TopoLS: Lattice Surgery Compilation via Topological Program Transformations

Junyu Zhou
University of Pennsylvania
Philadelphia, USA
junyuzh@seas.upenn.edu

Yuhao Liu
University of Pennsylvania
Philadelphia, USA
liuyuhao@seas.upenn.edu

Ethan Decker
University of Pennsylvania
Philadelphia, USA
ecd5249@upenn.edu

Justin Kalloor
UC Berkeley
Berkeley, USA
jkalloor3@berkeley.edu

Mathias Weiden
UC Berkeley
Berkeley, USA
mtweiden@berkeley.edu

Kean Chen
University of Pennsylvania
Philadelphia, USA
keanchen@seas.upenn.edu

Costin Iancu
Lawrence Berkeley National Laboratory
Berkeley, USA
cciancu@lbl.gov

Gushu Li
University of Pennsylvania
Philadelphia, USA
gushuli@seas.upenn.edu

Abstract

Fault-tolerant quantum computing with surface codes can be achieved by compiling logical circuits into lattice-surgery instructions. To minimize space–time volume, we present TopoLS, a topological compiler that combines ZX-diagram optimizations with Monte Carlo tree search guided by different operation placements and topology-aware circuit partitioning. Our approach enables scalable exploration of lattice surgery structures and consistently reduces resource overhead. Evaluations of various benchmark algorithms across multiple architectures show that TopoLS achieves an average 33% reduction in space-time volume over prior heuristic-based compilers, while maintaining linear compilation time scaling. Compared to the SAT-solver-based compiler, which provides optimal results only for small circuits before becoming intractable, TopoLS offers an effective and scalable solution for lattice-surgery compilation.

1 Introduction

Quantum error correction (QEC) is widely recognized as a cornerstone for achieving scalable and reliable quantum computation [4]. Unlike classical systems, quantum devices suffer from decoherence [23], relaxation [3], and measurement errors [9], all of which accumulate rapidly as circuits grow in depth and complexity. Without effective error correction, even modestly sized computations would quickly become unreliable. Among the various codes that have been proposed, the surface code [8] stands out due to its high error threshold, locality of interactions, and compatibility with two-dimensional hardware layouts. These properties make it one of the most promising candidates for building large-scale, fault-tolerant quantum computers and for realizing significant quantum algorithms [12]. Recent experiments have demonstrated essential components of surface code operation [1, 19], further supporting its practical viability.

Despite its advantages, the surface code introduces fundamental challenges when it comes to implementing logical operations. In devices with nearest-neighbor connectivity, such as superconducting qubits [18] and spin qubits [10], a major challenge is the lack of transversal logical two-qubit gate support. To overcome this limitation, researchers have developed alternative techniques, with lattice surgery [14] emerging as one of the most promising methods. Lattice surgery enables logical operations by merging and splitting code patches in a controlled manner, effectively performing entangling operations at the logical level without relying on transversal gates. Lattice surgery offers a practical method for implementing logical operations, but it comes with significant resource overhead. In particular, it imposes constraints on qubit layout as well as the careful coordination of measurement sequences. The physical cost of lattice surgery is most naturally quantified in terms of its space–time volume [7, 13], the product of the spatial area occupied by qubits and the time required to perform the operations.

Minimizing this space–time volume is therefore essential for reducing overhead and improving the efficiency of fault-tolerant quantum computation. Compared with the conventional circuit model, lattice surgery exhibits several distinctive properties. On the one hand, its fundamental operations are merges and splits, which differ entirely from the standard gate-based representation of quantum operations. On the other hand, lattice surgery admits a natural graphical description through the ZX-calculus [5], where the merges and splits correspond to ZX spiders, and only the topological connectivity of the diagram is relevant. As highlighted by this topological perspective, the design of a lattice-surgery

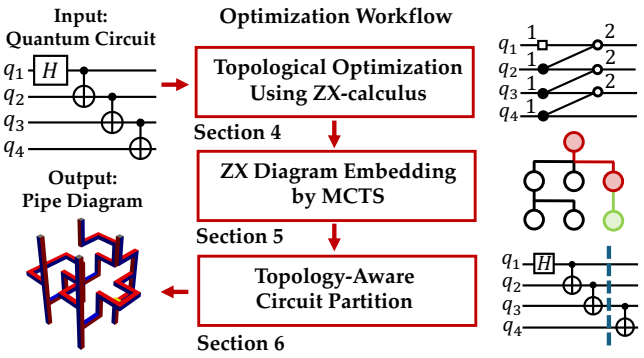

Figure 1. Overview of TopoLS workflow

schedule can be approached with greater flexibility. One may reorganize the coordination of qubits and operations while ensuring that the underlying topological connections are preserved.

Existing scalable compilers [20, 28] for lattice surgery remain far from optimal, as they primarily adopt gate-based perspectives that fail to leverage the unique properties of lattice surgery. These approaches emphasize qubit routing and gate scheduling, yet overlook that merge and split operations are the true execution primitives—operations that can be more effectively optimized within a topological framework such as ZX-calculus. On the other hand, formal methods like SAT-solver-based compiler [26] encode spatial constraints more directly, but they suffer from limited scalability and restricted applicability. Consequently, current compilers either miss key opportunities to reduce space–time volume through topological simplification, or they cannot scale to larger problem sizes,

To this end, we propose TopoLS, an efficient lattice surgery compiler that fully exploits the topological nature of quantum computation. Our approach integrates the ZX-calculus representation with Monte Carlo Tree Search (MCTS): the ZX diagram preserves the high-level topological structure of the quantum program, while MCTS searches for efficient scheduling and placement for merge–split operations. In doing so, TopoLS discovers lattice surgery layouts that minimize space–time volume, striking a balance between scalability and optimality that existing heuristic and formal methods cannot achieve. Figure 1 presents an overview of TopoLS . **First**, we introduce topological-level optimization using ZX representation, in which quantum circuits are transformed into ZX diagrams and simplified through spider fusion. The resulting diagrams are sliced into layers based on their topological connections, thereby directly capturing merge–split operations and enabling reductions in space–time volume that are unattainable with gate-based models. **Second**, we apply 3D diagram layout optimization with Monte Carlo tree search (MCTS). Using the ZX diagram enriched with layer-slicing information, MCTS searches for efficient embeddings

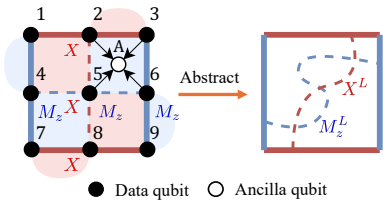

Figure 2. Logical qubit and patch abstraction

of operations in 3D space, guiding the compilation toward layouts with reduced volume. **Third**, we enable scalability through topology-aware circuit partitioning. By dynamically partitioning circuits based on spider connectivity, we limit the number of operations per layer, making the embedding problem tractable while preserving topological advantages.

Across benchmarks, TopoLS achieves an average 33% reduction in space–time volume over state-of-the-art baselines, with improvements ranging from 4% to 87%. These gains remain similar across different hardware architectures, delivering 20% to 26% reductions on grid layouts of varying shapes and sizes. In terms of scalability, TopoLS efficiently compiles circuits up to 100 qubits, while maintaining linear growth in compilation time.

The major contributions of this paper can be summarized as follows:

1. We present TopoLS, a scalable and effective compiler optimization framework for lattice surgery that fully exploits its topological properties.
2. TopoLS incorporates a suite of optimization techniques, including ZX-diagram simplification and slicing, 3D diagram layout optimization, and topology-aware circuit partitioning.
3. Our evaluation shows that TopoLS consistently outperforms baseline compilers by either achieving substantial reductions in space–time volume or demonstrating greater scalability.

2 Preliminary

This section provides background on lattice surgery for error-corrected quantum computing, together with an overview of ZX-calculus and Monte Carlo tree search, which form the basis of our optimization methods. Additional details and in-depth discussions can be found in the references cited within each subsection. For general knowledge of quantum computing, we refer readers to [21].

