A Survey of Quantum Resource Estimation Tools

Brian Goldsmith [§]
Unitary Fund
Buena Park, CA
brian.goldsmith@gmail.com

Arnav Sharma [§]
Lincoln Laboratory
Massachusetts Institute of Technology, Lexington, MA
Viterbi School of Engineering
University of Southern California, Los Angeles, CA
arnavsha@usc.edu

***Abstract*—Quantum computing offers the potential to solve many currently intractable problems using quantum algorithms. Knowing the resource requirements for important quantum algorithms to run fault-tolerantly is critical for understanding the current hardware capabilities, as well as planning for future hardware advancements. To that end, quantum resource estimation tools attempt to solve this by calculating estimates of those requirements. This paper reviews four versatile and robust resource estimation tools, pyLIQTR, Qualtran, Azure Quantum Resource Estimator, and BenchQ. The tools are assessed by analyzing their code, documentation, and scientific papers used for building the tools. This survey details the techniques used to produce the estimates, as well as highlights the features that differentiate the tools. In particular, the inputs required by the tools and the produced results are reviewed to allow readers to have an understanding of each tool's requirements and capabilities. The survey provides the information and analysis to enable readers to determine which tool would be best for individual use-cases.**

I. Introduction

Quantum computing presents a revolutionary advancement in computation, holding the potential to increase computational efficiency and push the boundaries of tractability. Many novel quantum algorithms have been proposed throughout the years; some well-known examples include Shor's factoring algorithm [25] and Grover's search algorithm [9] which feature exponential and quadratic speed-ups over their classical counterparts, respectively. However, realizing these speed-ups requires advances in quantum hardware. As hardware advances, it is important to know if current hardware can support particular quantum algorithms or how close the hardware is to having the resources to run them. Quantum resource estimation (QRE) addresses these important questions by determining the resources that algorithms require.

When new and exciting quantum algorithms are proposed, authors will often make rigorous mathematical arguments for their time and space complexities in terms of abstract operations and assumed perfect quantum bits (qubits). However, real hardware will not implement the operations directly, but rather construct them using the system's native quantum gates. Furthermore, real qubits are subject to noise that can interfere with computation. This necessitates error correction and mitigation techniques for accurate results, which requires additional resources. QRE tools can take these considerations into account when generating the requirements.

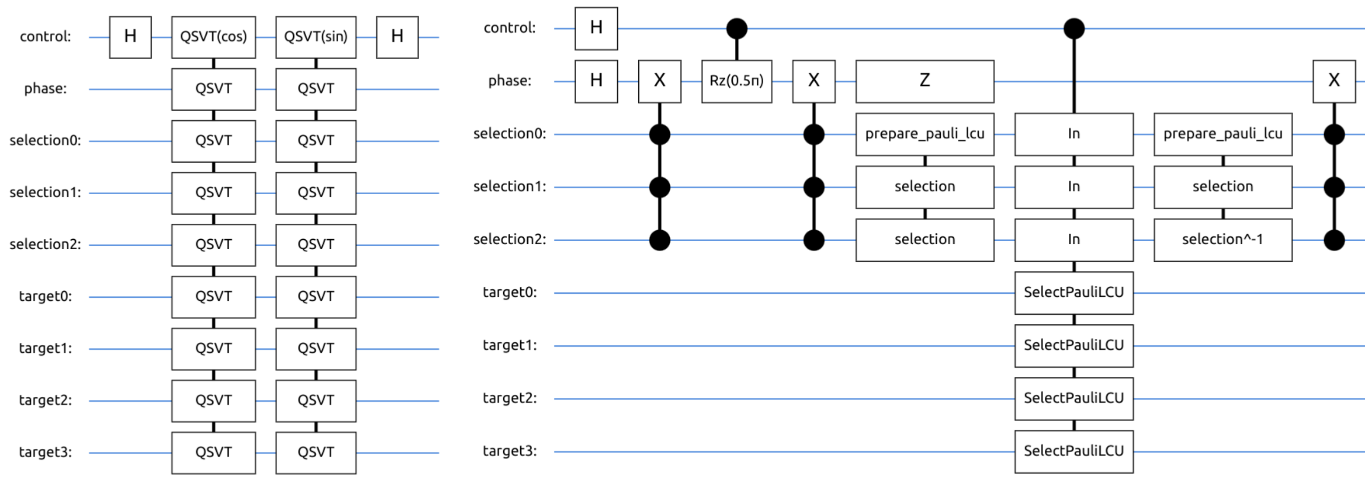
The resource estimates generated by QRE tools give a sense of what must be directly executed on a quantum computer for the quantum operations or algorithms. Typically, a user will define these operations as a sequence of "high level" operators or gates in a quantum circuit which is agnostic with respect to implementation or hardware. The tools decompose this representation into one which consists of a set of elementary quantum gates that are native to a device or class of hardware. Some tools carry the process further by taking into account information about the error correction scheme. These tools estimate how many more qubits, along with the corresponding operations, are needed to implement the "logical" elementary gates on a physical device. In either case, a tool can use this lower level decomposition by counting the number of quantum gates (or particularly resource-heavy quantum gates like T gates), as well as the depth of a quantum circuit executing them. These counts can be used to estimate the basic operations needed to execute the circuit and the resources those operations would need. In this way, QRE tools help characterize what hardware milestones are required for quantum algorithms.

QRE has become an area of increasing interest as researchers try to determine if or when a quantum algorithm can be run on near-term hardware. [5] [23] [24] This has led to the development of several different resource estimation tools. This work reviews a set of QRE tools, comparing and contrasting their capabilities and requirements to gain insights on which particular use cases are best for each tool. Specifically, we chose to analyze the following four tools: pyLIQTR, Qualtran, Azure Quantum Resource Estimation, and BenchQ. While this is not an exhaustive list, it does review the most active and functionally rich QRE tools, serving as a good first step towards guiding users in QRE.

II. Methods

Our initial approach was to compare resource outputs for a number of basic quantum operations among the different tools. To start, we considered single-qubit rotation gates, CNOT gates, and Toffoli gates. However, this quickly revealed the difficulty of comparing the tools quantitatively as pyLIQTR currently focuses on logical estimates, while Qualtran, Azure Quantum Resource Estimator and BenchQ produce logical and

[§]These authors contributed equally.



```
{'LogicalQubits': 9, 'T': 95373, 'Clifford': 197471}
```

Fig. 1: Example outputs of pyLIQTR's implementations/estimations for system evolution under a transverse-field Ising Hamiltonian on a (2 qubit by 2 qubit) square lattice. Left: High-level circuit including four system qubits ("targets") and ancillae used to approximate the evolution via Quantum Singular Value Transform. Right: Twice-decomposed version of this circuit, truncated for space. Bottom: Logical estimates of the Clifford+T gates used in the above circuits.

physical estimates. The physical estimates account for qubit connectivity and error correction while the logical estimates are not concerned with qubit connectivity or fidelity. As a result, we reviewed qualitative metrics in addition to the numerical experiments, to compare and analyze the tools:

- **How Resources Estimates are calculated:** What assumptions are made about hardware and implementation, and what parameters can be changed that will determine the resource estimates?
  - **Inputs:** What format/definition of a given quantum computation is required for each tool?
  - **Outputs:** What specific results do the tools provide?
- **Special Features or Highlights:** What are the unique capabilities for different tools?

