nature computational science

Resource

https://doi.org/10.1038/s43588-025-00792-y

Benchmarking the performance of quantum computing software for quantum circuit creation, manipulation and compilation

Received: 22 September 2024

Accepted: 17 March 2025

Published online: 18 April 2025

Check for updates

Paul D. Nation[1] ✉, Abdullah Ash Saki[1], Sebastian Brandhofer[2], Luciano Bello[3], Shelly Garion[4], Matthew Treinish[1] & Ali Javadi-Abhari[1]

We present Benchpress, a benchmarking suite for evaluating the performance and range of functionality of multiple quantum computing software development kits. This suite consists of a collection of over 1,000 tests measuring key performance metrics for a wide variety of operations on quantum circuits composed of up to 930 qubits and $\mathcal{O}(10^6)$ two-qubit gates, as well as an execution framework for running the tests over multiple quantum software packages in a unified manner. Here we give a detailed overview of the benchmark suite and its methodology and generate representative results over seven different quantum software packages. The flexibility of the Benchpress framework enables benchmarking that not only keeps pace with quantum hardware improvements but also can preemptively gauge the quantum circuit processing costs of future device architectures.

The promise of quantum computation lies in its ability to perform specific tasks more efficiently than otherwise possible using classical methods alone. However, quantum computers do not operate in isolation and require classical computing resources for data pre- and postprocessing. As quantum computers continue to grow in size, it is imperative that the associated classical computing costs be evaluated for scalability and the construction, manipulation and optimization of quantum circuits play an outsized role in terms of classical overhead. Moreover, the performance of quantum computation software for output circuit quality, runtime and memory consumption is critical to successful adoption of this technology. There is a variety of quantum software development kits (SDKs) that perform all or some fraction of quantum circuit pre-execution processing, such as Braket[1], BQSKit[2], Cirq[3], Qiskit[4], Qiskit Transpiler Service (QTS) extension[5], Staq[6] and Tket[7], among others, focusing primarily on quantum circuit construction, manipulation and optimization. The need to evaluate the performance of these software stacks has led to the creation of several collections of assessment circuits[8–10] and Hamiltonians[11], as well as some examples of how to execute collections of circuits in a benchmarking framework[12,13].

However, these earlier works have several limitations. First, collections of quantum circuits are usually stored in OpenQASM[14] compatible format and, thus, do not consider the performance of circuit synthesis[15] in the transpilation process. Doing so requires abstract circuits that must be written directly in the language of the SDK, making testing across several SDKs difficult. In addition, previous testing frameworks are not made to accommodate the testing of circuits at large qubit counts and runtimes, nor are they flexible enough to allow testing over collections of two-qubit (2Q) entangling gate topologies (coupling maps), quickly adding additional SDKs or storing diverse sets of output metrics.

So far, there has been no systematic study of the relative performance of mainstream quantum computing SDKs over extensive collections of quantum circuits at scale, nor has there been a flexible method by which these comparisons can be easily generated. Such a study is critical as quantum devices and experiments push toward 100 qubits or more[16–35], and the differences in software performance and scaling become pronounced. An open-source test suite that not only evaluates multiple SDKs at these scales but also provides an execution framework that yields uniform testing over quantum software with

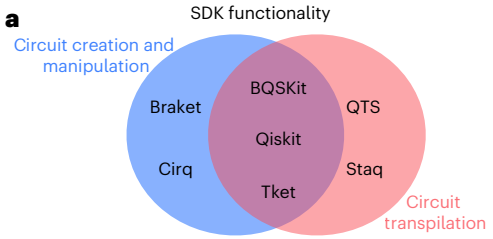
[1]IBM Quantum, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA. [2]IBM Quantum, IBM Germany Research and Development, Böblingen, Germany. [3]IBM Quantum, IBM Research Europe, Zurich, Switzerland. [4]IBM Quantum, IBM Research Israel, Haifa, Israel. ✉e-mail: paul.nation@ibm.com

https://doi.org/10.1038/s43588-025-00792-y

Table 1 | SDK and corresponding version numbers used in generating the reported sample results

SDK	Version number
Amazon-braket-sdk (Braket)	1.86.1
BQSKit	1.1.2
Cirq	1.4.1
Pytket (Tket)	1.31.0
Qiskit	1.2.0
QTS	0.4.8
Staq	3.5

Each version represents the latest release with pertinent functionality as of 30 August 2024. Tket version 1.31.1, which was released while testing was in progress, includes only cosmetic changes to documentation. In addition, after testing was completed, the qiskit_transpiler_service changed names to qiskit_ibm_transpiler.



b

	PASSED	SKIPPED	FAILED	XFAIL
BQSKit	841	22	201	2
Braket	7	1,057	2	0
Cirq	10	1,054	2	0
Qiskit	1,044	22	0	0
QTS	1,013	34	19	0
Staq	549	515	2	0
Tket	957	22	87	0

Fig. 1 | SDK functionality and corresponding test results. a, Venn diagram of SDK functionality. **b**, Status of all 1,066 tests for each SDK based on the definitions given in the Methods. Twenty-two tests are universally skipped due to insufficient qubit count for the target used in device transpilation (see the Methods for details).

disparate feature sets and capabilities would help researchers, developers and end users alike to ascertain the relative value of quantum computing software packages when targeting current and future quantum computing devices. With fault-tolerant computation out of reach in the near term, this initial version of Benchpress is focused on test cases and metrics compatible with execution on noisy quantum processors.

To address these needs, we have developed Benchpress, an open-source collection of tests explicitly designed to measure the performance of quantum computing software for quantum circuit creation and transformation. Benchpress stands out for its common framework that allows testing across three key areas: quantum circuit construction, manipulation and optimization. Although the intersection of functionality varies dramatically across the quantum computing software landscape, Benchpress utilizes notional collections of tests called workouts that allow the full test suite to be executed across any quantum SDK with tests defaulting to being skipped if they are not explicitly overwritten with an SDK-specific implementation. Benchpress is thus able to quantify not only relative performance metrics among SDKs but also the breadth of functionality in a given software package.

To supplement functionality missing in other SDKs, Benchpress uses Qiskit[4] throughout its infrastructure, particularly its compatibility with other SDKs and OpenQASM import and export capabilities. In addition, Qiskit allows the generation of reference implementations for constructs such as abstract backend coupling maps that can, with minimal effort, be consumable by the other SDKs. This yields a uniform, less error-prone testing environment that simplifies benchmarking. Moreover, in making Benchpress open-source we endeavor to create an open and transparent platform by which progress in quantum computing software can be faithfully evaluated.

In this Article, we present results running the initial version of Benchpress, evaluating 1,066 tests for each of seven different quantum SDKs considered. We give a detailed analysis of these results, present the test selection criteria and highlight the key takeaways from the findings.

Results

Results are presented covering the seven SDKs listed in Table 1. Other packages, such as Nvidia CUDA-Q[36], are not included in this study as their feature sets at the time of testing were insufficient to accommodate the test cases. Benchpress is designed to be modular, and other SDKs beyond those considered here can be added straightforwardly. Unlike other SDKs, the QTS augments the transpilation tools of Qiskit with reinforcement learning methods for Clifford synthesis and qubit routing and, thus, is better viewed as an extension to Qiskit rather than a competing package.

