QASMTrans: A QASM based Quantum Transpiler Framework for NISQ Devices

Fei Hua
Pacific Northwest National Laboratory
Richland,USA
Rutgers University
New Jersey, USA

Meng Wang
The University of British Columbia
Vancouver, Canada
Pacific Northwest National Laboratory
Richland, USA

Gushu Li
University of Pennsylvania
Pennsylvania, USA

Bo Peng
Pacific Northwest National Laboratory
Richland, USA

Chenxu Liu
Pacific Northwest National Laboratory
Richland, USA

Muqing Zheng
Pacific Northwest National Laboratory
Richland, USA

Samuel Stein
Pacific Northwest National Laboratory
Richland, USA

Yufei Ding
University of California San Diego
California, USA

Eddy Z. Zhang
Rutgers University
New Jersey, USA

Travis S. Humble
Oak Ridge National Laboratory
Tennessee, USA

Ang Li
Pacific Northwest National Laboratory
Richland, USA

ABSTRACT

The success of a quantum algorithm hinges on the ability to orchestrate a successful application induction. Detrimental overheads in mapping general quantum circuits to physically implementable routines can be the deciding factor between a successful and erroneous circuit induction. In QASMTrans, we focus on the problem of rapid circuit transpilation. Transpilation plays a crucial role in converting high-level, machine-agnostic circuits into machine-specific circuits constrained by physical topology and supported gate sets. The efficiency of transpilation continues to be a substantial bottleneck, especially when dealing with larger circuits requiring high degrees of inter-qubit interaction. QASMTrans is a high-performance C++ quantum transpiler framework that demonstrates 3-1111× speedups compared to the commonly used Qiskit transpiler. We observe speedups on large dense circuits such as 'uccsd_n24' which require $O(10^6)$ gates. QASMTrans successfully transpiles the aforementioned circuits in 7.9s, whilst Qiskit needs 502 seconds with optimization 1 and exceeds an hour of transpilation time with optimization 3. With QASMTrans providing transpiled circuits in a fraction of the time of prior transpilers, potential design space exploration, and heuristic-based transpiler design becomes substantially more tractable. QASMTrans is released at http://github.com/pnnl/qasmtrans.

CCS CONCEPTS

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

KEYWORDS

QASMTrans, Compiler, IO, Optimizaton

ACM Reference Format:
Fei Hua, Meng Wang, Gushu Li, Bo Peng, Chenxu Liu, Muqing Zheng, Samuel Stein, Yufei Ding, Eddy Z. Zhang, Travis S. Humble, and Ang Li. 2023. QASMTrans: A QASM based Quantum Transpiler Framework for NISQ Devices. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624062.3624222

1 INTRODUCTION

The past decade has witnessed tremendous development in *Noisy Intermediate-Scale Quantum* (NISQ) computers [9, 35], where a few hundred physical qubits are available with relatively limited coherence times and high error rates. These NISQ machines, while offering great potential, are constrained by various factors such as non-trivial noise [26][41], limited connectivity [6] and machine-specific basis gate sets [25]. Due to the limited qubit number and short coherence time, effectively mapping application circuits to the constrained NISQ machine poses a considerable challenge and can

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SC-W 2023, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624222

significantly impact the fidelity of the execution results. Transpilation is the specific terminology referring to the compilation process of transforming a high-level quantum circuit into an equivalent circuit that is compatible with the specifications of a quantum device, including: the basis gate set, topology of the quantum chip, timing constraints, fidelity of operations, etc. The goal of a transpiler is to perform this transformation while minimizing the impact on the functionality of the circuit and optimizing its performance delivery.

Several attempts on quantum transpilation have already been made by the community (see a summary in Section 5), but there are still technical gaps. On the one hand, commercial transpilers such as those embedded in Qiskit [34] and Cirq [1] provide comprehensive functionalities, but are typically slow, especially for deep circuits arising in practical quantum applications such as chemistry [4, 17], optimization [11, 44] and nuclear physics [14, 40]. Additionally, the slow transpilation speed limits their capability to explore larger design space and integrate more advanced but expensive optimizations. This is especially the case when dynamic circuit generation and transpilation is needed, such as in variational quantum algorithms (VQAs) [5, 38] and when optimized to mitigate state-dependent bias at runtime [42].

On the other hand, most of the research studies in academia have focused on specific transpilation techniques, such as gate decomposition, circuit optimization, mapping and routing, etc. [23][49][48]. These approaches lack end-to-end demonstrations and are often implemented and validated by embedding into or replacing part of Python-based commercial frameworks such as Qiskit and Cirq. Consequently, they are also constrained by the limitations of the underlying frameworks, such as slow speed, difficulties in launching large circuits, binding to certain device features, lack of flexibility, and frequent interface upgrades, etc. In this paper, we present QASMTrans,

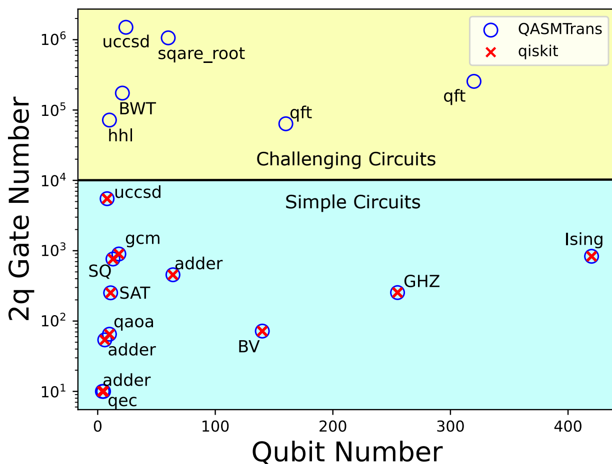

Figure 1: QASMTrans is designed to transpile challenging deep circuits from QASMBench [21], while broadly used tools such as Qiskit cannot finish within around a minute.

an end-to-end, self-contained, light-weight quantum transpiler entirely realized in C++ for effectively parsing and compiling large QASM circuits. QASMTrans comprises four major components:

(1) An ***IO*** module that uses a QASM Paser for parsing an input OpenQASM file, and translating it into a structure acting as the internal intermediate representation (IR). The output will export the transpiled QASM circuits for a particular NISQ device, such as those provided by IBMQ, Rigetti, IonQ, Quantinuum, etc.
(2) A ***Configuration*** module for preparing the coupling graph of the device, generating the DAG for the circuit, and decomposing the 3-qubit gates into 1-qubit and 2-qubit gates.
(3) An ***Optimization*** module for the various optimization passes. This includes decomposition into basis gates, routing, and mapping. These passes are made with respect to the topology, basis gate set, fidelity, and features of the circuit. The goal of the backend optimization is to allow the circuits to run more efficiently on the targeted NISQ devices or simulators.
(4) A main ***Transpiler*** component to do the routing and mapping and also decompose into basis gates based on specific NISQ devices.

