EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

GPU Enabled Automated Reasoning

Citation for published version (APA):
Mahmoud, M. O. (Accepted/In press). *GPU Enabled Automated Reasoning*. Eindhoven University of Technology.

Document status and date:
Accepted/In press: 10/03/2022

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 09. mrt.. 2022

GPU Enabled Automated Reasoning

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op donderdag 10 maart 2022 om 16:00 uur

door

Muhammad Osama Mahmoud

geboren te Minya, Egypte

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr. Johan J. Lukkien
1e promotor:	prof.dr. Mark G.J. van den Brand
co-promotor:	dr.ing. Anton J. Wijs
externe leden:	prof.dr.ir. Joost-Pieter Katoen (RWTH Aachen University)
	prof.dr. Armin Biere (University of Freiburg)
	dr.ir. Ana L. Varbanescu (Universiteit Twente)
lid TU/e:	prof.dr. Hans Zantema
	dr.ir. Michel A. Reniers

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

This work is part of the project GEARS with project number TOP2.16.044, which is (partly) financed by the Dutch Research Council (NWO).

In addition, this work made use of the Dutch national e-infrastructure with the support of the SURF Cooperative using grant numbers 2020.024 and EINF-1688.

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2022-02.

A catalogue record is available from the Eindhoven University of Technology Library

ISBN: 978-90-386-5445-4

An electronic version of the thesis is also available online via the link: https://gears.win.tue.nl/publications

Printed by: Gildeprint – www.gildeprint.nl

Cover design by: Walid Osama Mahmoud

Copyright © 2022 by Muhammad Osama Mahmoud. All Rights Reserved. You may not reproduce any content in whole or in part without the consent of the owner.

Contents

List of Figures

List of Tables

List of Acronyms

ALU	Arithmetic and Logic Unit	32
AWS	Amazon Web Services	2
BCE	Blocked Clause Elimination	20
BCP	Boolean Constraint Propagation	4
BDD	Binary Decision Diagram	3
BMC	Bounded Model Checking	1
BVE	Bounded Variable Elimination	20
BVIPE	Bounded Variable-Independent Parallel Elimination	38
CC	Compute Capability	31
CDCL	Conflict-Driven Clause Learning	2
CIM	Computation in Memory	154
CNF	Conjunctive Normal Form	13
CPU	Central Processing Unit	2
CTL	Computational Tree Logicg	2
CUDA	Compute Unified Device Architecture	4
DIMACS	Discrete Mathematics and Theoretical Computer Science	16
DPLL	Davis, Putnam, Logemann, and Loveland	4
DRAT	Deletion Resolution Asymmetric Tautology	65
ERE	Eager Rendundancy Elimination	11
FPGA	Field Programmable Gate Arrays	29
GC	Garbage Collection	67
GFLOPS	Giga FLoating-point Operations Per Second	28

GPU	Graphics Processing Unit	1
HRE	Hidden Rendundancy Elimination	11
IC	Integrated Circuit	27
ID	Thread identifier in a grid	32
IG	Implication Graph	9
IPC	Instructions Per Cycle	28
ITS	Independent Thread Scheduling	32
LBD	Literal Block Distance	19
LCV	Least Constrained Variable	54
LCVE	Least-Constrained Variable Elections	38
LTL	Linear Temporal Logic	2
MC	Model Checking	1
MCV	Most Constrained Variable	54
MDM	Multiple Decision Making	2
OL	Occurrence List	41
OT	Occurrence Table	41
ParaFROST	Parallel Formal Reasoning On SaTisfiability	7
PC	Program Counter	32
PCI-E	Peripheral Component Interconnect Express	53
RES	BVE by resolution	136
RT	Ray Tracing	31
SAT	Satisfiability	1
SIGmA	SAT sImplification on GPU Architectures	6
SIMT	Single-Instruction Multiple-Thread	29
SM	Streaming Multiprocessor	29
SMT	Satisfiability Modulo Theory	3
SpMV	Sparse Matrix-Vector	154
SUB	Subsumption Elimination	20
TDP	Thermal Design Power	31
VCE	Variable-Clause Eliminations	65
VSIDS	Variable State Independent Decaying Sum	5
WL	Watched Literal	5

List of Symbols

k	Counterexample maximum length	3
$\mathcal{S}$	SAT Boolean formula (conjunction of clauses)	13
m	Number of clauses in $\mathcal{S}$	13
C	Clause (disjunction of literals)	13
r	Number of literals in a clause	13
ℓ	Literal (Boolean variable x or its negation $\neg x$)	13
$var(\ell)$	Boolean variable of ℓ	13
$\mathbb{L}$	Domain of all literals	13
$var(\mathbb{L})$	Domain of all variables	14
$\mathbb{L}(\mathcal{S})$	Set of all literals in $\mathcal{S}$	14
$\mathcal{S}_\ell$	All clauses containing ℓ in $\mathcal{S}$	14
E_x	All clauses containing x or $\neg x$ in $\mathcal{S}$	14
$\mathbb{B}$	Domain of the Booleans	17
$\top$	True	17
$\bot$	False	17
$\uparrow$	Undetermined	17
$Free(C)$	All unassigned literals in C	17
α	VSIDS variable score	19
$\mathcal{L}$	Set of learnt clauses	21
$R(\mathcal{S})$	Set of resolvents in $\mathcal{S}$	21
$G_\ell(\mathcal{S})$	Gate clauses containing ℓ in $\mathcal{S}$	21
$H_\ell(\mathcal{S})$	Non-gate clauses containing ℓ in $\mathcal{S}$	21

$\mathcal{A}$	Set of authorised variables	43
h	Histogram function	43
μ	Maximum occurrences of a variable in $\mathcal{S}$	43
$x \mathsf{D} y$	y depends on x	43
$\varPhi$	Set of elected variables	43
$\mathcal{F}$	Set of frozen variables	43
$\mathcal{T}$	Instance of `OT`	44
σ	Set of assigned literals	100
$var(\mathcal{S})$	Set of all variables in $\mathcal{S}$	101
$\mathcal{M}$	Set of multiple decisions	105
$\mathcal{W}_C(\ell)$	Predicate of ℓ being watched in C	108
M	Finite state machine (model)	125
φ	Temporal property	125
$[\![M, \varphi]\!]_k$	SAT formula encoding M and φ up to length k	129
π	Finite trace in M	129
s_i	i-th state in M	129
$\mathcal{I}(s_0)$	Initial set of states	129
$\tau(s_i, s_{i+1})$	Set of transitions from s_i to s_{i+1}	129
$[\![M]\!]_k$	SAT formula encoding M up to k states	129
$[\![\neg\varphi]\!]_k$	SAT formula encoding $\neg\varphi$ up to k states	129
$\mathcal{E}(i)$	Predicate encoding error states at step i	129
$e(s_j)$	j-th error state in M	129
θ	Set of assumptions (unit clauses)	129
$\mathbf{AG}\varphi$	φ holds by every reachable state	130
$\mathcal{M}_\theta$	Set of multiple assumed decisions	133
$\mathcal{M}_L$	Set of multiple non-assumed decisions	133

Preface

My fondness for GPUs started when my father brought me my first computer in 1994, when I was 6 years old. I now realize, without him, I would not be in this position writing my PhD thesis in computer engineering. Back in time, we used to play together, an Indie game called *rescue rover*. I was fascinated with the graphics; even though, the resolution could barely touch the 320 $\times$ 200 quality. My only knowledge about a graphics card was that it commercially sold as a Video Graphics Array (VGA); a technology to process streaming frames or display images. Back then, the quality of a VGA was measured by the size of video memory, since processing power was not on demand. The only thing that mattered to upgrade my desktop as far I remember, was to replace my old 8 MB VGA with a top-notch edition with 32 MB of memory, manufactured by ATI (Radeon now).

Years passed, after which it was time to find a suitable post-secondary school to pursue my higher education. The funny thing is, I chose Petroleum engineering as a field of study to make a lot of money. After three months, I knew back in my mind this is not my track. If I like video games, perhaps I should learn how to make them, and this was where my career really started. So, I chose to study computer engineering. At the third year, we had a course called *parallel computing* taught by Tarek Hagras, but the material focused more on theoretical complexity of parallel patterns and network topologies. I wanted to program some of these patterns in parallel to see its impact on run time. Fate brought me again to GPUs, when I learned about a programming model released by NVIDIA in 2007, called CUDA. Unfortunately, there were not many resources about this new technology, and anyone interested would have to read the programming guide (manual) to get acquainted with its capabilities. During my period as a *teaching assistant*, I had the opportunity to teach the same course, but somehow felt that my help to the students was not enough due to the lack of equipment.

Therefore, I contacted NVIDIA about any funding possibilities to establish my own CUDA lab. They told me about a program that can provide six high-end GPUs in exchange for preparing a complete course about CUDA fundamentals. Even with all my doubts, especially for someone who just started his career in academia, I took my chance and submitted the required material built from scratch with great support from Aiman Tarek. We gratefully received the grant from NVIDIA and started our CUDA education (with help from Omar Shaaban) as part of the parallel computing course. We also dedicated this lab to post-graduate research and to conduct my master project. Now I find myself with a PhD thesis which I would not have managed to complete on my own. For that, I am grateful to my former professors who helped me during my time as a Bachelor and Master student, and working as a TA: Mohammad Moness, Aziza Ibrahim, Hassan Al-Ansary, and Tarek Hagras.

Now, my warmest gratitude goes to my promoters, Anton Wijs and Mark van den Brand. First, let me begin with how I found my PhD project and had the privilege to work with them. After obtaining my MSc degree, I started looking for a PhD project on using GPUs for accelerating SAT-based model checking. I kept searching desperately for two years until finally I found Anton offering a project called GEARS with the aim to perform GPU-enabled automated reasoning about system designs. To the best of my knowledge, after sweeping over 100 universities for PhD positions, this project was the only one available all over the world; I mean, what are the odds. Anton had a very ambitious idea to do robust and fast SAT-based symbolic checking on GPUs and I emphasize 'very' because the SAT problem is truly hard to parallelize (let alone, it is NP-Complete). I had the exact ambition for parallel SAT solving, model checking, and GPU computing in general. Therefore, I got the position and began a new endeavor in my career.

I am grateful to Anton and Mark for giving me absolute confidence that one can achieve anything in life by working hard and being determined. Mark, I will not forget the recommendation letter (two pages long) you wrote me to join the Marktoberdorf summer school. During my work on multi-GPU setup in SAT simplifications, you inspired me with an excellent idea to fairly distribute the workload among n GPUs. Later, we named this scheme 'the ping-pong distribution'. Even though, Mark was not directly responsible for the project's main objectives, I learned a great deal on how to carefully plan, achieve and present my work.

Anton, on the other hand, acted as my main supervisor during my PhD study. However, I felt him being a friend more than a formal advisor, and we are doing this as joint work. I enjoyed our adventure in Oxford and watching the finale of the 2018 world cup during our visit to the FLoC conference. There, after my first paper rejection, I remember you saying: ‘in this line of work, one should have the skin of an elephant’. Despite all disappointments and setbacks, I never gave up, and eventually we had five excellent publications in top-tier conferences and two additional journal papers to be submitted.

In that occasion, we had the privilege to collaborate with Armin Biere on two papers related to SAT inprocessing. Thanks, Armin, for sharing your knowledge and experience. My special thanks to the committee members: Joost-Pieter Katoen, Armin Biere, Michel Reniers, Hans Zantema, and Ana Varbanescu for spending their precious time reading my thesis and providing me with feedback to fix any remaining issues in the text.

Through my time at the Formal Systems Analysis (FSA) and Software Engineering Technology (SET) groups, I presented my work regularly in the colloquia and always received rich feedback from Jan Friso Groote, Hans Zantema, Alexander Serebrenik, Erik de Vink, Wieger Wesselink, and Tim Willemse. This allowed me to improve my research outputs. As a PhD candidate, I was also responsible for organising the SET colloquia and for helping in several courses: programming, SET seminar, program verification techniques, and GPU computing (part of IPA advanced courses). Thanks to Kees Huizing, Anton Wijs, and Alexander Serebrenik for the invaluable experience.

Next, my thanks go to my former office mates in the FSA group, Maurice Laveaux, Olav Bunte, Mauricio Verano Merino, Thomas Neele, Alexander Fedotov, and Ruud van Vijfeijken for the warm welcoming and the inducing work atmosphere. To my new mates from the SET group: Lars van den Haak, Nathan Cassee, Hossain Muctadir, and David Manrique Negrin. I enjoyed our interesting discussions at the coffee machine. Omar al Duhaiby, thanks for our friendly time and talks about learning new languages. I am also thankful to Thomas Neele, Mahmoud Talebi, and Sander de Putter; who made me feel at home when I came to Eindhoven.

I wish to thank all other colleagues from the Model-Driven Software Engineering (MDSE) group for the exquisite time we had together: Rick Erkens, Önder Babur, Fei Yang, Yousra Hafidi, Sangeeth Kochanthara, Ana-Maria Sutii, Dan Zhang, Mark Bouwman, Rodin Aarssen, Nan Yang, Gema Rodriguez, Mahdi Saeedi, Jouke Stoel, Priyanka Karkhanis, Kousar Aslam, Weslley Silva Torres, Felipe Ebert, Lina Ochoa, Miguel Botto Tobar, Arash Khabbaz, and Ferry Timmers. Thanks for the support and wonderful memories you have given to me. My gratitude to Margje Mommers and Agnes van den Reek, for taking care of all the arrangements of my travels and work activities.

There are several people outside TU/e to whom, I would like to send my gratitude. Hassan Ramadan, many thanks for hosting me during my first three days in Eindhoven and letting me taste, for the first time, the local snacks: ‘Kibbeling’ fries and ‘Stroop’ waffles. During the TACAS 2019 conference in Prague, I spent a delightful time discovering the city with my friend Mahmoud Khaled and had cheerful dinners with him, Anton and others.

Last but not least, I wish to thank my family for their outstanding resilience that inspired me along my career. First, to my brother Walid, thanks for designing the thesis’ cover and being there during the toughest times. To my parents, I am truly grateful for your endless prayers and support to complete my education and be successful in my career. To my wife Sara, despite my pitfalls and impatience sometimes, your love and solace, kept me going on the right path. You were always a source of inspiration and motivation.

Muhammad Osama
Den Bosch, December 2021

Chapter **1**

Introduction

"Sometimes it is the people no one can imagine anything of who do the things no one can imagine."

– Alan Turing

The development of complex hardware and software systems is error-prone and costly. Testing can be effective to detect the presence of bugs in these designs, but it cannot prove their absence [Dij72]. One technique that can provide worthful feedback on the correctness of system designs is *model checking* (MC) [CE81, BK08]. It involves exhaustively analysing a system design to determine whether it satisfies desirable functional specifications. However, it is computationally very demanding. Bounded Model Checking (BMC) is currently a contemporary symbolic technique that can analyse large designs in reasonable time. In this thesis, we investigate how Graphics Processing Units (GPUs) can be employed effectively for BMC, narrowing our focus to the reasoning on propositional Satisfiability (SAT) [FM09]. BMC determines whether a model satisfies a certain property expressed in temporal logic, by translating the model checking problem to a SAT problem, for instance. GPUs offer great potential for parallel computation, while keeping power consumption low. However, not all types of computation can trivially be performed on GPUs, in most applications, the algorithms need to be entirely redesigned.

We focus on the simplifications of SAT formulas prior to the solving process (i.e. preprocessing); a strategy that leads to a drastic prune of the formula size, and the search space [OW19a, OW19b]. Next, we present a new SAT solver which rigorously interleaves the search with simplifications (i.e. inprocessing) [OWB21b, OWB21a]. Inprocessing has proven to be powerful in modern SAT solvers, particularly when applied on SAT formulas encoding software and hardware verification problems. The new solver is hybrid, capable of running the parallel part on the GPU while the actual solving will run sequentially on the Central Processing Unit (CPU). Further, we discuss the design aspects of the data structures and the memory management of our parallel implementations, leading to substantial improvements in execution performance.

Concerning the solving part, we extend the Conflict-Driven Clause Learning (CDCL) search algorithm with Multiple Decision Making (MDM) [OW20]. The MDM procedure has the ability to make and propagate multiple decisions at once. Moreover, it is augmented with local search to improve the accuracy in assigning truth values to these decisions [OW21b].

Finally, we describe the integration of the hybrid solver to a state-of-the-art bounded model checker. After optimising further the inprocessing engine and making the solving process incremental, we study the impact of GPU-enabled BMC on software verification [OW21a] using Amazon Web Services (AWS) C99 library.

1.1 Model Checking

MC is a computer-aided method for reasoning about system designs to formally uncover violations of the given specifications. A state-transition diagram can be used to model the design behavior, while the specification is described by temporal properties. A property is formalised in temporal logic such as Linear Temporal Logic (LTL) [Pnu77] or Computational Tree Logic (CTL) [EH83]. Over decades, research and industry have been shifting towards model checking rather than testing in hardware and software verification. Common examples are verifying railway interlocking systems [GH21], nuclear control systems [PBB21], medical imaging [BBC+20], and microprocessor designs [BCRZ99, VB01]. Companies such as Amazon [CKK+18], Microsoft [BCLR04], and Facebook [DFLO19] use and develop MC technology to ensure their products behave functionally correct. MC can be applied with *explicit-state* [CE81] or *symbolic* [BCM+92] techniques. The former suffers the notorious *state-explosion* problem [CG87] as it exhaustively traverses the entire state space, considering all possible values of the state variables (e.g. SPIN [Hol97]). Symbolic techniques can represent groups of states by symbolic expressions (as in the NuSMV checker [CCGR99]), rather than explicitly enumerating states or transitions. The encoding of symbolic expressions may be implemented as Binary Decision Diagrams (BDDs) [Bry86], or propositional formulas. BMC is a prodigious example of symbolic checking without BDDs [BCCZ99]. This thesis addresses the propositional SAT solving and its application in BMC.

1.2 SAT-based Bounded Model Checking

SAT-based BMC was first proposed by Biere *et al.* [BCCZ99, BCC+03] to determine whether a model satisfies a certain property, by translating the application to a SAT or Satisfiability Modulo Theories (SMT) [BSST09] problem. The term *bounded* refers to the fact that the BMC procedure searches for a counterexample to the property, i.e., an execution trace, which is bounded in length by some integer value k. If no counterexample up to this length exists, k can be increased and BMC can be applied again. This process can continue until a counterexample has been found, a user-defined threshold has been reached, or it can be concluded (via k-induction [BCCZ99]) that increasing k further will not result in finding a counterexample. A major advantage of BMC is that for many system designs, it scales better than explicit-state MC [Hol97, WB14]. CBMC (C Bounded Model Checker) [CKL04, KT14] is an example of a successful BMC tool that uses SAT solving. CBMC can check ANSI-C programs. The verification is performed by *unwinding* the loops in the program under verification a finite number of times, and checking whether the bounded executions of the program satisfy a particular safety property [KS16]. These properties may address common program errors, such as null-pointer exceptions and array out-of-bound accesses, and user-provided assertions.

The performance of BMC heavily relies on the performance of the solver. Over the last decade, efficient SAT solvers have been developed and applied for BMC [BCCZ99, DLL62, DKW08, BCMD90, Bro13, Bra11, SG99, SBS96, JS05]. For instance, CBMC has been using MiniSat solver for years to check the satisfiability of the generated formulas.

Leiserson *et al.* [LTE+20] concluded that in the future, advances in computational performance will come from *many-threaded* algorithms that can employ hardware with a massive number of processors such as Graphics processors (GPUs). Even though, effectively *parallelising* BMC is hard. Parallel SAT solving often involves running several solvers, each solving the problem in their own way [ABK+13, HJS09]. For BMC, multiple solvers can be used to solve the problem for different values of the bound k in parallel [ÁSB+11, WNH09, KT11]. However, in these approaches, the individual solvers are still single-threaded. Multi-threaded BMC model checkers have been proposed, such as in [IT20, CRDL20, PMP15], but these address only tens of threads, not thousands.

GPUs have become attractive for general-purpose computing with the availability of the Compute Unified Device Architecture (CUDA) programming model. CUDA is widely used to accelerate applications that are computationally intensive w.r.t. data processing. For instance, GPUs have been applied to accelerate explicit-state MC [BESW10, WB14], bisimilarity checking [Wij15], metaheuristic SAT solving [YIMO15], and SAT-based test generation [OGHM18]. Since BMC relies on SAT solving, the GPU acceleration of BMC should start with the GPU acceleration of SAT solving.

1.3 SAT Solving

The objective of SAT solving is to determine, given a Boolean formula, whether an assignment of truth values to the propositional variables exists, such that the formula evaluates to true. If such an assignment exists, the formula is said to be *satisfiable*; otherwise, it is *unsatisfiable*. SAT solving is performed with two approaches, using complete and incomplete search algorithms. The former always proceeds with a backtracking search procedure until it finds a solution; if it cannot find a solution the (un)satisfiability of the problem is proven. The latter might be able to satisfy a problem or run out of time without giving a determinate decision. The first complete algorithm was introduced back in 1961 by Davis, Putnam, Logemann, and Loveland and was called DPLL after their names [DLL62]. The basic idea behind it is to assign a truth value to a Boolean variable, propagate the effect of that assignment through the formula and then recursively check if the resulting formula is satisfiable. If this is the case, the procedure terminates; otherwise, the recursion assumes the opposite truth value. The propagation of the assignment is known as Boolean Constraint Propagation (BCP).

After nearly 40 years, Marques-Silva and Sakallah proposed the CDCL algorithm [SS99] which introduced a significant improvement over DPLL in the sense that the search space can be pruned by *learning* so-called *conflict* clauses. Learning these clauses prevents the solver from repeating bad assignments. Modern SAT solvers employ CDCL, including (but not limited to) Grasp [SS99], Chaff [MMZ$^+$01], BerkMin [GN07], MiniSat [ES03a], Glucose [AS09], and CaDiCaL [BFFH20]. Grasp was the first tool applying CDCL, after which Chaff introduced the so-called *two watched literals* (*2*-WL) optimisation and the Variable State Independent Decaying Sum (VSIDS) decision heuristic. BerkMin and MiniSat introduced further implementation and heuristics optimisations. The authors behind Glucose presented robust clause deletion and restart heuristics [AS12]. The CaDiCaL solver introduced the effective use of SAT simplifications [SP04, EB05, BJK21] as an in-processing technique during the solving process [JHB12].

Acceleration of DPLL on a GPU has been done, in which some parts of the search tree were implemented in parallel and the remainder is handled by the CPU [PDFP15]. Incomplete approaches are more amenable to be executed entirely on a GPU, e.g., an approach using metaheuristic algorithms [YIMO15]. Throughout the thesis, we propose novel parallel algorithms and various optimisations in a new SAT solver which is capable of running parts of the solving procedure on one or more GPUs.

1.4 Contributions and Thesis Hierarchy

Recall that the running performance of a SAT-based model checker relies heavily on the performance of the SAT solver. The main goal of this thesis is to accelerate SAT solving targeting GPU architectures and apply this on BMC.

> A journey of a thousand miles begins with a single step.

The first step in our GPU-accelerated BMC journey began with *parallel SAT simplifications*. Along the road we stumbled on some obstacles, attempting to parallelise the CDCL search algorithm itself, but we ended up improving the original sequential version which is something we never expected. Moreover, we showed how efficiently GPU-accelerated simplifications work on the GPU using its hardware capabilities and together with the CPU part without sacrificing any achieved speedup in communications. These contributions are discussed briefly below.

First, Chapter 2 gives the mathematical notions for all contributions and background including *SAT encoding*, history of *SAT solving*, and *SAT simplifications*. Furthermore, we show how these notions relate to each other. Next, Chapter 3 prepares the reader to understand the GPU architecture and the

CUDA programming model. The main body of the thesis will take the reader deeply through these topics:

⋆ How can GPUs be effectively employed to solve SAT problems that stem from real-world applications, e.g., hardware and software verification?

⋆ How can this be extended to perform SAT-based BMC on GPUs?

Recently, the authors of [HW12, BS18] discussed the current main challenges in parallel SAT solving. One of these challenges concerns the parallelisation of SAT simplification in modern SAT solvers. Massively parallel computing systems such as GPUs offer great potential to speed up computations, but to achieve this, it is crucial to engineer new parallel algorithms and data structures from scratch to make optimal use of those architectures. Therefore, we pose the following research question:

> **RQ1:** How can GPUs be employed to perform scalable parallel SAT simplifications?

Chapter 4 introduces a solution for this question in which parallel algorithms are proposed for various techniques widely used in SAT simplification. We discuss the various performance aspects of the proposed implementations and data structures, dealing with the main challenges in CPU-GPU memory management and how to address them. In a nutshell, we aim to effectively simplify SAT formulas, even if they are extremely large, in only a few seconds using the massive computing capabilities of GPUs. Building on our results for **RQ1**, we ask ourselves if the proposed parallel simplifications scale up as well on multiple GPUs:

> **RQ2:** Can we run parallel simplifications on a multi-GPU environment, with or without sharing information?

We propose in Chapter 4 a generalisation of all developed algorithms to distribute simplification work over single or multiple GPUs, if these are available in a single machine. This work is implemented in a new tool called *SAT sImplification on GPU Architectures* (SIGmA). We discuss the experimental results of SIGmA and its impact on different SAT solvers.

Next, we investigate the feasibility of applying simplifications frequently within SAT solving which is known as *inprocessing* [JHB12]:

> **RQ3:** Is it possible to run parallel simplifications regularly during CDCL SAT solving, considering the growing size of the formula by *learning* new clauses and transferring data back and forth to the CPU?

Since 2013, the leading SAT solvers in the SAT competition[1]all use inprocessing. However, applying inprocessing frequently can still be a bottleneck, i.e., for hard or large formulas. In Chapter 5, we discuss the first attempt to parallelise inprocessing on GPU architectures. As memory is a scarce resource in GPUs, we present new space-efficient data structures and devise a data-parallel garbage collector. It runs in parallel on the GPU to reduce memory consumption and improves memory access locality. Our new parallel variable elimination algorithm is twice as fast as the one presented in Chapter 4. Furthermore, a new SAT solver called *Parallel Formal Reasoning On SaTisfiability* (ParaFROST) is devised from scratch based on the heuristics of CaDiCaL solver [BFFH20] bolstered with the GPU-accelerated inprocessing. This imposes the next research question:

> **RQ4:** Can we harness the intrinsic CUDA capabilities such as intra-warp communications and concurrent streams for pushing ParaFROST performance beyond its limits?

The compute capabilities of modern GPUs rapidly evolve, and some of the ideas suggested in previous work should be reconsidered for today's GPUs. Thereby, we continue improving ParaFROST by leveraging these capabilities. In Chapter 5, we discuss the implementation of warp-level primitives in accelerating heavy-used operations like *insertions* to data vectors and parallel reduction with *shuffling*.

Correctness of the implemented algorithms on the GPU is crucial for the soundness of our tools, especially if they are used in critical applications such as model checkers [OW21a]. If the solver claims that a formula is satisfiable, the generated solution (model) can be checked linearly in the size of the formula. However, if a solver declares a formula is unsatisfiable (i.e. has no solutions), there is no guarantee that the GPU code is sound and correct. Thus, this leads to the following question:

[1] http://www.satcompetition.org/

RQ5: Can we generate a clausal proof for the GPU code implemented in our solver ParaFROST?

The answer is yes and we can do that quite efficiently without causing any penalty to the solver's overall performance. Clausal proofs are generated by the solver to express its progress in both simplifications and search in the form of *lemmas* where each lemma is a sequence of literals. Essentially, a lemma expresses the outcome of a resolution step or clause deletion. The generated lemmas are verified via an external tool called drat-trim. In Chapter 5, we explain in detail how this can be achieved effectively, without adding a significant overhead to the GPU memory or time.

Regarding parallel SAT solving, most papers in the existing literature are focused at the *portfolio* approach [ABK+13] where multiple instances of a SAT solver run simultaneously in solving the same problem with different configurations. On the other hand, the *cube-and-conquer* approach [HKWB11] tries to decompose the original formula into many smaller subformulas then solve all of them in parallel via multiple solvers. Therefore, the next question is:

RQ6: Can CDCL search be partially or entirely parallelizable?

The answer is *yes* to "partially" and *no* to "entirely". During our work on implementing SAT solving on the GPU, we proposed an algorithm called *multiple decision making* (MDM) which is explained in detail at Chapter 6. MDM is capable of making thousands, even millions of decisions in the CDCL decision-making step that are independent of each other and hence can be assigned and propagated in parallel. Nonetheless, doing so in parallel is actually slower than the sequential propagation. Additionally, parts of the CDCL search are inherently sequential and cannot be run in parallel. Later, we observed that applying MDM sequentially has a positive impact on large formulas, particularly stemming from verification problems.

In our last research question, we address the integration and the impact of a GPU-accelerated solver on existing BMC tools:

RQ7: How can GPUs be employed effectively to speed up BMC?

The structure of typical BMC SAT formulas suggests that GPU pre- and in-processing will be effective due to large variable redundancy. Chapter 7 addresses the integration of ParaFROST with the CBMC checker and applying

ParaFROST on programs from the Core C99 package of Amazon Web Services [Ama21].

The new extension provides an interface with CBMC that is implemented in C++. Moreover, it offers patches to CBMC in order to allow configuring ParaFROST via a configuration file. This file contains all options supported by ParaFROST. The latter, the interface, and all patches are part of the framework GPU4BMC. Due to the massive amount of redundancies in BMC formulas, the acceleration on the GPU offers particular challenges. We explain these challenges and provide effective solutions in Chapter 7.

Additionally, to support BMC, ParaFROST should be an *incremental* solver, i.e., it must exploit that a number of very similar SAT problems are solved in sequence [ES03b, WKS01]. In Chapter 7, we discuss how that can be implemented in ParaFROST without causing any harm on parallel inprocessing.

Finally, we conclude the thesis by discussing the strengths and weaknesses of the proposed contributions in Chapter 8 and suggest future work.

1.5 How to Read the Thesis

The work presented in this thesis consists of three main parts. The first part concerns the GPU acceleration of *preprocessing* and *inprocessing* in Chapters 4 and 5, respectively. The second part deals with SAT solving itself using CDCL search and is explained in Chapter 6. The last part introduces the application of the former parts to BMC which is discussed in Chapter 7. Figure 1.1 visualises the flow of our chapters by an *implication graph* (IG) [APT79] where the vertices represent the thesis chapters and edges indicate whether to move to the next chapter or backtrack non-chronologically to a previous vertex. Notice that an implication here implies that the reader cannot advance to the next chapter without reading the current one (finished reading means **true**).

1.6 Origin of the Chapters

The first part (Chapters 4 and 5) partially originates from the following three publications:

[OW19a] M. Osama and A. Wijs. Parallel SAT Simplification on GPU Architectures. In *Proc. of TACAS (Apr. 2019), Prague, Czech Republic*, volume 11427 of *LNCS*, pages 21–40. Springer. doi:10.1007/978-3-030-17462-0_2

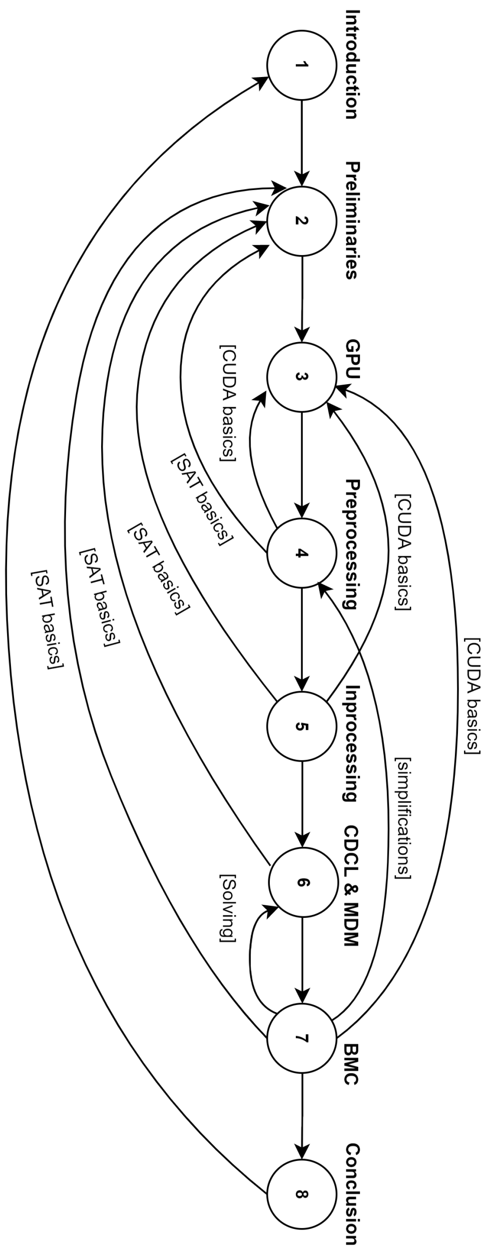


Figure 1.1: Thesis implication graph

[OW19b] M. Osama and A. Wijs. SIGmA: GPU Accelerated Simplification of SAT Formulas. In *Proc. of IFM (Dec. 2019), Bergen, Norway*, volume 11918 of *LNCS*, pages 514–522. Springer. doi:10.1007/978-3-030-34968-4_29

[OWB21b] M. Osama, A. Wijs, and A. Biere. SAT Solving with GPU Accelerated Inprocessing. In *Proc. of TACAS (Mar. 2021), Luxembourg*, volume 12651 of *LNCS*, pages 133–151. Springer. doi:10.1007/978-3-030-72016-2_8

[OWB21a] M. Osama, A. Wijs, and A. Biere. Certified SAT Solving with GPU Accelerated Inprocessing. 2021. To be submitted

The second publication extends the first one with multi-GPU support for SAT preprocessing, a new simplification technique called *hidden redundancy elimination* (HRE), and extensive evaluation of our simplifier SIGmA. The last publication discusses how parallel simplifications can be applied frequently within SAT solving with efficient data structure and parallel garbage collection. Moreover, the *eager redundancy elimination* (ERE) which is a variant of HRE is presented with various optimisations to deal with learnt clauses effectively. Part of the contributions in Chapter 5 concerning GPU proof generation and finding general gate definitions is recently submitted as a journal manuscript in [OWB21a]. The work discussed in these chapters is based on the SAT encoding, SAT simplifications, and GPU programming model in Chapters 2 and 3 (see Figure 1.1). The former summarises all the preliminaries in the above publications.

The second part of this thesis (Chapter 6) partially originates from the following manuscripts:

[OW20] M. Osama and A. Wijs. Multiple Decision Making in Conflict-Driven Clause Learning. In *Proc. of ICTAI (Nov. 2020), Baltimore, USA*, pages 161–169. IEEE. doi:10.1109/ICTAI50040.2020.00035

[OW21b] M. Osama and A. Wijs. Multiple Decision Making in CDCL SAT Solvers. 2021. To be submitted

The first publication discusses the basic MDM extension to CDCL search dealing with MiniSat and Glucose heuristics. However, Chapter 6 has additional content addressing the local search and the usage of multiple decision queues. This work has been recently submitted as a journal manuscript in [OW21b]. Overall, the CDCL search is explained in Chapter 6. The basic SAT notations and unit propagation are defined in Chapter 2.

The last part of this thesis concerns the application of GPU SAT solving on bounded model checking which originates partially from the following publication:

[OW21a] M. Osama and A. Wijs. GPU Acceleration of Bounded Model Checking with ParaFROST. In *Proc. of CAV (Jul. 2021), USA*, volume 12760 of *LNCS*, pages 447–460. Springer. doi:10.1007/978-3-030-81688-9_21

The preliminaries of pre- and inprocessing are discussed in Chapter 2; while the basics of CDCL and MDM notations are explained in detail in Chapter 6 (check Figure 1.1).

Finally, two sequential configurations of ParaFROST (executed by a single CPU thread) were among the **top ten** solvers at the SAT competition 2021. The solver description is available through the following technical report:

[OW21c] M. Osama and A. Wijs. ParaFROST at the SAT Race 2021. In *Proc. of SC (2021)*, volume B-2021-1 of *Report Series B*, pages 32–34. University of Helsinki. URL `http://hdl.handle.net/10138/333647`

Chapter **2**

Preliminaries

"A mathematical formula should never be owned by anybody! Mathematics belong to God."

– Donald Knuth

This chapter gives the basic mathematical notions and the heuristics that are used in the thesis. These include the *SAT encoding*, *SAT simplifications*, *history of SAT solving*, common heuristics used in CDCL SAT solvers.

2.1 SAT Encoding

Given a Boolean formula, SAT is the problem of finding an assignment of Boolean values to the propositional variables in that formula, such that the formula evaluates to true. If such an assignment exists, the formula is said to be *satisfiable*; otherwise, it is *unsatisfiable*.

A SAT Boolean formula $\mathcal{S}$ is typically converted to Conjunctive Normal Form (CNF) before it is solved by current SAT solvers. A CNF formula is a conjunction of clauses $\bigwedge_{i=1}^{m} C_i$ where each clause C_i is a disjunction of literals $\bigvee_{j=1}^{r} \ell_j$ such that $(m \geq 1)$ and $(r \geq 1)$. A literal is a Boolean variable x or its negation $\neg x$. For a literal ℓ, $var(\ell)$ denotes the referenced variable, i.e., $var(x) = x$ and $var(\neg x) = x$. The domain of all literals is $\mathbb{L}$. The domain of

Table 2.1: CNF representations for logical operators

Logical Operator	**Definition**	**CNF Representation**
implication	$y \rightarrow a$	$\{\neg y, a\}$
equivalence	$y \leftrightarrow a$	$\{\{y, \neg a\}, \{\neg y, a\}\}$
not	$y \leftrightarrow \neg a$	$\{\{y, a\}, \{\neg y, \neg a\}\}$
and	$y \leftrightarrow a \wedge b$	$\{\{\neg y, a\}, \{\neg y, b\},$ $\{\neg a, \neg b, y\}\}$
or	$y \leftrightarrow a \vee b$	$\{\{y, \neg a\}, \{y, \neg b\},$ $\{a, b, \neg y\}\}$
xor	$y \leftrightarrow a \otimes b$	$\{\{y, \neg a, b\}, \{y, a, \neg b\},$ $\{\neg y, a, b\}, \{\neg y, \neg a, \neg b\}\}$
$y \leftrightarrow$ if a then b else c	$((a \wedge b) \rightarrow y) \wedge$ $((a \wedge \neg b) \rightarrow \neg y) \wedge$ $((\neg a \wedge c) \rightarrow y) \wedge$ $((\neg a \wedge \neg c) \rightarrow \neg y)$	$\{\{\neg a, \neg b, y\}, \{\neg a, b, \neg y\},$ $\{a, \neg c, y\}, \{a, c, \neg y\}\}$

all variables is $var(\mathbb{L})$. With $\mathbb{L}(\mathcal{S})$, we denote all literals in $\mathcal{S}$. We interpret a clause C as a set of literals $\{\ell_1, \ldots, \ell_r\}$ representing the clause $\ell_1 \vee \ldots \vee \ell_r$, and a SAT formula $\mathcal{S}$ as a set of clauses $\{C_1, \ldots, C_m\}$ representing the formula $C_1 \wedge \ldots \wedge C_m$. Furthermore, we denote the set of all clauses of $\mathcal{S}$ in which ℓ occurs by $\mathcal{S}_\ell = \{C \in \mathcal{S} \mid \ell \in C\}$. The set of clauses $E_x = \mathcal{S}_x \cup \mathcal{S}_{\neg x}$ is called the *environment* of x.

The most common CNF translation of SAT formulas was introduced by Tseitin in 1983 [Tse83] which generates a linear number of clauses at the cost of introducing a linear number of new variables, and generates an equivalent satisfiable formula (i.e. satisfiable if and only if the original formula is satisfiable). Tseitin encodings work by adding new variables to the CNF formula, one for every sub-formula of the original formula. The resulting CNF encoding is linear in the size of the original formula as long as the Boolean operators that appear in the formula have linear clausal encodings. Table 2.1 gives the clausal representations of the most common operators in CNF encoding. A gate definition of y is written as $y \leftrightarrow f(v_1, \ldots, v_n)$. The simplest example is the AND gate $y \leftrightarrow a \wedge b$. The CNF translation of the xor gate $(y \leftrightarrow a \otimes b)$ in Table 2.1 can be derived as follows:

$$\begin{aligned}
y \leftrightarrow a \otimes b &= (y \leftrightarrow (a \vee b) \wedge (\neg a \vee \neg b)) \\
&= \underbrace{\big(\neg y \vee ((a \vee b) \wedge (\neg a \vee \neg b))\big) \wedge \big(y \vee \neg((a \vee b) \wedge (\neg a \vee \neg b))\big)}_{\text{equivalence translation}} \\
&= \underbrace{\big((\neg y \vee a \vee b) \wedge (\neg y \vee \neg a \vee \neg b)\big)}_{\text{distributing “}\neg y\text{”}} \wedge \big(y \vee \underbrace{(\neg(a \vee b) \vee \neg(\neg a \vee \neg b))}_{\text{dist. “}\neg\text{”}}\big) \\
&= \big((\neg y \vee a \vee b) \wedge (\neg y \vee \neg a \vee \neg b)\big) \wedge \big(y \vee (\underbrace{(\neg a \wedge \neg b)}_{\text{dist. “}\neg\text{”}} \vee \underbrace{(a \wedge b)}_{\text{dist. “}\neg\text{”}})\big) \\
&= \big((\neg y \vee a \vee b) \wedge (\neg y \vee \neg a \vee \neg b)\big) \wedge \\
&\qquad \big(y \vee \underbrace{((\neg a \vee a) \wedge (\neg a \vee b) \wedge (\neg b \vee a) \wedge (\neg b \vee b))}_{\text{dist. “}(\neg a\ \wedge\ \neg b)\text{”}}\big) \\
&= \big((\neg y \vee a \vee b) \wedge (\neg y \vee \neg a \vee \neg b)\big) \wedge \big(y \vee ((\neg a \vee b) \wedge (\neg b \vee a))\big) \\
&= \big((\neg y \vee a \vee b) \wedge (\neg y \vee \neg a \vee \neg b)\big) \wedge \underbrace{\big((y \vee \neg a \vee b) \wedge (y \vee \neg b \vee a)\big)}_{\text{dist. “}y\text{”}}
\end{aligned}$$

Similarly, using the CNF representation of the implication translation (see Table 2.1), the *if-then-else* relation is encoded as follows:

$$\begin{aligned}
y \leftrightarrow \text{if } a \text{ then } b \text{ else } c &= ((a \wedge b) \rightarrow y) \wedge ((a \wedge \neg b) \rightarrow \neg y) \wedge \\
&\quad\ ((\neg a \wedge c) \rightarrow y) \wedge ((\neg a \wedge \neg c) \rightarrow \neg y) \\
&= \underbrace{(\neg(a \wedge b) \vee y)}_{\text{impl. transl.}} \wedge \underbrace{(\neg(a \wedge \neg b) \vee \neg y)}_{\text{impl. transl.}} \wedge \\
&\quad\ \underbrace{(\neg(\neg a \wedge c) \vee y)}_{\text{impl. transl.}} \wedge \underbrace{(\neg(\neg a \wedge \neg c) \vee \neg y)}_{\text{impl. transl.}} \\
&= (\underbrace{\neg a \vee \neg b}_{\text{dist. “}\neg\text{”}} \vee y) \wedge (\underbrace{\neg a \vee b}_{\text{dist. “}\neg\text{”}} \vee \neg y) \wedge \\
&\quad\ (\underbrace{a \vee \neg c}_{\text{dist. “}\neg\text{”}} \vee y) \wedge (\underbrace{a \vee c}_{\text{dist. “}\neg\text{”}} \vee \neg y)
\end{aligned}$$

Example 2.1. Consider the following formula $\mathcal{S} = ((a \wedge b) \to \neg c)$. By introducing new variables to all subformulas, the formula is decomposed into

$$x_1 \leftrightarrow a \wedge b \qquad x_2 \leftrightarrow \neg c \qquad x_3 \leftrightarrow x_1 \to x_2$$

The Tseiting encoding is then obtained by taking the conjunction of all subformulas shown above with the main formula itself as follows: $\mathit{Tseitin}(\mathcal{S}) = x_3 \wedge (x_1 \leftrightarrow a \wedge b) \wedge (x_2 \leftrightarrow \neg c) \wedge (x_3 \leftrightarrow x_1 \to x_2)$. Finally, the equivalent CNF formula is generated by substituting the logical operators for their clausal representations (see Table 2.1). For instance, $x_2 \leftrightarrow \neg c$ is expressed by the clause set $\{\{\neg x_2, \neg c\}, \{x_2, c\}\}$.

The CNF representation has the advantage of being simple to encode and implement in a common file format. A widely used format was developed for the DIMACS challenge of 1993 [JT96], and it has been adopted ever since. This has allowed the research community to generate and store an enormous range of SAT benchmark problems over the years in the regular SAT solving competitions[2]. The file may contain optional comment lines that begin with the letter `c` followed by a line of the form `p cnf variables clauses`. The `variables` and `clauses` state the total number of variables and clauses in the file respectively. The variables stored in the file are numbered from 1 to `variables` in integer format. The rest of the file contains the clauses, each of which is represented by a list of non-zero integers, followed by a zero to mark the end of the clause. The integers can be separated by any amount of white spaces. Furthermore, a positive integer v represents a literal x_v, while $-v$ represents the negated literal $\neg x_v$. For instance, the following line (`2 -3 1 0`) represents the clause $\{x_2, \neg x_3, x_1\}$. The order in which the numbers are written is irrelevant, as the logical or is commutative and associative.

In this thesis, we interpret constants and data structures with all-capital letters in the format `CONSTANT` or `STRUCT`. All arrays/lists and structure members are named in the format `array` or `member`. Function and solver names are written as FUNCTION or SOLVER. The variables defined within the algorithms have the font shape *variable*.

2.2 SAT Solving

SAT solving can be performed with two approaches, using complete and incomplete search algorithms. The former always proceeds with a backtracking search

[2] http://www.satcompetition.org/

procedure until it finds a solution; if it cannot find a solution the (un)satisfiability of the problem is proven. The latter might be able to satisfy a problem or run out of time without giving a determinate decision. The first complete algorithm was the DPLL algorithm [DLL62]. The basic idea behind it is to assign a truth value to a Boolean variable, propagates the effect of that assignment through the formula and then recursively checks if the resulting formula is satisfiable. If this is the case, the procedure terminates; otherwise, backtracking is applied and the recursion assumes the opposite truth value. Propagating the effects of an assignment in a CNF formula is called Boolean Constraint Propagation (BCP).

Let the domain of the Booleans be $\mathbb{B} = \{\top, \bot, \uparrow\}$, where **true** is represented by $\top$, **false** by $\bot$, and *undetermined* by $\uparrow$. We have $\neg\top = \bot$, $\neg\bot = \top$, and $\neg\uparrow = \uparrow$. An *assignment* ℓ refers to assigning $\top$ to literal ℓ and is denoted by $\ell \models \top$. An unassigned literal is denoted by $\ell \models \uparrow$. An assignment is called a *decision* iff it is not implied by another assignment. We refer to a clause being satisfied by $C \models \top$ iff $\exists(\ell \in C).\ell \models \top$. A clause C is called *unit* iff $\exists(\ell \in C).\ell \models \uparrow$ and $\forall(\ell' \neq \ell \in C).\ell' \models \bot$. An assignment to that unit is called an *implication.* A unit clause can be implied by another decision or implication. Further, with $\mathit{Free}(C) = \{\ell \in C \mid \ell \models \uparrow\}$, we refer to the set of unassigned literals in C.

Definition 2.1 (Boolean Constraint Propagation). For a CNF formula $\mathcal{S}$, BCP simplifies $\mathcal{S}$ based on unit clauses by repeating the following until fixpoint: If there is a unit clause $C \in \mathcal{S}$, set the literal $\ell \in \mathit{Free}(C)$ to $\top$ for satisfying all clauses that have the assignment ℓ, and set all occurrences of the complementary literal $\neg\ell$ in $\mathcal{S}$ to $\bot$. We call the resulting formula BCP($\mathcal{S}$). A formula $\mathcal{S}$ is satisfied iff $\forall(C \in \text{BCP}(\mathcal{S})).C \models \top$. If for some clause $C \in \text{BCP}(\mathcal{S})$, we have $\mathit{Free}(C) = \emptyset$ and $C \models \bot$, we say that BCP derives a *conflict* in $\mathcal{S}$.

Example 2.2. Consider the formula $\mathcal{S} = \{\{a, b, c\}, \{\neg b, a\}, \{\neg c, \neg a\}, \{\neg b, \neg c\}\}$. A DPLL solver will pick a variable, say a and set it to an arbitrary truth value (in this example $\top$ is assigned). As a result, the first two clauses are satisfied, and the literal $\neg a$ from the third clause is set to $\bot$, producing a new unit clause. Therefore, BCP sets $\neg c$ to $\top$, by which the last clause is satisfied. Since all clauses are now satisfied, we declare $\mathcal{S}$ satisfiable.

The CDCL algorithm [SS99] introduced a significant improvement over DPLL in the sense that the search space can be pruned by *learning* so-called *conflict* clauses. Learning these clauses prevents the solver from repeating bad assignments. Next, in Chapters 4 and 5, we explain how effectively pre- and inprocessing are integrated with the CDCL search algorithm. For this, we consider clauses to be either `LEARNT` or `ORIGINAL`. A `LEARNT` clause is added to the formula by the CDCL clause learning process, and an `ORIGINAL` clause is part of the formula from the very start. Furthermore, each assignment is associated with a *decision level* that acts as a time stamp, to monitor the order in which assignments are performed. The first assignment is made at decision level 1. When propagating, implications are given the same decision level as the most recent assignment.

Example 2.3. Consider the formula in Example 2.2. The variable a is assigned $\top$ at decision level 1. Hence, the new implication $\{\neg c\}$ resulting from falsifying $\neg a$ in the third clause is, in turn, assigned at level 1.

Regarding the incomplete algorithms to solve a SAT formula, one could use the WalkSAT strategy proposed in [SK93]. WalkSAT tries to find the best truth values that satisfy the most unsatisfied clauses in the formula by randomly flipping literals weighed by their scores. A literal score is the number of clauses that will change from satisfied to unsatisfied if that literal is flipped. The *random walking* starts by picking a random clause which is unsatisfied by the current assignment and flipping a literal within that clause. A literal is chosen with a probability determined by the fewest previously satisfied clauses if the literal is assigned **true** becoming unsatisfied. The local search can be stopped once the number of clauses inspected reach some threshold value.

2.2.1 Optimisations

Two-Watched Literal Optimisation

The BCP procedure as explained in Definition 2.1 is considered the hotspot of a SAT solver. Therefore, the authors in [MMZ+01] introduced the so-called *two-watched literals* (*2-WL*) optimisation which significantly reduces the memory accesses and the effort spent in propagating assignments. The idea is that in each clause $C \in \mathcal{S}$, two unassigned literals $\ell_1, \ell_2 \in C$ are marked as *watched*, and as soon as one is set to **false**, another unassigned literal is selected for watching, unless there are no unassigned, unwatched literals left. The intuition is, if ℓ is assigned **true**, all clauses $C \in \mathcal{S}_\ell$ become satisfied. Thus, checking them can be completely avoided and turn the focus to the watched clauses among the set $\mathcal{S}_{\neg\ell}$ in which implications are likely to be produced.