The breadth of functionality available in the tested SDKs varies widely and must be accounted for when benchmarking. In general, current quantum software packages can be categorized as having the ability to create and manipulate circuits and/or offering predefined transpilation toolchains that map quantum circuits to quantum hardware systems. This Venn diagram of functionality is shown in Fig. 1.

In what follows, we break the test suite into two components: grouping circuit creation and manipulation and evaluating circuit transpilation tests as a whole. All results presented here are generated using an AMD 7900 processor with 128 GB memory running Linux Mint 21.3. SDKs were run on Python 3.12 with the version numbers presented in Table 1. Complete system information is recorded in the JavaScript Object Notation (JSON) files output by Benchpress and is openly available (see 'Data availability').

Circuit construction and manipulation

Results for the circuit creation and manipulation tests, as defined in the Methods, are shown in Fig. 2. While there is no clear winner when looking at each test in isolation, the aggregate timing information and number of failed or skipped tests tell a different story. Qiskit is the only SDK that passes all of the circuit construction tests, and does so in a time of 2.0 s. The next closest competitor is Tket, which completes all but one test in 14.2 s. BQSKit fails two tests and clocks the slowest total completion time over successful tests at 50.9 s. Results worth highlighting include Cirq's performance at constructing a set of Hamiltonian simulation circuits (test_DTC100_set_build) in a time 55× faster than the nearest competitor Qiskit. Likewise, Qiskit outperforms the other SDKs in the parameter binding test (test_param_circSU2_100_bind), recording a time 13.5× faster than the next closest SDK.

The most notable feature in Fig. 2 is the number of skipped, fails or expected-fail tests. Here, we address those issues. Starting at the top of Fig. 2, the first XFAIL test is for BQSKit on multicontrolled circuit building. BQSKit heavily uses dense numerical linear algebra throughout its compilation pipeline, and the 16-qubit multicontrolled X-gate used in the test took more memory than the test machine had. A similar reason is behind the other XFAIL for BQSKit in the multicontrolled decomposition manipulation test (test_multi_control_decompose). Next, Braket failed the QV OpenQASM import test because there is no native support for the qelib1.inc file that is a standard include file in OpenQASM 2, for example, both the Feynman[8] and QasmBench[9] collections of circuits

https://doi.org/10.1038/s43588-025-00792-y

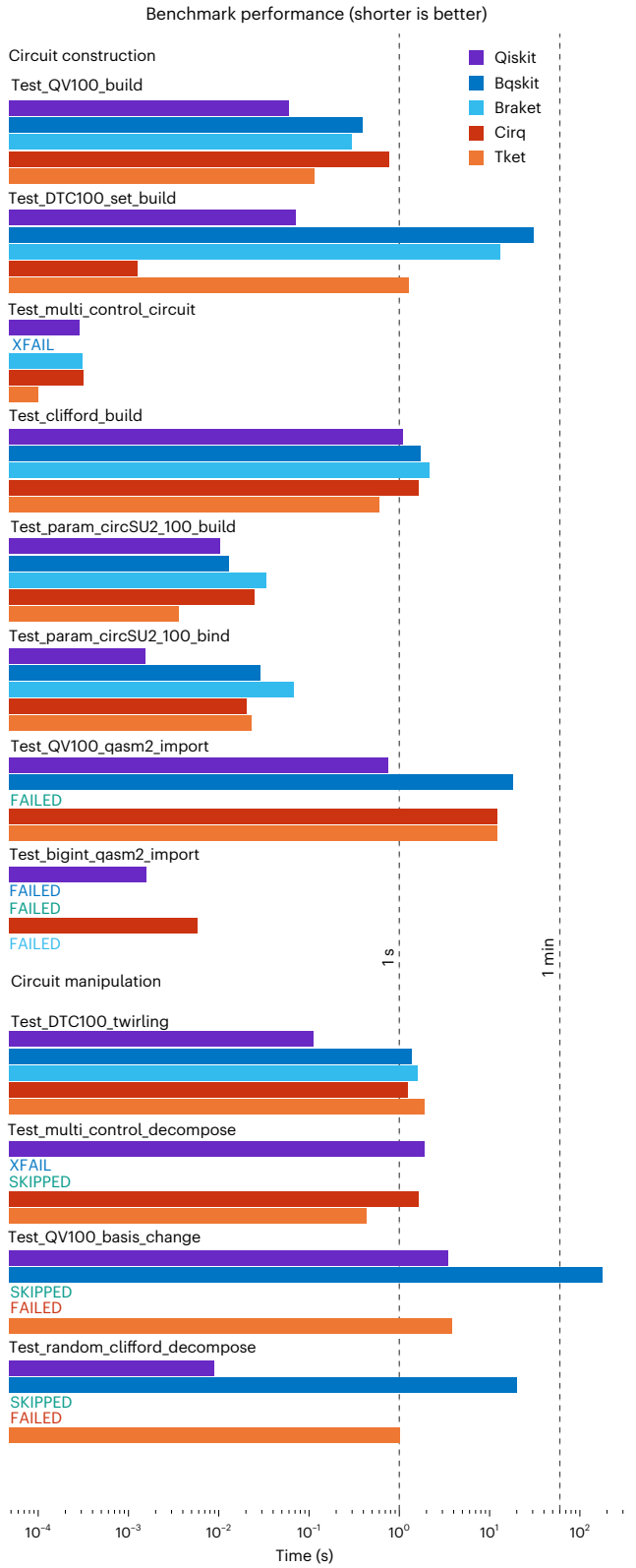


Fig. 2 | Benchmark results for the circuit construction and manipulation portions of Benchpress. Benchpress was used to measure the performance of five SDKs for quantum circuit creation and transformation. Shorter times are better. Tests that are SKIPPED or marked as FAILED or XFAIL are labeled accordingly.

use this include file. Finally, three out of five SDKs could not load an OpenQASM file with an integer larger than 64 bits. Here, BQSKit failed because of the lack of dynamic circuit support, Braket failed in the same manner as the previous QASM test, and the C++ JSON library used in Tket lacks support for these 'BigInt' numbers.

For circuit manipulation, both Qiskit and Tket complete all tests, with Qiskit completing all four tests in 5.5 s versus 7.1 s for Tket. The large number of skipped tests for Braket is due to the lack of basis transformation capabilities. However, add-on packages such as the Qiskit Braket Provider[37] allow this to be done in other SDKs and then passed back to Braket. The two failed tests for Cirq are due to using the basis set of gates: RZ, X, $\sqrt{X}$ and CZ, which, despite being supported gates, hits a recursion limit. In addition to timing information, the multicontrolled decomposition test allows one to examine the quality of the synthesis algorithms in each SDK via the number of 2Q gates in the output circuit. Here, Tket produced the circuit with the fewest 2Q gates, 4,457, versus 7,349 and 17,414 for Qiskit and Cirq, respectively. The relative gate counts and depth for the circuit manipulation tests are shown in the Supplementary Information.

