HiLight: A Comprehensive Framework for High-Performance and Lightweight Scalability in Surface Code Communication

Sunghye Park, Dohun Kim and Seokhyeong Kang
Pohang University of Science and Technology, Pohang, Republic of Korea
{shpark96,dohunkim,shkang}@postech.ac.kr

ABSTRACT

In pursuing fault-tolerant quantum computing (FTQC), the surface code (SC) serves as a key quantum error correction protocol. The double-defect mode of the SC enables long-range two-qubit communication via braiding. However, intersecting braiding paths create communication bottlenecks, leading to increased circuit latency.

In this paper, we introduce **HiLight**, an optimization framework designed for enhancing SC communication. HiLight integrates qubit-mapping strategies with program- and hardware-level optimizations, providing high-performance and lightweight scalable solutions. Featuring SWAP-less initial placement, HiLight utilizes qubit-proximity and pattern matching to minimize path congestion. In routing, HiLight maximizes gate parallelism and speeds up path selection through fast gate-ordering and braiding path-finding. The combined optimizations improve latency and resource utilization. Compared with the state-of-the-art approach, HiLight achieves a remarkable reduction in latency and runtime by 43.5% and 91.9%, respectively, signifying its potential to advance the FTQC era.

KEYWORDS

Double-Defect Surface Code (SC), Fault-Tolerant, Qubit Mapping

1 INTRODUCTION

Quantum computing (QC) is an emerging paradigm that utilizes the principles of quantum mechanics. Using qubits that exist in superposition states, QC enables high computing speeds through concurrent calculations. Despite recent advances enabling hundreds of qubits [1, 2], they still operate at the noisy intermediate-scale quantum (NISQ) level. This highlights the progress for fault-tolerant QC (FTQC) [3, 4], which will be achieved by quantum error correction (QEC) [5–7]. QEC encodes many physical qubits into fewer error-resilient logical qubits, ensuring reliable and practical QC.

The surface code (SC) [8] is a prominent QEC protocol that utilizes a 2D lattice structure for scalable qubit expansion and a high error threshold. This work emphasizes the long-range two-qubit (2Q) communication facilitated by the double-defect SC [8–12]. With this mode, a logical qubit is formed by physical qubit clusters with two intentional defects. While a single-qubit (1Q) gate only affects these defects, a 2Q gate influences a qubit pair and nearby logical qubits in between. 2Q communication relies on *braiding* [8, 13] to create entangled states via space–time topological manipulation. Despite braiding exhibiting constant latency regardless of path length, optimizing SC communication (Fig. 1) is crucial to prevent path intersections for different gates and enhance path parallelism.

Optimizing SC communication involves solving the qubit-mapping problem in FTQC to avoid communication congestion. During initial placement, program qubits (q_i) of a quantum circuit are assigned to suitable hardware qubits (Q_i) to enhance the parallelism of braiding paths. In the routing stage, appropriate gate ordering and path selection are crucial to enable the efficient execution of 2Q gates without path overlaps. To this end, Hua *et al.* [14] introduced an automatic framework for scheduling the braiding paths. However, their placement depends on the addition of SWAPs for layout changes, increasing gate overheads. Furthermore, the recurrent graph generation for gate ordering leads to longer runtimes.

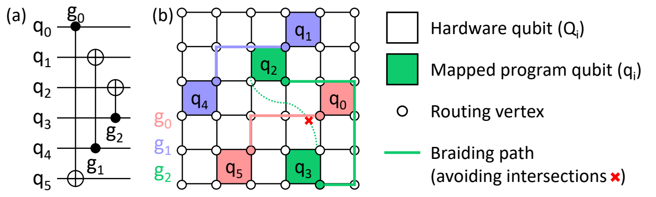


Figure 1: Overall flow of the surface code (SC) communication problem: (a) Example quantum circuit. (b) Illustration of braiding-based two-qubit gate execution on an SC hardware grid.

Along with addressing the qubit-mapping, it is important to consider further optimization processes. For example, optimization strategies can be implemented for individual input circuit programs and target hardware configurations. These processes are vital for enhancing communication efficiency alongside the mapping.

In this work, we propose **HiLight**, a high-performance and lightweight scalable framework for enhanced SC communication. HiLight conducts mapping-level optimizations to enhance braiding parallelism while minimizing latency and runtime. To further improve system efficiency, HiLight integrates program and hardware optimizations before mapping-level optimization. The main contributions are as follows.

- Utilizing our qubit-proximity method, we establish an efficient SWAP-less initial placement, alleviating communication bottlenecks and avoiding potential gate overheads. Additionally, we implement pattern matching to further reduce runtime.
- In the routing stage, we leverage fast gate ordering to maximize the parallel execution of braiding, thereby minimizing latency. Furthermore, we integrate rapid braiding path-finding methods to promptly identify the optimal path with high scalability.
- Our program-level optimization enhances braiding parallelism, further reducing latency. Hardware-level optimization is applied to improve resource efficiency for practical SC implementation.
- Compared to [14], HiLight significantly reduces latency by 43.5% and runtime by 91.9%. In summary, HiLight streamlines the FTQC era through comprehensive SC communication optimization.

2 BACKGROUND

2.1 Double-Defect SC and Braiding Process

SC employs Z- and X-stabilizer measurements to efficiently detect and correct errors. Among its modes, the double-defect SC [8–12] enhances qubit connectivity. It introduces two intentional defects in the SC lattice by disabling two stabilizers. By forming the operator chains through these defects, logical Z- or X-operators can be established. These logical operators commute with all stabilizers in the array, satisfying the critical anti-commutation requirement by converting the double-defect into a logical qubit.

