Solving SAT and MAX-SAT on GPUs

Pranav Maneriker , Arnab Ghosh , Subhajit Roy
mpranav@iitk.ac.in , arnabgho@iitk.ac.in , subhajit@iitk.ac.in
Computer Science And Engineering
Indian Institute of Technology, Kanpur

Abstract

In this project we first looked at the satisfiability problem (SAT). In particular, we focused on approaches to parallelize SAT solving to enable us to implement a GPU based algorithm for SAT and MAXSAT. We also looked at how we can best combine existing GPU subroutines for the implementation. We came up with an algorithm quite different from the existing DPLL based heuristics for parallel implementation of a SAT solver. The algorithm retains all satisfying assignments and eliminates portions of the search space that do not have any solution and hence can be easily extended to the MAX-SAT problem. We first tested an iterative sequential version of the algorithm and then implemented it on a GPU. implementation.

1 Introduction

1.1 SAT

" The Boolean Satisfiability problem was the first problem identified as NP-Complete . Some factors still make it one of the most important problems in Computer Science . Those factors are its simplicity, its numerous applications in real life problems such as software testing with model checkers , theorem proving and Electronic Design Automation .But perhaps the most important factor is the ability to reduce any NP-Complete problem into SAT.

The Davis Putnam Logemann Loveland (DPLL) algorithm introduced in 1962 [Davis *et al.*, 1962] solves the problem by recursively traversing the entire search space with several improvements over a more basic brute-force algorithm finishing only when a solution is found or when the algorithm has exhaustively checked every possibility and found no solution. This algorithm forms the basis of modern SAT Solvers , however in recent times the global performance of state of the art SAT solvers has been stalling and only minor optimizations have been introduced to improve them despite the efforts of the whole community .

Moreover CPUs are hitting their physical limits in terms of a single core speed and to continue to improve in speed SAT solvers had to keep up with the hardware and go parallel. The initial method to parallelize SAT Solving was to split the search space between the cores/threads, and, with load balancing techniques re-split the space every time a core/thread finished its search . " Our algorithm tries to load balance by assigning the partitions to the threads so that roughly all the threads have the same overload in terms of the job to be done .

1.2 GPU

" The GPU is especially well-suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology [NVidia,]

Highly parallelizable algorithms such as the Neural Networks Backpropagation led to huge resurgence back into Parallel Programming into GPUs and led to scalable systems capable of computing huge amounts of data in parallel .

Several algorithms started using the GPU based parallel programming architecture to leverage the power of the GPUs for enormous parallel computation and several libraries have come up which leverage the power of the GPUs "

2 Overview of the Algorithm

In this section, we give an overview of the algorithm and the differences from other work done for solving SAT. We also describe how this algorithm can be adapted for MAXSAT solving.

2.1 Intuition

The most popular algorithms for solving SAT have been DPLL [Davis *et al.*, 1962] based approaches. Such approaches try to find a single satisfying assignment for the clauses. This requires branching of parallel threads based on propagated assignment, which may not be well suited for NVIDIA CUDA model. This is because divergent branches increase the time taken for a fixed number of iterations.[Harris and others, 2007].

In each iteration of the algorithm each partition looks at its clauses and generates the conflict clauses that push the search to the space of solutions.Our approach focuses on retaining all satisfying assignments and eliminating portions of the search space that do not have any solution. We remove variables that are satisfied trivially at each stage and keep our constraints by means of the conflicts clauses arising from assignments. We keep the clauses entirely, unlike unit propagation, where one possible value of a clause is being tested each time when the assignment to a literal is not implied by another. Thus there is no *branching* in our algorithm. This has two implications:

- The CUDA parallel paradigm is well suited for our algorithm
- Since no possible solution is lost, we can use adapt this algorithm to solve MAX-SAT easily. The exact implementation of this idea is described later.

