A PARALLEL APPROACH TO SOLVING SATISFIABILITY PROBLEMS ON GRAPHICS PROCESSING UNITS USING NEURAL NETWORKS

by

Melih Mert

B.S, in Computer Engineering, Boğaziçi University, 2010

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2016

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Prof. Taflan Gündem for many enlightening suggestions and guidance during the development of this thesis, and helpful comments on the thesis text. Without his good will, help, and support, it would not be possible to finish this thesis.

I am grateful to Assist. Prof. B. Atay Özgövde and Assist. Prof. Arzucan Özgür for their participation in my thesis committee and for their insightful and constructive comments to improve my thesis work.

I would like to offer my deepest thanks to Gökçe Gürsöz for her invaluable advice, assistance, patience, and faith in me.

I also offer my gratitude to my colleagues who are also my dear friends Mustafa Ekmen and Barış Akşanlı for their friendship, ideas, and support.

Last, I would like to thank my family members for their patience and support through all the phases of my education.

ABSTRACT

A PARALLEL APPROACH TO SOLVING SATISFIABILITY PROBLEMS ON GRAPHICS PROCESSING UNITS USING NEURAL NETWORKS

Graphics Processing Units (GPUs) have become popular for parallelization of general purpose applications recently. GPUs are composed of huge number of powerful processors in a readily available package. The Satisfiability Problem (SAT) is one of the earliest NP-complete problems. The solution to SAT has various application areas, including automated theorem proving, circuit design, artificial intelligence, and software verification. Although many algorithms exist to experimentally solve SAT, they are not believed to be efficient on all SAT instances. In this thesis, we propose a novel GPU based parallel approach using neural networks as the algorithm selection mechanism to solve the SAT. We demonstrate speedups of up to 3 times on benchmarks by choosing the correct algorithms (solvers) to solve sub problems to reach the final result in our system.

ÖZET

SİNİR AĞLARI KULLANARAK GRAFİK İŞLEME ÜNİTELERİ ÜZERİNDE GERÇEKLENEBİLİRLİK PROBLEMLERİ ÇÖZME İÇİN PARALEL BİR YAKLAŞIM

Grafik İşleme Üniteleri (GİÜ'ler) son zamanlarda genel amaçlı uygulamaların paralelleştirilmesi konusunda popüler olmuştur. GİÜ'ler çok sayıda güçlü işlemciden oluşup hazır paket olarak sunulmaktadır. Gerçeklenebilirlik Problemi (GP) bilinen en eski NP-karmaşıklık problemlerinden biridir. GP çözümünün otomatik teorem ispatı, devre tasarımı, yapay zeka ve yazılım doğrulama gibi çeşitli uygulama alanları vardır. GP'yi deneysel olarak çözen birçok algoritma olmasına karşın, bunların tüm GP örnekleri üzerinde etkili olduğuna inanılmamaktadır. Bu tezde, GP'yi çözmek için algoritma seçim mekanizması olarak yapay sinir ağlarını kullanan yeni bir GİÜ tabanlı paralel bir yaklaşım öneriyoruz. Bizim sistemimizde, nihai sonuca ulaşmak için yapılan deneyler üzerinde oluşturulan alt problemlerin, doğru algoritmalar (çözücüler) seçilerek çözülmesi ile 3 kata kadar hızlanmalar olduğunu gösteriyoruz.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

A_p	Primary best algorithm to solve the problem
A_s	Secondary best algorithm to solve the problem
C_{dh}	Communication overhead to transfer results from device to host
C_{hd}	Communication overhead to transfer input files from host to device
F^l_{max}	Maximum frequency of literals
F^l_{min}	Minimum frequency of literals
M_g	Global memory
M_s	Shared memory
N^d_{avg}	Average number of disjoint clauses
N^s_{avg}	Average number of similar clauses
N_{core}	Number of CUDA cores
N_c	Number of clauses
N_l	Number of literals
N_{lu}	Number of unique literals
N_t	Number of threads
T_{cpu}	Total time spent on CPU
T_f	Time spent to read input problem file
T_d	Time spent to divide input problem file into sub problem files
T_{gpu}	Total time spent on GPU
T_q	Time spent to interrogate all sub problem files
T_{qi}	Time spent to interrogate sub problem file i
T_{si}	Time spent to solve sub problem file i

LIST OF ACRONYMS/ABBREVIATIONS

API	Application Programming Interface
CDCL	Conflict-Driven Clause Learning
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
LBD	Literals Blocks Distance
LOWL	Literal Occurrence Weight Learning
MID	Message Passing Inspired Decimation
MP	Message Passing
NN	Neural Network
SAT	Satisfiability Problem
SLS	Stochastic Local Search

1. INTRODUCTION

Graphics Processing Units (GPUs) are gaining popularity for parallelization of general purpose applications. The emergence of GPU devices in reasonable prices and practical programming environments for general purpose applications such as Compute Unified Device Architecture (CUDA) by NVIDIA [2] have increased the number of GPU based parallel applications for various types of problems. CUDA has been developed as an extension to C/C++ languages. Later on, Java extension [6] has also been published. CUDA has a wide range of libraries and methods to facilitate memory management, thread management, and code analysis in order to achieve speedups. General Purpose GPU (GPGPU) computing with CUDA has various application areas (beyond its original purpose of graphics processing [7]) including but not limited to bioinformatics, computational finance, electronic design automation [8], and computational chemistry.

Satisfiability Problem (SAT) is one of the most important problems in computer science from theoretical point of view. SAT is not only interesting for theoretical purposes, but it has also several practical implementations in different areas such as computer-aided design, where it is used for automatic test pattern generation, equivalence checking, symbolic model checking, and timing analysis [9]. Due to the theoretical and practical interest of the problem, there had been many algorithms proposed in the literature. The Davis-Putnam-Logemann-Loveland (DPLL) algorithm [4] is one of the best well known algorithms, which has been the bases of many other heuristics. It is a complete, backtracking-based algorithm for deciding the satisfiability of propositional logic formula in conjunctive normal form (CNF), i.e. for solving the CNF-SAT.

Modern SAT solvers have at least two basic steps in solving the problem [10]. First, they check if there are any conflicting clauses in the original problem. Second, certain SAT solvers use the restart strategy where the SAT solver abandons the current search tree (without completing it) and starts a new one. Another approach used in modern SAT solvers is parallelizm. There are two main approaches in parallelization:

using multi-core/multi-threaded architecture and using distributed architecture over network connection. Parallel algorithms such as PSATO [11], Satz [12], GridSAT [13], and PaMira [9] use distributed architecture. Since the communication is done over network, they have loose integration between SAT solving units. Alef [14] and DMSAT [15] are examples of parallel SAT solvers targeting multi-core/multi-threaded architecture. It's been stated that most of these SAT solvers work efficiently on specific SAT instances, in other words, they are unable to solve all types of problems in reasonable amount of time [16].

In this thesis, we propose a novel GPU based parallel approach using neural networks as the SAT solver selection method to solve any type of SAT correctly by achieving speedups. SAT solvers available in the literature are optimized to solve only certain type of problems. Our solution solves all types of SAT that have been known to be solved in the literature (SAT Competition Benchmarks). To the best of our knowledge, there is no known algorithm that uses neural networks as the solver selection mechanism in collaboration with multi-threaded architecture of GPUs. We have demonstrated the effectiveness of our design with several experiments that use the benchmarks from SAT Competition 2014 [5].

1.1. Contributions of This Thesis

Our key contributions in our thesis can be summarized as follows:

- We propose a novel approach to solve SAT on GPUs using neural networks. Our approach uses neural networks as the decision mechanism to solve the SAT using the best possible algorithms on GPUs and achieves to discover the primary or secondary best algorithms correctly in approximately 90% of the experiments.
- We have designed a parallel system that divides the initial SAT into sub problems in order to allocate each sub problem to a CUDA core on GPU to solve them in parallel. We achieve speedups of up to 3 times.
- Our solution solves all types of SAT that have been known to be solved in the literature (SAT Competition Benchmarks) unlike other solvers which are optimized

to solve only specific type of problems.

1.2. Thesis Outline

The outline of this thesis is as follows:

First, we discuss the underlying concepts (satisfiability problem, CUDA architecture, and neural networks.) of the proposed solution in Chapter 2.

Next, in Chapter 3, we review the literature and summarize related work on SAT solving.

In Chapter 4, we present our novel SAT solving approach, namely *A Parallel Approach To Solving Satisfiability Problems on Graphics Processing Units Using Neural Networks*. The system model and algorithm (problem division, distribution, parallel interrogation and solution, and final calculation) are explained in detail.

The performance evaluation is provided in Chapter 5. This section explains experimental results based on the benchmarks and comments on various scenarios validated using our approach.

Finally, Chapter 6 concludes the thesis by summarizing and discussing our results, and rendering some research directions and open problems for parallel and effective SAT solving context.

2. BACKGROUND

2.1. Satisfiability Problem

Satisfiability Problem (SAT) is one of the most basic problems in computer science. Its essential goal is to determine the set of literals of a given Boolean formula in such a way that the formula evaluates to true. If this is the case, the formula is called satisfiable. On the other hand, if no such assignment exists, the problem expressed by the formula is false for all possible literal assignments and then it is called unsatisfiable. Since a literal can only be in two states, true or false, there are $2^{N_{lu}}$ assignment sets which exponentially grow when the number of unique literals increase.

SAT is one of the first problems that was proven to be NP-complete. This means that all problems in the complexity class NP, which includes a wide range of decision making and optimization problems, are at most as difficult to solve as SAT. There is no known algorithm that efficiently solves SAT, and it is generally believed that no such algorithm exists; yet this belief has not been proven mathematically.

There are several special cases of the Boolean satisfiability problem in which the formulas are required to have a particular structure. A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses (or a single clause). For example, "$x1$" is a positive literal, "$\neg x2$" is a negative literal, "$x1 \wedge \neg x2$" is a clause, and "$(x1 \vee \neg x2) \wedge (\neg x1 \vee x2 \vee x3) \wedge \neg x1$" is a formula in conjunctive normal form. Determining the satisfiability of a formula in conjunctive normal form where each clause is limited to at most three literals is NP-complete also; this problem is called 3-SAT. 3-SAT can be generalized to k-SAT when formulas in CNF are considered with each clause containing up to k literals. Figure 2.1 (barrowed from [1]) shows the graphical representation a 3-SAT instance.

In this thesis, we use SAT formulas in CNF. These benchmarks are obtained from SAT Competition 2014 [5] to validate our approach.

