Parallelization of SAT Algorithms on GPU

Carlos Filipe da Silva Portinha e Costa

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Doutor Luís Jorge Brás Monteiro Guerra e Silva
Prof. Doutor Paulo Ferreira Godinho Flores

Examination Committee

Chairsperson: Prof. Doutor Mario Rui Fonseca dos Santos Gomes
Supervisors: Prof. Doutor Luís Jorge Brás Monteiro Guerra e Silva
Member of the Committee: Prof. Doutor Vasco Miguel Gomes Nunes Manquinho

June 2014

Abstract

The Boolean Satisfiability (SAT) problem is one of the most important NP-complete problems. However, during recent years, the performance of SAT Solvers has started to stagnate. This lead to efforts being done to parallelize SAT solvers for them to use multi-core systems. Meanwhile, GPUs became available for general programming. However, only recently there have been efforts to use them to solve SAT. Nevertheless, these techniques misuse the GPU and limit its applicability. This happens because they use the scalability of the GPU in order to be able to accommodate larger problems. However, when problems gain certain proportions, the GPU can not handle them anymore.

In this thesis we present a novel approach to use the GPU that, by carefully selecting the work to be performed, allows it to do, with few computing resources, what took others the entire GPU. This approach lets us have a portfolio of search threads, which can scale better and enables the GPU to solve problems of any size, which was not possible before with GPU-enabled solvers.

Keywords: GPU, Boolean Satisfiability

Resumo

O Problem de Satisfatibilidade Booleana (SAT) é um problema NP-Completo muito importante. No entanto, recentemente, a performance dos SAT solvers tem vindo a abrandar, o que levou à realização de trabalho na àrea da paralelização de SAT solvers para que estes pudessem tirar partido de sistemas multi-core. Entretanto, os GPUs deixaram de servir apenas para processamento gráfico, para passarem a servir para fazer qualquer tipo de processamento. Apesar disso, só recentemente é que começou a haver trabalho no sentido de se utilizar GPUs para resolver SAT. No entanto, as técnicas aplicadas não utilizam todo o potencial do GPU e por isso, limitam a sua aplicabilidade. Isto acontece porque estes trabalhos utilizam a escalabilidade do GPU para conseguirem acomodar problemas cada vez maiores, no entanto, quando os problemas atingem um certo tamanho, o GPU deixa de conseguir processá-lo da melhor forma.

Nesta tese apresentamos uma nova abordagem à utilização do GPU, que, pelo facto de escolhermos cuidadosamente o trabalho que este tem que fazer, permite ao GPU conseguir fazer com poucos recursos, o que em outras abordagens só seria possivel com todo o GPU. Esta abordagem permite-nos ter um portfolio de procura, que consegue escalar melhor e permite ao GPU resolver problems de qualquer tamanho, o que não era possivel anteriormente.

Keywords: GPU, Problema de Satisfatibilidade Booleana

This dissertation culminates my research which started as part of the ParSAT Project, under the supervision of professor Luis Guerra e Silva and professor Paulo Flores. I want to thank them for their guidance and patience during this whole process. I know that without their support I would not have learned so much.

I am also grateful to the ParSAT research group that, during our meetings, provided invaluable insights that proved to be crucial to this thesis.

Last but not least, I want to thank my friends and family for their unconditional support.

The work developed on this thesis was partially supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under the project "ParSat: Parallel Satisfiability Algorithms and its Applications" (PTDC/EIA-EIA/103532/2008)".

Contents

List of Tables

List of Figures

List of Acronyms

CPU Central Processing Unit

GPU Graphical Processing Unit

CDCL Conflict-Driven Clause-Learning

GPGPU General Programming GPU

SMT Simultaneous Multi-Threading

SIMD Single Instruction Multiple Data

SAT Boolean Satisfability

API Application Programming Interface

Chapter 1

Introduction

The Boolean Satisfiability (SAT) Problem consists of, given a Boolean formula, find an assignment to its variables that makes it true, or prove that no such assignment exists. It was the first NP-Complete[1] problem to be identified [2]. For many reasons this is still one of the most important problems in Computer Science. Two of such reasons are its simplicity and its numerous practical applications, namely software testing, theorem proving, Electronic Design Automation, among many others. But, perhaps the most important reason, is the ability to reduce any NP-Complete [1] problem to an instance of the SAT [3] problem, making it an essential tool in solving this class of problems.

One standard algorithm to solve SAT is the Davis-Putnam-Logemann-Loveland (DPLL) [4], introduced in 1962 as an improvement of the DP algorithm [5], proposed two years before. This algorithm solves the SAT problem by recursively traversing the entire search space, finishing only when a solution is found or when the search has exhausted all the possibilities. Nevertheless, it introduces several improvements over a more basic brute-force approach. The DPLL algorithm [4] is still the basis for most modern SAT solvers [6], further enhanced with efficient data structures, advanced implementation techniques and heuristics. Examples of such enhancements are: the use of watched literals [7] to monitor clause status; conflict analysis [8], which enables the solver to learn every time a conflict occurs, such that it will not be repeated; and non-chronological backtrack [8], which enables the solver to backtrack several levels at a time, instead of just one. Furthermore, heuristics of variable selection [7] and search restarts [9] were also introduced, with the objective of improving decisions and reducing the penalty for bad ones, respectively [9]. These improvements enabled modern SAT solvers to outperform the initial DPLL by several orders of magnitude, enabling them to scale from a few variables to tens of thousands [7].

Despite the continuous and significant efforts of the SAT community, in recent years the global performance of state-of-the-art solvers has stagnated and only minor optimizations have been introduced to improve it [10, 11]. Moreover, as CPUs hit their physical limit in terms of single core speed, improving

SAT solvers performance requires the development of parallel approaches [10]. The initial method to parallelize SAT solving was to split the search space between cores/threads, and, with load balancing techniques re-split the space every time a core/thread finished its search [12, 13]. These solvers could also share important clauses, induced by conflict analysis, during their search [14]. The most successful solvers are the so-called portfolio solvers [15]. A portfolio solver has several different single-threaded solvers, running in each core/thread competing against each other while also sharing meaningful learnt clauses [16]. In this approach, each solver explores the entire search space, but resorting to different techniques and/or configurations.

In recent years, with the introduction of General Purpose GPUs (GPGPUs), the GPU have ceased to be graphics only hardware, to be used to perform general purpose computing. GPUs have hundreds of cores and a raw computing power that outperforms that of the Central Processing Unit (CPU), if used properly [17]. However, the execution paradigm of the GPU, Single Instruction Multiple Data (SIMD), is different than that of CPUs, Simultaneous Multi-Threading (SMT). In this paradigm, unlike in the SMT model, where each thread can be issued a different instruction, in SIMD all threads will execute the same instruction, meaning that branches and memory accesses, may be serialized.

Even though extensive research was conducted on the implementation of parallel SAT solvers, running on multi-core CPUs, only recently there have been efforts to take advantage of the GPU. These efforts use the massive parallelism of the GPU in order to be able to accommodate larger problems, as they analyze all clauses in each propagation, allocating one thread per clause. This approach makes sense from a SIMD perspective as it results in an execution that is very regular. This is due to the fact that the work done during the analysis of different clauses is very similar. However, with this approach, most of the work done in the GPU will be useless since not all clauses need to be analyzed. Furthermore, this approach misuses the GPU, as small problems are not able to take advantage of this parallelism as they are not large enough to occupy the entire GPU. Moreover, the opposite happens for big problems, where the GPU parallelism will not be enough to handle them effectively, as its resources become too low for the problems at hand.

To address these issues, in this thesis we propose a novel approach that, by doing a careful selection of the work to be executed on the GPU, can keep the resource utilization low, while maintaining a balance between regularity of execution and usefulness of the work done. This way, we are able to use the parallelism of the GPU to parallelize not only clause analysis but search as well. Furthermore, since we can fit one search thread in each GPU block, we eliminate the need for inter-block communication, and thus, we reduce the execution switching between the GPU and the CPU.

This way, we are able to have a portfolio of search threads cooperating to solve a single problem, which will essentially enable us to scale our performance by adding more blocks, and consequently, more search threads. Another advantage is that, by having a portfolio of search threads we can also add

diversity through different heuristics and randomization, which is not possible otherwise. The result is a solver that is not limited by the problem size, and can scale through the addition of more blocks regardless of the problem.

The remainder of this thesis is organized as follows. In Chapter 2 we review the relevant concepts that we believe will facilitate the understanding of this thesis. We will review the basic SAT solving concepts and techniques, and we will also present a brief introduction to the CUDA programming model, summarizing its advantages and disadvantages. Afterwards, in Chapter 3, we will discuss related work conducted in the area of parallel SAT solving, with particular emphasis to the work that uses the GPU. The remaining chapters of the thesis are focused on our work. In Chapter 4 we explain our initial approaches and the tests that we conducted, which helped us to understand their limitations and, in doing that, helped us to converge to the final solution. In Chapter 5, we will introduce our proposed approach for solving SAT using the GPU, and also explain a few optimizations that were required to ensure maximal performance. Finally, in the last two chapters of this thesis, we will evaluate our solver and compare it with other GPU-based which will enable us to draw some conclusions regarding its performance, scalability and limitations.

Chapter 2

Background

In this chapter we provide a comprehensive explanation of the different concepts and techniques that are important to the understanding of this thesis, inclusively, some that are used by our solution. We will postpone, to the next chapter, both a description of the state of the art and a comparison to systems that stand to achieve the same goal as ours. The rest of this chapter will be organized as follows first, in Section 2.1 we will introduce the SAT Problem and its concepts. Then in Section 2.2 we will briefly introduce the DPLL algorithm and in Sections 2.3, 2.4 and 2.5 we will introduce the concept of conflict-driven clause-learning (CDCL) SAT solvers, search heuristics and problem preprocessing techniques, respectively. Finally, in Section 2.6 we will present a short introduction to the CUDA programming and execution model.

2.1 The Boolean Satisfiability Problem

2.1.1 Problem Statement

The Boolean Satisfiability Problem consists of, given a Boolean formula, determine if there is an assignment, true or false, to all, or a subset, of the Boolean variables, such that the formula evaluates to true [2].

$$((x \lor \neg z) \land y) \lor (\neg x \land y)$$

Figure 2.1: Example of a Boolean formula.

In a formula, like the one in Figure 2.1, there might be more than an instance of each variable. To each instance of a variable or its negation we call a literal. So both x and $\neg\, x$ are literals of the same variable x.

A variable can have one of three values, *t*rue, *f*alse or not a value. In a satisfied formula, that evaluates to true, not all variables may have an assigned value.

2.1.2 Conjunctive Normal Form

Every Boolean formula can be converted, and expressed in the conjunctive normal form (CNF). In this form a formula is composed of conjunctions of clauses, that in their turn are the disjunction of the literals that compose them, see Figure 2.2 as an example of a Boolean CNF Formula.

$$(x \vee \neg z) \wedge (\neg x \vee y) \wedge \neg z$$

Figure 2.2: A Boolean formula in CNF form.

So, in order to satisfy a CNF formula, one needs to satisfy all the clauses that compose it. This representation provides several advantages that make it a central part in modern SAT Solvers. The advantages of this representation are that, it makes inference easier [5], and it also makes the analysis of the formula more modular because the problem can be analyzed in parts instead of as a whole. This representation also simplifies the process of conflict analysis and clause learning, which will be discussed further ahead.

As mentioned before, one of the advantages of the CNF representation is the fact that we can infer values to variables that are in accordance with the current assignment, and we can do that with ease. To do that, first we need to locate unit clauses. Unit clauses are clauses where all literals, but one, are assigned and evaluate to false. So, if we want to satisfy all clauses, we must set the value of the last variable with respect to its representing literal, so that the clause is satisfied. This required assignment, or inference, is called an implication [5].

2.1.3 Complete and Incomplete Solvers

SAT solvers can be classified as complete or incomplete. The difference between the two is that complete solvers will terminate whether a solution is found or not because they exhaustively try all possible solutions. On the other hand, incomplete solvers, that are mostly based on stochastic local search, will only terminate when a solution is found or by timeout, as they have no way of knowing if every solution was verified. Moreover, incomplete solvers have advantages over complete solvers, as in general, they are easily parallelizable as they have very few data dependencies [18].

2.2 DPLL Algorithm

The DPLL [4] is a complete algorithm to solve the SAT problem. It was created in 1962 and it still forms the base of most complete solvers. This algorithm is a simple backtracking scheme that tries all possible combinations to the problem, but it differs from more simplistic brute-force schemes by employing unit propagation, a technique that uses the problem constraints to avoid unnecessary assignments, among other simplifications. A modern, iterative version of the DPLL, is shown in Algorithm 1.

