Analysis of optimization techniques for SAT solvers

Badrinarayanan Nandhakumar
Dept. of Electrical and Computer Engineering
Purdue University, West Lafayette
bnandhak@purdue.edu

Venkatesh Bharadwaj Srinivasan
Dept. of Electrical and Computer Engineering
Purdue University, West Lafayette
sriniv91@purdue.edu

***Abstract*—Satisfiability solvers are software programs to check whether a given boolean expression is satisfiable or not through some combination of inputs. Since a satisfiability problem is NP-Complete, there has been a lot of research in the area of SAT solvers to develop better algorithms for reducing time complexity in practical applications. SAT solvers always have the problem as to which clauses have to be learnt/delete, which variable has to be branched etc. The ubiquity of Machine Learning in recent times has encouraged us to apply its concepts to implement an advanced heuristic to improve existing SAT solver. In this paper, we implement a Machine Learning optimization called 'Stochastic Gradient Descent' on Minisat Solver. We also try to optimize Minisat Solver to run on GPUs for improved speed of computation. Our implementation takes cnf formula in DIMACS format as an input and the output tells us whether the given formula is satisfiable. We also benchmark our code to measure the speed improvements against the baseline Minisat solver implementation.**

***Index Terms*—SAT solver, machine learning, CUDA, GPU**

I. Introduction

Boolean satifiability (SAT) problem is a problem of determining if there exists a combination of inputs, that satifies a given Boolean formula. The tremendous growth in the field of VLSI and rapid increase in the size and complexity of circuits necessitated the advent of SAT solvers to automate the process of determining solutions to Boolean satisfiability problems. Of late, the speed of operation of Boolean Satisfiability (SAT) solvers have started to stagnate. Machine learning has emerged as one of the promising tools to enhance the speed of the current generation of solvers. That being said, the Maple series of SAT solvers is a family of novel conflict-driven clause-learning SAT solvers outfitted with machine learning-based heuristics. A key innovation in the MapleSAT series of SAT solvers is the use of the learning rate branching heuristic (LRB) [1], a departure from the 'Variable State Independent Decaying Sum' (VSIDS) branching heuristic that has been the status quo for the past decade of SAT solving. Here we try to emulate their work in our style using basic Machine learning optimization concept called 'Stochastic Gradient Descent' on the MiniSat solver code that is available.

Liang et. al. [2] gives a detailed depiction of various optimization algorithms used for building the satisfiability solvers. We find that, in order to optimize the learning rate (#conflicts/#decisions), Global Learning rate is used, which serves as a substitute to solve the time of the SAT solver and quality of the branching heuristic is chosen. The rate at which the solver learns the clauses via Conflict-driven clause learning can be improved by maximizing the Learning rate. The greedy GLR branching (GGB) heuristic is proposed to observe branching on variables that maximize GLR greedily. It helps in maximizing GLR and minimizing solving time. But, it has a large overhead/computational complexity as it involves high cost while executing the function. So, the thesis report suggests ways to mitigate this cost.

Costa et. al. [3] presents a general overview of GPU optimizations that can be done on SAT solvers. Most of the code in SAT solver is sequential, hence, trying to run the entire code on GPU will likely provide reduced performance when compared to CPU due to increased latency of operations in Graphics Accelerators. So, certain parts of branching heuristic alone can be sent to the GPU for optimization. In this method however, there is an overhead involved in transferring the data from CPU memory (system memory) to GPU memory (global memory), which might actually lead to increased overall run time of the algorithm. However, if the input size is sufficiently large, the data transfer time can be amortized over all the inputs. Hence the transfer time will become negligible compared to the computation time and we can expect to see performance gain.

