Algorithms for Maximum Satisfiability using GPU

André Reis
andre.reis@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

May 2017

Abstract

Interest in algorithms for solving Maximum Satisfiability (MaxSAT) has been increasing greatly in the last few years. However, there seems to exist a great lack in the use of the power available in GPUs. For this work we presented a proposal to harness the GPU and its great parallel computing power, integrating it into an approach that solves MaxSAT instances. We reviewed some of the current state of MaxSAT solver algorithms in order to explore the direction of our approach. Starting with development of a base version that utilizes the GPU, and a variation that uses only the CPU we were able to compare each other early on to see where we stood. Focusing on the use of a Genetic Algorithm in this base version in order to improve the quality of results by our approach. Further variations of our approach were developed, introducing algorithms that could improve the quality of the results obtained by the MaxSAT solver that represents our approach. Through testing we could conclude that the current state of the art solvers are extremely advanced and hard to compete against on a level playing field, being difficult to achieve the same quality of results in the same span of time. Due to the vastness of possibilities to use GPU for MaxSAT solving we conclude that we have not explored all options within the domain of use of GPU and there is still room to explore.
Keywords: MaxSAT, SAT, GPU, CUDA

1. Introduction

This thesis consists of an exploratory research into existing Maximum Satisfiability (MaxSAT) solvers and related algorithms. We conduct this analysis of existing work with the purpose of developing a MaxSAT solver design able to leverage on the GPU's specific capabilities.

In the following section we describe the motivation that make us believe this is work worth pursuing.

1.1. Motivation

Over the past few years, there has been renewed interest in the development of MaxSAT algorithms [11], an optimization version of the Satisfiability (SAT) problem. Annual improvements are fuelled by this renewed interest and competitions have been taking place to compare the performance of different solutions [3].

One notable absence in the most common approaches is the attempt to use GPUs to aid in the computation of the solutions.

In this work we explored the current approaches and improvements developed in MaxSAT, and GPU solutions for similar problems such as SAT. We applied this knowledge to try to further improve the development of an approach that uses GPU computation at its core.

The use of GPU as a general purpose computation device, instead of its original graphics driven purpose, is now well documented [14]. It has been used in many fields of research due to its great parallel power [10, 6]. This compute capability comes mainly from how its architecture is driven to run the same type of code many times in many different cores in parallel. These cores are in fact designed to run the same piece of code for every pixel on a computer screen so it is understandable why they do have parallel computation power that makes them also useful in general purpose computing, and why we also hope to be able to harness their power [4].

With GPUs having a more generalised use in other fields we decided to explore how it already fits or could fit into the MaxSAT problem.

1.1.1 Applications

MaxSAT has been applied in many fields from optimization of schedules to health care. We highlight some examples below: being used with a model of employee schedules defined as a MaxSAT problem in order to find more efficient schedules [17]; educing synthesis of semantics of malware signatures to a MaxSAT problem as a way to identify malware family signatures [5]; in preference-based planning problems, an approach that makes use of MaxSAT

techniques as a way of, according to defined desirable properties or plan preferences, attaining the best quality for a given plan [8]; solving repair suggestions with MaxSAT in automotive configuration such as inconsistent configuration constraints or non-constructible vehicle orders. [16]; served as the base for an automatic test algorithm for cancer therapy design as a means to allow the selection of the most effective drugs [9].

1.2. Goals

As an objective goal for this work we intended to develop a solution that consists of a MaxSAT solver, which through the use of GPU would achieve the same results, but with a better performance than traditional CPU-only solvers.

This main goal can be described as smaller ones, each of them serving as a purpose for the whole project.

- develop a CPU/GPU hybrid solver;
- develop a CPU-only solver and compare with CPU/GPU hybrid solver design in order to conclude if there are advantages to the inclusion of the GPU;
- compare performance of developed designs and existing state of the art solvers;
- explore and implement further variations of solver design in order to improve performance;
- perform a critical analysis of the results.

2. Maximum Satisfiability (MaxSAT)

Conjunctive Normal Form (CNF) is a representation of a logic formula through a conjunction of clauses. A clause is a disjunction of literals where a literal is either a Boolean variable (x_i) or its negation ($\neg x_i$).

The problem of Satisfiability (SAT) is defined in logic as a problem in which, given a logic expression in CNF, we try to find if there is an assignment to the variables that satisfies (have a truth value of "true") all the clauses or prove that no such assignment exists.

We define an unsatisfied clause as a clause which has a false value as the result of the logic evaluation of its proposition. A clause with a logic truthful value (logic value "true") is defined as a satisfied clause.

Maximum Satisfiability (MaxSAT), is often described as the problem of determining, which values have to be assigned to each variable such that we maximize the number truthful clauses. Though that description of the MaxSAT problem fits the concept of maximization problem, we can think of it, and often do, as a problem of minimization. In this case we must think not on how to satisfy the most clauses, but how to minimize the number of unsatisfied clauses.

