Quantum Circuit Mapping Based on Incremental and Parallel SAT Solving

Jiong Yang
National University of Singapore, Singapore

Yaroslav A. Kharkov
AWS Quantum Technologies, New York, NY, USA

Yunong Shi
AWS Quantum Technologies, New York, NY, USA

Marijn J. H. Heule
Carnegie Mellon University, Pittsburgh, PA, USA
Amazon Web Services, Seattle, WA, USA

Bruno Dutertre
Amazon Web Services, Santa Clara, CA, USA

Abstract

Quantum Computing (QC) is a new computational paradigm that promises significant speedup over classical computing in various domains. However, near-term QC faces numerous challenges, including limited qubit connectivity and noisy quantum operations. To address the qubit connectivity constraint, circuit mapping is required for executing quantum circuits on quantum computers. This process involves performing initial qubit placement and using the quantum SWAP operations to relocate non-adjacent qubits for nearest-neighbor interaction. Reducing the SWAP count in circuit mapping is essential for improving the success rate of quantum circuit execution as SWAPs are costly and error-prone. In this work, we introduce a novel circuit mapping method by combining incremental and parallel solving for Boolean Satisfiability (SAT). We present an innovative SAT encoding for circuit mapping problems, which significantly improves solver-based mapping methods and provides a smooth trade-off between compilation quality and compilation time. Through comprehensive benchmarking of 78 instances covering 3 quantum algorithms on 2 distinct quantum computer topologies, we demonstrate that our method is 26× faster than state-of-the-art solver-based methods, reducing the compilation time from hours to minutes for important quantum applications. Our method also surpasses the existing heuristics algorithm by 26% in SWAP count.

2012 ACM Subject Classification Software and its engineering → Compilers; Computer systems organization → Quantum computing; Computing methodologies → Artificial intelligence

Keywords and phrases Quantum computing, Quantum compilation, SAT solving, Incremental solving, Parallel solving

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.29

Funding *Marijn J. H. Heule*: Supported by the NSF under grant CCF-2229099.

Acknowledgements Part of the research was conducted during Jiong Yang's internship at Amazon Web Services. We thank Eric Kessler and Soonho Kong for the insightful discussions on this work. We are grateful to the anonymous reviewers for their constructive comments to improve this paper.

1 Introduction

There is compelling evidence that Quantum Computing (QC) can solve certain computational problems exponentially more efficiently than classical computers [33, 39]. As a result, transformative applications are expected to emerge in fields such as optimization [1, 11, 32], machine learning [8, 38], finance [10, 18, 21, 34], pharmaceuticals [13, 16, 24], and cryptography [17, 39, 40].

However, the practical realization of QC still faces numerous challenges, including limited qubit connectivity on quantum computers and noisy quantum operations. Limited qubit connectivity decreases the ability of quantum devices to execute arbitrary quantum circuits, and noisy operations restrict the sizes of executable quantum circuits. Fortunately, these hardware challenges can be mitigated at the software level by compiler optimizations.

Quantum compilers perform numerous transformations and optimizations to produce compact and optimized circuit executables. The specific transformation we are concerned with in this paper is *circuit mapping*. Circuit mapping involves the insertion of a special quantum operation called the SWAP gate to map arbitrary quantum circuits to devices. Since SWAP gates are costly and error-prone, the compiler must minimize the SWAP count. Currently, there are two primary approaches to the circuit mapping problem: solver-based algorithms [28, 44] and heuristics-based algorithms [26, 41]. Both approaches have their drawbacks: solver-based algorithms achieve optimal SWAP count but suffer from long compilation time; heuristic algorithms are fast, but the SWAP counts are usually suboptimal.

We propose a novel circuit mapping method based on incremental and parallel solving for Boolean Satisfiability (SAT). Our approach aims to find a minimum number of SWAP gates that accommodate the circuit mapping requirement by iteratively decreasing the SWAP-gate count and checking feasibility with SAT solving. We use a dedicated SAT encoding that enables incremental solving and we combine incremental and parallel solving techniques. Compared to current solver-based algorithms, our method is 26x faster on average. Compared to current heuristic algorithms, our method reduces the SWAP count by 26% on average.

In summary, our contributions are:

1. We design a novel SAT encoding for determining the satisfiability of mapping a circuit with a given SWAP count.
2. By combining the novel SAT encoding and parameter search developed, we develop a new circuit mapping method that achieves a smooth trade-off between compilation quality (in terms of SWAP count) and compilation time.
3. By exploiting the problem structure, we develop an efficient implementation of the proposed mapping method that combines incremental and parallel techniques.
4. We perform an extensive evaluation to show that the resulting approach is 26× faster than the state-of-the-art solver-based method and outperforms the heuristic approaches in 76% of the instances.

In the rest of this paper, Section 2 introduces the background on quantum circuit mapping and preliminaries on SAT solving. We then present our SAT-based quantum circuit mapper in Section 3 and give details on the full encoding in Section 4. Finally, we present our experimental evaluation in Section 5 and conclude in Section 6.

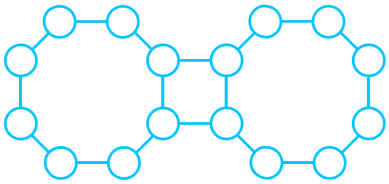
2 Background

2.1 Quantum Circuit Mapping

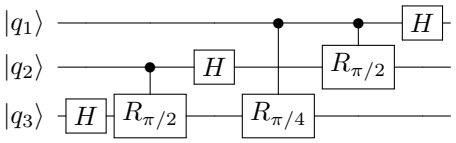
We first illustrate the problem of quantum circuit mapping with an example. Subsequently, we delve into the existing approaches addressing this problem.

2.1.1 An Illustrative Example

Figure 1a shows the qubit connectivity of (part of) the Aspen M-3 quantum computer, manufactured by Rigetti Computing [36]. Figure 1b shows the circuit diagram of a 3-qubit quantum circuit for the famous Quantum Fourier Transform (QFT) algorithm [14, 33]. The QFT circuit has a 2-qubit gate between each pair of the three qubits.



(a) Qubit connectivity graph of (part of) the Rigetti Aspen M-3 quantum computer. Circles are physical qubits and lines are physical links that allow 2-qubit gates to be performed on.

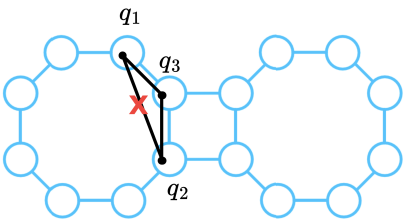


(b) An example 3-qubit QFT circuit. The 3 horizontal lines represent the time schedules of the quantum gates on the 3 algorithmic qubits q_1, q_2, and q_3. This circuit consists of 1-qubit Hadamard gates (boxes labeled H) and 2-qubit controlled rotation gates (solid dot and box labeled R that are connected by a vertical line). In our problem setting, the two qubits in controlled rotation gates can be treated equally.

