INSTITUT FÜR INFORMATIK

SAT Solving mit GPU Unterstützung

Philipp Sieweck

Bericht Nr. 1207
Juni 2012
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

SAT Solving mit GPU Unterstützung

Philipp Sieweck

Bericht Nr. 1207
Juni 2012
ISSN 2192-6247

e-mail: psi@informatik.uni-kiel.de

Dieser Bericht basiert auf der Diplomarbeit des Verfassers

SAT Solving mit GPU Unterstützung

Philipp Sieweck

Juni 2012

Zusammenfassung

Vorgestellt wird eine Implementierung des Survey Propagation Algorithmus (SP) auf einer GPU mit CUDA, sowie ein SAT Solver auf Basis von MiniSAT, der SP als zusätzliche Heuristik verwendet, um zufällige k-SAT Probleme schneller lösen zu können.

Inhaltsverzeichnis

1 Einleitung

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT) ist das klassische Problem, das Anfang der 1970er von Stephen A. Cook als NP-vollständig bewiesen wurde. Es ist einerseits eine Grundlage für Beweise von weiteren NP-vollständigen Problemen, andererseits wird es auch zum Lösen derselbigen herangezogen. Es ist u. a. zu einem zentralen Bestandteil bei der Verifikation von Software und elektronischen Schaltkreisen geworden. Das funktioniert, weil die Problemstellung in polynomieller Zeit in eine SAT Formel umkodiert werden kann, sodass für diese genau dann eine erfüllende Belegung existiert, wenn es eine Lösung für das ursprüngliche Problem gibt.

Die Suche nach immer schnelleren SAT Solvern führt mehr und mehr zu einer Suche nach einer geeigneten Parallelisierung. In den SAT Competitions der letzten Jahre wurde eigens dafür eine Sparte für paralleles SAT Solven eingerichtet. Jedoch nehmen daran vergleichsweise wenig Programme teil, was damit zu erklären ist, dass sich das SAT Problem mit dem bislang so erfolgreichen DPLL Unterbau nur schwer parallel verarbeiten lässt. Bislang gibt es bei parallelem DPLL zwei Richtungen[7].

Die eine versucht das Problem mit der typischen Parallelisierungsstrategie Divide-and-Conquer zu lösen. Der Suchbaum wird in viele Teile zerkleinert, mit denen die Rechenknoten gleichmäßig ausgelastet werden. Besser zu funktionieren scheint die andere Richtung, die mehr dem Geist eines modernen DPLL Solvers entspricht. Beim Portfolio Methode werden mehrere Solver Instanzen auf das gesamte Problem losgelassen. Sie unterscheiden sich anhand ihrer Startparamter, sodass dadurch unterschiedliche Suchverhalten entsehen. Ab und zu können — je nach Solver — Konfliktklauseln ausgetauscht werden. Sind diese kurz und für die jeweils anderen Instanzen relevant, können durch die Parallelisierung Synergien erziehlt werden. ManySAT[8] baut auf diesem Konzept auf.

Beide Richtungen skalieren jedoch nicht so einfach auf Systeme mit vielen Rechenknoten. Für GPUs ist sogar DPLL egal in welcher Form völlig ungeeignet; allein deshalb, weil DPLL eine für GPUs sehr komplexe Programmstruktur aufweist und sich daher kaum mit dem SIMT Modell vereinbaren lässt.

Neben DPLL gibt es andere Verfahren, die mit stochastischen Mitteln versuchen, eine Lösung zu finden, in dem SAT als ein globales Optimierungsproblem verstanden wird. Diese Verfahren sind inhärent *unvollständig*, d. h. es kann mit ihnen nicht die Unerfüllbarkeit einer Formel nachgewiesen werden. Für manche Klassen von SAT Problemen sind sie jedoch DPLL haushoch überlegen.

Dazu gehören zufällige k-SAT Formeln (Random k-SAT) mit $k \geq 3$. Diese Formeln bestehen aus zufälligen Klauseln mit genau k Literalen und wurden aus einer Normalverteilung generiert. Es stellte sich heraus, dass die Wahrscheinlichkeit, ob

eine solche Formel erfüllbar ist, maßgeblich vom Verhältnis $\alpha = \frac{|\mathcal{C}|}{|\mathcal{V}|}$ zwischen Klauseln und Variablen bestimmt wird. In [3] wird das wie folgt zusammengefasst:

Sei $P_N(\alpha)$ die Wahrscheinlichkeit, dass eine zufällige Formel mit N Variablen und αN Klauseln erfüllbar ist. $P_N(\alpha)$ ist eine monoton fallende Funktion. Gibt es keine Constraints, ist die Formel trivialerweise erfüllt ($P_N(0) = 1$). Je mehr Klauseln hinzu kommen, desto wahrscheinlicher wird es, dass die Formel nicht mehr erfüllbar ist ($\lim_{\alpha\to\infty} P_N(\alpha) = 0$). Irgendwo muss es deswegen eine Stelle α_c mit $P_N(\alpha_c) = \frac{1}{2}$ geben. Man hat ausgerechnet, dass die Stelle etwa bei $\alpha_c = 4,27$ liegt. Formeln mit $\alpha < \alpha_c$ neigen eher zur Erfüllbarkeit, bzw. Formeln mit $\alpha > \alpha_c$ zur Unerfüllbarkeit. Interessanterweise wird der Übergang zwischen den Bereichen mit steigender Variablenzahl immer schärfer. Dieses Phänomen wird *Phasentransition* genannt.

Formeln, die in der Nähe von α_c angesiedelt sind, erweisen sich als besonders schwer zu lösen. Während DPLL schnelle Ergebnisse bei strukturellen Formeln mit Millionen von Variablen liefern kann, brauchen dieselben Programme bei zufälligen 3-SAT Formeln mit 500 Variablen nahe an α_c weit mehr als 10 Stunden.

In dieser Arbeit wird der DPLL Solver MiniSAT[5] mit einer Heuristik erweitert, die es erlaubt, auch mit zufälligen SAT Formeln gut zurecht zu kommen. Diese Heuristik besteht im Kern aus dem Survey Propagation (SP) Algorithmus[3], der zur Zeit mit Abstand schnellste Algorithmus für diese Art von Formeln ist. Da SP in hohem Maße parallel ausgeführt werden kann, wird die Heuristik auf der GPU berechnet.

Danksagungen

Diese Arbeit ist meiner Familie, insbesondere meinen Eltern, gewidmet. Dank ihnen ist mir die Arbeit viel leicher von der Hand gegangen. Ich danke auch meinem Betreuer Dirk Nowotka für seine ermutigenden Worte in Zeiten, in denen es nicht vorwärts ging.

2 Grundlagen

2.1 SAT

Gegeben ist eine aussagenlogische Formel ϕ mit n boolschen Variablen. Sie ist in konjunktiver Normalform (CNF), d. h. sie besteht aus einer Konjunktion ($\wedge$) von Klauseln, wobei jede Klausel eine Disjunktion ($\vee$) von Variablen und ihren Negationen ist.

Ein Beispiel mit den Variablen $\mathcal{V} = \{x_1, x_2, x_3, x_4\}$ kann wie folgt aussehen:

$$\phi = \underbrace{(x_1 \vee \overline{x_2} \vee x_3)}_{C_1} \wedge \underbrace{(\overline{x_2} \vee x_3)}_{C_2} \wedge \underbrace{(x_1 \vee \overline{x_3} \vee \overline{x_4})}_{C_3} \wedge \underbrace{(\overline{x_2}, x_4)}_{C_4}$$

Gesucht wird für die Variablen $\mathcal{V}$ eine Belegung $\mathcal{V} \to \{\mathrm{T}, \mathrm{F}\}$, die die gesamte Formel wahr macht, z. B. $\{x_1 \mapsto \mathrm{T},\ x_2 \mapsto \mathrm{F},\ x_3 \mapsto \mathrm{F},\ x_4 \mapsto \mathrm{T}\}$. Ein naiver Algorithmus könnte das Problem mit einem Suchbaum oder einer Wahrheitstabelle angehen, wofür er durchschnittlich $\mathcal{O}(2^{|\mathcal{V}|})$ Schritte bräuchte. Jedoch spielt diese Komplexität in der Praxis kaum eine Rolle, weil die dortigen SAT Formeln in den meisten Fällen eine gewisse Struktur inne haben, anhand der große Teile des Suchbaums abgeschnitten werden können.

2.2 Notation und formale Definition

Sei $\mathcal{V}$ eine Menge von boolschen Variablen. Jede Variable $v \in \mathcal{V}$ und ihre Negation $\overline{v}$ bilden je ein *Literal*, sodass die Menge aller Literale $\mathcal{L} = \bigcup_{v \in \mathcal{V}} \{v, \overline{v}\}$ ist. Die Negation eines Literals $l \in \mathcal{L}$ wird mit $\neg l$ bezeichnet und bedeutet $\overline{v}$, falls $l = v$ ist, bzw. v, falls $l = \overline{v}$ ist. Eine *Klausel* $C = l_1 \vee \ldots \vee l_n$ ist eine Menge von Literalen, die durch eine Disjunktion verkettet sind. $\widetilde{C} = \{\neg l \mid l \in C\}$ bezeichnet eine Klausel, die alle Literale aus C negiert enthält. Eine *SAT Formel* ϕ in konjunktiver Normalform (*CNF*) ist eine Menge von Klauseln, die durch eine Konjunktion verkettet sind ($\phi = C_1 \wedge \ldots \wedge C_m$).

Weiterhin bezeichnet $\mathcal{C}$ die Menge aller Klauseln in ϕ. $\mathcal{C}^+(v)$ ist die Menge der Klauseln, in denen die Variable v positiv vorkommt. $\mathcal{C}^-(v)$ ist die Menge der Klauseln, die das Literal $\overline{v}$ enthalten. Schließlich ist $\mathcal{C}(v) = \mathcal{C}^+(v) \cup \mathcal{C}^-(v)$ die Klauselmenge, in der v egal in welcher Polarität vorkommt. Analog dazu sind in $\mathcal{C}(l)$ alle Klauseln, in denen das Literal l vorkommt.

Eine *partielle Belegung* M für eine Formel ϕ ist eine Abbildung $\mathcal{V} \to \{\mathrm{T}, \mathrm{F}, ?\}$. Der Wert ? besagt, dass für die jeweilige Variable noch keine Entscheidung getroffen

wurde. Die Abbildung wird zwecks kürzerer Schreibweise für Literale erweitert. Für ein Literal $l = v$ gilt $M(l) = M(v)$. Im Falle von $l = \overline{v}$ ist $M(l) = \neg M(v)$ mit $\neg = \{\mathrm{T} \mapsto \mathrm{F}, \mathrm{F} \mapsto \mathrm{T}, ? \mapsto ?\}$. Eine Belegung ist *vollständig*, wenn jeder Variablen ein Wert zugewiesen wurde, sodass es keine Variable mit $M(v) = ?$ gibt. Alternativ ist M als eine Menge von Literalen mit der Einschränkung $\nexists l \in \mathcal{L} : \{l, \neg l\} \subseteq M$ definiert, d. h. es dürfen nicht gleichzeitig ein Literal und seine Negation enthalten sein. Für ein $l \in \mathcal{L}$ gilt $M(l) = \mathrm{T} \Leftrightarrow l \in M$, $M(l) = \mathrm{F} \Leftrightarrow \neg l \in M$ und ansonsten $M(l) = ?$.

Eine Belegung M *erfüllt* ein Literal l, wenn $M(l) = \mathrm{T}$. l ist *unerfüllt*, wenn $M(l) = \mathrm{F}$ und im letzten Fall $M(l) = ?$ ist l *unentschieden*.

Eine Klausel C hat unter der Belegung M einen der folgenden Zustände:

- Sie ist *erfüllt* ($M \models C$), wenn mindestens ein Literal erfüllt ist ($\exists l \in C : M(l) = \mathrm{T}$ bzw. $M \cap C \neq \emptyset$).
- Sie *steht in Konflikt* ($M \models \neg C$), wenn jedes Literal den Wert F hat ($\forall l \in C : \neg l \in M$).
- Sie ist *unit*, wenn sie nicht erfüllt ist und genau eines ihrer Literale unentschieden ist ($\exists l \in C : M(l) = ? \wedge \forall l' \in C \setminus \{l\} : M(l') = \mathrm{F}$).
- Ansonsten ist sie *unentschieden*.

Eine Formel ϕ wird durch M *erfüllt*, wenn jede ihrer Klauseln durch M erfüllt ist. M ist dann ein *Modell* für ϕ (geschrieben $M \models \phi$). Falls kein Modell für ϕ existiert, ist ϕ *unerfüllbar*.

Beispiel. Aus der Formel ϕ von Abschnitt 2.1 und einer Belegung $M = \{x_2, \overline{x_4}\}$ folgt, dass C_1 unentschieden, C_2 eine Unit-Klausel und C_3 erfüllt ist. C_4 steht in Konflikt. Damit ist M und jede Erweiterung von M kein Modell für ϕ ($M \not\models \phi$).

3 DPLL

Das Grundgerüst, auf dem die aktuell erfolgreichsten SAT Solver[22] aufbauen, wurde größtenteils in den 1960er Jahren veröffentlicht. Die Urform des Davis-Putnam-Logemann-Loveland (DPLL) Verfahrens sucht nach einer erfüllenden Belegung, in dem es den Suchbaum in Verbindung mit Backtracking abarbeitet. Der Unterschied zum naiven Vorgehen liegt darin, dass bei jeder Entscheidung sofort alle trivialen Implikationen propagiert werden, die den Zustand der unentschiedenen Variablen und Klauseln betreffen.

Die Abfolge der internen Zustände in einem DPLL-Algorithmus lässt sich durch ein *Transistionssystem* $(\mathscr{S}, \Longrightarrow)$ mit dem Zustandsraum $\mathscr{S}$ und einer Relation $\Longrightarrow \subset \mathscr{S} \times \mathscr{S}$ modellieren [19]. Wir sagen, dass eine Transition von S nach S' genau dann existiert, wenn $(S, S') \in \Longrightarrow$. Eine Transitionskette $S_0 \Longrightarrow S_1$, $S_1 \Longrightarrow S_2$, $S_2 \Longrightarrow S_3$, ... kürzen wir mit $S_0 \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \ldots$ ab.

Jeder Zustand aus $\mathscr{S}$ hat die Form $M \mid\mid \phi$, bestehend aus einer partiellen Belegung M und einer Formel ϕ, bzw. einer Menge von Klauseln. Im Zusammenhang mit dem DPLL Transistionssystem erweitern wir die Definition einer partiellen Belegung um folgende Aspekte:

1. Eine partielle Belegung ist eine geordnete Menge.
2. Jedes Literal $l \in M$ kann mit einer Markierung versehen werden (geschrieben l^d), die anzeigt, ob es sich dabei um ein sogenanntes Decision-Literal handelt (mehr dazu später).

Beispiel. Um uns ein wenig mit der Schreibweise vertraut zu machen, sei die Variablenmenge $\mathcal{V} = \{1, 2, 3, 4, 5\}$ und $\phi = (\overline{1} \vee 2) \wedge (\overline{1} \vee \overline{3} \vee 5) \wedge (\overline{2} \vee 4)$ gegeben. Beim Start des DPLL Solvers wurde noch keiner Variablen ein Wert zugewiesen; daher ist der Startzustand $\emptyset \mid\mid \phi$. Der DPLL Prozess könnte später auf den Zustand $1^d 2 4 5^d \overline{3} \mid\mid \phi$ treffen, der sogar ein Endzustand ist, weil ϕ durch die Belegung erfüllt wird.

Neben den normalen Zuständen in $\mathscr{S}$ gibt es noch den speziellen Zustand FAIL, der genau dann erreicht wird, wenn die Eingabeformel ϕ unerfüllbar ist.

