SAT-Accel: A Modern SAT Solver on a FPGA

Michael Lo
milo168@ucla.edu
University of California Los Angeles
Los Angeles, California, USA

Mau-Chung Frank Chang
mfchang@ee.ucla.edu
University of California Los Angeles
Los Angeles, California, USA

Jason Cong
cong@cs.ucla.edu
University of California Los Angeles
Los Angeles, California, USA

Abstract

Boolean satisfiability (SAT) solving is the first known NP-complete problem and is widely used in many application domains. Over the years, there have been so many consistent improvements in this area such that larger instances can be solved relatively quickly. Although these improvements have found their way onto CPU implementations, there has been limited progress adopting this on hardware accelerators mainly because it is difficult to implement the dynamic data structures needed to support a modern SAT solving algorithm.

In this work, we present SAT-Accel, an algorithm-hardware co-design solver that applies many of the core improvements found in modern SAT solvers. SAT-Accel uses a novel memory management system and representation that supports the dynamic data structures required by a modern SAT solving algorithm. Our design can achieve on average a 17.9x speedup against MiniSat, the previous state of the art CPU solver, and a 2.8x speedup against Kissat, the current state of the art CPU solver. Compared to the current state-of-the-art stand-alone hardware accelerator, SAT-Hard, SAT-Accel achieves on average 800.0x speedup.

CCS Concepts

• **Computer systems organization → Reconfigurable computing**.

Keywords

High-Level Synthesis, Boolean Satisfiability, FPGA, Accelerator

ACM Reference Format:
Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2025. SAT-Accel: A Modern SAT Solver on a FPGA. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '25), February 27-March 1, 2025, Monterey, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3706628.3708869

1 Introduction

With the end of Dennard scaling [23], customized computing is important to improving computing efficiency [20, 16]. Specialized accelerators built using ASICs and Field Programmable Gate Arrays (FPGAs) have been deployed for various critical application domains, such as, machine learning [31, 79, 74, 78, 11, 35, 32, 62, 36, 28, 83], data processing [58, 67, 60, 57, 7, 37], genome sequencing [76, 15, 46, 63, 25], and many more.

FPGA '25, February 27-March 1, 2025, Monterey, CA, USA

ACM ISBN 979-8-4007-1396-5/25/02
https://doi.org/10.1145/3706628.3708869

The satisfiability problem is a fundamental problem in computer science and is the first problem shown to be NP-complete [19]. SAT solvers have been widely used in many applications like automated reasoning [61, 8, 50], circuit verification [10, 49, 64], FPGA routing [75, 53, 54, 29, 52], program synthesis [66, 5], and quantum computing compilation [69, 68, 45, 70, 44]. Thanks to the advancements made to the algorithm, modern SAT solvers are increasingly more powerful in that they can compute large instances. At the same time, the CPU hardware that runs these software algorithms has improved. A study by Fichte et al. [27] shows that while software and hardware advancements are both important, algorithmic improvements have a larger impact. The authors revealed that early 2000's SAT algorithms running on modern CPUs solved fewer instances than modern SAT algorithms running on early 2000's CPUs. This experiment underscores the importance of algorithm innovation for SAT solving. We believe that accelerating a modern SAT algorithm can lead further speedup using customized architectures. Since HLS tools for FPGAs can help to greatly reduce design turnaround times [33, 18], and facilitate algorithm-hardware co-designs, our initial acceleration is based on FPGAs.

The usual representation of a SAT instance is in the conjunctive normal form (CNF), where

$$F = (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4)... \quad (1)$$

In this equation, x_1, x_2, x_3, and x_4, are variables that can take on the value TRUE or FALSE. Together, these variables form clauses where each clause uses only the logical operators $\vee$ (OR) and $\neg$ (NOT). In the above example, there are 3 clauses where C_1=($\neg x_1 \vee x_2$), C_2=($x_1 \vee \neg x_3$), and C_3=($x_2 \vee x_3 \vee x_4$). Finally, the clauses are concatenated with the logical operator $\wedge$ (AND). The goal is to answer whether the formulation has a set of variable assignments that makes the formula evaluate to TRUE. However, there are multiple challenges to support SAT solving on a hardware accelerator.

Challenge 1: Managing Dynamic Memory. The first is that an SAT instance could have any number of clauses with each clause having arbitrary length. Moreover, during the solving process, additional clauses can be added and deleted. This makes it challenging for hardware accelerators to manage the memory space efficiently. For example, one design challenge is to consider how to limit memory fragmentation and how to make memory allocation and deletion consume a fraction of the overall solving time. There have been several proposed works to address this issue for the FPGA (e.g. [30, 24, 42]). However, our design requires much lower overhead and a simpler customized solution.

Challenge 2: Coping With Scan-Like Operations. Another example is the sequential scanning operations used extensively in the solver. Using equation (1) as an example, if C_3 had 2 out of its 3 variables assigned as FALSE, the solver needs to know the last variable so that it can assign it as TRUE to avoid C_3 being falsified. A scan of

C_3 would be needed, however, such a scan would be slow on an FPGA when compared to a CPU. For example, a 3 GHz CPU can check 10 elements by the time an 300 MHz FPGA is able to check 1.

Challenge 3: Extracting More Task Parallelism. The SAT algorithm is sequential in nature and there are strong dependencies between tasks. This makes it hard for the algorithm to have computational overlap through task-level parallelism. There is fine-grain parallelism within some tasks but that is not enough to achieve significant speedup. This limitation is why there are very few parallel SAT solvers on CPUs and GPUs. In particular, GPUs are based on the Single Instruction Multiple Data (SIMD) architecture. SAT solving, however, requires complex control flow (Algorithm 2), something that a FPGA is flexible enough to support.

To address these three major challenges, we present SAT-Accel, a stand-alone **algorithm-hardware co-designed** FPGA based accelerator for modern SAT solving. It is a stand-alone solver as it only communicates with the host to receive the input CNF and to notify the host of an answer. The host does not do any other operations when the FPGA solver is executing. In addition, SAT-Accel has some configurable runtime parameters, making our design flexible for adaption without resynthesis.

In summary, we make the following contributions:

- SAT-Accel is the first algorithm-hardware co-design to implement the core features used in modern SAT solving.
- SAT-Accel employs a novel page based memory management system that efficiently allocates and reclaims pages with low overhead. This system minimally impacts the solver runtime as it does not require shifting memory around to reduce memory fragmentation.
- SAT-Accel uses a novel technique to create the signature of a clause with a fixed size regardless of clause length and uses it to identify the last variable in a clause without scanning.
- We find additional task parallelism to the modern SAT solving algorithm which are missed by previous works.
- SAT-Accel supports 32k variables and 131k clauses, an 8x increase in the number of clauses compared to other accelerated stand-alone solvers.
- We evaluate SAT-Accel and show that it outperforms state-of-the-art solvers (17.9x over MiniSat, CPU-based solver, 2.8x over Kissat, CPU-based solver, and 800.0x over SAT-Hard, FPGA-based solver).

2 Modern SAT Solver Overview

The basic algorithm used for SAT solving is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm show in Algorithm 1 [21, 22]. It is a resolution based algorithm that decides on a variable and assigns it a TRUE or FALSE value. The solver then takes the assigned value, called a *literal*, and searches if any clauses evaluate to FALSE from the assignment. If no clauses are FALSE, the solver decides on another variable. However, if there is a clause where only one variable remains while all the other literals are falsified, that variable is assigned the value that would make the clause evaluate to TRUE. This is called *propagation* in that the literal is implicated and will trigger a new round of clause checking. If a literal did falsify a clause, the solver would backtrack to the last decided variable and try its negation. If the negation also resulted in a falsified clause, it would use the decided variable before it and try its negation. This process would be repeated in a recursive way until the instance was solved, resulting in exponential run-time in the worst case.

Algorithm 1 Basic (DPLL) SAT Solving Algorithm

```
isBT = FALSE                                   ▷ is backtracking
dl = 0                                         ▷ decision level
propagateAsserted()                            ▷ cls. of length 1
if a cls. == FALSE then
    return FALSE
end if
while instance F is unknown do
    if isBT == FALSE then
        dl++
        AssignVar()                            ▷ pick a random var.
    end if
    Propagate()          ▷ scan all clauses to check for implication or false clauses
    if All var. assigned && all cls. == TRUE then
        return TRUE
    end if
    if dl == 0 && a cls. == FALSE then
        return FALSE
    end if
    isBT = FALSE
    if a cls. == FALSE then
        Backtrack()      ▷ undo to last decision and try ¬
        isBT = TRUE
    end if
end while
```

There have been significant improvements to the DPLL algorithm in the past three decades. For an improvement to be considered as a notable contribution, it must be robust against input shuffling. Biere et al. have shown that many solvers making such claims struggle to solve the same instance when the inputs are reordered [14]. In addition, Katebi et al. conducted an ablation study of a modern SAT solver [39]. Their study summarizes their finding when one of the feature was excluded and listed below are the major improvements that have been widely adopted:

- Conflict Driven Clause Learning (CDCL)
- Clause Minimization
- Clause Deletion
- Solver Restart Policy
- Phase Saving
- Variable Decision Heuristic
- Efficient Clause Checking (ECC)

Algorithm 2 depicts a typical modern SAT solver and it consists of the aforementioned points marked in blue. In the rest of this section, we describe how each improvement impacts SAT solving.