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3655980

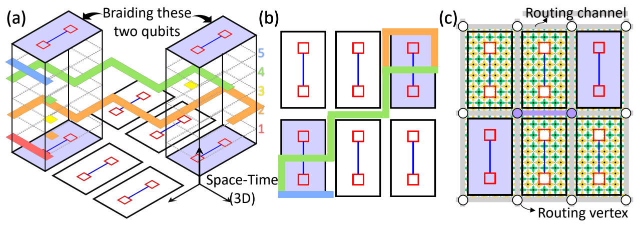


Figure 2: Overview of the braiding process: (a) Topological transformation within the 3D domain. (b) Conversion of (a) into the 2D domain. (c) The double-defect SC lattice corresponds to (b).

Within the double-defect SC, 2Q gates such as CX are executed through a 3D topological transformation called *braiding* [8, 13]. Fig. 2a illustrates a vertical time-stacked braiding process between light-purple colored qubits. This process involves a consistent five-step transformation between defects for encoded quantum data manipulation. When transitioning from 3D to 2D, the braiding process is abstracted as a pathway formation between qubits (Figs. 2b and 2c). This path follows five uniform steps that ensure constant latency, regardless of the distance between qubits. Each tile refers to a hardware qubit consisting of physical qubits in the SC grid and capable of mapping the program qubit. Each tile has four edges, called *routing channels*, and four vertices, called *routing vertices*, facing other qubit tiles. To create 2D braiding paths, two qubits of 2Q gates must be routed using their own routing vertex, choosing one path from a total of 16 possible paths for routing.

2.2 SC Communication Problem

The goal of the problem for a given circuit on an SC hardware grid is to execute the circuit while maximizing parallel braiding without intersections (Fig. 1). For example, with an arbitrary layout, the pink and purple paths can be executed in parallel without crossing. However, the dashed green intersects with the pink, requiring execution in the next cycle. In contrast, a solid green path allows all gates to function together. This highlights the importance of optimizing SC communication to minimize path overlaps and latency.

SC communication optimization involves mapping program qubits to hardware qubits while addressing inherent hardware constraints in FTQC. Similar to traditional mapping for NISQ, it adopts a two-step heuristic: *initial placement* and *routing*. The initial placement aims to reduce congestion, while the subsequent routing stage focuses on executing 2Q gates, emphasizing parallel braiding. Efficiently solving this problem can significantly improve the performance and scalability of SC-based FTQC platforms.

2.3 Previous Work

Recent studies have addressed the mapping problem in the SC context. Javadi-Abhari *et al.* [13] transformed the SC problem into a 2D routing problem, considering two distinct SC modes: *lattice-surgery* and *double-defect*. Lao *et al.* [15] introduced a mapping strategy for lattice-surgery-based circuits, whereas Hua *et al.* [14] presented a scheduling method for braiding paths in the double-defect SC. Ding *et al.* [16] proposed an optimization technique for *magic-states*.

Since this paper focuses on the double-defect SC communication, we consider AutoBraid [14] as a baseline. AutoBraid introduced iterative graph-partitioning to place frequently used qubits closely and later adjusted the layout using SWAP gates. However, this results in additional gates, and determining the timing for SWAP insertion is tricky. Although their gate-ordering method utilizes local parallelism between gates, it requires recurrent graph constructions, resulting in reduced scalability and increased runtime. Motivated by these challenges, we strive to provide high-performance, lightweight, and SWAP-less mapping solutions in the SC context.

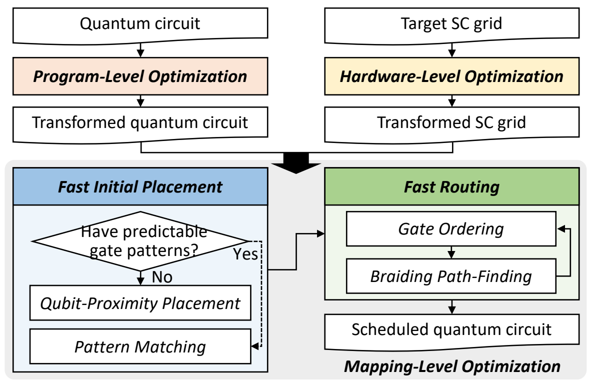

Quantum circuit
Target SC grid
Program-Level Optimization
Hardware-Level Optimization
Transformed quantum circuit
Transformed SC grid
Fast Initial Placement
Fast Routing
Have predictable gate patterns?
Yes
No
Gate Ordering
Braiding Path-Finding
Qubit-Proximity Placement
Pattern Matching
Scheduled quantum circuit
Mapping-Level Optimization

Figure 3: Overall flow of the HiLight framework.

Algorithm 1: Fast Initial Placement with Qubit-Proximity Placement

```
inputs  : Quantum circuit (circ), Grid hardware (grid)
output  : Initial layout ($\pi$)
Initialize $\pi$ to -1 /* To represent the unmapped qubit, where $\pi$[q]=-1 */
circGraph ← MakeAdjacencyMatix(circ)
circQueue ← SortByMaxDegree(circGraph)
while !circQueue.Empty & UnmappedQubits=$\emptyset$ do
    q ← circQueue.PopFront
    Neighbors[q] ← SortByMaxDegree(circQueue[q])
    if q is the first element in circQueue then
        $\pi$[q] ← CalculateCenter(grid)
    else if $\pi[q]$ = -1 then
        refLoc ← FindMappedLoc(Neighbors[q])
        $\pi$[q] ← FindClosestUnmappedLoc(refLoc)
    adjacentQubit ← FindUnmappedQubit(Neighbors[q])
    adjacentLoc ← FindUnmappedNeighborPositions($\pi$[q])
    for i=0 to MinSize(adjacentQubit, adjacentLoc) do
        $\pi$[adjacentQubit[i]] ← adjacentLoc[i]
return $\pi$
```