Die Ausführung eines DPLL Solvers entspricht folgender Definition:

Definition 3.1 (DPLL Durchlauf)**.** Gegeben sei eine SAT Formel ϕ. Ein *DPLL Durchlauf* ist eine Folge von Zuständen $(S_0, S_1, \ldots, S_n)$, sodass $S_0 \Longrightarrow S_1 \Longrightarrow \ldots \Longrightarrow S_n$ gilt. Der Startzustand S_0 ist gleich $\emptyset \mid\mid \phi$. Der Endzustand S_n muss entweder FAIL sein oder die Form $M \mid\mid \phi'$, haben, wobei M eine erfüllende Belegung für ϕ ist.

```
function DPLL(φ) : {SAT, UNSAT}
    if !BCP() then return UNSAT;
    loop
        if !DECIDE() then return SAT;
        else
            while (!BCP()) do
                backtrackLevel := ANALYZE−CONFLICT();
                if backtrackLevel < 0 then return UNSAT;
                else BACKTRACK(backtrackLevel);
```

Abbildung 3.1: High-Level Sicht des DPLL Algorithmus [12, 17]

Bis jetzt haben wir noch nichts über den Aufbau von $\Longrightarrow$ ausgesagt, geschweige denn, welche Abzweigung im Falle von Alternativen gewählt werden soll. Im weiteren Verlauf dieses Kapitels werden wir nach und nach einige *Transistionsregeln* definieren, die die Relation $\Longrightarrow$ mit Inhalt füllen, sodass am Ende folgender Satz gilt:

Satz 3.1 (Korrektheit von DPLL-basierten SAT-Solvern)**.** *Ein SAT Solver, der sein Ergebnis nur mit Hilfe der erlaubten Regeln des Transitionssystems von der Eingabe abgeleitet hat (Definition 3.1), ist korrekt.*

Für den Beweis sei auf [19] verwiesen. Das Transitionsystem soll uns daher helfen, den Spielraum bei der Implementation eines DPLL-basierten SAT Solvers zu verstehen.

DPLL besteht aus folgenden Komponenten, die im Laufe dieses Kapitels erläutert werden: DECIDE, BOOLEAN CONSTRAINT PROPAGATION (BCP), ANALZYE CONFLICT und BACKTRACK. Abbildung 3.1 zeigt zur Verdeutlichung des Zusammenspiels der Komponenten eine High-Level Sicht des DPLL Schemas.

DECIDE weist einer zuvor unentschiedenen Variable den Wert T oder F zu. Welche Variable der Solver auswählt und wie sie polarisiert wird, hängt von einer Heuristik ab, auf die wir später näher eingehen. Jedoch sei bereits an dieser Stelle darauf hingewiesen, dass die Heuristik den größten Einfluss auf die Performanz eines Solvers hat und die Suche nach einer Optimierung dieser Komponente das zentrale Thema dieser Arbeit ist.

Die Transitionsregel, auf der DECIDE beruht, lautet:

Definition 3.2 (Decide Transition)**.** Sei $l \in \mathcal{L}$ ein unentschiedenes Literal in M. Dann gilt

$$M \mid\mid \phi \Longrightarrow Ml^d \mid\mid \phi.$$

Mit anderen Worten: DECIDE sucht sich eine Variable v aus und gibt ihr einen Wert, sodass eines der beiden Literale v oder $\overline{v}$ wahr wird. Dieses Literal l wird an die Liste der erfüllten Literale M angehängt und mit einer Markierung versehen, die

besagt, dass das Literal durch eine Decide Transition wahr wurde; wir bezeichnen es daher als *Decision-Literal*.

Jedes erfüllte Literal gehört einem *Decision-Level (DL)* an, das gleich der Anzahl der vorangegangenen Decision-Literale (inklusive dem Literal selbst) ist. Mit $l@dl$ meinen wir ein Literal l, welches im Decision-Level dl erfüllt wurde. Beispielsweise erfüllt die Belegung $M = 123^d\overline{48}^d9$ u.a. die Literale 1@0, 3@1, $\overline{4}$@1 und 9@2.

Die Markierung eines Literals zu einem Decision-Literal ist eine Notiz für den Solver, dass er an dieser Stelle in der Belegungsliste eine Fallunterscheidung vorgenommen hat. Sollte er später herausfinden, dass die Formel nach dieser Entscheidung unerfüllbar ist, kann er an den Entscheidungspunkt zurückkehren und das Literal l^d und alle Nachfolger in der Belegungsliste auf unentschieden zurücksetzen und stattdessen mit $\neg l$ weiterarbeiten.

Dieses Zurücknehmen einer schlechten Entscheidung ist unter dem Begriff *Backtracking* bekannt und wird in unserem Schema von der Komponente Backtrack implementiert. Transitionsregeln, die diese Technik beschreiben, definieren wir weiter unten.

Decide kommt immer dann zum Einsatz, wenn der Solver sich für eine Abzweigung im Suchbaum entscheiden muss, weil keine unmittelbaren Folgerungen anhand der aktuellen Variablenbelegung gezogen werden können (siehe Definition 3.3). Falls eine Variable belegt werden konnte, gibt Decide an die Hauptschleife (Abbildung 3.1) True zurück.

Wenn bereits alle Variablen eine Belegung haben, lautet der Rückgabewert False. In diesem Fall hat der Solver einen Zustand $M \mid\mid \phi$ erreicht, in dem M eine vollständige Belegung für ϕ ist. Zudem ist M eine erfüllende Belegung für ϕ, weil Decide nur aufgerufen wird, wenn durch M keine Klausel in ϕ in Konflikt steht. Daher terminiert der Algorithmus mit Sat.

Kommt es zu einer Belegung, unter der eine Klausel C unit wird, muss das verbliebene unentschiedene Literal l (*Unit-Literal* gennant) erfüllt werden. Wir sagen dazu, dass l von C impliziert wird. Im Transitionssystem ist das durch folgende Definition realisiert:

Definition 3.3 (Unit-Propagate Transition)**.** Sei l ein unentschiedenes Literal in M und sei weiterhin $M \models \neg C$, dann gilt

$$M \mid\mid \phi \wedge (C \vee l) \Longrightarrow Ml \mid\mid \phi \wedge (C \vee l).$$

Die Komponente Boolean-Constraint-Propagation (BCP) macht nichts anderes, als solange Unit-Literale zu propagieren, bis es keine Unit-Klauseln mehr gibt oder ein Konflikt auftritt.

Ein *Konflikt* ist genau dann entstanden, wenn nach der Anwendung einer Unit-Propagate Transition ein Zustand mit einer widersprüchlichen Belegung erreicht wurde, also einer Belegung in der ein Literal und seine Negation vorkommt.

BCP liefert an die Hauptschleife True zurück, wenn alle Unit-Klauseln erfolgreich propagiert werden konnten, worauf Decide die nächste Entscheidung für eine Variable treffen kann. Ansonsten setzt False die Konflikt-Behandlung in Gang.

Wie wird mit einem Konflikt umgegangen? Analyse-Conflict sucht nach der Ursache des Konflikts und versucht ihn dadurch zu beheben, indem an einer anderen Stelle im Suchbaum weitergemacht wird.

Im ursprünglichen DPLL wurde ein Konflikt mit einem simplen Backtrack Mechanismus aufgelöst. Dieser setzt das aktuelle Decision-Literal l samt allen darauf folgenden Implikationen zurück und hängt $\neg l$ als logische Folgerung an das ehemals darunter liegende Decision-Level. Für unser Transitionssystem ergibt das folgende Regel, in der C die Klausel ist, die den Konflikt auslöst:

Definition 3.4 (Einfache Backtrack Transition)**.** Gegeben sei ein Zustand der Form $Ml^dN \mid\mid \phi \wedge C$, sodass $Ml^dN \models \neg C$ gilt und N keine Decision-Literale enthält. Dann existiert die Transition

$$Ml^dN \mid\mid \phi \wedge C \Longrightarrow M\neg l \mid\mid \phi \wedge C.$$

Man beachte, dass es nach dieser Transition ein Decision-Literal weniger gibt. Da sich eine frühere Entscheidung l als falsch erwiesen hat, ist mit $\neg l$ die Fallunterscheidung abgeschlossen.

Nach dem Backtracking wird BCP erneut gestartet. Kommt es dabei abermals zu einem Konflikt, nimmt der Backtrack-Schritt ein weiteres Decision-Level zurück. Das Ganze wird solange wiederholt, bis der Solver entweder einen Zustand erreicht hat, in dem BCP konfliktfrei durchläuft, oder es keine Decision-Literale mehr gibt, die rückgängig gemacht werden können.

Letzteres liefert dann den Beweis für die Unerfüllbarkeit der Formel, denn es ist durch BCP ein Konflikt aufgetreten, ohne dass der Solver eine Entscheidung für eine Variable gefällt hat. Im Transistionssystem sieht dieser Fall wie folgt aus:

Definition 3.5 (Fail Transition)**.** Sei $M \mid\mid \phi \wedge C$ ein Zustand, bei dem M keine Decision-Literale enthält und $M \models \neg C$ gilt. Dann existiert die Transition

$$M \mid\mid \phi \wedge C \Longrightarrow \textsc{Fail}.$$

Die vier Transitionsregeln Decide Transition, Unit-Propagate Transition, Einfache Backtrack Transition und Fail Transition reichen bereits aus, um jede SAT Formel korrekt lösen zu können. Darum ist das eine günstige Gelegenheit für ein DPLL Durchlauf Beispiel (entnommen aus [19]).

Beispiel.

$$
\begin{array}{rcll}
\emptyset & \| & (\overline{1} \vee \overline{2}) \wedge (2 \vee 3) \wedge (\overline{1} \vee \overline{3} \vee 4) \wedge (2 \vee \overline{3} \vee \overline{4}) \wedge (1 \vee 4) & \Longrightarrow \quad \text{(Decide)} \\
1^d & \| & (\overline{1} \vee \overline{2}) \wedge (2 \vee 3) \wedge (\overline{1} \vee \overline{3} \vee 4) \wedge (2 \vee \overline{3} \vee \overline{4}) \wedge (1 \vee 4) & \Longrightarrow \quad \text{(Unit-Propagate)} \\
1^d\overline{2} & \| & (\overline{1} \vee \overline{2}) \wedge (2 \vee 3) \wedge (\overline{1} \vee \overline{3} \vee 4) \wedge (2 \vee \overline{3} \vee \overline{4}) \wedge (1 \vee 4) & \Longrightarrow \quad \text{(Unit-Propagate)} \\
1^d\overline{2}3 & \| & (\overline{1} \vee \overline{2}) \wedge (2 \vee 3) \wedge (\overline{1} \vee \overline{3} \vee 4) \wedge (2 \vee \overline{3} \vee \overline{4}) \wedge (1 \vee 4) & \Longrightarrow \quad \text{(Unit-Propagate)} \\
1^d\overline{2}34 & \| & (\overline{1} \vee \overline{2}) \wedge (2 \vee 3) \wedge (\overline{1} \vee \overline{3} \vee 4) \wedge (2 \vee \overline{3} \vee \overline{4}) \wedge (1 \vee 4) & \Longrightarrow \quad \text{(Einfach Backtrack)} \\
\overline{1} & \| & (\overline{1} \vee \overline{2}) \wedge (2 \vee 3) \wedge (\overline{1} \vee \overline{3} \vee 4) \wedge (2 \vee \overline{3} \vee \overline{4}) \wedge (1 \vee 4) & \Longrightarrow \quad \text{(Unit-Propagate)} \\
\overline{1}4 & \| & (\overline{1} \vee \overline{2}) \wedge (2 \vee 3) \wedge (\overline{1} \vee \overline{3} \vee 4) \wedge (2 \vee \overline{3} \vee \overline{4}) \wedge (1 \vee 4) & \Longrightarrow \quad \text{(Decide)} \\
\overline{1}4\overline{3}^d & \| & (\overline{1} \vee \overline{2}) \wedge (2 \vee 3) \wedge (\overline{1} \vee \overline{3} \vee 4) \wedge (2 \vee \overline{3} \vee \overline{4}) \wedge (1 \vee 4) & \Longrightarrow \quad \text{(Unit-Propagate)} \\
\overline{1}4\overline{3}^d2 & \| & (\overline{1} \vee \overline{2}) \wedge (2 \vee 3) \wedge (\overline{1} \vee \overline{3} \vee 4) \wedge (2 \vee \overline{3} \vee \overline{4}) \wedge (1 \vee 4) &
\end{array}
$$

Mit dieser einfachen Variante des Backtrackings kommt man beim Lösen von Problemen mit vielen Variablen nur sehr langsam voran. Sie ist zu einer Zeit entstanden, in der das Lösen von SAT Formeln kaum praxisrelevant war. Bei zufällig generierten Formeln ist es aufgrund ihrer mangelnden Struktur schwer, einen besseren Mechanismus zu finden. Anders sieht es bei Formeln aus, die aus einer Übersetzung eines praxisnahen Problems entstanden sind. Diese sind in der Regel hoch strukturiert, sodass der Solver Erfahrung sammeln kann, die ihm auch in anderen Teilen des Suchbaums zu Gute kommt.

Diese „Erfahrung" spiegelt sich dabei in einer Menge von Klauseln wieder, die logisch von der Eingabeformel abgeleitet sind, aber nicht in ihr vorkommen. Diese Klauseln bezeichnen wir als *Konfliktklauseln*[1], weil jede von ihnen das Resultat einer Konfliktanalyse ist und jeweils die *Ursache für das Zustandekommen eines Konflikts* kodieren.

In der Literatur wird die Generierung von Klauseln als Reaktion auf einen Konflikt *conflict-driven learning* genannt; ein Meilenstein, der das Lösen von großen SAT Formeln erst praxistauglich machte. Um zu verstehen, wie Konfliktklauseln entstehen und welche Wirkung sie haben, machen wir uns zunächst mit Implikationsgraphen vertraut.

Definition 3.6 (Implikationsgraph)**.** Ein *Implikationsgraph* ist ein azyklischer, gerichteter Graph $G = (M, E)$. Die Knotenmenge M entspricht der aktuellen Menge der erfüllten Literale. Sie sind annotiert mit dem Decision-Level, in dem sie erfüllt wurden.

Eine Kante aus $E = \{(l_i, l_j) \mid \neg l_i \in \mathit{antecedent}(l_j)\}$ steht für den Grund für die Erfüllung eines Literals. Decision-Literale haben keine eingehenden Kanten. Die sonstigen Literale sind allesamt durch Unit-Propagierung entstanden und haben mindestens eine eingehende Kante.

$\mathit{antecedent}(l)$ bezeichnet die Klausel, in der das Literal l unit wurde. Falls es mehrere Klauseln gibt, die l implizieren, dann ist die Klausel gemeint, die die Propagierung zuerst ausgelöst hat.

[1] Nicht zu verwechseln mit Klauseln, die in Konflikt stehen.

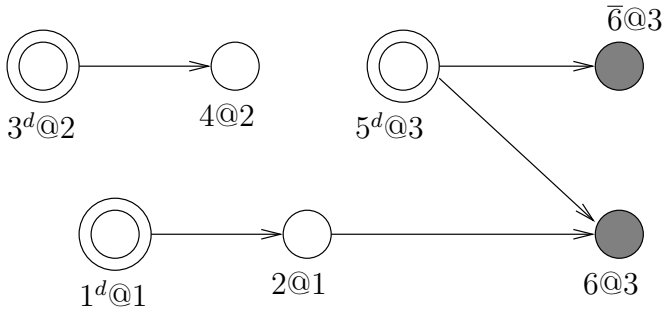

Abbildung 3.2: Ein Konfliktgraph, der bei einem Zustand $1^d23^d45^d6\overline{6} \mid\mid \phi$ entsteht, wobei $\phi = (\overline{1} \vee 2) \wedge (\overline{3} \vee 4) \wedge (\overline{5} \vee \overline{6}) \wedge (6 \vee \overline{5} \vee \overline{2})$. Die Knoten 3^d@2 und 4@2 sind laut Definition nicht Teil des Konfliktgraphen und wurden rein zur Veranschaulichung ihres fehlenden Einflusses auf den Konflikt abgebildet.

Definition 3.7 (Partieller Implikationsgraph)**.** Ein *partieller Implikationsgraph* ist ein Ausschnitt aus dem Implikationsgraph, in dem nur die Literale aus dem aktuellen Decision-Level sowie ihre direkten Vorgängerknoten enthalten sind.

Definition 3.8 (Konfliktgraph)**.** Ein partieller Implikationsgraph ist ein *Konfliktgraph*, genau dann wenn es eine Variable v gibt, für die beide Literalknoten v und $\overline{v}$ im Graph auftauchen. Diese Knoten heißen *Konfliktknoten*.

