

Optimization and Simulation of Quantum Circuits

Mingkuan Xu

CMU-CS-26-111

May 2026

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Zhihao Jia, Co-Chair

Umut A. Acar, Co-Chair

Ryan O'Donnell

Yongshan Ding (Yale University)

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Copyright © 2026 Mingkuan Xu

This research was sponsored by: the National Science Foundation under award numbers CCF1901381, CCF2107241, CCF2115104, CCF2119352, CNS2239351, and CNS2310949; and the Alfred P. Sloan Foundation under award number FG202524405.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Quantum computing, quantum compilers, quantum circuit optimization, quantum circuit simulation, formal verification, distributed GPU computing, superoptimization, cost-based backtracking search, integer linear programming, dynamic programming

For my parents

Abstract

Optimizing and simulating quantum circuits at scale are critical bottlenecks in realizing the full potential of quantum computing. This thesis delivers a comprehensive suite of systems to overcome these limitations, spanning both physical compilation and classical simulation.

For quantum circuit optimization, we first automate the discovery and verification of transformation rules by introducing Equivalence Circuit Class (ECC) sets and an efficient generation algorithm for arbitrary gate sets. We deploy these rules within the superoptimizer Quartz, which utilizes a cost-based backtracking search to escape local minima that trap traditional rule-based methods. To eliminate the exponential runtime penalties inherent to pure search, we introduce QALM, a hybrid optimization framework. By unifying search-based exploration with greedy rule-based exploitation in an interleaved control loop, QALM dynamically escapes local minima without exhaustive backtracking. Evaluated across a large benchmark suite of 248 diverse circuits, QALM outperforms existing search-based optimizers in final circuit quality while operating with the computational efficiency of rule-based systems.

While the prior two approaches aim for global optimization, this problem is intrinsically QMA-hard, creating a bottleneck for large programs. To circumvent this issue and scale up, we introduce OAC, a cut-and-meld circuit optimization algorithm. OAC cuts a circuit into subcircuits, applies an existing oracle optimizer independently, and seamlessly melds the results. This algorithm operates with a linear number of oracle calls while attaining local optimality. Empirical evaluation shows that OAC significantly improves the efficiency of state-of-the-art optimizers while enhancing overall quality.

Beyond physical execution, the scalable simulation of quantum circuits on classical hardware presents another major challenge. To address this, we present Atlas, a distributed GPU-based simulator that hierarchically partitions circuits to exploit data parallelism while minimizing communication overhead. By leveraging integer linear programming to allocate structurally related gates to nearby GPUs and dynamic programming for kernel scheduling, Atlas runs significantly faster than prior state-of-the-art GPU simulators.

Together, these frameworks provide a robust toolchain, improving both the execution of quantum circuits on physical devices and their scalable classical simulation.

Acknowledgments

When people asked why I chose Carnegie Mellon University for my Ph.D., I would give a simple answer: I wanted to learn, and CMU has the largest computer science department in which to do it. Over the last few years, I discovered that the true value wasn't in the sheer size of the department, but in the people who fill it.

First and foremost, I want to thank my advisors, Zhihao Jia and Umut Acar. Coming from different research backgrounds, they created a dynamic advising environment. Our joint meetings were always incredibly engaging; seeing their different perspectives collide often sparked the best ideas, and their meticulous feedback consistently sharpened the trajectory of my research.

Zhihao, thank you for introducing me to such compelling research problems right from the start. During my first year, you spent countless hours working with me hand-in-hand to navigate the rigorous standards of top-tier systems research. That intensive early mentorship led directly to our PLDI paper on Quartz, and getting that first major milestone under my belt so early made the rest of my Ph.D. significantly happier and more confident.

Umut, thank you for holding my work to a high standard of algorithmic rigor, and for providing much-needed perspective on how to navigate the academic world. When the pace of research and collaboration became intense, you taught me the invaluable lesson that it is okay to set professional boundaries and say “no.” Knowing that you always had my back helped me find the balance required to truly enjoy the rest of my time at CMU.

I would also like to sincerely thank the members of my thesis committee: Ryan O'Donnell and Yongshan Ding. I am deeply grateful for the time and energy they invested in evaluating this dissertation. Ryan's insightful feedback elevated the final quality of this work, and Yongshan's broader discussions regarding the quantum computing landscape have been an invaluable resource to me throughout my Ph.D.

My journey into research, however, began long before I arrived at CMU. I am deeply grateful to my undergraduate mentor, Yuanming Hu, who first showed me how to tackle large, open-ended systems problems rather than just solving known algorithms. Looking even further back, I owe a special thanks to my high school competitive programming coach, Weidong Hu. His dedicated coaching fundamentally shaped my problem-solving skills, and his early guidance on my first research topics laid the technical foundation that carried me all the way to this dissertation.

This dissertation was built on the foundation of collaborative science. I am deeply grateful to the core collaborators who drove these projects forward, particularly Pengyu Liu, Zikun Li, Shiyi Cao, Jatin Arora, Sam Westrick, and Aidan

Wagner. None of the research would have been possible without your contributions and help.

Beyond my core co-authors, I am incredibly fortunate to have learned from a wider network of researchers, professors, and friends within the quantum computing community. I want to express my sincere gratitude to Runzhou Tao, Yuxiang Peng, Zihan Chen, Jens Palsberg, Xiaodi Wu, Yunong Shi, Jianxin Chen, and Hanrui Wang. Whether through collaborations or informal discussions about the field, your willingness to share your expertise and bounce ideas around significantly broadened my technical perspective and made the research process far less solitary.

Throughout my time at CMU, I have been fortunate to interact with a wide network of faculty whose support took many forms. I am deeply thankful to Elaine Shi, Minchen Li, Mor Harchol-Balter, Phil Gibbons, Richard Peng, Danny Sleator, Tianqi Chen, and Greg Ganger, for offering everything from formal mentorship and rigorous research discussions to genuine friendship.

Beyond my interactions with faculty, my life in Pittsburgh was shaped by an amazing community of friends. Ke Wu, Yue Yao, Kaiyang Zhao, Ziyue Qiu, Yuang Chen, and my roommate Longxin Xie, thank you for every interesting dinner we have had together and the intriguing discussions we have had. Alongside these dinners, my daily academic routine was deeply enriched by my peers. I want to thank Mengdi Wu, Yonghao Zhuang, Gabriele Oliaro, Matias Scharager, Andrew Park, Long Pham, Ruihang Lai, Zhihao Zhang, Yixuan Mei, Yige Hong, Zhouzi Li, Yiran Lei, Jiayuan Liu, Yixin Dong, Siyuan Chen, Xinhao Cheng, Guying Lin, Jingxun Liang, Renfei Zhou, Ziqian Zhong, Junzhao Yang, William He, George Li, Juntian Zheng, Michael Liu, Luiz de Sa, Afonso Tinoco, Edward Chen, Hao Chung, Mingxun Zhou, Juncheng Yang, Yiwei Zhao, Hongyi Jin, Bohan Hou, Zhuoming Chen, Yingyi Huang, Shushi Hong, Jianan Ji, Yiyang Zhai, and Mike Rainey, for the constant camaraderie as officemates, the supportive environment of the Catalyst research group, and the engaging interactions at various departmental seminars.

I also want to extend my heartfelt thanks to the administrative staff in the Computer Science Department, particularly Matthew Stewart, Deb Cavlovich, Emily Spencer, and Catherine Copetas. From navigating reimbursements and travel logistics to scheduling my defense, their behind-the-scenes support kept my Ph.D. on track and saved me countless hours of stress. I also want to thank Phyllis Starz and Ellen Hughes from QCiT, and Karen Lindenfelser and Joan Digney from PDL, for organizing all the meetings and events.

Outside research, the bridge table was a much-needed mental escape. Thank you, Alec Sun, Christina Yao, Yuchen Xu, Yewen Fan, Jiaheng Hu, Viola Chen, Aaron Balleisen, Raphael Stone, P.J. Duers, James Sturgill, Daniel Weiss, Austin Shi, Gan Yang, Lena Liang, Noel Walkington, Tianyang Chen, Yilun Sheng, Jinhao Zhao, and Yanlai Yang, for the countless hands and tournaments we played together.

Taking another step away from the CMU campus, I also had the invaluable opportunity to gain external perspectives through industry internships. I want to sincerely thank my managers, mentors, colleagues, and fellow interns, particularly Sean McLaughlin, Georges-Axel Jaloyan, Jenny Xiang, Yongwei Yuan, Wujie Huang,

Wenjun Qiu, Fang Wu, Yichen Yang, Dunzhu Li, Lang Liu, Yunyi Shen, Cheuk Hei Chu, and Tristan Chaang. Thank you for the technical guidance, the collaborative environment, and for making my time in industry such a rewarding part of my Ph.D. journey.

Navigating a Ph.D. requires a life entirely disconnected from academia, and I am thankful to my broader circle of friends who shared my downtime through trips, gym sessions, and everyday life. I want to thank Xingjian Bai, Jingbang Chen, Junjie Chen, Junkun Chen, Lijie Chen, Hengrui Chen, Tong Chen, Weizhe Chen, Weiying Chen, Yuhe Chen, Shuqi Dai, Yan Dai, Xingyu Dang, Mingyang Deng, Shiyi Du, Yuhao Du, Zhengyang Geng, Zhongtian He, Chenyuan Hu, Zikuan Huang, Jinchun Jiang, Lin Jiang, Xiaoyu Jiang, Weikai Jin, Lingjing Kong, Chao Li, Dantong Li, Jiutu Li, Runming Li, Yuchen Li, Honghao Lin, Charlie Liu, Mingrui Liu, Runda Liu, Zining Liu, Rui Lu, Ji Luo, Lingxiao Luo, Zhaofeng Luo, Zhezhen Luo, Xin Lyu, Zijian Lyu, Xiao Mao, Nj Mukherjee, Siqi Ouyang, Crystal Qiao, Feiran Qin, Justin Raizes, Hugo Sadok, Yiheng Shen, Junxi Song, Mingjie Sun, Mingjun Sun, Nuozhou Sun, Haohua Tang, Bingkai Wang, Dongqing Wang, Hongjian Wang, Leping Wang, Sipeng Wang, Yuchao Wang, Zihang Wen, Jiayi Weng, Wentao Weng, Hongxun Wu, Jiacheng Wu, Jianyu Wu, Jinzhao Wu, Yifei Wu, Yi Wu, Zuofan Wu, Dejia Xu, Feiyang Xu, Haike Xu, Jeff Xu, Shengwei Xu, Yi Xu, Yixuan Xu, Donghui Yan, Haoning Yang, Jingqin Yang, Kunhe Yang, Peng Ye, Tian Ye, Longhui Yin, Chenkai Yu, Weiqiang Yuan, Xiaodi Yuan, Chi Zhang, Lujing Zhang, Qifan Zhang, Yifan Zhang, Yinghuan Zhang, Lux Zhao, Runlong Zhou, Zikai Zhou, Shuhao Zheng, Hanlin Zhu, Yuehua Zhu, Zhenting Zhu, and so many more. I appreciate every single one of you, and I hope that we will stay in touch.

Finally, and most importantly, I want to express my deepest gratitude to my parents. None of this would have been possible without your unwavering love and unconditional belief in me. Thank you for being my constant anchor, especially during the most challenging moments of this journey. Everything I have achieved is built upon your endless support and sacrifices, and I dedicate this dissertation to you.

Contents

1	Introduction	1
1.1	Quantum Circuit Optimization	1
1.2	Quantum Circuit Simulation	3
1.3	Contributions	4
2	Background	7
2.1	Fundamentals of Quantum Computing	7
2.1.1	Qubits and Superposition	7
2.1.2	Quantum Gates	8
2.1.3	Multi-Qubit Systems and Entanglement	9
2.2	The Quantum Circuit Model	11
2.2.1	Circuit Representation	12
2.2.2	Symbolic Circuits	13
2.2.3	Circuit Equivalence and Transformations	13
3	Equivalence Generation	15
3.1	System Overview	15
3.2	Equivalent Circuit Classes	16
3.2.1	Completeness for ECC sets	16
3.3	Circuit Generator	17
3.3.1	The REPGEN Algorithm	17
3.3.2	Correctness of REPGEN	19
3.3.3	Complexity of REPGEN	21
3.4	Equivalence Verifier	21
3.4.1	Phase Factors	22
3.4.2	Trigonometric Functions	22
3.5	Pruning Redundant Transformations	23
3.5.1	ECC Simplification	23
3.5.2	Common Subcircuit Pruning	24
3.6	Summary	24
4	Quartz: Search-based Superoptimization	25
4.1	Circuit Optimizer	25
4.2	Preprocessing	27

4.3	Implementation	28
4.3.1	Floating-Point Arithmetic	28
4.3.2	Supported Gate Sets	29
4.3.3	Symbolic Parameter Expressions	29
4.3.4	Alternative Verification Backends	29
4.4	Evaluation	30
4.4.1	Experiment Setup	30
4.4.2	Circuit Optimization Results	30
4.4.3	Analyzing Quartz’s Generator and Verifier	32
4.4.4	Scalability	37
5	QALM: Interleaved Exploration and Exploitation	51
5.1	Background	52
5.1.1	Rule-Based Optimization	52
5.1.2	Search-Based Optimization	54
5.2	Method	54
5.2.1	Optimization Spectrum	54
5.2.2	The QALM Algorithm	55
5.3	Implementation	57
5.3.1	ROQC: Rule-based Optimizer for Quantum Circuits	57
5.3.2	Search-Based Optimization	58
5.3.3	Greedy Mode	58
5.4	Evaluation	60
5.4.1	Experiment Setup	60
5.4.2	Main Results	61
5.4.3	Optimization Efficiency	62
5.4.4	Quartz’s Benchmark	62
5.4.5	Greedy Mode	64
5.4.6	Exploration Depth	64
5.4.7	Exploration Strategy	66
5.5	Conclusion	66
6	OAC: Local Optimization	69
6.1	Introduction	69
6.2	Local Optimality	70
6.2.1	Circuit Syntax and Semantics	70
6.2.2	Local Optimality	72
6.2.3	Circuit Rewriting for Local Optimality	72
6.2.4	Correspondence between local optimality and local rewrites	73
6.2.5	Termination	74
6.3	Local Optimization Algorithm	76
6.3.1	The algorithm	76
6.3.2	Meld Example	78
6.3.3	Correctness and Efficiency	79

6.3.4	OAC*: controlling convergence	83
6.4	Implementation	83
6.4.1	Circuit Representation and Compaction	84
6.5	Evaluation	84
6.5.1	Benchmarks and gate set	85
6.5.2	RQ I: Efficiency of OAC	85
6.5.3	RQ II: Optimization quality and importance of MELD	88
6.5.4	RQ III: Empirical versus asymptotic performance	90
6.5.5	RQ IV: Impact of compaction and Ω	90
6.6	Experiments with Clifford+T Gate Set	92
6.6.1	RQ I: Effectiveness of OAC	92
6.6.2	RQ IV: Impact of varying segment size Ω	93
6.7	Conclusion	96
7	Atlas: Optimization for Simulation on Classical Hardware	99
7.1	Architectural Model	100
7.2	Hierarchical Partitioning	101
7.3	Circuit Staging	102
7.4	Circuit Kernelization	106
7.5	Implementation	109
7.5.1	Reducing Size of DP State in KERNELIZE	109
7.5.2	Cost Function in KERNELIZE	112
7.5.3	Optimizations to the KERNELIZE Algorithm	113
7.5.4	An Alternate Algorithm for Kernelization	115
7.5.5	Atlas	117
7.6	Evaluation	117
7.6.1	Experimental Setup	117
7.6.2	End-to-end Performance	118
7.6.3	DRAM Offloading	121
7.6.4	Circuit Staging	121
7.6.5	Circuit Kernelization	122
8	Related Work	133
8.1	Quantum Circuit Optimization	133
8.1.1	Compiler Infrastructures and Rule-Based Optimization	133
8.1.2	Automated Discovery and Superoptimization	134
8.1.3	Search-Based and Hybrid Strategies	134
8.1.4	Compiler Verification	134
8.1.5	Resynthesis Methods	135
8.1.6	Scalability and Local Optimality	135
8.1.7	Distinction from Quantum Circuit Cutting and Knitting	136
8.1.8	Optimization Objectives and Hardware Constraints	136
8.1.9	Dynamic Circuit Compilation	137
8.2	Quantum Circuit Simulation	137

8.2.1	State-Vector Simulation and Partitioning	137
8.2.2	Kernelization and Gate Fusion	137
8.2.3	Alternative Simulation Paradigms	138
9	Conclusion and Future Directions	139
9.1	Summary of Contributions	139
9.2	Future Directions	140
9.2.1	Bounded Approximate Optimization via Cut-and-Meld	140
9.2.2	Adaptive Simulator Selection	140
	Bibliography	143

List of Figures

2.1	Visual representation of an interference example.	8
2.2	Quantum circuits and transformations. The green highlighted box in (a) is a subcircuit, while the red dashed area is not; the $U_1(-\pi)$ and X gates also form a subcircuit.	11
2.3	Graph representation for Figure 2.2a's circuit. The green box (subcircuit, also convex subgraph) and red dashed area (not a subcircuit, also non-convex subgraph) match those of Figure 2.2a.	12
3.1	Quartz overview.	16
3.2	Illustrating optimization via transformations.	17
3.3	Illustrating a redundant circuit transformation.	23
4.1	A transformation sequence applied by Quartz that reduces the total gate count in the <code>gf2^4_mult</code> circuit by swapping the control and target qubits of three $CNOT$ gates. Note that the first three transformations do not reduce gate count.	26
4.2	Optimization effectiveness with (n, q) -complete ECC sets for varying n and q after a 24-hour search timeout. For $n = 0$ there are no transformations and the results match the "Quartz Preprocess" column of Table 4.2.	37
4.3	Optimization effectiveness over time ($q=3; 2 \leq n \leq 7$). For each time point, "best" computes the reduction in geometric mean gate count obtained by selecting the most effective value for n at that time point for each circuit (i.e., different circuits may use different n 's, and the same circuit may use different n 's at different time points).	39
4.4	<code>adder_8</code> (900 gates).	40
4.5	<code>barenco_tof_3</code> (58 gates).	40
4.6	<code>barenco_tof_4</code> (114 gates).	40
4.7	<code>barenco_tof_5</code> (170 gates).	41
4.8	<code>barenco_tof_10</code> (450 gates).	41
4.9	<code>cs1a_mux_3</code> (170 gates).	41
4.10	<code>csum_mux_9</code> (420 gates).	42
4.11	<code>gf2^4_mult</code> (225 gates).	42
4.12	<code>gf2^5_mult</code> (347 gates).	42
4.13	<code>gf2^6_mult</code> (495 gates).	43
4.14	<code>gf2^7_mult</code> (669 gates).	43
4.15	<code>gf2^8_mult</code> (883 gates).	43

4.16	gf2^9_mult (1095 gates).	44
4.17	gf2^10_mult (1347 gates).	44
4.18	mod5_4 (63 gates).	44
4.19	mod_mult_55 (119 gates).	45
4.20	mod_red_21 (278 gates).	45
4.21	qcla_adder_10 (521 gates).	45
4.22	qcla_com_7 (443 gates).	46
4.23	qcla_mod_7 (884 gates).	46
4.24	rc_adder_6 (200 gates).	46
4.25	tof_3 (45 gates).	47
4.26	tof_4 (75 gates).	47
4.27	tof_5 (105 gates).	47
4.28	tof_10 (255 gates).	48
4.29	vbe_adder_3 (150 gates).	48
4.30	7 runs of mod5_4 with a $(n, 3)$ -complete ECC set for each $3 \leq n \leq 7$. Each marker denotes one run. For example, the three markers on the 57.1%-reduction line of $n = 5$ show three runs resulting in 27 gates after 24 hours.	49
5.1	Comparison of optimization trajectories across three strategies: rule-based (pure exploitation), search-based (pure exploration), and QALM (interleaved exploration and exploitation). Rule-based optimization greedily descends to a local minimum in a single step and terminates (shown as <i>A</i>). Search-based optimization reaches the same local minimum more slowly, then continues to explore and discovers a candidate region; however, because this new region has a higher cost (shown as <i>B</i>), and the search-based method cannot determine whether this region is truly promising, escaping the current basin requires exponentially many search steps. QALM interleaves the two phases: the exploit phase quickly reaches the local minimum, and the subsequent explore phase discovers the promising region just as the search-based method does. Crucially, the next exploit phase immediately capitalizes on this discovery, descending to a new solution (shown as <i>C</i>) that surpasses the original local minimum. Through this interleaving, QALM finds much better solutions than both methods with speed comparable to rule-based optimization.	52
5.2	Percentage of circuits on which QALM matches or exceeds each baseline across fidelity, <i>CX</i> gate reduction, and total gate reduction. GUOQ and QUESO are each shown under two configurations: one optimizing for <i>CX</i> count and one for total gate count.	53
5.3	Hadamard self-inverse property: $HH = I$, where <i>I</i> denotes the identity gate, representing no operation on the qubit.	53
5.4	Example of an optimization rule for a gate pattern with multiple <i>CNOT</i> and R_z gates.	53
5.5	Search-based optimization example. The first transformation increases gate count ($3 \rightarrow 4$), but enables subsequent <i>CX</i> cancellation ($4 \rightarrow 2$).	54

5.6	Gate count distribution in our benchmark. The dashed line indicates where the total gate count is twice the two-qubit gate count, or equivalently, where the number of single-qubit gates equals the number of two-qubit gates. The left panel shows the distribution before optimization, and the right panel shows the distribution after optimization using QALM and GUOQ with two-qubit gate optimization mode, which is the strongest baseline in our evaluation.	60
5.7	Per-circuit comparison of QALM against GUOQ, QUESO, Quartz, and VOQC. Each column corresponds to a baseline optimizer, and each row shows a different metric: output circuit fidelity (top), CX gate reduction percentage (middle), and total gate reduction percentage (bottom). Circuits are sorted by the QALM metric value.	61
5.8	Effect of the greedy mode. Each curve shows QALM run with greedy mode (accepting only cost-reducing transformations) applied up to depth $k \leq \text{greedy_k}$, with unrestricted search beyond.	64
5.9	Effect of fixing the exploration depth k in QALM. Each curve runs QALM with a single fixed $k \in \{1, \dots, 5\}$ (no depth advancement), averaged over 26 benchmark circuits.	65
5.10	Effect of pool size N_{pool} and branch factor N_{branch} on average gate count reduction, over 26 benchmark circuits with a one-hour timeout. N_{pool} has little effect on final quality, while N_{branch} exhibits a sweet spot at $N_{\text{branch}} = 3$	67
6.1	Syntax of LAQE and well-formed circuits.	71
6.2	Definition of local optimality, parameterized by a COST function, an ORACLE optimizer, and a segment length Ω	71
6.3	Local optimization rewrite rules.	73
6.4	The figure illustrates how our rewriting semantics optimizes circuits for $\Omega = 2$. The figure implicitly assumes an oracle that removes any two consecutive H and X gates. At each step, our semantics either selects a segment of size 2 (denoted by green boxes) and performs an optimization, or picks a gate and shifts it left.	74
6.5	Algorithm OAC produces locally optimal circuits with respect to a given ORACLE, COST, and segment length Ω . To achieve local optimality, OAC only uses the oracle on small segments of length 2Ω . The algorithm repeatedly optimizes and compacts the circuit until convergence. The function <code>SEGOPT()</code> implements our optimization algorithm and uses <code>MELD()</code> to efficiently produce segment optimal circuits.	77

6.6	The figure shows a three-step MELD operation and illustrates how it propagates optimizations at the boundary of two optimal circuits. At the top, the figure shows specific optimizations “1” and “2”, and towards the bottom, the figure shows the optimization steps of MELD. Before the first step, two individual circuit segments that are optimal are separated by a dashed line. Each MELD step considers a segment, represented by a box with solid/dotted/shaded lines, and applies an optimization to it, reducing the gate count. The first step focuses on the boundary segment within the solid green box, overlapping with both circuits, and applies “Optimization 1”. This step introduces a flipped <i>CNOT</i> gate, which interacts with a neighboring <i>CNOT</i> gate and triggers “Optimization 2” in the purple dotted box. The third step merges two neighboring rotation gates in the yellow shaded box.	79
6.7	OAC* terminates when the cost improvement ratio falls below the <i>convergence threshold</i> , ϵ	83
6.8	The number of oracle calls versus input circuit size for all our circuits (Nam gate set). The plots show that the number of calls scales linearly with the number of gates.	90
6.9	The figure plots the impact of the parameter Ω on the output T count (lower is better) and running time of OAC optimizer on an <code>hhl</code> circuit with 7 qubits. The dotted red lines in the plots denote the output T count and the running time of the oracle optimizer FeynOpt on the whole circuit. For almost all values of Ω , the output quality matches the oracle optimizer, demonstrating that local optimality is a robust quality criterion for T count optimization.	95
6.10	Zooming in: Time vs. Omega plot	97
7.1	An example application of circuit partitioning and execution. We stage the circuit so that qubits of each gate map to local qubits (i.e., green lines in each stage). The notation $q_i[p_j]$ indicates that the i -th logical qubit maps to the j -th physical qubit. The KERNELIZE algorithm then partitions the gates of each stage into kernels that provide for data parallelism.	103
7.2	Sharding with different types of communication. The simulation has 1 local, 1 regional, and 1 global qubit. $p_i - q_j$ indicates that the i -th physical qubit is mapped to the j -th logical qubit. Inter-node communication is triggered if we update any global qubits (Figure 7.2a), and only intra-node communication is triggered otherwise (Figure 7.2b).	104
7.3	Kernel examples. The two green dashed kernels satisfy Constraint 1. The two blue dotted kernels do not satisfy Constraint 1 and thus are not considered by the KERNELIZE algorithm.	107
7.4	An example dynamic programming state in the implementation. The circuit sequence is $\mathcal{C} = [g_0, g_1, \dots, g_9]$	110
7.5	If we swap an X gate with an adjacent <i>CNOT</i> gate, we need to flip the control state of the new <i>CNOT</i> gate to $ 0\rangle$	114

7.6	Weak scaling of Atlas, HyQuas, cuQuantum, and Qiskit with 28 local qubits as the number of global qubits increases from 0 (on 1 GPU) to 8 (on 256 GPUs). Qiskit scales poorly and usually does not fit into our charts.	119
7.7	Simulation time breakdown: The average communication time and its percentage in the average total simulation time of Atlas.	120
7.8	Atlas outperforms QDAO by $61\times$ on average. Log-scale simulation time (single GPU) with DRAM offloading for <code>qft</code> circuits.	120
7.9	DRAM offloading scales. Atlas simulation time of a 32-qubit <code>qft</code> circuit with 1, 2, and 4 GPUs.	120
7.10	Number of stages, Atlas versus SnuQS: The geometric mean over all our benchmark circuits with 31 qubits.	121
7.11	Number of stages, Atlas versus SnuQS: The geometric mean over all our benchmark circuits with 42 qubits.	121
7.12	Kernelization effectiveness: The relative geometric mean cost of KERNELIZE compared to greedy packing up to 5 qubits.	122
7.13	The relative geometric mean cost among all 99 circuits in Table 7.2 of KERNELIZE compared to greedy packing up to 5 qubits, and the running time of the kernelization algorithm when the pruning threshold T varies. Each data point is annotated with the pruning threshold T . “Atlas” denotes KERNELIZE. “Atlas-Naive” denotes ORDEREDKERNELIZE.	123
7.14	The performance of different kernelization algorithms on the circuit <code>ae</code>	124
7.15	The performance of different kernelization algorithms on the circuit <code>dj</code>	124
7.16	The performance of different kernelization algorithms on the circuit <code>ghz</code>	125
7.17	The performance of different kernelization algorithms on the circuit <code>graphstate</code>	125
7.18	The performance of different kernelization algorithms on the circuit <code>ising</code>	125
7.19	The performance of different kernelization algorithms on the circuit <code>qft</code>	126
7.20	The performance of different kernelization algorithms on the circuit <code>qpeexact</code>	126
7.21	The performance of different kernelization algorithms on the circuit <code>qsvm</code>	126
7.22	The performance of different kernelization algorithms on the circuit <code>su2random</code>	127
7.23	The performance of different kernelization algorithms on the circuit <code>vqc</code>	127
7.24	The performance of different kernelization algorithms on the circuit <code>wstate</code>	127
7.25	The performance of different kernelization algorithms on the circuit <code>hhl</code>	128
7.26	The run time of different kernelization algorithms on the circuit <code>ae</code>	128
7.27	The run time of different kernelization algorithms on the circuit <code>dj</code>	128
7.28	The run time of different kernelization algorithms on the circuit <code>ghz</code>	129
7.29	The run time of different kernelization algorithms on the circuit <code>graphstate</code>	129
7.30	The run time of different kernelization algorithms on the circuit <code>ising</code>	129
7.31	The run time of different kernelization algorithms on the circuit <code>qft</code>	130
7.32	The run time of different kernelization algorithms on the circuit <code>qpeexact</code>	130
7.33	The run time of different kernelization algorithms on the circuit <code>qsvm</code>	130
7.34	The run time of different kernelization algorithms on the circuit <code>su2random</code>	131
7.35	The run time of different kernelization algorithms on the circuit <code>vqc</code>	131
7.36	The run time of different kernelization algorithms on the circuit <code>wstate</code>	131
7.37	The run time of different kernelization algorithms on the circuit <code>hhl</code>	132

8.1 Conceptual distinction between the software partitioning used in OAC and the physical circuit cutting found in prior execution strategies. OAC uses software boundaries to optimize a single, fully entangled program before execution (top). In contrast, traditional circuit knitting must physically sever entangling gates, executing multiple smaller, independent subcircuits that require exponential classical overhead to reconstruct the final results (bottom). 136

List of Tables

4.1	Gate sets used in our evaluation. At the time of this research, IBM QX5 utilized the $\{U_1, U_2, U_3, CNOT\}$ gate set, and we refer to this gate set throughout this thesis. IBM has subsequently updated the native gate set supported by its newer quantum processors.	28
4.2	Gate count results for the Nam gate set. The best result for each circuit is in bold. “Quartz Preprocess” lists gate count after Quartz’s preprocessor (Section 4.2).	31
4.3	Gate count results for the IBM gate set. The best result for each circuit is in bold. “Quartz Preprocess” lists gate count after Quartz’s preprocessor (Section 4.2).	33
4.4	Gate count results for the Rigetti gate set. The best result for each circuit is in bold. “Quartz Preprocess” lists gate count after Quartz’s preprocessor (Section 4.2).	34
4.5	Metrics for Quartz’s generator, when generating (n, q) -complete ECC sets for $q=3$ and varying values of n for the three gate sets. $ \mathcal{T} $ is the resulting number of transformations, $ \mathcal{R}_n $ is the size of the resulting representative set, and ch is the characteristic (see Algorithm 1 and Theorem 2).	35
4.6	Number of circuits considered when using REPGEN with or without the pruning techniques of Section 3.5 to generate (n, q) -complete ECC sets for $q=3$ and varying values of n for the three gate sets. Circuits are counted by their sequence representation, as REPGEN considers multiple sequences for each actual circuit (Section 3.3). Parentheses show reduction relative to the number of all possible circuits for n and q . “REPGEN” corresponds to REPGEN without additional pruning. “+ ECC Simplification” corresponds to REPGEN combined with ECC simplification. “+ Common Subcircuit” corresponds to REPGEN combined with all pruning techniques and ultimately represents Quartz’s generator.	36
4.7	Metrics for Quartz’s generator, when generating (n, q) -complete ECC sets for $2 \leq n \leq 7$ and $1 \leq q \leq 4$ for the Nam gate set. $ \mathcal{T} $ is the resulting number of transformations. The characteristics (see definition in Section 3.3.3) for $q = 1, 2, 3, 4$ are 7, 16, 27, 40, respectively.	36
4.8	Gate count results for the Nam gate set when using (n, q) -complete ECC sets with varying values of n and q , and a search timeout of 24 hours. “Pr.” shows the result when $n = 0$ or $n = 1$: the ECC set is empty, so the results match the “Quartz Preprocess” column of Table 4.2. When $q = 1$, the results are the same when $2 \leq n \leq 7$	38

5.1	Per-metric averages across the benchmark suite. Each optimizer is given one hour per circuit, except for “QALM (1 minute)”, which is given only one minute per circuit.	62
5.2	Comparing QALM with existing optimizers on total gate count reduction under a 1-hour time limit, on 26 benchmark circuits used by Quartz. The best result for each circuit is in bold.	63
6.1	The table shows the running time in seconds of the four optimizers, using gate count as the cost metric. The column “OAC Speedup” is the speed of our OAC with respect to VOQC, calculated as VOQC time divided by OAC time.	86
6.2	Optimization Quality. The table shows the percentage reduction in gate count achieved by the four optimizers. We write “N.A.” in cases where the optimizer did not finish within the allotted 12-hour deadline. The experiments show that our cut-and-meld optimizer OAC optimizes slightly better on average than all other optimizers.	87
6.3	Ablation study of MELD. The table shows percentage reduction in gate count achieved by the version of OAC, called OACMinus, that uses simply concatenation instead of the MELD algorithm, compared with VOQC and the original OAC. The measurements show that with the MELD algorithm, our OAC optimizer reduces gate counts better than using VOQC alone or simply splitting the circuit into segments and optimizing them using VOQC separately.	89
6.4	The number of rounds and the percentage of optimizations in each optimization round of OAC.	91
6.5	Choice of Ω : performance of OAC with different Ω on the <code>hh1</code> circuit with 9 qubits, which initially contains 63392 gates. For reference, we also present the performance of VOQC at the end.	92
6.6	The figure shows the optimization results of optimizers OAC and FeynOpt, with T count as the cost function. It shows the running time in seconds for both optimizers (lower is better) and calculates the speedup of our OAC by taking the ratio of the two timings. The figure also shows the T count reductions of both tools. The results show that our OAC delivers excellent time performance and runs almost an order of magnitude ($9.9\times$) faster than FeynOpt on average. Our OAC optimizer achieves this speedup without any sacrifice in circuit quality, producing circuits of the same quality as FeynOpt.	94
6.7	Results for OAC with FeynOpt on the <code>hh1</code> circuit with 7 qubits.	96
7.1	Our benchmark circuits and their size (number of gates).	113
7.2	Our benchmark circuits and their size (number of gates).	117
7.3	The number of gates in the <code>hh1</code> circuit.	123

Chapter 1

Introduction

Quantum computing has established an advantage over classical computing, especially in areas such as cryptography, machine learning, and physical sciences [14, 17, 23, 24, 26, 51, 83, 99, 103, 123, 124, 128, 160]. Key to realizing the advantage of quantum computing in these and similar areas is achieving the scale of thousands of qubits and millions of quantum operations (a.k.a., gates), often with high fidelity (low error rate) [7, 54, 62, 102]. Over the past decade, the potential of quantum computing and the challenges of scaling it have motivated much work on both hardware and software.

On the hardware front, there are over a dozen proposals for realizing quantum computing in practice, and nearly all these proposals support different kinds of quantum operations, i.e., instruction set architectures (ISAs). For example, quantum computers based on superconducting circuits [66, 77], trapped ions [94, 95], and Rydberg atom arrays [28, 49, 100, 122] have advanced rapidly, scaling to thousands of qubits and achieving entanglement fidelity over 99.9%. The increasing diversity in quantum processors makes it challenging to design optimizing compilers for quantum programs, since the compilers must consider a variety of ISAs and carry optimizations specific to different ISAs.

On the software front, a plethora of programming languages, optimizing compilers, and runtime environments have been proposed, both in industry and in academia (e.g., [27, 56, 61, 97, 105, 113, 125, 130, 139, 149, 155, 156]). However, due to the limitations of modern quantum hardware and the need to scale the hardware to a larger number of gates, optimizing quantum programs or circuits remains a critical bottleneck to realizing the potential of quantum computing. Furthermore, quantum computers are expensive resources—many remain inaccessible beyond a small group. Therefore, there is also significant interest in quantum circuit simulation, which enables performing robust quantum computation on classical parallel machines.

In the remaining part of this chapter, we introduce our works in optimization and simulation of quantum circuits.

1.1 Quantum Circuit Optimization

Designing optimizing compilers for quantum programs is challenging. Unlike conventional compilers that target fixed hardware instruction set architectures (ISAs), such as x86 for CPUs, de-

playing quantum circuits on modern quantum devices requires quantum compilers to target a diversity of ISAs. Additionally, achieving optimized performance on a quantum device requires optimizations specific to the ISA supported by the quantum device.

To reduce the execution cost of a quantum circuit, the most common form of optimization is *circuit transformations* that substitute a subcircuit matching a specific pattern with a functionally equivalent new subcircuit with improved performance (e.g., using fewer quantum gates). Many existing quantum compilers rely on circuit transformations manually designed by experts and applied greedily. For example, Qiskit [5] and t|ket> [130] use greedy rule-based strategies to optimize a quantum circuit and perform circuit transformations whenever applicable. VOQC [61] formally verifies circuit transformations but still requires users to manually specify them. Although rule-based transformations can reduce the cost of a quantum circuit, they have two key limitations.

First, because existing optimizers rely on domain experts to design transformations, they require significant human effort and may also miss subtle optimizations that are hard to discover manually, resulting in sub-optimal performance.

Second, circuit transformations designed for one quantum device do not directly apply to other devices with different ISAs, which is problematic in the emerging diverse quantum computing landscape. For example, IBM QX5 [47] supports the U_1 , U_2 , U_3 and $CNOT$ gates, while Rigetti Agave [117] supports the $R_x(\pm\frac{\pi}{2})$, $R_x(\pi)$, $R_z(\lambda)$, and CZ gates. As a result, circuit transformations tailored for IBM QX5 cannot optimize circuits on Rigetti Agave, and vice versa.

Quanto [109] proposed to automatically discover transformations by computing concrete matrix representations of circuits. Its main restriction is that it does not discover symbolic transformations, which are needed to deal with common *parametric* quantum gates in a general way.

Our work, Quartz [149], overcomes this by acting as a quantum circuit superoptimizer. It automatically generates and formally verifies symbolic circuit transformations for arbitrary gate sets, including parametric gates. Quartz provides two key advantages over existing quantum circuit optimizers. First, for a given set of gates, Quartz generates symbolic circuit transformations and formally verifies their correctness in a fully *automated* way, without any manual effort to design or implement transformations. Second, Quartz explores a more comprehensive set of circuit transformations by discovering *all* possible transformations up to a certain size, outperforming existing optimizers with manually designed transformations.

While Quartz successfully automates the discovery of a comprehensive set of circuit transformations, applying these optimizations to large circuits introduces a new hurdle. This problem is provably hard: determining whether two circuits are equivalent is QMA-complete [67], a quantum generalization of NP-completeness. In practice, this means that no polynomial-time algorithm can globally optimize arbitrary circuits. This situation mirrors classical compiler optimization, where problems like optimal register allocation and instruction scheduling are also NP-hard [4], forcing compilers to rely on heuristics.

Consequently, algorithms that cast optimization as an explicit search over this enormous space scale poorly. Quartz [149], or even its faster successors QUESO [144], Quarl [82], and GUOQ [146], can incur substantial runtime for large circuits and may not provide formal guarantees regarding global optimality.

To bridge the gap between optimization quality and scalability, we introduce two distinct but complementary solutions: QALM and OAC [13].

First, to address the inherent runtime penalties of pure search, we introduce QALM, a hybrid optimization framework. QALM interleaves rule-based exploitation with search-based exploration to navigate the optimization landscape. By interleaving fast local passes with targeted exploration, it escapes local minima without the exponential overhead of exhaustive search. As a result, QALM outperforms existing rule-based and search-based optimizers.

Second, to further scale these optimizations to larger circuits, we introduce OAC. Given any optimization algorithm as an oracle, OAC addresses a fundamental question: *is it possible to offer a formal quality guarantee while also ensuring efficiency?* The answer lies in shifting from global to *local optimality*. Rather than attempting an intractable global optimization, OAC utilizes a novel rewriting semantics and a cut-and-meld algorithm to optimize quantum circuits in iterative rounds.

By hierarchically cutting circuits into smaller subcircuits, optimizing them independently using the chosen oracle, and efficiently melding them back together across the cuts, OAC achieves segment optimality. This algorithm significantly improves optimization efficiency while closely matching or exceeding the optimization quality of state-of-the-art tools.

Once these large-scale quantum programs are highly optimized, the next critical challenge is actually executing them. In the current Noisy Intermediate-Scale Quantum (NISQ) era [110], physical hardware remains a scarce resource with strict limitations on qubit count and fidelity. Consequently, we must rely heavily on classical simulation, not only to validate program behavior before deployment, but also to run circuits that currently exceed physical hardware capabilities. However, this reliance introduces its own severe computational bottlenecks.

1.2 Quantum Circuit Simulation

While classical simulation provides a necessary and robust execution environment for quantum computation, performing it at scale is a formidable computational challenge. Several approaches to simulating quantum circuits have been proposed, including Schrödinger- and Feynman-style simulation. Even though Feynman-style simulation can require a small amount of space, its time complexity is exponentially higher for deep circuits. Therefore, most modern simulators employ Schrödinger-style quantum circuit simulation, which maintains an entire quantum state in a *state vector* of size 2^n for n qubits, and applies each gate of the circuit to it.

To tackle the scalability challenges of state-vector simulation, researchers exploit the plethora of parallelism available in this task by using CPUs, GPUs, and parallel machines [5, 8, 15, 16, 58, 72, 104, 132, 133, 157, 159]. To cope with the increasing demand for simulating larger circuits, recent work has proposed techniques for storing state vectors on DRAM and secondary storage (e.g., disks) [65, 132]. Even after over a decade of research, quantum circuit simulation remains challenging for performance and scalability.

At a high level of abstraction, there are three key challenges to (Schrödinger style) state-vector simulation. The first challenge is the space requirements of storing the state vector with 2^n amplitudes, each of which is a complex number. Overcoming this challenge requires distributing the state vector in a parallel machine with heterogeneous memory. Second, simulating the application of a quantum gate to one or multiple qubits involves strided accesses to the state vector, which requires *fine-grained* inter-node and/or inter-device communications. The cost of

these communications becomes the performance bottleneck for large-scale circuit simulation. Third, because quantum gates typically operate on a small number of qubits, simulating a quantum gate involves multiplying a sparse matrix with the state vector, resulting in low arithmetic intensity, e.g., as little as 0.5 on real-world quantum circuits [65]. This harms performance on modern accelerators (e.g., GPUs) designed for compute-intensive workloads.

Our work, Atlas [151], proposes techniques for performant and scalable quantum circuit simulation on modern GPU systems. Specifically, consider a multi-node GPU architecture, where each node may host a number of GPUs. On this architecture, we perform quantum circuit simulation in the (Schrödinger) state-vector style by using the available memory across all nodes to store the state. To minimize the communication costs, we partition the quantum circuit into a number of *stages*, each of which consists of a (contiguous) subcircuit of the input circuit that can be simulated on a single GPU without requiring non-local accesses outside the memory of the GPU. To maximize the benefits from parallelism, we further partition the subcircuit of each stage into *kernels* or groups of gates that are large enough to benefit from parallelism but also small enough to prevent exponential blowups in cost.

1.3 Contributions

This thesis makes the following primary contributions:

Generation and Application of Circuit Transformations (Quartz, QALM) : In Chapters 3 and 4, we introduce the theoretical foundation for Equivalence Circuit Class (ECC) sets and present Quartz, a superoptimizer that automates the generation and verification of circuit transformations for arbitrary gate sets. We then present QALM in Chapter 5, an optimizer that combines search-based exploration with rule-based exploitation. This allows the optimizer to escape local minima without incurring exponential runtime overhead.

Scalable Optimization via Local Optimality (OAC): In Chapter 6, we introduce OAC, a cut-and-meld circuit optimization algorithm that guarantees segment optimality. OAC operates with a linear number of oracle calls, improving the efficiency of existing optimizers by more than an order of magnitude.

High-Performance Classical Simulation (Atlas): In Chapter 7, we present Atlas, a distributed GPU-based simulator that hierarchically partitions circuits to exploit data parallelism. By minimizing communication costs through integer linear programming and optimizing kernel scheduling, Atlas runs significantly faster than prior state-of-the-art GPU-based simulators.

Publications The thesis contains work that appeared in the following publications.

1. Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, Zhihao Jia. Quartz: Superoptimization of Quantum Circuits. Published at PLDI 2022.
2. Aidan Wagner*, Mingkuan Xu*, Pengyu Liu*, Zhihao Jia, Umut A. Acar. QALM: Escaping Local Minima via Interleaved Exploration and Exploitation in Quantum Circuit Optimization. Under submission.

3. Jatin Arora, Mingkuan Xu, Sam Westrick, Pengyu Liu, Dantong Li, Yongshan Ding, Umut A. Acar. Local Optimization of Quantum Circuits. Published at QCE 2025.
4. Mingkuan Xu, Shiyi Cao, Xupeng Miao, Umut A. Acar, Zhihao Jia. Atlas: Hierarchical Partitioning for Quantum Circuit Simulation on GPUs. Published at SC 2024.

*Denotes equal contribution.

Chapter 2

Background

Quantum computing lies at the intersection of quantum mechanics and computer science. To understand the challenges involved in optimizing and simulating quantum circuits, we must first establish a common vocabulary that bridges these disciplines. This chapter introduces the mathematical formalism of qubits and gates, abstracting away the underlying device physics to focus on the circuit model that serves as the foundation for the algorithms presented in this thesis.

2.1 Fundamentals of Quantum Computing

2.1.1 Qubits and Superposition

Classical computers operate on *bits*, each storing either 0 or 1. In quantum computing, the basic unit is a *qubit*¹, which can exist in a *superposition* of both 0 and 1 states simultaneously [99]. This is analogous to how a probability distribution can assign weights to multiple outcomes. However, modeling quantum states purely as probability distributions fails to capture the full behavior of the system.

To see why, consider a thought experiment involving two qubits prepared in distinct states. If we measure either state immediately, they appear statistically identical: both yield a 0 or 1 with equal 50% probability. A classical description would therefore treat these states as indistinguishable.

Yet, if we apply a specific operation to both before measuring, the results diverge completely. One state might interfere constructively to produce a deterministic 0, while the other interferes destructively to produce a deterministic 1. Figure 2.1 illustrates this interference, while the formal definition of such an operation is detailed in Section 2.1.2.

If the states were merely random distributions, they would remain random after the same operation. The fact that they can be distinguished deterministically implies that they possess an underlying structure that classical probabilities cannot capture [52]. Capturing this behavior mathematically requires replacing real probabilities with *complex probability amplitudes*.

¹While the physical realization of qubits varies across technologies (e.g., superconducting circuits [77], trapped ions [33], or neutral atoms [121]), this thesis operates at the logical level. We abstract away specific device physics to focus on algorithmic optimization and simulation.

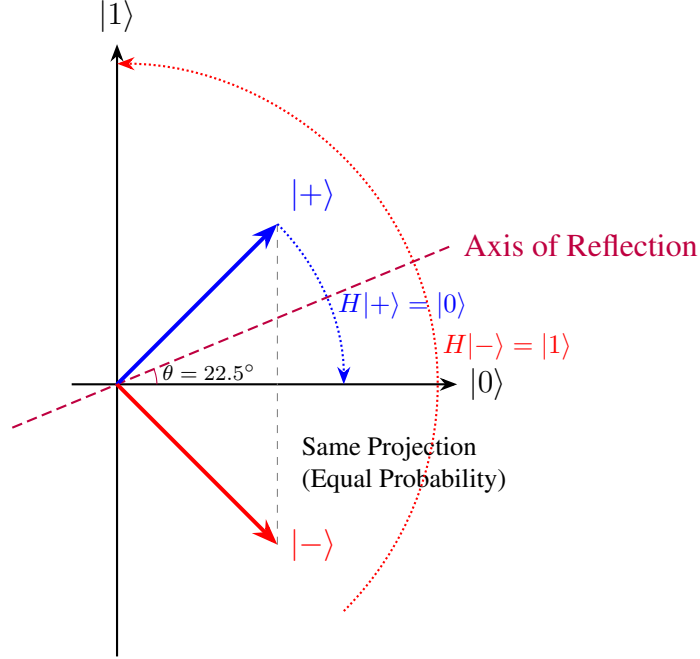


Figure 2.1: Visual representation of an interference example.

We formalize this linear algebraic structure using Dirac’s bra-ket notation, where quantum states are denoted as “kets” $|\psi\rangle$, and their conjugate transposes as “bras” $\langle\psi|$ [46]. In this formalism, a qubit state is treated as a column vector in a two-dimensional complex Hilbert space $\mathbb{C}^2$. We define the *computational basis* states, corresponding to the classical bit values 0 and 1, as the orthonormal basis vectors [99]:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \quad (2.1)$$

A general qubit state $|\psi\rangle$ is a linear combination of these basis vectors:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \quad (2.2)$$

where $\alpha, \beta \in \mathbb{C}$ are the probability amplitudes. The Born rule [29] dictates that the probability of measuring outcome $|0\rangle$ is $|\alpha|^2$ and outcome $|1\rangle$ is $|\beta|^2$, subject to the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. The corresponding dual vector $\langle\psi|$ is defined as the conjugate transpose $\langle\psi| = (|\psi\rangle)^\dagger = [\alpha^* \ \beta^*]$.

2.1.2 Quantum Gates

Just as classical logic gates (e.g., AND, NOT) manipulate bits, *quantum gates* manipulate qubits. Physically, the time evolution of a closed quantum system is described by linear operators acting on the state vector. To ensure that the sum of probabilities remains equal to 1 throughout the computation (conservation of the norm), these operators must be *unitary*.

A linear operator U is unitary if its conjugate transpose $U^\dagger$ is also its inverse:

$$U^\dagger U = U U^\dagger = I. \quad (2.3)$$

This property implies that quantum gates are inherently reversible. Acting a gate U on a state $|\psi\rangle$ yields a new state $|\psi'\rangle = U|\psi\rangle$. For a single qubit, the quantum gates are defined by 2×2 unitary matrices. Some common single-qubit gates include:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \quad (2.4)$$

The X gate is analogous to a classical NOT gate, as it swaps the probability amplitudes between $|0\rangle$ and $|1\rangle$. The Hadamard gate (H) is particularly significant as it generates superposition from computational basis states:

$$H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}, \quad H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}. \quad (2.5)$$

The two resulting states are also denoted $|+\rangle$ and $|-\rangle$, i.e.,

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}, \quad |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}. \quad (2.6)$$

In Figure 2.1, if we measure $|+\rangle$ or $|-\rangle$ in the computational basis, the squared magnitude of the amplitude for $|0\rangle$ or $|1\rangle$ is $\frac{1}{2}$ in both cases, denoting equal probability for measuring outcomes $|0\rangle$ or $|1\rangle$. However, after applying a Hadamard gate, we have

$$H|+\rangle = |0\rangle, \quad H|-\rangle = |1\rangle. \quad (2.7)$$

If we measure them at this time, the former produces $|0\rangle$ deterministically, and the latter results in $|1\rangle$ deterministically. This mathematical result, where $H|+\rangle$ deterministically returns to $|0\rangle$, precisely models the “constructive interference” described in the thought experiment in Section 2.1.1.

2.1.3 Multi-Qubit Systems and Entanglement

While single qubits exhibit superposition, the true power of quantum computing arises from the interaction of multiple qubits. Mathematically, the state space of an n -qubit system is the *tensor product* of the individual qubit spaces: $\mathcal{H} = \mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \dots \otimes \mathbb{C}^2 \cong \mathbb{C}^{2^n}$.

The basis states for this combined space are formed by the tensor product of single-qubit basis states. For a two-qubit system, the computational basis is:

$$|00\rangle = |0\rangle \otimes |0\rangle, \quad |01\rangle = |0\rangle \otimes |1\rangle, \quad |10\rangle = |1\rangle \otimes |0\rangle, \quad |11\rangle = |1\rangle \otimes |1\rangle. \quad (2.8)$$

A general two-qubit state is a superposition of these four basis states:

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle, \quad (2.9)$$

subject to the normalization condition $\sum |\alpha_{ij}|^2 = 1$.

Entanglement A multi-qubit state is said to be *separable* if it can be written as the product of individual qubit states, i.e., $|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$. However, most states in the Hilbert space are *entangled*, meaning they cannot be factored in this way. A canonical example is the *Bell state* (or EPR pair):

$$|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}. \quad (2.10)$$

In this state, measuring the first qubit yields a 0 or 1 with equal probability. However, the measurement outcome of the second qubit is perfectly correlated with the first, regardless of the spatial separation between them [99].

Multi-Qubit Gates To generate entanglement, we require multi-qubit gates. The most common two-qubit primitive is the *Controlled-NOT* (*CNOT*) gate. It flips the state of the *target* qubit if and only if the *control* qubit is in the state $|1\rangle$:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \quad (2.11)$$

The *CNOT* gate is essential for universality; any multi-qubit unitary operation can be decomposed into single-qubit gates and *CNOT* gates [19].

Another critical two-qubit gate, often used in hardware implementations and optimization, is the *Controlled-Z* (*CZ*) gate. Unlike *CNOT*, the *CZ* gate is symmetric with respect to its inputs:

$$CZ = \text{diag}(1, 1, 1, -1) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}. \quad (2.12)$$

These two primitives are locally equivalent. A *CNOT* can be implemented using a *CZ* gate by applying Hadamard gates to the target qubit before and after the interaction:

$$CNOT = (I \otimes H) \cdot CZ \cdot (I \otimes H). \quad (2.13)$$

Parameterized Gates Many gates supported by modern quantum devices take real-valued parameters. For example, the IBM QX5 quantum device supports the U_1 gate which takes one parameter and rotates a qubit about the x -axis (on the Bloch sphere), and the U_2 gate which takes two parameters for rotating about the x - and z -axes [47]. The matrix representations of U_1 and U_2 are:

$$U_1(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}, U_2(\phi, \lambda) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{bmatrix}. \quad (2.14)$$

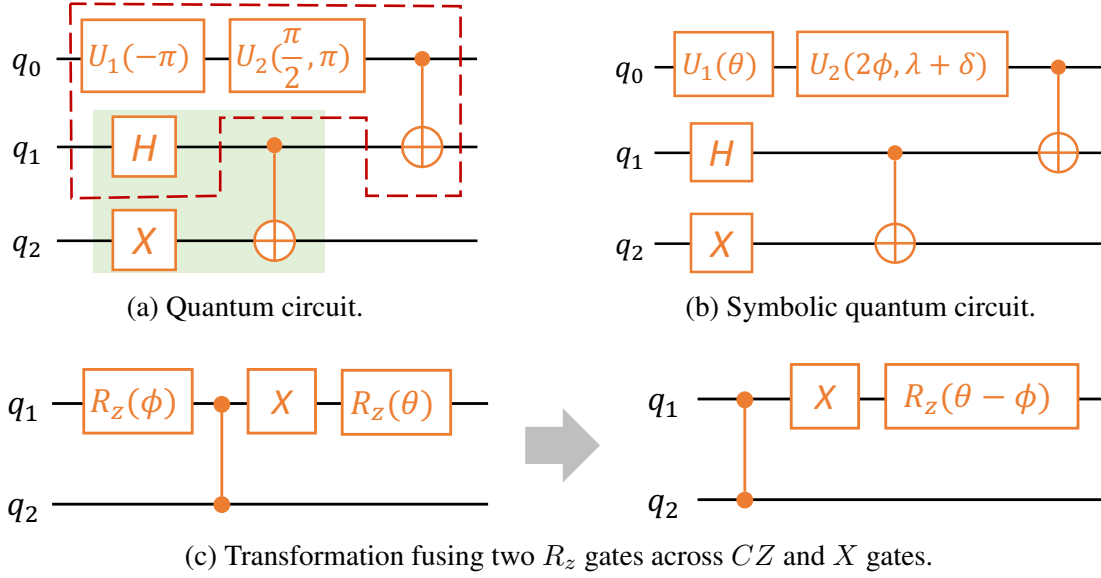


Figure 2.2: Quantum circuits and transformations. The green highlighted box in (a) is a subcircuit, while the red dashed area is not; the $U_1(-\pi)$ and X gates also form a subcircuit.

2.2 The Quantum Circuit Model

Quantum programs are typically represented as *quantum circuits* [99], as shown in Figure 2.2a. In this graphical model, horizontal wires represent qubits (carrying time-evolved state vectors), and boxes represent the quantum gates defined in Sections 2.1.2 and 2.1.3.

As established in the previous section, the state $|\psi\rangle$ of an n -qubit system resides in a 2^n -dimensional Hilbert space. It is written as a superposition of integer basis states:

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle, \quad (2.15)$$

where $|i\rangle$ represents the tensor product basis state corresponding to the binary representation of integer i . The *state vector* of an n -qubit system is defined as

$$\vec{\alpha} = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{2^n-1} \end{bmatrix}. \quad (2.16)$$

The semantics of a k -qubit gate is specified by a $2^k \times 2^k$ unitary complex matrix U , and applying the gate to a quantum circuit with state $|\psi\rangle$ results in a new state: $|\psi\rangle \rightarrow U |\psi\rangle$. For example, applying a single-qubit gate on the q -th qubit of a quantum system updates its state vector as follows:

$$\begin{bmatrix} \alpha_{f(i)} \\ \alpha_{f(i)+2^q} \end{bmatrix} \rightarrow \begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix} \times \begin{bmatrix} \alpha_{f(i)} \\ \alpha_{f(i)+2^q} \end{bmatrix}, \quad (2.17)$$

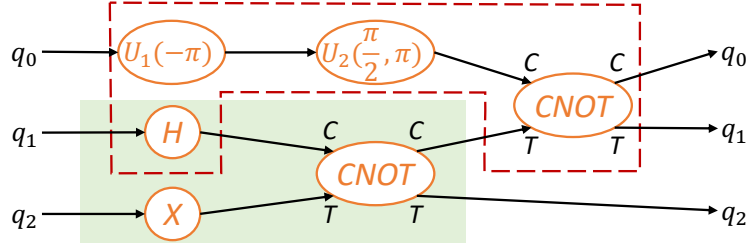


Figure 2.3: Graph representation for Figure 2.2a’s circuit. The green box (subcircuit, also convex subgraph) and red dashed area (not a subcircuit, also non-convex subgraph) match those of Figure 2.2a.

where $\begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix}$ is the 2×2 unitary complex matrix that represents the single-qubit gate, and $f(i) = 2^{q+1} \lfloor \frac{i}{2^q} \rfloor + (i \bmod 2^q)$ for all integers between 0 and $2^{n-1} - 1$.

The semantics of a quantum circuit over n qubits is given by a $2^n \times 2^n$ unitary complex matrix. This matrix can be computed from matrices of individual gates in a compositional manner, using matrix multiplications (for sequential composition of subcircuits that operate on the same qubits) and tensor products (for parallel composition of subcircuits that operate on different qubits). For example, the matrix for the circuit of Figure 2.2a is $(CNOT \otimes I) \cdot (U_2(\frac{\pi}{2}, \pi) \otimes CNOT) \cdot (U_1(-\pi) \otimes H \otimes X)$.

A circuit C' is a *subcircuit* of C if, for some qubit permutation, the matrix computation for C can be structured as $_ \cdot (M_{C'} \otimes I \otimes \cdots \otimes I) \cdot _$, where $M_{C'}$ is the matrix for C' . For example, the green box in Figure 2.2a highlights a subcircuit, while the red dashed area is not a subcircuit. The subcircuit notion is invariant under qubit permutation; e.g., the X and U_1 gates in Figure 2.2a also form a subcircuit. A circuit’s matrix is invariant under replacing one subcircuit with another that has the same matrix (but possibly different gates), which underpins peephole optimization for quantum circuits.

2.2.1 Circuit Representation

There are many ways to represent a quantum circuit. One way is to represent the circuit in a directed acyclic graph (DAG). In the graph representation, a circuit C over q qubits is represented as a directed graph G , where each gate over d qubits is a vertex with in- and out-degree d . Edges are labeled to distinguish between qubits in multi-qubit gates (e.g., the control and target qubits of a $CNOT$ gate). G also includes q sources and sinks, one for each qubit. Figure 2.3 illustrates the graph representation of Figure 2.2a’s circuit. As the figure also illustrates, subcircuits correspond to convex subgraphs². Chapter 4 uses the graph representation.

Although each circuit has a unique graph representation, it is often easier to deal with sequences. OpenQASM [41, 42] standardizes this sequence representation by expressing the circuit as straight-line code. This format is supported by nearly all quantum computing frameworks, enabling our implementations to interact seamlessly with off-the-shelf tools. Any topological order-

²For a graph G , G' is a convex subgraph of G if for any two vertices u and v in G' , every path in G from u to v is also contained in G' .

ing of the graph representation is a valid sequence representation for the circuit. For example, in sequence representation, Figure 2.2a can be represented as $U1(-\pi) \text{ q}[0]; U2(\pi/2, \pi) \text{ q}[0]; H \text{ q}[1]; X \text{ q}[2]; CNOT \text{ q}[1], \text{ q}[2]; CNOT \text{ q}[0], \text{ q}[1]$. We utilize this sequence representation extensively in Chapters 3 and 7.

Alternatively, quantum circuits can be compactly structured as a sequence of layers. In this layer representation, each layer contains mutually independent gates that can execute concurrently at the same time step on their respective qubits. This format closely mirrors how a quantum circuit is visually drawn and explicitly captures available parallelism. Using Figure 2.2a as an example, translating it into a layer representation groups the first three independent gates into the initial layer, the subsequent two gates into the second layer, and the final gate into the third layer. We adopt this layer representation in Chapter 6 to facilitate structural analysis.

Given a circuit C , its *size* ($|C|$) is the total number of gates used, while the *width* (n) represents the number of qubits. The *depth* (d) is the number of circuit layers, wherein each qubit participates in at most one gate.

