GPU Acceleration of Bounded Model Checking with ParaFROST

Muhammad Osama (✉) * and Anton Wijs

Eindhoven University of Technology, Eindhoven, The Netherlands
{o.m.m.muhammad, a.j.wijs}@tue.nl

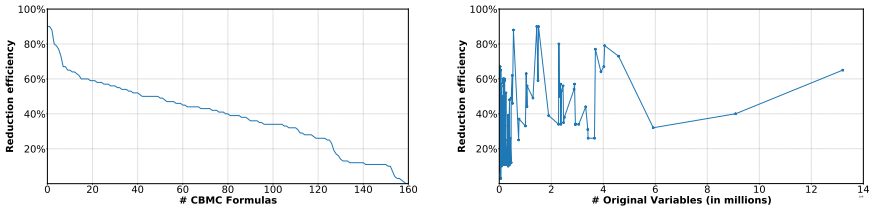
Abstract. The effective parallelisation of Bounded Model Checking is challenging, due to SAT and SMT solving being hard to parallelise. We present ParaFROST, which is the first tool to employ a graphics processor to accelerate BMC, in particular the simplification of SAT formulas before and repeatedly during the solving, known as pre- and inprocessing. The solving itself is performed by a single CPU thread. We explain the design of the tool, the data structures, and the memory management, the latter having been particularly designed to handle SAT formulas typically generated for BMC, i.e., that are large, with many redundant variables. Furthermore, the solver can make multiple decisions simultaneously. We discuss experimental results, having applied ParaFROST on programs from the Core C99 package of Amazon Web Services.

Keywords: Bounded model checking, SAT solving, GPU computing.

1 Introduction

Bounded Model Checking (BMC) [5] determines whether a model M satisfies a certain property φ expressed in temporal logic, by translating the model checking problem to a propositional satisfiability (SAT) problem or a Satisfiability Modulo Theories (SMT) problem. The term *bounded* refers to the fact that the BMC procedure searches for a counterexample to the property, i.e., an execution trace, which is bounded in length by an integer k. If no counterexample up to this length exists, k can be increased and BMC can be applied again. This process can continue until a counterexample has been found, a user-defined threshold has been reached, or it can be concluded (via k-induction [38]) that increasing k further will not result in finding a counterexample. CBMC [14] is an example of a successful BMC model checker that uses SAT solving. CBMC can check ANSI-C programs. The verification is performed by *unwinding* the loops in the program under verification a finite number of times, and checking whether the bounded executions of the program satisfy a particular safety property [22]. These properties may address common program errors, such as null-pointer exceptions and array out-of-bound accesses, and user-provided assertions.

* 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).



(a) The amount of variable redundancy in CBMC formulas

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

Fig. 1: Variable redundancy in CBMC SAT formulas

The performance of BMC heavily relies on the performance of the solver. Over the last decade, efficient SAT solvers [3, 6, 17, 26] have been developed and applied for BMC [5, 10–12, 25]. Effectively *parallelising* BMC is hard. Parallel SAT solving often involves running several solvers, each solving the problem in its own way [18]. For BMC, multiple solvers can be used to solve the problem for different values of the bound k in parallel [1, 21]. However, in these approaches, the individual solvers are still single-threaded.

Recently, Leiserson *et al.* [23] concluded that in the future, advances in computational performance will come from *many-threaded* algorithms that can employ hardware with a massive number of processors. Graphics processors (GPUs) are an example of such hardware. Multi-threaded BMC model checkers have been proposed, such as in [13, 19, 35], but these address tens of threads, not thousands.

In this paper, we propose the application of GPUs to accelerate SAT-based BMC. To the best of our knowledge, this is the first time this is being addressed. Recently, GPUs have been applied for explicit-state model checking and graph analysis [8, 9, 40, 41]. In SAT solving, we used GPUs to accelerate test pattern generation [31], metaheuristic search [42], *preprocessing* [32, 33] and *inprocessing* [34]. In these operations, a given SAT formula is simplified, i.e., it is rewritten to a formula with fewer variables and/or clauses, while preserving satisfiability, using various simplification rules. In preprocessing, this is only done once before the solving starts, while in inprocessing, this is done periodically during the solving. While the impact of accelerating these procedures has been demonstrated [34], its impact on BMC has not yet been addressed.