Consider the following formula:

$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_3)$

In this case, no assignment satisfies all of the clauses. However, some assignments leave less unsatisfied clauses than others. The assignments of (x_1, x_2, x_3) with values $(0, 1, 1), (1, 0, 1)$, and $(1, 1, 1)$ all leave a single unsatisfied clause when compared to the other assignments. Hence, any of these assignments are optimal solutions of the MaxSAT problem instance represented in this formula.

3. Related Work

This chapter presents some of the existing work we studied as a means of understanding existing solutions as a way to explore possible directions for our approach for solving Maximum Satisfiability (MaxSAT) with the use of a GPU. The types of work we discuss range from techniques related to past GPU uses in both MaxSAT and SAT algorithms, as well as various techniques used in various other approaches, both those restricted to specific categories of MaxSAT and those with a wider range of adaptability. In this chapter, unless otherwise indicated, it is assumed that a formula or MaxSAT instance is of the unweighted type.

3.1. MaxSAT algorithm

The main objective of this work is to develop an algorithm that makes the use of the GPU to solve MaxSAT problems with a special focus on solving benchmarks of the *random* category [3]. In this section we discuss some of the previous work in the field that we reviewed in order to inform the design of our own algorithm.

Dependent on if proof is provided by the solver to the instance's solution, we can define different types of approaches. An exact algorithm is one that guarantees and produces the proof of having found the optimal solution to the problem. There are however approaches that are called *incomplete algorithms*, in which an optimal solution is not guaranteed to be found.

Algorithms can also be classified according to other characteristics. One common type of algorithm is one that calls a SAT solver in an iterative fashion. Through the use of a SAT solver, these approaches will check for the satisfiability of the original expression with an added constraint regarding the number of unsatisfied clauses. This constraint is then set higher or lower until it finds an optimal solution. Also of notice is the branch and bound approach which, with the addition of new inference rules, makes the problem easier to solve and have been demonstrated to be effective on the *random*

and *crafted* types of benchmarks.

Branch and Bound (B&B)[1] solvers typically move through the search space in a depth-first manner. At each level of the search space the algorithm compares its *upper bound* and its *lower bound*. ***Upper bound*** is the best solution it has found so far, that is to say the assignment with the least unsatisfied clauses. The ***lower bound*** being defined as the number of unsatisfied clauses encountered so far through the current assignments of variables.

If the lower bound is greater or equal to the upper bound, the algorithm safely backtracks to try and find a better solution.

3.2. Use of GPUs in SAT algorithms

GPUs have a great capacity for performing the same computations, in parallel, on different data (vector processing). Since the 2000s this capacity has been harnessed to solve problems unrelated to graphics processing (general-purpose). This is also true for problems related to SAT. In this section we provide a brief review of previous work in this field.

In the CUD@SAT project [2] the authors explore the use of GPUs to improve performance of SAT solvers.

The approaches implemented in this project are SAT solvers based on an implementation of the DPLL algorithm. The two main design variations that were studied in this paper are described next.

In one of them, designated "mode 2", the GPU executes the lower part of a search tree, having different sub-trees for each thread, this is a stark contrast with classic GPU computing which executes the same instruction on multiple threads.

The second designed solution, also "mode 1", executes each unit propagation phase of the DPLL algorithm in parallel, by making use of the typical non-divergent thread nature of GPU computing.

3.3. Use of GPUs in MaxSAT algorithms

Munawar et al. [12] developed an implementation of a MaxSAT solver on CUDA by making use of a genetic algorithm (GA). The approach proposed by the authors works in two stages. First, on the CPU, it computes the number of random numbers needed for the whole process. With this information it generates all the random numbers on the GPU. To apply the genetic algorithm, this implementation makes use of a GPU thread for each member of the algorithm's population. The application of GA itself is pretty typical, using a model where selection and migration between neighbourhoods can occur before the recombination process computes the new generation. Mutation can also occur after recombination, within a given probability. The selection process makes use of the traditional fitness value, in the case of this algorithm the fitness value of each assignment is the number of satisfied clauses obtained when this assignment is used. After a new generation of assignments is evaluated this solution makes use of a hill-climbing algorithm as the method for local search. When finishing what constitutes the new generation the approach used replaces the parents with the child only when the child outperforms the parent in its fitness value.

Typically this local search and genetic algorithm hybrid would stop immediately when fitness did not improve, but because of the GPU architecture, all parallel threads have to exit at the same time. To help with this fact the developers made the algorithm execute a fixed number of iterations. This was done with the introduction of a feedback system which changes the number of iterations.

