SmartQCache: Fast and Precise Pulse Control With Near-Quantum Cache Design on FPGA

Liqiang Lu, Wuwei Tian, Xinghui Jia, Zixuan Song, Siwei Tan, and Jianwei Yin, *Member, IEEE*

***Abstract*—Quantum pulse serves as the machine language of superconducting quantum devices, which needs to be synthesized and calibrated for precise control of quantum operations. However, existing pulse control systems suffer from the dilemma between long synthesis latency and inaccuracy of quantum control systems. compute-in-CPU synthesis frameworks, like IBM Qiskit Pulse, involve massive redundant computation during pulse calculation, suffering from a high computational cost when handling large-scale circuits. On the other hand, field-programmable gate array (FPGA)-based synthesis frameworks, like QuMA, faces inaccurate pulse control problem. In this article, we propose both compute-in-CPU and all-in-FPGA solutions to collaboratively solve the latency and inaccuracy problem. First, we propose QPulseLib, a novel compute-in-CPU library with reusable pulses that can directly provide the pulse of a circuit pattern. To establish this library, we transform the circuit and apply convolutional operators to extract reusable patterns and precalculate 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 calculation, compared to Qiskit Pulse and AccQOC. Moreover, we extend the design as a fast and precise all-in-FPGA pulse control approach using near-quantum cache design, SmartQCache. To be specific, we employ a two-level cache to hold reusable pulses of frequently-used circuit patterns. Such a design enables pulse prefetching in near-quantum peripherals, dramatically reducing the end-to-end synthesis latency. To achieve precise pulse control, SmartQCache incorporates duration optimization and pulse sequence calibration to mitigate the execution errors from imperfect hardware, crosstalk, and time shift. Experimental results demonstrate that SmartQCache achieves 294.37× and 145.43× speedup in pulse synthesis compared to Qiskit Pulse and AccQOC. It also reduces the pulse inaccuracy by 1.27× compared to QuMA.**

***Index Terms*—Field-programmable gate array (FPGA), presynthesis, pulse generation, quantum computing.**

Received 12 April 2024; revised 26 October 2024; accepted 4 November 2024. Date of publication 13 November 2024; date of current version 23 April 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62472374; in part by the National Key Research and Development Program of China under Grant 2023YFF0905200; and in part by the Zhejiang Pioneer (Jianbing) Project under Grant 2023C01036. This article was recommended by Associate Editor M. Soeken. *(Corresponding authors: Siwei Tan; Jianwei Yin.)*

Liqiang Lu, Wuwei Tian, Xinghui Jia, Siwei Tan, and Jianwei Yin are with the College of Computer Science, Zhejiang University, Hangzhou 310058, Zhejiang, China (e-mail: liqianglu@zju.edu.cn; sonder@zju.edu.cn; jxhhhh@zju.edu.cn; siweitan@zju.edu.cn; zjuyjw@zju.edu.cn).

Zixuan Song is with ZJU-Hangzhou Global Scientific and Technological Innovation Center, Hangzhou 311215, Zhejiang, China (e-mail: 21836036@zju.edu.cn).

Digital Object Identifier 10.1109/TCAD.2024.3497839

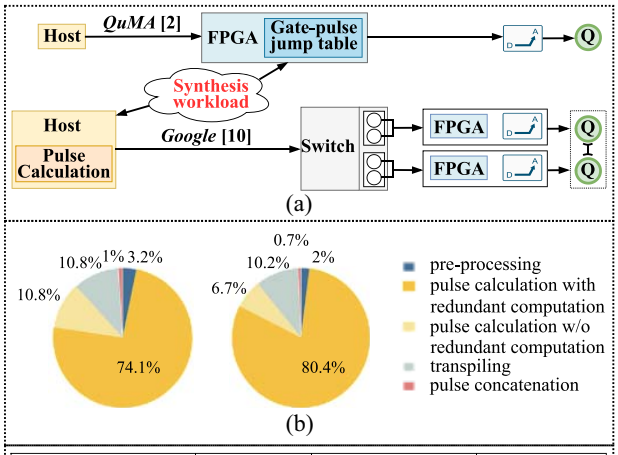
I. Introduction

QUANTUM computing attracts wide attention for its potential acceleration in dealing with tough computation challenges, such as molecular simulation [4], many-body physics [5], [6], and cryptography [7]. For superconducting quantum devices, quantum pulses, generated from quantum circuits, are used to describe all kinds of quantum operations at the hardware level [1]. The pulse control system plays an important role in high-performance quantum computing. As the superconducting qubits are very fragile and sensitive to external environmental disturbances [8], the synthesis latency and accuracy of pulse sequences directly affect the efficiency of quantum program execution [9].

Typical quantum pulse control system, such as Google Sycamore [10], employs the architecture of host CPU and distributed field-programmable gate arrays (FPGAs) to conduct discrete pulse synthesis and pulse execution, respectively, as illustrated in Fig. 1(a). To be specific, the host is responsible for pulse calculation and centralized pulse calibration. The distributed FPGAs connected with digital-analog converter (DAC) boards focus on the timing control of analog pulses for the operation of quantum devices. Alternatively, QuMA [2] and QuAPE [11] present instruction-based quantum control microarchitecture that applies pulse jump tables combined with gate pulse calibration on distributed FPGAs to achieve complex analog pulse control.

Generally, the pulse calculation process suffers from a high latency cost, especially for large-scale quantum circuits on compute-in-CPU systems, due to the overwhelming computational complexity of calculating the parameters of pulses. For example, it takes 2190.67 s to generate the pulse sequence for a 300-qubit quantum multiplier circuit using Qiskit Pulse [1], which involves 191 903 single-qubit gates and 121 095 two-qubit gates. Fig. 1(b) provides the breakdown of the synthesis latency on the circuits of hamiltonian simulation (HS) [12] and QKNN [13], 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 calculation, 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 Qiskit Pulse is used for the same circuit pattern.

Fig. 1(c) summarizes the features of different pulse control systems, showing that existing systems suffer from the dilemma between synthesis latency and execution accuracy. Although the compute-in-CPU system exhibits high execution



Architecture	Google [10]	QuMA [2]	SmartQCache
Workload assignment	Host CPU	Jump table	Pulse caching
Sequence calibration	✓	⊗	✓
Duration optimization	✓	⊗	✓
Synthesis latency	8.3 *s* ☹	5.9 *ms* ☺	37.1 *ms* ☺
10-qubit GHZ fidelity	80.2% ☺	69.3% ☹	76.1% ☺

(c)

Fig. 1. Dilemma between synthesis latency and accuracy of superconducting quantum control system. (a) Existing pulse control system. (b) Time breakdown for circuit synthesis for HS and QKNN. (c) Comparison of 3 quantum pulse control system architectures.

accuracy by collecting and calibrating pulse sequences, it faces long synthesis latency due to limited parallelism and bandwidth. For example, the host CPU takes around 8.3 s to generate a sequence of a 15-qubit quantum Fourier transform (QFT) circuit using Qiskit Pulse. On the other hand, the existing all-in-FPGA architecture design fails to provide accurate pulse sequences because of the coarse-grained optimization of complex pulses. For example, QuMA only adopts single gate pulse calibration, not the entire sequence, without the optimization for reducing hardware defects and crosstalk in neighboring qubits and couplers. When executing a 10-qubit GHZ circuit, the state fidelity reaches 69.3%.

In this article, we propose QPulseLib, a compute-in-CPU pulse library that consists of reusable patterns to reduce the overall synthesis latency for pulse calculation. 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 subcircuit 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 precalculated 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 words, we can directly obtain the pulse of the matched pattern from QPulseLib without calculating the pulse gate by gate.

In addition, we extend the method as an efficient fast, and precise all-in-FPGA pulse control system, SmartQCache. The key idea is near-quantum computing that elaborates pulse sequences on an embedded cache, which can not only shrink the overall latency but also achieve high pulse accuracy. Concretely, SmartQCache features a two-level quantum cache to house reusable pulse segments and accelerate pulse synthesis by pulse prefetching. The L1 cache is dynamically updated depending on the reuse frequency of pulse segments. While the L2 cache is static, housing pregenerated pulses of multiple circuit blocks. SmartQCache adopts gate list representation and an encoding method to fetch the matching pulse segments from the quantum cache. After obtaining the generated sequence from the two-level cache, we fine-tune the time interval between pulses for optimal timing control. To avoid redundant computational costs, we propose a calibration-in-cache scheme that utilizes partially calibrated pulses of XY channels in the cache. Besides, the cache is integrated with crosstalk calibration by adding compensation sequences. As a result, as shown in Fig. 1(c), SmartQCache is able to balance both pulse synthesis latency and pulse accuracy.

