SIGmA: GPU Accelerated Simplification of SAT Formulas

Muhammad Osama(✉) and Anton Wijs(✉)

Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands
{o.m.m.muhammad,a.j.wijs}@tue.nl

Abstract. We present SIGmA (SAT sImplification on GPU Architectures), a preprocessor to accelerate SAT solving that runs on NVIDIA GPUs. We discuss the tool, focussing on its full functionality and how it can be used in combination with state-of-the-art SAT solvers. SIGmA performs various types of simplification, such as variable elimination, subsumption elimination, blocked clause elimination and hidden redundancy elimination. We study the effectiveness of our tool when applied prior to SAT solving. Overall, for our large benchmark set of problems, SIGmA enables MiniSat and Lingeling to solve many problems in less time compared to applying the SatElite preprocessor.

Keywords: Boolean satisfiability · SAT decomposition · Parallel SAT preprocessing · Multi-GPU computing

1 Introduction

Simplifying SAT formulas prior to solving them has proven to be effective in modern SAT solvers [2,5], particularly when applied on SAT formulas encoding software and hardware verification problems [7]. Many techniques based on variable elimination, clause elimination, and equivalence reasoning are being used to simplify SAT formulas [4,6,11]. However, applying variable and clause elimination iteratively to large formulas may actually be a performance bottleneck, or increase the number of literals, negatively impacting the solving time.

Graphics processors (GPUs) have become attractive for general-purpose computing with the availability of the Compute Unified Device Architecture (CUDA) programming model.[1] CUDA is widely used to accelerate applications that are computationally intensive w.r.t. data processing and memory access. For instance, we have applied GPUs to accelerate explicit-state model checking [3,12–15], metaheuristic SAT solving [16], and SAT-based test generation [9]. SIGmA is the first SAT simplifier to exploit GPUs.

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). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the GeForce Titan Xp's used for this research.

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide.

In related work, Eén et al. [4] provided the first powerful preprocessor (SatElite) which applies variable elimination with subsumption elimination. However, the subsumption check is only performed on the clauses resulting from variable elimination, hence no reductions are obtained if there are no variables to resolve. Gebhardt et al. [6] presented the first attempt to parallelise SAT preprocessing on a multi-core CPU using a locking scheme to prevent threads corrupting the SAT formula. Yet, they reported a limited speedup of 1.88× on average when running on 8 cores. The methods above may still consume considerable time when processing large problems.

Contributions. We present the SIGmA tool to accelerate SAT simplification on CUDA-supported NVIDIA GPUs. In earlier work [10], we presented parallel algorithms for variable elimination, subsumption elimination and self-subsuming resolution. Implementations of these are now publicly available in SIGmA. Moreover, in this work, we have added new implementations for *blocked clause elimination* (BCE) and a new type of elimination we call *hidden redundancy elimination* (HRE). Finally, we propose a generalisation of all algorithms to distribute simplification work over multiple GPUs, if these are available in a single machine. We discuss the potential performance gain of SIGmA and its impact on SAT solving for 344 problems, a larger set than considered previously in [10].

2 SIGmA Functionality and Architecture

SIGmA is developed in CUDA C/C++ version 9.2, runs on Windows and Linux, and requires a CUDA-capable GPU with at least compute capability 3.5 (see Footnote 1). It is freely available at https://gears.win.tue.nl/software. The tool is published with a large set of benchmarks and detailed documentation.

SIGmA accepts as input a SAT formula in Conjunctive Normal Form (CNF), stored in the DIMACS format.[2] A CNF formula is a conjunction of *clauses*, where each clause C_i is a disjunction of *literals*, and each literal ℓ_i is a Boolean variable x or its negation $\bar{x}$ (or $\neg x$). Below, we interpret a formula as a set of clauses, and a clause as a set of literals. The output of SIGmA is a simplified CNF formula in the DIMACS format, and hence can directly be used as input for state-of-the-art SAT solvers.

