GpuShareSat: a portfolio SAT solver using the GPU for clause sharing

Nicolas Prevot
nicolas.prevt@gmail.com

Abstract

We describe a novel SAT solver using both the GPU (CUDA) and the CPU. The CPU runs a classic multithreaded portfolio CDCL SAT solver. Each CPU thread exports all the clauses it learns to the GPU. The GPU makes a heavy usage of bitwise operations. It notices when a clause would have been useful for a CPU thread and notifies that thread, in which case it imports that clause. This relies on the GPU repeatedly checking millions of clauses against hundreds of assignments. All the clauses are checked independantly from each other (which allows the GPU massively parallel approach), but against all the assignments at once, using bitwise operations. This allows CPU threads to only import clauses which would have been useful for them.

Our solver is based upon glucose-syrup. Experiments show that this leads to a strong performance improvement, with 22 more instances solved on the SAT 2020 competition than glucose-syrup. The source code is available at https://github.com/nicolasprevot/GpuShareSat

1 Introduction

Boolean satisfiability (SAT) is a fundamental problem in computer science. Despite being NP complete, modern SAT solvers are able to handle large instances with millions of variables. It is widely used for a variety of application, which includes theorem proving and hardware verification.

Graphical Computing Unit (GPU) are highly parallel devices, and are able to perform many times more operations per second than the CPU. CUDA, provided by NVIDIA, is a popular programmable platform to perform general purpose computation on the GPU. However, transposing SAT algorithms from the CPU to the GPU is not an easy task.

A problem for portfolio Parallel SAT solvers is to decide which clauses to share between threads and which ones not to. We present a novel algorithm which relies on the GPU via CUDA to tackle this problem. Our solver builds upon glucose-syrup 4.1, a parallel portfolio CPU SAT solver which is itself based upon MiniSat. The code for the CDCL procedure of CPU solver threads is mostly kept. The code to exchange clauses between them is not.

2 Related work

2.1 Sat solving and the GPU

Modern SAT solvers algorithm (like the DPLL procedure) which are effective on the CPU do not translate easily to the GPU. One difficulty is that alghough the GPU can run many times more threads at once than the CPU, the amount of memory and cache available per thread is much smaller on the GPU. Running the DPLL procedure with the watched literal scheme requires a large amount of memory, especially on large formulas. This would make running the DPLL procedure separately in each GPU thread challenging. Alternatively, using several GPU

threads to do propagations for a single DPLL procedure would requires heavy communication between several GPU threads, which is also difficult. Various approaches have been proposed that use the GPU to solve SAT but they have not become widespread. In [7], the GPU is used to do unit propagation. In [5], the GPU is used for pre-processing the formula (variable elimination, subsumption)... In [6], the GPU is used for the survey propagation algorithm.

However, most of the modern SAT solvers rely exclusively on the CPU.

2.2 CPU parallelism with portfolio solvers

In a parallel SAT solver: on the CPU, each thread runs the DPLL procedure separately from the others and learns clauses. In the portfolio approach, threads each use different parameters from each other. For threads to benefit from each other's work, they share some clauses with each other. One difficulty with this approach is to decide which clauses should be shared with the other threads and which should not. Sharing too many clauses would increase the number of clauses that each thread has, which slows down the search. However, if not enough clauses are shared, each thread does not benefit for the work of other threads. So a heuristic is used to identify which clause is good and should be shared. In [3], clauses are shared only if their size (or lbd) is smaller than a constant. [2] uses a metric based on the community structure. In [1], a thread will only share a clause if it has been useful another time. In [4], a thread will only export a clause to another thread if they are connected by an edge in a graph.