In the following sections we provide an overview of each tool, as well as the results of looking at these different metrics. Finally, we compare these results and give recommendations regarding use cases.

III. pyLIQTR

As described on the public GitHub page [21], "pyLIQTR (*LI*ncoln Laboratory *Q*uantum algorithm *T*est and *R*esearch), is a Python library for building quantum circuits derived from quantum algorithms and generating Clifford+T resource estimates." The package currently builds off of Cirq and Qualtran, and has its own implementation of multiple tools, including:

- **Problem Instances**: highly understandable Hamiltonians of interest, relevant to algorithms that they support / the user might seek to implement.
- **Block Encodings**: various encodings of the actions of Hamiltonians.
- **QSP/QSVT**: circuit constructions for Quantum Signal Processing (QSP) and Quantum Singular Value Transforms (QSVT), as well as tools to calculate the **phase angles** necessary for such constructions.
- **Gate/Circuit decomposition**: robust decomposition by combining Cirq functionality with pyLIQTR's own constructions, as well as a rotation gate decomposer based off of [19]. Circuits can be decomposed down to Clifford + T and 2-qubit gates.
- **Resource Analysis**: can provide a Clifford + T cost of any circuit generated in pyLIQTR using the aforementioned decompositions.

A. *How Resource Estimates are Calculated*

1) Inputs: pyLIQTR's resource estimations are calculated from instances of Cirq's quantum circuits.

2) Decompositions: Cirq circuits can be decomposed using the "circuit_decompose_multi" function in pyLIQTR's utilities. In addition to the circuit, this function takes an argument, N, which refers to the number of times the circuit is decomposed. Recall that pyLIQTR has defined its own custom operations pertaining to QSP/QSVT. These operations are constructed from simpler customized operations which after several iterations (i.e. high value of N) will be decomposed. The operations are decomposed into Clifford + T, as well as single qubit rotation gates that are handled at estimation time. For the more elementary gates, functions port over some Clifford + T decompositions from Cirq/Qualtran, including CNOT and Toffoli gates which were tested. The resulting circuit can be printed with standard Cirq operations (i.e. SVGCircuit or a simple print statement), and/or exported to an OpenQASM File. An example of decompositions for a simple QSVT operation is given in Figure 1.

3) Logical Estimates: Currently, pyLIQTR does not support physical resource estimations, but does include logical estimates. These are provided using "estimate_resources" in the resource analysis module. This function takes a circuit (or circuit element) and returns the number of logical qubits used, as well as Clifford and T gates. The function computes this by counting the numbers of qubits, Clifford gates, and T gates, and including additional Clifford+T for each rotation gate. The number of T gates per rotation is sampled from a Gaussian with mean $3.02 * \log \frac{1}{\epsilon} + 0.77$ and standard deviation 2.06 where ϵ is the accuracy of the Clifford + T representation. For the default $\epsilon = 10^{-10}$ this means 101 T gates. Clifford gates per rotation are equal to two times the number of T gates plus one. These formulae for the resources associated with rotation gates are derived empirically based on 1000 decompositions using a method based on [19], which can be found in pyLIQTR's "gate_decomp" modules.

4) Results: These all support the user in both implementing quantum algorithms, as well as resource estimation. It is important to note that pyLIQTR's resource analysis tools are at the *logical level*. Therefore, error correction and hardware properties, like connectivity or native gate set, are not used in the estimates. Rather, the circuits are decomposed to Clifford + T with full connectivity. The package is designed with enough flexibility to have *some* hardware considerations. For example, it is very straightforward to perform decompositions while preserving rotation gates by using Cirq's native tools for counting gates or depths on the decomposed circuit, rather than using pyLIQTR's module which reports Clifford+T estimates for each rotation gate. This limited flexibility for hardware might give a better logical estimate for a NISQ device, but pyLIQTR mostly functions as a high-level tool for resource estimation.

B. Quantum Chemistry

pyLIQTR also contains infrastructure for quantum chemistry algorithms, including common techniques used therein. Particularly, it provides constructions, and thus *deconstructions* for QSP/QSVT operations, all the way down to the Clifford+T level. Quantum Signal Processing and the more generalized Quantum Signal Value Transform are powerful quantum algorithms for calculating polynomial functions of matrices, often Hamiltonians that one may wish to simulate [18]. These techniques are very useful for simulating a number of quantum systems as well as other calculations like Quantum Imaginary Time Evolution [7]. In the spirit of this, pyLIQTR defines its "Problem Instances" as Hamiltonians describing systems of interest, including different molecules and electronic structures. These Hamiltonians are then used (along with the desired function) to determine how the corresponding QSP/QSVT operations are constructed.

IV. QUALTRAN

Qualtran (quantum algorithms translator) is "a set of abstractions for representing quantum programs and a library of quantum algorithms expressed in that language," created by

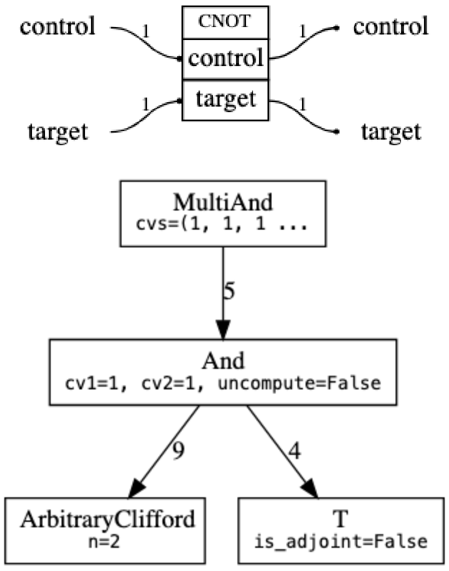


Fig. 2: Examples of Bloq and Call Graph diagrams as given in [1]. Top: Bloq diagram for native CNOT operation Bottom: Call Graph for native MultiAnd (specifically 5-and) Bloq, detailing the decompositions and different counts associated with the constituent Bloqs, all the way down to Clifford + T. The total (logical) count in this case is 45 Cliffords and 20 Ts.

Google [1]. Note that Qualtran is currently in an experimental preview release, so it is likely to grow and change with future releases. In fact, there have been significant updates since this work began, so the authors would like to acknowledge that the information presented in this section could be incomplete or very quickly outdated. Also note that with the tool's release, Google's previous QRE tool, Cirq-FT, has been migrated into Qualtran and deprecated from Cirq [10]. Currently, Qualtran supports both logical and physical resource estimates, and includes their own structure for quantum operations called **Bloqs**. The Bloq is an abstract base class which allows the user to represent high-level quantum programs and subroutines as a hierarchical collection of Python objects. Bloqs can be "wired up" together from other more basic Bloqs, and can be considered analogues to quantum gates which keep track of input and output registers. An example of a CNOT Bloq Diagram is given in Figure 2. Qualtran contains pre-implemented Bloqs for common operations, but the user may also define their own.

A. How Resource Estimates are Calculated

1) Inputs: Qualtran takes different inputs depending on the task at hand. For logical estimates, Qualtran provides Clifford + T estimates using a Bloq as input. For physical estimates, users can utilize either an Azure Cost Model or a CCZ cost model. The former takes the following user-defined inputs:

1) The physical assumptions about the hardware: Latency of Clifford gates and measurements, and physical error

rate.