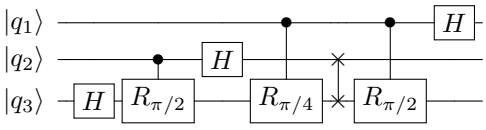
Figure 1 The mismatch between qubit connectivity of the Rigetti Aspen M-3 hardware and that of the QFT circuit. On the Aspen M-3 device in Figure 1a, no three qubits are connected to each other. In the QFT circuit in Figure 1b, each qubit is connected to the other two by a 2-qubit gate.

To execute the QFT circuit on the Aspen M-3 device, a *circuit mapping* procedure must be performed by a quantum compiler. Circuit mapping involves two steps: initial qubit placement and qubit routing. During initial qubit placement, the quantum compiler maps each algorithmic qubit in the circuit to a physical qubit on the device, as shown in Figure 2a.

The QFT circuit requires each algorithmic qubit to interact with the other two. However, on the qubit connectivity graph of Aspen M-3, no subgraph forms a three-qubit ring that would allow pairwise qubit interaction. As a result, after initial qubit placement, the QFT circuit still cannot be directly executed since it requires a non-local 2-qubit interaction that is not supported by the device (see Figure 2a for a specific initial assignment). This gap necessitates the second step in circuit mapping – *qubit routing*. When two (algorithmic) qubit operands in a 2-qubit gate are mapped to non-adjacent physical qubits, the compiler performs qubit routing to remap the two qubit operands to adjacent physical qubits before scheduling the 2-qubit gate.



(a) The initial qubit placement step associates each algorithmic qubit in the QFT circuit with a physical qubit on the Aspen M-3 device. Black lines are connections required by the QFT circuit in Figure 1b. The line with red cross is not available on the device.



(b) The qubit routing step inserts a SWAP gate (denoted as two crosses connected by a vertical line) in the QFT circuit. After the SWAP insertion, the last 2-qubit gate is now between q_1 and q_3 (it was between q_1 and q_2, as shown in Figure 1b).

Figure 2 Mapping the 3-qubit QFT circuit in Figure 1b onto the Aspen M-3 device in Figure 1a. First, algorithmic qubits are placed onto the device (Figure 2a). The last 2-qubit gate between q_1 and q_2 cannot be scheduled as there is no link between them on the device.. After the SWAP insertion (Figure 2b), the last 2-qubit gate is re-targeted to q_1 and q_3, which then can be scheduled.

Qubit routing is performed by inserting quantum SWAP gates [33]. A SWAP gate is a special 2-qubit quantum gate that is not responsible for entangling qubit states for computation, but for exchanging the qubit states for routing. Since a SWAP gate exchanges qubit states, it affects the quantum gates scheduled behind it. After a SWAP insertion, the gates scheduled after the SWAP gate that use one of the swapped qubits must be re-targeted to the other qubit that it swaps to. Figure 2b gives an example of SWAP insertion for the 3-qubit QFT circuit from Figure 1b. From the example, we can see that SWAP gates can alter the connectivity requirements of the quantum circuits to match them with the qubit connectivity of the underlying quantum hardware.

2.1.2 Current Circuit Mapping Approaches

Currently, there are two main-stream approaches to the circuit mapping problem:

Heuristic Algorithms. Heuristic mapping algorithms usually optimize metrics designed by humans and calculable within a bounded search depth. Examples of these metrics include the total 2-qubit gate distance of the 2-qubit gates that remain t to be scheduled (the gate distance of a 2-qubit gate measures how far away the two qubit operands are on the device given the current mapping). In industrial quantum compilers, heuristic algorithms are among the most popular choices because they provide fast compilation time. Notable examples include the SABRE (SWAP-based BidiREctional heuristic search) algorithm [26] used in the Qiskit compiler [2] and the architecture-aware mapping algorithm in the TKET compiler [41]. Optimality studies have shown that heuristic algorithms are far from the theoretical optimal in terms of the output SWAP count [43].

Solver-based Algorithms. Alternatives to the heuristic methods rely on complete algorithms to search for the minimal SWAP count. Existing work employs different types of solvers for this purpose. The state-of-the-art solver-based algorithm is the TB-OLSQ2 mapper [28], which translates the mapping problem to a satisfiability problem that can be solved by the Z3 [15] SMT solver. SATMap [31] is another solver-based algorithm that is based on MaxSAT solving but SATMap is not as efficient as TB-OLSQ2 due to a different choice of encoding.

2.2 Modern SAT Solving

Let $\{x_1, x_2, ..., x_n\}$ be a set of Boolean variables. A literal l is a variable x or its negation $\neg x$. A clause C of size k is a disjunction of k literals, i.e., $C = (l_1 \vee l_2 \vee ... \vee l_k)$. A formula φ in Conjunctive Normal Formal (CNF) is a conjunction of clauses, i.e., $\varphi = (C_1 \wedge C_2 \wedge ... \wedge C_m)$ for some $m > 0$. An assignment of truth values to the variables in φ is called a *solution* if it makes φ evaluate to true. We call φ *satisfiable* if there exists a solution and *unsatisfiable* otherwise. Given a formula φ, modern SAT solvers can effectively find a solution or prove that it is unsatisfiable by using the Conflict Driven Clause Learning (CDCL) scheme [29]. Many efficient solver implementations have emerged in recent years, including but not limited to Kissat, CaDiCaL [9], CryptoMiniSat [42], MapleSAT [27], and Glucose [5].

Incremental Solving. Incremental solving is an effective technique supported by most modern SAT solvers for solving a series of similar formulas. Incremental solving supports the addition of new clauses and assumptions between SAT calls. A key benefit is that the solver can keep and reuse internal states and clauses learned in previous SAT calls to speed up solving for the new formula.

Parallel Solving. Parallel solving seeks to distribute the computational workload of solving a formula across multiple processors. There are primarily two categories of work. The *partition* approaches try to split the formula equally into many sub-problems using heuristics and solve them in parallel [4, 20, 23, 45]. Recent work in this line includes cube-and-conquer [23], AmPharoS [4], and Paracooba [20]. Another line of work, called the*portfolio* approach, runs multiple SAT solvers with different configurations in parallel to solve the original formula and share information, such as, learned clauses between solvers. These approaches mainly leverage the solver diversity to improve the overall solving speed. Portfolio approaches are implemented by competition-winning solvers such as ManySAT [19], Mallob [37], and ParKissat-RS [7].

2.3 Helper Functions

In this section, we introduce some of the helper functions that we use in our approach.

2.3.1 Linear Encoding of At-Most-One Constraints

Several constraints in our SAT formulation are *At-Most-One* (AMO) constraints. We leverage a recursive scheme (also used in other works [22]), as a general helper function for encoding AMO constraints in our SAT formulation. Each step of the recursion introduces a new Boolean variable y. To encode the AMO constraint for a general set of Boolean variables $B = \{b_1, ..., b_i, ..., b_n\}$, we have,

$$\mathrm{AMO}(b_1, \ldots, b_n) = \begin{cases} \bigwedge_{1 \le i < j \le n} (\neg b_i \vee \neg b_j) & n \le 4 \\ \mathrm{AMO}(b_1, b_2, b_3, y) \wedge \mathrm{AMO}(\neg y, b_4, \ldots, b_n) & \text{otherwise} \end{cases}$$

For a constraint with n variables, this encoding introduces $\frac{n-3}{2}$ auxiliary variables and $3(n-2)$ clauses compared to $\frac{n^2-n}{2}$ clauses using a naive pairwise encoding.

