QPulseLib: Accelerating the Pulse Generation of Quantum Circuit with Reusable Patterns

Wuwei Tian
Zhejiang University
Hangzhou, China
sonder@zju.edu.cn

Xinghui Jia
Zhejiang University
Hangzhou, China
jxhhhh@zju.edu.cn

Siwei Tan
Zhejiang University
Hangzhou, China
siweitan@zju.edu.cn

Zixuan Song
ZJU-Hangzhou Global Scientific and Technological Innovation Center
Hangzhou, China
21836036@zju.edu.cn

Liqiang Lu*
Zhejiang University
Hangzhou, China
liqianglu@zju.edu.cn

Jianwei Yin*
Zhejiang University
Hangzhou, China
zjuyjw@cs.zju.edu.cn

2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323711

***Abstract*—Quantum circuit serves as a popular programming model that describes the computation using a set of quantum gates, which requires generating a sequence of pulses that collect the operation of each gate for superconducting quantum devices. However, existing quantum synthesis frameworks, like IBM OpenPulse [1], involve massive redundant computation during pulse generation, suffering from a high computational cost when handling large-scale circuits. In this paper, we propose QPulseLib, a novel library with reusable pulses that can directly provide the pulse of a circuit block. To establish this library, we transform the circuit and apply convolutional operators to extract reusable patterns and pre-calculate their resultant pulses. Then, we develop a matching algorithm to identify such patterns shared by the target circuit. Experiments show that QPulseLib achieves 158.46× and 16.03× speedup for pulse generation, compared to OpenPulse and AccQOC [2].**

***Index Terms*—quantum computing; pulse generation; pre-synthesis;**

I. Introduction

Quantum computing has been demonstrated with the ability to offer polynomial or even exponential speedup in certain areas, such as chemistry simulation [3], database search [4], and combinatorial optimization [5]. Quantum circuit is a widely-used programming model that describes the computation by quantum gates [6]. When deploying these applications to superconducting quantum hardware, there require specific pulses to implement the operation of quantum gates and control the evolution of qubits. Specifically, the transformation from quantum circuits to pulses refers to the synthesis process that generates a discrete pulse for each gate, with specific shapes and timing dependency.

However, the synthesis process suffers from a high time cost, especially for large-scale quantum circuits, due to the overwhelming computational complexity for calculating the parameters of pulses. For example, it takes 2,190.67 seconds to generate the pulse sequence for a 300-qubit quantum multiplier circuit using OpenPulse [1], which involves 191,903 single-qubit gates and 121,095 two-qubit gates. Figure 1 provides the breakdown of the synthesis time on the circuits of Hamiltonian Simulation [7] and QKNN [8], suggesting that the most time is spent on the computation of pulses for quantum gates. Such time-consuming synthesis fundamentally results from gate-by-gate pulse generation, which exhibits massive redundant computation that repeatedly calculates the pulse for the same circuit pattern. Taking the 300-qubit quantum multiplier circuit as an example, we observe that 65% computation of OpenPulse is used for the same circuit pattern.

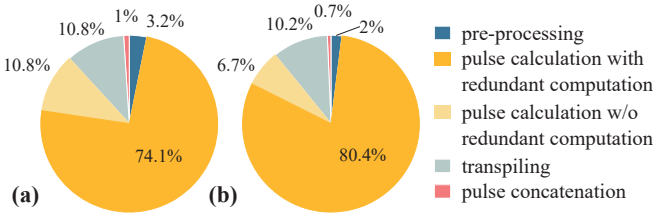


Fig. 1. Time breakdown of circuit synthesis for (a) Hamiltonian simulation [7] and (b) QKNN [8] at 300 qubits with OpenPulse [1], indicating pulse calculation accounts for 84.9% and 87.1% synthesis time, respectively. Among them, 74.1% and 80.4% of the time is redundant.

In this paper, we propose QPulseLib, a pulse library that consists of reusable patterns to reduce the overall synthesis time for pulse generation. To establish this library, we first parameterize the circuit into a matrix representation that records the gate position and the gate type in each element. Then, a convolutional operator is applied to this matrix, which extracts the features of a sub-circuit block into a single value. By setting an elaborate convolutional kernel, the blocks that share the same value indicate the same circuit feature, identified as the reusable pattern. The library is built upon the pulses that are pre-calculated from the reusable patterns of various circuit benchmarks. For a target circuit, we develop a greedy-based algorithm to match the patterns in the pulse library. In other

*Corresponding Authors: Jianwei Yin, Liqiang Lu

words, we can directly obtain the pulse of the matched pattern from QPulseLib without calculating the pulse gate by gate. The contributions of this paper are summarized as follows:

- We propose *QPulseLib* to accelerate the pulse generation of quantum circuits, which provides significant acceleration in the pulse generation, compared to the current state-of-the-art method [2]. Moreover, our approach exhibits high scalability that can support the synthesis with hundreds of qubits.
- We propose a novel pulse library that covers the reusable patterns derived from various circuit benchmarks, which leverages a convolution-based method to identify the sub-circuit that is reused by multiple circuit benchmarks.
- We propose a greedy-based matching algorithm to allocate the block that shares the same sub-circuit in the library, to enable the end-to-end acceleration.

Experiments show that, compared to OpenPulse [1] and AccQOC [2], QPulseLib achieves 158.46× and 16.03× speedup for pulse generation time, 10.91× and 1.21× speedup for end-to-end synthesis time, respectively. Our pulse library exhibits up to 82.3% reuse rate on 15 circuit benchmarks.

II. Background

A. Circuit Synthesis

A quantum program is commonly represented as a quantum circuit and then synthesized into pulse sequences for hardware control. For superconducting quantum devices, the pulse sequences are sent via digital-to-analog converters (DACs) to manipulate the quantum processor. There are mainly two steps in circuit synthesis: circuit transpilation and pulse generation.

Quantum circuit. The quantum circuit is regarded as the representation of high-level quantum programs for the implementation on real-world quantum devices. The primary components of a quantum circuit are qubits and sequential quantum gates [9], [10]. Qubits store information about the circuit, while the quantum gates indicate the evolution of this information. We define *layer* as the fundamental unit of the qubit timeline. Within each layer, gates for each operated qubit are executed in parallel.

