, GPU und dem hybriden modellbasierten Ansatz dar.

		alarm	child	insurance	win95pts	hailfinder	pathfinder	grid15	water
Initialisierung	CPU-Index	0.345	**0.130**	53.510	19.935	24.931	65.048	5283.907	35535.459
	GPU	2.094	1.026	3.534	9.026	2.948	7.478	169.000	1007.839
	GPU-Model	**0.309**	**0.130**	**2.546**	**3.960**	**1.193**	**5.898**	**164.574**	**1005.618**
MP	CPU-Index	**0.172**	**0.104**	4.613	**0.926**	3.394	15.247	804.871	1515.683
	GPU	2.605	1.714	2.289	3.786	3.642	8.237	564.096	468.129
	GPU-Model	0.294	0.192	**2.044**	1.282	**1.875**	**7.030**	**561.901**	**467.373**
Normalisierung	CPU-Index	0.147	**0.063**	3.059	0.509	1.773	8.856	572.162	1284.876
	GPU	3.126	2.026	2.604	4.811	4.319	11.136	77.988	65.292
	GPU-Model	**0.116**	0.064	**0.955**	**0.489**	**0.551**	**5.063**	**76.954**	**64.025**
Summe	CPU-Index	**0.664**	**0.297**	61.182	21.370	30.098	89.150	6660.941	38336.017
	GPU	7.825	4.767	8.427	17.623	10.909	26.852	811.083	1541.261
	GPU-Model	0.719	0.386	**5.546**	**5.730**	**3.619**	**17.991**	**803.428**	**1537.016**

muss gegebenenfalls auf eine Rückkopieroperation gewartet werden. Für größere Cliquen tritt der selbe Fall wie zuvor ein: Die Laufzeit wird durch die großen Cliquen dominiert und das Modell bringt nur marginale Vorteile mit sich. Bei der Normalisierung bietet sich dasselbe Bild wie bei der Initialisierung – der hybride Ansatz dominiert, jedoch nimmt der Vorteil mit der Größe des Graphen ab. Insgesamt kann der hybride Ansatz in sechs von acht Fällen die beste Laufzeit erzielen und ist nie schlechter als der pure GPU-Ansatz.

Des Weiteren wurde die Technik auf künstlichen Daten evaluiert. Die Ergebnisse für eine variierende Anzahl an Zuständen ist in Abbildung 5.7 zu sehen. Die abgebildete Laufzeit ist dabei jeweils die Summe aus den drei Operationen. Für den Ketten- und Sterngraph zeichnet sich ein ähnliches Muster ab. Sofern die Knoten des Basisgraphen weniger als 50 Zustände besitzen, ist die CPU-Implementierung am schnellsten. Sobald die Knoten mehr als 50 Zustände annehmen können, ist der GPU-Algorithmus besser als der CPU-Algorithmus. Der modellbasierte Ansatz kombiniert die Vorteile beider Ansätze. Sofern die Knoten kleine Zustände haben, wird auf die CPU- und ansonsten auf die GPU-Implementierung zurückgegriffen. Außerdem ist der modellbasierte Ansatz immer etwas schneller als die GPU-Implementierung, da er kleine Potentialtabellen, wie beispielsweise die der Separatoren, auf der CPU initialisieren und normalisieren kann. Bei dem Kreisgraphen steigt die Größe der Potentialtabellen so schnell, dass sehr zügig alle Operationen auf der GPU ausgeführt werden. Jedoch hat das Modell auch hier keinen Nachteil in der Laufzeit. Für Graphen mit einer variierenden Anzahl an Knoten bietet sich ein ähnliches Bild (siehe Abbildung. 5.6). Die Cliquengröße der Junction-Trees der Kreis- und Gittergraphen wächst sehr schnell mit der Anzahl an Knoten, weshalb alle Operationen auf der GPU durchgeführt werden. Für den Ketten- und Sterngraph kann das Modell wie zuvor die beste Operation auswählen und so die Laufzeit minieren.

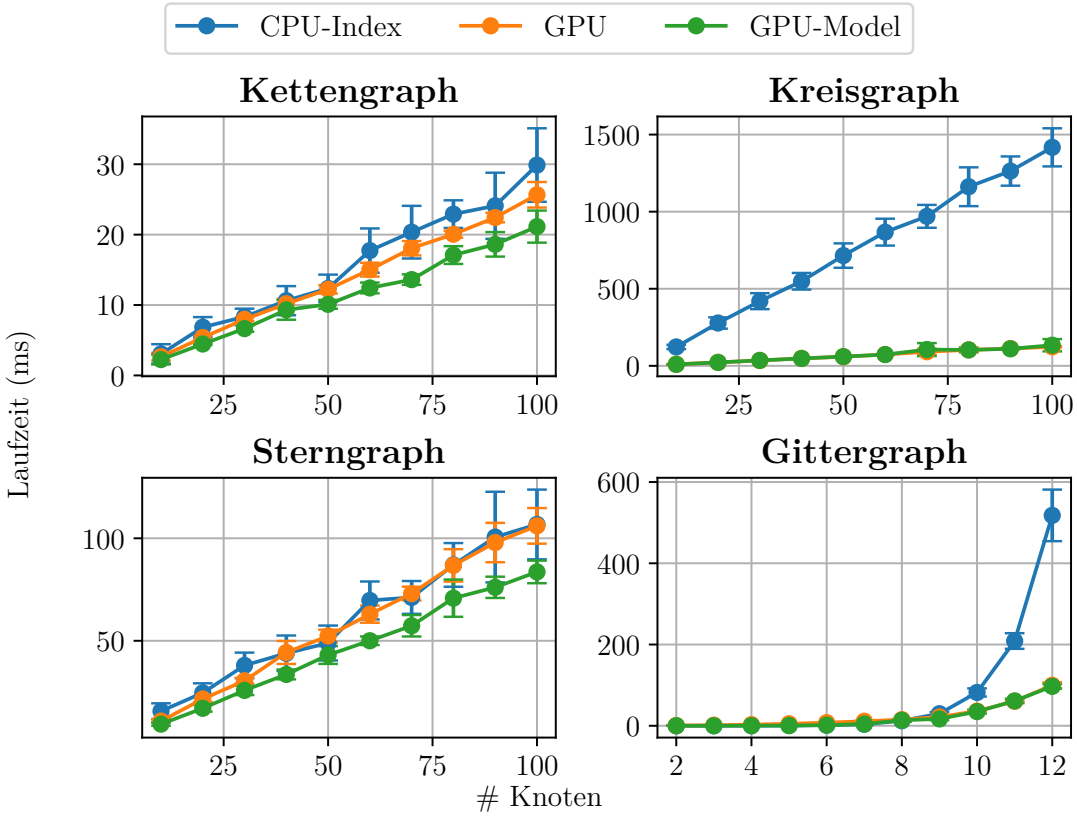


Abbildung 5.6: In dieser Abbildung ist die mittlere Laufzeit in Millisekunden für eine variierende Anzahl an Knoten im Vergleich zwischen der CPU, GPU und modellbasierten Implementierung für unterschiedliche Graphtypen zu sehen.

In den Experimenten wurde gezeigt, dass der hybride modellbasierte Schedulingansatz gut funktioniert und die Laufzeit der Inferenzroutine verringern kann, indem er die beste Plattform für ein Problem auswählt. Graphen mit vielen sehr großen Cliquen profitieren kaum von dieser Technik, erleiden jedoch auch keine Nachteile, da die linearen Modelle sehr effizient angewendet werden können. Deshalb wird diese Technik im weiteren Verlauf der Experimente angewandt.

5.5 Wahl der Wurzel

Der nächste Schritt der Experimente besteht aus der Evaluierung der Strategie zur Wahl der Wurzel. In diesen Experimenten wurde, wie auch bei den Index-Mappings, nur die Laufzeit des eigentlichen Message-Passing betrachtet, da nur dieses durch die Auswahl beeinflusst wird. Genutzt wurde die in Abschnitt 4.3 beschriebene Multi-GPU-Implementierung. Verglichen wurde die Standardstrategie, den Knoten mit dem Index null zu wählen mit der Strategie den Knoten mit der höchsten Betweenness Centrality zu wählen und der zufälligen Wahl der Wurzel. Die Ergebnisse sind in Tabelle 5.4 dargestellt. In der oberen

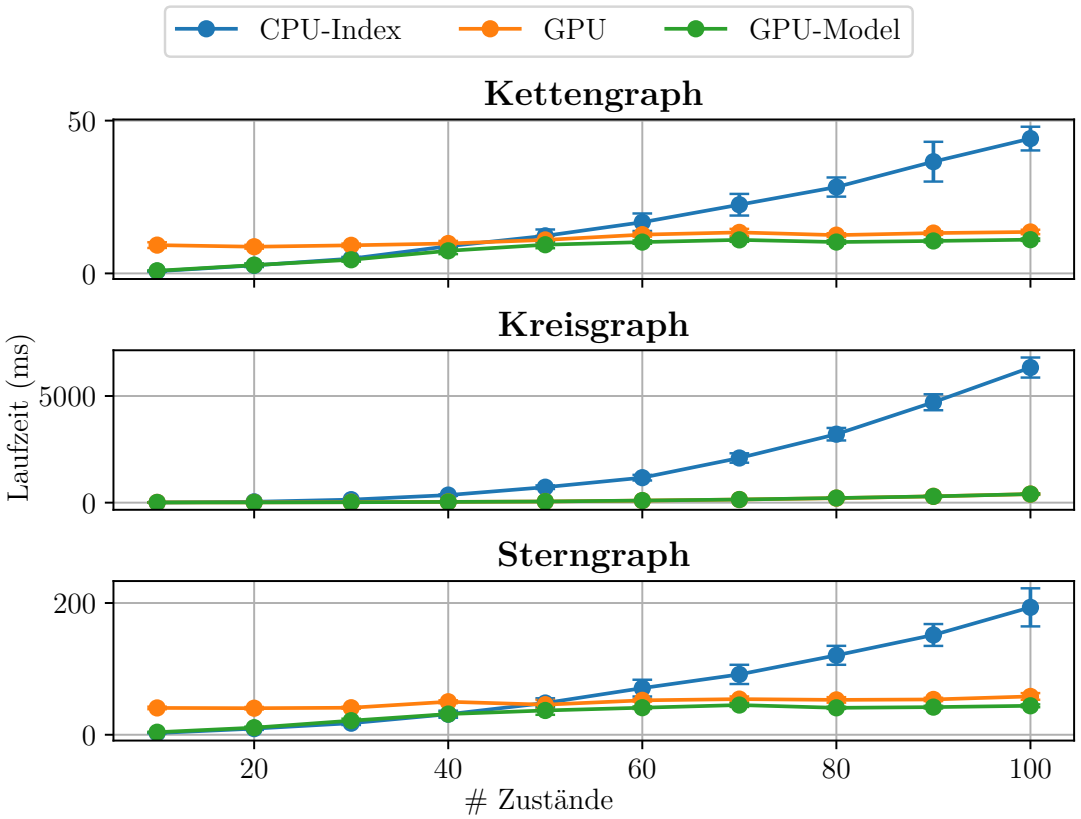


Abbildung 5.7: In dieser Abbildung ist die mittlere Laufzeit in Millisekunden für eine variierende Anzahl an Zuständen im Vergleich zwischen der CPU, GPU und modellbasierten Implementierung zu sehen. Der Ketten- und Kreisgraph bestand aus jeweils 50 Knoten und der Sterngraph hatte acht Arme mit jeweils 50 Knoten.