2.3.2 Totalizer Encoding

Our approach utilizes a cardinality constraint with an iteratively decreasing cardinality bound. We hence describe the Totalizer encoding [6, 30] for cardinality constraints with varying bounds. Below we present an example to encode the AtMostK constraint ($l_1+l_2+l_3+l_4+l_5 \le k$). Figure 3 illustrates the encoding structure where every node is represented by a node name and a list of variables. The leaf node indicates an input variable, e.g., node C with a variable l_1. The root node includes the indicator variables for the sum of input variables. For example, assigning variable o_2 in node O indicates that at least two variables of $l_1, ..., l_5$ are true, while a false assignment to o_2 indicates that at most one input variable is assigned to true. Every internal node represents an intermediate sum over its two children. For example, node A represents the sum of variables from nodes C and D. The variables a_1 and a_2 indicate whether the sum is at least one and two, respectively.

Following the structure, we can derive the encoding below. For any non-leaf node $P : p_1, ..., p_{n_p}$ with two children denoted by $Q : q_1, ..., q_{n_q}$ and $R : r_1, ..., r_{n_r}$, we require at least $\alpha + \beta$ variables of P to be true when node Q implies α many variables assigned to true and node R indicates β many variables to be true, i.e.,

$$\begin{aligned} &(q_\alpha \wedge r_\beta) \rightarrow p_\gamma \\ &\iff \neg q_\alpha \vee \neg r_\beta \vee p_\gamma \quad \text{for } \alpha + \beta = \gamma,\ 0 \le \alpha \le n_q,\ 0 \le \beta \le n_r,\ 0 \le \gamma \le n_p \end{aligned}$$

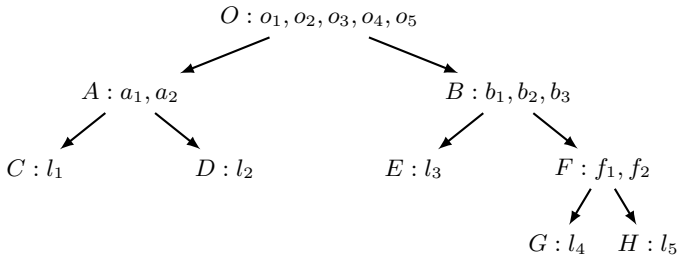


Figure 3 Totalizer encoding for $l_1 + l_2 + l_3 + l_4 + l_5 \leq k$.

where $q_0 = r_0 = p_0 = \text{true}$. For the cardinality bound, we simply add a unit clause

$$\neg o_{k+1}$$

for AtMostK constraint or add o_k for AtLeastK constraint. In our application, we are interested in AtMostK constraints with iteratively decreasing bounds. Since $\neg o_{k_1}$ implies $\neg o_{k_2}$ for $k_1 < k_2$, we can add the unit clause for a smaller bound without deleting the previous one, which is called *incremental strengthening* in the literature [3].

For a cardinality constraint with n variables and bound k, the Totalizer encoding requires $\mathcal{O}(n \log n)$ auxiliary variables and $\mathcal{O}(nk)$ clauses after simplification [12, 25].

3 SAT-based Circuit Mapping

Our novel quantum circuit mapper is called SATmapper. It is depicted in Figure 4. SATmapper seeks to find a minimum number of SWAP gates that accommodate the quantum circuit mapping requirement by iteratively decreasing the SWAP-gate count (S) and checking feasibility for S using a modern SAT solver. Section 3.1 provides an overview of the SATmapper framework. We then discuss the encoding and decoding processes in Section 3.2. We discuss the solving techniques used by SATmapper in Section 3.3.

3.1 Framework

Figure 4 presents the framework of SATmapper. For a given number of SWAP count S, SATmapper reduces the mapping problem to a SAT encoding and utilizes a SAT solver to compute the feasibility of using no mote than S SWAP gates. The workflow runs from left to right. Given three inputs: a quantum circuit, a quantum device (QPU), and an initial SWAP count (S), SATmapper encodes the quantum circuit mapping problem into a SAT formula in conjunctive normal form (CNF). A SAT solver takes the CNF as input and checks its satisfiability. A satisfiable (SAT) result indicates that there is a valid mapping that uses no more than S SWAP gates. In this case, we reduce the SWAP count S and continue the loop to search for a mapping with fewer SWAP gates. We exit the loop when the solver returns UNSAT, which indicates that we cannot decrease the SWAP count. Finally, we decode a mapped circuit from the best result we've obtained so far.

Pseudo-code for this procedure is given in Algorithm 1 . Given a quantum circuit and a device. we initialize S with the value produced by the state-of-the-art heuristic approach, TKET. If TKET generates a mapping without any SWAP gates, we terminate and return the mapping at Line 3. In the common case, TKET provides a mapping with $S(> 0)$ SWAP gates. Then, we start our SAT-based optimization from this S at Lines 4-11.

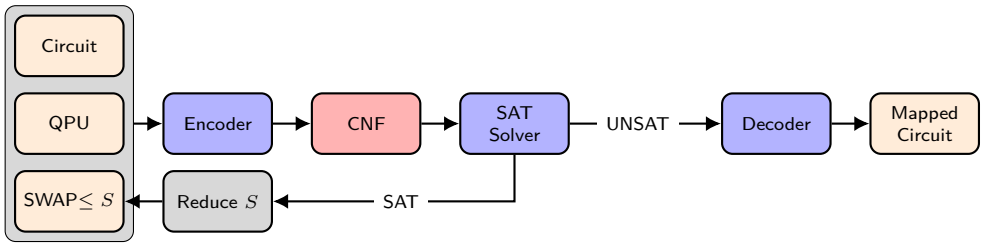


Figure 4 SATmapper framework.

Algorithm 1 SATmapper(circuit, qpu).

```
1:  S ← TKET(circuit, qpu);
2:  if S = 0 then
3:      return mapped circuit from TKET;                    ▷ It's already optimal.
4:  repeat
5:      S ← S − 1;
6:      φ ← Encode(circuit, qpu, S);
7:      sat ← Solve(φ);
8:      if sat is True then
9:          solution ← GetSolution(φ);                      ▷ Store the best result so far.
10:         S ← CountSWAP(solution);          ▷ Calculate the SWAP count in the solution.
11: until (sat is False) or (S = 0) or Timeout;
12: if solution is NULL then
13:     return mapped circuit from TKET;          ▷ No better result from SATmapper.
14: else
15:     mappedCircuit ← Decode(solution, circuit, qpu);
16:     return mappedCircuit;              ▷ Retrieve the best result within the time limit.
```

In every iteration, we reduce the best-known S by one to explore the possibility of producing a lower SWAP count. We encode the circuit mapping with an upper bound of S SWAPs into a formula φ at Line 6 and invoke a SAT solver to solve φ at Line 7. If φ is satisfiable, the circuit can be mapped with no more than S SWAP gates and we store the solution at Line 9. The solution contains the best mapping up to this point and can later be decoded into a mapped circuit. We calculate the actual SWAP count implied by the solution as this SWAP count can be smaller than the current upper bound S. Therefore, we update the best-known SWAP count S with the actual value implied by the solution at Line 10. We repeat the loop until the solver can't find a better mapping, or S is zero, or we exceed the time limit. If we can't find a single solution in the loop, we return the initial mapping from TKET at Line 13. Otherwise, a solution was stored and we decode at Line 15 a mapped circuit from this solution, which represents the mapping with the smallest SWAP count obtained within the time limit.