3 Checking which clauses trigger with bitwise operations

3.1 Preliminaries

We define a truth value as a member of the set {T, F, U} where T stands for True, F for False, and U for Undef
The negation of a truth value w is denoted as $\neg w$, with $\neg T = F$, $\neg F = T$, $\neg U = U$
An assignment A is a function from a set of variables to a truth value
A literal l is a variable v or its negation $\neg v$
Given an assignment A and a literal l, $A(l) = A(v)$ if $l = v$ and $A(l) = \neg A(v)$ if $l = \neg v$
A clause C of size s is a disjunction of n literals: $C_1 \vee C_2 ... \vee C_s$
A formula is a conjunction of clauses

Definition 1. *A clause C of size s triggers on an assignment A if:*

- *For each literal l in C,* $A(l) \neq T$
- *For at least s - 1 literals l in C,* $A(l) = F$

A clause which triggers on an assignment is either in conflict, or implying one literal

3.2 Checking which clauses trigger

As described in [9], we can use bitwise operations to check if a clause triggers on multiple assignments. Given N assignment A_i for $1 \leq i \leq N$ and a variable v: we can effectively represent the values $A_i(v)$ using only two N-bits variables: (isTrue, isSet).
The i-th bit is isTrue represents whether $A_i(v) = T$
The i-th bit is isSet represents whether $A_i(v) \neq U$

Given a literal $l = \neg v$, the values $A_i(l)$ can be computed with $(\sim isTrue, isSet)$ This allows us to check if a clause C of size s triggers over N assignments at once using bitwise operations:

```
assignmentTriggers(B bitwise assignments, C clause)
    allFalse <- ~0
    oneUndef <- 0
    for l in C:
        oneUndef = (allFalse & ~B.isSet(l)) | (oneUndef & B.isFalse(l))
        allFalse &= B.isFalse(l)
    return allFalse | oneUndef
```

The previous algorithm returns an N bits variables, where the bit i is set if the clause C triggers on the assignment i

3.3 GPU implementation

Given a set of up to 32 assignments, and a set of clauses (whose number can reach several millions): We test in parallel if each of the clauses triggers on these assignments (i.e. is in conflict or is implying something currently undef). This approach fits very well with the GPU since it is massively parallisable. To make reading the clauses on the GPU faster, We also coalesce reads to the clauses by reordering how we represent them in memory.

4 Usage in a portfolio SAT solver

Like in traditional portfolio parallel SAT solvers, GpuShareSat relies on having several CPU threads, each one running the CDCL procedure in a separate solver.

In GpuShareSat, each CPU solver thread exports all clauses it learns to the GPU (not directly to other CPU threads). Whenever a CPU thread reaches a conflict, it sends the assignment of the parent of the conflict to the GPU. During a GPU run, the GPU checks all of the clauses it has against all the assignments sent by the CPU solver threads it has not checked yet. It reports all the clauses that trigger. Unfortunately, it is possible that the GPU is not be able to cope with all the assignments coming from the CPU. This is because the GPU can only check a finite number of assignments during a run. In this case, some of these assignments will not be sent to the GPU. The CPU threads imports the clauses reported to them. This required a modification of the code which imports clauses: rather than only importing clauses at level 0, we needed CPU solver threads to be able to import clauses at any time, and keep the watched literal scheme consistent:

- If at least 2 literals of the imported clause are undef or true, the clause can directly be attached.
- If all literals are false except for one which is undef: we take the highest level of the false literals, backtrack to that level, and imply the undef literal there.
- If all literals are false except for one which is true: we take the highest level of the false literals, call it l. If the level of the true literal is strictly higher than l, we backtrack to l and imply it there. Otherwise, we do nothing
- If all literals are false: we take the highest level of them, backtrack to that level, and do conflict analysis there.

Clauses that are reported would have triggered on a previous assignment, so they would have been useful to have. They would either have led to a conflict or would have implied a variable that was not set on that assignment.

The duration of a GPU run is usually in the order of a few milliseconds.