Device and abstract transpilation

Figure 3 shows the results for the five SDKs that support transpilation (Fig. 1) over the combined set of device and abstract tests defined in the Methods. Out of 1,054 total tests, 22 are device transpilation tests larger than the target Heron device and are SKIPPED regardless of the SDK. In addition, the Staq compiler takes OpenQASM files as input and, thus, is unable to execute the Hamiltonian simulation tests, defaulting to those tests being skipped. Only passing tests are shown in Fig. 3. As Qiskit is the only SDK that passes the entire set of transpilation tests, we use those values as a baseline when presenting results. Here, we focus on the statistics of ensembles of circuits as a whole, as opposed to looking at individual tests cases. A lower-level analysis can be done using the published results (see 'Data availability'). A breakdown of the test results per open-source test library is presented in the Supplementary Information.

Figure 3 breaks the results apart by target topology for each of the three reported metrics. The dashed diagonal line represents the Qiskit baseline above which any data points indicate that the specified SDK test result is worse than the corresponding Qiskit value. The markers below this line highlight an SDK performing better than Qiskit. Table 2 numerically quantifies these results, tabulating the statistical values of results across each SDK and topology.

From Fig. 3 and Table 2, we see a few trends emerge. First, we see that, as a whole, Qiskit outperforms BQSKit, Staq and Tket in terms of 2Q gate count, a trend clearly seen in the BQSKit and Tket data. This is particularly true when targeting topologies with limited connectivity. By contrast, while the 2Q gate depth is worse than Qiskit for BQSKit and Staq, Tket outperforms Qiskit overall, with prominent gains for more connected topologies. In particular, there is a collection of Hamiltonian simulation tests on which Tket yields substantial 2Q depth reduction relative to Qiskit. This set is the largest for 'all-to-all' coupling topologies, suggesting that this improvement comes from the synthesis steps performed by Tket. This is corroborated by the synthesis results in 'Circuit construction and manipulation'. Data for other topologies indicate that this improved synthesis becomes less critical as the connectivity decreases and circuit routing becomes dominant. Note that, for transpilation pipelines that include stochastic components, such as the Sabre-based routing routine in Qiskit, both the number and depth of 2Q gates can vary across different transpilations of the same circuit. When looking at a single test case, this randomness can have a large impact on relative performance numbers. However, when looking across large collections of tests, as done here, the fluctuations between runs approximately average out. We have confirmed that this is the case, running multiple instances of Benchpress for Qiskit and noting that the results do not change appreciably between runs.

The QTS utilizes much of the Qiskit transpilation pipeline internally, differing only in the use of a routing method based on reinforcement learning[5]. Therefore, we expect to see identical results for

https://doi.org/10.1038/s43588-025-00792-y

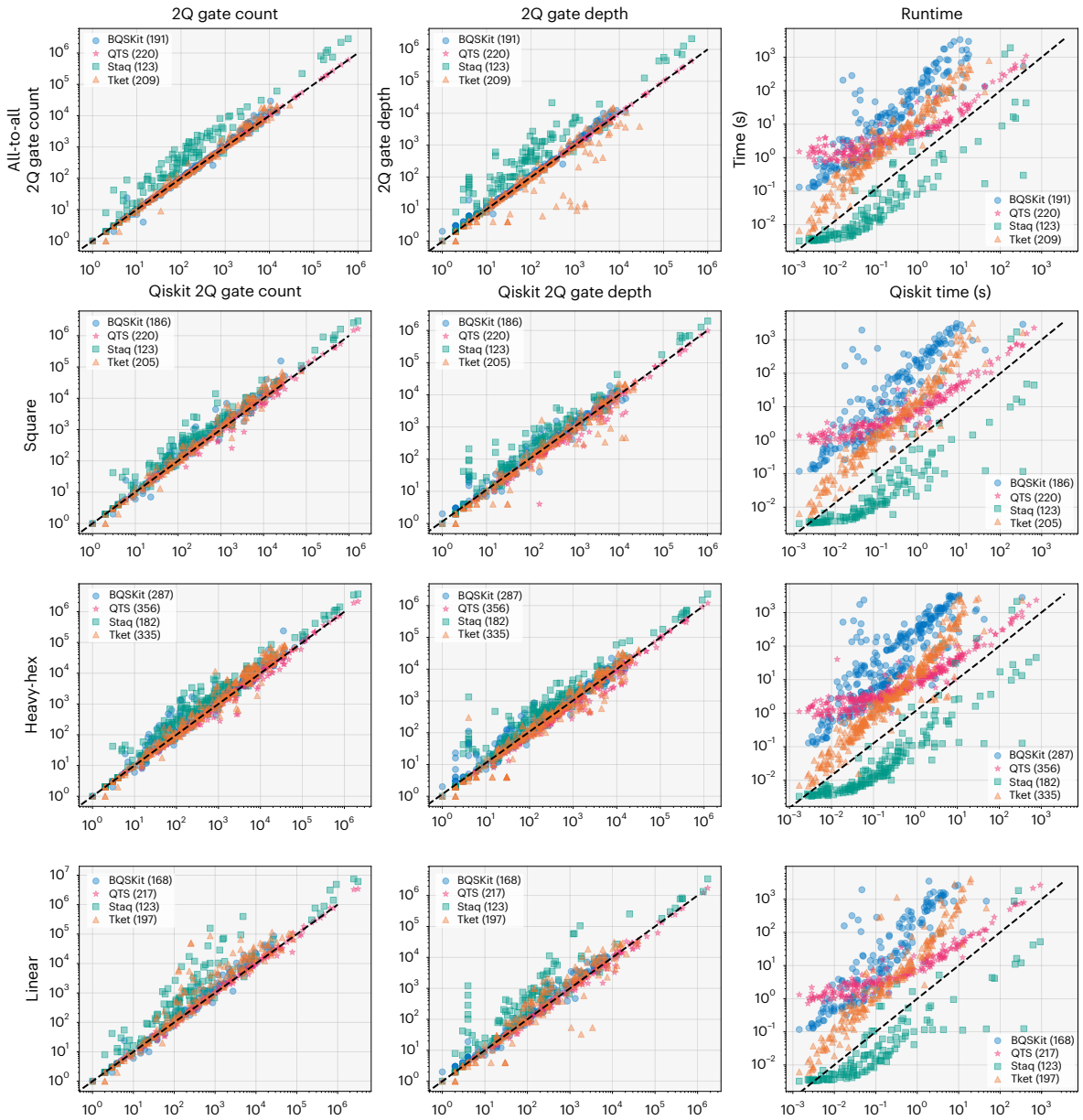


Fig. 3 | Results generated from the device and abstract transpilation tests in Benchpress. The rows indicate the topology used in the tests, whereas the columns label the reported metric. For brevity, only the top row of plots is labeled. The target topology for the device transpilation tests is 'heavy-hex', using a snapshot of the IBM Torino device as the target processor. Markers above the dashed line indicate tests where Qiskit performs better on a given metric relative to the specified SDK. By contrast, markers below highlight SDK performance that is better than that of Qiskit. The total number of passed tests in each dataset is given in the legends.

