A Resource Estimation and Verification Workflow in Q#

Special session paper

Mathias Soeken, Mariia Mykhailova, Vadym Kliuchnikov, Christopher Granade, Alexander Vaschillo
Microsoft Quantum, Redmond, United States

***Abstract*—An important branch in quantum computing involves accurate resource estimation to assess the cost of running a quantum algorithm on future quantum hardware. A comprehensive and self-contained workflow with the quantum program in its center allows programmers to build comprehensible and reproducible resource estimation projects. We show how to systematically create such workflows using the quantum programming language Q#. Our approach uses simulators for verification, debugging, and resource estimation, as well as rewrite steps for optimization.**

I. Introduction

Quantum computing has the potential to efficiently solve some computational problems that are intractable to solve on classical computers [1], [2]. Quantum programming languages [3] allow developers to formalize quantum algorithms in terms of quantum programs that can be executed. While execution on a physical scalable fault-tolerant quantum computer is not yet possible, the execution of quantum programs is already of high interest today, as it allows the developer to verify the implementation and to estimate the cost of running the algorithm, once the quantum hardware is available.

Verification is the task of checking whether the quantum program correctly implements the quantum algorithm. Several methods for verification exists, ranging from full-state simulation (e.g., [4]), automatic functional verification (e.g., [5], [6]), formal verification with program logics (e.g., [7]), or interactive theorem provers (e.g., [8]). Resource estimation is the task of assessing the cost of running a quantum algorithm on a future quantum computer. Assuming specific quality and performance capabilities of the quantum computer, such as clock speed, topological constraints, noise models, and error correction overheads, resource estimation computes the required space in number of physical qubits and required runtime. Recent studies report resource estimates for various quantum computing applications, including quantum chemistry [9], [10], quantum cryptanalysis [11], [12], constraint satisfaction and optimization [13], and quantum simulation [14], [15]. In all these studies, the authors have not applied a systematic workflow to obtain the resource estimates. As a consequence, the presented results are difficult to compare, to reproduce, and to verify. A few frameworks for resource estimation have been proposed in the past. Recently, Oumarou et al. presented QUANTIFY [16], an open-source framework for the quantitative resource analysis of quantum circuits; Suchara et al. presented QuRE [17], a layout estimation tool targeting evaluation on physical quantum technologies.

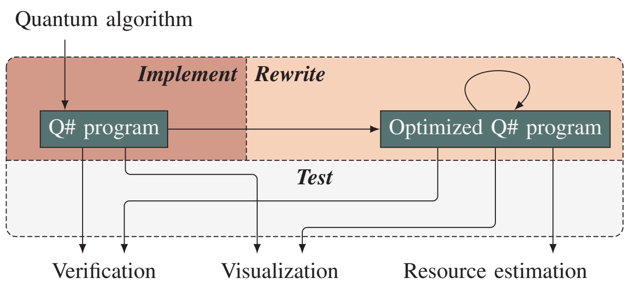

Fig. 1. The proposed resource estimation and verification workflow starts from a quantum algorithm and returns verification, visualization, and resource estimation results. The workflow consists of three stages: in the *Implement* phase, the programmer expresses the algorithm as a Q# program. In the *Rewrite* phase, the compiler can optimize the program using automatic rewrite steps. Finally, the *Test* phase addresses various tasks such as verification, visualization, and resource estimation.

In this work, we show how to systematically integrate quantum resource estimation workflows in Q# [18]. In addition to using the quantum programming language, we achieve this by making use of *simulators* and *rewrite steps*. Simulators are algorithms that execute a Q# program and perform various actions for intrinsic quantum operations. Rewrite steps statically translate a Q# program into a different Q# program by applying rewrite rules on the program's abstract syntax tree.

