StarQ: A Minimal and Accessible Quantum Circuit Representation

A Preprint

Zeeshan Ahmed
International Institute of Information Technology, Hyderabad
zeeshan.ahmed@research.iiit.ac.in

Pulak Malhotra
International Institute of Information Technology, Hyderabad
pulak.malhotra@students.iiit.ac.in

January 21, 2023

Abstract

The number of QASM dialects is growing and the interconversion between these dialects is a hassle. cQASM solves this issue by building upon the dialects of QASM and providing a common interface for all the functionality and representation. cQASM on its own is a very low-level language that is difficult to use, we improve upon the shortcomings of the same and propose a higher-level Lisp-like language for circuit representation with reusable and pattern-preserving properties.

1 Introduction

Quantum algorithms are designed on a higher level rather than in terms of gates. In the current ecosystem of quantum programming, there are plenty of options ranging from quantum SDKs like qiskit[1] to programming languages Q# [2]. Considering representations for quantum circuits, cQASM promises to form a hardware-independent intermediate between higher-level frameworks and other variants of QASM.

1.1 cQASM

It is often the case that algorithm design happens in high-level representation that is then converted to machine-level instructions via multiple compiler passes. common QASM or cQASM is a representation that tries to connect all the dialects of QASM. In doing so it constructs a new dialect that has the functionality of being converted to any other QASM dialect. cQASM also takes the user experience into account and makes it easier to use.

1.1.1 Hardware Independence

The primary goal of cQASM was to eliminate hardware dependency from the quantum assembly programs to make them portable. Rather than compiling the languages directly to QASM, programs that construct circuits for algorithms produce cQASM that remains to be hardware-independent. cQASM code can then be converted by tailoring it to hardware or can be directly simulated. The final assembly is in the form of an executable QASM or eQASM that is able to run on different hardware. Apart from portability, a second advantage of having it hardware independent is that it can directly be simulated on a classical simulator without considering the hardware into account.

1.1.2 Special Features

cQASM supports the "parallel" representation of a moment, i.e. quantum operations applied in parallel on a disjoint set of qubits. The addition of such operations skips steps for the circuit optimizer to reduce the overall circuit depth as a lot of the gates can be "pressed" together. Along the lines of reducing the overall assembly size, the single gate multiple

qubit representations reduce the list of instructions that apply the same operation to multiple qubits to a single assembly instruction with spliced index representation for the quantum registers. An additional feature to further compress the assembly code size, there are static loops integrated into the language that will allow the user to repeat a section of the circuit multiple times.

1.1.3 Shortcomings of cQASM

cQASM has limitations that are specified by the makers themselves [3]. The language currently does not support the re-usage of quantum circuits, which would be crucial to further preserve information about the circuit. Apart from repetitions of the same block of code, it is observed that there are similar patterns that are repeated in different forms which would further simplify the representation. One such example that we will later on see is the concept of un-computation which is often used in the definition of oracle-like sub-circuits.

2 Related Work

The domain of quantum software development kits is rich with frameworks curated for specific tasks. SDKs in the ecosystem rely on the functionality of popular programming languages like python. pyZX [2] is a notable example of a framework that is specific to constructing and optimizing circuits via ZX graphs. openQASM [3] is an assembly that has high-level functionalities like functions and variables. This language still lacks concepts introduced in StarQ such as higher-order functions. The idea of functional quantum programming language is not new, QML [4] uses linear logic and type system for integrating reversible and irreversible computation. Quipper [5] uses QRAM model that also allows it to achieve hardware independence. The aforementioned sources describe higher level languages that do not rely on circuits as the intermediate representation.

3 Language Description

StarQ circuit representation improves upon the work done in cQASM. Being a standalone representation makes it different from other SDKs that rely on a programming language for its functionality. To improve circuit representation, the loss of language features such as libraries do not play a major role. Due to its Lisp-like syntax it becomes very easy to pickup. The higher order functions listed below are not claimed to be complete, new functions can be released in upcoming versions based on requirements.

3.1 Fundamental Gates

Gates and circuits are treated as functions that take numbers as arguments. Table 1 contains the list of quantum operations supported in StarQ, it covers most of the operations implemented in cQASM and has similar naming to ensure consistency. The classical operations are set aside for now and the language focuses on pure quantum circuits only.

3.2 Qubits

The first noticeable difference is the lack of quantum registers. All the qubits are treated as zero-indexed numbers. These numbers get mapped differently based on the way they are invoked. To apply a circuit or gate on a qubit we use the parenthesis notation from Lisp. For example, $(\text{Toffoli } 0\ 1\ 2)$ applies the Toffoli gate with inputs as the zeroth, first and second qubit.

