Parallel hybrid SAT solving using OpenCL

Sander Beckers [a b]   Gorik De Samblanx[a]   Floris De Smedt [a]

Toon Goedemé [a]   Lars Struyf [a]   Joost Vennekens [a b]

[a] *EAVISE, Lessius - Campus De Nayer, Sint-Katelijne-Waver, Belgium*
[b] *CS-DTAI, KULeuven, Leuven, Belgium*

Abstract

In the last few decades there have been substantial improvements in approaches for solving the Boolean satisfiability problem. Many of these consisted in elaborating on existing algorithms, both on the side of complete solvers as in the area of incomplete solvers. Besides the improvements to existing solving methods, however, recent evolutions in SAT solving take the form of combining several solvers into one, resulting in parallel solvers and so-called hybrid solvers. Our goal is to combine both approaches, by presenting a parallel hybrid solver. The parallelism exists on two levels: we run a complete solver on the CPU concurrently with an incomplete solver on the GPU, where the latter in turn consists of a massively parallel local search algorithm. We implemented our approach using the OpenCL framework, and present preliminary experimental results.

1 Introduction

The last few decades have seen an enormous amount of progress on solving the Boolean satisfiability problem. Much of the progress in this area was due to elaborations of existing algorithms. On the side of the complete solvers this led to more efficient branching heuristics, the use of watched literals for unit propagation and conflict driven clause learning; incomplete solvers on the other hand have benefited from using random walks and from integrating evolutionary algorithms into the search process. Besides the improvements to existing solving methods, however, many of the evolutions in SAT-solving over the last decade resulted from coming up with ways to combine several solvers into one. These new type of solvers are based on the more traditional ones, but add a novel element by making several of them cooperate (or compete, in the case of portfolio strategies as in [5, 9]).

We can, roughly speaking, separate these cooperative approaches into two classes. On the one hand we have hybrid solvers, the motivation for which comes from attempts to respond to one of the Ten Challenges in Propositional Reasoning and Search posed by Selman et al. [20], namely

> CHALLENGE 7 : Demonstrate the succesful combination of stochastic search and systematic search techniques, by the creation of a new algorithm that outperforms the best previous examples of both approaches.

The idea is that if we succeed in combining ideas from complete and incomplete methods, the deficiencies of the one method are complemented by the advantages of the other in order to obtain the best of both worlds.

On the other hand, with gains in processor speeds becoming limited and the resulting advent of multi-core processors, a lot of work has been done on parallel solvers. Usually these take the form of divide-and-conquer strategies, where the search space is divided over several sequential solvers, who work together by sharing learned clauses, as is the case in [6, 10].[1]

Both approaches have already resulted in substantial improvements by themselves, however in our opinion they still fall short in two ways. First of all, none of the existing state-of-the-art parallel solvers make

[1]Notable exceptions are the winners of the 2009 and 2011 SAT competition parallel track, ManySAT and Plingeling, which use a portfolio approach [5, 9].

use of the possibilities that have been opened up over the last couple of years by General Purpose GPU-programming. Typical parallel solvers make use of four or eight threads working concurrently, whereas the massive parallelism offered by GPUs nowadays allows up to thousands of threads. Second of all, these lines of research have evolved separately from each other. Taking our cue from the seventh SAT-challenge and taking it a step further, one can conjecture that combining the algorithms from both approaches into one framework is likely to be mutually beneficial. This work forms the first effort in overcoming these shortcomings. It presents a solver that is both hybrid as well as massively parallel.[2]

Our strategy is the following: we start out with parallelizing the basic hybrid solving method presented in [14], where the trace of an incomplete solver is used to guide a complete solver. Rather than sequentially calling the complete solver after every iteration of the incomplete solver, we run both at the same time. Furthermore, we will run a massively parallel adapted version of the used local search algorithm. To be more concrete, we implemented these ideas using the OpenCL framework. Our solver consists in running MiniSAT on the CPU, while a random walk algorithm will run on the GPU from which we derive a heuristic to guide MiniSAT, replacing its normal VSIDS-like heuristic.