Algorithm 1 Iterative DPLL pseudo-code

```
1:  procedure ITERATIVE_DPLL
2:      status ← preprocess()
3:      if (status = / = UNKNOWN) then return status
4:      while (true) do
5:          decideNextBranch()
6:          while (true) do
7:              status ← propagate()
8:              if (status == CONFLICT) then
9:                  btstatus ← backtrack()
10:                 if (btstatus ==UNSATISFIABLE) then return UNSATISFIABLE
11:             else if (status == SATISFIABLE) then return SATISFIABLE
12:             else break
```

It has several phases, the first phase is the variable selection procedure in which the algorithm selects the first unassigned variable and sets it to true. After this assignment the clauses are analyzed and, if unit clauses are found, this constraint is propagated. This propagation phase goes until no more unit clauses are left or a conflict is found. If this phase ends with no conflicts, another variable is picked and assigned. If a conflict was found, the search backtracks. A conflict happens when all the literals in a clause evaluate to false. This means that the current assignment, partial or not, will not satisfy the formula so we can stop searching in that direction. When this happens, as noted before, the search backtracks. A backtrack is nothing more that invalidating all assignments done until we undo the last picked variable. When we reach this state, we either flip the assignment, or if we already did so, we look further back for yet another picked assignment. If during this process, we reach a point that no more variables are left to undo, the search finishes and we classify the problem as unsatisfiable (UNSAT). If during the search, we find an assignment that satisfies all clauses, we classify the problem as satisfiable (SAT).

2.3 Conflict-Driven Clause-Learning

Modern solvers, based on the simple DPLL, employ the Conflict-Driven Clause-Learning (CDCL) scheme [19]. This name comes from the fact that unlike DPLL, modern solvers learn every time a conflict happens [20]. This allows the solver to prevent the same mistake from happening again and it also enables the solver to better lead the search based on what it already knows [7].

2.3.1 Implication Graphs and Clause Learning

The central part of the conflict analysis, is a concept known as implication graph [20]. The implication graph represents the reasoning leading to a certain conclusion, generally a conflict. In fact, at any given time, the implication graph represents the current assignments and the implications they generated. This graph is a concept, as it does not exist, it is instead built implicitly in the solver's state.

This graph has, in each vertex, a variable, with some information that is relevant, such as the implication level, and its assignment. The level starts at zero, and is increased every time a variable is picked, rather than implied. Levels will decrease during backtrack. Joining these vertices, we have edges that hold information about the clause that links the two variables.

A vertex may have more than on edge pointing to it, one for each variable that form the clause that lead to the implication. Similarly, it can also have more than one edge pointing out, one for each implication.

Every time a conflict happens, this graph can be analyzed and through this analysis we can create new clauses that prune the search by allowing us not to repeat the same mistakes. Additionally, by analyzing the conflict clause, we can also conclude to what level we should backtrack.

After we are done with the analysis of the implication graph, we store the reason for the conflict in the form of a new clause. The method of determining this new clause is simple.

We start from the clause that has the conflict. If we find a variable that is not from the current level, we mark it. With the other variables, the ones from the current level of assignment, we follow the graph backwards to the clause that generated that assignment and we repeat the procedure with this clauses until we have no more clauses to analyze. Then, after this is over, we collect the marked variables and also the last variable from the current decision level to be seen, we call this the asserting literal, we negate their assignment, and we create a clause from this set of variables and their assignments [20]. This clause will allow us to avoid making this mistake ever again.

2.3.2 Non-chronological Backtrack

A learnt clause can help the search in another way. This clause will enable us to, under certain conditions, backtrack more than one level at a time.

As said before, we collect all the variables that we find during the analysis, that do not belong to the current level of assignment, in a clause. So, it is easy to see that, until we undo one of these variables, the conflict clause will still be negated and the conflict will continue to be, as its cause has not been undone. That is, until we backtrack to the level that contains a variable present in the clause, we still have the conflict. So by backtracking directly to that level, we end up avoiding these subsequent mistakes [20].

When we reach that level, we pick the asserting literal, assigning it the value that satisfies the clause and the search goes on.

2.3.3 Two Watched Literals

Another addition to the modern SAT Solvers is a data structure known as two watched literals [7]. This is a lazy data structure, in the sense that it is only evaluated when it needs to, that aids in the process of unit propagation and backtrack. It does that by facilitating the analysis of a clause.

With watched literals, instead of having to process the entire clause to check for any implications, we just check two literals. These literals should be either satisfied or unassigned. Every time one of those two literals evaluates to false, we move our whatch to another literal in the clause that that is not falsified. If we have only one place left and it is unassigned, we have found an implication. If, on the other hand, we have no places left to put any of the watches, we have a conflict.

The two watched literals structure has another feature. As the structure is lazy, it is only evaluated when it is used, so the position of the watches does not need to be updated when the search backtracks [7].

Moreover, when propagating an assignment, we do not need to check all the clauses the assigned variable belongs to. We only need to analyze the clauses where it is being watched, thus further reducing the work needed to be done for each propagation.

2.4 Search Heuristics

The search for a solution can be troublesome and a bad decision can have an high cost. To avoid this, and other similar problems throughout the search, some heuristics were devised.

In this section we will introduce the heuristics that have the greater impact in modern SAT solvers. We will first introduce a variable ordering heuristic that takes into account the most recent events to increase

the importance of a variable. After that, we will present a way of picking the polarity of a variable that takes into account the value set by the constraints of the problem. Thirdly, we are going to present a way of classifying the clauses we learn so we can remove the least important to keep the problem size manageable and we are also going to present ways of deciding the best time to do this cleanup. Lastly, we are going to present restart policies, these are crucial to the solving process because they restart the search, but, this time with all the information that was collected till then.

2.4.1 Variable Selection Heuristics

The Variable State Independent Decaying Sum (VSIDS) heuristic [7] was introduced with CHAFF [7]. This heuristic gives priority to variables that were involved in recent conflicts. It does this by increasing the activity of a variable by an amount, during conflict analysis. Then, periodically, it divides all the activities effectively making old conflicts worth less than recent conflicts.

By doing this we try to assign first the variables that are more active and causing more conflicts, saving for last the variables that have lower activity.

2.4.2 Phase Saving

Deciding which polarity to assign to a variable is a task as difficult as choosing the right order for them. One way to decide this is to attribute random, or even fixed, polarities, in which we always attribute the same polarity, true of false, to all the variables. The variables only get the opposite polarity when implied, or, when their polarity is flipped during backtrack.

Phase saving is a dynamic polarity heuristic. It tries to be smarter than random or fixed schemes as it uses the information about polarity given by the problem itself. It achieves this by saving a variable's polarity every time it is implied, and using this saved polarity later if the variable is picked during the decision process [21].

2.4.3 Clause Rating Heuristics

With the learning process, the number of clauses that we need to analyze will grow until a solution is found. However, even though each learnt clause will prune the search space, having too many clauses, will also slow the search down. This means that there is a balance between the number of learnt clauses, and the overall performance of the solver. Because of this, it is important to reduce the number of clauses by eliminating some, periodically. This approach rises two problems, how to select the clauses to remove and when to remove them.

The VSIDS for Clauses, is a dynamic method that follows the same concept of VSIDS [22]. It privileges clauses that were involved in recent conflicts. Just like VSIDS for variables, when a clause is directly or indirectly causing a conflict, it has its activity increased and just like with the variables this activity is decreased periodically. The selection process discards the clauses that are less active in detriment of more active ones.

The literal blocking distance (LBD) method, is a static rating method where clauses are rated at creation. This method works during conflict analysis. While the analysis is being ran, we keep the decision level of the variable being processed. In the end, we count the number of different decision levels, the variables involved in the conflict belong to, and we attribute that value to the clause as its rating [11]. The authors think that clauses that have very few participating levels are the ones most likely to produce conflicts in the short term [11]. Moreover, when generated conflict clauses, the clauses with a low LBD value, usually of 6 or less, are selected for a stronger clause strengthening process, which will take more time than the usual clause strengthening routine. Additionally, the authors found that clauses with LBD of two or less, are as important as binary clauses, and they called them glue clauses, this name comes from the fact that they signalize a strong connection between the variables that constitute it. This heuristic chooses the clauses to eliminate by sorting them by their LBD value and, inside the same LBD Value sets, by their activity, effectively discarding the second half of the clauses. However, binary and glue clauses are never eliminated. This rule as only one exception, if all clauses have similar LBDs, this means that the LBD is not a reliable method for rating. If that is the case, the clauses are eliminated solely based on activity, regardless of their LBD rating.

A different approach is Polarity with freeze/unfreeze, which was proposed by Audemard, et al. [23] where clauses are rated based on the saved polarity of the variables during phase saving. First, clauses are assigned score points for each literal that is in accordance with the saved variable polarity. Then, clauses with low score are unfrozen and made available to the solver once again. The choice goes towards low score because they will more probably produce an implication or a conflict. To use a term we introduced before, clauses with low score have an higher probability of becoming active. This method has the advantage that it is dynamic, as in, clauses can be re-rated at will and thus they never become obsolete. Because of this, the authors opted for not removing clauses but rather freezing and unfreezing them when they become relevant.

However, rating the clauses is just first step to tell us which clauses we should remove. When it comes to actually removing them, different strategies can be used, both to find the moment to clean the database, and to find the right size for the database. The most usual methods consists on counting the number of clauses, and when this number surpasses a certain threshold the cleaning process is triggered. The threshold itself usually grows, either linearly [11] or exponentially [22].

2.4.4 Restart policies

The search process, in its beginning, only has the information that is present in the problem itself. However, as said before, during the solving process modern SAT solvers will learn additional clauses, better variable orderings and polarities, among other things. Nonetheless, mistakes made earlier in the search, before this information was available, make it harder to capitalize on it. Moreover, these mistakes sometimes lead the search in a direction and it takes a long time for them to be undone [9].

One way to contradict this effect is to periodically restart the search. By restarting the search, as we have increasingly more information, we are able to switch the focus of our search to more interesting parts of the search space preventing us from focusing our time on the wrong place.

We are going to explain the tree restart strategies that we consider the most important for this thesis: the luby restart strategy, the exponential back off and the LBD dynamic restart.

Luby restart strategy This restart policy is based in the luby sequence [24], which is a sequence that results in fast and repeated restarts.

Exponential back off is a restart strategy in which the number of conflicts between restarts grows exponentially [22].

LBD Restart strategy This restart strategy was meant to be an improvement to the luby restart strategy. In luby, the focus of the search switches rapidly due to fast restarts. However, one does not know if this change was beneficial or not, as it does not use a metric to try to measure if the search is close to a solution [11].

This strategy tries to switch to ever better search tree regions by comparing the latest search results with a global view of the search [11].

It does this by having two metrics of the quality of the ongoing search:

- the LBD of the conflict clauses rating, that decreases when we approach the solution of the problem;
- the number of assigned variables at each conflict, that, on its hand, grows as we approach the solution;

To have a global and a local view of the search so far, the authors use two moving averages. A moving average is an average that uses the latest N elements of a set. A regular average, contrastingly, would use an N equal to the total number of elements.

For the LBD rating averages, the authors use a smaller moving average of the last 50 conflicts for the local view, and the last 1000 conflicts for the global view. The objective is to keep the local average below

the global average. Every time the local average surpasses the global average, the search restarts. This calculations only start when both moving averages have enough elements to be calculated. When a restart is triggered the smaller sized moving average clears and only when it reaches 50 elements again a restart can happen.

For the number of assigned variables, we have a similar scheme, but we look for an increasing number of assigned variables instead.

However, the number of assigned variables does not trigger a restart. It does the opposite of that. Every time the limit is crossed, the restart is blocked. This means that the current local average is cleared and thus we have 50 conflicts more to find a solution.

A benefit of this strategy, other than the focus on search quality, is that by forcing a decreasing LBD rating for the clauses, we are forcing the solver to always learn clauses with increasingly better quality [25].

2.5 Problem Preprocessing

Another technique to aid in the solving process, is to preprocess the problem with the objective of reducing its complexity.

One of the most easy to implement preprocessing rules are to perform unit propagation on the problem by checking if any clause is unit, and to perform pure literal elimination, which consists on finding variables whose literals come only in one polarity. By assigning that polarity we can safely remove the clauses containing that variable from the problem [4].

More sophisticated techniques focus on deriving units, implications and equivalent literals [26, 27]. Other techniques focus on reducing the size of the clauses [28].

2.6 CUDA

In this section we will introduce CUDA, a programming framework for machines with nVidia GPUs. It is composed by a programming language, a compiler and a runtime environment.

CUDA enables the GPU, whose processing power gap is growing larger each year when compared to CPUs, to execute regular code, rather than just graphics. However, it only performs to the extent of its capabilities when applied to data parallel intensive processing problems.

The difference between the CPU and the GPU is that CPUs are optimized for fast single thread execution, they are good for complex control logic and out of order execution. They have a large cache, to hide RAM accesses, and the cores are optimized to take advantage of these caches. Contrastingly, GPUs

are optimized for high multi-threaded throughput. However, these threads are not good at handling conditional execution splits and their caches are small. Therefore, GPUs hide memory access times by using thousands of threads cooperatively.

2.6.1 CUDA Programming Model

The CUDA programming model is called Single Instruction Multiple Threads (SIMT) and it has some advantages as well as some drawbacks when compared to Simultaneous Multi threading (SMT), the programming model of multi-core CPUs. One advantage is that it has that it allows a much higher number of threads to be ran concurrently, enabling a cheap high-throughput, high-latency design, moreover, each thread may have up to 63 registers. A disadvantage is that SMT provides mechanisms of concurrency that are not present in GPUs such as locks and semaphores.

CUDA programs are called kernels and are organized into blocks of threads. Each block may have up to 2^{16} threads organized in up to three dimensions.

