AccQOC: Accelerating Quantum Optimal Control Based Pulse Generation

Jinglei Cheng
University of Southern California
chen520@usc.edu

Haoqing Deng
University of Southern California
haoqinde@usc.edu

Xuehai Qian
University of Southern California
xuehai.qian@usc.edu

Abstract—In the last decades, we have witnessed the rapid growth of Quantum Computing. In the current Noisy Intermediate-Scale Quantum (NISQ) era, the capability of a quantum machine is limited by the decoherence time, gate fidelity and the number of Qubits. Current quantum computing applications are far from the real "quantum supremacy" due to the fragile physical Qubits, which can only be entangled for a few microseconds. Recent works use quantum optimal control to reduce the latency of quantum circuits, thereby effectively increasing quantum volume. However, the key challenge of this technique is the large overhead due to long compilation time.

In this paper, we propose *AccQOC*, a comprehensive static/dynamic hybrid workflow to transform gate groups (equivalent to matrices) to pulses using QOC (Quantum Optimal Control) with a reasonable compilation time budget. AccQOC is composed of static pre-compilation and accelerated dynamic compilation. After the quantum program is mapped to the quantum circuit with our heuristic mapping algorithm considering crosstalk, we leverage static pre-compilation to generate pulses for the frequently used groups to eliminate the dynamic compilation time for them. The pulse is generated using QOC with binary search to determine the latency. For a new program, we use the same policy to generate groups, thus avoid incurring overhead for the "covered" groups. The dynamic compilation deals with "un-covered" groups with accelerated pulse generation. The key insight is that the pulse of a group can be generated faster based on the generated pulse of a *similar* group. We propose to reduce the compilation time by generating an ordered sequence of groups in which the sum of similarity among consecutive groups in the sequence is minimized. We can find the sequence by constructing a similarity graph — a complete graph in which each vertex is a gate group and the weight of an edge is the similarity between the two groups it connects, then construct a Minimum Spanning Tree (MST) for SG. With the methodology of AccQOC, we reached a balanced point of compilation time and overall latency. The results show that accelerated compilation based on MST achieves $9.88\times$ compilation speedup compared to the standard compilation of each group while maintaining an average $2.43\times$ latency reduction compared with gate-based compilation.

Index Terms—quantum computing; quantum optimal control; pre-compilation;

I. Introduction

The idea of Quantum Computer [2] has been proposed for decades. In recent years, we have witnessed many breakthroughs in building a quantum machine [7], [10], [19]. IBM [21] and Google [16] have demonstrated working quantum machine with 50 bits and 72 bits. Google claims to have built the first quantum computer that can carry out calculations beyond the ability of today's most powerful supercomputers [4]. This indicates that we have entered the era of Noisy Intermediate-Scale Quantum(NISQ) [33], where we expect to see a quantum machine with hundreds or thousands of Qubits outperforming classical supercomputers in the coming decades. However, the Qubits inside a NISQ device are far from perfect. First, the connection between Qubits is sparse. Second, the operations of Qubits are vulnerable to errors due to insufficient decoherence time. The sparsity of Qubits means that 2-bit operations are not supported for all Qubit pairs. Furthermore, the limited number of Qubits makes it unrealistic to implement quantum error correction code(ECC) [6], [31], [38], which would need thousands of Qubits. Therefore, current quantum computers suffer from high gate error rates. Consequently, large-scale programs like Shor algorithm [36] or Grover Search algorithm [14] could not be implemented on NISQ devices.

In the NISQ era, the capacity of a quantum machine is limited by its *quantum volume* — determined by the number of Qubits and the decoherence time — and gate error. For current IBM's quantum devices, the longest decoherence time achieved is less than one hundred microseconds [1]. Thus, the reduction of latency is critical to NISQ devices for two reasons. First, we could run larger programs within the same decoherence time on a given quantum machine. Second, for the current quantum programs, the coherence error can decrease substantially by latency reduction [30], [35]. Based on our calculations in Section II-E, the coherence error is *comparable* to gate error. Thus, the fidelity, affected by both coherence and gate error, can be improved by latency reduction.

The current compilation for quantum system is gate-based [3] — quantum programs are first compiled into specific 1-bit or 2-bit operations that the quantum machine supports. This step is often referred to as synthesis [34]. These hardware-supported gates will be further translated into corresponding electrical signals, often referred to as *pulses*. The gate-based compilation method requires little compilation time but leads to longer latency, because the synthesis turns the desired unitary matrix into multiple instructions whose corresponding pulses will be concatenated to reach the target function.

To mitigate these drawbacks, physicists proposed quantum optimal control (QOC) [12], which could directly compile quantum state transfer or a functional unitary matrix that emulates the operation performed by a group of gates into control pulses. There exist multiple algorithms developed for quantum optimal control, such as gradient ascent pulse

engineering (GRAPE) [9] and Krotov algorithms [22], most of which are based on gradient methods. The pulses control the unitary matrix that represents the function of a quantum circuit in a systematic manner. The evolving path towards the final states could be divided into small steps. By optimizing these steps through analytical or numerical algorithms, the quantum optimal control can generate a sequence of control pulses that could approximate the target matrix [12].

Quantum optimal control could effectively reduce the latency of groups of quantum gates. Since the size of the unitary matrix that represents the function of quantum gates does not increase with the number of gates, the latency of pulses generated by QOC based compilation does not scale with the number of gates as much as the gate-based compilation does. Recent work [35] has utilized this method to reduce the latency of the aggregated gates. Together with proper gate scheduling, it could bring considerable latency reduction.

On the other side, this method has the inevitable drawback of requiring enormous computational resources. Currently GRAPE only supports 10 Qubit for very simple operations (such as 10 concurrent single-bit operations). For groups composed of more than 5 bits, we observe that it could take several hours to generate the corresponding pulses even for a simple Qubit model. As a result, it is impossible to compile a quantum program with hundreds of thousands of gates within a day. To mitigate this problem, [25] uses automatic differentiation with the GPU to accelerate the quantum optimal control. However, such acceleration method only achieves significant speedup with utilization of GPU when more than 10 Qubits are involved. The acceleration for small groups is limited. [35] also points out the large compilation overhead caused by QOC.

Recent work [13] addresses this problem for quantum variational algorithms such as VQE and QAOA with partial compilation. The variational quantum programs are executed iteratively, the parameters (mostly rotation angle) of the current iteration are determined by the output distribution of the previous iteration. For this specific type of algorithms, pre-computation can be applied to obtain the proper hyperparameters for compilation that can accelerate pulse generation of groups with different rotation angles. This method reduces the compilation time of variational algorithms thanks to its iterative nature: for a given group, it is "parameterized" with only the rotation angles for different iterations, and the selected hyparparameters can accelerate the pulse generation of all concrete groups with rotation angles determined. To ease the discussion, we call the concrete groups determined by the same parameterized group as a *group family*. This methodology does *not* work for non-variational algorithms such as Shor algorithm [36] for the following reason. For these algorithms, the program does not change during execution. Therefore, the execution does not involve groups with changing parameters. The gates implementing the whole algorithm can be only decomposed into *static groups* which *do not belong to the same group family*. If we use the method of [13], then the hyparparameters of *each* static group need to be generated *but*

not reused. Clearly, [13] does not help the pulse generation of static groups. Moreover, the recent works that optimize the QOC based compilation do not consider its scalability, making it not realistic to compile large quantum programs.