Als nächstes schauen wir uns die Generierung einer Konfliktklausel anhand eines kleinen Konfliktgraphen in Abbildung 3.2 an. Der Konflikt ist entstanden, weil nach der letzten Entscheidung 5 zwei Klauseln unit wurden, die zusammen 6 und $\overline{6}$ propagieren.

Anhand der Wurzelknoten des Graphen, die mit dem Konflikt in Verbindung stehen, erhalten wir die Konfliktklausel $(\overline{5} \vee \overline{1})$; ein Lemma für den Solver, das besagt, dass die Entscheidung 5 nicht mit der Entscheidung 1 vereinbar ist. Diese Klausel ist aus der Formel logisch ableitbar und ändert daher nichts am Ergebnis. Wird sie in die Klauseldatenbank des Solvers aufgenommen, beschneidet sie den Suchraum, weil der Solver durch die Aufnahme aus der fehlerhaften Entscheidungskombination gelernt hat.

Um das Lernen einer abgeleiteten Klausel im Transitionssystem zu erlauben, definieren wir folgende Regel:

Definition 3.9 (Learn Transition)**.** Sei $C \subset \mathcal{L}$ eine Klausel, die von ϕ abgeleitet ist ($\phi \models C$). Dann existiert die Transition

$$M \mid\mid \phi \Longrightarrow M \mid\mid \phi \wedge C.$$

Nicht jede gefundene Konfliktklausel ist es wert, gelernt zu werden[1]. Lange Klauseln mit bspw. mehr als 20 Literalen verkörpern ein zu spezielles Wissen, wodurch sie

mehr belastend als nutzbringend sind. Ein Solver darf sich deswegen auch Klauseln entledigen, solange dabei die Lösung nicht beeinträchtigt wird.

Definition 3.10 (Forget Transition)**.** Sei C ein Klausel mit $\phi \models C$. Dann existiert die Transition

$$M \mid\mid \phi \wedge C \Longrightarrow M \mid\mid \phi.$$

An der Konfliktklausel $(\overline{5} \vee \overline{1})$ fällt auf, dass die Literale 3 und 4 aus Decision-Level 2 überhaupt nicht am Konflikt beteiligt sind. Einfaches Backtracking ignoriert diesen Umstand und löst den Konflikt mit $1^d23^d45^d6\overline{6} \mid\mid \phi \Longrightarrow 1^d23^d4\overline{5} \mid\mid \phi$ auf; es wird nur das letzte Decision-Literal 5 rückgängig gemacht.

Wäre die Klausel $(\overline{5}\vee\overline{1})$ bereits zu Anfang bekannt gewesen, hätte der Solver nach der Entscheidung 1@1 sofort $\overline{5}$@1 propagiert und wäre im Zustand $1^d2\overline{5} \mid\mid \phi\wedge(\overline{5}\vee\overline{1})$ gelandet.

Und das ist genau der Effekt, um den es beim *Konflikt-basierten Backtracking*, auch *Backjumping* genannt, geht: Der Solver springt immer zum *zweithöchsten Decision-Level* zurück, das in der Konfliktklausel vorkommt. (Besteht die Klausel nur aus einem Literal, dann geht es bei Decision-Level 0 weiter.) Dadurch werden Decision-Levels übersprungen, die am Konflikt gar nicht beteiligt waren (hier 3@2 und 4@2). Generell sind große Sprünge nichts Ungewöhnliches.

Mit dem Backjump zum zweithöchsten Decision-Level wird erreicht, dass nur das Literal der Konfliktklausel mit dem höchsten Decision-Level zurückgesetzt wird und alle weiteren Literale weiterhin unerfüllt bleiben. Daher wird *die Konfliktklausel nach dem Backjumping sofort unit*, sodass die Negation des letzten Konfliktliterals propagiert wird. Voraussetzung ist allerdings, dass eine Konfliktklausel gefunden wurde, in der genau ein Literal aus dem aktuellen Decision-Level vorkommt, weil sie ansonsten nach dem Backtracking nicht unit werden würde. Daher sind wir beim Ableiten neuer Klauseln nur an solchen interessiert, die dieses Kriterium erfüllen.

Wir definieren jetzt die Backjump Regel für das Transitionssystem. Mit C bezeichnen wir die Klausel, die vor der Transition in Konflikt steht. $C' \vee l'$ ist die gefundene Konfliktklausel, bei der l' das Literal ist, welches nach dem Backjump unit wird.

Definition 3.11 (Backjump Transition)**.** Sei C eine Klausel mit $Ml^dN \models \neg C$. Sei weiterhin $C' \vee l' \subset \mathcal{L}$ eine Klausel mit $\{l', \neg l'\} \cap M = \emptyset$, sodass $\phi \wedge C \models C' \vee l'$ und $M \models \neg C'$. Dann existiert die Transition

$$Ml^dN \mid\mid \phi \wedge C \Longrightarrow Ml' \mid\mid \phi \wedge C.$$

Eine Konfliktklausel kann immer erzeugt werden, indem der Solver die Negation der Literale der Wurzelknoten des Konfliktgraphen nimmt. Statt $(\overline{5} \vee \overline{1})$ wäre aber auch $(\overline{5}\vee\overline{2})$ denkbar gewesen. Diese Alternative sieht intuitiv auch allgemeingültiger aus, weil sich 2 näher am Konflikt befindet und auch unter anderen Umständen hätte wahr werden können.

Abbildung 3.3 soll demonstrieren, wie diese „universelleren" Klauseln mit Hilfe eines Konfliktgraphen gefunden werden. Gesucht wird ein Schnitt, der den Graphen

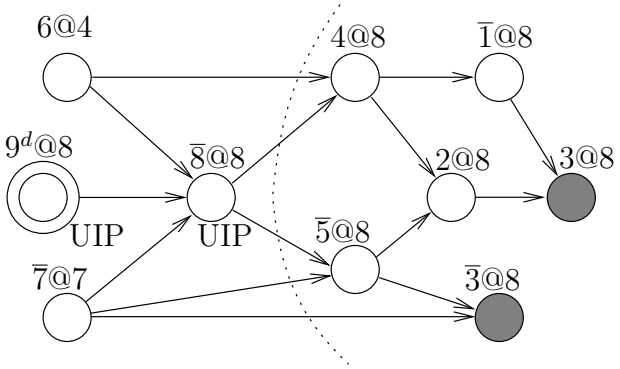

Abbildung 3.3: Konfliktgraph mit zwei UIPs.

in eine linke und eine rechte Seite teilt. Die linke Seite ist die *Ursachenseite*. Sie muss mindestens alle Knoten ohne eingehende Kanten enthalten. Die rechte Seite ist die *Konfliktseite* und muss mindestens alle Konfliktliterale enthalten.

Es ist leicht, sich davon zu überzeugen, dass der Konflikt eine Konsequenz aus genau den Literalen ist, deren ausgehende Kanten geschnitten wurden. Der Schnitt in Abbildung 3.3 bildet darum die Konfliktklausel $(\overline{6} \vee 8 \vee 7)$.

Zu beachten ist, dass der Schnitt so gewählt sein muss, dass von genau einem Literal aus dem aktuellen Decision-Level die Kanten geschnitten werden, denn wir brauchen Klauseln, die nach dem Backtracking sofort unit werden (siehe oben).

In der Literatur wird dieses eine Literal der *Unique Implication Point (UIP)* genannt. Dort ist der UIP als der Knoten definiert, der im Konfliktgraph auf allen Pfaden vom aktuellen Decision-Literal bis zu den Konfliktknoten liegt. Der UIP kann daher als der *Auslöser des Konflikts* gesehen werden. Mindestens ein UIP existiert immer, weil das aktuelle Decision-Literal trivialerweise auf allen Pfaden liegt.

Es sei an dieser Stelle darauf hingewiesen, dass die hier beschriebene Art und Weise, wie Konfliktklauseln gefunden werden, keinesfalls die Einzige ist. Doch seitdem 2001 die bis dato bekanntesten Verfahren in einem Experiment verglichen wurden[30], ist der First UIP Ansatz am gängisten. Der Erfolg dieser Methode lässt sich darauf zurückführen, weil sie durch Nützlichkeit der Klausel in Verbindung mit dem Aufwand ihrer Herleitung besticht. Nichtsdestotrotz wird auch hier aktiv nach Verbesserungen gesucht [29].

Weil es mehrere UIPs geben kann, bedarf es einer weiteren Definition: Der *First UIP* ist der UIP, der den geringsten Abstand zum Konfliktknoten hat. In Abbildung 3.3 ist $\overline{8}$@8 der First UIP. Von allen UIPs bildet der First UIP die kleinste Klausel, die daher gegenüber den Alternativen am *meisten Information* enthält.

Diese Tatsache lässt sich leicht an Abbildung 3.4 nachvollziehen. Seien l_1, l_2 und l_3 die Literale des ersten, zweiten und dritten UIPs (von rechts nach links). Die daraus resultierenden Konfliktklauseln lauten jeweils $C_1 = \neg l_1 \vee \overline{1} \vee 2$, $C_2 = \neg l_2 \vee \overline{5} \vee \overline{1} \vee 2$ und $C_3 = \neg l_3 \vee 4 \vee \overline{5} \vee \overline{1} \vee 2$.

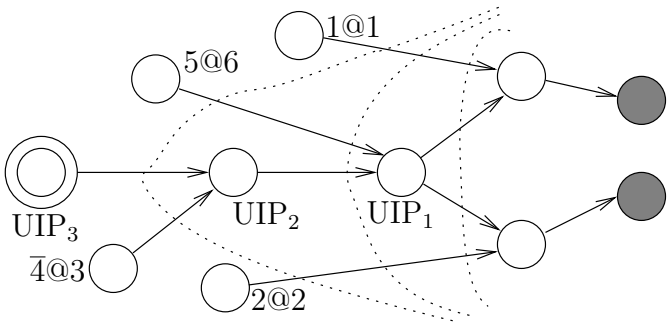

Abbildung 3.4: Ein Konfliktgraph mit drei UIPs

Definition 3.12 (Binäre Resolution mittels Implikationsgraph)**.** Sei C eine Klausel aus ϕ und $l \in C$ eines ihrer Literale, deren Negation $\neg l$ mindestens eine eingehende Kante im Implikationsgraphen hat. Dann gilt $\phi \models C'$ mit

$$C' = (C \cup \mathit{antecedent}(\neg l)) \setminus \{l\}.$$

Wir drücken diese Ableitung mit $C \overset{l}{\rightsquigarrow} C'$ aus.

Das heißt, wir können aus einer bekannten Klausel eine neue Klausel erzeugen, in dem wir eines ihrer Literale l durch die Negationen der Vorgänger von $\neg l$ aus dem Implikationsgraphen ersetzen.

Mit dieser Ableitungsregel ergibt sich die Beziehung $C_1 \overset{\neg l_1}{\rightsquigarrow} C_2 \overset{\neg l_2}{\rightsquigarrow} C_3$. Damit steht fest, dass C_1 als Konfliktklausel am meisten leistet, weil Konflikte, die durch C_2 und C_3 erkannt werden, nur eine Teilmenge von denen sind, die C_1 abdeckt.

Wenden wir uns wieder dem Konflikt aus Abbildung 3.3 zu. Die Teilung des Konfliktgraphen in eine Ursache- und eine Konflikthälfte zeigt deutlich, wie sich eine Konfliktklausel ableiten lässt, die den entstandenen Konflikt mit der letzten Fehlentscheidung verbindet. Implementiert wird dieser Schnitt effizienter durch eine Ableitungskette (Definition 3.12), die folgendermaßen zustande kommt:

Der Solver beginnt mit der Klausel, die in Konflikt steht. Auf die Klausel wird die Ableitungsregel wiederholt in der umgekehrten Reihenfolge angewendet, wie die Literale durch BCP erfüllt wurden. Die Kette endet, sobald die Ableitung genau ein Literal aus dem aktuellen Decision-Level enthält.

Auf diese Weise muss der Solver nicht extra Datenstrukturen für einen Implikationsgraphen verwalten, sondern muss sich für jedes implizierte Literal l nur $\mathit{antecedent}(l)$ merken.

Beim Konflikt aus Abbildung 3.3 entsteht folgende Kette:

$$
\begin{aligned}
1 \vee \overline{2} \vee 3 \quad &\overset{3}{\rightsquigarrow} \quad 1 \vee \overline{2} \vee 5 \vee 7 \\
&\overset{\overline{2}}{\rightsquigarrow} \quad 1 \vee \overline{4} \vee 5 \vee 7 \\
&\overset{1}{\rightsquigarrow} \quad \overline{4} \vee 5 \vee 7 \\
&\overset{\overline{4}}{\rightsquigarrow} \quad \overline{6} \vee 8 \vee 5 \vee 7 \\
&\overset{5}{\rightsquigarrow} \quad \overline{6} \vee 7 \vee 8
\end{aligned}
$$

Wir schließen das Thema konfliktbasiertes Lernen und Backtracking mit *Restarts* ab.

Definition 3.13 (Restart Transition)**.**

$$M \mid\mid \phi \Longrightarrow \emptyset \mid\mid \phi.$$

Ein Restart bricht den DPLL Algorithmus ab und startet ihn mit einer leeren Variablenbelegung von neuem, wobei die bis dahin gelernten Klauseln erhalten bleiben.

Sowohl experimentell als auch theoretisch wurde gezeigt, dass es von Vorteil ist, wenn der Solver ab und zu mit seinen aus Konflikten gewonnenen Erkenntnissen wieder an der Wurzel des Suchbaums startet. Die DECIDE-Heuristik VSIDS (wird weiter unten erläutert) gekoppelt mit den Konfliktklauseln sorgt für zielführendere Entscheidungen gleich zu Beginn des Suchprozesses. Dadurch wird vermieden, dass sich der Solver in die weniger relevanten Teile des Suchbaums verfängt. Gerade in der Anfangsphase, bei der Konfliktklausln in der Regel klein ausfallen, verkürzt sich die Suchzeit selbst bei unerfüllbaren Formeln enorm.

Der Erfolg von Restarts unterstreicht die Bedeutung von Konfliktklauseln bei der Kürzung des Suchbaums. Andererseits stellt sich die Frage, ob ein Solver mit Restarts auch immer terminiert. Durch allzu häufige Restarts könnte eine Endlosschleife entstehen, weil der DPLL Algorithmus ab einem gewissen Punkt jedes Mal unterbrochen wird, bevor er etwas Entscheidendes lernt.

Um dem entgegen zu wirken ist eine Reihe von Restart Strategien entstanden, die festlegen, wann ein Restart durchgeführt wird. Mit einem Vergleich der gebräuchlichsten Strategien befasst sich [10]. Die meisten von ihnen haben gemein, dass der Restart durchgeführt wird, nachdem eine vorher festgelegte Anzahl von Konflikten aufgetreten ist. In der Regel wird diese Zahl nach jedem Restart um einen bestimmten Faktor erhöht, damit Restarts mit der Zeit immer seltener werden, wodurch die Terminierung garantiert wird.

Eine andere Frage, die sich bei Restarts stellt, ist, welche gelernten Klauseln wieder vergessen werden dürfen. Die schwierigen Teile einer Formel produzieren sehr viele Konfliktklauseln, die den Solver mehr und mehr erdrücken, wenn sie nicht wieder gelöscht werden. Andererseits sollte ein gewisser Teil des gesammelten Wissens einen Restart überdauern, weil sonst bereits gelöste Problemstellen immer wieder von neuem angegangen werden.

Hier bietet sich eine einfache Lösung an: Jede Klausel bekommt einen Aktivitätszähler, der immer dann erhöht wird, wenn die Klausel an einem Konflikt oder einer Unit-Propagierung beteiligt ist. In regelmäßigen Abständen wird dann ein Teil der am wenigsten aktiven Klauseln vergessen.

Zum Ende dieses Kapitels gehen wir näher auf die Heuristik in DECIDE ein. Ihr Verhalten hat mit Abstand den größten Einfluss auf die Laufzeit des Solvers. Weil das SAT Problem NP-vollständig ist, kann es keine Heuristik geben, die für alle Formeln gleichermaßen gut geeignet sind. Für jede Heuristik lassen sich Formeln konstruieren, sodass die Lauftzeit des Solvers dem Worst-case $\mathcal{O}(2^{|\mathcal{V}|})$ entspricht. Trotz alledem gibt es Heuristiken, die für eine bestimmte Klasse von Problemen gut geeignet sind. Bei der Entwicklung einer Heuristik ist zu beachten, dass nicht allein die Qualität ihrer Entscheidungen ausschlaggebend ist, sondern auch der Berechnungsaufwand zählt.