The threads in a block can be synchronized within the block with barriers and can communicate using the per-block shared memory. Concurrency treatment is limited and solely handled by atomic functions that operate at memory position level. This is the only concurrency permitted, accesses to one memory position. This limits the ways in which concurrency can be applied, as there is no way to allow several operations to be carried out in succession.

Each group of 32 consecutive threads is called a warp[1]. A warp is the fundamental unit of execution in the program. The same instruction is issued to all threads in the warp. This means that if one thread in a warp executes conditional code, the others will have to wait until that branch is done. Therefore, if the code has an high degree of branch divergence, this results in performance loss, as its execution is serialized. Other cause for loss of performance is data dependency, if a thread in a warp is waiting for data, the others have to wait as well.

These blocks, are organized into Grids. Grids can also be organized in one or two dimensions. The maximum number of blocks that can run at the same time is hardware specific.

2.6.2 CUDA Memory Model

In CUDA there are several memory levels with different speeds and characteristics. The efficient usage of all these levels is key in achieving the maximum possible performance. However, this is not always easy to achieve mostly due to space constraints.

[1] this is the current warp size. However it is architecture dependent, so it may change in the future

2.6.2.1 Global Memory

This memory is generally in the order of the Gigabytes and is accessible by all threads.

This memory has the duration of the application and can be used by different kernels. However, its access speed is slow, usually taking between 400 to 800 cycles per access. The preferred way of accessing global memory is by doing coalesced accesses. A coalesced access is made when all the threads in a half warp[1] access contiguous memory positions. If this happens, these accesses are condensed in a transaction. This reduces the access time and is the only way to achieve peak memory throughput. If the memory positions are not sequential, the read instruction is repeated until all accesses are performed. However, the GPU only stops in case of data dependency making the accesses asynchronous.

2.6.2.2 Pinned Memory

Pinned memory, is an extension to Global Memory, which allows the GPU to access memory allocated on the GPU. It is called pinned memory, because it cannot be swapped. This memory is good for data that is only read once, but is used several times, during the execution of the kernel.

2.6.2.3 Shared Memory

Shared memory, uses the same memory bank as L1 cache. It is the fastest memory directly available for the programmer to use. It takes two cycles per access, but it is very small much like L1 cache, ranging from 16KB to 48KB. Shared memory is shared between all threads in a block and can be used for communication. The memory is divided in 32 banks, which can not be accessed at the same time. If a concurrent bank access exists, they are serialized resulting in a repetition of the instruction, causing performance degradation. This memory should be used when the access pattern to the data is random resulting in performance degradation if global memory is used. This memory has the lifetime of a block.

2.6.2.4 Local Memory

This memory is the local thread memory and it resides in global memory sharing its disadvantages. This memory has the lifetime of the thread that owns it. It is allocated automatically by the compiler if there is no more register space available. This creates a tradeoff between the number of threads one wants to have running at the same time, as more threads may result in more local memory, as the register count is limited and architecture dependent.

[1] the first half of the threads the last half of threads of a warp

2.6.2.5 Registers

Registers are allocated automatically by the compiler. However, the number of registers can be reduced by hand at compile time, through a compiler flag. Registers are, like in the CPU, the fastest form of memory but also the scarcer one.

2.6.2.6 Memory Bound Scalability

Each semantic multiprocessor (SM), the name given to GPU processors, has limited shared memory and registers, so the number of blocks that can be allocated at the same time in a SM is limited by the amount of shared memory and registers they use. For instance, if a SM has 32768 registers and each block has 256 threads using 16 registers each, each SM can have 8 blocks running at a given time[1]. A similar calculation can be made for the shared memory.

2.6.3 Kernel Performance Limitations

When profiling, there are three types of kernels. This classification show us where to perform more effective optimizations. The limiters are often the number of instructions one can execute, and the speed at which one can read from memory. To better explain these cases, we are going to present extreme cases of each type of kernel.

A kernel is considered instruction bound when the execution time of a kernel is spent doing calculations, and not memory accesses. The way to optimize this kind of kernels is to choose a more efficient way to compute the result, like using better instructions or a completely different algorithm. This happens when the bulk of the computation is to be done in the same data or when the access patterns are such that they become meaningless performance wise.

Unlike with instruction bound kernels, when memory accesses dominate the kernel's execution time, the kernels are called Memory bound. This means that even with the best access patterns possible, the kernel will still be limited by the speed at which it can obtain new information. The way to optimize this kind of kernel is to try to do the processing in batches. These batches, are moved to shared memory where they can be processed at the fastest available speed. However, not all problems can be compartmentalized in batches and so this optimization is only applicable to a certain subset of problems.

When neither instruction, neither memory dominates the execution time, there is another limiter to the performance, Latency. Latency happens when instructions are repeated, mostly due to bad memory accesses patterns or due to serialization as a result of branch divergence, atomic operations or bank conflicts in shared memory. An example would be if an instruction is issued to read from random

[1] 32768 registers / (256 threads *16 registers) = 8 blocks

memory positions. As said before, each half warp, will be executing this memory access command, and if coalesced, it will result in a single memory access for all threads. However, in this example, with random memory positions that is unlikely and, the accesses will be serialized. This instruction will be repeated 16 times until all threads have the information they need. This happens mostly when the data model of the problem, does not fit the GPU memory model well, and that limits the performance of the kernel.

Chapter 3

Related Work

With the advent of multi-core CPUs, and with the per-core speed coming to a stall, there was a need to develop SAT Solvers that could take advantage of these systems. Moreover, GPUs, that were once only used to perform graphic processing, are now able to perform more general computing. Therefore, several attempts to take advantage of the massive parallelism of GPU architectures to speed up the solving process have recently been proposed.

In this chapter we will introduce some of these improvements. We start by introducing parallel SAT solvers. We will present both the Divide-and-Conquer and the portfolio approaches. Afterwards, in Section 3.2, we will introduce the main approaches to SAT Solving using the GPU. In Section 3.2.1, we will focus in the auxiliary procedure approach, where solvers run a complete solver in the CPU and an incomplete solver in the GPU, cooperatively. Finally, we will end this chapter with the introduction, in Section 3.2.2, of solvers that use GPUs to assist the solving process.

3.1 Parallel SAT Solvers

The development of parallel solvers went from a basic divide-and-conquer approach, where solvers run in non-overlapping search spaces, that was soon improved with load-balancing and better space-splitting heuristics, to a new method (that uses solvers with different heuristics), taking advantage of the problem type specialization that can be achieved by doing so. These solvers, designated by portfolio solvers, unlike solvers based in a divide and conquer approach, all run in the same search space, with surprisingly good results.

3.1.1 Divide-and-Conquer Based Solvers

The first approach to parallel SAT solving was based on a simple divide-and-conquer scheme. The rationale underlying this approach is that by having each solver search a disjoint part of the search space, they will eventually reach the solution, or exhaust all possible combinations, faster than a single solver searching the entire space alone.

3.1.1.1 pSATO

In PSATO [12] Zhang et al. proposed a parallel and distributed solver based on their previous serial solver SATO [29]. This solver uses a master-slave model where the master tries to balance the work of the slaves. Moreover, each solver explores a disjoint search space so that no two solvers could repeat combinations between themselves. These disjoint search spaces are generated by setting initial values to a set of variables, so that each solver is assigned a different sub problem to work on. Another advantage of this architecture is that PSATO [12] had fault tolerance in case one of the slaves was lost.

3.1.1.2 GrADSAT

In GRADSAT [14] a solver that targeted the grid, a scalable learning mechanism, was introduced, where the slave solvers share clauses. This decision to share clauses was introduced because, while each individual solver was learning from its search, as soon as they terminated, this information was discarded. However, some of these clauses, even though they are collected in disjoint search spaces, can be relevant to other slave solver. Still, the losses, in terms of communication overhead, between the different solvers, made it difficult so share them [14]. Therefore, in GRADSAT [14], limits were established such that only clauses that respected certain conditions could be shared, more specifically, clauses with less than 10 literals. Moreover, when slave solvers received clauses, they only added them when they backtracked to the zeroth level. Adding a clause can result in one of four outcomes:

- If the clause only has one unknown literal, then it will result in an implication.
- If the clause has more than one unknown literal then it will be added to the clause database
- If all the literals in the clause are falsified, then the sub-problem is unsatisfiable.
- If all the literals are true, then the clause is discarded since it is useless to prove the search space.

The decision, of only adding clauses on the zeroth level, was made because it made implementation simpler and also because it enabled clauses to be added in batches [14].

3.1.2 Portfolio Based Solvers

The Portfolio approach to SAT solving tries to leverage on the fact that heuristics play an important role in the effectiveness of the solving process, and that, with the right heuristic a problem that is otherwise hard to solve can be solved in a fraction of the time it once took. The main challenge is finding the right heuristic for each problem. Portfolio solvers address this by having different solvers employing different heuristics and competing to solve the same problem. Thus increasing the chance of having a solver with a heuristic suitable for the problem at hand. This approach has yet another advantage over its predecessor. By having different solvers working on the entire problem, when a solver reaches a solution, this result is the solution for the problem, hence, eliminating the need for load balancing. These ideas, coupled with the clause sharing technique already mentioned, makes the state of the art in parallel SAT solving.

3.1.2.1 ManySAT

MANYSAT [16] introduced the idea of portfolio SAT solvers. According to its authors, sequential SAT solvers suffered from weaknesses such as their sensitivity to input parameters and lack of robustness, in the sense that a solver could be great for some problems and useless for other kind [16]. Therefore, to tackle this problem, MANYSAT [16] used a portfolio (hence its name) of sequential solvers that were carefully tuned. Each of these carefully tuned sequential solvers shared clauses with the others to improve the overall performance of the system. MANYSAT [16] used at the time of its creation, four solvers tuned the following way:

- Solver 0: geometric restarts, chooses variables using VSIDS with 3% of random decisions. The polarity mode is to greedily assign the polarity that solves most clauses and it learns clauses from solver 1;
- Solver 1: Dynamic (fast) restarts, uses VSIDS with 2% randomness, polarity mode is phase saving and does not receive clauses.
- Solver 2: arithmetic restarts, uses VSIDS with 2% randomness, always chooses false to the assigns and does not receive clauses.
- Solver 3: luby 512 restarts, uses VSIDS with 2% randomness, polarity mode is phase saving, and learns clauses from Solver 1.

This way, MANYSAT [16] was capable of achieving above average performance over standard Divide-and-Conquer algorithms [16].

3.1.2.2 Plingeling

The current state of the art in portfolio and parallel SAT solvers is PLINGELING [30]. It runs several instances of LINGELING [30] that are organized on a master-slave architecture and cooperate through clause sharing. To share clauses, they are sent to the master node that sorts them by the order they were received and eliminates them when all slave nodes got them. This operation takes place in regular intervals during the CDCL loop. The solvers only share three types of information: unit literals, equivalent literals and clauses that meet the following two conditions 1) to have forty or less literals and 2) their LBD is of 8 or less. In PLINGELING, the solvers have the same heuristics and parameters, instead, to ensure different decisions, the authors opted to give different, random, initial orderings to the variables.

3.2 GPU-enabled Solvers

With the emergence of GPGPUs some attempts were made to capitalize on their processing power to solve the SAT problem. That being said, there are two main approaches to use a GPU in a SAT solver, the first is to use it to run an auxiliary incomplete solver, while having a complete, state-of-the-art solver running in the CPU. The second approach is to use the solver to tackle some part of the solving process, like unit propagation, while executing the rest on the CPU. This is the approach that our solution will follow, as we relegate work to the GPU, instead of using the GPU to run a different solver in a portfolio like scheme.

3.2.1 Auxiliary GPU procedure

The first approach to leverage on the GPU to solve the SAT problem, is, as said before, to use an hybrid solver. Hybrid solvers use two different approaches, one in the CPU and another in the GPU. In the CPU they use a regular complete solver, such as MINISAT, which can run completely separate of the solver running in the GPU. The solver running in the GPU is usually an incomplete solver. These have several advantages over the CPU counterparts that make them more suitable to be ran on the GPU. Such advantages are that usually incomplete solvers have very few, or even none, data dependencies and are easily adaptable to the GPU architecture [31].

By doing this, they keep having the speed of the state of the art solver they choose to run and they have in the GPU a solver that is more tailored to it. Moreover, the solution remains complete because of the CPU side of the solver, which allows us to prove if a formula unsatisfiable rather than to just the satisfiability of some.

3.2.1.1 MESP

In MESP (Minisat Enhanced with Survey Propagation) [18] the authors proposed using a GPU implementation of Survey SAT to enhance MINISAT's variable picking solution, VSIDS. The authors chose SURVEY SAT because it was easily parallelizable as the key parts of the algorithm did not have data dependencies, aspect that also makes it very well suited for a GPU. The authors can, with their GPU version of Survey SAT, achieve speedups greater than 100x when compared with the CPU version of the same algorithm [18].

Has the name suggests, in MESP, MINISAT relies on this implementation of SURVEY SAT. When the application starts MINISAT reads the problem and a copy is sent to the GPU. During the search, from the set of generated learnt clauses only those with size less than 50 are copied to the GPU. Then, SURVEY SAT is ran on these clauses and once its stochastic search converges to a steady state, this information is used to increase the activity of the variables used in the VSIDS heuristic in MINISAT. This way, MINISAT can take advantage of the high rate of information that the GPU can produce to better guide its search.