As the first attempt to accelerate the pulse generation of static groups, we propose *AccQOC*, a comprehensive static/dynamic hybrid workflow to transform gate groups (equivalent to matrices) to pulses using QOC with reasonable compilation time budget. AccQOC is composed of static precompilation and accelerated dynamic compilation. First, the quantum program is mapped to the quantum computer with our heuristic mapping algorithm. Our mapping algorithm takes into consideration the cross-talk effect. We aim to increase fidelity through the mitigation of cross-talk effect. Then, we leverage pre-compilation to generate pulses for a category of groups to eliminate the dynamic compilation time for them. To get this category of groups, we perform profiling on a randomly selected set of programs with certain grouping policy. The corresponding pulses are then generated using QOC with binary search to determine the shortest possible latency. Given a target program, it is first decomposed with the same policy used in pre-compilation. For the gate groups of which the pulse is available, there is no compilation cost. We only need to focus on the groups that are not "covered". When pulses of all groups are generated, they are concatenated together to determine the overall latency of the target quantum program. Since different grouping policy will lead to different overall latency, we performed six different policies on some sample programs and pick the policy of best performance.

The second component of AccQOC deals with the dynamic compilation of "un-covered" groups with accelerated pulse generation. In fact, the technique applies to both those "uncovered" groups as well as the static pre-compilation (but it is a one time cost). The key insight is that the pulse of a group can be generated faster based on the generated pulse of a *similar* group. Thus, we can reduce the compilation time by generating an ordered sequence of groups in which *the sum of similarity among consecutive groups in the sequence is minimized*. We can find the sequence by 1) constructing *similarity graph or SG* — a complete graph in which each vertex is a gate group and the weight of an edge is the similarity between the two groups it connects; and 2) constructing a Minimum Spanning Tree (MST) for SG. We will include the identity matrix (recall that a group corresponds to a matrix) as a vertex, if no group is similar enough, the compilation will start from the pulse of identity matrix.

With this workflow, we are able to reach a balance between compilation time and latency reduction. By using "map2b4l" strategy described in section IV-B, We achieve an average of $2.43\times$ reduction of overall latency and $9.88\times$ reduction of compilation time. AccQOC is more general than the method in [13], and can handle the parameterized groups and static groups in the same manner. Specifically, for the variational algorithms, AccQOC will treat the groups with different rotation angles simply as different static groups and accelerate the pulse generation by keeping previously generated pulses

and selecting the most similar group's pulse as the initial condition. Thus, AccQOC can generate new groups with arbitrary rotation angles — our method does not use it as a parameter. As explained before, the fidelity is affected by both coherence error and gate error, the reduction of latency by QOC mainly affects coherence error. In Section II-E, we demonstrate with calculation that the two sources of errors are comparable. Thus, reducing latency is an important way to improve fidelity.

The rest of the paper is organized as follows: section II gives the basic background of Quantum Computing and states our main motivation; section III presents an overview of our methodology; section IV describes in detail how pre-compilation works to reduce overall latency; section V describes further optimization to reduce pre-compilation time; section VI shows our results of pre-compilation and further optimization; we relate previous work in section VII and summarize our work in section VIII.

II. BACKGROUND

This section presents the necessary background of quantum computing and quantum optimal control.

A. Basics of Quantum Bit

The essential difference between quantum computing and classical computing lies in the property of Quantum Bit (Qubits) [30]. A Qubit has an infinite number of states which are different superposition of logical states 0 and 1, rather than only two logical states as in classical bits. The state of one Qubit could be generally represented as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where a and b are complex numbers satisfying "$|\alpha|^2 + |\beta|^2 = 1$". The matrix that represents the state of this one bit is $|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$. When this Qubit is measured on a basis of 0/1, the quantum states collapse, and the probability of measured results being 0 and 1 is $|\alpha|^2$ and $|\beta|^2$ respectively. Similarly a quantum system of two Qubits have four orthogonal basis, and the quantum state could be represented as "$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$". When measured, the quantum state would have probabilities of $|\alpha|^2$, $|\beta|^2$, $|\gamma|^2$ and $|\delta|^2$ being 00,01,10,11 respectively. Therefore, a quantum system with N Qubits would have 2^N quantum states and needs 2^N complex parameters to describe. The exponential feature of quantum states gives quantum computers the ability to solve problems that are intractable to classical computers. It also leads to the fundamental difficulty in simulating a quantum system.

B. Basics of Quantum Computing

The basic element of a quantum program is the quantum gate. These gates have different functions and could be represented by different unitary matrices. An N-bit quantum gate could be represented by a $2^N \times 2^N$ matrix. The function of multiple gates can be computed by simply multiplying matrices of the individual gate, respectively. For instance, the state transfer function of a 10-bit quantum system with 5
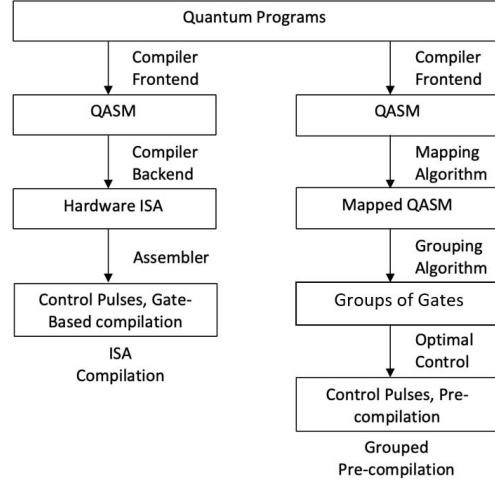

Fig. 1: Comparison between two compilation methodologies. The left describes classic gate-based compilation approach [35]. Our approach on the right utilizes pre-compilation.

gates could be computed by multiplying 5 matrices of size 1024×1024.

In the control flow of running a quantum program, the quantum algorithm is first synthesized into several discrete basic quantum logical gates. However, these gates may not match the basic gates supported directly on quantum machines. For example, a Toffoli gate is commonly used in quantum algorithms. However, it could not be supported on quantum hardware [11]. So it will first be decomposed into smaller gates, then translated to control pulses.
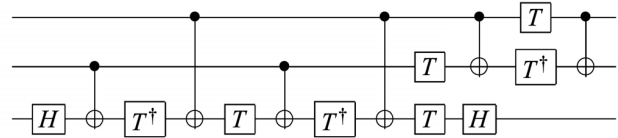

Fig. 2: The Toffli gate often used in quantum programs is not directly supported by a quantum computer, thus it must be decomposed into basic gates that are compatible with quantum hardware.

The second step is to map Qubits to physical bits residing on the actual machine. Since the quantum program usually can not be directly executed on certain quantum machines due to the sparse connection between quantum bits, swap gates are needed to be inserted to make the quantum program compatible with the hardware. This step could be referred to as the Qubit mapping problem. The mapping algorithms have been discussed in detail [26], [40], [43]. Since it is essentially an NP problem, the mapping algorithm remains possible to be further optimized.

After the device-dependent program is generated, the gates need to be scheduled and translated to pulses, then they can be executed on quantum machines. Finally the program is run multiple times, and the result is the distribution of the output.
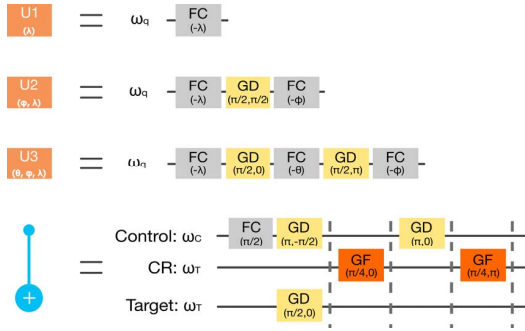
C. Gate-based Compilation

Fig. 3: Each gate corresponds to a pulse [3]. One-by-one concatenation of all pulses realize the function of many gates.