2.1 Rotated Surface Code

In this paper, we focus on the 2D rotated surface code [8, 14] architecture. It is one of the leading technology roadmaps for fault-tolerant quantum computing due to its high error threshold, 2D nearest-neighbor interaction, and well-defined, efficient implementation of logical operations. Figure 2 shows one logical qubit patch for a distance-3 rotated

surface code. This logical qubit is composed of nine data qubits. Ancilla qubits are employed to perform syndrome measurements for error detection, with different patch colors indicating different types of checks. For example, ancilla qubit A carries out the $Z_2Z_3Z_6Z_5$ measurement over four neighboring data qubits in the blue patch to detect bit-flip (X) errors, while the red patches correspond to multi-X measurements for detecting phase-flip (Z) errors. In this framework, the blue and red boundaries are determined by the type of syndrome measurement along the edges, and logical operations are represented by paths connecting opposite boundaries of the same color. For instance, applying a logical X gate (commonly denoted with a superscript L, as X^L) can be achieved by applying physical X gates to qubits 2, 5, and 8, while a logical Z-basis measurement can be implemented by measuring qubits 4, 5, and 6 in the Z basis. In practice, logical X^L and Z^L operations are implemented in software [8], while other logical operations will be discussed in later sections.

Although large-scale realizations of surface code patches with larger code distance require more physical qubits to construct a single logical qubit—thereby increasing robustness—the inherent structure of colored boundaries and logical operators remains unchanged. As a result, detailed layouts are often replaced by a simplified patch abstraction, as illustrated on the right of the Figure 2. We will use this simplified representation for the following sections.

2.2 Lattice Surgery on Surface Code

Lattice surgery [14] can enable efficient multi-logical-qubit operations on the rotated surface code, where it can be reasoned as merges and splits. An example is shown in Figure 3 (a). When the blue (Z) boundaries of two logical qubits are merged, the product of their logical X^L operators becomes connected and is measured as $X_1^L X_2^L$. The subsequent split operation separates the two logical qubits after this measurement. Similarly, a $Z_1^L Z_2^L$ measurement can be performed by merging and then splitting the red (X) boundaries. Further details are provided in [14].

A CNOT gate can be implemented via multi-qubit measurements and, consequently, realized through lattice surgery, as illustrated in Figure 3 (b). In Step 1, an ancilla qubit A is initialized in the $|0\rangle$ state, and an X-type multi-qubit measurement is performed between q_2 and A; based on the outcome, a conditional Pauli Z gate is applied to q_1. Step 2 performs a Z-type multi-qubit measurement between q_1 and A, followed by a conditional Pauli X gate on q_2. Finally, the ancilla is measured in the X basis, and a correction is applied to q_1. As noted in Section 2.1, the red-highlighted corrections can be implemented in software via Pauli frame updates, rather than as physical gate operations.

2.3 Pipe Diagram for Lattice Surgery

Tracking quantum programs implemented via lattice surgery requires the use of the pipe diagram [26]. The pipe diagram

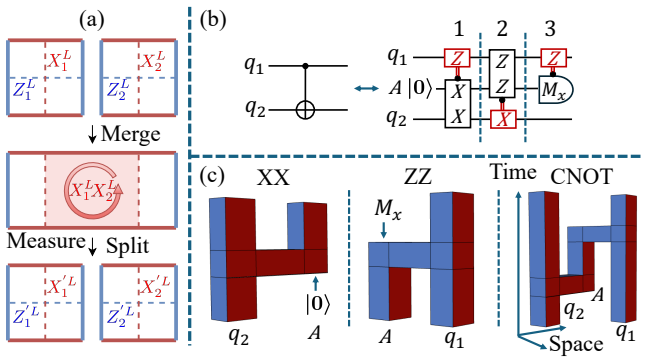

Figure 3. Lattice surgery and pipe diagram

represents the trajectory traced by a logical qubit patch in space and time during execution. In Figure 3 (c), we illustrate the execution of a CNOT gate in the pipe diagram as an example. First, an XX measurement is performed between q_2 and A, causing the blue boundaries of these two logical qubits to merge and then split. The resulting space–time trajectory is shown in left of Figure 3 (c), where the blue boundaries of the two logical qubits disappear and their red boundaries become connected. The ancilla qubit has no prior history in time, as it is initialized to $|0\rangle$ immediately before the merge–split operation. Next, a ZZ measurement is performed between q_1 and A, merging and then splitting their red boundaries. The corresponding space–time trajectory is shown in middle of Figure 3 (c), where the red boundaries of the two logical qubits vanish and their blue boundaries are connected. The ancilla qubit has no subsequent history in time, as it is measured immediately after the merge–split operation. Concatenating these two operations yields the space–time pipe diagram for a CNOT gate, shown in right of Figure 3 (c). The **space–time volume**, which estimates the physical resources required for program execution, is calculated from the product of the space (2×2) and the time steps (2), giving a total of 8 in this example.

2.4 ZX-calculus

In lattice surgery, the fundamental operations are merge and split, which are used to construct quantum gates. As a result, the gate-level representation, such as a quantum circuit, cannot adequately capture lattice-surgery-based computation. Instead, a more suitable and expressive framework is provided by the ZX-calculus [5].

The ZX-calculus is a graphical language for representing and reasoning about quantum computations. Unlike quantum circuits, which are sequences of gates applied over time, ZX diagrams represent quantum processes as networks of nodes and edges, emphasizing algebraic relationships and topological structure rather than temporal order.

Spiders The basic building blocks in ZX calculus are spider nodes representing specific tensor networks. There are two

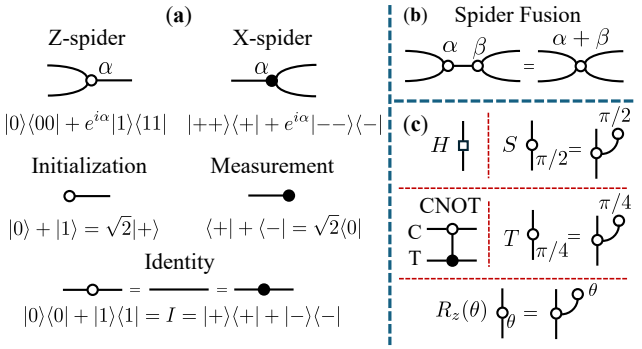

Figure 4. ZX-calculus examples

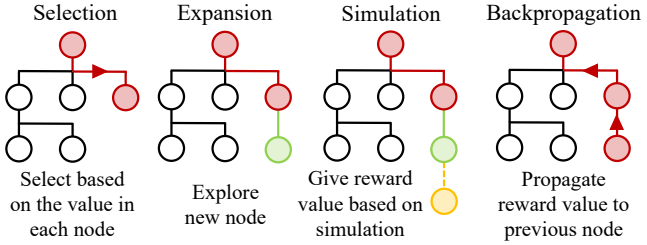

Figure 5. Overview of Monte Carlo tree search

types: Z-spider (white) and X-spider (black). As shown in Figure 4 (a), a Z-spider with phase α corresponds to a tensor network expressed in the Z basis, while an X-spider with phase α corresponds to a tensor network in the X basis. The number of wires attached to the right and left of a spider matches the number of qubits appearing in the ket and bra components of its associated tensor network respectively. Spiders with a single connected wire correspond to either initialization or measurement operations. Additionally, two-wire spiders with phase 0 represent the identity.

Spider Fusion Spiders of the same type can be merged into a single spider, with the resulting phase equal to the sum of their original phases, as illustrated in Figure 4 (b).

Quantum Gate in ZX-calculus Figure 4 (c) shows the ZX diagrams for several commonly used quantum gates. The Hadamard (H) gate is represented as a box. The S gate corresponds to a Z-spider with phase $\pi/2$. It can equivalently be represented as a Z-spider connected to an auxiliary phase-$\pi/2$ node using spider fusion, making explicit its interpretation as a Y-basis measurement or initialization. For the T gate or the R_Z gate, the single auxiliary node corresponds to a state-injection process. The X and Z gates are omitted here, as they are executed in software.

2.5 Monte Carlo Tree Search

In this section, we introduce the concept of Monte Carlo tree search (MCTS) [2], which will later be employed to optimize the space–time volume of the pipe diagram.

MCTS is a heuristic search algorithm widely used in decision making problems with large search space. It incrementally builds a search tree by iteratively simulating possible future outcomes, balancing exploration (trying less-visited actions) and exploitation (focusing on actions with high estimated value). Each iteration consists of four stages: **Selection**, where the algorithm traverses the current tree using a policy such as Upper Confidence Bound for Trees (UCT) to select a promising node; **Expansion**, where one or more new child nodes are added to the tree; **Simulation**, where the outcome of the problem is estimated via random or policy-guided playouts; and **Backpropagation**, where the results of the simulation are propagated up the tree to update value estimates. We demonstrate the four steps in Figure 5.