II. Estimation and Verification Workflow

Fig. 1 illustrates the overall resource estimation and verification workflow. Input to the flow is a quantum algorithm; outputs can be verification results, visualization artifacts, and resource estimates. The workflow is made up of three stages: *implement*, *rewrite*, and *test*. In the first stage, the developer implements the quantum algorithm using Q#, a high-level quantum programming language with built-in support for quantum computing specific constructs such as qubit allocation, intrinsic quantum operations, generation of the adjoint and controlled variants of an operation, or classical control based on measurement results. It is important to note that in this workflow the developer writes a single quantum program, which will serve as "single source of truth" for all subsequent steps.

In the second phase, the Q# compiler performs automatic rewriting steps to optimize the Q# program. This phase addresses two concerns: (i) some optimizations are difficult to detect manually and dedicated optimization algorithms can help to find these; (ii) some optimizations require to break abstractions in the implementation level, for example, the requirement to inline code, so as to unlock further cross-boundary optimization opportunities. The second phase is mostly automatic, although some rewrite steps may require manual configuration.

The third phase executes the resulting Q# programs (the manually written one and the automatically optimized one) using a variety of simulators, depending on the required results. A simulator controls the execution of a program and performs specific actions whenever qubits are allocated and intrinsic operations are executed. Here are several examples of most commonly used simulators and their purposes:

- The `ResourcesEstimator` updates counters for different operations, calculates the depth, and keeps track of the maximum number of allocated qubits.
- The `QuantumSimulator` performs a full-state simulation and can be used to test various properties of the code (for example, show that two operations implement the same unitary), but the number of qubits it allows to allocate is very limited.
- The `ToffoliSimulator` can only be applied to quantum operations that use only classical gates, so that the quantum state can be represented by assigning each qubit a Boolean value. It is useful for testing such operations.

The Microsoft Quantum Development Kit provides API to extend existing simulators or build custom simulators from scratch for various applications. A CHP simulator [19] implementation[1] is one example for such a custom simulator.

The workflow is universal. Eventually, an actual quantum hardware backend can be used instead of a simulator—without changing the initial Q# program.

III. Running Example

Throughout this paper, we use quantum fanout as a running example. Quantum fanout implements the unitary operation

$$F : |\varphi\rangle \otimes |0\rangle^{\otimes k} \mapsto |0\rangle \otimes |\psi_1 \dots \psi_k\rangle, \tag{1}$$

where $\varphi\rangle = \alpha|0\rangle + \beta|1\rangle$ and $|\psi\rangle = |\psi_1 \dots \psi_k\rangle = \alpha|0\rangle^{\otimes k} + \beta|1\rangle^{\otimes k}$. Pham and Svore have shown that F can be implemented in constant depth with CNOT and Hadamard operations, as well as measurements, by using $O(k)$ helper qubits [20]. Fig. 2 illustrates the operation as a quantum circuit diagram for $k = 4$. The left-hand side depicts a straightforward linear-depth implementation that uses only CNOT gates. Note that the first k CNOT gates share the same control line - the source qubit $|\varphi\rangle$. The last CNOT gate brings the source qubit $|\varphi\rangle$ into the $|0\rangle$ state. In the constant-depth optimized implementation on the right-hand side, we use H gates to initialize some of the helper qubits in the state $|+\rangle = H|0\rangle$. This implementation of quantum fanout applies three layers of CNOT gates, independent of the value of k.

[1]github.com/qsharp-community/chp-sim

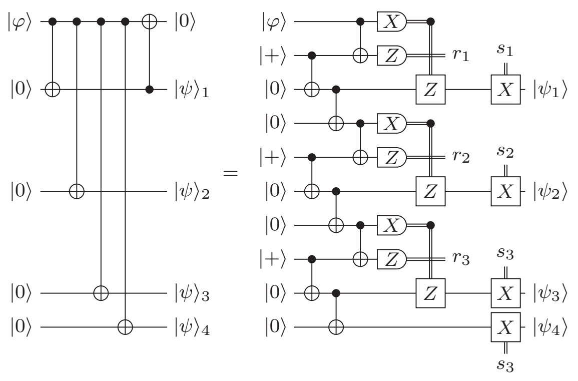

Fig. 2. Quantum circuit representation for quantum fanout with $k = 4$. The left-hand side shows a linear-depth implementation using CNOT gates. The right-hand side shows the constant-depth implementation. Here, $s_i = r_1 \oplus \cdots \oplus r_i$. If $|\varphi\rangle = \binom{\alpha}{\beta}$, we obtain for the resulting qubit register $|\psi\rangle = |\psi_1\psi_2\psi_3\psi_4\rangle = \alpha|0000\rangle + \beta|1111\rangle$.

