

Abstract Resource Cost Derivation for Logical Quantum Circuit Descriptions

Andrei Lapets

Raytheon BBN Technologies, 10 Moulton Street,
Cambridge, MA 02138, USA
alapets@bbn.com

Martin Rötteler

NEC Laboratories America, 4 Independence Way,
Suite 200, Princeton, NJ 08540, USA
mroetteler@nec-labs.com

Abstract

Resources that are necessary to operate a quantum computer (such as qubits) have significant costs. Thus, there is interest in finding ways to determine these costs for both existing and novel quantum algorithms. Information about these costs (and how they might vary under multiple parameters and circumstances) can then be used to navigate trade-offs and make optimizations within an algorithm implementation. We present a domain-specific language called QuIGL for describing logical quantum circuits; the QuIGL language has specialized features supporting the explicit annotation and automatic derivation of descriptions of the resource costs associated with each logical quantum circuit description (as well as any of its component procedures). We also present a formal framework for defining abstract transformations from QuIGL circuit descriptions into labelled, parameterized quantity expressions that can be used to compute exact counts or estimates of the cost of the circuit along chosen cost dimensions and for given input sizes. We demonstrate how this framework can be instantiated for calculating costs along specific dimensions (such as the number of qubits or the T -depth of a logical quantum circuit).

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—specialized application languages; D.3.3 [Programming Languages]: Language Constructs and Features—frameworks

Keywords domain-specific languages; quantum programming languages; abstract interpretation

1. Introduction

Logical quantum circuits are a standard representation of algorithms that can be executed on quantum computers; QuIGL is a domain-specific language for describing logical quantum circuits. While the QuIGL language can be used as a user-facing low-level quantum programming language, it is primarily designed to be an intermediate representation for a compiler for a high-level quantum programming language. A QuIGL program can be viewed as a concise description of a circuit that can be expanded into an explicit

logical quantum circuit. The definition of QuIGL is biased towards fewer language features (*i.e.*, it is not a *high-level* or *feature-rich* language) while maintaining the goal of being sufficiently *expressive* to describe all possible logical quantum circuits. A QuIGL circuit description (as well as all component procedures within the description) can be accompanied by parameterized annotations that describe the circuit along other dimensions of interest that represent resource quantities, such as the total number of qubits or the circuit depth. These annotations may represent exact counts or estimates.

Background and Motivation. A language for describing logical quantum circuits must support some collection of logical quantum gates (*i.e.*, primitive unitary transformations of the quantum state). Many logical gate sets are known that afford universal fault-tolerant quantum computation [17]. QuIGL supports a collection of several quantum gates, and this collection is a superset of the gate set that contains all Clifford gate together with the so-called T -gate, which is the diagonal unitary matrix $\text{diag}(1, e^{i\pi/4}) \in \mathbb{C}^{2 \times 2}$. The reason to support at least this gate set is that it arises canonically for several families of quantum codes [13], in particular the surface code [6] and several concatenated codes [5, 18], making it one of the most studied universal gate sets in fault-tolerant quantum computation.

It should be noted that there is a quite skewed cost metric associated with this gate set: the cost of T -gates is significantly higher than the cost for all other gates, dominating all other gates by a huge margin. This is due to the fact that T -gates are typically implemented by preparing suitable states using a distillation procedure [4, 15] and are then teleported into the circuit [13]. The distillation procedure is very costly in terms of hardware resources (*i.e.*, time and number of physical qubits). For instance in case of the surface code, it is reasonable to assume that the cost of a single T -gate is 100 times that of a single CNOT gate.

In order to minimize the resources required for a quantum computation it is therefore imperative to minimize: (1) total number of T -gates, (2) the T -depth, and (3) the width of the circuit (measured in qubits). Minimizing the total number of T -gates is crucial as this number determines the size of the ancilla state factory used to implement the logical T -gates. More specifically, the number of T -gates drives the desired precision which dictates the required number of state distillation rounds to prepare the ancilla states and, consequently, the physical space and time of the process. Next, minimizing the total T -depth of circuit is important as the depth is related to the overall physical runtime. Finally, minimizing the width of the circuit, *i.e.*, the number of qubits which are kept alive simultaneously, is essential in controlling resource costs.

While it is desirable to minimize cost along all of these dimensions, complications arise due to the fact that the actual resources can be traded off against each other and it is often not entirely clear which choice leads to the best possible physical metrics. This is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FPCDSL '13, September 22, 2013, Boston, MA, USA.
Copyright © 2013 ACM 978-1-4503-2380-2/13/09...\$15.00.
<http://dx.doi.org/10.1145/2505351.2505358>

further complicated by the fact that not all logical qubits are equal: some qubits are only needed for a short period and are then released, not all time steps are equal, and some time steps may require more T -gates than others. Thus, it is important to be able to express and calculate these costs so that optimization techniques (e.g., as performed by circuit synthesis tools) are applied appropriately to minimize the quantum resources as desired.