II. Data Structures

A. Machine Learning Implementation over MiniSat

We used MiniSat 2.2's data structures, over which, we have implemented Greedy VSIDS and Stochastic Gradient Descent Algorithm in addition to Literal Blocking Distance based Clause Deletion and Rapid Deletion techniques. No data structure was removed from the MiniSat code. The operations within each function in the Solver constructor was optimized. We introduce a parameter called Global Learning Rate (GLR), which can be considered as a proxy for solver time.

$$GLR = \frac{\#conflicts}{\#literals}, \tag{1}$$

We consider this as an objective function, which has to be maximized. In addition, we bring in the total count of Literal Blocking Distance (LBD) which is a metric for learnt clauses that is widely believed to be negatively correlated with the quality of the clause. LBD of a clause is defined as the number of distinct decision levels for each variable in a clause. Clauses with bad Literal Block Distance will be taken out from the database. It is also termed as 'reward'. We assume that one clause is learnt per conflict. These rubrics are incorporated with the already pre-existing data structres.

B. GPU implementation using CUDA C

Minisat is originally implemented in C++. However, we have implemented our code in C to make it easy for CUDA programming. Thus we have used structures to group all the clauses in the input problem and also to encapsulate all literals in a clause. The member functions of C++ classes in Minisat solver have been replaced with C-style functions that operate on one or more clauses as required. The function type qualifiers **__host__** and **__device__** represent functions are callable from and executable on CPU and GPU respectively. Another qualifier **__global__** is used to denote kernel functions that are called from CPU and executable on GPU. Unlike ordinary function calls used in C/C++ language, calls to kernels functions are of the general form - $function\_name <<< dimGrid, dimBlock >>> arguments$, where dimGrid and dimBlock are used to dictate the number of threads to be launched on the GPU

III. Methodology

A. Greedy VSIDS

In order to show the significance of Stochastic Gradient Descent that we will be implementing Greedy Global Learning Rate implementation for which, we will use the already exisiting branching heuristic, VSIDS. Unlike VSIDS, this doesn't take into consideration of the future literals that it might get into conflict with. This is a very computational expensive way of solving the Satisfiability test though. Greedy VSIDS prioritizes decision variables where there is a conflict for branching during the call to Boolean Constrained Propagation. We observe that Greedy VSIDS performs better than VSIDS if you disregard the computational overhead. Hence, to get to a neutral ground with respect to both the computational complexity and reward, we try implementing a Stochastic Gradient Descent based branching heuristic. When considering just branching, the high GLRs help in understanding that the objective function has been maximized.

B. Stochastic Gradient Descent based Branching Heuristic

Based on the guidelines from the thesis report of Liang et. al. [2] in Page 68, we have currently implemented the basic optimization concept called Stochastic Gradient Descent judiciously adhering to the algorithm given there, which is an iterative algorithm where we pick datapoint (batch in our case) randomly from the entire dataset, take the gradient and update them on each run, which has a significant impact in lowering the computations. In ordinary gradient descent, we take the gradient of the entire dataset and update the gradient after each iteration. This is done by modelling our SAT solver problem as a classification problem involving the cost as a partial assignment feature vector as the training data (1 indicates conflict on applying Boolean Constrained Propagation on the partial assignment; 0 indicates non-conflict). To give a brief idea on the mindmap's content (Fig. 1), the data is passed to