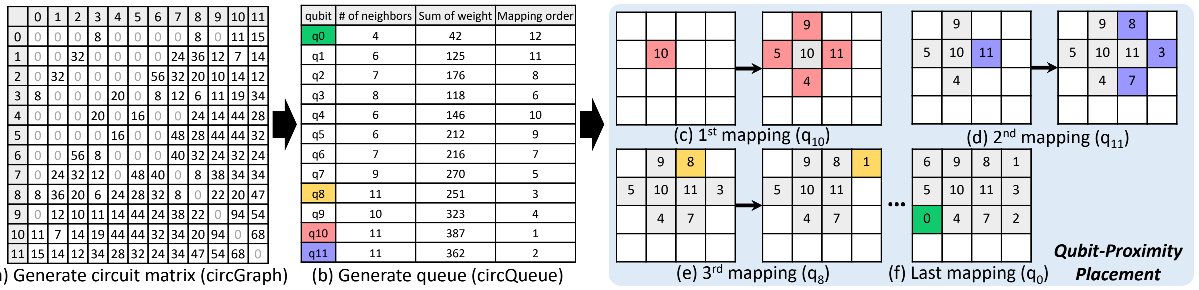
3 HILIGHT FRAMEWORK

Our **HiLight** framework (Fig. 3) focuses on enhancing SC communication through three stages: 1) Mapping-level optimization offers fast initial placement and routing to maximize braiding parallelism. Our high-performance mapping ensures efficient 2Q gate execution with minimized latency and runtime. 2) At the program-level optimization, we employ quantum circuit optimization (QCO) techniques to enhance algorithm performance, thus reducing communication overhead. 3) Hardware-level optimization emphasizes resource efficiency and scalability, particularly in the SC grid allocation. Through these strategies, we provide lightweight scalable solutions for SC computations, paving the way for practical FTQC.

3.1 Fast Initial Placement

The proposed initial placement focuses on alleviating communication bottlenecks in the early stages. By employing a graph-inspired qubit-proximity placement (Alg. 1), we establish shorter and direct braiding paths, promoting parallel execution and thus decreasing circuit latency. We also utilize pattern matching to accelerate the generation of the placement. The resulting layout is used as input for the routing step and is maintained throughout the routing without requiring SWAPs, further reducing possible gate overheads.

3.1.1 Qubit-Proximity Placement. We strategically position qubit pairs closely on the grid to ease communication bottlenecks. Inspired by NISQ mappings [17], we leverage isomorphic parts of the circuit interaction graph and hardware coupling graph to optimize initial placement. This layout shortens the lengths of the braiding path and reduces intersections, thus boosting parallelism. However, directly applying these previous methods significantly increases runtime for larger FTQC circuits. To address this, we introduce a



	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	8	0	0	0	0	8	0	11	15
1	0	0	32	0	0	0	0	24	36	12	7	14
2	0	32	0	0	0	0	56	32	20	10	14	12
3	8	0	0	0	20	0	8	12	6	11	19	34
4	0	0	0	20	0	16	0	0	24	14	44	28
5	0	0	0	0	16	0	0	48	28	44	44	32
6	0	0	56	8	0	0	0	40	32	24	32	24
7	0	24	32	12	0	48	40	0	8	38	34	34
8	8	36	20	6	24	28	32	8	0	22	20	47
9	0	12	10	11	14	44	24	38	22	0	94	54
10	11	7	14	19	44	44	32	34	20	94	0	68
11	15	14	12	34	28	32	24	34	47	54	68	0

(a) Generate circuit matrix (circGraph)

qubit	# of neighbors	Sum of weight	Mapping order
q0	4	42	12
q1	6	125	11
q2	7	176	8
q3	8	118	6
q4	6	146	10
q5	6	212	9
q6	7	216	7
q7	9	270	5
q8	11	251	3
q9	10	323	4
q10	11	387	1
q11	11	362	2

(b) Generate queue (circQueue)

Figure 4: Qubit-proximity placement procedure for initial placement stage: (a) Example quantum circuit ('*sqrt8_260*') represented as a matrix. (b) Placement order determined by the degree of each node and sum of weights. (c) Initial placement with the first qubit at the grid center, followed by neighboring nodes in four cardinal positions. (d)-(e) Steps repeated. (f) Completion when all program qubits are positioned.

unique FTQC-oriented method (Fig. 4). We adopt a matrix representation for circuit interactions and utilize grid indices for hardware. By avoiding the traditional heavier node-edge-based graph creation, we ensure enhanced scalability while delivering an efficient layout.

In Alg. 1, we use a matrix called circGraph to represent the number of CX gates between qubit pairs. The algorithm sequentially positions unmapped qubits on the grid based on their degree and adjacency in descending order (Lines 4–15). The first qubit in the circuit queue, circQueue, occupies the grid center (Lines 7–8), while other qubits are placed based on adjacency (Lines 9–11). Then, the neighbors are mapped in four cardinal directions (Lines 12–15). When determining each position, we leverage the row/column information of the grid to reduce runtime. After mapping all qubits, the resulting initial layout is obtained (Line 16). Preserving the layout without SWAP insertions fully leverages the superior connectivity of the double-defect SC, eliminating extra gates and latency.

3.1.2 *Pattern Matching.*

We utilize predefined patterns as an initial layout, particularly for benchmarks with repetitive or predictable gate operations, to reduce runtime and latency. The pattern-matching strategy efficiently maximizes gate parallelism, especially in scalable benchmarks, offering high scalability and applicability.

First, we utilize a linear pattern for the optimal initial layout in algorithms with a linear interaction graph, connecting qubit 0 to the program qubit size n. This maximizes parallel execution of CX gates in circuits such as the 1D *Ising model*, *Graph state*, *Greenberger-Horne-Zeilinger (GHZ) state*, *Variational Quantum Eigensolver (VQE) state*, and *W-state*. Next, for algorithms involving dynamic qubit interactions, random mapping enhances gate parallelism by distributing qubit pairs. For example, this approach is effective for *quantum Fourier Transform (QFT)*, which forms a complete interaction graph with the gate set $S_i = \{CX(i, (i\text{-}1)\text{-}(k\text{-}1))\}$, where i ranges from 1 to n and k ranges from 1 to i-1. However, for non-scalable benchmarks, the proposed qubit-proximity placement surpasses pattern matching in addressing communication bottlenecks.