2.1 Conflict Driven Clause Learning

Conflict Driver Clause Learning (CDCL) is the backbone of today's modern SAT solver. CDCL is a powerful feature that allows non-chronological backtracking. In the base DPLL algorithm, when a

clause becomes FALSE, it can only backtrack to the last decided variable (latest decision level) and try its negation. This is called *chronological backtracking*. CDCL enables non-chronological backtracking, or backtracking up multiple decision levels. The learned clause contains the backtracking level and prevents the solver from assigning a specific set of conflicting assignments from happening again [47, 48, 80]. There are many strategies to produce a learned clause, but many modern solvers use an implication graph and cut the graph utilizing the first unique implication point (1-UIP).

2.2 Clause Minimization

The learned clauses being produced can be long and can be further shrunk due to redundancy [65]. The reduction of the learned clause decreases memory usage and speeds up solving time as there are fewer literals to check during propagation. Clause minimization reuses the same implication graph for clause learning to determine if a variable is removable.

2.3 Clause Deletion

Throughout the solving process, many clauses are learned, but some of them are not as useful. To determine usefulness, Audemard and Simon proposed scoring the quality of learned clauses by calculating the *literal block distance*, which is defined as the number of unique decision levels. They employ this technique in their Glucose solver [9]. Clauses with the least number of unique decision levels are believed to have higher quality since they are more likely to lead to propagation. Modern solvers delete clauses of low quality after a certain amount of learning or during a restart.

2.4 Solver Restart Policy

SAT solvers can get stuck when they are in too deep in their current search path. To resolve this issue, solvers restart from the beginning by backtracking all the way to decision level 0 while keeping the learned clauses. There are many types of restart strategies ranging from constant to custom sequences. Instead, many modern solvers utilize the Luby sequence (formula in section 5.9) as it has been empirically proven to be better overall to other strategies [13].

2.5 Phase Saving

During backtracking, it is possible that variable assignments for a sub-problem not related to the current backtracking process are removed. To ensure the effort of solving the sub-problem is not in vain, the solver saves those assignments [56], known as *phase saving*. By saving those assignments, phase saving guides the solver to a more optimal search path since it can use the saved assignments instead of guessing when it selects a variable. However, propagation can override the saved assignments as it takes priority in that it tries not to falsify a clause.

2.6 Variable Decision Heuristic

Choosing which variable for the next search can make a large difference in a solver's runtime. There are many heuristics a solver can employ but the most widely utilized heuristic in modern SAT solving is Variable State Independent Decaying Sum (VSIDS). The goal of VSIDS is to choose variables that were actively utilized during clause learning at the decision step [51]. VSIDS is a powerful heuristic because it exploits community structures in the graphical representation of the SAT instance [43].

Algorithm 2 Typical Modern SAT Solving Algorithm

```
isBT = FALSE                                          ▷ is backtracking
dl = 0                                                ▷ decision level
PropagateAsserted()                                   ▷ cls. of length 1
if a cls. == FALSE then
    return FALSE
end if
while instance F is unknown do
    if isBT == FALSE then
        dl++
        AssignVar()
            VariableDec.Heur.()                       ▷ pick by some metric
                                                        such as VSIDS
    end if
    PropagateEnh.(&impGraph)                          ▷ (ECC) Check necessary
                                                        clauses, build igraph
    if All var. assigned && all cls. == TRUE then
        return TRUE
    end if
    if dl == 0 && a cls. == FALSE then
        return FALSE
    end if
    isBT = FALSE
    if a cls. == FALSE then
        AnalyzeConflict()
            dl,[lrnCls]=LearnCls(impGraph)            ▷ (CDCL)
                                                        learn clause
            [minCls]=ClsMin([lrnCls],impGraph)        ▷ shorten
                                                        clause
            Save([minCls])
        Backtrack(dl)                                 ▷ undo up to dl
            Restart()                                 ▷ undo to dl=0 on restart policy met
            SavePhase()                               ▷ save variable assignment
        isBT = TRUE
    end if
    if newClsCount == threshold then
        ClauseDelete()                                ▷ delete clauses by some metric
    end if
end while
```

2.7 Efficient Clause Checking

The base DPLL algorithm checks all clauses during propagation even though the literal being propagated is not seen in some clauses. This unnecessarily bloats the solving time. Another issue is that SAT solving has a memory access pattern that has little spatial or temporal locality. To reduce these negative affects, modern SAT solvers employ the 2 Watched Literal scheme. This scheme reduces the number of clauses that need to be checked by tracking two variables in each clause [51]. Whenever a variable is assigned, clauses that track those variables find another to track. If a clause has the assigned variable but is not tracking it, the clause does not need to find a new variable. Another benefit of 2 Watched Literal is that it

Table 1: SAT solver acceleration feature comparison. There are 2 acceleration categories: stand-alone and co-processing design. SAT-Accel is the first stand-alone to incorporate many of the modern solving features.

	Stand -alone	Clause Learning	Clause Minimi.	Clause Deletion	Solver Restart	Phase Saving	Variable Decision Heuristic	Efficient Clause Checking	Problem Size
Park et.al [55]	-	-	-	-	-	-	-	✓ Not 2WL	32M Cls Estimate
Ustaoglu et .al [73]	✓	✓ 1-UIP	-	-	-	-	✓ Static	✓ Not 2WL	16k Cls 3-SAT
Thong et. al [71]	-	-	-	-	-	-	-	✓ Not 2WL	250k Cls
Yuan et. al [77]	✓	-	-	-	-	-	✓ Static	✓ Not 2WL	<20 Var
Safar et. al [59]	✓	✓ 1-UIP	-	-	-	-	✓ Static	✓ Not 2WL	511 Var 511 Cls 3-SAT
Haller et. al [34]	-	-	-	-	-	-	-	✓	64k Var 176k Cls
SAT-Accel	✓	✓ **1-UIP**	✓ **Recursive**	✓ **LBD Del. on reset**	✓ **Luby**	✓	✓ **VSIDS**	✓ **State-based**	**32k Var 131k Cls k-SAT**

is stateless, meaning that backtracking does not require any further clause visits. Whereas a stateful approach requires additional clause visits to increment a tracking counter.

3 Related Hardware Accelerator Works

There has been much less progress on the application of hardware accelerators for SAT solving than advances in the algorithm and software implementation. As mentioned in section 2, these algorithm advancements are more significant than the 20-plus years of CPU and hardware accelerator improvements. For hardware accelerators to be competitive against their CPU counterparts, the algorithm advancements need to be considered.

There have been two waves for accelerating SAT on hardware accelerators. From the early 1990s to the early 2000s, the SAT instance in question would be directly implemented on FPGAs or ASICs [81, 41, 4]. This led to a long turnaround time as every new instance had to be compiled to the hardware and most of the time would be spent on compilation.

From the mid-2000s, there was a shift from instance-specific acceleration to accelerating the SAT solving algorithm itself. Most of these works use accelerators to offload the most time consuming functions. It was well known at the time that the propagation function contributed to most of the run time [55, 34, 71]. Even though these co-processor accelerators improved computation performance, communication overhead between the CPU and the accelerator was not taken into account. This overhead can be significant as the number of call for propagation can easily be in the millions.

Some works have attempted to implement a stand-alone SAT solver onto hardware. However, these designs were not robust or were lacking many of the characteristics of a modern SAT solver. For example, Yuan et al. enumerated the last few remaining variables onto a hardware accelerator exhaustively [77]. However, this was not a scalable solution as the number of possible combinations grew exponentially. Safar et al. and Salem proposed an FPGA accelerator design that included clause learning [59]. Similarly, the authors of SAT-Hard implemented CDCL on top of the DPLL algorithm, allowing for non-chronological backtracking [73]. Both Safar and SAT-Hard implementations did not include other improvements and were restricted to 3-SAT instances, where each clause had at most 3 variables. SatIn proposed a system on chip in which a distributed clause processing architecture, paired with an integrated CPU, could run many modern heuristics [82]. However, only simulation results are reported and it is limited by having a fixed clause length of 8. Although it is possible to convert any k-SAT into a 3-SAT using the Tseitin transformation, the search time may increase exponentially as the number of intermediate variables introduced by the transformation increases [40].

4 Overview of the SAT-Accel Architecture

Figure 1 shows an overview of SAT-Accel's architecture where it consists of 5 modules and 2 memory managers. These modules communicate with each other while the memory managers handle memory operations for them.