A preliminary version of this article will appear in ICCAD 2023 [14], we proposed to accelerate pulse calculation with reusable patterns. In this article, we extend previous work with near-quantum cache design, serving as a fast and precise all-in-FPGA quantum pulse control system. To be specific, we employ a two-level cache to hold reusable patterns for pulse prefetching. We also incorporate duration optimization and pulse sequence calibration to mitigate execution errors. Finally, we apply our cache design to real-world quantum hardware, balancing the tradeoff between synthesis latency and pulse accuracy. The contributions of this article are summarized as follows.

1) We propose *QPulseLib* to reduce redundant calculations, which provides significant compute-in-CPU acceleration in the pulse calculation, compared to the current state-of-the-art method [3].
2) 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 subcircuit that is reused by multiple circuit benchmarks.
3) We propose *SmartQCache*, a novel pulse control architecture with a near-quantum cache that provides speedup in pulse synthesis meanwhile keeping high accuracy.
4) We propose a complete pulse calibration method to reduce the dominant error of imperfect hardware and crosstalk. We also develop a precalculated distortion method for pulses in XY channels, further utilizing cached data to reduce redundant calculations.

The evaluation results demonstrate that compared to Qiskit Pulse [1] and AccQOC [3], QPulseLib achieves 158.46× and 16.03× speedup for pulse calculation latency. In addition, compared to Qiskit Pulse [1] and AccQOC [3], SmartQCache achieves 294.37× and 145.43× speedup in pulse calculation synthesis. Meanwhile, SmartQCache achieves over 1.27× pulse accuracy improvement compared to QuMA [2], serving as a fast and precise pulse control system.

II. Background

A. Quantum Pulse Control System

Quantum pulse is the hardware-level representation of quantum circuits to control the state evolution of superconducting

qubits [15]. Pulses offer a time-evolving path in Hilbert space to guide state changing for qubits on each time step [1]. Generally, the pulse control system consists of pulse synthesis and pulse execution. The pulse synthesis unit conducts discrete pulse calculation and pulse optimization according to quantum hardware device settings. The pulse execution unit performs precise timing control following a chronological schedule and produces analog signals to control the quantum chips.

The pulse control system is expected to produce accurate discrete pulses in a low synthesis latency, which requires adopting a reasonable architecture to perform pulse synthesis and execution. Existing quantum pulse control systems include two types of architectures. The architecture of compute-in-CPU system assigns pulse synthesis workload to the host and employs on-board FPGAs to conduct pulse execution [10]. The architecture of all-in-FPGA system employs an FPGA to perform pulse synthesis and precise timing control with DACs [2].

B. Quantum Pulse Synthesis

The pulse synthesis includes 4 steps to generate a pulse sequence with high accuracy, including pulse calculation, pulse duration optimization, pulse calibration, and timing optimization.

Pulse Calculation: The pulses of gates in the quantum circuit are calculated and separated into five types of channels, XY I-Q channels, the Z channel, the readout channel, and the Z channel of couplers. The pulse sequence within each channel consists of a series of gate pulses arranged on fixed-time steps *dt*, depending on the sampling frequency of DAC. Gate pulses are calculated through basic pulses, such as Gaussian and sinusoidal pulses. These basic pulses undergo overlaying, stretching, and mixing to form the eventual gate pulses. For example, the description of an RX gate pulse at each time step t_i is

$$S[i] = \text{Amp} \times \text{h}_{\text{mix}}(\text{Gaussian}(\text{f}, \sigma)(\text{t}_\text{i})) \tag{1}$$

where the f and σ indicates the frequency and standard deviation of Gaussian pulse, h_{mix} means differential mixing function, and *Amp* is the amplitude of the pulse [1].

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 [1], as shown in Fig. 2(a). However, with the help of QPulseLib illustrated in Fig. 2(c), pulse can be retrieved and concatenated with the pregenerated pulses from the pulse library after circuit matching, as illustrated in Fig. 2(b). Finally, many individual pulse segments are scheduled in time and concatenated together to form the final sequence.

Pulse Duration Optimization: Different physical implementations of operations for XY and Z channels in superconducting quantum hardware lead to redundant intervals between adjacent RX gate execution and CZ gate execution. On the other hand, due to the existence of decoherence errors, reducing the duration of pulse sequences can directly enhance the precision of quantum program execution. To this end, pulse duration optimization indicates adjusting the execution intervals among different channels with quantum optimal control (QOC) [16], [17], benefiting precise pulse control.

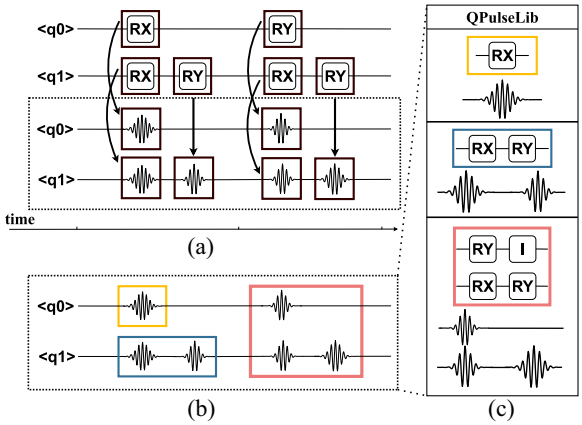


Fig. 2. (a) Pulse generated by time schedule. Pulses are calculated following time schedule. (b) Pulse generated by QPulseLib. Pulse sequence is generated using pregenerated pulses after circuit matching. (c) QPulseLib.

Pulse Calibration: The ideal pulse sequence for real-world pulse control demands accurate pulse shape [9], [18], which requires overcoming crosstalk and pulse distortion: 1) crosstalk arises from neighboring flux leakage from other qubits and couplers. The proportion of flux leakage within each channel can be determined through routine calibration steps and 2) pulse distortion arises due to nonlinear response, wire noise, or unwanted coupling effects. To describe the distortion, the transfer function of wires and analog devices can be modeled as

$$H(\omega) = 1 + \epsilon \frac{i\omega\tau}{1 + i\omega\tau} \tag{2}$$

where ϵ represents the fraction of amplitude of distortion and τ is a characteristic time-scale. The noise often occurs as amplitude or phase changes in the flux-bias line, requiring frequency domain compensation and amplitude compensation calculated from the response data of hardware. To this end, pulse compensation of crosstalk and distortion should be added to correct imperfect pulse sequences.

Timing Optimization: Executing quantum pulses on each qubit following chronological order requires pulse sequences of all channels to be orthogonal and arrive at the quantum chip simultaneously. Time shift arises from inaccurate arrival time when transmitting pulse sequence to the target quantum device through wires and analog devices. To overcome the timing inaccuracy, it needs to add accurate time compensation for each DAC channel to conduct precise analog timing control.

III. QPulseLib Overview

The insight of designing *QPulseLib* lies in the fact that although pulse calculation for executing quantum programs on hardware is inevitable, pulses can be reused through the pulse library. The workflow of QPulseLib is shown in Fig. 3. Fig. 3(a) illustrates the initial step of constructing

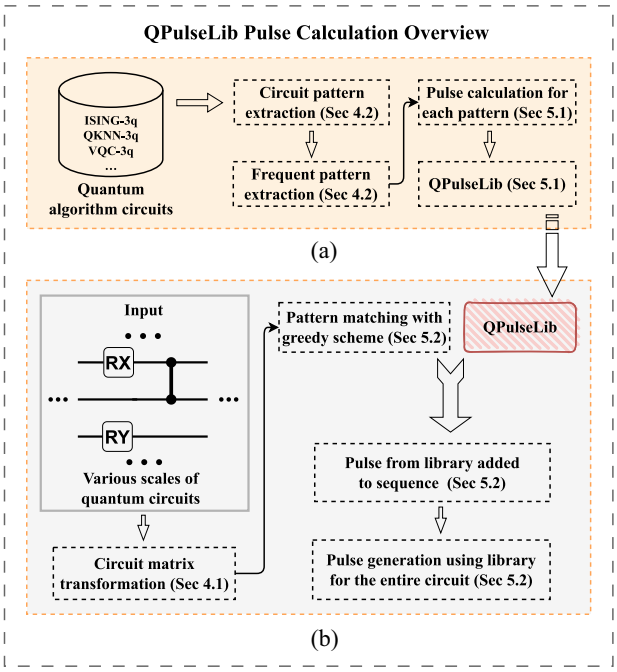


Fig. 3. Overview of QPulseLib, containing (a) construction of QPulseLib, and (b) pattern matching and pulse calculation.

the QPulseLib with a set of small-scale quantum circuits, such as the ISING model [19], QKNN model [13], and variational quantum classifier (VQC) model [20]. 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 a circuit matrix for circuit matching, which will be described in Section IV-A. The library consists of frequent reusable subcircuits extracted from these circuits and their corresponding pulses.