2) A summary of the circuit/algorithm to execute, including how many algorithm qubits are necessary, numbers of Clifford measurements, T, Toffoli and rotation gates, as well as the rotation depth.
3) Quantum Error Correction Scheme: defined by logical error rate, number of physical qubits needed per logical qubit, number of logical time steps.

For the CCZ cost model, the inputs include number of algorithm qubits, error correction parameters, and a manually-defined magic state factory.

2) Logical Estimates: Logical resource estimates can be calculated from the decomposition of Bloqs. Whether user-defined or imported from Qualtran's pre-built operations, a decomposition of any Bloq can be viewed graphically from the "Call Graph" protocol (an example given in Figure 2). The "bloq_counts" method provide the numerical counts for a Bloq's immediate children in their decomposition. After enough layers of decomposition, this will return Clifford + T counts. These counts, encoded in the Bloq structures, can also serve to provide the algorithm summary information used for the physical estimates. Also note that for the built-in Bloqs, Qualtran ports over decompositions from Cirq, which directly determines the resource estimates. In terms of the elementary gates of interest, this translates to considering rotation gates as one gate (no further decomposition), CNOTs decomposing to a circuit depth of 3 while utilizing 0 T gates, and Toffoli gatess decomposing to a circuit depth of 35 while utilizing 6 T gates.

3) Azure Cost Model: The Azure Cost Model implemented in Qualtran utilizes the user input as described above, but also makes a few assumptions. Namely, the magic state factory is a T state factory, and the model for rotation gates (i.e. the number of T gates needed to approximate a rotation gate to a certain accuracy) is based on the work of [11].

4) Results: Right now, Qualtran serves as a very flexible tool for *logical* QRE, and a promising one for physcial QRE as it updates. As described above, logical estimates ultimately come down to how Bloqs are defined in terms of simpler Bloqs. Thus, a user can get quick and easy resource estimates for custom Bloqs. The visualization tools associated with the Bloq framework are also very informative. Finally, the continual updates to build out physical estimation tools suggest that Qualtran will also have some utility in the realm of physical estimates.

B. Interoperability

Naturally, being made by Google, Qualtran features interoperability with Cirq. Quantum Circuits constructed in Cirq can be transformed into a qualtran Bloq structure, and vice versa (hence Qualtran's usage of some Cirq decompositions). Qualtran also features interoperability with Microsoft's Azure QRE (see section V). Logical counts derived from Bloqs can be fed to the tool to obtain physical estimates, which can provide the user with extra verification for Qualtran's own tools.

V. Azure Quantum Resource Estimator

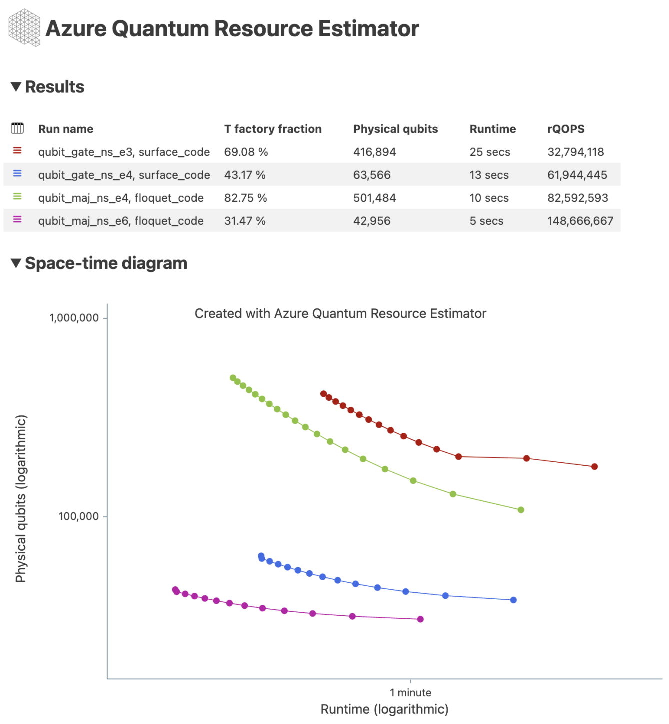
The Microsoft Azure Quantum Resource Estimator (Azure QRE) is an open-source [27] tool that focuses on flexibility to provide useful resource estimations for running algorithms on fault-tolerant quantum computers. This flexibility is seen throughout the process of calculating the resource estimates from the multiple inputs, including Q# and Qiskit, to the results which can include space-time diagrams with multiple estimates or detailed statistical information. In the following sections, a high level overview of the steps taken by Azure QRE are discussed, as well as particular features of note in the tool.

A. How Resource Estimates Are Calculated

The process used by Azure QRE to calculate resource estimates is explained in detail in [27]. What follows is a high level overview of this process, highlighting useful pieces.

1) Inputs: Azure QRE can take input from Q#, Qiskit, Quantum Intermediate Representation (QIR) [17], or known logical estimates.

- **Q#:** There are two methods of using Q# with the Azure QRE. First, a Q# file can be used directly in VS Code with the Azure Quantum Development Kit extension. If the code is in a Q# file, selecting "Q#: Calculate Resource Estimation" from the VS Code Command Palette opens a new VS Code window to display the resource estimation results. A major benefit of this feature is that it does not require any additional code or commands in the Q# program to call the resource estimation. However, the results cannot be programmatically accessed or exported. Also, Q# can be used with the "qsharp" Python package. Using the Python package, it is easy to access the individual statistics from the full resource estimate result that is generated. This allows users to easily compare and export results.
- **Qiskit:** If the algorithm is written using Qiskit, the Azure Quantum service is used with an active Azure account to access the 'microsoft.estimator' Qiskit backend. Azure QRE for Qiskit resides in the online Azure environment and the processing of information happens on the Azure servers. The user does have programmatic access to the results, similar to the Q# with Python described above.
- **QIR:** If either Q# or Qiskit is used, Azure QRE translates the circuit into QIR code. However, existing QIR code can be used with Azure QRE by using the Qiskit options above. This is necessary because to use QIR code directly the Qiskit 'microsoft.estimator' backend is required.
- **Logical Estimates:** When Azure QRE has the input in QIR format (whether from Q#, Qiskit, or QIR directly), it determines some pre-layout information including the number of logical qubits and gate counts for T gates, single qubit rotations, CCZ gates, CCiX gates, and measurements. This information is referred to as the logical estimates. If these logical estimates are known, the estimates can be used as input allowing Azure QRE to skip the QIR related steps.