2Q gate count and depth between Qiskit and the QTS for the all-to-all coupling tests. This is confirmed in Table 2. In terms of 2Q-gate count, the QTS performs a few percent better than Qiskit on restricted coupling maps, with on average an 11% reduction on heavy-hex topologies. However, larger gains from the QTS appear in the 2Q-gate depth where a geometric mean improvement of 12% over Qiskit is observed over all topologies, and up to 22% when looking solely at heavy-hex results. These results are in line with the fact that the QTS routing step is trained only over heavy-hex topologies and, thus, is expected to perform better there. Like the Tket 2Q-depth results, the performance gains from the QTS come from a handful of tests with markedly lower depths than the corresponding Qiskit circuits, as opposed to an across-the-board advantage. However, in contrast to Tket where the synthesis step is responsible for the improved depth characteristics, the QTS benefits are isolated to the routing stage.

The differences in test runtimes shown in Fig. 3 clearly show that Staq is the fastest of the transpilation pipelines, while the QTS and Tket are over an order of magnitude slower that Qiskit. BQSKit has the longest runtimes, performing two orders slower than Qiskit. Note that, for circuits over restricted topologies that take ~10 s or more, as measured by Qiskit, the corresponding Tket time begins to diverge, suggesting an unfavorable scaling for the routing algorithm used by Tket compared with that of Qiskit. The results for Staq are an excellent example of the quality-versus-time trade-off made when designing transpilation pipelines; it is possible to gain performance at the cost of reduced circuit quality or vice versa. As discussed in the Methods, Staq also does not perform the same collection of steps as the other SDKs. Being a service, the QTS has a minimum input–output (IO) time of a few seconds, as seen in Fig. 3 for tests with short Qiskit runtimes. By contrast, this IO overhead is immaterial for more complex circuits

Table 2 | Geometric mean[49]/median values for SDK performance metrics, normalized to the corresponding Qiskit values

	BQSKit			QTS			Staq[b]			Tket		
	2Q gates	2Q depth	Time	2Q gates	2Q depth	Time	2Q gates	2Q depth	Time	2Q gates	2Q depth	Time
All tests[a]	1.26/1.13	1.27/1.18	108/98.0	0.96/1.0	0.89/1.0	18.7/14.5	2.8/2.45	2.91/2.40	0.26/0.22	1.31/1.09	0.98/1.00	13.3/12.7
All-to-all	1.04/1.00	1.11/1.05	75.0/71.0	1.0/1.0	1.0/1.0	18.3/15.8	3.35/3.76	4.02/4.07	0.36/0.34	1.08/1.00	0.75/1.00	14.2/14.5
Square	1.31/1.25	1.30/1.22	104/112	0.96/1.0	0.85/0.97	19.1/14.8	2.15/2.13	2.36/2.14	0.23/0.19	1.21/1.12	0.95/1.00	12.4/12.1
Heavy-hex	1.50/1.32	1.47/1.29	125/129	0.90/1.0	0.82/0.92	18.6/13.5	2.27/2.09	2.31/1.77	0.21/0.19	1.30/1.16	1.04/1.07	12.0/11.2
Linear	1.23/1.08	1.25/1.20	115/92.1	0.98/1.0	0.91/0.99	21.5/17.0	3.92/3.48	3.66/2.75	0.30/0.31	1.76/1.31	1.21/1.10	17.2/14.5

Results are presented over all tests combined, as well as sorted by target device topology. [a]All tests for BQSKit, QTS and Tket includes all nonfailing tests out of a total possible 1,032. [b]Staq results are over the subset of 551 tests that do not include a synthesis step.

that take longer time, and it is seen that the QTS runtime remains longer than that of Qiskit irrespective of the target topology, but this overhead is approximately constant for each topology.

While Qiskit completes the full set of transpilation tests in 0.18 days, the failed tests for the other SDKs makes a direct timing comparison impractical. However, using the relative timing information in Table 2 together with the corresponding Qiskit test times, we can estimate the full transpilation test suite time of the remaining SDKs. As the fastest SDK tested, the estimated full time of the Staq tests comes in less than Qiskit at 0.15 days. By contrast, BQSKit, QTS and Tket are estimated to run more than a day at 20.5, 1.3 and 4.73 days, respectively.

Targeting a fake IBM Quantum system that includes timing information, we can use the device transpilation tests to gauge the relative timing for circuit transpilation verses that of actual execution on hardware. To this end, we schedule each Qiskit circuit to obtain the execution time on chip and add the idle time between circuits, called the default_rep_delay in Qiskit, to get the total time per circuit execution. Note that these data were added after the initial version of Benchpress and are not included in the published results found in 'Data availability'. Multiplying this value by the default number of executions for IBM Quantum hardware, currently 4,096, gives us a good estimate for circuit execution time without requiring hardware usage. For the Qiskit device transpilation tests, the total compilation time was 1,055 s, whereas the total execution time of all circuits would be 968 s. Restricting ourselves to circuits with ≥100 qubits, the total compilation time is ~1.4× longer than the 78-s hardware runtime; the compilation costs begin to dominate at larger qubit counts. Table 2 shows that other comparable SDKs such as BQSKit and Tket take one to two orders of magnitude longer to compile circuits, while only marginally increasing the overall circuit depth, indicating that the ratio of compilation to runtime is more pronounced when using these compilation stacks to target the same quantum device. Note, however, that this conclusion is dependent on the targeted quantum hardware modality. For example, platforms such as trapped-ion systems have greater connectivity and, in general, require less compilation time because of this. In addition, trapped-ion runtimes are two orders of magnitude or more longer than those on superconducting devices[38], and therefore the ratio of compilation to hardware runtimes may be less than those presented here when targeting diffing hardware platforms.

Not included in Fig. 3 and Table 2 is information on skipped or failed tests. Outside of the 22 device transpilation tests that do not fit the target device, Qiskit is the only SDK that passed the full 1,032 collection of tests. BQSKit failed 200 (19%), QTS 19 (2%), Staq 2 (0.3%) and Tket 86 (8%) tests. Note that, as an OpenQASM-based compiler, Staq can execute only the 551 tests that do not include synthesis. These failures are, in large part, due to the timeout limit set on the tests. However, additional failure modes include QASM parsing issues in BQSKit, IO service errors in the QTS, C++ errors in Tket and circuit validation failures in Staq, where the output circuit did not match the topology of the target device.

Discussion

Benchpress is designed for transparent and reproducible comparisons between quantum SDKs. Benchpress, along with all of the results presented here, are open-sourced and can be readily executed by anyone looking to validate our findings. Indeed, our aim is to make the benchmarking process community-driven, with experts fluent in each SDK helping to refine the testing process or push it in new directions. Unlike most other benchmarking frameworks, we envision a fluid approach to quantum SDK testing, with tests being refined as quantum hardware and software mature, and target performance metrics being adjusted or expanded as necessary, using Benchpress version numbering to track the underlying test suite changes.

