Leveraging GPUs for Effective Clause Sharing in Parallel SAT Solving[⋆]

Nicolas Prevot[1], Mate Soos[2], and Kuldeep S. Meel[2]

[1] Independent Researcher
[2] School of Computing, National University of Singapore

Abstract. The past two decades have witnessed an unprecedented improvement in runtime performance of SAT solvers owing to clever software engineering and creative design of data structures. Yet, most entries in the annual SAT competition retain the core architecture of MiniSat, which was designed from the perspective of single core CPU architectures. Since 2005, however, there has been a significant shift to heterogeneous architectures owing to the impending end of Dennard scaling.

The primary contribution of this work is a novel multi-threaded CDCL-based framework, called GPUShareSat, designed to take advantage of CPU+GPU architectures. The core underlying principle of our approach is to divide the tasks among the CPU and the GPU so as to attempt to achieve the best of both worlds. We observe that bit-vector based operations can allow a GPU to efficiently determine the usefulness of a learnt clause to different threads and accordingly notify the thread of the presence of relevant clauses. This approach of checking all clauses against all assignments from different threads allows the GPU to exploit its potential for massive parallelism through clever group-testing strategy and bitwise operations.

Our detailed empirical analysis shows practical efficiency of our approach: in particular, GPUShareSat augmented with the state-of-the-art single-threaded solver Relaxed_LCMDCBDL_newTech solved 19 more instances than the winner of the 2020 SAT competition's parallel track, P-MCOMSPS-STR.

1 Introduction

Given a Boolean formula φ over the set of variables $\boldsymbol{v}$, the problem of Boolean Satisfiability (SAT) is to determine whether there exists an assignment $\mathcal{A}$ such that φ evaluates to True under $\mathcal{A}$. SAT is a fundamental problem in computer science with applications in various domains ranging from computational biology, automated theorem proving, spectrum allocations, and the like. The past 25 years have witnessed the development of efficient SAT solvers allowing modern solvers to handle problems involving millions of variables. Such an unprecedented improvement in the runtime performance of SAT solvers is often dubbed as the

[⋆] The accompany open source library is available at https://github.com/nicolasprevot/GpuShareSat

SAT revolution. Quoting Knuth, "The story of satisfiability is the tale of a triumph of software engineering, blended with rich doses of beautiful mathematics" [16].

From the perspective of software engineering and the creative design of data structures, Een and Sorrenson's MiniSat [13], first introduced in 2005, taking concepts from both Chaff [17] and GRASP, has remained a cornerstone for the design of modern SAT solvers. Even fifteen years later, most of the entries in the annual SAT competition continue to retain the architecture proposed by MiniSat. It is worth noting that the design of data structures of MiniSat was primarily targeted for single-core CPUs. Since 2006, however, the end of Dennard scaling [10] has prompted the leading hardware designers to explore multi-core and heterogeneous architectures. The SAT community acknowledged the importance of multi-core architectures since the early days, and there has been a consistent interest in the design of parallel SAT solvers.

Returning to the hardware landscape, among a wide array of hardware architectures proposed over the years, one of the dominant architectures is CPU+GPU, wherein the CPU consist of a small number of *fat* cores while GPUs consist of a large number of *thin* cores, and the general computational paradigm is to perform complex computational tasks on CPUs while employing GPUs to perform embarrassingly parallel computational tasks, often involving matrix-based arithmetic operations. The crucial importance of CPU+GPU to unprecedented advances in machine learning [9] serves as a strong motivator for the design of frameworks in other domains to take advantage of heterogeneous architectures. While there have been recent efforts in the context of SAT to take advantage of GPUs as well as the CPUs, such efforts have not materialized in achieving runtime performance improvements. To summarize, a major challenge in the community is the *design of an efficient framework for heterogeneous architectures.*

The primary contribution of this work is a novel CDCL-based framework called GPUShareSat, designed to take advantage of CPU+GPU heterogeneous architectures. Our aim is to divide the tasks in CDCL to CPU and GPU so as to achieve the best of both worlds. To this end, we focus on the major Achilles heel of parallel SAT solvers: identifying which clauses to import from other threads. We observe that efficient bit-vector-based operations can allow a GPU to efficiently determine the usefulness of a learnt clause to different threads and accordingly notify the thread of the presence of these relevant clauses. The identification of relevance of a clause for a thread is based on the assumption of the locality of assignments. Furthermore, based on the observation that a clause is often not useful to most threads, we design a group testing-based strategy to perform efficient checks of usefulness.

To demonstrate the practical efficiency of our framework, we augment two state of the art SAT solvers, one multi-threaded, glucose-syrup [3], and one single-threaded, Relaxed_LCMDCBDL_newTech [1], with GPUShareSat and perform detailed analysis on benchmarks from the 2020 SAT competition on a high-performance single-GPU multi-core CPU computing cluster. Our empirical analysis demonstrates that glucose-syrup, when augmented with GPUShareSat, solves 11 more instances than glucose-syrup alone. Similarly, Relaxed_LCMDCBDL_newTech, augmented with GPUShareSat, solves 19 instances more than P-MCOMSPS-STR, the winner of the parallel track of the 2020 SAT competition.

To encourage adoption, we sought to perform minimal changes to the MiniSat-based architecture of glucose-syrup and Relaxed_LCMDCBDL_newTech. To this end, we achieved adding GPUShareSat to the respective solvers by keeping largely intact the source code running CDCL on CPU threads, modifying only the code responsible for sharing clauses.

2 Definitions

We will use lowercase to represent variables and boldface to represent (possibly, multi-dimensional) sets/vectors. For a vector $\boldsymbol{x}$, we use x_i to represent the i-th coordinate of $\boldsymbol{x}$. Let $\boldsymbol{v} = \{v_1, v_2, \dots v_n\}$ be the set of Boolean variables. A literal ℓ is a variable v or its negation $\neg v$. A clause C of size s is a disjunction of s literals, i.e., $C = (\ell_1 \vee \ell_2 \dots \ell_k)$. A formula φ in Conjunctive Normal Form (CNF) is represented as conjunction of finitely many clauses.