QASMTrans is primarily designed as an open-source transpiler infrastructure serving as a baseline for implementing and validating advanced transpilation technologies while supporting novel devices and computation models. We evaluate QASMTrans using diverging circuits with some of them being quite challenging (from 4 to 420 qubits, and from 10 to 2.2M gates, see Figure 1) from QASMBench [21]. Remarkably, most of the benchmarks can be completed within a few seconds. Even the largest and most demanding benchmark that Qiskit cannot finish within several minutes, can be transpiled by QASMTrans in only 3-7 seconds. This work thus makes the following main contributions:

- We propose an end-to-end, self-contained, light-weighted opensource quantum compiler in C++ that can significantly reduce the transpilation time for a wide range of applications, improving the efficiency of quantum computations on NISQ devices.
- QASMTrans is equipped with optimization techniques for generating specific basis gates towards different target machines or classical simulators.
- Through comprehensive experiments and analysis over multiple quantum platforms, we show that QASMTrans can transpile circuits with comparable fidelity on real NISQ devices from Rigetti, IBMQ, IonQ, and Quantinuum, but at a much faster speed compared to existing transpilers such as Qiskit and MQT-Qmap [45].

The remainder of this paper is structured as follows: Section 2 provides background information. Section 3 presents the QASMTrans transpiler. Section 4 shows the evaluation results. Section 5 summarizes related work about quantum transpilation. Section 6 concludes.

2 BACKGROUND

2.1 Noisy Intermediate-Scale Quantum (NISQ)

NISQ systems refer to near-term quantum platforms featuring fifty to less than a thousand physical qubits [33]. These qubits are fabricated based on various technologies, such as superconducting [9, 35], trapped-ion [8, 19], photonic [2, 30], spin qubits [27, 32],

Table 1: OpenQASM gate definition (5 basic gates + 11 standard gates + 18 composition gates).

Gates	Meaning	Gates	Meaning	Gates	Meaning
U3	3 parameter 2 pulse 1-qubit	TDG	conjugate of sqrt(S)	CRZ	Controlled RZ rotation
U2	2 parameter 1 pulse 1-qubit	RX	X-axis rotation	CU1	Controlled phase rotation
U1	1 parameter 0 pulse 1-qubit	RY	Y-axis rotation	CU3	Controlled U3
CX	Controlled-NOT	RZ	Z-axis rotation	RXX	2-qubit XX rotation
ID	Idle gate or identity	CZ	Controlled phase	RZZ	2-qubit ZZ rotation
X	Pauli-X bit flip	CY	Controlled Y	RCCX	Relative-phase CXX
Y	Pauli-Y bit and phase flip	SWAP	Swap	RC3X	Relative-phase 3-controlled X
Z	Pauli-Z phase flip	CH	Controlled H	C3X	3-controlled X
H	Hadamard	CCX	Toffoli	C3XSQRTX	3-controlled sqrt(X)
S	sqrt(Z) phase	CSWAP	Fredkin	C4X	4-controlled X
SDG	conjugate of sqrt(Z)	CRX	Controlled RX rotation		
T	sqrt(S) phase	CRY	Controlled RY rotation		

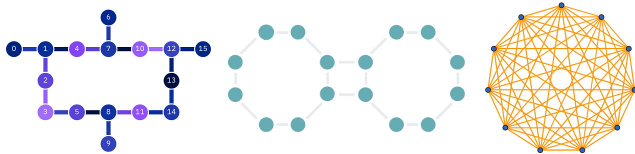

Figure 2: NISQ device topology: IBMQ-Guadalupe (left), Rigetti-Aspen (middle) and IonQ-QPU (right).

neutral atoms [3, 13], etc. To accomplish the execution of a circuit, the physical qubits need to stay coherent for a sufficiently long time. However, before all the circuits can be executed on the real quantum machine, it must (1) fit the basis gates of the quantum machine and (2) meet the coupling constraints of the machine topology.

2.1.1 Basis Gates. Each NISQ device has its own basis gate set, known as the *quantum instruction set architecture* (QISA). It defines the basic operations that are physically supported by the underlying platform. During quantum transpilation, all the logic gates will be decomposed and transpiled into gate sequences purely formed by basis gates. Table 2 shows the basis gate set for IBMQ, Rigetti, IonQ, and Quantinuum devices. Typically, quantum device vendors only provide profiling or calibration data for the basis gates (per qubit or system-wide average), including T1, T2, duration, fidelity, etc. These basis gates also represent the operations to be implemented by a classical simulator.

2.1.2 Topology. Physical qubits in a quantum processor are interconnected. In a quantum device, the 1-qubit gates are directly performed on individual qubits. The 2-qubit gates, however, have to be performed on a qubit-pair that is interconnected. This is especially the case for superconducting devices (e.g., IBMQ and Rigetti), where the connectivity of qubits follows a certain topology, as shown in Figure 2. The topology thus limits the sites where two-qubit gates can be performed: if a two-qubit gate is desired for remote qubits, a series of `SWAP` gates are required to physically move the two qubits to a connected tuple following the path defined by the topology, known as *routing*. `SWAP` gates are costly, usually achieved through three `CNOT` or `CX` gates.

These extra `SWAP`s are one of the major factors contributing to deep circuits and considerable noise for superconducting devices, as compared to contemporary small-scale trapped-ion devices practicing all-to-all connectivity (see Figure 2). Our previous study [39] shows that, for a 17-gate variational circuit, from the 5-qubit IBMQ Cairo to the 5-qubit IonQ QPU, a fidelity increase from 72% to 80%

Table 2: Basis gates for IBM-Q, Rigetti, IonQ and Quantinuum NISQ devices.

NISQ	Technology	1-qubit basis	2-qubit basis
IBMQ	Superconducting	ID, RZ, SX, X	CX/ECR
Rigetti	Superconducting	RX, RZ	CZ (XY)
IonQ	Trapped-Ion	GPI, GPI2, GZ	MS
Quantinuum	Trapped-Ion	RX, RZ	ZZ

(ideally 97.8%) has been observed. This is mainly due to the 7 extra `SWAP` gates when transpiling to comply with the topology of IBMQ Cairo.