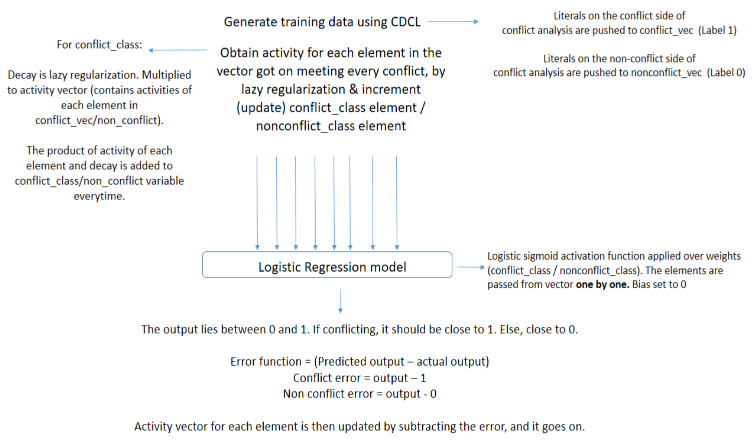
Generate training data using CDCL
Literals on the conflict side of conflict analysis are pushed to conflict_vec (Label 1)
Literals on the non-conflict side of conflict analysis are pushed to nonconflict_vec (Label 0)
For conflict_class:
Obtain activity for each element in the vector got on meeting every conflict, by lazy regularization & increment (update) conflict_class element / nonconflict_class element
Decay is lazy regularization. Multiplied to activity vector (contains activities of each element in conflict_vec/non_conflict).
The product of activity of each element and decay is added to conflict_class/non_conflict variable everytime.
Logistic Regression model
Logistic sigmoid activation function applied over weights (conflict_class / nonconflict_class). The elements are passed from vector one by one. Bias set to 0
The output lies between 0 and 1. If conflicting, it should be close to 1. Else, close to 0.
Error function = (Predicted output – actual output)
Conflict error = output – 1
Non conflict error = output - 0
Activity vector for each element is then updated by subtracting the error, and it goes on.

Fig. 1. Stochastic Gradient Descent - Mindmap

the logistic regression model one by one from both the conflict non-conflict vector. The data is generated in Solver::analyze() section of the code that we submitted. Lazy regularization is done only for unassigned variables.

We used the logistic sigmoid function as the activation function to calculate the weights and update them accordingly as we traverse across the search space. We perform the conflict analysis by passing the BCP and then concatenate the literals on the conflict side by negating the literals in the learnt clause and assign 1 for the partial assignment. We assign 0 to the partial assignment with non-conflict literals, thereby giving us x and y value to be fit as the train model. We make it to branch to the unassigned variable that has the highest weight.

In order to avoid overfitting of the training data, we perform lazy regularization as well in order to update the weights of only the feature vectors which are non-zero. In a way, we design this algorithm to be very similar to VSIDS in the sense that, VSIDS incremented the activities of the variables in the Partial Assignment of the Conflict Class by 1 whereas SGDB increases the weights of the variables in the conflict class of partial assignment by a gradient.

C. GPU implementation using CUDA C

The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the host refers to the CPU and its memory, while the device refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches kernels which are functions executed on the device. These kernels are executed by many GPU threads in parallel. This conceptually works for highly parallel computations/instructions because the GPU can hide memory access latencies with computation instead of avoiding memory access latencies through large data caches and flow control. Hence constructs like for loops with matrix computations (where the data accessed by one thread does not depend on the data accessed by other threads) can be very well optimized on GPUs.

However, it is imperative to note that, a major portion of the code used in SAT solvers is sequential in nature. This leaves us with very few avenues where GPU optimization could be performed. Upon inspection of Minisat code, we discovered that there are a few functions like clause addition and clause simplification where, the same operations are performed on all literals in a given clause. Likewise, there are also operations that are performed on all input clauses in the given input data set. These sections of the code are chosen as candidates to be optimized using GPUs. For instance let us consider the new clause generation code in C.

```
    for (i = 0; i < size; i++)
            c->lits[i] = begin[i];
```

This code reads a list of integers and assigns them to a corresponding list inside a clause c. Assuming the size value is large, this loop can take a lot of time to be executed on a CPU. However due to the independent nature of data access in this code, a kernel function can be written to execute the same statements on a GPU.

```
__global__
void clause_new_kernel(int* begin, int size,
    clause* c) {
    int i = threadIdx.x + blockIdx.x *
        blockDim.x;
    if(i < size)
        c->lits[i] = begin[i];
}
```

