Paulihedral: A Generalized Block-Wise Compiler Optimization Framework for Quantum Simulation Kernels

Gushu Li
University of California
Santa Barbara, USA
gushuli@ece.ucsb.edu

Anbang Wu
University of California
Santa Barbara, USA
anbang@cs.ucsb.edu

Yunong Shi
Amazon Braket
New York, USA
shiyunon@amazon.com

Ali Javadi-Abhari
IBM Quantum
Yorktown Heights, USA
ali.javadi@ibm.com

Yufei Ding
University of California
Santa Barbara, USA
yufeiding@cs.ucsb.edu

Yuan Xie
University of California
Santa Barbara, USA
yuanxie@ece.ucsb.edu

ABSTRACT

The quantum simulation kernel is an important subroutine appearing as a very long gate sequence in many quantum programs. In this paper, we propose Paulihedral, a block-wise compiler framework that can deeply optimize this subroutine by exploiting high-level program structure and optimization opportunities. Paulihedral first employs a new Pauli intermediate representation that can maintain the high-level semantics and constraints in quantum simulation kernels. This naturally enables new large-scale optimizations that are hard to implement at the low gate-level. In particular, we propose two technology-independent instruction scheduling passes, and two technology-dependent code optimization passes which reconcile the circuit synthesis, gate cancellation, and qubit mapping stages of the compiler. Experimental results show that Paulihedral can outperform state-of-the-art compiler infrastructures in a wide-range of applications on both near-term superconducting quantum processors and future fault-tolerant quantum computers.

CCS CONCEPTS

• **Computer systems organization** → **Quantum computing**; • **Software and its engineering** → **Compilers**; • **Hardware** → **Emerging languages and compilers**.

KEYWORDS

quantum computing, compiler, quantum simulation

ACM Reference Format:
Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, and Yuan Xie. 2022. Paulihedral: A Generalized Block-Wise Compiler Optimization Framework for Quantum Simulation Kernels. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503222.3507715

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507715

1 INTRODUCTION

One of the most important quantum algorithm design principles is quantum simulation (or Hamiltonian simulation). Simulating a quantum physical system, which motivated Feynman's proposal to build a quantum computer [19], is by itself an important application of quantum computing [2, 33]. Later, the idea of quantum simulation was extended to quantum algorithms for other applications, e.g., linear systems [24], quantum principal component analysis [35], and quantum support vector machine [46]. These algorithms involve simulating an artificial quantum system crafted based on the target problem. In recently developed variational quantum algorithms for near-term quantum computers (e.g., VQE for chemistry [45] and QAOA for combinatorial optimization [18]), the program structures are also inspired by the simulation principle.

Because the quantum simulation principle is shared among many algorithms, one subroutine, which we term the *quantum simulation kernel* in this paper, appears frequently in quantum programs. This kernel is to implement the operator (controlled-)$\exp(iHt)$ where H is the Hamiltonian of the simulated system and $t \in \mathbb{R}$ is system evolution time. Since it is hard in general to directly compile $\exp(iHt)$ into executable single- and two-qubit gates, a compiler usually decomposes H into the sum of local Hamiltonians [33] (simulation of which can be easily compiled to basic gates) and then synthesize them one-by-one. Consequently, the quantum simulation kernel will be compiled to a very long gate sequence and constitute the vast majority of cost in post-compilation quantum programs.

Optimizing the compilation of this kernel can immediately benefit a wide range of quantum applications. However, three key challenges have so far hindered deeper compiler optimizations for quantum simulation kernels.

First, existing quantum compilers (e.g., Qiskit [1], Quilc [52], t|ket⟩ [50]) lack a good formal high-level intermediate representation (IR). Once programs are converted to low-level gate sequences, the high-level semantics of quantum simulation kernels are lost and hard to reconstruct from assembly-style gate sequences. Moreover, simulation kernels face different constraints in different algorithms. Previous ad-hoc optimizations of quantum simulation [3, 13, 14, 17, 23, 25, 29, 31, 48, 56, 61, 62] are mostly algorithm-specific and do not generalize due to the lack of a formal IR that can uniformly represent simulations kernels as well as varying constraints that are attached to them in different algorithms.

Second, most optimizations (e.g., circuit rewriting [53], gate cancellation [42], template matching [37], qubit mapping [39]) in today's quantum compilers [1, 52] are local program transformations at small scale. However, these passes are designed for generic input program and fail to leverage the deeper optimization opportunities present in quantum simulation kernels. These opportunities are mainly from the properties of Pauli strings which naturally appear in the (Suzuki-)Trotter Hamiltonian approximation [33, 55], Jordan-Wigner [26] or Bravyi-Kitaev [9] fermion-to-qubit transformation, etc.

Third, quantum simulation kernels appear in a wide range of algorithms. Some algorithms [2, 24, 33, 35, 46] are designed for fault-tolerant quantum computers with quantum error correction while others [18, 45] target near-term noisy quantum computers. The hardware models of these backends can be very different and one single optimization pass may not be suitable for all of them. Adapting the high-level algorithmic optimizations to the various (and ever-evolving) hardware platforms with different constraints and optimization objectives naturally invokes a reconfigurable compiler infrastructure.

To overcome these challenges, we propose Paulihedral, a compiler framework backed by a formal IR to deeply optimize quantum simulation kernels. A brief comparison between Paulihedral and conventional quantum compilers is shown in Figure 1. **First**, Paulihedral comes with a new IR, namely Pauli IR, to represent the quantum simulation kernels at the Pauli string level rather than the gate level. The syntax of Pauli IR has a novel block structure which can uniformly represent the simulation kernels of different forms and constraints. The semantics of Pauli IR is defined on the commutative matrix addition operation. Such semantics guarantees that the follow-up high-level algorithmic optimizations are always semantics-preserving and can be safely applied. **Second**, we propose several novel optimization passes to reconcile instruction scheduling, circuit synthesis, gate cancellation, and qubit layout/routing at the Pauli IR level. All these passes are much more effective than their counterparts in conventional gate-based compilers because they are operating in a large scope where the algorithmic properties of Pauli strings (quantum simulation kernel) are fully exploited. The optimization algorithms in these passes are also highly scalable since analyzing and processing Pauli strings are much easier than handling the gate matrices on a classical computer. **Third**, we decouple the technology-independent and technology-dependent optimizations at different stages and Paulihedral can be extended to different backends by adding/modifying the technology-dependent passes. To showcase, we develop technology-dependent optimizations for two different backends, the fault-tolerant quantum computer and the noisy near-term superconducting quantum processor.

Our comprehensive evaluations show that Paulihedral outperforms state-of-the-art baseline compilers (Qiskit [1], t|ket⟩ [50] and algorithm-specific compilers [3–5]) with significant gate count and circuit depth reduction on both fault-tolerant and superconducting backends, and only introduces very small additional compilation time. We also perform real-system experiments to show that Paulihedral can significantly increase the end-to-end success rate of QAOA programs on IBM's superconducting quantum devices.

Our major contributions can be summarized as follows:

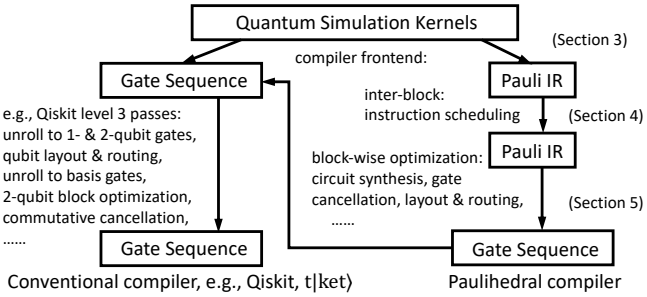

Figure 1: Paulihedral vs conventional compilers

(1) We propose Paulihedral, an extensible algorithmic compiler framework that can deeply optimize quantum simulation kernels and thus benefit the compilation of a wide range of quantum programs, with passes that make it retargetable to various backends and optimization objectives.
(2) We define a new Pauli IR with formal syntax and semantics which can uniformly represent quantum simulation kernels and encode algorithmic constraints of seemingly very different algorithms, and safely expose high-level information to the compiler for optimizations.
(3) We propose several compiler passes for different optimization objectives and backends. They can outperform previous works by systematically leveraging the algorithmic information and they are scalable to efficiently handle larger-size programs.
(4) Our experiments on 31 different benchmarks show that Paulihedral can outperform state-of-the-art baseline compilers with significant gate count and circuit depth reduction. For example, compared with t|ket⟩ [50], Paulihedral achieves 53.1% gate count reduction and 53.3% circuit depth reduction on average on the superconducting backend, as well as 33.6% gate count and 65.0% circuit depth reduction on the fault-tolerant backend, using only $\sim$ 5% additional compilation time. For QAOA on a real quantum device, Paulihedral achieves end-to-end 1.24× success probability improvement on average (up to 1.87×) against the baseline Qiskit compiler [1].

2 BACKGROUND

In the section we introduce the necessary background about quantum simulation kernels. We do not cover basic quantum computing concepts (e.g., qubit, gate, linear operator, circuit) and we recommend [43] for more details.

2.1 Pauli String and Compilation

We start with the Pauli string, the basic concept in quantum simulation. For an n-qubit system, a Pauli string is defined as $P = \sigma_{n-1}\sigma_{n-2}\cdots\sigma_0$ where $\sigma_i \in \{I, X, Y, Z\}$, $0 \le i \le n-1$. X, Y, Z are the three Pauli operators, and I is the identity. σ_i corresponds to the i-th qubit. The operators in a Pauli string P can represent a Hermitian operator $\otimes_{i=0}^{n-1}\sigma_i$ ($\otimes$ is the Kronecker product), which can be denoted by P without ambiguity. In the rest of this paper,

we do not distinguish a Pauli string P and the Hermitian operator generated by P.

One important property of a Pauli string is that the operator $\exp(\mathrm{i}P\frac{\theta}{2})$ can be easily synthesized into basic gates. Figure 2 shows an example of synthesizing $\exp(\mathrm{i}Y_4Z_3I_2X_1Z_0\frac{\theta}{2})$. There are two identical layers of single-qubit gates at the beginning and the end of the synthesized circuit. In this single-qubit gate layer, there are H or Y gates on those qubits whose operators are X (i.e., $q1$) or Y (i.e., $q4$) in the Pauli string, respectively. In the middle is a left CNOT tree, a central $\mathrm{Rz}(\theta)$ gate, and a right CNOT tree. The left tree can be generated in different ways and the only requirement is to connect all the qubits whose operators are not the identity in P (e.g., $q0$, $q1$, $q3$, $q4$ in Figure 2). The lower half of Figure 2 shows three different but valid ways to generate the CNOT tree circuits and their corresponding tree graphs. In these trees, the CNOT gates should connect the qubits from the leaf nodes to the root node. Any qubit in the tree can become the root (e.g., $q4$ in Figure 2 (1) (2), $q1$ in Figure 2 (3)). The central $\mathrm{Rz}(\theta)$ gate is applied on the root qubit and the right CNOT tree has the same CNOT gates in the left tree but in a reversed order. Paulihedral uses this algorithmic flexibility in synthesis to increase gate cancellation and reduce the mapping overhead.

2.2 Quantum Simulation Kernels

The quantum simulation kernel is to (approximately) implement the operator $\exp(\mathrm{i}Ht)$ where H is the Hamiltonian of the simulated system and $t \in \mathbb{R}$. Since directly compiling $\exp(\mathrm{i}Ht)$ into single- and two-qubit gates is hard, a compiler usually expands H in the Pauli basis, i.e., $H = \sum_{j=1}^{N} w_j P_j$ where $w_j \in \mathbb{R}$ and P_j is a Pauli string. Then $\exp(\mathrm{i}Ht)$ is approximated using the Trotter formula [60]: $\exp(\mathrm{i}Ht) = \left[\prod_{j=1}^{N} \exp(\mathrm{i}P_j w_j \Delta t)\right]^{\frac{t}{\Delta t}} + O(t\Delta t)$. Δt is a parameter determined by the simulation accuracy. Figure 3 (a) shows the expansion process. $\exp(\mathrm{i}Ht)$ is first converted to $\frac{t}{\Delta t}$ terms of $\exp(\mathrm{i}H\Delta t)$. Each $\exp(\mathrm{i}H\Delta t)$ is then expanded to an array of $\exp(\mathrm{i}P_j w_j \Delta t)$ and converted to basic gates.