We define a truth value as a member of the set $\{T, F, U\}$ where T stands for True, F for False, and U for unassigned. The negation of a truth value w is denoted as $\neg w$, with $\neg T = F$, $\neg F = T$, $\neg U = U$. An assignment $\mathcal{A}$ is a function that maps a variable to a truth value. Given an assignment $\mathcal{A}$ and a literal ℓ, $\mathcal{A}(l) = \mathcal{A}(v)$ if $l = v$ and $\mathcal{A}(l) = \neg\mathcal{A}(v)$ if $l = \neg v$. An assignment $\mathcal{A}$ is complete when each variable is mapped to either T or F. We say that $\mathcal{A}$ satisfies φ if $\mathcal{A}$ is complete and φ evaluates to T under $\mathcal{A}$.

In rest of the text, we assume the reader is familiar with the standard terminology such as *conflict, propagation* related to Conflict Driven Clause Learning (CDCL) paradigm; the interested reader is referred to [19] for details.

Bitwise representation of assignments A vector $\boldsymbol{x}$ of truth values of size k can be represented using two size k bit-vectors $(\mathsf{Se}, \mathsf{Tr})$ wherein x_i is represented by $(\mathsf{Se}[i], \mathsf{Tr}[i])$. $\mathsf{Se}[i]$ is set if $x_i \neq U$. $\mathsf{Tr}[i]$ is set if $x_i = T$, not set if $x_i = F$, and may be set or not if $x_i = U$

We now define three functions, isFalse, isUndef, and isTrue, which are employed in our algorithmic descriptions. For a given vector $\boldsymbol{x}$, $\mathsf{isFalse}(\boldsymbol{x})$ returns a bit-vector of size k whose i-th bit is set to 1 if $x_i = F$. Observe that $\mathsf{isFalse}(\boldsymbol{x})$ can be obtained from bitwise operations over Se and Tr, i.e., $\mathsf{isFalse}(\boldsymbol{x}) = \mathsf{Se} \& \sim \mathsf{Tr}$ wherein & represents the bitwise AND, which operates over k bits, and $\sim$ represents the bitwise negation. Similarly, $\mathsf{isUndef}(\boldsymbol{x})$ returns a bit-vector such that $\mathsf{isUndef}(\boldsymbol{x}) = \sim \mathsf{Se}$. Finally, $\mathsf{isTrue}(x)$ is a size k bit-vector which can be computed as $\mathsf{Se}\&\mathsf{Tr}$.

3 Related work

Our work touches on two separate topics within the area of SAT solving: (1) multi-threaded SAT solving and (2) using GPGPUs for improving the speed of SAT solving.

Multithreaded SAT solving There are two main categories of parallel SAT solving strategies: one called divide-and-conquer [28] that divides the problem into always non-overlapping, and hopefully equal parts, and one called portfolio [14] that does not attempt to divide the problem, and instead relies on the cooperation of the different solver threads to attack the problem from different angles thanks to their differing configurations. This latter is the approach we take in this paper.

The original divide-and-conquer method by Zhang et al. [28] relied on so-called guiding paths to cut the problem into smaller chunks. A more modern and performant version of this approach by van der Tak et al. is called concurrent cube-and-conquer [27] that uses a lookahead solver to cut the problem into many smaller chunks that are expected to be of equivalent complexity.

ManySAT [14] pioneered the so-called portfolio approach that uses a CDCL system configured differently for each thread, each sharing some information with the other. The different configurations used by the threads are such as using different restart strategies [5][25] or different variable polarity policies [24]. The different threads in ManySAT share certain clauses with each other using lockless queues: if the CDCL algorithm's learnt clause is less than eight long, the clause is shared with other threads. Modern portfolio method SAT solvers use a number of heuristics to decide which clauses to share. In glucose-syrup [3] and Plingeling [6], clauses are shared only if their size and LBD (Literal Block Distance) is smaller than a constant. Finally, Vallade et al. [26] improved on these results by using a metric based on both community structure [7] and LBDs in their winning parallel SAT solver of the 2020 SAT competition, P-MCOMSPS-STR.

GPU aided SAT solving The state-of-the-art SAT solving approach of CDCL SAT solvers does not translate well from the CPU to the GPGPU domain. One difficulty is that although the GPU can run many times more threads at once than the CPU, the memory and cache available per thread is much smaller on the GPU, and a group of threads not following the same decision path (i.e., diverging threads) cause all other threads in the same so-called 'warp' to stall. Running CDCL using the well-known watched literal scheme requires both a large amount of memory and a highly non-uniform decision path for each thread. This would make running the CDCL procedure separately in each GPU thread too slow.

Various approaches have been proposed to take advantage of GPUs to speed up SAT solving but have not yet seen widespread adoption. In [23], the GPU is used to perform unit propagation. In [21,22], the GPU is used for pre-processing the formula with techniques such as Bounded Variable Elimination and subsumption [12]. In [18], the GPU is used to perform survey propagation [8]. Finally, in [4] the authors use the GPU to execute a version of the Tabu Search algorithm [20] while the CPU is executing a multi-threaded CDCL algorithm.

4 GPUShareSat: GPU-based Parallel SAT Solving

We now present the primary technical contribution of this paper, GPUShareSat, a GPU-based framework for parallel SAT solving. GPUShareSat bears similarity to the traditional parallel SAT solvers in its reliance on several CPU threads, wherein each thread runs its own CDCL algorithm. GPUShareSat differs crucially from its contemporaries in its usage of a dedicated CPU thread, henceforth referred to as `MasterThread`, to which all CPU threads export and import learnt clauses. `MasterThread`, in turn, relies on the adjoining GPU to inform the threads on the clauses that they should import. We present the high-level overview of GPUShareSat in Figure 1.

