Parallel SAT Simplification on GPU Architectures

Muhammad Osama(✉) and Anton Wijs(✉)

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

Abstract. The growing scale of applications encoded to Boolean Satisfiability (SAT) problems imposes the need for accelerating SAT simplifications or preprocessing. Parallel SAT preprocessing has been an open challenge for many years. Therefore, we propose novel parallel algorithms for variable and subsumption elimination targeting Graphics Processing Units (GPUs). Benchmarks show that the algorithms achieve an acceleration of 66× over a state-of-the-art SAT simplifier (SatELite). Regarding SAT solving, we have conducted a thorough evaluation, combining both our GPU algorithms and SatELite with MiniSat to solve the simplified problems. In addition, we have studied the impact of the algorithms on the solvability of problems with Lingeling. We conclude that our algorithms have a considerable impact on the solvability of SAT problems.

Keywords: Satisfiability · Variable elimination · Subsumption elimination · Parallel SAT preprocessing · GPU

1 Introduction

Algorithms to solve propositional Boolean Satisfiability (SAT) problems are being used extensively for various applications, such as artificial intelligence, circuit design, automatic test pattern generation, automatic theorem proving, and bounded model checking. Of course, SAT being NP-complete, scalability of these algorithms is an issue. Simplifying SAT problems prior to solving them has proven its effectiveness in modern conflict-driven clause learning (CDCL) SAT solvers [6,9], particularly when applied on real-world applications relevant to software and hardware verification [8,12,17,19]. It tends to produce reasonable reductions in acceptable processing time. Many techniques based on, e.g., variable elimination, clause elimination, and equivalence reasoning are being used to simplify SAT problems, whether prior to the solving phase (preprocessing) [8,10,12,15,16,24] or during the search (inprocessing) [3,18]. However, applying variable and clause

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

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 21–40, 2019.
https://doi.org/10.1007/978-3-030-17462-0_2

elimination iteratively to large problems (in terms of the number of literals) can be a performance bottleneck in the whole SAT solving procedure, or even increase the number of literals, negatively impacting the solving time.

Recently, the authors of [2,11] discussed the current main challenges in parallel SAT solving. One of these challenges concerns the parallelisation of SAT simplification in modern SAT solvers. Massively parallel computing systems such as Graphics Processing Units (GPUs) offer great potential to speed up computations, but to achieve this, it is crucial to engineer new parallel algorithms and data structures from scratch to make optimal use of those architectures. GPU platforms have become attractive for general-purpose computing with the availability of the Compute Unified Device Architecture (CUDA) programming model [20]. CUDA is widely used to accelerate applications that are computationally intensive w.r.t. data processing and memory access. In recent years, for instance, we have applied GPUs to accelerate explicit-state model checking [26,27,30], state space decomposition [28,29] and minimisation [25], metaheuristic SAT solving [31], and SAT-based test generation [22].

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

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

The paper is organised as follows: Sect. 2 introduces the preliminaries. The main GPU challenges for SAT simplification are discussed in Sect. 3, and the proposed algorithms are explained in Sect. 4. Section 5 presents our experimental evaluation. Section 6 discusses related work, and Sect. 7 provides a conclusion and suggests future work.

2 Preliminaries

All SAT formulas in this paper are in conjunctive normal form (CNF). A CNF formula is a conjunction of clauses $\bigwedge_{i=1}^{m} C_i$ where each clause C_i is a disjunction of literals $\bigvee_{j=1}^{k} \ell_j$ and a literal is a Boolean variable x or its complement $\neg x$,

which we refer to as $\bar{x}$. We represent clauses by sets of literals $C = \{\ell_1, \ldots, \ell_k\}$, i.e., $\{\ell_1, \ldots, \ell_k\}$ represents the formula $\ell_1 \vee \ldots \vee \ell_k$, and a SAT formula by a set of clauses $\{C_1, \ldots, C_m\}$, i.e., $\{C_1, \ldots, C_m\}$ represents the formula $C_1 \wedge \ldots \wedge C_m$.

Variable Elimination. Variables can be removed from clauses by either applying the *resolution rule* or *gate-equivalence reasoning*. Concerning the former, we represent application of the resolution rule w.r.t. some variable x using a *resolving operator* $\otimes_x$ on C_1 and C_2. The result of applying the rule is called the *resolvent* [24]. It is defined as $C_1 \otimes_x C_2 = C_1 \cup C_2 \setminus \{x, \bar{x}\}$, and can be applied iff $x \in C_1$, $\bar{x} \in C_2$. The $\otimes_x$ operator can be extended to resolve sets of clauses w.r.t. variable x. For a formula S, let $S_x \subseteq S$, $S_{\bar{x}} \subseteq S$ be the set of all clauses in S containing x and $\bar{x}$, respectively. The new resolvents are defined as $R_x(S) = \{C_1 \otimes_x C_2 \mid C_1 \in S_x \wedge C_2 \in S_{\bar{x}} \wedge \neg\exists y.\{y, \bar{y}\} \subseteq C_1 \otimes_x C_2\}$. The last condition addresses that a resolvent should not be a *tautology*, i.e. a self-satisfied clause in which a variable and its negation exist. The set of non-tautology resolvents can replace S, producing an equivalent SAT formula.

Gate-Equivalence Reasoning. This technique substitutes eliminated variables with deduced logical equivalent expressions. In this work, we focus on the reasoning of AND-OR gates since they are common in SAT-encoded problems and OR gates are likely to be found if AND-equivalence reasoning fails. In general, gate-equivalence reasoning can also be applied using other logical gates. A logical AND gate with k inputs $\ell_1, \ldots, \ell_k$ and output x can be captured by the two implications $x \implies \ell_1 \wedge \ldots \wedge \ell_k$ and $\ell_1 \wedge \ldots \wedge \ell_k \implies x$. In turn, these two implications can be encoded in SAT clauses $\{\{\bar{x}, \ell_1\}, \ldots, \{\bar{x}, \ell_k\}\}$ and $\{\{x, \bar{\ell_1}, \ldots, \bar{\ell_k}\}\}$, respectively. Similarly, the implications of an OR gate $x \implies \ell_1 \vee \ldots \vee \ell_k$ and $\ell_1 \vee \ldots \vee \ell_k \implies x$ are expressed by the SAT clauses $\{\{x, \bar{\ell_1}\}, \ldots, \{x, \bar{\ell_k}\}\}$ and $\{\{\bar{x}, \ell_1, \ldots, \ell_k\}\}$, respectively.