IV. Implementation Phase

In this section, we describe how to implement the quantum fanout operation in Q#. We will only highlight some noteworthy parts in the implementation, and refer the reader to the QDK documentation[2] for further details. The implementation is as follows:

```
operation ApplyFanout(
    source : Qubit,
    target : Qubit[]) : Unit is Adj {
  body (...) {
    AssertAllZero(target);

    let k = Length(target);
    Fact(k > 1, "There must be at least two copies");

    using (qs = Qubit[2 * k - 3]) {
      let evens = qs[0..2...];
      let odds = qs[1..2...];

      ApplyToEach(H, evens);
      ApplyToEach(CNOT,
          Zipped(evens, Most(target)));
      ApplyToEach(CNOT,
          Zipped(Most(target), odds + [Tail(target)]));
      ApplyToEach(CNOT,
          Zipped([source] + evens, evens));

      let rx = ForEach(MResetX, [source] + odds);
      let rz = Mapped(IsResultOne,
                ForEach(MResetZ, evens));

      let s = Mapped(Fold(Xor, false, _),
               Prefixes(rz) + [rz]);

      ApplyToEach(ApplyIfOne(_, Z, _),
          Zipped(rx, Most(target)));
      ApplyToEach(ApplyIf(X, _, _),
          Zipped(s, target));
    }
    AssertAllZero([source]);
  }

  adjoint (...) {
    AssertAllZero([source]);

    let rx = Mapped(IsResultOne,
              ForEach(MResetX, Rest(target)));
    if (Fold(Xor, false, rx)) {
```

[2]aka.ms/get-started-qdk

```
      Z(Head(target));
    }
45  SWAP(source, Head(target));

    AssertAllZero(target);
  }
}
```

The operation takes as input the qubit `source` in the state $|\varphi\rangle$ and the register `target` in the state $|0\rangle^{\otimes k}$. The algorithm is implemented in the `body`-block (Line 4). The code uses assertions to ensure that the target qubits are indeed in the all-zero state when the operation is called (Line 5). After determining the number of copies k, and asserting that there are at least two copies, $2k-3$ helper qubits are allocated (Line 10). All of them are in state $|0\rangle$. For convenience, we split these helper qubits into the $k-1$ qubits at even indexes (`evens`) and the $k-2$ qubits at odd indexes (`odds`). Next, we initialize the helper qubits at even indexes in the $|+\rangle$ state (Line 14). The use of the library operation `ApplyToEach` helps to emphasize that all initializations take place at the same time. Similarly, the next three lines apply the three layers of CNOT gates. Note that the CNOT operation in Q# takes as input a control qubit as first parameter and a target qubit as second parameter. We can prepare the complete list of control and target pairs by calling `Zipped` on two arrays. The input arrays for each CNOT layer are constructed by combining source qubit, target qubits, and the helper qubits accordingly.

Next, the measurements in X basis and Z basis are performed, using `ForEach` instead of `ApplyToEach`, because we are interested in the return value. The `MResetX` (Line 22) and `MResetZ` (Line 24) operations return a variable of type `Result`, which we convert into Boolean variables in case of the measurements in Z basis, and use as classical controls in case of the measurements in X basis. The `ApplyIfOne` operation takes as input a result value, an operation, and an argument to the operation. By fixing the second parameter to the Z gate and replacing the first and third argument with '`_`', one obtains a partial operation that takes two inputs. This can be used similarly how the CNOT operation was used to apply it to a list of result and qubit pairs (Line 30). The results of Z measurements r_i need to be preprocessed into linear sums $s_i = r_1 \oplus \cdots \oplus r_i$ as illustrated in Fig. 2. We can use array helper functions such as `Mapped`, `Fold`, and `Prefixes` in order to compute them (Line 27). Note the additional element `[rz]`, because the sum s_{k-1} is used twice.

