KetGPT – Dataset Augmentation of Quantum Circuits using Transformers

Boran Apak[0009−0000−3370−7473], Medina Bandic[0000−0003−4670−0988], Aritra Sarkar[0000−0002−3026−6892], and Sebastian Feld[0000−0003−2782−1469]

Quantum Machine Learning group, QuTech,
Department of Quantum & Computer Engineering,
Delft University of Technology, The Netherlands
`boranapak1998@gmail.com,{m.bandic,a.sarkar-3,s.feld}@tudelft.nl`

Abstract. Quantum algorithms, represented as quantum circuits, can be used as benchmarks for assessing the performance of quantum systems. Existing datasets, widely utilized in the field, suffer from limitations in size and versatility, leading researchers to employ randomly generated circuits. Random circuits are, however, not representative benchmarks as they lack the inherent properties of real quantum algorithms for which the quantum systems are manufactured. This shortage of 'useful' quantum benchmarks poses a challenge to advancing the development and comparison of quantum compilers and hardware.
This research aims to enhance the existing quantum circuit datasets by generating what we refer to as 'realistic-looking' circuits by employing the Transformer machine learning architecture. For this purpose, we introduce KetGPT, a tool that generates synthetic circuits in OpenQASM language, whose structure is based on quantum circuits derived from existing quantum algorithms and follows the typical patterns of human-written algorithm-based code (e.g., order of gates and qubits). Our three-fold verification process, involving manual inspection and Qiskit framework execution, transformer-based classification, and structural analysis, demonstrates the efficacy of KetGPT in producing large amounts of additional circuits that closely align with algorithm-based structures. Beyond benchmarking, we envision KetGPT contributing substantially to AI-driven quantum compilers and systems.

Keywords: quantum circuits · generative AI · dataset augmentation · Quantum Assembly · quantum compilation

1 Introduction

The journey from knowledge and rule-based artificial intelligence to the contemporary era of data-driven deep neural networks-based machine learning (ML) has marked significant milestones in artificial intelligence (AI). This type of AI, termed deep learning (DL), focuses on recognizing and extracting patterns from vast datasets. A proliferation of popular DL models and architectures contributed to use cases such as image and speech recognition, sequence prediction, and reinforcement learning. However, the application landscape changed

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_17

dramatically with the emergence of generative models [16], such as generative adversarial networks (GAN) and variational autoencoders (VAE). These models marked a profound shift in the capabilities of DL, allowing machines not only to recognize patterns in the data but also to generate new, coherent data that closely resembles the patterns learned from the training data.

Amid this diversity, the model that stands out in recent advances is the generative pre-trained transformer (GPT) [34] based on the transformer architecture [42]. Transformers achieve impressive performance on tasks like realistic text and code generation [30,29] by capturing important information about the structure of sequences of data. GPT's ability to leverage massive scale with billions of parameters and self-supervised learning makes it the model of choice for natural language understanding and generation. A wide spectrum of AI applications can be formulated as a language modeling and generation task, like chatbots, text summarization, question answering, code generation, medical diagnosis, and legal document review.

Simultaneously, another groundbreaking technology is being developed: quantum computers. Quantum computers can solve certain problems faster than classical computers [28] by employing information processing capabilities governed by the laws of quantum mechanics. To solve such problems, quantum algorithms, typically expressed as quantum circuits, need to be executed on quantum computers. Besides serving the target use case, these circuits, defined in quantum assembly languages (QASM) [8], are often also used to characterize, evaluate, and benchmark the quantum processors and related system software. Moreover, system software, like the quantum compiler, often employs DL-based approaches to tackle the complexity of controlling large quantum processors. This presents the need for large datasets of quantum circuits [27,11] for the training of the ML-based quantum compilation passes, such as routing and mapping the circuits to a quantum processor. However, at the moment, only a handful of quantum algorithms [22] are known to provide quantum computational benefits. Due to the lack of large quantum circuit databases, these ML-based compilation techniques resort to randomly generated quantum circuits to train the model. This use of unrepresentative training data can critically affect the performance of the quantum computer when deployed for pragmatic use cases.

In an attempt to address this problem in quantum computing and inspired by the paradigm shift in language generation, in this work, we *employ transformer models to generate realistic-looking quantum circuits to augment quantum circuit datasets.*

This paper's contribution is threefold:

1. Introducing KetGPT, a transformer model capable of generating realistic-looking quantum circuits in the QASM language;

2. Developing a method to determine the quality of the generated QASM code using a different transformer model specifically designed for this task; and

3. Analyzing the generated circuits by extracting their structural parameters and comparing them to those of previously existing circuits.

KetGPT can immediately be applied to the following use cases:

• **Extending quantum circuit benchmarks datasets**: KetGPT circuits offer a valuable expansion to existing circuit suites, such as those in [33,5], commonly employed for benchmarking and comparing quantum compilers and systems. Unlike typical synthetic circuits that consist of random gates on random qubits, KetGPT circuits emulate the behavior of real quantum algorithms, enhancing their relevance as benchmarks. Moreover, compared to the current practice of employing entirely random circuits with consistent width and depth, they present a compelling alternative for evaluating success metrics like quantum volume [7]. Given that quantum computers are designed to accelerate specific algorithms challenging for classical computers, assessing them using circuits that closely resemble these algorithmic structures is imperative. A dataset of KetGPT-generated quantum circuits is available as part of this software in Sec. 6.

