IBM's Qiskit Tool Chain:
Working with and Developing for Real Quantum Computers
(Special Session Summary)

Robert Wille[1] Rod Van Meter[2] Yehuda Naveh[3]
[1]Institute for Integrated Circuits, Johannes Kepler University, Linz, Austria
[2]Keio University, Japan
[3]IBM Research – Haifa, Israel
robert.wille@jku.at, rdv@sfc.wide.ad.jp, naveh@il.ibm.com

***Abstract*—Quantum computers promise substantial speedups over conventional machines for many practical applications. While considered "dreams of the future" for a long time, first quantum computers are available now which can be utilized by anyone. A leading force within this development is IBM Research which launched the *IBM Q Experience* – the first industrial initiative to build universal quantum computers and make them accessible to a broad audience through cloud access. Along this initiative, the tool *Qiskit* has been launched which enables researchers, teachers, developers, and general enthusiasts to write corresponding code and to run experiments on those machines. At the same time, this provides an ideal playground for the design automation community which – through Qiskit – can deploy improved solutions e.g. on designing and realizing quantum applications. This special session summary aims to provide an introduction into Qiskit and is showcasing selected success stories on how to work with and develop for it. In addition to that, it provides corresponding references to further readings in terms of tutorials and scientific papers as well as links to publicly available implementations for Qiskit extensions.**

I. Introduction

Quantum computers [20] promise substantial speedups over conventional computers for many practical applications such as quantum chemistry, optimization, machine learning, cryptography, quantum simulation, systems of linear equations, and many more [24]. While considered "dreams of the future" which mainly electrified the academic community only, recent accomplishments leading to the first real quantum computers which can be utilized by everyone make this topic more and more relevant for the interested mainstream. A leading force within this development is IBM Research which launched the *IBM Q Experience* in 2017 [1].

This initiative represents the first industrial approach to build universal quantum computers *and* make them accessible to a broad audience through cloud access. While the project initially started with the 5-qubit quantum processor *IBM QX2* in March 2017, today it offers a total of four available machines, plus additional in development. As of today, IBM Q machines have been used by more than 100,000 users, who have run more than 6.5 million experiments, resulting in more than 100 academic papers. Moreover, a worldwide network of Fortune 500 companies, academic institutions, and startups work within this initiative and collaborate to advance quantum computing. In addition to real-hardware machines, IBM also provides state-of-the-art simulators for simulating quantum programs on conventional machines.

In order to write corresponding code and run experiments on those quantum computers, IBM also launched *Qiskit* – an open-source framework aimed for researchers, teachers, developers, and general enthusiasts. For the design automation community, this is an ideal playground. In fact, many problems in the domain of quantum computing can be addressed perfectly by automated methods and experiences [32]. Unfortunately, there is still far too little coordination between the design automation community and the quantum community. Consequently, many automatic approaches proposed in the past have either addressed the wrong problems or failed to reach the end users. Qiskit provides an ideal platform to bring together both communities and to exploit those synergies.

This special summary aims to foster this potential by providing an introduction into Qiskit as well as showcasing selected success stories on how to work with and develop for it. The descriptions shall provide an entry point for the interested but yet unexperienced reader. In combination with the references to further readings in terms of tutorials and scientific papers as well as links to publicly available implementations for Qiskit extensions, this shall equip the reader to efficiently design and execute own applications on a real quantum computer. To this end, we will provide a brief overview of different aspects, namely

- a short high-level description of Qiskit (covered in Section III)
- the user's perspective on how Qiskit can be utilized to actually work with quantum computers and simulators (covered in Section IV), as well as,
- the developer's perspective on how to develop new methods for Qiskit, possibly outperforming existing state of the art solutions by using expertise from design automation (covered in Section V).

Before that, a basic introduction into quantum computation, IBM QX, and the corresponding QX architectures are provided in the next section.

978-3-9819263-2-3/DATE19/©2019 EDAA

II. Background

Before we start diving into Qiskit, this section first provides a brief review on the basics of quantum computation in general and the IBM Q project as well as the corresponding QX architectures in particular. Note that a comprehensive review of the wide field of quantum computation is out of scope for this summary paper. Hence, for anyone who wishes to enter the field, we refer to a more detailed treatment of the basics as provided e.g. in [20].