The authors report a speedup of 2.25x when MESP is running the 3-SAT version of random instances compared to MINISAT which was running the original problem instances (kSAT). When comparing to MINISAT run times for the 3-SAT version of the SAT instances, MESP was on average about 2.42$\times$ faster.

3.2.1.2 Hybrid Solver with TWSAT

In his paper, Beckers et al. presented a way to use OpenCL to solve SAT [32]. This Solver is an Hybrid solver as well and it combines MINISAT, running in the CPU with an incomplete solver, a GPU implementation of TWSAT [33]. The approach is similar to one presented by Mazure et al. [34] when he noticed that VSIDS was not good for large unsatisfiable instances as it would not converge to an ordering. The author noticed that, even though it would not solve unsatisfiable problems, TWSAT would converge to a set of variables that were more important than the ones provided by VSIDS [34]. Noticing this, the author proceed to use both MINISAT and TWSAT cooperatively so TWSAT could enhance the VSIDS variable weights. Beckers followed the exact same methodology, but instead of having both solvers running on the CPU, he pushed TWSAT [33] to the GPU.

Beckers managed to get speedups in the order of 3.5x for unsatisfiable instances and 4x for satisfiable instances, in random 3-sat problems with clause to variable ratio of 4.26 which lies in the hard region [32].

3.2.2 CPU/GPU Cooperation

Another way of using GPUs in the process of SAT solving is to use the GPU either to solve the entire problem or to speedup a specific task, such as unit propagation. We are going to present three solvers

that use the GPU. Fujii et al. [35] that uses the GPU to speedup the clause analysis only, the CUD@SAT project [36] which uses the GPU with various configurations, some only do unit propagation, others, when the search reaches a certain number of set variables, the subsequent tree is sent to the GPU for processing, and finally we are going to present a project by Meyer et al. [37] that did not intend to set the newest trend in SAT Solving, but rather to try and make a scalable framework to Solve SAT on the GPU.

3.2.2.1 GPU Accelerated Boolean Unit Propagation

Fujii et al [35], proposed to use the GPU as an accelerator for the unit propagation procedure of 3-SAT problems, mimicking a previous work by Davis et al. where Davis et al. [38] used a FPGA to accelerate the analysis procedure, instead of using GPUs. This Solver uses a basic DPLL approach and only parallelizes the clause analysis procedure. Every time a variable is picked and set, the GPU is called to analyze all clauses to search for implications. If an implication is found, the GPU is called again with this new information, if no implication is found, a variable is picked instead. In this implementation, the CPU holds the state of the problem, and as the objective of the work was only to speedup analysis, the backtracks are done on the CPU and are chronological. They the solver with the same implementation on the CPU and report a speedup of at most 6.5x [35].

This approach shows the superiority of the GPU in processing great amounts of data but, by having no conflict analysis, and consequently, no clause learning, they end up with a solver that is not competitive.

3.2.2.2 CUD@SAT

The CUD@SAT [36] project was developed by the Constraint and Logic Programming Lab of the university of Udine. This solver has several possible configurations that range from CPU only approaches, to having the CPU and the GPU solve the same problem cooperatively. There have several methods to achieve this cooperation:

- they use the same approach as Fujii et al. [35], doing only clause analysis in the GPU, but they introduce conflict analysis, as opposed to Fujii's [35] method;
- they run a solver in the CPU and when the search reaches an advanced state, they pass the search to the GPU.
- they do everything in the GPU, alternating between a search kernel, and a conflict analysis kernel, with the CPU serving as synchronization.

Noting that, all these methods can be ran with or without a watched literal scheme. In all it's flavors the project remains constant in the fact that the only processing being parallelized is the clause analysis. However, the project does not propose a default configuration.

The authors report speedups ranging from 2x to 8x comparatively with the same algorithms running exclusively on the CPU. Yet, these speedups are not an average of all the options but instead achieved using the best possible configuration for each problem.

3.2.2.3 Scalable GPU Framework

Meyer [37], proposed a parallel 3-SAT solver on the GPU that featured a pipeline of kernels that processed the problem, the author discarded most common techniques and relied only on the massive thread parallelism and the scalability of the GPU to solve problems. The focus of the work was to determine the scalability and applicability of GPUs to SAT solving rather than trying to devise the next high end SAT solver [37].

Chapter 4

Initial Approaches for Solving SAT on the GPU

Because our final approach is the result of the refinement of previous attempts, we will now present those attempts, and the ideas we took from them, before introducing our final solution.

We started off by trying to solve the problem only by relying in the extreme parallelism of the GPU by doing a matrix multiplication scheme. Noticing that we could not tackle even small sized problems, we turned ourselves to DPLL based approaches, as they, unlike the previous approach, do not try every possible combination. Finally, we implemented the DPLL algorithm with a divide-and-conquer strategy but the running times where still high so we ended up improvising it with conflict analysis.

This chapter is organized in the following way. In section 4.1 we will explain the various brute force approaches and, in section 4.2, we will do the same for the DPLL based approaches. In the final section of this chapter, section 4.3, we will draw some conclusions from these attempts and explain how they lead us to the final approach.

4.1 Brute-Force Approaches

Our first approach was to try and take advantage of the GPU's massive parallelism capabilities. To do this we used a simple approach, encode the problem in matrix form and multiply it by a possible solution. If the resulting vector is all 1's then the problem is satisfiable.

$$(x \lor \neg z) \land (\neg x \lor y) \land \neg z$$

$$\begin{pmatrix} 1 & 0 & -1 \\ -1 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Figure 4.3: The before mentioned CNF formula in matrix form.

$$(x \lor \neg z) \land (\neg x \lor y) \land \neg z$$

$$\begin{pmatrix} 1 & 0 & -1 \\ -1 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix}$$

Figure 4.4: The previous problem being tested against a possible solution $x, \neg y$ and z. There are no matches in the second clause meaning this solution does not solve the problem.

4.1.1 Matrix Multiplication

As mentioned before, our first approach was to use a matrix multiplication scheme to solve our problems. To encode the problem in matrix form the following steps were taken, first a matrix with variables x clauses is created and zeroed. Each row of the matrix represents a clause. For each clause, if a literal is positive a one is put in that variable place, if it is negated, a -1 is put instead, the remaining positions, keep their original value, zero. The multiplying vector is encoded the same way, a positive variable is a 1 a negative is a -1, in this representation there are no unassigned literals, as can be seen in the matrix depicted in Figure 4.3. After the encoding, each solution is tested in the following way, if the values, the one on the solution and the one in the matrix, are different, the result is zero. Else, if the variable is equal to the literal then the result is one. Note that this works because there are no zeros in the assignment vector. If the sum of the results is not zero, the clause is satisfied. However, if the result is zero, the clause is unsatisfied, this is depicted in figure 4.4.

This solution had several problems that we will now explain. The first problem was that the size it took to represent the matrix was prohibitive. With $\mathrm{O}(variables \times clauses)$ size the memory limit of a GPU was easily surpassed. Second, because we only tried complete assignments, the algorithm would not do any sort of assessment to try to avoid unnecessary ones. As it could not imply variables, no conflict analysis could be done. This rendered the number of possible tries at $\mathrm{O}(2^x)$ where x is the number of variables in the problem. Another problem was that the majority of the operations were useless. As the matrices generated with this method are sparse, which means that they are mostly populated with zeros, most operations are comparisons with those zeros. Comparisons we already know the result in advance.

This method, although a brute force approach, its very efficient from the viewpoint of the GPU. The

reason is that there is memory locality in every operation. All accesses, both in the assignment vector and in the matrix, are sequential which means they are coalesced operations. Furthermore, there is almost no divergence in the code executed.

However, this method, due to its complexity, failed to solve even simple problems so we set to improvise it.

4.1.2 Compressed Matrix Multiplication

To try to address the space constraint, and the inefficiency of the method itself because of the sparseness of the matrix, we implemented the same scheme but this time with a compressed matrix representation. This representation is the one we carried over to the final solution. The matrix is represented as an array of clauses, with the size of the clause as the first element. An auxiliary array has the information of the first position of each clause as shown in Figure 4.5.

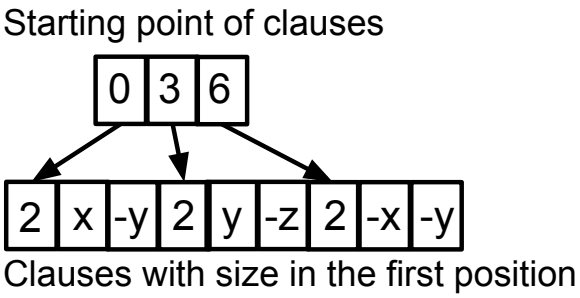

Figure 4.5: Compressed Matrix representation.

This allowed us to reduce the space necessary to hold the same information and at the same time, to skip useless comparisons. However, one fact still held, the fact that we could do no conflict analysis or implications. which lead us to implement a different approach that did not relied solely on massive parallelism but on a smarter algorithm as well.

4.2 DPLL Based Approaches

Our third approach tried to capitalize on the new representation with an improved algorithm. To do this we implemented a basic iterative DPLL. The algorithm is basic in the sense that it has the limitations of the first DPLL algorithm. However, we ran a parallelized version of the algorithm on the GPU. This algorithm, as expected, improved the results of previous approaches but they were still not very interesting as the time it took to solve problems was still high. Because of this we incremented this algorithm with conflict analysis.

4.2.1 Search Space Partitioning

As told before, we proceeded to implement an iterative version of the DPLL algorithm. This algorithm improved over the previous approaches as instead of trying all the combinations possible, we let the problem guide us to some conclusions through implications, that, in a way, cut the search space and this resulted in improved running times.

We used the compressed matrix representation, from before, and we only propagated the clauses we needed. To fully use the GPU we divided the search space by pre-attributing different values to a set of variables, so each solver would search in a different search space.

This solution however, had one problem. As we had no conflict analysis, we only had chronological backtrack and we also could not learn during the search. So, even if we could tackle simple problems, it failed to solve problems that took solvers, like MINISAT, less than a second.

4.2.2 Non-Chronological Backtracking

To partially defeat the last challenge, the problem of having chronological backtracks and no learning, we implemented conflict analysis on the GPU. Which means that each individual solver would, every time a conflict occurred, analyze the conflict graph and it would, sometimes, backtrack more than the normal chronological backtrack.

This implementation had two problems, first this analysis was done on a single thread, and defeats the GPU advantage of having thousands of threads running in parallel. Second, in addition to being single threaded, it added so much divergence to the code that with non-chronological backtracks, it became as fast, and sometimes slower than the initial code.

4.3 Conclusions

With the last exercise, we realized that it would not be possible to achieve the level of performance we strived to achieve. We wanted a competitive SAT Solver and these initial attempts were not able to solve problems of reasonable size of complexity. This failure, to add additional intelligence to the algorithm, showed us that doing everything on the GPU was not an easy task. Moreover, besides the fact that we did the entire analysis, adding a clause to the problem was hard. There were no global synchronization mechanisms, because not all blocks run at the same time, so having those would cause frequent deadlocks, and even the block level mechanisms were not enough to, for instance, add clauses to the problem on the go. Other problem common was that if a solver ended the search in its search space, it was not possible to return to the CPU to do load balancing as this block could not communicate

with others.

These first attempts and the problems they posed, helped us conclude that to have a full fledged SAT Solver that took advantage of the GPU, we needed to use the CPU as a synchronization point. This way, by moving conflict analysis and memory management to the CPU, we could have conflict analysis, which meant learning clauses, non-chronological backtracks and possibly better results.

Chapter 5

MinisatGPU: Our Proposed Approach

This chapter presents our proposed approach for solving SAT, which relies on the cooperation between the CPU and the GPU. First, in Section 5.1, we will present the key aspects and a top-level view of the system, its components and data structures. Afterwards, in Section 5.2 we present the most relevant implementation details of both, the CPU and the GPU, components of the solver, as well as several decisions that lead us to the final version of the solver.

5.1 System Overview

As explained in the previous chapter, our first approach was to design a solver that would run exclusively on the GPU. However, that idea was discarded since we realized that in trying to further optimize the solver, we would end up introducing inefficiencies which eventually would lead us to hit a wall in terms of performance. Such inefficiencies were the direct result of a tradeoff between a simple and less effective algorithm, albeit more appropriate to the GPU design, and a more effective one, but less GPU-friendly. We were able to use this tradeoff to our advantage, up to a certain point, where further optimizations would not translate into performance gains. These attempts eventually lead us to realize that, to continue improving the performance, it was necessary to move some of the computation from the GPU to the CPU. We decided to keep the execution of the unit propagation procedure in the GPU, while the conflict analysis was to be moved into the CPU, as it was a cause of significant performance degradation on the GPU. One side benefit of this decision is the ability to add new clauses to the problem before every call to the GPU, which enables learning from conflicts.