MCTS offers a powerful balance between exploration and exploitation, enabling it to navigate large and complex decision spaces without relying on explicit evaluation functions. It progressively focuses on high-value solutions while retaining the flexibility to discover new possibilities. The anytime property ensures that a workable solution is available early and can be refined with additional computation, making it particularly advantageous for optimization tasks such as reducing the space–time volume in pipe diagrams, where the search space is vast and the optimal solution is difficult to determine analytically.

3 Motivation

In this section, we present the motivation for our compiler, TopoLS. We begin by discussing how the ZX-calculus can be used to represent lattice surgery, along with the key topological properties that underpin lattice-surgery operations.

3.1 Interpreting Lattice Surgery Using ZX-calculus

ZX-calculus has been widely used in high-level QEC-agnostic logical quantum circuit analysis and optimization [6, 17, 22]. Recent works employed ZX-calculus to represent surface code lattice surgery [5], assist in manually designing fault-tolerant layouts [11], and verify the functionality of lattice surgery operations [26]. However, it is not yet well explored how ZX-calculus can be used for automating the construction and optimization of lattice surgery operations.

In this paper, our objective is to map and optimize logical operations to lattice-surgery structures, enabling scalable fault-tolerant circuit design. We select ZX-calculus rather then the quantum circuit representation. In the conventional quantum circuit diagrams, the spatial dimensions and the temporal dimension are mostly fixed. However, lattice surgery allows interchange among spatial and temporal dimensions. A lot of design and optimization space are lost when directly constructing the lattice surgery operation from a quantum circuit. In the rest of this section, we will elaborate on why our compiler framework is built upon ZX-calculus representations.

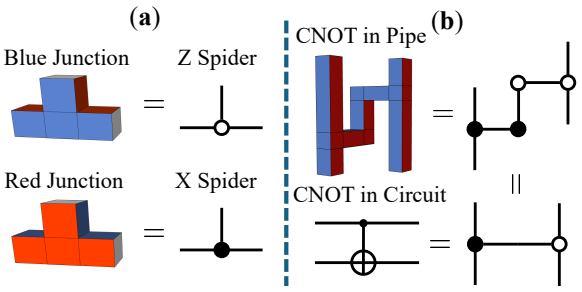

Figure 6. Pipe diagram to ZX-calculus

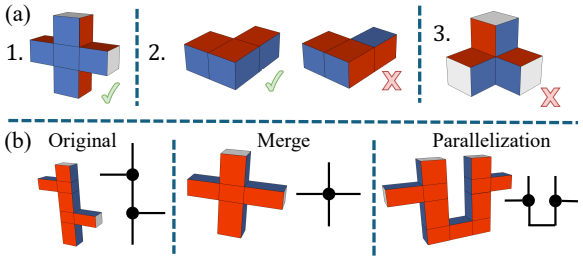

Figure 8. Pipe restrictions and transformations

We first introduce how ZX diagrams can be used to represent lattice surgery. The blue junction in the merge–split process corresponds to a Z spider, while the red junction corresponds to an X spider, as illustrated in Figure 6 (a). Based on this, the quantum circuit can be directly connected to the lattice surgery pipe diagram using the ZX diagram, without explicitly considering the multi-qubit measurements. In Figure 6 (b), we illustrate this with the example of a CNOT gate. Directly translating the colored junctions into spiders yields two X spiders and two Z spiders. However, as discussed in Section 2.4, two of these spiders have only two wires and thus correspond to the identity operation. Consequently, the configuration is equivalent to a three-wire X spider connected to a three-wire Z spider, which precisely matches the ZX representation of a CNOT gate in the circuit model.

3.2 Topological Nature of Lattice Surgery

Topology of ZX diagram In a ZX diagram, only the topology—that is, the connectivity between spiders—matters. Wires can be stretched, bent, or rearranged without altering the represented computation, as shown in left of Figure 7. As discussed in Section 3.1, there is a correspondence between a lattice surgery program and its ZX diagram, which suggests that lattice surgery is also insensitive to specific operation instantiations.

Using CNOT gate as an example, the execution of a CNOT operation in lattice surgery is independent of the specific path connecting the three-port red junction and the three-port blue junction, providing ample flexibility in identifying feasible connection paths (the path finding is also called **routing problem**). As shown in right of Figure 7, the three

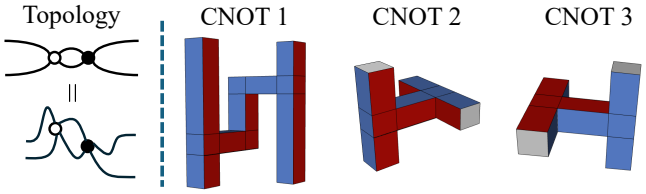

Figure 7. Left: topological property of ZX diagram; Right: different implementations of CNOT

different executions of CNOTs result in the same computation. This topological property also extends to the execution of other quantum gates. **Constraints** Some constraints need to be satisfied when constructing the pipe diagram and an arbitrary colored path in space and time may not constitute a valid pipe diagram. For example, the orientation of blue and red color as well as its layout must satisfy the constraints in Figure 8 (a):

1. All ports of a junction must lie in the same plane, and each junction may have at most four ports,
2. The colors at a junction must be consistent,
3. Three-dimensional corners are not permitted.

There are additional pipe constraints outlined in [26] and all of them are considered in our pipe diagram construction algorithm later.

3.3 Our Motivation

The topological property described above naturally gives rise to multiple optimization opportunities.

First, the ZX diagram provides a representation of the program at the topological level, enabling optimizations that are not easily accessible in the gate-based model. For example, spider fusion can be applied to combine merge–split operations, as illustrated in middle of Figure 8 (b), or the execution order of merge–split operations can be rearranged to enhance parallelism, see right of Figure 8 (b). Such optimizations arise from the fact that lattice surgery, like to ZX diagrams, is agnostic to routing details; consequently, structural transformations applied at the ZX-diagram level directly translate into high-level modifications of the lattice surgery protocol.

Second, the topological nature of lattice surgery allows us to explore a much larger solution space for the routing problem. Prior work has primarily addressed routing by considering only a 2D slice of the quantum circuit, overlooking the inherently 3D structure and topological properties of lattice surgery. This distinction is illustrated in Figure 9. In the example circuit with two CNOT gates, restricting routing to the 2D circuit slice (where yellow nodes denote data qubits and green nodes denote ancilla qubits) prevents both CNOTs

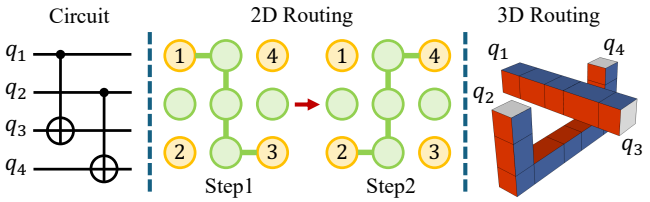

Figure 9. 2D routing vs 3D routing

from being executed simultaneously. In contrast, when routing is extended into the full 3D space, spare regions from earlier execution steps can be reused to establish the necessary topological connections, enabling more efficient scheduling.

4 Optimization at Topological Level Using ZX Representation

This section presents the optimizations performed at the topological level. We begin by transforming the circuit into a ZX diagram and applying spider fusion to simplify the program. Subsequently, we optimize the execution order of merge-split using layer slicing.

4.1 ZX Representation and Spider Fusion

To enable topological optimization, the first step is to transform the quantum circuit into its ZX diagram representation. We illustrate this process with the example in Figure 10 (a), which has a circuit depth of 6. Although only H, S, and CNOT gates are shown, other gates can be handled using the same techniques described below. The corresponding ZX diagram obtained from this direct transformation is shown in Figure 10 (b).

Following this transformation, we can optimize the lattice surgery merge–split operations by fusing spiders of the same type in the ZX diagram, as described in Figure 8. In the example above, three Z spider fusions and one X spider fusion are performed, with the resulting diagram shown in Figure 10 (c). When merging spiders, it is important to ensure that the resulting spider has no more than four wires, since a junction can accommodate at most four ports, as illustrated in Figure 8 (a).