The purpose of the QuIGL circuit description language is to support resource-aware assembly and representation of logical quantum circuit descriptions while letting users employ common programming language constructs such as procedures, control-flow constructs, and parametrization. QuIGL achieves this by furnishing programmers with both these common constructs and a specialized annotation language for abstractly describing resource quantities along dimensions of interest (e.g., T -depth). Together, the circuit description and resource quantity languages form a framework for describing multiple, possibly interdependent resource counting or estimation algorithms for logical quantum circuit descriptions: such algorithms can be formally defined (and implemented) as transformations from circuit descriptions to resource quantity annotations.

Organization. The rest of this paper is organized as follows. Section 2 describes other quantum programming and circuit description languages. Section 3 introduces the QuIGL concrete and abstract syntax definitions, provides a few illustrative examples, and describes some of the QuIGL language constructs in more detail. Section 4 presents a framework for defining interdependent abstract interpretations of QuIGL circuit descriptions along multiple resource quantity dimensions. Finally, Section 5 concludes and discusses future directions for this work.

2. Related Work

There exists earlier and ongoing work on quantum programming languages [3, 7, 8, 14, 16]. These languages also allow users to explicitly describe quantum circuits or linear transformations of quantum states. Some of these languages are high-level and may not be appropriate as an intermediate representation within a quantum programming infrastructure. However, they may be amenable to resource cost analysis techniques such as those described in this paper, either directly at a high level or by means of compilation to an intermediate representation such as QuIGL. Quipper [9] is an embedded quantum programming language that is also designed to support analysis of resource costs for quantum algorithms. Unlike Quipper, QuIGL is not an embedded language. While QuIGL provides some familiar programming language control flow and encapsulation constructs (mainly to support assembly and representation of low-level, *concise* circuit descriptions), and these can aid in quantum resource estimation for user-defined circuits, it is primarily designed to be a low-level intermediate representation that can be used to estimate resource costs for compiler-generated circuits.

Other low-level logical quantum circuit description languages are also in use, including QASM [1] and the .qc format used by QCViewer [2]. However, these are less abstract than QuIGL in that they do not allow parametrization of circuit description constructs (such as procedures and blocks that might repeat sequentially or in parallel) using natural number parameters. Thus, QuIGL circuit descriptions can often retain more of the structure that might be found in a high-level description of an algorithm, and can also be more concise. Because circuit descriptions in formats such as QASM and .qc are low-level, cost calculation often amounts to simple counting. QuIGL is distinct in that its more abstract circuit descriptions, even when they are parameterized by natural number parameters that have not been instantiated, can still be abstractly interpreted as mathematical expressions that can then be evaluated to obtain resource costs without assembling the actual circuit.

3. QuIGL Syntax and Language Constructs

3.1 Concrete and Abstract Syntax

The abstract syntax of the QuIGL circuit description language is presented in Table 1. We also present some simple examples of QuIGL circuit descriptions to illustrate the concrete syntax.

Examples. Figure 1 presents a QuIGL circuit description corresponding to an implementation of Grover iteration (circuit shown in Figure 2). Figure 3 presents a QuIGL circuit description corresponding to an implementation of the Quantum Fourier transform.

```

procedure mapH [n] (q) {
  parallel [x] [n] { H (q[x]) };
};

procedure groverIterate [] (q, work) {
  H (work);
  control (q[0], q[1], q[4]) {T (q[2]); T (q[3]);};
  H (work);
  call mapH [5] (q);
  pauliX (q[4]);
  control (q[0], q[1], q[2], q[3]) {pauliZ (q[4]);};
  pauliX (q[4]);
  call mapH [5] (q);
};

q := wire [5];
work := wire [1];
call mapH [5] (q);
repeat [j] [4] {
  call groverIterate [] (q, work);
};

```

Figure 1. QuIGL implementation of Grover iteration.

```

procedure qft [n] (q) {
  H (q[0]);
  repeat [j] [n - 2] {
    repeat [k] [j] {
      phaseZ [pi * (1/2^(1+j-k))] (q[k]);
    };
    H (q[j + 1]);
  };
};

q := wire [5];
call qft [5] (q);

```

Figure 3. A QuIGL implementation of the Quantum Fourier transform.

3.2 Language Constructs

Quantity expressions. Quantity expressions (as defined in Table 1) describe classical computations that can be evaluated (when all variables are instantiated) to numerical values. They do not represent data that is stored inside qubits; thus, they can be viewed as static expressions to be evaluated at the time of static analysis, compilation, or optimization. Within QuIGL statements, quantity expressions are always delimited by square brackets [...] (e.g., when specifying a qubit within a data structure by its index, or when specifying static arguments in a procedure call).