2.2.2 Symbolic Circuits

Figure 2.2b shows *symbolic quantum circuits*, which include (symbolic) parameters ($\theta, \phi, \lambda, \delta$, etc.) and arithmetic operations on these parameters. The semantics of a *symbolic quantum circuit*, denoted $\llbracket \cdot \rrbracket$, has type $\llbracket C \rrbracket: \mathbb{R}^m \rightarrow \mathbb{C}^{2^q \times 2^q}$ where C is a circuit over m (symbolic) parameters and q qubits. For a vector of parameter values $\vec{p} \in \mathbb{R}^m$, $\llbracket C \rrbracket(\vec{p})$ is a $2^q \times 2^q$ unitary complex matrix representing a (concrete) quantum circuit over q qubits. For example, eq. (2.14) can be seen as defining the semantics of U_1 and U_2 as single-gate symbolic quantum circuits. The semantics of a multi-gate symbolic circuit (e.g., Figure 2.2b) is derived from that of single-gate circuits using matrix multiplications and tensor products exactly as for concrete circuits.

2.2.3 Circuit Equivalence and Transformations

In quantum computing, the quantum states $|\psi\rangle$ and $e^{i\beta}|\psi\rangle$ (for $\beta \in \mathbb{R}$) are said to be *equivalent up to global phase*, and from an observational point of view they are identical [99]. This leads to the following equivalence definition for symbolic circuits.

Definition 1 (Circuit Equivalence). *Two symbolic quantum circuits C_1 and C_2 are equivalent if:*

$$\forall \vec{p} \in \mathbb{R}^m. \exists \beta \in \mathbb{R}. \llbracket C_1 \rrbracket(\vec{p}) = e^{i\beta} \llbracket C_2 \rrbracket(\vec{p}). \quad (2.18)$$

That is, two circuits are equivalent if for every valuation of the parameters they differ only by a phase factor. The phase factor may in some cases be constant, but generally it may be different for different parameter values. For example, the matrix representations of U_1 and R_z gates are:

$$U_1(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}, R_z(\theta) = \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}. \quad (2.19)$$

The equivalence between these two gates, i.e.,

$$U_1(\theta) = e^{i\frac{\theta}{2}} R_z(\theta), \quad (2.20)$$

requires a parameter-dependent phase factor. Crucially for peephole optimization, circuit equivalence is invariant under replacing a subcircuit with an equivalent subcircuit.

A *circuit transformation* (C_T, C_R) is a pair of distinct equivalent circuits, where C_T is a *target circuit* to be matched with a subcircuit of the circuit being optimized, and C_R is a *rewrite circuit* that can replace the target circuit while maintaining equivalence of the optimized circuit and the input circuit. Figure 2.2c illustrates a circuit transformation.

Chapter 3

Equivalence Generation

Circuit equivalences and transformations are the key to a broad family of quantum circuit optimizers. In this chapter, we present the core algorithms behind Quartz [149], a system that systematically generates and utilizes these equivalences. Specifically, we describe how Quartz uses an algorithm called REPGEN to generate candidate circuit transformations by systematically exploring small circuits, and how it verifies the discovered transformations using an automated theorem prover.

3.1 System Overview

Figure 3.1 illustrates the high-level flow of Quartz, which operates in two distinct phases:

1. **Equivalence Generation (this chapter):** Quartz generates and verifies a comprehensive set of circuit transformations for a given gate set.
2. **Optimization (Chapter 4):** It uses these transformations to optimize input circuits via a cost-based backtracking search.

This chapter focuses on the Equivalence Generation phase. To ensure the generated transformations are exhaustive up to a certain size, Quartz uses an *interleaving* approach: it iteratively generates candidate circuits, eliminates redundancy, and verifies equivalences.

The process is driven by two core algorithmic components:

- **Representative-Based Generation (Section 3.3):** Instead of enumerating all exponentially many circuits, we introduce the REPGEN algorithm. It selectively extends only “representative” circuits that are unique up to equivalence, using hash-based fingerprints to prune the search space efficiently.
- **Symbolic Verification (Section 3.4):** To guarantee correctness, candidate equivalences are formally verified. We reduce the verification problem, which involves complex quantum gates and parameters, to quantifier-free formulas over nonlinear real arithmetic, which are then checked by an SMT solver.

The core data structure facilitating this process is the *Equivalent Circuit Class (ECC)*, which we define formally in the next section.

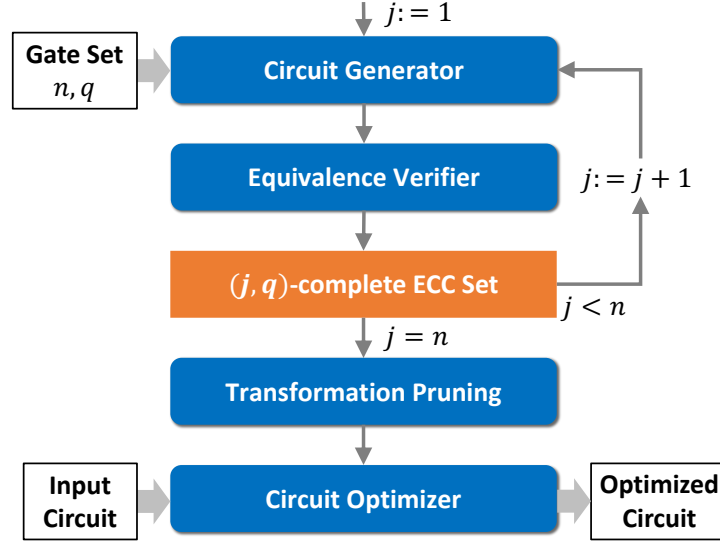


Figure 3.1: Quartz overview.

3.2 Equivalent Circuit Classes

We use *equivalent circuit classes* (ECCs) to represent many circuit transformations compactly. An ECC is a set of equivalent circuits. A transformation is *included* in an ECC if both its target and rewrite circuits are in the ECC. ECCs provide a compact representation of circuit transformations: an ECC with x circuits includes $x(x - 1)$ transformations.

A *circuit rewriting* is a sequence of applications of circuit transformations, which Quartz uses for optimization as illustrated in Figure 3.2. Figure 3.2a shows a common optimization that removes four Hadamard gates (i.e., H) by flipping a $CNOT$ gate. Figure 3.2b shows how to perform this optimization as a circuit rewriting consisting of three (more basic) circuit transformations, which are instances of the two transformations specified by the ECC set in Figure 3.2c.

3.2.1 Completeness for ECC sets

For any number of gates n , qubits q , and parameters m , assume a *finite* set $\mathcal{C}^{(n,q,m)}$ of circuits that can be constructed with at most n gates over q qubits and m parameters. The collection $\{\mathcal{C}^{(n,q,m)}\}_{n,q,m \in \mathbb{N}}$ is determined by the *gate set* $\mathcal{G}$ (finite set of possibly parametric gates) as well as a specification Σ of how parameter expressions may be constructed; e.g., a finite set of arithmetic operations and some bounds on the depth of expressions or the number of times each parameter can be used, ensuring finiteness of every $\mathcal{C}^{(n,q,m)}$. Henceforth, we fix the gate set $\mathcal{G}$, the parameter-expression specification Σ , and the number of parameters m ; and elide m by writing $\mathcal{C}^{(n,q)}$.

A transformation (C_T, C_R) is *subsumed* by an ECC set if there is a circuit rewriting from C_T to C_R that only uses transformations included in the ECC set. Sections 3.3 to 3.5 describe our approach for constructing a verified (n, q) -complete ECC set.

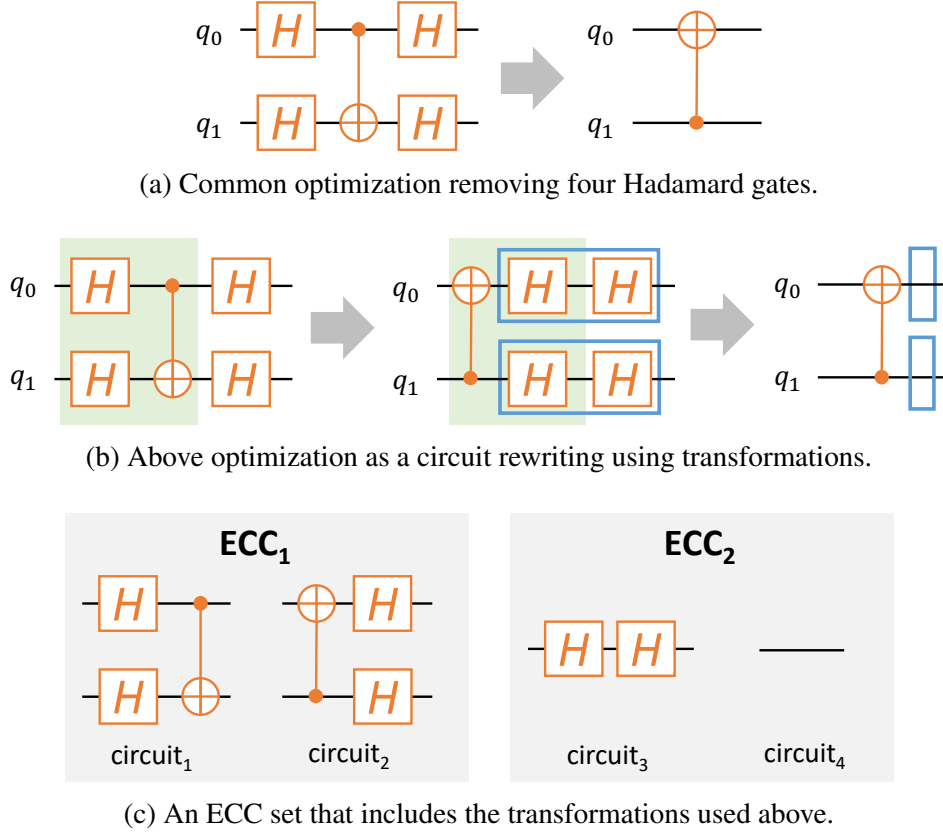


Figure 3.2: Illustrating optimization via transformations.

Definition 2 ((n, q) -Completeness). *Given $\mathcal{G}$, Σ , and m as above, an ECC set is (n, q) -complete (for $\mathcal{G}$, Σ , and m) if it subsumes all circuit transformations over $\mathcal{C}^{(n, q)}$.*

An (n, q) -complete ECC set can be used to rewrite any two equivalent circuits with at most n gates over q qubits to each other. Any ECC set is by default $(0, q)$ -complete because any transformation involves at least one gate in the target or rewrite circuits. An ECC set is $(1, q)$ -complete if it subsumes all possible transformations between single gates.

3.3 Circuit Generator

Quartz uses the sequence representation of quantum circuits to generate ECC set, developed in this section, which interleaves circuit generation and equivalence verification.

3.3.1 The REPGEN Algorithm

A straightforward way to generate an (n, q) -complete ECC set is to examine all circuits in $\mathcal{C}^{(n, q)}$, but there are exponentially many such circuits. To tackle this challenge, REPGEN uses *representative-based circuit generation*, which significantly reduces the number of circuits considered. The key idea is to extend a (j, q) -complete ECC set to a $(j + 1, q)$ -complete one by

Algorithm 1 REPGEN.

```
1: Input: Number of gates  $n$  and qubits  $q$ .
2: Output:  $(n, q)$ -complete ECC set.
3:  $\mathcal{D} = \emptyset$  // Hash table of circuit sets indexed by fingerprint
4:  $\mathcal{D}[\text{FINGERPRINT}(())] = \{()\}$  // Initialize with empty circuit
5:  $\mathcal{R}_0 = \{()\}$  // Representatives with 0 gates
6:  $\mathcal{S}_0 = \emptyset$  //  $(0, q)$ -complete ECC set
7: for  $j = 1$  to  $n$  do
8:   // Step 1: construct circuits (sequences) with  $j$  gates
9:   for each circuit  $L \in \mathcal{R}_{j-1}$  such that  $|L| = j - 1$  do
10:    for each gate  $g \in \mathcal{G}$  do
11:      for each  $\iota$  such that  $L' = L.(g \ \iota)$  satisfies  $\Sigma$  do
12:        if  $\text{DROPFIRST}(L') \in \mathcal{R}_{j-1}$  then
13:          add  $L'$  to  $\mathcal{D}[\text{FINGERPRINT}(L')]$ 
14:   // Step 2: examine circuits with equal fingerprints
15:    $\mathcal{S}_j = \bigcup \{\text{ECCIFY}(\gamma) : \gamma = \mathcal{D}[f] \text{ for some } f\}$ 
16:    $\mathcal{R}_j = \{\min(\mathcal{E}) : \mathcal{E} \in \mathcal{S}_j\}$  // Under  $\prec$  (Definition 3)
17: return  $\{\mathcal{E} \in \mathcal{S}_n : |\mathcal{E}| \geq 2\}$  // Ignore singleton ECCs
18:
19: // Partition circuits into verified ECCs
20: function ECCIFY( $\gamma$ )
21:    $\hat{\mathcal{S}} = \emptyset$  // Set of ECCs for circuits in  $\gamma$ 
22:   for each circuit  $C \in \gamma$  do
23:     if  $C$  is equivalent to a circuit in ECC  $\mathcal{E} \in \hat{\mathcal{S}}$  then
24:        $\hat{\mathcal{S}} = \hat{\mathcal{S}} \setminus \{\mathcal{E}\} \cup \{\mathcal{E} \cup \{C\}\}$  // Add  $C$  to  $\mathcal{E}$ 
25:     else
26:        $\hat{\mathcal{S}} = \hat{\mathcal{S}} \cup \{\{C\}\}$  // Create a new ECC for  $C$ 
27:   return  $\hat{\mathcal{S}}$ 
```

selecting a representative circuit for each ECC and using these representatives to build larger circuits.

Let $|L|$, $\text{DROPFIRST}(L)$, and $\text{DROPLAST}(L)$ denote the number of gates in a circuit L , its suffix with $|L| - 1$ gates, and its prefix with $|L| - 1$ gates. Note that each of the latter two represents a subcircuit. Fix an arbitrary total order over single-gate circuits (i.e., $\mathcal{C}^{(1,q)}$), and lift it to a total order of circuits (i.e., sequences).

Definition 3 (Circuit Precedence). Say L_1 precedes L_2 , written $L_1 \prec L_2$, if $|L_1| < |L_2|$, or if $|L_1| = |L_2|$ and L_1 is lexicographically smaller than L_2 .

Algorithm 1 lists the REPGEN algorithm, which proceeds in rounds and maintains a database $\mathcal{D}$ of circuits grouped by fingerprints (defined below), a (j, q) -complete ECC set $\mathcal{S}_j$, and a *representative set* (set of representatives) $\mathcal{R}_j$ for ECCs in $\mathcal{S}_j$. The j -th round produces a (j, q) -complete ECC set from the $(j - 1, q)$ -complete ECC set generated in the previous round. Each round proceeds in two steps.

Step 1: Constructing circuits. Before the first round, the initial ECC set is $\mathcal{S}_0 = \emptyset$ and the representative set is $\mathcal{R}_0 = \{()\}$, i.e., a singleton set consisting of the empty circuit (over q qubits). In the j -th round, REPGEN uses the $(j-1, q)$ -complete ECC set $\mathcal{S}_{j-1}$ and its representative set $\mathcal{R}_{j-1}$ computed previously, and constructs possible size- j circuits by appending a single gate to each circuit in $\mathcal{R}_{j-1}$ with size $j-1$. REPGEN enumerates all possible gates g and arguments ι according to $\mathcal{G}$ and Σ . For each generated circuit L' , REPGEN checks if $\text{DROPFIRST}(L')$ is a representative from the previous round. If so, REPGEN concludes that L' extends existing representatives and must be considered in generating $\mathcal{S}_j$. Otherwise, circuit L' is considered redundant and ignored.

To identify potentially equivalent circuits, REPGEN computes circuit *fingerprints*, and uses them as keys for storing circuits in the hash table $\mathcal{D}$. The fingerprint is computed using fixed, randomly selected parameter values and quantum states. Recall that for a circuit C over q qubits and m parameters, and parameter values $\vec{p} \in \mathbb{R}^m$, $\llbracket C \rrbracket(\vec{p})$ is a (concrete) $2^q \times 2^q$ complex matrix. The fingerprint of a circuit C is

$$\text{FINGERPRINT}(C) = |\langle \psi_0 | \llbracket C \rrbracket(\vec{p}_0) | \psi_1 \rangle|, \quad (3.1)$$

where the parameter values $\vec{p}_0$ and quantum states $|\psi_0\rangle$ and $|\psi_1\rangle$ are fixed and randomly selected, and $|\cdot|$ denotes the modulus of a complex number. With infinite precision, eqs. (2.18) and (3.1) ensure that equivalent circuits have identical fingerprints. In implementation, Quartz uses finite-precision floating-point arithmetic and additionally checks circuits with fingerprints close to each other to account for floating-point errors.

Step 2: Examining circuits with equal fingerprints. In Step 1, REPGEN generates circuits and stores them in the hash table $\mathcal{D}$ grouped by their fingerprints. In Step 2, REPGEN partitions each set $\gamma = \mathcal{D}[f]$ of potentially equivalent circuits into a verified ECC set using the function ECCIFY. ECCIFY considers each circuit in γ and checks if it is equivalent to some existing ECC in γ by querying the verifier (Section 3.4); the circuit is then either added to the matching ECC, or becomes a new singleton ECC. REPGEN then combines the ECC sets for each γ to get the ECC set $\mathcal{S}_j$.

Having constructed an ECC set $\mathcal{S}_j$, REPGEN computes $\mathcal{S}_j$'s representative set $\mathcal{R}_j$, which is the set of representatives of the ECCs in $\mathcal{S}_j$ (Algorithm 1 line 16). The representative of an ECC is its $\prec$ -minimum circuit (Definition 3). During the operation of REPGEN, singleton ECCs are important: their representatives must be considered when generating circuits in the next round, and they may grow to non-singleton ECCs as more circuits are generated. However, singleton ECCs in $\mathcal{S}_n$ ultimately yield no transformations, so we remove them from the result of the REPGEN algorithm (line 17).

3.3.2 Correctness of REPGEN

When constructing circuits of size j (in round j), REPGEN only considers circuits L' that extend previously constructed representatives, i.e., $\text{DROPLAST}(L') \in \mathcal{R}_{j-1}$, and only when the extension leads to $\text{DROPFIRST}(L') \in \mathcal{R}_{j-1}$. In this section we prove that in spite of that, REPGEN always generates an (n, q) -complete ECC set. Below we use $\mathcal{D}_j$ to denote the value of $\mathcal{D}$ after

Step 1 in REPGEN's j -th round (i.e., at Algorithm 1 line 15) and $\mathcal{D}_0$ for the initial value of $\mathcal{D}$ (line 4).

Lemma 1. *Algorithm 1 maintains the following invariants (for any $1 \leq j \leq n$, and writing $\sqsubseteq$ for Hoare ordering, i.e., $\mathcal{X} \sqsubseteq \mathcal{Y} \equiv \forall X \in \mathcal{X}. \exists Y \in \mathcal{Y}. X \subseteq Y$):*

1. $\mathcal{D}_{j-1} \sqsubseteq \mathcal{D}_j$, $\mathcal{S}_{j-1} \sqsubseteq \mathcal{S}_j$, and $\mathcal{R}_{j-1} \subseteq \mathcal{R}_j$;
2. for any $L \in \mathcal{C}^{(n,q)}$, $L \in \bigcup \mathcal{D}_j$ iff $|L| \leq j$ and either $L = ()$ or $\text{DROPFIRST}(L) \in \mathcal{R}_{j-1}$, $\text{DROPLAST}(L) \in \mathcal{R}_{j-1}$; and
3. for any $L \in \mathcal{C}^{(n,q)}$, $L \in \mathcal{R}_j$ iff $|L| \leq j$ and L does not have a $\prec$ -smaller equivalent circuit in $\mathcal{C}^{(n,q)}$.

Proof. For item 1, $\mathcal{D}_{j-1} \sqsubseteq \mathcal{D}_j$ and $\mathcal{S}_{j-1} \sqsubseteq \mathcal{S}_j$ follow from the monotonic updating of $\mathcal{D}$ (i.e., circuits are only added), and from monotonicity (w.r.t. Hoare ordering) of ECCIFY. To see that $\mathcal{R}_{j-1} \subseteq \mathcal{R}_j$, observe that in the j -th round, all circuits constructed are of size j , so any circuit added to an existing ECC has more gates than its representative in $\mathcal{R}_{j-1}$, which will therefore remain its representative in $\mathcal{R}_j$ (recall that if $|L| < |L'|$ then $L \prec L'$).

We prove item 2 by induction on j . Both the base case ($j = 1$) and the induction step follow from Algorithm 1 lines 9–13, combined with item 1 and either the definition of $\mathcal{D}_0$ or the induction hypothesis.

We prove item 3 by induction on j . Slightly generalizing from the statement above, we take the base case to be $j = 0$, which follows from line 5. In the induction step, for sufficiency, consider a circuit L , $1 \leq |L| \leq j$, with no $\prec$ -smaller equivalent circuit. (The $|L| = 0$ case follows from line 5 and item 1.) Both $\text{DROPFIRST}(L)$ and $\text{DROPLAST}(L)$ are of size $\leq j - 1$ with no $\prec$ -smaller equivalent circuits (if either had a $\prec$ -smaller equivalent circuit, we could use it to construct a $\prec$ -smaller equivalent circuit for L). By the induction hypothesis, $\text{DROPFIRST}(L), \text{DROPLAST}(L) \in \mathcal{R}_{j-1}$, so by item 2, $L \in \bigcup \mathcal{D}_j$. By lines 15–16, $\mathcal{R}_j$ includes the $\prec$ -minimal element of each class of equivalent circuits in $\bigcup \mathcal{D}$, so it must include L . Necessity follows from sufficiency, combined with the fact that two circuits in $\mathcal{R}_j$ cannot be equivalent and that $\mathcal{R}_j$ does not contain circuits of size greater than j . $\square$

Theorem 1 (REPGEN). *In Algorithm 1, every $\mathcal{S}_j$ ($0 \leq j \leq n$) is a (j, q) -complete ECC set, and the algorithm returns an (n, q) -complete ECC set.*

Proof. We proceed using proof by contradiction. Let j be the smallest such that $\mathcal{S}_j$ is not (j, q) -complete. We must have $j > 0$, with $\mathcal{S}_{j-1}$ $(j-1, q)$ -complete, and by Lemma 1 item 1 $\mathcal{S}_j$ is also $(j-1, q)$ -complete. (As $\mathcal{S}_{j-1} \sqsubseteq \mathcal{S}_j$, $\mathcal{S}_j$ only includes more transformations.) Let (L, L') be the minimal (under the pairwise lexicographic lifting of $\prec$) pair of equivalent circuits of size $\leq j$ that cannot be rewritten to each other using transformations included in $\mathcal{S}_j$. We must have $|L'| = j$, since otherwise $|L|, |L'| \leq j - 1$, but $\mathcal{S}_j$ is $(j-1, q)$ -complete.

If $\text{DROPFIRST}(L')$ has a $\prec$ -smaller equivalent circuit, then $\mathcal{S}_j$ can rewrite L' to a $\prec$ -smaller equivalent circuit, which it must also not be able to rewrite to L , contradicting the minimality of L' . Therefore, $\text{DROPFIRST}(L')$ does not have a $\prec$ -smaller equivalent circuit. The same argument works for $\text{DROPLAST}(L')$. Therefore, by using Lemma 1 item 3 we get $\text{DROPFIRST}(L') \in \mathcal{R}_{j-1}$, $\text{DROPLAST}(L') \in \mathcal{R}_{j-1}$, and by Lemma 1 item 2, $L' \in \mathcal{D}_j$. But if $L' \in \mathcal{D}_j$ then either $\mathcal{S}_j$ includes a transformation that rewrites L' to a smaller equivalent circuit, that it cannot rewrite

to L , contradicting the minimality of L' ; or L' does not have a $\prec$ -smaller equivalent circuit, contradicting the definition of the pair (L, L') . $\square$

3.3.3 Complexity of REPGEN

We analyze the time complexity of REPGEN (its space complexity is the same). First, observe that the number of single-gate circuits $|\mathcal{C}^{(1,q)}| - 1$, which is determined by the gate set $\mathcal{G}$, parameter-expression specification Σ , number of qubits q , and parameters m , provides an upper bound for the number of single-gate extensions of any existing circuit. (The -1 is due to $() \in \mathcal{C}^{(1,q)}$.) This *characteristic* of $\mathcal{G}$, Σ , q , and m , denoted $\text{ch}(\mathcal{G}, \Sigma, q, m) = |\mathcal{C}^{(1,q)}| - 1$, bounds the number of iterations of the loops in Algorithm 1 lines 10 and 11 in each round. (The bound may not be tight as Σ may impose more restrictions, e.g., single use of parameters.)

While $\sum_{j=0}^n \text{ch}(\mathcal{G}, \Sigma, q, m)^j$ provides a trivial upper bound on the complexity of REPGEN, the following theorem shows that REPGEN's running time can be bounded using the number of *resulting representatives* $|\mathcal{R}_n|$. In practice, this number is significantly smaller than $\text{ch}(\mathcal{G}, \Sigma, q, m)^n$ (see Table 4.5).

Theorem 2 (Complexity of REPGEN). *The time complexity of Algorithm 1, excluding the verification part (line 15), is*

$$O(|\mathcal{R}_n| \cdot \text{ch}(\mathcal{G}, \Sigma, q, m) \cdot n).$$

Proof. The j -th round of Algorithm 1 considers circuits from $\mathcal{R}_{j-1}$ with size $j - 1$, and for each one it considers at most $\text{ch}(\mathcal{G}, \Sigma, q, m)$ possible extensions. It takes $O(n)$ to construct a new circuit and add it to $\mathcal{D}$. (We assume $O(1)$ complexity for hash table insert and lookup, i.e., we use average and amortized complexity.) Summing over all rounds of Algorithm 1, and recalling that $\mathcal{R}_j \subseteq \mathcal{R}_n$ (Lemma 1 item 1):

$$\begin{aligned} & \sum_{j=1}^n |\{L \in \mathcal{R}_{j-1} : |L| = j - 1\}| \cdot \text{ch}(\mathcal{G}, \Sigma, q, m) \cdot n \\ & \leq |\mathcal{R}_n| \cdot \text{ch}(\mathcal{G}, \Sigma, q, m) \cdot n. \end{aligned}$$

$\square$

Note that if $\mathcal{G}$, Σ , q , and m are considered as constant then the time complexity of Algorithm 1 is $O(|\mathcal{R}_n| \cdot n)$.

Table 4.5 lists some empirical $\text{ch}(\mathcal{G}, \Sigma, q, m)$ and $|\mathcal{R}_n|$ values.

3.4 Equivalence Verifier

Given two circuits C_1 and C_2 over q qubits and m parameters, the verifier checks if they are equivalent (i.e., up to a global phase). Recalling eq. (2.18), that means checking if

$$\forall \vec{p} \in \mathbb{R}^m. \exists \beta \in \mathbb{R}. \llbracket C_1 \rrbracket(\vec{p}) = e^{i\beta} \llbracket C_2 \rrbracket(\vec{p}). \quad (3.2)$$

Note that the equality here is between two $2^q \times 2^q$ complex matrices.

There are two challenges in automatically checking eq. (2.18). One is the quantifier alternation, which may be needed to account for global phase; the other is the use of trigonometric function, which is common in quantum gates' matrix representations. For example, the U_3 gate supported by the IBM QX5 quantum processor has the following matrix representation:

$$\llbracket U_3 \rrbracket(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\frac{\theta}{2}) & -e^{i\lambda} \sin(\frac{\theta}{2}) \\ e^{i\phi} \sin(\frac{\theta}{2}) & e^{i(\phi+\lambda)} \cos(\frac{\theta}{2}) \end{bmatrix}. \quad (3.3)$$

While some SMT solvers support quantifiers and trigonometric functions [20, 38], preliminary attempts showed they cannot directly prove eq. (2.18) for the circuit transformations generated by Quartz. Quartz's verification approach is therefore to reduce eq. (2.18) to a quantifier-free formula over nonlinear real arithmetic by eliminating both the quantification over β and the trigonometric functions. The resulting verification conditions are then checked using an SMT solver.

3.4.1 Phase Factors

To eliminate the existential quantification over the phase β , Quartz's verifier searches over a finite space of linear combinations of the parameters $\vec{p}$ for a value that can be used for β . We consider $\beta(\vec{p}) = \vec{a} \cdot \vec{p} + b$, where $\vec{a} \in A$ and $b \in B$ for some *finite* sets $A \subseteq \mathbb{R}^m$ and $B \subseteq \mathbb{R}$. Given circuits C_1 and C_2 , Quartz finds candidates for the coefficients $\vec{a}$ and b using an approach similar to the one it uses for generating candidate transformations. Quartz selects random parameter values $\vec{p}_0$ and quantum states $|\psi_0\rangle$ and $|\psi_1\rangle$ and finds all combinations of $\vec{a}$ and b as above that satisfy the following equation up to a small floating-point error (note that unlike eq. (3.1), $|\cdot|$ is not used):

$$\langle \psi_0 | \llbracket C_1 \rrbracket(\vec{p}_0) | \psi_1 \rangle = e^{i(\vec{a} \cdot \vec{p}_0 + b)} \cdot \langle \psi_0 | \llbracket C_2 \rrbracket(\vec{p}_0) | \psi_1 \rangle, \quad (3.4)$$

For every such candidate coefficients $\vec{a}$ and b , we then attempt to verify the following equation,

$$\forall \vec{p} \in \mathbb{R}^m. \llbracket C_1 \rrbracket(\vec{p}) = e^{i(\vec{a} \cdot \vec{p} + b)} \llbracket C_2 \rrbracket(\vec{p}), \quad (3.5)$$

which unlike eq. (2.18), does not existentially quantify over β . If eq. (3.5) holds for some candidate coefficients, then C_1 and C_2 are verified to be equivalent. Otherwise, we consider the transformation given by C_1 and C_2 to fail verification, but that case does not occur in our experiments.

3.4.2 Trigonometric Functions

Matrices of parametric quantum gates we encounter only use their parameters inside arguments to $\sin$ or $\cos$ (after applying Euler's formula). Under this assumption, we reduce eq. (3.5) to nonlinear real arithmetic in three steps. First, we eliminate expressions such as $\frac{\theta}{2}$ that occur in some quantum gates (e.g., eq. (3.3)) by introducing a fresh variable $\theta' = \frac{\theta}{2}$ and substituting $\theta' + \theta'$ for θ . After this step, all arguments to $\sin$ and $\cos$ are linear combinations of variables and constants (e.g., from phase factors) with integer coefficients. Second, we exhaustively apply Euler's formula $e^{i\phi} = \cos \phi + i \sin \phi$, and trigonometric identities for parity and sum of angles: $\sin(-x) = -\sin(x)$, $\cos(-x) = \cos(x)$, $\sin(x+y) = \sin(x)\cos(y) + \cos(x)\sin(y)$, and

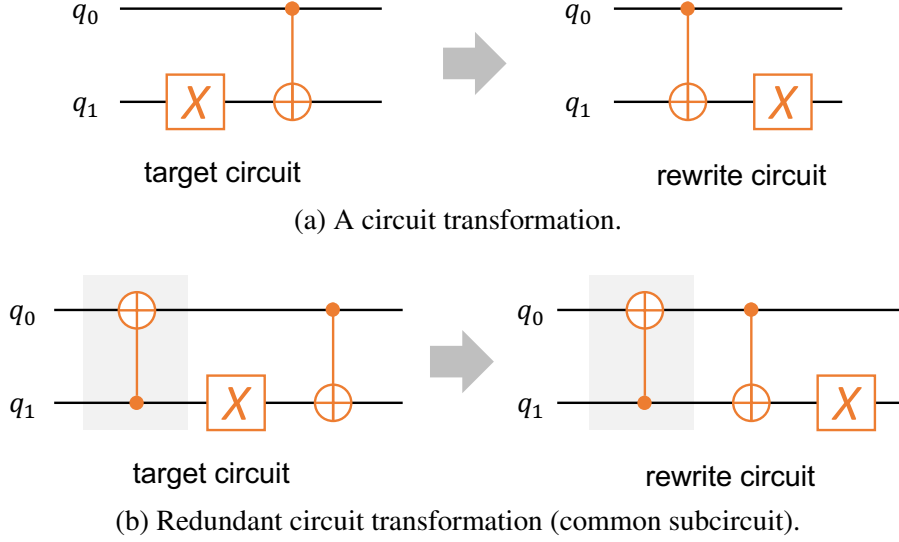


Figure 3.3: Illustrating a redundant circuit transformation.

$\cos(x + y) = \cos(x)\cos(y) - \sin(x)\sin(y)$. After these steps, $\sin$ and $\cos$ are only applied to atomic terms (variables and constants). For each constant c , we require precise symbolic expressions for $\sin(c)$ and $\cos(c)$ (e.g., $\sin(\frac{\pi}{4}) = \frac{\sqrt{2}}{2}$), and eliminate $\sin$ and $\cos$ over constants using these expressions. Third, for every variable t such that $\sin(t)$ or $\cos(t)$ is used we substitute s_t for $\sin(t)$ and c_t for $\cos(t)$, where s_t and c_t are fresh variables with a constraint $s_t^2 + c_t^2 = 1$, which fully eliminates trigonometric functions.

Ultimately, we formulate eq. (3.5) as a problem in quantifier-free nonlinear real arithmetic, allowing an SMT solver to automatically verify the equivalence. Because solver performance can vary, Section 4.3.4 also discusses alternative verification backends.

3.5 Pruning Redundant Transformations

Quartz applies two pruning steps after REPGEN generates an (n, q) -complete ECC set to further eliminate redundancy. These steps maintain (n, q) -completeness while reducing the number of transformations the optimizer needs to consider.

3.5.1 ECC Simplification

All ECCs generated by REPGEN have circuits with exactly q qubits. For each ECC, a qubit (or a parameter) is *unused* if all circuits in the ECC do not operate on the qubit (or the parameter). An *ECC simplification* pass removes all unused qubits and parameters from each ECC. After this pass, some ECCs may become identical, among which only one is kept.

Because there is no specific order on parameters in a circuit, Quartz also finds identical ECCs under a permutation of the parameters and maintains only one of them.

3.5.2 Common Subcircuit Pruning

Quartz eliminates transformations whose target and rewrite circuits include a common subcircuit at the beginning or the end. Figure 3.3 illustrates this *common subcircuit pruning*; the common subcircuit is highlighted in grey. Theorem 3 explains why such transformations are always redundant.

Definition 4. A subset of gates C' in a circuit C is a subcircuit at the beginning of C if all gates in C' are topologically before all gates in $C \setminus C'$. Similarly, a subset of gates C' in a circuit C is a subcircuit at the end of C if all gates in C' are topologically after all gates in $C \setminus C'$.

Theorem 3. For any two quantum circuits C_1 and C_2 with a common subcircuit at the beginning or the end, if C_1 and C_2 are equivalent, then eliminating the common subcircuit from C_1 and C_2 generates two equivalent circuits.

Proof. Recall that $\llbracket C \rrbracket(\vec{p})$ (for all $\vec{p}$ —we elide $\vec{p}$ in this proof) denotes the matrix representation of circuit C . Let $\llbracket C \rrbracket^\dagger$ be the conjugate transpose of $\llbracket C \rrbracket$, and recall that as $\llbracket C \rrbracket$ is unitary, we have $\llbracket C \rrbracket^\dagger \llbracket C \rrbracket = \llbracket C \rrbracket \llbracket C \rrbracket^\dagger = I$. Let C_s denote the common subcircuit shared by C_1 and C_2 . Let C'_1 and C'_2 represent the new circuits obtained by removing C_s from C_1 and C_2 . When C_s is a common subcircuit at the beginning of C_1 and C_2 , the matrix representations for the new circuits are $\llbracket C'_i \rrbracket = \llbracket C_s \rrbracket^\dagger \llbracket C_i \rrbracket$, where $i = 1, 2$. Equivalence between C_1 and C_2 implies the existence of β such that $\llbracket C_1 \rrbracket = e^{i\beta} \llbracket C_2 \rrbracket$, therefore $\llbracket C'_1 \rrbracket = e^{i\beta} \llbracket C'_2 \rrbracket$. The case where C_s is a common subcircuit at the end is similar. $\square$

Theorem 3 shows that every transformation pruned in common subcircuit pruning must be subsumed by other transformations (assuming initial (n, q) -completeness).

Observe that if two circuits have a common subcircuit at the beginning (resp. the end), then they must have a common gate at the beginning (resp. the end). Therefore, to implement common subcircuit pruning, Quartz only checks for a single common gate at the beginning or the end.

3.6 Summary

In this chapter, we established a method to automatically generate and verify a complete set of circuit transformations. However, possessing a set of transformations is only the first step. The challenge remains: how do we navigate the vast space of equivalent circuits to find the optimal one? Chapters 4 to 6 address this by introducing a search-based superoptimizer, integrating it with a rule-based optimizer, and scaling it up for large circuits.

Chapter 4

Quartz: Search-based Superoptimization

Given the verified Equivalent Circuit Classes (ECCs) generated in Chapter 3, the optimization problem transforms into a search problem: identifying the sequence of valid rewrites that minimizes a specific cost function.

In this chapter, we present Quartz’s *circuit optimizer*, which solves this problem using a cost-based backtracking search algorithm adapted from TASO [69]. Unlike greedy approaches that may get stuck in local minima, Quartz’s optimizer can explore cost-preserving (or temporarily cost-increasing) transformations to discover deep optimization opportunities.

We also detail the system implementation required to make this search practical, including preprocessing passes (Rotation Merging and Toffoli Decomposition) that handle optimizations difficult to capture via local search in Section 4.2. We then present a detailed evaluation comparing Quartz against state-of-the-art optimizers like Qiskit [6], t|ket> [130], and VOQC [61] in Section 4.4.

4.1 Circuit Optimizer

Quartz’s *circuit optimizer* applies the verified transformations generated by the generator to find an optimized equivalent circuit for a given input circuit (see Figure 3.1).

A key step is computing $\text{APPLY}(C, T)$, the set of circuits that can be obtained by applying transformation $T = (C_T, C_R)$ to circuit C . This involves finding all possible ways to match C_T with a subcircuit of C . Quartz’s optimizer uses a *graph representation for circuits*, explained below, to implement this operation. In the graph representation, subcircuits correspond to convex subgraphs,¹ and Quartz adapts the graph-matching procedure from TASO [69] to find all matches between C_T and a convex subgraph of C .

In the graph representation, a circuit C over q qubits is represented as a directed graph G , where each gate over d qubits is a vertex with in- and out-degree d . Edges are labeled to distinguish between qubits in multi-qubit gates (e.g., the control and target qubits of a $CNOT$ gate). G also includes q sources and sinks, one for each qubit. Figure 2.3 illustrates the graph representation of Figure 2.2a’s circuit. As the figure also illustrates, subcircuits correspond to convex

¹For a graph G , G' is a convex subgraph of G if for any two vertices u and v in G' , every path in G from u to v is also contained in G' .

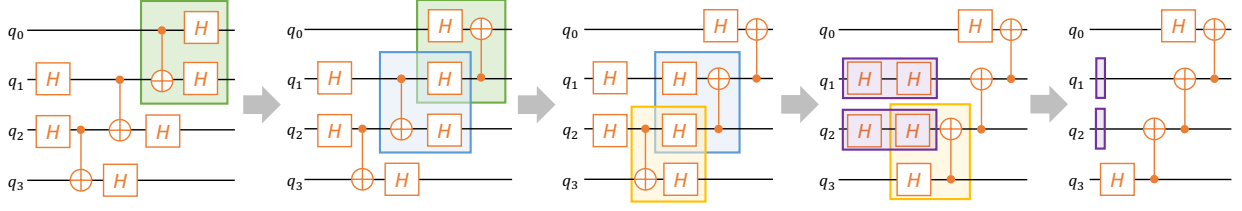


Figure 4.1: A transformation sequence applied by Quartz that reduces the total gate count in the $\text{gf2}^4.\text{mult}$ circuit by swapping the control and target qubits of three $CNOT$ gates. Note that the first three transformations do not reduce gate count.

Algorithm 2 Cost-Based Backtracking Search Algorithm.

```

1: Inputs: Verified transformations  $\mathcal{T}$ , a cost model  $\text{COST}(\cdot)$ , a hyper-parameter  $\gamma$ , and an input circuit  $C_{in}$ .
2: Output: an optimized circuit  $C_{best}$ 
3: //  $\mathcal{Q}$  is a priority queue of circuits sorted by their  $\text{COST}(\cdot)$ .
4:  $\mathcal{Q} = \{C_{in}\}$ 
5:  $C_{best} = C_{in}$ 
6:  $\mathcal{D}_{seen} = \{C_{in}\}$ 
7: while  $\mathcal{Q} \neq \emptyset$  and the search has not timed out do
8:    $C = \mathcal{Q}.\text{dequeue}()$ 
9:   if  $\text{COST}(C) < \text{COST}(C_{best})$  then
10:     $C_{best} = C$ 
11:   for each transformation  $T \in \mathcal{T}$  do
12:     for each  $C_{new} \in \text{APPLY}(C, T) \setminus \mathcal{D}_{seen}$  do
13:       if  $\text{COST}(C_{new}) < \gamma \cdot \text{COST}(C_{best})$  then
14:          $\mathcal{Q}.\text{enqueue}(C_{new})$ 
15:          $\mathcal{D}_{seen} = \mathcal{D}_{seen} \cup \{C_{new}\}$ 
16: return  $C_{best}$ 

```

subgraphs.

The optimizer first converts an (n, q) -complete ECC set into a set of transformations (in the graph representation). For each ECC with x equivalent circuits $C_1, \dots, C_x$, the optimizer considers a pair of transformations between the representative and each other circuit. For example, if C_1 is the representative circuit in the ECC, then the optimizer considers transformations $C_1 \rightarrow C_i$ and $C_i \rightarrow C_1$ for $1 < i \leq x$. These $2(x - 1)$ transformations guarantee that any two circuits from the same ECC are reachable from each other.

To optimize an input circuit using the above transformations, the optimizer uses a *cost-based backtracking search* algorithm adapted from TASO [69, 70]. The search is guided by a cost function $\text{COST}(\cdot)$ that maps circuits to real numbers. In our evaluation, the cost is given by the number of gates in a circuit, but other cost functions are possible.

Algorithm 2 shows the pseudocode of our search algorithm. To find an optimized circuit, candidate circuits are maintained in a priority queue $\mathcal{Q}$. At each iteration, the lowest-cost circuit C is dequeued, and Quartz applies all transformations to get equivalent new circuits C_{new} , which are enqueued into $\mathcal{Q}$ for further exploration. Circuits considered in the past are ignored using

$\mathcal{D}_{seen}$.

The search is controlled by a hyper-parameter γ . Quartz ignores candidate circuits whose cost is greater than γ times the cost of the current best circuit C_{best} . The parameter γ trades off between search time and the search’s ability to avoid local minima. For $\gamma = 1$, Algorithm 2 becomes a greedy search that only accepts transformations that strictly improve cost. On the other hand, a higher value for γ enables application of transformations that do not immediately improve the cost, which may later lead to otherwise inaccessible optimization opportunities. For example, Figure 4.1 depicts a sequence of five transformations that reduce the total gate count in the `gf2^4_mult` (see Section 4.4.2) circuit by four, via flipping three *CNOT* gates; note that the first three transformations do not reduce the gate count at all.

Quartz’s circuit optimizer is designed to optimize circuits before *mapping*. *Circuit mapping* converts a quantum circuit to an equivalent circuit that satisfies hardware constraints of a given quantum processor. These constraints include connectivity restrictions between qubits and the directions to perform multi-qubit operations. While transformations discovered by Quartz are also applicable to circuits after mapping, applying them naively may break hardware constraints. Therefore, we leave it as future work to build an optimizer for after-mapping circuits using Quartz’s transformations.

4.2 Preprocessing

Before invoking Quartz’s optimizer, Quartz preprocesses circuits by applying two optimizations: *rotation merging* and *Toffoli decomposition* [97]. Our preliminary experiments show that an approach solely based on local transformations and a cost-based search cannot reproduce these optimization passes for large circuits. Rotation merging combines multiple R_z/U_1 gates that may be arbitrarily far apart (separated by *X* or *CNOT* gates), and appears difficult to represent as local circuit transformations. Toffoli decomposition decomposes a Toffoli gate into the Nam gate set, which involves simultaneous transformation of 15 quantum gates and interacts with rotation merging [97, p. 11]. We therefore implement these two optimization passes from prior work [97] as a preprocessing step. Toffoli decomposition requires selecting a polarity for each Toffoli gate, which is computed heuristically by prior work [97]. Instead, we use a greedy approach: we process the Toffoli gates sequentially and for each gate we consider both polarities and greedily pick the one that results in fewer gates after rotation merging.

For the Nam and IBM gate sets, Quartz directly applies rotation merging and Toffoli decomposition as a preprocessing step before the optimizer. For the Rigetti gate set, which includes *CZ* rather than *CNOT*, the algorithm from the prior work [97] is not directly applicable; therefore, Quartz uses several additional preprocessing steps, as follows. Rather than directly transpiling an input circuit to Rigetti, Quartz first transpiles it to Nam and applies Toffoli decomposition and rotation merging. Next, Quartz rewrites each *CNOT* gate to a sequence of *H*, *CZ*, *H* gates, cancels out adjacent *H* or *CZ* pairs, and then fully converts the circuit to Rigetti by transforming *X* to $R_x(\pi)$ and *H* to $R_x(\pi) \cdot R_z(\frac{\pi}{2}) \cdot R_x(\frac{\pi}{2}) \cdot R_z(-\frac{\pi}{2})$. Ultimately, Quartz invokes the optimizer, using a suitable (n, q) -complete ECC set for Rigetti. Note that elimination of adjacent *H* or *CZ* pairs during the translation from Nam to Rigetti leads to more optimized circuits: a pair of adjacent *H* gates (that are canceled out) would otherwise be transformed into a sequence of eight

Table 4.1: Gate sets used in our evaluation. At the time of this research, IBM QX5 utilized the $\{U_1, U_2, U_3, CNOT\}$ gate set, and we refer to this gate set throughout this thesis. IBM has subsequently updated the native gate set supported by its newer quantum processors.

Name	Supported Gates
Nam [61, 97]	$H, X, R_z(\lambda), CNOT$
IBM [47]	$U_1(\theta), U_2(\phi, \lambda), U_3(\theta, \phi, \lambda), CNOT$
Rigetti [117]	$R_x(\frac{\pi}{2}), R_x(-\frac{\pi}{2}), R_x(\pi) = X, R_z(\lambda), CZ$

R_x and R_z gates that cannot be canceled out by Quartz since the cancellation is only correct for specific parameter values, while Quartz considers symbolic transformations valid for arbitrary parameter values.

4.3 Implementation

Quartz is publicly available as an open-source project [114] and also in the artifact supporting the paper [150]. In this section, we discuss the core implementation details of Quartz.

To realize the theoretical verification framework introduced in Section 3.4, we utilized the Z3 SMT solver [45] as our backend. During the development of Quartz, we occasionally encounter verification failures. These are consistently traced back to implementation bugs rather than theoretical flaws, and the concrete counterexamples generated by Z3 prove invaluable during the debugging process. Thus, the verification pipeline is useful not only for ensuring the ultimate mathematical correctness of the generated transformations, but also as a rigorous safeguard during software development.

4.3.1 Floating-Point Arithmetic

The REPGEN algorithm as presented in Section 3.3 uses real-valued fingerprints, where two equivalent circuits always have the same fingerprint. Our implementation of REPGEN uses floating-point arithmetic, which introduces some imprecision that can potentially lead to different fingerprints for equivalent circuits. To account for this imprecision, the implementation assumes there exists an *absolute error threshold* E_{max} , such that fingerprints of equivalent circuits differ by at most E_{max} when computed with floating-point arithmetic. The implementation therefore computes, using floating-point arithmetic, the integer $\left\lfloor \frac{\text{FINGERPRINT}(C)}{2E_{max}} \right\rfloor$, and uses it as the key for storing circuit C in $\mathcal{D}$. Under our assumption, equivalent circuits may still have different integer hash keys, but they may differ only by 1. Therefore, the implementation introduces an additional step after line 15 of Algorithm 1, in which ECCs that correspond to circuits with hash keys h and $h + 1$ are checked for equivalence and merged if found equivalent. In our experiments we set $E_{max} = 10^{-15}$ based on preliminary exploration of the floating-point errors that occur in practice. This is an empirical choice based on the machine epsilon.

4.3.2 Supported Gate Sets

Quartz is a generic quantum circuit optimizer supporting arbitrary gate sets, and it accepts a gate set $\mathcal{G}$ as part of its input. In our experiments, input circuits are given over the “Clifford + T” gate set: H , T , $T^\dagger$, S , $S^\dagger$, and $CNOT$; and output (optimized) circuits are in one of the three gate sets listed in Table 4.1: Nam, IBM, and Rigetti. Nam is a gate set commonly used in prior work [61, 97], IBM is derived from the IBM QX5 quantum processor [47], and Rigetti is derived from the Rigetti Agave quantum processor [117, 118].

4.3.3 Symbolic Parameter Expressions

As explained in Section 3.2.1, Quartz assumes a fixed number of parameters m and a specification Σ for parameter expressions used in circuits. Quartz takes m as input, and supports a flexible form for Σ defined by a finite set of parameter expressions and either allowing or disallowing parameters to be used more than once in a circuit.

Our experiments use $m = 2$ for the Nam and Rigetti gate sets, and $m = 4$ for the IBM gate set because it contains gates with up to three parameters. For Σ , we consider the expressions p_i , $2p_i$ and $p_i + p_j$ where $0 \leq i < m$ and $i < j < m$ (recall that $\vec{p} \in \mathbb{R}^m$ is the vector of parameters), and restrict each parameter to be used at most once in a circuit. This restriction significantly reduces the number of circuits REPGEN considers, especially for the IBM gate set because the U_3 gate requires three parameter expressions.

As explained in Section 3.4, the verifier searches for phase factors of the form $\vec{a} \cdot \vec{p} + b$. In our experiments we use $a \in \{-2, -1, 0, 1, 2\}^m$ and $b \in \{0, \frac{\pi}{4}, \frac{2\pi}{4}, \dots, \frac{7\pi}{4}\}$, which proves to be useful and sufficient in our preliminary experimentation with various gates. We later find that for the gate sets of Table 4.1, $\vec{a} = \vec{0}$ is actually sufficient. That is, these gate sets do not admit any transformations with parameter-dependent phase factors for the circuits we considered; they do however need various constant phase factors.

4.3.4 Alternative Verification Backends

For some large circuits, relying solely on quantifier-free nonlinear real arithmetic poses scalability challenges. In this section, we discuss alternative verification formulations and backends that can bypass the limitations of SMT solvers for larger quantum circuits.

In Section 3.4.2, we eliminate half-angle expressions by introducing $\theta' = \frac{\theta}{2}$ and utilizing the trigonometric functions $\cos \theta'$ and $\sin \theta'$. Alternatively, it is feasible to use $e^{i\theta'}$ directly as the primitive variable, representing each expression in the unitary matrix as a Laurent polynomial. Framing the verification problem through this lens opens up two distinct alternative approaches:

First, because these polynomial functions are analytic, equivalence can be proven by identifying a mathematical “cutoff” bound. If two symbolic circuits are evaluated and proven equivalent for a specific, finite number of concrete parameter instantiations, they are mathematically guaranteed to be equivalent for all possible continuous parameters [89, 120].

Second, after representing each expression in the unitary matrix as a Laurent polynomial, we can leverage symbolic algebra systems, such as SymPy [88], to simplify the expressions directly.

In many cases, SymPy can algebraically reduce the difference between two expressions to zero, proving equivalence without requiring an SMT solver.

In preliminary exploratory testing, we find that the symbolic approach with SymPy is neither strictly dominated by nor strictly dominating Z3. While Z3 is highly efficient and perfectly suited for verifying the relatively small circuits required to generate complete ECC sets (as utilized in our core pipeline), symbolic simplification shows promise for verifying large circuits that well exceed the scope of peephole optimization.

4.4 Evaluation

4.4.1 Experiment Setup

We compare Quartz with existing quantum circuit optimizers on a benchmark suite of 26 circuits developed by prior work [11, 97]. The benchmarks include arithmetic circuits (e.g., adding integers), multiple controlled X and Z gates (e.g., CCX and CCZ), the Galois field multiplier circuits, and quantum Fourier transformations. We use Quartz to optimize the benchmark circuits to the three gate sets of Table 4.1.

As in prior work [61, 97], we measure cost of a circuit in terms of the total gate count. We therefore define the `COST` function in Algorithm 2 as the number of gates in a circuit.²

Setting n and q for generating an (n, q) -complete ECC set determines the resulting transformations. Our experiments use: $n = 6, q = 3$ for the Nam gate set; $n = 4, q = 3$ for the IBM gate set; and $n = 3, q = 3$ for the Rigetti gate set, which provided good results for our benchmarks. Section 4.4.4 discusses the impact of n and q on Quartz’s performance.

Quartz’s backtracking search (Algorithm 2) is controlled by the hyper-parameter γ and the timeout threshold. Our experiments use $\gamma = 1.0001$, which yields good results for our benchmarks. This value for γ essentially means we consider cost-preserving transformations but not cost-increasing ones. For the search timeout, we use 24 hours. Section 4.4.4 discusses the timeout threshold and how it interacts with the settings for n and q . To stop the search from consuming too much memory, whenever the priority queue of Algorithm 2 contains more than 2,000 circuits we prune it and keep only the top 1,000 circuits. Our preliminary experimentation with this pruning suggested that it does not affect Quartz’s results.

We perform all experiments on an `m6i.32xlarge` AWS EC2 instance with a 128-core CPU and 512 GB RAM.

4.4.2 Circuit Optimization Results

Nam gate set. Table 4.2 compares Quartz to Qiskit [6], Nam [97], and VOQC [61] for the Nam gate set.³ (The performance of `t|ket` [130] for this gate set is similar to Qiskit, see [61].) The table also shows the gate count following Quartz’s preprocessing steps (rotation merging and Toffoli decomposition, see Section 4.2). For each benchmark circuit, the bold numbers highlight

²Quartz can in principle be used to optimize for other metrics, e.g. number of $CNOT$ or T gates, but here we focus on total gate count.

³The Nam optimizer generates an incorrect circuit for `qcla_mod.7` [75, Table 1].

Table 4.2: Gate count results for the Nam gate set. The best result for each circuit is in bold. “Quartz Preprocess” lists gate count after Quartz’s preprocessor (Section 4.2).

Circuit	Orig.	Qiskit	Nam	VOQC	Quartz Preprocess	Quartz End-to-end
adder_8	900	869	606	682	732	724
barenco_tof_3	58	56	40	50	46	38
barenco_tof_4	114	109	72	95	86	68
barenco_tof_5	170	162	104	140	126	98
barenco_tof_10	450	427	264	365	326	262
csla_mux_3	170	168	155	158	164	154
csum_mux_9	420	420	266	308	308	272
gf2 ⁴ _mult	225	213	187	192	186	177
gf2 ⁵ _mult	347	327	296	291	287	277
gf2 ⁶ _mult	495	465	403	410	401	391
gf2 ⁷ _mult	669	627	555	549	543	531
gf2 ⁸ _mult	883	819	712	705	703	703
gf2 ⁹ _mult	1095	1023	891	885	879	873
gf2 ¹⁰ _mult	1347	1257	1070	1084	1062	1060
mod5_4	63	62	51	56	51	26 [†]
mod_mult_55	119	117	91	90	105	93
mod_red_21	278	261	180	214	236	202
qcla_adder_10	521	512	399	438	450	422
qcla_com_7	443	428	284	314	349	292
qcla_mod_7	884	853	- ^{††}	723	727	719
rc_adder_6	200	195	140	157	174	154
tof_3	45	44	35	40	39	35
tof_4	75	73	55	65	63	55
tof_5	105	102	75	90	87	75
tof_10	255	247	175	215	207	175
vbe_adder_3	150	146	89	101	115	85
Geo. Mean Reduction	-	3.9%	27.3%	18.7%	18.6%	28.7%

[†] Computed as the median of seven runs: 25, 26, 26, 26, 32, 32, 32.

the number of gates achieved by the best-performing optimizer. Quartz outperforms Qiskit and VOQC on almost all circuits, indicating that it discovers most transformations used in these optimizers and also explores new optimization opportunities arising from new transformations and from the use of a cost-guided backtracking search (rather than a greedy approach, e.g., see Figure 4.1).

Quartz achieves on-par performance with Nam [97], a circuit optimizer highly tuned for this gate set. Nam applies a set of carefully chosen heuristics such as floating R_z gates and canceling one- and two-qubit gates (see [97] for more detail). While Quartz’s preprocessor implements two of Nam’s optimization passes, the results of the preprocessor alone are not close to Nam.⁴ By using the automatically generated transformations, Quartz is able to perform optimizations similar to some of Nam’s other hand-tuned optimizations, and even outperform Nam on roughly half of the circuits.

For `mod5_4`, we observe significant variability between runs, caused by randomness in ordering circuits with the same cost in the priority queue ($\mathcal{Q}$ in Algorithm 2). Therefore, Table 4.2 reports the median result from seven runs as well as individual results. This variability also suggests that Quartz’s performance can be improved by running the optimizer multiple times and taking the best discovered circuit, or by applying more advanced stochastic search techniques [78].

IBM gate set. Table 4.3 compares Quartz with Qiskit [6], $t|ket\rangle$ [130], and VOQC [61] on the IBM gate set. Qiskit and $t|ket\rangle$ include a number of optimizations specific to this gate set, such as merging any sequence of U_1 , U_2 , and U_3 gates into a single gate [112] and replacing any block of consecutive 1-qubit gates by a single U_3 gate [111]. Quartz is able to automatically discover some of these gate-specific optimizations by representing them each as a sequence of transformations. Overall, Quartz outperforms these existing compilers.

Rigetti gate set. Table 4.4 compares Quartz with Quilc [131] and $t|ket\rangle$ [130] on the Rigetti gate set. Quartz significantly outperforms $t|ket\rangle$ and Quilc on most circuits, even though Quilc is highly optimized for this gate set. We also note that while we employ some simplifications in the preprocessing phase for the Rigetti gate set (see Section 4.2), most of the reduction in gate count comes from the optimization phase.

4.4.3 Analyzing Quartz’s Generator and Verifier

We now examine Quartz’s circuit generator and circuit equivalence verifier. Table 4.5 shows the run times of the entire generation procedure, and also the amount of that time spent in verification, for each of the three gate sets and for varying values of n , while fixing $q = 3$. The table also lists the number of resulting circuit transformations $|\mathcal{T}|$, the size of the resulting representative set $|\mathcal{R}_n|$, and the characteristic (see Algorithm 1 and Theorem 2). For all gate sets, $|\mathcal{T}|$ and $|\mathcal{R}_n|$ grow exponentially with n . In spite of this exponential growth, the generator and verifier can

⁴We observe that for the `gf2^n_mult` circuits, Quartz’s preprocessor outperforms Nam. We attribute this difference to our greedy Toffoli decomposition, discussed in Section 4.2, which happens to work well for these circuits.

Table 4.3: Gate count results for the IBM gate set. The best result for each circuit is in bold. “Quartz Preprocess” lists gate count after Quartz’s preprocessor (Section 4.2).

Circuit	Orig.	Qiskit	$t \text{ket}\rangle$	VOQC	Quartz Preprocess	Quartz End-to-end
adder_8	900	805	775	643	736	583
barenco_tof_3	58	51	51	46	46	36
barenco_tof_4	114	100	100	89	86	67
barenco_tof_5	170	149	149	135	126	98
barenco_tof_10	450	394	394	347	326	253
csla_mux_3	170	153	155	148	164	139
csum_mux_9	420	382	361	308	364	340
gf2 ⁴ _mult	225	206	206	190	186	178
gf2 ⁵ _mult	347	318	319	289	287	275
gf2 ⁶ _mult	495	454	454	408	401	388
gf2 ⁷ _mult	669	614	614	547	543	530
gf2 ⁸ _mult	883	804	806	703	703	692
gf2 ⁹ _mult	1095	1006	1009	882	879	866
gf2 ¹⁰ _mult	1347	1238	1240	1080	1062	1050
mod5_4	63	58	58	53	55	51
mod_mult_55	119	106	102	83	109	91
mod_red_21	278	227	224	191	246	205
qcla_adder_10	521	460	460	409	450	372
qcla_com_7	443	392	392	292	349	267
qcla_mod_7	884	778	780	666	726	594
rc_adder_6	200	170	172	141	186	151
tof_3	45	40	40	36	39	31
tof_4	75	66	66	58	63	49
tof_5	105	92	92	80	87	67
tof_10	255	222	222	190	207	157
vbe_adder_3	150	133	139	100	115	82
Geo. Mean Reduction	-	11.0%	11.2%	23.1%	17.4%	30.1%

Table 4.4: Gate count results for the Rigetti gate set. The best result for each circuit is in bold. “Quartz Preprocess” lists gate count after Quartz’s preprocessor (Section 4.2).

Circuit	Orig.	Quilc	t ket)	Quartz Preprocess	Quartz End-to-end
adder_8	5324	3345	3726	4244	2553
barenco_tof_3	332	203	207	256	148
barenco_tof_4	656	390	408	500	272
barenco_tof_5	980	607	609	744	386
barenco_tof_10	2600	1552	1614	1964	960
csla_mux_3	1030	614	641	864	654
csum_mux_9	2296	1540	1542	1736	1100
gf2^4_mult	1315	809	827	1020	796
gf2^5_mult	2033	1301	1277	1573	1231
gf2^6_mult	2905	1797	1823	2235	1751
gf2^7_mult	3931	2427	2465	3021	2371
gf2^8_mult	5237	3208	3276	4033	3081
gf2^9_mult	6445	4070	4037	4933	3986
gf2^10_mult	7933	4977	4967	6048	5039
mod5_4	369	211	238	293	197
mod_mult_55	657	420	452	531	361
mod_red_21	1480	880	1020	1166	738
qcla_adder_10	3079	- [†]	1884	2464	1615
qcla_com_7	2512	1540	1606	1954	1095
qcla_mod_7	5130	3164	3202	4029	2525
rc_adder_6	1186	706	747	984	606
tof_3	255	150	160	201	135
tof_4	425	271	270	333	199
tof_5	595	354	380	465	271
tof_10	1445	878	930	1125	631
vbe_adder_3	900	534	557	705	366
Geo. Mean Reduction	-	38.6%	36.3%	21.9%	49.4%

[†] Quilc supports up to 32 qubits while qcla_adder_10 has 36.