2.2 QASM

OpenQASM (Open Quantum Assembly Language, we particularly refer to OpenQASM 2.0 in this work) [10], also known colloquially as QASM, is an intermediate representation (IR) of quantum instructions. QASM acts as a unified low-level assembly language for IBMQ and other quantum machines. Many of these NISQ devices, accessible through the IBMQ network [16], have been widely explored by existing works. Table 1 lists the types of gates that are defined in the QASM specification (i.e., the `"qelib1.inc"` header file) [10]. Within these gates, the first five, i.e., `U3`, `U2`, `U1`, `CX`, and `ID`, are *basic gates* that are expected to be supported by the quantum backend. From `X` to `RZ` are *standard gates* defined atomically in OpenQASM. The remaining gates from `CZ` to `C4X` are *composition gates* that are constructed by standard gates. These gates are frequently used gates defined in `qelib1.inc`. OpenQASM 2.0 is a low-level IR, which is executed sequentially without any loops, branches, or jumps, making it very convenient for static analysis and simulating in a classical simulator [20, 22]. A QASM code can be directly launched in IBMQ or through Qiskit. With all these benefits, QASMTrans uses QASM as the primary format for input and output.

3 QASMTRANS TRANSPILER

We elaborate on the QASMTrans transpiler framework in this section, The main structure is shown in Figure 3. QASMTrans contains the following main components:

(1) **Input/Output (IO)**:
- *Input*: QASMTrans starts with a QASM parser. The parser reads the QASM file, and translates it into a gate IR. Meanwhile, the input module also extracts pertinent hardware details from a JSON file that describes the backend device.

We plan to support other input formats such as QIR [29] and Quil [37].
- *Output*: Once the transpilation is complete, the circuit is saved to a new QASM file, primed for execution on real quantum hardware. QIR [29] is another format to be supported.

(2) **QASMTrans Configuration**:
- *Gate Decomposition*: In this phase, gates with three qubits are methodically broken down into combinations of one- and two-qubit gates. For example, the `CCX` gate will be decomposed into `CX` and `T` gates.
- *Directed Acyclic Graph* (DAG): A DAG will be generated for the gates describing the dependency. In the DAG, every vertex represents a physical qubit, whereas each edge represents a coupling link.
- *Coupling Graph*: We generate the coupling graph based on the input hardware JSON file, where each vertex represents a physical qubit, and each edge represents the link between qubits. The coupling graph is essential for routing/mapping.

(3) **QASMTrans Process**:
- *Routing and Mapping*: This involves aligning the given quantum circuit to the specific topology of different quantum machines. To achieve this, we introduce `SWAP` gates where necessary. As a starting point, we implement the Sabre algorithm [23] that is also widely used in frameworks such as Qiskit and XACC [28].
- *Basis Gate Decomposition*: Depending on the desired quantum machines, like Rigetti or Quantinuum, the circuit is further decomposed into the directly executable basis gates of the specific hardware.

(4) **Simulation-Oriented Optimization**:
- *Simulation-Aware Constrained Routing*: To date, many quantum circuits and algorithms are still evaluated in classical simulators. Given the exponential cost of having more qubits to simulate, in QASMTrans, we introduce a method that can limit the number and index of qubits used for the transpilation. This can significantly reduce the transpilation time as well as simulation time.
- *Qubit Priority Rescheduling*: Based on user-specified qubit priorities, QASMTrans can optimize and realign the qubit mapping. This is especially useful for distributive classical simulation, as the number of gates over globally shared qubits can be minimized.

3.1 QASM Parser

The QASM parser is responsible for parsing the input OpenQASM to the internal gate IR, which will be discussed in more detail below.

3.1.1 Tokenization using Lexertk. The parser begins its operation by tokenizing the QASM text, a process that involves breaking down the text into smaller chunks known as tokens. This is achieved by incorporating Lexertk [31], a high-performance lexer tool written in C++ and distributed through a single C++ header file. The parser of QASMTrans uses Lexertk to scan through the QASM code and break it down into various tokens. Each token is a string of characters that conforms to the *Backus–Naur Form* (BNF), an important notation technique for context-free grammars, defining a set of syntax rules for valid tokens.

3.1.2 Qubit/Classical Register Management. The QASM parser automatically flattens the qubit register indices and translates them into a singular range of qubit indices. This process significantly enhances the system's proficiency for transpilation and simulation by replacing the typically used *REG_NAME[INDEX]* qubit addressing, seen in QASM, with a more streamlined one-dimensional qubit range. Classical registers are used to store the outcomes of measurements from qubit registers, typically achieved through commands such as:

$$\text{measure } q[0] \rightarrow c[0];$$

In this example, 'q' denotes a qubit register, and 'c' denotes a classical register. The QASM parser keeps track of the qubit register remapping, ensuring accurate measurement operations.

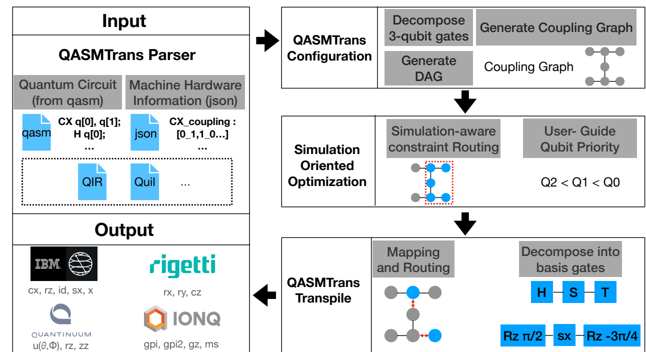

Figure 3: QASMTrans framework, which includes four major components: 1) Input/Output: the Input is the parser that reads in QASM and stores them internally as gate IRs. The Output saves the transpiled circuit in the QASM format. 2) Configuration: perform pre-transpilation work such as generating the coupling graph, gate DAG, and 3-qubit gates decomposition. 3) Simulation-oriented Optimization. 4) Transpilation, including mapping, routing, and decomposition into basis gates of the target device.

3.1.3 Gate Sets and Abstraction. In the rapidly evolving field of quantum computing, it is crucial to have a robust and flexible system capable of accommodating an extensive range of quantum gates, from the most common to the more advanced. QASMTrans currently supports all the gates (except `C4X`) defined by the OpenQASM 2 specification, see Table 1.

The parser supports standard gates such as `Pauli-X`, `Pauli-Y`, `Pauli-Z`, `Hadamard`, `CNOT`, and `Toffoli`, as well as parameterized gates like `RX`, `RY`, `RZ`, and `U` gates. It also accommodates more complex gates like the `SWAP` gate and the controlled versions of various gates. These are by no means an exhaustive list, and the parser's design allows for easy extension to incorporate additional or newer gate types.

Key to the flexibility and functionality of the QASMTrans is the Gate IR. It is a custom C++ class that encapsulates four crucial aspects of each quantum gate:

- **Gate Name**: Represents the type of quantum gate.
- **Target Qubits**: Specifies the individual qubits upon which the quantum gate operation is performed.
- **Gate Parameters**: Contains the parameters relevant to certain quantum gates.
- **Gate Matrix**: Encapsulates the matrix representation of quantum gate, stored as two arrays — one for the real and the other for the imaginary components.