Each module is responsible for executing a certain part of the modern SAT solving algorithm. The variable decision heuristic module is responsible for choosing which variable the propagation module should check and uses the VSIDS heuristic as described earlier. The propagation module checks whether a clause becomes unit or unsatisfied. When there are no more clauses to check and none have been falsified, it notifies the decision heuristic module to pick another variable. If a clause becomes falsified, the learning module is notified. The learning module determines the backtrack level and the learned clause by employing the 1-UIP strategy [80]. It also tells the decision heuristic module to update the scores of the variables. Once a learned clause is produced, control is given to the clause minimize and backtrack module. Both backtracking and clause minimization happen in parallel. The backtracker undoes the states created by the propagation module while also saving the variable assignment. Meanwhile, the clause minimizer checks if the variable in the learned clause is removable. Once all the variables have been checked, the minimized clause is then saved. If the reset counter has been reached, the deletion module is activated to prune a certain percentage of learned clauses before handing control back to the propagation module. The entire design works in a dataflow style with aggressive coarse-grain pipelining, which is intended to overcome Challenge 3 (sequential execution) stated in section 1.

4.1 Extracting Parallelism Between and Within Modules

Previous works have identified the *Propagate()* from Algorithm 1 and *PropagateEnhanced()* as parallelizable [55, 73, 71, 77, 59, 34]. SAT-Accel's propagation module contains 4 processing elements (PEs) and can check up to 4 clauses in parallel. Each PE handles a range of address where multiple addresses may be binned to the same PE, possibly causing stalls on the broadcast channel. SAT-Accel utilizes 8 PEs for backtracking, and they reverse the operation done by the propagation PEs. Section 6 explains the reasoning on the number of PEs used.

SAT-Accel follows the recursive clause minimization algorithm as described by Sorensson and Biere [65]. The algorithm for clause minimization itself is parallelizable. Each variable in the learned clause can be checked if its removable independently. SAT-Accel utilizes 2 PEs for clause minimization since on-chip memory is configured as dual port.

Although Algorithm 2 seems sequential in nature, there is still some task overlap. The variable decision heuristic module can update the variable scores as the learn module is resolving clauses. Clause minimization can be overlapped with backtracking since the level to backtrack to is known after learning a clause.

4.2 Memory Access Handling

All modules except for the decision heuristic module contain a memory access unit. This memory access unit communicates to the on-chip memory manager. Since there are multiple processing elements within the modules but only 1 access unit, with the clause minimize module being the exception, the access unit requests a large word of (word size)*(#PEs).

4.3 Software Configurable Parameters

SAT-Accel exposes some parameters during runtime. Users can customize the parameters such as the reset multiplier, deletion rate, VSIDS decay factor, and page size on the host side. This customizability enables the users to tune SAT-Accel to maximize its performance for a given problem set. To figure out what parameters to tune is beyond the scope of this paper and was not conducted but there have been research competitions looking into this [38].

5 SAT-Accel Implementation

In this section, we describe how SAT-Accel implements the core features of a modern SAT solver. In addition, we provide a detailed explanation along with the trade-offs of designing the memory management system and an FPGA friendly propagation function (Challenges 1 and 2). Our design only utilizes on-chip memory during the solving process.

5.1 VSIDS Decision Heuristic

SAT-Accel uses the well known decision heuristic called VSIDS. VSIDS selects the variable with the highest activity as the next variable to be checked for propagation. A variable is considered to be more active than others if it was used frequently to generate learned clauses [51]. Variables that are used less frequently will decay, and this decay rate is configured by the user. Initially, all variables have a score of 0 so the variable that appears lexicographically is selected. To retrieve the variable with the highest score, SAT-Accel utilizes a priority queue implemented using BRAMs and URAMs. Since the number of variables in a SAT instance does not change, dynamic updates of elements are swapped either higher (insert) or lower (pop) within the on-chip memory.

5.2 Accelerator Friendly Propagation

Given that an SAT instance is represented as a CNF, SAT-Accel needs to know which clauses to access for a given variable during propagation and does this by using a transposed representation. The transposed representation has each variable containing 2 lists of clauses. One list contains clauses that appears as non-negated while the other list has the negated. This representation is different from the CNF representation (non-transposed) where each clause contains a list of variables as seen in Figure 4. The list to access depends on the assigned value of the variable.

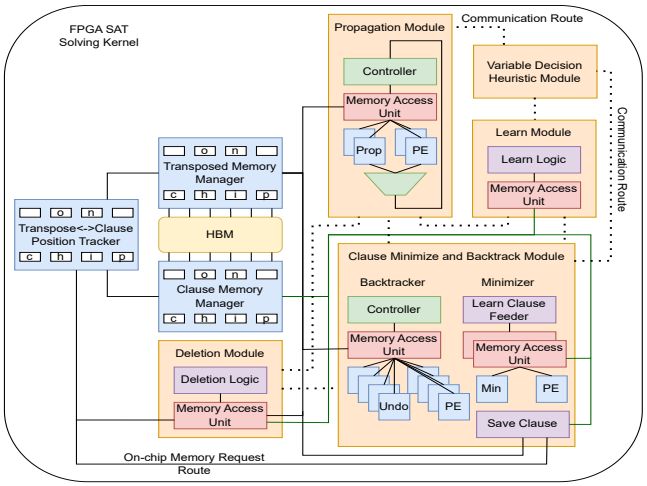

Figure 1: The overall architecture of SAT-Accel consists of 5 modules and 2 memory managers. Each module is responsible for a part of the feature used in a modern solver.

Many previous works identified propagation as the bottleneck of SAT solving. For hardware accelerators, the scan-like operations for propagation as stated in Challenge 2 particularly costly. This means that the 2 Watched Literal mentioned in section 2.7 must be replaced with an FPGA friendly alternative. Since SAT-Accel only cares when a clause becomes unit (1 unassigned variable left) or falsified during propagation, it uses an 8 byte representation of the clause. 4 bytes are used for the clause's signature by XOR all the variables in that clause, and another 4 bytes to track the number of unassigned variables. For example:

$$(x_1(001) \vee x_2(010) \vee x_4(100)) \rightarrow \text{sig: } 7(111), \text{ ucnt: } 3 \qquad (2)$$

where $\rightarrow$ represents the transformation. During propagation, when a variable is falsified, it is XOR with the clause's signature. If the clause is only XOR with falsified variables, the last unassigned variable in the clause has to be unit. Using the above example, if x_1 and x_2 were assigned FALSE, then XOR of x_1 and x_2 would produce x_4. Both propagation and backtracking utilize signatures.

5.2.1 Trade-Off Over 2-Watched Literal. The signature of a clause and unassigned count changes during propagation, making this method a state-based approach. Backtracking is needed so that a clause's state can be reverted. However, this is acceptable since backtracking can be overlapped with clause minimization. 2-Watched Literal does not have this issue since it does not use states.

5.3 Implication Graph Representation

The implication graph is used to represent the implication of a variable and is used for clause learning and minimization. The vertices of the graph are variables determined by propagation or decision while the edges are unit clauses. The relation between vertices is determined by drawing directed edges from the variables of the unit clause they are in. Decided variables are considered as source vertices since no clause propagates them as demonstrated in Figure 2. SAT-Accel implements the implication graph as a stack where it

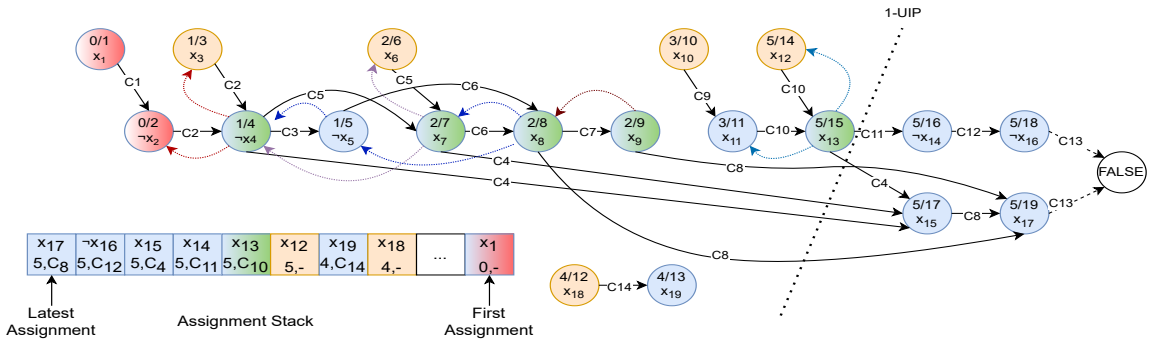

Figure 2: The implication graph and its stack representation employed for clause learning and minimization. Decided variables are shaded orange while propagated variables are shaded blue. In the graph, 5/19 represents decision level 5 and position 19 in the stack. For clause minimization, red is ASSERTED, orange is DECIDED, and green is LEARNED. The dotted colored arrows represent the clauses used for resolution to check if the learned variable is removable. The dashed line represents the 1-UIP cut since it contains only one variable from the latest decision level(x_{13}, level 5) and is closest to the conflict source.

stores the decision level and the clause id. Decided variables do not have an clause id associated with it, but propagated variables do. Multiple clauses may propagate the same variable, which requires a dynamic data structure to store them. Instead, SAT-Accel works around this by using the shortest clause as this helps with clause minimization as explained later.