The `adjoint`-block (Line 37) implements the inverse direction of the algorithm (its adjoint). Assertions help to state pre- and post-conditions, which now appear in inverse order: when calling the inverse operation, the source register must be in state $|0\rangle$, and when leaving the operation, the target register must be in state $|0\rangle^{\otimes k}$. Measurement operations (Line 41) and phase correction (Line 43) are used to transform the target register from $\alpha|0\rangle^{\otimes k} + \beta|1\rangle^{\otimes k}$ to $(\alpha|0\rangle + \beta|1\rangle) \otimes |0\rangle^{\otimes k-1}$. In other words, the first qubit of the target register is the expected state for the source qubit. Since the source qubit is in state $|0\rangle$, we can exchange these two using a simple SWAP operation (Line 45).

V. Rewriting phase

The rewrite phase applies a sequence of rewrite steps to the Q# program. These operate on the abstract syntax tree of the program. The Q# compiler contains several rewrite steps targeting performance and resource optimization, including constant propagation, function inlining, and adjoint gate cancellation [21]. An extension API in the compiler allows developers to add custom compilation steps to their projects [22].

VI. Testing phase

In this section, we are describing the testing phase, in which the Q# program is executed using various simulators. Each simulation targets a different purpose. We discuss verification, visualization, and resource estimation as the examples of most common goals of program simulation in the following subsections.

There are several ways to execute Q# programs in the testing phase. The most convenient approach uses Q# or C# testing projects. If the complete testing code can be written in Q#, one can decorate an operation that has no input parameters and a `Unit` return value with a `Test` attribute, marking it as a test; we'll show an example for such tests in this section. C# tests can help to write more elaborate tests, especially useful for sophisticated test data preparation and for post-processing of simulation results.

A. Verification

We can perform various verification tasks as part of resource estimation workflow. An example of functional verification is comparing that two operations act identically for all input states. We use the Choi-Jamiołkowski isomorphism [23], [24] to reduce this equivalence check to one of a qubit state assertion on two entangled registers. We can compute the fanout operation from our running example using only classical gates, as also illustrated by the quantum circuit on the left-hand side in Fig. 2. The following Q# operation `ApplyFanoutClassical` implements this approach:

```
operation ApplyFanoutClassical(source : Qubit,
                               target : Qubit[])
    : Unit is Adj {
  AssertAllZero(target);
  Fact(not IsEmpty(target),
       "There must be at least one copy");

  ApplyToEachA(CNOT(source, _), target);
  CNOT(Head(target), source);
}
```

We then check that the optimized constant-depth quantum fanout implementation and the classical CNOT-based implementation act identically. The following Q# test does this for operation instances that use from 2 to 10 qubits (i.e., produce 1 to 9 copies of the input state).

```
@Test("QuantumSimulator")
operation CheckEquivalence() : Unit {
  let Transform = ApplyWithInputTransformationA(
                    MostAndTail, _, _);

  for (n in 2..10) {
    AssertOperationsEqualReferenced(n,
      Transform(ApplyFanout, _),
      Transform(ApplyFanoutClassical, _));
10  }
}
```

To be generally applicable to a variety of quantum operations, the `AssertOperationsEqualReferenced` operation takes as input two operations which act on a single qubit register, whereas the `ApplyFanout` and `ApplyFanoutClassical` operations expect a pair consisting of a single source qubit and a target register as input. We can use Q# standard library functions and partial application to apply an input transformation to match the required signature (Line 4). This example uses `ApplyWithInputTransformationA`, which takes as input a transformation function that maps values of type `'U` to type `'T`, an operation that expects values of type `'T`, and an a value of type `'U`. The operation `ApplyFanout` expects `(Qubit, Qubit[])` as input argument (type `'T`), which is obtained by applying the function `MostAndTail` to an input type `Qubit[]` (type `'U`), required in the signature for `ApplyOperationsEqualReferenced`.