3.2 Encoding Sketch

This section presents the key encoding idea for mapping a quantum circuit to a target device using a given number of SWAP gates. We introduce the core variables that encode the basic information for mapping circuits in Section 3.2.1. Next, Section 3.2.2 presents the encoding of SWAP constraints which has a crucial impact on our system design in the following sections.

Other variables and constraints capture the mapping interaction between the circuit and SWAP gates, and we defer the details of full encoding to Section 4. Finally, Section 3.2.3 illustrates how to decode the mapped circuit from a solution to our encoding.

3.2.1 Transition Step

We consider the circuit mapping in different temporal steps. Instead of the actual time step in the circuit that increase gate by gate, we define a *transition* step that increases only after SWAP gate insertion. The layout of a quantum device changes when the transition step increases. Each layout update event is encoded as a new time step. Therefore, we start with an initial layout at step 0 and move to step 1 after inserting SWAP gates to update the layout. Given a qubit connectivity graph of a quantum device, we use a Boolean variable c_k^t for every positive step t and edge k to indicate whether the edge k is selected to perform a SWAP at step t or not. c_k^t is assigned to true if the edge is selected to perform a SWAP and is assigned to false otherwise.

For every gate g and step t, we use a Boolean variable o_g^t to indicate whether the gate g has been scheduled by step t (including t) or not. If o_g^t is assigned to true, gate g is scheduled at step i with $0 \leq i \leq t$ and is assigned to false otherwise. For example, $o_g^1 =$ True indicates that the gate g has been scheduled by step 1, that is, scheduled at either step 0 or step 1. Otherwise, o_g^1 is false and the gate g has to be scheduled at step $t > 1$.

3.2.2 SWAP Constraints

The constraint on SWAP count is crucial to our design of SATmapper as it is the only varying constraint across iterations. Suppose we consider T transition steps and K edges of the connectivity graph. During the transition, multiple SWAPs can be scheduled simultaneously if there are no conflicts among them. For every step $t > 0$, adjacent edges can't be selected for SWAP at the same time because they share a common vertex, For any two edges k and k' sharing a common vertex, we assert then

$$\neg c_k^t \vee \neg c_{k'}^t \text{ for } t \in [1, T]$$

The total number of SWAPs over T steps is at most S, i.e.,

$$\sum_{t=1}^{T} \sum_{k=1}^{K} c_k^t \leq S \tag{1}$$

where c_k^t indicates whether the edge k is selected to perform SWAP at step t. The cardinality constraint (Equation 1) can be encoded into clauses using the Totalizer encoding detailed in Section 2.3.2.

3.2.3 Mapped Circuit Decoding

Algorithm 2 describes how to decode a mapped circuit from a solution to our encoding. We first decode an initial qubit placement from the solution, that is, the mapping from each algorithmic qubit to a physical qubit at step 0. Then, we retrieve the number of transition steps from the heuristic approach, TKET. Starting from Line 3, we decode the mapped circuit step by step. We use a list l initialized at Line 4 to store the gates to be scheduled in sequence. Except for the initial step ($t = 0$), we retrieve all SWAP gates at step t and schedule them at Line 8 to update the layout. Since there is no conflict between the SWAP

Algorithm 2 Decode(solution, circuit, qpu).

```
1: mappedCircuit ← GetInitPlacement(solution);
2: T ← TKETstep(circuit, qpu);
3: for t ← 0 to T do
4:     l ← an empty list;
5:     if t > 0 then
6:         for c_k^t from solution do
7:             if c_k^t is True then
8:                 mappedCircuit adds a SWAP gate on edge k;
9:     for o_g^t from solution do
10:        if o_g^t is True then
11:            for g' ← the last to the first of l do
12:                if g depends on g' in circuit then
13:                    l inserts g after g';
14:                    break
15:            if g ∉ l then
16:                l inserts g at the beginning;
17:    for g ∈ l do
18:        mappedCircuit adds g;
19: return mappedCircuit;
```

gates at the same step, they can be scheduled in any order. Following that, we retrieve other gates scheduled at step t one by one starting from Line 9. For a gate g, we traverse the list l in a *reverse* order (Line 11) to insert g after the last gate g' that g depends on at Line 13, which ensures g is scheduled after all dependent gates. If g has no dependent gates in l, we insert g at the beginning of l at Line 16. After that, l contains a sequence of gates where $\forall g_i, g_j \in l$ with $i < j$, g_i doesn't depend on g_j and can be safely scheduled before g_j. Finally, we schedule every gate in l in sequence at Line 18 and return the mapped circuit at Line 19.

3.3 Solving Techniques

Any modern SAT solver can serve to solve the encoded CNF in Figure 4, such as Kissat, CaDiCaL [9], and the like. To embrace the recent advance in parallel (cloud) SAT solving, we can use a competition-winning parallel SAT solver instead like Mallob [37] or ParKissat-RS [7]. Furthermore, Section 3.3.1 presents the application of incremental solving techniques to SATmapper. Finally, we introduce the combination of incremental and parallel solving in Section 3.3.2 to benefit SATmapper from the best of both worlds.

3.3.1 Incremental Solving

Incremental solving is beneficial in solving a series of similar input formulas. For SATmapper, only the AtMostK cardinality constraint on the SWAP count (Equation 1) varies over iterations. According to Section 2.3.2, the Totalizer encoding of the AtMostK constraint allows us to use a list of variables o_i to indicate different upper bounds for the same encoding. In every iteration of SATmapper, we can run an incremental SAT solver, for example, CaDiCaL, with a different assumption. For example, we can impose Equation 1 in our encoding by adding the assumption of $\neg o_{S+1}$ to the SAT solver, where $\neg o_{S+1}$ indicates that there won't be $S+1$ input variables to be true, that is, at most S SWAP gates are allowed in the mapping. When SATmapper moves to a lower SWAP count S' with $S' < S$, we add another assumption $\neg o_{S'+1}$ to indicate the new bound of no more than S' SWAPs.

As a consequence, we can avoid encoding and reading the whole CNF at every iteration on line 6 and 7 of Algorithm 1. Instead, the solver uses a new assumption literal and continues to solve the formula. This incremental solving allows the solver to start from the previous internal state and reuse learned clauses from previous iterations, which effectively reduces the overall runtime across iterations.