4. MaxSAT on GPU

At the start of the development of our solution, the decision was made to start with a prototype that could leverage the strengths of the GPU: computing multiple assignments in parallel. In this chapter we discuss how a base version centred on this idea was implemented and how we developed variations to try and improve the quality of our solution. We start with an overview of the solution digging a bit deeper in explanation of each variation in their own sections.

4.1. Overview

In our earliest prototype, which would be the basis of our solution, we started by parsing an instance of the MaxSAT problem and generating kernel code specific for that MaxSAT instance during compile time. This kernel code would then run on the GPU for a variety of different value assignments to this MaxSAT problem instance.

Inside this GPU kernel code we would evaluate how each clause was, or was not satisfied by each assignment. For every unsatisfied clause we would add one to a counter. This counter represents how many clauses are left unsatisfied in a given assignment, a lower value representing a better solution to the problem instance.

4.2. Pre-processing

Due to the operations that we decided to execute on the GPU and in order to improve the execution speed, part of the programs code is generated specifically for each different instance of the problem.

This process starts with a small C++ program that parses the CNF instance file into structures that contain the same information organized into different clauses. From this information the GPU kernel code can then be generated. In the kernel code, clause evaluations are translated to bitwise operations so that each variable of an assignment can be compared within each clause and use this

result as a means to efficiently calculate the value of unsatisfied clauses for that assignment in the GPU.

As an example if our instance is represented in CNF as the following clause:

$$x_1 \vee \neg x_2$$

When this clause is parsed as described in this section it will become C code that is able to be evaluated through a bitwise operation of the form:

```
!(((x[0]&mask1)!= 0) |
((x[0]&mask2) == 0))
```

The value assignments are stored in a way so that each variable is represented by a single bit of an integer instead of a whole integer itself. This was done primarily as performance consideration and as a way to take better advantage of the fast, but physically limited GPU shared memory. The form we chose to represent this clause for evaluation with bitwise operations instead of conditional statements such as "`if`" was also due to the better theoretical performance of GPUs when there is no existence of diverging paths between the various simultaneous threads.

With the variables stored in bits instead of integers we need a way to access the value of just one bit, we make use of a mask with bitwise *and* operation to get the value of a specific variable of all of the ones stored in the same integer. Taking a look at an example, imagine an access to the second leftmost bit of an integer. To retrieve the value of only the bit that represents the second variable, the second most bit we need to nullify everything else but that bit, we do that with a *bitwise and* operation and the right mask. Since the second variable is in fact represented by the index 1 we use a modulo operation to access an array with pre-calculated powers of two, with 1%32 we get 1, and in that position is the decimal value 1073741824, which in binary translates to `0100 0000 0000 0000 0000 0000 0000 0000`. We then come to:

```
w[0]&1073741824
```

A *bitwise and* like the one above with this value then will return whatever value is in that bit position we want, be it a 1 or a 0.

The clause itself is transformed into different evaluations according to the literals present, if a literal is negated, i.e. is true when the variable is 0 we compare it with "`== 0`" else we evaluate it against "`!= 0`." The whole clause is subject to a bitwise *or* evaluation which determines if the clause as a whole was satisfied or not. In the case it was not, the zero value is negated to one to be added to the number of unsatisfied clauses.

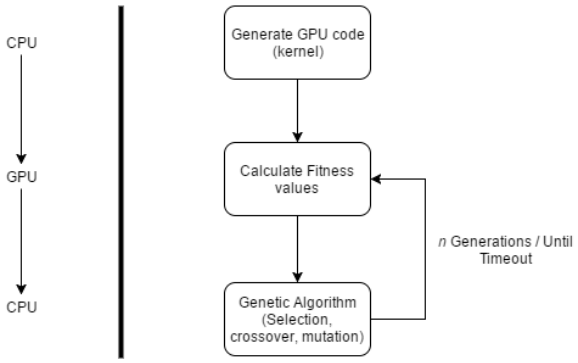

Figure 1: Solver flow for computation of values using the Genetic Algorithm

The result of multiple of these operations being pre-processed into the kernel code is that we can then compute the unsatisfied clauses very fast, just based on the outcome of this evaluation being equal to zero or one.

4.3. Use of a Genetic Algorithm

To evolve this solution we decided to introduce the use of a genetic algorithm. A high-level description of the flow of data and function interaction throughout the solver with the use of the genetic algorithm can be seen in a simple visual representation in figure 1, where we see transactions between GPU and CPU, in the following paragraphs we explain this interactions in greater detail.

As previously explained in 4.1, we start on the CPU in compile time by analysing the MaxSAT problem instance and generating the code that will allow each thread to compute the value for the evaluation of a given assignment to that specific instance of the problem. Afterwards, we run this code for each GPU thread, for each assignment, and compute the quality value for each assignment as described in section 4.1. The first set of assignment values is randomly generated on the CPU prior to being copied over to GPU where the computation is performed.

