arXiv:1412.0625v1 [cs.PL] 1 Dec 2014

Quipper: Concrete Resource Estimation in Quantum Algorithms

Jonathan M. Smith
University of Pennsylvania,
Philadelphia, U.S.A.

Neil J. Ross
Peter Selinger
Dalhousie University,
Halifax, Canada

Benoît Valiron
PPS, UMR 7126, Univ Paris Diderot,
Sorbonne Paris Cité, F-75205 Paris, France

Despite the rich literature on quantum algorithms, there is a surprisingly small amount of coverage of their concrete logical design and implementation. Most resource estimation is done at the level of complexity analysis, but actual concrete numbers (of quantum gates, qubits, *etc.*) can differ by orders of magnitude. The line of work we present here is a formal framework to write, and reason about, quantum algorithms. Specifically, we designed a language, Quipper, with scalability in mind, and we are able to report actual resource counts for seven non-trivial algorithms found in the quantum computer science literature.

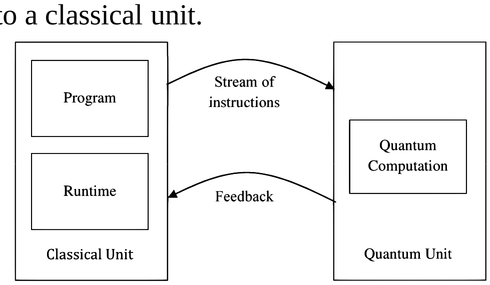
1 Introduction

Quantitative analysis of programs implies a tool base with which resource estimates can be made. In the case of a quantum programming language, the relevant units are ultimately quantum bits (qubits), protected by complex error correction codes. The role of a quantum programming language in the larger context of quantum computing is discussed by van Meter and Horsman [12]. We implemented a quantum programming language, Quipper [5, 4, 10], which is sufficiently complete that we can compile a quantum algorithm from a high-level description into a low-level logical quantum circuit, i.e., a sequence of quantum gates. From such a gate sequence, the resources required by a quantum computation can be estimated. We have performed resource estimates for seven quantum algorithms, with problem sizes selected to be at the point where the quantum algorithm should outperform a classical computer. The power of our approach is that it carries with it the generality of a programming language, needing only to be parameterized by the characteristics of the problem. The resource estimates have also identified ripe optimization targets for enhancements to Quipper.

2 Quantum computation

Quantum computation deals with data encoded on the state of particles governed by the laws of quantum physics. A piece of quantum data can be seen as a complex combination (or *superposition*) of pieces of classical data. This is reminiscent of probabilistic computation, with the difference that the coefficients are complex numbers.

The most common model for a quantum computer is Knill's QRAM model [9], where a quantum device serves as co-processor to a classical unit.



Extended abstract for a talk given at QAPL 2014.

The classical unit performs tasks such as compilation and bookkeeping, and can also send streams of instructions to the quantum unit, which only performs purely quantum operations. The instructions sent to the quantum co-processor are of three kinds:

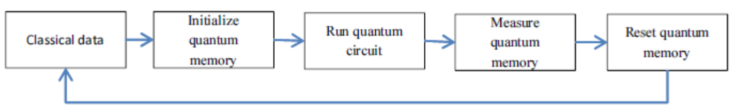
- initializations (to send classical data to the quantum device),
- unitaries (which are reversible operations), and
- measurements (to retrieve classical information from the quantum device).

Measurements are the only way for the quantum device to provide feedback to the classical unit. This operation is probabilistic and can globally affect the state of the quantum device.

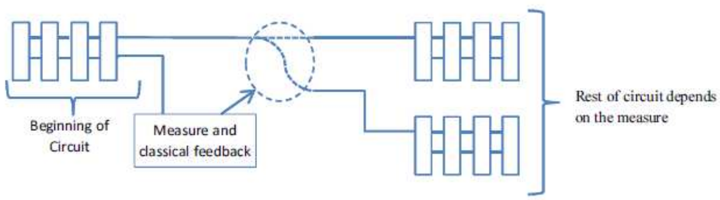
3 Generalized circuit model

There is no control flow on the quantum co-processor. Any loop or conditional branching has to come from the classical device controlling the co-processor. As a result, a quantum computation can be pictured as a linear circuit, representing the flow of elementary instructions sent to the co-processor. In this representation, a wire stands for a quantum register (i.e., a *quantum bit*), and a box represents an elementary operation.

