Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# The Hobel algorithm

## SAT Lösung mit GPU über DPLL

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering/Internet Computing**

eingereicht von

**Csaba Vaczula**
Matrikelnummer 1027394

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Asst.-Prof. Dr. Ezio Bartocci
Mitwirkung: Univ.-Prof. Dr. Armin Biere
DI Andreas Fröhlich

Wien, 31. März 2016

Csaba Vaczula

Ezio Bartocci

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# The Hobel algorithm

## SAT solving with GPU beyond DPLL

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering/Internet Computing**

by

**Csaba Vaczula**
Registration Number 1027394

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Asst.-Prof. Dr. Ezio Bartocci
Assistance: Univ.-Prof. Dr. Armin Biere
DI Andreas Fröhlich

Vienna, 31st March, 2016

Csaba Vaczula | Ezio Bartocci

# Erklärung zur Verfassung der Arbeit

Csaba Vaczula
Bartók Béla utca 8, 8800 Nagykanizsa, Hungary

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 31. März 2016

Csaba Vaczula

# Danksagung

Zu allererst möchte ich meinem Betreuer, Prof. Ezio Bartocci und Mitwirker, Prof. Armin Biere und Andreas Fröhlich auf das Beste danken. Seit der Darstellung meiner Grundidee habe ich viel Aufmerksamkeit bekommen. Während der ganzen Zeit wurde meine Arbeit durch Hilfsbereitschaft und konstruktive Besprechungen gekennzeichnet. Die viele ermunternde Worte waren bedeutend, wann ich mich mit Problemen konfrontiert fand.

Ein weiterer Dank gebührt meiner Familie, die mich während der Arbeit mit ihre Geduld unterstützt hat. Ich konnte viel Zeit mit Brainstorming verbringen und das führte zur Entwicklung vieler Fähigkeiten des Algorithmus.

Ich habe viele weiteren Hilfe in Englisch von einem meiner besten Freunde, Gergely Antal bekommen.

# Acknowledgements

First of all I would like to thank my advisor, Prof. Ezio Bartocci and co-advisors, Prof. Armin Biere and Andreas Fröhlich for the best. Since the presentation of my basic idea I got a lot of attention. All the while my work was characterized by helpfulness and constructive meetings. The many encouraging words were important, when I found myself confronted with problems.

Another thanks to my family who supported me while working with their patience. I could spend a lot of time with brainstorming and this led to the development of many features of the algorithm.

I also got a lot of help in English from one of my best friends, Gergely Antal.

# Kurzfassung

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT, von englisch satisfiability, 'Erfüllbarkeit') ist einer der wichtigsten NP-vollständiges Problem, das vielen Fällen in einer Vielzahl von Anwendungsgebieten im Bereich von Schaltung und Hardware-Design zu Rechensystembiologie finden. Trotz der Komplexität des Problems, sind moderne SAT Solver sehr ausgefeilte Werkzeuge, die Hunderttausende von Variablen und Millionen von Klauseln leicht verarbeiten kann. Eines der wirksamsten Verfahren SAT Probleme zu lösen, ist der Davis-Putnam-Logemann-Loveland (DPLL) -Algorithmus. DPLL ist eine Backtracking Suche auf Basis sequentieller Algorithmus und es ist weit verbreitet in den State-of-the-art-SAT-Solver.
Mit dem Aufkommen der hochparallele Architekturen, das Interesse der Gemeinschaft SAT Solver hat in Richtung der Parallelisierung von sequentiellen Algorithmen wie DPLL bewegt, um die Vorteile der leistungsfähigen Verarbeitungsmöglichkeiten in den modernen Multicore-Hardware-Architekturen in Anspruch zu nehmen. In dieser Arbeit nähern wir uns der Herausforderung, aus einem anderen Blickwinkel effektiv parallel SAT Solver zu entwickeln. Anstatt einen Suchalgorithmus wie die meisten Löser zu verwenden, wir verarbeiten die Klauseln nacheinander. Der Algorithmus beschreibt die möglichen Modelle zu den bereits verarbeitet Klauseln. Wenn eine neue Klausel verarbeitet worden ist, werden die beschriebenen Modelle so modifiziert werden, so dass die neu verarbeitet Klausel zufrieden sein werden.
Im Falle des Verfahrens, eine der wichtigsten Bedingungen war die Ausnutzung der GPU. Wir führen eine neue logische Darstellung ein und wir konzentrieren uns auf Single Instruction Multiple Data (SIMD). Verschiedene Art von Aufgaben wurden durchgeführt, die die GPU passen. Der Algorithmus wurde mit C ++ und CUDA implementiert. Die Tests wurden an einer NVIDIA Geforce GTX 860M Grafikkarte durchgeführt. Während der Entwicklung wurde der Algorithmus auf der Grundlage der Testergebnisse verbessert. Die Arbeit beschreibt die Grundidee und die wichtigsten Schritte dieser Verbesserungen mit den Testergebnissen.

# Abstract

The boolean satisfiability (SAT) is one of the most important NP-complete problem that find many instances in a large variety of application areas ranging from circuit and hardware design to computational systems biology. Despite the complexity of the problem, modern SAT solvers are very sophisticated tools that can easily handle hundred thousands of variables and millions of clauses. One of the most effective procedure to solve SAT problems is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. DPLL is a backtracking-search based sequential algorithm now widely employed in the state-of-the-art SAT solvers.

With the advent of highly parallel architectures, the interest of the SAT solvers community has moved toward the parallelization of sequential algorithms such as DPLL to take advantage of the powerful processing capabilities offered in the modern multi-core hardware architectures. In this thesis we approach the challenge of developing effective parallel SAT solvers from a different point of view. Instead of use a search algorithm like most of the solvers, we process the clauses after each other. The algorithm describes the possible models to the already processed clauses. When a new clause has been processed, the described models will be modified so, such that the newly processed clause will be satisfied.

In case of the procedure, one of the most important conditions was the utilization of the GPU. We introduce a new logical representation, and we concentrate on one single instruction multiple data (SIMD). Different sort of tasks have been performed which fit the GPU. The algorithm has been implemented with C++ and CUDA. The tests were performed on an NVIDIA Geforce GTX 860M graphics card. During the development, the algorithm has been improved based on the test results. The thesis describes the basic idea and the main steps of these improvements with test results.

# Contents

CHAPTER 1

# Introduction

SAT problem is one of the most important problems in computer science. It was the first proved NP-complete problem. It means that a wide range of decision and optimization problems are at most as difficult to solve as SAT. There is not any known algorithms, which can solve every possible problem instance effectively, but there are many scalable algorithms that can handle problems with hundreds of thousands variables and millions of clauses efficiently.
If a SAT problem contains N variables, there exists $2^N$ possible assignments. It is simply very useless approach to check all possible assignments one by one. DPLL procedure is still the basic of the State-of-the-art SAT solvers. The procedure got lot of improvements, better and better heuristics during the years, using the results from the actual researches of areas machine learning, probability theory.
The main problem of the modern SAT solvers is the parallel execution: it is a big challenge to use the DPLL algorithm to develop an effective parallel solver, because the algorithm has been designed to sequential work. The quad-, octa-, and more core processors need different way of thinking. On the GPU side there is also a significant revolution: today's GPUs calculate not only tasks from areas of video game, image or video processing, but also bioinformatics, computational finance, medical imaging, etc. The GPGPU offers plenty of options, but the GPU has very different architecture than the CPU and therefore, the GPUs need different way in algorithm design.

## 1.1 Motivation

The NVIDIA's new microarchitecture, the Maxwell, lifted the degree of efficiency to a very high level [Harb]. The improvements in datapath organization and instruction scheduler provide more than 40 % higher delivered performance per CUDAs core, and overall twice the efficiency of the predecessor architecture. Many other developments [Hara] in control logic partitioning, workload balancing, clock-gating granularity, instruction

scheduling, number of instructions issued per clock cycle made the in 2014 introduced Maxwell the most advanced CUDA GPU ever. The difference between the CPU and GPU in delivered GFLOP per Watt continued to increase. The numbers suggest that the designing algorithms to Maxwell is remunerative.
In this thesis we investigate the possibilities of SAT solving with such a GPU. The widely used DPLL has been designed to CPU and therefore a simple transformation of such an algorithm into GPU version is not the best way. The CPU is able to perfectly handle one or some threads with many possible if-then branches. The GPU is a different world: the main tasks of these units are the image-, video processing and calculations for 3D video games. The best is for a GPU is a SIMD task: executing one, relatively simple operation on many data elements. "Relative simple" means, that the operation should not have lot of branches, because different evaluations of many if-then statements can lead to very different set of instructions and therefore very different execution time. The GPU performs the tasks on many data element parallel and the ready threads will be in idle, until all other threads finish the work.
The question is, whether if we throw the DPLL and search approach away, is it possible to make an algorithm which fits better to the GPU? The answer is not easy: first of all, the algorithm should have many independent data element. In the moment of this writing, the strongest Maxwell based Geforce card is the Titan X. It has 3072 shader units and 12 GB of RAM. In the best case, the algorithm can feed the threads with data in many steps of the execution (if we speak about a hard problem) but the memory consumption of the procedure does not exceed the onboard memory of the GPU. It is an interesting question, whether there is a problem description, which is not minimal, so can give work for many threads and the GPU is faster with this, than the CPU with a more compact description. The following sections show that to find this balance is not easy.

## 1.2 Structure of the work

First, since the DPLL is the most widely used basic of the state-of-the-art sat solver, we give a short overview about it. Our goal is not to compare the different heuristics of the DPLL. In addition, we describe the focal points of the two main directions of the development, and we concentrate on the problems in case of parallelization. Moreover, we analyze some other parallel solvers and all of their advantages and disadvantages.
In Section 3, our developed algorithm, the Hobel and the new logical approach of the SAT problem will be introduced on a more detailed way. In the subsequent chapters, all the experiences and the resulting improvements regarding data representation and algorithm will be represented. In Section 4, the results will be concluded and we analyze the possible future developments of the approach.

CHAPTER 2

# The SAT problem

A boolean expression (propositional logic formula) consists of set of boolean variables, logical operators and parenthesis. A Boolean Satisfiability (SAT) problem is to assign true/false values to the variables so that, the formula (based on the truth table) evaluates to true.
In the following we provide the necessary definitions to understand the SAT problem.
**Definition 1 (variable):** a variable or an atom $X$ is a boolean variable which can gets value from set true (1), false (0); its negation is: $\neg X$.
**Definition 2 (literal):** a literal l is a propositional variable X (positive literal) or its negation ($\neg X$, negative literal).
**Definition 3 (set of literals):** The atoms and their negations form the set of literals.
**Definition 4 (assignment):** an assignment is a mapping from a boolean variable to a truth value. We speak about complete assignment, if for all variables there exists such a mapping, otherwise it is called partial assignment.

A clause C is built from literal(s) and a formula $\varphi$ is built from a set of clauses. We say a clause is a Horn clause, if it contains at most one positive literal. The logical connections among literals are AND (conjunction, also denoted with $\wedge$) and OR (disjunction, also denoted with $\vee$). We speak about quantified boolean formula if quantors are used, namely "there exists" ($\exists$) and "for all" ($\forall$) are accepted. If a formula evaluates to true with every possible assignments, we say the formula is a tautology. Most of the SAT solvers work with conjunctive normal form (CNF) which is the most used from the family of boolean normal forms [SP]. This formulation has the following restrictions:

- Every clause is a disjunction of literals $\bigvee_i l_i$.
- Every CNF formula is a conjunction of clauses $\bigwedge_j c_j$.
- Negation is only allowed in front of the literals.

**Definition 5 (CNF clause states):** a CNF clause can be in one of the following states:

- satisfied: the clause contains at least one literal which resolves to true.
- unsatisfied: all literals resolve to false.
- undefined: there exists at least one unassigned literal and all assigned literals resolve to false.
- unit: exactly one literal is unassigned and all assigned literals resolve to false.

**Example 1a** - Examples to CNF clause states from Definition 5.
Let be M = { A, ¬B, C, ¬D }.

| $(A \vee B)$ | $(\neg A \vee B \vee \neg C)$ | $(\neg A \vee B \vee E \vee \neg F)$ | $(\neg A \vee B \vee \neg C \vee E)$ |
|---|---|---|---|
| satisfied | unsatisfied | undefined | unit |

## 2.1 Algorithms for SAT [Ais13]

### 2.1.1 The resolution [BB10]

The resolution is a simple refutation procedure to the CNF which can prove the unsatisfiability and/or satisfiability of a formula. The main or "resolution" step is built on the following: a literal with its negative form are called complement literal pair. If there exists two clauses, in which there is only one complement literal pair, we can resolve the clauses so that, we remove the literal pair and connect the remained literals with OR connection. The produced new clause called as *resolvent*. Example:

$$(X_1 \vee X_2) \wedge (\neg X_1 \vee X_3) \Longrightarrow (X_2 \vee X_3)$$

The procedure produces a resolution graph with a set of clauses or an empty clause at the end. On the graph, each node represents an original clause or a resolvent, each arc represents a resolution step. If we get an empty clause at the end, the formula is unsatisfiable. Figure 2.1 shows an example.
Although the resolution procedure can decide, whether an expression is satisfiable, or not, it does not give a solution. Therefore, the method is useful as a proving procedure.

### 2.1.2 The DPLL [Ahm09], [GS05]

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a refinement of the Davis-Putnam algorithm, which uses the resolution procedure. DPLL contains two main steps: first, it assigns a value to a variable at each step (decision), simplifying the formula and then it checks, whether the remaining formula is satisfiable. Due to the decision steps,

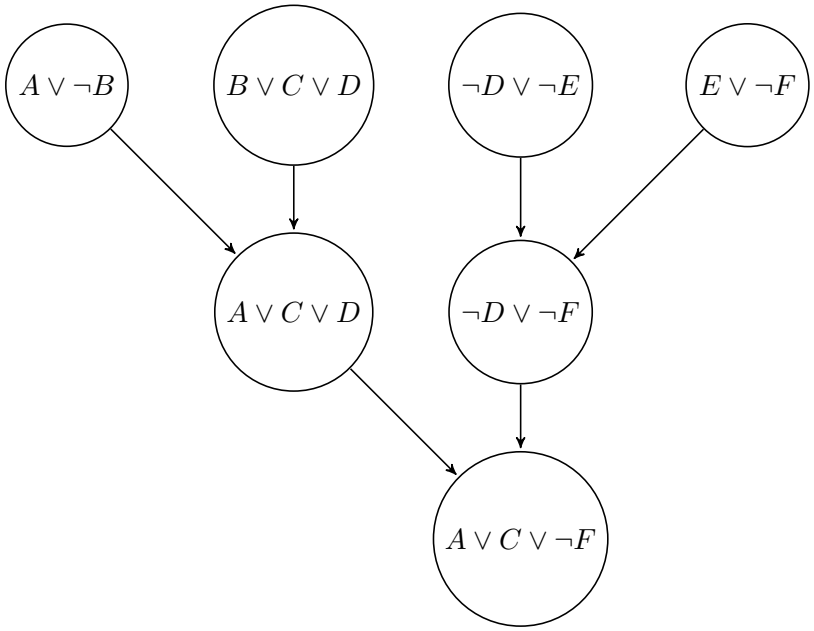


Figure 2.1: Resolution graph

a set of literals M will be built up during the process. In each iteration, the algorithm reaches a new decision level. During the simplification, *unit propagation* will be executed for all unit clauses: the unassigned literal gets a truth value so that, the clause will be satisfied. If there are two unit clauses with the same unassigned variable $X$ but with different polarity ($X$ and $\neg X$), the problem is unsatisfiable with the current M (based on the two unit clauses we should add $X$ and $\neg X$ to M). Within the simplification, the clauses (which are true at the actual assignment) and the false literals will be removed from the formula.
The classical DPLL algorithm consists of the following five rules (C is a clause, $l$ is a literal, $l^d$ indicates that the literal appears in M through a decision). M $\parallel \varphi$ means that our partial assignment is M for the formula $\varphi$, M $l$ indicates that the set M has been expanded with literal $l$, A $\Longrightarrow$ B indicates state change from A to B, if the conditions listed in the brackets are met [GS05, p.23]. In the following we provide a formal description of the five rules.

**Decide**

$$\text{M} \parallel \varphi \Longrightarrow \text{M } l^d \parallel \varphi \text{ if} \begin{cases} l \text{ or } \neg l \text{ occurs in } \varphi \\ l \text{ is undefined in M} \end{cases}$$

**Unit propagation**

$$\mathrm{M} \parallel \varphi, \mathrm{C}\backslash\{l\} \vee l \Longrightarrow \mathrm{M}\ l \parallel \varphi, \mathrm{C}\backslash\{l\} \vee l \text{ if } \begin{cases} l \in \mathrm{C} \\ \mathrm{M} \models \neg(\mathrm{C}\backslash\{l\}) \\ l \text{ is undefined in M} \end{cases}$$

**Pure Literal**

$$\mathrm{M} \parallel \varphi \Longrightarrow \mathrm{M}\ l \parallel \varphi \text{ if } \begin{cases} l \text{ occurs in some clause of } \varphi \\ \neg l \text{ does not occur in } \varphi \\ l \text{ is undefined in M} \end{cases}$$

**Fail**

$$\mathrm{M} \parallel \varphi, \mathrm{C} \Longrightarrow \text{fail if } \begin{cases} \mathrm{M} \models \neg\mathrm{C} \\ \mathrm{M} \text{ contains no decision literals} \end{cases}$$

**Backtrack**

$$\mathrm{M}\ l^d\ \mathrm{N} \parallel \varphi, \mathrm{C} \Longrightarrow \mathrm{M}\ \neg l \parallel \varphi, \mathrm{C} \text{ if } \begin{cases} \mathrm{M}\ l^d\ \mathrm{N} \models \neg\mathrm{C} \\ \mathrm{N} \text{ contains no decision literals} \end{cases}$$

Modern DPLL procedures may use Pure Literal rule as preprocessing and Backjump is applied instead of Backtracking.