3.2 Transpile configuration

Before the transpilation process, we need to perform some preliminary configuration.

Generation Coupling Graph (full/limited). Based on the topology of the hardware device, we generate a coupling graph that embeds essential elements such as a distance matrix and an adjacent_edge_list. According to the size of the topology, there are two potential approaches: (i) Build the full graph for all the qubits and links. This, however, introduces excessive overhead towards large devices (e.g., the 433-qubit IBM Seattle). (ii) Alternatively, and in most cases, the qubit number of a circuit is smaller than that of the device. Thus, we can limit the qubits and links of the device (through a partial coupling graph) that are taken into the transpilation consideration, drastically shrinking the search space.

Directed Acyclic Graph (DAG) Generation. From the input circuit, a DAG can be constructed to indicate the gate dependency. For example, nodes with an in-degree of zero can be executed immediately without any dependency. Otherwise, any nodes with non-zero in-degree require all of their parent nodes to be executed beforehand to satisfy the dependency. Considering the efficiency, we only maintain two lists: one is the front list that contains executable gates; the other is the future list comprises gates for future execution.

Decompose three-qubit gates. In our transpiler, we first decompose all the 3-qubit gates into 1-qubit and 2-qubit gates, given most of the quantum devices use 1-qubit and 2-qubit gates as the basis gate set. For example, the widely used `Toffoli` gate, or `CCX` gate, will be decomposed into 6 `CX` gates and 9 one-qubit gates.

3.3 Routing and mapping

After the initial decomposition of 3-qubit gates, the next step is to map the virtual qubits to the physical qubits. Various strategies exist for performing this mapping and routing, with each method optimized for different targets. For instance, Sabre is designed to minimize the number of swaps required [23]. Time-optimal qubit mapping emphasizes minimizing the circuit depth [48]. The Noise-Adaptive approach is geared towards minimizing the error of the transpiled circuit [43].

In QASMTrans, we use Sabre as the primary approach, due to its significant advantages in compilation time compared to the others. The major remaining overhead in Sabre routing and mapping includes: 1) After the execution of each gate, we need to update the DAG and regenerate the new front list of gates with in-degree equals to 0 in the DAG (if the gate is in the execution list, its dependency must have already been satisfied and it is ready for execution). The original Sabre method traverses the entire circuit (i.e., all DAG nodes) and identifies the gates that are ready to be executed. As QASMTrans is designed to address very deep circuits, this cost of traversing can be huge. To accelerate this process, we propose to keep the same front layer for each step, but only delete the nodes that are just executed, and fetch any new gates whose dependencies are just resolved through the step. Given that in each time step, only n gates can be simultaneously executed, our proposed optimization can essentially reduce the searching cost of Sabre from $O(G)$ where G is the total number of gates, to $O(n)$ where n is the number of qubits. When the gate number is huge, the benefit of this improvement can be tremendous.

2) When a `SWAP` operation is required, selecting the appropriate `SWAP` requires the calculation of all possible swaps, creating a large search space and significant overhead. This is particularly the case for large machine targets. Consequently, we propose a new method that prunes the pool of `SWAP` candidates by constraining the physical qubit area. This will be discussed in Section 3.6.

3.4 Decompose to basis gates

Here we perform the final decomposition towards the basis gates of the device after routing and mapping. The main consideration is efficiency and simplicity, as decomposing into basis gates before routing and mapping can drastically enlarge the search space during routing and mapping.

The decomposition here is a translation from general gates to the targeted basis gates. The basis gate set for IBMQ, Rigetti, Quantinuum, and IonQ can be found in Figure 3. The detailed translation rules can be found in the open-source code of QASMTrans.

3.5 Statistics

Based on the circuits, QASMTrans can print out the following circuit metrics based on statistics of the quantum gates in the circuit. The detailed definition can be found in [21].

- **Circuit Depth** represents the minimum count of time-evolution steps needed to complete a quantum circuit, calculated based on standard QASM gates.
- **Gate Density** indicates the utilization of gate slots during the time evolution of a quantum circuit, similar to pipeline occupancy in classical processors.
- **Retention Lifespan** quantifies the maximum longevity of a qubit within a system. Its relationship with the T1 and T2 time of the device dictates the feasibility of the circuit execution on the targeted device.
- **Measurement Density** evaluates the importance of measurement operations in a circuit, with respect to the overall induction fidelity.
- **Entanglement Variance** measures the balance of entanglement across the qubits for a circuit. It indicates the level of connectivity and the potential error reduction through an advanced transpiler.

3.6 Simulation-oriented Optimization

As mentioned, most of the contemporary circuit inductions are still performed through classical simulations. In QASMTrans, we propose two classical simulation-oriented optimizations during

Table 3: Benchmark information, it shows the qubit number, single-qubit and total gates, Depth of the circuit

Benchmarks	CIrcuit Information			
Name	Qubits	1-q gate	total gate	Depth
square_root	18	1415	2313	1269
vqe_uccsd	8	5320	10K	7252
vqe_uccsd	24	767K	2.2M	1.1M
sat	11	53	53	51
bwt	21	66K	87K	53K
gcm	13	2387	3149	2447
hhl	10	114K	186K	147K
qaoa	6	222	276	110
qec	5	20	30	18
adder	4	17	27	12
adder	10	10	35	24
adder	64	93	212	78
bv	140	419	491	76
ghz	255	256	510	256
qft	320	153K	255K	2550
ising	420	4196	5034	16

transpilation to generate circuits that can be simulated more efficiently.

Constrained qubit routing/mapping. During the routing and mapping phase, instead of considering all the physical qubits of the device, we limit the number and coupling of qubits that will be considered during the transpilation, based on the number of virtual qubits used in the circuit. This is achieved by first adopting the isomorphic algorithm to find the most relevant connected graph from the hardware architecture, using the number of virtual qubits as input. The qubits of the obtained graph should contain equal or more qubits than the circuit virtual qubits, but less or equal to the number of physical qubits in the device. We then refer to the routing algorithm as normal. Although constrained routing and mapping with partial graphs can lead to more swaps, the benefit of simulating fewer qubits can extraordinarily speed up the transpilation process.