The structure of typical BMC SAT formulas suggests that GPU pre- and inprocessing will be effective. Fig. 1a shows for a BMC benchmark set taken from the Core C99 package of Amazon Web Services (AWS) [1] [2], consisting of 168 problems of various data structures, that propositional formulas produced by CBMC tend to have a substantial amount of redundant variables that can be removed using simplification procedures. For approximately 50% of the cases, 40% of the variables can be removed. Furthermore, Fig. 1b presents the amount of redundancy in relation to the total number of variables in the formula. It

[1] We thank Daniel Kroening and Natasha Jebbo for pointing us to this package.

indicates that when a formula contains one million variables or more, at least 25% of those are redundant, and often many more. In the benchmark set, the maximum number of variables in one formula is 13 million (encoding the verification of the `priority-queue shift-down` routine), of which 65% is redundant. In contrast, the largest formula we encountered in the application track of the 2013-2020 SAT competitions that is not encoding a verification problem only has 0.2 million variables (it encodes a graph coloring problem [29]).

Contributions. We present the SAT solver ParaFROST that applies Conflict Driven Clause Learning (CDCL) [26] with GPU acceleration of pre- and inprocessing [32–34], tuned for BMC. It has been implemented in CUDA C++ v11 [28], is based on CaDiCaL [6], and interfaces with CBMC.

Having to deal on a GPU with large formulas with a lot of redundancy offers particular challenges. The elimination of variables typically leads to actually adding new clauses, and since the amount of memory on a GPU is limited, this cannot be done carelessly. Therefore, first of all, we have worked on compacting the data structure used to store formula clauses in ParaFROST as much as possible, while still allowing for the application of effective solving optimisations. Second of all, we introduce *memory-aware* variable elimination, to avoid running out of memory due to adding too many new clauses. In practice, we experienced this problem when applying the original procedure of [34] for BMC.

Additionally, to support BMC, ParaFROST must be an *incremental* solver, i.e., it must exploit that a number of very similar SAT problems are solved in sequence [16]. The procedure in [34] does not support this, so we extended it.

Finally, because of the many variables in BMC SAT formulas, ParaFROST supports *Multiple Decision Making* (MDM) in the solving procedure, as presented in [30]. With MDM, multiple decisions can be made at once, periodically during the solving. In case there are many variables, there is more potential to make many decisions simultaneously. We have generalised the original MDM decision procedure [30], making it easier to integrate MDM in solvers other than MiniSat and Glucose [3]. The effectiveness of MDM in BMC has never been investigated before, nor has been combined with GPU pre- and inprocessing.

2 Background

SAT solving. We assume that SAT formulas are in conjunctive normal form (CNF). A CNF formula is a conjunction of m clauses $C_1 \wedge \cdots \wedge C_m$, and each clause C_i is a disjunction of n literals $\ell_1 \vee \cdots \vee \ell_n$. A literal is a Boolean variable x or its negation $\neg x$, also referred to as $\bar{x}$. The domain of all literals is $\mathbb{L}$. A clause can be interpreted as a set of literals, i.e., $\{\ell_1, \ldots, \ell_n\}$ encodes $\ell_1 \vee \ldots \vee \ell_n$, and a SAT formula $\mathcal{S}$ as a set of clauses, i.e., $\{C_1, \ldots, C_m\}$ encodes $C_1 \wedge \ldots \wedge C_m$. With $Var(C)$, we refer to the set of variables in C: $Var(C) = \{x \mid x \in C \vee \bar{x} \in C\}$. The set $\mathcal{S}_\ell$ consists of all clauses in $\mathcal{S}$ containing ℓ: $\mathcal{S}_\ell = \{C \in \mathcal{S} \mid \ell \in C\}$.

In CDCL, clauses are `LEARNT` or `ORIGINAL`. A `LEARNT` clause has been derived by the CDCL clause learning process during solving, and an `ORIGINAL` clause is part of the formula. We refer with $\mathcal{L}$ to the set of `LEARNT` clauses.

For a set of assignments Σ, consisting of all literals that have been assigned **true**, a formula $\mathcal{S}$ evaluates to **true** iff $\forall C \in \mathcal{S}.\exists \ell \in C.\ell \in \Sigma$. When a *decision* is made, a literal is picked and added to Σ. Each assignment is associated with a *decision level* (time stamp) to monitor the assignment order. We call a clause C *unit* iff a single literal in it is still unassigned, and the others are assigned **false**, i.e., $|\mathit{Var}(C) \setminus \mathit{Var}(\Sigma)| = 1$ and $C \cap \Sigma = \emptyset$.

***Variable-Clause Elimination* (VCE).** Variables and clauses can be removed from formulas by applying *simplification rules* [15, 20]. They rewrite a formula to an equi-satisfiable one with fewer variables and/or clauses. Applying them is referred to as pre- and inprocessing, before and during the solving, respectively.

