



Project Title

NeuroTKET: An ML-Enhanced Quantum Circuit Compiler Using TKET

Summary

This project proposes a novel approach to quantum circuit compilation that integrates machine learning with Quantinuum's TKET compiler. The goal is to automatically optimize quantum circuits beyond the capabilities of standard rule-based compilers, without relying on deep reinforcement learning (RL). Instead of training an RL agent as in prior work [1](#) [2](#), we will develop a lightweight learning-based optimizer that can learn circuit simplification patterns and adapt to hardware constraints. Using Quantinuum's open-source **TKET** toolchain and APIs, the project will involve implementing an ML-guided optimization module that works alongside TKET's native passes. Over 8 weeks, we will iteratively build and test this system on Quantinuum's device emulator (and real hardware if available), measuring improvements in gate count and expected fidelity. The outcome will be a prototype "neuro-compiler" capable of reducing circuit depth or gate count with minimal overhead, plus a demonstration of its effectiveness on example circuits using open-source tools.

Novelty/Innovation

Unlike existing quantum compilers that rely on deep RL agents to find gate sequences [3](#) [2](#), our project explores a new machine learning paradigm for quantum compilation. **Firstly**, we avoid the heavy pre-training and complex action-space handling of RL-based compilers [4](#). Prior RL compilers (e.g. Quarl and others) had to address enormous action spaces and state representations with specialized neural architectures [5](#), and required long training phases to learn a policy [1](#). In contrast, our approach will use supervised or self-supervised learning to guide circuit optimizations, significantly simplifying the learning problem. **Secondly**, instead of learning from scratch how to build circuits, our method will leverage TKET's powerful existing optimizations as a baseline and then learn *residual* improvements. This means the ML model focuses on identifying non-obvious circuit transformations that TKET's built-in passes might miss – for example, merging rotations or reordering gates in ways that standard heuristics don't attempt. Notably, Quarl's RL agent surprisingly learned complex optimizations like rotation merging that usually require a dedicated pass [6](#); our project aims to achieve similar advanced optimizations via a different learning technique. **Thirdly**, the project incorporates **hardware-awareness** directly into the compiler. By using Quantinuum's API and device noise models, the ML module can learn to prefer transformations that not only shorten the circuit but also *improve fidelity on Quantinuum hardware*. Traditional compilers optimize mostly for gate count or depth, but our ML-enhanced compiler can be tuned to also account for hardware specifics (e.g. which gate types or qubit mappings yield higher fidelity on H1-1). This kind of adaptive, machine-learned optimization goes beyond prior RL compilers that mostly focused on gate count minimization [2](#). In summary, the project is innovative in combining TKET's deterministic compiling strength with a data-driven ML layer to achieve **faster, approximate optimizations** without the heavy infrastructure of deep RL. It charts a new path for quantum compiling by using modern ML (graph neural networks, pattern recognition, etc.) to automate circuit improvements – a strategy that has seen only limited exploration (e.g. one recent predictor targeted a few compiler passes [7](#), but no comprehensive

ML-driven optimizer exists). By the end, we expect to demonstrate a prototype that can automatically discover circuit reductions or reorders that a human or default compiler wouldn't easily find, all within a 2-month effort and using only open-source resources.

2-Month Timeline and Milestones

Milestone 1 (Week 1) – Background Research & Environment Setup: The first week is devoted to grounding the project in existing work and preparing the tooling. The student will survey prior research at the intersection of quantum compiling and ML to refine the idea (covering RL-based compilers like Moro *et al.* ¹, recent learning-based optimizers like Quarl ⁴, and any non-RL approaches). This literature review will clarify how our approach differs (e.g. no Q-learning loop, focusing on supervised pattern learning instead). On the practical side, the student will install and configure **pytket** and the **pytket-quantinuum** extension ⁸ ⁹. We will ensure access to Quantinuum's systems by creating an account and obtaining the API key (if needed); the first API call will prompt for login to store credentials ¹⁰. Using a simple test circuit, we will practice TKET compilation and execution: for example, create a 2-qubit circuit in pytket, then compile it for the Quantinuum H1-1E emulator. Code snippet from documentation for reference:

```
from pytket import Circuit
from pytket.extensions.quantinuum import QuantinuumBackend
circ = Circuit(2).H(0).CX(0,1).CZ(0,1)
backend = QuantinuumBackend('H1-1E')
backend.default_compilation_pass().apply(circ) # apply TKET optimizations 11
```