Many algorithms use the quantum device in a simple, batch-style fashion as follows:



In some algorithms, however, the circuit is conditioned on the result of intermediary measurements. Such a circuit is generated "on the fly" by the classical device, and a particular part of the circuit can depend on a measurement done at a previous stage:



A scalable quantum programming language must therefore accommodate such a dynamic representation of circuits.

4 Our proposal: Quipper

We introduce Quipper [5, 4, 10], a functional language for quantum computation embedded in Haskell. Quipper is intended to offer a unified general-purpose programming framework for quantum computation. Its main features are:

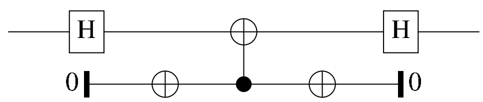
- An extended circuit model. Initializations and terminations of qubits are tracked for the purpose of ancilla management.
- Hierarchical circuits. Quipper features subroutines (or *boxing*) at the circuit level. This permits compact representation of circuits in memory.
- A circuit description language. It can handle procedural and applicative paradigms of computation, and its monadic semantics allows high-level manipulations of circuits with programmable operators.

- Two run-times. A Quipper program describes a *family* of circuits, which may depend on some classical parameters. After compilation, a program is first executed to generate a circuit (circuit generation time), and then the circuit is executed (circuit execution time).
- Parameter/input distinction. Quipper has two notions of classical data: parameters, which are known at circuit generation time, and inputs, which are known at circuit execution time. For example, the type `Bool` stands for parameters and the type `Bit` for inputs.
- Extensible datatypes. Quipper offers an abstract view of the notion of quantum data using the powerful type class mechanism of Haskell's type system.
- Automatic generation of quantum oracles. Concrete quantum algorithms come with non-trivial classical operations that have to be lifted to quantum operations. Quipper comes with a facility to turn ordinary Haskell programs into reversible circuits, using Template Haskell.

Similarly to what is done in Lava [3], the semantics of circuit generation is captured in a monad. This permits both imperative-style programming, where a circuit is described gate by gate, and declarative-style, where circuits are manipulated as first-class objects, and transformed with combinators.

An example of code with the corresponding circuit is shown below.

```
circ :: Qubit -> Circ Qubit
circ x = do
    hadamard_at x
    with_ancilla $ \y -> do
       qnot_at y
       qnot x ‘controlled‘ y
       qnot_at y
    hadamard_at x
    return x
```



5 Achievements

The language Quipper has been designed in the context of the IARPA-funded QCS program [8]. Seven algorithms were used as benchmarks. These algorithms were chosen by IARPA to provide a reasonably representative cross-section of current algorithms.

- Binary Welded Tree (BWT). To find a labeled node in a graph [2].
- Boolean Formula (BF). To evaluate a NAND formula [1]. The version of this algorithm implemented in Quipper computes a winning strategy for the game of Hex.
- Class Number (CL). To approximate the class group of a real quadratic number field [6].
- Ground State Estimation (GSE). To compute the ground state energy of a particular molecule [14].
- Quantum Linear Systems (QLS). To solve a linear system of equations [7].
- Unique Shortest Vector (USV). To choose the shortest vector among a given set [13].
- Triangle Finding (TF). To exhibit a triangle inside a dense graph [11].

These algorithms use of a wide variety of quantum primitives, such as amplitude amplification, quantum walks, the quantum Fourier transform, and quantum simulation. Several of them also require the implementation of complex classical oracles. The starting point for each of our algorithm implementations was a detailed description of the algorithm provided by IARPA. They were all coded in Quipper and are running, in the sense that one can generate the circuit (and portion thereof).