Circuit transpilation. The process of *transpilation* [11] involves converting the original quantum circuit into an executable form while incorporating various optimizations, such as circuit mapping optimization [12] and noise optimization [13], [14]. Advanced quantum programs may consist of diverse gate operations. However, not all these gates can be implemented on quantum hardware [15], [16]. To address this challenge, these gates must be decomposed into basis gates that can be executed locally. Through mapping processes and gate transformation, the optimized circuit becomes executable on specific quantum devices.

Pulse generation. Generally, control pulses for superconducting quantum devices are basically generated and scheduled following the timeline of the circuit. For each gate, the related functions are called to obtain its pulses, which are then concatenated in chronological order. Finally, many individual pulse segments are scheduled in time and concatenated together to form the final sequence.

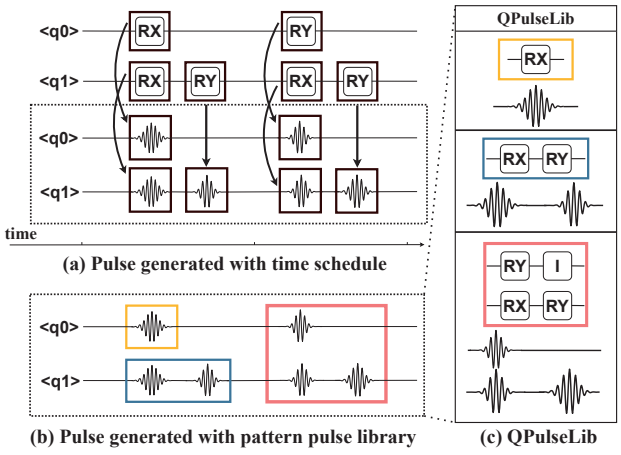


Fig. 2. (a) Pulse generated by time schedule. Pulses are calculated following time schedule. (b) Pulse generated by QPulseLib. Pulse sequence is generated using pre-generated pulses after circuit matching. (c) QPulseLib.

B. Specific Pulse Generation Process

Superconducting quantum qubits undergo state evolution by applying control pulses [17], [18]. Changes of states and phases for qubits in Hilbert space are made through accurate control of continuous pulses, labeled as gates in the quantum circuit. A *pulse sequence* is defined as the collection of pulses for the circuit. Pulse sequence consists of a time series with amplitudes based on a fixed sampling time step [1]. A cycle-time, referred to as dt, serves as a fundamental time unit for superconducting devices. It is typically determined by the sample rate of the coprocessor's pulse generators, such as the field programmable gate array (FPGA). For example, the control pulse data for a π pulse [19] is calculated and discretized as $[s_0, s_1, \cdots, s_n]$ for each time unit dt on n sampling points. The amplitude A_n of pulse data in each time unit is defined as follows:

$$A_n = Re[e^{i2\pi ndt} s_n]$$

where *Re* refers to the real part of a complex value. During the pulse generation process, various calculation functions are called, including Gaussian, rectangular, and sinusoidal pulses [1]. Discrete pulse signals are then converted by DAC and sent to the quantum processor for effective hardware control [20].

The pulses are calculated and concatenated sequentially in OpenPulse [1], as shown in Figure 2 (a). However, with the help of QPulseLib illustrated in Figure 2 (c), pulse can be retrieved and concatenated with the pre-generated pulses from the pulse library after circuit matching, as illustrated in Figure 2 (b).

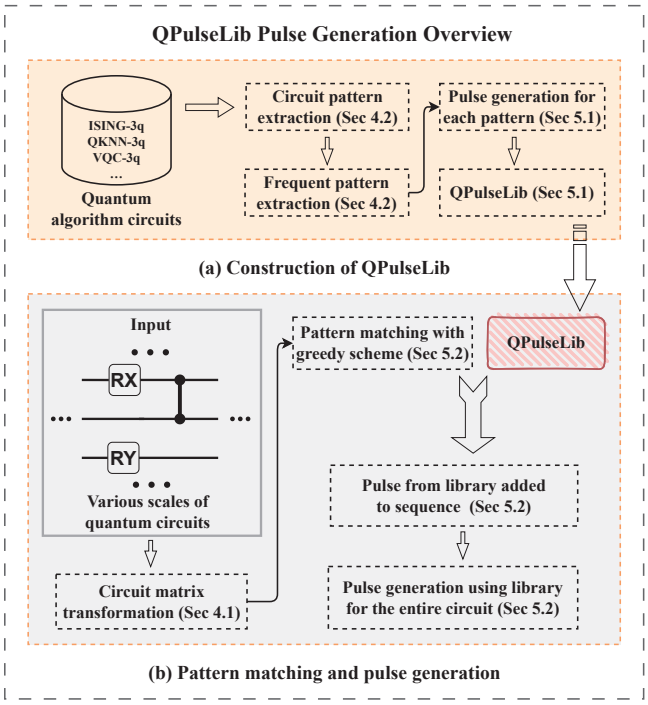


Fig. 3. Overview of QPulseLib, containing (a) construction of QPulseLib, and (b) pattern matching and pulse generation.

III. QPulseLib Overview

The insight of designing *QPulseLib* lies in the fact that although pulse generation for executing quantum programs on hardware is inevitable, pulses can be reused through the pulse library. The workflow of QPulseLib is shown in Figure 3. Figure 3 (a) illustrates the initial step of constructing the QPulseLib with a set of small-scale quantum circuits, such as the ISING model [21], QKNN model [8], and VQC model [22]. For instance, the *ISING-3q* indicates a quantum circuit of ISING model with 3 qubits. The construction of the QPulseLib will be presented in Section V-A. After circuit transformation, the quantum circuit is represented as circuit matrix for circuit matching, which will be described in Section IV-A. The library consists of frequent reusable sub-circuits extracted from these circuits and their corresponding pulses.

QPulseLib leverages a pulse library to reuse pulses and minimize synthesis time for large-scale circuits, even those comprising over a hundred qubits sharing patterns with the library. Specifically, the process involves initiating an empty pulse sequence, and subsequently collecting the pulses with matched patterns into the sequence, as detailed in Section V-B. Figure 3 (b) illustrates the initial step of transforming the target circuit into a circuit matrix (Section IV-A), where each sub-matrix captures the information about adjacent gates in the circuit, as explained in Section IV-B. QPulseLib then applies convolutional operators to the circuit matrix and compares the results to the pattern in the pulse library, introduced in Section V-A. For the matched pattern, the resultant pulse is concatenated to the pulse sequence. Through the complete matching of the circuit matrix, QPulseLib finally generates a pulse sequence for the entire circuit.