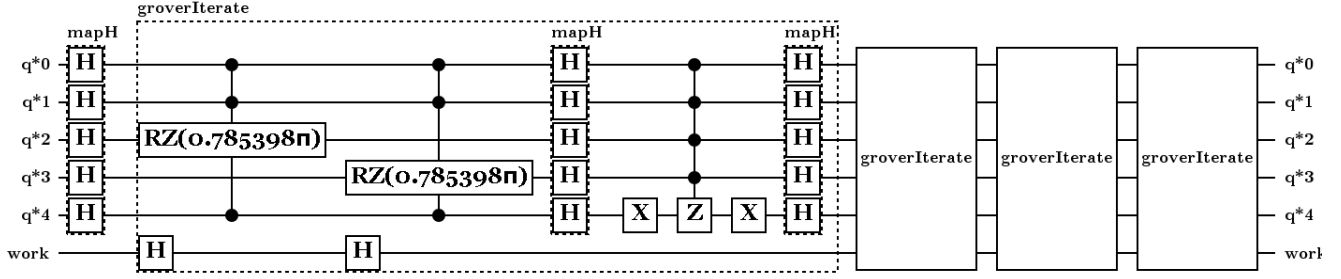


Figure 2. QCViewer [2] visualization of Grover iteration definition in Figure 1.

natural	n	\in	\mathbb{N}
variable	x, y, f	\in	Variable
resource dimension	d	\in	Dimension
quantity	q	$::=$	$n \mid x$ $\mid q_1 + q_2 \mid q_1 - q_2 \mid q_1 * q_2 \mid q_1 / q_2 \mid q_1 \wedge q_2$ $\mid \max(q_1, q_2) \mid \sum_{x=0}^{q_0} \{q_1\} \mid \max_{x=0}^{q_0} \{q_1\}$ $\mid f_d(q_1, \dots, q_n)$ $\mid (q)$
quangle	θ	$::=$	$\text{pi} * q$
quregion	r	$::=$	$x \mid x[q_1, \dots, q_n] \mid (x_1, \dots, x_n) \mid \text{wire}[q_1, \dots, q_n]$
gates	g	$::=$	$\text{I} \mid \text{H} \mid \text{T} \mid \text{T}^* \mid \text{phase} \mid \text{pauliX} \mid \text{pauliY} \mid \text{pauliZ}$ $\mid \text{phaseX} \mid \text{phaseY} \mid \text{phaseZ} \mid \text{NOT} \mid \text{CNOT} \mid \text{CSWAP} \mid \text{TOF}$
parameter	p	$::=$	$\theta \mid q$
resource annotation	a	$::=$	$\langle d_1 : q_1 ; \dots ; d_n : q_n \rangle \mid \langle \rangle$
statement	s	$::=$	$x := r$ $\mid (x_1, \dots, x_n) := r$ $\mid g[p_1, \dots, p_n](r_1, \dots, r_n)$ $\mid \text{control}(r_1, \dots, r_n) b$ $\mid \text{control not}(r_1, \dots, r_n) b$ $\mid \text{repeat}[x][q] b$ $\mid \text{parallel}[x][q] b$ $\mid \text{parallel } b_1 \dots b_n$ $\mid \text{with } b_1 \text{ do } b_2$ $\mid \text{call } f[p_1, \dots, p_n](r_1, \dots, r_n)$
statement block	b	$::=$	$\{s_1 ; \dots ; s_n\}$
top-level statement	t	$::=$	s $\mid \text{procedure } f[x_1, \dots, x_n] a(y_1, \dots, y_n) b$
circuit	c	$::=$	$\text{circuit } a\{t_1 ; \dots ; t_n\}$

Table 1. Definition of the QuIGL abstract syntax.

Qubits and Quregions. In a QuIGL circuit description, it is possible to allocate and refer to lists, or *quregions*, of qubits (denoted using a variable, a variable followed by a bracketed list of indices, or the `wire` construct) and assign them to *dynamic* variables that represent quregions within circuit descriptions. It is also possible to allocate and utilize a finite collection of quregions; this is denoted using the tuple notation. Thus, each variable represents a collection of one or more quregions, each consisting of one or more qubits.

Note that when multiple quregions are passed as an argument to a procedure (or are assigned to a tuple of variables representing a collection of quregions¹) they are treated semantically as a single quregion. However, the actual assignment statement or procedure definition may break this collection up. For example:

```
procedure example [] (a,b,c) { /* ... */ };
q := wire [3];

// The following are equivalent.
call example [] (q);
call example [] (q[0], q[1], q[2]);

// The following are equivalent.
(a,b,c) := q
(a,b,c) := (q[0], q[1], q[2]);
```

Variable scope. The scope of a variable (either bound through an assignment statement or defined as a parameter in a procedure definition) is the next closing delimiter “}”, excepting the bodies of all procedure declarations that define a parameter with the same name as the variable.

Gates. QuIGL has built-in syntax for the identity, Hadamard, T , T^{-1} , phase, Pauli, NOT, CNOT, CSWAP, and Toffoli gates. Figure 4 illustrates the concrete syntax for these (circuit shown in Figure 5).

Control structures. QuIGL supports several control structures for describing circuits. The purpose of these constructs is to allow more concise circuits descriptions.