Hälfte ist die Länge des längsten Pfades von der Wurzel zu einem Blatt zu sehen. Die Idee hinter der Betwenness-Strategie ist, dass die Wahl eines möglichst zentralen Knoten die Länge aller Pfade von den Blättern zur Wurzel verkürzt und es mehr parallele Pfade ähnlicher Länge gibt. Es ist zu sehen, dass es in allen Fällen kürzere Pfade als bei Nutzung der Standardstragie gibt. Insbesondere bei dem Junction-Tree des Gittergraphen kann die Länge drastisch verkürzt werden. Dies spiegelt sich auch in der Laufzeit wieder – die Inferenzroutine benötig auf einem Grid nur 60% der originalen Laufzeit. Auf den anderen Datensätzen ist die Strategie nicht signifikant schneller. Ebenfalls wurde die Strategie für künstliche Graphen mit einer variablen Anzahl an Knoten getestet (siehe Abbildung 5.8). Für den Ketten-, Kreis- und Gittergraph funktioniert die Strategie gut und es kann viel Laufzeit gegenüber den anderen Strategien eingespart werden. Dies lässt sich über die Struktur der Junction-Trees erklären – mit der genutzten Triangulation nehmen die Junction-Trees dieser eine Ketten- oder kettennahe Struktur an. Wenn für den Kettengraph dann der mittlere Knoten als Wurzel gewählt wird, können die Nachrichten von den beiden Blättern perfekt parallel abgearbeitet werden. Insbesondere wenn alle

Tabelle 5.4: In der oberen Hälfte dieser Tabelle ist die Länge des längsten Pfades von einem Blatt zur Wurzel abgebildet. Die untere Hälfte zeigt die Laufzeit des Message-Passing für verschiedene Wahlen der Wurzel.

	alarm	child	insurance	win95pts	hailfinder	pathfinder	grid15	water
Default	14	7	9	9	15	12	209	9
Betweenness	9	5	6	7	9	10	105	6
Random	11	8	5	9	9	16	158	8
Default	1.385	**0.616**	2.726	3.610	3.287	**8.631**	554.408	510.413
Betweenness	1.378	0.652	**2.326**	**3.184**	**3.086**	8.752	**298.727**	**508.984**
Random	**1.362**	0.740	2.466	3.421	3.330	8.776	412.350	513.612

Cliquen dieselbe Zustandsgröße besitzen, gibt es auf jedem Pfad die selbe Menge an Arbeit. Zusammenfassend lässt sich sagen, dass diese Strategie zwar ihren Zweck die Pfade zu verkürzen erfüllt, sie jedoch nur für Kettengraphen eine Verkürzung der Laufzeit erzielt.

5.6 Multi-GPU

Zuletzt wurde untersucht, wie die parallele und verteilte Implementierung der Junction-Tree-Inferenz mit allen Techniken zusammen funktioniert. Die einzelnen Nachrichten wurden, wie in Abschnitt 4.2.2 beschrieben, parallel berechnet. Verteilt wurden die einzelnen Arbeitspakete auf die verschiedenen GPUs der Maschine mit den in 4.3.2 vorgestellten Algorithmen. Außerdem wurde die Doppelpufferungstechnik und das modellbasierte Scheduling eingesetzt. Diese Variante wurde mit einer GPU-Implementierung verglichen, die ebenfalls diese beiden Features nutzt, jedoch nur auf einer GPU. Außerdem wurden zum Vergleich die Laufzeiten der Baseline sowie der CPU-Implementierung herangezogen. Die Ergebnisse sind in Tabelle 5.5 dargestellt.

Zunächst wird die Laufzeit der Initialisierung betrachtet. Es ist zu erkennen, dass diese auf der CPU schnell mit der Größe der Modelle zunimmt. Der Grund hierfür ist, dass sehr viele Zustände sequentiell kodiert und dekodiert werden müssen, um die entsprechenden Gewichte des Markov Random Fields in die Potentialtabellen des Junction-Trees zu überführen. Die einfache GPU-Implementierung kann die Laufzeit bereits sehr stark verringern, da die gesamte Parallelität der GPU ausgenutzt werden kann. Der größte Speedup dieser Implementierung beträgt 35 und ist beim Modell *water* zu sehen. Unter Hinzunahme von mehreren GPUs konnte dieser Wert sogar noch verbessert werden – es wurde ein Speedup von 79 erreicht. Insgesamt konnten alle Modelle während der Initialisierung von der Multi-GPU-Implementierung profitieren und erreichten oft gute Speedups gegenüber der CPU-Implementierung.

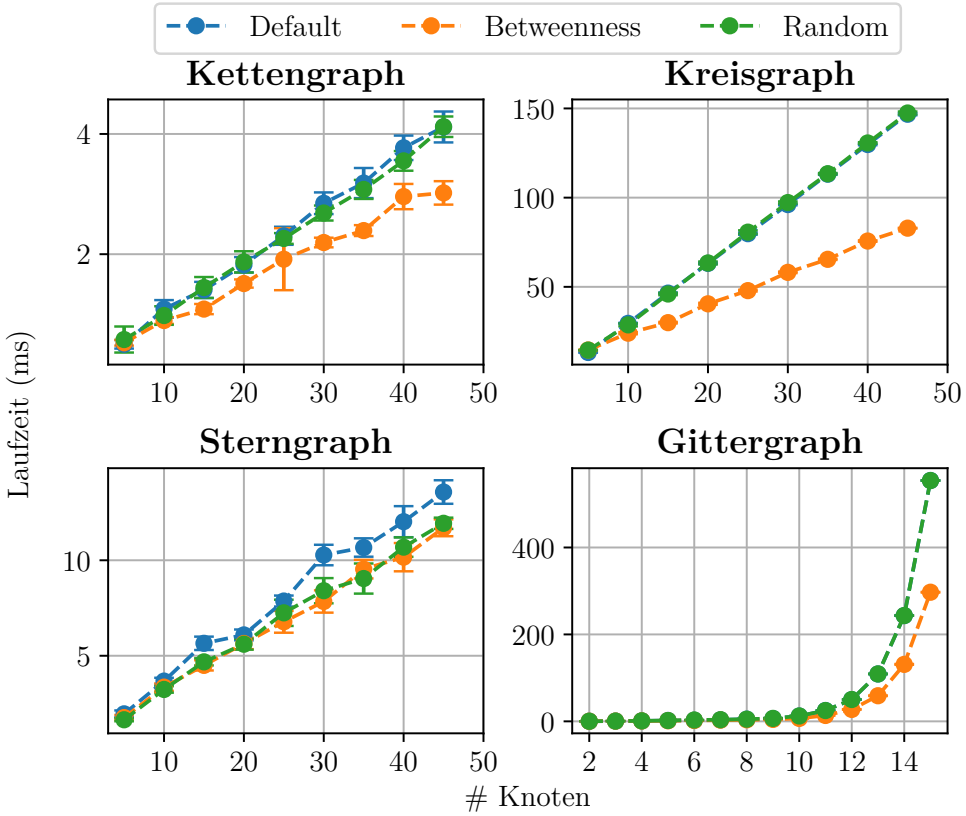


Abbildung 5.8: Es ist die mittlere Laufzeit der Multi-GPU-Implementierung in Abhängigkeit der Anzahl an Knoten und Wahl der Wurzel zu sehen.

Wird hingegen die Laufzeit des Message-Passing betrachtet, zeichnet sich ein anderes Bild. Die CPU-Implementierung auf Basis der Index-Mapping-Vektoren erreicht bereits Speedups von $10 - 20$ im Vergleich zur Baseline. Von der GPU-Implementierung werden gegenüber der index-basierten Implementierung nur kleine Speedups von $2 - 3$ erreicht. Dieser Effekt ist möglicherweise darauf zurückzuführen, dass das Problem eher durch die Speicherbandbreite als die parallele Rechenleistung begrenzt ist. Auf Seiten der GPU führen die unregelmäßigen Speicherzugriffe durch die Indizes zu unvorteilhaften Zugriffsmustern, wodurch die Speicherlatenz weiter erhöht wird. Verschiedene Speicherlayouts konnten, wie sie in anderen Arbeiten beschrieben wurden [32, 65], die Laufzeit nicht weiter verringern. Wird die Multi-GPU-Implementierung betrachtet, ist diese beim Message-Passing nur in der Hälfte der Fälle besser als die Single-GPU-Implementierung und es kann meistens nur ein marginaler Speedup erreicht werden. Nur die Junction-Trees, die eine kettennahe Struktur, wie beispielsweise ein Stern- oder ein Gittergraph, besitzen, können von dem Multi-GPU-Ansatz stark profitieren. Wie im vorherigen Abschnitt erwähnt, funktioniert das Message-Passing für diese Strukturen sehr gut, denn bei einer guten Wahl des Wurzelknoten existieren Pfade ähnlicher Länge mit ähnlich viel Arbeit. Für andere Graphstrukturen lässt sich der

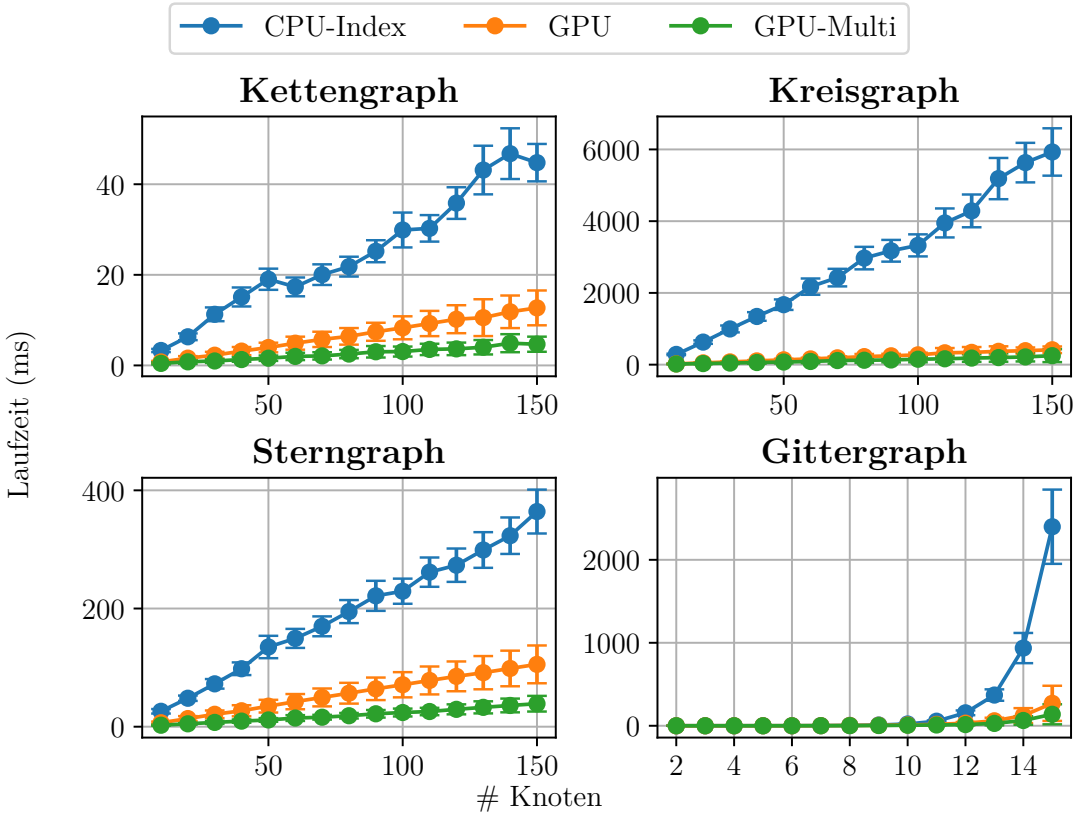


Abbildung 5.9: Es ist die mittlere Laufzeit im Vergleich zwischen der index-basierten CPU-Implementierung, der Single-GPU- und der Multi-GPU-Implementierung in Abhängigkeit der Knoten des Graphen zu sehen.

marginale Speedup, falls überhaupt einer gegeben ist, auf mehrere Probleme zurückführen. Das erste Problem ist das Locking an den Synchronisationsknoten. Falls Prozess eins und Prozess zwei dieselbe Tabelle aktualisieren möchten, muss diese gelockt werden. Angenommen Prozess eins lockt und aktualisiert die Tabelle, muss Prozess zwei auf die Fertigstellung der Operation warten. Nachdem die Tabelle wieder freigegeben wurde, besteht jedoch das Problem, dass Prozess eins die Tabelle im Speicher der ersten GPU aktualisiert hat und der zweite Prozess die Daten zunächst anfordern und auf sie warten muss. Dies zerstört ebenfalls den Flow der Doppelpufferungstechnik, was neben Wartezeiten zu einer weiteren Erhöhung der Latenz führt. Werden die Visualisierungen der Junction-Trees betrachtet (siehe Anhang A.2), ist zu erkennen, dass viele der vermeintlich parallelen Pfade gar nicht parallel ausgeführt werden können, da sich der nächste Synchronisationsknoten nur einen Schritt näher in Richtung der Wurzel befindet. Dies fällt speziell bei den Junction-Trees der Datensätze *win95pts*, *hailfinder* und *pathfinder* auf. Da der Junction-Tree in der Single-GPU-Implementierung nur von einem Thread traversiert wird, können diese Probleme vermieden werden. Es muss nie gelockt werden und nur die nötigsten Daten