5.4 Linear Time Logic Resolution

Both clause learning and clause minimization utilize the logic resolution rule to generate new clauses. The resolution rule creates a new clause by excluding complementary variables. Given 2 clauses, $C_x : (x_1 \vee x_2 \vee x_3)$ and $C_y : (\neg x_3 \vee x_4 \vee x_5)$, the new clause when C_x and C_y are resolved is $C_{new} : (x_1 \vee x_2 \vee x_4 \vee x_5)$.

A naive solution to resolve clauses would take $O(n^2)$ as each element of the clause needs to be checked against the other. SAT-Accel implements the logic resolution in linear time using a position table and a 2-bit masking table stored in BRAMs shown in Figure 3. The position table tracks the position of the variable in the learned clause array. The 2-bit masking table is used to track if a variable in its non-negated and negated form is seen. The first bit is for the non-negated and the second bit is for the negated. If both bits are enabled, the variable is removed from the learned clause because the negated and non-negated variable resolved each other.

5.5 1-UIP Clause Learning

Many modern solvers learn clauses that have the Unique Implication Point (UIP) property. A learned clause is UIP when it contains only one variable from the latest decision level [80]. By only having one variable from the latest decision level, SAT solvers can immediately propagate after backtracking. A clause is **1-UIP** when the learned clause only contains a variable from the latest decision level **and** is closest to the conflict source. This can be represented as a specific cut in the implication graph as seen in Figure 2. SAT-Accel resolves clauses by scanning the stack represented implication graph. The clause to resolve is determined by the variable that was inserted latest in the stack and is in the current learned clause.

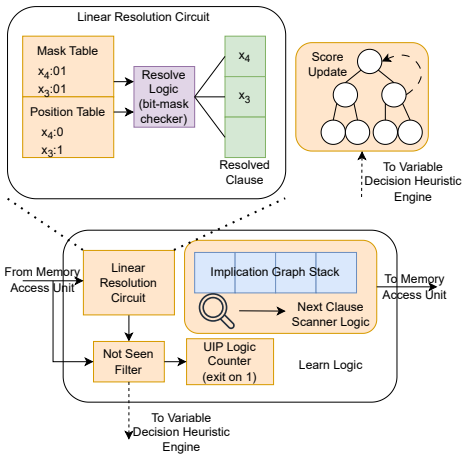

Figure 3: The learning logic works in parallel with the heuristic module to update variable scores. Variables that are scanned during clause learning are sent to the decision heuristic. Each variable's score is updated once per learning. A 2-bit table to track each variable's negation and non-negation so that resolution can be done in linear time.

Figure 3 shows how SAT-Accel determines if the learned clause is 1-UIP by tracking the number of variables of the latest decision level of the learned clause. A 1-UIP clause is produced when the counter is 1, otherwise resolution with another clause is applied. Using Figure 2 as an example, C_{13} (falsified clause, 1-UIPCnt: 2) is read in and then resolved with C_8 (x_{17} in stack pos. 19, 1-UIPCnt: 2), C_{12} ($\neg x_{16}$ in stack pos. 18, 1-UIPCnt: 2), C_4 (x_{15} in stack pos. 17, 1-UIPCnt: 2), and lastly C_{11} ($\neg x_{14}$ in stack pos. 16, 1-UIPCnt: 1).

5.6 Recursive Minimization

Clause minimization is utilized to decrease the length of the 1-UIP learned clause. First, SAT-Accel marks the variables in the stack as

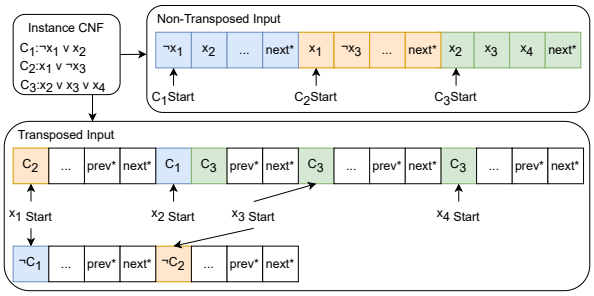

Figure 4: The memory representation of the formulation and its tranposition. The non-transposed (original) input is used for clause learning and minimization while the tranposed input is employed for propagation.

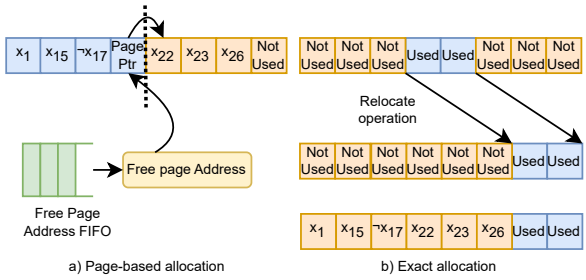

Figure 5: Exact allocation requires defragmentation if the memory manager cannot find a contiguous memory for the clause. Whereas for the page-based allocation, the clause can be stored as long as there are enough pages at the cost of some pages being partially utilized.

DECIDED (from VSIDS), **ASSERTED (level 0)**, or **LEARNED** during clause learning as seen in Figure 2. Next, a minimize PE resolves clauses based on the implication graph. The resolution step is applied until the resolved clause has a decided variable or contains only asserted and learned variables. A counter is used to represent the above conditions by counting the number of asserted and learned variables. If the counter is equal to the length of the resolved clause, the variable being checked can be removed. This means that the resolved clause does not have a **DECIDED** marker or non-learned variable. Using Figure 2 as an example, x_8 starts with C_6 which contains the non-learned variable $\neg x_5$ and so needs to resolve one more time.

5.7 Low Overhead Memory Manager

SAT-Accel employs a page-based memory management system to solve Challenge 1. The pages are mapped to the FPGA's URAMs and users can specify how large the pages are. A list of available page addresses are stored in a first-in-first-out (FIFO) queue. When clauses are saved, page addresses are popped from the queue while page addresses are inserted back when clauses are deleted.

There could be a possibility that non-contiguous page addresses are allocated and because of this, the last element on the page is used to point to the next given page address. For the transposed input representation, the last two elements are reserved for pointing to the next and previous page. The use of the previous pointer is explained in the next section. For the non-transposed input representation, the last element is reserved for pointing to the next page. Both representations are implemented using URAMs.

5.7.1 Alternative to Page-Based. The overhead for the page-based implementation is due to the page pointers and possible under utilization on the last page of each list. The alternative is to employ an exact allocation strategy such that space is not wasted. However, fragmentation becomes a problem and defragmentating requires scanning through the entire memory space as depicted in Figure 5. This strategy is not scalable if the total memory space increases and although the defragmentation step would rarely occur, a page-based solution is still preferred as it does not perform the defragmentation step at all. Prior works have implemented a dynamic memory manager to efficiently utilize the on-chip memory space such as [30, 24, 42], but our design requires a simpler one.

5.8 Clause Deletion

SAT-Accel utilizes the literal block distance scoring proposed by the Glucose solver [9]. The literal block distance is calculated as the number of unique decision levels in the learned minimized clause and clauses of higher distance are deleted first. SAT-Accel has 10 buckets implemented as FIFOs to retrieve clause ids to be deleted. SAT-Accel utilizes an user specified deletion rate to remove clauses at every restart.

To delete a clause from the non-transposed memory, the page addresses are inserted back into the available page address queue. If multiple pages were allocated, the next page pointers are followed. Since the clause id is no longer valid, the transposed memory needs to be updated. The deleted clause id is replaced by the latest element from the latest page of the variable. This is the preferred method since it does not require shifting all the elements. If the latest page contains zero elements, it is reclaimed. The previous page pointer is read in to let SAT-Accel know that the variable's latest page is now at a different address. Instead of having to traverse from starting page, SAT-Accel can directly start from the last page so that it can save new clause ids or allocate new pages quickly.

When removing the clause id from the transposed memory, an original-to-transposed and transposed-to-original position array is used. This is to prevent the scanning of the transposed memory to locate the clause id to be deleted. The original-to-transposed position array stores the location of where the clause is located in the transposed CNF array while the transposed-to-original array does the inverse. Both these arrays are implemented using URAMs.