User-guided qubit prioritization. Another simulation-oriented optimization is to enforce user-defined qubit prioritization. Users can specify a priority order such as $q3 < q1 < q0 < q2$, then for classical simulation, we can perform a qubit remapping with respect to this partial order. This is achieved by counting the number of gates performed on each qubit, sorting, and then re-indexing the qubits to assign high-priority qubits to perform more gates. For example, if $q2$ shows the best performance or least error rate, which is set to have the highest priority, the qubit with the most number of gates can be remapped to it. On the other hand, if the coefficients of $q3$ are distributed across multiple nodes for large-scale distributive simulation (i.e., a global qubit), because of the overwhelming cost from inter-node communication, it is set to the lowest priority, we would want the least number of gates to be mapped to $q3$.

4 EVALUATION

4.1 Experimental setup

We primarily use the NERSC Perlmutter HPC system for our evaluation. Perlmutter is built by HPE. Each of the Cray EX systems is equipped with an AMD EPYC 7763 CPU and four NVIDIA A100 GPUs. The other platforms used for the transpilation are listed in Table 6. We compare QASMTrans to two state-of-the-art and most relevant quantum transpilers for comparison: Qiskit [34] (with Sabre algorithm [23]) and MQT-Qmap [45]. We focus on transpilation efficiency, quality, and fidelity. The efficiency is measured by transpilation time. The quality is measured by the depth, total number of gates, and number of `CX` gates of the transpiled circuit. The fidelity is measured by calculating the fidelity of execution for the transpiled circuit over five real quantum devices: *IBM-Brisbane*, *Rigetti-AspenM2*, *IonQ-Aria1* and *Quantinuum-H1-1*). We test on different benchmark circuits varying from 10 qubits to 400 qubits from QASMBench [21], We show all the benchmark information in Table 3.

4.2 Transpilation Efficiency and Quality

The evaluation results are listed in Table 5. We use IBMQ devices as the transpilation target so that: (i) the basis gate set is `X`, `SX`, `CX`, and `RZ`; (ii) for topology, when the number of qubits of the circuit is less than 27, we use the topology of IBMQ Toronto. When it is larger than 27, we use the topology of the latest 433-qubit IBM Seattle as the objective device.

Quality: Overall, QASMTrans can generate transpiled circuits with comparable depth, gates and 2-qubit gates as Qiskit and Qmap. The slight difference is due to the fact that as the initial effort, QASMTrans hasn't yet implemented or integrated advanced front-end gate transformation & cancellation passes.

Efficiency: As listed, QASMTrans shows a tremendous performance advantage over Qiskit and Qmap for the 16 benchmark circuits. The speedup can be as much as 1111× over Qiskit and 277× over Qmap. In particular, for some challenging circuits, such as the vqe_uccsd_n24 with 2.2M gates, and qft_n320 with 255K gates, neither Qiskit nor Qmap can produce transpiled circuits within a reasonable time (i.e., 1 minute), while QASM can accomplish in 7.9s and 3s, respectively.

Scalability: We further look at the performance scalability. Figure 4 shows the scaling of the transpilation time with respect to the number of gates of the input circuits for the various benchmarks. As can be seen, the performance advantage over Qiskit and Qmap is quite consistent.

4.3 Transpilation Fidelity

To evaluate the correctness of transpilation, we use the transpiled circuits generated by Qiskit and QASMTrans as the inputs, and launch them onto four real NISQ devices (IBMQ, Rigetti, Quantinuum, and IonQ) to assess the difference in their induction results, shown in Figure 5. Please be aware that these input circuits, despite having already been transpiled, may go through another round of internal transpilation or optimization within the backend processing of the NISQ device. This is not under our control. However, we

Table 5: Evaluation of QASMTrans compared to Qiskit and Qmap in terms of transpilation quality and efficiency. QT represents our QASMTrans method ,O1/O2/O3 represent three optimization levels for qiskit.

Benchmarks		Efficiency: Transpilation Time (ms)						Quality: Transpiled by Qiskit-O1/O2/O3/QASMTrans		
Name	# qubit	Qiskit-O1	Qiskit-O2	Qiskit-O3	QMAP	QT	Ratio/O1	single-qubit gate	two-qubit gate	Depth
square_root	18	360	630	2800	50	10	36.00	1516/1417/2404/2078	2623/2518/2249/2851	2566/2518/2249/2641
vqe_uccsd	8	1400	2800	13120	1100	16	87.50	4392/3927/7667/9485	5842/5836/5339/706	8113/7796/9711/13K
vqe_uccsd	24	502000	1167000	4628000	272000	7910	63.46	634K/603K/1.6M/3/4M	2.3M/2.2M/2.1M/3.2M	1.7M/2.2M/2.1M/2.8M
sat	11	170	240	820	90	2	85.00	479/460/560/625	632/617/560/654	765/746/828/768
bwt	21	92000	167000	274000	21300	1910	48.17	268K/263K/485K/408K	635K/630K/584K/585K	504K/499K/546K/497K
gcm	13	320	490	3200	140	0.5	640.00	2386/2187/2352/2386	1354/1357/975/1407	3020/2936/2067/2935
hhl	10	32000	54000	147000	25600	283	113.07	139K/125K/173K/209K	89K/89K/60K/103K	199K/193K/152K/259K
qaoa	6	150	180	420	70	0.51	294.12	324/299/211/948	82/81/61/93	224/203/125/546
qec	5	80	80	130	9	0.074	1081.08	36/32/33/36	15/13/13/13	27/24/23/27
adder	4	80	80	130	20	0.072	1111.11	17/16/16/17	16/16/16/16	20/19/19/19
adder	10	110	110	250	30	0.46	239.13	101/99/131/109	129/120/115/155	191/188/195/200
adder	64	250	370	1190	130	82	3.05	701/698/1092/757	1586/1571/1276/1594	1083/1055/1021/1079
BV	140	130	230	990	90	10	13.00	837/837/969/838	453/389/297/756	332/282/316/375
GHZ	255	260	390	1500	70	54	4.81	3/3/257/3	3575/3543/2843/5426	3165/3163/2850/5110
QFT	320	86000	253000	990000	42100	3040	28.29	36K/19K/284K/23K	336K/305K/276K/342K	26K/19K/23K/29K
Ising	420	1150	2350	5200	420	150	7.67	2097/1960/2058/6296	6131/5825/4309/4702	258/253/160/352

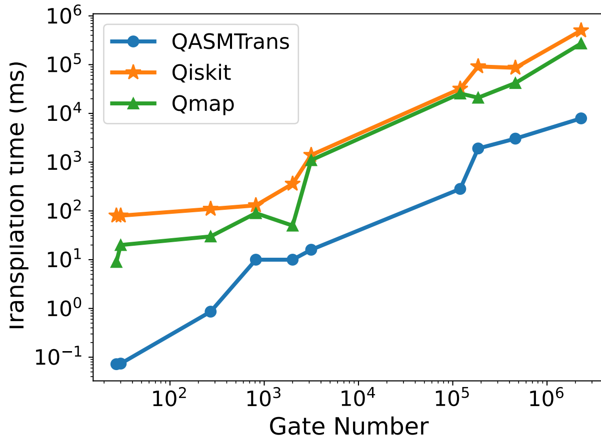

Figure 4: Transpilation time with respect to the number of gates of the input circuits. The advantage of QASMTrans over Qiskit and Qmap is consistent.