The method to translate gates to pulses is gate-based compilation. This method uses a gate-pulse look-up table to find the corresponding pulse for the gate, then concatenates all pulses together to form the final pulses for the whole program. For example, current IBM quantum machines support u1,u2,u3, and CNOT gates, which contain rotation operations and common 2-bit gate. Their corresponding pulses are also given in Figure 3. The category of hardware-supported gates will decide how quantum programs are decomposed. There exist mismatches between basic gates of quantum programs and basic gates supported by hardware. The Toffoli gate mentioned above is a perfect example. The 3-bit gate is not directly supported by current quantum machines. So it must first be decomposed into 15 basic gates, which are then directly translated into corresponding pulses. For gates with parameters, like rotation gates, different rotation angles determine the strength and the length of a certain pulse. Gate-based compilation requires minimum compilation time, but it is inefficient compared with QOC based compilation in terms of pulse latency.

D. Quantum Optimal Control and GRAPE

Quantum optimal control [12] takes advantage of a feature of quantum machines — states of the Qubits could be manipulated with the unique and time-dependent Hamiltonian matrix [25], [32], [39]. Among many algorithms that are based on gradient descent, we choose one implementation of GRAPE [20] as our tool of quantum optimal control. To approach the target unitary matrix, the tool uses a numerical method to evolve from initial condition to the final state. The control pulses are divided into small steps. By gradually modifying theses small steps with enough iterations, the output fidelity will increase to an acceptable value. The procedure is similar to the training algorithm of neural networks. The cost functions (typically fidelity) could be chosen manually to emphasize the expected characteristics of output pulses. Since the path

from an initial condition to target states is achieved through evolution, it is not unique. How to find the optimal control for a given target unitary matrix is still an open problem.

E. Fidelity: Coherence vs. Gate Error

In this section, we analyze the relative importance of coherence error and gate error for different technologies. For superconducting qubits, the ratio of gate times to decoherence times is relatively high, so even after a few gates, we do experience the exponential decay due to decoherence. For example, the average relaxation and coherence times for Melbourne qubits are $T1 = 57.35\mu s$ and $T2 = 61.82\mu s$ [1] , the time needed for a CX gate is approximately $974.9ns$ [5]. The error caused by decoherence in the $974.9ns$ is $1 - e^{-0.9749/57.35} = 1.69 \times 10^{-2}$. Such error is in the same order and comparable with the average CX gate error of 2.46×10^{-2}. The calculation shows that latency reduction can indeed improve overall fidelity for superconducting quantum computers where error caused by qubit's decoherence is relatively high. For trapped ion system, decoherence times are extremely long relative to gate times, so the reduced latency may not improve fidelity as much in this scenario, and just leads to faster time-to-solution. Moreover, [35] shows that shorter pulses generated by QOC have simpler shape than those generated by gate-based concatenation and are easier to implement.

F. Cross-talk

Cross-talk is an important source of noise in quantum computers. When instructions are executed in parallel, the cross-talk effect will substantially reduce the fidelity of these instructions [27]. Cross-talk comes from leakage of control signals, and the control signals of one instruction could interact with the control signals of parallel instructions. The strength of the interaction is affected by the physical distance of bits these parallel instructions operate on. Crosstalk noise is prevalent across many of the leading qubits such as superconducting [] and trapped ion qubits [8], [15].

G. Motivation

Two recent works [13], [35] are also closely related to QOC and both have the notion of grouping. Here we elaborate the differences by group examples shown in Figure 14. Figure 4a and 4b show two groups in the same "group family" (defined in Section I) for variational algorithms, we can see that the group structures are the same and the only difference is the rotation angles ($\theta_1 = \pi/4$ vs. $\theta_2 = \pi/3$). [13] generates the hyparparameters (e.g. learning rate) by pre-computation, which can accelerate the pulse generation of groups in the same family. In comparison, the goal of AccQOC is to accelerate the pulse generation for *static groups* after decomposition as shown in Figure 4d. We can see that these groups can be completely different, instead of just having different rotation angles. Therefore, they clearly belong to the different group family with different hyparparameters. Also, each group just needs to be compiled *once*. This explains why the method of [13] does not apply to non-variational algorithms.

(a) Group with rotation $\theta_1 = \pi/4$ (b) Group with rotation $\theta_2 = \pi/3$
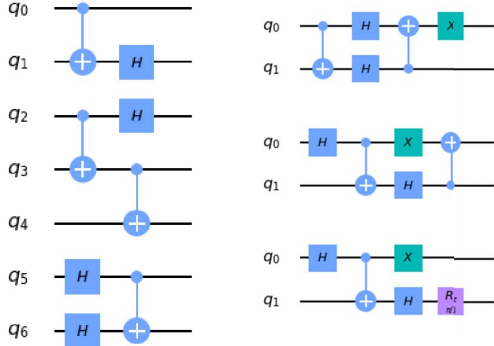
(c) Group with many qubits

(d) Typical group in our paper

Fig. 4: Groups generated by our method compared to recent work [13], [35]. We limit our group size to ensure fast compilation and good coverage. The groups generated by [35] could be too large and not quite scalable. The grouping algorithm proposed by [13] can only benefit variational algortihm.
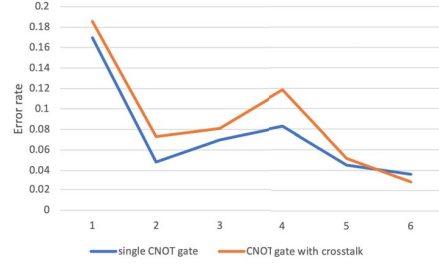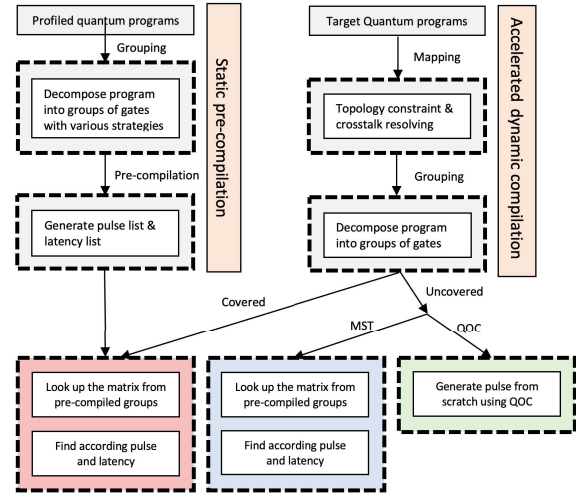

Fig. 5: Crosstalk and Error Rate.

Fig. 6: Our method utilizes pre-compilation to generate a group list and a pulse list for future reuse, so that they do not need to undergo the time-consuming compilation process again. For groups that are "uncovered" by pre-compilation, we construct the MST to accelerate the computation of their pulses.

Compared to the groups of [35] shown in Figure 4c, the groups of AccQOC is much smaller. This is due to the different goals. The goal of [35] is to achieve higher parallelism and minimize the latency of pulses. Specifically, it finds the commutative gates that provide more flexibility in scheduling, i.e., execute the gates in the alternative order. Then, Commutativity-aware Logical Scheduling (CLS) attempts to schedule many gates to achieve high parallelism and thus reduce latency. As the result, the group size tends to be large. Without manually limiting the number of qubits or layers, the aggregation methodology discussed in [35] would generate groups with up to 10 qubits. It is costly to generate pulses for groups of this size. The goal of AccQOC is to accelerate pulse generation with the central idea of pre-compilation and group similarity. To ensure a good coverage, the group size needs to be relatively small (such as the examples in Figure 4d). Moreover, a large group size will lead to many possible groups (matrices) of that size, making it hard to take advantage of group similarity.

From the discussion, we see that QOC can considerably reduce latency, but with a huge computation overhead. The prior work [13] addresses this problem for variational algorithms, but the long compilation time for non-variational algorithms is still an *open problem*. This paper makes the first attempt to accelerate *pulse generation for static groups* after decomposition based on certain policy. Specifically, our solution leverages pre-compilation and gate group similarity to accelerate the pulse generation with QOC. With pulses generated for groups, the latency of a given program can be greatly reduced. Nevertheless, our solution is a step toward faster pulse generation but has not yet fully addressed the problem of QOC's large overhead of compilation time, which scales with the size of the input quantum program.

