Towards Turing-Complete Quantum Computing Coming From Classical Assembler

Thomas Gabor, Marian Lingsch Rosenfeld, Claudia Linnhoff-Popien
LMU Munich
thomas.gabor@ifi.lmu.de, M.Rosenfeld@campus.lmu.de

Abstract—Instead of producing quantum languages that are fit for current quantum computers, we build a language from standard classical assembler and augment it with quantum capabilities so that quantum algorithms become a subset of it. This paves the way for the development of hybrid algorithms directly from classical software, which is not feasible on today’s hardware but might inspire future quantum programmers.

Index Terms—quantum computing, quantum language, assembler

I. INTRODUCTION

The capabilities of current quantum computers remain very limited, most importantly with respect to the depth of the executed algorithm. As astonishing as it is that we can still run notably important algorithms (maybe even better than classical ones [1], [2]) on this kind of hardware, the algorithms that can be fully implemented on a quantum computer lack several characteristics that programmers are used to for classical algorithms, most notably: recursion. The absence of arbitrary-depth recursion (or simply jumps) prohibits pure quantum algorithm from emulating every algorithm that can be run on a Turing machine. For today’s quantum computers, this is a more than sensible assumption as decoherence times only allow for a couple of gates to be executed in sequence before all useful information is lost. But what if we had the best quantum computer we could wish for?

In contrast to most other approaches (most notably quantum assembly languages [3]) we attempt to design an assembly language not for quantum computers with today’s capabilities but for an imaginary quantum computer that is not limited by today’s challenges in hardware design and maybe not even limited by all the rules of quantum mechanics at all times. However, we argue that such a language (even though conceivably it will not run on any real quantum computer for a long time) can still be useful for two main reasons:

• First, for now and at least for the immediate future a large amount of quantum algorithms will be run on classical simulators. As these classical simulators run on classical machines, it is reasonable to write algorithms that can access these classical machines’ capabilities directly and thus forego any quantum simulation when possible. This may produce algorithms that – while not compatible with (current) quantum computers – may at least not be as superfluously slow when run on a quantum simulator for testing purposes or simply due to a lack of better quantum options.

• Second, we need tools to discover new quantum algorithms that are more accessible to programmers well versed in standard programming languages. As of now, most setups of quantum software are built around a few algorithm archetypes which are known or suspected to provide some form of quantum advantage. As these few archetypes already hold up all the fascination regarding quantum computers, the discovery of new archetypes should be a big step towards more usage scenarios for quantum computers. Enabling programmers to write an algorithm classically first and then translate it part by part to quantum-suitable reformulations (while maintaining a runnable program) could enable new creativity in algorithm design.

II. APPROACH

In this work-in-progress paper we provide a very simple assembly language that we augmented with a few quantum-specific instructions and capabilities. Table I shows all instructions at our disposal. We provide basic mathematical and logical operations as well as the “set” and “swap” instructions. We then provide a set of instructions to manipulate program flow including the very simple “setpc” and “jump” instructions and the more elaborate “ifte” instruction, which provides the functionality of a standard if-then-else expression in most programming languages. However, all of our registers can be quantum registers, which means that they can contain a superposition instead of a single scalar value. This also means that our program counter can be in superposition and possibly execute different instructions in each of its superposed branches.

To actually introduce said superpositions into the registers, Table II also shows a few quantum-specific instructions. Most notably, we can apply a Hadamard gate to a single qubit or a range of qubits. We also provide a shortcut for the diffusion operator used within Grover’s algorithm as well as for phase multiplication.

Obviously, this set of instruction is far from complete even for classical algorithms but it allows us to formulate a few
with Shor's algorithm [6], which we will discuss in a bit. Naturally, this is a bit counterintuitive to anyone familiar from classical ones. We use a staple example of quantum computing, i.e., finding the prime factors of a number; assume that in the classical world we would simply derive to that problem, i.e., the sieve of Eratosthenes [4]. If we start out with the most common classical approach computing, i.e., finding the prime factors of a given number; but we use a staple example of quantum computing, i.e., finding the prime factors of a number by trying out all possible divisors. In line 8 then creates a superposition of all numbers of the same length as our input “NUMBER1” and with line 9 we ensure that this superposition no longer contains the values 0 and 1 so that we can try all remaining values as possible divisors. In line 10 we again utilize the superposed value in d1 to compute the remainder for all divisors. We then set up the parameters for Grover’s search (lines 11–16) and subsequently perform the calculated amount of iterations within Grover’s search, looking for a single 0 remainder within the superposed value of d2. Note that we implemented the instructions “phase” and “diffusion” within the instruction set specifically to enable a straightforward definition of algorithms like Listing 2 based on Grover’s search. Specialized instructions like these not only increase readability but might also allow future versions of the simulator to implement classical shortcuts for certain behavior.

We compared the quasi-quantum program in Listing 2 to its purely classical counterpart, for which we do not provide an additional listing to save space (and because it looks entirely as expected). It is important to note that we also wrote the classical algorithm for the sieve of Eratosthenes using the same quantum-enabled assembly language and ran it using the same quantum simulator [7]. Figure 1 shows the respective run times of both algorithms for the whole range of inputs. We can see that the classical algorithm takes the longest on actual prime numbers where it needs to try out every possible divisor while the quantum algorithm needs to produce the superposition of all possible divisors for every single input value regardless. However, it is interesting to note that within our simulator, both approaches show a remarkably similar performance. This justifies our approach to offer a “soft transition” from a purely classical execution of a purely classical algorithm to a classical simulation of an algorithm.

