Parallelization of SAT Algorithms on GPUs

Carlos Costa
carlos.silva.costa@ist.utl.pt

Instituto Superior Técnico

Abstract. The Boolean Satisfability Problem is one of the most important problems in computer science with applications spannig many areas of research. Despite this importance and the extensive study and improvements that have been made, no efficient solution to the problem has been found to the date.
During the last years, nVidia introduced CUDA, a platform which lets developers take advantage of the great computing power of the GPUs, that was once unavailable to use.
In this work, we propose a methodology of using GPUs to accelerate the resolution of the SAT problem. Since the problem does not map well the GPU paradigm, we will explain our approach, which uses both the GPU and the CPU to solve the parts that suit them the best.
We also propose developing a novel SAT solver, capable of solving current benchmark problems, as well as most industry problems.

Keywords: Satisfability, GPU, CUDA, parallelization

1 Introduction

The Boolean Satisfiability (SAT) problem was the first NP-Complete problem to be identified [1], and it is still one of the most important problems in computer science. The reason that makes it so important has to do with three factors: i) they are easy to understand and have inumerous applications in real life, such as software testing with model checkers, theorem proving [2] and Electronic Design Automation; ii) all NP-Complete Problems, can be reduced into other NP-Complete problems, and if reduced repeatedly, they will eventualy reduce to the SAT problem [3], and iii) despite the extensive study and effort put into developing efficient solvers, as problems grow in size, they will always become undecidable, as the time it will take to solve them, will grow exponentially.

Since the first attempt at solving SAT problems with the DPLL algorithm [4], SAT solvers have improved a lot, as important contributions have been made since then which contributed to having Sat solvers that outperform the initial DPLL algorithm by three orders of magnitude [5], and now allow problems to have tens of thousands of variables, instead of just a few.

However, modern SAT solvers are hitting a wall in terms of performance, and new techniques only make small running time improvements. Moreover, solvers are also becoming less robust as they are becoming increasingly sensitive to the parameters used in their heuristics.

At the same time, processors started to become parallel, and, as their processing power continues to increase, their per core speed came to an halt. These were the reasons that made it clear that Solvers had to keep up with the hardware and become parallel aswell.

To make solving parallel there are two main approaches. Some solvers split the search space into parts in which they run the algorithm, while others have several algorithms racing against each other in the same search space, sharing what they learn during that search.

Meanwhile, improvements in the area of Graphical Processing Units (GPUs) made them available to do everyday computing, not only graphic processing as they used to. GPUs have hundreds of cores and a computing power that outperforms that of the Central Processing Unit (CPU) by orders of magnitude. Still, they are not used in state of the art parallel SAT solvers.

GPUs have some drawbacks as their architectures is made to execute the same instruction in different threads (SIMT), and so, for highly irregular problems, GPUs cannot perform at the maximum of their capabilities.

We are going to propose a way of using the GPU as a way of aiding the CPU achieve better results at solving SAT instances. To do so, we are going to propose a way to regularize the SAT problem, and we will use the CPU to tackle the parts of the problem that cause that irregularity instead of having them made on the GPU.

To demonstrate our results we will run standard benchmarks against our project and compare results with some state of the art solvers. As SAT solvers are in the center of some model checkers we propose to use our solver in an open source model checker to compare results in a real life application.

In section 2 we will propose the project goals and in section 3 we will explain the architecture of the GPU as opposed to the CPU. In section 4 we will discuss some work that is relevant to our project and in the last three sections we will present our proposed solution, a way to evaluate it and our conclusions.

2 Goals

Our main goal is to explore ways to parallelize the execution of SAT solvers with special focus on the use of GPUs.

The major goals of this project are:

- The proposed solution has a performance comparable with other SAT solvers
- The solution is configurable (number of GPUs, number of GPU cores,...)
- The system provides statistics and metrics to evaluate performance
- The system is easily combined with existing projects via some interface

3 Background

In this section we will introduce the main concepts used throughout this work. Firstly, we will explain what is the SAT problem. Secondly, we will describe one

of the first algorithms used to try and solve it, the DPLL algorithm. We will, finally, talk about the workings of the GPUs, their memory model and explain in what they differ in relation to the CPUs.

3.1 The Boolean Satisfiability Problem

The Boolean Satisfiability Problem refers to the problem of, given a boolean expression, determining if there exists an assignement, true or false, to all boolean variables that make the entire expression to be true.

As every boolean formula can be transformed into it, usually the formulas are expressed in Conjunctive Normal Form (CNF), which is a conjunction of clauses where each clause is a disjunction of literals. These are easier to understand, parse and process than regular forms, and allow for some simplifications to be made [6].