A. Quantum Computation

Quantum computation significantly differs from the conventional computation paradigm. Conventional computations and circuits use bits as information units. In contrast, quantum circuits perform their computations on qubits [20]. These qubits can not only be in one of the two basis states $|0\rangle$ or $|1\rangle$, but also in a superposition of both – allowing for the representation of all possible 2^n basis states of n qubits concurrently. This so-called quantum parallelism, together with quantum correlations in the form of entanglement and quantum interference effects, serve as the basis for algorithms that are significantly faster on quantum computers than on conventional machines.

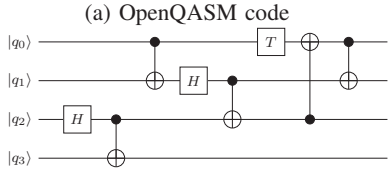
To this end, the qubits of a quantum circuit are manipulated by quantum operations represented by so-called quantum gates. These operations can either operate on a single qubit, or on multiple ones. For multi-qubit gates, we distinguish target qubits and control qubits. The value of the target qubits is modified in the case that the control qubits are set to basis state $|1\rangle$. The *Clifford+T* library [10], which is composed of the single-qubit gates *H* (Hadamard gate) and *T* (Phase shift by $\pi/4$), as well as the two-qubit gate *CNOT* (controlled NOT), represents a universal set of quantum operations (i.e. all quantum computations can be implemented by a circuit composed of gates from this library).

To describe quantum circuits, high level quantum languages (e.g. Scaffold [9] or Quipper [13]), quantum assembly languages (e.g. OpenQASM 2.0 developed by IBM [12]), or circuit diagrams are employed. In a circuit diagram, qubits are represented by horizontal lines, which are passed through quantum gates. In contrast to conventional circuits, this however does not describe a connection of wires with a physical gate, but defines (from left to right) in which order the quantum gates are applied to the qubits.

Example 1. *Fig. 1 shows a quantum circuit described in OpenQASM (a) as well as in terms of a circuit diagram (b).*