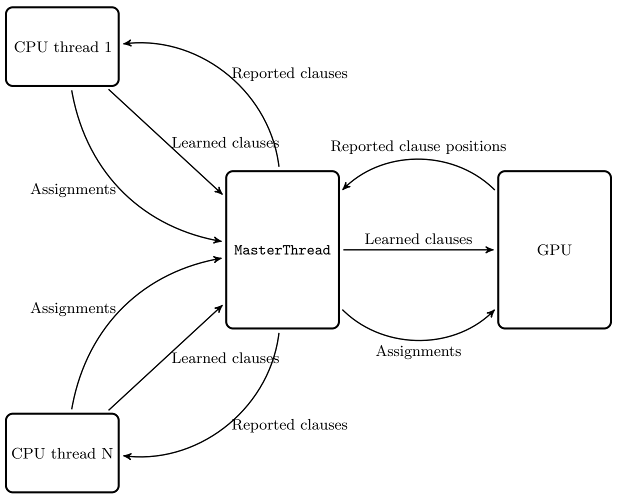

Fig. 1. Interactions between CPU threads and with the GPU

In our design of GPUShareSat, we sought to minimize modification to the overall CDCL architecture so as to ease adoption to other solvers. To this end, we only modified the subroutine for importing clauses in the context of CPU clauses wherein instead of only importing clauses at toplevel (i.e. decision level 0), we enable the solver to import clauses at the highest decision level possible while keeping the watched literal scheme consistent. Another significant modification lies in the export of the current assignment trails to `MasterThread` since the `MasterThread` decides on exporting clauses to CPU threads based on their recent assignments. We present the simplified pseudocode for the CPU thread's solver in Algorithm 1 in Appendix. The pseudocode follow the standard CPU solver along with three additions: (1) The CPU thread seeks to eagerly import the clauses from GPU (line 2), (2) the CPU thread sends the learnt clause to GPU (line 7), (3) the CPU thread send the current assignment to the GPU thread (line 13).

4.1 Usefulness of Clauses

From the perspective of a CPU thread, a natural desiderata would be to import clauses that will be useful in its search in the future. To this end, we hypothesize

Algorithm 1 CPUSolver

```
1:  while noOtherThreadHasFoundAnAnswer() do
2:      importFromGPU()
3:      if propagate() == conflict then
4:          c ← ConflictAnalysis()
5:          if level == 0 then
6:              return UNSAT
7:          SendtoGPU(c)
8:          addClauseToDatabase(c)
9:          backtrack
10:     else
11:         if allVariablesAreSet() then
12:             return SAT
13:         sendCurrentAssignToGpu()
14:         decide()
```

that clauses that would have been useful recently are likely to be useful in the future since the search space of CDCL tends to be local. To this end, `MasterThread` needs to decide how to determine whether a clause cl would be useful to a given thread, say `Thread`. Consider an assignment of a thread where unit propagation completed without conflict, and a clause cl which this thread does not have. If all literals of this clause are false except for one which is undef, then this clause would have implied this undef literal. So it would have been used in unit propagation. If all literals of this clause are false, then this clause would have been in conflict, so it would have been useful as well.

We formalize the above observation via the notion of *triggering*: let $\mathcal{A}$ be a partial assignment. We say that cl would trigger $\mathcal{A}$ if cl would result in unit propagation with respect to $\mathcal{A}$ or trigger a conflict with respect to $\mathcal{A}$. We capture the notion of trigger, formally, in the following definition:

Definition 1. *A clause* cl *of size s triggers on an assignment $\mathcal{A}$ if both of the following conditions hold true:*

1. *For each literal ℓ in* cl*, $\mathcal{A}(\ell) \neq T$*
2. *For at least s - 1 literals $\ell \in$ cl, $\mathcal{A}(\ell) = F$*

Therefore, we view that a clause cl would have been useful to `Thread`, if it would have triggered on a recent assignment trail of `Thread`. Informally, the presence of such a clause cl would have influenced the recent state of the solver in `Thread`. It is worth remarking that Audemard and Simon had also put forth a similar thesis in the context of `glucose` solver [2] in that they measured a clause usefulness by how often it is used in unit propagation or conflict analysis. At this point, a natural question is to determine the partial assignments for which we should determine if a given clause would trigger them or not. While on one end of the spectrum, we could perform such a check for every partial assignment but this would overwhelm `MasterThread`. Therefore, a given thread `Thread` only sends partial assignment where unit propagation completed without conflict.

4.2 Assignment Trigger Check

We now discuss how to perform an efficient check whether a clause triggers on the assignments sent from different CPU threads. To this end, we focus on the efficient bitwise operations for checking whether a clause triggers on the assignments, which were first proposed in [15]. We consider that we are given the k assignments $\boldsymbol{\mathcal{A}}$; we use $\mathcal{A}_i$ for $1 \leq i \leq k$ to represent the i-th assignment. Recall, for every variable v, we represent $\mathcal{A}(v)$ with two bit-vectors of size k. Their i-th bit will represent $\mathcal{A}_i(v)$. For a literal $\ell = \neg v$, $\mathcal{A}(\ell)$ is computed as the negation of $\mathcal{A}(v)$. These representations allows us to check if a clause cl of size s triggers over k assignments at once using bitwise operations. We present the pseudocode of $\mathsf{assignmentTrigger}$ in Algorithm 2. $\mathsf{assignmentTrigger}$ returns a bit-vector res of size k, where $\mathsf{res}[i]$ is set to 1 if the clause cl triggers on the assignment $\mathcal{A}_i$. In this algorithm, & represent the bitwise AND. | represents the bitwise OR. They operate over k bits at once and are executed in only one clock cycle on the GPU provided that $k \leq 32$.