There are some problems associated with SAT, like 3-SAT, or the more generic k-SAT problem, where all the formulas have the same size. Our Solver will accept any problem, as long as it is in CNF form.

3.2 Incomplete Solvers

A subset of the SAT Solvers, designated by incomplete solvers employ greedy and random approaches to find a solution, and unlike complete solvers, do not explore the entire search space. Since they are incomplete they can only prove satisfiability.

Since we want to know if the problem is Satisfiable or Unsatisfiable, we are only going to study complete solvers.

3.3 DPLL Algorithm

Most complete solvers are based on the DPLL algorithm.

The DPLL algorithm [4] was proposed in 1962 by Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland and was a refinement of the Davis-Putnam algorithm that preceded it. The algorithm tries to iteratively perform four operations to try to combine all the variables into an satisfiable solution, the said operations are unit propagation, pure literal elimination, free literal search and backtrack.

The unit propagation procedure, tries to generate implications by analyzing unit clauses. To do so, it searches for clauses that are unit, i.e. non-satisfied clauses in which all, but one literal, is assigned, meaning that the last unassigned literal must evaluate to true, thus avoiding a conflict.

Conflicts can appear in two ways, if all the variables in a clause are assigned and yet the clause is unsatisfied, or, if during the propagation phase, different values are assigned to the same variable.

The pure literal elimination searches for variables that only have one polarity in the formula, positive or negative, and so by assigning the value that makes the

literals associated with the variable true, one can remove the clauses containing them from the problem.

```
DPLL(S) ≡
while (1)
    decideNextBranch()
    while (1)
        status = propagate();
        if (status == CONFLICT)
          then
            lvl = backtrack(btlevel);
            if (lvl == 0) then returnUNSATISFIABLE fi
          else if (status == SATISFIABLE) then returnSATISFIABLE fi;
          else break;
```

In the algorithm, the decision for the next variable works the following way: the program searches for the first unassigned variable and sets it to true; then, if conflicts occur and the search backtracks back to it, the value is flipped to false. If the backtrack procedure gets to the variable again, the variable is unassigned and the search backtracks to the previous level.

The backtrack phase, in addition to the work described earlier, needs to undo other things, such as the variables assigned in the unit propagation procedure and also needs to replace the clauses eliminated in the pure literal elimination phase. If the backtrack procedure backtracks to a level below the first assignement, it means that no assignement exists to the problem and it is deemed unsatisfiable.

The other ways the algorithm may end are by satisfying all clauses, meaning the problem is satisfiable, or by time limit, or program termination, in which no conclusions can be made and the problem is undecidable.

3.4 CUDA

CUDA is a programming framework for machines with GPUs, composed of a programming language, a compiler and a runtime environement. It enables the GPU, whose processing power gap is growing larger each year when comparing with the CPUs (Fig. 1), to do generic-propose computation on preferencially data parallel, intensive processing problems, where before one could only do graphic processing.

When programming with CUDA, the CPU is optimized for fast single thread execution, so it is good for complex control logic and out of order execution, it has a large cache, to hide RAM accesses, and the cores are optimized to take advantage of those caches. The GPU is optimized for high multi-thread troughput however, they are not good at handling conditional execution flow splits, and their caches are small, so the way to hide memory access times is to work with a lot of threads.

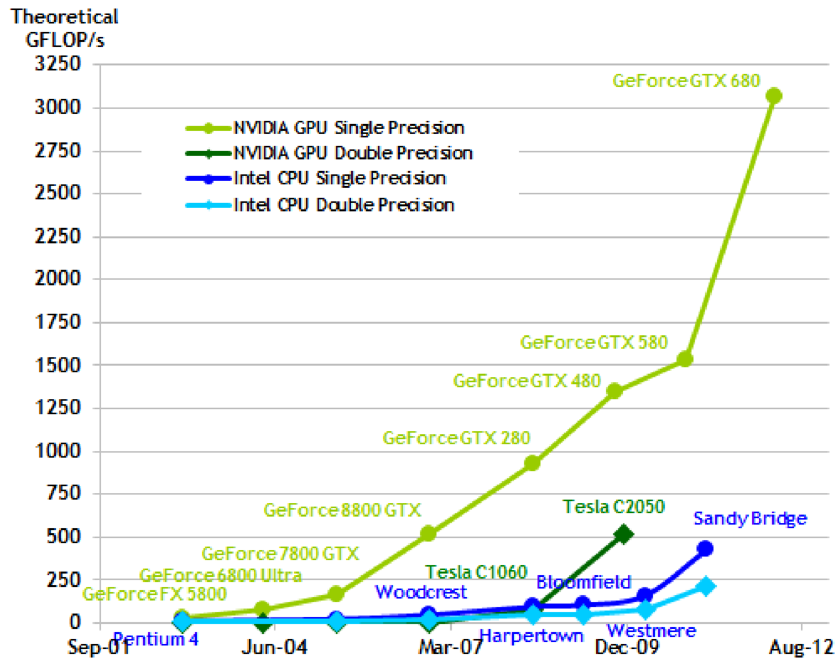

Fig. 1. The increasing processing power of the GPUs in comparison with CPUs

The hardware architecture (Fig. 2) of CUDA enabled GPUs shows why this happens: the GPU devotes most of its transistors to data processing as opposing to the CPU, which devotes them to data and instruction caching.

