ABAQUS

Automated Benchmarking of Algorithms for Quantum Systems

August 2021

AUTHOR:

Samuel González-Castillo

SUPERVISORS:

Elías F. Combarro
Alberto Di Meglio
Sofia Vallecorsa

PROJECT SPECIFICATION

With this project, we aim to develop a benchmarking platform for both the software frameworks and hardware devices used in the simulation of quantum computers. This benchmarking platform should, by default, incorporate some basic benchmarks and be able to run on some of the most common quantum simulation frameworks.

At this initial stage, the goal we pursue is not writing as many benchmark tests as possible for as many software frameworks as possible. Instead, our focus is on building a solid foundation upon which it be easy to write new benchmarks and add support for new frameworks. This is done, with an eye on the future, on the hope of making this benchmarking platform a useful tool for the whole community, in a way that will enable anyone to easily include new benchmark tests and add interfaces to new software frameworks.

ABSTRACT

We have developed a software platform, ABAQUS (Automated Benchmarking of Algorithms for Quantum Systems), that can be used to benchmark the performance of both software frameworks and hardware devices used in the simulation of quantum computers. This platform has been developed as a Python package which can be easily extended with interfaces to any quantum software framework, on top of which the benchmarks implemented in the package can be run. What is more, these interfaces do not define or rely on the specific benchmarks that are run: these and their details are fully controlled by the ABAQUS package and they are common to all the frameworks.

So far, ABAQUS incorporates some basic benchmarks based on random circuits, single-qubit and two-qubit gates and the Quantum Fourier Transform.

In addition to this, we have designed a user-friendly web application in Flask that presents all the benchmark results in a clear and interactive manner. This web application is fully connected to the ABAQUS framework in such a way that any changes in ABAQUS are automatically reflected on the companion web application — without any changes whatsoever on the web application source code.

TABLE OF CONTENTS

1. Introduction

Nowadays, we are witnessing the development of a very wide variety of frameworks for the simulation of quantum algorithms. In this scenario, any user willing to work in quantum computing faces an important issue of choice, for some frameworks are better than others based on different parameters (algorithms that are going to be run, number of qubits that are going to be used, or available hardware). Thus, it seems very pertinent to develop a benchmarking tool that could allow anyone to make informed choices about what quantum software frameworks to use. In addition, of course, such a benchmarking tool could be useful towards fostering some healthy competition between framework developers.

On the other hand, we also have this issue of choice when it comes to hardware. Any user can find themselves wondering what specifications they should be looking for in a specific device to get the best performance in the simulation of certain quantum algorithms with certain frameworks.

ABAQUS is an answer to both these issues. To begin with, ABAQUS provides a Python package that implements a suite of benchmarks that can be potentially run on any quantum framework and, of course, that can be executed on any type of device. All these benchmarks are, by design, consistent across all the different frameworks, and the ABAQUS package can be easily extended by anyone to include new tests and to add interfaces to new frameworks. In addition, this package logs information about the hardware specifications of the device in which the benchmarks are run and makes it possible for anyone to submit their benchmark results to a central repository. Using such a repository, a web application that we have developed can interactively compute and present benchmarking scores for both hardware devices and software frameworks.

ABAQUS is a benchmarking platform that will be both reliable and open to the community. In particular, its by-design flexibility to incorporate new benchmark tests will enable ABAQUS to fit into the needs of any group of users. Therefore, ABAQUS sets a foundation for what can become a community-wide effort that will allow anyone involved in quantum computing to make informed choices about hardware and software.

In the upcoming sections, we will discuss how ABAQUS came to be: what design choices were made and why and how these choices were implemented in ABAQUS.

2. Collection and storage of information

Before implementing any benchmarking framework, we need to decide what information it is going to collect in order to compute benchmarks. In our case, we first need to identify the hardware specifications of the device in which the benchmarks are going to be run. At the time of writing, this is the data that ABAQUS collects about every device:

- A device name, which should identify it uniquely. By default, this value is the hostname of the device.

- An integer "version number" that can uniquely identify all the different configurations of a device (i.e., a version number that changes every time a device undergoes any configuration changes). By default, it is set to zero and it is increased automatically whenever ABAQUS identifies any hardware change.

- Some technical specifications: RAM, processor and number of cores. All this information is automatically registered. At the moment, ABAQUS does not log information about GPUs, but this might change in the future.