For instance, consider the following formula:

$$S = \{\{x, \bar{a}, \bar{b}\}, \{\bar{x}, a\}, \{\bar{x}, b\}, \{x, c\}, \{\bar{x}, \bar{b}\}, \{y, f\}, \{\bar{y}, d, e\}, \{y, \bar{d}\}, \{y, \bar{e}\}\}$$

The first three clauses in S capture the AND gate (x, a, b) and the last three clauses capture the OR gate (y, d, e). By substituting $a \wedge b$ for x and $d \vee e$ for y in the fourth, fifth, and sixth clauses, a new formula $\{\{a, c\}, \{b, c\}, \{\bar{a}, \bar{b}\}, \{d, e, f\}\}$ can be constructed. Combining AND/OR-gate equivalence reasoning with the resolution rule tends to result in smaller formulas compared to only applying the resolution rule [8,23].

Subsumption Elimination. A clause C_2 is said to *subsume* clause C_1 iff $C_2 \subseteq C_1$. The subsumed clause C_1 is redundant and can be deleted from the original SAT equation. A special form of subsumption is called *self-subsuming resolution*. It is applicable for two 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 $C_2' \subseteq C_1'$. Consider the clauses: $C_1 = \{x, a, b\}$ and $C_2 = \{\bar{x}, b\}$; C_2 self-subsumes C_1 since $x \in C_1$, $\bar{x} \in C_2$ and $\{b\} \subseteq \{a, b\}$. The self-subsuming literal x can be discarded, producing clause $C_1 = \{a, b\}$. In other words, we say that C_1 is *strengthened* by C_2.

3 GPU Challenges: Memory and Data

GPU Architecture. CUDA is a programming model developed by NVIDIA [20] to provide a general-purpose programming paradigm and allow using the massive capabilities of GPU resources to accelerate applications. Regarding the processing hardware, a GPU consists of multiple streaming multiprocessors (SMs) and each SM resembles an array of streaming processors (SPs) where every SP can execute multiple threads grouped together in 32-thread scheduling units called *warps*. On the programming level, a program can launch a *kernel* (GPU global function) to be executed by thousands of threads packed in thread *blocks* of up to 1,024 threads or 32 warps. All threads together form a *grid*. The GPU manages the execution of a launched kernel by evenly distributing the launched blocks to the available SMs through a hardware warp scheduler.

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

- *Global memory* is accessible by all threads with high bandwidth but also high latency. The CPU (host) can access it as an interface to the GPU.
- *Shared memory* is on-chip memory shared by the threads in a block; it is smaller in size and has lower latency than global memory. It can be used to efficiently communicate data between threads in a block.
- *Registers* provide thread-local storage and provide the fastest memory.

To make optimal use of global memory bandwidth and hide its latency, using *coalesced accesses* is one of the best practices in global memory optimisation. 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.

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.

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

To better explain the coherency problem in reallocation, suppose there is an array A allocated and loaded with some data X, then X is visible from both the CPU and GPU memories. When A is reallocated from the host side, the memory is not physically allocated until it is first accessed, particularly when

using an NVIDIA GPU with the Pascal architecture [20]. Once new data Y is written to A from the device side, both sides will observe a combination of X and Y, leading to memory corruptions and page faults. To avoid this problem, A must be reset on the host side directly after memory reallocation to assert the physical allocation. After that, each kernel may store its own data safely in the global memory. In the proposed algorithms, we introduce two types of optimisations addressing memory space and latency.

Regarding memory space optimisation, allocating memory dynamically each time a clause is added is not practical on a GPU while variables are eliminated in parallel. To resolve this, we initially launch a GPU kernel to calculate an upper bound for the number of resolvents to be added before the elimination procedure starts (Sect. 4.1). After this, reallocation is applied to store the new resolvents. Furthermore, a global counter is implemented inside our CNF data structure to keep track of new clauses. This counter is incremented atomically by each thread when adding a clause.

Concerning memory latency optimisation, when thread blocks produce resolvents, these can initially be stored in shared memory. Checking for tautologies can then be done by accessing shared memory, and non-tautologies can be written back to the global memory in a new CNF formula. Also, the definitions of AND-OR gates can be stored in shared memory, to be used later when applying clause substitution (see Sect. 4.1). This has the advantage of reducing the number of global memory accesses. Nevertheless, the size of shared memory in a GPU is very limited (48 KB in most architectures). If the potential size of a resolvent is larger than the amount pre-allocated for a single clause, our BVIPE algorithm automatically switches to the global memory and the resolvent is directly added to the new CNF formula. This mechanism reduces the global memory latency when applicable and deals with the shared memory size limitation dynamically.

Data Structures. The efficiency of state-of-the-art sequential SAT solving and preprocessing is to a large extent due to the meticulously coded data structures. When considering SAT simplification on GPUs, new data structures have to be tailored from scratch. In this work, we need two of them, one for the SAT formula in CNF form (which we refer to as *CNF*) and another for the literal *occurrence table* (*occurTAB*), via which one can efficiently iterate over all clauses containing a particular literal. In CPU implementations, typically, they are created using *heaps* and *auto-resizable vectors*, respectively. However, heaps and vectors are not suitable for GPU parallelisation, since data is inserted, reallocated and sorted dynamically. The best GPU alternative is to create a nested data structure with arrays using unified memory (see Fig. 1). The *CNF* contains a raw pointer (linear array) to store CNF literals and a child structure *Clause* to store clause info. Each clause has a *head pointer* referring to its first literal. The *occurTAB* structure has a raw pointer to store the clause occurrences (array pointers) for each literal in the formula and a child structure *occurList*. The creation of an *occurList* instance is done in parallel per literal using atomic operations. For each clause C, a thread is launched to insert the occurrences of C's literals in the associated *occurLists*. One important remark is that two threads storing the occurrences of different literals do not have to wait for each other. For instance,

occurTAB in Fig. 1 shows two different atomic insertions executed at the same time for literals 2 and –1 (if an integer i represents a literal x, then $-i$ represents $\bar{x}$). This minimises the performance penalty of using atomics.

The advantages of the proposed data structures are: as mentioned above, *occurTAB* instances can be constructed in parallel. Furthermore, coalesced access is guaranteed since pointers are stored consecutively (the gray arrows in Fig. 1), and no explicit memory copying is done (host and device pointers are identical) making it easier to integrate the data structures with any sequential or parallel code.

4 Algorithm Design and Implementation

4.1 Parallel Variable Elimination

In order to eliminate Boolean variables simultaneously in SAT formulas without altering the original satisfiability, a set of variables should be selected for elimination checking that contains only variables that are *independent* of each other. The LCVE algorithm we propose is responsible for electing such a subset from a set of authorised candidates. The remaining variables relying on the elected ones are frozen.