Azure Quantum Resource Estimator

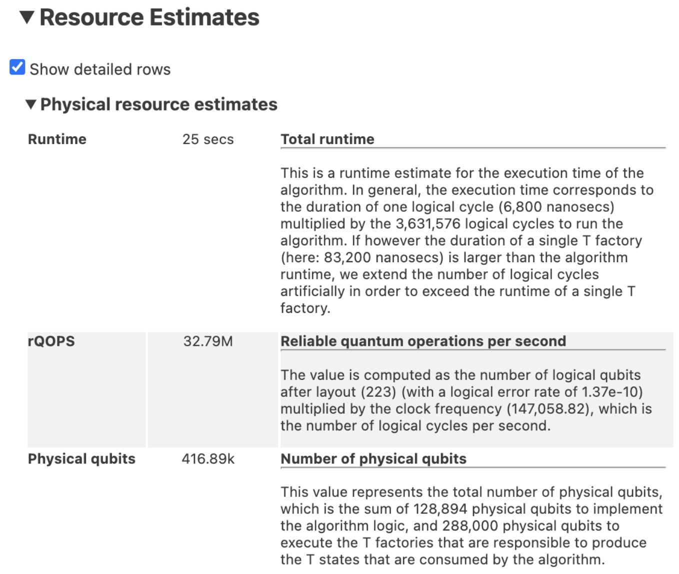
▼ **Results**

	Run name	T factory fraction	Physical qubits	Runtime	rQOPS
🟥	qubit_gate_ns_e3, surface_code	69.08 %	416,894	25 secs	32,794,118
🟦	qubit_gate_ns_e4, surface_code	43.17 %	63,566	13 secs	61,944,445
🟩	qubit_maj_ns_e4, floquet_code	82.75 %	501,484	10 secs	82,592,593
🟪	qubit_maj_ns_e6, floquet_code	31.47 %	42,956	5 secs	148,666,667

▼ **Space-time diagram**

Created with Azure Quantum Resource Estimator
1,000,000
100,000
Physical qubits (logarithmic)
1 minute
Runtime (logarithmic)

Fig. 3: Results of using batch estimations and Pareto frontier estimations in Azure QRE.



▼ **Resource Estimates**

☑ Show detailed rows

▼ **Physical resource estimates**

Runtime	25 secs	**Total runtime** This is a runtime estimate for the execution time of the algorithm. In general, the execution time corresponds to the duration of one logical cycle (6,800 nanosecs) multiplied by the 3,631,576 logical cycles to run the algorithm. If however the duration of a single T factory (here: 83,200 nanosecs) is larger than the algorithm runtime, we extend the number of logical cycles artificially in order to exceed the runtime of a single T factory.
rQOPS	32.79M	**Reliable quantum operations per second** The value is computed as the number of logical qubits after layout (223) (with a logical error rate of 1.37e-10) multiplied by the clock frequency (147,058.82), which is the number of logical cycles per second.
Physical qubits	416.89k	**Number of physical qubits** This value represents the total number of physical qubits, which is the sum of 128,894 physical qubits to implement the algorithm logic, and 288,000 physical qubits to execute the T factories that are responsible to produce the T states that are consumed by the algorithm.

Fig. 4: A small sample of the resource estimate results provided by Azure QRE. It is important to note there are many other sections included in the full result report.

2) Algorithmic Logical Estimates: After the algorithm is broken down into logical estimates, Azure QRE calculates the algorithmic logical estimates which consist of the number of algorithmic logical qubits (adding qubits for all-to-all connectivity [12]), the rotation depth (the number of non-Clifford layers of gates), the algorithmic logical depth (the number logical cycles needed to run the algorithm), and the total number of required T states.

3) Physical Estimates: Once the algorithmic logical estimates have been calculated, the physical estimates are determined by using information about the physical qubit model, error correction scheme, and T factory distillation units. Azure QRE has pre-defined configurations for each of these, but they can be customized, as will be discussed later. When calculating the physical estimates, Azure QRE uses only rotation gates, T gates, CCZ gates, and CCiX gates. The CCZ and CCiX gates use 4 T gates each and the rotation gates use the formula: $0.53\log_2(1/\epsilon) + 4.86$. [11]

4) Results: A noteworthy feature of Azure QRE is the clear, detailed display of results. The best example is the space-time diagram when using batch estimations and Pareto frontier estimations, as seen in Figure 3. In addition to fantastic visual results, the text results are well organized and include descriptions with equations and cited references as seen in Figure 4.

B. Customizable Components

Azure QRE is unique in that it offers flexibility in various areas lower in the quantum stack. Users can customize the following components [16]:

- **Physical Qubit Model**: There are a total of 6 pre-configured physical qubit models, with 4 configured for gate-based instruction and 2 for Majorana instruction. Azure QRE also allows for a completely user-defined model. The parameters for a model include times and error rates for both one and two qubit measurements and gates, as well as T gate and idle error rates.
- **Quantum Error Correction (QEC) Scheme**: The defined QEC schemes include the gate-based surface code from [8] [29], the Majorana surface code from [26] [6], and the Majorana floquet_code from [22]. These QEC schemes can be customized, or a new scheme can be created with a crossing pre-factor, error correction threshold, logical cycle time, and physical qubits per logical qubit.
- **Distillation Units**: Similar to the other options, Azure QRE has predefined T factory distillation units, ”15-1 RM” and ”15-1 space-efficient”, or a custom distillation unit can be used. For the predefined distillation units, 15-1 refers to 15 input T states for 1 output T state. For a customized distillation unit, the following parameters can be set: the number of input T states, number of output T states, failure probability formula, output error rate formula, physical qubit specification, logical qubit specification, and logical qubit specification first round override. The formula parameters can include ”clifford_error_rate” (or ”c”), ”readout_error_rate” (or ”r”), or ”input_error_rate” (or ”z”) along with constants. While both physical qubit specification and logical qubit specification may be provided, at least one should be used in a custom distillation unit.

- **Error Budget**: Azure QRE provides two ways to set the error budget, either by setting a total error budget or by setting the error budgets for implementing logical qubits, producing T states through distillation, and synthesizing rotation gates with arbitrary angles. If the total error budget is set, then the error budget is uniformly distributed to each error category.
- **Constraints**: By using constraints, users are able to provide thresholds to better drive the estimates with a focus on reducing either the number of qubits or the runtime of the algorithm. To emphasize reducing the number of qubits, the ”max_t_factories” and ”max_physical_qubits” parameters should be used. By setting the ”max_t_factories” parameter, the number of T factory copies can be limited in order to reduce the number of physical qubits. By limiting the maximum number of T factory copies, the number of logical cycles, runtime, and overhead is increased. Similarly, with the ”max_physical_qubits” constraint set, the priority is given to the number of qubits and the optimal runtime is determined while conforming to the threshold. To prioritize reducing the runtime of the algorithm, ”logical_depth_factor” and ”max_duration” can be used. By default, the algorithm is optimized to have the smallest initial number of logical cycles, emphasizing speed over the number of qubits. However, the ”logical_depth_factor” constraint can be used to adjust the initial number of logical cycles used by an algorithm. By setting the ”logical_depth_factor” to greater than one, the number of logical cycles is increased, providing more time for the T state factories to generate more T states. Since each factory can produce more T states, fewer factories are needed and therefore fewer qubits are required. The ”max_duration” parameter sets the maximum runtime of the full algorithm, and the optimal number of qubits is then determined with adherence to the time constraint. Due to the trade-off between number of physical qubits and runtime, either the ”max_duration” or the ”max_physical_qubits” can be set, but not both.

C. Estimate Caching and Known Estimates

The flexible nature of Azure QRE makes it well-suited for gathering estimates for large Q# programs. However, to improve performance there are additional features that can be utilized involving caching and known estimates. [15] Large quantum programs may have Q# operations that are called repeatedly, with the resource estimator processing the operations each time. With estimate caching, it is possible to cache the resource estimates for the operation after the initial run, then re-use the cached results in subsequent calls of the operation. Additional flexibility with estimate caching can be achieved by using a variant value which allows for multiple results to be cached and re-used. While estimate caching is unique to Q#, a similar feature of using known estimates is available in both Q#, as the AccountForEstimates operation, and Python, as the LogicalCounts operation. The AccountForEstimates and LogicalCounts operations allow a user to provide the resource estimate information that is automatically consumed by Azure QRE. This is particularly helpful if a component has not been coded yet, but existing resource estimates are available. Also, this feature lets users test estimates from research papers to determine if new components or methods would improve the quantum program.