Quantum simulation kernels also appear in recently developed variational quantum algorithms, in which the vast majority of the program is an ansatz (parameterized quantum circuit). One popular type of ansatz with good trainability is the application-inspired ansatz [10] which can be considered as a simulation kernel. Compared with implementing $\exp(\mathrm{i}H\Delta t)$, the only difference is that the Δt is changed to some tunable parameters associated with different

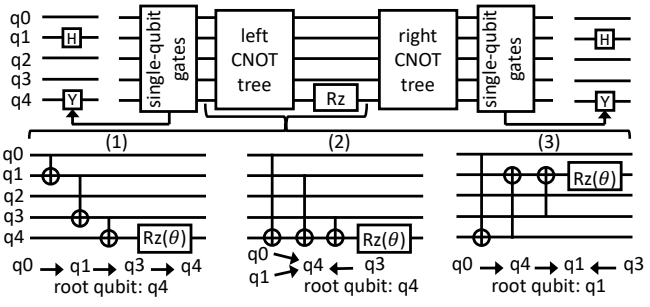

Figure 2: Synthesis example of $\exp(\mathrm{i}Y_4Z_3I_2X_1Z_0\frac{\theta}{2})$

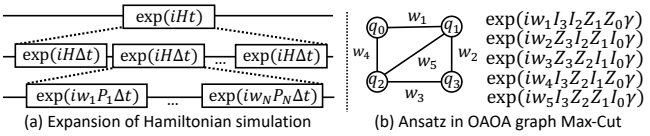

Figure 3: Example of quantum simulation kernels

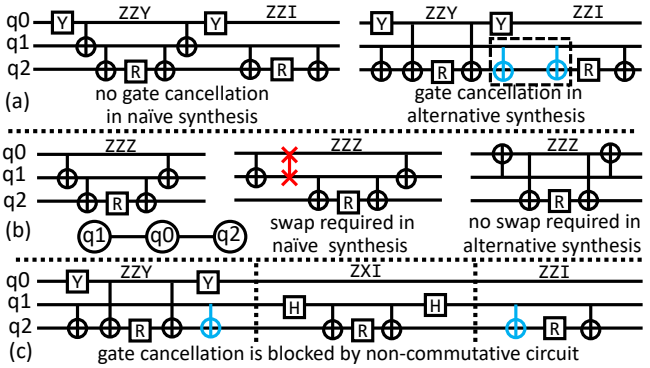

Figure 4: Optimization opportunities and challenges

Pauli strings and *the overall program structure remains the same*. For example, Figure 3 (b) shows the ansatz of QAOA algorithm [18] on a 4-node graph Max-Cut problem. The graph of the problem has 5 edges of different weights, and the Hamiltonian of this problem is the weighted sum of the 5 Pauli strings associated with the 5 edges. The majority of the QAOA ansatz [18] is to implement the 5 operators on the right (γ is the parameter).

3 FOUNDATIONS OF PAULIHEDRAL

In this section, we first introduce the opportunities and challenges of compiler optimizations for the simulation kernel. Then we formally introduce a new IR that maintains the high-level information and the algorithm constraints in Paulihedral.

3.1 Opportunities and Challenges

The optimization opportunities used in this paper come from the properties of Pauli strings mentioned above. We introduce them by the examples in Figure 4. **1) Gate cancellation:** It is possible to have more gate cancellation by selecting a different synthesis plan for the $\exp(\mathrm{i}P\theta)$. Suppose the naive synthesis is the one in Figure 2 (1) and we have two Pauli strings, ZZY and ZZI. Under the naive synthesis (on the left of Figure 4 (a)) there is no gate cancellation. However, in an alternative synthesis of ZZY, we can have two CNOT gates cancelled (on the right of Figure 4 (a)). **2) Mapping:** The mapping overhead onto connectivity-constrained architectures can also be reduced. For example, we wish to map the ZZZ simulation circuit onto a linear architecture with the current mapping shown in Figure 4 (b). Under the naive synthesis we need to insert one SWAP between $q0$ and $q1$. While a better synthesis plan on the right of Figure 4 (b) does not require any SWAPs.

Although there is much optimization space for quantum simulation kernels, such optimizations are not yet widely deployed in

$$
\begin{aligned}
\langle program \rangle &::= \langle pauli\_block \rangle \\
&\quad | \ \langle program \rangle \ ; \ \langle pauli\_block \rangle \\
\langle pauli\_block \rangle &::= \{ \langle pauli\_str\_list \rangle, parameter \} \\
\langle pauli\_str\_list \rangle &::= \langle pauli\_str, weight \rangle \\
&\quad | \ \langle pauli\_str\_list \rangle \ ; \ \langle pauli\_str, weight \rangle \\
\langle pauli\_str \rangle &::= \sigma_{n-1}\sigma_{n-2}\cdots\sigma_0 \\
\sigma_i &::= I \mid X \mid Y \mid Z, \ (0 \le i \le n-1) \\
parameter,\ weight &\in \mathbb{R}
\end{aligned}
$$

Figure 5: Formal syntax of an n-qubit Pauli IR program

today's quantum compiler infrastructures due to the following challenges. **1) Missing high-level information:** Once the program is converted to basic gates, where today's compilers perform most optimizations, it is hard to identify and reconstruct the high-level semantics of Pauli string simulation circuit blocks from an assembly-style gate sequence. **2) Non-semantics-preserving optimization:** To leverage some optimization opportunities would require non-semantics-preserving operations that are usually not allowed in a compiler. For example, consider the program in Figure 4 (c). It is known from Figure 4 (a) that gates can be cancelled between ZZY and ZZI but now there is an ZXI simulation circuit between them. We observe that the order of the simulation terms with respect to different Pauli strings is not specified in the Trotter formula or the variational form requirement. So, from an algorithmic perspective, the compiler may exchange the order of ZZI and ZXI, making ZZY and ZZI adjacent for gate cancellation. However, such operation is not semantics preserving from a gate-level perspective because, in general, $\exp(iZZI\theta_1)\exp(iZXI\theta_2) \neq \exp(iZXI\theta_2)\exp(iZZI\theta_1)$. This would be impossible to leverage without an IR that is able to encode such algorithmic knowledge.

3.2 Pauli IR: Syntax and Semantics

To overcome the challenges above, the objective of the new IR is to maintain high-level algorithmic information and make all transformations semantics-preserving. Our new IR, namely Pauli IR, realizes them with its syntax and semantics.

Syntax: The syntax is shown in Figure 5 and explained as follows. A *program* is recursively defined as a list of *pauli_block*s. Each *pauli_block* is a tuple with two elements. The first element is a list of weighted Pauli strings (*pauli_str_list*s) and the second element is a real-valued *parameter* shared by all Pauli strings in this *pauli_block*. One element in the *pauli_str_list* is an n-qubit Pauli string and a real-value *weight*. Figure 6 shows the Pauli IR code of three example programs. Figure 6 (a) simulates the Hamiltonian of H_2 and each *pauli_block* has one *pauli_str*. Figure 6 (b)(c) are variational quantum algorithms so that parameters are labeled by θ and γ. In the UCCSD program (Figure 6 (b)), each *pauli_block* has multiple *pauli_str*s which share the same θ in the *pauli_block*. In the QAOA program (Figure 6 (c)), all *pauli_str*s are in one *pauli_block*, sharing the parameter γ.

Encoding constraints: One key advantage of the IR syntax is that the algorithmic constraints in all simulation kernels, as far as we know, can be naturally encoded. In some simulation kernels (e.g., UCCSD [45], QAOA for constrained optimization [47]), the

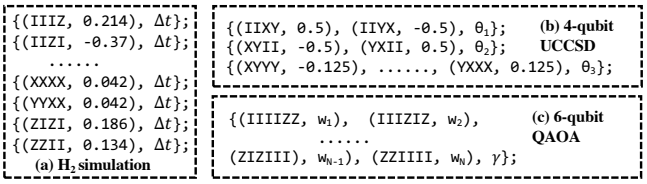

Figure 6: Example Puali IR programs

$$
\begin{aligned}
[\![\varnothing]\!] &= 0 \\
[\![\langle program \rangle ; \langle pauli\_block \rangle]\!] &= [\![\langle program \rangle]\!] + [\![\langle pauli\_block \rangle]\!] \\
[\![\{\langle pauli\_str\_list \rangle, parameter\}]\!] &= parameter \times [\![\langle pauli\_str\_list \rangle]\!] \\
[\![\langle pauli\_str\_list \rangle ; \langle pauli\_str, weight \rangle]\!] &= [\![\langle pauli\_str\_list \rangle]\!] \\
&\quad + [\![\langle pauli\_str, weight \rangle]\!] \\
[\![\langle pauli\_str, weight \rangle]\!] &= weight \times [\![pauli\_str]\!] \\
[\![\sigma_{n-1}\sigma_{n-2}\cdots\sigma_0]\!] &= \sigma_{n-1} \otimes \sigma_{n-2} \otimes \cdots \otimes \sigma_0
\end{aligned}
$$

Figure 7: Formal semantics of an n-qubit Pauli IR program

algorithm requires that some Pauli strings should always appear together for some algorithmic purposes like symmetry preserving [21], parameter sharing [18, 45], error suppression [23], etc. Pauli IR employs a *pauli_block* structure to represent such constraints. The compiler can extract such information and all the *pauli_str*s inside one *pauli_block* are always scheduled together in follow-up optimization passes. In the rest of this paper, *pauli_block* is denoted by block for simplicity.

Semantics: The IR's semantics function, which is denoted by $[\![\langle program \rangle]\!]$, can be formally defined by the rules in Figure 7. This function is a mapping from the IR syntax to the set of all Hermitian operators in a 2^n-dimensional Hilbert space as our IR is to represent the Hamiltonian to be simulated. Note that the rules in the second and the fourth rows are defined based on *matrix addition* which is always commutative. As a result, exchanging the order of the *pauli_block*s in a *program* or the order of $\langle pauli_str, weight \rangle$s in a *pauli_block* will not change the semantics.

4 BLOCK-WISE INSTRUCTION SCHEDULING PASSES

The first step in Paulihedral is to schedule the blocks and the instructions within each block. Intuitively for two adjacent Pauli strings, more gates can be cancelled if they share the same non-identity operators on more qubits. Trying to maximize the number of shared operators between consecutive strings would be desirable. Also, it is possible to execute multiple blocks which have non-identity operators on disjoint sets of qubits in parallel and reduce the final circuit depth. In this paper, we present two block scheduling algorithms for two optimization objectives, reducing the total gate count or the circuit depth. We explain our block scheduling optimizations using the example in Figure 8. Suppose we will schedule 10 blocks on 8 qubits (Figure 8 (a)). In these blocks, each column is a Pauli string. The identity operators are omitted since they do not result in any circuit.

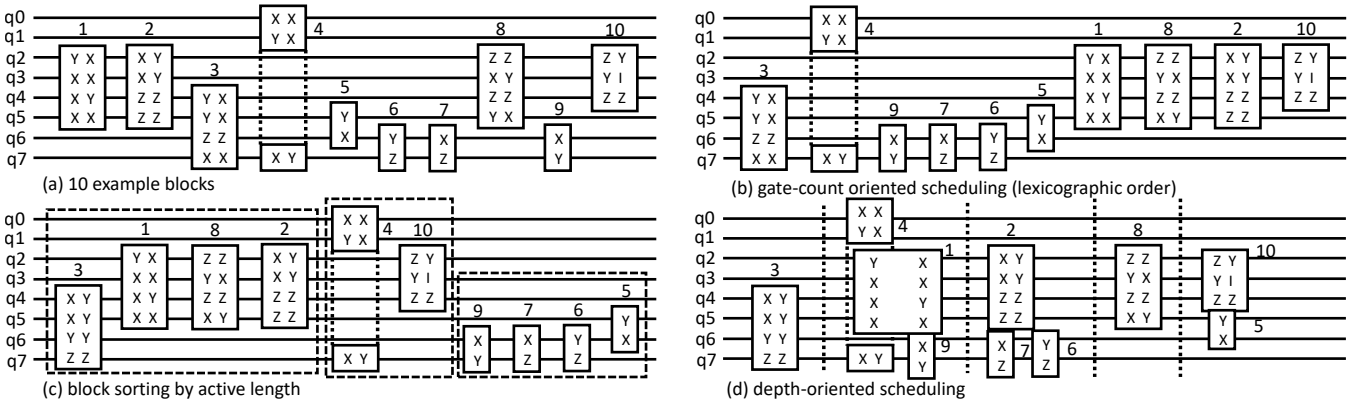

Figure 8: Example of block scheduling optimizations

4.1 Gate-Count-Oriented Scheduling

Lexicographic ordering of Pauli strings has been shown to be effective at enabling gate cancellation between them [23, 59]. Here we adapt this principle to the multi-string-per-block case for our gate-count-oriented scheduling algorithm. In the lexicographic order, the Pauli strings are scheduled in the alphabetical order. In Figure 8, we assume $X < Y < Z < I$ and use little-endian to lexicographically order from $q7$ down to $q0$. When a block has multiple strings, we first apply the lexicographic order on all strings in this block and then use the first string to represent this block when compared with other blocks. The first Pauli string can be representative because the strings in one block are usually mutually commutative in practical algorithmic constraints [18, 23, 45]. Two strings in a mutually commutative set can share the same operators on many qubits and all strings in one block are similar. Figure 8 (b) shows the result of gate-count-oriented scheduling.