Table 4.5: Metrics for Quartz’s generator, when generating (n, q) -complete ECC sets for $q=3$ and varying values of n for the three gate sets. $|\mathcal{T}|$ is the resulting number of transformations, $|\mathcal{R}_n|$ is the size of the resulting representative set, and ch is the characteristic (see Algorithm 1 and Theorem 2).

	n	$ \mathcal{T} $	$ \mathcal{R}_n $	Verification Time (s)	Total Time (s)
Nam ch = 27	2	62	397	1.2	1.3
	3	196	4,179	2.6	3.7
	4	1,304	36,177	8.5	21.4
	5	8,002	269,846	49.5	174.7
	6	56,152	1,777,219	370.3	1,400.4
	7	379,864	10,432,127	2,673.6	10,461.2
IBM ch = 1, 362	2	1,912	22,918	22.9	38.6
	3	5,086	224,281	100.4	225.9
	4	16,748	1,552,185	356.9	1,290.0
	5	225,068	7,847,203	1,844.8	8,363.1
Rigetti ch = 30	2	66	361	1.3	1.5
	3	66	3,143	2.6	3.7
	4	224	22,043	5.8	15.4
	5	2,396	134,423	22.7	100.2
	6	15,464	729,842	132.0	675.3

generate, in a reasonable run time of a few hours, an (n, q) -complete ECC set for values of n and q that are sufficiently large to be useful for circuit optimization. The growth in the number of transformations significantly affects the optimizer. For Nam and IBM, our selected values of $n = 6$ and $n = 4$ result in a similar order of magnitude for $|\mathcal{T}|$. For Rigetti, we use $n = 2$, resulting in much smaller $\mathcal{T}$. This choice is related to the fact that circuits in the Rigetti gate set are larger by roughly an order of magnitude compared to Nam and IBM (compare “Orig.” in Table 4.4 with Tables 4.2 and 4.3; see discussion in Section 4.4.4).

Table 4.7 shows the run times of generation and verification for the Nam gate set, also for different numbers of qubits. $q = 3$ achieves the best optimization performance according to Table 4.8.

REPGEN algorithm and pruning techniques. We now evaluate the effectiveness of REPGEN and the pruning techniques described in Section 3.5 for reducing the number of circuits Quartz must consider (which is closely correlated with the number of resulting transformations). To evaluate the relative contribution of each technique, Table 4.6 reports the number of circuits considered when applying: (i) REPGEN without additional pruning, (ii) REPGEN combined with ECC simplification, and (iii) REPGEN combined with both ECC simplification and common subcircuit pruning; and compares each of these to a brute force approach of generating all possible circuits with up to q qubits and n gates. Both REPGEN and the pruning techniques play an important role in eliminating redundant circuits while preserving (n, q) -completeness. Ulti-

Table 4.6: Number of circuits considered when using REPGEN with or without the pruning techniques of Section 3.5 to generate (n, q) -complete ECC sets for $q=3$ and varying values of n for the three gate sets. Circuits are counted by their sequence representation, as REPGEN considers multiple sequences for each actual circuit (Section 3.3). Parentheses show reduction relative to the number of all possible circuits for n and q . “REPGEN” corresponds to REPGEN without additional pruning. “+ ECC Simplification” corresponds to REPGEN combined with ECC simplification. “+ Common Subcircuit” corresponds to REPGEN combined with all pruning techniques and ultimately represents Quartz’s generator.

n	Possible Circuits	REPGEN	+ ECC Simplification	+ Common Subcircuit
Nam	2	604	400 (2×)	50 (12×)
	3	11,404	1,180 (10×)	231 (49×)
	4	198,028	5,178 (38×)	2,170 (91×)
	5	3,246,220	31,517 (103×)	18,244 (178×)
	6	51,021,964	195,466 (261×)	131,554 (388×)
	7	776,616,076	1,196,163 (649×)	875,080 (887×)
IBM	2	35,005	23,413 (1×)	1,708 (20×)
	3	533,857	62,594 (9×)	10,287 (52×)
	4	6,446,209	185,315 (35×)	65,343 (99×)
	5	68,078,785	921,611 (74×)	512,975 (133×)
Rigetti	2	778	469 (2×)	51 [†] (15×)
	3	17,518	965 (18×)	117 (150×)
	4	367,843	2,293 (160×)	548 (671×)
	5	7,354,093	10,568 (696×)	4,949 (1,486×)
	6	141,763,468	58,193 (2,436×)	35,690 (3,972×)
	7			15,240 (9,302×)

[†] For Rigetti, $n = 2$ and $n = 3$ result in identical transformations—each 3-gate transformation is subsumed by 2-gate transformations in a way identified by Quartz.

Table 4.7: Metrics for Quartz’s generator, when generating (n, q) -complete ECC sets for $2 \leq n \leq 7$ and $1 \leq q \leq 4$ for the Nam gate set. $|\mathcal{T}|$ is the resulting number of transformations. The characteristics (see definition in Section 3.3.3) for $q = 1, 2, 3, 4$ are 7, 16, 27, 40, respectively.

n	$ \mathcal{T} $				Verification Time (s)				Total Time (s)			
	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q = 1$	$q = 2$	$q = 3$	$q = 4$
2	14	38	62	78	0.5	0.7	1.2	2.5	0.5	0.7	1.3	2.8
3	14	90	196	208	0.8	1.3	2.6	7.8	0.8	1.5	3.7	12.0
4	44	332	1,304	2,988	1.1	2.3	8.5	48.0	1.2	3.8	21.4	118.3
5	78	1,334	8,002	27,942	1.5	8.8	49.5	917.0	1.6	19.2	174.7	2,452.0
6	120	5,794	56,152	273,532	1.9	52.5	370.3	5,802.6	2.2	138.9	1,400.4	19,448.0
7	164	21,824	379,864	— [†]	2.3	71.8	2,673.6	— [†]	2.8	222.9	10,461.2	— [†]

[†] We were unable to generate a $(7, 4)$ -complete ECC set using 512 GB of RAM.

mately, REPGEN and the pruning techniques reduce the number of transformations the optimizer must consider by one to three orders of magnitude.

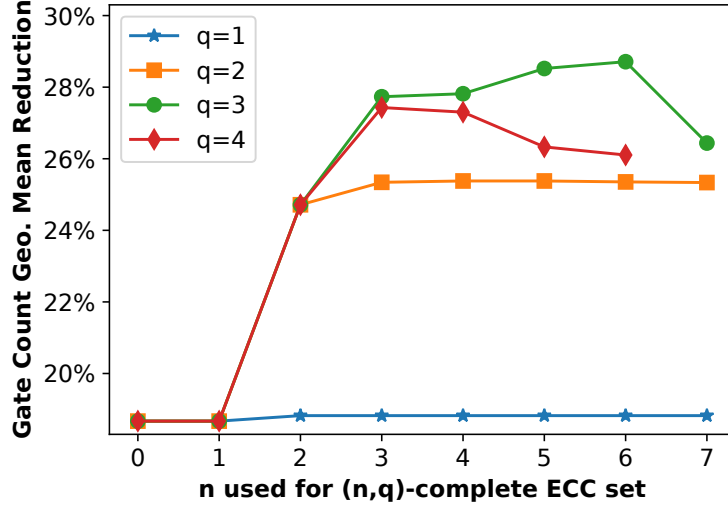


Figure 4.2: Optimization effectiveness with (n, q) -complete ECC sets for varying n and q after a 24-hour search timeout. For $n = 0$ there are no transformations and the results match the “Quartz Preprocess” column of Table 4.2.

4.4.4 Scalability

We now examine Quartz’s circuit optimizer when using an (n, q) -complete ECC set for varying values of n and q . For this study we focus on the Nam gate set, and compare different values for n and q by the *optimization effectiveness* they yield, defined as the reduction in geometric mean gate count over all circuits (as in the bottom line of Table 4.2). For `mod5_4`, when $q = 3$ and $3 \leq n \leq 7$, we use the median of 7 runs due to the variability discussed in Section 4.4.2.

As we increase n and q we expect Quartz’s optimizer to: (i) be able to reach more optimized circuits, and (ii) require more time per search iteration. Both of these follow from the fact that increasing n and q yields more transformations. Under a fixed search time budget, we expect the increased cost of search iterations to reduce the positive impact of larger n ’s and q ’s. Because each iteration (Algorithm 2) considers a candidate circuit C and computes $\text{APPLY}(C, T)$ for each transformation $T \in \mathcal{T}$, the cost per iteration scales linearly with the number of transformations $|\mathcal{T}|$. Since $|\mathcal{T}|$ varies dramatically as n and q change,⁵ we expect the second effect (slowing down the search) to be significant, especially for large circuits which typically require more search iterations (and additionally increase APPLY ’s running time).

Figure 4.2 shows optimization effectiveness (reduction in geometric mean gate count) for varying values of n and q , under a search timeout of 24 hours. The figure supports the tradeoff discussed above. Using too small values for n and q results in low effectiveness, and as we increase n or q effectiveness increases but then starts decreasing, as the negative impact of the large number of transformations starts outweighing their benefit. (See Table 4.7 for $|\mathcal{T}|$ in each configuration.) As expected, the optimal setting for n and q generally varies across circuits—smaller circuits tend to be better optimized with larger values of n (Table 4.8). Still, Figure 4.2

⁵For example: with $q = 3$, $|\mathcal{T}| = 196$ for $n = 3$ and $|\mathcal{T}| = 56,152$ for $n = 6$; with $q = 4$, $|\mathcal{T}| = 208$ for $n = 3$ and $|\mathcal{T}| = 273,532$ for $n = 6$ (see Table 4.7). We are unable to generate a $(7, 4)$ -complete ECC set using 512 GB of RAM.

shows that there are several settings that yield good overall results: $3 \leq n \leq 6$ for $q = 3$, and $3 \leq n \leq 4$ for $q = 4$. Interestingly, $q = 3; 3 \leq n \leq 6$ cover the best optimization results for all circuits obtained among all configurations considered in Figure 4.2 (Table 4.8).

Table 4.8: Gate count results for the Nam gate set when using (n, q) -complete ECC sets with varying values of n and q , and a search timeout of 24 hours. “Pr.” shows the result when $n = 0$ or $n = 1$: the ECC set is empty, so the results match the “Quartz Preprocess” column of Table 4.2. When $q = 1$, the results are the same when $2 \leq n \leq 7$.

Circuit	Orig.	Pr.	$q = 1$	$q = 2, n = -$						$q = 3, n = -$						$q = 4, n = -$					
				2	3	4	5	6	7	2	3	4	5	6	7	2	3	4	5	6	
tof_3	45	39	39	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35		
barenco_tof_3	58	46	46	42	42	42	42	42	42	42	40	40	40	38	38	42	40	40	40	38	
mod5_4	63	51	51	51	51	51	51	51	51	51	45	41	27	26	31	51	45	37	32	26	
tof_4	75	63	63	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	
tof_5	105	87	87	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75	
barenco_tof_4	114	86	86	78	78	78	78	78	78	78	72	72	72	68	68	78	72	72	72	68	
mod_mult_55	119	105	105	97	94	94	94	94	94	97	92	93	93	93	94	97	92	93	93	94	
vbe_adder_3	150	115	115	95	95	95	95	95	95	95	91	91	89	85	85	95	91	90	91	91	
barenco_tof_5	170	126	126	114	114	114	114	114	114	114	104	104	104	98	100	114	104	104	104	102	
cs1a_mux_3	170	164	164	160	152	152	152	152	152	160	146	148	149	154	156	160	150	148	153	153	
rc_adder_6	200	174	174	154	152	152	152	152	152	154	152	152	152	154	154	154	152	152	154	156	
gf2^4_mult	225	186	186	186	180	180	180	180	180	186	176	175	177	177	178	186	175	177	178	180	
tof_10	255	207	207	175	175	175	175	175	175	175	175	175	175	175	175	175	175	175	175	191	
mod_red_21	278	236	226	202	202	202	202	202	202	202	202	202	202	202	210	202	202	202	202	228	
gf2^5_mult	347	287	287	287	279	279	279	279	279	287	273	273	273	277	279	287	277	277	279	283	
csum_mux_9	420	308	308	308	308	308	308	308	308	308	280	280	280	272	302	308	280	280	305	307	
qcla_com_7	443	349	347	293	293	293	293	293	293	293	289	288	288	292	339	293	289	288	321	347	
barenco_tof_10	450	326	326	294	294	294	294	294	294	294	268	271	271	262	316	294	268	275	298	324	
gf2^6_mult	495	401	401	401	391	391	391	391	391	401	381	383	386	391	393	401	386	391	391	401	
qcla_adder_10	521	450	450	416	414	414	414	414	414	416	407	408	408	422	444	416	408	414	436	450	
gf2^7_mult	669	543	543	543	531	531	531	531	531	543	517	519	529	531	539	543	524	530	537	543	
gf2^8_mult	883	703	703	703	703	703	703	703	703	703	690	703	703	703	703	703	703	703	703	703	
qcla_mod_7	884	727	727	657	657	657	657	657	657	657	654	651	677	719	727	657	655	697	725	727	
adder_8	900	732	732	644	640	640	640	640	644	644	638	634	688	724	732	644	634	706	730	732	
gf2^9_mult	1095	879	879	879	877	869	869	877	877	879	857	856	870	873	879	879	857	871	879	879	
gf2^10_mult	1347	1062	1062	1062	1062	1058	1058	1058	1058	1062	1030	1049	1052	1060	1062	1062	1061	1058	1062	1062	

Figure 4.3 shows how the search time impacts optimization for different choices for n (focusing on $q = 3$). For each value of n , we observe a quick initial burst, followed by a gentle increase. At the end of the initial burst, effectiveness monotonically decreases as n increases, for all $3 \leq n \leq 6$. As time progresses the gaps diminish and eventually the order is reversed: at around 21 hours $n = 6$ surpasses $n = 3$. The settings $n = 2$ and $n = 7$ yield poor effectiveness: $n = 2$ does not contain an adequate number of transformations and quickly saturates the search time, while $n = 7$ contains too many transformations and progresses too slowly.

Figure 4.3 also shows the effectiveness of a hypothetical run constructed by taking the best setting for each circuit at each time. This “best” curve considerably outperforms the others, because the best setting for n varies across circuits with different sizes.

Figures 4.4 to 4.29 show plots akin to Figure 4.2 and Figure 4.3 for each circuit. In each figure, the left plot shows the optimization result with (n, q) -complete ECC sets for varying n and q after a 24-hour search timeout. The right plot shows the optimization result over time for $q = 3$ and $2 \leq n \leq 7$. Each 24-hour result on the right plot corresponds to a green round marker ($q = 3$) on the left plot.

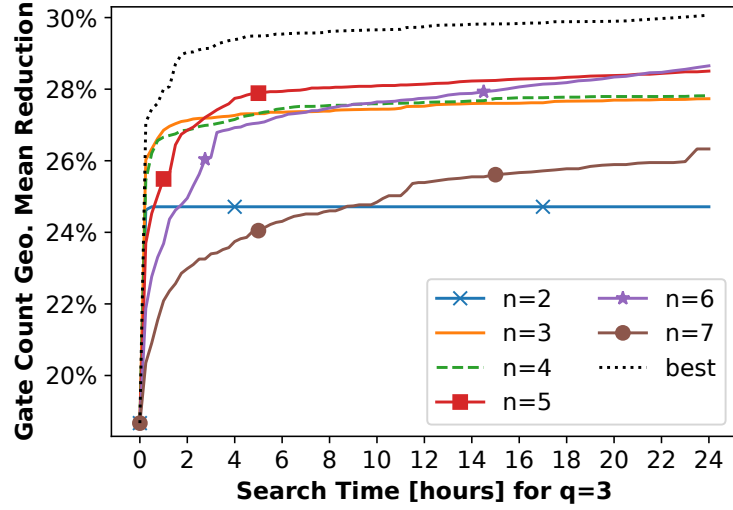


Figure 4.3: Optimization effectiveness over time ($q=3$; $2 \leq n \leq 7$). For each time point, “best” computes the reduction in geometric mean gate count obtained by selecting the most effective value for n at that time point for each circuit (i.e., different circuits may use different n ’s, and the same circuit may use different n ’s at different time points).

Among these figures, Figure 4.18 shows the median run for `mod5_4`. The median run is defined to be the run with the final gate count being the median of the 7 runs. We present the results of all 7 runs for `mod5_4` for $q = 3$ and $3 \leq n \leq 7$ in Figure 4.30.

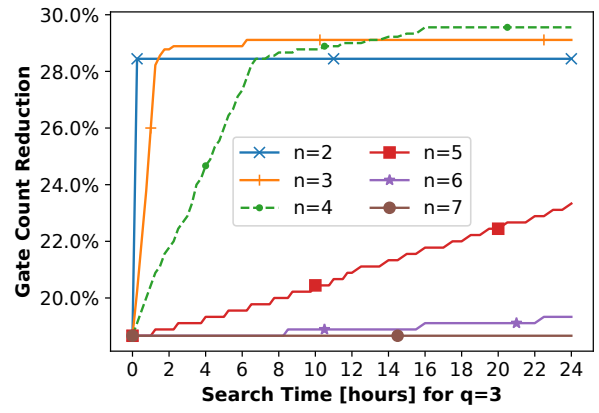
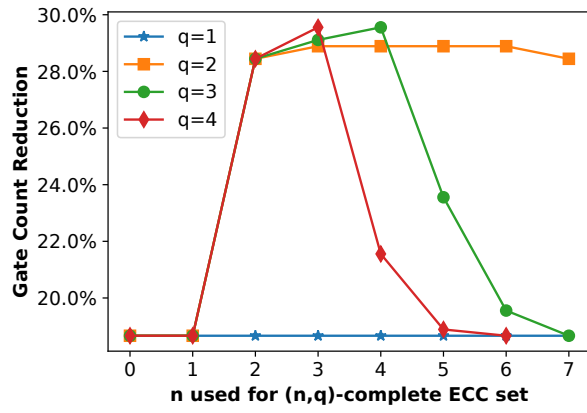


Figure 4.4: adder_8 (900 gates).

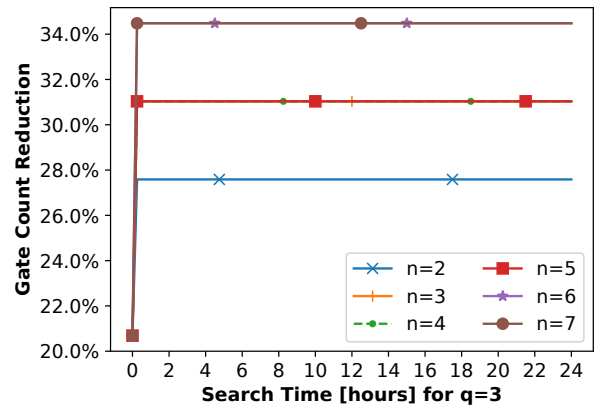
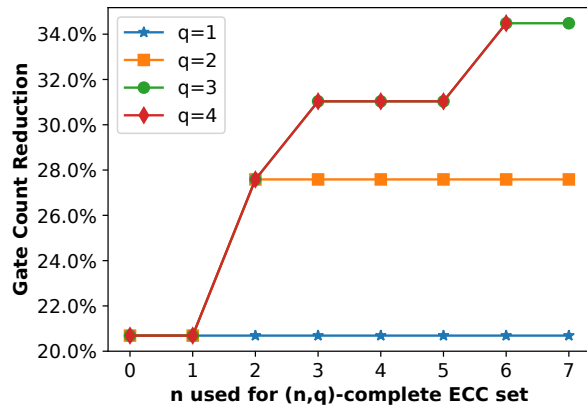


Figure 4.5: barenco_to_f3 (58 gates).

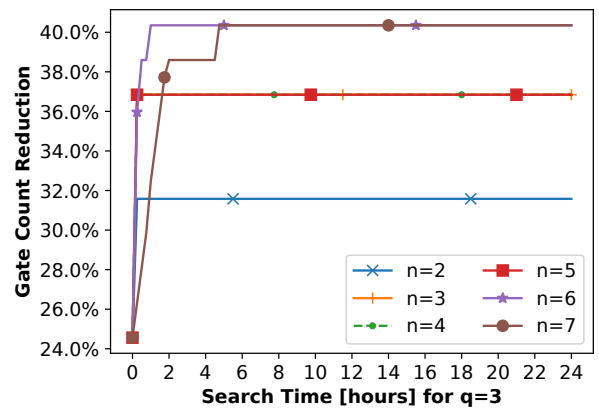
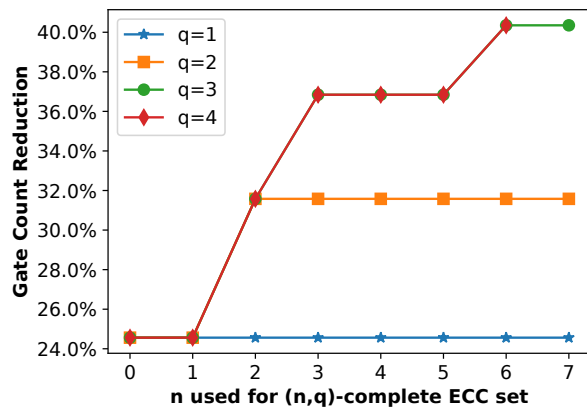


Figure 4.6: barenco_to_f4 (114 gates).

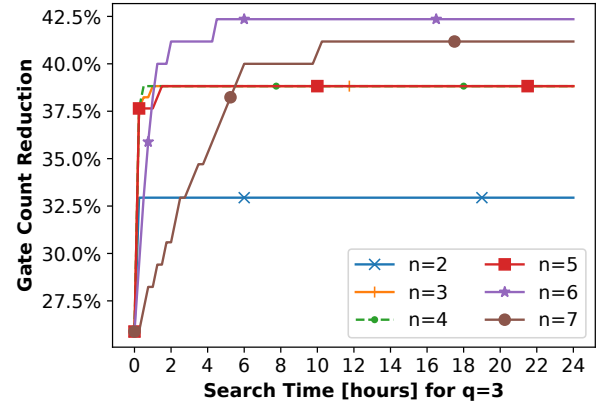
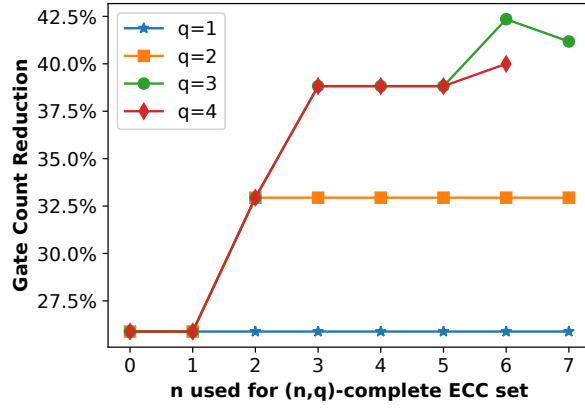


Figure 4.7: barenco_tof_5 (170 gates).

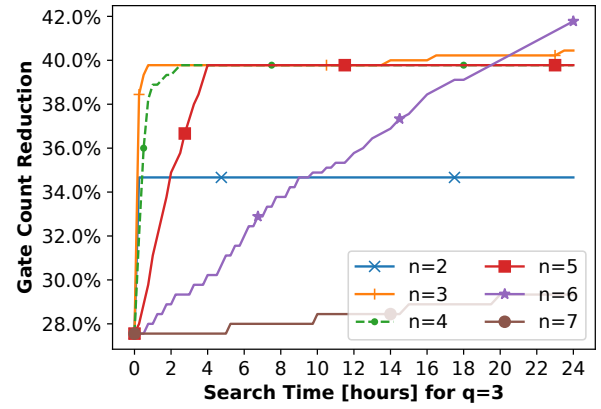
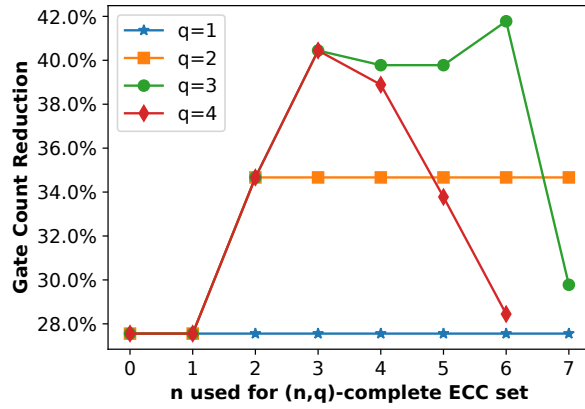


Figure 4.8: barenco_tof_10 (450 gates).

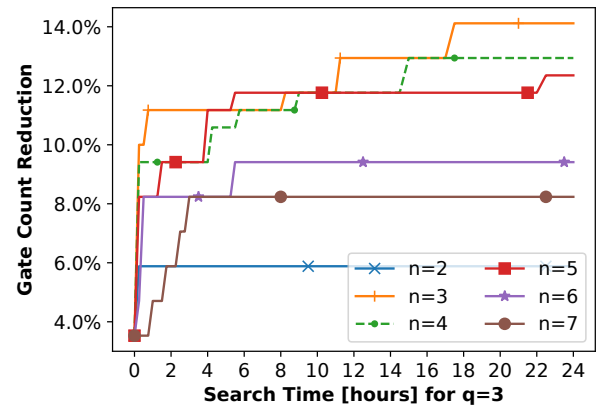
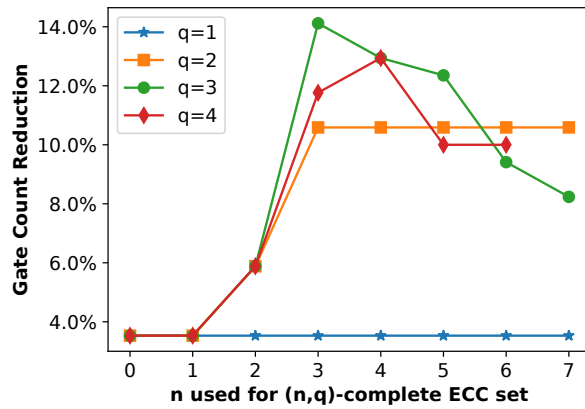


Figure 4.9: cs1a_mux_3 (170 gates).

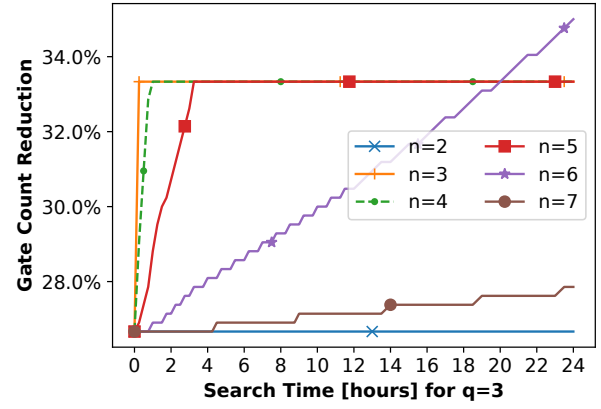
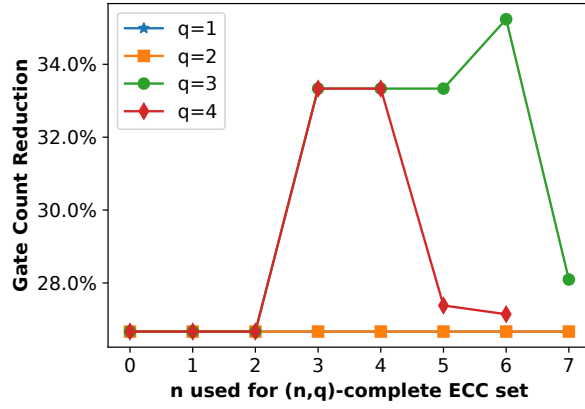


Figure 4.10: csum_mux_9 (420 gates).

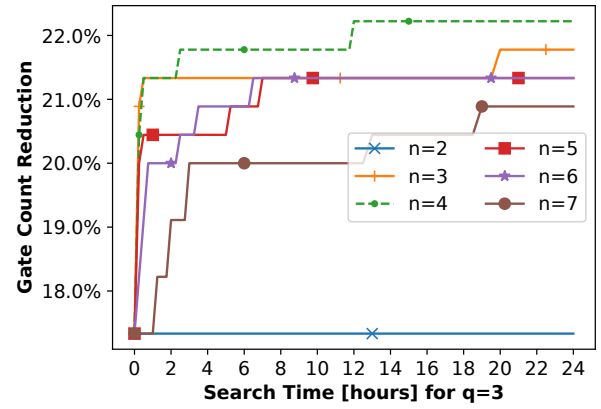
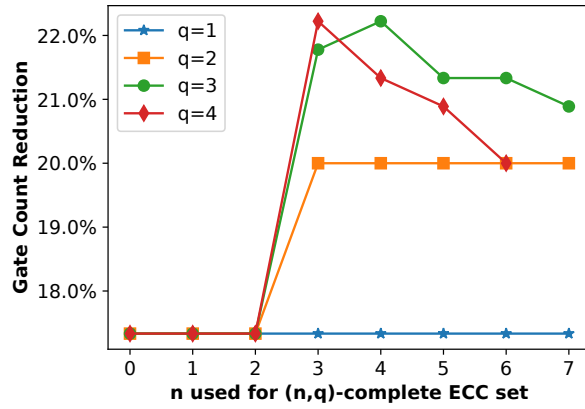


Figure 4.11: gf2⁴_mult (225 gates).

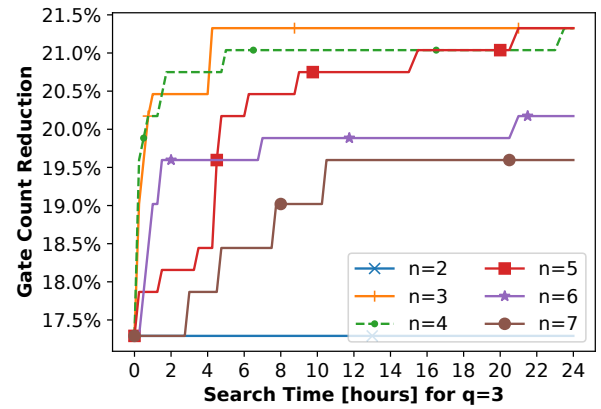
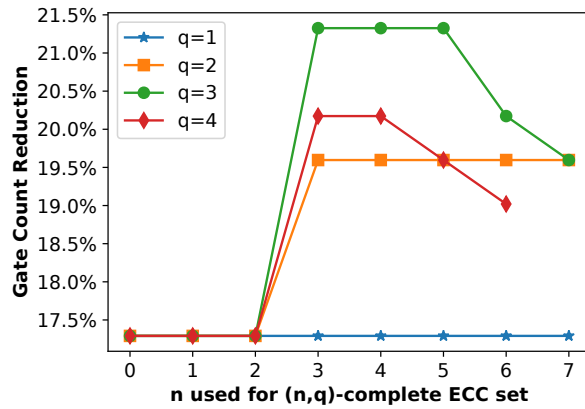


Figure 4.12: gf2⁵_mult (347 gates).

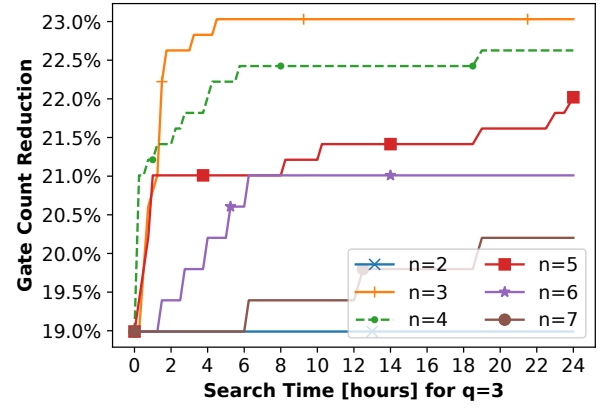
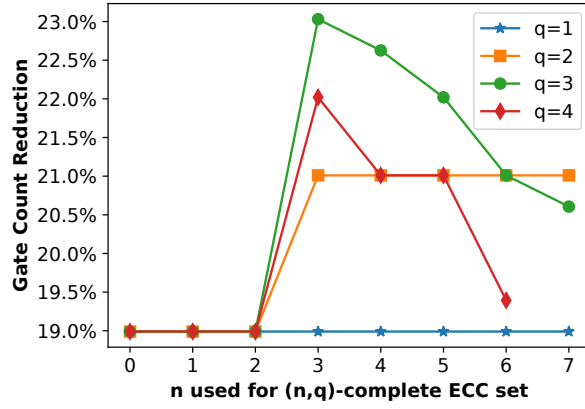


Figure 4.13: $gf2^6_mult$ (495 gates).

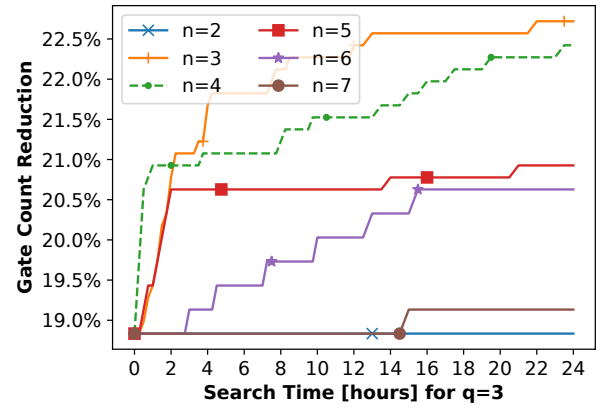
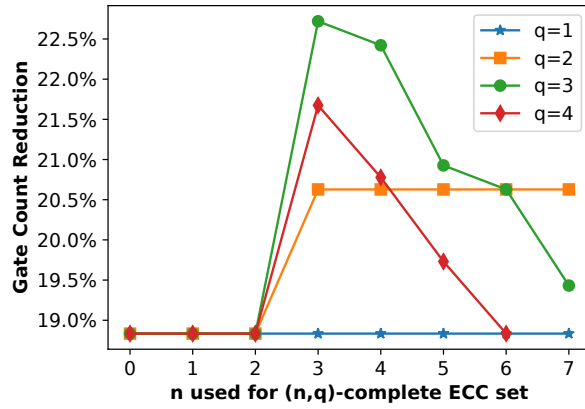


Figure 4.14: $gf2^7_mult$ (669 gates).

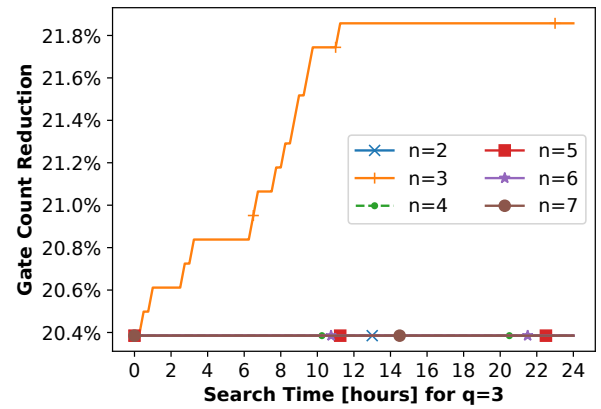
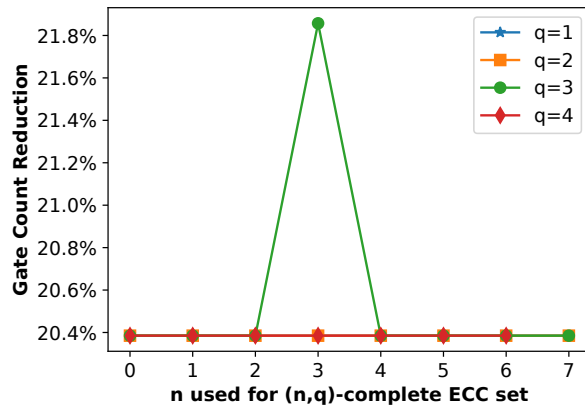


Figure 4.15: $gf2^8_mult$ (883 gates).

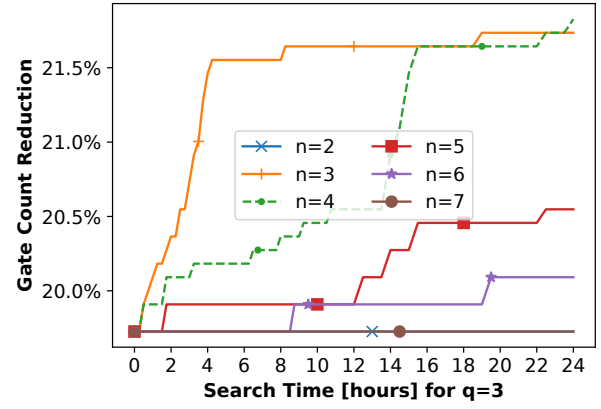
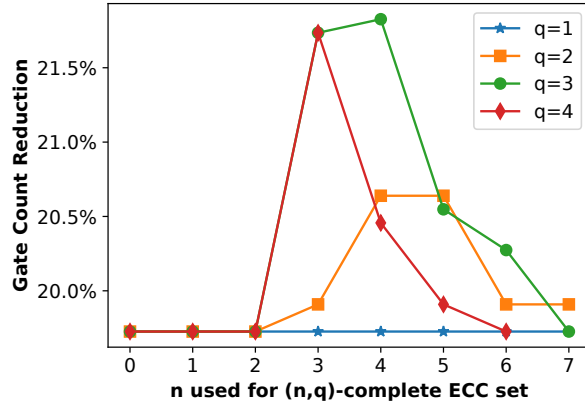


Figure 4.16: $gf2^9_mult$ (1095 gates).

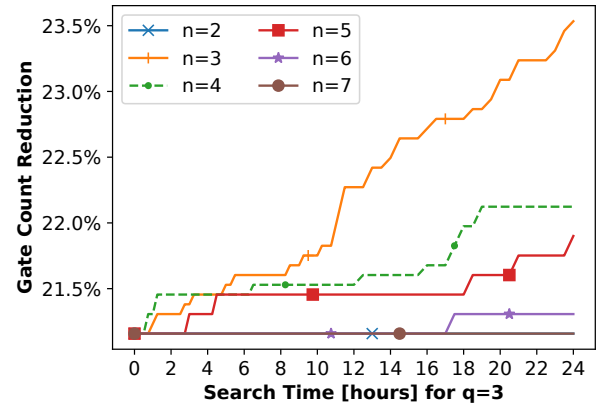
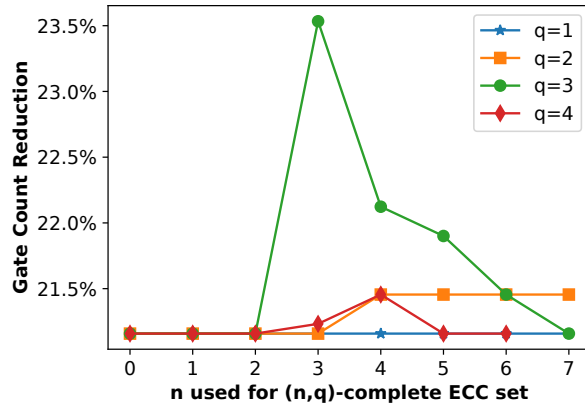


Figure 4.17: $gf2^{10}_mult$ (1347 gates).

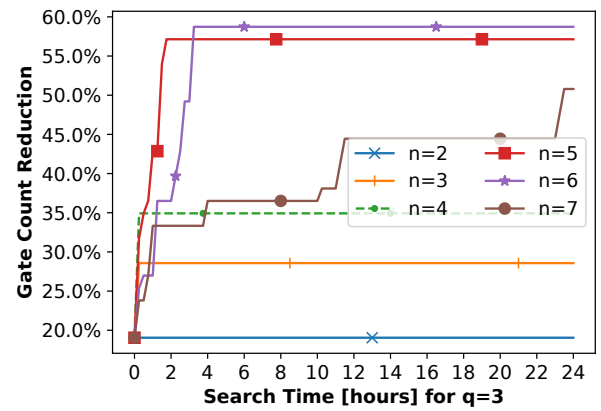
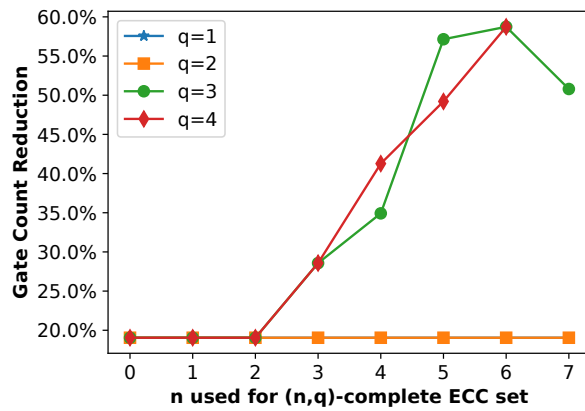


Figure 4.18: $mod5_4$ (63 gates).

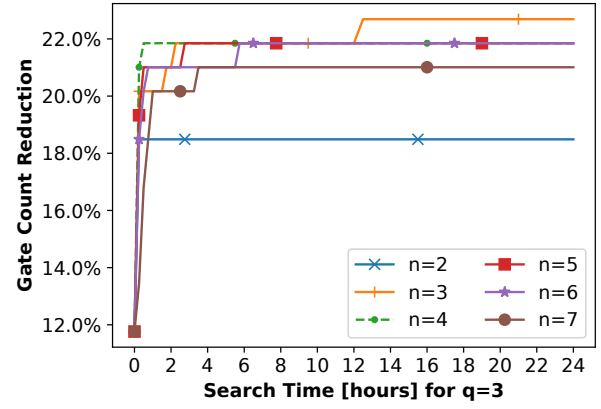
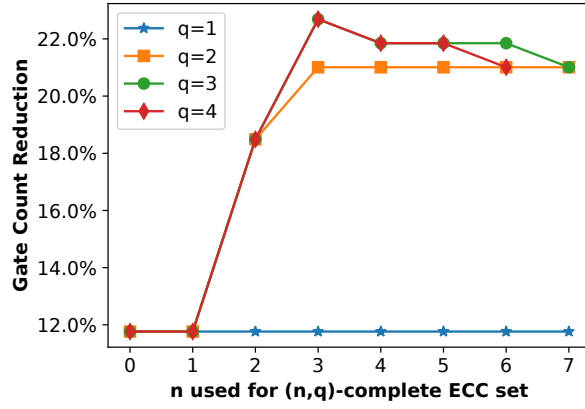


Figure 4.19: mod_mult_55 (119 gates).

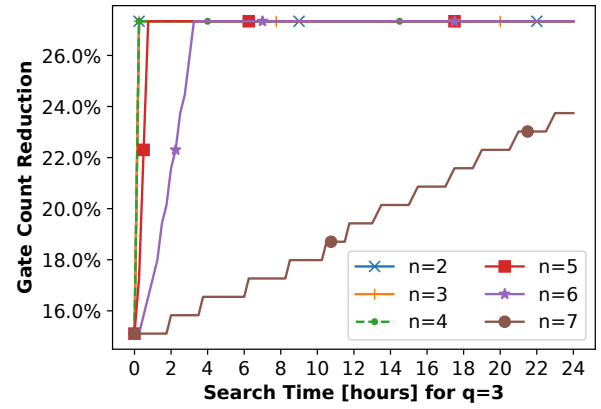
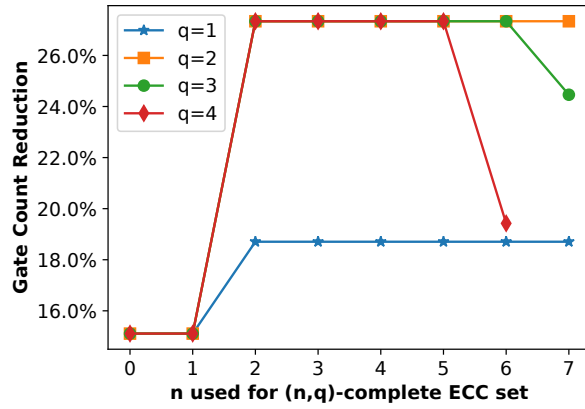


Figure 4.20: mod_red_21 (278 gates).

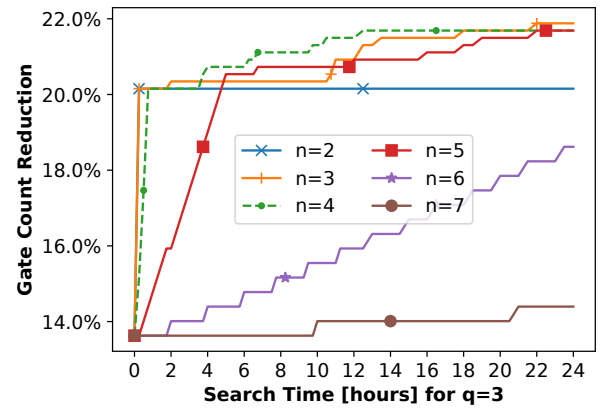
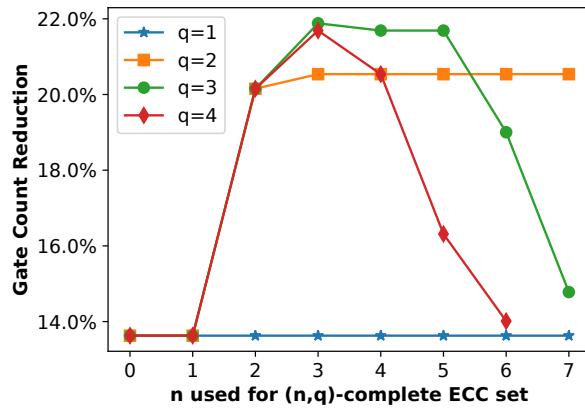


Figure 4.21: qcla_adder_10 (521 gates).

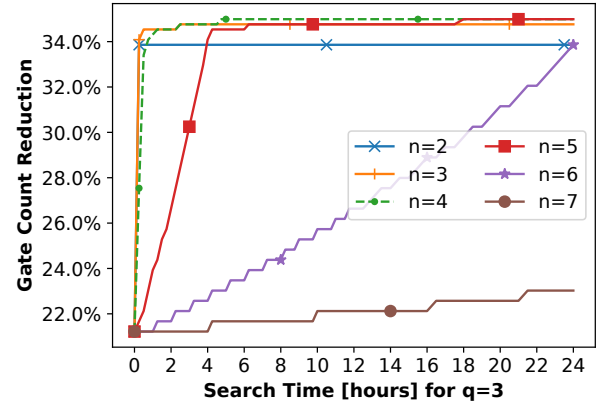
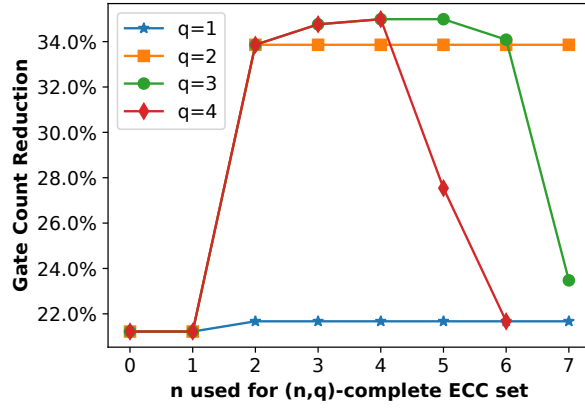


Figure 4.22: qcla_com_7 (443 gates).

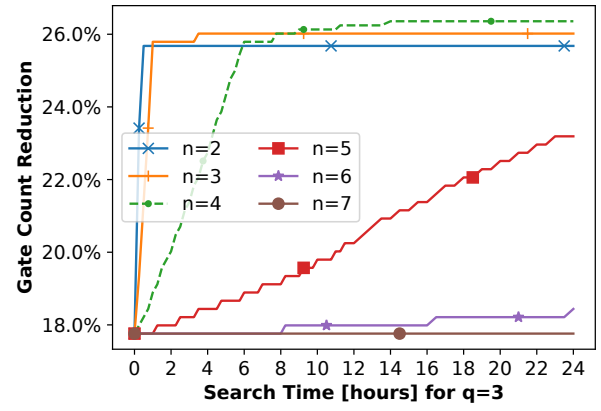
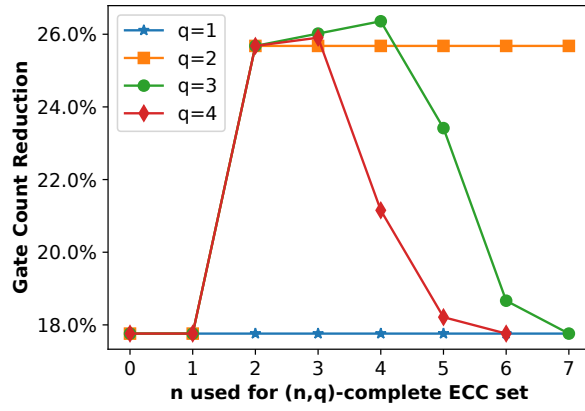


Figure 4.23: qcla_mod_7 (884 gates).

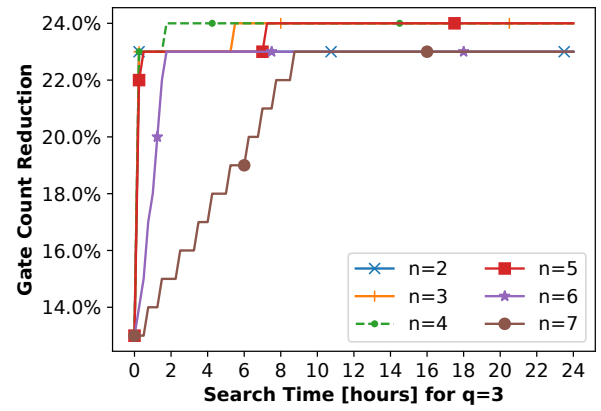
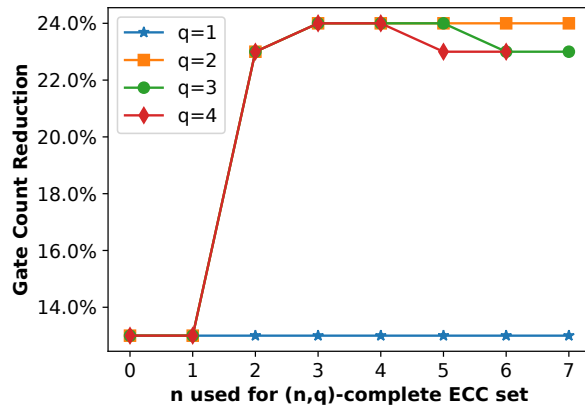


Figure 4.24: rc_adder_6 (200 gates).

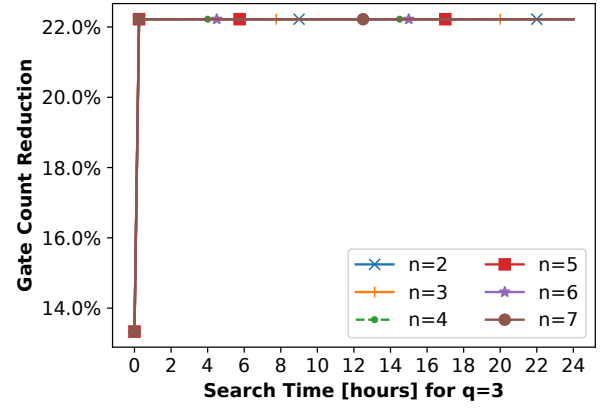
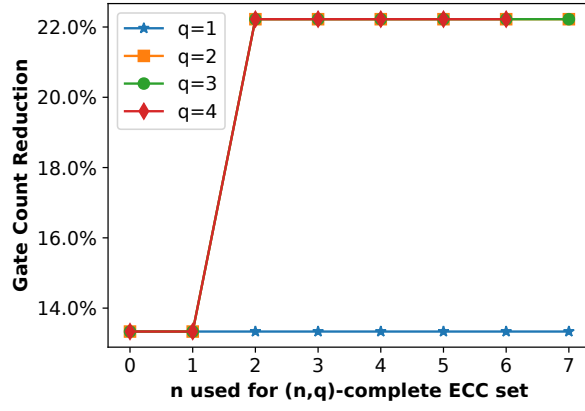


Figure 4.25: tof_3 (45 gates).

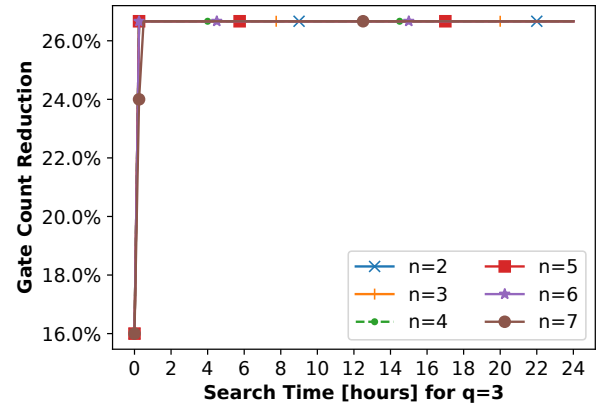
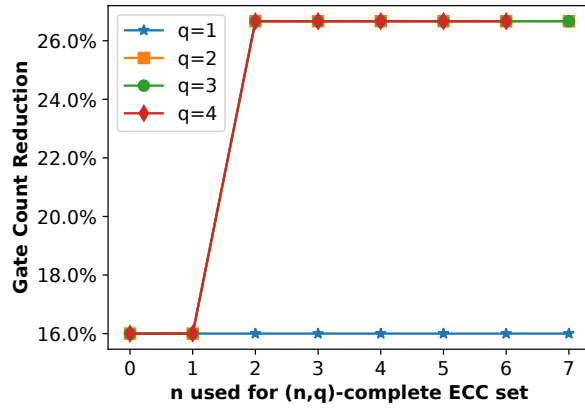


Figure 4.26: tof_4 (75 gates).

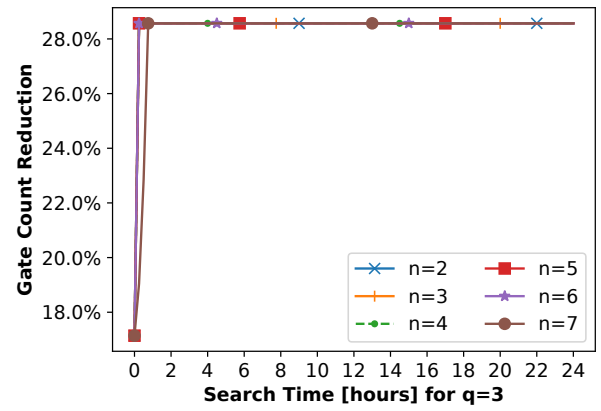
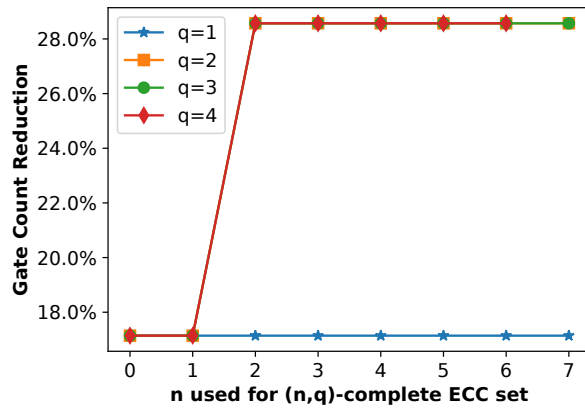


Figure 4.27: tof_5 (105 gates).

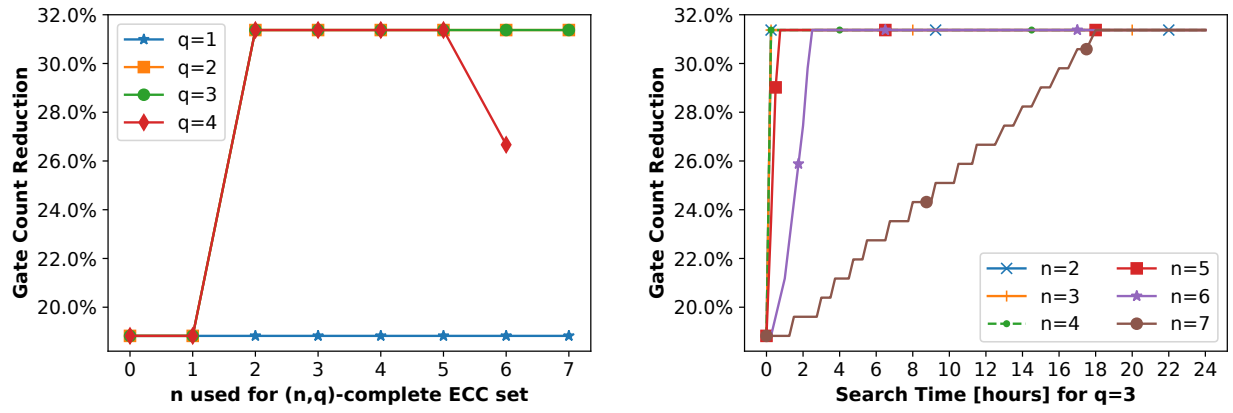


Figure 4.28: tof_10 (255 gates).

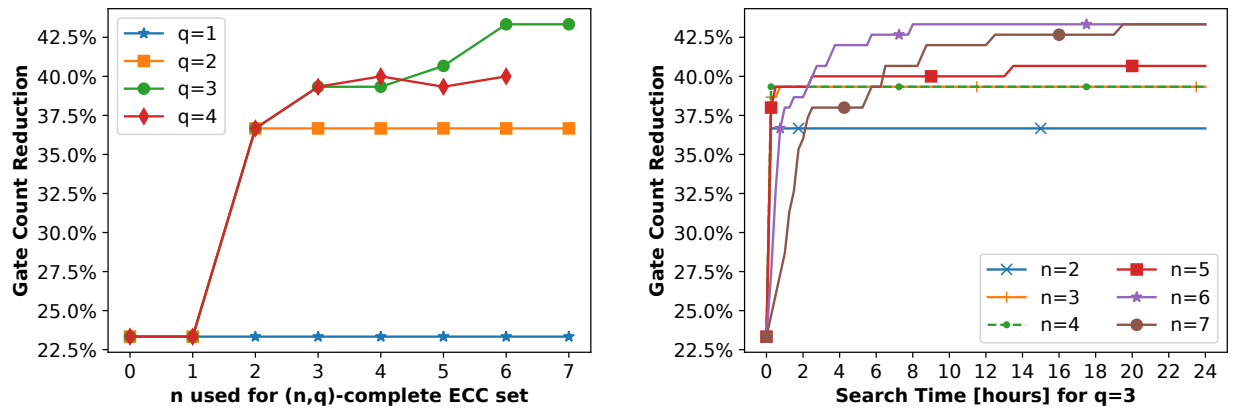


Figure 4.29: vbe_adder_3 (150 gates).

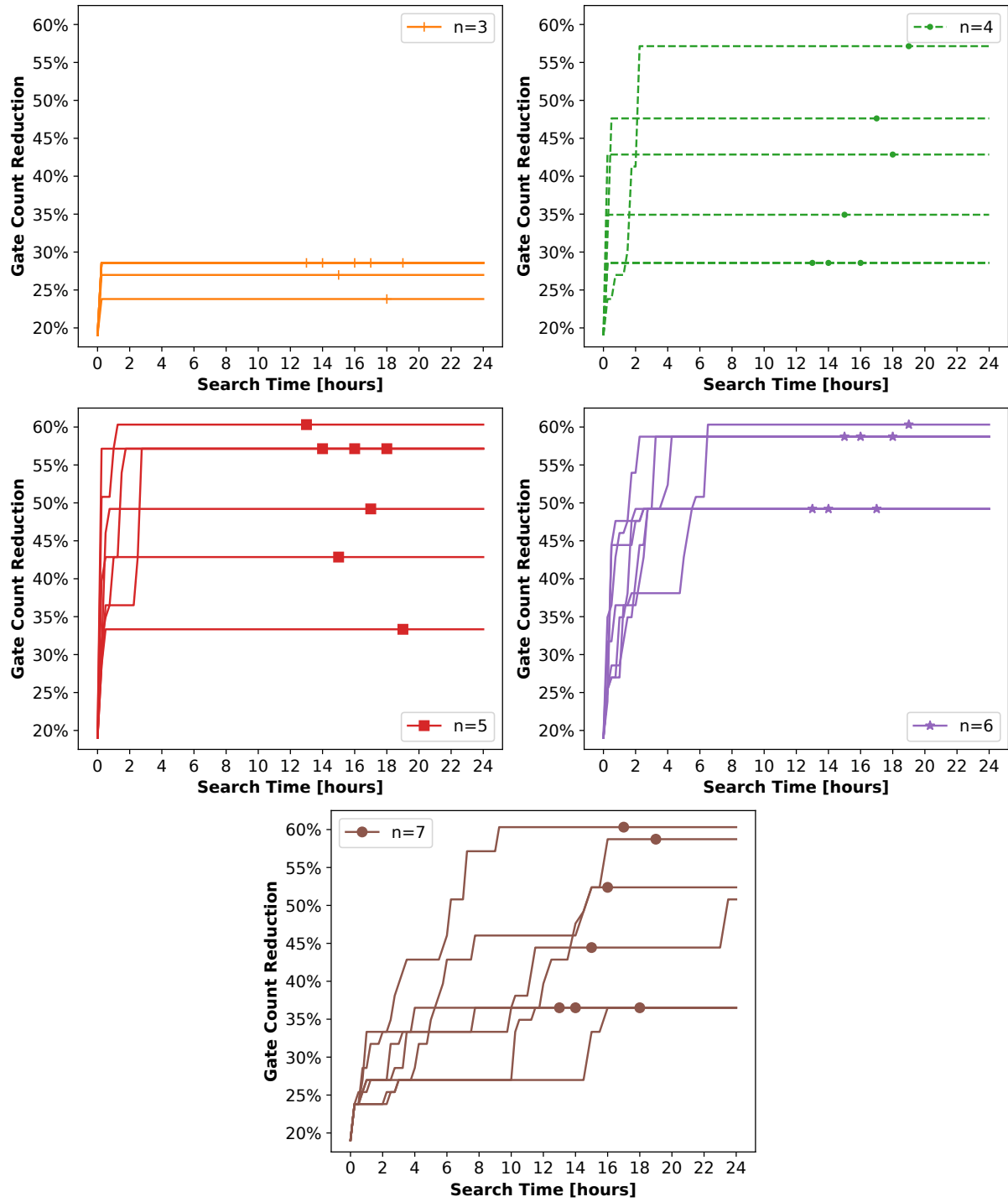


Figure 4.30: 7 runs of `mod5_4` with a $(n, 3)$ -complete ECC set for each $3 \leq n \leq 7$. Each marker denotes one run. For example, the three markers on the 57.1%-reduction line of $n = 5$ show three runs resulting in 27 gates after 24 hours.