Incremental Bounded Model Checking. Since 2001, incremental BMC has been applied to hardware and software verification [16, 39]. It relies on incremental SAT solving [16]. In CDCL, clauses are learnt during the solving each time a wrong decision has been made, to avoid making those decisions again in the future. Incremental SAT solving builds on this: when multiple SAT formulas with similar characteristics are solved sequentially, then in each iteration, the clauses learnt in previous iterations are reused. An efficient approach to add and remove clauses is by using *assumptions* [16], which are initial assignments.

For BMC, the transition relation of a system design and the (negation of) the property to be verified are encoded in a SAT formula. A predicate $\mathcal{I}(s_0)$ identifies the initial states, $\delta(s_i, s_{i+1})$ encodes the transition relation at trace depth i, and $\mathcal{E}(i) = \bigvee_{0 \leq j \leq i} e(s_j)$ encodes the reachability of an error state up to trace depth i, where $e(s_j)$ is **true** iff state s_j is an error state. For incremental BMC, additional unit clauses σ_i are used. These predicates are combined to define the following series of SAT formulas $\mathcal{S}(i)$ that must be solved incrementally:

$$\mathcal{S}(0) = \mathcal{I}(s_0) \wedge (\mathcal{E}(0) \vee \sigma_0), \text{ under assumption } \neg\sigma_0$$
$$\mathcal{S}(i+1) = \mathcal{S}(i) \wedge \delta(s_i, s_{i+1}) \wedge \sigma_i \wedge (\mathcal{E}(i+1) \vee \sigma_{i+1}), \text{ under assumption } \neg\sigma_{i+1}$$

Formula $\mathcal{S}(i)$ is satisfiable iff an error state is reachable via a trace with a length up to i [16, 39]. At iteration $i+1$, we know that $\mathcal{E}(i)$, included via $\mathcal{S}(i)$, cannot be satisfied (otherwise iteration $i+1$ would not have been started). This means that $\mathcal{E}(i)$ must be removed to avoid that $\mathcal{S}(i+1)$ is unsatisfiable. To effectively remove $\mathcal{E}(i)$, σ_i is assigned **true**, resulting in $\mathcal{E}(i) \vee \sigma_i$ being satisfied. In general, at iteration i, σ_i is assigned **false**, while in iterations $i' > i$, it is assigned **true**.

GPU Programming. CUDA [28] is NVIDIA's parallel computing platform that can be used to develop general purpose GPU programs. A GPU consists of multiple streaming multiprocessors (SMs), and each SM contains several streaming processors (SPs). A GPU program consists of a *host* part, executed on a CPU, and *device* functions, or *kernels*, executed on a GPU. Each time a kernel is launched, the number of threads that need to execute it is given. On the SPs, the threads are executed. Compared to a CPU thread, GPU threads perform a relatively simple task. In particular, they read some data, perform a computation, and write the result. This allows the SPs to switch contexts easily.

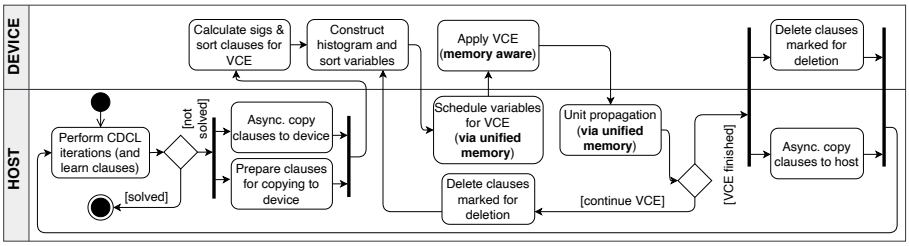


Fig. 2: An activity diagram for the workflow of ParaFROST.

In practice, one to two orders of magnitude more threads are typically launched than the number of SPs, which results in hiding the memory latency: whenever a thread is waiting for some data, the associated SP can switch to another thread.

A GPU has various types of memory. Relevant here are *registers* and *global* memory. Global memory is used to copy data between the host and the device. *Registers* are used for on-chip storage of thread-local data. Global memory has a much higher latency than registers. We use *unified memory* [28] to store clauses. Unified memory creates one virtual memory pool for host and device. In this way, the same memory addresses can be used by the host and the device, combining the main memory of the host side and the global memory of the device side.

3 GPU-Accelerated Bounded Model Checking

We implemented ParaFROST[2] with CUDA C++ v11. It is a hybrid CPU-GPU tool, with (sequential) solving done on the host side, and (parallel) VCE done on the device side. An interface with CBMC is implemented in C++. CBMC is patched to read a configuration file before ParaFROST is instantiated. This file contains all options supported by ParaFROST.