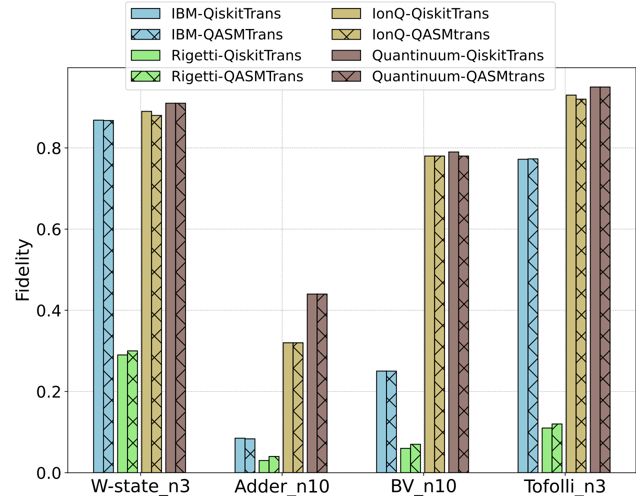

Figure 5: QASMTrans fidelity analysis compared with Qiskit transpiler on different machines, the X-axis shows the benchmarks, and Y-axis shows the fidelity obtained on real NISQ machines (IBM *ibm_brisbane*, Rigetti *Aspen-M2*, IonQ *Aria-1*, and Quantinuum *H2*) with respect to the Qiskit results.

argue that this will not significantly impact the fidelity results since both input circuits go through the same backend processes.

As can be seen in Figure 5, the fidelity with Qiskit result is quite consistent across input circuits and underlying hardware, with $< 1\%$ deviation. This underscores the robustness and stability of QASMTrans.

4.4 Optimization for Classical Simulation

Both the constrained qubit routing/mapping and user-guided qubit prioritization presented in Section III-F can harvest performance gain for classical simulation. Constrained qubit routing/mapping limits the number of qubits for the simulation, for which the performance gain is quite obvious. Here, we mainly focus on demonstrating the benefit of user-guided qubit prioritization.

We have already discussed why minimizing the number of gates over the global qubits can reduce the overhead from communication. Here, we use SV-Sim [20] as the classical simulator. We use all the 8 GPUs from 2 Perlmutter nodes for the distributive circuit simulation. Consequently, 3 qubits are sharing their corresponding coefficients across the 8 GPUs. Figure 6 shows the difference in simulation time for the transpiled circuits with and without user-guided qubit prioritization. The performance gain can be quite significant given the log-scale of the Y-axis. This benefit mainly comes from switching some expensive gates over the three global qubits to local qubits through the final remapping of qubit prioritization.

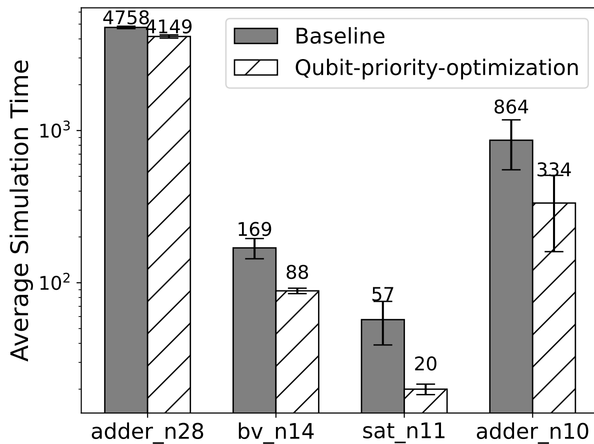

Figure 6: Performance gain in classical simulation through user-guided qubit prioritization using SV-Sim for the transpiled circuits on 8 GPUs of Perlmutter. Note, the Y-axis of simulation time is in log-scale.

4.5 Platform Portability

We evaluate QASMTrans across different computing platforms, from various HPC systems, including NERSC Perlmutter, OLCF Frontier, Crusher, and Summit, ALCF Theta, to a desktop and laptop (Intel P8168 and Apple M2), to an embedded device (JetsonTX2 with ARM8). The platforms are listed in Table 6. The results are shown in Figure 7. The transpilation on all the platforms can be finished within 100s and most of them below 1s.

With these results, we have three observations: (i) QASMTrans can be portable on various platforms, given its efficient C++ based implementation and non-external library dependency (the *json* and *lexertk* are included as header files). In particular, the successful and efficient running on an ARM8 CPU shows the potential of practical deployment on an FPGA of a real quantum system or testbed, such as LBNL AQT. (ii) The transpilation speed across applications circuits and platforms is consistent. (iii) The majority (nearly 90%) of the transpilation time is devoted to routing and mapping for the current implementation of QASMTrans.

5 RELATED WORK

5.1 Quantum Intermediate Representation

In quantum computing, gate IR provides an essential abstraction layer, offering a structured, machine-agnostic representation of quantum circuits. Among the existing quantum IRs, the Microsoft QIR [29] is an LLVM-based IR that defines a set of rules for representing quantum constructs. QIR attempts to serve as a common interface between various quantum languages (e.g., Q#) and platforms. QASM [10] is a widely recognized quantum assembly language developed by IBM for its hardware platforms and software tool-chain. Quil is a portable quantum instruction language developed by Rigetti. Lastly, XACC (eXtreme-scale ACCelerator) [28] is a compilation framework for hybrid quantum-classical computing architectures developed at ORNL, supporting IBM, Rigetti, D-Wave

Table 6: Platforms for Portability Evaluation

Platform	CPU	Vendor	Core	Mem	Compiler
MacBook Pro	Apple M2	Apple	12	16GB	AppleClang 14.0.3
Perlmutter	Authentic AMD	AMD	128	256GB	g++ 11.2.0
JetsonTX2	ARMV8	NVIDIA	4	8GB	g++ 5.4.0
Crusher	Authentic AMD	AMD	128	512GB	g++ 12.2.0
Frontier	Authentic AMD	AMD	128	512GB	g++ 12.2.0
Summit	POWER9	IBM	176	512GB	g++ 9.1.0
Tonga	Intel P8168	Intel	96	128GB	g++ 11.2.0
Theta	Intel Phi 7230 (KNL)	Intel	256	192GB	intel 19.1.0

QPUs, and various classical simulators such as SV-Sim [20] and DM-Sim [22].