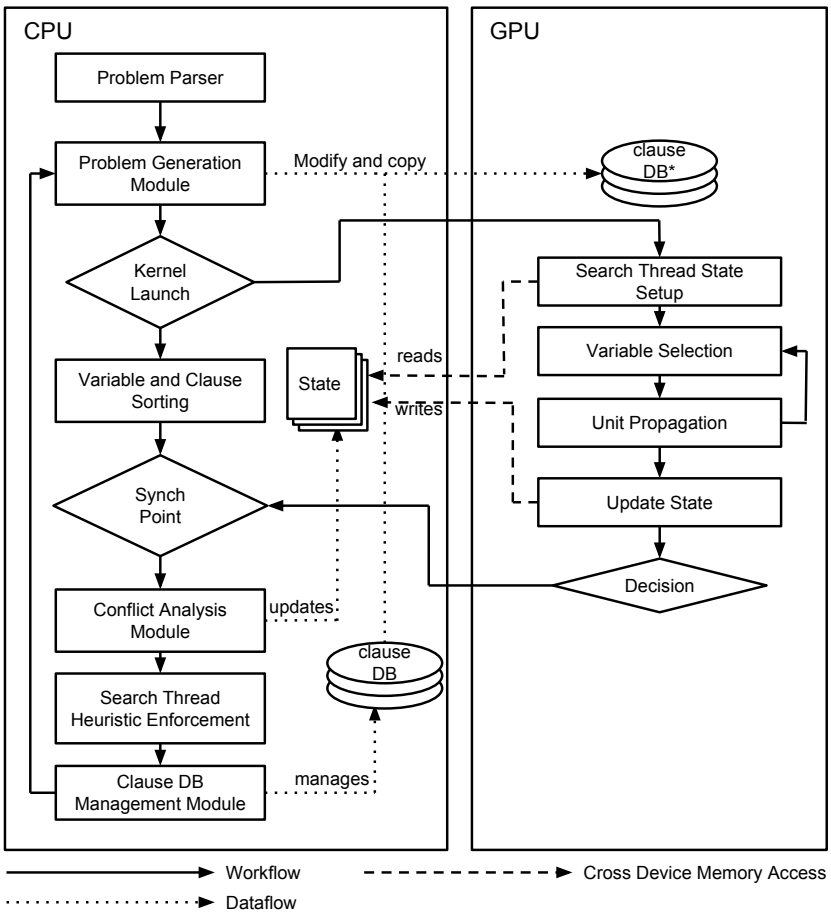

Figure 5.6: Diagram of the system architecture.

5.1.1 A Novel Approach

Like ours, most GPU-enabled SAT solvers, employ both the CPU and the GPU. However, they use the GPU in a suboptimal fashion, both in the way they approach the problem, and in the consequences that the said approach produces. We will first explain the approach, its drawbacks, and, after that, its practical consequences.

When trying to solve a problem, most GPU-enabled solvers use a single search thread [1], and they use the entire GPU to solve that same problem. The way they use it, is very GPU-friendly as they only use the GPU for propagations, a very regular procedure, using one thread per clause, with clauses split between various different blocks. This approach, however, exhibits a fundamental flaw. Since there is a limit to the number of blocks that can be ran at the same time on the GPU, this approach

[1] we call search thread to the path a solver takes during search. For instance, MINISAT is single threaded, meaning it only has one search path, and consequently a search thread. Contrastingly, a portfolio solver, such as PLINGELING, searches through a different path with each sub-solver, meaning it has several search threads. When we refer to search threads, we will always write search thread, as opposed to CPU or GPU threads which we will refer to as threads, only.

will fail to scale to bigger problems, as the GPU will fail to process all the clauses at the same time. Additionally, in small problems, that do not have enough clauses to fill the entire GPU, there will not be a way to improve speed by adding more blocks consequently, the GPU will be under performing. This approach has other disadvantages, other than the ones mentioned before. The consequence of using all blocks to perform propagation of the same search thread, is that, since there is no method for inter-block communication, the GPU will have to use the CPU as a synchronization point. This means that, every decision and implication must be made definitive on the CPU, that will also have to determine if there are no conflicting assignments between the blocks, before switching the focus back to the GPU. This results in unnecessary focus switches, that as previously said, are expensive.

In this work, we propose a different approach, that addresses these problems. Instead of analyzing all clauses, we chose to analyze only the clauses we need to. We do this by analyzing the effects caused by one assignment, decision or implication, at a time. By doing this careful clause selection, we execute in a single block, the work that, in the previous approaches takes, the entire GPU. This is possible because, with this approach, the limiting factor is no longer the total number of clauses, but the ratio of clauses per variable, which will grow much slower than the total number of clauses.

Our proposed approach has several advantages. First, since there is no need for inter-block communication, the entire search process, (decisions and propagations) can be executed in the GPU. The execution returns to the CPU only when a conflict or a solution is found, without relying on it for synchronization. Second, as we can fit a search thread per block, we can scale to several blocks by adding more search threads. This allows effective scaling our solution to problems of any size, without wasting the capabilities of the GPU.

The architecture of our GPU-enabled SAT solver is illustrated in Figure 5.6.It consists of three key phases:

1. The search phase, that comprises both the decision phase and the unit propagation phase;
2. the conflict analysis phase, that, after checking if no solution was found, either SAT or UNSAT, analyzes the conflicts that were returned by the GPU;
3. The problem regeneration phase, in which the clause database, with the newly learnt clauses, is converted to a GPU-friendly format and sent to the GPU.

5.1.2 Architecture

Our system is partitioned into four components. One of these components is executed in the GPU, while the others are executed in the CPU. The four components are:

- Search Engine

- Conflict Analyzer
- Clause Database Manager
- Search Thread Handler

The **Search Engine** is the only component that is executed in the GPU, while the others are executed exclusively in the CPU. This partitioning was adopted because, as the execution pattern of the search process has little divergence, it suits the GPU extremely well. Furthermore, searching is the bulk of the work in CPU solvers consuming around 80% of the running time [7], meaning that in this case the majority of the workload will be done on the GPU.

The other three components are executed on the CPU. The first component is the **Conflict Analyzer**. It takes the states returned by the GPU and analyzes the conflicts, computing a minimized conflict clause as a result.

The second is the **Clause Database Manager**. This component holds all the clauses, both the initial set and the ones generated during conflict analysis. Besides this, it is also responsible for the regular cleanups and for the conversion to the GPU notation. Furthermore, this component was modified to be able to handle the database even though that, unlike in MINISAT, the CPU side of the solver is stateless, as will be explained in Section 5.2.1.

The last component is the **Search Thread Handler**. This component manages our portfolio of search threads. Each search thread has an handler that will decide its behavior in accordance with its heuristics, also, it handles all the information a search thread needs, such as state and polarity mode. These handlers are read and updated both in the GPU and in the CPU.

5.1.3 Data structures

SAT Solvers need to hold the information about the problem, however, we need to hold this information both in the CPU and in the GPU with slight differences. We are now going to proceed to introduce the main data structures used by our solver, both in the GPU and the CPU.

The primary data structure is the Clause Database, since it holds the problem definition. Both the CPU and the GPU have a copy of the Clause Database, but their format is slightly different. As shown in figure 5.7, we need some information when in the GPU that is not necessary in the CPU, and vice versa. Specifically when on the GPU, we do not need the metadata that MINISAT stores in each clause, or the LDB weights. That being so, when we move the clauses to the GPU, we replace this information with the size of the clause and the Watched Literal Index. So when we create the Database that we are going to copy to the GPU, we replace that information, with information that better suits our needs.

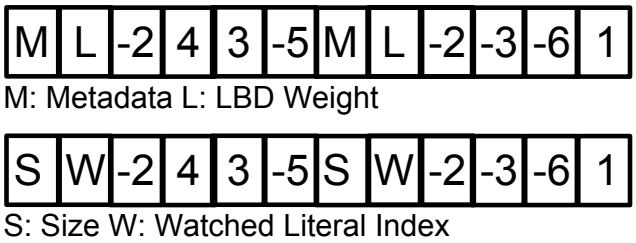

Figure 5.7: The clause information in the CPU and in the GPU.

The literal vector itself is of little use, as there is no way to know where a clause starts, and another ends. We make the search engine aware of this by sending an additional structure that, for each variable, has the list of clauses it belongs to. This structure is depicted in Figure 5.8. This structure has pointers to the start of the list, and in the first position of the list, there is a value representing the number of elements. This way, we can loop the list without crossing bounds to clauses of other variable.

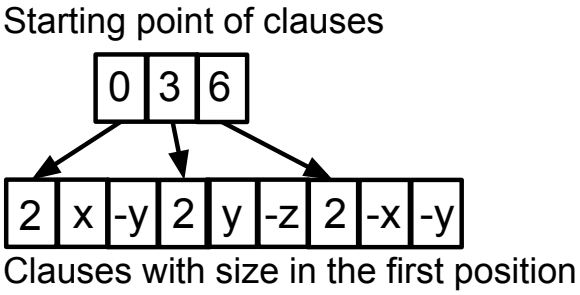

Figure 5.8: Compressed Matrix representation, with clause pointer.

Lastly, we need the information about the state of each search thread. This information, depicted in Table 5.1, is stored in a vector of structures. Since the vectors needed to hold the additional information, such as the current assignments, is used, and accessed by both the CPU and the GPU, we store them sequentially in one allocation only after the problem is processed. We do this because to access CPU data from the GPU that is sequential is faster than doing the same with data that is scattered.

5.1.4 Workflow

The components of our system cooperate combining their information, so that together they can solve the problem. The general workflow of our solver is the following:

- we process the clause database in the CPU and we send it to the GPU.
- we assign values to variables and propagate those decisions in the GPU until either a conflict, or a solution, is found.

Parameter	information
current level	indicates the current level of assignment
implying clause	vector that stores the clause that implied each variable
variable level	vector that stores the assignment level of each variable
variables	vector that stores the assignment of each variable
trail	ordered assignment trail
trail size	number of variables on the trail
trail limits	keeps where each assignment level begins in the trail
new variables in trail	number of new variables on trail (set by the CPU)
number of vars	number of variables in the problem
conflict	clause where the conflict happened (set by the GPU)

Table 5.1: Contents of the state structure.

- we return to the CPU where we will perform conflict analysis and, if necessary, a clause database cleanup.

- we are back in the beginning, where we process the clause database before we send it to the GPU.

As depicted in Figure 5.6, the whole process starts when a problem is given to the solver. MINISAT'S built-in parsers will read the problem and store it in internal data structures. After this, the solver will configure itself according to the command switches, some of which are already present in the original MINISAT, and others were added, as shown in Table 5.2.

Switches	Description	Default Value
–blocks	Number of solvers to be launched concurrently (one per GPU block)	128
–pfreq	Frequency at which the clause database is pushed to the GPU	1
–gpu-threads	Number of threads to be used during unit propagation	192
–l-lin	Increase the clause database capacity linearly, glucose style	true

Table 5.2: MinisatGPU's configuration parameters.

The next step is to set up the search threads. Each search thread will have its own context and heuristic, for restarts and polarity mode. This constitutes the setup phase. After, comes the solving phase, which will only end when an answer is found. The solving phase, uses the GPU, so the first step is to read the Clause Database and send the problem definition to the GPU formatted adequately. In this step, as said before, additional information is added to each clause, before it is sent to the GPU to help during search.

At this point, the problem definition already resides in the GPU, therefore we can start the execution on the GPU, simultaneously, we work in parallel in the CPU, performing some activities that can be done in advance, such as sorting the variables with respect with their VSIDS weights, pre-sorting the clauses when a database cleanup is eminent so can skip that before removing the clauses. This way, we use the CPU to advance work, when it would, otherwise, be idle.

In the GPU we will do a one-time copy of each search thread state to an on-chip-memory auxiliary container. Since these states, as mentioned before, are stored in pinned memory, which can be accessed from the GPU. The next phase is to propagate the variables that were implied in the CPU, either as a result of conflict analysis, or as a result of other search thread work. If these don't exit, we select one variable, instead. After this, each unassigned variable, and implication, is picked in succession, one at a time, and the clauses where they show up are processed in parallel, one per GPU thread. If during this stage, an implication is found, it is marked for analysis. We leave this stage with one out of two conditions:

1. there are no more variables to propagate, in which we proceed to select another one;
2. a conflict was found, in which case the search terminates;

A solution was found when no more variables are left unassigned, and there is no more work to be done. The execution on the GPU stops when all threads have finished their work with either a detected conflict, or provided a solution.

At this point, the CPU-side of the solver, as mentioned before, we will either have a solution, or as many conflicts as search threads. In which case, we will have to analyze the returned conflicts. To be able to perform conflict analysis, we had to modify MINISAT's procedure since MINISAT stores state in the clauses, and several steps of the solving process depend on it, such as clause management, conflict analysis and clause strengthening. However, as we have several search threads and, consequently, states, we cannot have this built-in state.

After conflict analysis finishes, we will proceed to add the resulting clauses to the database. However, unit clauses, which have only one literal, will not be added to the database, those are added directly to each search thread as an assumption instead. If two assumptions are inconsistent, the solving process ends with UNSAT. When adding new clauses, if we exceed the limit of clauses that the database can have, we clean the database, again using a modified procedure, we increase the limit, either exponentially or linearly, depending on the selected configuration, and we send the problem to the GPU again. At this point we are at the beginning of the solving loop, again.