IV. Extracting Circuit Features

The overview in Section III suggests that the speedup of QPulseLib mainly results from identifying reusable pulses. To this end, we should extract the frequently-used patterns that occur in various quantum circuits and find out the same sub-circuit shared among them. In this section, we propose an efficient pattern extraction method to capture such patterns, which consists of matrix-based circuit transformation and convolution-based quantification.

A. Transforming Circuit to Matrix

Each element of the circuit matrix represents one gate, which stores the information about its operation and its parameters of a gate. For a circuit with N qubits and M layers, the matrix size is $N \times M$. For example, each element $e_{n,m}$ in the n^{th} row and m^{th} column of the matrix corresponds to a gate that operates on qubit timeline q_n in the m^{th} layer. Note that we only consider three basis gates after circuit transpilation: single-qubit gate RX, single-qubit gate RY, and two-qubit gate CZ. Specifically, there are three cases for the matrix assignment:

- If there is a single-qubit gate (RX, RY), its value is expressed as a complex indicating the gate types and the rotation of the gate. As current superconducting quantum hardware does not implement z-axis rotation [23], we adopt a complex value to record the gate rotations in the x-axis (RX gate) or y-axis (RY gate), using the real value or imaginary value. For example, the value of an RY gate, with a rotation angle of $\frac{\pi}{4}$, is $\frac{\pi}{4}i$.
- As there is only one type of two-qubit gate (CZ), its value records the position of the operated qubits. For example, for a two-qubit gate (CZ) in layer k, with the operated qubits on the n^{th} and n'^{th} timeline, its value is expressed as $\beta \times (n-n')$ and $\beta \times (n-n')i$. $\beta \times (n-n')$ is the value of the control qubit, while $\beta \times (n-n')i$ is the value of the target qubit. β is a constant determined by the voltage bias of CZ gates.
- For the element with no gate operation or just an identity gate (a wait cycle for a single-qubit gate), its value is set to 0.

In a word, the position of gates is encoded as the index of elements in the matrix. And the parameter of gates is encoded as the value. That is to say, a block from the circuit corresponds to a sub-matrix, as shown in Figure 4. This encoding scheme ensures to thoroughly transform the entire circuit into a quantitative manner.

B. Extraction using Convolutional Operator

To accelerate the generation of pulses, our goal is to identify the reusable blocks in the matrix representation. Therefore, we propose to utilize a convolution kernel, which acts like a sliding window in the circuits and performs dot-product in the window. For a kernel $K_{r,s}$ with size $r \times s$, the sliding

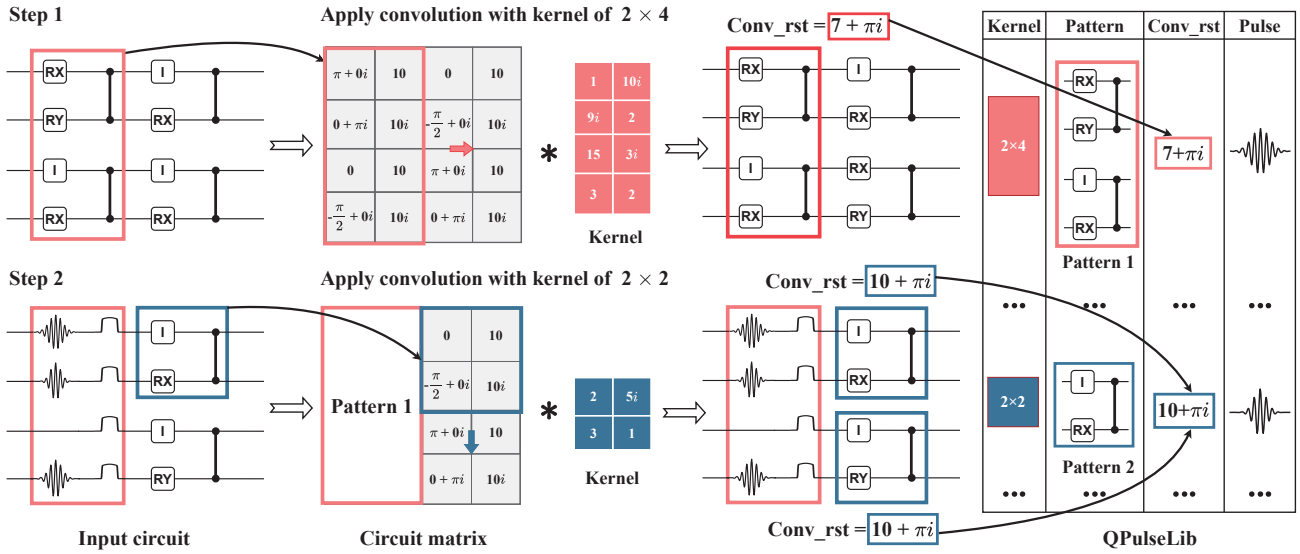


Fig. 4. Two-step process to conduct circuit matching with QPulseLib through convolution. In this example, we set β to 10.

window includes r qubits in s layers. The convolution result is denoted as $conv\_rst$. The kernel is designed to ensure that the sub-matrix containing the same elements always produces the same convolution output. The same $conv\_rst$ indicates the same sub-matrix occurs in the circuit. If we observe a $conv\_rst$ that matches an entry in the library, we then conduct a comparison between the matched pattern in the library and the block in the target circuit. Once the comparison results are also consistent, we can leverage the pre-calculated pulse in the library. For example, the sub-matrix in the red box in *Step 1* of Fig 4, is firstly applied with a kernel of 2×4, obtaining the $conv\_rst$ of $7 + \pi i$. Next, we find the same $conv\_rst$ with matched pattern in the library. Last, we fetch the pulse under this pattern. Similarly, in step 2 of Figure 4, the sub-matrix at the top-right of the input circuit, highlighted in the blue box, leads to the result of $10 + \pi i$, matching a pattern in the pulse library. After conducting the convolution operation for the rest parts in the matrix, we can generate the final pulse sequence for the entire circuit.

