Formal Methods in System Design
https://doi.org/10.1007/s10703-023-00432-z

ORIGINAL ARTICLE

Certified SAT solving with GPU accelerated inprocessing

Muhammad Osama[1] · **Anton Wijs**[1] · **Armin Biere**[2]

Received: 23 December 2021 / Accepted: 7 June 2023

Abstract
Since 2013, the leading SAT solvers in SAT competitions all use inprocessing, which, unlike preprocessing, interleaves search with simplifications. However, inprocessing is typically a performance bottleneck, in particular for hard or large formulas. In this work, we introduce the first attempt to parallelize inprocessing on GPU architectures. As one of the main challenges in GPU programming is memory locality, we present new compact data structures and devise a data-parallel garbage collector. It runs in parallel on the GPU to reduce memory consumption and improve memory locality. Our new parallel variable elimination algorithm is roughly twice as fast as previous work. Moreover, we augment the variable elimination with the first parallel algorithm for functional dependency extraction in an attempt to find more logical gates to eliminate that cannot be found with syntactic approaches. We present a novel algorithm to generate clausal proofs in parallel to validate all simplifications running on the GPU besides the CDCL search, giving high credibility to our solver and its use in critical applications such as model checkers. In experiments, our new solver ParaFROST solves numerous benchmarks faster on the GPU than its sequential counterparts. With functional dependency extraction, inprocessing in ParaFROST was more effective in reducing the solving time. Last but not least, all proofs generated by ParaFROST were successfully verified.

Keywords SAT solving · Inprocessing · GPUs · Clausal proofs · Functional dependency extraction

1 Introduction

During the past decade, SAT solving has been used extensively in a plethora of applications, such as combinational equivalence checking [53], automatic test pattern generation [43, 57],

✉ Muhammad Osama
o.m.m.muhammad@tue.nl

Anton Wijs
a.j.wijs@tue.nl

Armin Biere
biere@cs.uni-freiburg.de

1 Eindhoven University of Technology, Eindhoven, The Netherlands

2 Albert-Ludwigs-Universität, Freiburg, Germany

Published online: 02 August 2023

automatic theorem proving [14, 33, 60], and symbolic model checking [6, 13, 15, 16, 31]. Simplifying SAT problems prior to solving them has proven its effectiveness in modern conflict-driven clause learning (CDCL) SAT solvers [5, 8, 19], particularly when applied on real-world applications relevant to software and hardware verification [18, 22, 26, 27, 47].

Since 2013, simplification techniques [9, 18, 21, 25, 58] are also used periodically *during* SAT solving, which is known as *inprocessing* [4, 5, 8, 27]. Applying inprocessing iteratively to large problems can be a performance bottleneck in SAT solving procedure, or even increase the size of the formula in case of bounded variable elimination [9, 18, 58], negatively impacting the solving time.

Graphics processors (GPUs) have become more and more attractive for general-purpose computing with the availability of programming models such as Open Computing Language (OpenCL) [38] and the Compute Unified Device Architecture (CUDA) [39]. On the one hand, OpenCL is portable to both CPUs and GPUs manufactured by different vendors such as AMD and NVIDIA. On the other hand, CUDA is designed by NVIDIA to make the optimum use of their hardware capabilities in accelerating applications that are computationally intensive w.r.t. data processing. For instance, we have applied CUDA to accelerate explicit-state model checking [12, 65–67], bisimilarity checking [64], wind turbine emulation [37], term rewriting [61, 62], metaheuristic SAT solving [68, 69], SAT-based test generation [43] and bounded model checking [42, 47]. Recently, we introduced SIGmA [44, 45] as the first SAT simplification preprocessor to utilize GPUs.

In [48], we introduced 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, while in inprocessing, this is done periodically during the solving. A new dynamically expanded data structure was introduced for clauses supporting both 32-bit [19] and 64-bit references with a minimum of 16 bytes per clause. In addition, a new parallel garbage collector was presented, tailored for GPU inprocessing.

In our latest work [47], we have worked on compacting the data structure used in PARAFROST as much as possible, while still allowing for the application of effective solving optimisations. Furthermore, we introduced the *memory-aware* Bounded Variable Elimination (BVE), 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 [48] for Bounded Model Checking (BMC).

Contributions The current article is an extension of our earlier papers [47, 48], to which we have added the following original contributions:

- ⋆ The BVE in PARAFROST can reduce the number of added clauses by detecting in parallel logical gates that are syntactically translated to Conjunctive Normal Form (CNF) using Tseitin encoding [59]. Such definition of x is written as $x \leftrightarrow f(v_1, \ldots, v_n)$. The simplest example is the AND gate $x \leftrightarrow v_1 \wedge v_2$. However, this approach fails to recognize *irregular* gates not encoded as simple gate types such as *AND*, *XOR* or *If-Then-Else* gates. In this work, we provide a semantic way using functional dependency extraction to detect any gate definition in parallel that could not be found by the syntactic approach. We use function tables and binary magic numbers to compress the variables to bit-vectors [29]. Moreover, the number of added clauses is further reduced by finding and resolving a Minimal Unsatisfiable Set (MUS) of clauses representing the gates [34].
- ⋆ While the impact of accelerating these procedures has been demonstrated in [47, 48], their correctness in refuting a formula has not yet been addressed, especially if used in critical applications such as BMC [47]. 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. For this reason, we propose an efficient parallel approach to generate clausal proofs for the GPU-accelerated simplifications in DRAT (Deletion Resolution Asymmetric Tautology) format [24, 63] with two goals: the proof should be compact and not to pose an overhead to the GPU solver.

⋆ We provide a comprehensive evaluation of the proposed algorithms with and without clausal proof generation. Furthermore, PARAFROST with the new extensions is compared to the state-of-the-art sequential solvers developed by the third author.

2 Preliminaries

All SAT formulas in this article are in CNF format. 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 all variables is $var(\mathbb{L})$. With $\mathbb{L}(\mathcal{S})$, we refer to 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 $C_1 \wedge \ldots \wedge C_m$. Further, 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 set of clauses $\mathcal{S}_\ell|_{\neg\ell}$ is called the set of *co-factors* of $\mathcal{S}_\ell$ and is defined as $\mathcal{S}_\ell|_{\neg\ell} = \{C \backslash \{\ell\} \mid C \in \mathcal{S}_\ell\}$.

In this article, 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*.

The GPU-accelerated inprocessing is integrated with the CDCL [46, 54] search algorithm. One important feature of CDCL is to learn from previous assignments to prune the search space and make better decisions in the future. This learning process involves the periodic adding of new *learnt* clauses to the input formula while CDCL is running. We consider clauses to be either `LEARNT` or `ORIGINAL` (*redundant* and *irredundant* in [27] and in the SAT solver CADICAL [8]). 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 beginning. Moreover, 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 one. The number of distinct levels in a clause at which literals are assigned is called the *literal block distance* (LBD) or glucose level of that clause [2].

2.1 Bounded variable elimination

BVE can remove variables completely from the CNF formula by trivially eliminating *pure literals* [17], applying the *resolution rule* [17, 30, 58] or *gate-equivalence reasoning* [18, 32, 49].

Definition 1 (*Pure literal*) For a formula $\mathcal{S}$, a literal ℓ is called *pure* iff $\mathcal{S}_{\neg\ell} = \emptyset$. A pure literal can be eliminated from $\mathcal{S}$, resulting in the new formula $\mathcal{S}' = \mathcal{S} \backslash \mathcal{S}_\ell$.

Definition 2 (*Resolution rule*) Let us consider 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

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

The result of applying the rule is called the *resolvent* [58]. Moreover, The $\otimes_x$ operator can be extended to resolve sets of clauses w.r.t. variable x.

Definition 3 (*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

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

Notice that learnt clauses can be ignored [27] (i.e., in practice, it is not effective to apply resolution on learnt clauses). The last condition ensures that a resolvent is not a tautology.

Definition 4 (*Tautology*) A clause C is called a *tautology* (i.e. $C \not\models \top$) iff $\exists x.\{x, \neg x\} \subseteq C$.

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

In gate-equivalence reasoning, we substitute eliminated variables with 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 [18, 27]. 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 (e.g. AND), 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 [27].

Example 1 Consider the following formula:

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

The first three clauses in the formula above together capture the AND gate ($x \leftrightarrow a \wedge b$) and the last three clauses capture the OR gate ($y \leftrightarrow d \vee e$), hence resolving the fourth clause with the second and the third clauses yield the resolvents $\{a, c\}$ and $\{b, c\}$. Similarly, eliminating y results in $\{f, d, e\}$.

In this article, we focus on finding definitions for irregular gates by checking the unsatisfiability of the co-factors formula $\{\mathcal{S}_x|_{\neg x} \cup \mathcal{S}_{\neg x}|_x\}$. In [5, 9], a BDD-based approach is used to solve the co-factors. In this work, we replace the BDD structure with a function table (bit-vector) encoding clausal core of the co-factors. Checking the bit-vector of the latter can be done effectively on the GPU. The clausal core is mapped back to the original gate clauses G_x and $G_{\neg x}$ by adding back x and $\neg x$, respectively. Then, the set of resolvents $R_x = \mathcal{S}_x \otimes \mathcal{S}_{\neg x}$ is reduced to $\{\{G_x \otimes G_{\neg x}\}, \{G_x \otimes H_{\neg x}\}, \{G_{\neg x} \otimes H_x\}\}$, dropping the redundant part $\{H_x \otimes H_{\neg x}\}$. In contrast to gate substitution (see Example 1), the resolvents $\{G_x \otimes G_{\neg x}\}$ are not necessarily tautological.

Example 2 Consider the following formula:

$$\{\{\neg x, \neg a, \neg b\}, \{\neg x, a\}, \{x, b\}, \{x, \neg a\}\}$$

The co-factors formula is $\{\{\neg a, \neg b\}, \{a\}, \{b\}, \{\neg a\}\}$ in which two unsatisfiable cores are extractable: $\{\{\neg a, \neg b\}, \{a\}, \{b\}\}$ or the minimal one $\{\{a\}, \{\neg a\}\}$. If the former is considered, the original gate clauses G_x and $G_{\neg x}$ are identified by adding back x resp. $\neg x$ to the core as follows:

$$\{\underbrace{\{\neg x, \neg a, \neg b\}, \{\neg x, a\}}_{G_{\neg x}}, \underbrace{\{x, b\}}_{G_x}, \underbrace{\{x, \neg a\}}_{H_x}\}$$

Thus, the new resolvents are $\{a, b\}$ and $\{\neg a, \neg b\}$. The reader should be aware that the minimal core $\{\{a\}, \{\neg a\}\}$ would produce the same resolvents in this example.

2.2 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 be removed [18]. If C_2 is a `LEARNT` clause, it must be considered as `ORIGINAL` in the future, to prevent deleting it during learnt clause reduction [8].

Definition 5 (*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 $C_1 = C_1' \cup \{x\}$, $C_2 = C_2' \cup \{\neg x\}$, and $C_2' \subseteq C_1'$. In that case, x can be 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.

Example 3 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 from $\mathcal{S}$ and the simplified formula becomes $\{\{b, c\}, \{\neg a, b\}\}$.

2.3 Eager redundancy elimination

ERE is a new elimination technique that we proposed in [48], which repeats the following until a fixpoint has been reached: for a given formula $\mathcal{S}$ and clauses $C_1 \in \mathcal{S}, C_2 \in \mathcal{S}$ with $x \in C_1$ and $\neg x \in C_2$ for some variable x, if there exists a clause $C \in \mathcal{S}$ for which $C \equiv C_1 \otimes_x C_2$, then let $\mathcal{S} := \mathcal{S} \backslash \{C\}$. In this method, we restrict removing C to the condition

$$C \text{ is } \texttt{LEARNT} \vee (C_1 \text{ is } \texttt{ORIGINAL} \wedge C_2 \text{ is } \texttt{ORIGINAL})$$

That is, if C is `LEARNT` or the resolved clauses (C_1, C_2) and C are `ORIGINAL`, then C is called a *redundancy* and can be removed.