3.2 Fast Routing

During routing, 2Q gates are sequentially executed while avoiding path intersections. Our approach emphasizes improving parallelism through iterative gate ordering and efficient braiding path-finding (Fig. 5). The proposed gate-ordering method capitalizes on the inherent parallelism in quantum programs. Furthermore, braiding path-finding techniques are employed to quickly identify optimal paths. In Alg. 2, we construct a gate list called circList for each program qubit, streamlining the gate-ordering process (Lines 2–4). In an iterative manner, 2Q gates are processed using valid and efficient braiding paths with high scalability (Lines 7–11). This method effectively alleviates communication bottlenecks, contributing to optimizing the latency and runtime of the overall systems.

Algorithm 2: Fast Routing with Proposed Gate Ordering and Path-Finding

```
   inputs  :Quantum circuit (circ), Grid hardware (grid), Initial layout ($\pi$)
   output  :Final circuit (Fcirc)
 1 Initialize FrontList to -1 /* To wait for both qubits of 2Q gates to be front */
 2 circList ← MakeListOfGates(circ)
 3 do
 4     $G_{single}$, $G_{two}$ ← GateOrdering(circList, FrontList)
 5     Fcirc ← Execute1Qgates($G_{single}$, circList)
 6     isPassed ← False
 7     foreach $g_{two} \in G_{two}$ do
 8         if !PathFinding.Empty then
 9             Fcirc ← Execute2Qgate($g_{two}$, circList)
10             isPassed ← UpdateOccupiedPath(path, grid)
11             FrontList.Remove($g_{two}$.control, $g_{two}$.target)
12 while !CheckLoopEnd(circList);
13 return Fcirc
14 Function PathFinding($g_{two}$, grid, $\pi$, isPassed, path):
15     $c_{control}$, $c_{target}$ ← FindQubitGridCell($g_{two}$, grid, $\pi$)
16     $p_{control}$, $p_{target}$ ← FindMinManhattanDistPoint($c_{control}$, $c_{target}$, grid)
17     path ← FindValidBraidingPath($p_{control}$, $p_{target}$, grid, isPassed)
```

3.2.1 *Gate Ordering.*

To enhance parallelism, the gate-ordering stage determines the most efficient gate sequence at each time by adapting the as-soon-as-possible approach. It constantly tracks the execution status of each qubit using FrontList (Line 1), facilitating the prompt scheduling of subsequent gates. This simplifies the selection of the best braiding path, thus reducing the runtime. We utilize circList to sequence the execution of gates (Line 2). 1Q gates are immediately executable after gate ordering (Line 5), while 2Q gates need to be cross-referenced. If only one qubit is currently executable, the corresponding qubit is pushed to FrontList and not immediately executable. Otherwise, the corresponding 2Q gates are updated to currently executable G_{two} (Line 7). After checking all program qubits in the circList, we obtain a list of *ready-to-execute* gates (Line 4). This approach harnesses the inherent parallelism of the circuit, significantly reducing the total idle time, circuit latency, and communication bottlenecks. With its speed and efficiency, it is possible to seamlessly integrate with other QCO techniques.

3.2.2 *Braiding Path-Finding.*

In this phase, 2Q gates are executed using braiding communication (Lines 7–11). The goal is to identify the most efficient path and thereby reduce communication bottlenecks. If we exploit the optimal gate order, the shortest path between qubits can be an optimal path to minimize routing congestion. However, given the potential 16 paths between qubits, finding the optimal path with speed becomes the next challenge. To address this, our path-finding algorithm (Lines 14–17) streamlines the process by identifying grid points for the control and target qubits of a gate with minimal Manhattan distance. We then utilize the A* search algorithm to pinpoint the near-optimal braiding path, while extremely reducing runtime. Integrating this lightweight method with the optimized gate order collectively reduces the latency.

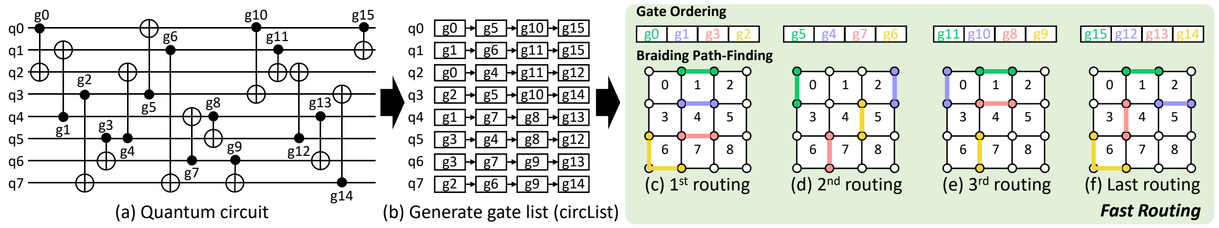


Figure 5: Proposed routing procedure with our gate ordering and braiding path-finding: (a) Example quantum circuit. (b) Quantum circuit represented as lists of gates. (c) Gate execution order determined using (b), followed by braiding path-finding search. (d)-(e) Steps repeated. (f) Completion when all quantum gates are routed.

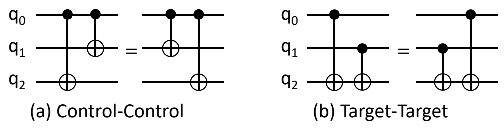


Figure 6: Rules for quantum circuit optimization (QCO) at the program level. Exchange of sequential CX gates sharing (a) the same control qubit and (b) the same target qubit.