4.2 Depth-Oriented Scheduling

The blocks can also be scheduled for reducing circuit depth. For example, in Figure 8, $q0$ to $q5$ are idle when executing block 9, 7, and 6. We may execute block 1 with them in parallel so that the overall circuit depth can be reduced. We propose a new depth-oriented block scheduling algorithm, whose pseudocode is in Algorithm 1. For the example in Figure 8, we first sort all blocks by the active length of the Pauli strings of the blocks in a decreasing order. The active length of a block is defined by the number of qubits which have a non-identity operator in at least one Pauli string of this block. This is an over-approximated estimation on how a block will occupy the qubits. The blocks of the same active length are ordered by the lexicographic order above. Figure 8 (c) shows the sorting result. Block 3, 1, 8, 2 have the largest active length of 4 so they are at the beginning. Block 9, 7, 6, 5 have the smallest active length of 2 and they are at the end.

Then we begin to schedule all blocks and put the blocks in different layers to increase the parallelism. For each layer, we first schedule a large active length block. Then we search for small active length blocks that can be executed in parallel with the large block. For the example in Figure 8 (d), we initialize the first layer by selecting the first block after the sorting. We place the block 3 at the beginning. Then we search for small blocks that have no overlapped active qubits with the large block and can be executed in parallel. There are no such small blocks for block 3 so we continue by start another layer with block 1. In this layer, block 4, 9, 7, 6 can be placed in parallel with block 1. We iterate over the sorted block and find the first few blocks that can padded in this layer. In this example, we select block 4 and 9. We also estimate the depth of these small blocks so that total depth of these blocks will not exceed the depth of the original large block in this layer. We repeat this padding process until we cannot find any new blocks that can be added in this layer. We then continue to the next layer and start with block 2 because its first Pauli string has the most overlapped Pauli operators with the Pauli strings at the end of the previous layer. We iterate until all blocks are scheduled. Figure 8 (d) shows the final result of our depth-oriented scheduling and we can expect

Algorithm 1: Depth-oriented scheduling

Input: List of Pauli blocks.
Output: Pauli Layers L.

```
Sort Pauli blocks by active-length-decreasing order, then
  sort blocks of the same active length by lexicographic
  order;
R = the set of all Pauli blocks remaining; L = ∅;
Initialize the first layer;
while R is not empty do
    next_block =
      arg max_{block∈R} Overlap(block, last Pauli layer);
    pauli_layer = [next_block]; R.remove(next_block);
    while total depth of the small padding blocks < the depth
      of next_block do
        find small Pauli block not overlapped with
          next_block;
        Append these blocks to pauli_layer;
        Remove these blocks from R;
    end
    L.append(pauli_layer);
end
```

that the circuit depth can be reduced even if we do not convert the program to the gates. This is another benefit of Pauli IR because the compiler can operate on a fairly compact description of the program. Once the program is lowered to gates, then the size blows up and parallelizing gates becomes much more expensive.

5 BLOCK-WISE OPTIMIZATION PASSES

In this section, we introduce two optimization passes that can exploit the gate cancellation potential created by our scheduling passes in the last section, and convert the Pauli IR programs to gate sequences with different optimization objectives onto the fault-tolerant quantum computer (FT) backend and the near-term superconducting quantum computer (SC) backend.

5.1 On the Fault-Tolerant Backend

Our strategy for the FT backend is to adaptively find the synthesis plan that can maximize gate cancellation since the mapping overhead can usually be neglected after applying quantum error correction [20]. The pseudocode is shown in Algorithm 2, and we explain it with Figure 9. To capture the major gate cancellation opportunities, we scan over all layered blocks and try to select consecutive layer pairs that share the most Pauli operators. There should be significant operator overlap between consecutive layers since this was considered in our scheduling. The blocks on the left of Figure 9 are in five layers. Layer 1, 2, 3, 5 have one block in each and layer 4 has two blocks. We will pair the layer 3 and 4 together first since they share the same Pauli operators on 6 qubits. Then the first two layers are paired since they share Pauli operators on only 2 qubits. The last layer is left alone.

We first realize gate cancellation between the paired layers. For all layer pairs, we synthesize the Pauli strings at the end of the first layer and the Pauli strings at the beginning the second layer in the pair. For the layer 3 (block 3) and the layer 4 (block 4 and 5), we need to handle the $IYXXYXXI$ in the layer 3 and ($IIIIYXXX$,$YYXXIIII$) in the layer 4. There are two sets of overlapped operators, YXX on qubit 3-1 and YXX on qubit 6-4. For each set, most gates can be directly cancelled, and we can select one qubit from each set and connect them with CNOT gates. The synthesis result for these two layers with gates cancelled is shown on the right of Figure 9. We repeat this process to optimize the synthesis of Pauli strings at the junction of two paired layers. Here we will synthesize the last string in layer 1 and the first string in layer 2.

We then realize the gate cancellation between strings inside a block. For those Pauli strings in the paired layer but not synthesized (one block with multiple strings), we employ a similar strategy at the string level for all Pauli strings inside one block. For each block, we search for string pairs that share the same Pauli operators on the most qubits and then synthesize these pairs first. In the block 1 in Figure 9, the first three Pauli strings are not yet synthesized. We will pair and synthesize the first two Pauli strings since they share 5 Pauli operators and a lot of gates can be cancelled. For the individual Pauli strings left, they are not paired with other strings (e.g., the third string in block 1). We check if it shares more Pauli operators with its left neighbor string or right neighbor string. Then we select the one with more gate cancellation and synthesize the Pauli string accordingly. For the blocks that are not paired with other blocks

Algorithm 2: Optimization for FT backend

Input: List of Pauli layers pls
Output: A quantum circuit of basic gates

```
pl_paired = []; // paired Pauli layer list
while neighboring layers exist in pls do
    i = argmax_{i∈IndexSet(pls)} Overlap(pl_i, pl_{i+1}) ;
    pls.remove(pl_i); pls.remove(pl_{i+1});
    pl_paired.append((pl_i, pl_{i+1}));
end
for (pl_1, pl_2) in pl_paired do
    ps_list_1 = last Pauli string of pl_1;
    ps_list_2 = first Pauli string of pl_2;
    analyze string overlap then do synthesis on
      (ps_list_1, ps_list_2);
    pl_1.remove(ps_list_1); pl_2.remove(ps_list_2);
    for pb in (pl_1 + pl_2) do
        most_overlap_sort(pb); // find overlap at
            Pauli-string-level
        analyze string overlap then do synthesis on the
          sorted strings in pb;
    end
end
for pb in pls do
    most_overlap_sort(pb); analyze string overlap then do
      synthesis on the sorted strings in pb;
end
```

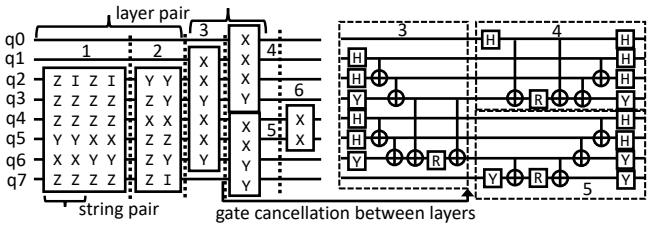

Figure 9: Example of compilation onto FT backend

at the beginning of this algorithm (e.g., block 6), we treat them as unsynthesized Pauli strings and apply the same strategy, pairing and synthesizing the strings with high gate cancellation potential first then dealing with individual strings. Finally, all Pauli strings are compiled and we obtain a gate sequence of the input Pauli IR program. The final gate count is substantially reduced because the gate cancellation potential created by our block scheduling passes is maximally exploited through the adaptive synthesis plan in our block-wise optimization pass.

5.2 On the Near-Term Superconducting Backend

The compilation is more complicated for the SC backend because the SWAP gates are necessary to change the qubit mapping due to the qubit connectivity constraints. The gates are not uniform as they have different error rates on different qubits. We assume that the device calibration information (qubit coupling graph and

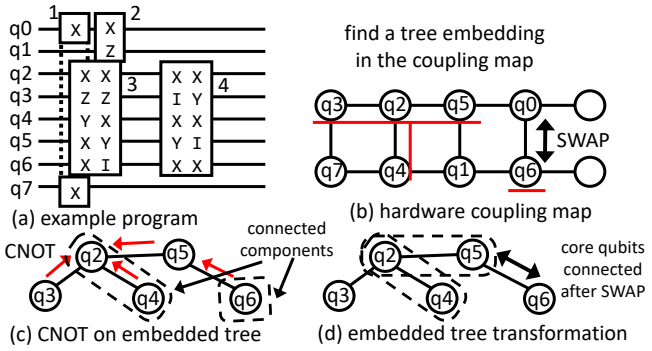

Figure 10: Example of compilation onto SC backend

the gate error rates on each qubit and qubit pair) is provided by the vendor. The major objective on the SC backend is to reduce the mapping overhead.

Our key idea is to find a tree embedding in the coupling map that can support the Pauli strings in the current layer and also minimize the mapping transition overhead between layers. Algorithm 3 shows the pseudocode, and we explain it using the example in Figure 10. For the initial qubit layout, we map all qubits to the most connected subgraph in the device coupling map. Suppose the coupling map and the current mapping of Figure 10 (b). We then begin to generate the simulation circuits and insert SWAPs for the blocks that appear in the critical path. In our block scheduling, we have already placed the blocks in different layers. In each layer, the largest block (involving the most qubits) is most likely on the critical path. Our optimization pass will first process the largest block in each layer, followed by the small blocks remaining. The program in Figure 10 (a) has two layers in which block 3 and 4 are the largest blocks.

For each block, we first select a root qubit. We define that the core qubit list of a block contains the qubits which have a non-identity operator on all Pauli strings in the block (e.g., $q2$-5 for block 3, q(2,4,6) for block 4). For block 3, since it is the first layer, we only need to consider itself. For $q2$-5 in its core list, they are already in a connected subgraph (Figure 10 (b)). We select any one of them (e.g., $q2$) as the root. And we only need to attach $q6$ to this subgraph by connecting it to any node of this graph. Suppose we swap $q6$ with $q0$ and now all active qubits in this block are connected in a subgraph. Active qubits are those qubits that have a non-identity operator in at least one string in this block. We can naturally generate an embedded tree from the coupling map (Figure 10 (c)).

Next we can synthesize the strings in block 3. The key idea is to naturally implement the CNOT tree in the Pauli circuits on the embedded tree so that we do not need to insert SWAPs for all individual CNOTs. We generate CNOT gates and single-qubit gates from the outermost qubits to the root for all the Pauli strings in the current block. If a qubit is active in the current Pauli string, we will check if its parent node is also active in the current Pauli string. If so, we insert a CNOT between the qubit and its parent. Otherwise, we swap it with its parent so that the qubit can get closer to the root and will be connected by CNOT later. In Figure 10 (c), the generated CNOTs are labeled by red arrows. After we determine the left CNOT tree, the right CNOT tree can be generated by reversing the order of CNOTs in the left tree.

After we process block 3, we will compile block 4, the next block in the critical path. As our block scheduling passes tend to maximize the overlap between two consecutive layers, the core lists of two consecutive layers are similar. For example, q(2,4,6) are in the core list of block 4 and they all appear in the core list of block 3. We evaluate all these qubits to select the root qubit with the largest connected component (within the core list) in the current mapping (Figure 10 (c)) to minimize the transition overhead. For q(2,4,6), we will select $q2$ or $q4$ since they are in a size-2 connected component while $q6$ is in a size-1 connected component. Similarly, we then move all other active qubits to the tree through the path with the

Algorithm 3: Optimization for SC backend

Input: List of Pauli layers *pls*, device information
Output: Hardware compatible circuit Q

```
Map logical qubits to the most connected subgraph of the
    device coupling map; // Initial mapping
for pauli_layer in pls do
    pb = the largest Pauli block in pauli_layer;
    s = core qubit list of pb;
    T1 = node in s with largest connected component;
    connect active qubits in pb to tree T1 through shortest
     path (lowest error rate);
    wl =leaves of T1 sorted by depth;
    for ps in pb do
        while wl ≠ ∅ do
            n = wl.deque(); np = n.parent;
            if n is the root of T1 then continue;
            if ps[n] ≠ I and ps[np] ≠ I then
                add single-qubit gates based on ps[n] and
                 ps[np]; Q.append(CNOT(n, np));
            else if ps[n] ≠ I and ps[np] == I then
                Q.append(SWAP(n, np));
            end
            wl.append(np);
        end
        generate the right half circuit of ps reversely;
    end
    for spb in remaining blocks of pauli_layer do
        T2 = try_construct_tree(spb); // Return NULL if
             changes T1
        synthesize spb with T2 if T1 not changed; otherwise
         add spb to remain_layers;
    end
end
while remain_layers is not empty do
    Sort remain_layers by cumulative distance between
     active qubits;
    Synthesize first layer of remain_layers with the same
     strategy and remove it from remain_layers;
end
```

smallest error rate estimated by the device information. Here we select $q2$ as the root and then swap $q6$ and $q5$ to transit from block 3 to 4 with only 1 SWAP (Figure 10 (d)). After that the core qubits in block 4 are connected and we can begin synthesizing all strings in block 4.