Equivalence checking tests such as the one described in the previous section make use of full-state quantum simulation, which scales exponentially in the number of qubits, and is therefore limited in its application. We call a quantum operation classical if it is implemented using only classical gates. Quantum oracles are typical examples of classical operations. Note that this does not mean that the operation computes classically, since we can apply it to a quantum state in superposition. However, since quantum operations are linear, simulating classical operations can be done by simulating their behavior on basis states, tracking the Boolean state of each qubit. Q# provides the Toffoli simulator for this purpose, which enables simulation of quantum programs that use many more qubits compared to full-state quantum simulation.

The following operation implements a conditional swap of two n-qubit quantum registers `target1` and `target2`. To reduce depth, it fans out the `control` qubit $n-1$ times, and then uses them to apply controlled SWAP operations in parallel.

```
operation MultiSwap(control : Qubit,
                    target1 : Qubit[],
                    target2 : Qubit[]) : Unit is Adj {
  using (fanout = Qubit[Length(t1) - 1]) {
    within {
      ApplyFanout(control, fanout);
    } apply {
      for ((c, t1, t2) in
           Zipped3(fanout, target1, target2)) {
        Controlled SWAP([c], (t1, t2));
      }
    }
  }
}
```

Since the `ApplyFanout` operation is implemented using non-classical gates, our implementation of `MultiSwap` operation is not classical, although the operation itself is conceptually classical. Replacing `ApplyFanout` with `ApplyFanoutClassical` allows to use the Toffoli simulator, but then we cannot compute accurate resource estimates which exploit the depth-optimization of `ApplyFanout`. The emulation feature [25] in Q# can solve this problem. It allows to override the behavior of specific operations for specific simulators. We can, for example, override the behavior of the `ApplyFanout` operation in the Toffoli simulator, by applying the `ApplyFanoutClassical` operation instead. One way to achieve this is to re-route the operations when creating a `ToffoliSimulator` instance in a C# host program:

```
var sim = new ToffoliSimulator();
sim.Register(typeof(ApplyFanout),
             typeof(ApplyFanoutClassical),
             typeof(IAdjointable));
```

The same technique can be applied, for example, to use quantum-AND gates in the implementation of ripple-carry adders [26].

Several `Dump*` operations provide simulation snapshots when being used in full-state simulation. The `DumpMachine` operation prints all amplitudes of the current quantum state over all active qubits; `DumpRegister` is similar, but can be restricted to a subset of all qubits. The `DumpOperation` operation prints the matrix of a transformation implemented by an operation. The `Dump*` operations can be useful when debugging the code or writing more sophisticated tests.

B. *Visualization*

Quantum circuits help illustrate quantum computation. For example, we have used them in this paper to illustrate quantum fanout for a small example. Note that quantum circuits are not sufficient to visualize arbitrary Q# programs, but they can represent execution traces of Q# programs. Consequently, the simulation architecture can be used to create quantum circuit visualization during the execution of a quantum program, by emitting visualization instructions whenever qubits are allocated and released and whenever intrinsic quantum operations are invoked. A blog post [27] describes such a custom simulator implementation using $\langle$q|pic$\rangle$[3] as visualization language.

Other visualizations are possible, for example, visualization of quantum states (see, e.g., [28]), histograms for individual qubit usages, or layout visualizations when assuming a topological structure for the qubits. Simulators for visualization are particularly useful when working in an interactive development environment such as Jupyter notebooks [29].