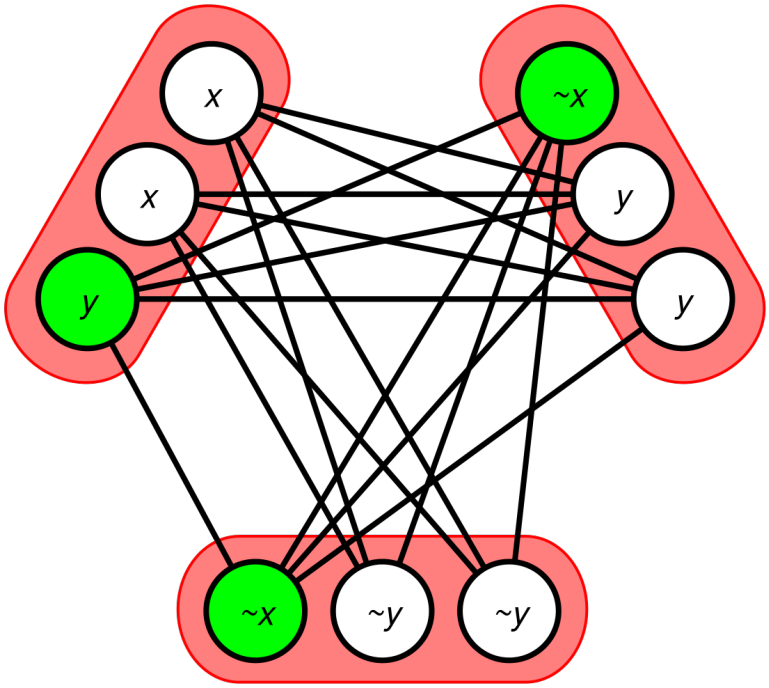

Figure 2.1. 3-SAT instance with satisfying assignment x=false, y=true [1].

2.2. NVIDIA CUDA Architecture

CUDA (Compute Unified Device Architecture) is co-designed hardware and software to expose the computational horsepower of NVIDIA GPUs for GPU computing. The GPU is a compute device which serves as a co-processor for the host CPU, has its own device memory on the card and executes many threads in parallel. In addition, GPU computing is used in various areas such as computational chemistry, computational finance, image processing etc. GPU computing can speed up applications up to 150 times [17].

The smallest unit in CUDA is a thread. Thousands of threads can work concurrently at a time. Since CUDA enables us to share data between threads, redundant computations shall be avoided via thread cooperation. A batch of threads forms a block in CUDA architecture. Threads within a block can cooperate within the shared memory. A CUDA block can have at most 1024 threads and each thread has a unique id. A batch of blocks forms grid in CUDA architecture. All threads in all blocks have

unlimited access to the global memory, that is threads in different blocks can cooperate within the global memory. Grid can have at most 2 dimensions. Threads within a block has unique built-in block id. Parallel portions of an application are executed on the device as kernels. A kernel is executed by a grid of thread blocks. One kernel is executed at a time. Figure 2.2 (barrowed from [2]) shows the high level architecture of host (CPU) and device (GPU).

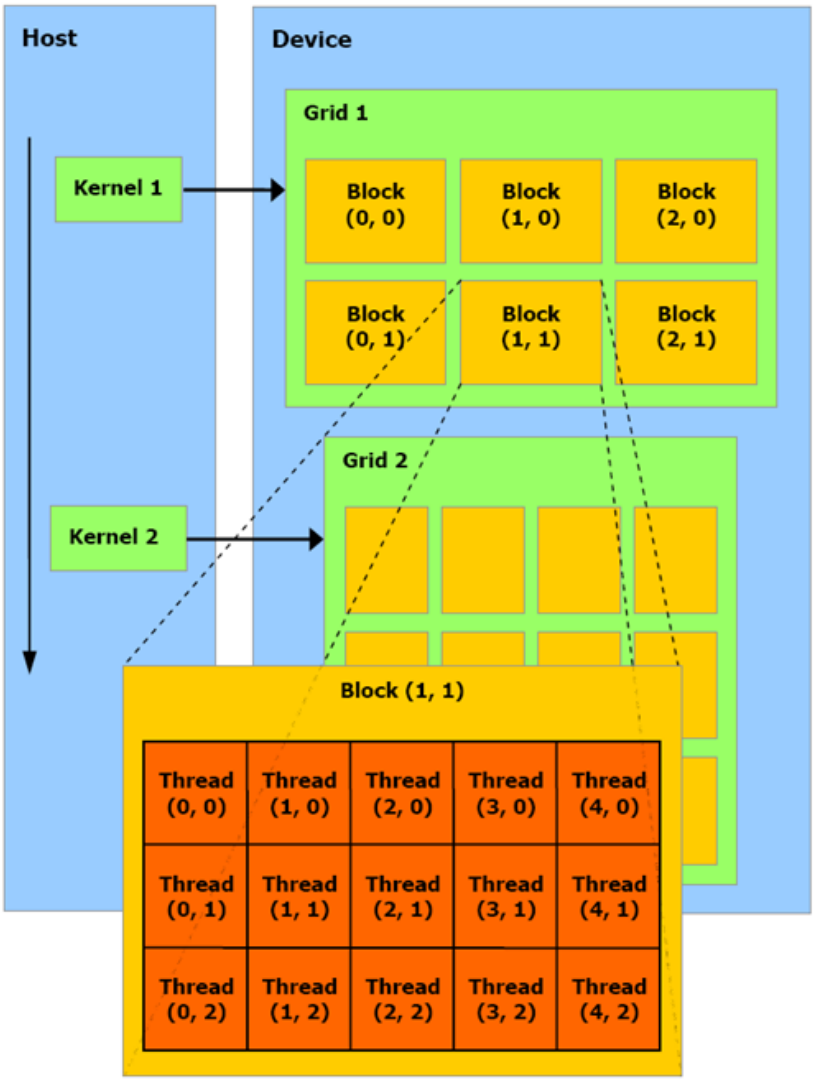

Figure 2.2. High level architecture of host and device [2].

CUDA Architecture provides different types of memory spaces available to threads during the execution. Each thread has a private local memory in addition to the registers allocated to them. In addition to those, each thread block has a shared memory

visible to all threads of the block within the lifetime of the block. Furthermore, each thread has access to the same global memory throughout the application. There are also certain special memory types other than the mentioned above. Read-only texture (for storing concurrent thread specific values), constant (for storing constants), and page-locked memory (for providing data to multiple kernels, etc.) spaces are defined by this architecture to be specialized in the host to meet the needs of a fast memory. All types of memory spaces are limited in size, so they should be handled carefully in the program. Figure 2.3 (barrowed from [2]) illustrates the memory types of CUDA.

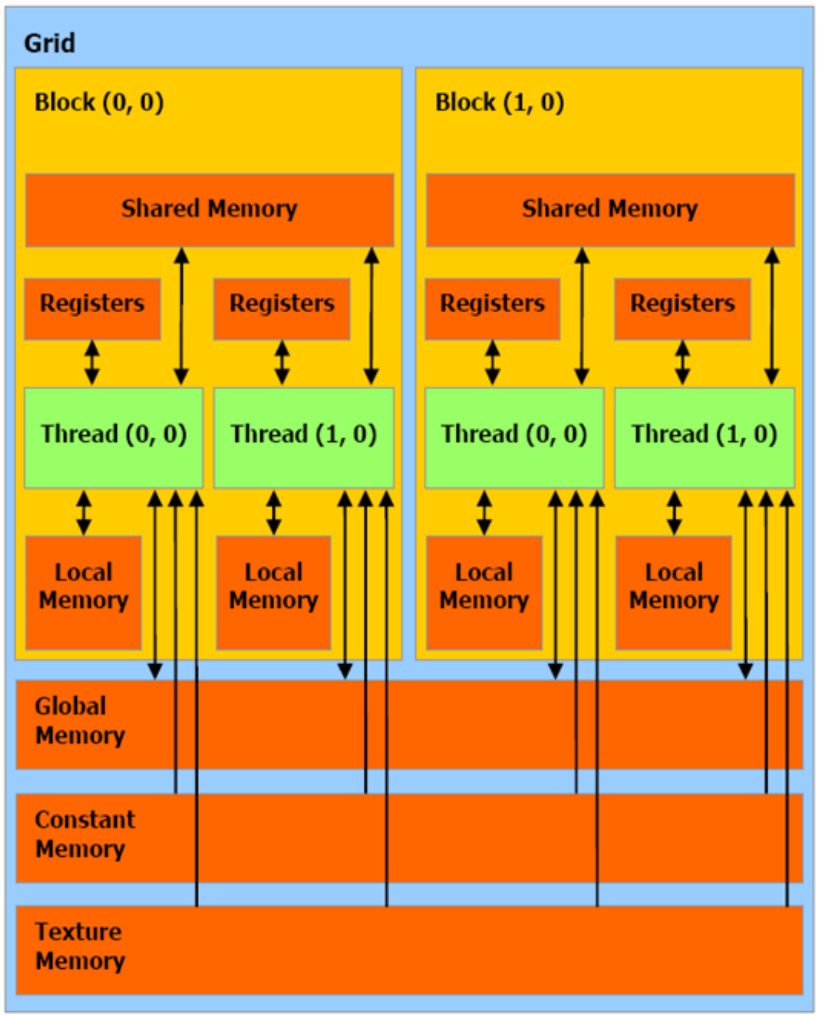

Figure 2.3. Memory types available in CUDA architecture [2].

In this thesis, we have performed our experiments on GeForce GTX 560M GPU designed by NVIDIA. This device has 192 CUDA cores and it means that 192 efficient

threads blocks can be used in parallel execution [18].

2.3. Neural Networks

Neural Network (NN) is a simulation of biological neural system in computer science. Given a number of parameters each of which corresponds to an input, it determines what the outputs will be. Neural Network has three layers; input layer, hidden layer, and output layer. Each input layer takes one of the given parameters. This layer interprets these parameters and sends them to hidden layer with edges (synapses) between every input and hidden layer. Pre-trained NN instance decides the output mappings by checking all combinations it has learnt so far. A simple neural network is shown in Figure 2.4 (barrowed from [3]).