Note that this method is entirely different from *Asymmetric Tautology Elimination* in [25]. The latter requires adding so-called hidden literals to all clauses to check which is a hidden tautology. ERE can operate on learnt clauses and does not require literals addition, making it more effective and adequate to data parallelism.

Example 4 Consider $\mathcal{S} = \{\{a, \neg c\}, \{c, \neg b\}, \{\neg d, \neg c\}, \{\neg b, a\}, \{a, d\}\}$. Resolving the first two clauses gives the resolvent $\{a, \neg b\}$ which is equivalent to the fourth clause in $\mathcal{S}$. Also,

resolving the third clause with the last clause yields $\{a, \neg c\}$ which is equivalent to the first clause in $\mathcal{S}$. ERE can remove either $\{a, \neg c\}$ or $\{a, \neg b\}$ but not both.

The effectiveness of ERE relies on the fact that *resolution is unit propagation*. Consider the previous example, if the solver sets c to true or false, then either a or $\neg b$ can be directly assigned as implications, forcing the solver to propagate the fourth clause $\{\neg b, a\}$ in which the same implications are disjoined. Hence, removing the latter from the formula, saves unnecessary overhead during unit propagation.

2.4 DRAT clausal proof

DRAT is a clausal proof format that can be constructed from smaller lemmas. Each line of the proof stores a lemma which is either a sequence of literals terminated by 0 or a deletion instruction (a prefix represented by the character d). The Lemmas are emitted to an output file in DIMACS format [28] and are validated using both the Resolution Asymmetric Tautology (RAT) [24] and Reverse Unit Propagation (RUP) [23] checks via the external proof checker DRAT-TRIM [63]. The mathematical notions behind RAT and RUP are out of this article scope. Both learnt clauses and resolvents are considered lemma additions. Clause deletions are not essential for the proof to succeed; however, they help reduce the computation time in verifying the proof. A proof terminating with an empty clause (i.e. a line containing only a zero), declares the input formula is UNSAT.

3 GPU architecture and data structures

3.1 GPU architecture

Since 2007, NVIDIA has been developing a parallel computing platform called CUDA [39] that allows developers to use GPU resources for general purpose processing. A GPU contains multiple streaming multiprocessors (SMs), each SM consisting of an array of streaming processors (SPs) or cores. Every SM can execute multiple threads grouped together in 32-thread scheduling units called *warps*.

GPU Kernel A GPU computation can be launched in a program by the *host* (CPU side of a program) by calling a GPU function called a *kernel*, which is executed by the *device* (GPU side of a program). When a kernel is called, it is specified how many threads need to execute it. These threads are partitioned into thread *blocks* of up to 1,024 threads (or 32 warps). Each block is assigned to an SM. All threads together form a *grid*. Threads and blocks can be indexed by a one-dimensional, two-dimensional, or three-dimensional unique identifier (ID) accessible within the kernel. By using this ID, we can achieve that different threads in the same block work on different data. A hardware warp scheduler evenly distributes the launched blocks to the available SMs.

We express a thread dimension with a bold italic font $\boldsymbol{dimension}$. For example, threads or blocks can be launched in the $\boldsymbol{x}$ or $\boldsymbol{y}$ or $\boldsymbol{z}$ dimension. Additionally, in our developed kernels, we use two conventions. First of all, with $t_{\boldsymbol{x}}$, we refer to the *block-local* ID of the working thread in $\boldsymbol{x}$. Second 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.

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.

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.

Optimisations In this work, we use *unified memory* [39] 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 virtually shared between the host and the device. This pool is accessible to both sides using the same addresses.

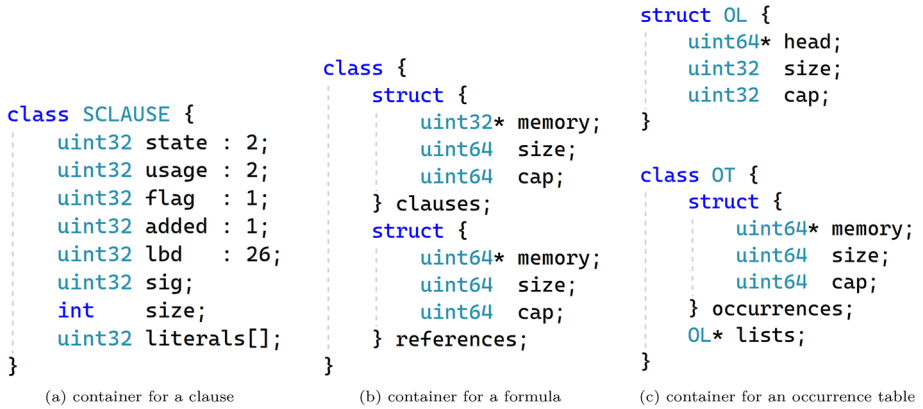
To hide the latency of global memory, ensuring that the threads perform *coalesced accesses* is one of the best practices. When the threads in a warp try to access a consecutive block of 32-bit words, their accesses are combined into a single (coalesced) memory access. Uncoalesced memory accesses can, for instance, be caused by data sparsity or misalignment.

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

3.2 Data structures

To efficiently implement inprocessing techniques (i.e. Variable-Clause Eliminations (VCE)) for GPU architectures with the ability to generate clausal proof, we adopted the clause data structure described in our latest work [47]. The new structure requires only 12 bytes of bookkeeping, compared to 16 bytes consumed by our initial design in [48] excluding literals. Figure 1a, b shows the proposed structures to store a clause (denoted by `SCLAUSE`) and the SAT formula represented in CNF form (denoted by `CNF`), respectively. The following information is stored for each clause:

- The `state` field (*2 bits*) stores if the state is `ORIGINAL`, `LEARNT` or `DELETED`.



(a) container for a clause (b) container for a formula (c) container for an occurrence table

Fig. 1 Data structures to store a formula and an occurrence table on the GPU

- The `usage` field (*2 bits*) keeps track of how many search iterations a `LEARNT` clause can still be used before it gets deleted during database reduction. As a heuristic, `LEARNT` clauses are used at most twice [8, 55].
- The `added` field (*1 bit*) is used to mark a clause as resolvent.
- The `flag` field (*1 bit*) marks the clause when it contributes to a gate (when applying substitution).
- The *literal block distance* (`lbd`) (*26 bits*) stores the number of decision levels contributing to a conflict, if there is one [2]. A maximum value of 2^{26} turns out to be sufficient. This field is updated when the clause is altered. Both `used` and `lbd` can be altered via clause *strengthening* [8] in SUB.
- 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 [18].

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 heuristics. CaDiCaL [8] uses $28 + 4t$ bytes, since it applies solving and VCE via the same clause structure. In ParaFROST, the GPU is only used for VCE; therefore, heuristic information for *probing* [35] and *vivification* [51], for instance, is irrelevant.

As implemented in MiniSat, we use the `clauses` field in `CNF` (Fig. 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 (Fig. 1c), denoted by `OT`, is created to record the 64-bit clause references for each literal in the formula. The references to all clauses in the formula are stored in a single container called `occurrences` in `OT`. The `lists` array in `OT` is created of type *occurrence list* (`OL`) to facilitate direct access to the `occurrences` memory by saving for each literal a pointer to its first occurrence. 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.

Initially, we pre-allocate 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. During variable elimination, every GPU thread checks first before adding new clauses if there is enough memory allocated using the `cap` variable in `CNF`. Variables exceeding memory boundaries are skipped from elimination. Hence, BVE is guaranteed to terminate safely no matter how much memory is allocated [47]. More on this is in Sect. 8. The `OT` memory is reallocated dynamically if needed after each variable elimination. Further, we check the amount of free GPU memory before allocation. If no memory is available, the inprocessing step is skipped and the solving continues on the CPU.

4 Parallel garbage collection

Modern sequential SAT solvers implement a *garbage collection* (GC) algorithm to reduce memory consumption and maintain data locality [2, 8, 19].

Since GPU global memory is a scarce resource and coalesced accesses are essential to hide the latency of global memory (see Sect. 3.1), we decided to develop an efficient and parallel GC algorithm for the GPU without adding overhead to the GPU computations. Figure 2 visualises 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 Fig. 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 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,[1] 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 [11]. Finally, the actual clauses are copied to their new locations in `clauses`.

[1] https://github.com/NVIDIA/cub.

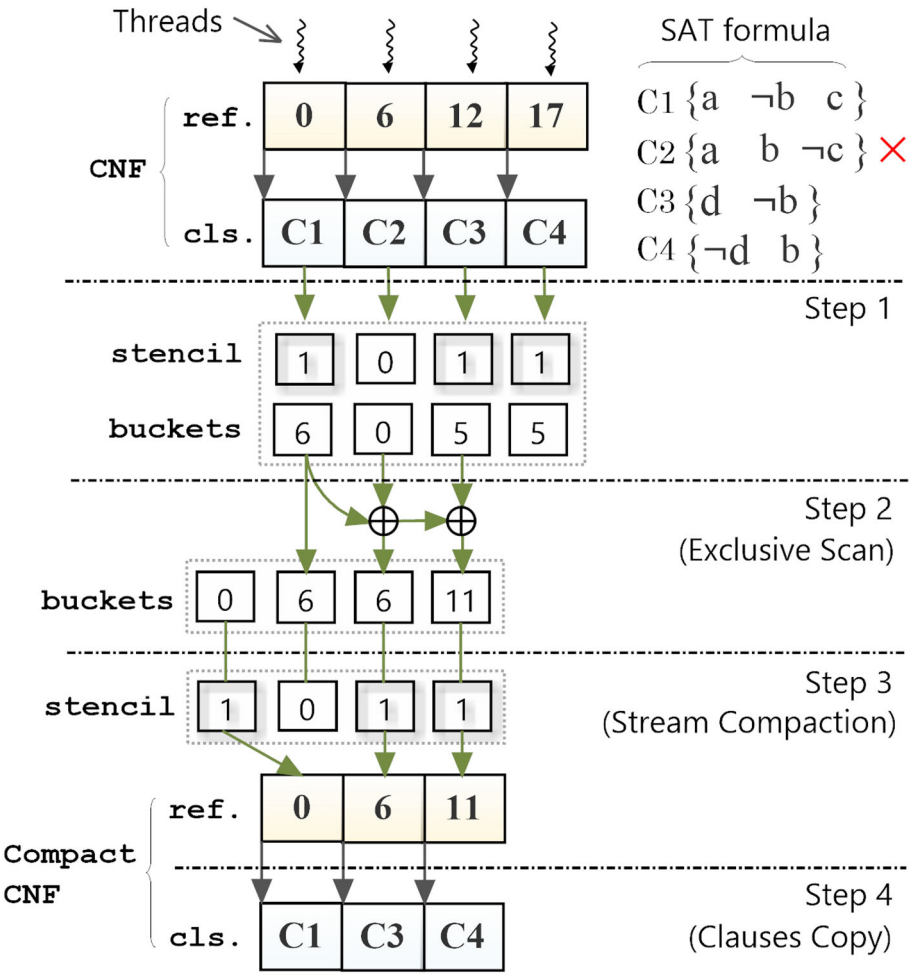


Fig. 2 An example of parallel GC on a GPU

Algorithm 1 describes in detail the GPU implementation of the parallel GC. As input, Algorithm 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. The highlighted lines in light green are executed on the GPU. 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.