• **Automating quantum system software**: Recent research uses machine learning models to enhance quantum compilation and error correction [27,11,31,1]. The substantial data required for training these models often leads researchers to resort to generating random circuits. However, a system that solves a certain problem should be trained on representative problem instances. Therefore, training a compiler to handle realistic circuits is more beneficial than training it on a random sample of gates, which makes KetGPT ideal for such a purpose [5]. In an ongoing project, KetGPT is being used to train a reinforcement learning agent for quantum circuit mapping on noisy quantum processors.

The remainder of this paper is structured as follows: The transformer models are introduced in Sec. 2. Sec. 3 introduces the main contribution of this work, KetGPT, a transformer model specifically designed to generate QASM files useful for benchmarking quantum system software. Additionally, a method is proposed to quantify how realistic these QASM files are. In Sec. 4, the generated code is examined and results are presented and discussed. Ultimately, Sec. 5 contains the conclusion of this work and presents suggestions for future work.

2 Evolution and Structure of Transformers

Transformer models, as introduced in the groundbreaking work [42], have changed the landscape of natural language processing. Their applications extend to code generation [40,29] and music generation [2]. Renowned for their proficiency in capturing dependencies within sequential data, these widely adopted machine-learning models have proven effective in various domains.

Before the advent of transformers, conventional models for natural language processing tasks, such as text generation, primarily relied on Convolutional Neural Networks (CNN) [24], Recurrent Neural Networks (RNN) [37], and Long Short-Term Memory networks (LSTM) [18]. However, these models encountered several challenges, including difficulties in handling long-range dependencies and a lack of parallelizability [42]. A transformer, on the other hand, is a highly parallelizable model, well-suited for training on extensive datasets, that excels at

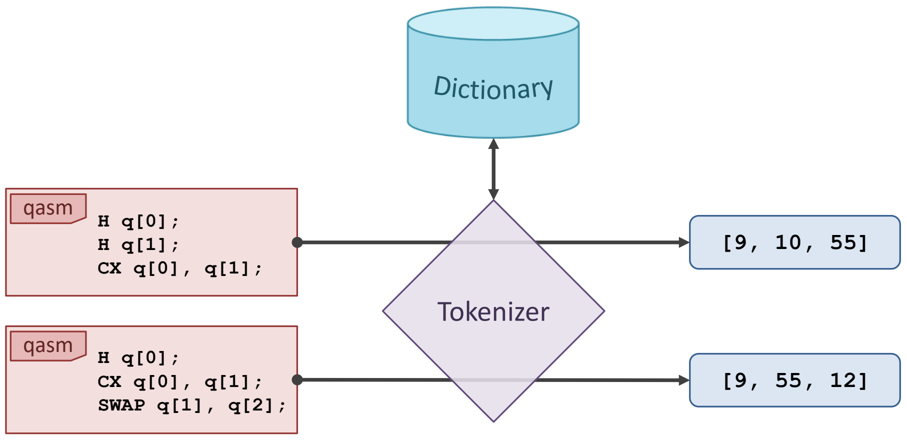


Fig. 1: Tokenization Example. A sequence of QASM operations (in text file form) is provided as input, and each statement (a line of QASM code) is assigned to a number. The number assigned to each statement does not have an intuitive meaning; rather, it just depends on how the tokenization algorithm orders its vocabulary. Consequently, tokenizing a sequence of statements will create a list of numbers. It is important to note that both gate and qubit(s), we apply the gate on, matter for the assigned token. For instance, h q[0]; and h q[1]; would have different numbers assigned as shown.

capturing longer-range dependencies and, therefore offers a significant improvement over earlier models.

In what follows, we review the three main components of the transformer model with quantum assembly language as the data.

2.1 Tokenizer

It is well known that performing any kind of computations on strings necessitates converting them to numerical *tokens* through a process called *tokenization*. While this tokenization step is not explicitly outlined in the transformer architecture defined in [42] (as it falls under the domain of dataset preparation), it plays a crucial role in comprehending how information flows through a transformer model. A tokenizer plays a significant role in our case as using QASM code as input requires a different preprocessing type than with standard text. An example of the QASM code tokenization process is presented in Fig. 1.

To fully describe a tokenization process, it is required to have a system for segmenting a sequence and a ‘dictionary’ to establish the numerical association for each possible segment encountered using this segmentation system. There are different types of tokenization algorithms available. For instance, instead of the scheme shown in Fig. 1, every character can be converted to a number.

Thus, `h q[0];` would be tokenized into 7 integers, one for each character and whitespace, instead of just a single token.

2.2 Feed-forward Neural Network

Neural networks [12] play a key role in various machine-learning approaches and are one of the fundamental segments of transformer models. They consist of a series of layers that each perform a linear operation on the input followed by a (non-linear) activation function.

To be precise, the value of each node in the network will be a linear combination of the values of the nodes in the previous layer weighted by the corresponding weights, passed through an activation function. Then a non-linear activation function (such as softmax [6] or ReLu [17]) is applied so that the network can capture complex non-linear patterns.