With the values of this first set of assignments computed we can start using the genetic algorithm to improve the assignments themselves. We do this by copying the results from the GPU to the CPU. In the CPU we use the information of that first set of assignments (generation) to apply the genetic algorithm. We save some of the best existing solutions and we generate new ones from the existing pool of assignments mixing parts of each of assignment with each other, with potential mutations to each assignment also being possible.

In more depth, for each new assignment/individual two parent assignments are picked from the previous generation, these are then used as the forming parts of the new assignment.

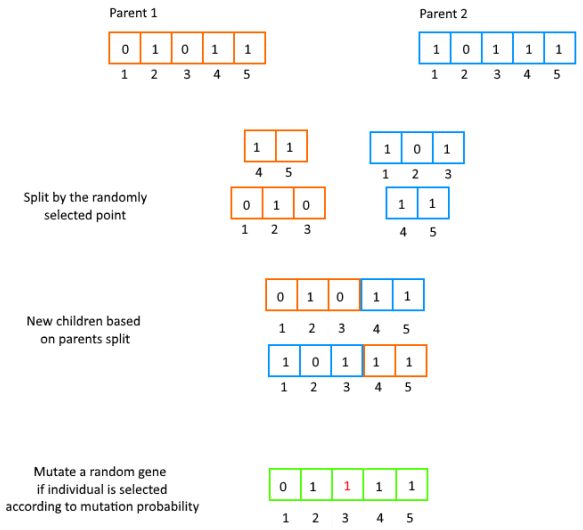

Figure 2: Our use of the genetic algorithm

To form the new assignment, a random split point is chosen and from that split point backward the new assignment will inherit from the first parent and from the split point forward from the second parent. After this part of the process the mutations take place. These mutations occur according to a rate of mutation parameter previously defined, if a given individual falls within this mutation rate, one of its variables, chosen at random, has its value changed.

These solver functions can be seen in a visual representation in figure 2. In that figure we start by seeing two different assignments, designated "parent 1" and "parent 2", these are picked randomly from the previous generation. When a division point is selected random we use the halves of both parents cut by that division point to create new children. Finally, there is the possibility of random mutation occurring as it is show happening to the third variable of the first child.

When we complete this process we order the GPU to process the values for this new generation. We then repeat this cycle for as long as we want.

On the first version of this implementation, parent assignments were picked according to their fitness level, which was derived from the unsatisfied clauses value of each assignment. However we soon found that the selection based on these values had very little significance, as most of them were too close together to make it an effective selection process. We then tried to give each assignment an even chance of being chosen, selecting them through a stochastic process with uniform probability. We observed this change had no significant impact on the quality of the best solutions found, while speeding up the process.

4.4. CPU-only implementation

Early in the development process we decided to test if using GPU would give us a potential edge against a similar implementation using just the CPU. As a means of being able to test this we implemented a version of the solver using just the CPU.

This version of the solver is very similar to the one using GPU. At the start of execution, the MaxSAT problem instance file is parsed, and subsequently from this information code is generated. This code will take the form of a function that will run for every assignment in a serial fashion whereas in the GPU enabled version of this code would form the kernel that would run in multiple threads simultaneously.

4.5. Weighted problems variation

As we explored our solver's capabilities, we decided to see if we could expand them. Therefore, we developed a variation capable of solving problems of the weighted category.

The weighted variation is very similar at its core to the unweighted one it is based on, requiring mainly small changes to the kernel code generated in pre-processing. We started by changing the parser slightly to make it able to deal with files that contain weighted clauses and store that information in our data structures. Then the main change to the kernel code was that instead of just penalizing the score of the assignment by adding one for each unsatisfied clause we added the value of entire weight of the unsatisfied clause.

4.6. Restarts

As we strove to improve the results of the solver, we implemented a variation of our solver with restart capability. Even with this capability, the flow of the solver program stayed largely the same, the main difference being that after a pre-defined number of generations with no improvement of the solution, the solver is able to restart from a new random group of assignments rather than continuing on creating assignments from the same "gene pool". This is designed as an attempt to break out of a local minimum valley and attempt to find a global minimum. This means that the MaxSAT assignments for that particular generation when the restart occurs are not generated with the use of Genetic Algorithm and instead, we apply the restart algorithm for one generation, going back to the genetic algorithm on the next one until the defined period of no improvement happens again. With the use of this restart algorithm we created two slight variations:

1. One with the use elitism, where we saved a percentage of the best individuals of the previous generation and filling the remainder of the

generation with freshly generated individuals,

2. and one without elitism where we use nothing of the previous generation and completely filled the new generation with newly generated individuals.