Algorithm 1: Parallel Garbage Collection

```
Input  : global S_in, stencil, cindex
Input  : shared shCls, shLits
Input  : constant CB
Output : S_out
1  numRefs, numLits ← countSurvived(S_in)
2  S_out ← allocate(numRefs, numLits)
3  stencil, cindex ← computeStencil(S_in)
4  cindex ← exclusiveScan(cindex)
5  references(S_out)← compactRefs(cindex, stencil)
6  copyClauses(S_out, S_in, cindex, stencil)
7  kernel countSurvived(S_in):
8      register rCls ← 0, rLits ← 0, b ← blockDim
9      for all tid ∈ [[ 0, |S_in| ]] do in parallel
10         register C ← S_in[tid]
11         if state(C) ≠ DELETED then
12             rCls ← rCls + 1, rLits ← rLits + |C|
13         end
14     end
15     if t_x < |S_in| then
16         shCls[t_x] = rCls, shLits[t_x] = rLits
17     else
18         shCls[t_x] = 0, shLits[t_x] = 0
19     end
20     syncThreads( )
21     for b : b/2 → 32 do        // b will be blockDim/2, (blockDim/2)/2, …, 32
22         if t_x < b then
23             shCls[t_x] ← shCls[t_x] + shCls[t_x + b]
24             shLits[t_x] ← shLits[t_x] + shLits[t_x + b]
25         end
26         syncThreads( )
27     end
28     if b = 32 then shuffleReduction(shCls,shLits)
29     if t_x = 0 then
30         atomicAdd(numRefs, shCls[t_x])
31         atomicAdd(numLits, shLits[t_x])
32     end
33 end
34 kernel computeStencil(S_in):
35     for all tid ∈ [[ 0, |S_in| ]] do in parallel
36         register C ← S_in[tid]
37         if state(C) = DELETED then
38             stencil[tid] ← 0 , cindex[tid] ← 0
39         else
40             stencil[tid] ← 1 , cindex[tid] ← nbuckets(C)
41         end
42     end
43 end
44 kernel copyClauses(S_out, S_in, cindex, stencil):
45     for all tid ∈ [[ 0, |S_in| ]] do in parallel
46         if stencil[tid] then
47             register& C_dest ← (SCLAUSE &)(clauses(S_out) + cindex[tid])
48             C_dest ← S_in[tid]
49         end
50     end
51 end
```

The values *rCls* and *rLits* at line 8 will hold the current number of clauses and literals, respectively, counted by the executing thread. The vcalue *b* is used as a loop counter and initially holds the current block size. These variables are stored in the thread-local register memory. Within the loop at lines 9–14, 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 per block is in the end stored by thread 0 in the shared memory, 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 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 `&` symbol at line 47 denotes a memory reference. At line 48, the actual data in $\mathcal{S}_{in}$[*tid*] is copied to the new destination C_{dest}.

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.

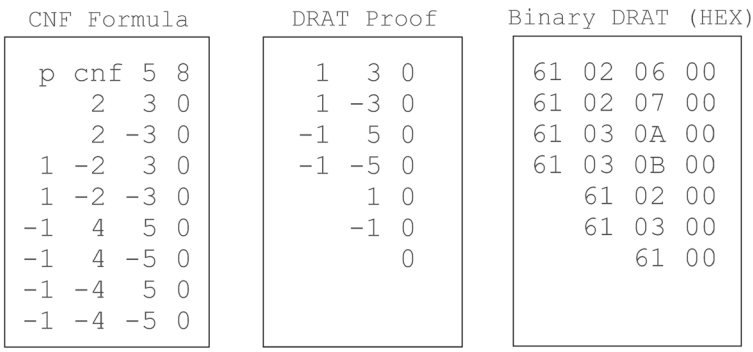


Fig. 3 An example showing the DRAT proof generated by ParaFROST

5 Proof memory management

In this work, we adopt the variable-length encoding in generating the binary DRAT proof. This format saves significant amount of memory, particularly on the GPU side (ideally, by a factor of 3 as reported in [63]). Let l and $-l$ be the positive and negatives integers to represent the literals ℓ resp. $\neg\ell$ in DIMACS format. To encode l in the binary form, it has to be mapped first to the unsigned literal l^+:

$$l^+ = \begin{cases} 2l, & \text{if } \; l \geq 0 \\ -2l+1, & \text{if } \; l < 0 \end{cases}$$

The mapped value can then be compressed into a variable-byte sequence of 7-bit words (w_i):

$$\text{WORDS}(l^+) = \sum_{i=0}^{4} w_i \times 2^{(7\times i)}$$

The $8th$ bit of a byte in this sequence indicates whether there are still more bytes to follow. Moreover, 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 the end of the lemma.

Example 5 Consider the CNF formula in Fig. 3. Eliminating the variables 2 and 4 yields the resolvents $\{1, 3\}$, $\{1, -3\}$, and $\{-1, 5\}$, $\{-1, -5\}$, respectively. Further, eliminating the variables 3 and 5 produces two unit clauses $\{1\}$ and $\{-1\}$, 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 p proof is calculated for all literals. The idea is to compute, per unsigned literal l^+, the minimum number of bytes as indicated by $\text{WORDS}(l^+)$. 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

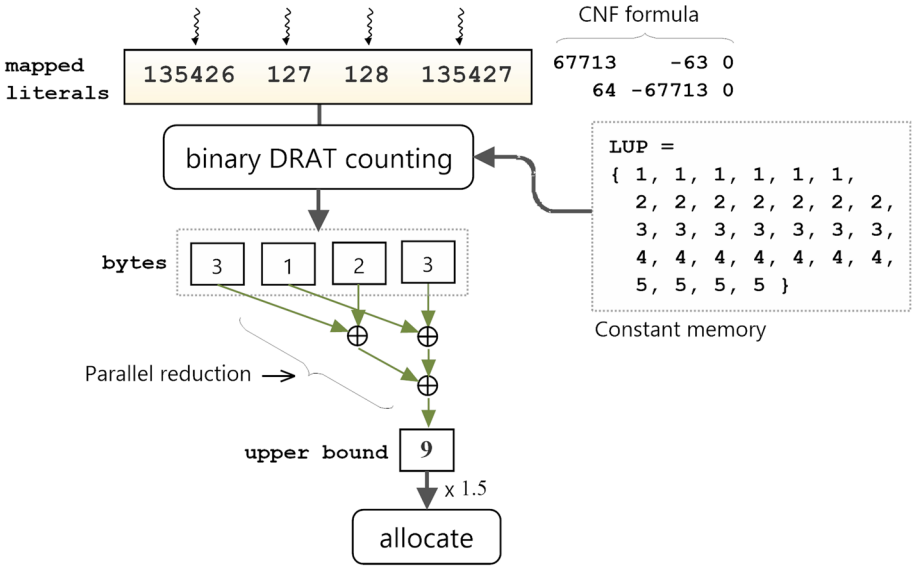


Fig. 4 An example of binary DRAT counting on a GPU

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 4 gives a working example of the parallel computation of the upper-bound for the literals 67,713, −63, 64, and −67713. Initially, the literals are mapped to the unsigned integers 135,426, 127, 128, and 135,427, 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, 135,427 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 2: GPU Proof Memory Allocator

```
   Input  : global S_in, literals, pbytes
   Input  : shared shBytes
   Input  : constant LUP
   Output : global P
1  literals ← FLATTEN(S_in)
2  proofbound, pbytes ← COUNTPROOF(literals)
3  P ← ALLOCATE(proofbound)
4  kernel COUNTPROOF(literals):
5      register rBytes ← 0, b ← blockDim
6      for all tid ∈ ⟦ 0, |literals| ⟧ do in parallel
7          l⁺ ← literals[tid]
8          if pbytes[l⁺] = 0 then
9              pbytes[l⁺] ← LUP[30 − CLZ(l⁺)]          // offset (32 − CLZ(l⁺)) by −2
10             rBytes ← rBytes + pbytes[l⁺]
11         else                  // the more threads taking this path, the better
12             rBytes ← rBytes + pbytes[l⁺]
13         end
14     end
15     if t_x < |S_in| then
16         shBytes[t_x] = rBytes
17     else
18         shBytes[t_x] = 0
19     end
20     SYNCTHREADS( )
21     for b : b/2 → 32 do        // b will be blockDim/2, (blockDim/2)/2, …, 32
22         if t_x < b then
23             shBytes[t_x] ← 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
```

Algorithm 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 unsigned literals and stored in the array `literals`. At line 2, the kernel COUNTPROOF is launched (given at lines 4–31) 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. Again, the value b initially holds the current block size (used later as a loop counter). Within the loop at lines 6–14, the counter *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 Fig. 4). Having a non-zero value at `pbytes`$[l^+]$ means the current literal is a duplicate and its variadic size is already computed before. Accordingly, in this case, we rely on data racing and thread divergence, which contradicts the convention of parallel programming. The following example explains this phenomenon.

Example 6 Suppose we have a set of unsigned `literals` $= \{3, 3, 5, 6\}$ and four threads $t_0, \ldots, t_3$ where t_i represents the *tid* of thread i. In Algorithm 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.

6 Variable scheduling

In our GPU-accelerated inprocessing, each simplification method is applied on multiple variables simultaneously. Doing so may lead to data hazards, due to the disjunction between literals in all clauses (i.e. data dependency). Two variables x and y are dependent iff there exists a clause C with $(x \in C \vee \neg x \in C) \wedge (y \in C \vee \neg y \in C)$. If the two dependent variables x and y were to be processed for simplification, two threads might manipulate C at the same time. To guarantee soundness of the parallel simplifications, we apply our *least constrained variable elections* algorithm (LCVE) [44] prior to simplification. It 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.

Moreover, we map no more than 12 frozen variables to local variables named q_1 to q_{12}. Any variable beyond this range is set to 0 (i.e. a unique stamp to identify out-of-range variables). The mapped variables are used later in BVE to build the function tables (denoted by FUNTAB) with size $2^{12} = 4096$ bits.

The authorised, elected and frozen candidates are defined as follows:

Definition 6 (*Authorised candidates*) Given a 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 7 (*Candidate Dependency Relation*) We call a relation D: $\mathcal{A} \times \mathcal{A}$ a *candidate dependency relation* iff $\forall x, y \in \mathcal{A}$, $x \mathsf{D} y$ 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 8 (*Elected candidates*) Given a set of authorised candidates $\mathcal{A}$, we call a set $\Phi \subseteq \mathcal{A}$ a set of *elected candidates* iff $\forall x, y \in \Phi.\ \neg(x \mathsf{D} y)$

Definition 9 (*Frozen variables*) Given the sets $\mathcal{A}$ and Φ, the set of *frozen variables* $\mathcal{F} \subseteq \mathcal{A}$ is defined as $\mathcal{F} = \{x \mid x \in \mathcal{A} \wedge \exists y \in \Phi.\ x \mathsf{D} y\}$

Algorithm 3: Constructing $\mathcal{A}$

```
Input  : global S_d, μ
Output : global A, scores, h
1  h ← HISTOGRAM(S_d)
2  A, scores ← ASSIGNSCORES(h, A, scores)
3  A ← PRUNE(SORT(A, scores), h, μ)
4  kernel ASSIGNSCORES(h, A, scores):
5      for all tid ∈ ⟦ 0, |var(L)| ⟧ do in parallel
6          x ← tid + 1, A[tid] ← x
7          if h[x] = 0 ∨ h[¬x] = 0 then
8              scores[x] ← MAX(h[x], h[¬x])
9          else
10             scores[x] ← h[x] × h[¬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 [18]). 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 3. As input, it requires a SAT formula $\mathcal{S}$ and a cut-off point μ. At line 1, a histogram array h, 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.[2] 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.

LCVE Algorithm Next, Algorithm 4 is executed on the host, 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. Initially, all elements in $\mathcal{F}_{map}$ are set to 0 (line 1). From this point forward, all violet routines or notations suggest a new contribution compared to our previous work in [47, 48]. 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 q in the range $1 \leq q \leq 12$ and store it in $\mathcal{F}_{map}$ (lines 10–12).

[2] https://github.com/NVIDIA/thrust.

Algorithm 4: LCVE with $\mathcal{F}$-mapping functionality

Input : $\mathcal{S}, \mathcal{A}, h, \mathcal{T}$
Output : Φ