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[4];
h q[2];
cx q[2],q[3];
cx q[0],q[1];
h q[1];
cx q[1],q[2];
t q[0];
cx q[2],q[0];
cx q[0],q[1];
```

(a) OpenQASM code



(b) Circuit diagram

Fig. 1: Descriptions of a quantum circuit

B. IBM QX and Corresponding QX Architectures

The *IBM Quantum Experience* (IBM QX, [3]) is a web portal which allows users to write quantum programs, either in OpenQASM or through a graphical interface, and run them on actual IBM Q hardware or on conventional simulators of the quantum hardware. IBM Q implementation consist of superconducting transmon qubits [17] on silicon chips. Control and measurements are conducted through microwave pulses transfered into and out of dilution refrigirators, in which the quantum chips are set at an operating temperature of around 15 mK. Communication into, out of, and among the qubits is done through on-chip resonators. Diagrams of the various IBM Q quatum chips can be observed in [2].

The *IBM QX* architectures support the elementary single qubit operation $U(\theta, \phi, \lambda) = R_z(\phi)R_y(\theta)R_z(\lambda)$ (i.e. an Euler decomposition) that is composed by two rotations around the z-axis and one rotations around the y-axis, as well as the CNOT operation. By adjusting the parameters θ, ϕ, and λ, single-qubit operations can be realized. For specific gates such as H or T, direct implementations are also available.

The first backend composed of 5 qubits and called *IBM QX2* was launched in March 2017. In June 2017, IBM launched a second one called *IBM QX3* which is composed of 16 physical qubits that are connected with coplanar waveguide bus resonators [4]. In September 2017, IBM launched revised versions of their 5-qubit and 16-qubit backends named *IBM QX4* and *IBM QX5*, respectively.

When executing quantum circuits or algorithms (such as sketched in the previous section) on these architectures, coupling restrictions have to be satisfied. In fact, the user first has to decompose all non-elementary quantum operations (e.g. Toffoli gate, SWAP gate, or Fredkin gate) to the elementary operations $U(\theta, \phi, \lambda)$ and $CNOT$. Moreover, two-qubit gates, i.e. CNOT gates, cannot arbitrarily be placed in the architecture but are restricted to prescribed pairs of qubits only. Even within these pairs, it is firmly defined which qubit is the target and which is the control. These restrictions are given by the so-called *coupling-map* illustrated in Fig. 2, which sketches the layout of the *IBM QX4* architecture. The circles indicate physical qubits (denoted by Q_i) and arrows indicate the possible CNOT applications, i.e. an arrow pointing from physical qubit Q_i to qubit Q_j defines that a CNOT with control qubit Q_i and target qubit Q_j can be applied. These restrictions are called *CNOT-constraints* and need to be satisfied in order to execute a quantum circuit on a QX architecture.

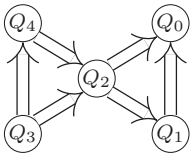


Fig. 2: Coupling map of the *IBM QX4* architectures [4]

III. The Qiskit Toolset

Qiskit is an end-to-end open-source software library for quantum computing, covering the full stack from the actual interaction with the IBM Q hardware, through simulation and emulation, and up to application-level algorithms. The tool itself is thereby arranged in four libraries named after the four classical elements terra, aqua, aer, and ignis. In the following, each library is briefly discussed.

Terra: The Terra library covers all low-level sections of Qiskit. These include tools for specifying and manipulating quantum circuits through the OpenQASM language [12], or at the pulse levels through OpenPulse [19]. It provides transpilers to make quantum circuits more optimized for running on real hardware e.g. by minimizing occurrences of CNOT gates. This way, the user can write a circuit which captures the required functionality without investing much effort in optimizing for the specific hardware, and then letting the transpiler find a more optimized circuit while maintaining the exact functionality prescribed by the user. Terra also includes infrastructure for specifying and modeling physical noise processes. These are especially important in order to analyze behavior of a quantum algorithm when run on a noisy quantum computer, as is the case with present-day hardware. Finally, Terra provides the suitable data structures and interfaces to define the various software constructs relevant to quantum computing, and pass those constructs among the different Qiskit libraries, and to the hardware.

Aqua: The Aqua library implements the other edge of the spectrum – the high-level quantum algorithms for a multitude of applications. Here, the user is provided with high level interfaces he or she can use in order to be able to use quantum hardware and simulators, but without the necessity to learn the details of how to construct quantum circuits. Once the user provides the structure and parameters of the application, the actual quantum circuits are created by Aqua using the Terra constructs. In addition, conventional flows are constructed to build and run the quantum circuit, such that each application is transformed into a conventional-quantum hybrid algorithm. Those algorithms can then run on a conventional machine which in turn calls the quantum hardware (or a conventional simulator of the quantum hardware) in order to implement the full application. Aqua provides solutions taken from application domains such as chemistry and finance. Many of those applications are based on implementations of hybrid conventional-quantum algorithms from machine learning, optimization, and other underlying technologies. Most notably, the *Variational Quantum Eigensolver* (VQE) algorithm [15] is at the basis of many of Aqua's applications. Tuning this algorithm (e.g. specifying the optimization procedure to be used by the algorithm) can be done by the user, or be set as default by the Aqua application. Hence, Aqua provides the full range from simple push-button applications, to full tunability by the user.

Aer: Aer is expected to include a set of simulators and emulators for running quantum circuits and applications on conventional machines. This can serve various practices. It will provide handy educational means to explore and experiment with quantum circuits and algorithms without the burden of waiting for the scarcer quantum hardware. It will also allow the exploration of the behavior of quantum hardware under controlled conditions e.g. by injecting specific noise processes into the circuits and observing their effect on the results. Finally, it will allow fast development of quantum algorithms by again allowing for a highly-accessible way to run quantum algorithm prototypes on a conventional machine. These algorithms can be run on "clean" (noiseless) simulators in order to observe the expected results and enable design-space exploration. Subsequently, the algorithms can also be run on noisy simulators in order to analyze to what extent realistic noise levels deteriorate the results of the algorithms. The Aer library is not yet released.

Ignis: Finally, the Ignis library will include all constructs and implementations of methods related to quantum hardware characterization, verification, mitigation, and correction. These include methods of rigorously categorizing and analyzing noise processes in the hardware through randomized benchmarking, tomography, and multi-faceted comparisons with simulation. It will also include pulse schemes for mitigation of systematic gate-implementation errors, as well as a portfolio of error correcting codes and algorithms. Also the Ignis library is not yet released.

Taken together, the four Qiskit libraries provide the most comprehensive back-to-back software solutions for quantum computing, all seamlessly connected and communicating through the same data structure constructs. The Qiskit software library is complemented by a thorough tutorial library [8] covering the full range of tutorials from novice to the expert, and from quantum theory to low-level notebook-assisted implementations of special-purpose quantum circuits.

IV. User's Perspective: Working with Qiskit

Using Qiskit, anyone can easily define and execute desired quantum computations, through cloud access, all the way down to the actual quantum computer. To this end, proper description means and methods e.g. for simulation or mapping to corresponding architectures are provided. In the following, we illustrate that by a quick run-through how to install and how to make the first steps with the tool. By this, we provide a brief glimpse into the user's perspective of Qiskit.

Qiskit can be downloaded through https://qiskit.org/ and the links to corresponding github-repositories provided there. After downloading the tool, it can be installed by executing:

```
$ mkdir qiskit/
$ cd qiskit/
$ python3 -m venv .qiskitvenv
$ source .qiskitvenv/bin/activate
[within .qiskitvenv]$ pip install qiskit
```

For the hardware backends (i.e. the access to the QX architectures), additionally a registration at IBM QX is required which yields a token to be deposited.[1] Afterwards, you can load the QX architecture you would like to work with by running a Python script with:

```
from qiskit import IBMQ
IBMQ.load_accounts()
ibmqx4 = IBMQ.get_backend('ibmqx4')
```

In this example, this loads the QX4 architecture whose corresponding coupling map is shown in Fig. 2.

Having that, any desired quantum circuit to be executed on this architecture can be defined; either directly in Python or through one of the available languages such as the OpenQASM. For example, the quantum circuit depicted in Fig. 1b can be defined in Python by:

```
from qiskit import QuantumCircuit, QuantumRegister,
    ClassicalRegister
q = QuantumRegister(4, 'q')
circ = QuantumCircuit(q)
circ.h(q[2])
circ.cx(q[2], q[3])
circ.cx(q[0], q[1])
circ.h(q[1])
circ.cx(q[1], q[2])
circ.t(q[0])
circ.cx(q[2], q[0])
circ.cx(q[0], q[1])
```

Alternatively, the quantum circuit can be defined in OpenQASM as shown in Fig. 1a and, afterwards, loaded into Qiskit using the command `load_qasm_file`.

Then, as reviewed in Section II-B, every circuit has to be properly mapped for the respective architecture, i.e. the coupling restrictions sketched by the coupling map shown in Fig. 2 have to be satisfied. Qiskit offers corresponding methods for this *compiling* process (in the community, this is also often referred to as *mapping* process). More precisely, the circuit considered here can be made compatible for the QX4 architecture using:

```
from qiskit import compile, qobj_to_circuits
qobj = compile(circ, ibmqx4)
compiled_circ = qobj_to_circuits(qobj)[0]
```

The resulting circuit eventually can be executed on a real quantum computer. To this end, however, some measurements should be defined to make clear what outputs are of interest. If we are interested in the result of all qubits after the execution, this can be set up by:

```
from qiskit import execute, Aer
c = ClassicalRegister(5, 'c')
measurement = QuantumCircuit(q, c)
measurement.measure(q, c)
measured_circ = circ + measurement
```

This can then be simulated e.g. by executing:

```
from qiskit.tools.visualization
    import plot_histogram
job = execute(measured_circ,
    backend=Aer.get_backend('qasm_simulator'))
result = job.result()
plot_histogram(job.result().get_counts())
```

If the simulation shows the intended result (visualized by a plot generated by the last line), an execution on a real quantum device can be triggered by changing the backend from "qasm_simulator" to the previously loaded backend "ibmqx4" in the `execute`-command. In contrast to the simulation, this eventually yields results that have indeed be generated by a real quantum computer.

The run-through from above of course only provided a glimpse of Qiskit's functionality. However, Qiskit provides an extensive documentation which can be found at https://qiskit.org/documentation/ and provides a more detailed treatment. Furthermore, several tutorials (in terms of Jupyter Notebooks) are available at [8]. Using them provides an easy entry point to working with real quantum computers.

V. Developer's Perspective: Improving Qiskit

Although Qiskit is a powerful tool, it still offers much room for improvement. In fact, many problems to be addressed by Qiskit are solved in a rather straight-forward fashion and provide potential for enhancement. In this section, we exemplify that by the two representative functions introduced previously in Section IV: simulation and compilation (called mapping in the following). More precisely, we sketch how simulation can be improved by utilizing a dedicated data-structure (in terms of decision diagrams) and how the mapping procedure can be improved by utilizing different heuristics. Afterwards, we briefly discuss how improvements like this can be incorporated into Qiskit. By this, we provide a brief glimpse into the developer's perspective of Qiskit.

A. Improving Simulation

Simulation takes a given quantum state (usually denoted by $|\psi\rangle$) and determines its transformation when a sequence of quantum operations is applied to it. This requires a proper mathematical description of both quantum states and quantum operations.

Usually, the state of a single qubit is described by $|\psi\rangle = \alpha_0 \cdot |0\rangle + \alpha_1 \cdot |1\rangle$, where $\alpha_0, \alpha_1 \in \mathbb{C}$ denote the *amplitudes* which indicate how much the qubit is related to the basis states $|0\rangle$ and $|1\rangle$, respectively.[2] If a quantum system is

[1] Details on this are provided at https://qiskit.org/documentation/install.html.

[2] Note that the amplitudes of a quantum state $|\psi\rangle$ must satisfy the normalization constraint $|\alpha_0|^2 + |\alpha_1|^2 = 1$.

composed of multiple, i.e. $n > 1$, qubits, the description can accordingly be extended to

$$|\psi\rangle = \sum_{x\in\{0,1\}^n} \alpha_x \cdot |x\rangle\,, \text{ where } \sum_{x\in\{0,1\}^n} |\alpha_x|^2 = 1 \text{ and } \alpha_x \in \mathbb{C}.$$

Such states can be also represented by a column vector $\psi = [\psi_i]$ with $0 \leq i < 2^n$ and $\psi_i = \alpha_x$, where $nat(x) = i$.

In turn, quantum operations are described by unitary matrices, i.e. complex square matrices whose inverse is their conjugate transposed. Prominent examples (working on single qubits) include e.g.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \text{and } Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix},$$

where X complements the current state of the qubit, H adjusts the state of superposition of the qubit, and Z changes the phase of the qubit. Besides that, an operation involving two qubits is e.g. defined by

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

and performs a so-called controlled inversion.

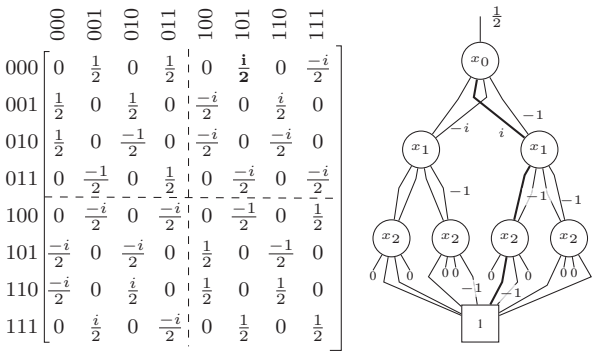
Having both, a description of a quantum state and a quantum operation in terms of a state vector and a unitary matrix, respectively, the transformation executed by that can easily be described through matrix-vector multiplication. For example, applying a *CNOT* operation to a two qubits system which currently is in state $|\psi\rangle = |11\rangle$ yields a successor state defined by

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{CNOT} \cdot \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}}_{\psi} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \equiv |10\rangle\,.$$

Accordingly, simulating a quantum circuit conceptually boils down to a sequence of matrix-vector multiplications using a given input quantum state as well as the matrices provided by the operations defined in the circuit.

However, a serious obstacle in the simulation of quantum circuits is that the corresponding descriptions (the vectors and matrices) grow exponentially with respect to the number of qubits. This poses a limit to simulation techniques – including the one used in Qiskit. In order to address that, many researchers are investigating alternative approaches in order to optimize simulation. The spectrum includes solutions utilizing parallelization (as done e.g. in [16], [29]), emulation (as done e.g. in [14], [30]), or decision diagrams [31], [40].