This will transpile the circuit to satisfy the H1-1 device constraints and apply level-2 optimizations by default ¹¹. We'll validate that the compiled circuit runs on the emulator (with `backend.process_circuits([circ], n_shots=100, noisy_simulation=False)` to get a noiseless result on the emulator). At this stage, the student also measures baseline metrics: e.g., use `circ.n_gates` or similar to record gate count after TKET's optimization, and `backend.cost(circ)` to estimate the Hardware Quantum Credits cost ¹². If possible, retrieve the unitary or output distribution and compare it to the ideal (especially if using an emulator with `noisy_simulation=False` to simulate ideally ¹³). These baseline experiments (perhaps on a known small circuit like a 3-qubit QFT) will familiarize the student with TKET's API and quantinuum's stack. By end of Week 1, we expect a working environment, some baseline data, and a clearer plan of attack grounded in literature insights.

Milestone 2 (Weeks 2-3) – Design of ML-Assisted Optimizer & Data Preparation: In the second and third weeks, the focus is on formulating the machine learning approach and gathering any training data needed. The core idea is to teach an ML model to recognize when a quantum circuit (or a portion of it) can be simplified or rewritten for improvement. The student will decide on a suitable ML model and representation: a promising choice is a **graph neural network (GNN)** that takes the quantum circuit's directed acyclic graph (DAG) representation as input. Each gate can be a node with connections indicating qubit flow, allowing the GNN to capture circuit structure (this follows the intuition from Quarl that local graph-based reasoning is effective ¹⁴). Alternatively, a simpler approach for the time available is to use a sequence model (e.g. an LSTM or small Transformer) that reads a serialized circuit (perhaps in QASM format) and suggests optimizations. By Week 2, the student will finalize this choice and outline the model architecture and features (e.g. encoding of gates, qubit indices, etc.).

With the approach defined, Week 3 is about **data generation**. Since we are not using an online RL loop, we need example circuits to train or guide our model. The plan is to leverage **self-supervision**: we can generate random or pseudo-random quantum circuits and then use TKET (and possibly other optimizers) to produce optimized versions. For instance, generate 50 random circuits (with random sequences of gates on say 3-5 qubits), then run `backend.get_compiled_circuit(circ)`¹⁵ to obtain a TKET-optimized circuit. This gives pairs of (original, optimized) circuits. Additionally, we can introduce known redundant patterns into circuits deliberately: for example, insert pairs of inverse gates (XX^\dagger or ZZ^\dagger) or add unnecessary rotations that cancel out, to create a “harder” unoptimized input with a known simpler equivalent. The model’s job will be to learn to remove such redundancies. We might also incorporate small-known optimal circuits from literature: e.g., for a given unitary, use brute-force or existing tools to find a minimal circuit, and use that as a target for the model. However, given the 2-month scope, a practical strategy is to stick with **synthetic data**: auto-generate circuits and rely on TKET or a brute-force method (for very small circuits) to find improved versions. The student will write scripts (using Python with pytket and possibly Qiskit for alternate optimizations) to produce a dataset. By the end of Milestone 2, there should be a dataset of many circuit pairs or at least a function that, given a circuit, can compute metrics like gate count and identify if an alternative sequence is shorter. The design of the ML optimizer is also finalized, with a clear plan to implement it next.

Milestone 3 (Weeks 4–5) – Prototype Implementation of the ML Optimizer: In weeks 4 and 5, the student will implement and train the machine learning model for circuit optimization. This involves coding the model (likely in PyTorch or TensorFlow) and feeding it the data from Milestone 2. If using a GNN, the student can utilize libraries like PyTorch Geometric to define a custom graph model where each gate is a node with features (gate type, qubits involved) and edges connect gates that share a qubit (capturing circuit topology). The model could be trained to output a prediction such as “how much can this circuit be further optimized” or even directly suggest an action (e.g. which gate to remove or which subcircuit to replace). A simpler framing is a classification or regression: for each candidate subcircuit (a sliding window of consecutive gates), predict if it can be rewritten into fewer gates. The training labels for this could be obtained by taking each window in the training circuits and checking if an optimized circuit had a shorter equivalent for that window. For instance, if a random 5-gate sequence on 2 qubits can actually implement the same unitary as a 3-gate sequence, label that sequence as optimizable. This may require using a brute-force solver for small sequences (perhaps using TKET’s `SynthesiseTket()` pass or the `FullPeepholeOptimise()` which applies local optimizations¹⁶). We will use such TKET passes programmatically to generate ground truth for local optimizations.