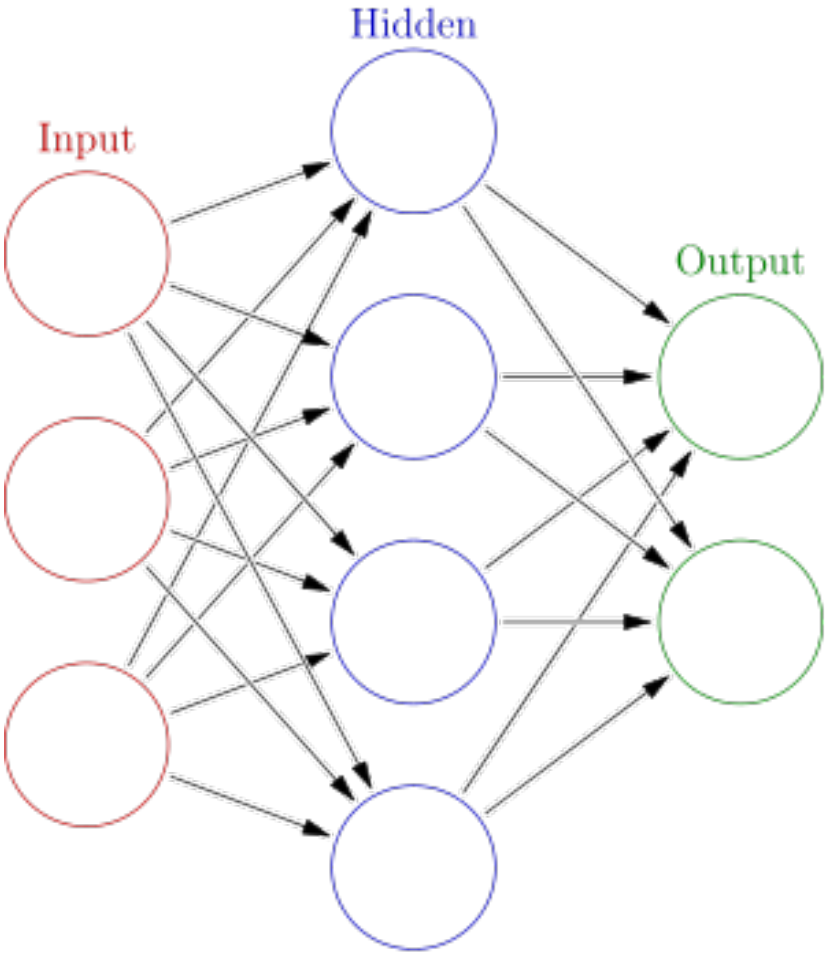

Figure 2.4. A simple neural network [3].

In this thesis, we use Joone as the neural network framework [19]. It is an open source project that offers a highly adaptable neural network for developers. Our approach uses 7 different input parameters and 2 different output parameters in the neural network scheme. In order to train the neural network instance, we have applied benchmarks selected from SAT Competition 2014. We have obtained 4 different neural network instances to use in our experiments by performing training with various combinations input and output parameter sets, and different set of benchmarks. The list of neural network training parameters and trained neural network objects are explained in Section 4.3 in detail.

3. RELATED WORK

In this section, related work is briefly explained. To the best of our knowledge, there is no known solution that uses neural networks as the decision method in solving satisfiability problem and integrates it with GPUs. Moreover, none of them can work efficiently on all problem types. The following work has been examined in order to provide an overview of approaches in the literature and indicate the differences of our work.

3.1. General Techniques

In this section, we briefly explain general techniques that are used in SAT solving.

3.1.1. Unit Propagation

This technique simplifies the SAT formula by using unit clauses that are composed of a single literal. Let's say the formula contains the unit clause **a**.

- Every clause containing **a** (other than the unit clause) is removed.
- In every clause that contains ¬**a**, this literal is deleted.

For example; if we apply these two rules of unit propagation to the following CNF formula respectively, it becomes simplified smaller formula: "$a \wedge (a \vee \neg b) \wedge (\neg a \vee b \vee c) \wedge (\neg c \vee d)$" $\rightarrow$ "$a \wedge (\neg a \vee b \vee c) \wedge (\neg c \vee d)$" $\rightarrow$ "$a \wedge (b \vee c) \wedge (\neg c \vee d)$"

3.1.2. Pure Literal Elimination

This is another simplification technique which is useful and commonly applied in SAT solving.

- If a literal occurs with only one polarity in the formula, it is called pure.

- Pure literals can be assigned in a way that makes all clauses containing them true.
- Thus, these clauses can be deleted.

For example; if we apply pure literal elimination to the following CNF formula, it is simplified as shown: "$a \wedge (b \vee \neg c) \wedge (a \vee c \vee d) \wedge (\neg c \vee d) \wedge (\neg e \vee a)$" $\rightarrow$ "$(b \vee \neg c) \wedge (\neg c \vee d)$"

3.1.3. Conflict-Driven Clause Learning (CDCL)

This technique aims to find a satisfying assignment by removing conflicts. A candidate assignment for a literal is chosen. Then, CDCL checks whether it creates a conflict in other clauses based on previous assignments. If it finds a conflicting situation, it backtracks to the most effective level of the search graph, therefore another candidate assignment can be tested from this level. If it does not lead to any conflict, CDCL continues with choosing assignment for the next literal. The key steps of the technique are given below:

- Select a literal and assign true or false.
- Apply unit propagation. Build the implication graph.
- If there is any conflict, analyze it and non-chronologically backtrack.
- Otherwise, continue from step 1 until all variables are assigned.

An example CDCL is shown in Section 3.2.2 where a CDCL based solver is described.

3.1.4. Stochastic Local Search (SLS)

This technique moves from one solution to the next one in the search space by applying local changes in order for getting closer to the optimal solution. Here are the important steps of SLS that can be applied in SAT solving:

- Choose a candidate truth assignment.
- Check satisfiability of clauses by the assignment.
- Count the number of satisfied clauses.
- Move to one of the neighbor assignments to maximize the number of clauses satisfied by the assignment.

3.1.5. Davis–Putnam–Logemann–Loveland (DPLL) Algorithm

DPLL algorithm runs by choosing a literal, assigning a truth value to it, simplifying the formula and then recursively checking if the simplified formula is satisfiable. If this is the case, the original formula is satisfiable; otherwise, the same recursive check is done assuming the opposite truth value. This algorithm makes use of unit propagation and pure literal elimination techniques in its simplification part. The DPLL algorithm can be summarized in Figure 3.1 (barrowed from [4]).

```
Algorithm DPLL
  Input: A set of clauses Φ.
  Output: A Truth Value.

function DPLL(Φ)
   if Φ is a consistent set of literals
       then return true;
   if Φ contains an empty clause
       then return false;
   for every unit clause l in Φ
      Φ ← unit-propagate(l, Φ);
   for every literal l that occurs pure in Φ
      Φ ← pure-literal-assign(l, Φ);
   l ← choose-literal(Φ);
   return DPLL(Φ ∧ l) or DPLL(Φ ∧ not(l));
```

Figure 3.1. DPLL Algorithm [4].

3.2. SAT Competition Algorithms

In this section, we explain certain SAT solvers that have participated in SAT Competition, their algorithmic structures and key parameters, the techniques they use, and their improvement points.

3.2.1. Dimetheus

The functionality that the Dimetheus solver uses is separated into various phases of solving which each one can apply strategies to achieve their respective tasks as shown for search phase in Figure 3.2 (barrowed from [5]).

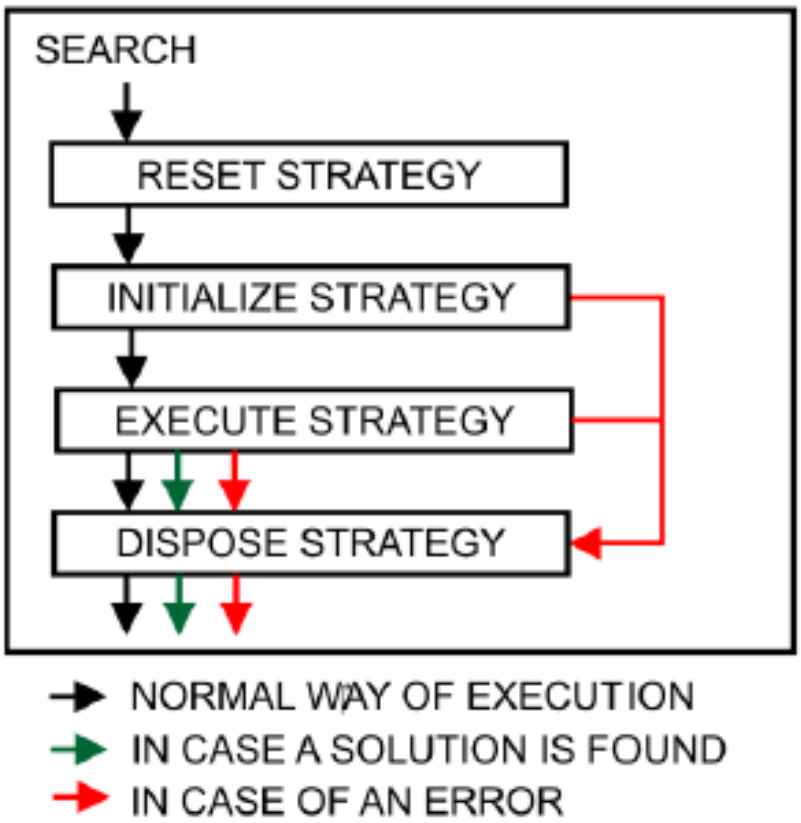

Figure 3.2. Dimetheus search strategy phase [5].

First, the solver performs a simple pre-processing before solving the formula. This pre-processing consists of unit propagation, pure literal elimination, failed literal

elimination, and etc.

Second, it applies Message Passing (MP). The application of MP is commonly done in order to provide biases, which are estimators for the values of variables in the solution. These biases are used to perform Message Passing Inspired Decimation (MID) in order to simplify the formula. On the other hand, the MP algorithm can be used to perform literal occurrence weight learning (LOWL). In contrast to MID, the LOWL approach does not assign values to variables. It merely influences how MP computes biases. The Dimetheus solver can dynamically decide which approach (either MID or LOWL) it uses in order to solve a formula. The decision which of the techniques is used is done based on a random forest classification of the formula.

Third, the solver should require search because MID or LOWL can not determine a satisfying assignment by themselves, the solver will apply stochastic local search (SLS). The SLS approach of the solver follows the ProbSAT approach [20].

This solver is efficient to find results for random type of SAT because its algorithm completes execution for small-sized problems in tolerable amount of time. However, it takes too much time to finish execution for the other types of problems. Moreover, it sometimes ends up in unknown results when application or hard problems are given as SAT input.

3.2.2. MiniSat-ClauseSplit

The main idea of this solver is that short clauses are usually simpler to solve than long clauses. For example, the average case complexity of random k-SAT increases in k. In conflict-driven clause learning (CDCL) solvers [21], short clauses are more useful, because they can be used for unit propagation sooner than long clauses. Long clauses pose the risk that the inconsistency of variable assignments that are bound to lead to a conflict will only become apparent after a large number of steps. Therefore, it makes nearly impossible to prune useless parts of the search space early on. An example CDCL solution is shown in Figure 3.3 (barrowed from [5]).