Variable State Independent Decaying Sum (VSIDS) ist seit ihrer Einführung des SAT Solvers Chaff[17] im Jahr 2001 nach wie vor die populärste Heuristik. Sie zeichnet sich zum einen durch eine nahezu konstante Berechnungszeit aus und zum anderen ist sie darauf ausgerichtet, den konfliktbasierten Lernprozess voll zu unterstützen, indem vorzugsweise Literale aus den eben gelernten Klauseln ausgewählt werden.

Dieses Konzept hob die Messlatte beim Lösen schwieriger Probleme ein ganzes Stück nach oben. Der Erfolg beruht auf der Erkenntnis, dass die Ergebnisfindung eines modernen DPLL Solvers, der Restarts und konfliktbasiertes Lernen implementiert, nicht mehr allein durch einfache Tiefensuche und Backtracking zustande kommt. Es sind vor allem die Konfliktklauseln, die der Suche eine Richtung verleihen.

4 GPGPU mit CUDA

Dieses Kapitel bietet eine kleine Einführung in die Welt der Grafikkarten Programmierung mit CUDA.

Vor einigen Jahren sind wir an einen Punkt gelangt, an dem es nicht mehr praktikabel ist, die Rechenleistung von CPUs durch Anhebung der Taktrate zu steigern. Mit dem Erscheinen von Mehrkern Prozessoren waren auch die Anbieter von rechenintensiver Software gezwungen, sich an den Trend anzupassen. Es ist nun nicht mehr allein damit getan, hoch-optimierten Code zu erzeugen. Man muss darüber hinaus eine geeignete *Zerlegung* des Berechnungsproblems finden, um das Potenzial der neueren CPUs voll ausschöpfen zu können.

Diese Zerlegung zu finden und zu implementieren, ist mit traditionellen imperativen Programmiersprachen keine einfache Angelegenheit, weil deren Konzept einer sich verändernden Variable nur mit viel Mühe parallelisierbar ist. Beispielsweise funktioniert es nicht ohne weiteres, eine Liste durch mehrere Threads gleichzeitig bearbeiten zu lassen, die Elemente hinzufügen und löschen können. Alte Techniken wie Mutexes, um den Zugriff von Threads auf Variablen zu koordinieren, sind schwer zu analysieren und führen nicht selten zu Code, der weder stabil, noch performant ist.

Als Reaktion darauf zeichnet sich ein Trend zur funktionalen Programmierung ab, die in ihrer Reinform keine Variablen kennt. Code, der etwas berechnet und dabei nichts überschreibt, hat keinerlei Seiteneffekte und lässt sich daher unverändert parallelisieren.

Ein bekanntes Konzept aus der Welt der funktionalen Sprachen ist der Operator *map*. Anhand einer Menge $X \subset \mathbb{X}$ mit Elementen eines bestimmten Typs $\mathbb{X}$ und einer Abbildung $f : \mathbb{X} \to \mathbb{Y}$ transformiert *map* jedes Element aus X mit Hilfe von f.

$$map(f, X) := \{f(x) \mid x \in X\}$$

Weil f eine reine Funktion ist, spielt es bei der Korrektheit der Implementierung von *map* keine Rolle, in welcher Reihenfolge die Elemente verarbeitet werden oder ob es sich um eine sequenzielle oder parallele Ausführung handelt.

Der *map* Operator ist leicht verständlich und wurde bereits in einer Vielzahl von imperativen Programmiersprachen integriert, um die parallele Datenverarbeitung zu vereinfachen und für die Programmierer schmackhaft zu machen. In C/C++ heißt die Erweiterung OpenMP und *map* spiegelt sich dort u. a. in der Parallelisierung von for-Schleifen mit festen Iterationen wieder.

Graphics Processing Units (GPU) treiben die parallele Datenverarbeitung ins

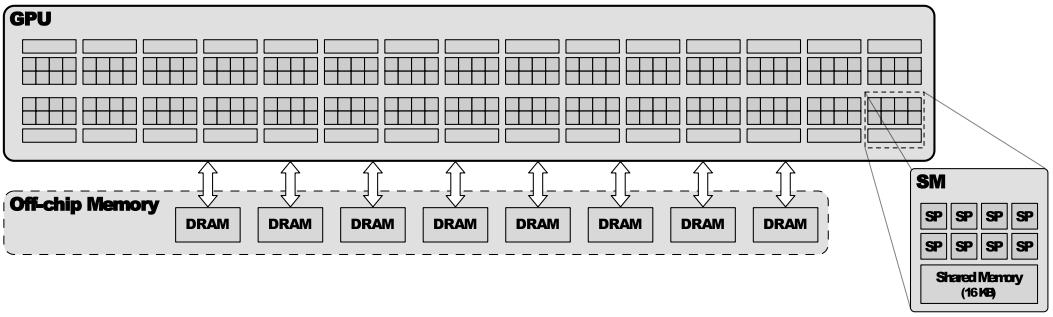

Abbildung 4.1: GeForce GTX 280 GPU mit 240 Cores (SPs), organisiert in 30 Multiprozessoren (SMs). (aus [23])

Extreme. Angetrieben von den ständig wachsenden Anforderungen der Spieleindustrie haben sich die Grafikkarten zu gewaltigen Rechenmaschinen entwickelt. Ihre Rechenkapazität hat sich bis zuletzt alle 12 bis 18 Monate verdoppelt und liegt zur Zeit bei etwa einem TFLOP/s. GPUs sind optimiert für riesige Datenmengen, deren Einzelelemente nahezu unabhängig voneinander verarbeitet werden (*map* Operator) und deren Berechnung einen hohen arithmetischen Anteil aufweist. Da für jedes Element dasselbe Programm ausgeführt wird, ist ein ausgeklügeltes Caching- und Kontrollfluss-System wie bei CPUs nicht notwendig. Die mit diesen Einschränkungen verbundene Erhöhung der Speicherzugriffszeiten werden durch den hohen arithmetischen Anteil verbunden mit geschicktem Scheduling kaschiert.

Anfangs hatten Grafikkarten eine feste Pipeline und waren kaum programmierbar. Das änderte sich schrittweise durch Einführung von Vertex- und Pixelshadern. Ein Shader ist ein kleines Programm, das einen Vertex bzw. Pixel verarbeitet und für jedes Eingabeelement in einem eigenen Thread ausgeführt wird.

Die Grafikkarten dieser Generation waren in punkto Rechenleistung den CPUs weit voraus und es dauerte nicht lange, bis sie für Berechnungen verwendet wurden, die nichts mit dem Rendern von Grafiken zu tun hatten. Für die Verwendung von GPUs für nicht-grafische Berechnungen hat sich der Begriff *General Purpose processing using GPU (GPGPU)* eingebürgert.

Die Programmierung über Shader war jedoch ein umständliches Unterfangen, da das Programmiermodell für das Rendern von Grafiken konzipiert war. So wurden beispielsweise keine Integer unterstützt und Datenstrukturen mussten über Texturen umgesetzt werden. Die Trennung der Hardware in eine Vertex- und eine Pixelverarbeitung machte damals Sinn, denn die Anforderungen bezüglich Fließkommagenauigkeit und Datenvolumen unterschieden sich stark. Mit der Einführung von Geometry- und Fragmentshadern änderte sich das. Mit *Tesla*[13] entwickelte NVIDIA eine neue GPU Architektur (Abbildung 4.1), die beide Rechenaufgaben in sich vereint.

Die GPUs dieser neuen Generation waren um Funktionen erweitert, die speziell für GPGPU zugeschnitten sind, u. a: Integer-Arithmetik, die Synchronisation und

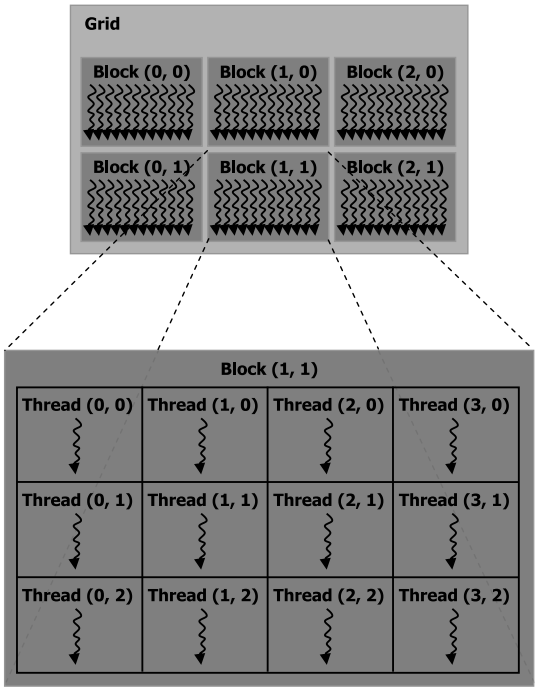

Abbildung 4.2: Thread Gruppierung [21]

der Datenaustausch zwischen Threads und eine Speicheradressierung, wie man sie von aus der CPU Welt gewohnt ist. Um GPGPU dem Mainstream zugänglicher zu machen, schuf NVIDIA mit *CUDA (Compute Unified Device Architecture)* ein Programmiermodell, das von der Software Industrie schnell adoptiert wurde. Einige bekannte Anwendungsgebiete sind z. B. Video Kodierung, physikalische Simulationen, Finanzsimulationen oder Kryptographie.

CUDA Programme werden in der Sprache C geschrieben, die um ein paar wenige syntaktische Elemente erweitert wurde. Es wurde zum einen darauf Wert gelegt, langjährigen C/C++ Entwicklern eine vertraute Umgebung beim Einstieg in GPGPU zu bieten. Zum anderen musste darauf eingegangen werden, dass die Anzahl der GPU Cores immer steigt. Es ist daher wichtig, dass sich ein einmal compiliertes Programm dynamisch an die aktuelle Hardware anpasst. Diese nahtlose Skalierbarkeit spiegelt sich im Programmiermodell durch eine zwei-stufige Thread-Gruppierung wieder (Abbildung 4.2).

Alle Threads werden in Blöcke gleicher Größe eingeteilt. Innerhalb eines Blocks sind die Threads in einem Array mit bis zu drei Dimensionen angeordnet. *Threads desselben Blocks laufen zusammen auf demselben Core.* Dadurch haben sie die Möglichkeit, sich untereinander zu synchronisieren und Daten auszutauschen. Die Anzahl von Threads auf einem Core ist begrenzt; ein Block kann zur Zeit deshalb maximal 512 Threads groß sein. In der Regel ist aber die Anzahl der Datenelemente größer als ein Block, weshalb mehrere Blöcke zu einem Grid zusammengefasst sind,

welches ebenfalls aus bis zu drei Dimensionen besteht.

Jeder Thread kennt seine Position im Block (`threadIdx`), die Position seines Blocks im Grid (`blockIdx`) und die Größe eines Blocks (`blockDim`). Mit diesen Informationen weiß er, um welchen Teil der Daten er sich kümmern muss.

Threads aus verschiedenen Blöcken laufen völlig unabhängig voneinander. Dem GPU Scheduler steht völlig frei, welche Blöcke er in welcher Reihenfolge auf welchem Core startet.

Parallel zur Gruppierung von Threads in Blöcke und der Anordnung der Blöcke zu einem Grid gibt es verschiedene Arten von Speicher.

Jeder Thread besitzt einen eigenen *lokalen Speicherbereich*, der vom Compiler zum Ablegen temporärer Daten verwendet wird falls die Register der GPU nicht ausreichen.

Alle Threads eines Blocks teilen sich einen 16kB großen Speicher, der in CUDA *Shared Memory* genannt wird. Er ist nahe am GPU Core, auf dem der Block läuft, angesiedelt und hat dadurch sehr geringe Latenzzeiten. Shared Memory ist die erste Wahl, wenn Threads miteinander kommunizieren müssen. Nach Möglichkeit wird er auch zum Vorladen von Daten aus dem globalen Speicher verwendet.

Auf den *globalen Speicher* kann vom gesamten Grid und dem Host (über CUDA API) zugegriffen werden. Er ist langsamer als der Shared Memory, bietet aber ausreichend Platz für große Datenmengen und wird nicht nach Beendigung einer Berechnung zurückgesetzt. Er wird auch *Device Memory* genannt, weil die Speicherbausteine auf der Grafikkarte liegen und der Adressraum nicht mit dem *Host Memory* der CPU geteilt wird. Während der Shared Memory für die Kommunikation zwischen Threads desselben Blocks zuständig ist, wird der globale Speicher für den Datenaustausch zwischen *hintereinander* gestarteten GPU Programmen genutzt.

Listing 4.1 zeigt ein kleines CUDA Programm, das zwei $N \times N$ Matrizen addiert. Die Prozedur `MatAdd` wurde mit dem Schlüsselwort `__global__` als *CUDA Kernel* deklariert. Ein Kernel das Programm, das auf der GPU in jedem Thread läuft.

Im Kernel werden die zwei Variablen `i` und `j` definiert, die zusammen die globale Position des Threads im Grid bestimmen. Da die Threads durch 16×16 Blöcke in einem $\frac{N}{16} \times \frac{N}{16}$ Grid organisiert sind, gleicht ihre Anordnung der Struktur der Matrizen. Die `if`-Abfrage ist für den Fall da, wenn N nicht durch 16 teilbar ist und es deswegen mehr Threads als Datenelemente gibt.

In der Hauptprozedur `main` wurden die Details, wie Daten auf die GPU geladen werden, übersprungen. Die API dafür ist der C API für Speicherallokation und Kopieren sehr ähnlich.

Interessant wird es beim Aufruf des Kernels. In CUDA ist dafür die `<<<GridSize, BlockSize>>>` Syntax vorgesehen, mit der die Dimensionen für einen Block und das Grid festgelegt werden. Reicht eine Dimension (d. h. nur die x-Komponente von `threadIdx` und `blockIdx`), dann kann statt dem neuen `dim3` Datentyp auch `int` verwendet werden.

Wir fassen zusammen: Der Entwickler wird durch die Abstraktion, die ihm CUDA bietet, angeleitet, eine Berechnung zuerst in grobe Teilberechnungen zu gliedern (Blöcke) und diese wiederum in noch kleinere Berechnungen aufzuteilen, die mitein-

```
// Kernel definition
__global__
void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < N && j < N)
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
  // Allokieren von Speicher fuer A, B, C auf der GPU
  ...
  // Laden von Daten in A und B
  ...
  // Aufruf des Kernels
  dim3 threadsPerBlock(16, 16);
  dim3 numBlocks(N / 16, N / 16);
  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
  // Ergebnis C vom GPU Speicher herunterladen
  ...
}
```

Listing 4.1: Kleines CUDA Programm, das eine $N \times N$ Matrix addiert [21].

ander kooperieren können. Erfüllt er diese Vorbedingungen, sorgt CUDA bzw. der Grafikkartentreiber dafür, dass die Cores — unabhängig davon, wieviele vorhanden sind — gleichmäßig ausgelastet werden.

Jedoch weist die Architektur einige Schwächen auf, die die Vorteile einer GPU bei einer gewissen Klasse von Algorithmen zunichte machen, obgleich diese hochparallelisierbar sind. Generell kann der Aufwand für die Konstruktion eines Algorithmus recht hoch sein, um der GPU Höchstleistungen abzuverlangen [2, 23].

Der Flaschenhals ist zum einen bei der immensen Zahl von parallelen Zugriffen auf den globalen Speicher zu finden. Haben die Threads einen gemeinsames reguläres Speicherzugriffsmuster und greifen benachbarte Threads auf benachbarte Speicherstellen zu, dann kann ein Datenpaket viele Threads auf einmal bedienen. Bei unserem `MatAdd` Programm trifft das zu.

Problematisch wird es dann, wenn die Datenstruktur keine regulären Zugriffsmuster erlaubt. Das ist in der Regel bei Graphen der Fall; auch bei denen in dieser Arbeit (siehe Abbildung 5.1). Auch wenn den GPUs solche Datenstrukturen nicht liegen, verbessert sich die Lage schrittweise durch intelligenteres Caching und steigenden Durchsatz beim Datentransport, wie Experimente in [4] zeigen.