2.2 Algorithm

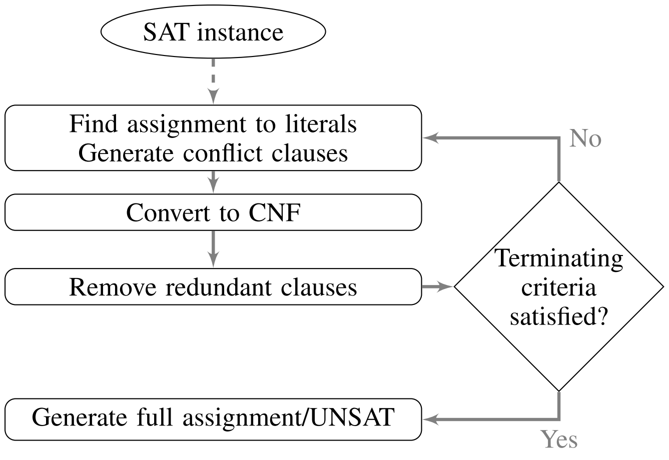
SAT instance
Find assignment to literals
Generate conflict clauses
Convert to CNF
Remove redundant clauses
Terminating criteria satisfied?
No
Yes
Generate full assignment/UNSAT

Figure 1: Summary of SAT algorithm

The parallelization of the algorithm depends on partitioning the clauses so that the different partitions can work independently and we can combine the solutions to get an overall solution.

The basic steps in the algorithm are:

- **Generate conflict clauses for each partition:** We need to first check if any partition has a trivial assignment, that is, all clauses containing a particular literal x_i contains only the positive literal or only the negative literal. This means that we can assign either T or F to this variable accordingly. If we have both x_i and $\neg x_i$ then we have to satisfy either x_i and the clauses containing $\neg x_i$ or vice versa. Thus, if C is the set of all clauses, and $Cp_i = \{C_i \in C | x_i \in C_i\}$ and $Cn_i = \{C_i \in C | \neg x_i \in C_i\}$ then we have

  $$(x_i \Rightarrow Cn_i) \wedge (\neg x_i \Rightarrow Cp_i)$$

  Which can be simply learnt as

  $$Cp_i \vee Cn_i \tag{1}$$

  And we can infer the assignment of xi from the set of satisfied clauses after the algorithm is complete.

- **Simplify to CNF:** Equation (1) is in DNF form and to continue our algorithm, we need to convert it to CNF form. In order to do this, we use the following identity:

  $$(c_1 \wedge c_2 \wedge c_3 \ldots c_k) \vee (c'_1 \wedge c'_2 \wedge c'_3 \ldots c'_{k'}) = (c_1 \vee c'_1) \wedge (c_1 \vee c'_2) \ldots (c_1 \vee c'_{k'}) \wedge (c_2 \vee c'_1) \wedge (c_2 \vee c'_2) \ldots (c_k \vee c'_{k'}) \tag{2}$$

  After this step, we may have a large number of redundant clauses (either due to an assignment to some literal or just repeated clauses)

- **Cleanup clauses:** In this step, we cleanup the clauses by eliminating repetitions among clauses followed by filtering of clauses that are still unsatisfied by the literal assignments so far. These two steps are executed using GPU primitives from standard CUDA Thrust library[2010].

We repeat these steps until there are no new clauses generated (we hit a fixed point) or until we reach UNSAT state (generating an empty clause at any stage) .

The algorithm can run entirely on the GPU once started. There is no need to have frequent memory copies, which allows significant speedup.

2.3 Extension to MAX-SAT

To adapt the algorithm for MAX-SAT, we introduce new literals to each clause without adding partitions).

So for each clause $c_i \in C$, we write the clause as $c_i \vee \lambda_i$

While solving this new instance of SAT, we don't introduce any partitions for the $\lambda'_i s$. We continue along the steps of the previous algorithm.

If we get a partition containing only $\lambda'_i s$, we ignore it until convergence.

At the point of convergence, we have a set of partitions containing clauses consisting of $\lambda'_i s$.

Since any remaining $\lambda'_i s$ would correspond to unsatisfied partitions.