QPulseLib leverages a pulse library to reuse pulses and minimize synthesis latency 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. Fig. 3(b) illustrates the initial step of transforming the target circuit into a circuit matrix (Section IV-A), where each submatrix 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

A. Transforming Circuit to Matrix

Each element of the circuit matrix represents one gate, which stores information about its operation and the parameters of the 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 nth row and mth column of the matrix corresponds to a gate that operates on qubit timeline q_n in the mth layer. Note that we only consider three basis gates after circuit transpilation: 1) single-qubit gate RX; 2) single-qubit gate RY; and 3) two-qubit gate CZ. Specifically, there are three cases for the matrix assignment.

1) 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 [21], 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 $\pi/4$, is $\pi/4i$.
2) 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 nth 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.
3) 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. The parameter of gates is encoded as the value. That is to say, a block from the circuit corresponds to a submatrix, as shown in Fig. 4. This encoding scheme ensures to thorough transformation of the entire circuit into a quantitative manner.

B. Extraction Using Convolutional Operator

To accelerate the calculation 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 window includes r qubits in s layers. The convolution result is denoted as *conv_rst*. The kernel is designed to ensure that the submatrix containing the same elements always produces the same convolution output. The same *conv_rst* indicates the same submatrix 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 precalculated pulse in the library. For example, the submatrix in the red box in *Step 1* of Fig. 4, is first applied with a kernel of 2×4, obtaining the *conv_rst* of $7 + \pi i$. Next, we find the same *conv_rst* with a matched pattern in the library. Last, we fetch the pulse under this pattern. Similarly, in step 2 of Fig. 4, the submatrix 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.

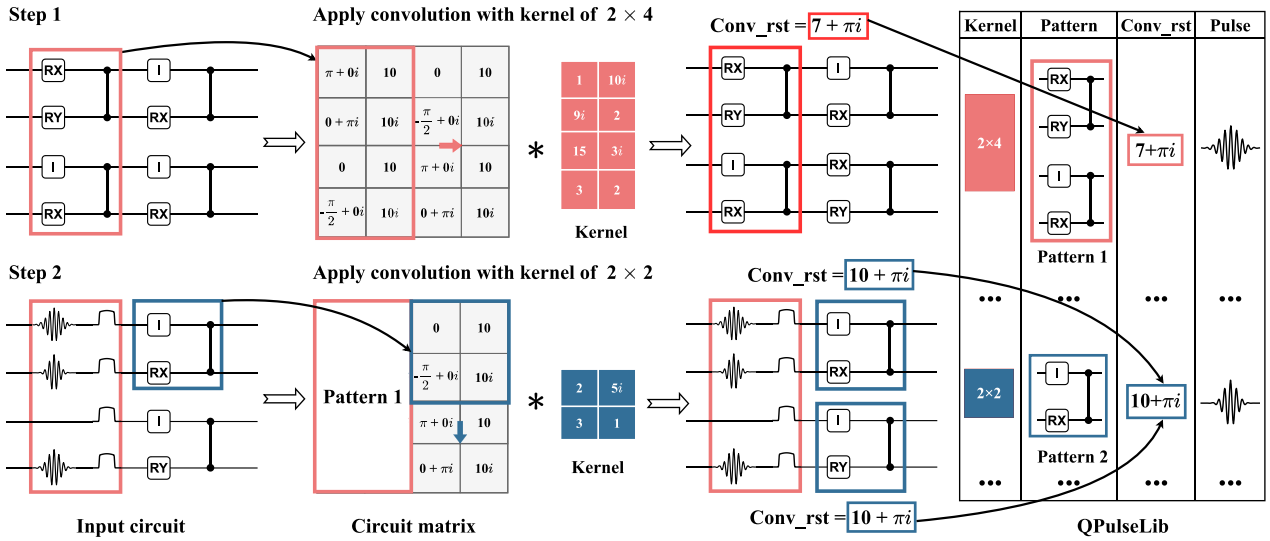


Fig. 4. Two-step process to conduct circuit matching with QPulseLib through convolution. In this example, we set β to 10.

TABLE I
QUANTUM BENCHMARKS APPLIED IN OUR EXPERIMENTS

Algorithm	Description	Qubits	#1q	#2q
QFT	Quantum Fourier Transformation [22]	15	499	231
GHZ	Preparing Greenberger–Horne–Zeilinger State [23]	15	29	14
QFT_IN	Inverse Quantum Fourier Transformation [24]	15	509	246
BV	Bernstein-Varzirani Algorithm [25]	15	11	4
DJ	Deutsche-Jozsa Algorithm [26]	15	23	14
HS	Hamiltonian Simulation [12]	15	42	28
ISING	Linear Ising Model [19]	15	46	28
QKNN	Quantum K-nearest Neighbors Algorithm [13]	15	85	56
QSVM	Quantum Support Vector Machine [27]	15	63	56
QAOA	Quantum Approximate Optimization Algorithm [28]	15	144	66
VQC	Variational Quantum Classifier [20]	15	184	448
QEC	Quantum Error Correction Code [29]	15	33	18
MUL	Quantum Multiplier [30]	15	372	242
W-STATE	Preparing Quantum W_State [31]	15	51	28
SIMON	Quantum Simon Algorithm [32]	20	48	17

Once convolution is applied to each submatrix 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 submatrix. 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. We do not apply any larger kernels due to the low matching rate, which will reduce the efficiency of pulse calculation.

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 genetic algorithm (GA) [33] 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 [3] that relies on frequent subgraph extraction, we employ convolutional operators to find reusable submatrices with pattern matching, which is a widely-used technique for identifying identical adjacent submatrices within a matrix [34]. 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) [35], thereby enhancing the efficiency of pulse calculation.

V. ACCELERATING PULSE CALCULATION

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

Algorithm 1: Algorithm for Matching and Pulse Calculation

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**
    submatrix = 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;

listed in Table I. These circuits are first transformed using Qiskit [36] 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 tradeoff 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 subcircuits. Our experiments will provide a visualization of these structural patterns and detailed evaluation results.

B. Matching Algorithm for Pulse Calculation

QPulseLib takes the target quantum circuit as input and initiates the synthesis by transforming it into a 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 subcircuit. 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 submatrix with the current pattern, we collect the submatrix and terminate the search for the involved gates.

An illustrative example of the matching algorithm is presented in Fig. 4. Given an input circuit, we consider two convolution kernels: 1) 2×4 and 2) 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 submatrix and the existing pattern, the submatrix 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 the 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.

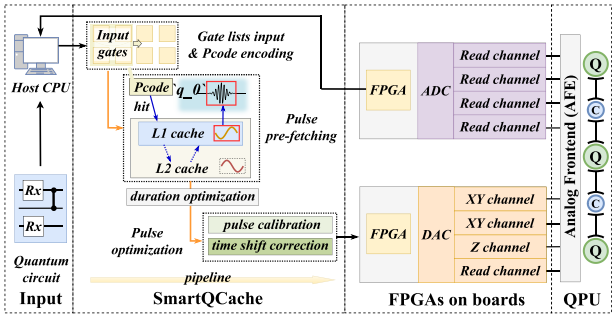


Fig. 5. Pulse control system with SmartQCache.

VI. SmartQCache Architecture

A. System Architecture Overview

The insight of SmartQCache architecture is the near-quantum design for fast synthesis and low-latency pulse transmission, meanwhile achieving high pulse accuracy by adopting centralized pulse sequence calibration in an embedded cache. We deploy a smart cache to enable pulse segments prefetching. The L1 cache is dynamically updated depending on the reuse frequency of pulses. While the L2 cache is static, containing pregenerated pulses of multiple patterns. We also add pulse optimization steps, including duration optimization with GA, pulse calibration for distortion and crosstalk, and timing optimization for time shift reduction, heading for more precise pulse control.