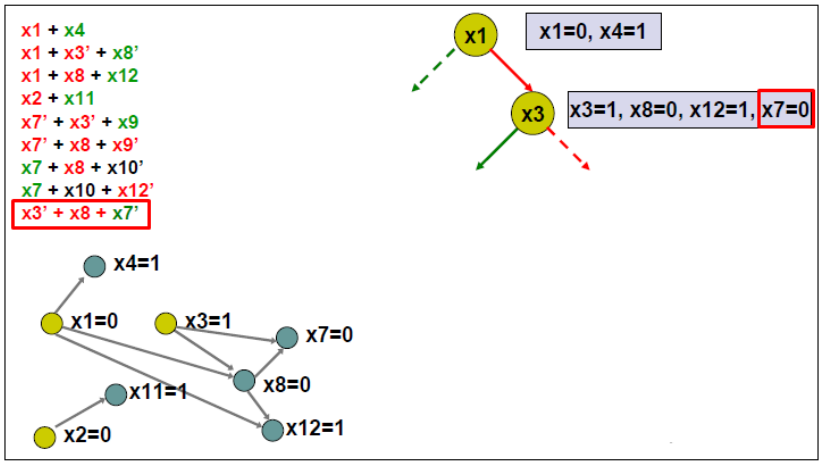

Figure 3.3. An example CDCL solution [5].

In order to mitigate the negative effect of long clauses, this solver splits them into shorter ones. The following parameters play important roles in MiniSat-ClauseSplit:

- Determine lower and upper thresholds on the lengths of clauses that should be split. Too short clauses should not be split because splitting them will not decrease the length (e.g., splitting a clause of length 3 would result in a clause of length 2 and one of length 3). Too long clauses should not be split because that would largely increase the number of clauses. Hence, only clauses whose original length is between the two thresholds should be split.
- Set target clause length for splitting. This is the length of the clauses that will result from splitting, except for the last one which may be shorter.
- Whether the optional clauses of length 2 should be generated.

The values for these parameters are tuned based on experiments with benchmarks. MiniSat-ClauseSplit tries to eliminate long clauses that drop between thresholds to optimize solution time. However, it exceeds the time limit specified by the algorithm for a given SAT formula when the formula has a different structure and has several clauses just outside of the threshold boundaries.

3.2.3. CLAS

CLAS [22] is a parallel solver that executes the SLS solver Sparrow in parallel to the CDCL solver Pcasso for partitioned search space. Since Pcasso runs a CDCL solver in parallel to its search space partitioning, CLAS represents a portfolio of an SLS solver, a CDCL solver, and a look-ahead based search space partitioning solver. During the parallel execution of these solvers, no information is exchanged between Sparrow and Pcasso.

In CLAS, three solution approaches are combined: CDCL, look ahead, and SLS. Sparrow uses the same configuration as in the original structure of sequential solver. Before Sparrow is executed, co-processor is used to simplify the formula. One core of the CPU is reserved for Sparrow. The remaining 11 cores are used for Pcasso, which also uses co-processor to simplify the input formula. The simplification techniques are the same as for the SAT solver Riss, which was also a sequential solver.

CLAS implements a parallel approach that uses two solvers and a simplification technique in order to achieve the optimal solution. However, it does not consider the type of problem which may affect the performance of the algorithms it includes, that's why it only works effectively in solving hard problems.

3.2.4. SparrowToRiss

The solver SparrowToRiss [23] combines the SLS solver Sparrow with the CDCL solver Riss. For both solvers a separate formula simplification is executed with co-processor. Then, Sparrow is run for at most 900 seconds, or until 500 million flips are executed. If Sparrow cannot solve the formula within these limits, then the formula is passed to Riss, which tries to solve the formula.

Sparrow is a clause weighting SLS solver that uses promising variables and probability distribution based selection heuristics. It updates the weights of unsatisfied clauses in every step. Sparrow arranges restarts and the smoothing parameter is

changed for each restart. Therefore, Sparrow effectively acts as a portfolio solver of different configurations where the solver itself manages scheduling the different strategies.

This solution has been focused on only hard instances of the benchmark set. In addition to that, it does not offer any tuning or optimization techniques for random or application type of problems. Therefore, SparrowToRiss has not been successful in solving problems from these two types, which is also confirmed in our experiments.

3.2.5. ProbSAT

ProbSAT solver is an efficient implementation of the probSAT algorithm [20] with slightly different parametrization. The ProbSAT solver is a pure stochastic local search (SLS) solver based on the following algorithm shown in Figure 3.4 (barrowed from [5]).

Algorithm 1: ProbSAT Algorithm

```
Data: SAT Formula F, maxTries, maxFlips
Result: Satisfying assignment A or UNKNOWN
for i = 0 to maxTries do
    A ← RandomlyGeneratedAssignment;
    for j = 0 to maxFlips do
        if A satisfies F then
            return A;
        end
        C ← RandomlySelectedUnsatClause;
        for x in C do
            compute f(x,A);
        end
        var ← RandomVariablexfromProbabilityFunction;
        flip(var);
    end
end
return UNKNOWN;
```

Figure 3.4. ProbSAT stochastic local search algorithm [5].

ProbSAT uses only the break values of a variable in the probability functions which can have an exponential or a polynomial shape. The parallel solver ProbSAT (pprobSAT) starts n instances of ProbSAT in parallel and returns the result once one of the solvers have found a solution. The last two instantiations of ProbSAT use restarts after 10^7 flips, 10^8 flips respectively.

The solver ProbSAT is implemented to solve random type of problems efficiently. It does not state any configurations or techniques for application and hard types. On the other hand, ProbSAT uses the same solver for all instances of the problem with different configurations. It takes the result of the solver which is the fastest. The execution of the same problem with different configurations on a distributed environment

results in communication overhead to transmit input files and collect results.

3.2.6. Pcasso

The SAT solver Pcasso is a parallel SAT solver based on partitioning the search space iteratively and using clause sharing among the nodes as shown in Figure 3.5 (barrowed from [5]). Pcasso proceeds by creating and solving partitions of the input instance.

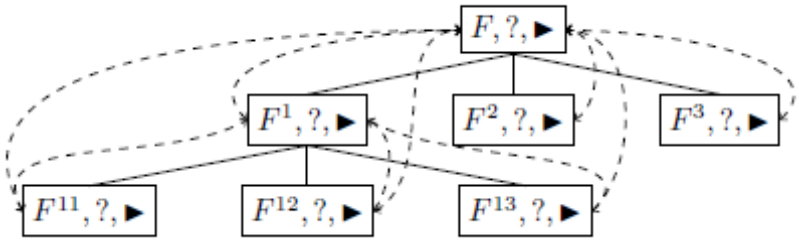

Figure 3.5. Visualization of a partition tree with clause sharing and overlapped solving [5].

The major parameters of the solver specify the number of threads that should be used, the number of partitions that should be created for each node, and how sharing should be performed. Pcasso use 12 threads, and produce 8 partitions. Furthermore, it shares learned clauses among the nodes. Finally, the treatment of the only-child scenario (where there is only one unsolved node at some partition level) can be specified as well. There are only minor magic constants that control the run time of the look-ahead procedures that are applied during partitioning, whose values have been chosen according to the literature.

Each node in the partition tree is associated to a pool of shared clauses, where a pool is implemented as a vector of clauses. This lets us separate the life of a shared clause from the life of the learnt clause. Instead of tagging each clause with a position, clauses are tagged with integers representing a level in the partition tree (root node has level zero). It observes that this is sufficient to simulate the position based approach,

that is, each thread working over a node can only access the pools for the nodes at determined positions in the algorithm. Concurrent access to pools is regulated by read-write locks.

The solver Pcasso uses the advantage of parallelizm when there are at least 12 cores which can run 12 efficient threads. It requires a powerful execution environment to be able to perform and complete solution in tolerable amount of time.

4. A PARALLEL APPROACH TO SOLVING SATISFIABILITY PROBLEMS ON GRAPHICS PROCESSING UNITS USING NEURAL NETWORKS

In this section we present our novel approach in solving the satisfiability problem making use of GPU and neural network decision methods. Our approach consists of 5 main parts. The first part is for the division of the SAT problem into sub problems which can be solved in parallel using the GPU CUDA cores. The second one is for the distribution of the sub problems to the CUDA cores. In the third part, each CUDA core interrogates its own sub problem in order to find the primary and the secondary most time efficient solvers using the neural network instance. In the fourth part, each CUDA core solves its own sub problem in parallel to other cores using the algorithms determined in the third part so that the results of sub problems can be found. Finally, sub results are sent to the CPU for the computation of the result of the original SAT formula. High level flow of our system is shown below in Figure 4.1.

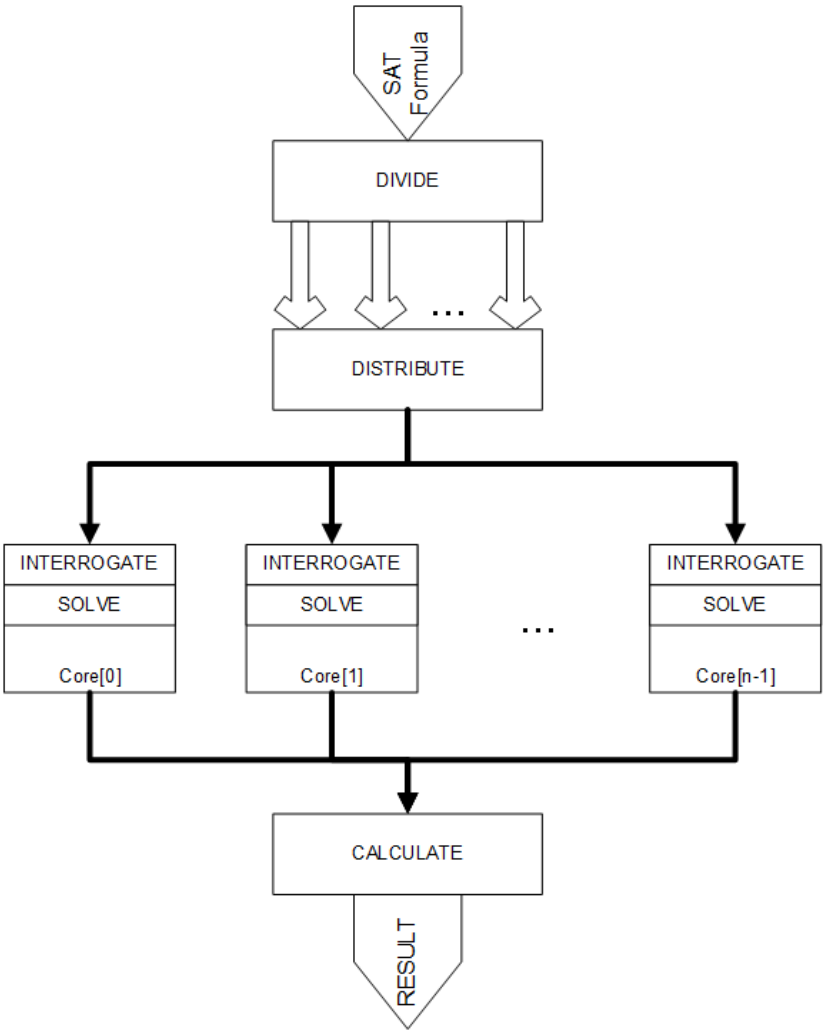

Figure 4.1. High level flow of our system.