Clause Deletion Policy

Keeping learnt clauses forever will eventually exhaust all available memory to the solver and drastically slows down BCP procedure. Therefore, it is crucial to frequently delete a fraction of these clauses. All modern SAT solvers now use the *literal block distance* (LBD or GLUCOSE level) as a metric of the clause importance [AS09]. LBD is the number of distinctive decision levels in a learnt clause. Clauses with an LBD value of 2 are called *glue* clauses and never deleted. A percentage of clauses with higher LBD values can be deleted every some interval.

2.2.2 Heuristics

Search Restarts

Another optimization used by current state-of-the-art CDCL solvers is called a restart. A restart is a termination of the current search loop and undoing either all assignments [GSK98] or *reusing* some of them [vdTRH11]. Even though, the search space is terminated, current learnt clauses are always preserved. The frequency of restarts can be configured during the solving time and initialised to some interval by the algorithm input. For example, the MINISAT solver applies the geometric [Wal99] and Luby [LSZ93] restarts which are scaled by the current number of conflicts. One drawback of this technique is that it can cause the exploration to bail out early when it is close to finding a solution to a satisfiable formula. Later, as reported in [AS12], this was improved by blocking restarts whenever the current search seems to be close to a solution.

Decision Heuristics

The *decision making* step in CDCL solvers determines which literals should be selected and assigned **true** for the next decisions. Actually, in existing solvers, a *variable* is selected and specific heuristics are used to set it either to **true** or **false**. For the latter, an effective way is to reuse the last phase of a variable x saved during the backtracking step [PD07]. Saved phases can be *rephased* by flipping, randomly set, or improved by stochastic local search every once in a while [BFFH20]. Furthermore, WALKSAT can be combined with CDCL search to improve the quality of the picked decisions.

Regarding variable selection, one could use Variable State Independent Decaying Sum (VSIDS) [MMZ$^+$01] to improve the quality of the decisions made. In VSIDS, each variable has a counter (sum), denoted as α, which initially has the value 0. Once a conflict clause is deduced, the α-values of all variables it refers to are incremented (i.e. *bumped*), and the α-values of all variables in the formula are divided by some constant (simulating decay). Each time a decision must be made, the variable x with the highest α is selected. The BerkMin [GN07] solver improves VSIDS by incrementing the α-values of those variables referenced by any clause involved in the conflict analysis.

2.3 SAT Simplifications

Simplifying SAT problems prior to solving (preprocessing) [BJK21] or during the solving (inprocessing) [JHB12] has proven its effectiveness in modern CDCL SAT solvers [ES03a, BFFH20, AS09]. It can achieve larger reductions in reasonable processing time. In this section, we formally describe the following simplification methods: *bounded variable elimination* (BVE) [SP04, EB05], *subsumption elimination* (SUB) [EB05, Zha05], and *blocked clause elimination* (BCE) [JBH10].

In parallel computing, *confluence* is a very beneficial property of a parallel algorithm. That means, for some algorithms, all threads can reach a fixpoint (i.e., produce the same formula). Therefore, we address confluence of the simplifications discussed in this chapter, and also in Chapter 4, when we focus on the parallel execution of the simplifications.

2.3.1 Bounded Variable Elimination

BVE can remove variables completely from the CNF formula by trivially eliminating *pure literals* [DLL62], applying the *resolution rule* [Kul99, DLL62, SP04] or *gate-equivalence reasoning* [Li00, OGMS02, EB05].

Definition 2.2 (Pure literal)**.** For a formula $\mathcal{S}$, a literal ℓ is called *pure* iff $\mathcal{S}_{\neg\ell} = \emptyset$. This literal can be eliminated from $\mathcal{S}$, resulting in the new formula $\mathcal{S}' = \mathcal{S} \setminus \mathcal{S}_\ell$.

Definition 2.3 (Resolution rule)**.** Given two clauses C_1 and C_2 such that for some variable x, we have $x \in C_1$ and $\neg x \in C_2$. We represent the application of the rule w.r.t. some variable x using a *resolving operator* $\otimes_x$ on C_1 and C_2. Given that $x \in C_1$ and $\neg x \in C_2$, the operator $\otimes_x$ is defined as

$$C_1 \otimes_x C_2 = (C_1 \setminus \{x\}) \cup (C_2 \setminus \{\neg x\})$$

The result of applying the rule is called the *resolvent* [SP04]. Moreover, The $\otimes_x$ operator can be extended to resolve sets of clauses w.r.t. variable x.

Definition 2.4 (Resolvents set)**.** For a formula $\mathcal{S}$, let $\mathcal{L} \subset \mathcal{S}$ be the set of learnt clauses when we apply the resolution rule during the solving procedure. The set of new resolvents is then defined as

$$R(\mathcal{S}) = \{C_1 \otimes_x C_2 \mid C_1 \in \mathcal{S}_x \setminus \mathcal{L} \wedge C_2 \in \mathcal{S}_{\neg x} \setminus \mathcal{L} \wedge C_1 \otimes_x C_2 \not\models \top\}$$

Notice that the learnt clauses can be ignored [JHB12] (i.e., in practice, it is not effective to apply resolution on learnt clauses). The last condition expresses that a resolvent should not be a tautology.

Definition 2.5 (tautology)**.** A clause C is called a *tautology* (i.e. $C \models \top$) iff $\exists x.\{x, \neg x\} \subseteq C$.

The resolvents set $R_x(\mathcal{S})$ replaces $\mathcal{S}$, producing a logically-equivalent SAT formula. A *bounded* version of variable elimination restricts replacing $\mathcal{S}$ by $R_x(\mathcal{S})$ iff $|R_x(\mathcal{S})| \leq |E_x|$.

In gate-equivalence reasoning, we substitute eliminated variables by deduced logical equivalent expressions. Combining gate equivalence reasoning with the resolution rule tends to result in smaller formulas compared to only applying the resolution rule [JHB12, EB05]. Let $G_\ell(\mathcal{S})$ be the gate clauses having ℓ as the gate output and $H_\ell(\mathcal{S})$ the non-gate clauses, i.e., clauses not contributing to the gate itself. For regular gates (i.e., all logical operators in Table 2.1), substitution can be performed by resolving non-gate with gate clauses as follows: $R_x(\mathcal{S}) = \{\{G_x \otimes H_{\neg x}\}, \{G_{\neg x} \otimes H_x\}\}$, omitting the tautological and the redundant parts $\{G_x \otimes G_{\neg x}\}$ and $\{H_x \otimes H_{\neg x}\}$, respectively [JHB12].

Example 2.4. Consider the following formula:

$$\{\underbrace{\{x, \neg a, \neg b\}}_{G_x}, \underbrace{\{\neg x, a\}, \{\neg x, b\}}_{G_{\neg x}}, \underbrace{\{x, c\}}_{H_x}, \underbrace{\{y, f\}}_{H_y}, \underbrace{\{\neg y, d, e\}}_{G_{\neg y}}, \underbrace{\{y, \neg d\}, \{y, \neg e\}}_{G_y}\}$$

Given the CNF representations in Table 2.1, the first three clauses in the formula above capture the AND gate ($x \leftrightarrow a \wedge b$) and the last three clauses capture the OR gate ($y \leftrightarrow d \vee e$). Thus, resolving the fourth clause with the second and the third clauses yield the resolvents $\{G_{\neg x} \otimes H_x\} = \{\{a, c\}, \{b, c\}\}$. Similarly, eliminating y results in $\{G_{\neg y} \otimes H_y\} = \{\{f, d, e\}\}$.

Proposition 2.1. BVE is non-confluent when applied on an arbitrary set of variables in a CNF formula.

Proof. Consider the following formula $\mathcal{S} = \{C_1 = \{x, a, b\},\ C_2 = \{a, d\},\ C_3 = \{\neg x, \neg a, c\},\ C_4 = \{\neg a, x\}\}$. Suppose that the goal is to obtain a simplified formula by eliminating the variables x and a, in that order. Resolving x yields the following resolvents

$$\begin{aligned}R_x(\mathcal{S}) &= \{C_1 \otimes C_3\} \cup \{C_4 \otimes C_3\} \cup \{C_2\}\\ &= \{\{x, a, b\} \otimes \{\neg x, \neg a, c\}\} \cup \{\{\neg a, x\} \otimes \{\neg x, \neg a, c\}\} \cup \{C_2\}\\ &= \{\{\neg a, c\}\} \cup \{C_2\}\\ &= \{C_1' = \{\neg a, c\}, C_2\}\end{aligned}$$

Then by resolving a, we get the simplified formula

$$\begin{aligned}R_a(R_x(\mathcal{S})) &= \{C_1' \otimes C_2\}\\ &= \{\{\neg a, c\} \otimes \{a, d\}\} = \{\{c, d\}\}\end{aligned}$$

On the other hand, eliminating a first gives the resolvents

$$\begin{aligned}R_a(\mathcal{S}) &= \{C_1 \otimes C_3\} \cup \{C_2 \otimes C_3\} \cup \{C_1 \otimes C_4\} \cup \{C_2 \otimes C_4\}\\ &= \{\{x, a, b\} \otimes \{\neg x, \neg a, c\}\} \cup \{\{a, d\} \otimes \{\neg x, \neg a, c\}\} \cup\\ &\quad \{\{x, a, b\} \otimes \{\neg a, x\}\} \cup \{\{a, d\} \otimes \{\neg a, x\}\}\\ &= \{\{\neg x, d, c\}\} \cup \{\{x, b\}\} \cup \{\{x, d\}\}\\ &= \{C_1' = \{\neg x, d, c\},\ C_2' = \{x, b\},\ C_3' = \{x, d\}\}\end{aligned}$$

Then by eliminating x, we get the simplified formula

$$\begin{aligned}R_x(R_a(\mathcal{S})) &= \{C_1' \otimes C_2'\} \cup \{C_1' \otimes C_3'\}\\ &= \{\{\neg x, d, c\} \otimes \{x, b\}\} \cup \{\{\neg x, d, c\} \otimes \{x, d\}\}\\ &= \{\{b, c, d\}\} \cup \{\{c, d\}\} = \{\{b, c, d\}, \{c, d\}\}\end{aligned}$$

Clearly, $R_x(R_a(\mathcal{S}))$ is different from $R_a(R_x(\mathcal{S}))$, which subsumes the former. Thus, the order in which the variables are eliminated influences the resulting formula, i.e., BVE does not have a unique fixpoint for all formulas. In practice, a fixpoint is rather determined by the variable ordering heuristics. □

Later, in Chapter 4, we prove that applying BVE on a set of *mutually-independent* variables is actually *confluent*. In other words, the elimination of these particular variables in any arbitrary order always produces the same simplified formula.

2.3.2 Subsumption Elimination

Suppose that we have two clauses C_1, C_2 and $C_2 \subset C_1$. In *subsumption elimination*, C_1 is said to be subsumed by C_2 or C_2 subsumes C_1. The subsumed clause C_1 is redundant and can removed [EB05]. If C_2 is a `LEARNT` clause, it must be considered as `ORIGINAL` in the future, to prevent deleting it during learnt clause reduction [BFFH20] (see Section 2.2.1).

Definition 2.6 (self-subsuming resolution)**.** The *self-subsuming resolution* is a special case of subsumption. The former can be applied on clauses C_1, C_2 iff for some variable x, we have one of the following three cases:

- $C_1 = C_1' \cup \{x\}$, $C_2 = C_2' \cup \{\neg x\}$, and $C_2' \subset C_1'$. In this case, x can be removed from C_1 and we say that C_1 is *strengthened* by C_2.
- $C_1 = C_1' \cup \{\neg x\}$, $C_2 = C_2' \cup \{x\}$, and $C_2' \subset C_1'$. In this case, $\neg x$ can be removed from C_1.
- $C_1 = C_1' \cup \{x\}$, $C_2 = C_2' \cup \{\neg x\}$, and $C_1' = C_2'$. In this case, x is removed from C_1.

In this work, with SUB, we refer to the application of self-subsuming resolution followed by subsumption elimination until a heuristic fixpoint is reached.

Proposition 2.2. Self-subsuming resolution is a combination of resolution and subsumption elimination.

Proof. Consider the first case in Definition 2.6. The outcome of self-subsuming resolution can be deduced by applying the resolution step $C_1 \otimes_x C_2$ which yields the resolvent $C_1' \cup C_2'$. Since, $C_2' \subset C_1'$ then the resolvent can be simplified to just C_1' which, in turn, subsumes the original clause C_1. Thus, the latter is removed while the former and C_2 are kept, stimulating the removal of x from C_1. The same reasoning applies on the second and third cases. $\square$

Example 2.5. Consider the formula $\{\{a, b, c\}, \{\neg a, b\}, \{b, c, d\}\}$. The first clause is self-subsumed by the second clause w.r.t. variable a and can be strengthened to $\{b, c\}$ which in turn subsumes the last clause $\{b, c, d\}$. The latter clause is then removed and the simplified formula becomes $\{\{b, c\}, \{\neg a, b\}\}$.

Proposition 2.3. SUB (the mixture of self-subsuming resolution and subsumption elimination) is confluent when applied on an arbitrary set of clauses in a CNF formula.

Proof. It suffices to prove that both self-subsuming resolution and subsumption elimination are confluent when applied separately. It follows from this that applying one after the other (SUB) is also confluent.

1. Suppose that we have $C_1, C_2 \in \mathcal{S}$, with $C_1 \neq C_2$. Two possible scenarios are feasible. First, both C_1 and C_2 may be strengthened by other clauses $C', C'' \in \mathcal{S}$, where possibly $C' = C''$. In that case, since by Definition 2.6, C' and C'' are not altered when C_1 and C_2 are strengthened with self-subsuming resolution w.r.t. C' and C'', the order in which C_1 and C_2 are strengthened does not influence the outcome, even if $C' = C''$. Second, C_1 can strengthen C_2, or vice versa. By Definition 2.6, either one of these cases hold, but not both, i.e., C_1 and C_2 cannot strengthen each other. Thus, either C_1 or C_2 can be strengthened, regardless of the order in which they are checked for self-subsuming resolution, leading to the same simplified formula.

2. Similarly, in subsumption elimination, two clauses C_1 and C_2, $C_1 \neq C_2$, may be subsumed by other clauses $C', C'' \in \mathcal{S}$, with possibly $C' = C''$, or either C_1 is subsumed by C_2 or C_2 by C_1, but not both. Thus, either both can be subsumed by other clauses, and removing them in any order produces the same outcome, or either C_1 or C_2 is removed. Regardless of the order in which they are checked for subsumption, the same simplified formula is always obtained.

By proofs (1) and (2), SUB is confluent, i.e., the order of clauses in which SUB is applied does not influence the simplified formula. □

2.3.3 Blocked Clause Elimination

The BCE simplification removes so-called *blocked clauses* from a given formula [JBH10].

Definition 2.7 (Blocking literal)**.** For a given formula $\mathcal{S}$, a literal ℓ is blocking a clause $C \in \mathcal{S}$ iff $\forall C' \in \mathcal{S}_{\neg\ell}$, the resolvent $C \cup C' \setminus \{\ell, \neg\ell\}$ obtained by the operation $C \otimes_\ell C'$ is a tautology.

Definition 2.8 (Blocked clause)**.** A clause is blocked iff it has a blocking literal.

Example 2.6. Consider the formula $\{\{a,b,c,d\},\{\neg a,\neg b\},\{\neg a,\neg c\}\}$. Both the literals a and c are blocking the first clause, since resolving a produces the tautologies $\{\{b,c,d,\neg b\},\{b,c,\neg c,d\}\}$. Likewise, resolving c yields the tautology $\{a,b,\neg a,d\}$. Hence the blocked clause $\{a,b,c,d\}$ can be removed from $\mathcal{S}$.

Proposition 2.4. BCE is non-confluent when applied on an arbitrary set of clauses in a CNF formula.

Proof. Consider the formula $\{C_1 = \{\neg a, \neg b, c\}, C_2 = \{a, b\}\}$. Picking C_1 first gives $\{a, b\}$, since $C_1 \otimes_a C_2 = \{\neg b, c, b\}$. However, picking C_2 first gives $\{\neg a, \neg b, c\}$, since $C_2 \otimes_a C_1 = \{b, \neg b, c\}$. Hence, the order of clauses being checked for BCE, influences the resulting formula. Interestingly, note that our proof contradicts the proposition by the authors Järvisalo *et al.* [JBH10] in which they state that BCE is confluent. $\square$

Chapter **3**

Graphics Processing Units

"Moore's law is dead and GPUs will soon replace CPUs."

– Jensen Huang

This chapter gives a brief introduction to the GPU architecture and CUDA programming model which is the core technology utilized in the parallel implementations in this thesis. First, we discuss Moore's law and how the GPUs evolved so quickly over the years, dominating both the game industry and general-purpose computing. Second, we take a glance over the GPU architecture, outlining its capabilities and the different types of memory supported. Third, the CUDA model and its algorithmic notations are explained. Finally, we discuss concurrent kernel execution and data transfer overlapping via asynchronous calls.

3.1 50 Years of Microprocessors

Since 1970, microprocessors based on a single-core CPU, such as those in the Intel and the AMD Opteron families, enhanced the computation performance of all developed software while reducing the costs and power consumption. Gordon Moore predicted that the number of transistors in an integrated circuit (IC) doubles about every two years which is subsequently called *Moore's law* [Moo65]. For 40 years, the law was guiding the semiconductor industry to advance the

CPU manufacturing by almost doubling the number of transistors per die every two years, which drastically impacted the computer application functionality and performance.

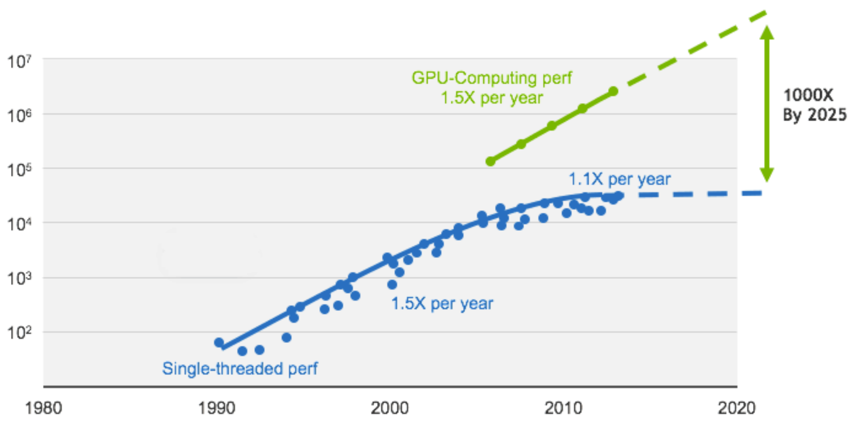


Figure 3.1: 40 years of microprocessors trend data[3]

Nevertheless, the computational power of CPUs measured in *giga floating-point operations per second* (GFLOPS) has stopped doubling since 2003 due to the increasing heat dissipation, which limits the clock frequency and the *instructions per cycle* (IPC). From this point, microprocessor vendors started to add more cores to the CPU, trying to keep the energy consumption to a minimum while boosting the computational power.

After four decades of achieving steady gains in performance of about 200% per two years, Moore's law has finally run its course (see Figure 3.1). The reason is that the industry has reached the upper bound of how many cores can be cost-effectively installed on a single CPU chip which means that the CPU performance now only grows by 10 percent per year. In contrast, since NVIDIA introduced the first many-core architecture for general-purpose programming around 2007, GPUs have been growing rapidly in terms of number of cores and peak performance. NVIDIA is expecting to reach a 1,000× speedup by 2025.

[3]https://www.nvidia.com/en-gb/about-nvidia/ai-computing/

3.2 GPU Architecture

The GPU provides a remarkable data throughput and memory bandwidth compared to the CPU within a reasonable price and low power consumption. Thanks to CUDA, a wide range of scientific applications can now leverage its hardware capabilities to run faster on the GPU than on the CPU. Other accelerators, such as *field programmable gate arrays* (FPGAs), can be energy efficient, but offer less programming flexibility than GPUs.

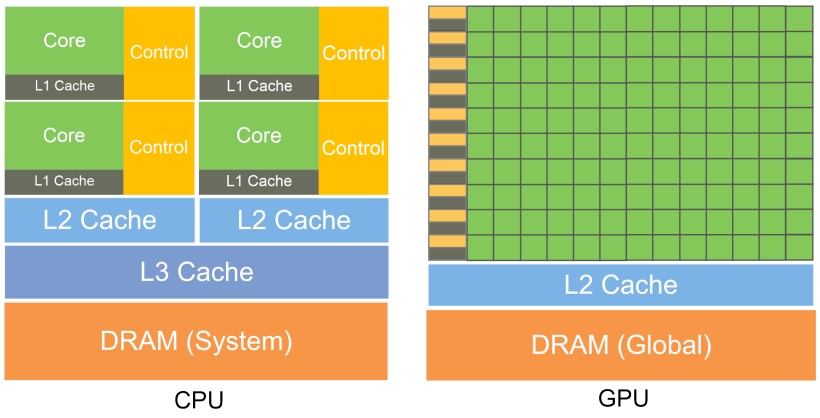


Figure 3.2: CPU vs GPU

The design scheme in Figure 3.2 shows an example distribution of chip resources for a CPU versus a GPU. The main difference between the CPU and the GPU is that the GPU is specialized for massively parallel computations in contrast to data caching and flow control, which are what the CPU is optimised for. Therefore, the CPU is fast in executing a sequence of operations (called *thread*) and can execute up to tens of these threads in parallel; the GPU executes thousands of threads for simple operations to achieve much higher throughput.

A GPU typically consists of multiple *streaming multiprocessors* (SMs) and each SM resembles an array of streaming processors or cores (green squares in Figure 3.2) where every core can execute multiple threads grouped together in 32-thread scheduling units called *warps*. To handle all these threads, an SM employs the Single-Instruction Multiple-Thread (SIMT) paradigm where the instructions are pipelined, utilizing both instruction-level parallelism within a single thread, and thread-level parallelism through simultaneous warp scheduling.

Unlike CPU threads, GPU threads are issued in order and there is no branch prediction or any complex control flow.

Table 3.1 makes a comparison between various GPU micro-architectures designed by NVIDIA during the last 5 years. All entries represent the peak values of all GPU models that belong to a certain architecture. For example, the Quadro P6000 GPU has a clock speed of 1,506 MHz while the Titan Xp has 1,406 MHz. The reader can observe a gigantic leap in the number of cores between Pascal and Ampere architectures by a growth of 300%. Add to that, the technology of ray-tracing and tensor cores which contributed to major improvements in both graphics and AI applications. Compared to the CPU generations, in the same period of time, this is remarkable. Since the introduction of Zen 32-core CPU by AMD in 2017, the number of cores has only increased to 64 in Zen 2 generation. Keep in mind that a GPU of 10,496 cores (RTX 30 series) only consumes up to 350 Watt against 280 Watt of 64-core CPU (Ryzen Threadripper series).

3.3 CUDA Programming Model

The CUDA programming model defines how the programmer sees the GPU as a general-purpose computing facility [FVS11]. In this section, we give the GPU hierarchy starting from a top-level C/C++ program to all the way down to the smallest execution unit so-called *thread*, including different types of memory that can be used to store the program and any relative data.

3.3.1 GPU Kernel

In a GPU program, a CPU thread launches a *kernel* (GPU global function) to be executed by thousands of threads packed in thread *blocks* of up to 1,024 threads or 32 warps. All threads together form a *grid*. Threads and blocks can be indexed by a one-dimensional, two-dimensional, or three-dimensional unique identifier accessible within the kernel. The GPU manages the execution of a launched kernel by evenly distributing the launched blocks to the available SMs through a hardware warp scheduler. A warp executes a single shared instruction at a time, therefore optimal efficiency is achieved when all 32 threads of a warp agree on their execution path. If threads in a warp diverge via a conditional branch, the warp executes each path taken, disabling threads that are not on the same path. Control divergence occurs only within a warp, whilst different warps are executed simultaneously regardless of the path they are executing.

Table 3.1: Microarchitectures released by NVIDIA during the last 5 years

Architecture	**Launch Date**	**Lithography**	**CC**[4]	**CUDA Cores**	**Tensor**[5] **Cores**	**RT**[6] **Cores**	**Base Clock Speed (MHz)**	**Memory**		**TDP**[7] **(Watt)**	**ITS**[8] **Support**	**GPU Model Examples**
								Size (GB)	**Bandwidth (GB/s)**			
Pascal	2016	14/16 *nm*	6.1	3,840	—	—	1,506	24	432	250	no	Titan Xp, Quadro P6000
Volta	2017	12 *nm*	7	5,120	640	—	1,200	32	870	250	yes	Titan V, Quadro GV100
Turing	2018	12 *nm*	7.5	4,608	576	72	1,455	48	672	295	yes	Titan RTX, Quadro RTX 8000
Ampere A100	2020	7 *nm*	8	6,912	432	—	765	40/80	1,555/1,935	250/300	yes	A100 40/80 GB
Ampere RTX 30	2021	8 *nm*	8.5	10,496	328	82	1,395	24	936	350	yes	RTX 3080, RTX 3090

[4] Compute Capability: gives the device compute specifications and CUDA features.
[5] They are designed specifically to accelerate mixed-precision computations (e.g. 8, 16, and 32 bits).
[6] Ray Tracing: a new technology for improving and accelerating the light rendering.
[7] Thermal Design Power: maximum amount of heat generated by the computer chip.
[8] Independent Thread Scheduling: groups idle threads into a single warp to execute the same next instruction.

Before the Volta architecture, warps used a single *program counter* (PC) for all the 32 threads with an active mask specifying the active threads within a warp. Hence, threads from the same warp in divergent paths cannot signal each other or exchange data, and algorithms requiring data guards by locks or mutexes often led to deadlocks on a GPU. Starting with the Volta architecture, the Independent Thread Scheduling (ITS) allows independent execution of threads within a single warp. ITS maintains execution state per thread, including a PC and call stack (these were implemented per warp in older architectures), either to leverage all available resources (e.g. ALUs) or to allow one thread instead of a whole warp to wait for data that is produced by another thread. This allows the threads to diverge and reconverge at sub-warp granularity [NVI20a].

Through the next chapters, we use three conventions in our developed kernels. First of all, we express a thread dimension with a bold italic font ***dimension***. For example, threads or blocks can be launched in the $\boldsymbol{x}$ or $\boldsymbol{y}$ or $\boldsymbol{z}$ dimension. Second of all, with $t_{\boldsymbol{x}}$, we refer to the *block-local* identifier (ID) of the working thread. By using this ID, we can achieve that different threads in the same block work on different data. Third of all, we use so-called *grid-stride loops* to process data elements in parallel. The statement **for all** $tid \in [\![0, N]\!]$ **do in parallel** expresses that all natural numbers in the range $[0, N)$ must be considered in the loop, and that this is done in parallel by having each executing thread start with element $t_{\boldsymbol{x}}$, i.e., $tid = t_{\boldsymbol{x}}$, and before starting each additional iteration through the loop, the thread adds to tid the total number of threads on the GPU. If the updated tid is smaller than N, the next iteration is performed with this updated tid. Otherwise, the thread exits the loop. A grid-stride loop ensures that when the range of numbers to consider is larger than the number of threads, all numbers are still processed.

Moreover, we use intra-warp device functions when possible in our kernels to facilitate communications between threads inside a warp. Such functions offer *data exchange* primitives among a group of threads per warp that do not require synchronization between kernel blocks. A common example is `activemask` which indicates the current active threads executing a branch. With their IDs, we can do warp voting to elect a leader. A leader is particularly useful to propagate the result of an atomic operation to other thread registers without the need to use atomics across all warp threads. An ideal situation, with a full warp taking the same branch, the number of atomics and global memory accesses is reduced by a factor of 31, i.e., only the leader does the atomic operation. Another example is `shfl_sync` which is employed by most developers to do parallel reduction on the last warp per block. We deployed device functions in several algorithms in Chapter 5 to rationalise atomics and shared memory synchronizations.

3.3.2 Memory Hierarchy

Concerning the memory hierarchy, a GPU has multiple types of memory:

- *Global memory* with high bandwidth but also high latency is accessible by both GPU threads and CPU threads and thus acts as interface between CPU and GPU.

- *Constant memory* is read-only for all GPU threads. It has a lower latency than global memory, and can be used to store any pre-defined constants.

- *Shared memory* is on-chip memory shared by the threads in a block. Each SM has its own shared memory. It is much smaller in size than global and constant memory (in the order of tens of kilobytes), but has a much lower latency. It can be used to efficiently communicate data between threads in a block.

- *Registers* are used for on-chip storage of thread-local data. They are very small, but provide the fastest memory and the possibility for threads in a warp to exchange register data.

- *Local memory* is part of the global memory but provides extra storage for thread-local data with faster access using the ability of interleaved addressing.

Regarding atomicity, a GPU is capable of executing *atomic* operations on both global and shared memory. A GPU *atomic* function typically performs a *read-modify-write* memory operation on one 32-bit or 64-bit word.

To hide the latency of global memory, ensuring that the threads perform *coalesced accesses* is one of the best practices. When the threads in a warp try to access a consecutive block of 32-bit words, their accesses are combined into a single (coalesced) memory access. Uncoalesced memory accesses can, for instance, be caused by data sparsity or misalignment. Furthermore, we use *unified memory* [NVI20a] to store the main data structures that need to be regularly accessed by both the CPU (host) and the GPU (device). Unified memory creates a pool of managed memory that is shared between the host and the device. This pool is accessible to both sides using the same addresses.

To maximise the bandwidth of memory transfers of device and host arrays allocated via dedicated memory (non-unified), we use *page-locked* (or *pinned*) memory. Memory allocations on the host are pageable by default and the GPU cannot access data directly from pageable host memory. Therefore, when a data

Algorithm 3.1: Naive matrix transpose with shared memory

```
   Input  : global A
   Input  : shared tile
   Output : global B
1  A, B ← ALLOCATE($width$, $height$)          // allocate global memory for matrices A, B
2  B ← MATRIXTRANSPOSE(A)                                     // invoke the transpose kernel
3  kernel MATRIXTRANSPOSE(A):
4      shared tile[$blockDim$][$blockDim$]                 // shared memory for a matrix tile
5      for all $tid_y \in [\![ 0, height ]\!]^y$ do in parallel      // grid-stride loop for y threads
6          for all $tid_x \in [\![ 0, width ]\!]^x$ do in parallel   // grid-stride loop for x threads
7              register $row \leftarrow tid_y$, $col \leftarrow tid_x$
8              if $row < height \wedge col < width$ then
9                  tile[$t_y$][$t_x$] = A[$row \times width + col$]  // load block data to shared memory
10             end
11             SYNCTHREADS()              // synchronize a block after each load to tile
12             $col \leftarrow t_x + blockid_y \times blockDim$              // $blockid_y$: block ID in y
13             $row \leftarrow t_y + blockid_x \times blockDim$              // $blockid_x$: block ID in x
14             if $col < height \wedge row < width$ then
15                 B[$row \times height + col$] = tile[$t_x$][$t_y$]      // write back to global memory
16             end
17         end
18     end
19 end
```

transfer from device to host pageable memory (and vice versa) is invoked, the CUDA driver must first allocate a temporary pinned buffer, and copy the data to the buffer first before it reaches its destination. We can avoid this extra transfer by directly allocating a host array in the pinned memory. However, pinned-memory allocations should be avoided for large data structures (a SAT formula, for instance) as they may reduce the physical memory available for the operating system.

Algorithm 3.1 shows a simple kernel for matrix transpose to point out the algorithmic notations commonly used in Chapters 4, 5, and 7. First of all, all device codes and kernel invocations are highlighted with light gray background to distinguish them from the host executable code. For example, line 1 allocates global memory (depicted by the **global** keyword) on the host for matrices `A` and `B`. At line 2, a kernel is invoked by the host and executed on the device. The kernel is given at lines 3-19. At line 4, a **shared** memory 2D array is allocated on-demand to transpose tiles of the input matrix at lines 9 and 15. The grid-stride loops at lines 5-6 are responsible for sweeping all matrix elements in parallel and iteratively over multiple grids if necessary. Local thread variables are stored in **register** memory at line 7. The conditions at lines 8 and 14 ensure that the matrix indices cannot go out of boundary in case of irregular size (i.e., when it is not a power of two). The syncThreads procedure at line 11 is needed to synchronize all threads per block after writing to shared memory. At the end of the kernel, all blocks are synchronized automatically and there is no need to explicitly call syncThreads again.

In SAT applications, a CNF formula is usually (if not always) irregular; implying that the number of (variables, clauses, literals) in the formula and the clause size are expected to be a non-power-of-two and to change constantly during the solving procedure. Therefore, in our algorithmic design of GPU SAT solving, we always assume irregularity to guarantee a correct behaviour of the proposed techniques.

3.4 Streams and Concurrent Execution

A stream is a sequence of instructions issued in order by one or more host threads. These instructions shape the GPU kernels described above. On the other hand, different streams may execute their kernels out of order or concurrently. This behavior is not guaranteed as it relies on the available resources on the GPU (e.g. SMs, memory, etc.) or the data dependency between kernels. This dependency may arise from different kernels on the same stream or from other streams. In such case the programmer must be aware when and where to synchronize the launched kernels on the device w.r.t. the host. With multiple streams, it is possible to overlap the memory copies with the kernel launches assuming they run on different streams.

Similarly, a device task (i.e. a kernel launch or data transfer) can be executed *asynchronously* w.r.t. the host thread. This is facilitated by asynchronous CUDA functions that return control to the host immediately after their call and before the device completes its task. Asynchronous calls have the benefit of queuing different device operations while the host is busy doing other tasks. The data transfers between the host and the device can be overlapped as well iff the host memory is page-locked as discuss earlier regarding memory optimisations.

3.5 Specifications of our GPUs

Finally, with Table 3.2, we wrap up this chapter by summarising the key features between the GPUs that are used in this thesis (Chapters 4, 5, and 7). A comparison between different types of architectures is outlined at Table 3.1. More information on CUDA compute capabilities can be found in [NVI20a].

Table 3.2: Comparison between the GPUs used in this thesis

Feature	**Titan Xp**	**RTX 2080 Ti**	**Titan RTX**
Usage	Chapter 4	Chapter 7	Chapter 5
Architecture	Pascal	Turing	Turing
SMs	30	68	72
CUDA cores per SM	128	64	64
CUDA cores	3,840	4,352	4,608
Core base speed (MHz)	1,405	1,350	1,350
Core boost speed (MHz)	1,582	1,545	1,770
Maximum threads per SM	2,048	1,024	1,024
Maximum threads per block	1,024	1,024	1,024
Maximum blocks per SM	32	16	16
Warp size	32	32	32
Maximum 32-bit registers per thread	255	255	255
Local memory per thread (KB)	512	512	512
Global memory type	GDDR5X	GDDR6	GDDR6
Global memory size (GB)	12	11	24
Global memory speed (MHz)	1,426	1,750	1,750
Global memory bus width (bits)	384	352	384
Global memory bandwidth (GB/s)	547.7	616	672
Shared memory per block (KB)	48	64	64
Constant memory size (KB)	64	64	64
NVLink support	no	yes	yes
CUDA CC	6.1	7.5	7.5

Chapter 4

SAT Preprocessing

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it"

– Brian Kernighan

Simplifying SAT problems prior to solving them has proven its effectiveness in modern conflict-driven clause learning (CDCL) SAT solvers [SS99, Bie13], particularly when applied on real-world applications relevant to software and hardware verification [HS07, BCMD90, JS05, EB05]. It tends to produce reasonable reductions in acceptable processing time. Many techniques based on e.g. variable elimination, clause elimination, and equivalence reasoning are being used to simplify SAT problems, prior to the solving phase (preprocessing) [SP04, EB05, GM13, HS07, HJB10, JBH10]. However, applying variable and clause eliminations iteratively to large problems (in terms of the number of literals) may actually be a performance bottleneck in the whole SAT solving procedure, or increase the number of literals, therefore negatively impacting the solving time.

Recently, the authors of [HW12, BS18] discussed the current main challenges in parallel SAT solving. One of these challenges concerns the parallelisation of SAT simplification in modern SAT solvers. Massively parallel computing systems such as GPUs offer great potential to speed up computations, but to achieve this, it is crucial to engineer new parallel algorithms and data structures from scratch to make optimal use of those architectures. GPU platforms have become attractive for general-purpose computing with the availability of the CUDA programming model. More information about CUDA is provided in Section 3.3.

In this chapter, we introduce the first parallel algorithms for various techniques widely used in SAT simplification and discuss the various performance aspects of the proposed implementations and data structures. Also, we discuss the main challenges in CPU-GPU memory management and how to address them. In a nutshell, we aim to effectively simplify SAT formulas, even if they are extremely large, in only a few seconds using the massive computing capabilities of GPUs.

Contributions

We propose novel parallel algorithms to simplify SAT formulas using GPUs, and experimentally evaluate them, i.e., we measure both their runtime efficiency and their effect on the overall solving time, for a large benchmark set of SAT instances encoding real-world problems. We show how multiple variables can be eliminated simultaneously on a GPU while preserving the original satisfiability of a given formula. We call this technique Bounded Variable-Independent Parallel Elimination (BVIPE). The eliminated variables are elected first based on some criteria using the proposed algorithm Least-Constrained Variable Elections (LCVE). The Bounded Variable Elimination (BVE) procedure includes both the so-called *resolution rule* and *gate equivalence reasoning.*

In addition to BVIPE, we introduce the following contributions:

- ⋆ We propose new parallel algorithms for subsumption and blocked clause eliminations on the GPU. Further, we introduce a new simplification method called *hidden redundancy elimination* (HRE) with an optimized GPU algorithm. To that extent, proofs of correctness are provided for all parallel algorithms and the new simplification method.
- ⋆ We present the first multi-GPU support with load balancing for various SAT simplifications along with a comprehensive evaluation of different combinations of these simplifications to assess their impact on SAT solving.

The chapter is organised as follows: the main GPU challenges for SAT simplification are discussed in Section 4.1, and the proposed algorithms are explained in Section 4.2. Section 4.3 presents our experimental evaluation. Section 4.4 discusses related work, and Section 4.5 provides a conclusion and suggests future work.

4.1 GPU Challenges: Memory and Data

In this section, we discuss the main challenges regarding the dynamic allocation of GPU memory in SAT preprocessing and how nested data structures can be designed for that purpose. The data structures are designed from scratch to store the relative CNF information and to facilitate fast access to literals and clauses during simplifications. Further, we present various optimisations to potentially reduce the latency of accessing global memory (see Section 3.3.2) and to maximize the data transfer rates with the CPU or other GPUs.

4.1.1 Memory Management

When small data packets need to be accessed frequently, both on the host (CPU) and device (GPU) side (which is the case in the current work), *unified memory* can play a crucial role in boosting the transfer rates by avoiding excessive memory copies (see Section 3.3.2 for more details). An important advantage of unified memory is that it allows the CPU to allocate multidimensional pointers referencing global memory locations or nested structures. However, if a memory pool is required to be reallocated (resized), one must maintain memory coherency between the CPU-side and GPU-side memories. A reallocation procedure is necessary for our variable elimination algorithm, to make memory available when producing resolvents and reduce the memory use when removing clauses.

To better explain the coherency problem in reallocation, suppose there is an array A allocated and loaded with some data X, then X is visible from both the CPU and GPU memories. When A is reallocated from the host side, the memory is not physically allocated until it is first accessed, particularly when using an NVIDIA GPU with the Pascal architecture (see Table 3.1). Once new data Y is written to A from the device side, both sides will observe a combination of X and Y, leading to memory corruptions and page faults. To avoid this problem, A must be reset on the host side directly after memory reallocation to assert the physical allocation. After that, each kernel may store its own data safely in the global memory. In the proposed algorithms, we introduce two types of optimisations addressing memory space and latency.

Regarding memory space optimisation, allocating memory dynamically each time a clause is added is not practical on a GPU while variables are eliminated in parallel. To resolve this, we initially launch a GPU kernel to calculate an upper bound for the number of resolvents to be added before the elimination procedure starts (Section 4.2). After this, reallocation is applied to store the new resolvents. Furthermore, a global counter is implemented inside our CNF data

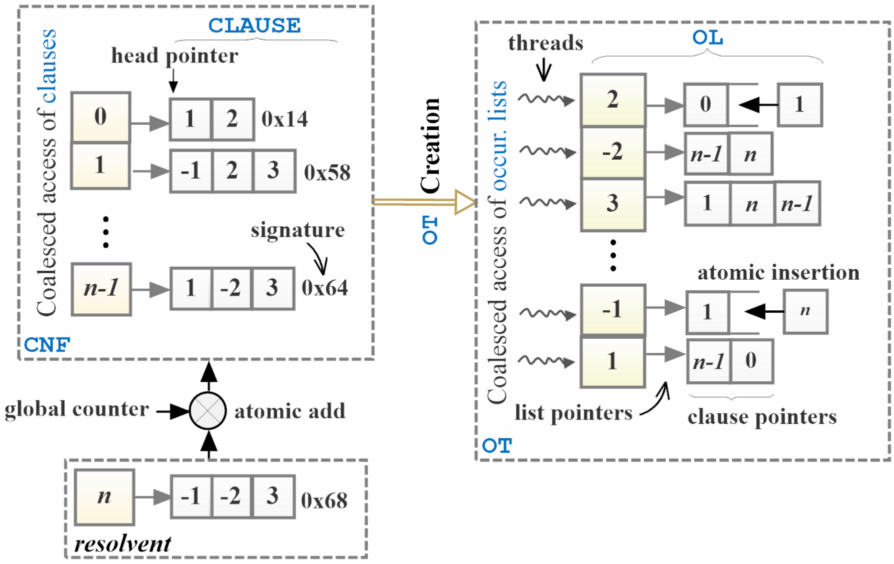


Figure 4.1: An example of CNF and OT data structures.

structure to keep track of new clauses. This counter is incremented atomically by each thread when adding a clause.

Concerning memory latency optimisation, we use shared memory to temporarily store the resolvents upon eliminating a variable and fast check for tautologies. This has the advantage of reducing the number of global memory accesses. Nevertheless, the size of shared memory in a GPU is very limited (48 KB per block in Titan Xp GPU as listed in Table 3.2). If the potential size of a resolvent is larger than the amount pre-allocated for a single clause, our BVIPE algorithm automatically switches to the global memory and the resolvent is directly added to the new CNF formula. This mechanism reduces the global memory latency when applicable and deals with the shared memory size limitation dynamically.

4.1.2 Data Structures

The efficiency of state-of-the-art sequential SAT solving and preprocessing is to a large extent due to the meticulously coded data structures. When considering

SAT simplification on GPUs, new data structures have to be tailored from scratch. In this work, we need two of them, one for the SAT formula in CNF form (which we refer to as `CNF`) and another for the literal *occurrence table* (`OT`), via which one can efficiently iterate over all clauses containing a particular literal. In CPU implementations, typically, they are created using *heaps* and *auto-resizable vectors*, respectively. However, heaps and vectors are not suitable for GPU parallelisation, since data is inserted, reallocated and sorted dynamically. The best GPU alternative is to create a nested data structure with arrays using unified memory (see Figure 4.1). The CNF contains a raw pointer (linear array) to store CNF literals and a child structure `CLAUSE` to store clause info.

Each clause has a *head pointer* referring to its first literal. The `OT` structure has a raw pointer to store the clause occurrences (array pointers) for each literal in the formula and a child structure `OL` (*occurrence list*). The creation of an `OL` instance is done in parallel per literal using atomic operations. For each clause C, a thread is launched to insert the occurrences of C's literals in the associated `OL`'s. One important remark is that two threads storing the occurrences of different literals do not have to wait for each other. For instance, `OT` in Figure 4.1 shows two different atomic insertions executed at the same time for literals 2 and -1 (represented in DIMACS format (Section 2.1)). This minimises the performance penalty of using atomics.

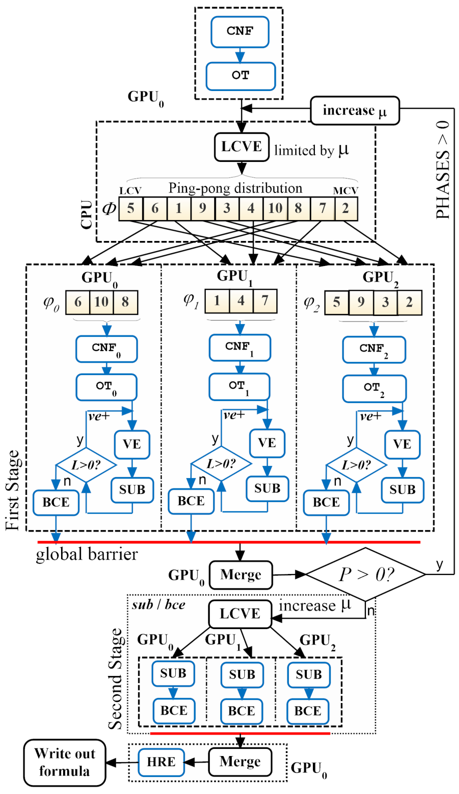
The main advantage of the proposed data structures is that as mentioned above, `OT` instances can be constructed in parallel. Furthermore, coalesced access is guaranteed since pointers are stored consecutively (the gray arrows in Figure 4.1), and no explicit memory copying is done (host and device pointers are identical) making it easier to integrate the data structures with any sequential or parallel code.

4.2 Algorithm Design and Implementation

4.2.1 Parallelisation Approach

All proposed algorithms in this chapter are implemented in SIGmA[9] using CUDA/C++. The complete workflow including the new simplifications and multi-GPU support is depicted by Figure 4.2. SIGmA accepts as input a SAT formula in CNF stored in the DIMACS format. Similarly, the output of SIGmA is a simplified formula that is written to an output file in the DIMACS format. Each block in Figure 4.2 is explained in detail in the upcoming subsections.

[9]The tool can be downloaded here: https://gears.win.tue.nl/software.



Tool Options	Functionality
VE	enables variable elimination only in the first stage
SUB	enable subsumption elimination
VE+	iterate over VE & SUB until no literals (*L*) can be removed in the first stage
BCE	enables blocked clause elimination
PHASES=<*n*>	allows (*n*) number of iterations (*P*) of the first stage
HRE	enables hidden redundancy elimination on a single GPU in the second stage
GPUS=<*n*>	activate (*n*) number of GPUs

Figure 4.2: Complete workflow of SIGmA with multi-GPU support.

Regarding the proofs of correctness of the upcoming parallel algorithms implemented in SIGmA, we first try to refute the presence of data racing, then we affirm the resulting formula is identical to the one obtained by applying the sequential implementation, i.e., single-threaded execution of the parallel algorithm. All simplifications introduced in Chapter 2 may cause data racing due to the strong dependency between clauses, which is undesirable for parallel computations. For this reason, we restrict the application of the simplification methods to variables that are *mutually independent*.

The LCVE algorithm we propose is responsible for *electing* a subset of independent variables from a set of authorised candidates. The remaining variables relying on the elected ones are frozen.

Definition 4.1 (Authorised candidates)**.** Given a CNF formula $\mathcal{S}$, we call $\mathcal{A}$ the set of *authorised candidates*: $\mathcal{A} = \{x \mid 1 \leq h[x] \leq \mu \vee 1 \leq h[\neg x] \leq \mu\}$, where

- h is a histogram array ($h[x]$ is the number of occurrences of x in $\mathcal{S}$).
- μ denotes a given maximum number of occurrences allowed for both x and its complement, representing the cut-off point for the LCVE algorithm.

Definition 4.2 (Candidate Dependency Relation)**.** We call a relation $\mathsf{D}\colon \mathcal{A} \times \mathcal{A}$ a *candidate dependency relation* iff $\forall x, y \in \mathcal{A}$, $x \mathrel{\mathsf{D}} y$ implies that $\exists C \in \mathcal{S}.(x \in C \vee \neg x \in C) \wedge (y \in C \vee \neg y \in C)$

Definition 4.3 (Elected candidates)**.** Given a set of authorised candidates $\mathcal{A}$, we call a set $\Phi \subseteq \mathcal{A}$ a set of *elected candidates* iff $\forall x, y \in \Phi.\ \neg(x \mathrel{\mathsf{D}} y)$

Definition 4.4 (Frozen candidates)**.** Given the sets $\mathcal{A}$ and Φ, the set of *frozen candidates* $\mathcal{F} \subseteq \mathcal{A}$ is defined as $\mathcal{F} = \{x \mid x \in \mathcal{A} \wedge \exists y \in \Phi.\ x \mathrel{\mathsf{D}} y\}$

Algorithm 4.1: Constructing $\mathcal{A}$

```
Input  : global $\mathcal{S}$, $\mu$
Output : global $\mathcal{A}$, scores, $h$
1  $h \leftarrow$ HISTOGRAM($\mathcal{S}$)
2  $\mathcal{A}$, scores $\leftarrow$ ASSIGNSCORES($h, \mathcal{A}$, scores)
3  $\mathcal{A} \leftarrow$ PRUNE(SORT($\mathcal{A}$, scores), $h, \mu$)
4  kernel ASSIGNSCORES($h, \mathcal{A}$, scores):
5      for all $tid \in [\![ 0, |var(\mathbb{L})| ]\!]$ do in parallel
6          $x \leftarrow tid + 1$, $\mathcal{A}[tid] \leftarrow x$
7          if $h[x] = 0 \vee h[\neg x] = 0$ then
8              scores[$x$] $\leftarrow$ MAX($h[x], h[\neg x]$)
9          else
10             scores[$x$] $\leftarrow h[x] \times h[\neg x]$
11         end
12     end
13 end
```

Before LCVE is executed, a sorted list of the variables in $\mathcal{S}$ needs to be created, ordered by the number of occurrences in that formula, in ascending order (following the same rule as in [EB05]). From this list, the authorised candidates $\mathcal{A}$ can be straightforwardly derived, using μ as a cut-off point. Construction of this list can be done efficiently on a GPU using Algorithm 4.1. As input, it requires a SAT formula $\mathcal{S}$ and a cut-off point μ. At line 1, a histogram array h,

Algorithm 4.2: LCVE

```
Input  : $\mathcal{S}, \mathcal{A}, h, \mathcal{T}$
Output : $\Phi$
$\mathcal{F} \leftarrow \emptyset$
foreach $x \in \mathcal{A}$ do
    if $x \notin \mathcal{F}$ then
        $\Phi \leftarrow \Phi \cup x$
        foreach $C \in \mathcal{S}[\mathcal{T}[x]] \cup \mathcal{S}[\mathcal{T}[\neg x]]$ do
            foreach $\ell \in C$ do
                $v \leftarrow var(\ell)$
                if $v \neq x$ then $\mathcal{F} \leftarrow \mathcal{F} \cup v$
            end
        end
    end
end
```

providing for each literal the number of occurrences in $\mathcal{S}$, is constructed. This histogram can be constructed on the GPU using the histogram method offered by the THRUST library [NVI20b]. Once ASSIGNSCORES kernel execution has terminated, at line 2, the candidates in $\mathcal{A}$ are sorted on the GPU based on their scores in `scores` while μ is used to prune candidates with too many occurrences. We used the radix-sort algorithm as provided in THRUST.

In ASSIGNSCORES, at line 6, the thread index is used as a variable index (variable indices start at 1). At lines 7-11, a score is computed for the currently considered variable x. This score should be indicative of the number of resolvents produced when eliminating x, which depends on the number of occurrences of both x and $\neg x$, and can be approximated by the formula $h[x] \times h[\neg x]$. To avoid score zero in case exactly one of the two literals does not occur in $\mathcal{S}$, we consider that case separately.