Tabelle 5.5: Es sind die mittleren Laufzeiten der jeweiligen Implementierungen pro Datensatz und Operation zu sehen.

		alarm	child	insurance	win95pts	hailfinder	pathfinder	grid15	water
Initialisierung	CPU	0.257	0.118	56.605	21.295	19.507	61.053	5824.415	36208.234
	CPU-Index	0.266	0.099	61.341	23.700	19.326	57.199	5902.633	36252.126
	GPU	0.288	0.120	2.492	3.563	1.137	5.633	175.274	1035.849
	GPU-Multi	**0.163**	**0.062**	**2.277**	**2.480**	**0.694**	**3.880**	**100.442**	**454.070**
MP	CPU	0.567	0.399	54.529	15.773	25.330	112.712	24119.804	24874.079
	CPU-Index	**0.127**	**0.079**	4.988	**1.102**	2.676	13.522	880.311	1571.897
	GPU	0.316	0.194	2.374	1.360	2.750	8.467	547.682	**461.909**
	GPU-Multi	0.351	0.238	**2.036**	1.671	**2.439**	**7.309**	**302.495**	478.594
Normalisierung	CPU	0.119	0.057	3.177	0.535	1.420	8.234	630.573	1435.361
	CPU-Index	0.124	**0.047**	3.330	0.589	1.420	7.863	643.802	1449.868
	GPU	0.111	0.062	0.956	0.422	**0.436**	3.087	77.334	64.516
	GPU-Multi	**0.074**	0.059	**0.257**	**0.273**	0.597	**1.331**	**17.937**	**22.782**
Summe	CPU	0.944	0.573	114.311	37.603	46.257	181.999	30574.792	62517.674
	CPU-Index	**0.516**	**0.225**	69.659	25.391	23.422	78.584	7426.745	39273.891
	GPU	0.715	0.375	5.822	5.345	4.323	17.186	800.289	1562.273
	GPU-Multi	0.588	0.359	**4.57**	**4.425**	**3.730**	**12.521**	**420.874**	**955.446**

werden kopiert. Anstatt Potentialtabellen zwischen zwei Devices hin und her zu kopieren, liegen die aktuellsten Änderungen immer im GPU-Speicher und können entweder zurück in den Hauptspeicher geschrieben oder direkt weiterverwendet werden. Außerdem hängt die Laufzeit größtenteils von wenigen großen Cliquen ab, die eher zentral im Junction-Tree gelegen sind und ebenfalls nicht verteilt berechnet werden können [55]. Zur Verdeutlichung wurde die Größe der Potentialtabellen und die mögliche Parallelität in Abhängigkeit der Distanz zur Wurzel in Abbildung 5.10 visualisiert. Um die Grafik zu erstellen, wurde die Distanz jedes Knotens des Junction-Trees zur gewählten Wurzel berechnet. Anschließend wurden die Größen der Potentialtabellen extrahiert und nach der Distanz gruppiert und aufsummiert. Dies spiegelt in etwa die Arbeit in Abhängigkeit von der Distanz zur Wurzel wider. Auf der anderen Seite wurde überprüft, wie viele Knoten überhaupt mit der jeweiligen Distanz zur Wurzel existieren, da dies eine obere Schranke für die Parallelität ist. Diese wird außerdem weiter durch das Locken an den Synchronisationsknoten eingeschränkt. In der Grafik sind die Ebenen, an denen mindestens ein solcher Knoten existiert, mit einer grauen vertikalen Linie markiert. In vielen Datensätzen muss also in mindestens jeder zweiten Ebene gelockt und gegebenenfalls auf einen Prozess gewartet werden. Die grüne Linie stellt das Verhältnis aus Arbeit zu Parallelität dar. Wenn dieses niedrig ist, sollte die Berechnung schnell gehen, denn entweder kann viel parallel berechnet werden oder es existiert wenig Arbeit. Für alle realen Datensätze zeigt sich ein ähnlicher Trend. Weit entfernt von der Wurzel sind nur kleine Cliquen gelegen, es gibt jedoch verhältnismäßig viele Knoten, die parallel berechnet werden können. Näher an der Wurzel dreht sich dieser Trend – es muss viel berechnet werden, da große Potentialtabellen aktualisiert werden

müssen, jedoch kann an der Wurzel nur ein Prozess die Tabelle aktualisieren. Außerdem wird am Plot des Grids deutlich, wieso es hier so gut funktioniert. Es muss nur an der Wurzel synchronisiert werden und es gibt durchgehend eine gleichmäßige Verteilung der Arbeit auf die unterschiedlichen Knoten. Allerdings existieren nur zwei parallele Pfade, weshalb maximal zwei GPUs genutzt werden können und sich der maximal erreichbare Speedup auf zwei beläuft. Zusammenfassend lässt sich zum Message-Passing sagen, dass die Multi-GPU-Implementierung außer für kettenähnliche Junction-Trees nur marginale Vorteile oder sogar Nachteile mit sich bringt. Kann viel parallel berechnet werden, gibt es meist wenig zu berechnen und vice versa. Mit Blick auf das Amdahlsche Gesetz wird deutlich, dass der Speedup stark beschränkt ist. Dieser ist sowohl durch den längsten Pfad beschränkt, da dieser nicht parallel ausgeführt werden kann, als auch das viele Locken an den Synchronisationsknoten. Der verbleibende erreichbare Speedup wird durch die Wartezeit und die teuren Kopieroperationen zwischen den Devices minimiert.

Bei der Normalisierung sehen die Ergebnisse wieder vielversprechender aus. In sechs von acht Fällen weist die Multi-GPU-Implementierung die geringste Laufzeit auf. Wie vermutet, skaliert diese Routine aufgrund ihrer Unabhängigkeit der Arbeitspakete untereinander direkt mit der Anzahl an GPUs. Der beste Speedup gegenüber der CPU-Implementierung kann auf dem *water*-Datensatz erreicht werden und beläuft sich auf 63.

Abschließend lässt sich sagen, dass sich der Multi-GPU-Ansatz für große Junction-Trees lohnt. Insbesondere die Initialisierung und Normalisierung profitieren extrem von der Parallelität, die moderne GPUs bieten. Ferner können diese Operationen weiter durch die Verteilung der Berechnungen auf mehrere GPUs profitieren. Außerdem hat sich gezeigt, dass das eigentliche Message-Passing nach Nutzung der Index-Mapping-Vektoren gar nicht der Flaschenhals war, sondern die Initialisierung, welche erfolgreich beschleunigt werden konnte. Bei der verteilten Message-Passing-Implementierung konnten, außer für besondere Strukturen, nur marginale Speedups erreicht werden.

Limitierungen

Allerdings sollte auch erwähnt werden, dass dieses Verfahren unter der exponentiell wachsenden Größe der Potentialtabellen leidet. Während der Experimente wurde ebenfalls versucht, dieses Verfahren für größere als den *water*-Datensatz einzusetzen. Ein Beispiel wäre der *barley*-Datensatz, dessen Eigenschaften demonstrativ in Tabelle A.1 aufgenommen wurden. Die größte Potentialtabelle des zugehörigen Junction-Tree bestand aus mehr als 600 Millionen Zuständen, für die entsprechend große Puffer (4.8 GB) auf der GPU angelegt werden mussten. Jedoch bot die Nvidia GTX 1080 nicht genügend Speicher, um unter Nutzung der Doppelpufferungs- und Index-Mapping-Technik jeweils eine Gruppe von anliegenden Cliquen zu bearbeiten. Eine weitere Partitionierung einer Potentialtabelle und die Verteilung dieser auf unterschiedliche GPUs wäre in Anbetracht der raschen CPU-Laufzeit

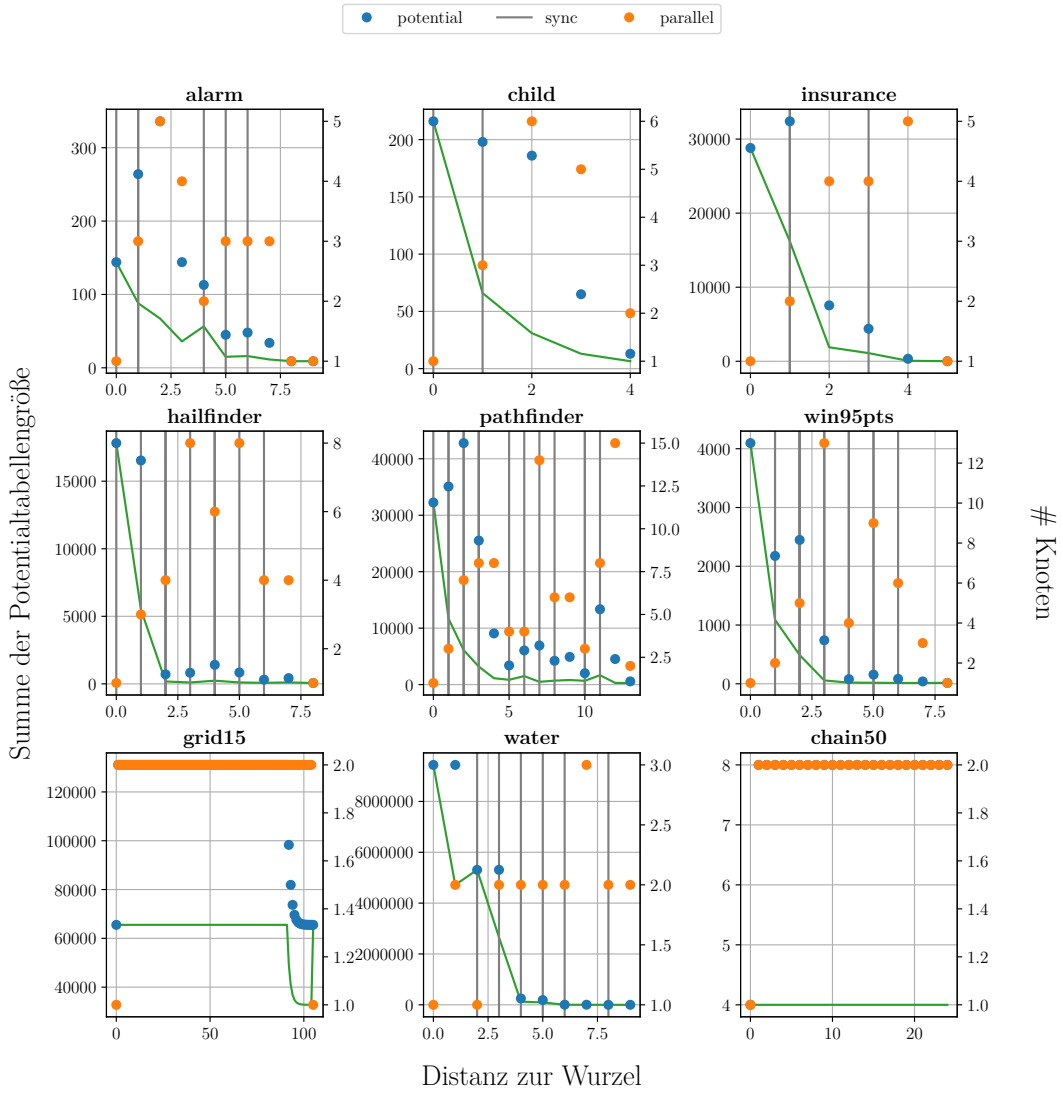


Abbildung 5.10: Zu sehen sind die Größen der Potentialtabellen und die maximale topologiebasierte Parallelität in Abhängigkeit zur Distanz zur Wurzel. Auf der X-Achse ist diese zu sehen. Die linke Y-Achse stellt die Summe der Potentialtabellengröße gruppiert nach Distanz dar. Auf der rechten Y-Achse ist die Anzahl an Knoten der jeweiligen Ebene zu sehen. Die vertikalen grauen Balken markieren die Ebenen, an denen synchronisiert werden muss. Die grüne Linie stellt das Verhältnis aus Arbeit durch Parallelität dar.

nicht sinnvoll gewesen. Eine weitere Einschränkung ist der gesamte Speicherbedarf der Index-Mapping-Vektoren auf Seiten der CPU. Für die großen Datensätze kam es nicht selten vor, dass bis zu mehreren 100 GB an Hauptspeicher benötigt wurde. Ebenfalls konnte das Verfahren nicht zur Inferenz auf dem Junction-Tree des ICE-Cube-Netzwerkes angewandt werden, da die Potentialtabellen mit $\mathcal{O}(20^{|\mathcal{C}|})$ wuchsen. Somit kann diese Implementierung für Junction-Trees mit sehr großen Zustandsraum nicht verwendet werden.