Chapter 5

QALM: Interleaved Exploration and Exploitation

As demonstrated by Quartz in the previous chapter, casting circuit optimization as an explicit search over an enormous space of equivalent circuits yields highly optimized results. However, existing circuit optimizers face a fundamental limitation in how they tolerate temporary cost increases to escape local minima (Figure 5.1). At one extreme, greedy rule-based optimizers immediately apply any cost-reducing transformation, achieving high efficiency but quickly becoming trapped. At the other extreme, search-based optimizers accept cost-increasing moves to explore the circuit space. But because search-based optimizers cannot cheaply determine whether a given point is promising, that is, whether its neighborhood contains a deeper local minimum, they must blindly explore higher-cost regions, taking exponentially many steps to escape the current basin.

This chapter shows that this limitation can be overcome: good optimization quality and efficient runtime are not inherently in tension. Rather than simply balancing exploration and exploitation, we introduce QALM, a hybrid framework that *interleaves* search-based and rule-based optimization in a unified control loop. As illustrated in Figure 5.1, exploit phases descend to a local minimum as quickly as a rule-based optimizer, while interleaved explore phases uncover candidate promising points via search; the following exploit phase then tests each candidate by descending from it, keeping the one that reaches the deepest local minimum. This design identifies the truly promising point within a single exploit phase, sidestepping the exponential wait incurred by pure search.

Our evaluation across 248 circuits demonstrates that QALM does not merely strike a balance; it outperforms existing rule-based and search-based optimizers in circuit reduction rates while operating with the computational efficiency of rule-based systems. Given the same time budget (one hour), QALM achieves 5.97% more gate count reduction than the strongest baseline, and even when given only one minute, it still achieves 4.98% more reduction than the strongest baseline. Furthermore, it matches or exceeds the fidelity of the strongest baseline on 83.9% of the evaluated circuits.

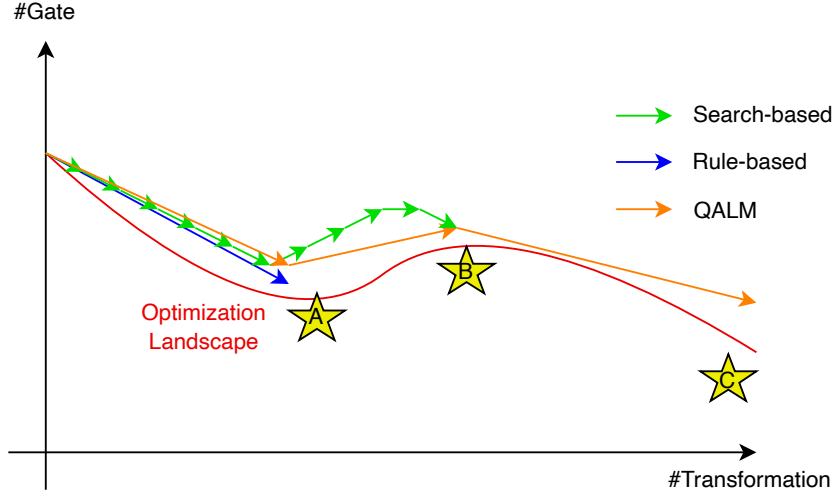


Figure 5.1: Comparison of optimization trajectories across three strategies: rule-based (pure exploitation), search-based (pure exploration), and QALM (interleaved exploration and exploitation). Rule-based optimization greedily descends to a local minimum in a single step and terminates (shown as *A*). Search-based optimization reaches the same local minimum more slowly, then continues to explore and discovers a candidate region; however, because this new region has a higher cost (shown as *B*), and the search-based method cannot determine whether this region is truly promising, escaping the current basin requires exponentially many search steps. QALM interleaves the two phases: the exploit phase quickly reaches the local minimum, and the subsequent explore phase discovers the promising region just as the search-based method does. Crucially, the next exploit phase immediately capitalizes on this discovery, descending to a new solution (shown as *C*) that surpasses the original local minimum. Through this interleaving, QALM finds much better solutions than both methods with speed comparable to rule-based optimization.

5.1 Background

In this chapter, we primarily focus on minimizing the total gate count. This metric is hardware-independent, making it suitable for benchmarking across different quantum architectures, and is widely adopted by prior work [82, 97, 149], enabling direct comparison. We also evaluate two-qubit (*CNOT*) gate count reduction in Section 5.4.

Existing quantum circuit optimizers fall into two main categories: *rule-based* and *search-based* strategies.

5.1.1 Rule-Based Optimization

Rule-based optimization applies a predefined set of local transformation rules, analogous to *peephole optimization* in classical compilers. Each rule specifies a pattern of gates that can be replaced with an equivalent but cheaper sequence. One simple example is that two adjacent Hadamard gates acting on the same qubit can be reduced to the identity operation, as the Hadamard gate is

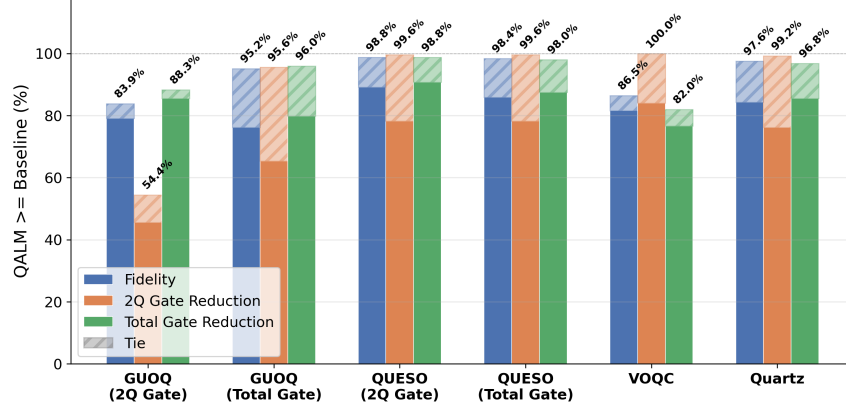


Figure 5.2: Percentage of circuits on which QALM matches or exceeds each baseline across fidelity, CX gate reduction, and total gate reduction. GUOQ and QUESO are each shown under two configurations: one optimizing for CX count and one for total gate count.

its own inverse. This is illustrated in Figure 5.3. For a more complex reduction rule, see Figure 5.4, in which we reduce the total number of rotation gates applied to the bottom qubit. These are just two of the many rules that can be applied to reduce the total gate count.

$$q_0 \begin{array}{|c|} \hline H \\ \hline \end{array} \begin{array}{|c|} \hline H \\ \hline \end{array} \Rightarrow q_0 \begin{array}{|c|} \hline I \\ \hline \end{array}$$

Figure 5.3: Hadamard self-inverse property: $HH = I$, where I denotes the identity gate, representing no operation on the qubit.

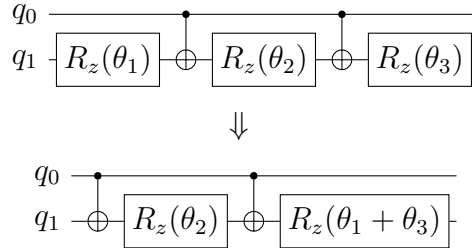


Figure 5.4: Example of an optimization rule for a gate pattern with multiple $CNOT$ and R_z gates.

Rule-based optimizers repeat these substitutions heuristically with the goal of reducing the total gate count of the quantum circuit. However, rule-based optimization is inherently greedy: substitutions are made without considering how these changes may impact future opportunities for additional rule applications. This can cause the optimizer to become stuck in a local minimum, unable to explore the optimization landscape at a large enough scale to find better solutions. One possible strategy for remedying this concern is to use a search-based approach.

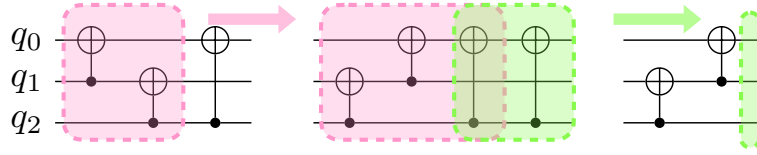


Figure 5.5: Search-based optimization example. The first transformation increases gate count ($3 \rightarrow 4$), but enables subsequent CX cancellation ($4 \rightarrow 2$).

5.1.2 Search-Based Optimization

Similar to rule-based strategies, search-based optimization considers a set of possible transformations that preserve the equivalence of the circuit. However, unlike rule-based approaches that greedily apply beneficial transformations, search-based optimizers systematically explore the space of equivalent circuits, tolerating transformations that do not reduce—or even increase—circuit size. This allows them to escape local minima by exploring a much larger portion of the optimization landscape. As a result, search-based optimizers usually require longer runtimes to explore a reasonably large portion of the search space, but they typically achieve better optimization results than rule-based optimizers given sufficient time.

To illustrate the advantage of search-based optimization, consider the example in Figure 5.5. In the first step of this example, two $CNOT$ gates are transformed into three $CNOT$ gates. While this individual transformation increases the total gate count (and thus might be rejected by a greedy rule-based approach), it alters the circuit structure in a way that enables subsequent optimizations, such as $CNOT$ cancellations, that were not previously accessible. This highlights the ability of search-based methods to traverse non-improving steps to reach a better local optimum.

5.2 Method

Current approaches to quantum circuit optimization typically fall into two distinct paradigms: rule-based and search-based optimization. We propose that these are not mutually exclusive methodologies, but rather two extremes of a single, continuous optimization spectrum. In this section, we introduce a unified framework that parameterizes the trade-off between exploration (search) and exploitation (rules). We define a parameter k , representing the depth of search exploration permitted before applying a rule-based reduction. By tuning k , we can recover existing distinct methodologies or interpolate between them to create a hybrid approach, the QALM algorithm, that captures the efficiency of rule-based systems and the comprehensiveness of search-based systems.

5.2.1 Optimization Spectrum

We model the optimization process as a traversal through a circuit space, seeking to minimize a cost function (e.g., total gate count or two-qubit gate count).

Rule-based optimization (Exploitation): Optimizers like Qiskit [68], VOQC [61], and t|ket> [130] rely on pattern matching. They are computationally efficient but “greedy”, immedi-

ately applying transformations when a pattern is matched to reduce the cost. This often leads to being trapped in local minima.

Search-based optimization (Exploration): Optimizers like Quartz [149], QUESO [144], and GUOQ [146] utilize cost-increasing transformations to explore the search space exhaustively. They effectively escape local minima but suffer from exponential time complexity as the search depth increases.

We unify these approaches by introducing an exploration depth parameter, denoted as k . This parameter dictates the number of transformation steps the optimizer takes to expand the search frontier before applying a rule-based reduction pass to collapse the frontier. The spectrum can be defined as follows:

- $k = 0$ (Pure rule-based): The optimizer performs zero search steps before applying rules. This is equivalent to rule-based optimizers.
- $k \rightarrow \infty$ (Pure search-based): The optimizer performs infinite (or exhaustive) search steps. This is equivalent to search-based optimizers.
- $0 < k < \infty$ (Hybrid): This is the domain of our proposed method. By setting a finite, non-zero k , we allow the optimizer to explore the neighborhood of the current circuit before using the rule-based optimizer to jump to a deeper local minimum.

5.2.2 The QALM Algorithm

Our algorithm, QALM, operates within the hybrid region of this spectrum. It utilizes a multi-pass strategy that progressively increases the exploration depth k . This allows the optimizer to harvest “low-hanging fruit” with fast, shallow searches before committing computational resources to deeper exploration.

Since the branching factor and effective search depth of general search-based optimization techniques scale with the size of the circuit, employing rule-based reduction as a preprocessing step significantly shrinks the search space. This ensures that the computationally expensive exploration (high k) is performed on a compacted circuit topology, preventing the search algorithm from wasting its limited depth budget on trivial cancellations that could have been resolved greedily.

Algorithm 3 details the QALM execution flow. QALM runs iteratively in passes, with the exploration depth k increasing from one until the time limit is reached. In each pass, QALM maintains a priority queue for circuit candidates (Algorithm 3 line 10), similar to search-based optimizers.

In each iteration of the search, QALM creates a circuit candidate pool according to the schedule $N_{\text{pool}}^{(k)}$ (lines 14 to 20) for parallel exploration. For each circuit in the candidate pool, QALM branches $N_{\text{branch}}^{(k)}$ times (line 23) and explores for k steps per branch (line 24). Each of the k steps is a search step in the search-based optimizer (line 8 in Algorithm 4, Section 5.3.2).

After the k steps, QALM calls a rule-based optimizer for an exploitation step (line 25, Section 5.3.1). QALM then enqueues all $N_{\text{branch}}^{(k)}$ new circuits after the exploitation step if they have not been visited already.

While k governs the balance between exploration and exploitation, the schedules $N_{\text{pool}}^{(k)}$ and

Algorithm 3 QALM Algorithm.

```

1: Input: Quantum circuit  $C_{\text{in}}$ , cost model  $\text{COST}(\cdot)$ , schedules  $N_{\text{pool}}^{(k)}$ ,  $N_{\text{branch}}^{(k)}$ , and a time limit.
2: Output: An optimized circuit  $C_{\text{out}}$ .
3:  $C_{\text{out}} \leftarrow C_{\text{in}}$ 
4: for  $k = 1, 2, \dots$  while time limit not exceeded do
5:    $C_{\text{out}} \leftarrow \text{OPTIMIZE}(C_{\text{out}}, k)$ 
6: return  $C_{\text{out}}$ 
7:
8: function  $\text{OPTIMIZE}(C_{\text{in}}, k)$ 
9:   //  $\mathcal{Q}$  is a priority queue of circuits sorted by their  $\text{COST}(\cdot)$ .
10:   $\mathcal{Q} \leftarrow \{C_{\text{in}}\}$ 
11:   $\mathcal{V} \leftarrow \{C_{\text{in}}\}$  ▷  $\mathcal{V}$  is the set of visited circuits
12:   $C_{\text{best}} \leftarrow C_{\text{in}}$ 
13:  while  $\mathcal{Q} \neq \emptyset$  and time limit not exceeded do
14:    // Make an initial circuit candidate pool of size  $N_{\text{pool}}^{(k)}$  from the priority queue.
15:     $\mathcal{P} \leftarrow \emptyset$ 
16:    for  $i \leftarrow 1$  to  $N_{\text{pool}}^{(k)}$  and  $\mathcal{Q} \neq \emptyset$  do
17:       $C \leftarrow \mathcal{Q}.\text{dequeue}()$ 
18:      if  $\text{COST}(C) < \text{COST}(C_{\text{best}})$  then
19:         $C_{\text{best}} \leftarrow C$ 
20:         $\mathcal{P} \leftarrow \mathcal{P} \cup \{C\}$ 
21:    // Branch, explore, and exploit.
22:    for each circuit  $C \in \mathcal{P}$  do
23:      for  $i \leftarrow 1$  to  $N_{\text{branch}}^{(k)}$  do ▷ Branch  $N_{\text{branch}}^{(k)}$  times
24:         $C \leftarrow \text{EXPLORE}(C, k)$  ▷ Explore  $k$  steps
25:         $C \leftarrow \text{EXPLOIT}(C)$  ▷ Exploit for one step
26:        if  $C \notin \mathcal{V}$  then
27:           $\mathcal{Q}.\text{enqueue}(C)$ 
28:           $\mathcal{V} \leftarrow \mathcal{V} \cup \{C\}$ 
29:  return  $C_{\text{best}}$ 

```

Algorithm 4 The exploration step.

Require: Circuit transformations $\mathcal{T}$.

```

1: function EXPLORE( $C, k$ )
2:    $S \leftarrow C$   $\triangleright$  Search from all indices in the first step, then from modified indices from the
   previous step in the current iteration
3:   for  $j \leftarrow 1$  to  $k$  do
4:     while true do
5:       transformation  $t \leftarrow \text{SAMPLEUNIFORM}(\mathcal{T})$ 
6:       position  $c \leftarrow \text{SAMPLEUNIFORM}(S)$ 
7:       if  $t$  is applicable at  $c$  then
8:          $(C, S) \leftarrow \text{APPLY}(C, c, t)$   $\triangleright$  Returns new circuit and modified indices
9:         break
10:  return  $C$ 

```

$N_{\text{branch}}^{(k)}$ determine the exploration *strategy*. Specifically, setting $N_{\text{pool}}^{(k)} = 1$ reduces to a best-first search, whereas higher $N_{\text{pool}}^{(k)}$ and $N_{\text{branch}}^{(k)}$ produce a wider beam search. The schedule formulation makes the framework general: in particular, the greedy mode of Section 5.3.3 is a different schedule for $k \leq 2$ that effectively forces $N_{\text{pool}} = N_{\text{branch}} = 1$ and rejects cost-increasing moves. For our main search loop, we use the constant schedule $N_{\text{pool}}^{(k)} = 1$ and $N_{\text{branch}}^{(k)} = 3$ and analyze their impact in Section 5.4.7.

5.3 Implementation

This section describes the implementation of QALM, which integrates a rule-based optimizer and a search-based optimizer under a unified control loop. We first describe our rule-based component, ROQC (Section 5.3.1), which synthesizes techniques from several prior rule-based optimizers [36, 61, 97]. We then present the search-based component (Section 5.3.2), which leverages transformation rules from Quartz [149]. Finally, we discuss a greedy mode that accelerates termination for small k (Section 5.3.3).

5.3.1 ROQC: Rule-based Optimizer for Quantum Circuits

ROQC is our rule-based optimizer that implements all optimization routines from VOQC [61], including Not propagation, Hadamard reduction, single-qubit cancellation, two-qubit gate cancellation, and rotation merging. To further improve optimization quality and speed, we extend rotation merging with the tagging technique from PhasePoly [36], which assigns a tag to each R_z gate so that gates with the same tag can be merged. This finds more merging opportunities than the previous R_z - $CNOT$ block approach.

5.3.2 Search-Based Optimization

For the search-based component, we leverage the transformation rules from Quartz [149]. Specifically, we use the $(5, 3)$ -complete circuit transformation set (ECC set), which generates equivalence-preserving transformations involving at most 3 qubits and at most 5 gates. The transformation set $\mathcal{T}$ in Algorithms 4 and 5 is pre-computed and verified for correctness, allowing QALM to apply these transformations efficiently during the search phase.

During exploration, QALM biases transformation selection toward recently modified regions. This encourages the discovery of optimization sequences in which one transformation enables another, ensuring that the k exploration steps cannot be easily obtained by performing one exploration step k times independently.

Similar to Quartz [149], to prevent the exploration from consuming too much memory, whenever the priority queue of Algorithm 3 contains more than 2,000 circuits, we prune it and keep only the top 1,000 circuits. In addition, to save memory, we implement the set of visited circuits $\mathcal{V}$ as a hash table, and only store the hash value instead of the entire circuit.

5.3.3 Greedy Mode

In preliminary experiments, we observe that many circuits contain “low-hanging fruit”, i.e., optimizations discoverable with minimal search depth. To exploit this, we introduce a *greedy mode* for the two initial passes with $k = 1$ and $k = 2$, terminating each branch early once an improvement is found. This allows QALM to quickly harvest obvious optimizations before committing to deeper exploration.

Specifically, similar to setting N_{pool} and N_{branch} to 1, we maintain only one circuit candidate, and only accept the circuit transformation if it (after the exploitation step) reduces the circuit cost, by replacing OPTIMIZE in Algorithm 3 with OPTIMIZEGREEDY in Algorithm 5.

Instead of looping until a fixed time limit, as shown in Algorithm 5 line 3, we only perform the exploration for one pass for each circuit transformation. In the exploration step, instead of applying a transformation uniformly at random, we start from the position where we last applied a circuit transformation to find the next available position (Algorithm 5 line 12), making the pass more deterministic as the greedy mode is fast and we can afford to loop over all transformation-position combinations for one pass.

For $k \geq 2$ during the greedy pass, the function GREEDYSTEP does not go to the exploitation phase immediately, and we explore all possible transformations from the modified indices recursively at line 29. After $k - 1$ levels of recursion, we run the exploitation step, and stop the recursion chain to move on to the next iteration if the circuit cost is reduced after the k transformations and the exploitation step. We evaluate the empirical benefit of this greedy mode in Section 5.4.5.

Algorithm 5 The greedy mode ($k \leq 2$).

Require: Circuit transformations $\mathcal{T}$.

```

1: function OPTIMIZEGREEDY( $C_{\text{in}}, k$ )
2:    $C_{\text{best}} \leftarrow C_{\text{in}}$ 
3:   for  $t \in \mathcal{T}$  do
4:      $S \leftarrow C_{\text{best}}$   $\triangleright$  Search from all indices in the first step, then from modified indices
      from the previous step in the current iteration
5:     while true do
6:        $\text{found\_improvement} \leftarrow \text{false}$ 
7:       for position  $c \in S$  do
8:         if  $t$  is applicable at  $c$  then
9:            $(C, S) \leftarrow \text{APPLY}(C_{\text{best}}, c, t)$   $\triangleright$  Returns new circuit and modified indices
10:           $\text{GREEDYSTEP}(C, k - 1, S)$ 
11:          if  $\text{found\_improvement}$  then
12:             $S \leftarrow \{c' \in C_{\text{best}} \mid c' \geq c\} \cup \{c' \in C_{\text{best}} \mid c' < c\}$   $\triangleright$  Wrap-around
      from current position
13:       break
14:       if not  $\text{found\_improvement}$  then
15:         break
16:   return  $C_{\text{best}}$ 
17:
18: function GREEDYSTEP( $C, k, S$ )
19:   if  $k = 0$  then
20:      $C \leftarrow \text{EXPLOIT}(C)$   $\triangleright$  Exploit for one step after exploring for  $k$  steps
21:     if  $\text{COST}(C) < \text{COST}(C_{\text{best}})$  then
22:        $C_{\text{best}} \leftarrow C$ 
23:        $\text{found\_improvement} \leftarrow \text{true}$ 
24:     return
25:   for position  $c \in S$  do
26:     for  $t \in \mathcal{T}$  do
27:       if  $t$  is applicable at  $c$  then
28:          $(C', S') \leftarrow \text{APPLY}(C, c, t)$   $\triangleright$  Returns new circuit and modified indices
29:          $\text{GREEDYSTEP}(C', k - 1, S')$ 
30:         if  $\text{found\_improvement}$  then
31:           return

```

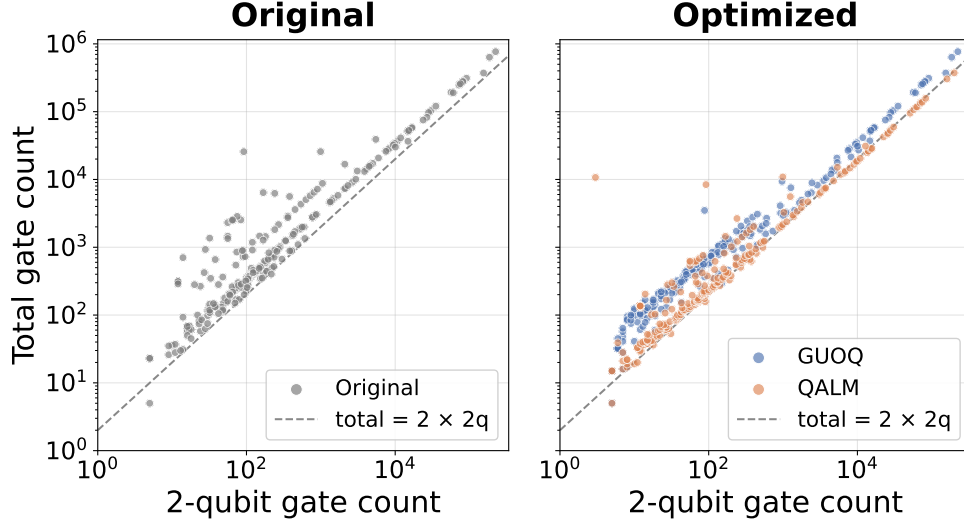


Figure 5.6: Gate count distribution in our benchmark. The dashed line indicates where the total gate count is twice the two-qubit gate count, or equivalently, where the number of single-qubit gates equals the number of two-qubit gates. The left panel shows the distribution before optimization, and the right panel shows the distribution after optimization using QALM and GUOQ with two-qubit gate optimization mode, which is the strongest baseline in our evaluation.

5.4 Evaluation

5.4.1 Experiment Setup

To evaluate our framework, we employ a suite of quantum circuit benchmarks consisting of 248 circuits compiled by GUOQ [145, 146]. This circuit set originally consists of 250 circuits; however, `qft_10` and `qft_16` are removed because they are identified as identity circuits. This suite aggregates diverse quantum circuits from several established frameworks [11, 61, 82, 97, 144, 149, 164]. These benchmarks include variational circuits (QAOA, VQE), simulation circuits (Trotterized TFIM, Heisenberg, and XY models), classical arithmetic circuits (adders, GF multipliers, modular arithmetic, and square-root circuits), and quantum algorithms (Grover, Shor, QFT, and QPE).

We use gate count in the Nam gate set ($X, R_z, H, CNOT$) as our main benchmark metric, following existing studies. Figure 5.6 shows the gate count distribution of the benchmark suite. The 248 circuits span five orders of magnitude in gate count.

In the following experiments, we compare QALM against Quartz [149] (v0.3.2), GUOQ [146], QUESO [144], and VOQC [61].

We execute GUOQ and QUESO within the Docker environment provided by their respective authors. For QUESO, when optimizing for two-qubit gates, we use the parameters `-g NAM -opt TWO_Q --rules-dir /home/queso_rules/ -search BEAM -temp 0 -q 8000 -resynth NONE`; when optimizing for total gates, we use the parameters `-g NAM -opt TOTAL --rules-dir /home/queso_rules/ -search BEAM -temp 0 -q 8000 -resynth NONE`, consistent with the authors’ recommendations. For GUOQ, when

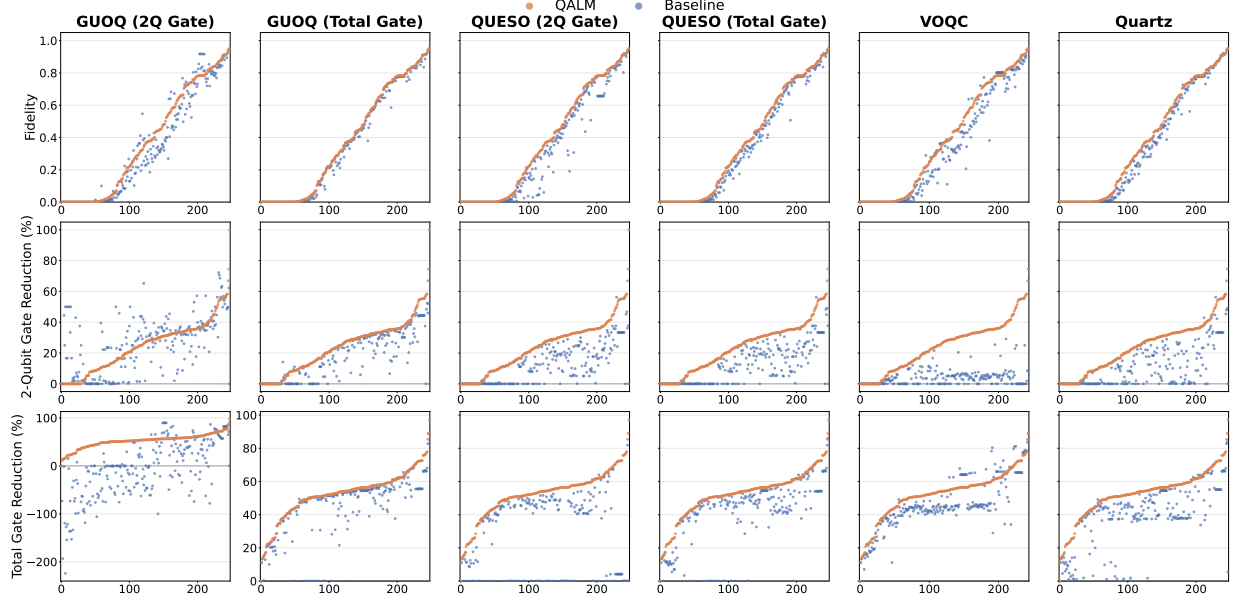


Figure 5.7: Per-circuit comparison of QALM against GUOQ, QUESO, Quartz, and VOQC. Each column corresponds to a baseline optimizer, and each row shows a different metric: output circuit fidelity (top), CX gate reduction percentage (middle), and total gate reduction percentage (bottom). Circuits are sorted by the QALM metric value.

optimizing for two-qubit gates, we run GUOQ with `-g NAM -opt TWO_Q --rules-dir /home/queso_rules/ --resynth-weight 180`; when optimizing for total gate count, we use the parameters `-g NAM -opt TOTAL --rules-dir /home/queso_rules/ --resynth-weight 180`. GUOQ only performs circuit resynthesis when optimizing for two-qubit gates. Note that resynthesis is an approximate optimization and does not preserve circuit semantics exactly.

All experiments are conducted on servers equipped with AMD EPYC 7763 processors. Each optimizer is given one CPU core, 32 GiB of RAM, and a one-hour time limit for each run. To capture the joint effect of $CNOT$ gates and single-qubit gates on circuit quality, we estimate fidelity as the product of per-gate fidelities [146], with $f_{1q} = 0.999$ for each single-qubit gate and $f_{2q} = 0.99$ for each two-qubit gate, consistent with typical NISQ noise models [96, 135]:

$$F = f_{1q}^{n_{1q}} \cdot f_{2q}^{n_{2q}}, \quad (5.1)$$

where n_{1q} and n_{2q} denote the single- and two-qubit gate counts, respectively.

5.4.2 Main Results

Figure 5.7 presents a per-circuit comparison against GUOQ, QUESO, VOQC, and Quartz on output circuit fidelity, $CNOT$ gate reduction, and total gate reduction. QALM achieves higher fidelity than all four baselines on the majority of circuits. We run Quartz without its preprocessing pass, and we evaluate this further in Section 5.4.4.

Metric	QALM	QALM (1 minute)	GUOQ (2Q Gate)	GUOQ (Total Gate)	QUESO (2Q Gate)	QUESO (Total Gate)	VOQC	Quartz
Fidelity	0.3864	0.3747	0.3475	0.3676	0.3362	0.3554	0.3428	0.3526
2Q Gate Reduction (%)	24.61	21.18	25.41	18.83	13.19	13.37	4.24	12.42
Total Gate Reduction (%)	52.34	51.35	-2.63	43.89	27.40	39.72	46.37	40.73

Table 5.1: Per-metric averages across the benchmark suite. Each optimizer is given one hour per circuit, except for “QALM (1 minute)”, which is given only one minute per circuit.

The gate reduction rows of Figure 5.7, together with the averages in Table 5.1, reveal distinct trade-off profiles. Against GUOQ in two-qubit gate optimization mode, QALM achieves nearly identical *CNOT* reduction (24.61% vs. 25.41%) but dramatically better total gate reduction (52.34% vs. -2.63%): GUOQ’s resynthesis frequently introduces many additional single-qubit gates, yielding negative total gate reduction on many circuits, whereas QALM avoids this entirely. Against VOQC, the situation is reversed: both achieve similar total gate reductions (52.34% vs. 46.37%), but QALM yields significantly greater *CNOT* reduction (24.61% vs. 4.24%).

In terms of fidelity, QALM (average 0.3864) outperforms both VOQC (0.3428) and GUOQ in two-qubit mode (0.3475), as it balances single- and two-qubit gate reduction most effectively. Against the remaining baselines, QUESO (in both optimization modes), Quartz, and GUOQ in total gate optimization mode, QALM achieves better results on all three metrics, as shown in Table 5.1.

5.4.3 Optimization Efficiency

In Section 5.4.2, each optimizer is given a one-hour time limit per circuit. To demonstrate QALM’s fast convergence, we additionally evaluate it with only a one-minute time limit per circuit. As shown in Table 5.1, reducing the time budget by a factor of 60 degrades QALM’s average fidelity and gate reduction by less than 1%, yet it still surpasses the one-hour fidelity and total gate reduction of every baseline.

5.4.4 Quartz’s Benchmark

Section 4.2 describes a preprocessing pass used by Quartz to translate *CCZ* gates into the Nam gate set (X , R_z , H , *CNOT*). However, our main benchmark suite is already in the Nam gate set, and it is hard to reconstruct the *CCZ* gates to fully leverage the Toffoli decomposition in Quartz’s preprocessing pass. Therefore, we also compare with Quartz on the 26 benchmark circuits in Section 4.4. This evaluation favors Quartz with preprocessing because the preprocessing is run separately, and its time is not counted towards the one-hour time limit.

Table 5.2 presents the gate counts achieved by each optimizer on all 26 benchmark circuits. Quartz achieves a better result than Table 4.2 because this evaluation is done on a newer version of Quartz (v0.3.2), with more engineering improvements. QALM achieves the lowest or equally lowest gate count on all of them except for one circuit compared to Quartz with preprocessing. Compared to the purely rule-based VOQC, QALM reduces gate count by an additional 7.3%

Table 5.2: Comparing QALM with existing optimizers on total gate count reduction under a 1-hour time limit, on 26 benchmark circuits used by Quartz. The best result for each circuit is in bold.

Circuit	Orig.	VOQC	QUESO	GUOQ	Quartz w/o Preprocess	Quartz w/ Preprocess	QALM
adder_8	900	596	645	578	897	621	512
barenco.tof_3	58	40	38	39	38	38	38
barenco.tof_4	114	72	68	72	90	72	62
barenco.tof_5	170	104	98	102	134	104	94
barenco.tof_10	450	264	248	262	420	264	236
csla_mux_3	170	158	148	145	146	148	144
csum_mux_9	420	280	392	346	400	279	280
gf2^4_mult	225	186	195	169	199	173	167
gf2^5_mult	347	287	306	271	316	276	265
gf2^6_mult	495	401	426	376	462	383	370
gf2^7_mult	669	543	603	500	648	524	500
gf2^8_mult	883	706	815	673	880	695	660
gf2^9_mult	1095	879	996	824	1079	861	805
gf2^10_mult	1347	1065	1225	1046	1327	1040	972
mod5_4	63	51	33	26	24	24	24
mod_mult_55	119	92	100	101	98	93	86
mod_red_21	278	184	198	198	215	202	175
qcla_adder_10	521	416	452	395	488	401	385
qcla_com_7	443	269	331	289	418	288	252
qcla_mod_7	884	678	791	657	878	646	622
rc_adder_6	200	152	176	180	180	152	152
tof_3	45	35	35	35	35	35	35
tof_4	75	55	55	55	55	55	51
tof_5	105	75	75	75	76	75	71
tof_10	255	175	175	180	177	175	165
vbe_adder_3	150	89	83	80	114	89	71
Geo. Mean Reduction	–	27.0%	23.9%	29.2%	16.5%	29.7%	34.3%

on average. Against the search-based optimizers, QALM outperforms Quartz (without preprocessing) by 17.8%, QUESO by 10.4%, GUOQ by 5.1%, and Quartz (with preprocessing) by 4.6%.

These numbers are significant because they are absolute percentages (relative to the original gate count instead of optimized gate count), and each gate has a significant runtime and fidelity cost on near-term quantum computers. In addition, Quartz with preprocessing takes advantage of higher-level knowledge about which gates originally compose a Toffoli gate, yet QALM still matches or outperforms it on all but one circuit. These results demonstrate that the interleaved combination of rule-based and search-based optimization in QALM yields consistently better circuit quality than either approach alone. The following ablation studies use the same circuits as Table 5.2.

5.4.5 Greedy Mode

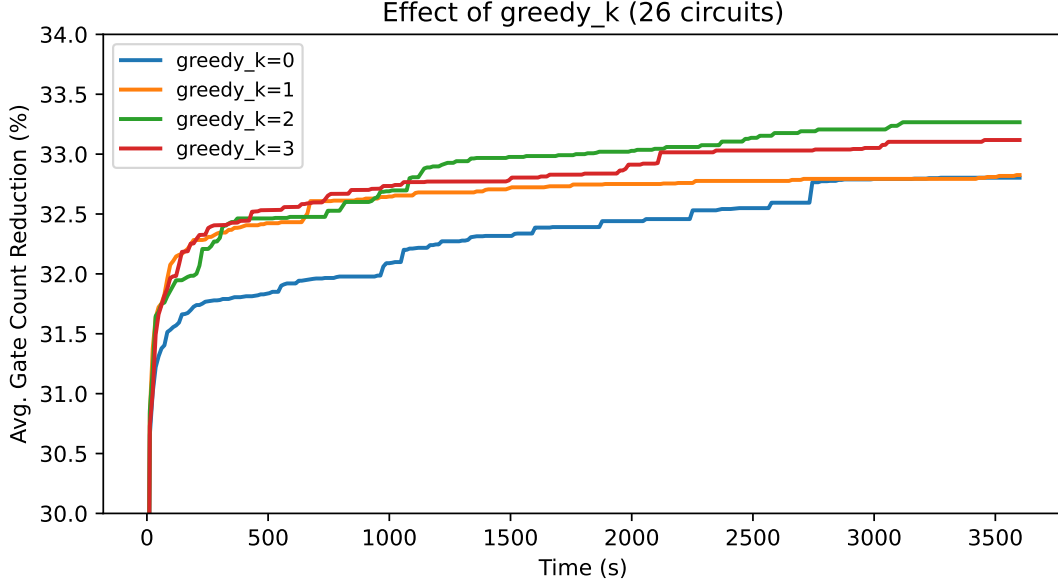


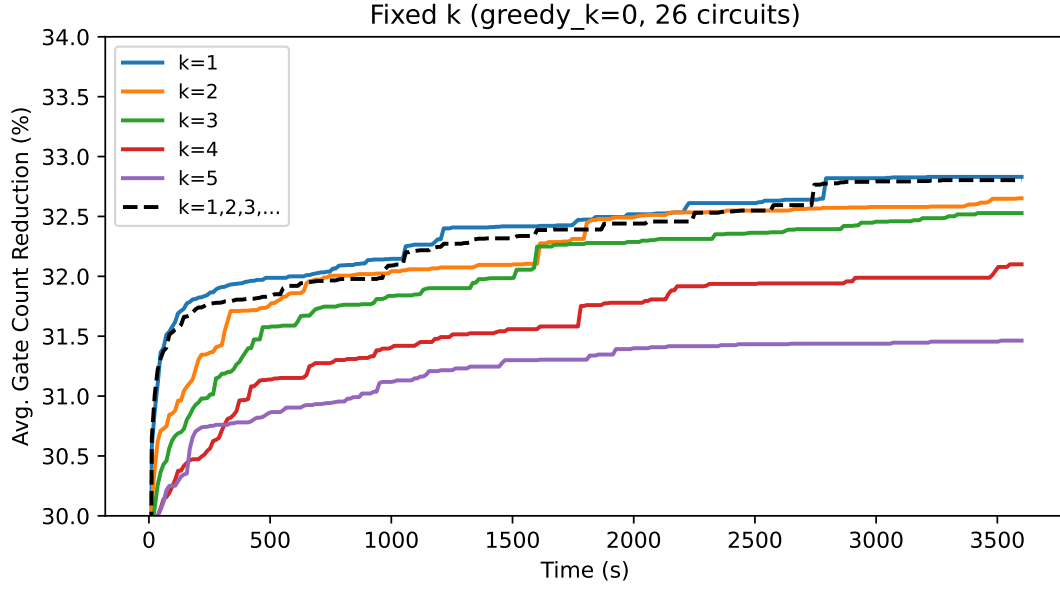
Figure 5.8: Effect of the greedy mode. Each curve shows QALM run with greedy mode (accepting only cost-reducing transformations) applied up to depth $k \leq \text{greedy_k}$, with unrestricted search beyond.

Figure 5.8 shows the effect of the greedy mode. While the final gate count reductions differ by less than 0.5% on average (32.80% without greedy mode versus 33.27% with greedy mode for $k \leq 2$), the speed of convergence varies substantially. Without the greedy mode, QALM takes 968 seconds to reach an average gate count reduction of 32% on this benchmark suite. Enabling the greedy mode for $k = 1$ alone reaches the same threshold in 89 seconds, an $11\times$ speedup.

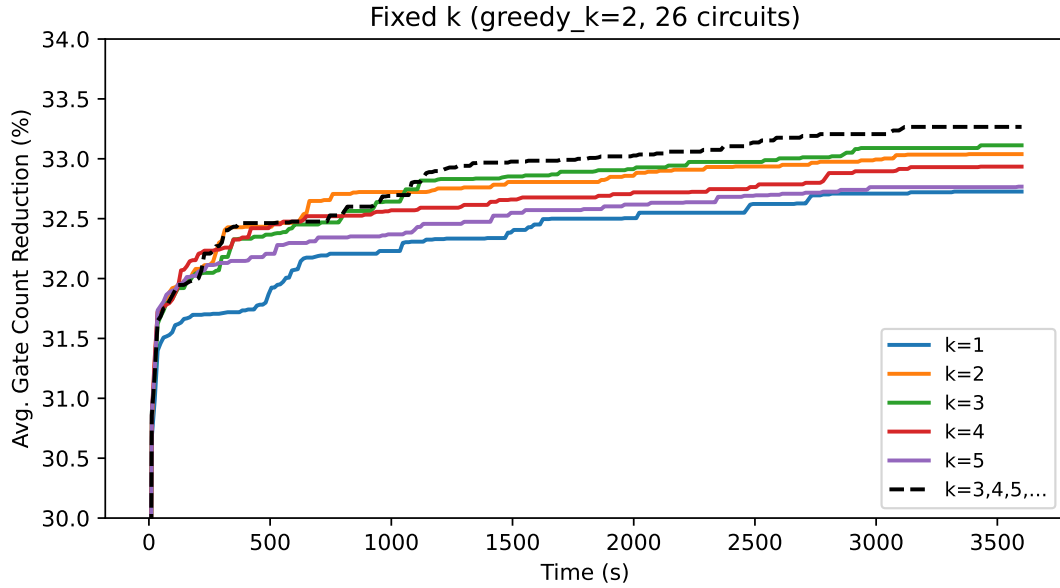
However, it is not always beneficial to extend greedy mode to larger search depths. When the greedy mode is only enabled for $k = 1$, the greedy phase takes 5.5 seconds on average. For $k \leq 2$, it takes 82 seconds on average. Extending greedy mode to $k \leq 3$ pushes the greedy phase to 724 seconds on average, and two of the 26 circuits (`gf2^9_mult` and `gf2^10_mult`) fail to finish the greedy phase within the one-hour budget. The resulting final reduction (33.12%) is also slightly below the $k \leq 2$ setting (33.27%), since the extra time spent exhausting $k = 3$ greedy transformations crowds out the main search. With a time limit of one hour, enabling greedy mode for $k \leq 2$ strikes a balance for harvesting “low-hanging fruit” in circuit optimization.

5.4.6 Exploration Depth

Figure 5.9 shows the results of fixing the exploration depth k in QALM. Larger fixed k explores deeper per iteration but fewer iterations fit in the budget; small fixed k iterates quickly but saturates early. Without the greedy mode (Figure 5.9a), the final average reduction decreases monotonically with k , from 32.83% at $k = 1$ down to 31.46% at $k = 5$, because there are still



(a) Fixed k with greedy mode disabled.



(b) Fixed k after greedy mode with $k = 1, 2$.

Figure 5.9: Effect of fixing the exploration depth k in QALM. Each curve runs QALM with a single fixed $k \in \{1, \dots, 5\}$ (no depth advancement), averaged over 26 benchmark circuits.

shallow optimizations left unharvested, and deeper exploration incurs a higher per-iteration cost, resulting in fewer iterations within the one-hour budget.

With the greedy mode enabled for $k \leq 2$ and a fixed k after that (Figure 5.9b), the ordering shifts: $k = 3$ becomes the best single fixed depth (33.11%), since the greedy phase already harvests the shallow rewrites, and the remaining budget is spent on deeper moves. Note that no single fixed k matches the advancing- k baseline (33.27%), and this motivates the advancing- k schedule used in QALM: it exploits “easier” optimizations first with small k , then spends the remaining budget on deeper exploration.

5.4.7 Exploration Strategy

We now examine the impact of pool size N_{pool} and branch factor N_{branch} on optimization quality. As described in Section 5.3.3, the greedy passes do not use N_{pool} and N_{branch} parameters directly (equivalent to setting them to 1). For the main search loop, we vary each parameter independently over $\{1, 2, 3, 4, 5\}$ while holding the other at its default ($N_{\text{pool}} = 1$, $N_{\text{branch}} = 3$), with greedy mode enabled for $k \leq 2$.

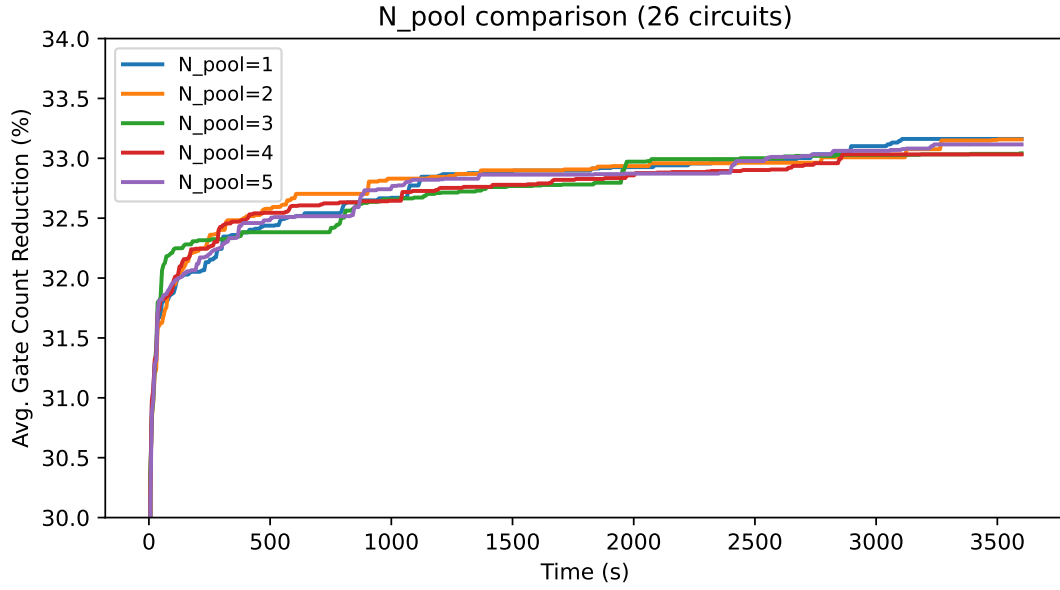
Figure 5.10 shows the results. Pool size N_{pool} has no meaningful effect on optimization quality: across all five settings, the final average reduction ranges from 33.03% to 33.16% (a spread of 0.13%), well within run-to-run variance. This insensitivity indicates that the cost-guided best-first search already selects promising candidates from a single-pool frontier, so widening the beam yields no additional benefit within the one-hour budget.

Branch factor N_{branch} , in contrast, materially affects final quality and exhibits a sweet spot at $N_{\text{branch}} = 3$. At $N_{\text{branch}} = 1$, every expansion retains only a single successor, so the search queue holds at most one circuit at any time, the procedure collapses to a single trajectory with no ability to recover from an unfavorable local choice, and the average reduction drops to 31.33%. Raising N_{branch} introduces the diversity needed to escape such local minima, and quality climbs to 32.75% at $N_{\text{branch}} = 2$ and peaks at 33.37% at $N_{\text{branch}} = 3$. Beyond this point, the additional branches enlarge the frontier faster than the one-hour budget can explore it, diluting the per-branch search effort and slightly degrading quality (33.30% and 33.21% at $N_{\text{branch}} = 4$ and 5). We therefore use $N_{\text{pool}} = 1$ and $N_{\text{branch}} = 3$ as the default configuration for all other experiments.

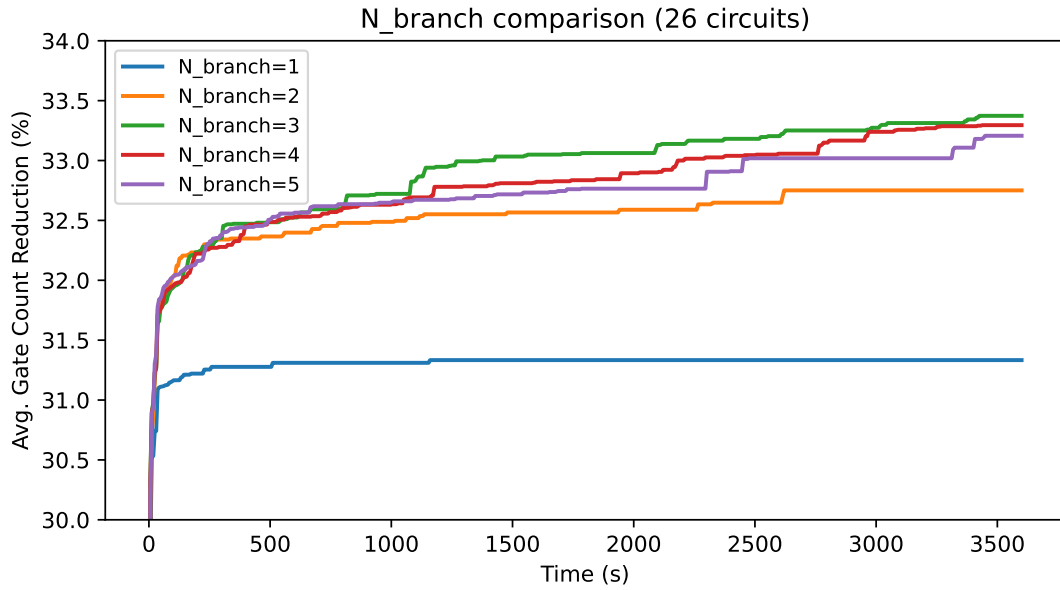
5.5 Conclusion

We identify a fundamental bottleneck in search-based quantum circuit optimization: once a truly promising point has a higher cost than the current local minimum, a pure search must expend exponentially many steps to verify that the point actually leads to a better basin. This inability to cheaply distinguish genuinely promising points from cost-increasing detours is what forces the long-standing compromise between speed and quality.

We present QALM, which resolves this bottleneck by interleaving search-based exploration with rule-based exploitation in a single control loop. Each exploit phase immediately descends from every candidate uncovered by the preceding explore phase, so promising points are verified in a single rule-based pass rather than through exponential search. By progressively advancing the exploration depth k , QALM exhausts shallow optimizations first and reserves its budget for



(a) Pool size N_{pool} ($N_{\text{branch}} = 3$ fixed).



(b) Branch factor N_{branch} ($N_{\text{pool}} = 1$ fixed).

Figure 5.10: Effect of pool size N_{pool} and branch factor N_{branch} on average gate count reduction, over 26 benchmark circuits with a one-hour timeout. N_{pool} has little effect on final quality, while N_{branch} exhibits a sweet spot at $N_{\text{branch}} = 3$.

the deeper moves that matter. Our evaluation confirms that this design eliminates the historical trade-off between speed and quality: QALM outperforms existing rule-based and search-based optimizers on total gate count reduction, and it surpasses the one-hour results of all search-based baselines within the first minute.

While QALM significantly advances the speed and quality of circuit optimization, applying any global optimization strategy directly to massive, monolithic quantum circuits eventually hits a scalability wall. To overcome this fundamental bottleneck, the next chapter introduces OAC. By shifting the optimization paradigm from global search to segment-level local optimality, OAC provides a scalable cut-and-meld framework that can accelerate existing optimizers by more than an order of magnitude.

Chapter 6

OAC: Local Optimization

6.1 Introduction

As discussed in Chapter 1, realizing the practical advantages of quantum computing requires highly optimized circuits that can execute within the strict fidelity limits of near-term hardware. Optimization remains a critical bottleneck in the compilation pipeline. However, because global optimization of quantum circuits is QMA-hard [97], attaining perfect global optimality is unlikely to be efficient.

Consequently, researchers have largely turned to heuristic or peephole optimization strategies. Early approaches, such as the heuristics developed by Nam et al. [97] and formally verified by VOQC [61], take at least quadratic time in the number of gates, severely hindering scalability for circuits containing hundreds of thousands of operations. Later automated approaches, such as Quartz [149] (Chapters 3 and 4) and its successors [82, 144], improve the discovery of optimizations but still require significant compilation time and offer no formal quality guarantees on the resulting circuit. In practice, these optimizers can take hours to process moderately large circuits.

Given this state of the art, this chapter explores a fundamental theoretical and practical question: *is it possible to offer a formal quality guarantee while also ensuring efficiency?*

In this chapter, we answer this question affirmatively by bridging quality and efficiency guarantees through a concept called *local optimality*. We introduce a rewriting semantics and a highly efficient local-optimization algorithm called OAC (Optimize-and-Compact). Unlike global optimality, local optimality ensures that each segment of the circuit is optimal with respect to an abstract *oracle* optimizer.

To achieve this efficiently, OAC employs a variant of the circuit partitioning technique—a concept often utilized in state-vector simulation, as will be seen in Chapter 7—to hierarchically divide circuits. It optimizes small segments independently and uses a novel *melding* technique to optimize over the cuts, ensuring no crucial optimizations are missed at the boundaries. The algorithm operates in optimization and compaction phases, repeating until convergence.

Crucially, OAC is “unopinionated”: it does not make restrictions on the specific optimizations performed, meaning it can be instantiated with existing optimizers to amplify their effectiveness on large-scale circuits.

This chapter presents the formal definition of local optimality alongside rigorous proofs establishing segment optimality and linear time cost bounds. Extensive empirical evaluations demonstrate that OAC improves optimization efficiency by over an order of magnitude while maintaining strict quality guarantees.

Specific contributions detailed in this chapter include the following:

- The formulation and formal definition of local optimality, alongside a rewriting semantics proving that saturating rewrites yield locally optimal circuits.
- The OAC algorithm for optimizing quantum circuits locally via cut-and-meld techniques.
- Proofs of correctness and segment optimality for OAC.
- Run-time complexity bounds proving OAC accepts a linear time cost bound in terms of oracle calls.
- A comprehensive empirical evaluation demonstrating the practicality, scalability, and performance benefits of OAC across a variety of quantum circuits.

6.2 Local Optimality

In this section, we introduce *local optimality* for quantum circuits using a circuit language called LAQE, which represents circuits as sequences of layers (see Section 2.2.1 for the layer representation). We define local optimality based on three components: (1) a base optimizer called the *oracle*, (2) a *cost* function that evaluates the circuit quality, and (3) a *segment size* Ω , which determines the scope of *local optimizations*. A segment refers to a contiguous sequence of layers. Roughly speaking, a circuit is locally optimal when it satisfies the following conditions:

1. No local optimizations are possible, i.e., the oracle cannot optimize any segments of size Ω .
2. All circuit segments are as compact as possible with no unnecessary gaps.

For a locally optimal circuit, the oracle cannot find more optimizations unless it operates on segments larger than Ω . Because our definition is parametric in terms of the oracle, we can define local optimality for any quantum gate set by instantiating appropriate oracles.

We then develop a circuit rewriting semantics that produces locally optimal circuits. The semantics only uses the oracle on small circuit segments, each containing at most Ω contiguous layers. Using this semantics, we prove that for a general class of cost functions, *any* circuit can be transformed into a locally optimal circuit. This makes local optimality applicable to various cost functions such as gate count, T count, and many others.

6.2.1 Circuit Syntax and Semantics

We present our circuit language called LAQE (Layered Quantum Representation) which represents a quantum circuit as a sequence of layers. Figure 6.1 shows the abstract syntax of the language. We let the variable q denote a qubit, and G denote a gate. For simplicity, we consider only unary gates $g(q)$ and binary gates $g(q_1, q_2)$, where g is a gate name in the desired gate set. These definitions can be easily extended to support gates of any arity.

<i>Qubit</i>	q	$\text{QUBITS}(G) \triangleq \begin{cases} \{q\}, & G = g(q) \\ \{q_1, q_2\}, & G = g(q_1, q_2) \end{cases}$
<i>Gate Name</i>	g	$\text{QUBITS}(L) \triangleq \bigcup_{G \in L} \text{QUBITS}(G)$
<i>Gate</i>	$G ::= g(q) \mid g(q_1, q_2)$	
<i>Layer</i>	$L ::= \{G_0, G_1, \dots, G_{t-1}\}$	$G_1 \diamond G_2 \Leftrightarrow \text{QUBITS}(G_1) \cap \text{QUBITS}(G_2) = \emptyset$
<i>Circuit</i>	$C ::= \langle L_0, L_1, \dots, L_{n-1} \rangle$	$L_1 \diamond L_2 \Leftrightarrow \text{QUBITS}(L_1) \cap \text{QUBITS}(L_2) = \emptyset$
		$C \text{ well-formed} \Leftrightarrow$
		$\forall L \in C. \forall G_1, G_2 \in L. G_1 \diamond G_2$

Figure 6.1: Syntax of LAQE and well-formed circuits.

$$\begin{array}{c}
\frac{\forall i. \forall G \in L_i. L_{i-1} \not\in \{G\}}{\langle L_0, \dots, L_{n-1} \rangle \text{ compact}} \quad \frac{\forall i, j. i \leq j \leq i + \Omega \Rightarrow \text{COST}(\text{ORACLE}(C[i : j])) = \text{COST}(C[i : j])}{C \text{ segment-optimal}_\Omega} \\
\frac{C \text{ compact} \quad C \text{ segment-optimal}_\Omega}{C \text{ locally-optimal}_\Omega}
\end{array}$$

Figure 6.2: Definition of local optimality, parameterized by a COST function, an ORACLE optimizer, and a segment length Ω .

A LAQE circuit C consists of a sequence of layers $\langle L_0, \dots, L_{n-1} \rangle$, where each layer L_i is a set of gates that are applied to qubits in parallel. The circuit is *well formed* if the gates of every layer act on disjoint qubits, i.e., no layer can apply multiple gates to the same qubit. As a shorthand, we write $G_1 \diamond G_2$ to denote that gates G_1 and G_2 act on disjoint qubits, i.e., $\text{QUBITS}(G_1) \cap \text{QUBITS}(G_2) = \emptyset$. We similarly write $L_1 \diamond L_2$ for the same condition on layers. Note that we implicitly assume well-formedness throughout the section because it is preserved by all our rewriting rules.

We define the *length* of a LAQE circuit as the number of layers, and the *size* of a circuit C , denoted $|C|$, as the total number of gates. A *segment* is a contiguous subsequence of layers of the circuit, and a *k-segment* is a segment of length k . We use the Python-style notation $C[i : j]$ to represent a segment containing layers $\langle L_i, \dots, L_{j-1} \rangle$ from circuit C . In the case of overflow (where either $i < 0$ or $j > \text{LENGTH}(C)$), we define $C[i : j] = C[\max(0, i) : \min(j, \text{LENGTH}(C))]$.

Two circuits C and C' can be concatenated together as $C; C'$, creating a circuit containing the layers of C followed by layers of C' . Formally, if $C = \langle L_0 \dots L_{n-1} \rangle$ and $C' = \langle L'_0 \dots L'_{m-1} \rangle$, then $C; C' = \langle L_0 \dots L_{n-1}, L'_0 \dots L'_{m-1} \rangle$.

6.2.2 Local Optimality

We introduce two optimality properties on circuits written in our language LAQE. These properties are defined on the following parameters. First, we assume an abstract **COST** function over circuits, where a smaller cost means a better quality circuit. Second, we introduce a parameter Ω , which represents the maximum segment length that can be considered for optimization. In this context, the optimizations are *local* because each optimization can only optimize a circuit segment of length Ω . Third, we assume an **ORACLE** optimizer that takes a circuit of length Ω and produces an equivalent circuit, optimized w.r.t. the cost function. We assume that the oracle and the cost function are compatible, meaning that the oracle can only decrease the cost:

$$\forall C. \text{COST}(\text{ORACLE}(C)) \leq \text{COST}(C).$$

Segment optimal circuits. A circuit is segment-optimal if each and every Ω -segment of the circuit is optimal for the given **ORACLE** and **COST** function. Figure 6.2 defines segment optimal circuits as the judgment C **segment-optimal** $_{\Omega}$. The judgment checks that any segment $C[i : j]$ whose length is smaller than Ω (i.e., $i \leq j \leq i + \Omega$) can not be further optimized by the oracle. Thus, calling the oracle on any such segment does not improve the cost function.

Compact circuits. A circuit is *compact* if every gate is in the left-most possible position, i.e., in the earliest layer possible. Figure 6.2 formalizes this with the judgement C **compact**. The judgment checks every gate G and ensures that at least one qubit used by G is also used by the previous layer (i.e., $G \in L_i$ and $L_{i-1} \not\subseteq \{G\}$). If every gate satisfies this condition, the circuit is compact.

Ensuring that a layered circuit is as compact as possible is important because compact circuits are more amenable to local optimizations. For example, consider two H gates on the same qubit, one of them in layer 0 and the other in layer 3, with no gates in between. Suppose we have an optimization that cancels two H gates on the same qubit, when they are on adjacent layers. This optimization would not apply to our circuit because the two H gates are not adjacent. However, if the circuit did not have such unnecessary “gaps”, we could apply the optimization and eliminate the two H gates from our circuit. Compaction ensures that such optimizations are not missed.

Locally Optimal Circuits. A circuit is locally optimal if it is both compact and segment optimal. This means that each and every Ω -segment of the circuit is optimal for the given **ORACLE** and **COST** function and the circuit as a whole is compact, ensuring that no more local optimizations are possible. Figure 6.2 defines locally optimal circuits as the judgment C **locally-optimal** $_{\Omega}$.