The `control` construct indicates that the entire circuit description within its delimited block is controlled by the specified quregion (under the `control not` variant, the delimited block is controlled by the logical negation of all the qubits in the specified quregion). The `control` constructs can be nested. The `repeat` construct indicates that the entire circuit description within its delimited block should be repeated the specified number of times. An iteration counter must be named, and its scope is the circuit description within the delimited block.

```
q := wire [1];
repeat [j] [4] { // The range of j is 0,1,2,3.
  T (q);        // Apply T to qubit q four times.
};
```

The `parallel` construct indicates that the entire circuit description within its delimited block should be applied in parallel over the range specified for the counter.

```
q := wire [4];
parallel [j] [4] { // The range of j is 0,1,2,3.
  T (q[j]);      // Apply T to each qubit in q.
};
```

¹ Only tuples of *variables* appear on the left-hand side of an assignment in Table 1 because QuIGL is designed to be an intermediate representation; more complex operations can be built up using sequences of statements.

```
p := wire [1];
q := wire [1];
r := wire [1];

I (p);          // Apply an identity gate to p.
H (p);          // Apply a Hadamard gate to p.
T (p);          // Apply a T gate to p.
T* (p);         // Apply an inverse T gate to p.
phase (p);      // Apply a P (a.k.a. S) gate to p.
pauliX (q);     // Apply a Pauli X gate to q.
pauliY (q);     // Apply a Pauli Y gate to q.
pauliZ (q);     // Apply a Pauli Z gate to q.

// Apply phase X gate with angle pi*1/16 to r.
phaseX [pi*1/16] (r);

// Apply phase Y gate with angle pi*1/32 to r.
phaseY [pi*1/32] (r);

// Apply phase Z gate with angle pi*1/64 to r.
phaseZ [pi*1/64] (r);

// Apply NOT gate to q.
NOT (q);

// Apply NOT gate to q controlled by p.
CNOT (p,q);

// Apply SWAP gate to q and r controlled by p.
CSWAP (p,q,r);

// Apply Toffoli (NOT gate on q controlled by p, r).
TOF (p,r,q);
```

Figure 4. QuIGL syntax for gates (see corresponding QCViewer [2] visualization in Figure 5).

Note that the `repeat` construct may be equivalent to the `parallel` construct (*i.e.*, they can be interpreted as the same circuit) if there is no sequential dependency between each iteration. The `repeat` example below is equivalent to the `parallel` example above.

```
q := wire [4];
repeat [j] [4] { // The range of j is 0,1,2,3.
  T (q[j]);      // Apply T to each qubit.
};
```

The two constructs can be combined to create various structures within circuits.

```
q := wire [4];
repeat [j] [4] {
  parallel [k] [j] { T (q[k]); };
};
```

The `with ... do` construct makes it possible to apply a circuit description and its inverse.

```
q := wire [1];
with {
  pauliX (q); // Applied before and after do block.
} do {
  T (q);
};
```

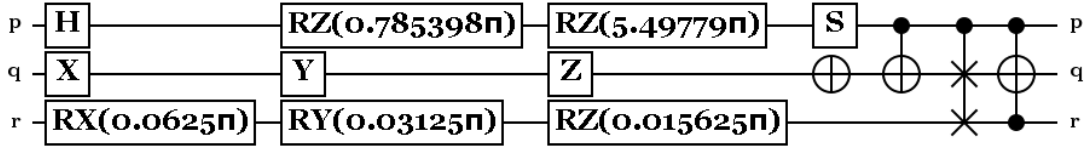


Figure 5. QCViewer [2] visualization for Figure 4.

Because the `with { ... }` portion must be invertible, it cannot contain any statements that allocate new qubits.

Procedure definition and invocation. It is possible to define procedures and to invoke them. A procedure can have both static and dynamic arguments.

```
procedure example [a, b] (p, q) {
  repeat [j] [a] {
    T (q[j]); T (q[b]);
  };
};
```

4. Circuit Interpretations along Multiple Resource Quantity Dimensions

As defined in Table 1, QuIGL procedures, as well as the entire circuit description, can be annotated with multiple parameterized quantity expressions (one for each resource quantity dimension that might be of interest). These annotations can be supplied by the user for each procedure, as well as for the entire circuit (*e.g.*, as temporary placeholders indicating the target cost of an unoptimized circuit). Each expression is preceded by a dimension label indicating the dimension of the quantity to compute. For example:

```
procedure example [n, m] < #T: 1 + n * m > (q) {
  T (q[0]);
  repeat [j] [n] {
    repeat [k] [m] {
      T (q[j, k]);
    };
  };
};
```

Just as procedures can invoke other procedures defined elsewhere, quantity expression can contain invocations of: (1) quantity expressions for the same procedure, but a different dimension, and (2) resource quantity annotations of another named procedure. For example, suppose we want to count both the number of controlled NOT gates and the number of NOT gates *without* control qubits:

```
procedure g [n] < #NOT: n; #CNOT: 0 > (q) {
  repeat [j] [n] { NOT (q); };
};

procedure f [n] <
  #NOT: 1; #CNOT: (n-1) * g #NOT (n)
> (q, r) {
  NOT (r);
  control (r) {
    repeat [j] [n-1] {
      call g [n] (q);
    };
  };
};
```

While users are allowed to supply their own quantity annotations (as in the above examples), this is mostly intended for situations in which a procedure might be a placeholder that is an approximation, an incomplete implementation, or a stub (*i.e.*, a missing implementation). This allows users to document the cost of a procedure before it has been implemented, or to document necessary or desirable bounds for an implementation of a circuit or procedure.