Da die Initialisierung den größten Anteil der Laufzeit einnimmt, könnte diese, falls der Speicherbedarf der größten Potentialtabelle die Menge an verfügbarem GPU-Speicher nicht übersteigt, weiterhin auf der GPU ausgeführt werden.

5.7 Diskussion

In diesem Kapitel wurden die Implementierung einer parallelen und verteilten Variante der Junction-Tree-Inferenz evaluiert. Zu Beginn wurde die Effektivität der verschiedenen Techniken überprüft. Die Index-Mapping-Vektoren konnten die Laufzeit des Message-Passing sowohl auf der CPU als auch auf der GPU stark verkürzen. Anschließend wurde gezeigt, dass die Nutzung der Doppelpufferungstechnik fast immer von Vorteil ist. Da viele Cliquen des Junction-Trees nur einen kleinen Zustandsraum besitzen, lohnt sich die Ausführung dieser auf der GPU nicht. Zur Lösung dieses Problems wurden Laufzeitmodelle entwickelt und basierend auf deren Vorhersage die zur Clique besser passende Plattform gewählt. Anschließend wurde untersucht, ob die Wahl der Wurzel einen Einfluss auf die Laufzeit nimmt. Für Graphen, deren Junction-Tree die Struktur eines Kettengraphen annahm, konnten gute Speedups erreicht werden. Andere Graphen, die aus vielen Synchronisationsknoten und nur kurzen oder nicht parallel ausführbaren Pfaden bestanden, konnten von dieser Technik nicht profitieren. Zuletzt wurde untersucht, wie sich die zusammengesetzte Multi-GPU-Implementierung im Vergleich zur einfachen GPU und den CPU-Implementierungen schlägt. Die GPU- und Multi-GPU-Implementierung gingen als Sieger aus diesem Vergleich hervor. Bei der Initialisierung und Normalisierung konnte die Multi-GPU-Implementierung durch die Verteilung der Berechnungen auf mehrere GPUs profitieren, jedoch konnten beim Message-Passing aufgrund einiger Problematiken nur marginale Speedups erzielt werden. Zur Verbesserung könnte zur Laufzeit entschieden werden, ob es sich bei einem Junction-Tree um einen Kettengraph handelt und falls ja, könnte auch das Message-Passing mit der geeigneten Wurzel auf mehreren GPUs ausgeführt werden. Insgesamt konnten durch die vorgestellten Techniken beachtliche Speedups sowohl für echte als auch künstliche Datensätze erzielt werden.

Außerdem sollten die Limitierungen nicht außer Acht gelassen werden, Aufgrund der exponentiell wachsenden Größe der Potentialtabellen können große Probleme nicht gelöst werden oder sind mit einem erheblichen Speicherverbrauch verbunden. Möglicherweise können die Probleme durch eine andere Technik zur Triangulation eingeschränkt werden, denn der MCS-Algorithmus ist bekannt dafür [2], dass er teils sehr große Cliquen produziert. Jedoch können aufgrund der exponentiellen Komplexität die Probleme nie komplett eliminiert werden. Die parallele und verteilte Junction-Tree-Inferenz lässt sich also gut für größere Probleme anwenden, solange die Speicherlimitationen auf dem Device für die Puffer und auf dem Host für die Indizes nicht ausgereizt sind. Ist eine dieser Limitierungen verletzt,

sollte besser auf eines der in Kapitel 2 vorgestellten approximativen Inferenzverfahren [3,33] zurückgegriffen werden.

Kapitel 6

Fazit und Ausblick

Das Ziel dieser Arbeit war es, eine parallele und verteilte Implementierung der Junction-Tree-Inferenz zu entwickeln. In Kapitel 2 wurden dazu zunächst die benötigten theoretischen Grundlagen zu dem Verfahren erläutert. Das darauffolgende Kapitel 3 hat die Herausforderungen und Chancen der parallelen Programmierung näher dargestellt. Außerdem wurden die genutzten Techniken Nvidia CUDA und OpenMP dort vorgestellt. Anschließend wurde in Kapitel 4 beschrieben, wie die einzelnen Operationen der Inferenz auf der CUDA-Plattform beschleunigt wurden. Da die einzelnen Operationen aus mehreren, teils unabhängigen Funktionsaufrufen bestehen, wurden diese Unabhängigkeiten identifiziert. Anhand dieser wurden Schedules erstellt, welche die Verteilung der Berechnungen auf verschiedene GPUs ermöglicht. Des Weiteren wurden diverse Techniken zur Optimierung der Datentransfers, wie beispielsweise Pinned-Memory oder Pipelining, vorgestellt. Außerdem wurde ein hybrides, modellbasiertes Schedulingsystem entwickelt, welches basierend auf den Eingabeparametern des Algorithmus die beste Plattform für die Eingabe auswählt. Schließlich wurde die Implementierung in Kapitel 5 mit Blick auf die Laufzeit evaluiert. Zunächst wurde die Effektivität der einzelnen Techniken nachgewiesen. Anschließend wurde untersucht, wie die kombinierte Multi-GPU-Implementierung im Vergleich zur Single-GPU- und den CPU-Implementierungen abschneidet. Für Modelle, die genügend Parallelität bereitstellen, konnte die Single-GPU-Implementierung die CPU-Varianten in der Laufzeit unterbieten. Durch die Verteilung der Berechnungen konnten insbesondere die Initialisierung und die Normalisierung profitieren. Die parallele Ausführung der Pfade von den Blättern zur Wurzel führte aufgrund diverser Einschränkungen nur zu marginalen Speedups. Nur für Graphen, deren Junction-Trees die Struktur eines Kettengraphen annahm, konnte mit geeigneter Wahl der Wurzel ein guter Speedup durch die Verteilung der Berechnungen erzielt werden. Insgesamt konnten beachtliche Speedups durch die Parallelisierung und Verteilung der Berechnungen erzielt werden. Jedoch sei erwähnt, dass aufgrund der exponentiellen Komplexität weiterhin einige Limitierungen existieren. Für große Graphen können die Indizes sehr viel Hauptspeicher benötigen. Ebenfalls ist das größte lösbare

Problem durch die Größe des globalen GPU-Speichers limitiert. Zusammenfassend lässt sich sagen, dass zunächst versucht werden sollte, ein Problem mit exakter Inferenz zu lösen. Falls jedoch eine der Limitierungen greift, kann auf eines der vorgestellten approximativen Inferenzverfahren zurückgegriffen werden.

6.1 Ausblick

In zukünftigen Arbeiten könnte untersucht werden, wie eine Multi-Core-CPU-Implementierung des Message-Passing im Vergleich zur Multi-GPU-Implementierung abschneidet. Möglicherweise könnten geeignete CPUs durch ihre riesigen Caches die Latenz, welche durch unregelmäßige Speicherzugriffe entsteht, verringern. Außerdem bieten diese, insbesondere wenn sie im Dual-Socket-Modus genutzt werden, ebenfalls viele Threads. Wenn diese über das hybride Schedulingsystem eingebunden werden, könnten insgesamt noch mehr Operationen parallel ausgeführt werden, da die Kernel parallel zu Berechnungen auf der CPU laufen können. In diesem Zuge könnten außerdem die Laufzeitmodelle aktualisiert und um hardwarespezifische Merkmale, wie beispielsweise die Taktrate, Anzahl an Kernen und Cacheeigenschaften, erweitert werden. Anschließend könnten Trainingsdaten auf verschiedenen Systemen akquiriert und untersucht werden, ob sich die Laufzeit sowohl in Abhängigkeit der verwendeten Hardware als auch der Eingabegrößen gut modellieren lässt. Falls diese Modelle gut generalisieren, könnte in der Zukunft auf die Datenakquisition und Konfiguration der Modelle pro System verzichtet werden. Außerdem wäre es interessant andere Techniken zur Triangulation einzusetzen, insbesondere solche, die Junction-Trees mit kleineren Potentialtabellen erzeugen [10, 15]. Dies würde die Inferenz auf größeren Graphen ermöglichen, welche eventuell noch stärker von der Parallelität profitieren könnten. Außerdem wäre es interessant die selbe Technik zur Triangulation wie in den anderen Veröffentlichungen zu nutzen, um vergleichbare Junction-Trees zu erhalten. Jedoch wurde nicht dokumentiert, welche Technik genutzt wurde. Die Nutzung dieser Technik würde den direkten Vergleich der Implementierungen ermöglichen, welcher so aufgrund drastischer Unterschiede sowohl in der Struktur als auch Größe der Potentialtabellen nicht sinnvoll ist. Außerdem könnten für größere Graphen Techniken eingesetzt werden, welche Junction-Trees zu einer gegebenen Baumweite berechnen [38]. Dies würde es ermöglichen, anhand der Zustandsraumgröße die maximale Baumweite zu errechnen, so dass die Potentialtabellen noch in den Speicher der GPU passen. Des Weiteren könnten untersucht werden, ob andere Triangulationstechniken in Junction-Trees resultieren, deren Strukturen besser durch die Verteilung der Berechnungen profitieren können.