Algorithm 2 $\mathsf{assignmentTrigger}(\boldsymbol{\mathcal{A}}, \mathsf{cl})$

```
1: allFalse ← $[1, \cdots 1]$        ▷ bit-vector of size $k$ with each bit initialized to 1
2: oneUndef ← $[0, \cdots, 0]$       ▷ bit-vector of size $k$ with each bit initialized to 0
3: for $\ell \in$ cl do
4:     oneUndef ← (allFalse&isUndef($\boldsymbol{\mathcal{A}}(\ell)$)) | (oneUndef&isFalse($\boldsymbol{\mathcal{A}}(\ell)$))
5:     allFalse ← allFalse&isFalse($\boldsymbol{\mathcal{A}}(\ell)$)
6: res ← oneUndef | allFalse
7: return res
```

All the clauses can be tested in parallel from each other, so $\mathsf{assignmentTrigger}$ is massively parallelisable, it fits well with the GPU.

4.3 Pooling-based Efficient Trigger Check

While the standard GPU-based implementations allow us to perform check for k up to 32 assignments in parallel, the latency of each check is not sufficient to handle the rate of assignments sent by CPU threads. Observe that a CPU thread sends an assignment for every decision that did not result in conflict upon unit propagation. In this context, one wonders whether we need to perform a check for every assignment. To this end, we relied on two preliminary analyses:

Locality We performed a preliminary evaluation on 100 instances randomly chosen from the SAT 2020 competition, and for each variable, we compute the set of all the values taken between 32 conflicts. Our preliminary evaluation results are presented in Table 1

The first row of Table 1 lists the set of values while the second row indicates the fraction of variables with the corresponding set of values. For example, the

Set of values	{T}	{F}	{T, F}	{U}	{T, U}	{F, U}	{T, F, U}
Fraction of variables	0.063	0.182	0.000	0.588	0.031	0.085	0.048

Table 1. Behavior of successive assignments

entry, $\{T, U\}$ indicates that 3.1% of the variables were assigned the values T and U at least once, but not F, between the 32 conflicts. The entry $\{U\}$ tells us that 58.8% of variables only took the value U between the 32 conflicts. Therefore, we observe that the successive assignments coming from the same CPU thread share a significant similarity.

Sparsity The second key observation, based on preliminary evaluation, is that, on average, less than 1% assignments are triggered by a given clause.

The observations of *locality* and *sparsity* lead us to draw parallels to a fundamental problem in information theory: group testing, pioneered by Dorfman [11]. In the context of the second world war, the task under consideration was to determine the relatively small fraction of sick soldiers in a large army by performing as few tests as possible. Dorfman suggested a simple but effective idea: pool the blood samples of soldiers into groups and perform individualized testing only for the groups that test positive.

We seek to design a similar strategy in the context of determining assignments triggers by a clause. To this end, we aim to capture the concept of *pooling* by defining the notion of an *pooled assignment.*

Definition 2. *Given the assignments* $(\mathcal{A}_i)_{1 \le i \le k}$ *, the pooled assignment* $P : V \mapsto 2^{\{T,F,U\}}$ *is defined by* $P(v) = \bigcup_{1 \le i \le k} \{\mathcal{A}_i(v)\}$*. Similarly, for a literal* ℓ*, we define* $P(\ell)$ *as* $\bigcup_{1 \le i \le k} \{\mathcal{A}_i(\ell)\}$

Definition 3. *For* $p \subseteq \{T, F, U\}$*, we denote* $\neg p$=$\{\neg x \mid x \in p\}$

Proposition 1. *Given the assignments* $(\mathcal{A}_i)_{1 \le i \le k}$*,* P *their pooled assignments and a literal* $\ell = \neg v$*,* $P(\ell) = \neg P(v)$

Proof. $P(\ell) = P(\neg v) = \bigcup_{1 \le i \le k} \{\mathcal{A}_i(\neg v)\} = \bigcup_{1 \le i \le k} \{\neg \mathcal{A}_i(v)\} = \neg \bigcup_{1 \le i \le k} \{\mathcal{A}_i(v)\} = \neg P(v)$

Example 1. Given the assignments $(v_1 \mapsto T, v_2 \mapsto F, v_3 \mapsto U)$ and $(v_1 \mapsto T, v_2 \mapsto U, v_3 \mapsto T)$, their pooled assignment is $(v_1 \mapsto \{T\}, v_2 \mapsto \{F, U\}, v_3 \mapsto \{U, T\})$

Remark 1. The value of a pooled assignment for a variable is a set of truth value p. It can be represented using three Boolean variables (t, f, u), where t (resp. f, u) indicates if $T \in p$ (resp. $F \in p$, $U \in p$).

Remark 2. Given a vector of m pool assignments $\boldsymbol{P}$ and a variable v, we can represent the vector values for v as $(\mathsf{canBeTrue}(\boldsymbol{P}(v)), \mathsf{canBeFalse}(\boldsymbol{P}(v)), \mathsf{canBeUndef}(\boldsymbol{P}(v)))$. $(\mathsf{canBeTrue}(\boldsymbol{P}(v))$, $\mathsf{canBeFalse}(\boldsymbol{P}(v))$ and $\mathsf{canBeUndef}(\boldsymbol{P}(v)))$ are bit-vectors of size m. For a literal $\ell = \neg v$, we have $\mathsf{canBeTrue}(\boldsymbol{P}(\ell)) = \mathsf{canBeFalse}(\boldsymbol{P}(v)), \mathsf{canBeFalse}(\boldsymbol{P}(\ell)) = \mathsf{canBeTrue}(\boldsymbol{P}(v)), \mathsf{canBeUndef}(\boldsymbol{P}(\ell)) = \mathsf{canBeUndef}(\boldsymbol{P}(v))$

Akin to group testing, we seek to first determine whether a clause cl *triggers* a polled assignment. To this end, we extend the definition of trigger.