Der andere Flaschenhals hängt mit der Art und Weise zusammen, wie auf der GPU Threads ausgeführt werden. Damit die große Anzahl Threads eines Cores, die zudem alle dasselbe Programm ausführen, effizient gescheduled werden kann, schließt die Hardware bis zu 32 Threads zu einer Einheit zusammen, die als ein *Warp*

bezeichnet wird. Ein Warp wird im Single Instruction Multiple Threads (SIMT) Stil ausgeführt, d. h. alle Mitglieder arbeiten im Gleichschritt.

Verfolgen einige Threads einen anderen Codepfad (z. B. den else-Block eines if-Statements), muss der Warp-Scheduler die unterschiedlichen Pfade serialisieren und hintereinander ausführen, während er dabei jeweils nur einen Teil der Threads aktiviert. Das wirkt sich verheerend auf die Performanz von Kernels mit vielen Codepfaden aus. Anzutreffen ist das bei Schleifen, deren Iterationen zwischen Threads weit auseinanderliegen. Unbedingt zu vermeiden sind daher verschachtelte Schleifen. Zwei oft angewendete Tricks, um verschachtelte Schleifen aufzulösen, sind:

- Die zwei Schleifen werden zu einer Ganzen verschmolzen, indem die Anzahl der äußeren Iterationen mit der Anzahl der inneren Iterationen multipliziert wird. Mit dem `div`- und `mod`-Operator werden die Werte der inneren und äußeren Schleifenvariable rekonstruiert.

- Auslagerung der Schleife in einen seperaten Kernel. Falls in der inneren Schleife ein Wert akkumuliert wird, ist es manchmal sinnvoll, diesen in einem seperaten Kernel vorauszuberechnen.

Einfacher sieht die Lage bei if-Branches aus. Diese sind zwar in der Regel unverzichtbar, doch [4] fanden heraus, dass die sequentielle Codepfad Ausführung ausbleibt, wenn die Branches so klein sind, dass sie zusammen in den Instruction-Cache passen. Damit stellen Konstrukte wie `x > y ? x : y` kein Problem dar.

Ein weiterer wichtiger Aspekt, der beim Implementieren der Algorithmen in dieser Arbeit zum tragen kam, ist die Anzahl der Register, die ein Kernel benötigt. Jeder Prozessor verfügt je nach Modell über eine bestimmte Anzahl an Registern, die er nach Bedarf auf seine acht Cores aufteilen kann. Braucht ein Warp zuviele Register für seinen Kernel, dann muss der Prozessor ein oder mehrere Cores abschalten, da nicht genügend für alle acht gleichzeitig vorhanden sind. Die Grafikkarte des Rechners, auf der diese Arbeit verfasst wurde, konnte die Kernels aus Kapitel 6 nur mit 67% Auslastung ausführen. Sobald der Compiler einem Thread mehr als acht Register zubilligt, reichen die Register nicht aus. Die Kernels aus Kapitel 6 brauchen bis zu 14 Register pro Thread. Glücklicherweise sind aktuellere GPUs mit weit mehr Registern ausgestattet, sodass keine Veranlassung bestand, das Problem mit einem kompromissbehafteten Trick zu beheben.

Registermangel ist sehr unbequem, weil ihm außer der Optimierung des Kernels schwer beizukommen ist. Ein paar Ideen zur Migration wären:

- Aufteilung des Kernels, sodass der Host mehrere Teilkernels hintereinander aufruft. Diese Strategie scheint wenig erfolgsversprechend zu sein, denn einerseits kosten Kernelaufrufe Zeit und andererseits würde der mit der Aufteilung verbundene Datenaustausch über den globalen Speicher die Verbesserung wieder zunichte machen.

- Variablen entfernen, die temporäre Zwischenergebnisse enthalten, und ihren Inhalt stattdessen bei jeder Verwendung neu berechnen. Auf diese Weise wird

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per second on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

Listing 4.2: Beispiel Thrust Programm [26]

Speicherzugriffszeit mit billiger Rechenzeit getauscht. Das geht selbstverständlich nur gut, wenn dafür kein Zugriff auf den globalen Speicher notwendig ist. Außerdem ist zu beachten, dass es auch teure Berechnungen gibt, z. B. rät NVIDIA auf Divisionen wenn möglich zu verzichten [20].

Alle Ideen haben gemeinsam den Nachteil, dass sich der Programmierer über die CUDA Abstraktion hinaus Gedanken über die Beschaffenheit einer bestimmten GPU machen muss und „Optimierungen" vornimmt, die bei einer neueren Generation von Grafikkarten gar nicht notwendig wäre.

Wir schließen das Kapitel mit der C++ Bibliothek *Thrust* ab, die Teil des CUDA Toolkits ist. Thrust verbindet CUDA mit der in C++ typischen Art zu programmieren. Die CUDA API Funktionen (u. a. `cudaMemcpy`, `cudaMalloc` und `cudaFree` zur Speicherverwaltung) werden in Thrust typ-sicherer verpackt, wie das Beispiel in Listing 4.2 zeigt.

Neben `thrust::sort` werden zahlreiche weitere oft benötigte Algorithmen angeboten, bei denen allesamt viel Arbeit in die dahinter verborgenen Kernels gesteckt wurde. Der Stil der Thrust API wurde dabei dem der C++ Standard Template Library nachempfunden.

5 Stochastische Verfahren

5.1 Random Walk SAT (WSAT) und CGWSAT

WSAT[24] ist eine vergleichsweise alte, aber erstaunlich gut funktionierende Methode, um Lösungen für zufällige 3-SAT Formeln zu finden. WSAT versucht im Grunde nichts weiter, als das Minimum einer Funktion zu finden, die anhand einer Variablenbelegung die Anzahl der noch unerfüllten Klauseln zurück gibt. Wie sein Vorgänger Greedy SAT sucht WSAT durch das Umkehren einer Variable den größten Zuwachs an erfüllten Klauseln. Um jedoch nicht durch lokale Minima in einer Sackgasse zu landen, sind zufällige Sprünge erlaubt, auch wenn die Anzahl der unerfüllten Klauseln dadurch steigt.

Der Nachteil von WSAT ist, dass er sich ähnlich wie DPLL nicht in simpler Form parallelisieren lässt. Die triviale Form der Parallelisierung wäre, mehrere Instanzen mit unterschiedlich initialisierten Zufallsgeneratoren zu starten und sie untereinander nach der Lösung wetteifern zu lassen.

CGWSAT[6] ist ein Solver, der das Konzept von WSAT mit zellulären genetischen Algorithmen verbindet. Die „DNA" entspricht dabei der Variablenbelegung. Je mehr Klauseln die DNA erfüllt, desto mehr kann sie sich in der Population durchsetzen. Der Ansatz ist hoch parallelisierbar und wurde bereits auf einer GPU umgesetzt[18, 28].

5.2 Survey Propagation

Die Algorithmen, die in dieser Arbeit auf der GPU implementiert wurden, setzen auf die Kommunikation zwischen Klauseln und Variablen in einem Faktor-Graph. Der Schwerpunkt liegt dabei auf dem Survey Propagation (SP) Algorithmus[3, 14].

Der *Faktor-Graph* einer SAT Formel ist ein ungerichteter, bipariter Graph, dessen Knotenmenge aus den Variablen und Klauseln der Formel besteht (Abbildung 5.1). Die Klauselknoten werden rechteckig, die Variablenknoten rund dargestellt. Kanten gibt es ausschließlich zwischen einem Variablen- und einem Klauselknoten und zwar genau dann, wenn die Variable in der Klausel vorkommt. Die Kante wird gestrichelt dargestellt, wenn die Variable in negierter Form in der Klausel enthalten ist.

Gefunden wurde SP 2002 von Mezard, Parisi und Zecchina und ist zur Zeit das schnellste Mittel für schwere k-SAT Probleme. Selbst Formeln mit einer Millionen Variablen lassen sich damit in nahezu linearer Zeit bewältigen[11].

Im weiteren Verlauf dieser Arbeit sind a und b Bezeichner für Klauselindizes; i und j stehen für Variablenindizes. Bei SP kommunizieren Klauseln und Variablen

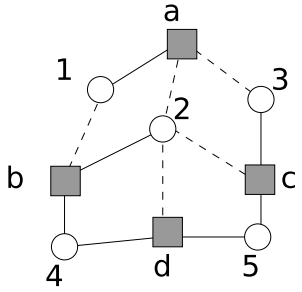

Abbildung 5.1: Faktor-Graph der Formel $\phi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_4) \wedge (\overline{x_2} \vee x_3 \vee x_5) \wedge (\overline{x_2} \vee x_4 \vee x_5)$ (aus [14]).

über die Kanten des Faktor-Graphs folgendermaßen:

- $\eta_{a \to i} \in [0..1]$ ist eine *Warnung*, die von der Klausel C_a zur Variablen x_i geschickt wird. Ist $\eta_{a \to i} = 1$, bedeutet dies, dass x_i die Klausel unter allen Umständen erfüllen muss, weil keine andere Variable in C_a dafür in Frage kommt. Umgekehrt bedeutet $\eta_{a \to i} = 0$, dass x_i in keiner Weise durch C_a eingeschränkt ist.

- $(\Pi^u_{i \to a}, \Pi^s_{i \to a}, \Pi^*_{i \to a})$ ist ein Nachrichtentupel von x_i nach C_a, mit dem die Variable den Grad ihrer *Unterstützung* für die Klausel mitteilt. $\Pi^u_{i \to a}$ steht für die Nachricht „unsupported", was bedeutet, dass die Variable zu einem Wert tendiert, der C_a nicht erfüllt. $\Pi^s_{i \to a}$ steht analog für die Zusage, dass die Variable genau zu dem Wert tendiert, der C_a erfüllt. Die dritte Komponente $\Pi^*_{i \to a}$ ist eine „don't-care" Nachricht, die aussagt, dass die Variable von keiner Klausel im System benötigt wird und es daher egal ist, welchen Wert sie annimmt.

Die Nachrichten sind als Wahrscheinlichkeitswerte zu interpretieren, wobei die $\Pi_{i \to a}$ Werte normalisiert betrachtet werden müssen:

$$\frac{\Pi^u_{i \to a}}{\Pi^u_{i \to a} + \Pi^s_{i \to a} + \Pi^*_{i \to a}} \in [0..1]$$

Survey Propagation geht davon aus, dass $\Pi^u_{i \to a} + \Pi^s_{i \to a} + \Pi^*_{i \to a}$ immer größer als 0 ist. Der SP Algorithmus sieht so aus:

1. Am Anfang werden alle $\eta_{a \to i}$ Werte mit zufälligen Zahlen im Bereich (0..1) initialisiert.

2. Aus den $\eta_{a\to i}$ Werten, werden die $\Pi_{i\to a}$ Nachrichten wie folgt berechnet:

$$\begin{aligned}
\Pi^u_{i\to a} &= \left[1 - \prod_{b\in\mathcal{C}^u_a(i)} (1-\eta_{b\to i})\right] \prod_{b\in\mathcal{C}^s_a(i)} (1-\eta_{b\to i}) \\
\Pi^s_{i\to a} &= \left[1 - \prod_{b\in\mathcal{C}^s_a(i)} (1-\eta_{b\to i})\right] \prod_{b\in\mathcal{C}^u_a(i)} (1-\eta_{b\to i}) \\
\Pi^*_{i\to a} &= \prod_{b\in\mathcal{C}(i)\setminus\{a\}} (1-\eta_{b\to i})
\end{aligned}$$

$\mathcal{C}^u_a(i)$ ist die Menge aller Klauseln, in denen die Variable x_i in anderer Polarität wie in C_a vorkommt. Analog dazu ist $\mathcal{C}^s_a(i)$ die Menge der Klauseln außer C_a, in denen x_i in der gleichen Polarität wie in C_a vorkommt.

Es ist ein wichtiges Element in SP, dass die Nachricht, die an eine Klausel verschickt wird, unabhänig von der Nachricht ist, die die Variable zuvor von eben dieser Klausel bekommen hat.

3. Aus den $\Pi_{i\to a}$ Werten werden wiederum $\eta_{a\to i}$ Nachrichten erzeugt.

$$\eta_{a\to i} = \prod_{j\in\mathcal{V}(a)\setminus\{i\}} \frac{\Pi^u_{j\to a}}{\Pi^u_{j\to a} + \Pi^s_{j\to a} + \Pi^*_{j\to a}}$$

Die Schritte 2 und 3 werden solange wiederholt, bis sich die $\eta_{a\to i}$ Werte kaum noch verändern oder ein Iterationslimit erreicht wurde.

4. Aus den konvergierten Nachrichten werden die positiven und negativen Ausrichtungen der Variablen bestimmt.

$$\theta^+_i = \frac{\Pi^+_i}{\Pi^+_i + \Pi^-_i + \Pi^*_i} \qquad \theta^-_i = \frac{\Pi^-_i}{\Pi^+_i + \Pi^-_i + \Pi^*_i} \qquad \theta^*_i = 1 - \theta^+_i - \theta^-_i$$

mit

$$\begin{aligned}
\Pi^+_i &= \left[1 - \prod_{a\in\mathcal{C}^+(i)} (1-\eta_{a\to i})\right] \prod_{a\in\mathcal{C}^-(i)} (1-\eta_{a\to i}) \\
\Pi^-_i &= \left[1 - \prod_{a\in\mathcal{C}^-(i)} (1-\eta_{a\to i})\right] \prod_{a\in\mathcal{C}^+(i)} (1-\eta_{a\to i}) \\
\Pi^+_i &= \prod_{a\in\mathcal{C}(i)} (1-\eta_{a\to i})
\end{aligned}$$

5. Suche die Variable mit dem stärksten Ausschlag in eine Richtung, d. h. maximale Differenz von θ_i^+ und θ_i^-, und fixiere die Variable auf entsprechenden Wert. In der Regel ist an dieser Stelle ein Grenzwert definiert, den die Differenz überschreiten muss, damit die Variable fixiert werden darf.

   Sind dagegen alle $\eta_{a\to i}$ Werte nahe bei 0, bricht der Algorithmus mit „trivial" ab. Mit WSAT können dann die noch verbliebenen Variablen belegt werden.

6. Finde alle Klauseln, die im Schritt 5 erfüllt wurden, und nehme sie aus dem weiteren Verlauf heraus. Falls alle Variablen belegt werden konnten, bzw. alle Klauseln erfüllt wurden, dann terminiere mit „satisfiable". Ansonsten springe zurück zu Schritt 2.

Die Propagierungsregeln in Schritt 2 zeigen, dass es durchaus sein kann, dass $\Pi^u_{i\to a}$, $\Pi^s_{i\to a}$ und $\Pi^*_{i\to a}$ gleichzeitig null wird. Dieser Fall tritt genau dann auf, wenn zwei Klauseln eine Variable vollständig in zwei entgegengesetzte Richtungen zwingen wollen. In diesem Fall bricht SP mit „contradiction" ab.

Eine Schwäche von SP ist, dass Konvergenz nicht garantiert ist, d. h. die Anzahl der Iterationen in Schritt 2 und 3 ist nach oben offen. In [9] wurden die SP Aktualisierungsregeln mit der Expectation Maximization Methode neu hergeleitet, die Konvergenz garantiert. Bei den neuen Regeln — genannt EMSPG — muss allerdings in Kauf genommen werden, dass sie sich leicht in lokalen Minima verfangen und das Ergebnis zu ungenau ausfallen kann. In der Implementierung auf der GPU (Kapitel 4) wurden beide Verfahren ausprobiert.

6 Implementierung

6.1 Entwicklung der Datenstrukturen für die GPU

Die Survey Propagierung besteht aus zwei sich abwechselnden Phasen. Da am Anfang die $\eta_{a\to i}$ Werte zufällig initialisiert wurden, legen wir als erste Phase die Berechnung von den $\Pi_{i\to a} = \Pi^u_{i\to a}/(\Pi^u_{i\to a} + \Pi^s_{i\to a} + \Pi^*_{i\to a})$ Werten fest, die sich aus den $\eta_{a\to i}$ Werten ergeben. Die zweite Phase berechnet auf Basis der $\Pi_{i\to a}$ Werte die $\eta_{a\to i} = \prod_{j\in\mathcal{V}(a)\setminus\{i\}} \Pi_{j\to a}$ Werte neu.