4.1 Why it is effective

In a state-of-the-art portfolio SAT solver, a thread may decide to export a clause it learns to other CPU threads. As discussed previously, the difficulty with this approach is to decide which clauses to export. Our approach allows a thread to only attach a clause (coming from the GPU) if it would have been useful in the past few milliseconds. This has the following benefits:

- Clauses that would have been useful are probably better than those that wouldn't, so threads tend to attach to good rather than bad clauses
- If a clause would have been useful recently, it is related to the current search space, so it is more likely to be useful soon
- This clause may still be useful at the time the CPU thread imports it. Unfortunately, in practice, this happens for less than 1% of reported clauses
- If a CPU thread has deleted a clause as part of its clause deletion policy, the GPU may later notify it that this clause would have been useful, in which case it will re-attach the clause

The GPU suffers from the same problem as the CPU solvers in that it also needs to delete clauses. Otherwise, it would become slow and run out of memory. To do that, we perform activity based clause deletion, in a way similar to MiniSat on the CPU. In our case, the activity of a clause is bumped whenever it triggers.

5 Use of aggregate operations to boost performance

5.1 Commonalities between successive runs

We wanted to know how different the values of variables in successive assignments coming from the same CPU thread are. So we devised an algorithm based upon single threaded glucose. In between 32 conflicts, for each variable, we computes all the values taken. We only consider the values of variables after backtracking, or in assignments where unit propagation has completed without a conflict. This gives us a subset of $\{T, F, U\}$ i.e. True, False, Undef. We then compute the ratio of the number of times each such subset happens over the entire run.

We ran it over 10 instances taken at random from the SAT 2020 competition. The average results follow:

subset	ratio
{T}	0.127
{F}	0.064
{T, F}	0.000
{U}	0.660
{T, U}	0.060
{F, U}	0.068
{T, F, U}	0.021

We see that successive assignments coming from the same CPU thread have a good amount in common. This is useful when checking a clause againsts these successive assignments at once. If a clause C has the literals l1 l2 and the values taken for them does not include F: then we are sure that the clause will not trigger for any of these successive assignments. This is because for a clause to trigger, it needs all literal values except for one to be F Alternatively, if there is a single literal among the clause for which the only value taken is T, then again, the clause will not trigger for any of these successive assignments.

So, given 32 assignments: by only looking at these three bits for a given variable: T, F, U, we are, in most cases, able to tell that the clause does not trigger. The remaining of this section will formalise this and use it to check up to 32 times 32 assignments at once.

5.2 Definitions and theorem

Definition 2. *An aggregate g is a subset of truth values ie* $\{T, F, U\}$

Definition 3. *Given an aggregate g, we denote by* $\neg g$ *the aggregate* $\{\neg w\}$ *for* $w \in g$

An aggregate g can be efficiently represented using only three booleans: $(T \in g, F \in g, U \in g)$

Definition 4. *Given a set of variables, An aggregate assignment G is a function which map a variable to an aggregate*

Definition 5. *Given an aggregate assignment G and a literal l, we define* $G(l)$ *as* $G(v)$ *if* $l = v$ *and* $\neg G(v)$ *if* $l = \neg v$

Definition 6. *Given the assignments* $(A_i)_{1 \leq i \leq N}$ *, their associated aggregate assignment G is defined by* $G(v) = \{A_i(v)\}_{1 \leq N}$.

As seen previously, N truth values can be represented by the two N-bits variables: isTrue, isSet
Their associated aggregate can be computed using only bitwise operations by: $(isTrue \neq 0, (isSet \& \sim isTrue) \neq 0, (\sim isSet) \neq 0)$

Proposition 1. *Given the assignments* $(A_i)_{1 \leq i \leq N}$*, G their associated aggregate assignment, and a literal l:* $G(l) = \{A_i(l)\}$

Demononstration: This is true by definition if l = v
If $l = \neg v$, then:
$G(l) = \neg G(v) = \neg\{A_i(v)\}_{1 \leq i \leq N} = \{\neg A_i(v)\}_{1 \leq i \leq N} = \{A_i(l)\}_{1 \leq i \leq N}$

Definition 7. *A clause C of size s triggers on an aggregate assignment G if:*

- *for every literal l in C,* $F \in G(l)$ *or* $U \in G(l)$
- *for at least s - 1 literals l in C,* $F \in G(l)$

Theorem 1. *Given a set of assignments and their associated aggregate assignment: if a clause triggers on an assignment, it triggers on the aggregate assignment*

Demonstration: Given the assignments $(A_i)_{1 \leq i \leq N}$ and their associated aggregate assignment G: this is obvious using the definitions of a clause triggering and the fact that $A_i(l) \in G(l)$

By contraposition of this theorem: if a clause does not trigger on an aggregate assignment, it does not trigger on any individual assignment.