4.2 Optimize the Instruction Order Through Topological Connection

Now that we have a ZX diagram representing the topological connections of the operations in the lattice surgery program, it needs to be translated into an executable program with a specific schedule. This requires determining the execution order of the operations in the ZX diagram. We refer to this process as layer slicing. An effective slicing strategy can significantly reduce execution overhead.

For this slicing process, we apply a breadth-first search on the ZX diagram. As shown in Figure 10 (d), We start from the leftmost nodes of the diagram, they are labeled as layer 1. Firstly, we visit all their immediate neighbors and label them as layer 2. For each of these neighbors, we then visit their unvisited neighbors and label them as layer 3. This process continues until all reachable spiders have been visited and assigned a layer. In this example, the execution layer is reduced to 4 layers, compared to the original circuit depth of 6. Note that we do not label single-wire spiders with a phase, as they serve as primitives for the spiders they are connected to. Their detailed execution will be illustrated in Figure 13 in the following section.

At this stage, the layer labels provide a chronological execution order for the ZX spiders (which correspond to merge–split operations in lattice surgery). The task of determining their detailed placement and exact time of execution will be addressed by the MCTS embedding in the next section. To facilitate efficient MCTS embedding, we need to identify certain information for the spiders within the same layer. Using the third layer from the previous example as a case study, we illustrate this in Figure 10 (e). For a spider in layer 3, we define input as the edges coming from the previous layer (layer 2), shown with blue arrows; output as the edges going to the next layer (layer 4), shown with red arrows; and interconnection as the edges linking spiders within the same layer (layer 3), shown with yellow edge.

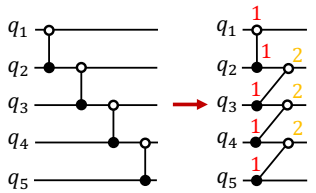

Figure 11. Ladder circuit example

Before moving on to the next section, we present a small example to illustrate how this breadth-first search layer slicing method can help optimize the execution of lattice surgery. Figure 11 depicts a ladder circuit with 5 qubits. Executing the circuit gate by gate would require 5 steps to complete. However, if we treat the merge-split operation as the fundamental instruction and apply our slicing method, the process can be completed in just 2 steps, regardless of the number of qubits. This effect is directly illustrated in the GHZ state generation shown in Figure 16 of the evaluation section.

5 3D Diagram Layout Optimization

In this section, we present how MCTS can be used to convert the ZX diagram, enriched with layer slicing information, into an executable lattice surgery program, we refer to this task as the layout embedding problem.

5.1 Embedding Strategy at High Level

We first provide a high-level overview of our method for embedding the ZX diagram into 3D space (two spatial axes and one temporal axis), as shown in Figure 12.

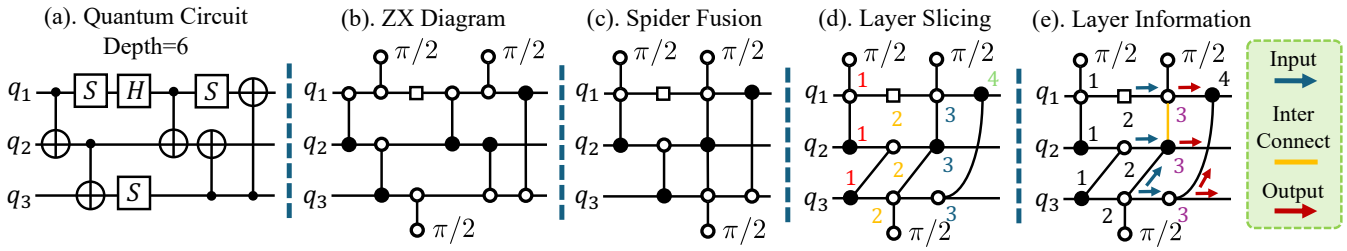

Figure 10. Topological optimization via ZX representation

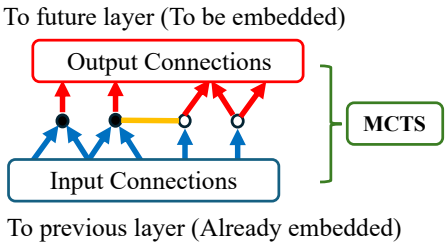

Figure 12. High-level overview of MCTS embedding

We perform the embedding layer by layer, following the order determined by the slicing process described in the previous section. Before embedding a specific layer (in the middle of Figure 12), all spiders in the preceding layer must already be embedded. Their topological connections to the spiders in the current layer are represented by blue arrows, which we refer to as input connections (as defined in Section 4.2). At this stage, the role of MCTS is to determine a concrete placement for the spiders in the current layer, along with the instantiation of the input connections (blue arrows), interconnections within the layer (yellow edges), and output connections (red arrows). The output connections link to future layers, and these will be embedded only after the current layer has been completed. Layout optimization arises from encoding the space-time volume of the embedding as the MCTS objective function, guiding the search toward layouts with reduced volume.

5.2 MCTS for 3D Embedding

In this section, we describe our implementation of MCTS for solving the ZX diagram embedding problem. We begin by explaining how frequently used quantum gates (spiders) are interpreted within TopoLS using ZX-calculus, as shown by the orange boxes in Figure 13, noting that the CNOT gate has already been introduced in Section 3.1.

The **H gate** is represented as a yellow box that swaps the red and blue faces in the pipe diagram. In ZX-calculus, the **S gate** is represented by a three-port blue junction together with a Y-basis measurement or initialization, shown as a green cap in the pipe diagram. The **T gate** follows a similar interpretation: it involves a three-port blue junction for magic state injection, which produces a $-\pi/4$ phase with 50% probability and necessitates an conditional S correction to obtain the desired $\pi/4$ phase. Recently, the R_z **gate** has gained attention [24], also realized via magic state injection; here, the injection produces $-\theta$ with probability 1/2, necessitating a corrective rotation of 2θ by another injection. This process is probabilistic in depth, continuing until the sum of applied injection angles equals θ, with the failure probability decreasing exponentially with the number of injections.

One major limitation of above magic state injection techniques is that the scheduler must wait for each injection to complete before proceeding with subsequent operations, making performance heavily dependent on the speed and throughput of the magic state factories. Leveraging the topological properties of lattice surgery, we can route the qubit out of the main program for magic state injection, allowing the injection process to occur asynchronously, and then route it back in, as illustrated by the green box in Figure 13. This is because our focus is on the execution topology rather than the specific routing details. From the program's perspective, the in-circuit injection appears to take only two time steps, while the actual injection is performed externally. The only requirement is that all X and Z operations tracked in software for the qubit **must** be clearly known before performing the magic state injection, otherwise, it will lead to an incorrect interpretation of the measurement during the injection process. This is related to the causality. Although the T gate is used as an example, the same approach applies to R_z gate injections. TopoLS adopts this implementation.

Having established the execution details of the different types of spiders, we are now ready to introduce the four phases of MCTS.

Embedding State serves as the tree node in MCTS and is defined by the following primitives: (1) **Position**: a dictionary storing the 3D positions of all embedded spiders; (2) **Spider Type**: a dictionary indicating whether a spider is an X-spider, a Z-spider (optionally with a phase), or a Hadamard box; (3) **Orientation**: a dictionary specifying the face direction of the pipe junction for the embedded spider, which is perpendicular to all its ports at the junction. As illustrated by the green arrow in Figure 14 (a), no path may connect to the spider from this face direction. This serves as a strict constraint on the pipe structure, as described in Section 3.1. (4) **Path**: a list of paths connecting the spiders.