D. Comparing Resource Estimates

The power of Azure QRE can be seen in the batch estimation and Pareto frontier estimation [14] features, which can be used for generating a report with multiple results for a single quantum program with varying components and trade-offs. Batch estimations can be run with different parameters (qubit model, QEC scheme, error budget, distillation unit, constraints) or in the case of QIR programs, different operation arguments. The main benefit of the batch estimations is that the logical estimates are calculated once for the algorithm and then used with the different configurations. Since calculating the logical estimates is the most time-consuming aspect of the resource estimation process, batch estimation is highly efficient, enabling the generation of multiple resource estimates with minimal additional time compared to calculating a single estimate. In addition to the batch estimations, a Pareto frontier estimation can also be used to show the trade-off between the number of physical qubits and runtime. When the Pareto frontier estimation is used, multiple estimates are run for the algorithm adjusting the level of parallelism in the T factories. These adjustments show that as the number of physical qubits decreases due to fewer T factories, the runtime increases as the wait time for viable T states from the T factories increases.

Azure QRE is an innovative tool offering flexibility and customizations. It would be especially useful in situations requiring large quantum algorithms and comparing varying configurations' effect on the resource estimates of a single quantum algorithm. Azure QRE provides transparent resource estimates with clear visual and statistical results.

VI. BenchQ

BenchQ is an open-source resource estimation tool [2] developed as part of the DARPA Quantum Benchmarking program [3] by Zapata AI. The goal of the tool is to provide more accurate resource estimates by incorporating graph states which can be mapped onto surface codes and optimized. Since utilizing the graph states can be resource intensive, BenchQ offers multiple variants of estimators using graph states, a footprint estimator, and even integration with Azure QRE. Below is a high level overview of how BenchQ calculates the estimates, followed by a more detailed description of the problem ingestion, problem embedding, and graph estimator.

A. How Resource Estimates Are Calculated

While BenchQ does allow for each piece of the resource estimation process to be accessed directly, the most common way to use BenchQ is with a pipeline. The five default pipelines adjust the configurations of the tool to handle the tradeoff between precision and resources required to extract the estimates. The discussion below assumes the use of a pipeline.

1) Inputs: An AlgorithmImplementation object is passed to the estimator and contains a QuantumProgram, an error budget, and the number of shots.

- **QuantumProgram**: The QuantumProgram can be created from an existing Qiskit, Cirq, or Orquestra circuit. Also, BenchQ includes problem ingestion and problem embedding which allows users to easily input a problem instance, like a molecular Hamiltonian, and use pre-built algorithm implementations, like QAOA.
- **Error Budget**: The error budget consists of algorithm failure tolerance, transpilation failure tolerance, and hardware failure tolerance. A total error budget can be passed in and evenly split or weights can be used to adjust the error distribution.

When using a pipeline, it is also necessary to include a hardware_model to give information about the physical qubits.

- **BASIC_ION_TRAP_ARCHITECTURE_MODEL**: This is used for ion trapped qubits with a physical qubit error rate of 1e-4 and a surface code cycle time of 1e-3 seconds.
- **BASIC_SC_ARCHITECTURE_MODEL**: This is used for superconducting qubits with a physical qubit error rate of 1e-3 and a surface code cycle time of 1e-7 seconds.
- **DETAILED_ION_TRAP_ARCHITECTURE_MODEL**: This extends the basic ion trap model in order to provide more detailed hardware estimates, including power consumption, number of ions, and elementary logic unit (ELU) related information.

Pipelines also take an optional decoder model which describes the belief-propagation decoder. The decoder model can be created from a CSV file with details about the power, area, and delay.

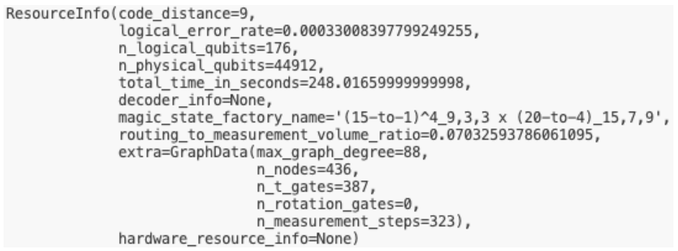
If using an extrapolation pipeline then additional information is required.

- **steps_to_extrapolate_from**: This parameter specifies which steps or sub-circuits to analyze for extrapolation.
- **n_measurement_steps_fit_type** By default, this parameter is set to “logarithmic”, but “linear” is also available. Which option to use will depend on the specific circuit being analyzed.

2) Pipelines: The BenchQ pipelines are very helpful since they require less input from the user while building out the estimator and transformers needed to run the resource estimation.

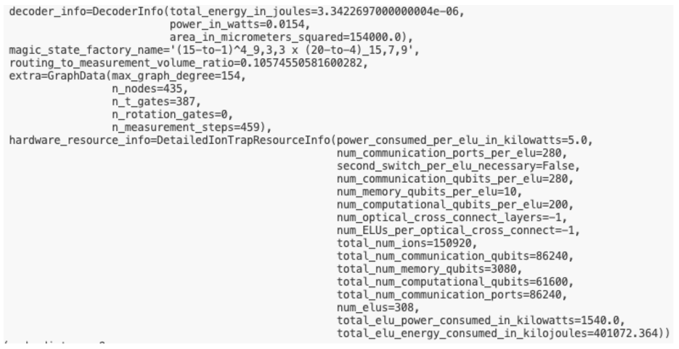
- **get_precise_graph_estimate**: This pipeline takes the given algorithm and reduces the gates to only native gates and uses pyLIQTR to transpile the circuit to Clifford + T gates. After the transpilation, a full graph is generated of the circuit and resource estimates are determined. Since this creates a full graph and uses Clifford + T gates, this is the slowest running, but most accurate resource estimation.
- **get_fast_graph_estimate**: Similar to the pipeline above, this pipeline takes the given algorithm and reduces the gates to native gates only. However, the full circuit is not decomposed to Clifford + T which is why it is faster, but results in higher estimates since the full circuit cannot be optimized.
- **get_precise_extrapolation_estimate**: After the algorithm is reduced to native gates and transpiled to Clifford+T only part of the full graph is analyzed, specified by the steps_to_extrapolate_from parameter. The information from parts of the graph, along with the n_measurement_steps_fit_type property (defaulted to logarithmic), are used to extrapolate estimations for the full graph.
- **get_fast_extrapolation_estimate**: This pipeline is the same as the get_precise_extrapolation_estimate, but it does not transpile the circuit to Clifford+T. Again, this reduces the time it takes to determine the estimates, but it results in higher estimates.
- **get_footprint_estimate**: Unlike the other pipelines, the graph estimator is not used to determine the resource estimates in this pipeline. Instead, after the circuit is reduced to native gates, the total number of T gates is passed along with the number of qubits, hardware model, hardware_failure_tolerance, and decoder model to the footprint estimator. The footprint estimator then uses that information to generate coarse estimates. This pipeline is useful if the graph size for the algorithm is too large to be processed by the other pipelines.