3.4.1 CUDA programming model

The CUDA programming model is called Single Instruction Multiple Threads (SIMT) and it has some advantages and some drawbacks when compared to Simultaneous Multi Threading (SMT), the programming model of multi-core CPUs. The advantages are that it has that a much higher number of threads that can be ran at the same time, enabling a cheap high-troughput, high-latency design. This allows the creation of cheap thread registers, and so each thread may have up to 63 registers.

CUDA's memory accesses are very expensive. So it relies on having enough threads running, because thread switching in CUDA is really fast and when a thread is reading from memory, CUDA switches to a thread that hopefully is ready to work.

This is also true for the CPU, where threads can be used to hide some memory latency. The difference is that in the GPU thread switching is very cheap and so, one can and should, have thousands of threads running at the same time.

These threads are organized in one or two dimensional thread blocks, meaning each CUDA program can have thousands of blocks with thousands of threads.

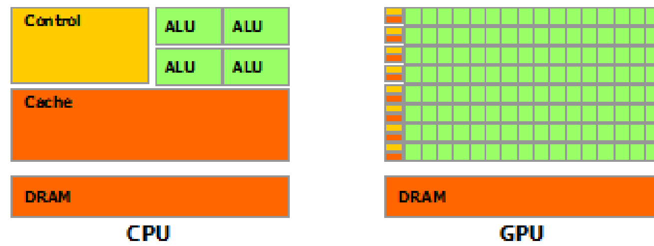

Fig. 2. The difference in architecture between the CPU and the GPU

The blocks, are then organized in Grids, which can also be organized in one or two dimensions. Inside a block, threads are organized into warps, the size of a warp is hardware specific and all the threads in a warp are executed at the same time. This means that if one thread is stalled reading from memory, the other threads in the warp, are also stalled waiting for it.

3.4.2 CUDA memory model

As said before, in the GPU, memory accesses have a high cost, and to hide this cost, a lot of threads should be used. Despite its slowness, there are four memory levels in CUDA with different peculiarities and access times.

Each block has several registers, accessible only by one thread, and so the number of registers per thread depends on the number of threads per block. The registers are very fast, and so increasing the number of threads per block reduces the number of registers available, being the variables stored in local memory, which is as slow as global memory, possibly hurting the overall performance.

There is one type of memory accessible, and shared, by all threads in a block, shared memory. In terms of speed it is as fast as registers but, unlike registers, shared memory has a major drawback, has it is is splat in 16 banks, and different banks can be accessed in parallel. However, if the same bank is being accessed by different threads, the access to that bank is serialized, making the warp of threads wait until all the accesses to that bank are made.

The other memory level, is global memory. Allocated items, go into global memory, and it is the slowest type of memory available. However, if the accesses of a warp are made to consecutive positions of it, they all happen at the same time, this is called coalesced access to memory. The opposite also happens, if the accesses are random, several accesses need to be made to the memory to fetch all the necessary information.

4 Related Work

4.1 Overview

There are several approaches to SAT solving. Here we discuss the main approaches and the most recent work using each approach.

4.2 Complete DPLL-Based solvers

As said before, the DPLL was one of the first algorithms proposed to solve the SAT problem. Solvers based on the DPLL algorithm are still the fastest solvers available.

Since its presentation, the DPLL algorithm has been continuosly improved and the most modern solvers perform several orders of magnitude better than the standard DPLL.

The DPLL algorithm backtracks chronologically, which means that, no effort to understand the cause of the conflict is made. GRASP intruduces non-chronological backtracking, where the solver analyzes the conflict and is capable of backtracking more than one level.

The DPLL algorithm analyzes the whole clause in order to search for unit clauses. In CHAFF and SATO novel approaches were devised to optimize this step.

The DPLL algorithm chooses the first non-assigned variable and sets it to true. Some heuristics choose variables in a different order, and such heuristics, lead to much better results in terms of running time.

In the remainder of this section we will describe several techniques implemented in modern DPLL-based SAT Solvers that led to the state of the art solvers we know today.

4.2.1 Conflict Driven SAT Solvers

When running, SAT solvers frequently generate conflicting attributions, and the same conflict may happen several times. To prevent the same conflict from happening recurrently, modern SAT solvers, find the cause of the conflict and learn it so they can avoid it for the rest of the solving process.