3.3 Program-Level Optimization

To further reduce latency, QCO techniques are integrated at the program level. The QCO optimizes circuit efficiency through techniques such as gate reordering, compression, and cancellation, thereby reducing the overall cost of quantum circuits while maintaining their functionality. In this work, we specifically utilize gate-reordering of multiple CX gates for parallel braiding (Fig. 6).

In our strategy, we merge the QCO and gate-list, called circList, generation procedure outlined in Alg. 2. Instead of sequentially appending gates to a list, our QCO-enabled algorithm examines if a current CX gate shares control or target qubits with previously listed gates. Then, the possibilities of reducing depth are assessed to identify optimal gate orderings. If a depth reduction is not immediately viable, multiple branches are explored to find the best QCO option, thus eliminating unnecessary back-tracking. In this manner, we significantly curtail unnecessary searches, leading to reduced runtime. The resulting optimal gate order is then utilized in the qubit-mapping-level optimization discussed in Section 3.2.

3.4 Hardware-Level Optimization

In the SC communication, hardware resource utilization during braiding operations is quantified by the metric *ResUtil* as follows:

$$ResUtil = \frac{\sum Braiding\ path\ length}{Gird\ size \times Latency} = \frac{\sum_{k=1}^{L}\sum_{i=1}^{i_k} Length(b_i)}{(M \times M) \times L}, \quad (1)$$

where M is $\lceil\sqrt{n}\rceil$ based on the program qubit size n, i_k denotes the number of braidings in the k-th layer within a total latency L, and b_i represents the i-th braiding. This metric quantifies the total length of braiding paths in relation to the total number of available vertices within the grid, considering the entire latency.

Optimizing resource utilization with a balance is crucial to avoid routing congestion and ensure scalability. Furthermore, we have to consider real-world SC hardware, which allocates a portion of the grid for critical FTQC components, such as the *magic-state factory* [18] (Fig. 7). Considering these factors, this work first encapsulates the factory unit as a singular and non-braiding logical qubit. Then, resource allocation is optimized by focusing on two geometric structures: an $M \times M$ square and a slightly diminished $M \times (M-1)$ rectangular grid. For example, the 4×4 square lattice shown in Fig. 4 can be restructured into a 4×3 grid to execute the circuit. This lattice reconstruction balances resource utilization with reduced computational demands. It also enhances routing flexibility and preserves space for essential FTQC components.

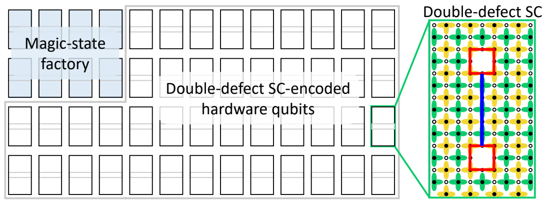


Figure 7: Tiled architecture for the double-defect SC. For such configurations, a portion of the grid is reserved for the vital FTQC components, the *magic-state factory*.

4 EXPERIMENTAL RESULTS

The **HiLight** framework was implemented in *C++* and compiled with GCC version 8.1.0. Experiments were conducted using benchmark circuits [19–22], which encompass basic and advanced applications (see Table 1 for details). For the target hardware, a square or rectangular grid was used. We evaluated the performance using these three metrics: 1) *Latency* representing the routing steps for the circuit execution, 2) *Runtime* referring to the total framework execution time, and 3) *Resource utilization* indicated by Eq. 1.

4.1 Fast Initial Placement Analysis

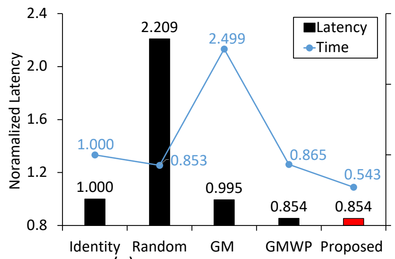
We evaluated the efficiency of our initial placement against five layouts (Fig. 8a): **Identity**, where program qubits are matched to qubit tiles with identical indices; **Random**, where layouts were randomly generated, averaged after 100 trials; **GM**, a previous graph-inspired method [17]; **GMWP**, where GM was combined with our pattern matching; and **Proposed**, our proposed method (see Section 3.1).

The experimental results highlight the superior performance of our initial placement in reducing communication bottlenecks, runtime, and latency. Compared with Identity mapping, both Random and GM methods achieved some reductions in either latency or runtime, with trade-offs. Random mapping reduced runtime by 14.7% but increased latency by 2.2×; whereas GM reduced latency by 0.5% but increased runtime by 2.5×. By adding pattern matching to the GM method, both runtime and latency improved, but not optimally. By contrast, our method significantly reduces runtime by 45.7% and latency by 14.6%, demonstrating its ability to quickly generate an efficient and scalable layout suitable for FTQC.

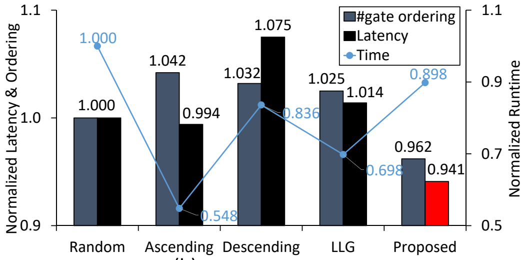
4.2 Fast Routing Analysis

To validate our gate-ordering method for enhancing parallelism, we compared it with four other methods (Fig. 8): **Random**, where ordering is based on a random selection with an average of over 100 trials. **Ascending & Descending**, where executable gates are sorted by the corresponding order; **LLG**, which adopts the path-finding from [14]; and **Proposed**, our method (see Section 3.2.1). Here, the gate-ordering algorithm was invoked when there were more than four executable 2Q gates, according to the analysis in [14].