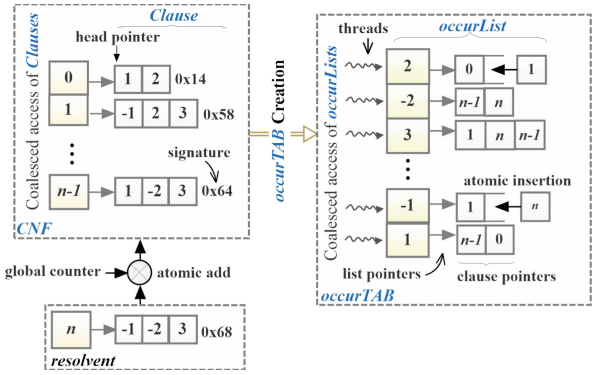

Fig. 1. An example of *CNF* and *occurTAB* data structures.

Algorithm 1: Constructing $\mathcal{A}$ - *GPU*

Input : __global__ $S[\,]$, n, μ
Output: __global__ $\mathcal{A}[\,]$

1 $h \leftarrow$ histogram(S, n);
2 $\mathcal{A} \leftarrow [\,]$; *scores* $\leftarrow [\,]$;
3 $\mathcal{A}$, *scores* $\leftarrow$assignScores(h, $\mathcal{A}$, *scores*);
4 $\mathcal{A} \leftarrow$ prune(sort($\mathcal{A}$, *scores*), h, μ);

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 complement, 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* $\varphi \subseteq \mathcal{A}$ *a set of* elected candidates *iff* $\forall x, y \in \varphi. \neg(x \mathcal{D} y)$

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

Before LCVE is executed, a sorted list of the variables in the CNF formula needs to be created, ordered by the number of occurrences in the formula, in ascending order (following the same rule as in [8]). 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 Algorithms 1 and 2. Algorithm 1 is labelled *GPU*, indicating that each individual step can be launched as a GPU computation. In contrast, algorithms providing kernel code, i.e., that describe the steps each individual GPU thread should perform as part of a GPU computation, such as Algorithm 2, are labelled *GPU kernel*.

Algorithm 2: Assign Scores to SAT formula variables - *GPU kernel*

Input : __global__ $h[\,], \mathcal{A}[\,], scores[\,]$
Output: __global__ $\mathcal{A}[\,], scores[\,]$

```
1 tid ← xThread + blockDim × xBlock; stride ← gridDim × blockDim;
2 while tid < n do
3     x ← tid + 1;
4     A[tid] ← x;
5     if h[x] = 0 ∨ h[x̄] = 0 then
6         scores[x] ← max(h[x], h[x̄])
7     else
8         scores[x] ← h[x] × h[x̄]
9     tid ← tid + stride
```

As input, Algorithm 1 requires a SAT formula S as an instance of *CNF*. Furthermore, it requires the number of variables n occurring in S and a cut-off point μ. At line 1, a histogram array h, providing for each literal the number of occurrences in S, is constructed. This histogram can be constructed on the GPU using the histogram method offered by the *Thrust* library [4,5]. At line 2, memory is allocated for the arrays $\mathcal{A}$ and *scores*. With h, these are input for the kernel *assignScores*. Once the kernel execution has terminated, at line 4, the candidates in $\mathcal{A}$ are sorted on the GPU based on their scores in *scores* and μ is used on the GPU to prune candidates with too many occurrences. We used the radix-sort algorithm as provided in the Thrust library [5].

In the kernel of Algorithm 2, at line 1, the *tid* variable refers to the thread identifier in the launched grid where *xThread* is the thread index inside a block of size *blockDim* and *xBlock* is the block index inside the entire grid of size *gridDim*.

The *stride* variable is used to determine the distance between variables that have to be processed by the same thread. In the subsequent **while** loop (lines 2–9), the thread index is used as a variable index (variable indices start at 1), jumping ahead *stride* variables at the end of each iteration. At lines 5–8, 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 $\bar{x}$, and can be approximated by the formula $h[x] \times h[\bar{x}]$. To avoid score zero in case exactly one of the two literals does not occur in S, we consider that case separately. On average, Algorithm 1 outperforms the sequential counterpart 52×, when considering our benchmark set of problems [21].

LCVE Algorithm. Next, Algorithm 3 is executed on the host, given S, $\mathcal{A}$, h and an instance of *occurTAB* named OT. This algorithm accesses $2 \cdot |\mathcal{A}|$ number of *occurList* instances and parts of S. The use of unified memory significantly improves the rates of the resulting transfers and avoids explicitly copying entire data structures to the host side. The output is φ, implemented as a list. Function **abs** is defined as follows: $\mathbf{abs}(x) = x$ and $\mathbf{abs}(\bar{x}) = x$. The algorithm considers all variables x in $\mathcal{A}$ (line 2). If x has not yet been frozen (line 3), it adds x to φ (line 4). Next, the algorithm needs to identify all variables that depend on x. For this, the algorithm iterates over all clauses containing either x or $\bar{x}$ (line 5), and each literal ℓ in those clauses is compared to x (lines 6–8). If ℓ refers to a different variable v, and v is an authorised candidate, then v must be frozen (line 9).

Algorithm 3: The LCVE algorithm - *CPU*

```
   Input  : S[], A[], h[], OT[]
   Output: φ
1  F ← [];
2  foreach x ∈ A do
3      if ¬F[x] then
4          φ ← φ ++x;
5          foreach C ∈ S[OT[x]] ∪ S[OT[x̄]] do
6              foreach ℓ ∈ C do
7                  v ← abs(ℓ);
8                  if v ≠ x ∧ v ∈ A then
9                      F[v] ← true;
```

BVIPE GPU Algorithm. After φ has been constructed, a kernel is launched to compute an upper bound for the number of resolvents (excluding tautologies) that may be produced by eliminating variables in φ. This kernel accumulates the number of resolvents of each variable using parallel reduction in shared memory within thread blocks. The resulting values (resident in shared memory) of all blocks are added up by atomic operations, resulting in the final output, stored in global memory (denoted by $|\tilde{S}|$). Afterwards, the *CNF* is reallocated according to the extra memory needed. The parallel variable elimination kernel (Algorithm 4) is now ready to be performed on the GPU, considering both the *resolution rule* and *gate-equivalence reasoning* (Sect. 2).

In Algorithm 4, first, each thread selects a variable in φ, based on *tid* (line 4). The *eliminated* array marks the variables that have been eliminated. It is used to distinguish eliminated and non-eliminated variabels when executing Algorithm 6.

Each thread checks the control condition at line 5 to determine whether the number of resolvents ($h[x] \times h[\bar{x}]$) of x will be less than the number of deleted clauses ($h[x]+h[\bar{x}]$). If the condition evaluates to **true**, a list *resolvents* is created in shared memory, which is then added to the simplified formula $\tilde{S}$ in global memory after discarding tautologies (line 8). The *markDeleted* routine marks resolved clauses as deleted. They are actually deleted on the host side, once the algorithm has terminated.