The workflow. Fig. 2 presents the general workflow of ParaFROST in the form of an activity diagram with host and device lanes. The diagram is focussed on inprocessing; preprocessing works similarly on the device. First, the host performs a predetermined number of solving iterations. Once those have finished, and (un)satisfiability has not yet been proven, relevant clause data is copied to the global memory. To hide the latency of this operation as much as possible, clauses are copied asynchronously in batches. One batch is copied while the next is formatted for the GPU, as not all clause information on the host side is relevant for the device (see the next paragraph on data structures). On the device, signatures are computed for fast clause comparison, and the clauses are sorted for VCE (more on VCE later). Next, the device constructs a histogram, for fast lookup of clauses, and sorts the variables. The Thrust library is used for sorting.[3] After that, the host *schedules variables* for VCE, marking those

[2] The tool is available at https://gears.win.tue.nl/software/gpu4bmc.
[3] https://docs.nvidia.com/cuda/thrust.

variables in the global memory using unified memory. Next, the device applies VCE, marking clauses to be removed as `DELETED`. The host propagates units (literals in unit clauses are assigned **true**), which directly has an effect on the formula in the global memory. The VCE procedure is repeated until it has been performed a predetermined number of times. After each time, `DELETED` clauses are removed, and after the last iteration, this is done while the new clauses are copied to the host. Once this has been done, the overall procedure is repeated.

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

- The `state` field (*2 bits*) stores if the state is `ORIGINAL`, `LEARNT` or `DELETED`.
- The `used` field (*2 bits*) keeps track of how many search iterations a `LEARNT` clause can still be used. `LEARNT` clauses are used at most twice [6].
- Two fields (*1 bit each*) are used for VCE bookmarking.
- The *literal block distance* (`lbd`) (*26 bits*) stores the number of decision levels contributing to a conflict, if there is one [3]. A maximum value of 2^{26} turns out to be sufficient. This field is updated when the clause is altered.
- The `size` (*32 bits*) of the clause, i.e., the number of literals.
- A signature `sig` (*32 bits*) is a clause hash, for fast clause comparison [15].

In addition, a list of literals is stored, each literal taking 32 bits (1 bit to indicate whether it is negated or not, and 31 bits to identify the variable). In total, a clause requires $12 + 4t$ bytes, with t the number of literals in the clause. For comparison, MiniSat only requires $4 + 4t$ bytes, but it does not involve the `used`, `lbd` and `sig` fields, thereby not supporting the associated optimisations. CaDiCaL [6] uses $28 + 4t$ bytes, since it applies solving and VCE on the same structures. In ParaFROST, the GPU is only used for VCE, in which information for *probing* [24] and *vivification* [36], for instance, is irrelevant. Finally, in [34], $20 + 4t$ bytes are used, storing the same information as ParaFROST.

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

Parallel* VCE.** ParaFROST supports the VCE rules *substitution* (i.e., gate equivalence reasoning), *resolution* (RES), *subsumption elimination* (SUB) and *eager redundancy elimination* (ERE) [15, 20]. Substitution applies to patterns representing logical gates, and substitutes the involved variables with their gate definitions. ParaFROST supports ***`AND/OR`, ***`Inverter`***, ***`If Then Else`*** and ***`XOR`***.

In Fig. 3, we provide rewrite rules for SUB and RES. If clauses exist in $\mathcal{S}$ of the form expressed by the left hand side of a rule, then the rule is applicable, and the involved clauses are replaced by the clauses (called *resolvents*) on the right hand side. RES is applicable if there are two clauses of the form $x \cup C_1$ and

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

Fig. 3: VCE rules in ParaFROST. C_1 and C_2 are non-empty sets of literals.

$\bar{x} \cup C_2$, and applying it results in replacing those with a clause $C_1 \cup C_2$. SUB consists of two rules; the second is applied once the first is no longer applicable.

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

VCE is applied in parallel by ParaFROST by scheduling sets of *mutually-independent* variables for analysis. Two variables x and y are independent in $\mathcal{S}$ iff $\mathcal{S}$ does not contain a clause containing literals that refer to both variables, i.e., $\mathcal{S}_x \cup \mathcal{S}_{\bar{x}}$ and $\mathcal{S}_y \cup \mathcal{S}_{\bar{y}}$ are disjoint. This ensures that two threads focussing on x and y, respectively, does not lead to data races. In incremental solving, variables referred to by assumptions must be excluded from VCE. In each VCE iteration, a different set Ψ of variables is selected. This is achieved by using an upper-bound μ for the number of occurrences of a variable in $\mathcal{S}$. After each iteration, μ is increased, allowing the selection of more variables. ParaFROST supports configuring μ and the number of VCE iterations.