Moreover, considering the two extreme policies of dividing the program into many one-gate groups and into one many-gates large group: the latter could achieve substantially more reduction in latency than the former with huge compilation overhead. However, the two extremes tell us little about what is in between. In essence, our solution finds a balanced point between the compilation time and the latency reduction.

III. AccQOC Overview

Figure 6 explains the overview of AccQOC and highlights the main contributions compared with standard gate-based compilation. Our framework shares the same part of compilation front-end with gate-based compilation. But for the part of the translation from quantum gates to control pulses, we use pre-compilation method explained in Section IV rather than a gate-based compilation. The first component is the *static pre-compilation*. A category of groups of gates, whose size is determined by parameters of different decomposition policies, is first generated through randomly profiling a set of quantum programs inside our benchmark. We use GRAPE [32] to generate the pulse list and latency list from the category of groups, respectively. The pulse generation is based on QOC with binary search to determine the latency. To generate the overall latency, dynamic programming is used solve the gate dependency issue and concatenate pulses of all groups together. Different decomposition policies are compared against each other in terms of their corresponding overall latency, and we choose the one that produces the lowest latency. Based on this policy, the pulses of the most frequent groups will be re-generated with different parameter settings to gain better latency. In summary, this step pre-compiles pulses for a set of gate groups.

The second component is the *accelerated dynamic compilation*. Given a new quantum program, we first decompose it into groups using the chose policy above. Some groups will be "covered" by the pre-compiled pulses, and they can be directly used. For the "un-covered" groups, we generate an ordered sequence from the complete similarity graph (ST) with the similarity between a pair of groups as the weight of each edge. This sequence minimizes *the sum of similarity among consecutive groups in the sequence* and can be generated by computing the Minimum Spanning Tree (MST) of SG. After the pulse of all groups are obtained, the pulses for the whole program is concatenated in a similar manner as in pre-compilation based on DAG with dynamic programming.

Overall, AccQOC accelerates the pulse generation based on QOC with static pre-compilation and accelerates dynamic compiling due to similarity between groups.

IV. Static Pre-Compilation

A. Cross-talk Consideration

We take the cross-talk effect into account in the mapping process, because pulses are generated from hardware-dependent quantum programs after swap insertion. When inserting swap gates to make the quantum program compatible with the hardware topology, we utilize the mapping tool developed by [43]. This method uses an A* search with a heuristic function to find the mapping.

We extend the heuristic function to take the cross-talk effect into consideration. The extended heuristic function is:

$$h(\sigma_i^j) = \sum_{g \in l_i} h(g, \sigma_i^j) + \sum_{g_m, g_n \in l_i} I_{g_m, g_n}$$

Where the $h(\sigma_i^j)$ is the function that represents the heuristic cost of a layer l_i's mapping σ_i^j. And the function $h(g, \sigma_i^j)$ calculates the physical distance between the two Qubits of gate g. The extension we add is the indicator function of g_m and g_n which will be 1 if these two gates are too close.

Figure 5 shows the cross-talk effect of real quantum computer IBM Q Melbourne. The x-axis indicates different Qubit pairs and y-axis shows error rate. Different Qubit pairs have differnet error rate. In general, the lower the curve, the higher fidelity is achieved. The lower curve shows the results for the single CNOT gate without cross-talk. The higher curve represents the fidelity of a CNOT gate operating on the same pair of Qubits but affected by a nearby CNOT gate. We see that these six Qubits pairs suffer from average 20% higher error rate due to the effect of cross-talk. To evaluate the effects of cross-talk consideration as a part of the mapping algorithm, we quantify the total cross-talk effect as the sum of occurrences of close CNOT pairs in each layer. This metric is adopted from the paper [29]. The qubits are dispersively coupled on quantum devices, which means that the cross-talk effect is much stronger on closer qubits. This metric also makes characterizing crosstalk more efficient [29]. As the results shown in Figure 11 in Section VI-C, we are able to achieve 17.6% reduction of the total cross-talk effect.

B. Grouping policy

We try several policies of generating gate groups based on different values of n in the 2bnl cataloging system, where 2 represents the maximum number of Qubits in a gate group and n the number of layers. We limit the maximum number of Qubits to be 2 because a group composed of more than 2 Qubits takes too much time to train with QOC, which violates our goal of reducing compilation time. We choose 2b2l, 2b3l, 2b4l as candidate policies, for their relatively small size makes it realistic to profile.

There are two ways of dealing with swap operations generated during the mapping process: treating swap as an independent operation or decomposing swap into three CNOT gates. Since different quantum machines may implement swap operations in both ways, we experiment with both methods, and differentiate them as "map" and "swap". For each method, we experiment with the 3 candidate policies mentioned above.

Different policies	# Qubits	# Layers
swap / map	2	2
swap / map	2	3
swap / map	2	4

TABLE I: The parameter settings of our 6 policies

In summary, there are a total of 6 candidate grouping policies, and we label them as "map2b2l", "map2b3l", "map2b4l", "swap2b2l", "swap2b3l", "swap2b4l". And the benchmarks we use to evaluate these policies cover lots of functions used in existing quantum algorithms such as QFT and algorithmic functions in classical computing.

C. Generating Group List

The grouping process is divided into two steps. First, we partition gates into subgroups based on 2-bit constraint. Second, we partition each subgroup into smaller groups based on n-layer restriction. The two steps are illustrated in Algorithm 1 and Algorithm 2, respectively.

In the bit partition step, we first transform a quantum program into a Directed Acyclic Graph (DAG). We iterate through the DAG following its topological order to ensure that a node always finds its group after its predecessor does. This way, we are able to greedily group a node with its parent nodes whenever possible, thereby dividing DAG into subgroups of largest possible size. In the layer partition step, we first label each node with its global depth, then divide nodes within each subgroup into smaller groups based on this labeled depths.

After dividing a quantum program into groups, we "de-duplicate" these groups by calculating their corresponding matrices and eliminating duplicated ones. Two groups with permutated Qubits but same operations are also treated as duplicate.

In the static pre-compilation stage, we randomly select one-third of quantum programs from our set of benchmarks. We use these programs to generate a category of groups for the profiling purpose. The corresponding pulses of these groups will be stored and reused in future quantum programs.

D. Generating Latency List and Pulse List

We use QOC to generate the pulse list and latency list from the category of groups. The GRAPE tool we use requires target unitary matrix, target fidelity, and target latency as inputs. The latency of a certain group is determined by a binary search. Short latency leads to more iterations with long training time

---

Algorithm 1 Bit Dividing

Require: qasm files, bit constraint(bc)
Ensure: large-groups
 1: Initialize large-groups
 2: **for** qasm in all qasm files **do**
 3:     DAG = ToDAG(qasm)
 4:     **for** node in DAG.topological-order: **do**
 5:         **if** node can be grouped with both predecessor **then**
 6:             Merge the groups the two predecessors are in
 7:         **else if** node can be grouped with one predecessor **then**
 8:             Group the node with the predecessor
 9:             Update large-groups
10:         **else if** node can be group with no predecessor **then**
11:             Put the node in a new group
12:             Update large-groups
13:         **end if**
14:     **end for**
15: **end for**
16: **return** large-groups

Algorithm 2 Layer Dividing

Require: large-groups, layer constraint(lc)
Ensure: group list
 1: Initialize group-list
 2: **for** node in DAG.topological-order: **do**
 3:     Depth[node] = max(Depth[node's predecessor(s)]) + 1
 4: **end for**
 5: **for** subgroup in large-groups: **do**
 6:     startDepth = depth of shallowest node
 7:     layer = 0
 8:     Initialize temp-group
 9:     **for** node in subgroup: **do**
10:         diff = depth[node] - start
11:         **if** diff mod lc ≤ layer **then**
12:             Append node to temp-group
13:         **else**
14:             Append temp-group to group-list
15:             Clear temp-group
16:             Append node to temp-group
17:             layer += 1
18:         **end if**
19:     **end for**
20: **end for**
21: **return** group-list

---

and does not guarantee the convergence, while long latency loses the advantages of quantum optimal control. Therefore, binary search is necessary to ensure optimal latency within the target fidelity convergence requirement. We set the target fidelity cost function to be a typical value 1×10^{-4} and maximum run time budget to be 600s for each iteration of binary search. The available methods include ADAM, BFGS, L-BFGS-B, and SLSQP. We choose BFGS as our optimization method for training the pulses. To verify our idea, we use a model of a two-level spin Qubit($\omega/2\pi$: 3.9 GHz).