Anhang A

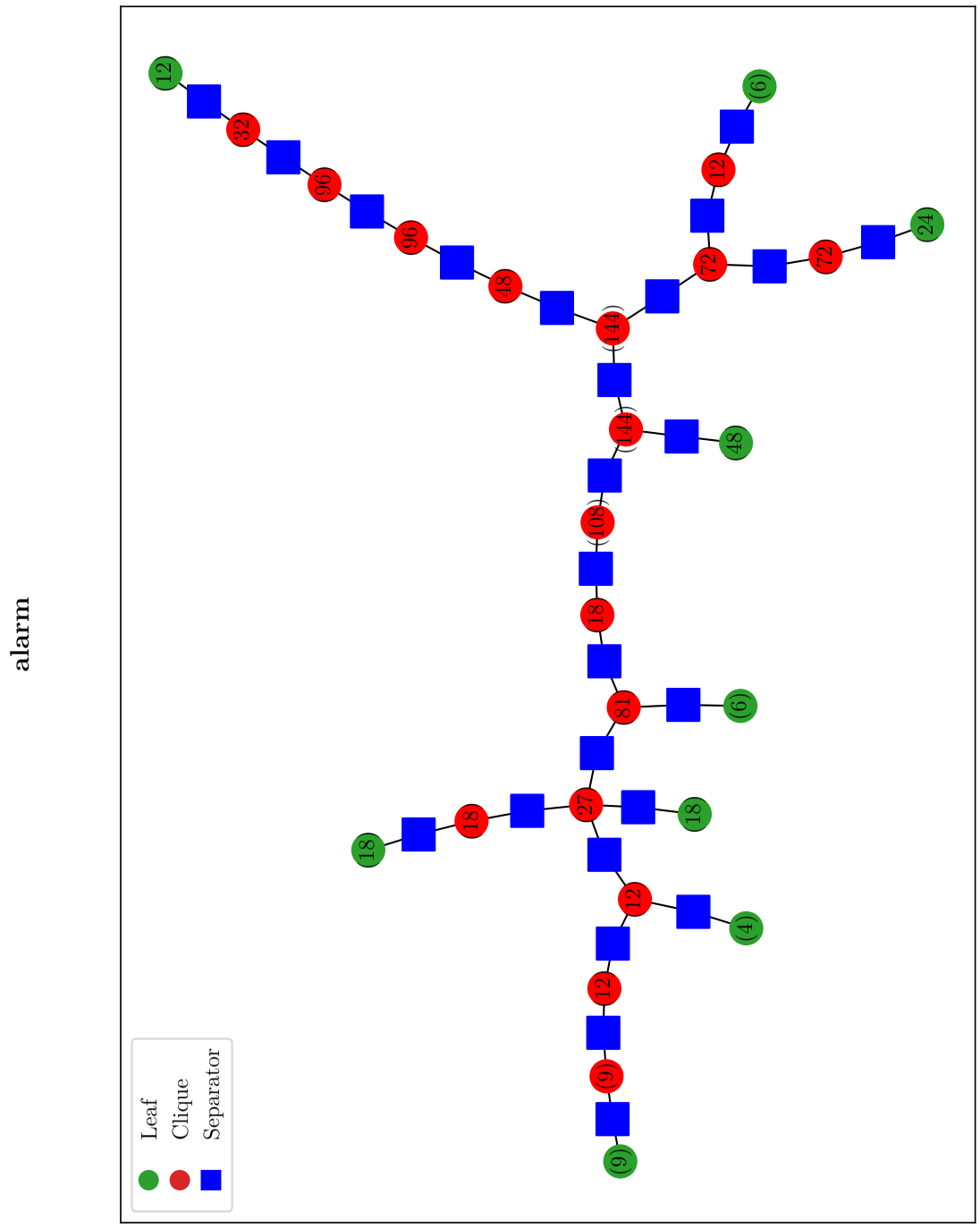
Weitere Informationen

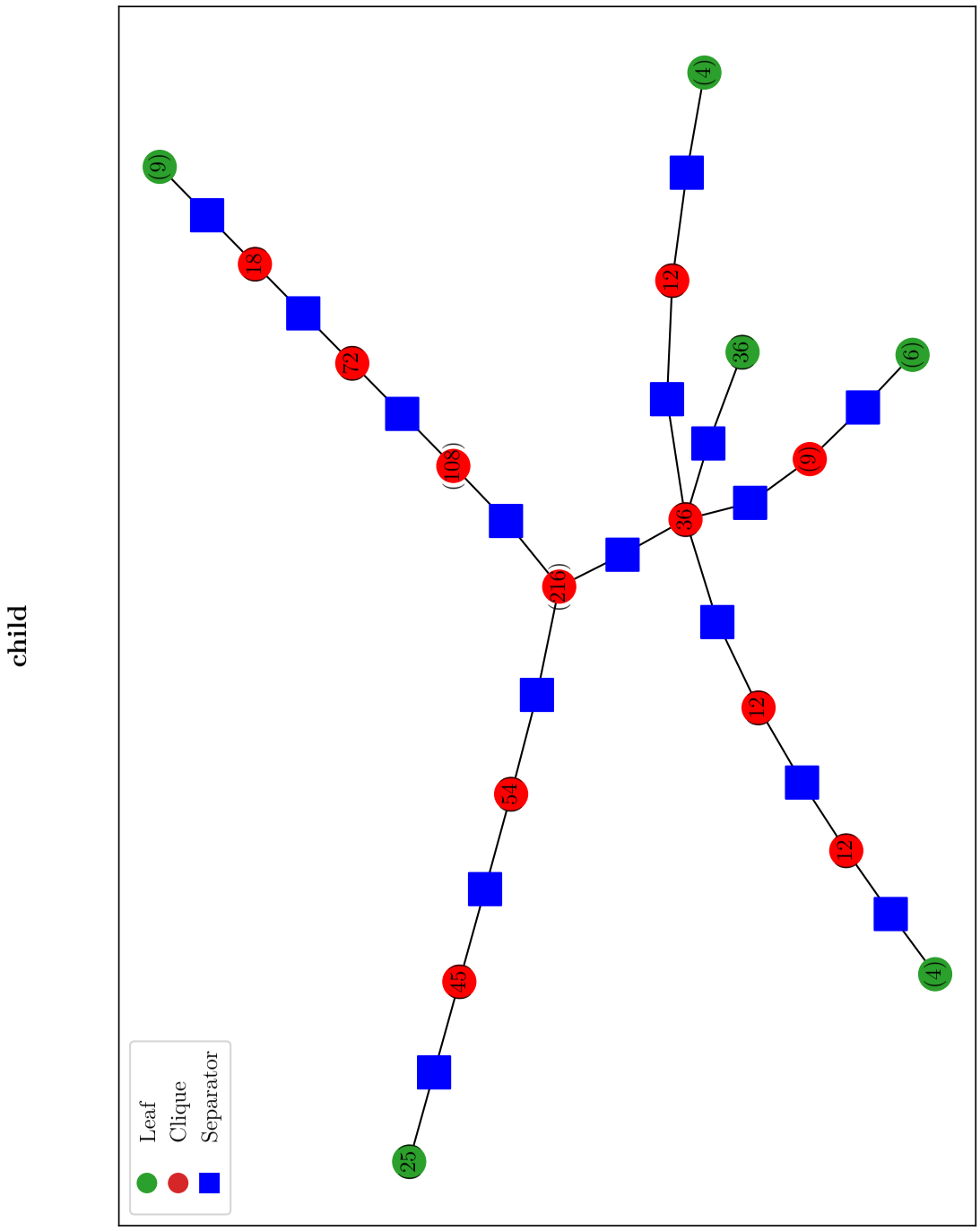
A.1 Eigenschaften der Junction-Trees

Tabelle A.1: In dieser Tabelle sind die Eigenschaften verschiedener Graphen und deren Junction-Trees zu sehen.

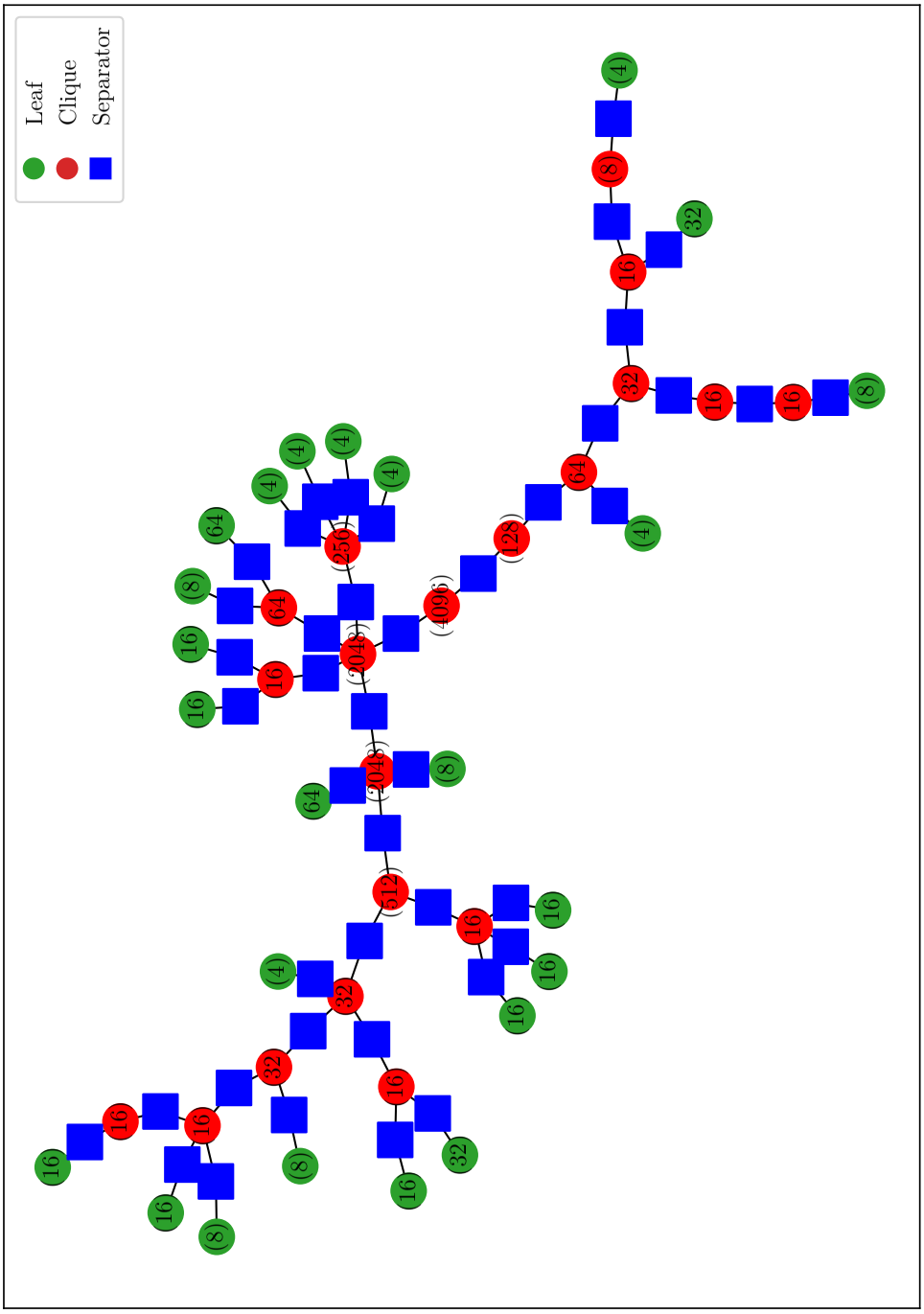
	alarm	child	insurance	win95pts	hailfinder	pathfinder	grid15	water	mildew	barley
Basisgraph # Knoten	37	20	27	76	56	109	225	32	35	48
Basisgraph # Kanten	65	30	70	225	99	208	420	123	80	126
Basisgraph Zustandsgröße (Avg.)	2	3	3	2	3	4	2	3	17	8
Basisgraph Zustandsgröße (Min)	2	2	2	2	2	2	2	3	3	2
Basisgraph Zustandsgröße (Max)	4	6	5	2	11	63	2	4	100	67
JT Zustandsgröße der Cliquen (Avg.)	44	39	4322	223	998	2142	61791	1575651	7264032	58172142
JT Zustandsgröße der Cliquen (Min)	4	4	8	4	16	4	8	9	336	216
JT Zustandsgröße der Cliquen (Max)	144	216	28800	4096	17820	32256	65536	9437184	128832000	600112800
JT Zustandsgröße der Separatoren (Avg.)	10	10	678	56	63	353	30886	251940	162603	3313019
JT Zustandsgröße der Separatoren (Min.)	2	2	4	2	3	2	4	3	72	7
JT Zustandsgröße der Separatoren(Max.)	48	36	7200	1024	1485	8064	32768	2359296	2112000	75014100
JT # Knoten	51	33	33	87	77	177	419	37	55	63
JT # Blätter	9	6	5	24	16	53	2	8	12	12
JT Größte Clique	5	4	9	12	7	8	16	12	6	10
JT # Synchronisationsknoten	7	3	4	15	12	16	1	7	11	10
Größe der Index-Mapping-Vektoren	2654	1596	179236	31172	130429	609592	25886696	68568429	346489976	1268969918
Ebenen des Schedule	3	3	3	9	7	9	0	7	5	5
Längster Pfad (Blatt → Wurzel)	15	8	10	10	16	13	210	10	14	14