Die erste Überlegung bei der Implementierung ist, wie die Daten auf Threads verteilt werden sollen (siehe Kapitel 4, *map*-Operator). Da $\Pi_{i\to a}$ und $\eta_{a\to i}$ als Nachrichten zwischen Klauseln und Variablen interpretiert werden können, bzw. sie den Kanten des Faktor-Graphs entsprechen, bietet sich an, einen Thread pro Kante zu erzeugen. Jedoch kann eine neue Nachricht nicht anhand der Kante selbst erzeugt werden. $\Pi_{i\to a}$ hängt von allen eingehenden $\eta_{b\to i}$ Nachrichten bzw. $\eta_{a\to i}$ von allen eingehenden $\Pi_{j\to a}$ Nachrichten ab.

Aus diesem Grund erzeugen wir in der ersten Phase einen Thread pro Variable x_i, der alle $\Pi_{i\to a}$, $a \in C(i)$ berechnet. Analog dazu wird in Phase zwei ein Thread pro Klausel a erzeugt, der alle $\eta_{a\to i}$, $i \in \mathcal{V}(a)$ berechnet. Die Datenstrukturen müssen auf diese Phasen zugeschnitten sein.

Wie in Kapitel 4 erläutert wurde, muss darauf Wert gelegt werden, irreguläre Speicherzugriffe niedrig zu halten. Ein anderes wichtiges Ziel sollte sein, dass die Daten keinen unnötigen Platz verschwenden, damit auch sehr große Formeln auf der GPU Platz haben. Diese zwei Kriterien erfordern einige Verarbeitungsschritte der Eingabeformel. Die Datenstruktur darf daher unflexibel sein, d. h. dass im Nachhinein keine neuen Klauseln hinzugefügt werden können. Falls es eine Änderung der Klauselmenge vorgenommen werden soll, muss die Datenstruktur von neuem initialisiert werden.

Im folgenden werden wir eine Datenstruktur unter diesen Gesichtspunkten schrittweise herleiten. Da die Idee dahinter anhand eines Beispiels leichter als durch formale Definitionen nachzuvollziehen ist, werden wir mit dieser SAT Formel arbeiten:

$$C_0 = x_0 \vee x_2 \quad C_1 = \overline{x_0} \vee x_1 \vee x_2 \vee \overline{x_4} \quad C_2 = x_2 \vee x_4 \vee \overline{x_5}$$
$$C_3 = x_1 \vee x_4 \qquad C_4 = \overline{x_0} \vee \overline{x_2} \vee x_3 \qquad C_5 = \overline{x_1} \vee x_3$$

Wir beginnen mit der einfachsten Datenstruktur, der Kantenmatrix.

$$E = \begin{bmatrix} (C_0, x_0) & \cdot & (C_0, x_2) & \cdot & \cdot & \cdot \\ (C_1, x_0) & (C_1, x_1) & (C_1, x_2) & \cdot & (C_1, x_4) & \cdot \\ \cdot & \cdot & (C_2, x_2) & \cdot & (C_2, x_4) & (C_2, x_5) \\ \cdot & (C_3, x_1) & \cdot & \cdot & (C_3, x_4) & \cdot \\ (C_4, x_0) & \cdot & (C_4, x_2) & (C_4, x_3) & \cdot & \cdot \\ \cdot & (C_5, x_1) & \cdot & (C_5, x_3) & \cdot & \cdot \end{bmatrix}$$

Die Kantenmatrix liefert ein Datenlayout, um Informationen in Kanten zu speichern. Zum einen lassen sich damit die $\eta_{a \to i}$ und $\Pi_{i \to a}$ Werte speichern, zum anderen aber auch Informationen über den Graph selbst, z. B. wohin eine Kante führt oder welche Kanten mit einem Knoten verbunden sind.

Die $\cdot$ Stellen verweisen auf Kanten, die im Faktor-Graph nicht enthalten sind und markieren daher nicht verwendeten Platz. Der Nachteil ist offensichtlich: Der Platzverbraucht ist mit $|\mathcal{V}| \cdot |\mathcal{C}|$ unzumutbar bei großen Formeln. Außerdem ist das Iterieren über alle Kanten einer Variablen aufwändig, weil dem Algorithmus beim Durchlaufen einer Spalte große Löcher begegnen. Das gilt natürlich auch beim Durchlaufen einer Zeile der Matrix, aber hier sind wir mehr an der Spalte interessiert.

Der Grund dafür ist folgender: Wir gehen davon aus, dass die Matrix in Row-Major Form im Speicher liegt, d. h. dass die Zeilen nebeneinander liegen. Wenn wir für jede Spalte der Matrix einen Thread erstellen, der durch alle Elemente seiner Spalte iteriert, bekommen wir das gewünschte reguläre Speicherzugriffsmuster, d. h. es greifen zum selben Zeitpunkt alle Threads eines Warps auf einen zusammenhängenden Speicherbereich zu.

Der Trick, um die nutzlosen $\cdot$ Stellen der Matrix loszuwerden, besteht darin, E als eine Abbildung von Variablen zu ihrer Kantenmenge zu sehen, also:

$$E : v \longrightarrow \{(C, v) \mid C \in \mathcal{C}_v\}$$

Es ergibt sich dadurch folgendes neues Datenlayout:

$$E = \begin{bmatrix} (C_0, x_0) & (C_1, x_1) & (C_0, x_2) & (C_4, x_3) & (C_1, x_4) & (C_2, x_5) \\ (C_1, x_0) & (C_3, x_1) & (C_1, x_2) & (C_5, x_3) & (C_2, x_4) & \cdot \\ (C_4, x_0) & (C_5, x_1) & (C_2, x_2) & \cdot & (C_3, x_4) & \cdot \\ \cdot & \cdot & (C_4, x_2) & \cdot & \cdot & \cdot \end{bmatrix}$$

Einem Thread muss nur E, die Größe von E und seine Spaltennummer übergeben werden, damit er über alle Kanten der Spalte iterieren kann. Er ist fertig, wenn der Schleifenzähler die Zeilenanzahl von E erreicht hat oder wenn er auf eine $\cdot$ Stelle trifft.

Speicherplatz wird zwar immernoch verschwendet (obgleich weniger als zuvor), aber die Performanz des Kernels leidet nicht mehr darunter. Nebenbei erlaubt es die kompakte Anordnung der Kante nicht mehr ohne weiteres, anhand einer Variablen v und einer Klausel C, die richtige Stelle für (C, v) in E zu lokalisieren. Es ist allein möglich, die i-te Kante einer Variable v zu erfragen. Wie wir später sehen werden,

reicht uns das vollkommen.

Um den Rest der ungenutzen · Stellen loszuwerden, bedarf es einer Neusortierung der Variablen nach der Anzahl ihrer Vorkommen. Wir bezeichnen mit $\hat{x_i}$ die i-te Variable nach der Sortierung. So ist $\hat{x_0}$ die Variable, die in den meisten Klauseln vorkommt. Für die Kommunikation mit Algorithmen außerhalb der GPU Implementation merken wir uns die Beziehung $\mathcal{V} \leftrightarrow \hat{\mathcal{V}}$. Nach der Sortierung sieht E wie folgt aus:

$$E = \begin{bmatrix} (C_0, \hat{x_0}) & (C_0, \hat{x_1}) & (C_1, \hat{x_2}) & (C_1, \hat{x_3}) & (C_4, \hat{x_4}) & (C_2, \hat{x_5}) \\ (C_1, \hat{x_0}) & (C_1, \hat{x_1}) & (C_3, \hat{x_2}) & (C_2, \hat{x_3}) & (C_5, \hat{x_4}) & \\ (C_2, \hat{x_0}) & (C_4, \hat{x_1}) & (C_5, \hat{x_2}) & (C_3, \hat{x_3}) & & \\ (C_4, \hat{x_0}) & & & & & \end{bmatrix}$$

E ist jetzt keine Matrix mehr, sondern ein eindimensionales Array von Kanten. Zur Navigation in E benötigen wir noch zwei Hilfsarrays:

$$\begin{aligned} \mathit{offsets}_E &= (0, 6, 11, 15) \\ \mathit{count}_E &= (4, 3, 3, 3, 2, 1) \end{aligned}$$

$\mathit{offsets}_E$ enthält die Indizes der Zeilenanfänge. Das erste Element ist trivialerweise immer 0. Wir lassen die 0 der Einfachheit halber drin. In der Implementierung könnte durch Weglassen eine 4 Byte Übertragung aus dem globalen Speicher gespart werden. count_E enthält die Anzahl der Elemente jeder Spalte.

Sei ein Spaltenindex s gegeben, dann kann über die Elemente der Spalte wie folgt iteriert werden:

```
const int numElems = count_E[s];
for (int i = 0; i < numElems; ++i)
  elem = E[s + offsets_E[i]];
```

Dieser Code erzeugt in einem CUDA Kernel, für den ein Thread pro Spalte startet, ausschließlich reguläre Speicherzugriffsmuster. $\mathit{count}_E[s]$ veranlasst benachbarte Threads eines Warps, zur selben Zeit auf benachbarte Elemente des Arrays zuzugreifen. Der Ausdruck $\mathit{offsets}_E[i]$ liefert einen Wert, der für alle Threads des Warps gleich ist. [4] hat herausgefunden, dass Lese-Zugriffe mehrerer Threads auf dieselbe Adresse sehr schnell sind. Weil $\mathit{offsets}_E[i]$ für alle Threads gleich ist, folgt, dass $E[s + \mathit{offsets}_E[i]]$ wieder ein reguläres Zugriffsmuster generiert.

Die eingangs gestellten Ziele wurden bezüglich dem Iterieren über Kanten von Variablen erfüllt. $\Pi_{i \to a}$ lässt sich dadurch effizient von der GPU berechnen. Für $\eta_{a \to i}$ muss jedoch über die Kanten von Klauseln iteriert werden. Wir generieren daher analog zu E die Datenstruktur E^T, sodass die Klauseln die Spalten sind und $\hat{C_0} \in \hat{\mathcal{C}}$ die Klausel mit den meisten Literalen ist.

Allerdings muss der Kernel, der über die Kanten einer Variable iteriert (Phase eins), und sein Gegenspieler, der über die Kanten einer Klausel iteriert (Phase zwei), auf $\Pi_{i \to a}$ und $\eta_{a \to i}$ Werte zugreifen, weshalb wir uns bei Π und η auf ein Datenlayout

einigen müssen. Je nachdem, ob dafür E oder E^T gewählt wird, kann entweder der eine oder der andere Kernel mit ausschließlich regulären Speicherzugriffen darauf arbeiten. Da wir auf einem Graphen operieren, ist ein gewisser Anteil an irregulären Speicherzugriffen nicht vermeidbar.

Der Versuch, irreguläre Speicherzugriffe minimal zu halten, sieht folgendermaßen aus: Wir wählen für Π und η das Datenlayout E. Zusätzlich definieren wir einen Übersetzer T mit Datenlayout E^T, der zu einer i-ten Kante einer Klausel auf die entsprechende Kante im Datenlayout E durch einen Arrayindex verweist.

In unserem Beispiel sieht T so aus:

$$
\begin{aligned}
T &= \begin{bmatrix}
\mathrm{ix}_E(C_1,x_0) & \mathrm{ix}_E(C_2,x_2) & \mathrm{ix}_E(C_4,x_0) & \mathrm{ix}_E(C_0,x_0) & \mathrm{ix}_E(C_3,x_1) & \mathrm{ix}_E(C_5,x_1) \\
\mathrm{ix}_E(C_1,x_1) & \mathrm{ix}_E(C_2,x_4) & \mathrm{ix}_E(C_4,x_2) & \mathrm{ix}_E(C_0,x_2) & \mathrm{ix}_E(C_3,x_4) & \mathrm{ix}_E(C_5,x_3) \\
\mathrm{ix}_E(C_1,x_2) & \mathrm{ix}_E(C_2,x_5) & \mathrm{ix}_E(C_4,x_3) & & & \\
\mathrm{ix}_E(C_1,x_4) & & & & &
\end{bmatrix} \\
&= \begin{bmatrix}
\mathrm{ix}_E(\hat{C_0},\hat{x_1}) & \mathrm{ix}_E(\hat{C_1},\hat{x_0}) & \mathrm{ix}_E(\hat{C_2},\hat{x_1}) & \mathrm{ix}_E(\hat{C_3},\hat{x_1}) & \mathrm{ix}_E(\hat{C_4},\hat{x_2}) & \mathrm{ix}_E(\hat{C_5},\hat{x_0}) \\
\mathrm{ix}_E(\hat{C_0},\hat{x_2}) & \mathrm{ix}_E(\hat{C_1},\hat{x_3}) & \mathrm{ix}_E(\hat{C_2},\hat{x_0}) & \mathrm{ix}_E(\hat{C_3},\hat{x_0}) & \mathrm{ix}_E(\hat{C_4},\hat{x_3}) & \mathrm{ix}_E(\hat{C_5},\hat{x_4}) \\
\mathrm{ix}_E(\hat{C_0},\hat{x_0}) & \mathrm{ix}_E(\hat{C_1},\hat{x_5}) & \mathrm{ix}_E(\hat{C_2},\hat{x_4}) & & & \\
\mathrm{ix}_E(\hat{C_0},\hat{x_3}) & & & & &
\end{bmatrix} \\
&= \begin{bmatrix}
9 & 11 & 14 & 3 & 7 & 12 \\
1 & 8 & 15 & 0 & 13 & 10 \\
6 & 5 & 4 & & & \\
2 & & & & &
\end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{offsets}_{E^T} &= (0,6,12,15) \\
\mathit{count}_{E^T} &= (4,3,3,2,2,2)
\end{aligned}
$$

Sei s der Index der Klausel C_s. Dann kann über die Informationen ihrer Kanten im Datenlayout E wie folgt iteriert werden:

```
const int numVars = count_{E^T}[s];
for (int i = 0; i < numVars; ++i)
  elem = E[T[s + offsets_{E^T}[i]]];
```

Auch wenn $E[T[s + \mathit{offsets}_{E^T}[i]]]$ suspekt erscheinen mag; der Ausdruck $T[s + \mathit{offsets}_{E^T}[i]]$ generiert ein reguläres Speicherzugriffsmuster. Allein $E[...]$ sorgt für irreguläre Zugriffe.

In der nächsten Optimierung von E (und E^T) wird die Anordnung der Zeilen verbessert. Aus Kapitel 4 wissen wir, dass es günstig ist, wenn ein Warp einen Speicherbereich aus dem globalen Speicher anfordert, der an 64 Byte Blöcken ausgerichtet ist. Wir können davon ausgehen, dass der Speicherbereich, den CUDA für uns allokiert, mit einer Adresse anfängt, die sich durch 64 teilen lässt. D. h. die erste Zeile von E/E^T ist perfekt ausgerichtet. Für alle weiteren Zeilen gilt das nicht, da ihre Länge nur sehr selten ein Vielfaches von 64 Bytes ist. Der Warp muss deshalb mehr Anfragen an den globalen Speicher schicken, als notwendig wäre.

Um daher alle Zeilen am jeweils nächst höheren 64 Byte Block auszurichten, muss $offsets_E$ bzw. $offsets_{E^T}$ angepasst werden. Zur Demonstration wenden wir die Optimierung auf T mit einer 16 Byte Ausrichtung an, wobei ein Integer 4 Bytes groß ist.

$$\begin{aligned} T &= \begin{bmatrix} 9 & 11 & 14 & 3 & 7 & 12 & \cdot & \cdot \\ 1 & 8 & 15 & 0 & 13 & 10 & \cdot & \cdot \\ 6 & 5 & 4 & \cdot & & & & \\ 2 & & & & & & & \end{bmatrix} \\ offsets_{E^T} &= (0, 8, 16, 20) \end{aligned}$$

6.2 SP CUDA Kernels

Mit den Datenstrukturen in Tabelle 6.1 können die zwei Phasen von SP implementiert werden (Listing 6.1 und 6.2). Die gezeigten Kernels sind nicht effizient implementiert, da sie eine innere for-Schleife besitzen, sodass es abhängig von `numDepClauses`/`numDepVars` quadratisch viele Schleifendurchläufe gibt.