Next, Algorithm 4.2 is executed on the host (see Figure 4.2), given $\mathcal{S}$, $\mathcal{A}$, h and an instance of `OT` named $\mathcal{T}$. This algorithm accesses $2 \cdot |\mathcal{A}|$ number of `OL` instances and parts of $\mathcal{S}$. The use of unified memory significantly improves the rates of the resulting transfers and avoids explicitly copying entire data structures to the host side. The output is Φ, implemented as a list. The algorithm considers all variables x in $\mathcal{A}$ (line 2). If x has not yet been frozen (line 3), it adds x to Φ (line 4). Next, the algorithm needs to identify all variables that depend on x. For this, the algorithm iterates over all clauses containing either x or $\neg x$ (line 5), and each literal ℓ in those clauses is compared to x (lines 6-8). If ℓ refers to a different variable v, and v is an authorised candidate, then v must be frozen.

4.2.2 Parallel Variable Elimination

After Φ has been constructed, a kernel is launched to compute an upper bound for the number of resolvents (excluding tautologies) that may be produced by eliminating variables in Φ. This kernel accumulates the number of resolvents of each variable using parallel reduction in shared memory within thread blocks. The resulting values (resident in shared memory) of all blocks are added up by atomic operations, resulting in the final output, stored in global memory (denoted by $|\tilde{\mathcal{S}}|$). Afterwards, the CNF formula $\mathcal{S}$ is reallocated according to the extra memory needed. The parallel BVE kernel (Algorithm 4.3) is now ready to be performed on the GPU, considering both the *resolution rule* and *gate-equivalence reasoning*. In Algorithm 4.3, first, each thread selects a variable in Φ, based on *tid* (lines 1-2). The `eliminated` array marks the variables that have been eliminated. It is used to distinguish eliminated and non-eliminated variables when executing Algorithm 4.4.

Each thread checks the control condition at line 3 to determine whether the number of resolvents $(h[x] \times h[\neg x])$ of x will be less than the number of deleted clauses $(h[x] + h[\neg x])$. If the condition evaluates to **`true`**, a list `resolvents` is created in shared memory, which is then added to the simplified formula $\tilde{\mathcal{S}}$ in global memory after discarding tautologies (lines 4-6). The markDeleted routine marks resolved clauses as deleted. They are actually deleted on the host side, once the algorithm has terminated.

At line 8, definitions of AND and OR gates are deduced by the gateReasoning routine, and stored in shared memory in `defs`. If at least one gate definition is found, the clauseSubstitution routine substitutes the involved variable with the underlying definition (line 9), creating the resolvents.

In some situations, even if $h[x]$ and $h[\neg x]$ are greater than 1, the number of resolvents can be less than the number of deleted clauses, when a sufficient number of resolvents are tautologies which are subsequently discarded. For this reason, we provide a third alternative to look ahead for tautologies in order to conclusively decide whether to resolve a variable if the conditions at lines 3 and 8 both evaluate to **false**. This third option (line 13) has lower priority than gate equivalence reasoning (line 8), since the latter in practice tends to perform more reduction than the former.

The sequential running time of Algorithm 4.3 is $\mathcal{O}(|C| \cdot |\Phi|)$, where $|C|$ is the length of a resolved clause C in $\mathcal{S}$. In practice, $|C|$ often ranges between 2 and tenths of literals. Therefore, the worst case is linear w.r.t. $|\Phi|$. Consequently, the parallel complexity is $\mathcal{O}(|\Phi|/p)$, where p is the number of threads. Since a GPU is capable of launching thousands of threads, that is, $p \approx |\Phi|$, the parallel

Algorithm 4.3: BVIPE

```
   Input  : global $\mathcal{S}, \Phi, h, \mathcal{T}$
   Input  : shared resolvents, defs
   Output : global $\tilde{\mathcal{S}}$, eliminated
 1 for all $tid \in [\![ 0, |\Phi| ]\!]$ do in parallel
 2     $x \leftarrow \Phi[tid]$, eliminated$[x] \leftarrow$ false, $numTautologies \leftarrow 0$
 3     if $h[x] = 1 \lor h[\neg x] = 1$ then
 4         resolvents $\leftarrow$ RESOLVE$(x, \mathcal{S}, \mathcal{T}[x], \mathcal{T}[\neg x])$
 5         MARKDELETED$(\mathcal{S}, \mathcal{T}[x], \mathcal{T}[\neg x])$
 6         $\tilde{\mathcal{S}} \leftarrow \mathcal{S} \cup$ resolvents
 7         eliminated$[x] \leftarrow$ true
 8     else if defs $\leftarrow$ GATEREASONING$(x, \mathcal{S}, \mathcal{T}[x], \mathcal{T}[\neg x]) \neq \emptyset$ then
 9         resolvents $\leftarrow$ CLAUSESUBSTITUTION$(x, \mathcal{S}, \mathcal{T}[x], \mathcal{T}[\neg x]$, defs$)$
10         MARKDELETED$(\mathcal{S}, \mathcal{T}[x], \mathcal{T}[\neg x])$
11         $\tilde{\mathcal{S}} \leftarrow \mathcal{S} \cup$ resolvents
12         eliminated$[x] \leftarrow$ true
13     else
14         $numTautologies \leftarrow$ TAUTOLOGYLOOKAHEAD$(x, \mathcal{S}, \mathcal{T}[x], \mathcal{T}[\neg x])$
15         $numResolvents \leftarrow h[x] \times h[\neg x]$, $numDeleted \leftarrow h[x] + h[\neg x]$
16         if $(numResolvents - numTautologies) < numDeleted$ then
17             resolvents $\leftarrow$ RESOLVE$(x, \mathcal{S}, \mathcal{T}[x], \mathcal{T}[\neg x])$
18             MARKDELETED$(\mathcal{S}, \mathcal{T}[x], \mathcal{T}[\neg x])$
19             $\tilde{\mathcal{S}} \leftarrow \mathcal{S} \cup$ resolvents
20             eliminated$[x] \leftarrow$ true
21         end
22     end
23 end
```

complexity is an amortized constant $\mathcal{O}(1)$. For Algorithm 4.3, we prove the following property.

Lemma 4.1. For any given CNF formula $\mathcal{S}$, BVIPE has no data racing and its results are identical to the ones obtained by the sequential elimination.

Proof. $\tilde{\mathcal{S}}$ is produced by threads executing lines 6, 11, and 19 in Algorithm 4.3 when applicable. The resolvents added to $\tilde{\mathcal{S}}$ are produced at lines 4, 9, and 17. Clearly, since each thread works on a different $x \in \Phi$, the threads produce different sets of resolvents. Moreover, since the variables in Φ are independent of each other, it is guaranteed that for each pair of variables $x, y \in \Phi$, there exists no clause $C \in \mathcal{S}$ with both $x \in C$ and $y \in C$. By the definition of S_x and $\mathcal{S}_{\neg x}$, it follows that $S_x \cup \mathcal{S}_{\neg x}$ contains no clauses C with $y \in C$. Let thread t be the thread assigned to x. If applicable, t executes either line 4, 9 or 17, producing *resolvents*, which are derived from S_x and $\mathcal{S}_{\neg x}$, hence no clauses are produced containing y. Therefore, t does not produce any clauses containing variables that are elected for removal, hence the resolvents added by t to $\tilde{\mathcal{S}}$ do not need to be processed by other threads. By the same reasoning, the clauses marked for deletion by t at lines 5, 10, and 18 do not need to be processed by other threads. Hence, the threads work independently, meaning that the sequential execution of their operations is guaranteed to produce the same result as when they are executed in parallel. $\square$

Proposition 4.1. BVIPE is confluent.

Proof. It follows from Lemma 4.1 that BVIPE is confluent, i.e., the actual order in which the variables in Φ are eliminated does not influence the resulting $\tilde{\mathcal{S}}$. $\square$

4.2.3 Parallel Subsumption Elimination

The parallel SUB (PSUB) is executed on elected variables that could not be eliminated earlier by variable elimination. (Self)-subsumption elimination tends to reduce the number of occurrences of non-eliminated variables as it usually removes many literals. After performing PSUB (Algorithm 4.4), the BVIPE algorithm is executed again, after which PSUB can be executed, and so on, until no more literals can be removed (the inner loop at the stage 1 of Figure 4.2).

The parallelism in the PSUB kernel is achieved on the variable level. In other words, each thread is assigned to a variable when executing PSUB. At line 3, previously eliminated variables are skipped. At line 5, a new clause is loaded, referenced by $\mathcal{T}[x]$, into shared memory (we call it the *shared clause*, C_s).

The shared clause is then compared in the loop at lines 4-19 to all clauses referenced by $\mathcal{T}[\neg x]$ to check whether x is a self-subsuming literal using SIG and SELFSUB functions. The SIG function compares the identifiers of two clauses. The identifier of a clause is computed by hashing its literals to a 32-bit value [EB05]. It has the property of refuting many non-subsuming clauses, but not all of them since hashing collisions may occur. The SELFSUB routine run a set intersection algorithm in linear time. If the conditions at lines 7 or 9 evaluate to **true**, both the original clause C, which resides in the global memory, and C_s must be strengthened (via the STRENGTHEN function). At line 11, the clause C' is marked for deletion because it became subsumed by the strengthened clause C. The condition $|C| > 2$ is tested at lines 7 and 9 to avoid producing unit clauses if C is strengthened. Propagation of these units may cause data racing between threads; thus, it was not yet implemented in SIGMA. In the next chapter, we

Algorithm 4.4: Parallel SUB

```
   Input   : global $\mathcal{S}, \Phi$, eliminated, $\mathcal{T}$
   Input   : shared $C_s$
1  for all $tid \in [\![ 0, |\Phi| ]\!]$ do in parallel
2      $x \leftarrow \Phi[tid]$
3      if $\neg$eliminated$[x]$ then
4          foreach $C \in \mathcal{S}[\mathcal{T}[x]]$ do
5              $C_s \leftarrow C$
6              foreach $C' \in \mathcal{S}[\mathcal{T}[\neg x]]$ do
7                  if $|C| > 2 \wedge |C'| < |C| \wedge \textsc{sig}(C', C) \neq 0 \wedge \textsc{selfsub}(C', C_s)$ then
8                      $\textsc{strengthen}(C, C_s, x)$
9                  else if $|C| > 2 \wedge |C'| = |C| \wedge \textsc{sig}(C', C) \neq 0 \wedge \textsc{selfsub}(C', C_s)$ then
10                     $\textsc{strengthen}(C, C_s, x)$
11                     $\textsc{markDeleted}(C')$
12                 end
13             end
14             foreach $C'' \in \mathcal{S}[\mathcal{T}[x]]$ do
15                 if $|C''| \leq |C| \wedge \textsc{sig}(C'', C) \neq 0 \wedge \textsc{subsume}(C'', C_s)$ then
16                     $\textsc{markDeleted}(C)$
17                 end
18             end
19         end
20     end
21 end
```

present a possible solution to propagate these units safely. In the loop at lines 14-18, the strengthened C_s is used for subsumption checking.

Regarding the complexity of Algorithm 4.4, the worst-case is that a variable x occurs in all clauses of $\mathcal{S}$. However, in practice, the number of occurrences of x is bounded by the threshold value μ (see Definition 4.1). The same applies for its complement. Therefore, worst case, a variable and its complement both occur μ times. As PSUB considers all variables in Φ and worst case has to traverse each loop μ times, its sequential complexity is $\mathcal{O}(|\Phi| \cdot \mu^2)$ and its parallel complexity is $\mathcal{O}(\mu^2)$. The SUB mode (second stage in Figure 4.2) allows SUB to be executed independently of BVE (i.e. applied directly on the input formula without allocating any extra memory).

Lemma 4.2. For any given CNF formula $\mathcal{S}$, PSUB has no data racing and its results are identical to the ones obtained by the sequential SUB.

Proof. Suppose we have the elected candidates $x, y \in \Phi$ and the parallel threads t_1, t_2. Let t_1 and t_2 be responsible for applying the SUB function (depicted by lines 4-19) on x and y respectively. It suffices to prove that the threads work on disjoint sets of clauses $\mathcal{S}_x, \mathcal{S}_{\neg x}$ and $\mathcal{S}_y, \mathcal{S}_{\neg y}$. Since $x, y \in \Phi$, then $(\mathcal{T}[x] \cup \mathcal{T}[\neg x]) \cap (\mathcal{T}[y] \cup \mathcal{T}[\neg y]) = \emptyset$ (from lines 4, 6, and 14). As such, $(\mathcal{S}_x \cup \mathcal{S}_{\neg x}) \cap (\mathcal{S}_y \cup \mathcal{S}_{\neg y}) = \emptyset$. Therefore, t_1 and t_2 do not influence each other's results. □

Proposition 4.2. PSUB is confluent.

Proof. It follows from Lemma 4.2 and Proposition 2.3 that PSUB is confluent, i.e., the order in which the clauses are checked for (self-)subsumption does not influence the resulting simplified formula. □

4.2.4 Parallel Blocked Clause Elimination

Parallel BCE (PBCE) is executed on an extended list of elected candidates Φ to increase the possibility of finding *blocked clauses* in the input formula. The extended list is obtained by performing the elections again with a large value of μ (second stage in Figure 4.2). In a similar way to the SUB, we launch a kernel to scan the occurrence lists of elected candidates in parallel looking for blocked clauses. Algorithm 4.5 shows how PBCE can be implemented.

The control condition $|\mathcal{T}[x]| \leq |\mathcal{T}[\neg x]|$ at line 4 maximizes the locality of the shared clause C_s, as clauses referred by the shorter occurrence list are frequently shared to others. That is, if the positive occurrence of x (i.e. C that has the literal x) is loaded to the shared memory, then the number of accesses to C_s will be $|\mathcal{T}[\neg x]|$. The **else** statement at line 5 handles the opposite of the former condition.

Each thread at lines 8-18 in the BLOCKED function compares the shared clauses having an occurrence of ℓ with all clauses that have the complement of ℓ (opposite occurrence), looking for tautologies. If the number of tautologies is equal to the number of opposite occurrences, then C is deleted. The parallel complexity of Algorithm 4.5 is $\mathcal{O}(\mu^2)$.

For Algorithm 4.5, we prove the following property.

Lemma 4.3. For any given CNF formula $\mathcal{S}$, PBCE has no data racing and its results are identical to the ones obtained by the sequential BCE.

Proof. Suppose we have the elected candidates $x, y \in \Phi$ and the parallel threads t_1, t_2. Let t_1 and t_2 be responsible for applying the BCE function (depicted by lines 8-18) on x and y respectively. It suffices to prove that the threads work on disjoint sets of clauses $\mathcal{S}_x, \mathcal{S}_{\neg x}$ and $\mathcal{S}_y, \mathcal{S}_{\neg y}$. Since $x, y \in \Phi$, then $(\mathcal{T}[x] \cup \mathcal{T}[\neg x]) \cap (\mathcal{T}[y] \cup \mathcal{T}[\neg y]) = \emptyset$ (from lines 9 and 11). As such, $(\mathcal{S}_x \cup \mathcal{S}_{\neg x}) \cap (\mathcal{S}_y \cup \mathcal{S}_{\neg y}) = \emptyset$. Therefore, t_1 and t_2 do not influence each other's results. □

Algorithm 4.5: Parallel BCE

```
Input : global $\mathcal{S}$, $\Phi$, eliminated, $\mathcal{T}$
Input : shared $C_s$
for all $tid \in [\![\, 0, |\Phi| \,]\!]$ do in parallel
    $x \leftarrow \Phi[tid]$
    if $\neg$eliminated$[x]$ then
        if $|\mathcal{T}[x]| \leq |\mathcal{T}[\neg x]|$ then BLOCKED($x$)
        else BLOCKED($\neg x$)
    end
end
device function BLOCKED($\ell$):
    foreach $C \in \mathcal{S}[\mathcal{T}[\ell]]$ do
        $numTautologies \leftarrow 0$, $C_s \leftarrow C$
        foreach $C' \in \mathcal{S}[\mathcal{T}[\neg\ell]]$ do
            if ISTAUTOLOGY($x, C_s, C'$) then
                $numTautologies \leftarrow numTautologies + 1$
            end
        end
        if $numTautologies = |\mathcal{T}[\neg\ell]|$ then MARKDELETED($C$)
    end
end
```

Proposition 4.3. PBCE is not confluent.

Proof. It follows from Algorithm 4.5 that a set of clauses is chosen based on a heuristic (condition at line 4), i.e., the parallel algorithm has no fixpoint. Further, we have shown by Proposition 2.4 that BCE is not confluent. Thus, PBCP is not confluent. ☐

4.2.5 Hidden Redundancy Elimination

We propose a new simplification technique called *hidden redundancy elimination* (HRE) which repeats the following until fixpoint has been reached: for a given formula $\mathcal{S}$ and clauses $C_1 \in \mathcal{S}, C_2 \in \mathcal{S}$ with $\ell \in C_1$ and $\neg\ell \in C_2$, if there exists a clause $C \in S$ for which $C \equiv C_1 \otimes_\ell C_2$ and C is not a tautology, then let $\mathcal{S} := \mathcal{S} \setminus \{C\}$. The clause C is called a *hidden redundancy* and can be removed without altering the original satisfiability.

Lemma 4.4. For any formula $\mathcal{S}$, HRE($\mathcal{S}$) is logically equivalent to $\mathcal{S}$.

Proof. Suppose we have the clauses $\{C, C_1, C_2\} \subseteq \mathcal{S}$ where $C_1 = C_1' \cup \{\ell\}$, $C_2 = C_2' \cup \{\neg\ell\}$, $\ell \notin C_1'$, $\neg\ell \notin C_2'$ and $C = C_1' \cup C_2'$. Now, if ℓ is set to $\top$, then

$C_1 \models \top$ and $C_2 = C_2'$, implying $C_2 \subseteq C$. If $\ell \models \bot$, then $C_2 \models \top$ and $C_1 = C_1'$, implying $C_1 \subseteq C$. Therefore, in both cases, clause C is subsumed by either C_1 or C_2. Hence $\mathcal{S} \setminus \{C\}$ is logically equivalent to $\mathcal{S}$. □

Example 4.1. For example, consider the formula

$$\mathcal{S} = \{\{a, \neg c\}, \{c, b\}, \{\neg d, \neg c\}, \{b, a\}, \{a, d\}\}$$

Resolving the first two clauses gives the resolvent $\{a, b\}$ which is equivalent to the fourth clause in $\mathcal{S}$. Also, resolving the third clause with the last clause yields $\{a, \neg c\}$ that is equivalent to the first clause in $\mathcal{S}$. HRE can remove either $\{a, \neg c\}$ or $\{a, b\}$ but not both.

Proposition 4.4. HRE is not confluent.

Proof. Example 4.1 shows that the order in which the variables $\{c, d\}$ are resolved, influences the resulting simplified formula. Thus, HRE is not confluent. □

Parallel Algorithm

Similar to the algorithms explained earlier, parallel HRE is also performed on an extended version of Φ to increase the chance of finding hidden redundancies. A *2-dimensional* kernel (e.g. each thread is identified by its $\boldsymbol{x}$- and $\boldsymbol{y}$-*dimension* indices) executes the following operations on the elected candidates:

1. Each thread selects a variable $x \in \Phi$ based on its $\boldsymbol{y}$ ID ($tid_{\boldsymbol{y}}$), and checks for clauses being mergeable w.r.t. x (lines 1-8).

2. Each merged clause (C_m) is inspected for equality against all clauses referenced by the literals inside C_m (we call this operation, the *forward equality check*). This check is done at lines 9-11 by threads using their $\boldsymbol{x}$ ID ($tid_{\boldsymbol{x}}$).

3. If a clause is found equal to C_m in the input formula, then the former is removed (lines 12-14).

Algorithm 4.6 describes in detail how those operations are implemented. Notice that the resolved clause (C_m) obtained by the function RESOLVE at line 7 is only needed for the equality check and not added to the simplified formula. The loop at line 9 sweeps all literals in C_m. The threads with $\boldsymbol{x}$ ID search for a successful match among the clauses referenced (via the occurrence lists) by the

Algorithm 4.6: Parallel HRE

```
   Input :global $\mathcal{S}, \Phi$, eliminated, $\mathcal{T}$
   Input :shared $C_s, C_m$
1  for all $tid_y \in [\![\, 0, |\Phi|\, ]\!]^y$ do in parallel
2      $x \leftarrow \Phi[tid_y]$
3      if $\neg$eliminated$[x]$ then
4          foreach $C \in \mathcal{S}[\mathcal{T}[x]]$ do
5              $C_s \leftarrow C$
6              foreach $C' \in \mathcal{S}[\mathcal{T}[\neg x]]$ do
7                  $C_m \leftarrow$ RESOLVE$(x, C_s, C')$
8                  if $C_m \neq \emptyset$ then
                       /* $\forall \acute{C} \in \mathcal{S}$ with $\ell \in \acute{C}$ do the forward equality check    */
9                      foreach $\ell \in C_m$ do
10                         for all $tid_x \in [\![\, 0, |\mathcal{T}[\ell]|\, ]\!]^x$ do in parallel
11                             $\acute{C} \leftarrow \mathcal{S}[\mathcal{T}[\ell][tid_x]]$
12                             if $|\acute{C}| = |C_m| \wedge$ SIG$(\acute{C}, C_m) \neq 0 \wedge \acute{C} \equiv C_m$ then
13                                 MARKDELETED$(\acute{C})$, break
14                             end
15                         end
16                         if DELETED$(\acute{C})$ then break
17                     end
18                 end
19             end
20         end
21     end
22 end
```

literals in the merged clauses. If there is a hit, then the loop is broken assuming that the same clause cannot appear in the input formula multiple times.

The worst-case parallel complexity of this algorithm is

$$\mathcal{O}(\frac{|\Phi|}{p_y}(\underbrace{\mu^2}_{\text{loops at lines 4,6}} \overbrace{(|C_m|(}^{\text{loop at line 9}}\frac{\mu}{p_x}))))$$

where $|C_m|$ is the length of the merged clause and p_y, p_x are the number of threads in dimensions $\boldsymbol{y}$ and $\boldsymbol{x}$, respectively. By launching sufficient threads in both dimensions with $p_x \ll p_y$ and assuming that μ is considerably larger than p_x, the upper bound of the parallel loop at line 10 cannot be neglected.

Thus, the parallel complexity will be $\mathcal{O}(|C_m|(\mu^3/p_x))$. That is, the parallel running time is a cubic function of the occurrences upper-bound of x or $\neg x$. In fact, the loop at lines 9-17 could be performed in parallel with threads operating in a third dimension (***z**-dimension*). However, this will impose us to limit the number of threads in the other two dimensions because the block dimensionality (i.e. number of threads per block) is constrained by the GPU compute capability to 1024 threads only (see Table 3.2), which in turn, decreases the overall performance of a *3-dimensional* kernel.

For Algorithm 4.6, we prove the following property.

Lemma 4.5. For any given CNF formula $\mathcal{S}$, PHRE has no data racing and its results are identical to the ones obtained by the sequential HRE.

Proof. Suppose we have the elected candidates $x, y \in \Phi$ and the parallel threads t_1, t_2 with $\boldsymbol{y}$ IDs. Let t_1 and t_2 be responsible for applying the HRE function (depicted by lines 4-20) on x and y respectively. It suffices to prove that the threads work on disjoint sets of clauses $\mathcal{S}_x, \mathcal{S}_{\neg x}$ and $\mathcal{S}_y, \mathcal{S}_{\neg y}$. Since $x, y \in \Phi$, then $(\mathcal{T}[x] \cup \mathcal{T}[\neg x]) \cap (\mathcal{T}[y] \cup \mathcal{T}[\neg y]) = \emptyset$ (from lines 4 and 6). As such, $(\mathcal{S}_x \cup \mathcal{S}_{\neg x}) \cap (\mathcal{S}_y \cup \mathcal{S}_{\neg y}) = \emptyset$. Moreover, by Definition 4.4, all literals in the resolvent C_m obtained by line 7 are frozen. That is, if a redundant clause equivalent to C_m is found and removed by an $\boldsymbol{x}$ thread (lines 10-15), the removed clause cannot be resolved again by either t_1 or t_2. Therefore, t_1 and t_2 do not influence each other's results. $\square$

4.2.6 Multi-GPU Support

By default SIGmA runs on the first GPU of the computing machine, i.e., the one installed on the first Peripheral Component Interconnect Express (PCI-E) bus, which we refer to as GPU_0. SIGmA is also capable of running SAT simplification on n of the same-type GPUs installed in the same machine. When $n > 1$, the elected variables in Φ are distributed evenly among the GPUs. The complexity of processing an elected variable x during simplification depends on the number of occurrences of literals x and $\neg x$ in the formula. As SIGmA uses μ as upper-bound to the number of occurrences, the complexity of applying SAT simplification for a single variable is $\mathcal{O}(\mu^2)$. When distributing SAT simplification over multiple GPUs, ensuring that the associated work is well balanced is important. Therefore, we balance the number of variable occurrences each GPU needs to process. Algorithm 4.7 presents our Shuffle Distribution algorithm. The sortDescending function sorts the elements in Φ in descending order based on the number of occurrences, as provided by function h, and the result

Algorithm 4.7: Shuffle Distribution of Φ

```
Input  : $\Phi, n, h$
Output : $\Phi_0, \ldots, \Phi_{n\text{-}1}$
1  $\Phi_0, \ldots \Phi_{n\text{-}1} \leftarrow \emptyset$
2  $\mathit{forward} \leftarrow$ true, $i \leftarrow 0$
3  $L \leftarrow$ sortDescending($\Phi, h$)
4  foreach $x \in L$ do
5      $\Phi_i \leftarrow \Phi_i \cup \{x\}$
6      if $\mathit{forward}$ then
7          $i \leftarrow i + 1$
8          if $i = n$ then
9              $\mathit{forward} \leftarrow$ false, $i \leftarrow i - 1$
10         end
11     else
12         $i \leftarrow i - 1$
13         if $i = -1$ then
14             $\mathit{forward} \leftarrow$ true, $i \leftarrow i + 1$
15         end
16     end
17 end
```

is stored in the list L. In the loop at lines 4-17, the variables are considered in that order. Index i always points to the GPU device to which the currently considered variable needs to be assigned. This index is updated in each iteration such that it ping-pongs between values 0 and $n - 1$.

Figure 4.2 shows an example of such a distribution with three GPUs, which is applied after LCVE in the first stage. The variables are ordered by the number of literals in the formula; the variable with index 5 is the *least constrained* (LCV), i.e., has the smallest number, while the last one with index 2 is the most constrained variable (MCV), i.e., has the largest number of occurrences. Starting with the MCV, the first three variables are distributed over the three GPUs, followed by the next three, which are distributed in reversed order, etc. As the number of literals is indicative of the computational effort needed to eliminate a variable, this distribution method achieves good load balancing.

In Figure 4.2, the global barrier acts as a synchronization point for all GPUs, and in the subsequent *merge* step, all simplified subformulas of the various GPUs are communicated to GPU_0. HRE cannot be run on multiple GPUs, because the entire formula must be accessed.

4.3 Benchmarks and Analysis

We evaluated all GPU simplifications on an NVIDIA Titan Xp GPU. See Table 3.2 for the main specifications. For multi-GPU experiments, we used two NVIDIA Titan Xp GPUs installed in the same machine. The GPU machine was running Linux Mint v18.

We selected 344 SAT problems from the application track of the 2013, 2016 and 2017 SAT competitions[10]. This set consists of all problems from those tracks that are more than 1 MB in file size. The largest size of problems occurring in this set is 1.2 GB. These problems have been encoded from 46 different real-world applications that have a whole range of dissimilar logical properties. Before applying any simplifications using the experimental tools, any occurring unit clauses were propagated. The presence of unit clauses immediately leads to simplification of the formula. By only considering formulas without unit clauses, the benchmark results directly indicate the true impact of our preprocessing algorithms.

In the benchmark experiments, besides the implementations of our new GPU algorithms, we involved a CPU-only version of SIGmA, the SatElite preprocessor [EB05], the MiniSat and Lingeling [Bie13] SAT solvers for the solving of problems, and executed these on the compute nodes of the DAS-5 cluster [BEdL+16]. Each problem was analysed in isolation on a separate computing node, with a timeout of 24 hours. Each computing node had an Intel Xeon E5-2630 CPU running at a core clock speed of 2.4 GHz with 128 GB of system memory, and runs on the CentOS 7.4 operating system. We performed the equivalent of 6 years of uninterrupted processing on a single node to measure how SIGmA impacts SAT solving in comparison to using sequential simplification or no simplification.

The benchmarks are categorised according to the following performance considerations:

- *SAT-Simplification Benchmarks* evaluate SIGmA against the sequential counterparts, SatElite and Lingeling preprocessors. Moreover, we analysed the amount of reductions obtained by SIGmA using different number of `phases` of the first stage (see Figure 4.2).

- *SAT-Solving Benchmarks* provide a thorough assessment of the proposed SAT simplifications with multiple execution orders and combinations on SAT

[10]http://www.satcompetition.org

Table 4.1: SIGmA acceleration with single/multi-GPU configuration. The +t_c and -t_c notations denote with and without communication t_c, respectively.

Configuration	**SIGmA (GPU vs CPU)**						**SIGmA (2 GPUs vs 1 GPU)**				
Method	**ve**	**sub**	**bce**	**hre**	**all**(-t_c)	**all**(+t_c)	**ve+** **ve**	**ve+** **sub**	**bce**	**all**(-t_c)	**all**(+t_c)
Average speedup	37×	5×	29.5×	17×	14×	7.3×	1.7×	2.64×	1.34×	1.96×	0.85×
Simplified faster (%)					330 (95)					79 (22)	

Table 4.2: SIGmA performance compared to various simplifiers

SIGmA (mode)	**Counterpart**	**Speedup**	**Simplified faster (%)**
CPU: ve+/sub	SatElite	36.96×	339 (98)
GPU: ve+/sub	SatElite	69.25×	339 (98)
GPU: all	SatElite	49.32×	326 (94)
GPU: all	Lingeling	32.19×	315 (91)

solving. The results are compared to MiniSat without simplification, SatElite + MiniSat, and Lingeling.

4.3.1 SAT-Simplification Benchmarks

For all experiments, we set μ initially to 64, which in practice tends to produce good results (i.e. balance the workload on the GPU with the amount of reductions). Table 4.1 summarises the amount of speedup and the number of problems simplified faster (last row) by running SIGmA on one and two GPUs[11]. We compare the one GPU mode with the CPU-only and two GPUs mode. For each of those options, we list the achieved speedup when running them on a GPU instead of a CPU. For the comparison of single vs. two GPUs, we disabled `hre`, since it is only supported in single GPU mode. Compared to CPU-only SIGmA, the GPU achieves an acceleration of up to 37×. The average speedup of `all` methods with and without communication (t_c) is 14× and 7.3×, respectively, allowing 330 problems to be simplified faster (95%).

If we ignore the time needed for data transfer between the GPUs (t_c), SIGmA scales very well when the number of GPUs is increased. In mode SUB, the average acceleration is 2.64×, and overall, the average speedup is 1.96×. When

[11]Tables with all the data are available at http://gears.win.tue.nl/software.

Table 4.3: SIGmA reductions with different phases on the largest 34 cases. The V, C, and L symbols denote the number of variables, clauses, and literals (in hundreds), respectively.

CNF	**Original**			**SIGmA (first stage - 1 GPU)**								
				1 phase			**3 phases**			**5 phases**		
	V	**C**	**L**	**V**	**C**	**L**	**V**	**C**	**L**	**V**	**C**	**L**
9dlx_vliw_at_b_iq9	4820	96763	281949	4394	95910	283783	4377	95865	284599	4377	95865	284599
9vliw_m_9stages_iq3_C1_bug1	5211	133786	392048	5054	133470	392417	5052	133464	392452	5052	133464	392452
SAT_dat.k85-24_1_rule_3	11637	45622	114601	6343	34573	97884	5743	33494	98241	5743	33494	98241
ak128modbtbg1btaig	5730	17169	40061	2380	9572	27070	1881	8580	26234	1881	8580	26234
atco_enc3_opt2_18_44	10825	43268	107956	10767	43173	107794	10764	43158	107756	10763	43155	107753
barman-pfile10-038.sas.ex.15	3596	9119	602748	1408	7005	598552	1030	6535	597900	1030	6535	597900
barman-pfile10-040.sas.ex.15	3597	9120	602767	1409	7005	598568	1030	6534	597913	1030	6534	597913
cube-11-h13-unsat	4530	13635	31833	1781	8415	24511	1413	8021	24652	1413	8021	24652
dspam_dump_vc950	1091	3561	9492	639	2764	8252	608	2690	8074	601	2671	8021
esawn_uw3.debugged	122007	480499	1331270	77303	372710	1109428	73686	362234	1084505	memory out		
gaussian.c.75.smt2-cvc4	19871	85610	234893	14269	77171	231277	14160	76802	230403	14157	76741	230340
hwmcc15deep-beemfwt4b1-k48	23008	50894	117550	9454	31967	85745	7537	29821	84499	7333	29585	84709
ibm-2002-23r-k90	2094	8646	21760	1154	6726	19177	1059	6506	19055	1057	6491	19002
itox_vc1130	1412	4276	11247	645	2684	8309	523	2410	7755	502	2351	7637
manol-pipe-f10ni	3688	11004	25678	1218	5470	15996	1018	5077	15926	1018	5077	15926
newton.8.3.i.smt2-cvc4	3027	13096	36257	2115	10978	32710	2075	10866	32563	2073	10856	32564
openstacks-sequencedstrips-p30	1039	4042	9511	642	3797	9029	582	3746	8927	582	3746	8926
partial-10-19-s	2691	12824	30338	2072	11244	29028	1996	10911	28484	1982	10882	28426
pipe_16	2664	194700	579431	2534	194437	579343	2531	194428	579517	2531	194428	579517
q_query_3_L200_coli.sat	5428	30037	89433	2624	21372	70007	2415	20191	65940	2414	20186	65927
safe030-h29-unsat	1290	4377	10566	468	2737	8116	449	2705	8074	449	2704	8073
sin2.c.20.smt2-cvc4	19629	79547	216386	12241	62616	186580	11924	61910	185625	11886	61821	185748
sin2.c.75.smt2-cvc4	79169	325919	894196	51467	262387	782177	50280	259743	778584	50134	259410	779003
sncf_model_ixl_bmc_depth_15	18016	59139	146686	10566	43846	121519	9220	40268	114953	8878	39173	112332
sokoban-p20.sas.ex.23	36329	72587	481190	10097	46146	429294	3624	39109	417396	2623	37433	416074
sortnet-8-ipc5-h18-unsat	3382	11868	28847	1271	7672	22769	1216	7612	22815	1216	7612	22815
test_v7_r17_vr5_c1_s25451	5603	25091	70028	4086	22218	66211	4040	22058	66156	4040	22058	66156
transport-1city-35n-50	14694	74791	177446	10441	72889	173784	10293	72791	173587	10292	72789	173585
valves-gates-1-k617-unsat	9704	30623	78421	4737	20299	59043	3712	17215	51362	3629	16874	50414
				SIGmA (first stage - 2 GPUs)								
T106.2.0	83638	292588	758482	49865	224148	657833	45406	216087	652754	45369	216004	652704
T50.2.0	83638	292589	758484	49865	224148	657834	45406	216088	652755	45369	216004	652704
T96.1.1	89058	323225	861424	59090	261471	767722	56853	257007	762781	56825	256959	762825
qurt.c.20.smt2-cvc4	79564	340538	937006	52842	283754	848135	51877	281596	845438	51760	281327	845760
esawn_uw3.debugged	122007	480499	1331270	77303	372710	1109428	73650	362196	1084465	72624	357793	1072039

we consider data transfer, the latter drops to 0.85×. Still, in two GPUs mode, SIGmA managed to simplify 79 (22%) problems faster compared to the single GPU mode, even if the communication overhead is taken into account. Hardware improvements of inter-GPU communication will make the multi-GPU mode of

Table 4.4: SIGmA impact on MiniSat solving performance

Solver	**Total solved (%)**	**Solved faster (%) (vs. MiniSat)**	**Solved faster (%) (vs. SatElite + MiniSat)**	**Processing Time (hr.)**
MiniSat	192 (56)	—	—	3989
SatElite + MiniSat	215 (62)	97 (50)	—	3474
SIGmA + MiniSat	230 (67)	139 (72)	122 (57)	3214

SIGmA increasingly attractive in the future. Furthermore, the multi-GPU configuration may allow more reductions to be obtained, where a single GPU cannot due to limited memory capacity.

Table 4.2 compares SIGmA against the best sequential simplifiers available (e.g. SatElite and the preprocessing module integrated in Lingeling). Similar to the multi-GPU experiments, we used the heavy mode (`-ve+`) in combination with other simplifications in all of SIGmA benchmarks. On average, SIGmA on a single GPU, with all simplifications enabled, beats SatElite and Lingeling by accelerations up to 49× and 32×, respectively.

Table 4.3 provides an evaluation of the achieved simplifications with SIGmA using varying numbers of phases (up to five). The second part of the table shows the amount of reductions obtained by two GPUs. The first four problems (*T106, T50, T96, qurt*) could not be simplified at all by a single GPU using any number of phases due to lack of memory. The last problem (*esawn*) managed to be simplified by two GPUs up to five phases. In most problems, the reductions on (V, C, L) get more stable (no more can be obtained) after four or five phases. Therefore, for solving experiments, we set the number of phases to five while simplifying the benchmark set with SIGmA.

4.3.2 SAT-Solving Benchmarks

Figures 4.3 and 4.4 show the impact of different modes in SIGmA on SAT solving by MiniSat and Lingeling. The `org` line refers to the SAT solver time when running on the original formulas. The `ve+sub` mode is executed in several rounds until no more literals can be removed (`ve+` loop in Figure 4.2). It seems that both `ve+sub` and `ve+bce+hre` were competing with each other in solving the problem set using MiniSat. At the end, the latter mode wins because it solved more problems (from 228 to 230) as shown by the (plus)-marked line in Figure 4.3. For Lingeling, the modes `ve+bce` and `ve+bce+hre` allowed Lingeling to solve the same number of problems (283). However, the latter is slightly faster.

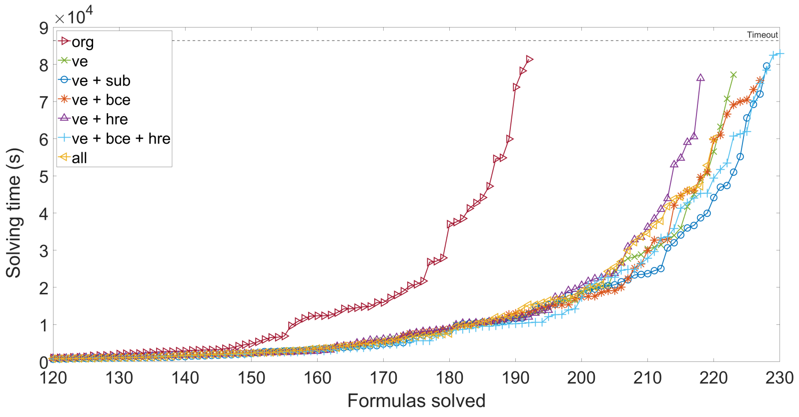


Figure 4.3: SIGMA impact on MINISAT for various simplification modes

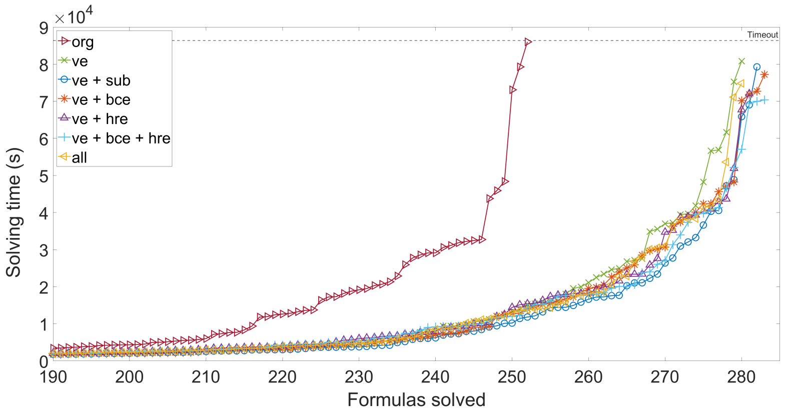


Figure 4.4: SIGMA impact on LINGELING for various simplification modes

Table 4.5: SIGmA impact on Lingeling solving performance

Solver	**Total solved (%)**	**Solved faster (%) (vs. Lingeling)**	**Processing Time (hr.)**
Lingeling	252 (73)	—	2566
SIGmA + Lingeling	283 (82)	128 (51)	1854

Tables 4.4 and 4.5 summarize the improvement of SIGmA on MiniSat and Lingeling using the best simplification mode `ve+bce+hre` with comparison to the original solvers and SatElite + MiniSat. In addition, we only show a summary for SatElite + MiniSat according to the best preprocessing mode `ve` that allowed the largest number of problems to be solved. The processing time (hr.) includes the simplification time if the formula is simplified and the timeout (24 hours) if the problem ran out of time. The percentages in the *total solved* column are based on the total number of problems (344). The percentages in *solved faster* columns are based on the minimum number of problems solved by the counterpart. For instance, SIGmA + MiniSat solved 139 and 122 problems faster than MiniSat only and SatElite + MiniSat by factors of 139/192 = 72% and 122/215 = 57% respectively. In Table 4.5, SIGmA allowed Lingeling to solve 57% of the problems faster than Lingeling only.

4.4 Related Work

Subbarayan *et al.* [SP04] have provided the first BVE technique based on the resolution rule of the search algorithm DPLL [DLL62], called NiVER. Eén *et al.* [EB05] extended NiVER with subsumption elimination and clause substitution. However, the subsumption check is only performed on the clauses resulting from variable elimination, hence no reductions are obtained if there are no variables to resolve. Zhang *et al.* [Zha05] presented an algorithm to provide a subsumption-free formula by detecting and removing subsumed clauses as the resolvents are being added to the simplified formula.

Gebhardt *et al.* [GM13] presented the first attempt to parallelise SAT preprocessing on a multi-core CPU using a locking scheme to prevent threads corrupting the SAT formula. However, they reported only a very limited speedup of on average 1.88× when running on eight cores. Heule *et al.* [HJB10] have introduced several approaches for clause elimination that can be effective in SAT simplifications, e.g. blocked clause, hidden blocked clause, and hidden tautology eliminations. Most of these methods produce extra reduction in terms of clauses and literals compared to former methods used in NiVER and SatElite.

All these methods introduce sound simplifications, yet none of them is implemented to fully utilise SIMT architectures such as GPUs. Because of this, they may consume considerable time in preprocessing when applied to large problems.

Finally, BVE as introduced in [SP04, EB05, JBH10], is not confluent, as noted by the authors Järvisalo *et al.* [JBH10]. Due to dependency between variables, altering the elimination order of these variables may result in different simplified formulas. This drawback is circumvented by our LCVE algorithm, which makes it possible to perform parallel variable elimination.

4.5 Conclusion

We have shown that SAT simplifications can be performed efficiently on many-core systems, producing impactful reductions in a fraction of a second, even for larger problems consisting of millions of variables and tens of millions of clauses. The proposed BVIPE algorithm provides the first methodology to eliminate multiple variables in parallel while preserving satisfiability. Furthermore, we have presented the SIGmA tool, the first simplifier for SAT formulas that exploits the power of GPUs. It can be configured to apply a combination of various elimination procedures, among which is a new one (HRE) proposed by us. Experimentally, we have demonstrated the impact of SIGmA on state-of-the-art SAT solving. In particular, our new mode, involving BCE and HRE, positively affects both the solving speed and the ability to solve formulas, when using the MiniSat and Lingeling solvers. Having multiple GPUs in a single machine as shown in this chapter, can definitely mitigate the lack of memory of a single GPU, particularly for extremely large CNF formulas. Balancing the workload of various simplification methods effectively across the available graphics processors is feasible, using the proposed work distribution scheme.

Concerning future work, the results of this chapter motivate us to take the capabilities of GPU SAT simplification further by supporting more simplification techniques and apply them frequently within the solving procedure (see Chapter 5). Regarding the multi-GPU setup, we will consider using a fast data link such as NVLink to speed up the GPU-to-GPU communication. We will also investigate the possibility of partitioning the workload in SAT simplifications across the CPU and all GPUs nodes as presented in [SVL+16] to achieve optimal performance.

Chapter **5**

SAT Inprocessing

"It's hardware that makes a machine fast. It's software that makes a fast machine slow."

– Craig Bruce

Since 2013, simplification techniques [SP04, EB05, GM13, HJB10, BJK21] are also used periodically *during* SAT solving, which is known as *inprocessing* [JHB12, Bie13, BFFH20, BGJ$^+$18]. Applying inprocessing iteratively to large problems can be a performance bottleneck in the SAT solving procedure, or even increase the size of the formula, negatively impacting the solving time. Both pre- and inprocessing have been an essential part of state-of-the-art CDCL SAT solvers [AS09, Bie13, LGPC16, BFFH20], particularly when applied on real-world applications relevant to software and hardware verification [OW21a, HS07, JBH12].

Contributions

In this chapter, we introduce the first SAT solver (ParaFROST) with GPU-accelerated inprocessing which supports various simplification rules to rewrite a SAT formula into a compact equisatisfiable one with fewer variables and/or clauses. Preprocessing is done only once before the solving starts (Chapter 4), while in inprocessing, this is done periodically during the solving. Embedding

GPU inprocessing in a SAT solver is highly non-trivial and has never been attempted before, according to the best of our knowledge. Robust data structures are needed that allow parallel processing, and that support efficient adding and removing of clauses. Further, we make use of the intra-warp intrinsic functions such as shuffle and warp voting machine instructions to exploit the GPU hardware capabilities in our implementations.

For this purpose, we list the following contributions:

- ⋆ We propose a new dynamically expandable data structure for clauses supporting both 32-bit [ES03a] and 64-bit references with a minimum of 20 bytes per clause. Further, a new parallel garbage collector is presented, tailored for GPU inprocessing to maximize locality and reduce memory consumption.
- ⋆ The HRE rule (see Section 4.2.5) is further extended in this chapter to deal with learnt clauses and its parallel implementation is improved. Thus, it is renamed to *eager redundancy elimination* or ERE.
- ⋆ Our new parallel BVE is twice as fast as Algorithm 4.3 and together with other improvements yields much higher performance and robustness. The latter algorithm in Chapter 4 could reduce the number of added clauses by detecting logical gates that are syntactically translated to CNF using Tseitin encoding (Section 2.1). Such a definition of gate output y is written as $y \leftrightarrow f(v_1, \ldots, v_n)$. The simplest example is the AND gate $y \leftrightarrow v_1 \wedge v_2$. To improve the functionality of regular gate detection (also called the syntactic approach), we reason about *equivalence*, *xor*, and *if then else* statements. Additionally, we provide a semantic way using function tables to detect any gate definition in parallel that could not be found by the syntactic method.
- ⋆ While the impact of accelerating these procedures is being investigated, their correctness in refuting a formula has not yet been addressed, especially if used in critical applications such as BMC (Chapter 7). If the solver claims that a formula is satisfiable, the generated solution (model) can be checked linearly in the size of the formula. However, if a solver declares a formula is unsatisfiable (i.e. has no solutions), there is no guarantee that the GPU code is sound and correct due to ill logic or data hazards that could be introduced at the implementation level. Certifying SAT solvers is becoming crucial to validate the results in tools such as theorem provers and model checkers. Therefore, we propose an effective parallel approach

to generate clausal proofs for the GPU-accelerated simplifications in DRAT (Deletion Resolution Asymmetric Tautology) format [WHH14, HJW13] with two goals: the proof should be compact and not pose an overhead to the GPU solver.

The chapter is organised as follows: the GPU data structures are discussed in Section 5.1. The proposed garbage collector and the memory management of our proof system are explained in Sections 5.2 and 5.3, respectively. Sections 5.4-5.9 introduce our new inprocessing techniques and a way to auto-tune their GPU kernels. Section 5.10 presents our experimental evaluation. Section 5.11 discusses related work, and Section 5.12 provides a conclusion and suggests future work.

5.1 GPU Memory and Data Structures

To efficiently implement inprocessing techniques (i.e., Variable-Clause Eliminations (VCE)) for GPU architectures, we designed a new data structure from scratch to consider learnt clauses, and store other relevant clause information. The new structure requires 16 bytes of bookkeeping, compared to 24 bytes consumed by our initial design for preprocessing in SIGMA excluding literals. Fig. 5.1 shows the proposed structures to store a clause (denoted by `SCLAUSE`) and the SAT formula represented in CNF form (denoted by `CNF`). The following information is stored for each clause:

- The `state` field (1 byte) stores if the state is `ORIGINAL`, `LEARNT` or `DELETED`.
- The `used` field (1 byte) keeps track of how many search iterations a `LEARNT` clause can still be used before it gets deleted during database reduction. `LEARNT` clauses are used at most twice [BFFH20].
- The `added` field (1 byte) is used to mark a clause as resolvent.
- The `flag` field (1 byte) marks the clause when it contributes to a gate (when applying substitution).
- The *literal block distance* (`lbd`) (4 bytes) stores the number of decision levels contributing to a conflict, if there is one [AS09]. This field is updated when the clause is altered. Both `used` and `lbd` can be altered via clause *strengthening* [BFFH20] in SUB.
- The `size` (4 bytes) of the clause, i.e., the number of literals.
- A signature `sig` (4 bytes) is a clause hash, for fast clause comparison [EB05].

In addition, a list of literals is stored, each literal taking 32 bits (1 bit to indicate whether it is negated or not, and 31 bits to identify the variable). In

```
class SCLAUSE {
    char state, flag;
    char added, used;
    int size, lbd;
    uint32 sig;
    uint32 literals[];
}
```

(a) container for a clause

```
class CNF {
    struct {
        uint32* memory;
        uint64 size, cap;
    } clauses;
    struct {
        uint64* memory;
        uint64 size, cap;
    } references;
}
```

(b) container for a formula

Figure 5.1: Data structures to store a SAT formula on a GPU

total, a clause requires $16 + 4t$ bytes, with t the number of literals in the clause. Previously, in our tool SIGMA, we stored a pointer in each clause referencing the first literal, with the literals being in a separate array. This consumes 8 bytes of the clause space. However, `SCLAUSE` does not require any bytes for the `literals` array, resulting in the clause occupying 16 bytes in total, including the extra information of the learnt clause, compared to 24 bytes in our previous work.

As implemented in MiniSat, we use the `clauses` field in `CNF` (Fig. 5.1b) to store the raw bytes of `SCLAUSE` instances with any extra literals in 4-byte buckets with 64-bit reference support. The `cap` variable indicates the total memory capacity available for the storage of clauses, and `size` reflects the current size of the list of clauses. We always have `size` $\leq$ `cap`. The `references` field is used to directly access the clauses by saving for each clause a reference to their first bucket. The mechanism for storing references works in the same way as for clauses.

In a similar way, an *occurrence table* structure, denoted by `OT`, is created which has a raw pointer to a member structure `OL`. The latter is designed to record the 64-bit clause references for each literal in the formula. The creation of an `OL` instance is done in parallel on the GPU for each literal using atomic instructions. For each clause C, a thread is launched to insert the occurrences of C's literals in the associated lists. More details on this are explained in Section 4.1.

Initially, we preallocate unified memory for `clauses` and `references` which is in size twice as large as the input formula, to guarantee enough space for the original and learnt clauses. This amount is assumed to be enough as we enforce that the number of resolvents never exceeds the number of `ORIGINAL` clauses. Later, when applying our solver in BMC, we found that this assumption is wrong and cannot guarantee the success of the BVE procedure. However, we propose a solution in Chapter 7 to allow the elimination of variables dynamically according only to the amount of memory space allocated, no matter how large it is. The `OT` memory is reallocated dynamically if needed after each variable elimination. Furthermore, we check the amount of free available GPU memory before allocation. If no memory is available, the inprocessing step is skipped and the solving continues on the CPU.