This work aims to benefit the community of quantum researchers, developers and end users at large by providing a trusted source by which absolute and/or relative improvements in quantum software can be faithfully examined. For researchers and developers, this can mean highlighting performance bottlenecks that need to be improved. For example, the small number of circuits on which both Tket and QTS outperform Qiskit in terms of 2Q-gate depth are of interest to us and identify areas for improved circuit synthesis and routing, respectively, in Qiskit. For others, this work may serve as a guide for where to target future SDK improvements for improving runtime and/or fixing coding bugs. As an open-source project, Benchpress is aimed at benefiting the quantum community as a whole, and we welcome contributions in the form of bug fixes, code improvements and new test cases at the project's GitHub website in 'Code availability'. Finally, end users can use this work as a guide for selecting the appropriate SDK, or perhaps combinations of SDKs, that are optimal for a given task. In all cases, shedding light on the performance of quantum computing software can only push the field further and aid in the successful adoption of this computing paradigm.

Methods

Test suite

The organization of Benchpress is shown in Extended Data Fig. 1. The test suite does not need to be installed, but executing the tests requires pytest[39] and any SDK-specific dependencies. In addition, we use pytest-benchmark[40], which provides a robust method for timing tests and generating result information in JSON format. However, because these tools are typically used in the context of unit testing, they are primarily aimed at tests requiring only short durations of time. Here, we are interested in the opposite regime where test times can easily approach an hour or more and, in some cases, have run for a week before being manually terminated. With the large number of tests considered here, running all tests to completion is impractical. As such, we run all tests using a modified version of pytest-benchmark that wraps each test in a subprocess that can optionally be terminated after a specified timeout. Tests that exceed a timeout are automatically added to the skipfile.txt shown in Extended Data Fig. 1. This file dramatically reduces the overhead from repeated test evaluations. However, it

is tied to the specific computer on which it is generated and can mask performance improvements if used across different SDK versions. The file used in this work was generated using the same SDK versions given in Table 1. Note that using subprocesses to enforce timing can have adverse effects when timing software uses parallel processing. As such, the modified version of pytest-benchmark used for each SDK differs in the mechanism by which it spawns processes.

To define a uniform collection of tests across SDKs, Benchpress uses abstract classes of tests called workouts, where each test is defined as a method to a Python class, where the name of the method defines the test name, and each test is decorated with xpytest-benchmark by default. Each class of tests is also logically organized into groups using pytest-benchmark. Implementing the actual tests for a given SDK requires overloading the abstract definitions in the workouts with a specific implementation. These are included in the 'gym' directory corresponding to the given SDK (Fig. ??) and grouped into their respective categories. We have included a verification mechanism that verifies that only those tests defined in the workouts are allowed to be present at the gym level. We enforce this gym partitioning so that each SDK can be run in isolation, and we execute the tests in each gym in a separate environment. However, to make a uniform testing experience across SDKs, Benchpress makes use of Qiskit throughout its infrastructure, in particular its compatibility with other SDKs and OpenQASM import and export capabilities, to supplement functionality missing in other SDKs, as well as provide reference implementations for data such as abstract backend entangling gate topologies that can, with minimal effort, be consumable by the other SDKs. Thus, Qiskit is a requirement common across all SDKs, but the version of Qiskit can differ, needing to satisfy the minimal requirements only.

Customizing the execution process is done via a configuration file, default.conf, that allows one to set Benchpress-specific options such as the target system used for device transpilation, as well as the basis gates and set of topologies utilized for the abstract transpilation tests defined in the Methods. We have decided to allow multiple device topologies within a given benchmark run, but the basis gates are fixed throughout. This choice is motivated by the fact that we explicitly focus only on the number and depth of 2Q gates in a circuit. With most 2Q gates equal in number up to additional single-qubit rotations, the basis set has less impact on final results than the choice of topology. SDK-specific settings, such as optimization level, can also be set in this file, and additional options can be easily added as there is no hard coding of parameters. In addition, pytest and pytest-benchmark options can be set using the standard pytest.ini file. In this file, we add the flag to allow only a single execution of a test to be performed, as opposed to the usual minimum of five to make runtimes manageable.

Test result definitions

Benchpress accommodates SDKs with disparate feature sets by running the full test suite over each SDK, regardless of whether the individual tests are supported. Our test environment is based on pytest?, and we map each of the standard pytest output types to the following definitions:

- PASSED: This indicates that the SDK has the functionality required to run the test, and doing so completed without error and within the desired time limit, if any.
- SKIPPED: The SDK does not have the required functionality to execute the test, or the test does not satisfy the problem's constraints, for example, the input circuit is wider than the target topology. This is the default status for all notional tests.
- FAILED: The SDK has the necessary functionality, but the test failed or was not completed within the set time limit, if any.
- XFAIL: The test fails irrecoverably. It is therefore tagged as 'expected fail' rather than being executed. For example, a test is trying to use more memory than available. Note that, because we execute tests in subprocesses to implement a timeout mechanism, some failures can kill the subprocess but otherwise not affect the remaining tests. These are considered FAILED per the definition here.

All tests have notional definitions called workouts (Methods) that are placeholders for SDK-specific implementations and default to SKIPPED unless explicitly overwritten in each SDK test suite. In this way, Benchpress can use skipped tests as a proxy for measuring the breadth of SDK functionality, and this can be tracked automatically when additional functionality is added in the form of new tests. Figure 1 shows the distribution of tests by status for the results presented here when running the full Benchpress test suite against each SDK. While the specifics of test failures will be discussed below, we note that 97% of failures occur when running benchmarks originating from other test suites, as opposed to those created explicitly for Benchpress.

Circuit construction and manipulation tests

Although nominally a tiny part of an overall circuit processing workflow time budget, as compared with circuit transpilation, measuring the timing of circuit construction and manipulation gives a holistic view of quantum SDK performance. Moreover, if suitably chosen, such tests can provide insights into other parts of the entire circuit compilation process. The present version of Benchpress includes 12 such tests, with tests aimed at representing scenarios encountered during real-world SDK usage. Circuit construction includes eight tests that look at timing information needed to build 100-qubit circuits for families of circuits such as quantum volume (QV)[41] (test_QV100_build), Hamiltonian simulation[16] (test_DTC100_set_build), random Clifford circuits[42] (test_clifford_build), 16-qubit iterative construction of multi-controlled gates (test_multi_control_circuit) and parameterized ansatz circuits with circular entangling topology (test_param_circSU2_100_build). We will reuse many of these circuits in device benchmarking. We also include the time to bind values to parameterized circuits (test_param_circSU2_100_bind) and import from OpenQASM files into the construction category. This latter set of QASM tests includes importing a 100-qubit QV circuit and reading a file with an integer corresponding to a 301-bit classical register, test_QV100_qasm2_import and test_bigint_qasm2_import, respectively. Our focus on 100-qubit circuits stems from the need for sufficient complexity for gathering faithful timing information and the fact that these circuits are within the number of qubits available on present-day quantum processors.