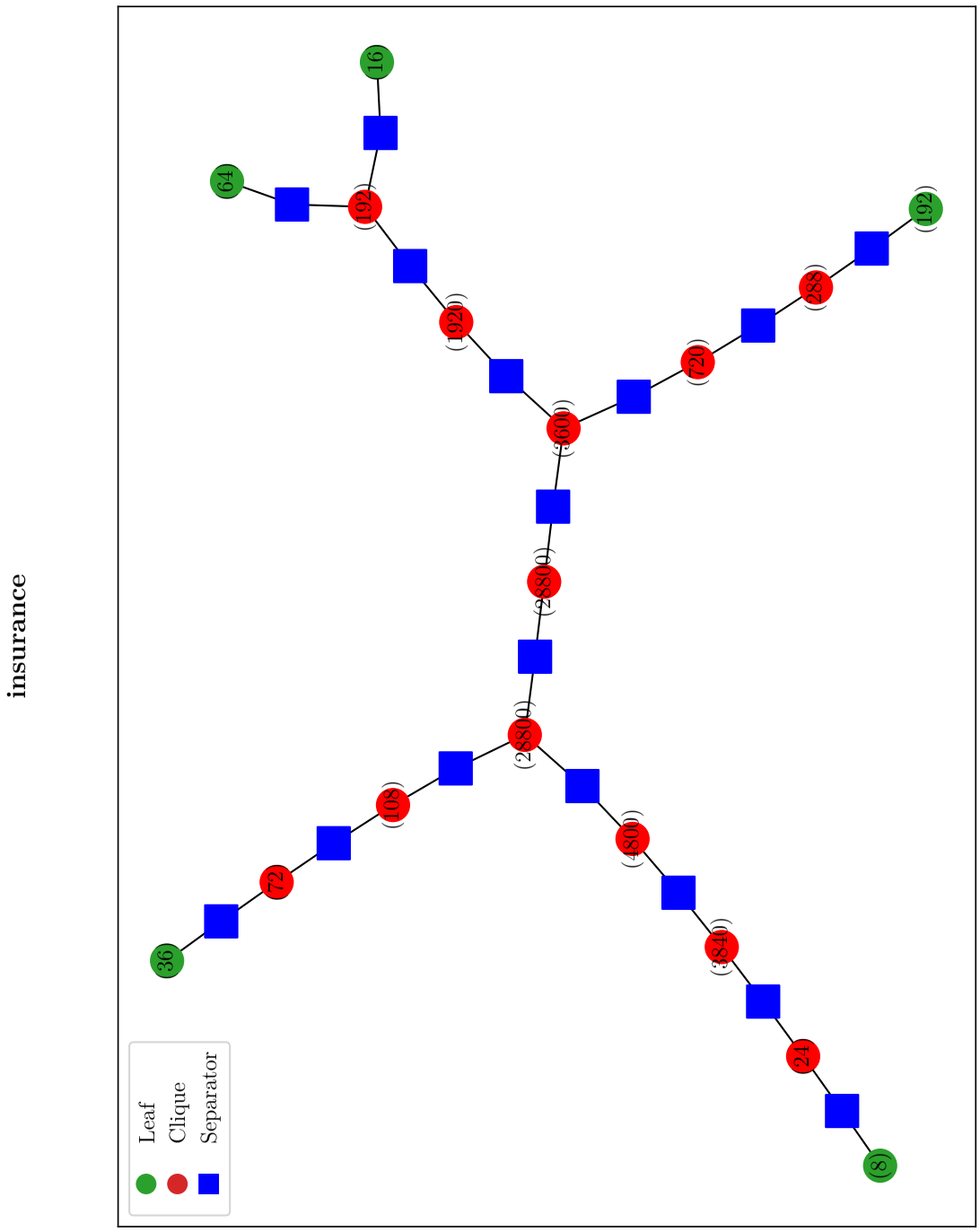
A.2 Junction-Tree Visualisierungen


alarm
Leaf
Clique
Separator
12
32
96
96
48
144
144
48
108
18
81
6
27
18
18
12
4
12
9
9
6
12
72
72
24

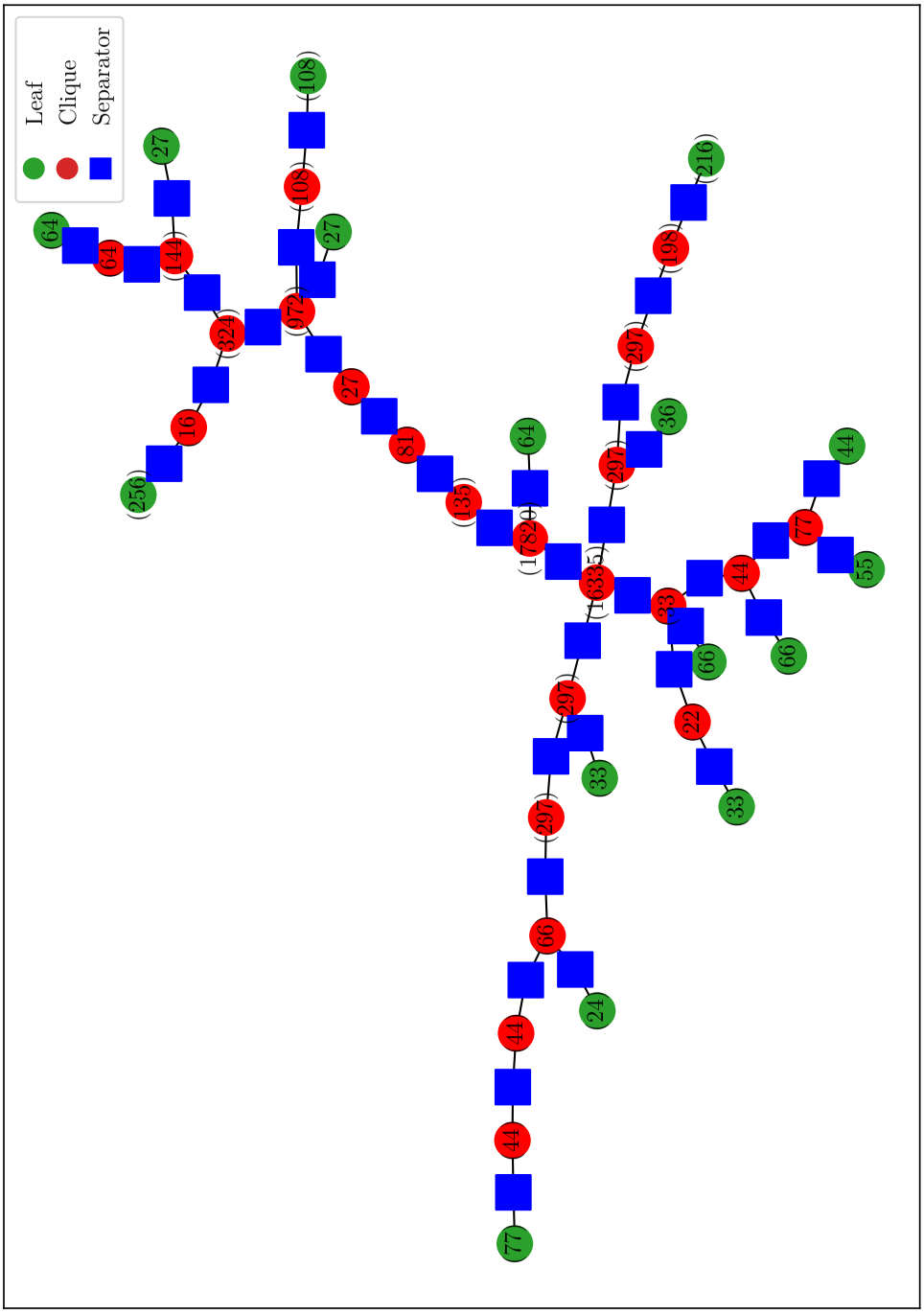

child
Leaf
Clique
Separator
(4)
(9)
18
12
72
36
(6)
(108)
(9)
36
216
12
54
12
45
(4)
25

