A. Artifact Appendix

A.1 Abstract

In this letter we provide detailed information that will facilitate the artifact evaluation process. In the artifact checklist section, we present brief information about this artifact, and outline basic requirements to reproduce the experiment results. Then we describe the directory tree of our codebase and go into more detail about the requirements. Finally, in the experiment workflow section we explain step by step how to reproduce the experiments

A.2 Artifact check-list (meta-information)

- **Algorithm:** we propose QuCT, a unified framework for extracting, analyzing, and optimizing quantum circuits. The fidelity prediction model performs a linear transformation on all gate vectors and aggregates the results to estimate the overall circuit fidelity. By identifying critical weights in the transformation matrix, we propose two optimizations to improve the fidelity. In the unitary decomposition model, we significantly reduce the search space by bridging the gap between unitary and circuit via gate vectors.
- **Program:** Python **version**
- **Model:** We define a two-step model. In the upstream model, each gate is transformed into a vector that captures its neighboring circuit features. The downstream models take these vectors as input for various analysis tasks, such as fidelity prediction and unitary decomposition.
- **Data set:** We prepare two datasets, one containing random circuits and algorithmic circuits, each circuit containing its fidelity executed on IBM QPU; the other with the same structure, and the execution results come from our own.
- **Run-time environment:** Ubuntu 22.04, Python 3.9 or higher
- **Hardware:** Intel Xeon Gold 6348 CPU
- **Output:** The results of the key experiments are displayed in the form of graphs and tables, which compare our method and the baseline method, and we expect our results to be better than the baseline results.
- **Experiments:** We provide IPython/Jupyter notebook, that can be used to reproduce the results of the experiments
- **How much disk space required (approximately)?:** The codebase, datasets and trained model take up about 20GB in total.
- **How much time is needed to prepare workflow (approximately)?:** Less than an hour
- **How much time is needed to complete experiments (approximately)?:**
  a) Training an upstream model could take from 1 minute to 10 minutes, depending on the number of qubits.

  Before training a downstream model for the fidelity prediction task, we need to get the labels of circuits in the dataset by simulating or executing on a QPU. Executing 3000 circuits on a QPU with 5000 shots for each circuit takes about 2 hours.

  Training a fidelity prediction model with 200 epochs takes 15 minutes to 1 hour as the number of qubits increases. In order to evaluate scalability, we need to simulate up to 350 qubits circuits and train a fidelity prediction model, so it could be extremely time-consuming. It costs more than 1 day on an Intel Xeon 64-Core Processor.

  Applying a compilation-level optimization on one circuit costs less than 1 minute. It takes 1 hour to evaluate how the size of the path table in an upstream model changes as the qubit number increases.

  For the unitary decomposition task, constructing a Unitary-to-Vector model could take 0.5 hour and decompose a unitary of a 5 bit random-generated circuit within 8 hours in comparison to more than 3 days with QFast method.
- **Publicly available?:** Yes

A.3 Description

Below we introduce some important files and directories in the codebase. For a more detailed description, please refer to the **README** in the GitHub repository.

- **circuit/:** We randomly select a gate of the circuit, and choose one of the gates of its former layer and the parallel layer. These two gates constitute a path, and the length of path is set by **n_step**. All the different paths in the data set form a path table. The vectorization of a gate is the one-hot representation of the path table corresponding to the path it starts from
- **upstream/:** This sub-directory holds code related to the generation of path table and the vecterization of circuits.We randomly select a gate of the circuit, and choose one of the gates of its former layer and the parallel layer. These two gates constitute a path, and the farthest distance is set by n-step. All the different paths in the data set form a path table. The vectorization of a gate is the one-hot representation of the path table corresponding to the path it starts from.
- **downstream/:** This sub-directory contains the implement of two different downstream sub-tasks. For the sake of clarity, we will describe this part separately in the next section.
- **ae/:** This sub-directory holds the IPython/Jupyter notebook, for running experiments. Detailed introduction in section A.5
- **simulator/:** This sub-directory holds code related to the simulation of circuit with and without noise. The error of the gate itself is modeled as bit flip, phase flip, and depolarizing errors. The error from the interaction between gates is modeled by applying an RX operator with a random angle ($[-\pi/20, \pi/20]$) to a 1-step path. In other words, the two gates of a 1-step path will be added with the RX operator if this path is injected with a noise. To make the simulation results more realistic, we simulate each circuit multiple times, each time adding a layer of random gates to simulate different inputs, and use the average as the final result.
- **plot/:** This sub-directory holds code related to the figure presentation.

We then describe the files in the **downstream/ sub-directory** that contains the implement of two different downstream sub-tasks. Each sub-directory corresponds to a downstream sub-tasks.