A Feed-forward neural network is fully defined by specifying the number of layers, the number of nodes in each layer, the weights of every connection between nodes of a layer and a previous layer, a bias per node and the activation function per layer. To train a network, the desired architecture is initialized with (random) weights and biases. During training, the inputs are iteratively presented to the network and the weights and biases are adjusted to progressively align the network's output with the expected output for each specific input. This adjustment is typically done using a method called Stochastic Gradient Descent [36]. In this paper we are not focusing on the details of the neural networks, even though it represents the core of the transformer model, as it is widely and generally used as a base of most machine learning models. Instead, we will focus on the segments of the transformer that are specifically significant for our model, like *self-attention*.

2.3 Self-attention

Self-attention is a mechanism that helps a transformer understand the relation between words and represents the main innovation in transformer models. Consider the sentence, "The computer executes the program because it is told to." Humans effortlessly discern that "it" refers to the computer, not the program, but making automated systems distinguish this difference is very challenging. The inclusion of a self-attention component empowers transformers to establish such connections.

The input to the attention mechanism consists of queries, keys, and values. Each token in the input sequence corresponds to one query and key vector with dimension d_k and a value vector with dimension d_v, but for computational purposes, the queries, keys and values for all tokens are packed into, respectively, matrices Q, K and V. Thereafter, the main equation [42] describing the attention process is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{1}$$

where softmax is the softmax function [6] and K^T is the transpose of the K matrix.

The underlying idea of this equation is in the QK^T term, representing the dot product between queries and keys to discern their "inter-relation." Subsequently, this information forms an attention matrix akin to a correlation matrix. However, unlike a correlation matrix with values between -1 and 1, the attention matrix adopts the form of a probability distribution, with values ranging from 0 to 1. The $\sqrt{d_k}$ scaling factor is there to obtain a more dimension-independent dot product, which helps train the network easier [42]. Multiplying this attention matrix with V produces the final result, enriching the original matrix V with insights into the inter-relations between queries and keys. For instance, elements with low scores in the attention matrix, close to 0, are drowned out. To illustrate, in the context of encoding the sentence "The computer executes the program because it is told to." represented by matrices Q, K, and V, the operation Attention(Q, K, V) returns a matrix that embodies this sentence with information about the inter-relations between the words (e.g., clarifying that "it" refers to the computer and not the program).

3 KetGPT - Transformers for Quantum Circuit Generation

This section presents KetGPT, a novel software tool designed to generate quantum algorithm-based circuits. These circuits can serve as essential benchmarks for evaluating the performance of both existing and forthcoming quantum systems. Within this section, we delve into the technical intricacies of KetGPT, offering a comprehensive understanding of its architecture and methodology. Fig. 2 shows an overview of the KetGPT design and overall workflow.

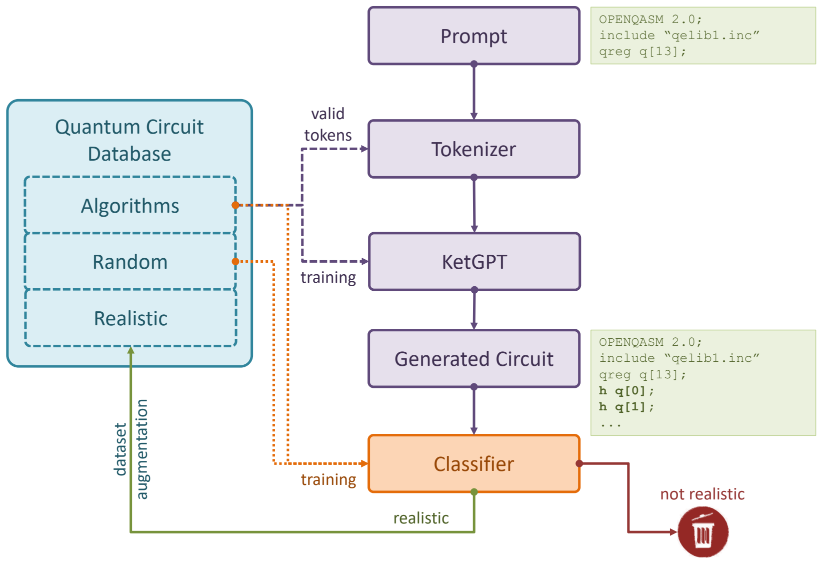
3.1 Input Dataset and Data Preprocessing

Several datasets of quantum circuits suitable for benchmarking are available [26,43,5], including MQT Bench [33], which is utilized in this study. QASM files were generated to depict circuits implementing algorithms spanning 2 to 100 qubits, employing OpenQASM 2.0 [8]. In cases where algorithms were incompatible with a specific qubit count, such as those requiring an uneven number of qubits, all valid circuits within the feasible range were generated. The full dataset and additional details can be found in Sec. 6.

The files taken from the dataset require preprocessing in order to comply with the transformer model. This involves making minor adjustments to the QASM files in the dataset (e.g., removing comments). Due to technical constraints – specifically, the model's incapacity to process large files – a maximum circuit length of 1024 QASM statements is enforced. This limitation is specific to the hardware's RAM constraints and not a general technical restriction. Following the preprocessing step, the final dataset comprises 713 QASM files.