6.2.3 Circuit Rewriting for Local Optimality

In this section, we present a rewriting semantics for producing locally optimal circuits. The rewriting semantics performs local optimizations, each of which rewrites an Ω -segment, and compacts the circuit as needed. Given a **COST** function, an **ORACLE**, and a segment length Ω , we define the rewriting semantics as a relation $C \xrightarrow{\Omega} C'$ which rewrites the circuit C to

$$\begin{array}{c}
\frac{\text{LENGTH}(C) \leq \Omega \quad C' = \text{ORACLE}(C) \quad \text{COST}(C') < \text{COST}(C)}{P; C; S \xrightarrow{\Omega} P; C'; S} \text{ LOPT} \\
\\
\frac{G \in L_2 \quad L_1 \diamond \{G\} \quad L'_1 = L_1 \cup \{G\} \quad L'_2 = L_2 \setminus \{G\}}{P; \langle L_1, L_2 \rangle; S \xrightarrow{\Omega} P; \langle L'_1, L'_2 \rangle; S} \text{ SHIFTLLEFT}
\end{array}$$

Figure 6.3: Local optimization rewrite rules.

circuit C' . Figure 6.3 shows the rewriting rules LOPT and SHIFTLLEFT for local optimization and compaction, respectively.

Local Optimization Rule. The rule LOPT (Figure 6.3) performs one local optimization on a circuit. It takes a segment C of length Ω , feeds it to the ORACLE, and retrieves the output segment C' . If the cost of C' is lower than the cost of C , then the rule replaces the segment C with C' . The rule does not modify the remaining parts P (prefix) and S (suffix) of the circuit.

Compaction Rule. To compact the circuit, we have the rule SHIFTLLEFT (Figure 6.3). In one step, the rule shifts a gate to the “left” by removing it from its current layer and adding it to the preceding layer. A circuit is compact whenever all gates have been fully shifted left. Formally, the SHIFTLLEFT rule considers consecutive layers L_1 and L_2 and moves a gate G from layer L_2 into L_1 , creating new layers $L'_1 = L_1 \cup \{G\}$ and $L'_2 = L_2 \setminus \{G\}$. To maintain well-formedness, the rule checks that no gate in the previous layer operates on the same qubits ($L_1 \diamond \{G\}$).

Example. Figure 6.4 illustrates how our rewriting rules can optimize a circuit with $\Omega = 2$. The dotted lines in the circuit separate the four layers. The optimizations in the figure implicitly use an oracle that performs the following actions: it removes two consecutive H gates on the same qubit because they cancel each other; similarly, it removes two consecutive X gates on the same qubit.

In the figure, moving from (a) to (b), the rule LOPT optimizes the 2-segment enclosed in the green box and removes two consecutive X gates on qubit q_0 . Then, going from (b) to (c), we apply the rule SHIFTLLEFT twice, moving the gate H gates together, and compacting the circuit. Because of this compaction, the rule LOPT can step from (c) to (d), canceling the two consecutive H gates.

6.2.4 Correspondence between local optimality and local rewrites

It is easy to show that our definition of local optimality is consistent with the rewriting semantics, in the sense that (1) locally optimal circuits cannot be further rewritten (Lemma 2), and (2) if a circuit is not locally optimal, then it always can be rewritten (Lemma 3).

Lemma 2. *For every circuit C where C **locally-optimal** $_{\Omega}$, there is no C' such that $C \xrightarrow{\Omega} C'$.*

Lemma 3. *For every circuit C , either C **locally-optimal** $_{\Omega}$ or $C \xrightarrow{\Omega} C'$.*

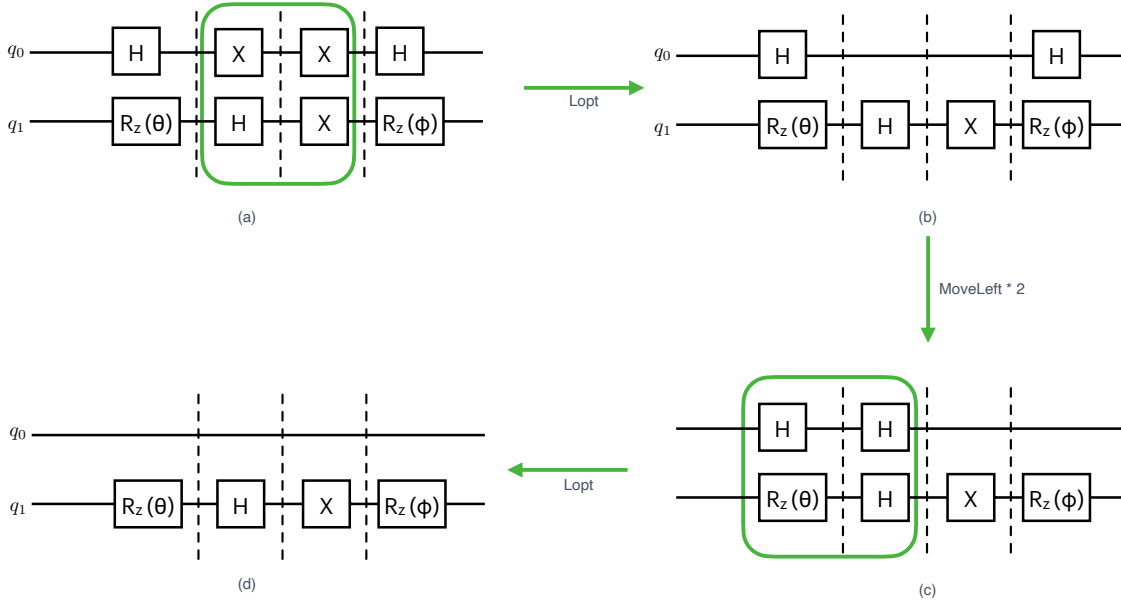


Figure 6.4: The figure illustrates how our rewriting semantics optimizes circuits for $\Omega = 2$. The figure implicitly assumes an oracle that removes any two consecutive H and X gates. At each step, our semantics either selects a segment of size 2 (denoted by green boxes) and performs an optimization, or picks a gate and shifts it left.

6.2.5 Termination

Given any circuit, we intend to use the rewriting semantics to produce an equivalent locally optimal circuit. However, this approach only succeeds if the rewriting semantics terminates, i.e., if eventually no further rewrites are possible. (Note that when our semantics terminates, Lemma 3 guarantees that the circuit is locally optimal.) Below are two examples where termination is not guaranteed.

1. **Infinite Decrease in Cost:** The cost function could be such that it decreases infinitely. For instance, consider the (contrived) cost function $\text{COST}(C) = \prod_{R_z(\theta) \in C} \theta$, which takes the product of the angles of gates $R_z(\theta)$ performing Z-axis rotations. Suppose we have a circuit containing a single gate $R_z(1)$, setting $\theta = 1$ and its cost is 1. We can replace the $R_z(1)$ gate by two consecutive $R_z(1/2)$ gates, which achieve the same rotation, but their combined cost is $1/4$. This process can repeat infinitely, halving the angle at each step, and the cost function would keep decreasing. For this circuit and cost function, our rewriting semantics does not terminate.
2. **Cycles Due to Cost Function:** A local optimization on one segment can increase the cost of a nearby segment, creating an infinite loop of local optimizations. To demonstrate this, consider the badly behaved cost function: $\text{COST}(C) = 0$ if $|C|$ is even, and otherwise $\text{COST}(C) = 1$. Now, “optimizing” one segment (to toggle its number of gates from odd to even) can cause a nearby overlapping segment to toggle from even to odd, potentially repeating forever.

Ensuring termination. We can prove that our semantics terminates under certain conditions on the cost function. First, we require $\text{COST}(C) \in \mathbb{N}$; this guarantees that the cost cannot decrease infinitely. Second, we require that the cost function is *additive* according to the following definition.

Definition 5. A function $\text{COST} : \text{Circuit} \rightarrow \mathbb{N}$ is additive iff both of the following conditions hold:

1. $\text{COST}(C_1; C_2) = \text{COST}(C_1) + \text{COST}(C_2)$ for all circuits C_1 and C_2 .
2. $\text{COST}(\langle L_1 \cup L_2 \rangle) = \text{COST}(\langle L_1 \rangle) + \text{COST}(\langle L_2 \rangle)$ for all layers L_1 and L_2 such that $L_1 \diamond L_2$.

Note that all cost functions that take a linear combination of counts of each gate in the circuit are additive. These include metrics such as gate count (number of gates), T count (number of T gates), $CNOT$ count (number of $CNOT$ gates), and two-qubit count (number of two-qubit gates).

For additive cost functions, we prove (Theorem 4) that the rewriting semantics always terminates.

Theorem 4 (Termination). Let $\text{COST} : \text{Circuit} \rightarrow \mathbb{N}$ be an additive cost function. For any initial circuit C_0 , there does not exist an infinite sequence of circuits such that

$$C_0 \xrightarrow{\Omega} C_1 \xrightarrow{\Omega} C_2 \xrightarrow{\Omega} \dots$$

where each $C_i \xrightarrow{\Omega} C_{i+1}$ represents a step of our rewriting semantics.

To prove Theorem 4, we define a “potential” function $\Phi : \text{Circuit} \rightarrow \mathbb{N} \times \mathbb{N}$ and show that each step of our rewriting semantics decreases the potential function, with the ordering $\Phi(C') < \Phi(C)$ defined lexicographically over $\mathbb{N} \times \mathbb{N}$. A suitable definition for Φ is as follows.

$$\Phi(C) \triangleq (\text{COST}(C), \text{INDEXSUM}(C)) \quad \text{where} \quad \begin{aligned} \text{INDEXSUM}(C) &\triangleq \sum_i i|L_i| \\ C &= \langle L_0, L_1, \dots \rangle \end{aligned}$$

This potential function has two components: (1) the cost of the circuit, which decreases as optimizations are performed, and (2) an “index sum”, which decreases as gates are shifted left. Together these components guarantee that the potential function decreases at each step, ensuring termination. Following is the full proof.

Proof. There are two cases for $C \xrightarrow{\Omega} C'$: either **LOPT** or **SHIFTLEFT**. In each case we show $\Phi(C') < \Phi(C)$.

In the case of **LOPT**, we have $C \xrightarrow{\Omega} C'$ where $C = (P; C''; S)$ and $C' = (P; \text{ORACLE}(C''); S)$. In C' , the segment C'' has been improved by one call to the **ORACLE**, i.e., $\text{COST}(\text{ORACLE}(C'')) < \text{COST}(C'')$. Because the **COST** function is additive, we have that

$$\text{COST}(C') = \text{COST}(P; \text{ORACLE}(C''); S) < \text{COST}(P; C''; S) = \text{COST}(C).$$

This in turn implies $\Phi(C') < \Phi(C)$ due to lexicographic ordering on the potential function.

In the case of **SHIFTLEFT**, we have $C \xrightarrow{\Omega} C'$ where $C = (P; \langle L_1, L_2 \rangle; S)$ and $C' = (P; \langle L'_1, L'_2 \rangle; S)$ and $L'_1 = L_1 \cup \{G\}$ and $L'_2 = L_2 \setminus \{G\}$. Because the cost function is additive, we have $\text{COST}(\langle L_1, L_2 \rangle) = \text{COST}(\langle L'_1, L'_2 \rangle)$ and therefore $\text{COST}(C) = \text{COST}(C')$. To show $\Phi(C') < \Phi(C)$, due to the lexicographic ordering, it remains to show $\text{INDEXSUM}(C') < \text{INDEXSUM}(C)$.

$\text{INDEXSUM}(C)$. This in turn follows from the definition of INDEXSUM ; in particular, plugging in $|L'_1| = |L_1| + 1$ and $|L'_2| = |L_2| - 1$ we get $\text{INDEXSUM}(C') = \text{INDEXSUM}(C) - 1$. Thus we have $\Phi(C') < \Phi(C)$. $\square$

6.3 Local Optimization Algorithm

In Section 6.2, we presented a rewriting semantics consisting of just two rules (corresponding to optimization of a segment and compaction) and proved that any saturating rewrite that applies the two rules to exhaustion yields a locally optimal circuit. This result immediately suggests an algorithm: simply apply the rewriting rules until they no longer may be applied, breaking ties between the two rules arbitrarily. Even though it might seem desirable due to its simplicity, such an algorithm is not efficient, because searching for a segment to optimize requires linear time in the size of the circuit (both in worst and the average case), yielding a quadratic bound for optimization. For improved efficiency, it is crucial to reduce the search time needed to find a segment that would benefit from optimization.

Our algorithm, called OAC, controls search time by using a circuit cutting-and-melding technique. The algorithm cuts the circuit hierarchically into smaller subcircuits, optimizes each subcircuit independently. The hierarchical cutting naturally reduces the search time for the optimizations by ensuring that most of the optimizations take place in the context of small circuits. Because the algorithm optimizes each subcircuit independently, it can miss crucial optimizations. To compensate for this, the algorithm melds the optimized subcircuits and optimizes further the melded subcircuits starting with the *seam*, or the boundary between the two subcircuits. The meld operation guarantees local optimality and does so efficiently by first optimizing the seam and further optimizing into each subcircuit only if necessary. By melding locally optimal subcircuits, the algorithm can guarantee that the subcircuits or any of their “untouched” portions (what it means to be “untouched” is relatively complex) remain optimal. We make this intuitive explanation precise by proving that the algorithm yields a locally optimal circuit. We note that circuit cutting techniques have been studied for the purposes of simulating quantum circuits on classical hardware [32, 106, 134]. We are not aware of prior work on circuit melding techniques that can lazily optimize across circuit cuts. Section 8.1.7 explains the difference between OAC and circuit cutting.

6.3.1 The algorithm

Figure 6.5 shows the pseudocode for our algorithm. The algorithm (OAC) organizes the computation into rounds, where each round corresponds to a recursive invocation of OAC. A round consists of a compaction phase (via the function `COMPACT`) and a segment-optimization phase, via the function `SEGOPT`. The rounds repeat until convergence, i.e., until no more optimization is possible (at which point the final circuit is guaranteed to be *locally optimal*). As the terminology suggests, the segment optimization phase always yields a segment-optimal circuit, where each and every segment is optimal (as defined by our rewriting semantics). Compaction rounds ensure that the algorithm does not miss optimization opportunities that arise due to compaction. We also present a relaxed version of our algorithm that stops early when a user-specified


```

1  $\Omega$ : int
2 ORACLE: circuit  $\rightarrow$  circuit
3 COST: circuit  $\rightarrow$  int
4 compact: circuit  $\rightarrow$  circuit
5
6 def OAC( $C$ ):
7    $C' = \text{segopt}(\text{compact}(C))$ 
8   if  $C' = C$ :
9     return  $C$ 
10  else:
11    return OAC( $C'$ )
12 def segopt( $C$ ):
13    $d = \text{length}(C)$ 
14   if  $d \leq 2\Omega$ :
15      $C' = \text{ORACLE}(C)$ 
16     if COST( $C'$ ) < COST( $C$ ):
17       return  $C'$ 
18     else:
19       return  $C$ 
20   else:
21      $m = \lfloor d/2 \rfloor$ 
22      $C_1 = C[0 : m]$ 
23      $C_2 = C[m : d]$ 
24      $C'_1 = \text{segopt}(C_1)$ 
25      $C'_2 = \text{segopt}(C_2)$ 
26     return meld( $C'_1, C'_2$ )
27 def meld( $C_1, C_2$ ):
28    $d_1 = \text{length}(C_1)$ 
29    $d_2 = \text{length}(C_2)$ 
30    $W = C_1[d_1 - \Omega : d_1] + C_2[0 : \Omega]$ 
31    $W' = \text{ORACLE}(W)$ 
32   if COST( $W'$ ) = COST( $W$ ):
33     return ( $C_1; C_2$ ) // concatenate
34   else:
35      $M = \text{meld}(C_1[0 : d_1 - \Omega], W')$ 
36     return meld( $M, C_2[\Omega : d_2]$ )

```

Figure 6.5: Algorithm OAC produces locally optimal circuits with respect to a given ORACLE, COST, and segment length Ω . To achieve local optimality, OAC only uses the oracle on small segments of length 2Ω . The algorithm repeatedly optimizes and compacts the circuit until convergence. The function SEG OPT() implements our optimization algorithm and uses MELD() to efficiently produce segment optimal circuits.

convergence threshold $0 \leq \epsilon \leq 1$, is reached.

Function SEG OPT. The function SEG OPT takes a circuit C and produces a segment optimal output. To achieve this, it uses a divide-and-conquer strategy to cut the circuit hierarchically into smaller and smaller circuits: it splits the circuit into the subcircuits C_1 and C_2 , optimizes each recursively, and then calls MELD on the resulting circuits to join them back together without losing segment optimality. This recursive splitting continues until the circuit has been partitioned into sufficiently small segments, specifically where each piece is at most 2Ω in length. For such small segments, the function directly uses the oracle and obtains optimal segments.

Function MELD. Figure 6.5 (right) presents the pseudocode of the MELD function. The function takes segment-optimal inputs C_1 and C_2 and returns a segment-optimal circuit that is functionally equivalent to the concatenation of the input circuits.

Given that the inputs C_1 and C_2 are segment optimal, all Ω -segments that lie completely within C_1 or C_2 are already optimal. Therefore, the function only considers and optimizes “boundary segments” which have some layers from circuit C_1 and other layers from circuit C_2 .

To optimize segments at the boundary, the function creates a “super segment”, named W , by concatenating the last Ω layers of circuit C_1 with the first Ω layers of circuit C_2 . The function denotes this concatenation as $C_1[d_1 - \Omega : d_1] + C_2[0 : \Omega]$ (see Section 6.3.1). The MELD function calls the oracle on W and retrieves the W' , which is guaranteed to be segment-optimal because it is returned by the oracle. The MELD function then considers the costs of W and W' .

If the costs of W and W' are identical, then W is already segment optimal. Consequently,

all Ω -segments at the boundary of C_1 and C_2 are also optimal. The key point is that the “super segment” W encompasses all possible Ω -segments at the boundary of C_1 and C_2 . To see this, let’s choose an Ω -segment at boundary, which takes the last $i > 0$ layers of circuit C_1 and the first $j > 0$ layers of circuit C_2 ; we can write this as $C_1[d_1 - i : d_1] + C_2[0 : j]$, where d_1 is the number of layers in C_1 . Given that this is an Ω -segment and has $i + j$ layers, we get that $i + j = \Omega$ and $i < \Omega$ and $j < \Omega$. Now observe that our chosen segment $C_1[d_1 - i : d_1] + C_2[0 : j]$ is contained within the super segment $W = C_1[d_1 - \Omega : d_1] + C_2[0 : \Omega]$ (Section 6.3.1), because $i < \Omega$ and $j < \Omega$. Given that W is segment optimal, our chosen segment is also optimal (relative to the oracle).

Returning to the MELD algorithm, consider the case where the segment W' improves upon the segment W . In this case, MELD incorporates W' into the circuit and propagates this change to the neighboring layers. To do this, MELD works with three segment optimal circuits: circuit $C_1[0 : d_1 - \Omega]$, which contains the first $d_1 - \Omega$ layers of circuit C_1 , is segment optimal because C_1 is segment optimal; the circuit W' is segment optimal because it was returned by the oracle; and the circuit $C_2[\Omega : d_2]$, which contains the last $d_2 - \Omega$ layers of circuit C_2 , is segment optimal because C_2 is segment optimal. Thus, we propagate the changes of window W' , by recursively melding these segment optimal circuits.

In Figure 6.5, the function MELD first melds the remaining layers of circuit C_1 with the segment W' , obtaining circuit M (see Section 6.3.1), and then melds the circuit M with the remaining layers of C_2 .

6.3.2 Meld Example

We present an example of how MELD joins two circuits by optimizing from the “seam” out, and does so “lazily”, as needed.

Figure 6.6 shows a three-step MELD operation that identifies optimizations at the boundary of two circuits. All the circuits in the figure are expressed using the H gate (Hadamard gate), the R_z gate (rotation around Z-axis), and the two-qubit $CNOT$ gate (Controlled Not gate), which is represented using a dot and an XOR symbol. We first provide background on the optimizations used by MELD and label them “Optimization 1” and “Optimization 2”. Optimization 1 shows that when a $CNOT$ gate is surrounded by four H gates, all of these gates can be replaced by a single $CNOT$ gate whose qubits are flipped. Optimization 2 shows that when two $CNOT$ gates are separated by an R_z gate as shown, they may be removed.

The steps in the figure describe a MELD operation on the two circuits separated by a dashed line, which represents their seam. To join the circuits, the MELD operation proceeds outwards in both directions and optimizes the boundary segment, represented as a green box with solid lines. The MELD applies Optimization 1 to the green segment and this introduces a flipped $CNOT$ gate.

The MELD propagates this change in step 2, by considering a new segment, which includes a neighboring layer. We represent this segment by a purple box with dotted lines, and it contains two $CNOT$ gates, one of which was introduced by the first optimization. The MELD then applies Optimization 2, removing the $CNOT$ gates and bringing the two rotation gates next to each other. Note that Optimization 2 became possible only because of Optimization 1, which introduced the flipped $CNOT$ gate. In the final step, the MELD considers the segment represented

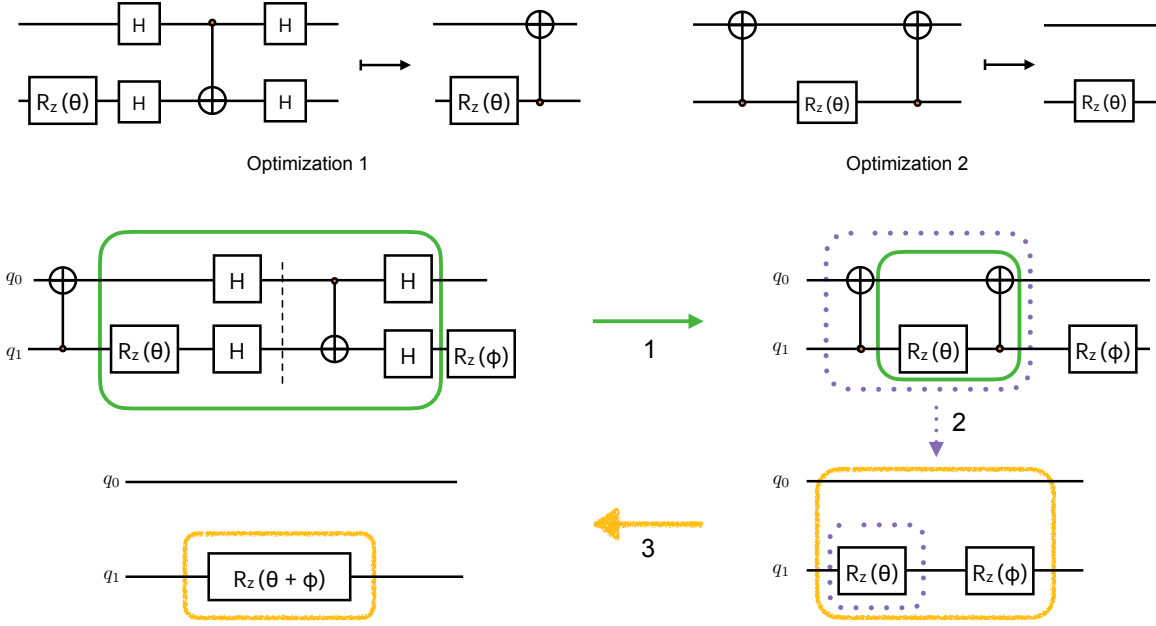


Figure 6.6: The figure shows a three-step MELD operation and illustrates how it propagates optimizations at the boundary of two optimal circuits. At the top, the figure shows specific optimizations “1” and “2”, and towards the bottom, the figure shows the optimization steps of MELD. Before the first step, two individual circuit segments that are optimal are separated by a dashed line. Each MELD step considers a segment, represented by a box with solid/dotted/shaded lines, and applies an optimization to it, reducing the gate count. The first step focuses on the boundary segment within the solid green box, overlapping with both circuits, and applies “Optimization 1”. This step introduces a flipped $CNOT$ gate, which interacts with a neighboring $CNOT$ gate and triggers “Optimization 2” in the purple dotted box. The third step merges two neighboring rotation gates in the yellow shaded box.

by a yellow box with shaded lines and performs a third optimization, merging the two rotation gates. Overall, this sequence of optimizations, at the boundary of two circuits, reduces the gate count by seven.

6.3.3 Correctness and Efficiency

Because our algorithm cuts the input circuit into subcircuits and optimizes them independently, it is far from obvious that its output is segment optimal. We prove that this is indeed the case with Theorem 5 below. The reason for this is the MELD operation that is able to optimize circuit cuts. We also prove, with Corollary 1, that even though MELD behaves dynamically and its cost varies from one circuit to another, it remains efficient, in the sense that the number of calls to oracle is always linear in the size of the circuit plus the improvement in the cost.

Lemma 4 (Segment optimality of MELD). *Given any additive COST function and any segment optimal circuits C_1 and C_2 , the result of $MELD(C_1, C_2)$ is a segment optimal circuit C and*

$$\text{COST}(C) \leq \text{COST}(C_1) + \text{COST}(C_2).$$

Proof. We show the lemma by induction on the total cost of the input, i.e., $\text{COST}(C_1) + \text{COST}(C_2)$. In the base case, the input cost is zero, and therefore all segments of the input circuits have zero cost, due to additivity of the COST function. The output of MELD in this case is the concatenation $(C_1; C_2)$, because the oracle can not improve the boundary segment W . For the inductive step, we consider two cases, depending on whether the oracle optimizes the boundary segment W .

Case I When the oracle finds no improvement in W , the output is the concatenation $C = (C_1; C_2)$. To prove that the output C is segment optimal, we show that every Ω -segment of the output is optimal relative to the oracle. For any such Ω -segment X , there are two cases.

- If X is a segment of either C_1 or C_2 , then X is optimal relative to the oracle because the input circuits are segment optimal and X is of length Ω .
- If X is not a segment of either C_1 or C_2 , then X is a sub-segment of W , because the segment W has length 2Ω and contains all possible Ω -segments at the boundary of C_1 and C_2 . Because the oracle could not improve segment W , it can not improve its sub-segment X .

Thus, all Ω -segments of the output are optimal relative to the oracle, and the output is segment optimal.

Case II If the oracle improves the boundary segment W , then the cost of the optimized segment W' is less than cost of segment W , i.e., $\text{COST}(W') < \text{COST}(W)$. In this case, the function first melds the segment $C_1[0 : d_1 - \Omega]$ and the segment W' , by making a recursive call. We apply induction on the costs of the inputs to prove that the output circuit M is segment optimal. Formally, we have the cost of inputs $\text{COST}(C_1[0 : d_1 - \Omega]) + \text{COST}(W') < \text{COST}(C_1[0 : d_1 - \Omega]) + \text{COST}(W) \leq \text{COST}(C_1) + \text{COST}(C_2)$, because W' has a lower cost than W . Therefore, by induction, the output M is segment optimal and $\text{COST}(M) \leq \text{COST}(C_1[0 : d_1 - \Omega]) + \text{COST}(W')$.

For the second recursive call $\text{MELD}(M, C_2[\Omega : d_2])$, we can apply induction because $\text{COST}(M) + \text{COST}(C_2[\Omega : d_2]) < \text{COST}(C_1) + \text{COST}(C_2)$. This inequality follows from the cost bound on circuit M above and using that W' has a lower cost than W . Specifically, $\text{COST}(M) + \text{COST}(C_2[\Omega : d_2]) \leq \text{COST}(C_1[0 : d_1 - \Omega]) + \text{COST}(W') + \text{COST}(C_2[\Omega : d_2])$, which is strictly less than $\text{COST}(C_1[0 : d_1 - \Omega]) + \text{COST}(W) + \text{COST}(C_2[\Omega : d_2]) = \text{COST}(C_1) + \text{COST}(C_2)$. Therefore, by induction on the second recursive call, we get that MELD returns a segment optimal circuit that is bounded in cost by $\text{COST}(C_1) + \text{COST}(C_2)$. Note that for both recursive calls, the induction relies on the fact that W' has a lower cost than W , showing that the algorithm works because the recursion is tied to cost improvement. $\square$

Theorem 5 (Segment optimality algorithm). *For any circuit C , the function $\text{SEGOPT}(C)$ outputs a segment optimal circuit.*

Proof. We prove the theorem by induction on $d = \text{LENGTH}(C)$. In the base case, $d \leq 2\Omega$ and the circuit is fed to the oracle, thereby guaranteeing segment optimality. For the inductive case, the algorithm splits the circuit into two smaller circuits C_1 and C_2 , and recursively optimizes them

to obtain segment optimal outputs C'_1 and C'_2 . The result $C = \text{MELD}(C'_1, C'_2)$ is then segment optimal, by Lemma 4. $\square$

Having shown that the functions `SEGOPT` and `MELD` produce segment optimal outputs, we now analyze the runtime efficiency of these functions. The runtime efficiency of both functions is data-dependent and varies with the number of optimizations found throughout the circuit. To account for this, we charge the running time to the cost improvement between the input and the output, represented by Δ . We prove the following theorem.

Theorem 6 (Efficiency of segment optimization). *The function $\text{SEGOPT}(C)$ calls the oracle at most $\text{LENGTH}(C) + 2\Delta$ times on segments of length at most 2Ω , where Δ is the improvement in the cost of the output.*

The theorem shows that our segment optimality algorithm is productive in its use of the oracle. Consider the terms, $\text{LENGTH}(C)$ and 2Δ , in the bound on the number of oracle calls: The first term, $\text{LENGTH}(C)$, is for checking segment optimality—even if the input circuit is already optimal, the algorithm needs to call the oracle on each segment and confirm it. The second term 2Δ , shows that all further calls result from optimizations, with each optimization requiring up to two oracle calls in the worst case. This shows that our algorithm (alongside `MELD`) carefully tracks the segments on which the oracle needs to be called and avoids unnecessary calls.

The theorem also highlights one of the key features of our `SEGOPT` function: it only uses the oracle on small circuit segments of length 2Ω . Suppose q is the number of qubits in the circuit. Then the `SEGOPT` function only calls the oracle on manageable circuits of size less than or equal to $q * 2\Omega$, which is significantly smaller than circuit size. This has performance impacts because, for many oracles, it is faster to invoke the oracle many times on small circuits rather than invoking the oracle a single time on the full circuit (see Section 5.4).

Since we bound number of oracle calls in terms of the cost improvement Δ , an interesting question is how large can Δ be? When optimizing for gate-counting metrics such as T count (number of T gates), $CNOT$ count (number of $CNOT$ gates), gate count (total number of gates), the cost improvement is trivially bounded by the circuit size, i.e., $\Delta \leq |C|$. This observation shows that our `SEGOPT` function only makes a linear number of calls (with depth and gate count) to the oracle.

Corollary 1 (Linear calls to the oracle). *When optimizing for gate count, our $\text{SEGOPT}(C)$ makes a linear, $O(\text{LENGTH}(C) + |C|)$, number of calls to the oracle.*

We experimentally validate this corollary in Section 6.5.4, where we study the number of oracle calls made by our algorithm for many circuits. The crux of the proofs of Theorem 6 and the corollary is bounding the number of calls in the `MELD` function, because it can propagate optimizations through the circuit. We show the proof below.

Lemma 5 (Bounded calls to oracle in `MELD`). *Computing $C = \text{MELD}(C_1, C_2)$ makes at most $1 + 2\Delta$ calls to the oracle, where Δ is the improvement in cost, i.e., $\Delta = \text{COST}(C_1) + \text{COST}(C_2) - \text{COST}(C)$.*

Proof. We prove this by induction on the input cost $\text{COST}(C_1) + \text{COST}(C_2)$. The base case is straightforward: when the input cost is zero, the `MELD` function only makes the one call and returns. For the inductive step, we consider two cases depending on whether the oracle improves the boundary segment W .

Case I If the oracle finds no improvement in the segment W , then there is exactly one call to the oracle. Thus we have $1 \leq 1 + 2(\text{COST}(C_1) + \text{COST}(C_2) - \text{COST}(C))$.

Case II When the oracle finds an improvement in the segment W , it returns W' such that $\text{COST}(W') \leq \text{COST}(W) - 1$. (There is a difference of at least 1 because $\text{COST}(-) \in \mathbb{N}$.) In this case, the MELD function recurs twice, first to compute $M = \text{MELD}(C_1[0 : d_1 - \Omega], W')$, and then to compute the output $C = \text{MELD}(M, C_2[\Omega : d_2])$. The total number of calls to the oracle can be decomposed as follows:

- 1 call to the oracle for the segment W .
- Inductively, at most $1 + 2(\text{COST}(C_1[0 : d_1 - \Omega]) + \text{COST}(W') - \text{COST}(M))$ calls to the oracle for the first MELD.
- Inductively, at most $1 + 2(\text{COST}(M) + \text{COST}(C_2[\Omega : d_2]) - \text{COST}(C))$ calls to the oracle in the second recursive MELD.

Adding these up yields at most $1 + 2(\text{COST}(C_1) + \text{COST}(C_2) - \text{COST}(C))$ calls to the oracle, as desired. (We use here the facts $\text{COST}(W') \leq \text{COST}(W) - 1$ and $\text{COST}(C_1[0 : d_1 - \Omega]) + \text{COST}(W) + \text{COST}(C_2[\Omega : d_2]) = \text{COST}(C_1) + \text{COST}(C_2)$.) $\square$

The OAC algorithm uses the segment optimality guarantee, given by SEGOPT, to produce a locally optimal circuit. In Figure 6.5, the function OAC takes the input circuit C and computes the circuit $C' = \text{SEGOPT}(\text{COMPACT}(C))$. It then checks if C' differs from the input C and if so, it recurses on C' . The function repeats this until convergence, i.e., until the circuit does not change. In the process, it computes a sequence of segment optimal circuits $\{C_i\}_{1 \leq i \leq \kappa}$, where κ is the number of rounds until convergence:

$$C \rightarrow C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_\kappa \quad \text{where} \quad \begin{array}{l} \text{COST}(C_i) > \text{COST}(C_{i+1}) \\ C_i \text{ segment-optimal}_\Omega \\ C_\kappa \text{ locally-optimal}_\Omega \end{array}$$

Each intermediate circuit C_i is segment optimal and gets compacted before optimization in the next round. Given that OAC continues until convergence, the final circuit, C_κ , is locally optimal.

Using Theorem 6, we observe that the number of oracle calls in each round is bounded by $\text{LENGTH}(C_i) + 2\Delta_i$, where Δ_i is the improvement in cost (i.e., $\Delta_i = \text{COST}(C_i) - \text{COST}(C_{i+1})$). Thus, the total number of oracle calls performed by OAC can be bounded by $O(\Delta + \text{LENGTH}(C) + \sum_{1 \leq i \leq \kappa} \text{LENGTH}(C_i))$ where $\Delta = \text{COST}(C) - \text{COST}(C_\kappa)$ is the overall improvement in cost between the input and output.

Theorem 7. *Given an additive cost function, the output of $\text{OAC}(C)$ is locally optimal and requires $O(\Delta + \text{LENGTH}(C) + \sum_{1 \leq i \leq \kappa} \text{LENGTH}(C_i))$ oracle calls, where C_i denotes the circuit after i rounds, κ is the number of rounds, and Δ is the end-to-end cost improvement.*

The bound on the number of oracle calls in Theorem 7 is very general because it applies to any oracle (and gate set). For particular cost functions, we can use it to deduce a more precise bound. Specifically, in the case of gate count, where $\text{COST}(C) = |C|$, we get the following bound.

Corollary 2. *When optimizing for gate count, $OAC(C)$ performs at most $O(|C| + \kappa \cdot \text{LENGTH}(C))$ oracle calls.*

An interesting question is whether or not it is possible to bound the number of rounds, κ . In general, this will depend on a number of factors, such as the quality of the oracle and how quickly the optimizations it performs converge across compactions. In practice (Section 5.4), we find that the number of rounds required is small, and that nearly all optimizations are performed in the first round itself. For gate count optimizations in particular, Corollary 2 tells us that when only a small number of rounds are required, the scalability of OAC is effectively linear in circuit size.

```
def OAC*( $\epsilon$ ,  $C$ ):
     $C' = \text{segopt}(\text{compact}(C))$ 
    if  $1 - \frac{\text{cost}(C')}{\text{cost}(C)} \leq \epsilon$ :
        return  $C'$ 
    else:
        return OAC*( $\epsilon$ ,  $C'$ )
```

Figure 6.7: OAC^* terminates when the cost improvement ratio falls below the *convergence threshold*, ϵ .

6.3.4 OAC^* : controlling convergence

We observe that, except for the very last round, each round of our OAC algorithm improves the cost of the circuit. This raises a practical question: how does the improvement in cost vary across rounds? For the vast majority of our evaluation, we observe that nearly all optimization ($> 99\%$) occur in the first round itself (see Section 6.5.5); the subsequent rounds have a small impact on the quality. Based on this observation, we propose OAC^* which uses a convergence threshold $0 \leq \epsilon \leq 1$ to provide control over how quickly the algorithm converges.

The OAC^* algorithm, in Figure 6.7, terminates as soon as the cost is reduced by a smaller fraction than ϵ . For example, if we measure cost as the number of gates and $\epsilon = 0.01$, then OAC^* will terminate as soon as an optimization round removes fewer than 1% of the remaining gates. Note that OAC^* gives two guarantees: 1) the output circuit is segment optimal, and 2) the fractional cost improvement in the last round is less than ϵ . In addition, setting $\epsilon = 0$ results in identical behavior to OAC and guarantees local optimality.

6.4 Implementation

To evaluate whether the OAC algorithm (Section 6.3) is practically feasible, we implemented OAC in SML (Standard ML), which comes with an optimizing compiler, MLton, that can generate fast executables. Our implementation is publicly available as an open-source project [12], and it closely follows the algorithm description. It uses the layered circuit representation and represents circuits as an array of arrays, where each array denotes a “layer” of the circuit. The

implementation splits and joins circuit segments by splitting and joining the corresponding arrays, performing rounds of optimization and compaction.

As described in Section 6.3.4, our implementation allows user control over convergence through a specified convergence ratio ϵ , where $0 \leq \epsilon \leq 1$. In the evaluation, we choose $\epsilon = 0.01$, and analyze this choice in Section 6.5.5. Note that regardless of ϵ , the implementation always guarantees that output is segment optimal.

Our implementation is parametric in the oracle being used. To allow calls to existing optimizers, we use MLton’s foreign function interface, which supports cross-language calls to C++. Specifically, to use an existing optimizer as an oracle, we only need to provide a C++ wrapper that takes a circuit in QASM format as input and returns an optimized circuit as output.

6.4.1 Circuit Representation and Compaction

Our algorithm represents circuits using a hybrid data structure that switches between sequences and linked lists for splitting and concatenating circuits in $O(1)$ time. Initially, it represents the circuit as a sequence of layers, enabling circuit splits in $O(1)$ time. Later during optimization when circuit concatenation is required, the algorithm switches to a linked list of layers, enabling $O(1)$ concatenation. At the end of each optimization round, the algorithm uses a function $\text{COMPACT}(C)$ to revert to the sequence of layers representation.

The function $\text{COMPACT}(C)$ ensures that the resulting circuit is equivalent to C while eliminating any unnecessary “gaps” in the layering, meaning that every gate has been shifted left as far as possible. For brevity, we omit the implementation of the COMPACT function from Figure 6.5. Our implementation in practice is straightforward: we use a single left-to-right pass over the input circuit and build the output by iteratively adding gates. The time complexity is linear, i.e., $\text{COMPACT}(C)$ requires $O(|C| + \text{LENGTH}(C))$ time.

6.5 Evaluation

We perform an empirical evaluation of the effectiveness of the local optimality approach for quantum circuits. Specifically, we consider the following research questions (RQ).

RQ I: Is the OAC algorithm efficient and scalable?

RQ II: Is there any degradation in optimization quality? How important is the MELD operation?

RQ III: Is empirical performance consistent with the asymptotic bound?

RQ IV: What is the impact of segment size Ω and compaction on the local optimality and performance of OAC algorithm?

To answer these questions, we implement the OAC algorithm and integrate it with VOQC [61] as an oracle. We chose VOQC because it is overall the best optimizer both in terms of efficiency and quality of optimization among all the optimizers that we have experimented with. We evaluate the effectiveness of local optimality and OAC, on the Nam gate set [97] and compare it with three state-of-the-art optimizers Quartz, QUESO, and VOQC [61, 144, 149].

In brief, these experiments show that our cut-and-meld algorithm delivers fast optimization while closely matching (within 0.1%) or improving the optimization quality for all circuits.

These results show that the local optimality approach can be effective in optimizing large quantum circuits, and can help scale existing optimizers.

In Section 6.6, we also present results for the Clifford+T gate set by using the FeynOpt [9], as an oracle. The results are similar to this section in terms of efficiency and quality.

6.5.1 Benchmarks and gate set

To evaluate our OAC algorithm, we consider a benchmark suite of eight circuit families that include both near-term and future fault-tolerant quantum algorithms. For each family, we select circuits with different sizes by changing the number of qubits. Our benchmark suite includes advanced quantum algorithm such as Grover’s algorithm for unstructured search [57], the HHL algorithm for solving linear systems of equations [59], Shor’s algorithm for factoring large integers [128], and the Binary Welded Tree (bwt) quantum walk algorithm [37]. In addition, our benchmarks include near-term algorithms like Variational Quantum Eigensolver (vqe) [108] and reversible arithmetic algorithms [11, 97] such as boolean satisfaction problems (boolsat) and square-root algorithm (sqrt).

The benchmark suite is written in the Nam gate set [97], which consists of the Hadamard (H), Pauli-X (X), controlled-NOT ($CNOT$), and Z-rotation (R_z) gates [97]. We preprocess all our benchmarks with the Quartz preprocessor, which merges rotation gates [149].

6.5.2 RQ I: Efficiency of OAC

To evaluate the efficiency of OAC, we use our OAC implementation with VOQC as the oracle on segments of size $\Omega = 40$ and compare it to optimizers Quartz, QUESO, and VOQC. The approach works for many different settings of Ω and we analyze the impact of Ω in Section 6.5.5 in detail. We give each optimizer a 12-hour cut-off time (excluding time for parsing and printing), to allow completion of the experiments within a reasonable amount of time. Throughout, we omit circuit-parsing time for timings of VOQC, whose parser appears to scale superlinearly and can take significant time (sometimes more than the optimization itself). This approach is consistent with prior work on VOQC, which also excludes parse time. When we use VOQC as an oracle of OAC, however, we do include the parse time. This makes the comparison somewhat unfair for OAC.

We evaluate the running times of these optimizers on benchmarks from the Nam gate set with sizes ranging from thousands to hundreds of thousands of gates.

Time Performance. Table 6.1 show the time for our OAC implementation (with VOQC oracle) compared against Quartz, QUESO, and VOQC. The figure includes eight families of circuits, where horizontal lines separate families and circuits within each family arranged body increasing qubit and gate counts. The optimizers Quartz and QUESO use the maximum allotted time of 12 hours in all circuits, because they explore a very large search space of all optimizations. In a few cases, QUESO throws an error or runs out of memory (denoted “OOM”). The VOQC optimizer and our OAC optimizer terminate much faster. Specifically, OAC optimizes all circuits between 0.2 seconds and 3 hours depending on the size, and VOQC finishes for all but six benchmarks within 12 hours. In the figure, we highlight in bold the fastest optimizer(s) for each circuit.

Table 6.1: The table shows the running time in seconds of the four optimizers, using gate count as the cost metric. The column “OAC Speedup” is the speed of our OAC with respect to VOQC, calculated as VOQC time divided by OAC time.

Benchmark	Qubits	Input Size	Time (s)				OAC speedup
			Quartz	Queso	VOQC	OAC	
boolsat	28	75670	12h	12h	68.6	45.2	1.52
	30	138293	12h	12h	307.3	98.7	3.11
	32	262548	12h	12h	1266.2	213.6	5.93
	34	509907	12h	12h	6151.0	462.8	13.29
bwt	17	262514	12h	12h	8303.1	524.9	15.82
	21	402022	12h	12h	23236.8	1062.1	21.88
	25	687356	12h	12h	>12h	2341.7	> 18.45
	29	941438	12h	12h	>12h	3982.6	> 10.85
grover	9	8968	12h	12h	9.3	4.8	1.94
	11	27136	12h	12h	106.5	20.8	5.12
	13	72646	12h	12h	815.7	68.1	11.97
	15	180497	12h	12h	5743.2	223.9	25.65
hhl	7	5319	12h	12h	0.3	0.9	0.27
	9	63392	12h	12h	74.1	22.9	3.24
	11	629247	12h	12h	14868.8	434.3	34.24
	13	5522186	12h	Parsing Error	>12h	8243.1	> 5.24
shor	10	8476	12h	OOM	8.8	5.2	1.70
	12	34084	12h	12h	179.9	26.3	6.84
	14	136320	12h	12h	3638.4	126.0	28.88
	16	545008	12h	12h	70475.2	648.9	108.60
sqrt	42	79574	12h	12h	30.0	81.4	0.37
	48	186101	12h	12h	191.2	268.0	0.71
	54	424994	12h	12h	3946.5	679.8	5.81
	60	895253	12h	12h	>12h	1653.5	> 26.13
statevec	5	31000	12h	OOM	1.6	4.3	0.38
	6	129827	12h	12h	45.9	27.1	1.70
	7	526541	12h	12h	1812.2	164.7	11.00
	8	2175747	12h	12h	>12h	1345.1	> 32.12
vqe	12	11022	12h	12h	0.2	1.2	0.13
	16	22374	12h	12h	0.6	3.4	0.18
	20	38462	12h	12h	2.0	7.0	0.29
	24	59798	12h	12h	5.4	13.4	0.41
avg							> 12.62

Table 6.2: Optimization Quality. The table shows the percentage reduction in gate count achieved by the four optimizers. We write “N.A.” in cases where the optimizer did not finish within the allotted 12-hour deadline. The experiments show that our cut-and-meld optimizer OAC optimizes slightly better on average than all other optimizers.

Benchmark	Qubits	Input Size	Optimizer			
			Quartz	Queso	VOQC	OAC
boolsat	28	75670	-41.1%	-30.4%	-83.2%	-83.7%
	30	138293	-23.5%	-30.2%	-83.3%	-83.7%
	32	262548	-6.9%	0.0%	-83.3%	-83.5%
	34	509907	-2.9%	0.0%	-83.3%	-83.4%
bwt	17	262514	-8.7%	-0.1%	-30.0%	-31.1%
	21	402022	-3.1%	-0.1%	-38.4%	-40.0%
	25	687356	-0.8%	0.0%	N.A.	-43.8%
	29	941438	-0.4%	0.0%	N.A.	-44.5%
grover	9	8968	-9.4%	-13.7%	-29.4%	-29.4%
	11	27136	-9.5%	-9.6%	-29.9%	-30.0%
	13	72646	-9.6%	-0.3%	-29.7%	-29.7%
	15	180497	-9.6%	-0.2%	-29.5%	-29.5%
hhl	7	5319	-26.6%	-28.7%	-55.4%	-55.3%
	9	63392	-24.4%	-23.8%	-56.3%	-56.5%
	11	629247	-1.1%	0.0%	-53.7%	-53.7%
	13	5522186	0.0%	N.A.	N.A.	-52.6%
shor	10	8476	0.0%	-5.4%	-11.1%	-11.0%
	12	34084	0.0%	-3.8%	-11.2%	-11.2%
	14	136320	0.0%	0.0%	-11.3%	-11.2%
	16	545008	0.0%	0.0%	-11.3%	-11.3%
sqrt	42	79574	-16.2%	-0.1%	-33.0%	-33.0%
	48	186101	-15.2%	0.0%	-32.7%	-32.6%
	54	424994	-5.1%	0.0%	-32.4%	-32.3%
	60	895253	-2.1%	0.0%	N.A.	-34.3%
statevec	5	31000	-48.4%	-74.5%	-78.8%	-78.9%
	6	129827	-35.0%	-29.7%	-78.4%	-78.4%
	7	526541	-2.0%	-29.7%	-78.1%	-78.1%
	8	2175747	-0.1%	0.0%	N.A.	-78.7%
vqe	12	11022	-35.4%	-69.1%	-63.0%	-69.5%
	16	22374	-33.7%	-64.2%	-60.1%	-66.3%
	20	38462	-32.2%	-60.8%	-57.4%	-63.4%
	24	59798	-30.8%	-36.4%	-54.9%	-60.6%
avg			-13.6%	-16.5%	-48.1%	-49.4%

Comparing between VOQC and our OAC, we observe the following:

- **Performance:** OAC is the fastest across the board except for `vqe` and except perhaps for the smallest circuits in some families.
- **Scalability:** the gap between OAC and VOQC increases as the circuit size increases, with OAC performing as much as $100\times$ faster in some cases.
- **Overall:** OAC is over an order of magnitude faster than VOQC on average.

In the case of the `vqe` family, VOQC is consistently faster, but as we discuss next, this comes at the cost of poorer optimization quality. For the small circuits of families `hhl`, `statevec`, and `sqr`, our optimizer is slower than VOQC. This is due to the overheads that our implementation incurs for (1) splitting and joining circuits, (2) serialization/deserialization of input/output circuits for each oracle call, and (3) various system-level calls needed to support calls to an external oracle. For example, for the 7-qubit `hhl` benchmark and the 42-qubit `sqr` benchmark, we have measured that at least 30% of the running time is spent parsing and serializing/deserializing circuits.

6.5.3 RQ II: Optimization quality and importance of MELD

We use VOQC as the oracle because it scales better and achieves a better optimization quality than all other previous works on all circuits presented in Table 6.1. Table 6.3 shows the output quality (measured by gate count) of VOQC and OAC for eight families of circuits.

For almost all families, we observe that the output quality of OAC matches that of VOQC within 0.1% or improves it, sometimes significantly. This shows that OAC scales better without sacrificing optimization quality. Specifically, for the `vqe` family, OAC optimizes better than VOQC. For example, on the 24-qubit `vqe` circuit, OAC improves the gate count by 60.6%, and VOQC improves the gate count by 54.9%.

To better understand what contributes to the optimization quality, we perform an ablation experiment of the MELD operation. Specifically, we implement an ablating version of our OAC, called OACMinus, that simply concatenates the optimized subcircuits instead of the MELD operation.

Table 6.3 shows that the MELD operation contributes an improvement of 2.1% in absolute and 4.4% in relative terms. Even though the percentage degradation due to ablation may seem modest, it is significant, because

- each and every gate has a significant runtime and fidelity cost on modern and near-term quantum computers, and
- the optimization quality does not decrease uniformly: in some circuit families, such as `vqe`, the ablating version is more than 5% worse in absolute and 12% in relative terms.

Due to these differences the ablating version (without the MELD operation) performs consistently worse than VOQC, whereas the non-ablating version (OAC) outperforms VOQC in all but 5 of the 32 circuits. In the 5 circuits, where VOQC outperforms our local-optimization, the difference is 0.1%.

Table 6.3: Ablation study of MELD. The table shows percentage reduction in gate count achieved by the version of OAC, called OACMinus, that uses simply concatenation instead of the MELD algorithm, compared with VOQC and the original OAC. The measurements show that with the MELD algorithm, our OAC optimizer reduces gate counts better than using VOQC alone or simply splitting the circuit into segments and optimizing them using VOQC separately.

Benchmark	Qubits	Input Size	Optimizer		
			VOQC	OAC	OACMinus
boolsat	28	75670	-83.2%	-83.7%	-83.0%
	30	138293	-83.3%	-83.7%	-83.1%
	32	262548	-83.3%	-83.5%	-83.1%
	34	509907	-83.3%	-83.4%	-83.1%
bwt	17	262514	-30.0%	-31.1%	-28.3%
	21	402022	-38.4%	-40.0%	-36.3%
	25	687356	N.A.	-43.8%	-40.0%
	29	941438	N.A.	-44.5%	-41.0%
grover	9	8968	-29.4%	-29.4%	-26.1%
	11	27136	-29.9%	-30.0%	-26.3%
	13	72646	-29.7%	-29.7%	-26.0%
	15	180497	-29.5%	-29.5%	-26.0%
hhl	7	5319	-55.4%	-55.3%	-54.4%
	9	63392	-56.3%	-56.5%	-55.6%
	11	629247	-53.7%	-53.7%	-53.1%
	13	5522186	N.A.	-52.6%	-52.1%
shor	10	8476	-11.1%	-11.0%	-10.3%
	12	34084	-11.2%	-11.2%	-10.7%
	14	136320	-11.3%	-11.2%	-10.8%
	16	545008	-11.3%	-11.3%	-10.9%
sqrt	42	79574	-33.0%	-33.0%	-31.5%
	48	186101	-32.7%	-32.6%	-31.2%
	54	424994	-32.4%	-32.3%	-30.9%
	60	895253	N.A.	-34.3%	-32.9%
statevec	5	31000	-78.8%	-78.9%	-78.1%
	6	129827	-78.4%	-78.4%	-77.9%
	7	526541	-78.1%	-78.1%	-77.8%
	8	2175747	N.A.	-78.7%	-78.6%
vqe	12	11022	-63.0%	-69.5%	-62.7%
	16	22374	-60.1%	-66.3%	-59.7%
	20	38462	-57.4%	-63.4%	-57.2%
	24	59798	-54.9%	-60.6%	-54.8%
avg			-48.1%	-49.4%	-47.3%

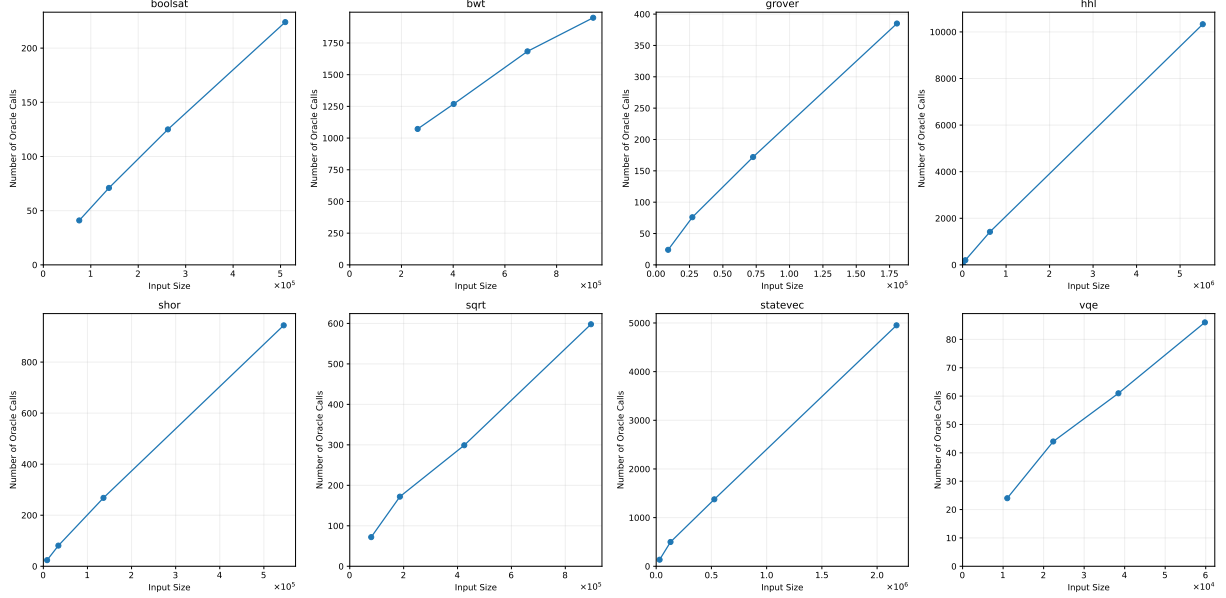


Figure 6.8: The number of oracle calls versus input circuit size for all our circuits (Nam gate set). The plots show that the number of calls scales linearly with the number of gates.

6.5.4 RQ III: Empirical versus asymptotic performance

In Section 6.3, we establish bounds on the number of oracle calls performed by our OAC algorithm. In this section, we check that our implementation is consistent with these bounds by analyzing the number of oracle calls with respect to the circuit size. Figure 6.8 plots the number of oracle calls made by our OAC optimizer. The Y-axis represents the number of oracle calls and the X-axis represents the input circuit size. The plot shows that the number of oracle calls increases linearly with circuit size, for all circuit families.

We note that it would be more desirable to establish that the total run-time, rather than the number of oracle calls, is linear, but this is not the case because the oracle optimizers can take asymptotically non-linear time. For example, the oracle VOQC can require at least quadratic time in the number of gates in the circuit being optimized, which varies as we increase the qubit counts.

6.5.5 RQ IV: Impact of compaction and Ω

Impact of compaction

Table 6.4 shows the number of rounds and percentage optimizations for each round of our OAC algorithm, using the convergence ratio $\epsilon = 0.01$. The results show that OAC converges very quickly, always terminating after 2 rounds of optimization, and that it consistently finds over 99% of the optimizations in the first round. This is because our OAC ensures the slightly weaker segment optimality after the first round of optimization (see Section 6.3), which ensures that all segments are optimal, though there may be gaps. The experiment shows that although compaction can enable some optimizations by removing the gaps, its impact on these benchmarks is

Table 6.4: The number of rounds and the percentage of optimizations in each optimization round of OAC.

Family	Qubits	#Rounds	Optimizations	
			Round 1	Round 2
boolsat	28	2	100.00%	0.00%
	30	2	100.00%	0.00%
	32	2	100.00%	0.00%
	34	2	100.00%	0.00%
bwt	17	2	99.59%	0.41%
	21	2	99.79%	0.21%
	25	2	99.83%	0.17%
	29	2	99.90%	0.10%
grover	9	2	99.96%	0.04%
	11	2	99.90%	0.10%
	13	2	99.99%	0.01%
	15	2	99.97%	0.03%
hhl	7	2	99.76%	0.24%
	9	2	99.95%	0.05%
	11	2	99.96%	0.04%
	13	2	99.95%	0.05%
shor	10	2	100.00%	0.00%
	12	2	99.97%	0.03%
	14	2	99.96%	0.04%
	16	2	99.99%	0.01%
sqrt	42	2	99.90%	0.10%
	48	2	99.81%	0.19%
	54	2	99.97%	0.03%
	60	2	100.00%	0.00%
statevec	5	2	100.00%	0.00%
	6	2	99.92%	0.08%
	7	2	99.92%	0.08%
	8	2	99.99%	0.01%
vqe	12	2	99.95%	0.05%
	16	2	99.99%	0.01%
	20	2	99.98%	0.02%
	24	2	99.99%	0.01%

Table 6.5: Choice of Ω : performance of OAC with different Ω on the `hh1` circuit with 9 qubits, which initially contains 63392 gates. For reference, we also present the performance of VOQC at the end.

Ω	Time (s)	Output gate count (reduction)
2	17.4	27600 (-56.46%)
5	13.9	27620 (-56.43%)
10	17.7	27609 (-56.45%)
20	18.9	27590 (-56.48%)
40	21.8	27570 (-56.51%)
80	35.3	27564 (-56.52%)
160	66.6	27559 (-56.53%)
320	122.8	27551 (-56.54%)
VOQC	74.1	27673 (-56.35%)

minimal. We separately run the same experiments with $\epsilon = 0$, which forces the algorithm to run up to perfect convergence, and observe that OAC requires 4 rounds of optimization on average over all circuits. These results show that in practice a small number of compaction rounds suffice to obtain results that are within a very small fraction of the local optimal.

Impact of varying segment size Ω

Table 6.5 shows the running time and the gate count reduction of OAC with different values of Ω on the `hh1` circuit with 9 qubits, which initially contains 63392 gates. The results show that for a wide range of Ω values, our optimizer produces a circuit of similar quality to the baseline VOQC, and typically does so in significantly less time. When Ω is large, OAC’s running time scales linearly with Ω , and the output gate count reduces marginally when Ω increases. We choose $\Omega = 40$ in our evaluation to achieve a balance between running time and output quality but note that many different values work similarly well.

6.6 Experiments with Clifford+T Gate Set

In this section, we present the results of experiments with the Clifford+T gate set. These results largely mirror the results presented in the previous section, showing similar efficiency improvements and quality guarantees.

6.6.1 RQ I: Effectiveness of OAC

We validate the effectiveness of local optimality with the Clifford+T gate set. The Clifford+T gate set contains the Hadamard (H), Phase (S), controlled-NOT ($CNOT$), and the T gate. Our benchmark suite includes seven circuit families. We generate the Clifford+T circuits by transpiling the preprocessed Nam circuits using Qiskit and gridsynth [113, 119]. For `bwt` and `hh1`,

the two largest circuits of these families failed to transpile (due to running out of memory) so we exclude these from the Clifford+T evaluation.

For this gate set, we run our OAC optimizer using FeynOpt [9] as the oracle optimizer with segment depth $\Omega = 120$. We describe in the next section, why we chose this value for Ω . We evaluate the running time and output quality of our optimizer against the baseline FeynOpt. The cost function is the T count, that is the number of T gates of the circuit.

We note that another possibility for optimizing Clifford+T circuits is the PyZX tool, which uses the ZX-diagrams for optimizations. This tool, however, can require significant time to translate between ZX-diagrams and the circuit representation. For example, in our experiments, we found that for many circuits, the PyZX tool spends more than 50% of its time on average translating between circuits and diagrams. Because our algorithm invokes the optimizer many times, the translation times between circuits and diagrams can become a bottleneck. We therefore use the FeynOpt in our evaluation, which does not suffer from this problem, as it operates directly on the circuit.

Table 6.6 shows the results of this experiment. The figure separates circuit families with horizontal lines and sorts circuits within families by their size/number of qubits. It presents the initial T count for each circuit and the running times of both optimizers, highlighting the fastest one in bold. It computes the speedup achieved by our OAC; a speedup of $10\times$ means our optimizer runs $10\times$ faster. The figure also shows percentage reductions in T count achieved by optimizers FeynOpt and OAC.

The results show that our OAC generates high-quality circuits for the Clifford+T gate set reasonably quickly, taking between 4 seconds and approximately 20 minutes (for a circuit containing over 250,000 T gates).

