Parallelization of SAT algorithms in GPUs

Carlos Filipe Costa
Instituto Superior Técnico
carlos.silva.costa@ist.utl.pt
May 2014

ABSTRACT
The boolean satisfiability problem is a very important NP-complete problem. However, during recent years, the performance of SAT Solvers has came to an halt, which lead to work being done to use GPUs to aid in the solving process. However, the techniques employed misuse the GPU and limit its applicability. In this thesis we present a novel approach to use the GPU that removes these limitations and allows the GPU to solve problems of any size, which was not possible with former approaches.

Keywords
GPU, Boolean Satisfiability

1. INTRODUCTION
The Boolean Satisfiability Problem was the first NP-Complete problem to be identified[6]. Some factors still make it one of the most important problems in computer science. Those factors are its simplicity, its numerous practical application in real life problems such as software testing with model checkers, theorem proving and Electronic Design Automation. But, perhaps the most important factor is the ability to reduce any NP-Complete problem to SAT[14] making it a cornerstone in what comes to solving this class of problems.

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm[8], introduced in 1962, solves the problem by recursively traversing the entire search space with several improvements over a more basic brute-force algorithm finishing only when a solution is found or when the algorithm has exhaustively checked every possibility and found no solution. This algorithm serves, even today, as the basis for Modern SAT Solvers[11] as they build upon it. However, in recent years the global performance of state of the art solvers has been stalling and only minor optimizations have been introduced to improve them despite important efforts of the whole community[16][2].

Moreover, the CPUs are hitting their physical limit in terms of single core speed and to continue to improve in speed SAT Solvers had to keep up with the hardware and go parallel[16]. The initial method to parallelize SAT Solving was to split the search space between the cores/threads, and, with load balancing techniques re-split the space every time a core/thread finished its search[20][4]. These Solvers could also share important clauses they would find during their search[5]. The most successful method so far is to use portfolio solvers[17]. A portfolio Solver consists on a solver that has several different single-threaded solvers, running in each core/thread racing against each other while also sharing meaningful clauses[13].

Meanwhile, in recent years, GPUs, with the introduction of General Propose GPUs, have ceased to be a graphics only hardware to become able to do everyday computing. GPUs have hundreds of cores and a computing power that, if used properly[15], outperforms that of the Central Processing Unit (CPU). However, even tough SAT Solvers made the jump to multi-cores, they are not taking advantage of this piece of hardware that, like the processor, exists in every computer.

To address this gap, in this thesis we propose a way of using the GPU to aid the process of SAT Solving, and even though we believe GPUs can be used to obtain substantial improvements in the performance of state-of-the-art SAT solver algorithms, their applications pose several challenges. First, GPUs make use of a different parallel paradigm than multi-core systems. Whereas in multi-core systems each CPU can execute an arbitrary instruction at any point in time, in GPUs all cores have to be synchronized executing the same instruction (even if using different data). Second, the memory model of the GPU is also different. While the CPU is optimized for random accesses to the memory, the GPU, in order to achieve top performance, should have all threads accessing the memory in sequential positions. So to achieve optimal performance improvements, we believe that the creation of a novel mechanism that reconciliates both GPUs and SAT solvers is mandatory.

With this in mind, 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 present a method to split the computation in order to regularize it, and we will use the CPU to tackle the parts of the problem that cause the irregularity instead of having them made on the GPU. This separation comes from the fact that some parts in the DPLL algorithm add irregularity to the algorithm if they are executed in the GPU, which due to the SIMT paradigm, becomes very expensive. More specifically, conflict analysis and clause learning as the first adds irregularity to the algorithm, and the second is difficult as well as on the GPU no proper synchronization methods exist. However, without conflict analysis and clause learning, we fall back to a GPU adaptation of the simplistic DPLL. So to tackle this challenge we have the, highly regular, search

done in the GPU, while having the, complicated and single threaded, analysis and clause management done in the CPU.

To demonstrate our results we will run standard benchmarks against our project and compare results with some state of the art solvers and with some solvers that leverage on the GPU as well.

2. CUDA

CUDA is a programming framework for machines with nVidia GPUs. It is composed by a programming language, a compiler and a runtime environment.

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 multicore CPUs. An advantage is that it has that a much higher number of threads that can be ran at the same time, 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.

