Received May 2, 2020, accepted May 28, 2020, date of publication June 2, 2020, date of current version June 11, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2999382

An Effective SAT Solver Utilizing ACO Based on Heterogenous Systems

HASSAN YOUNESS[1], MUHAMMAD OSAMA[1], AZIZA HUSSEIN[1], MOHAMMED MONESS[1], (Member, IEEE), AND AMMAR MOSTAFA HASSAN[2]

[1]Department of Computers and Systems Engineering, Minia University, Minia 61519, Egypt
[2]Arab Academy for Science, Technology, and Maritime Transport, South Valley Branch, Aswan 81516, Egypt

Corresponding author: Hassan Youness (hassan_youness@mu.edu.eg)

ABSTRACT This paper presents new parallel strategies for preprocessing and solving the issue of Boolean Satisfaction (SAT) on Heterogeneous systems of multicore and many-core CPU and Graphics Processing Unit (GPU) using Open Multi-Processor (OpenMP) and NVIDIA - CUDA. We propose exceptionally proficient and parallel techniques for SAT simplifications using the variable elimination method based on the Davis-Putnam-Logemann-Loveland (DPLL) slitting rule algorithm performed with a shared-memory model on a multicore CPU platform, where the clause elimination subsumption and the pure-literal removal techniques are completely performed on the CUDA framework. We demonstrate how efficient an evolutionary SAT solver is by using the suggested heterogeneous pre-processing, leading to important acceleration improvements in the solution's quality enhancement. The penalization of the transformative SAT solver is executed with Ant Colony Optimization (ACO) scheme utilizing CUDA. (Compute Unified Device Architecture) We perform thorough benchmarks to test the performance of our preprocessor and solver implementations against various random SAT formulas. The promoted H-SAT pre-processor scheme has gotten a speed-up of a factor 15x over the sequential implementation with statistical reductions on the original CNF which becomes up to 49% and 43% in case of literals and clauses numbers exclusively, where the H-SAT gain strength the solvability of the ACO solver by 100% in some cases.

INDEX TERMS Ant colony optimization, CUDA, GPU, heterogeneous, pure-literal elimination, satisfiability, subsumption.

I. INTRODUCTION

For a multitude of reasons, interest in Boolean satisfaction is growing as more issues are now being solved more quickly by SAT solvers over others. This is undeniable because satisfaction is at the intersection of logic, fault diagnosis [1]–[3] automatic program testing [4]–[6], auto debugging systems based on real-time [7], biological systems [8], [9], and computer engineering studies in general [10]–[13]. Particularly many problems stemming from one of these areas has various Satisfaction translations or encodings, and there are numerous numerical techniques accessible for the SAT solution to help in solving them with improved performance.

In particular, several modern evolutionary solvers have been implemented for heterogeneous parallel architectures with prior simplifications. Numerous techniques are generally used in the field of computational problems, which can produce significantly faster algorithms: complete and incomplete (tentative) techniques.

Although no schemes are familiar for fixing the SAT problem effectively, or optimally for all possible cases or formulas, there are some of the problems as circuit design [16], [17], and automatic theorem proofing [18], can be resolved rather efficiently using incomplete or empirical SAT-solvers. Such schemes are not accepted to be effective on all SAT occasions, however tentatively these schemes will in general function admirably for some reasonable applications. Even though the solution quality, i.e. how many trials required to find best possible solution, of this kind of solvers can be further enriched by using variable and clause eliminations for reducing SAT input formulas.

Modern solvers and preprocessors such as Non-increasing Variable Elimination Resolution (NiVER) [19] and SatELite [20] are based on DPLL algorithm [21]. Subsumption, Unit propagation, and pure-literal removal [22], [20], [23], [24] are the best well-known simplification methods

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

used in most of the DPLL based SAT preprocessors. The NiVER process is like to other simplifiers accepts a Conjunctive Normal Form (CNF) as inputs and outputs with a less or equal number of variables by resolving away variables that have a limited number of occurrences, i.e., how many times a variable appears in a SAT formula.

The NiVER technique considers the least constrained variables to be removed away, where our implementation of a variable elimination technique considers the Extreme Variated Variables (EVVs), i.e., the variables that appear in the highest number of clauses. These factors/ variables are settled utilizing the slitting rule of the DPLL algorithm, which depends on crafted by Moritz and Springer [25]. The SatELite extends the NiVER process by subsumption elimination, where the slitting of a CNF with a particular variable yield an enormous number of subsumed clauses and pure-literals. The clauses which enfold a purely literal and thus that have been subsumed ought to be detected and removed instantly to save memory and simplify further the output of the CNFs.

Currently, multicore and many-core multiprocessors are becoming prevalent in SAT tackling problem, where several major solvers are skillfully introduced to minimize the timing output as observed in [26], [27], and [28]–[30]. The previous analysts utilized an arrangement of comparing successive schemes got through cautious varieties of the standard DPLL scheme to build on CPU. The other is a parallel version from the Mini-SAT solver [31]–[35] that uses the farm strategy which creates a master process responsible for slitting the original formula with guiding paths (assumptions) and sending to slaves.

There might be multiple sub-formula per slave, but each receives one at a time. When a slave is finished with its offer, it sends its outcomes to the master and waits for further research. A master sends more work to the slaves while no solution is found or while there are sub-formulae to be solved. The latter introduces a 3-SAT solver that uses CUDA to adopt a deterministic strategy implemented on the GPU. Since every one of its clauses is 3 literals long, it picks a clause and tests 3 mixes of factors attributions for its literals: The first is true; the first is strict and the second is real and the first and subsequent literals are independently false and genuine. It is intended for arbitrary occurrences, which are commonly difficult to comprehend, despite when little, for lacking inward structures to be exploited.

So far, none of these measures were intended to be parallel incomplete SAT solver that enhanced with a parallel preprocessor on the heterogeneous multi-core processing unit architecture. This paper shows a proficient, and quick parallel heuristic SAT solution with H-SAT pre-processor. The solver applies the ACO algorithm [36]–[39] based on Springer's work [25] and implemented with CUDA on GPU. [40], [41]. Our suggested H-SAT preprocessor uses OpenMP to display a skilled variable removal method [42], [43] to make full use of the multicore CPU based on a shared-memory model and notable quick parallel subsumption algorithms and pure-literal elimination architecture based on (Single-Instruction Multiple Thread) SIMT shared-memory architecture for complete GPU operation with CUDA.

The primary enrichment of this article is to use variable elimination, subsumption and pure-literal cuts on the CPU-GPU system using the parallel SIMD architectures to achieve a fine-simplified SAT CNF that is proper for our solver utilizing the Max-Min Ant System (MMAS) method to SAT solving [44]–[47], requiring trivial formulas to be processed.

II. BACKGROUND AND RELATED WORK

This area audits the SAT issue and how it very well may be preprocessed utilizing the DPLL slitting rule (variable elimination), the clause subsumption elimination algorithm, pure-literal elimination algorithm, and outlines the basic procedure of the MMAS for SAT ex-plaining. For extra subtleties, we urge the per user to read the authentic works of Subbarayan and Pradhan [19]; Eén and Biere [20]; Zhang [48]; Stützle [44]; Moritz and Springer [25]; Villagra and Barán [45]; Youness *et al.* [46].

A. BOOLEAN SATISFIABILITY ISSUE

The Boolean or propositional satisfaction problem can only be answered in one sentence: given a Boolean formula, it is conceivable to decide if Boolean qualities are allocated to the propositional factors in the recipe so that the formula is assessed as real. The formula is considered satisfactory if such an assignment exists; otherwise, it is unsatisfactory.

For a combinatorial problem to be solved using the latest SAT schemes, it usually has to be encoded into a CNF: sequence of clauses $\wedge_i c_i$, where, each clause c_i is a disjunction of k literals $\vee_m l_m$, and every literal l_m being either a Boolean variable v or its negative $\bar{v}$. There are some variations of k-SAT formulas like 3-SAT ($k = 3$) which also falls within the NP-complete problem category. 3-SAT restricts the literal numbers for each clause to precisely 3 literals. CNF is a simple form, easy to implement, and its common format for files. A format for files CNF SAT issues conceived and pursued since in the DIMACS Challenge [49]. The popular file format promoted the compilation of SAT benchmark issues on the SATLIB website, as well as the periodic SAT solver competitions [50], which stimulated much research into effective algorithms and implementations.

There are two primary algorithm classes that were created to fix SAT cases. The first class is the full algorithms that are guaranteed to end with a right choice as to whether the CNF is satisfied or unsatisfied. DPLL and Clause Learning (CDCL) Conflict-Driven algorithms fall into the full algorithm category [26], [21], [51]–[55]. The second class is incomplete schemes that don't give the assurance that a good satisfactory assignment will either be reported in a preset time limit or declared unsatisfactory, but a solution can be found quicker than a complete algorithm. Our parallel ACO-SAT solver on GPU [46] is based on incomplete approaches which will be revised briefly in this paper.

Algorithm 1 Elimination of Dynamic Variable Based on DPLL Slitting Algorithm, Where *Slit Has Two Parameters (Form = Formula, Pro = Propositional)*

```
1:  Slit(Form, Pro)
2:  $i = 0$            // i is a variable
3:  while ($i \neq limit$ (Pro))   // limit is a size
4:      $v =$ Pro ($i$)
5:      for ($C \subseteq For\ m$)   // Clause subset of Formula
6:          if ($v \in C$)   // belong to
7:              delete $C$; reduce Form
8:          elseif (($\bar{v} \notin C$) ($limit(C) == 1$)
9:                  return false
10:         end if
11:             delete $\bar{v}$
12:                 Pro $\leftarrow$ Pro $\cup C$
13:     end for
14: end while
```

B. DPLL SLITTING METHOD

In this method, the SAT formula is factored or split by choosing a variable v, generating two simplified formulae. Factored formulas can again be factored by another variable [25], [56], [57]. We assign the chosen variable v once a true value to get one of the two new formulas and once a false value to get the other formula. If we can prescribe a CNF formula in the following form: $S = (C_i \vee v) \wedge \ldots \wedge (C_n \vee v) \wedge (C_t \vee \bar{v}) \wedge \ldots \wedge (C_n \vee \bar{v}) \wedge S_r$, where C_i and C_t are clauses wherein v and $\bar{v}$ do not appear together, and S_r is a set of clauses in which v and $\bar{v}$ do not appear, then we can acquire two formulae $S' = C_i \wedge \ldots \wedge C_n \wedge S_r$ and $S'' = C_j \wedge \ldots \wedge C_n \wedge S_r$.