```
1  F ← ∅, F_map ← {0}, q ← 1
2  foreach x ∈ A do
3  |  if x ∉ F then
4  |  |  Φ ← Φ ∪ x
5  |  |  foreach C ∈ S[T[x]] ∪ S[T[¬x]] do
6  |  |  |  foreach ℓ ∈ C do
7  |  |  |  |  v ← var(ℓ)
8  |  |  |  |  if v ≠ x then
9  |  |  |  |  |  F ← F ∪ v
10 |  |  |  |  |  if q ≤ 12 then
11 |  |  |  |  |  |  F_map[v] ← q, q ← q + 1
12 |  |  |  |  |  end
13 |  |  |  |  end
14 |  |  |  end
15 |  |  end
16 |  end
17 end
```

7 Main inprocessing procedure

A top-level description of GPU parallel inprocessing is shown in Algorithm 5. 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. A stream is a sequence of instructions that are executed in issue-order on the GPU [39]. The use of concurrent streams allows the running of multiple GPU kernels concurrently, if there are enough resources. The ORDERVARIABLES routine produces an ordered array of authorised candidates $\mathcal{A}$ following Definition 6.

Algorithm 5: Certified Parallel Inprocessing

```
   Input  : S_h, μ, phases
1  S_d ← copyToDevice(S_h)
2  prepareFormula(S_d, stream0)
3  A ← orderVariables(S_d, stream1)
4  P_h, P_d ← proofAllocator(S_d, stream0)
5  for p ← 0 to phases do
6      syncAll( )                                   // Synchronize all streams
7      T ← createOT(S_d)
8      bcp(U_h, S_d, T)
9      Φ ← lcve(S_d, T, A, μ)
10     if p = phases then
11         ere(S_d, T, Φ)
12         break
13     end
14     sortOT(T, Φ, listKey)
15     U_d, P_d ← eliminate(S_d, T, Φ)              // Applies SUB then BVE
16     P_h ← copyToHostAsync(P_d, stream1)
17     U_h ← copyToHostAsync(U_d, stream2)
18     collect(S_d, stream3)
19     sync(stream1), writeProof(P_h)
20     μ ← μ × 2
21 end
22 device function listKey(a, b):
23     C_a ← S_d[a], C_a ← S_d[b]        // C_a = {x_1, x_2, ..., x_k}, C_b = {y_1, y_2, ..., y_k}
24     if |C_a| ≠ |C_b| then return |C_a| < |C_b|
25     if x_1 ≠ y_1 then return x_1 < y_1
26     if x_k ≠ y_k then return x_k < y_k
27     if sig(C_a) ≠ sig(C_b) then return sig(C_a) < sig(C_b)
28     return a < b
29 end
```

At line 4, Algorithm 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 Sect. 3.1) on the host side with the same size as $\mathcal{P}_d$.

The **for** 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 Φ (see Definition 8) as explained earlier in Algorithm 4.

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. These 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 [8]. The SORTOT routine launches a kernel to sort the lists pointed to by the variables in Φ in parallel. Each thread runs an insertion sort to in-place swap clause references using LISTKEY.

The ELIMINATE procedure at line 15 calls SUB 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 Φ in parallel. More information on this is in Sects. 8 and 9. 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. Therefore, 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 10 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. It should be noted that asynchronous data transfers to the host are only permitted if the host memory is page-locked [39]. The COLLECT procedure does the GC as described by Algorithm 1 via *stream3*. At line 19, we synchronise the proof data transfer performed by *stream1* and write the byte stream to the proof output file. Other active streams are synchronised at line 6.

8 Three-phase parallel variable elimination

The BVIPE algorithm in our previous work [44] had a main shortcoming due to the heavy use of atomic operations in adding 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).

Algorithm 6: Certified Parallel BVE with FUNTAB Reasoning

```
Input  : global Φ, T, U_d, P_d, S_d, litstack, varinfo, cindex, rindex, pbytes
Input  : constant CB
varinfo ← VCESCAN(Φ, S_d, T)
cindex ← COMPUTECLAUSEINDICES(varinfo, SIZE(clauses))
rindex ← COMPUTECLAUSEREFINDICES(varinfo, SIZE(references))
VCEAPPLY(Φ, S_d, T, varinfo, cindex, rindex)
kernel VCESCAN(Φ, S_d, T):
    for all tid ∈ ⟦ 0, |Φ| ⟧ do in parallel
        register x ← Φ[tid], T_x = T[x], T_¬x = T[¬x]
        register t ← NONE, rCls ← 0, rLits ← 0
        varinfo[tid] ← 0, cindex[tid] ← 0, rindex[tid] ← 0
        if T_x = ∅ ∨ T_¬x = ∅ then litstack ← TOBLIVION(x, S_d, T_x, T_¬x)
        else
            t, rCls, rLits ← GATEREASONING(x, S_d, T_x, T_¬x)
            if t = NONE then t, rCls, rLits ← FUNREASONING(x, S_d, T_x, T_¬x)
            if t = NONE then t, rCls, rLits ← MAYRESOLVE(x, S_d, T_x, T_¬x)
            varinfo[tid] ← t, rindex[tid] ← rCls, cindex[tid] ← CB × rCls + rLits
        end
    end
end
kernel VCEAPPLY(Φ, S_d, T, varinfo, cindex, rindex):
    for all tid ∈ ⟦ 0, |Φ| ⟧ do in parallel
        register x ← Φ[tid]
        register t ← varinfo[tid], cidx ← cindex[tid], ridx = rindex[tid]
        reqSpace ← cidx + CB × rCls + rLits
        reqRefs ← ridx + rCls
        if t ≠ NONE ∧ reqSpace ≤ CAP(clauses) ∧ reqRefs ≤ CAP(references) then
            if t = RES then
                (S_d, U_d, P_d) ← (S_d, U_d, P_d) ∪ RESOLVE(x, S_d, T, ridx, cidx, pbytes)
            else if t = SUBST then
                (S_d, U_d, P_d) ← (S_d, U_d, P_d) ∪ SUBSTITUTE(x, S_d, T, ridx, cidx, pbytes)
            else if t = CORE then
                (S_d, U_d, P_d) ← (S_d, U_d, P_d) ∪ CORESUBSTITUTE(x, S_d, T, ridx, cidx, pbytes)
            end
            litstack ← TOBLIVION(x, S_d, T[x], T[¬x])
        end
    end
end
device function FUNREASONING(x, S_d, T_x, T_¬x):
    local f_p ← {1}, f_n ← {1}
    withinRange ← BUILDFUNTAB(x, S_d, T_x, f_p) ∧ BUILDFUNTAB(¬x, S_d, T_¬x, f_n)
    if withinRange ∧ AND(f_p, f_n) = {0} then
        G_ℓ ← S_d[T[x]] ∪ S_d[T[¬x]]
        forall C ∈ G_ℓ do
            if FLAG(C) = 0 ∧ ISFALSEFUN(G_ℓ \ {C}) then FLAG(C) ← 1
        end
    end
end
device function BUILDFUNTAB(ℓ, S_d, T_ℓ, f):
    forall C ∈ S_d[T_ℓ] do
        shared f_s ← {0}
        forall ℓ' ∈ C do
            if ℓ' ≠ ℓ then
                q ← F_map[var(ℓ')]
                if q = 0 then return false
                f_s ← OR(f_s, MAGICNUM(q))
            end
        end
        f ← AND(f, f_s)
    end
end
```

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 (Sect. 4). 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 are relatively small compared to the eliminated variables,[3] hence the penalty of using atomic instructions is almost negligible. It would be an overkill to use a segmented scan for adding proof bytes or units.

At line 1 of Algorithm 6, 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 *elimination type* (i.e. `NONE`, `RES`, `SUBST` or `CORE`), 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 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 at lines 37–46 is responsible for finding irregular gate definitions as explained in Sect. 2.1. At line 38, two Boolean function tables f_p, f_n are created in threads' **local** memory.[4] Both are bit-vectors of length 4096 entries initialized to ones. Each table represents a 12-variables Boolean function stored in a vector of 4096 bits. The bit at index i (where i is written as $q_{12}, q_{11}, \ldots, q_1$ in binary) gives the value of the Boolean function for $q_{12}, q_{11}, \ldots, q_1$. Note that the maximum number of variables we support is 12. In real implementation, only enough frozen variables in $\mathcal{F}$ are mapped to values in the domain [1, 12]. At line 39, we encode the clause sets in $\mathcal{S}_d[\mathcal{T}_x]$ and $\mathcal{S}_d[\mathcal{T}_{\neg x}]$ into their truth tables f_p resp. f_n via BUILDFUNTAB. More on how BUILDFUNTAB is implemented is coming later. 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). The clauses set G_ℓ can be reduced by finding a shorter clausal core (not necessarily minimal, though). The loop at lines 42–44 removes a clause at a time from G_ℓ and tests for all-zero bit string via ISFALSEFUN. If $G_\ell \backslash C$ 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 48–58 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 binary magic numbers. A binary magic number is a unique constant in which a sequence of bits is repeating itself multiple times. These numbers can be used to extract and pack integer values into a single bit string (e.g. the Boolean function table). At line 52, the frozen variable $var(\ell')$ is mapped to q. If it has the value 0 (line 53), then we bail out immediately with a **false**. At line 54, MAGICNUM(q) fetches the corresponding magic number from a 64-bit constant array using q's value as an index. With these constants, the truth table of each variable q can be stored in memory. Afterwards, the elementary truth tables are combined using the bitwise operators (OR, AND) to build the truth table of the formula $\mathcal{S}_d$. If the resulting truth table f contains only zeros, the formula is unsatisfiable.

The condition $rCls \leq (|\mathcal{T}_x| + |\mathcal{T}_{\neg x}|)$ is always tested implicitly by all above routines to limit the number of resolvents per x. The `varinfo`, `rindex`, and `cindex` arrays are

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

[4] Actually a global memory with interleaved addressing for fast parallel access.

updated at line 15. The total number of buckets needed to store all added clauses is calculated by the formula ($\texttt{CB} \times \mathit{rCls} + \mathit{rLits}$) and stored in $\texttt{cindex}[\mathit{tid}]$.

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 ($t \neq \texttt{NONE}$) and there is enough memory space, 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 into $\mathcal{P}_d$ as stream bytes using the transformation in WORDS(l^+). Recall that the number of bytes per literal is already stored in `pbytes` and is not required to be computed again.

The sequential running time of Algorithm 6 is $\mathcal{O}(m \cdot |\Phi|)$, where m is the maximum length of a resolved clause in $\mathcal{S}$. In practice, a limit over a resolvent length is set to a small constant value ($\leq$ 100, for instance). Hence, 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 complexity is an amortised constant $\mathcal{O}(1)$.

9 Parallel subsumption elimination

Parallel SUB through Algorithm 7 is executed on elected variables before variable elimination. (Self)-subsumption elimination tends to reduce the number of occurrences of these variables as it usually removes many literals and clauses. The parallelism in Algorithm 7 is achieved on the variable level. In other words, each thread is assigned to a variable x and performs 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.

Algorithm 7: Certified Parallel SUB