**Backjump**

$$\mathrm{M}\ l^d\ \mathrm{N} \parallel \varphi, \mathrm{C} \Longrightarrow \mathrm{M}\ l' \parallel \varphi, \mathrm{C} \text{ if } \begin{cases} \mathrm{M}\ l^d\ \mathrm{N} \models \neg\mathrm{C}, \text{ and there is} \\ \text{some clause C' } \vee\ l' \text{ such that:} \\ \mathbf{A},\ \varphi,\ \mathrm{C} \models \mathrm{C'} \vee l' \text{ and } \mathrm{M} \models \neg\mathrm{C'}, \\ \mathbf{B},\ l' \text{ is undefined in M and} \\ \mathbf{C},\ l' \text{ or } \neg l' \text{ occurs in } \varphi \text{ or in M } l^d\ \mathrm{N} \end{cases}$$

Algorithm 2.1 shows the pseudocode of the DPLL algorithm with unit propagation, in which $\varphi|X_i$ ($\varphi|\neg X_i$) means, variable $X_i$ has been assigned with true (false) in CNF formula $\varphi$.

There are two points, where we can thoroughly modify the algorithm: at branching and backtracking. The *branching rule* defines the next literal to be appended to the set M.

**Algorithm 2.1:** DPLL

**Input**: A CNF problem $\varphi$
**Output**: A boolean value whether the problem is satisfiable

```
1  while φ includes a clause C such that |C| ≤ 1, for each C do
2  |  if C = ∅ then return false;
3  |  else if C = {X_i} then
4  |  |  φ| X_i
5  end
6  if φ = ∅ then
7  |  return true
8  end
9  Choose a literal X_j using a branching rule;
10 if DPLL(φ|X_j) = true then
11 |  return true
12 end
13 if DPLL(φ|¬X_j) = true then
14 |  return true
15 end
16 return false
```

For example, with **Dynamic Largest Individual Sum (DLIS)** we get those literals with those forms (positive/negative), of which occurrence (number of clauses) is the largest. In case **Dynamic Largest Combined Sum (DLCS)** we are interested in the maximum occurrence of both positive and negative forms.
The other point is the backtracking: after a decision, the algorithm can simplify the formula, such as the satisfied clauses; and false literals will be removed. If the algorithm reaches a point, where a clause is false with the actual assignments (for example, after a unit propagation), it has to change a decision from the previous ones. The backtracking defines the decision level, by which the latter decisions will be cancelled. There are some ways to do it, we mention two examples [GS05, p.27].
**Chronological backtracking:** simplest mode, the algorithm steps always back to the previous decision level.
**Conflict-driven clause** learning: whenever the algorithm reaches a point, when the formula is unsatisfiable with the current assignment, algorithm produces a new clause involving the literals, which are affected in the conflict. This clause called conflict clause.
**Backjump:** similar to the conflict-driven clause learning, it supports adding new unit literals on a lower decision level, but it does not attach clause to the formula.
The following example is from the reference [NOT05], with addition of clause learning. We write in each step "SAT" (satisfiable) instead of the clause, if it is satisfiable; "UNIT" (indicates unit propagation), if the clause is not true with the actual assignments and there is just one literal in the clause, which does not have assignment; and UNSAT, if the clause is unsatisfiable. Note: the decisions follow the original example, we do not use

DLIS or DLCS branching rule.

**Example 1b** - DPLL with clause learning

| | | | | | |
|---|---|---|---|---|---|
| 1 $\varphi$ = | (¬A ∨ B) ∧ | (¬C ∨ D) ∧ | (¬E ∨ ¬F) ∧ | (F ∨ ¬E ∨ ¬B) | Decision: A |
| 2 $\varphi$ = | UNIT ∧ | (¬C ∨ D) ∧ | (¬E ∨ ¬F) ∧ | (F ∨ ¬E ∨ ¬B) | Unit propagation: B |
| 3 $\varphi$ = | SAT ∧ | (¬C ∨ D) ∧ | (¬E ∨ ¬F) ∧ | (F ∨ ¬E ∨ ¬B) | Decision: C |
| 4 $\varphi$ = | SAT ∧ | UNIT ∧ | (¬E ∨ ¬F) ∧ | (F ∨ ¬E ∨ ¬B) | Unit propagation: D |
| 5 $\varphi$ = | SAT ∧ | SAT ∧ | (¬E ∨ ¬F) ∧ | (F ∨ ¬E ∨ ¬B) | Decision: E |
| 6 $\varphi$ = | SAT ∧ | SAT ∧ | UNIT ∧ | UNIT | Unit propagation: ¬F |
| 7 $\varphi$ = | SAT ∧ | SAT ∧ | SAT ∧ | UNSAT | Clause learning: ¬A ∨ ¬E |

| Line | M |
|---|---|
| 1 | $\emptyset$ |
| 2 | $\{A^d\}$ |
| 3 | $\{A^d$,B$\}$ |
| 4 | $\{A^d$, B, $C^d\}$ |
| 5 | $\{A^d$, B, $C^d$, D$\}$ |
| 6 | $\{A^d$, B, $C^d$, D, $E^d\}$ |
| 7 | $\{A^d$, B, $C^d$, D, $E^d$, ¬F$\}$ |

It is obvious, that in line 6 we have two UNIT clauses, but we have to set F to false and to true, which means, the formula is unsatisfiable with our partial assignment. The learned clause is (¬A ∨ ¬E) and the algorithm tracks back to the decision level 1.
With an implication graph we can graphically present the decision levels and their logical connections. The following implication graph (Figure 2.2) illustrates the steps of the above demonstrated example. Each circle represents an assignment. The number in the circle indicates the decision level. If a circle does not have incoming edge, then this is a decision, otherwise it is a logical consequence of a decision, or a unit propagation.

On the graph we can see that a unit propagation (D) belongs to decision 2. The problematic unit propagations have red colour and it is also clear that the assignment of literal A and E are affected. With the help of the implication graph it is visible, why algorithm tracks back to the level 1 instead of level 2. This means, A ∧ E leads to incorrect assignment, so A and/or E must be false. In this way we get the learned clause: (¬A ∨ ¬E). If algorithm uses backjump, instead of clause learning, after conflict we get the set M = {A, B, ¬E}, without plus clause in the formula.
The algorithm will go back to decision level 1. In contrast to this, chronological backtracking simply goes one step back and modifies the decision about literal E to false but not remove the decision level 2 and its logical consequence. If the conflict remains, algorithm goes back to decision about C, and so on.

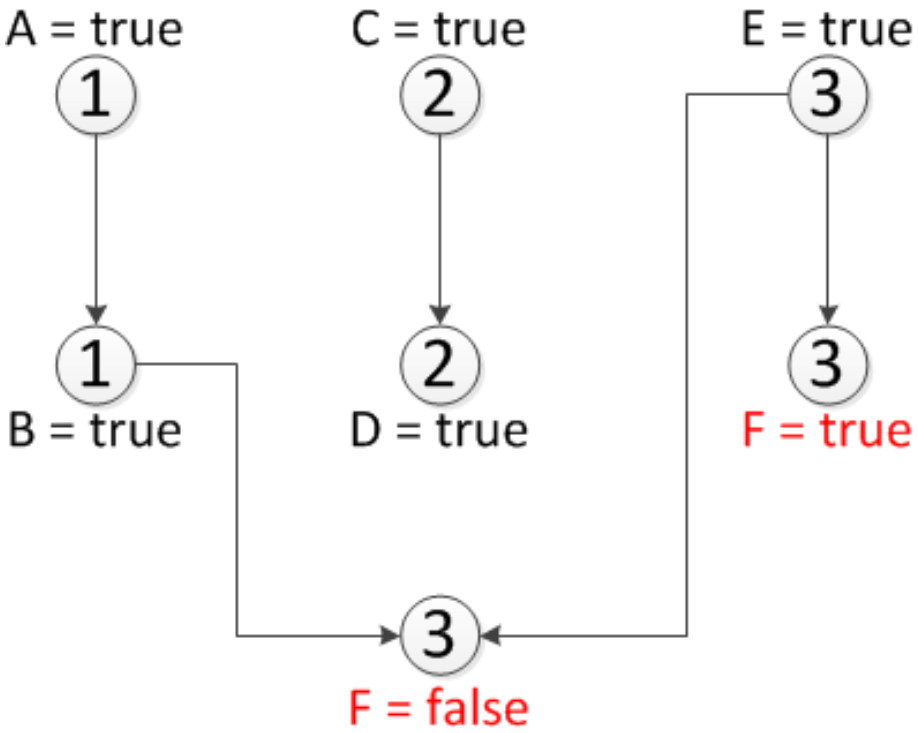


Figure 2.2: Implication graph

### 2.1.3 Parallelization options [HW13]

There are two main ways to parallelize a DPLL-based algorithm: **divide-and-conquer** and **parallel portfolio**. In the first case, the search space will be divided into sub spaces and distributed among sequential solvers. Load balancing transfers subspaces to idle solvers dynamically and a strategy to exchange learned clauses is needed. The parallel portfolio approach has been introduced in 2008 and it has become dominant. The essence of the approach is the multiple copies of the problem instance and the using of different sequential solvers. The idea is based on the complementarity of different sequential DPLL strategies. Each solver works on the whole original problem, so the issue around load balancing is resolved. The cooperation of the solvers based on the information exchange (learned clauses). The differences between the solvers can be in the strategy [PKA+06], restart policy, branching heuristic, conflict clause learning, etc. All presented algorithms use multi core CPU.
**Marijn J.H. Heule**, **Oliver Kullmann**, **Siert Wieringa** and **Armin Biere** introduced the **Cube and Conquer** [HKWB12] [vdTHB12]. The method combines two solver types to bring out the best in them. Conflict-driven clause learning (CDCL) solvers are appropriate to solve huge problems with help of learned clauses. By contrast, lookahead solvers use sophisticated heuristics to solve small hard problems. Cube and Conquer uses a lookahead solver to partition the search space into many cubes (a cube

mean a partial assignment). The switch from the lookahead to CDCL is based on two conditions. Let be D $\geq$ 0 a depth parameter. For all cubes, if exactly D decisions have been made, or the total number of assigned variables is at least D, the lookahead finishes the work. Than, a CDCL solver takes the cubes iteratively and try to solve the simplified problem.
**Nishant Totla** and **Aditya Devarakonda** describe a solver, which combines the divide-and-conquer and portfolio approaches [TD]. First, the search space will be divided into disjoint parts with the help of different true/false values on a small set of problem variables. On each part one-one process starts the work. Each process, however, is a parallel portfolio of multiple threads running independent solvers to solve the assigned part of the search space.
The portfolio based solvers are typically non-deterministic. This depends on the way they synchronize each other: the information (learned clauses) exchange between the parallel solvers is weak, because the attempt is to maximize the performance. **Youssef Hamadi**, **Said Jabbour**, **Cédric Piette** and **Lakhdar Saïs** made a **Deterministic Parallel DPLL** [HJPS11] algorithm which uses synchronization barriers for regular information exchange through a controlled environment. The frequency of exchanges affects the performance of the solvers: frequent synchronizations increase the speed of learning new foreign clauses but they are computationally expensive. Less frequent synchronization leads to delayed foreign conflict-clauses integration.

## 2.2 CPU and GPU parallelization [Chi12] [Das11] [cud]

A CPU typically consists of 4-8 cores with lots of cache memory. Each core contains an arithmetic logic unit (ALU); and the task of the cores is to handle different software threads at a given time. A modern CPU offers many features like interrupts and virtual memory, which are important to serve the modern operating systems.
The GPU has significantly different architecture that makes it better suited to different tasks (Figure 2.3). The task of these devices is to process large amounts of data in many threads. GPUs were initially used for graphical tasks - for example video processing, image manipulating and 3D rendering. In recent years, the capabilities of the GPU have been used for accelerating computational workloads in areas such as financial modelling, cutting-edge scientific research and oil and gas exploration.

### 2.2.1 The CUDA programming model

Now, we give an overview about GPU architecture and programming model. A GPU contains a scalable array of Streaming Multiprocessors (SM). Each SM consists of around 100-200 Streaming Processor cores (SP). In each SP, there is an ALU and a floating point unit (FPU). The GPU, which is used for testing in this thesis, is based on the first generation of Maxwell architecture; it has 4 SMs and 128 SPs per SM (total of 512 SPs).
The Compute Unified Device Architecture (CUDA) is an Application Programming Interface (API), provided by NVIDIA. With a set of special keywords and language

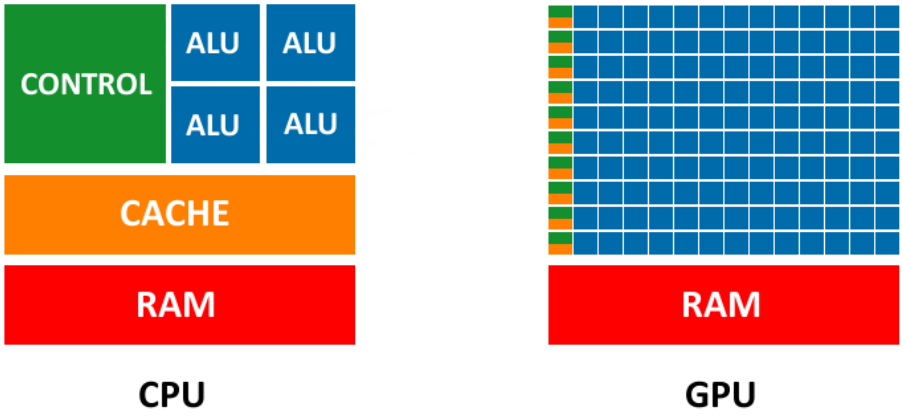


Figure 2.3: Differences between CPU and GPU

primitives, we can send tasks to a CUDA capable GPU from our program, written in C, C++ or FORTRAN languages. Assembly language is not required. The program starts the execution on the CPU similarly to any other programs. The code, which will be executed on the GPU, can be defined in a *kernel* function. In this context, CPU is called as *host*, the GPU as *device*. When the program calls a kernel function, one-, two-, or three-dimensional blocks of threads execute the code on the GPU. Blocks are organized into a one-, two-, or three-dimensional grid of thread blocks. 32 threads within a block will be arranged into a *warp*. Each warp will be executed by an SM. An execution of a warp will be finished, when all threads of the warp are terminated. Therefore, the divergence of the threads execution is caused by branches (if-else, switch-case); and loops (while) degrades the performance of the parallel execution.
As Figure 2.4 shows, in the GPU we have three different memories: global, per-block shared memory and per-thread memory. The global memory has the biggest capacity and its performance is the slowest. The host copies the data for processing into this kind of memory, from where it copies the results back into the main memory after the execution of the kernel. Very important is the principle of locality: the reuse of specific data within a relatively small time duration or data elements within relatively close storage locations leads to optimized memory using and faster execution. The size of the global memory is typically 1-12 GB. Each SM has a per-block shared memory. This memory is partitioned among all threads. Therefore, using a large number of registers per thread will limit the number of threads that can run concurrently. The limit of memory usage is 48 KB per warp, and NVIDIA recommends the maximum usage of 32 KB. The size of this memory is 96 KB in case of the first Maxwell generation. Finally, each thread has its own local memory, which is not accessible by other threads.
In general, it is very difficult to implement an algorithm to the GPU effectively. If we

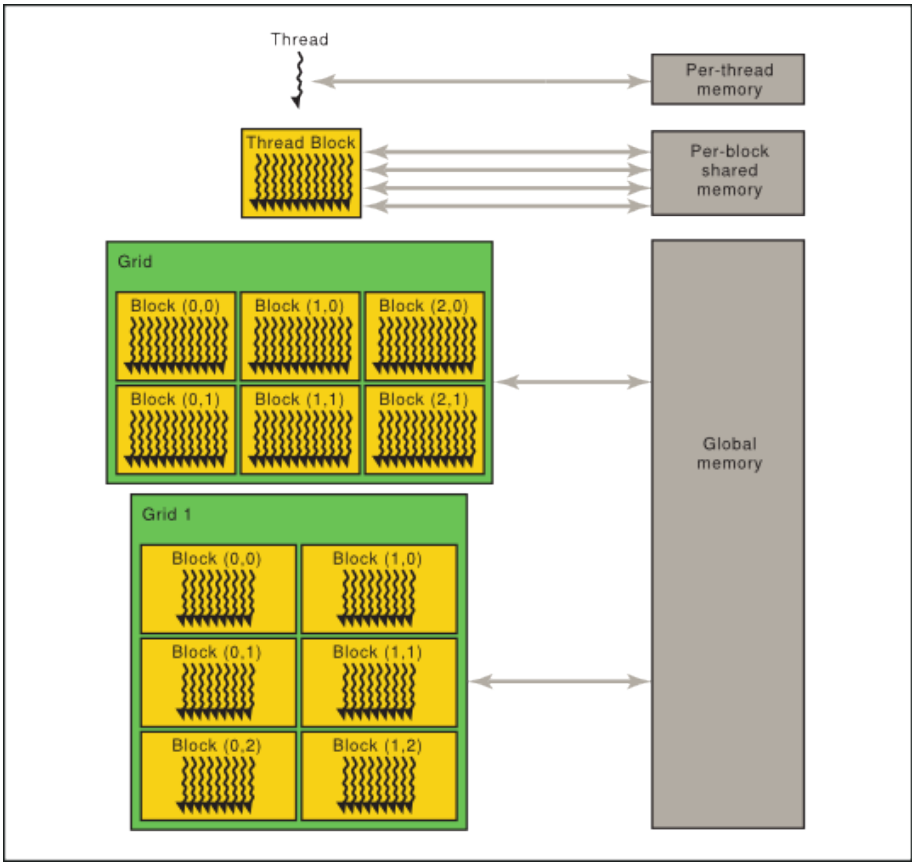


Figure 2.4: GPU memory hierarchy

consider the described DPLL, we found a lot of difficulties. The algorithm itself fits better to the CPU due to the complexity. On the one hand, we have to deal with the load balance in the case of divide-and-conquer. And the information exchange has to be highlighted in the case of portfolio solvers, on the other hand. A current middle class NVIDIA GPU has 512-640 ALUs. The high number of ALUs raises the question, when and how to manage load balance and information exchange.
The new formulation and approach try to gain an advantage at this point: the algorithm might be probably not the most effective in area of memory consumption or complexity, but at least it can utilize the extreme performance of the GPUs, and it might probably offer an other interesting option, as well. In the next chapter, we introduce our new approach, the Hobel algorithm.