Here several threads are launched in parallel and each thread executes the same function. To restrict a thread to use its own data, *int i* is defined this way. Usually the number of threads lauched are in powers of 2. To prevent unwanted threads from storing garbage data in the data structure, and *if* condition is used. To invoke this function on the GPU (device) from the CPU (Host) side, the following statement is used:

```
cudaMalloc((void**) &d_begin,
    size*sizeof(int));
cudaMemcpy(d_begin, begin, size*sizeof(int),
    cudaMemcpyHostToDevice);
clause_new_kernel<<<dimGrid,
    dimBlock>>>(d_begin, size, c);
cudaFree(d_begin);
```

IV. Simulation Results

We have used the .cnf files from SATLIB which has a huge list of benchmarking problems. We have used uf50-01.cnf, uf75-01.cnf, uf100-01.cnf, uf150-01.cnf for Satisfiability test and uf50-01.cnf, uf75-01.cnf, uf100-01.cnf, uf150-01.cnf for Unsatisfiability test for our testing on VSIDS, Greedy VSIDS, Stochastic Gradient Descent algorithm in addition to the CUDA implementation. We had also implemented a simple expression to be tested on the Stochastic gradient descent algorithm.

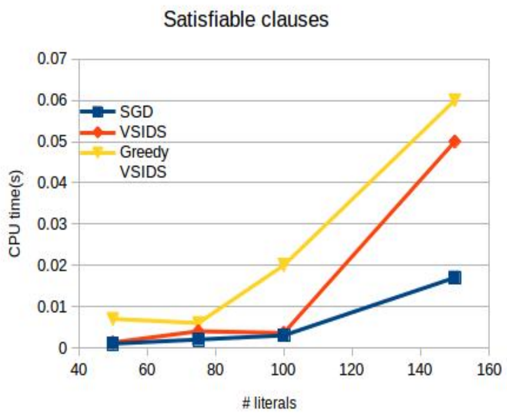

Fig. 2. Literals vs CPU time

A. Comparison of ML branching heuristic with other heuristics

We perform an exhaustive comparative study of various branching heuristics against the Stochastic Gradient Descent based Machine Learning heuristic. The results for Satisfiability and Unsatisfiability clauses for CPU time with respect to the number of literals are shown in the graphs (Fig. 2) and (Fig. 3). We see that the all the files except for the file with 150 literals (an anomaly) have more number of solver instances via Greedy VSIDS branching, as they're computationally more expensive, and as seen in the methodology, they keep branching over every conflict variable.

Hence, the reward is higher for Greedy VSIDS, though the solver time is high. We see from the graphs that greedy VSIDS has the highest solver time (CPU time). We went ahead with a concept called Stochastic Gradient Descent that was assumed to produce the results taking the best of both worlds, and it rightly turned out to be so!

We can also observe that the Stochastic Gradient Descent Branching Heuristic has better rewards when compared to VSIDS via the tables shown (Table I), (Table II) and (Table III).

TABLE I
VSIDS - Performance analysis

Parameters taken	**Number of Literals**			
	50	***75***	***100***	***150***
GLR	0.46	0.389	0.38	0.343
LBD	20	51	811	2869
Solver instances	22	22	201	488

B. CUDA implementation

Table IV compares the average simulation time for input clauses with different number of literals. The total execution

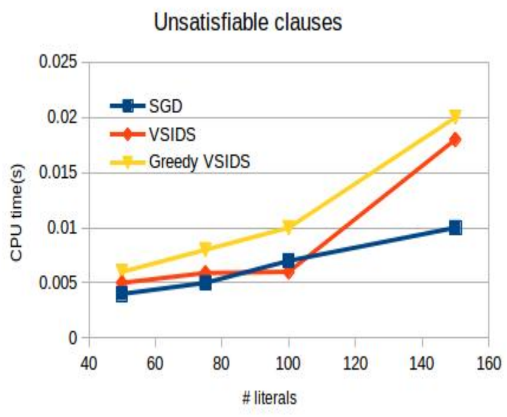

Fig. 3. Literals vs CPU time

TABLE II
GREEDY VSIDS - PERFORMANCE ANALYSIS

Parameters taken	**Number of Literals**			
	50	***75***	***100***	***150***
GLR	0.53	0.48	0.44	0.41
LBD	56	151	1096	8402
Solver instances	24	47	298	1457