3.3.2 Incremental Parallel Solving

Given the application of incremental or parallel solving to SATmapper individually, one would wonder whether SATmapper can benefit from the best of both worlds. This section gives an affirmative answer to this question by presenting an incremental and parallel SAT solver, IncParKissat.

We have extended the state-of-the-art parallel SAT solver ParKissat-RS to support incremental solving. Figure 5 depicts the framework of IncParKissat. IncParKissat takes an original CNF as input and initially forks multiple Kissat with different configurations to run in parallel. Each Kissat keeps an individual copy of the input CNF and maintains its clause database throughout running. During the solving process, different solvers communicate with each other by sending and receiving important learned clauses to facilitate the global solving. When a solver finds a solution, a global termination signal is sent to the other solvers, and all solver threads join the main thread. Alternatively, if a solver proves that the input CNF is UNSAT, it terminates and joins the main thread. In either case, we output the result into the main thread and continue the incremental solving by forking multiple solver threads again. Every solver adds additional clauses φ to their database and continues to solve the input formula from the previous internal state.

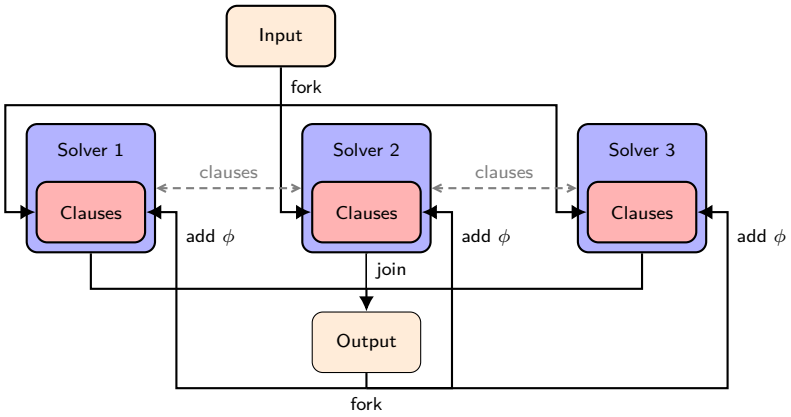


Figure 5 Framework of incremental and parallel SAT solver, IncParKissat.

The underlying solver Kissat used by IncParKissat does not currently support solving with assumptions, but our application, assumptions are not required. Our implementation allows IncParKissat to add extra clauses across incremental iterations. Because SATmapper only requires a cardinality constraint with a decreasing upper bound across iterations, the ability to add a unit clause across iterations is sufficient. This is because $\neg o_{k_1}$ implies $\neg o_{k_2}$ for $k_1 < k_2$ in the Totalizer encoding, so we can add the unit clause for a smaller bound without deleting the previous one, as described in Section 2.3.2. IncParKissat could be extended to support solving with assumptions by replacing the underlying SAT solver Kissat with CaDiCaL.

4 Full SAT Encoding

We present the full SAT encoding for the quantum circuit mapping problem in this section. Our encoding uses a *transition*-based step instead of the actual time steps in the circuit. After SWAP insertion, the layout of the quantum device is updated and we call the layout update a *transition*. Each layout transition event will be encoded as a new time step. The decision of whether a gate has been scheduled by a layout transition or not is encoded as a Boolean variable.

As follows, Section 4.1 introduces the formal definition for input and Section 4.2 discusses the pre-processing steps. We define the encoding variables in Section 4.3 and constraints in Section 4.4. We analyze the asymptotic encoding size in Section 4.5.

4.1 Problem Input

There are three inputs to the SAT formulation: (1) The quantum circuit to be mapped, which is represented as an ordered list of n 2-qubit[1] quantum gates $G = \{g_1, ..., g_i, ..., g_n\}$ and V algorithmic qubits; (2) The qubit connectivity graph of the quantum device, including P physical qubits and K connecting edges e_{ij} for $i, j \in [1, P]$ and $i \neq j$; (3) the desired physical SWAP gate number S.

4.2 Pre-processing

We perform two simple initialization steps as pre-processing of the SAT encoding:

- **Initialization of the input qubit connectivity graph.** We initialize an ordered list $E = \{d_1, ..., d_k, ..., d_K\}$ of edges, where each d_k uniquely corresponds to an edge $e_{i_k j_k}$ in the input qubit connectivity graph, and K is the total number of edges in the connectivity graph. The order of edges e_{ij} in E can be chosen arbitrarily. For each d_k, we denote the larger qubit index it connects as $d_k.op_{\max}$ and the smaller qubit index as $d_k.op_{\min}$.
- **Initialization of gate dependency list.** For g_i in G, we denote the larger gate operand as $g_i.op_{\max}$ and the smaller operand as $g_i.op_{\min}$ and initialize them in an array. Both $g_i.op_{\max}$ and $g_i.op_{\min}$ are fixed for $i \in [1, n]$. Further, we initialize the gate dependency list $l_g = \{(g_1^1, g_2^1), ..., (g_1^i, g_2^i), ...\}$, each pair of gates denotes that g_2^i is dependent on g_1^i and thus g_2^i cannot be scheduled before g_1^i. l_g can be generated by enumerating the gates on each qubit.

4.3 Encoding Variables

- Schedule $o_{g_i}^t$: if gate g_i has been scheduled in step t, $t \in [0, T]$, then $o_{g_i}^t$ = True, otherwise, $o_{g_i}^t$ = False. 2-qubit gates can only be scheduled after their algorithmic qubit operands are mapped to connected physical qubits on the connectivity graph. We set the number of transition steps T to be the same as the output of the heuristic approach, TKET.
- Layout π_{ij}^t: After step t (and before step $t+1$ if $t < T$), if qubit i is mapped to qubit j, then π_{ij}^t = True, else π_{ij}^t = False. Here $i \in [1, V], j \in [1, P], t \in [0, T]$.
- SWAP operand selection c_k^t: if edge $k \in [1, K]$ is selected for performing a SWAP at step $t \in [1, T]$, then c_k^t = True, otherwise, c_k^t = False. There are no SWAPs in the initial step, when $t = 0$.

[1] We don't consider 1-qubit gates because they can always be scheduled.