We observe the following:

1. **Time Performance:** Our OAC is faster for all circuit families, except perhaps for some smaller circuits.
2. **Scalability:** Within each circuit family, the speedup of our OAC increases with increasing T counts. It is over $100\times$ faster for some cases and $9.9\times$ faster on average.
3. **Circuit Quality:** Our OAC matches the T count reductions of FeynOpt on all benchmarks.

Summary. Table 6.6 shows that our OAC optimizer can be significantly faster, especially for larger circuits, because it scales better, and does so without sacrificing optimization quality when optimizing for T count. The experiment thus shows that the local optimality approach can work well for Clifford+T circuits, especially for larger circuits.

6.6.2 RQ IV: Impact of varying segment size Ω

We evaluate the impact of parameter Ω on the output quality and running time of our OAC algorithm. We use OAC on the Clifford+T gate set and optimize for T count with FeynOpt as the oracle optimizer. Figure 6.9 plots the output T count (number of T gates) and the running time against Ω . The figure includes Ω values 2, 5, 15, 30, 60, 120 . . . 7680; we provide the raw data for the plot in Table 6.7. For the different values of Ω , we run our optimizer on the `hh1` circuit with 7 qubits, which initially contains 61246 T gates.

Table 6.6: The figure shows the optimization results of optimizers OAC and FeynOpt, with T count as the cost function. It shows the running time in seconds for both optimizers (lower is better) and calculates the speedup of our OAC by taking the ratio of the two timings. The figure also shows the T count reductions of both tools. The results show that our OAC delivers excellent time performance and runs almost an order of magnitude ($9.9\times$) faster than FeynOpt on average. Our OAC optimizer achieves this speedup without any sacrifice in circuit quality, producing circuits of the same quality as FeynOpt.

Family	Qubits	Input T Count	Time		OAC speedup	T Count Reduction	
			FeynOpt	OAC		FeynOpt	OAC
bwt	17	169330	35730.0	354.8	100.69	-13.2%	-13.2%
	21	214585	68301.4	569.1	120.01	-19.5%	-19.4%
grover	9	3927	2.6	3.3	0.79	-31.2%	-31.2%
	11	13720	12.0	10.5	1.15	-33.7%	-33.7%
	13	36920	140.9	34.8	4.05	-33.1%	-33.1%
	15	92016	2609.6	104.5	24.97	-32.7%	-32.7%
hhl	7	61246	409.8	31.4	13.03	-31.2%	-31.2%
	9	565183	T.O.	535.4	> 80.69	T.O.	-34.1%
hwb	8	5887	4.1	5.9	0.69	-25.7%	-25.7%
	10	29939	250.8	45.8	5.48	-29.8%	-29.8%
	11	84196	4767.3	129.9	36.7	-31.3%	-31.3%
	12	171465	21880.4	362.6	60.35	-34.1%	-34.1%
qft	48	44803	195.1	77.1	2.53	-20.2%	-20.2%
	64	61027	531.4	138.8	3.83	-20.3%	-20.3%
	80	77251	1083.4	204.4	5.3	-20.3%	-20.3%
	96	93475	1931.9	355.6	5.43	-20.3%	-20.3%
shor	10	6104	2.0	4.0	0.51	-19.7%	-19.7%
	12	20180	21.0	16.5	1.27	-20.3%	-20.3%
	14	70544	999.5	76.9	12.99	-20.5%	-20.5%
	16	266060	28382.3	396.8	71.53	-20.6%	-20.6%
sqrt	42	25104	569.2	69.3	8.21	-37.4%	-37.4%
	48	60366	5441.6	189.8	28.67	-39.9%	-39.9%
	54	140830	36747.0	631.8	58.16	-41.7%	-41.7%
	60	261308	T.O.	1212.1	> 35.64	T.O.	-29.8%
average					> 9.91	-27.1%	-27.5%

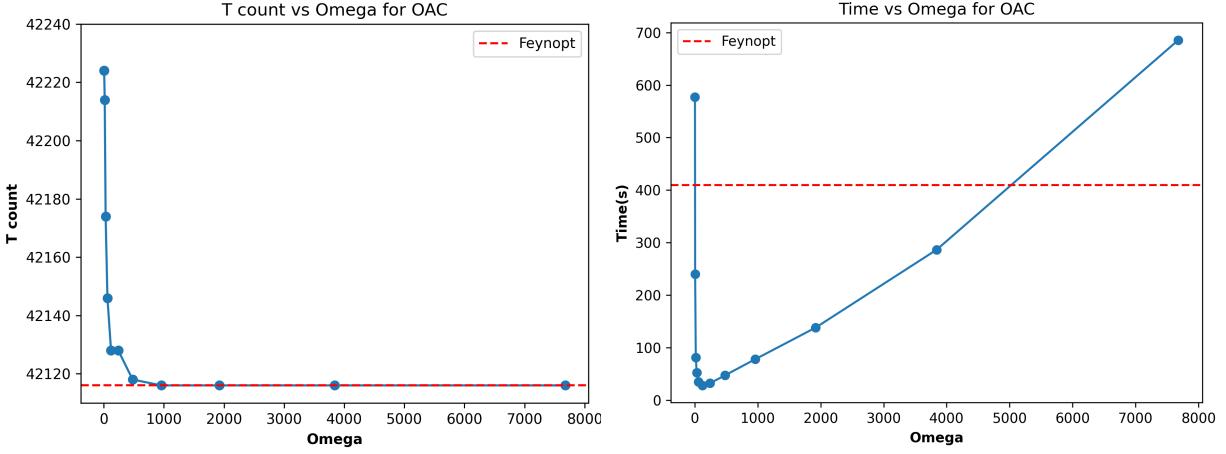


Figure 6.9: The figure plots the impact of the parameter Ω on the output T count (lower is better) and running time of OAC optimizer on an `hhl` circuit with 7 qubits. The dotted red lines in the plots denote the output T count and the running time of the oracle optimizer *FeynOpt* on the whole circuit. For almost all values of Ω , the output quality matches the oracle optimizer, demonstrating that local optimality is a robust quality criterion for T count optimization.

The red dotted line in the plots show the output T count and running time of the baseline, where the oracle *FeynOpt* runs on the entire circuit. The results show that for a wide range of Ω values, our optimizer produces a circuit of similar quality to the baseline *FeynOpt* and typically does so in significantly less time (with the exception of two values of Ω at the extremities: 2 and 7680).

T count. The plot for T count shows that for small Ω (around 2), increasing it improves the output quality of OAC. This is because the local optimality guarantee of OAC strengthens with increasing Ω , as it guarantees larger segments to be optimal. However, the benefits of increasing Ω become incremental around $\Omega = 60$, where the T count stabilizes around 42140 (20 gates from optimal). This demonstrates that local optimality is an effective quality criterion, generating high-quality circuits even with relatively small values of Ω (around 60).

Run time. One might expect the running time of OAC to increase with segment size Ω because: 1) OAC uses the oracle on segments of size 2Ω , thus each oracle call consumes more time and 2) OAC gives a stronger quality guarantee—larger the Ω , stronger the guarantee given by local optimality. Indeed, this intuition holds for most values in practice. For Ω values ranging from 120 to 7680, the running time increases with increasing Ω . In this range, the increased cost of oracle call dominates the running time, making it faster to partition circuits into smaller segments and make many (smaller) calls to the oracle.

Table 6.7: Results for OAC with FeynOpt on the `hhl` circuit with 7 qubits.

Ω	Output T count	Time (s)
2	42224	577
5	42224	240.16
15	42214	81.10
30	42174	52.47
60	42146	34.58
120	42128	28.16
240	42128	32.08
480	42118	47.22
960	42116	77.85
1920	42116	138.16
3840	42116	286.31
7680	42116	685.69

However, when Ω is very small, we observe the opposite: increasing Ω reduces the running time. For reference, we include Figure 6.10, which zooms in on the running time plot from Figure 6.9 for initial values of Ω , ranging from 2, 5, 15 . . . 120. For these smaller Ω values, although each oracle call is fast, the number of oracle calls dominates the time cost. The OAC algorithm splits the circuit into a large number of small segments and queries the oracle on each one, resulting in many calls to the optimizer. When a circuit segment is small, it is more efficient to directly call FeynOpt, which optimizes it in one pass. For this reason, $\Omega = 120$ is a good value for our optimizer OAC, as it does not send large circuits to the oracle, and also does not split the circuit into a large number of really small segments.

Overall, we observe that for a wide range of Ω values, our OAC outputs high quality circuits and does so in a shorter running time than the baseline.

6.7 Conclusion

Quantum circuit optimization is a fundamental problem in quantum computing. State-of-the-art optimizers require at least quadratic time in the size of the circuit, which does not scale to larger circuits that are necessary for obtaining quantum advantage, and are unable to make strong quality guarantees. This chapter defines a notion of local optimality and shows that it is possible to optimize circuits for local optimality efficiently by proposing a circuit cutting-and-melding technique. With this cut-and-meld technique, the algorithm cuts a circuit into subcircuits, optimizes them independently, and melds them efficiently, while also guaranteeing optimization quality. Our implementation and experiments show that the algorithm is practical and performs well, leading to more than an order of magnitude performance improvement (on average) while also improving optimization quality. These results show that local optimality can be effective in optimizing large quantum circuits, which are necessary for quantum advantage. These results, however, do not suggest stopping to develop global optimizers, which remains to be an impor-

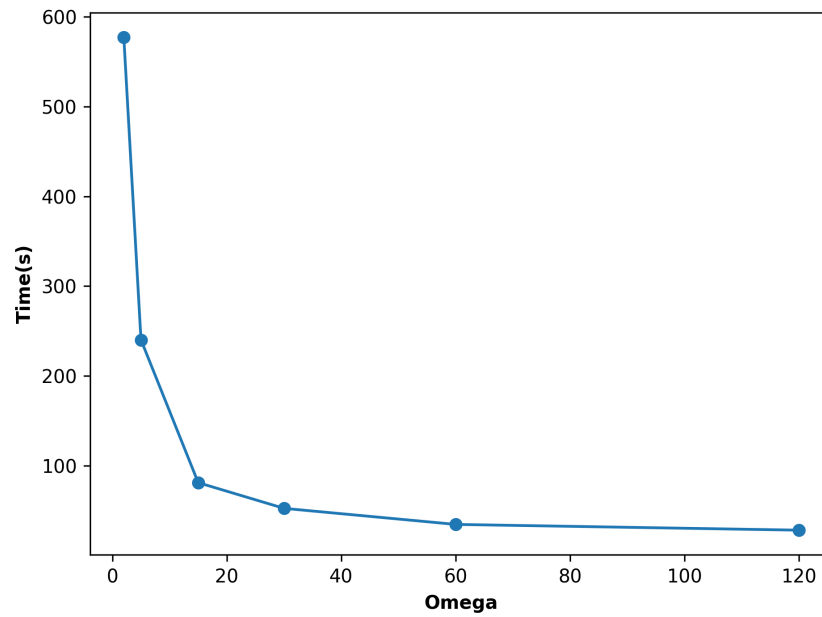


Figure 6.10: Zooming in: Time vs. Omega plot

tant goal. It is likely, however, due to the inherent complexity of the problem (it is QMA-hard), global optimizers may struggle to scale to larger circuits efficiently. Because our approach to local optimality is generic, it can scale global optimizers to larger circuits by employing them as oracles for local optimization.

Chapter 7

Atlas: Optimization for Simulation on Classical Hardware

The optimization strategies presented in previous chapters serve to minimize resource usage and extend the reach of near-term algorithms. However, even with these reductions, the robustness demands of many quantum applications exceed those of modern *noisy intermediate-scale quantum* (NISQ) computers, which suffer from decoherence and lack of error-correction [40, 110]. Furthermore, NISQ computers are expensive resources—many remain inaccessible beyond a small group of researchers. Consequently, there is significant interest in quantum circuit simulation, which enables the validation of optimized circuits and the performance of robust quantum computation on classical parallel machines.

Several approaches to simulating quantum circuits have been proposed, including Feynman-style (path integral) and Schrödinger-style (state vector) simulation. While Feynman-style simulation requires polynomial space, its time complexity is often prohibitive for large circuits. Consequently, most modern simulators adopt Schrödinger-style simulation, which maintains the full quantum state in a *state vector* of size 2^n (for n qubits) and updates this vector by applying circuit gates sequentially.

To tackle the scalability challenges of state-vector simulation, researchers exploit the inherent parallelism of the task using CPUs, GPUs, and distributed systems [5, 8, 15, 16, 58, 72, 104, 132, 133, 157, 159]. Furthermore, to cope with the memory demands of larger circuits, recent works have proposed offloading state vectors to DRAM or secondary storage (e.g., disks) [65, 132]. Despite over a decade of research, however, achieving both high performance and scalability in quantum circuit simulation remains an open challenge.

At a high level of abstraction, there are three primary bottlenecks in distributed Schrödinger-style simulation:

1. **Memory Capacity:** Storing a state vector with 2^n complex amplitudes requires exponential memory. Overcoming this requires distributing the state vector across a parallel machine with heterogeneous memory resources.
2. **Communication Overhead:** Simulating gates on qubits that span across partition boundaries involves strided memory accesses. This necessitates *fine-grained* inter-node or inter-device communication, which becomes the dominant performance bottleneck at scale.

3. **Low Arithmetic Intensity:** Because quantum gates typically operate on a small number of qubits, the simulation involves multiplying a sparse matrix by a dense vector. This results in low arithmetic intensity (e.g., as little as 0.5 [65]), which underutilizes modern accelerators (such as GPUs) designed for compute-intensive workloads.

In this chapter, we present techniques for performant and scalable quantum circuit simulation on modern multi-node GPU architectures. We adopt a Schrödinger-style approach, utilizing the aggregated memory across all nodes to store the state. To minimize the communication costs identified above, we partition the quantum circuit into a sequence of *stages*. Each stage consists of a contiguous subcircuit that can be simulated on a single GPU without requiring non-local accesses (i.e., communication outside the GPU’s local memory). To address the arithmetic intensity challenge, we further partition the subcircuit of each stage into *kernels*—groups of gates large enough to maximize parallel throughput but small enough to prevent exponential blowups in computational cost.

Given this hierarchical structure (stages $\rightarrow$ kernels), we present a simulation algorithm that restricts expensive communication to the boundaries between stages. To maximize throughput, the algorithm *shards* the state vector into contiguous pieces, allowing multiple shards to be processed in parallel or sequentially, depending on resource availability.

We formulate this partitioning strategy as two distinct optimization problems:

1. **The Staging Problem:** We provide a solution based on Integer Linear Programming (ILP) to minimize the number of stages (and thus global communication steps).
2. **The Kernelization Problem:** We present a dynamic programming solution to group gates into efficient kernels.

We realize these techniques in an implementation named Atlas [151]. Our evaluation on up to 256 GPUs (across 64 nodes) demonstrates that Atlas is up to $20.2\times$ faster than state-of-the-art GPU-based simulators. In addition, for large quantum circuits that exceed GPU memory capacity, Atlas is up to $105\times$ faster than existing offloading-based simulators.

In summary, this chapter makes the following contributions:

- A hierarchical partitioning approach to scaling performant quantum circuit simulation based on staging and kernelization.
- An ILP algorithm to stage a circuit for simulation that can minimize the number of stages.
- A dynamic programming algorithm for kernelizing each stage to ensure efficient parallelism.
- An implementation that realizes hierarchical partitioning and significantly outperforms existing simulators.

7.1 Architectural Model

We assume a multi-node GPU architecture with 2^G nodes. Each node contains multiple GPUs and a single CPU with an attached DRAM module. Each GPU can store in its *local* memory 2^L amplitudes (complex numbers). Each node can store in its *regional* memory 2^{L+R} amplitudes, where 2^R can be the number of GPUs per node or can be set such that 2^{L+R} equals the number of amplitudes that can be stored in the DRAM memory of the node.

7.2 Hierarchical Partitioning

We present our quantum simulation algorithm with hierarchical circuit partitioning.

Algorithm 6 Hierarchical partitioning of a quantum circuit $\mathcal{C}$ and the simulation algorithm for simulating the circuit starting with the input state $state$. The parameters L, R, G describe the distributed execution model with L local qubits, R regional qubits, and G global qubits.

```

1: function PARTITION( $\mathcal{C}, L, R, G$ )
2:    $stages = \text{STAGE}(\mathcal{C}, L, R, G)$ 
3:    $stagedKernels = []$ 
4:   for  $i = 0 \dots |stages| - 1$  do
5:      $(\mathcal{C}_i, \mathcal{Q}) = stages[i]$ 
6:      $kernels_i = \text{KERNELIZE}(\mathcal{C}_i)$ 
7:      $stagedKernels[i] = (kernels_i, \mathcal{Q})$ 
8:   return  $stagedKernels$ 
9: function EXECUTE( $stagedKernels, state$ )
10:   $\mathcal{P} = \text{identity permutation}$ 
11:   $shards = [state]$ 
12:  for  $i = 0 \dots |stagedKernels| - 1$  do
13:     $(kernels_i, \mathcal{Q}) = stagedKernels[i]$ 
14:     $(shards, \mathcal{P}) = \text{SHARD}(shards, \mathcal{Q}, \mathcal{P})$ 
15:    parfor  $shard$  in  $shards$  do
16:      for  $k$  in  $kernels_i$  do
17:        LAUNCHKERNEL( $k, shard$ )
18: function SIMULATE( $\mathcal{C}, state, L, R, G$ )
19:   $stagedKernels = \text{PARTITION}(\mathcal{C}, L, R, G)$ 
20:  EXECUTE( $stagedKernels, state$ )

```

Algorithm 6 shows the pseudocode for our algorithms for quantum circuit simulation with hierarchical partitioning. The algorithm PARTITION takes as arguments the input circuit $\mathcal{C}$ and the architecture parameters consisting of the number of local, regional, and global qubits, L, R, G respectively. It partitions the input circuit $\mathcal{C}$ into stages using the STAGE function. Each *stage* consists of a subcircuit and a partition of the logical qubits of the (entire) circuit into sets of local, regional, and global qubits, such that the subcircuit can be simulated entirely locally by a single GPU, performing local memory accesses only. We describe the staging algorithm STAGE in Section 7.3.

After staging the circuit, the algorithm PARTITION, proceeds to kernelize each circuit by calling KERNELIZE on the subcircuit of each stage. The algorithm KERNELIZE further partitions the subcircuit for the stage into subcircuits each of which may be efficiently executed on a single GPU by taking advantage of data parallelism. The basic idea behind the algorithm is to group gates to amortize the cost of data parallelism on modern GPUs (e.g., due to cost of launching kernels) and do so without leading to an exponential blow-up of costs, because grouping gates can increase the cost exponentially in the number of qubits involved. We describe the kernelization

algorithm `KERNELIZE` in Section 7.4. The algorithm `PARTITION` completes by returning a list of stages, each of which consists of a list of kernels.

The algorithm `EXECUTE` takes a list of kernelized stages and input state consisting of a vector of amplitudes, and executes each stage in order. For each stage, the algorithm permutes the state vector and shards it into contiguous sections as demanded by the qubit partition $\mathcal{Q}$ for that stage. To minimize communication costs during this permutation and sharding process, the algorithm keeps track of the current permutation $\mathcal{P}$. The algorithm then considers each shard in parallel and applies the kernels of the stage to the shard sequentially. Because our algorithm staged and kernelized the circuit to maximize throughput for a distributed GPU architecture, each kernel can run efficiently on a single GPU, by applying each kernel to all the amplitudes in a shard of the state vector in a data parallel fashion. If sufficiently many GPUs are available, then each shard may be assigned to a GPU for parallel application. If, however, fewer GPUs are available, the shards may be stored in the shared DRAM at each node, and swapped in and out of the GPUs for execution.

Our simulation algorithm builds on top of our `PARTITION` and `EXECUTE` algorithms. Given the input circuit $\mathcal{C}$ and an input state *state*, along with architecture parameters L, R, G , our simulation algorithm `SIMULATE` starts by partitioning the input circuit into a list of kernelized stages. It then calls the algorithm `EXECUTE` to execute the stages.

Figure 7.1 shows an example application of the algorithm. We note that `PARTITION` does not depend on *state*. Both `STAGE` and `KERNELIZE` work for arbitrary input states.

7.3 Circuit Staging

We present an algorithm for (circuit) staging that partitions the circuit into *stages*, each of which is a contiguous subcircuit of the input circuit, along with a partition of the qubits into local, regional, and global sets such that each gate in the subcircuit of the stage operates on the local qubits.

Definition 6 (Local, regional, and global qubits). *For a quantum circuit simulation with n qubits, let q_i ($0 \leq i \leq n - 1$) be the i -th physical qubit. Let each shard include 2^L states, and let the DRAM of each node save 2^{L+R} states.*

- The first L physical qubits (i.e., $q_0, \dots, q_{L-1}$) are local qubits.
- The next R physical qubits (i.e., $q_L, \dots, q_{L+R-1}$) are regional qubits.
- The final $G = n - L - R$ physical qubits (i.e., $q_{L+R}, \dots, q_{n-1}$) are global qubits.

We categorize physical qubits into local, regional, and global subsets based on the communications required to simulate the application of a general quantum gate to these qubits. First, applying a gate to a local qubit only requires accessing states within the same shard, and therefore avoids communications. Second, applying a gate to a regional qubit requires accessing states in different shards stored on the same compute node, which only requires inter-device (intra-node) communications. Finally, applying a gate to a global qubit requires states stored on different compute nodes and thus involves inter-node communications.

Given this partition of local, regional, and global qubits, we aim to map logical qubits of a circuit to physical qubits in a way that avoids excessive communication. To minimize com-

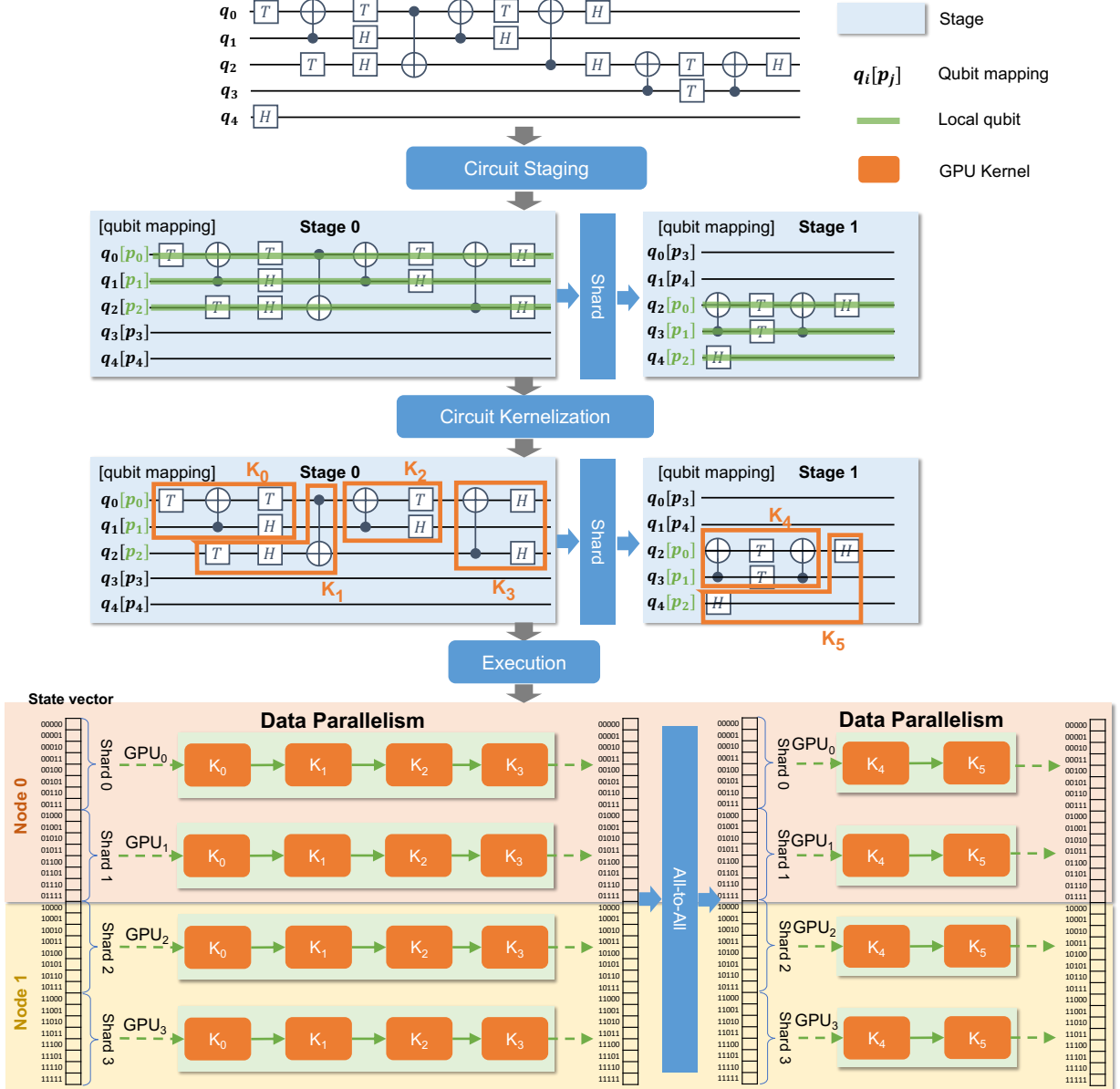


Figure 7.1: An example application of circuit partitioning and execution. We stage the circuit so that qubits of each gate map to local qubits (i.e., green lines in each stage). The notation $q_i[p_j]$ indicates that the i -th logical qubit maps to the j -th physical qubit. The KERNELIZE algorithm then partitions the gates of each stage into kernels that provide for data parallelism.

munication cost, we leverage a specific type of qubits in certain types of gates, called *insular qubits*.

Definition 7 (Insular Qubit). For a single-qubit gate, the qubit is insular if the unitary matrix of the gate is diagonal or anti-diagonal, i.e., the non-zero entries are along the (anti) diagonal. For

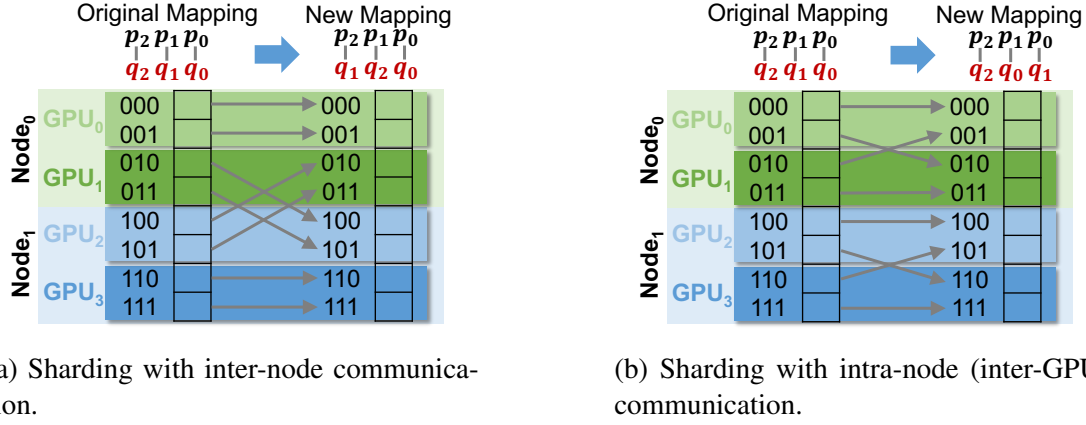


Figure 7.2: Sharding with different types of communication. The simulation has 1 local, 1 regional, and 1 global qubit. $p_i - q_j$ indicates that the i -th physical qubit is mapped to the j -th logical qubit. Inter-node communication is triggered if we update any global qubits (Figure 7.2a), and only intra-node communication is triggered otherwise (Figure 7.2b).

multi-qubit controlled- U gates¹, all control qubits are insular². All other qubits are non-insular.

Our idea of insular qubits is inspired by *global gate specialization* introduced by Häner *et al.* [65]. Intuitively, for a single-qubit gate with an insular qubit, computing each output state only depends on one input state (since the gate’s unitary matrix is diagonal or anti-diagonal), which allows Atlas to map insular qubits to regional and/or global physical qubits without introducing any extra communication. This property allows Atlas to only consider the non-insular qubits of quantum gates when mapping qubits.

The Atlas staging algorithm (Algorithm 7) splits a quantum circuit’s simulation into multiple stages, each of which includes a subcircuit and uses a different mapping from logical to physical qubits. Within each subcircuit, all non-insular qubits of all gates can only operate on *local* physical qubits. This approach avoids any inter-device or inter-node communications within a stage and only requires all-to-all communications between stages to perform qubit remapping.

We formalize circuit staging as a constrained optimization problem and design a binary integer linear programming (ILP) algorithm to discover a staging strategy that minimizes the total communication cost.

Circuit staging problem Given an inter-node communication cost factor c , for a given input circuit $\mathcal{C}$ with n qubits and an integer s , Atlas circuit staging algorithm splits $\mathcal{C}$ into at most s stages $\mathcal{C}_0, \dots, \mathcal{C}_{s-1}$ and determines a qubit partition $\mathcal{Q}$ of local/regional/global qubits for each

¹A *controlled- U* gate has some *control* qubits controlling a U gate (can be any unitary gate) on some *target* qubits. If at least one of the control qubits is $|0\rangle$, then the target remains unchanged. If all of the control qubits are $|1\rangle$, then the gate U is applied to the target qubits.

²In some controlled- U gates, any qubit can be chosen as the control qubit without changing the output. In such gates, all qubits are insular.

stage, minimizing the total communication cost:

$$\sum_{i=1}^{s-1} \left(|\mathcal{Q}_i^{local} \setminus \mathcal{Q}_{i-1}^{local}| + c \cdot |\mathcal{Q}_i^{global} \setminus \mathcal{Q}_{i-1}^{global}| \right). \quad (7.1)$$

where $\mathcal{Q}_i^{local}$ is the local qubit set of stage i and $\mathcal{Q}_i^{global}$ is the global qubit set of stage i . $|\mathcal{Q}_i^{local} \setminus \mathcal{Q}_{i-1}^{local}|$ is the number of local qubits that need to be updated, approximating the inter-GPU communication cost; $|\mathcal{Q}_i^{global} \setminus \mathcal{Q}_{i-1}^{global}|$ is the number of global qubits that need to be updated, approximating the extra inter-node communication cost. Although regional qubits do not appear directly in (7.1), they are critical for allowing the number of global qubits that need to be updated to be smaller than the local one. Without regional qubits, any inter-GPU communication will also be inter-node.

Let $\mathcal{G}$ denote the gates of the circuit $\mathcal{C}$, and let $\mathcal{E}$ denote their dependencies (adjacent gate pairs on the same qubit). Let $A_{q,k}$ and $B_{q,k}$ ($0 \leq q < n$, $0 \leq k < s$) denote whether the q -th logical qubit is mapped to a local/global physical qubit at the k -th stage, and let $F_{g,k}$ ($g \in \mathcal{G}$, $0 \leq k < s$) denote if the gate g is finished by the end of the k -th stage. For each $A_{q,k}$ and $B_{q,k}$ ($k < s - 1$), we also introduce two ancillary variables $S_{q,k}$ and $T_{q,k}$ that indicate whether the q -th logical qubit is updated from local to non-local or from global to non-global from the k -th to the $(k + 1)$ -th stage (i.e., $S_{q,k} = 1$ if and only if $A_{q,k} = 0$ and $A_{q,k+1} = 1$; $T_{q,k} = 1$ if and only if $B_{q,k} = 0$ and $B_{q,k+1} = 1$). Note that $A_{q,k}$, $B_{q,k}$, $F_{g,k}$, $S_{q,k}$, and $T_{q,k}$ are all binary variables. Minimizing the total cost of local-to-non-local and global-to-non-global updates with constraints yields the following objective:

$$\min \sum_{k=0}^{s-2} \sum_{q=0}^{n-1} (S_{q,k} + c \cdot T_{q,k}) \quad (7.2)$$

subject to

$$A_{q,k+1} \leq A_{q,k} + S_{q,k} \quad \forall q \in [0, n) \quad \forall k \in [0, s - 1) \quad (7.3)$$

$$B_{q,k+1} \leq B_{q,k} + T_{q,k} \quad \forall q \in [0, n) \quad \forall k \in [0, s - 1) \quad (7.4)$$

$$F_{g,k} \leq F_{g,k+1} \quad \forall g \in \mathcal{G} \quad \forall k \in [0, s - 1) \quad (7.5)$$

$$F_{g,k} \leq F_{g,k-1} + A_{q,k} \quad q \text{ is a non-insular qubit of } g \quad (7.6)$$

$$F_{g_1,k} \geq F_{g_2,k} \quad \forall (g_1, g_2) \in \mathcal{E} \quad \forall k \in [0, s) \quad (7.7)$$

$$F_{g,s-1} = 1 \quad \forall g \in \mathcal{G} \quad (7.8)$$

$$A_{q,k} + B_{q,k} \leq 1 \quad \forall q \in [0, n) \quad \forall k \in [0, s) \quad (7.9)$$

$$\sum_{q=0}^{n-1} A_{q,k} = L \quad \sum_{q=0}^{n-1} B_{q,k} = G \quad \forall k \in [0, s) \quad (7.10)$$

We now describe these constraints. First, constraints 7.3, 7.4 and 7.5 can be derived from the definitions of $S_{q,k}$, $T_{q,k}$ and $F_{g,k}$. Second, constraint 7.6 is a locality constraint that a gate g can be executed at the k -th stage only if g 's non-insular qubits q are all mapped to local physical qubits (i.e., $A_{q,k} = 1$). Third, constraint 7.7 represents a dependency constraint that all gates

Algorithm 7 The Atlas circuit staging algorithm.

```

1: function STAGE( $\mathcal{C}, L, R, G$ )
2:   for  $s = 1, 2, \dots$  do
3:      $status, A, B, F = \text{SOLVEILP}(\mathcal{C}, L, R, G, s)$ 
4:     if  $status = \text{feasible}$  then
5:       Find  $(\mathcal{C}_0, \mathcal{Q}_0), \dots, (\mathcal{C}_{s-1}, \mathcal{Q}_{s-1})$  based on  $A, B, F$ 
6:       return  $(\mathcal{C}_0, \mathcal{Q}_0), \dots, (\mathcal{C}_{s-1}, \mathcal{Q}_{s-1})$ 

```

must be executed by following a topological order with respect to dependencies between them. Fourth, constraint 7.8 is derived from the completion constraint that all gates must be executed in the s stages. Next, constraint 7.9 declares that any qubit cannot be both local and global at the same time. Finally, constraint 7.10 specifies the hardware constraint that each stage has L local physical qubits and G global qubits.

ILP-based circuit staging. For a given input circuit $\mathcal{C}$ and an integer s that specifies the maximum number of stages $\mathcal{C}$ can be partitioned into, Atlas sends the objective (i.e., Equation (7.2)) and all constraints (i.e., Equations (7.3) to (7.10)) to an off-the-shelf integer-linear-programming (ILP) solver, which returns an assignment for matrices A, B, F, S, T that minimizes the objective while satisfying all constraints. Atlas then stages the circuit $\mathcal{C}$ into $(\mathcal{C}_0, \mathcal{Q}_0), \dots, (\mathcal{C}_{s-1}, \mathcal{Q}_{s-1})$ based on A, B and F , where $\mathcal{C}_i$ is a subcircuit and $\mathcal{Q}_i$ is a qubit partition into local/regional/global qubit sets. Specifically, for a quantum gate $g \in \mathcal{G}$, g will be executed by the stage with index: $\min\{k | F_{g,k} = 1\}$. Atlas maps the q -th logical qubit to a local (physical) qubit at the k -th stage if $A_{q,k} = 1$, to a global qubit at the k -th stage if $B_{q,k} = 1$, and to a regional qubit if $A_{q,k} = B_{q,k} = 0$.

Algorithm 7 shows the Atlas staging algorithm. It uses the ILP solver as a subroutine and returns the first feasible result.

Theorem 8 (Optimality of STAGE). *Algorithm 7 returns the minimum feasible number of stages.*

Proof. Algorithm 7 returns the first feasible result of the ILP solver, and the loop in line 2 ensures that the number of stages is minimized. The ILP constraints are the feasible constraints for stages, so Algorithm 7 returns the minimum feasible number of stages. $\square$

On top of the minimum number of stages, the ILP minimizes the total communication cost (the ILP objective).

7.4 Circuit Kernelization

After partitioning an input circuit into multiple stages, Atlas must efficiently execute the gates of each stage on GPUs. GPU computations are organized as *kernels*, each of which is a function simultaneously executed by multiple hardware threads in a single-program-multiple-data (SPMD) fashion [43]. A straightforward approach to executing gates is to launch a kernel for each gate, which yields suboptimal performance due to the low arithmetic intensity of each gate. Another

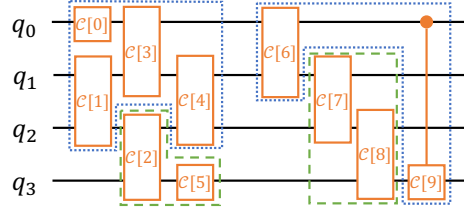


Figure 7.3: Kernel examples. The two green dashed kernels satisfy Constraint 1. The two blue dotted kernels do not satisfy Constraint 1 and thus are not considered by the KERNELIZE algorithm.

extreme is to fuse all gates into a single kernel, which would exponentially increase the amount of computation.

To improve simulation performance on GPUs, Atlas uses a dynamic programming algorithm (Algorithm 8) to partition the gates of a stage into kernels. We formulate circuit kernelization as an optimization problem as follows.

Problem 1 (Optimal circuit sequence kernelization). *Given a cost function $\text{COST}(\cdot)$ that maps one or multiple kernels (a sequence of gates) to their cost (i.e., a real number), the optimal circuit sequence kernelization problem takes as input a quantum circuit $\mathcal{C}$ represented as a sequence of gates, and returns a kernel sequence $\mathcal{K}_0, \dots, \mathcal{K}_{b-1}$ that minimizes the total execution cost:*

$$\sum_{i=0}^{b-1} \text{COST}(\mathcal{K}_i) \quad (7.11)$$

such that each kernel consists of a contiguous segment of the gate sequence $\mathcal{C}$.

We note that summing up the cost of individual kernels (Equation (7.11)) is a faithful representation of the total cost, because on a GPU, each kernel may execute in parallel, but multiple kernels are executed sequentially.

Also, note that each kernel consisting of a contiguous segment of the gate sequence $\mathcal{C}$ is a conservative requirement. It is feasible to execute any kernel sequence that forms a topologically equivalent sequence to $\mathcal{C}$. However, there are exponentially many ways to consider kernels with non-contiguous segments. To consider them efficiently, we present a key constraint on the kernels of the KERNELIZE algorithm in Algorithm 8.

Constraint 1 (Constraints on kernels). *Each kernel $\mathcal{K}$ considered by the KERNELIZE algorithm satisfies the following constraints:*

1. *Weak convexity: $\forall j_1 < j_2 < j_3$, if $\mathcal{C}[j_1] \in \mathcal{K}$, $\mathcal{C}[j_2] \notin \mathcal{K}$, $\mathcal{C}[j_3] \in \mathcal{K}$, then $\text{QUBITS}(\mathcal{C}[j_1]) \cap \text{QUBITS}(\mathcal{C}[j_2]) \cap \text{QUBITS}(\mathcal{C}[j_3]) = \emptyset$. Informally, weak convexity requires that for any three gates $\mathcal{C}[j_1], \mathcal{C}[j_2], \mathcal{C}[j_3]$, if only the middle gate is not in the kernel, then they cannot share a qubit.*
2. *Monotonicity: for any gate $\mathcal{C}[j] \notin \mathcal{K}$, if $\mathcal{C}[j]$ shares a qubit with $\mathcal{K} \downarrow j$ then $\text{QUBITS}(\mathcal{K}) = \text{QUBITS}(\mathcal{K} \downarrow j)$ where $\mathcal{K} \downarrow j = \mathcal{K} \cap \{\mathcal{C}[0], \mathcal{C}[1], \dots, \mathcal{C}[j-1]\}$. Informally, if we decide to exclude a gate from $\mathcal{K}$ while it shares a qubit with $\mathcal{K}$, then we fix the qubit set of $\mathcal{K}$ from that gate on.*

Algorithm 8 The KERNELIZE algorithm. The operators “+” on lines 13 and 14 mean adding the cost and appending the kernel set to the sequence. We maintain the order in place in line 11 (replacing $\mathcal{K} \setminus \{\mathcal{C}[i]\}$ with $\mathcal{K}$) if the monotonicity constraint in Constraint 1 applies to $\mathcal{K}$, or move $\mathcal{K}$ to the end of the kernel set otherwise. We require $\tilde{\kappa}$ to be a suffix of κ' in line 13.

```

1: function KERNELIZE( $\mathcal{C}$ )
2:   Input: A quantum circuit  $\mathcal{C}$  represented as a sequence of gates.
3:   Output: A sequence of kernels.
4:   // Suppose that we have an ordered kernel set  $\kappa$  in the first  $i$  gates,  $DP[i, \kappa]$  stores the
      minimum cost to kernelize the first  $i$  gates except for the gates in  $\kappa$  and the corresponding
      kernel sequence.
5:    $DP[i, \kappa] = (\infty, [])$  for all  $i, \kappa$ 
6:    $DP[0, \emptyset] = (0, [])$ 
7:   for  $i = 0$  to  $|\mathcal{C}| - 1$  do
8:     for each  $\kappa$  such that  $\exists \mathcal{K}, \mathcal{C}[i] \in \mathcal{K} \in \kappa$  and  $\mathcal{K}$  satisfies Constraint 1 do
9:        $\tilde{\kappa} = \kappa \setminus \{\mathcal{K}\}$ 
10:      if  $|\mathcal{K}| > 1$  then
11:         $DP[i + 1, \kappa] = DP[i, \tilde{\kappa} \cup \{\mathcal{K} \setminus \{\mathcal{C}[i]\} \}]$ 
12:      else //  $\mathcal{K} = \{\mathcal{C}[i]\}$ 
13:         $DP[i + 1, \kappa] = \min_{\kappa' \supseteq \tilde{\kappa}} \{DP[i, \kappa'] + \text{COST}(\kappa' \setminus \tilde{\kappa})\}$ 
14:       $DP_{best} = \min_{\kappa} \{DP[|\mathcal{C}|, \kappa] + \text{COST}(\kappa)\}$ 
15:      return  $DP_{best}.kernels$ 

```

In Figure 7.3, the blue dotted kernel on the left violates weak convexity because $\mathcal{C}[1]$, $\mathcal{C}[2]$ and $\mathcal{C}[4]$ share q_2 , and only $\mathcal{C}[2]$ is not in the kernel. If we allow this kernel to be considered, it will be mutually dependent with the kernel containing $\mathcal{C}[2]$, yielding no feasible results. The blue dotted kernel on the right does not violate weak convexity, but it violates monotonicity. $\mathcal{C}[7]$ is excluded from the kernel while sharing the qubit q_1 with the kernel, so the qubit set of the kernel should be fixed to $\{q_0, q_1\}$. So it cannot include $\mathcal{C}[9]$. In fact, including $\mathcal{C}[9]$ in this kernel causes mutual dependency with the kernel $\{\mathcal{C}[7], \mathcal{C}[8]\}$.

Constraint 1 ensures the kernels to form a valid sequence in Theorem 9.

Theorem 9 (Correctness of the KERNELIZE algorithm). *Algorithm 8 returns a kernel sequence such that concatenating the kernels forms a sequence topologically equivalent to $\mathcal{C}$.*

Proof. Proof by induction on i that for each $DP[i, \kappa].cost < \infty$, appending κ to $DP[i, \kappa].kernels$ results in the sequence $\mathcal{C}_{\kappa} \downarrow i$ which is topologically equivalent to a the sequence $\mathcal{C}[0], \mathcal{C}[1], \dots, \mathcal{C}[i - 1]$ (denote this sequence $\mathcal{C} \downarrow i$). This holds for $i = 0$ where $DP[i, \emptyset].kernels$ and $\mathcal{C} \downarrow 0$ are both empty.

Suppose the induction hypothesis holds for i . For $i + 1$, for any $\mathcal{K}$ satisfying Constraint 1, if $|\mathcal{K}| = 1$, line 13 in the algorithm makes the order of appending κ to $DP[i + 1, \kappa].kernels$ the same as the order of appending κ' to $DP[i, \kappa'].kernels$ for some κ' when $\tilde{\kappa}$ is a suffix of κ' , so we can directly apply the induction hypothesis to prove it for $i + 1$.

Suppose $|\mathcal{K}| > 1$. If the monotonicity constraint in Constraint 1 applies to $\mathcal{K}$, line 11 just inserts $\mathcal{C}[i]$ to the end of $\mathcal{K}$ in $\mathcal{C}_{\kappa} \downarrow i$. So it suffices to prove that $\mathcal{C}[i]$ does not depend on any gate

after the end of $\mathcal{K}$ in $\mathcal{C}_\kappa \downarrow i$, i.e., $\mathcal{C}[i]$ does not share any qubit with these gates.

By monotonicity,

$$\text{QUBITS}(\mathcal{K}) = \text{QUBITS}(\mathcal{K} \downarrow i). \quad (7.12)$$

For any gate $\mathcal{C}[j_2]$ after the last gate of $\mathcal{K}$ in the sequence $\mathcal{C}_\kappa \downarrow i$ before the insertion, by weak convexity, $\forall j_1 < j_2 < i$ such that $\mathcal{C}[j_1] \in \mathcal{K}$ (we know that $\mathcal{C}[j_2] \notin \mathcal{K}, \mathcal{C}[i] \in \mathcal{K}$), $\text{QUBITS}(\mathcal{C}[j_1]) \cap \text{QUBITS}(\mathcal{C}[j_2]) \cap \text{QUBITS}(\mathcal{C}[i]) = \emptyset$. Because all gates in $\mathcal{K}$ are before $\mathcal{C}[j_2]$ in the sequence $\mathcal{C}_\kappa \downarrow i$ before the insertion,

$$\text{QUBITS}(\mathcal{K} \downarrow i) \cap \text{QUBITS}(\mathcal{C}[j_2]) \cap \text{QUBITS}(\mathcal{C}[i]) = \emptyset \quad (7.13)$$

where $\mathcal{K} \downarrow i$ is the kernel $\mathcal{K}$ before the insertion of $\mathcal{C}[i]$.

By Equation (7.12),

$$\text{QUBITS}(\mathcal{C}[i]) \subseteq \text{QUBITS}(\mathcal{K} \downarrow i). \quad (7.14)$$

Plugging this into Equation (7.13),

$$\text{QUBITS}(\mathcal{C}[j_2]) \cap \text{QUBITS}(\mathcal{C}[i]) = \emptyset \quad (7.15)$$

for any gate $\mathcal{C}[j_2]$ after the last gate of $\mathcal{K}$ in the sequence $\mathcal{C}_\kappa \downarrow i$ before the insertion.

So we can safely insert $\mathcal{C}[i]$ to the end of $\mathcal{K}$ in $\mathcal{C}_\kappa \downarrow i$ without breaking the topological equivalence.

If the monotonicity constraint in Constraint 1 does not apply to $\mathcal{K}$, line 11 moves $\mathcal{K}$ to the end of $\mathcal{C}_\kappa \downarrow i$. Because the monotonicity constraint does not apply to $\mathcal{K}$, for any gate $\mathcal{C}[j_0] \in \mathcal{K}$, for any gate $\mathcal{C}[j] \notin \mathcal{K}$ after $\mathcal{C}[j_0]$ in the original sequence $\mathcal{C}_\kappa \downarrow i$, $\mathcal{C}[j]$ and $\mathcal{C}[j_0]$ do not share any qubits. So we can safely move the entire kernel $\mathcal{K}$ to the end without breaking the topological equivalence.

In all cases, we have proved that $\mathcal{C}_\kappa \downarrow (i + 1)$ is topologically equivalent to $\mathcal{C} \downarrow (i + 1)$. So $\mathcal{C}_\kappa \downarrow |\mathcal{C}|$ is topologically equivalent to $\mathcal{C} \downarrow |\mathcal{C}|$, which proves that any state in line 14 in the algorithm returns a kernel sequence such that concatenating the kernels forms a sequence topologically equivalent to $\mathcal{C}$. $\square$

Theorem 10 (Constraint 1 allows all contiguous kernels). *Any kernel of a contiguous segment of the gate sequence $\mathcal{C}$ satisfies Constraint 1.*

Proof. For any kernel $\mathcal{K}$ of a contiguous segment of gates, for any $j_1 < j_2 < j_3$ such that $\mathcal{C}[j_1] \in \mathcal{K}, \mathcal{C}[j_2] \notin \mathcal{K}$, we know that $\mathcal{C}[j_3] \notin \mathcal{K}$ by contiguity. So weak convexity holds, and monotonicity holds because $\mathcal{K} \downarrow j_2 = \mathcal{K}$. $\square$

7.5 Implementation

7.5.1 Reducing Size of DP State in KERNELIZE

In a quantum circuit, the number of gates is typically orders of magnitude greater than the number of qubits. Therefore, instead of tracking the gates for each kernel in Algorithm 8, we maintain the qubit set as well as an *extensible qubit set* for each kernel.

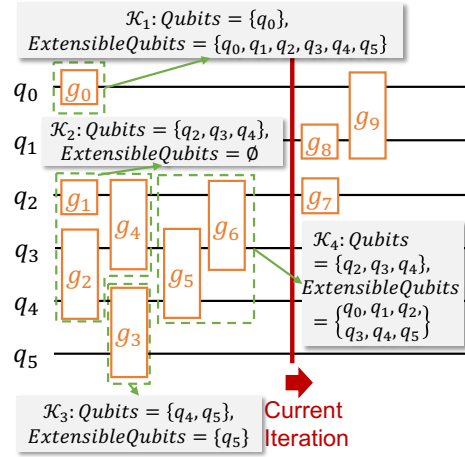


Figure 7.4: An example dynamic programming state in the implementation. The circuit sequence is $\mathcal{C} = [g_0, g_1, \dots, g_9]$.

Definition 8 (Extensible qubit for a kernel). *For a kernel $\mathcal{K}$ at position i , i.e., $\mathcal{K} \downarrow i$, a qubit q is extensible if and only if adding a gate operating on qubit q to $\mathcal{K} \downarrow i$ satisfies Constraint 1. Formally, a qubit q is extensible for $\mathcal{K} \downarrow i$ if and only if both of the following hold:*

1. *Weak convexity: $\forall j_1 < j_2 < i$, if $\mathcal{C}[j_1] \in \mathcal{K} \downarrow i$, $\mathcal{C}[j_2] \notin \mathcal{K} \downarrow i$, then $q \notin \text{QUBITS}(\mathcal{C}[j_1]) \cap \text{QUBITS}(\mathcal{C}[j_2])$.*
2. *Monotonicity: for any gate $\mathcal{C}[j] \notin \mathcal{K} \downarrow i$, if $j < i$ and $\mathcal{C}[j]$ shares a qubit with $\mathcal{K} \downarrow j$, then $q \in \text{QUBITS}(\mathcal{K} \downarrow j)$.*

This definition is a direct characterization of Constraint 1: adding the gate $\mathcal{C}[i]$ to a kernel $\mathcal{K}$ satisfies Constraint 1 if and only if all qubits of $\mathcal{C}[i]$ are extensible for $\mathcal{K} \downarrow i$. For example, the extensible qubit set is $\{q_5\}$ for $\mathcal{K}_3$ in Figure 7.4—we can only add gates operating only on this qubit to $\mathcal{K}_3$ to satisfy Constraint 1.

We append the code fragment of Algorithm 9 to the end of each iteration of the for loop in line 8 of Algorithm 8 to maintain the extensible qubit set for each kernel.

Theorem 11. *Algorithm 9 correctly computes the extensible qubit set for each kernel.*

Proof. Proof by induction on the position i . For the kernel $\mathcal{K}$ such that $\mathcal{C}[i] \in \mathcal{K}$, if $\mathcal{K} = \{\mathcal{C}[i]\}$, all qubits are extensible by Definition 8. Otherwise, $\text{EXTQ}(\mathcal{K}, i)$ is computed in the previous iteration.

$\forall \mathcal{K}' \in (\kappa \setminus \{\{\mathcal{C}[i]\}\})$, suppose $\text{EXTQ}(\mathcal{K}', i)$ is computed correctly.

Case 0. Consider the kernel $\mathcal{K}$ such that $\mathcal{C}[i] \in \mathcal{K}$ while $|\mathcal{K}| > 1$. In the condition for weak convexity, $\mathcal{C}[j_2]$ cannot be $\mathcal{C}[i]$ because it requires $\mathcal{C}[j_2] \notin \mathcal{K} \downarrow (i+1)$, so this constraint is exactly the same as weak convexity for $\mathcal{K} \downarrow i$.

If monotonicity applies to $\mathcal{K} \downarrow i$, it still applies to $\mathcal{K} \downarrow (i+1)$, and $\text{QUBITS}(\mathcal{K} \downarrow (i+1)) = \text{QUBITS}(\mathcal{K} \downarrow i)$; if monotonicity does not apply to $\mathcal{K} \downarrow i$, $\mathcal{C}[j]$ cannot be $\mathcal{C}[i]$ in the condition because it requires $\mathcal{C}[j] \notin \mathcal{K} \downarrow (i+1)$, so it still does not apply to $\mathcal{K} \downarrow (i+1)$. So $\text{EXTQ}(\mathcal{K}, i+1) = \text{EXTQ}(\mathcal{K}, i)$.

Now we case over each kernel $\mathcal{K}' \in (\kappa \setminus \mathcal{K})$. If monotonicity applies to $\mathcal{K}' \downarrow i$, there will

Algorithm 9 Maintaining the extensible qubit set EXTQ for each kernel after each iteration of the for loop in line 8 of Algorithm 8.

```

1: //  $\mathcal{C}[i] \in \mathcal{K} \in \kappa$ 
2: if  $\mathcal{K} = \{\mathcal{C}[i]\}$  then // a new kernel
3:   EXTQ( $\mathcal{K}, i + 1$ ) = ALLQUBITS
4: else
5:   EXTQ( $\mathcal{K}, i + 1$ ) = EXTQ( $\mathcal{K}, i$ )
6: for  $\mathcal{K}' \in (\kappa \setminus \mathcal{K})$  do
7:   if EXTQ( $\mathcal{K}', i$ ) = ALLQUBITS then
8:     if QUBITS( $\mathcal{C}[i]$ )  $\cap$  QUBITS( $\mathcal{K}'$ )  $\neq \emptyset$  then
9:       EXTQ( $\mathcal{K}', i + 1$ ) = QUBITS( $\mathcal{K}'$ )  $\setminus$  QUBITS( $\mathcal{C}[i]$ )
10:    else
11:      EXTQ( $\mathcal{K}', i + 1$ ) = ALLQUBITS
12:    else
13:      EXTQ( $\mathcal{K}', i + 1$ ) = EXTQ( $\mathcal{K}', i$ )  $\setminus$  QUBITS( $\mathcal{C}[i]$ )

```

be a gate $\mathcal{C}[j] \notin \mathcal{K}' \downarrow i$ satisfying $j < i$ and $\text{QUBITS}(\mathcal{C}[j]) \cup \text{QUBITS}(\mathcal{K}' \downarrow j) \neq \emptyset$, causing all qubits in this intersection to be not extensible for $\mathcal{K} \downarrow i$ by weak convexity (picking $j_2 = j$). So if the condition in line 7 of Algorithm 9 holds, i.e., all qubits are extensible for $\mathcal{K}' \downarrow i$, then monotonicity does not apply to $\mathcal{K}' \downarrow i$.

Case 1. If the condition in line 8 of Algorithm 9 does not hold, i.e., $\text{QUBITS}(\mathcal{C}[i]) \cap \text{QUBITS}(\mathcal{K}' \downarrow (i + 1)) = \emptyset$, then monotonicity does not apply to $\mathcal{K}' \downarrow (i + 1)$. Consider the additional constraint of weak convexity for $\mathcal{K}' \downarrow (i + 1)$ than weak convexity for $\mathcal{K}' \downarrow i$, that is, $\forall j_1 < j_2 = i$, if $\mathcal{C}[j_1] \in \mathcal{K}' \downarrow (i + 1)$, $\mathcal{C}[i] \notin \mathcal{K}' \downarrow (i + 1)$, then $q \notin \text{QUBITS}(\mathcal{C}[j_1]) \cap \text{QUBITS}(\mathcal{C}[i])$. However, $\text{QUBITS}(\mathcal{C}[j_1]) \cap \text{QUBITS}(\mathcal{C}[i]) = \emptyset$, so $\text{EXTQ}(\mathcal{K}', i + 1) = \text{ALLQUBITS}$.

Case 2. If the condition in line 8 of Algorithm 9 holds, i.e., $\text{QUBITS}(\mathcal{C}[i]) \cap \text{QUBITS}(\mathcal{K}' \downarrow (i + 1)) \neq \emptyset$, then monotonicity applies to $\mathcal{K}' \downarrow (i + 1)$ because $\mathcal{C}[i] \notin \mathcal{K}' \downarrow (i + 1)$ and $\mathcal{C}[i]$ shares a qubit with $\mathcal{K}' \downarrow (i + 1)$. So only qubits in $\text{QUBITS}(\mathcal{K}' \downarrow (i + 1))$ can possibly be extensible.

$\forall q \in \text{QUBITS}(\mathcal{K}' \downarrow (i + 1))$, q is extensible if and only if it satisfies weak convexity. Because all qubits were extensible for $\mathcal{K}' \downarrow i$, we only need to consider the constraint on $\forall j_1 < j_2 = i$. Because $q \in \text{QUBITS}(\mathcal{K}' \downarrow (i + 1))$, we can always find some $j_1 < i$ satisfying $q \in \text{QUBITS}(\mathcal{C}[j_1])$ and $\mathcal{C}[j_1] \in \mathcal{K}' \downarrow (i + 1)$, so q is extensible if and only if $q \notin \text{QUBITS}(\mathcal{C}[i])$. Therefore, $\text{EXTQ}(\mathcal{K}', i + 1) = \text{QUBITS}(\mathcal{K}' \downarrow (i + 1)) \setminus \text{QUBITS}(\mathcal{C}[i])$.

Case 3. If the condition in line 7 of Algorithm 9 does not hold, i.e., not all qubits are extensible for $\mathcal{K}' \downarrow i$, we know that line 9 has been executed for some $i' < i$ (so monotonicity applies to $\mathcal{K}' \downarrow (i' + 1)$), so $\text{EXTQ}(\mathcal{K}', i' + 1) \subseteq \text{QUBITS}(\mathcal{K}' \downarrow i')$. Because line 13 of Algorithm 9 ensures $\text{EXTQ}(\mathcal{K}', i'' + 1) \subseteq \text{EXTQ}(\mathcal{K}', i'')$ for all $i' < i'' < i$, we have $\text{EXTQ}(\mathcal{K}', i) \subseteq \text{QUBITS}(\mathcal{K}' \downarrow i') \subseteq \text{QUBITS}(\mathcal{K}' \downarrow i)$.

For each qubit q , if it is not in $\text{EXTQ}(\mathcal{K}', i)$, it cannot be in $\text{EXTQ}(\mathcal{K}', i + 1)$ because Definition 8 is more restrictive when i increases; for each $q \in \text{EXTQ}(\mathcal{K}', i)$, $q \in \text{EXTQ}(\mathcal{K}', i + 1)$ if and only if it satisfies the additional weak convexity constraint $\forall j_1 < j_2 = i$. Because $\text{EXTQ}(\mathcal{K}', i) \subseteq \text{QUBITS}(\mathcal{K}' \downarrow i)$, we can always find $j_1 < i$ such that $q \in \text{QUBITS}(\mathcal{C}[j_1])$ and

$\mathcal{C}[j_1] \in \mathcal{K} \downarrow i$, so q is extensible if and only if $q \notin \text{QUBITS}(\mathcal{C}[i])$. Therefore, we conclude that $\text{EXTQ}(\mathcal{K}', i+1) = \text{EXTQ}(\mathcal{K}', i) \setminus \text{QUBITS}(\mathcal{C}[i])$.

In all of these cases, we have proved the correctness of $\text{EXTQ}(\mathcal{K}', i+1)$ for any $\mathcal{K}'$. So Algorithm 9 correctly computes the extensible qubit set for each kernel. $\square$

Because we cannot add any gates to kernels with an empty extensible qubit set such as $\mathcal{K}_2$, we compute their cost and remove them from κ in line 13 of Algorithm 8. Because line 9 of Algorithm 9 restricts the extensible qubit set to be a subset of the qubit set of $\mathcal{K}'$, we no longer need to keep track of the qubit set because it will no longer change.

Theorem 12 (Time complexity of the KERNELIZE algorithm). *With the extensible qubit sets maintained in Algorithm 9, the KERNELIZE algorithm runs in $O\left(\left(\frac{3.2n}{\ln n}\right)^n \cdot |\mathcal{C}|\right)$, where n is the number of qubits.*

Proof. For each iteration i in Algorithm 8, let us compute the maximum number of kernel sets κ that Atlas must consider.

We observe that if we consider the qubit set for each kernel with the extensible qubit set being all qubits, and the extensible qubit set for other kernels, any two of these sets do not intersect. This can be proven by induction on i and the fact that whenever we add $\text{QUBITS}(\mathcal{C}[i])$ to $\mathcal{K}$, we effectively remove these qubits from all other sets.

Because kernels with empty extensible qubit sets are removed from κ , there are at most n kernels in κ and the number of ways to partition all qubits into disjoint qubit sets is at most the Bell number B_{n+1} , which is less than $\left(\frac{0.792(n+1)}{\ln(n+2)}\right)^{n+1}$ [25]. Each qubit set can be either the qubit set or the extensible qubit set of a kernel, so we need to multiply this number by at most 2^n ; each kernel can be either fusion or shared-memory, so we need to multiply this number by another 2^n . With the extensible qubit sets maintained, the order of kernels does not affect the DP transitions, so the number of different kernel sets is at most $4^n B_{n+1}$. Each iteration of the for loop in line 8 of Algorithm 8 takes polynomial time, so the time per iteration i is at most $O\left(\left(\frac{3.2n}{\ln n}\right)^n\right)$. There are $|\mathcal{C}|$ iterations, so the total time complexity is $O\left(\left(\frac{3.2n}{\ln n}\right)^n \cdot |\mathcal{C}|\right)$. $\square$

We also implemented other optimizations in the KERNELIZE algorithm. See Section 7.5.3 for details.