SIGmA's architecture is depicted in Fig. 1. First, we consider running SIGmA on a single GPU. Once an input formula has been parsed and loaded into the GPU global memory, an Occurrence Table (OT) is created using atomic write operations. This OT stores for each variable x references to all clauses containing either x or $\bar{x}$. Next, a configured combination of simplifications can be applied. The various types of simplification supported by the tool are:

– *Variable elimination* (VE). This eliminates a set of variables by applying *resolution* [11] and *substitution* (also known as gate equivalence reasoning) [4]. When resolution is applied for a variable x, pairs of clauses C_1, C_2 with

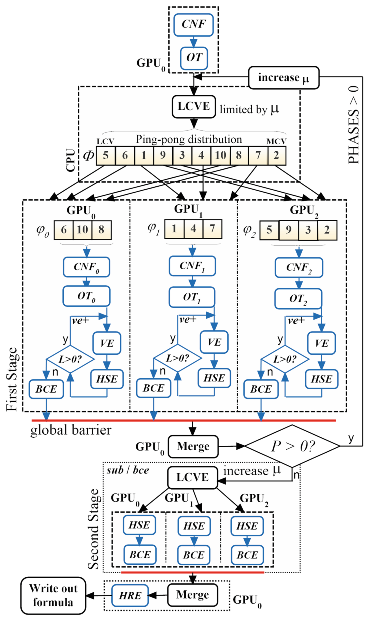
[2] See https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html.

$x \in C_1$, $\bar{x} \in C_2$ are combined into a new clause $C = C_1 \cup C_2 \setminus \{x, \bar{x}\}$, called a *resolvent*. We add such a resolvent C to the formula iff C is not a *tautology*, i.e., there exists no variable y for which both $y \in C$ and $\bar{y} \in C$. Substitution detects patterns encoding logical gates, and substitutes the involved variables with their gate-equivalent counterparts. For instance, in the formula $\{\{x, \bar{a}, \bar{b}\}, \{\bar{x}, a\}, \{\bar{x}, b\}, \{x, c\}\}$, the first three clauses together encode a logical AND-gate, hence in the final clause, we can substitute $a \wedge b$ for x. After removal of the AND-gate, we end up with the new formula $\{\{a, c\}, \{b, c\}\}$. SIGmA supports substitution for both AND- and OR-gates. The SAT-encoded representation of the OR gate $x = a \vee b$ is $\{\{\bar{x}, a, b\}, \{x, \bar{a}\}, \{x, \bar{b}\}\}$.

- *Hybrid subsumption elimination* (HSE) [10]. It performs *self-subsuming resolution* followed by *subsumption elimination* [4]. The former can be applied on clauses C_1, C_2 iff for some variable x, we have $C_1 = C_1' \cup \{x\}$, $C_2 = C_2' \cup \{\bar{x}\}$, and $C_2' \subseteq C_1'$. In that case, x can be removed from C_1. The latter is applied on clauses C_1, C_2 with $C_2 \subseteq C_1$. In that case, C_1 is redundant and can be removed. For instance, consider the formula $\mathcal{S} = \{\{a, b, c\}, \{\bar{a}, b\}, \{b, c, d\}\}$. The first clause is self-subsumed by the second clause over variable a and can be strengthened to $\{b, c\}$ which in turn subsumes the last clause $\{b, c, d\}$. The subsumed clause is removed from $\mathcal{S}$ and the simplified formula will be $\{\{b, c\}, \{\bar{a}, b\}\}$.
- *Blocked clause elimination* (BCE) [8]. It removes clauses on which variable elimination can be applied, but doing so results only in tautologies. Consider the formula $\{\{a, b, c, d\}, \{\bar{a}, \bar{b}\}, \{\bar{a}, \bar{c}\}\}$. Both the literals a and c are blocking the first clause, since resolving a produces the tautologies $\{\{b, c, d, \bar{b}\}, \{b, c, \bar{c}, d\}\}$. Likewise, resolving c yields the tautology $\{\{a, b, \bar{a}, d\}$. Hence the blocked clause $\{a, b, c, d\}$ can be removed from $\mathcal{S}$.
- *Hidden redundancy elimination* (HRE) is a new elimination procedure 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 $\bar{x} \in C_2$ for some variable x, if there exists a clause $C \in S$ for which $C \equiv C_1 \otimes_x C_2$ and C is not a tautology, then let $\mathcal{S} := \mathcal{S} \setminus \{C\}$. The clause C is called a *hidden redundancy* and can be removed without altering the original satisfiability. For example, consider the formula
$$\mathcal{S} = \{\{a, \bar{c}\}, \{c, b\}, \{\bar{d}, \bar{c}\}, \{b, a\}, \{a, d\}\}$$
Resolving the first two clauses gives the resolvent $\{a, b\}$ which is equivalent to the fourth clause in $\mathcal{S}$. Also, resolving the third clause with the last clause yields $\{a, \bar{c}\}$ which is equivalent to the first clause in $\mathcal{S}$. HRE can remove either $\{a, \bar{c}\}$ or $\{a, b\}$ but not both.