As already mentioned, clauses that can be removed are marked `DELETED` before they are removed. The removal of clauses is done once VCE has finished (see Fig. 2) to avoid data races. However, because of this, VCE may at first require more memory to store clauses. The clauses added during VCE must fit in the memory, otherwise the procedure fails. To ensure this, we have developed a memory-aware mechanism for VCE. Next, we explain this mechanism for the RES rule and substitution, as the application of those rules results in new clauses.

Alg. 1 presents how RES and substitution are applied in ParaFROST. It requires $\mathcal{S}$, stored in a clause array `clauses`. As clauses are of varying sizes, we need an array `references` that provides a reference to each clause. In addition, arrays `varinfo`, `cindex` and `rindex` are given, which are filled in the first lines.

At line 1, the kernel VceScan is called in which a different thread is assigned to each variable $x \in \Psi$. Each thread checks the applicability of VCE rules on its variable and computes the number of clauses and literals that will be produced by the first applicable rule. A thread with ID i stores the type τ of the applicable rule (`NONE`, `RESOLVE`, or `SUBSTITUTE`) and the number of clauses β and literals γ produced by that rule in one integer at `varinfo`$[i]$. At lines 2-3, kernels computeClauseIndices and computeClauseRefIndices are called to add

Algorithm 1: Parallel memory-aware application of RES and substitution

```
   Input : global Ψ, clauses, references, varinfo, cindex, rindex
 1 varinfo ← VceScan(Ψ, S)
 2 cindex ← computeClauseIndices(varinfo, size(clauses))
 3 rindex ← computeClauseRefIndices(varinfo, size(references))
 4 VceApply(Ψ, clauses, references, varinfo, cindex, rindex)
 5 kernel VceApply(Ψ, clauses, references, varinfo, cindex, rindex):
 6     for all i ∈ [0, |Ψ|⟩ do in parallel
 7         register cidx ← cindex[i], ridx = rindex[i]
 8         register τ, β, γ ← varinfo[i]
 9         if τ = RESOLVE ∧ memorySafe(ridx, cidx, β, γ) then
10             ResApply(clauses, references, x, ridx, cidx)
11         if τ = SUBSTITUTE ∧ memorySafe(ridx, cidx, β, γ) then
12             SubApply(clauses, references, x, ridx, cidx)
13 device function memorySafe(ridx, cidx, β, γ):
14     reqSpace ← cidx + 12 × β + (4 · γ)                    // required number of bytes
15     if reqSpace > capacity(clauses) then return false
16     numRefs ← ridx + β                                    // required number of clause references
17     if numRefs > capacity(references) then return false
18     return true
```

up the β's and γ's to obtain offsets into the arrays `references` and `clauses` (the method size(A) refers to the amount of data in array A). Both methods apply a parallel exclusive prefix sum [37], involving the β's and γ's. The result is that thread i, assigned to x, is instructed to start writing clause references at `references`[`rindex`[i]] and clauses at `clauses`[`cindex`[i]] when applying the next VCE rule for x. Whether the data actually fits is checked later.

Next, the kernel VceApply is called (lines 5-12). To each variable in Ψ, a thread is assigned. It retrieves the precomputed data (lines 7-8) and either applies the RES rule (lines 9-10), substitution (lines 11-12), or nothing, in case $\tau =$ `NONE`. However, a condition for applying a rule is that there is enough space, which is checked using the device function memorySafe (lines 13-18). The amount of allocated space for A is reflected by capacity(A), and memorySafe checks if there is enough space in `clauses`, starting at $cidx$ (lines 14-15). If there is, it is checked if the references can be stored in `references` (lines 16-17).

4 Multiple Decision Making in Incremental Solving

Given the fact that BMC SAT formulas often have many variables, a recently proposed extension of CDCL [30], in which periodically multiple decisions are made (MDM) at the same time, has much potential to speed up BMC. When the MDM method is called, it constructs a set $\mathcal{M} = \{\ell \in \mathbb{L} \mid Var(\{\ell\}) \cap Var(\Sigma) = \emptyset\}$ such that there does not exist a clause $C \in \mathcal{S}$ with $|Var(C) \setminus Var(\Sigma)| = 1$. In other words, the decisions $\mathcal{M}$ do not lead to logical follow-up assignments, i.e., implications. The reason for this restriction is that implications may lead to conflicts (clauses that cannot be satisfied). When a single decision is made, this decision needs to be rolled back when a conflict is caused, but when multiple decisions are made, detecting which decisions actually cause a conflict is more

Algorithm 2: Decision making method DECIDE, with integrated MDM

```
   Input: Σ, 𝕃, decqueue, r, nConflicts, ConfFactor, prevMDsize
1  freevars ← Var(𝕃) \ Var(Σ)
2  if r > 0 then
3      M ← MDM(freevars, decqueue)
4      r ← r − 1, prevMDsize ← |M|
5  else
6      M ← singleDecision(freevars, decqueue)
7  if r = 0 ∧ |freevars| ≥ prevMDsize then
8      r ← periodicFuse(nConflicts, ConfFactor)
9  return M
10 function periodicFuse(nConflicts, ConfFactor):
11     if nConflicts ≥ ConfFactor then
12         updateFactor(ConfFactor)
13         return mdmrounds
14     else
15         return 0
```

difficult. Note that MDM cannot always make multiple decisions; implications are needed to solve a formula, so single decisions still have to be made frequently.