Dafür ist der Code gut geeignet, um die Arbeit mit den Datenstrukturen zu demonstrieren. Hier ist der Grund zu sehen, warum für Π und η das E Format statt dem E^T gewählt wurde: `computePiValues` ist länger, hat mehr lokale Variablen und mehr Programmzweige als `computeEtaValues`. Als Ausgleich kommt `computePiValues` ganz ohne irreguläre Speicherzugriffe aus, während `computeEtaValues` in Zeile 18 und 21 jeweils einen hat.

Der Code in Listing 6.1 und 6.2 wurde anhand einer zufällig generierten 3-SAT Formel mit 2360 Variablen und 10000 Klauseln ($\alpha = 4.24$) getestet. `computePiValues` war etwa fünf mal so schnell wie `computeEtaValues` auf einer NVIDIA Quadro FX 1700M mit Compute Capability 1.1. Das zeigt, wie teuer diese zusätzlichen Speicherzugriffe sind.

Die entgültige Fassng des SP Codes enthält keine verschachtelten Schleifen mehr und die Idee, die der Optimierung zugrunde liegt, leitete sic aus [25, 15] ab. Wir rufen uns nochmals in Erinnerung, wie die $\Pi_{i\to a}$ Werte berechnet werden:

$$\Pi_{i\to a} = \frac{\Pi^u_{i\to a}}{\Pi^u_{i\to a} + \Pi^s_{i\to a} + \Pi^*_{i\to a}}$$

$$\Pi^u_{i\to a} = (1-\alpha)\beta \qquad \Pi^s_{i\to a} = (1-\beta)\alpha \qquad \Pi^*_{i\to a} = \alpha\beta$$

$$\alpha = \prod_{b\in\mathcal{C}^u_a(i)} (1-\eta_{b\to i}) \qquad \beta = \prod_{b\in\mathcal{C}^s_a(i)} (1-\eta_{b\to i})$$

D. h. alle $\Pi_{i\to a}$ mit demselben i unterscheiden sich in ihrer Berechnung von β nur um einen Faktor, weil $\mathcal{C}^s_a(i)$ nicht a selbst enthält. Die innere Schleife in Lis-

Bezeichnung	Datenlayout	Beschreibung
`C`	E	Nur Lesen. `C` beschreibt die Abbildung $$i \longrightarrow \left\{h(C, \hat{x_i}) \mid C \in \hat{\mathcal{C}}(\hat{x_i})\right\}$$ $$h(\hat{C_a}, v) = \begin{cases} 2a & v \in \hat{C_a} \\ 2a+1 & \overline{v} \in \hat{C_a} \end{cases}$$ `C` liefert zu einer Variablen $\hat{x_i}$ die Indizes der Klauseln, in denen $\hat{x_i}$ vorkommt, plus der Polaritäten der Variable. Die Art und Weise, wie mittels h der Index und die Polarität platzsparend in einen Integer kodiert wird, wurde aus MiniSAT[5] übernommen. Mit dem Right-Shift `C`[...] $\gg 1$ wird der Klauselindex und mit der Bit-Operation AND `C`[...] & 1 wird die Polarität extrahiert.
T	E^T	Nur Lesen. Abbildung des E^T Datenlayouts nach E.
$count_{E/E^T}$	Array	Nur Lesen. Anzahl der Zeilen einer Spalte.
$offsets_{E/E^T}$	Array	Nur Lesen. Indizes der Zeilenanfänge in E/E^T Datenstrukturen.
η	E	Lesen und Schreiben. Enthält alle $\eta_{a \to i}$ Werte.
Π	E	Lesen und Schreiben. Enthält alle $\Pi_{i \to a}$ Werte.
`V`	E^T	Nur Lesen. `V` liefert zu einer Klausel $\hat{C_a}$ die Indizes ihrer Variablen plus den Polaritäten. $$a \longrightarrow \left\{l(\hat{C_a}, v) \mid v \in \hat{C_a}\right\}$$ $$l(C, \hat{x_i}) = \begin{cases} 2i & \hat{x_i} \in C \\ 2i+1 & \overline{\hat{x_i}} \in C \end{cases}$$
μ^+/μ^-	Array	Lesen und Schreiben. Enthält von jeder Variable den positiven bzw. negativen Freiheitswert.

Tabelle 6.1: Datenstrukturen der GPU Implementierung. Die Strukturen der zweiten Tabellenhälfte werden für die optimierten Kernels ohne verschachtelte Schleifen benötigt.

```
__global__
void computePiValues(const int |V|,
                     const int  *count_E,
                     const int  *offsets_E,
                     const int  *C,
                     const float *η,
                     float      *Π)
{
  const int vid = blockIdx.x * blockDim.x + threadIdx.x;

  if (vid < |V|) {
    const int numDepClauses = count_E[vid];

    for (int i = 0; i < numDepClauses; ++i) {
      const int idx = vid + offsets_E[i]; // Klauselindex a = C[idx] >> 1
      const int polarity = (C[idx] & 1); // Polaritaet von vid in Ĉ_a

      float prodU = 1.0f;  // Akkumulator fuer ∏_{b∈C^u_a(vid)}(1 − η_{b→vid})
      float prodS = 1.0f;  // Akkumulator fuer ∏_{b∈C^s_a(vid)}(1 − η_{b→vid})

      for (int j = 0; j < numDepClauses; ++j) if (i != j) {
        const int idx2 = vid + offsets_E[j];
        const int polarity2 = (C[idx2] & 1);
        const float etaV = 1.0f - η[idx2];

        if (polarity == polarity2)
          prodS *= etaV;
        else
          prodU *= etaV;
      }

      const float U = (1.0f - prodU) * prodS;
      const float d = U + prodU;
      if (d > 0.0f)
        Π[idx] = U / d; // U / d ist eine Umformung von Π^u_{vid→a} / (Π^u_{vid→a}+Π^s_{vid→a}+Π^*_{vid→a})
      else
        // Wiederspruch!
    }
  }
}
```

Listing 6.1: Vereinfachter CUDA Code zur Berechnung der $\Pi_{i\to a}$ Werte (Phase 1). Weggelassen wurde die Aussortierung von Variablen und Klauseln, die nicht mehr an der Berechnung teilnehmen. Desweiteren ist die Implementierung wegen der Verschachtelten for-Schleifen ineffizient.

```
__global__
void computeEtaValues(const int |C|
                      const int  *count_{E^T},
                      const int  *offsets_{E^T},
                      const int  *T,
                      const float *Π,
                      float      *η)
{
  const int cid = blockIdx.x * blockDim.x + threadIdx.x;

  if (cid < |C|) {
    const int numDepVars = count_{E^T}[cid];

    for (int i = 0; i < numDepVars; ++i) {
      float prod = 1.0f; // Akkumulator fuer ∏_{j∈V(cid)\{i}} Π_{j→cid}

      for (int j = 0; j < numDepVars; ++j) if (i != j) {
        prod *= Π[T[cid + offsets_{E^T}[j]]];
      }

      η[T[cid + offsets_{E^T}[i]]] = prod;
    }
  }
}
```

Listing 6.2: CUDA Code zur Berechnung der $\eta_{a\to i}$ Werte (Phase 2). Auch in diesem Kernel werden durch die verschachtelte for-Schleife unnötig viele Berechnungen und Zugriffe auf den globalen Speicher getätigt.

ting 6.1 lässt viele Threads fast dieselben Werte multiplizieren. Um die Redundanz aufzulösen, wird der Kernel `computePiValues` so abgeändert, dass statt $\Pi_{i\to a}$ die Freiheitswerte μ_i^+ und μ_i^- berechnet werden mit:

$$\mu_i^+ = \prod_{a\in\mathcal{C}^+(i)} (1-\eta_{a\to i}) \qquad \mu_i^- = \prod_{a\in\mathcal{C}^-(i)} (1-\eta_{a\to i})$$

Der optimierte Kernel `computeEtaValues` bekommt die μ^+ und μ^- Arrays als Eingabe. Beim Iterieren durch die Variablen seiner zugewiesenen Klausel C_a, kann er die $\Pi_{i\to a}$ Werte wie folgt rekonstruieren:

$$\Pi_{i\to a} = \begin{cases} (1-\mu_i^-)\frac{\mu_i^+}{1-\eta_{a\to i}} & \hat{x}_i \in C_a \\ (1-\mu_i^+)\frac{\mu_i^-}{1-\eta_{a\to i}} & \overline{\hat{x}_i} \in C_a \end{cases}$$

Das Teilen durch $1-\eta_{a\to i}$ nimmt den einen überschüssigen Faktor „zurück". Ähnlich wird bei der Berechnung von $\eta_{a\to i}$ verfahren.

$$\eta_{a\to i} = \prod_{j\in\mathcal{V}(a)\setminus\{i\}} \Pi_{j\to a}$$

Hier sind bis auf einen Faktor alle $\eta_{a\to i}$ mit demselben a gleich. Daher wird der Wert $\prod_{j\in\mathcal{V}(a)} \Pi_{j\to a}$ vorberechnet und später durch das jeweilige $\Pi_{i\to a}$ wieder geteilt.

Bei der Implementierung sind zwei Details zu beachten:

Erstens müssen die Fälle abgedeckt werden, bei denen ein oder mehrere der Produktfaktoren $(1 - \eta_{a\to i})$ bzw. $\Pi_{i\to a}$ gleich 0 sind. Wenn genau einer der Faktoren gleich 0 ist, dann wird er vom Produkt ausgeschlossen. Das Resultat der Multiplikationen wird dann mit einer Notiz versehen, die Auskunft gibt, welcher der Faktoren 0 war. Wenn das Produkt später durch einen Faktor geteilt werden soll, wird dieser Faktor mit der Notiz verglichen. Stimmt er überein, dann liegt mit dem Produkt das Ergebnis bereits vor, da der Null-Faktor bereits ausgeschlossen ist. Im anderen Fall ist das Ergebnis 0.

Das andere Detail, dass aus der Optimierung folgt, liegt in der Genauigkeit der Berechnungen. Je nachdem wie weit die einzelnen Faktoren auseinander liegen, geht die Präzision bei der Division verloren. Bei den CPU Implementierungen fällt der Effekt weniger ins Gewicht, weil dort mit 64 Bit Gleitkommazahlen gerechnet wird. Auf der GPU sind Gleitkommazahlen mit doppelter Genauigkeit erst ab Compute Capability 1.3 verfügbar. Jedoch arbeitet der Kernel damit spürbar langsamer. Der Mangel an Genauigkeit scheint die Berechnungen nicht nennenswert zu beeinflussen. Ein Test mit einer zufälligen Formel mit 2000 Variablen und 8400 Klauseln ergab, dass die maximale Ungenauigkeit – ermittelt beim Vergleichen der Summe alle $\eta_{a\to i}$ Werte nach jeder Iteration – im Bereich von $4 \cdot 10^{-4}$ liegt. Die Entscheidungen für die Literalbelegungen und deren Reihenfolge blieb unverändert.

6.3 Anbindung an MiniSAT

Die Verbindung der Survey Propagierungen mit einem DPLL Solver ist nicht neu. DPLL ist gegenüber SP nicht nur ein vollständiges Verfahren, sondern es kommt zudem sehr gut mit hoch strukturierten Formeln zurecht. Bei schweren randomisierten SAT Formeln nahe der Phasentransition benötigen moderne DPLL Solver hingegen enorme Rechenzeit[16]. Bei Formeln mit 500 Variablen brauchen sie oft weit mehr als zehn Stunden. Survey Propagation löst dagegen Formeln mit Millionen Variablen in nahezu linearer Zeit[11]. Jedoch versagt das Verfahren bei strukturellen Problemen. In so gut wie allen Fällen bricht es mit einem Widerspruch ab (Zeile 37 in Listing 6.1), da es mit den engen Zyklen im Faktor-Graph nicht zurecht kommt. Außerdem ist Survey Propagation ein unvollständiges Verfahren; es kann nicht dazu verwendet werden, die Unerfüllbarkeit einer Formel zu beweisen.

Die Idee, beide Welten in einem Solver zu vereinen, der sowohl komplett ist, als auch mit schweren Zufallsformeln zurecht kommt, ist daher naheliegend. Vielleicht, so eine Hoffnung, könnte SP sogar den DPLL Prozess bei strukturellen Problemen unterstützen, z. B. wenn Teile der Formel eine ähnlich wirre Unstrukturiertheit auf-

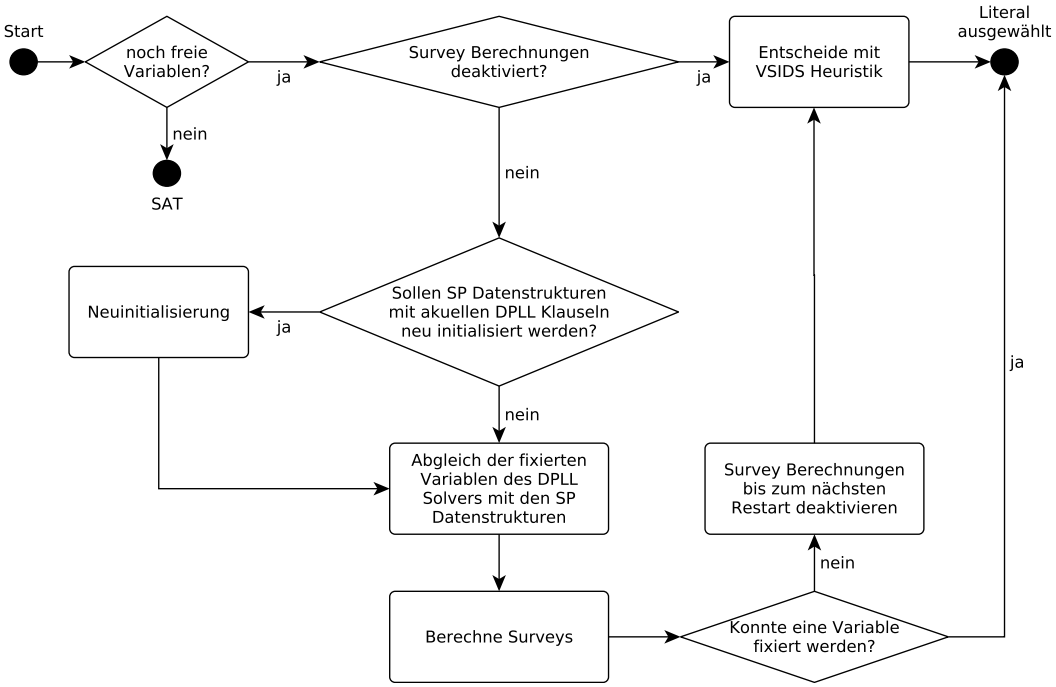

Abbildung 6.1: Erweiterte DECIDE Komponente zur Anbindung von SP an DPLL.

weisen, wie das bei zufälligen Formeln der Fall ist.

Der Solver VARSAT[9] ist der Versuch einer solchen Vereinigung. VARSAT erweitert den Solver MiniSAT[5] u.a. mit SP, EMSPG und vier weiteren Variationen, um experimentell herauszufinden, was am besten mit DPLL und mit welchen Formeln funktioniert[27]. Einerseits ist bekannt, dass SP am besten mit zufälligen Formeln zurecht kommt. Andererseits braucht es von allen Varianten am meisten Rechenleistung und seine Konvergenz ist nicht garantiert. Hinzu kommt, dass durch sein Streben nach globaler Konsistenz ohnehin bei strukturellen Problemen keine Hilfe ist.

Im Zuge dieser Arbeit ist ebenfalls ein Solver auf Basis von MiniSAT entstanden, der im Grunde dieselbe Idee wie VARSAT verfolgt, jedoch zusätzlich versucht, die Rechenpower der GPU unterstüzend mit einzubinden. Daher stammt auch ein großer Teil der Art und Weise, wie hier die Survey Algorithmen in den DPLL Prozess integriert wurden, von VARSAT ab und wird im folgenden beschrieben.

Basierend auf den Ergebnissen der VARSAT Experimente[9, 27], wurden für die GPU die Verfahren SP und EMSPG implementiert. Welches Verfahren der Solver verwenden soll, kann mittels einer Programmoption eingestellt werden.