The general workflow of SIGmA and the challenges are discussed next.

Variable Dependency and Completeness. To exploit parallelism in SIGmA, each simplification is applied on several variables simultaneously. Doing so is non-trivial, since variables may *depend* on each other; two variables x and y are dependent iff there exists a clause C with $(x \in C \vee \bar{x} \in C) \wedge (y \in C \vee \bar{y} \in C)$. If both x and y were to be processed for simplification, two threads might manipulate C at the same time. To guarantee soundness and completeness, we apply



Tool Options	Functionality
VE	enables variable elimination only in the first stage
HSE	enables VE followed by 1-time hybrid subsumption elimination in the first stage
VE+	iterate over VE & HSE until no literals (*L*) can be removed in the first stage
BCE	enables blocked clause elimination
PHASES=<*n*>	allows (*n*) number of iterations (*P*) of the first stage
SUB	enables 1-time HSE in the second stage
HRE	enables hidden redundancy elimination on a single GPU in the second stage
GPUS=<*n*>	activate (*n*) number of GPUs

Fig. 1. SIGmA architecture with supported options.

our *least constrained variable elections* algorithm (LCVE) [10]. This algorithm is responsible for electing a set of mutually independent variables from a set of authorised candidates. The remaining variables relying on the elected ones are frozen. These notions are defined by Definitions 1–4.

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

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

Definition 2 (Candidate Dependency Relation). *We call a relation $\mathcal{D} : \mathcal{A} \times \mathcal{A}$ a* candidate dependency relation *iff $\forall x, y \in \mathcal{A}$, $x \,\mathcal{D}\, y$ implies that $\exists C \in S.(x \in C \vee \bar{x} \in C) \wedge (y \in C \vee \bar{y} \in C)$.*

Definition 3 (Elected candidates). *Given a set of authorised candidates $\mathcal{A}$, we call a set $\Phi \subseteq \mathcal{A}$ a set of* elected candidates *iff $\forall x, y \in \Phi.\ \neg(x \,\mathcal{D}\, y)$.*

Definition 4 (Frozen candidates). *Given the sets $\mathcal{A}$ and Φ, the set of* frozen candidates *$\mathcal{F} \subseteq \mathcal{A}$ is defined as $\mathcal{F} = \{x \mid x \in \mathcal{A} \wedge \exists y \in \Phi.\ x \,\mathcal{D}\, y\}$.*

SIGmA Modes of Operation. SIGmA simplifies formulas in two stages (Fig. 1). In the first stage, variable elimination is applied. Hence, the first stage is executed only if VE is selected, and may in addition apply HSE and/or BCE. Before applying VE, a parallel GPU algorithm is run to estimate the number of resolvents that will be produced, in order to appropriately allocate memory. HSE can be executed after VE to remove or strengthen (self)-subsumed clauses of non-eliminated variables. VE and HSE can be applied iteratively until no more literals can be removed, with the VE+ option. The `phases=<n>` option (outer loop for the first stage in Fig. 1), applies the first stage for a configured number of iterations, with increasingly large values of the threshold μ.

The second stage is entirely focussed on eliminating redundant clauses, using a configured combination of HSE/BCE/HRE.