In [30], MDM was integrated into MiniSat and Glucose, and since multiple decisions should be selected periodically, a mechanism was proposed that decides when to make multiple decisions based on the solver restart policy. However, since solvers can differ greatly in this policy, we wanted to create an alternative mechanism not depending on this. ParaFROST is based on CaDiCaL [6], which has a very different restart policy compared to MiniSat and Glucose.

Alg. 2 presents ParaFROST's DECIDE method, which is called every time a decision must be made. Besides Σ and $\mathbb{L}$, it is given a queue *decqueue*, in which the variables are ordered based on a decision heuristic. In ParaFROST, the heuristics Variable State Independent Decaying Sum (VSIDS) [27] and Variable Move-To-Front (VMTF) [7] are alternatingly used. The latter was not used before in [30]. DECIDE also gets a variable r, initially set to the constant `mdmrounds`. These values are used to control the periodic call of MDM, in which a set of multiple decisions is made per round. Experiments have shown that `mdmrounds` $= 3$ is effective [30]. Finally, the number of conflicts so far (*nConflicts*), a variable *ConfFactor* used to switch MDM on and off, and a variable *prevMDsize*, storing the size of the most recent set of multiple decisions, are given.

To select new decisions, the set of unassigned variables is created at line 1. If we are calling MDM `mdmrounds` times (line 2), then MDM is called again and r is updated. The alternative is to make a single decision (line 6). If we have stopped calling MDM, and enough unassigned variables are present (line 7), method periodicFuse is called, which either sets r back to `mdmrounds` or to 0, depending on *nConflicts* (lines 10-15). There are enough unassigned variables if there are more unassigned variables than variables in the most recent multiple decisions set. In periodicFuse, *nConflicts* is compared to *ConfFactor*, which is initially set to a configurable value (default 2,000). *ConfFactor* is updated using a function updateFactor. This makes *ConfFactor* grow linearly, to achieve a suitable balance between *ConfFactor* and *nConflicts* as the solving progresses.

5 Benchmarks

We conducted experiments with CBMC in combination with MiniSat (the default), Glucose, CaDiCaL, ParaFROST, ParaFROST with MDM, and a CPU-only version, referred to as ParaFROST (noGpu).[4] We used the AWS benchmarks in which the data structures `hash table`, `array list`, `array buff`, `linked list`, `priority queue`, `byte cursor` and `string` were analysed. The loop unwinding upper-bounds `8`, `16`, `64`, `128` and `1,000` were used, resulting in 168 different verification problems.

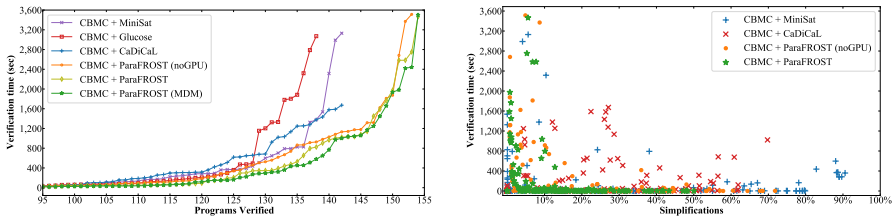
All experiments were executed on the DAS-5 cluster [4]. Each program was verified in isolation on a separate node, with a time-out of 3,600 seconds. Each node had an Intel Xeon E5-2630 CPU (2.4 GHz) with 64 GB of memory, and an NVIDIA RTX 2080 Ti, with 68 SMs (64 cores/SM) and 11 GB global memory.

Fig. 4 presents the decision procedure runtime, and how much time was spent on VCE. ParaFROST outperforms all sequential solvers including CaDiCaL (plot 4a). Even though ParaFROST is based on CaDiCaL, its different data structures, simplification mechanism and parameters tuned for large formulas makes ParaFROST more effective in these experiments. MDM further improves ParaFROST. Plot 4b demonstrates that CBMC with MiniSat often spends most of the time on VCE. ParaFROST significantly reduces the time spent on VCE compared to other solvers.