Once the training data is ready, the student trains the model (Week 4). This involves typical ML experimentation: splitting data, training the neural network to minimize error, and monitoring performance. By mid-project (around week 5), we expect to have a **prototype ML optimizer** that can take an input circuit (or segment) and predict an optimized outcome or at least identify opportunities for optimization. As a basic test, the student will input some circuits not seen in training (for example, a simple arithmetic circuit or a small QFT circuit) to see if the model correctly spots known optimizations (like cancelling inverse gates or merging rotations). The result of this milestone is a working ML component that we can integrate with the TKET compiler flow.

Milestone 4 (Week 6) – Integration with TKET and End-to-End Compilation: With the ML model in hand, the next step (Week 6) is to integrate it into an end-to-end compilation pipeline targeting Quantinuum’s

backend. The student will develop a routine that takes a high-level circuit, runs the standard TKET compiler, and then applies the ML-guided optimization as an additional pass. There are a couple of ways to integrate:

1. **Out-of-circuit post-optimization:** After TKET's `default_compilation_pass()`, convert the circuit into a format the ML model understands (perhaps a custom JSON or graph object). Run the model to find candidate optimizations, then apply those to the circuit. For example, if the model indicates that a subcircuit from gate i to j can be simplified, we can call a function to replace that segment with a simpler equivalent (either one suggested by the model if we trained it to output the new sequence, or by invoking a brute-force search if the model only flags it). The student will implement utility functions for splicing circuits – fortunately, TKET's `Circuit` class provides methods to add or remove gates, and we can use TKET's own functions to verify equivalence (perhaps by comparing unitary matrices for small segments).
2. **Custom TKET Pass:** Alternatively, TKET may allow custom compiler passes. If feasible, we can wrap the ML suggestion as a pass that iterates over the circuit. Given time constraints, the first approach is simpler: do a separate Python post-processing using the model.

During this integration, the student will make heavy use of Quantinuum's **device-specific features**. We will ensure that any transformation keeps the circuit valid for the target device (H1-1 or H1-2). For instance, if the model suggests a gate that's not native, we must rebase it to {Rz, PhasedX, ZZPhase} which are H1's native gates ¹⁷ ¹⁸. TKET provides an `AutoRebase` pass to handle that if needed ¹⁷. Also, after modifications, we will run a final `backend.get_compiled_circuit()` to enforce all device constraints (qubit count, connectivity, etc.) ¹⁹. By the end of week 6, we should be able to take an arbitrary input circuit, compile it with TKET, then apply our ML optimizer to get a *final* circuit. This final circuit will be submitted to Quantinuum's emulator backend for validation. Using `QuantinuumBackend('H1-1E')` with `noisy_simulation=False` for ideal behavior ¹³, we can simulate the circuit's output probabilities. If the circuit is meant to perform a known function (like QFT), we check that the output matches the expectation with high fidelity. If the circuit is just a random unitary, we can compute the fidelity by comparing its unitary matrix to the original's matrix (for small qubit counts, using TKET or NumPy to get unitary matrices). The integration milestone is accomplished when we can seamlessly go from high-level circuit -> TKET compile -> ML optimize -> execution, all using the Quantinuum stack.

Milestone 5 (Week 7) – Experimental Evaluation: In week 7, the project focus shifts to evaluating how well the ML-enhanced compiler performs. We will design a set of test circuits and compare three approaches: (a) **TKET-only compilation**, (b) **ML-enhanced compilation (our method)**, and optionally (c) a baseline from another compiler like Qiskit for reference. Test circuits may include: random circuits of various sizes (to test generality), known algorithms like Quantum Fourier Transform (which have known optimal gate counts), small instances of variational circuits (e.g. a few layers of a VQE ansatz), and any specific circuits from prior papers (for instance, the 3-qubit QFT example where RL found a 7 CZ gate version ² – we can see if our method also finds an equally short circuit). For each circuit, we will measure **gate count and depth** after compilation, and if possible the **fidelity** of execution on a noisy simulator. Quantinuum's emulator by default includes a noise model reflecting the real device ¹³, so by running with `noisy_simulation=True` (the default), we can get an estimate of how the shorter circuits improve final fidelity. The expectation is that circuits with fewer two-qubit gates should suffer less error on NISQ hardware ²⁰. We will confirm this by comparing the output distributions: e.g., measure the compiled circuit 1000 times on the noisy emulator and use the result to compute the fidelity (overlap with ideal output state). The student will also track the **compilation time** differences – did our ML pass add a lot of overhead?