The set S is unsatisfiable if and only if S' and S'' are unsatisfiable, where S' and S'' are pure-literal formulations. The slitting rule is applied by recursively removing the clauses that have the positive literal v because it is now satisfied, otherwise we remove the negative literal $\bar{v}$ from any clauses if found. The factored formula is unsatisfiable if we end up with an empty clause and satisfiable if we end up with no clauses.

We introduce a new sequential implementation for this method using dynamic programming to over-whelm the pitfalls of recursion and the out-of-memory exceptions (see Algorithm 1). Dynamic programming is an approach for optimization that converts a complicated issue into a sub-problem series; its essential characteristic is overlapping these sub-problems without any recursion taking far less time than the other traditional methods.

The input parameter, *Pro* of the procedure is a vector of variables chosen to fragment the input formula. Each variable v returned by this vector is stored in a CPU register (step 4) to accelerate the operations performed on v thru minimizing the system memory traffic. Note that the formula and propositional vectors are updated and reallocated inclusively inside the procedure; which is considered a major advantage for using the dynamic programming concepts. A modern innovated algorithm calculates the number

Algorithm 2 The Proposed Algorithm for Parameters Settings, Where, *max_nop* Is the Max. No. of Parameters, *Occurrence* Is the No. of Occurrence for Every Variable in Each Clause, And *Occur* Is the Occurrence Parameter

```
1:  parameters _ settings (EVV, max_nop)
2:  $i = 0$, $nop = 0$, $occurrence = 1$
3:  While ($i \neq limit$ (EVV)))
4:    $v \leftarrow$EVV($i$)
5:   if (occur($v$) >= occurrence) then
6:      nop ++, $i$ ++
7:       if (nop >max_nop)then
8:          occurrence ++, nop = 0; i = 0
9:       end if
10:  elseif (nop == 0 )then
11:      nop = max_nop
12:  end if
13: end while
14: return nop, max_nop
```

of variables (parameters) in a propositional that called *parameters_settings* (as shown in Algorithm 2). This algorithm acquires the most appropriate number of the EVVs consistent with their appearance in the formula and an input limit variable called *max_nop* initiated by the user.

C. SUBSUMPTION

Assume that *lit* (C) in the formula of CNF shows the set of literals in clause C. Given the clauses C_1 and C_2, if *lit* $(C_1) \subseteq$ *lit* (C_2) then C_1 subsumes C_2. A subsumed section is redundant and can be withdrawn without changing the representation of Boolean functions from the CNF formula. The main drawback of variable elimination is that it produces many extra clauses which subsume or is subsumed by another clause. Since redundant clauses ingest memory and time in SAT solving, it is more desirable to detect and remove the subsumed clauses immediately after the variable elimination phase is completed.

In modern SAT preprocessors like SatELite, whenever new clauses are added to the formula, it is checked against current provisions in the database to see if they are subsumed or not. This check is called a reverse/backward subsumption [20] that can be applied during SAT goals, which is now being presented in most SAT solvers. Furthermore, the fresh clause is checked against the current provisions to see whether it is subsumed by any of them; this check is referred to as the forward subsumption [20].

In our implementation, we introduce a straightforward brute-force algorithm that is suitable for SIMT architectures and performs extremely fast if it executed on these parallel platforms. The sequential and the parallel techniques are presented in this paper as shown in Algorithms 3, 6, respectively.

First and foremost, at the presented sequential algorithm, we sort the clauses in the input formula according to their sizes, then we do a brute-force search for the subsumed clauses and flag them at once to be removed later. If the size

Algorithm 3 Serial Subsumed Removal

```
1:  Subsumed_removal (Form)
2:  for (C ∈ Form)
3:    for (C' ∈ Form)
4:      subsume = true
5:      if (C' ≠ C)
6:        for (L ∈ C )    //L is a group of literal
7:          if (L ⊄ C')
8:            subsume = false
9:        else
10:             Remove (C' )
11:        end if
12:       end for
13:     end if
14:   end for
15: end for
```

of the formula is n, it will have a complexity of $O(n^2)$ which is considered high compared to its parallels in the present state-of-the-art pre-processors, conversely, the complexity can be *O(1)* if we allocate each clause to separate worker to do the check. With that granularity, our algorithm is more likely to perform faster than any other counterpart exists if it is implemented and executed on SIMT architectures such as the GPU.

D. ELIMINATION OF PURE-LITERALS

Initially, the pure-literal principle was expected for the advancement of the unit-clause spread of the DPLL technique. The unit propagation or the Boolean Constrained Propagation (BCP) searches over each clause (except for the unit clause itself) that comprises the unit literal l to be removed. If a clause found inclosing l, this literal is deleted.

In the pure-literal rule, in a CNF formula S, a literal l is called pure if and only if $\bar{l}$ does not occur in S. Pure-literal words may still be added, without affecting satisfaction, which adds together with the removal of the clauses. As this can make other literals pure, the method has to be iterated in order to produce an equal formula for satisfaction utilize with no pure-literal materials. This is called pure literal removal [24]. Consider the following formula S of the CNF with a pure b literal:

$$S \rightarrow (a \vee b) \wedge (-c \vee a) \wedge (c \vee d) \wedge (-a \vee -d \vee b)$$

By implementing a fresh set, S' of pure-literal removal will be

$$S' \rightarrow (-c \vee a) \wedge (c \vee d)$$

A complete list of the new parallel algorithm for elimination in pure literality can be found in the next section. The algorithm is made up of two principal steps:

1: detect each variable for all clauses, storing their footsteps, counting their appearance (polarity is excluded), and finally summing their values (polarity is included), see Algorithm 7 (detector algorithm).

2: checking the virtue of every factor and correspondingly evacuating the clauses which hold any factor that has been observed pure, as seen in Algorithm 8 (implementer algorithm).

E. MAX-MIN ANT SYSTEM

The SAT procedure Max-Min Ant System (MMAS) has three primary phases attempting to discover the best possible alternative: the production of an ant colony; refreshing of pheromones, and obscuring of pheromones [25]. All phases are repeated until the condition of termination is fulfilled. The ant colony comprises m artificial ants, where m is a parameter defined by the user.

Each ant j constructs its alternative by comparing a random value with a likelihood (random proportional rule) for selecting a literal $\left(l \in L^{2\times n}\right)$ positive or negative, where L represents a twin set of factors n and their complement in a b clauses SAT formula. The random proportional law lies in the attractiveness of the pheromone. and the heuristic EVV [25], [46], i.e., the factors appear more desirable by ants in most provisions/clauses.

$$p_j(l) \leftarrow \frac{ph_{lj}^{\alpha}.evv_{lj}^{\beta}}{\sum_{l\in L} ph_l^{\alpha}.evv_l^{\beta}} \tag{1}$$

where: ph_{lj} is the measure of pheromone as of now arranged by subterranean ant j on literal l; evv_{lj} is the extreme variated variables heuristic of exacting l for an ant j; and α *and* β are client characterized parameters to control the adequacy of ph_{lj} and evv_{lj}.

When an ant chooses a candidate for a solution, the candidate will be assessed to compute the quantity of clauses that satisfiable in the SAT formula and the quality of the assessment. The quality of the assessment is the number of clauses complied with multiplied by the weights of certain things calculated by the heuristic rule of weight adaptation [45], [46]. Heuristic weight adjustment is to increase the importance of unresolved provisions during each period of assessment.

After every one of the ants has scanned for choices, it is important to refresh the pheromone esteems for all literals and their limits. To evade the slump of the populace or falling into local optima, every quantity of pheromone is reduced by a factor defined by the client called the dissipation rate ρ [36], [37], [45]. This enables ants to overlook terrible assignments, where all pheromones amount and points of confinement are refreshed as follows:

$$\tau_l \leftarrow (1-\rho)\,\tau_l + \Delta\tau\,(x,l)$$
$$\text{With}\quad \Delta\tau\,(x,l) \leftarrow \begin{cases} f\,(x) \\ f\,(\bar{x}) \end{cases},\ (x,l) \in i^* \tag{2}$$

where (ρ) is the dissipation rate $0 \leq \rho \leq 1$, τ_l is the ant pheromones level, and i^* is the present best arrangement per cycle i. The amount of a pheromone is legitimately corresponding to the target function *f(x),* i.e., the picked task x duplicated by the assessment quality. The points of confinement of the pheromones refreshed by the accompanying

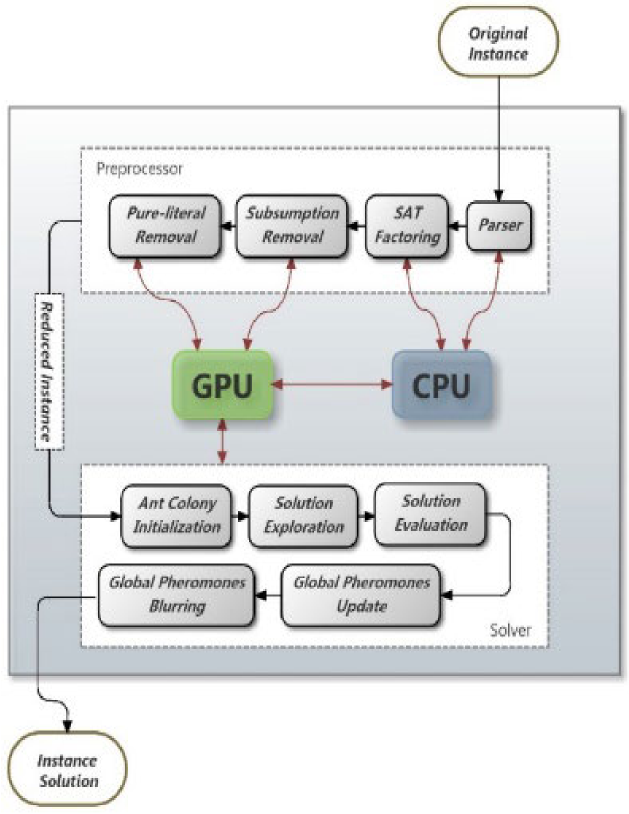

FIGURE 1. **The recommended pre-processor and solver outline.**

equations as follows [36], [45].

$$\begin{aligned} &\text{if } \tau_l < \tau_{min}, \quad then\ \tau_l \leftarrow \tau_{min} \\ &if\ \tau_l > \tau_{\max}, \quad then\ \tau_l \leftarrow \tau_{\max} \end{aligned} \tag{3}$$

where $\tau_{max} = \frac{n}{1-\rho}$ and $\tau_{min} = \frac{\tau_{max}}{2n}$

III. CPU-GPU IMPLEMENTATIONS

This segment presents the parallel methodology of the H-SAT preprocessing scheme and the ACO solver on CPU and GPU using OpenMP and CUDA as seen in Fig. 1. A flow diagram in Fig. 2 reviews the workflow of the H-SAT preprocessor implementation on both multicore CPU and many core GPU.