3.2 Generator: Architecture and Tokenizer

When it comes to generating text and code, a decoder-only transformer architecture [40] is a popular choice. Accordingly, for the generation of QASM files, we have opted for the GPT-2 model architecture [35], known for its use of a decoder-only transformer.



```
OPENQASM 2.0;
include “qelib1.inc”
qreg q[13];
```

```
OPENQASM 2.0;
include “qelib1.inc”
qreg q[13];
h q[0];
h q[1];
...
```

Fig. 2: KetGPT Workflow: Firstly, a given text prompt is tokenized. These tokens are fed into the KetGPT model, which was trained with quantum circuits from an existing quantum circuit database. KetGPT then generates text to continue the given prompt, yielding a synthetic circuit. A separate transformer classifier model, trained to distinguish real from random quantum circuits, tests if the generated circuit is realistic. If the test is positive, it can be used to augment the quantum circuit database.

The Python code to construct this architecture is openly accessible through the GPT-2 implementation in the Hugging Face “Transformer” python library [44,21].

As discussed in Sec. 2.1, we employ a *tokenization* approach to transform the dataset text into tokens. The original implementation of GPT-2 relies on a form of tokenization known as Byte Pair Encoding (BPE). To comprehend this method intuitively, it dissects text into components (e.g., ‘training’ into ‘train’ and ‘ing’), facilitating a better grasp of the full word’s meaning. However, a drawback is that it may allow the generation of QASM code that is not syntactically correct, such as the potential generation of the line “hh q0q1;”. To address this, we modified the tokenization method for the generator to only permit syntactically correct QASM code as tokens. This modification was implemented by adjusting the GPT2Tokenizer class. By compiling a list of all valid QASM statements in the dataset and using it as our vocabulary, we ensure that any token generated by the model will be a valid QASM statement. The generator model workflow consists of the following four parts:

Preparation: The process of generating tokens using the generator model unfolds as follows: i) A list is compiled containing the qubit count for every circuit in the dataset, along with another list containing the number of gates for each circuit; ii) From these lists, a qubit count and a number of gates are randomly selected, establishing the parameters for the QASM file to be generated; and finally, iii) With these parameters in hand, any invalid QASM statement related to the selected qubit count is filtered out. For instance, if the chosen qubit count is 5, all gates involving qubit 13 are disregarded. This is achieved by preventing the generator model from producing these tokens.

Model input: The model will receive input as the following:

```
OPENQASM 2.0;
include "qelib1.inc"
qreg q[{}];
```

where {} will contain the chosen qubit count. This is the way all the QASM files in our dataset start, and it gives us an opportunity to control the qubit count in a simple manner.

Generation scheme: Whenever a new probability distribution over the tokens is generated, the top-k strategy [10] is employed, where the $k = 5$ most probable tokens are identified. From this subset, a new token is selected based on the renormalized probability distribution over these five tokens (the renormalization ensures that all probabilities add up to one). This approach introduces additional randomness into the QASM file generation process while maintaining the realism of the generated tokens, as the five most probable tokens are typically viable candidates. Furthermore, it is specified that sequences of 15 tokens should not repeat within the file. While this constraint may not align perfectly with QASM code generation, in which algorithms often contain repetitive sequences, it serves to prevent instances where the transformer model becomes stuck in a loop, repeatedly predicting the same sequence. The top-k generation process iterates until the desired number of gates is reached.

Post-processing: Finally, to guarantee the validity of all generated files, all quantum and classical registers utilized in the generated file are instantiated at the beginning of the QASM file. This ensures every file, including its header, is syntactically correct.

3.3 Verification Method: KetGPT Classifier

Once the generator produces the QASM files, the next step is to assess their authenticity. To determine whether the generated QASM files exhibit a "realistic" quality, we employed a binary classifier. This classifier's task is to distinguish whether a generated QASM file bears a closer resemblance to files from our algorithm-based circuit dataset or aligns more with a randomly generated QASM file [5].

The classifier adopts an encoder-only transformer model, specifically the architecture of the DistilBERT model [38], leveraging the implementation from the Huggingface transformers library [21]. This model is a smaller version of the highly influential encoder-only BERT model [9] and is chosen for quicker training and inference.

Unlike the generator, which required a customized tokenization method to ensure the generation of valid QASM code, the classifier employs the *tokenization* method used to train the original DistilBERT model, known as WordPiece [45]. This method, similar to the BPE tokenizer briefly mentioned in Sec. 3.2, breaks down words into sub-words. It is important to note that the choice of how these sub-words are determined distinguishes WordPiece from BPE, but this is not pertinent to this work. To adapt the QASM sequences for the classifier, the tokenization truncates them after 512 tokens. Since these tokens represent sub-words instead of complete QASM lines, the 512-token limit corresponds to approximately 50 lines of QASM code, dependent on the sequence. This adjustment ensures compatibility with the maximum input size of the classifier model used. While this approach has the drawback of only considering the initial portion of the QASM file in determining its authenticity, it offers the advantage of expedited training and inference, necessitating a less technically intricate model. Moreover, the initial segment of a QASM file typically provides sufficient cues to discern its nature as random or structured.

During the *training* phase of the classifier, a dataset is prepared in which all real quantum circuits are assigned the label '0' (total of 1112 QASM files). Correspondingly, an equal number of QASM files are randomly generated, comprising gates randomly selected from a list of all unique QASM statements in the dataset, and labeled '1'. To ensure fairness in the classification process, akin to the methodology employed for generating KetGPT QASM files, the randomly generated QASM files are structured to encompass the same distribution of qubit counts and number of gates as the original dataset. Subsequently, the model is trained on the labeled dataset, and upon completion of training, the trained model is employed to predict whether the KetGPT-generated circuits are classified as '0' or '1', indicating their proximity to genuine algorithms or random circuits, respectively.