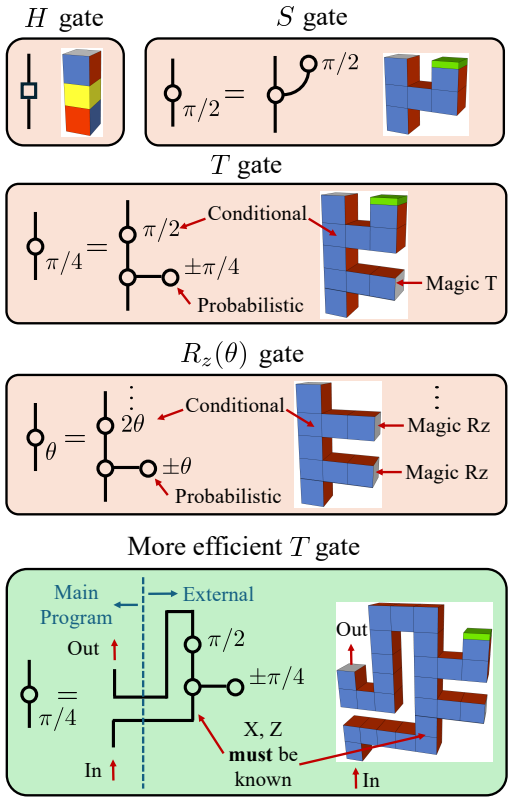

Figure 13. Quantum gates in lattice surgery

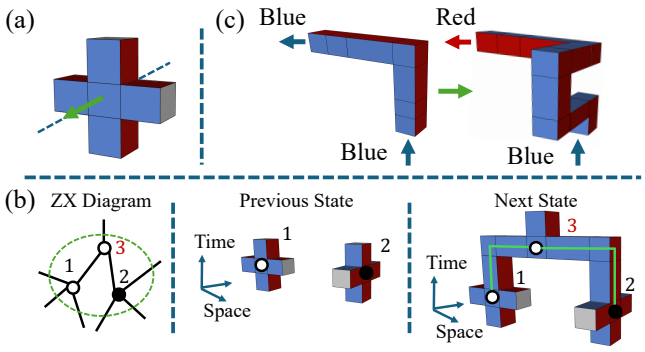

Figure 14. Techniques in MCTS Embedding

Selection follows the UCT policy, where the value of each tree node is accumulated based on the simulated space-time volume of the pipe diagram.

Expansion is defined as a one-step transformation between different Embedding States. In this process, a non-embedded spider is selected and placed at a vacant position in 3D space. The embedding order of spiders within the same layer is chosen randomly, and the position of each spider is assigned as a neighboring location of its already embedded topological neighbors in the ZX diagram. A straightforward placement strategy is to position the spider directly above one of its neighbors; this deterministic approach significantly accelerates the search process but sacrifices the ability to explore more complex 3D placements that may lead to further optimization. To leverage the full 3D structure, we allow placements to occur in any neighboring direction, with the number of candidate positions configurable by the user—an option we refer to as spider **placement optimization**. Once the spider placed, its connections to the embedded spiders are established by finding the shortest available path in 3D space. See Figure 14 (b) for an example. We consider two Z-spiders and one X-spider, with numbers indicating their embedding order. The first two spiders have already been embedded, as shown in the "Previous State" in the middle. During expansion, a position is determined for the third spider, and its connections to spiders 1 and 2 are established, as illustrated by the green lines in the "Next State" of Figure 14 (b). In addition, if a spider carries a phase (such as in the case of an S gate) or is associated with a Hadamard box, additional routing or procedures are applied as illustrated in Figure 13.

However, the shortest path does not always yield the correct pipe color, since the colors at junctions must remain consistent. In such cases, a "color switch" is required. An example is shown in Figure 14 (c), where the color orientation is adjusted by applying two additional merge-split operations at a pipe corner.

Simulation proceeds by embedding spiders sequentially until all spiders in the layer have been placed. Once completed, the resulting space-time volume is assigned as the reward value.

Backpropagation revisits the parent nodes in the MCTS tree, adding the obtained reward (space-time volume) to their cumulative value and incrementing their visit counts.

This MCTS strategy is applied to each layer in the ZX diagram until all layers have been embedded in the 3D space. In the first layer, all qubits are placed at grid positions. If the embedding of a given layer fails, the qubits are reassigned to new grid positions and the embedding procedure is retried. Should this attempt also fail, the process defaults to the baseline compilation method for that layer, which is guaranteed to succeed.

Correctness of TopoLS Embedding The correctness of the pipe diagram as a logical quantum circuit is ensured by construction. First, the topological-level optimizations using the ZX representation in Section 4 do not alter the circuit's functionality, as ZX-spider fusion preserves semantics. Second, the MCTS-based embedding introduced in this section also maintains logical correctness. This is because we ensure not only that each ZX node is accurately placed within the 3D space, but also that the logical connections between nodes are faithfully preserved throughout the embedding construction.

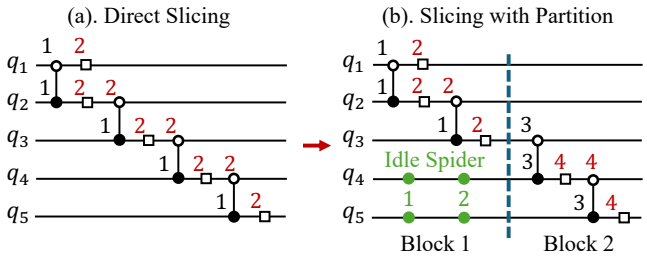

Figure 15. Circuit partition example

6 Enable Scalability by Topology-Aware Circuit Partition

The layer slicing technique introduced in Section 4.2 can significantly parallelize merge-split operations. However, in some cases it may produce a large number of spiders within the same layer, making it difficult for MCTS to efficiently find an embedding solution.

An example is shown in Figure 15 (a), where 5 qubits diagram have 8 spiders in the second layer. As the number of qubits increases, the operations in the second layer grow proportionally, reaching twice the number of qubits.

To address this, we apply circuit partitioning to reduce the accumulation of spiders within a layer. As shown in Figure 15 (b), introducing a single partition in the middle reduces the maximum number of spiders per layer from 8 to 6. However, this also increases the total number of layers from 2 to 4. An extreme case occurs when the circuit is partitioned at every depth, resulting in the maximum number of operations per layer being equal to the number of qubits, while the total number of layers equals the circuit depth.

After partitioning, the circuit is divided into separate blocks, requiring slight modifications to the slicing algorithm described in Section 4.2. For each new block, we assign spider labels starting from the maximum layer index of its preceding blocks. For example, in Figure 15 (b), block 2 begins at layer 3. However, partitioning may sometimes disrupt layer consistency, causing neighboring spiders to have non-consecutive layer numbers. To resolve this, we insert idle spiders to enforce consistency, as illustrated by the green node in Figure 15 (b).

Good partitioning of the circuit can help reduce execution overhead. A natural approach is uniform partitioning, which typically requires shallow partitions—such as splitting every 5 circuit depths—to keep the number of spiders per layer at a reasonable level. However, this increases the total layer count to approximately match the original circuit depth, thereby diminishing the advantages of using the ZX representation.

Our **circuit partition optimization** uses a topology-aware partitioning method that explicitly incorporates spider connectivity into the partitioning process. The circuit depth of each partition block is determined dynamically. For each new block, the partitioning begins with a circuit depth of 2. We then apply slicing and evaluate the spider connectivity across all its layers, since this measure directly reflects the number of pipes that must be established between layers. If the connectivity remains below a predefined threshold, the block is extended by one additional circuit depth. Once the threshold is exceeded, the partition is finalized at the preceding circuit depth.

7 Evaluation

In this section, we evaluate TopoLS by comparing it against state-of-the-art baselines, analyzing the impact of each optimization step, examining the effects of different architectures, and assessing scalability performance.

7.1 Experiments Setup

Baseline: Our baseline compilers include two heuristic compilers based on the quantum circuit model for lattice surgery compilation: Liblsqecc [28] and DASCOT [20], as well as a SAT-solver-based compiler LaSsynth [26] for Clifford circuits. Note that Liblsqecc can manage the physical qubit resources in two configurations: 'sparse layout', where data qubits are interleaved with ancilla qubits, and 'tight layout', where all data qubits are arranged in two rows with a single row of ancilla qubits placed between them. TopoLS, DASCOT, and LaSsynth all adopt the 'sparse layout'.

Experiment Configurations To illustrate the effect of each optimization step, we design three experiment configurations. 1. 'Place-Opt' enables placement optimization during the expansion phase of MCTS (see Section 5.2), allowing exploration of spider placement in 3D space. 2. 'Part-Opt' applies the topology-aware circuit partitioning method introduced in Section 6, in contrast to a uniform partitioning applied every 5 circuit depths. 3. 'Full-Opt' is to apply all TopoLS optimizations. Note that all the tree configurations above incorporate the topological optimization based on ZX diagrams introduced in Section 4, as they are constructed upon this optimization.

Hardware Configuration We adopt a two-dimensional lattice coupling of physical qubits as our device architecture. The logical qubits are square patches on the physical qubit lattice. This is a common setting widely used in previous works [20, 26, 28].

Benchmarks We evaluate TopoLS using a variety of quantum algorithms of different types and sizes, with detailed information provided in Table 1.