So to achieve MAX-SAT, we would have to solve the following linear programming problem:

$$\min \sum_i \lambda_i \tag{3}$$

subject to:

$$\sum_c \lambda_{c_i} > 0 \tag{4}$$

Where c are the clauses only containing $\lambda's$ and $\lambda_i \in \{0,1\}$

The problem thus gets reduced to a integer linear programming problem which when we solve gives us the MAX-SAT solution.

2.4 An example

The Algorithm always keeps all the solutions intact in the search space and hence can be used for both SAT and MAX-SAT.

An Example of the algorithm will make it clearer
Let's say we have the following set of clauses

$\bar{x_3} \vee x_1$
$x_1 \vee x_2$
$x_2 \vee x_3 \vee \bar{x_1}$
$\bar{x_2} \vee x_3$

Partitions :

x_1	x_2	x_3
$x_1 \vee x_2$	$x_2 \vee x_1$	$\bar{x_3} \vee x_1$
$x_1 \vee \bar{x_3}$	$x_2 \vee x_3 \vee \bar{x_1}$	$x_3 \vee x_2 \vee \bar{x_1}$
$\bar{x_1} \vee x_2 \vee x_3$	$\bar{x_2} \vee x_3$	$x_3 \vee \bar{x_2}$

Conflict Clauses:

x_1	x_2	x_3
$x_2 \vee x_2 \vee x_3$	$x_3 \vee x_1$	$x_1 \vee x_2 \vee \bar{x_1}$
$\bar{x_3} \vee x_2 \vee x_3$	$x_3 \vee x_3 \vee \bar{x_1}$	$x_1 \vee \bar{x_2}$

Cleanup :

x_1	x_2	x_3
$x_2 \vee x_3$	$x_3 \vee x_1$	x_2
x_2	$x_3 \vee \bar{x_1}$	$x_1 \vee \bar{x_2}$

New Set of clauses for iteration 2 :

$x_2 \vee x_3$
x_2
$x_1 \vee x_3$
$\bar{x_1} \vee x_3$
$x_1 \vee \bar{x_2}$

Partitions:

x_1	x_2	x_3
$x_1 \vee \bar{x_2}$	x_2	$x_3 \vee x_2$
$x_1 \vee x_3$	$x_2 \vee x_3$	$x_3 \vee x_1$
$\bar{x_1} \vee x_3$		$x_3 \vee \bar{x_1}$

Conflict Clauses:

x_1	x_2	x_3
$x_3 \vee \bar{x_2}$	$x_2 = 1$	$x_3 = 1$
$x_3 \vee x_3$		

Cleanup Leads to 1 since $x_3 = 1$ hence the set of clauses is satisfiable

To get the assignment for x_1 we use the original set of clauses and do one step of the algorithm with the resulting set of clauses

We use the assignment $x_3 = 1$ and $x_2 = 1$ on the original set of clauses to get the clause :

x_1

Then another run of the algorithm gives the assignment $x_1 = 1$

2.5 For SAT

The algorithm proceeds iteratively in primarily having 3 steps in each iteration and each of the steps have been implemented using a Kernel . These iterations continue until a fixed point is reached and then the original clauses are brought back and the current assignments are used to simplify the original clauses and then run a single iteration of the algorithm.

Procedure Partition
The main tasks of this procedure are :

- Analyze the clauses in each partition and find whether the partition can be trivially satisfied i.e. all clauses in the partition have the same form of the variable associated with the partition i.e. only x or only $\bar{x}$
- If trivial assignment is not possible then write the relevant clauses associated with x and $\bar{x}$ in a relevant data structure for the expand kernel to generate the conflict clauses

Procedure Expand Clauses
The main tasks of this procedure are :