4.4 Constraints

- **Gate schedule initialization.** For step $t \in [0, T)$ and gate $g_i \in G$, it's impossible that g_i is scheduled after step t, but not scheduled after step $t+1$, i.e.,

$$\begin{aligned} &\neg(o_{g_i}^t \wedge \neg o_{g_i}^{t+1}) \\ \Longleftrightarrow &\neg o_{g_i}^t \vee o_{g_i}^{t+1} \text{ for } i \in [1, n],\ t \in [0, T) \end{aligned}$$

  by de Morgan's law. Also, all gates should be scheduled after step T, i.e.,

$$o_{g_i}^T = \text{True for } i \in [1, n]$$

- **Gate dependency.** For $(g_i, g_j) \in l_g$, g_j cannot be scheduled before g_i:

$$\neg(\neg o_{g_i}^t \wedge o_{g_j}^t) \text{ for } (g_i, g_j) \in l_g, \text{ and } t \in [0, T)$$

  similarly, by de Morgan's law, it can be re-written as

$$o_{g_i}^t \vee \neg o_{g_j}^t \text{ for } (g_i, g_j) \in l_g, \text{ and } t \in [0, T)$$

- **Gate schedule continuation.** For all $i \in [1, n]$ and $t \in [1, T]$, if gate g_i is not scheduled after step $t-1$ and its qubit operands are not mapped to connected physical qubits after step t, then g_i will not be scheduled after step t.

$$\left(\neg o_{g_i}^{t-1} \wedge \bigwedge_{(j,l)\in E} \neg\left((\pi_{g_i.op_{\max} j}^t \wedge \pi_{g_i.op_{\min} l}^t) \vee (\pi_{g_i.op_{\max} l}^t \wedge \pi_{g_i.op_{\min} j}^t)\right)\right) \rightarrow \neg o_{g_i}^t$$

  for $i \in [1, n]$, and $t \in [1, T]$, which can be re-written as

$$o_{g_i}^{t-1} \vee \neg o_{g_i}^t \vee \bigvee_{(j,l)\in E} (\pi_{g_i.op_{\max} j}^t \wedge \pi_{g_i.op_{\min} l}^t) \vee (\pi_{g_i.op_{\max} l}^t \wedge \pi_{g_i.op_{\min} j}^t)$$

  where the big disjunction can be resolved by introducing auxiliary variables. For $t = 0$,

$$\neg o_{g_i}^0 \vee \bigvee_{(j,l)\in E} (\pi_{g_i.op_{\max} j}^0 \wedge \pi_{g_i.op_{\min} l}^0) \vee (\pi_{g_i.op_{\max} l}^0 \wedge \pi_{g_i.op_{\min} j}^0) \text{ for } i \in [1, n]$$

- **Bounded SWAP selection.** For every step $t > 0$, adjacent edges can't be selected for SWAP at the same time to avoid simultaneous SWAPs on the same qubit, i.e., for any two edges d_k and $d_{k'}$ sharing a common vertex,

$$\neg c_k^t \vee \neg c_{k'}^t \text{ for } t \in [1, T], k, k' \in [1, K]$$

  The total number of SWAPs over T steps is at most S, i.e.,

$$\sum_{t=1}^{T} \sum_{k=1}^{K} c_k^t \leq S \tag{2}$$

  where the cardinality constraint (Equation 2) can be encoded into clauses using the Totalizer encoding detailed in Section 2.3.2.

- **Bijective layout mapping.** The layout mapping π_{ij}^t after each step t is required to be injective, i.e.,

$$\begin{aligned} &\text{AMO}(\{\pi_{ij}^t\}_{j=1}^P) \text{ for } i \in [1, V], \text{ and } t \in [0, T] \\ &\text{AMO}(\{\pi_{ij}^t\}_{i=1}^V) \text{ for } j \in [1, P], \text{ and } t \in [0, T] \end{aligned}$$

- **Layout mapping update.** After each step $t > 0$, π_{ij}^{t-1} will be updated to π_{ij}^{t}. If a SWAP in step t is performed on the k-th edge d_k in E (w.l.o.g., we assume d_k stores edge e_{ij}, i.e., the vertices i and j are endpoints of edge k.), then we have the following: (1) if an algorithmic qubit m is mapped to physical qubit i after step $t-1$ (i.e., $\pi_{mi}^{t-1} = \text{True}$), then $\pi_{mj}^{t} = \text{True}$:

$$\begin{aligned} & (c_k^t \wedge \pi_{mi}^{t-1}) \rightarrow \pi_{mj}^{t} \quad \text{for } m \in [1,V], k \in [1,K], t \in [1,T] \\ \Rightarrow \quad & \neg c_k^t \vee \neg \pi_{mi}^{t-1} \vee \pi_{mj}^{t} \quad \text{for } m \in [1,V], k \in [1,K], t \in [1,T] \end{aligned}$$

(2) similarly, if an algorithmic qubit m is mapped to physical qubit j after step $t-1$, then $\pi_{mi}^{t} =$True,

$$\neg c_k^t \vee \neg \pi_{mj}^{t-1} \vee \pi_{mi}^{t} \quad \text{for } m \in [1,V], k \in [1,K], t \in [1,T]$$

(3) if an algorithmic qubit m is not mapped to i, j, then the qubit mapping after step $t-1$ and after step t are the same,

$$\neg c_k^t \vee \neg \pi_{ml}^{t-1} \vee \pi_{ml}^{t} \quad \text{for } m \in [1,V], l \in [1,P], l \neq i, l \neq j, k \in [1,K], t \in [1,T]$$

4.5 Complexity Analysis

Theorem 1 states the asymptotic number of variables and clauses introduced by our encoding. The encoding size increases approximately linearly with the number of transition steps.

▶ **Theorem 1.** *Given a quantum circuit of n gates, a quantum device of k edges in its connectivity graph, and t transition steps with s SWAP gates, the* SATmapper *encoding requires $\mathcal{O}\left(tk(n + \log tk)\right)$ variables and $\mathcal{O}\left(tk(k + s + n^2)\right)$ clauses.*

Proof. Assume the quantum circuit has v algorithmic qubits and the device has p physical qubits. We first consider the number of variables. We have $\mathcal{O}(tn + tvp + tk)$ original variables introduced in Section 4.3. The gate schedule continuation introduces $\mathcal{O}(tnk)$ auxiliary variables for resolving the big disjunction. The bounded SWAP selection introduces $\mathcal{O}(tk \log tk)$ auxiliary variables for the cardinality constraint. The bijective layout mapping introduces $\mathcal{O}(tpv)$ auxiliary variables for AMO constraints. As a result, we have $\mathcal{O}(tvp + tnk + tk \log tk) \subseteq \mathcal{O}\left(tk(n + \log tk)\right)$ variables in total, assuming, w.l.o.g., $v \in \mathcal{O}(n)$ and $p \in \mathcal{O}(k)$.

For the number of clauses, the gate schedule initialization requires $\mathcal{O}(tn)$ clauses and the gate dependency constraints produces $\mathcal{O}(tn^2)$ clauses. The gate schedule continuation has $\mathcal{O}(tnk)$ original clauses and $\mathcal{O}(tnk)$ auxiliary clauses introduced to resolve the big disjunction. The bounded SWAP selection requires $\mathcal{O}(tk^2)$ clauses for non-conflict selection and $\mathcal{O}(tks)$ auxiliary clauses for encoding the cardinality constraint. Finally, the bijective layout mapping introduces $\mathcal{O}(tvp)$ clauses to encode AMO constraints, and the layout mapping update produces $\mathcal{O}(tkv^2)$ clauses. In total, we have $\mathcal{O}(tn^2 + tnk + tk^2 + tks + tvp + tkv^2) \subseteq \mathcal{O}\left(tk(k + s + n^2)\right)$ clauses assuming $v \in \mathcal{O}(n)$ and $p \in \mathcal{O}(k)$. ◀

5 Experimental Evaluation

To evaluate our approach, we have implemented a prototype of SATmapper and compared it with the state-of-the-art solver-based method, TB-OLSQ2, and the best heuristic approach,TKET. We have conducted a comprehensive evaluation on seventy-eight instances ranging over two quantum computer devices, OQC Lucy [35] and Rigetti Aspen M-3 [36], and

three quantum algorithms including QAOA ansatz for k-regular graphs, Quantum Fourier Transform (QFT), and Quantum Volume (QV) circuits, with various numbers of algorithmic qubits. All experiments were conducted on Amazon EC-2 instances c6a.48xlarge featuring 192 CPUs.