Circuits when defined are accompanied with the size in terms of qubits, this gives user the freedom to make gate application details implicit on some occasions. If the circuit is three qubit gates and the user wants to apply Toffoli on the qubits in order $0, 1, 2$, then the user need not specify this order when adding the gate, this feature can be seen in ccz described in Grover's search at 3.3.

When applying, we preserve the single-gate multi-qubit format that is provided by cQASM, this has been further improved by the all-qubit operation that will treat the application of a single-qubit gate with no arguments as an application over all the qubits in the scope. "(H)" in a 3 qubit circuit expands to a Hadamard gate on each qubit.

The idea of single-gate multi-qubit can be extended to beyond single qubit gates using a zip-like mechanism. We let the user define an implicit zip as arguments for gates. If a single-qubit gate receives a list of qubits, then it applies the gate on all the specified qubits. We extend this logic to multi-qubit gates, an example can be seen in equation 1. This logic can be extended to any number of qubits.

Name	Description	Example
X	Pauli-X	(X 0)
Y	Pauli-Y	(Y 1)
Z	Pauli-Z	(Z 2)
H	Hadamard	(H 3)
I	Identity	(I 5)
S	Phase	(S 2)
Sdag	Phase dagger	(Sdag 1)
T	T	(T 5)
Tdag	T dagger	(Tdag 2)
CNOT	CNOT	(CNOT 0 1)
CZ	CZ	(CZ 0 1)
Rx	Aribitrary x-rotation	(Rx 3.14 0)
Ry	Aribitrary y-rotation	(Rx -3.14 1)
Rz	Aribitrary z-rotation	(Rx 6.28 3)
CR	Controlled Phase Shift (arbitrary angle)	(CR angle 0 1)
CR	Controlled Phase Shift $\pi/2^k$	(CRk k 0 1)
Toffoli	Toffoli	(Toffoli 2 1 0)
measure, measure_z	Measurement in x basis	(measure 2)
measure_x	Measurement in y basis	(measure_x 1)
measure_y	Measurement in z basis	(measure_y 0)

Table 1: Inbuilt gates in StarQ, covering all fundamental quantum gates in cQASM

```
(ccz 3 (H 2) (Toffoli) (H 2))
(hx 3 (H) (X))

(oracle 3 (X 0) (ccz) (X 0))
(diffusion 3 (hx) (ccz) (! (hx)))

(shift 3 (oracle) (diffusion))

(grover 3 (H) (repeat (shift) 2))
```

```
(top-circuit 5
        (H 0)
        (CRk 2 1 0)
        (CRk 3 2 0)
        (CRk 4 3 0)
        (CRk 5 4 0))

(fourier 5
        (top-circuit)
        (-> (% (top-circuit) 1) 1)
        (-> (% (top-circuit) 2) 2)
        (-> (% (top-circuit) 3) 3)
        (-> (% (top-circuit) 4) 4))
```

Listing 1: Grover Search in StarQ on the left and Quantum Fourier Transform on the right

$$(\text{CNOT } (0\ 1\ 2)\ (1\ 2\ 3)) \rightarrow (\text{CNOT } (0\ 1))\ (\text{CNOT } (1\ 2))\ (\text{CNOT } (2\ 3)) \tag{1}$$

3.3 Sub-circuits

Mentioned as one of the drawbacks of cQASM, the addition of reusing circuits would make the circuit representation more readable. The advantage of re-usability is evident in many cases. Even if sub-circuits are going to be used once, the division helps in understanding the overall algorithm as parts of different pieces.

The power of the sub-circuit is very restricted unless it is possible to change the qubits to the set of gates that are being applied. We introduce qubit remapping to further enhance sub-circuit capabilities. In circuit 3.3, we have re-used the bell operation by changing the qubits on which they will be applied. $(\text{bell } 1\ 0)$ indicates the cross mapping between zero and one. This allows the circuit to be defined independent of the physical qubit location.

```
(recover 3 (CNOT 0 2) (CZ 1 2))
(bell 2 (H 0) (CNOT))

(teleportation 3
    (bell 0 2)
    (! (bell 1 0))
    (measure (0 1))
    (recover))
```

Listing 2: Quantum Teleportation in StarQ

3.4 Higher Order Functions

A sub-circuit unchanged on it's own is not very reusable, it is often the case that the circuit is used elsewhere with a minor change, such as the gates being pruned as seen in Quantum Fourier Transform. The functions are designed based on the general pattern that is observed in the circuits.