(It might, if the model is large or if it triggers expensive searches for replacements, but we can note it anyway.) All these results will be recorded. If the ML optimizer is effective, we expect to see a reduction in gate counts compared to TKET alone, at least for some circuits, and potentially higher final fidelities on the noisy model. If the results are mixed, that's also a learning opportunity: the student can analyze which cases the ML helped and which it didn't. This might involve looking into the model's decisions – e.g., did it incorrectly suggest a rewrite that didn't yield improvement? We will use such insights to adjust the approach if time permits.

Milestone 6 (Week 8) – Refinement and Wrap-Up: In the final week, the student will refine the approach based on the experimental findings and complete the project deliverables. Refinement could include retraining the model with additional data focusing on cases where it underperformed, or adjusting the model's complexity to avoid any overfitting observed. For example, if certain optimizations were missed, we might augment the training data with more examples of those patterns. Conversely, if the model proposed some invalid transformations, we will add constraints in the inference stage (for instance, verify equivalence of any suggested change by checking unitary matrices – a safety net ensuring correctness). On the integration side, if there were issues with the Quantinuum API (like long queue times or job failures), we will document those and perhaps shift to local simulation for final tests. The student will also experiment with the **Quantinuum hardware** if accessible – even a single test run on the real H1-1 device for a small circuit could provide a valuable data point about real-hardware performance (since our project emphasized low-cost, this would only be done if free credits or educational access is available; otherwise, the noisy emulator suffices). Finally, the student prepares the project report and possibly a presentation. The report will include the project's motivation, methodology, results (with plots of gate count reduction and fidelity improvements), and a discussion. Crucially, it will highlight how our ML-enhanced compiler offers a new trade-off: a modest upfront training and implementation effort yielding a flexible optimizer that can adapt to hardware and potentially discover complex optimizations automatically – in contrast to the fixed strategies of rule-based compilers and the heavy training of prior RL compilers ①. By the end of Week 8, all milestones should be achieved, and the project concludes with a working prototype and a comprehensive analysis of its performance.

Implementation Details and Tools

(For each milestone, key tools, libraries, and instructions are summarized here for clarity.)

- **Environment & Tools:** Python will be the main language. We will use **pytket** (the Python API for TKET) ⑧ for circuit representation and compilation. The **pytket-quantinuum** plugin provides the **QuantinuumBackend** class to interact with Quantinuum's devices and emulators ⑨. Setting `device_name='H1-1E'` uses the 20-qubit H1-1 emulator (cloud-based), and `'H1-1LE'` if available for a local emulator ⑩ ⑪. We will install these via pip. For machine learning, **PyTorch** is recommended given its flexibility with graph networks. Additional libraries: NetworkX for manipulating circuit graphs, and possibly Qiskit or PyZX if we want to compare or generate alternative circuit forms. All development will be done in a local environment or a Jupyter notebook, with version control (git) to track code. No paid cloud compute is required; training can run on a standard CPU/GPU available to the student (the models will be relatively small given the small circuit sizes in the dataset).
- **Data & Training:** Data generation scripts will use TKET and possibly brute-force search for small subcircuits. If needed, the student can use **QSearch** or **SQUANDER** (open-source tools for optimal

circuit synthesis) for generating minimal circuits for small unitaries, though this is optional. Training the ML model will be done on the generated dataset using PyTorch. We will likely train for a few epochs (not too long, as dataset is not huge), and use validation to avoid overfitting. All training is offline on the student's machine – no cloud services required.