2.1 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.

2.1.1 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[2] 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.

2.1.2 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.

2.1.3 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.

2.2 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.

A kernel is considered Memory bound 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. 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

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

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

to bad memory accesses patterns or due to serialization as a result of branch divergence, atomic operations or bank conflicts in shared memory. 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.

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.

3.1 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 an 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 same search tree, when one of the solvers reaches a solution, this result is in fact the solution for the problem, hence, there is no need for load balancing. These ideas, coupled with the clause sharing techniques mentioned before, makes the state of the art in parallel SAT solving.

3.2 GPU enhanced Solvers

In MESP (Minisat Enhanced with Survey Propagation) [12] 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.

Fujii et al [10], took another approach and 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. [7] 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 analyse 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.

The CUD@SAT [1] 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. [10], doing only clause analysis in the GPU, but they introduce conflict analysis, as opposed to Fujii's [10] 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 synchronisation.

Noting that, all these methods can be ran with, or without, a watched literal scheme. In all it's flavours 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.

3.2.1 Scalable GPU Framework

Meyer [18], 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 [18].

4. MINISATGPU

This section presents our concept solution for solving sat which relies on the cooperation between the CPU and the GPU. First, in Section4.1, we will present the idea and a top-level view of the system, its components and data structures. Afterwards, in Section4.2 we will present relevant implementation details, of both the CPU and the GPU components of the solver, as well as some decisions that lead us to the final solver.

4.1 System Overview

To take full advantage of the GPU capabilities we opted to use an approach similar to CUD@SAT where they use the CPU and GPU in cooperation. Houwver, unlike with CUD@SAT We decided to execute the BCP and variable picking in the GPU, while the conflict analysis was to be executed on the CPU, as it is a single threaded procedure. This decision allows us to send a new problem every time we returned to the GPU, thus, we were able to add clauses to the problem, enabling us to learn from conflicts.

4.1.1 A Novel Approach

Like us, most GPU solvers, use both the CPU and the GPU together. However, they use the GPU in a different way. They use the entire GPU to run a single search thread[3].

[3]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

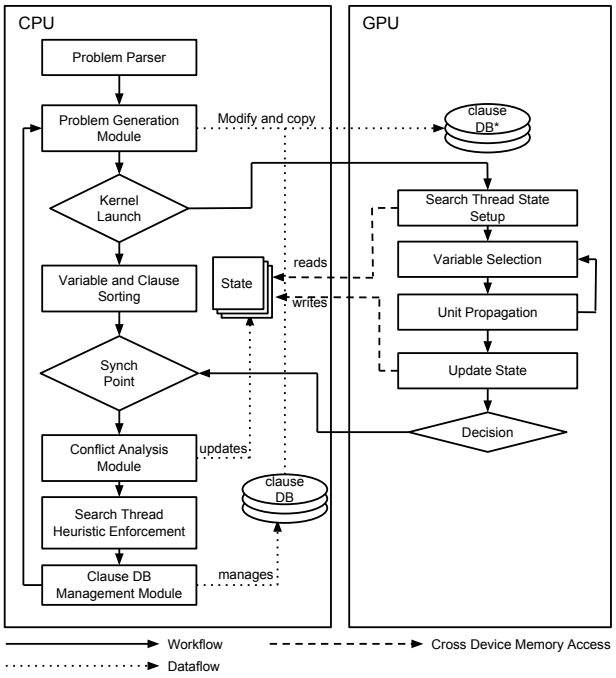

Figure 1: Diagram of the sustem architecture

Each block is given a different set of clauses and they will analyze that set of clauses and propose propagations. They only propose because GPU blocks can not communicate with each other, which produces a very practical effect. To make propagations and decisions, they must return to the CPU as it serves as the synchronization point. This results in frequent focus switches between the CPU and the GPU, which means that an extra overhead is added to the computation.

We decided against this for two reasons. First, focus switches are expensive and it is good practice to maximize the work done in each kernel call. Second, analyzing all clauses in every kernel call, although it results in an extremely regular kernel, is very inefficient and will fail to scale to bigger problems, as we will show in Section 5. The problem with this approach is that the number of clauses is the limiting factor, as there is a limit to the number of blocks that can be ran in parallel and thus, to the number of clauses that can be analyzed at a time.