CHAPTER 3

# The Hobel

## 3.1 The basic algorithm

As previously mentioned, Hobel is fundamentally different to the DPLL. DPLL selects a variable, a truth value to it and considers the consequences. If there is not any conflicts, it selects the next variable. In case of unsatisfiability, some decisions will be revoked and the algorithm makes other decision(s). The goal is to satisfy as many undefined clauses - at each step - as possible. In addition, Hobel uses a different strategy: it goes not from variable, but from clause to clause. At each clause, it determines the partial assignments which evaluates the clause to true. Then, it considers the partial assignments of the already processed clauses and searches for compatibility. At the end, if the problem is satisfiable, the algorithm will have all possible models to it.
First, we introduce the **Hobel Normal Form (HNF)**, which is the structure for carrying information about the input problem. We write either upper-case letters or numbers.
**Definition 6 (HNF cube):** a HNF cube $hc$ is a conjunction of literals: $\bigwedge_i l_i$.
**Definition 7 (HNF formula):** a HNF formula $hf$ is an exclusive disjunction of HNF cubes: $\bigoplus_j hc_j$

According to the concept description, each CNF clause has a HNF formula. Each HNF formula describes the partial assignments which evaluate the clause to true. See the Example 2.

**Example 2** – A CNF clause and its HNF formula

| | |
|---|---|
| input CNF clause | (A $\vee$ $\neg$B $\vee$ C) |
| HNF formula | A$\neg$B$\neg$C $\oplus$ C$\neg$A$\neg$B $\oplus$ AB$\neg$C $\oplus$ AC$\neg$B $\oplus$ BC$\neg$A $\oplus$ ABC $\oplus$ $\neg$A$\neg$B$\neg$C |

**Example 3** – A CNF formula and its HNF formulas

Input CNF problem: $(A \vee \neg B \vee C) \wedge (B \vee D) \wedge (\neg B \vee E \vee \neg A) \wedge (\neg C \vee \neg A \vee \neg E)$

HNF formulas:

| | |
|---|---|
| 1. | $A\neg B\neg C \oplus \neg A\neg BC \oplus AB\neg C \oplus A\neg BC \oplus \neg ABC \oplus ABC \oplus \neg A\neg B\neg C$ |
| 2. | $B\neg D \oplus \neg BD \oplus BD$ |
| 3. | $\neg A\neg BE \oplus EA\neg B \oplus \neg AEB \oplus EAB \oplus A\neg B\neg E \oplus \neg AB\neg E \oplus \neg A\neg B\neg E$ |
| 4. | $A\neg C\neg E \oplus \neg AC\neg E \oplus \neg A\neg CE \oplus AC\neg E \oplus A\neg CE \oplus \neg ACE \oplus \neg A\neg C\neg E$ |

The cubes describe different partial assignments, therefore we can not use two or more of them at the same time. As we want to satisfy the corresponding CNF clause, we have to choose exactly one cube. The main goal of the algorithm is to find one cube from each HNF formulas so, that there would not be any conflicts between them (for example, one of the cubes defines literal A as positive, and another as negative). Instead of search for a good set, our method exploits the independency between the cubes by merging in parallel all the possible partial assignment (see formal description on the next page). On one hand, the GPU can be used to perform a relevant parallel task and no backstep needed (after each clause we have all possible assignments which evaluate the already processed clauses to true). On the other hand, the number of collected cubes heavily depends on the processed clauses. See the Example 4 and 5.

**Example 4** - Merging without common variables

| | |
|---|---|
| 2. | $B\neg D \oplus \neg BD \oplus BD$ |
| 4. | $A\neg C\neg E \oplus \neg AC\neg E \oplus \neg A\neg CE \oplus AC\neg E \oplus A\neg CE \oplus \neg ACE \oplus \neg A\neg C\neg E$ |
| output 2-4 | $BA\neg C\neg D\neg E \oplus BC\neg A\neg D\neg E \oplus BE\neg A\neg C\neg D$ $\oplus BAC\neg D\neg E \oplus BAE\neg C\neg D \oplus BCE\neg A\neg D$ $\oplus B\neg A\neg C\neg D\neg E \oplus DA\neg B\neg C\neg E \oplus DC\neg A\neg B\neg E \oplus DE\neg A\neg B\neg C$ $\oplus ACD\neg B\neg E \oplus ADE\neg B\neg C \oplus DCE\neg A\neg B \oplus D\neg A\neg B\neg C\neg E$ $\oplus BDA\neg C\neg E \oplus BDC\neg A\neg E \oplus BDE\neg A\neg C \oplus BDAC\neg E$ $\oplus BDAE\neg C \oplus BDCE\neg A \oplus BD\neg A\neg C\neg E$ |

**Example 5** - Merging with common variables

| | |
|---|---|
| 1. | $A\neg B\neg C \oplus \neg A\neg BC \oplus AB\neg C \oplus A\neg BC \oplus \neg ABC \oplus ABC \oplus \neg A\neg B\neg C$ |
| 3. | $\neg A\neg BE \oplus EA\neg B \oplus \neg AEB \oplus EAB \oplus A\neg B\neg E \oplus \neg AB\neg E \oplus \neg A\neg B\neg E$ |
| output 1-3 | $A\neg B\neg CE \oplus A\neg B\neg C\neg E \oplus \neg A\neg BCE \oplus$ $\neg A\neg BC\neg E \oplus AB\neg CE \oplus A\neg BCE \oplus A\neg BC\neg E \oplus$ $\neg ABCE \oplus \neg ABC\neg E \oplus ABCE \oplus \neg A\neg B\neg CE \oplus \neg A\neg B\neg C\neg E$ |

If the variables in the problem have low incidence, the CNF clauses have less common variables and the number of cubes grows faster. For example, in the case of $\bigwedge_{i=1}^{N} (a_i \vee b_i)$

**Algorithm 3.1:** HNF Merging

```
   Input: Set of HNF formulas (SetOfHNFs)
   Output: A HNF formula (list of models to the problem)
 1 repeat
 2 |   FormulaA = SetOfHNFs[0];
 3 |   FormulaB = SetOfHNFs[1];
 4 |   Removing Formulas at positions 1 and 0 in SetOfHNFs;
 5 |   FormulaOut = new HNF Formula;
 6 |   forall the cube c_A ∈ FormulaA do
 7 |   |   forall the cube c_B ∈ FormulaB do
 8 |   |   |   cube cOut = new cube;
 9 |   |   |   forall the literal l_A ∈ c_A do
10 |   |   |   |   if l_A ∈ c_B then
11 |   |   |   |   |   Append l_A to cOut;
12 |   |   |   |   else
13 |   |   |   |   |   if ¬l_A ∈ c_B then
14 |   |   |   |   |   |   goto next c_B;
15 |   |   |   |   |   else
16 |   |   |   |   |   |   Append l_A to cOut;
17 |   |   |   |   |   end
18 |   |   |   |   end
19 |   |   |   end
20 |   |   |   forall the literal l_B ∈ cube c_B do
21 |   |   |   |   if l_B ∉ c_A then
22 |   |   |   |   |   Append l_B to cOut;
23 |   |   |   end
24 |   |   |   if |cOut| > 0 then
25 |   |   |   |   Append cOut to FormulaOut;
26 |   |   end
27 |   end
28 |   if |FormulaOut| > 1 then
29 |   |   Append FormulaOut to SetOfHNFs;
30 until there are more than 1 HNFs;
31 return SetOfHNFs;
```

we have $2^N$ solutions and therefore the size of the cubes grows exponential from clause to clause. At each merging task, the number of common variables of the formulas is inversely related to the output length. Therefore, preferring HNF formulas with many common variables is a critical task. In the next lines we introduce the *content list*, which has two main goals: first, it shortens the formula length, because we do not need to list negative literals in each cube. Secondly, it makes easier to find HNF formulas with many common variables.

**Definition 8 (Content list):** given a HNF formula $hf$. The content list $cl$ of the $hf$ is a conjunction of positive literals. For every HNF cube $hc \in hf$, if the literal $l \notin hc$ and $l \in cl$, then $\neg l \in hc$.

After we remove the negative literals from the cubes, the HNF cube would be a conjunction of positive literals. If a cube does not contain any literals, we label this cube as "empty". Examples 2b, 3b, 4b and 5b demonstrate the difference.

**Example 2b** – A CNF clause and its HNF formula

| | | |
|---|---|---|
| input CNF clause | (A ∨ ¬B ∨ C) | |
| HNF formula | A ⊕ C ⊕ AB ⊕ AC ⊕ BC ⊕ ABC ⊕ empty | content: ABC |

**Example 3b** – A CNF formula and its HNF formulas

Input CNF problem: (A ∨ ¬B ∨ C) ∧ (B ∨ D) ∧ (¬B ∨ E ∨ ¬A) ∧ (¬C ∨ ¬A ∨ ¬E)
HNF formulas:

| | | |
|---|---|---|
| 1. | A ⊕ C ⊕ AB ⊕ AC ⊕ BC ⊕ ABC ⊕ empty | content: ABC |
| 2. | B ⊕ D ⊕ BD | content: BD |
| 3. | E ⊕ EA ⊕ EB ⊕ EAB ⊕ A ⊕ B ⊕ empty | content: ABE |
| 4. | A ⊕ C ⊕ E ⊕ AC ⊕ AE ⊕ CE ⊕ empty | content: ACE |

**Example 4b** - Merging without common variables

| | | |
|---|---|---|
| 2. | B ⊕ D ⊕ BD | content: BD |
| 4. | A ⊕ C ⊕ E ⊕ AC ⊕ AE ⊕ CE ⊕ empty | content: ACE |
| output 2-4 | BA ⊕ BC ⊕ BE ⊕ BAC ⊕ BAE ⊕ BCE ⊕ B ⊕ DA ⊕ DC ⊕ DE ⊕ ACD ⊕ ADE ⊕ DCE ⊕ D ⊕ BDA ⊕ BDC ⊕ BDE ⊕ BDAC ⊕ BDAE ⊕ BDCE ⊕ BD | content: BDACE |

**Example 5b** - Merging with common variables

| | | |
|---|---|---|
| 1. | A ⊕ C ⊕ AB ⊕ AC ⊕ BC ⊕ ABC ⊕ empty | content: ABC |
| 3. | E ⊕ EA ⊕ EB ⊕ EAB ⊕ A ⊕ B ⊕ empty | content: ABE |
| output 1-3 | AE ⊕ A ⊕ CE ⊕ C ⊕ ABE ⊕ ACE ⊕ AC ⊕ BCE ⊕ BC ⊕ ABCE ⊕ E ⊕ empty | content: ABCE |

The main step of the algorithm is to repeat merging until only one HNF formula remains. If we merge the output of the Example 4b and 5b, the new formula will list all possible solutions to the input problem. Consequently, the content list will contain the variables of the original problem:

BC ⊕ BAE ⊕ BCE ⊕ DA ⊕ DC ⊕ DE ⊕ ACD ⊕
ADE ⊕ DCE ⊕ D ⊕ BCD ⊕ BDAE ⊕ BDCE, content: ABCDE

**CNF-HNF conversion:** any non-empty CNF clause $c$ is convertible into HNF formula with the following rules: Let be $L$ the set of the literals of $c$, $L^+$ the set of positive literals of $L$, $L^-$ the set of negative literals of $L$: $L = L^+ \cup L^-$.

1. IF $c$ contains at least one positive and at least one negative literal:
   a) creating HNF cubes from each non-empty subset of $L^+$ with all possible subset of $L^-$. $\forall(s \subset L^+, t \subset L^-) : s \cup t, |s| > 0$
   b) creating HNF cubes from each non-full subset of $L^-$. $\forall s \subset L^-, |s| \neq |L^-|$
2. IF $c$ consists of only negative literals:
   a) creating HNF cubes from each non-full subset of $L^-$. $\forall s \subset L^- : |s| \neq |L^-|$.
3. IF $c$ consists of only positive literals:
   a) creating HNF cubes from each non-empty subset of $L^+$. $\forall s \subset L^+ : |s| \neq 0$.

The content list consists of all variables of the CNF clause.
Justification: a HNF formula describes the possible solutions to a CNF clause. Each non-empty CNF clause has at least one solution, so each CNF clause has at least one corresponding HNF formula. The other way around is not true: the easiest way to prove it is the CNF clause from the Example 2. It has two positive literals. It is clear that every possible subset of the positive literals is solution to the clause. If we remove all cubes with one literal from its HNF formula, we can not find an appropriate CNF clause to the new formula: such a CNF clause does not exist, for which any two positive literals together are models, but one by one not.
We prove the correctness of the conversion with indirect proof: we suppose that the conversion does not describe all models to a CNF clause.
Case 1: the first point the case is excluded, when the positive literals are false with any

subset of $L^-$. These cases are covered by point two, except when all negative literals are true. This is obviously not a model to the clause (all positive literals are false and all negative are true). Since there is exactly one assignment, which does not satisfy the clause, all other assignments are models - and have been covered.
Case 2 and 3 are evident: if we have only negative literals, at least one has to be false. In case of positive literals, at least one has to be true.

The formal description of the merging is the following: let be $FormulaA$ and $FormulaB$ the two HNF formulas, $cl_A$ ($cl_B$) is the content list of the $FormulaA$ ($FormulaB$). Merging has to be performed on cube-pairs (one cube from each formula). $FormulaA$ contains M, $FormulaB$ N cubes. $FormulaOUT$ is the output HNF formula of the merging task, $nb$ is the output cube of a cube-pair, $cl_{OUT}$ is the content list of $FormulaOUT$.
Merging: for all cube $c_i \in FormulaA$ and $c_j \in FormulaB$ (i = 1...M, j = 1...N):

1. IF $cl_A \cap cl_B = \emptyset$: $nb = c_i \cup c_j$.
2. IF $cl_A \cap cl_B \neq \emptyset$:
   Criteria 1: $\forall$ v, v $\in cl_A$ and v $\in cl_B$:
   if the variable v is true (false) in the $c_i$, it has to be true (false) in $c_j$.
   If Criteria 1 has not been satisfied, there is no output cube. Otherwise: $nb = c_i \cup c_j$.
3. $cl_{OUT} = cl_A \cup cl_B$.

Algorithm 3.2 demonstrates the formal description of the algorithm.
The merging of the HNF formulas is independent from each other and therefore they can be performed parallel. If we use a variant of parallel DPLL algorithm, we will have a lot of problems regarding decomposition of the search space/problem instance and sharing learned clauses between the threads. Hobel does not use any search algorithm and therefore it does not have learned clauses. The threads always work on independent „local problems“. It does not need to handle search tree and backtracking: merging of two formulas is always possible. Consequently, the main phase of the algorithm contains exactly N-1 merging tasks at the start, if N is the number of clauses in the original CNF problem. Since there is no dependency between the merging operations, the algorithm fits to GPU computing and this is the most significant usefulness as opposed to other SAT solvers.
As initial step, we have to convert the CNF into HNF. Therefore we lost time and we need extra computation. We suppose that perhaps there are rules, with which we can transform boolean expressions direct into HNF. But we have to mention that the biggest challenge is to solve the problem itself. If we can solve the problem with this formulation significantly faster than with other solvers, the lost computation capacity will not play notable role. In addition, the conversion can be calculated in parallel.
The main problem with the algorithm is the exponential number of cubes. Due to this problem, the algorithm in this form is unusable in the practice. The question is that,

**Algorithm 3.2:** HNF Merging with content list

```
   Input: Set of HNF formulas (SetOfHNFs)
   Output: A HNF formula (list of models to the problem)
 1 repeat
 2 |   FormulaA = SetOfHNFs[0];
 3 |   FormulaB = SetOfHNFs[1];
 4 |   Removing Formulas at positions 1 and 0 in SetOfHNFs;
 5 |   FormulaOut = new HNF Formula;
 6 |   ContentOut = new ContentList;
 7 |   forall the cube c_A ∈ FormulaA do
 8 |   |   forall the cube c_B ∈ FormulaB do
 9 |   |   |   cube cOut = new cube;
10 |   |   |   forall the literal l_A ∈ c_A do
11 |   |   |   |   if l_A ∈ FormulaBContent then
12 |   |   |   |   |   if l_A = l_B then
13 |   |   |   |   |   |   Append l_A to cOut;
14 |   |   |   |   |   else
15 |   |   |   |   |   |   goto next c_B;
16 |   |   |   |   |   end
17 |   |   |   |   else
18 |   |   |   |   |   Append l_A to cOut;
19 |   |   |   |   end
20 |   |   |   end
21 |   |   |   forall the literal l_B ∈ cube c_B do
22 |   |   |   |   if l_B ∉ FormulaAContent then
23 |   |   |   |   |   Append l_B to cOut;
24 |   |   |   end
25 |   |   |   if |cOut| > 0 then
26 |   |   |   |   Append cOut to FormulaOut;
27 |   |   end
28 |   end
29 |   forall the literal l_A ∈ FormulaAContent do
30 |   |   if l_A ∉ ContentOut then
31 |   |   |   Append l_A to ContentOut;
32 |   end
33 |   forall the literal l_B ∈ FormulaBContent do
34 |   |   if l_B ∉ ContentOut then
35 |   |   |   Append l_B to ContentOut;
36 |   end
37 |   if |FormulaOut| > 1 then
38 |   |   Append FormulaOut to SetOfHNFs;
39 until there are more than 1 HNFs;
40 return SetOfHNFs;
```

whether there are any improvement in the procedure which could significantly decrease the memory consumption and transform the algorithm into usable solution.
The first approach we have considered is to prefer the merging of similar HNF formulas. "Similar" means the higher number of common variables in the content lists. If formula A and B do not have common variables and the number of cubes are M and N, the output formula will contains M * N cubes. Merging similar formulas leads to the execution of line 14 - the algorithm will generate less cubes. This modification can decrease the number of cubes, but can not solve the problem itself.
**Definition 9 (similar formula):** a HNF formula FormulaC is more similar to FormulaA than FormulaB, if $|cl_C \cap cl_A| > |cl_B \cap cl_A|$.