3.4 Implementation Details

Our experiments were conducted using a Jupyter notebook [23] executed on the *Google Colab* environment [13]. This Notebook is provided in Section 6. The Google Colab GPU has 16Gb of GDDR6 memory, 320 Turing tensor cores and 2560 CUDA cores. At the time of writing, Google Colab uses Python version 3.10.12. Relevant packages for the code used to obtain the results of this work are the transformers [44] (version 4.34.0) and datasets [25] (version 2.14.5) libraries from Huggingface, PyTorch [32] (version 2.0.1+cu118) and NumPy [15] (version 1.23.5).

Tab. 1 contains the parameters that define the structure of our generator model. Default values correspond to those used in the original GPT-2 implementation [35]. The training settings are specified in Tab. 2. On the other hand, Tab. 3 specifies the settings that were used to define the *classifier model*. The training settings for the classifier model are in Tab. 4. All the parameters' detailed definitions can be found in [20].

Table 1: Generator model settings

Name	**Value**
n_embd	768 (default)
n_layer	3
n_head	4
n_positions	1024 (default)
vocab_size	48291

Table 2: Generator training settings

Name	**Value**
Epochs	5
Learning Rate	5e-5 (default)
Batch Size	4
Optimiser	AdamW (default)
Loss function	Cross-entropy (default)

Table 3: Classifier model settings

Name	**Value**
n_embd	768 (default)
n_layer	6 (default)
n_head	12 (default)
n_positions	512 (default)
vocab_size	30522

Table 4: Classifier training settings

Name	**Value**
Epochs	3
Learning Rate	5e-5 (default)
Batch Size	4
Optimiser	AdamW (default)
Loss function	Cross-entropy (default)

It is worth noting that KetGPT training time was 240 seconds, and generating 1000 QASM files took 8818 seconds (147 minutes), or 8.8 seconds per generated file on average. However, the QASM files are of varying size (as explained in Sec. 3.1), and the amount of time needed to generate one file is dependent on its size, so this number should be taken as a rough estimate.

4 Results and Discussion

In this section, we unveil outcomes of this work by showing the results of the three verification steps: manual inspection and Qiskit execution, transformer-based classifier and structural analysis of the circuits. Note that the usage of the term 'realistic' or 'real' when describing the circuits generated by KetGPT is not meant to be interpreted as describing circuits that implement useful quantum algorithms. The circuits might describe some undiscovered quantum algorithms, but it is nearly impossible to reverse engineer an explainable description.

4.1 Manual Inspection

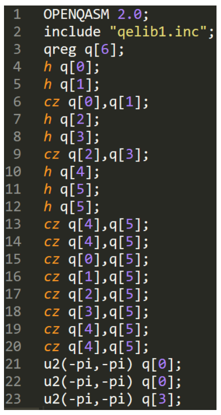
First, we manually examine the QASM lines of a circuit produced by KetGPT. We juxtapose this with the initial lines of both a genuine and a completely random circuit to establish a comparative analysis. One can observe some patterns shown in the files of Fig. 3: The lines within the KetGPT file and the real file exhibit structured patterns, such as the repetition of Hadamard and 2-qubit gates (CX and CZ), whereas the fully random circuit lacks such repetitive sequences. Additionally, it is noteworthy that the order in which the Hadamard gates are applied in the KetGPT and the real circuit follows an ascending order based on qubit numbers, whereas in the fully random circuit, as expected, there is no logical order of operations. Importantly, the random circuit includes invalid statements, such as operations on nodes that were never defined (e.g., an operation on node 4 is instructed, but node 4 was never defined). However, this error is also occasionally present in files generated by KetGPT, albeit seemingly less frequently. The fact that it is not specifically forbidden for KetGPT to generate invalid statements, but it still generates such statements considerably less often than random files, can also be seen as a realistic feature of KetGPT-generated data. Note that we also ran all our circuits within the Qiskit framework [3] where 96% of the circuits passed the compilation process successfully.

Based on the provided examples and the illustration in Fig. 3, a visual examination strongly indicates that KetGPT-generated circuits exhibit characteristics reminiscent of real quantum circuits. This observation underscores the promise of employing transformers to generate quantum circuit data.

4.2 Classifier-Based Evaluation

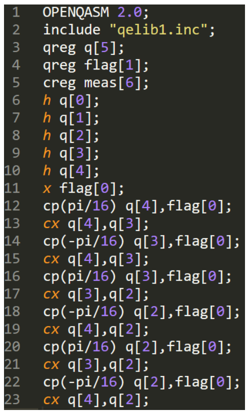
As a second measure of verification, we developed and trained a classifier model to determine whether KetGPT circuits resemble more real algorithm-based or random quantum circuits. As input, we created a dataset with the same amount of real and random circuits (1112 each) and used 85% of the data for training and 15% for testing the classifier.