- Read from the data structure written by the Partition Procedure to generate the conflict clauses for the Cleanup Procedure to work.
- The technique it uses to expand the clauses is :
  if clauses are of the form
  $x \vee pos_1 \ldots x \vee pos_{npos}$
  $\bar{x} \vee neg_1 \ldots \bar{x} \vee neg_{nneg}$
  Then the npos*nneg conflict clauses generated are : $pos_1 \vee neg_1 \ldots pos_1 \vee neg_{nneg}$
  $\vdots$
  $pos_{npos} \vee neg_1 \ldots pos_{npos} \vee neg_{nneg}$

Procedure Cleanup Clauses
The main tasks of this procedure are :

- Iterate over the generated conflict clauses and reduce the clauses by assigning the values to the already resolved variables
- Reduce the trivial inconsistencies like $x \vee \bar{x} = 1$
- If a clause becomes satisfied by the current assignment then its no longer a conflict clause since it is already satisfied so remove the clause.
- If there's a repetition of the clauses then only one copy is used for the next iteration .
- If any clause becomes null because of the assignment then the original set clauses is UNSAT

3 Detailed Information

Various Kernels Used

3.1 Partition Kernel

Each Thread deals with a particular partition's partition function . Each thread first checks if the partition has a variable only in a single form , if it has only a single form then that partition can be trivially satisfied by putting the assignment as the current variable's form , it updates a write flag telling whether the particular partition has to be expanded .

`d_prod_clauses[var][0]` stores the indices of the clauses occurring in the positive format and `d_prod_clauses[var][1]` stores the indices of the clauses occurring in in the negative form .

`varSearch` is a binary search on the GPU and takes the CSR format and checks whether a variable is present in a particular clause .

`d_conflict_lenP` & `d_conflict_lenN` stores the number of clauses in the positive and negative forms respectively for the prescan on `d_conflict_tot` to work correctly and for the expand kernel to write without any contention on the memory location that it is accessing .

3.2 Expand Kernel

Each Thread deals with a particular partition's expansion , the thread recognizes the variable from the Thread Id and then reads the data structures written by the partition kernel namely `d_prod_clauses` and then writes back the expanded clauses in the `d_expanded_clauses_buffer`. The original clauses are in the CSR format but the result of this kernel is in the Dense Matrix Format for the cleanup kernel to work fast on it.

The expansion step is the bottleneck step of our algorithm and to improve the performance several threads can work simultaneously on different parts of the result i.e the `d_expanded_clauses_buffer` and lead to better performance .

3.3 Cleanup Kernel

The Cleanup Kernel uses the assignments obtained so far from the partition kernel and then assigns the assignments and simplifies all clauses . After simplification of clauses it also removes redundancies among the clauses by using A thrust primitive : `thrust::unique` which takes a particular clause and then a custom comparator to check equality , here equality has to be checked among the rows and all the columns of the 2 row .

After the clauses have been compacted the `thrust::copy_if` primitive is used which copies only if a clause is meaningful in the next step i.e. not trivially satisfied by the current set of assignments .

4 Conclusion

Our algorithm provides a method to preserve all the solutions of the given SAT instance and is inherently parallelizable and provides good results if the average number of clauses in which a particular variable occurs is quite few .

Several heuristics might come into the picture for the load balancing among the threads by selecting the variables for a particular thread so that all the threads in a particular block get enough work and almost equal work for the algorithm to fully leverage the capability of the GPUs .

Future research in extending and bettering the algorithm's performance might be to allow different threads working on different parts of the expand kernel matrix since that is the main bottleneck computation in our algorithm .

Acknowledgments

This work was possible because of the GPU (subhajit1.cse.iitk.ac.in) and the Undergraduate Project at the Indian Institute of Technology , Kanpur .We thank Professor Subhajit Roy for guiding us throughout the semester and granting us access to the GPU for our experiments . We are also thankful to the Department of Computer Science and Engineering ,IIT Kanpur for providing us cloud based services for some of our experiments(lxc14.cse.iitk.ac.in).

References

[Davis *et al.*, 1962] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[Harris and others, 2007] Mark Harris et al. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2(4), 2007.

[Hoberock and Bell, 2010] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.7.0.

[NVidia,] NVidia. Cuda c programming guide.