Circuit manipulation is the set of operations that can be performed on a fully built circuit. Out of the four such tests included, two represent basis transformations, test_QV100_basis_change and test_random_clifford_decompose[43], taking an input OpenQASM file and expressing them in a differing set of gates. In a similar vein, we use the same multicontrolled circuit used in the circuit construction tests and time the decomposition into a QASM-compatible gate set in test_multi_control_decompose. In contrast to the previous tests, this decomposition requires a nontrivial synthesis step and provides additional insight into how well the SDKs transform abstract quantum circuits into primitive components. This is captured in the number of 2Q gates in the circuit returned at the end of the test, and this value is also recorded. Finally, we also implement Pauli twirling[44,45] in each SDK, test_DTC100_twirling, recording the time it takes to twirl 19,800 CNOT gates in a Hamiltonian simulation circuit.

Circuit transpilation tests

Due to their vast array of possible input parameters and a large fraction of overall runtime, transpilation tests form the bulk of the tests in Benchpress. We split these tests into two groups depending on whether they target a model of a real quantum device, or if their target is an abstract topology defined by a generating function. We label these as 'device' and 'abstract' transpilation tests, respectively. These tests

differ because device testing targets a fixed model of a quantum system, regardless of input quantum circuit size, and includes error rates that can be utilized in noise-aware compilation routines. Noise-aware heuristics can have an impact on both the duration of the compilation process, as well as the 2Q gate count and depth; they can paradoxically lead to worsened performance if applied overly aggressively. However, because we do not execute the resulting circuits on hardware, the impact of these techniques on output fidelity is not included. By contrast, abstract transpilation tests take an input circuit and finds the smallest topology compatible with the circuit. In this manner, we can benchmark SDKs across arbitrary circuit sizes and topology families, allowing for user configuration of the basis gates in the default.conf file (Fig. ??).

Device transpilation tests come from three sources. First, we include a collection of tests that focus primarily on 100-qubit circuits representing circuit families such as QV, quantum Fourier transform, Bernstein–Vazirani (BV) and random Clifford circuits. In addition, circuits generated from Heisenberg Hamiltonians over a square lattice and the quantum approximate optimization algorithm circuits corresponding to random instances of a Barabási–Albert graph are also included. We also add 100- and 89-qubit instances of the same parameterized ansatz circuits used for the circuit construction and manipulation tests, where the former can be embedded precisely on a heavy-hex device, that is, there is an ideal mapping, while the latter cannot. This set also includes a circuit with a BV-like structure, but where the circuit can be optimized down to single-qubit gates if transpiled appropriately. Because this set of circuits is represented in OpenQASM form or generated using QASM-compatible gates only, they do not test the synthesis properties of each SDK. To do so, we include a set of 100 abstract circuits using Hamiltonians included in the HamLib library[11] for time evolution. The choice of Hamiltonians is described in 'Hamiltonian selection criteria' and results in a set of Hamiltonians from 2 to 930 qubits in size. Finally, we include the Feynman collection[8] of circuits that are up to 768 qubits, and also OpenQASM-based, in device transpilation tests. Depending on the target quantum system for device transpilation, some device tests may be skipped due to insufficient physical qubit count. For benchmarking against abstract topologies, we run the same set of Hamiltonian simulation circuits run for device transpilation and include OpenQASM tests from QasmBench[9] that go up to 433 qubits.

Our performance metrics for both sets of tests are 2Q gate count, 2Q gate depth and transpilation runtime. In addition, we record the number of qubits in the input circuit, QASM load time (if any) and number and type of circuit operations at the output. Any additional metrics that are compatible with JSON serialization can be included. The target system used in the device transpilation tests is the FakeTorino system, which is a snapshot of a 133-qubit Heron system from IBM Quantum that includes calibration data suitable for noise-aware compilation. Abstract topologies tested are all-to-all, square, heavy-hex and linear, which includes most typical device topologies, and are predefined graphs in the rustworkx library[46]. We have configured the abstract models to use the basis set ['sx', 'x', 'rz', 'cz']. Finally, to limit the duration of the tests, we have set a timeout limit of 3,600 s (1 h), after which the test is marked as FAILED.

In this Article, we focus on testing the predefined transpilation pipelines in each SDK. In this way, we aim to measure the relative performance that a typical user would see, and eliminate the bias involved when creating bespoke transpilation workflows in SDKs of which we have less knowledge than Qiskit. In making Benchpress open-source, we hope that comparisons of optimal performance can be led by community experts in each SDK. Here, we use the default optimization values for both Tket (2) and BQSKit (1). Qiskit does not have a well-defined default optimization level, with the transpile function having a default value of 1, whereas the newer generate_preset_passmanager interface must have the optimization level explicitly set. In this work, we use optimization level 2 for Qiskit that will be the default value for both ways of calling the transpiler functionality starting in version 1.3.0. This same optimization value is used for the QTS as well. Staq was set to optimization level 2 to generate circuits valid for the target topologies. All other transpiler values are left unchanged.

Hamiltonian selection criteria

Hamiltonians included in Benchpress originate from the HamLib Hamiltonian library[11], which includes problems from chemistry, condensed matter physics, discrete optimization and binary optimization. We randomly selected Hamiltonians from HamLib to be included in the benchmark suite presented in this work. The random selection is biased toward reflecting the distribution of Hamiltonian characteristics prevalent in HamLib such as the number of qubits and number of Pauli terms. Furthermore, we limited the number of qubits in the selected Hamiltonians to ≤1,092 and the number of Pauli terms to 10,000 or fewer. Furthermore, the random selection is biased toward 'unique' Hamiltonians; the selection of different encodings of the same Hamiltonian is discouraged. One-hundred HamLib Hamiltonians are included in this version of Benchpress, where 35 Hamiltonians are from chemistry and condensed matter physics problem classes each, and 15 Hamiltonians are chosen from both discrete and binary optimization problem classes.

SDK-specific considerations

Given quantum computing software's nascent stage of development, it is common to encounter pitfalls when benchmarking these software stacks. Here, we detail some of these issues as they pertain to executing tests in Benchpress.

BQSKit. BQSKit performs unitary synthesis up to a maximum size specified by the max_synthesis_size argument to the compiler. The compiler will fail if a unitary is larger than this value. However, given an OpenQASM circuit, an end user must first parse the file to learn the correct size for this argument; the returned error message does not include the required value. As there is no manner outside of parsing files to gain this information, we have this parameter to the default value, max_synthesis_size=3, letting tests fail if they have unitary gates outside of this value.

In addition, BQSKit does not support coupling maps that correspond to 2Q entangling gates with directionality; the topology is assumed to be symmetric. The *CZ* gate used in this work is a symmetric gate, and thus the circuits returned from the BQSKit transpiler pass the structural validation performed here. Selecting a directional gate, such as an echoed cross-resonance gate, would, in general, fail validation.