7.5.2 Cost Function in KERNELIZE

The cost function used in Equation (7.11) should faithfully represent the running time of each kernel. Inspired by HyQuas [157], we use two approaches to execute each kernel:

1. *Fusion*: Fuse all gates in a kernel into a single gate by pre-computing the product of the gate matrices and executing that single gate using cuQuantum [22]. The cost function maps the number of qubits in the kernel to a constant, reflecting the running time of a matrix multiplication of that size.
2. *Shared-memory*: Load the state vector into GPU shared memory in batches and execute the gates one by one instead of fusing them to a single matrix.³ The cost function is

³Shared-memory kernels correspond to SHM-GROUPING in HyQuas. Similar to HyQuas, to improve the I/O

Table 7.1: Our benchmark circuits and their size (number of gates).

Circuit Name	Description	Number of qubits								
		28	29	30	31	32	33	34	35	36
ae	amplitude estimation	514	547	581	616	652	689	727	766	806
dj	Deutsch–Jozsa algorithm	82	85	88	91	94	97	100	103	106
ghz	GHZ state	28	29	30	31	32	33	34	35	36
graphstate	graph state	56	58	60	62	64	66	68	70	72
ising	Ising model	302	313	324	335	346	357	368	379	390
qft	quantum Fourier transform	406	435	465	496	528	561	595	630	666
qpeexact	exact quantum phase estimation	432	463	493	524	559	593	628	664	701
qsvm	quantum support vector machine	274	284	294	304	314	324	334	344	354
su2random	SU2 ansatz with random parameters	1246	1334	1425	1519	1616	1716	1819	1925	2034
vqc	variational quantum classifier	1873	1998	2127	2260	2397	2538	2683	2832	2985
wstate	W state	109	113	117	121	125	129	133	137	141

$\alpha + \sum_{g \in \mathcal{K}} \text{COST}(g)$, where α is a constant that corresponds to the time to load a micro-batch of state coefficients into GPU shared memory, and $\text{COST}(g)$ is the time to simulate the application of the gate g in the GPU shared memory.

The way to determine these constant values is described in Section 7.6.1.

In addition to the qubit set and the extensible qubit set, we add a Boolean tag for the type of each kernel (fusion or shared-memory) in `KERNELIZE`, and maintain the cost of the kernel during the DP algorithm according to the type. Whenever a new kernel is created, we generate two copies of DP states with the new kernel’s type being fusion or shared-memory correspondingly.

7.5.3 Optimizations to the `KERNELIZE` Algorithm

Here we describe the optimizations when implementing Algorithm 8 with Algorithm 9.

Insular qubits Recall that the Atlas circuit staging algorithm allows mapping the insular qubits of a gate to regional and global qubits. Before loading a batch of states to GPU, the values of all global and regional qubits for the batch are known, which are leveraged by Atlas to simplify the computation. Specifically, for gates with insular non-local qubits, Atlas retrieves the non-local qubits’ values and converts these gates into local gates with fewer qubits.

We can do more along this line for shared-memory kernels. A physical qubit is *active* for a shared-memory kernel if its states are loaded into the shared memory in a single micro-batch. For *non-active* local qubits, Atlas applies a similar optimization to gates with insular non-active local qubits that converts these gates into smaller gates with only active qubits. Because we can swap two adjacent gates if all qubits they share are insular to both gates, we can slightly lift the Constraint 1 by ignoring any qubit in line 8 of Algorithm 9 that is insular to $\mathcal{C}[i]$ and all gates in $\mathcal{K}'$. In order to know this, we additionally maintain an active qubit set for each shared-memory kernel $\mathcal{K}'$ for the non-insular qubits of all gates in $\mathcal{K}'$.

efficiency of shared-memory kernels, we require the three least significant qubits of the state vector to be in all shared-memory kernels. As a result, each state vector load consists of at least $2^3 = 8$ complex numbers (i.e., 128 bytes as each complex number consists of 2 double-precision floating-point numbers).

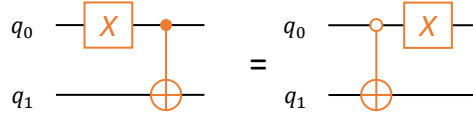


Figure 7.5: If we swap an X gate with an adjacent $CNOT$ gate, we need to flip the control state of the new $CNOT$ gate to $|0\rangle$.

Furthermore, we additionally maintain an extensible insular qubit set that is maintained in the same way as the extensible qubit set in line 13 of Algorithm 9 but initialized as

$$\text{EXTENSIBLEINSULARQUBITS}(\mathcal{K}', i + 1) = \text{ALLQUBITS} \setminus \text{QUBITS}(\mathcal{C}[i])$$

in line 9 of Algorithm 9. We can then further lift the Constraint 1 by allowing to add a gate $\mathcal{C}[i]$ to a shared-memory kernel $\mathcal{K}'$ as long as its insular qubits are in the extensible insular qubit set and its non-insular qubits are in the extensible qubit set.

After the DP algorithm, we reconstruct the topological order to see which gates are swapped. If one of the gates being swapped is anti-diagonal (e.g., the X gate) and the other gate is a control gate, Atlas flips the control state of that gate, as shown in Figure 7.5.

This optimization opens up more ways to reorder the circuit sequence.

Optimizing DP transition When a new gate $\mathcal{C}[i]$ can be subsumed by an existing kernel $\mathcal{K}$ or vice versa, i.e., $\text{QUBITS}(\mathcal{C}[i]) \subseteq \text{QUBITS}(\mathcal{K})$ or $\text{QUBITS}(\mathcal{K}) \subseteq \text{QUBITS}(\mathcal{C}[i])$ (as well as $\text{QUBITS}(\mathcal{C}[i]) \subseteq \text{EXTQ}(\mathcal{K})$ by Constraint 1), Atlas directly adds $\mathcal{C}[i]$ into $\mathcal{K}$. In other words, Atlas will not consider any κ where $\mathcal{C}[i] \in \mathcal{K}' \in \kappa$ for any $\mathcal{K}' \neq \mathcal{K}$. For example, in Figure 7.4, Atlas will directly add g_7 into $\mathcal{K}_4$ without considering other options. This optimization greatly reduces the amount of DP transitions Atlas must explore.

Deferring adding gates to extensible kernels To further reduce the number of DP states, we keep gates as single-gate kernels when their extensible qubit sets are all qubits instead of trying to add them to each of the kernels. Upon running line 9 of Algorithm 9 for $\mathcal{K}'$, we search for all other kernels $\mathcal{K}''$ with $\text{EXTQ}(\mathcal{K}'', i) = \text{ALLQUBITS}$ to be merged together with $\mathcal{K}'$ at that time.

For example, after adding g_7 into $\mathcal{K}_4$ in Figure 7.4, we create a singleton kernel $\mathcal{K}_5$ for g_8 . In the next iteration, we find that both $\mathcal{K}_1$ and $\mathcal{K}_5$ are subsumed by g_9 , so we pick one arbitrarily to add g_9 into it. Suppose we add g_9 into $\mathcal{K}_5$, then we will run line 9 of Algorithm 9 for $\mathcal{K}_1$. Before running this line, we search for all other kernels with the extensible qubit set being all qubits, i.e., $\mathcal{K}_4$ and $\mathcal{K}_5$. So we will either leave $\mathcal{K}_1$ alone or merge it with $\mathcal{K}_4$ or $\mathcal{K}_5$, resulting in different κ 's. In this particular example, merging $\mathcal{K}_1$ with $\mathcal{K}_5$ will be the best option.

Attaching single-qubit gates Our preliminary experiments show that the single-qubit gates of a quantum circuit can easily result in an exponential blow-up in the number of DP states because of the independence between single-qubit gates applied to different qubits. To address this issue, we attach each single-qubit gate to an adjacent multi-qubit gate. If the single-qubit gate is non-insular, we mark the corresponding qubit non-insular in the attached gate. For single-qubit gates

that are only adjacent to other single-qubit gates, we attach them together to the closest multi-qubit gate.

Post-processing After the DP algorithm, Atlas needs to merge some kernels together in κ before taking the cost as a post-processing step. Computing the minimum cost for merging kernels is NP-hard because the bin packing problem can be reduced to this problem. Atlas uses a simple approximation algorithm that greedily packs the kernels in κ , running in quasilinear time: (1) for fusion kernels, Atlas packs them towards the most cost-efficient density; (2) for shared-memory kernels, Atlas packs them towards the maximum shared-memory kernel size.

Pruning DP states Another key optimization to reduce the DP states is *pruning*. A simple pruning based on the cost of the DP states would be suboptimal because we have not considered the cost of the kernels in κ which may contribute a lot. During the DP, whenever the number of states at the same position i in the circuit reaches a user-defined threshold T at the beginning of an iteration (i.e., Line 7 in Algorithm 8), Atlas performs the post-processing step introduced above at the current position for all DP states and keeps $\frac{T}{2}$ DP states with the lowest cost after post-processing. This is the only optimization that may worsen the results. A study of the effect of T is included in Section 7.6.5. We set $T = 500$ in our evaluation, which achieves good performance.

7.5.4 An Alternate Algorithm for Kernelization

Algorithm 10 The ORDEREDKERNELIZE Algorithm. The “+” operator in line 7 means adding the cost and appending the kernel to the sequence.

```

1: function ORDEREDKERNELIZE( $\mathcal{C}$ )
2:   Input: A quantum circuit  $\mathcal{C}$  represented as a sequence of gates.
3:   Output: A sequence of kernels.
4:   //  $DP[i]$  stores the minimum cost to kernelize the first  $i$  gates and the corresponding kernel sequence.
5:    $DP[0] = (0, [])$ 
6:   for  $i = 0$  to  $|\mathcal{C}| - 1$  do
7:      $DP[i + 1] = \min_{0 \leq j \leq i} \{DP[j] + \text{Cost}(\mathcal{C}[j, \dots, i])\}$ 
8:   return  $DP[|\mathcal{C}|].kernels$ 

```

As KERNELIZE is complicated, we present Algorithm 10, an alternate dynamic programming (DP) algorithm that also solves the optimal circuit kernelization problem, considering only kernels with contiguous gate segments. In Theorem 13, we prove that the KERNELIZE algorithm is at least as good as the alternate algorithm, called ORDEREDKERNELIZE, and evaluate the empirical difference in Section 7.6.5.

ORDEREDKERNELIZE takes the circuit representation as a sequence of gates and outputs a sequence of kernels. The variable DP maps a gate (position) in the circuit to a pair of the total cost for the circuit prefix up to that gate (excluding the gate) and the sequence of kernels

for that prefix. The algorithm starts with a total cost of zero and an empty kernel sequence. It then considers each possible gate $\mathcal{C}[i]$ in increasing order of i and computes for that gate the optimal cost. To compute the optimal cost at a position, ORDEREDKERNELIZE examines the cost of all possible kernels ending at that position and takes the minimum. Because for each gate, the algorithm considers all possible kernels ending at that gate, its run-time is asymptotically bounded by the square of the length of the circuit, i.e., $O(|\mathcal{C}|^2)$.

Although ORDEREDKERNELIZE is easier to implement, it has a significant limitation: It only considers a specific sequential ordering of the circuit. It is a priori difficult to determine an ordering that yields the lowest cost. For example, the optimal kernel sequence for Figure 7.4 might be $\{g_0, g_8, g_9\}, \{g_1, g_2, g_4\}, \{g_3\}, \{g_5, g_6, g_7\}$. ORDEREDKERNELIZE cannot find this kernel sequence if it is not provided with the corresponding ordering of the circuit. We show that the KERNELIZE algorithm is better than ORDEREDKERNELIZE both theoretically (Theorem 13) and empirically (Figure 7.13).

Theorem 13 (The KERNELIZE algorithm is at least as good as ORDEREDKERNELIZE). *Algorithm 8 always returns a kernel sequence with a total execution cost no more than Algorithm 10, which is optimal for Problem 1.*

Proof. Let $\mathcal{K}_0, \dots, \mathcal{K}_{b-1}$ be the kernel sequence that minimizes the total execution cost such that each kernel consists of a contiguous segment of the gate sequence $\mathcal{C}$. It suffices to prove that Atlas kernelization algorithm returns a kernel sequence with a total execution cost no more than Equation (7.11).

For any $0 \leq i < |\mathcal{C}|$, suppose $\mathcal{C}[i] \in \mathcal{K}_t$ ($0 \leq t < b$). We prove by induction on i that for any $0 \leq i < |\mathcal{C}|$, there is a kernel set κ_{i+1} such that

1. $\{\mathcal{K}_t \downarrow (i+1)\} \subseteq \kappa_{i+1} \subseteq \{\mathcal{K}_0, \dots, \mathcal{K}_{t-1}, \mathcal{K}_t \downarrow (i+1)\}$
2. and after the iteration i (0-indexed) of the loop in line 7 of Algorithm 8, $DP[i+1, \kappa_{i+1}]$ has a cost no more than $\text{COST}(\{\mathcal{K}_0, \dots, \mathcal{K}_{t-1}, \mathcal{K}_t \downarrow (i+1)\} \setminus \kappa_{i+1})$.

When $i = 0$, line 13 transits from $DP[0, \emptyset] = (0, [])$ to $DP[1, \{\mathcal{C}[0]\}] = (0, [])$ where $\kappa_1 = \kappa = \{\mathcal{C}[0]\}$, $\kappa' = \tilde{\kappa} = \emptyset$, so the induction hypothesis holds.

When $i > 0$, $\mathcal{K}_t \downarrow (i+1)$ satisfies Constraint 1 by Theorem 10. If $|\mathcal{K}_t \downarrow (i+1)| > 1$, we apply the induction hypothesis on $i-1$ and it holds for i because line 11 sets $DP[i+1, \kappa_{i+1}]$ to $DP[i, \kappa_i]$ in the iteration where $\kappa_{i+1} = \kappa = \kappa_i \setminus \{\mathcal{K}_t \downarrow i\} \cup \{\mathcal{K}_t \downarrow (i+1)\}$. If $|\mathcal{K}_t \downarrow (i+1)| = 1$, consider the transition in line 13 in the iteration where $\kappa_{i+1} = \kappa = \{\mathcal{K}_t \downarrow (i+1)\}$. After taking the minimum when $\kappa' = \kappa_i$,

$$\begin{aligned}
& DP[i+1, \kappa_{i+1}].\text{cost} \\
& \leq DP[i, \kappa_i].\text{cost} + \text{COST}(\kappa_i) \\
& \leq \text{COST}(\{\mathcal{K}_0, \dots, \mathcal{K}_{t-2}, \mathcal{K}_{t-1} \downarrow i\} \setminus \kappa_i) + \text{COST}(\kappa_i) \\
& = \text{COST}(\{\mathcal{K}_0, \dots, \mathcal{K}_{t-1}\}) \\
& = \text{COST}(\{\mathcal{K}_0, \dots, \mathcal{K}_{t-1}, \mathcal{K}_t \downarrow (i+1)\} \setminus \kappa_{i+1}).
\end{aligned}$$

So after the last iteration of the loop in line 7, there is a kernel set $\kappa_{|\mathcal{C}|}$ such that $DP[|\mathcal{C}|, \kappa_{|\mathcal{C}|}]$

Table 7.2: Our benchmark circuits and their size (number of gates).

Circuit Name	Description	Number of qubits								
		28	29	30	31	32	33	34	35	36
ae	amplitude estimation	514	547	581	616	652	689	727	766	806
dj	Deutsch–Jozsa algorithm	82	85	88	91	94	97	100	103	106
ghz	GHZ state	28	29	30	31	32	33	34	35	36
graphstate	graph state	56	58	60	62	64	66	68	70	72
ising	Ising model	302	313	324	335	346	357	368	379	390
qft	quantum Fourier transform	406	435	465	496	528	561	595	630	666
qpeexact	exact quantum phase estimation	432	463	493	524	559	593	628	664	701
qsvm	quantum support vector machine	274	284	294	304	314	324	334	344	354
su2random	SU2 ansatz with random parameters	1246	1334	1425	1519	1616	1716	1819	1925	2034
vqc	variational quantum classifier	1873	1998	2127	2260	2397	2538	2683	2832	2985
wstate	W state	109	113	117	121	125	129	133	137	141

has a cost no more than $\text{COST}(\{\mathcal{K}_0, \dots, \mathcal{K}_{b-1}\} \setminus \kappa_{|C|})$, which evaluates to

$$\text{COST}(\{\mathcal{K}_0, \dots, \mathcal{K}_{b-1}\}) = \sum_{i=0}^{b-1} \text{COST}(\mathcal{K}_i)$$

and is taken minimum with the returned kernel sequence on line 14. $\square$

7.5.5 Atlas

We built our system Atlas on top of FlexFlow [71], a distributed multi-GPU system for DNN training. Atlas uses the mapper interface of FlexFlow to distribute the state vector across the DRAM and GPU device memory of a GPU cluster, and uses Legion [21], FlexFlow’s underlying task-based runtime, to launch simulation tasks on CPUs and GPUs and automatically overlap communications with computations. Furthermore, Atlas uses the NCCL library [101] to perform inter-GPU communication for sharding. We set the cost factor of the inter-node communication in the circuit staging algorithm $c = 3$ in Equation (7.1).

Atlas is publicly available as an open-source project [147] and also in the artifact supporting the paper [148].

7.6 Evaluation

7.6.1 Experimental Setup

We use the Perlmutter supercomputer [98] to evaluate Atlas. Each compute node is equipped with an AMD EPYC 7763 64-core 128-thread processor, 256 GB DRAM, and four NVIDIA A100-SXM4-40GB GPUs. The nodes are connected with HPE Slingshot 200 Gb/s interconnects. The programs are compiled using GCC 12.3.0, CUDA 12.2, and NCCL 2.19.4.

The cost function used in KERNELIZE has some constants that require benchmarking on the GPU. For fusion kernels, we measure their execution time with different numbers of qubits. For

shared-memory kernels, we measure the run time of an empty shared-memory kernel to estimate the cost of loading a state vector to GPU shared memory, and profile the run times for different types of gates using the GPU shared memory.

Benchmarks. We collect 11 types of scalable quantum circuits from the MQT Bench [115] and NWQBench [81], where we can select the desired number of qubits for each circuit type. This allows us to conduct a weak-scaling evaluation, where we can compare the performance of Atlas and existing GPU-based quantum circuit simulators using different numbers of GPUs. Table 7.1 summarizes the circuits used in our evaluation.

Preprocessing Circuits. To speedup simulation, we partition each circuit by (1) splitting it into stages using the STAGE algorithm, and (2) kernelizing each stage using the KERNELIZE algorithm. This preprocessing needs to be done once per circuit and finishes in 7.2 seconds on average for each circuit in a single thread on an Intel Xeon W-1350 @ 3.30GHz CPU. In our evaluation, we use the PuLP library [48] with the HiGHS solver [64] for ILP. The ILP-based STAGE algorithm takes 3.3 seconds on average, and KERNELIZE takes 3.9 seconds on average. A detailed analysis of the running time of KERNELIZE is included in Section 7.6.5.

7.6.2 End-to-end Performance

We first compare Atlas against existing distributed GPU-based quantum circuit simulators, including HyQuas [157], cuQuantum [22], and Qiskit [5]. UniQ [158] is extremely similar to HyQuas in state-vector-based GPU-based quantum circuit simulation, so we do not compare with UniQ. Both Qiskit and cuQuantum use the Qiskit Python interface as the frontend. cuQuantum uses Aer’s cuStateVec integration (*cusvaer*) to simulate quantum circuits, and Qiskit directly uses its native Aer GPU backend. All baselines perform state-vector-based simulation and directly store the entire state vector on GPUs. While Atlas also supports more scalable quantum circuit simulation by offloading the state vector to a much larger CPU DRAM, to conduct a fair comparison, in this experiment, we disable Atlas DRAM offloading and directly store the entire state vector on GPUs. We further evaluate Atlas DRAM offloading performance in Section 7.6.3.

Figure 7.6 shows the end-to-end simulation performance of the 11 circuit families on up to 256 GPUs (on 64 nodes). We use 28 local qubits and increase the number of non-local qubits from 0 (on 1 GPU) to 8 (on 256 GPUs). The number of regional qubits is at most 2 (there are 4 GPUs in each node). Qiskit scales poorly, so we only evaluate it on up to 4 GPUs and present a log-scale plot in Figure 7.6f.

Atlas is up to $20.2\times$ ($4.0\times$ on average) faster than HyQuas, $7.2\times$ ($3.2\times$ on average) faster than cuQuantum, and $2,126\times$ ($286\times$ on average) faster than Qiskit across all types of circuits and possible numbers of GPUs supported by the baselines. The speedups over existing systems are achieved by two important optimizations. First, the ILP-based Atlas circuit staging algorithm can discover a staging strategy that minimizes expensive inter-node communications, allowing Atlas to scale highly effectively as the number of GPUs increases. For example, to scale *graphstate* from 28 qubits (on 1 GPU) to 36 qubits (on 256 GPUs), HyQuas’ simulation time increases by

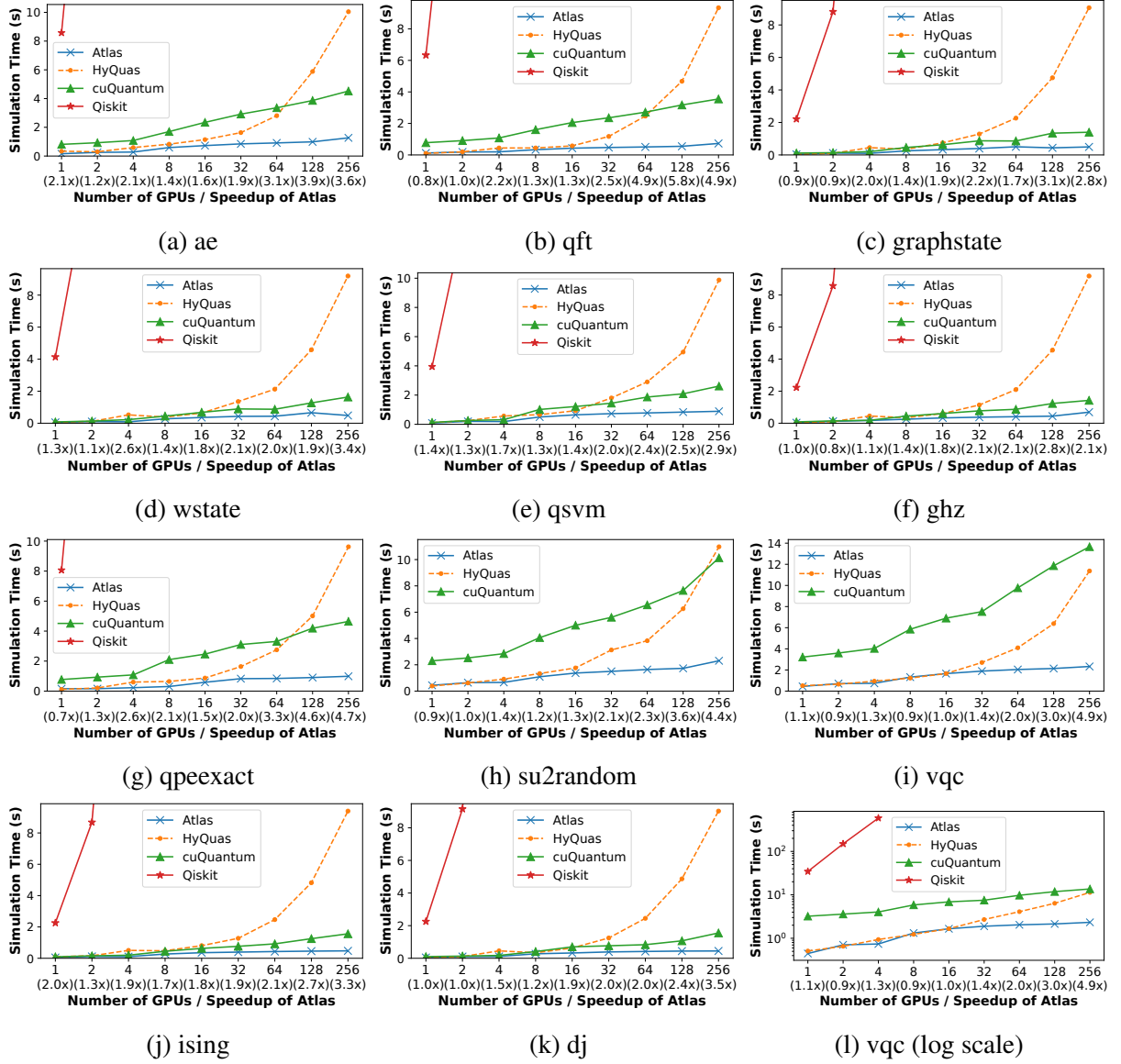


Figure 7.6: Weak scaling of Atlas, HyQuas, cuQuantum, and Qiskit with 28 local qubits as the number of global qubits increases from 0 (on 1 GPU) to 8 (on 256 GPUs). Qiskit scales poorly and usually does not fit into our charts.

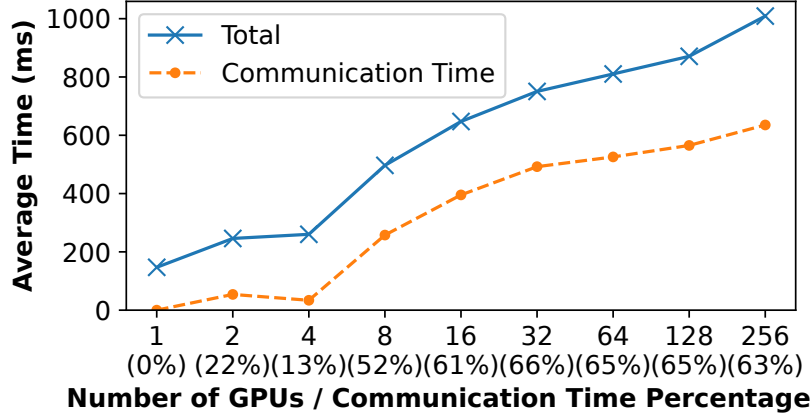


Figure 7.7: Simulation time breakdown: The average communication time and its percentage in the average total simulation time of Atlas.

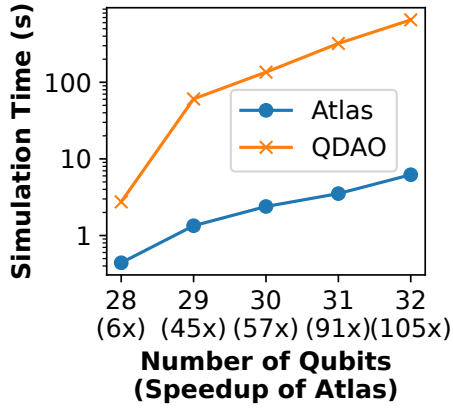


Figure 7.8: Atlas outperforms QDAO by $61\times$ on average. Log-scale simulation time (single GPU) with DRAM offloading for `qft` circuits.

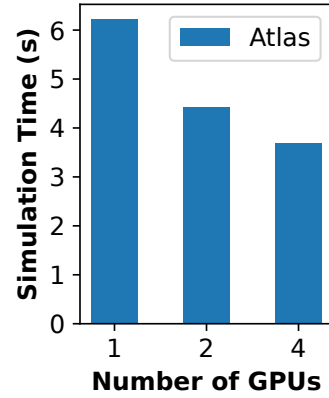


Figure 7.9: DRAM offloading scales. Atlas simulation time of a 32-qubit `qft` circuit with 1, 2, and 4 GPUs.

$267.9\times$, while the simulation time of Atlas only increases by $13.7\times$. Second, the DP-based `KERNELIZE` algorithm enables Atlas to achieve high-performance circuit performance on each GPU.

Figure 7.7 breaks down the total simulation time into communication time (including intra-node and inter-node communications) and computation time. We take the average of the 11 benchmark circuits for each number of GPUs. Inter-node communication is much more expensive than intra-node communication, so the computation is dominant when there is only one node (at most 4 GPUs). When the number of nodes increases, inter-node communication becomes dominant. We note that the communication time takes a greater portion with 2 GPUs than with 4 GPUs, because each pair of the 4 GPUs in the node is connected. The total bandwidth of each GPU increases by $3\times$ when the number of GPUs available increases from 2 to 4, so intra-node communication is faster with 4 GPUs than with 2 GPUs.

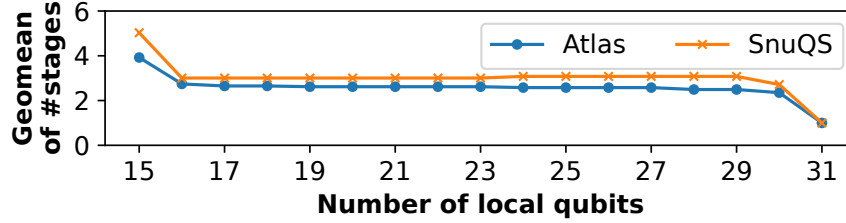


Figure 7.10: Number of stages, Atlas versus SnuQS: The geometric mean over all our benchmark circuits with 31 qubits.

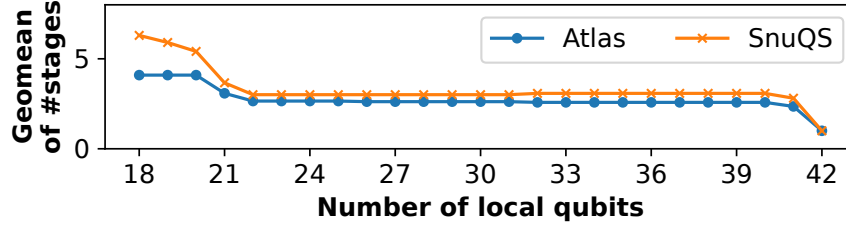


Figure 7.11: Number of stages, Atlas versus SnuQS: The geometric mean over all our benchmark circuits with 42 qubits.

7.6.3 DRAM Offloading

In addition to improving quantum circuit simulation performance, another key advantage of Atlas over existing systems is its ability to scale to larger circuits that go beyond the GPU memory capacity. As described in Section 7.2, Atlas does not require that the entire state vector of a quantum circuit be stored in GPU memory and uses DRAM offloading to support larger circuits. QDAO [162] also uses DRAM offloading to support larger circuits and scales to 32 qubits. We compare Atlas with QDAO with the Qiskit backend. We use 28 local qubits, and all remaining qubits are regional. For QDAO, we set $m = 28$, and $t = 19$ which runs the fastest.

Figure 7.8 shows the simulation performance of `qft` circuits with different numbers of qubits on a single GPU. For a 28-qubit circuit, the GPU memory is sufficient, so both Atlas and QDAO are fast; for circuits with 29 to 32 qubits, Atlas runs $74\times$ faster than QDAO on average, and also scales better. The speedups are also achieved by both the `STAGE` and the `KERNELIZE` algorithms.

Figure 7.9 shows the simulation performance of the 32-qubit `qft` circuit on 1, 2, and 4 GPUs. Atlas scales across multiple GPUs. The simulation time of QDAO stays the same when the number of GPUs increases.

7.6.4 Circuit Staging

We further compare the ILP-based circuit staging algorithm with heuristic-based approaches in existing simulators. In particular, we use the heuristics used in SnuQS as a baseline, which greedily selects the qubits with more gates operating on non-local gates to form a stage and uses the number of total gates as a tiebreaker [104]. Other works do not describe how the heuristics were implemented.

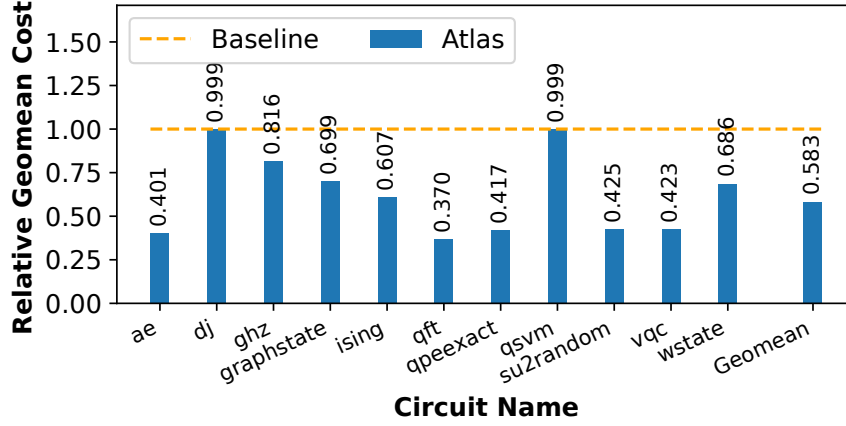


Figure 7.12: Kernelization effectiveness: The relative geometric mean cost of `KERNELIZE` compared to greedy packing up to 5 qubits.

Similar to Section 7.6.2, we set at most 2 non-local qubits to be regional, and all other non-local qubits to be global. Figures 7.10 and 7.11 show the geometric mean number of stages for all 11 circuits with 31 and 42 qubits in Table 7.2, accordingly. Our circuit staging algorithm is guaranteed to return the minimum number of stages by Theorem 8, and it always outperforms `SnuQS`’ approach. Note that `SnuQS` may give a worse circuit partition for the same circuit when the number of local qubits increases (from 23 to 24 in Figure 7.10 and from 31 to 32 in Figure 7.11), but this is guaranteed not to happen in our ILP-based approach.

7.6.5 Circuit Kernelization

We evaluate our `KERNELIZE` algorithm and compare it with a baseline that greedily packs gates into fusion kernels of up to 5 qubits, the most cost-efficient kernel size in the cost function used in Section 7.6.1. Recall that the cost faithfully represents the actual execution time of the kernels. The cost numbers are derived from GPU benchmarking, as detailed in Section 7.6.1.

Each circuit family exhibits a pattern, so we take the geometric mean of the cost for 9 circuits with the number of qubits from 28 to 36 for each circuit family. Figure 7.12 shows that the greedy baseline performs well in `dj` and `qsvm` circuits, but it does not generalize to other circuits. The `KERNELIZE` algorithm is able to exploit the pattern for each circuit and find a low-cost kernel sequence accordingly.

Results for each individual circuit. Figures 7.14 to 7.24 show the performance of `Atlas`’ kernelizer in terms of the cost for each circuit compared against a baseline greedily packing gates into fusion kernels of up to 5 qubits. “`Atlas-Naive`” denotes `ORDEREDKERNELIZE` described in Section 7.5.4. Figures 7.26 to 7.36 show the run time of these kernelization algorithms. This time is considered as preprocessing in Section 7.6.1, and excluded from Section 7.6.3.

Case study on a circuit with many gates. We additionally present a case study where the number of gates is far larger than the number of qubits. We take the `hh1` circuit in `NWQBench` [81]

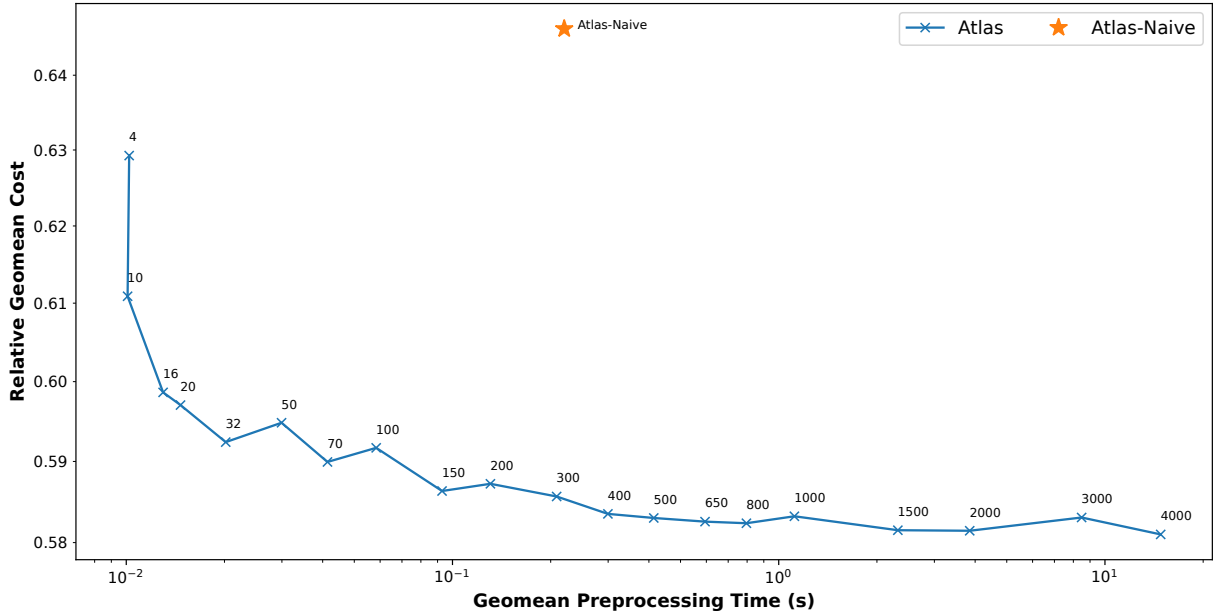


Figure 7.13: The relative geometric mean cost among all 99 circuits in Table 7.2 of `KERNELIZE` compared to greedy packing up to 5 qubits, and the running time of the kernelization algorithm when the pruning threshold T varies. Each data point is annotated with the pruning threshold T . “Atlas” denotes `KERNELIZE`. “Atlas-Naive” denotes `ORDEREDKERNELIZE`.

with 4, 7, 9, and 10 qubits, and pad each circuit to 28 qubits to restrict the kernelization algorithms to use GPU instead of computing the matrix for a 10-qubit gate directly. The number of gates is given in Table 7.3.

Table 7.3: The number of gates in the `hhl` circuit.

Number of qubits	Number of gates
4	80
7	689
9	91,968
10	186,795

Figure 7.25 shows the resulting cost and Figure 7.37 shows the preprocessing time of the kernelization algorithms. `KERNELIZE` runs in linear time to the number of gates, and it is faster than `ORDEREDKERNELIZE` on each of these circuits while achieving the same resulting cost.

Pruning threshold and running time of `KERNELIZE`. `KERNELIZE` implements a pruning threshold T in Section 7.5.3. As shown in Figure 7.13, the running time of `KERNELIZE` increases and the total execution cost decreases when T increases. The cost reduction benefit diminishes quickly and the running time of `KERNELIZE` grows exponentially when T increases exponentially; even when $T = 4$, `KERNELIZE` achieves a lower total execution cost than

ORDEREDKERNELIZE and also preprocesses faster. We choose $T = 500$ in our evaluation as the geometric mean cost starts to flatten and the running time of KERNELIZE (3.9 seconds on average) is on the same order of magnitude as the ILP-based circuit staging algorithm (3.3 seconds on average). Because some large circuits take a much longer time to preprocess than small circuits, the average preprocessing time is longer than the geometric mean. $T = 4,000$ or even higher would give us better results, but that would require a much longer circuit preprocessing time.

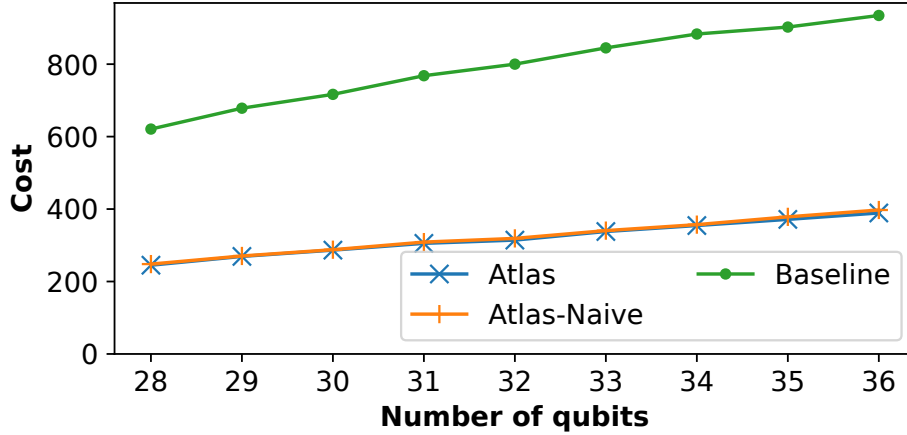


Figure 7.14: The performance of different kernelization algorithms on the circuit `ae`.

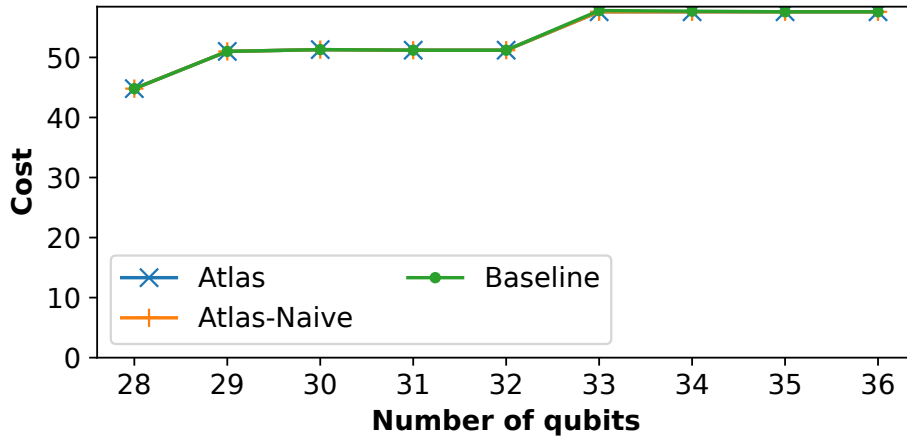


Figure 7.15: The performance of different kernelization algorithms on the circuit `dj`.

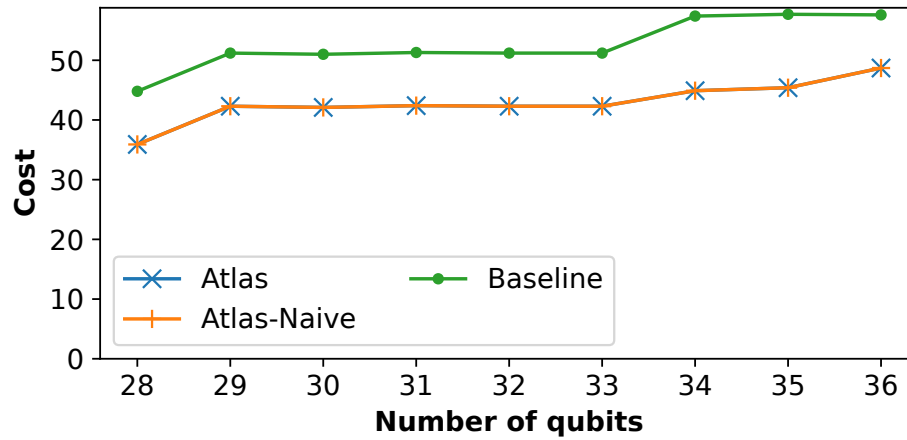


Figure 7.16: The performance of different kernelization algorithms on the circuit `ghz`.

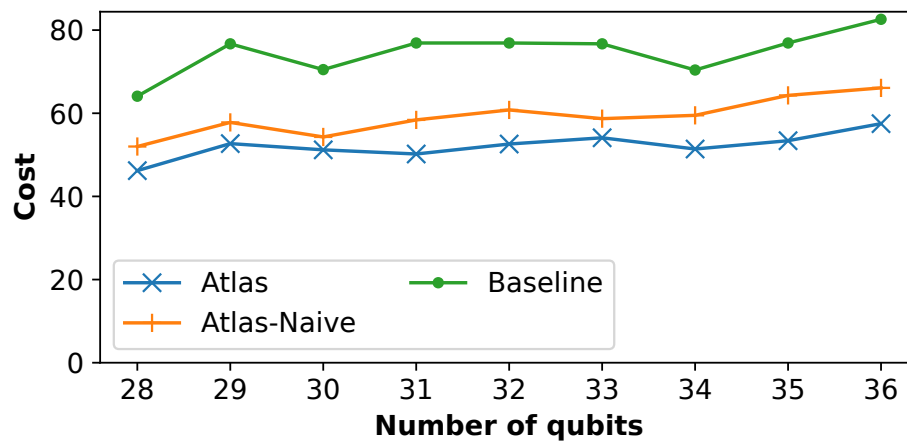


Figure 7.17: The performance of different kernelization algorithms on the circuit `graphstate`.

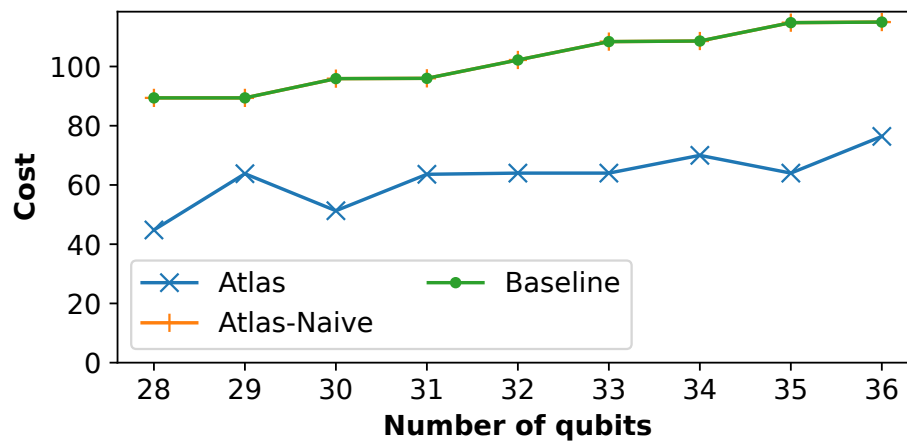


Figure 7.18: The performance of different kernelization algorithms on the circuit `ising`.

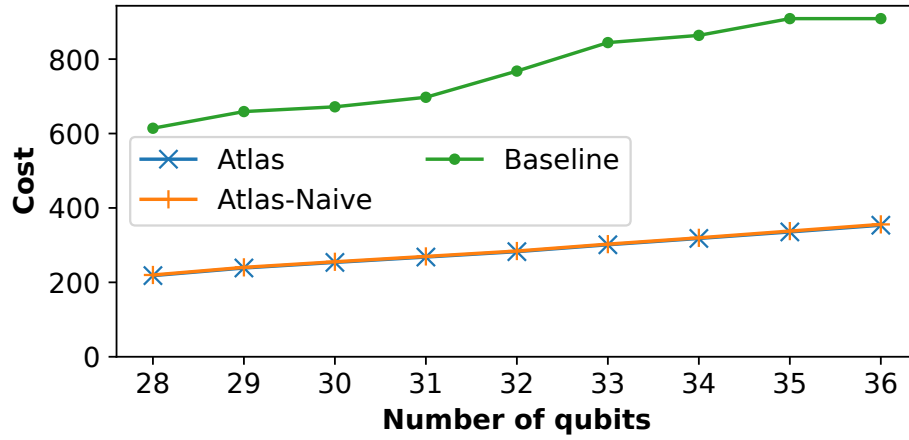


Figure 7.19: The performance of different kernelization algorithms on the circuit qft .

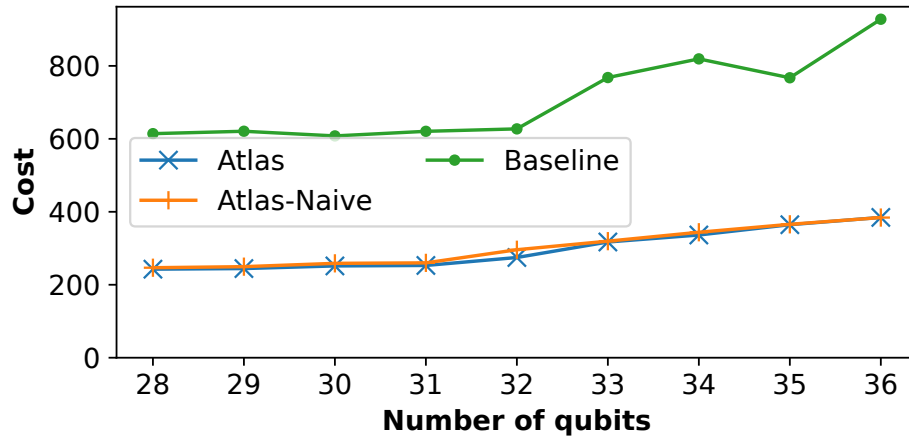


Figure 7.20: The performance of different kernelization algorithms on the circuit $qpeexact$.

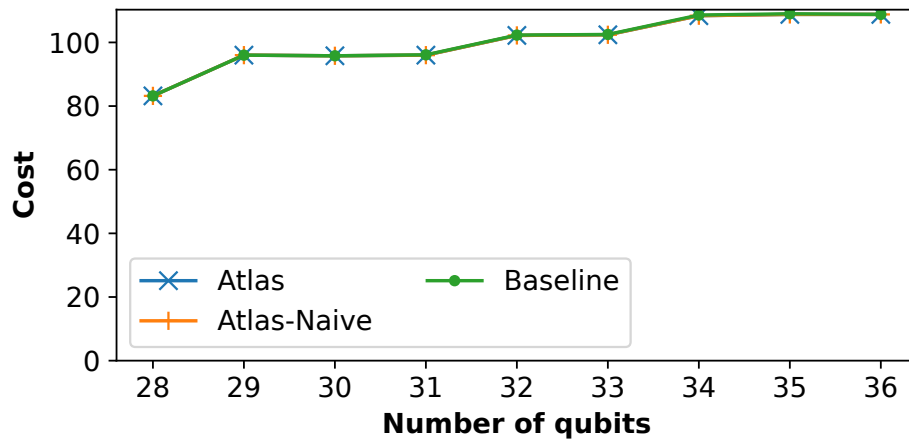


Figure 7.21: The performance of different kernelization algorithms on the circuit $qsvm$.

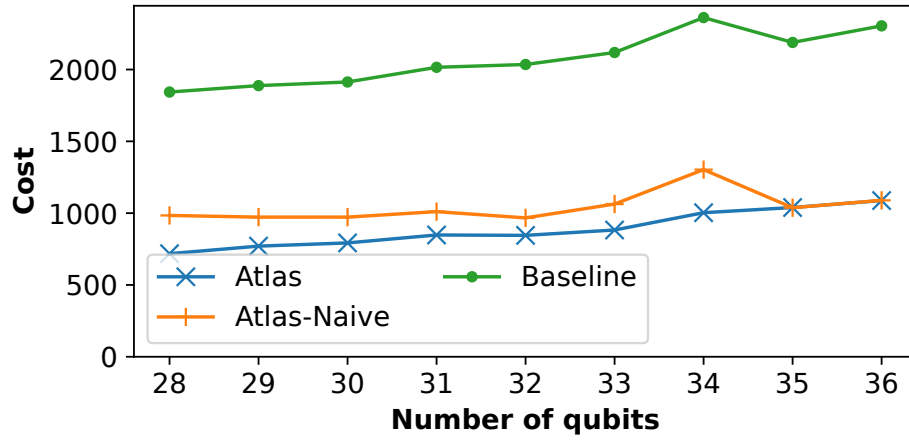


Figure 7.22: The performance of different kernelization algorithms on the circuit `su2random`.

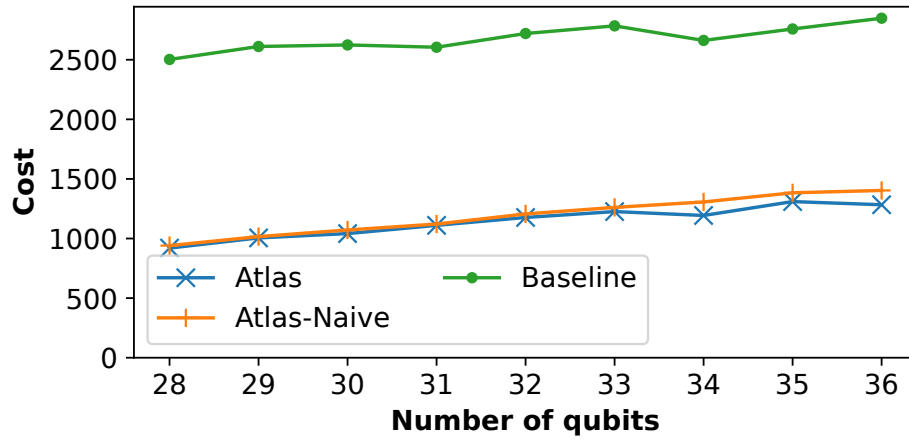


Figure 7.23: The performance of different kernelization algorithms on the circuit `vqc`.

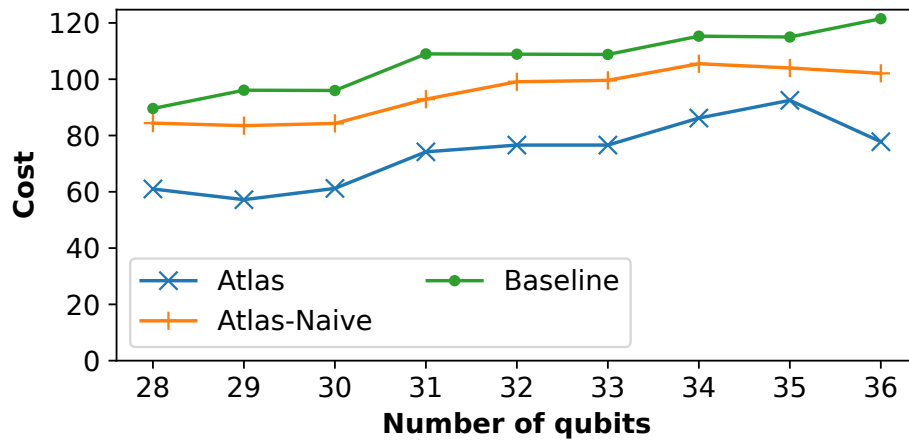


Figure 7.24: The performance of different kernelization algorithms on the circuit `wstate`.

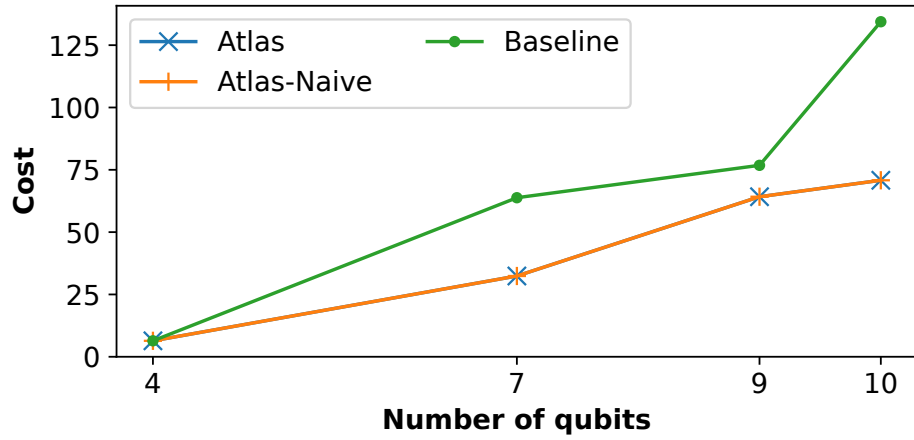


Figure 7.25: The performance of different kernelization algorithms on the circuit hh1.

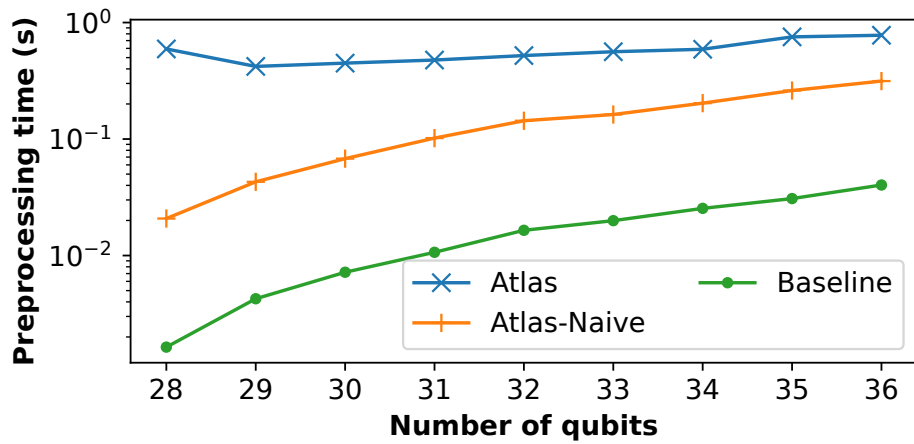


Figure 7.26: The run time of different kernelization algorithms on the circuit ae.

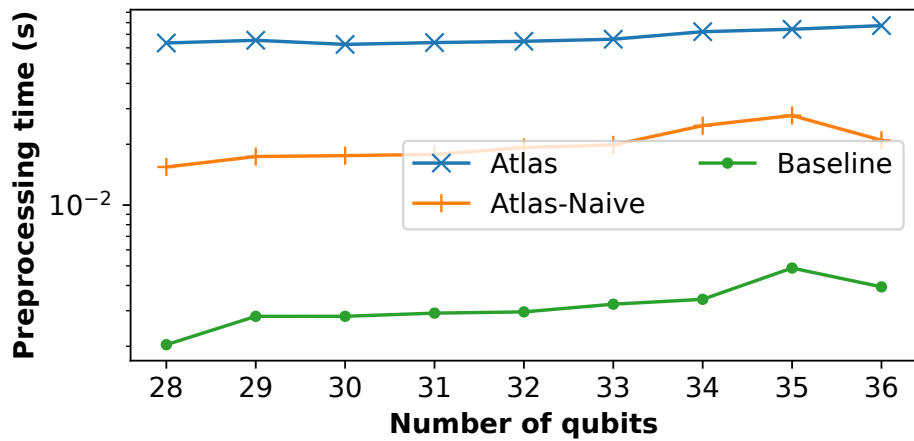


Figure 7.27: The run time of different kernelization algorithms on the circuit dj.

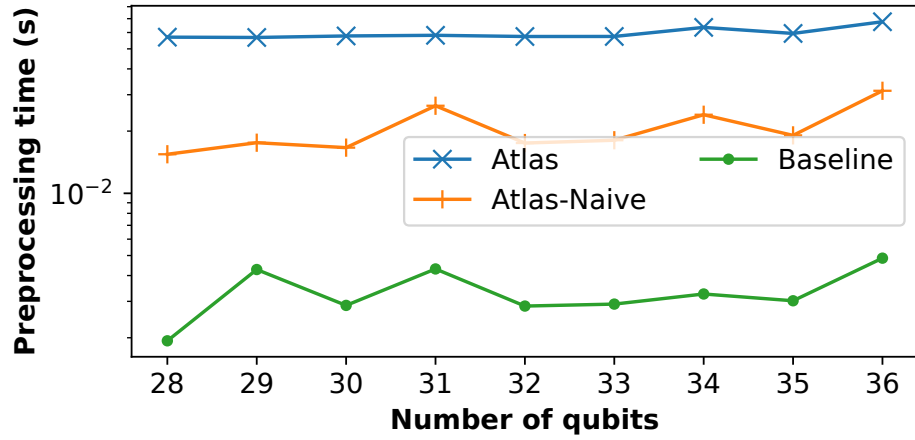


Figure 7.28: The run time of different kernelization algorithms on the circuit ghz.

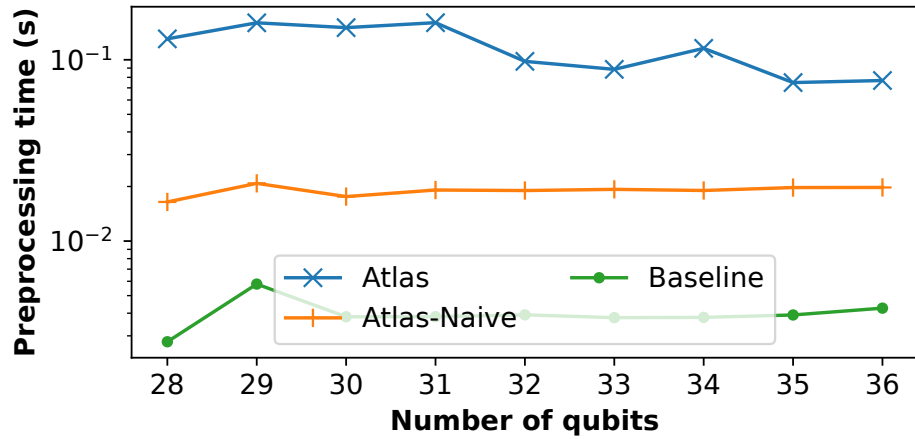


Figure 7.29: The run time of different kernelization algorithms on the circuit graphstate.

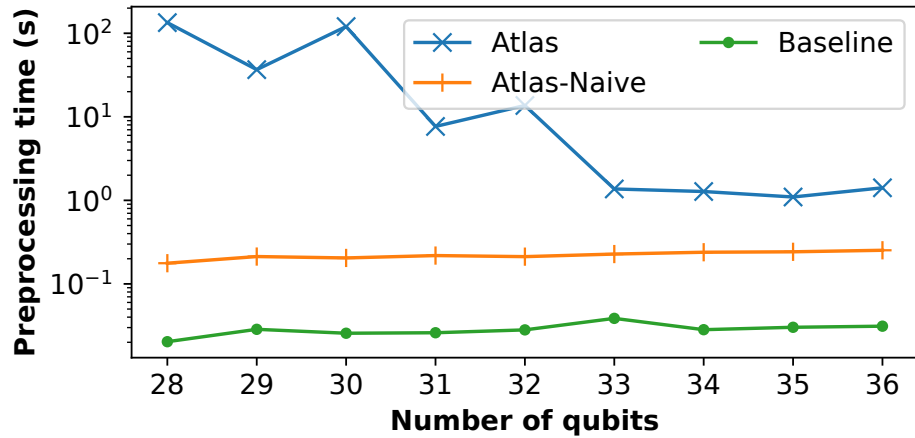


Figure 7.30: The run time of different kernelization algorithms on the circuit ising.