To assess the model's performance, a confusion matrix is employed to ascertain the alignment between the model's predictions and the actual labels of the data. The



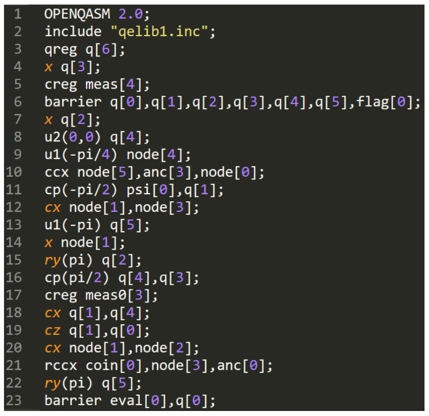
```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[6];
h q[0];
h q[1];
cz q[0],q[1];
h q[2];
h q[3];
cz q[2],q[3];
h q[4];
h q[5];
h q[5];
cz q[4],q[5];
cz q[4],q[5];
cz q[0],q[5];
cz q[1],q[5];
cz q[2],q[5];
cz q[3],q[5];
cz q[4],q[5];
cz q[4],q[5];
u2(-pi,-pi) q[0];
u2(-pi,-pi) q[0];
u2(-pi,-pi) q[3];
```

(a) KetGPT



```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
qreg flag[1];
creg meas[6];
h q[0];
h q[1];
h q[2];
h q[3];
h q[4];
x flag[0];
cp(pi/16) q[4],flag[0];
cx q[4],q[3];
cp(-pi/16) q[3],flag[0];
cx q[4],q[3];
cp(pi/16) q[3],flag[0];
cx q[3],q[2];
cp(-pi/16) q[2],flag[0];
cx q[4],q[2];
cp(pi/16) q[2],flag[0];
cx q[3],q[2];
cp(-pi/16) q[2],flag[0];
cx q[4],q[2];
```

(b) Real



```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[6];
x q[3];
creg meas[4];
barrier q[0],q[1],q[2],q[3],q[4],q[5],flag[0];
x q[2];
u2(0,0) q[4];
u1(-pi/4) node[4];
ccx node[5],anc[3],node[0];
cp(-pi/2) psi[0],q[1];
cx node[1],node[3];
u1(-pi) q[5];
x node[1];
ry(pi) q[2];
cp(pi/2) q[4],q[3];
creg meas0[3];
cx q[1],q[4];
cz q[1],q[0];
cx node[1],node[2];
rccx coin[0],node[3],anc[0];
ry(pi) q[5];
barrier eval[0],q[0];
```

(c) Random

Fig. 3: Side-by-side comparison between the lines of a 6 qubit QASM file generated by KetGPT (a), algorithm-based circuit (b) and a random circuit(c).

corresponding confusion matrix for this evaluation is depicted in Fig. 4. A total of 328 out of 334 test dataset values are predicted correctly, which means that the classifier model achieved an accuracy of 98.2%.

Subsequently, the classifier was tasked to classify 1000 KetGPT QASM files as either more similar to its training dataset (real algorithm-based) or to completely random qauantum circuits. Among the 1000 circuits evaluated, 999 were classified as authentic, indicating a classification accuracy of 99.9%.

It is difficult to evaluate the reliability of the model. The high accuracy could potentially be explained by the fact that the test dataset consists of a random subset of the total data. It is possible, for instance, that the Deutsch-Jozsa algorithm on 6 qubits is part of the training dataset, and Deutsch-Jozsa on 5 qubits is in the test dataset. The similarity between the training and testing data may influence the accuracy metric calculation. Nonetheless, using different instances of the same algorithms for the datasets was inevitable due to the limited availability of diverse algorithms. The random QASM files in the test set, however, are not similar to the random files in the training dataset, and are still predicted correctly every time.

Taking all of these considerations into account, including the classifier's accuracy when evaluated, it appears that the classifier is capable of discerning realistic features within the data. However, determining whether this proficiency results from the model overfitting to specific features of QASM files or genuinely learning relevant aspects of realistic circuits presents a challenge.

4.3 Analysis Based on Circuit Structure

Another approach to quantifying and validating KetGPT involves extracting structural parameters from circuits. Within this approach, a circuit is transformed into interaction and gate dependency graphs [5] and then analyzed based on quantum compilation-related, graph theory-based (e.g., degree of nodes) parameters. Following this methodology, we extracted the suggested 23 metrics [4] from our KetGPT circuits in order to compare them with existing circuit dataset. For comparison, we followed

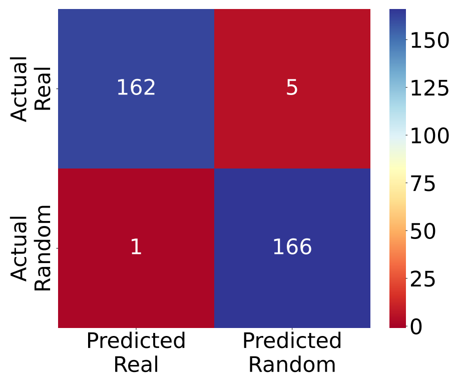


Fig. 4: Classifier performance on a test dataset illustrated by a confusion matrix. Diagonal values of the matrix are correctly predicted: only 5 QASM files that are actually "Real" are predicted as being "Random", and 1 QASM file that is 'Random' is predicted as being 'Real'.

another method suggested in [5] and clustered the circuits (KetGPT and qbench [5] circuits) based on the extracted parameters to discover groups of ultimately structurally similar circuits. The benchmark set *qbench* consists of real algorithm-based circuits, random circuits, and QUEKO circuits (synthetic circuits with predefined depth and gate count)[41], so by doing clustering, we could see where KetGPT would belong within these groups, or if it would form its own. Notably, we refrained from utilizing this benchmark set for creating KetGPT circuits, ensuring an unbiased evaluation.