E. Generating Overall Latency

After generating a group list, we restructure the original DAG into a new DAG by turning each group into a node. We obtain the latency of each node by iterating through the profiling table to find its match.

Following the topological order of the new DAG, we use dynamic programming to compute and store the until-this-step latency at each node by adding the largest latency of its predecessors to the latency of itself. The overall latency computed at the last node is the overall latency of the whole group. The detailed algorithm could be found at Algorithm 3.

F. The Effect of Sequence of Mapping and Grouping

We use two methods to solve quantum hardware constraints. The first method is to first resolve the conflicts then perform the grouping operation. Such method is called as "swap then group". The second option is to first group these gates then concatenate the pulses of swap gates when necessary. Such

Algorithm 3 Overall Latency

Require: qasm, profile-table, latency-table
Ensure: overall latency
  1: Initialize latency (a list)
  2: **for** node in DAG.topological-order: **do**
  3:   latency[node] = max(latency[node's predecessor(s)]) + latency-table[node]
  4: **end for**
  5: overall-latency = last index of latency
  6: **return** overall-latency

---

method is called as "group then swap". We explore both options because on some quantum computer the swap operation is directly supported. We find that under certain circumstances, mapping then grouping has lower overall latency, in which case a swap gate is decomposed into three CNOT gates. Such method appears to have advantage most likely because those CNOT gates are more flexible in joining other gates. Furthermore, these CNOT gates are more likely to be cancelled with other CNOT gates such that the overall latency is reduced.

G. Optimizing the most frequent group

We select the group of highest frequency and spend more time training it with different methods so that the latency of this particular group could be further reduced. The goal is to further reduce overall latency without the high overhead when the pre-compiled pulses are used for a new program.

V. ACCELERATED DYNAMIC COMPILATION

A. The Notion of Coverage

After the static pre-compilation, we have the pulses for 133 profiled groups. Given a new program, it will be first decomposed into groups with "map2b4l" For the groups that fall into the pre-compiled set, we can directly use the pulses. We call them as "covered" groups and AccQOC does not incur any training overhead to generate pulses for them. We define the coverage as follows:

$$Coverage\ Rate = \frac{\#\ Groups\ covered\ by\ our\ category}{\#\ Groups\ of\ the\ program}$$

Clearly, the coverage is an important factor determining the benefit of our approach. For the "un-covered" groups, we will use dynamic QOC based compilation to generate pulses. We also call this step as training since it resembles the training process in machine learning. The following sections will describe our ideas to accelerate the training process.

B. Identify Similar Groups

Since the quantum control evolves from the initial matrix to the target matrix by multiplying time-dependent, hardware-specific control Hamiltonian matrices, similar matrices could share similar pulses. Therefore, pre-computed pulses could be used as inputs to GRAPE to reduce the number of iterations of gradient descent when we train a new group.
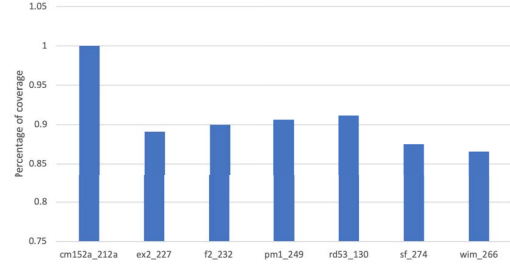

Fig. 7: The coverage of these programs are measured under the policy of "map2b4l". These programs are first decomposed into groups and compared with pre-compiled groups to determine the coverage.

Currently we use 4 similarity functions to decide whether groups are similar. The first two are simply the difference between the matrices: $d_1(\mathbf{A}, \mathbf{B}) = \sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij} - b_{ij}|$; $d_2(\mathbf{A}, \mathbf{B}) = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} (a_{ij} - b_{ij})^2}$. The last two similarity functions represent the fidelity of quantum unitary operations: $d_3(\mathbf{A}, \mathbf{B}) = Tr(A^*B)$; $d_4(\mathbf{A}, \mathbf{B}) = F(A, B) = \left(\mathrm{tr}\sqrt{\sqrt{A}B\sqrt{A}} \right)^2$.
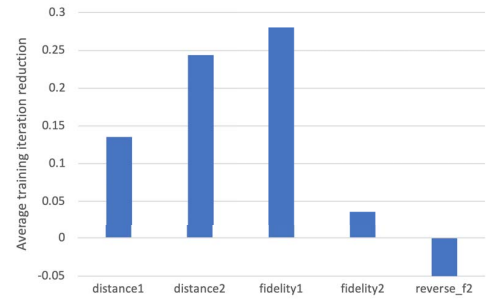

Fig. 8: It shows the average reduction of iterations required for computing pulses using each of the five similarity functions.

Figure 8 shows the partial results of accelerated training. We apply different similarity functions together with the SG algorithm described below. The first four similarity function attempts to measure the similarity between two groups, and the fifth one is the inverse of the fourth function and aims to measure the opposite of similarity. The result shows that "fidelity1" function achieves most reduction of iterations, and the fifth function causes the the number of iterations to increase.

C. Determining Compiling Sequence with Similarity

Based on the insights that the pulse of a new group can be generated faster with a similar group subject to a similarity function, for all un-covered groups, our goal is to generate an ordered *Compilation Sequence* $CS = [g_1, g_2, ..., g_n]$ (with n uncovered groups) that minimizes the estimated compile time. According to CS, the pulse for g_1 will be generated first, and it will be used as the initial matrix to train the pulse of g_2. To

minimize the total compile (training) time, we ensure that the sum of similarity between consecutive groups in the sequence is minimized. With n groups, we have $n!$ possible sequences (permutations), following steps are used to obtain CS.

First, we can compute the similarity of *any* two pairs of groups and construct a complete graph, denoted as *similarity graph (SG)*. In SG, each vertex is a group and the weight of edge is the similarity between two groups. We also include identity matrix as a special group in SG — when a new group is not close enough to any groups with pulse generated, the training of the new group will start the with identity matrix. With SG constructed, we can compute the Minimum Spanning Tree (MST) of SG, which minimizes the sum of edge weights. In the process of generating MST using the greedy algorithm, i.e., Prim algorithm, we can remember the sequence that all vertices are selected, this sequence is exactly what we need for CS. We choose the identity matrix as the starting point to generate MST. Figure 9 shows an example of this procedure.

D. Parallelizing Compilation with Balanced MST Partition

For a large program, the number of group can be large, the reduction of compilation time based on group similarity may not be sufficient. The good news is that the dependency of groups in CS is "soft" — even the pulse of predecessor groups are not available, we can always train a group starting from identity matrix. In fact, we can choose any order to train the un-covered groups, the consequence is just the longer training time. In this sense, we can consider that the training of each group is independent. Thus, the job of training all groups can be perfectly parallelized with multiple "workers" [1].

Since there is no dependency between the groups, the major factor determining the overall performance of parallel training is the workload assigned to each worker. In our problem formulation with SG and MST, the problem is how to partition the MST into multiple sub-graphs such that the difference of critical paths of them is minimized.