Metrics We mainly use the following two metrics: (**a**) **Space-Time Volume:** The product of space and time occupied by the lattice-surgery program, representing the physical resources consumed. (**b**) **Compilation Time:** The CPU time required to translate a quantum circuit into executable lattice-surgery instructions.

Table 1. Benchmark Information

Benchmark	Qubit#	Depth	Gate#
Bernstein-Vazirani (BV)	16	12	31
	20	13	50
	30	21	78
	40	25	102
	50	28	125
	60	35	152
	80	42	199
	100	52	249
Deutsch-Jozsa (DJ)	16	17	50
	20	24	87
	40	43	162
	60	63	251
	80	83	341
	100	103	424
Grover Search	6	457	653
Quantum Fourier Transform (QFT)	16	236	1102
Quantum Phase Estimation (QPE)	16	325	1211
Variational Quantum Eigensolver (VQE)	16	34	256
GHZ State	16	16	16
W State	16	95	232
Quantum Approximate Optimization Algorithm (QAOA)	16	34	136
Random Clifford Circuit	4	8	12
	6	7	18
	8	9	24
	10	9	30
	12	9	36

Implementation We implemented TopoLS in Python, utilizing the PyZX package [16] to help with ZX-diagram-based optimizations. All experiments were conducted on a server with a 2.4 GHz CPU and 0.75 TB of memory. For the MCTS implementation, we selected two random seeds for each layer embedding to change the embedding order of spiders, with the number of iterations fixed at 1000 and a timeout of 2 seconds. Among the results from different spider embedding orders, the one yielding the smaller space-time volume was chosen. Regarding the magic state, we take the *Rz* state as the magic state and adopt the assumption from the baseline [20]: the magic state factory is not included in the space-time volume, and the distillation time is omitted.

7.2 Overall Result

Table 2 summarizes the overall compilation results of TopoLS for nine benchmark algorithms with 16 qubits (except for Grover search, which uses 6 qubits). The circuit depths range from 12 to 457, and the gate counts range from 16 to 1211. For the sparse layout configuration, we adopt a square lattice hardware topology in which each row and column contains 4 data qubits interleaved with ancilla qubits, forming a 4×4 grid. For Grover's search, the layout is a 2×3 grid.

Compared with the best space-time volume results among the baseline compilers, the best results from TopoLS achieve an average reduction of 33%, we also provide the detailed reduction for each benchmark, listed alongside their names in the table. The SAT-solver-based compiler LaSsynth is not included in the table, as many of the benchmarks are non-Clifford circuits and the system sizes are too large for it to handle. Across benchmarks, the reduction ranges from 4% to 87%, reflecting their inherently different structures. Even in the case of Grover search, where the space-time volume reduction is only 4%, we still observe a 59% reduction in time, albeit with a trade-off of increased space. We show the pipeline diagram for GHZ state generation in Figure 16, which spans 4 time steps, with 2 of them allocated to the input and output ports. Because GHZ state generation requires only a single H gate and a ladder circuit, this aligns with the 2-layer ladder circuit implementation illustrated in Figure 11.

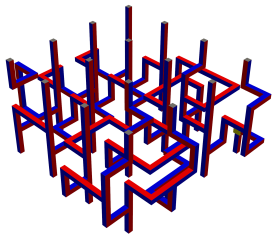

Figure 16. Compiled pipe diagram for 16-qubit GHZ state

In addition, Table 2 presents the breakdown of results for different optimization techniques. On average, enabling placement optimization yields a 14% reduction in space-time volume, while enabling topology-aware circuit partitioning yields a 19% reduction.

7.3 Hardware Structure Impact

We studied the impact of different hardware structures in this section, with the results summarized in Table 3. We select QFT, QPE, VQE as benchmark circuits. For TopoLS, full optimization was applied, while for Liblsqecc only the sparse layout was considered, since its compact layout is fixed for a given qubit number and has already been examined in Section 7.2. The hardware architectures evaluated include a 6×3 grid, an 8×2 grid, and a linear 16×1 grid. The average reductions achieved are 21%, 26%, and 25%, respectively, which are consistent across different architectures and comparable to the results reported in Section 7.2 for the 4×4 grid. The detailed reduction for each benchmark is listed alongside their names in Table 3.

7.4 Scalability Study

We evaluate the scalability of TopoLS on larger-qubit circuits in this section, with the results summarized in Table 4. BV and DJ algorithms are selected as benchmarks, with qubit counts ranging from 20 to 100. The average space–time volume reductions achieved are 63% for BV and 43% for DJ. We

Table 2. Overall Comparison. '*' indicates the best result among three configurations of TopoLS; '◇' indicateds the best result among three baseline configurations. Reduction in space-time volume is in the brackets next to the benchmark name.

Benchmark (Ave: 33%)	BV (56%)		DJ (48%)		Grover Search (4%)	
Compiler / Metrics	Space-time	Comp time (s)	Space-time	Comp time (s)	Space-time	Comp time (s)
TopoLS (Place-Opt)	1134	40.5	1377	68.7	29680	1974.6
TopoLS (Part-Opt)	567*	9.9	1215	10.0	36120	1171.5
TopoLS (Full-Opt)	648	25.2	1134*	34.9	29505*	2032.7
DASCOT	2511	1.1	3969	1.1	33425	3.2
Liblsqecc (tight)	1290◇	0.05	2190◇	0.08	30675◇	1.8
Liblsqecc (sparse)	4257	0.08	7029	0.14	44415	1.4
	QFT (17%)		QPE (21%)		VQE (12%)	
Compiler / Metrics	Space-time	Comp time (s)	Space-time	Comp time (s)	Space-time	Comp time (s)
TopoLS (Place-Opt)	47304	2715.5	54432	2932.2	5508	339.7
TopoLS (Part-Opt)	51111	1291.0	57753	1375.6	6237	175.6
TopoLS (Full-Opt)	46332*	2777.5	50787*	2984.9	5346*	440.7
DASCOT	56133◇	675.9	64152◇	821.2	6075◇	15.9
Liblsqecc (tight)	93180	3.6	96930	3.8	20580	0.77
Liblsqecc (sparse)	175230	4.3	192357	4.5	42471	0.95
	GHZ State (87%)		W State (25%)		QAOA (23%)	
Compiler / Metrics	Space-time	Comp time (s)	Space-time	Comp time (s)	Space-time	Comp time (s)
TopoLS (Place-Opt)	1134	37.1	12555	620.5	5751	288.6
TopoLS (Part-Opt)	567	13.4	14337	309.7	5751	119.7
TopoLS (Full-Opt)	324*	12.8	10611*	673.8	4941*	268.8
DASCOT	2511◇	1.2	14094◇	39.2	6399◇	6.0
Liblsqecc (tight)	3000	0.11	14850	0.59	10320	0.39
Liblsqecc (sparse)	3267	0.07	37521	0.84	17424	0.43

Table 3. Compilation results under different hardware structures. Reduction in space-time volume is in the brackets next to the benchmark name.

6 × 3 grid (Ave: 21%)	QFT (17%)		QPE (24%)		VQE (21%)	
Compiler / Metrics	Space-time	Comp time (s)	Space-time	Comp time (s)	Space-time	Comp time (s)
TopoLS	52234	2811.0	55146	2974.6	5369	391.0
DASCOT	62972	716.1	72709	875.3	6825	17.7
Liblsqecc	186795	3.5	203700	3.7	45045	0.78
8 × 2 grid (Ave: 26%)	QFT (20%)		QPE (22%)		VQE (35%)	
Compiler / Metrics	Space-time	Comp time (s)	Space-time	Comp time (s)	Space-time	Comp time (s)
TopoLS	46920	2683.4	52615	2983.2	4165	306.8
DASCOT	58650	771.4	67235	912.2	6375	19.1
Liblsqecc	169955	3.2	185250	3.4	40755	0.70
16 × 1 grid (Ave: 25%)	QFT (23%)		QPE (25%)		VQE (27%)	
Compiler / Metrics	Space-time	Comp time (s)	Space-time	Comp time (s)	Space-time	Comp time (s)
TopoLS	53064	2621.6	59796	2946.0	5445	326.8
DASCOT	69102	923.8	79596	921.8	7425	28.1
Liblsqecc	191940	2.7	210210	2.8	45465	0.56

observe that the compilation time of TopoLS introduces a constant overhead compared with the baseline compiler, but increases linearly with system size. The trend of compilation time growth is illustrated in Figure 17.