Fig. 5 presents the entire all-in-FPGA quantum control system with SmartQCache. At the beginning, for an input quantum circuit, the host CPU generates gate lists and sends them to SmartQCache. Upon receiving the input gates, we perform *Pcode* encoding and pattern matching, which generates input gates for cache utilization. With *Pcode*, we perform pulse calculation by prefetching the pulse segments from the two-level cache. In the pulse calculation process, initially, we conduct pattern matching for the input quantum circuit and fetch the pulses from L1 cache once receiving a *hit*. If missed in L1 cache, the search extends to the L2 cache to locate the target pulse segments. The fetched pulse segment is added to the pulse sequence to fulfill pulse calculation. Next, pulse optimization steps of duration optimization and pulse

calibration are also considered. For different stages in pulse synthesis, we design a pipeline scheme for these pulse synthesis steps. After the pulse synthesis process, SmartQCache sends the calibrated pulse sequence to on-board FPGAs. These FPGAs undertake the transmission of discrete pulses to the quantum processing unit (QPU) by producing synchronous analog signals through the channels of DAC. The analog pulses undergo wire transmission and analog devices, such as I-Q mixer and low-pass filters to form accurate control signals. The readout pulse sequence returned from QPU is captured and decoupled by analog digital converters (ADCs), and sent to the host CPU afterward. SmartQCache aims to optimize the intermediate process of pulse synthesis, to construct a fast and precise quantum pulse control system by reducing the time cost of pulse calculation and considering the complete process of pulse optimization.

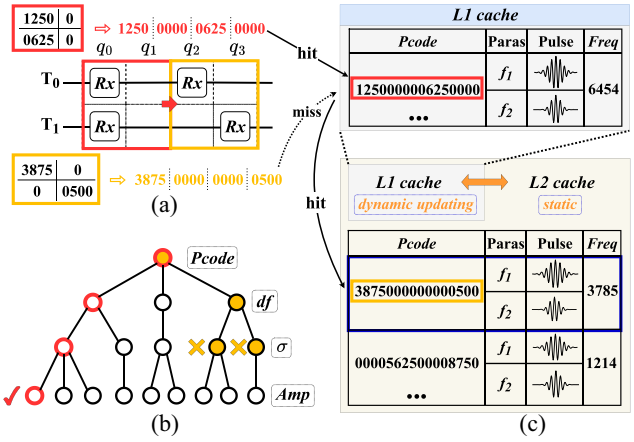


Fig. 6. Pattern matching and two-level quantum cache. (a) Pattern matching with *Pcode*. (b) Paras matching with search trees. (c) Two-level quantum cache.

B. Two-Level Cache Design

The insight of designing the two-level cache lies in the observation that there are massive reusable circuit patterns in quantum circuits, whose corresponding pulses can be cached in advance. Furthermore, we can store the pulses of circuit patterns in different levels of the cache based on their frequency of occurrence. The design of the two-level cache includes gate list format transformation, encoding method for cache hit principle, and L1 cache update policy. SmartQCache also incorporates a GA of duration optimization for the entire sequences.

Generating Circuit Gate Lists: As explicated in Section II-B, we need gate lists to describe gate collections and parameter-setting lists for quantum device setups for each gate. Each gate list stores the position along with its key parameters and each parameter-setting list contains other hardware-level set-ups about the operation. For a circuit of N qubits and M layers, the RX gate list can be stored in a sparse format, recording gate location and the phase of x-axis rotation. The *location* of an RX gate in nth qubit and mth layer of the circuit is regarded as $(n\text{th}, m\text{th})$, then we store the phase into the *rotation*. In the gate list of CZ gates, we similarly record the position of CZ gates, but store its voltage bias instead of *rotation*. Also, for coupler lists, the *location* indicates the two coupling qubits and the timeline of layers. The format of gate lists containing essential information for the circuit and the parameter-setting lists ensures accurate reproduction of pulses, allowing adequate utilization of the two-level cache.

To accelerate the synthesis of pulses, our goal is to identify the reusable patterns in the matrix representation. Therefore, we propose an encoding method for gates in the gate list. Different from QPulseLib, we propose an encoding method for circuit patterns to save the bandwidth and exploit the pulse library extracted beforehand. As the rotation angles range between $-\pi$ and π, to ensure accuracy, we select the upper 32 bits of the fractional part of values as the phase parameter. With the transmission bandwidth of k bits, $k/32$ adjacent gates in the quantum circuit are combined into a new value indicating the identity of circuit patterns, labeled as *Pcode*. *Pcode* is also the search key of the cache. If the *Pcode* of the input circuit within the sliding window matches the *Pcode* in the cache, we can fetch the pregenerated pulse and add it to the sequence of corresponding channels. For instance, Fig. 6(a) shows a matching example of RX gate lists, we set the sliding window to be square with the width $w = 2$. The rotation values of RX gates in the red box are encoded as a *Pcode* for subcircuit matching, so as the right yellow box.

Cache Details: The two-level cache stores *Pcodes* as keys and precalculated pulses for reusable circuit patterns as corresponding values. We set a two-level cache, an L1 cache that offers high-speed access with a small capacity, and an L2 cache that contains massive pulse segments and offers relatively low-speed access. When constructing the cache, we extract frequently-used circuit patterns from small-scale regular quantum algorithm circuits for pulse caching, as frequently-used circuit patterns from small-scale quantum algorithm circuits can also be reused in large-scale quantum circuits [14]. We record their frequency of occurrence and store all of the pulse segments for reusable circuit patterns in the L2 cache. For the L1 cache, due to limited storage, we set a threshold proportion β to select the top patterns according to access frequency and store the corresponding pulses for pulse prefetching. For patterns with the same *Pcode* but different parameters like gate frequency or amplitude, we store a small search tree under each *Pcode* to further examine other gate-setting parameters for accurate cache utilization.

To exploit the two-level cache, we conduct subcircuit pattern matching for a new input quantum circuit. We apply a square slide window according to the gate list and generate adjacent gates to encode a *Pcode*. We define a *hit* indicates a successful matching of *Pcode* and a *miss* for an unsuccessful matching. As illustrated in Fig. 6(a) and (c), when the *Pcode* of the red sliding box matches a *Pcode* in the L1 cache (a *hit*), we further continue parameters matching involving gate frequency f, standard deviation σ and amplitude *Amp* with search trees, as shown in Fig. 6(b). If matched, the pulses in the cache can be fetched and added to the sequence. Equation (1) indicates

that if f and σ match but the *Amp* is different, we can just fetch an existing pulse under the current *Pcode* and change the *Amp* simply through multiplication. However, the failure of search for f or σ is regarded as a *miss*. When encountering a *miss* in L1-Cache, we can turn to the L2 cache for further pattern matching, as shown in Fig. 6(c). If the search in the L2 cache still returns a *miss*, we further conduct function-based calculations to generate the pulse segment.

Considering various input quantum circuits, the two-level cache should be updated at times. We set an attribute *Freq* to record the *hit* times of the *Pcode*. The two-level pulse performs *hit* frequency analysis based on heap sorting. After a fixed period t_{up}, the pulses of patterns with higher *Freq* are transferred to the L1 cache, and pulse segments for patterns of lower *Freq* are relocated to the L2 cache.

In superconducting quantum devices, pulse envelopes (e.g., Gaussian) are fine-tuned through grid search and manual adjustments. This process involves sweeping and adjusting parameters, such as pulse amplitude, pulse width, pulse timing (center), rise and fall times, and pulse frequency. These parameters are optimized according to the optimal combination within the parameter space that yields the highest fidelity. Once identified, this parameter set generates the pulses during the current calibration cycle. Note that after each calibration, the L1/L2 cache is fully recalculated and updated according to the newly optimized parameters. This update process occurs at the beginning of each calibration cycle, ensuring the prestored pulses align with the current calibration requirements. These updated pulses are then uploaded to the FPGA to ensure efficient execution during the calibration cycle.