4.7. Assignment Repair

When looking for more techniques that could show improvements in our results, we turned to the concept of repairing assignments to improve the quality of their solution.

In this context repairing refers to the search for variable assignments with no positive action (satisfying clauses) on the outcome of the solution. If variable assignments that meet that criteria are found their values are changed so that they might provoke an improvement to the solution. We designed this variable search in a "greedy" way, looking for the first variable in a clause that satisfied said clause, and flagging the variable. Then we could use this information to change the value of all variables that were not flagged as an attempt to satisfy previously unsatisfied clauses.

To achieve this we started by modifying the kernel code in a way that in addition to previous functionality it also evaluated each piece of a clause until it found the first variable that satisfied the whole clause, it was after finding each of these variables that their data representation in an array was then flagged.

A previously defined parameter marks how many generations without an improvement of the solution must occur for the repair to happen. If the value was set to one thousand, and the solution had not improved in the last one thousand iterations we would then call a new function to apply the repair. The repair function combs through the generation and flips every variable value that was not flagged as being used to satisfy a clause to the opposite of its current value, so a 0 becomes a 1 and vice versa.

4.8. Second Repair variation

As a way to further improve the repair solution, in relation to the first approach described in 4.7, we considered how we could change the selection process of the variables to be flipped. We decided to reduced the "greediness" of the first repair variation. Instead of simply searching for the assignment that satisfied that clause, we would go for a more careful selection. Starting with a modification of the existing base kernel, when evaluating the clause for their value we added the functionality of flagging variables in unsatisfied clauses in the same pass. Then when we reach the defined point of no improvement we call a secondary kernel on the GPU which flags the variables that contribute to satisfying the clause, trying to avoid marking those also present in the unsatisfied clauses. These are only flagged when another variable of the same clause cannot satisfy it. With this information collected we perform the repair function flipping every variable value which was flagged for being present in unsatisfied clause but was not flagged for being necessary to solve another.

4.9. Hillclimb in Assignment Generation

Next, we looked into incorporating a Hillclimbing algorithm into our solver. We set up this variation in a similar way to the other versions of the solver. We start with the standard generation of random assignments and evolution of these assignments through the use of the genetic algorithm, this stops after we reach a number of generations with no improvement, the solver then switches to hillclimb mode, and for the rest of its execution time it picks the top performing individual from the current generation and fills the remaining slots in the generation with as many possible variations of a set of bits.

The hillclimb portion of the algorithm starts with filling an array with random indices to represent the variables to change the values of. This array has the size that allows the most possible variations for our generation size. Then we find the top performer from the current generation and insert it into the new generation, this individual is then copied to fill the generation. After filling the new generation, each individual is modified in the variable positions corresponding to the mentioned indices. Each individual is a different permutation because we use a value that is simultaneously incremented to contain all the variable permutations.

5. Results

As development of the solution went on we put it through testing, this was especially important when we started developing variations of the initial solver. Instead of waiting until the end of development to test all the variations, ongoing testing could allow us to change the focus of the development depending on the achieved results. At the very end of the project we did make a more considered analysis of the data and undertake further testing for comparisons when needed. The following sections of this chapter describe how we evaluated our solution and how each variation of the solver performed under this testing methodology.

5.1. Evaluation Methodology

In order to perform evaluation on our developed solution we compare its performance to existing solutions. To measure the level of performance we first and foremost concentrated on quality, which we defined as how close to the optimum solution of a problem we came. We compared some of the varia-

tions of the developed solution between themselves, such as the GPU-enabled and CPU-only solvers as well as comparing our solution to results set by CCEHC2AKMS and CCLS2AKMS solvers, which are among the best performing solvers in the type of problems we tested against.

All tests made use of the problems (or benchmarks) provided for the yearly MaxSAT Evaluation competition [3]. Our main focus was on the *Unweighted Random* variety while using *Weighted Random* instances as a secondary evaluation set.

As a tool for measurements we made use of *runsolver* [15] which allows the measurement of time in great detail, capturing standard output and sending Unix signals to interrupt the solutions within the given timeout. We ran the main benchmarks with 1000 seconds of timeout after which the solver would react to the signal and output the current state of the problem, which could range from no solution found to an optimum solution with proof in the case of complete solvers.

Hardware installed on the computer used for testing included, an Intel Xeon E5-2620 v2 CPU at 2.1 Ghz Base Frequency, 32GB of DDR3, and a Nvidia Tesla K40C GPU.

5.2. Finding the right parameters

Our solution is able to be configured to some extent, from rates of mutation (4.3) and percentage of elitism in the genetic algorithm, to the size of each generation of individuals. Given these possible variations we went ahead and tried to define values that produced the best results in order to lock them into those values and reduce the variables when testing for different problems.