4.1. Divide the Original SAT into Sub Problems

Since one of our goals is to decrease the processing time, we divide the original SAT problem into sub problems. Dividing SAT into 2^n sub problems where n is calculated according to the Equation 4.1 is one of the essential parts of our approach. Algorithm given in Figure 4.2 shows our recursive method to divide the problem. It recursively calls itself twice with two newly generated clause sets and the number of divisions decremented by 1 until the system obtains the targeted number of sub problems obtained from equation.

$$\boxed{n = \log_2(N_{core})} \tag{4.1}$$

Algorithm 2: divideProblem

```
Data: SAT: clauses & Number of division iterations: n
Result: 2^n Sub Problems
if n := 0 then
    return 0;
else
    if n := 1 then
        ▷ Two new clause sets and two files to save them are created;
        create newClauses1 & newClauses2;
        create tempFile1 & tempFile2;
            ▷ Given clauses is divided into two new clause sets;
        generateNewClauses(clauses, newClauses1, newClauses2);
        ▷ Two clause sets are saved to files to create sub problems;
        CreateSubProblem(newClauses1, tempFile1);
        CreateSubProblem(newClauses2, tempFile2);
    else
        create newClauses1 & newClauses2;
        generateNewClauses(clauses, newClauses1, newClauses2);
            ▷ Call recursively twice with newly generated clauses;
        divideProblem(newClauses1, n-1);
        divideProblem(newClauses2, n-1);
    end
end
```

Figure 4.2. Divide the original SAT algorithm.

The algorithm shown in Figure 4.3 is used to generate two separate clauses from

a given clause set. This function clones the same clause set to two different newly created clause sets. After creating new sets, it finds the literal that has the maximum frequency in the original clause set. One of the newly created clause sets includes all the clauses that include selected literal as a new clause. The other one is made up of those clauses as well, however it includes the negation of that literal as a new clause. Then, unit propagation is applied to each sub problem in order for simplification. In the next iteration, the algorithm searches for the literal that has the next maximum frequency for the original clause set that contains it and applies the same operations for division and simplification.

Algorithm 3: generateNewClauses

Data: Input SAT and 2 sub problems; newClauses1 and newClauses2

Result: Two new clauses created from a given clause set

▷ Clone original clause set to 2 new clause sets;

newClauses1 ← cloneClauses(clauses);

newClauses2 ← cloneClauses(clauses);

▷ Choose the literal with maximum frequency;

literal ← chooseLiteral(clauses);

▷ Create 2 new clauses using the selected literal;

clause1.add(literal);

clause2.add(negated(literal));

▷ Insert new clauses to the clause sets respectively;

newClauses1.add(clause1);

newClauses2.add(clause2);

▷ Apply unit propagation to sub problems;

newClauses1 ← unitPropagation(newClauses1, literal);

newClauses2 ← unitPropagation(newClauses2, negated(literal));

Figure 4.3. Generate new clauses algorithm.

The algorithm to select the literal with maximum frequency is given in Figure 4.4. If the problem consists of more than 10^5 clauses, all clauses are not scanned to find the literal with the next highest frequency in order not to increase our processing time too

much. We search 10^5 clauses at most and count the occurrences of literals that belong to them. The key part of this method is to keep the list of all selected literals in order not to choose them again in further division iterations.

Algorithm 4: chooseLiteral

Data: Input SAT: clauses

Result: Returns the literal with the next maximum frequency

▷ Count occurrences of literals and return the one with the next highest freq;

lit ← findLiteralWithMaximumFreq(clauses);

selectedLiterals.add(lit) ▷ Add the literal to the list of selected literals;

return lit;

Figure 4.4. Choose literal with the next highest frequency algorithm.

4.2. Distribute Sub Problems to GPU Cores

We use GPU architecture in parallel execution of the interrogation and solution parts of the system. Since device (GPU) memory is separated from the host (CPU) memory, it is required to move sub problems to device memory to enable CUDA cores to access them. In addition to that, thread blocks in each CUDA core work much faster when they are using data from shared memory of their spaces. If shared memory of a thread block is out of available space, the block starts using space from global memory which is slower than shared memory, but has much larger capacity.

Sub problems that are created from the original SAT formula are mapped to blocks in CUDA. Each CUDA block has a unique Id to allocate sub problems. We use dynamic memory allocation methods of CUDA (cudaMalloc, cudaMemcpy, etc.) to allocate available space for sub problems. Using shared memory is the first choice of mapping when there is enough space. When shared memory is full, global memory has been used for the rest of the problem.

The following algorithm shown in figure 4.5 briefly explains how sub problems are mapped to GPU memory.

Algorithm 5: distributeSubProblems

Data: Sub problems generated as the result of divideProblem algorithm 4.2

Result: Sub problems are distributed to device memory for each CUDA core

```
▷ Create object references to host and device sub problems;
Clause *subProblem_h, *subProblem_d;
▷ Write sub problem (created in division part) into temp file;
initializeSubProblem(&subProblem_h, tempFile);
▷ Calculate size of sub problem to use it for device memory allocation;
size_t problemSize ← numOfClauses * sizeof(Clause);
▷ Allocate space in device memory;
cudaMalloc(&subProblem_d, size);
▷ Copy sub problem from host to device memory;
cudaMemcpy(subProblem_d, subProblem_h, problemSize,
cudaMemcpyHostToDevice);
▷ Execute sub problems to interrogate and solve in parallel;
▷ executeSubProblems method is explained in Figure 4.7;
executeSubProblems<<<numBlocks, blockSize>>>(subProblem_d, size);
▷ Calculate size of result table;
size_t resultSize ← numBlocks * blockSize * sizeof(int);
▷ Copy results from device to host;
cudaMemcpy(result_h, result_d, resultSize, cudaMemcpyDeviceToHost);
```

Figure 4.5. Distribute sub problems algorithm.

4.3. Interrogate Sub Problems with Neural Network Object

Interrogation is one of the essential parts of our solution since it allows us to find the primary and secondary best solvers to solve the sub problem. In order to do that, we generate a neural network instance which has been trained by using different types of parameters obtained from SAT Benchmarks. In NN training, we have used 85% of the benchmarks. The remaining 15% have been used in our experiments to validate our solution. Types of parameters used in the neural network training are determined

by our solution after we analyze the structure of those benchmarks.

In NN training, we generate a training data set using the SAT benchmarks which we select for this operation. For each SAT file, we calculate the values of 7 input and 2 output parameters, and they are added to the training set in order to be used to train our NN object. After the training set is prepared, NN monitor object is created with the parameters of learning rate (0.8), momentum (0.3), total circles (10000), and learning (true) to initialize and start training. Trained NN object is generated for interrogation part after the training is completed.

The list of neural network input & output training parameters and their descriptions are given in Table 4.1.

Table 4.1. Neural network training parameters.

Parameter	Description
Problem Type	The category of SAT formula in CNF from SAT Competition 2014: Random, Application, Hard.
Number of Literals	The number of literals in a given SAT formula.
Number of Clauses	The number of clauses in a given SAT formula.
Maximum Frequency of Literals	The number of maximum occurrence of a literal in a given SAT formula.
Minimum Frequency of Literals	The number of minimum occurrence of a literal in a given SAT formula.
Average Number of Disjoint Clauses	The average number of clauses which differ in every literal, calculated by selecting a random clause and a sample of 100 SAT files from the same problem type.
Average Number of Similar Clauses	The average number of clauses which have at least one common literal, calculated by selecting a random clause and a sample of 100 SAT files from the same problem type.
Primary Best Algorithm	The algorithm that solves a given SAT in the shortest time.
Secondary Best Algorithm	The algorithm that solves a given SAT in the second shortest time.

As a result of using various combinations of parameters mentioned above, 4 different neural network objects are created in order to find the one that is the most efficient in deciding the correct solver. The list of neural network objects and their descriptions are given in Table 4.2.

Table 4.2. Neural network objects.

Neural Network Object	Description
NNetSat1.obj	Trained with full benchmarks (85%) and first 5 input parameters. It takes 6 solvers into account when deciding output solver.
NNetSat2.obj	Trained with full benchmarks (85%) and all 7 input parameters. It takes 6 solvers into account when deciding output solver.
NNetSat3.obj	Trained with full benchmarks (85%) and all 7 input parameters. It takes 9 solvers into account when deciding output solver.
NNetSat4.obj	Trained with half of benchmarks (50%) and all 7 input parameters. It takes 9 solvers into account when deciding output solver.

Interrogation is the method to find the primary and secondary best solvers of a given SAT formula. The solvers are taken from SAT Competition 2014. The list of solvers is given below in Table 4.3. The first 3 best solvers are selected according to the 3 competition criteria (random, application, hard). In total we have 9 possible solvers for a given SAT formula. In case a better solver for a category is introduced, we can plug in this new solver to our solution. The existing (old) solver can be removed in order to keep the solution up to date.

Table 4.3. The list of SAT solvers.

ID	Name	Category
1	Lingeling	Application
2	Minisat	Application
3	Glucose	Application
4	Dimetheus	Random
5	Pprobsat	Random
6	March	Random
7	Gluesplit	Hard
8	Sparrow	Hard
9	Treengeling	Hard

Interrogate algorithm uses the neural network object and evaluation parameters of the input SAT formula to decide the best two possible solvers that can solve the formula. The algorithm is summarized in Figure 4.6.