The second heuristic we have considered is to look at the other CNF clauses during merging. A CNF clause c has $|c|^2$ possible truth assignments, from this $\sum_{k=1}^{|c|} \binom{|c|}{k}$ models and exactly one not (the case when all positive literals become false and all negative literals become true). The algorithm focuses on the last case: each CNF clause specifies a partial assignment which is not good for it (the clause will be unsatisfied). During the merging, the algorithm checks these partial assignments. The output formula will not contain such a cube, which has a conflict with any CNF clauses. In this respect, we call the clauses **CNF rules** , because we want to highlight the logical meaning of the clauses from the Hobel's point of view. We have three different CNF rules based on the different types of CNF clauses (there is only positive/negative literals, or both).

**Example 6** - Different CNF clauses and the corresponding CNF rules

| CNF clause | non-model information | rule type | rule representation |
|---|---|---|---|
| (A ∨ ¬B ∨ C) | "If B is true, A or C must be true" | Effect | B → A,C |
| (B ∨ D) | "B or D must be true" | MINset | B,D |
| (¬C ∨ ¬A ∨ ¬E) | "A, C and E can not be together" | Exclusion | !ACD |

**Definition 10 (condition, effect part)**: in case of Effect rule, the set of negative literals $L^-$ forms the *Condition part* (in rule representation to left from the arrow), and the set of positive literals $L^+$ forms the *Effect part* of the rule (to right from the arrow). L = $L^+ \cup L^-$.

An experiment has been performed earlier to solve the problem with these rules. According to the main idea, a concise description should have been developed to the bad assignments, which could be modified from rule to rule. At the end, the algorithm could generate the models based on the description. The development of such a description was unsuccessful. We also introduced some rules with which the algorithm can exclude cubes and formulas. We call them **Hobel rules**:

1. If we have two CNF rules $c_i$, $c_j$ with MINset type and $c_i \subset c_j$, we can remove the CNF clause $c_j$ from the problem.

Example: MINset(A,B) and MINset(A,B,C). The first rule says, that A or B must be true. This rule will satisfy the second rule, so the second one is unnecessary.

2. If we have two CNF rules $c_i$, $c_j$ with Exclusion type and $c_i \subset c_j$, we can remove the CNF clause $c_j$ from the problem.
   Example: !AB and !ABC. The first rule says, that A and B can not be together. This one and the second rule will exclude all cases, so we can remove the appropriate CNF clause from the problem.

3. If we have two CNF rules $c_i$, $c_j$ with Effect type, and the Condition part of $c_j$ is equivalent with the Effect part of the $c_i$, we can remove the CNF clause $c_j$ from the problem and we have to remove the corresponding HNF cube (condition part of $c_j$) from the merged HNF formula.
   $c_i : A...M \rightarrow X$
   $c_j : X \rightarrow A, ..., N$

4. Let be $c_v$ the set of occurring variables in clause c and FormulaA a HNF formula with content list $cl_A$. If $c_v \subset cl_A$, we can accept the CNF rule of c to exclude invalid partial assignments from FormulaA.

With the above described rules the algorithm can exclude HNF formulas at preprocessing (rules 1, 2, 3) and HNF cubes during the process (rule 4).

**Example 7** – Using Hobel and CNF rules
input CNF problem:
(A ∨ ¬B ∨ C)∧(B ∨ D)∧(¬B ∨ E ∨ ¬A)∧(¬C ∨ ¬A ∨ ¬E)∧(B ∨ C ∨ D)∧(A ∨ B ∨ ¬E)∧(¬C ∨ ¬A ∨ ¬E ∨ ¬B)∧(B ∨ A ∨ D)

**HNF formulas:**

| | | |
|---|---|---|
| 1. | A ⊕ C ⊕ AB ⊕ AC ⊕ BC ⊕ ABC ⊕ empty | content: ABC |
| 2. | B ⊕ D ⊕ BD | content: BD |
| 3. | E ⊕ EA ⊕ EB ⊕ EAB ⊕ A ⊕ B ⊕ empty | content: ABE |
| 4. | A ⊕ C ⊕ E ⊕ AC ⊕ AE ⊕ CE ⊕ empty | content: ACE |
| 5. | B ⊕ C ⊕ D ⊕ BC ⊕ BD ⊕ CD ⊕ BCD | content: BCD |
| 6. | A ⊕ B ⊕ AB ⊕ AE ⊕ BE ⊕ ABE ⊕ empty | content: ABE |
| 7. | C ⊕ A ⊕ E ⊕ B ⊕ CA ⊕ CE ⊕ CB ⊕ AE ⊕ AB ⊕ EB ⊕ CAE ⊕ CAB ⊕ CEB ⊕ AEB ⊕ empty | content: ABCE |
| 8. | B ⊕ AB ⊕ BD ⊕ ABD ⊕ D ⊕ AD ⊕ empty | content: ABD |

**CNF Rules:**

1: B → A,C  2: MINset(B,D)  3: AB → E  4: !ACE
5: MINset(B,D,E)  6: E → A,B  7: !ABCE  8: A → B,D

**New HNF formula set after preprocessing:**

| | | |
|---|---|---|
| 1. | A ⊕ C ⊕ AB ⊕ AC ⊕ BC ⊕ ABC ⊕ empty | content: ABC |
| 2. | B ⊕ D ⊕ BD | content: BD |
| 3. | E ⊕ EA ⊕ EB ⊕ EAB ⊕ A ⊕ B ⊕ empty | content: ABE |
| | (dropped out: Hobel rule 3) | |
| 4. | A ⊕ C ⊕ E ⊕ AC ⊕ AE ⊕ CE ⊕ empty | content: ACE |
| 5. | (dropped out: Hobel rule 1) | |
| 6. | (dropped out: Hobel rule 3) | |
| 7. | (dropped out: Hobel rule 2) | |
| 8. | B ⊕ AB ⊕ BD ⊕ ABD ⊕ D ⊕ AD ⊕ empty | content: ABD |
| 1-2: | AD ⊕ CD ⊕ AB ⊕ ABD ⊕ ACD ⊕ BC ⊕ BCD ⊕ ABC ⊕ ABCD ⊕ D | content: ABCD |
| 3-4: | AE ⊕ EB ⊕ EBC ⊕ EAB ⊕ A ⊕ AC ⊕ BC ⊕ B ⊕ C ⊕ empty | content: ABCE |
| | (dropped out: Hobel rule 4, CNF rule 1) | |
| 8. | B ⊕ AB ⊕ BD ⊕ ABD ⊕ D ⊕ AD ⊕ empty | content: ABD |
| 1-2-3-4: | ADE ⊕ AD ⊕ CD ⊕ ABE ⊕ ABDE ⊕ ACD ⊕ EBC ⊕ BC ⊕ BCDE ⊕ BCD ⊕ D | content: ABCDE |
| 8. | B ⊕ AB ⊕ BD ⊕ ABD ⊕ D ⊕ AD ⊕ empty | content: ABD |
| Models: | ADE ⊕ AD ⊕ CD ⊕ ABE ⊕ ABDE ⊕ ACD ⊕ EBC ⊕ BC ⊕ BCDE ⊕ BCD ⊕ D | |

Unfortunately, the practice demonstrated that the decrease of the number of cubes is not significant. The problem is opposed to the preference of similar formulas: the algorithm can not use a CNF rule to exclude cubes, if the formula does not contain all variables from the rule. For example, in the case of the first HNF formula we could exclude many cubes based on the third and eighth CNF rules. But the content list does not contain some variables and therefore the state of some variables from the rules is open question (variables A and D have not been specified by the formula). Preference of the similar formulas leads to shorter content lists but this hampers the usage of the CNF rules.
Another possible way to speed up the algorithm is the dividing of the formulas into sets of cubes (smaller tasks). The algorithm takes a set from each formula during the merging. There are two possibilities: after the processing of the last CNF clause, the output of the merging is not empty, or at a point, the merging leads to empty output. In the first case, the algorithm found model(s) to the problem. In the second case, the algorithm has to take the next set of the actual formula. If no other set is available, the process continues with the next set of the previous formula. We call this technique *branching*. The advantage of the branching is the potential avoid of unnecessary work: if one of the tasks leads to a model, the algorithm solves the problem significantly faster, than without branching. See the Example 8. The first formula has been divided into seven smaller tasks (each one contains one cube, represented with red color). All three tasks lead to different models and all other remained merging tasks are unnecessary.

**Example 8** - Solving with branching

| | | | |
|---|---|---|---|
| 1. | A ⊕ C ⊕ AB ⊕ AC ⊕ BC ⊕ ABC ⊕ empty | content: ABC | |
| 2. | B ⊕ D ⊕ BD | content: BD | ⟹ AD |
| 1-2. | AD | content: ABCD | |
| 3. | E ⊕ EA ⊕ EB ⊕ EAB ⊕ A ⊕ B ⊕ empty | content: ABE | ⟹ ADE, AD |
| 1-2-3. | ADE ⊕ AD | content: ABCDE | |
| 4. | A ⊕ C ⊕ E ⊕ AC ⊕ AE ⊕ CE ⊕ empty | content: ACE | ⟹ ADE, AD |
| 1. | A ⊕ C ⊕ AB ⊕ AC ⊕ BC ⊕ ABC ⊕ empty | content: ABC | |
| 2. | B ⊕ D ⊕ BD | content: BD | ⟹ CD |
| 1-2. | CD | content: ABCD | |
| 3. | E ⊕ EA ⊕ EB ⊕ EAB ⊕ A ⊕ B ⊕ empty | content: ABE | ⟹ CDE, CD |
| 1-2-3. | CD ⊕ CDE | content: ABCDE | |
| 4. | A ⊕ C ⊕ E ⊕ AC ⊕ AE ⊕ CE ⊕ empty | content: ACE | ⟹ CD, CDE |
| ... | | | |
| 1. | A ⊕ C ⊕ AB ⊕ AC ⊕ BC ⊕ ABC ⊕ empty | content: ABC | |
| 2. | B ⊕ D ⊕ BD | content: BD | ⟹ BC, BCD |
| 1-2. | BC ⊕ BCD | content: ABCD | |
| 3. | E ⊕ EA ⊕ EB ⊕ EAB ⊕ A ⊕ B ⊕ empty | content: ABE | ⟹ BCE, BC, BCDE, BCD |
| 1-2-3. | BC ⊕ BCD ⊕ BCE ⊕ BCDE | content: ABCDE | |
| 4. | A ⊕ C ⊕ E ⊕ AC ⊕ AE ⊕ CE ⊕ empty | content: ACE | ⟹ BC, BCD, BCE, BCDE |

The algorithm discovers the models sequentially. When the first group of models have been discovered, the algorithm can stop and return. Basically, the approach is more reasonable, because we do not need all possible solutions to a problem. But the main problem is still the waste of memory with the ineffective formulation.

## 3.2 MHNF

To solve the problem around the ineffective formulation, we created a new formulation. This is the **Minimal Hobel Normal Form (MHNF)**. First, we introduce a new

element in the Hobel formulation: the *optional element*. This is represented in the formulation as underlined variable (for example $\underline{A}$ in case of letters), or negative numbers. In fact, its truth assignment is not important in the corresponding cube (both truth values are accepted). Consequently, an MHNF cube can describe $2^N$ HNF cubes, where N is the number of optional elements in the cube. An MHNF cube has to list the optional elements, because the non-occurring elements from the content list will be automatically false – like in the case of HNF.

**Definition 11 (MHNF cube):** an MHNF cube mc is a conjunction of literals: $\bigwedge_i l_i$.
**Definition 12 (MHNF formula):** an MHNF formula mf is an exclusive disjunction of MHNF cubes: $\bigoplus_j mc_j$.
**Definition 13 (Optional element):** an optional element represents a variable and its negation. The optional element $\underline{l} \in$ mc defines two MHNF cubes: (mc $\setminus \underline{l}$) $\cup\, l$ and (mc $\setminus$ $\underline{l}$) $\cup\, \neg l$.
**Definition 14 (MHNF Content list):** given an MHNF formula mf. The content list $mc_l$ of mf is a conjunction of positive literals. It defines the negative literals for the MHNF cubes: $\forall$ mc $\in$ mf, if any literal $l \notin$ mc, $\underline{l} \notin$ mc and $l \in mc_l$, then $\neg$l $\in$ mc.

Example 2c shows the difference again plain HNF. The description of the possible assignments is much more compact.

**Example 2c** – A CNF clause and its HNF and MHNF formulas

| | | |
|---|---|---|
| input CNF clause | (A $\vee$ $\neg$B $\vee$ C) | |
| HNF formula | A $\oplus$ C $\oplus$ AB $\oplus$ AC $\oplus$ BC $\oplus$ ABC $\oplus$ empty | content: ABC |
| MHNF formula | $\underline{\text{C}} \oplus \text{A}\underline{\text{C}} \oplus \text{BC} \oplus \text{AB}\underline{\text{C}}$ | content: ABC |

In connection with CNF – MHNF conversion, we have to introduce some sets. Definitions 15-18 describe the *Hobel sets* which are necessary to transform the CNF clauses into MHNF formulas. Each Hobel set describes with MHNF cubes a set of full assignments with certain property. For example, the NONemptySet describes all possible assignments such that at least one of the input variables becomes true. The type of the CNF rule determines, which set should be used with which variables to get the correct MHNF formula (see page 26). One of the simplest case is the MINset: we have to use only the NONemptySet with all variables to get the appropriate MHNF formula to the CNF clause (at least one variable becomes true). Let be VIN the set of input variables, $s_{V+}$ ($s_{V-}$) the set of true (false) variables of the assignment s.

**Definition 15 (NONemptySet):** $\forall$ s: |VIN| $\neq |s_{V-}|$
**Definition 16 (NONfullSet):** $\forall$ s: |VIN| $\neq |s_{V+}|$
**Definition 17 (OptionalSet):** no restriction on the assignments
**Definition 18 (FullSet):** $\forall$ s: |VIN| $= |s_{V+}|$

Example 9 lists the different sets for an example input set of variables. Algorithm 3.3 and 3.4 show the algorithm of creation NONemptySet and NONfullSet.

**Algorithm 3.3:** MakeNonEmptySets

```
   Input: Set of variables VIN
   Output: MHNF formula
 1 Resultset = new MHNF formula;
 2 forall the variable v ∈ VIN do
 3 |   Subset = new set;
 4 |   forall the variable w ∈ VIN and w ≥ v do
 5 |   |   if v = w then
 6 |   |   |   Append +w to Subset;
 7 |   |   else
 8 |   |   |   Append -w to Subset;
 9 |   |   end
10 |   end
11 |   Append Subset to ResultSet;
12 end
13 return ResultSet;
```

**Algorithm 3.4:** MakeNonFullSets

```
   Input: Set of variables VIN
   Output: MHNF formula
 1 Resultset = new MHNF formula;
 2 forall the variable v ∈ VIN do
 3 |   Subset = new set;
 4 |   forall the variable w ∈ VIN do
 5 |   |   if v = w then
 6 |   |   |   goto next w;
 7 |   |   else if w > v then
 8 |   |   |   Append -w to Subset;
 9 |   |   else
10 |   |   |   Append +w to Subset;
11 |   |   end
12 |   end
13 |   Append Subset to ResultSet;
14 end
15 return ResultSet;
```

**Example 9** – Different sets for input variables A, B, C, D and E

| Set name | MHNF |
|---|---|
| NONemptySet | $\underline{\text{A}}\text{BCDE} \oplus \text{B}\underline{\text{CDE}} \oplus \text{C}\underline{\text{DE}} \oplus \text{D}\underline{\text{E}} \oplus \text{E}$ |
| NONfullSet | $\text{ABCD} \oplus \text{ABD}\underline{\text{E}} \oplus \text{AB}\underline{\text{CE}} \oplus \text{A}\underline{\text{CDE}} \oplus \underline{\text{BCDE}}$ |
| OptionalSet | $\underline{\text{ABCDE}}$ |
| FullSet | ABCDE |

To transform a CNF clause into MHNF, we have three cases. The set VIN is the set of the variables of the clause. The CNF clause consists of

- only negative literals (Exclusion): NONfullSet(VIN)
- only positive literals (MINset): NONemptySet(VIN)
- both negative and positive literals (Effect):
  1. $\forall$ mc $\in$ NONfullSet($L^-$): mc $\cup$ OptionalSet($L^+$), $L^- \cup L^+ =$ VIN
  2. $\forall$ mc $\in$ NONemptySet($L^+$): mc $\cup$ FullSet($L^-$), $L^- \cup L^+ =$ VIN

The goal of the formulation is to use the minimum number of literals to describe the models to a CNF clause. We use the CNF clause (A $\vee$ $\neg$B $\vee$ C) in the following example. On the Figure 3.1, all possible assignments are represented as a tree.