Multi-GPU Support. By default SIGmA runs on the first GPU of the computing machine, i.e., the one installed on the first PCI-Express bus, which we refer to as GPU_0. The command `gpus=<n>` utilises SAT simplification on n identical GPUs installed in the same machine. When $n > 1$, the variables in Φ are distributed evenly among the GPUs. This distribution is based on the number of literal occurrences of each variable. Figure 1 shows an example of such a distribution, after LCVE in the first stage. The variables are ordered by the number of literals in the formula; the variable with ID 5 has the smallest number, while the last one with ID 2 has the largest number. These variables are distributed over the GPUs in a 'ping-pong' fashion: in the example, starting with the variable with the largest number of literals, the first three variables are distributed over the three GPUs, followed by the next three, which are distributed in reversed order, etc. Finally, in general, in case the number of variables is not a multiple of the number of GPUs, the left-over variables are assigned to GPU_0, which in the example is the case for variable 5. As the number of literals is indicative of the computational effort needed to eliminate a variable, this distribution method achieves good load balancing.

At the end of the distribution, every GPU d gets its own set of elected variables Φ_d. For n GPUs, we say that $\bigcup_{0 \leq d < n} \Phi_d = \Phi$ and for all $0 \leq d, d' < n$ $(d \neq d')$, we have $\Phi_d \cap \Phi_{d'} = \emptyset$. We can now define the subformula of $\mathcal{S}$ assigned to GPU d based on the occurrence list of each variable in Φ_d as follows.

Definition 5 (GPU sub-formula). *Given the input formula $\mathcal{S}$ and a set of elected variables Φ_d, we define subformula $\mathcal{S}_d \subseteq \mathcal{S}$ as*

$$\mathcal{S}_d = \{C \mid C \in \mathcal{S} \wedge \exists x \in \Phi_d . x \in C \vee \bar{x} \in C\}$$

Definition 6 (non-elected sub-formula). *Given the input formula $\mathcal{S}$ and set of elected variables Φ. The* non-elected sub-formula *$\mathcal{S}_{ne} \subseteq \mathcal{S}$ is defined as*

$$\mathcal{S}_{ne} = \{C \mid C \in \mathcal{S} \wedge \neg\exists x \in \Phi . x \in C \vee \bar{x} \in C\}$$

The remaining part of $\mathcal{S}$ that belongs to the non-elected variables (Definition 6) can be processed by GPU_0. In Fig. 1, the global barrier acts as a synchronisation point for all GPUs, and in the subsequent *merge* step, all simplified subformulas are sent to GPU_0. HRE cannot be run on multiple GPUs, because the entire formula must be accessed.

Table 1. SIGmA performance analysis with CPU and one GPU configurations.

Mode	`-ve+ -sub -bce -hre`					`-ve -bce -hre`			
Method	**ve+**		**bce**	**hre**	**all**	**ve**	**bce**	**hre**	**ve/bce/hre**
	ve	**hse**							
Average Speedup	36.5×	6.4×	29.5×	17×	14×	24×	4×	12×	10×
#Formulas simp. faster					329 (95%)				314 (92%)

Table 2. SIGmA performance analysis with one and two GPU configurations.

Mode	`-ve+ -sub -bce`					`-ve -bce`			
Method	**ve+**		**bce**	**all**(-t_c)	**all**(+t_c)	**ve**	**bce**	**ve/bce**(-t_c)	**ve/bce**(+t_c)
	ve	**hse**							
Average Speedup	1.7×	2.64×	1.34×	1.96×	0.85×	2×	5.3×	2.6×	1×
#Formulas simp. faster					79 (22%)				129 (38%)

3 Benchmarks

We evaluated SIGmA using two NVIDIA Titan Xp GPUs. Each GPU has 30 streaming multiprocessors, with 128 cores each, 12 GB global memory and 48 KB shared memory. The GPU machine is running Linux Mint 18.3, has a 3.5 GHz Intel Core i5 CPU, and 32 GB of memory.