At line 10, definitions of AND and OR gates are deduced by the *gateReasoning* routine, and stored in shared memory in the lists α and β, respectively. If at least one gate definition is found, the *clauseSubstitution* routine substitutes the involved variable with the underlying definition (line 11), creating the resolvents.

In some situations, even if $h[x]$ or $h[\bar{x}]$ is greater than 1, the number of resolvents can be smaller than the deleted clauses, due to the fact that some resolvents may be tautologies that are subsequently discarded. For this reason, we provide a third alternative to lookahead for tautologies in order to conclusively decide whether to resolve a variable if the conditions at lines 5 and 10 both evaluate to **false**. This third option (line 15) has lower priority than gate equivalence reasoning (line 10), since the latter in practice tends to perform more reduction than the former.

Algorithm 4: The BVIPE algorithm - *GPU kernel*

```
   Input  : __global__ S, φ[], h, OT[], eliminated[]
   Input  : __shared__ resolvents[], α[], β[]
   Output: __global__ S̃
1  tid ← xThread + blockDim × xBlock;
2  stride ← gridDim × blockDim;
3  while tid < |φ| do
4      x ← φ[tid], eliminated[x] ← false, numTautologies ← 0;
5      if h[x] = 1 ∨ h[x̄] = 1 then
6          resolvents ← resolve(x, S, OT[x], OT[x̄]);                  /* computes R_x(S) */
7          markDeleted(S, OT[x], OT[x̄]);
8          S̃ ← S ∪ resolvents;
9          eliminated[x] ← true;
10     else if (α, β) ← gateReasoning(x, S, OT[x], OT[x̄]) ≠ (∅, ∅) then
11         resolvents ← clauseSubstitution(x, S, OT[x], OT[x̄], α, β);
12         markDeleted(S, OT[x], OT[x̄]);
13         S̃ ← S ∪ resolvents;
14         eliminated[x] ← true;
15     else
16         numTautologies ← tautologyLookahead(x, S, OT[x], OT[x̄]);
17         numResolvents ← h[x] × h[x̄], numDeleted ← h[x] + h[x̄];
18         if (numResolvents − numTautologies) < numDeleted then
19             resolvents ← resolve(x, S, OT[x], OT[x̄]);
20             markDeleted(S, OT[x], OT[x̄]);
21             S̃ ← S ∪ resolvents;
22             eliminated[x] ← true;
23     tid ← tid + stride;
```

The sequential running time of Algorithm 4 is $\mathcal{O}(k \cdot |\varphi|)$, where k is the maximum length of a resolved clause in S. In practice, k often ranges between 2

and 10. Therefore, the worst case is linear w.r.t. $|\varphi|$. Consequently, the parallel complexity is $\mathcal{O}(|\varphi|/p)$, where p is the number of threads. Since a GPU is capable of launching thousands of threads, that is, $p \approx |\varphi|$, the parallel complexity is an amortised constant $\mathcal{O}(1)$. Our empirical results show an average speedup of 32× compared to the sequential alternative [21].

4.2 Parallel Subsumption Elimination

Algorithm 5 presents our PSE algorithm. Notice that the subsumption check (lines 6–7) has a higher priority than self-subsumption (lines 8–12) because the former often results in deleting more clauses. We apply PSE on the *most constrained variables*, that is, the variables that occur most frequently in S, to maximise parallelism on the GPU. For each literal ℓ, the PSE algorithm is launched on a *two-dimensional* GPU grid, in which threads and blocks have two identifiers. Each thread compares the two clauses in $OT[\ell]$ that are associated with the thread coordinates in the grid. At line 4, those two clauses are obtained. At line 5, it is checked whether subsumption or self-subsumption may be applicable. For this to be the case, the length of one clause needs to be larger than the other. The *sig* routine compares the identifiers of two clauses. The identifier of a clause is computed by hashing its literals to a 64-bit value [8]. It has the property of refuting many non-subsuming clauses, but not all of them since hashing collisions may occur. At line 6, the *isSubset* routine runs a set intersection algorithm in linear time $\mathcal{O}(k)$ assuming that both clauses are sorted. If one clause is indeed a subset of the other, the latter clause is marked for deletion later (line 7).

Algorithm 5: The PSE algorithm - *GPU kernel*

Input: __global__ $S, \ell, OT[\,]$

```
1  tx ← xThread + blockDim × xBlock, ty ← yThread + blockDim × yBlock;
2  stride ← gridDim × blockDim;
3  while ty < |OT[ℓ]| ∧ tx < |OT[ℓ]| do
4      C ← S[OT[ℓ][ty]]; Ć ← S[OT[ℓ][tx]];
5      if |Ć| < |C| ∧ sig(Ć, C) ≠ 0 then
6          if isSubset(Ć, C) then
7              markDeleted(C);
8          else if |C| > 1 then
9              litPos ← isSelfSub(Ć, C);
10             if litPos > −1 then
11                 atomicStrengthen(C, litPos);
12                 markSelfSub(C);
13     ty ← ty + stride, tx ← tx + stride;
```

As an alternative, the applicability of self-subsumption is checked at line 8. If C is not a unit clause, i.e., $|C| > 1$, then self-subsumption can be considered. We exclude unit clauses, because they cannot be reduced in size; instead, they should be propagated over the entire formula before being removed (doing so is planned for future work). If applicable, the routine *isSelfSub* returns the position in C of the literal to be removed, otherwise –1 is returned. This literal is marked for removal in C, using an atomic operation, to avoid race conditions on writing

and reading the literal. Finally, the clause is marked as being altered by self-subsumption (line 12), as after execution of the algorithm, the marked literals need to be removed, and the clause resized accordingly.

Finally, following the concept of thread reuse, each thread jumps ahead *stride* variables in both dimensions at the end of each **while** loop iteration (line 13). Sequential subsumption elimination has time complexity $\mathcal{O}(k \cdot |OT[\ell]|^2)$. By launching enough threads in both dimensions (i.e., $|OT[\ell]|^2 \approx p$), the parallel complexity becomes $\mathcal{O}(k)$. In practice, the average speedup of PSE compared to sequential SE is 9× [21].

Correctly Strenghtening Clauses in PSE. Strengthening self-subsumed clauses cannot be done directly at line 9 of Algorithm 5. Instead, clauses need to be altered in-place. Consider the case that a clause C can be self-subsumed by two different clauses C_1, C_2 on two different self-subsuming literals, and that this is detected by two different threads at the exact same time. We call this phenomenon the parallel-effect of self-subsumption. Notice that the same result can be obtained by applying self-subsumption sequentially in two successive rounds. In the first round, C is checked against C_1, while in the second round, what is left of C is checked against C_2. If C would be removed by t_1 and replaced by a new, strengthened clause, then t_2 may suddenly be accessing unallocated memory. Instead, C is atomically strengthened in-place at line 11; t marks the self-subsumed literal using an atomic Compare-And-Swap operation. At the end, all clauses marked as self-subsumed have to be shrunk in size, in essence, overwriting reset positions. This is performed in a separate kernel call.