The ACO solver may processed a simplified CNF stemmed from the H-SAT or the input original CNF, so we perform benchmarks to test the solving performance of the ACO solver on both fed formulas and compare the solving times of both runs.

Toward the beginning of the ACO strategy, we set the calculation parameters (m, α, β, etc.), parse the SAT occasion which is spoken to in DIMACS file (streamlined or unique), assign memory for memory requirements and grids, make CUDA streams [58], for example an arrangement of instructions that execute in issue-request on the GPU. Furthermore, a CUDA arbitrary number generator RNG sent in cuRAND library [59] that accompanies NVIDIA SDK with various sorts of excellent RNG schemes has been set up. The host

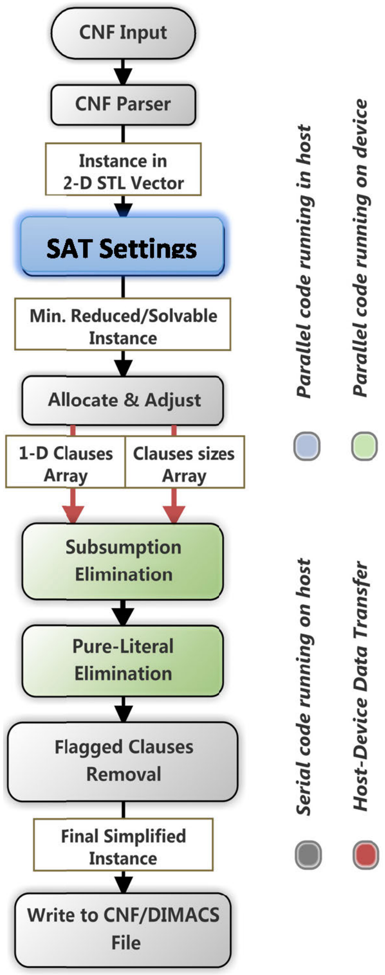
CNF Input
CNF Parser
Instance in 2-D STL Vector
SAT Settings
Min. Reduced/Solvable Instance
Allocate & Adjust
1-D Clauses Array
Clauses sizes Array
Subsumption Elimination
Pure-Literal Elimination
Flagged Clauses Removal
Final Simplified Instance
Write to CNF/DIMACS File
Parallel code running in host
Parallel code running on device
Serial code running on host
Host-Device Data Transfer

FIGURE 2. **The workflow of the H-SAT implementation.**

(CPU + Machine Memory) side initializes the pheromone concentrations, heuristic EVV, and probabilities. We only copy all data synchronized to the global memory of the GPU once to avoid the possible overhead communication with the CPU.

A. THE CNF PARSER

The initial parsing of the SAT formula, which is usually reproduced in CNF or DIMACS format, is required [49] in case to begin the preprocessing stage. In these formats, each clause is shown as a collection of signed entries, where a negative value is the negated variable; for formula, (1 – 42) stands for the clause ($v_1 \vee \bar{v}_4 \vee v_2$). The CNF parser peruses out the number of factors and clauses from the content document, at that point peruses every clause and put it in a separate (Standard Template Library) STL vector [60].

Each vector of the clause is pushed to the STL *2-D* vector of formula. We use *1-D* host side arrays to allocate memory (CPU + System Memory) to formula clauses and their dimensions. We then only once copy the assigned information to the device's memory (GPU worldwide memory) to perform the necessary subsumption and literal removal computations

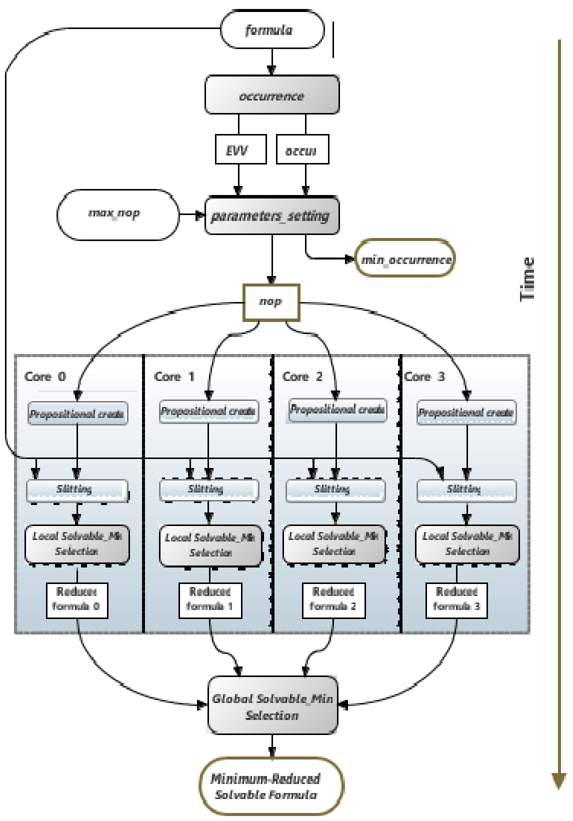

FIGURE 3. SAT settings segment.

on the GPU. The information transfers performance penalty is not more than a few microseconds and may not be taken into account. As appeared in Fig. 2 that outline the H-SAT usage.

B. SAT SETTINGS

This phase applies the DPLL slitting rule in the formula at which phases from 3-5 are performed on each processor of the CPU (Fig. 3):

1: calculating the variable's occurrences number in the formula (*occurrence* subroutine).

2: Performing the procedure *parameters_settings*.

3: generating the propositional array (Propositional create (*Pro_Parallel*) subroutine).

4: applying the slitting algorithm (*slit* subroutine).

5: checking the reduced formulas for their literal number and solution (*local Solvable_Min Selection* routine).

In 1 and 2, the CPU sequentially performs the *occurrence* and *parameters_settings* subroutines. The occurrence subroutine performs two operations; the number of occurrences in each variable in the original formula is counted for each clause and stored in a vector (*occur*), then the variables are sorted according to their appearance.

The sorted variables are transferred to the EVV vector after completion. The *parameters_settings* subroutine as mentioned in Algorithm 2 calculates the amount of factors (number of eliminated variables).

Algorithm 4 Thread Identifications

```
1:  Thread ($Form, Par, EVV, thr_{id}$)
2:  $i = 0, S = false, limit = 0$
3:  while ($i$ != true)
4:      $Pro = Pro\_Parallel$ ($Parameters, EVV, i$)
5:      $C\_S = Slit$ ($r\_Form, Pro$)   // $C\_S$ is Clause Set
6:      for ($c = 0$ to $limit$) do
7:          $Limit += r\_Form(C)$   //$r\_Form$ reduced Form
8:      end for
9:        if ($C\_S \neq false$) then
10:           if ($limit < min$ ||$S = false$) then
11:             $min = limit; S = C\_S; min\_r\_Form = r\_Form$
12:           end if
13:       end if
14:     $r\_Form \leftarrow Form$
15:   End while
16:  $Global(thr_{id}) \leftarrow min\_r\_Form$   //min. reduced formulas
```

Each removed variable produces two more propositionals v and $\bar{v}$, so we should have ($2^{factors}$) new propositionals or variable candidate combinations g, to be removed from the initial formula. Let q be the CPU amount of cores, consequently, we decompose this number g into q tasks; sequentially, each task runs a removal package called ω such that ($\omega = g/q$). The Kit result is a simple, solvable formula that is saved in an array which is a global with a thread ID (thr_{id}) index.

The two logical *threads* are attached to each core, and we generate the p threads using the integrated OpenMP subroutine [61], [42] and tie every thread by its *index*, thr_{id} such that ($0 \leq thr_{id} \leq 2q$) to any subsequent core physique y_i. This is possible with the *OpenMP KMP_AFFINITY* environment variable [62] that established to *scatter* mode. This mode allocates the threads throughout the entire scheme as uniformly as possible. The granularity of the *core* is correspondingly defined so that each thread can migrate to any thread context within a core. Following many tests on multiple SAT CNFs, this setup has demonstrated the highest timing efficiency. A detailed description for the task operations is outlined in process Algorithm 4.

For the subroutines discovered in Fig. 3 steps 3 through 5, we allocate a distinct address space for non-shared of each operative. In step 3, the (*Pro_Parallel)* subroutine is carried out in parallel as what we show in Fig. 3 with different iterators i to generate a binary-like random mixture of variables under the algorithm 5 operation.

The operation is very rapid as it only utilizes a low-memory change using shift and logical AND operations.

After all the steps are over, the *Global* procedure transfers the reduced formulas stored in the global *Solvable_Min* selection phase, in order to determine what formula has minimum literal dimensions and whether or not it is resolvable. This means that a solvable formula is returned with the information stored in *min_r_Form* and *C_S* arrays via the succession of the call to the slit procedure.

Algorithm 5 Propositional Create

```
1:  Pro_Parallel (Parameters, EVV, i)
2:  for (par = 1 to Parameters) do
3:      $shift = 1 \ll (bitshift_{left})\ par$
4:      $fast = i\ \&\ shift$     /fast operation
5:      $pro(k) = (fast\ \ ?\ -EVV(par)\ :\ EVV(par))$
6:  end for
7:  return Pro
```

Algorithm 6 Subsumed Parallelism Where *No_C* Is the No. of Clauses, and *new_C* Is the Next Clause

```
1:  subsumed_kernel (Form, limit, delete, No_C)
2:  C = thread.y;  new_C = thread.x
3:  If ((C && new_C) < No_C) && (~ delete (new_C)
    && (limit(new_C)) > limit(C)))  then
4:      while (par = 1 to limit(C))
5:          literal =  clauses (C * width + par);
6:          if (~ find (Form(new_C), literal, limit(new_C)))
            then
7:              break
8:          end if
9:      end while
10:     if (par == limit(C))  then
11:         delete(new_C) =  true
12:     end if
13: end if
```

In the solvable formula, the last simplified CNF formulation in the SAT settings technique is more important than the other ones.