5.2 Parallel Garbage Collection

Modern sequential SAT solvers implement a *garbage collection* (GC) algorithm to reduce memory consumption and maintain data locality [ES03a, AS09, BFFH20]. Since GPU global memory is a scarce resource and coalesced accesses are essential to hide the latency of global memory (see Section 3.2), we decided to develop an efficient and parallel GC algorithm for the GPU without adding overhead to the GPU computations.

Figure 5.2 demonstrates the proposed approach for a simple SAT formula $\{\{a, \neg b, c\}, \{a, b, \neg c\}, \{d, \neg b\}, \{\neg d, b\}\}$, in which $\{a, b, \neg c\}$ is to be deleted. The figure shows, in addition, how the `references` and `clauses` lists in Figure 5.1b are updated for the given formula. The reference for each clause C is calculated based on the sum of the sizes (in buckets) of all clauses preceding C in the list of clauses. For example, the first clause C_1 requires $12 + 4t = 24$ bytes or $\texttt{CB} + t$ buckets, where a bucket consists of four bytes, and the constant `CB` is the number of buckets needed to store `SCLAUSE`, in our case 12 bytes / 4 bytes. Given the number of buckets needed for C_1 is 6, the next clause (C_2) must be stored starting from position 6 in the list of clauses. This position plus the size of C_2 determines in a similar way the starting position for C_3, and so on.

The first step towards compacting the `CNF` instance when C_2 is to be deleted is to compute a *stencil* and a list of corresponding clause sizes in terms of numbers of buckets. In this step, each clause C_i is inspected by a different thread that writes a '0' at position *tid* of a list named `stencil` if the clause must be deleted, and a '1' otherwise. The size of `stencil` is equal to the number of clauses. In a list of the same size called `buckets`, the thread writes at position *tid* '0' if

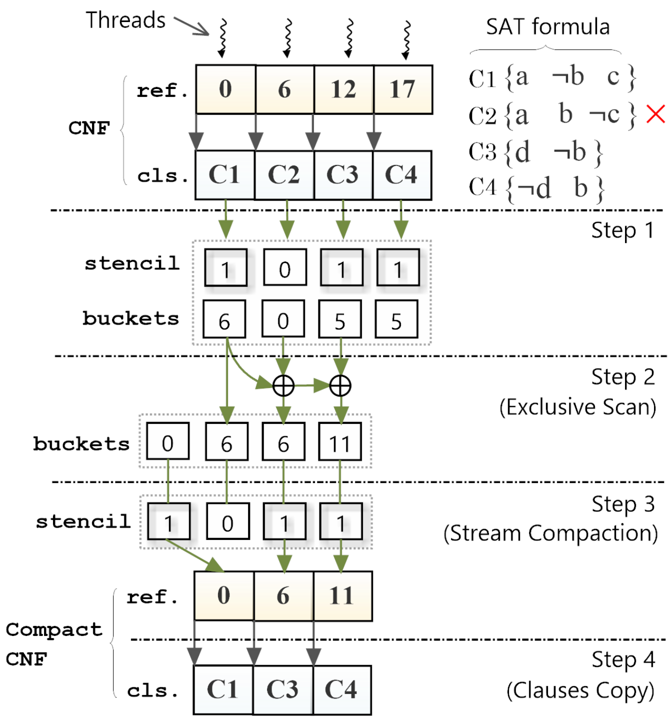


Figure 5.2: An example of parallel GC on a GPU

the clause will be deleted, and otherwise the size of the clause in terms of the number of buckets.

At step 2, a parallel *exclusive-segmented scan* operation is applied on the `buckets` array to compute the new references. In this scan, the value stored at `buckets`[*tid*], masked by the corresponding `stencil`, is the sum of the values stored at positions 0 up to, but not including, *tid*. An optimised GPU implementation of this operation is available via the CUDA CUB library [Mer20], which transforms a list of size n in $\log(n)$ iterations. In the example, this results in C_3 being assigned reference 6, thereby replacing C_2.

At step 3, the `stencil` list is used to update `references` in parallel. The `DeviceSelect::Flagged` standard function of the CUB library can be deployed for this, keeping clause references in consecutive positions via stream compaction [BOA09]. Finally, the actual clauses are copied to their new locations in `clauses`.

Algorithm 5.1 describes in detail the GPU implementation of the parallel GC. As input, Algorithm 5.1 requires a SAT formula $\mathcal{S}_{in}$ as an instance of `CNF`. The constant `CB` is kept in GPU constant memory for fast access. To begin GC, we count the number of clauses and literals in the $\mathcal{S}_{in}$ formula after simplification has been applied (line 1). The counting is done via the parallel reduction kernel COUNTSURVIVED, listed at lines 7-33.

The values *rCls* and *rLits* at line 8 will hold the current number of clauses and literals, respectively, counted by the executing thread. The value b is used as a loop counter and initially holds half of the current block size. These variables are stored in the thread-local register memory. Within the loop at lines 9-13, the counters *rCls*, *rLits* are updated incrementally if the clause at position *tid* in `clauses` is not deleted. Once a thread has checked all its assigned clauses, it stores the counter values in the (block-local) shared memory arrays (`shCls`, `shLits`) at line 16.

A non-participating thread simply writes zeros (line 18). Next, all threads in the block are synchronized by the SYNCTHREADS call. The loop at lines 21-27 performs the actual parallel reduction to accumulate the number of non-deleted clauses and literals in shared memory within thread blocks. In each iteration, the counter b is divided by 2 until it is equal to 32 (note that blocks always consist of a power of two number of threads). The last 32 threads assembling a full warp reduce the data in the shared memory via warp shuffle reduction (line 28). This operation allows all-reduce direct communication between the threads in a single warp without the need for synchronization.

The total number of clauses and literals is in the end stored by thread 0, and this thread adds those numbers using atomic instructions to the globally stored counters *numRefs* and *numLits* at lines 30-31, resulting in the final output. In the procedure described here, we prevent having each thread perform atomic instructions on the global memory, by which we avoid a potential performance bottleneck. The computed numbers are used to allocate enough memory for the output formula at line 2 on the CPU side.

The kernel COMPUTESTENCIL, called at line 3, is responsible for checking clause states and computing the number of buckets for each clause. The COMPUTESTENCIL kernel is given at lines 34-43. If a clause C is set to `DELETED` (line 37), the corresponding entries in `stencil` and `cindex` are cleared at line 38, otherwise the `stencil` entry is set to 1 and the `cindex` entry is updated with the number of clause buckets.

The EXCLUSIVESCAN routine at line 4 calculates the new references to store

Algorithm 5.1: Parallel Garbage Collection

```
Input  : global $\mathcal{S}_{in}$, stencil, cindex   shared shCls, shLits   constant CB
Output : $\mathcal{S}_{out}$
numRefs, numLits ← countSurvived($\mathcal{S}_{in}$)
$\mathcal{S}_{out}$ ← allocate(numRefs, numLits)
stencil, cindex ← computeStencil($\mathcal{S}_{in}$)
cindex ← exclusiveScan(cindex)
references($\mathcal{S}_{out}$) ← compactRefs(cindex, stencil)
copyClauses($\mathcal{S}_{out}$, $\mathcal{S}_{in}$, cindex, stencil)
kernel countSurvived($\mathcal{S}_{in}$):
    register $rCls \leftarrow 0,\ rLits \leftarrow 0, b \leftarrow blockDim/2$
    for all tid $\in [\![\, 0, |\mathcal{S}_{in}|\, ]\!]$ do in parallel
        register $C \leftarrow \mathcal{S}_{in}[tid]$
        if state($C$) $\neq$ DELETED then
            $rCls \leftarrow rCls + 1,\ rLits \leftarrow rLits + |C|$
        end
    end
    if $t_x < |\mathcal{S}_{in}|$ then
        shCls[$t_x$] = $rCls$, shLits[$t_x$] = $rLits$
    else
        shCls[$t_x$] = 0, shLits[$t_x$] = 0
    end
    syncThreads( )
    for $b : b/2 \to 32$ do                // b will be blockDim/2, (blockDim/2)/2, ..., 32
        if $t_x < b$ then
            shCls[$t_x$] ← shCls[$t_x$] + shCls[$t_x + b$]
            shLits[$t_x$] ← shLits[$t_x$] + shLits[$t_x + b$]
        end
        syncThreads( )
    end
    if $b = 32$ then shuffleReduction(shCls,shLits)
    if $t_x = 0$ then
        atomicAdd(numRefs, shCls[$t_x$])
        atomicAdd(numLits, shLits[$t_x$])
    end
end
kernel computeStencil($\mathcal{S}_{in}$):
    for all tid $\in [\![\, 0, |\mathcal{S}_{in}|\, ]\!]$ do in parallel
        register $C \leftarrow \mathcal{S}_{in}[tid]$
        if state($C$) = DELETED then
            stencil[tid] ← 0 , cindex[tid] ← 0
        else
            stencil[tid] ← 1 , cindex[tid] ← CB + $t$
        end
    end
end
kernel copyClauses ($\mathcal{S}_{out}$, $\mathcal{S}_{in}$, cindex, stencil):
    for all tid $\in [\![\, 0, |\mathcal{S}_{in}|\, ]\!]$ do in parallel
        if stencil[tid] then
            register & $C_{dest}$ ← (SCLAUSE &)(clauses($\mathcal{S}_{out}$) + cindex[tid])
            $C_{dest} \leftarrow \mathcal{S}_{in}[tid]$
        end
    end
end
```

the remaining clauses based on the collected buckets. For that, we use the exclusive scan method offered by the CUB library. The compactRefs routine called at line 5 groups the *valid* references, i.e., those flagged by `stencil`, into consecutive values and stores them in `references`($\mathcal{S}_{out}$), which refers to the `references` field of the output formula $\mathcal{S}_{out}$. Finally, copying clause contents (literals, state, etc.) is done in the copyClauses kernel, called at line 6. This kernel is described at lines 44-51. If a clause in $\mathcal{S}_{in}$ is flagged by `stencil` via thread *tid*, then a new `SCLAUSE` reference is created in `clauses`($\mathcal{S}_{out}$), which refers to the `clauses` field in $\mathcal{S}_{out}$, offset by `cindex`[*tid*].

The GC mechanism described above resulted from experimenting with several less efficient mechanisms first. In the first attempt, two atomic additions per thread were performed for each clause, one to move the non-deleted clause buckets and the other for moving the corresponding reference. However, the excessive use of atomics resulted in a performance bottleneck and produced a different simplified formula on each run, that is, the order in which the new clauses were stored depended on the outcome of the atomic instructions. The second attempt was to maintain stability by moving the GC to the host side. However, accessing unified memory on the host side results in a performance penalty, as it implicitly results in copying data to the host side. The current GPU approach is faster and always results in the same output formula because both segmented scan and stream compaction preserve the original data order.

5.3 Proof Memory Management

In this article, we adopt the binary format in generating the DRAT proof to save memory, particularly on the GPU side. Let l and $-l$ be the positive and negatives integers to represent the literals ℓ and $\neg\ell$ respectively in DIMACS format. To encode l in the binary form, it has to be mapped to an unsigned integer first using the mapping function: `map(l) := (l > 0) ?  2 * l :  -2 * l + 1`. The mapped value can then be compressed into a variable-byte sequence of 7-bit words (w_i):

$$l = \sum_{i=0}^{4} w_i \times 2^{(7\times i)} \tag{5.1}$$

In addition, every sequence has two additional bytes. The first byte acts as a prefix to express whether a lemma is added (character `'a'` or 61 in hexadecimal) or deleted (character `'d'` or 64 in hexadecimal). The last byte is zero to mark

```
  CNF Formula        DRAT Proof      Binary DRAT (HEX)

 p cnf 5 8            1  3 0         61 02 06 00
     2  3 0           1 -3 0         61 02 07 00
     2 -3 0          -1  5 0         61 03 0A 00
  1 -2  3 0          -1 -5 0         61 03 0B 00
  1 -2 -3 0              1 0            61 02 00
 -1  4  5 0             -1 0            61 03 00
 -1  4 -5 0                0               61 00
 -1 -4  5 0
 -1 -4 -5 0
```

Figure 5.3: An example showing the DRAT proof generated by ParaFROST

the end of the lemma. Ideally the binary form can save memory by a factor of 3 as reported in [WHH14].

Example 5.1. Consider the CNF formula in Figure 5.3 (leftmost side). Eliminating the variables 2 and 4 yields the resolvents $\{1, 3\}, \{1, \text{-}3\}$, and $\{\text{-}1, 5\}, \{\text{-}1, \text{-}5\}$, respectively. Finally, eliminating the variables 3 and 5 produces two unit clauses $\{1\}$ and $\{\text{-}1\}$, respectively. Thus, the formula is declared unsatisfiable due to the contradicting units. In the middle, the DRAT proof is provided by the ParaFROST solver, revealing all resolvents added after each resolution step. A binary-equivalent DRAT format is also shown on the rightmost side.

Before applying certified simplifications on the GPU, an upper-bound of memory space required to store the binary DRAT proof is calculated for all literals. The idea is to compute, per unsigned literal l, the minimum number of bytes (see Equation 5.1). To do that efficiently on the GPU, one can count the number of leading zeros (Z) in the bit string of the integer value using the intrinsic function clz. By subtracting Z from 32, we get the position of the most-significant high bit M (i.e., minimum number of bits to represent l). Dividing the latter by 7 (remember binary DRAT uses 7-bit wording), gives the lower bound of the number of words W. To get the upper bound, W needs to be rounded up using the integer division $(W + 7 - 1)/7$. However, doing this operation for each literal, particularly in large formulas incurs a performance penalty. Since the position cannot be higher than 32, a small lookup table is created for all possible values of $roundup(W)$ and stored in the constant memory. The table has a fixed length of 31 and its values range between 1 and 5.

Figure 5.4 gives a working example of the parallel computation of the upper-

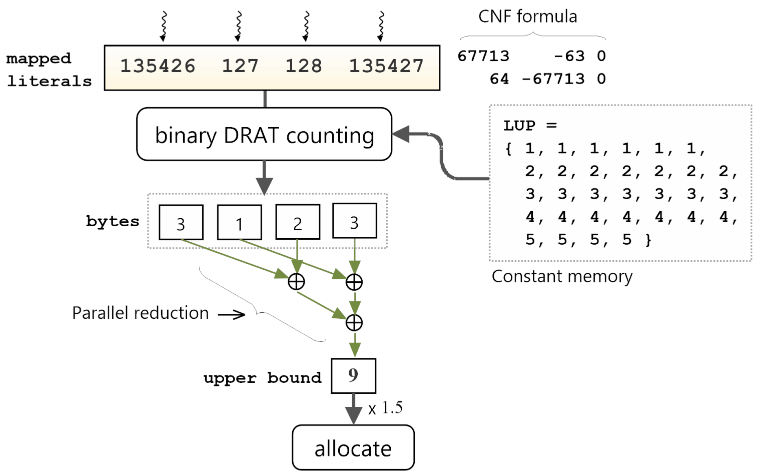


Figure 5.4: An example of binary DRAT counting on a GPU

bound for the literals 67713, −63, 64, and −67713. Initially, the literals are mapped to the unsigned integers 135426, 127, 128, and 135427, respectively. Next, each thread calculates the minimum number of bytes per literal as described above and the results are rounded up using the lookup table `LUP` stored in the constant memory. For example, 135427 would occupy a minimum of 3 bytes to store. Finally, parallel reduction is applied on the `pbytes` array to sum up the contents and obtain the upper bound (9 in this example). Ideally, we need a memory space equal to the literals upper bound plus 2 times the number of clauses in a CNF formula (recall that DRAT requires two additional bytes per clause). However, in practice, 1.5 times this bound is needed to guarantee enough space for emitting the proof on the GPU side.

Algorithm 5.2 lists in detail the GPU proof memory allocator. It takes as input the formula $\mathcal{S}$ and the lookup table `LUP`. First, all clauses are flattened into consecutive literals and stored in the array `literals`. At line 2, the kernel COUNTPROOF is launched (given at lines 4-30) to create both the `pbytes` array and the memory bound *proofbound*. The former is needed as reference to emit the proof later in VCE using atomic instructions.

The variable *rBytes* at line 5 will hold the current number of bytes counted by the executing thread. The value *b* initially holds half of the current block size (used later as a loop counter). Within the loop at lines 6-14, the counter

Algorithm 5.2: GPU Proof Memory Allocator

```
   Input  : global $\mathcal{S}_{in}$, literals, pbytes
   Input  : shared shBytes
   Input  : constant LUP
   Output : global $\mathcal{P}$
1  literals $\leftarrow$ FLATTEN($\mathcal{S}_{in}$)
2  proofbound, pbytes $\leftarrow$ COUNTPROOF(literals)
3  $\mathcal{P} \leftarrow$ ALLOCATE(proofbound)
4  kernel COUNTPROOF(literals):
5      register $rBytes \leftarrow 0, b \leftarrow blockDim/2$
6      for all tid $\in [\![ 0, |$literals$| ]\!]$ do in parallel
7          $l \leftarrow$ literals[tid]
8          if pbytes[$l$] $= 0$ then
9              pbytes[$l$] $\leftarrow$ LUP[$30 -$ CLZ($l$)]   // offset position ($32 -$ CLZ($l$)) by $-2$
10             $rBytes \leftarrow rBytes +$ pbytes[$l$]
11         else                   // the more threads taking this path, the better
12             $rBytes \leftarrow rBytes +$ pbytes[$l$]
13         end
14     end
15     if $t_x < |\mathcal{S}_{in}|$ then
16         shBytes[$t_x$] $= rBytes$
17     else
18         shBytes[$t_x$] $= 0$
19     end
20     SYNCTHREADS( )
21     for $b : b/2 \rightarrow 32$ do
22         if $t_x < b$ then
23             shBytes[$t_x$] $\leftarrow$ shBytes[$t_x$] $+$ shBytes[$t_x + b$]
24         end
25         SYNCTHREADS( )
26     end
27     if $b = 32$ then SHUFFLEREDUCTION(shBytes)
28     if $t_x = 0$ then
29         ATOMICADD(proofbound, shBytes[$t_x$])
30     end
31 end
```

rBytes is updated incrementally if the value pbytes[l] is zero, i.e., the number of bytes has not been computed before for the literal l. Notice that we subtract the bit position CLZ (l) at line 9 from 30 rather than 32 as the table is indexed from 0 to 30 (see Figure 5.4). Having a non-zero value at pbytes[l] means the current literal is a duplicate and its variadic size has already been computed before. Therefore, in this case, we rely on data racing and thread divergence, which contradicts the convention of parallel programming. Example 5.2 explains this phenomenon.

Example 5.2. Suppose we have a set of `literals` = $\{3, 3, 5, \texttt{-}5\}$ and four threads $t_0, \ldots, t_3$ where t_i represents the *tid* of thread i. In Algorithm 5.2, when t_0 and t_1, both inspect literal 3, the following two scenarios are possible:

1. Either one of the threads t_0 or t_1 is *faster* than the other and takes the control path at lines 8-10, updating `pbytes`[3] to 1. The other thread has seen the new updated value `pbytes`[3] = 1; thus, taking the control path at lines 11-12. In that case, the more threads executing that path, the better.

2. Both threads t_0 and t_1 check the condition at line 8 at the exact same time. Thus, they both do the counting and update `pbytes`[3] simultaneously. This is not problematic, as they both write the same value.

Once a thread has checked its literal, it stores the counter value in the (block-local) shared memory array `shBytes` at line 16. The values in `shBytes` are then reduced in parallel to the global variable *proofbound*. With this value, memory is allocated to the proof stream $\mathcal{P}$ in bytes at line 3.

5.4 Variable Scheduling

As previously discussed in Chapter 4, each elimination method is applied on multiple variables simultaneously. Doing so may lead to data hazards, due to the disjunction between literals in all clauses. That is, if two dependent variables x and y were to be processed for simplification, two threads might manipulate clause C containing x and y at the same time. To guarantee soundness of the parallel simplifications, we proposed our LCVE algorithm (Chapter 4) prior to simplification which is responsible for electing a set of mutually independent variables (candidates) from a set of authorised candidates. The remaining variables relying on the elected ones are frozen. We encourage the reader to check the Definitions 4.1-4.4 for more details.

In this chapter, we map the frozen variables to the domain $\{x_1, \ldots, x_{12}\}$ in addition to variable elections. For frozen variables that would be mapped to a value higher than 12, their mappings are set to 0 (i.e., a unique stamp to identify out-of-range variables). The mapped variables are used later in the next section to build the function tables (denoted by FUNTAB) with size $2^{12} = 512$ bits. Algorithm 5.3 extends the original (Algorithm 4.2) to support the $\mathcal{F}$-mapping

Algorithm 5.3: LCVE (updated version to support $\mathcal{F}$-mapping)

```
Input  : $\mathcal{S}, \mathcal{A}, h, \mathcal{T}$
Output : $\Phi$
$\mathcal{F} \leftarrow \emptyset$, $\mathcal{F}_{map} \leftarrow \{0\}$, $limit \leftarrow 1$
foreach $x \in \mathcal{A}$ do
    if $x \notin \mathcal{F}$ then
        $\Phi \leftarrow \Phi \cup x$
        foreach $C \in \mathcal{S}[\mathcal{T}[x]] \cup \mathcal{S}[\mathcal{T}[\neg x]]$ do
            foreach $\ell \in C$ do
                $v \leftarrow var(\ell)$
                if $v \neq x$ then
                    $\mathcal{F} \leftarrow \mathcal{F} \cup v$
                    if $limit \leq 12$ then
                        $\mathcal{F}_{map}[v] \leftarrow limit$, $limit \leftarrow limit + 1$
                    end
                end
            end
        end
    end
end
```

functionality. Initially, all elements in $\mathcal{F}_{map}$ are set to 0 (line 1). Afterwards, the algorithm considers all variables x in $\mathcal{A}$ (line 2). If x has not yet been frozen (line 3), it adds x to Φ (line 4). Next, the algorithm needs to identify all variables that depend on x. For this, it iterates over all clauses containing either x or $\neg x$ (line 5), and each literal ℓ in those clauses is compared to x (lines 6-8). If ℓ refers to a different variable v, then v must be frozen. In addition, we map v to a value *limit* in range $1 \leq limit \leq 12$ and stores it in $\mathcal{F}_{map}$ (lines 10-12).

5.5 Main Inprocessing Procedure

A top-level description of GPU parallel inprocessing is shown in Algorithm 5.4. As input, it takes the current formula $\mathcal{S}_h$ from the solver (executed on the host) and copies it to the device global memory as $\mathcal{S}_d$ (line 1). Initially, before simplification, we compute the clause signatures and sort clause literals via stream 0 at line 2 (PREPAREFORMULA procedure). Concurrently, via stream 1, variables are ordered at line 3. Concurrent execution is discussed at Section 3.4. The ORDERVARIABLES routine does the same operations explained earlier for Algorithm 4.1 to produce an ordered array of authorised candidates $\mathcal{A}$ following

Algorithm 5.4: Certified Parallel Inprocessing

```
Input  : $\mathcal{S}_h$, $\mu$, *phases*
$\mathcal{S}_d \leftarrow$ copyToDevice($\mathcal{S}_h$)
prepareFormula($\mathcal{S}_d$, *stream0*)
$\mathcal{A} \leftarrow$ orderVariables($\mathcal{S}_d$, *stream1*)
$\mathcal{P}_h, \mathcal{P}_d \leftarrow$ proofAllocator($\mathcal{S}_d$, *stream0*)
while $p : 0 \rightarrow phases$ do
    syncAll( )                                      // Synchronize all streams
    $\mathcal{T} \leftarrow$ createOT($\mathcal{S}_d$)
    BCP($\mathcal{U}_h, \mathcal{S}_d, \mathcal{T}$)
    $\Phi \leftarrow$ LCVE($\mathcal{S}_d, \mathcal{T}, \mathcal{A}, \mu$)
    if $p = phases$ then
        ere($\mathcal{S}_d, \mathcal{T}, \Phi$)
        break
    end
    sortOT($\mathcal{T}, \Phi$, listKey)
    $\mathcal{U}_d, \mathcal{P}_d \leftarrow$ eliminate($\mathcal{S}_d, \mathcal{T}, \Phi$)          // Applies SUB then BVE
    $\mathcal{P}_h \leftarrow$ copyToHostAsync($\mathcal{P}_d$, *stream1*)
    $\mathcal{U}_h \leftarrow$ copyToHostAsync($\mathcal{U}_d$, *stream2*)
    collect($\mathcal{S}_d$, *stream3*)
    sync(*stream1*), writeProof($\mathcal{P}_h$)
    $\mu \leftarrow \mu \times 2$
end
device function listKey($a, b$):
    $C_a \leftarrow \mathcal{S}_d[a], C_a \leftarrow \mathcal{S}_d[b]$        // $C_a = \{x_1, x_2, \ldots, x_k\}, C_b = \{y_1, y_2, \ldots, y_k\}$
    if $|C_a| \neq |C_b|$ then return $|C_a| < |C_b|$
    if $x_1 \neq y_1$ then return $x_1 < y_1$
    if $x_k \neq y_k$ then return $x_k < y_k$
    if sig($x_k$) $\neq$ sig($y_k$) then return sig($C_a$) $<$ sig($C_b$)
    return $a < b$
end
```

Definition 4.1. At line 4, Algorithm 5.2 is executed via the proofAllocator routine on the same stream as prepareFormula. The space allocated for $\mathcal{P}_d$ resides in global memory; whilst $\mathcal{P}_h$ gets a pinned memory space (see Section 3.3.2) on the host side with the same size as $\mathcal{P}_d$.

The **while** loop at lines 5-21 applies SUB and BVE, for a configured number of iterations (indicated by *phases*), with increasingly large values of the threshold μ. Increasing μ exponentially allows LCVE to elect additional variables in the next elimination phase since after a phase is executed on the GPU, many elected variables are eliminated. In addition, mapping a new set of frozen variables is essential for the effectiveness of funTab in finding new gate definitions. The ERE method is computationally expensive. Therefore, it is only executed once in the final iteration, at line 11. At line 6, syncAll is called to synchronize all streams being executed. At line 7, the occurrence table $\mathcal{T}$ is created. Next, the LCVE routine produces the set $\varPhi$ (see Definition 4.3) as explained previously in Algorithm 5.3.

The parallel creation of the occurrence lists in $\mathcal{T}$ results in the order of these lists being chosen non-deterministically. Directly applying the eliminate procedure called at line 15, which performs the parallel simplifications, would produce results non-deterministically as well. To remedy this effect, the lists in $\mathcal{T}$ are sorted according to a unique key in ascending order before eliminate is called. Besides the benefit of stability, this allows SUB to abort early when performing subsumption checks.

The sorting key is given as the device function listKey at lines 22-29. It takes two references a, b and fetches the corresponding clauses C_a, C_b from $\mathcal{S}_d$ (line 23). First, clause sizes are tested at line 24. If they are equal, the first and the last literal in each clause are checked, respectively, at lines 25-26. If the literals are equal, clause signatures are tested at line 27. Otherwise, clause references are compared at line 28. The references are always distinct; thus, they guarantee sorting stability. However, they should be tested as a last resort. Experiments have shown that using only clause reference in addition to its size has a negative impact overall on the CDCL search. CaDiCaL implements a similar function, but only considers clause sizes [BFFH20]. The sortOT routine launches a kernel to sort the lists pointed to by the variables in $\varPhi$ in parallel. Each thread runs an insertion sort to in-place swap clause references using listKey.

The eliminate procedure at line 15 applies the SUB method to remove any subsumed clauses or strengthen clauses if possible, after which BVE is applied. The SUB and BVE methods call kernels that scan the occurrence lists of all variables in $\varPhi$ in parallel. More information on this is in Sections 5.6 and 5.7. Both the BVE and SUB methods emit the proof to $\mathcal{P}_d$ and may add new unit clauses atomically to a separate array $\mathcal{U}_d$. The propagation of these units cannot be done immediately on the GPU due to possible data races, as multiple variables in a clause may occur in unit clauses. For instance, if we have unit clauses $\{a\}$ and $\{b\}$, and these would be processed by different threads, then a clause $\{\bar{a}, \bar{b}, c\}$ could be updated by both threads simultaneously. Thus, this propagation is delayed until the next iteration, and performed by the host at line 8. Note that $\mathcal{T}$ must be recreated first to consider all resolvents added by BVE during the previous phase. The ERE method at line 11 is executed only once at the last phase (*phases*) before the loop is terminated. Section 5.8 explains in detail how ERE can be effective in simplifying both `ORIGINAL` and `LEARNT` clauses in parallel. Again, clausal proof of ERE correctness is emitted to $\mathcal{P}_d$.

At line 16-17, the proof stream and new units are copied from the device to the host arrays $\mathcal{P}_h$ and $\mathcal{U}_h$, respectively. The data transfers are done asynchronously via *stream1* and *stream2*. Similar to $\mathcal{P}_h$, the $\mathcal{U}_h$ array is allocated in pinned host memory. Recall from Section 3.3.2 that asynchronous data transfers to the host are only allowed if the host memory is page-locked. The COLLECT procedure does the GC as described by Algorithm 5.1 via *stream3*. At line 19, we synchronize the proof data transfer performed by *stream1* and write the byte stream to the proof output file. Other active streams are synchronized at line 6.

5.6 Three-Phase Parallel Variable Elimination

The BVIPE algorithm in Chapter 4 had a main shortcoming due to the heavy use of atomic operations to add new resolvents. Per eliminated variable, two atomic instructions were performed, one for adding new clauses and the other for adding new literals. Besides performance degradation, this also resulted in the order of added clauses being chosen non-deterministically, which impacted reproducibility (even though the produced formula would always at least be logically the same).

The approach to avoiding the excessive use of atomic instructions when adding new resolvents is to perform parallel BVE in *three phases*. The first phase scans the constructed list Φ to identify the elimination type (e.g., resolution or gate substitution) of each variable and to calculate the number of resolvents and their corresponding buckets.

The second phase computes an exclusive scan to determine the new references for adding resolvents, as is done in our GC mechanism (Section 5.2). At the last phase, we store the actual resolvents in their new locations in the simplified formula. For solution reconstruction, we use an atomic addition to count the resolved literals. The order in which they are resolved is irrelevant. The same is done for emitting the proof and adding units. For the latter, experiments show that the number of added units and proof bytes is relatively small compared to the number of eliminated variables[12], hence the penalty for using atomic instructions is almost negligible. It would be overkill to use a segmented scan for adding proof bytes or units.

[12]Deleted clauses in BVE are not added to the proof in order to save GPU memory.

Algorithm 5.5: Certified 3-Phase Parallel BVE with funTab

```
   Input  : global $\Phi$, $\mathcal{T}$, $\mathcal{U}_d$, $\mathcal{P}_d$, $\mathcal{S}_d$, litstack, varinfo, cindex, rindex, pbytes
   Input  : constant CB
 1 varinfo ← VceScan($\Phi, \mathcal{S}_d, \mathcal{T}$)
 2 cindex ← computeClauseIndices(varinfo, size(clauses))
 3 rindex ← computeClauseRefIndices(varinfo, size(references))
 4 VceApply($\Phi, \mathcal{S}_d, \mathcal{T}$, varinfo, cindex, rindex)
 5 kernel VceScan($\Phi, \mathcal{S}_d, \mathcal{T}$):
 6     for all $tid \in [\![\, 0, |\Phi| \,]\!]$ do in parallel
 7         register $x \leftarrow \Phi[tid], \mathcal{T}_x = \mathcal{T}[x], \mathcal{T}_{\neg x} = \mathcal{T}[\neg x]$
 8         register $t \leftarrow$ NONE, $rCls \leftarrow 0, rLits \leftarrow 0$
 9         varinfo$[tid] \leftarrow 0$, cindex$[tid] \leftarrow 0$, rindex$[tid] \leftarrow 0$
10         if $\mathcal{T}_x = \emptyset \vee \mathcal{T}_{\neg x} = \emptyset$ then litstack ← toblivion($x, \mathcal{S}_d, \mathcal{T}_x, \mathcal{T}_{\neg x}$)
11         else
12             $t, rCls, rLits \leftarrow$ gateReasoning($x, \mathcal{S}_d, \mathcal{T}_x, \mathcal{T}_{\neg x}$)
13             if $t =$ NONE then $t, rCls, rLits \leftarrow$ funReasoning($x, \mathcal{S}_d, \mathcal{T}_x, \mathcal{T}_{\neg x}$)
14             if $t =$ NONE then $t, rCls, rLits \leftarrow$ mayResolve($x, \mathcal{S}_d, \mathcal{T}_x, \mathcal{T}_{\neg x}$)
15             varinfo$[tid] \leftarrow t$, rindex$[tid] \leftarrow rCls$, cindex$[tid] \leftarrow$ CB $\times\, rCls + rLits$
16         end
17     end
18 end
19 kernel VceApply($\Phi, \mathcal{S}_d, \mathcal{T}$, varinfo, cindex, rindex):
20     for all $tid \in [\![\, 0, |\Phi| \,]\!]$ do in parallel
21         register $x \leftarrow \Phi[tid]$
22         register $t \leftarrow$ varinfo$[tid]$, $cidx \leftarrow$ cindex$[tid]$, $ridx =$ rindex$[tid]$
23         $reqSpace \leftarrow cidx +$ CB $\times\, rCls + rLits$
24         $reqRefs \leftarrow ridx + rCls$
25         if $t \neq$ NONE then
26             if $t =$ RES then
27                 $(\mathcal{S}_d, \mathcal{U}_d, \mathcal{P}_d) \leftarrow (\mathcal{S}_d, \mathcal{U}_d, \mathcal{P}_d) \cup$ resolve($x, \mathcal{S}_d, \mathcal{T}, ridx, cidx$, pbytes)
28             else if $t =$ SUBST then
29                 $(\mathcal{S}_d, \mathcal{U}_d, \mathcal{P}_d) \leftarrow (\mathcal{S}_d, \mathcal{U}_d, \mathcal{P}_d) \cup$ substitute($x, \mathcal{S}_d, \mathcal{T}, ridx, cidx$, pbytes)
30             else if $t =$ CORE then
31                 $(\mathcal{S}_d, \mathcal{U}_d, \mathcal{P}_d) \leftarrow (\mathcal{S}_d, \mathcal{U}_d, \mathcal{P}_d) \cup$
                     coreSubstitute($x, \mathcal{S}_d, \mathcal{T}, ridx, cidx$, pbytes)
32             end
33             litstack ← toblivion($x, \mathcal{S}_d, \mathcal{T}[x], \mathcal{T}[\neg x]$)
34         end
35     end
36 end
```

At line 1 of Algorithm 5.5, phase 1 is executed by the VceScan kernel (given at lines 5-18). Every thread scans the clause set of its designated literals x and $\neg x$ (line 7). References to these clauses are stored at $\mathcal{T}_x$ and $\mathcal{T}_{\neg x}$. Moreover, register variables t, $rCls$, $rLits$ are created to hold the current *type*, number of *added clauses*, and number of *added literals* of x, respectively (line 8). If x is *pure* at line 10, then there are no resolvents to add and the clause sets of x and $\neg x$ are directly marked as `DELETED` by the routine toblivion. Moreover, this routine adds the marked literals atomically to `litstack`. Note that these clauses are

Algorithm 5.6: funTab Reasoning

```
   Input    : global $\mathcal{S}_d$, $\mathcal{T}$
1  device function funReasoning($x, \mathcal{S}_d, \mathcal{T}_x, \mathcal{T}_{\neg x}$):
2      local $f_p \leftarrow \{1, \ldots, 1\}, f_n \leftarrow \{1, \ldots, 1\}$
3      $withinRange \leftarrow$ buildFunTab($x, \mathcal{S}_d, \mathcal{T}_x, f_p$) $\wedge$ buildFunTab($\neg x, \mathcal{S}_d, \mathcal{T}_{\neg x}, f_n$)
4      if $withinRange \wedge$ and($f_p$, $f_n$) $= \{0, \ldots, 0\}$ then
5          $G_\ell \leftarrow \mathcal{S}_d[\mathcal{T}[x]] \cup \mathcal{S}_d[\mathcal{T}[\neg x]]$
6          forall $C \in G_\ell$ do
7              if flag($C$) $= 0 \wedge$ isFalseFun($G_\ell \setminus \{C\}$) then  flag($C$) $\leftarrow 1$
8          end
9      end
10 end
11 device function buildFunTab($\ell, \mathcal{S}_d, \mathcal{T}_\ell, f$):
12     forall $C \in \mathcal{S}_d[\mathcal{T}_\ell]$ do
13         shared $f_s \leftarrow \{0\}$
14         forall $\ell' \in C$ do
15             if $\ell' \neq \ell$ then
16                 $v \leftarrow \mathcal{F}_{map}[var(\ell')]$
17                 if $v = 0$ then  return false
18                 $f_s \leftarrow$ or($f_s$, magicNum($v$))
19             end
20         end
21         $f \leftarrow$ and($f$, $f_s$)
22     end
23 end
```

not emitted to the proof. At line 12, we check first if x contributes to a regular logical gate using the routine gateReasoning, and save the corresponding *rCls* and *rLits*. If this is the case, the type t is set to `SUBST`, otherwise we try funTab reasoning at line 13 or resolution at line 14.

The funReasoning procedure is given in Algorithm 5.6 at lines 1-10 and is responsible for finding irregular gate definitions as explained earlier in the introduction. At line 2, two function tables f_p, f_n are created in threads **local** memory (see Section 3.3.2). Both tables are bit-vectors of length 512 bits initialized to ones. The maximum number of variables supported is 12. In the implementation, frozen variables in $\mathcal{F}$ are mapped to values in the domain $[1, 12]$. At line 3, we encode the clause sets in $\mathcal{T}_x$ and $\mathcal{T}_{\neg x}$ into their bit representations in f_p and f_n via buildFunTab, respectively. If all literals are successfully mapped to the above range, then *withinRange* is set to **true**. A gate is found, in case both tables are built and their bit-wise and is all-zeros (i.e., unsatisfiable). Next, it is attempted to reduce the clause set G_ℓ to a smaller clausal core (not necessarily minimal, though). The loop at lines 6-8 removes a clause at a time from G_ℓ and tests for all-zero bit string via isFalseFun. If $G_\ell \setminus C$ is unsatisfiable, C is marked as a non-gate clause. After the loop terminates, all non-flagged clauses in G_ℓ together form a clausal core.

The loops at lines 12-22 in the BUILDFUNTAB function transform only the frozen literals (i.e., ℓ is skipped) in each clause $C \in \mathcal{S}_d[\mathcal{T}_\ell]$ to bit-vectors using magical numbers[13]. A magical number is unique constant in which a sequence of bits is repeating itself multiple times. For example, 4,278,255,360 is a magical number. These numbers can be used to extract and pack integer values into a single bit string (e.g., the function table). At line 16, the frozen variable $var(\ell')$ is mapped to v. If it has the value 0 (line 17), then we bail out immediately with a **false**; otherwise, the literals of C are disjoined using a bit-wise OR and stored in f_s (line 18). At line 21, the shared f_s is conjoined with the global f using a bit-wise AND.

Back to Algorithm 5.5, the condition $rCls \leq (|\mathcal{T}_x| + |\mathcal{T}_{\neg x}|)$ is always tested implicitly by the routines GATEREASONING, FUNREASONING, and MAYRESOLVE (lines 12-14) to limit the number of resolvents per x. The `varinfo`, `rindex`, and `cindex` arrays are updated at line 15. The total number of buckets needed to store all added clauses is calculated by the formula ($\texttt{CB} \times rCls + rLits$) and stored in `cindex`[$tid$]. Finally, in phase 3, we use the calculated indices in `rindex` and `cindex` to guide the new resolvents to their locations in $\mathcal{S}_d$. The kernel is described at lines 19-36. Each thread either calls the procedure RESOLVE or SUBSTITUTE or CORESUBSTITUTE, based on the type stored for the designated variables. However, a condition for applying an elimination is that $t \neq \texttt{NONE}$, which is checked using line 25. Any produced units are saved into $\mathcal{U}_d$ atomically. The $cidx$ and $ridx$ variables indicate where resolvents should be stored in $\mathcal{S}_d$ per variable x. Similarly, these resolvents are saved in $\mathcal{P}_d$ as stream bytes using the transformation in Equation 5.1. Recall that the number of bytes per literal is already stored in `pbytes` and is not required to be computed again.

5.7 Parallel Subsumption Elimination Revisited

The PSUB algorithm in Chapter 4 is revisited in this section to address the following updates:

- The learnt clauses added during the solving process.
- New unit clauses that may arise from self-subsumption resolution.
- Emitting deleted and strengthened clauses to the proof.

[13] https://graphics.stanford.edu/~seander/bithacks.html

Algorithm 5.7: Certified Parallel SUB

```
   Input    : global $\mathcal{S}_d, \Phi, \mathcal{T}, \mathcal{U}_d, \mathcal{P}_d$
1  kernel PSUB($\Phi, \mathcal{S}_d, \mathcal{T}, \mathcal{U}_d, \mathcal{P}_d$):
2      for all $tid \in [\![ 0, |\Phi| ]\!]$ do in parallel
3          register $x \leftarrow \Phi[tid], \mathcal{T}_x = \mathcal{T}[x], \mathcal{T}_{\neg x} = \mathcal{T}[\neg x]$
4          foreach $C \in \mathcal{S}_d[\mathcal{T}_x]$ do
5              shared $C_s \leftarrow C$
6              foreach $C' \in \mathcal{S}_d[\mathcal{T}_{\neg x}]$ do
7                  if $|C'| \leq |C| \wedge \textsc{sig}(C', C) \neq 0 \ \wedge\ \textsc{selfsub}(C', C_s)$ then
8                      $\textsc{strengthen}(C, C_s, x)$
9                      if $|C_s| = 1$ then $\mathcal{U}_d \leftarrow \mathcal{U}_d \cup C_s$
10                     $\textsc{proofAddClause}(\mathcal{P}_d,\ C_s)$
11                 end
12             end
13             foreach $C'' \in \mathcal{S}_d[\mathcal{T}_x]$ do
14                 if $|C''| \leq |C| \wedge \textsc{sig}(C'', C) \neq 0 \ \wedge\ \textsc{subsume}(C'', C_s)$ then
15                     if $\textsc{state}(C'') = \texttt{LEARNT} \wedge \textsc{state}(C) = \texttt{ORIGINAL}$ then
16                         $\textsc{state}(C'') \leftarrow \texttt{ORIGINAL}$
17                     end
18                     $\textsc{state}(C) \leftarrow \texttt{DELETED}$
19                     $\textsc{proofDelClause}(\mathcal{P}_d,\ C_s)$
20                 end
21             end
22         end
23     end
24 end
```

Similar to Algorithm 4.4, Algorithm 5.7 is executed on elected variables before variable elimination. This time, (Self)-subsumption resolution may reduce the number of occurrences of these variables by removing many literals through strengthening or producing unit clauses. Recall that the parallelism in PSUB is achieved by assigning each thread to a variable x and performing subsumption checks on all clauses in E_x. At line 5, a new clause is loaded, referenced by $\mathcal{T}[x]$, into shared memory C_s for faster access.

The shared clause is compared in the loop at lines 6-12 to all clauses referenced by $\mathcal{T}[\neg x]$ to check whether x is a self-subsuming literal. If so, both the original clause C, which resides in the global memory, and C_s must be strengthened (via the STRENGTHEN function). If C_s is a unit clause, it is added atomically to $\mathcal{U}_d$ (line 9). At line 10, we write this clause to the proof as an added lemma. Later on, the propagation of all unit clauses stored in $\mathcal{U}_d$ is performed sequentially on the host side after the current elimination phase is done (line 8 in Algorithm 5.4).

Algorithm 5.8: Certified Parallel ERE for Inprocessing

```
Input    : global $\Phi$, $\mathcal{S}_d$, $\mathcal{T}$, $\mathcal{P}_d$
kernel PERE($\Phi, \mathcal{S}_d, \mathcal{T}, \mathcal{P}_d$):
    for all $tid_y \in [\![ 0, |\Phi| ]\!]^y$ do in parallel
        register $x \leftarrow \Phi[tid_y], \mathcal{T}_x = \mathcal{T}[x], \mathcal{T}_{\neg x} = \mathcal{T}[\neg x]$
        for $C \in \mathcal{S}_d[\mathcal{T}_x]$ do
            for $C' \in \mathcal{S}_d[\mathcal{T}_{\neg x}]$ do
                if $(C_m \leftarrow$ RESOLVE$(x, C, C')) \neq \emptyset$ then
                    if STATE$(C)$ = LEARNT $\vee$ STATE$(C')$ = LEARNT then
                        $st \leftarrow$ LEARNT
                    else
                        $st \leftarrow$ ORIGINAL
                    end
                    FORWARDEQUALITY$(C_m, \mathcal{S}_d, \mathcal{T}, st)$
                end
            end
        end
    end
end
device function FORWARDEQUALITY($C_m, \mathcal{S}_d, \mathcal{T}, st$):
    $\mathcal{T}_{min} \leftarrow$ FINDMINLIST$(\mathcal{T}, C_m)$
    for all $tid_x \in [\![ 0, |\mathcal{T}_{min}| ]\!]^x$ do in parallel
        $C \leftarrow \mathcal{S}_d[\mathcal{T}_{min}[tid_x]]$
        if $C = C_m \wedge$ (STATE$(C)$ = LEARNT $\vee$ STATE$(C) = st$) then
            STATE$(C) \leftarrow$ DELETED
            PROOFDELCLAUSE$(\mathcal{P}_d, C)$
        end
    end
end
```

Subsequently, the strengthened C_s is used for subsumption checking in the loop at lines 13-21. In case the subsuming clause C'' is `LEARNT` and C is `ORIGINAL`, C'' must be turned to `ORIGINAL` (see Section 2.3.2). This time, the subsumed clause is written to the proof as a deleted lemma.

5.8 Eager Redundancy Elimination

Algorithm 5.8 describes a *two-dimensional* kernel similar to Algorithm 4.6, in which from each thread ID, an x and y coordinate is derived. However, the sequential loop at line 9 in Algorithm 4.6 is optimised out by finding the literal with the shortest occurrence list. This significantly minimizes the time spent on the search for redundant clauses.

Based on the kernel's $\boldsymbol{y}$ ID (line 2), each thread merges where possible two clauses of its designated variable x and its complement $\neg x$ (lines 3-6), and writes the result in shared memory as C_m. This new clause is produced by the routine resolve at line 6. At lines 7-11, we check if one of the resolved clauses is `LEARNT`, and if so, the state *st* of C_m is set to `LEARNT` as well, otherwise it is set to `ORIGINAL`. This state of C_m will guide the forwardEquality routine called at line 12 to search for redundant clauses of the same type. In this function (lines 18-27), the thread's $\boldsymbol{x}$ ID is used to search the clauses referenced by the shortest occurrence list $\mathcal{T}_{min}$, which is produced by findMinList at line 19. It has the minimum size among the lists of all literals in C_m. If a clause C is found that is equal to C_m and is either `LEARNT` or has a state equal to the one of C_m, it is set to `DELETED` (lines 23). Finally, at line 24, the deleted clause is emitted to the binary proof $\mathcal{P}_d$.

The worst-case parallel complexity of this algorithm is

$$\mathcal{O}\Big(\frac{|\Phi|}{p_{\boldsymbol{y}}}\big(\underbrace{\mu^2}_{\text{loops at lines 4-5}}\ (\overbrace{|C_m|}^{\text{findMinList at line 19}}(\frac{|\mathcal{T}_{min}|}{p_{\boldsymbol{x}}}))\big)\Big)$$

The value $|\mathcal{T}_{min}|$ is usually very small compared to the upper bound μ. Therefore, the former length can be neglected w.r.t. $p_{\boldsymbol{x}}$, and the parallel complexity in such case will be $\mathcal{O}(\mu^2|C_m|)$, that is, the parallel running time is dropped from cubic to quadratic order of magnitude w.r.t. μ.

5.9 Kernel Automated Tuning

A GPU kernel needs to be configured prior to launch. The kernel configuration sets up the number of threads per block (*blockDim*) and the total number of blocks per grid (*gridDim*). These parameters are calculated intuitively based on the data size. As a rule of thumb, the total number of threads (*blockDim* $\times$ *gridDim*) should cover the data given to the kernel to process and not to exceed the limit (p) supported by the GPU. If the data size is larger than p, a good practice is to use the grid-stride loops as discussed in Section 3.3.1. However, the GPU occupancy is crucial to achieve a near-optimal balance of the kernel workload across the GPU resources. The occupancy defines how many SMs are busy (occupied) by

Algorithm 5.9: Kernel Auto Tuner

```
Input  : $N$, $p$, $SMs$, $initBlockDim$, $minBlockDim$, $minOccupancy$
Output : $blockDim$, $gridDim$
1 $blockDim \leftarrow initBlockDim$
2 $gridDim \leftarrow \textsc{ceil}(N\ /\ blockDim)$
3 $maxBlocks \leftarrow p\ /\ initBlockDim$
4 $minBlocks \leftarrow maxBlocks \times minOccupancy$
5 while $blockDim > minBlockDim \wedge gridDim \leq minBlocks$ do
6     $blockDim \leftarrow blockDim\ /\ 2$
7     $gridDim \leftarrow \textsc{ceil}(N\ /\ blockDim)$
8 end
9 $gridDim \leftarrow \textsc{min}(gridDim, maxBlocks)$
```

the thread blocks. That is, a good occupancy means a fair distribution of the launched blocks across the available SMs.

Example 5.3. Consider an array of 1,000 data elements to be read in parallel on a GPU having 64 SMs. If we choose the block size to be 256 threads, then we need at least $\textsc{ceil}(1,000/256) = 4$ blocks to process all the data in parallel. The occupancy in this case is 4 blocks/64 SMs = 0.0625 or 6.25% which is quite low. However, if we choose a block size of 16 threads, the occupancy goes significantly up to 98.4% (63 blocks/64 SMs = 0.984).

Algorithm 5.9 illustrates a way to automate the tuning of the kernel configuration for maximum occupancy. It takes as input: the data size N, maximum number of supported threads p, number of SMs, and the initial block size *initBlockDim*. The minimum block size *minBlockDim* and the minimum occupancy *minOccupancy* are user-defined lower boundaries. Initially, the *blockDim* value is set to *initBlockDim* at line 1. Next, *gridDim* is calculated based on the latter and the data size. This step gives the initial configuration without tuning. At line 3, the maximum number of blocks that can be launched at once is computed based on the initial block size. Given this value, and the minimum occupancy desired, *minBlocks* is obtained at line 4. The loop at lines 5-8 is triggered iff *blockDim* has not gone lower than the boundary *minBlockDim* and the *gridDim* has not reached the limit *minBlocks*. The goal of this loop is to keep cutting down the *blockDim* by 2 until a maximum value of *gridDim* (within bounds) is reached. There is still a possibility that *gridDim* grows beyond *minBlocks*; therefore, always the minimum of the most recent value of *gridDim* and *maxBlocks* is targeted (line 9). In this work, we have set the *minBlockDim* and *minOccupancy* to 4 and 0.5, respectively.

We observed that the number of scheduled variables $|\Phi|$ in the preceding kernels VceScan, VceApply, PSUB, and PERE usually goes down as the solver progresses due to the elimination of many variables in the preceding call of the inprocessing procedure. Therefore, we applied the tuner in Algorithm 5.9 on the previous kernels to maximally increase the number of blocks that are scheduled for execution on the available SMs. Turning the auto tuning on in ParaFROST, led to an overall reduction in the running time of the inprocessing procedure by 2%.