Once convolution is applied to each sub-matrix in the circuit, we can reformulate the input circuit as a collection of multiple convolutional results. The size of the convolution kernel determines the size of each sub-matrix. Aiming to precisely extract more underlying patterns, we try various kernels with different sizes. Considering that a larger kernel size potentially brings higher speedup but requires more computation time, we empirically choose the kernel with sizes of 3×3, 4×2, 2×4, 3×2, 2×3, 2×2, 1×2. To be specific, when performing convolution, we employ kernels in the order of their kernel size, where a small kernel allows a more fine-grained matching. As shown in Figure 4, we first use the kernel of 2×4 in *Step 1*, then we apply kernel of 2×2 in *Step 2*, avoiding repeat calculation on the left part of the circuit matrix. We do not apply any larger kernels due to the low matching rate, which will reduce the efficiency of pulse generation.

Kernel values are assigned with the purpose of distinguishing different patterns that feature different convolution results. This ensures that the same pattern consistently shares the same value, while different patterns produce distinct results. To enable this, kernels are initially assigned random values ranging from 1 to 10. Then, their values are optimized using

TABLE I
QUANTUM BENCHMARKS APPLIED IN OUR EXPERIMENTS

Algorithm	Description	Qubits	#1q	#2q
QFT	Quantum Fourier Transformation [24]	15	499	231
GHZ	Preparing Greenberger–Horne–Zeilinger State [25]	15	29	14
QFT_IN	Inverse Quantum Fourier Transformation [26]	15	509	246
BV	Bernstein-Varzirani Algorithm [27]	15	11	4
DJ	Deutsche-Jozsa Algorithm [28]	15	23	14
HS	Hamiltonian Simulation [7]	15	42	28
ISING	Linear Ising Model [21]	15	46	28
QKNN	Quantum K-nearest Neighbors Algorithm [8]	15	85	56
QSVM	Quantum Support Vector Machine [29]	15	63	56
QAOA	Quantum Approximate Optimization Algorithm [5]	15	144	66
VQC	Variational Quantum Classifier [22]	15	184	448
QEC	Quantum Error Correction Code [30]	15	33	18
MUL	Quantum Multiplier [31]	15	372	242
W-STATE	Preparing Quantum W_State [32]	15	51	28
SIMON	Quantum Simon Algorithm [33]	20	48	17

genetic algorithm (GA) [34] with a dataset of 15 benchmarks ranging from 6 to 10 qubits, as listed in Table I. In this table, #1q refers to the number of single-qubit gates, and #2q is for two-qubit gates. We first generate a set of random values for each kernel as candidates. GA algorithm is then applied to conduct a stochastic search, which iteratively adjusts the value and evaluates the candidate kernel. In each iteration, we update the kernel values through a combination of random selection and modifications, using the *crossover* and *mutation* steps in the GA algorithm. Once it exceeds the maximum number of iterations or there are no further updates within 3 iterations, the search stops. According to our test, such a constraint is enough to distinguish the patterns in the dataset circuit.

Different from the prior approach AccQOC [2] that relies on frequent sub-graph extraction, we employ convolutional operators to find reusable sub-matrices with pattern matching, which is a widely-used technique for identifying identical adjacent sub-matrices within a matrix [35]. Both methods exhibit a time complexity of $\mathcal{O}(n^2)$ determined by the number of circuit gates. While, the matrix multiplication involved in convolution can be accelerated using graphics processing units (GPUs) [36], thereby enhancing the efficiency of pulse generation.

V. Accelerating Pulse Generation

A. Pulse Library Construction

The QPulseLib is a static repository that is designed to store highly-reusable patterns along with their pulses. Besides, it contains the convolutional results together with patterns for circuit matching. To construct this library, we select 60 circuits comprising 2-5 qubits for 15 quantum algorithms listed in Table I. These circuits are first transformed using Qiskit [37] to comply with hardware constraints such as basis gates decomposition and quantum processor topology.

The number of patterns labeled in QPulseLib directly affects the efficiency of circuit matching. In other words, we need to balance the trade-off between library storage overhead and the number of extracted patterns. To tackle this, we set a threshold to decide the number of patterns in the library, which is formulated as the frequency of patterns occurring in the benchmarking circuits. Patterns below the threshold are filtered out, while the patterns with high reuse-frequency are included to build the QPulseLib.

We observe that both large-scale circuits and small-scale circuits share a number of same patterns when building QPulseLib. This is because, currently, quantum circuits are orchestrated from highly-specialized quantum algorithms, which usually feature structural sub-circuits. For example, the circuit of the QFT algorithm [24] contains iteratively-applied CZ gates from each qubit to other qubits, leading to repeated patterns when applying Qiskit to transform them into basic gates. Our experiments will provide a visualization of these structural patterns and detailed evaluation results.

Algorithm 1: Algorithm for Matching and Pulse Generation.

Input: Circuit: C , Convolution kernels: K, QPulseLib: L
Output: Pulse sequence: P

Initialize empty pulse sequence P;
M = Circuit matrix of C;
Sort K by kernel size;
foreach *kernel k in K* **do**
  feature map = convolution result of k on M;
  foreach *$conv\_rst$ in feature map* **do**
    sub-matrix = matrix in slide window with size of k in M corresponding to $conv\_rst$;
    if *$conv\_rst$ in L **and** sub-matrix in $L[conv\_rst]$* **then**
      pattern = $L[conv\_rst][pattern]$;
      sub-matrix matched with pattern and locked;
      Add pulse of $L[conv\_rst][pulse]$ to P;
    end
    Update M;
  end
end
Calculate pulses for gates not synthesized;

B. Matching Algorithm for Pulse Generation

QPulseLib takes the target quantum circuit as input and initiates the synthesis by transforming it into matrix representation. As mentioned before, we employ different convolution kernels to identify the reusable patterns, which may lead to the case that multiple patterns match the same sub-circuit. To avoid redundant calculation, we propose a matching algorithm by prioritizing the reuse opportunity of convolution kernels. As shown in Algorithm 1, we employ a greedy-based algorithm that always compares $conv\_rsts$ from larger kernels before proceeding to smaller ones. To be specific, we define P as the output pulse sequence of the target circuit. Convolution kernels are sorted depending on their size to ensure that the matching begins with the largest kernel. For each kernel, we use the convolutional operator on the circuit to obtain a set of output feature maps, and then search for the matched $conv\_rsts$ in QPulseLib. After successfully finding the matched sub-matrix with the current pattern, we collect the sub-matrix and terminate the search for the involved gates.