Clustering is done in a two-level manner: first based on size and then sub-clusters based on the structure of the quantum circuits, resulting in a final tally of 18 clusters. For clarity, we consolidated clusters sharing identical circuit structures (in terms of circuit types) into one and illustrated the distribution in Fig. 5. The depicted clustering reveals that KetGPT circuits consistently align with real circuits and never with completely random ones. Additionally, a smaller portion of QUEKO circuits exhibit a similar association with both KetGPT and real circuits. Given that QUEKO circuits aim to mimic realistic behaviors more closely than classical random circuits [41], this observation is logical. Fig. 5 also suggests how much KetGPT contributes to having more realistic circuits in the whole set (green segments of the inner circle).

5 Conclusion and Outlook

The scarcity of quantum circuits 'useful' for benchmarking, stemming from limitations in existing datasets, poses a significant challenge to the progress of quantum compiler and hardware development. To address this gap, our research introduces KetGPT, a tool that utilizes the Transformer machine learning architecture to generate synthetic circuits resembling real-world quantum algorithms. We verified our resulting circuits three-fold by: 1) Running the circuits with Qiskit framework and manual inspection, we achieved a 96% success rate (without error or warning); 2) Implementing and training a transformer-based classifier for distinguishing between 'real' and random algorithms which classified KetGPT circuits as real in 99% of the cases; and 3) Characterizing

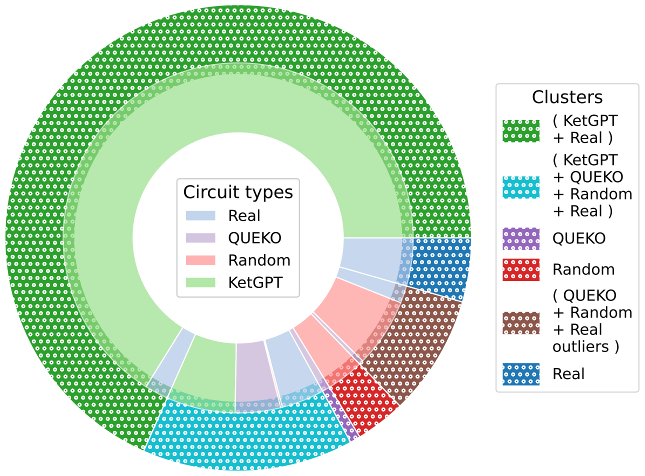


Fig. 5: The distribution of clusters obtained through structural parameters analysis is depicted. Each segment in the outer circle represents clusters characterized by the same types of circuits (e.g., the dark green segment encompasses all clusters that consist of KetGPT and real circuits). The inner circles display the quantity of each circuit type within the respective outer circle segment.

the generated circuits by extracting structure-based properties and clustering them together with another dataset containing real and random circuits. The analysis revealed that all our circuits closely resembled the structure of algorithm-based ones, and showcased the expansion of the dataset. In conclusion, this three-step, extensive verification shows that KetGPT can augment realistic and executable quantum circuit dataset(s).

Our future steps in expanding and improving KetGPT include: i) Exploring alternative generation schemes, such as top-p [19], beam search [14], or contrastive search [39], to compare their effectiveness in generating QASM files or, development of a generation scheme tailored specifically for QASM file generation; ii) Reconsidering the representation of QASM statements as discrete tokens: Introducing an arbitrary gate token to accommodate QASM files with arbitrary angles, using a transformer trained for this purpose in post-processing; and iii) Modifying the tokenization scheme by separating gates and target qubits into distinct tokens (e.g., treating 'Hadamard gate' and 'on qubit 1' as separate tokens) and ensuring that the adjusted scheme generates only valid QASM expressions and exploring its scalability for higher qubit counts.

In summary, we are confident that KetGPT holds the promise to not only significantly influence the benchmarking of quantum systems, but also to serve as a valuable input for data-intensive, AI-based solutions in the development of innovative quantum compilers and systems.

6 Software Availability

The code that was used for this work is provided as a Jupyter notebook [23], which was executed in the Google Colab environment [13], available at:
https://colab.research.google.com/drive/1dbtJX6q8sED4yrb1I09KUuXWYH0AVN8r.

The data that was used for this work, comprising of the training dataset, and a KetGPT folder that contains: the pre-trained KetGPT model, the KetGPT tokenizer, the pre-trained classifier model, all KetGPT generated circuits and all random circuits, is available at: https://www.kaggle.com/datasets/boranapak/ketgpt-data.

7 Acknowledgments

MB and SF acknowledge funding from Intel Corporation. AS acknowledges funding from the Dutch Research Council (NWO).