To achieve such division, we utilize METIS [18], a well-known graph partitioner to divide the MST as balanced as possible. However, this balanced partitioning method operates on a graph with node weights and divides it into connected sub-parts each having a similar sum of weights. Our MST generated from SG has only weights on its edges.

To fit METIS to our problem, we transformed the MST graph with cost on edges into a new graph with weight on nodes. Following the optimal sequence, we shift the cost of each edge to the weight of its newly added neighboring node. The first node in the sequence is specially assigned with a value proportional to the time it takes to train the first node from identity matrix. This step is shown in the Figure 9 from b to c.

After the transformation, we use METIS [18] to divide the new graph into sub-parts, each of which is computed on a separated computer, based on its local sequence. We merge

---

(a) A 6-node SG   (b) Minimum spanning tree

(c) MST with shifted weight   (d) Balance partition of MST

Fig. 9: The procedure from SG to partitioned MST: extract MST (b) from SG (a); shift the weight on edges to nodes (c); partition the graph into several balanced groups of similar sum of node weight, denoted by a distinct color (d).

pulses of all matrices after computation is done. Figure 9d shows an example of the partitioned graph. With the generated sub-graphs, we can assign workload to each worker to obtain the balanced execution during parallel processing.

VI. EVALUATION

In this section, we introduce our benchmark methodology based on the topology of IBM's 14-Qubit Quantum Computer available. Then we present our results of latency reduction and compilation time reduction.

A. Benchmarks

The benchmarks are selected from previous work [43]. Their test circuit data, which originated from the RevLib [41] is available to the public. The set of benchmarks from RevLib contains various reversible functions implemented using quantum gates. The functions include encoding functions, arithmetic functions, miscellaneous functions and symmetric functions etc. Several other functions like QFT(Quantum Fourier Transform) and GSE(Ground State Estimation)from ScaffCC [17] are also in the benchmark suite. All programs are mapped to the 14-bit IBM Q Melbourne chip [1] based on superconducting technology. The comprehensive benchmarks cover a variety of functions used in existing quantum algorithms, such as QFT in Shor algorithm. All evaluations are conducted on a 3.4 GHz machine with 4 cores and 40 GB RAM. The whole benchmark suite includes 159 programs. We randomly sampled some quantum programs with between 200 and 2000 gates, and two QFT programs to verify our idea. Table II shows the instruction mixes of 6 programs and the average of all programs.

---

[1]In this paper, we consider "workers" in abstract sense, and just focus on how to partition the workload. In reality, it can be a thread, process, or a CPU/GPU, or even a node in distributed computing platform.
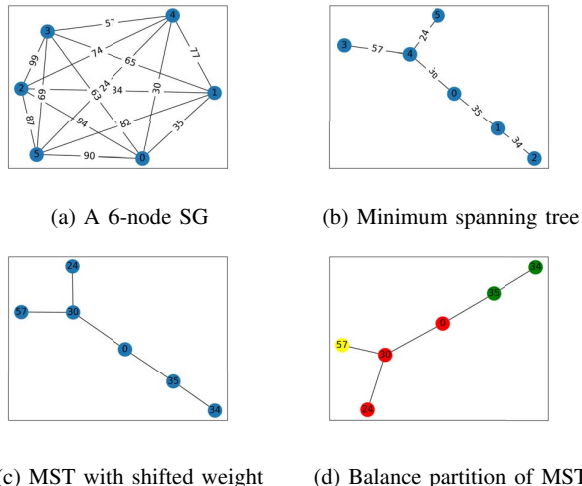
	x	t	h	cx	rz	tdg
4gt4-v0_79	0	56	28	105	0	42
cm152a_212	5	304	152	532	0	228
qft_10	0	0	20	90	90	0
qft_16	0	0	32	240	240	0
ex2_227	5	156	78	275	0	117
f2_232	6	300	150	525	0	225
all	0.10%	22%	15%	45%	1.1%	17%

TABLE II: Instruction Mixes of Benchmark Programs

B. Hardware Model

We use the topology from IBM Q Melbourne chip where the two-Qubit gates are not symmetric and therefore have certain directions. CNOT gate is only allowed in one direction. We extend the mapping algorithm of [43] with consideration of cross-talk to insert swap gates into the program. The direction of the 2-bit gate and topology of Melbourne is showed in Figure 10.
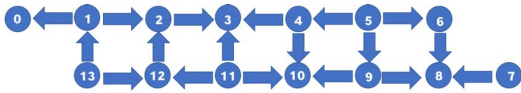

Fig. 10: Qubits connection of IBM's melbourne chip. [3]

C. Mitigation of Cross-talk in Mapping

We quantify the overall cross-talk effect as the total occurrences of close CNOT operations in each layer. This metric is adopted from the paper [29]. The rational to use this metric is discussed in Section IV-A. Before and after our mapping algorithm, the cross-talk effect deceases for most of the tested quantum programs and we observe an average of 17.6% reduction of cross-talk effect. The systematic method to mitigate cross-talk effect is still an open question. Nevertheless, our workflow is the among the first to tackle the problem of cross-talk effect by Qubit mapping. We leave the systematic study of the problem for future work.
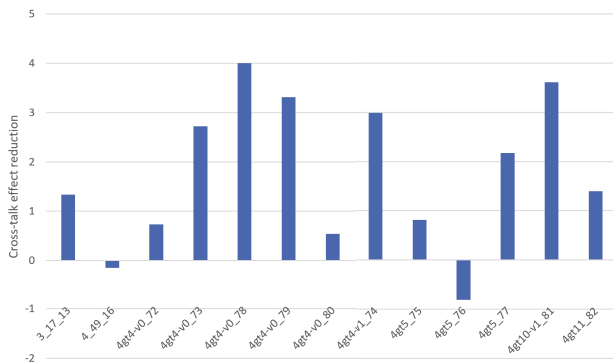

Fig. 11: The effect of cross-talk effect is quantified by the sum of occurrences of nearby pairs of CNOT gates. After the heuristic takes it into consideration, the cross-talk effect deceases for most of the tested quantum programs.

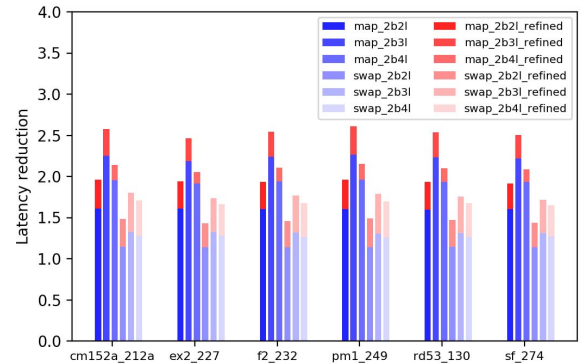## D. Reduction of Overall Latency

Fig. 12: The latency reduction for 6 quantum programs is shown, each with 6 policies differentiated by the gradually-fading color scheme. The red ones represent the case in which the most frequent group is targeted for optimization, compared against blue ones that omit this step.
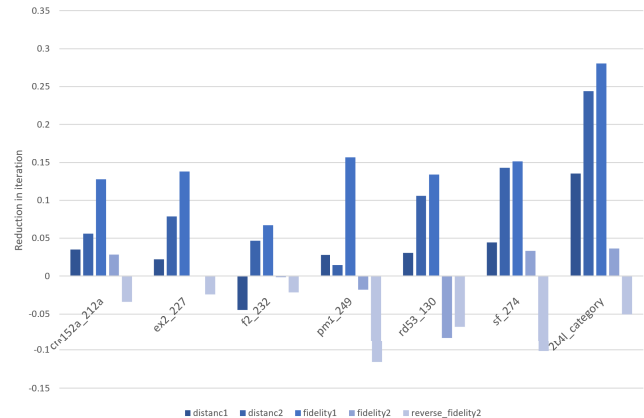

Fig. 13: Reduction of training iteration: for each of the 7 quantum programs (including a set of profiled groups by map2b4l), 5 similarity functions are applied.