As seen in Ascending and Descending, prioritizing gate execution based on their index to leverage inherent parallelism yielded superior results. Although LLG ordering is ideal for parallel braiding, it slightly increases latency when combined with our routing



(a) **Initial Placement Methods**



(b) **Gate-ordering Methods**

Initial placement		Routing		Normalized Results	
Qubit-Proximity placement	Pattern matching	Gate ordering	Fast braiding	Latency	Runtime
Identity	-	Ours	Ours	1.114	1.295
GM	-	Ours	Ours	1.088	4.246
Ours	-	Ours	Ours	1.046	1.212
Ours	Ours	Ours	Ours	**1.000**	**1.000**
Ours	Ours	Ours	-	0.999	5.181
Ours	Ours	LLG	Ours	1.020	1.331

(c) **Comparison Results of the Individual Mapping Steps**

Figure 8: Experimental results for mapping step analysis: (a) Initial placement according to the generation methods. (b) Routing according to the gate-ordering methods. (c) Summary of comparison results for the individual qubit-mapping steps.

Table 1: Comparison of latency, runtime, and resource utilization at the mapping level.

Benchmark					autobraid-sp [14]			autobraid-full [14]			Ours (hilight-map)		
type	function	name	n	g	latency	runtime [s]	ResUtil	latency	runtime [s]	ResUtil	latency	runtime [s]	ResUtil
Building blocks	Compare input	*4gt11_82*	5	20	**18**	0.01	0.14	**18**	0.02	0.14	**18**	**0.00**	**0.11**
		4gt5_75	5	48	36	0.01	0.24	36	0.01	0.28	**33**	**0.00**	**0.13**
	ALU by Gupta	*alu-v0_26*	5	48	37	0.02	0.18	37	0.01	0.18	**35**	**0.00**	**0.12**
	Bit adder	*rd32_270*	5	46	36	0.01	0.17	36	0.12	0.18	**35**	**0.00**	**0.11**
	Square root	*sqrt8_260*	12	1.69K	1,305	0.10	0.15	1,301	0.12	0.17	**1,123**	**0.01**	**0.09**
		squar5_261	13	1.12K	852	0.07	0.17	853	0.20	0.17	**720**	**0.01**	**0.08**
		square_root_7	15	4.07K	2,999	0.20	0.17	2,998	1.29	0.14	**2,521**	**0.02**	**0.09**
	Unstructured reversible function	*urf1_278*	9	32.8K	24,943	1.41	0.25	24,934	**0.53**	0.23	**22,315**	0.68	**0.15**
		urf2_277	8	12.3K	9,302	0.50	0.23	9,327	3.21	0.25	**8,316**	**0.16**	**0.15**
		urf5_158	9	92.5K	71,335	3.30	0.24	71,350	1.19	0.22	**64,752**	**2.55**	**0.14**
		urf5_280	9	29.5K	22,369	1.27	0.26	22,342	**0.03**	0.22	**19,895**	0.54	**0.15**
Real world applications	Quantum Fourier Transform	*QFT*	10	100	**37**	0.01	0.48	40	0.05	0.41	38	**0.00**	**0.33**
			16	256	80	0.03	0.69	**72**	0.06	0.67	**72**	**0.00**	**0.48**
			100	10.0K	1,458	1.44	**0.62**	1,297	2.00	0.64	**1,120**	**0.26**	0.64
			150	22.5K	2,745	5.46	**0.54**	2,649	5.80	0.55	**2,136**	**0.99**	0.62
			200	40.0K	4,419	13.58	**0.52**	4,047	12.82	0.55	**3,420**	**2.42**	0.60
			400	0.16M	15,267	127.09	0.48	15,232	124.28	**0.45**	**11,089**	**21.05**	0.57
			500	0.25M	21,138	310.04	0.46	20,511	254.80	**0.45**	**16,012**	**46.87**	0.55
	Bernstein Vazirani	*BV*	10	28	**9**	0.01	0.19	**9**	0.03	0.17	**9**	**0.00**	**0.06**
			100	299	**99**	0.04	0.16	**99**	0.07	0.09	**99**	**0.00**	**0.03**
			150	449	**149**	0.05	0.10	**149**	0.12	0.08	**149**	**0.01**	**0.02**
			200	599	**199**	0.06	0.09	**199**	0.13	0.06	**199**	**0.01**	**0.02**
	Counterfeit-Coin Finding	*CC*	11	20	**10**	**0.00**	0.11	**10**	0.02	0.16	**10**	**0.00**	**0.06**
			18	34	**17**	0.01	0.09	**17**	0.03	0.09	**17**	**0.00**	**0.04**
			100	198	**99**	0.02	0.08	**99**	0.12	0.05	**99**	**0.00**	**0.03**
			200	398	**199**	0.04	0.05	**199**	0.15	0.03	**199**	**0.01**	**0.02**
			300	598	**299**	0.07	0.04	**299**	0.19	0.03	**299**	**0.01**	**0.02**
	1D-Ising Model	*Isimg Model*	10	200	**20**	0.03	0.84	**20**	0.01	0.41	**20**	**0.00**	**0.28**
			13	263	30	0.02	0.93	**20**	0.02	0.56	**20**	**0.00**	**0.38**
			16	326	35	0.03	1.00	**20**	0.02	0.69	**20**	**0.00**	**0.47**
			500	5.29K	14	0.23	1.10	**4**	0.19	0.94	**4**	**0.05**	**0.47**
			1,000	11.0K	8	0.84	1.78	**4**	0.51	0.98	**4**	**0.11**	**0.49**
	Binary Welded Tree	*BWT*	179	265	93	0.05	0.10	90	0.28	0.05	**39**	**0.00**	**0.04**
			240	365	113	0.07	0.14	96	0.32	**0.05**	**40**	**0.01**	0.06
	Quantum Approximate Optimization Alg.	*QAOA*	100	2.72K	**12**	0.08	**0.45**	247	0.74	1.55	**12**	**0.01**	**0.45**
	Shor's Algo.	*Shor's*	471	36.6K	15,717	2.67	0.04	15,648	4.71	0.03	**7,361**	**0.40**	**0.02**
Normalized to Ours					1.201	7.407	1.679	1.771	20.058	1.559	**1.000**	**1.000**	**1.000**