In the following, we briefly review the approach based on decision diagrams (as this has also been integrated into Qiskit). Using decision diagrams allows for a much more compact representation of the exponentially large matrices and vectors and already led to substantial improvements e.g. for synthesis [21], [23], [41] or verification [22], [33]. The main idea is briefly



(a) Matrix (b) Decision diagram

Fig. 3: Matrix and decision diagram of a 3-qubit computation

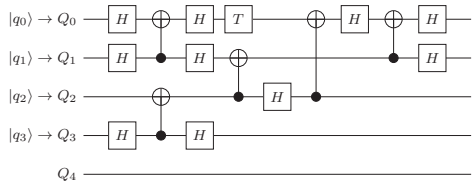
sketched by means of Fig. 3 which shows the matrix and a functionally equivalent decision diagram comprised of a 3-qubit quantum operation. Here, the $2^n \times 2^n$ matrix (represented by the top node) is split into four sub-matrices of dimension $2^{n-1} \times 2^{n-1}$ (represented by the top node's successors). Doing this recursively eventually yields a "sub-matrix" which is composed of a single entry only. When additionally sharing (structurally) equivalent sub-matrices by the same node (as it is e.g. the case for the top-left and bottom-left sub-matrix in Fig. 3a which only differ by the factor $-i$), a representation may result which is much more compact than the exponentially large matrix representation. The difference in the factor is represented by corresponding edge weights ($-i$ in case of the third successor; if no edge weight is annotated, the factor of 1 is assumed).

Overall, this allows for a much faster simulation of quantum computations as described and evaluated in detail in [40]. On top of that, further optimizations are possible with respect to the precision of the simulation (c.f. [38]) or the run-time performance (c.f. [43]). As briefly sketched later in Section V-C, such improvements can be (and have been) integrated into Qiskit.[3]

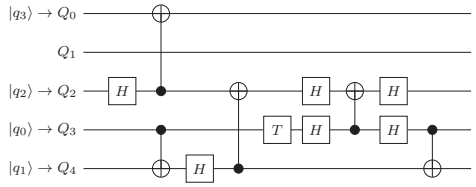
B. Improving the Mapping to QX Architectures

Besides simulation, mapping a given quantum circuit to the desired QX architecture constitutes another important step of the Qiskit tool-chain (see the compilation step discussed in Section IV). Recall that, here, it has to be ensured that every two-qubit operations is performed on qubits which are adjacent in the coupling map of the considered architecture and that the respective positions of control and target qubits are in-line (as defined e.g. in the coupling map illustrated in Fig. 2 and discussed in Section II-B). This may require the mapping of qubits to change during the execution of a quantum circuit. To this end, Hadamard (H) and SWAP gates (implemented as three alternating CNOT gates) can be applied

[3]More details on that, including an open-source implementation is available at http://iic.jku.at/eda/research/quantum_simulation/.



(a) Mapped circuit obtained by Qiskit



(b) Optimized circuit

Fig. 4: Mapping to QX4 architectures

to flip the direction of control and target qubits and to change the mapping of the logical qubits, respectively.

For example, consider again the quantum circuit shown in Fig. 1. Just mapping all qubits q_i to corresponding physical qubits Q_i (i.e. conducting a 1:1 mapping) does not work since the QX4 architecture prohibits e.g. the interaction between q_2 as a control and q_3 as a target in the second gate (only the opposite is allowed) or between q_0 as a control and q_1 as a target in the third gate. This can be resolved by adding additional H gates as shown in Fig. 4a. This circuit results when applying the command `compile` in the corresponding compilation step discussed before in Section IV.

However, inserting the additional gates to satisfy the constraints imposed by the coupling graph may drastically increases the number of gates – which in turn significantly increase the probability of errors during the computation. Hence, minimizing the number of added H and SWAP gates is a primary objective (this is similar to optimizations for nearest neighbor quantum architectures as considered e.g. in [25]–[27], [34]–[37]). Unfortunately, this is an $\mathcal{NP}$-hard problem (as recently proved in [11]) and, hence, requires efficient methods. Triggered by call for solutions from the Qiskit team, many researchers were motivated to work on improvements. In very short time, this already led to approaches such as presented in [7], [18], [28], [39], [42]. Moreover, even competitions seeking the best possible solution for this problem have been conducted in order to further trigger development in this area (see [6]). They have yielded results as e.g. shown in Fig. 4b, which introduced improved mapping, together with the application of H gates only before and after the seventh gate is applied.[4] This eventually led to a more efficient overall map of the given input circuit to the QX4 architecture.

[4]An implementation of the method generating this result is available at http://iic.jku.at/eda/research/ibm_qx_mapping/.