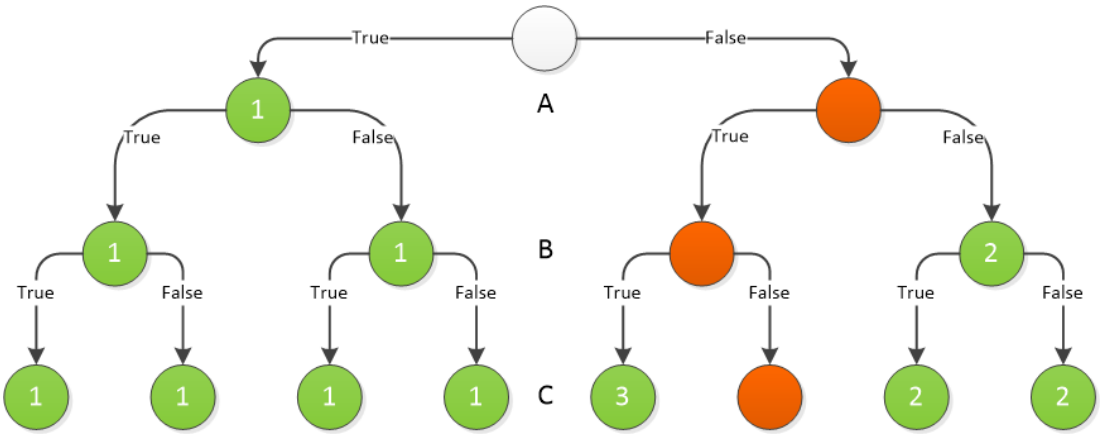


Figure 3.1: All possible assignments

As we already mentioned, a CNF clause has exactly one assignment, which falsifies the clause. This case is marked with red: A and C become false, B becomes true. The models of the clause form subtrees, correspondingly we have marked these with numbers. We can describe a subtree, if we specify the root node. For example, for the first subtree the

description is a single A - this variable has to be true, the truth values of the remained variables are irrelevant. The conjunction of the subtree descriptions forms the *minimal description* of the models. In an MHNF cube we list the true and the irrelevant variables. The MHNF cube can be deduced according to a subtree description, if we list the non-occurring variables as optional elements and remove the negative literals. Example 10 shows the logical connection between the minimal description and the MHNF.

**Example 10** - CNF – minimal description – MHNF

| CNF | minimal description | MHNF |
|---|---|---|
| (A $\vee$ $\neg$B $\vee$ C) | A $\vee$ $\neg$A$\neg$B $\vee$ $\neg$ABC | A$\underline{\text{BC}}$ $\oplus$ $\underline{\text{C}}$ $\oplus$ BC content: ABC |

The introduction of the optional element modifies the merging (Algorithm 3.5).

In practice, the variables are coded with integers, because the low number of letters would limit the problem size. In a MHNF cube, the positive literals are represented with positive, the optional elements with negative numbers. Let be $c_a$ ($c_b$) a cube from MHNF formula A (B), $cl_A$ ($cl_B$) the content list of formula A (B). The essence of the algorithm in words:

- if $c_a$ contains a variable $v$ as positive number (true), $c_b$ has to contain $v$ as positive or negative number, if $v \in cl_B$ (the variable will be true in the output because the restriction of $c_a$)
- if $c_a$ has a variable $v$ as negative number (optional) and $v \notin cl_B$, the variable will be negative in the output
- if $c_a$ has a variable $v$ as negative number and $v \in cl_B$:
  1. if $v$ is positive (true) in cube $c_b$, the variable will be positive in the output
  2. if $v$ is negative (optional) in $c_b$, the variable will be negative in the output

The variables in a cube are in ascending order based on the absolute values. When the algorithm merges two MHNF cubes, two indexes (A-, Bindex) designate one-one literal from each cube. At start, the indexes point to the first literals (line 5). The algorithm compares the designated literals based on the absolute values (lines 6-12). There are two cases:

- The absolute values are equal
- The absolute values are not equal

The variable *InvestigationMode* shows the result of the comparison. In the first case the algorithm found a variable which is contained by both cubes (InvestigationMode is 3).

**Algorithm 3.5:** MHNF Merging

```
   Input: Two MHNF formulas FormulaA and FormulaB
   Output: The merged MHNF formula
 1 OutputFormula = new MHNF formula;
 2 forall the cube c1 ∈ FormulaA do
 3     forall the cube c2 ∈ FormulaB do
 4         Outputcube = new MHNF cube;
 5         Aindex = 0, Bindex = 0, InvestigationMode = 0;
 6         if c1[Aindex] < c2[Bindex] then
 7             InvestigationMode = 1;
 8         else if c1[Aindex] > c2[Bindex] then
 9             InvestigationMode = 2;
10         else
11             InvestigationMode = 3;
12         end
13         switch InvestigationMode do
14             case 1
15                 if c1[Aindex] ∈ FormulaBContent then
16                     if c1[Aindex] > 0 then
17                         goto next c2;
18                     else
19                         Append c1[Aindex] to Outputcube;
20                     end
21                     IncreaseIndex();
22             case 2
23                 if c2[Bindex] ∈ FormulaAContent then
24                     if c2[Aindex] > 0 then
25                         goto next c2;
26                     else
27                         Append c2[Bindex] to Outputcube;
28                     end
29                     IncreaseIndex();
30             case 3
31                 if c1[Aindex] + c2[Bindex] >= 0 then
32                     Append +c1[Aindex] to Outputcube;
33                 else
34                     Append -c1[Aindex] to Outputcube;
35                 end
36                 IncreaseIndex();
37         endsw
38         Append Outputcube to OutputFormula;
39     end
40     Append Subset to ResultSet;
41 end
42 return OutputFormula;
```

In the second case, a variable occurs only in one of the cubes (InvestigationMode is 1 or 2). The function *IncreaseIndex* increases the index of the cube, in which the absolute value is lower (InvestigationIndex is 1 or 2), or both simultaneously (InvestigationIndex 3). Example 11 demonstrates an example. Red numbers show the state of the indexes.

**Example 11** - Index increasing and InvestigationMode during MHNF merge

c1 1356 InvestigationMode: 1
c2 346

c1 1356 InvestigationMode: 3
c2 346

c1 1356 InvestigationMode: 2
c2 346

c1 1356 InvestigationMode: 1
c2 346

c1 1356 InvestigationMode: 3
c2 346

The main difference against the HNF is the optional element: if a cube specifies a variable as true or false, it is stronger than the optional state. A variable can be optional in the output cube, if both input cubes specify it as optional, or one of them specifies as optional, and the other formula does not contain this variable.
MHNF is a more compact formulation than HNF and therefore the number of cubes and the runtime are smaller. It is also easier to use branching, because the different stages of the process need less memory and the algorithm can save the state of the process more times into the memory.
Figure 3.2 shows the states of the input problem in different phases of the algorithm. Each circle represents a MHNF formula. The diameter of the circles represents the magnitude of the formula size. In phase *Initial merging*, the algorithm collects the similar unprocessed MHNF formulas (orange circles) and they will be merged (green circle). If the formula length hits a specified threshold value, the algorithm switches to *Branching mode*. The blue circle represents the output formula of the previous phase. The algorithm continues the merging step by step – it always gets one cube from each remained formula. If the combination of the cubes does not lead to a model, the algorithm selects an other combination.
Unfortunately, the practical tests showed that the improvements have effect only to the small problems (Less than 50 variables).

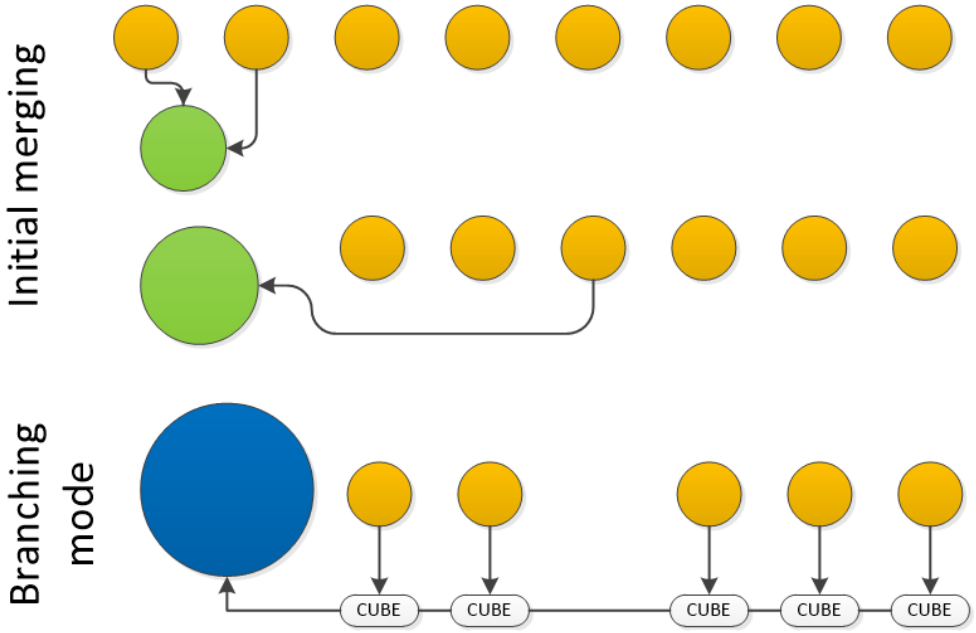


Figure 3.2: Phases in the branching

## 3.3 Inverse Hobel

In spite of improvements, Hobel had too many formulas, cubes, possible assignments and data. Any algorithm around merging were blind alley and we had to find a way to drastically decrease the size of the problem description. We reversed the operation of the algorithm and we got a much better one. This is the **Inverse Hobel**.
The Hobel starts with an empty set and takes the solutions of a CNF clause in form of MHNF formula. Then, it will be merged with other MHNF formulas – a merged formula always lists all solutions to the appropriate CNF clause set. The Inverse Hobel starts with all possible assignments to the problem (one MHNF cube: the optional set of all variables from the problem). The input CNF clauses will be transformed into CNF rules . In practice, it means that the scanned input CNF clauses will be stored in three different sets based on the rule types. The algorithm takes a rule and converts the corresponding MHNF clauses in such a way that the rule will be satisfied.
Note: Example 12 on the page 34 helps understand the following formal description.
Algorithm 3.6 presents the method to application of a MINset rule. Until line 11 the algorithm collects the set of true, optional and false literals from the input cube. If we have at least one true literal, the rule is satisfied by the cube (line 13). If there are only false literals, the cube can not satisfy the rule $\Longrightarrow$ the cube has to be removed (line 15). If there is only one optional element, the only way to satisfy the rule is to disable the false value of this variable $\Longrightarrow$ it must be true (line 17). In the case of more optional elements, NONemptySet from the optional elements will be generated - these represent all cases, when the original cube satisfies the rule.

**Algorithm 3.6:** Application: MINset

```
   Input: MHNFcube c, MINset m
   Output: MHNF formula
 1 SetOfTrueLiterals = new set;
 2 SetOfOptionalLiterals = new set;
 3 SetOfFalseLiterals = new set;
 4 forall the literal l ∈ m do
 5 |   if l in c is true then
 6 |   |   Append l to SetOfTrueLiterals;
 7 |   else if l in c is optional then
 8 |   |   Append l to SetOfOptionalLiterals;
 9 |   else
10 |   |   Append l to SetOfFalseLiterals;
11 |   end
12 end
13 if |SetOfTrueLiterals| > 0 then
14 |   return c;
15 if |SetOfOptionalLiterals| = 0 then
16 |   return null;
17 if |SetOfOptionalLiterals| = 1 then
18 |   set optional element in c to true;
19 |   return c;
20 else
21 |   nonemtyset = MakeNonEmptySets(SetOfOptionalLiterals);
22 |   forall the cube nc ∈ nonemptyset do
23 |   |   forall the literal l_2 ∈ c do
24 |   |   |   if l_2 ∉ SetOfOptionalLiterals then
25 |   |   |   |   Append l_2 to nc;
26 |   |   end
27 |   end
28 |   return nonemptyset;
29 end
```

**Algorithm 3.7:** Application: Exclusion

**Input**: MHNFcube c, Exclusion e
**Output**: MHNF formula

```
1  SetOfTrueLiterals = new set;
2  SetOfOptionalLiterals = new set;
3  SetOfFalseLiterals = new set;
4  forall the literal l ∈ m do
5  |  if l in c is true then
6  |  |  Append l to SetOfTrueLiterals;
7  |  else if l in c is optional then
8  |  |  Append l to SetOfOptionalLiterals;
9  |  else
10 |  |  Append l to SetOfFalseLiterals;
11 |  end
12 end
13 if /SetOfFalseLiterals/ > 0 then
14 |  return c;
15 if /SetOfOptionalLiterals/ = 0 then
16 |  return null;
17 if /SetOfOptionalLiterals/ = 1 then
18 |  set optional element in c to false;
19 |  return c;
20 else
21 |  nonfullset = MakeNonFullSets(SetOfOptionalLiterals);
22 |  forall the cube nc ∈ nonfullset do
23 |  |  forall the literal l2 ∈ c do
24 |  |  |  if l2 ∉ SetOfOptionalLiterals then
25 |  |  |  |  Append l2 to nc;
26 |  |  end
27 |  end
28 |  return nonfullset;
29 end
```

The thinking is same to the Exclusion rule. The Effect rule has a different realization. For example, to the CNF clause ($1 \vee \neg 2 \vee \neg 3 \vee 4$) the rule 2,3 $\rightarrow$ 1,4 will be saved in the form 2 3 0 1 4. 0 separates the condition and effect parts. We use the following graphical representation to demonstrate the different cases:

NrOfTrueLiterals NrOfOptionalLiterals NrOfFalseLiterals | NrOfTrueLiterals NrOfOptionalLiterals NrOfFalseLiterals

The pipe in the middle separates the condition and effect parts of the rule. We write "!", if we know that there is at least one literal, "?", if we do not have information. We write a number in case of specified value. There are three different options: at first, the rule is irrelevant (IRRELEVANT), because it is satisfied with the current cube, the cube has to be modified to satisfy the rule (MODIFY), or another cubes have to be generated from the cube to cover all options (MORE OPTIONS). In case of MODIFY, an optional element will be set to true or false.
Let be C the cube, R the Effect rule, $R_{cp}$ the condition, $R_{ep}$ is the effect part of the rule: $R_{cp} \cup R_{ep} \cup 0 = R$. $C^T$ symbolizes the true literals, $C^O$ the optional elements of C: $C^T \cup C^O = C$.

| Case | Graphical representation | Formal description |
|---|---|---|
| **IRRELEVANT** | ? ? ? \| ! ? ? | $R_{ep} \cap C^T \neq \emptyset$ |
| **IRRELEVANT** | 0 0 ! \| ? ? ? | $R_{cp} \cap C^T = \emptyset$, $R_{cp} \cap C^O = \emptyset$ |
| **MODIFY** | ! 0 0 \| 0 1 ? | $\|R_{cp}\| = \|R_{cp} \cap C^T\|$, $\|R_{ep} \cap C^T\| = 0$, $\|R_{ep} \cap C^O\| = 1$ |
| **MODIFY** | ! 1 0 \| 0 0 ! | $\|R_{cp} \cap C^T\| > 0$, $\|R_{cp} \cap C^O\| = 1$, $\|R_{cp}\| = \|R_{cp} \cap C^T\| + \|R_{cp} \cap C^O\|$, $\|R_{ep} \cap C^T\| + \|R_{ep} \cap C^O\| = 0$ |
| **MORE OPTIONS** | ! 0 0 \| 0 ! ? | $\|R_{cp}\| = \|R_{cp} \cap C^T\|$, $\|R_{ep} \cap C^T\| = 0$, $\|R_{ep} \cap C^O\| > 1$ |
| **MORE OPTIONS** | ? ! 0 \| 0 0 ! | $\|R_{cp} \cap C^O\| > 1$, $\|R_{cp}\| = \|R_{cp} \cap C^T\| + \|R_{cp} \cap C^O\|$, $\|R_{ep} \cap C^T\| + \|R_{ep} \cap C^O\| = 0$ |
| **MORE OPTIONS** | ? ! 0 \| ? ! ? | $\|R_{cp} \cap C^O\| > 1$, $\|R_{ep} \cap C^O\| > 1$ |

The IRRELEVANT cases are simple: if the condition part is not satisfied, or the effect part is satisfied, then the rule is surely satisfied. The MODIFY cases are also not complex: if the condition part is satisfied, but the effect part not and it has only one optional element from the rule, it has to be true. The other option, when the effect part is not satisfied; and the fulfillment of the condition part depends on one optional element: it must be false.
In the MORE OPTIONS cases, the fulfillment of the parts depends on more optional elements. The tasks to the different cases are the following:

- ! 0 0 | 0 ! ? - NONemptySet(RightSideOptionalLiterals). Effect part must be true.

- ? ! 0 | 0 0 ! - NONfullSet(LeftSideOptionalLiterals). Condition part should not be true.
- ? ! 0 | ? ! ? - NONfullSet(LeftSideOptionalLiterals). $\forall$ cube c $\in$ NONemptySet(RightSideOptionalLiterals) : c $\cup$ FullSet(LeftSideOptionalLiterals).

Note: NONemptySet of variables lists all cases in MHNF, such that there is always at least one true variable. By contrast, a NONfullSet describes cases in MHNF, such that the variables are never true together. These sets are useful to list possibilities for different conditions.