Qiskit transpiler service. The default timeout value for the QTS is less than the 3,600-s timeout used in this work. As such, we explicitly set the timeout value to match when calling the QTS service for each test.

Tket. The OpenQASM import functionality in Tket requires the user to specify the size of classical registers in the circuit if those registers are larger than 32 bits. Given an arbitrary OpenQASM file, the user must first parse the file to gather the size of the classical registers or try importing the file first, capturing the exception and reading the register size from the error message. To get around this limitation, Benchpress includes a maxwidth parameter in the Tket section of the default.conf file that allows one to specify a maximum allowed classical register size. Given that the maximum number of qubits in the OpenQASM files is 433, we have set this value to 500 in the default.conf.

Staq. Staq cannot return quantum circuits in the basis set of the target backend. Instead, the output is always expressed in generic one-qubit unitary *U* and CNOT gates. Because of this, we perform structural validation only on circuits returned by Staq. In addition, we compute

2Q gate counts and depth on the CNOT gates. This is valid provided the target 2Q gate is equivalent to a CNOT gate up to single-qubit rotations. For the *CZ* gate used here, this relation holds.

Optimization level 3 of Staq includes the compiler flag -c that applies a CNOT optimization pass. This pass generates output OpenQASM files that do not obey the entangling gate topology of the target device; the output circuits fail the structural validation check at the end of each test. As such, we have set the default optimization level of Staq to 2.

Data availability

Results used for all figures and tables in this study are available via Zenodo at https://doi.org/10.5281/zenodo.14977295 (ref. 47) or the published_results directory at the 1.0 branch of the Benchpress repository via GitHub at https://github.com/Qiskit/benchpress/tree/1.0, where data are stored in JSON format. Source data are provided with this paper.

Code availability

All code used in generating this dataset is open-source and available via Zenodo at https://doi.org/10.5281/zenodo.14977334 (ref. 48) or the Benchpress repository via GitHub at https://github.com/qiskit/benchpress.

References

1. Amazon Braket. *AWS* https://aws.amazon.com/braket/ (2024).
2. Younis, E. et al. Berkeley Quantum Synthesis Toolkit (bqskit) v1. *US Department of Energy* https://doi.org/10.11578/dc.20210603.2 (2021).
3. Cirq Developers. Cirq. *Zenodo* https://doi.org/10.5281/zenodo.4062499 (2024).
4. Javadi-Abhari, A. et al. Quantum computing with Qiskit. Preprint at https://arxiv.org/abs/2405.08810 (2024).
5. Kremer, D. et al. Practical and efficient quantum circuit synthesis and transpiling with reinforcement learning. Preprint at https://arxiv.org/abs/2405.13196 (2024).
6. Amy, M. & Gheorghiu, V. staq—a full-stack quantum processing toolkit. *Quant. Sci. Technol.* **5**, 034016 (2020).
7. Sivarajah, S. et al. t|ket⟩: a retargetable compiler for nisq devices. *Quant. Sci. Technol.* **6**, 014003 (2020).
8. Amy, M. Towards large-scale functional verification of universal quantum circuits. In *Proc. 15th Int. Conf. Quantum Phys. Log.* (eds. Selinger, P. & Chiribella, G.) 1–21 (EPTCS, 2019); https://doi.org/10.4204/EPTCS.287.1
9. Li, A., Stein, S., Krishnamoorthy, S. & Ang, J. QASMBench: a low-level quantum benchmark suite for NISQ evaluation and simulation. *ACM Trans. Quant. Comput.* **4**, 1–26 (2023).
10. Mori, Y. et al. Quantum circuit unoptimization. Preprint at https://arxiv.org/abs/2311.03805 (2023).
11. Sawaya, N. P. et al. HamLib: a library of Hamiltonians for benchmarking quantum algorithms and hardware. In *2023 IEEE Int. Conf. Quantum Comp. Eng.* vol. 2, 389 (IEEE, 2023); https://doi.org/10.1109/QCE57702.2023.10296
12. Kharkov, Y., Ivanova, A., Mikhantiev, E. & Kotelnikov, A. Arline benchmarks: automated benchmarking platform for quantum compilers. Preprint at https://arxiv.org/abs/2202.14025 (2022).
13. Quetschlich, N., Burgholzer, L. & Wille, R. MQT Bench: benchmarking software and design automation tools for quantum computing. *Quantum* **7**, 1062 (2023).
14. Cross, A. et al. Openqasm 3: a broader and deeper quantum assembly language. *ACM Trans. Quant. Comput.* **3**, 1 (2022).
15. Yan, G. et al. Quantum circuit synthesis and compilation optimization: overview and prospects. Preprint at https://arxiv.org/abs/2407.00736 (2024).
16. Zhang, V. & Nation, P. D. Characterizing quantum processors using discrete time crystals. Preprint at https://arxiv.org/abs/2301.07625 (2023).
17. Kim, Y. et al. Evidence for the utility of quantum computing before fault tolerance. *Nature* **618**, 500–505 (2023).
18. Yu, H., Zhao, Y. & Wei, T.-C. Simulating large-size quantum spin chains on cloud-based superconducting quantum computers. *Phys. Rev. Res.* **5**, 013183 (2023).
19. Majumdar, R., Rivero, P., Metz, F., Hasan, A. & Wang, D. S. Best practices for quantum error mitigation with digital zero-noise extrapolation. Preprint at https://arxiv.org/abs/2307.05203 (2023).
20. Shtanko, O. et al. Uncovering local integrability in quantum many-body dynamics. *Nat. Commun.* **16**, 552 (2025).
21. Yasuda, T. et al. Quantum reservoir computing with repeated measurements on superconducting devices. Preprint at https://arxiv.org/abs/2310.06706 (2023).
22. Chen, E. H. et al. Nishimori transition across the error threshold for constant-depth quantum circuits. *Nat. Phys.* **21**, 161–167 (2025).
23. Farrell, R. C., Illa, M., Ciavarella, A. N. & Savage, M. J. Scalable circuits for preparing ground states on digital quantum computers: the Schwinger model vacuum on 100 qubits. *PRX Quantum* **5**, 020315 (2024).
24. Pelofske, E., Bärtschi, A., Cincio, L., Golden, J. & Eidenbenz, S. Scaling whole-chip QAOA for higher-order Ising spin glass models on heavy-hex graphs. *npj Quant. Inf.* **10**, 109 (2024).
25. Bäumer, E. et al. Efficient long-range entanglement using dynamic circuits. *PRX Quant.* **5**, 030339 (2024).
26. Acharya, R. et al. Quantum error correction below the surface code threshold. *Nature* **638**, 920–926 (2024).
27. Bluvstein, D. et al. Logical quantum processor based on reconfigurable atom arrays. *Nature* **626**, 58–65 (2024).
28. Farrell, R. C., Illa, M., Ciavarella, A. N. & Savage, M. J. Quantum simulations of hadron dynamics in the Schwinger model using 112 qubits. *Phys. Rev. D* **109**, 114510 (2024).
29. Shinjo, K., Seki, K., Shirakawa, T., Sun, R.-Y. & Yunoki, S. Unveiling clean two-dimensional discrete time quasicrystals on a digital quantum computer. Preprint at https://arxiv.org/abs/2403.16718 (2024).
30. Miessen, A., Egger, D. J., Tavernelli, I. & Mazzola, G. Benchmarking digital quantum simulations above hundreds of qubits using quantum critical dynamics. *PRX Quant.* **5**, 040320 (2024).
31. Robledo-Moreno, J. et al. Chemistry beyond exact solutions on a quantum-centric supercomputer. Preprint at https://arxiv.org/abs/2405.05068 (2024).
32. Montanez-Barrera, J. A. & Michielsen, K. Towards a universal QAOA protocol: evidence of a scaling advantage in solving some combinatorial optimization problems. Preprint at https://arxiv.org/abs/2405.09169 (2024).
33. Cadavid, A. G., Dalal, A., Simen, A., Solano, E. & Hegade, N. N. Bias-field digitized counterdiabatic quantum optimization. Preprint at https://arxiv.org/abs/2405.13898 (2024).
34. Alevras, D. et al. mRNA secondary structure prediction using utility-scale quantum computers. Preprint at https://arxiv.org/abs/2405.20328 (2024).
35. Sachdeva, N. et al. Quantum optimization using a 127-qubit gate-model IBM quantum computer can outperform quantum annealers for nontrivial binary optimization problems. Preprint at https://arxiv.org/abs/2406.01743 (2024).
36. cuda-quantum. *GitHub* https://github.com/NVIDIA/cuda-quantum (2024).
37. qiskit-braket-provider. *GitHub* https://github.com/qiskit-community/qiskit-braket-provider (2024).
38. Lötstedt, E. & Yamanouchi, K. Comparison of current quantum devices for quantum computing of heisenberg spin chain dynamics. *Chem. Phys. Lett.* **836**, 140975 (2024).