There are several approaches to learning, but all of them start with an implication graph. An implication graph is the graph generated with the implications that follow an assignment, during the unit propagation phase.

This implication graph is defined as follows:

1. each vertex in the grahp corresponds to a variable assignement
2. the predecessors of a vertex in the grahp are the antecedent assignements, corresponding to the variables in the unit clause that led to the implication. The directed edges from the variables in the unit clause are labeled with the name of the clause, vertices with no predecessors correspond to decision assignements.

3. Conflict vertices are added to the grahp to indicate conflicts. The predecessors of that vertex correspond to the assignements that forced the clause to become unsatisfied.
4. The decision level of an implied variable is the same as the maximum level of its antecedent variables.

With this implication graph, there are several ways to generate a conflict clause. In GRASP [7], the variables present in the graph that were not implicated in the current decision level are gathered and the conflict clause is created by inverting the current assignement of those variables.

These clauses are what enables solvers to don't redo mistakes, and that's the first step towards achieving non-chronological backtracking.

Even though clause learning provides major improvements, there are some drawbacks. First, the amount of spaces the clause database uses becomes increasingly higher, and the time it takes to process those clauses increases aswell. However there are techniques to limit the size of the clause database. Some SAT solvers learn clauses to a predefined limit and some use more sophisticated techniques. For instance, BerkMin [8] removes clauses according to an heuristic, and also according to their satisfiability.

4.2.2 Non-chronological backtracking

As said before, non-chronological backtracking and clause learning are connected, as the former relies on the later, and conflict-clauses can be used to backtrack several levels by simple analysis [9].

Once a conflict clause is created, it is analyzed and the search will backtrack to the highest level exhibited by the variables in the clause. The rationale behind this is that, until the first conflicting assignement is undone, the mistake will continue to happen. That happens, because some conflicts happen not because of assignements in the current level but because of prior assignements, this is the kind of problem that is solved with this type of analysis.

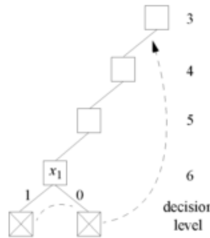

Fig. 3. With Non-chronological Backtrack the search can backtrack several levels at a time

This means that, by discarding all the branches that once needed to be explored and backjumping several levels(Fig.3), the amount of search is reduced and consequently the solving time reduces aswell.

Non-chronological backtracking is not the only scheme to backjump more than one level. For instance, Bayardo et al. [11] applied CSP look-back techniques to SAT solving, and was also able to significantly reduce search times, when compared to chronological backtracking solvers.

4.2.3 Unit Propagation

Pruning the search space can reduce the execution time of SAT solvers, but as solvers spend about 80% of their execution time doing unit propagation [5], some previous work fine tuned this procedure to its maximum.

The first attempt to optimize the propagation phase was made in SATO [12]. It introduced head-tail lists, which consists of two pointers to the first and last unassigned literals. When a variable pointed by a pointer gets negated, the pointer moves to the next unassigned variable, with the head pointer always moving towards the tail and vice versa. When the two pointers get to the same variable, that variable is an implication and needs to be propagated. If, on the other hand, no good place is found to place the two pointers, a conflict exists, and the solver needs to backtrack. During backtrack the pointers are moved to a previous position.

In Chaff [5] another approach was taken. It also has two pointers, but they behave differently. They point to either an unassigned or satisfied literal. If the pointed literal gets negated, that pointer moves to an unassigned or satisfied literal, different from the literal pointed by the other pointer. When only one unassigned literal is found, that variable is set in order to satisfy the clause, and propagated. In this approach the pointers can be anywhere in the clause, thus no backtracking is necessary.

In Fig. 4 [13] an example of the evolution of both algorithms troughout time can be seen.