The next section will first provide a very basic overview on SAT solvers and then introduces the ideas behind hybrid solvers. We will focus on the work done by Mazure and his colleagues, the success of which serves as the justification for our methodology. We then proceed, in section 3, to outline our approach. First we motivate the CPU-GPU implementation, after which we introduce the OpenCL framework. In subsection 3.3 we sketch the details of the OpenCL-SAT solver. The results on a set of test-cases are discussed in section 4. In conclusion, we reflect on the possibilities our research presents for future work.

2 SAT solving essentials

2.1 Background on SAT solvers

Traditionally there have been two types of solvers for the SAT problem, namely complete solvers and incomplete solvers. Complete solvers are guaranteed to decide for every SAT instance whether or not it is satisfiable, given enough time, whereas incomplete solvers may stop before finding a solution even if one exists. The benefit of the latter is that they are in general a lot faster on satisfiable instances than complete solvers, but they have the obvious disadvantage that they are incomplete. To repair this defect and have the best of both worlds, hybrid solvers were developed. These combine both types of solvers in a way that tries to retain as much as possible of their respective advantages.

The backbone of all complete solvers that have been developed over the last few decades remains the DPLL algorithm that was discovered in the 60's [7]. The algorithm is a recursive, depth-first enumeration of all possible assignments in the model space, which can be seen as a binary tree. The progress in this area was mainly due to intelligently optimizing the different parts of the algorithm, such as the heuristics used to choose the next branch in the search tree, and the data structures used for the propagation of unit literals. An additional improvement came from expanding the original problem by adding new clauses that have been learned from conflicting assignments (so-called Conflict Driven Clause Learning, CDCL). For our research the branching heuristic will be most significant. It determines in which order the search tree is traversed, and choosing a relevant heuristic can therefore drastically reduce the number of recursive calls to be made in order to find a solution. A lot of modern SAT solvers use variants of the Variable State Independent Decaying Sum (VSIDS) heuristic [17], which assigns a score to each literal according to its activity in the search process.

In contrast to complete solvers, a typical incomplete solver (as described in [18]) uses a greedy local search algorithm, where the variables are initiated with a random assignment, and at each iteration a single variable is 'flipped' to its opposite value. The variable to be flipped is chosen such that the number of satisfied clauses is maximized. This continues until either a solution has been found, or a fixed limit of flips has been performed. In the latter case, the process is repeated, for a set number of tries. To avoid ending up in local minima, the greedy approach is usually combined with a random walk, so that with a certain probability a variable that appears in an unsatisfied clause is chosen at random instead of the greedy choice [19]. This simple algorithm performs well on a wide set of instances, but is not guaranteed to find a solution.

[2]At an earlier stage of our research, the outline of our approach appeared already in [4].

Algorithm 1 DP+TSAT: basic version

```
Procedure DP+TSAT
Input : a set of clauses S
Output : a satisfying truth assignment of S, if found
         or a definitive statement that S is inconsistent
Begin
    Unit_propagate(S);
    if the empty clause is generated then return (false);
    else if all variables are assigned then return (true)
        else begin
            if TSAT( S ) succeeds then return (true)
            else begin
                    p := the most often falsified literal during TSAT;
                return (DP+TSAT($S \wedge p$)$\vee$DP+TSAT($S \wedge \neg p$));
            end;
        end;
End
```

2.2 The hybrid approach

Motivated by the seventh challenge mentioned in the introduction, Mazure et al. [1, 2, 3, 8, 11, 12, 13, 14, 15] have investigated several ways of combining complete and incomplete solvers. Mazure demarcates three types of hybrid approaches to be found in the literature [15].