Duration Optimization: After obtaining raw pulse sequences with a two-level cache, SmartQCache fine-tunes the time interval between pulses to adjust the duration of sequences. We employ a GA to find the optimal interval for pulse execution. We first randomly select the time sequence and shapes of pulses in the two-level cache as candidates. With the optimization subjective of higher pulse accuracy and lower latency, GA then performs a stochastic search, which iteratively evaluates the value of the candidates. The search concludes upon reaching the predefined iteration threshold, obtaining the optimized sequences before calibration.

VII. Pulse Calibration and Optimization

Existing approaches face the tradeoff between synthesis latency and pulse accuracy. Compute-in-CPU system leads to long calibration synthesis latency for tremendous pulse data and transmission cost to co-processors. While existing all-in-FPGA methods get low accuracy for adopting single-gate pulse calibration, not the entire sequence, neglecting crosstalk and sequence distortion caused by imperfect hardware. We provide FPGA-based precalibration strategy to store distortion-calibrated XY pulses in cache, avoiding redundant convolutions for cached pulses. Through in-cache precalibration strategy, we speedup pulse calibration by fetching precalibrated pulses and improve pulse accuracy by offering complete calibration steps to reduce crosstalk and pulse distortion.

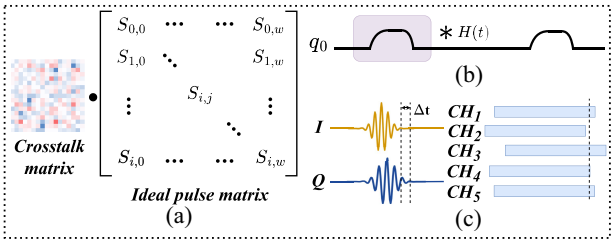


Fig. 7. Pulse calibration and time shift correction. (a) Crosstalk pulse calibration. (b) Pulse distortion calibration. (c) Time shift correction.

A. Pulse Calibration in Cache

The insight of pulse calibration lies in the observation that the dominant device noise and crosstalk from imperfect hardware can be eliminated by adding pulse compensation. SmartQCache employs a distortion-calibrated RX gate cache to avoid repetitive convolution and a timeline buffer to reuse the crosstalk matrix when calculating crosstalk compensation. We also carefully arrange the pulse correction order for different types of noise to minimize redundant calculations during the pulse calibration process.

Pulse distortion often occurs as changes in amplitude or phase of executed pulses arise in the flux-bias line, represented by $H(\omega)$ as detailed in (2). Therefore, we can convert the pulse sequence to the frequency domain and multiply it with $H(\omega)$, or convert $H(\omega)$ to the time domain as $H(t)$ and deploy convolutional operators, as shown in Fig. 7(b). Hereby we effectively mitigate pulse distortion, optimizing the pulse shape within each channel. For quantum hardware devices, crosstalk comes from unwanted geometric coupling due to neighboring flux leakage. Simply, we can measure the flux leakage for an operation among channels to obtain a *crosstalk matrix* [37], as shown in Fig. 7(a). Calibration is then achieved by multiplying *crosstalk matrix* with the pulse sequence matrix to generate compensation pulses for the target channels.

Varying degrees of crosstalk in channels lead to different calibration strategies: 1) for pulses within the XY channel, due to little crosstalk when executing RX gates, we precalibrate distortion while loading the RX gate cache. Therefore, when prefetching the pulse sequence with the two-level cache, we can directly add the distortion-calibrated pulse into the blank sequence, saving computing overhead by avoiding redundant convolutions. Subsequently, crosstalk calibration is applied to these sequences and 2) for pulses within the Z channel and coupler Z channel, because of inevitable crosstalk when executing CZ gates, we cannot conduct pulse distortion calibration before crosstalk calibration, as amplitudes of the pulse sequence change significantly. We generate duration-optimized sequences and propose crosstalk calibration first, then deploy distortion calibration for sequences of these channels.

B. Timing Optimization and Pipelining

Nonorthogonal I-Q sequences bring errors when executing quantum pulses, leading to poor accuracy. As illustrated in Fig. 7(c), the I-Q sequence in the XY channels for controlling the same qubit exhibits a time difference Δt, which is called

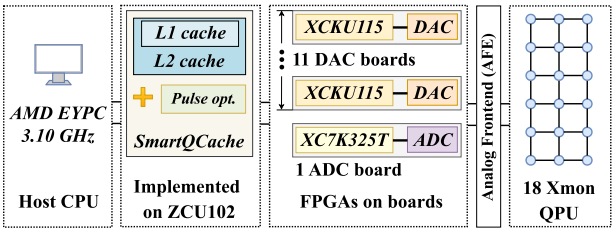


Fig. 8. SmartQCache implementation.

time shift caused by signal transmission errors. In addition, various lengths of wires will also cause time differences in quantum execution. We collect time compensation for each channel to ensure the synchronization of pulse control. For sequences across all DAC channels, we set up a standard time step and adjust *time shifts*, as shown in Fig. 7(c).

SmartQCache employs a 3-stage process for efficient overlapped computation with pipelining. The transformation of gate lists is computed offline for a quantum circuit to save on-chip resources. In stage 1, we capture the operations of valid qubits from gate lists to obtain the timeline buffer, then we conduct *Pcode* encoding for input gates. In stage 2, we conduct circuit pattern matching to prefetch the pulse segments from the cache and calculate pulses for the unmatched gates. We also apply duration optimization to adjust the time interval between channels. In stage 3, we deploy pulse calibration and *time shift* correction to generate the entire pulse sequence for the input quantum circuit.

VIII. Evaluation

A. Experimental Setup

QPulseLib 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 [38]. We set the duration of single-qubit gate to be 50 ns while the two-qubit gate to be 150 ns, similar to Rigetti quantum device. Considering QPulseLib focus on pulse calculation on CPU, we measure the latency of this process for QPulseLib synthesis [39].

SmartQCache Platform and Implementation: Publicly available quantum computers, such as IBMQ [40], do not reserve interfaces for near-quantum pulse-level control and execution with hardware-level calibration [1]. We set the target device of SmartQCache to a self-developed superconducting quantum device, with 18 Xmon qubits arranged in the 3×6 grid qubit topology. Our quantum device deploys RX, RZ, and CZ gates as basis gates. Fig. 8 presents the implementation of SmartQCache architecture. We employ AMD EYPC 9554, 3.10 GHz as our host CPU. SmartQCache is implemented on Xilinx ZCU102 with an operated frequency of 200 MHz. Since SmartQCache involves complete pulse generation steps, including pulse calculation and pulse calibration, we test the overall pulse generation synthesis latency and pulse accuracy for SmartQCache. For sequential pulse execution, we deploy 11 boards equipped with Xilinx XCKU115 and DAC, and a board equipped with Xilinx XC7K325T and ADC. We employ arbitrary waveform generators, I-Q mixers, amplifiers, and low-pass filters as analog frontend, benefiting precise control of 18 Xmon QPU.

QPulseLib Comparison: For the pulse calculation synthesis on the host, we compare QPulseLib with Qiskit Pulse and AccQOC. Since Qiskit Pulse is implemented in Python, we have also implemented QPulseLib and AccQOC in Python. For QPulseLib, we use `Numpy` and `Scipy` packages for matrix multiplication and 2-D convolution. For AccQOC, we strictly follow the pulse-compilation speedup part provided in this article and successfully reproduce their circuit subgraph-matching pulse generation method with a similar speedup (9.97x in our code versus 9.88x reported in AccQOC). We use `Networkx` package to generate the complete similarity graph for circuit analysis in our implementation of AccQOC. The codes for QPulseLib and the reproduction version of AccQOC are available at (https://github.com/JanusQ/QPulseLib).

SmartQCache Comparison: Besides QPulseLib and AccQOC, we also implement the pulse duration optimization method PAQOC in Python on the host CPU, while QuMA is implemented on Xilinx ZCU102. We reproduce gate-by-gate pulse generation method in C++ with high-level synthesis, as described in QuMA paper.

Two-Level Cache Construction: We exploit 5 well-known quantum algorithms to cache reusable pulses for frequently-used circuit patterns, such as Bernstein–Varzirani (BV), Greenberger–Horne–Zeilinger (GHZ), QFT, VQC, and HS at the range of 2–4 qubits. We implement the L1 cache using Flip-Flops and L2 cache on BRAM. It takes 0.17 and 2.26 MB to implement L1 cache and L2 cache for RX gates, CZ gates and couplers, respectively.