We have implemented 6 different grouping policies. These policies feature different sizes of groups. The latency reduction of 6 grouping policies mostly lies between $1.2\times$ and $2.6\times$ compared with gate-based compilation.

In our static pre-compilation process, a target program will have groups of gates falling into the profiled category. Among these groups, we pick the one with highest frequency and re-generate the pulse for this group. We will spend more computational resource training this group such that its latency could be reduced. This brings further latency reduction as plotted in the Figure 12.

E. The effect of the sequence of mapping and grouping

The sequence of mapping and pulse generation in the control flow is also considered. The mapping algorithm of [43] is adopted. We find that under certain circumstances, "mapping then grouping" has lower overall latency, in which case the swap gate is decomposed into three CNOT gates. These CNOT gates could form a more flexible group and are more likely to get diminished such that the overall latency could be reduced. However, when swap operations are directly supported in the target machine, it is better to choose the policy "grouping then mapping", since hardware-supported operations are often optimized to reach high fidelity with low latency.

F. Coverage

Here we show the coverage of 7 quantum programs under the "map2b4l" policy. Since the category of groups is profiled by one third of our benchmarks. The profiled category is likely to contain most of groups that will appear in other programs. We achieve an average coverage of 89.7%.

Moreover, Figure 14 shows that the number of 2b4l groups grows much slower than linearly(though not strictly logarithmic) with the number of gates, meaning the probability of encountering uncovered groups does not scale with the size of a quantum program. It demonstrates the high reusability of pre-compiled groups.

Higher coverage corresponds to lower compilation time, since pulses of covered groups are pre-computed. However, with larger quantum programs in the future, we expect to see lower coverage but more latency reduction for each group. When that happens, MST-accelerated QOC will come to use and demonstrate its powerfulness. Hence, there exists a clear trade-off to be considered between the pre-compilation overhead and coverage, especially when dealing with large quantum programs.
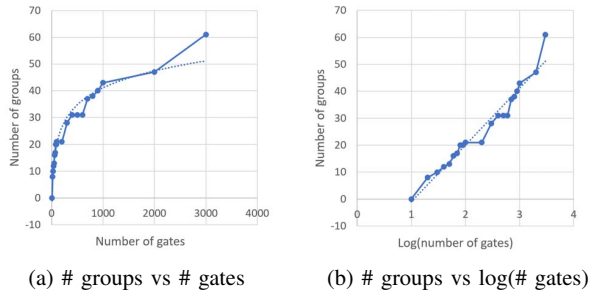

(a) # groups vs # gates   (b) # groups vs log(# gates)

Fig. 14: The figure demonstrates the relationship between the growth of group number and that of gate number.

G. Accelerated Training

In this part, we compare the training iterations of groups with and without accelerated training. To directly show the better performance of accelerated training, we demonstrate this methodology with a pre-compiled category under the "map2b4l" policy, which has 133 groups. As shown in Figure 13, we could reach a max iteration reduction of 28%. The
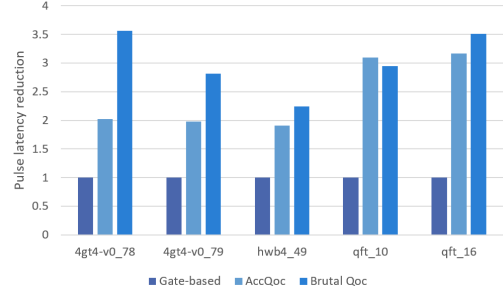

Fig. 15: Comparing AccQOC with Brute-Force QOC Training

acceleration highly relies on the size of MST we have. For a larger MST, the two group connected are more likely to be very close to each other. We expect to see more iteration reduction with larger quantum programs. We use iteration reduction as our metric for computing source because the running time of optimal control grows linearly with the number of iterations. In our experiments, multiple optimal control programs are executed in a multi-programmed fashion. We cannot ensure each execution get equal system resource. Therefore, when measuring the improvement of AccQOC, it is more reasonable and accurate to report the number of iterations, instead of running time.

H. Compilation Time

The most time-consuming part of our workflow is the step of generating pulse list and latency list. After the pre-compilation is finished, it takes no time to look up the pulses for covered groups. Quantum programs with higher coverage will gain more compilation time reduction, since less groups need to be compiled to pulses. Moreover, since the size of our pre-compiled category does not scale linearly with the size of quantum programs, larger quantum programs will feature more compilation reduction.

To show the relation between compile time reduction and latency reduction, we choose the "brute force" QOC's compilation time as our baseline. The same methodology is also used in [13]. We form the "brute force QOC" groups by including as many qubits and gates as possible. The Figure 15 shows that the average latency reduction of *AccQOC* is $2.43\times$ while the "brute force QOC" achieves $3.01\times$ reduction. [13] achieves similar latency reduction as AccQOC. On the other side, the compilation time reduction compared to "brute force" QOC compilation is $9.88\times$. Thus, our method trades off minor latency reduction ($2.43\times$ compared with $3.01\times$) for significant compile time reduction ($9.88\times$).

VII. Related Work

The traditional gate-based workflow of quantum computing has been well studied [24], [28], [33]. Techniques have been proposed to optimize the compiler frontend [17] and the mapping problem [26], [43]. After the idea of quantum optimal control is proposed, some research has focused on the concrete implementation of algorithms [9], [20], [21], [37], [42]. The

optimal control tool developed by [25] uses GPU-based automatic differentiation to accelerate the iterations when input matrix evolving towards target matrix. Recent work [35] noticed the problem of large overhead of quantum optimal even with GPU acceleration. However, the methodology used in the paper does not address this problem and their methodology does not scale with the size of the quantum programs. The idea of utilizing commutativity could increase parallelism among groups; but those groups are relatively large to show benefits, making it slow for quantum optimal control to compute. For example, the paper shows that the aggregated gates form groups that contain up to 10 qubits, the QOC of such groups could take hours to generate the pulses. Our solution will not require much computation resource once our category of groups is pre-compiled. The prior work [23] builds a library of gates with specific rotation angles. A more recent work [13] applies idea of pre-compilation on Quantum Variational Algorithms. The proposed methodology of hyperparameter tuning works well with the iterative execution of variational algorithms. However, the methodology cannot be applied to the static groups in non-variational algorithms. Compared to them, AccQOC is more general and not limited to any special types of quantum algorithms. The hybrid characteristic of our method makes it suitable for both variational algorithms like VQE and non-variational algorithms like Shor algorithm. It can support arbitrary rotation angles since it simply corresponds to a different matrix.

Most importantly, our workflow does not scale linearly with the size of input program. Therefore, our workflow *AccQOC* is more scalable and more suitable for future compilation of large quantum programs.

VIII. CONCLUSION

In this paper, we propose *AccQOC*, a comprehensive static/dynamic hybrid workflow to transform gate groups (equivalent to matrices) to pulses using QOC (Quantum Optimal Control) with a reasonable compilation time budget. AccQOC is composed of static pre-compilation and accelerated dynamic compilation. We leverage static pre-compilation to generate pulses for the frequently used groups to eliminate the dynamic compilation time for them. The pulse is generated using QOC with binary search to determine the latency. For a new program, the dynamic compilation deals with "uncovered" groups with accelerated pulse generation. The key insight is that the pulse of a group can be generated faster based on the generated pulse of a *similar* group. We propose to reduce the compilation time by generating an ordered sequence of groups in which the sum of similarity among consecutive groups in the sequence is minimized. With the methodology of AccQOC, we reached a balanced point of compilation time and overall latency. The results show that accelerated compilation based on MST achieves $9.88\times$ compilation speedup compared to the standard compilation of each group while maintaining an average $2.43\times$ latency reduction compared with gate-based compilation.