5.9 Phase Saving and Restart

The backtracking PEs save the variables assignment value using a 1-bit array where the value denotes non-negation or negation. VSIDS will use the assignment value for that variable when selected as a decision. SAT-Accel follows the Luby sequence and is defined as the following [13]:

$$t_i = \begin{cases} 2^{k-1} & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1} & \text{if } 2^{k-1} \leq i < 2^k - 1 \end{cases} \quad (3)$$

Table 2: The resource utilization broken down by modules.

Module	BRAM	DSP	FF	LUT	URAM
Decision	2%	63%	9%	10%	5%
Propagation	35%	0%	21%	18%	0%
Learn	51%	0%	21%	16%	0%
Min/Btrk	1%	0%	25%	27%	0%
Deletion	0%	0%	6%	5%	0%
Cls Store	9%	37%	15%	15%	33%
Tran. Store	0%	0%	1%	2%	29%
Tran<->Cls Position	0%	0%	3%	6%	33%
Absolute	419 (21%)	48 (0.5%)	324,891 (12%)	251,283 (19%)	778 (81%)
Available	2,016	9,024	2,607,360	1,303,680	960

where t_i is the number of restarts. A scaling factor is multiplied on top as the base sequence results in premature restarts. SAT-Accel allows the user to specify what this scaling factor is before solving. A restart is triggered after a certain number of clauses are learned. When a restart is triggered, the design backtracks all the way to the start of decision level 1, ignoring the backtrack level calculated from the conflict analysis. Clause deletion is triggered on a restart with the latest clause learned being exempt from removal.

6 Evaluation

Our setup runs on a 16nm AMD/Xilinx U55c FPGA written in High Level Synthesis (HLS) [18] and is compiled using Vitis HLS 2022.2. SAT-Accel runs at 230 MHz and due to routing congestion, Vitis was unable to meet timing at higher frequencies. Our critical path from the congestion report stems from the priority queue as it requires many logical operations to be done in a single cycle. Table 2 reports resource utilization used for synthesis as percentages.

The CPU solvers we tested are running on a 7nm AMD EPYC 7V13 64 Core Processor at 2.5 GHz. This CPU has a L1 and L3 cache size of 4MB and 512MB respectively. The available system RAM size is 512GB. To get a sense of our design's performance in a comparison where the same or more SAT improvements are employed, we tested it against MiniSat (previous SoTA) and Kissat (today's SoTA). MiniSat won the SAT competition in 2005 and had spawned its own category in the SAT competition up until 2014 [1, 26] Kissat is considered to be the current SoTA since it won the SAT competition in 2020 and there have been many specialized versions based off of it [12, 2]. For MiniSat and Kissat, scaling to multiple cores does not offer any benefit as both were written for sequential execution. Both solvers were configured with the default options and include the aforementioned improvements and more. SAT-Accel was configured with a prune percentage of 10%, reset multiplier of 100, VSIDS decay factor of 0.95, transposed and non-transposed page size of 16 and 4 words respectively.

The benchmarks we have chosen originate from the SAT competition [1], SATLIB [3], or from our own applications. Our own benchmarks that were generated are nqueens_16, nqueens_32, sudoku_9_3_3, and QtmCkt from [45] for quantum compilation. The benchmarks listed in the first group of Table 4 are from the SAT competition, while those not mentioned are from SATLIB. The benchmarks from these sources that could not finish on the FPGA due to lack of sufficient on-chip memory are excluded from the comparison. The goal is to measure the time it took for a solver to produce an answer. If the FPGA solver ran out of on-chip memory, it was unable to determine if the instance is SAT. Furthermore, it is not possible to determine the progress of how close a solver is to producing an answer, making it difficult to estimate the execution time as if it had enough on-chip memory.

Table 3: Comparison of the instances used by SAT-Hard.

Problem Name	Var	Cls	Time in ms (SA/SAT-Hard)	
hole7_unsat	56	204	125	330
hole8_unsat	72	297	691	2,270
hole9_unsat	90	415	N/A	15,290
uf100-010_sat	100	430	1	580
uuf100-02_unsat	100	430	4	4,940
uf125-01_sat	125	538	4	1,160
uuf125-05_unsat	125	538	7	4,900
uf150-08_sat	150	645	1	3,920
CBS_k3_n100_m403_b10_1_sat	100	403	2	2,340
aim-200-3_4-yes1-4	200	680	4	1,200
aim-200-1_6-no-4	200	320	0.3	10
ii16e2_sat	532	7,825	4	5,760
ii32e1_sat	222	1,186	0.1	20
	SAT-Accel (SA)	Spdup	Avg:	800

Kissat's timing resolution is 10 milliseconds and for some instances, Kissat and SAT-Accel solved it in less than the timing resolution. For those cases, we exclude those instances when calculating the speedup as it would inflate the speedup for SAT-Accel. On average SAT-Accel is faster than MiniSat by 17.9x and faster than Kissat by 2.8x.

SAT-Accel's implements most of the heuristics seen in MiniSat. These include the same decision heuristics, clause learning, and clause minimization. Kissat, a newer solver compared to MiniSat, uses many more and newer heuristics that are not implemented in SAT-Accel. Despite these similarities and differences, SAT-Accel was able to outperform MiniSat while having some respectable speedup over Kissat.

SAT-Accel is more effective on quantum compilation examples than the SAT competition instances. More study is needed to understand why, but one hypothesis is that SAT competition instances are meant to find a faster solver in all domains versus a domain specific SAT solver[6]. Originally, our SAT-solver was designed for solving instances in the quantum computing domain.

6.1 Comparison Against SAT-Hard

SAT-Hard is considered the latest hardware-accelerated stand-alone SAT-solver using an FPGA [73]. SAT-Hard is an improvement over their previous stand-alone solver that was compared against a microprocessor implemented on an FPGA running MiniSat [72]. We compare SAT-Accel against SAT-Hard by using the SATLIB benchmarks utilized in their paper. Table 3 shows SAT-Accel is able to outperform SAT-Hard on all the compared instances except for the

Table 4: Benchmark results for SAT-Accel, MiniSat, and Kissat. The total solving time estimation of a co-processor that accelerates propagation is in the right hand column. In bold is if the estimated time is longer than running purely on CPU.

Problem Name	Literals	Clauses	Time in ms			MiniSat co-processor estimate in ms PCIE overhead in μs			
			SA	MS	KS	0.3μs	1μs	160μs	MS Prop. Cnt
dp10s10.shuffled	7,759	23,004	48	48	380	9	12	**725**	4,484
rand_net60-30-1	3,600	10,681	401	57	110	10	15	**967**	5,989
battleship-6-9-unsat	54	171	214	105	240	41	101	**13,604**	84,927
logistics-rotate-07t5	2,721	88,373	1,088	398	1,050	70	95	**5,702**	35,263
289-sat-6x30	720	27,360	556	104,294	50	19,077	27,268	**1,887,770**	11,701,275
php-010-008	80	370	1,951	203	690	65	148	**18,805**	117,345
marg3x3add8	41	224	328	868	90	583	**1,641**	**242,016**	1,511,787
marg3x3add8ch	41	272	6,387	4,088	220	2,233	**6,019**	**866,012**	5,408,764
bmc-ibm-1	9,685	55,870	54	63	80	12	17	**1,170**	7,257
bmc-ibm-2	2,810	11,683	1	7	<10	2	3	**279**	1,739
bmc-ibm-5	9,396	41,207	8	69	10	12	15	**785**	4,842
bmc-ibm-7	8,710	39,774	3	60	10	11	14	**852**	5,268
ssa7552-160	1,391	3,126	1	4	<10	1	1	**6**	33
ssa0432-003	435	1,027	1	2	<10	0.4	0.5	**18**	112
ssa2670-141	4,843	2,315	7	3	<10	1	1	**77**	481
ssa6288-047	17,303	34,238	0.4	4	<10	1	1	3	17
qg3-08	512	10,469	1	11	<10	2	2	**121**	745
qg6-10	1,000	33,466	8	14	10	2	3	**142**	873
qg6-11	1,331	49,204	37	53	60	9	13	**863**	5,345
qg6-12	1,728	69,931	925	373	960	63	81	**4,068**	25,079
qg7-10	1,000	33,736	1	11	<10	2	2	**118**	729
qg7-11	1,331	49,534	26	27	20	5	7	**419**	2,592
qg7-12	1,728	70,327	292	110	230	19	26	**1,538**	9,513
nqueens_16	256	6,336	0.2	11	<10	2	2	**13**	70
nqueens_32	1,024	52,608	9	50	10	8	8	**146**	865
sudoku_9_3_3	729	8,910	0.4	4	<10	1	1	**25**	155
blocksworld	459	7,054	0.3	4	<10	1	1	**7**	39
blocksworldc	3,016	50,457	25	30	30	5	5	**109**	655
4blocks	758	47,820	69	44	30	9	16	**1,518**	9,445
logisticsa	828	6,718	2	5	<10	1	1	**83**	514
logisticsb	843	7,301	4	6	<10	1	2	**193**	1,201
logisticsc	1,141	10,719	6	14	<10	3	4	**355**	2,205
logisticsd	4,713	21,991	7	33	10	5	6	**99**	590
QtmCkt4_4_1	82	434	0.2	2	<10	0	0.3	**9**	52
QtmCkt4_4_2	176	1,054	1	3	<10	1	1	**54**	338
QtmCkt4_4_3	252	1,614	3	1	<10	0.2	0.4	**46**	284
QtmCkt4_4_4	346	2,246	1	3	<10	1	1	**58**	359
QtmCkt9_6_1	312	2,261	1	1	10	0.2	0.2	**4**	24
QtmCkt9_6_2	642	5,377	9	10	<10	2	2	**90**	556
QtmCkt9_6_3	945	8,403	3	9	<10	2	2	**153**	947
QtmCkt9_6_4	1,275	11,537	6	14	<10	2	3	**94**	573
QtmCkt9_8_7	3,132	28,620	0.3	35	10	5	5	33	175
QtmCkt16_8_5	3,948	45,759	0.5	61	10	9	10	**67**	359
QtmCkt16_8_6	4,716	55,311	1	38	10	6	6	**71**	406
QtmCkt16_8_7	5,484	64,894	1	44	10	7	7	**90**	521
QtmCkt16_16_3	5,544	56,261	1	37	10	6	6	**81**	470
	SA	Speedup	Avg:	17.86	2.77	2.93	3.54	142.63	