C. Integrating into Qiskit

Qiskit is a Github-based open source software project. This makes contributing advanced algorithms and methods particularly easy. The regular structure for contributions typically involves creating your own fork of the Qiskit repository, implementing and testing your contributions, and then creating a pull request with the contribution back into Qiskit. This pull request will typically include a description of the contribution and its benefits and drawbacks. In many cases, discussions with the community is a highly effective method to improve your algorithm and make it fit in the general Qiskit framework. Such discussions are most easily evoked by openning an issue in the Qiskit Github space before or during development of the new feature.

Once implementation is done and a pull request was created, the Qiskit community will proceed to review, test, and further analyze your contribution. This process may result in interesting technical discussions which again may evoke ways of making your contribution even more powerful. Eventually, after all discussions have concluded and the code is at its best, it may be pulled into Qiskit and become an itegral part of it.

For more holistic contributions, for example a full software tool such as a new simulator, the Qiskit team has created another way of contributing. Here, the aim is to provide wide access to the tool through the Qiskit package, while retaining the tool's holistic functionality. In order to do that, the tool must first conform to the look and feel of the Qiskit code, and especially support the same interfaces as Qiskit. This allows the wide community using Qiskit to also use the new tool out of the box, minimizing any educational pain in using it. Once this stage is done, a new repository can be created under the general Qiskit repository. Then, the entire code of the new tool can be deposited in this repository. This way, the tool's functionality is retained in whole, while benefiting from the wide dissemination and simplicity of use brought by Qiskit. A prime example of such a full-scale contribution to Qiskit is the decision-diagram based simulator described in Section V-A [5].

Whether a simple improvement or a full-fledged tool, any contribution into Qiskit reaches a vast community of enthusiasts in the field. There is no better way for this contribution to create the impact it is intended for on the future of quantum computing.

VI. Conclusions

This special session summary provided a brief glimpse into IBM's Qiskit which allows researchers, teachers, developers, and general enthusiasts to write code for and to run experiments on real quantum computers. We covered both the user's perspective as well as the developer's perspective. We hope this triggered further interest. In this case, we are referring to further tutorials and references as cited above for a more detailed treatment on the respective issues, as well as to personal interaction with the Qiskit team.

VII. Acknowledgments

We thank the full team of Qiskit developers for enabling this special session summary. We also thank Stefan Hillmich, Alexandru Paler, and Alwin Zulehner for their specific contributions to the work presented here. We acknowledge the use of IBM Q for this work. The views expressed are those of the authors and do not reflect the official policy or position of IBM or the IBM Q team. This work has partially been supported by the European Union through the COST Action IC1405 and the Google Research Award Program.

References

[1] IBM Q. https://www.research.ibm.com/ibm-q/.
[2] IBM Q Devices. https://www.research.ibm.com/ibm-q/technology/devices.
[3] IBM Quantum Experience. https://quantumexperience.ng.bluemix.net/qx/editor.
[4] IBM QX backend information. https://github.com/QISKit/ibmqx-backend-information.
[5] JKU Qiskit Addon Simulator. https://github.com/Qiskit/qiskit-jku-provider.
[6] QISKit Developer Challenge. https://qx-awards.mybluemix.net/#qiskitDeveloperChallengeAward.
[7] QISKIT SDK. https://qiskit.org/.
[8] Qiskit Tutorials. https://nbviewer.jupyter.org/github/Qiskit/qiskit-tutorial/blob/master/index.ipynb.
[9] A. J. Abhari, A. Faruque, M. J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, and F. Chong. Scaffold: Quantum programming language. Technical report, 2012.
[10] M. Amy, D. Maslov, M. Mosca, and M. Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(6):818–830, 2013.
[11] A. Botea, A. Kishimoto, and R. Marinescu. On the complexity of quantum circuit compilation. In *Symposium on Combinatorial Search*, 2018.
[12] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.
[13] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: a scalable quantum programming language. In *Conf. on Programming Language Design and Implementation*, pages 333–342, 2013.
[14] T. Häner, D. S. Steiger, M. Smelyanskiy, and M. Troyer. High performance emulation of quantum circuits. In *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, page 74, 2016.