Automated derivation. Resource quantity annotations for procedures and circuits can be derived automatically, assuming that a transformation from QuIGL statements to quantity expressions has been defined.² Let \mathcal{C} denote the set of QuIGL circuit descriptions, let \mathcal{S} denote the set of QuIGL statements and statement blocks, and let \mathcal{Q} denote the set of quantity expressions (corresponding to the sets of abstract syntax trees $\{c \dots\}$, $\{s \dots\} \sqcup \{b \dots\} \sqcup \{t \dots\}$, and $\{q \dots\}$, respectively, as defined in Table 1).

For a given resource quantity dimension $d \in \text{Dimension}$, we denote a transformation from the set of QuIGL statements \mathcal{S} to the set of quantity expressions \mathcal{Q} using the notation $|\bullet|_d : \mathcal{S} \rightarrow \mathcal{Q}$, where for any $s \in \mathcal{S}$, $|s|_d \in \mathcal{Q}$. Notice that these transformations are *abstract*: they transform an abstract syntax tree corresponding to a QuIGL statement into another abstract syntax tree corresponding to a quantity expression. Furthermore, these trees are themselves abstract, since they may contain procedures, variables, and other constructs, and may be evaluated (statements can be evaluated to logical quantum circuits, and quantity expressions can be evaluated to numerical values).

Given a collection $D \subset \text{Dimension}$ of resource quantity dimensions where $D = \{d_1, \dots, d_n\}$, we define an *ensemble* of transformations for a set of dimensions D as a collection of transformations $\alpha = \{|\bullet|_{d_1}, \dots, |\bullet|_{d_n}\}$. An ensemble α for D is *closed* if for all $s \in \mathcal{S}$, $d \in D$, and $|\bullet|_d \in \alpha$, the quantity expression $|s|_d$ only references dimensions in D .

Given an ensemble α for D and a circuit $c \in \mathcal{C}$, we define the annotated circuit $\alpha(c) \in \mathcal{C}$, in which all the procedure definitions, as well as the circuit itself, are annotated with quantity expressions returned by the transformations in α (explicit annotations provided by the user are not replaced). That is, every procedure definition of the form

```
procedure f [x1, ..., xn] <> (...) b
```

in c is replaced with a procedure definition of the form

```
procedure f [x1, ..., xn] <d1 : |b|d1; ...; dn : |b|dn> (...) b
```

and the circuit itself, if it is of the form

```
circuit <> {t1; ...; tn}
```

is replaced with

```
circuit <d1 : |b'|d1; ...; dn : |b'|dn> {t'1; ...; t'n}
```

where $b' = \{t'_1; \dots; t'_n\}$ and the t'_i are top-level statements that have already had the annotation transformation applied to them.

²Currently, programmers cannot define their own dimension labels and corresponding transformations within QuIGL itself.

QuIGL statement s	$ s _{\#H}$	$ s _{\#NOT}$	$ s _{\#CNOT}$	$ s _{\#T}$	$ s _{dT}$
H $[] (q)$	1	0			
T $[] (q)$	0	0		1	
T* $[] (q)$	0	0		1	
pauliX $[] (q)$	0	1	0		
NOT $[] (q)$	0	1	0		
CNOT $[] (q, q')$	0		1	0	
control $(q_1, \dots, q_n) \ b$	$ b _{\#H}$	0	$ b _{\#NOT} + b _{\#CNOT}$	$ b _{\#T}$	$ b _{dT}$
control not $(q_1, \dots, q_n) \ b$	$ b _{\#H}$	0	$ b _{\#NOT} + b _{\#CNOT}$	$ b _{\#T}$	$ b _{dT}$
repeat $[x] \ [e] \ b$	$\text{sum}_{x=0}^{e-1} \{ b _d\}$ for corresponding d				
parallel $[x] \ [e] \ b$	$\text{sum}_{x=0}^{e-1} \{ b _d\}$ for corresponding d				$\max_{x=0}^{e-1} \{ b _{dT}\}$
parallel $b_1 \dots b_n$	$\text{sum}_{i=1}^n \{ b_i _d\}$ for corresponding d				$\max_{i=1}^n \{ b_i _{dT}\}$
with b_1 do b_2	$2 \cdot b_1 _d + b_2 _d$ for corresponding d				
call $f \ [e_1, \dots, e_n] \dots$	$f_d(e_1, \dots, e_n)$ for corresponding d				
$\{ \ s_1; \dots; s_n \ ; \}$	$\text{sum}_{i=1}^n \{ s_i _d\}$ for corresponding d				
<i>all others</i>	0				

Table 2. QuIGL circuit interpretations along simple gate count and depth quantity dimensions: number of Hadamard gates (#H), number of NOT gates without control qubits (#NOT), number of controlled NOT gates (#CNOT), number of T -gates including inverses (#T), and the T -gate depth (dT).