5.10 Benchmarks and Analysis

We implemented the proposed algorithms in our solver ParaFROST[14] with CUDA C++ version 11.0 [NVI20a]. We evaluated all GPU experiments on the compute nodes of the Lisa GPU cluster[15]. Each problem was analysed in isolation on a separate computing node, with a time-out of 3,600 seconds. Each computing node has an Intel Xeon Gold 6130 CPU running at a base clock speed of 2.1 GHz with 96 GB of system memory, is equipped with an NVIDIA Titan RTX GPU (see Table 3.2), and runs on the Debian Linux operating system.

In the experiments, besides the implementations of our new GPU algorithms, we involved a CPU-only version of ParaFROST (called ParaFROST (noGPU)) for the solving of problems. Additionally, we compare to the CaDiCaL and Kissat [BFFH20] solvers developed by Armin Biere. The sequential solvers were executed on the compute nodes of the DAS-5 cluster [BEdL+16]. Each node has an Intel Xeon E5-2630 CPU (2.4 GHz) with 64 GB of memory. The proofs generated by the GPU solver were verified separately by the drat-trim tool [WHH14] on the DAS-5 cluster, with a timeout of 20,000 seconds. Note that the CPU on DAS-5 is 15% faster than the CPUs on the Lisa cluster.

To find benchmarks with potential for simplifications on the GPU (i.e. having sufficient amount of redundant variables and clauses), we have selected all formulas that are larger than 5 MB from the 2013-2021 SAT competitions, excluding redundancies (repeated formulas across competitions). That is, a total of 641 distinct formulas were selected which encode around 80+ different real-world applications, with various logical properties. For reproducibility, the benchmark suite can be downloaded from [Osa21a].

[14]Latest version: `https://gears.win.tue.nl/tools/parafrost-v3.0.zip`.

[15]This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

With this information, we adhere to four out of five principles laid out in the SAT manifesto (version 1) [**?**]:

1. Benchmarks should be available for research purposes.
2. Solvers should be available in binary form for research purposes.
3. A recent generic benchmark set (e.g. competition benchmarks) should be chosen among those of the last 3 years.
4. Experimental results should include a comparison with the state of the art.
5. Details on the experimental conditions should be provided (e.g. hardware, OS).

As for the third principle, we refrained from using a single benchmarks set from a particular year, as most of the included benchmarks are very small in size for the GPU to work with (i.e. only few variables and clauses can be removed).

5.10.1 SAT-Simplification Speedup

The first part of our experiments discusses the speedup obtained by the GPU algorithms for applying GC, BVE, funTab, and proof generation compared to their previous implementations in SIGmA [OW19a, OW19b] or sequential counterparts in ParaFROST (noGPU). For these experiments, we set μ and *phases* initially to 32 and 5, respectively. Preprocessing is only enabled to measure the speedup. Figure 5.5 gives the speedup of running parallel GC against a sequential version on the host. For almost all cases, Algorithm 5.1 achieved a high acceleration when executed on the device with a maximum speed up of 72.6× and an average of 35×. Figure 5.6 reveals how fast the 3-phase parallel BVE is compared to a version using more atomic instructions. On average, the new algorithm is twice as fast as the old BVIPE algorithm in Chapter 4. In addition, we get reproducible results. Figure 5.7 evaluates the funTab method in Algorithm 5.6 against the sequential counterpart. Cases with zero runtime are ignored. Clearly, the GPU achieved a remarkable speedup in finding general gate definitions compared to the CPU with a maximum of 342× and an average of 11.33×. Likewise, the acceleration of proof generation on the GPU as presented in Figure 5.8 is significant, with a maximum speedup of 186× and 12× on average.

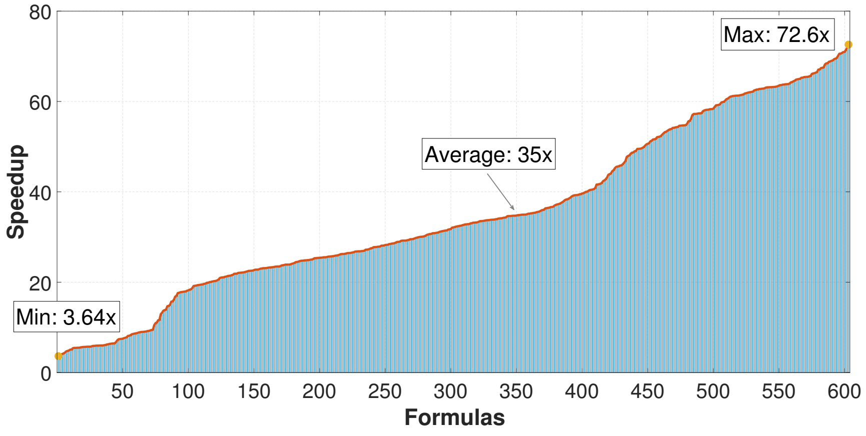


Figure 5.5: Parallel GC vs. sequential speedup

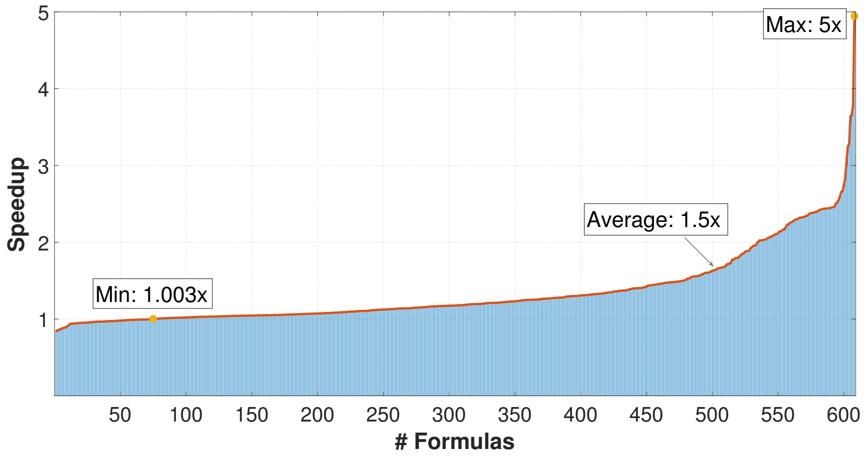


Figure 5.6: Three-Phase BVE vs. atomic version speedup

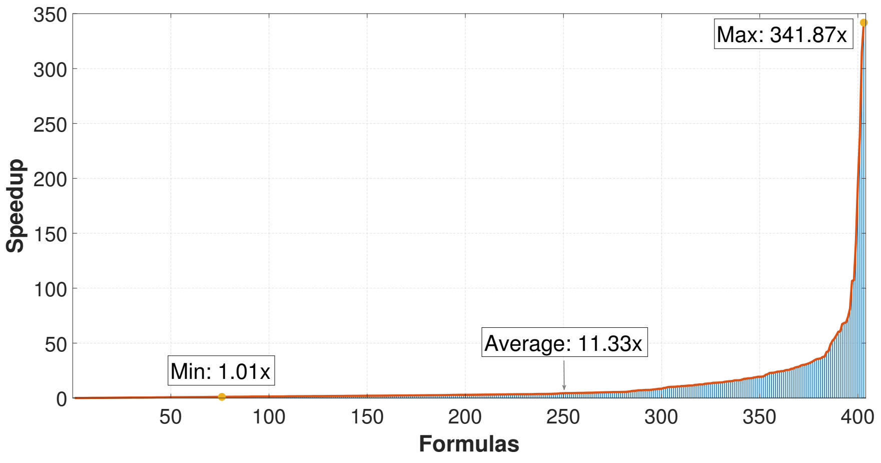


Figure 5.7: Parallel funTab vs. sequential speedup

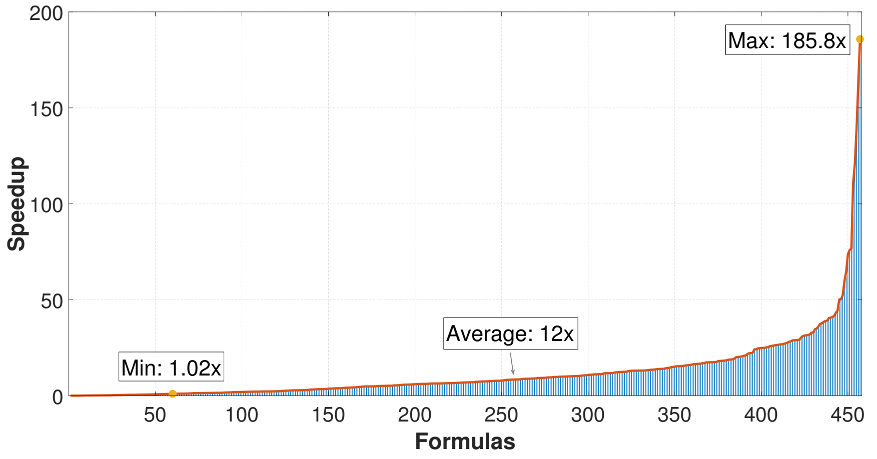


Figure 5.8: Parallel proof generation vs. sequential speedup

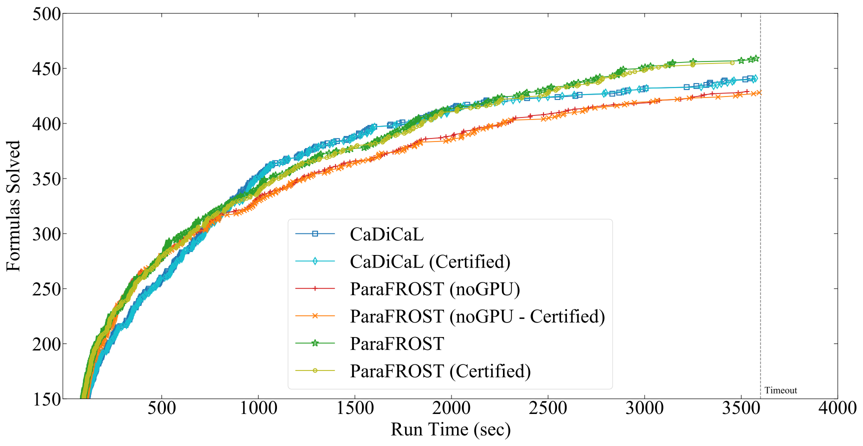


Figure 5.9: PARAFROST vs. CADICAL with(out) proof emission

5.10.2 SAT-Solving Benchmarks

The second part of experiments provides a thorough assessment of our CPU/GPU solver, the CPU-only version, CADICAL, and KISSAT on SAT solving with inprocessing turned on. The features/options in all solvers are left untouched. However, unlike CADICAL and KISSAT solvers, preprocessing is enabled by default in PARAFROST. The timeout is set to 3,600 seconds for all experiments.

Figure 5.9 demonstrates the runtime results for all solvers with(out) proof emission over the benchmark suite. The keyword *certified* means the proof generation is enabled in the solver instance. Data are sorted w.r.t. the x-axis. The simplification time accounts for data transfers in PARAFROST. Overall, PARAFROST (even with proof enabled) dominates over PARAFROST (NOGPU) and CADICAL. Keep in mind that PARAFROST and PARAFROST (NOGPU) has the same CDCL engine. Thus, PARAFROST is faster due to the GPU accelerated inprocessing.

Figure 5.10 compares PARAFROST to KISSAT with(out) irregular gate reasoning[16]. As indicated by the green and the blue lines, finding such gates has a noticeable impact on both PARAFROST and KISSAT more than PARAFROST (NOGPU). Recall that the parallel FUNTAB had a considerable speedup com-

[16]CADICAL solver does not have this feature

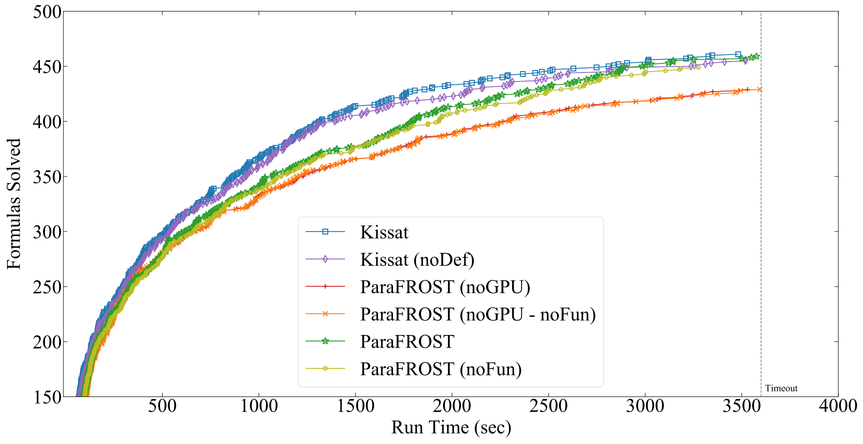


Figure 5.10: ParaFROST vs. Kissat with(out) irregular-gate reasoning

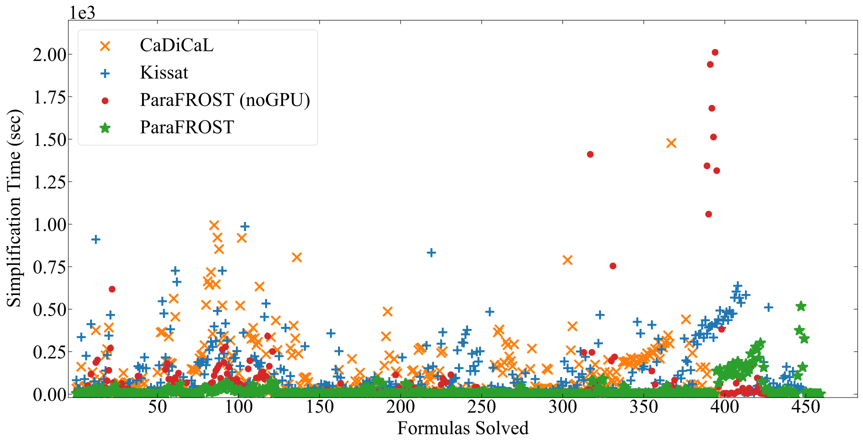


Figure 5.11: Time spent on simplifications

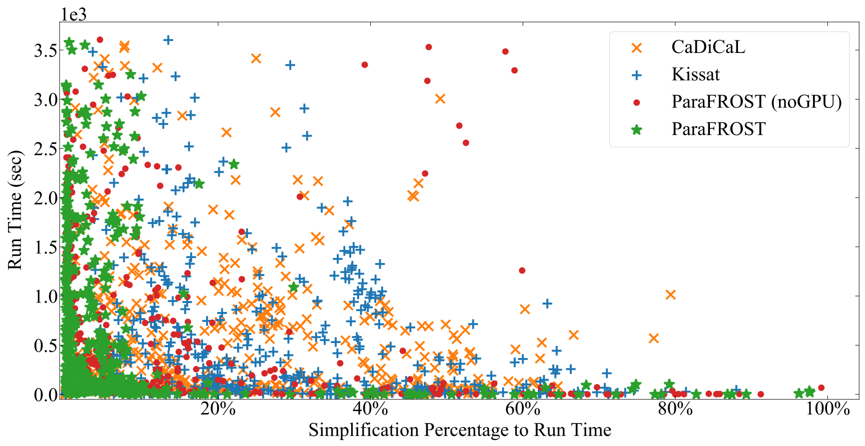


Figure 5.12: Percentage of simplification time to runtime

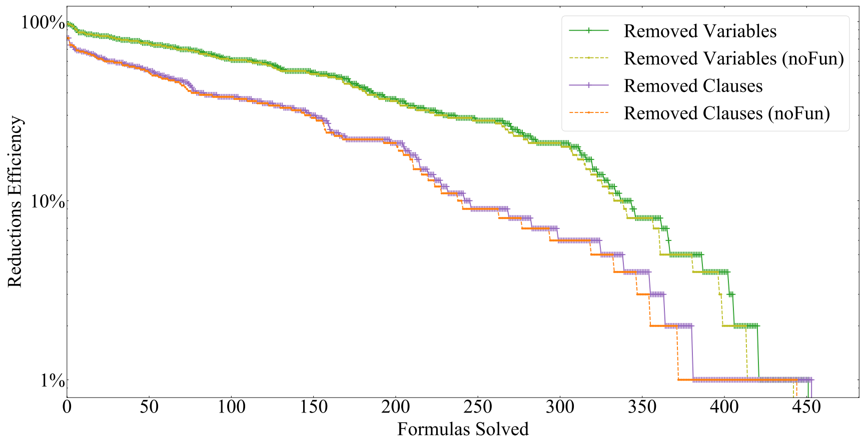


Figure 5.13: Reduction efficiency with(out) irregular-gate reasoning

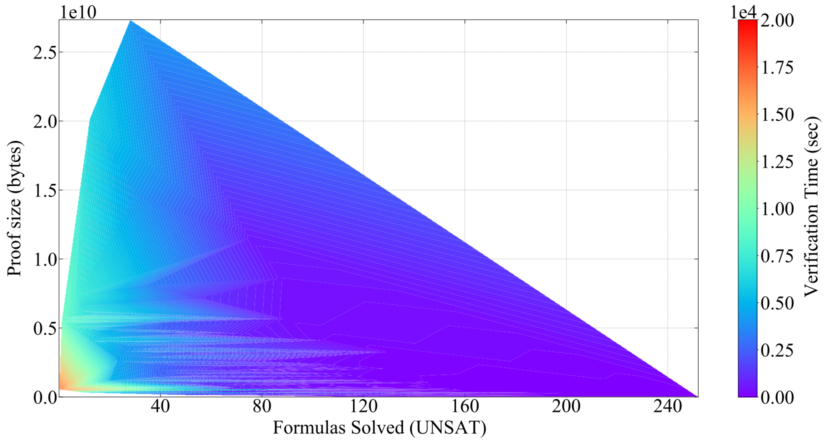
pared to its sequential counterpart (see Figure 5.7), which explains why funTab is not as competitive in ParaFROST (noGPU) as for ParaFROST. On the other hand, Kissat uses a very different method than funTab to find general definitions. It calls a simple solver called Kitten which is responsible for solving and extracting the clausal core from variable environments scheduled for elimination. Further, as expected, Kissat is more efficient than both CaDiCaL and ParaFROST. The reason for the solving discrepancies is that the ParaFROST CDCL heuristics (run on the host) is based on CaDiCaL which is not as up-to-date as Kissat. We expect ParaFROST to compete with Kissat if the same heuristics implemented in the latter is used.

Figures 5.11 and 5.12 show simplification time and its percentage of the total run time, respectively. Clearly, the CPU/GPU solver outperforms the sequential solvers due to the parallel acceleration. Figure 5.12 tells us that ParaFROST keeps the workload in the majority of cases in the region between 0 and 20% as the elimination methods are scheduled on a bulk of mutually independent variables in parallel. In CaDiCaL and Kissat, variables and clauses are simplified sequentially, which takes more time.

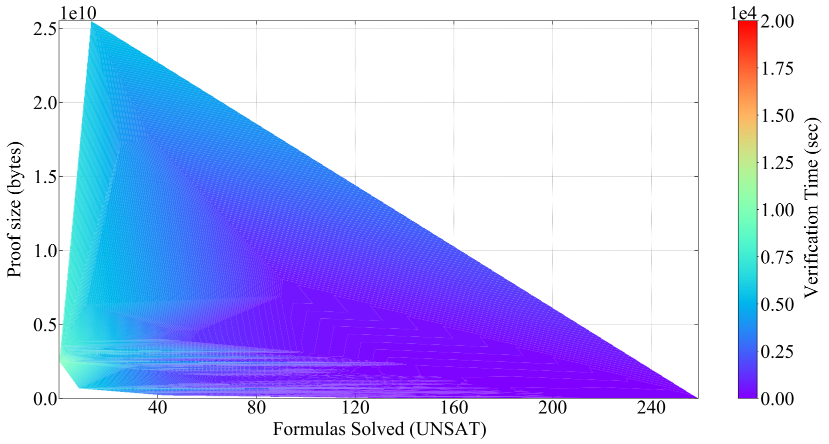
Figure 5.13 reflects the overall efficiency of parallel inprocessing on variables and clauses with(out) funTab on solved formulas with successful clause reductions. Data are sorted in descending order. Reductions can remove up to 95% and 80% of the variables and clauses, respectively.

Figure 5.14 shows the heat-map distribution of the time spent on verification and the memory consumed by the proofs generated by ParaFROST [Osa21b] and CaDiCaL. All proofs are successfully verified via the drat-trim tool. The verification times are represented by the *colormap* and are sorted in descending order w.r.t. the number of *unsatisfiable* instances solved. Plot 5.14a reveals that drat-trim takes more time to verify ParaFROST proofs which appears by the hotspot on the left side of x-axis (ranges between 1.5×10^4 and 1.75×10^4 seconds). That is foreseen as the deleted lemmas in Algorithm 5.5 are not saved to the proof in order to avoid GPU memory exhaustion. On the other hand, CaDiCaL proofs as demonstrated by Plot 5.14b take less time to verify (e.g. 0.75×10^4 to 1×10^4 seconds) due to the saving of all deleted lemmas in BVE which helps drat-trim to cut down the resolution steps. Also, proofs generated by ParaFROST for the formulas 30-40 consume slightly more memory than CaDiCaL owning to the extra resolvents and deleted lemmas produced by the funTab method and ERE, respectively. Those methods are not implemented in CaDiCaL.

Table 5.1 zooms into the impact of applying the funTab method in BVE on solving a sample of 30 formulas using ParaFROST and ParaFROST (noFun),



(a) Verification of ParaFROST proofs



(b) Verification of CaDiCaL proofs

Figure 5.14: Heatmap showing the time-memory distribution of DRAT proof

Table 5.1: funTab impact on 30 formulas solved by ParaFROST. The letters *V* and *C* refer to *Variables* and *Clauses*, respectively. The keywords *org* and *rem* denote *original* and *removed*, respectively.

CNF			**ParaFROST**			**ParaFROST (noFun)**		
Formula	**V (org)**	**C (org)**	**V (rem)**	**C (rem)**	**t (sec)**	**V (rem)**	**C (rem)**	**t (sec)**
01-integer-programming-20-30-40	3,518,573	891,394	**648,093**	**1,776,849**	1,948.05	Time out ($> 3,600$)		
hwmcc15deep-beemhanoi4b1-k37	1,070,494	455,572	**356,713**	**734,111**	3,576.13	Time out ($> 3,600$)		
sokoban-p20.sas.cr.27	852,527	101,334	**50,858**	**294,398**	3,500.59	Time out ($> 3,600$)		
T96.1.1	32,322,587	8,905,808	**4,682,382**	**10,522,621**	1,314.91	4,672,738	10,486,383	982.60
T97.2.1	14,110,352	4,038,010	**2,461,187**	**5,412,098**	2,190.44	2,453,334	5,354,951	1,948.94
newton.2.2.i.smt2-cvc4	843,395	196,151	**100,600**	**446,511**	2,174.61	95,467	382,463	2,028.31
vlsat2_30744_3925645.dimacs	3,887,186	30,744	**1,926**	**343,699**	320.68	Time out ($> 3,600$)		
snw_16_9_Encpre	1,641,152	72,327	**1,763**	**28,757**	3,120.85	Time out ($> 3,600$)		
T99.2.1	21,859,028	6,269,521	**3,790,028**	7,918,822	**1,920.36**	3,788,312	7,926,209	2,025.92
string_compare_safety_cbmc_940	17,856,438	3,416,996	**955,052**	**3,907,091**	3,251.30	953,420	3,857,959	2,849.89
SAT_dat.k80-24_1_rule_1	4,248,961	1,084,904	**752,757**	**2,019,402**	**1,256.88**	751,544	2,008,237	1,292.18
string_compare_safety_cbmc_840	15,657,846	3,062,432	**862,597**	**3,319,353**	**1,803.59**	861,602	3,313,347	1,931.94
9dlx_vliw_at_b_iq5	2,465,696	151,661	**49,339**	**158,940**	**235.79**	48,374	151,153	236.66
hwmcc15deep-beemlifts3b1-k29	3,431,266	1,238,025	**888,976**	**1,830,867**	1,826.12	888,134	1,828,131	1,752.91
HCP-446-105	247,619	29,934	**7,285**	**111,534**	**907.21**	6,655	103,789	1,020.07
manol-pipe-f9b	547,070	183,368	**155,150**	**359,780**	**16.19**	154,581	355,075	17.45
string_compare_safety_cbmc_720	13,123,560	2,636,430	**729,704**	**2,793,919**	1,615.06	729,161	2,738,158	1,472.11
SAT_dat.k70-24_1_rule_3	3,622,259	927,108	**638,741**	1,663,433	**722.73**	638,270	1,664,674	815.90
string_compare_safety_cbmc_760	13,957,146	2,778,972	**769,135**	2,938,157	1,760.92	768,846	2,946,636	1,556.80
hwmcc15deep-6s188-k44	596,873	202,373	**140,408**	**372,555**	1,891.48	140,204	369,277	1,875.26
HCP-470-105	254,981	31,580	**3,773**	**72,682**	536.78	3,573	69,922	477.60
T65.2.0	5,233,340	1,504,336	**933,201**	**2,067,246**	**156.03**	933,011	2,063,029	160.73
T124.2.1	10,333,765	2,943,418	**1,712,069**	3,788,816	**538.11**	1,711,965	3,805,170	746.29
string_compare_safety_cbmc_900	16,967,922	3,275,344	**922,385**	**3,645,490**	2,720.94	922,286	3,641,942	2,684.22
Mickey_out250_known_last147_0	518,695	72,078	**2,442**	**21,009**	**124.08**	2,350	16,664	128.14
sv-comp19_prop-reachsafety.sigma	5,053,926	1,094,922	**132,589**	**497,325**	**1,031.37**	132,506	430,720	1,144.78
HCP-446-60	227,594	27,377	**2,981**	46,528	**532.47**	2,951	50,300	627.87
sted1_0x0-637	336,166	13,431	**28**	**10,259**	**954.43**	Time out ($> 3,600$)		
at-least-two-traffic_b_unsat	1,590,063	78,183	**26,119**	**115,567**	**292.80**	26,115	115,527	295.51
test_v7_r17_vr5_c1_s25451.smt2	2,509,155	560,348	**204,239**	569,220	**2,352.81**	204,236	571,675	2,391.08

respectively. Bold entries in the V's and C's columns indicate that more variables and clauses are removed by enabling funTab in ParaFROST. For example, funTab allowed ParaFROST to remove 4,682,382 variables in the formula *T96.1.1* compared to 4,672,738 by the configuration without funTab. That is 9,644 extra variables are eliminated as an effect of funTab. Additionally, ParaFROST solved many cases faster than ParaFROST (noFun) within the time limit (3,600 seconds). For instance, the formula *HCP-446-105* was solved by ParaFROST with funTab in just 907.21 seconds, while it took 1,020.07 seconds to solve for ParaFROST without funTab.

5.11 Related Work

A simple GC monitor for GPU term rewriting has been proposed by van Eerd *et al.* [vEGH+21]. The monitor tracks deleted terms and stores their indices in a list. New terms can be added at those indices. Maas *et al.* [MRM+12] and Abhinav *et al.* [AN16] investigated the challenges for offloading garbage collectors to an Accelerated Processing Unit (APU). Springer *et al.* [SM19] introduced a promising alternative for stream compaction [BOA09] via parallel defragmentation on GPUs. Our GC, on the other hand, is tailored to SAT solving, which allows it to be simple yet efficient.

Regarding inprocessing, Järvisalo *et al.* [JHB12] introduced certain rules to determine how and when inprocessing techniques can be applied. Biere *et al.* [Bie13] presented LINGELING, the first solver with the ability of finding general gate definitions using a BDD-based approach. Acceleration of the DPLL SAT solving algorithm on a GPU has been done in [PDFP15], where some parts of the search were performed on a GPU and the remainder is handled by the CPU. Incomplete approaches are more amenable to be executed entirely on a GPU, e.g., an approach using metaheuristic algorithms [YIMO15, YOH+20]. Recently, Prevot *et al.* [PSM21] used the GPUs to determine the usefulness of a learnt clause for parallel Portfolio-based solvers. Nonetheless, we are the first to work on CDCL solvers with GPU accelerated inprocessing.

5.12 Conclusion

We have presented compact data structures tailored for SAT inprocessing and various ways to do GPU memory management. Our solver PARAFROST achieved substantial gains through GPU-accelerated inprocessing compared to its sequential version and the state-of-the-art solver CADICAL. With the improvements made to the BVE procedure in this chapter, the usage of atomic operations has been considerably reduced which leads to an average speedup of 1.6× compared to the atomic version. Owing to FUNTAB reasoning, more logical gates can be detected and removed with an average speedup of 11.33× compared to the sequential counterpart.

We proposed the first parallel GC and proof generation on the GPU for SAT applications with average accelerations of 35× and 11×, respectively. The garbage collector helped reduce the GPU memory consumption while stimulating coalesced memory access. The proof generator allowed PARAFROST to validate all the SAT simplifications running on the GPU besides the CDCL search, giving absolute credibility to our solver and its use in critical applications such as model checkers.

Regarding future work, we aim to adopt the inprocessing techniques and the memory management concerning the former to a multi-GPU setup with sufficient load balancing. Another direction is to use ParaFROST in Portfolio-based parallel SAT solving and exploit the GPU capabilities in managing the shared clause database as recently introduced by [PSM21].

Chapter **6**

Multiple Decision Making

"An embarrassing fact that nobody has ever been able to come up with an efficient algorithm to solve the general satisfiability problem, in the sense that the satisfiability of any given formula of size n could be decided in $n^{\mathcal{O}(1)}$ steps."

– Donald Knuth

Most modern and successful SAT solvers are based on the Conflict-Driven Clause-Learning (CDCL) algorithm [SS99, OW20]. CDCL learns from previous assignments whenever a conflict occurs, and based on this, prunes the search space to make better decisions in the future.

Many solvers have been introduced that employ CDCL, such as Grasp [SS99], Chaff [MMZ+01], BerkMin [GN07], MiniSat [ES03a], Glucose [AS09], and CaDiCaL [BFFH20]. Grasp was the first tool applying CDCL, after which Chaff introduced the so-called *two watched literals* optimisation and the Variable State Independent Decaying Sum (VSIDS) decision heuristic (more on these in Section 2.2.2). BerkMin and MiniSat introduced further implementation and heuristics optimisations. The authors behind Glucose presented robust clause deletion and restart heuristics. The CaDiCaL solver introduced the effective use of SAT simplifications [EB05, BJK21, OW19a, OW19b] as an in-processing technique during the solving process [JHB12, OWB21b] and the Variable Move To Front (VMTF) decision queue [BF15].

Contributions

One aspect of CDCL that has always remained the same is that to explore all possible assignments, a single decision is made at a time. In this chapter, we implement MDM in our solver ParaFROST (see Chapter 5) to extend CDCL with the ability to make and propagate multiple decisions at once. Our motivation for this is to further improve the runtime performance of CDCL. To ensure effective selection of non-singleton sets of decisions, we require that the decisions in such a set do not lead to implications. To that extend, we introduce the following contributions:

- ⋆ Local search is becoming more attractive to guide the solver in finding solutions for satisfiable formulas. In this work, we implement a minimal version of WalkSAT [SK93] and run it occasionally to find better truth values to the selected multiple decisions.
- ⋆ ParaFROST restart is extended with interleaved policies as implemented in CaDiCaL which combines both MiniSat and Glucose heuristics and we show how that can be effective in MDM. Moreover, the MDM decision heuristic is extended with the VMTF queue.
- ⋆ We provide a comprehensive evaluation of different MDM configurations. Furthermore, ParaFROST with MDM heuristic is compared to different solvers without MDM including the latest version of Kissat.

The chapter is organised as follows: Section 6.1 presents the preliminaries in SAT solving and our generalisation of CDCL. In Section 6.2, it is explained how MDM can be implemented, focusing on heuristics and optimisations. In Section 6.3, we integrate MDM into CDCL and address its correctness. Finally, benchmark results are given and discussed in Section 6.4, and conclusions are drawn in Section 6.6.

6.1 SAT Solving with CDCL

In this section, we will focus on SAT solving with CDCL approach which is used by most contemporary and successful SAT solvers. CDCL learns from previous assignments, and based on this, prunes the search space to make better decisions in the future (Section 2.2).

During SAT solving, we keep track of a set σ consisting of all literals that have been assigned $\top$. When *applying* an assignment ℓ, σ is updated to $\sigma \cup \{\ell\}$.

With $\ell \models_\sigma \top$, we express that literal ℓ evaluates to **true** w.r.t. σ. This is defined as $\ell \models_\sigma \top \triangleq \ell \in \sigma$. With $\ell \models_\sigma \bot$, we refer to $\neg\ell \in \sigma$. The evaluation of ℓ is *undetermined*, denoted by $\ell \models_\sigma\uparrow$, in case neither $\ell \in \sigma$ nor $\neg\ell \in \sigma$. We refer to the set of variables in σ as $\mathit{var}(\sigma) = \{x \mid x \in \sigma \vee \neg x \in \sigma\}$. The set of all variables in the formula $\mathcal{S}$ is denoted by $\mathit{var}(\mathcal{S}) = \{x \mid \exists C \in \mathcal{S}.x \in C \vee \neg x \in C\}$.

For a clause $C \in \mathcal{S}$, $C \models_\sigma \top$ expresses that C is satisfiable w.r.t. σ. We have $C \models_\sigma \top$ iff $\exists \ell \in C.\ell \models_\sigma \top$. If for all $\ell \in C$, we have $\ell \models_\sigma \bot$, then $C \models_\sigma \bot$. If neither $C \models_\sigma \top$ nor $C \models_\sigma \bot$, we have $C \models_\sigma\uparrow$. Formula $\mathcal{S}$ is satisfiable w.r.t. σ, i.e., $\mathcal{S} \models_\sigma \top$, iff $\forall C \in \mathcal{S}.C \models_\sigma \top$. In that case, σ is a *model* for $\mathcal{S}$. With $\mathit{Free}_\sigma(C)$, we refer to the set of free, unassigned literals in C w.r.t. σ, i.e., $\mathit{Free}_\sigma(C) = \{\ell \in C \mid \ell \notin \sigma \wedge \neg\ell \notin \sigma\}$. A clause C becomes an implication iff $C \models_\sigma\uparrow$ and $|\mathit{Free}_\sigma(C)| = 1$.

6.1.1 The CDCL procedure

Given a CNF formula $\mathcal{S}$, a CDCL-based SAT solver tries to find a model for $\mathcal{S}$. The search is performed in three main steps: *decision making*, *propagating* the effects of assignments, and *analysing* in case so-called conflicts arise. The CDCL procedure is described by Algorithm 6.1. This description generalises the one given in [SS99] without recursion to support MDM. The extension we propose in this manuscript affects the operations at lines 9-10. In the extension, DECIDE (line 9) may not just make a single decision when called, as in standard CDCL, but it may make a set of decisions. How and when DECIDE makes multiple decisions should be ignored for now. This is covered in Sections 6.2 and 6.3.

The global variables of the CDCL procedure are $\mathcal{S}$, σ, and a set ρ of truth values (phases) that were previously assigned to the literals in σ, but have been removed due to backtracking (the purpose of the latter is to apply progress saving, as proposed in [PD07]). In addition, the procedure takes a *decision level function* δ, which records at which *decision/search level* (d) each literal has been assigned a value. The function *source* is also used to keep track of which implications are caused by which clauses. More on this later.

At lines 1-15 of Algorithm 6.1, the CDCL procedure is given, which first calls the procedure BCP (Definition 2.1) at the initial search level ($d = 0$). After that, if no conflicting clause is found, we start searching for a σ that models $\mathcal{S}$. The procedure DECIDE at line 9 implements the decision making step; it decides which decisions to make next, i.e., to which literals $\top$ should be assigned. The level d is subsequently increased by the number of decisions made (for instance, d is incremented if one decision is made).

If no decisions could be made, then $\mathcal{S}$ is satisfied by the current σ, and the

Algorithm 6.1: CDCL, generalised to support MDM

```
Input  : formula $\mathcal{S}$, assignments $\sigma$, saved-phases $\rho$, level function $\delta$, implication function
         $source$
Output : $sat$
procedure CDCL():
    while $sat = \texttt{UNSOLVED}$ do
        $C' = \text{BCP}()$
        if $C' \neq \emptyset$ then
            if $d = 0$ then $sat \leftarrow \texttt{UNSAT}$, break
            $(\hat{C}, \overleftarrow{d}) \leftarrow \textsc{analyse}(C', d)$, $\mathcal{S} \leftarrow \mathcal{S} \cup \hat{C}$
            $\textsc{backjump}(\overleftarrow{d})$, $d \leftarrow \overleftarrow{d}$
        else
            $L \leftarrow \textsc{decide}()$
            $d \leftarrow d + |L|$
            if $L = \emptyset$ then $sat \leftarrow \texttt{SAT}$
            $\sigma \leftarrow \sigma \cup L$
        end
    end
end
procedure BCP():
    $source \leftarrow \emptyset$
    while $\{C \in \mathcal{S} \mid |Free_\sigma(C)| = 1\} \neq \emptyset$ do
        pick a clause $C \in \mathcal{S}$ for which $|Free_\sigma(C)| = 1$
        $\sigma \leftarrow \sigma \cup Free_\sigma(C)$
        $source \leftarrow source \cup \{(Free_\sigma(C), C)\}$
        if $(\exists C' \in \mathcal{S}).C' \models_\sigma \bot$ then return $C'$
    end
    return $\emptyset$
end
procedure $\textsc{backjump}(d)$:
    $\rho \leftarrow \{(var(\ell), \rho(var(\ell))) \mid \delta(\ell) < d\} \cup \{(var(\ell), (\ell \in \sigma)) \mid \delta(\ell) \geq d\}$
    $\sigma \leftarrow \sigma \setminus \{\ell \mid \delta(\ell) \geq d\}$
    $\delta \leftarrow \delta \setminus \{(\ell, d) \mid \delta(\ell) \geq d\}$
end
procedure $\textsc{analyse}(C', d)$:
    $\hat{C} \leftarrow \textsc{learnCClause}\ (C', d)$
    $\overleftarrow{d} \leftarrow \textsc{max}(\{0\} \cup (\{\delta(\hat{\ell}) \mid \hat{\ell} \in \hat{C}\} \setminus \{d\}))$
    return $(\hat{C}, \overleftarrow{d})$
end
procedure $\textsc{learnCClause}(C', d)$:
    $\hat{C} \leftarrow C'$
    while $|\{\ell \in \hat{C} \mid \delta(\ell) = d\}| > 1$ do
        pick a literal $\ell \in \hat{C}$ for which $\exists C.(\neg\ell, C) \in source$
        $\hat{C} \leftarrow \hat{C} \otimes_\ell source(\neg\ell)$
    end
    return $\hat{C}$
end
```

search loop terminates by setting *sat* to `SAT` at line 11. Otherwise, at line 12, the new decisions in L are added to σ.

The propagation of a decision and all its resulting implications is done by the BCP procedure, called repeatedly at line 3. The description of BCP is given at lines 16-25. As long as there are unit clauses (line 18), a unit clause C is picked (line 19), and its unassigned literal is added to σ (line 20). If the update does not make $\mathcal{S}$ unsatisfied (line 22), the procedure is repeated. After every update of σ, the function *source* is updated to record the source of the implication (i.e. the fact that C caused $\mathit{Free}_\sigma(C)$ to be added to σ). This is relevant when clause learning needs to be applied, which is discussed later. The BCP procedure identifies all implications, unless a *conflict* is detected in some clause C'.

Definition 6.1 (Conflict)**.** Given a formula $\mathcal{S}$ and a set of assignments σ, there is a *conflict* iff $\exists C' \in \mathcal{S}$ such that $C' \models_\sigma \bot$.

Recall that at the start of CDCL, BCP is called. The purpose of this call is to propagate implications initially present in $\mathcal{S}$ or decisions made recently by previous calls to DECIDE. In case BCP returns a non-empty clause at line 3, there is a conflict, and this must be analysed (line 6). First of all, if the current level is 0 (i.e. top level), then the formula is unsatisfied and the loop is terminated at line 5. Otherwise, the ANALYSE procedure, given at lines 31-35, tries to identify all assignments causing the conflict and the backtrack level $\overleftarrow{d}$ required to undo those assignments.

The backtrack step is done in the BACKJUMP procedure at line 7, described by lines 26-30. All assignments made at the current level up to the backtracking level are removed from σ and δ, and their phases are saved to ρ (old phases are overwritten). In this way, ρ can be used to recall the last value assigned to a variable (phase). This is relevant in DECIDE; in Section 6.2, we specify how ρ is used in decision making. After that the current decision level d is set to the backtracking level $\overleftarrow{d}$ at line 7.

The negations of the backtracked assignments can be recorded by in a new clause, called the *learnt clause* ($\hat{C}$), to avoid making that combination of assignments in the future. This clause can be constructed by a sequence of resolution steps [SS99], starting with the clause that caused the conflict, where each step produces an intermediate new clause (*resolvent*). A resolvent is the result of applying the *resolution rule* [DLL62] w.r.t. some implication ℓ at level d (see Definition 2.3). For this, we use the *resolving operator* $\otimes_\ell$ on clauses C_1 and C_2 with $\ell \in C_1$, $\neg\ell \in C_2$. The procedure stops once a fix-point has been reached.

At resolution step i $(0 \leq i < |var(\mathcal{S})|)$, resolvent $\hat{C}_i^d$ is defined as follows.

$$\hat{C}_i^d = \begin{cases} C' & ,\text{if } i = 0 \wedge C' \in \mathcal{S} \wedge C' \models_\sigma \bot \\ \hat{C}_{i-1}^d \otimes_\ell source(\neg\ell) & ,\text{if } i > 0 \wedge \ell \in \hat{C}_{i-1}^d \wedge source(\neg\ell) \neq \emptyset \\ \hat{C}_{i-1}^d & ,\text{if } i > 0 \wedge |\{\ell \in \hat{C}_{i-1}^d \mid \delta(\ell) = d\}| = 1 \end{cases} \tag{6.1}$$

In the initial case ($i = 0$) the conflicting clause is selected, while the second and following cases give the intermediate resolvent at resolution step i. Note that if there are multiple assignments at d, all but one of them have a source. The final case defines the fix-point, resulting in the learnt clause. The fix-point is reached when only one assignment remains in $\hat{C}_{i-1}^d$ at the conflict level d. This assignment is called the *first unique implication point* (*1-UIP*) which is the closest node in the implication graph to the conflict within the same decision level [ZMMM01].

Equation 6.1 is implemented by the LEARNCCLAUSE procedure given at lines 36-43. As long as more than one literal in $\hat{C}$ was assigned a value at conflict level d (line 38), $\hat{C}$ is rewritten. This is done by selecting a literal ℓ that obtained a value due to an implication at d (i.e. $source(\neg\ell)$ is defined) at line 39, and combining the clause that caused that implication ($source(\neg\ell)$) with $\hat{C}$ using the resolving operator, resulting in a new clause in which ℓ no longer appears (line 40).

Given $\hat{C}$, the backtrack level $\overleftarrow{d}$ is defined as $\text{MAX}(\{0\} \cup (\{\delta(\ell) \mid \ell \in \hat{C}\} \setminus \{d\}))$ (line 33): the highest level involved in $\hat{C}$ that is smaller than d is selected. In case no such level exists ($\hat{C}$ contains a single literal at d), 0 is selected. Both $\hat{C}$ and $\overleftarrow{d}$ are returned at line 6. Next, $\hat{C}$ is added to the input formula, and the backtracking to $\overleftarrow{d}$ is initiated by BACKJUMP as explained above.

Example 6.1. A small example of applying single-decision CDCL is illustrated in Figure 6.1 by an implication graph [SS99]. Consider a formula $\mathcal{S}$ containing, among others, the following clauses:

$$\mathcal{S} = \{\{\neg x_8, x_2, \neg x_7\}, \{\neg x_8, \neg x_2, x_3\}, \{\neg x_3, \neg x_7\}, \{x_7, x_5, x_6\}, \\ \{\neg x_5, x_4\}, \{\neg x_6, x_4, \neg x_1\}, \{x_9, \neg x_{10}\}, \{\neg x_9, \neg x_{10}, x_{11}\}, \ldots\}$$

In addition, consider x_8, x_1, x_9 and $\neg x_4$ as the decisions made so far. The decisions and their levels are indicated in Figure 6.1 by the green ovals and blue numbers, respectively. Once these decisions are made, BCP identifies implications. Each implication (white ovals in Figure 6.1) takes the highest level among its parents. The procedure first sets $\neg x_5$, $\neg x_6$ and x_7 to $\top$, before there

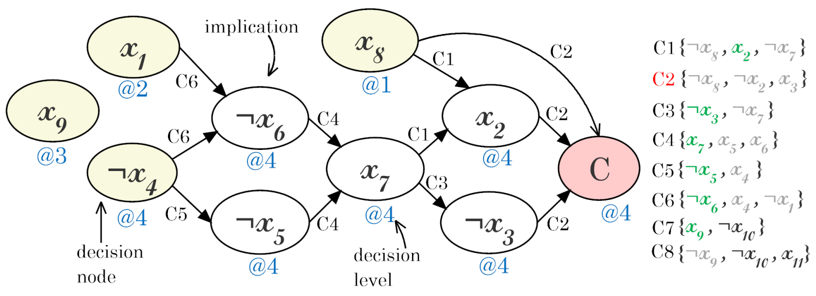


Figure 6.1: A visualization of CDCL solving on a small example

are a number of possible scenarios that all lead to a conflict. The order in which unit clauses are analysed by BCP determines which scenario occurs. In the figure, both x_2 and $\neg x_3$ are set to $\top$, before C_2 causes a conflict.

At this point, LEARNCCLAUSE can produce the clause $\hat{C} = \{\neg x_7, \neg x_8\}$, as the result of $(C_2 \otimes_{x_2} C_1) \otimes_{x_3} C_3$. Finally, the backtrack level $\overleftarrow{d}$ is determined by the highest decision level other than the conflict level in $\hat{C}$; in this example, $\overleftarrow{d} = 1$.

6.1.2 Multiple Decision Making

Next, we reason about making multiple decisions simultaneously with DECIDE. Making a decision can always lead to conflicts, and making more than one decision at once increases the likelihood of a conflict occurring. To avoid repeatedly selecting sets of decisions that cause conflicts, we wish to construct multiple decisions sets, i.e., non-singleton sets of decisions, in such a way that it is guaranteed that no conflicts will occur. For this, we define multiple decisions set as follows.

Definition 6.2 (Multiple decisions set)**.** Given a formula $\mathcal{S}$ and a set of assignments σ, we call a set $\mathcal{M} \subseteq \mathbb{L} \setminus \{\ell, \neg\ell \mid \ell \in \sigma\}$ with $|\mathcal{M}| > 1$ a set of *multiple decisions* iff $\{C \in \mathcal{S} \mid |\mathit{Free}_{\sigma \cup \mathcal{M}}(C)| = 1\} = \emptyset$.

In words, a set of multiple decisions $\mathcal{M}$ can be selected, i.e., is *valid*, iff $\mathcal{M}$ does not result in unit clauses. Note that $\mathcal{M}$ only contains new decisions, and not previously selected literals or ones that contradict σ. Definition 6.2 cannot be efficiently used to construct a set of multiple decisions, since it refers to $\mathcal{M}$ as a whole. The basic approach to construct $\mathcal{M}$ is to iteratively select a

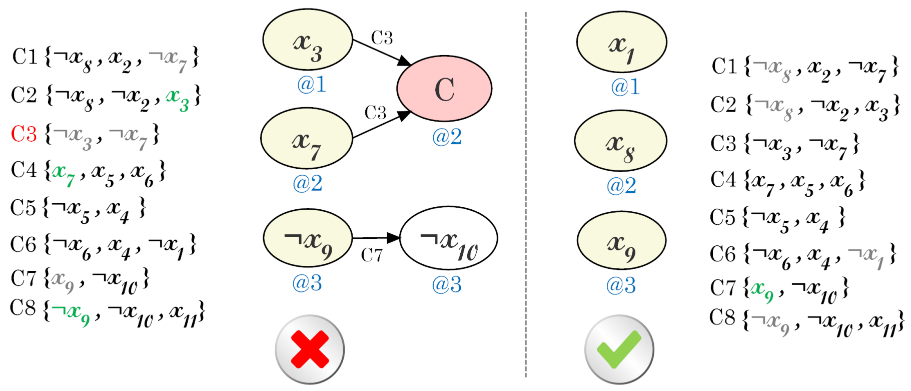


Figure 6.2: An example of (non)-valid multiple decisions sets

decision ℓ and add it to $\mathcal{M}$ iff it does not lead to a violation of the condition in Definition 6.2.

Example 6.2. By Definition 6.2, the set of literals $\{x_3, x_7, \neg x_9\}$ for the formula in Figure 6.2 is initially, with $\sigma = \emptyset$, not a valid multiple decisions set, as it results in C_3 being unsatisfied and C_7 being a unit clause. On the other hand, $\{x_1, x_8, x_9\}$ is valid.

Since a valid multiple decisions set does not produce any unit clauses, DECIDE should not always, but periodically select multiple decisions. In the next section, we discuss a possible mechanism to achieve this.

6.2 MDM with Decision Heuristics

So far, we have presented MDM mathematically. In this section, we discuss how to make it efficient in practice, and we address the correctness of CDCL with MDM.

6.2.1 Decision Heuristics

Recall that the decision-making step via the DECIDE routine in Algorithm 6.1 determines which literals should be selected and assigned **true** for the next decisions (Section 2.2). In this article we use the decision heuristics VSIDS [MMZ+01]

Algorithm 6.2: Variable ranking of different MDM decision queues

```
    Input  : $\mathcal{S}$, restart mode, VSIDS score $\alpha$, VMTF score $\gamma$
 1  procedure MDMRANK($Q$):
 2      $h \leftarrow$ HISTOGRAM($\mathcal{S}$)
 3      $freevars \leftarrow \{\, x \mid x \in Q \wedge x \models_\sigma \uparrow \,\}$
 4      forall $x \in freevars$ do
 5          $\beta(x) \leftarrow h(x) \times h(\neg x)$
 6      end
 7      if $mode$ = STABLE then
 8          return SORT($freevars$, VSIDSKEY)
 9      else
10          return SORT($freevars$, VMTFKEY)
11      end
12  end
13  procedure VSIDSKEY($x$, $y$):
14      if $\alpha(x) \neq \alpha(y)$ then return $\alpha(x) > \alpha(y)$
15      if $\beta(x) \neq \beta(y)$ then return $\beta(x) > \beta(y)$
16      return $x > y$
17  end
18  procedure VMTFKEY($x$, $y$):
19      return $\gamma(x) > \gamma(y)$
20  end
```

and VMTF [BF15] to improve the quality of the multiple decisions made. In contrast to VSIDS, the variable score in VMTF queue (denoted as γ) is the number of conflicts at which it was last bumped. VMTF is implemented in CaDiCaL and our solver with a doubly-linked list. In CDCL, in the DECIDE step in Algorithm 6.1, one can alternate between VSIDS and VMTF queues based on the restart mode in a ping-pong manner. More on restart modes in Section 6.3.