Using Quipper, we were able to perform semi- or completely automated logical gate count estimations for each of the algorithms. For example, in the case of the triangle finding algorithm,

```
./tf -f gatecount -o orthodox -l 31 -n 15 -r 6
```

produces the gate count for the complete algorithm. This runs to completion in under two minutes on a laptop computer, and produces a count of 30,189,977,982,990 (over 30 trillion) total gates and 4676 qubits.

6 Conclusion and future work

Quipper is a scalable language able to manipulate "realistic" quantum algorithms and input sizes. The process of compiling Quipper into a quantum circuit has provided fascinating quantitative insights into the hardware required to execute a quantum computation. As shown above with the triangle finding algorithm, the gate counts are large and may present a fundamental barrier to quantum computing unless significant optimizations in the transformations from algorithm to gates can be found. Both these insights into resource estimates and evaluation of possible optimizations are enabled with Quipper.

However, while Quipper paves the way in the direction of a formal framework to analyze quantum algorithms, much remains to be done. In particular:

- Quipper does not have a dedicated type system. As a result, certain programming errors specific to quantum computing, like violating the non-duplicability of quantum registers, remain the responsibility of the programmer.
- Although one can do concrete gate counts for specific input sizes, it is not clear how to automatically get asymptotic gate counts as a function of the parameters of the algorithm.
- Quipper misses a tool to validate that programs do compute the correct algorithm.

These questions, together with many others, open a rich and potentially fruitful research avenue.

References

[1] A. Ambainis, A. M. Childs, B.W. Reichardt, R. Špalek & S. Zhang (2010): *Any AND-OR Formula of Size N Can Be Evaluated in Time* $N^{\frac{1}{2}+o(1)}$ *on a Quantum Computer*. *SIAM J. Comput.* 39, pp. 2513–2530.

[2] A. M. Childs, R. Cleve, E. Deotto, E. Farhi, S. Gutmann & D. A. Spielman (2003): *Exponential algorithmic speedup by a quantum walk*. In: *Proc. 35th Annual ACM Symposium on Theory of Computing*, pp. 59–68.

[3] Koen Claessen (2001): *Embedded Languages for Describing and Verifying Hardware*. Ph.D. thesis, Chalmers University of Technology and Göteborg University.

[4] Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger & Benoît Valiron (2013): *An Introduction to Quantum Programming in Quipper*. In: *Reversible Computation*, *LNCS* 7948, pp. 110–124.

[5] Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger & Benoît Valiron (2013): *Quipper: a scalable quantum programming language*. In: *Proceedings of PLDI*, pp. 333–342.

[6] Sean Hallgren (2007): *Polynomial-Time Quantum Algorithms for Pell's Equation and the Principal Ideal Problem*. *J. ACM* 54(1), pp. 4:1–4:19, doi:10.1145/1206035.1206039.

[7] Aram W. Harrow, Avinatan Hassidim & Seth Lloyd (2009): *Quantum algorithm for linear systems of equations*. *Phys. Rev. Lett.* 103(15), p. 150502.

[8] IARPA Quantum Computer Science Program (2010): *Broad Agency Announcement IARPA-BAA-10-02*. Available from https://www.fbo.gov/notices/637e87ac1274d030ce2ab69339ccf93c.

[9] Emanuel H. Knill (1996): *Conventions for Quantum Pseudocode*. LANL report LAUR-96-2724.

[10] The Quipper Language (2013): Available from http://www.mathstat.dal.ca/~selinger/quipper/.

[11] Frédéric Magniez, Miklos Santha & Mario Szegedy (2003): *Quantum algorithms for the triangle problem*. Available from quant-ph/0310134.

[12] Rodney Van Meter & Clare Horsman (2013): *A Blueprint for Building a Quantum Computer*. *Communications of the ACM* 56(10), pp. 84–93.

[13] Oded Regev (2004): *Quantum computation and lattice problems*. *SIAM J. Comput.* 33(3), pp. 738–760.

[14] James D. Whitfield, Jacob Biamonte & Alán Aspuru-Guzik (2011): *Simulation of electronic structure Hamiltonians using quantum computers*. *Molecular Physics* 109(5), pp. 735–750.