5.3 Usage in GpuShareSat

We previously proposed an algorithm which returned whether a clause triggers over any of N assignments (with $N \leq 32$). We are going to propose a new algorithm which returns whether a clause triggers on a much larger number of assignments

This matters because when the number of CPU solver thread increases, it is less likely that the GPU is going to be able to keep up with the assignments coming from them.

As seen previously, successive assignments coming from a single CPU solver thread have similarities with each other. That is, frequently, their value for a given variable is the same.

For this reason, we are going to group the assignments sent to the GPU by the CPU solver thread they come from. We are going to create M assignment groups. For example, M = 32. If there are M CPU solver threads, there will be one assignment group for each CPU solver thread. If the number of CPU solver threads is lower, there will be several assignment groups for each CPU solver thread.

For each assignment group, we are going to compute the aggregate of its assignments. This gives us M aggregates assignments.

Each aggregate assignment G_i gives has three booleans for each variable: $(t_i(v), f_i(v), u_i(v))$ We can represent the M aggregate assignments using three M-bit variables:

$$(canBeTrue(v), canBeFalse(v), canBeUndef(v))$$

Where the bit i of $canBeTrue(v)$ represents $t_i(v)$ and respectively for $canBeFalse(v)$ and $canBeUndef(f)$

The following algorithm returns on which aggregate assignments a clause C triggers:

```
aggregateTriggers(G bitwise aggregate assignments, C clause)
    allFalse <- ~0
    oneUndef <- 0
    for l in C:
        oneUndef = (allFalse & G.canBeUndef(l)) | (oneUndef & G.canBeFalse(l))
        allFalse &= canBeFalse(l)
    return allFalse | oneUndef
```

It returns an M bit variable whose bit i is set if C triggers on the aggregate assignment G_i

As previously demonstrated, if a clause does not trigger on an aggregate assignment, it does not trigger on any individual assignment. Therefore, we can run the algorithm above first, and only check if the clause triggers on an assignment group if it did not trigger on their aggregate assignment.

```
multiTriggers(G bitwise aggregate assignments, A assignments, C clause)
    agTriggers = aggregateTriggers(G, C)
    for bit i set in agTriggers:
        assigTrigger = assignmentTriggers(A[i], C)
        if assigTriggers != 0:
            report(i, assigTriggers)
```