Initially, a sorted list of the *unassigned* variables in $\mathcal{S}$ is created and returned by the MDMRANK procedure in Algorithm 6.2. As input, it requires the formula $\mathcal{S}$, the restart *mode*, the variable scores α, γ, and the decision queue Q. This queue includes all variables in $\mathcal{S}$ ranked w.r.t. α and γ in ascending order (i.e. first variable has always the highest score). At line 2, the histogram h is calculated for all literals in $\mathcal{S}$ where $h(\ell) = |\mathcal{S}_\ell|$. Next, a list of unassigned variables *freevars* is constructed from Q (line 3). Then, at line 4, we iterate over *freevars* to calculate the histogram score β. Finally, based on the restart *mode*, the SORT procedure ranks all variables in *freevars* w.r.t. vsidsKey (lines 13-17) or vmtfKey (lines 18-20). It is important to remark that the comparison at line 19 is sufficient to obtain stable sorting, as all variables have a distinct γ value.

6.2.2 2-WL Optimisation

When constructing a multiple decisions set, checking whether the condition of Definition 6.2 (i.e. multiple decisions cannot produce unit clauses) still holds every time a literal ℓ is selected can be optimised by using the 2-WL optimisation [MMZ+01]. We refer the reader to Section 2.2.1 for more details. This optimisation is particularly suitable to check for violations of the condition of Definition 6.2, as it allows us to only consider the clauses in $\mathcal{S}_{\neg\ell}$ in which $\neg\ell$ is watched. If $\neg\ell$ is not watched in some clause $C \in \mathcal{S}_{\neg\ell}$, then there are at least two other, unassigned literals in C, hence C cannot become unit when $\neg\ell$ is assigned $\bot$. If there are no more than two watched literals in C, then $\neg\ell$ has to be watched, since it was unassigned before being selected. We formalise the predicate that a literal ℓ is watched in a clause C with $\mathcal{W}_C(\ell)$.

6.2.3 Decision Freezing

During the construction of a multiple decisions set, we need to avoid the repeated selection of literals that cause the condition of Definition 6.2 to be violated. Consider the scenario in Figure 6.3 where the decisions $\{x_2, x_8, x_9\}$ are selected. By doing so, C_2 will become a unit clause as a consequence of assigning $\top$ to both x_2, x_8. One way to resolve this is to drop all variables that exist in the same watched clauses (i.e. that are dependent) of a previously selected decision. For this reason, we introduce the notion of *freezing*. With it, we over-approximate the potential to produce unit clauses. If we add a valid literal ℓ to $\mathcal{M}$, the clauses in $\mathcal{S}_{\neg\ell}$ have $\bot$ assigned to $\neg\ell$, and thereby have more potential to become unit. Subsequently selecting a literal ℓ' that also appears in any of those clauses in $\mathcal{S}_{\neg\ell}$ can possibly produce a unit clause. To avoid this, we freeze all unassigned literals in $\mathcal{S}_{\neg\ell}$ after the selection of ℓ, thus not allowing them to be selected for the multiple decisions set.

Checking whether a literal is frozen or not is straightforward than repeatedly checking for a violation of the condition in Definition 6.2. A literal ℓ' is frozen if either ℓ' or $\neg\ell'$ *depends* on a previously selected decision:

Definition 6.3 (Decision dependency relation)**.** We call a relation $\mathsf{D}\colon \mathbb{L} \times \mathbb{L}$ a *decision dependency relation* iff for all $\ell, \ell' \in \mathbb{L}$, we have $\ell' \mathrel{\mathsf{D}} \ell$ iff there exists a $C \in \mathcal{S}_{\neg\ell}$ such that $\ell' \in C \vee \neg\ell' \in C$.

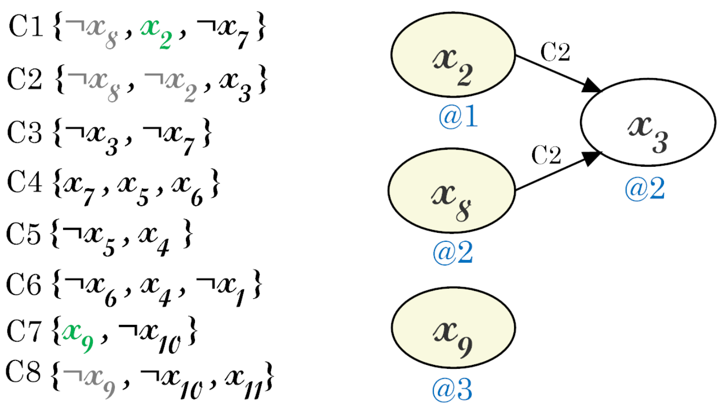

C1 $\{\neg x_8, x_2, \neg x_7\}$
C2 $\{\neg x_8, \neg x_2, x_3\}$
C3 $\{\neg x_3, \neg x_7\}$
C4 $\{x_7, x_5, x_6\}$
C5 $\{\neg x_5, x_4\}$
C6 $\{\neg x_6, x_4, \neg x_1\}$
C7 $\{x_9, \neg x_{10}\}$
C8 $\{\neg x_9, \neg x_{10}, x_{11}\}$
x_2
@1
C2
x_3
@2
C2
x_8
@2
x_9
@3

Figure 6.3: An example of decision dependency

Given a multiple decisions set $\mathcal{M}$, the set of frozen decisions is defined as follows.

Definition 6.4 (Frozen decisions)**.** Given a formula $\mathcal{S}$, a set of assignments σ, and a multiple decisions set $\mathcal{M}$, the set of *frozen decisions* $\mathcal{F}$ is defined as $\mathcal{F} = \{var(\ell') \mid \ell' \in \mathbb{L} \setminus \{\ell'', \neg\ell'' \mid \ell'' \in \sigma\} \wedge \exists \ell \in \mathcal{M}.\ell' \text{ D } \ell\}$.

In fact, we freeze variables, to exclude their corresponding positive and negative literals in the formula.

6.2.4 The MDM Procedure with Optimisations

Next, we explain how the MDM procedure can be implemented. As input, Algorithm 6.3 requires the current decision level d, the saved phases ρ, and the level function δ. The MDM main routine, given by lines 1-15, takes the decision queue Q and produces a set of multiple decisions $\mathcal{M}$. At line 2, the current level is stored in d', which is used to assign consecutive levels to the decisions selected by MDM. Sets $\mathcal{M}$ and $\mathcal{F}$ are initially empty. At line 3, we create a ranked list of free variables *rankedvars* using the MDMRANK procedure. Next, we iterate over *rankedvars* (line 4).

For each unfrozen variable (line 5), a value is picked to select a literal. The LIT function at line 6 takes a variable x and the saved phase $\rho(x)$ and returns an assignment ℓ. At line 7, it is checked whether the selected literal ℓ is valid,

Algorithm 6.3: Multiple Decision Maker

```
Input    : saved-phases $\rho$, decision level $d$, level function $\delta$
procedure MDM($Q$):
    $d' \leftarrow d$, $\mathcal{M} \leftarrow \emptyset$, $\mathcal{F} \leftarrow \emptyset$
    $rankedvars \leftarrow$ MDMRANK($Q$)
    forall $x \in rankedvars$ do
        if $x \notin \mathcal{F}$ then
            $\ell \leftarrow$ LIT($x, \rho(x)$)
            if ISVALID($\ell$) $\wedge$ DEPFREEZE($\ell$) then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{\ell\}$
                $\delta(x) \leftarrow d'$, $\delta(\neg x) \leftarrow d'$
                $d' \leftarrow d' + 1$
            end
        end
    end
    return $\mathcal{M}$
end
procedure ISVALID($\ell$):
    forall $C \in \mathcal{S}_{\neg\ell}$ do
        if $\mathcal{W}_C(\neg\ell) \wedge |Free_\sigma(C)| = 1$ then return false
    end
    return true
end
procedure DEPFREEZE($\ell$):
    forall $C \in \mathcal{S}_{\neg\ell}$ do
        if $\mathcal{W}_C(\neg\ell)$ then
            forall $\ell' \in C$ do
                if $\ell' \in \mathcal{M}$ then return false
                if $var(\ell') \neq var(\ell) \wedge var(\ell') \in rankedvars$ then
                    $\mathcal{F} \leftarrow \mathcal{F} \cup var(\ell')$
                end
            end
        end
    end
    return true
end
```

according to Definition 6.2. If ℓ is valid, it is attempted to freeze dependent variables. If successful, ℓ is added to $\mathcal{M}$ and recorded with δ at level d' (lines

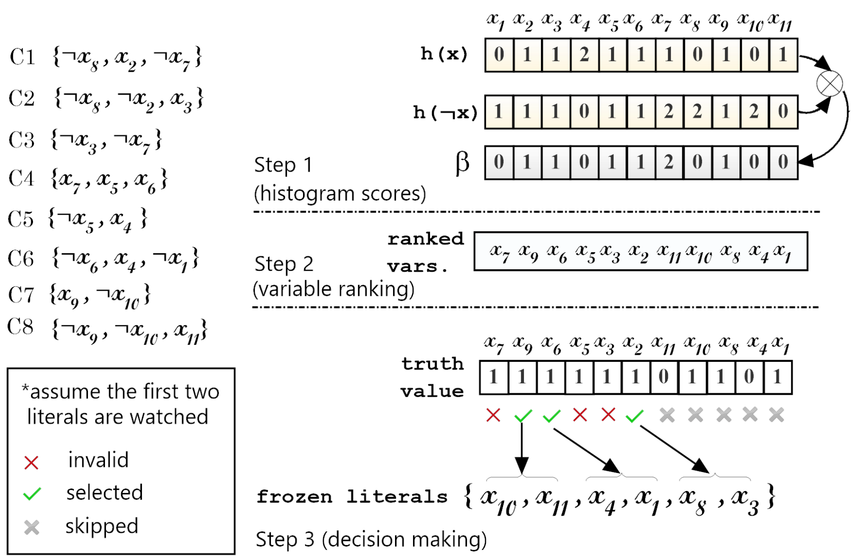

C1 {¬x8, x2, ¬x7}
C2 {¬x8, ¬x2, x3}
C3 {¬x3, ¬x7}
C4 {x7, x5, x6}
C5 {¬x5, x4}
C6 {¬x6, x4, ¬x1}
C7 {x9, ¬x10}
C8 {¬x9, ¬x10, x11}
x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11
h(x) 0 1 1 2 1 1 1 0 1 0 1
h(¬x) 1 1 1 0 1 1 2 2 1 2 0
β 0 1 1 0 1 1 2 0 1 0 0
Step 1 (histogram scores)
ranked vars. x7 x9 x6 x5 x3 x2 x11 x10 x8 x4 x1
Step 2 (variable ranking)
x7 x9 x6 x5 x3 x2 x11 x10 x8 x4 x1
truth value 1 1 1 1 1 1 0 1 1 0 1
frozen literals {x10, x11, x4, x1, x8, x3}
Step 3 (decision making)
*assume the first two literals are watched
invalid
selected
skipped

Figure 6.4: A working example of multiple decision making procedure

8-9). Finally, the level d' is incremented at line 10.

The procedure ISVALID (given at lines 16-21) restricts the proof of a literal ℓ validity to clauses in $\mathcal{S}_{\neg\ell}$ in which $\neg\ell$ is watched. The procedure DEPFREEZE (given at lines 22-34) tries to freeze all variables dependent on ℓ, according to Defs. 6.3 and 6.4. We apply the 2-WL optimisation to speed up iterating over clauses in $\mathcal{S}_{\neg\ell}$. This has the drawback that some literals that have to be frozen may be ignored, if they do not appear in a clause in which $\neg\ell$ is watched. To remedy this, we check whether a literal has already been added to $\mathcal{M}$ before freezing it (line 26). If it was added, then clearly, ℓ should not have been added, and the latter's selection is canceled.

Example 6.3. Figure 6.4 shows a working example of Algorithm 6.3. For simplicity, we assume that the α- and γ-scores are all 0. As a first step, the histogram scores β are computed for all variables in the formula. Next, we rank variables using the MDMRANK procedure in Algorithm 6.2. Finally, we start from the highest ranked variable, trying to select valid decisions. Variables are assigned to their assumed saved phases. Obviously, x_7 is invalid due to the implication in C_3. Thereby, x_9 is selected instead and all dependent variables on $\neg x_9$ are frozen. At the end of the procedure, the decisions set $\{x_9, x_6, x_2\}$ is successfully constructed.

6.3 MDM Integration in CDCL

As already noted in Section 6.1.1, DECIDE cannot always select multiple decisions, as the production of implications cannot indefinitely be avoided if there are still variables left to assign. The main question is therefore when MDM should be applied. In [OW20], MDM was integrated into MINISAT and GLUCOSE, and since multiple decisions should be selected periodically, a mechanism was proposed that decides when to make multiple decisions based on the solver restart policy and rate. However, since solvers can differ greatly in this policy, we wanted to create an alternative mechanism not depending on this.

PARAFROST is based on CADICAL, which has a very different restart policy compared to MINISAT and GLUCOSE. The policy is to interleave different restart sequences together to remedy the shortcomings of one another and play to the strengths of all. The idea is to start with *geometric* [ES03a] style with less frequent restarts (i.e. called in CADICAL as `STABLE` mode) then switch to a more aggressive style using dynamic restarts [AS12, BF18] after some interval scaled up by a quadratic function based on the number of conflicts [BFFH20].

In Algorithm 6.4, at lines 1-20, an extended version of the DECIDE procedure is proposed. Initially, the variable *lastRounds* is set to the constant *rounds*. The WALKSAT procedure is called at line 4 per first round at the top level (line 3) to improve the saved phases as discussed in Sec. 6.2. If we are in the process of calling MDM *lastRounds* times, then MDM is called again at line 6 and *lastRounds* is decremented. To select decisions, the queue Q is needed. The alternative is the standard decision making at line 14 (i.e. single decision selection). This queue prioritises a variable based on α if the restart mode *mode* is `STABLE` otherwise γ is used (see Algorithm 6.2). Only after the first MDM call ($mdmCalls = 1$), the PUMPFROZEN procedure is called at line 10, to mitigate what we call the *overjump effect*. More on this in the next section. If we have stopped calling MDM, and enough unassigned variables are present, method PERIODICFUSE is called, which either sets *lastRounds* back to *rounds* or to 0, depending on the total number of conflicts *numConflicts*, the number of MDM calls *mdmCalls*, and the MDM conflicts limit *mdmLimit*. There are enough unassigned variables if there are more free variables compared to the most recent multiple decisions selected.

In PERIODICFUSE, *numConflicts* is compared to *mdmLimit*, which is initially

Algorithm 6.4: DECIDE, with integrated MDM

```
Input    : decision queue $Q$, assignments $\sigma$, saved-phases $\rho$, MDM *rounds*, nr.
           last selected decisions *lastMDs*, nr. conflicts *numConflicts*, MDM
           conflicts limit *mdmLimit*, MDM conflicts step *mdmStep*, nr. MDM
           calls *mdmCalls*, decision level $d$
procedure DECIDE():
    if $lastRounds > 0$ then
        if $d = 0 \land lastRounds = rounds$ then
            WALKSAT($\rho$)
        end
        $L \leftarrow$ MDM($Q$), $mdmCalls \leftarrow mdmCalls + 1$
        $lastRounds \leftarrow lastRounds - 1$
        $lastMDs \leftarrow |\mathcal{M}|$
        if $mdmCalls = 1$ then
            PUMPFROZEN($mode, \mathcal{M}$)
        end
    end
    else
        $L \leftarrow$ SINGLEDECISION($Q$)
        if $lastMDs \leq |Q \setminus var(\sigma)|$ then
            $lastRounds \leftarrow$ PERIODICFUSE()
        end
    end
    return $L$
end
procedure PERIODICFUSE():
    if ($numConflicts \geq mdmLimit$) then
        INCREASELIMIT(SCALEFUNCTION, $mdmCalls$, $mdmLimit$)
        return $rounds$
    else
        return 0
    end
end
```

set to a configurable value (default 2,000). The *mdmLimit* parameter is monotonically increased using a function INCREASELIMIT. This makes *mdmLimit* grow linearly, quadratically, or logarithmically based on the configurable scaling function SCALEFUNCTION and the current value of *mdmCalls*, to achieve a suitable balance between *mdmLimit* and *numConflicts* as the solving progresses.

6.3.1 The Overjump Effect

Consider the multiple decisions set $\{x_1, x_8, x_9\}$ selected in Example 6.2. Assume that after MDM has made this set, singleDecision is called by decide, and that this procedure selects $\neg x_4$, as Figure 6.1 shows. Since this leads to a conflict, assume that the binary clause $\{\neg x_7, \neg x_8\}$ is learned. Thus, all assignments up to level 1 must be undone, including the decision x_9 at level 3 (see Figure 6.1), even though it is not related to the conflict at all (in other words, x_9 is jumped over). Unnecessarily undoing decisions can in general negatively affect the solving procedure, killing any potential to explore large parts of the search tree. Although this can occur in standard CDCL as well, suitable heuristics, such as VSIDS, mitigate this effect. They tend to favour sequences of decisions that are closely related, due to how α evolves over time, making it unlikely that a sequence such as the one in Example 6.2, with x_9 not being related to the other decisions at all, is made.

Moreover, in standard CDCL, sequences of decisions without implications are not specifically stimulated, making it more likely that conflicts occur earlier, which in turn influences the variable scores. On the other hand, when MDM makes a set of decisions, the variable activity (α, γ) can only evolve after the complete set has been selected, hence they have much less influence on the subsequent selection of the decisions when that set is constructed. A subsequent call of singleDecision may then select a decision that is (possibly transitively) dependent on any of the previously made multiple decisions.

To remedy the overjump effect, we therefore have to stimulate at the start of solving (initial MDM call, see line 10 in Algorithm 6.4) that after making multiple decisions, the subsequent standard single decisions are dependent on the most recently made multiple decision, i.e., the one with the highest decision level. If no such decisions can be made, the ones dependent on the decision with the second highest level must be stimulated, and so on.

The prioritisation algorithm in Algorithm 6.5 plays a vital role in this. Recall that it is called by decide after the initial selection of decisions (line 10 in Algorithm 6.4). It gives priority to the single decisions implied by the most recent multiple decisions, taking into account the type of the variable activity. If the current *mode* is `STABLE` (line 2), then the decisions in $\mathcal{M}$ are ordered by $\geq_\delta$, which is defined as $\ell \geq_\delta \ell' \triangleq \delta(\ell) \geq \delta(\ell')$. Next, at line 4, the VSIDS factor $factor_\alpha$ is computed in terms of the number of decisions made. We ensure that this value does not exceed 1, to prevent this prioritisation from interfering with the standard VSIDS activity α. Alternatively, the VMTF factor $factor_\gamma$ is calculated as the maximum score in VMTF activity. Recall that VMTF scores

Algorithm 6.5: Single decisions prioritisation, with variable activity

```
Input : VSIDS score $\alpha$, VMTF score $\gamma$, decision level function $\delta$
procedure pumpFrozen($mode, \mathcal{M}$):
    if $mode = $ STABLE then
        sort($\mathcal{M}, \geq_\delta$)
        $factor_\alpha \leftarrow 1 \ / \ |\mathcal{M}|$
    else
        $factor_\gamma \leftarrow$ max($\gamma$) $+ 1$
    end
    for $\ell \in \mathcal{M}$ do
        forall $C \in \mathcal{S}_{\neg\ell}$ with $\mathcal{W}_C(\neg\ell)$ do
            forall $\ell' \in C$ with $var(\ell') \in \mathcal{F}$ do
                if $mode = $ STABLE then
                    $\alpha(var(\ell')) \leftarrow \delta(\ell) \times factor_\alpha$
                else
                    $\gamma(var(\ell')) \leftarrow factor_\gamma, factor_\gamma \leftarrow factor_\gamma + 1$
                end
                $\mathcal{F} \leftarrow \mathcal{F} \setminus var(\ell')$
            end
        end
    end
end
```

are always monotonically bumped with an integer value, thereby, prioritising single decisions based on VMTF activity can only be achieved by bumping their scores with a value higher than the local maximum.

The **for** loop at lines 8-19 iterates over the decisions in $\mathcal{M}$. In the nested **for** loops at lines 9-18, the variable activity of every *frozen* variable referred to by a literal in a clause in which $\neg\ell$ is watched is increased (pumped) by the corresponding *factor* only once (notice that a frozen variable once updated is removed from the $\mathcal{F}$ set at line 16). For VSIDS, at line 12, a normalised value ($\delta(\ell) \times factor_\alpha$) is computed such that $0 < \alpha(var(\ell')) < 1$, based on the decision level $\delta(\ell)$. For VMTF, the factor $factor_\gamma$ is only incremented such that, the last decision ℓ in $\mathcal{M}$, with the highest level, pumps the largest score to $\gamma(var(\ell'))$.

6.3.2 Correctness of Applying MDM in CDCL

Finally, we can reason about the correctness of applying MDM in CDCL, i.e., as in Algorithm 6.1 returns SAT iff $\mathcal{S}$ is satisfiable, and returns UNSAT otherwise. We argue that this is the case, by showing that each execution of CDCL with MDM, which, for clarity, we refer to as MDCL (Multiple Decision Clause Learning), can be matched by an execution of CDCL without MDM, or CDCL for short. MDCL and CDCL only differ in the fact that MDCL sometimes calls MDM, leading to a forward jump to a level higher than the current level. Clearly, every time DECIDE executes SINGLEDECISION in MDCL, CDCL can match the decision made. When DECIDE executes MDM in MDCL, multiple decisions $\ell_1, \ldots, \ell_n$ are made, each decision not leading to any implications, by Definition 6.2. Note in Algorithm 6.3 that each time a decision ℓ_i $(1 \leq i \leq n)$ is added to $\mathcal{M}$ at line 8, the validity constraint at line 7 is satisfied, which implies that there are no unit clauses, but also that there is no clause C for which $C \models_{\sigma \cup \{\ell_i\}} \bot$. Selecting n decisions in Algorithm 6.1 results at line 10 in d possibly being increased to some value higher than $d + 1$, which directly leads to one call of BCP to check for conflicts at line 3 of Algorithm 6.1. In CDCL, this can be matched by successively making the same decisions $\ell_1, \ldots, \ell_n$ in n calls of DECIDE at line 9 of Algorithm 6.1. After each of the first n decisions, BCP is called at line 3, and since no clause is unsatisfied and no unit clauses exist, BCP returns **true**. After the next decision ℓ_{n+1} has been made, BCP detects conflicts iff MDCL does as well, and if no conflicts occur, DECIDE is called again.

6.4 Benchmarks and Analysis

We implemented the proposed algorithms in C++ in our solver ParaFROST[17]. We evaluated the experiments on the full benchmark suite of the SAT competition 2020 main track which contains 400 formulas. The time-out per formula is set to 5,000 seconds. The experiments involve different configurations of MDM in ParaFROST to find the most impactful one on SAT solving. Moreover, we compare ParaFROST with(out) MDM with the state-of-the-art solvers, MiniSat v2.2, Glucose v3.0 (the MDM version in [OW20] is also included), CaDiCaL v1.4, and Kissat (pulled from GitHub as of April 12, 2021).

The default settings are used in all solvers. The MiniSat and Glucose versions with simplifications are used in the experiments. As in CaDiCaL and Kissat, all inprocessing features such as *vivification*, *autarky* and *probing* are implemented in ParaFROST with various optimisations and better scheduling. More information on this is available through the solver repository [17].

All experiments were conducted using the compute nodes of the DAS-5 cluster [BEdL+16]. Each problem was analysed in isolation on a separate node. Each node has an Intel Xeon E5-2630 CPU running at 2.4 GHz with 128 GB

[17] https://github.com/muhos/ParaFROST

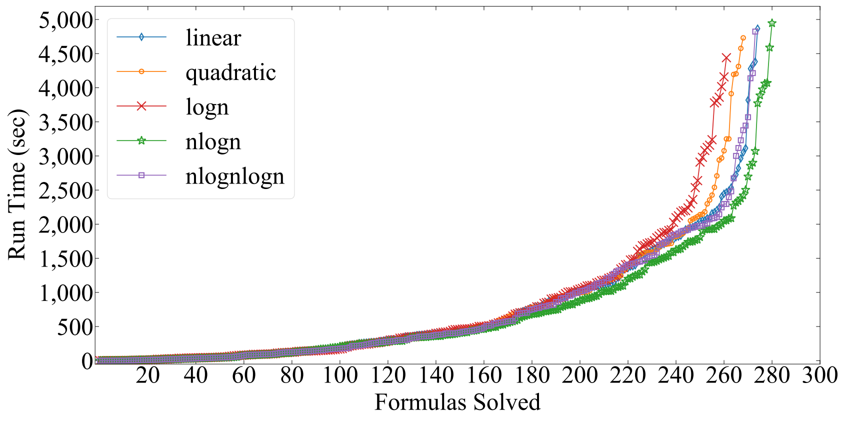


Figure 6.5: MDM scaling functions (*rounds* $= 3$)

of system memory, and runs on the CentOS 7.4 operating system. With this information, we adhere to all of the five principles laid out in the SAT manifesto (version 1) available at [BJLB+20].

Figures 6.5-6.10 presents the results of evaluating different MDM configurations in ParaFROST and their impact on solving. Data are sorted w.r.t. the y-axis. For all figures, we change only one function or parameter while keeping the others fixed. These include the following: scaleFunction, *rounds*, *mdmStep*, pumpFrozen, mdmrank, WalkSAT. For example, Figure 6.5 compares different scaling functions while keeping the *rounds* and other settings to the same values. In Figure 6.9, we compare sorting the variables in ascending and descending order w.r.t. the histogram score β. Variables occurring the most in the formula are called the *most constrained*. Judging from these experiments, the following optimal configuration is concluded: scaleFunction(*nlogn*), *rounds*(3), *mdmStep*(2,000), pumpFrozen(*VMTF-only*), mdmrank(*most-constrained*), WalkSAT(*enabled*).

Figure 6.11 provides an assessment of our solver ParaFROST with(out) MDM compared to others. Clearly, our solver with MDM enabled was the fastest among all, solving 6 more problems than the version with MDM disabled. The main reason of MDM outperforming the others is that it tries to assign and propagate as many decisions with better phases improved by WalkSAT. We expect that, given more time, MDM will allow ParaFROST to solve even

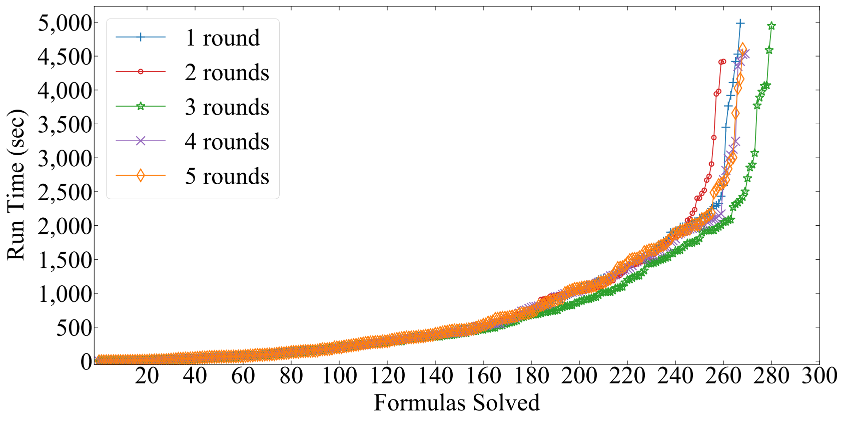


Figure 6.6: Number of MDM *rounds* (SCALEFUNCTION = *nlogn*)

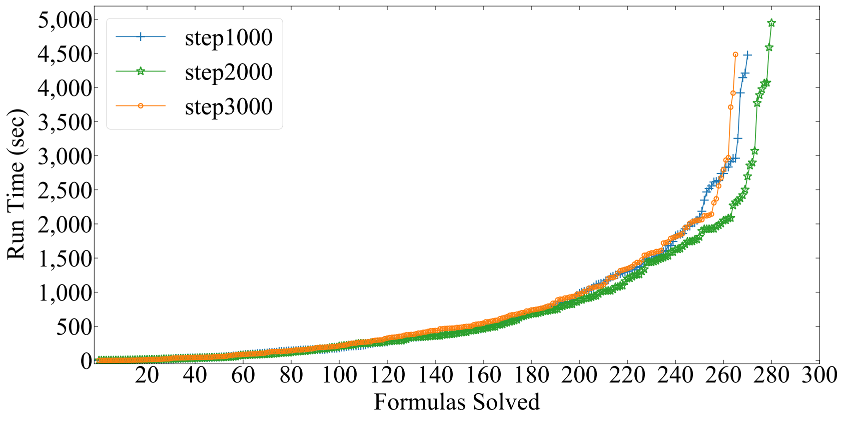


Figure 6.7: MDM steps (SCALEFUNCTION = *nlogn*)

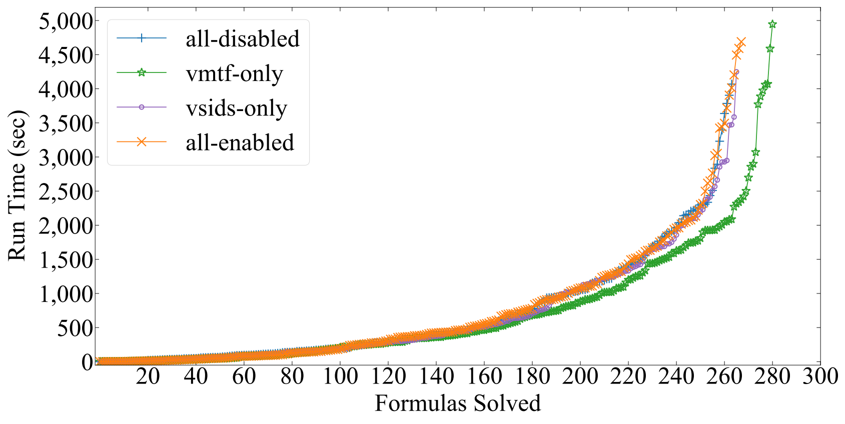


Figure 6.8: MDM prioritisation ($mdmStep$ = 2,000)

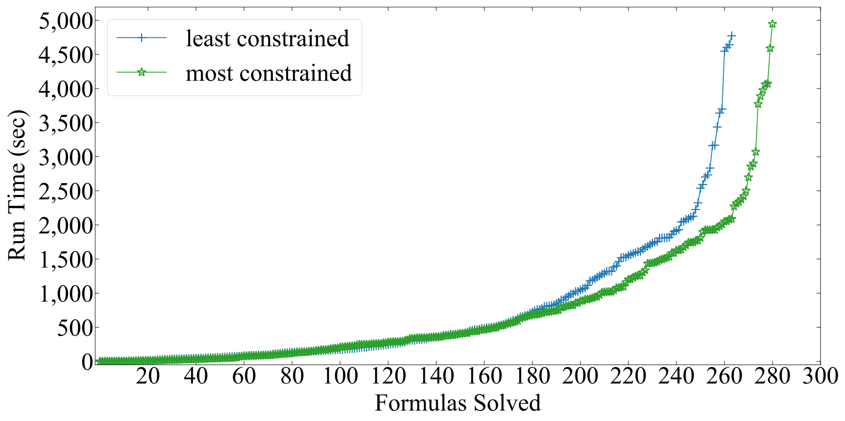


Figure 6.9: MDM ranking with histogram score β

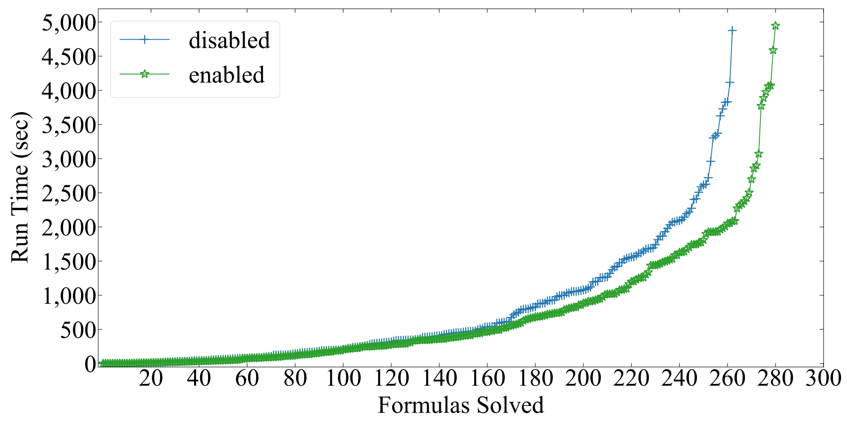


Figure 6.10: MDM with local search WalkSAT

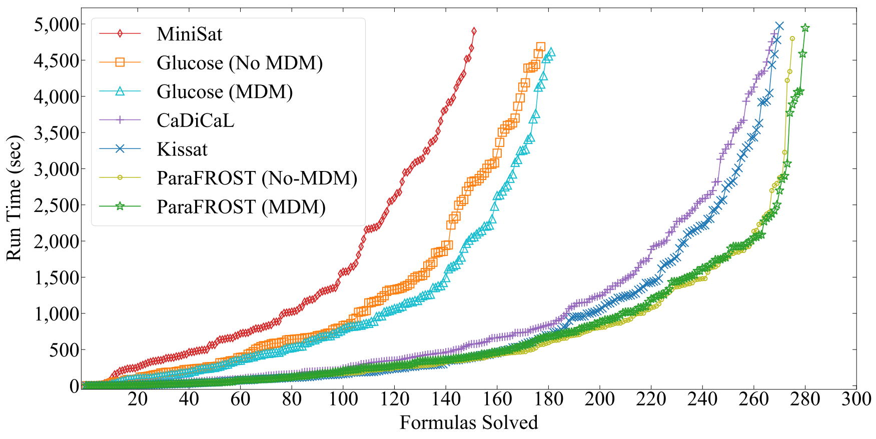


Figure 6.11: Different solvers comparison

harder problems that cannot be solved with ParaFROST without MDM or other solvers.

To get better verdict on the performance of these solvers, we report in

Table 6.1: Solvability evaluation of all solvers

Solver	**SAT**	**UNSAT**	**All Solved**	**PAR-2 Score**
MiniSat	79	73	152	6,742.04
Glucose (No-MDM)	80	98	178	6,078.43
Glucose (MDM)	79	**103**	**182**	**5,919.35**
CaDiCaL	143	126	269	3,897.14
Kissat	144	127	271	3,757.83
ParaFROST (No-MDM)	151	125	276	3,531.95
ParaFROST (MDM)	**157**	124	**281**	**3,473.7**

Table 6.1, the number of solved formulas and the *Penalized Average Runtime-2* (PAR-2) metric used by the SAT competitions to determine the winners. PAR-2 score accumulates the running times of all solved instances with 2× the time-out of unsolved ones, divided by the total number of formulas. The solver with the lowest score is the winner. The *SAT* and *UNSAT* columns give the number of satisfiable and unsatisfiable instances per solver respectively. As expected, ParaFROST (MDM) is the winner achieving a minimum score of 3,473.7, solving more *SAT* instances in less time, while being competitive with the *UNSAT* ones. The impact of MDM on Glucose can be also noticed from the third row, outperforming Glucose (No-MDM) with 182 cases solved in total and a score of 5,919.35.

Tables 6.2 and 6.3 show the performance of the MDM procedure on solving a sample of 50 formulas using ParaFROST and Glucose, respectively. SDs and MDs denote the Single and Multiple Decisions issued by the singleDecision and MDM procedures, respectively. The first 40 entries in Table 6.2 give a sample of 40 formulas that are solved the fastest when MDM is enabled. For instance, the formula *170222843* was solved by ParaFROST with MDM in just 0.24 seconds; while it took 16 seconds to solve for ParaFROST without MDM. The last 10 rows show some cases solved slower when MDM is applied. In general, the running times of these cases were slightly increased, concluding that, applying MDM can further strengthen the solver and hardly weaken it (see also Fig. 6.11). Similarly, the effect of MDM on Glucose was noticeable as concluded in [OW20]. For example, the *mp1-9_49* case was solved by Glucose with MDM in 101.63 seconds while it took 3,548.35 seconds to solve by Glucose without MDM.

Table 6.2: MDM impact on 50 formulas solved by ParaFROST

CNF	ParaFROST (No MDM)			ParaFROST (MDM)				
	Conflicts	SDs	time (s)	Conflicts	SDs	MDs	Calls	time (s)
170222843	504,662	1,035,854	16.64	6,009	11,324	173	3	**0.24**
vlsat2_16676_1598591.dimacs	1,968,342	2,624,200	342.19	33,981	75,105	23,206	3	**9.05**
preimage_80r_494m_160h_seed_378	380,526	855,725	625.81	30,120	121,938	11,281	3	**20.55**
170223547	8,897,575	16,292,047	557.69	749,466	1,475,447	391	18	**27.64**
combined-crypto1-wff-seed-102	2,125,451	4,221,042	90.78	163,410	355,941	3,076	9	**4.66**
DLTM_twitter774_83_17	3,493,705	9,319,802	838.25	196,632	660,229	14,356	6	**43.96**
abw-N-bcsstk07.mtx-w35	11,888,924	152,967,705	5,194.31	655,083	7,659,099	1,072,953	9	**338.69**
4g_6color_366_060_06.cnf.xz	**Time out** $(>5,000)$			6,344,930	14,810,896	113,486	24	**566.88**
ps_300_312_20	18,907	181,632	16.68	463	13,915	173,439	3	**2.01**
combined-crypto1-wff-seed-8	1,811,923	3,447,883	77.81	291,283	603,515	3,435	9	**10.58**
TT7F-33-24E.cnf.xz	**Time out** $(>5,000)$			30,123,269	238,106,877	16,619	63	**792.57**
8-5-6.cnf.xz	**Time out** $(>5,000)$			3,536,204	13,729,557	1,076	1	**811.77**
DLTM_twitter249_74_11.cnf.xz	**Time out** $(>5,000)$			13,062,732	27,277,182	16,385	6	**855.54**
Steiner-135-32-bce.cnf.xz	**Time out** $(>5,000)$			12,909,251	25,522,252	32,849	4	**857.69**
combined-crypto1-wff-seed-132	1,727,806	3,402,736	70.23	410,368	864,398	3,782	12	**13.45**
apn-sbox6-cut4-helpbox26.cnf.xz	**Time out** $(>5,000)$			73,263,705	107,014,205	279,896	12	**1,013.66**
sgp_5-6-8.sat05-2669.reshuffled-07	9,606,270	67,109,899	1,479.77	2,032,331	15,159,460	70,825	18	**302.65**
combined-crypto1-wff-seed-110	15,335,966	28,521,515	1,327.54	5,387,541	10,490,932	8,355	33	**282.33**
size_5_5_5_i235_r12	21,841,588	57,667,581	1,748.71	5,674,577	16,266,784	5,491	33	**378.46**
fsf-300-354-2-2-3-2.23.opt	3,750,521	11,571,976	206.26	889,192	4,546,579	15,769	18	**44.99**
6g_6color_366_050_04	1,798,311	4,736,005	2,132.61	3,390,465	8,806,040	183,404	18	**475.77**
4g_6color_182_100_02.cnf.xz	**Time out** $(>5,000)$			89,112	333,289	162,951	3	**1,127.60**
DLTM_twitter454_70_12	3,987,396	8,262,208	765.74	1,030,136	2,498,033	19,661	15	**183.58**
ps_300_301_30	820,773	4,434,767	334.68	185,951	648,581	337,853	6	**82.33**
mp1-9_49	655,455	1,295,215	38.16	218,926	532,416	557	4	**9.54**
4g_5color_170_050_05	806,173	1,729,731	427.79	218,454	470,960	16,144	6	**107.88**
170153306	2,289,868	4,309,014	96.63	705,674	1,341,365	383	18	**25.51**
abw-N-bcsstk07.mtx-w44	478,203	14,327,127	284.35	168,483	169,987	544,104	6	**75.10**
homer17.shuffled	52,948,584	62,009,273	1,916.45	87,372,177	101,586,730	1,267	111	**519.14**
size_5_5_5_i041_r12	10,723,084	29,850,870	907.04	4,119,234	12,636,126	4,899	30	**255.36**
combined-crypto1-wff-seed-18	11,408,381	20,680,350	864.15	4,962,583	9,220,805	9,379	33	**254.20**
dislog_a11_x11_n21	728,565	1,169,711	64.01	318,324	495,495	10,125	9	**18.97**
abw-X-can___715.mtx-w103	1,763,005	68,653,155	2,210.29	405,495	16,424,401	1,944,167	6	**735.30**
170105432	4,536,129	8,210,612	221.23	1,778,501	3,432,729	507	27	**73.92**
abw-R-dwt___503.mtx-w64.cnf.xz	**Time out** $(>5,000)$			3,010,055	29,229,440	3,025,747	18	**1,736.13**
ps_300_314_20	20,833	241,507	17.91	3,965	29,012	173,662	3	**6.30**
full-bg-gb-9-ce	27,533,241	75,377,997	989.50	4,096,329	12,207,019	2,094	3	**357.07**
ssAES_4-4-8_round_8-10_faultAt_8	**Time out** $(>5,000)$			12,467,042	24,828,849	140,698	21	**1,957.68**
6g_5color_164_100_01	76,411	267,237	1,336.40	10,692	50,231	92,638	3	**542.04**
schur-triples-7-90	3,952,462	25,553,268	454.16	1,808,727	13,343,916	179,483	18	**186.84**
01-integer-programming-20-30-40	1,110,522	1,891,914	911.89	2,185,080	3,839,021	199,215	2	1,689.43
22930-0426-195.smt2	12,070,787	20,344,613	1,100.91	9,385,868	16,272,702	12,314	45	1,896.79
4g_5color_166_100_02	10,189	40,701	114.90	24,574	75,796	96,059	3	552.62
abw-K-dwt___234.mtx-w51	27,387,989	110,582,547	810.18	**Time out** $(>5,000)$				
beempgsol5b1	1,449,242	4,232,973	176.75	1,883,976	5,161,022	4,089	4	222.83
bv-term-small-rw_503.smt2	33,022,570	73,858,349	327.41	**Time out** $(>5,000)$				
DLTM_twitter454_70_11	2,307,948	4,761,784	419.65	2,458,861	4,837,849	28,167	24	462.19
combined-crypto1-wff-seed-3	2,407,675	4,473,036	107.34	3,351,731	6,493,030	4,820	16	173.29
fermat-33106286870663	666,993	1,583,875	108.15	737,781	1,688,943	4,761	3	123.71
grain-53-80-0s0-seed-125-8-init-35	5,313,551	5,902,105	286.31	5,336,097	5,984,423	12,193	33	288.23

Table 6.3: MDM impact on 50 formulas solved by Glucose

CNF	**Glucose (No MDM)**			**Glucose (MDM)**				
	Conflicts	**SDs**	**time (s)**	**Conflicts**	**SDs**	**MDs**	**Calls**	**time (s)**
newpol4-6.cnf.xz	**Time out** (> 5,000)			1,259,963	1,591,006	16,979	5	**96.50**
newpol6-6.cnf.xz	**Time out** (> 5,000)			1,366,550	1,780,307	40,177	3	**105.58**
baseballcover11with25_and5positions	**Time out** (> 5,000)			745,842	1,283,439	532,773	10	**142.80**
mp1-9_49	15,419,564	19,624,955	3,548.35	1,305,557	1,799,351	187	9	**101.63**
baseballcover11with22_and2positions	**Time out** (> 5,000)			943,260	1,530,188	555,938	8	**175.97**
ssp-0.497665446947731	1,163,493	2,421,665	264.06	98,357	474,306	31,514	3	**13.19**
Kakuro-easy-052-ext.xml.hg_4	2,206,749	6,157,186	319.35	215,421	768,129	15,080	3	**33.09**
fermat-907547022132073	2,594,852	3,522,728	664.42	601,075	1,005,121	7,206	16	**71.45**
preimage_80r_493m_160h_seed_457	1,762,301	2,485,535	2,907.10	290,727	358,608	29,534	2	**445.75**
Kakuro-easy-132-ext.xml.hg_9	5,022,043	26,392,224	2,830.91	791,792	5,420,750	352,154	27	**446.46**
w19-20.1	20,434,684	23,034,212	3,006.32	5,251,487	6,081,254	7,400	7	**598.97**
170059722.cnf.xz	**Time out** (> 5,000)			5,243,784	6,042,354	166	3	**1,037.43**
newpol36-4	310,606	680,820	795.88	102,548	117,399	3,599,907	3	**166.56**
DLTM_twitter799_70_13	7,918,249	9,867,058	1,705.96	1,929,147	2,492,989	74,079	41	**377.23**
baseballcover14with25_and2positions	344,776	744,357	3,597.29	3,892,792	5,163,659	556,382	31	**852.59**
sqrt_ineq_3.c	10,396,356	18,764,704	1,323.30	3,085,685	6,038,651	210,889	7	**333.17**
vlsat2_24450_2770239.dimacs	13,896,440	64,984,350	3,059.61	4,670,224	8,835,839	38,195	3	**814.65**
abw-N-bcsstk07.mtx-w44	2,846,905	3,308,703	448.87	907,753	967,175	186,638	1	**123.77**
ps_300_311_20	3,421,138	13,786,770	770.84	1,383,693	7,468,990	1,196,205	17	**251.23**
preimage_80r_493m_160h_seed_249	907,686	1,190,677	1,462.66	473,571	835,079	30,266	2	**483.56**
DLTM_twitter690_74_16	819,321	1,240,615	185.21	242,076	381,312	19,075	5	**65.56**
Kakuro-easy-127-ext.xml.hg_6	**Time out** (> 5,000)			6,201,935	21,931,146	225,415	21	**2,128.16**
Steiner-15-7-bce	240	265	0.005	259	291	115	2	**0.002**
baseballcover13with25_and1positions	1,981,777	2,980,432	316.99	738,214	1,185,580	170,291	6	**129.07**
bivium-40-200	12,884,734	13,397,611	2,573.73	6,291,734	6,654,291	237	1	**1,070.25**
baseballcover15with25	4,258,361	5,378,094	1,015.38	2,187,129	2,813,194	167,495	17	**425.45**
mod2c-rand3bip-sat-220-2.sat05-2489	4,394,710	4,574,944	630.81	2,557,081	2,692,069	160	3	**299.13**
sted5_0x0-157	11,016,120	13,617,051	2,829.15	6,296,215	7,761,100	3,477	16	**1,356.66**
Kakuro-easy-148-ext.xml.hg_8	764,142	4,891,733	395.99	325,877	2,553,440	95,255	5	**200.30**
dislog_a09_x11_n20	2,367,916	2,641,651	346.47	1,541,199	1,770,551	4,364	2	**179.81**
fclqcolor-20-15-12.cnf.gz.CP3-cnfmiter	649,787	91,921,398	384.54	327,439	69,204,636	132,960	3	**199.57**
Timetable_C_392_E_62_Cl_26_S_28	27,513	7,781,777	13.68	9,276	3,876,955	335,922	3	**7.10**
DLTM_twitter845_79_19	930,074	1,411,441	195.76	440,274	665,285	12,433	3	**103.78**
preimage_80r_492m_160h_seed_136	840,880	1,218,805	1,327.08	487,572	695,584	27,313	2	**747.17**
vlsat2_16676_1598591.dimacs	8,257,673	33,496,638	1,493.49	5,166,854	9,215,253	22,364	3	**860.52**
baseballcover12with23_and2positions	**Time out** (> 5,000)			13,881,635	18,851,965	555,008	14	**2,925.81**
ncc_none_2_19_5_3_1_0_435991723	517,246	90,231,345	4,686.26	286,245	69,519,894	7,074,352	3	**2,772.90**
filter_iir_true-unreach-call.c	1,762,715	8,479,414	638.63	962,585	5,519,255	3,375,440	39	**398.00**
Kittell-k7	1,511,139	3,791,884	176.26	1,155,773	3,143,522	25,295	11	**131.19**
009-80-8	11,711,355	189,837,650	586.23	8,702,483	136,478,414	129,700	109	**527.10**
Timetable_C_392_E_50_Cl_26_S_28	1,586,245	62,875,574	157.97	819,385	49,976,239	6,050,393	87	**106.48**
g2-T83.2.1	481,489	5,619,601	1,163.91	449,183	4,377,464	8,163,020	19	**786.87**
3bitadd_32.cnf.gz.CP3-cnfmiter	3,783,742	405,135,211	1,219.83	3,438,441	392,086,335	2,748,120	71	**1,099.77**
sgen4-unsat-89-1	3,721,064	4,250,647	224.49	9,493,297	10,798,826	73	13	1,030.22
ncc_none_2_18_9_3_0_0_435991723	811,274	24,585,680	1,516.02	653,181	22,520,190	3,105,523	5	**1,378.37**
course0.2_2018_3-sc2018	6,536,769	45,951,875	5,015.21	4,888,468	35,868,009	2,572,504	24	**4,284.58**
stable-300-0.1-20-98765432130020	1,987,350	2,584,178	343.80	1,846,085	2,549,574	8	3	**323.50**
velev-pipe-uns-1.0-9	1,544,644	13,856,479	131.82	1,178,028	10,135,640	568,213	19	**109.90**
simon03:sat02bis:k2fix_gr_2pinvar	16,738,745	61,867,126	1,526.77	**Time out** (> 5,000)				
simon-mixed-s02bis-05	1,385,046	1,755,488	125.58	1,384,373	1,793,524	9,468	4	136.25

6.5 Related Work

The SAT competition has a major influence on SAT technology. Every year, new SAT solvers come to light, offering novel heuristics to the CDCL and simplifications. With this rapid advancement, it is almost impossible to cover every related work to ours. However, we will try to zoom into some of the most impactful research (relative point of view) in this field. Audemard *et al.* [AS09, AS12] have devised one of the most successful heuristics in CDCL history. The Glucose-level metric learnt clauses reduction and dynamic restarts are now essential in all state-of-the-art SAT solvers including ParaFROST, CaDiCaL, CryptoMiniSat, and more. Later on, Biere *et al.* [BF18] performed a complete evaluation of all restart mechanisms, paving the way for a hybrid method combining both the dynamic and geometric restarts.

Regarding the decision heuristics, Moskewicz *et al.* [MMZ+01] introduced VSIDS which was considered the first heuristic for variable selection based on its frequency in conflict analysis. Biere *et al.* [BF15] proposed the VMTF queue and a way to alternate between VMTF and VSIDS queues based on restarts. Habet *et al.* [HT19] proposed a new branching heuristic called Conflict-History Search (CHS) which is adopted to the history of search failures. Cherif *et al.* [CHT20] extended the former with the Multi-Arm Bandit (MAB) refinement. Similar to the work in [BF15], MAB tries to find a balance between VSIDS and CHB depending on the restarts.

6.6 Conclusion

We have extended the CDCL search with the ability to make multiple decisions at once with different decision queues and restart policies. Moreover, we have proposed a minimal local search via WalkSAT strategy to be combined with MDM, which proved to be effective, in particular, when solving satisfiable formulas compared to the state of the art. We have shown in-depth analysis of different MDM configurations and its impact on solving real-world SAT problems and have compared its performance through our new solver ParaFROST to the state of the art.

Concerning future work, we are motivated to implement and study the impact of more decision queues on MDM quality. Furthermore, the positive effect of WalkSAT on MDM opens the door to further investigate alternative evolutionary algorithms such as ant colony optimisation.

Chapter **7**

Bounded Model Checking

"SAT solving is a key technology for 21st century computer science."