The procedure above is to process the largest blocks in each layer. For other small blocks in the same layer, we follow a similar strategy and attempt to construct the trees for active qubits in those small blocks. If the trees of the small blocks do not affect the tree construction of the large block, we just process the small blocks in parallel with the large block since they will not affect each other. This will create parallelism and reduce the depth of the generated circuit. For example, block 2 and 3 can be processed in parallel because $q0$ and $q1$ are connected after swapping $q6$ with $q0$. However, if the trees of the small blocks affect the processing of the large block, we will put it in *remain_layer* and process them at the end. For example, block 1 will be in the *remain_layer* because connecting $q0$ and $q7$ will affect block 3.

After we process all the large blocks in the critical path and those small blocks that can executed in parallel, we will compile the blocks in the *remain_layer*. The order of processing these blocks is determined by whether the active qubits are close in the current mapping. We compute the cumulative distance between active qubits in a block and then compile the block with the smallest cumulative distance and update the qubit mapping. This process is repeated until all the blocks are processed.

6 EVALUATION

In this section, we evaluate Paulihedral by comparing with state-of-the-art baselines, analyze the effects of individual passes, and perform real system study.

6.1 Experiment Setup

Backend: The optimizations in this paper target two different backends, the fault-tolerant backend (FT) and near-term superconducting backend (SC). We will cover both of them. We select IBM's latest 65-qubit Manhattan architecture [11] as the SC backend. For real system study, we use IBM's 16-qubit Melbourne chip, the largest publicly available one.

Metric: We use the CNOT/single-qubit gate count, and the circuit depth in the post-compilation program to evaluate Paulihedral. For the SC backend, the CNOT gate count is more important due to its higher error rate and latency. The depth is also important due to short qubit coherence times. For the FT backend, T gate is usually more expensive but for the simulation kernels, the ratio between the H, Y, CNOT gate count and the T gate count grows linearly as the number of qubits increases. Because a Pauli string of length n will have $O(n)$ H, Y, and CNOT gates but the number of Rz gates (the only source of T gates) is always one. It has also been shown that CNOT count is a significant cost in fault-tolerant algorithms and should not be neglected compared to T gates [36]. Hence, we estimate the performance with total gate count and circuit depth, following convention in previous work [13, 14, 25, 61].

Benchmark: We select 31 benchmarks of different sizes and various applications. For the SC backend, we select VQE UCCSD ansatzes [45] of six sizes, and the QAOA programs [18] for graph max-cut on regular (REG) graphs of degrees 4, 8, 12, and random (Rand) graphs of edge probability 0.1, 0.3, 0.5, as well as traveling salesman problem (TSP) of different sizes. These benchmarks are generated by Qiskit [1]. For the FT backend, we first generate the Hamiltonians of five molecules using PySCF [54] (N_2, H_2S, MgO, CO_2, NaCl). We also prepare the Hamiltonians of three Ising models and three Heisenberg models, both of which are widely used in condensed matter physics, of different dimensions. We finally generate random Hamiltonians (Rand) of various sizes (30 to 80 qubits) for a more comprehensive evaluation. For a Hamiltonian of n qubits, we prepare $5n^2$ Pauli strings. In each Pauli string, we first randomly select one integer m between 1 and n. Then we randomly select m qubits and assign random Pauli operators to them. The rest $n - m$ qubits will be assigned with the identity. Table 1 shows the details of these benchmarks. Note that 'Pauli #' represents the number of Pauli strings. We include the CNOT and single-qubit gate counts when naively converting these benchmarks into gates without any optimization/transformations, and neglecting mapping overhead.

Table 1: Benchmark information

Backend	Type	Name	Qubit #	Pauli #	CNOT #	Single #
SC	UCCSD	UCCSD-8	8	144	1134	1240
		UCCSD-12	12	1476	16192	15588
		UCCSD-16	16	4200	56558	47044
		UCCSD-20	20	8316	132326	109248
		UCCSD-24	24	9300	146312	115584
		UCCSD-28	28	20724	353984	270196
	QAOA	REG-20-4	20	40	80	40
		REG-20-8	20	80	160	80
		REG-20-12	20	120	240	120
		Rand-20-0.1	20	18	37	18
		Rand-20-0.3	20	56	113	56
		Rand-20-0.5	20	93	187	93
		TSP-4	16	112	192	112
		TSP-5	25	225	400	225
FT	Ising	Ising-1D	30	29	58	29
		Ising-2D	30	49	98	29
		Ising-3D	30	59	118	59
	Heisenberg	Heisen-1D	30	87	174	319
		Heisen-2D	30	147	294	539
		Heisen-3D	30	177	354	649
	Molecule	N_2	20	2951	39594	32151
		H_2S	22	4582	66026	52686
		MgO	28	24239	388258	310519
		CO_2	30	16154	252402	202282
		NaCl	36	67667	1249768	945935
	Random	Rand-30	30	4500	132939	99123
		Rand-40	40	8000	316039	229240
		Rand-50	50	12500	618763	441532
		Rand-60	60	18000	1068153	754071
		Rand-70	70	24500	1699771	1190101
		Rand-80	80	32000	2540640	1768117

Implementation: We prototype Paulihedral in Python 3.8 (denoted by 'PH'). The entire compilation flow has two stages. The first stage is the quantum simulation program optimizations. The baselines include the CQC t|ket⟩ compiler [50] which employs the simultaneous diagonalization [13, 14, 17], a popular technique for optimizing quantum simulation programs ('TK'), and the QAOA compiler [3–5], an algorithm-specific compiler for unconstrained optimization QAOA on graphs ('QAOA compiler'). The second stage is the generic compilation and we have two industry generic compilers, the IBM's Qiskit [1] at the highest optimization level 3 ('Qiskit_L3') and the CQC t|ket⟩ generic compiler [50] at the highest optimization level 2 ('tket_O2'), The experiments are performed on a server with a 28-core Intel Xeon Platinum 8280 CPU and 1TB

Table 2: Compilation time and results compared with t|ket⟩ [50]

	Time(s)		PH+Qiskit_L3				Time(s)	PH+tket_O2				Time(s)		TK+Qiksit_L3				Time(s)	TK+tket_O2			
	PH	Qiskit	CNOT	Single	Total	Depth	tket	CNOT	Single	Total	Depth	TK	Qiskit	CNOT	Single	Total	Depth	tket	CNOT	Single	Total	Depth
UCCSD-8	0	9	1165	667	1832	1404	16	1282	538	1820	1415	0	17	2247	1188	3435	1975	1	1775	346	2121	1527
UCCSD-12	1	120	17322	8607	25929	17808	485	20794	6816	27610	19439	0	331	35130	16113	51243	28910	31	26713	4007	30720	21747
UCCSD-16	3	471	58441	27607	86048	55566	7300	77274	19157	96431	65749	2	1525	140932	55678	196610	107177	179	106868	11837	118705	87064
UCCSD-20	8	827	106464	54951	161415	97168	11033	128195	37972	166167	107750	4	3171	315911	123823	439734	235862	655	223106	23736	246842	179618
UCCSD-24	9	1235	126013	68258	194271	113293	33155	187653	40357	228010	156431	6	4015	383444	145586	529030	285638	944	260282	29101	289383	213557
UCCSD-28	25	2759	280514	145672	426186	258326	69940	473093	92118	565211	386285	17	8933	870722	326600	1197322	644877	3313	595631	58782	654413	482484
REG-20-4	0	1	329	108	437	161	2	311	43	354	128	0	0	1594	523	2117	762	3	1441	40	1481	710
REG-20-8	0	2	519	266	785	290	7	574	84	658	267	0	0	1985	663	2648	894	3	1726	80	1806	816
REG-20-12	0	2	694	393	1087	396	12	842	125	967	404	0	0	1893	705	2598	765	3	1675	120	1795	775
Rand-20-0.1	0	1	188	46	234	97	1	167	20	187	67	0	0	577	184	761	255	0	448	18	466	214
Rand-20-0.3	0	1	414	181	595	236	5	439	60	499	202	0	0	1539	502	2041	688	2	1355	56	1412	644
Rand-20-0.5	0	2	571	313	884	335	9	674	98	772	333	0	0	1812	622	2434	735	3	1614	93	1707	746
TSP-4	0	2	500	381	881	257	8	583	112	695	283	0	0	1006	556	1562	446	2	1207	112	1319	530
TSP-5	0	3	1093	683	1776	636	24	1457	225	1682	534	0	0	2886	1250	4136	1030	9	3057	225	3282	1249
N_2	6	255	16632	11223	27855	17486	37	16002	10861	26863	17682	2	116	19762	9785	29547	21087	15	18928	9702	28630	20446
H_2S	16	450	25726	17580	43306	27676	94	24813	16836	41649	28051	3	209	35248	17453	52701	36450	47	33968	17075	51043	35329
MgO	615	2489	116973	87654	204627	126259	3291	113575	78585	192160	127619	20	1840	198428	92012	290440	203064	3818	192499	85694	278193	194905
CO_2	293	1656	96829	65243	162072	93679	2663	93391	61386	154777	95302	15	1182	126634	58368	185002	129697	1699	121768	56277	178045	124557
NaCl	7377	7492	316456	247630	564086	338763	28920	307038	223261	530299	342775	63	7262	626692	267424	894116	626665	87838	605201	247500	852701	599322
Ising-1D	0	0	58	29	87	6	0	58	29	87	6	0	2	508	29	537	450	0	508	29	537	450
Ising-2D	0	0	98	49	147	18	0	98	49	147	18	0	1	306	49	355	219	0	306	49	355	219
Ising-3D	0	0	118	59	177	18	0	118	59	177	18	0	1	290	59	349	188	0	290	59	349	188
Heisen-1D	0	1	87	204	291	13	0	87	190	277	13	0	1	172	176	348	100	0	169	200	369	126
Heisen-2D	0	1	216	315	531	43	0	212	284	496	47	0	1	293	239	532	89	0	293	294	587	98
Heisen-3D	0	2	305	366	671	65	0	295	335	630	67	0	2	365	271	636	118	0	364	328	692	125
Rand-30	13	1386	94222	47315	141537	70787	2329	89152	50490	139642	78429	5	779	114043	55665	169708	87196	141	108943	62158	171101	97897
Rand-40	42	4253	233266	108988	342254	166989	175842	223946	119225	343171	189856	11	1775	270928	125264	396192	199567	961	259985	142805	402790	227284
Rand-50	104	9144	470240	211605	681845	329388	55853	455607	234635	690242	379553	24	3575	533631	239379	773010	388722	5220	514235	276401	790636	447315
Rand-60	203	14007	834418	364816	1199234	575973	>72hrs	N.A.				44	5930	926982	404279	1331261	666338	22096	895895	471005	1366900	768714
Rand-70	393	27849	1345439	576378	1921817	924730	>72hrs	N.A.				74	9672	1476952	636284	2113236	1050970	138031	1430214	746249	2176463	1220344
Rand-80	709	39322	2033283	856105	2889388	1386577	>72hrs	N.A.				116	17092	2205935	938304	3144239	1555808	285101	2138232	1104321	3242553	1811440

RAM. Note that due to the limited representation ability of t|ket⟩, the algorithmic constraints are hard to be encoded in 'TK'. To run our experiments and perform a fair comparison at our best, we relax those constraints in 'TK' and this relaxation allows a larger optimization space.

6.2 Comparing with t|ket⟩ and the QAOA Compiler

Table 2 shows the compilation time and results of the four configurations of all benchmarks on the two backends. Note that '>72 hrs' indicates that the 'tket_O2' takes over 72 hours and was shut down in the middle. In summary, 'PH' outperforms 'TK' with substantial gate count and circuit depth reduction while only introducing ~ 5% additional time ('PH' vs 'Qiskit/tket') in the entire compilation flow.

On the SC backend, 'PH' can reduce the CNOT, single-qubit, total gate count, and circuit depth by 66.2% (43.3%), 53.4% (-22.7%), 62.6% (41.2%), and 60.8% (44.3%), respectively on average, compared with 'TK' using 'Qiskit_L3' ('tket_O2') generic compilation. 'PH' can achieve such significant improvement because 'TK' does not support mapping-aware optimization for general Pauli strings and can only do a inefficient generic qubit mapping. The single-qubit gate count increases when using 'tket_O2' but this does not affect the overall improvement since the CNOT gates have much higher error rates on the SC backend and latency and the total single-qubit gate count is still relatively low.