QuIGL statement s	$ s _{Dq}$	$ s _{Aq}$	$ s _{Pq}$
$x := \text{wire } [q_1, \dots, q_n]$	$e_1 \cdot \dots \cdot e_n$		0
control (q_1, \dots, q_n) b	$ b _d$ for corresponding d		
control not (q_1, \dots, q_n) b	$ b _d$ for corresponding d		
repeat [x] [e] b	$\sum_{x=0}^{e-1} \{ b _{Aq}\} + \max_{x=0}^{e-1} \{ b _{Pq}\}$	$\sum_{x=0}^{e-1} \{ b _{Aq}\}$	$\max_{x=0}^{e-1} \{ b _{Pq}\}$
parallel [x] [e] b	$\sum_{x=0}^{e-1} \{ b _{Dq}\}$	$\sum_{x=0}^{e-1} \{ b _{Aq}\}$	$\sum_{x=0}^{e-1} \{ b _{Pq}\}$
parallel $b_1 \dots b_n$	$\sum_{i=1}^n \{ b_i _{Dq}\}$	$\sum_{i=1}^n \{ b_i _{Aq}\}$	$\sum_{i=1}^n \{ b_i _{Pq}\}$
with b_1 do b_2	$ b_2 _{Dq}$	$ b_2 _{Aq}$	$ b_2 _{Pq}$
call f [e_1, \dots, e_n] ...	$f_{Dq}(e_1, \dots, e_n)$	0	$f_{Pq}(e_1, \dots, e_n)$
{ $s_1; \dots; s_n$; }	$\sum_{i=1}^n \{ s_i _{Aq}\} + \max_{i=1}^n \{ s_i _{Pq}\}$	$\sum_{i=1}^n \{ s_i _{Aq}\}$	$\max_{i=1}^n \{ s_i _{Pq}\}$
<i>all others</i>	0		

Table 3. QuIGL circuit interpretations along a standard qubit quantity dimensions: total number of qubits allocated (Dq), number of new qubits or ancillae allocated by assignment statements (Aq), and the number of new qubits allocated within invoked procedures (Pq).

Note that given a procedure definition of the following form:

procedure f $[x_1, \dots, x_k]$ $\langle a \rangle$ (\dots) b ;

the quantity expression $f_d(q_1, \dots, q_k)$ can be evaluated as $|b|_d$ wherever it appears (*i.e.*, in procedure or circuit annotations). In other words, if $\sigma \in \text{Variable} \rightarrow \mathbb{N}$ were used to denote environments that map variables to numeric quantity values, and $\sigma \vdash q \Downarrow n$ denoted that $q \in \mathcal{Q}$ evaluates to $n \in \mathbb{N}$ under σ , then the evaluation rule for named procedure invocations within quantity expressions would be:

$$\frac{\sigma \vdash q_1 \Downarrow n_1 \quad \dots \quad \sigma \vdash q_k \Downarrow n_k \quad \sigma \uplus \{x_1 \mapsto n_1, \dots, x_k \mapsto n_k\} \vdash |b|_d \Downarrow n}{\sigma \vdash f_d(q_1, \dots, q_k) \Downarrow n}$$

Instantiation. Tables 2 and 3 present a collection of relevant resource dimensions (the importance of some of these dimensions is discussed and motivated in Section 1) along which circuit descriptions can be interpreted: number of Hadamard gates (#H), number of NOT gates without control qubits (#NOT), number of controlled NOT gates (#CNOT), number of T -gates including inverses (#T), the T -gate depth (dT), the total number of qubits allocated (Dq), the number of new qubits or ancillae allocated by assignment statements (Aq), and the number of new qubits allocated within invoked procedures (Pq). Each column in Tables 2 and 3 recursively defines the transformation $|\bullet|_d$ for a particular dimension $d \in \{\#H, \#NOT, \#CNOT, \#T, dT, Dq, Aq, Pq\}$: each column corresponds to a dimension; each row corresponds to a QuIGL statement (including top-level statements and blocks); and each entry corresponds to the definition of $|s|_d$ given that particular dimension and statement combination. Thus, the tables collectively define a closed ensemble of transformations that can be applied to any QuIGL circuit $c \in \mathcal{C}$.

Note that the operations used within the expressions in each of the table entries *do not* represent operations on natural numbers; they represent the node constructors that should be used to build the abstract syntax tree representing that expression. Also note how this particular ensemble of transformations takes advantage of the interdependencies between annotations along different dimensions. For example, the number of NOT gates under a control construct is counted towards the total number of CNOT gates. Also, qubits allocated under different conditions are tracked along different dimensions (Dq, Aq, and Pq) in order to reflect how allocated qubits can be reused within the circuit.