C. SUBSUMPTION ELIMINATION

To carry out the subsumption trial, a device kernel (as Algorithm 6 for the parallel algorithm and Fig. 4 for a visual insight of the kernel execution) is called by the host which initializes a 2-*D* grid with blocks (set of threads that can be run parallelly) of ceiling length *ceil* (*N/32*) on the *x*-size and *ceil*($N/32$) on the *y*-size side, each block size is 32×32 threads, with a clause in the formula simulated for each thread. So, the SAT Setting procedure results in a linear array of the streamlined input 2-D SAT formula using the *Allocate & Adjust* step as shown in Fig. 2.

In a separate array, the *limit* variable (number of literal *k* in each clause) is stored for description of head and tail of each clause in the mapping array.

The range of clauses has been loaded for use in the global device memory at the assignment point for execution. The other array was loaded into the device's constant memory for quicker reading [63]. A substantial performance was achieved using GPU SMPs (Streaming Multi Processors), using up to 99 percent of its maximum load with the existing configuration as in Algorithm 6.

As discussed earlier, we can reduce the algorithm's complexity to *O(1)* by creating the largest possible threads to cover all the clauses that equal to ($N \times N$). The *delete* array in

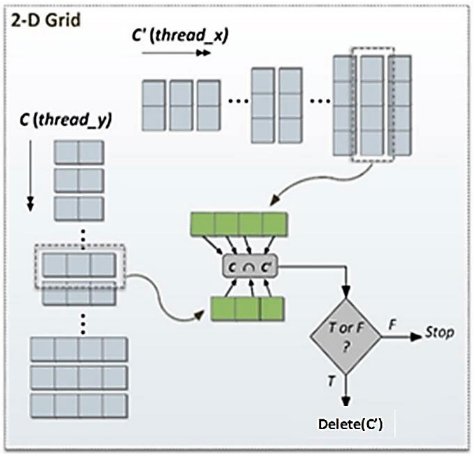

FIGURE 4. A subsumption removal kernel execution.

Algorithm 7 Detector Technique

```
1:  Kernel_detector (Form, Detect, Sum, No_C, No_V,
    Counter)
2:  v = thread.y+ 1, c  = thread.x
3:  while (v  < No_V && c  < No_C)  // No_V is No. of
    variables
4:      P_Var = find (Form(c), v) // P_Var is positive
        variables
5:      N_Var = find (Form(c), ¬v) // N_Var is negative
        variables
6:      clearness  = Ø
7:      Detect (v− 1, c) = false
8:      if (P_Var = N_Var && P_Var = true  then
9:          clearness  = 2
10:     else if (P_Var && ~ N_Var) then
11:         clearness  = 1
12:     else if (~ P_Var && N_Var) then
13:         clearness  = −1
14:     end if
15:     if (clearness ~ = Ø) then
16:         Detect (v − 1, c) = true
17:         AtomicAdd(&Counter(v− 1),  1)
18:         AtomicAdd(&Sum(v− 1),  clearness  )
19:     end if
20:  end if
21: end while
```

the subsumed kernel is initialized to zeros, and its elements represent the clause positions, where the element *x* is set to *1* if a clause *x* need to be removed from the SAT formula.

D. PURE-LITERAL REMOVAL

We implement this type of elimination totally on GPU utilizing two successive kernels dispatches as talked about in section II.D. The first kernel (detector) starts a 2-D grid of blocks with length *ceil(N/32)* in *x*-size and the *ceil(M/32)* in *y*-size, each block size encapsulates *32* × *32* threads,

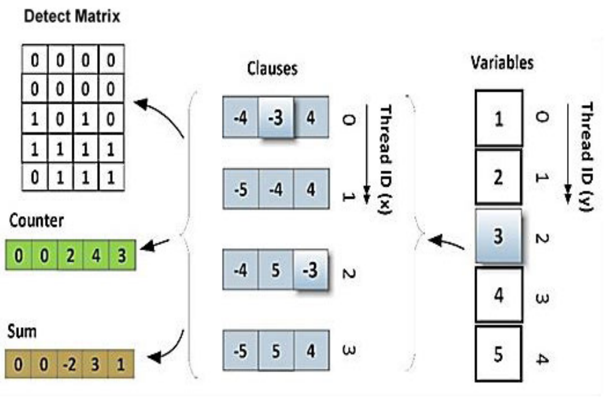

FIGURE 5. A visual model for the detector execution.

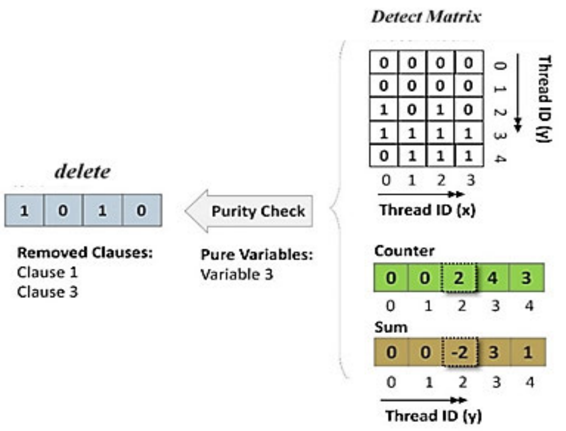

FIGURE 6. A visual model for the implementing execution.

Algorithm 8 Implementer Scheme

```
1: kernel_implementing (Form, Detect, Sum, delete,
     No_C, No_V, Counter)
2: $v = thread.y, c = thread.x$
3: while ($v < No\_V$)
4:     $summing = Sum(v)$, $counting = Counter(v)$
5:     if ($c < No\_C$ && (($summing = counting$)
         || ($-summing = counting$)) && $Detect(v, c)$) then
6:         $Delete = 1$
7:     end if
8: end while
```

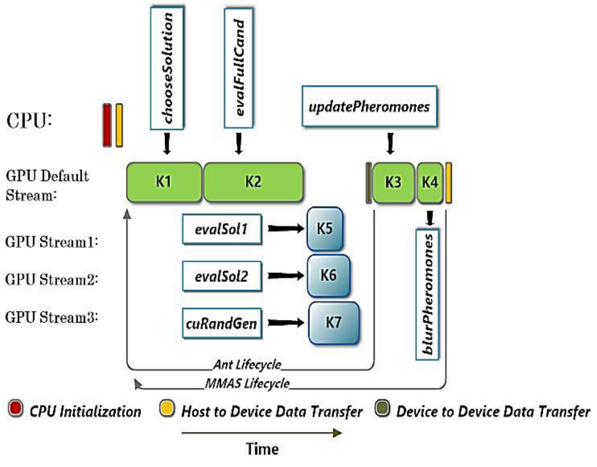

FIGURE 7. GPU usage of MMAS for SAT scheme.

that represents the clauses and parameters amount in every formula's SAT as (*N*,*M*).

The variable *v* is simulated from 1 to *M* by using the *y*-thread. Then *v* is searched in every clause using the *x*-thread, to locate its position, and count its appearance (exclude polarity). The *x*-thread sums up its value whenever it is + ve/−ve (include polarity) using the AtomicAdd operations [64]. i.e., a few cycles of memory lock until the threads complete their procedure.

Algorithm 7 sums up the parallel detector that is in the past kernel. In this algorithm, the detect array is a Boolean matrix with a row indicating the absolute variable value and a column indicating the clause position. The matrix is initialized to zeros and, if a clause *x* contains a variable *y*, the matrix element is set to 1 at index (*y*, *x*). Fig. 5 presents a graphic illustration of the detector technique parallelization.

In the subsequent kernel (*implementing*) proposed in Algorithm 8, we utilize a similar arrangement as the detector kernel starts. Anyway, the *x*-thread checks for the virtue of the variable utilizing the insights we get from the detector portion (*counter*, *Sum*,*Detect*). If the variable is uncorrupted, the SAT formula recognized by the *delete* array removes all the provisions which contain this variable. The *implementing* algorithm provides a visual example of the parallel execution in Fig 6.

E. MMAS PROCEDURE

This part shows our parallel usage of the MMAS Procedure for SAT calculation on the GPU utilizing CUDA (see Fig. 7).

First, we parse the reduced SAT CNF that yielded by the H-SAT preprocessor, then execute each routine on that formula, and as expressed previously, we have three principles arranges in the MMAS system. We are making the counterfeit subterranean ACO framework in the main stage, which is responsible for the accompanying two stages.:

Stage 1: scanning for a competitor answer (*ChooseSolution* Procedure).

Stage 2: Evaluate the elective examination (*evalSolution* Procedure).

The *ChooseSolution* subroutine is run on GPU, which originates a kernel that estimates the probability $p_j(l)$ and looks at an arbitrary number r_l to $p_j(l)$ and bounces a positive task *TRUE*, if $r_l \leq p_j(l)$, and a negative task *FALSE* something else.

The inherent capabilities that performed as Special Function Units (SFUs) inside the GPU system [58] are utilized in the calculation of $p_j(l)$, which needs power and division activities. SFUs can manage transcendental and graphical guidelines for interpolation with a minimum amount of IPCs (Instructions per cycle) but reduces the precision of floating-point numbers. This allows us to cut down the quantity of floating and integer instructions to 84% and 54% respectively, that lower the execution period by a factor of 2.68 for the *ChooseSolution* subroutine [46].

An irregular number sequencer r_n of consistently dispersed floating-point esteems anywhere in the range of 0.0 and 1.0 (where, 0.0 is rejected) is at first created at the beginning of MMAS run utilizing *curandGenerateUniform* work in the *cuRAND* library. In this manner, we create new groupings for the subsequent ants in the territory covering the scattering work with others GPU parts as being shown in Fig. 5 to get the improvement of the simultaneous bit execution ability in the GPU. The competitive execution of several kernels can withstand 32 kernels if various streams are assigned.

Our use of *evalSolution* includes 3 kernels, *as GPU streams1,2, and 3.*

1) The kernel *evalFullCand* Initializes a 2-D range block grid of $ceil(m/32)$ in x-axis and $ceil(n/32)$ in y-axis; every block size is (32×32) threads, where apiece thread mimics a literal. The formulas of SAT were a table of clauses that being complete [13], where a table is a q matrix of $n \times m$ elements as $q \in \mathcal{B}\hat{}(n \times m \times 2))$, where $\mathcal{B}$ is a boolean bit. The candidate for the solution is displayed in an array $d \in \mathcal{B}^{1 \times m}$.