[15] A. Kandala, A. Mezzacapo, K. Temme, M. Takita, M. Brink, J. M. Chow, and J. M. Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242, 2017.
[16] N. Khammassi, I. Ashraf, X. Fu, C. Almudever, and K. Bertels. QX: A high-performance quantum computer simulation platform. In *Design, Automation and Test in Europe*, 2017.
[17] J. Koch, M. Y. Terri, J. Gambetta, A. A. Houck, D. Schuster, J. Majer, A. Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf. Charge-insensitive qubit design derived from the cooper pair box. *Physical Review A*, 76(4):042319, 2007.
[18] G. Li, Y. Ding, and Y. Xie. Tackling the qubit mapping problem for NISQ-era quantum devices. *arXiv preprint arXiv:1809.02573*, 2018.
[19] D. C. McKay, T. Alexander, L. Bello, M. J. Biercuk, L. Bishop, J. Chen, J. M. Chow, A. D. Córcoles, D. Egger, S. Filipp, et al. Qiskit backend specifications for OpenQASM and OpenPulse experiments. *arXiv preprint arXiv:1809.03452*, 2018.
[20] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.
[21] P. Niemann, R. Wille, and R. Drechsler. Efficient synthesis of quantum circuits implementing Clifford group operations. In *Asia and South Pacific Design Automation Conf.*, pages 483–488, 2014.
[22] P. Niemann, R. Wille, and R. Drechsler. Equivalence checking in multi-level quantum systems. In *Int'l Conf. of Reversible Computation*, pages 201–215, 2014.
[23] P. Niemann, R. Wille, and R. Drechsler. Improved synthesis of Clifford+T quantum functionality. *Design, Automation and Test in Europe*, 2018.
[24] J. Preskill. Quantum computing in the NISQ era and beyond. *arXiv preprint arXiv:1801.00862*, 2018.
[25] M. Saeedi, R. Wille, and R. Drechsler. Synthesis of quantum circuits for linear nearest neighbor architectures. *Quantum Information Processing*, 10(3):355–377, 2011.
[26] A. Shafaei, M. Saeedi, and M. Pedram. Optimization of quantum circuits for interaction distance in linear nearest neighbor architectures. In *Design Automation Conf.*, pages 41–46, 2013.
[27] A. Shafaei, M. Saeedi, and M. Pedram. Qubit placement to minimize communication overhead in 2D quantum architectures. In *Asia and South Pacific Design Automation Conf.*, pages 495–500, 2014.
[28] M. Siraichi, V. F. Dos Santos, S. Collange, and F. M. Q. Pereira. Qubit allocation. In *International Symposium on Code Generation and Optimization*, pages 1–12, 2018.
[29] M. Smelyanskiy, N. P. D. Sawaya, and A. Aspuru-Guzik. qHiPSTER: The quantum high performance software testing environment. *CoRR*, abs/1601.07195, 2016.
[30] D. S. Steiger, T. Häner, and M. Troyer. ProjectQ: an open source software framework for quantum computing. *arXiv preprint arXiv:1612.08091*, 2018.
[31] G. F. Viamontes, I. L. Markov, and J. P. Hayes. High-performance QuIDD-based simulation of quantum circuits. In *Design, Automation and Test in Europe*, page 21354. IEEE Computer Society, 2004.
[32] R. Wille, A. Fowler, and Y. Naveh. Computer-aided design for quantum computation. In *Int'l Conf. on CAD*, 2018.
[33] R. Wille, D. Große, D. M. Miller, and R. Drechsler. Equivalence checking of reversible circuits. In *Int'l Symp. on Multi-Valued Logic*, pages 324–330, 2009.
[34] R. Wille, O. Keszöcze, M. Walter, P. Rohrs, A. Chattopadhyay, and R. Drechsler. Look-ahead schemes for nearest neighbor optimization of 1D and 2D quantum circuits. In *Asia and South Pacific Design Automation Conf.*, pages 292–297, 2016.
[35] R. Wille, A. Lye, and R. Drechsler. Exact reordering of circuit lines for nearest neighbor quantum architectures. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 33(12):1818–1831, 2014.
[36] R. Wille, N. Quetschlich, Y. Inoue, N. Yasuda, and S. Minato. Using πDDs for nearest neighbor optimization of quantum circuits. In *Int'l Conf. of Reversible Computation*, pages 181–196, 2016.
[37] A. Zulehner, S. Gasser, and R. Wille. Exact global reordering for nearest neighbor quantum circuits using A^*. In *International Conference on Reversible Computation*, pages 185–201. Springer, 2017.
[38] A. Zulehner, P. Niemann, R. Drechsler, and R. Wille. Accuracy and compactness in decision diagrams for quantum computation. In *Design, Automation and Test in Europe*, 2019.
[39] A. Zulehner, A. Paler, and R. Wille. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2018.
[40] A. Zulehner and R. Wille. Advanced simulation of quantum computations. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2018.
[41] A. Zulehner and R. Wille. One-pass design of reversible circuits: Combining embedding and synthesis for reversible logic. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 37(5):996–1008, 2018.
[42] A. Zulehner and R. Wille. Compiling SU(4) quantum circuits to IBM QX architectures. In *Asia and South Pacific Design Automation Conf.*, 2019.
[43] A. Zulehner and R. Wille. Matrix-vector vs. matrix-matrix multiplication: Potential in DD-based simulation of quantum computations. In *Design, Automation and Test in Europe*, 2019.