References

1. Acampora, G., Schiattarella, R.: Deep neural networks for quantum circuit mapping. Neural Computing and Applications **33**(20), 13723–13743 (2021)
2. Agostinelli, A., et al.: Musiclm: Generating music from text. arXiv preprint arXiv:2301.11325 (2023)
3. Anis, M.S., et al.: Qiskit: An open-source framework for quantum computing (2021). https://doi.org/10.5281/zenodo.2573505
4. Bandic, M., et al.: Qauntum benchmarks structural analysis for improvement of quantum circuit mapping for single- and multi-core quantum computation (2024), (work in progress)
5. Bandic, M., Almudever, C.G., Feld, S.: Interaction graph-based characterization of quantum benchmarks for improving quantum circuit mapping techniques. Quantum Machine Intelligence **5**(2) (2023)
6. Bridle, J.: Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. Advances in neural information processing systems **2** (1989)
7. Cross, A.W., Bishop, L.S., Sheldon, S., Nation, P.D., Gambetta, J.M.: Validating quantum computers using randomized model circuits. Physical Review A **100**(3) (2019)
8. Cross, A.W., Bishop, L.S., Smolin, J.A., Gambetta, J.M.: Open quantum assembly language. arXiv preprint arXiv:1707.03429 (2017)
9. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (2019)
10. Fan, A., Lewis, M., Dauphin, Y.: Hierarchical neural story generation. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 889–898. Association for Computational Linguistics, Melbourne, Australia (2018)
11. Fösel, T., Niu, M.Y., Marquardt, F., Li, L.: Quantum circuit optimization with deep reinforcement learning. arXiv preprint arXiv:2103.07585 (2021)
12. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT press (2016)
13. Google, LLC: Google colaboratory (2023), https://colab.research.google.com
14. Graves, A.: Sequence transduction with recurrent neural networks. arXiv preprint arXiv:1211.3711 (2012)
15. Harris, C.R., et al.: Array programming with NumPy. Nature **585**, 357–362 (2020)
16. Harshvardhan, G., Gourisaria, M.K., Pandey, M., Rautaray, S.S.: A comprehensive survey and analysis of generative models in machine learning. Computer Science Review **38**, 100285 (2020)
17. Hendrycks, D., Gimpel, K.: Gaussian error linear units (gelus). arXiv preprint arXiv:1606.08415 (2016)
18. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
19. Holtzman, A., Buys, J., Du, L., Forbes, M., Choi, Y.: The curious case of neural text degeneration. arXiv preprint arXiv:1904.09751 (2019)

20. HuggingFaceInc.: Openai gpt2. https://huggingface.co/transformers/v3.5.1/model_doc/gpt2.html (2020)
21. HuggingFaceInc.: Transformers: State-of-the-art natural language processing. https://github.com/huggingface/transformers (2021)
22. Jordan, S.: Quantum algorithm zoo, https://quantumalgorithmzoo.org, accessed: 25-09-2023
23. Kluyver, T., et al.: Jupyter notebooks-a publishing format for reproducible computational workflows. Elpub **2016**, 87–90 (2016)
24. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
25. Lhoest, Q., et al.: Datasets: A community library for natural language processing. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. pp. 175–184. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic (Nov 2021)
26. Li, A., Stein, S., Krishnamoorthy, S., Ang, J.: Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation. ACM Transactions on Quantum Computing **4**(2), 1–26 (2023)
27. van der Linde, S., de Kok, W., Bontekoe, T., Feld, S.: qgym: A gym for training and benchmarking rl-based quantum compilation. arXiv preprint arXiv:2308.02536 (2023)
28. Montanaro, A.: Quantum algorithms: an overview. npj Quantum Information **2**(1), 1–8 (2016)
29. Nijkamp, E., et al.: Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474 (2022)
30. OpenAI: Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
31. Overwater, R.W., Babaie, M., Sebastiano, F.: Neural-network decoders for quantum error correction using surface codes: A space exploration of the hardware cost-performance tradeoffs. IEEE Transactions on Quantum Engineering **3**, 1–19 (2022)
32. Paszke, A., et al.: Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019)
33. Quetschlich, N., Burgholzer, L., Wille, R.: Mqt bench: Benchmarking software and design automation tools for quantum computing. Quantum **7**, 1062 (2023), mQTbench is available at https://www.cda.cit.tum.de/mqtbench/
34. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al.: Improving language understanding by generative pre-training (2018)
35. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)
36. Robbins, H., Monro, S.: A stochastic approximation method. The annals of mathematical statistics pp. 400–407 (1951)
37. Rumelhart, D.E., Hinton, G.E., Williams, R.J., et al.: Learning internal representations by error propagation (1985)
38. Sanh, V., Debut, L., Chaumond, J., Wolf, T.: Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. ArXiv **abs/1910.01108** (2019)
39. Su, Y., Lan, T., Wang, Y., Yogatama, D., Kong, L., Collier, N.: A contrastive framework for neural text generation. Advances in Neural Information Processing Systems **35**, 21548–21561 (2022)
40. Svyatkovskiy, A., Deng, S.K., Fu, S., Sundaresan, N.: Intellicode compose: Code generation using transformer. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1433–1443 (2020)
41. UCLA: Queko benchmark. https://github.com/UCLA-VAST/QUEKO-benchmark (2020)
42. Vaswani, A., et al.: Attention is all you need. Advances in neural information processing systems **30** (2017)
43. Wille, R., Große, D., Teuber, L., Dueck, G.W., Drechsler, R.: Revlib: An online resource for reversible functions and reversible circuits. In: 38th International Symposium on Multiple Valued Logic (ismvl 2008). pp. 220–225. IEEE (2008)
44. Wolf, T., et al.: Transformers: State-of-the-Art Natural Language Processing. pp. 38–45. Association for Computational Linguistics (Oct 2020)
45. Wu, Y., et al.: Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv preprint arXiv:1609.08144 (2016)