Die Anbindung an den DPLL Teil liegt in der DECIDE Komponente (siehe Abbildung 6.1). Jedes Mal, wenn der Solver eine Entscheidung für ein Literal fällen muss, kann er dies entweder mit der sehr schnellen VSIDS Heuristik tun oder er versucht mit der langsamen Survey Propagierung Literale zu finden, die mit hoher Wahr-

scheinlichkeit erfüllt sein müssen. Die hohe Wahrscheinlichkeit ist dadurch gegeben, weil eine Variable nur festgelegt werden darf, wenn die Differenz ihrer positiven und negativen Ausrichtung einen gewissen Grenzwert überschreitet, den sogenannten *Deactivation Threshold*. Konnte keine Variable fixiert werden — entweder weil keine davon stark genug in eine Richtung tendiert oder SP mit einem Widerspruch abbricht — dann sucht der Solver mit der traditionellen VSIDS Heuristik bis zum nächsten Restart weiter.

Kurz zusammengefasst: Der Solver versucht von der Wurzel des Suchbaums so weit wie möglich, mit SP Entscheidungen zu treffen, die mit hoher Wahrscheinlichkeit korrekt sind. Die Hoffnung ist, dass SP sogenannte Backbone Variablen findet, nach deren Belegung sich die Formel stark vereinfacht. Ab der Stelle, wo SP nicht mehr weiter weiß, setzt die eigentliche DPLL Maschinerie mit Konflikt basiertem Lernen und Backtracking ein. Bei Bedarf können dann einige der gelernten Klauseln ein Feedback an SP geben. Diese Klauseln sollten jedoch nicht mehr als fünf Literale besitzen, weil längere Klauseln in der Regel zu wenig Informationen enthalten und nur die Performanz beeinträchtigen.

Weil SP auch auf einer GPU merklich Zeit benötigt, die zudem verloren geht, wenn der Algorithmus ohne Ergebnis abbricht, wurde der Code, der die GPU Kernels anwirft, in einen seperaten Thread ausgelagert. Jeder Restart des DPLL Teils ist ein möglicher Rendezvous Punkt zwischen dem DPLL Thread und dem GPU Management Thread. Abbildung 6.2 skizziert wie die GPU Berechnungen parallel zum normalen DPLL Prozess laufen können, ohne sich dabei gegenseitig aufzuhalten. Man könnte dieses Design auch als eine Art Portfolio Ansatz bezeichnen.

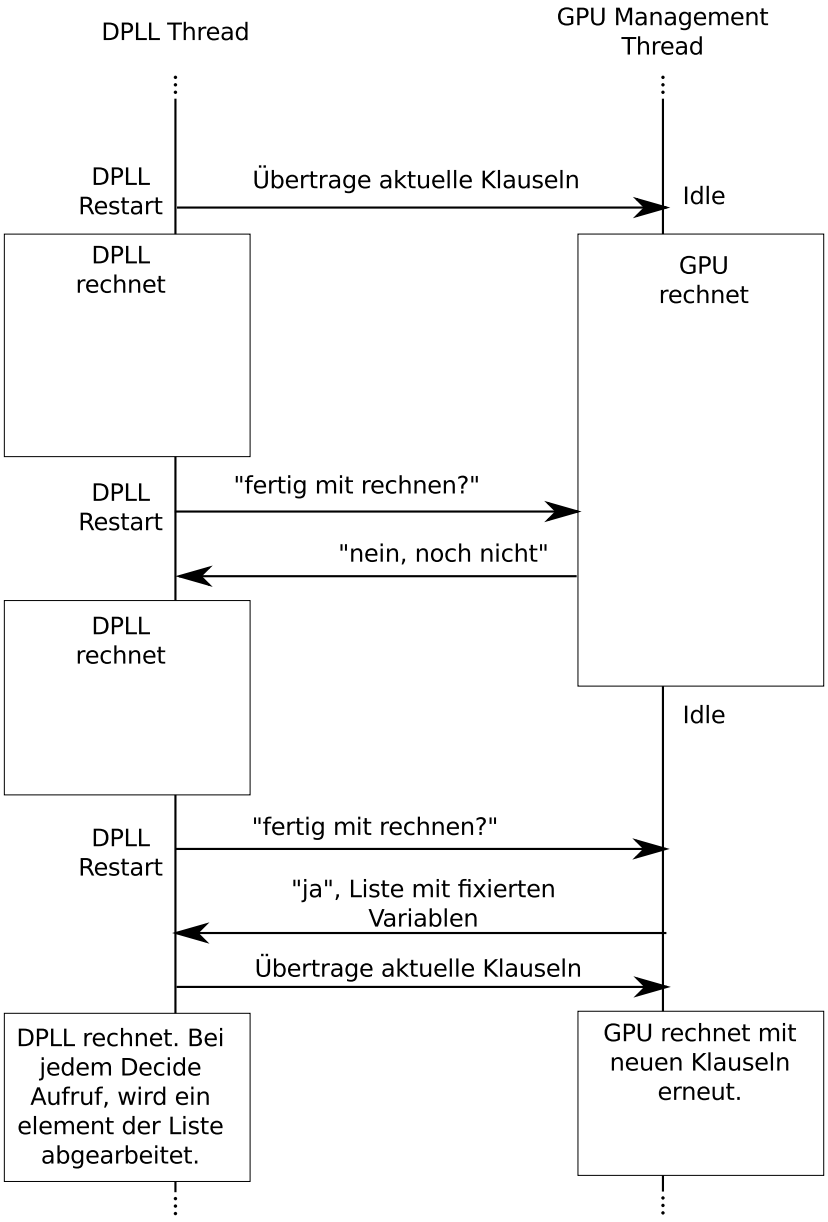

Abbildung 6.2: Skizzierung einer Kommunikation zwischen dem DPLL Thread und dem GPU Management Thread.

7 Fazit

Der SAT Solver, der aus dieser Arbeit hervorgegangen ist, erfüllt die Zielsetzung; allerdings mit gemischten Ergebnissen. Positiv zu bewerten ist, dass DPLL und SP bei schweren Zufallsformeln zusammenarbeiten können. SP bringt die notwendige globale Sicht auf die Formel, während DPLL einem Sicherungsnetz gleicht, dass mittels Backtracking und Konfliktklauseln den SP Algorithmus auf Kurs hält, falls er sich doch mal irren sollte. Zusätzlich sorgt die Auslagerung der GPU Steuerung in einen seperaten Thread dafür, dass der DPLL Teil nicht verlangsamt wird, falls SP keine brauchbaren Ergebnisse liefern kann.

Jedoch lässt diese Art der Integration noch einige Wünsche offen. Bei der Programmierung wurden einige Ideen versucht, um SP mit DPLL noch enger zusammenarbeiten zu lassen.

Eine davon stellt die Frage, ob DPLL nicht behilflich sein kann, wenn SP mit einem Widerspruch abbricht (siehe Zeile 37 in Listing 6.1). Die Variable, die die widersprüchlichen Forderungen ihrer Klauseln in sich vereint, könnte dem DPLL Teil als Decision-Literal empfohlen werden, damit er daraus Konfliktklauseln erzeugen kann, die SP helfen den Widerspruch künftig zu umgehen. So kann SP auch auf eine andere Art als Heuristik dienlich sein, selbst bei strukturellen Formeln. Leider hat die Umsetzung dieser Idee nicht überzeugen können. Es schien so, als wären die Konfliktvariablen in SP nicht die, die für DPLL von entscheidender Bedeutung sind. Möglicherweise sind es gerade die, die in DPLL sowieso häufig Unit propagiert werden.

In VARSAT wird das Thema „Abbruch durch Widerspruch" mit einem Trick ignoriert, indem die ausgehenden Nachrichten der betroffenen Variable auf $\Pi^u_{i\to a} = \Pi^s_{i\to a} + \Pi^*_{i\to a} = \frac{1}{3}$ gesetzt werden. Dadurch kann der Algorithmus weiter laufen und vielleicht gibt es am Ende doch noch die ein oder andere Variable, die stark in eine Richtung tendiert. Das könnte bei Formeln funktionieren, die nur zum Teil unstrukturiert sind. Der Nachteil ist allerdings, dass SP nicht mehr konvergiert, wenn der Trick ständig angwendet werden muss, und deswegen immer die maximale Anzahl der Iterationen erreicht wird. Aus diesem Grund wurde in dieser Arbeit neben SP auch EMSPG[9] implementiert, das eine abgewandelte Form von SP ist, bei dem Konvergenz garantiert wird. EMSPG — wenn es bei Restarts wie in Abbildung 6.1 eingesetzt wird — stellt sich als wenig nutzbringend heraus. Während SP hundere von Variablen am Stück fixieren kann, schafft EMSPG nur einen Bruchteil. Außerdem konvergiert auch EMSPG nicht, wenn der oben beschriebene Trick angewendet wird.

Nebenbei bemerkt, haben die VARSAT Autoren dagegen gute Erfahrungen mit EMSPG gemacht, während SP wenig überzeugend wirkte[27]. Wenn man jedoch ge-

nauer hinschaut, lässt sich das leicht erklären: Die Autoren habe es irgendwie — sie haben den Weg leider nicht beschrieben — geschafft, die SP Schritte 2 und 3 in Abschnitt 5.2 in eine einzige Update Regel zusammenzufassen, sodass $\eta_{a\to i}$ und $\Pi_{i\to a}$ wegfällt und direkt auf den θ_i^+ und θ_i^- Werten (Schritt 4) gearbeitet wird. Obwohl die alternative Formulierung von SP korrekt zu sein scheint, so konvergiert deren Implementierung in VARSAT in den meisten Fällen zu einem trivialen Ergebnis, d. h. alle $\theta_i^* = 1$.

Abschließend noch ein paar Worte zu den CUDA Kernels. Der Overhead, der durch die Initialisierung der Datenstrukturen, die Kernelaufrufe, das fixieren von Variablen und das Hoch- und Runterladen der Variablenbelegungen entsteht, liegt bei etwas weniger als 9% der Gesamtlaufzeit. Verglichen wurde die unoptimierten und die optimierten Versionen der Kernels mit einer Nvidia Geforce GTX 285 und einer zufälligen 3-SAT Formel mit 2360 Variablen und 10000 Klauseln ($\alpha = 4.23$). Die unoptimierte Fassung schaffte im Durchschnitt 1051 Iterationen/s. Die optimierte Version war mit 5425 Iterationen/s etwa fünf mal so schnell.

Allerdings darf dabei nicht unerwähnt bleiben, dass die unoptimierte Fassung zum Zeitpunkt der Abgabe mehr Variablen fixieren kann; selbst wenn mit `double` gerechnet wird. Der Grund für dieses Phänomen bleibt Gegenstand weiterer Untersuchungen.

Literaturverzeichnis

[1] Gilles Audemard und Laurent Simon: *Predicting learnt clauses quality in modern SAT solvers.* In: *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, Seiten 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[2] Nathan Bell und Michael Garland: *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dezember 2008.

[3] A. Braunstein, M. Mézard und R. Zecchina: *Survey propagation: An algorithm for satisfiability.* Random Struct. Algorithms, 27:201–226, September 2005, ISSN 1042-9832.

[4] Frank Dehne und Kumanan Yogaratnam: *Exploring the Limits of GPUs With Parallel Graph Algorithms.* CoRR, abs/1002.4482, 2010.

[5] Niklas Eén und Niklas Sörensson: *An Extensible SAT-solver.* In: Enrico Giunchiglia und Armando Tacchella (Herausgeber): *SAT*, Band 2919 der Reihe *Lecture Notes in Computer Science*, Seiten 502–518. Springer, 2003.

[6] Gianluigi Folino, Clara Pizzuti und Giandomenico Spezzano: *Parallel hybrid method for SAT that couples genetic algorithms and local search.* IEEE Trans. Evolutionary Computation, 5(4):323–334, 2001.

[7] Youssef Hamadi, Saïd Jabbour, Cédric Piette und Lakhdar Sais: *Deterministic Parallel DPLL.* JSAT, 7(4):127–132, 2011.

[8] Youssef Hamadi und Lakhdar Sais: *ManySAT: a parallel SAT solver.* JSAT, 6, 2009.

[9] Eric I. Hsu und Sheila A. McIlraith: *VARSAT: Integrating Novel Probabilistic Inference Techniques with DPLL Search.* In: Oliver Kullmann (Herausgeber): *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, Band 5584 der Reihe *Lecture Notes in Computer Science*, Seiten 377–390. Springer, 2009, ISBN 978-3-642-02776-5.

[10] Jinbo Huang: *The effect of restarts on the efficiency of clause learning.* In: *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, Seiten 2318–2323, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[11] Lukas Kroc, Ashish Sabharwal und Bart Selman: *Survey propagation revisited*. In: *In 23rd UAI*, Seiten 217–226, 2007.

[12] Daniel Kroening und Ofer Strichman: *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1. Auflage, 2008, ISBN 3540741046, 9783540741046.

[13] Erik Lindholm, John Nickolls, Stuart Oberman und John Montrym: *NVIDIA Tesla: A Unified Graphics and Computing Architecture*. IEEE Micro, 28:39–55, March 2008, ISSN 0272-1732.

[14] Elitza Maneva, Elchanan Mossel und Martin J. Wainwright: *A new look at survey propagation and its generalizations*. J. ACM, 54, July 2007, ISSN 0004-5411.

[15] Panagiotis Manolios und Yimin Zhang: *Implementing Survey Propagation on Graphics Processing Units*. In: Armin Biere und Carla P. Gomes (Herausgeber): *SAT*, Band 4121 der Reihe *Lecture Notes in Computer Science*, Seiten 311–324. Springer, 2006, ISBN 3-540-37206-7.

[16] David Mitchell, Bart Selman und Hector Levesque: *Hard and easy distributions of SAT problems*. In: *Proceedings of the tenth national conference on Artificial intelligence*, AAAI'92, Seiten 459–465. AAAI Press, 1992, ISBN 0-262-51063-4.

[17] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang und Sharad Malik: *Chaff: engineering an efficient SAT solver*. In: *Proceedings of the 38th annual Design Automation Conference*, DAC '01, Seiten 530–535, New York, NY, USA, 2001. ACM, ISBN 1-58113-297-2.

[18] Asim Munawar, Mohamed Wahib, Masaharu Munetomo und Kiyoshi Akama: *Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework*. Genetic Programming and Evolvable Machines, 10:391–415, December 2009, ISSN 1389-2576.

[19] Robert Nieuwenhuis, Albert Oliveras und Cesare Tinelli: *Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)*. J. ACM, 53:937–977, November 2006, ISSN 0004-5411.

[20] NVIDIA: *CUDA C Best Practices Guide*, Mai 2011.

[21] NVIDIA: *CUDA C Programming Guide*, Mai 2011.

[22] *SAT Competition 2011*. `http://www.satcompetition.org/2011`.

[23] Nadathur Satish, Mark Harris und Michael Garland: *Designing efficient sorting algorithms for manycore GPUs*. In: *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, Seiten 1–10, Washington, DC, USA, 2009. IEEE Computer Society, ISBN 978-1-4244-3751-1.

[24] Bart Selman, Henry A. Kautz und Bram Cohen: *Noise strategies for improving local search*. In: *In Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-94*, Seiten 337–343, 1994.

[25] *Survey Propagation C Implementierung*. `http://users.ictp.it/~zecchina/SP`.

[26] *Thrust Homepage*. `http://code.google.com/p/thrust`.

[27] *VARSAT Homepage*. `http://www.cs.toronto.edu/~eihsu/VARSAT`.

[28] Yandong Wang: *NVIDIA CUDA Architecture-based Parallel Incomplete SAT Solver*. Diplomarbeit, Rochester Institute of Technology, 2010.

[29] Christoph Zengler und Wolfgang Küchlin: *Extending clause learning of SAT solvers with Boolean Gröbner bases*. In: *Proceedings of the 12th international conference on Computer algebra in scientific computing*, CASC'10, Seiten 293–302, Berlin, Heidelberg, 2010. Springer-Verlag, ISBN 3-642-15273-2, 978-3-642-15273-3.

[30] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz und Sharad Malik: *Efficient conflict driven learning in a boolean satisfiability solver*. In: *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, Seiten 279–285, Piscataway, NJ, USA, 2001. IEEE Press, ISBN 0-7803-7249-2.