time for the GPU is 60-100 times more than the corresponding run time in CPU! Table V shows the detailed profiling information of GPU run times. We can clearly observe that the computation part of the kernel alone takes less than 1% of the total run time. Almost all the time is spent in data copying. This is due to the fact that CPU and GPU have dedicated memory space. The host and device can't access each other's memory locations. This necessitates copying data from host to device and vice versa for each and every kernel execution. The memory transfer time also includes the time required by the GPU to fetch data from global memory for its computation. GPUs unlike CPUs don't have multi level caches to store data for faster accesses. This architecture is required in order to maximize the number of computation units. So every data access has to come from the global data memory.

Since the functions optimized for CUDA operate on all the literals of a clause at a given time, it would be beneficial in scenarios where the number of literals per clause is comparable to the number of clauses itself. All the synthetic inputs tested on our implementation have only 3 literals per clause, making it a 3-SAT problem. In this case, only 3 threads of the GPU will be active during a kernel execution. Hence the device is terribly underutilized to provide any tangible boost to program's execution time. Another point to note is that, the data transfer time will be amortized if the quantity of data to be moved from host-to-device and device-to-host is large, as the system internally coalesces spatially local data to complete the transfer in a single go. Another way in which the data transfer time can be hidden is by having large number of computations/instruction executions per thread. However this is also not the case in SAT solvers. The baseline code is predominantly sequential with very few avenues for parallelization. Thus, the negatives associated with GPU computations far outweigh the throughput advantage that they provide and we conclude that it is not a good idea to try to parallelize SAT solvers for implementation on GPUs (especially when the no. of literals per clause is very low). These findings are in line with the data presented in [3].

TABLE III
SGD - PERFORMANCE ANALYSIS

Parameters taken	**Number of Literals**			
	50	***75***	***100***	***150***
GLR	0.54	0.42	0.40	0.35
LBD	17	75	1031	14858
Solver instances	20	28	263	2627

TABLE IV
BASELINE MINISAT VS GPU MINISAT RUN TIMES

# Literals	**CPU Exec. time (ms)**	**GPU Overall Exec. time (ms)**
50	1	110
75	2	120
100	3	250
150	17	300

TABLE V
SAT SOLVER EXECUTION TIME WITH GPU KERNELS

No. of Literals	**GPU Exec. Time splitup (in ms)**	
	Kernel exec. time	***CudaMalloc Memcpy***
50	0.138	49.86
75	0.213	119.7
100	0.327	248.7
150	0.414	299.5

V. FUTURE WORK

As an extension for the current work, we intend to extend the current work by parallelizing the code of SAT Solver to run on Multiple CPU cores, instead of making it run on a GPU as discussed by [4]. This technique has the advantage of avoiding the data copying overhead that is present in our current implementation. In addition, with respect to the Machine Learning implementation, we wish to look into ways to learn the clauses via a Recurrent Neural Network, in which, the clauses to be learnt and the variables to be branched can be considered as learnable parameters.

ACKNOWLEDGEMENT

The authors would like to thank Prof. Anand Raghunathan for giving his valuable inputs to enhance the information presented during the presentation of ECE595Z project on May 8, 2020.

References

[1] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for sat solvers," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2016, pp. 123–140.

[2] J. H. Liang, "Machine learning for sat solvers," Ph.D. dissertation, University of Waterloo, Dec. 2018.

[3] C. Costa, "Parallelization of sat algorithms on gpus," Technical report, INESC-ID, Technical University of Lisbon, Tech. Rep., 2013.

[4] G. Chu, P. J. Stuckey, and A. Harwood, "Pminisat: a parallelization of minisat 2.0," *SAT race*, 2008.