5.2 Implementation

To implement our system, we decided it would be wasteful and time consuming to start a solver from scratch, and, instead we chose to customize MINISAT [22], a lightweight and extensible solver. This decision was motivated by the fact that since MINISAT [22] already provides a basic infrastructure such as problem parsing and clause management routines, our job would be less prone to errors. Since,

MINISAT [22] has been extensively tested, when integrating our changes, we could be fairly certain that wrong answers to problems were the result of our modifications. However, integrating with MINISAT [22] also presents some disadvantages as we have to make the GPU side of the solver compatible with the data structures of MINISAT [22]. Also, as we use MINISAT's [22] own conflict analysis and clause strengthening routines, we had to make them GPU-friendly as in our solver, unlike in MINISAT [22], the CPU-side of the solver does not have an implicit state.

This section is organized as follows, first we will introduce the changes we made to MINISAT [22] for it to suit our needs. After that, we are going to present the adaptations we had to make on the GPU code, being that some of it was reused from the previous DPLL approach. Finally we are going to present the mechanisms used for the cooperation between the CPU and the GPU.

5.2.1 Minisat Customization

During the process of customizing MINISAT [22] to our needs, we had to make several changes to the way MINISAT [22] operates. Since MINISAT [22] is highly optimized to have only one search thread, and wanted to make it able to handle several search threads, we had to modify it to do so. In Table 5.3 we present a summary of these customization.

The first thing we had to change was to make MINISAT [22] stateless. Besides the assignment stack, MINISAT [22] stores state in the clauses, as it changes the order of the literals in them so that the two watched literals are always the first two in the clause. As we have multiple search threads running in the GPU, they cannot change this order as the order for one search thread, would not suit another. However, as mentioned before, MINISAT [22] makes use of this state when during conflict analysis, conflict clause strengthening and clause management.

Conflict analysis makes use of this ordering because the first literal in a clause is always the implied literal. This means that, because during analysis we follow these implications backwards, when we leave a clause, to analyze another, the variable we followed to reach the new clause, points to the first one. If we were to use this literal in our analysis again, we would start an endless analysis loop. To prevent that, we had to change the analysis procedure to have in mind that clauses were stateless. We did this by introducing a marker that skips the analysis of variables that already were, or are, in the queue to be analyzed. This is problem is depicted in Figure 5.9.

Another problem was that this queue should have the asserting literal added in last place. However, due to concurrency issues in the GPU, the assignment trail may not be in the assigning order that MINISAT [22], which runs only in one thread, understands. So we had to use a special queue that realized which literal was the asserting literal and returned it in last place so the analysis could proceed normally.

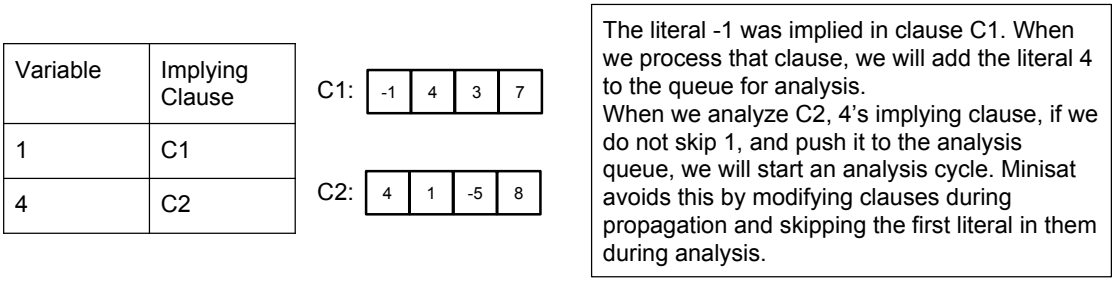

Figure 5.9: How Minisat uses state to avoid analysis loops.

In the conflict clause minimization procedure, which finds redundant literals in conflict clauses and removes them, a similar approach had to be taken as it also relies in the clause order to optimize its path through the clauses. This procedure removes literals from the conflict clause if the clause that implied them is contained in it, as shown in Figure 5.10. The procedure looks for these clauses recursively, so it can find as much redundant literals as it possibly can. It uses the order in the same way the conflict analysis does, since it skips the first literal of each clause. However, the solution cannot be the same. To adapt this procedure to our solver, we had to check, before adding a literal to the recursion stack, if it was the same literal that lead us to that clause. By skipping it and not adding that literal to the analysis stack again, we avoid the loop and the procedure works as expected.

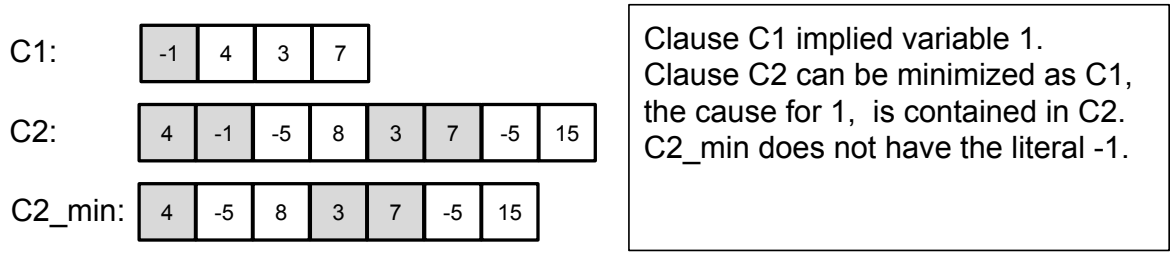

Figure 5.10: Literal (-1) is removed from clause C2 as its implying clause, clause C1, is contained on it.

Another place where MINISAT [22] needs to be altered is in the database management procedure, specially during cleanups. MINISAT [22] works with a single thread and it is fine tuned to excel at doing so. However, when instead of a single search thread, we have many, MINISAT's [22] clause locking mechanisms cease to function properly, as they once again rely on the ordering of the literals. This results in the elimination of clauses that are part some search thread reasoning. This adaptation is a two step process. The first step is to prevent clauses that belong to reasonings from being removed. This is achieved by storing the clause pointers they are using, and checking if those clause pointers are stored when trying to remove clauses. The second part is when we actually clean the database. MINISAT [22] cleans the database by creating another, empty, database where it will copy the clauses to. This means that we will

have all clauses with new clause pointers.

To be able to update our stored pointers accordingly, we had to intercept and modify MINISAT's [22] clause allocation routine so it would return us the new position. As we stored the two pointers, the new and the old, in a hashtable, we just need to replace the pointers stored in each search thread's state with the value returned by the hash table for the pointer they currently hold.

Component	Action	Comments
Clause Database	Customized	Adapted to enable handling of multiple states and added LBD heuristic
Propagation Routine	Removed	Not needed as it is executed in the GPU
Decision Routines	Removed	Not needed as decisions are now done in the GPU
Conflict Analysis	Customized	Customized to remove state dependency and to use GPU state
Clause Minimization Routine	Customized	Customized to remove state dependency and to use GPU state

Table 5.3: Minisat components and customizations.

These are the modifications that enable MINISAT [22] to hold several search states instead of just one, like it usually does. To actually transform MINISAT [22] in a beacon of search threads, some additional steps are required. The first, as we do the search in the GPU, is to modify the main loop to skip the procedures that make up search, the literal selection and unit propagation routines. After that, we replace these two routines with a kernel call, that will launch the GPU which will in turn, run the search kernel. After the launch, when the focus returns to the CPU, we need to verify if the search is over before doing conflict analysis, so we look for solutions, SAT, or UNSAT in all the returned states. If a solution is found, we return it, if not, we proceed to conflict analysis. After conflict analysis we backtrack each state and we add the conflict clause to the problem. If that is the case, a clause database cleanup is performed. After this, we return to the beginning of the search loop, where we regenerate the problem, send it back to the GPU, and we re-launch the search kernel.

The conflict analysis and backtrack steps can be performed regardless of the number of search threads that are running in the GPU, and so can the solution checking routine. So when adding search threads we had to decide which approach to follow: a divide-and-conquer approach or a portfolio of search threads. Initially we implemented a divide-and-conquer solver. However, we spent a lot of time doing load balancing, because some threads would finish their search instantly, while others would need a lot of time. To mitigate this problem, we chose to only do load balance when a restart was triggered and to make sure that restarts happened often enough, we opted to use the luby restart strategy as it restarts faster. To optimize the initial work division we followed an approach similar to Martins, et al. where the author initially uses a single thread. This way instead of doing a blind division, the division is made with respect to the order provided by VSIDS after this warm up procedure [15].

In the end we decided to change our solver to a portfolio approach, mainly for two reasons. First, a portfolio approach eliminates the necessity of load balance. Second, with hundreds of search threads, having different approaches was crucial to succeed. However, it is hard to come up with hundreds of different combinations of heuristics, so we followed the approach used in PLINGELING [30] where they do not use different heuristics, instead, they add diversity by randomly attributing different initial variable selection orderings to each search thread [30]. So to accommodate hundreds of different solvers, we chose different restart heuristics and polarity modes, and we gave each solver a different initial variable ordering. The selected restart heuristics and polarity modes are shown in Table5.4.

Restart Heuristic	Description
Exponential	Increases the restart interval exponentially
Luby	Restarts with intervals that respect the Luby sequence
LBD Restart	Dynamic strategy that restarts when the search deviates from a solution
Polarity Saving Mode	**Description**
Total Phase saving	Saves the last implied polarity
Partial Phase saving	Saves the polarities implied in the last level of assignment
Negative Polarity	Always assigns negative values to the variables
Positive Polarity	Always assigns positive values to the variables

Table 5.4: MinisatGPU's heuristic selection.

When running a portfolio solver, the way we handle learnt clauses differs from the usual approaches. Normally, solvers would only share clauses that were considered important. They do this because sharing all clauses would cause an overhead that would make the benefits of clause sharing irrelevant [14]. However, we do not face this problem since all search threads are halted while conflict analysis and clause database management is being done. This way, we are able to have only one clause database that receives clauses from all search threads. Another reason for this decision is that there is not enough space in the GPU to have a clause database for each search thread, so frequent database cleanups would be required, generating additional overhead. Moreover, if we had a clause database per thread, the overhead of sending the databases to the GPU would increase, as we would have to send hundreds of different databases. Furthermore, having only one clause database for all threads will result in an highly optimized set of clauses. Since we did not modify the limits that trigger a cleanup, it results in frequent cleanups that will guarantee that only the best learnt clauses will survive this process.

5.2.2 GPU Side

Even with all the changes made to MINISAT [22], further adaptations were necessary on the GPU side so they could work together. We started with the DPLL solver described in the previous chapter and we stripped it down until only the search loop, which only makes decisions and propagations, was left. To make the two sides work together, we needed additional information:

- the **trail**, where the assignment order is stored;
- the **trail limits**, which stores the index where each level ends in the trail;
- the **assignment level** itself;
- other **variable information**, their assignment, the clause that implied them, and the implication level they belong to.

This information needed to be updated during the search routine. However, much of the information regarding variables was already stored as it was needed to do conflict analysis in the previous solution. The adaptation was in the location where we stored it. Nevertheless, in the previous implementation there was neither trail nor trail limits, and no information about implication levels.

In the initial implementation, the search routine analyzed all clauses every time a literal was picked or that an implication was made. To maximize the usage of the GPU, we only analyzed the clauses that we needed to, so it would do a more fine grained processing. This let us use less threads in each block, and it enabled us to use the remaining resources to run more search threads. The search routine itself also changed. Initially we had a three step search, where the first step was the variable selection, the second was propagation, which stored the result in an auxiliary location, and the third was to make these results effective. The last two steps were repeated until no more propagations were made, and another decision had to be made. We did this because there are no concurrency routines, like mutexes, that let us do several, isolated, steps at a time.

However, this third step was eventually removed since we modified the procedure to let us perform these operations atomically. To do so, we reduced the race condition to one atomic operation, compare-and-set, since it will only write the memory position if the value, that the position currently holds matches the value being compared. The operation returns the value that was previously on memory. This enabled us to do concurrent operations since, when performing the implication, it will produce one out of three outcomes:

1. if the returned value is *unassigned* which is the value we are comparing against. If this happens, it means that we are the ones performing the implication.
2. if the returned value matches the value we were going to write, this means that we lost the race condition. However, we have no conflict as we were going to assign the same value.
3. if the returned value is not *unassigned*, and does not match the one we were going to assign, it means we have a conflict.

This change, although it slightly increases code divergence, results in a better performance since the third step, the confirmation step, would have to check all variables for new information. This is an

issue for two reasons. First, we had additional memory accesses. Second, in addition to these accesses, when the problem at hand had thousands of variables, this step was extremely costly, as it was repeated several times during search.

Our next step was to increase performance in the clause analysis procedure. At this time, the procedure would process the whole clause unless one of two conditions was found: 1) there was a satisfied literal in the clause; 2) there were more than one unassigned literals in the clause. If this was not the case, there were, again, two options. The first was that all literals were falsified and so a conflict was found. The second was that only one literal was unassigned, meaning that the clause was unit. This would trigger an implication. Each implication has two steps. The first, as explained before, is an atomic operation that guarantees safety in race conditions. The second part of the implication process is when the implication is made effective. This is achieved with two further atomic operations, one to increase the implication counter and another one to get the implication's place in the trail. With these two values we can now store the information knowing that no further race conditions will occur.