```
   Input  : global S_d, Φ, T, U_d, P_d
 1 kernel SUB(Φ, S_d, T, U_d, P_d):
 2   for all tid ∈ [[ 0, |Φ| ]] do in parallel
 3     register x ← Φ[tid], T_x = T[x], T_¬x = T[¬x]
 4     foreach C ∈ S_d[T_x] do
 5       shared C_s ← C
 6       foreach C' ∈ S_d[T_¬x] do
 7         if |C'| ≤ |C| ∧ SIG(C', C) ≠ 0 ∧ SELFSUB(C', C_s) then
 8           STRENGTHEN(C, C_s, x)
 9           if |C_s| = 1 then U_d ← U_d ∪ C_s
10           PROOFADDCLAUSE(P_d, C_s)
11         end
12       end
13       foreach C'' ∈ S_d[T_x] do
14         if |C''| ≤ |C| ∧ SIG(C'', C) ≠ 0 ∧ SUBSUME(C'', C_s) then
15           if STATE(C'') = LEARNT ∧ STATE(C) = ORIGINAL then
16             STATE(C'') ← ORIGINAL
17           end
18           STATE(C) ← DELETED
19           PROOFDELCLAUSE(P_d, C_s)
20         end
21       end
22     end
23   end
24 end
```

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. 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, then C'' must be turned to ORIGINAL (see Sect. 2.2). This time, the subsumed clause is written to the proof as a deleted lemma.

Regarding the complexity of Algorithm 7, 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 6). The same applies for its complement. In a worst case scenario, a variable and its negation both occur μ times. As SUB 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 $\mathcal{O}(\mu^2)$.

10 Eager redundancy elimination

Algorithm 8 describes a *two-dimensional* kernel, in which from each thread ID, an x and y coordinate is derived. This allows us to use two nested grid-stride loops. In the loops, we specify which of the two coordinates should be used to initialise *tid* in the first iteration.

Algorithm 8: Certified Parallel ERE for Inprocessing

```
   Input  : global Φ, S_d, T, P_d
1  kernel ERE(Φ, S_d, T, P_d):
2      for all tid_y ∈ ⟦ 0, |Φ| ⟧^y do in parallel
3          register x ← Φ[tid_y], T_x = T[x], T_¬x = T[¬x]
4          for C ∈ S_d[T_x] do
5              for C' ∈ S_d[T_¬x] do
6                  if (C_m ← RESOLVE(x, C, C')) ≠ ∅ then
7                      if STATE(C) = LEARNT ∨ STATE(C') = LEARNT then
8                          st ← LEARNT
9                      else
10                         st ← ORIGINAL
11                     end
12                     FORWARDEQUALITY(C_m, S_d, T, st)
13                 end
14             end
15         end
16     end
17 end
18 device function FORWARDEQUALITY(C_m, S_d, T, st):
19     T_min ← FINDMINLIST(T, C_m)
20     for all tid_x ∈ ⟦ 0, |T_min| ⟧^x do in parallel
21         C ← S_d[T_min[tid_x]]
22         if C = C_m ∧ (STATE(C) = LEARNT ∨ STATE(C) = st) then
23             STATE(C) ← DELETED
24             PROOFDELCLAUSE(P_d, C)
25         end
26     end
27 end
```

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 minimum 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

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

where $p_{\boldsymbol{y}}$ and $p_{\boldsymbol{x}}$ are the number of launched threads in $\boldsymbol{y}$ resp. $\boldsymbol{x}$ dimensions. Note that $|\mathcal{T}_{min}|$ is usually very small compared to the upper bound μ. Hence, 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 a quadratic function of the upper-bound μ.

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

Example 7 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 $\text{CEIL}(1,000/256) = 4$ 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 9 illustrates a way to automate the tuning of the kernel configuration for maximum occupancy. It takes as input: the data size N, maximum supported threads p, 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 till 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 PARAFROST, we have set the *minBlockDim* and *minOccupancy* to 4 resp. 0.5.

Algorithm 9: Kernel Auto Tuner