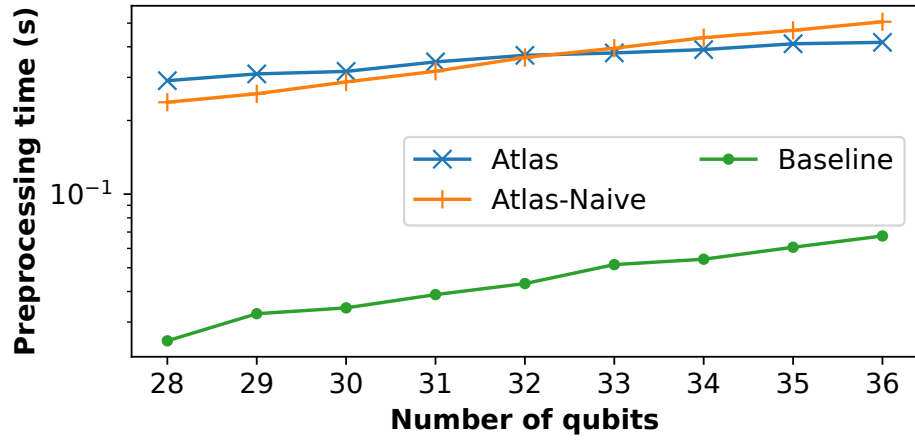


Figure 7.31: The run time of different kernelization algorithms on the circuit qft .

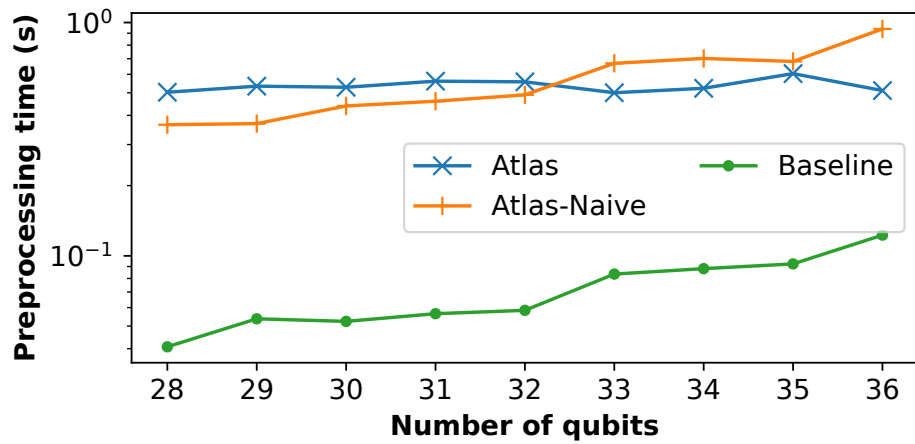


Figure 7.32: The run time of different kernelization algorithms on the circuit $qpeexact$.

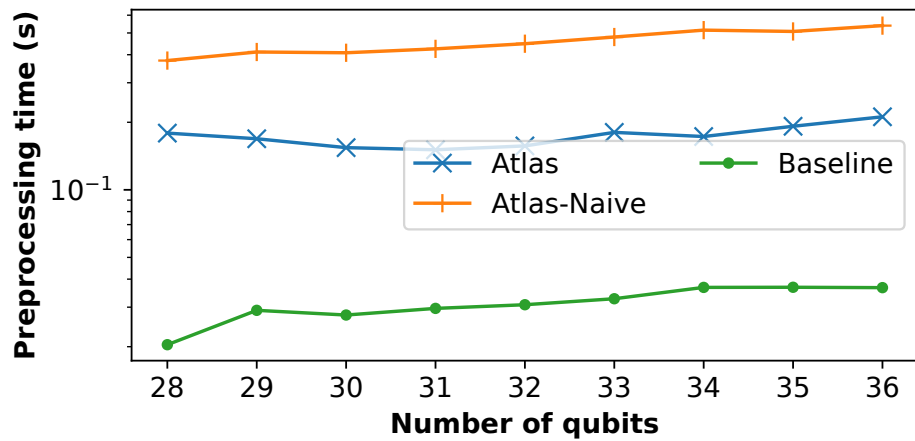


Figure 7.33: The run time of different kernelization algorithms on the circuit $qsvm$.

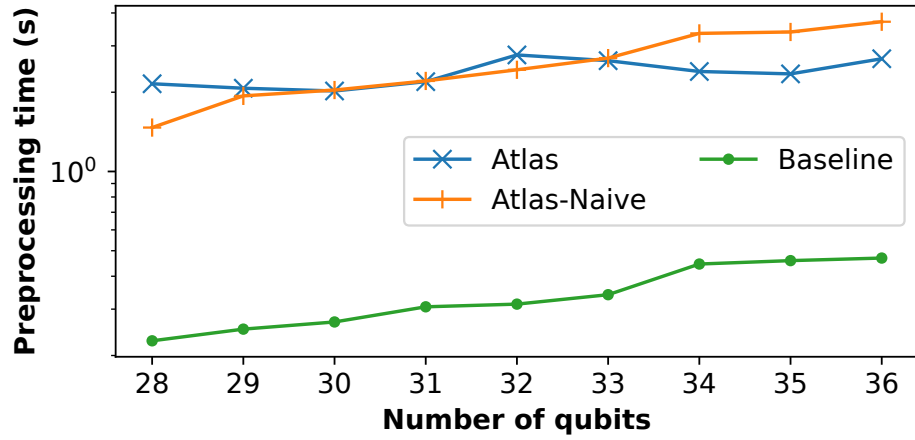


Figure 7.34: The run time of different kernelization algorithms on the circuit `su2random`.

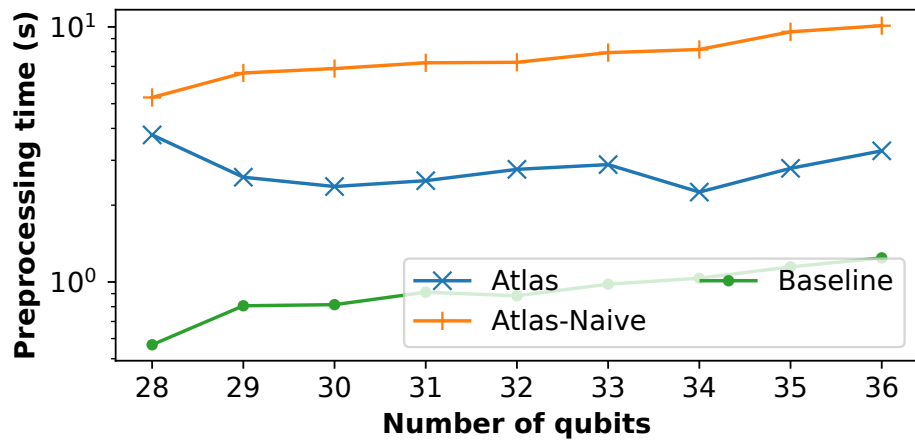


Figure 7.35: The run time of different kernelization algorithms on the circuit `vqc`.

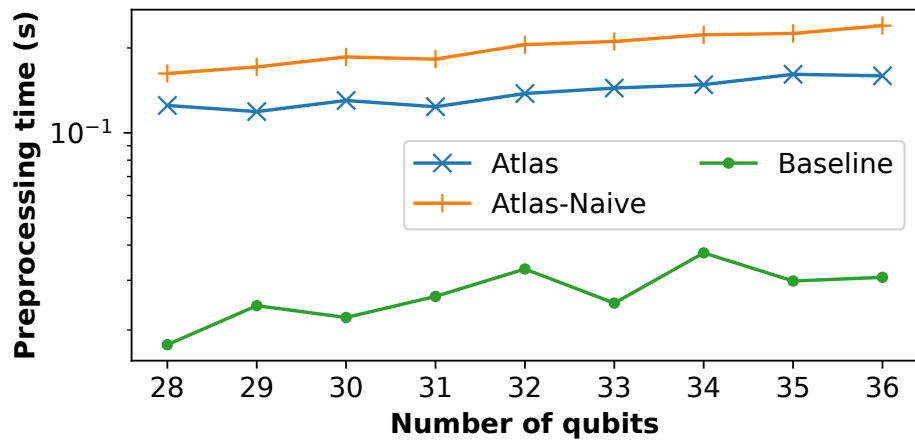


Figure 7.36: The run time of different kernelization algorithms on the circuit `wstate`.

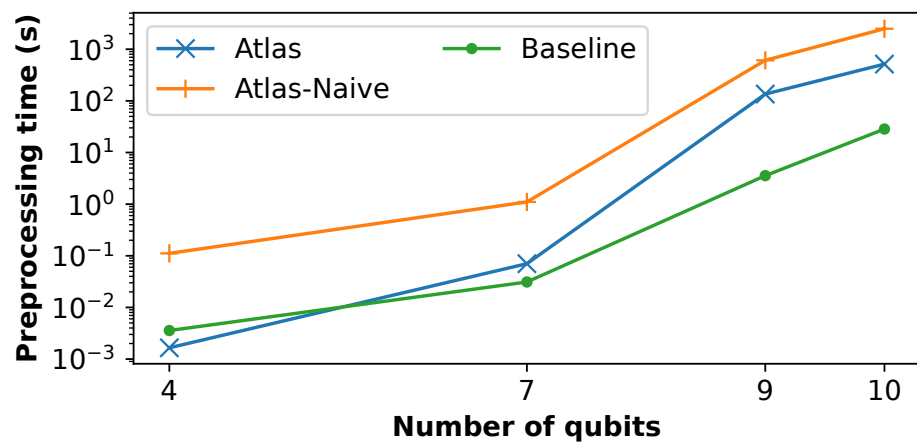


Figure 7.37: The run time of different kernelization algorithms on the circuit `hh1`.

Chapter 8

Related Work

This thesis includes contributions on both quantum circuit optimization and quantum circuit simulation. Rather than surveying prior work paper-by-paper, we group related work by the *recurring themes* that appear across our chapters: (i) compiler infrastructures and equivalence-based optimization, (ii) automated transformation discovery and verification, and (iii) high-performance simulation techniques. This structure makes explicit how each body of work relates to the corresponding thesis contributions and sets the stage for the specific compilation bottlenecks addressed by our frameworks.

8.1 Quantum Circuit Optimization

Quantum circuit optimizers span a range of design points, from general-purpose compiler stacks to domain-specific engines, and from expert-curated rewrite systems to automated rule discovery.

8.1.1 Compiler Infrastructures and Rule-Based Optimization

General-purpose toolchains such as Qiskit [6] and t|ket> [130] provide end-to-end compilation pipelines across multiple backends and gate sets, while Quilc [131] targets Rigetti processors. To reduce execution costs, these systems typically rely on rule-based optimizers that apply pre-defined circuit transformation rules in a manner analogous to peephole optimization in classical compilers.

Representative rule-based approaches include heuristics developed by Nam et al. [97], which systematically applies rotation merging and gate cancellation rules, and VOQC [61], which provides formally verified implementations of these rules. PyZX [76] takes a different approach by converting circuits to ZX-diagrams and applying ZX-calculus rewrite rules, though extracting optimized gate-based circuits from ZX-diagrams is computationally expensive [44]. Phase-polynomial-based methods [9, 11, 36] offer another rule-based strategy by representing circuits algebraically for rotation merging.

While these approaches are relatively efficient and scalable, their inherently greedy nature limits overall optimization quality.

8.1.2 Automated Discovery and Superoptimization

To move beyond expert-provided rules, several systems aim to *discover* circuit equivalences automatically. Superoptimization is a technique originally designed to search for an optimal sequence of instructions for classical programs [87]. Our approach to generating quantum transformations is inspired by prior work on X86 peephole optimizations [18, 60] and tensor algebra graph substitutions like TASO [69] and TENSAT [152].

In the quantum domain, Quanto [109] identifies transformations by computing concrete matrix representations of circuits, but it handles parameters only through instantiation at concrete values. In contrast, Quartz handles the significantly different semantics of quantum computing by using a combination of fingerprinting, SMT-based verification, and pruning techniques to scale rule generation in the presence of *symbolic* parameters.

8.1.3 Search-Based and Hybrid Strategies

Rule-based optimizers can be limited by a small set of local rules. Search-based methods address this by exploring larger neighborhoods and tolerating temporary cost-increasing transformations. Following Quartz, QUESO [144] also automatically generates equivalence rules and uses circuit cost to guide a beam search. While these methods achieve better optimization quality, they face massive scalability challenges due to exponential search space growth.

Several recent works attempt to bridge the gap between rule-based efficiency and search-based quality. Learning-based methods train models via reinforcement learning to choose optimization actions [53, 82]. While effective for specific families of parameterized circuits [92] or iterative gate pruning [129], they typically incur substantial training costs and often struggle to generalize to the deep circuits required in fault-tolerant regimes. GUOQ [146] introduces a fast-slow framework, though its fast phase still relies on search-based exploration (its slow phase utilizes resynthesis methods detailed below). In contrast, we designed QALM to avoid these overheads by strategically interleaving lightweight rule-based passes with targeted search exploration.

8.1.4 Compiler Verification

Complementary to discovery and search is the critical need for correctness. VOQC [61] provides formally verified circuit rewrites in Rocq (formerly Coq), and CertiQ [127] supports verifying Qiskit compiler passes. Building on these foundations, Tao et al. introduce Giallar [136], which provides push-button verification for automated compiler passes. Peng et al. demonstrate a formally certified end-to-end compilation pipeline for Shor’s algorithm [107]. Recently, Abdulla et al. [2] addressed the verification of parameterized quantum circuits for global properties. Quartz tackles a distinct challenge by proving the equivalence of local subcircuits with continuous parameters using automated theorem provers, focusing on the automated *generation* of these verified rules rather than checking an existing compiler pass.

8.1.5 Resynthesis Methods

Resynthesis methods focus on decomposing unitaries into sequences of smaller unitaries using algebraic structures of matrices. Examples include the Cartan decomposition [138], the Cosine-Sine Decomposition (CSD), the Khaneja Glaser Decomposition (KGD) [74], and the Quantum Shannon Decomposition (QSD).

Some synthesis methods demonstrate optimality for arbitrary unitaries of small size (typically for fewer than five qubits), particularly in terms of gate counts like CNOT gates [116]. For example, tools like QSearch [79] utilize heuristic search algorithms combined with numerical optimization to synthesize depth-optimal circuits for these smaller unitaries. However, their efficiency degrades significantly when dealing with larger unitaries; furthermore, they require the time-consuming step of turning the circuit into a full unitary matrix.

To address this scaling limitation, several frameworks employ hierarchical or block-by-block synthesis approaches. The BQSKit framework [153], for instance, integrates algorithms like QFast [154], which instantiates circuit templates to hierarchically decompose large unitaries. Similarly, QGo [143] uses a hierarchical approach that partitions and resynthesizes circuits block-by-block.

However, due to the lack of optimization across these isolated blocks, the performance of such hierarchical methods—including QFast, BQSKit, and QGo—depends heavily on how the circuits are initially partitioned. Our OAC framework directly addresses this limitation; its cut-and-meld algorithm explicitly optimizes across partition boundaries to attain local optimality.

8.1.6 Scalability and Local Optimality

As circuits grow, applying any global optimization strategy becomes computationally intractable. Our OAC contribution is closely related to prior work on classical peephole scaling [39], but it differs radically in scale. While classical peephole optimizers target a handful of instructions, OAC applies to large segments of quantum circuits containing thousands of gates.

The cut-and-meld strategy of OAC also shares a deep structural lineage with logic synthesis in classical Electronic Design Automation (EDA). In Boolean circuit optimization, state-of-the-art tools like Berkeley’s ABC [90] frequently employ Directed Acyclic Graph (DAG) rewriting and “windowing” techniques. These classical methods enumerate k -feasible cuts (small sub-networks), optimize them locally using truth tables or SAT solvers, and substitute them back into the main circuit.

However, applying this paradigm to quantum circuits introduces fundamental semantic challenges. A classical cut represents a simple Boolean function with independent inputs and outputs. In contrast, a quantum subcircuit represents a multi-qubit unitary operation where local transformations can inadvertently alter global phase or disrupt entanglement boundaries. OAC bridges this gap by introducing a quantum-native melding protocol that safely recombines these unitary partitions, achieving segment-level optimality without violating the strict algebraic constraints of quantum mechanics.

Furthermore, OAC is designed to use external optimizers as *oracles*. For example, it can invoke VOQC for gate reduction or specialized tools like FeynOpt for Clifford+ T circuits. This cleanly separates the outer algorithmic structure, which provides the local optimality guarantees,

from the inner optimizers that capture domain-specific cost models.

8.1.7 Distinction from Quantum Circuit Cutting and Knitting

Because the terms “cutting” and “partitioning” appear frequently in this thesis, it is necessary to distinguish our techniques from the entirely separate subfield of quantum circuit cutting, sometimes referred to as circuit knitting.

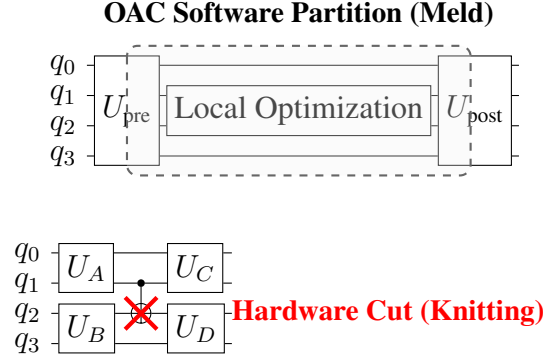


Figure 8.1: Conceptual distinction between the software partitioning used in OAC and the physical circuit cutting found in prior execution strategies. OAC uses software boundaries to optimize a single, fully entangled program before execution (top). In contrast, traditional circuit knitting must physically sever entangling gates, executing multiple smaller, independent subcircuits that require exponential classical overhead to reconstruct the final results (bottom).

In the broader quantum literature, circuit cutting refers to a hardware-oriented execution strategy. Techniques like those proposed by Peng et al. [106], Mitarai and Fujii [91], and the CutQC framework [134] involve physically decomposing a large quantum circuit into smaller subcircuits by severing entanglement wires or multi-qubit gates. These smaller subcircuits are then executed independently on near-term devices with limited qubit counts. The results are recombined via classical post-processing, which incurs an exponential overhead in the number of required measurement samples.

This thesis uses cutting and partitioning in fundamentally different contexts. Neither OAC nor Atlas alters the mathematical execution model of the circuit or relies on exponential classical sampling. In OAC, “cutting” is strictly a software compilation boundary used to isolate subcircuits for the application of local optimization algorithms. The fully optimized circuit is then melded back together and executed as a single, unified program on quantum hardware. Similarly, in Atlas, “partitioning” refers to the distribution of a classical state vector across multiple GPUs to optimize memory bandwidth and parallel classical simulation, completely unrelated to the physical severing of quantum entanglement.

8.1.8 Optimization Objectives and Hardware Constraints

Quantum circuit optimization spans diverse objectives. In the NISQ era, a primary concern is improving end-to-end fidelity by accounting for noise, motivating hardware-specific compilation

for topology and pulse characteristics [93, 126]. Emerging architectures also require specialized routing and scheduling; for instance, frameworks like Atomique [140] have been developed specifically to manage the unique atom movement and gate scheduling constraints of reconfigurable neutral atom arrays. In fault-tolerant settings, resource costs are often dominated by non-Clifford operations, making T -count a common objective using techniques like phase folding [9, 10, 11] or ZX-calculus rewriting [76].

8.1.9 Dynamic Circuit Compilation

As quantum hardware and advanced algorithms increasingly incorporate mid-circuit measurements and real-time classical control flow, optimizing *dynamic circuits* has emerged as a critical challenge. Recent compilation frameworks have primarily focused on qubit reuse, employing heuristics or Integer Linear Programming (ILP) to reduce physical qubit requirements by resetting and reusing qubits post-measurement [30, 63]. Concurrently, equivalence checking for these non-unitary programs has advanced. Recent works, such as those by Burgholzer et al. [34], utilize Tensor Decision Diagrams or the principle of deferred measurement to verify the equivalence of dynamic circuits.

8.2 Quantum Circuit Simulation

This thesis also considers the performance of classical simulation of quantum circuits, which is essential for validating optimized programs.

8.2.1 State-Vector Simulation and Partitioning

Distributed state-vector simulators exploit data parallelism by partitioning the state vector across compute nodes [35, 104, 132, 142, 157, 158, 162, 163]. However, applying multi-qubit gates that straddle partitions incurs substantial communication overhead. Prior work proposes heuristic circuit partitioning and scheduling strategies to coalesce operations on smaller subspaces [50, 104, 133, 157]. Atlas advances this by formulating partitioning as an integer linear programming (ILP) problem, enabling mathematically optimal solutions under chosen cost models.

8.2.2 Kernelization and Gate Fusion

Another common performance lever is fusing multiple gates acting on nearby qubits into larger kernels for cache-friendly execution [5, 141, 157]. Tools like Qulacs provide heuristic light and heavy fusion modes [133]. Atlas formalizes this kernelization as an explicit optimization problem, proposing a dynamic programming algorithm to systematically construct kernels under strict hardware resource constraints.

8.2.3 Alternative Simulation Paradigms

Beyond general-purpose Schrödinger-style simulation, specialized simulators exploit specific algebraic or structural properties of restricted quantum circuit families [73, 158, 161]. Stabilizer simulators, founded on the Gottesman-Knill theorem [55], efficiently process Clifford circuits in polynomial time. However, their runtime scales exponentially with the injection of non-Clifford resources, often quantified by the “magic” or stabilizer rank of the circuit [3, 31].

Alternatively, tensor network simulators represent quantum states as interconnected tensors. These are highly efficient for circuits with low entanglement but degrade rapidly when simulating the deep, volume-law entanglement characteristic of complex algorithms [84, 85, 86].

Conversely, Feynman-style sum-over-paths simulators compute specific measurement amplitudes with minimal memory overhead, though at the cost of exponential time complexity [1]. For circuits producing highly concentrated output distributions, sparse state-vector simulators can mitigate the $O(2^n)$ memory bottleneck by tracking only non-zero amplitudes, but their efficiency inevitably collapses under heavy superposition.

Finally, while error-aware simulation engines have been developed to track physical noise processes alongside the quantum state [80, 137], they incur substantially higher computational overhead. In this thesis, we focus on general circuit simulation within an idealized (noise-free) model. This approach avoids the steep computational costs of density matrix or stochastic trajectory simulations, providing a foundational baseline for evaluating algorithmic behavior and compiler optimizations applicable to the fault-tolerant era.

Chapter 9

Conclusion and Future Directions

9.1 Summary of Contributions

Optimizing and simulating quantum circuits at scale present significant challenges in the quantum computing software stack. This dissertation contributes a suite of frameworks designed to address these challenges through systematic, mathematically grounded approaches. Specifically, we demonstrate how automated rule generation, hybrid search strategies, and hierarchical partitioning can be leveraged to improve compilation quality, reduce runtime overhead, and accelerate classical simulation.

We first address the manual, hardware-specific nature of prior circuit transformations by introducing Quartz. By establishing the theoretical foundation for Equivalence Circuit Class (ECC) sets and leveraging automated theorem proving, Quartz automates the generation and verification of symbolic circuit transformations for arbitrary gate sets. Furthermore, by employing a cost-based backtracking search, Quartz successfully escapes the local minima that trap traditional rule-based compilers, yielding highly optimized circuits with mathematically verified equivalence.

Building on Quartz, we further improve compilation efficiency by creating a strategy with exploration and exploitation. To bypass the exponential runtime costs associated with pure search, our implementation, called QALM, strategically combines the exhaustive exploration of search algorithms with the rapid execution of rule-based rewriting. By dynamically interleaving these two phases, this approach efficiently navigates the optimization landscape to escape local minima.

To scale these high-quality optimization capabilities to larger quantum programs and bypass the QMA-hard nature of global circuit optimization, we introduce a paradigm shift from global to segment-level local optimality. To realize this shift, we present the OAC algorithm, which uses a novel cut-and-meld technique to partition circuits, apply an existing oracle optimizer independently to each subcircuit, and seamlessly meld the results. This algorithm requires only a linear number of oracle calls, significantly improving optimization efficiency while rigorously maintaining quality guarantees.

Finally, recognizing that scalable classical simulation is an essential pillar of the quantum software stack, we present Atlas. To manage the large memory and communication demands of

exact state-vector simulation, Atlas optimizes the distribution of the state vector across multiple GPUs. By formulating circuit partitioning as an integer linear programming problem and using dynamic programming for kernel scheduling, Atlas effectively exploits data parallelism and minimizes communication, accelerating quantum circuit simulation significantly.

9.2 Future Directions

The frameworks presented in this dissertation open several promising avenues for future research in quantum compilation and simulation.

9.2.1 Bounded Approximate Optimization via Cut-and-Meld

The OAC algorithm presented in this dissertation guarantees segment-level optimality under the strict constraint of exact unitary equivalence. However, in both NISQ and fault-tolerant regimes, significant resource reductions can often be achieved by relaxing this constraint and allowing for *approximate* optimization (e.g., discarding negligibly small-angle rotations or approximating unitaries up to a small threshold ϵ).

At first glance, extending OAC to this domain seems straightforward: one might simply replace the exact oracle with an approximate optimization oracle. However, this is fundamentally non-trivial due to the mechanics of the cut-and-meld protocol. When OAC recombines partitioned subcircuits, it propagates the unitary transformations across the cut boundaries. If the local segments contain approximation errors, the melding process mathematically propagates and composes these errors. Because coherent errors in quantum circuits can accumulate constructively, naively melding approximate subcircuits can lead to an unbounded escalation of the global approximation error, rapidly destroying the overall fidelity of the algorithm.

A critical direction for future research is developing an error-aware extension of the OAC algorithm. This requires establishing a rigorous theoretical foundation for how approximation errors are composed during the melding phase. Future work must design a bounded-error melding protocol that can dynamically allocate an overall global error budget ϵ_{global} across local cuts, ensuring that the oracle optimizations remain effective without violating the strict global fidelity requirements of the final reconstructed circuit.

9.2.2 Adaptive Simulator Selection

Atlas significantly advances the performance of Schrödinger-style simulation, but its fundamental $O(2^n)$ memory requirement remains a strict theoretical limit. Different simulation paradigms offer distinct and complementary trade-offs: Feynman-style path integrals require polynomial memory but exponentially more time, stabilizer simulations are highly efficient for Clifford circuits but scale exponentially with the non-Clifford “magic” of the circuit, and tensor networks excel at separable circuits but struggle with deep volume-law entanglement.

Because no single paradigm can efficiently simulate all quantum programs, a highly compelling future direction is the development of an intelligent, circuit-aware selection framework. Rather than forcing all circuits through a single backend, future compilation toolchains could

dynamically analyze the structural properties of an incoming circuit—such as its degree of superposition, entanglement volume, or non-Clifford gate density. Based on this static analysis, the framework could route the entire program to the most efficient simulation backend. For instance, it could automatically dispatch highly entangled, dense circuits to a distributed state-vector engine like Atlas, while offloading weakly entangled circuits to a tensor network solver, optimizing overall resource utilization across classical computing clusters.

Bibliography

- [1] Scott Aaronson and Lijie Chen. Complexity-theoretic foundations of quantum supremacy experiments. In *Proceedings of the 32nd Computational Complexity Conference, CCC '17*, Dagstuhl, DEU, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 9783959770408. 8.2.3
- [2] Parosh Aziz Abdulla, Yu-Fang Chen, Michal Hečko, Lukáš Holík, Ondřej Lengál, Jyun-Ao Lin, and Ramanathan S Thinniyam. Parameterized verification of quantum circuits. *Proceedings of the ACM on Programming Languages*, 10(POPL):2021–2050, 2026. 8.1.4
- [3] Dorit Aharonov, Zeph Landau, and Johann Makowsky. The quantum fft can be classically simulated. *arXiv preprint quant-ph/0611156*, 2006. 8.2.3
- [4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN 0321486811. 1.1
- [5] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, F Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, et al. Qiskit: An open-source framework for quantum computing. *Accessed on: Mar, 16, 2019*. 1.1, 1.2, 7, 7.6.2, 8.2.2
- [6] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O’Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyantov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Tylour, Kenso Trabing, Matthew Treinish, Wes

- Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. Qiskit: An Open-source Framework for Quantum Computing. Zenodo, January 2019. URL <https://doi.org/10.5281/zenodo.2562111>. 4, 4.4.2, 4.4.2, 8.1.1
- [7] Yuri Alexeev, Dave Bacon, Kenneth R Brown, Robert Calderbank, Lincoln D Carr, Frederic T Chong, Brian DeMarco, Dirk Englund, Edward Farhi, Bill Fefferman, et al. Quantum computer systems for scientific discovery. *PRX quantum*, 2(1):017001, 2021. 1
 - [8] Andrei Amariutei and Simona Caraiman. Parallel quantum computer simulation on the gpu. In *15th International Conference on System Theory, Control and Computing*, pages 1–6. IEEE, 2011. 1.2, 7
 - [9] Matthew Amy. *Formal methods in quantum circuit design*. PhD thesis, University of Waterloo, 2019. 6.5, 6.6.1, 8.1.1, 8.1.8
 - [10] Matthew Amy and Joseph Lunderville. Linear and non-linear relational analyses for quantum program optimization. *Proceedings of the ACM on Programming Languages*, 9(POPL):1072–1103, 2025. 8.1.8
 - [11] Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-time t-depth optimization of clifford+t circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, 2014. doi: 10.1109/TCAD.2014.2341953. 4.4.1, 5.4.1, 6.5.1, 8.1.1, 8.1.8
 - [12] Jatin Arora, Mingkuan Xu, Dantong Li, and Umut A Acar. Quantum circuit local optimizer. <https://github.com/UmutAcarLab/oac>, 2025. 6.4
 - [13] Jatin Arora, Mingkuan Xu, Sam Westrick, Pengyu Liu, Dantong Li, Yongshan Ding, and Umut A Acar. Local optimization of quantum circuits. In *2025 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 572–583. IEEE, 2025. 1.1
 - [14] Alán Aspuru-Guzik, Anthony D Dutoi, Peter J Love, and Martin Head-Gordon. Simulated quantum computation of molecular energies. *Science*, 309(5741):1704–1707, 2005. 1
 - [15] Anderson Avila, Adriano Maron, Renata Reiser, Mauricio Pilla, and Adenauer Yamin. Gpu-aware distributed quantum simulation. In *Proceedings of the 29th Annual ACM symposium on applied computing*, pages 860–865, 2014. 1.2, 7
 - [16] Anderson Avila, Renata HS Reiser, Mauricio L Pilla, and Adenauer C Yamin. Optimizing d-gm quantum computing by exploring parallel and distributed quantum simulations under gpus architecture. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 5146–5153. IEEE, 2016. 1.2, 7
 - [17] Ryan Babbush, Jarrod McClean, Dave Wecker, Alán Aspuru-Guzik, and Nathan Wiebe. Chemical basis of trotter-suzuki errors in quantum chemistry simulation. *Physical Review A*, 91(2):022311, 2015. 1
 - [18] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming*

Languages and Operating Systems, ASPLOS XII, page 394–403, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934510. doi: 10.1145/1168857.1168906. URL <https://doi.org/10.1145/1168857.1168906>. 8.1.2

- [19] Adriano Barenco, Charles H Bennett, Richard Cleve, David P DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Physical review A*, 52(5):3457, 1995. 2.1.3
- [20] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi: 10.1007/978-3-642-22110-1_14. URL https://doi.org/10.1007/978-3-642-22110-1_14. 3.4
- [21] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012. 7.5.5
- [22] Harun Bayraktar, Ali Charara, David Clark, Saul Cohen, Timothy Costa, Yao-Lung L. Fang, Yang Gao, Jack Guan, John Gunnels, Azzam Haidar, Andreas Hehn, Markus Hohnerbach, Matthew Jones, Tom Lubowe, Dmitry Lyakh, Shinya Morino, Paul Springer, Sam Stanwyck, Igor Terentyev, Satya Varadhan, Jonathan Wong, and Takuma Yamaguchi. cuQuantum SDK: A high-performance library for accelerating quantum science, 2023. URL <http://arxiv.org/abs/2308.01999>. 1, 7.6.2
- [23] Paul Benioff. The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. *Journal of Statistical Physics*, 22:563–591, 05 1980. doi: 10.1007/BF01011339. 1
- [24] Charles H Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. *Theoretical computer science*, 560:7–11, 2014. 1
- [25] Daniel Berend and Tamir Tassa. Improved bounds on bell numbers and on moments of sums of random variables. *Probability and Mathematical Statistics*, 30(2): 185–205, 2010. URL <https://www.math.uni.wroc.pl/~pms/files/30.2/Article/30.2.1.pdf>. 7.5.1
- [26] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017. 1
- [27] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300, 2020. 1
- [28] Dolev Bluvstein, Simon J Evered, Alexandra A Geim, Sophie H Li, Hengyun Zhou, Tom Manovitz, Sepehr Ebadi, Madelyn Cain, Marcin Kalinowski, Dominik Hangleiter, et al. Logical quantum processor based on reconfigurable atom arrays. *Nature*, 626(7997):58–65, 2024. 1

- [29] Max Born. Quantenmechanik der stoßvorgänge. *Zeitschrift für physik*, 38(11):803–827, 1926. 2.1.1
- [30] Sebastian Brandhofer, Ilia Polian, and Kevin Krsulich. Optimal qubit reuse for near-term quantum computers. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 859–869. IEEE, 2023. 8.1.9
- [31] Sergey Bravyi and David Gosset. Improved classical simulation of quantum circuits dominated by clifford gates. *Physical review letters*, 116(25):250501, 2016. 8.2.3
- [32] Sergey Bravyi, Oliver Dial, Jay M Gambetta, Darío Gil, and Zaira Nazario. The future of quantum computing with superconducting qubits. *Journal of Applied Physics*, 132(16), 2022. 6.3
- [33] Colin D Bruzewicz, John Chiaverini, Robert McConnell, and Jeremy M Sage. Trapped-ion quantum computing: Progress and challenges. *Applied physics reviews*, 6(2), 2019. 1
- [34] Lukas Burgholzer and Robert Wille. Handling non-unitaries in quantum circuit equivalence checking. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 529–534, 2022. 8.1.9
- [35] Zhao-Yun Chen, Qi Zhou, Cheng Xue, Xia Yang, Guang-Can Guo, and Guo-Ping Guo. 64-qubit quantum circuit simulation. *Science Bulletin*, 63(15):964–971, 2018. 8.2.1
- [36] Zihan Chen, Henry Chen, Yuwei Jin, Minghao Guo, Enhyeok Jang, Jiakang Li, Caitlin Chan, Won Woo Ro, and Eddy Z Zhang. Phasepoly: An optimization framework for phase polynomials in quantum circuits. *arXiv preprint arXiv:2506.20624*, 2025. 5.3, 5.3.1, 8.1.1
- [37] Andrew M Childs, Richard Cleve, Enrico Deotto, Edward Farhi, Sam Gutmann, and Daniel A Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 59–68, 2003. 6.5.1
- [38] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Satisfiability modulo transcendental functions via incremental linearization. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 95–113. Springer, 2017. doi: 10.1007/978-3-319-63046-5_7. URL https://doi.org/10.1007/978-3-319-63046-5_7. 3.4
- [39] Keith D Cooper and Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2022. 8.1.6
- [40] Antonio D. Corcoles, Abhinav Kandala, Ali Javadi-Abhari, Douglas T. McClure, Andrew W. Cross, Kristan Temme, Paul D. Nation, Matthias Steffen, and Jay M. Gambetta. Challenges and opportunities of near-term quantum computing systems. *Proceedings of the IEEE*, 108(8):1338–1352, aug 2020. doi: 10.1109/jproc.2019.2954005. 7
- [41] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta,

- and Blake R. Johnson. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):12:1–12:50, 2022. doi: 10.1145/3505636. URL <https://dl.acm.org/doi/10.1145/3505636>. 2.2.1
- [42] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017. URL <http://arxiv.org/abs/1707.03429>. 2.2.1
- [43] Frederica Darema, David A George, V Alan Norton, and Gregory F Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1): 11–24, 1988. 7.4
- [44] Niel de Beaudrap, Aleks Kissinger, and John van de Wetering. Circuit extraction for zx-diagrams can be# p-hard. *arXiv preprint arXiv:2202.09194*, 2022. 8.1.1
- [45] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24. 4.3
- [46] Paul Adrien Maurice Dirac. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 35(3):416–418, 1939. 2.1.1
- [47] Eugene F Dumitrescu, Alex J McCaskey, Gaute Hagen, Gustav R Jansen, Titus D Morris, T Papenbrock, Raphael C Pooser, David Jarvis Dean, and Pavel Lougovski. Cloud quantum computing of an atomic nucleus. *Physical review letters*, 120(21):210501, 2018. 1.1, 2.1.3, 4.1, 4.3.2
- [48] Iain Dunning, Stuart Mitchell, and Michael O’Sullivan. PuLP: A linear programming toolkit for python, 2011. URL <https://optimization-online.org/?p=11731>. 7.6.1
- [49] Sepehr Ebadi, Tout T Wang, Harry Levine, Alexander Keesling, Giulia Semeghini, Ahmed Omran, Dolev Bluvstein, Rhine Samajdar, Hannes Pichler, Wen Wei Ho, et al. Quantum phases of matter on a 256-atom programmable quantum simulator. *Nature*, 595(7866): 227–232, 2021. 1
- [50] Stavros Efthymiou, Sergi Ramos-Calderer, Carlos Bravo-Prieto, Adrián Pérez-Salinas, Diego García-Martín, Artur Garcia-Saez, José Ignacio Latorre, and Stefano Crazza. Qibo: a framework for quantum simulation with hardware acceleration. *Quantum Science & Technology*, 7(1):015018, 2022. 8.2.1
- [51] Richard P Feynman. Simulating physics with computers. In *Feynman and computation*, pages 133–153. cRc Press, 2018. 1
- [52] Richard P Feynman, Robert B Leighton, and Matthew Sands. *The Feynman Lectures on Physics, Vol. III: Quantum Mechanics*. Addison-Wesley, 1965. Chapter 1: Quantum Behavior. 2.1.1
- [53] Thomas Fösel, Murphy Yuezhen Niu, Florian Marquardt, and Li Li. Quantum circuit

optimization with deep reinforcement learning. *arXiv preprint arXiv:2103.07585*, 2021. 8.1.3

- [54] Craig Gidney and Martin Ekerå. How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, 2021. 1
- [55] Daniel Gottesman. The heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006*, 1998. 8.2.3
- [56] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 333–342, 2013. 1
- [57] Lov K Grover. A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043*, 1996. 6.5.1
- [58] Eladio Gutiérrez, Sergio Romero, María A Trenas, and Emilio L Zapata. Quantum computer simulation using the cuda programming model. *Computer Physics Communications*, 181(2):283–300, 2010. 1.2, 7
- [59] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical review letters*, 103(15):150502, 2009. 6.5.1
- [60] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: Automatically learning the x86-64 instruction set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16*, page 237–250, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342612. doi: 10.1145/2908080.2908121. URL <https://doi.org/10.1145/2908080.2908121>. 8.1.2
- [61] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages*, 5 (POPL):1–29, 2021. 1, 1.1, 4, 4.1, 4.3.2, 4.4.1, 4.4.2, 4.4.2, 5.2.1, 5.3, 5.3.1, 5.4.1, 5.4.1, 6.1, 6.5, 8.1.1, 8.1.4
- [62] Torsten Hoefler, Thomas Häner, and Matthias Troyer. Disentangling hype from practicality: On realistically achieving quantum advantage. *Communications of the ACM*, 66(5): 82–87, 2023. 1
- [63] Fei Hua, Yuwei Jin, Yanhao Chen, Suhas Vittal, Kevin Krsulich, Lev S Bishop, John Lapeyre, Ali Javadi-Abhari, and Eddy Z Zhang. Caqr: A compiler-assisted approach for qubit reuse through dynamic circuit. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 59–71, 2023. 8.1.9
- [64] Q. Huangfu and J. A. J. Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018. ISSN 1867-2957. doi: 10.1007/s12532-017-0130-5. URL <https://doi.org/10.1007/s12532-017-0130-5>. 7.6.1
- [65] Thomas Häner and Damian S. Steiger. 0.5 petabyte simulation of a 45-qubit quantum

circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 1–10, New York, NY, USA, November 2017. Association for Computing Machinery. ISBN 978-1-4503-5114-0. doi: 10.1145/3126908.3126947. URL <https://doi.org/10.1145/3126908.3126947>. 1.2, 7, 3, 7.3

- [66] IBM Quantum. Ibm delivers new quantum processors, software, and algorithm breakthroughs, 2025. URL <https://newsroom.ibm.com/2025-11-12-ibm-delivers-new-quantum-processors,-software,-and-algorithm-breakthroughs-on-path-to-advantage-and-fault-tolerance>. 1
- [67] Dominik Janzing, Pawel Wocjan, and Thomas Beth. Identity check is qma-complete, 2003. 1.1
- [68] 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. Quantum computing with Qiskit, 2024. 5.2.1
- [69] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019. 4, 4.1, 8.1.2
- [70] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019. 4.1
- [71] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019. 7.5.5
- [72] Tyson Jones, Anna Brown, Ian Bush, and Simon C Benjamin. Quest and high performance simulation of quantum computers. *Scientific reports*, 9(1):1–11, 2019. 1.2, 7
- [73] Richard Jozsa and Noah Linden. On the role of entanglement in quantum-computational speed-up. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 459(2036):2011–2032, aug 2003. 8.2.3
- [74] Navin Khaneja and Steffen J Glaser. Cartan decomposition of $su(2n)$ and control of spin systems. *Chemical Physics*, 267(1-3):11–23, 2001. 8.1.5
- [75] Aleks Kissinger and John van de Wetering. Reducing t-count with the zx-calculus. *arXiv preprint arXiv:1903.10477*, 2019. 3
- [76] Aleks Kissinger and John van de Wetering. PyZX: Large Scale Automated Diagrammatic Reasoning. In Bob Coecke and Matthew Leifer, editors, *Proceedings 16th International Conference on Quantum Physics and Logic*, Chapman University, Orange, CA, USA., 10-14 June 2019, volume 318 of *Electronic Proceedings in Theoretical Computer Science*, pages 229–241. Open Publishing Association, 2020. doi: 10.4204/EPTCS.318.14. 8.1.1, 8.1.8

- [77] Morten Kjaergaard, Mollie E Schwartz, Jochen Braumüller, Philip Krantz, Joel I-J Wang, Simon Gustavsson, and William D Oliver. Superconducting qubits: Current state of play. *Annual Review of Condensed Matter Physics*, 11(1):369–395, 2020. 1, 1
- [78] Jason R. Koenig, Oded Padon, and Alex Aiken. Adaptive restarts for stochastic synthesis. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 696–709. ACM, 2021. doi: 10.1145/3453483.3454071. URL <https://doi.org/10.1145/3453483.3454071>. 4.4.2
- [79] Costin Lancu, Marc Davis, Ethan Smith, and USDOE. Quantum search compiler (qsearch) v2.0, version v2.0, 10 2020. URL <https://www.osti.gov/biblio/1769276>. 8.1.5
- [80] Ang Li, Omer Subasi, Xiu Yang, and Sriram Krishnamoorthy. Density matrix quantum circuit simulation via the bsp machine on modern gpu clusters. In *Sc20: international conference for high performance computing, networking, storage and analysis*, pages 1–15. IEEE, 2020. 8.2.3
- [81] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. Qasmbench: A low-level qasm benchmark suite for nisq evaluation and simulation. *arXiv preprint arXiv:2005.13018*, 2021. 7.6.1, 7.6.5
- [82] Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. Quarl: A learning-based quantum circuit optimizer. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):555–582, 2024. 1.1, 5.1, 5.4.1, 6.1, 8.1.3
- [83] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. Quantum algorithms for supervised and unsupervised machine learning. *arXiv preprint arXiv:1307.0411*, 2013. 1
- [84] Danylo Lykov, Roman Schutski, Alexey Galda, Valeri Vinokur, and Yuri Alexeev. Tensor network quantum simulator with step-dependent parallelization. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 582–593, 2022. doi: 10.1109/QCE53715.2022.00081. URL <https://ieeexplore.ieee.org/document/9951269>. 8.2.3
- [85] Danylo Lykov, Ruslan Shaydulin, Yue Sun, Yuri Alexeev, and Marco Pistoia. Fast simulation of high-depth QAOA circuits. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23*, pages 1443–1451. Association for Computing Machinery, 2023. ISBN 9798400707858. doi: 10.1145/3624062.3624216. URL <https://doi.org/10.1145/3624062.3624216>. 8.2.3
- [86] Igor L Markov and Yaoyun Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008. 8.2.3
- [87] Henry Massalin. Superoptimizer: A look at the smallest program. *SIGARCH Comput. Archit. News*, 15(5):122–126, October 1987. ISSN 0163-5964. doi: 10.1145/36177.36194. URL <https://doi.org/10.1145/36177.36194>. 8.1.2
- [88] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kir-

- pichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL <https://doi.org/10.7717/peerj-cs.103>. 4.3.4
- [89] Hector Miller-Bakewell. Finite verification of infinite families of diagram equations. *arXiv preprint arXiv:1904.00706*, 2019. 4.3.4
- [90] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In *Proceedings of the 43rd annual Design Automation Conference*, pages 532–535, 2006. 8.1.6
- [91] Kosuke Mitarai and Keisuke Fujii. Constructing a virtual two-qubit gate by sampling single-qubit operations. *New Journal of Physics*, 23(2):023021, 2021. 8.1.7
- [92] Kosuke Mitarai, Makoto Negoro, Masahiro Kitagawa, and Keisuke Fujii. Quantum circuit learning. *Physical Review A*, 98(3):032309, 2018. 8.1.3
- [93] Abtin Molavi, Amanda Xu, Martin Diges, Lauren Pick, Swamit Tannu, and Aws Albarghouti. Qubit mapping and routing via maxsat. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1078–1091. IEEE, 2022. 8.1.8
- [94] Christopher Monroe, Wes C Campbell, L-M Duan, Z-X Gong, Alexey V Gorshkov, Paul W Hess, Rajibul Islam, Kihwan Kim, Norbert M Linke, Guido Pagano, et al. Programmable quantum simulations of spin systems with trapped ions. *Reviews of Modern Physics*, 93(2):025001, 2021. 1
- [95] Steven A Moses, Charles H Baldwin, Michael S Allman, R Ancona, L Ascarrunz, C Barnes, J Bartolotta, B Bjork, P Blanchard, M Bohn, et al. A race-track trapped-ion quantum processor. *Physical Review X*, 13(4):041052, 2023. 1
- [96] Prakash Murali, Jonathan M Baker, Ali Javadi-Abhari, Frederic T Chong, and Margaret Martonosi. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1015–1029. ACM, 2019. 5.4.1
- [97] Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1):23, 2018. 1, 4.2, 4.1, 4.3.2, 4.4.1, 4.4.2, 5.1, 5.3, 5.4.1, 6.1, 6.5, 6.5.1, 8.1.1
- [98] National Energy Research Scientific Computing Center. The perlmutter supercomputer. <https://docs.nersc.gov/systems/perlmutter/>, 2023. 7.6.1
- [99] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 10th anniversary edition, 2010. 1, 2.1.1, 2.1.1, 2.1.3, 2.2, 2.2.3

- [100] Matthew A Norcia, Hyemin Kim, William B Cairncross, M Stone, A Ryou, M Jaffe, MO Brown, K Barnes, P Battaglino, TC Bohdanowicz, et al. Iterative assembly of 171 yb atom arrays with cavity-enhanced optical lattices. *PRX Quantum*, 5(3):030316, 2024. 1
- [101] NVIDIA Corporation. Nvidia nccl. <https://developer.nvidia.com/nccl>, 2021. 7.5.5
- [102] Sivaprasad Omanakuttan, Zichang He, Zhiwei Zhang, Tianyi Hao, Arman Babakhani, Sami Boulebnane, Shouvanik Chakrabarti, Dylan Herman, Joseph Sullivan, Michael A Perlin, et al. Threshold for fault-tolerant quantum advantage with the quantum approximate optimization algorithm. *arXiv preprint arXiv:2504.01897*, 2025. doi: 10.48550/arXiv.2504.01897. URL <http://arxiv.org/abs/2504.01897>. 1
- [103] Román Orús, Samuel Mugel, and Enrique Lizaso. Quantum computing for finance: Overview and prospects. *Reviews in Physics*, 4:100028, 2019. 1
- [104] Daeyoung Park, Heehoon Kim, Jinpyo Kim, Taehyun Kim, and Jaejin Lee. SnuQS: scaling quantum circuit simulation using storage devices. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, pages 1–13, New York, NY, USA, June 2022. Association for Computing Machinery. ISBN 978-1-4503-9281-5. doi: 10.1145/3524059.3532375. URL <https://dl.acm.org/doi/10.1145/3524059.3532375>. 1.2, 7, 7.6.4, 8.2.1
- [105] Jennifer Paykin, Robert Rand, and Steve Zdancewic. Qwire: a core language for quantum circuits. *ACM SIGPLAN Notices*, 52(1):846–858, 2017. 1
- [106] Tianyi Peng, Aram W Harrow, Maris Ozols, and Xiaodi Wu. Simulating large quantum circuits on a small quantum computer. *Physical review letters*, 125(15):150504, 2020. 6.3, 8.1.7
- [107] Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. A formally certified end-to-end implementation of shor’s factorization algorithm. *Proceedings of the National Academy of Sciences*, 120(21):e2218775120, 2023. 8.1.4
- [108] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O’Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5:4213, 2014. 6.5.1
- [109] Jessica Pointing, Oded Padon, Zhihao Jia, Henry Ma, Auguste Hirth, Jens Palsberg, and Alex Aiken. Quanto: Optimizing quantum circuits with automatic generation of circuit identities. *Quantum Science and Technology*, 9(4):045009, 2024. 1.1, 8.1.2
- [110] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018. 1.1, 7
- [111] Qiskit Contributors. Qiskit consolidateblocks. <https://qiskit.org/documentation/stubs/qiskit.transpiler.passes.ConsolidateBlocks.html>, 2021. 4.4.2
- [112] Qiskit Contributors. Qiskit optimize1qgates. <https://qiskit.org/documentation/stubs/qiskit.transpiler.passes>.

Optimize1qGates.html, 2021. 4.4.2

- [113] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023. 1, 6.6.1
- [114] Quartz Contributors. The quartz quantum circuit superoptimizer. <https://github.com/quantum-compiler/quartz>, 2022. 4.3
- [115] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. MQT Bench: Benchmarking software and design automation tools for quantum computing, 2022. MQT Bench is available at <https://www.cda.cit.tum.de/mqtbench/>. 7.6.1
- [116] Péter Rakyta and Zoltán Zimborás. Approaching the theoretical limit in quantum gate decomposition. *Quantum*, 6:710, 2022. 8.1.5
- [117] Matthew Reagor, Christopher B. Osborn, Nikolas Tezak, Alexa Staley, Guenevere Prawiroatmodjo, Michael Scheer, Nasser Alidoust, Eyob A. Sete, Nicolas Didier, Marcus P. da Silva, and et al. Demonstration of universal parametric entangling gates on a multi-qubit lattice. *Science Advances*, 4(2), Feb 2018. ISSN 2375-2548. doi: 10.1126/sciadv.aao3603. URL <http://dx.doi.org/10.1126/sciadv.aao3603>. 1.1, 4.1, 4.3.2
- [118] Rigetti Computing. Rigetti gates. <https://pyquil-docs.rigetti.com/en/v2.7.0/apidocs/gates.html>, 2021. 4.3.2
- [119] Neil J. Ross and Peter Selinger. Optimal ancilla-free clifford+t approximation of z-rotations, 2016. 6.6.1
- [120] Neil J Ross and Scott Wesley. Cutoff theorems for the equivalence of parameterized quantum circuits. In *50th International Symposium on Mathematical Foundations of Computer Science (MFCS 2025)*, pages 85–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025. 4.3.4
- [121] Mark Saffman, Thad G Walker, and Klaus Mølmer. Quantum information with rydberg atoms. *Reviews of modern physics*, 82(3):2313–2363, 2010. 1
- [122] Pascal Scholl, Michael Schuler, Hannah J Williams, Alexander A Eberharter, Daniel Barredo, Kai-Niklas Schymik, Vincent Lienhard, Louis-Paul Henry, Thomas C Lang, Thierry Lahaye, et al. Quantum simulation of 2d antiferromagnets with hundreds of rydberg atoms. *Nature*, 595(7866):233–238, 2021. 1
- [123] Maria Schuld and Nathan Killoran. Quantum machine learning in feature hilbert spaces. *Physical review letters*, 122(4):040504, 2019. 1
- [124] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. An introduction to quantum machine learning. *Contemporary Physics*, 56(2):172–185, 2015. 1
- [125] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004. 1
- [126] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I Schuster, Henry Hoffmann, and Frederic T Chong. Optimized compilation of aggregated instructions for realistic quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1031–

1044. ACM, 2019. 8.1.8

- [127] Yunong Shi, Runzhou Tao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. Certiq: A mostly-automated verification of a realistic quantum compiler, 2020. 8.1.4
- [128] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994. 1, 6.5.1
- [129] Sukin Sim, Jonathan Romero, Jérôme F Gonthier, and Alexander A Kunitsa. Adaptive pruning-based optimization of parameterized quantum circuits. *Quantum Science and Technology*, 6(2):025019, 2021. 8.1.3
- [130] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket : a retargetable compiler for nisq devices. *Quantum Science and Technology*, 6(1):014003, 2020. 1, 1.1, 4, 4.4.2, 4.4.2, 4.4.2, 5.2.1, 8.1.1
- [131] Mark Skilbeck, Eric Peterson, appleby, Erik Davis, Peter Karalekas, Juan M. Bello-Rivas, Daniel Kochmanski, Zach Beane, Robert Smith, Andrew Shi, Cole Scott, Adam Paszke, Eric Hulburd, Matthew Young, Aaron S. Jackson, BHAVISHYA, M. Sohaib Alam, Wilfredo Velázquez-Rodríguez, c. b. osborn, fengdlm, and jmackeyrigetti. rigetti/quilc: v1.21.0. Zenodo, July 2020. URL <https://doi.org/10.5281/zenodo.3967926>. 4.4.2, 8.1.1
- [132] Mikhail Smelyanskiy, Nicolas PD Sawaya, and Alán Aspuru-Guzik. qhipster: The quantum high performance software testing environment. *arXiv preprint arXiv:1601.07195*, 2016. 1.2, 7, 8.2.1
- [133] Yasunari Suzuki, Yoshiaki Kawase, Yuya Masumura, Yuria Hiraga, Masahiro Nakadai, Jiabao Chen, Ken M Nakanishi, Kosuke Mitarai, Ryosuke Imai, Shiro Tamiya, et al. Qulacs: a fast and versatile quantum circuit simulator for research purpose. *Quantum*, 5:559, 2021. 1.2, 7, 8.2.1, 8.2.2
- [134] Wei Tang, Teague Tomesh, Martin Suchara, Jeffrey Larson, and Margaret Martonosi. Cutqc: using small quantum computers for large quantum circuit evaluations. In *Proceedings of the 26th ACM International conference on architectural support for programming languages and operating systems*, pages 473–486, 2021. 6.3, 8.1.7
- [135] Swamit S Tannu and Moinuddin K Qureshi. Not all qubits are created equal: a case for variability-aware policies for nisq-era quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 987–999. ACM, 2019. 5.4.1
- [136] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W Cross, Frederic T Chong, and Ronghui Gu. Giallar: Push-button verification for the qiskit quantum compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 641–656, 2022. 8.1.4
- [137] Doan Binh Trieu. *Large-scale simulations of error prone quantum computation devices*, volume 2. Forschungszentrum Jülich, 2010. 8.2.3

- [138] Robert R Tucci. An introduction to cartan’s kak decomposition for qc programmers. *arXiv preprint quant-ph/0507171*, 2005. 8.1.5
- [139] Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. Qunity: A unified language for quantum and classical computing. *Proceedings of the ACM on Programming Languages*, 7(POPL):921–951, 2023. 1
- [140] Hanrui Wang, Pengyu Liu, Daniel Bochen Tan, Yilian Liu, Jiaqi Gu, David Z Pan, Jason Cong, Umut A Acar, and Song Han. Atomique: A quantum compiler for reconfigurable neutral atom arrays. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 293–309. IEEE, 2024. 8.1.8
- [141] Sam Westrick, Pengyu Liu, Byeongjee Kang, Colin McDonald, Mike Rainey, Mingkuan Xu, Jatin Arora, Yongshan Ding, and Umut A. Acar. Grafeyn: Efficient parallel sparse simulation of quantum circuits. In *Proceedings of the IEEE International Conference on Quantum Computing and Engineering*, 2024. 8.2.2
- [142] Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T Chong. Full-state quantum circuit simulation by using data compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–24, 2019. 8.2.1
- [143] Xin-Chuan Wu, Marc Grau Davis, Frederic T Chong, and Costin Iancu. Qgo: Scalable quantum circuit optimization using automated synthesis. *arXiv preprint arXiv:2012.09835*, 2020. 8.1.5
- [144] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. Synthesizing quantum-circuit optimizers. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. doi: 10.1145/3591254. URL <https://doi.org/10.1145/3591254>. 1.1, 5.2.1, 5.4.1, 5.4.1, 6.1, 6.5, 8.1.3
- [145] Amanda Xu, Abtin Molavi, Swamit Tannu, and Aws Albarghouthi. Guoq benchmark suite. https://github.com/qqq-wisc/guoq/tree/8c4c3a5a6dfc9f7fc375ec16c2180139f0a8cb1a/benchmarks/nam_rz, 2024. 5.4.1
- [146] Amanda Xu, Abtin Molavi, Swamit Tannu, and Aws Albarghouthi. Optimizing quantum circuits, fast and slow. *arXiv preprint arXiv:2411.04104*, 2024. 1.1, 5.2.1, 5.4.1, 5.4.1, 8.1.3
- [147] Mingkuan Xu and Shiyi Cao. Atlas: High-performance gpu-based quantum circuit simulator. <https://github.com/quantum-compiler/atlas>, 2024. 7.5.5
- [148] Mingkuan Xu and Shiyi Cao. Artifact for sc24 paper: Atlas: Hierarchical partitioning for quantum circuit simulation on gpus. Zenodo, June 2024. URL <https://doi.org/10.5281/zenodo.12588145>. 7.5.5
- [149] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. Quartz: Superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 625–640, New

- York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523433. URL <https://doi.org/10.1145/3519939.3523433>. 1, 1.1, 3, 5.1, 5.2.1, 5.3, 5.3.2, 5.4.1, 5.4.1, 6.1, 6.5, 6.5.1
- [150] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. Artifact for PLDI 2022 paper: Quartz: Superoptimization of quantum circuits. Zenodo, April 2022. URL <https://doi.org/10.5281/zenodo.6508992>. 4.3
 - [151] Mingkuan Xu, Shiyi Cao, Xupeng Miao, Umut A. Acar, and Zhihao Jia. Atlas: Hierarchical partitioning for quantum circuit simulation on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2024. 1.2, 7
 - [152] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 255–268, 2021. 8.1.2
 - [153] Ed Younis, Costin C Iancu, Wim Lavrijsen, Marc Davis, and Ethan Smith. Berkeley quantum synthesis toolkit (bqskit) v1. Technical report, Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA (United States), 2021. 8.1.5
 - [154] Ed Younis, Koushik Sen, Katherine Yelick, and Costin Iancu. Qfast: Conflating search and numerical optimization for scalable quantum circuit synthesis, 2021. 8.1.5
 - [155] Charles Yuan and Michael Carbin. Tower: data structures in quantum superposition. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):259–288, 2022. 1
 - [156] Charles Yuan, Christopher McNally, and Michael Carbin. Twist: Sound reasoning for purity and entanglement in quantum programs. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–32, 2022. 1
 - [157] Chen Zhang, Zeyu Song, Haojie Wang, Kaiyuan Rong, and Jidong Zhai. Hyquas: Hybrid partitioner based quantum circuit simulation system on gpu. In *Proceedings of the ACM International Conference on Supercomputing*, ICS ’21, page 443–454, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383356. doi: 10.1145/3447818.3460357. URL <https://doi.org/10.1145/3447818.3460357>. 1.2, 7, 7.5.2, 7.6.2, 8.2.1, 8.2.2
 - [158] Chen Zhang, Haojie Wang, Zixuan Ma, Lei Xie, Zeyu Song, and Jidong Zhai. UniQ: A unified programming model for efficient quantum circuit simulation. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2022. doi: 10.1109/SC41404.2022.00054. ISSN: 2167-4337. 7.6.2, 8.2.1, 8.2.3
 - [159] Pei Zhang, Jiabin Yuan, and Xiangwen Lu. Quantum computer simulation on multi-gpu incorporating data locality. In *Algorithms and Architectures for Parallel Processing: 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015, Proceedings, Part I 15*, pages 241–256. Springer, 2015. 1.2, 7

- [160] Xin-Ding Zhang, Xiao-Ming Zhang, and Zheng-Yuan Xue. Quantum hyperparallel algorithm for matrix multiplication. *Scientific reports*, 6(1):24910, 2016. 1
- [161] Yilun Zhao, Yanan Guo, Yuan Yao, Amanda Dumi, Devin M Mulvey, Shiv Upadhyay, Youtao Zhang, Kenneth D Jordan, Jun Yang, and Xulong Tang. Q-gpu: A recipe of optimizations for quantum circuit simulation using gpus. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 726–740. IEEE, 2022. 8.2.3
- [162] Yilun Zhao, Yu Chen, He Li, Ying Wang, Kaiyan Chang, Bingmeng Wang, Bing Li, and Yinhe Han. Full state quantum circuit simulation beyond memory limit. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2023. 7.6.3, 8.2.1
- [163] Alwin Zulehner and Robert Wille. Advanced simulation of quantum computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):848–859, 2018. 8.2.1
- [164] Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the ibm qx architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(7):1226–1236, 2018. 5.4.1