– Edmund Clarke

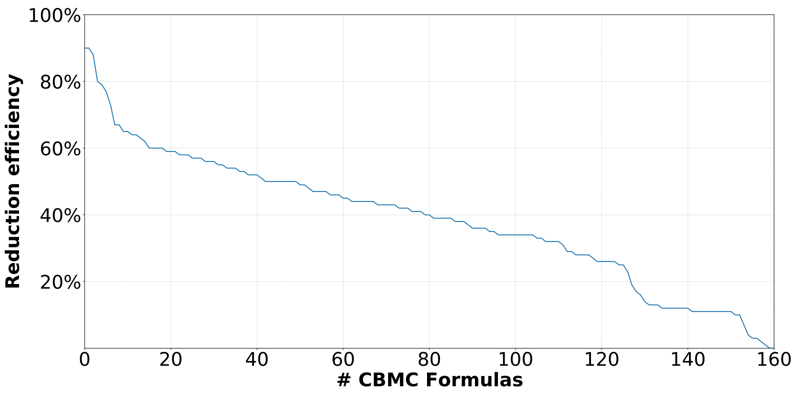
Bounded Model Checking (BMC) [BCCZ99] determines whether a model M satisfies a certain property φ expressed in temporal logic, by translating the model checking problem to a SAT or SMT problem. The term *bounded* refers to the fact that the BMC procedure searches for a counterexample to the property, i.e., an execution trace, which is bounded in length by an integer k. If no counterexample up to this length exists, k can be increased and BMC can be applied again. This process can continue until a counterexample has been found, a user-defined threshold has been reached, or it can be concluded (via k-induction [SSS00]) that increasing k further will not result in finding a counterexample. CBMC [CKL04] is an example of a successful BMC model checker that uses SAT solving. CBMC can check ANSI-C programs. The verification is performed by *unwinding* the loops in the program under verification a finite number of times, and checking whether the bounded executions of the program satisfy a particular safety property [KS16]. These properties may address common program errors, such as null-pointer exceptions and array out-of-bound accesses, and user-provided assertions.

The performance of BMC heavily relies on the performance of the solver. Over the last decade, efficient SAT solvers [SS99, ES03a, AS09, BFFH20] have been developed and applied for BMC [BCCZ99, BCMD90, Bro13, Bra11, SG99]. Nonetheless, effectively *parallelising* BMC through SAT solving is a challenging task. Parallel SAT solving often involves running several solvers, each exploring the problem in its own way [HJS09]. For BMC, multiple solvers can be used to solve the problem for different values of the bound k in parallel [ÁSB+11, IT20, KT11]. However, in these approaches, the individual solvers are still single-threaded.

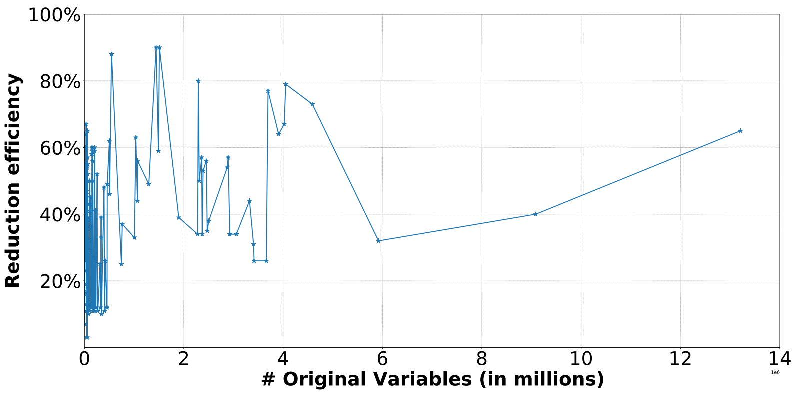
Recently, Leiserson *et al.* [LTE+20] concluded that in the future, advances in computational performance will come from *many-threaded* algorithms that can employ hardware with a massive number of processors. As previously discussed in Section 3.2, GPUs are an example of such hardware. Multi-threaded bounded model checkers have been proposed, such as in [IT20, CRDL20], but these address tens of threads running different tasks in parallel.

In this chapter, we propose the application of GPUs to accelerate SAT-based BMC. To the best of our knowledge, this is the first time this is being addressed. Recently, GPUs have been applied for explicit-state model checking and graph analysis [WB14, BESW10]. In SAT solving, we used GPUs to accelerate test pattern generation [OGHM18], metaheuristic search [YIMO15], *preprocessing* (Chapter 4) and *inprocessing* (Chapter 5). In preprocessing, this is only done once before the solving starts, while in inprocessing, this is done periodically during the solving. Although, the impact of accelerating these procedures has been demonstrated in Chapters 4 and 5, its impact on BMC has not yet been addressed.

The structure of BMC SAT formulas having millions of redundant variables and clauses suggests that GPU pre- and inprocessing will be effective. Figure 7.1a shows for a BMC benchmark set taken from the Core C99 package of AWS [Ama21], consisting of 168 problems of various data structures, that propositional formulas produced by CBMC tend to have a substantial amount of redundant variables that can be removed using simplification procedures. For approximately 50% of the cases, 40% of the variables can be removed. Furthermore, Figure 7.1b presents the amount of redundancy in relation to the total number of variables in the formula. It indicates that when a formula contains one million variables or more, at least 25% of those are redundant, and often many more. In the benchmark set, the maximum number of variables in one formula is 13 million (encoding the verification of the `priority-queue shift-down` routine), of which 65% is redundant. In contrast, the largest formula we encountered in the application track of the 2013-2020 SAT competitions that is not encoding a verification problem only has 0.2 million variables (it encodes a graph coloring problem [OMH20]).



(a) The amount of reductions in CBMC formulas



(b) The amount of reductions w.r.t. the number of variables

Figure 7.1: Variable redundancy in CBMC SAT formulas

Contributions

We present the SAT solver ParaFROST that applies Conflict Driven Clause Learning (CDCL) [SS99] with GPU acceleration of pre- and inprocessing [OW19a, OW19b, OWB21b], tuned for BMC. It has been implemented in CUDA C++ v11 [NVI20a], is based on CaDiCaL [BFFH20], and interfaces with CBMC.

Having to deal on a GPU with large formulas with a lot of redundancy offers particular challenges. The elimination of variables typically leads to actually adding new clauses, and since the amount of memory on a GPU is limited, this cannot be done carelessly. Therefore, first of all, we have worked on compacting the data structure used to store formula clauses in ParaFROST as much as possible, while still allowing for the application of effective solving optimisations. Second of all, we introduce *memory-aware* variable elimination, to avoid running out of memory due to adding too many new clauses. In practice, we experienced this problem when applying the original procedure of Chapter 5 for BMC.

Additionally, to support BMC, ParaFROST must be an *incremental* solver, i.e., it must exploit that a number of very similar SAT problems are solved in sequence [ES03b]. The procedure in Chapter 5 does not support this, so we extended it.

Finally, because of the many variables in BMC SAT formulas, ParaFROST supports MDM in the solving procedure, as presented in Chapter 6. With MDM, multiple decisions can be made at once, periodically during the solving. In case there are many variables, there is more potential to make many decisions simultaneously. The effectiveness of MDM in BMC has never been investigated before, nor MDM has been combined with GPU pre- and inprocessing.

7.1 Incremental Bounded Model Checking

Since 2001, incremental BMC has been applied to hardware and software verification [Str01, ES03b, JS05, SKB+17]. It relies on incremental SAT solving [ES03b, WKS01]. In CDCL, clauses are learnt during the solving each time a wrong decision has been made, to avoid making those decisions again in the future. Incremental SAT solving builds on this: when multiple SAT formulas with similar characteristics are solved sequentially, then in each iteration, the clauses learnt in previous iterations are reused. The same idea can be applied on a single large formula by incremental decomposition into smaller subformulae where the next in line is monotonically increased in size. To disable the influence of subsuming clauses in smaller subformulae on the solving outcome of the current subformula, these clauses should be removed. An efficient approach to add (*enable*) and remove (*disable*) clauses is by using *assumptions* [ES03b], which are initial assignments.

Given a transition relation M representing the system design, a temporal logic formula φ to verify, and an upper-bound k, the bounded model checker generates the SAT formula $[\![M,\varphi]\!]_k$. This formula is satisfiable iff the property φ is **true** for some finite trace $\pi = s_0, s_1, \ldots, s_k$ in M. Every state s_i in that path represents a vector of Boolean variables. Before the model is verified in BMC, the SAT formula $[\![M]\!]_k$ is constructed such that π is a valid trace in M.

Definition 7.1 (Transition relation in SAT)**.** Given the transition relation M and the bound k, the SAT formula encoding M is given as follows:

$$[\![M]\!]_k = \mathcal{I}(s_0) \wedge \bigwedge_{i=0}^{k-1} \tau(s_i, s_{i+1})$$

where the parameter:

- $\mathcal{I}(s_0)$ is a predicate identifying the set of initial states.
- $\tau(s_i, s_{i+1})$ encodes the transition relation at trace depth i.

In practice, the conjunction of $[\![M]\!]_k$ and $[\![\neg\varphi]\!]_k$ (the negation of the property) is actually checked in BMC. If the outcome is *satisfiable*, a counterexample π is generated, pointing to the states violating the property. Let $\mathcal{E}(i) = \bigvee_{0\leq j\leq i} e(s_j)$ be the predicate encoding the error states at trace depth i. The predicate $e(s_j)$ is **true** iff s_j is an error state.

For incremental BMC, additional unit clauses θ_i are used. These predicates are combined to define the following series of SAT formulas $\mathcal{S}(i)$ that must be solved incrementally:

$$\begin{aligned} \mathcal{S}(0) &= [\![M,\neg\varphi]\!]_0 = \mathcal{I}(s_0) \wedge (\mathcal{E}(0) \vee \theta_0) \\ \mathcal{S}(i+1) &= [\![M,\neg\varphi]\!]_{i+1} = \mathcal{S}(i) \wedge \tau(s_i, s_{i+1}) \wedge \theta_i \wedge (\mathcal{E}(i+1) \vee \theta_{i+1}) \end{aligned} \tag{7.1}$$

where $\mathcal{S}(0)$ and $\mathcal{S}(i+1)$ are under the assumptions $\neg\theta_0$ and $\neg\theta_{i+1}$ respectively. Formula $\mathcal{S}(i)$ is satisfiable iff an error state is reachable via a trace with a length up to i [Str01, ES03b]. At iteration $i+1$, we know that $\mathcal{E}(i)$, included via $\mathcal{S}(i)$, cannot be satisfied (otherwise iteration $i+1$ would not have been started). This means that $\mathcal{E}(i)$ must be removed to avoid that $\mathcal{S}(i+1)$ is unsatisfiable. To effectively remove $\mathcal{E}(i)$, θ_i is assigned **true**, resulting in $\mathcal{E}(i) \vee \theta_i$ being satisfied. In general, at iteration i, θ_i is assigned **false**, while in iterations $i' > i$, it is assigned **true**.

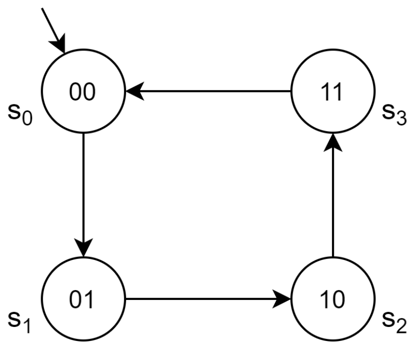


Figure 7.2: A 2-bit counter transition system

Example 7.1. Consider the 2-bit counter given by the transition system in Figure 7.2. Assume the least and most significant bits are being represented by the Boolean variables a and b respectively. The transition relation ($s_i \rightarrow s_{i+1}$) can then be expressed as (with $\otimes$ being the *xor* operator)

$$\tau(s_i, s_{i+1}) = (a_{i+1} \leftrightarrow \neg a_i) \wedge (b_{i+1} \leftrightarrow a_i \otimes b_i)$$

Let the initial state be $\mathcal{I}(s_0) = (\neg a_0 \wedge \neg b_0)$. Suppose we wish to check the safety property denoted by the CTL formula $\mathbf{AG}\varphi$ (i.e. φ is satisfiable by every reachable state) where φ is the state 11 (i.e. $a \wedge b$) being reachable from the initial state up to time step 3. According to Definition 7.1, the reachability check is achieved by unfolding both τ and $\mathcal{E}$ up to depth $k = 3$ as follows

$$\mathcal{S}(0) = \underbrace{(\neg a_0 \wedge \neg b_0)}_{\mathcal{I}(s_0)} \wedge \underbrace{\neg(a_0 \wedge b_0)}_{e(s_0):\neg\varphi(s_0)} = \neg a_0 \wedge \neg b_0 \wedge (\neg a_0 \vee \neg b_0) = \textbf{true}$$

$$\mathcal{S}(1) = \mathcal{S}(0) \wedge \underbrace{((a_1 \leftrightarrow \neg a_0) \wedge (b_1 \leftrightarrow a_0 \otimes b_0))}_{\tau(s_0,s_1)} \wedge \underbrace{(\neg a_0 \vee \neg b_0 \vee \neg a_1 \vee \neg b_1)}_{(e(s_0)\ \vee\ e(s_1))}$$

$$\wedge \underbrace{a_0 \ \wedge \ b_0}_{\theta_0\ =\ \neg e(s_0)} = \textbf{true}$$

$$\mathcal{S}(2) = \mathcal{S}(1) \wedge \tau(s_0, s_1) \wedge \tau(s_1, s_2) \wedge \underbrace{(\neg a_0 \vee \neg b_0 \vee \neg a_1 \vee \neg b_1 \vee \neg a_2 \vee \neg b_2)}_{(e(s_0) \,\vee\, e(s_1) \,\vee\, e(s_2))}$$

$$\wedge\ a_0\ \wedge\ b_0\ \wedge\ \underbrace{a_1\ \wedge\ b_1}_{\theta_1 \,=\, \neg e(s_1)}\ = \textbf{true}$$

$$\mathcal{S}(3) = \mathcal{S}(2) \wedge \tau(s_0, s_1) \wedge \tau(s_1, s_2) \wedge \tau(s_2, s_3) \wedge a_0 \wedge b_0 \wedge a_1 \wedge b_1 \wedge \underbrace{a_2\ \wedge\ b_2}_{\theta_2 \,=\, \neg e(s_2)}$$

$$\wedge \underbrace{(\neg a_0 \vee \neg b_0 \vee \neg a_1 \vee \neg b_1 \vee \neg a_2 \vee \neg b_2 \vee \neg a_3 \vee \neg b_3)}_{(e(s_0) \,\vee\, e(s_1) \,\vee\, e(s_2) \,\vee\, e(s_3))} = \textbf{false}$$

Clearly, for formulas up to and including $\mathcal{S}(2)$, the state s_3 is not reachable as they are satisfiable by the error states $\bigvee_{0 \leq j \leq 2} e(s_j)$. Therefore, the assumptions $\bigwedge_{0 \leq j \leq 2} \neg e(s_j)$ were added incrementally to $\mathcal{S}(3)$. At depth 3, this formula becomes *unsatisfiable* due to e_3 is being falsified which implies that s_3 is indeed reachable.

7.1.1 Incremental SAT Solving

Algorithm 7.1 extends the generalised CDCL search in Algorithm 6.1 with incremental solving. This extension takes, in addition, the assumption θ encoded as a set of literals. Before the CDCL procedure attempts to select a new decision, first it checks if at least one assignment in θ is available to pick in the loop at lines 11-22. In case ℓ is unassigned (line 12), it is added to L as a new decision with a new level. If it is already satisfied (assigned $\top$), only the current level is advanced (line 16) so that if a conflict occurs, the ANALYSE procedures can use the correct conflict level. Otherwise, ℓ is conflicting and the IANALYSE procedure is called. The only difference compared to ANALYSE (see Algorithm 6.1) is that IANALYSE tries to find all the antecedents of ℓ that are assigned as decisions. The negations of these assignments are added to $\hat{C}$ at line 18 and all assignments are backtracked to the initial decision level (line 19). The current search is terminated by returning the learnt clause $\hat{C}$. This clause is particularly useful for bounded model checkers to avoid augmenting the next incremental solving with the failed assumption [ES03b].

The IANALYSE procedure, given at lines 29-41, takes the current failing assignment ℓ as input. Initially, at line 30, the negation of ℓ is added to $\hat{C}$ and the `antecedents` set. The loop at lines 31-39 iterates over all assignments in `antecedents` . At lines 32-33, the first literal $\ell' \in$ `antecedents` is picked and then removed from `antecedents` . If that literal is an implication (line 34), then

Algorithm 7.1: Incremental CDCL Solving

```
Input   : assumption $\theta$, assignments $\sigma$, level function $\delta$, implication function $source$
procedure CDCL():
    $d \leftarrow 0$, $\sigma \leftarrow \emptyset$, $\sigma' \leftarrow \emptyset$, $\delta \leftarrow \emptyset$, $sat \leftarrow$ UNSOLVED
    while $sat =$ UNSOLVED do
        $C' =$ BCP()
        if $C' \neq \emptyset$ then
            if $d = 0$ then $sat \leftarrow$ UNSAT, break
            $(\hat{C}, \overleftarrow{d}) \leftarrow$ ANALYSE$(C', d)$, $\mathcal{S} \leftarrow \mathcal{S} \cup \hat{C}$
            BACKJUMP$(\overleftarrow{d})$, $d \leftarrow \overleftarrow{d}$
        end
        else
            forall $\ell \in \theta$ do
                if $\ell \models_\sigma \uparrow$ then
                    $L \leftarrow L \cup \{\ell\}$, $d \leftarrow d + 1$
                    $\delta(\ell) \leftarrow d$, $\delta(\neg\ell) \leftarrow d$
                    break
                else if $\ell \models_\sigma \top$ then $d \leftarrow d + 1$
                else
                    $\hat{C} \leftarrow$ IANALYSE$(\ell)$
                    BACKJUMP$(0)$, $d \leftarrow 0$
                    return $\hat{C}$
                end
            end
            if $L = \emptyset$ then $L \leftarrow$ DECIDE(), $d \leftarrow d + |L|$
            if $L = \emptyset$ then $sat \leftarrow$ SAT
            $\sigma \leftarrow \sigma \cup L$
        end
    end
end
procedure IANALYSE$(\ell)$:
    antecedents $\leftarrow \neg\ell$, $\hat{C} \leftarrow \hat{C} \cup \{\neg\ell\}$
    while antecedents $\neq \emptyset$ do
        pick first literal $\ell' \in$ antecedents
        antecedents $\leftarrow$ antecedents $\setminus \{\ell'\}$
        if $\exists (C \in \mathcal{S}).(\ell', C) \in source$ then
            antecedents $\leftarrow$ antecedents $\cup \{\ell'' \in C \mid \ell'' \neq \ell'\}$
        else
            $\hat{C} \leftarrow \hat{C} \cup \{\ell'\}$
        end
    end
    return $\hat{C}$
end
```

all literals in its source except ℓ' itself are added to `antecedents` . Otherwise, ℓ' must be a decision, hence it is added to $\hat{C}$ (line 37). The loop terminates if there are no more assignments to analyse in `antecedents` .

7.1.2 MDM in Incremental Solving

Given the fact that BMC SAT formulas often have many variables, MDM has much potential to speed up BMC. Recall that MDM periodically assigns and propagates multiple decisions at the same time. In the same spirit during incremental solving, a set of multiple assumed-decisions can be defined as follows:

Definition 7.2 (Multiple assumed-decisions set)**.** Given a formula $\mathcal{S}$, a set of assumptions θ, and a set of assignments σ, we call a set $\mathcal{M}_\theta = \{\ell \in \theta \mid \ell \models_\sigma \uparrow\}$ with $|\mathcal{M}_\theta| > 1$ a set of *multiple assumed-decisions* iff $\{C \in \mathcal{S} \mid |\mathit{Free}_{\sigma \cup \mathcal{M}_\theta}(C)| = 1\} = \emptyset$.

When the MDM method is called, the set of multiple decisions $\mathcal{M}$ defined in Chapter 6 can be extended to include $\mathcal{M}_\theta$ such that $\mathcal{M} = \mathcal{M}_\theta \cup \mathcal{M}_L$, where $\mathcal{M}_L = \{\ell \in (\mathbb{L} \backslash \theta) \mid \ell \models_\sigma \uparrow\}$ is the set of multiple decisions excluding assumptions.

7.2 GPU-Accelerated Bounded Model Checking

The proposed extensions in this chapter are implemented in ParaFROST with CUDA C++ v11. It is a hybrid CPU-GPU tool, with (sequential) solving done on the host side, and (parallel) VCE done on the device side. An interface with CBMC is implemented in C++. CBMC is patched to read a configuration file before ParaFROST is instantiated. This file contains all options supported by ParaFROST. The framework GPU4BMC[18] offers both the interface and the former patches as an open source.

7.2.1 The Workflow

Figure 7.3 presents the general workflow of ParaFROST in the form of an activity diagram with host and device lanes. The diagram is focused on inprocessing; preprocessing works similarly on the device. First, the host performs a predetermined number of solving iterations. Once those have finished, and (un)satisfiability has not yet been proven, relevant clause data is copied to the global memory. To hide the latency of this operation as much as possible,

[18]The framework is available at `https://gears.win.tue.nl/software/gpu4bmc`.

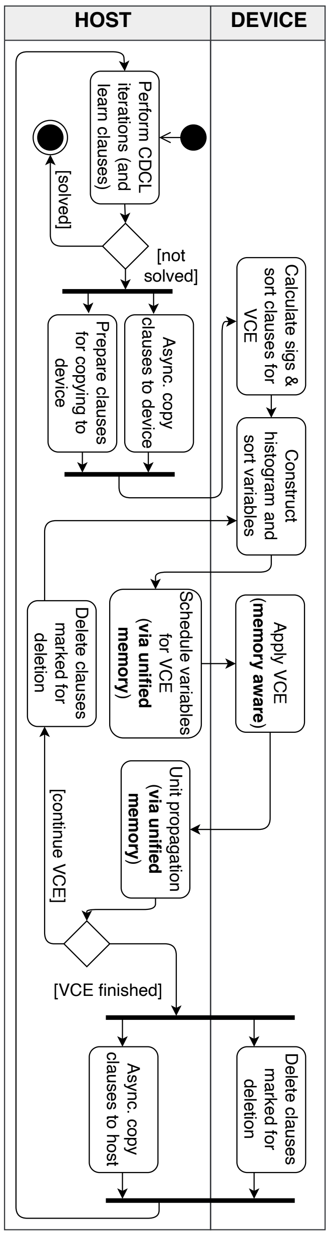

HOST
DEVICE
Perform CDCL iterations (and learn clauses)
[solved]
[not solved]
Async. copy clauses to device
Prepare clauses for copying to device
Calculate sigs & sort clauses for VCE
Construct histogram and sort variables
Schedule variables for VCE (via unified memory)
Delete clauses marked for deletion
Apply VCE (memory aware)
Unit propagation (via unified memory)
[continue VCE]
[VCE finished]
Async. copy clauses to host
Delete clauses marked for deletion

Figure 7.3: An activity diagram for the workflow of ParaFROST

clauses are copied asynchronously in batches. One batch is copied while the next is formatted for the GPU, as not all clause information on the host side is relevant for the device (see the next paragraph on data structures). On the device, signatures are computed for fast clause comparison, and the clauses are sorted for *variable-clause eliminations* (more on VCE later). Next, the device constructs a histogram, for fast lookup of clauses, and sorts the variables. The THRUST library is used for sorting.[19] After that, the host *schedules variables* for VCE, marking those variables in the global memory using unified memory. Next, the device applies VCE, marking clauses to be removed as `DELETED`. The host propagates units (literals in unit clauses are assigned **true**), which directly has an effect on the formula in the global memory. The VCE procedure is repeated until it has been performed a predetermined number of times. After each time, `DELETED` clauses are removed, and after the last iteration, this is done while the new clauses are copied to the host. Once this has been done, the overall procedure is repeated.

7.2.2 Data Structures and Memory Management

We have worked on making the storage of each clause in the GPU global memory as efficient as possible. However, we also wanted to annotate each clause with sufficient information for effective optimisations. In PARAFROST, the following information is stored for each clause:

- The `state` field (*2 bits*) stores if the state is `ORIGINAL`, `LEARNT` or `DELETED`.
- The `used` field (*2 bits*) keeps track of how many search iterations a `LEARNT` clause can still be used. `LEARNT` clauses are used at most twice [BFFH20].
- Two fields (*1 bit each*) are used for VCE bookmarking.
- The *literal block distance* (`lbd`) (*26 bits*) stores the number of decision levels contributing to a conflict, if there is one [AS09]. A maximum value of 2^{26} turns out to be sufficient. This field is updated when the clause is altered.
- The `size` (*32 bits*) of the clause, i.e., the number of literals.
- A signature `sig` (*32 bits*) is a clause hash, for fast clause comparison [EB05].

In addition, a list of literals is stored, each literal taking 32 bits (1 bit to indicate whether it is negated or not, and 31 bits to identify the variable). In total, a clause requires $12 + 4t$ bytes, with t the number of literals in the clause. For comparison, MINISAT only requires $4 + 4t$ bytes, but it does not involve the `used`, `lbd` and `sig` fields, thereby not supporting the associated optimisations.

[19] https://docs.nvidia.com/cuda/thrust.

$$
\begin{array}{rlll}
\text{RES:} & \{x\} \cup C_1, \{\bar{x}\} \cup C_2 & \rightarrow C_1 \cup C_2 & (\{x\} \cup C_1 \notin \mathcal{L} \wedge \{\bar{x}\} \cup C_2 \notin \mathcal{L}) \\
\text{SUB1:} & \{x\} \cup C_1 \cup C_2, \{\bar{x}\} \cup C_2 & \rightarrow C_1 \cup C_2, \{\bar{x}\} \cup C_2 & \\
\text{SUB2:} & C_1 \cup C_2, C_2 & \rightarrow C_2 & (C_2 \in \mathcal{L} \rightarrow \mathcal{L}' = \mathcal{L} \setminus \{C_2\}) \\
\text{ERE:} & \{x\} \cup C_1, \{\bar{x}\} \cup C_2, C_1 \cup C_2 & \rightarrow \{x\} \cup C_1, \{\bar{x}\} \cup C_2 & (\{\{x\} \cup C_1, \{\bar{x}\} \cup C_2\} \cap \\
 & & & \mathcal{L} \neq \emptyset \rightarrow C_1 \cup C_2 \in \mathcal{L})
\end{array}
$$

Figure 7.4: VCE rules in ParaFROST

CaDiCaL [BFFH20] uses $28 + 4t$ bytes, since it applies solving and VCE on the same structures. In ParaFROST, the GPU is only used for VCE, in which information for *probing* [LS03] and *vivification* [PHS08], for instance, is irrelevant. Finally, in [OWB21b], $20 + 4t$ bytes are used, storing the same information as ParaFROST.

To store a formula $\mathcal{S}$, a clause array is preallocated in the global memory, and filled with the clauses of $\mathcal{S}$. More space is allocated than the size of $\mathcal{S}$, to allow the addition of clauses that result from VCE. As the amount of allocated space is the limiting factor for the addition of new clauses, we have developed a memory-aware VCE mechanism, which we explain later in the current section.

7.2.3 Parallel VCE

As previously mentioned in Chapters 4 and 5, ParaFROST supports the VCE rules: *substitution* (i.e., gate equivalence reasoning), *resolution* (RES), *subsumption elimination* (SUB) and *eager redundancy elimination* (ERE) [EB05, JHB12]. Substitution applies to patterns representing logical gates, and substitutes the involved variables with their gate definitions. ParaFROST supports ***AND/OR***, ***Inverter***, ***If Then Else*** and ***XOR***.

In Figure 7.4, we provide simple rewrite rules for SUB, RES and ERE. The reader can find more information on these rules with formal proofs in Chapters 4 and 5. If clauses exist of the form expressed by the left hand side of a rule, then the rule is applicable, and the involved clauses are replaced by the clauses (called *resolvents*) on the right hand side. Both clauses C_1 and C_2 are non-empty sets of literals. RES is applicable if there are two clauses of the form $x \cup C_1$ and $\bar{x} \cup C_2$, and applying it results in replacing those with a clause $C_1 \cup C_2$. SUB consists of two rules; the second is applied once the first is no longer applicable.

Conditions are given between parentheses. For RES, only `ORIGINAL` clauses are considered. Besides that, if $C_1 \cup C_2$ evaluates to **true**, it is actually not created. As LEARNT clauses are sometimes deleted during solving, SUB2 should only produce ORIGINAL clauses; if C_2 is LEARNT before applying the rule, it will become ORIGINAL ($\mathcal{L}'$ refers to the set of LEARNT clauses after the application). For ERE, LEARNT clauses cannot cause the deletion of an ORIGINAL clause.

VCE is applied in parallel by ParaFROST by scheduling sets of *mutually-independent* variables for analysis. Recall that two variables x and y are independent in a formula $\mathcal{S}$ iff $\mathcal{S}$ does not contain a clause containing literals that refer to both variables, i.e., $\mathcal{S}_x \cup \mathcal{S}_{\bar{x}}$ and $\mathcal{S}_y \cup \mathcal{S}_{\bar{y}}$ are disjoint. This ensures that two threads focusing on x and y, respectively, does not lead to data races. In incremental solving, variables referred to by assumptions must be excluded from VCE. Thus, Definition 4.1 is modified to allow the exclusion of θ from the set of authorised candidates as follows.

Definition 7.3 (Authorised candidates excl. θ)**.** Given a CNF formula $\mathcal{S}$, and a set of assumed variables $var(\theta)$, the set of *authorised candidates* excluding assumptions is defined as: $\mathcal{A} = \{x \mid x \notin var(\theta) \wedge (1 \leq h[x] \leq \mu \vee 1 \leq h[\neg x] \leq \mu)\}$.

As a consequence of the above definition, the set Φ is produced free of assumptions and hence all variables $x \in \Phi$ are eligible for simplifications. In each VCE iteration, a different set Φ is selected. This is achieved by using an upper-bound μ for the number of occurrences of a variable in $\mathcal{S}$ (see Section 4.2.2). After each iteration, μ is increased, allowing the selection of more variables. ParaFROST supports configuring μ and the number of VCE iterations.

As already mentioned, clauses that can be removed are marked DELETED before they are removed. The removal of clauses is done once VCE has finished (see Figure 7.3) to avoid data races. However, because of this, VCE may at first require more memory to store clauses. The clauses added during VCE must fit in the memory, otherwise the procedure fails. To ensure this, we have developed a memory-aware mechanism for VCE. Next, we explain this mechanism for the RES rule and substitution, as the application of those rules results in new clauses.

Algorithm 7.2 presents how RES and substitution are applied in ParaFROST. It requires $\mathcal{S}$, stored in a clause array `clauses`. As clauses are of varying sizes, we need an array `references` that provides a reference to each clause. In addition, arrays `varinfo`, `cindex` and `rindex` are given, which are filled in the first lines.

At line 1, the kernel VceScan is called in which a different thread is assigned to each variable $x \in \Phi$. Each thread checks the applicability of VCE rules on its variable and computes the number of clauses and literals that will be produced by the first applicable rule. A thread with ID *tid* stores the type τ

Algorithm 7.2: Parallel memory-aware application of RES and substitution

```
   Input : global $\Phi$, clauses, references, varinfo, cindex, rindex
 1 varinfo ← VceScan($\Phi, \mathcal{S}_d$)
 2 cindex ← computeClauseIndices(varinfo, size(clauses))
 3 rindex ← computeClauseRefIndices(varinfo, size(references))
 4 VceApply($\Phi$, clauses, references, varinfo, cindex, rindex)
 5 kernel VceApply($\Phi$, clauses, references, varinfo, cindex, rindex):
 6     for all $tid \in [\![\ [0, |\Phi|\rangle\ ]\!]$ do in parallel
 7         register $cidx$ ← cindex[$tid$], $ridx$ = rindex[$tid$]
 8         register $\tau$, $rCls$, $rLits$ ← varinfo[$tid$]
 9         if $\tau$ = RES $\wedge$ memorySafe($ridx$, $cidx$, $rCls$, $rLits$) then
10             resolve(clauses, references, $x$, $ridx$, $cidx$)
11         else if $\tau$ = SUBST $\wedge$ memorySafe($ridx$, $cidx$, $rCls$, $rLits$) then
12             substitute(clauses, references, $x$, $ridx$, $cidx$)
13         end
14     end
15 end
16 device function memorySafe($ridx$, $cidx$, $rCls$, $rLits$):
17     $reqSpace \leftarrow cidx + \texttt{CB} \times rCls + rLits$          // number of buckets
18     if $reqSpace >$ cap(clauses) then return false
19     $reqRefs \leftarrow ridx + rCls$                       // number of clause references
20     if $reqRefs >$ cap(references) then return false
21     return true
22 end
```

of the applicable rule (NONE, RES, or SUBST) and the number of clauses β and literals γ produced by that rule in one integer at varinfo[i]. At lines 2-3, kernels computeClauseIndices and computeClauseRefIndices are called to add up the β's and γ's to obtain offsets into the arrays references and clauses (the method size(A) refers to the amount of data in array A). Both methods apply a parallel exclusive prefix sum [SHGO10], involving the β's and γ's. The result is that thread i, assigned to x, is instructed to start writing clause references at references[rindex[i]] and clauses at clauses[cindex[i]] when applying the next VCE rule for x. Whether the data actually fits is checked later.

Next, the kernel VceApply is called (lines 5-15). To each variable in Φ, a thread is assigned. It retrieves the precomputed data (lines 7-8) and either applies the RES rule (lines 9-10), substitution (lines 11-12), or nothing, in case τ = NONE. However, a condition for applying a rule is that there is enough space, which is checked using the device function memorySafe (lines 16-22).

The amount of allocated space for A in buckets is reflected by cap(A), and

MEMORYSAFE checks if there is enough space in `clauses`, starting at *cidx* (lines 17-18). If there is, it is checked if the references can be stored in `references` (lines 19-20). Recall from Section 5.2, the constant `CB` is the number of buckets needed to store `SCLAUSE` which is 12 bytes / 4 bytes. Note that Algorithm 7.2 does not reason about function tables as it was developed and published before Algorithm 5.5.

7.3 Benchmarks and Analysis

We conducted experiments with CBMC in combination with MINISAT (the default), GLUCOSE, CADICAL, PARAFROST, PARAFROST with MDM, and a CPU-only version, referred to as PARAFROST (NOGPU).[20] We used the AWS benchmarks in which the data structures `hash table`, `array list`, `array buff`, `linked list`, `priority queue`, `byte cursor` and `string` were analysed. The loop unwinding upper-bounds `8`, `16`, `64`, `128` and `1,000` were used, resulting in 168 different verification problems.

All experiments were executed on the DAS-5 cluster [BEdL+16]. Each program was verified in isolation on a separate node, with a time-out of 3,600 seconds. Each node had an Intel Xeon E5-2630 CPU (2.4 GHz) with 64 GB of memory, and an NVIDIA RTX 2080 Ti (specifications are listed in Table 3.2).

Figures 7.5 and 7.6 present the decision procedure runtime and how much time was spent on VCE, respectively. As seen by the former, PARAFROST outperforms all sequential solvers including CADICAL. Even though PARAFROST is based on CADICAL, its different data structures, simplification mechanism and parameters tuned for large formulas makes PARAFROST more effective in these experiments. Further, MDM improves PARAFROST on most test cases. Figure 7.6 emphasizes that CBMC with MINISAT often spends most of the time on VCE. PARAFROST, on the other hand, significantly reduces the time spent on VCE compared to other solvers.

In Table 7.1, the `Verified` column lists per solver the number of verified programs, and `PAR-2` gives the *penalized average runtime-2* metric. The `PAR-2` score accumulates the running times of all solved instances with 2× the time-out of unsolved ones, divided by the total number of formulas. The solver with the lowest score is the winner. The **`MINISAT`** column lists how many programs were verified faster with the other solvers compared to MINISAT. Between parentheses, it is given how many of those programs were not solved by MINISAT

[20]We also tried to use CBMC with Z3, but were not able to correctly configure this combination at the time of writing.

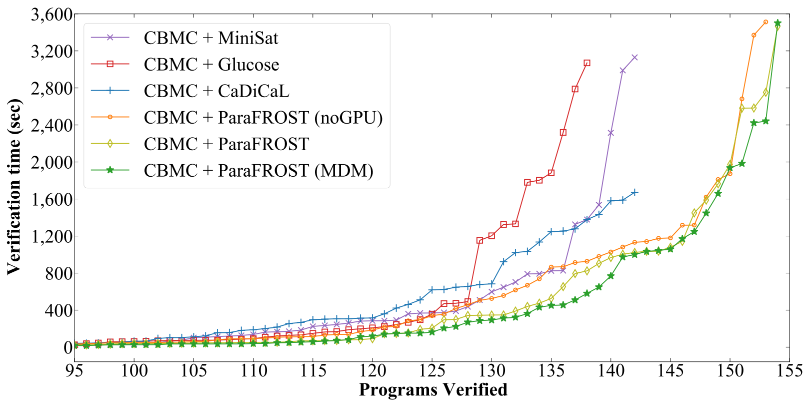


Figure 7.5: Verification time (timeout: 3,600 seconds)

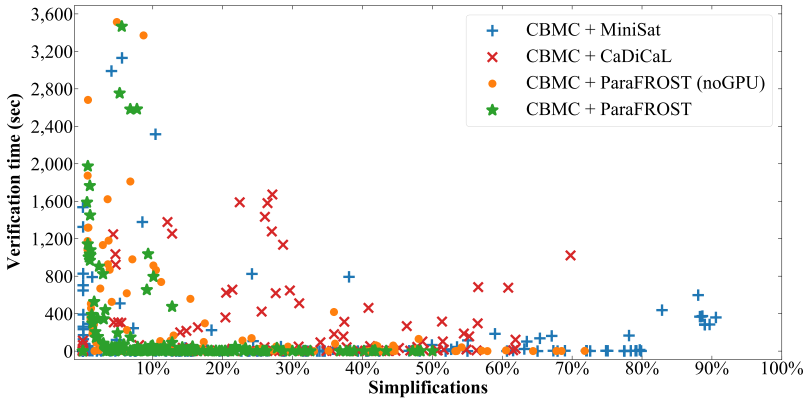


Figure 7.6: Percentage of verification time used for VCE

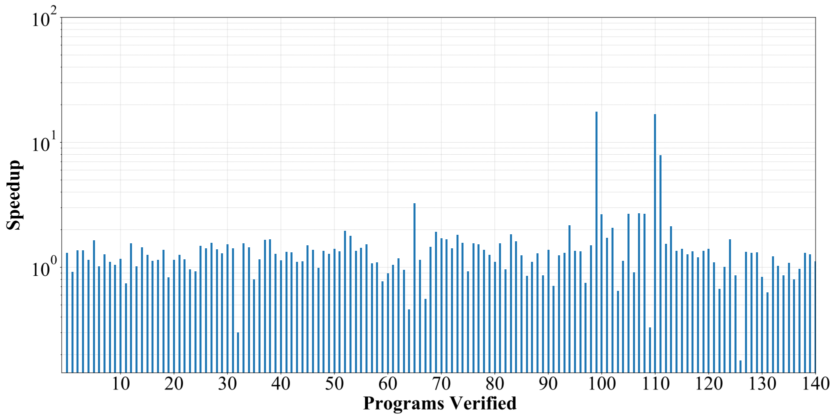


Figure 7.7: ParaFROST vs. ParaFROST (noGPU) speedup

at all. The final four columns serve the same purpose for the other solvers. For example, ParaFROST-MDM verified 123 programs faster than CaDiCaL, of which 12 could not be verified by the latter. The last two rows provide a similar comparison. Clearly, ParaFROST-MDM verified the largest number of programs, with the lowest score.

Figures 7.7-7.12 presents the speedups of the ParaFROST configurations for the individual cases. In Figure 7.7, the GPU solver without MDM is compared against the CPU-only version. The maximum and the geometric average accelerations are 18× and 1.3×, respectively. In Figures 7.8 and 7.9, ParaFROST-MDM achieved higher accelerations up to around 27× and 21× against ParaFROST (noGPU) and ParaFROST versions, respectively. Last, in Figures 7.10-7.12, the impact of ParaFROST-MDM is measured against the state-of-the-art sequential solvers MiniSat, Glucose, and CaDiCaL. Compared to the latter, the GPU solver with MDM enabled achieved a significant speed up to 85×, with a geometric mean of 1.8×.

7.4 Related Work

Wieringa *et al.* [WNH09] presented a framework called tarmo for parallelized BMC on a multi-core architecture or a distributed system such as a cluster

Table 7.1: CBMC performance analysis using the various solvers. The triangles ▲ and ▼ mean significantly better and worse, respectively. PFGPU and PFCPU are shorthands for ParaFROST and ParaFROST (noGPU), respectively.

Configuration	**Verified**	**PAR-2**	**MiniSat**	**Glucose**	**CaDiCaL**	**PFCPU**	**PFGPU**
CBMC + MiniSat	143	1219	n/a	n/a	n/a	n/a	n/a
CBMC + Glucose	139	▼ 1388	▼ 49 (−4)	n/a	n/a	n/a	n/a
CBMC + CaDiCaL	143	1226	43	53 (+4)	n/a	n/a	n/a
CBMC + PFCPU	154	824	51 (+11)	62 (+15)	83 (+11)	n/a	n/a
CBMC + PFGPU	**155**	▲ **765**	▲ **66 (+12)**	▲ **83 (+16)**	▲ **96 (+12)**	**120 (+1)**	n/a
CBMC + PFGPU-MDM	**155**	▲ **743**	▲ **84 (+12)**	▲ **102 (+16)**	▲ **123 (+12)**	**133 (+1)**	**121**

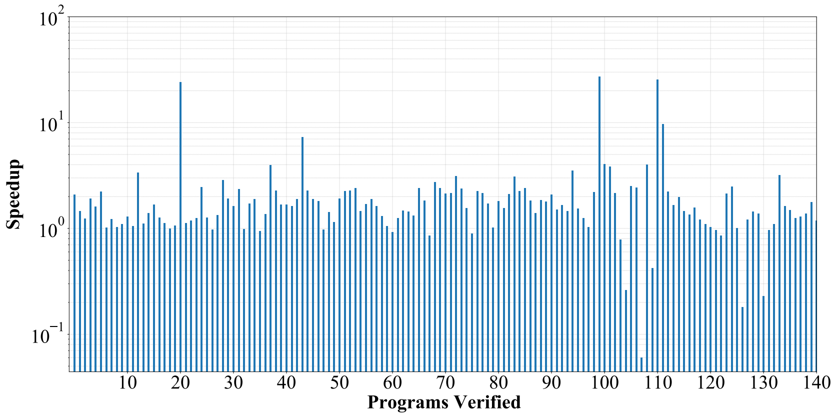


Figure 7.8: ParaFROST (MDM) vs. ParaFROST (noGPU) speedup

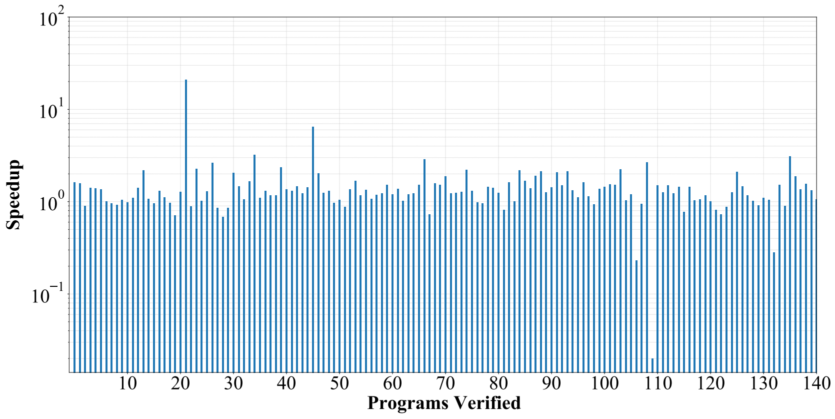


Figure 7.9: ParaFROST (MDM) vs. ParaFROST speedup

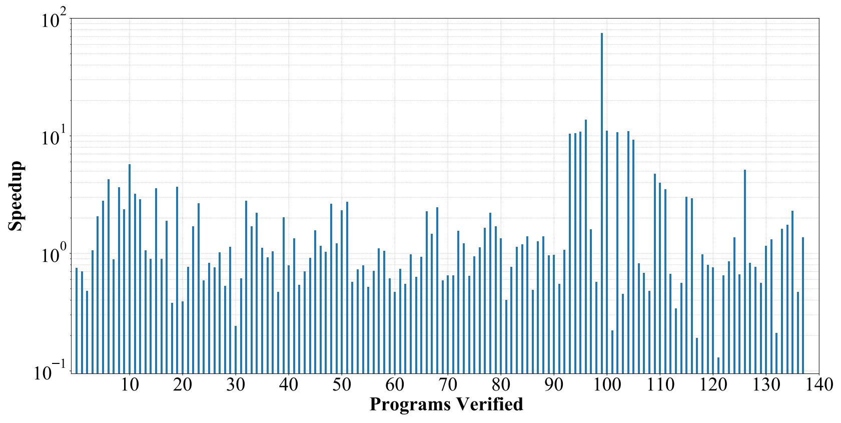


Figure 7.10: ParaFROST (MDM) vs. MiniSat speedup

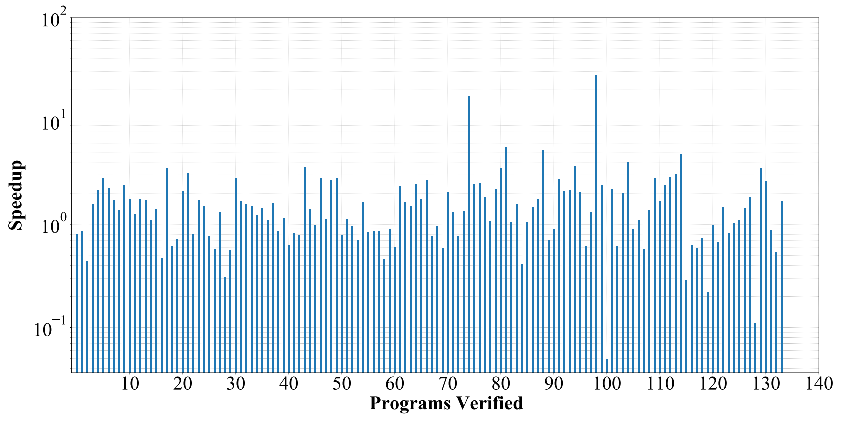


Figure 7.11: ParaFROST (MDM) vs. Glucose speedup

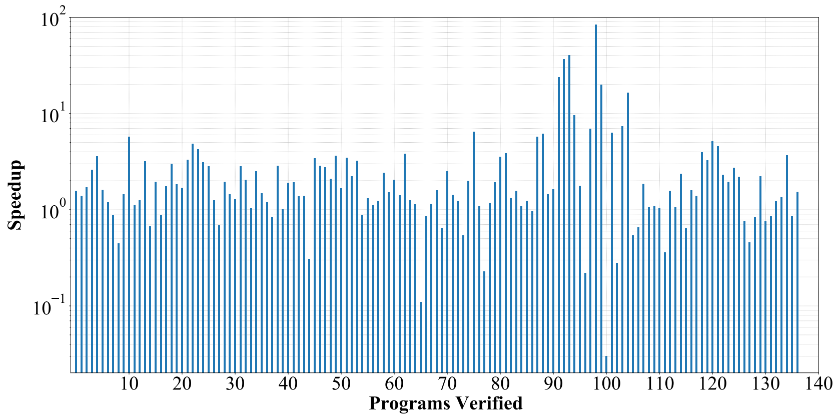


Figure 7.12: ParaFROST (MDM) vs. CaDiCaL speedup

of computing nodes. The tool runs multiple incremental SAT problems in parallel allowing efficient clause sharing between the launched workers. Likewise, Ábrahám *et al.* [ÁSB$^+$11] applied parallel SAT solvers to check the satisfiability of a BMC problem. Their approach works on formulas for different counterexample lengths. Chatterjee *et al.* [CRDL20] introduced a dynamic technique to split a BMC program into sub-tasks that can be verified in parallel with an SMT solver.

On the other hand, Inverso *et al.* [IT20] presented a structure-aware parallel technique for bounded analysis of concurrent programs. A set of concurrent traces are decomposed into symbolic subsets that are simultaneously explored by multiple instances of the same decision procedure running in parallel. Unlike the related work above, the decision procedures work on different parts of the search space without any sharing involved. Nevertheless, none of these techniques investigate the GPU acceleration of BMC. To the best of our knowledge, we are the first to investigate this.

7.5 Conclusion

We have presented ParaFROST, the first tool to accelerate BMC using GPUs. Given that BMC formulas tend to have much redundancy, ParaFROST effectively reduces solving times with GPU pre- and inprocessing, and by using MDM, which is particularly effective when many variables are present.

In the future, we will combine our approach with (existing) multi-threaded BMC. We expect these techniques to strengthen each other. As an alternative to BMC, the IC3 algorithm is parallelisable as reported in [Bra11]. IC3 is currently the state-of-the-art in symbolic MC which is capable of generating many small lemmas (SAT formulas) independent of each other. This could be attractive for SIMT architectures such as the GPU.

Chapter 8

Conclusions

"That your book has been delayed I am glad, since you have gained an opportunity of being more exact."

– Samuel Johnson

In this chapter, first, we reflect on the main contributions of this thesis in the context of the research questions stated in Chapter 1. For each of these, we recall a summary of the experimental results and draw conclusions. Second, directions of future work are discussed.

8.1 Contributions

We have conducted research on the acceleration of *symbolic model checking* through *SAT-based bounded* checking on many-core architectures. Due to the complexity of the control flow and data dependency, we have not yet ported the CDCL search entirely to the GPU. However, we have shown that SAT simplifications via *pre-* and *inprocessing*, which form almost 50% of the solver workload, can be accelerated effectively on the GPU, yielding massive gains in the solver performance. Further, we have made the sequential CDCL search faster by the MDM procedure even when it runs sequentially. To this extend, seven research questions were formulated. Next, we discuss how the research questions relate to the chapters presented in this thesis.

SAT Preprocessing. The current main challenges in parallel SAT solving were discussed by the authors of [HW12, BS18]. A major challenge concerns the *parallelisation* of SAT simplification in modern SAT solvers. Massively parallel platforms such as GPUs offer great potentiality to speed up computations. The following research question addresses how to accomplish this by designing new parallel algorithms and data structures from scratch to make optimal use of those platforms.

RQ1: How can GPUs be employed to perform scalable parallel SAT simplifications?

We addressed this question in Chapter 4. The main challenge we tackled in applying simplification techniques in parallel is the strong dependency between variables in a SAT formula. Hence, we proposed the LCVE algorithm which is responsible for scheduling a set of mutually independent variables that are eligible for parallel simplification. Consequently, we introduced the first GPU algorithms for variable and subsumption eliminations, which are essential nowadays in any SAT solver. In addition, we presented the HRE method and its GPU implementation to further remove redundant clauses from SAT problems. Correctness of the proposed parallel algorithms is discussed with formal proofs. Building on our work for **RQ1**, we ask ourselves if the proposed parallel simplifications scale up as well on multiple GPUs:

RQ2: Can we run parallel simplifications on a multi-GPU environment, with or without sharing information?

We proposed in Chapter 4 a generalisation of all developed algorithms to distribute simplification work over single or multiple GPUs, if these are available in a single machine. Having multiple GPUs, can definitely mitigate the lack of memory of a single GPU, particularly for extremely large CNF formulas. Balancing the workload of various simplification methods effectively across the available graphics processors is feasible, using the proposed balancing scheme labeled *ping-pong* distribution.