An illustrative example of the matching algorithm is presented in Figure 4. Given an input circuit, we consider two convolution kernels: 2×4 and 2×2. In *Step 1*, we apply the larger kernel of 2×4, obtaining a $conv\_rst$ of $7 + \pi i$ matched in QPulseLib. After a comparison between the sub-matrix and the existing pattern, the sub-matrix shown in the red box is labeled as searched, adding the pulse of this pattern to the pulse sequence. In *Step 2*, we continue to apply a smaller kernel 2×2 to perform convolution to the rest part of circuit matrix. For gates that are not covered by patterns, we dynamically calculate the pulse via traditional functions, until the complete pulse sequence for the circuit is generated and concatenated.

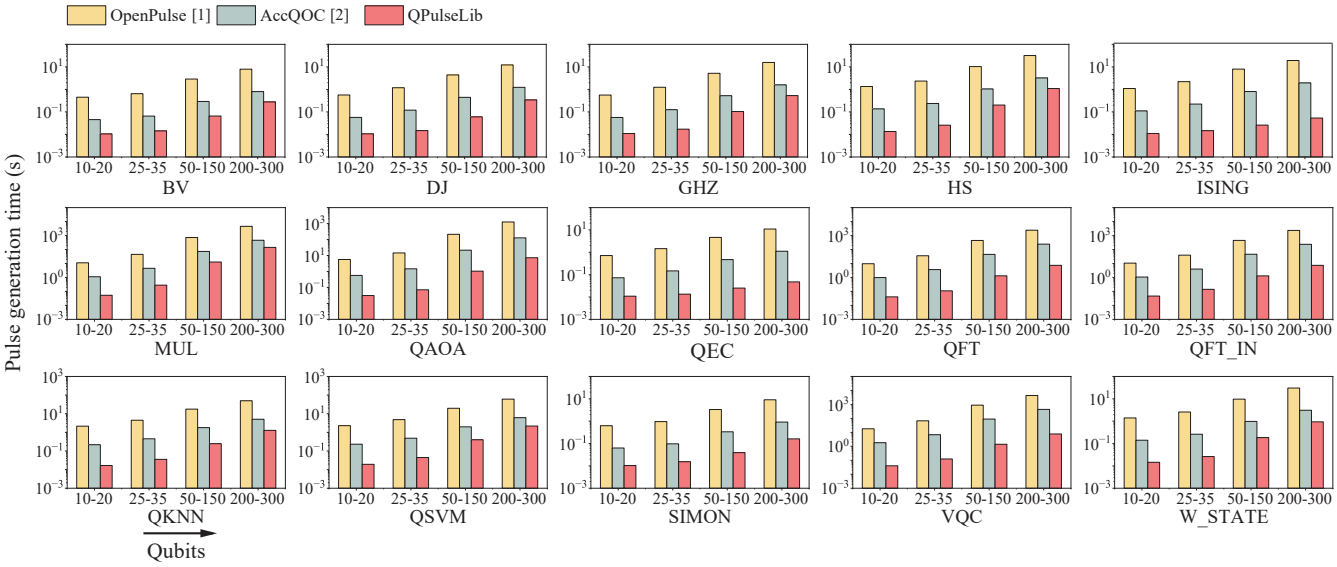


Fig. 5. Evaluation of pulse generation time in 15 algorithms. The x-axis is the number of qubits.

VI. QPulseLib Evaluation

A. Experimental Setup

Implementation. We implement QPulseLib with Python (3.8.10) and the `NumPy` package (1.23.1) [38]. We use the `transpiler` and `pulse` APIs in `Qiskit` package (0.44.0) [39] to ensure the synthesized circuits match the hardware constraint. We utilize a hash dictionary to store the extracted patterns and their corresponding control pulses, referring to QPulseLib.

Configuration. To balance the convolution time and the speedup, we use convolution kernel with sizes of 3×3, 4×2, 2×4, 3×2, 2×3, 2×2, 1×2. The threshold is set to 0.13 for pulse library construction, which is then evaluated in Section VI-C.

Simulated hardware. We simulate the pulse synthesis process on quantum devices with 10, 15, 20, 25, 30, 35, 50, 100, 150, 200, 250, and 300 qubits. The voltage bias β is set to 10 in the simulation. The topology of these processors is configured to be similar to the 79-qubit Rigetti quantum device [40]. Besides, the setting of other pulse parameters is also similar to the Rigetti quantum device, including the single-qubit gate time (50 ns), and the two-qubit gate time (150 ns) [41]. We set the sampling rate of the pulses to 2 GHz.

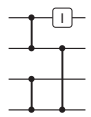
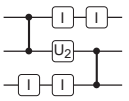
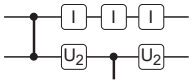
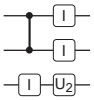
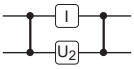
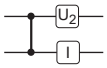
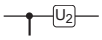
Baselines. We compare QPulseLib with OpenPulse [1] and AccQOC [2]. OpenPulse is the pulse generation tool of IBMQ. AccQOC is the recent state-of-the-art pulse optimal control method. We compare their pulse generation times on 15 algorithm circuits (Table I) with 10, 15, 20, 25, 30, 35, 50, 100, 150, 200, 250, and 300 qubits. These circuits are transpiled by the Qiskit transpiler [37] to transform the quantum circuit into basis quantum gates.

B. Speedup for Pulse Generation

Figure 5 presents the pulse generation time of OpenPulse [1], AccQOC [2], and QPulseLib on the 15 algorithms. The x-axis represents the number of qubits, while the y-axis represents the average pulse generation time, measured in seconds. We repeat the generation process of each circuit 10 times and calculate the average generation time. Overall, QPulseLib achieves significant speedup, with a speedup of 158.46× and 16.03× compared to OpenPulse [1] and AccQOC [2] in pulse generation time, and 10.91× and 1.21× speedup for end-to-end circuit synthesis time, respectively. This is attributed to the fact that our pulse library effectively reduces redundant computation by pulse reuse. In contrast, the pulse generation of OpenPulse involves massive redundant computation. As a result, it requires 571.22 seconds when synthesizing the QAOA algorithm [5] at 300 qubits, while QPulseLib spends 3.48 seconds (164.16× speedup). On the other hand, AccQOC utilizes a pulse table. However, it lacks an efficient pattern matching method. For example, for QFT algorithm [24] at 300 qubits, AccQOC requires 114.06 seconds for pattern matching, while QPulseLib spends only 3.44 seconds based on its convolution-based pattern extraction method.