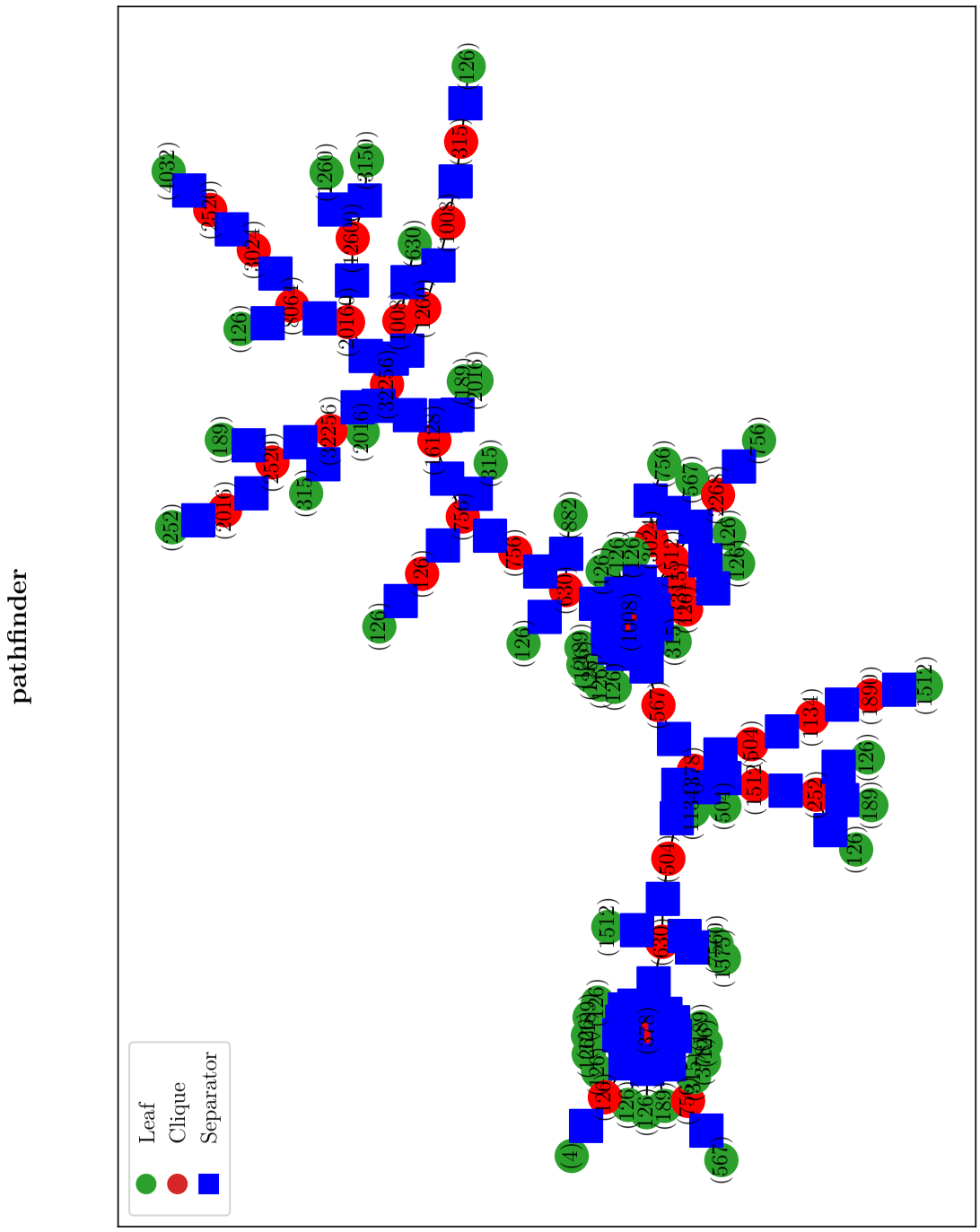
win95pts

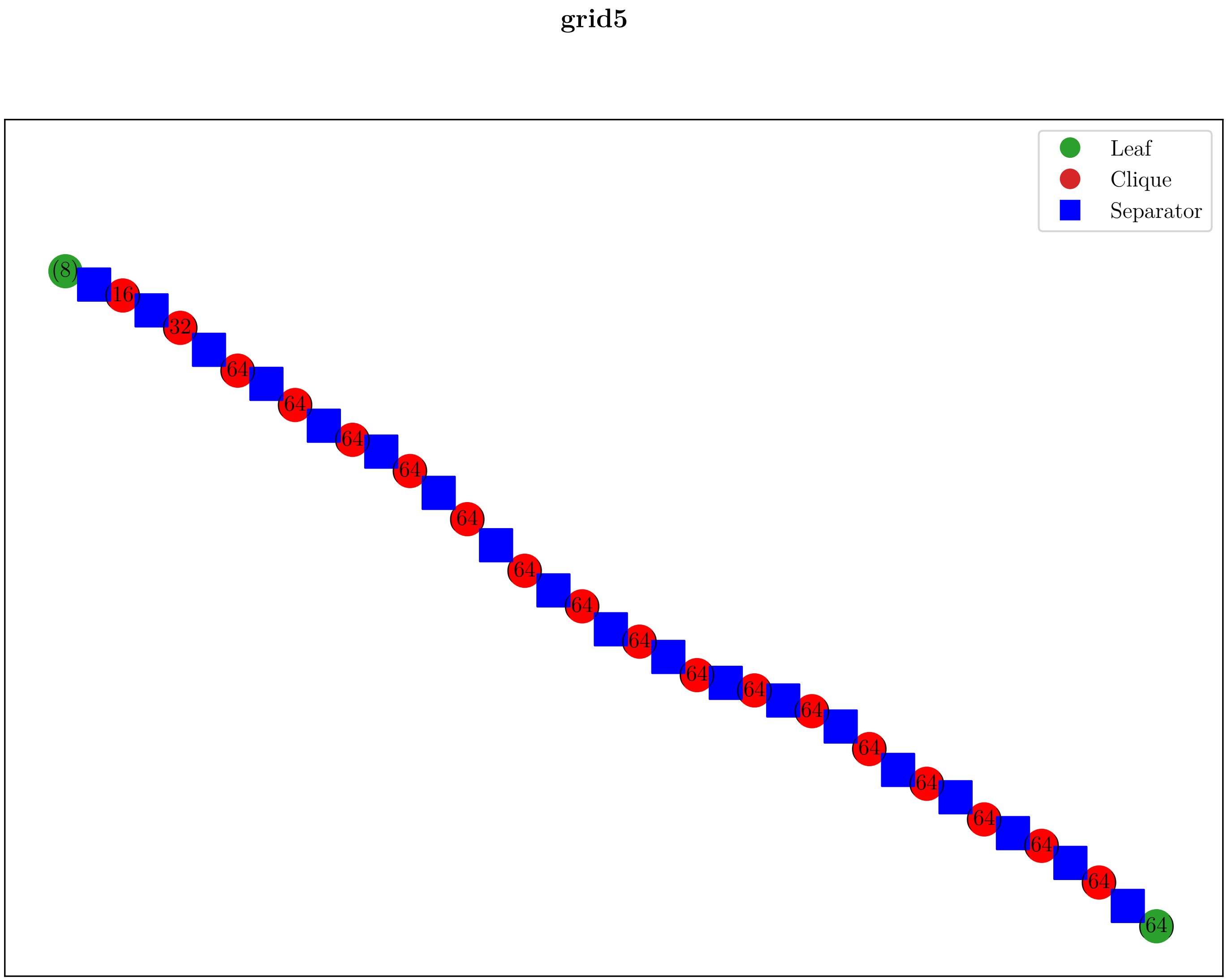



insurance
16
64
(192)
(192)
(720)
(288)
192
(360)
(2880)
(2880)
(2880)
(108)
72
36
(480)
(384)
24
8
Leaf
Clique
Separator

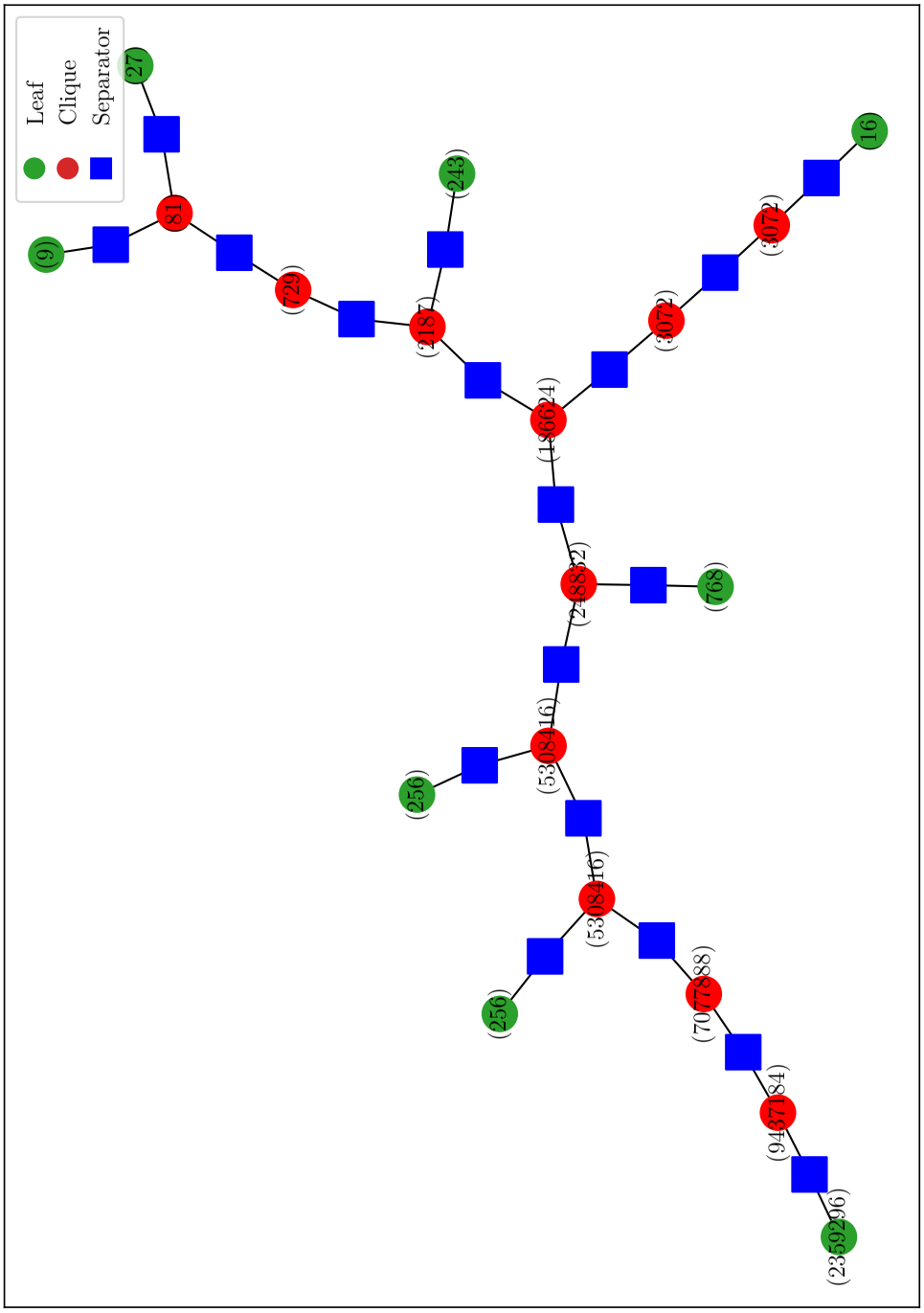
hailfinder




pathfinder
Leaf
Clique
Separator


grid5
Leaf
Clique
Separator
(8)
(16)
(32)
(64)
(64)
(64)
(64)
(64)
(64)
(64)
(64)
(64)
(64)
(64)
(64)
(64)
(64)
(64)
(64)
(64)

water


Leaf
Clique
Separator
27
81
(9)
729
243
2187
16
3072
3072
186624
248832
768
530816
256
530816
256
7077888
9437184
2359296

A.3 Standardabweichung der Tabellen

Tabelle A.2: Standardabweichung der Doppelpufferungstechnik.

		alarm	child	insurance	win95pts	hailfinder	pathfinder	grid15	water
Initialisierung	GPU	0.040	0.037	0.210	0.376	0.188	0.306	9.823	26.050
	GPU-Double-Buffer	0.041	0.030	0.065	0.228	0.067	0.184	23.011	30.654
MP	GPU	0.162	0.093	0.063	0.163	0.059	0.096	1.853	1.261
	GPU-Double-Buffer	0.302	0.195	0.064	0.586	0.437	0.149	0.444	0.429
Normalisierung	GPU	0.019	0.013	0.113	0.037	0.022	0.024	0.055	0.237
	GPU-Double-Buffer	0.022	0.017	0.013	0.029	0.024	0.022	0.378	0.232

Tabelle A.3: Modellbasiertes Scheduling: Standardabweichung

		alarm	child	insurance	win95pts	hailfinder	pathfinder	grid15	water
Initialisierung	CPU-Index	0.034	0.012	9.490	4.444	5.477	10.018	89.243	169.473
	GPU	0.041	0.030	0.065	0.228	0.067	0.184	23.011	30.654
	GPU-Model	0.022	0.040	0.086	0.666	0.092	0.158	16.310	23.960
MP	CPU-Index	0.021	0.013	0.800	0.184	0.625	1.703	35.520	26.405
	GPU	0.302	0.195	0.064	0.586	0.437	0.149	0.444	0.429
	GPU-Model	0.024	0.045	0.101	0.214	0.120	0.402	1.720	0.477
Normalisierung	CPU-Index	0.061	0.006	0.455	0.094	0.358	1.397	21.562	409.157
	GPU	0.022	0.017	0.013	0.029	0.024	0.022	0.378	0.232
	GPU-Model	0.042	0.017	0.033	0.083	0.413	7.238	0.389	0.214
Summe	CPU-Index	0.092	0.028	10.397	4.714	6.306	12.507	96.162	415.571
	GPU	0.313	0.210	0.088	0.446	0.442	0.220	23.067	30.586
	GPU-Model	0.072	0.097	0.181	0.957	0.522	7.525	16.283	23.930

Tabelle A.4: Wahl der Wurzel: Standardabweichung

	alarm	child	insurance	win95pts	hailfinder	pathfinder	grid15	water
Default	0.117	0.109	0.156	0.307	0.219	0.787	2.535	2.032
Betweenness	0.082	0.063	0.092	0.287	0.186	0.581	2.351	3.027
Random	0.083	0.135	0.091	0.253	0.321	0.708	62.426	3.504

Tabelle A.5: Multi-GPU im Vergleich: Standardabweichung

		alarm	child	insurance	win95pts	hailfinder	pathfinder	grid15
Initialisierung	CPU	0.053	0.020	9.847	4.522	4.056	10.233	372.859
	CPU-Index	0.091	0.018	10.923	4.861	3.568	8.081	426.167
	GPU	0.022	0.040	0.086	0.666	0.092	0.158	16.310
	GPU-Multi	0.015	0.010	0.040	0.348	0.082	0.075	38.858
MP	CPU	0.118	0.067	5.551	2.593	3.598	11.298	1256.702
	CPU-Index	0.041	0.013	0.455	0.198	0.420	1.197	73.035
	GPU	0.024	0.045	0.101	0.214	0.120	0.402	1.720
	GPU-Multi	0.161	0.091	0.334	0.539	0.363	1.452	1.538
Normalisierung	CPU	0.053	0.010	0.596	0.083	0.185	1.414	53.612
	CPU-Index	0.068	0.007	0.280	0.098	0.233	1.048	79.474
	GPU	0.042	0.017	0.033	0.083	0.413	7.238	0.389
	GPU-Multi	0.023	0.026	0.053	0.041	0.865	0.250	0.424
Summe	CPU	0.224	0.097	15.995	7.198	7.838	22.945	1683.172
	CPU-Index	0.200	0.038	11.658	5.157	4.220	10.325	578.676
	GPU	0.088	0.102	0.220	0.964	0.625	7.797	18.419
	GPU-Multi	0.199	0.127	0.427	0.928	1.309	1.778	40.820

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] Amdahl, Gene M.: *Validity of the single processor approach to achieving large scale computing capabilities.* In: *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, Seiten 483–485, 1967.

[2] Andersen, S. K., K. G. Olesen und F. V. Jensen: *Readings in Uncertain Reasoning.* Kapitel HUGIN - Shell for Building Bayesian Belief Universes for Expert Systems, Seiten 332–337. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[3] Andrieu, Christophe, Nando de Freitas, Arnaud Doucet und Michael I. Jordan: *An Introduction to MCMC for Machine Learning.* Machine Learning, 50(1):5–43, Jan 2003.