We propose a different way to use the GPU. Instead of analyzing all clauses, we choose to analyze only the clauses we needed to, by analyzing the effects caused by one assignment at a time. By doing this careful clause selection, we can do what takes the others the entire GPU, in a single block. This is possible because with this approach, the limiting factor changes from being the total number of clauses, to be the ratio of clauses per variable. By fitting the propagation step in one block only, enables us to do more in the GPU in two different ways. First, as we eliminated the need to do inter-block communication, we can do the entire search in the GPU. Second, as we can do the entire search in a single block, we can use the rest of the GPU to run additional search threads.

This being said, our approach at using GPUs to solve SAT, as depicted in figure1 consists on three key phases: 1) The search phase, that comprises both the decision phase and the unit propagation phase; 2) the problem 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 problem definition is converted to a more GPU friendly notation and sent to the GPU.

4.1.2 Architecture

Our system is partitioned into four modules. 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
- Search Thread Handler

The **Search Engine** is the only component that is executed in the GPU, while the others run 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 search time[19], 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**. This component holds all the clauses, both the initial set and the ones generated during conflict analysis. 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.

The last component is the **Search Thread Handler**. Which enforces the search heuristics picked for each thread, allowing us to have a portfolio of search threads. These handlers are read and updated both in the GPU and in the CPU.

4.1.3 Data structures

These structures will be used both in the GPU and in the CPU.

A crucial structure is the Clause Database. This Database is moved to the GPU with a modification when compared with the CPU version. As shown in figure 2, we need some information when in the GPU that is not necessary in the CPU, and vice versa. So when we create the Database that we are

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.

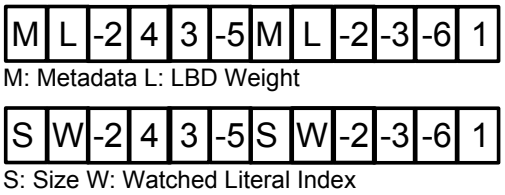

Figure 2: The clause information in the CPU and in the GPU

going to copy to the GPU, we replace this information, with the information that better suits our needs.

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 3. 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.

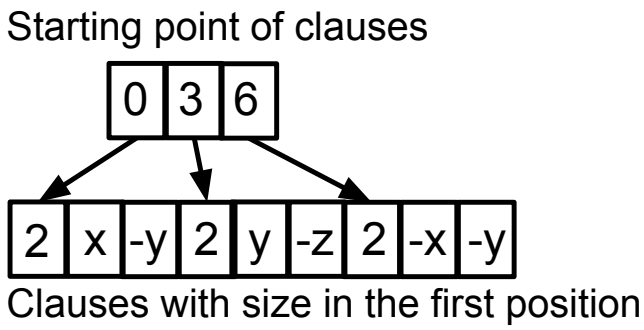

Figure 3: Compressed Matrix representation

Lastly, we need the information about each search thread state. This information, depicted in Figure4, is stored in a vector of structs, the vectors that represent all the additional information, such as variable assignments and the clause that caused such assignment, are stored in one vector only allocated after the problem is processed. We do this because this information is used both in the GPU and the CPU and to copy data from the GPU that is sequential is faster than doing the same with data that is scattered.

4.1.4 Workflow

Our system's components cooperate, and work, with each others information, so that together they can solve problems. The general workflow of our system is the following, we process the clause database in the CPU and we send it to the GPU. In the GPU we assign values to variables and propagate those decisions until either a conflict, or a solution, is found. After that, we return to the CPU where we will do conflict analysis and, if necessary, a clause database cleanup. At this time, we are back in the beginning, where we process the clause database before sending it to the GPU.

```
typedef struct{
  // current level of assignment
  int curLev;
  // clause that implied the variable
  int* varClause;
  // assignment level of each variable
  int* varLevel;
  // polarity of each variable
  int* var;
  // assignment trail
  int* trail;
  // trail Limits
  int* trailLim;
  // number of variables on the trail
  int trailSize;
  // number new variables on trail (set by CPU)
  int newTrail;
  // number of variables in the problem
  int varSize;
  // conflict clause (set by GPU)
  int confl;
}state;
```

Figure 4: structure that represents the state of each search thread.