The device name and version number can, of course, be manually set by the user, and this would only have to be done once in every session.

In ABAQUS, devices are represented as objects of a `Device` class. The object representing the current device can be obtained with the nullary function `this_device`, and the name and version number of this object can be manually set calling a function, `set_device`, which accepts three optional parameters: `name`, `version` and `upgraded`. The latter is a Boolean which, if set to true, tells the function to increase the version number of `this_device()` in a unit.

A device `dev` can be saved invoking `dev.save()`. When doing so, the device information will be logged in a CSV file, and the device object will be loaded automatically in any future session. It will be accessible through a global variable, `devices`, that contains a dictionary of all the device objects indexed by their name and version number. Of course, saving a device more than once has no effect.

A similar approach is taken when handling quantum software frameworks. Frameworks are represented internally as objects of a `Framework` class. Each of these objects stores

- a unique identifier (uid) of the framework,

- the name of the framework,

- the developer of the framework

- and the framework website.

ABAQUS keeps track of framework objects through a global variable, `frameworks`, which references a dictionary of "known" frameworks indexed by their uid. Frameworks can be both created and retrieved using the `get_framework` function, which takes a positional argument for the uid and some optional arguments for the name, developer and website of the framework. If a framework with the provided uid exists in `frameworks`, it is returned by the function and the remaining arguments are ignored. Otherwise, a new framework is created with the supplied information and it is added to `frameworks`. In this way, the user can invoke the very same function every time they need to access a particular framework object, regardless of whether it has already been created or not.

As with devices, frameworks objects incorporate a nullary `save` method, which allows the user to store the framework information in a CSV file and will lead to the automatic re-loading of the framework object in any future session.

We can now discuss some ideas about how benchmark results are handled in ABAQUS. We will explore further details about benchmarks in the following sections.

The metric we have used to benchmark performance has been execution time. Moreover, since there is an intrinsic variance in the time it can take for any task to be executed in a computer, we log the execution time several times and then store the mean and the standard deviation. The handling of this information is eased thanks to a `TimeLog` class. Time-log objects are, for the most part, invisible to the average end-user — one only needs to deal with them, on a very superficial level, when extending ABAQUS with new interfaces or benchmark tests, — so we will not discuss them in detail. For our purposes, it will suffice to understand that time-log objects store the execution time of all the executions of a particular task and that, in order to do this, they provide methods that allow a user to toggle a timer at the beginning and at the end of a task.

Most quantum software frameworks go through two distinct phases when executing a quantum algorithm: a loading phase (where they prepare the quantum circuit) and an execution phase, where the corresponding circuit is run. The `TimeLog` class can store time metrics for these two distinct processes in a single object. The following table shows how this can be done.

Distinct loading and execution phases	Framework combines both phases

`timer = TimeLog()`
`timer.start()`
`# Loading phase`
`timer.log_load(); timer.start()`
`# Execution phase`
`timer.log_run()`
	`timer = TimeLog()`
	`timer.start()`
	`# Task is executed`
	`timer.log_total()`

The mean and standard deviation of the different metrics are stored in some object variables (`load_mean, load_std, run_mean, run_std, total_mean, total_std`), which are initialised to a `None` value and are updated dynamically as new logs are added.

In ABAQUS, benchmarks are internally encoded as objects of a `Benchmark` class. Objects of this class contain

- a framework and device object (representing, obviously, the framework and the device on which the benchmark was executed),
- the version of the framework
- and a collection of time-log objects with the benchmark results.

All benchmark objects include a `save` method that, after saving the associated framework and device objects, stores the benchmark information in a CSV file. The framework is stored in this file by just logging its uid and the device is stored by logging its name and version number. The time-log objects are stored by logging the values of their `load_mean, load_std, run_mean, run_std, total_mean` and `total_std` variables (the values of the individual execution times are lost).

Of course, saved benchmarks are recovered in future sessions and they are accessible from a global variable, `benchmarks`, which points to a dictionary with all the known benchmark objects indexed by device, framework and framework version. In addition to this, framework and device objects have a `benchmarks` variable, which is nothing more than an array containing all the benchmark objects in which they are involved. Naturally, all these variables are updated automatically as new benchmark objects are generated, without further intervention from the user's side.