On the FT backend, 'PH' can reduce the CNOT, single-qubit, total gate count, and circuit depth by 38.7% (44.5%), 18.6% (3.0%), 32.8% (34.4%), and 61.7% (68.0%), respectively on average, compared with 'TK' using 'Qiskit_L3' ('tket_O2'). The circuit depth reduction is significant due to the depth-oriented scheduling in 'PH'. Our block-wise optimization is also much effective compared with 'TK' strategy. The details of 'TK' are not public and what we can infer, at our best, from their limited documents [13, 14, 17, 50] is that the simultaneous diagonalization may introduce too much overhead. For example, the 'Ising-1D' program has even more gates after 'TK'. One possible reason is that all Pauli strings in Ising-1D are mutually commutative and it takes many additional gates to simultaneously diagonalize all these Pauli strings.

Table 3: Comparing with QAOA compiler [3]

	PH+Qiskit_L3					QAOA_Compiler+Qiskit_L3				
Benchmark	CNOT	Single	Total	Depth	Time(s)	CNOT	Single	Total	Depth	Time(s)
REG-20-4	329	108	437	161	0.14	394	101	495	171	6.32
REG-20-8	519	266	785	290	0.23	727	141	868	297	10.27
REG-20-12	694	393	1087	396	0.29	1020	181	1201	399	14.55
Rand-20-0.1	188	46	234	97	0.08	212	80	292	111	4.52
Rand-20-0.3	414	181	595	236	0.1	546	118	664	230	7.74
Rand-20-0.5	571	313	884	335	0.12	842	155	997	334	12.3

Table 3 shows the compilation results of 'PH' and the QAOA compiler [3] on the 6 MaxCut problems. We ran the QAOA compiler with 20 random seeds for each program and collected the averaged compilation results. Comparing with the QAOA compiler, Paulihedral can achieve 24.6%, 12.2%, and 3.2% reduction in CNOT count, total gate count, and circuit depth, respectively on average, using only 1.7% compilation time. The overhead is about 40% in single-qubit gate count, but in QAOA the CNOT count is usually over 3−4× higher than single-qubit gate count and CNOT error rate is usually 10× higher on the SC backend. Therefore, 'PH' significantly outperforms QAOA compiler, even though it is more general purpose and not tailored to a single algorithm. This is because 'PH' employs a block-wise optimization for searching SWAPs and the search scope is much larger than that of the QAOA compiler's greedy search.

6.3 Pass Option Comparison

Now we study the effect of different pass options in Paulihedral. We first compare the two block scheduling passes.

Table 4: Pass option effect comparison

	DO vs GCO				Block-Wise Compilation improvement			
	CNOT	Single	Total	Depth	CNOT	Single	Total	Depth
UCCSD-8	-4.43%	-1.19%	-3.27%	0.72%	-51.07%	-51.14%	-51.09%	-37.93%
UCCSD-12	6.32%	-8.41%	0.93%	2.37%	-57.38%	-55.35%	-56.73%	-50.06%
UCCSD-16	-2.62%	-6.32%	-3.84%	-1.94%	-59.61%	-45.26%	-55.90%	-51.89%
UCCSD-20	-3.81%	-8.55%	-5.48%	-6.49%	-72.40%	-56.43%	-68.47%	-67.66%
UCCSD-24	-5.25%	3.90%	-2.23%	-10.60%	-68.72%	-49.02%	-63.80%	-63.38%
UCCSD-28	-1.19%	5.00%	0.84%	-2.34%	-73.97%	-56.31%	-69.80%	-67.22%
REG-20-4	N.A.				-21.85%	-6.90%	-18.62%	27.78%
REG-20-8	N.A.				-34.63%	1.14%	-25.73%	18.37%
REG-20-12	N.A.				-36.62%	-12.86%	-29.69%	13.79%
Rand-20-0.1	N.A.				-11.74%	-16.36%	-12.69%	36.62%
Rand-20-0.3	N.A.				-28.99%	-2.16%	-22.53%	26.20%
Rand-20-0.5	N.A.				-38.60%	-4.28%	-29.67%	10.20%
TSP-4	N.A.				-43.12%	-12.81%	-33.05%	-19.18%
TSP-5	N.A.				-47.53%	-7.58%	-37.07%	-2.30%
N_2	13.30%	3.13%	8.97%	-7.59%	-4.54%	-2.93%	-3.90%	-6.77%
H_2S	17.43%	6.24%	12.61%	-3.09%	-6.32%	-2.94%	-4.98%	-8.25%
MgO	31.16%	8.51%	20.40%	-0.48%	-6.89%	-8.17%	-7.44%	-9.45%
CO_2	26.08%	6.43%	17.36%	-8.84%	-5.13%	-1.34%	-3.64%	-6.04%
NaCl	25.03%	6.55%	16.18%	-5.46%	-12.18%	-8.48%	-10.60%	-13.62%
Ising-1D	0.00%	0.00%	0.00%	-93.10%	0.00%	0.00%	0.00%	0.00%
Ising-2D	0.00%	0.00%	0.00%	-68.42%	0.00%	0.00%	0.00%	0.00%
Ising-3D	0.00%	0.00%	0.00%	-71.43%	0.00%	0.00%	0.00%	0.00%
Heisen-1D	0.00%	0.00%	0.00%	-92.57%	0.00%	7.37%	5.05%	0.00%
Heisen-2D	-19.10%	-12.01%	-15.04%	-82.30%	0.00%	3.28%	1.92%	0.00%
Heisen-3D	-8.41%	-13.68%	-11.36%	-80.83%	0.00%	1.95%	1.05%	0.00%
Rand-30	7.25%	6.95%	7.15%	-9.76%	-8.74%	-3.53%	-7.06%	-29.45%
Rand-40	6.11%	5.55%	5.93%	-9.46%	-9.68%	-2.99%	-7.65%	-31.80%
Rand-50	4.83%	4.98%	4.88%	-9.21%	-10.48%	-2.19%	-8.06%	-33.15%
Rand-60	4.10%	4.36%	4.18%	-9.30%	-10.75%	-1.90%	-8.23%	-33.44%
Rand-70	3.60%	3.79%	3.66%	-8.80%	-11.07%	-1.63%	-8.44%	-33.76%
Rand-80	3.27%	3.34%	3.29%	-8.70%	-11.19%	-1.54%	-8.54%	-34.17%

DO vs GCO scheduling: On the left of Table 4 we show the difference between the depth-oriented (DO) scheduling and the gate-count-oriented (GCO) scheduling (in Section 4). Overall, across the 17 benchmarks on the FT backend, 'DO' can yield low-depth circuits while 'GCO' can reduce the gate count more. The circuit depth of DO is 46.7% (geomean) compared with that of GCO and the gate count overhead is 5.9%, 0.64%, and 3.3% for CNOT, single-qubit, and total gate count, respectively. For benchmarks on the SC backend, the effect of the block scheduling is largely amortized by mapping overhead reduction since the tested Manhattan architecture has very sparse qubit connection. For the UCCSD benchmarks, 'DO' and 'GCO' share similar overall performance. For the QAOA benchmarks, there is no difference between 'DO' and 'GCO' since the entire kernel has only one block.

BC improvement: Our block-wise compilation (BC) passes (in Section 5) can significantly reduce the gate count and circuit depth. On the right of Table 4 we show the comparison between using BC against a naive synthesis and Qiskit_L3. For the 17 benchmarks on the FT backend, BC reduces the circuit depth, the CNOT, single-qubit, and total gate counts by 15.5%, 6.0%, 3.1%, and 5.0%, respectively. On the SC backend, the BC pass is even more effective since the large mapping overhead can be greatly reduced. For the UCCSD (QAOA) benchmarks, BC can reduce the CNOT, single-qubit, total gate count, and circuit depth by 60% (33%), 45% (8%), 56% (26%), and 53% (−14%), respectively on average. The circuit depth of QAOA benchmarks is increased since the BC focus more on SWAP reduction, leading to fewer gates but deeper circuits because the effect of SWAP reduction is relatively limited in the small-size QAOA benchmarks.

Pauli string pattern effects: It can be observed that the effect of the passes vary on different benchmarks. The reason is that the Pauli strings in the benchmarks have different patterns which can be classified into two categories based on the numbers of non-identity operators in each Pauli string. As mentioned in Section 2, a Pauli string with more non-identity operators on more qubits will in general be converted to a larger circuit block involving more qubits and gates. The first category includes the molecule Hamiltonians, the random Hamiltonians, and the UCCSD. In these Hamiltonians, many Pauli strings have non-identity operators on various numbers of qubits (up to all qubits). The second category includes the Ising, Heisenberg, and the selected QAOA benchmarks, of which the Hamiltonians only have Pauli strings with non-identity operators on at most two qubits. Such a difference in the operator distribution affects the compilation results.

On the FT backend, benchmarks in the first category (molecule and random Hamiltonians) benefit more from the BC optimizations since Pauli strings with more non-identity operators have larger potential in gate cancellation and depth reduction. Benchmarks in the second category (Ising and Heisenberg) cannot benefit from BC since those Pauli strings with only two non-identity operators can only be synthesized in a single way and there is no space BC can explore to further reduce the gate count and circuit depth. However, these benchmarks can benefit a lot from DO. GCO turns out to be inefficient in both gate count and circuit depth for them because GCO cannot create gate count reduction while DO can create additional single-qubit gate reduction opportunities between consecutive layers by putting many small-size blocks in one layer. On these benchmarks, DO completely outperforms GCO with on average 84.2% circuit depth reduction and 7.5% total gate count reduction. Similarly on the SC backend, the BC improvement on the UCCSD benchmarks (first category) is also more significant compared with the QAOA benchmarks (second category) because more gate can be cancelled and more SWAPs in the mapping overhead can be eliminated when the tree sizes are large for Pauli strings with more non-identity operators.

6.4 Pass Benefit Breakdown

To show the separate effect of scheduling passes and optimization passes, we prepare breakdown experiments on four benchmarks (two random Hamiltonian RAND-40 and RAND-50, two molecule Hamiltonians N_2 and H_2S). Their information is in Table 1. We have four configurations by combining two ordering options and whether to apply block-level optimizations. The baseline order is the original order of the problem Hamiltonians, which for RAND-40 and -50 is random and for N_2 and H_2S is determined by the Hamiltonian generation module in PySCF [54] (which is based on ordering the electron orbitals from low energy to high energy level). The DO order is the depth-oriented order in Section 4.2. 'BC' means that the block-level optimization in Section 5.1 is applied. All configurations are followed by Qiskit Level 3 optimization by default. The following table shows the CNOT gate count, single-qubit gate count, circuit depth, and their corresponding reduction percentage of each configuration compared to the 'Baseline order'.

In Table 5 we can find that both the DO order and the BC have substantial contributions to the final CNOT/single-qubit gate count

Table 5: Benefit breakdown of the block ordering and block-wise optimization passes

Benchmark	Metric	Baseline order	Baseline order + BC	DO order	DO order + BC
RAND-40	CNOT	313011	279165 (-10.8%)	259477 (-17.1%)	234264 (-25.1%)
	Single	157853	149234 (-5.5%)	143686 (-9.0%)	131302 (-16.8%)
	Depth	275175	188267 (-31.6%)	245675 (-10.7%)	168003 (-38.9%)
RAND-50	CNOT	613714	548499 (-10.6%)	529220 (-13.8%)	474999 (-22.6%)
	Single	300955	283647 (-5.8%)	279465 (-7.1%)	255104 (-15.2%)
	Depth	529844	360004 (-32.1%)	490793 (-7.4%)	329169 (-37.9%)
N_2	CNOT	36484	23371 (-35.9%)	17423 (-52.2%)	15981 (-56.2%)
	Single	20124	14713 (-26.9%)	11557 (-42.6%)	11366 (-43.5%)
	Depth	42525	25216 (-40.7%)	18749 (-56.0%)	16788 (-60.5%)
H_2S	CNOT	61127	38213 (-37.5%)	27752 (-54.6%)	24792 (-59.4%)
	Single	32885	24173 (-26.5%)	18154 (-44.8%)	17307 (-47.4%)
	Depth	70618	40985 (-42.0%)	30430 (-56.9%)	26581 (-62.4%)

and circuit depth reduction. Averaging over the results of the four selected benchmarks, the effect of BC is 69% of that of DO order scheduling in CNOT reduction, 65% in single-qubit gate reduction, and 1.62x in circuit depth reduction.

6.5 Real System Study

Finally, we evaluate 'PH' on IBM's 16-qubit Melbourne chip with 8 QAOA MaxCut programs. We generate 4 regular graphs of 7 to 10 nodes with 4 edges per node ('REG-n(7-10)-d4'), and 4 random graphs of 7 to 10 nodes with edge probability 0.5 ('RD-n(7-10)-p0.5'). We prepare 1-level QAOA circuits on these graphs and then optimize the parameters in the simulator. Those circuits with the optimized parameters are then evaluated on the Melbourne chip (40960 shots per circuit). The baseline is 'Qiskit_L3' with the Pauli strings ordered by iterating over the adjacency matrix (Qiskit default configuration).