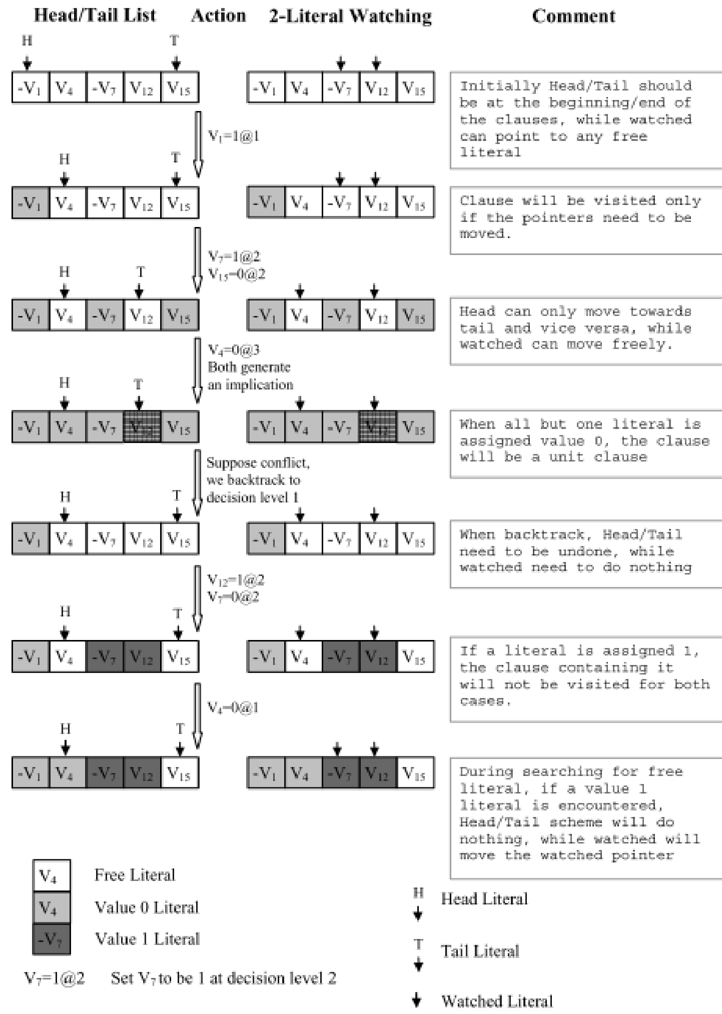

Fig. 4. The increasing processing power of the GPUs in comparison with CPUs

4.2.4 Variable Selection Heuristics

There are many decision heuristics and Marques-Silva [14] and Hooker [15] did an extensive study in the importance and impact of several of them. We are going to explain Variable State Independent Decaying Sum (VSIDS), one of the most used heuristics in modern solvers.

VSIDS Heuristic was introduced in Chaff [5] that works as follows:

1. Each variable in each polarity has an associated counter initialized to 0
2. When a clause is added to the database, the counter associated with each literal in the clause is incremented
3. The (unassigned) variable and polarity with the highest counter is chosen at each decision
4. Periodically, all the counters are divided by a constant

This strategy, by giving more importance to variables involved in recent conflicts, has a clear goal of trying to satisfy the most recently generated conflict clauses, because according to the authors, these are what drives the search process on difficult problems.

In his paper, Marques-Silva [14] relativized branch heuristics importance to SAT solvers and added that search pruning techniques like non-chronological backtracking and clause learning have a much more significant impact when the objective is to reduce running time and the amount of search.

4.2.5 Restarting the search

While solving SAT problems, as the search is processed in a depth first manner, early wrong decisions are very expensive [16].
Initially, decision heuristics cannot make good decisions because they are not well informed. In modern SAT solvers, knowledge is accumulated during conflict analysis with, previously explained, conflict-clause generation, and heuristics like VSIDS, that are updated with the addition of conflict clauses.

By restarting the search from time to time, solvers prevent the search from being stuck in a small region of the search space [5]. Also, to ensure completeness the restart interval is increased everytime so that eventually the whole search tree is traversed.

As the solver as gathered information from previous runs, it allows the heuristics to do better decisions and it allows the new search to focus on parts previously identified as important, mostly because when the search is restarted it has all the previously identified conflict-clauses so it will avoid previously generated errors, and by doing so, it will explore other paths.

4.3 Parallel DPLL based solvers

The effectiveness of the techniques described in the previous section made it difficult to improve the overall runtime of state of the art SAT Solvers by orders of magnitude. Only minor improvement were possible and with the use of

heuristics, SAT Solvers became highly stochastic, as they were very sensible to their parameters [17]. To continue to increase in performance, solvers had to become parallel to keep up with the hardware trend, where multi-core processors continue to increase in speed, but single core performance is staled.

Several approaches to parallel SAT solving have been proposed and we will proceed to explain the ones considered more relevant.

4.3.1 Divide-and-Conquer

Divide-and-Conquer is the typical method to parallelize the search. A set of constraints is used to split the decision tree and then the algorithm is ran on each part of the decision tree with a different thread or core. However there are problems with standard divide-and-conquer since the search is irregular and given so, mechanisms of work balancing are needed to make the search scale. An examples of a divide-and-conquer solver is PSATO [18], by Zhang et al. that relies only on dividing the search space and load-balancing,

4.3.2 Clause sharing

Another technique to enhance parallel SAT solving is clause sharing. By sharing the conflict clauses generated on each core, each core is able to help the others to further prune their search space, resulting in reduced runtimes [19].