In a more detailed view, as is depicted in Figure1, the whole process starts when a problem is given to the system. Minisat's built in parsers will read the problem and store it in minisat's own data structures. After this, the solver will read the parameters given to it, some are native to minisat, some are new parameters, and it will configure itself accordingly.

Parameter	Description	Default Value
–blocks	Number of solvers to be launched concurrently (one per GPU block)	64
–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 1: MinisatGPU's configuration parameters

The next step is to set up the search threads. Each search thread will have its own context and heuristics, for restarts and polarity mode. This constitutes the setup phase. After, comes the solving phase, which will only end when a solution 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 in a GPU friendly way. In this step, as said before, additional information is stored in each clause, before it is sent to the GPU to help during search.

At this point, the problem definition is on the GPU, so we can launch the GPU to start working on this. When we

launch the GPU, we work in parallel in the CPU, to do some activities that can be done in advance.

In the GPU we will do a one time copy of each search thread state to a local container. This states, as said before, are stored in pinned memory, which can be accessed from the GPU. The next phase is to pre-load, to the appropriate container, the implication vector, the variables that are going to be propagated, either as a result from conflict analysis, or as a result of other search thread work. If these don't exit, we select one variable, instead. After this, each variable is picked in succession, one at a time, and the clauses where they show up, are processed in parallel, one per thread. If during this stage, an implication is found, it is added to the end of the implication vector. We leave this stage with one out of two conditions: 1) we have no more variables to propagate, in which we proceed to select another one; 2) we have found a conflict, in which the search thread ends its search; We have a solution when no more variables are left unassigned, and there is no more work to do. We leave the GPU when all threads have finished their work and have either a conflict, or a solution.

When we return to the CPU, we will either have a solution, and the solver returns it and ends, or as many conflicts as search threads. If it is the latter, we will have to analyze the returned conflicts. We use a modified version of minisat's conflict analysis procedure. These modifications have to do with the fact that minisat stores it's search thread state in the problem definition, and makes use of it in several steps of the solving process. However, as we have several search threads and, consequently, states, we cannot have state built in. After we are done with conflict analysis, we will add the resulting clauses to the database. However, clauses that have only one literal, are not added to the database, they are added directly to each search thread as an assumption instead. If two assumptions collide, the solving process ends with UNSAT. When adding these new clauses, if we exceed the limit of clauses that the database can have, we clean the database using with a modified procedure, we increase the limit, either exponentially or linearly, and we send the problem to the GPU again. At this point we are at the beginning of the solving phase loop, again.

As said previously, when we launch the GPU kernel we do some work in parallel, in the CPU. This work consists on sorting the variables with respect with their VSIDS weights, also, when a database cleanup is eminent, we also pre-sort the clauses so that we do not need to do that before removing the clauses. This way, we use the CPU to advance work, when it would, otherwise, be idle.

4.2 Implementation

To implement our system, we decided against starting a solver from scratch and to integrate our changes into minisat[9], a lightweight and extensible solver, instead. This decision came from the fact that by having facilities such as problem parsing and clause management routines, our job would be less prone to errors. Moreover, Minisat[9] has been extensively tested. That being so, when integrating our changes, we could be fairly certain that wrong answers to problems were the result of our modifications. However, integrating with minisat[9] also had disadvantages as we have to make the GPU side of the solver, minisat[9] compatible. That is, we had to store and return all the information that minisat[9] needed to do its job. Also, as we use minisat's[9] own conflict analysis and clause strengthning routines, we had to make them GPU-friendly as in our solver, unlike in minisat[9], 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[9] 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.

4.2.1 Minisat adaptation

In adapting minisat[9] to our needs, we had to make several changes to the way minisat[9] operated. This happens because minisat[9] is highly optimized to have only one search thread, during the solving process. As we wanted to make minisat[9] able to handle several search threads, we had to prepare it so it was able to do so. The first thing we had to change was to make minisat[9] stateless. Besides the assignment stack, minisat[9] stores state in the problem itself, as it changes the order of the literals in a clause so that the two watched literals are always the first two literals 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, minisat[9] makes use of this order when it comes to conflict analysis and conflict clause strengthning.

Conflict analysis makes use of this ordering because the first literal in a clause is always the last one to be assigned due to that same clause. More specifically, due to unit propagation.

This means that, because during analysis we follow these propagations 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.