Algorithm 6: The PHSE algorithm - *GPU kernel*

```
   Input: __global__ S, φ[ ], eliminated[ ], OT[ ]
   Input: __shared__ sh_C[ ]
1  tid ← xThread + blockDim × xBlock, stride ← gridDim × blockDim;
2  while tid < |φ| do
3      x ← φ[tid];
4      if ¬eliminated[x] then
5          foreach C ∈ S[OT[x]] do
6              sh_C ← C;
7              foreach C′ ∈ S[OT[x̄]] do
8                  if |C′| < |C| ∧ sig(C′, C) ≠ 0 ∧ isSelfSub(C′, sh_C) then
9                      strengthenWT(C, sh_C, x);
10                 else if |C′| = |C| ∧ sig(C′, C) ≠ 0 ∧ isSelfSub(C′, sh_C) then
11                     strengthenWT(C, sh_C, x);
12                     markDeleted(C′);
13             foreach C′′ ∈ S[OT[x]] do
14                 if |C′′| < |C| ∧ sig(C′′, C) ≠ 0 ∧ isSubset(C′′, sh_C) then
15                     markDeleted(C);
16     tid ← tid + stride;
```

Parallel Hybrid Subsumption Elimination (PHSE). PHSE is executed on elected variables that could not be eliminated earlier by variable elimination.

(Self)-subsumption elimination tends to reduce the number of occurrences of these non-eliminated variables as they usually eliminate many literals. After performing PHSE (Algorithm 6), the BVIPE algorithm can be executed again, after which PHSE can be executed, and so on, until no more literals can be removed.

Unlike PSE, the parallelism in the PHSE kernel is achieved on the variable level. In other words, each thread is assigned to a variable when executing PHSE. At line 4, previously eliminated variables are skipped. At line 6, a new clause is loaded, referenced by $OT[x]$, into shared memory (we call it the *shared clause*, *sh_C*).

The shared clause is then compared in the loop at lines 7–12 to all clauses referenced by $OT[\bar{x}]$ to check whether x is a self-subsuming literal. If so, both the original clause C, which resides in the global memory, and *sh_C* must be strengthened (via the *strengthenWT* function). Subsequently, the strengthened *sh_C* is used for subsumption checking in the loop at lines 13–15.

Regarding the complexity of Algorithm 6, the worst-case is that a variable x occurs in all clauses of S. However, in practice, the number of occurrences of x is bounded by the threshold value μ (see Definition 1). The same applies for its complement. Therefore, worst case, a variable and its complement both occur μ times. As PHSE considers all variables in φ and worst case has to traverse each loop μ times, its sequential worst-case complexity is $\mathcal{O}(|\varphi| \cdot \mu^2)$ and its parallel worst-case complexity is $\mathcal{O}(\mu^2)$.

5 Benchmarks

We implemented the proposed algorithms in CUDA C++ using CUDA toolkit v9.2. We conducted experiments using an NVIDIA Titan Xp GPU which has 30 SMs (128 cores each), 12 GB global memory and 48 KB shared memory. The machine equipped with the GPU was running Linux Mint v18. All GPU SAT simplifications were executed in four phases iteratively. Every two phases, the resulting CNF was reallocated to discard removed clauses. In general, the number of iterations can be configured, and the reallocation frequency can be also set to a desired value.

We selected 214 SAT problems from the industrial track of the 2013, 2016 and 2017 SAT competitions [13]. This set consists of almost all problems from those tracks that are more than 1 MB in file size. The largest size of problems occurring in this set is 1.2 GB. These problems have been encoded from 31 different real-world applications that have a whole range of dissimilar logical properties. Before applying any simplifications using the experimental tools, any occurring unit clauses were propagated. The presence of unit clauses immediately leads to simplification of the formula. By only considering formulas without unit clauses, the benchmark results directly indicate the true impact of our preprocessing algorithms.

In the benchmark experiments, besides the implementations of our new GPU algorithms, we involved the SatELite preprocessor [8], and the MiniSat and Lingeling [6] SAT solvers for the solving of problems, and executed these on the compute nodes of the DAS-5 cluster [1]. Each problem was analysed in isolation

on a separate computing node. Each computing node had an Intel Xeon E5-2630 CPU running at a core clock speed of 2.4 GHz with 128 GB of system memory, and runs on the CentOS 7.4 operating system. We performed the equivalent of 6 months and 22 days of uninterrupted processing on a single node to measure how GPU SAT simplification impacts SAT solving in comparison to using sequential simplification or no simplification. The time out for solving experiments and simplification experiments was set to 24 h and 5,000 s, respectively. Time outs are marked 'T' in the tables in this section. Out-of-memory cases are marked 'M' in the tables.

Table 1 gives the MiniSat solving performance summed over the CNF families for both the original and simplified problems produced by GPU SAT simplification and SatELite. The *ve+* and *ve* modes in GPU simplification represent variable elimination with and without PHSE, respectively. The *all* mode involves both *ve+* and PSE. The numbers between brackets of columns 2 to 7 denote the number of solved instances at each family. Bold results in column 6 indicate that the combination of GPU SAT simplification and MiniSat reduced the net solution times (preprocessing + solving), or allowed more problems to be solved compared to the *ve* column of SatElite + MiniSat (column 3). Likewise, the *all* column for our results (column 7) is compared to column 4. The final four columns summarise the number of instances solved faster by MiniSat when using GPU SAT simplification (GPU+M) compared to not using GPU SAT simplification, and compared to using SatElite for simplification (SatElite+M). Numbers in bold indicate that 50% or more of the CNF formulas were solved faster.

The final row accumulates all problems solved faster. The percentage expresses the induced improvement by GPU SAT simplification over MiniSat and SatElite. Similarly, Table 2 gives the performance of solving the problems with Lingeling and GPU SAT simplification+Lingeling (GPU+L). Since SatELite is the groundwork of preprocessing in Lingeling, there is no reason to apply it again.[1]

The presented data shows that GPU+L solved many instances faster than SatElite with Lingeling, especially for the *ak128*, *blockpuzzle*, and *sokoban* problems. In addition, GPU SAT simplification allowed Lingeling to solve more instances of the *hwmcc*, *velev*, and *sncf* models. This effect did not occur for MiniSat.

Table 3 provides an evaluation of the achieved simplifications with GPU SAT simplification and SatELite, where V, C, L are the number of variables, clauses, and literals (in thousands), respectively, and t(s) is the runtime in seconds. In case of GPU SAT simplification, t includes the transfer time between host and device. Bold results indicate significant improvements in reductions on literals for more than 50% of the instances.