5.2 Quantum Transpilation

Quantum transpiler plays a crucial role in quantum computing by translating high-level quantum algorithms into a series of low-level hardware-specific instructions that quantum hardware can execute. Qiskit is a widely used quantum software development package developed by IBM. The Qiskit transpiler provides a flexible and extensible framework, offering a wide array of compilation passes that can be combined in different ways to create customized and hardware-tailored transpilation pipelines.

In addition to Qiskit, there are various transpilers aiming at different purposes: 1) *application-oriented transpilation:* These transpilers focus on specific domain applications. For example, Paulihedral [24] focuses on VQE, Twoqan [18] concentrates on QAOA circuits. 2) *hardware-oriented transpilation:* These transpilers focus on supporting the new features of a particular quantum platform. For instance, CaQR emphasizes the support for dynamic circuit generation and the opportunities from qubit reset [15]. Pulse transpilers delve into the nuances of low-level pulse scheduling, optimizing quantum operations at the physical layer [7, 12, 36]. AutoComm [47] and QuComm [46] present transpiler optimization techniques for distributive quantum devices. 3) *Optimization for mapping/routing:* there are a bunch of works aiming at improving general transpilation performance, like Sabre [23] and Zulhner [49] attempt to minimize the number of additional gates in mapping/routing. TOQM [48] aims at shrinking the depth of the transpiled circuit. Shi *et al.* [36] presents the complete transpilation and optimization flow, including gate aggregation and cancellation. QASMTrans falls into the third category, aiming at improving the transpilation performance of large and deep QASM circuits.

6 CONCLUSION

In this paper, we present QASMTrans, a C++ based quantum transpiler framework for NISQ devices. It outperforms prevalent counterparts, notably achieving up to more than 300× speedups over the Qiskit transpiler. We demonstrate the quality, efficiency, and fidelity

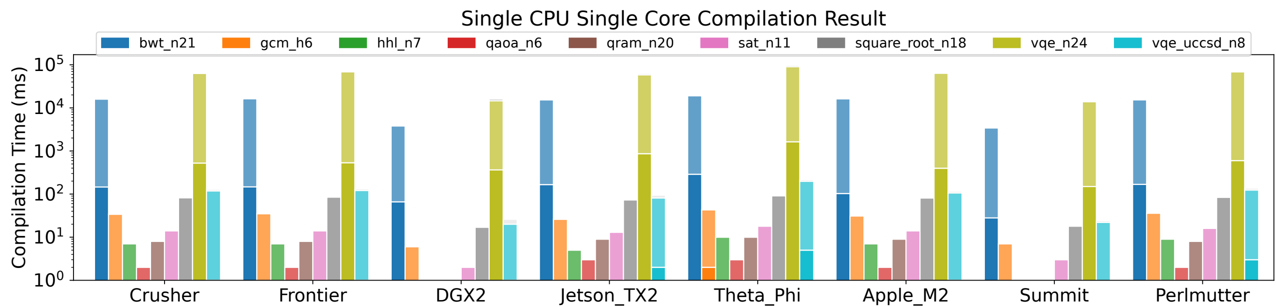

Figure 7: Compilation time on various platforms. The X-axis shows the names of different platforms, while the Y-axis is the compilation time using a single core of a CPU of the system. The empty bars indicate the condition that the compilation time is less than 1ms. The breakdown of each bar implies the time of (upper) routing & mapping, and (lower) decomposition. Please be aware that the Y-axis is in the log scale.

of QASMTrans across various classical and quantum platforms. Future work includes continuously improving QASMtrans by adding new passes such as gate cancellation, new platform support such as for distributed quantum computing and cavity-based systems, as well as the support of new input/output formats such as QIR.

ACKNOWLEDGMENTS

This material is mainly based upon work supported by the U.S. Department of Energy, Office of Science, National Quantum Information Science Research Centers, Co-design Center for Quantum Advantage (C2QA) under contract number DE-SC0012704. The contribution from Meng Wang, Yufei Ding, and Travis Humble are supported by the U.S. Department of Energy, Office of Science, National Quantum Information Science Research Centers, Quantum Science Center (QSC). This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. We acknowledge support from Microsoft's Azure Quantum for providing credits and access to the ion-trap quantum hardware. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC05-76RL01830.

REFERENCES

[1] [n. d.]. Cirq, a python framework for creating, editing, and invoking Noisy Intermediate Scale Quantum (NISQ) circuits. ([n. d.]). url-https://github.com/quantumlib/Cirq.
[2] Alán Aspuru-Guzik and Philip Walther. 2012. Photonic quantum simulators. *Nature physics* 8, 4 (2012), 285–291.
[3] H-J Briegel, Tommaso Calarco, Dieter Jaksch, Juan Ignacio Cirac, and Peter Zoller. 2000. Quantum computing with neutral atoms. *Journal of modern optics* 47, 2-3 (2000), 415–451.
[4] Yudong Cao, Jonathan Romero, Jonathan P Olson, Matthias Degroote, Peter D Johnson, Mária Kieferová, Ian D Kivlichan, Tim Menke, Borja Peropadre, Nicolas PD Sawaya, et al. 2019. Quantum chemistry in the age of quantum computing. *Chemical reviews* 119, 19 (2019), 10856–10915.
[5] Marco Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, et al. 2021. Variational quantum algorithms. *Nature Reviews Physics* 3, 9 (2021), 625–644.
[6] 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. *Physical Review X* 10, 1 (2020), 011022.
[7] Yanhao Chen, Yuwei Jin, Fei Hua, Ari Hayes, Ang Li, Yunong Shi, and Eddy Z Zhang. 2023. A Pulse Generation Framework with Augmented Program-aware Basis Gates and Criticality Analysis. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 773–786.
[8] Juan I Cirac and Peter Zoller. 1995. Quantum computations with cold trapped ions. *Physical review letters* 74, 20 (1995), 4091.
[9] John Clarke and Frank K Wilhelm. 2008. Superconducting quantum bits. *Nature* 453, 7198 (2008), 1031–1042.
[10] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. 2017. Open quantum assembly language. *arXiv preprint arXiv:1707.03429* (2017).
[11] Vedran Dunjko and Hans J Briegel. 2018. Machine learning & artificial intelligence in the quantum domain: a review of recent progress. *Reports on Progress in Physics* 81, 7 (2018), 074001.
[12] Pranav Gokhale, Ali Javadi-Abhari, Nathan Earnest, Yunong Shi, and Frederic T Chong. 2020. Optimized quantum compilation for near-term algorithms with openpulse. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 186–200.
[13] Loïc Henriet, Lucas Beguin, Adrien Signoles, Thierry Lahaye, Antoine Browaeys, Georges-Olivier Reymond, and Christophe Jurczak. 2020. Quantum computing with neutral atoms. *Quantum* 4 (2020), 327.
[14] Zoe Holmes, Gopikrishnan Muraleedharan, Rolando D Somma, Yigit Subasi, and Burak Şahinoğlu. 2022. Quantum algorithms from fluctuation theorems: Thermal-state preparation. *Quantum* 6 (2022), 825.
[15] Fei Hua, Yuwei Jin, Yanhao Chen, Suhas Vittal, Kevin Krsulich, Lev S Bishop, John Lapeyre, Ali Javadi-Abhari, and Eddy Z Zhang. 2023. CaQR: A Compiler-Assisted Approach for Qubit Reuse through Dynamic Circuit. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 59–71.
[16] IBM. [n. d.]. IBM Quantum. URL: https://quantum-computing.ibm.com/.
[17] Walter Kauzmann. 2013. *Quantum chemistry: an introduction*. Elsevier.
[18] Lingling Lao and Dan E. Browne. 2021. 2QAN: A quantum compiler for 2-local qubit Hamiltonian simulation algorithms. https://doi.org/10.48550/ARXIV.2108.02099
[19] Dietrich Leibfried, Rainer Blatt, Christopher Monroe, and David Wineland. 2003. Quantum dynamics of single trapped ions. *Reviews of Modern Physics* 75, 1 (2003), 281.
[20] Ang Li, Bo Fang, Christopher Granade, Guen Prawiroatmodjo, Bettina Heim, Martin Roetteler, and Sriram Krishnamoorthy. 2021. SV-Sim: Scalable pgas-based state vector simulation of quantum circuits. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
[21] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2023. Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM Transactions on Quantum Computing* 4, 2 (2023), 1–26.
[22] Ang Li, Omer Subasi, Xiu Yang, and Sriram Krishnamoorthy. 2020. Density matrix quantum circuit simulation via the BSP machine on modern GPU clusters. In *Sc20: international conference for high performance computing, networking, storage and analysis*. IEEE, 1–15.
[23] 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. ACM, 1001–1014.