In particular, ancilla qubits allocated within a procedure can be reused after the procedure has been invoked, which means that for any sequence of statements, it is sufficient to take the maximum over all procedure invocations in the sequence:

$$\left| \{s_1; \dots; s_n\} \right|_{Pq} = \max_{i=1}^n \{|s_i|_{Pq}\}$$

On the other hand, if any allocations occur when procedures are applied within parallel blocks, it is necessary to add all the allocations:

$$\left| \text{parallel } b_1 \dots b_n ; \right|_{Pq} = \sum_{i=1}^n \{|b_i|_{Pq}\}$$

Furthermore, explicit allocations that employ the assignment statement are treated differently: it is assumed all assignment statements that employ the **wire** construct within a scope allocate new qubits (unlike qubits allocated within invoked procedures, which may be reused). Thus, in Table 3 the quantum expression for the dimension Dq that measures the *total* number of qubits allocated is as follows for the case of a statement block:

$$\left| \{s_1; \dots; s_n\} \right|_{Dq} = \sum_{i=1}^n \{|s_i|_{Aq}\} + \max_{i=1}^n \{|s_i|_{Pq}\}$$

In other words, the total number of qubits allocated is the sum of the new qubits allocated plus the maximum number of qubits allocated by any procedure (or set of parallel procedures) that may have been invoked within the sequence. Since no allocations can occur inside **with** $\{ \dots \}$ blocks, the definitions for those cases are simple:

$$\begin{aligned} \left| \text{with } b_1 \text{ do } b_2 ; \right|_{Dq} &= |b_2|_{Dq} \\ \left| \text{with } b_1 \text{ do } b_2 ; \right|_{Aq} &= |b_2|_{Aq} \\ \left| \text{with } b_1 \text{ do } b_2 ; \right|_{Pq} &= |b_2|_{Pq} \end{aligned}$$

For another example, we can note the distinction between the *count* of T -gates #T and the T -gate *depth* dT in the case of parallel blocks:

$$\begin{aligned} \left| \text{parallel } b_1 \dots b_n ; \right|_{\#T} &= \sum_{i=1}^n \{|b_i|_{\#T}\} \\ \left| \text{parallel } b_1 \dots b_n ; \right|_{dT} &= \max_{i=1}^n \{|b_i|_{dT}\} \end{aligned}$$

All blocks contribute to the overall number of T -gates; however, if blocks are parallel, only the block with the greatest depth contributes to the overall depth.

Figure 6 illustrates the results of applying the automated annotation procedure defined in Tables 2 and 3 on the example circuit description in Figure 1. Note that, as seen in this example output, some algebraic simplification of quantity expressions can take place within an implementation of the transformation procedure.

Implementation. A programming environment supporting the use of QuIGL both as a user-facing programming language and as an intermediate representation, including a parser, data structures and XML schemas, and the automated derivation capabilities for resource quantity annotations along the dimensions presented in Tables 2 and 3, has been implemented as part of a more extensive quantum programming tool chain. Translations in both directions between the .qc format [2] and QuIGL have also been implemented.

One important observation about Tables 2 and 3 is that for any circuit that contains phase gates with non-trivial angles, the computed resource quantity might be an under-estimate because implementing phase gates with non-trivial angles may require the use of multiple T -gates. In order for this instantiation of the framework to act as an exact resource count rather than an estimate, it is necessary to first replace all phase gates within the circuit with non-trivial angles with sequences of Clifford and T -gates. Within the quantum programming tool chain of which QuIGL is one component, this substitution is accomplished using other tools [10, 11].

5. Conclusion and Future Work

We have presented QuIGL, a circuit description language that support explicit and automated annotation of circuit descriptions with resource cost descriptions, and an accompanying framework for defining resource cost calculation algorithms. Possible future directions for this work include the instantiation of the framework with additional dimensions of interest (such as the number of non-trivial rotations, or more refined costs related to actual physical parameters [12]), as well as the definition of optimization algorithms on QuIGL circuit descriptions that optimize specific quantity dimensions. It is also possible to use user-specified resource annotations as *constraints*, in the tradition of type systems that allow user-specified type annotations or contracts, on implementations of procedures and circuits (such a scheme can also be seen as a validation mechanism for user-supplied annotation).