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[9], 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.

Another place where minisat[9] needs to be altered is in the database management procedure, specially during cleanups. Minisat[9] works with a single thread and it is fine tuned to excel at doing so. However, when we have no state and instead of a single search thread, we have many, minisat's[9] 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[9] 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[9] 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.

These are the modifications that enable minisat[9] to hold several search states instead of just one, like it usually does. To actually transform minisat[9] 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, we regenerate the problem and send it back to the GPU for further analysis.

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.

We decided to handle our search threads with a portfolio approach. This happened for two reasons: 1) There is no need to do load balance 2) with hundreds of search threads, having different approaches is 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[3] were they do not have a lot of different heuristics, just one, they add diversity by randonly attributing different initial variable selection orderings to each search thread[3]. 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.

We choose, for restart heuristics, the two provided by minisat[9], exponential and luby, and the LBD restart strategy introduced in glucose[2]. Additionally, we have three polarity modes: 1) all variables are assigned true, 2) all variables are assigned false, 3) variables are assigned their last implied value, also known as phase saving.

In running a portfolio solver, we differ in the way we handle learnt clauses. Normally, solvers would share only clauses that were considered important. However, they do this because sharing all clauses would cause an overhead that would make the benefits of clause sharing irrelevant[5]. We do not face this problem. Because all search threads are halted when conflict analysis and clause database management is being done, we are able to have only one clause database that receives all 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 greatly, as we would have to send hundreds of databases instead of one single database. There is one last benefit to having one clause database for all threads. As we let the limits that would trigger a cleanup unchanged from the single threaded minisat[9], in the beginning, frequent cleanups are made leaving us with a very optimized set of clauses in the database.

4.2.2 GPU Side

Even with all the changes made to minisat[9], further adaptations were necessary in the GPU side to make them work together. We started with the solver described in the previous section and we stripped it down so only the search loop was left. At this point we only had the literal selection and the clause analysis procedure. To make the two sides work together, we had to store information we did not need before: **trail**, where the assignment order is stored; **trail limit**, which stores the index where each level ends in the trail; the **level** itself; **variable information**, their assignment, the clause that implied them, and the implication level they belong to.

That being said, we have to update this information during the search routine. Much of the information regarding variables was already stored as it was needed to do conflict analysis, the adaptation was in the location where we stored it. However, there was no trail, no trail limits, and no information about implication levels, before. Our search procedure uses an atomic operation to let us do several operations atomically, as can be seen in Figure5. To achieve this we leverage on the fact that only one thread will have the result required to enter the 'if' clause in line 165. This change, although it increases code divergence is an optimization as the third step, the confirmation step, would have to check all variables for new information. This is an issue for two reasons, the first is that we had to do additional memory accesses. The second reason is that, when the problem at hand had thousands of variables, this step was extremely costly.

The first optimization we made was to the clause analisys procedure, for it to take advantage of one fact regarding memory accesses in the GPU, as said before, reads are asynchronous. With this in mind, there are two pieces of information that we need to get in order to analyze a clause. The first is the clause itself, that will be in contiguous positions in memory. The second is the value that each variable, associated with the literals that compose the clause, hold. To read these values in the most optimized way, we fetch four literals at a time. As all literals are stored in contiguous positions, when we fetch the first, as the GPU will get a chunk of memory with each fetch, we will most likely get the others as well. Meaning that we fetch all four literals, for the price

```
// we do an atomic compare and set with
// -1 (unassigned)
// if the comparison is successful we
// replace the value in memory with the
// new polarity
// the operation returns the previous
// value stored in memory
int oldval=atomicCAS(vars[UnassignedLit],-1,
                     ourPolarity);
// if this value is equal to our polarity
// or was unassigned
// there is no conflict. If that is not
// the case, we report a conflict
int conf=(oldval!=ourPolarity) && oldval!=-1;
if(conf){
  return conf;
}
if(oldval!=-1){
  return -1;
}
// here we set the rest of the information
```

Figure 5: Using an atomic compare and set operations to provide concurrency for more than one operation

of the first. After that, when we fetch the value associated with each variable, as we fetch the four in succession, we can start processing the first values while waiting for the others to complete.