1. DPLL-based solvers: a stochastic local search algorithm (sls) is used to guide the DPLL-like complete solver in choosing literals. That is, every time the solver backtracks because of an occurred conflict, the next literal is chosen with the help of a statistic obtained by running the sls solver.
2. Local search based solvers: a complete solver is used to help out an incomplete one. This can be done in several ways, such as letting DPLL look for dependencies between variables and then limiting the sls solver to a certain subset of variables.
3. Solvers where the incomplete and complete work together but retain their independence: first the local search is run for a certain time, if it fails to find a solution a CDCL solver is executed on the set of currently unsatisfied clauses to find out if they are unsatisfiable.

The original hybrid algorithm described by Mazure et al. [14] belongs to the first category: it uses a local search algorithm to provide a branching heuristic for DPLL. This approach led to a dramatic decrease in the runtime of hard unsatisfiable random instances, especially those that have a locally inconsistent kernel. Their local search algorithm (TSAT, or TWSAT) extends the basic GSAT version from [18, 19] by adding a Tabu list, so that recently flipped variables may not be flipped, preventing the algorithm from getting stuck in local minima and from moving back and forth between a small set of assignments.[3] During the local search, the following trace is recorded: for each literal, it is counted how many times it appears in a falsified clause, taking a flip as a measure of time. Literals that have a high score are more likely to belong to the inconsistent part of the instance than those with low scores. Indeed Mazure et al. discovered that using this literal score as a branching heuristic for a standard DPLL algorithm, whereby TSAT is called every time the DPLL algorithm needs to choose another literal and the literal with the highest score is selected, gives excellent results on locally inconsistent problems.[4] A basic schema of the algorithm is given by Algorithm 1, taken from [14].

In subsequent years, Audemard et al. have also developed other hybrid approaches, belonging to the second category [2, 3]. Nevertheless, the method set forth by the DPLL+TSAT algorithm continues to be pursued and improved upon, as the work of Fourdrinoy demonstrates [8]. In the next section, we explain how we improved this method by making extensive use of parallelism, thereby setting the stage for a new type of hybrid solvers.

[3]Taken by itself, TSAT already proves to be a substantial improvement over the basic GSAT algorithm. See [13].

[4]The popular VSIDS heuristic depends on a somewhat similar scoring mechanism, except that there the focus lies on learned rather than falsified clauses.

3 OpenCL-SAT: combining CPU and GPU

3.1 Motivation

With the increase of CPU speeds coming to a halt in the last few years, it becomes more and more clear that progress in purely sequential SAT solving can no longer expect to benefit from significant improvements in hardware. Luckily, parallel hardware is becoming ever more widespread, and multicore-CPU's and GPUs can be found in standard PC's. Yet current work on parallel solvers has been limited mostly to the former option, although thanks to the creation of new libraries such as OpenCL it is becoming possible to perform platform-independent GPGPU-computing. The goal of our research is to tap into these unexplored possibilities for SAT solving by designing a solver that exploits parallelism to the fullest.

A naive method would be to simply start from an existing multicore-based parallel approach using four or eight threads, and expanding it to several hundred threads on a GPU. The difference in architecture between both types of hardware makes such an approach impractical:

1. each thread on a GPU has only a very limited amount of (reasonably fast) memory compared to a CPU-core,
2. communication between threads is slow because of the limited bandwidth and is difficult to synchronize,
3. GPUs are made for SIMT (Single Instruction Multiple Threads) algorithms, and thus threads should essentially perform the same algorithm.

One of the main reasons why recent parallel solvers outperform sequential ones, is due to information sharing between threads. Each thread examines one part of the search space, using its own clause database, and passes on clauses to other threads based on their relevance for them. The memory requirements for giving each thread its own database conflict with the first difference and exchanging clauses conflicts with the second. Besides this, the third difference substantially limits the possibilities for threads to perform different tasks concurrenty, as can be done on a multicore-CPU.[5]

The fact that all state-of-the-art complete solvers have been developed for the CPU-architecture and thus don't lend themselves to GPU-implementations, is probably the main reason why so little work has been done on complete GPU-based SAT solvers. If we shift our focus to the emerging field of hybrid solvers, however, a new perspective opens up. We can make use of an optimized sequential complete solver on the CPU, while at the same time tap into the enormous processing power offered by the GPU. To implement such a solver, we made use of OpenCL.