Experimental Configurations: Considering hardware device implementation, we use RX, RZ, and CZ gates as basis gates for circuit transpilation before pulse synthesis. As RZ gates are typically realized as virtual gates with a latency of 0 ns [41], we only focus on RX gates for the XY channel and CZ gates for the Z channel in the process of pulse calculation. In the configuration of SmartQCache, we set the transmission bandwidth k of gate lists to 128 bits. The selection threshold proportion β for the L1 cache is established at 3%, and the update period t_{up} for the two-level cache is set to 2 min. The sample rate of the pulse sequences is set to 2 GHz.

B. Speedup for Pulse Calculation With QPulseLib

Fig. 9 presents the pulse calculation latency of Qiskit Pulse [1], AccQOC [3], and QPulseLib on the 15 algorithms. The x-axis represents the number of qubits, while the y-axis represents the average pulse calculation latency, measured in seconds. Overall, QPulseLib achieves significant speedup, with a speedup of 158.46× and 16.03× compared to Qiskit Pulse [1] and AccQOC [3] in pulse calculation latency, and 10.91× and 1.21× speedup for end-to-end circuit synthesis latency, respectively. This is attributed to the fact that our pulse library effectively reduces redundant computation by pulse reuse. In contrast, the pulse calculation of Qiskit Pulse involves massive redundant computation. As a result, it requires

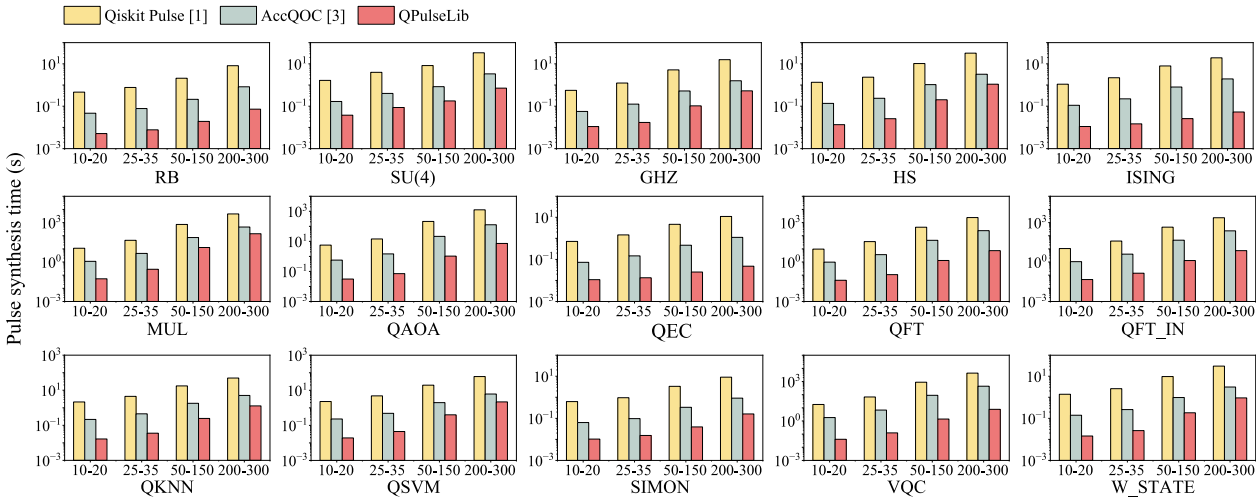


Fig. 9. Evaluation of pulse calculation time in 15 algorithms. The x-axis is the number of qubits.

571.22 s when synthesizing the QAOA algorithm [28] at 300 qubits, while QPulseLib spends 3.48 s (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 [22] at 300 qubits, AccQOC requires 114.06 s for pattern matching, while QPulseLib spends only 3.44 s based on its convolution-based pattern extraction method.

The `Numpy` and `Scipy` packages used in QPulseLib are highly optimized for numerical computations. Matrix multiplication and convolution can be parallelized efficiently, and these libraries take full advantage of modern hardware. This results in significant performance boosts, especially when working with large-scale matrix operations or signal-processing tasks involved in pulse generation. However, AccQOC introduces algorithmic complexity through graph processing, which can result in performance degradation with increasing circuit size due to the combinatorial nature of subgraph matching. While `Networkx` is a powerful graph library that supports various graph algorithms, it is a general-purpose tool. It may not be optimized explicitly for computing the minimum spanning tree (MST). Therefore, beyond the reduction in computational overhead, QPulseLib's software architecture offers significantly better execution efficiency on underlying hardware than AccQOC. This enhanced efficiency is a critical factor in QPulseLib's ability to accelerate quantum pulse generation.

C. Threshold and Reuse Rate of QPulseLib

Note that as the size of the pulse library grows, QPulseLib involves more matching time but more opportunities to eliminate redundant computation. We evaluate the matching time per circuit under different thresholds in Fig. 10(a), as the threshold increases, fewer patterns are preserved in the library. 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 1.2 MB of memory, profiled by `memray` package.

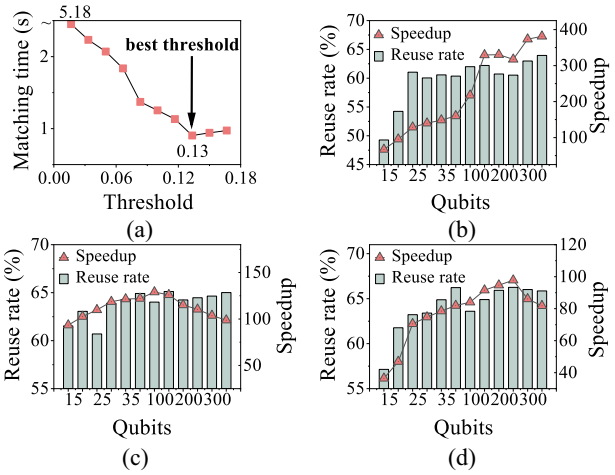


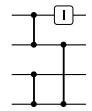
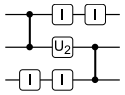
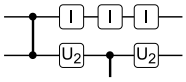
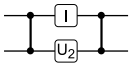
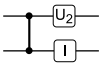
Fig. 10. (a) Evaluation of matching time under different thresholds. (b)–(d) Evaluation of pattern reuse rate and speedup under different numbers of qubits in ISING, QKNN, and DJ algorithms, respectively.

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 [19] increases from 49.2% to 68.6%. This is because such algorithms involve many repeated circuit patterns. 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×.

D. 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, $\begin{bmatrix} CZ & I \\ CZ & CZ \end{bmatrix}$ 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

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)

VQC [20] and multiplier [30] algorithms. In addition, 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-occurring 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.

E. Speedup of Pulse Synthesis With SmartQCache

Fig. 11 depicts the end-to-end synthesis latency, including pulse calculation and pulse optimization, compared to Qiskit Pulse [1] and QuMA [2] on 5 algorithm circuits at 5 [Fig. 11 (a)], 10 [Fig. 11 (b)], 15 [Fig. 11 (c)], 18 [Fig. 11 (d)] qubits. We repeat the pulse synthesis of each circuit 5 times and calculate the average synthesis latency. Overall, SmartQCache achieves a speedup of 294.37× compared to Qiskit Pulse for pulse synthesis. Considering accurate pulse calibration steps, SmartQCache marginally expands the average synthesis latency from 5.3 to 27.5 ms, compared to the nonsequence calibration method, QuMA. This is attributed to the fact that we employ a two-level cache to prefetch stored pulse segments for pulse calculation and design an encoding method for rapid circuit pattern matching. In contrast, Qiskit Pulse generates pulse sequences gate-by-gate, involving massive redundant calculations. For example, it takes around 9.76 s to accomplish the pulse synthesis for an 18-qubit VQC circuit with the typical architecture, while SmartQCache requires only 76.3 ms. As the scale of qubit increases, the synthesis latency grows more slowly compared to Qiskit Pulse, which indicates utilization of pulse caching at large-scale quantum circuits saves more time. QuMA [2] supports pulse calculation and pulse calibration. It is hard to decouple the calculation latency, so we only compare the end-to-end latency with QuMA. State-of-the-art methods