This optimization attacks one of the most crucial performance issues of SAT Solvers in the GPU. The fact that this problem has low memory locality. When we read the clause, as the positions are sequential, we can take advantage of caches to speed up the process. However, when we need to read the values assigned to the variables, the reads will be completely random, and 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, instead of the optimal value of one.

In order to further optimize this procedure, we implemented watched literals in the GPU. However, the procedure where we verified the literals was so unoptimized in relation to memory localness that the benefits of using watched literals, were hidden by the memory access overhead. Furthermore, the fact that all threads must wait for the others, in the same warp, to finish the work in a clause, means that if one thread fails to fall into the watched literal rule, there will not be a noticeable difference in performance for that particular round of analysis.

4.2.3 GPU/CPU cooperation

The last part of the implementation was to actually connect the GPU with minisat[9]. The cooperation is mainly made through information exchange, this being said, optimizing memory transfers and postponing some work in the CPU to be done while the GPU is busy, is crucial to achieve optimal performance.

To achieve this, we aimed to reduce the time needed to send the clause database to the GPU. This procedure was was dominated, executionwise, by the successive calls to the memory copy API, so instead of copying each clause to the GPU, we modified the procedure to copy the clauses to an auxiliary buffer. With the aid of this buffer we could copy the entire database in one operation, making this operation as efficient as possible. Furthermore, we noticed that these copies, however small, were still taking some time to complete, and there was work to be done after them. To address this, we started using Asynchronous memory copies, so we could continue our work while the data was being copied. These asynchronous operations, are only asynchronous in respect to the CPU and will maintain order with respect to other GPU operations, so 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, come hundreds of sort operations. By delaying, by one search, the ordering and making them after the kernel launch, 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 comes to reduce the database size. This optimization works because clauses added in the last round of searches, will be locked, as they will belong to a reasoning, 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 section an evaluation of our solver against solvers that use the CPU exclusively and solvers that use both cooperatively.

5. EVALUATION

To evaluate our solver we will compare it against the state-of-the-art in SAT solvers that use the GPU, as a metric, we will also test our solver against minisat, the original solver, and glucose, from which we took the clause rating mechanism and the restart heuristic. The test suit is a subset of well known problems taken from SATLib, a popular repository of SAT problems. We will first test our solver against the best solver we have found that uses the GPU during the solving process, CUD@SAT.

CUD@SAT has several configurations, presented in its website. Therefore, we will present the minimum time it took to solve each problem by the best combination of options, instead of having several columns, one for each configuration.

The tests used are described in Table 2 and we chose these problems because these sets reflect well what happens across other problem sets, in general, when we compare the two solvers.

5.1 Evaluation against GPU Solvers

As can be seen in Table3, our solver is very competitive with CUD@SAT in small problems, and both solvers stuggle with hard problems like hole10. However, when it comes to big problems, CUD@SAT can not compete with us as they do not scale well.

Type	Name	Description
Industrial	bmc-ibm series	Module checking problems from IBM
Industrial	bmc-gallileo series	Mmodule checking problems from Gallileo
Encodings	holeN series	Encodings of pigeon hole problems
Encodings	parity series	Encodings of parity functions problems.
Encodings	gus-md5	search for md5 hash colisions

Table 2: Problem sets

Problem	MinisatGPU	CUD@SAT (best)
bmc-gallileo-8	13.96	>600secs
bmc-galilleo-9	17.28	>600secs
bmc-ibm-10	19.47	>600secs
bmc-ibm-11	16.90	>600secs
bmc-ibm-12	40.65	>600secs
bmc-ibm-13	3.99	>600secs
bmc-ibm-1	4.17	>600secs
bmc-ibm-2	0.39	0.07005
bmc-ibm-3	5.89	>600secs
bmc-ibm-4	3.77	>600secs
bmc-ibm-5	0.72	>600secs
bmc-ibm-6	8.66	>600secs
bmc-ibm-7	0.23	>600secs
hole8	9.27	4.90
hole10	672.71	>1000secs
gus-md5-04	128	22.29
gus-md5-07	160.09	>600secs
par16-1-c	0.63	>600secs
par16-2-c	1.18	32.50
par16-3-c	1.66	>600secs
par16-4-c	0.88	360.91
par16-5-c	2.37	35.00

Table 3: Results agains the best option of CUD@SAT

5.2 Evaluation against CPU 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. This being said, we are now going to compare our solver with minisat, the starting point, and glucose, from which we took the clause rating and restart heuristics. We are going to use the same problem sets, and both minisat and glucose are tested using their default settings. Even though we have a highly competitive solver against other GPU implementations, there is still work to be done when it comes to competing with CPU only implementations, see Table4. 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 all the overhead that the GPU introduces. Tests we made show that 100.000 empty kernel launches if synchronized, take 1 second. 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. Which limits the applicability of GPUs to larger problems.