We calculate procedure $q_{ij} \bigwedge d_j$ bit by bit for each clause such that $1 \leq i \leq n$ and $1 \leq j \leq m$ to test the applicant d, where d fulfills q iff each matrix $u_i^{2 \times m}$ arising from the previous procedure includes at smallest one *TRUE*. q is stacked in the initialization period of the common GPU memory (as previously stated) and d has been stacked into the shared memory during the execution of the kernel to gain from a regular access to data.

2, 3) The kernels *evalSol1* and *evalSol2* calculate the sum of the clauses that have been resolved as $s \in \mathcal{B} \wedge (1 \times n))$, where $s_i = \bigvee_i u_i$ and the evaluation quality (E) according to the succeeding calculations:

$$E \leftarrow \sum\nolimits_{i=1}^{k} s_i w_i \tag{4}$$

$$S \leftarrow \sum\nolimits_{i=1}^{k} s_i \tag{5}$$

where w_i is the clause capacity, $1 \leq i \leq k_i$, and k is the number of the satisfiable clauses.

The summation in equalizations 4 and 5 is measured utilizing the parallel decrease calculation as Harris [65]. Brent's theorem [66] says that each thread should add $O\,(logN)$ components to the shared memory, and then the tree-based reduction system [67] will be applied to the shared memory. We have adjusted the scheme to include the last partials sums consequence with the atomic-add activity (see Fig. 8) to help the dot product in the form (4) safeguard the time complexity of the scheme to $O(N/logN)$.

In the subsequent stage, the *updatePheromones* subroutine updates all pheromones focus τ_l and points of confinement (τ_{min}, τ_{max}) as indicated by the best elective found

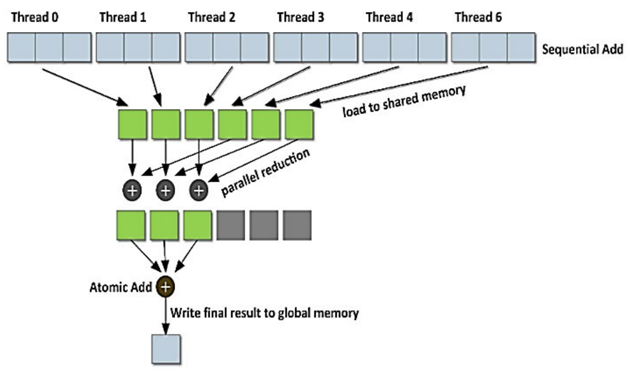

FIGURE 8. **Atomic add process reducing scheme.**

based on colony of ants and of the highest quality appraisal techniques.

This subroutine dispatches a GPU kernel with enough threads comparable to the m variables. There are 512 threads in a block, where each thread is mapped to a trail of pheromones to vanishing and deposit forms to restrain the pheromones limits.

Finally, the *blurPheromones* subroutine foggy spots all pheromones τ_l by adding the worth $r_l.ph_l$ to every pheromone amount, where r_l is an irregular number with the end goal that $-max_i \leq r_l \leq max_i$, where max_i is called the most extreme difference parameter and is determined [25] as follows:

$$max_i \leftarrow \mu.e^{-\frac{i}{\sigma}} \tag{6}$$

where μ is the base blurring and σ is the factor of decline. This strategy has given phenomenal outcomes in choosing arrangements competitors just with the obscuring in periodic cycles. The procedure of *blurPheromones* begins a kernel comparable to the previous *updatePheromones* kernel, but the method of evaporation and depositing is substituted by the blurring method.

IV. PERFORMANCE BENCHMARKING

In this section, on countless random SAT CNFs, we conduct the benchmarks acquired by executing our preprocessor and ACO solver implementations. We compare these benchmarks to the sequential implementation of our algorithm. The benchmarks introduced in this paper includes the following criteria for efficiency:

1. The running times of our H-SAT preprocessor parallel implementation against serial implementation.
2. The execution times of our parallel implementation of ACO SAT against serial deployment.
3. Acceleration acquired towards the serial equivalents.
4. Statistics on cuts and percentages of the literal and the clauses compared to the initial SAT cases.
5. Comparison of time with and without our preprocessing H-SAT.
6. Comparison of the quality for the solution with and without our preprocessing H-SAT.

TABLE 1. The proposed sequential and parallel implementations execution times (ms).

			H-SAT Pre-processor								
S	Set	Clauses	T_S (ms)			T_P (ms)			Speed_up		
			V_E	S_E	PLR	V_E	S_E	PLR	V_E	S_E	PLR
1	4_SAT_100	200: 400	998	0.18	0.55	368.6	0.11	0.14	2.7x	2x	3.8x
2	4_SAT_100	450:650	1300	0. 52	0.69	452. 54	0.11	0. 16	2.8x	4.7x	4.3x
3	3_SAT_200	400: 480	446.5	0.24	1.83	161. 4	0.08	0. 22	2.7x	3x	8.3x
4	3_SAT_200	500: 580	590	0.40	2.3	229	0.09	0. 24	2. 5x	3.9x	8.8x
5	4_SAT_200	400:600	785	0. 36	1.86	287. 9	0.1	0. 29	2.7x	3.6x	6.4x
6	3_SAT_300	600:680	888	0. 42	4.24	355.14	0.12	0. 36	2. 5x	3.5x	11. 8x
7	3_SAT_300	700: 780	1368	0. 56	4.41	510.75	0.14	0. 38	2.7x	4x	11.6x
8	4_SAT_300	600:680	446.4	0. 42	4.2	167. 9	0.14	0. 46	2.6x	3x	9.2x
9	4_SAT_300	600:1000	264.9	0.81	4.05	96. 41	0. 22	0. 78	2.7x	3.7x	5.2x
10	4_SAT_350	700:1100	3445	1.28	5.62	1113	0.27	0. 98	3.2x	4.8x	5.7x
11	3_SAT_400	600: 680	279	0.40	7.00	121	0.15	0. 47	2.3x	2.6x	14.8x
12	3_SAT_400	700: 780	104.6	0.37	7.64	45.88	0.16	0. 53	2.3x	2.3x	14.4x
13	4_SAT_400	800:1200	1058	1.20	7.30	340.00	0. 3	1. 27	3.2x	3.9x	5.7x
14	3_SAT_500	700: 780	326.6	0. 33	10.93	123.12	0.18	1. 08	2.7x	2.1x	10.2x

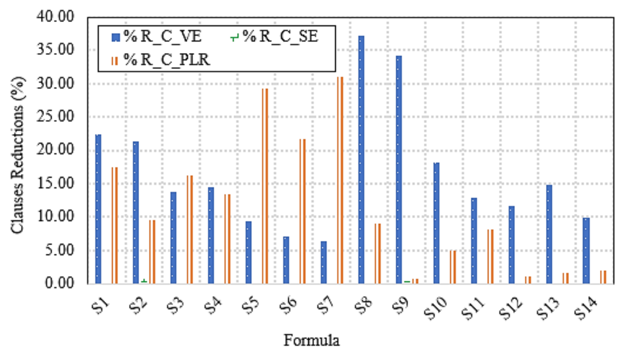

FIGURE 9. The average reductions in clauses in H-SAT pre-processor vs. different CNF sets.

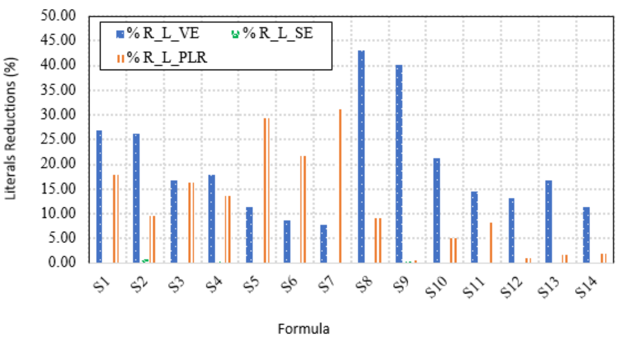

FIGURE 10. The average cuts in our H-SAT preprocessor literals against different CNF sets.

We have implemented the SAT Factoring in our H-SAT preprocessor using OpenMP V2.0 with C++ backed by Microsoft Visual Studio compiler (VS2013) running on Intel Core i7 3770 K with 4 cores and 4 threads (one thread/core) running at 3.9 GHz and 8 GB memory. Instructions set for the Advanced Vector Extensions (AVX) [68] is subjugated to boost host execution cycle throughout the host-side program execution phase. Intel AVX is a 256-bit extension to the Intel Streaming SIMD Extensions (SSE) set of guidelines and is specifically intended to enhance intensive data applications efficiency due to bigger vectors, fresh extendable syntax and

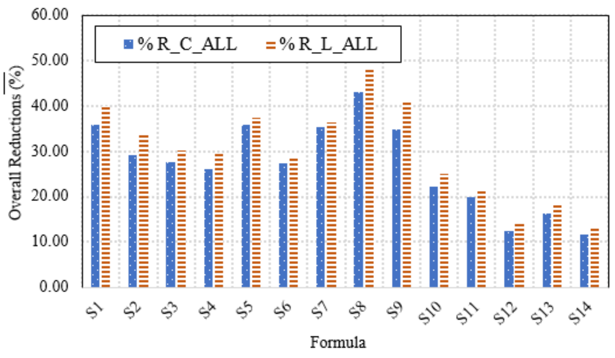

FIGURE 11. The average reduction after all simplification of clauses and literal.

TABLE 2. H-SAT pre-processing power dependent on ACO solution.

	H-SAT ACO Program to Solve				
	#Satisfying Formulae		Solve the problem (# Iterations)		
S	None Pre-processing	Pre-processing	None Pre-processing	Pre-processing	Percentage of patch
1	0	5	*Excess*	1760.98	100%
2	0	5	*Excess*	4254.82	100%
3	0	3	*Excess*	4626.79	100%
4	0	5	*Excess*	3581.78	100%
5	0	5	*Excess*	5134.65	100%
6	3	5	5850	7432	43%
7	3	4	7411	9311	24%
8	5	5	682	600	13%
9	2	5	1751	1180	60%
10	0	5	*Excess*	660	100%
11	5	5	57	63	6%
12	2	5	120	1371	63%
13	3	5	44	130	43%
14	3	5	7530	9811	44%

rich characteristics. Our subsumption in aggregation with pure-literal removals along with the MMAS SAT solver that programmed utilizing CUDA C++ running with 2880 processing cores on NVIDIA Geforce GTX (15 multiprocessors with 192 processing cores each) operating at 1 GHz and 6 GB of memory [69].