3. Interfaces with quantum frameworks

Every quantum framework is, in some way or another, different. They all have different Python interfaces and different ways of handling algorithms, so how can we possibly implement a benchmarking platform that runs tests across a wide range of frameworks while ensuring consistency? Implementing each test in each framework would make ABAQUS very difficult to extend, so we have taken another approach.

In ABAQUS, all the benchmark tests are implemented as methods of the `Benchmark` class, and these methods are common to all the benchmark objects, regardless of frameworks. In order to accomplish this, the `Benchmark` class relies on a `run_circuit` method, which is empty and is re-defined in subclasses that implement the interfaces of ABAQUS with quantum frameworks.

The `run_circuit` function takes two arguments: an integer specifying a number of qubits, and an array which serves as a universal blueprint for a circuit on that number of qubits. In this way, we can let the `Benchmark` class handle all the details of benchmark tests, thus ensuring consistency, while leaving the actual execution of the circuits involved in these tests to the `run_circuit` function. This function only needs to be implemented once in any framework-specific subclass, and it makes the framework compatible with any of the current benchmark tests and with any other test that could be added to ABAQUS in the future.

The following array is an example of one such blueprint that can be passed to `run_circuit`:

```
[  [“RX”, [3]   , 0.4]

   [“H” , [0]   , 0  ]

   [“CX”, [0, 1], 0  ]   ]
```

This blueprint would ask the function to run a circuit consisting of

1. an X-Rotation gate on the fourth qubit (qubits are zero-indexed) parameterised by the value 0.4,

2. a Hadamard gate on the first qubit,

3. and a controlled-not gate on the first two qubits (with qubit 0 serving as control).

The function should be able to parse the Pauli X, Y, Z gates, the Hadamard gate, the controlled X, Y and Z gates, and a special “QFT” gate that would apply a quantum Fourier transform on all the available qubits.

One could perhaps spot some redundancy here, for the QFT gate can be easily expressed in terms of the other available elementary gates, so why bother including it as a special gate? The reason is that most frameworks include a QFT gate and, since, for some, using this QFT gate may lead to a more efficient execution, we believe it is necessary to allow the frameworks to use their built-in implementation of QFT.

In the future, more special gates will be added as the number of benchmark tests grows. In any case, to ensure backwards-compatibility, this will be done in a way that the ability to parse these special gates will not be a requirement, but an option.

At the time of writing, the author of this report has implemented interfaces for the Qiskit Statevector simulator (with and without GPU) and Pennylane, and has included the following benchmark tests:

- A random circuit test, which executes a circuit consisting of 1000 gates chosen and random and, when applicable, parameterized by random parameters.
- A Quantum Fourier Transform test.
- A single-qubit gate and two-qubit gate test.

At the time of writing, all the tests in ABAQUS are run on both 8 and 16 qubits, but this number will increase in the near future. When the time comes for this change, it will just require editing a single line of code.

4. Score computation

We have a software platform that allows us to obtain a lot of information, in terms of execution-time metrics, about the performance of hardware and software for the simulation of quantum computers. But now that we have all this raw information, we need to face another problem: how do we digest it? How do we present it to the user in a way that can be useful, informative and easy to understand? We should do this by computing some kind of score.

We have endowed ABAQUS with some simple yet powerful score systems. The scores produced by ABAQUS range from 0 (worst framework/device) to 100 (best framework/device) and they are not absolute, but relative. They are computed among a subset of the known frameworks/devices and the score of a particular framework/device will change as this subset changes. In particular, if this subset only contains a single framework/device, its score is guaranteed to be 100.

a. Framework scores

Given a subset of frameworks, we can compute their scores using a certain subset of tests run on a certain number of qubits. This score is computed following this algorithm:

```
For every device and for every test (in a certain number of qubits):
    Get the execution time of every framework and compute the lowest of those computation times, we shall call it t.
    For each framework with execution time x, compute the individual score: 100*x/t
The score of each framework is the mean of its individual scores.
```

b. Hardware scores

In full analogy with the way framework scores are computed, the algorithm is the following:

```
For every framework and for every test (in a certain number of qubits):
    Get the execution time on every device and compute the lowest of those computation times, we shall call it t.
    For each device with execution time x, compute the individual score: 100*x/t
The score of each device is the mean of its individual scores.
```