```
   Input  : N, p, initBlockDim, minBlockDim, minOccupancy
   Output : blockDim, gridDim
1  blockDim ← initBlockDim
2  gridDim ← CEIL(N / blockDim)
3  maxBlocks ← p / initBlockDim
4  minBlocks ← maxBlocks × minOccupancy
5  while blockDim > minBlockDim ∧ gridDim ≤ minBlocks do
6  |   blockDim ← blockDim / 2
7  |   gridDim ← CEIL(N / blockDim)
8  end
9  gridDim ← MIN(gridDim, maxBlocks)
```

We observed that the number of scheduled variables $|\Phi|$ in the preceding kernels VCESCAN, VCEAPPLY, SUB, and ERE usually go 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 9 on the previous kernels to maximally increase the number of blocks that are scheduled for execution on the available SMs. This led to an overall reduction in the running time of the inprocessing procedure by 2%.

12 Experimental evaluation

12.1 Setup

We implemented the proposed algorithms in our solver ParaFROST[5] with CUDA C++ version 11.0 [39]. 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 CaDiCaL and Kissat [8] solvers developed by the third author.

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 [40].

We evaluated all GPU experiments on the compute nodes of the Lisa GPU cluster.[6] Each problem was analysed in isolation on a separate computing node, with a time-out of 3600 s. 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, and runs on Debian Linux operating system. This GPU has 72 SMs (64 cores each), 24 GB global memory and 48 KB shared memory. It operates at a base clock of 1.3 GHz.

The sequential solvers are executed on the compute nodes of a different cluster called DAS-5 [3] to dedicate our computing hours on Lisa cluster only to the GPU experiments. Each node of DAS-5 had an Intel Xeon E5-2630 CPU (2.4 GHz) with 64 GB of memory. The proofs generated by the GPU solver are verified separately by the drat-trim tool [63] on DAS-5, with a time-out of 20,000 s. It is worth mentioning that the CPU time of a single-threaded task is around 10% faster on DAS-5 compared to Lisa.

With this information, we adhere to four out of five principles laid out in the SAT manifesto (version 1) [10]:

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).

12.2 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 [44, 45] or sequential counterparts in ParaFROST (noGPU). For these

[5] Available from: https://github.com/muhos/ParaFROST.

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

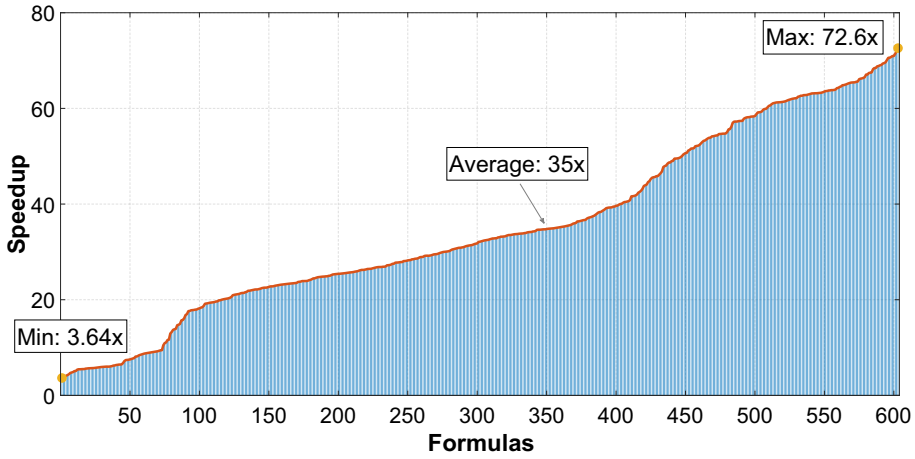


Fig. 5 Parallel GC versus sequential speedup

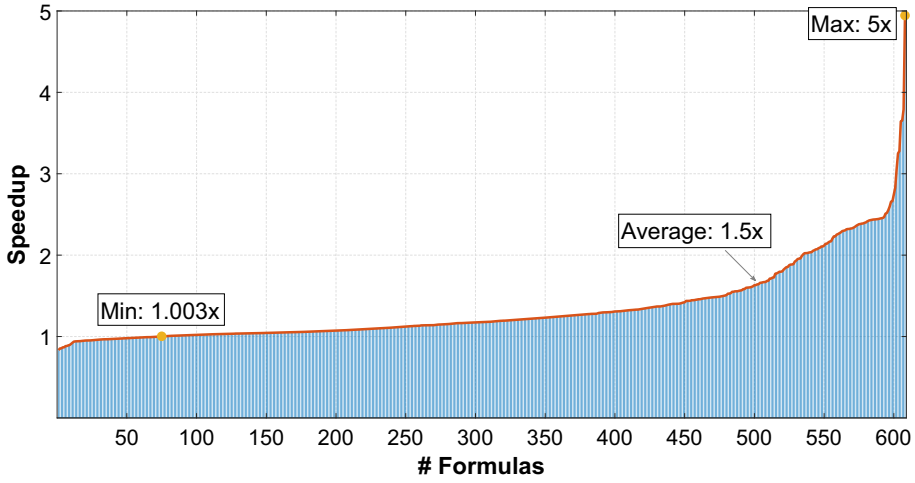


Fig. 6 Three-phase BVE versus atomic version speedup

experiments, we set μ and *phases* initially to 32 resp. 5. Preprocessing is only enabled to measure the speedup.

Figure 5 gives the speedup of running parallel GC against a sequential version on the host. For almost all cases, Algorithm 1 achieved a high acceleration when executed on the device with a maximum speed up of 72.6× and an average of 35×. Figure 6 reveals how fast the 3-phase parallel BVE is compared to a version using more atomic instructions. On average, the new algorithm is 1.5× faster than the old BVIPE algorithm [44]. In addition, we get reproducible results.

Figure 7 evaluates the FUNTAB method in Algorithm 6 against the sequential counterpart. Note the logarithmic scale on the y-axis. 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, as shown by Fig. 8,

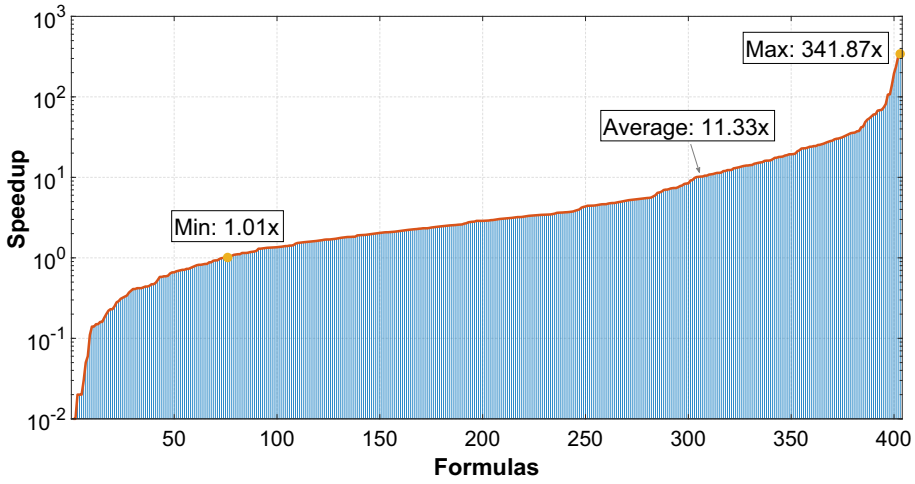


Fig. 7 Parallel FUNTAB versus sequential speedup

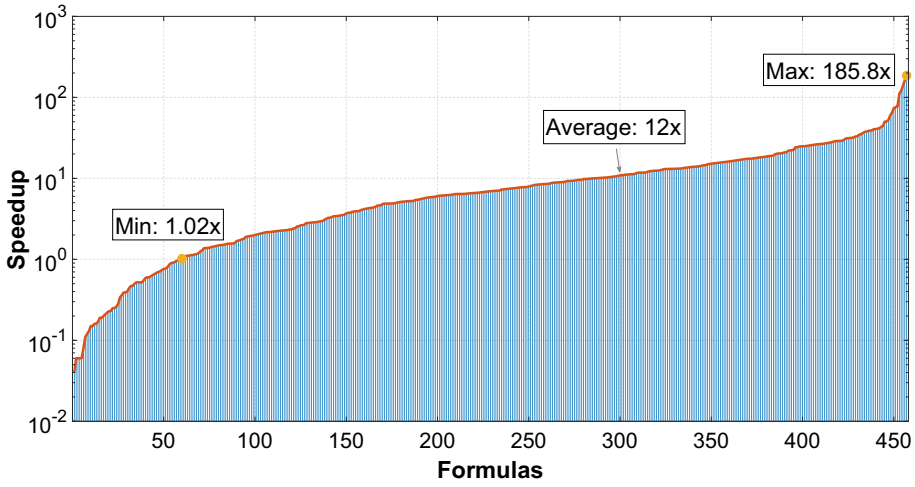


Fig. 8 Parallel proof generation versus sequential speedup

the acceleration of proof generation on the GPU is significant, with a maximum speedup of 186× and 12× on average.

12.3 SAT solving performance

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. To gain advantage of the GPU resources, preprocessing in PARAFROST is enabled by default while disabled in CADICAL and KISSAT solvers. We could enable preprocessing in other solvers but we preferred to leave their default options untouched. The timeout is set to 3,600 s for all experiments.

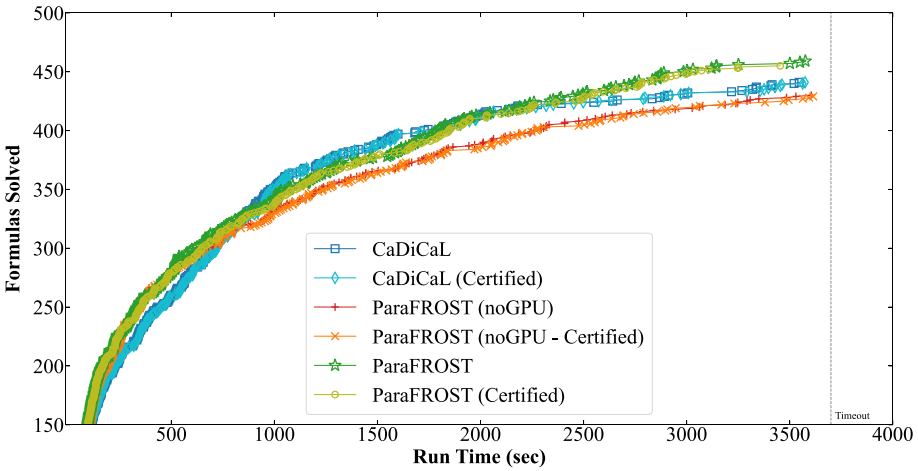


Fig. 9 PARAFROST versus CADICAL with(out) proof emission

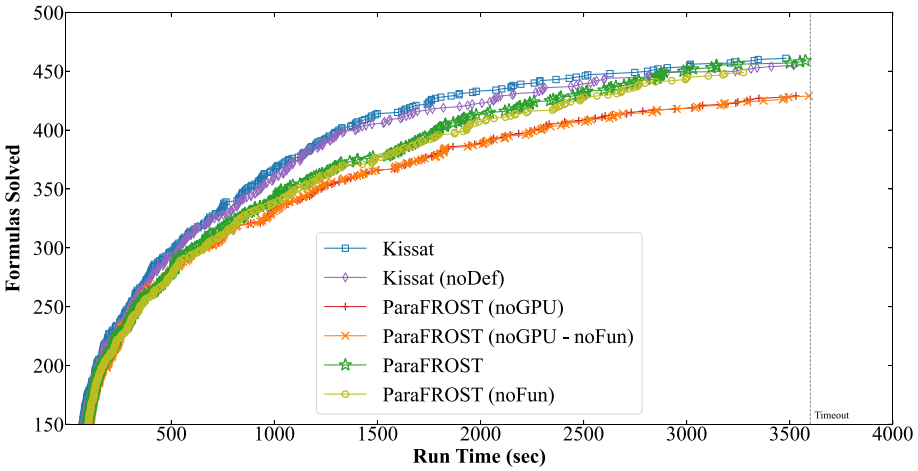


Fig. 10 PARAFROST versus KISSAT with(out) irregular-gate reasoning

Figure 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 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 10 compares PARAFROST to KISSAT with(out) irregular gate reasoning. 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 compared to its sequential counterpart (see Fig. 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 built-in solver called KITTEN which is responsible for solving and extracting

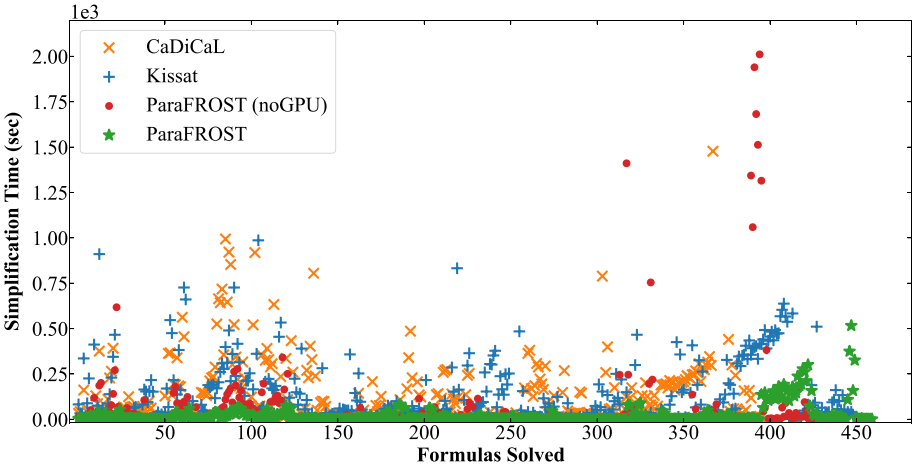


Fig. 11 Time spent on simplifications

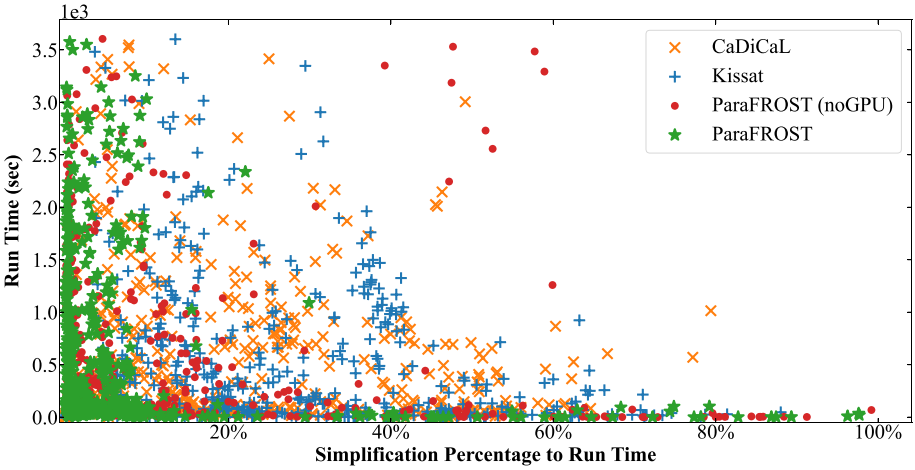


Fig. 12 Percentage of simplification time to runtime

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 PARAFROST CDCL heuristics (run on the host) is based on CADICAL which is not up-to-date as KISSAT. We expect PARAFROST to compete with KISSAT if the same heuristics implemented in the latter is used.

Figures 11 and 12 show simplification time and its percentage of the total run time, respectively. Clearly, the PARAFROST solver outperforms the sequential solvers due to the parallel acceleration. Figure 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 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

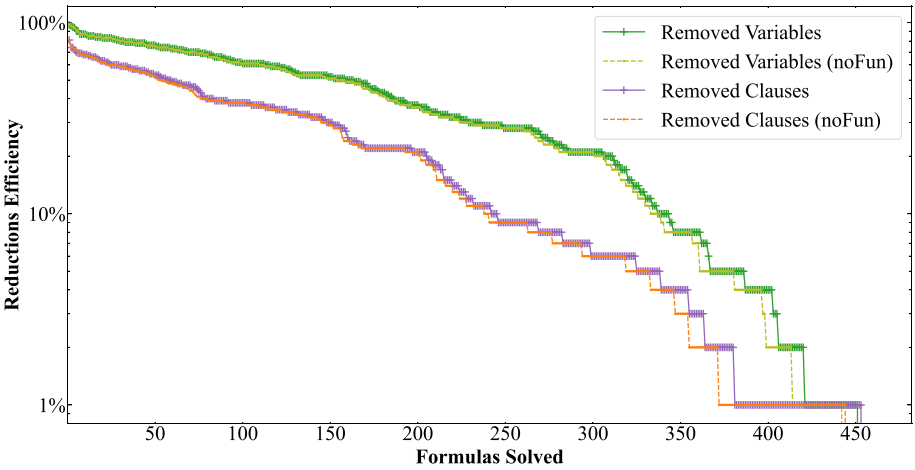


Fig. 13 Reduction efficiency with(out) irregular-gate reasoning

descending order. Reductions can remove up to 97% and 80% of the variables and clauses, respectively.

Figure 14 gives the heat-map distribution of the time spent on verification and the memory consumed by the proofs generated by ParaFROST [41] and CaDiCaL. All proofs are successfully verified via DRAT-TRIM tool. The verification times are reperesented by the *colormap* and are sorted in descending order w.r.t. the *unsatisfiable* instances solved. Figure 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). That is foreseen as the deleted lemmas in Algorithm 6 are not saved to the proof in order to avoid GPU memory exhaustion. On the other hand, CaDiCaL proofs as demonstrated by Plot 14b take less time to verify (e.g. 0.75×10^4 to 1×10^4) 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.

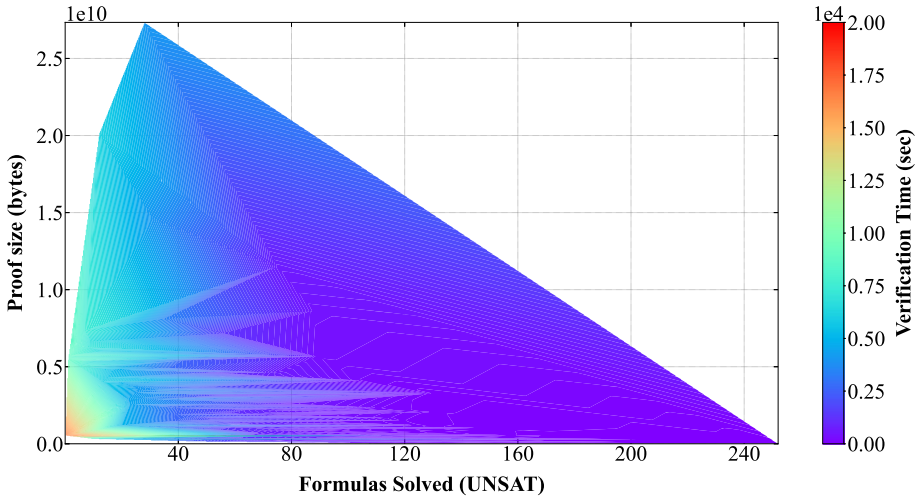
Tables 1 zooms into the impact of applying the funTab method in BVE on solving a sample of 30 formulas using ParaFROST and ParaFROST (noFun), respectively. The letters *V* and *C* refer to *Variables* and *Clauses*, respectively. The keywords *org* and *rem* denote *original* and *removed*, respectively. Removed clauses include both `ORIGINAL` and `LEARNT` types. 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 a side effect of funTab. Additionally, ParaFROST solved many cases faster than p (noFun) within the time limit (3,600 s). For instance, the formula *HCP-446-105* was solved by ParaFROST with funTab in just 907.21 s, while it took 1,020.07 s to solve for ParaFROST without funTab.

Table 1 Zooming into 30 formulas solved by ParaFROST with(out) funTab

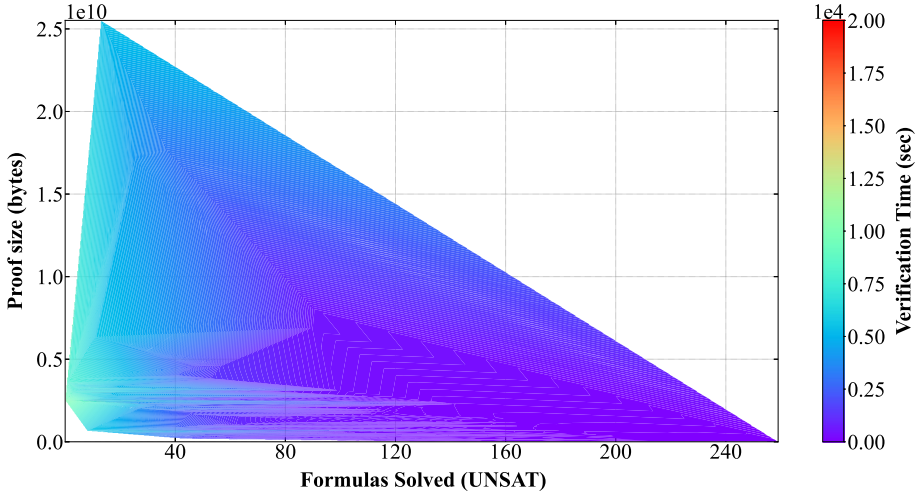
CNF	ParaFROST					ParaFROST (noFun)		
Formula	V (org)	C (org)	V (rem)	C (rem)	t (s)	V (rem)	C (rem)	t (s)
01-integer-programming-20-30-40	3,518,573	891,394	**648,093**	**1,776,849**	1948.05	**Time out** (> 3600)		
hwmcc15deep-beemhanoi4b1-k37	1,070,494	455,572	**356,713**	**734,111**	3576.13	**Time out** (> 3600)		
sokoban-p20.sas.cr.27	852,527	101,334	**50,858**	**294,398**	3500.59	**Time out** (> 3600)		
T96.1.1	32,322,587	8,905,808	**4,682,382**	**10,522,621**	1314.91	4,672,738	10,486,383	982.60
T97.2.1	14,110,352	4,038,010	**2,461,187**	**5,412,098**	2190.44	2,453,334	5,354,951	1948.94
newton.2.2.i.smt2-cvc4	843,395	196,151	**100,600**	**446,511**	2174.61	95,467	382,463	2028.31
vlsat2_30744_3925645.dimacs	3,887,186	30,744	**1926**	**343,699**	320.68	**Time out** (> 3600)		
snw_16_9_Encpre	1,641,152	72,327	**1763**	**28,757**	3120.85	**Time out** (> 3600)		
T99.2.1	21,859,028	6,269,521	**3,790,028**	7,918,822	**1920.36**	3,788,312	7,926,209	2025.92
string_compare_safety_cbmc_940	17,856,438	3,416,996	**955,052**	**3,907,091**	3251.30	953,420	3,857,959	2849.89
SAT_dat.k80-24_1_rule_1	4,248,961	1,084,904	**752,757**	**2,019,402**	**1256.88**	751,544	2,008,237	1292.18
string_compare_safety_cbmc_840	15,657,846	3,062,432	**862,597**	**3,319,353**	**1803.59**	861,602	3,313,347	1931.94
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**	1826.12	888,134	1,828,131	1752.91