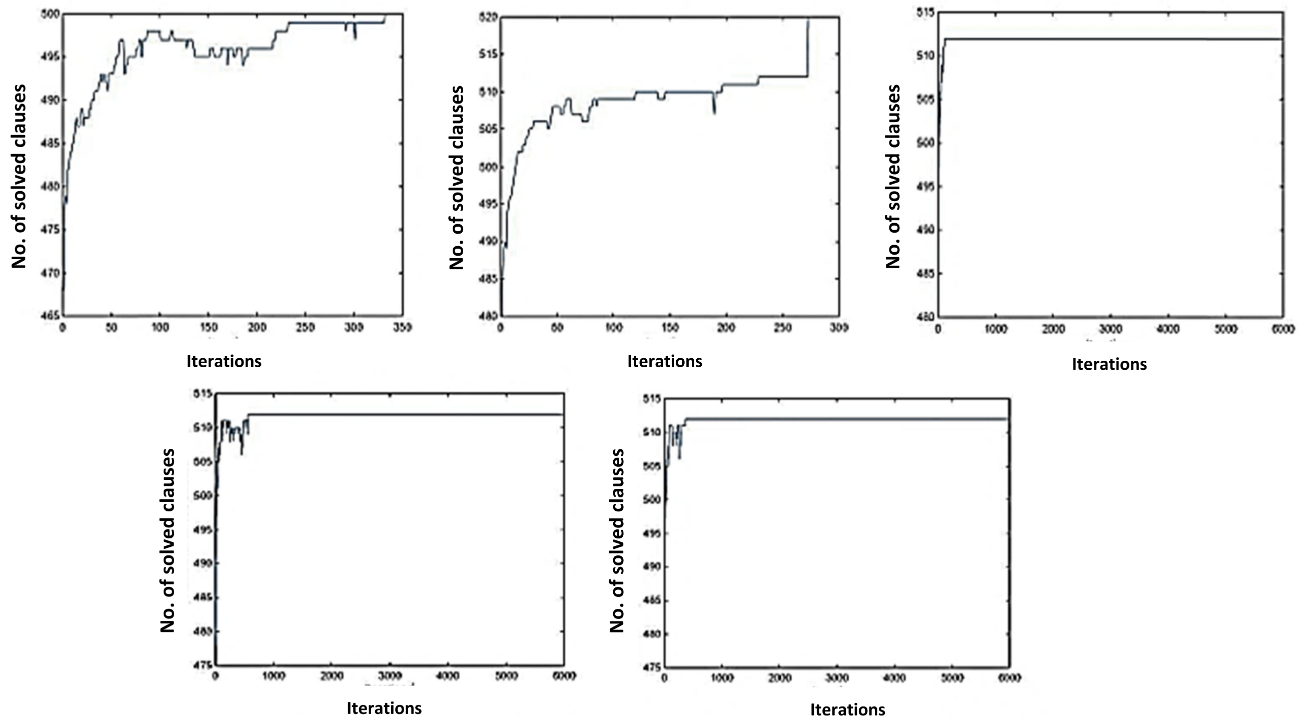

FIGURE 12. Solution performance convergence without the use of the H SAT pre processor.

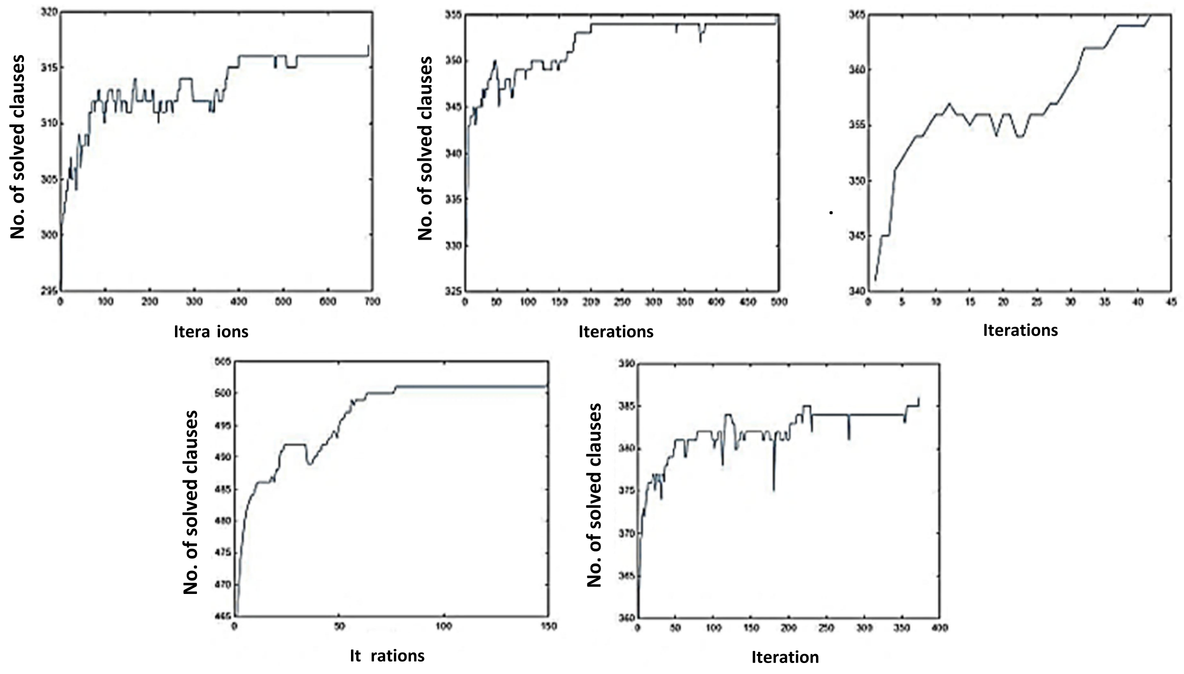

FIGURE 13. Solution performance convergence with the use of the H-SAT pre-processor.

NVIDIA SDK is used on Windows 10 × 64 with CUDA Tool Kit v6.5. CUDA Compute Capability 3.5 optimized the GPU binary code [63] to take the advantage of the NVIDIA's Kepler GK110, which is the architecture for the next generation GPU [70]. The executable sequential code was introduced using C++ and performed

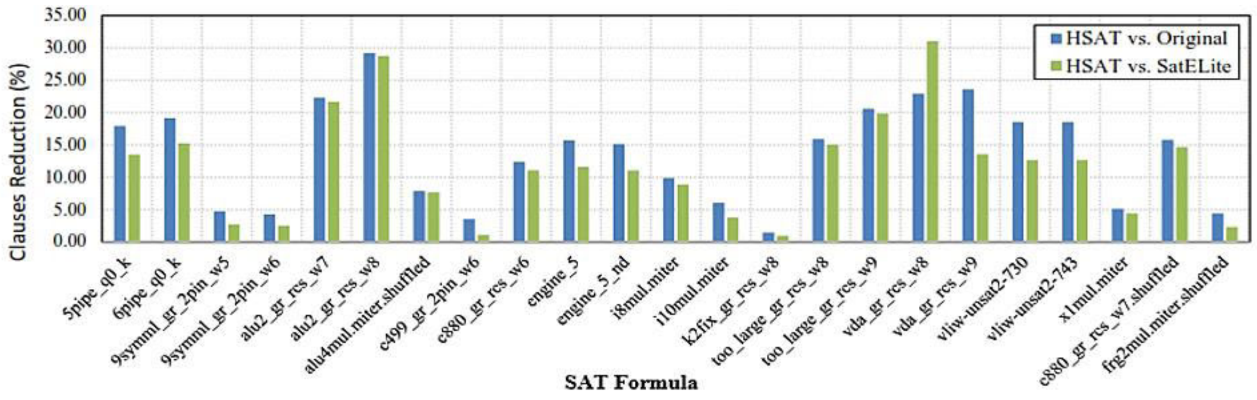

FIGURE 14. The reductions proportion of our H-SAT clauses to different SAT CNFs of set 4 (S4) benchmark.

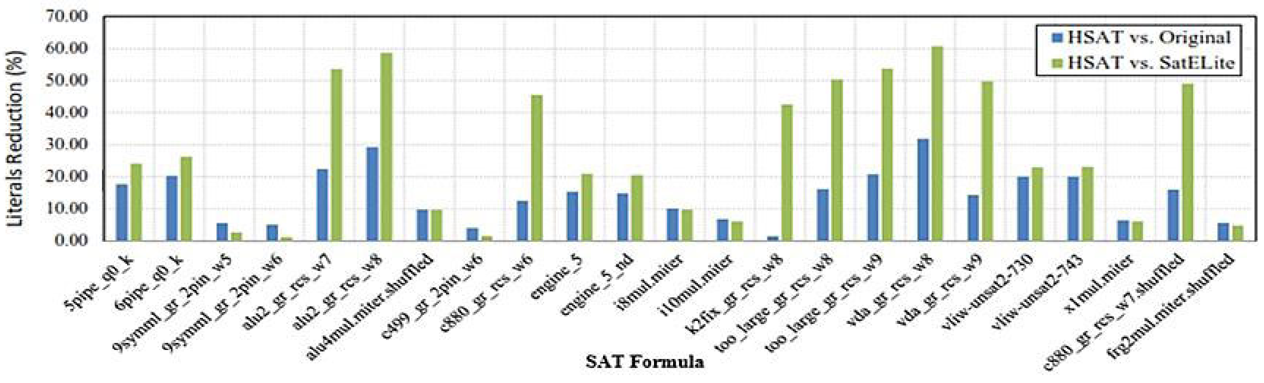

FIGURE 15. The reductions proportion of our H SAT literals to different SAT CNFs of set 4 (S4).

on a single core operating at 3.9 GHz with a single thread.

We have generated 14 benchmark formulas sets of irregular k-SAT (5 each) in DIMACS arrangement utilizing ToughSAT library [71]. Each set name comprises of the SAT sort pursued by the number of factors in this set. For formula, the benchmark set 5_sat_600 implies that the SAT sort is 5 SAT or 5 CNF and the quantity of factors in each occurrence is 600 factors/variables.