The main problem with clause sharing is the large number of clauses that can be generated. A solution to this problem is to only export clauses that are smaller than a pre-defined size. This way, many of the generated clauses are not shared and the overhead is reduced. This has other, very significant, advantage. The smaller the clause, the bigger is the cut on the search space. For instance, if a conflict clause has k variables, the cut is of $2^{(n-k)}$ tuples, where n is the number of variables [20].

4.3.3 Portfolio Solvers

The above mentioned ways of parallelizing SAT solving, share one problem with serial solvers. They are not very robust, as they heavily rely on heuristics and their parameters. To address this lack of robustness, portfolio solvers make several DPLL-based solvers compete, and cooperate with each other, to be the first to solve a given instance. These solvers rather than splitting the search space, are all running in the same search space with different, decision and restart, heuristics sharing only the conflict clauses they generate.

The state of the art portfolio solver is MANYSAT [21]. It runs four DPLL instances, with four different restart policies: one solver increases the time between restarts geometrically, other increases it arithmetically and finally the last two policies are a luby sequence and a dynamic restart scheme. As for clause sharing, the scheme follows, the core that restarts dynamically sends clauses to two other cores, that only receives clauses, and the other core does not share

clauses. Moreover, for better results, MANYSAT also controls both the quantity, so learned clauses size does not grow without control, and quality of the shared clauses, by filtering the learned clauses by relevance [20].

4.3.4 Parallel Unit Propagation

As Unit propagation is where most of the run time is spent during the search process, efforts were made to parallelize it. In RISS [22], the author parallelizes the propagation process, speeding up the solving process. However, the speed is only marginally improved, mainly because of synchronization overhead, however it does not scale for more than two cores.

4.4 SAT Solving on the GPU

There are a few published works on using GPUs in SAT solving. The approaches vary a lot, going from simple matrix multiplication schemes, to DPLL based algorithms, to using incomplete approaches to optimize the variable decision heuristic.

In MESP, Gulati et al. [23] propose using SurveySAT, which is based in an incomplete method, survey propagation [24], on the GPU, and at the same time they are running a complete solver, based on MiniSAT [25] on the CPU. Basically they enhance the VSIDS selection heuristic, which, as explained before, is based in variable activity, with the output of the survey propagation algorithm that is running on the GPU. This approach can be viewed as a portfolio solver composed of two solvers, only in this solver one is DPLL based and the other is a Survey Propagation Solver.

A similar approach is taken by Beckers et al. [26] by combining TWSAT [27] another incomplete solver, with MiniSat, using the first to provide the decision heuristic, instead of using the more common VSIDS heuristic, that ships with MiniSat.

Fujii et al [28], 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. [29] used a FPGA to accelerate the BCP procedure, instead of using GPUs.

Meyer [30], proposed a parallel 3-SAT solver on the GPU that featured a pipeline of kernels that process the problem, the work discards most common techniques and relies only on massive thread parallelism to solve problems. The work was focused in determining the scalability and applicability of GPUs to sat solving rather than trying to devise the next high end SAT solver.

5 Proposed Approach

This section provides an overview on our proposed approach. In order to solve SAT problems using the GPU, we propose a system consisting of three components(fig. 5):

1. A converter between the representation used on the CPU and the one used on the GPU
2. A SAT Solver
3. A Clause Learning and Problem Regeneration module

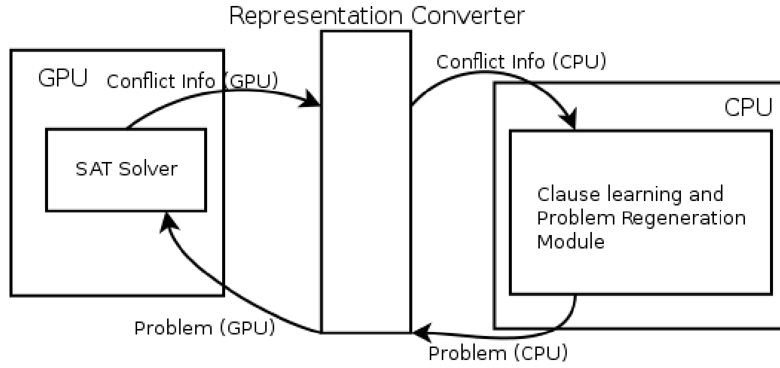

Fig. 5. Proposed system's architecture

5.1 System Overview

Because the problem does not fit the GPU paradigm completely, the system uses the CPU and the GPU cooperatively and each component processes the part of the problem that suits it the best.

The GPU will do the heavier, more systematic, work while the CPU will do the lighter and less structured work.

This idea comes from the fact that since all the threads in a stream multiprocessor (SM) of the GPU are executing the same instruction (vector processing), if they are performing work that is less systematic, computing power is going to be wasted, as some threads will be waiting for others.