• "autobraid-sp" uses only a stack-based pathfinder, while "-full" uses both a pathfinder and layout optimization from [14]. • "hilight-map" integrates our routing method with our initial placement methods. • "n" and "g" indicate the number of qubits and gates for each benchmark. • The *QFT* results for our method are averaged over 100 experiments with random initial mapping, as discussed in Section 3.1.2 for the proposed pattern-matching. • The best results are highlighted in **bold**. • Benchmark descriptions: 1) The hidden string of *BV* is all 1s. 2) *QAOA* is generated with 180 alternating ZZs.

approach, which selects only a single minimum path to reduce runtime. The proposed gate-ordering achieved the lowest latency with a 5.9% reduction compared to Random ordering, with a slight increase in runtime attributed to the execution of more gates. These results demonstrate that our gate-ordering enhances braiding parallelism and reduces circuit latency, even when using the same braiding path-finding and initial placement method. The impact of each step in our qubit-mapping method is illustrated in Fig. 8c.

4.3 Mapping-Level Optimization Results

To demonstrate the advantages of the proposed mapping, we compared it with the current state-of-the-art framework, AutoBraid [14]. To ensure a fair comparison with AutoBraid, we executed their open-source release in the same server setting. Table 1 shows that our approach outperformed significantly in terms of latency, runtime, and resource utilization. We achieved substantial latency reductions by 16.8% and 43.5%, compared to "autobraid-sp" and "autobraid-full," consistently across all benchmarks. This was due to our effective initial placement strategy, which avoided extra latency from potential SWAP gates. Furthermore, capitalizing on inherent parallelism in gate ordering achieved efficient latency reductions.

In addition, our strategy resulted in substantial reductions in runtime, which decreased by 86.5% and 95.0% compared to AutoBraid. This is possible because we leverage the efficient graph structures for qubit-proximity placement and the use of both pattern matching and fast braiding path-finding. Moreover, these enhancements led to a considerable reduction in resource utilization by 40.5% and 35.8%. Thus, it is possible to reduce computational capacity and enable efficient SC communication even with smaller hardware.

4.4 Scalability Analysis

To evaluate the scalability of the proposed qubit-mapping method, we observed latency and runtime while increasing the circuit size. Fig. 9a indicates that HiLight consistently delivers the lowest latency for all benchmarks, underscoring its efficiency regardless of circuit size. Fig. 9b shows that our final version, "hilight-map," consistently showed the shortest runtime, whereas the "hilight-gm" occasionally had longer runtimes than "autobraid-full." Remarkably, for benchmarks up to 100 qubits, the runtime of our final version was virtually zero. This analysis confirms that our qubit-mapping method excels in providing lightweight and scalable mapping solutions, effectively reducing both latency and runtime.

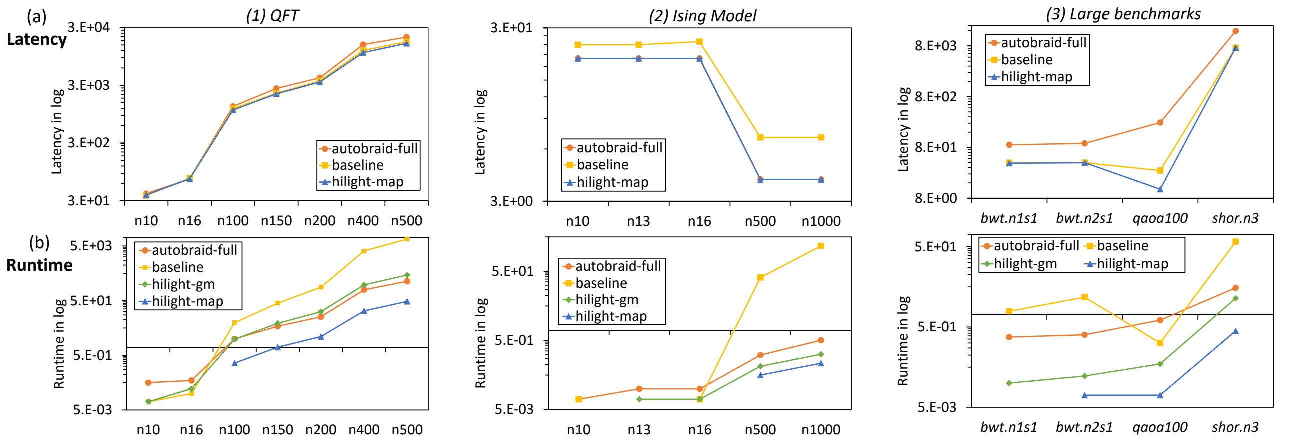


Figure 9: Summary of scalability analysis in terms of (a) latency and (b) runtime for different benchmarks. "baseline" is implemented using the existing graph-inspired heuristic [17] and braiding path-finding with all 16 possible cases. "hilight-gm" applies only to our proposed initial placement method. (Please note, because of the logarithmic scale of the graph, experimental results reported as "0 (s)" are not depicted.)

4.5 Program-Level Optimization Results

We evaluated the advantages of integrating program-level optimization with the proposed mapping (Fig. 10). Using only our mapping already improved efficiency over previous methods, yet had sub-optimal latency. By incorporating program-level optimization, we achieved a 0.7% reduction by simply adjusting the CX gate order. Furthermore, combining this with the QCO during gate-list generation resulted in a significant runtime reduction of 92.5% compared to AutoBraid. Resource utilization saw a slight increase of 0.3%, which correlates with the reduced latency according to Eq. 1. These results underscore the ability of our framework to efficiently tackle SC communication challenges, delivering enhanced performance.