Table 5: The memory management system is efficient as there is no noticeable increase when aggressively increasing the number of deleted clauses(from 10% to 90%). Propagation is still the bottleneck, followed by minimization and backtracking.

Operations	Load		Decide		Propagate		Learn		Min|Bktrk		Allocate		Delete	
Cls. Deletion Threshold	10%	90%	10%	90%	10%	90%	10%	90%	10%	90%	10%	90%	10%	90%
SAT-CMP	0.04%	0.01%	2.59%	2.37%	**42.25%**	37.83%	13.25%	17.81%	39.10%	**38.51%**	0.33%	0.43%	2.45%	3.05%
IBM	4.01%	3.91%	25.02%	25.01%	**39.51%**	**40.17%**	9.05%	8.95%	22.16%	21.55%	0.11%	0.11%	0.14%	0.30%
SSA	24.63%	24.49%	11.65%	11.01%	**29.65%**	**28.88%**	18.40%	17.76%	15.39%	17.17%	0.16%	0.16%	0.11%	0.53%
QG	3.76%	3.61%	0.95%	0.96%	**45.25%**	**45.20%**	19.91%	20.47%	29.48%	28.68%	0.25%	0.26%	0.40%	0.81%
PUZZLE	10.10%	9.95%	6.37%	6.24%	**37.34%**	**36.76%**	11.75%	12.44%	33.81%	32.93%	0.18%	0.18%	0.46%	1.50%
PLAN	1.33%	1.09%	13.44%	15.19%	**38.20%**	**39.12%**	15.44%	14.00%	31.02%	29.24%	0.32%	0.28%	0.25%	1.09%
QTMCKT	21.03%	22.25%	2.91%	3.22%	**33.34%**	**31.27%**	20.99%	20.74%	20.97%	20.51%	0.44%	0.43%	0.32%	1.57%

hole9_unsat benchmark due to running out of memory to store new clauses. This is because clause deletion allows for a slow growth of memory utilization since only a percentage is reclaimed. On average, SAT-Accel is 800.0x faster than SAT-Hard. The speedup seen over SAT-Hard could be attributed to SAT-Accel having applied more features of a modern SAT solver while SAT-Hard only implemented a selected few as shown in Table 1. This reinforces the point that hardware accelerators need to include the latest SAT solving techniques to remain competitive [27].

6.2 Comparison Against Propagation Based Accelerated Co-processor

We estimate the performance against a propagation co-processor that utilizes MiniSat. A critical measurement that cannot be ignored is the communication overhead between the host and the FPGA as the number of calls to propagation is significant. We make the following assumptions:

- PCIE communication overhead between host and the FPGA is non-zero. We use a lower and upper range of values (0.3μs and 160 μs) measured from the prior work [17].
- Time spent in propagation is 80-90% of MiniSat's total run time as measured in this paper [55].
- Enough resources to achieve the speedup of 6.7x [55].

Table 4 displays the number of times propagation is called in MiniSat for our benchmarks and is employed for calculating the overhead. Our equation utilized for estimating the solving time in *ms*:

$$t_{MS*} = \frac{t_{MiniSat}}{6.7} + \frac{MS\_Prop\_Cnt * PCIE_{\mu s} * 1ms}{1000\mu s} \quad (4)$$

We include a 1 μs column in Table 4 as this was the observed latency from non-related experiments. Even with a propagation co-processor, our design is calculated to be 2.9x (lower-bound) to 142.6x (upper-bound) faster and for some instances, the estimated solving time increases.

6.3 Analysis and Design Space Exploration

Table 5 breaks down each step that the solver spends for certain types of instances. Most of the time is still spent in propagation followed closely by clause minimization and backtracking. Highlighted in bold indicates which step took the longest time.

6.3.1 Propagation Latency. When breaking down SAT-Accel's solving time using the benchmarks from Table 4, most of the time was consumed in propagation as a result of the propagation module design. The propagation module of SAT-Accel has a minimum latency of $34+\lceil\frac{n}{4}\rceil$ cycles where n is the number of clauses a variable is in. If a variable does not cause any or very few clauses to become unit, this operation is relatively costly. This cost is usually amortized if there are many variables already in the pipeline. We see an average of 40 cycles across our benchmark.

6.3.2 Low Overhead Memory Management. SAT-Accel's memory management system spends a small fraction of the overall solving time doing memory management operations and these are marked as **Allocate** and **Delete** in Table 5. For the SAT competition instances, SAT-Accel spent on average less than 2.5% doing memory management operations while other types spent less than 1%.

We repeated the experiment using a 90% clause deletion rate to see if our memory management system is robust. As shown in Table 5, there is an insignificant increase during allocation at 90% prune rate. The time spent in deletion did increase, with the increase of at-most roughly 1%.

7 Conclusion and Future Work

In this work, we propose SAT-Accel, an algorithm-harware co-design SAT solver that implements many of the core features used in today's SAT solvers. We successfully overcame the challenges described previously. For managing the data structure, SAT-Accel utilizes a novel page-based memory management system that is able to allocate and reclaim pages efficiently. SAT-Accel replaces the 2 Watched Literal scheme, that is friendly towards CPUs, with a friendly-toward FPGAs alternative. SAT-Accel limits the effects of the sequential nature of the algorithm by maximizing fine grain and task parellelism throughout the design. Compared to MiniSat, Kissat, and SAT-Hard, we are able to achieve on average a 17.9x, 2.8x, and 800.0x speedup respectively with ASIC in parentheses.

Future work will focus on increasing capacity. We have identified a few places where the use of on-chip memory can be improved. First, the clause store uses on-chip to store the instance and the learned clauses. However, it is not along the critical path of the design as clause learning and minimization does not take up most of the time. Therefore, the clause store could utilize HBM at the expense of its initial latency. Second, the original-to-transposed and transposed-to-original position array used for deletion can be removed. The performance will be impacted since the value to be removed needs to be searched. However, we expect that the time lost will not be significant, as demonstrated with an aggressive deletion rate in Table 5.

A total a number of 384 URAM instances will be freed, which is roughly half of the design's current usage. We can use these URAMs to support more variables, clauses, and increase the size of the transposed array. To see how much of an impact this has, we estimate the number of instances from the 2023 SAT competition that could be loaded onto our design. If we increase the supported number of variables and clauses to 128k and 1024k respectively, the number of instances increases from 205 to 270 out of 401. However, place and route becomes challenging as the transposed array crosses multiple SLRs, impacting the frequency. To be able to migrate the transposed array to HBM requires major architectural revisions. As stated earlier, the propagation latency has a minimum of 34 cycles, adding HBM latency would push it to 100 cycles. In addition, propagation memory access pattern is random as it is not known which variables to check before-hand. One possible solution, and is being explored, is to create a cache tailored for SAT solving.

Acknowledgment and Disclosure

This work is partially supported by the CDSC industrial partners (https://cdsc.ucla.edu/partners/), especially Cadence and TSMC, and the PRISM Center under the SRC/DARPA JUMP 2.0 Program. The authors would also like to thank AMD/Xilinx for donating the HACC computing cluster. J. Cong has a financial interest in AMD and TSMC.

References

[1] [n. d.] (). http://www.satcompetition.org/.

[2] [n. d.] (). https://satcompetition.github.io/2022/results.html.

[3] [n. d.] (). https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html.

[4] Miron Abramovici, Jose T de Sousa, and Daniel Saab. 1999. A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, 684–690.

