NeuroTKET: An ML-Enhanced Quantum Circuit Compiler Using TKET

Nathan Delcid

BS Computer Science | CU Boulder '26

Abstract

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 by developing a lightweight learning-based optimizer that can:

- Adapt to hardware constraints
- Learn circuit simplification patterns
- Leverage existing TKET optimizations as a baseline

Leveraging Quantinuum's open-source TKET toolchain and APIs, I'd implement an ML-guided optimization module that works alongside TKET's native passes. This work would include building and testing the 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" (NeuroTKET)** 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

Existing quantum compilers rely heavily on deep RL agents to find gate sequences. This project explores a new machine learning paradigm for quantum compilation by adhering to the following guidelines:

1. **Avoid heavy pre-training and complex action-space handling of RL-based compilers.**
    - Prior RL compilers (e.g. Quarl [3] and others) had to address enormous action spaces and state representations with specialized neural architectures, and required long training phases to learn a policy. In contrast, this approach would use self-supervised learning to guide circuit optimizations, with the hopes of significantly simplifying the learning problem.
2. **Instead of models learning from scratch, leverage TKET's powerful existing optimizations as a baseline and then learn residual improvements.**
    - This means the ML module 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 [3] RL agent surprisingly learned complex optimizations like rotation merging that usually require a dedicated pass; this project aims to achieve similar advanced optimizations via a different learning technique.
3. **Prioritize incorporating hardware-awareness directly into the compiler.**
    - Using Quantinuum's API and device noise models, the ML module could learn to prefer transformations that improve fidelity on Quantinuum hardware. Traditional compilers optimize mostly for gate count or depth, but this compiler, enhanced by ML, 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.

In summary, the project's novelty lies 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 could chart a new path for

quantum compiling by using modern ML algorithms (graph neural networks, pattern recognition, etc.) to automate circuit improvements.

This strategy has seen only limited exploration (e.g. Smith et al. [6] targeted a few compiler passes, but no comprehensive ML-driven optimizer exists). By the end, I'd like to demonstrate a prototype that can automatically discover circuit reductions/reorders (that a human or default compiler wouldn't easily find) by leveraging the open-source resources at hand.

Milestones

Milestone 1: Background Research & Environment Setup

Grounding the project in existing work and preparing tooling. Here I'd 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 [3], and any non-RL approaches). This literature review will clarify how my approach differs (e.g. no Q-learning loop, focusing on supervised pattern learning instead).

This milestone would also involve installing and configuring `pytket` and the `pytket-quantinuum` extension, and ensuring access to Quantinuum's systems by creating an account and obtaining the API key. It is here where I will also begin to practice TKET circuit creation, compilation, and execution.

```
# Create a 2-qubit circuit in pytket
# Compile it for the Quantinuum H1-1E emulator
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
```

This code above would transpile the circuit to satisfy the H1-1 device constraints and apply level-2 optimizations by default. Next steps here would be validating 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). It would also include measuring 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. 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). These baseline experiments (perhaps on a known small circuit like a 3-qubit QFT) will familiarize me with TKET's API and Quantinuum's stack. Upon completion, I'd expect a working environment, some baseline data, and a clearer plan of attack grounded in literature insights.

Milestone 2: Design of ML-Assisted Optimizer & Data Preparation

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. This involves deciding 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 [3] that local graph-based reasoning is effective). Alternatively, a simpler approach 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.

With the approach defined, the next phase focuses on data generation. The plan here is to leverage self-supervision: 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)` to obtain a TKET-optimized circuit. This gives pairs of (original, optimized) circuits. This step could also introduce known redundant patterns into circuits deliberately: for example, insert pairs of inverse

gates (XX^+ or ZZ^+) 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. Here it might be smart to 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.

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. Write scripts (using Python with pytket and possibly Qiskit for alternate optimizations) to produce a dataset. Upon completion, 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: Prototype Implementation of the ML Optimizer

This milestone involves implementing and training the 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, an approach could be to 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 optimizations). The idea is to use such TKET passes programmatically to generate ground truth for local optimizations.