Within 30 distinct runs, all solving times of k-SAT sets are averaged over 6,000 iterations in the algorithm. We used three kinds of CUDA generators in the GPU application of the ACO SAT solver, PSEUDO_MTGP32, PSEUDO_XORWOW and PSEUDO_MRG32K3A [59]; the average timing for each generator is over 10 distinct runs, that is, 30 runs in total.

Table 1 presents the SAT parameters (Variable Elimination or *VE*) with a *max_nop* set to 12, the Subsumption Elimination (*SE*) kernel, and the parallel Pure-Literal Removal (*PLR*) algorithm. Furthermore, the speed-ups achieved against the sequential implementation of our methods are also provided for the chosen CNF cases.

Table 1 findings reveal a velocity up to 4.75x quicker than the sequential counterpart in the parallel execution of kernel deletion subsumption in the GPU and acceleration in parallel execution of CPU of the factor 3.12x elimination variable. Relatively velocity up to 15x demonstrates our fresh parallel algorithm of pure-literal removal.

In Figures 9, 10 The percentages of clause and literal reductions observed for each of our H-SAT preprocessor's simplification methods that appropriately are shown vs. the initial formula. In Fig. 11, We observe a large part of the complete cuts owing to all the simplifications, up to 43% and 48.5% in the number of clauses and literals respectively. Note that the prefix (R_C) concerns the reduction of the clause in a set, with the prefix (R_L) concerns the literal reduction.

Table 2 demonstrates our solver's solving times with and without pre-processing H-SAT. The results show that in the case of using the preprocessor actions in solving big SAT cases, the ACO solver is so efficient with a small amount of time. In Table 2, the notation (*Excess*) implies that the solver surpassed the maximum amount of iterations or tests attempting to fix the SAT equations in a set.

Fig. 12, represents the H-SAT solver's solution characteristics without using the preprocessor in S2 set. Similarly, in Fig. 13, we demonstrate the H-SAT solver solution characteristics using the same set of preprocessors. We observe in the graphs (3, 4 and 5) in Fig.13 a quick convergence to the solution over the ones found in Fig. 12.

V. EXPERIMENTAL RESULTS OF REDUCTIONS

The SAT, as a generic problem has many solutions, where the authors used many platforms to solve and to speed up. So, some libraries as SATLIB are used as test banks to get some of the benchmarks for their solutions. Also, present-day SAT solvers are exceptionally subject to heuristics. Consequently, benchmarking is of prime significance in assessing the exhibitions of various solvers. In any case, applicable benchmarking isn't direct.

Figures 14, 15 indicate the percentages of actual H-SAT and clause reductions compared to the initial formulae and thus acquired by the SatELite. The figures show that the preprocessor H-SAT defeats the SatELite by 13.5% and 24% respectively in removing the additional amount of literals and clauses, even quicker by 4.41. H-SAT also accomplished more clause and literal cuts than those acquired through the SatELite by 31% and 60.7%. The average decreases in H-SAT provisions and literals compared to the initial cases are respectively 13.7 percent and 14.2 percent.

VI. CONCLUSION

In this research paper, we performed a fresh effective parallel SAT heuristic solver with parallel preprocessor on heterogeneous CPU-GPU systems. We have developed a variable elimination technique that executes the SAT factoring exhausting the most constrained variables of a SAT formula on the CPU's multicore architecture.

We have shown how the subsumption and pure-literal eliminations implemented in our H-SAT preprocessor can benefit from the massive-data parallelism of the GPU CUDA platform. Our benchmarks divulge an increase in speed-up to 15x more quickly than the sequential implementation and significant reductions of 43% and 49% in clauses and literal respectively, compared with the initial CNF.

With the help of our parallel preprocessor, we have shown how a metaheuristic SAT solution can be so efficient in solving broad CNF formulae. In our implemented task, we have considered numerous highlights of the host and gadget SIMT designs, for example, the multicores working with strong CPU frequencies, shared memory, and simultaneous part execution.

Also, they work with atomic activities, two-dimensional grid with a large number of threads that run in parallel, and quick steady memory. Likewise, we have exploited the AVX guidance built-in CPU's sets and GPU's SFUs to produce an effective and quick executable code.

ACKNOWLEDGMENT

The authors gratefully wish to acknowledge the NVIDIA Corporation for supporting our research accompanied in this paper with the donation of the Titan Black GPU.

REFERENCES

[1] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 10, pp. 1606–1621, Oct. 2005.
[2] A. Q. Dao, M. P.-H. Lin, and A. Mishchenko, "SAT-based fault equivalence checking in functional safety verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 12, pp. 3198–3205, Dec. 2018.
[3] A. Diedrich, A. Maier, and O. Niggemann, "Model-based diagnosis of hybrid systems using satisfiability modulo theory," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, 2019, pp. 1452–1459.
[4] L. Lingappan and N. K. Jha, "Satisfiability-based automatic test program generation and design for testability for microprocessors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 5, pp. 518–530, May 2007.
[5] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of SAT solutions for testing," in *Proc. 40th Int. Conf. Softw. Eng.*, May 2018, pp. 549–559.
[6] I. A. Lyapunova and N. A. Fomenko, "Modification of ALL–SAT solver to search verification kits in testing," *IOP Conf. Ser., Mater. Sci. Eng.*, vol. 537, no. 5, 2019, Art. no. 052029.
[7] S. Andrei, W. N. Chin, A. M. K. Cheng, and M. Lupu, "Automatic debugging of real-time systems based on incremental satisfiability counting," *IEEE Trans. Comput.*, vol. 55, no. 7, pp. 830–842, Jul. 2006.
[8] I. Lynce and J. Marques-Silva, "Efficient haplotype inference with Boolean satisfiability," in *Proc. Nat. Conf. Artif. Intell.*, 2006, p. 104.
[9] E. Dubrova and M. Teslenko, "An efficient SAT-based algorithm for finding short cycles in cryptographic algorithms," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, Apr. 2018, pp. 65–72.
[10] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel Core i7 processor execution engine validation," presented at the 21st Int. Conf. Comput. Aided verification, Grenoble, France, 2009, pp. 414–429.
[11] S. Khurshid and D. Marinov, "TestEra: Specification-based testing of Java programs using SAT," *Automated Softw. Eng.*, vol. 11, no. 4, pp. 403–434, Oct. 2004.
[12] S. Keshavarz, F. Schellenberg, B. Richte, C. Paar, and D. Holcomb, "SAT-based reverse engineering of gate-level schematics using fault injection and probing," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, Apr. 2018, pp. 215–220.
[13] K. Juretus and I. Savidis, "Importance of multi-parameter SAT attack exploration for integrated circuit security," in *Proc. IEEE Asia Pacific Conf. Circuits Syst. (APCCAS)*, Oct. 2018, pp. 366–369.
[14] M. Soos and K. S. Meel, "Bird: Engineering an efficient CNF-XOR sat solver and its applications to approximate model counting," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, 2019, pp. 1592–1599.
[15] S. Huang, Y. Li, and Y. Li, "An SVM-based prediction method for solving SAT problems," *Chin. J. Electron.*, vol. 28, no. 2, pp. 246–252, Mar. 2019.
[16] X. Wang, Q. Zhou, Y. Cai, and G. Qu, "Parallelizing SAT-based decamouflaging attacks by circuit partitioning and conflict avoiding," *Integration*, vol. 67, pp. 108–120, Jul. 2019.
[17] G. Cabodi, P. E. Camurati, M. Palena, P. Pasini, and D. Vendraminetto, "Reducing interpolant circuit size through SAT-based weakening," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, May 7, 2019, doi: 10.1109/TCAD.2019.2915317.
[18] A. Biere and D. Kröning, "SAT-based model checking," in *Handbook of Model Checking*. Cham, Switzerland: Springer, 2018, pp. 277–303.
[19] S. Subbarayan and D. K. Pradhan, "NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances," in *Theory and Applications of Satisfiability Testing*. Berlin, Germany: Springer, 2005, pp. 276–291,
[20] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Proc. Int. Conf. Theory Appl. Satisfiability Test.*, 2005, pp. 61–75.
[21] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability* (Frontiers in Artificial Intelligence and Applications), vol. 185. Amsterdam, The Netherlands: IOS Press, 2009.
[22] O. Gableske and M. H. Heule, "EagleUP: Solving random 3-SAT using SLS with unit propagation," in *Theory and Applications of Satisfiability Testing—SAT*, vol. 6695, K. Sakallah and L. Simon, Eds. Berlin, Germany: Springer, 2011, pp. 367–368.