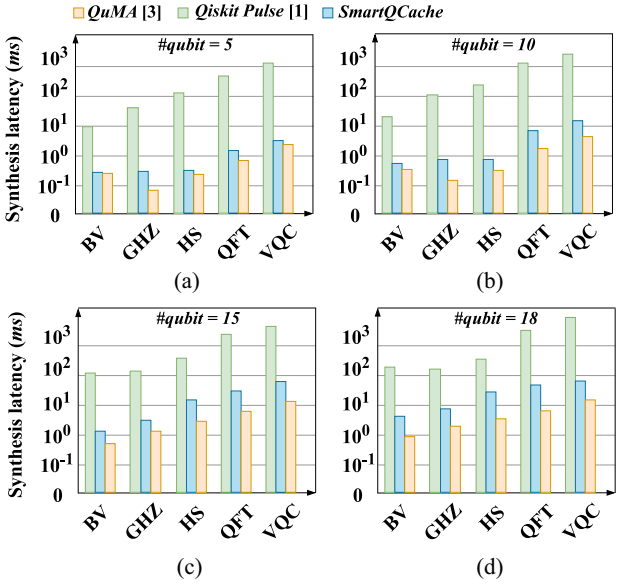


Fig. 11. Synthesis latency comparison. (a) Synthesis latency for #qubit=5. (b) Synthesis latency for #qubit=10. (c) Synthesis latency for #qubit=15. (d) Synthesis latency for #qubit=18.

like AccQOC [3] and QPulseLib [14] only focus on pulse calculation, not suitable for comparison shown in Fig. 11. We adopt compute-in-CPU architecture with these methods to accelerate pulse calculation on the host and compare SmartQCache to the optimized scheme for 5 algorithm circuits at 10 qubits. Table III presents the latency of pulse calculation of 3 methods, showing SmartQCache achieves 145.43× and 15.12× speedup of pulse calculation compared to AccQOC and QPulseLib. The results indicate that although typical architecture adopts state-of-the-art optimization approaches, SmartQCache can still outperform existing pulse calculation methods, indicating that the architecture of the near-quantum pulse synthesis is more suitable for fast pulse control systems.

F. Pulse Accuracy

Fig. 12(a) presents the pulse duration of 5 algorithm circuits at 10 qubits for QuMA [2], PAQOC [16] and SmartQCache. Overall, SmartQCache achieves 47% duration reduction for

TABLE III
EVALUATION OF PULSE CALCULATION SYNTHESIS LATENCY

Algorithm	Qiskit Pulse [1]	AccQOC [3]	QPulseLib	SmartQCache
BV	39*ms*	4.05*ms*	3.04*ms*	**0.12*ms***
GHZ	116*ms*	11.88*ms*	3.19*ms*	**0.30*ms***
HS	345*ms*	35.23*ms*	4.12*ms*	**0.28*ms***
QFT	1353*ms*	138.15*ms*	9.16*ms*	**0.55*ms***
VQC	2550*ms*	260.21*ms*	8.17*ms*	**0.94*ms***

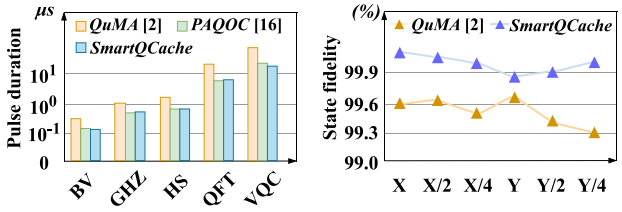


Fig. 12. (a) Duration optimization comparison. (b) Pulse accuracy of QuMA [2] and SmartQCache.

the generated sequence, compared to QuMA. Compared to the CPU-based state-of-the-art method, PAQOC, SmartQCache also achieves 2.3% duration reduction on average. The pulse duration optimization with GA finds proper intervals among sequences from different channels, reducing the execution latency of quantum programs at the hardware level.

Although compute-in-CPU architecture takes a long time to conduct pulse synthesis, it calibrates the pulse very carefully with complete pulse calibration methods to obtain accurate pulse sequences, achieving high fidelity for circuit execution. We use grid search and precise manual modulation to find out the best pulse parameters for the real-world quantum device. Qutip [42] is then used for the pulse simulation, and we also add relaxation noise and crosstalk into the simulation model. We generate pulses of X, X/2, X/4, Y, Y/2, and Y/4 produced from SmartQCache and QuMA [2] and obtain the state fidelity. As represented in Fig. 12(b), SmartQCache achieves 1.27× improvement in fidelity compared to QuMA. SmartQCache not only deploys the two-level cache to conduct fast pulse calculation but also adopts complete pulse calibration for exact pulse control, while QuMA lacks calibration for the sequence, resulting in low pulse accuracy.

IX. RELATED WORK

Typical control methods, such as Google [10] adopt the *host-FPGAs* architecture that contains the host to perform pulse synthesis and *on-board FPGAs* to conduct pulse execution, which achieves high fidelity but takes massive time for pulse synthesis. Near-quantum solutions, including QuMA [2] and QuAPE [11] offers FPGA-based solutions for quantum pulse control system. Both of them adopt *jump table on distributed FPGAs* to directly work on pulse synthesis and execution, which reduces synthesis overhead but neglects centralized pulse sequence calibration, resulting in high execution error of pulse sequences.

As for pulse synthesis optimization on CPUs, PAQOC [16], and Liang et al. [43] achieved duration optimization of the generated pulse sequence, reducing the time of quantum program execution. AccQOC [3] and QPulseLib [14] achieve pulse calculation speedup by utilizing pulses of reusable patterns through subgraph matching and convolution. However, the implementation of these methods on near-quantum hardware leads to extremely high synthesis overhead, which limits the efficiency of pulse generation acceleration.

X. CONCLUSION

In this study, we present QPulseLib and SmartQCache to figure out the dilemma between the synthesis latency and pulse accuracy as compute-in-CPU and all-in-FPGA solutions. For QPulseLib, first, we extract reusable subcircuits as patterns from small-scale quantum algorithm circuits and pregenerate pulses for each pattern. We then use convolution operators for pattern matching and conduct pulse calculation and concatenation to obtain the pulse sequence for the circuit. The method has good scalability, making QPulseLib useful for large-scale circuit synthesis. Moreover, SmartQCache proposes a novel pulse control architecture with a two-level cache for pulse synthesis acceleration. We also develop a pulse optimization scheme, including duration optimization, pulse calibration, and time shift correction. The method of SmartQCache exhibits significant acceleration in pulse synthesis while keeping high pulse accuracy, benefiting quantum pulse control system design.

REFERENCES

[1] T. Alexander et al., "Qiskit pulse: Programming quantum computers through the cloud with pulses," *Quantum Sci. Technol.*, vol. 5, no. 4, 2020, Art. no. 44006.
[2] X. Fu et al., "An experimental microarchitecture for a superconducting quantum processor," in *Proc. 50th Annu. IEEE/ACM MICRO*, 2017, pp. 813–825.
[3] J. Cheng, H. Deng, and X. Qia, "AccQOC: Accelerating quantum optimal control based pulse generation," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2020, pp. 543–555.
[4] H. R. Grimsley, S. E. Economou, E. Barnes, and N. J. Mayhall, "An adaptive variational algorithm for exact molecular simulations on a quantum computer," *Nat. Commun.*, vol. 10, no. 1, p. 3007, 2019.
[5] Q. Guo et al., "Observation of energy-resolved many-body localization," *Nature Phys.*, vol. 17, no. 2, pp. 234–239, 2021.
[6] Y. Yao et al., "Observation of many-body Fock space dynamics in two dimensions," *Nature Phys.*, vol. 19, pp. 1459–1465, Jul. 2023.
[7] C. Portmann and R. Renner, "Security in quantum cryptography," *Rev. Modern Phys.*, vol. 94, no. 2, 2022, Art. no. 25008.
[8] Z. Hu, Y. Lin, Q. Guan, and W. Jiang, "Battle against fluctuating quantum noise: Compression-aided framework to enable robust quantum neural network," in *Proc. 60th ACM/IEEE DAC*, 2023, pp. 1–6.
[9] C. Neill et al., "A blueprint for demonstrating quantum supremacy with superconducting qubits," *Science*, vol. 360, no. 6385, pp. 195–199, 2018.
[10] F. Arute et al., "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.
[11] M. Zhang et al., "Exploiting different levels of parallelism in the quantum control microarchitecture for superconducting qubits," in *Proc. 54th Annu. IEEE/ACM MICRO*, 2021, pp. 898–911.
[12] G. H. Low and I. L. Chuang, "Optimal hamiltonian simulation by quantum signal processing," *Phys. Rev. Lett.*, vol. 118, no. 1, 2017, Art. no. 10501.
[13] Y. Ruan, X. Xue, H. Liu, J. Tan, and X. Li, "Quantum algorithm for K-nearest neighbors classification based on the metric of hamming distance," *Int. J. Theor. Phys.*, vol. 56, pp. 3496–3507, Nov. 2017.
[14] W. Tian, X. Jia, S. Tan, Z. Song, L. Lu, and J. Yin, "QPulseLib: Accelerating the pulse generation of quantum circuit with reusable patterns," in *Proc. 42th IEEE/ACM ICCAD*, 2023, pp. 1–7.
[15] T. Patel and D. Tiwari, "DisQ: A novel quantum output state classification method on IBM quantum computers using OpenPulse," in *Proc. 39th IEEE/ACM ICCAD*, 2020, pp. 1–9.
[16] Y. Chen et al., "A pulse generation framework with augmented program-aware basis gates and criticality analysis," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2023, pp. 773–786.