Once the training data is ready, the next step would be to train the model. This involves typical ML experimentation: splitting data, training the neural network to minimize error, and monitoring performance. The goal is 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. Basic testing could include inputting 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 would be a working ML component that I can integrate with the TKET compiler flow.

Milestone 4: Integration with TKET and End-to-End Compilation

With the ML model in hand, the next step is to integrate it into an end-to-end compilation pipeline targeting Quantinuum's backend. This involves developing 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 integration options that are immediately apparent:

- **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 trained to output the new sequence, or by invoking a brute-force search if the model only flags it). I'd 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).

- **Custom TKET Pass:** Alternatively, TKET may allow custom compiler passes. If feasible, I can wrap the ML suggestion as a pass that iterates over the circuit. The first approach is simpler: do a separate Python post-processing using the model.

During this integration, it is paramount to 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. TKET provides an AutoRebase pass to handle that if needed. Also, after modifications, I'll run a final

`backend.get_compiled_circuit()` to enforce all device constraints (qubit count, connectivity, etc.). Upon completion, we should be able to take an arbitrary input circuit, compile it with TKET, then apply the optimizer to get a final circuit. This final circuit will be submitted to Quantinuum's emulator backend for validation. Then, simulate the circuit's output probabilities by using `QuantinuumBackend('H1-1E')` with `noisy_simulation=False` for ideal behavior. If the circuit is meant to perform a known function (like QFT), check that the output matches the expectation with high fidelity. If the circuit is just a random unitary, 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

$$\text{High-Level Circuit} \rightarrow\; \text{TKET Compile}\; \rightarrow\; \text{ML Optimization} \rightarrow\; \text{Execution}$$

All by using the Quantinuum stack

Milestone 5: Experimental Evaluation

Here focus shifts to evaluating how well the ML-enhanced compiler performs. This involves designing a set of test circuits and compare three approaches:

1. TKET-only compilation
2. ML-enhanced compilation (this project's method)
3. (optionally) 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)
- Any specific circuits from prior papers (for instance, the 3-qubit QFT example where RL found a 7 CZ gate version).

For each circuit, 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, 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. Confirming this includes 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). Additionally, I'll track the compilation time differences (i.e., did my ML pass add a lot of overhead?)

- **NOTE:** It might, if the model is large or if it triggers expensive searches for replacements, so we'll note it if so.

All these results will be recorded. If the ML optimizer is effective, expectations would be 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 where I can analyze which cases the ML helped and which it didn't by (perhaps) looking into the model's decisions.

Milestone 6: Refinement and Wrap-Up

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, perhaps augment the training data with more examples of those patterns. Conversely, if the model proposed some invalid transformations, 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), document those and perhaps shift to local simulation for final tests. 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 this project emphasizes low-cost, this would only be done if free credits or educational access is available; otherwise, the noisy emulator suffices).

Finally, prepare 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 this 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.**

This is to be contrasted with the fixed strategies of rule-based compilers and the heavy training of prior RL compilers. Here, the hope is that the **project concludes with a working prototype and a comprehensive analysis of its performance**.

References

[1] Moro, L., et al. "Quantum compiling by deep reinforcement learning." *Communications Physics* 4, 178 (2021). https://www.nature.com/articles/s42005-021-00684-3

[2] Zhang, Y., et al. "RL-based quantum compiler for a superconducting processor." *arXiv preprint arXiv:2406.12195* (2024). https://arxiv.org/abs/2406.12195

[3] Li, G., et al. "Quarl: A Learning-Based Quantum Circuit Optimizer." *arXiv preprint arXiv:2307.10120* (2023). https://arxiv.org/abs/2307.10120

[4] Quantinuum TKET Documentation – pytket and pytket-quantinuum API guides. https://docs.quantinuum.com/

[5] Quantinuum Backend Info – Device and emulator availability (H1-1, H1-1E, etc.) and usage of default compilation passes. https://docs.quantinuum.com/

[6] Smith, J., et al. "MQT Predictor: predicting optimal quantum compiler passes." In *Proceedings of the International Conference on Quantum Computing* (2022). https://pages.cs.wisc.edu/ (illustrating prior ML approach limited to few passes).