4.6 Hardware-Level Optimization Results

We evaluated hardware resource utilization by adjusting the grid size (Fig. 10). In Table 1, experiments were conducted for a total of 20 benchmarks on a rectangular grid of size $M \times (M-1)$. Through hardware-level optimization, the resource utilization ratio was adjusted to 73.3%, with only a 0.5% increase in latency. When combined with program-level optimization, the latency increased to the same level as that in the proposed qubit-mapping method. At this point, a resource utilization of 74.4% was achieved. These experimental results demonstrate well-balanced hardware utilization using the proposed HiLight framework. Thus, we effectively addressed the SC communication problem while considering an actual FTQC hardware implementation.

5 CONCLUSION

We presented the **HiLight**, a powerful tool for optimizing SC communication in the FTQC era. Through a combination of advanced qubit-mapping techniques and program- and hardware-level optimizations, we provide lightweight and scalable solutions, featuring SWAP-less initial placements while effectively maximizing braiding parallelisms. Compared to previous methods, HiLight demonstrates high-performance outcomes with respect to runtime, latency, and resource utilization. Our experimental results underscore the remarkable potential of HiLight in advancing the FTQC era. Future endeavors will focus on exploring further optimization opportunities, such as those for single-qubit gates and the magic-state factory.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2019R1A5A1027055) and was supported by nanomaterials development program through the NRF (2022M3H4A1A04096496) funded by the Ministry of Science and ICT, Korea.

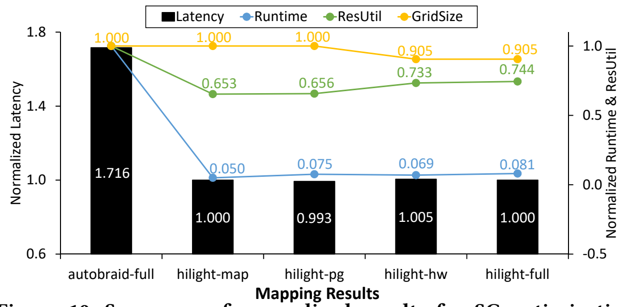


Figure 10: Summary of normalized results for SC optimization framework. "-pg" applies program-level optimization, "-hw" applies hardware-level optimization, and "-full" incorporates both program- and hardware-level optimizations with the proposed qubit mapping.

REFERENCES

[1] P. Ball, "First 100-QUBIT Quantum Computer Enters Crowded Race," *Nature*, 599 (2021), p. 542.
[2] IBM Newsroom, "IBM Unveils 400 Qubit-Plus Quantum Processor," 2022.
[3] PW. Shor, "Fault-Tolerant Quantum Computation," *Proc. IEEE FOCS*, 1996, pp. 56-65.
[4] J. Preskill, "Fault-Tolerant Quantum Computation," *Introduction to quantum computation and information*, 1998. pp. 213-269.
[5] J. Chiaverini et al., "Realization of Quantum Error Correction," *Nature*, 432(7017) (2004), pp. 602-605.
[6] SJ. Devitt et al., "Quantum Error Correction for Beginners," *Rep. Prog. Phys.*, 76(7) (2013), p. 076001.
[7] DA. Lidar et al., "Quantum Error Correction," Cambridge University Press, 2013.
[8] AG. Fowler et al., "Surface Codes: Towards Practical Large-Scale Quantum Computation," *Phys. Rev. A*, 86(3) (2012), p. 032324.
[9] AG. Fowler et al., "A Bridge to Lower Overhead Quantum Computation," *arXiv preprint arXiv:1209.0510*, 2012.
[10] NC. Jones et al., "Layered Architecture for Quantum Computing," *Phys. Rev. X*, 2(3) (2012), p. 031007.
[11] A. Paler et al., "Synthesis of Arbitrary Quantum Circuits to Topological Assembly," *Scientific reports*, 6(1) (2016), p. 30600.
[12] M. Hanks et al., "Effective Compression of Quantum Braided Circuits Aided by ZX-Calculus," *Phys. Rev. X*, 10(4) (2020), p. 041030.
[13] A. Javadi-Abhari et al., "Optimized Surface Code Communication in Superconducting Quantum Computers," *Proc. IEEE/ACM MICRO*, 2017, pp. 692-705.
[14] F. Hua et al., "Autobraid: A Framework for Enabling Efficient Surface Code Communication in Quantum Computing," *Proc. IEEE/ACM MICRO*, 2021, pp. 925-936.
[15] L. Lao et al., "Mapping of Lattice Surgery-Based Quantum Circuits on Surface Code Architectures," *Quantum Sci. Technol.*, 4(1) (2018), p. 015005.
[16] Y. Ding et al., "Magic-State Functional Units: Mapping and Scheduling Multi-Level Distillation Circuits for Fault-Tolerant Quantum Architectures," *Proc. IEEE/ACM MICRO*, 2018, pp. 828-840.
[17] S. Park et al., "A Fast and Scalable Qubit-Mapping Method for Noisy Intermediate-Scale Quantum Computer," *Proc. ACM/IEEE DAC*, 2022, pp. 13-18.
[18] S. Bravyi et al., "Universal Quantum Computation with Ideal Clifford Gates and Noisy Ancillas," *Phys. Rev. A*, 71(2) (2005), p. 022316.
[19] A. Cross, "The IBM Q Experience and QISKit Open-Source Quantum Computing Software," *APS March meeting abstracts*, 2018 (2018), pp. L58-003.
[20] A. Javadi-Abhari et al., "ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs," *Proc. ACM CF*, 2014, pp. 1-10.
[21] R. Wille et al., "RevLib: An Online Resource for Reversible Functions and Reversible Circuits," *Proc. IEEE ISMVL*, 2008, pp. 220-225.
[22] V. Omole et al., "Cirq: A Python Framework for Creating, Editing, and Invoking Quantum Circuits," *URL https://github. com/quantumlib/Cirq*, 2020.