We started by testing different dimensions for the groups of assignments when the kernel was called (generations). We varied both the total size of the generation when testing GPU-enabled and CPU-only, and in the case of GPU-enabled we made some additional tests where we varied only the ratio between number of threads (block size) and number of blocks (grid size). With generation sizes fixed in place we made the mutation rate fluctuate in order to try and find a value in which we could see improvements. We found that having a fifty percent chance of mutation saw a very slight improvement over rates on the lower end, but increasing past those values was unwarranted as there was no visible improvement.

As far as elitism in the genetic algorithm is concerned, very soon we realized that given the very large generations we would be better served with a smaller elitism percentage, having no benefit to the solution in having many of the top individuals stay from one generation to the next. We quickly fixed this parameter at one percent.

5.2.1 Occupancy

At this point, searching for possible performance gains we delved into exploring the concept of occupancy in Nvidia CUDA. This concept is related to the ratio of number simultaneous active warps on the multiprocessor to the maximum number of warps supported by the GPU [13].

This occupancy number is naturally dependent on the resources used by the code present in the kernel being executed on the GPU. A compiler option allows us to determine values of registers per thread used. Along with a occupancy calculator distributed by the hardware vendor (Nvidia) this allows us to better define the number of threads and blocks to achieve greater occupancy and possible performance improvement. It must be said that occupancy does not always guarantee higher performance as the kernel might not be bandwidth-limited nor latency-limited access to memory but bottlenecked by other factors. On top of that, efforts to increase occupancy beyond fifty percent see improvements diminished [7].

As we experimented with the values using this calculator we saw no significant impact on our solution, which lead us to believe that our solver was not limited in its memory access but by computation. Given there was no detected negative impact either, we would end up adjusting the solution by using the values that allowed higher occupancy.

5.3. CPU-only *vs.* GPU comparison

To evaluate the benefits of a GPU enabled solver we tested the solutions that were developed head to head. Both the GPU enabled and CPU-only variations in this test used the same methodologies and internal algorithms wherever possible. However testing with a fixed time frame data shows some difference. The following chart shows how the values of solutions for MaxSAT problems of both versions and the complete solver CCLS2AKMS are in comparison to each other.

Results showed, when on a fixed time frame with a timeout of 1000 seconds, the CPU got really close to the GPU variation. Although the GPU enabled version retains an edge we can see that the results are very close to each other.

This is due to the fact that the methodology of these solvers allows them to get fairly close to the values of the complete solver early on in the execution, but getting just slight improvements to that value afterwards is hard. This means that while running each iteration is faster on the GPU enabled version is advantage it is somewhat mitigated in this test scenario.

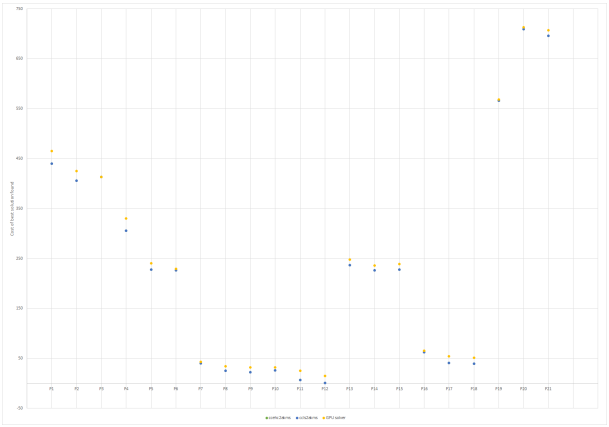

Figure 3: GPU Solver *vs.* CCLS2AKMS *vs.* CCEHC2AKMS

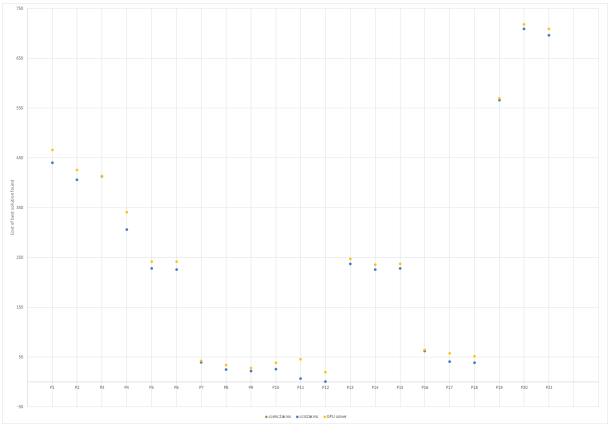

Figure 4: GPU Solver *vs.* CCLS2AKMS *vs.* CCEHC2AKMS with shorter timeout

5.4. Comparisons to the state of the art

Evaluating our solution against current state of the art solvers was always seen as a requirement early on in the project. This would allow us to see how our methodology performed against the state of the art and make us able to draw conclusions on its capabilities.