[5] John Ahlgren and Shiu Yin Yuen. 2013. Efficient program synthesis using constraint satisfaction in inductive logic programming. *The Journal of Machine Learning Research*, 14, 1, 3649–3682.

[6] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. 2009. On the structure of industrial sat instances. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 127–141.

[7] Mikhail Asiatici and Paolo Ienne. 2021. Large-scale graph processing on FPGAs with caches for thousands of simultaneous misses. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 609–622. DOI: 10.1109/ISCA52012.2021.00054.

[8] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. 2002. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In *International Conference on Automated Deduction*. Springer, 195–210.

[9] Gilles Audemard and Laurent Simon. 2009. Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, 7–8.

[10] John Backes, Brian Fett, and Marc D. Riedel. 2008. The analysis of cyclic circuits with boolean satisfiability. In *2008 IEEE/ACM International Conference on Computer-Aided Design*, 143–148. DOI: 10.1109/ICCAD.2008.4681565.

[11] Suhail Basalama, Atefeh Sohrabizadeh, Jie Wang, Licheng Guo, and Jason Cong. 2023. FlexCNN: an end-to-end framework for composing CNN accelerators on FPGA. *ACM Trans. Reconfigurable Technol. Syst.*, 16, 2, Article 23, (Mar. 2023), 32 pages. DOI: 10.1145/3570928.

[12] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions* (Department of Computer Science Report Series B). Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, (Eds.) Vol. B-2020-1. University of Helsinki, 51–53.

[13] Armin Biere and Andreas Frohlich. 2019. Evaluating CDCL restart schemes. In *Proceedings of Pragmatics of SAT 2015 and 2018* (EPiC Series in Computing). Daniel Le Berre and Matti Jarvisalo, (Eds.) Vol. 59. EasyChair, 1–17. DOI: 10.29007/89dw.

[14] Armin Biere and Marijn Heule. 2019. The effect of scrambling CNFs. *Proceedings of Pragmatics of SAT*, 59, 111–126.

[15] Mau-Chung Frank Chang, Yu-Ting Chen, Jason Cong, Po-Tsang Huang, Chun-Liang Kuo, and Cody Hao Yu. 2016. The SMEM seeding acceleration for DNA sequence alignment. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 32–39. DOI: 10.1109/FCCM.2016.21.

[16] Yuze Chi, Weikang Qiao, Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. 2022. Democratizing domain-specific computing. *Commun. ACM*, 66, 1, (Dec. 2022), 74–85. DOI: 10.1145/3524108.

[17] Young-Kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2019. In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms. *ACM Trans. Reconfigurable Technol. Syst.*, 12, 1, Article 4, (Feb. 2019), 20 pages. DOI: 10.1145/3294054.

[18] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS today: successes, challenges, and opportunities. *ACM Trans. Reconfigurable Technol. Syst.*, 15, 4, Article 51, (Aug. 2022), 42 pages. DOI: 10.1145/3530775.

[19] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (STOC '71). Association for Computing Machinery, Shaker Heights, Ohio, USA, 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047.

[20] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM*, 63, 7, (June 2020), 48–57. DOI: 10.1145/3361682.

[21] Martin Davis, George Logemann, and Donald W. Loveland. 1962. A machine program for theorem-proving. In *Communications of the ACM 5*.

[22] Martin Davis and Hilary Putnam. 1960. A computing procedure for quantification theory. In *Journal of the ACM 7*.

[23] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9, 5, 256–268. DOI: 10.1109/JSSC.1974.1050511.

[24] Ghada Dessouky, Michael J. Klaiber, Donald G. Bailey, and Sven Simon. 2014. Adaptive dynamic on-chip memory management for fpga-based reconfigurable architectures. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 1–8. DOI: 10.1109/FPL.2014.6927471.

[25] Tim Dunn, Harisankar Sadasivan, Jack Wadden, Kush Goliya, Kuan-Yu Chen, David Blaauw, Reetuparna Das, and Satish Narayanasamy. 2021. SquiggleFilter: an accelerator for portable virus detection. In (MICRO '21). Association for Computing Machinery, Virtual Event, Greece, 535–549. ISBN: 9781450385572. DOI: 10.1145/3466752.3480117.

[26] Niklas Eén and Niklas Sörensson. 2004. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*. Enrico Giunchiglia and Armando Tacchella, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 502–518. ISBN: 978-3-540-24605-3.

[27] Johannes K. Fichte, Markus Hecher, and Stefan Szeider. 2020. A time leap challenge for SAT-solving. In *Principles and Practice of Constraint Programming*. Helmut Simonis, (Ed.) Springer International Publishing, Cham, 267–285. ISBN: 978-3-030-58475-7.

[28] Jeremy Fowers et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 1–14. DOI: 10.1109/ISCA.2018.00012.

[29] Henri Fraisse, Abhishek Joshi, Dinesh Gaitonde, and Alireza Kaviani. 2016. Boolean satisfiability-based routing and its application to Xilinx ultrascale clock network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (FPGA '16). Association for Computing Machinery, Monterey, California, USA, 74–79. ISBN: 9781450338561. DOI: 10.1145/2847263.2847342.

[30] Nicholas V. Giamblanco and Jason H. Anderson. 2019. A dynamic memory allocation library for high-level synthesis. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 314–320. DOI: 10.1109/FPL.2019.00057.

[31] Yijin Guan, Guangyu Sun, Zhihang Yuan, Xingchen Li, Ningyi Xu, Shu Chen, Jason Cong, and Yuan Xie. 2020. Crane: mitigating accelerator under-utilization caused by sparsity irregularities in CNNs. *IEEE Transactions on Computers*, 69, 7, 931–943. DOI: 10.1109/TC.2020.2981080.

[32] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. 2017. FPGA-based accelerator for long short-term memory recurrent neural networks. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 629–634. DOI: 10.1109/ASPDAC.2017.7858394.

[33] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: coupling coarse-grained floorplanning and pipelining for high-frequency hls design on multi-die fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 81–92.

[34] Leopold Haller and Satnam Singh. 2010. Relieving capacity limits on FPGA-based SAT-solvers. In *Formal Methods in Computer Aided Design*, 217–220.

[35] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture* (ISCA '16). IEEE Press, Seoul, Republic of Korea, 243–254. ISBN: 9781467389471. DOI: 10.1109/ISCA.2016.30.

[36] Song Han et al. 2017. ESE: efficient speech recognition engine with sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (FPGA '17). Association for Computing Machinery, Monterey, California, USA, 75–84. ISBN: 9781450343541. DOI: 10.1145/3020078.3021745.

[37] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. 2021. SPAGHETTI: streaming accelerators for highly sparse GEMM on FPGAs. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 84–96. DOI: 10.1109/HPCA51647.2021.00017.

[38] Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger Hoos, and Kevin Leyton-Brown. 2017. The configurable sat solver challenge (cssc). *Artificial Intelligence*, 243, 1–25. DOI: https://doi.org/10.1016/j.artint.2016.09.006.

[39] Hadi Katebi, Karem A. Sakallah, and Joao P. Marques-Silva. 2011. Empirical study of the anatomy of modern SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2011*. Karem A. Sakallah and Laurent Simon, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 343–356. ISBN: 978-3-642-21581-0.

[40] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2023. Tseitin or not tseitin? the impact of cnf transformations on feature-model analyses. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (ASE '22) Article 110. Association for Computing Machinery, , Rochester, MI, USA, 13 pages. ISBN: 9781450394758. DOI: 10.1145/3551349.3556938.

[41] Tracy Larrabee. 1992. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11, 1, 4–15.

[42] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. 2020. Heterorefactor: refactoring for heterogeneous computing with fpga. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (ICSE '20). Association for Computing Machinery, Seoul, South Korea, 493–505. ISBN: 9781450371216. DOI: 10.1145/3377811.3380340.

[43] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. 2015. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. (2015). arXiv: 1506.08905 [cs.LO].

[44] Wan-Hsuan Lin, Jason Kimko, Bochen Tan, Nikolaj Bjørner, and Jason Cong. 2023. Scalable optimal layout synthesis for NISQ quantum processors. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 1–6. DOI: 10.1109/DAC56929.2023.10247760.

[45] Wan-Hsuan Lin, Bochen Tan, Murphy Yuezhen Niu, Jason Kimko, and Jason Cong. 2022. Domain-specific quantum architecture optimization. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 12, 3, 624–637. DOI: 10.1109/JETCAS.2022.3202870.

[46] Michael Lo, Zhenman Fang, Jie Wang, Peipei Zhou, Mau-Chung Frank Chang, and Jason Cong. 2020. Algorithm-hardware co-design for BQSR acceleration in genome analysis toolkit. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 157–166. DOI: 10.1109/FCCM48280.2020.00029.