- **downstream/fidelity_predict:** This sub-directory holds code related to training a model and using it to predict the fidelity of a circuit.The number of parameters that can be trained is the same as the size of path table, and each parameter represents the error rate of its corresponding path. During training process, we adopt the strategy of starting training with circuits that have fewer gates for better convergence.
- **downstream/synthesis:** This sub-directory holds code related to the decomposition of a Unitary including the method we propose and the baseline we compare to. In sub-directory **synthesis_baseline**, we provide APIs to run benchmarks of CSD, QSD ,Qiskit, QFast, QSearch systhesis. All these baseline methods are encapsulated so that thay can be run easily. In QuCT, we consider the gate vector as a search candidate. By identifying

the vectors that may be involved in the circuit of the target unitary, we can prune the candidate space. We set the number of candidates to 5.

A.3.1 How to access

You can access our codebase from here:https://github.com/JanusQ/QuCT-Micro2023/tree/master

A.3.2 Hardware dependencies

Experiments require least 100G memory and 20G storage.

A.3.3 Software dependencies

Software experiments require Python 3.9 or higher. Other dependent Python packages are listed in **requirements.txt**.

A.3.4 Data sets

The fidelity dataset is a hardware-dependent dataset, which is built by collecting the real results on the target quantum device. We randomly divide the dataset into training and testing sets. We use the `random_circuit` API of Qiskit to generate randomized circuits, which are then mirrored to make the final circuit identity. The circuit depth ranges from 5 to 100. Each circuit is sampled 2000 times on the target device.

A.3.5 Models

Firstly, vectorization model transformes each gate into a vector that captures its neighboring circuit features. Secondly, fidelity model takes vectors as input and provides predictions of the fidelity of a circuit. Thirdly, Unitary-to-Vector model help to find the candidate vectors that tend to be part of the resultant circuit in the target unitary.

A.4 Installation

- Create a virtual environment with a Python version of at least 3.9.
- Install all dependent Python packages in requirements.txt using pip
- We recommend using qiskit==0.41.0 with jax==0.4.12 to avoid unexpected compatibility issues.

A.5 Experiment workflow

Below we describe the process of verifying the software implementation.

Evaluate fidelity prediction

- **Train a fidelity prediction model:** You can execute the Python script **test_predict.py** in **ae/test_predict**. The script uses the dataset we provide and generates models of config-0 to config-2, as Figure 9 shown in the paper. You can change parameter **n_steps** if you want to try other config.
- **Evaluation of predict performance:** You can execute the Python scripts **compare_predict_performance.ipynb** in **ae/ test_predict** and compare the performance of models we trained with RB-based model and XEB-based model. The parameters of RB-based model and XEB-based model are collected from quantum device.
- **Evaluateion of scalability:** We design seven Qiskit [**?**] simulators to simulate 50, 100, 150, 200, 250, 300, and 350 qubits. We construct separable circuits introduced in Section **??** to make them efficiently simulated. The error of the gate itself is modeled as bit flip, phase flip, and depolarizing errors. The error from the interaction between gates is modeled by applying an RX operator with a random angle ($[-\pi/20, \pi/20]$) to a 1-step path. In other words, the two gates of a 1-step path will be added with the RX operator if this path is injected with a noise.To demonstrate the scalability, you can execute the Python scripts **test_scalability.py** in **ae/test_scalability** to evaluate QuCT on 50-qubit to 350-qubit. Note that simulating so many qubits can be time consuming. In the script, we also compare the result with the RB-based method.
- **Evaluation of predict on algorithm:** We collected 13 different algorithms as benchmarks to compare the accuracy of QuCT and RB based model for their fidelity prediction. You can run the Python scripts **test_predict_alg.py** in **ae/test_predict_alg** to see the result.
- **Evaluation of optimization:** To compare different routing and scheduling schemes for fidelity optimization on algorithms benchmarks, You can just run **eval_optimization.py** in **ae /optimization**. You can also run jupyter notebook **eval_calibration.ipynb** to see how calibration-level optimization works.
- **Evaluation of time drift:** We collected the results of the same circuit for a continuous period of time, and compared the performance of finetune the quct model with different numbers of circuits. You can run the Python scripts **test_time_drift.ipynb** in **ae/time_drift** to see the result.
- **Evaluation of size of path table:** You can run the Python scripts **table_size.py** in **ae/table_size** to see how size of path table in a upstream model changes as the qubit number increases.

Evaluation of Unitary Synthesis

- **Baseline synthesis methods:** The baselines include CSD, QSD, and QFast. You can execute the a Python script **eval_random.py** in **downstream/synthesis/synthesis_baseline** to view the performance of these baseline methods, which includes multiple-process time, number of two-bit gates, circuit depth, and many other metrics. All results are serialized and stored in the directory **ae/synthesis/baseline**
- **QuCT synthesis:** You can execute the a Python script **quct_synthesis.py** in **ae/synthesis**. The script generate a synthesis model and load the unitaries baseline use to decompose. At last we compare the performance of all the synthesis method we mentioned.

A.6 Evaluation and expected results

Following the workflow described above, you should be able to reproduce the results in Figure 3, Figure 5, Figure 9, Figure 10, Figure 11, Figure 12, Figure 13, Table 4 of the paper.