We thought that moving only the Unit Propagation, that is completely systematic, to the GPU wasn't going to bring the kind of performance we wanted, because as it is true that Propagation takes most of the time of the solving process, each Propagation takes so little time that most of the times we would lose time because of the overhead that is starting a GPU kernel.

This made us consider into taking more of the solving to the GPU, but always taking into account that if we take too much the GPU will not perform at its best.

5.2 Representation Converter

In our approach we need to constantly exchange information between the GPU and the CPU. Because they have different internal representations for the problem, this information needs to be transformed before it can be of use.

The structure used in the CPU to represent the problem being solved, does not fit well into the GPU model, because in the GPU memory accesses are much more expensive than in the CPU, and moving the CPU representation directly to the GPU may negatively impact the GPU Solver performance. As explained before, in the GPU, if memory accesses are coalesced, they can be converted into one access rather than many. The main reason for this convertion is to allow each side to work with the representation that suits it the best, meaning that the representation may have to be slightly converted, or it may have to be changed altogether.

The representation we will use in the GPU will have no dynamic structures. Initially, once the problem gets to the GPU it will remain unchanged until a restart happens, and a new problem is converted and loaded onto it.

5.3 SAT Solver

The SAT solver will run on the GPU, it will feature GPU optimized versions of the unit propagation, backtrack and variable selection procedures and will also feature a conflict analysis step that will allow for non-chronological backtracking. However, due to the representation chosen for the problem, the solver will not learn clauses on the go. That will be the job destined to the last module.

5.4 Clause Learning and Problem Regeneration module

When the Solver analyzes a conflict to decide the level it will have to backtrack to, it sends the data used in the analysis to the CPU and it is the CPU job to make an in depth analysis of the conflict and to generate a conflict clause that may, or may not, be added to the problem. When the CPU retrieves the data sent from the GPU, the representation converter is used, to convert it back again to the CPU representation.

This separation of roles was considered because the algorithm is not regular enough to fit the GPU, and by removing some irregularity from the algorithm and by handling it on the CPU, we can combine the best of the both worlds and have an algorithm that is runnig at full power on the GPU while, at the same time, it is using the CPU to do some relevant work.

Another reason to consider this separation is that if the quantity of conflicts that the GPU can produce is high, having the CPU to solely take care of this job, makes it usefull.

The resulting problem generated on the CPU will be sent to the GPU everytime a restart happens, when the Solver restarts, the old problem is removed from the GPU and replaced with the new one, with the newly learned clauses.

5.5 Work already conducted

To understand where the GPU could be of use in solving SAT problems, we developed several solvers that used different approaches or representations. We will now explain some of the experiments we performed so far.

5.5.1 Brute Force with JIT Compilation

The first experiment we did was to take advantage of the Just-In-Time compilation facilities that are provided by CUDA, in which one can generate kernels at runtime.

We read the problems, converted them into code, compiled them and inserted them into the graphics card. This solution is not of much use because even tough the JIT compilation result is fast, it fails to solve even small problems, because it had to try every combination possible, and those grow exponentialy with the problem size.

5.5.2 Brute Force with Problem Matrix

The second experiment was made to try and use the matrix multiplication facilities provided by the CUDA framework. This experiment had several problems. Firstly, it greatly limited the problem because the matrix representation, as it takes $O(variables \times clauses)$ of space. Secondly, the matrix that was representing the problem was almost always sparse. That lead to bad running times, as most of the work wasn't usefull at all, this approach besides these two problems, also shares the main problem of the JIT experiment.

5.5.3 Brute Force with Compressed Problem Matrix

This experiment was performed to try and eliminate the problems with the previous approach: the memory limitation and the useless work. This way, we compressed the matrix and this lead to better running times and space consumed. However, like the previous experiments, trying every assignement option is only doable for small problems.

5.5.4 DPLL algorithm on the GPU with simplified Divide-and-Conquer

In the last experiment performed, we coded the DPLL algorithm in the GPU and let all the solving take place on it. We had a simplified divide-and-conquer where we gave each block an initial assignement and let it work from there. There is also some rudimentary work balancing, as we spawn more blocks than those the GPU can handle, and these leftover blocks are only executed when one other block completes its job, making sure that every processor of the GPU is kept busy. The running times are not great, as it does chronological backtracking and does not learn, but the kind of performance ehxibited in terms of conflicts generated and decision tree coverage, led us to conclude that, with some work, this could be part of a possible solution.