- **Quantinuum Integration:** The `QuantinuumBackend` requires login; after initial login, tokens are stored so the student can submit jobs seamlessly ¹⁰. For simulation, using the `E` (emulator) backends with `noisy_simulation=False` yields an ideal simulator ¹³, whereas the default (`True`) gives a realistic noisy simulation. We will use the **cost estimation** via `backend.cost(circuit, n_shots)` to get the HQC cost ¹², which helps ensure we stay within any credit limits (emulator runs typically don't consume credits ²⁴). The API will also allow retrieving results; for example, after `process_circuits`, we can fetch bitstring results from the `ResultHandle`. We will use those to compute observed fidelities by comparing to expected outputs (for known algorithms, this is straightforward; for random circuits, we'll compare to statevector simulation). Throughout, we ensure that any circuit sent to `QuantinuumBackend` satisfies device constraints by using the default compilation pass or at least calling `backend.validate_circuit(circ)` (if such a method exists) or relying on the built-in **syntax checker** backends like '`'H1-1SC'`' which can verify the circuit without execution ²².

- **Coding Suggestions:** Each milestone involves coding tasks:

- For data generation (Milestone 2), write a function `generate_random_circuit(n_qubits, depth)` that returns a pytket `Circuit` with random gates. Leverage TKET's support for adding gates like `circ.Ry(angle, qubit)` etc. and random angles. Then write `optimize_with_tket(circ)` that returns a copy of the circuit after `FullPeepholeOptimise` or `get_compiled_circuit` at high optimization level ¹¹. Use these to build training pairs.
- For the ML model (Milestone 3), if using a GNN, define a PyTorch Geometric `Data` object from a TKET circuit: nodes = gates, edges connect consecutive gates on same qubit. Node features can be one-hot encodings of gate type plus maybe numerical parameter values. The model could output a score per node or per edge indicating "cut here to remove a gate" or something, or a single score for the whole circuit (not as informative). A simpler approach: use a sequence model – represent the circuit as a sequence of gate tokens, maybe linearize by topological order. Then train the model to predict an "optimized sequence". This might be framed as sequence-to-sequence learning where input is the unoptimized QASM and output is the optimized QASM. The student might use an encoder-decoder Transformer from PyTorch with a small number of layers (so it trains in reasonable time). We will start with something small (maybe just predict the length or the existence of optimization, as a classification) to get the pipeline working, then potentially increase ambition to generating actual gate sequences.
- For integration (Milestone 4), code will involve scanning through a circuit. For example, iterate over all contiguous subcircuits of length L (for L from say 3 to 6 gates) and ask the ML model if this segment can be simplified. If yes, either directly apply a known simplification or use a brute-force verifier: e.g., try to use TKET's `SynthesiseTket` on that subcircuit to see if it finds a smaller equivalent. If it does, replace the segment. Because this could be computationally heavy, we'll prioritize segments the ML flagged with high confidence. Python pseudocode:

```

for start_index in range(len(circ_sequence) - L):
    subcirc = circ.extract_segment(start_index, L) # hypothetical method
    if model.predict_can_optimize(subcirc):
        new_sub = brute_force_optimize(subcirc)
        if new_sub.gate_count < subcirc.gate_count:
            circ.replace_segment(start_index, L, new_sub)

```

This illustrates how the ML model's prediction guides a classical optimization attempt. We will use TKET's data structures to implement `extract_segment` and `replace_segment` (one might do this by creating a new Circuit for the subsegment and then splicing, or by low-level editing of the DAG).

- For evaluation (Milestone 5), we will write scripts to run batches of circuits through both TKET and our enhanced compiler. Use Python's `time` module to measure runtime of compilation. Use `circ.n_gates` or iterate through `circ.get_commands()` to count specific gate types (especially two-qubit gates). Collect results in lists or pandas DataFrames and use matplotlib to plot comparisons. For fidelity, if using the emulator, we can do something like:

```

result = backend.process_circuits([compiled_circ], n_shots=1000,
noisy_simulation=True)
counts = backend.get_result(result).get_counts() # obtain counts of bitstrings
fidelity = compute_fidelity(counts, ideal_distribution)

```

The `compute_fidelity` function we'll implement to compare the probability distributions. For `ideal_distribution`, we might get it from a noiseless simulation of the circuit or from theoretical expectation. In cases where the circuit performs a known unitary on a basis state (like QFT on $|100\rangle$ etc.), we know the expected correct output and can define fidelity as probability of getting the correct output.