3.2 OpenCL

GPGPU-programming is still a very young discipline, and until recently CUDA was the only widespread language available to do so. CUDA is a C-based programming language and environment for General Purpose GPU-computing created by NVIDIA. Its syntax and use are quite similar to regular C, except that special attention has to be paid with regard to synchronization for code that will be executed in parallel.

There have been some attempts of implementing massively parallel incomplete solvers on GPUs using CUDA, see for example [16] and [22]. In fact, we used the source code of the latter as a framework to implement our first prototype of a massively parallel incomplete solver.

Since CUDA was designed to run solely on NVIDIA hardware, a group of companies and researchers under the name of the Khronos group decided to develop an open standard, with the intention of providing cross-platform portability. In 2008 this resulted in the conception of OpenCL, or Open Computing Language, an open and royalty free programming framework that enables GPUs, FPGAs, and co-processors to work in tandem with the CPU.

OpenCL is based on ISO C99, with some added extensions and restrictions. The basic unit of executable code that runs on an OpenCL device (a GPU in our case) is called a kernel. The host program is executed on the host system (usually a CPU) and sends kernels to execute on OpenCL devices using a command queue. Besides the programming language, OpenCL also consists of a platform API, which contains routines to

[5]For example, ManySAT's portfolio approach, where each thread can use a completely different solving method, becomes impossible. Also, most parallel solvers use a load-balancing mechanism through work stealing, it is hard to see how this could fit into a SIMT framework.

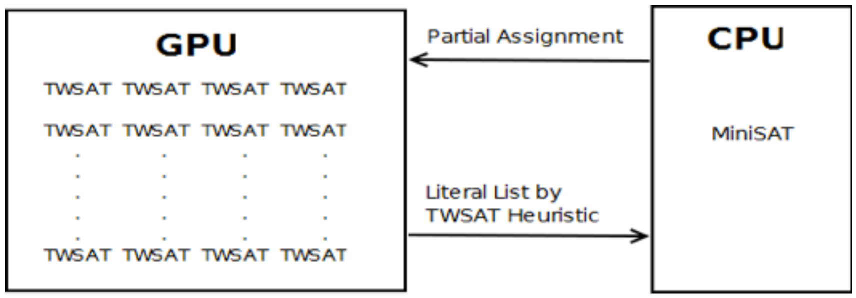

Figure 1: Schema of OpenCL-SAT.

query the system and set up OpenCL resources, and a runtime API, that is used to manage kernel objects, memory objects, and execute kernels on OpenCL devices.

The OpenCL language contains many commands to control the flow of data and the order of execution, thereby giving the programmer a large amount of freedom in specifying the details of how an algorithm should be executed. Therefore it allows one to optimize memory usage and thread synchronization, and offers a tight integration of CPU and GPU execution.

3.3 From hybrid to parallel

The big picture.

We applied the CPU-GPU perspective to parallelize the DPLL+TSAT technique described in subsection 2.2. The OpenCL framework provides us with the tools for writing programs that combine CPU and GPU resources to suit one's needs, which fits in nicely with the hybrid character: the CPU is used to run a version of MiniSAT [21], and we use the GPU to run a massively parallel local search that will complement the complete solver. For obvious reasons, we baptized our solver OpenCL-SAT.

In essence, the workings of OpenCL-SAT are rather simple. We adapt MiniSAT so that it no longer uses its VSIDS-like heuristic, but a form of the TSAT heuristic instead. In contrast to previous versions of this hybrid algorithm, we run TSAT in parallel on the GPU at the same time as MiniSAT. (As opposed to the sequential DPLL+TSAT algorithm, where TSAT is called every time MiniSAT needs to choose a literal, so that MiniSAT has to wait for TSAT to finish.) The CPU informs TSAT which variables have been assigned already at every stage, so that future tries of TSAT can run on the partial problem determined by the current subspace in which MiniSAT is looking. Likewise, all literal scores obtained from the local search trace are sent to the CPU, which keeps an ordered list of literals based on this score. Figure 1 depicts the idea behind OpenCL-SAT.