There are some important components that are not pipelines, but are closely related to the pipelines. BenchQ includes a function, automatic_resource_estimator, which takes the same information as a pipeline, and determines which pipeline to use based on the estimated graph size of the pipeline-specific graphs. The following is the order in which pipelines are tested: get_precise_graph_estimate, get_precise_extrapolation_estimate, get_fast_graph_estimate, get_fast_extrapolation_estimate, and get_footprint_estimate. The other component worth noting is the AzureResourceEstimator which is not part of a pipeline, but can be included in a custom resource estimation call where the algorithm implementation, estimator, and transformers are specified. The AzureResourceEstimator uses the “microsoft.estimator” Qiskit



```
ResourceInfo(code_distance=9,
             logical_error_rate=0.00033008397799249255,
             n_logical_qubits=176,
             n_physical_qubits=44912,
             total_time_in_seconds=248.01659999999998,
             decoder_info=None,
             magic_state_factory_name='(15-to-1)^4_9,3,3 x (20-to-4)_15,7,9',
             routing_to_measurement_volume_ratio=0.07032593786061095,
             extra=GraphData(max_graph_degree=88,
                             n_nodes=436,
                             n_t_gates=387,
                             n_rotation_gates=0,
                             n_measurement_steps=323),
             hardware_resource_info=None)
```

Fig. 5: Results from running BenchQ without a decoder and with a BASIC_ION_TRAP_ARCHITECTURE_MODEL.



```
decoder_info=DecoderInfo(total_energy_in_joules=3.3422697000000004e-06,
                         power_in_watts=0.0154,
                         area_in_micrometers_squared=154000.0),
magic_state_factory_name='(15-to-1)^4_9,3,3 x (20-to-4)_15,7,9',
routing_to_measurement_volume_ratio=0.10574550581600282,
extra=GraphData(max_graph_degree=154,
                n_nodes=435,
                n_t_gates=387,
                n_rotation_gates=0,
                n_measurement_steps=459),
hardware_resource_info=DetailedIonTrapResourceInfo(power_consumed_per_elu_in_kilowatts=5.0,
                                                   num_communication_ports_per_elu=280,
                                                   second_switch_per_elu_necessary=False,
                                                   num_communication_qubits_per_elu=280,
                                                   num_memory_qubits_per_elu=10,
                                                   num_computational_qubits_per_elu=200,
                                                   num_optical_cross_connect_layers=-1,
                                                   num_ELUs_per_optical_cross_connect=-1,
                                                   total_num_ions=150920,
                                                   total_num_communication_qubits=86240,
                                                   total_num_memory_qubits=3080,
                                                   total_num_computational_qubits=61600,
                                                   total_num_communication_ports=86240,
                                                   num_elus=308,
                                                   total_elu_power_consumed_in_kilowatts=1540.0,
                                                   total_elu_energy_consumed_in_kilojoules=401072.364))
```

Fig. 6: Partial results from running BenchQ with a decoder model and with the DETAILED_ION_TRAP_ARCHITECTURE_MODEL. The full results include additional information

backend by calling the Azure Quantum service.

3) Results: BenchQ provides statistical results including the code distance, physical and logical qubit numbers, and total time in seconds, along with additional information as seen in Figure 5. If a decoder model or the detailed ion trap hardware model is provided, BenchQ generates more accurate estimates along with specific decoder information and hardware information as seen in Figure 6. The entire circuit, including Clifford gates, is utilized in determining the estimates. When determining the total number of T gates used, rotation gates assume a gridsynth scaling, resulting in a synthesis scaling (SYN_SCALING) of 4 in the following formula for the number of T gates per rotation gate:

$$SYN\_SCALING \times \log_{10}\left(\frac{1}{1-(1-\epsilon)^{1/total\_rots}}\right) \times \frac{1}{log_{10}(2)}.$$

B. Problem Ingestion and Problem Embedding

One of the useful features of BenchQ, is the ability to easily generate Hamiltonians through problem ingestion and then feed that Hamiltonian into existing algorithm implementations with problem embedding. Along with BenchQ's own implementations, other open source projects are seamlessly incorporated in the tool, including PySCF, OpenFermion, and pyLIQTR. These powerful features allow users to generate quantum circuits for complex problems in only a few lines of code.

1) Problem Ingestion: BenchQ broadly refers to problem ingestion as the process of taking information for a problem, and generating the data to be used by the quantum computer to address the problem. In practice, this takes the form of creating Hamiltonians, including:

- **Molecular Hamiltonians** - using PySCF and OpenFermion
- **Plasma Hamiltonians** - using pyLIQTR
- **Ising model** - using pyLIQTR
- **Heisenberg model** - using pyLIQTR
- **Fermi-Hubbard model** - using OpenFermion

2) Problem Embedding: This term is used by BenchQ to describe using the Hamiltonian from problem ingestion and generating a quantum circuit with an existing algorithm implementation including:

- **Quantum Signal Processing (QSP)** - using pyLIQTR
- **Quantum Phase Estimation (QPE)** - using OpenFermion
- **Quantum Approximate Optimization Algorithm (QAOA)**
- **Trotterization**
- **Linear Differential Equation Solver**

C. GraphResourceEstimator

The GraphResourceEstimator is the main feature that makes BenchQ unique as a resource estimator since it gets closer to the physical hardware, giving a more accurate estimate.

1) Graph State Compilation: BenchQ has two different compilers to take a circuit and generate an algorithm-specific graph state. If the circuit has been decomposed to Clifford+T gates, then the Jabalizer compiler [28] can be used. The Ruby Slippers compiler, on the other hand, can handle circuits that have been reduced to Clifford+T gates or not, and generates the graph state using a graph state simulator from [4].

2) Substrate Scheduler: This component takes the graph state and uses the rules from [12] to determine an optimized layout design, logical qubit allocation, and measurement sequence for stabilizer generators as described in [13]. The optimizations include reducing the stabilizer generators, determining the optimal vertex to qubit mapping, and calculating the preferred schedule with the goal of reducing the space-time volume cost from [12].

3) Magic State Factory Selection and Estimate Calculations: After the number of measurement steps are determined from the substrate scheduler, the GraphResourceEstimator attempts to find a magic state factory with a minimal code distance. It does this by finding a code distance where the distilled magic state error rate is less than the "logical cell error rate", which is determined by the surface code being used. If a viable magic state factory is found, then the space time volume is calculated and together the information is used in determining the routing to measurement volume ratio, the logical error rate, the number of physical qubits, and the time in seconds to run the algorithm.

BenchQ is a powerful resource estimation tool which allows for coarse estimation through footprint analysis, as well as more detailed estimation using graph states that are mapped onto surface codes. With the problem ingestion and embedding, users are able to quickly and easily generate circuits to be used by an estimator. Since larger circuits are more resource intensive to run using a graph estimator, BenchQ includes options letting users adjust the balance of time it takes to run the estimation and the accuracy of the estimates. By using graph state compilation, BenchQ is able to provide more accurate estimates, pushing the frontier of resource estimation.

VII. Other Resource Estimation Tools

It should be noted that the four resource estimation tools discussed in this work do not represent an exhaustive list.

However, the four chosen represent active projects that can be used for estimating a broad range of quantum circuits. OpenFermion includes a resource estimation module [20], but it is specifically designed for use with chemical Hamiltonians. Due to this narrow scope, it was not included in this work though it is worth mentioning that the resource estimation module is used by BenchQ in the Quantum Phase Estimation problem embedding.