In Table 1, the `Verified` column lists per solver the number of verified programs, and `PAR-2` gives the *penalized average runtime-2* metric. `PAR-2` score accumulates the running times of all solved instances with 2× the time-out of unsolved ones, divided by the total number of formulas. The solver with the lowest score is the winner. The triangles ▲ and ▼ mean significantly better and worse, respectively. The `MiniSat` column lists how many programs were verified faster with the other solvers compared to MiniSat. Between parentheses, it is given how many of those programs were not solved by MiniSat at all. The final four columns serve the same purpose for the other solvers. For example, ParaFROST-MDM verified 123 programs faster than CaDiCaL, in which 12 could not be verified by the latter. The last two rows provide a similar comparison. Clearly, ParaFROST-MDM verified the largest number of programs, with the lowest score.

Fig. 5 presents the speedups of the ParaFROST configurations for the individual cases. Overall, SAT solving was accelerated effectively with ParaFROST and ParaFROST-MDM. Compared to ParaFROST (noGpu), ParaFROST (and ParaFROST-MDM), accelerated multiple instances by up to 18× (and 27×), and the geometric average speedup for all programs was 1.3× (and 1.6×).

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



(a) Verification time (timeout: 3600 seconds)

(b) Percentage of verification time used for VCE

Fig. 4: CBMC runtimes for all solvers over the benchmark suite.

Table 1: CBMC performance analysis using the various solvers.

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

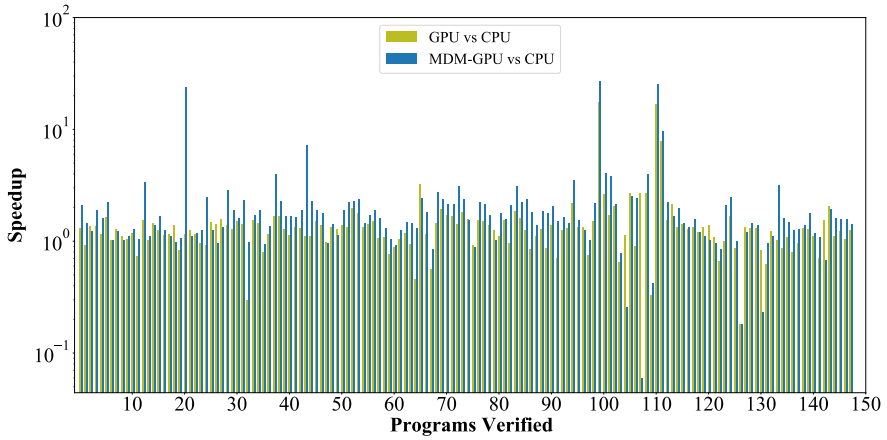


Fig. 5: Speedups of the individual cases.

6 Conclusion

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

future, we will combine our approach with (existing) multi-threaded BMC. We expect these techniques to strengthen each other.

References

1. Ábrahám, E., Schubert, T., Becker, B., Fränzle, M., Herde, C.: Parallel SAT Solving in Bounded Model Checking. Journal of Logic and Computation **21**(1), 5–21 (2009)
2. Amazon: The Amazon Web Services Core C99 Package Benchmark Set (2021), https://github.com/awslabs/aws-c-common/tree/main/verification/cbmc/proofs
3. Audemard, G., Simon, L.: Predicting Learnt Clauses Quality in Modern SAT Solvers. In: IJCAI. pp. 399–404. Morgan Kaufmann Publishers Inc. (2009)
4. Bal, H., Epema, D., de Laat, C., van Nieuwpoort, R., Romein, J., Seinstra, F., Snoek, C., Wijshoff, H.: A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. IEEE Computer **49**(5), 54–63 (2016)
5. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. pp. 193–207. Springer (1999)
6. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: SAT Competition 2020. pp. 51–53 (2020)
7. Biere, A., Fröhlich, A.: Evaluating CDCL Variable Scoring Schemes. In: SAT. LNCS, vol. 9340, pp. 405–422. Springer (2015)
8. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In: PDMC-HiBi. pp. 17–19. IEEE Computer Society (2010)
9. Bošnački, D., Odenbrett, M., Wijs, A., Ligtenberg, W., Hilbers, P.: Efficient reconstruction of biological networks via transitive reduction on general purpose graphics processors. BMC Bioinformatics **13**(281) (2012)
10. Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI. LNCS, vol. 6538, pp. 70–87. Springer (2011)
11. Brown, C.E.: Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems. Journal of Automated Reasoning **51**(1), 57–77 (Jun 2013)
12. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L.: Sequential Circuit Verification using Symbolic Model Checking. In: DAC. pp. 46–51. IEEE (1990)
13. Chatterjee, P., Roy, S., Diep, B., Lal, A.: Distributed Bounded Model Checking. In: FMCAD. pp. 47–56. TU Wien Academic Press (2020)
14. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004)
15. Eén, N., Biere, A.: Effective Preprocessing in SAT Through Variable and Clause Elimination. In: SAT. LNCS, vol. 3569, pp. 61–75. Springer (2005)
16. Eén, N., Sörensson, N.: Temporal Induction by Incremental SAT Solving. Electronic Notes in Theoretical Computer Science **89**(4), 543 – 560 (2003)
17. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: SAT. LNCS, vol. 2919, pp. 502–518. Springer (2004)
18. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: A Parallel SAT Solver. Journal on Satisfiability **6**, 245–262 (2009)
19. Inverso, O., Trubiani, C.: Parallel and Distributed Bounded Model Checking of Multi-Threaded Programs. In: PPoPP. pp. 202–216. ACM (2020)
20. Järvisalo, M., Heule, M.J., Biere, A.: Inprocessing Rules. In: IJCAR. LNCS, vol. 7364, pp. 355–370. Springer (2012)