As the number of qubits increases, we observe that the pulse generation time of QPulseLib grows slowly compared to OpenPulse [1] and AccQOC [2]. This is because QPulseLib can find more reusable patterns in large-scale quantum circuits. For example, OpenPulse and AccQOC show $\mathcal{O}(n^2)$ complexity in VQC [22], QFT [24] and QFT_IN [26] algorithms, while QPulseLib achieves close to linear time complexity. For 10-qubit and 20-qubit circuits, the speedups of QPulseLib are 133.38×, 13.50× compared to OpenPulse and AccQOC, respectively. For 300-qubit circuits, these speedup increases to 423.91× and 42.90×, respectively.

TABLE II
MOST FREQUENT REUSABLE PATTERNS IN 15 ALGORITHM CIRCUITS.

Kernel	4×2	3×3	2×4	3×2	2×3	2×2	1×2
Pattern							
Matrix	$\begin{bmatrix} CZ & I \\ CZ & CZ \\ CZ & I \\ CZ & CZ \end{bmatrix}$	$\begin{bmatrix} CZ & I & I \\ CZ & U_2(\pi,\frac{\pi}{2}) & CZ \\ I & I & CZ \end{bmatrix}$	$\begin{bmatrix} CZ & I & I & I \\ CZ & U_2(-\frac{\pi}{4},0) & CZ & U_2(\frac{\pi}{4},0) \end{bmatrix}$	$\begin{bmatrix} CZ & I \\ CZ & I \\ I & U_2(\pi,0) \end{bmatrix}$	$\begin{bmatrix} CZ & U_2(\pi,0) & CZ \\ CZ & U_2(-\frac{\pi}{4},0) & CZ \end{bmatrix}$	$\begin{bmatrix} CZ & U_2(\pi,0) \\ CZ & I \end{bmatrix}$	$\begin{bmatrix} CZ & U_2(0,\frac{\pi}{2}) \end{bmatrix}$
Occurrence	**12**	**7741**	**965**	**1330**	**7864**	**13450**	**7416**
Most Relevant Circuit	**VQC (12)**	**MUL (4837) HS (1161) QSVM (1161)**	**MUL (948) QKNN (7)**	**QFT (1048) QFT_IN (121) VQC (121)**	**MUL (7852) QKNN (12)**	**VQC (11081) MUL (2355) QSVM (14)**	**MUL (7396) W_STATE (12) QKNN (6)**

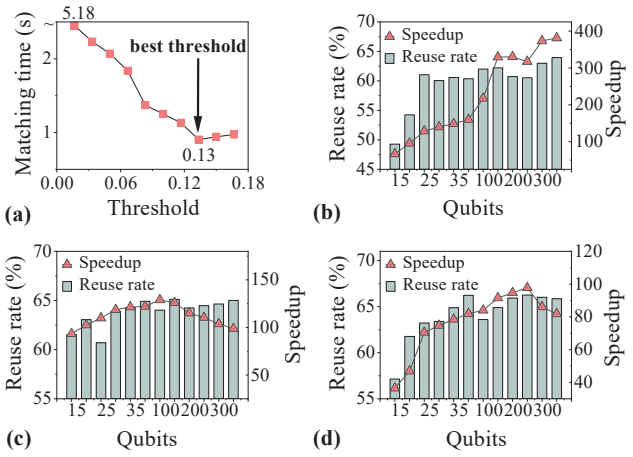


Fig. 6. (a) Evaluation of matching time under different thresholds. (b, c, d) Evaluation of pattern reuse rate and speedup under different numbers of qubits in ISING, QKNN, and DJ algorithms, respectively.

C. Evaluation of Threshold

Note that as the size of the pulse library grows, QPulseLib involves more matching between the library and extracted patterns from the target circuit. However, a large library increases the opportunities to eliminate redundant computation. As the threshold in Section V-A determines the size of the library, we evaluate the matching time per circuit under different thresholds in Figure 6 (a), which consist of the time for convolution and matrix comparison. The x-axis represents the threshold, while the y-axis indicates the matching time. As the threshold increases, fewer patterns are preserved in the library. The matching time converges to around 1 second after 0.13. Such libraries involve the minimum hash collisions, and their matching time is mainly spent on convolution, which is constant for different thresholds. We observe that the best threshold for efficient matching is 0.13, as it achieves a high reuse rate while requiring the minimum matching time. With appropriately setting the threshold, QPulseLib occupies 4 KB of memory, profiled by `memory_profiler` package [42].

D. Reuse Rate

We define the reuse rate as following:

$$\text{Reuse rate} = \frac{\text{\#Gates covered by patterns}}{\text{\#Total gates in circuit}}$$

It represents the proportion of reused pulses in the pulse sequence. QPulseLib shows a 50.3% average reuse rate and exhibits up to 82.3% rate in HS algorithm [7]. Figure 6 (b, c, d) illustrates the relationship between the reuse rate and speedup compared to OpenPulse [1] in synthesizing ISING [21], QKNN [8] and DJ [28] algorithms. The x-axis represents the number of qubits used for synthesis, while the left y-axis and right y-axis denote the pattern reuse rate and speedup of QPulseLib over OpenPulse, respectively.

As the number of qubits grows, the reuse rate of each algorithm increases and then converges to a specific value. For example, the reuse rate of ISING algorithm [21] increases from 49.2% to 68.6%. This is because such algorithms involve many repeated circuit blocks. A higher reuse rate suggests a higher speedup as more redundant computation can be eliminated. For example, when the reuse rate of DJ algorithm increases from 57.1% to 66.2%, the speedup increases from 36.41× to 97.91×. A larger-scale quantum circuit contains more gates and requires longer matching time, leading to large time growth in OpenPulse [1] and QPulseLib. However, as QPulseLib employ a greedy-based matching algorithm to maximize the reuse rate, It achieves an even higher speedup as the number of qubit increases.