HCP-446-105	247,619	29,934	**7285**	**111,534**	**907.21**	6655	103,789	1020.07

Table 1 continued

CNF			ParaFROST			ParaFROST (noFun)		
Formula	V (org)	C (org)	V (rem)	C (rem)	t (s)	V (rem)	C (rem)	t (s)
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**	1615.06	729,161	2,738,158	1472.11
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	1760.92	768,846	2,946,636	1556.80
hwmcc15deep-6s188-k44	596,873	202,373	**140,408**	**372,555**	1891.48	140,204	369,277	1875.26
HCP-470-105	254,981	31,580	**3773**	**72,682**	536.78	3573	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**	2720.94	922,286	3,641,942	2684.22
Mickey_out250_known_last147_0	518,695	72,078	**2442**	**21,009**	**124.08**	2350	16,664	128.14
sv-comp19_prop-reachsafety.sigma	5,053,926	1,094,922	**132,589**	**497,325**	**1031.37**	132,506	430,720	1144.78
HCP-446-60	227,594	27,377	**2981**	46,528	**532.47**	2951	50,300	627.87
sted1_0x0-637	336,166	13,431	**28**	**10,259**	**954.43**	**Time out** (> 3600)		
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	**2352.81**	204,236	571,675	2391.08



(a) Verification of ParaFROST proofs



(b) Verification of CaDiCaL proofs

Fig. 14 Heatmap showing the time-memory distribution of DRAT proof

13 Related work

A simple GC monitor for GPU term rewriting has been proposed by [62]. The monitor tracks deleted terms and stores their indices in a list. New terms can be added at those indices. The first author extended the former work by a stream GC similar to the one described in this article but much more simpler [61]. The authors in [1, 36] investigated the challenges for offloading garbage collectors to an Accelerated Processing Unit (APU) [56] introduced a

promising alternative for stream compaction [11] 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, [27] introduced certain rules to determine how and when inprocessing techniques can be applied. [5] presented LINGELING, the first solver with the ability of finding general gate definitions using a BDD-based approach. The works in [7, 8, 20] introduced a SAT-based variant of mining gate definitions by calling an embedded SAT solver named KITTEN as independent component of KISSAT. The GPU-based variant presented in this article was inspired by [5, 7].

Acceleration of the DPLL SAT solving algorithm on a GPU has been done in [50], 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 [68]. Recently, [52] used the GPUs to determine the usefulness of a learnt clause for parallel Portfolio-based solvers. Nonetheless, we are the first to work on certified CDCL solvers with GPU accelerated inprocessing.

14 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, the usage of atomic operations has been considerably reduced which lead to an average speedup of 1.5× 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 high credibility to our solver and its applicability in critical tools 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 robust load balancing. Another direction is to use PARAFROST in a Portfolio-based parallel SAT solving and exploit the GPU capabilities in managing the shared clause database as recently introduced by [52].

Acknowledgements This work is part of the GEARS project with project number TOP2.16.044, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO). In addition, this work made use of the Dutch national e-infrastructure with the support of the SURF Cooperative using Grant No. EINF-1688. The third author was supported by the LIT AI Lab funded by the Upper State of Austria.We would like to thank the anonymous reviewers for their valuable feedback to improve an earlier version of this manuscript.

Data availability The datasets generated during and/or analysed during the current study are available in the Zenodo repository [40, 41].

References

1. Abhinav, Nasre R (2016) FastCollect: offloading generational garbage collection to integrated GPUs. In: CASES, Pittsburgh, Pennsylvania, USA, October 1–7, 2016. Proceedings. ACM, pp 21:1–21:10. https://doi.org/10.1145/2968455.2968520
2. Audemard G, Simon L (2009) Predicting learnt clauses quality in modern SAT solvers. In: IJCAI, Pasadena, California, USA, July 11–17, 2009, pp 399–404. https://doi.org/10.5555/1661445.1661509
3. Bal HE, Epema DHJ, de Laat C et al (2016) A medium-scale distributed system for computer science research: infrastructure for the long term. Computer 49(5):54–63. https://doi.org/10.1109/MC.2016.127
4. Bao FS, Gutierrez CE, Jn-Charles J et al (2018) Accelerating Boolean satisfiability (SAT) solving by common subclause elimination. Artif Intell Rev 49(3):439–453. https://doi.org/10.1007/s10462-016-9530-6
5. Biere A (2013) Lingeling, plingeling and treengeling entering the SAT competition 2013. In: Proceedings of SAT competition 2013—solver and benchmark descriptions, Department of Computer Science report series B, vol B-2013-1. University of Helsinki, p 1
6. Biere A, Cimatti A, Clarke EM et al (1999) Symbolic model checking without BDDs. In: TACAS, Amsterdam, The Netherlands, March 22–28, 1999. Proceedings, LNCS, vol 1579. Springer, pp 193–207. https://doi.org/10.1007/3-540-49059-0_14
7. Biere A, Fleury M (2021) Mining definitions in Kissat with kittens. In: Workshop on the pragmatics of SAT 2021. http://www.pragmaticsofsat.org/2021/
8. Biere A, Fleury M, Heisinger M (2021) CADICAL, KISSAT, PARACOST. In: Proceedings of SAT competition 2021—solver and benchmark descriptions, Department of Computer Science report series B, vol B-2021-1. University of Helsinki, pp 10–13
9. Biere A, Järvisalo M, Kiesl B (2020) Preprocessing in SAT solving. In: Handbook of satisfiability, 2nd edn. Frontiers in Artificial Intelligence and Applications, IOS Press
10. Biere A, Järvisalo M, Le Berre D et al (2020). The SAT practitioner's manifesto. https://doi.org/10.5281/zenodo.4500928
11. Billeter M, Olsson O, Assarsson U (2009) Efficient stream compaction on wide SIMD many-core architectures. In: HPG, New Orleans, Louisiana, USA, August 1–3, 2009. Proceedings. Eurographics Association, pp 159–166. https://doi.org/10.2312/EGGH/HPG09/159-166
12. Bošnački D, Edelkamp S, Sulewski D et al (2010) GPU-PRISM: an extension of PRISM for general purpose graphics processing units. In: PDMC-HiBi, pp 17–19. https://doi.org/10.1109/PDMC-HiBi.2010.11
13. Bradley AR (2011) SAT-based model checking without unrolling. In: VMCAI, Austin, TX, USA, January 23–25, 2011. Proceedings, LNCS, vol 6538. Springer, pp 70–87. https://doi.org/10.1007/978-3-642-18275-4_7
14. Brown CE (2013) Reducing higher-order theorem proving to a sequence of SAT problems. J Autom Reason 51(1):57–77. https://doi.org/10.1007/s10817-013-9283-8
15. Cabodi G, Camurati P, Mishchenko A et al (2017) SAT solver management strategies in IC3: an experimental approach. Formal Methods Syst Des 50(1):39–74. https://doi.org/10.1007/s10703-017-0272-0
16. Clarke EM, Biere A, Raimi R et al (2001) Bounded model checking using satisfiability solving. Formal Methods Syst Des 19(1):7–34. https://doi.org/10.1023/A:1011276507260
17. Davis M, Logemann G, Loveland DW (1962) A machine program for theorem-proving. Commun ACM 5(7):394–397. https://doi.org/10.1145/368273.368557
18. Eén N, Biere A (2005) Effective preprocessing in SAT through variable and clause elimination. In: SAT, St. Andrews, UK, June 19–23, 2005. Proceedings, LNCS, vol 3569. Springer, pp 61–75. https://doi.org/10.1007/11499107_5
19. Eén N, Sörensson N (2003) An extensible SAT-solver. In: SAT, Italy, May 5–8, 2003, selected revised papers, LNCS, vol 2919. Springer, pp 502–518. https://doi.org/10.1007/978-3-540-24605-3_37
20. Fleury M, Biere A (2022) Mining definitions in Kissat with kittens. Extended version of our unpublished POS'21 paper [6], pp 1–22. Submitted to a special issue of a journal
21. Gebhardt K, Manthey N (2013) Parallel variable elimination on CNF formulas. In: KI, Koblenz, Germany, September 16–20, 2013. Proceedings, LNCS, vol 8077. Springer, pp 61–73. https://doi.org/10.1007/978-3-642-40942-4_6

22. Han H, Somenzi F (2007) Alembic: an efficient algorithm for CNF preprocessing. In: DAC, San Diego, CA, USA, June 4–8, 2007. IEEE, pp 582–587. https://doi.org/10.1145/1278480.1278628
23. Heule Jr MWAH, Wetzler N (2013) Trimming while checking clausal proofs. In: FMCAD, Portland, OR, USA, October 20–23, 2013. IEEE, pp 181–188. https://doi.org/10.1109/FMCAD.2013.6679408
24. Heule Jr MWAH, Wetzler N (2013) Verifying refutations with extended resolution. In: CADE-24, Lake Placid, NY, USA, June 9–14, 2013. Proceedings, LNCS, vol 7898. Springer, pp 345–359. https://doi.org/10.1007/978-3-642-38574-2_24