The algorithm in detail.

In order for the complete and the incomplete parts to be run concurrently in a sensible way, it is required that the average time for one try of the local search solver is of the same order of magnitude as the time in between two decisions of the complete solver.[6] Yet in the original implementation of the DPLL+TSAT algorithm in [1], one try of the incomplete solver takes as long as 45 decisions of the complete solver.[7]

A straightforward way of dealing with this is to use the literal score obtained from one try for 45 decisions of the complete solver. Doing this however results in a dramatic decrease of the effectiveness of the heuristics, making it all but useless. The only viable option is to drastically speed-up the calculation of the heuristic, without compromising its quality too much. We succeeded in this by putting to use the massive parallelism offered by a GPU. The GPU is used to obtain a heuristic which is based on the original idea - namely calculating a score for each literal by counting the number of its appearances in falsified clauses, but modified in several ways so as to function optimally on the chosen type of hardware.

[6]A stochastic local search algorithm consists in several *tries*, where each try is a certain amount of *flips* being performed on a random initial assignment.

[7]We obtained this number by running the original SUN-solver from [12] on the same test set of random instances that we also used in section 4.

Algorithm 2 basic local search solver

```
Procedure local search kernel function
Input : a set of clauses S over a set of variables V, an initial assignment A to V, MAX-ITER, $i_0 < |V|$
Output : an assignment to V satisfying S, if found
         or a score for all literals from V if not
Begin
    $i = i_0; PreviousChoice = i_0;$
    while (( $i <$ MAX-ITER $+ i_0) \wedge (i - PreviousChoice) < |V|)$
        $v = i \; mod \; |V|;$
        if $AlreadyAssignedMiniSAT(v)$ then $(i++; continue)$
        if $FalseClauses(S, A + flip(v)) < FalseClauses(S, A)$ then
            $A = A + flip(v);$
            $PreviousChoice = v;$
            $UpdateScore(A);$
            if $FalseClauses(A) = 0$ then return "satisfiable";
        else $i++;$
    end while;
    return "no satisfying assignment found";
End.
```

More specifically, we ran a large number of threads of a very basic stochastic local search solver depicted schematically in Algorithm 2.[8] The TSAT score for a literal is then taken to be the average of the scores given by each thread.

In a first attempt each thread started out with a different initial assignment, and ran until either a local minimum was reached or some maximum amount of iterations had occurred. We noticed two things:

1. even our very basic local search algorithm couldn't meet the speed requirements;

2. the quality of the heuristic didn't improve notably beyond using 200 threads.

So we decided to group several threads together into about 200 groups, and tried to let each group mimic the behavior of one independent try of the sls solver. We did this by letting each thread in a group start off with the same (random) initial assignment, and then flipped one randomly chosen variable for each thread. (Where we made sure that no two threads flipped the same variable, i.e. the assignments for two threads in a group differ in exactly two places.) Then we ran our basic local search algorithm, where we reduced the number of iterations by a factor of 100.[9] The idea is that assignments that are close to each other - measured by the Hamming distance - represent different iterations of a single initial assignment undergoing the local search algorithm. By using this strategy, we were able to bring the average time of one try of the incomplete solver down to around the same order of magnitude as an average decision by MiniSAT, and still succeeded in mimicking the original heuristic obtained in our first attempt.

There are several ways in which we had to adapt the TSAT local search algorithm in order to fit into this framework. Because of the very limited amount of available local memory on a GPU per thread, there was no space to remember useful statistics, such as a list of all clauses a certain variable appears in. Therefore every thread simply goes over all variables, and flips it if this decreases the number of falsified clauses in the current assignment. Variables that have already been assigned a value by MiniSAT are of course excluded from this process. After every flip we check whether all clauses are satisfied, if not then the literal score is updated. Most random walk search algorithms integrate random flips every now and then, but since we are averaging over a large number of threads we did not. Also, the use of a Tabu list[10] was left out of our algorithm for two reasons: firstly, in our set up it is impossible for the same variable to be flipped twice successively; secondly, it is less likely that a recently flipped variable is being considered to be flipped again.