39. pytest. *GitHub* https://github.com/pytest-dev/pytest (2024).
40. pytest-benchmark. *GitHub* https://github.com/ionelmc/pytest-benchmark/ (2024).
41. Cross, A. W., Bishop, L. S., Sheldon, S., Nation, P. D. & Gambetta, J. M. Validating quantum computers using randomized model circuits. *Phys. Rev. A* **100**, 032328 (2019).
42. Bravyi, S. & Maslov, D. Hadamard-free circuits expose the structure of the Clifford group. *IEEE Trans. Inf. Theory* **67**, 4546 (2021).
43. Bravyi, S., Shaydulin, R., Hu, S. & Maslov, D. Clifford circuit optimization with templates and symbolic Pauli gates. *Quantum* **5**, 580 (2021).
44. Knill, E. Fault-tolerant postselected quantum computation: threshold analysis. Preprint at https://arxiv.org/abs/quant-ph/0404104 (2004).
45. Wallman, J. J. & Emerson, J. Noise tailoring for scalable quantum computation via randomized compiling. *Phys. Rev. A* **94**, 052325 (2016).
46. Treinish, M., Carvalho, I., Tsilimigkounakis, G. & Sá, N. rustworkx: a high-performance graph library for python. *J. Open Source Softw.* **7**, 3968 (2022).
47. IBM Quantum & Nation, P. Results generated from Benchpress 1.0 used in Nat. Comput. Sci. publication. *Zenodo* https://doi.org/10.5281/zenodo.14977295 (2025).
48. IBM Quantum & Nation, P. Benchpress 1.0. *Zenodo* https://doi.org/10.5281/zenodo.14977334 (2025).
49. Fleming, P. J. & Wallace, J. J. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM* **29**, 218 (1986).

Acknowledgements

We thank J. Gacon, A. Ivrii and J. Lishman for helpful discussions. L.B., M.T. and A.J.-A. were supported by the US Department of Energy, Office of Science, National Quantum Information Science Research Centers, Co-design Center for Quantum Advantage (C2QA) under contract number DE-SC0012704. The funders had no role in the study design, data collection and analysis, decision to publish or preparation of the manuscript.

Author contributions

P.D.N. devised the testing infrastructure and methodology, and contributed to the code base. A.A.S. wrote the tests for both the BQSKit and Staq SDKs and contributed to the infrastructure code. S.B. wrote the code for Hamiltonian simulation tests from HamLib, and selected the subset of Hamiltonians that are included in this work. L.B. added the test logic that allows for skipping tests with timeout logic, while S.G. added random Clifford tests. M.T. added the Feynman suite of tests for the device transpilation tests, and A.J.-A. suggested circuit libraries for inclusion and test cases to consider. All authors contributed to the paper.

Competing interests

The authors declare no competing interests.

Additional information

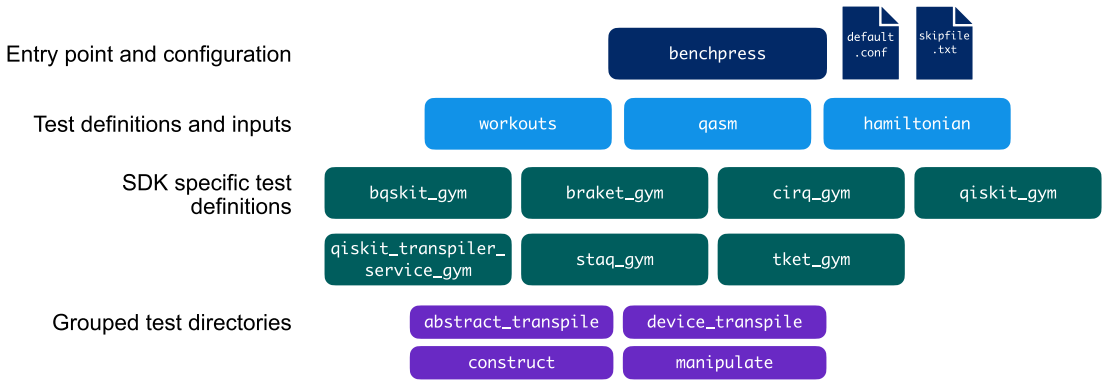
Extended data is available for this paper at https://doi.org/10.1038/s43588-025-00792-y.

Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s43588-025-00792-y.

Correspondence and requests for materials should be addressed to Paul D. Nation.

Peer review information *Nature Computational Science* thanks Tyson Jones and the other, anonymous, reviewer(s) for their contribution to the peer review of this work. Peer reviewer reports are available. Primary Handling Editor: Jie Pan, in collaboration with the Nature Computational Science team.

https://doi.org/10.1038/s43588-025-00792-y



Extended Data Fig. 1 | Organization of the Benchpress data set. The entry point is the benchpress directory, and SDK specific flags and options are set in the default.conf file. Optionally, tests taking longer than a specified timeout can be automatically included in the skipfile.txt. Abstract test definitions are included in the workouts directory, and test inputs in the form of OpenQASM files or Hamiltonians are included in the qasm and hamiltonian folders, respectively. Tests specific to each SDK are located in the corresponding "*_gym" directories. Inside each "gym", tests are organized in groups based on the target functionality to be tested.