[17] H. Wang, Z. Li, J. Gu, Y. Ding, D. Z. Pan, and S. Han, "QOC: Quantum on-chip training with parameter shift and gradient pruning," in *Proc. 59th ACM/IEEE DAC*, 2022, pp. 655–660.

[18] Z. Hu, P. Dong, Z. Wang, Y. Lin, Y. Wang, and W. Jiang, "Quantum neural network compression," in *Proc. 41th IEEE/ACM ICCAD*, 2022, pp. 1–9.

[19] B. K. Chakrabarti, A. Dutta, and P. Sen, *Quantum Ising Phases and Transitions in Transverse Ising Models*, vol. 41. Berlin, Germany: Springer, 2008.

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

[21] D. C. McKay, C. J. Wood, S. Sheldon, J. M. Chow, and J. M. Gambetta, "Efficient *Z* gates for quantum computing," *Phys. Rev. A*, vol. 96, no. 2, 2017, Art. no. 22330.

[22] L. Hales and S. Hallgren, "An improved quantum fourier transform algorithm and applications," in *Proc. 41st Annu. Symp. Found. Comput. Sci. (FOCS)*, 2000, pp. 515–525.

[23] S. Bagherinezhad and V. Karimipour, "Quantum secret sharing based on reusable Greenberger-Horne-Zeilinger states as secure carriers," *Phys. Rev. A*, vol. 67, no. 4, 2003, Art. no. 44302.

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

[25] E. Brainis, L.-P. Lamoureux, N. Cerf, P. Emplit, M. Haelterman, and S. Massar, "Fiber-optics implementation of the Deutsch-Jozsa and Bernstein-Vazirani quantum algorithms with three qubits," *Phys. Rev. Lett.*, vol. 90, no. 15, 2003, Art. no. 157902.

[26] D. Collins, K. Kim, and W. Holton, "Deutsch-Jozsa algorithm as a test of quantum computation," *Phys. Rev. A*, vol. 58, no. 3, 1998, Art. no. R1633.

[27] R. Mengoni and A. Di Pierro, "Kernel methods in quantum machine learning," *Quantum Mach. Intell.*, vol. 1, nos. 3–4, pp. 65–71, 2019.

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

[29] R. Laflamme, C. Miquel, J. P. Paz, and W. H. Zurek, "Perfect quantum error correcting code," *Phys. Rev. Lett.*, vol. 77, no. 1, p. 198, 1996.

[30] M. S. Islam, M. M. Rahman, Z. Begum, and M. Z. Hafiz, "Low cost quantum realization of reversible multiplier circuit," *Inf. Technol. J.*, vol. 8, no. 2, pp. 208–213, 2009.

[31] A. Cabello, "Bell's theorem with and without inequalities for the three-qubit Greenberger-Horne-Zeilinger and W states," *Phys. Rev. A*, vol. 65, no. 3, 2002, Art. no. 32108.

[32] P. W. Shor, "Introduction to quantum algorithms," in *Proc. Symposia Appl. Math.*, 2002, pp. 143–160.

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

[34] M. Cho and D. Brand, "MEC: Memory-efficient convolution for deep neural network," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 815–824.

[35] A. Xygkis, L. Papadopoulos, D. Moloney, D. Soudris, and S. Yous, "Efficient Winograd-based convolution kernel implementation on edge devices," in *Proc. 55th Annu. Design Autom. Conf. (DAC)*, 2018, pp. 1–6.

[36] "Qiskit Transpiler, Qiskit development team." IBM & Co. LLC. 2023. [Online]. Available: https://qiskit.org/documentation/apidoc/transpiler.html#transpiler-api

[37] S. Krinner et al., "Realizing repeated quantum error correction in a distance-three surface code," *Nature*, vol. 605, no. 7911, pp. 669–674, 2022.

[38] "Rigetti systems: Aspen-M-3 quantum processor," Rigetti & Co. LLC. 2019. [Online]. Available: https://pyquil-docs.rigetti.com/en/v3.5.4/apidocs/pyquil.quiltcalibrations.html

[39] "pyQuil document v3.5.4: Pulses and waveforms." Rigetti & Co. LLC. 2019. [Online]. Available: https://pyquil-docs.rigetti.com/en/v3.5.4/quilt_waveforms.html#Compile-Time-versus-Run-Time

[40] A. Cross, "The IBM Q experience and QISKit open-source quantum computing software," presented at the APS March Meeting Abstracts, 2018.

[41] A. Mills, D. Zajac, M. Gullans, F. J. Schupp, T. M. Hazard, and J. R. Petta, "Shuttling a single charge across a one-dimensional array of silicon quantum dots," *Nat. Commun.*, vol. 10, no. 1, p. 1063, 2019.

[42] J. R. Johansson, P. D. Nation, and F. Nori, "QuTiP: An open-source Python framework for the dynamics of open quantum systems," *Comput. Phys. Commun.*, vol. 183, no. 8, pp. 1760–1772, 2012.

[43] Z. Liang et al., "Hybrid gate-pulse model for variational quantum algorithms," in *Proc. 60th ACM/IEEE DAC*, 2023, pp. 1–6.

Liqiang Lu received the Ph.D. degree in computer science from Peking University (PKU), Beijing, China, in 2022.

He is a ZJU100 Young Professor with the College of Computer Science, Zhejiang University, Hangzhou, China.

He has authored more than 20 scientific publications in premier international journals and conferences in related domains, including ISCA, MICRO, HPCA, ASPLOS, FCCM, DAC, IEEE MICRO, and IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. His research interests include quantum computing, computer architecture, deep learning accelerator, and software-hardware codesign.

Mr. Lu also serves as a TPC Member in the premier conferences in the related domain, including ICCAD, FPT, and HPCC.

Wuwei Tian is currently pursuing the Ph.D. degree with the College of Computer Science, Zhejiang University, Hangzhou, China.

He has authored a scientific publication in premier conferences in related domains, including ICCAD 2023. His research interests include quantum pulse control systems and quantum software–hardware codesign.

Xinghui Jia received the bachelor's degree in computer science and technology from Sichuan University, Chengdu, China, in 2022. He is currently pursuing the master's degree with the College of computer science, Zhejiang University, Hangzhou, China.

He is interested in quantum computing and software systems.

Zixuan Song received the master's degree in physics from Zhejiang University, Hangzhou, China, in 2021.

She is currently an Engineer with the ZJU-Hangzhou Global Scientific and Technological Innovation Center, Hangzhou. She has authored or co-authored over six SCI papers, including *Nature Physics* and *Physical Review Letters*. She is engaged in research related to superconducting quantum computing.

Siwei Tan is currently pursuing the Ph.D. degree with the College of Computer Science, Zhejiang University, Hangzhou, China.

He has published papers in conferences and journals, including HPCA, MICRO, ASPLOS, VIS, TVCG, and SCC. His research interests are quantum algorithms, computer architecture, and computer systems.

Jianwei Yin (Member, IEEE) received the Ph.D. degree in computer science from Zhejiang University (ZJU), Hangzhou, China, in 2001.

He was a Visiting Scholar with the Georgia Institute of Technology, Atlanta, GA, USA. He is currently a Full Professor with the College of Computer Science, ZJU. He has published more than 100 papers in top international journals and conferences. His current research interests include quantum computing, service computing, and business process management.

Prof. Yin is an Associate Editor of the IEEE TRANSACTIONS ON SERVICES COMPUTING.