We compare TopoLS with the SAT-solver-based compiler LaSsynth in Table 5. While the SAT solver can find the optimal solution and achieve results that are approximately 57% better than ours when it succeeds, its scalability is highly limited. As the system size increases, the problem becomes

Table 4. Compilation results for two benchmarks of various sizes

Compiler	BV (qubit#)	20	40	60	80	100	DJ (qubit#)	20	40	60	80	100
TopoLS	Space-time	792	2925	4046	6498	8379	Space-time	1485	7995	20519	26353	47628
	Comp time (s)	18.7	44.8	41.5	58.6	67.4	Comp time (s)	63.7	259.7	608.7	503.6	907.4
DASCOT	Space-time	3465	13845	29189	44042	67032	Space-time	6039	23790	52598	86279	133182
	Comp time (s)	1.19	1.21	1.20	1.20	1.25	Comp time (s)	1.65	2.24	2.75	2.72	26.97
Liblsqecc (tight)	Space-time	1692	6270	12960	20538	31668	Space-time	3420	14454	28128	45612	83460
	Comp time (s)	0.06	0.16	0.31	0.45	0.66	Comp time (s)	0.12	0.39	0.66	1.01	1.82
Liblsqecc (sparse)	Space-time	5499	21375	43605	65037	98049	Space-time	10764	46125	90117	140049	239568
	Comp time (s)	0.09	0.31	0.58	0.82	1.22	Comp time (s)	0.19	0.73	1.28	1.86	3.32
Volume Reduction	Ave: 63%	53%	53%	69%	68%	74%	Ave: 43%	57%	45%	27%	42%	43%

Table 5. Comparison with LaSsynth. '-' indicates that the SAT solver does not finish within 24 hours.

Compiler	BV (qubit#)	20	30	40	50	60	Random Clifford (qubit#)	4	6	8	10	12
TopoLS	Space-time	792	1430	2925	3570	4046	Space-time	225	350	637	819	693
	Comp time (s)	18.7	28.8	45.2	48.0	41.5	Comp time (s)	23.3	28.2	42.3	43.8	55.8
LaSsynth	Space-time	396	572	-	-	-	Space-time	100	140	-	-	-
	Comp time (s)	153.0	431.8	-	-	-	Comp time (s)	4.8	14.7	-	-	-
Volume Reduction		-50%	-60%	-	-	-		-56%	-60%	-	-	-

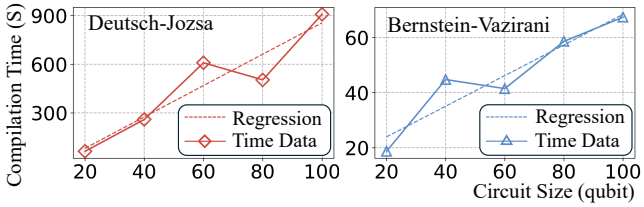

Figure 17. Scalability study: compilation time of TopoLS as circuit size increases.

exponentially harder to solve for LaSsynth. We imposed a one-day timeout for all compilation tasks, after which the LaSsynth failed to produce results beyond 40 qubits for the BV algorithm with simple structures and 8 qubits for the random Clifford circuit.

8 Related Work

Gate-Centric Quantum Compilers: Quantum compilation has made significant progress in the last decade, and several industry/academia compilers have been developed [15, 25, 29]. Most of them either target the NISQ architectures or compile to the Clifford+T gate set, staying at the purely logical level when targeting fault-tolerant quantum architecture. Recently, [28] and [20] were proposed to compile and optimize the lattice surgery operations on the surface code. However, they are still designed by the gate-centric philosophy, where the pipe diagrams are constructed by adding new structures for each two-qubit gate individually. The optimization opportunity from the topological properties of lattice surgery is largely overlooked. TQEC [27] is a design automation tool being actively developed for lattice surgery on surface code. It provides tools to build the pipe diagram from an input circuit, but it does not contain compiler optimization when synthesizing the pipe diagram, to the best of our knowledge. In contrast, TopoLS is designed to be topology-centric, which enables deeper optimizations by leveraging the topological optimization opportunities from the lattice surgery and thus outperforms previous compilers by yielding pipe diagrams with much smaller space-time volume.

SAT Solver for Lattice Surgery: In addition to heuristic compilation, Tan et al. [26] developed LaSsynth, an SAT-based approach to synthesize the pipe diagram for a quantum program by encoding the topological information and constraints into a SAT solver. Although it can find the pipe diagram with optimal space-time volume for a Clifford circuit block, it suffers from the scalability issue due to the nature of the SAT formulation. In contrast, this work can compile and optimize large circuits and support end-to-end compilation for general quantum circuits.

Acknowledgements

This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Contract No. DE-AC05-00OR22725 through the Accelerated Research in Quantum Computing Program MACH-Q project, the U.S. National Science Foundation CAREER Award No. CCF-2338773, and ExpandQISE Award No. OSI 2427020. GL was also supported by the Intel Rising Star Award.

References

[1] Rajeev Acharya, Dmitry A. Abanin, Laleh Aghababaie-Beni, Igor Aleiner, Trond I. Andersen, Markus Ansmann, Frank Arute, Kunal Arya, Abraham Asfaw, Nikita Astrakhantsev, Juan Atalaya, Ryan Babbush, Dave Bacon, Brian Ballard, Joseph C. Bardin, Johannes Bausch, Andreas Bengtsson, Alexander Bilmes, Sam Blackwell, Sergio Boixo, Gina Bortoli, Alexandre Bourassa, Jenna Bovaird, Leon Brill, Michael Broughton, David A. Browne, Brett Buchea, Bob B. Buckley, David A. Buell, Tim Burger, Brian Burkett, Nicholas Bushnell, Anthony Cabrera, Juan Campero, Hung-Shen Chang, Yu Chen, Zijun Chen, Ben Chiaro, Desmond Chik, Charina Chou, Jahan Claes, Agnetta Y. Cleland, Josh Cogan, Roberto Collins, Paul Conner, William Courtney, Alexander L. Crook, Ben Curtin, Sayan Das, Alex Davies, Laura De Lorenzo, Dripto M. Debroy, Sean Demura, Michel Devoret, Agustin Di Paolo, Paul Donohoe, Ilya Drozdov, Andrew Dunsworth, Clint Earle, Thomas Edlich, Alec Eickbusch, Aviv Moshe Elbag, Mahmoud Elzouka, Catherine Erickson, Lara Faoro, Edward Farhi, Vinicius S. Ferreira, Leslie Flores Burgos, Ebrahim Forati, Austin G. Fowler, Brooks Foxen, Suhas Ganjam, Gonzalo Garcia, Robert Gasca, Élie Genois, William Giang, Craig Gidney, Dar Gilboa, Raja Gosula, Alejandro Grajales Dau, Dietrich Graumann, Alex Greene, Jonathan A. Gross, Steve Habegger, John Hall, Michael C. Hamilton, Monica Hansen, Matthew P. Harrigan, Sean D. Harrington, Francisco J. H. Heras, Stephen Heslin, Paula Heu, Oscar Higgott, Gordon Hill, Jeremy Hilton, George Holland, Sabrina Hong, Hsin-Yuan Huang, Ashley Huff, William J. Huggins, Lev B. Ioffe, Sergei V. Isakov, Justin Iveland, Evan Jeffrey, Zhang Jiang, Cody Jones, Stephen Jordan, Chaitali Joshi, Pavol Juhas, Dvir Kafri, Hui Kang, Amir H. Karamlou, Kostyantyn Kechedzhi, Julian Kelly, Trupti Khaire, Tanuj Khattar, Mostafa Khezri, Seon Kim, Paul V. Klimov, Andrey R. Klots, Bryce Kobrin, Pushmeet Kohli, Alexander N. Korotkov, Fedor Kostritsa, Robin Kothari, Borislav Kozlovskii, John Mark Kreikebaum, Vladislav D. Kurilovich, Nathan Lacroix, David Landhuis, Tiano Lange-Dei, Brandon W. Langley, Pavel Laptev, Kim-Ming Lau, Loïck Le Guevel, Justin Ledford, Joonho Lee, Kenny Lee, Yuri D. Lensky, Shannon Leon, Brian J. Lester, Wing Yan Li, Yin Li, Alexander T. Lill, Wayne Liu, William P. Livingston, Aditya Locharla, Erik Lucero, Daniel Lundahl, Aaron Lunt, Sid Madhuk, Fionn D. Malone, Ashley Maloney, Salvatore Mandrà, James Manyika, Leigh S. Martin, Orion Martin, Steven Martin, Cameron Maxfield, Jarrod R. McClean, Matt McEwen, Seneca Meeks, Anthony Megrant, Xiao Mi, Kevin C. Miao, Amanda Mieszala, Reza Molavi, Sebastian Molina, Shirin Montazeri, Alexis Morvan, Ramis Movassagh, Wojciech Mruczkiewicz, Ofer Naaman, Matthew Neeley, Charles Neill, Ani Nersisyan, Hartmut Neven, Michael Newman, Jiun How Ng, Anthony Nguyen, Murray Nguyen, Chia-Hung Ni, Murphy Yuezhen Niu, Thomas E. O'Brien, William D. Oliver, Alex Opremcak, Kristoffer Ottosson, Andre Petukhov, Alex Pizzuto, John Platt, Rebecca Potter, Orion Pritchard, Leonid P. Pryadko, Chris Quintana, Ganesh Ramachandran, Matthew J. Reagor, John Redding, David M. Rhodes, Gabrielle Roberts, Eliott Rosenberg, Emma Rosenfeld, Pedram Roushan, Nicholas C. Rubin, Negar Saei, Daniel Sank, Kannan Sankaragomathi, Kevin J. Satzinger, Henry F. Schurkus, Christopher Schuster, Andrew W. Senior, Michael J. Shearn, Aaron Shorter, Noah Shutty, Vladimir Shvarts, Shraddha Singh, Volodymyr Sivak, Jindra Skruzny, Spencer Small, Vadim Smelyanskiy, W. Clarke Smith, Rolando D. Somma, Sofia Springer, George Sterling, Doug Strain, Jordan Suchard, Aaron Szasz, Alex Sztein, Douglas Thor, Alfredo Torres, M. Mert Torunbalci, Abeer Vaishnav, Justin Vargas, Sergey Vdovichev, Guifre Vidal, Benjamin Villalonga, Catherine Vollgraff Heidweiller, Steven Waltman, Shannon X. Wang, Brayden Ware, Kate Weber, Travis Weidel, Theodore White, Kristi Wong, Bryan W. K. Woo, Cheng Xing, Z. Jamie Yao, Ping Yeh, Bicheng Ying, Juhwan Yoo, Noureldin Yosri, Grayson Young, Adam Zalcman, Yaxing Zhang, Ningfeng Zhu, and Nicholas Zobrist. 2024. Quantum error correction below the surface code threshold. *Nature* 638, 8052 (Dec. 2024), 920–926. doi:10.1038/s41586-024-08449-y