On average, SatElite managed to eliminate more variables and clauses at the expense of literals explosion. Regarding scalability, GPU SAT simplification is able to preprocess extremely large problems (e.g., the *esawn* problem, *sokoban-p20.sas.ex.23*, and the high-depth *sncf* model) in only a few seconds while SatElite failed. For our benchmark set of SAT problems, GPU SAT

[1] The full tables with all obtained results are available at [21].

Table 1. MiniSat solving of original and simplified formulas (time in seconds).

Family (#CNF)	Minisat	SatElite + Minisat		GPU + Minisat			GPU+M vs MiniSat		GPU+M vs SatElite+M	
		ve	all	ve	ve+	all	ve+	all	ve+	all
dspam(8)	726(8)	143(8)	86(8)	194(8)	**127(8)**	**51(8)**	**6**	3	**6**	**7**
ACG(4)	10146(4)	5376(4)	2813(4)	9597(4)	4139(4)	3558(4)	**4**	**4**	**3**	1
ak128(30)	17353(11)	19604(11)	52185(11)	54302(11)	**34001(12)**	**5045(12)**	**12**	**7**	**12**	**12**
hwmcc(27)	291885(10)	294646(12)	408184(14)	205099(12)	145182(11)	262539(12)	**11**	**11**	**8**	**7**
UCG(3)	3048(3)	1467(3)	1022(3)	2186(3)	1831(3)	**725(3)**	**2**	**3**	1	**3**
UR(3)	12648(3)	12233(3)	4221(3)	27547(3)	**6431(3)**	20240(3)	**3**	**1**	**2**	0
UTI(1)	4594(1)	916(1)	1908(1)	2350(1)	4192(1)	**1244(1)**	**1**	**1**	0	**1**
itox(6)	7(6)	78(6)	151(6)	4.2(6)	**5.7(6)**	**12(6)**	**3**	0	**6**	**6**
manol-pipe(10)	1978(10)	3909(10)	3106(10)	1432(10)	**1239(10)**	**1213(10)**	**7**	**7**	**5**	**5**
blockpuzzle(14)	43584(13)	177545(13)	99569(10)	36698(14)	**55250(14)**	**55317(14)**	**7**	5	**12**	**13**
26-stack-cas(1)	0.9(1)	110(1)	110(1)	0.67(1)	**0.8(1)**	**2.81(1)**	**1**	0	**1**	**1**
9dlx-vliw(10)	14093(3)	141993(5)	133859(5)	76716(4)	33306(3)	**112862(5)**	1	**4**	0	**4**
dated(1)	3711(1)	4699(1)	3711(1)	3894(1)	**4553(1)**	3833(1)	0	0	**1**	0
podwr001(1)	14.5(1)	22(1)	53(1)	15.7(1)	**18(1)**	23(1)	0	0	**1**	**1**
transport(3)	2935(3)	1967(3)	2804(3)	5109(3)	2366(3)	7649(3)	1	1	0	1
valves(1)	3120(1)	770(1)	702(1)	1202(1)	1237(1)	873(1)	**1**	**1**	0	0
mizh-md5(6)	7492(6)	3272(6)	5091(6)	7519(6)	4457(6)	5805(6)	4	**5**	2	**3**
synthesis-aes(6)	15868(5)	10443(6)	54348(6)	7271(6)	**7645(6)**	8797(6)	**5**	**5**	**3**	**5**
test-c1-s(6)	32277(5)	86955(6)	23596(5)	23044(5)	28716(5)	81727(5)	**4**	**5**	1	2
cube-11-h13(1)	625(1)	324(1)	1173(1)	319(1)	**318(1)**	**122(1)**	**1**	**1**	**1**	**1**
esawn-uw3(1)	224(1)	M	M	426(1)	**563(1)**	**610(1)**	0	0	**1**	**1**
ibm-2002(1)	385(1)	66(1)	169(1)	61(1)	195(1)	**73(1)**	**1**	**1**	0	**1**
sin2(2)	38542(2)	44691(2)	39246(2)	45981(2)	54442(2)	**38016(2)**	**1**	**1**	**1**	**2**
traffic(1)	134(1)	133(1)	206(1)	133(1)	**41(1)**	282(1)	**1**	0	**1**	0
partial(8)	1696(1)	348(1)	605(1)	7062(2)	735(1)	T	**1**	0	0	0
newton(4)	24926(4)	6152(4)	5464(4)	15477(4)	19645(4)	13783(4)	**4**	**4**	**2**	**2**
safe(4)	5.5(4)	27(4)	102(4)	4.5(4)	**4.2(4)**	**20(4)**	**2**	0	**4**	**4**
arcfour(8)	1531(3)	831(3)	2196	1010(3)	1046(3)	**1051(3)**	**3**	**3**	0	**1**
sokoban(33)	127523(15)	116562(19)	91555(11)	119958(16)	101166(17)	**147564(18)**	**11**	**11**	8	**13**
#CNF solved faster (Percentage %)							98/135(73)	84/139(61)	82/140(59)	97/142(68)

Table 2. Lingeling solving of original and simplified formulas (time in seconds).

Family (#CNF)	Lingeling	GPU + Lingeling			GPU+L vs Lingeling	
		ve	ve+	all	ve+	all
dspam(8)	35(8)	52(8)	79(8)	65(8)	1	0
ACG(4)	1426(4)	1685(4)	1457(4)	**1392(4)**	**3**	**3**
ak128(30)	35692(16)	2602(16)	**41275(17)**	**2649(16)**	**15**	**8**
hwmcc15(27)	115970(27)	95530(27)	**96329(27)**	**109925(27)**	**18**	**16**
UCG(3)	605(3)	679(3)	762(3)	691(3)	0	1
UR(3)	2029(3)	1199(3)	2709(3)	**1739(3)**	1	**2**
UTI(1)	469(1)	412(1)	**402(1)**	**362(1)**	**1**	**1**
itox(6)	195(6)	153(6)	**92(6)**	**106(6)**	**3**	**3**
manol-pipe(10)	631(10)	662(10)	713(10)	659(10)	1	2
blockpuzzle(14)	30391(14)	57559(14)	57895(14)	**15328(14)**	**7**	**7**
26-stack-cas(1)	7(1)	1(1)	**1(1)**	**3(1)**	**1**	**1**
9dlx-vliw(10)	4287(10)	4713(10)	4579(10)	5121(10)	3	4
dated(1)	800(1)	704(1)	**751(1)**	**657(1)**	**1**	**1**
podwr001(1)	111(1)	108(1)	171(1)	**93(1)**	0	**1**
transport(3)	2526(3)	2246(3)	**1012(3)**	**2025(3)**	**2**	1
valves(1)	1536(1)	1501(1)	**1288(1)**	**1455(1)**	**1**	**1**
velev(2)	11717(2)	10645(2)	**10695(2)**	**9381(2)**	**2**	**2**
mizh-md5(6)	34104(6)	33327(6)	**32589(6)**	**23096(6)**	**3**	**5**
synthesis-aes(6)	49679(6)	16494(6)	**31927(6)**	**16151(6)**	**3**	**6**
test-c1-s(6)	84807(6)	55512(6)	17284(5)	4450(4)	**3**	2
cube-11-h13(1)	1495(1)	2022(1)	**1217(1)**	**813(1)**	**1**	**1**
esawn-uw3(1)	107(1)	271(1)	239(1)	463(1)	0	0
ibm-2002(1)	150(1)	191(1)	**129(1)**	194(1)	**1**	0
sin2(2)	4(1)	7(1)	4.2(1)	20(1)	0	0
traffic(1)	355(1)	694(1)	505(1)	869(1)	0	0
partial(8)	4691(2)	16026(2)	**1276(2)**	6735(2)	**2**	**1**
newton(4)	825(4)	913(4)	908(4)	861(4)	**2**	**2**
safe(4)	49(4)	57(4)	51(4)	63(4)	1	0
sncf-model(8)	73690(4)	135683(5)	T	**131301(5)**	0	**4**
arcfour(8)	1293(3)	1338(3)	1319(3)	1451(3)	0	1
sokoban(33)	339236(26)	139509(23)	248439(24)	166542(24)	**14**	**17**
#CNF solved faster (Percentage %)					90/179(50)	93/179(52)