c. Other considerations

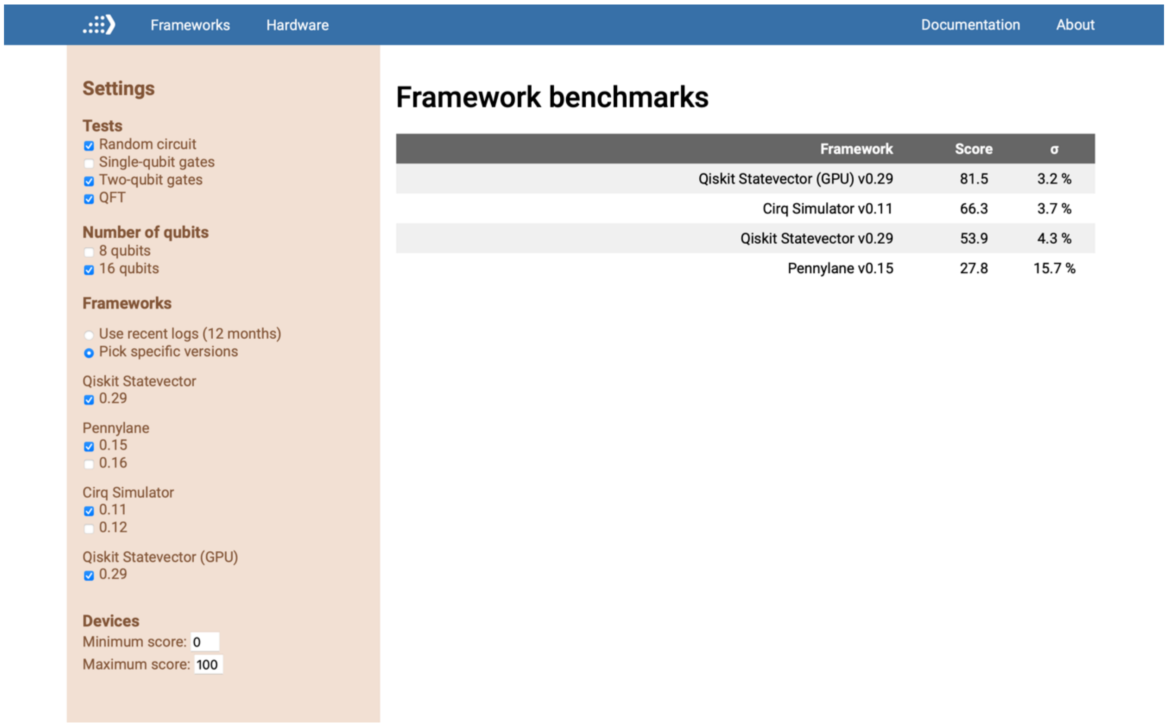
We can also restrict the devices that are used to compute framework scores to be those that have scores within a specific range. This can be useful if we, for instance, want to know how a framework performs with low-end or high-end devices.

5. The web interface

The Python package gives users full control and flexibility over how they want to process the raw data generated by ABAQUS. While this can be an interesting option for some, it is not reasonable to expect any end-user to take the time to download our Python package and learn how to use it. That is why we need to provide a more user-friendly way of accessing ABAQUS benchmark scores. That way is, of course, a web application.

As we mentioned in the beginning, we have developed a Flask web application that is tightly integrated with ABAQUS. The website offers a responsive design that works well on any kind of device, and it allows any user to easily and interactively find benchmark scores for both hardware and software. In order to compute this scores, the website will — upon deployment — get data from a central repository to which any user will be able to contribute.

We will conclude the discussion about this web interface presenting some screenshots.



Settings

Tests
- [✓] Random circuit
- [ ] Single-qubit gates
- [✓] Two-qubit gates
- [✓] QFT