REFERENCES

[1] https://www.research.ibm.com/ibm-q/technology/devices/.
[2] Dorit Aharonov and Michael Ben-Or. Fault-tolerant quantum computation with constant error rate. *arXiv preprint quant-ph/9906129*, 1999.
[3] Gadi Aleksandrowicz, Thomas Alexander, P Barkoutsos, L Bello, Y Ben-Haim, D Bucher, FJ Cabrera-Hernández, J Carballo-Franquis, A Chen, CF Chen, et al. Qiskit: An open-source framework for quantum computing. *Accessed on: Mar*, 16, 2019.
[4] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.
[5] Bikash K Behera, Tasnum Reza, Angad Gupta, and Prasanta K Panigrahi. Designing quantum router in ibm quantum computer. *Quantum Information Processing*, 18(11):328, 2019.
[6] Charles H Bennett, David P DiVincenzo, John A Smolin, and William K Wootters. Mixed-state entanglement and quantum error correction. *Physical Review A*, 54(5):3824, 1996.
[7] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J Bremner, John M Martinis, and Hartmut Neven. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595, 2018.
[8] DPL Aude Craik, NM Linke, MA Sepiol, TP Harty, JF Goodwin, CJ Ballance, DN Stacey, AM Steane, DM Lucas, and DTC Allcock. High-fidelity spatial and polarization addressing of ca+ 43 qubits using near-field microwave control. *Physical Review A*, 95(2):022337, 2017.
[9] P De Fouquieres, SG Schirmer, SJ Glaser, and Ilya Kuprov. Second order gradient ascent pulse engineering. *Journal of Magnetic Resonance*, 212(2):412–417, 2011.
[10] Edward Farhi and Aram W Harrow. Quantum supremacy through the quantum approximate optimization algorithm. *arXiv preprint arXiv:1602.07674*, 2016.
[11] Arkady Fedorov, Lars Steffen, Matthias Baur, Marcus P da Silva, and Andreas Wallraff. Implementation of a toffoli gate with superconducting circuits. *Nature*, 481(7380):170, 2012.
[12] Steffen J Glaser, Ugo Boscain, Tommaso Calarco, Christiane P Koch, Walter Köckenberger, Ronnie Kosloff, Ilya Kuprov, Burkhard Luy, Sophie Schirmer, Thomas Schulte-Herbrüggen, et al. Training schrödinger's cat: quantum optimal control. *The European Physical Journal D*, 69(12):279, 2015.
[13] Pranav Gokhale, Yongshan Ding, Thomas Propson, Christopher Winkler, Nelson Leung, Yunong Shi, David I Schuster, Henry Hoffmann, and Frederic T Chong. Partial compilation of variational algorithms for noisy intermediate-scale quantum machines. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 266–278. ACM, 2019.
[14] Lov K Grover. A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043*, 1996.
[15] Robin Harper, Steven T Flammia, and Joel J Wallman. Efficient learning of quantum noise. *arXiv preprint arXiv:1907.13022*, 2019.
[16] Jeremy Hsu. Ces 2018: Intel's 49-qubit chip shoots for quantum supremacy. *IEEE Spectrum Tech Talk*, 2018.
[17] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. Scaffcc: Scalable compilation and analysis of quantum programs. *Parallel Computing*, 45:2–17, 2015.
[18] George Karypis and Vipin Kumar. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*, 1998.
[19] Julian Kelly, Rami Barends, Austin G Fowler, Anthony Megrant, Evan Jeffrey, Theodore C White, Daniel Sank, Josh Y Mutus, Brooks Campbell, Yu Chen, et al. State preservation by repetitive error detection in a superconducting quantum circuit. *Nature*, 519(7541):66, 2015.

[20] Navin Khaneja, Timo Reiss, Cindie Kehlet, Thomas Schulte-Herbrüggen, and Steffen J Glaser. Optimal control of coupled spin dynamics: design of nmr pulse sequences by gradient ascent algorithms. *Journal of magnetic resonance*, 172(2):296–305, 2005.

[21] Will Knight. Ibm raises the bar with a 50-qubit quantum computer. *Sighted at MIT Review Technology*, 2017.

[22] Vadim Krotov. *Global methods in optimal control theory*, volume 195. CRC Press, 1995.

[23] Daniel Kudrow, Kenneth Bier, Zhaoxia Deng, Diana Franklin, Yu Tomita, Kenneth R Brown, and Frederic T Chong. Quantum rotations: a case study in static and dynamic machine-code generation for quantum computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 166–176, 2013.

[24] Ryan LaRose. Overview and comparison of gate level quantum software platforms. *Quantum*, 3:130, 2019.

[25] Nelson Leung, Mohamed Abdelhafez, Jens Koch, and David Schuster. Speedup for quantum optimal control from automatic differentiation based on graphics processing units. *Physical Review A*, 95(4):042318, 2017.

[26] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for nisq-era quantum devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1001–1014. ACM, 2019.

[27] Benjamin Lienhard, Jochen Braumüller, Wayne Woods, Danna Rosenberg, Greg Calusine, Steven Weber, Antti Vepsäläinen, Kevin O'Brien, Terry P Orlando, Simon Gustavsson, et al. Microwave packaging for superconducting qubits. *arXiv preprint arXiv:1906.05425*, 2019.

[28] Prakash Murali, Norbert Matthias Linke, Margaret Martonosi, Ali Javadi Abhari, Nhung Hong Nguyen, and Cinthia Huerta Alderete. Full-stack, real-system quantum computer studies: Architectural comparisons and design insights. *arXiv preprint arXiv:1905.11349*, 2019.

[29] Prakash Murali, David C McKay, Margaret Martonosi, and Ali Javadi-Abhari. Software mitigation of crosstalk on noisy intermediate-scale quantum computers. *arXiv preprint arXiv:2001.02826*, 2020.

[30] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[31] Joe O'Gorman and Earl T Campbell. Quantum computation with realistic magic-state factories. *Physical Review A*, 95(3):032338, 2017.

[32] Anthony P Peirce, Mohammed A Dahleh, and Herschel Rabitz. Optimal control of quantum-mechanical systems: Existence, numerical approximation, and applications. *Physical Review A*, 37(12):4950, 1988.

[33] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[34] Vivek V Shende, Stephen S Bullock, and Igor L Markov. Synthesis of quantum-logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1000–1010, 2006.

[35] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I Schuster, Henry Hoffmann, and Frederic T Chong. Optimized compilation of aggregated instructions for realistic quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1031–1044. ACM, 2019.

[36] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[37] Shlomo E Sklarz and David J Tannor. Loading a bose-einstein condensate onto an optical lattice: An application of optimal control theory to the nonlinear schrödinger equation. *Physical Review A*, 66(5):053619, 2002.

[38] Martin Suchara, Arvin Faruque, Ching-Yi Lai, Gerardo Paz, Frederic T Chong, and John Kubiatowicz. Comparing the overhead of topological and concatenated quantum error correction. *arXiv preprint arXiv:1312.2316*, 2013.

[39] J Werschnik and EKU Gross. Quantum optimal control theory. *Journal of Physics B: Atomic, Molecular and Optical Physics*, 40(18):R175, 2007.

[40] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. Mapping quantum circuits to ibm qx architectures using the minimal number of swap and h operations. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 142. ACM, 2019.

[41] Robert Wille, Daniel Große, Lisa Teuber, Gerhard W Dueck, and Rolf Drechsler. Revlib: An online resource for reversible functions and reversible circuits. In *38th International Symposium on Multiple Valued Logic (ismvl 2008)*, pages 220–225. IEEE, 2008.

[42] Wusheng Zhu, Jair Botina, and Herschel Rabitz. Rapidly convergent iteration methods for quantum optimal control of population. *The Journal of Chemical Physics*, 108(5):1953–1963, 1998.

[43] Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.