[24] 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*. 554–569.

[25] Sophia Fuhui Lin, Sara Sussman, Casey Duckering, Pranav S Mundada, Jonathan M Baker, Rohan S Kumar, Andrew A Houck, and Frederic T Chong. 2022. Let each quantum bit choose its basis gates. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1042–1058.

[26] Filip B Maciejewski, Zoltán Zimborás, and Michał Oszmaniec. 2020. Mitigation of readout noise in near-term quantum devices by classical post-processing based on detector tomography. *Quantum* 4 (2020), 257.

[27] R Maurand, X Jehl, D Kotekar-Patil, A Corna, H Bohuslavskyi, R Laviéville, L Hutin, S Barraud, M Vinet, M Sanquer, et al. 2016. A CMOS silicon spin qubit. *Nature communications* 7, 1 (2016), 1–6.

[28] Alexander J McCaskey, Dmitry I Lyakh, Eugene F Dumitrescu, Sarah S Powers, and Travis S Humble. 2020. XACC: a system-level software infrastructure for heterogeneous quantum–classical computing. *Quantum Science and Technology* 5, 2 (2020), 024002.

[29] Microsoft. 2023. Quantum intermediate representation. https://learn.microsoft.com/en-us/azure/quantum/concepts-qir

[30] Jeremy L O'Brien, Akira Furusawa, and Jelena Vučković. 2009. Photonic quantum technologies. *Nature Photonics* 3, 12 (2009), 687.

[31] Arash Partow. [n. d.]. Simple C++ Lexer Toolkit Library. https://github.com/ArashPartow/lexertk.

[32] Jarryd J Pla, Kuan Y Tan, Juan P Dehollain, Wee H Lim, John JL Morton, David N Jamieson, Andrew S Dzurak, and Andrea Morello. 2012. A single-atom electron spin qubit in silicon. *Nature* 489, 7417 (2012), 541–545.

[33] John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (2018), 79.

[34] QISKit: Open Source Quantum Information Science Kit. [n. d.]. https://https://qiskit.org/.

[35] Chad Rigetti, Jay M Gambetta, Stefano Poletto, BLT Plourde, Jerry M Chow, AD Córcoles, John A Smolin, Seth T Merkel, JR Rozen, George A Keefe, et al. 2012. Superconducting qubit in a waveguide cavity with a coherence time approaching 0.1 ms. *Physical Review B* 86, 10 (2012), 100506.

[36] 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)*. ACM, New York, NY, USA, 1031–1044. https://doi.org/10.1145/3297858.3304018

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

[38] Samuel Stein, Nathan Wiebe, Yufei Ding, Peng Bo, Karol Kowalski, Nathan Baker, James Ang, and Ang Li. 2022. EQC: ensembled quantum computing for variational quantum algorithms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 59–71.

[39] Samuel A Stein, Betis Baheri, Daniel Chen, Ying Mao, Qiang Guan, Ang Li, Shuai Xu, and Caiwen Ding. 2022. Quclassi: A hybrid deep neural network architecture based on quantum state fidelity. *Proceedings of Machine Learning and Systems* 4 (2022), 251–264.

[40] I Stetcu, A Baroni, and J Carlson. 2022. Projection algorithm for state preparation on quantum computers. *arXiv preprint arXiv:2211.10545* (2022).

[41] 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

[42] Swamit S Tannu and Moinuddin K Qureshi. 2019. Mitigating measurement errors in quantum computers by exploiting state-dependent bias. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 279–290.

[43] 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)*. ACM, New York, NY, USA, 987–999. https://doi.org/10.1145/3297858.3304007

[44] Andreas Wichert. 2020. *Principles of quantum artificial intelligence: quantum problem solving and machine learning*. World Scientific.

[45] Robert Wille and Lukas Burgholzer. 2023. MQT QMAP: efficient quantum circuit mapping. In *Proceedings of the 2023 International Symposium on Physical Design*. 198–204.

[46] Anbang Wu, Yufei Ding, and Ang Li. 2022. CollComm: Enabling Efficient Collective Quantum Communication Based on EPR buffering. *arXiv preprint arXiv:2208.06724* (2022).

[47] Anbang Wu, Hezi Zhang, Gushu Li, Alireza Shabani, Yuan Xie, and Yufei Ding. 2022. AutoComm: A Framework for Enabling Efficient Communication in Distributed Quantum Programs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1027–1041.

[48] Chi Zhang, Ari Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Edd Z. Zhang. 2021. Time-Optimal Qubit Mapping. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. ACM, Virtual.

[49] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2018. Efficient mapping of quantum circuits to the IBM QX architectures. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1135–1138.