- **Milestone Tracking:** Each week's tasks will be documented, and code will be kept in a repository. The student will use issue tracking to ensure each milestone's tasks (e.g., "generate 100 random circuits and optimize them" or "train GNN model to >90% accuracy on detecting optimizable patterns") are completed. Regular meetings/check-ins with a mentor (if available) will help troubleshoot any roadblocks, especially in integrating with the Quantinuum API or tuning the ML model.

By following this implementation plan, the project stays **feasible** for a single student: we leverage existing tools (TKET, simulators, PyTorch) heavily and keep the ML model at a manageable scale. The emphasis is on implementation and experimentation rather than large-scale theory, aligning with the user's requirement of using open-source tools and avoiding heavy reliance on paid cloud resources.

Creative Extension Idea

As a forward-looking extension, one could explore **Large Language Models (LLMs) for Quantum Compilation**. In a surprising crossover between quantum computing and modern AI, consider treating a quantum circuit's text representation (like QASM or Quil code) as a "program" and using an LLM-based code optimizer to improve it. This extension would involve fine-tuning a GPT-style model on a corpus of quantum

circuits: for example, generate thousands of random circuits and their optimized equivalents (using our ML compiler or TKET), then train the model to translate an unoptimized circuit into an optimized one. The model could learn high-level patterns and even **invent novel shortcuts** that human-designed passes or our earlier ML might miss. Such a quantum-savvy LLM could take into account the context of the entire circuit (something our local GNN might struggle with) and output a refined circuit. In essence, this turns quantum compiling into a sequence-to-sequence prediction task, akin to how AI models can optimize classical code. The idea is unconventional – effectively using a “Quantum Circuit GPT” – and could inspire follow-up research into foundation models that understand quantum operations. This extension would be more computationally intensive and speculative, but if successful, it might unlock a new paradigm where **AI-driven compilers** improve themselves by learning from vast numbers of examples, going far beyond what fixed algorithms or even reinforcement learning agents can do. Such an approach could eventually be combined with hardware-in-the-loop fine-tuning: for instance, an LLM that suggests circuit modifications which are then validated on a real quantum device, creating a feedback loop for continuous learning. While implementing this is beyond a 2-month project, the concept pushes the boundary of how we think about quantum compiler design, injecting creativity and the latest AI advances into quantum software development.

Ultimately, the NeuroTKET project and its extension lay the groundwork for **autonomous quantum compilers** – tools that not only apply human-crafted rules but actually *learn* the rules of optimal quantum program transformation from data. This could be a game-changer as quantum hardware grows and quantum programs become too complex for manual optimization, ensuring that even an advanced undergraduate’s 2-month project contributes a small but significant step toward that vision.

Sources:

1. Moro *et al.*, “Quantum compiling by deep reinforcement learning,” *Commun. Phys.* **4**, 178 (2021) [1](#).
2. Zhang *et al.*, “RL-based quantum compiler for a superconducting processor,” arXiv:2406.12195 (2024) [2](#) [20](#).
3. Li *et al.*, “Quarl: A Learning-Based Quantum Circuit Optimizer,” arXiv:2307.10120 (2023) [4](#) [6](#).
4. **Quantinuum TKET Documentation** – *pytket* and *pytket-quantinuum* API guides [11](#) [22](#) [23](#) [12](#).
5. **Quantinuum Backend Info** – Device and emulator availability (H1-1, H1-1E, etc.) and usage of default compilation passes [21](#) [17](#).
6. Smith *et al.*, “MQT Predictor: predicting optimal quantum compiler passes,” in *Proc. QC* (2022) [7](#) (illustrating prior ML approach limited to few passes).

[1](#) [3](#) Quantum compiling by deep reinforcement learning | Communications Physics

[https://www.nature.com/articles/s42005-021-00684-3?
error=cookies_not_supported&code=6556e674-13c1-464c-9552-782d42ec4a6b](https://www.nature.com/articles/s42005-021-00684-3?error=cookies_not_supported&code=6556e674-13c1-464c-9552-782d42ec4a6b)

[2](#) [20](#) Quantum Compiling with Reinforcement Learning on a Superconducting Processor
<https://arxiv.org/html/2406.12195v1>

[4](#) [5](#) [6](#) [14](#) [2307.10120] Quarl: A Learning-Based Quantum Circuit Optimizer
<https://arxiv.org/abs/2307.10120>

[7](#) Optimizing Quantum Circuits, Fast and Slow
<https://pages.cs.wisc.edu/~aws/papers/asplos25.pdf>