As a result of the above contributions, a new tool came to light called SIGMA. It takes a SAT formula in DIMACS format, applies variable-clause eliminations on one or multiple GPUs, and outputs a simplified formula with less variables and/or clauses. The tool was evaluated on a large set of benchmarks and compared to SatElite. Further, SIGmA impact on solving was tested by supplying the produced formulas as input to MiniSat and Lingeling (the state of the art 4 years ago) solvers.

Overall, our preprocessor SIGmA achieved considerable speedups compared to the sequential counterparts. For example, SIGmA outperformed SatElite and Lingeling by accelerations up to 49× and 32×, respectively. With two GPUs, if we ignore the time needed for data transfer between them, SIGmA scales very well when the number of GPUs is increased. In mode SUB, the average acceleration was 2.64×, and overall, the average speedup was 1.96×. When we considered data transfer, the latter dropped to 0.85×. Still, SIGmA managed to simplify 22% of the problems faster compared to the single GPU mode, even if the communication overhead is taken into account. The major advantage of the multi-GPU configuration is that it may allow more reductions to be obtained, where a single GPU cannot due to limited memory capacity. On the other hand, the hardware limitation of inter-GPU communication makes the data transfers slow. We expect the multi-GPU mode to be increasingly attractive in the future as the hardware advances (see a related discussion in the next section). Regarding the impact of SIGmA on SAT solving, SIGmA + MiniSat solved 139 and 122 problems faster than MiniSat only and SatElite + MiniSat, respectively. As for Lingeling, 128 problems were solved faster by applying SIGmA preprocessing.

SAT Inprocessing. Since 2013, simplification techniques [SP04, EB05, HJB10] are used periodically *during* SAT solving, which is known as inprocessing [JHB12, BGJ$^+$18]. Applying inprocessing iteratively to large problems can be a performance bottleneck in the solving routine, or even increase the size of the formula, negatively impacting the solving time. The next research question addresses the feasibility of running inprocessing frequently in SAT solving.

> **RQ3:** Is it possible to run parallel simplifications regularly during CDCL SAT solving, considering the growing size of the formula by *learning* new clauses and transferring data back and forth to the CPU?

To do this efficiently, as discussed in Chapter 5, we crafted a new space-efficient data structure for keeping the formula in the GPU memory, and a parallel garbage collector to maintain the GPU memory and achieve maximum data locality. A new parallel algorithm for BVE was presented which is twice as fast as the one previously presented in Chapter 4. Further, BVE was strengthened by parallel function-table reasoning to find irregular gates that could not been found by the syntactic approach.

The algorithms introduced in Chapter 5 were the seed of our hybrid SAT solver: ParaFROST. The solver runs the inprocessing part on the GPU side, making full use of its hardware capabilities, while the solving itself runs on the CPU side. The CDCL engine was devised from scratch based on CaDiCaL heuristics. Nonetheless, correctness (non-formal) of the GPU implementations is crucial for the soundness of our tools, especially if they are used in critical applications such as model checkers [OW21a]. This imposes the next two research questions:

> **RQ4:** Can we harness the intrinsic CUDA capabilities such as intra-warp communications and concurrent streams for pushing ParaFROST performance beyond its limits?

> **RQ5:** Can we generate a clausal proof for the GPU code implemented in our solver ParaFROST?

The compute capabilities of modern GPUs rapidly evolve, and some of the ideas suggested in Chapter 4 algorithms should be reconsidered for today's GPUs. Thereby, we are continuously improving ParaFROST by leveraging these capabilities. We already discussed some of these features in Chapter 5. For instance, shuffle instructions are used in Algorithms 5.1, 5.2, and 5.5 to do parallel reduction and exclusive scan per warp. Another example is the usage of concurrent streams to overlap kernel executions and memory transfers (Algorithm 5.4). To address **RQ5**, first, we explained how the proof memory can be managed efficiently on the GPU using binary DRAT format [WHH14]. Second, we developed a strategy to emit clausal proofs for the GPU simplifications without causing any degradation to the solver's overall performance. The generated proofs can be then verified off-the-shelf via drat-trim to validate the behavior of ParaFROST. Again, warp-intrinsic functions are exploited to do the DRAT proof counting in Algorithm 5.2.

Empirically speaking, our solver (ParaFROST) accomplished massive gains through GPU-accelerated inprocessing compared to its sequential version and the state-of-the-art solver CaDiCaL. With the improvements made to the BVE procedure in Chapter 5, the usage of atomic operations has been considerably reduced which lead to an average speedup of 1.6× compared to the atomic version.

Owing to FUNTAB reasoning, more logical gates can be detected and removed with an average speedup of 11.33× compared to the sequential counterpart. We proposed the first parallel GC and proof generation on the GPU for SAT applications with average accelerations of 35× and 11×, respectively. The garbage collector helped reduce the GPU memory consumption while stimulating coalesced memory access. The proof generator allowed PARAFROST to validate all the SAT simplifications running on the GPU besides the CDCL search, giving absolute credibility to our solver and its use in critical applications such as model checkers.

Multiple Decision Making. Most papers in the existing literature are focused on portfolio-based parallel SAT solving [ABK+13] where multiple SAT solvers run concurrently to solve the same problem. On the other hand, the *cube-and-conquer* solver [HKWB11] tries to decompose the original formula into many smaller subformulas then explore all of them in parallel via multiple solvers. Thus, the next research question is:

RQ6: Can CDCL search be partially or entirely parallelizable?

In Chapter 6, we introduced the MDM procedure in CDCL which is capable of making thousands, even millions of decisions that are independent of each other and hence can be assigned and propagated in parallel. The goal was to prune the search space and hence reduce the solving time. However, doing so in parallel turned to be slower than the sequential propagation. Later, we observed that applying MDM sequentially has a positive impact on large formulas, particularly stemming from verification problems. To make such decisions, MDM requires that they do not lead to implications or conflicts. Doing this may lessen the number of conflicts that arise from making bad assignments, which in turn prunes the search space.

Several optimizations have been made to MDM including the VSIDS and VMTF decision heuristics which were used to improve the quality of the picked decisions. Further, PARAFROST restart was extended with interleaved policies as implemented in CADICAL which combines both MINISAT and GLUCOSE heuristics and we showed how that can be effective in MDM. Based on restarts and the total number of conflicts encountered, the MDM procedure can alternate between the VSIDS and VMTF decision queues. As local search is more attractive to guide the solver in finding solutions for satisfiable formulas, we implemented a minimal version of the WALKSAT strategy to improve the truth values when selecting multiple decisions. Interleaving local search with CDCL has been adopted before and proved its efficacy on satisfiable problems in modern SAT solvers such as CaDiCaL and CryptoMiniSat.

Overall, a sequential version of ParaFROST with MDM enabled outperformed Kissat solver (2020 release) with a minimum PAR-2 score of 3,473.7 against 3,757.83 attained by the latter, solving more *SAT* instances in less time, while being competitive with the *UNSAT* ones.

Bounded Model Checking. Through our last research question, we address the integration and the impact of a GPU-accelerated solver on existing BMC tools:

> **RQ7:** How can GPUs be employed effectively to speed up BMC?

To address this question, we combined all the above achievements to accelerate BMC with our solver ParaFROST. We found that SAT formulas stemming from bounded model checkers such as CBMC have enormous redundancies in variables and clauses. This amount of redundancy takes a lot of memory space on the GPU. Thus, we compacted further the data structure that was first proposed in Chapter 5 in order to reduce the memory consumption of SAT simplification.

In BMC problems, the number of resolvents added by the BVE procedure is significant. Therefore, we introduced a memory *guard* to protect the space pre-allocated for BVE from access violations. In other words, the guard gives GPU threads the freedom of deciding whether a variable can be eliminated or not. Additionally, the feature of *incremental* SAT solving has been added to ParaFROST in order to support SAT-based k-induction BMC. To this end, ParaFROST was integrated into the CBMC model checker using a configurable interface called GPU4BMC.

Based on empirical results, first, we concluded that incremental solving has not affected the effectiveness of the GPU-accelerated simplifications, as the number of assumptions that have to be disabled during the incremental search is a small fraction of the amount of redundancy resulting from BMC problems. Second, we observed that program verification via CBMC can be accelerated effectively with the GPU solvers ParaFROST and ParaFROST-MDM (a configuration with MDM enabled). For example, compared to ParaFROST (noGPU) (both simplifications and the search run on the CPU), ParaFROST (and ParaFROST-MDM), accelerated multiple program verification tasks by up to 18× (and 27×), and the geometric average speedup for all programs was 1.3× (and 1.6×).

8.2 Future Work

The main drawback of distributing the workload of SAT simplifications across multiple GPUs was the communication between them. In Chapter 4, we have demonstrated how this can be done using a single machine with two GPUs installed. However, the data bandwidth offered by the PCI-E version 3 (i.e. the bus connecting the GPU with the CPU and the system memory) is very limited with a peak of 16 GB/s. Now with the release of versions 4 and 5, the bandwidths are doubled to 32 and 64 GB/s, respectively, which is a massive leap since SIGmA was developed and benchmarked. To make the data transfer even faster, NVIDIA announced the NVLink bridge (a GPU-to-GPU interconnect)[21], that can add another 50 GB/s to the PCI-E bandwidth. Testing these technologies on pre- and inprocessing is yet to be done. Further, Shen *et al.* [SVL+16] have presented effective strategies to dynamically partition and balance the workload of several case studies on heterogeneous systems consisting of CPU and GPU nodes. This presents an opportunity to investigate the application of these techniques on optimally partitioning SAT simplifications across the available processing nodes.

As discussed in Chapter 6, parallel SAT solving is challenging because of the heavy dependency between literal assignments and random memory accesses in BCP. The MDM procedure was presented to tackle the former by issuing a mutually-independent set of multiple decisions ready for propagation simultaneously. A multithreaded prototype for propagating these decisions has been already developed; however, initial experimentation has revealed that a single-threaded execution is actually faster. The main reason for the slowdown was sharing the same formula between all threads and accessing independent parts randomly, not taking advantage of the available cache and causing a contention to the system memory. A potential solution is to group all clauses that belong to one or more assigned variables to be accessed by the same responsible thread together. Another solution is to replicate heavily-accessed small arrays into separate buffers to avoid contention to the same data locations.

An entirely different approach to accelerate SAT solving is to run multiple instances of the same or different solvers concurrently to process a SAT formula. Whichever finds a solution first or declares the formula is unsatisfiable, is the winner and terminates the other working solvers. This approach is called portfolio and has been investigated by the research community over a decade [HJS09, ABK+13]. In our case, multiple instances of ParaFROST with different configurations and heuristics can be launched on a single machine

[21] https://www.nvidia.com/en-us/data-center/nvlink

using multiple CPU threads or a cluster of machines. This has the advantage of analysing SAT formulas generated by bounded model checkers across different CPU threads or machines [CRDL20, IT20].

In Chapter 7, we discussed how BMC can benefit from parallel eliminations on the GPU while running the MDM procedure within the search, particularly optimised for BMC problems. Alternatively, the IC3 algorithm is parallelisable as reported in [Bra11]. IC3 is currently revolutionary in symbolic MC which is capable of generating many small lemmas (SAT formulas) independent of each other. This could be attractive for SIMT architectures such as the GPU.

Another direction is to harness the massive GPU capabilities in accelerating probabilistic MC [dAKN+00]. The goal of this technique is to prove the correctness of stochastic systems that exhibit probabilistic behavior. Several tools are developed to automate the checking process such as PRISM [KNP02] and Storm [DJKV17]. A parallel implementation of the former has been developed in CUDA by Bošnački *et al.* [BESW10]. Later on, Wijs *et al.* [WB12] improved *sparse matrix-vector* (SpMV) multiplication. Recently, Khan *et al.* [KHK21] proposed an acceleration of SpMV multiplication in Storm by hiding the memory latency involved in these operations via asynchronous memory copies. This leads us to the possibility of using the NVIDIA tensor cores[22] to further optimise SpMV operations. Tensor cores are specially devised to speed up mixed integer and floating-point operations while preserving accuracy. Further step to the future is moving SpMV computations closer to the memory which is known as Computation in Memory (CIM) [YNH+16]. This paradigm helps the researchers reduce the communication overhead when it comes to processing huge chunks of data as in Big Data problems.

[22] https://www.nvidia.com/en-us/data-center/tensor-cores

Bibliography

[ABK$^+$13] M. Aigner, A. Biere, C. M. Kirsch, A. Niemetz, and M. Preiner. Analysis of Portfolio-Style Parallel SAT Solving on Current Multi-Core Architectures. In *Proc. of POS (Jul. 2013), Helsinki, Finland*, volume 29 of *EPiC Series in Computing*, pages 28–40. EasyChair. doi:10.29007/73n4.

[Ama21] Amazon. The Amazon Web Services Core C99 Package Benchmark Set, 2021. URL `https://github.com/awslabs/aws-c-common/tree/main/verification/cbmc/proofs`.

[AN16] Abhinav and R. Nasre. FastCollect: offloading generational garbage collection to integrated GPUs. In *Proc. of CASES (Oct. 2016), Pittsburgh, USA*, pages 21:1–21:10. ACM. doi:10.1145/2968455.2968520.

[APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. *Information Processing Letters*, 8(3):121–123, 1979. doi:10.1016/0020-0190(79)90002-4.

[AS09] G. Audemard and L. Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proc. of IJCAI (Jul. 2009), California, USA*, pages 399–404. doi:10.5555/1661445.1661509.

[AS12] G. Audemard and L. Simon. Refining Restarts Strategies for SAT and UNSAT. In *Proc. of CP (Oct. 2012), Québec, Canada*, volume 7514 of *LNCS*, pages 118–126. Springer. doi:10.1007/978-3-642-33558-7_11.

[ÁSB$^+$11] E. Ábrahám, T. Schubert, B. Becker, M. Fränzle, and C. Herde. Parallel SAT Solving in Bounded Model Checking. *Journal of Logic and Computation*, 21(1):5–21, 2011. doi:10.1093/logcom/exp002.

[BBC$^+$20] F. B. Buonamici, G. Belmonte, V. Ciancia, D. Latella, and M. Massink. Spatial logics and model checking for medical imaging. *International Journal on Software Tools for Technology Transfer*, 22(2):195–217, 2020. doi:10.1007/s10009-019-00511-9.

[BCC$^+$03] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003. doi:10.1016/S0065-2458(03)58003-2.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. of TACAS (Mar. 1999), Amsterdam, The Netherlands*, volume 1579 of *LNCS*, pages 193–207. Springer. doi:10.1007/3-540-49059-0_14.

[BCLR04] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Proc. of iFM (Apr. 2004), Canterbury, UK*, volume 2999 of *LNCS*, pages 1–20. Springer. doi:10.1007/978-3-540-24756-2_1.

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992. doi:10.1016/0890-5401(92)90017-A.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proc. of DAC (Jun. 1990), Florida, USA*, pages 46–51. IEEE Press. doi:10.1145/123186.123223.

[BCRZ99] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifiying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs. In *Proc. of CAV (Jul. 1999), Trento, Italy*, volume 1633 of *LNCS*, pages 60–71. Springer. doi:10.1007/3-540-48683-6_8.

[BEdL$^+$16] H. E. Bal, D. H. J. Epema, C. de Laat, R. van Nieuwpoort, J. W. Romein, F. J. Seinstra, C. Snoek, and H. A. G. Wijshoff. A

Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer*, 49(5):54–63, 2016. doi:10.1109/MC.2016.127.

[BESW10] D. Bošnački, S. Edelkamp, D. Sulewski, and A. Wijs. GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In *PDMC-HiBi (2010)*, pages 17–19. doi:10.1109/PDMC-HiBi.2010.11.

[BF15] A. Biere and A. Fröhlich. Evaluating CDCL Variable Scoring Schemes. In *Proc. of SAT (Sept. 2015), Austin, USA*, volume 9340 of *LNCS*, pages 405–422. Springer. doi:10.1007/978-3-319-24318-4_29.

[BF18] A. Biere and A. Fröhlich. Evaluating CDCL Restart Schemes. In *Proc. of POS (Sept. 2015), Austin, USA*, volume 59 of *EPiC Series in Computing*, pages 1–17. EasyChair. doi:10.29007/89dw.

[BFFH20] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SC (2020)*, volume B-2020-1 of *Report Series B*, pages 51–53. University of Helsinki. URL `http://hdl.handle.net/10138/318450`.

[BGJ+18] F. S. Bao, C. E. Gutierrez, J. Jn-Charles, Y. Yan, and Y. Zhang. Accelerating Boolean Satisfiability (SAT) solving by common subclause elimination. *Artificial Intelligence Review*, 49(3):439–453, 2018. doi:10.1007/s10462-016-9530-6.

[Bie13] A. Biere. Lingeling, Plingeling and Treengeling Entering the Sat Competition 2013. In *Proc. of SC (2013)*, volume B-2013-1 of *Report Series B*, pages 51–52. University of Helsinki. URL `http://hdl.handle.net/10138/40026`.

[BJK21] A. Biere, M. Järvisalo, and B. Kiesl. Preprocessing in SAT Solving. In *Proc. of Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2nd edition, 2021.

[BJLB+20] A. Biere, M. Järvisalo, D. Le Berre, K. S. Meel, and S. Mengel. The SAT Practitioner's Manifesto. In Zenodo (Sept. 2020). doi:10.5281/zenodo.4500928.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

[BOA09] M. Billeter, O. Olsson, and U. Assarsson. Efficient Stream Compaction on Wide SIMD Many-Core Architectures. In *Proc. of HPG (Aug. 2009), New Orleans, USA*, pages 159–166. ACM. doi:10.2312/EGGH/HPG09/159-166.

[Bra11] A. R. Bradley. SAT-Based Model Checking without Unrolling. In *Proc. of VMCAI (Jan. 2011), Austin, USA*, volume 6538 of *LNCS*, pages 70–87. Springer. doi:10.1007/978-3-642-18275-4_7.

[Bro13] C. E. Brown. Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems. *Journal of Automated Reasoning*, 51(1):57–77, 2013. doi:10.1007/s10817-013-9283-8.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. doi:10.1109/TC.1986.1676819.

[BS18] T. Balyo and C. Sinz. Parallel Satisfiability. In *Proc. of Handbook of Parallel Constraint Reasoning*, pages 3–29. Springer, 2018. doi:10.1007/978-3-319-63516-3_1.

[BSST09] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-825.

[CCGR99] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A New Symbolic Model Verifier. In *Proc. of CAV (Jul. 1999), Trento, Italy*, volume 1633 of *LNCS*, pages 495–499. Springer. doi:10.1007/3-540-48683-6_44.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Proc. of LP Workshop (May 1981), New york, USA*, volume 131 of *LNCS*, pages 52–71. Springer. doi:10.1007/BFb0025774.

[CG87] E. M. Clarke and O. Grumberg. Avoiding The State Explosion Problem in Temporal Logic Model Checking. In *Proc. of PODC (Aug. 1987), Vancouver, Canada*, pages 294–303. ACM. doi:10.1145/41840.41865.

[CHT20] M. S. Cherif, D. Habet, and C. Terrioux. On the Refinement of Conflict History Search Through Multi-Armed Bandit. In *Proc. of ICTAI (Nov. 2020), Baltimore, USA*, pages 264–271. IEEE. doi:10.1109/ICTAI50040.2020.00050.

[CKK+18] B. Cook, K. Khazem, D. Kroening, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Model Checking Boot Code from AWS Data Centers. In *Proc. of CAV (Jul. 2018), Oxford, UK*, volume 10982 of *LNCS*, pages 467–486. Springer. doi:10.1007/978-3-319-96142-2_28.

[CKL04] E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proc. of TACAS (Mar. 2004), Barcelona, Spain*, volume 2988 of *LNCS*, pages 168–176. Springer. doi:10.1007/978-3-540-24730-2_15.

[CRDL20] P. Chatterjee, S. Roy, B. P. Diep, and A. Lal. Distributed Bounded Model Checking. In *Proc. of FMCAD (Sept. 2020), Haifa, Israel*, pages 47–56. IEEE. doi:10.34727/2020/isbn.978-3-85448-042-6_11.

[dAKN+00] L. de Alfaro, M. Z. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic Model Checking of Probabilistic Processes Using MTBDDs and the Kronecker Representation. In *Proc. of TACAS 2000, Berlin, Germany*, volume 1785 of *LNCS*, pages 395–410. Springer. doi:10.1007/3-540-46419-0_27.

[DFLO19] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn. Scaling Static Analyses at Facebook. *Communication of the ACM*, 62(8):62–70, 2019. doi:10.1145/3338112.

[Dij72] E. W. Dijkstra. The Humble Programmer. *Communication of the ACM*, 15(10):859–866, oct 1972. doi:10.1145/355604.361591.

[DJKV17] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk. A Storm is Coming: A Modern Probabilistic Model Checker. In *Proc. of CAV (Jul. 2017), Heidelberg, Germany*, volume 10427 of *LNCS*, pages 592–600. Springer. doi:10.1007/978-3-319-63390-9_31.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008. doi:10.1109/TCAD.2008.923410.

[DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. doi:10.1145/368273.368557.

[EB05] N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proc. of SAT (Jun. 2005), St. Andrews, UK*, volume 3569 of *LNCS*, pages 61–75. Springer. doi:10.1007/11499107_5.

[EH83] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time. In *Proc. of POPL (Jan. 1983), Austin, USA*, pages 127–140. ACM Press. doi:10.1145/567067.567081.

[ES03a] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. of SAT (May 2003) Santa Margherita Ligure, Italy*, volume 2919 of *LNCS*, pages 502–518. Springer. doi:10.1007/978-3-540-24605-3_37.

[ES03b] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003. doi:10.1016/S1571-0661(05)82542-3.

[FM09] J. Franco and J. Martin. A History of Satisfiability. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 3–74. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-3.

[FVS11] J. Fang, A. L. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing*, pages 216–225. doi:10.1109/ICPP.2011.45.

[GH21] S. Geisler and A. E. Haxthausen. Stepwise development and model checking of a distributed interlocking system using RAISE. *Formal Aspects of Computing*, 33(1):87–125, 2021. doi:10.1007/s00165-020-00507-2.

[GM13] K. Gebhardt and N. Manthey. Parallel Variable Elimination on CNF Formulas. In *Proc. of KI (Sept. 2013), Koblenz, Germany*, volume 8077 of *LNCS*, pages 61–73. Springer. doi:10.1007/978-3-642-40942-4_6.

[GN07] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust Sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007. doi:10.1016/j.dam.2006.10.007.

[GSK98] C. P. Gomes, B. Selman, and H. A. Kautz. Boosting Combinatorial Search Through Randomization. In *Proc. of AAAI (Jul. 1998), Wisconsin, USA*, pages 431–437. AAAI Press / The MIT Press.

[HJB10] M. Heule, M. Järvisalo, and A. Biere. Clause Elimination Procedures for CNF Formulas. In *Proc. of LPAR (Oct. 2010), Yogyakarta, Indonesia*, volume 6397 of *LNCS*, pages 357–371. Springer. doi:10.1007/978-3-642-16242-8_26.

[HJS09] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2009. doi:10.3233/sat190070.

[HJW13] M. Heule, W. A. H. Jr., and N. Wetzler. Verifying Refutations with Extended Resolution. In *Proc. of CADE (Jun. 2013), New York, USA*, volume 7898 of *LNCS*, pages 345–359. Springer. doi:10.1007/978-3-642-38574-2_24.

[HKWB11] M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In K. Eder, J. Lourenço, and O. Shehory, editors, *Proc. of HVC (Dec. 2011), Haifa, Israel*, volume 7261 of *LNCS*, pages 50–65. Springer. doi:10.1007/978-3-642-34188-5_8.

[Hol97] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. doi:10.1109/32.588521.

[HS07] H. Han and F. Somenzi. Alembic: An Efficient Algorithm for CNF Preprocessing. In *Proc. of DAC (Jun 2007), San Diego, USA*, pages 582–587. IEEE. doi:10.1145/1278480.1278628.

[HT19] D. Habet and C. Terrioux. Conflict history based search for constraint satisfaction problem. In *Proc. of SAC (Apr. 2019), Limassol, Cyprus*, pages 1117–1122. ACM. doi:10.1145/3297280.3297389.

[HW12] Y. Hamadi and C. M. Wintersteiger. Seven Challenges in Parallel SAT Solving. In *Proc. of AAAI (Jul. 2012), Toronto, Canada*. AAAI Press.

[IT20] O. Inverso and C. Trubiani. Parallel and Distributed Bounded Model Checking of Multi-threaded Programs. In *Proc. of PPoPP (Feb. 2020), California, USA*, pages 202–216. ACM. doi:10.1145/3332466.3374529.

[JBH10] M. Järvisalo, A. Biere, and M. Heule. Blocked Clause Elimination. In *Proc. of TACAS (Mar. 2010), Paphos, Cyprus*, volume 6015 of *LNCS*, pages 129–144. Springer. doi:10.1007/978-3-642-12002-2_10.

[JBH12] M. Järvisalo, A. Biere, and M. Heule. Simulating Circuit-Level Simplifications on CNF. *Journal of Automated Reasoning*, 49(4):583–619, 2012. doi:10.1007/s10817-011-9239-9.

[JHB12] M. Järvisalo, M. Heule, and A. Biere. Inprocessing Rules. In *Proc. of IJCAR (Jun. 2012), Manchester, UK*, volume 7364 of *LNCS*, pages 355–370. Springer. doi:10.1007/978-3-642-31365-3_28.

[JS05] H. Jin and F. Somenzi. An Incremental Algorithm to Check Satisfiability for Bounded Model Checking. *Electronic Notes in Theoretical Computer Science*, 119(2):51–65, 2005. doi:10.1016/j.entcs.2004.06.062.

[JT96] D. S. Johnson and M. A. Trick. Cliques, Coloring, and Satisfiability. In *Proc. of DIMACS Workshop (Oct. 1996), New Jersey, USA*, volume 26. DIMACS/AMS. doi:10.1090/dimacs/026.

[KHK21] M. H. Khan, O. Hassan, and S. Khan. Accelerating spmv multiplication in probabilistic model checkers using gpus. In *Proc. of ICTAC (Sept. 2021), Nur-Sultan, Kazakhstan*, volume 12819 of *LNCS*, pages 86–104. Springer. doi:10.1007/978-3-030-85315-0_6.

[KNP02] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In *Proc. of TOOLS (Apr. 2002), London, UK*, volume 2324 of *LNCS*, pages 200–204. Springer. doi:10.1007/3-540-46029-2_13.

[KS16] D. Kroening and O. Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. doi:10.1007/978-3-662-50497-0.

[KT11] T. Kahsai and C. Tinelli. PKind: A Parallel k-Induction Based Model Checker. In *Proc. of PDMC (Jul. 2011), Snowbird, Utah, USA*, volume 72 of *EPTCS*, pages 55–62. doi:10.4204/EPTCS.72.6.

[KT14] D. Kroening and M. Tautschnig. CBMC - C Bounded Model Checker - (Competition Contribution). In *Proc. of TACAS (Apr. 2014), Grenoble, France*, volume 8413 of *LNCS*, pages 389–391. Springer. doi:10.1007/978-3-642-54862-8_26.

[Kul99] O. Kullmann. On a Generalization of Extended Resolution. *Discrete Applied Mathematics*, 96-97:149–176, 1999. doi:10.1016/S0166-218X(99)00037-2.

[LGPC16] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Proc. of SAT (Jul. 2016), Bordeaux, France*, volume 9710 of *LNCS*, pages 123–140. Springer. doi:10.1007/978-3-319-40970-2_9.

[Li00] C. M. Li. Integrating Equivalency Reasoning into Davis-Putnam Procedure. In *Proc. of AAAI (Aug. 2000), Austin, USA*, pages 291–296. AAAI Press / The MIT Press.

[LS03] I. Lynce and J. P. M. Silva. Probing-Based Preprocessing Techniques for Propositional Satisfiability. In *Proc. of ICTAI (Nov. 2003), California, USA*, page 105. IEEE. doi:10.1109/TAI.2003.1250177.

[LSZ93] M. Luby, A. Sinclair, and D. Zuckerman. Optimal Speedup of Las Vegas Algorithms. *Information Processing Letters*, 47(4):173–180, 1993. doi:10.1016/0020-0190(93)90029-9.

[LTE$^+$20] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science*, 368(6495), 2020. doi:10.1126/science.aam9744.

[Mer20] D. Merrill. CUB: A Parallel Primitives Library. *NVLabs*, 2020. URL `https://nvlabs.github.io/cub/`.

[MMZ$^+$01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of DAC (Jun. 2001), Las Vegas, USA*, pages 530–535. ACM. doi:10.1145/378239.379017.

[Moo65] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114, 1965. doi:10.1109/N-SSC.2006.4785860.

[MRM$^+$12] M. Maas, P. Reames, J. Morlan, K. Asanovic, A. D. Joseph, and J. Kubiatowicz. GPUs as an opportunity for offloading garbage collection. In *Proc. of ISMM (Jun. 2012), Beijing, China*, pages 25–36. ACM. doi:10.1145/2258996.2259002.

[NVI20a] NVIDIA. CUDA C Programming Guide, 2020. URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[NVI20b] NVIDIA. The Thrust library, 2020. URL `https://docs.nvidia.com/cuda/thrust`.

[OGHM18] M. Osama, L. Gaber, A. I. Hussein, and H. Mahmoud. An Efficient SAT-Based Test Generation Algorithm with GPU Accelerator. *Journal of Electronic Testing*, 34(5):511–527, 2018. doi:10.1007/s10836-018-5747-4.

[OGMS02] R. Ostrowski, É. Grégoire, B. Mazure, and L. Sais. Recovering and Exploiting Structural Knowledge from CNF Formulas. In *Proc. of CP (Sept. 2002), New York, USA*, volume 2470 of *LNCS*, pages 185–199. Springer. doi:10.1007/3-540-46135-3_13.

[OMH20] P. Oostema, R. Martins, and M. Heule. Coloring Unit-Distance Strips using SAT. In *Proc. of LPAR (May 2020), Alicante, Spain*, volume 73 of *EPiC Series in Computing*, pages 373–389. EasyChair. doi:10.29007/btmj.

[Osa21a] M. Osama. Large SAT Benchmark Suite for Certified SAT Solving with GPU Accelerated Inprocessing, July 2021.

[Osa21b] M. Osama. ParaFROST Proofs for Certified SAT Solving with GPU Accelerated Inprocessing, July 2021. URL `https://doi.org/10.5281/zenodo.5137887`.

[OW19a] M. Osama and A. Wijs. Parallel SAT Simplification on GPU Architectures. In *Proc. of TACAS (Apr. 2019), Prague, Czech Republic*, volume 11427 of *LNCS*, pages 21–40. Springer. doi:10.1007/978-3-030-17462-0_2.

[OW19b] M. Osama and A. Wijs. SIGmA: GPU Accelerated Simplification of SAT Formulas. In *Proc. of IFM (Dec. 2019), Bergen, Norway*, volume 11918 of *LNCS*, pages 514–522. Springer. doi:10.1007/978-3-030-34968-4_29.

[OW20] M. Osama and A. Wijs. Multiple Decision Making in Conflict-Driven Clause Learning. In *Proc. of ICTAI (Nov. 2020), Baltimore, USA*, pages 161–169. IEEE. doi:10.1109/ICTAI50040.2020.00035.

[OW21a] M. Osama and A. Wijs. GPU Acceleration of Bounded Model Checking with ParaFROST. In *Proc. of CAV (Jul. 2021), USA*, volume 12760 of *LNCS*, pages 447–460. Springer. doi:10.1007/978-3-030-81688-9_21.

[OW21b] M. Osama and A. Wijs. Multiple Decision Making in CDCL SAT Solvers. 2021. To be submitted.

[OW21c] M. Osama and A. Wijs. ParaFROST at the SAT Race 2021. In *Proc. of SC (2021)*, volume B-2021-1 of *Report Series B*, pages 32–34. University of Helsinki. URL `http://hdl.handle.net/10138/333647`.

[OWB21a] M. Osama, A. Wijs, and A. Biere. Certified SAT Solving with GPU Accelerated Inprocessing. 2021. To be submitted.

[OWB21b] M. Osama, A. Wijs, and A. Biere. SAT Solving with GPU Accelerated Inprocessing. In *Proc. of TACAS (Mar. 2021), Luxembourg*, volume 12651 of *LNCS*, pages 133–151. Springer. doi:10.1007/978-3-030-72016-2_8.

[PBB21] A. Pakonen, I. Buzhinsky, and K. Björkman. Model Checking Reveals Design Issues Leading to Spurious Actuation of Nuclear Instrumentation and Control Systems. *Reliability Engineering and System Safety*, 205:107237, 2021. doi:10.1016/j.ress.2020.107237.

[PD07] K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *Proc. of SAT (May 2007), Lisbon, Portugal*, volume 4501 of *LNCS*, pages 294–299. Springer. doi:10.1007/978-3-540-72788-0_28.

[PDFP15] A. D. Palù, A. Dovier, A. Formisano, and E. Pontelli. CUD@SAT: SAT solving on GPUs. *Journal of Experimental and Theoretical Artificial Intelligence*, 27(3):293–316, 2015. doi:10.1080/0952813X.2014.954274.

[PHS08] C. Piette, Y. Hamadi, and L. Sais. Vivifying Propositional Clausal Formulae. In *Proc. of ECAI (Jul. 2008), Patras, Greece*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 525–529. IOS Press. doi:10.3233/978-1-58603-891-5-525.

[PMP15] Q. Phan, P. Malacaria, and C. S. Pasareanu. Concurrent Bounded Model Checking. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–5, 2015. doi:10.1145/2693208.2693240.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proc. of FOCS (1977), Rhode Island, USA*, pages 46–57. IEEE. doi:10.1109/SFCS.1977.32.

[PSM21] N. Prevot, M. Soos, and K. S. Meel. Leveraging GPUs for Effective Clause Sharing in Parallel SAT Solving. In *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 471–487, Cham. Springer International Publishing. doi:10.1007/978-3-030-80223-3_32.

[SBS96] P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, 1996. doi:10.1109/43.536723.

[SG99] J. P. M. Silva and T. Glass. Combinational Equivalence Checking Using Satisfiability and Recursive Learning. In *Proc. of DATE (Mar. 1999), Munich, Germany*, pages 145–149. IEEE / ACM. doi:10.1109/DATE.1999.761110.

[SHGO10] S. Sengupta, M. J. Harris, M. Garland, and J. D. Owens. Efficient Parallel Scan Algorithms for Manycore GPUs. In *Proc. of Scientific Computing with Multicore and Accelerators*, CRC Computational Science Series, pages 413–442. CRC Press, 2010. doi:10.1201/b10376-29.

[SK93] B. Selman and H. A. Kautz. An Empirical Study of Greedy Local Search for Satisfiability Testing. In *Proc. of AAAI (Jul. 1993), Washington, USA*, pages 46–51. AAAI Press / The MIT Press.

[SKB+17] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller. Incremental bounded model checking for embedded software. *Formal Aspects of Computing*, 29(5):911–931, 2017. doi:10.1007/s00165-017-0419-1.

[SM19] M. Springer and H. Masuhara. Massively parallel GPU memory compaction. In *Proc. of ISMM (Jun. 2019), Phoenix, USA*, pages 14–26. ACM. doi:10.1145/3315573.3329979.

[SP04] S. Subbarayan and D. K. Pradhan. NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT instances. In *Proc. of SAT (May 2004), Vancouver, Canada*. doi:https://doi.org/10.1007/11527695_22.

[SS99] J. P. M. Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. doi:10.1109/12.769433.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Proc. of FMCAD (Nov. 2000), Austin, USA*, volume 1954 of *LNCS*, pages 108–125. Springer. doi:10.1007/3-540-40922-X_8.

[Str01] O. Strichman. Pruning Techniques for the SAT-Based Bounded Model Checking Problem. In *Proc. of CHARME (Sept. 2001), Scotland, UK*, volume 2144 of *LNCS*, pages 58–70. Springer. doi:10.1007/3-540-44798-9_4.

[SVL+16] J. Shen, A. L. Varbanescu, Y. Lu, P. Zou, and H. Sips. Workload Partitioning for Accelerating Applications on Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2766–2780, 2016. doi:10.1109/TPDS.2015.2509972.

[Tse83] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. doi:10.1007/978-3-642-81955-1_28.

[VB01] M. N. Velev and R. E. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. In *Proc. of DAC (Jun. 2001), Las Vegas, USA*, pages 226–231. ACM. doi:10.1145/378239.378469.

[vdTRH11] P. van der Tak, A. Ramos, and M. Heule. Reusing the Assignment Trail in CDCL Solvers. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):133–138, 2011. doi:10.3233/sat190082.

[vEGH$^+$21] J. van Eerd, J. F. Groote, P. Hijma, J. Martens, and A. Wijs. Term Rewriting on GPUs. In *Proc. of FSEN (May 2021), Virtual Event, Revised Selected Papers*, volume 12818 of *LNCS*, pages 175–189. Springer. doi:10.1007/978-3-030-89247-0_12.

[Wal99] T. Walsh. Search in a Small World. In *Proc. of IJCAI (Aug. 1999), Stockholm, Sweden*, pages 1172–1177. Morgan Kaufmann.

[WB12] A. Wijs and D. Bosnacki. Improving GPU sparse matrix-vector multiplication for probabilistic model checking. In *Proc. of SPIN (Jul. 2012), Oxford, UK*, volume 7385 of *LNCS*, pages 98–116. Springer. doi:10.1007/978-3-642-31759-0_9.

[WB14] A. Wijs and D. Bosnacki. GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs. In *Proc. of TACAS (Apr. 2014), Grenoble, France*, volume 8413 of *LNCS*, pages 233–247. Springer. doi:10.1007/978-3-642-54862-8_16.

[WHH14] N. Wetzler, M. Heule, and W. A. Hunt. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *Proc. of SAT (Jul. 2014), Vienna, Austria*, volume 8561 of *LNCS*, pages 422–429. Springer. doi:10.1007/978-3-319-09284-3_31.

[Wij15] A. Wijs. GPU Accelerated Strong and Branching Bisimilarity Checking. In *Proc. of TACAS (Apr. 2015), London, UK*, volume 9035 of *LNCS*, pages 368–383. Springer. doi:10.1007/978-3-662-46681-0_29.

[WKS01] J. Whittemore, J. Kim, and K. A. Sakallah. SATIRE: A New Incremental Satisfiability Engine. In *Proc. of DAC (Jun. 2001), Las Vegas, USA*, pages 542–545. ACM. doi:10.1145/378239.379019.

[WNH09] S. Wieringa, M. Niemenmaa, and K. Heljanko. Tarmo: A Framework for Parallelized Bounded Model Checking. In *Proc. of PDMC (Nov. 2009), Eindhoven, The Netherlands*, volume 14 of *EPTCS*, pages 62–76. doi:10.4204/EPTCS.14.5.

[YIMO15] H. A. Youness, A. Ibraheim, M. Moness, and M. Osama. An Efficient Implementation of Ant Colony Optimization on GPU for the Satisfiability Problem. In *Proc. of PDP (Mar. 2015), Turku, Finland*, pages 230–235. IEEE. doi:10.1109/PDP.2015.59.

[YNH$^+$16] J. Yu, R. Nane, A. Haron, S. Hamdioui, H. Corporaal, and K. Bertels. Skeleton-based design and simulation flow for computation-in-memory architectures. In *Proc. of NANOARCH (Jul. 2016), Beijing, China*, pages 165–170. ACM. doi:10.1145/2950067.2950071.

[YOH$^+$20] H. Youness, M. Osama, A. Hussein, M. Moness, and A. M. Hassan. An Effective SAT Solver Utilizing ACO Based on Heterogenous Systems. *IEEE Access*, 8:102920–102934, 2020. doi:10.1109/ACCESS.2020.2999382.

[Zha05] L. Zhang. On Subsumption Removal and On-the-Fly CNF Simplification. In *Proc. of SAT (Jun. 2005), St. Andrews, UK*, volume 3569 of *LNCS*, pages 482–489. Springer. doi:10.1007/11499107_42.

[ZMMM01] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proc. of ICCAD (Nov. 2001). IEEE/ACM Digest of Technical Papers*, pages 279–285. doi:10.1109/ICCAD.2001.968634.

Summary

GPU Enabled Automated Reasoning

Testing can be effective to detect the presence of bugs in system designs, but it cannot prove their absence. One technique that can provide worthful feedback on the correctness of system designs is Model Checking (MC). It involves exhaustively analysing a system design to determine whether it satisfies desirable functional specifications. Common examples are verifying autonomous vehicles, medical imaging, microprocessor designs, and many more. For these applications, it is vital to make sure they are bug free, and always behave (correctly) w.r.t the functional properties. MC is used in automated fashion to catch any potential bugs as early as possible–preferably at the design phase–to make the necessary modifications quickly and cost-effectively. However, it is computationally very demanding. Bounded Model Checking (BMC) is currently a contemporary symbolic technique that can analyse large designs in reasonable time. BMC determines whether a model satisfies a certain property expressed in temporal logic, by translating the model checking problem to a propositional Satisfiability (SAT) problem, for instance.

In this thesis, we investigate how GPUs can be employed effectively for reasoning on Satisfiability and its direct application on BMC. GPUs offer great potential for parallel computation, while keeping power consumption low. However, not all types of computation can trivially be performed on GPUs. In most applications, the algorithms need to be entirely redesigned.

The first part focuses on the simplifications of SAT formulas prior to the solving process (preprocessing) and how they can applied vigorously within SAT solving (inprocessing). Simplification is a strategy that leads to a drastic prune of the formula size, and the search space. The parallelisation of simplifications has been always a major challenge due to the strong dependency between

variables in a SAT formula. Hence, we proposed the Least Constrained Variable Elections (LCVE) algorithm, which is responsible for scheduling a set of mutually independent variables that are eligible for parallel simplification. Consequently, we introduced the first GPU algorithms for various eliminations, which are essential in any SAT solver. In addition, we presented a new elimination method called Eager Redundancy Elimination (ERE) and its GPU implementation to further remove redundant clauses from SAT problems. All preprocessing techniques were implemented in a tool called SIGmA. Next, we presented a new SAT solver (ParaFROST) which interleaves the search with simplifications. Inprocessing has proven to be powerful in modern SAT solvers, particularly when applied on formulas encoding software and hardware verification problems. The new solver is hybrid, capable of running the parallel part on the GPU while the actual solving will run sequentially on the CPU. We discussed the design aspects of the data structures and the memory management of our parallel implementations, leading to substantial improvements in execution performance.

The second part concerns the solving part. The standard decision making step in the Conflict-Driven Clause Learning (CDCL) algorithm, selects one decision at a time to explore the search space. We extended this step in CDCL with Multiple Decision Making (MDM) strategy. It has the ability to make thousands, even millions of multiple decisions that can be propagated at once. Doing so may lessen the number of conflicts that arise from making bad assignments, which in turn prunes the search space. Several optimizations have been augmented to MDM including local search, VSIDS, and VMTF decision heuristics to improve the quality of the picked decisions. Overall, MDM allowed ParaFROST to solve a significant number of SAT problems in less time compared to the state of the art.

Finally, in the third part, we combined all the above achievements to accelerate BMC with our solver ParaFROST. We found that SAT formulas stemming from bounded model checkers such as CBMC have enormous redundancies in variables and clauses. This amount of redundancy takes huge memory space on the GPU. Thus, we compacted further the data structure in ParaFROST to reduce the memory consumption. Additionally, the feature of *incremental* SAT solving has been added to ParaFROST in order to support SAT-based k-induction BMC. To this end, ParaFROST was integrated into CBMC model checker using a configurable interface called GPU4BMC. We observed that program verification via CBMC can be accelerated effectively with ParaFROST. For example, compared to ParaFROST (noGPU) (both simplifications and the search run on the CPU), ParaFROST accelerated multiple program verification tasks up to 27$\times$ faster.

Samenvatting

Automatisch Redeneren met GPUs

Testen kan effectief zijn om de aanwezigheid van bugs in systeemontwerpen te detecteren, maar het kan hun afwezigheid niet bewijzen. Een techniek die waardevolle feedback kan geven over de juistheid van systeemontwerpen is *modelverificatie* (MC). Het omvat een grondige analyse van een systeemontwerp om te bepalen of het voldoet aan de gewenste functionele specificaties. Veelvoorkomende voorbeelden zijn het verifiëren van autonome voertuigen, medische beeldvorming, microprocessorontwerpen en nog veel meer. Voor deze toepassingen is het essentieel om ervoor te zorgen dat ze vrij zijn van bugs en zich altijd (correct) gedragen met betrekking tot de functionele eigenschappen. MC wordt geautomatiseerd gebruikt om mogelijke bugs zo vroeg mogelijk op te sporen–bij voorkeur in de ontwerpfase–om de noodzakelijke wijzigingen snel en kosteneffectief door te voeren. Het is echter rekenkundig zeer veeleisend. *Begrensde modelverificatie* (BMC) is momenteel een hedendaagse symbolische techniek waarmee grote ontwerpen binnen een redelijke tijd kunnen worden geanalyseerd. BMC bepaalt of een model voldoet aan een bepaalde eigenschap uitgedrukt in temporele logica, bijvoorbeeld door het modelcontroleprobleem te vertalen naar een *propositioneel vervulbaarheidsprobleem* (SAT).

In dit proefschrift onderzoeken we hoe GPU's effectief kunnen worden gebruikt om te redeneren over vervulbaarheid en de directe toepassing ervan op BMC. GPU's bieden een groot potentieel voor parallelle berekeningen, terwijl het stroomverbruik laag blijft. Niet alle soorten berekeningen kunnen echter triviaal worden uitgevoerd op GPU's. In de meeste toepassingen moeten de algoritmen volledig opnieuw worden ontworpen.

Het eerste deel richt zich op het vereenvoudigen van SAT formules voorafgaand aan het oplossingsproces (preprocessing) en hoe vereenvoudigingen krachtig kunnen worden toegepast binnen het SAT oplossingsproces (inprocessing). Vereenvoudiging is een strategie die kan leiden tot een drastische reductie in de formuleomvang en de zoekruimte. De parallellisatie van vereenvoudigingen is altijd een grote uitdaging geweest vanwege de sterke afhankelijkheid tussen variabelen in een SAT formule. Daarom hebben we het *minst beperkte variabele selectie* (LCVE)-algoritme voorgesteld, dat verantwoordelijk is voor het identificeren van een reeks onderling onafhankelijke variabelen die in aanmerking komen voor parallelle vereenvoudiging. Op basis daarvan hebben we de eerste GPU algoritmen geïntroduceerd voor verschillende eliminaties, die essentieel zijn in elke SAT oplosser. Daarnaast presenteerden we een nieuwe eliminatiemethode genaamd *eager redundantie eliminatie* (ERE) en de GPU implementatie ervan om overtollige clausules van SAT problemen verder te verwijderen. Alle voorbewerkingstechnieken zijn geïmplementeerd in een tool genaamd SIGmA. Vervolgens presenteerden we een nieuwe SAT oplosser (ParaFROST) die vereenvoudigingen toepast tijdens de zoekopdracht. Inprocessing heeft bewezen krachtig te zijn in moderne SAT oplossers, vooral wanneer toegepast op formules die software- en hardwareverificatieproblemen coderen. De nieuwe oplosser is hybride, en is in staat om het parallelle deel op de GPU uit te voeren terwijl het daadwerkelijke oplossen op de CPU plaatsvindt. We bespraken de ontwerpaspecten van de datastructuren en het geheugenbeheer van onze parallelle implementaties, wat leidde tot substantiële verbeteringen in de uitvoeringsprestaties.

Het tweede deel adresseert het oplossen. De standaard besluitvormingsstap in het *conflictgestuurd leren van clausules* (CDCL) algoritme, selecteert één beslissing tegelijk om de zoekruimte te verkennen. We hebben deze stap in CDCL uitgebreid met de *meervoudige besluitvorming* (MDM) strategie. Het heeft de mogelijkheid om duizenden, zelfs miljoenen beslissingen te nemen die tegelijkertijd kunnen worden doorgevoerd. Dit kan het aantal conflicten verminderen dat voortkomt uit het maken van slechte beslissingen, wat op zijn beurt de zoekruimte inperkt. Er zijn verschillende optimalisaties toegevoegd aan MDM, waaronder lokaal zoeken, VSIDS en VMTF beslissingsheuristieken om de kwaliteit van de gekozen beslissingen te verbeteren. Over het algemeen stelde MDM ParaFROST in staat om een aanzienlijk aantal SAT problemen in minder tijd op te lossen in vergelijking met de state of the art.

Tenslotte hebben we in het derde deel alle bovenstaande prestaties gecombineerd om BMC te versnellen met onze oplosser ParaFROST. We ontdekten dat SAT formules die voortkomen uit begrensde modelverificatie tools zoals CBMC enorme redundantie in variabelen en clausules hebben. Deze hoeveelheid redundantie neemt enorm veel geheugenruimte in beslag op de GPU. Daarom hebben we de datastructuur in ParaFROST verder gecomprimeerd om het geheugenverbruik te verminderen. Bovendien is *incremental* SAT oplossen toegevoegd aan ParaFROST om op SAT gebaseerde k-inductie BMC te ondersteunen. Hiervoor werd ParaFROST geïntegreerd in de CBMC model checker met behulp van een configureerbare interface genaamd GPU4BMC. We hebben geconstateerd dat programmaverificatie via CBMC effectief versneld kan worden met ParaFROST. In vergelijking met bijvoorbeeld ParaFROST (noGPU) (waarin zowel het vereenvoudigingen als het zoeken op de CPU gebeurt), versnelde ParaFROST meerdere programmaverificatietaken tot wel 27×.

Curriculum Vitae

Muhammad Osama Mahmoud received the B.Sc. degree with distinction and honors in Computers and Systems Engineering from the Faculty of Engineering, Minia University, Minia, Egypt, in 2011. Shortly after, he joined the Department of Computers and Systems Engineering, Minia university, as a Teaching Assistant. He received his M.Sc. degree from the same university in Computer Engineering field. His Master's topic was about implementing an efficient GPU implementation of Ant Colony Optimization for solving Satisfiability problem.

He received several awards from NVIDIA corporation for his contributions in CUDA education and research including a GPU lab that is worth of $10,000 value. From 2014, he was also the Principal Investigator of the NVIDIA educational GPU-computing Lab, Minia University. Before working on his current Ph.D. project, he has developed a real-time wind turbine emulator on a heterogeneous embedded system and an efficient SAT-based test generator on a GPU accelerator.

Muhammad started as a Ph.D. candidate in 2017, at the Eindhoven University of Technology in the group Software Engineering Technology. His work in GEARS project was funded by NWO and supervised by Anton Wijs and Mark van den Brand. His research focused on the development of GPU-accelerated algorithms for automated reasoning on Satisfiability and how can they be effectively deployed in bounded model checking.

Since 2012, Muhammad published over 10 articles and his interests include formal verification, parallel programming, high-performance computing, and cyber-physical systems.

Titles in the IPA Dissertation Series since 2019

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow.* Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning.* Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management.* Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers.* Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems.* Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning.* Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities.* Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis.* Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Griffioen. *A Unit-Aware Matrix Language and its Application in Control and Auditing.* Faculty of Science, UvA. 2019-11

A.A. Sawant. *The impact of API evolution on API consumers and how this can be affected by API producers and language designers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12

W.H.M. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views.* Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Co-simulation Construction.* Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming.* Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models.* Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems.* Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing.* Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations.* Faculty of Mathematics and Computer Science, TU/e. 2020-09

D.Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security.* Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic.* Faculty of Sciences, Department of Computer Science, VUA. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components.* Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management.* Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS.* Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning.* Faculty of Mathematics and Computer Science, TU/e. 2022-02