We ran our set of random category problems on both our solver and the selected state of the art solvers. State of the art solver selection was based on the 2015 MaxSAT Evaluation competition results and public availability of said solvers. With the timeout of 1000 seconds we obtained the results in the chart present in 3.

As we can see in the chart in figure 3, while CCLS2AKMS and CCEHC2AKMS overlap in their results our solution's solver comes behind in the results of testing. The percentage of deviation from the optimal result is similar regardless of the problem solved. After seeing these results, we pondered if our solver, being in the incomplete category, would fare better against the state of the art solvers in a short timeout test. This would allow it to be used in situations where you might want a good solution quickly even if it's slightly further apart from the optimal solution. With a timeout fixed at 60 seconds we took to testing.

In the testing data represented on the chart 4 we can see both state of the art solvers (CCLS2AKMS and CCEHC2AKMS) being compared to our solution. The main difference in this test when compared to the one with longer timeout is that for the majority of data in this test the complete solvers are unable to prove their solutions as optimal in the short time frame as with our solution. Despite this fact the state of the art solvers continue to outperform our approach by a small margin while continuing to achieve the same level of quality when compared to each other.

With this in mind we started to ponder on possible modifications to the solver that could close the performance gap and went on to develop the Restart and Repair variations with this objective in mind.

5.5. Weighted Variation

Although we geared the methodology of our solution toward solving unweighted problems we created a variation with the changes needed to ingest and solve weighted problems. Though we suspected performance was not going to be as good as for unweighted we decided to run a small number of tests. In weighted problems, not all clauses are valued the same, so leaving a clause with a bigger weight unsatisfied is more costly and is therefore a worse solution.

With results at hand, and a wider gap to the state of the art solver than in unweighted tests, it was clear that this solver variation had quite a way to go on the weighted front, so we decided to not pursue it further, instead exploring other possible improvements for the unweighted variation of the solver.

5.6. Restart Solution

After developing variations that enabled a restart of the assignment pool after a period of no improvement we were faced with both finding what frequency restarts should happen in and how this variation compared to one without it.

On table 1 the solver and parameter variation are representing in the following way: "R" and "RE" are the restart variations of the solvers both with elitism turned off and on respectively; the number after R and RE represents the number of generations without improvement before a restart occurs; No Restart represents the version of our solution's solver without restart enabled; P1 through P21 represent the various problems ran by the solvers;

As we can see in the table there were no sig-

Table 1: Restart Variation Test Results

	R500	R1000	RE500	RE1000	No Restart
P1	461	461	461	461	463
P2	424	424	424	424	424
P3	413	41	413	413	413
P4	328	328	331	330	329
P5	236	232	234	234	233
P6	233	230	242	241	233
P7	43	43	43	43	43
P8	33	33	33	34	34
P9	28	28	28	28	29
P10	33	35	38	38	36
P11	30	27	24	26	30
P12	9	11	11	11	11
P13	246	246	246	246	247
P14	236	236	236	236	236
P15	236	236	236	236	236
P16	65	65	65	65	65
P17	52	52	54	54	53
P18	51	51	52	52	52
P19	566	566	566	568	567
P20	707	706	706	707	713
P21	707	706	706	707	707

nificant improvements from either restart variation even with varied parameters. We are unable to make a conclusion of a clear advantage of restarts can be drawn from these narrow deviations in results.

5.7. Repair Solution

For the variations utilising repair, as described in sections 4.7 and 4.8, the results of the testing undertaken on this variations can see in table 2. These variations showed no discernible improvement over the previous variations of our approach, and neither of them has distanced itself from the other in terms of quality of the achieved results.

Table 2: Repair Variation Test Results

	Repair Variation 1	Repair Variation 2
P1	461	461
P2	424	426
P3	413	413
P4	330	331
P5	232	238
P6	241	241
P7	43	43
P8	34	34
P9	28	28
P10	36	38
P11	26	25
P12	11	15
P13	246	246
P14	236	236
P15	236	236
P16	65	65
P17	54	56
P18	52	52
P19	567	567
P20	714	714
P21	708	708

5.8. Hillclimb Solution

From the results gathered (table 3) we were able to observe a slight improvement when using hillclimb technique over the base version of the solver. However this technique was still unable to achieve the same quality of results as the state of the art solver. The lack of change in some results may be explained by the generation of individuals in the algorithm being stuck in local maxima after one is found, with the variations generated by the hillclimbing algorithm not producing a better result and not finding a better performing "hill."