Problem	MinisatGPU	minisat	glucose (2.3)
bmc-gallileo-8	13.96	0.25	0.05
bmc-galilleo-9	17.28	0.27	0.12
bmc-ibm-10	19.47	0.28	0.12
bmc-ibm-11	16.90	0.32	0.09
bmc-ibm-12	40.65	0.98	1.04
bmc-ibm-13	3.99	0.48	0.45
bmc-ibm-1	4.17	0.09	0.03
bmc-ibm-2	0.39	0.01	0.01
bmc-ibm-3	5.89	0.09	0.04
bmc-ibm-4	3.77	0.09	0.03
bmc-ibm-5	0.72	0.02	0.01
bmc-ibm-6	8.66	0.33	0.05
bmc-ibm-7	0.23	0.01	0.01
hole8	9.27	0.43	4.90
hole10	672.71	211.46	71.66
gus-md5-04	128	1.41	0.98
gus-md5-07	160.09	24.41	54.50
par16-1-c	0.63	0.06	0.06
par16-2-c	1.18	0.12	0.08
par16-3-c	1.66	0.07	0.06
par16-4-c	0.88	0.02	0.01
par16-5-c	2.37	0.09	0.03

Table 4: Results agains minisat and glucose

However, the challenges do not end with these CPU-GPU interoperability issues. While 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 the variable polarity. This second access will most likely not be coalesced or cached and it will, therefore, result in the repetition of these read instructions, which are issued on average 8 to 9 times, which is close to limit of 16 repetitions[4]. 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, as we spend 80% of the execution time of each kernel waiting for these read operations to terminate.

6. CONCLUSION

In this thesis we have presented a new method to use GPUs to solve SAT problems. Our method uses three key insights: 1) Instead of using the whole GPU to run a single search thread, pick clauses carefully and run a search thread in a single block; 2) run the entire search in the GPU to reduce focus changes 3) scale by adding more search threads to the GPU. Using these ideas, we built a solver, minisatGPU that implement a portfolio of search threads that are run 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

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

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. And, while we believe that with new algorithms, GPUs can be an alternative to CPUs, with current SAT and GPU technology, CPUs only solvers will most likely, have the upper hand for the foreseeable future.

7. FUTURE WORK

We believe our solver can be improvised in two ways. When we return from kernel execution, the GPU is left waiting until the next round of search arrives. The first improvisation is 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 improvisation is made upon the first, by using different streams, which are basically a way of telling the GPU which operations can be done in parallel and which cannot, in relation to the GPU, we can hide the time we spend transferring the clause database by doing work in the GPU at the same time. This can only be achieved with two, or more kernels, which mean that in our implementation, we are subjected to these transfer times.

8. REFERENCES

[1] Exploiting unexploited computing resources for computational logics, June 2012.

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

[3] A. Biere. 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.

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

[5] W. Chrabakh and R. Wolski. *GrADSAT: A Parallel SAT Solver for the Grid*. 2003.

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

[7] J. D. Davis, Z. Tan, F. Yu, and L. Zhang. Designing an efficient hardware implication accelerator for SAT solving. In H. K. Büning and X. Zhao, editors, *Theory and Applications of Satisfiability Testing – SAT 2008*, number 4996 in Lecture Notes in Computer Science, pages 48–62. Springer Berlin Heidelberg, Jan. 2008.

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

[9] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, number 2919 in Lecture Notes in Computer Science, pages 502–518. Springer Berlin Heidelberg, Jan. 2004.

[10] H. Fujii and N. Fujimoto. GPU acceleration of BCP procedure for SAT algorithms. 2012.

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

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

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

[14] R. M. Karp. Reducibility among combinatorial problems. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 219–241. Springer Berlin Heidelberg, Jan. 2010.

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

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

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

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

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

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