Figure 11 shows the improvement of the success probability after applying 'PH' optimizations. The 'Estimated Success Probability' (ESP), a widely used metric in guiding the compiler optimization [39, 44, 57], is a theoretical estimation of the success probability based on the program and the hardware noise model. The 'Real System Success Probability' (RSP) is the number of trials with correct measurement results divided by the total number of trials when executing on the real machine. Applying 'PH' can improve the ESP by 2.11× on average (up to 3.00×) based on the noise model of the tested device, by reducing the CNOT count and circuit depth by 15.1% and 36.2%, respectively on average. On the real machine, 'PH' can improve the RSP by 1.24× on average (up to 1.87×). There is a gap between the results from ESP and RSP because the noise model only provides limited hardware information. We expect that the compilation can be further improved with more detailed hardware models.

Table 6 shows the detailed compilation results onto the IBM's 16-qubit Melbourne chip It can be observed that Paulihedral optimization leads to significant reduction in both gate count and circuit depth. More importantly, the reduction grows as the benchmark size increases, showing that Paulihedral will be more effective on larger size input programs. Such reduction can be turned into the final success probability improvement. We show the absolute success probabilities and the relative improvement in the second table. We can also observe that there is a significant difference between regular graphs and random graphs. This is because regular graphs have some symmetries and the solution space (the number

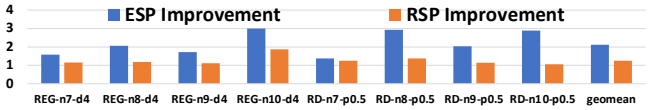

Figure 11: Success Probability Improvement for QAOA on IBM's 16-qubit Melbourne Chip

Table 6: Detailed compilation results onto IBM Melbourne

	Qiskit				Paulihedral+Qiskit			
Metric	CNOT	Single	Depth	RSP	CNOT	Single	Depth	RSP
REG-n7-d4	45	218	86	23.81%	43	131	56	27.50%
REG-n8-d4	78	202	100	10.97%	66	170	78	12.89%
REG-n9-d4	86	247	113	22.05%	67	124	70	24.69%
REG-n10-d4	122	238	114	10.22%	70	134	72	19.08%
RD-n7-p0.5	35	149	64	0.67%	40	82	39	0.85%
RD-n8-p0.5	68	166	102	0.09%	61	94	59	0.13%
RD-n9-p0.5	76	209	107	0.37%	71	126	67	0.42%
RD-n10-p0.5	141	314	174	0.08%	93	211	106	0.09%

of valid solutions) is much larger than that of random graphs (that is, the number of valid solutions are larger). This makes QAOA on regular graphs much more noise tolerant. The low absolute success probability comes from the fact that the Melbourne chip is an old device with higher error rates. But it is the only publicly accessible chip with 14 qubits and all other public devices are of 5-7 qubits.

7 DISCUSSION

It would be always desirable to have more effective quantum compiler optimizations to fully exploit the potential of quantum computing. One common approach is to model the hardware more precisely (e.g., from coarse-grained gate count [30, 49, 64] to independent non-uniform gate error [39, 58], then correlated crosstalk error [41], and finally low-level pulse optimizations [12, 22]). The compiler can naturally exploit more potential from the hardware with more detailed hardware information.

Different from these compiler innovations that are mostly driven by the underlying technologies, Paulihedral takes another approach which is to enable deeper compiler optimizations by leveraging the algorithmic properties of the high-level quantum programs. Relatively little attention has been given to this direction because 1) it is exceedingly difficult to extract useful high-level semantics from the gate-sequence representation in today's compiler infrastructures, and 2) scalable yet effective static analysis of quantum programs is also very hard as the size of the operation matrices grows exponentially with the number of qubits. We believe that these are two critical yet difficult open problems in the future development of quantum compiler/software infrastructure since they prevent the compiler from automatically detecting high-level and large-scale optimization opportunities.

Paulihedral tackles these two problems for the quantum simulation kernel, a widely used subroutine, and thus can benefit the compiler optimization for many quantum algorithms. In particular, we define a new Pauli IR which can capture the high-level semantics of simulation kernels. The domain knowledge of quantum simulation can thus be exploited by the compiler automatically, yielding optimizations that are hard to be implemented in the conventional

gate-based representation. We then design several new compiler passes, all of which are scalable block-wise circuit transformations since the analysis on Pauli strings can be efficiently handled by classical computers. The evaluation in this paper has covered a wide range of quantum simulation kernels and we expect that Paulihedral will continue to benefit future quantum algorithms since the quantum simulation has been a long-living algorithm design principle in the last few decades.

Looking forward, although Paulihedral is designed from an algorithmic perspective, it can incorporate those technology-driven optimizations. In this paper, we have supported two different backends with two technology-dependent optimization passes targeting different objectives and hardware constraints. These passes can also be further optimized once we have a deeper understanding of the quantum devices and come up with more comprehensive hardware models. Paulihedral can be further extended to other quantum architectures (e.g., ion-trap-based architectures [40, 63], photonics [7]) by adding new optimization passes.

It is also possible to make Paulihedral more intelligent by automatically managing the passes based on the input program characteristics. Currently Paulihedral has four passes and we have already observed that the different Pauli string patterns can affect the final improvement under different pass configurations as discussed in Section 6. In the future, more passes can be included to cover more backends, error resources, architectural constraints, and optimization objectives. How to automatically select the most suitable combination of passes from a pool of compiler passes is worth to explore.

Finally, the idea of quantum algorithmic compiler can be extended to other promising quantum algorithm domains. There are several other important common techniques in quantum algorithm design (e.g., quantum phase estimation [43], amplitude amplification [8]) and promising quantum application domains (e.g., quantum machine learning [34]). How to design new programming languages to maintain the high-level semantics of these programs and then propose corresponding algorithmic compiler optimizations is still an open problem which can be left as future work.

8 RELATED WORK

Paulihedral is a compiler framework with a new IR abstraction and deeper optimizations for general quantum simulation kernels. We first review the program representation and optimizations in quantum compilers. Then we discuss existing optimizations for quantum simulation programs.

IR in quantum compilers: Modern classical compilers employ multiple IRs (e.g., control flow graph, static single assignment) from high level to low level and different optimizations are applied on different IRs. Today's quantum compilers [1, 6, 27, 38, 50], on the other hand, are mostly built around low-level representations [15, 28, 51], which makes it difficult to extract high-level information about the semantics of the algorithm and discover non-commutative yet semantics-preserving re-orderings. The most recent version of open quantum assembly language (OpenQASM) [16] recognizes the need for higher-level semantics such as control, inverse, and power operations, but is still incapable of representing Pauli-level semantics which are prevalent in quantum simulation kernels. As we have shown, our Pauli IR can carry high-level semantics through multiple optimization stages, encode all known algorithm constraints, and is compatible with further low-level optimizations by these tools.

Quantum compiler optimizations: The state-of-the-art quantum compilers [1, 51] usually have multiple passes to execute different optimizations, (e.g., circuit rewriting [53], gate cancellation [42], template matching [37], qubit mapping [39]). These passes applied on the low-level gate sequences usually only rewrite the circuit locally on very few qubits or gates every time and only focus on one optimization objective in each pass. Different from these optimizations, all passes in Paulihedral performance program transformations at a much larger scope in a scalable way and multiple optimization opportunities can be reconciled because the high-level algorithmic information is leveraged. This makes Paulihedral optimizations more effective than simply combining those small-scale single-objective passes.

Optimizations for simulation algorithms: One common optimization technique is to group the Pauli strings into sets of mutually commutative strings and then apply simultaneous diagonalization [13, 14, 17, 61]. This technique, adopted by t|ket⟩ [13, 14, 17, 50], can simplify the circuit inside each set while the simultaneous diagonalization step introduces substantial overhead before and after the circuit of each set, limiting the overall optimization performance. Some other works [3, 23, 25, 31, 32, 48, 56, 62] explore the simulation program optimization or synthesis but these works are mostly ad-hoc, limited to specific algorithms/architectures, and not easily generalizable to a broader range of programs and employed by a compiler infrastructure. In Paulihedral, the Pauli IR's recursive, block-wise structure can support simulation kernels in all related algorithms, as far as we know. And our optimization algorithms have been shown to be much more effective in the evaluation above.

9 CONCLUSION

We propose Paulihedral, an algorithmic quantum compiler targeting the quantum simulation kernel, a subroutine widely used in many quantum algorithms. Paulihedral enables deep compiler optimizations by defining a new Pauli-string-based IR, which can encode high-level algorithmic information and constraints of many seemingly different quantum algorithms in a unified manner. All follow-up optimizations in Paulihedral operate at a large scope with good scalability and can reconcile multiple optimization opportunities. Paulihedral can be extended to different backends by adding or modifying technology-dependent passes. Comprehensive experimental results show that Paulihedral can significantly outperform state-of-the-art quantum compilers with more effective, scalable optimizations and better reconfigurability.

ACKNOWLEDGMENTS

We thank the anonymous reviews for their constructive feedback. This work was supported in part by NSF 2048144. G. L. was in part funded by NSF QISE-NET fellowship under the award DMR-1747426.

A ARTIFACT ABSTRACT

The artifact contains the source code of the Paulihedral compiler and other necessary code scripts to reproduce the key results (Table 2, 3, and 4) and compare with the baselines in our evaluation. The hardware requirement is a regular X86 server/laptop but the memory size may limit the size of the benchmark that can be compiled. The IBM Melbourne device used in our evaluation has just retired and the related results cannot be reproduced. But we still keep the original script of that experiment for your reference. The software dependencies only contain common software packages. We also provide our benchmark generation script and have pre-generated all benchmarks used in our evaluation. Note that for Table 3 the results are averaged over 20 randomly generated graphs per benchmark. While in our artifact, we show the result of one random seed and a slight deviation is expected.

B ARTIFACT CHECKLIST

- **Language:** Paulihedral has a new intermediate representation (IR), Pauli IR, which is implemented by a 2-dimensional Python list in this artifact. Examples can be found in 'Paulihedral.ipynb'.
- **Algorithm:** Paulihedral has four core algorithms.
  - Gate-count-oriented scheduling (Section 4.1) is the function 'gate_count_oriented_scheduling' in 'parallel_bl.py'.
  - Depth-oriented scheduling (Section 4.2) is the function 'depth_oriented_scheduling' in 'parallel_bl.py'.
  - Block-wise optimization on fault-tolerant backend (Section 5.1) is in function 'block_opt_FT' in 'synthesis_FT.py'.
  - Block-wise optimization on superconducting backend (Section 5.2) is in function 'block_opt_SC' in 'synthesis_SC.py'.
- **Benchmarks:** The benchmarks are the Pauli IR programs of the simulation kernels listed in Table 1.
- **Runtime environment:** Python, Jupyter Notebook.
- **Disk space required:** 10 GB is sufficient for the artifact and all software dependencies.
- **Hardware:** Intel CPU, Memory size depending on the benchmark size (the largest benchmarks can be processed with 1T RAM).
- **Experiments:** Compiling the Pauli IR programs using Paulihedral and follow-up generic quantum compilers.
- **Time to prepare workflow:** 10 minutes
- **Time to complete experiments:** The approximate execution time for each benchmark under different configurations can be found in Table 2. It will take hundreds of CPU hours to fully reproduce all results in Table 2, 3, and 4.
- **Output:** The output of the compilation is the quantum circuit containing CNOT gates and single-qubit gates only.
- **Metrics:** We consider the following metrics in the output
  - Number of single-qubit gates
  - Number of CNOT gates
  - Number of all gates
  - Circuit depth
  - Execution time

  All these metrics can be directly counted from the output quantum circuit.
- **Publicly available:** Yes
- **Code license:** Apache License 2.0
- **Workflow framework used:** Jupyter notebook, Qiskit, t|ket⟩
- **Archived repo:** https://zenodo.org/record/5780204
- **DOI:** 10.5281/zenodo.5748398

C DESCRIPTION

C.1 How to Access

The artifact is available at the following Zenodo link https://zenodo.org/record/5780204 with DOI 10.5281/zenodo.5780204. You can download the zip file and then decompress it.

C.2 Hardware Dependencies

A regular server with Intel CPUs can run our artifact while the amount of RAM may limit the size of benchmarks that can be executed. In our experiments, we use 1T RAM to execute all benchmarks. If you do not have enough RAM, it is possible that the large benchmarks like 'NaCl' and 'Rand-80' are not executable due to out of memory. Note that in Section 6.4, we have real system experiments on IBM's Melbourne device. This device has permanently retired and is not longer accessible. So we are not able to reproduce the results in Figure 11.