The first optimization we made to this procedure was to take advantage of the asynchronous memory accesses in the GPU. To analyze a clause we need two pieces of information. The first is the clause itself, that will be in contiguous positions in memory. The second is the value that each variable holds, associated with the literals that compose the clause. To read these values in the most optimized way, we fetched four literals at a time. As all literals are stored in contiguous positions, when we fetch the first, the others will be cached. This means that we fetch all four literals, for the price of the first, since cache access times are negligible in comparison with global memory accesses. After that, we fetch the four assignments in succession. Therefore, we can start processing the first values while the asynchronous accesses are completing for the other assignments. This procedure is depicted in Figure 5.11.

```
1 || //the arrays lit[] and value[] are allocated in registers
2 || //we read the four positions in one access
3 ||   lit[0]=clause[i];
4 ||   lit[1]=clause[i+1];
5 ||   lit[2]=clause[i+2];
6 ||   lit[3]=clause[i+3];
7 || // we get asynchronously get the assigned values and store them in value.
8 ||   value[0]=assignments[var(lit[0])];
9 ||   value[1]=assignments[var(lit[1])];
10 ||  value[2]=assignments[var(lit[2])];
11 ||  value[3]=assignments[var(lit[3])];
12 || // further operations over the values
```

Figure 5.11: Using Cached reads and asynchronous memory operations.

This optimization addresses one of the most crucial performance issues of SAT solvers in the GPU: low memory locality. When we read the clause, since the positions are sequential in memory, we can take advantage of caches to speed up the process. However, when we read the values assigned to the variables, the information will most likely not fall in the same cache lines. Therefore, these accesses will be issued repeatedly until all threads have fetched the value they need. In average, these instructions

are issued eight to nine times, per half-warp, instead of only one. This way, we can reduce the latency problems by performing computation that is not dependent on this data, since we increase the time between the read and the usage.

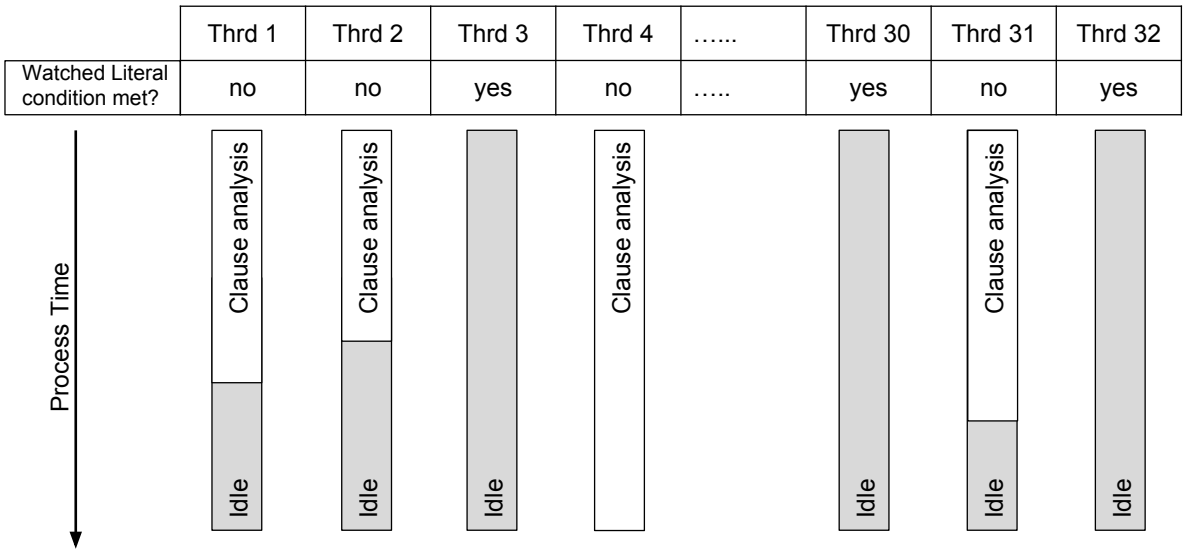

Figure 5.12: Idle time in Watched Literal procedure on the GPU. All threads must wait until thread 4 completes it's analysis.

In order to further optimize this procedure, we implemented watched literals in the GPU. However, verifying the two literals was not optimized in relation to memory locality, so the benefits of using them, were hidden by memory latency. Furthermore, the fact that in the same warp, all threads must wait for the others to finish the work in a clause, means that if one thread fails to fall into the watched literal conditions, there will not be a noticeable difference in performance for that particular round of analysis, as shown in figure 5.12.

5.2.3 GPU/CPU cooperation

The last part of the implementation was to actually connect the GPU with MINISAT [22]. The cooperation was made mainly through information exchange. Therefore, optimizing memory transfers and postponing some work in the CPU to be done in parallel with the GPU, is crucial to achieve optimal performance.

To achieve this, it is important to move to the GPU only the data that is necessary, as these transfers add an overhead to the solving process. In the first implementation, we used pinned memory only. Pinned memory, as said before, creates a shared memory environment where the CPU can access GPU memory and vice versa. We did this because we needed to transfer the information back and forth every

time, before and after the kernel launches. However, when we scaled the number of search threads, the memory accesses became the main performance bottleneck. We solved this mainly by moving the clause database to the GPU.

This solution performed better, but we could not send the problem to the GPU before every call, because of the overhead that the successive memory copies posed. This resulted in a suboptimal learning process, since we used MINISAT's [22] activity based heuristic and the majority of clauses was eliminated without ever going into the GPU, as they never became active. We thought that this rating method was unfair and we set to find a better clause rating heuristic. The heuristic we selected was the LBD rating method, described previously in Chapter 2. We resorted to this heuristic mainly due to the fact that it does not require usage to provide us with an estimation of the clause's worth. With this new heuristic we could still maintain fairness during clause management, even though we were not sending the clauses in every launch.

With the hopes of reducing the time needed to send the clause database to the GPU, we went on to improve the running time of this process. The time was dominated by the successive calls to the memory copy (Application Programming Interface) API, thus, instead of copying each clause directly to the GPU, we modified the procedure to copy the clauses to an auxiliary buffer, instead. With the aid of this buffer we could copy the entire database in one call, making this operation extremely efficient. Furthermore, we noticed that these copies, however small, were still taking some time to complete, and since there was work to be done after them, we started using Asynchronous memory copies. These asynchronous operations are only asynchronous in respect to the CPU and will maintain order with respect to other GPU operations, so no synchronization errors can occur.

Another way we used asynchronous operations, between the GPU and the CPU, was to reduce the effects of ordering all the variables with respect to the VSIDS weights. As discussed before, with hundreds of search threads, we have hundreds of sort operations. Instead of sorting these variables before launching the kernel, we delay the sorting process and we perform it after the kernel launch. This way, we can search in the GPU, while sorting these variables in the CPU. Another thing we can also sort in parallel with the GPU is the clause database. If we do that, we can skip the sorting process when the time database cleanup is triggered. This optimization works because clauses added in the last round of search, will be locked, as they will be part of the reasoning process of a search thread, and will never be removed regardless of their place in the clause list.

With this, we end up with a system that tries to use the GPU and the CPU cooperatively in the best possible way. We will leave to the next chapter an evaluation of our solver against solvers that use the CPU exclusively and solvers, that like the one proposed in this thesis, use both cooperatively.

Chapter 6

Evaluation

To evaluate our solver we compared it against the state-of-the-art in SAT solvers that use the GPU. In addition, we also tested it against MINISAT, the base solver for our solution, and GLUCOSE, from which we took the clause rating mechanism and a restart heuristic. The test suite is a subset of well known problems taken from SATLib[1], a popular repository of SAT problems, with the exception from the gus-md5 set, which was taken from the SAT Competition 2009[2] problem set.

We chose to test our system, MinisatGPU, against CUD@SAT, a GPU based sat solver. CUD@SAT was the most mature and most efficient GPU-based solver we found. Furthermore, it has several configuration options, some of them mimicking other authors' approaches. Of all the possible configurations, we chose three, two that are proposed in the projects website, and one that, although based on a proposed configuration, tries to perform much of the work in the GPU. We will only present the best time of the three configurations used. The configurations of CUD@SAT we used were:

- top part of the search tree is handled in the CPU, the GPU handles the bottom of the tree when 50 variables or less are left to propagate.
- top part of the search tree is handled in the CPU, the GPU handles the bottom of the tree when 150 variables or less are left to propagate, using more GPU during the search.
- only the propagation is ran in the GPU, the rest of the algorithm runs in the CPU, this mode follows the approach taken by Fujii et al [35].

It is important to note that, even though the last configuration follows the approach used by Fujii et al. [35], it has two improvements over it. First, it does not limit the solver to 3-SAT problems, which are

[1] http://www.satlib.org/
[2] http://www.satcompetition.org/2009/

problems in which all clauses must have 3 literals. Second, CUD@SAT features conflict analysis and clause learning.

As not all solvers featured a problem preprocessor, we first ran all SAT problems through SATELITE [28], with the option for very expensive optimizations on. We did this to ensure all solvers were running on the same simplified version of each problem. After that, we turned the built in problem preprocessors off to prevent further optimizations, and time loss. We ran all solvers with a time limit of 600 seconds, except on hole10, a hard combinatorial problem, which was ran with a 1000 seconds time limit. We increased this limit, in order to show how our solver compared with the others in this specific problem, as our solver surpassed the 600 seconds limit.

All tests were ran in a machine that is equipped with an Intel Core i7-950 CPU, with 12GB of RAM, coupled with an nVidia Tesla c2050 GPU with 3GB of on-chip memory.

We chose five sets of problems to evaluate the solvers, which are described in Table 6.5. We decided to use these problems, as they are a good sample of the behavior of the solvers in other problem sets from SATLib.

Type	Name	Description
Industrial	bmc-ibm series	Bounded module checking series taken from real industrial hardware designs, a contribution from IBM
Industrial	bmc-gallileo series	Bounded module checking series taken from real industrial hardware designs, a contribution from Gallileo
Encodings	hole series	Encodings of pigeon hole problems to SAT instances
Encodings	parity series	Encodings of problems to learn parity functions.
Encodings	gus-md5	search for md5 hash colisions

Table 6.5: Problem sets.

6.1 Comparison with GPU-enabled Solvers

We started by evaluating our solver against CUD@SAT, because it shows that our approach can match, or beat, the best option of this solver. We ran our solver with the default configurations (Table 5.2), against all three configurations of CUD@SAT, after which we proceeded to selected the best time.

As it can be seen in Table 6.6, our solver is very competitive with CUD@SAT in small problems, and both solvers struggle with hard, combinatorial problems like hole10. However, when it comes to big problems, but not necessarily hard problems, CUD@SAT cannot compete with us as they do not scale well.

Because they analyze all clauses, they limit the scope of their solver to smaller problems, where the capacity of the GPU is not exceeded. By carefully selecting the clauses we wish to propagate, we end

Problem	Variables	Clauses	MinisatGPU	CUD@SAT (best)
bmc-gallileo-8	18045	128581	13.96	>600secs
bmc-gallileo-9	20560	148628	17.28	>600secs
bmc-ibm-10	11771	103921	19.47	>600secs
bmc-ibm-11	8609	57399	16.90	>600secs
bmc-ibm-12	13781	104087	40.65	>600secs
bmc-ibm-13	3656	31137	3.99	>600secs
bmc-ibm-1	2582	24046	4.17	>600secs
bmc-ibm-2	77	484	0.39	0.07005
bmc-ibm-3	5411	39958	5.89	>600secs
bmc-ibm-4	3980	30904	3.77	>600secs
bmc-ibm-5	893	6416	0.72	>600secs
bmc-ibm-6	5432	92790	8.66	>600secs
bmc-ibm-7	533	3324	0.23	>600secs
hole8	63	288	9.27	4.90
hole10	99	550	672.71	>1000secs
gus-md5-04	25778	159074	17.23	>600secs
gus-md5-05	25979	160164	29.79	>600secs
gus-md5-06	26038	160565	55.50	>600secs
gus-md5-07	26082	160822	160.09	>600secs
par16-1-c	173	1100	0.63	>600secs
par16-1	181	1112	0.49	0.58
par16-2-c	190	1228	1.18	32.50
par16-2	204	1241	1.99	0.97
par16-3-c	180	1168	1.66	>600secs
par16-3	195	1181	2.10	2.37
par16-4-c	174	1128	0.88	360.91
par16-4	190	1140	0.30	6.96
par16-5-c	184	1196	2.37	35.00
par16-5	200	1209	2.21	30.67

Table 6.6: Results against the best configuration of CUD@SAT for each problem.

with a kernel that is not as GPU-friendly. However, we can scale to problems of any size as usually the ratio of clauses per variable remains low (i.e. lower than 192, our default parameter for number of threads in a block). Therefore, when CUD@SAT wants to scale to accommodate more clauses, they allocate more blocks, whereas when we allocate more blocks we increase both the performance and diversity of our solver. Moreover, they do not have a way to increase performance by adding more blocks, or scaling to more GPUs, as these new blocks, or GPUs, would have no work to do. Table 6.7, shows the effect of adding more blocks to our solver in three different problems.