**Example 12** – Inverse Hobel
Input CNF problem: $(A \vee \neg B \vee C) \wedge (B \vee D) \wedge (\neg B \vee E \vee \neg A) \wedge (\neg C \vee \neg A \vee \neg E)$

And the corresponding CNF rules:

1: $B \rightarrow A,C$ 2: MinSET(B,D) 3: $AB \rightarrow E$ 4: !ACE

| Models to the problem | Applied rule number |
|---|---|
| ABCDE | - |
| ACDE $\oplus$ ABCDE $\oplus$ BCDE | 1 |
| ACDE $\oplus$ ABCDE $\oplus$ BCDE | 2 |
| ACDE $\oplus$ ABCDE $\oplus$ BCDE | 3 |
| ACD $\oplus$ ADE $\oplus$ CDE $\oplus$ ABDE $\oplus$ BCDE | 4 |

At start, all literals are optional. Example 12 demonstrates that process. When we look at the first rule, we have to consider two cases: when B is false, the other literals are irrelevant (condition part of the rule is false - we do not have to make any changes, first cube in the second row). The other case, when B is true, at least one literal from the set A,C has to be true (condition part of the rule is true, the effect part has to be true, too - second and third cubes).

**Definition 19 (solution set):** According to the above described principle, the Inverse Hobel always has exactly one MHNF formula with variable number of cubes. This only formula is the **solution set** to the original problem.

If at a point the solution set is empty, the problem is unsatisfiable. After the application of all rules we get all models to the problem. The other notable thing is the lack of the content lists: the procedure lost the load of content list check and management.

### 3.3.1 The difference between Hobel and Inverse Hobel in practice

In all following tests we used an Intel Core i5-4310M as CPU and NVIDIA GTX 860M (GM107) as GPU. In the first test, we used a problem with 20 variables and 91 cubes, all clauses contained 3 literals. In the basic implementations, algorithms worked on the

input sequentially. In the "PS" mode, algorithm preferred the similar formulas or rules.

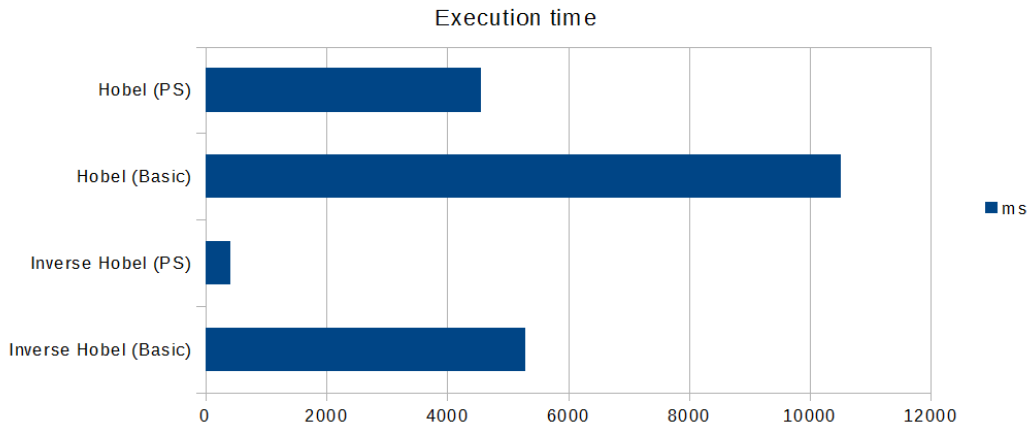


Figure 3.3: Execution time of different variants (Hobel, Inverse Hobel)

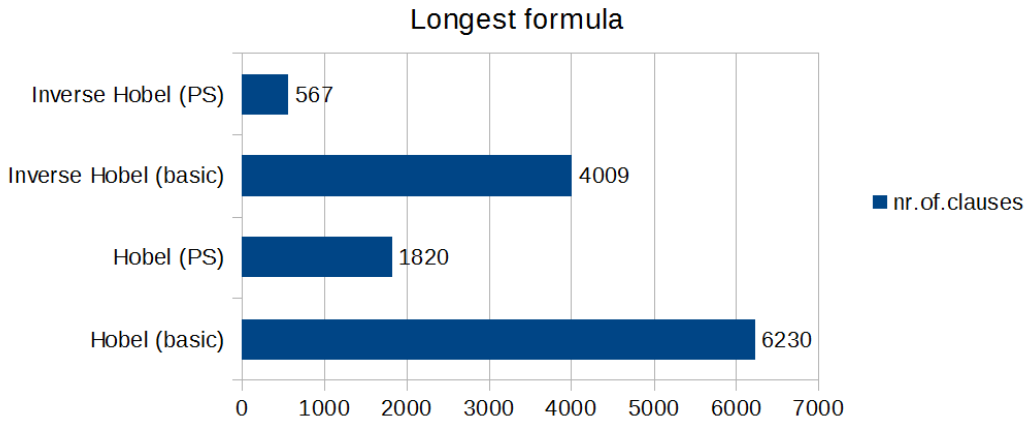


Figure 3.4: Longest formula during the execution (Hobel, Inverse Hobel)

The Inverse Hobel is much better than the Hobel: it finds the models much faster and uses less data (567 vs 1820 max formula length). It is very important that Inverse Hobel has always exactly one formula (solution set), but Hobel operates with many formulas parallel. 1820 was the longest formula length, but it had probably other 40 with less cubes. PS mode showed that the order of the data processing has a very huge effect on the memory using and execution time.

## 3.4 GPU implementation

The introduced improvements - for instance the preference of the similar formulas/rules, MHNF and inverse direction - always decreased the amount of the data; and the algorithm could always solve more complicated problems. In connection with this issue we performed the following experiment: We implemented the parallel version of the Inverse Hobel to the CPU, but this did not fulfill the hopes. The improvement in the execution speed was low and the more complicated problems were still unsolvable. The next step in the Hobel evolution is the GPU implementation. This version is designed to CUDA capable GPUs; and one of the important objectives is to avoid unnecessary cube generation. A typical characteristic of the Hobel algorithms is the "bulge" in the graph processed CNF clauses (x axis)/number of found models (y axis). After start, the number of possible models immediately grows, than after a point it is decreasing. The main difficulty is to reach the peak, because it means lot of formulas (Hobel) and cubes (Inverse Hobel). Avoidance of unnecessary cube generation is vitally important.
An another goal was to avoid multiple, unnecessary kernel calls and CPU-GPU data transfer. In the new implementation, most of the free memory will be allocated on the GPU; and the algorithm has to manage the memory using during the execution. The data structure is the following:
- MHNFsSlots: the MHNF cubes will be saved into this array.
- MHNFsState: to each MHNF slot there is a flag which indicates, whether the slot is in use.
- MHNFsIgnore: to each MHNF slot there is a flag which indicates, whether the cube has to be ignored for some reason.
- CurrentRuleContent: this is an MHNF slot which always shows the content of the currently being processed rule.

To each rule type there are three different data:
- one array to the rules itself.
- one array which describes for the threads, where starts and ends the data of the corresponding rules.
- a flag rendered to each rule which indicates, whether the rule has been already applicated.

- GeneratedMHNFs: each thread has some MHNF slots, if another MHNF cubes will be generated during the process (based on NONemptySet and NONfullSet).
- NrOfGeneratedMHNFs: each thread has an integer which stores the number of generated MHNFs.
- GeneratedMHNFsStatus: each generated MHNF has a flag, which indicates, whether the generated MHNF has to be appended to the solution set.
- MHNFErasingIndicator: each thread has a flag which indicates, when the MHNF of the thread has to be removed.
- OptionalLiterals: each thread has some integer slots to save the optional elements during the processing.

- RightSideOptionalLiterals: each thread has some integer slots to save the optional elements from the right side during the processing.

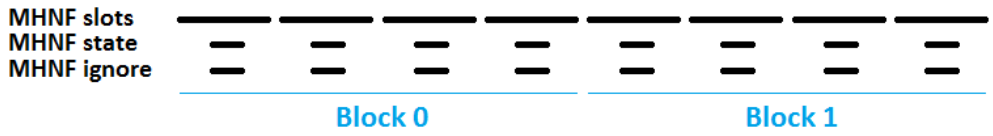


Figure 3.5: MHNFs slots and the flags

Figure 3.5 shows the data structure. If the kernel runs with four threads, the MHNF slots will be grouped into four slots-blocks. When the algorithm applies a rule, kernel starts with the Block 0 and continues until the last Block. An integer FirstFreeSlot and LastReservedSlot store the indexes of the first free and last used slots. Therefore the algorithm never processes a block, which does not contain any used MHNF slots. These flags are very important, because for example if 1 GB RAM has been allocated to the MHNFsSlots, there can be thousands of blocks. But in the early phase of the algorithm, most of them are surely not in use and the algorithm can avoid the excrescent iteration. At start, only the first MHNF slot is in use. The LastUsedSlot has value zero, the FirstFreeSlot one. The first MHNF state flag is one, all others are zero. All MHNF ignore flags with the value of zero.
The threads can apply a rule parallel and generate another MHNF cubes (MORE OPTIONAL), if it is needed. In this case, MHNFErasingIndicator flag of the thread will be set to one. This carries the information that the cube of the corresponding thread will be replaced with other options – it can be deleted. The next step is the filtering: the remained, unprocessed rules will be analysed and each generated cube will be modified (MODIFY) until there is not any other possible modifications. Of course, after some modifications it is possible that some cubes do not satisfy one or more rules anymore. In this case, these cubes are dead, consequently they have to be filtered out. Threads can indicate the result of the forfiltering with the GeneratedMHNFsStatus flags (one = dead).
The next stage is the appending the generated cubes to the MHNFsSlots. This will occure only sequentially. MHNFErasingIndicator and MHNFsIgnore flags play here an important role. Before appending, MHNFs have to be removed based on the MHNFErasingIndicator states. During appending, threads fill first these holes. In some cases, the existing and the generated cubes do not have enough place within the current block; and some generated cubes will be saved to the area of another block(s). In such a case, threads indicate that with the MHNFsIgnore flag. Accordingly, the application of the current rule on the cube is not necessary (the cube has been generated based on this rule an therefore the cube satisfies the current rule).
Regarding this case, Figure 3.6 demonstrates an example. We can see that thread 2 generated four cubes. The first one goes into the thread's slot. Two of them have been

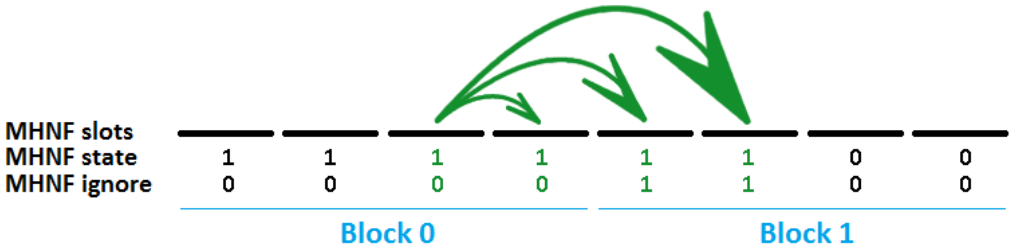


Figure 3.6: Appending cubes into MHNFsSlots

saved into the area of the other block, therefore the thread has to indicate the ignorance for the next block.
In addition, some words about memory allocation to generated cubes: the CPU allocates all memory for the kernel; therefore it is important not to waste the memory – or in other words - not to allocate so much unnecessary memory for different data. In case of GeneratedMHNFs we exactly know, what the maximum possible number of generated cubes is: the worst case is the last case of the Effect rule. The algorithm has to generate
- NONfullSet from the LeftSideOptionalLiterals.
- FullSet from the LeftSideOptionalLiterals with NONemptySet of RightSideOptionalLiterals.
NONfullSet and NONemptySet do not generate more cubes, than the number of the input variables. The input variables come from the rules, so the greatest possible number of generated cubes is the length of the longest rule. Therefore, the problems with less variable in CNF clause lengths fit better to memory allocation.
Algorithm 3.8 lists the formal description. Before kernel run, CPU initializes the data for the algorithm. ThreadSlot is the index to MHNF slot of the actual thread. A block is active, if there is at least one MHNF slot in use.

The Figure 3.7 shows the effectiveness of the new filtering. At a point, CPU version listed more than 800 possible MHNF models to the problem. By contrast, the GPU did not suffer from the ”bulge” effect.
The advantage of the GPU version leads to a very significant speed-up in a little more complicated problem (Figure 3.8). The new implementation has been compared to the parallel CPU version again. We emphasize the main difference between the versions: the GPU found all models  222 time faster, than the CPU just the first model with four threads.
There is also a huge progress in area of memory consumption. Regarding one aspect of the test problem (100 variables, 430 cubes) we never saw any solutions from the CPU, though the algorithm had more than 3 hours to find one of them. The peak memory consumption at the Hobel was  3,5 GB, meanwhile in case of Inverse Hobel (CPU) it was 175 MB – before manual shutdown. The memory consumption of the GPU variant is much less. We measured it with different number of used threads.

**Algorithm 3.8:** Inverse Hobel GPU

**Input**: MINsets ms, Exclusions es, Effects fs
**Output**: MHNF formula (models to the problem)

```
1  repeat
2      forall the active block bl do
3          ApplyRule();
4          forall the thread thr (performed by thread 0) do
5              if ErasingIndicator[thr] = 1 then
6                  Erasing data in the slot;
7                  NumberOfCurrentModels -= 1;
8                  if ThreadSlot < FirstFreeSlot then
9                      FirstFreeSlot = ThreadSlot;
10         end
11         forall the thread thr (performed sequentially) do
12             forall the GeneratedMHNF cube grc do
13                 if grc state = 0 then
14                     goto next cube;
15                 Copy grc into MHNFsSlots[FirstFreeSlot];
16                 MHNFsState[FirstFreeSlot] = 1;
17                 if FirstFreeSlot > LastUsedSlot then
18                     LastUsedSlot = FirstFreeSlot;
19                 NumberOfCurrentModels += 1;
20                 forall the sli > FirstFreeSlot do
21                     if MHNFsState[sli] = 0 then
22                         FirstFreeSlot = sli;
23                         break;
24                 end
25             end
26         end
27         SelectNewActualRule() (performed by thread 0);
28     end
29 until there exists free MHNF slot or there exists unprocessed rule;
30 return;
```

**Algorithm 3.9:** SelectNewActualRule

```
1  BestNrOfCommonVariables = -1;
2  forall the msr ∈ ms do
3  |  ActualNumberOfCommonVariables = 0;
4  |  if ms state = 0 then
5  |  |  goto next ms;
6  |  Update ActualNumberOfCommonVariables with current rule;
7  |  if ActualNumberOfCommonVariables > BestNrOfCommonVariables then
8  |  |  BestNrOfCommonVariables = ActualNumberOfCommonVariables;
9  |  |  set ms as next rule;
10 end
11 forall the esr ∈ es do
12 |  ActualNumberOfCommonVariables = 0;
13 |  if es state = 0 then
14 |  |  goto next es;
15 |  Update ActualNumberOfCommonVariables with current rule;
16 |  if ActualNumberOfCommonVariables > BestNrOfCommonVariables then
17 |  |  BestNrOfCommonVariables = ActualNumberOfCommonVariables;
18 |  |  set es as next rule;
19 end
20 forall the fsr ∈ fs do
21 |  ActualNumberOfCommonVariables = 0;
22 |  if fs state = 0 then
23 |  |  goto next fs;
24 |  Update ActualNumberOfCommonVariables with current rule;
25 |  if ActualNumberOfCommonVariables > BestNrOfCommonVariables then
26 |  |  BestNrOfCommonVariables = ActualNumberOfCommonVariables;
27 |  |  set fs as next rule;
28 end
29 Update CurrentRuleContent based on the next rule;
```

**Algorithm 3.10:** ApplyRule

```
1  forall the thread thr do
2      switch RuleType of actual rule do
3          case MINset
4              GeneratedMHNFs = ApplyMINsetRule();
5              MHNFState[ThreadSlot] = 0;
6              ErasingIndicator[thr] = 1;
7          case Exclusion
8              GeneratedMHNFs = ApplyExclusionRule();
9              MHNFState[ThreadSlot] = 0;
10             ErasingIndicator[thr] = 1;
11         case Effect
12             GeneratedMHNFs = ApplyEffectRule();
13             MHNFState[ThreadSlot] = 0;
14             ErasingIndicator[thr] = 1;
15     endsw
16     forall the generated cube gc do
17         Filtering gc;
18     end
19 end
```

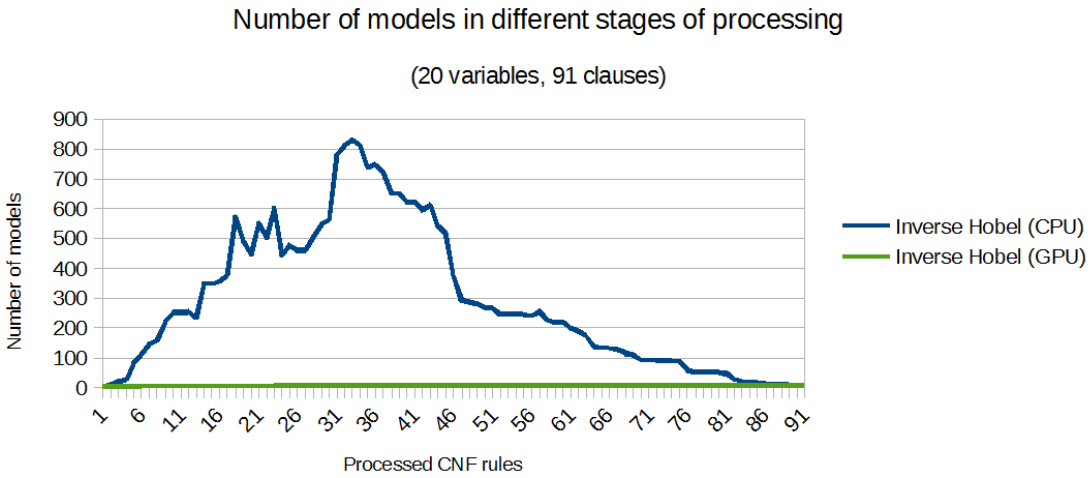


Figure 3.7: Number of MHNF cubes in different stages (Inverse Hobel CPU/GPU)

The measured memory consumption is based on the used MHNF slots. The algorithm allocates most of the available memory, but this is not equal with the used amount.
We also tested the speed of the process with different number of threads (Figure 3.10).