[4] Berry, Anne, Jean R. S. Blair, Pinar Heggernes und Barry W. Peyton: *Maximum Cardinality Search for Computing Minimal Triangulations of Graphs.* Algorithmica, 39(4):287–298, 2004.

[5] Bishop, Christopher M.: *Pattern recognition and machine learning, 5th Edition.* Information science and statistics. Springer, 2007.

[6] Blei, David M., Alp Kucukelbir und Jon D. McAuliffe: *Variational Inference: A Review for Statisticians.* Journal of the American Statistical Association, 112(518):859–877, 2017.

[7] Brandes, Ulrik: *A Faster Algorithm for Betweenness Centrality.* In: *Journal of Mathematical Sociology*, Band 25, Seiten 163–177, 2001.

[8] Casella, George und Edward I. George: *Explaining the Gibbs Sampler.* The American Statistician, 46(3):167–174, 1992.

[9] Chandrasekaran, Venkat, Nathan Srebro und Prahladh Harsha: *Complexity of Inference in Graphical Models.* In: *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, UAI'08, Seiten 70–78, Arlington, Virginia, United States, 2008. AUAI Press.

[10] Chechetka, Anton und Carlos Guestrin: *Efficient Principled Learning of Thin Junction Trees.* In: *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, Seiten 273–280, USA, 2007. Curran Associates Inc.

[11] Cook, Shane: *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st Auflage, 2013.

[12] Cover, Thomas M. und Joy A. Thomas: *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing).* Wiley-Interscience, New York, NY, USA, 2006.

[13] Cowell, Robert G., A. Philip Dawid, Steffen L. Lauritzen und David J. Spiegelhalter: *Probabilistic Networks and Expert Systems: Exact Computational Methods for Bayesian Networks.* Springer Publishing Company, Incorporated, 1st Auflage, 2007.

[14] Cross, G. R. und A. K. Jain: *Markov Random Field Texture Models.* IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-5(1):25–39, Jan 1983.

[15] Dafna, Shahaf und Carlos Guestrin: *Learning Thin Junction Trees via Graph Cuts.* In: Dyk, David van und Max Welling (Herausgeber): *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, Band 5 der Reihe *Proceedings of Machine Learning Research*, Seiten 113–120, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR.

[16] Dagum, Leonardo und Ramesh Menon: *OpenMP: An Industry-Standard API for Shared-Memory Programming.* IEEE Comput. Sci. Eng., 5(1):46–55, Januar 1998.

[17] Dawid, A. P.: *Conditional Independence in Statistical Theory.* Journal of the Royal Statistical Society. Series B (Methodological), 41(1):1–31, 1979.

[18] Forbes, Florence: *Modelling structured data with probabilistic graphical models.* In: Fraix-Burnet, D. und S. Girard (Herausgeber): *Statistics for Astrophysics-Classification and Clustering*, Band 77 der Reihe *EAS Publication Series*, Seiten 195–219. EDP Sciences, 2016.

[19] Forum, Message P: *MPI: A Message-Passing Interface Standard.* Technischer Bericht, Knoxville, TN, USA, 1994.

[20] Freedman, David und Robert Pisani: *Statistics.* Viva Books, 4th ed. Auflage, 2009.

[21] Freeman, Linton C.: *A Set of Measures of Centrality Based on Betweenness.* Sociometry, 40(1):35–41, 1977.

[22] Galinier, Philippe, Michel Habib und Christophe Paul: *Chordal Graphs and Their Clique Graphs.* In: *Graph-Theoretic Concepts in Computer Science, 21st International Workshop, WG '95, Aachen, Germany, June 20-22, 1995, Proceedings*, Seiten 358–371, 1995.

[23] Graham, R. L.: *Bounds for certain multiprocessing anomalies.* The Bell System Technical Journal, 45(9):1563–1581, Nov 1966.

[24] Grama, Ananth, George Karypis, Vipin Kumar und Anshul Gupta: *Introduction to Parallel Computing.* Addison-Wesley, Second Auflage, 2003.

[25] Halzen, Francis und Spencer R. Klein: *Invited Review Article: IceCube: An instrument for neutrino astronomy.* Review of Scientific Instruments, 81(8):081101, 2010.

[26] Hassner, Martin und Jack Sklansky: *The use of Markov Random Fields as models of texture.* Computer Graphics and Image Processing, 12(4):357 – 370, 1980.

[27] Hastie, Trevor, Robert Tibshirani und Jerome H. Friedman: *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition.* Springer series in statistics. Springer, 2009.

[28] Heggernes, Pinar: *Minimal triangulations of graphs: A survey.* Discrete Mathematics, 306(3):297 – 317, 2006. Minimal Separation and Minimal Triangulation.

[29] Herlihy, Maurice und Nir Shavit: *The Art of Multiprocessor Programming.* Morgan Kaufmann, April 2008.

[30] Huang, Cecil und Adnan Darwiche: *Inference in belief networks: A procedural guide.* International Journal of Approximate Reasoning, 15(3):225 – 263, 1996.

[31] Jensen, Finn V. und Frank Jensen: *Optimal Junction Trees.* In: *Proceedings of the Tenth International Conference on Uncertainty in Artificial Intelligence*, UAI'94, Seiten 360–366, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[32] Jeon, H., Y. Xia und V. K. Prasanna: *Parallel Exact Inference on a CPU-GPGPU Heterogenous System.* In: *2010 39th International Conference on Parallel Processing*, Seiten 61–70, Sep. 2010.

[33] Jordan, Michael I., Zoubin Ghahramani, Tommi S. Jaakkola und Lawrence K. Saul: *An Introduction to Variational Methods for Graphical Models.* Machine Learning, 37(2):183–233, Nov 1999.

[34] Kalaiselvi, T., P. Sriramakrishnan und K. Somasundaram: *Survey of using GPU CUDA programming model in medical image analysis.* Informatics in Medicine Unlocked, 9:133 – 144, 2017.

[35] Karp, R.: *Reducibility among combinatorial problems.* In: Miller, R. und J. Thatcher (Herausgeber): *Complexity of Computer Computations*, Seiten 85–103. Plenum Press, 1972.

[36] Kleinberg, Jon und Éva Tardos: *Algorithm Design.* Addison Wesley, 2006.

[37] Koller, Daphne und Nir Friedman: *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning.* The MIT Press, 2009.

[38] Korhonen, Janne und Pekka Parviainen: *Exact Learning of Bounded Tree-width Bayesian Networks.* In: Carvalho, Carlos M. und Pradeep Ravikumar (Herausgeber): *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, Band 31 der Reihe *Proceedings of Machine Learning Research*, Seiten 370–378, Scottsdale, Arizona, USA, 29 Apr–01 May 2013. PMLR.

[39] Kozlov, Alexander V. und Jaswinder Pal Singh: *A Parallel Lauritzen-Spiegelhalter Algorithm for Probabilistic Inference.* In: *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, Supercomputing '94, Seiten 320–329, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[40] Kruskal, Joseph B.: *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem.* Proceedings of the American Mathematical Society, 7(1):48–50, 1956.

[41] Lauritzen, S. L. und D. J. Spiegelhalter: *Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems.* Journal of the Royal Statistical Society. Series B (Methodological), 50(2):157–224, 1988.

[42] Lauritzen, S.L.: *Graphical Models.* Oxford Statistical Science Series. Clarendon Press, 1996.

[43] Liebig, Thomas, Nico Piatkowski, Christian Bockermann und Katharina Morik: *Route Planning with Real-Time Traffic Predictions.* In: *Proceedings of the 16th LWA Workshops: KDML, IR and FGWM*, Seiten 83–94, 2014.

[44] Manning, Christopher D. und Hinrich Schütze: *Foundations of Statistical Natural Language Processing.* MIT Press, Cambridge, MA, USA, 1999.

[45] Mooij, Joris, Bastian Wemmenhove, Bert Kappen und Tommaso Rizzo: *Loop Corrected Belief Propagation.* In: Meila, Marina und Xiaotong Shen (Herausgeber):

Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics, Band 2 der Reihe *Proceedings of Machine Learning Research*, Seiten 331–338, San Juan, Puerto Rico, 21–24 Mar 2007. PMLR.

[46] Murphy, Kevin P.: *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[47] Murphy, Kevin P., Yair Weiss und Michael I. Jordan: *Loopy Belief Propagation for Approximate Inference: An Empirical Study*. In: *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, UAI'99, Seiten 467–475, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[48] Namasivayam, V. K., A. Pathak und V. K. Prasanna: *Scalable Parallel Implementation of Bayesian Network to Junction Tree Conversion for Exact Inference*. In: *2006 18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'06)*, Seiten 167–176, Oct 2006.

[49] Nocedal, Jorge und Stephen J. Wright: *Numerical optimization*. Springer series in operations research and financial engineering. Springer, New York, NY, 2. ed. Auflage, 2006.

[50] NVIDIA Corporation: *NVIDIA CUDA C Programming Guide*, 2010. Version 3.2.

[51] Pacheco, Peter: *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st Auflage, 2011.

[52] Park, James D. und Adnan Darwiche: *Morphing the Hugin and Shenoy–Shafer Architectures*. In: Nielsen, Thomas Dyhre und Nevin Lianwen Zhang (Herausgeber): *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, Seiten 149–160, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[53] Pearl, Judea: *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[54] Sanders, Jason und Edward Kandrot: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st Auflage, 2010.

[55] Satish, Nadathur und Narayanan Satish: *Graphical Model Inference using GPUs*. 2007.

[56] Shafer, Glenn R. und Prakash P. Shenoy: *Probability propagation*. Annals of Mathematics and Artificial Intelligence, 2(1):327–351, Mar 1990.

[57] Sharma, Rita und David Poole: *Efficient Inference in Large Discrete Domains*. CoRR, abs/1212.2518, 2012.

[58] Stone, J. E., D. Gohara und G. Shi: *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems.* Computing in Science Engineering, 12(3):66–73, May 2010.

[59] Wainwright, Martin J. und Michael I. Jordan: *Graphical Models, Exponential Families, and Variational Inference.* Found. Trends Mach. Learn., 1(1-2):1–305, Januar 2008.

[60] Woods, J. W.: *Markov image modeling.* In: *1976 IEEE Conference on Decision and Control including the 15th Symposium on Adaptive Processes*, Seiten 596–600, Dec 1976.

[61] Xia, Y. und V. K. Prasanna: *Node Level Primitives for Parallel Exact Inference.* In: *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)*, Seiten 221–228, Oct 2007.

[62] Xia, Yinglong und Viktor K. Prasanna: *Scalable Node-Level Computation Kernels for Parallel Exact Inference.* IEEE Trans. Comput., 59(1):103–115, Januar 2010.

[63] Yannakakis, Mihalis: *Computing the Minimum Fill-In is NP-Complete.* SIAM Journal on Algebraic and Discrete Methods, 2, 03 1981.

[64] Zheng, Lu und Ole Mengshoel: *Optimizing Parallel Belief Propagation in Junction Treesusing Regression.* In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, Seiten 757–765, New York, NY, USA, 2013. ACM.

[65] Zheng, Lu, Ole Mengshoel und Jike Chong: *Belief Propagation by Message Passing in Junction Trees: Computing Each Message Faster Using GPU Parallelization.* In: *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, UAI'11, Seiten 822–830, Arlington, Virginia, United States, 2011. AUAI Press.

Eidesstattliche Versicherung
(Affidavit)

Heppe, Lukas
Name, Vorname
(Last name, first name)

158024
Matrikelnr.
(Enrollment number)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem folgenden Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present Bachelor's/Master's* thesis with the following title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution.

Titel der [redacted]/Masterarbeit*:
(Title of the Bachelor's/ Master's* thesis):

Parallele, verteilte Implementierung der Junction Tree Inferenz

*Nichtzutreffendes bitte streichen
(Please choose the appropriate)

Dortmund, 23.12.2019
Ort, Datum
(Place, date)

L. Heppe
Unterschrift
(Signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to €50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, section 63, subsection 5 of the North Rhine-Westphalia Higher Education Act (*Hochschulgesetz*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:**

Dortmund, 23.12.2019
Ort, Datum
(Place, date)

L. Heppe
Unterschrift
(Signature)

**Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.