C.3 Software Dependencies

The artifact in implemented in Python 3.8.12. We require Qiskit and t|ket⟩. In our experiments, we use Qiskit 0.23.5 and t|ket⟩ version 0.11.0. while other versions may or may not work. These two frameworks requires numpy 1.20.0. We also need jupyter notebook, which can installed from Anaconda, since we prepare the file 'Paulihedral.ipynb' that contains scripts to automatically and interactively reproduce the results in Table 2, 3, and 4 for easy validation. See 'README.md' for installing the software dependencies. Note that the PySCF version must be 1.7.6. The QAOA compiler used in our evaluation (Section 6.2, Table 3) is downloaded from https://github.com/mahabubul-alam/QAOA-Compiler and has already been integrated in this artifact (in the folder 'QAOA-Compiler'). The QAOA compiler requires networkx 2.5.0 and commentjson 0.9.0. The list of dependencies can be found in 'requirements.txt'.

C.4 Benchmarks

The benchmarks can be generated using the file 'gene_benchmark.py'. We have pre-generated all benchmarks used in our evaluation and they can be found in benchmark/data. You can also generate Pauli IR programs from you own Hamiltonians/applications following the format in the example in the first code block in 'Paulihedral.ipynb'.

D INSTALLATION

To use our artifact, you can first download the repo to your local machine. Then you can install the software dependencies by running the command:

```
pip install -r requirements.txt
```

E EVALUATION AND EXPECTED RESULTS

After you download the artifact and install all software dependencies, you can open the jupyter notebook file 'Paulihedral.ipynb'. The first code block will demonstrate an example of a Pauli IR program. Note that we just set all the rotation angles in the center Rz gates to '1.0'. The Rz gates will not be affected in the entire compilation flow. The second, third, and fourth code blocks will automatically reproduce the results in Table 2, 3, and 4, respectively. The results are printed out directly. Note that the results in Table 3 are averaged over 20 randomly generated graphs per benchmark. While in our artifact, we show the result of one random seed. Therefore, a slight deviation is expected. Note that the execution time cannot be perfectly reproduced because your local machine configurations can be different from the server we used in our evaluation while the trend should remain the same.

Since reproducing all the results are very time consuming (hundreds of CPU hours on a server), we add an option in 'config.py' so that small-size experiment results can be reproduced quickly. In 'config.py', if you set 'test_scale' to 'full', then the code will run all benchmarks; if you set 'test_scale' to 'small', then the code only run small benchmarks, which will take about a few CPU hours on a MacBook. By default, 'test_scale' is set to 'small'.

We also attach our code (in file 'real_system.py') for experiment on the IBM devices. This script can print out the compilation results when compiling the QAOA programs onto the IBM Melbourne chip. However, since the IBM Melbourne chip used in this paper is no longer available, the real system execution results in Figure 11 cannot be reproduced. You can change the device to other available IBM devices in the script.

REFERENCES

[1] Héctor Abraham, AduOffei, Rochisha Agarwal, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, Matthew Amy, Eli Arbel, Arijit02, Abraham Asfaw, Artur Avkhadiev, Carlos Azaustre, AzizNgoueya, Abhik Banerjee, Aman Bansal, Panagiotis Barkoutsos, Ashish Barnawal, George Barron, George S. Barron, Luciano Bello, Yael Ben-Haim, Daniel Bevenius, Arjun Bhobe, Lev S. Bishop, Carsten Blank, Sorin Bolos, Samuel Bosch, Brandon, Sergey Bravyi, Bryce-Fuller, David Bucher, Artemiy Burov, Fran Cabrera, Padraic Calpin, Lauren Capelluto, Jorge Carballo, Ginés Carrascal, Adrian Chen, Chun-Fu Chen, Edward Chen, Jielun (Chris) Chen, Richard Chen, Jerry M. Chow, Spencer Churchill, Christian Claus, Christian Clauss, Romilly Cocking, Filipe Correa, Abigail J. Cross, Andrew W. Cross, Simon Cross, Juan Cruz-Benito, Chris Culver, Antonio D. Córcoles-Gonzales, Sean Dague, Tareq El Dandachi, Marcus Daniels, Matthieu Dartiailh, DavideFrr, Abdón Rodríguez Davila, Anton Dekusar, Delton Ding, Jun Doi, Eric Drechsler, Drew, Eugene Dumitrescu, Karel Dumon, Ivan Duran, Kareem EL-Safty, Eric Eastman, Grant Eberle, Pieter Eendebak, Daniel Egger, Mark Everitt, Paco Martín Fernández, Axel Hernández Ferrera, Romain Fouilland, FranckChevallier, Albert Frisch, Andreas Fuhrer, Bryce Fuller, MELVIN GEORGE, Julien Gacon, Borja Godoy Gago, Claudio Gambella, Jay M. Gambetta, Adhisha Gammanpila, Luis Garcia, Tanya Garg, Shelly Garion, Austin Gilliam, Aditya Giridharan, Juan Gomez-Mosquera, Gonzalo, Salvador de la Puente González, Jesse Gorzinski, Ian Gould, Donny Greenberg, Dmitry Grinko, Wen Guan, John A. Gunnels, Mikael Haglund, Isabel Haide, Ikko Hamamura, Omar Costa Hamido, Frank Harkins, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Stefan Hillmich, Hiroshi Horii, Connor Howington, Shaohan Hu, Wei Hu, Junye Huang, Rolf Huisman, Haruki Imai, Takashi Imamichi, Kazuaki Ishizaki, Raban Iten, Toshinari Itoko, JamesSeaward, Ali Javadi, Ali Javadi-Abhari, Wahaj Javed, Jessica, Madhav Jivrajani, Kiran Johns, Scott Johnstun, Jonathan-Shoemaker, Vismai K, Tal Kachmann, Akshay Kale, Naoki Kanazawa, Kang-Bae, Anton Karazeev, Paul Kassebaum, Josh Kelso, Spencer King, Knabberjoe, Yuri Kobayashi, Arseny Kovyrshin, Rajiv Krishnakumar, Vivek Krishnan, Kevin Krsulich, Prasad Kumkar, Gawel Kus, Ryan LaRose, Enrique Lacal, Raphaël Lambert, John Lapeyre, Joe Latone, Scott Lawrence, Christina Lee, Gushu Li, Dennis Liu, Peng Liu, Yunho Maeng, Kahan Majmudar, Aleksei Malyshev, Joshua Manela, Jakub Marecek, Manoel Marques, Dmitri Maslov, Dolph Mathews, Atsushi Matsuo, Douglas T. McClure, Cameron McGarry, David McKay, Dan McPherson, Srujan Meesala, Thomas Metcalfe, Martin Mevissen, Andrew Meyer, Antonio Mezzacapo, Rohit Midha, Zlatko Minev, Abby Mitchell, Nikolaj Moll, Jhon Montanez, Gabriel Monteiro, Michael Duane Mooring, Renier Morales, Niall Moran, Mario Motta, MrF, Prakash Murali, Jan Müggenburg, David Nadlinger, Ken Nakanishi, Giacomo Nannicini, Paul Nation, Edwin Navarro, Yehuda Naveh, Scott Wyman Neagle, Patrick Neuweiler, Johan Nicander, Pradeep Niroula, Hassi Norlen, NuoWenLei, Lee James O'Riordan, Oluwatobi Ogunbayo, Pauline Ollitrault, Raul Otaolea, Steven Oud, Dan Padilha, Hanhee Paik, Soham Pal, Yuchen Pang, Vincent R. Pascuzzi, Simone Perriello, Anna Phan, Francesco Piro, Marco Pistoia, Christophe Piveteau, Pierre Pocreau, Alejandro Pozas-Kerstjens, Milos Prokop, Viktor Prutyanov, Daniel Puzzuoli, Jesús Pérez, Quintiii, Rafey Iqbal Rahman, Arun Raja, Nipun Ramagiri, Anirudh Rao, Rudy Raymond, Rafael Martín-Cuevas Redondo, Max Reuter, Julia Rice, Matt Riedemann, Marcello La Rocca, Diego M. Rodríguez, RohithKarur, Max Rossmannek, Mingi Ryu, Tharrmashastha SAPV, SamFerracin, Martin Sandberg, Hirmay Sandesara, Ritvik Sapra, Hayk Sargsyan, Aniruddha Sarkar, Ninad Sathaye, Bruno Schmitt, Chris Schnabel, Zachary Schoenfeld, Travis L. Scholten, Eddie Schoute, Joachim Schwarm, Ismael Faro Sertage, Kanav Setia, Nathan Shammah, Yunong Shi, Adenilton Silva, Andrea Simonetto, Nick Singstock, Yukio Siraichi, Iskandar Sitdikov, Seyon Sivarajah, Magnus Berg Sletfjerding, John A. Smolin, Mathias Soeken, Igor Olegovich Sokolov, Igor Sokolov, SooluThomas, Starfish, Dominik Steenken, Matt Stypulkoski, Shaojun Sun, Kevin J. Sung, Hitomi Takahashi, Tanvesh Takawale, Ivano Tavernelli, Charles Taylor, Pete Taylour, Soolu Thomas, Mathieu Tillet, Maddy Tod, Miroslav Tomasik, Enrique de la Torre, Kenso Trabing, Matthew Treinish, TrishaPe, Davindra Tulsi, Wes Turner, Yotam Vaknin, Carmen Recio Valcarce, Francois Varchon, Almudena Carrera Vazquez, Victor Villar, Desiree Vogt-Lee, Christophe Vuillot, James Weaver, Johannes Weidenfeller, Rafal Wieczorek, Jonathan A. Wildstrom, Erick Winston, Jack J. Woehr, Stefan Woerner, Ryan Woo, Christopher J. Wood, Ryan Wood, Stephen Wood, Steve Wood, James Wootton, Daniyar Yeralin, David Yonge-Mallo, Richard Young, Jessie Yu, Christopher Zachow, Laura Zdanski, Helena Zhang, Christa Zoufal, and Mantas Čepulkovskis. 2019. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2562110

[2] Daniel S. Abrams and Seth Lloyd. 1999. Quantum Algorithm Providing Exponential Speed Increase for Finding Eigenvalues and Eigenvectors. *Phys. Rev. Lett.* 83 (Dec 1999), 5162–5165. Issue 24. https://doi.org/10.1103/PhysRevLett.83.5162

[3] Mahabubul Alam, Abdullah Ash-Saki, and Swaroop Ghosh. 2020. Circuit Compilation Methodologies for Quantum Approximate Optimization Algorithm. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 215–228. https://doi.org/10.1109/MICRO50266.2020.00029

[4] Mahabubul Alam, Abdullah Ash-Saki, and Swaroop Ghosh. 2020. An efficient circuit compilation flow for quantum approximate optimization algorithm. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6. https://doi.org/10.1109/DAC18072.2020.9218558

[5] Mahabubul Alam, Abdullah Ash-Saki, Junde Li, Anupam Chattopadhyay, and Swaroop Ghosh. 2020. Noise resilient compilation policies for quantum approximate optimization algorithm. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–7. https://doi.org/10.1145/3400302.3415745

[6] Matthew Amy and Vlad Gheorghiu. 2020. staq—A full-stack quantum processing toolkit. *Quantum Science and Technology* 5, 3 (jun 2020), 034016. https://doi.org/10.1088/2058-9565/ab9359

[7] J. M. Arrazola, V. Bergholm, K. Brádler, T. R. Bromley, M. J. Collins, I. Dhand, A. Fumagalli, T. Gerrits, A. Goussev, L. G. Helt, J. Hundal, T. Isacsson, R. B. Israel, J. Izaac, S. Jahangiri, R. Janik, N. Killoran, S. P. Kumar, J. Lavoie, A. E. Lita, D. H. Mahler, M. Menotti, B. Morrison, S. W. Nam, L. Neuhaus, H. Y. Qi, N. Quesada, A. Repingon, K. K. Sabapathy, M. Schuld, D. Su, J. Swinarton, A. Száva, K. Tan, P. Tan, V. D. Vaidya, Z. Vernon, Z. Zabaneh, and Y. Zhang. 2021. Quantum circuits with many photons on a programmable nanophotonic chip. *Nature* 591, 7848 (01 Mar 2021), 54–60. https://doi.org/10.1038/s41586-021-03202-1

[8] G. Brassard and P. Hoyer. 1997. An exact quantum polynomial-time algorithm for Simon's problem. In *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems*. 12–23. https://doi.org/10.1109/ISTCS.1997.595153

[9] Sergey B. Bravyi and Alexei Yu. Kitaev. 2002. Fermionic Quantum Computation. *Annals of Physics* 298, 1 (2002), 210–226. https://doi.org/10.1006/aphy.2002.6254

[10] M. Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C. Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R. McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, and Patrick J. Coles. 2021. Variational quantum algorithms. *Nature Reviews Physics* 3, 9 (01 Sep 2021), 625–644. https://doi.org/10.1038/s42254-021-00348-9

[11] Christopher Chamberland, Guanyu Zhu, Theodore J. Yoder, Jared B. Hertzberg, and Andrew W. Cross. 2020. Topological and Subsystem Codes on Low-Degree Graphs with Flag Qubits. *Phys. Rev. X* 10 (Jan 2020), 011022. Issue 1. https://doi.org/10.1103/PhysRevX.10.011022

[12] J. Cheng, H. Deng, and X. Qia. 2020. AccQOC: Accelerating Quantum Optimal Control Based Pulse Generation. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 543–555. https://doi.org/10.1109/ISCA45697.2020.00052

[13] Alexander Cowtan, Silas Dilkes, Ross Duncan, Will Simmons, and Seyon Sivarajah. 2020. Phase Gadget Synthesis for Shallow Circuits. *Electronic Proceedings in Theoretical Computer Science* 318 (May 2020), 213–228. https://doi.org/10.4204/eptcs.318.13

[14] Alexander Cowtan, Will Simmons, and Ross Duncan. 2020. A Generic Compilation Strategy for the Unitary Coupled Cluster Ansatz. *arXiv preprint arXiv:2007.10515* (2020).