Definition 4. *A clause* cl *of size s triggers on an pooled assignment P if both of the following conditions hold true:*

1. *For each literal* ℓ *in* cl, $P(\ell) \cap \{U, F\} \neq \emptyset$
2. *For at least s - 1 literals* $\ell \in \mathsf{cl}$, $F \in P(\ell)$

Next we make a simple but crucial observation:

Theorem 1. *Given the assignments* $(\mathcal{A}_i)_{1 \leq i \leq k}$ *and their associated pooled assignment P, if there exists an assignment* $\mathcal{A}_i$ *such that a clause* cl *triggers on* $\mathcal{A}_i$*, then* cl *triggers on P.*

Proof. Let $(\mathcal{A}_i)_{1 \leq i \leq k}$ be assignments, P their associated pooled assignment, $1 \leq i \leq k$, and cl a clause of size s which triggers $(\mathcal{A}_i)$.

From condition 1 of Definition 1, for each literal ℓ in cl, $\mathcal{A}_i(\ell) \neq T$, so $\mathcal{A}_i(\ell) \in \{F, U\}$. In addition, $\mathcal{A}_i(\ell) \in P(l)$. So condition 1. of Definition 4 $P(\ell) \cap \{U, F\} \neq \emptyset$ is met. From condition 2 of Definition 1, there are at least s - 1 literals in cl with $\mathcal{A}_i(\ell) = F$. For all of these, since $\mathcal{A}_i(\ell) \in P(l)$, condition 2. of Definition 4: $F \in P(\ell)$ is met.

By contraposition of this theorem: if a clause does not trigger on a pooled assignment, it does not trigger on all the individual assignments.

Example 2. Given the assignments $(v_1 \mapsto F, v_2 \mapsto T)$ and $(v_1 \mapsto T, v_2 \mapsto F)$, The clause $v_1 \vee v_2$ does not trigger on either of them. It does trigger on their associated pooled assignment $(v_1 \mapsto \{T, F\}, v_2 \mapsto \{T, F\})$ though. This example shows that a clause may trigger on a pooled assignment without triggering on any individual assignment.

We now discuss how the notion of pooling can be efficiently employed to determining assignments triggered by a clause. We previously proposed an algorithm that tested which clause triggered over k assignments (with $k \leq 32$). We propose a new algorithm that returns whether a clause triggers over up to $k \times m$ assignments, where in practice, we choose $m = 32$.

To this end, we employ a two-step process: we first check which pooled assignment a clause triggers. For each pooled assignment it triggers, we employ $\mathsf{assignmentTrigger}$ to locate the corresponding assignments, if any. The successive assignments from a CPU solver thread are pooled together by the `MasterThread` into m pools (in practice, we choose $m = 32$). If the number of CPU threads is lower than m, then we can also create several pools for each CPU solver thread.

The assignments are: $(\mathcal{A}_{ij})_{1 \leq i \leq m, 1 \leq j \leq k}$ where i represents the groups and j the assignment within the pool. While it is possible in practice not to have the same number of assignments within a pool, we will assume that they all have k assignments for simplicity. Note that each assignment maps a variable to a truth value. For each $1 \leq i \leq m$, P_i will be the pooled assignment associated with the assignments $(\mathcal{A}_{ij})_{1 \leq j \leq k}$, that is for every variable v, $P_i(v) = \bigcup_{1 \leq j \leq k} \{\mathcal{A}_{ij}(v)\}$

Algorithm 3 poolTrigger($\boldsymbol{\mathcal{A}} = (\mathcal{A}_{ij})_{1\le i\le m, 1\le j\le k}, \boldsymbol{P} = (P_i)_{1\le i\le m}$, cl)

```
 1: allFalse ← $[1, \cdots 1]$          ▷ bit-vector of size $m$ with each bit initialized to 1
 2: oneUndef ← $[0, \cdots, 0]$         ▷ bit-vector of size $m$ with each bit initialized to 0
 3: for $\ell \in$ cl do
 4:     oneUndef ← (allFalse&canBeUndef($\boldsymbol{P}(\ell)$)) | (oneUndef&canBeFalse($\boldsymbol{P}(\ell)$))
 5:     allFalse ← allFalse&canBeFalse($\boldsymbol{P}(\ell)$))
 6: agTrigger ← oneUndef | allFalse
 7: for bit position $i$ set in agTrigger do
 8:     assigTrigger ← assignmentTrigger(($\mathcal{A}_{ij})_{1\le j\le k}$, cl)
 9:     if assigTrigger $\neq 0$ then
10:         report($i$, assigTrigger)
```

We present the pseudocode of the pooling-based trigger check procedure, called poolTrigger, in Algorithm 3. We now discuss the pseudocode of poolTrigger in detail. The algorithm first performs the check to determine the pooled assignments $\{P_i\}$ such that the clause cl triggers P_i. The check is similar to that of assignmentTrigger, wherein we substitute the usage of bit-vector(isSet, isUndef) with bit-vectors (canBeTrue, canBeFalse, canBeUndef) to account for the need for 3 bits to represent a pooled assignment. Observe that the bit-vectors (canBeTrue, canBeFalse, canBeUndef) represent the set of m pooled assignments, and therefore, are of the size $n \times m$. The bit-vector agTrigger stores which of the pooled assignments are triggered by cl. Observe that even though a pooled assignment P_i may be triggered by cl, it does not necessarily imply that there exists an assignment $\mathcal{A}_{i,j}$ in the assignments associated with P_i such that cl triggers $\mathcal{A}_{i,j}$. Therefore, for each of the pooled assignments triggered by cl, we employ assignmentTrigger to determine whether there exists an assignment triggered by cl and whenever such an assignment exists, the corresponding thread is notified of the clause cl, which is encapsulated in the subroutine report.

4.4 GPU implementation

The representation on the GPU of the assignments is as follows: for each solver thread, we have an array of size n (number of variables), whose elements are the two bit-vectors of size k (Se, Tr) which represent the values of this variable for the assignments of this solver. In addition, there is another array of size n representing the pooled assignments. Its elements are the three bit-vectors (canBeTrue, canBeFalse, canBeUndef).