We used a time limit of 1800 seconds per instance. We set 16 CPUs and recorded the wall-clock time for parallel solvers. We set the step number T to be the same as the output of TKET. The runtime of TKET is less than one second on the given instances. We don't include this time since it is negligible compared to the solving time. The Rigetti Aspen M-3 and OQC Lucy devices contain 32 and 8 physical qubits, respectively, and the number of algorithmic qubits to be mapped cannot be more than the available number of physical qubits. We define the following three metrics to measure the performance of SWAP count optimization for SATmapper and TB-OLSQ2. Lower is better for all metrics and both SATmapper and TB-OLSQ2 start with an initial upper bound obtained by TKET.

- **Failure Ratio.** The ratio of instances where no improvement against TKET is observed.
- **SWAP Ratio.** The ratio of the best SWAP count to that of TKET on average.
- **Median Runtime.** The median runtime across all instances.

Particularly, we aim to address the following questions:

RQ1. Is SATmapper able to outperform TB-OLSQ2?

RQ2. What is the best underlying solver for SATmapper?

Summary. SATmapper outperformed TB-OLSQ2 on all three metrics, achieving a reduction of 57 percent for failure ratio and six percent for SWAP ratio, and lowering the median runtime from 237 to 9 seconds. Additionally, IncParKissat is the best-performing underlying solver for SATmapper by combining incremental and parallel solving.

5.1 RQ1. SATmapper vs. TB-OLSQ2

Table 1 summarizes the comparison between TB-OLSQ2 and SATmapper (equipped with IncParKissat) through the three metrics and note that the lower is better for all metrics. The first row presents the failure ratio, where TB-OLSQ2 had no improvement on 81% instances while SATmapper only failed on 24% instances achieving a reduction of 57 percent. In the second row, TB-OLSQ2 attained a SWAP ratio of 0.80 on average while SATmapper lowered the ratio to 0.74 by an improvement of six percent. Finally, the median runtime of TB-OLSQ2 is 237 seconds across all instances. As a comparison, SATmapper median runtime is only 9 seconds, which achieves a 26-fold reduction compared to TB-OLSQ2.

Table 1 Performance comparison between TB-OLSQ2 and SATmapper.

Metric	TB-OLSQ2	SATmapper
Failure Ratio	0.81	**0.24**
SWAP Ratio	0.80	**0.74**
Median Runtime/s	237	**9**

Table 2 presents a detailed comparison on a subset of instances. The first column gives the instance name in a format that lists device name, algorithm name, and number of algorithmic qubits. The next three columns compare the number of SWAPs optimized by TKET, TB-OLSQ2, and SATmapper. On easy instances with fewer than ten algorithmic qubits, both TB-OLSQ2 and SATmapper can improve the results against TKET and SATmapper can

Table 2 SWAP optimization comparison on a subset of instances.

	SWAP Count			Runtime/s	
Instance	TKET	TB-OLSQ2	SATmapper	TB-OLSQ2	SATmapper
Aspen-qaoa-3reg-n-8	5	3	3	1.97	3.32
Aspen-qaoa-3reg-n-10	6	4	4	1.67	2.37
Aspen-qaoa-3reg-n-12	19	8	7	116.85	1732.92
Aspen-qaoa-3reg-n-14	11	8	8	192.78	5.33
Aspen-qaoa-3reg-n-16	17	12	11	925.10	414.73
Aspen-qft-n-8	18	13	12	201.95	45.80
Aspen-qft-n-9	24	19	16	793.87	447.15
Aspen-qft-n-10	37	Failed	24	Failed	609.81
Aspen-qft-n-11	58	Failed	34	Failed	1335.53
Aspen-qft-n-12	55	Failed	40	Failed	1651.05
Aspen-qv-n-8	11	10	8	89.19	18.43
Aspen-qv-n-10	47	Failed	27	Failed	1640.08
Aspen-qv-n-12	56	Failed	50	Failed	513.64
OQC-qaoa-3reg-n-8	7	7	7	0.17	1.12
OQC-qft-n-8	28	23	23	14.15	4.44
OQC-qv-n-8	17	13	13	3.03	5.59

find lower SWAP counts than TB-OLSQ2. On hard instances of at least ten algorithmic qubits, TB-OLSQ2 failed to produce any better result for the QFT and QV instances while SATmapper still improved the results of TKET by around 15 SWAPs.

The last two columns of Table 2 compare the runtime. SATmapper is slightly slower on easy instances that TB-OLSQ2 could solve within a few seconds because of a constant overhead, but SATmapper is consistently faster than TB-OLSQ2 on harder instances. We still observe that SATmapper spent a significantly larger time than TB-OLSQ2 on some instances, for example, Aspen-qaoa-3reg-n-12. This is because SATmapper tried to find a lower SWAP count than TB-OLSQ2, which considerably increases the difficulty. If SATmapper stopped at eight SWAPs on this instance, the total runtime would be twenty seconds only, which is lower than the runtime of TB-OLSQ2, but the effort to find a mapping of seven SWAPs took the remaining 1712 seconds. The example also revealed the drastically increased difficulty when lowering the SWAP count. It's worth highlighting that SATmapper even used less time to produce a lower SWAP count than TB-OLSQ2 on many instances such as Aspen-qaoa-3reg-n-16, Aspen-qft-n-8, and Aspen-qv-n-8.

5.2 RQ2. Underlying Solver for SATmapper

Table 3 compares the performance of SATmapper when using different underlying SAT solvers. Specifically, we aim to benchmark the performance of incremental and parallel-solving techniques for SATmapper. Every column indicates SATmapper equipped with a particular solver. Kissat [9] is a state-of-the-art sequential SAT solver without incremental and parallel-solving techniques. CaDiCaL refers to using the CaDiCaL SAT solver [9] in incremental mode. Pbop is a non-incremental, clause-sharing parallel SAT solver based on Kissat. IncParKissat is the incremental and parallel SAT solver described in Section 3.3.2. Table 3 indicates that SATmapper with IncParKissat achieved the best performance on all three metrics. Particularly, both CaDiCaL and Pbop outperformed Kissat, which reveals the

Table 3 Performance comparison for SATmapper with different underlying solvers.

Metric	Kissat	CaDiCaL	Pbop	IncParKissat
Failure Ratio	0.27	0.26	0.25	**0.24**
SWAP Ratio	0.78	0.78	0.76	**0.74**
Median Runtime/s	183	137	18	**9**

individual benefits gained from incremental and parallel solving, respectively. Finally, the combination of incremental and parallel solving allows IncParKissat to benefit from the best of both worlds.