[8]Depending on the GPU this can go up to 6000, for an NVIDIA Tesla GPU, and around 800 for a more standard NVIDIA Quadro FX2700M.

[9]The number of iterations was reduced to $0.02 * v$, where v stands for the number of variables that are currently unassigned by MiniSAT.

[10]I.e. a list of recently flipped variables that may not be chosen, in order to avoid getting stuck in local minima.

	OpenCL-SAT	DPLL+TSAT	MiniSAT
TOTAL RUNTIME	175.97	$2,431.31$	48.20
TOTAL DECISIONS	$866,898$	$432,389$	$418,908$

Figure 2: Results.

4 Results

We have tested the OpenCL-SAT solver on 50 random 3-SAT instances from the DIMACS SAT library [23], where we ran each instance with four different random seeds. We then took the average over all problems of the same size. The sizes ranged from 150 to 250 variables, with a clauses/variables ratio of 4.26 for all instances, meaning that they lie in the hardest region. We chose to focus on hard random instances because they are significantly smaller than crafted or industrial ones of comparable difficulty, small enough in fact to fit into local memory of the GPU, which makes the solver significantly faster than if it were using global memory.

We compared OpenCL-SAT to the original hybrid DPLL+TSAT solver, as well as to MiniSAT using its standard heuristics.[11] The tests for OpenCL-SAT and MiniSAT were run on an NVIDIA Tesla C2075 GPU and a Intel Xeon E5645 CPU.[12] In total there were 5600 threads running on the GPU, divided into workgroups of 20x20.[13]

Figure 2 shows the total runtimes and number of decisions.[14] First of all, we can observe that OpenCL-SAT is about 13 times faster than the DPLL+TSAT solver, confirming the possible advantages of the CPU-GPU framework. However, there is still room for improvement concerning the quality of the heuristics, as the DPLL+TSAT solver required about only half the number of decisions. It should be possible to divide the workload more efficiently between all threads, making the implementation more scalable. Furthermore, it's clear that for random instances the current approach cannot yet compete with MiniSAT concerning neither the quality of the heuristic nor the runtimes, even when we take into account the number of decisions.s

5 Conclusions and Future Work

We have built a SAT solver that is both parallel and hybrid. By doing so we have combined two trends in the SAT solving community that are becoming widespread. We were able to do so by making use of OpenCL, that taps into the resources offered by combined CPU-GPU computing. Our solver was based on the fairly basic hybrid algorithm from [14], but more complex hybrid algorithms have since been developed that perform better on a wide set of instances (see [15] for an overview). In the future, similar solvers can be developed that implement these complex algorithms. The challenge will be to adapt them in such a way that they can benefit optimally from the GPU's parallelism, without suffering too much from the GPU's limitations. In the long run, however, attention should shift to algorithms that are designed from the bottom-up with a CPU-GPU set up in mind. To conclude, although our solver is as yet still too basic to compete with state-of-the-art sequential solvers, it is a first step in the direction of a new type of solvers that contain a lot of potential.

References

[1] Audemard, G., Lagniez, J., Mazure, B. and Saïs, L.: Boosting local search techniques thanks to CDCL. In: 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, pp. 474–488 (2010)

[11] We did not make use of any restart policy, since in general only industrial and crafted instances benefit from this.

[12] For the DPLL+TSAT solver we used an Intel T9600 CPU, which makes the comparison of runtimes a bit unfair. However, when compared to the runtimes of our solver on this same CPU together with a more basic GPU, the results weren't that different. See the following footnote.

[13] We used a more modest NVIDIA Quadro FX2700M together with an Intel T9600 as well, running 864 threads. The total runtime increased only by 15%, so unfortunately in its present form the solver isn't very scalable. The main reason for this is that although adding more threats can greatly speed up the calculation of the heuristics to be used by MiniSAT, it has little influence on its quality.