Table 3: Hillclimb Variation Test Results

	Hill Variation	Genetic/No Restart	CCLS2AKMS
P1	461	463	440
P2	424	424	406
P3	413	413	413
P4	327	329	306
P5	233	233	228
P6	229	233	226
P7	43	43	40
P8	33	34	25
P9	28	29	22
P10	36	36	26
P11	17	30	7
P12	8	11	1
P13	246	247	237
P14	236	236	226
P15	236	236	228
P16	65	65	62
P17	52	53	41
P18	52	52	39
P19	566	567	566
P20	711	713	709
P21	705	707	696

5.9. Summary

Our continuous testing of the new variations of our approach as we developed them allowed us to adjust the direction of development and keep chasing possible improvements.

The results of our testing show our approach cannot match the current state of the art complete solvers in quality of assignments produced. These same results allowed us to also conclude which techniques beyond the application of a genetic algorithm did not show improvement to our approach, and this should help provide a direction for possible future work.

6. Conclusions

This thesis goal tackled the development a MaxSAT solver design with the capability to leverage the GPU's strong points.

The solution we ultimately arrived at, a solver that iterates, in a very short time frame, on a large number of possible assignments to the instance of MaxSAT problem being tested. These possible assignments were generated in a stochastic manner for

the most part, with some intelligence added with the introduction of algorithms that complemented the solution such as the genetic algorithm.

The evaluation through tests based on the benchmarks of the MaxSAT evaluation competition show that this solution is not able to go toe to toe against the very state of the art complete solvers. It was usually very close to the optimal solution, but very rarely hitting it.

We cannot however conclude from our experimentation that utilisation of GPU in solvers does not work. There may be a place for the GPU in MaxSAT, but given its very particular strengths and quirks pose challenges on how to include it in solvers, especially when competing with the ever improving solvers based on CPU.

6.1. Future Work

There are various goals and possible problems to explore for the use of GPU in MaxSAT, as it is a problem vast enough to be able to be partitioned into different categories.

For future work we present a list of items we suggest can be done as a follow up to this work: analyse which types of CPU-based MaxSAT techniques could work well when being translated to the GPU; explore further integration of *intelligence* on GPU solvers; analyse different splits on the types of work done on GPU *vs.* CPU; explore if pre-processing can be expanded to improve results during the core execution on the GPU; experiment with creating a flow of events that allows simultaneous use of GPU and CPU.

References

[1] A. Biere, M. Heule, and H. van Maaren. *Handbook of Satisfiability*, chapter 19, pages 616–617. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. `https://books.google.pt/books?id=shLvAgAAQBAJ`.

[2] A. Dal Palu, A. Dovier, A. Formisano, and E. Pontelli. CUD@ SAT: SAT solving on GPUs. *Journal of Experimental & Theoretical Artificial Intelligence*, 27(3):293–316, 2015.

[3] U. de Lleida. MaxSAT Evaluations, 2006-2015. `http://www.maxsat.udl.cat`.

[4] K. Fatahalian and M. Houston. A Closer Look at GPUs. *Commun. ACM*, 51(10):50–57, Oct. 2008.

[5] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand. Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability. *arXiv preprint arXiv:1608.06254*, 2016.

[6] N. Goodnight. CUDA/OpenGL fluid simulation. *NVIDIA Corporation*, 2007.

[7] A. B. Hayes, L. Li, D. Chavarría-Miranda, S. L. Song, and E. Z. Zhang. Orion: A Framework for GPU Occupancy Tuning. pages 18:1–18:13, 2016.

[8] F. Juma, E. I. Hsu, and S. A. McIlraith. Preference-based planning via MaxSAT. In *Canadian Conference on Artificial Intelligence*, pages 109–120. Springer, 2012.

[9] P.-C. K. Lin and S. P. Khatri. Application of Max-SAT-based ATPG to optimal cancer therapy design. *BMC genomics*, 13(6):S5, 2012.

[10] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA. *Computer Physics Communications*, 179(9):634–641, 2008.

[11] A. Morgado, F. Heras, M. Liffiton, J. Planes, and J. Marques-Silva. Iterative and Core-guided MaxSAT Solving: A Survey and Assessment. *Constraints*, 18(4):478–534, Oct. 2013. `http://dx.doi.org/10.1007/s10601-013-9146-2`.

[12] A. Munawar, M. Wahib, M. Munetomo, and K. Akama. Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework. *Genetic Programming and Evolvable Machines*, 10(4):391–415, 2009. `http://dx.doi.org/10.1007/s10710-009-9091-4`.

[13] Nvidia. Nvidia - occupancy, 2017. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#occupancy`.

[14] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[15] O. Roussel. Controlling a Solver Execution with the runsolver Tool system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:139–144, 2011.

[16] R. Walter, C. Zengler, and W. Küchlin. Applications of MaxSAT in Automotive Configuration. In *Configuration Workshop*, pages 21–28, 2013.

[17] F. Winter. MaxSAT Modeling and Metaheuristic Methods for the Employee Scheduling Problem. *TU Wien*, 2016.