VIII. Discussion

There are significant differences among the tools and it is helpful to compare and understand those differences. When comparing the tools, the most obvious distinction is if the tool produces only logical estimates or if physical hardware and error correction are used to calculate physical estimates too.

pyLIQTR produces logical estimates, while focusing on specific problem instances that are useful for quantum chemistry and simulation. This does not inherently provide information on resources with respect to hardware properties, like connectivity, native gate set, (opting for Clifford+T), or error correction. As a "first-look" at the cost of an algorithm, this is still very useful in determining whether or not further analysis is warranted, without regard to hardware. If an algorithm is unrealistically costly at the logical level, then it certainly will be after the use of any error correcting code. Until recently, Qualtran too focused on logical estimates, while also enabling the user to design operations as they would like using the Bloq structure. With a recent update, Qualtran has begun making strides towards physical estimates, including interoperability with Azure QRE, showing the potential of these tools to lean on each other for their different strengths.

Azure QRE and BenchQ incorporate physical qubit models and error correction schemes in order to produce additional results including the number of physical qubits and the runtime. Azure QRE is especially useful for very large circuits, since it includes caching and using known estimates. Also, many of the components used by Azure QRE are customizable allowing users to easily generate and compare estimates with different configurations. BenchQ uses graph states to map an algorithm on surface codes, giving a more accurate estimate. Also, BenchQ incorporates work from OpenFermion, pyLIQTR, and PySCF, to make the generation of the algorithm-specific circuit simpler.

When starting this project, the plan was to compare how the resource estimation tools handled a single CNOT, an arbitrary rotation gate, and a Toffoli gate. However, as the project continued, it became clear that the results of such simple circuits would not reveal valuable insights that highlight the differences between the tools. The most obvious issue from the tests was discovered when Azure QRE and BenchQ's footprint estimator would not generate estimates for the single CNOT gate. Azure QRE requires a T gate or measurement and BenchQ's footprint estimator requires at least one T gate or Toffoli gate. Also, Qualtran, Azure QRE, and BenchQ's graph estimator use information about the physical hardware and error correction scheme to generate resource estimates. These additional variables affect the results significantly, to the point that it is difficult to do a direct comparison. Even the comparison of only the logical estimates proved challenging for BenchQ's graph estimator since the logical estimates still incorporate the bus architecture. For example, for the arbitrary single rotation, the other tools had a logical qubit count of 1. However, BenchQ's logical qubit count when using the graph estimator was 2. It should be noted that BenchQ's footprint estimator did give a logical qubit count of 1. A similar result was seen with the single Toffoli gate where the other tools (and BenchQ's footprint estimator) produced 3 logical qubits, but BenchQ's graph estimator gave 8 logical qubits. The results can be seen in Figure 7 and the full outputs, along with the code, can be found at: https://github.com/bdg221/qre

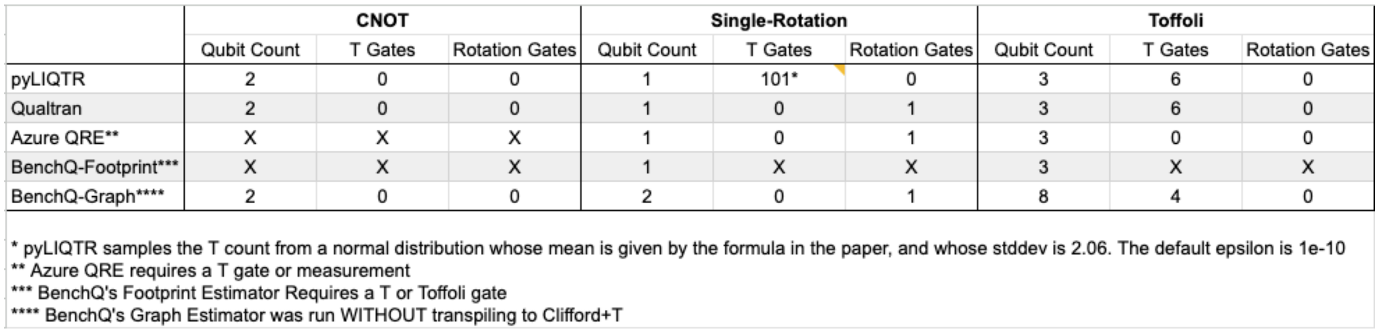
Potential future work on this topic could include configuring the resource estimation tools to use similar physical qubit and error correction parameters, though some parameters are not found in all of the tools and others do not directly align. This would provide a more comparable test between the tools. Also, testing circuits of different sizes would be useful to compare the resources needed by the tools to generate the estimates. This would uncover size or time limits of intractable circuits.

IX. Conclusion

The resource estimation tools discussed utilize different techniques and each has unique strengths.

If the user wishes for a broad look at the (logical) resource requirements for an algorithm employing some abstract operations, then pyLIQTR and Qualtran are both good options. Both will provide the user with representations of quantum circuits (Cirq circuit and Bloq diagram respectively) that perform said operations at the Clifford+T level. However, if one is interested in the particular problem instances included in pyLIQTR, then it would be the better option. Qualtran is preferable if the user insists on defining the decompositions themselves, or prefers the Bloq structure for visualization. If the priority is to gather multiple estimates for different qubit models and error correction schemes, Azure QRE is the best choice. It generates multiple estimates for different qubit models and error correction schemes for easy analysis, especially if using Q#. Qualtran's interoperability with Azure QRE also creates the opportunity to utilize these capabilities while also having access to Bloq structures/visualizations. BenchQ provides a more accurate estimate by optimizing the mapping of the circuit onto the surface codes. However, this additional process can be resource intensive and larger circuits may be intractable.

An important consideration is the question of how good are the estimates given by these tools. The perhaps unsatisfying answer is that they are the best available at this time. It is difficult to imagine the exact details of realized fault tolerant quantum computing, but most agree that many, if not all, aspects of quantum computing from the algorithm implementations to hardware and error correction will be significantly different from what we have today. As we continue to progress towards realizing fault tolerance, the resource estimation tools will need to progress as well, incorporating new formulas



	CNOT			Single-Rotation			Toffoli		
	Qubit Count	T Gates	Rotation Gates	Qubit Count	T Gates	Rotation Gates	Qubit Count	T Gates	Rotation Gates
pyLIQTR	2	0	0	1	101*	0	3	6	0
Qualtran	2	0	0	1	0	1	3	6	0
Azure QRE**	X	X	X	1	0	1	3	0	0
BenchQ-Footprint***	X	X	X	1	X	X	3	X	X
BenchQ-Graph****	2	0	0	2	0	1	8	4	0

* pyLIQTR samples the T count from a normal distribution whose mean is given by the formula in the paper, and whose stddev is 2.06. The default epsilon is 1e-10
** Azure QRE requires a T gate or measurement
*** BenchQ's Footprint Estimator Requires a T or Toffoli gate
**** BenchQ's Graph Estimator was run WITHOUT transpiling to Clifford+T

Fig. 7: The table includes the qubit count, number of T gates, and number of rotation gates resulting from the resource estimation tools execution of a single CNOT gate, arbitrary rotation gate, and Toffoli gate.

and techniques. This continued progression is evident by the significant changes to each of the four tools reviewed in the paper between December 2023 and March 2024. For now, it is important to review how the tools generate the estimates with the understanding that the estimates will continue to get more accurate as the field moves closer to realizing fault tolerant quantum computing.