We selected 344 SAT problems from the application track of the 2013–2017 SAT competitions.[3] This set consists of all problems from that track that are more than 1 MB in file size. The largest size of problems occurring in this set is 1.2 GB. These problems have been encoded from 46 real-world applications with dissimilar logical properties. Before applying simplifications, any occurring unit clauses (clauses with a single literal) were propagated. Unit clauses immediately lead to simplification. By eliminating them, the results more clearly indicate the impact of SIGmA.

In the experiments, we involved a CPU-only version of SIGmA and the SatELite preprocessor [4] for simplification, and the MiniSat and Lingeling [2] SAT solvers for solving. We chose Minisat as it forms the basis for many CDCL SAT solvers, and Lingeling, since it was the winner of several SAT competitions. The CPU-version of SIGmA applies the same simplifications on the elected variables as the GPU version, but performs them sequentially. All these were executed on the compute nodes of the DAS-5 cluster [1]. Each problem was analysed in isolation on a separate node. Each node has an Intel Xeon E5-2630 2.4 GHz CPU with 128 GB memory, and runs on the CentOS 7.4 operating system. We performed the equivalent of four years of uninterrupted processing on a single node to measure how SIGmA impacts SAT solving.

For all experiments, we set μ initially to 64, which in practice tends to produce good results. Tables 1 and 2 summarise the amount of speedup and the number

[3] See http://www.satcompetition.org.

Table 3. SIGmA performance compared to SatElite and Lingeling simplifiers.

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

of problems simplified faster by running SIGmA on one and two GPUs.[4] We compare the single GPU mode of SIGmA with both the CPU-only version and the two GPUs mode of SIGmA. For the CPU-GPU comparison, we used two modes, (`-ve+ -sub -bce -hre`) and (`-ve -bce -hre`) which represent one full iteration of the first stage, followed by the second stage (see Fig. 1). The former is called `all` in Table 1. For the hybrid mode (`-ve+`), we measured the acceleration achieved by using a GPU for both `ve` and `hse` as part of `ve+`. It appears that the speedup obtained by `bce` is influenced by the application of `hse`, while other methods (`ve, hre`) maintain similar speedups. Compared to CPU-only SIGmA, the GPU achieves an acceleration of up to 36.5×. The average speedups of `ve/bce/hre` and `all` modes are 10× and 14×, respectively.

For the comparison of SIGmA's single/multiple GPU modes, we disabled `-hre`, which is only supported in single GPU mode. If we ignore the time needed for data transfer between the GPUs (t_c), SIGmA's runtime scales very well with the number of GPUs. In mode (`-ve+ -sub -bce`), the average acceleration is 2.64×, and overall, the average speedup is 1.96×. When we consider data transfer, the latter drops to 0.85×. Still, disabling `hse` in the second mode (`-ve+ -sub -bce`) positively influences the performance of `bce`. The speedup of this method has improved for the multi-GPU configuration by 5 times higher compared to using a single GPU with `hse` enabled. The overall speedup with and without data transfer has grown to 2.6× and 1×, respectively.

Moreover, in the second mode, SIGmA managed to simplify 129 (38%) problems faster compared to the single GPU mode, even if the communication overhead is taken into account. We expect that hardware improvements of inter-GPU communication in the future will make the multi-GPU mode of SIGmA increasingly attractive.

Table 3 compares SIGmA against the best sequential simplifiers available (SatElite and the preprocessing module of Lingeling, which is very similar to SatElite). Similar to the multi-GPU experiments, we used the heavy mode (`-ve+`) in combination with other simplifications in all of SIGmA's benchmarks. On average, CPU SIGmA is faster than SatElite by 37× since we only consider eliminating elected variables and exclude those occurring in the resolvents (their length grows exponentially by resolution). Moreover, the OT is created once before elimination. The GPU-version, with all simplifications enabled, beats SatElite and Lingeling by accelerations up to 49× and 32× respectively.

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

Table 4. MiniSat solving of original and simplified formulas (timeout: 24 h).