C. *Resources estimation*

The Q# resources estimator is a simulator that updates counters for every executed intrinsic operation. It also keeps track of a maximum number of allocated qubits by observing qubit allocation and release. Finally, it uses a tetris-style packing algorithm to compute the circuit depth. It can be applied to a large number of qubits as illustrated in the following example that performs quantum fanout on one million qubits:

```
@Test("ResourcesEstimator")
operation ComputeResources() : Unit {
  using ((source, target) =
         (Qubit(), Qubit[1000000])) {
    ApplyFanout(source, target);
  }
}
```

[3]github.com/qpic/qpic

This test returns the overall resource costs for the code inside the test operation, including the number of CNOT, single-qubit Clifford, T, rotation, and measurement operations, as well as qubit count and depth. Resource constraints can check automatically whether the quantum operation obeys resource upper bounds. As an example, we can assert that implementation of quantum fanout does not use more than $3(k-1)$ CNOT gates, not more than $2(k-1)$ Hadamard gates, and allocates not more than $2k-3$ helper qubits.

```
@Test("ResourcesEstimator")
operation DoesNotExceedCNOTCount() : Unit {
  for (k in 10..30) {
    using ((source, target) = (Qubit(), Qubit[k])) {
      within {
        AllowAtMostNCallsCA(3 * (k - 1), CNOT,
            "Requires too many CNOT operations");
        AllowAtMostNCallsCA(2 * (k - 1), H,
            "Requires too many H operations");
        AllowAtMostNQubits(2 * k - 3,
            "Allocates too many qubits");
      } apply {
        ApplyFanout(source, target);
      }
    }
  }
}
```

Several quantum algorithm research papers have used Q# for concrete resource estimation, particularly in the field of quantum cryptanalysis. We refer the user to [12], [30], [31], [32] for further details and pointers to the Q# code used in their projects.

VII. Further Applications

In the previous sections we covered a typical workflow of developing a quantum library or an application and various tests to validate it. This workflow is used, for example, in the development of the libraries included in the Microsoft Quantum Development Kit.

Another example of an application that heavily relies on unit tests is developing learning materials, which include programming exercises and testing harnesses for them [33]. Learners can use such exercises for self-paced or guided learning.[4] The advantage of providing testing harnesses for programming exercises is that it enables immediate feedback for the learner's solution, which is an important component of an effective learning process. The variety of code properties that can be tested ensures the breadth of the topics that can be covered by such exercises, and the tests can be written to validate *what* the code does rather than *how* it accomplishes it. Consequently, different solutions with the same outcomes will be accepted as correct, as long as they operate within the constraints given in the task. Such constraints can include the same assertions used for resource testing, for example the number of times a certain operation can be called or the number of helper qubits that can be allocated.

Alternatively, such exercises can be used as automatically graded programming assignments in a quantum computing course. In [34], the authors describe an introductory quantum computing course that relies on the Quantum Katas to introduce quantum programming to the students and programming assignments to assess the students' work. In this case the automated testing harnesses allowed the instructors to use the time spent grading the assignments more efficiently compared to grading written assignments of similar complexity manually.

[4]github.com/microsoft/QuantumKatas

VIII. Conclusion

In this paper, we presented a systematic resource estimation and verification workflow for quantum algorithms based on the quantum programming language Q#. The key advantage of the proposed workflow is that a single quantum program is used for all conducted analyses. The program itself acts as reference implementation for the quantum algorithm and is agnostic to the simulators used in the testing. No dedicated code is written that targets resource estimation or verification. Instead the simulators control how a program is executed and apply the right semantics to intrinsic operation calls and qubit allocation. Developers can apply simulator-specific optimizations by means of rewriting steps. This is particularly important for resource estimation, in which clever optimization techniques can reduce the resource cost by trading off abstractions in the original program such as operation call hierarchies and generic compositional library operations.

Our proposed workflow allows to create reproducible resource estimation projects in a single self-contained environment. The workflows can be extended through custom simulators and custom rewriting steps. Several verification methods help to ensure correctness of the implementation, for example, when trying to improve resource counts.