During a GPU run, we start by updating the assignments on the GPU with the new values of the assignments sent by the CPU threads. Only the modifications with the previous assignments are sent. The pooled assignments are updated at the same time as the assignments. A precise description of the algorithm to update the assignments and pooled assignments is not in scope for this paper.

Then, the GPU finds which clauses trigger on which assignments, using the algorithm 3. Finally, the clauses that did trigger are reported to the CPU threads, which can then import them. We then start the next GPU run.

Similar to CPU solvers, `MasterThread` needs to regularly delete clauses to avoid running out of memory and becoming too slow. So, `MasterThread` performs memory management via activity-based clause deletion. We bump the activity of a clause whenever the clause triggers an assignment. Thus, similar to CPU solvers, we try to keep good clauses and delete bad ones on the GPU.

Assignment of GPU clauses to GPU threads In our implementation, during a GPU run, each clause is looked at by only one GPU thread. In the CUDA computation model, GPU threads are grouped into warps of fixed size, which, in practice, is 32. Therefore, in our case, the 32 threads in a warp will look at 32 different clauses and find which ones trigger by also reading assignments and pooled assignments. Once they are done with these, they will look at the next set of 32 clauses and so on. The GPU is most efficient if all the threads in a warp execute the same instructions at the same time. If divergence happens and only some threads take an execution path, then the others will have to wait. Some instructions have to be executed at the beginning of execution over a clause and at the end. So, it is better if all threads in a warp execute these instructions at the same time. We also chose to have all the threads in a warp look at clauses of the same size. This avoids the case where a single thread looks at a very long clause and blocks the entire warp from moving to the next set of clauses. To achieve this, we chose to group all the clauses on the GPU by their size. Each warp will only look at clauses of a given size.

Using memory coalescing in CUDA Reading the GPU memory is most efficient if the reads are coalesced, which happens, for example, if successive threads in a warp read successive 4-byte words in memory. We chose to coalesce reading the clauses by reordering how we represent them in memory to how they are accessed. Firstly, all the threads in a warp will read the first literal of their clauses. Then, they will read their second literal, and so on. So, we chose to order clauses in memory following this pattern. The first 32 4-bytes words will represent the first 32 literals of the clauses of a warp. The next 32 4-bytes words will represent the next 32 literals of the clauses of a warp, and so on.

Thanks to this coalescing, reading the clauses can be performed efficiently. Whenever a literal from a clause is read, we need to read the value of that literal for the pooled assignments or assignments. That variable might be anywhere in memory, so the access pattern for reading the assignments is not as efficient. However, they might be cached, especially if there are fewer variables

Whenever a GPU thread finds a clause that triggers, it needs to report the clause to the CPU. We chose to only report an identifier of the clause (its size and position within that size), as well as the assignment it triggers on. By keeping a copy of all the clauses on the CPU, we can reconstruct the full clause just from this clause identifier. The clause reported is added to a queue. We use an atomic operation to increment the position in the queue by one and write to that position. This avoids having two threads writing to the same position in the queue at the same time.

Avoiding clause duplication on the CPU We aim to avoid a CPU thread having the same clause multiple times in its database, as it slows down the thread due to unit propagation overhead and uses up memory. If a CPU thread has a clause cl in its database, then cl will not trigger on assignments sent by this thread to `MasterThread`, so cl will not be reported again. However, after a thread sends an assignment to `MasterThread`, the thread does not wait to receive clauses before sending more assignments. Therefore, it is possible for a thread to send two assignments to `MasterThread` and for a clause cl to trigger on both the assignments. Therefore, in order to avoid reporting cl multiple times to the same thread, `MasterThread` keeps for every thread a set of GPU clause identifiers that have been reported recently.

5 Evaluation

We developed a prototype implementation of GPUShareSat in the form of a library, and we augmented two state-of-the-art award-winning SAT solvers with GPUShareSat [3]. The objective of our empirical evaluation was two-fold: runtime performance comparison of solvers augmented with GPUShareSat vis-a-vis state of the art parallel solvers, and an in-depth investigation of the inner working of GPUShareSat. In particular, we sought to answer the following questions:

RQ 1 How does the runtime performance of solvers augmented with GPUShareSat compare to that of the winners of parallel track in recent SAT competitions?
RQ 2 How effective is the pooling-based strategy?
RQ 3 What are the characteristics of the workload on the GPU?

Experimental Setup Our empirical evaluation seeks to follow the recently released SAT practitioner manifesto. We use the standard 5000 seconds wall-clock timeout on the benchmarks from SAT competition 2020. All experiments were conducted on a high-performance computer cluster, each node consisting of nodes with 2×E5-2690v3 CPUs containing 2×12 real cores, 96GB of RAM, and an NVidia K40 GPU.

To answer **RQ 1**, we integrated GPUShareSat with Relaxed_LCMDCBDL_newTech, which scored 2nd place at the 2020 SAT Competition's main (i.e. single-core) track, and called the resulting implementation Relaxed-Gpu. In Relaxed-Gpu, we removed the duplicate learnt clause detection (`is_duplicate`) and associated data structures, as they could use up to 50GB of memory for a single thread. We used the same parameters and search strategy for all CPU threads to isolate the performance improvement to our system. The diversity of search came only from the fact that clause exchange is non-deterministic due to scheduling differences. If we were to use differing parameters for the threads, as solvers taking the portfolio approach usually do, we would expect performance to improve.

[3] The accompanying artefact consisting of detailed statistics is available at https://doi.org/10.5281/zenodo.4764813

We perform a runtime performance comparison of Relaxed-Gpu vis-a-vis P-MCOMSPS-STR, the winner of the SAT competition 2020 parallel track. We analyzed the performance of P-MCOMSPS-STR for 12 and 24 threads, and surprisingly, the implementation of P-MCOMSPS-STR with 12 threads outperformed P-MCOMSPS-STR with 24 threads by 14 more problems solved and 10% better PAR-2 score and therefore, we perform a comparison of Relaxed-Gpu for 12 threads.