Acknowledgment

The authors would like to thank Mariia Mykhailova from Microsoft for her mentorship and expertise. Additionally, we would like to acknowledge Kevin Obenland from Lincoln Laboratory, and Athena Caesura and Peter Johnson from Zapata AI, who provided valuable knowledge and assistance. This paper is the result of the Quantum Open Source Foundation (QOSF) mentorship program and we would like to express our gratitude to the foundation.

References

[1] Qualtran documentation — qualtran.readthedocs.io. https://qualtran.readthedocs.io/en/latest/index.html, 2023. [Accessed 26-03-2024].

[2] Zapata AI. Benchq. Available at https://github.com/zapatacomputing/benchq, March 2024. original-date: 2023-02-16.

[3] Joe Altepeter. Quantum Benchmarking. Available at url = https://www.darpa.mil/program/quantum-benchmarking.

[4] Simon Anders and Hans J. Briegel. Fast simulation of stabilizer circuits using a graph state representation. *Physical Review A*, 73(2):022334, February 2006. arXiv:quant-ph/0504117.

[5] Michael E. Beverland, Prakash Murali, Matthias Troyer, Krysta M. Svore, Torsten Hoefler, Vadym Kliuchnikov, Guang Hao Low, Mathias Soeken, Aarthi Sundaram, and Alexander Vaschillo. Assessing requirements to scale to practical quantum advantage, November 2022. arXiv:2211.07629 [quant-ph].

[6] Rui Chao, Michael E. Beverland, Nicolas Delfosse, and Jeongwan Haah. Optimization of the surface code design for Majorana-based qubits. *Quantum*, 4:352, October 2020. arXiv:2007.00307 [quant-ph].

[7] Luuk Coopmans, Yuta Kikuchi, and Marcello Benedetti. Predicting gibbs-state expectation values with pure thermal shadows. *PRX Quantum*, 4:010305, Jan 2023.

[8] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3):032324, September 2012. arXiv:1208.0928 [quant-ph].

[9] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[10] Tanuj Khattar. Cirq-ft: documentation improvements · issue #6141 · quantumlib/cirq — github.com. https://github.com/quantumlib/Cirq/issues/6141, 2024. [Accessed 26-03-2024].

[11] Vadym Kliuchnikov, Kristin Lauter, Romy Minko, Adam Paetznick, and Christophe Petit. Shorter quantum circuits via single-qubit gate approximation. *Quantum*, 7:1208, December 2023.

[12] Daniel Litinski. A Game of Surface Codes: Large-Scale Quantum Computing with Lattice Surgery. *Quantum*, 3:128, March 2019. Publisher: Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften.

[13] Sitong Liu, Naphan Benchasattabuse, Darcy QC Morgan, Michal Hajdušek, Simon J. Devitt, and Rodney Van Meter. A Substrate Scheduler for Compiling Arbitrary Fault-Tolerant Graph States, September 2023. arXiv:2306.03758 [quant-ph].

[14] Sonia Lopez. Batching with the Resource Estimator - Azure Quantum. Available at https://learn.microsoft.com/en-us/azure/quantum/resource-estimator-batching, February 2024.

[15] Sonia Lopez. Optimize Large Programs with the Resource Estimator - Azure Quantum. Available at https://learn.microsoft.com/en-us/azure/quantum/resource-estimator-handle-large-programs, February 2024.

[16] Sonia Lopez. Resource Estimator Target Parameters - Azure Quantum. Available at https://learn.microsoft.com/en-us/azure/quantum/overview-resources-estimator, January 2024.

[17] Sonia Lopez. Tutorial: Resource Estimation with QIR - Azure Quantum. Available at https://learn.microsoft.com/en-us/azure/quantum/tutorial-resource-estimator-qir, January 2024.

[18] Guang Hao Low and Isaac L. Chuang. Optimal hamiltonian simulation by quantum signal processing. *Physical Review Letters*, 118(1), January 2017.

[19] Ken Matsumoto and Kazuyuki Amano. Representation of quantum circuits with clifford and $\pi/8$ gates, 2008.

[20] Jarrod R. McClean, Nicholas C. Rubin, Kevin J. Sung, Ian D. Kivlichan, Xavier Bonet-Monroig, Yudong Cao, Chengyu Dai, E. Schuyler Fried, Craig Gidney, Brendan Gimby, Pranav Gokhale, Thomas Häner, Tarini Hardikar, Vojtěch Havlíček, Oscar Higgott, Cupjin Huang, Josh Izaac, Zhang Jiang, Xinle Liu, Sam McArdle, Matthew Neeley, Thomas O'Brien, Bryan O'Gorman, Isil Ozfidan, Maxwell D. Radin, Jhonathan Romero, Nicolas P. D. Sawaya, Bruno Senjean, Kanav Setia, Sukin Sim, Damian S. Steiger, Mark Steudtner, Qiming Sun, Wei Sun, Daochen Wang, Fang Zhang, and Ryan Babbush. OpenFermion: the electronic structure package for quantum computers. *Quantum Science and Technology*, 5(3):034014, June 2020. Publisher: IOP Publishing.

[21] Kevin Obenland, Justin Elenewski, Kaitlyn Morrell, Rylee Stuart Neumann, Arthur Kurlej, Robert Rood, John Blue, Joe Belarge, and Parker Kuklinski. pyliqtr: Release 1.1.1. https://github.com/isi-usc-edu/pyLIQTR, 2024.

[22] Adam Paetznick, Christina Knapp, Nicolas Delfosse, Bela Bauer, Jeongwan Haah, Matthew B. Hastings, and Marcus P. da Silva. Performance of planar Floquet codes with Majorana-based qubits. *PRX Quantum*, 4(1):010310, January 2023. arXiv:2202.11829 [quant-ph].

[23] Nils Quetschlich, Mathias Soeken, Prakash Murali, and Robert Wille. Utilizing Resource Estimation for the Development of Quantum Computing Applications. http://arxiv.org/abs/2402.12434, February 2024. arXiv:2402.12434 [quant-ph].

[24] Travis L. Scholten, Carl J. Williams, Dustin Moody, Michele Mosca, William Hurley, William J. Zeng, Matthias Troyer, and Jay M. Gambetta. Assessing the Benefits and Risks of Quantum Computers. https://arxiv.org/abs/2401.16317, 2024. Version Number: 2.

[25] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[26] Alan Tran, Alex Bocharov, Bela Bauer, and Parsa Bonderson. Optimizing Clifford gate generation for measurement-only topological quantum computation with Majorana zero modes. *SciPost Physics*, 8(6):091, June 2020. arXiv:1909.03002 [quant-ph].

[27] Wim van Dam, Mariia Mykhailova, and Mathias Soeken. Using Azure Quantum Resource Estimator for Assessing Performance of Fault Tolerant Quantum Computation. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 1414–1419, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Madhav Krishnan Vijayan, Alexandru Paler, Jason Gavriel, Casey R. Myers, Peter P. Rohde, and Simon J. Devitt. Compilation of algorithm-specific graph states for quantum circuits, December 2022. arXiv:2209.07345 [quant-ph].

[29] David S. Wang, Austin G. Fowler, and Lloyd C. L. Hollenberg. Quantum computing with nearest neighbor interactions and error rates over 1%. *Physical Review A*, 83(2):020302, February 2011. arXiv:1009.3686 [quant-ph].