[47] J.P. Marques-Silva and K.A. Sakallah. 1999. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48, 5, 506–521. DOI: 10.1109/12.769433.

[48] Armin Biere, Marijn Heule, Hans Van Maaren, and Toby Walsh, (Eds.) 2021. *Chapter 4: conflict-driven clause learning sat solvers*. English (US). *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications. Publisher Copyright: © 2021 The authors and IOS Press. All rights reserved. IOS Press BV, 133–182. DOI: 10.3233/FAIA200987.

[49] João P. Marques-Silva and Karem A. Sakallah. 2000. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Design Automation Conference* (DAC '00). Association for Computing Machinery, Los Angeles, California, USA, 675–680. ISBN: 1581131879. DOI: 10.1145/337292.337611.

[50] Fabio Massacci and Laura Marraro. 2000. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24, 165–203.

[51] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 530–535. DOI: 10.1145/378239.379017.

[52] Gi-Joon Nam, Fadi Aloul, Karem Sakallah, and Rob Rutenbar. 2001. A comparative study of two boolean formulations of FPGA detailed routing constraints. In *Proceedings of the 2001 International Symposium on Physical Design* (ISPD '01). Association for Computing Machinery, Sonoma, California, USA, 222–227. ISBN: 1581133472. DOI: 10.1145/369691.369777.

[53] Gi-Joon Nam, K.A. Sakallah, and R.A. Rutenbar. 2002. A new FPGA detailed routing approach via search-based boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21, 6, 674–684. DOI: 10.1109/TCAD.2002.1004311.

[54] Gi-Joon Nam, Karem A. Sakallah, and Rob A. Rutenbar. 1999. Satisfiability-based layout revisited: detailed routing of complex FPGAs via search-based boolean SAT. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays* (FPGA '99). Association for Computing Machinery, Monterey, California, USA, 167–175. ISBN: 1581130880. DOI: 10.1145/296399.296450.

[55] Soowang Park, Jae-Won Nam, and Sandeep K. Gupta. 2021. Hw-bcp: a custom hardware accelerator for sat suitable for single chip implementation for large benchmarks. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 29–34.

[56] Knot Pipatsrisawat and Adnan Darwiche. 2007. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing–SAT 2007: 10th International Conference, Lisbon, Portugal, May 28-31, 2007. Proceedings 10*. Springer, 294–299.

[57] Neha Prakriya, Yu Yang, Baharan Mirzasoleiman, Cho-Jui Hsieh, and Jason Cong. 2023. NeSSA: near-storage data selection for accelerated machine learning training. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems* (HotStorage '23). Association for Computing Machinery, Boston, MA, USA, 8–15. ISBN: 9798400702242. DOI: 10.1145/3599691.3603404.

[58] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2018. High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 37–44. DOI: 10.1109/FCCM.2018.00015.

[59] Mona Safar, M. Watheq El-Kharashi, Mohamed Shalan, and Ashraf Salem. 2011. A reconfigurable, pipelined, conflict directed jumping search SAT solver. In *2011 Design, Automation & Test in Europe*, 1–6. DOI: 10.1109/DATE.2011.5763199.

[60] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. 2020. Bonsai: high-performance adaptive merge tree sorting. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 282–294. DOI: 10.1109/ISCA45697.2020.00033.

[61] Roberto Sebastiani and Michele Vescovi. 2009. Automated reasoning in modal and description logics via SAT encoding: the case study of k (m)/alc-satisfiability. *Journal of Artificial Intelligence Research*, 35, 343–389.

[62] Yakun Sophia Shao et al. 2019. Simba: scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO '52). Association for Computing Machinery, Columbus, OH, USA, 14–27. ISBN: 9781450369381. DOI: 10.1145/3352460.3358302.

[63] Ashish Sirasao, Elliott Delaye, Ravi Sunkavalli, and Stephen Neuendorffer. 2015. FPGA based opencl acceleration of genome sequencing software. *System*, 128, 8.7, 11.

[64] A. Smith, A. Veneris, M.F. Ali, and A. Viglas. 2005. Fault diagnosis and logic debugging using boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24, 10, 1606–1621. DOI: 10.1109/TCAD.2005.852031.

[65] Niklas Sorensson and Armin Biere. 2009. Minimizing learned clauses. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing* (SAT '09). Springer-Verlag, Swansea, UK, 237–243. ISBN: 9783642027765. DOI: 10.1007/978-3-642-02777-2_23.

[66] Saurabh Srivastava. 2010. *Satisfiability-based program reasoning and program synthesis*. Ph.D. Dissertation.

[67] Bharat Sukhwani, Thomas Roewer, Charles L. Haymes, Kyu-Hyoun Kim, Adam J. McPadden, Daniel M. Dreps, Dean Sanner, Jan Van Lunteren, and Sameh Asaad. 2017. Contutto – a novel FPGA-based prototyping platform enabling innovation in the memory subsystem of a server class processor. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 15–26.

[68] Bochen Tan and Jason Cong. 2022. Layout synthesis for near-term quantum computing: gap analysis and optimal solution. In *Design Automation of Quantum Computers*. Springer, 25–40.

[69] Bochen Tan and Jason Cong. 2020. Optimal layout synthesis for quantum computing. In *Proceedings of the 39th International Conference on Computer-Aided Design* (ICCAD '20) Article 137. Association for Computing Machinery, Virtual Event, USA, 9 pages. ISBN: 9781450380263. DOI: 10.1145/3400302.3415620.

[70] Bochen Tan and Jason Cong. 2021. Optimality study of existing quantum computing layout synthesis tools. *IEEE Transactions on Computers*, 70, 9, 1363–1373. DOI: 10.1109/TC.2020.3009140.

[71] Jason Thong and Nicola Nicolici. 2013. FPGA acceleration of enhanced boolean constraint propagation for SAT solvers. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 234–241. DOI: 10.1109/ICCAD.2013.6691124.

[72] Buse Ustaoglu, Sebastian Huhn, Daniel Große, and Rolf Drechsler. 2018. Satlancer: a hardware sat-solver for self-verification. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI* (GLSVLSI '18). Association for Computing Machinery, Chicago, IL, USA, 479–482. ISBN: 9781450357241. DOI: 10.1145/3194554.3194643.

[73] Buse Ustaoglu, Sebastian Huhn, Frank Sill Torres, Daniel Große, and Rolf Drechsler. 2019. SAT-Hard: a learning-based hardware sat-solver. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, 74–81. DOI: 10.1109/DSD.2019.00021.

[74] Xuechao Wei, Yun Liang, Xiuhong Li, Cody Hao Yu, Peng Zhang, and Jason Cong. 2018. TGPA: tile-grained pipeline architecture for low latency CNN inference. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1–8. DOI: 10.1145/3240765.3240856.

[75] R Glenn Wood and Rob A Rutenbar. 1997. FPGA routing and routability estimation via boolean satisfiability. In *Proceedings of the 1997 ACM fifth international symposium on Field-programmable gate arrays*, 119–125.

[76] Lisa Wu et al. 2019. FPGA accelerated indel realignment in the cloud. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 277–290. DOI: 10.1109/HPCA.2019.00044.

[77] Zhongda Yuan, Yuchun Ma, and Jinian Bian. 2012. SMPP: generic SAT solver over reconfigurable hardware accelerator. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 443–448. DOI: 10.1109/IPDPSW.2012.57.

[78] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. 2016. Energy-efficient CNN implementation on a deeply pipelined FPGA cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design* (ISLPED '16). Association for Computing Machinery, San Francisco Airport, CA, USA, 326–331. ISBN: 9781450341851. DOI: 10.1145/2934583.2934644.

[79] Jiaxi Zhang, Wentai Zhang, Guojie Luo, Xuechao Wei, Yun Liang, and Jason Cong. 2019. Frequency improvement of systolic array-based CNNs on FPGAs. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1–4. DOI: 10.1109/ISCAS.2019.8702071.

[80] Lintao Zhang, C.F. Madigan, M.H. Moskewicz, and S. Malik. 2001. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, 279–285. DOI: 10.1109/ICCAD.2001.968634.

[81] Peixin Zhong, Pranav Ashar, Sharad Malik, and Margaret Martonosi. 1998. Using reconfigurable computing techniques to accelerate problems in the cad

domain: a case study with boolean satisfiability. In *Proceedings of the 35th annual Design Automation Conference*, 194–199.

[82] Chenzhuo Zhu, Alexander C. Rucker, Yawen Wang, and William J. Dally. 2023. Satin: hardware for boolean satisfiability inference. (2023). https://arxiv.org/abs/2303.02588 arXiv: 2303.02588 [cs.AR].

[83] Yilan Zhu, Xinyao Wang, Lei Ju, and Shanqing Guo. 2023. FxHENN: FPGA-based acceleration framework for homomorphic encrypted CNN inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 896–907. DOI: 10.1109/HPCA56546.2023.10071133.