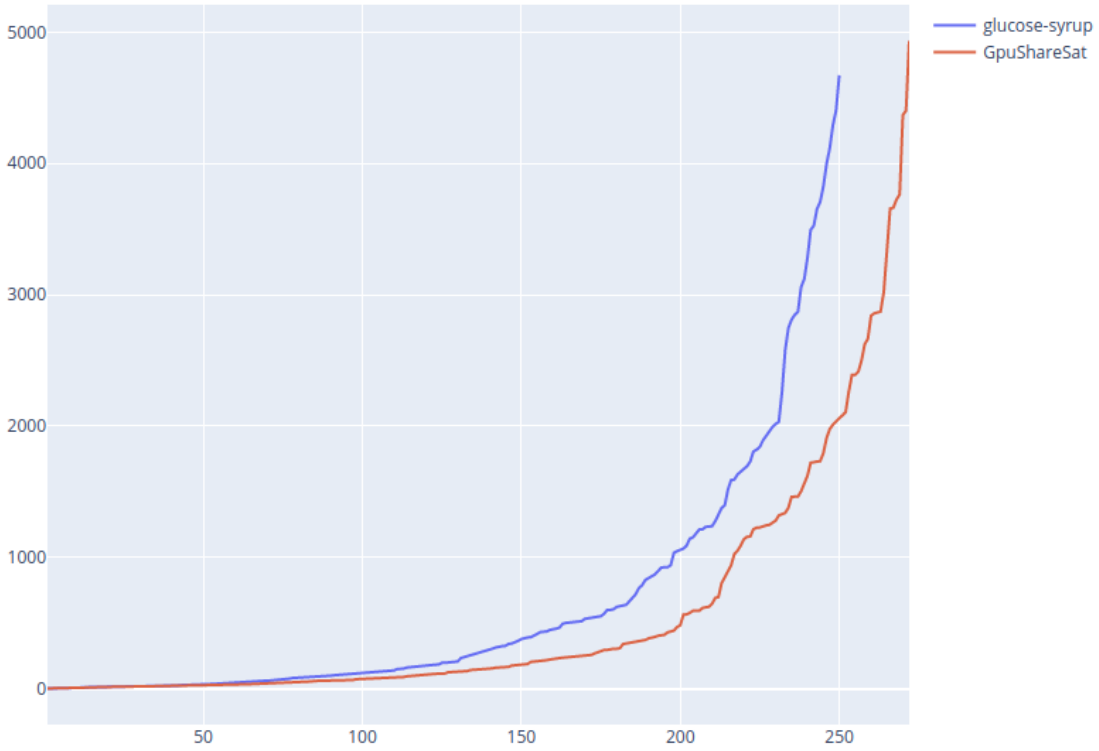
The previous algorithm calls report(i, assigTriggers) whenever C triggers on some assignments of assignment group i. The bit j of assigTriggers is set if C triggers on the assignment j of assignment group i. We will then find the CPU solver thread which owns the assignment group i, and report this clause to it.

6 Results

We compared GpuShareSat(GPU and CPU) to glucose-syrup 4.1(CPU only) on the 2020 SAT competition main track. For the CPU hardware, we used a i9-9960x processor, 3.1 Ghz, with a processor cache of 22MB, 16 cores, able to run 32 threads simulateously. 64 GB RAM. For the GPU, an NVIDIA Quadro P6000 with 24GB memory was used. 32 CPU solver threads were used in glucose-syrup, and 31 in GpuShareSat (since there is another thread to manage the interaction with the GPU)

	glucose-syrup	GpuShareSat
Total solved	250	272
Sat solved	123	138
Unsat solved	127	134
Score	1659500	1432316

Figure 1: Solved instances by running time



For each instance of the SAT 2020 competition main track, we computed the number of clauses checked by the GPU per second (for all CPU solver threads). The average value (weighted by the time spent on this instance) was 594 millions per second.

Similarly, we computed the ratio of assignments that CPU solver threads were unable to send to the GPU, due to it being already full. The average result was 0.103

We also computed the effectiveness of the aggregate check as: the ratio of the number of times when we didn't need to check a clause on an assignment group over the number of

assignment groups sent to the GPU. The result was 0.999422. As a reminder, for a GPU run, the assignments sent by a CPU solver thread will be divided into one or more assignment groups. This result is not so surprising considering that the learned clauses are often long (20 or more literals). As seen previously, we only need two literals of a clause to be Undef on all assignments of an assignment group for the aggregate check to be effective and conclude that the clause will not trigger.

Similarly we computed the average number of clauses imported for each assignment sent to the GPU, the result was 2.132. Similarly, the average number of clauses present on the GPU at the end of the run was 5.05 millions.

7 Comparison with watched literal scheme

7.1 The watched literal scheme

The two watched literal scheme [4] is used by modern CPU SAT solvers to tell when a clause is in conflict or implying some variable. For each clause, it keeps two literals such that either:

- The value of one of them is True. If the other one is False, then its level must be greater or equal to the level of the True one.
- The value of both of them is Undef.

Whenever a watched literal becomes False (and the other one is not True), the solver will iterate over the clause to find another literal with value True or Undef to watch. If no such literal can be found, the clause is either in conflict, or implying a variable.

7.2 Comparison

The algorithm code 5.3 is used to tell which clause triggers on some assignments, just like the watched literal scheme. The watched literal scheme has much lower latency, it is much faster at noticing that a clause triggers. It will notice all clauses that trigger before doing another decision. In comparison, our GPU based algorithm takes more time to notice which clause trigger. It is heavily parallelizable and able to run on the GPU, though.