5.6 Remaining work

- Months 1 - 4 (January to May)
  - Optimize the current implementation of our DPLL based solver
  - Analyze the learning procedures of modern SAT solvers
  - Implement the two-way representation converter
  - Implement a variable selection heuristic
  - Implement chronological backtrack on the GPU
  - Create the communication facilities between the CPU and the GPU
  - Implement the problem swapping and the restarts
  - Run common benchmarks
  - Run industry problems
- Month 5 (June)
  - Write the master's thesis document according to the work undertaken and results obtained.

5.7 Evaluating the work

To evaluate our proposed approach we will consider the following points:

- The scalability of our approach regarding the quantity of blocks, and GPUs used.
- The results of the common benchmarks used to evaluate solvers.
- The performance of our solver in real world SAT problems.

6 Conclusions

In this work we propose to investigate novel techniques for enabling a more effective use of GPUs in the acceleration of SAT solving. There are still only a few SAT solvers that use GPUs, and their role is quite limited. We envision, as the main outcome of this work, a better balance between the role of the GPU and the CPU in the solver, which will hopefully enable large performance gains. We will evaluate this work using both common benchmarks and real world SAT instances.

References

1. SA Cook. The Complexity of Theorem-Proving Procedures. *Proceedings of the third annual ACM symposium on ...*, 1971.
2. K McMillan. Applying SAT methods in unbounded symbolic model checking. *Computer Aided Verification*, 2002.
3. RM Karp. Reducibility among combinatorial problems. *50 Years of Integer Programming 1958-2008*, 2010.
4. Martin Davis. A Machine Program for theorem-proving. *Communications of the ...*, 1962.

5. MW Moskewicz, CF Madigan, and Y Zhao. Chaff: Engineering an efficient SAT solver. *Proceedings of the 38th ...*, 2001.
6. N Eén and A Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. *Theory and Applications of Satisfiability Testing*, 2005.
7. J.P. Marques-Silva and K.a. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.
8. E Goldberg and Y Novikov. BerkMin: A fast and robust SAT-solver. *Discrete Applied Mathematics*, 2007.
9. J Marques. GRASP—A New Search Algorithm for Satisfiability. *in Proceedings of the International Conference on ...*, 1996.
10. RJ Bayardo and RC Schrag. Using CSP look-back techniques to solve real-world SAT instances. *Proceedings of the National Conference on ...*, pages 203–208, 1997.
11. Hantao Zhang. SATO: An efficient prepositional prover. *Automated DeductionCADE-14*, 2(x), 1997.
12. Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. *Computer Aided Verification*, 2002.
13. J Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. *Progress in Artificial Intelligence*, pages 62–74, 1999.
14. JN Hooker and V Vinay. Branching Rules for Satisfability. *Journal of Automated Reasoning*, 1995.
15. CP Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. *... of the National Conference on Artificial ...*, (July), 1998.
16. Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT 1.5: solver description. *baldur.iti.uka.de*, pages 3–4, 2008.
17. H Zhang, MP Bonacina, and J Hsiang. PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems. *Journal of Symbolic Computation*, 1996.
18. Ruben Martins, Vasco Manquinho, and Ines Lynce. Improving Search Space Splitting for Parallel SAT Solving. *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, pages 336–343, October 2010.
19. Youssef Hamadi, S Jabbour, and L Sais. Control-based clause sharing in parallel SAT solving. *Proceedings of the 21st international jont ...*, 2009.
20. Youssef Hamadi, S Jabbour, and L Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean ...*, 6:245–262, 2009.
21. Norbert Manthey. A More Efficient Parallel Unit Propagation. *wv.inf.tu-dresden.de*.
22. Kanupriya Gulati and Sunil P. Khatri. Boolean satisfiability on a graphics processor. *Proceedings of the 20th symposium on Great lakes symposium on VLSI - GLSVLSI '10*, page 123, 2010.
23. A Braunstein. Survey propagation: An algorithm for satisfiability. *Random Structures & ...*, 2005.
24. N Eén and N Sörensson. An Extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, 2004.
25. Sander Beckers and Gorik De Samblanx. Parallel SAT-sovling with OpenCL. *Proceedings of the ...*, 2011.
26. B Mazure, L Sais, and E Gregoire. TWSAT: A new local search algorithm for SAT-performance and analysis. *Proceedings of the 14th National Conference on ...*, pages 1–12, 1997.
27. Hironori Fujii and Noriyuki Fujimoto. GPU Acceleration of BCP Procedure for SAT Algorithms. *elrond.informatik.tu-freiberg.de*.

28. J Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. Designing an efficient hardware implication accelerator for SAT solving. *. . . Applications of Satisfiability TestingSAT . . .*, pages 48–62, 2008.
29. Quirin Meyer, Fabian Schönfeld, Marc Stamminger, and Rolf Wanka. 3-SAT on CUDA : Towards a Massively Parallel SAT Solver. 2010.