simplification methods *ve+* and *all* achieve on average a speedup of 66× and 12× compared to SatElite *ve* and *all*, respectively [21].

Our appendix [21] presents larger tables, providing all our results related to Tables 1, 2 and 3. In addition, it presents a comparison between our GPU simplification algorithms and directly ported to the CPU versions of the algorithms, to provide more insight into the contribution of the GPU parallelisation. On average, subsumption elimination is performed 9× faster on the GPU, hybrid

Table 3. Comprehensive evaluation of simplification impact on original CNF size

CNF (Original)				SatElite								GPU							
Name	V	C	L	ve				all				ve+				all			
				V	C	L	t(s)	V	C	L	t(s)	V	C	L	t (s)	V	C	L	t(s)
dspam_dump_vc1103	275	920	2464	161	722	2190	12.9	155	695	2092	15.53	164	729	**2182**	**0.69**	164	729	2178	**3.52**
ACG-20-10p1	325	1316	3140	272	1293	3544	10.4	250	1194	3154	20.95	287	**1243**	**3135**	**0.32**	287	1242	**3134**	**5.94**
ak128astepbg2asisc	260	844	2144	130	681	2176	4.95	98	552	1853	11.32	162	**617**	**1722**	**0.32**	162	617	**1722**	**1.12**
hwmcc15deep-intel032	426	882	2051	69	334	1235	88.1	45	226	856	114.8	104	391	**1171**	**0.47**	104	390	1164	**1.51**
UCG-15-10p1	151	757	1845	119	739	2132	5.78	102	666	1849	13.51	122	**700**	**1838**	**0.23**	122	700	**1837**	**3.50**
UR-20-5p1	178	938	2320	140	918	2661	6.83	121	833	2332	15.97	145	**874**	**2317**	**0.32**	145	874	**2317**	**9.37**
UTI-20-10p1	203	1075	2656	160	1051	3042	7.74	138	954	2666	18.17	166	**1001**	**2651**	**0.29**	166	1001	**2650**	**11.8**
itox_vc1033	111	339	898	35	176	639	12.2	28	132	434	19.92	42	195	**629**	**0.56**	42	194	620	**1.42**
manol-pipe-f10ni	368	1100	2567	56	458	1855	15.2	56	456	1844	17.94	102	508	**1590**	**0.75**	102	508	**1589**	**5.69**
blockpuzzle_8x8_s1_f4	3	311	625	3	305	987	0.91	3	303	794	3.42	3	311	**632**	**0.08**	3	311	**632**	6.83
blockpuzzle_9x9_s1_f7	5	692	1389	5	681	2179	1.84	5	678	1764	8.53	5	691	**1414**	**0.13**	5	691	**1414**	**6.18**
26_stack_cas_longest	1505	3799	11287	1	11	44	110	1	10	39	110.4	3	15	**43**	**0.84**	3	14	41	**2.80**
9dlx_vliw_at_b_iq8	371	7170	20872	303	7042	22708	136	303	6994	21690	607.8	335	7098	**21071**	**5.80**	333	7069	**20408**	**48.8**
dated-10-17-u	177	813	1914	134	740	2074	8.49	67	425	1112	15.96	**127**	**678**	**1764**	**0.43**	127	678	1764	**4.22**
valves-gates-1-k617	970	3062	7842	302	1777	6027	30.9	249	1368	4454	52.40	371	**1680**	**4970**	**1.21**	366	1655	4856	**18.5**
velev-pipe-oun-1.1-05	236	7676	22782	143	7491	23933	45.0	143	7488	23914	84.85	178	7561	**22683**	**11.5**	178	7557	**22530**	109
velev-vliw-uns-2.0-uq5	151	2465	7141	120	2407	7653	41.2	120	2391	7388	99.17	131	2424	**7181**	**1.55**	130	2411	**6963**	**20.3**
esawn_uw3.debugged	12200	48049	133127	n/a	n/a	n/a	M	n/a	n/a	n/a	M	**7361**	**36184**	**108331**	**16.0**	**7385**	**36189**	**108187**	**333**
mizh-md5-12	69	226	585	28	171	576	1.53	28	163	552	2.17	37	**163**	**482**	**0.07**	37	163	**482**	**0.32**
mizh-md5-48-5	61	238	689	34	193	641	1.36	26	157	533	2.31	**26**	**167**	**563**	**0.12**	26	167	563	**2.05**
slp-synthesis-aes-top30	97	304	745	37	211	1110	41.9	37	191	1043	90.04	42	215	**658**	**0.07**	42	196	**599**	**1.71**
ibm-2002-23r-k90	209	864	2176	80	601	2179	9.93	72	523	1796	17.74	107	652	**1899**	**0.38**	107	645	1834	**5.26**
newton.2.3.i.smt2-cvc4	302	1309	3625	193	1116	3669	19.5	183	1025	3320	29.79	215	**1079**	**3193**	**0.77**	215	1074	**3160**	**7.12**
partial-10-19-s	269	1282	3033	207	1175	3293	13.0	168	995	2750	25.59	**198**	**1089**	**2845**	**0.56**	198	1089	2845	**7.64**
safe029	62	305	1048	41	261	1204	7.92	40	251	1095	23.66	56	291	**1025**	**0.27**	56	290	**932**	**3.01**
sin2.c.20.smt2-cvc4	1962	7954	21638	1063	6684	22579	136	867	5647	19591	306	1196	**6072**	**18167**	**3.24**	1196	6056	**18057**	**35.2**
sncf_model_depth_07	814	2641	6565	n/a	n/a	n/a	T	n/a	n/a	n/a	T	**423**	**1778**	**5089**	**1.97**	**424**	**1778**	**5084**	**7.82**
test_s18160	411	1831	5104	266	1595	5170	23.0	266	1595	5102	30.2	306	1607	**4770**	**0.75**	306	1599	**4731**	**10.6**
traffic_3_uc_sat	142	1312	4558	141	1311	4556	3.15	141	1311	4556	3.22	141	1311	4557	**1.06**	141	**1310**	**4554**	28.5
arcfour_6_14	105	404	1172	60	316	1365	4.99	60	316	1364	6.13	72	341	**1114**	**0.09**	72	341	**1114**	**1.21**
sokoban-p17.sas.ex.11	606	876	2312	6	151	1023	49.1	6	134	718	179	34	283	1180	**0.53**	34	282	1026	**2.44**
sokoban-p20.sas.ex.23	3632	7258	48119	111	2929	40787	11.0	n/a	n/a	n/a	T	**276**	**3716**	**41536**	**0.34**	**276**	**3714**	**38205**	**5.96**