6.2 Comparison with CPU-based Solvers

With our new way of using the GPU, we can scale to problems of any size, which was not possible with previous approaches, while remaining competitive in problems of smaller dimensions.

Problem	Blocks	Time (sec)
gus-md5-04	1	60.88
gus-md5-04	8	24.74
gus-md5-04	64	15.20
gus-md5-04	128	17.61
gus-md5-04	256	20.49
gus-md5-07	1	2451.28
gus-md5-07	8	436.46
gus-md5-07	64	157.66
gus-md5-07	128	134.40
gus-md5-07	256	139.93
bmc-ibm-12	1	637.64
bmc-ibm-12	8	138.28
bmc-ibm-12	64	48.57
bmc-ibm-12	128	42.01
bmc-ibm-12	256	47.31

Table 6.7: The effect of the addition of more blocks to the runtime of the solver.

We, then, compared our solver with MINISAT, the solver we based our solution on, and GLUCOSE, from which we took the clause rating and restart heuristics. We used the same problem sets, and all three solvers were tested with their default parameters, except for MINISAT, where simplification was disabled, as all problems are already simplified.

Even though we have a highly competitive solver against other GPU-enabled implementations, there is still work to be done when it comes to competing with CPU-based implementations (see Table 6.8). This difference in performance can be attributed to two reasons, first, MINISAT and GLUCOSE are highly optimized to run a single thread and do not have to deal with the overhead that the GPU introduces. Tests we made show that it takes around 1 second to launch 100.000 empty kernels if we synchronize the CPU and the GPU afterwards, which we need to do on our solver. This is relevant to the applicability of GPUs to SAT, as both MINISAT and GLUCOSE exceed this number in conflicts per second in several problems. In addition to this synchronization step, there is still overhead in moving clauses and other important data from CPU's memory to the GPU. This limits the applicability of GPUs to larger problems, where these overheads are more negligible.

One way to compensate for these overheads is to do more work in each kernel call, by using more blocks. However, as seen in Table 6.7, at a certain point, adding more blocks, will actually hurt performance, limiting the extent at which we can compensate for these overheads. For instance, when we ran our solver with the *hole8* problem, with 1 block only, it spent 83.66 seconds in CUDA calls, out of a running time of 129.59 seconds (this excludes kernel running times). However, when we ran it with 128 blocks, we got a running time of 6.82 seconds with only 1.11 seconds of overhead. This happens because in each kernel call, more work will be done, which will be shared through clause learning. Since all search

Problem	Variables	Clauses	MinisatGPU	Minisat	glucose (v2.3)
bmc-gallileo-8	18045	128581	13.96	0.25	0.05
bmc-gallileo-9	20560	148628	17.28	0.27	0.12
bmc-ibm-10	11771	103921	19.47	0.28	0.12
bmc-ibm-11	8609	57399	16.90	0.32	0.09
bmc-ibm-12	13781	104087	40.65	0.98	1.04
bmc-ibm-13	3656	31137	3.99	0.48	0.45
bmc-ibm-1	2582	24046	4.17	0.09	0.03
bmc-ibm-2	77	484	0.39	0.01	0.01
bmc-ibm-3	5411	39958	5.89	0.09	0.04
bmc-ibm-4	3980	30904	3.77	0.09	0.03
bmc-ibm-5	893	6416	0.72	0.02	0.01
bmc-ibm-6	5432	92790	8.66	0.33	0.05
bmc-ibm-7	533	3324	0.23	0.01	0.01
hole-8	63	299	9.27	0.43	4.90
hole-10	99	550	672.71	211.46	71.66
gus-md5-04	25778	159074	17.23	1.41	0.98
gus-md5-05	25979	160164	29.79	4.24	3.43
gus-md5-06	26038	160565	55.50	8.55	15.84
gus-md5-07	26082	160822	160.09	24.41	54.50
par16-1-c	173	1100	0.63	0.06	0.06
par16-1	181	1112	0.49	0.04	0.02
par16-2-c	190	1228	1.18	0.12	0.08
par16-2	204	1241	1.99	0.14	0.06
par16-3-c	180	1168	1.66	0.07	0.06
par16-3	195	1181	2.10	0.08	0.05
par16-4-c	174	1128	0.88	0.02	0.01
par16-4	190	1140	0.30	0.01	0.04
par16-5-c	184	1196	2.37	0.09	0.03
par16-5	200	1209	2.21	0.04	0.03

Table 6.8: Results against Minisat and glucose.

threads share the same clause database.

However, the challenges do not end with the CPU-GPU interoperability issues. In the GPU, the access pattern to memory is suboptimal. Even if we can optimize clause fetching, which we do, we still cannot optimize the verification of each assignment. This second access will have a high probability of not being coalesced or cached. Therefore, it will result in the repetition of the read instruction, which are issued on average 8 to 9 times, a value that is close to the maximum value of 16 repetitions[1]. This is the main issue in trying to solve SAT with the GPU. SAT solving is memory intensive and this poor access pattern renders our kernel latency bound, since we spend 80% of the execution time of each kernel waiting for the repetition of these memory operations to end.

[1] memory accesses are handled with the granularity of an half warp, or 16 threads

Chapter 7

Conclusion

In this thesis we have presented a new method to use GPUs to solve SAT problems. Our method uses three key insights: 1) pick clauses carefully and run a search thread in a single block, instead of using the whole GPU to run a single search thread; 2) run the entire search in the GPU to reduce focus changes; and 3) scale by occupying the rest of the GPU with more search threads. Using these ideas, we built a solver, MinisatGPU that implements a portfolio of search threads that are ran in parallel in the GPU, while conflict analysis is done in the CPU to avoid code divergence in the GPU.

The evaluation of MinisatGPU shows that it scales well to big problems, when compared to other GPU solvers, and it can solve problems that were once unsolvable by GPU based solvers. However, when compared with CPU based solvers, it is still behind in terms of performance. This is due to poor memory access patterns during the solving process that becomes dominated by the latency of instruction repetition. Therefore, while we believe that with new algorithms, GPUs can be an alternative to CPUs, with current SAT and GPU technology, CPU-only solvers will, most likely, have the upper hand for the foreseeable future.

7.1 Future Work

We believe our solver can be improved in two ways. First, when we return from kernel execution, the GPU is left waiting until the next round of search arrives. Therefore, the improvisation would be to try to capitalize on this and use the GPU while the CPU is busy doing analysis, by launching a second search kernel when the first returns to the CPU. However we did not invest in this because we do work in the CPU in parallel with the GPU meaning the CPU is busy most of the time. The other improvement is made upon the first, by using different streams, that are basically a way of telling the GPU which

operations, memory copies or kernels, can be executed in parallel and which cannot. Using streams, we can hide the time we spend transferring the clause database, by doing work in the GPU at the same time. However, this can only be achieved with two, or more, kernels.

While we believe that these improvements can bring performance gains, we do not believe that these gains will allow GPU-enabled solvers to surpass the performance of CPU-only solvers.

Bibliography

[1] Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)

[2] Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the third annual ACM symposium on Theory of computing. STOC '71, New York, NY, USA, ACM (1971) 151–158

[3] Karp, R.M.: Reducibility among combinatorial problems. In Jünger, M., Liebling, T.M., Naddef, D., Nemhauser, G.L., Pulleyblank, W.R., Reinelt, G., Rinaldi, G., Wolsey, L.A., eds.: 50 Years of Integer Programming 1958-2008. Springer Berlin Heidelberg (January 2010) 219–241

[4] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7) (July 1962) 394–397

[5] Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3) (July 1960) 201–215

[6] Goldberg, E., Novikov, Y.: BerkMin: a fast and robust sat-solver. Discrete Applied Mathematics **155**(12) (June 2007) 1549–1561

[7] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th annual Design Automation Conference. DAC '01, New York, NY, USA, ACM (2001) 530–535

[8] Marques-Silva, J., Sakallah, K.: GRASP: a search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5) (1999) 506–521

[9] Gomes, C., Selman, B., Crato, N.: Heavy-Tailed Distributions in Combinatorial Search. (1997)

[10] Lewis, M., Schubert, T., Becker, B.: Multithreaded SAT solving. In: Proceedings of the 2007 Asia and South Pacific Design Automation Conference. ASP-DAC '07, Washington, DC, USA, IEEE Computer Society (2007) 926–931

[11] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proceedings of the 21st international jont conference on Artifical intelligence. IJCAI'09, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2009) 399–404

[12] Zhang, H., Bonacina, M.P., Paola, M., Bonacina, Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation **21** (1996) 543–560

[13] Böhm, M., Speckenmeyer, E.: A fast parallel SAT-Solver - efficient workload balancing. In: Annals of Mathematics and Artificial Intelligence. (1996) 40–0

[14] Chrabakh, W., Wolski, R.: GrADSAT: A Parallel SAT Solver for the Grid. (2003)

[15] Martins, R., Manquinho, V., Lynce, I.: Improving search space splitting for parallel SAT solving. In: 2010 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI). Volume 1. (2010) 336–343

[16] Hamadi, Y., Sais, L.: ManySAT: a parallel SAT solver. JOURNAL ON SATISFIABILITY, BOOLEAN MODELING AND COMPUTATION (JSAT) **6** (2009)

[17] Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. SIGARCH Comput. Archit. News **38**(3) (June 2010) 451–460

[18] Gulati, K., Khatri, S.P.: Boolean satisfiability on a graphics processor. In: Proceedings of the 20th symposium on Great lakes symposium on VLSI. GLSVLSI '10, New York, NY, USA, ACM (2010) 123–126

[19] Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability. (2009) 131–153

[20] Marques, J., Silva, J.P.M., Silva, J.P.M., Sakallah, K.A., Sakallah, K.A.: GRASP—A new search algorithm for satisfiability. In: in Proceedings of the International Conference on Computer-Aided Design. (1996) 220–227

[21] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: In 10th International Conference on Theory and Applications of Satisfiability Testing. (2007) 294–299

[22] Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: Theory and Applications of Satisfiability Testing. Number 2919 in Lecture Notes in Computer Science. Springer Berlin Heidelberg (January 2004) 502–518

[23] Audemard, G., Lagniez, J.M., Mazure, B., Saïs, L.: On freezing and reactivating learnt clauses. In: Proceedings of the 14th international conference on Theory and application of satisfiability testing. SAT'11, Berlin, Heidelberg, Springer-Verlag (2011) 188–200

[24] Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Information Processing Letters **47** (1993) 173–180

[25] Huang, J.: The Effect of Restarts on the Efficiency of Clause Learning. (2007)

[26] Brafman, R.: A simplifier for propositional formulas with many binary clauses. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics **34**(1) (2004) 52–59

[27] Lynce, I., Marques-Silva, J.: Probing-based preprocessing techniques for propositional satisfiability. In: 15th IEEE International Conference on Tools with Artificial Intelligence, 2003. Proceedings. (2003) 105–110

[28] Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In Bacchus, F., Walsh, T., eds.: Theory and Applications of Satisfiability Testing. Number 3569 in Lecture Notes in Computer Science. Springer Berlin Heidelberg (January 2005) 61–75

[29] Zhang, H.: SATO: an efficient prepositional prover. In McCune, W., ed.: Automated Deduction—CADE-14. Number 1249 in Lecture Notes in Computer Science. Springer Berlin Heidelberg (January 1997) 272–275

[30] Biere, A.: Lingeling, plingeling and treengeling entering the SAT competition 2013. In: Proceedings of SAT Competition 2013, A. Balint, A. Belov, M. Heule, M. Järvisalo (editors). (2013)

[31] Manolios, P., Zhang, Y.: Implementing survey propagation on graphics processing units. In Biere, A., Gomes, C.P., eds.: Theory and Applications of Satisfiability Testing - SAT 2006. Number 4121 in Lecture Notes in Computer Science. Springer Berlin Heidelberg (January 2006) 311–324

[32] Beckers, S., Samblanx, G.D.: Parallel SAT-sovling with OpenCL. Proceedings of the IADIS International Conference on Applied Computing (2011) 435–441

[33] Mazure, B., Sais, L., Gregoire, E.: TWSAT: a new local search algorithm for SAT-performance and analysis. Proceedings of the Workshop "Studying an Solving Really Hard Problems" of the First International Conference on Principles and Practice of Constraint Programming" (1997) 1–12

[34] Mazure, B., Saïs, L., Grégoire: Boosting complete techniques thanks to local search methods. Annals of Mathematics and Artificial Intelligence **22**(3-4) (July 1998) 319–331

[35] Fujii, H., Fujimoto, N.: GPU acceleration of BCP procedure for SAT algorithms. (2012)

[36] Alessandro Dal Palu, Agostino Dovier, A.F.E.P.: Exploiting unexploited computing resources for computational logics (June 2012)

[37] Meyer, Q., Schönfeld, F., Stamminger, M., Wanka, R.: 3-SAT on CUDA: towards a massively parallel SAT solver. In: 2010 International Conference on High Performance Computing and Simulation (HPCS). (2010) 306–313

[38] Davis, J.D., Tan, Z., Yu, F., Zhang, L.: Designing an efficient hardware implication accelerator for SAT solving. In Büning, H.K., Zhao, X., eds.: Theory and Applications of Satisfiability Testing – SAT 2008. Number 4996 in Lecture Notes in Computer Science. Springer Berlin Heidelberg (January 2008) 48–62