3.4.1 Repeat

Algorithms such as Grover's search and amplitude amplification work by applying a circuit component numerous times. This is translated to static loops in low level languages such as cQASM. Static loops work remarkably as long as you do not want to denote the inner parts of the circuit that has to be repeated. Grover's search works with oracle and diffusor being applied sequentially numerous times, in this case you will not be able to separate them as 2 labels.

Starq uses a higher order function that accumulates a circuit given number of times. As it can be seen 3.3, Grover's Search is defined to be shift operation applied twice.

3.4.2 Moment

Moment denotes operations happening on different qubits in parallel. We represent it in the language using []. For example, [(H 1) (X 0)] will work, whereas, [(H 0) (X 0)] will throw an error as qubit zero is being used twice in the same time slice. The transpiler checks that this invariant holds even for complex subcircuits inside moments. Keeping simplicity in mind, a moment cannot be nested inside other moments.

3.4.3 Uncompute

Computation on n qubits in circuits pertaining to oracles are often uncomputed with the final value being stored in a single qubit. Uncomputation is the process of applying the hermitian transpose of the operations in reverse order. As seen in line 5 of Grover search, the ! operator is the hermitian conjugate and is used in the definition of the diffusion operator. The operation is recursive, i.e. it will compute the hermitian transpose of a circuit that has recursive sub-circuit definitions.

3.4.4 Chop

As mentioned earlier, there are instances where a quantum circuit needs to be pruned and applied elsewhere, we define chop to be a higher order function that will return a pruned circuit. Rather than working on gate level, this will chop based on the definition of the circuit. For example, if my sub-circuit has another sub-circuit as the last component in the circuit-body definition then chop would remove the entire sub-circuit for argument $>= 1$.

This feature has extensively been used in the intuitive circuit description of the Quantum Fourier Transform. Once the gate sequence for the first qubit is described, we can chop the last gate out from the list and use it for the next qubit. The operation in % in line 10 of 3.3 chops the last gate CRk from the new sub-circuit.

3.4.5 Shift

As seen in Quantum Fourier Transform, it is necessary for a circuit to be shifted entirely to a new set of qubits to make it re-usable. We provide shift that will allow the user to do so. This higher order function can also be treated as syntactic sugar for mapping qubit i to qubit $i + 1$ manually when passing qubit parameter to the circuit. We refer back to the example 3.3 as in the case of chop.

```
qubits 3
{ H q[0] | H q[1] | H q[2] }
.shift(2)
X q[0]
H q[2]
Toffoli q[0], q[1], q[2]
H q[2]
X q[0]
{ H q[0] | H q[1] | H q[2] }
{ X q[0] | X q[1] | X q[2] }
H q[2]
Toffoli q[0], q[1], q[2]
H q[2]
{ X q[0] | X q[1] | X q[2] }
{ H q[0] | H q[1] | H q[2] }
.end
```

```
qubits 5
H q[0]
CRk q[1], q[0], 2
CRk q[2], q[0], 3
CRk q[3], q[0], 4
CRk q[4], q[0], 5
H q[1]
CRk q[2], q[1], 2
CRk q[3], q[1], 3
CRk q[4], q[1], 4
H q[2]
CRk q[3], q[2], 2
CRk q[4], q[2], 3
H q[3]
CRk q[4], q[3], 2
H q[4]
```

Listing 3: cQASM generated by StarQ for Grover Search on the left and Quantum Fourier Transform on the right

3.5 Comments and Case sensitivity

The circuit representation is case-sensitive to maintain a more natural representation in terms of gate names such as H, X, etc. Comments start with the ";" symbol and can be from the end of the line or a new line.

4 Conclusion and Future Work

Inspired by Lisp, we have strayed away from imperative concepts like variables and global state in StarQ. The declarative paradigm encourages us to think of complex circuits as a combination of sub circuits using higher order functions rather than focusing on the gate level mechanics.
We do not claim that this language specification is the ultimate form. StarQ can be expanded in many ways. To continue the standardization of implementation and optimization of quantum circuits, inventing new higher order generics play a key role. We believe StarQ provides a sound direction and base for future developments in quantum circuit design.

References

[1] Qiskit: An open-source framework for quantum computing, 2021.

[2] Microsoft. *Q# Language Specification*, 2020.

[3] Nader Khammassi, Gian Giacomo Guerreschi, Imran Ashraf, Justin Hogaboam, Carmen Garcia Almudever, and Koen Bertels. cqasm v1.0: Towards a common quantum assembly language. *arXiv: Quantum Physics*, 2018.

[4] T. Altenkirch and J. Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. IEEE, 2004.

[5] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 333–342, New York, NY, USA, 2013. Association for Computing Machinery.