21. Kahsai, T., Tinelli, C.: PKind: A parallel k-induction based model checker. In: PDMC. EPTCS, vol. 72, pp. 55–62. Open Publishing Association (2011)
22. Kroening, D., Strichman, O.: Decision Procedures. Springer (2016)
23. Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D., Schardl, T.B.: There's plenty of room at the Top: What will drive computer performance after Moore's law? Science **368**(6495) (2020)
24. Lynce, I., Marques-Silva, J.: Probing-based preprocessing techniques for propositional satisfiability. In: ICTAI. pp. 105–110. IEEE (2003)
25. Marques-Silva, J., Glass, T.: Combinational equivalence checking using satisfiability and recursive learning. In: DATE. pp. 145–149. ACM (March 1999)
26. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5), 506–521 (1999)
27. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC. pp. 530–535. ACM (2001)
28. NVIDIA: CUDA C Programming Guide (2020), https://docs.nvidia.com/cuda/cuda-c-programming-guide
29. Oostema, P., Martins, R., Heule, M.: Coloring Unit-Distance Strips using SAT. In: LPAR23. EPiC Series in Computing, vol. 73, pp. 373–389. EasyChair (2020)
30. Osama, M., Wijs, A.: Multiple Decision Making in Conflict-Driven Clause Learning. In: ICTAI. pp. 161–169. IEEE (2020)
31. Osama, M., Gaber, L., Hussein, A.I., Mahmoud, H.: An Efficient SAT-Based Test Generation Algorithm with GPU Accelerator. JET **34**(5), 511–527 (Oct 2018)
32. Osama, M., Wijs, A.: Parallel SAT Simplification on GPU Architectures. In: TACAS. LNCS, vol. 11427, pp. 21–40. Springer, Cham (2019)
33. Osama, M., Wijs, A.: SIGmA: GPU Accelerated Simplification of SAT Formulas. In: iFM. LNCS, vol. 11918, pp. 514–522. Springer (2019)
34. Osama, M., Wijs, A., Biere, A.: SAT Solving with GPU Accelerated Inprocessing. In: TACAS. LNCS, vol. 12651, pp. 133–151. Springer (2021)
35. Phan, Q.S., Malacaria, P., Pasareanu, C.: Concurrent Bounded Model Checking. ACM SIGSOFT Software Engineering Notes **40**(1), 1–5 (2015)
36. Piette, C., Hamadi, Y., Saïs, L.: Vivifying Propositional Clausal Formulae. In: ECAI. pp. 525–529. IOS Press, NLD (2008)
37. Sengupta, S., Harris, M., Garland, M., Owens, J.: Efficient Parallel Scan Algorithms for Manycore GPUs. In: SCMA, pp. 413–442. Taylor & Francis (2011)
38. Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: FMCAD. LNCS, vol. 1954, pp. 108–125. Springer (2000)
39. Shtrichman, O.: Pruning Techniques for the SAT-Based Bounded Model Checking Problem. In: CHARME. LNCS, vol. 2144, pp. 58–70. Springer (2001)
40. Wijs, A.: BFS-based model checking of linear-time properties with an application on GPUs. In: CAV, Part II. LNCS, vol. 9780, pp. 472–493. Springer (2016)
41. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: Unleashing GPU Explicit-state Model Checking. In: FM. LNCS, vol. 9995, pp. 694–701. Springer (2016)
42. Youness, H., Ibraheim, A., Moness, M., Osama, M.: An Efficient Implementation of Ant Colony Optimization on GPU for the Satisfiability Problem. In: PDP. pp. 230–235 (2015)