Evaluation	MiniSat (org)	SIGmA+MiniSat				SatElite+MiniSat	
		ve+	ve/bce	ve/hre	ve/bce/hre	ve	ve/sub
#Formulas solved	192 (56%)	228 (66%)	227 (66%)	218 (63%)	230 (67%)	215 (62%)	201 (58%)
Processing time (h)	3989	3201	3256	3354	3214	3474	3754

Table 5. Lingeling solving of original and simplified formulas (timeout: 24 h).

Evaluation	Lingeling (org)	SIGmA+Lingeling			
		ve+	ve/bce	ve/hre	ve/bce/hre
#Formulas solved	252 (73%)	282 (82%)	283 (82.3%)	281 (82%)	283 (82.3%)
Processing time (h)	2566	1826	1862	1880	1854

For the subsequent experiments, we set `phases` to 5, and doubled μ each time before SIGmA performed another iteration of the first stage. With this setup, Table 4 shows the impact of SIGmA in various modes on SAT solving when combined with MiniSat. The processing time includes the solving times of the entire set (344) up to the timeouts, and the simplification and data transfer times in case of SIGmA and SatElite. Based on the experiments, we conclude that the new simplifications (BCE, HRE) proposed in this paper, when combined with all other options, allow 230 problems (67%) to be solved, thereby outperforming all the alternatives. Moreover, the processing time of the (`-ve -bce -hre`) mode (3,214 h) is still shorter than when MiniSat is applied without simplification, and when SatElite is used for simplification. Likewise, Table 5 shows SIGmA's impact on the Lingeling solver. Again, mode (`-ve -bce -hre`) is at least as good as all the alternatives, allowing 283 instances (82.3%) to be solved. The best competitor, mode (`-ve -bce`), took more time to be applied.

4 Conclusion

We have presented the SIGmA tool, the first simplifier for SAT formulas that exploits the power of GPUs. It can be configured to apply a combination of various elimination procedures, among which is a new one (HRE) proposed by us. Experimentally, we have demonstrated the impact of SIGmA on state-of-the-art SAT solving. In particular, our new mode, involving BCE and HRE, positively affects both the solving speed and the ability to solve formulas, when using the MiniSat and Lingeling solvers.

References

1. Bal, H., et al.: A medium-scale distributed system for computer science research: infrastructure for the long term. IEEE Comput. **49**(5), 54–63 (2016)
2. Biere, A.: Lingeling, plingeling and treengeling entering the SAT competition 2013. In: Proceedings of SAT Competition, pp. 51–52 (2013)

3. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: GPU-PRISM: an extension of PRISM for general purpose graphics processing units. In: PDMC, pp. 17–19. IEEE Computer Society Press (2010)
4. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_5
5. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
6. Gebhardt, K., Manthey, N.: Parallel variable elimination on CNF formulas. In: Timm, I.J., Thimm, M. (eds.) KI 2013. LNCS (LNAI), vol. 8077, pp. 61–73. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40942-4_6
7. Jin, H., Somenzi, F.: An incremental algorithm to check satisfiability for bounded model checking. ENTCS **119**(2), 51–65 (2005)
8. Kullmann, O.: On a generalization of extended resolution. Discrete Appl. Math. **97**, 149–176 (1999)
9. Osama, M., Gaber, L., Hussein, A.I., Mahmoud, H.: An efficient SAT-based test generation algorithm with GPU accelerator. J. Electron. Test. **34**(5), 511–527 (2018)
10. Osama, M., Wijs, A.: Parallel SAT simplification on GPU architectures. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 21–40. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_2
11. Subbarayan, S., Pradhan, D.K.: NiVER: non-increasing variable elimination resolution for preprocessing SAT instances. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 276–291. Springer, Heidelberg (2005). https://doi.org/10.1007/11527695_22
12. Wijs, A.: GPU accelerated strong and branching bisimilarity checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 368–383. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_29
13. Wijs, A.: BFS-based model checking of linear-time properties with an application on GPUs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 472–493. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_26
14. Wijs, A., Bošnački, D.: Many-core on-the-fly model checking of safety properties using GPUs. Int. J. Softw. Tools Technol. Transfer **18**(2), 169–185 (2016)
15. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: unleashing GPU explicit-state model checking. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 694–701. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_42
16. 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. IEEE (2015)