Furthermore, for **RQ 2** and **RQ 3**, we augmented GPUShareSat with glucose-syrup given the ease of collection of detailed statistics from glucose-syrup. We refer to the augmented version by glucose-syrup-gpu. Since the objective of **RQ 3** is to understand the workload with as many CPU threads as possible, we run the experiments with 24 threads. We report the average of all the statistics over all the benchmark instances.

Summary of the Results

We observe that Relaxed-Gpu significantly outperforms P-MCOMSPS-STR, the winning parallel SAT solver of the 2020 SAT competition, both in terms of the number of solved instances and PAR-2 score. In particular, while P-MCOMSPS-STR could only solve 276 instances with a PAR-2 score of 3493, Relaxed-Gpu could solve 296 instances with a PAR-2 score of 3164. In the context of **RQ 2**, we observe that usage of pooling allows us to skip invoking of assignmentTrigger for over 99.8% of the assignments. Finally, GPUShareSat managed to keep up with as many as 24 CPU threads both in clauses checked per second and in the number of clauses kept in memory, enabling GPUShareSat to efficiently and effectively inform the CPU threads of the clauses they should import. Observe that the GPU employed in our experiments, the NVidia K40, would be classified as a relatively slow GPU from the perspective of GPUs typically employed by the deep learning community. A more modern GPU such as an NVidia RTX 3080 has $\approx$ 10x the performance relative to the K40.

Overall, our results demonstrate that GPUShareSat can be used either as a drop-in tool to improve already parallel SAT solvers, or to make single-threaded SAT solvers into powerful, multi-threaded SAT solvers that can outperform state of the art parallel SAT solvers.

RQ 1: Relaxed-Gpu vis-a-vis P-MCOMSPS-STR Figure 2 presents the runtime performance comparison of Relaxed-Gpu vis-a-vis P-MCOMSPS-STR in form of a cactus plot. A point (x, y) represents the corresponding solver solved x instances in less than y seconds. The figure clearly shows the performance improvement achieved by Relaxed-Gpu in comparison to P-MCOMSPS-STR. In particular, Relaxed-Gpu solved 296 instances with a PAR-2 score of 3164, while P-MCOMSPS-STR could only solve 276 instances with a PAR-2 score of 3493. In particular, P-MCOMSPS-STR solved 143 SAT and 133 UNSAT instances while Relaxed-Gpu solved 175 SAT and 121 UNSAT instances. Note that Relaxed_LCMDCBDL_newTech was significantly stronger on SAT than

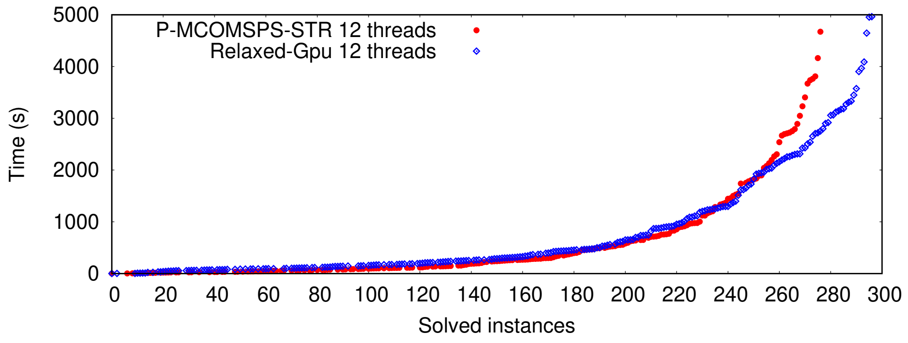

Fig. 2. Relaxed-Gpu vis-a-vis P-MCOMSPS-STR.

UNSAT instances in the SAT Competition of 2020; hence the slanted distribution towards SAT in the case of Relaxed-Gpu is expected.

RQ 2: Impact of Pooling-based Trigger Check The primary rationale behind the usage of pooling-based trigger check was the potential benefits due to avoidance of the redundant invocation for assignmentTrigger for a large number of assignments when their associated pooled assignment does not trigger the clause of interest. To this end, we computed *no-triggers*, defined as the fraction of tuples (P, cl) such that the clause cl did not trigger the pooled assignment P over the set of all possible tuples. Our evaluation indicated *no-triggers*, averaged over the entire set of instances, is 0.9984, which represents a significant reduction in the number of invocations of assignmentTrigger. The primary reason behind a surprisingly high value of *no-triggers* is that the average learnt clause size for glucose-syrup was 54.43 literals/learnt clauses. For such large learnt clauses, there is a high likelihood of a learnt clause containing two unassigned literals with respect to a given pooled assignment.

RQ 3: Characteristics of the workload of GPU Since our analysis of the characteristics appeals to the performance improvement due to GPUShareSat in the context of glucose-syrup, we first present the corresponding cactus plot in Figure 3. It is worth remarking that while glucose-syrup solved 252 instances with a PAR-2 score of 4208, glucose-syrup-gpu solved 263 instances with a PAR-2 score of 3853.

We now delve deeper into analyzing the characteristics of the workload of the GPU. The average number of clauses imported per thread was 0.120 per conflict or 0.210 per assignment sent to the GPU. In the case of glucose-syrup, the corresponding number was 1.27 clauses per conflict, which is 10.6 times higher. Therefore, glucose-syrup-gpu is able to import fewer but useful clauses.