Number of qubits
- [ ] 8 qubits
- [✓] 16 qubits

Frameworks
- ○ Use recent logs (12 months)
- ● Pick specific versions

Qiskit Statevector
- [✓] 0.29

Pennylane
- [✓] 0.15
- [ ] 0.16

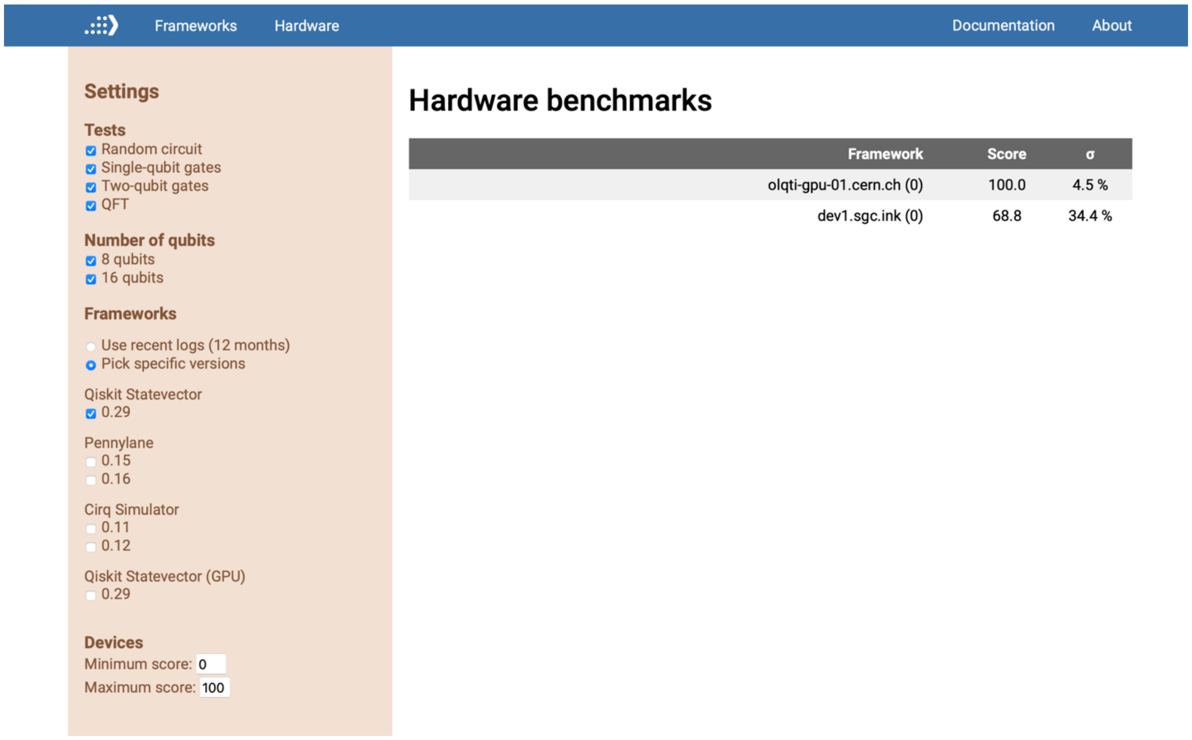
Cirq Simulator
- [✓] 0.11
- [ ] 0.12

Qiskit Statevector (GPU)
- [✓] 0.29

Devices
Minimum score: 0
Maximum score: 100

Framework benchmarks

Framework	Score	σ
Qiskit Statevector (GPU) v0.29	81.5	3.2 %
Cirq Simulator v0.11	66.3	3.7 %
Qiskit Statevector v0.29	53.9	4.3 %
Pennylane v0.15	27.8	15.7 %



Settings

Tests
- [✓] Random circuit
- [✓] Single-qubit gates
- [✓] Two-qubit gates
- [✓] QFT

Number of qubits
- [✓] 8 qubits
- [✓] 16 qubits

Frameworks
- ○ Use recent logs (12 months)
- ● Pick specific versions

Qiskit Statevector
- [✓] 0.29

Pennylane
- [ ] 0.15
- [ ] 0.16

Cirq Simulator
- [ ] 0.11
- [ ] 0.12

Qiskit Statevector (GPU)
- [ ] 0.29

Devices
Minimum score: 0
Maximum score: 100

Hardware benchmarks

Framework	Score	σ
olqti-gpu-01.cern.ch (0)	100.0	4.5 %
dev1.sgc.ink (0)	68.8	34.4 %

6. Conclusions

We have developed ABAQUS: a benchmarking platform for quantum computing simulators which can provide benchmarks for any simulation framework, subject only to the implementation of a simple Python interface. In addition, we have designed ABAQUS in such a way that the suite of benchmarks it performs can be easily extended in a way that preserves consistency across all the different frameworks.

We have also designed a web interface that — being tightly integrated with ABAQUS — allows any end-user to access interactive benchmark scores, for both hardware and software, computed from a data repository.

In conclusion, ABAQUS is a benchmarking tool that lays a foundation to provide anyone with accurate information about the performance of both hardware devices and software frameworks in quantum computing.