7.3 Efficiency of GPU checks

Given a set of successive assignments coming from a single CPU solver thread: we are going to show that if a clause triggers on their aggregate assignment: if we had applied the two-watched literal scheme on this same clause: it would have had to look at the clause.

Suppose that the clause C of size s triggers on the aggregate assignment. On the first assignment, let's call l1 and l2 the two literals it watches. Let's also assume that the clause does not trigger on this assignment. By definition of the aggregate assignment, there are at least s - 1 literals in C for which at least one assignment takes the value False. So at least one of l1 or l2 takes the value False for at least one assignment. So the watched literal scheme would have had to look at this clause at least once.

As a note, we do not send all assignments coming from a CPU solver thread to the GPU, only some of them. But the above stays true.

Let's consider the algorithm code 5.3, a clause and a CPU solver thread. The number of calls to assignmentTriggers for this clause and the assignments of this solver is lower than the

number of times we would have looked at this clause as part of the two-watched literal scheme if this clause was attached to this CPU solver thread.

8 Conclusion and future work

We have shown that the GPU can be used to improve the performance of a portfolio parallel SAT solver. Given a large set of assignment (up to 1024), and millions of clauses, the GPU is very efficient in noticing a posteriori which clause would have been useful to have on these assignments. This allows CPU solver threads to efficiently import only clauses that would have been useful to have.

The GPU reads the assignments from global memory. We could probably make the GPU much faster by reading some of them from shared memory. If, inside a GPU block, all the threads were to look at clauses that are very close to each other (i.e. similar variables): by putting the values of some of these variables in shared memory, we could probably improve GPU performance.

In our algorithm: checking an aggregate over 32 assignments is enough in the vast majority of cases to tell that a clause does not trigger. For this reason, we could try to compute the aggregate of 32 aggregates of up to 32 assignments. When the current algorithm handles up to 32 x 32 assignments, that would allow us to handle 32 x 32 x 32 assignments.

References

[1] Audemard, Gilles and Simon, Laurent. Lazy clause exchange policy for parallel SAT solvers. In International Conference on Theory and Applications of Satisfiability Testing, pages 197–205, Springer, 2014

[2] Vallade, Vincent and Le Frioux, Ludovic and Baarir, Souheib and Sopena, Julien and Ganesh, Vijay and Kordon, Fabrice Community and LBD-Based Clause Sharing Policy for Parallel SAT Solving In International Conference on Theory and Applications of Satisfiability Testing pages 11–27, Springer, 2020

[3] Audemard, Gilles and Hoessen, Benoît and Jabbour, Saïd and Lagniez, Jean-Marie and Piette, Cédric Revisiting clause exchange in parallel SAT solving In International Conference on Theory and Applications of Satisfiability Testing pages 200–213, Springer, 2012

[4] Thorsten Ehlers, Dirk Nowotka and Philipp Sieweck Communication in massively-parallel SAT Solving IEEE 26th International conference on tools with artificial intelligence, 2014

[5] Osama, Muhammad, and Anton Wijs. Parallel SAT simplification on GPU architectures. International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Cham, 2019

[6] Manolios, Panagiotis, and Yimin Zhang. Implementing survey propagation on graphics processing units. International Conference on Theory and Applications of Satisfiability Testing. Springer, Berlin, Heidelberg, 2006

[7] Dal Palù, Alessandro and Dovier, Agostino and Formisano, Andrea and Pontelli, Enrico. Cud@ sat: Sat solving on gpus. Journal of Experimental & Theoretical Artificial Intelligence 27.3 (2015): 293-316.

[8] Moskewicz, Matthew W., et al. Chaff: Engineering an efficient SAT solver. Proceedings of the 38th annual Design Automation Conference. 2001.

[9] Heule, Marijn, and Hans Van Maaren. Parallel SAT solving using bit-level operations. Journal on Satisfiability, Boolean Modeling and Computation 4.2-4 (2008): 99-116.