[15] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. 2017. Open quantum assembly language. *arXiv preprint arXiv:1707.03429* (2017).

[16] Andrew W Cross, Ali Javadi-Abhari, Thomas Alexander, Niel de Beaudrap, Lev S Bishop, Steven Heidel, Colm A Ryan, John Smolin, Jay M Gambetta, and Blake R Johnson. 2021. OpenQASM 3: A broader and deeper quantum assembly language. *arXiv preprint arXiv:2104.14722* (2021).

[17] Arianne Meijer-van de Griend and Ross Duncan. 2020. Architecture-aware synthesis of phase polynomials for NISQ devices. *arXiv preprint arXiv:2004.06052* (2020).

[18] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028* (2014).

[19] Richard P Feynman. 1982. Simulating physics with computers. *Int. J. Theor. Phys* 21, 6/7 (1982).

[20] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. 2012. Surface codes: Towards practical large-scale quantum computation. *Phys. Rev. A* 86 (Sep 2012), 032324. Issue 3. https://doi.org/10.1103/PhysRevA.86.032324

[21] Bryan T. Gard, Linghua Zhu, George S. Barron, Nicholas J. Mayhall, Sophia E. Economou, and Edwin Barnes. 2020. Efficient symmetry-preserving state preparation circuits for the variational quantum eigensolver algorithm. *npj Quantum Information* 6, 1 (28 Jan 2020), 10. https://doi.org/10.1038/s41534-019-0240-1

[22] P. Gokhale, A. Javadi-Abhari, N. Earnest, Y. Shi, and F. T. Chong. 2020. Optimized Quantum Compilation for Near-Term Algorithms with OpenPulse. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 186–200. https://doi.org/10.1109/MICRO50266.2020.00027

[23] Kaiwen Gui, Teague Tomesh, Pranav Gokhale, Yunong Shi, Frederic T Chong, Margaret Martonosi, and Martin Suchara. 2020. Term grouping and travelling salesperson for digital quantum simulation. *arXiv preprint arXiv:2001.05983* (2020).

[24] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum Algorithm for Linear Systems of Equations. *Phys. Rev. Lett.* 103 (Oct 2009), 150502. Issue 15. https://doi.org/10.1103/PhysRevLett.103.150502

[25] Matthew B. Hastings, Dave Wecker, Bela Bauer, and Matthias Troyer. 2015. Improving Quantum Algorithms for Quantum Chemistry. *Quantum Info. Comput.* 15, 1–2 (jan 2015), 1–21. https://doi.org/10.5555/2685188.2685189

[26] P. Jordan and E. Wigner. 1928. Über das Paulische Äquivalenzverbot. *Zeitschrift für Physik* 47, 9 (01 Sep 1928), 631–651. https://doi.org/10.1007/BF01331938

[27] N. Khammassi, I. Ashraf, J. V. Someren, R. Nane, A. M. Krol, M. A. Rol, L. Lao, K. Bertels, and C. G. Almudever. 2021. OpenQL: A Portable Quantum Programming Framework for Quantum Accelerators. *J. Emerg. Technol. Comput. Syst.* 18, 1, Article 13 (dec 2021), 24 pages. https://doi.org/10.1145/3474222

[28] Aleks Kissinger and John van de Wetering. 2020. PyZX: Large Scale Automated Diagrammatic Reasoning. *Electronic Proceedings in Theoretical Computer Science* 318 (May 2020), 229–241. https://doi.org/10.4204/eptcs.318.14

[29] Lingling Lao and Dan Browne. 2021. 2QAN: A quantum compiler for 2-local qubit Hamiltonian simulation algorithms. *arXiv preprint arXiv:2108.02099* (2021).

[30] Gushu Li, Yufei Ding, and Yuan Xie. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1001–1014. https://doi.org/10.1145/3297858.3304023

[31] Gushu Li, Yunong Shi, and Ali Javadi-Abhari. 2021. Software-Hardware Co-Optimization for Computational Chemistry on Superconducting Quantum Processors. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*. IEEE Press, 832–845. https://doi.org/10.1109/ISCA52012.2021.00070

[32] Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, and Yuan Xie. 2021. On the Co-Design of Quantum Software and Hardware. In *Proceedings of the Eight Annual ACM International Conference on Nanoscale Computing and Communication* (Virtual Event, Italy) *(NANOCOM '21)*. Association for Computing Machinery, New York, NY, USA, Article 15, 7 pages. https://doi.org/10.1145/3477206.3477464

[33] Seth Lloyd. 1996. Universal Quantum Simulators. *Science* 273, 5278 (1996), 1073–1078. https://doi.org/10.1126/science.273.5278.1073 arXiv:https://www.science.org/doi/pdf/10.1126/science.273.5278.1073

[34] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. 2013. Quantum algorithms for supervised and unsupervised machine learning. *arXiv preprint arXiv:1307.0411* (2013).

[35] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. 2014. Quantum principal component analysis. *Nature Physics* 10, 9 (01 Sep 2014), 631–633. https://doi.org/10.1038/nphys3029

[36] Dmitri Maslov. 2016. Optimal and Asymptotically Optimal NCT Reversible Circuits by the Gate Types. *Quantum Info. Comput.* 16, 13–14 (oct 2016), 1096–1112. https://doi.org/10.5555/3179430.3179432

[37] Dmitri Maslov, Gerhard W. Dueck, D. Michael Miller, and Camille Negrevergne. 2008. Quantum Circuit Simplification and Level Compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 3 (2008), 436–444. https://doi.org/10.1109/TCAD.2007.911334

[38] A. McCaskey and T. Nguyen. 2021. A MLIR Dialect for Quantum Assembly Languages. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE Computer Society, Los Alamitos, CA, USA, 255–264. https://doi.org/10.1109/QCE52317.2021.00043

[39] Prakash Murali, Jonathan M. Baker, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. 2019. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1015–1029. https://doi.org/10.1145/3297858.3304075

[40] Prakash Murali, Dripto M. Debroy, Kenneth R. Brown, and Margaret Martonosi. 2020. Architecting Noisy Intermediate-Scale Trapped Ion Quantum Computers. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 529–542. https://doi.org/10.1109/ISCA45697.2020.00051

[41] Prakash Murali, David C. Mckay, Margaret Martonosi, and Ali Javadi-Abhari. 2020. Software Mitigation of Crosstalk on Noisy Intermediate-Scale Quantum Computers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 1001–1016. https://doi.org/10.1145/3373376.3378477

[42] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information* 4, 1 (10 May 2018), 23. https://doi.org/10.1038/s41534-018-0072-4

[43] Michael A Nielsen and Isaac L Chuang. 2010. Quantum Computation and Quantum Information. *Quantum Computation and Quantum Information, by Michael A. Nielsen, Isaac L. Chuang, Cambridge, UK: Cambridge University Press, 2010* (2010).

[44] Shin Nishio, Yulu Pan, Takahiko Satoh, Hideharu Amano, and Rodney Van Meter. 2020. Extracting Success from IBM's 20-Qubit Machines Using Error-Aware Compilation. *J. Emerg. Technol. Comput. Syst.* 16, 3, Article 32 (May 2020), 25 pages. https://doi.org/10.1145/3386162

[45] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications* 5, 1 (23 Jul 2014), 4213. https://doi.org/10.1038/ncomms5213

[46] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. 2014. Quantum Support Vector Machine for Big Data Classification. *Phys. Rev. Lett.* 113 (Sep 2014), 130503. Issue 13. https://doi.org/10.1103/PhysRevLett.113.130503

[47] Zain H Saleem, Bilal Tariq, and Martin Suchara. 2020. Approaches to constrained quantum approximate optimization. *arXiv preprint arXiv:2010.06660* (2020).

[48] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I. Schuster, Henry Hoffmann, and Frederic T. Chong. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1031–1044. https://doi.org/10.1145/3297858.3304018

[49] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintao Pereira. 2018. Qubit Allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 113–125. https://doi.org/10.1145/3168822

[50] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. $t|ket\rangle$: a retargetable compiler for NISQ devices. *Quantum Science and Technology* 6, 1 (nov 2020), 014003. https://doi.org/10.1088/2058-9565/ab8e92

[51] Robert S Smith, Michael J Curtis, and William J Zeng. 2016. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355* (2016).

[52] R S Smith, E C Peterson, M G Skilbeck, and E J Davis. 2020. An open-source, industrial-strength optimizing compiler for quantum programs. *Quantum Science and Technology* 5, 4 (jul 2020), 044001. https://doi.org/10.1088/2058-9565/ab9acb

[53] Mathias Soeken and Michael Kirkedal Thomsen. 2013. White Dots Do Matter: Rewriting Reversible Logic Circuits. In *Proceedings of the 5th International Conference on Reversible Computation* (Victoria, BC, Canada) *(RC'13)*. Springer-Verlag, Berlin, Heidelberg, 196–208. https://doi.org/10.1007/978-3-642-38986-3_16

[54] Qiming Sun, Timothy C. Berkelbach, Nick S. Blunt, George H. Booth, Sheng Guo, Zhendong Li, Junzi Liu, James D. McClain, Elvira R. Sayfutyarova, Sandeep Sharma, Sebastian Wouters, and Garnet Kin-Lic Chan. 2017. PySCF: the Python-based simulations of chemistry framework. , e1340 pages. https://doi.org/10.1002/wcms.1340 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/wcms.1340

[55] Masuo Suzuki. 1976. Generalized Trotter's formula and systematic approximants of exponential operators and inner derivations with applications to many-body problems. *Communications in Mathematical Physics* 51, 2 (1976), 183–190. https://doi.org/10.1007/BF01609348

[56] Bochen Tan and Jason Cong. 2020. Optimal Layout Synthesis for Quantum Computing. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) *(ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, Article 137, 9 pages. https://doi.org/10.1145/3400302.3415620

[57] Swamit S. Tannu and Moinuddin Qureshi. 2019. Ensemble of Diverse Mappings: Improving Reliability of Quantum Computers by Orchestrating Dissimilar Mistakes. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 253–265. https://doi.org/10.1145/3352460.3358257

[58] Swamit S. Tannu and Moinuddin K. Qureshi. 2019. Not All Qubits Are Created Equal: A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 987–999. https://doi.org/10.1145/3297858.3304007

[59] Andrew Tranter, Peter J. Love, Florian Mintert, and Peter V. Coveney. 2018. A Comparison of the Bravyi–Kitaev and Jordan–Wigner Transformations for the Quantum Simulation of Quantum Chemistry. *Journal of Chemical Theory and Computation* 14, 11 (2018), 5617–5630. https://doi.org/10.1021/acs.jctc.8b00450 arXiv:https://doi.org/10.1021/acs.jctc.8b00450 PMID: 30189144.

[60] H. F. Trotter. 1959. On the Product of Semi-Groups of Operators. *Proc. Amer. Math. Soc.* 10, 4 (1959), 545–551. https://doi.org/10.2307/2033649

[61] Ewout van den Berg and Kristan Temme. 2020. Circuit optimization of Hamiltonian simulation by simultaneous diagonalization of Pauli clusters. *Quantum* 4 (Sept. 2020), 322. https://doi.org/10.22331/q-2020-09-12-322

[62] Vivien Vandaele, Simon Martiel, and Timothée Goubault de Brugière. 2021. Phase polynomials synthesis algorithms for NISQ architectures and beyond. *arXiv preprint arXiv:2104.00934* (2021).

[63] Xin-Chuan Wu, Dripto M. Debroy, Yongshan Ding, Jonathan M. Baker, Yuri Alexeev, Kenneth R. Brown, and Frederic T. Chong. 2021. TILT: Achieving Higher Fidelity on a Trapped-Ion Linear-Tape Quantum Computing Architecture. (2021), 153–166. https://doi.org/10.1109/HPCA51647.2021.00023

[64] A. Zulehner, A. Paler, and R. Wille. 2019. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 7 (2019), 1226–1236. https://doi.org/10.1109/TCAD.2018.2846658