References

[1] R. P. Feynman, "Simulating physics with computers," *Int'l Journal of Theoretical Physics*, vol. 21, no. 6, pp. 467–488, 1982. [Online]. Available: https://doi.org/10.1007/BF02650179

[2] K. M. Svore and M. Troyer, "The quantum future of computation," *IEEE Computer*, vol. 49, no. 9, pp. 21–30, 2016. [Online]. Available: https://doi.org/10.1109/MC.2016.293

[3] B. Heim, M. Soeken, S. Marshall, C. Granade, M. Roetteler, A. Geller, M. Troyer, and K. Svore, "Quantum programming languages," *Nature Reviews Physics*, 2020. [Online]. Available: https://doi.org/10.1038/s42254-020-00245-7

[4] T. Häner and D. S. Steiger, "0.5 petabyte simulation of a 45-qubit quantum circuit," in *Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 33:1–33:10.

[5] M. Amy, "Towards large-scale functional verification of universal quantum circuits," in *Quantum Physics and Logic*, ser. EPTCS, vol. 287, 2018, pp. 1–21. [Online]. Available: https://doi.org/10.4204/EPTCS.287.1

[6] S. Yamashita and I. L. Markov, "Fast equivalence-checking for quantum circuits," in *Int'l Symp. on Nanoscale Architectures*, 2010, pp. 23–28. [Online]. Available: https://doi.org/10.1109/NANOARCH.2010.5510932

[7] M. Ying, "Hoare logic for quantum programs," *arXiv preprint arXiv:0906.4586*, 2009.

[8] R. Rand, J. Paykin, and S. Zdancewic, "QWIRE practice: Formal verification of quantum circuits in Coq," in *Quantum Physics and Logic*, ser. EPTCS, vol. 266, 2017, pp. 119–132. [Online]. Available: https://doi.org/10.4204/EPTCS.266.8

[9] V. von Burg, G. H. Low, T. Häner, D. S. Steiger, M. Reiher, M. Roetteler, and M. Troyer, "Quantum computing enhanced computational catalysis," *arXiv preprint arXiv:2007.14460*, 2020.

[10] M. Reiher, N. Wiebe, K. M. Svore, D. Wecker, and M. Troyer, "Elucidating reaction mechanism on quantum computers," *Proceedings of the National Academy of Sciences*, vol. 114, no. 29, pp. 7555–7560, 2017.

[11] C. Gidney and M. Ekerå, "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits," *arXiv preprint arXiv:1905.09749*, 2019.

[12] T. Häner, S. Jaques, M. Naehrig, M. Roetteler, and M. Soeken, "Improved quantum circuits for elliptic curve discrete logarithms," in *Int'l Workshop on Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, J. Ding and J. Tillich, Eds., vol. 12100. Springer, 2020, pp. 425–444. [Online]. Available: https://doi.org/10.1007/978-3-030-44223-1_23

[13] E. Campbell, A. Khurana, and A. Montanaro, "Applying quantum algorithms to constraint satisfaction problems," *Quantum*, vol. 3, p. 167, 2019, arXiv preprint arXiv:1810.05582v2.

[14] A. M. Childs, D. Maslov, Y. Nam, N. J. Ross, and Y. Su, "Toward the first quantum simulation with quantum speedup," *Proceedings of the National Academy of Sciences*, vol. 115, no. 38, pp. 9456–9461, 2018. [Online]. Available: https://www.pnas.org/content/115/38/9456

[15] R. Babbush, C. Gidney, D. W. Berry, N. Wiebe, J. McClean, A. Paler, A. Fowler, and H. Neven, "Encoding electronic spectra in quantum circuits with linear t complexity," *Physical Review X*, vol. 8, p. 041015, 2018. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.8.041015

[16] O. Oumarou, A. Paler, and R. Basmadjian, "QUANTIFY: a framework for resource analysis and design verification of quantum circuits," *arXiv preprint arXiv:2007.10893*, 2020.