Algorithm 6: interrogateProblem

Data: SAT clauses: file

Result: Primary A_p and Secondary A_s solvers

▷ Scan SAT file to find inputs to NNet instance;

inputs := NNetTrainingSetCreator.getInputData(file);

N_l := inputs.literals;

N_c := inputs.numberOfClauses;

F^l_{max} := inputs.maxFreq;

F^l_{min} := inputs.minFreq;

N^d_{avg} := inputs.avgDisjointClauses;

N^s_{avg} := inputs.avgSimilarClauses;

parameters.add(N_l, N_c, F^l_{max}, F^l_{min}, N^d_{avg}, N^s_{avg});

▷ Read NNet instance;

NNet satObj := readNeuralNetworkObject(objectPath);

NeuralNetworkListener listener;

listener.inputFile(file);

listener.outputFile(output);

▷ Start interrogation;

satObj.interrogate(listener, inputs, parameters, solvers);

Figure 4.6. Interrogate sub problems algorithm.

4.4. Solve Sub Problems in Parallel

Once sub problems are created, they are mapped to CUDA cores. In parallel solution part, the sub problems are interrogated in parallel to find the best solvers. After interrogation is completed, our solution runs these solver algorithms on the sub problems that they match. The solution uses the idea to divide the original problem into several pieces so that each CUDA core can have 1 sub problem to solve. Since GPU devices can have different number of CUDA cores, the calculation of how many sub problems will be created is decided at the run time.

The following algorithm shown in Figure 4.7 briefly describes how sub problems are interrogated and solved at each CUDA core in parallel.

Algorithm 7: executeSubProblems

Data: Sub problems that are mapped to CUDA memory.

Result: Sub problems are interrogated and solved.

▷ Primary and secondary solvers are identified after interrogation;

▷ interrogateProblem method is explained in Figure 4.6;

solvers ← interrogateProblem(subProblem[blockIdx.x][]);

▷ Primary and secondary solvers are loaded from executables;

algo1 ← (SatAlgorithm)algorithms.get(solvers[0]);

algo2 ← (SatAlgorithm)algorithms.get(solvers[1]);

▷ Both solvers start execution at the same time;

▷ Whichever finds the solution first terminates the execution;

▷ solveProblem method is explained in Figure 4.8;

if *threadIdx.x := 0* **then**

result[blockIdx.x][threadIdx.x] ← solveProblem(subProblem[blockIdx.x][], algo1);

return;

else

if *threadIdx.x := 1* **then**

result[blockIdx.x][threadIdx.x] ← solveProblem(subProblem[blockIdx.x][], algo2);

return;

end

end

Figure 4.7. Execute sub problems algorithm.

SAT can have three different results according to the SAT Competition: satisfiable, unsatisfiable, and unknown. Satisfiable means that there is a set of assignment of literals so that the formula evaluates to true. Unsatisfiable is that there is no such assignment, that is to say, all assignments calculate the formula to false. Algorithm

returns unknown when it is unable to solve the formula. Each CUDA core executes the selected solvers on their own sub problems in parallel to find the result. Then, each of them writes the result of the execution to the assigned space of it in global memory.

The algorithm to solve a given SAT formula with the selected solver is shown in Figure 4.8.

Algorithm 8: solveProblem

```
Data: SAT: subProblem & Solver: algorithm
Result: Solves the sub problem with the given solver and return the result.
result[blockIdx.x][threadIdx.x] = algorithm.execute(subProblem);
if result[blockIdx.x][threadIdx.x] := SatAlgorithm.SATISFIABLE then
    return SATISFIABLE;
else
    if result[blockIdx.x][threadIdx.x] := SatAlgorithm.UNSATISFIABLE then
        return UNSATISFIABLE;
    else
        if result[blockIdx.x][threadIdx.x] := SatAlgorithm.UNKNOWN then
            return UNKNOWN;
        else
            return ERROR;
        end
    end
end
```

Figure 4.8. Solve sub problems algorithm.

4.5. Calculate Final Result

In CUDA cores, each block finds the result for its own sub problem and writes the output to its assigned space in the global memory. After all blocks complete this operation, sub result matrix is written back to the host memory of the CPU to calculate

the final result.

If one of the sub problems that are solved by CUDA cores have satisfiable result, the original SAT formula is satisfiable. In case all of the sub problems have unsatisfiable results, the original SAT formula is unsatisfiable. We have performed experiments with the benchmarks which have not been used in the neural network training part (15%). The results of our experiments are cross checked with the result of these benchmarks taken from SAT Competition.

5. PERFORMANCE EVALUATION

Our approach in solving the SAT (using our neural network object as solver selection mechanism and the GPU device as enhanced processing tool) has been evaluated according to the ratio of choosing the correct solver and the speedup in execution time.

In the following sections, we explain the types and structures of benchmarks used in our experiments first. Secondly, we summarize the properties of neural network objects created and trained in order to be used in our tests. We see that the success ratio of the most efficient NN object is approximately 90%. In the third section, the execution times of our solution compared to the times of best solvers for a subset of our experiments are listed. We achieve speedups of up to 3 times in these experiments as can be seen in the third section as well. Fourthly, the changes in execution time of our solution and best solver in the literature with the increasing number of variables and clauses are shown graphically for different type of problems.

5.1. Experimental Results

5.1.1. Benchmarks and Solvers

In our experiments we have used benchmarks and solvers from SAT Competition 2014 [5]. We use 85% of these benchmarks in NN training operation and the remaining 15% for the experiments to test our solution. In this competition, 3 different main problem types are available: random, application, and hard. Random formulas have been generated according to an algorithm defined by the evaluation criteria of SAT Competition. They are fairly easy and take shorter time to solve when compared to application and hard problem types. They contain $50 - 5000$ different variables and $500 - 50000$ clauses on average. Application type of problems are the formulas that have been used in the industry by companies that apply techniques for hardware and software verification. These types contain much greater number of variables and clauses compared to random and hard types. It also takes longer time to solve them. Hard

types are manually optimized to make problems harder to solve. There are SAT solvers which are optimized to solve them in optimum time so it is essential that correct solver has been chosen. It is known that the smaller clause to variable ratio of a SAT is the harder to solver it. The number of files used for each problem type and their level of difficulty are given in Figure 5.1.

ID	Category	Files	Ratio	Easy	Medium	Hard	Too-Hard
1	Random	225	3.96 - 87.79	90	90	45	0
2	Application	300		0	110	156	34
3	Hard	300		0	44	207	49

Figure 5.1. Benchmark distribution based on the problem type.

5.1.2. SAT Solver Selection Using Different Neural Network Objects

Our approach uses trained neural network instances in order to decide which algorithm will be used to solve a given SAT formula. We have generated 4 different NN objects based on the parameters and benchmark sets explained in Section 4.3.

Figure 5.2 shows a subset of our experiments and the decisions made by different neural network objects in these experiments. It compares the selection of our approach with the best selected solver in the competition. NNetObj3, fully trained NN instance with 9 solvers and 7 parameters, is the most successful one in choosing the correct SAT solver. It selects the correct solver with approximately 90% success rate when considering all the experiments performed. NNetObj4, half trained NN instance with 9 solvers and 7 parameters, has not been as efficient as instance 3 because of limited training set used to generate it. This comparison indicates that the coverage of training set has an effect on neural network performance in selecting the correct SAT solver. NNetObj1 and NNetObj2 have nearly 85% success rate. The difference between them is the number of input parameters they have used during the training. NNetObj2 has used all 7 parameters whereas NNetObj1 has been trained with the first 5. According to the decision results of these two instances, we can conclude that the last two parameters used in training have less effect on decision success rate compared to the other parameters.

Experiments show that neural network instances used in our approach have successfully detected the correct algorithm in approximately 90% of the tests. It is important to solve a given SAT with the correct algorithm and at the shortest possible time.

Exp ID	Formula	Type	NNetObj1	NNetObj2	NNetObj3	NNetObj4	Best Solver
1	unif-k3-r4.04-138.cnf	R	4	4	**4**	4	**4**
2	unif-k4-r9.6-312.cnf		4	4	**4**	6	**4**
3	unif-k3-r3.96-744.cnf		4	4	**4**	6	**4**
4	AProVE07-03.cnf	A	1	1	**1**	1	**1**
5	atco_enc2_opt2.cnf		2	2	**1**	1	**1**
6	SAT_dat.k75.cnf		1	1	**1**	3	**1**
7	crafted_n10_d6.cnf	H	7	8	**7**	8	**7**
8	ndist.b.27485.cnf		7	7	**7**	7	**7**
9	shift1add.24955.cnf		7	7	**9**	9	**7 & 9**
10	EDP3-50000.cnf		7	7	**9**	9	**7 & 9**

Figure 5.2. Solver selection using our neural network objects.

5.1.3. Execution Times and Speedups

In this section, we compare the execution times of our approach with the execution times of the best solvers from SAT competition to see the effect of the division and parallel running parts of our approach. As mentioned before, the correct selection of the solver has also improved the execution time. In Figure 5.3, execution times and speedups for a subset of our experiments are listed. The formulas that show how the execution time is calculated are given in Equation 5.1, 5.2, and 5.3.

The first equation 5.1 shows how the time spent on CPU is calculated. It adds up the time spent to read the input SAT file, the time to divide the problem into sub problems, and the communication overhead to transfer sub problems to GPU memory. The final parameter (α) is to indicate the time to make the final calculation after receiving results of sub problems.

$$\boxed{T_{cpu} = T_f + T_d + C_{hd} + \alpha} \tag{5.1}$$

The second equation 5.2 is for calculating the time spent on GPU. It sums the maximum of interrogation and solution times of CUDA cores. It also adds the communication overhead to transfer results from GPU memory to CPU memory.

$$\boxed{T_{gpu} = \max(T_{qi}) + \max(T_{si}) + C_{dh}} \tag{5.2}$$

In the third equation 5.3, these 2 times are summed to find the total execution time.

$$\boxed{T_{total} = T_{cpu} + T_{gpu}} \tag{5.3}$$

For random type problems consisting of relatively high number of clauses (¿ 10.000), we achieve speedups of up to 2 times. However, our solution executes a bit slow while running on small-sized random problems that contain less than 100 variables. Application type problems are the ones that have the highest number of variables and clauses so we expect to have the highest speedup. In such problems, division and interrogation parts consider more time than those of random and hard problems. Therefore, our gains are decreased due to the time spent in division and interrogation.