The number of clauses checked by the GPU in glucose-syrup-gpu was 160.3 million clauses/s on average. Clause checks were performed over a number of

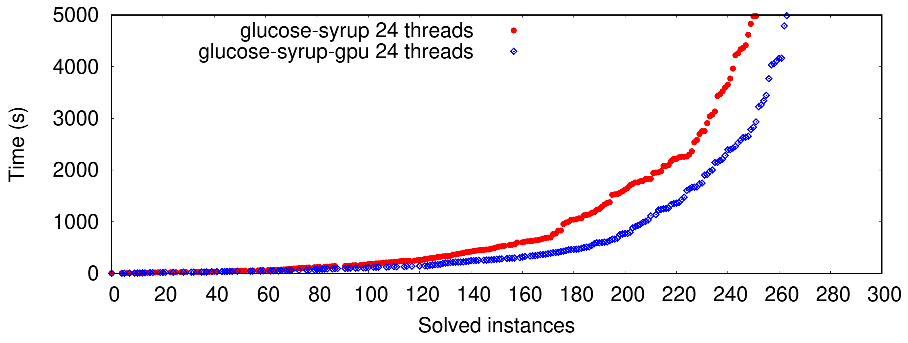

Fig. 3. glucose-syrup vis-a-vis glucose-syrup-gpu. glucose-syrup solved 124 SAT and 127 UNSAT instances. glucose-syrup-gpu solved 132 SAT and 131 UNSAT instances.

assignments at once, and the number of clause tests on individual assignments was 18.66 billion per second on average. Note that most of these clause tests did not require a call to assignmentTrigger thanks to the pool checks. To put this number into perspective, glucose-syrup propagated 0.850 million literals/thread/s and checked for propagation 16.69 million clauses/thread/s. Therefore, from the perspective of checking clauses, a single 12 core E5-2690v3 CPU performed on a much lower scale than a single NVidia K40 GPU.

When it comes to learnt clauses kept in memory, we see a similar pattern. glucose-syrup-gpu kept an average of 1.50 million learnt clauses in the GPU during solving, while a single thread of glucose-syrup kept 77,436, amounting to 1.85 million learnt clauses for 24 threads.

6 Conclusion

We design an efficient framework, called GPUShareSat, to take advantage of heterogeneous architectures. We identified that GPUs can efficiently determine the clauses that a CPU thread should import from other threads. We observed that GPUShareSat integrated with Relaxed_LCMDCBDL_newTech significantly outperforms the parallel track's winning solver from the 2020 SAT competition. We have released GPUShareSat as an open source library and we hope our results will encourage the community to integrate GPUShareSat into existing solvers.

Acknowledgments This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme[NRF-NRFFAI1-2019-0004] and Sung Kah Kay Assistant Professorship Endowment. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg.

References

1. Relaxed backtracking with rephasing. In *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*, Department of Computer Science Report Series B 2020-1, pages 15–16. University of Helsinki, 2020.
2. Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009*, pages 399–404, 2009.
3. Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel sat solvers. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 197–205, Cham, 2014. Springer International Publishing.
4. Sander Beckers, Gorik De Samblanx, Floris De Smedt, Toon Goedeme, Lars Struyf, and Joost Vennekens. Parallel hybrid SAT solving using OpenCL. In *Benelux Conference on Artificial Intelligence.* Springer, 2012.
5. Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008*, volume 4996 of *LNCS*, pages 28–33. Springer, 2008.
6. Armin Biere. Lingeling, plingeling and treengeling entering the SAT competition 2013. In *Proceedings of SAT Competition 2013*, vol. B-2013-1 of Department of Computer Science Series, pages 51–52. University of Helsinki, 2013.
7. Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10), 2008.
8. A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Struct. Algorithms*, 27(2):201–226, September 2005.
9. Adam Coates, Brody Huval, Tao Wang, David J. Wu, Bryan Catanzaro, and Andrew Y. Ng. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 1337–1345. JMLR.org, 2013.
10. R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.
11. Robert Dorfman. The detection of defective members of large populations. *The Annals of Mathematical Statistics*, 14(4):436–440, 1943.
12. Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Proc. of SAT*, pages 61–75, 2005.
13. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
14. Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel SAT solver. *J. Satisf. Boolean Model. Comput.*, 6(4):245–262, 2009.
15. Marijn Heule and Hans van Maaren. Parallel SAT solving using bit-level operations. *J. Satisf. Boolean Model. Comput.*, 4(2-4):99–116, 2008.
16. Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability.* Addison-Wesley Professional, 1st edition, 2015.
17. Sharad Malik, Ying Zhao, Conor F. Madigan, Lintao Zhang, and Matthew W. Moskewicz. Chaff: Engineering an efficient SAT solver. *Design Automation Conference*, pages 530–535, 2001.

18. Panagiotis Manolios and Yimin Zhang. Implementing survey propagation on graphics processing units. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *LNCS*, pages 311–324. Springer, 2006.
19. Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 131–153. ios Press, 2009.
20. Bertrand Mazure, Lakhdar Sais, and Éric Grégoire. Boosting complete techniques thanks to local search methods. *Ann. Math. Artif. Intell.*, 22(3-4):319–331, 1998.
21. Muhammad Osama and Anton Wijs. Parallel SAT simplification on GPU architectures. In Tomás Vojnar and Lijun Zhang, editors, *TACAS 2019*, volume 11427 of *LNCS*, pages 21–40. Springer, 2019.
22. Muhammad Osama, Anton Wijs, and Armin Biere. Sat solving with gpu accelerated inprocessing. In *Proc. of TACAS*, 2021.
23. Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Cud@sat: SAT solving on gpus. *J. Exp. Theor. Artif. Intell.*, 27(3):293–316, 2015.
24. Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 294–299, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
25. Vadim Ryvchin and Ofer Strichman. Local restarts. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008*, volume 4996 of *LNCS*, pages 271–276. Springer, 2008.
26. Vincent Vallade, Ludovic Le Frioux, Souheib Baarir, Julien Sopena, Vijay Ganesh, and Fabrice Kordon. Community and lbd-based clause sharing policy for parallel SAT solving. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020*, volume 12178 of *LNCS*, pages 11–27. Springer, 2020.
27. Peter van der Tak, Marijn Heule, and Armin Biere. Concurrent cube-and-conquer. *CoRR*, abs/1402.4465, 2014.
28. Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4):543–560, 1996.