[2] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.

[3] Malcolm Carroll, Sami Rosenblatt, Petar Jurcevic, Isaac Lauer, and Abhinav Kandala. 2022. Dynamics of superconducting qubit relaxation times. *npj Quantum Information* 8, 1 (2022), 132.

[4] Jerry M. Chow, Jay M. Gambetta, Easwar Magesan, David W. Abraham, Andrew W. Cross, B R Johnson, Nicholas A. Masluk, Colm A. Ryan, John A. Smolin, Srikanth J. Srinivasan, and M Steffen. 2014. Implementing a strand of a scalable fault-tolerant quantum computing fabric. *Nature Communications* 5, 1 (June 2014). doi:10.1038/ncomms5015

[5] Niel de Beaudrap and Dominic Horsman. 2020. The ZX calculus is a language for surface code lattice surgery. *Quantum* 4 (2020), 218.

[6] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John Van De Wetering. 2020. Graph-theoretic simplification of quantum circuits with the ZX-calculus. *Quantum* 4 (2020), 279.

[7] Austin G Fowler and Craig Gidney. 2018. Low overhead quantum computation using lattice surgery. *arXiv preprint arXiv:1808.06709* (2018).

[8] Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. 2012. Surface codes: Towards practical large-scale quantum computation. *Physical Review A—Atomic, Molecular, and Optical Physics* 86, 3 (2012), 032324.

[9] Lena Funcke, Tobias Hartung, Karl Jansen, Stefan Kühn, Paolo Stornati, and Xiaoyang Wang. 2022. Measurement error mitigation in quantum computers through classical bit-flip correction. *Physical Review A* 105, 6 (2022), 062404.

[10] F Pelayo García de Arquer, Dmitri V Talapin, Victor I Klimov, Yasuhiko Arakawa, Manfred Bayer, and Edward H Sargent. 2021. Semiconductor quantum dots: Technological progress and future challenges. *Science* 373, 6555 (2021), eaaz8541.

[11] Craig Gidney. 2024. Inplace access to the surface code y basis. *Quantum* 8 (2024), 1310.

[12] Craig Gidney and Martin Ekerå. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* 5 (2021), 433.

[13] Daniel Herr, Franco Nori, and Simon J Devitt. 2017. Lattice surgery translation for quantum computation. *New Journal of physics* 19, 1 (2017), 013034.

[14] Dominic Horsman, Austin G Fowler, Simon Devitt, and Rodney Van Meter. 2012. Surface code quantum computing by lattice surgery. *New Journal of Physics* 14, 12 (2012), 123011.

[15] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. Quantum computing with Qiskit. *arXiv preprint arXiv:2405.08810* (2024).

[16] Aleks Kissinger and John Van De Wetering. 2019. PyZX: Large scale automated diagrammatic reasoning. *arXiv preprint arXiv:1904.04735* (2019).

[17] Aleks Kissinger and John van de Wetering. 2019. Reducing T-count with the ZX-calculus. *arXiv preprint arXiv:1903.10477* (2019).

[18] Morten Kjaergaard, Mollie E Schwartz, Jochen Braumüller, Philip Krantz, Joel I-J Wang, Simon Gustavsson, and William D Oliver. 2020. Superconducting qubits: Current state of play. *Annual Review of Condensed Matter Physics* 11, 1 (2020), 369–395.

[19] Sebastian Krinner, Nathan Lacroix, Ants Remm, Agustin Di Paolo, Elie Genois, Catherine Leroux, Christoph Hellings, Stefania Lazar, Francois Swiadek, Johannes Herrmann, Graham J. Norris, Christian Kraglund Andersen, Markus Müller, Alexandre Blais, Christopher Eichler, and Andreas Wallraff. 2022. Realizing repeated quantum error correction

in a distance-three surface code. *Nature* 605, 7911 (May 2022), 669–674. doi:10.1038/s41586-022-04566-8

[20] Abtin Molavi, Amanda Xu, Swamit Tannu, and Aws Albarghouthi. 2025. Dependency-aware compilation for surface code quantum architectures. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 57–84.

[21] Michael A Nielsen and Isaac L Chuang. 2010. *Quantum computation and quantum information*. Cambridge university press.

[22] Tom Peham, Lukas Burgholzer, and Robert Wille. 2022. Equivalence checking of quantum circuits with the ZX-calculus. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 12, 3 (2022), 662–675.

[23] Maximilian Schlosshauer. 2019. Quantum decoherence. *Physics Reports* 831 (2019), 1–57.

[24] Sayam Sethi and Jonathan Mark Baker. 2025. RESCQ: Realtime Scheduling for Continuous Angle Quantum Error Correction Architectures. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1028–1043.

[25] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. t| ket>: a retargetable compiler for NISQ devices. *Quantum Science and Technology* 6, 1 (2020), 014003.

[26] Daniel Bochen Tan, Murphy Yuezhen Niu, and Craig Gidney. 2024. A sat scalpel for lattice surgery: Representation and synthesis of subroutines for surface-code fault-tolerant quantum computing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 325–339.

[27] TQEC contributors. 2025. TQEC. https://github.com/tqec. Accessed: 2025-08-20.

[28] George Watkins, Hoang Minh Nguyen, Keelan Watkins, Steven Pearce, Hoi-Kwan Lau, and Alexandru Paler. 2024. A high performance compiler for very large scale surface code computations. *Quantum* 8 (2024), 1354.

[29] Ed Younis, Costin C Iancu, Wim Lavrijsen, Marc Davis, and Ethan Smith. 2021. *Berkeley quantum synthesis toolkit (bqskit) v1*. Technical Report. Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA (United States).