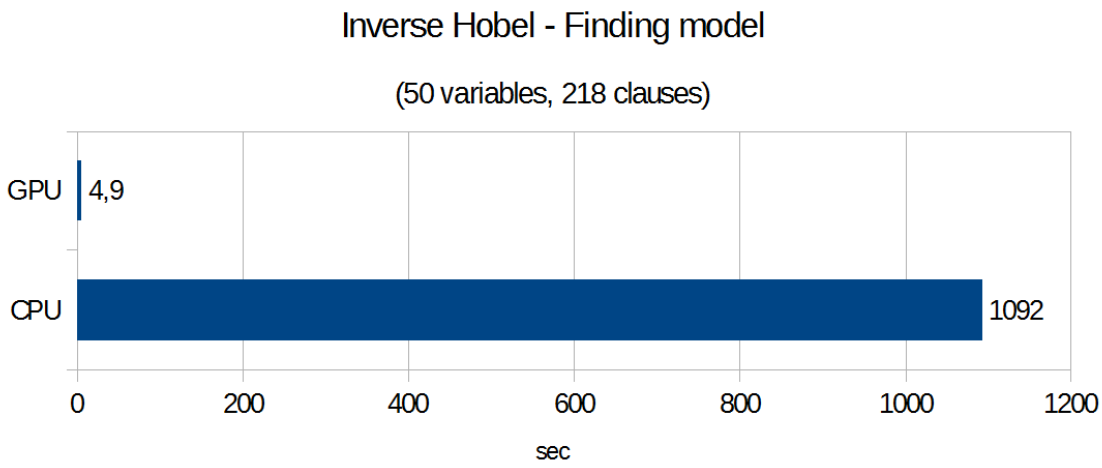


Figure 3.8: Finding model. CPU variant is parallel (Inverse Hobel CPU/GPU)

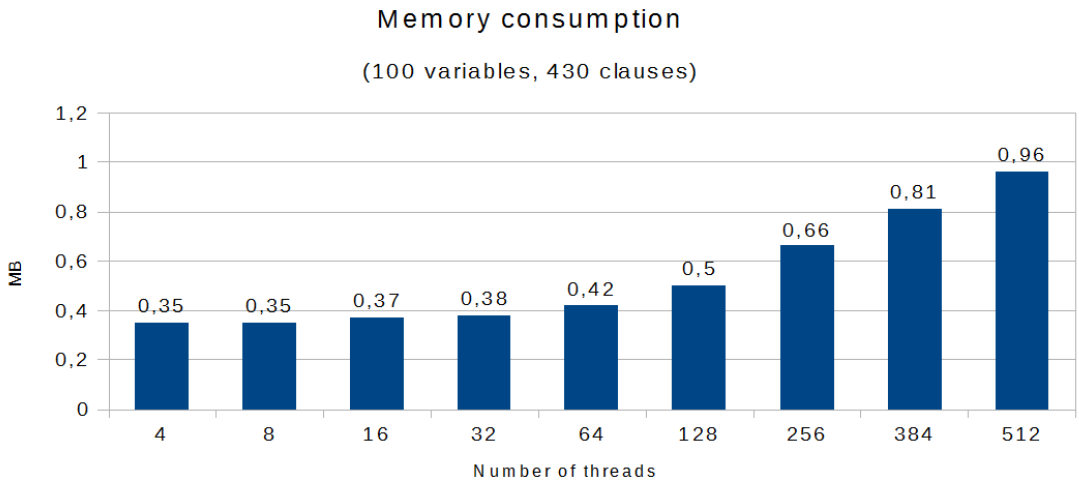


Figure 3.9: Peak memory consumption of the GPU during process

The algorithm always found all models. If we used more than 128 threads, the increasing of the number of threads did not have significant effect. This comes from the fact that in many cases most of the threads did not have data to process – the problem was not enough big, they were in idle. Under 128 threads, the result is fast halved execution time (per four times more threads). We think, the sequential parts of the algorithm prevent the better speed-up. The GPU version is far better than all previous CPU versions, however a bigger problem (250 variables and 1000+ cubes) is still too difficult to the actual variant (no result after 30 minutes). The tests showed, that the algorithm benefits

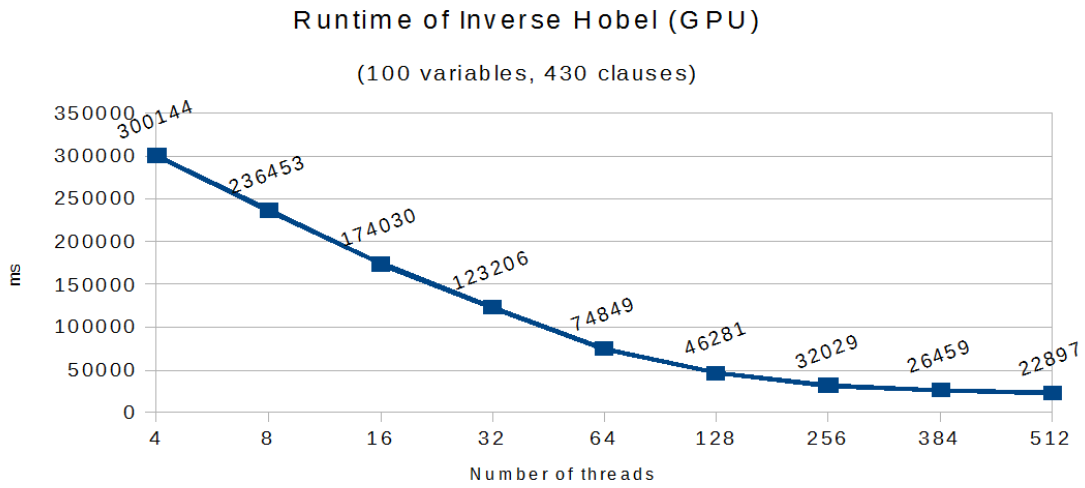


Figure 3.10: Speed-up based on the increasing of the threads

from the increase of the threads – provided there are enough data to process. This means, the algorithm would be useful for stronger GPUs.

## 3.5 SetStep

The algorithm has been significantly improved and it is much smarter than the first version. In this section, we try to solve an another outstanding problem: the algorithm collects too much possibilities during the run and this leads to ”out of memory” situation with additional difficult problems. To solve that issues, the algorithm should switch tactic and continue the work with no or minimal extra memory consumption. The name of this procedure is **SetStep**, which satisfies the conditions.
The main idea is to find a cube to each rule avoiding any conflicts between the cubes. With other words, there is no variable v $\in$ V, such that there are two selected cubes $c_1$ and $c_2$, such that v $\in c_1$ and $c_2 \rightarrow \neg$v. As first step, the rules will be ordered for processing. The ApplyRule runs until it has free MHNF slot. When ApplyRule terminated, some rules have been processed and the reserved MHNF slots contain different cubes. These cubes satisfy the already processed rules. Next, for each cube, a SetStep process goes from rule to rule and chooses a set which satisfy the rule and it does not have conflict with the cube. If it is needed, SetStep modifies the cube and step back, when there is not any compatible sets at a certain rule.

**Definition 20 (SetStep assignment or SA):** the SetStep assignment of a SetStep algorithm is the inherited cube from the ApplyRule. If the ApplyRule processed m rules, the SA has to satisfy at least m + n rules, if the SetStep selected n cubes.

We slightly modify the procedure of NONfullSet:

**Algorithm 3.11:** MakeNonFullSets2

```
   Input: Set of variables (VIN)
   Output: MHNF formula
 1 Resultset = new MHNF formula;
 2 forall the variable v ∈ VIN do
 3 |   Subset = new set;
 4 |   forall the variable w ∈ VIN do
 5 |   |   if v = w then
 6 |   |   |   goto next w;
 7 |   |   else if w > v then
 8 |   |   |   Append +w to Subset;
 9 |   |   else
10 |   |   |   Append -w to Subset;
11 |   |   end
12 |   end
13 |   Append Subset to ResultSet;
14 end
15 return ResultSet;
```

Example 13 shows the outputs of the NONemptySet and the new NONfullSet for input set A, B, C, D, and E.

**Example 13** – Outputs of NONemptySet and NONfullSet

| | |
|---|---|
| NONemptySet | A$\underline{\text{BCDE}}$ ⊕ B$\underline{\text{CDE}}$ ⊕ C$\underline{\text{DE}}$ ⊕ D$\underline{\text{E}}$ ⊕ E |
| NONfullSet | BCDE ⊕ $\underline{\text{A}}$CDE ⊕ $\underline{\text{AB}}$DE ⊕ $\underline{\text{ABC}}$E ⊕ $\underline{\text{ABCD}}$ |

Let be NES (NFS) the output of a NONemptySet (NONfullSet), VIN the set of input variables. The number of cubes in the output in both cases is equal to |VIN|. Let be $es_n$ ($ef_n$) the cubes in the output of NES (NFS), n = 1..|VIN|. The outputs have the following properties:

- each $es_n$ cube defines n assigned variables
- each $ef_n$ cube defines |VIN| - n + 1 assigned variables

The consequences of the above properties are:

- each $es_n$ cube defines $2^{|VIN|-n}$ possible partial assignments, which evaluates the corresponding clause to true

- each $ef_n$ cube defines $2^{n-1}$ possible partial assignments, which evaluates the corresponding clause to true

From the algorithm's point of view, these facts are important: when it is looking for compatible cubes, it should select the cubes so that the number of assigned variables is minimal - this maximizes the chance to find a model to the problem. As the Example 13 demonstrates, it should select a cube from the beginning (end) in case of NONempySet (NONfullSet).
When the algorithm selects a cube, it is consistent with the SA, or the algorithm has to modify the SA (switch a variable to false/true).
To use this algorithm, we have to describe the rules with Hobel sets. This provides the opportunity for the algorithm to select a cube from a set and to satisfy the actual rule.

**Example 14** – CNF rules and the Hobel sets

| CNF rule type | Set description |
|---|---|
| MINset | NONemptySet(literals) |
| Exclusion | NONfullSet(literals) |
| Effect | NONfullSet(Condition Part)NONemptySet(Effect part) |

The cases of MINset and Exclusion are evident: at MINset, the NONemptySet guarantees that there will be at least one true literal in each cube. At Exclusion, similarly, the NONfullSet provides alone the satisfiability. The Effect rule has to use both set types. The thread is as follows: the algorithm tries to find an appropriate cube from the NONfullSet. If this is not possible, then all variables from this set should be assigned to true. Consequently, the Condition part of the rule will be satisfied, so it has to be at least one variable from the Effect part, which is true. The NONemptySet (Effect part) ensures this, so selecting a cube from the Effect part satisfies the rule.
Because the number of cubes in the output of each Hobel set is equal to the number of input variables (VIN), selecting a cube is equivalent to selecting a v $\in$ VIN. The following properties are true for NONemptySet and NONfullSet:
Selecting the $es_n$ cube from a NONemptySet:

- the nth variable is true
- $\forall\ i < n$: $v_i$ is false
- $\forall\ j > n$: $v_j$ is optional

Selecting the $ef_n$ cube from a NONfullSet:

- the nth variable is false

- $\forall\ i < n$: $v_i$ is optional
- $\forall\ j > n$: $v_j$ is true

The best way to check these properties is detailed described in the Example 13. As above explained, for NONemptySet and NONfullSet there are different cube-selecting procedures. In both cases, the algorithm tries to find a cube avoiding any conflicts between the cube and SA. Furthermore, the number of assigned variables in the SA will be minimal. If it is needed, a variable in the SA will be switched from optional to true/false. As ApplyRule, SetStep uses Filtering, too.

**Definition 21 (Decision variable):** a variable v is a decision variable of rule r, if v $\in$ SA becomes true/false at the processing of r. $DEC_r$ is the set of decision variables of r.

When the algorithm does not find appropriate cube at a rule, it has to perform a BackStep. All decision variables of the rule will be removed from the SA and the previous rule has to find an another cube.

**Definition 22 (BackStep):** a BackStep at rule r is a state change of SA, if $DEC_r \neq \emptyset$: SA $\Longrightarrow$ $\forall$v$\in$ $DEC_r$: if $v \in$ SA: SA $\setminus$ $v$. SA $\cup$ -$v$.

Algorithms 3.12-15 show the formal description of the SetStep. ApplyRule processed the first m rules from the ordered set. For each rule, there is an index (I) which shows the last selected cube. If no cube selected yet, the index has value -1. After the formal description, the whole process is demonstrated by an example. Again, selecting a variable from a set is equivalent to selecting a cube.

Notes for algorithm 3.12: at line 1, we set the StepIndex to the first unprocessed rule. The algorithm goes from rule to rule until it is possible (line 32), or it can select a cube from the last set without conflict (lines 8, 16, 24). When it does not find appropriate cube at a set, it has to perform a BackStep (lines 10, 18, 29). At lines 26-30 the algorithm controls the Effect rules: if no cube has been selected from the Condition part, a cube has to be selected from the same rule in the next iteration. *WorkingInConditionPart* indicates the state of the processing.
In the other parts of the algorithm (3.13-3.15) there are some common principles. Basically, there are two cases:

- the algorithm will select the first cube;
- the algorithm has to select an another one, provided there were not any appropriate cubes for the next set. In this latter case, it has to perform a BackStep

In the first case, the value of I is -1 (line 1). The algorithm starts at the specific point of the set (see consequences of the sets at Example 13). If the algorithm finds an appropriate variable, it selects the cube. For example, in SelectCubeFromMINset, at line 3 the algorithm found a variable, which is true in the SA. It is possible that there were

**Algorithm 3.12:** SetStep

**Input**: MHNF cube SA, set of ordered rules R
**Output**: MHNF cube and a boolean variable, whether the cube is model

```
1  StepIndex = m + 1;
2  repeat
3      switch RuleType of R[StepIndex] do
4          case MINset
5              if SelectCubeFromMINset(R[StepIndex]) == true then
6                  StepIndex++;
7                  if StepIndex == |R| then
8                      Model found. Return with SA;
9              else
10                 BackStep;
11                 StepIndex-=1;
12         case Exclusion
13             if SelectCubeFromExclusion(R[StepIndex]) == true then
14                 StepIndex++;
15                 if StepIndex == |R| then
16                     Model found. Return with SA;
17             else
18                 BackStep;
19                 StepIndex-=1;
20         case Effect
21             if SelectCubeFromEffect(R[StepIndex]) == true then
22                 StepIndex++;
23                 if StepIndex == |R| then
24                     Model found. Return with SA;
25             else
26                 if WorkingInConditionPart == true then
27                     WorkingInConditionPart = false;
28                     Continue;
29                 BackStep;
30                 StepIndex-=1;
31     endsw
32 until StepIndex > m;
```

**Algorithm 3.13:** SelectCubeFromMINset

**Input**: A MINset rule r
**Output**: boolean variable, whether cube found

```
1  if I == -1 then
2      forall the variable v ∈ r do
3          if SA[v] == 1 then
4              Update I; Return true;
5          else if SA[v] == -1 then
6              SA[v] = 1;
7              DEC_r = DEC_r ∪ v;
8              Filtering;
9              if There is conflict then
10                 WithDrawing changes;
11                 SA[v] = 0;
12                 DEC_r = DEC_r ∪ v;
13                 Goto next v;
14             else
15                 Update I;
16                 Return true;
17             end
18     end
19     Return false;
20 else
21     if r[I] ∈ DEC_r then
22         SA[r[I]] = 0;
23         forall the variable v ∈ r, r[i] = v and i > I do
24             if SA[v] == 1 then
25                 Update I; Return true;
26             else if SA[v] == -1 then
27                 SA[v] = 1;
28                 DEC_r = DEC_r ∪ v;
29                 Filtering;
30                 if There is conflict then
31                     WithDrawing changes;
32                     SA[v] = 0;
33                     DEC_r = DEC_r ∪ v;
34                     Goto next v;
35                 else
36                     Update I;
37                     Return true;
38                 end
39         end
40     else
41         Return false;
42 end
```

**Algorithm 3.14:** SelectCubeFromExclusion

**Input**: An Exclusion rule r
**Output**: boolean variable, whether cube found

```
1  if I == -1 then
2      forall the variable v ∈ r do
3          if SA[v] == 0 then
4              Update I; Return true;
5          else if SA[v] == -1 then
6              SA[v] = 0;
7              DEC_r = DEC_r ∪ v;
8              Filtering;
9              if There is conflict then
10                 WithDrawing changes;
11                 SA[v] = 1;
12                 DEC_r = DEC_r ∪ v;
13                 Goto next v;
14             else
15                 Update I;
16                 Return true;
17             end
18     end
19     Return false;
20 else
21     if r[I] ∈ DEC_r then
22         SA[r[I]] = 1;
23         forall the variable v ∈ r, r[i] = v and i < I do
24             if SA[v] == 0 then
25                 Update I; Return true;
26             else if SA[v] == -1 then
27                 SA[v] = 0;
28                 DEC_r = DEC_r ∪ v;
29                 Filtering;
30                 if There is conflict then
31                     WithDrawing changes;
32                     SA[v] = 1;
33                     DEC_r = DEC_r ∪ v;
34                     Goto next v;
35                 else
36                     Update I;
37                     Return true;
38                 end
39         end
40     else
41         Return false;
42 end
```

**Algorithm 3.15:** SelectCubeFromEffect

**Input**: An Effect rule r
**Output**: boolean variable, whether cube found

```
1  if I == -1 then
2      WorkingInConditionPart = true;
3      forall the variable v ∈ r do
4          if SA[v] == 0 then
5              Update I; Return true;
6          else if SA[v] == -1 then
7              SA[v] = 0;
8              DEC_r = DEC_r ∪ v;
9              Filtering;
10             if There is conflict then
11                 WithDrawing changes;
12                 SA[v] = 1;
13                 DEC_r = DEC_r ∪ v;
14                 Goto next v;
15             else
16                 Update I;
17                 Return true;
18             end
19     end
20     Set I to position of 0;
21     Return false;
22 else
23     if WorkingInConditionPart == true then
24         Return SelectCubeFromExclusion(Condition Part of r);
25     else
26         Return SelectCubeFromMINset(Effect Part of r);
27 end
```

some previous variables, which were false in the SA; and the algorithm always went to the next variable. In case of NONemptySet, the selected variable has to be true and all previous variables false. So if the actual variable is false in the SA, the algorithm has to continue the search, because the cube of this variable would be incorrect. If a variable is optional in the SA, this can switched to true (in case NONfullSet to false) and the algorithm has a correct cube (line 6). The variable will be a decision variable of the rule (line 7) and a Filtering will be performed. The Filtering works more or less on the same way as at ApplyRule. During Filtering, the set of decision variables will be probably expanded. If there is a conflict, the changes of the Filtering will be withdrawn and the actual variable will be flipped to false (line 11). Again, in the next iteration of the for loop (line 2) we go to the next variable and all previous variables have to be false. Without any conflicts, the algorithm successfully found a cube and it can return with true (line 16).
In the other case, when at least one cube has been already selected from the set, the value of the I shows the index of the last selected cube. There are two possibilities: if this variable belongs to set decision variables of the rule, the variable was originally optional and this rule switched it to true. If the variable is switched by this rule, the variable will not have any effects on the satisfiability of all previous rules - it can be flipped from true to false (line 22). The other option is that the variable was originally true and the set returned at line 4 in the previous round. This means, the true value has been assigned at one of the previous rules; and this rule should not change this assignment. The consequence is the return with false (line 41), because all remained cubes are incorrect - one of the previous variables would be true.
The principle is same in the other parts. The treatment of the Effect rule is a bit different. When the algorithm selects cubes from the Condition part, the variables of the Effect part are irrelevant. When there is no other possible cube from the Condition part, then at least one variable has to be true from the Effect part (all variables from the Condition part have true value in the SA). After processing of the Condition part, the algorithm does not go to the next rule, but switches to the Effect part.