In the context of more extensive quantum programming environments or tool chains, it may be possible to provide software support for exploring trade-offs between different optimizations, and to

```

circuit <
  #H: mapH #H (5) + 4 * groverIterate #H ();
  #NOT: mapH #NOT (5) + 4 * groverIterate #NOT ();
  #CNOT: mapH #CNOT (5) + 4 * groverIterate #CNOT ();
  #T: mapH #T (5) + 4 * groverIterate #T ();
  dT: mapH dT (5) + 4 * groverIterate dT ();
  Dq: 5 + 1 + mapH Dq (5) + 4 * groverIterate Dq ();
  Aq: 5 + 1;
  Pq: mapH Pq (5) + 4 * groverIterate Pq ();
> {

  procedure mapH [n] <
    #H: n;
    #NOT: 0;
    #CNOT: 0;
    #T: 0;
    dT: 0;
    Dq: 0;
    Aq: 0;
    Pq: 0
  > (q) {
    parallel [x] [n] { H (q[x]) };
  };

  procedure groverIterate [] <
    #H: 1 + 1 + mapH #H (5) + mapH #H (5);
    #NOT: 1 + 1 + mapH #NOT (5) + mapH #NOT (5);
    #CNOT: mapH #CNOT (5) + mapH #CNOT (5);
    #T: 1 + 1 + mapH #T (5) + mapH #T (5);
    dT: 1 + mapH dT (5) + mapH dT (5);
    Dq: mapH Dq (5) + mapH Dq (5);
    Aq: 0;
    Pq: mapH Pq (5) + mapH Pq (5)
  > (q, work) {
    H (work);
    control (q[0], q[1], q[4]) {
      T (q[2]);
      T (q[3]);
    };
    H (work);
    call mapH [5] (q);
    pauliX (q[4]);
    control (q[0], q[1], q[2], q[3]) {
      pauliZ (q[4]);
    };
    pauliX (q[4]);
    call mapH [5] (q);
  };

  q := wire [5];

  work := wire [1];
  call mapH [5] (q);
  repeat [j] [4] {
    call groverIterate [] (q, work);
  };
}

```

Figure 6. QuIGL implementation of Grover iteration with automatically derived resource quantity annotations (full circuit description is listed).

characterize the optimizations according to which dimensions they favor. It is also possible to incorporate QuIGL as an intermediate representation within compilers for other, more high-level quantum programming languages. Considering a broader scope, it is likely

the technique of deriving quantity expressions from programs along multiple resource cost dimensions, which is arguably an example of applying ensembles of abstract interpretations to programs, can be applied in other application domains outside of quantum programming, *e.g.*, to predict energy consumption in classical computers.

Acknowledgments

Supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center Contract number DIIIPC20166. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC or the U.S. Government.

References

- [1] QASM. <http://www.media.mit.edu/quanta/qasm2cirq/>.
- [2] QCViewer: a tool for displaying, editing, and simulating quantum circuits, 2012. <http://qcirc.iqc.uwaterloo.ca/>.
- [3] T. Altenkirch and J. J. Grattage. A functional quantum programming language. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science, LICS 2005*, pages 249–258. IEEE Computer Science Press, 2005.
- [4] S. Bravyi and A. Kitaev. Universal quantum computation with ideal Clifford gates and noisy ancillas. *Phys. Rev. A*, 71:022316, 2005.
- [5] A. W. Cross, D. P. DiVincenzo, and B. M. Terhal. A comparative code study for quantum fault-tolerance. *Quant. Inf. Comp.*, 9(7&8): 541–571, 2009.
- [6] A. G. Fowler, A. M. Stephens, and P. Groszkowski. High threshold universal quantum computation on the surface code. *Phys. Rev. A*, 80: 052312, 2009.
- [7] S. J. Gay. Quantum programming languages: Survey and bibliography. *Math. Struct. in Comp. Sci.*, 16(4):581–600, 2006.
- [8] J. Grattage and T. Altenkirch. Qml: Quantum data and control. submitted for publication, February 2005.
- [9] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: a scalable quantum programming language. In H.-J. Boehm and C. Flanagan, editors, *PLDI*, pages 333–342. ACM, 2013. ISBN 978-1-4503-2014-6.
- [10] V. Kliuchnikov. Synthesis of unitaries with clifford+t circuits. arXiv:quant-ph/1306.3200v1, June 2013.
- [11] V. Kliuchnikov, D. Maslov, and M. Mosca. Fast and efficient exact synthesis of single qubit unitaries generated by clifford and t gates. arXiv:quant-ph/1206.523v4, June 2012.
- [12] C.-C. Lin, A. Chakrabarti, and N. K. Jha. Optimized quantum gate library for various physical machine descriptions. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP(99):1–1, 2013. ISSN 1063-8210. .
- [13] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.
- [14] B. Ömer. *Structured Quantum Programming*. PhD thesis, Technical University of Vienna, 2003. URL <http://tph.tuwien.ac.at/~oemer/doc/structqprog.pdf>.
- [15] B. W. Reichardt. Quantum universality by state distillation. *Quantum Inf. Comput.*, 9:1030–1052, 2009.
- [16] P. Selinger. Towards a quantum programming language. *Math. Struct. in Comp. Sci.*, 14:527–586, 2004.
- [17] A. M. Steane and B. Ibinson. Fault-tolerant logical gate networks for Calderbank-Shor-Steane codes. *Phys. Rev. A*, 72:052335, 2009.
- [18] K. M. Svore, D. P. DiVincenzo, and B. M. Terhal. Noise Threshold for a Fault-Tolerant Two-Dimensional Lattice Architecture. *Quant. Inf. Comp.*, 7(4):297–318, 2007.