[17] M. Suchara, J. Kubiatowicz, A. I. Faruque, F. T. Chong, C. Lai, and G. Paz, "QuRE: The quantum resource estimator toolbox," in *Int'l Conf. on Computer Design*, 2013, pp. 419–426. [Online]. Available: https://doi.org/10.1109/ICCD.2013.6657074

[18] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler, "Q#: Enabling scalable quantum computing and development with a high-level DSL," in *Real World Domain Specific Languages Workshop*, 2018, pp. 7:1–7:10. [Online]. Available: http://doi.acm.org/10.1145/3183895.3183901

[19] S. Aaronson and D. Gottesman, "Improved simulation of stabilizer circuits," *Physical Review A*, vol. 70, p. 052328, 2004. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.70.052328

[20] P. Pham and K. M. Svore, "A 2D nearest-neighbor quantum architecture for factoring in polylogarithmetic depth," *arXiv preprint arXiv:1207.6655*, 2012.

[21] R. Soiffer, "Q# compiler optimizations," 2019, Q# blog post. [Online]. Available: https://devblogs.microsoft.com/qsharp/q-compiler-optimizations/

[22] B. Heim, "Extending the Q# compiler," 2020, Q# blog post. [Online]. Available: https://devblogs.microsoft.com/qsharp/extending-the-q-compiler/

[23] M.-D. Choi, "Completely positive linear maps on complex matrices," *Linear Algebra and its Applications*, vol. 10, no. 3, pp. 285–290, 1975. [Online]. Available: https://doi.org/10.1016/0024-3795(75)90075-0

[24] A. Jamiołkowski, "Linear transformations which preserve trace and positive semidefiniteness of operators," *REMP*, vol. 3, no. 4, pp. 275–278, 1972. [Online]. Available: https://doi.org/10.1016/0034-4877(72)90011-0

[25] M. Soeken, "Emulation in Q#," 2020. [Online]. Available: https://devblogs.microsoft.com/qsharp/emulation-in-q/

[26] C. Gidney, "Halving the cost of quantum addition," *Quantum*, vol. 2, p. 74, 2018. [Online]. Available: https://doi.org/10.22331/q-2018-06-18-74

[27] M. Soeken, "Build your own Q# simulator – part 3: A circuit-diagram builder with $\langle$q|pic$\rangle$," 2020, Q# blog post. [Online]. Available: https://devblogs.microsoft.com/qsharp/build-your-own-q-simulator-part-3-a-circuit-diagram-builder-with-qpic/

[28] S. Marshall, "Visualizing quantum state with Q#," 2019. [Online]. Available: https://www.sarahmarshall.name/blog/visualizing-quantum-state-with-qsharp.html

[29] M. Brown, C. Granade, B. Heim, R. Koh, R. Larabi, S. Marshall, A. Paz, and R. Shaffer, "Visualizing high-level quantum programs," in *Int'l Workshop on Quantum Computing Software*, 2020.

[30] S. Jaques, M. Naehrig, M. Roetteler, and F. Virdia, "Implementing grover oracles for quantum key search on AES and LowMC," *arXiv preprint arXiv:1910.01700*, 2019.

[31] G. Banegas, D. J. Bernstein, I. van Hoof, and T. Lange, "Concrete quantum cryptanalysis of binary elliptic curves," *IACR Trans. on Cryptographic Hardware and Embedded Systems*, vol. 2021, 2021, cryptology ePrint Archive, Report 2020/1296.

[32] X. Bonnetain and S. Jaques, "Quantum period finding against symmetric primitives in practice," *arXiv preprint arXiv:2011.07022*, 2020.

[33] M. Mykhailova, "Inside the quantum katas, part 1," 2020. [Online]. Available: https://devblogs.microsoft.com/qsharp/inside-the-quantum-katas-part-1/

[34] M. Mykhailova and K. M. Svore, "Teaching quantum computing through a practical software-driven approach: Experience report," in *ACM Technical Symposium on Computer Science Education*, 2020, pp. 1019–1025. [Online]. Available: https://doi.org/10.1145/3328778.3366952