6 Conclusion

SWAP count optimization during quantum circuit compilation is critical to deploying quantum algorithms on current generation quantum devices. The existing solver-based method to address this problem does not scale well, but the fast heuristic approach tends to produce low-quality results. Our approach, based on SAT solving, outperforms the existing solver-based approach. It scales better and can produced smaller SWAP counts. It also produce higher-quality quantum circuits that the best heuristics methods. We implement the SWAP count optimization as a series of calls to a SAT solver. We introduced a novel SAT encoding, and developed an efficient implementation by combining incremental and parallel-solving techniques. A comprehensive evaluation on real-world quantum algorithms and devices demonstrates that our method is 26× faster than the existing solver-based approach and produces better results. Our method also improved on the heuristic approach on 76% of instances and achieved an average of 26% reduction in SWAP count.

References

1 Amira Abbas, Andris Ambainis, Brandon Augustino, Andreas Bärtschi, Harry Buhrman, Carleton Coffrin, Giorgio Cortiana, Vedran Dunjko, Daniel J Egger, Bruce G Elmegreen, et al. Quantum optimization: Potential, challenges, and the path forward. *arXiv preprint*, 2023. arXiv:2312.02279.

2 MD Sajid Anis, Héctor Abraham, R Agarwal AduOffei, Gabriele Agliardi, Merav Aharoni, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, M Amy, S Anagolum, et al. Qiskit: An open-source framework for quantum computing (2021). *SUPPLEMENTARY INFORMATION I. ALGORITHMS II. A RELAXATION BOUND (| E| 2+ 1 9 α| E| 2+ α) Remark*, 2021.

3 Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *Proc. of SAT*, 2009.

4 Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. An adaptive parallel sat solver. In *Proc. of CP*, 2016.

5 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proc. of IJCAI*, 2009.

6 Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *Proc. of CP*, 2003.

7 Tomas Balyo, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki, 2022.

8 Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 2017.

9 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, 2020.

10 Adam Bouland, Wim van Dam, Hamed Joorati, Iordanis Kerenidis, and Anupam Prakash. Prospects and challenges of quantum finance. *arXiv preprint*, 2020. `arXiv:2011.06492`.

11 Fernando G. S. L. Brandão, Amir Kalev, Tongyang Li, Cedric Yen-Yu Lin, Krysta M. Svore, and Xiaodi Wu. Quantum SDP Solvers: Large Speed-ups, Optimality, and Applications to Quantum Learning, 2019.

12 Markus Büttner and Jussi Rintanen. Satisfiability planning with constraints on the number of actions. In *Proc. of ICAPS*, 2005.

13 Yudong Cao, Jonathan Romero, Jonathan P. Olson, Matthias Degroote, Peter D. Johnson, Mária Kieferová, Ian D. Kivlichan, Tim Menke, Borja Peropadre, Nicolas P. D. Sawaya, Sukin Sim, Libor Veis, and Alán Aspuru-Guzik. Quantum chemistry in the age of quantum computing. *Chemical Reviews*, 2019.

14 Don Coppersmith. An approximate fourier transform useful in quantum factoring. *arXiv preprint quant-ph/0201067*, 2002.

15 Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proc. of TACAS*, 2008.

16 Richard P. Feynman. Simulating Physics with Computers. *International Journal of Theoretical Physics*, 1982.

17 Craig Gidney and Martin Ekerå. How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *Quantum*, 2021.

18 Andrés Gómez, Álvaro Leitao, Alberto Manzano, Daniele Musso, María R. Nogueiras, Gustavo Ordóñez, and Carlos Vázquez. A Survey on Quantum Computational Finance for Derivatives Pricing and VaR. *Archives of Computational Methods in Engineering*, 2022.

19 Youssefa Hamadi, Saidb Jabbour, and Lakhdarb Sais. Manysat: a parallel sat solver issue title: Parallel sat solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 2009.

20 Maximilian Heisinger, Mathias Fleury, and Armin Biere. Distributed cube and conquer with paracooba. In *Proc. of SAT*, 2020.

21 Dylan Herman, Cody Googin, Xiaoyuan Liu, Alexey Galda, Ilya Safro, Yue Sun, Marco Pistoia, and Yuri Alexeev. A survey of quantum computing for finance. *arXiv preprint*, 2022. `arXiv:2201.02773`.

22 Marijn J. H. Heule. Chinese remainder encoding for hamiltonian cycles. In *Proc. of SAT*, 2021.

23 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Hardware and Software: Verification and Testing*, 2012.

24 Tomi H. Johnson, Stephen R. Clark, and Dieter Jaksch. What is a quantum simulator? *EPJ Quantum Technology*, 2014.

25 Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat: A partial max-sat solver. *J. Satisf. Boolean Model. Comput.*, 2012.

26 Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for nisq-era quantum devices. In *Proc. of ASPLOS*, 2019.

27 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Proc. of SAT*, 2016.

28 Wan-Hsuan Lin, Jason Kimko, Bochen Tan, Nikolaj Bjørner, and Jason Cong. Scalable optimal layout synthesis for nisq quantum processors. In *Proc. of DAC*, 2023.

29 J.P. Marques Silva and K.A. Sakallah. Conflict analysis in search algorithms for satisfiability. In *Proc. of ICTAI*, 1996.

30 Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for maxsat. In *Proc. of CP*, 2014.

31 A. Molavi, A. Xu, M. Diges, L. Pick, S. Tannu, and A. Albarghouthi. Qubit mapping and routing via maxsat. In *Proc. of MICRO*, 2022.

32 Michele Mosca, Joao Marcos Vensi Basso, and Sebastian R. Verschoor. On speeding up factoring with quantum SAT solvers. *Scientific Reports*, 2020.

33 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

34 Román Orús, Samuel Mugel, and Enrique Lizaso. Quantum computing for finance: Overview and prospects. *Reviews in Physics*, 2019.

35 Oxford quantum circuits. `https://aws.amazon.com/braket/quantum-computers/oqc/`. Accessed: 2024-03-16.

36 Rigetti computing. `https://www.rigetti.com/`. Accessed: 2024-02-20.

37 Dominik Schreiber and Peter Sanders. Scalable sat solving in the cloud. In *Proc. of SAT*, 2021.

38 Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. An introduction to quantum machine learning. *Contemporary Physics*, 2015.

39 Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 1997.

40 P.W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proc. of FOCS*, 1994.

41 Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩: a retargetable compiler for nisq devices. *Quantum Science and Technology*, 2020.

42 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. of SAT*, 2009.

43 Bochen Tan and Jason Cong. Optimality study of existing quantum computing layout synthesis tools. *IEEE Transactions on Computers*, 2020.

44 Bochen Tan and Jason Cong. Optimal qubit mapping with simultaneous gate absorption. In *Proc. of ICCAD*, 2021.

45 HANTAO ZHANG, MARIA PAOLA BONACINA, and JIEH HSIANG. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 1996.