[23] A. Belov, A. Morgado, and J. Marques-Silva, "SAT-based preprocessing for MaxSAT," in *Logic for Programming, Artificial Intelligence, and Reasoning*, vol. 8312. K. McMillan, A. Middeldorp, and A. Voronkov, Eds. Berlin, Germany: Springer, 2013, pp. 96–111.
[24] J. Johannsen, "The complexity of pure literal elimination," in *SAT*. Cham, Switzerland: Springer, 2005, pp. 89–95.
[25] D. Moritz and M. Springer. (2010). *Solving Satisfiability With Ant Colony Optimization and Genetic Algorithms*. [Online]. Available: http://matthiasspringer.de
[26] Y. Hamadi, S. Jabbour, and L. Sais, "ManySAT: A parallel SAT solver," *J. Satisfiability, Boolean Model. Comput.*, vol. 6, no. 4, pp. 245–262, Jun. 2009.
[27] L. Gil, P. Flores, and L. M. Silveira, "PMSat: A parallel version of MiniSAT," *J. Satisfiability, Boolean Model. Comput.*, vol. 6, pp. 71–98, 2008.
[28] Q. Meyer, F. Schönfeld, M. Stamminger, and R. Wanka, "3-SAT on CUDA: Towards a massively parallel SAT solver," in *Proc. Int. Conf. High Perform. Comput. Simulation*, Jun. 2010, pp. 306–313.
[29] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "Modular and efficient divide-and-conquer SAT solver on top of the painless framework," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Cham, Switzerland: Springer, 2019, pp. 135–151.
[30] C. Izu and E. Vallejo, "Impact of network fairness on the performance of parallel systems," in *Proc. Australas. Comput. Sci. Week Multiconference (ACSW)*, 2019, p. 11.
[31] S. Beckers, "Parallel SAT-solving with OpenCL," in *Proc. IADIS Int. Conf. Appl. Comput.*, Rio de Janeiro, Brazil, 2011, pp. 435–441.
[32] Z. Newsham, V. Ganesh, and S. Fischmeister, "Predicting SAT solver performance on heterogeneous hardware," *Proc. Pragmatics SAT*, vol. 59, pp. 18–33, Mar. 2019.
[33] D. Selsam and N. Bjørner, "Guiding high-performance SAT solvers with unsat-core predictions," in *Proc. Int. Conf. Theory Appl. Satisfiability Test.* Cham, Switzerland: Springer, 2019, pp. 336–353.
[34] R. Hickey and F. Bacchus, "Speeding up assumption-based SAT," in *Proc. Int. Conf. Theory Appl. Satisfiability Test.* Cham, Switzerland: Springer, 2019, pp. 164–182.
[35] A. Niewiadomski, P. Switalski, T. Sidoruk, and W. Penczek, "Applying modern SAT-solvers to solving hard problems," *Fundamenta Informaticae*, vol. 165, nos. 3–4, pp. 321–344, Mar. 2019.
[36] M. Dorigo, E. Bonabeau, and G. Theraulaz, "Ant algorithms and stigmergy," *Future Gener. Comput. Syst.*, vol. 16, no. 8, pp. 851–871, Jun. 2000.
[37] T. Stützle and M. Dorigo, "ACO algorithms for the traveling salesman problem," *Evol. Algorithms Eng. Comput. Sci.*, vol. 4, pp. 163–183, Jun. 1999.
[38] P. S. E. Wong, R. Ku, and P. Xie, "Ant colony optimization," Univ. California, Los Angeles, CA, USA, Tech. Rep., Spring 2011.
[39] J. Luan, Z. Yao, F. Zhao, and X. Song, "A novel method to solve supplier selection problem: Hybrid algorithm of genetic algorithm and ant colony optimization," *Math. Comput. Simul.*, vol. 156, pp. 294–309, Feb. 2019.
[40] D. B. Kirk and W. H. Wen-Mei, *Programming Massively Parallel Processors: A Hands-on Approach*. Oxford, U.K.: Newnes, 2012.
[41] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Reading, MA, USA: Addison-Wesley, 2010.
[42] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 3, pp. 404–418, Mar. 2009.
[43] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.
[44] T. Stützle and H. H. Hoos, "MAX-MIN ant system," *Future Generat. Comput. Syst.*, vol. 16, no. 8, pp. 889–914, Jun. 2000.
[45] M. Villagra and B. Barán, "Ant colony optimization with adaptive fitness function for satisfiability testing," in *Logic, Language, Information and Computation*. Cham, Switzerland: Springer, 2007, pp. 352–361.
[46] H. Youness, A. Ibraheim, M. Moness, and M. Osama, "An efficient implementation of ant colony optimization on GPU for the satisfiability problem," in *Proc. 23rd Euromicro Int. Conf. Parallel, Distrib., Network-Based Process.*, Mar. 2015, pp. 230–235.
[47] Z. Ai, C. Fan, Y. Zhang, H. Rong, Z. Tian, and H. Fu, "Boundary evolution algorithm for SAT-NP," 2018, *arXiv:1903.01894*. [Online]. Available: http://arxiv.org/abs/1903.01894
[48] L. Zhang, "On subsumption removal and on-the-fly CNF simplification," in *Theory and Applications of Satisfiability Testing*. 2005, pp. 482–489.
[49] Rutgers. (2014). *Discrete Mathematics and Theoretical Computer Science*. [Online]. Available: http://dimacs.rutgers.edu/
[50] H. V. Maaren and J. Franco. (2014). *The International SAT Competitions*. [Online]. Available: http://www.satcompetition.org/
[51] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *IEEE/ACM Int. Conf. Comput. Aided Design. (ICCAD) Dig. Tech. Papers*, Nov. 2001, pp. 279–285.
[52] J. Elffers, J. Johannsen, M. Lauria, T. Magnard, J. Nordström, and M. Vinyals, "Trade-offs between time and memory in a tighter model of CDCL SAT solvers," in *Proc. Int. Conf. Theory Appl. Satisfiability Test.* Cham, Switzerland: Springer, 2016, pp. 160–176.
[53] J. Marques-Silva and S. Malik, "Propositional SAT solving," in *Handbook of Model Checking*. Cham, Switzerland: Springer, 2018, pp. 247–275.
[54] O. Bailleux, "DPLL with restarts linearly simulates CDCL," Tech. Rep., 2019.
[55] J. Elffers, J. Giráldez-Cru, S. Gocht, J. Nordström, and L. Simon, "Seeking practical CDCL insights from theoretical SAT benchmarks," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, Jul. 2018, pp. 1300–1308.
[56] A. Voronkov, "Satisfiability and theories," in *Proc. 11th Int. Symp. Symbolic Numeric Algorithms Sci. Comput.*, Sep. 2009, p. 16.
[57] L. Yin, F. He, W. N. N. Hung, X. Song, and M. Gu, "Maxterm covering for satisfiability," *IEEE Trans. Comput.*, vol. 61, no. 3, pp. 420–426, Mar. 2012.
[58] NVIDIA. (Sep. 2019). *CUDA C Programming Guide*. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/
[59] NVIDIA. (Sep. 2019). *CURAND*. [Online]. Available: http://docs.nvidia.com/cuda/curand/
[60] CPlusPlus. (Sep. 2019). *STL STD::Vector*. [Online]. Available: http://www.cplusplus.com/reference/vector/vector/
[61] MSDN. (Sep. 2019). *OpenMP in Visual C++*. [Online]. Available: https://msdn.microsoft.com/en-us/library/tt15eb9t.aspx
[62] INTEL. (Sep. 2019). *Thread Affinity Interface (Linux* and Windows*)*. [Online]. Available: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-thread-affinity-interface-linux-and-windows
[63] NVIDIA. (Sep.) *CUDA C Programming Guide*. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/
[64] (Sep. 2019). [Online]. Available: http://supercomputingblog.com/cuda/cuda-tutorial-4-atomic-operations/
[65] M. Harris. (Sep. 2019). *Optimizing Parallel Reduction in CUDA*. [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
[66] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *J. ACM*, vol. 21, no. 2, pp. 201–206, Apr. 1974.
[67] F. N. Abu-Khzam, D. Kim, M. Perry, K. Wang, and P. Shaw, "Accelerating vertex cover optimization on a GPU architecture," in *Proc. 18th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2018, pp. 616–625.
[68] (2019). *Intel*. [Online]. Available: https://software.intel.com/en-us/isa-extensions
[69] NVIDIA. (Sep. 2019). *NVIDIA GeForce GTX Titan Black*. [Online]. Available: https://www.nvidia.com/gtx-700-graphics-cards/gtx-titan-black/
[70] NVIDIA. (Sep. 2019). *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. [Online]. Available: https://www.microway.com/file/nvidia_kepler_gk110_gk210_architecture_whitepaper-pdf/
[71] H. Yuen and J. Bebel. (2019). Tough SAT project. ToughSAT. [Online]. Available: https://toughsat.appspot.com/

HASSAN YOUNESS received the B.Sc. and M.Sc. degrees from Assiut University, Asyut, Egypt, and the Ph.D. degree from the Graduate School of Information Science and Technology, Osaka University, Japan, with the cooperation of Ain Shams University, Egypt. He worked for IBM Company and Mentor Graphics in Egypt. He is currently an Associate Professor with Minia University, and the Chairman of the Computers and Systems Engineering Department. His research interests include integrated system design, fault tolerance, HW/SW co-design, parallel computers, embedded systems, GPGPU, APU, MPSoCs, homogeneous/heterogeneous systems, machine learning, deep learning, and AI.

MUHAMMAD OSAMA received the B.Sc. and M.Sc. degrees in computer engineering from Minia University, Egypt, in 2011 and 2015, respectively. He is currently pursuing the Ph.D. degree with the Eindhoven University of Technology (TU/e), Eindhoven, The Netherlands. In 2011, he joined the Department of Computer and Systems Engineering, Minia University, as a Teaching Assistant. He has been an Assistant Lecturer, since 2016, and a Principal Investigator with the NVIDIA Educational GPU-Computing Laboratory, Minia University, since 2014. His research interests include real-time embedded systems, parallel programming, high performance computing, formal verification, and SAT solving.

AZIZA HUSSEIN received the B.Sc. and M.Sc. degrees from Assiut University, Egypt, in 1983 and 1989, respectively, and the Ph.D. degree in electrical and computer engineering from Kansas State University, USA, in 2001. She joined Effat University, Saudi Arabia. She was the Head of the Electrical and Computer Engineering Department, Effat University, from 2007 to 2010. She was the Head of the Computer and Systems Engineering Department, Faculty of Engineering, Minia University, Egypt, from 2011 to 2016. Her research interests include microelectronics, analog/digital VLSI system design, RF circuit design, high-speed analog-to-digital converters design, and wireless communications.

MOHAMMED MONESS (Member, IEEE) received the B.Sc. degree (Hons.) in electrical engineering and the M.Sc. degree in electronics and communication engineering from Assiut University, Asyut, Egypt, in 1971 and 1975, respectively, and the Ph.D. degree in control engineering from the Budapest University of Technology and Economics (BME), Budapest, Hungary, in 1983. From 1975 to 1985, he was a Lecturer and an Assistant Professor with the Department of Electrical Engineering, Assiut University. In 1985, he joined the University of Minia, Minya, Egypt, where he was an Associate Professor and a Professor of systems and control engineering. From 1995 to 2008, he was the Chairman of the Department of Computers and Systems Engineering, and the Vice Dean and the Dean of the Faculty of Engineering, Minia University. His research interests include multivariable systems, evolutionary algorithms, computational intelligence, and embedded systems.

AMMAR MOSTAFA HASSAN received the B.Sc. and M.Sc. degrees from Assiut University, Asyut, Egypt, in 1995 and 2005, respectively, and the Ph.D. degree from Minia University, Minya, Egypt, in 2012. He spent two years at Otto-von-Guericke University, Magdeburg, Germany. He is currently the Head of the Computer Science Department, Arab Academy for Science, Technology, and Maritime Transport, South Valley Branch. His research interests include image processing, image hashing, cryptography, automatic/digital control, and image authentication.