E. Visualization of Reusable Patterns

Table II presents the patterns that are frequently reused in the evaluate 15 algorithms. We observe that algorithm circuits with different numbers of qubits still

share the same patterns. For instance, $\begin{bmatrix} CZ & I \\ CZ & CZ \\ CZ & I \\ CZ & CZ \end{bmatrix}$ and $\begin{bmatrix} CZ & I & I & I \\ CZ & U_2(-\frac{\pi}{4},0) & CZ & U_2(\frac{\pi}{4},0) \end{bmatrix}$ patterns frequently occur in VQC [22] and multiplier [31] algorithms. Additionally, smaller patterns such as $\begin{bmatrix} CZ & U_2(\pi,0) \\ CZ & I \end{bmatrix}$ and $\begin{bmatrix} CZ & U_2(0,\frac{\pi}{2}) \end{bmatrix}$ are more frequently observed in many algorithms such as VQC, MUL, and QSVM, providing high speedup in the pulse generation.

Most frequently-occurred patterns suggest opportunities for pulse reuse across algorithms, while some patterns mainly occur in certain algorithms. For instance, $\begin{bmatrix} CZ & U_2(\pi,0) & CZ \\ CZ & U_2(-\frac{\pi}{4},0) & CZ \end{bmatrix}$ mainly occurs in MUL algorithm. It suggests that as the scale of the quantum circuit increases, MUL contains more reusable patterns mentioned above to implement entanglement between qubits.

VII. Related Work

There are two methods available for pulse generation in a quantum system. One method involves calculating the overall unitary of the quantum circuit and generating the pulse for the whole circuit based on it. However, this approach imposes a significantly high computational cost, which grows exponentially with the number of qubits [20]. Another method involves generating individual pulses for each quantum gate operation in the quantum circuit and combining them into a pulse sequence. Previous studies show the correlation between quantum gates and control pulses [43], generate pulses using different quantum gate parameters, and accomplish pulse generation through modeling and transmission to quantum hardware [1]. AccQOC [2] introduces a modified approach to accelerate pulse generation. This approach utilizes the minimum spanning tree algorithm for group matching and dynamic updating. However, this approach results in a lower reuse rate, as dynamic circuit matching updating requires a significant amount of synthesis time.

Quantum optimal control is a methodology that aims to optimize pulse generation and calibration. The optimization can be conducted by guiding the system's state evolution along the optimal path [44]. The control objectives include improving the fidelity of quantum gates [45], preparing entangled qubit states [46], and determining optimal pulse sequences [47]. For example, determining the optimal pulse sequence involves studying the sequence under the maximum amplitude constraint [48], generating pulse sequences that consider the finite time resolution of arbitrary pulse generators [49], and determining the minimum time required to generate a pulse sequence [50] that evolves the quantum system to the desired state, while taking into account the coherence of the quantum system.

VIII. Conclusion

In this study, we present QPulseLib, a pulse generation design that reuses frequent pulses. Firstly, we extract reusable sub-circuit as patterns from small-scale quantum algorithm circuits and pre-generate pulses for each pattern. We then use convolution operators for pattern matching and conduct pulse generation and concatenation to obtain the pulse sequence for the circuit. The method has good scalability, making QPulseLib useful for large-scale circuit synthesis.

Acknowledgement

This work was funded by Zhejiang Pioneer (Jianbing) Project (No. 2023C01036). This work was also supported in part by the National Natural Science Foundation of China under Grant (No. 61825205). We would like to thank Qiujiang Guo for sharing superconducting quantum devices and providing insightful advice.

References

[1] T. Alexander, N. Kanazawa, D. J. Egger *et al.*, "Qiskit pulse: programming quantum computers through the cloud with pulses," *Quantum Science and Technology*, vol. 5, no. 4, p. 044006, 2020.

[2] J. Cheng, H. Deng, and X. Qia, "Accqoc: Accelerating quantum optimal control based pulse generation," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 543–555.

[3] L. Clinton, J. Bausch, and T. Cubitt, "Hamiltonian simulation algorithms for near-term quantum hardware," *Nature communications*, vol. 12, no. 1, p. 4989, 2021.

[4] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[5] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," *arXiv preprint arXiv:1411.4028*, 2014.

[6] T. Grurl, J. Fuß, and R. Wille, "Considering decoherence errors in the simulation of quantum circuits using decision diagrams," in *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, 2020, pp. 1–7.

[7] G. H. Low and I. L. Chuang, "Optimal hamiltonian simulation by quantum signal processing," *Physical review letters*, vol. 118, no. 1, p. 010501, 2017.

[8] Y. Ruan, X. Xue, H. Liu *et al.*, "Quantum algorithm for k-nearest neighbors classification based on the metric of hamming distance," *International Journal of Theoretical Physics*, vol. 56, pp. 3496–3507, 2017.

[9] X. Zhou, Y. Feng, and S. Li, "A monte carlo tree search framework for quantum circuit transformation," in *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, 2020, pp. 1–7.

[10] B. Tan and J. Cong, "Optimal layout synthesis for quantum computing," in *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, 2020, pp. 1–9.

[11] G. Aleksandrowicz, T. Alexander, P. K. Barkoutsos *et al.*, "Qiskit: An open-source framework for quantum computing," 2019.

[12] H. Fan, C. Guo, and W. Luk, "Optimizing quantum circuit placement via machine learning," in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 19–24.

[13] M. Alam, A. Ash-Saki, J. Li *et al.*, "Noise resilient compilation policies for quantum approximate optimization algorithm," in *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, 2020, pp. 1–7.

[14] H. Wang, J. Gu, Y. Ding *et al.*, "Quantumnat: quantum noise-aware training with noise injection, quantization and normalization," in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 1–6.

[15] P. Das, E. Kessler, and Y. Shi, "The imitation game: Leveraging copycats for robust native gate selection in nisq programs," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 787–801.

[16] S. Park, D. Kim, M. Kweon *et al.*, "A fast and scalable qubit-mapping method for noisy intermediate-scale quantum computers," in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 13–18.

[17] S. J. Glaser, U. Boscain, T. Calarco *et al.*, "Training schrödinger's cat: Quantum optimal control: Strategic report on current status, visions and goals for research in europe," *The European Physical Journal D*, vol. 69, pp. 1–24, 2015.

[18] N. Khaneja, T. Reiss, C. Kehlet *et al.*, "Optimal control of coupled spin dynamics: design of nmr pulse sequences by gradient ascent algorithms," *Journal of Magnetic Resonance*, vol. 172, no. 2, pp. 296–305, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1090780704003696

[19] G. S. Uhrig, "Keeping a quantum bit alive by optimized π-pulse sequences," *Physical Review Letters*, vol. 98, no. 10, p. 100504, 2007.