**Example 15** – SetStep

Note: the example does not use Filtering and starts with the OptionalSet of the variables. The column SA lists only the non-optional variables.

| | CNF clause | rule | set description |
|---|---|---|---|
| 1 | (A ∨ B ∨ C) | MIN(A,B,C) | NONemptySet(A,B,C) |
| 2 | (B ∨ C ∨ ¬D) | D → B,C | NONfullSet(D)NONemptySet(B,C) |
| 3 | (¬A ∨ ¬C ∨ ¬D) | !(A,C,D) | NONfullSet(A,C,D) |
| 4 | (B ∨ C ∨ E) | MIN(B,C,E) | NONemptySet(B,C,E) |
| 5 | (B ∨ D) | MIN(B,D) | NONemptySet(B,D) |
| 6 | (¬A ∨ ¬B) | !(A,B) | NONfullSet(A,B) |

| Index states | | | | | | Decision sets | | | | | | SA |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 | |
| 1 | -1 | -1 | -1 | -1 | -1 | A | | | | | | A |
| 1 | 1 | -1 | -1 | -1 | -1 | A | D | | | | | A ¬D |
| 1 | 1 | 3 | -1 | -1 | -1 | A | D | | | | | A ¬D |
| 1 | 1 | 3 | 1 | -1 | -1 | A | D | | B | | | A ¬D B |
| 1 | 1 | 3 | 1 | 1 | -1 | A | D | | B | | | A ¬D B |
| 1 | 2 | -1 | -1 | -1 | -1 | A | DB | | | | | A D B |
| 1 | 2 | 2 | -1 | -1 | -1 | A | DB | C | | | | A D B ¬C |
| 1 | 2 | 2 | 1 | -1 | -1 | A | DB | C | | | | A D B ¬C |
| 1 | 2 | 2 | 1 | 1 | -1 | A | DB | C | | | | A D B ¬C |
| 1 | 3 | -1 | -1 | -1 | -1 | A | DBC | | | | | A D ¬B C |
| 2 | -1 | -1 | -1 | -1 | -1 | AB | | | | | | ¬A B |
| 2 | 1 | -1 | -1 | -1 | -1 | AB | D | | | | | ¬A B ¬D |
| 2 | 1 | 3 | -1 | -1 | -1 | AB | D | | | | | ¬A B ¬D |
| 2 | 1 | 3 | 1 | -1 | -1 | AB | D | | | | | ¬A B ¬D |
| 2 | 1 | 3 | 1 | 1 | -1 | AB | D | | | | | ¬A B ¬D |
| 2 | 1 | 3 | 1 | 1 | 1 | AB | D | | | | | ¬A B ¬D |

MHNF model found: B<u>CE</u>

Figures 3.11, 3.12 and 3.13 demonstrate the test results. On the first one we can see the runtime with different number of threads. In the test, the phase ApplyRule had exactly as much MHNF slots as the number of threads. After they had been loaded, the algorithm switched the method; and a SetStep method were started for each MHNF cube. The figure is remarkable: under 64 threads the algorithm has very bad runtime. We list constant 350000 ms for the better readability. With 64 and more threads the overall picture is not better: SetStep is much slower than the pure ApplyRule. The situation is interesting, when the number of threads increases from 384 to 512. The reason is complex. Figure 3.12 shows the results of the work in phase ApplyRule. The number of processed rules does not grow as quickly, as the number of threads. This comes from the exponential nature of the approach. So 512 SetStep algorithms have to select almost as much cubes, as 128 or 64 have to select.

The SetStep does not fit the GPU so, as the ApplyRule does. In the latter case, the algorithm always applies the same rule with many different MHNF cubes. In case of SetStep, there is not any synchronizations between the threads, because after x iteration the threads work at different rules (think of the BackStep). The threads need different data from the memory; and the high number of threads leads to competition for resource.

The figure about the memory consumption is illusive. SetStep needed more memory in all cases, than the pure ApplyRule. In fact, the problem is not enough big to highlight the exponential nature of the ApplyRule. Filtering is effective in keeping the number of

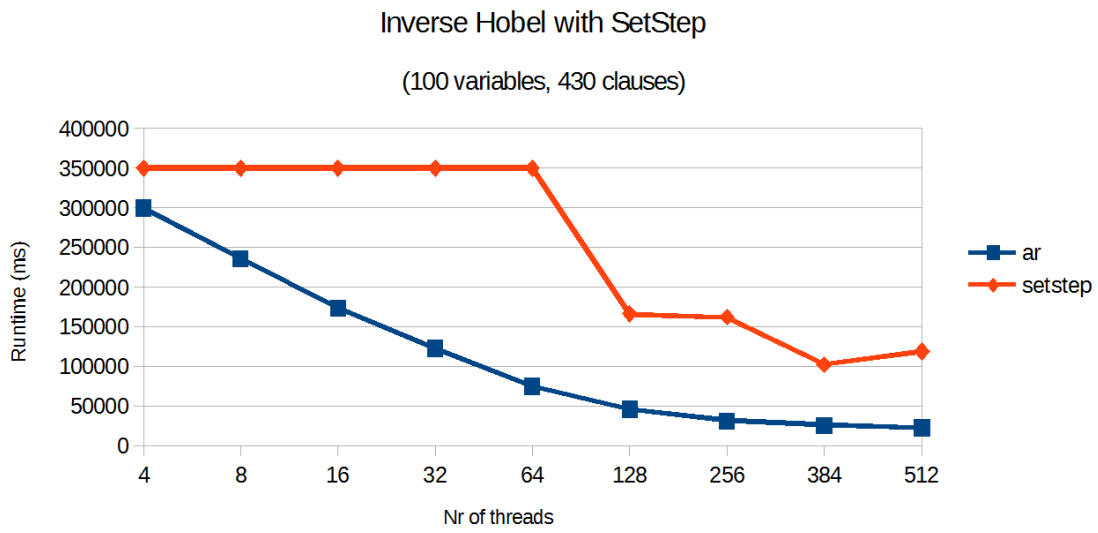


Figure 3.11: Runtime of SetStep again ApplyRule with different number of threads

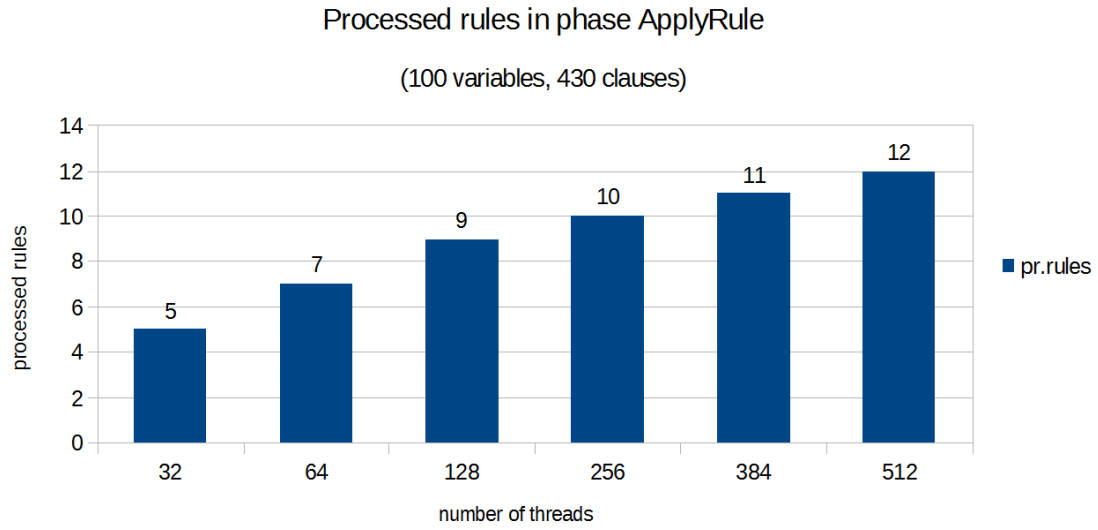


Figure 3.12: Number of processed rules in phase ApplyRule with different number of threads

busy MHNF slots low. SetStep needs extra data to each MHNF slot (to each instance), but these data do not have exponential size.

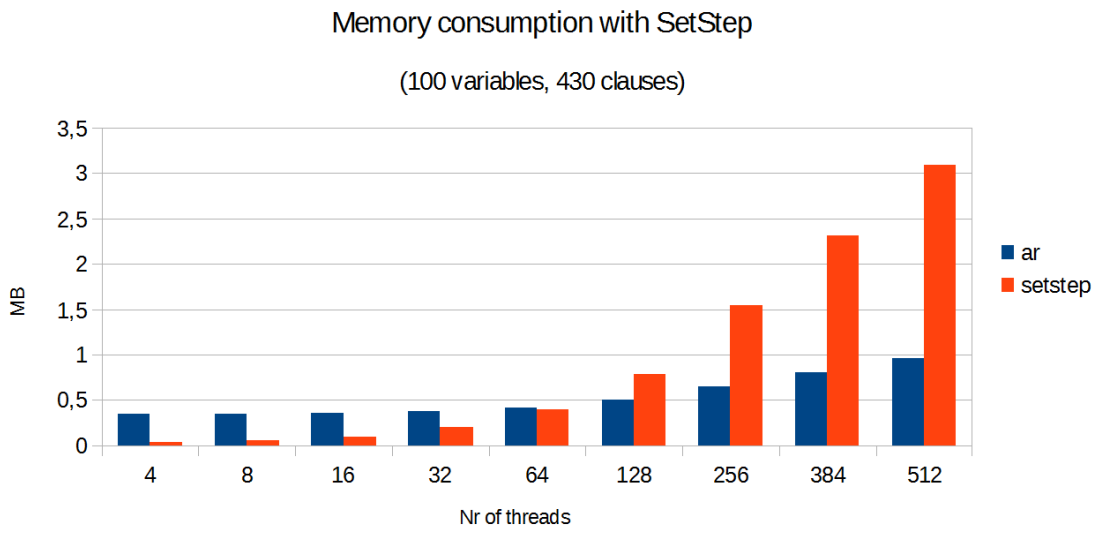


Figure 3.13: Memory consumption of SetStep again ApplyRule with different number of threads

CHAPTER 4

# Conclusion

In the previous chapter the Hobel algorithm was introduced and detailed presented. The main goal of the approach was to use the GPU effectively, and gain advantage against other SAT solvers with the GPU computing. The new data structure and mechanism of solving the problem were a good starting point. The merging tasks can be performed parallel and they have SIMD nature.
The fundamental problem is the exponential growing of necessary memory. The algorithm has got a lot of useful improvements such as MHNF, Filtering and inversion. There were some blind allays like branching, too. In the final state of the algorithm it is no unequivocal, whether the algorithm is useful or not. On the one hand, the Inverse Hobel can not solve big problems fast and effective in the current state. On the other hand, the approach is new; but its data structure and approach offer further opportunities. For example, the introduction of the Filtering had a big impact on the runtime and memory consumption. Another feature, which helps the solver at big problems, could transform the Inverse Hobel into effective competitor.
SetStep is an interesting development. Its original task is to help the Inverse Hobel to solve bigger problems, which could not be solved only with ApplyRule. The main advantage of this is that it can use only minimal extra memory. Unfortunately, the SetStep can not benefit from the higher number of GPU threads. The parallel executed instances need different data, and therefore they are MIMD tasks instead of SIMD. The instances are total independent from each other. There is no central scheduling just as in case of ApplyRule; and therefore they have to race for the resources. This property has a very bad effect on the whole process, when the device has hundreds of instances. Based on the experiences, the SetStep could be useful regarding any future developments as separated sequential solver standing alone - and not as part of the Inverse Hobel.
Instead of SetStep, the ApplyRule should be developed further with smart features to avoid unnecessary MHNF cube generation. Although, there is a common point with the DPLL-based solvers (Filtering is similar to Unit Propagation), the Inverse Hobel has

basically different logic and reasoning. The test results showed that the higher number of threads has beneficial effect on the runtime; and this refers to a GPU-fancier algorithm. Since the Hobel has been designed to the CUDA-capable NVIDIA GPUs and the DPLL-based solvers to the (multicore) CPUs, the changes in the CPU/GPU technologies play important role, too. The breakthrough in the GPU computing [KDK+11] is an important development: the size of the IGP in the newest Intel microarchitecture (Skylake) is bigger than the CPU cores. Using the power of the dedicated cards is getting easier and easier (think of the CUDA) [Nsi]. Algorithms, which do not use the IGP/GPU, will rather create disadvantages, and this fact favors the Hobel. It is possible that in the future a fusion of DPLL-based solvers and Inverse Hobel will use the CPU and IGP parts effectively and solves the SAT problems faster then ever.

# List of Figures

# List of Algorithms

# Index

# Glossary

**CUDA** CUDA is a parallel computing platform and programming model that makes using a GPU for general purpose computing simple and elegant. 2

**shader unit** Shader Unit is basically a very simple CPU, but a modern video card may contain thousands of these units. 2

# Acronyms

**ALU** arithmetic logic unit. 8

**GPGPU** general-purpose computing on graphics processing units. 1

**IGP** integrated graphics processor. 52

**MIMD** multiple instruction, multiple data. 51

**SA** SetStep assignment. 39

**SIMD** single instruction, multiple data. 2

# Bibliography


[Ahm09] Tanbir Ahmed. An implementation of the dpll algorithm. Master's thesis, Concordia University, 2009.

[Ais13] Robert Aistleitner. An Evaluation of Bit-Parallelization applied to Failed Literal Probing. Master's thesis, Johannes Kepler Univeristät Linz, Austria, 2013.

[BB10] Tomá Balyo and Tomá Balyo. *Solving Boolean satisfiability problems*. PhD thesis, Diploma Thesis, Charles University in Prague, 2010.

[Chi12] DarrenM. Chitty. Fast parallel genetic programming: multi-core cpu versus many-core gpu. *Soft Computing*, 16(10):1795–1814, 2012.

[cud] Cuda toolkit documentation. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#introduction`. Accessed: 2016-01-30.

[Das11] Abhranil Das. Process time comparison between gpu and cpu, 2011.

[GS05] Andrei Voronkov Geoff Sutcliffe. 12th international conference. In *Logic for programming, Artificial Intelligence, and reasoning*, 2005.

[Hara] Mark Harris. How to overlap data transfers in cuda c/c++. `http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/`. Accessed: 2015-11-05.

[Harb] Mark Harris. Maxwell: The most advanced cuda gpu ever made. `http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/`. Accessed: 2015-11-05.

[HJPS11] Youssef Hamadi, Said Jabbour, Cédric Piette, and Lakhdar Saïs. Deterministic Parallel DPLL. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):127–132, 2011.

[HKWB12] MarijnJ.H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65. Springer Berlin Heidelberg, 2012.

[HW13] Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel sat solving. *AI Magazine*, 34(2):99–106, 2013.

[KDK+11] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *Micro, IEEE*, 31(5):7–17, Sept 2011.

[NOT05] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[Nsi] Nvidia nsight visual studio edition 4.5 user guide. `http://docs.nvidia.com/nsight-visual-studio-edition/4.5/Nsight_Visual_Studio_Edition_User_Guide.htm`. Accessed: 2015-11-05.

[PKA+06] Stephen Plaza, Ian Kountanis, Zaher Andraus, Valeria Bertacco, and Trevor Mudge. Advances and insights into parallel SAT solving. In *IWLS 2006: 15th International Workshop on Logic and Synthesis*, June 2006.

[SP] Bernd Steinbach and Christian Posthoff. New results for sets of boolean functions.

[TD] Nishant Totla and Aditya Devarakonda. Massive parallelization of sat solvers.

[vdTHB12] Peter van der Tak, MarijnJ.H. Heule, and Armin Biere. Concurrent cube-and-conquer. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 475–476. Springer Berlin Heidelberg, 2012.