25. Heule M, Järvisalo M, Biere A (2010) Clause elimination procedures for CNF formulas. In: LPAR, Yogyakarta, Indonesia, October 10–15, 2010. Proceedings, LNCS, vol 6397. Springer, pp 357–371. https://doi.org/10.1007/978-3-642-16242-8_26
26. Järvisalo M, Biere A, Heule M (2012) Simulating circuit-level simplifications on CNF. J Autom Reason 49(4):583–619. https://doi.org/10.1007/s10817-011-9239-9
27. Järvisalo M, Heule M, Biere A (2012b) Inprocessing rules. In: IJCAR 2012, Manchester, UK, June 26–29, 2012. Proceedings, LNCS, vol 7364. Springer, pp 355–370. https://doi.org/10.1007/978-3-642-31365-3_28
28. Johnson DS, Trick MA (1996) Cliques, coloring, and satisfiability. In: DIMACS workshop, New Brunswick, New Jersey, USA, October 11–13, 1993. Proceedings, DIMACS series in discrete mathematics and theoretical computer science, vol 26. DIMACS/AMS. https://doi.org/10.1090/dimacs/026
29. Kovásznai G, Fröhlich A, Biere A (2016) Complexity of fixed-size bit-vector logics. Theory Comput Syst 59(2):323–376. https://doi.org/10.1007/s00224-015-9653-1
30. Kullmann O (1999) On a generalization of extended resolution. Discrete Appl Math 96–97:149–176. https://doi.org/10.1016/S0166-218X(99)00037-2
31. Kwiatkowska M, Norman G, Parker D et al (2021) Automatic verification of concurrent stochastic systems. Formal Methods Syst Des 58(1):188–250. https://doi.org/10.1007/s10703-020-00356-y
32. Li CM (2000) Integrating equivalency reasoning into Davis–Putnam procedure. AAAI, IAAI, July 30–August 3, 2000. USA. AAAI Press/The MIT Press, Austin, pp 291–296
33. Liang T, Reynolds A, Tsiskaridze N et al (2016) An efficient SMT solver for string constraints. Formal Methods Syst Des 48(3):206–234. https://doi.org/10.1007/s10703-016-0247-6
34. Liffiton MH, Sakallah KA (2008) Algorithms for computing minimal unsatisfiable subsets of constraints. J Autom Reason 40(1):1–33. https://doi.org/10.1007/s10817-007-9084-z
35. Lynce I, Silva JPM (2003) Probing-based preprocessing techniques for propositional satisfiability. In: ICTAI, 3–5 November 2003, Sacramento, California, USA. IEEE Computer Society, p 105. https://doi.org/10.1109/TAI.2003.1250177
36. Maas M, Reames P, Morlan J et al (2012) GPUs as an opportunity for offloading garbage collection. In: ISMM, Beijing, China, June 15–16, 2012. Proceedings. ACM, pp 25–36. https://doi.org/10.1145/2258996.2259002
37. Moness M, Mahmoud MO, Moustafa AM (2018) A real-time heterogeneous emulator of a high-fidelity utility-scale variable-speed variable-pitch wind turbine. IEEE Trans Ind Inform 14(2):437–447. https://doi.org/10.1109/TII.2017.2723960
38. Munshi A (2009) The OpenCL specification. In: 2009 IEEE hot chips 21 symposium (HCS), Stanford, CA, USA, August 23–25, 2009. IEEE, pp 1–314. https://doi.org/10.1109/HOTCHIPS.2009.7478342
39. NVIDIA (2022) CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide
40. Osama M (2021) Large SAT benchmark suite for certified SAT solving with GPU accelerated inprocessing. https://doi.org/10.5281/zenodo.5138008
41. Osama M (2021) PARAFROST proofs for certified SAT solving with GPU accelerated inprocessing. https://doi.org/10.5281/zenodo.5137887
42. Osama M (2022) GPU enabled automated reasoning. Ph.D. thesis, Eindhoven University of Technology. ISBN: 978-90-386-5445-4
43. Osama M, Gaber L, Hussein AI et al (2018) An efficient SAT-based test generation algorithm with GPU accelerator. J Electron Test 34(5):511–527. https://doi.org/10.1007/s10836-018-5747-4
44. Osama M, Wijs A (2019) Parallel SAT simplification on GPU architectures. In: TACAS, Prague, Czech Republic, April 6–11, 2019. Proceedings, Part I, LNCS, vol 11427. Springer, pp 21–40. https://doi.org/10.1007/978-3-030-17462-0_2
45. Osama M, Wijs A (2019) SIGMA: GPU accelerated simplification of SAT formulas. In: IFM, Bergen, Norway, December 2–6, 2019. Proceedings, LNCS, vol 11918. Springer, pp 514–522. https://doi.org/10.1007/978-3-030-34968-4_29
46. Osama M, Wijs A (2020) Multiple decision making in conflict-driven clause learning. In: ICTAI, Baltimore, MD, USA, November 9–11, 2020. IEEE, pp 161–169. https://doi.org/10.1109/ICTAI50040.2020.00035

47. Osama M, Wijs A (2021) GPU acceleration of bounded model checking with ParaFROST. In: CAV, Los Angeles, July 18–24, 2021. Proceedings, Part II. Springer, LNCS, pp 447–460. https://doi.org/10.1007/978-3-030-81688-9_21
48. Osama M, Wijs A, Biere A (2021) SAT solving with GPU accelerated inprocessing. In: TACAS, Luxembourg, March 27–April 1, 2021. Proceedings, Part I, LNCS, vol 12651. Springer, pp 133–151. https://doi.org/10.1007/978-3-030-72016-2_8
49. Ostrowski R, Grégoire É, Mazure B et al (2002) Recovering and exploiting structural knowledge from CNF formulas. In: CP, Ithaca, NY, USA, September 9–13, 2002. Proceedings, LNCS, vol 2470. Springer, pp 185–199. https://doi.org/10.1007/3-540-46135-3_13
50. Palù AD, Dovier A, Formisano A et al (2015) CUD@SAT: SAT solving on GPUs. J Exp Theor Artif Intell 27(3):293–316. https://doi.org/10.1080/0952813X.2014.954274
51. Piette C, Hamadi Y, Sais L (2008) Vivifying propositional clausal formulae. In: ECAI, Patras, Greece, July 21–25, 2008. Proceedings, Frontiers in Artificial Intelligence and Applications, vol 178. IOS Press, pp 525–529. https://doi.org/10.3233/978-1-58603-891-5-525
52. Prevot N, Soos M, Meel KS (2021) Leveraging GPUs for effective clause sharing in parallel SAT solving. In: SAT, Barcelona, Spain, July 5–9, 2021. Proceedings, LNCS, vol 12831. Springer, pp 471–487. https://doi.org/10.1007/978-3-030-80223-3_32
53. Silva JPM, Glass T (1999) Combinational equivalence checking using satisfiability and recursive learning. In: DATE, 9–12 March 1999, Munich, Germany. IEEE Computer Society/ACM, pp 145–149. https://doi.org/10.1109/DATE.1999.761110
54. Silva JPM, Sakallah KA (1999) GRASP: a search algorithm for propositional satisfiability. IEEE Trans Comput 48(5):506–521. https://doi.org/10.1109/12.769433
55. Soos M, Kulkarni R, Meel KS (2019) CrystalBall: gazing in the black box of SAT solving. In: SAT, Lisbon, Portugal, July 9–12, 2019. Proceedings, LNCS, vol 11628. Springer, pp 371–387. https://doi.org/10.1007/978-3-030-24258-9_26
56. Springer M, Masuhara H (2019) Massively parallel GPU memory compaction. In: ISMM, Phoenix, AZ, USA, June 23, 2019. Proceedings. ACM, pp 14–26. https://doi.org/10.1145/3315573.3329979
57. Stephan PR, Brayton RK, Sangiovanni-Vincentelli AL (1996) Combinational test generation using satisfiability. IEEE Trans Comput Aided Des Integr Circuits Syst 15(9):1167–1176. https://doi.org/10.1109/43.536723
58. Subbarayan S, Pradhan DK (2004) NiVER: non increasing variable elimination resolution for preprocessing SAT instances. In: SAT, 10–13 May 2004, Vancouver, BC, Canada, Online Proceedings. https://doi.org/10.1007/11527695_22
59. Tseitin GS (1983) On the complexity of derivation in propositional calculus. Springer, Berlin, pp 466–483. https://doi.org/10.1007/978-3-642-81955-1_28
60. Tung VX, Khanh TV, Ogawa M (2017) raSAT: an SMT solver for polynomial constraints. Formal Methods Syst Des 51(3):462–499. https://doi.org/10.1007/s10703-017-0284-9
61. van Eerd J, Groote JF, Hijma P et al (2023) Innermost many-sorted term rewriting on GPUs. Sci Comput Program 225:102910. https://doi.org/10.1016/j.scico.2022.102910
62. van Eerd J, Groote JF, Hijma P et al (2020) Term rewriting on GPUs. CoRR arXiv:2009.07174
63. Wetzler N, Heule Jr MWAH (2014) DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: SAT, Vienna, Austria, July 14–17, 2014. Proceedings, LNCS, vol 8561. Springer, pp 422–429. https://doi.org/10.1007/978-3-319-09284-3_31
64. Wijs A (2015) GPU accelerated strong and branching bisimilarity checking. In: TACAS, London, UK, April 11–18, 2015. Proceedings, LNCS, vol 9035. Springer, pp 368–383. https://doi.org/10.1007/978-3-662-46681-0_29
65. Wijs A, Bosnacki D (2014) GPUexplore: many-core on-the-fly state space exploration using GPUs. In: TACAS, Grenoble, France, April 5–13, 2014. Proceedings, LNCS, vol 8413. Springer, pp 233–247. https://doi.org/10.1007/978-3-642-54862-8_16
66. Wijs A, Osama M (2023) A GPU tree database for many-core explicit state space exploration. In: TACAS. Springer, to appear, LNCS
67. Wijs A, Osama M (2023b) GPUexplore 3.0: GPU accelerated state space exploration for concurrent systems with data. In: SPIN. Springer, to appear, LNCS
68. Youness HA, Ibraheim A, Moness M et al (2015) An efficient implementation of ant colony optimization on GPU for the satisfiability problem. In: PDP, Turku, Finland, March 4–6, 2015. IEEE Computer Society, pp 230–235. https://doi.org/10.1109/PDP.2015.59
69. Youness H, Osama M, Hussein A et al (2020) An effective SAT solver utilizing ACO based on heterogenous systems. IEEE Access 8:102920–102934. https://doi.org/10.1109/ACCESS.2020.2999382