[20] Y. Chen, Y. Jin, F. Hua *et al.*, "A pulse generation framework with augmented program-aware basis gates and criticality analysis," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 773–786.

[21] B. K. Chakrabarti, A. Dutta, and P. Sen, *Quantum Ising phases and transitions in transverse Ising models*. Springer Science & Business Media, 2008, vol. 41.

[22] A. Blance and M. Spannowsky, "Quantum machine learning for particle physics using a variational quantum classifier," *Journal of High Energy Physics*, vol. 2021, no. 2, pp. 1–20, 2021.

[23] D. C. McKay, C. J. Wood, S. Sheldon *et al.*, "Efficient z gates for quantum computing," *Physical Review A*, vol. 96, no. 2, p. 022330, 2017.

[24] L. Hales and S. Hallgren, "An improved quantum fourier transform algorithm and applications," in *Proceedings 41st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2000, pp. 515–525.

[25] S. Bagherinezhad and V. Karimipour, "Quantum secret sharing based on reusable greenberger-horne-zeilinger states as secure carriers," *Physical Review A*, vol. 67, no. 4, p. 044302, 2003.

[26] H. Qin, R. Tso, and Y. Dai, "Multi-dimensional quantum state sharing based on quantum fourier transform," *Quantum Information Processing*, vol. 17, pp. 1–12, 2018.

[27] E. Brainis, L.-P. Lamoureux, N. Cerf *et al.*, "Fiber-optics implementation of the deutsch-jozsa and bernstein-vazirani quantum algorithms with three qubits," *Physical review letters*, vol. 90, no. 15, p. 157902, 2003.

[28] D. Collins, K. Kim, and W. Holton, "Deutsch-jozsa algorithm as a test of quantum computation," *Physical Review A*, vol. 58, no. 3, p. R1633, 1998.

[29] R. Mengoni and A. Di Pierro, "Kernel methods in quantum machine learning," *Quantum Machine Intelligence*, vol. 1, no. 3-4, pp. 65–71, 2019.

[30] R. Laflamme, C. Miquel, J. P. Paz *et al.*, "Perfect quantum error correcting code," *Physical Review Letters*, vol. 77, no. 1, p. 198, 1996.

[31] M. S. Islam, M. M. Rahman, Z. Begum *et al.*, "Low cost quantum realization of reversible multiplier circuit," *Information technology journal*, vol. 8, no. 2, pp. 208–213, 2009.

[32] A. Cabello, "Bell's theorem with and without inequalities for the three-qubit greenberger-horne-zeilinger and w states," *Physical Review A*, vol. 65, no. 3, p. 032108, 2002.

[33] P. W. Shor, "Introduction to quantum algorithms," in *Proceedings of Symposia in Applied Mathematics*, vol. 58, 2002, pp. 143–160.

[34] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimedia Tools and Applications*, vol. 80, pp. 8091–8126, 2021.

[35] M. Cho and D. Brand, "Mec: memory-efficient convolution for deep neural network," in *International Conference on Machine Learning*. PMLR, 2017, pp. 815–824.

[36] A. Xygkis, L. Papadopoulos, D. Moloney *et al.*, "Efficient winograd-based convolution kernel implementation on edge devices," in *Proceedings of the 55th Annual Design Automation Conference (DAC)*, 2018, pp. 1–6.

[37] *Qiskit transpiler*, IBM & Co, LLC, Qiskit Development Team, 2023, https://qiskit.org/documentation/apidoc/transpiler.html#transpiler-api.

[38] C. R. Harris, K. J. Millman, S. J. Van Der Walt *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[39] *Qiskit Package*, IBM & Co, LLC, Qiskit Development Team, 2023, https://qiskit.org/.

[40] *Rigetti Systems: Aspen-M-3 Quantum Processor*, Rigetti & Co, LLC, 2019, https://pyquil-docs.rigetti.com/en/v3.5.4/apidocs/pyquil.quiltcalibrations.html.

[41] *pyQuil document v3.5.4: Pulses and Waveforms*, Rigetti & Co, LLC, 2019, https://pyquil-docs.rigetti.com/en/v3.5.4/quilt_waveforms.html#Compile-Time-versus-Run-Time.

[42] P. G. Fabian Pedregosa, "Memory profiler." January 2023, https://github.com/pythonprofilers/memory_profiler.

[43] A. Eckstein, B. Brecht, and C. Silberhorn, "A quantum pulse gate based on spectrally engineered sum frequency generation," *Optics express*, vol. 19, no. 15, pp. 13 770–13 778, 2011.

[44] N. Leung, M. Abdelhafez, J. Koch *et al.*, "Speedup for quantum optimal control from automatic differentiation based on graphics processing units," *Physical Review A*, vol. 95, no. 4, p. 042318, 2017.

[45] R. L. Kosut, M. D. Grace, and C. Brif, "Robust control of quantum gates via sequential convex programming," *Phys. Rev. A*, vol. 88, p. 052326, Nov 2013. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.88.052326

[46] M. H. Goerz, G. Gualdi, D. M. Reich *et al.*, "Optimizing for an arbitrary perfect entangler. ii. application," *Phys. Rev. A*, vol. 91, p. 062307, Jun 2015. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.91.062307

[47] T. W. Borneman, M. D. Hürlimann, and D. G. Cory, "Application of optimal control to cpmg refocusing pulse design," *Journal of Magnetic Resonance*, vol. 207, no. 2, pp. 220–233, 2010.

[48] T. E. Skinner, T. O. Reiss, B. Luy *et al.*, "Reducing the duration of broadband excitation pulses using optimal control with limited rf amplitude," *Journal of Magnetic Resonance*, vol. 167, no. 1, pp. 68–74, 2004.

[49] F. Motzoi, J. M. Gambetta, S. T. Merkel *et al.*, "Optimal control methods for rapidly time-varying hamiltonians," *Phys. Rev. A*, vol. 84, p. 022307, Aug 2011. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.84.022307

[50] Q.-M. Chen, R.-B. Wu, T.-M. Zhang *et al.*, "Near-time-optimal control for quantum systems," *Phys. Rev. A*, vol. 92, p. 063415, Dec 2015. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.92.063415