[14] For each problem size we took the average for both runtime and decisions.

[2] Audemard, G., Lagniez, J., Mazure, B. and Saïs, L.: Integrating Conflict Driven Clause Learning to Local Search. In: 6th International Workshop on Local Search Techniques in Constraint Satisfaction, pp. 55–68, (2009)

[3] Audemard, G., Lagniez, J., Mazure, B. and Saïs, L.: Learning in local search. In: 21st IEEE International Conference on Tools with Artificial Intelligence, pp. 417–424 (2009)

[4] Beckers, S., De Samblanx, G., De Smedt, F., Goedemé, T., Struyf, L. and Vennekens, J.: Parallel SAT-solving with OpenCL. In: IADIS International Conference on Applied Computing, pp. 435–440 (2011)

[5] Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical Report, FMV Reports Series (2010)

[6] Chu, G. and Stuckey, P.J.: PMiniSAT: a parallelization of MiniSAT 2.0. In: SAT race (2008)

[7] Davis M., Putnam H.: A Computing Procedure for Quantification Theory. Journal of the ACM, 7, 201–215 (1960)

[8] Fourdrinoy, O.: Utilisation de techniques polynomiales pour la résolution pratique d'instances de SAT. Thèse de doctorat. (2007)

[9] Hamadi, Y.; Saïs, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modelling and Computation (2009)

[10] Lewis, M., Schubert, T. and Becker B.: Multithreaded SAT Solving. In: 12th Asia and South Pacific Design Automation Conference, pp. 926–931 (2007)

[11] Mazure B., Saïs, L., and Grégoire E.: TWSAT : a new local search algorithm for SAT : performance and analysis. In: the Workshop CP95 on Solving Really Hard Problems, pp. 127–130 (1995)

[12] Mazure B., Saïs, L., and Grégoire E.: SUN : a Multistrategy Platform for SAT. In: First International Competition and Symposium on Satisfiability Testing, SAT solvers description (1996)

[13] Mazure B., Saïs, L., and Grégoire E.: Tabu search for sat. In: 14th National Conference on Artificial Intelligence, pp. 281–285 (1997)

[14] Mazure B., Saïs, L., and Grégoire E.: Boosting complete techniques thanks to local search methods. Annals of Mathematics and Artificial Intelligence, 22, 319–331 (1998)

[15] Mazure, B.: SAT et au-delà de SAT : Modèles et Algorithmes. Habilitation à Diriger des Recherches (2010)

[16] Meyer, Q., Schonfeld, F., Stamminger, M. and Wanka R.: 3-SAT on CUDA: Towards a massively parallel SAT solver. In: High Performance Computing Symposium, pp.306–313 (2010)

[17] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L. and Maliket, S.: Chaff: Engineering an Efficient SAT solver. In: the Design Automation Conference, pp. 530–535 (2001)

[18] Selman, B., Levesque, H. and Mitchell, D.: A New Method for Solving Hard Satisfiability Problems. In: 9th National Conference on Artificial Intelligence, pp. 440–446 (1992)

[19] Selman, B., Kautz, H. and Cohen, B.: Local Search Strategies for Satisfiability Testing. In: Working notes of the DIMACS Workshop on Maximum Clique, Graph Coloring, and Satisfiability (1993)

[20] Selman, B., Kautz, H. and McAllister, D.: Ten Challenges in Propositional Reasoning and Search. In: 15th International Joint Conference on Artificial Intelligence, vol. 1, pp. 50–54 (1997)

[21] Sörensson, N; Eén, N.: An Extensible SAT-solver. Lecture Notes in Computer Science, 2912, 333–336 (2004)

[22] Wang, Y.: NVIDIA CUDA Architecture-based Parallel Incomplete SAT solver. Master Project Final Report, Faculty of Rochester Institute of Technology (2010)

[23] http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html