Exp ID	Formula	Type	Best Solver (ms)	NNetObjs	Our Approach (ms)	Speedup
1	unif-k3-r4.04-138.cnf	R	16183	NNetObj1	12684	**1,28**
				NNetObj2	13112	**1,23**
				NNetObj3	13194	**1,23**
				NNetObj4	12789	**1,27**
2	unif-k4-r9.6-312.cnf	R	24161	NNetObj1	12594	**1,92**
				NNetObj2	12636	**1,91**
				NNetObj3	12032	**2,01**
				NNetObj4	12765	**1,89**
3	unif-k3-r3.96-744.cnf	R	1880	NNetObj1	2991	**0,63**
				NNetObj2	3427	**0,55**
				NNetObj3	3491	**0,54**
				NNetObj4	3265	**0,58**
4	AProVE07-03.cnf	A	76410	NNetObj1	49980	**1,53**
				NNetObj2	49988	**1,53**
				NNetObj3	49135	**1,56**
				NNetObj4	48994	**1,56**
5	atco_enc2_opt2.cnf	A	26385	NNetObj1	26121	**1,01**
				NNetObj2	26299	**1,00**
				NNetObj3	26242	**1,01**
				NNetObj4	26155	**1,01**
6	SAT_dat.k75.cnf	A	73622	NNetObj1	74781	**0,98**
				NNetObj2	75112	**0,98**
				NNetObj3	75965	**0,97**
				NNetObj4	75477	**0,98**
7	crafted_n10_d6.cnf	H	83753	NNetObj1	48328	**1,73**
				NNetObj2	52872	**1,58**
				NNetObj3	47311	**1,77**
				NNetObj4	49171	**1,70**
8	ndist.b.27485.cnf	H	53311	NNetObj1	44643	**1,19**
				NNetObj2	46732	**1,14**
				NNetObj3	44581	**1,20**
				NNetObj4	44650	**1,19**
9	shift1add.24955.cnf	H	47938	NNetObj1	45812	**1,05**
				NNetObj2	44891	**1,07**
				NNetObj3	45121	**1,06**
				NNetObj4	42678	**1,12**
10	EDP3-50000.cnf	H	213780	NNetObj1	98213	**2,18**
				NNetObj2	103766	**2,06**
				NNetObj3	71127	**3,01**
				NNetObj4	98357	**2,17**

Figure 5.3. Execution times and speedups.

We achieve speedups of up to 3 times for hard type of problems that have high number of clauses (> 1.000) and high clause to variable ratio (> 10). Hard problems take too much time to solve if you choose the wrong solver. Therefore, it is critical to decide the correct solver in interrogation part for these problems to achieve speedups.

5.1.4. Change of Execution Time with Increasing Number of Variables and Clauses for Different Problem Types

In this section, we present a set of plots in order to visualize the change in execution time when the number of variables and clauses increase. The blue line on plots represent the execution times of our approach; whereas the red line shows the solution time of the best solvers.

It is also very important to observe the performance of our approach when it is tested with the benchmarks that include large number of variables and clauses up to 10^6 in total.

In figures 5.4 and 5.5, the execution times of our solution (blue) and the best solver (red) in the literature with the increasing number of variables and clauses are shown for random type of problems. As can be seen from the graphs, the performance of our solution is better than the best solution in the literature for the random problems that have $500 - 2.500$ variables and $5.000 - 30.000$ clauses. The execution time does not change too much after 30.000 clauses, so it has a very low rate of increase for the SAT files with large number of clauses up to 100.000.

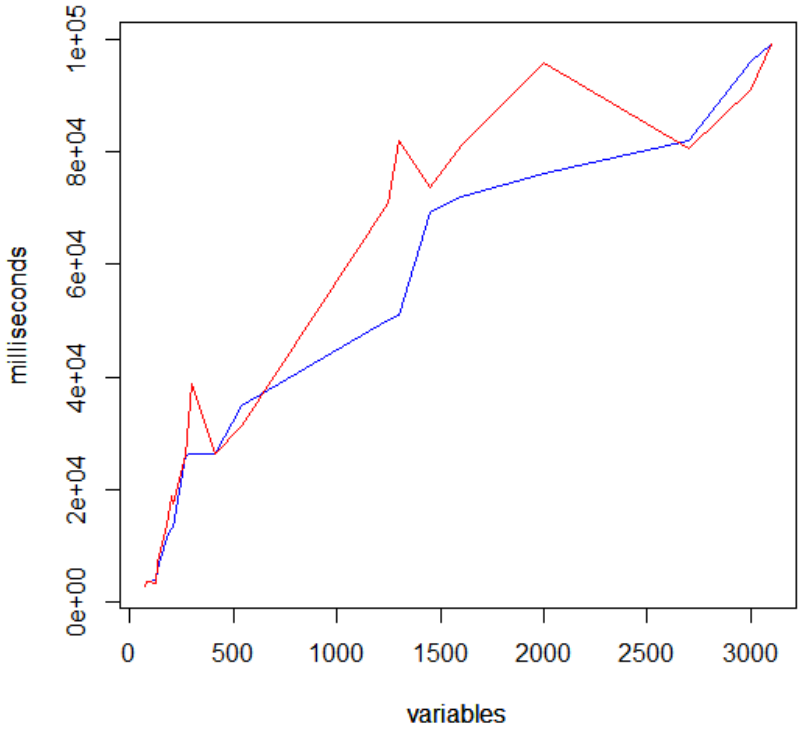

Figure 5.4. Random type, Execution time vs The number of variables

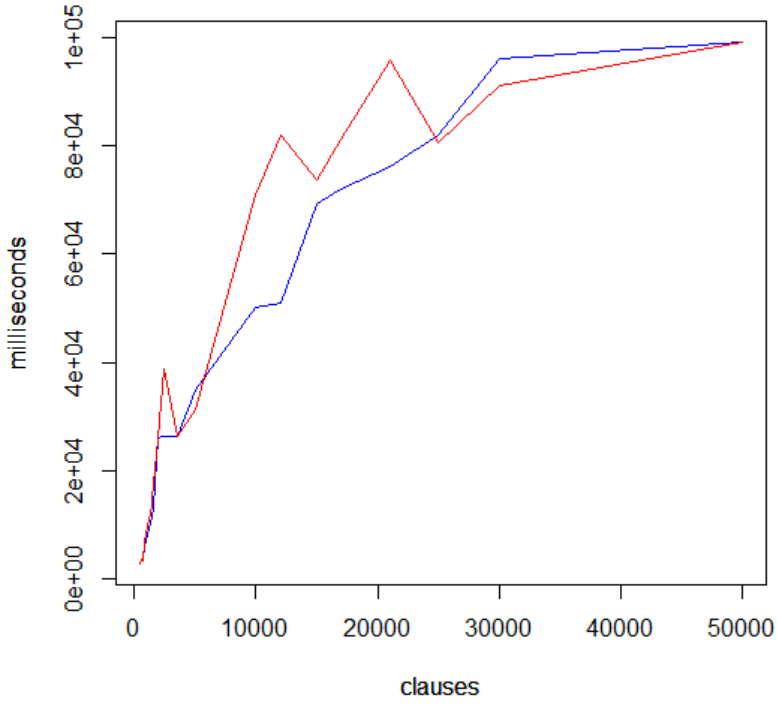

Figure 5.5. Random type, Execution time vs The number of clauses

In figures 5.6 and 5.7, the execution time shows a dramatic increase until they reach saturation point of 130.000 variables and 400.000 clauses for the application type of problems. After that point, it becomes steady and does not change too much.

As we see from the results of our experiments, solving application type problems takes longer than random type of problems. Our solution achieves better execution times for the problems that have less than 50.000 variables and 200.000 clauses. For larger problems, our approach (blue) has very close execution times to the times of best solver (red).

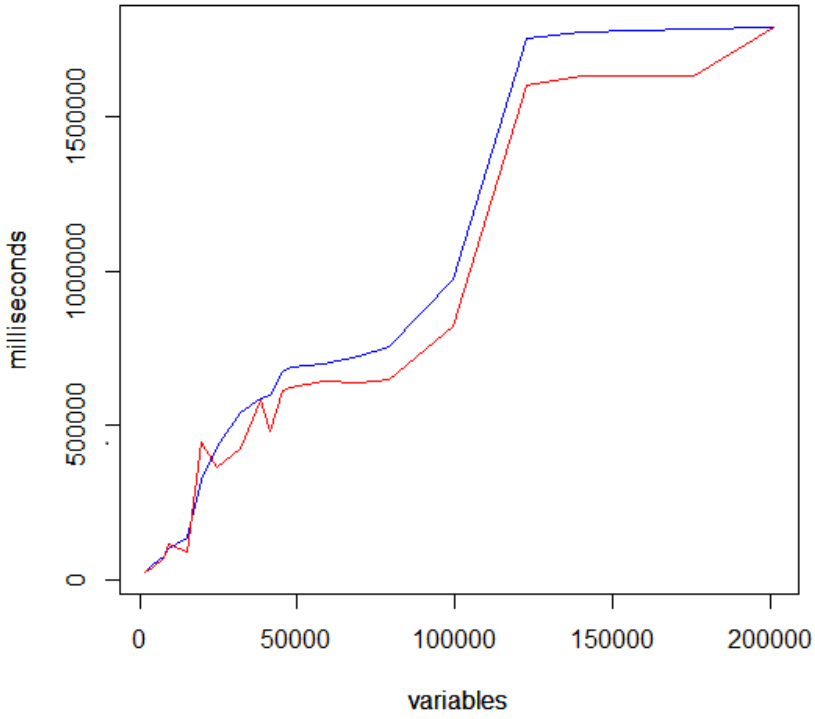

Figure 5.6. Application type, Execution time vs The number of variables

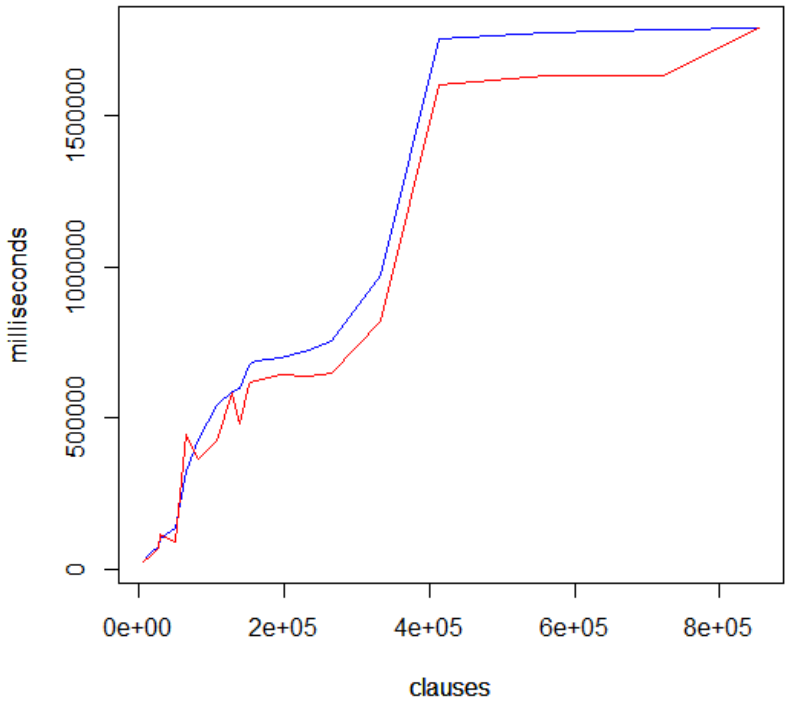

Figure 5.7. Application type, Execution time vs The number of clauses

The following figures 5.8 and 5.9 shows the trend of execution times for hard type of problems. Our approach achieves speedups for the problems that have more than 1.300 variables and 4.000 clauses.