subsumption elimination is performed 15× faster on the GPU, and BVE is conducted 32× faster on the GPU.

6 Related Work

Subbarayan and Pradhan [24] provided the first Bounded Variable Elimination (BVE) technique, called NiVER, based on the resolution rule of Davis-Putnam-Logemann-Loveland (DPLL) [7]. Eén and Biere [8] extended NiVER with subsumption elimination and clause substitution. However, the subsumption check is only performed on the clauses resulting from variable elimination, hence no reductions are obtained if there are no variables to resolve.

Heule *et al.* [14,16] introduced several approaches for clause elimination that can be effective in SAT simplifications, such as Blocked Clause Elimination. Bao *et al.* [3], on the other hand, have presented an efficient implementation of Common Sub-clause Elimination (CSE) performed periodically during SAT solving. CSE trims the search space, thereby decreasing the solving time, and is stable, i.e., the outcome of CSE does not depend on the chosen clause ordering. Gebhardt and Manthey [10] presented the first attempt to parallelise SAT preprocessing on a multi-core CPU using a locking scheme to prevent threads corrupting the SAT formula. However, they reported only a very limited speedup of on average 1.88× when running on eight cores.

All the above methods apply sound simplifications, but none are tailored for GPU parallelisation. They may consume considerable time when processing large problems.

Finally, it should be noted that BVE, as introduced in [8,10,16,24], is not confluent, as noted by the authors of [16]. Due to dependency between variables, altering the elimination order of these variables may result in different simplified formulas. This drawback is circumvented by our LCVE algorithm, which makes it possible to perform parallel variable elimination while achieving the confluence property.

7 Conclusion

We have shown that SAT simplifications can be performed efficiently on many-core systems, producing impactful reductions in a fraction of a second, even for larger problems consisting of millions of variables and tens of millions of clauses. The proposed BVIPE algorithm provides the first methodology to eliminate multiple variables in parallel while preserving satisfiability. Finally, PSE and PHSE have proven their effectiveness in removing many clauses and literals in a reasonable amount of time.

Concerning future work, the results of this work motivate us to take the capabilities of GPU SAT simplification further by supporting more simplification techniques or balancing the workload on multiple GPUs.

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. Balyo, T., Sinz, C.: Parallel satisfiability. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 3–29. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-63516-3_1
3. Bao, F.S., Gutierrez, C., Charles-Blount, J.J., Yan, Y., Zhang, Y.: Accelerating boolean satisfiability (SAT) solving by common subclause elimination. Artif. Intell. Rev. **49**(3), 439–453 (2018)
4. Bell, N., Hoberock, J.: Thrust: a productivity-oriented library for CUDA. In: GPU computing gems Jade edition, pp. 359–371. Elsevier, Atlanta (2012)
5. Bell, N., Hoberock, J.: A parallel algorithms library. Thrust Github (2018). https://thrust.github.io/
6. Biere, A.: Lingeling, plingeling and treengeling entering the SAT competition 2013. In: Proceedings of SAT Competition, pp. 51–52 (2013)
7. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
8. 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
9. 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
10. 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
11. Hamadi, Y., Wintersteiger, C.: Seven challenges in parallel SAT solving. AI Mag. **34**(2), 99 (2013)
12. Han, H., Somenzi, F.: Alembic: an efficient algorithm for CNF preprocessing. In: Proceedings of 44th ACM/IEEE Design Automation Conference, pp. 582–587. IEEE (2007)
13. Heule, M., Järvisalo, M., Balyo, T., Balint, A., Belov, A.: SAT Competition, vol. 13, pp. 16–17 (2018). https://satcompetition.org/
14. Heule, M., Järvisalo, M., Biere, A.: Clause elimination procedures for CNF formulas. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR 2010. LNCS, vol. 6397, pp. 357–371. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16242-8_26
15. Heule, M., Järvisalo, M., Lonsing, F., Seidl, M., Biere, A.: Clause elimination for SAT and QSAT. J. Artif. Intell. Res. **53**, 127–168 (2015)
16. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 129–144. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_10
17. Järvisalo, M., Biere, A., Heule, M.J.: Simulating circuit-level simplifications on CNF. J. Autom. Reasoning **49**(4), 583–619 (2012)
18. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 355–370. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_28
19. Jin, H., Somenzi, F.: An incremental algorithm to check satisfiability for bounded model checking. ENTCS **119**(2), 51–65 (2005)

20. NVIDIA: CUDA C Programming Guide (2018). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
21. Osama, M., Wijs, A.: Parallel SAT Simplification on GPU Architectures -Appendix (2019). http://www.win.tue.nl/~awijs/suppls/pss-app.pdf
22. 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)
23. Ostrowski, R., Grégoire, É., Mazure, B., Saïs, L.: Recovering and exploiting structural knowledge from CNF formulas. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 185–199. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46135-3_13
24. 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
25. 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
26. 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
27. 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)
28. Wijs, A., Katoen, J.-P., Bošnački, D.: GPU-based graph decomposition into strongly connected and maximal end components. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 310–326. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_20
29. Wijs, A., Katoen, J.P., Bošnački, D.: Efficient GPU algorithms for parallel decomposition of graphs into strongly connected and maximal end components. Formal Methods Syst. Des. **48**(3), 274–300 (2016)
30. 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
31. Youness, H., Ibraheim, A., Moness, M., Osama, M.: An efficient implementation of ant colony optimization on GPU for the satisfiability problem. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 230–235, March 2015