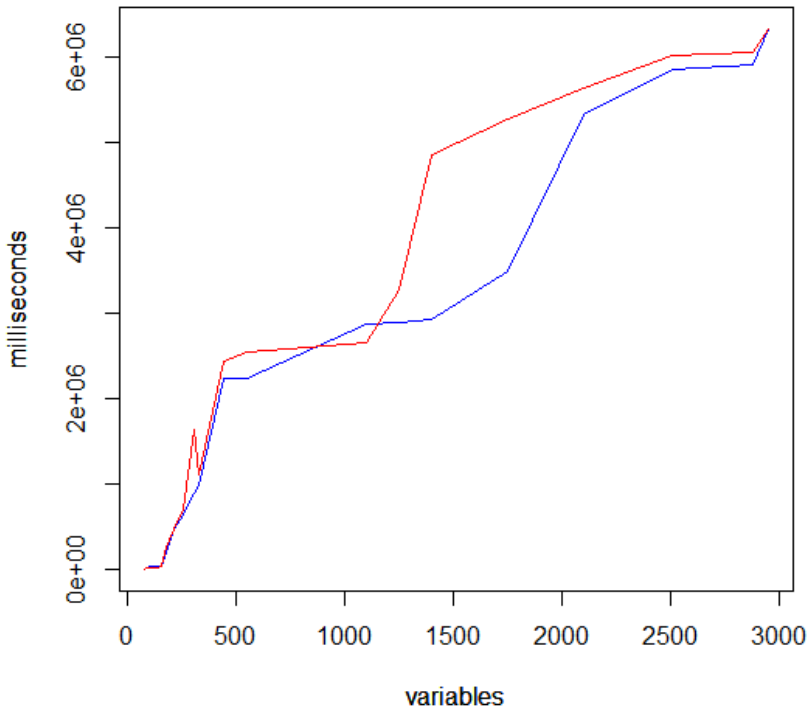

Figure 5.8. Hard type, Execution time vs The number of variables

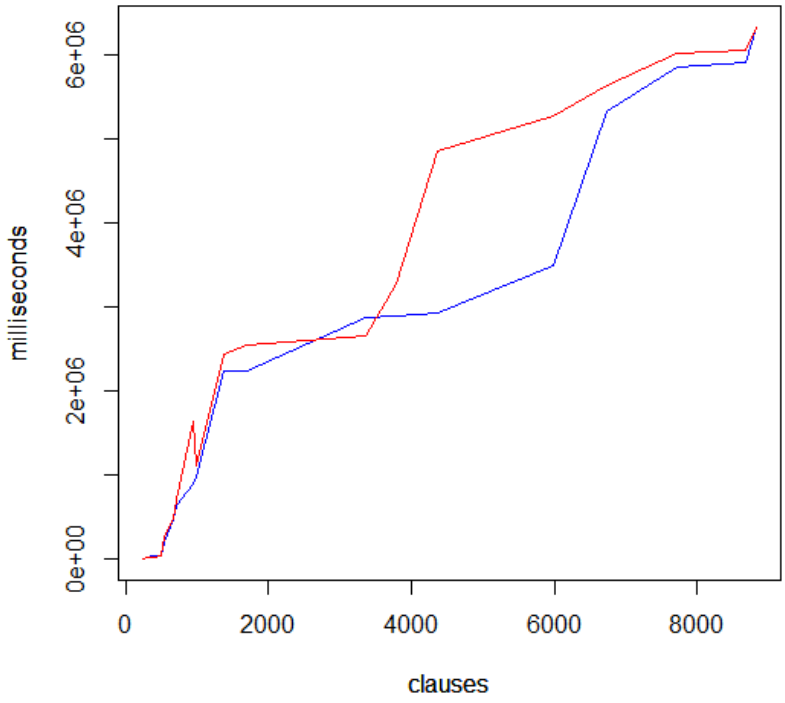

Figure 5.9. Hard type, Execution time vs The number of clauses

6. CONCLUSIONS

Satisfiability problem is one of the oldest proven NP-complete problems. SAT solvers are used by large scale companies such as IBM and Intel in verification of hardware and software tools. In addition to these areas, they are also used in biomedical engineering to verify protein patterns and structures of diseases.

In the literature [5], there are different types of SAT (random, application, hard) and SAT solvers are configured in order to solve a specific type of problem efficiently. In order words, there is no known algorithm to solve all types of benchmarks with the expected speed and accuracy. Therefore, we present our novel approach in solving the SAT using neural network to choose the best solvers and using GPU to solve the sub problems that are generated from the original SAT in parallel.

We have generated our neural network object trained with a set of benchmarks barrowed from SAT Competition. Trained NN object lets us find the best solver in the literature for a given problem. On the other hand, modern GPGPUs have been used in our design for parallelism so that sub problems generated from the original SAT are solved in parallel.

Our solution has several advantages including but not limited to the following:

- We propose our novel decision mechanism with neural network to find the primary and secondary best solvers for a given SAT formula. We achieve 90% success rate in selecting the correct solver.
- We use the computational power of GPU in order to speedup the execution time and reduce the overhead on the CPU of personal computers. We achieve speedups of up to 3 times.
- Our solution solves all types of SAT that have been known to be solved in the literature unlike other solvers that are configured to solve only specific type of problems.

As future work, the following enhancements can be applied:

- Problem division algorithm can be improved to simplify the clause set by unifying and/or removing proper clauses with the aid of certain techniques. It is also important to consider the execution time of this operation in order not to load the system too much overhead for optimization by other measures further decreasing execution time.
- Memory management of CUDA cores can be enhanced to speedup read and write operations of SAT files. Low level memory functions of CUDA can be investigated to enhance register and shared memory usage of threads and blocks.

REFERENCES

1. Wikipedia Encylopedia, "Boolean Satisfiability Problem", *https://en.wikipedia.org/wiki/Boolean_satisfiability_problem*, accessed December 2015.

2. NVIDIA Corporation, "CUDA Parallel Computing Platform", *http://www.nvidia.com/cuda*, accessed December 2015.

3. Wikipedia Encylopedia, "Artificial Neural Network", *http://en.wikipedia.org/wiki/Artificial_neural_network*, accessed December 2015.

4. Davis, M., H. Putnam, G. Logemann, and D. W. Loveland, "Davis Putnam Logemann Loveland Algorithm", *http://en.wikipedia.org/wiki/DPLL_algorithm*, accessed January 2015.

5. SAT Competition, "The International SAT Competitions", *http://satcompetition.org/*, accessed January 2015.

6. JCuda, "Java Bindings for CUDA", *http://www.jcuda.org/*, accessed January 2015.

7. NVIDIA Corporation, "GPU Applications", *http://www.nvidia.com/object/gpu-applications-domain.html*, accessed June 2015.

8. Sen, A., B. Aksanli, M. Bozkurt, and M. Mert, "Parallel Cycle Based Logic Simulation Using Graphics Processing Units", *2010 Ninth International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 71–78, July 2010.

9. Schubert, T., M. Lewis, and B. Becker, "PaMira - A Parallel SAT Solver with Knowledge Sharing", *Microprocessor Test and Verification, Sixth International Workshop*, pp. 29–36, November 2005.

10. Goldberg, E. and Y. Novikov, "BerkMin: A fast and robust Sat-solver", *Discrete Applied Mathematics*, Vol. 15, No. 2, pp. 1549–1561, June 2007.

11. Zhang, H., M. P. Bonacina, and J. Hsiang, "PSATO: a distributed propositional prover and its application to quasigroup problems", *Journal of Symbolic Computation - Special issue on parallel symbolic computation*, Vol. 21, No. 4–6, pp. 543–560, April 1996.

12. Martins, R., V. Manquinho, and I. Lynce, "An overview of parallel SAT solving", *Constraints*, Vol. 17, No. 3, pp. 304–347, July 2012.

13. Chrabakh, W. and R. Wolski, "GridSAT: A Chaff-based Distributed SAT Solver for the Grid", *Supercomputing, 2003 ACM/IEEE Conference*, pp. 1–37, November 2003.

14. Ezick, J. R., S. B. Luckenbill, D. Nguyen, P. Szilagyi, J. Starks, and R. A. Lethin, "Alef parallel SAT solver for HPC hardware", *Proceedings of the 2006 ACM/IEEE conference on Supercomputing Article*, , No. 179, 2006.

15. Asghar, S., E. Aubanel, and D. Bremner, "A Dynamic Moldable Job Scheduling Based Parallel SAT Solver", *Parallel Processing (ICPP), 2013 42nd International Conference*, pp. 67–73, November 2011.

16. Gomes, P., H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability Solvers", Van Harmelen, F., V. Lifschitz, and B. Porter (editors), *Handbook of Knowledge Representation*, pp. 89–134, Elsevier B.V., 2008.

17. NVIDIA Corporation, "GPU Speedups", *http://www.nvidia.com/object/gpu-applications.html*, accessed January 2015.

18. NVIDIA Corporation, "GeForce GTX 560M", *http://www.geforce.com/hardware/notebook-gpus*, accessed January 2015.

19. Joone Neural Network Framework, "Joone, an Object Oriented Neural Engine",

http://sourceforge.net/projects/joone/, accessed January 2015.

20. Balint, A. and U. Schöning, "Choosing Probability Distributions for Stochastic Local Search and the Role of Make versus Break", Cimatti, A. and R. Sebastiani (editors), *Theory and Applications of Satisfiability Testing SAT 2012*, Vol. 7317 of *Lecture Notes in Computer Science*, pp. 16–29, Springer Berlin Heidelberg, 2012.

21. Benhamou, B., T. Nabhani, R. Ostrowski, and M. R. Saidi, "Enhancing Clause Learning by Symmetry in SAT Solvers", *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference*, Vol. 1, pp. 329–335, October 2010.

22. Balint, A., D. Lanti, A. Irfan, and N. Manthey, "CLAS A Parallel SAT Solver that Combines CDCL, Look-Ahead and SLS Search Strategies", *Proceedings of SAT Competition 2014*, Vol. 2, 2014.

23. Balint, A. and N. Manthey, "SparrowToRiss", *Proceedings of SAT Competition 2014*, Vol. 2, 2014.