### III. Examples

Listing 1 shows a simple example of how we can easily produce when we apply superposition to the program counter. Lines 12–15 all multiply the value in d4 by 2 (see line 6). However, we do not always execute all of them. Instead, we initialize the program counter to the start of these lines (see line 5) and then add the equally distributed superposition of the values $\{0, 1, 2, 3\}$ to that value. When all superposed branches of the program have been executed, we end up with an equal superposition of values $\{1, 2, 4, 8\}$ saved in register d4. We were able to define such a superposed value using patterns familiar from imperative programming and the approach can be generalized to more complicated superposed program flows.

Listing 2 shows how we could evolve quantum algorithms from classical ones. We use a staple example of quantum computing, i.e., finding the prime factors of a given number; but we start out with the most common classical approach to that problem, i.e., the sieve of Eratosthenes [4]. If we assume that in the classical world we would simply derive the prime factors of a number by trying out all possible divisors, it appears evident that such a search for a divisor might be accelerated by employing Grover’s algorithm [5]. Naturally, this is a bit counterintuitive to anyone familiar with Shor’s algorithm [6], which we will discuss in a bit.

---

**Table I**

| Instruction Set for Assembly with Quantum Capabilities |
|--------------------------------------------------------|
| **Classic Instructions**                              |
| add $d_i d_j d_k$ : add the value of cell $i$ with the value of cell $j$ and write the result to cell $k$ |
| mul $d_i d_j d_k$ : multiply the value of cell $i$ with the value of cell $j$ and write the result to cell $k$ |
| div $d_i d_j d_k$ : divide the value of cell $i$ with the value of cell $j$ and write the result to cell $k$ |
| sub $d_i d_j d_k$ : subtract the value of cell $j$ from the value of cell $i$ and write the result to cell $k$ |
| sqrt $d_i d_j$ : write the square root of the value of cell $i$ into cell $j$ |
| mod $d_i d_j d_k$ : write into cell $k$ the modulo of the value of cell $i$ with respect to the value of cell $j$ |
| neg $d_i$ : negate the boolean value of the cell $i$ |
| and $d_i d_j d_k$ : write into cell $k$ the boolean value resulting from $v_i \land v_j$, where $v_l$ is the value of cell $d_l$ |
| or $d_i d_j d_k$ : write into cell $k$ the boolean value resulting from $v_i \lor v_j$, where $v_l$ is the value of cell $d_l$ |
| set $d_i j$ : set the value of cell $i$ to $j$ |
| swap $d_i d_j$ : swap the values of cells $i$ and $j$ |
| setpc $d_i$ : set the program counter to the value of cell $i$ |
| jump $i$ : set the program counter to the value of $i$ |
| skip : do nothing |
| stop : remain in this instruction indefinitely |
| ifte $d_i j k$ : if cell $i$ is true, set the pc to the value of $j$ else set the pc to the value of $k$ |

**Quantum Instructions**

- **havoc $d_i j k$** : apply Hadamard gates to the qubits in range from $j$ to $k$ of cell $i$
- **havob $d_i$** : apply a Hadamard gate to the data qubit $i$
- **diffusion** : apply Grover’s diffusion operator
- **phase $x y$** : multiply the phase of the state by $x + i \cdot y$

---

^2See [https://github.com/marian-lingsch/quantum-assembler](https://github.com/marian-lingsch/quantum-assembler)
with quantum parts without necessarily blow up the run time disproportionally.

IV. CONCLUSION

The assembly language shown here is very limited but is intended to provide an idea how to re-think hybrid programming coming from full-fledged classical languages. We have analyzed how quantum program flow might look like on an assembly level and how it may allow to construct complex quantum states in an intuitive and imperative manner (Listing 1). We have also presented how very versatile quantum abstractions for quantum computing that are both intuitive and powerful are still very much sought after.

In principle, there is no reason to keep this “transition language” at an assembly level; in fact, for usability it might be especially important to translate our concepts of quantum program flow to more high-level language models (and perhaps offer automatic translation back to assembly level to maintain compatibility). An ideal case might be to allow a programmer to use a modern programming language and insert quantum-compatible expressions one by one until an interpreter or compiler notices that a certain module is now fully quantum-compatible and then offers the option to also run it on quantum hardware (compare Tyagi et al. [8] for the case of reversible computing). However, high-level abstractions for quantum computing that are both intuitive and powerful are still very much sought after.

REFERENCES

[1] S. Bravyi, D. Gosset, and R. König, “Quantum advantage with shallow circuits,” Science, vol. 362, no. 6412, pp. 308–311, 2018.
[2] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell et al., “Quantum supremacy using a programmable superconducting processor,” Nature, vol. 574, no. 7779, pp. 505–510, 2019.
[3] A. W. Cross, A. Javadi-Abhari, T. Alexander, N. de Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, J. Smolin, J. M. Gambetta, and B. R. Johnson, “OpenQASM 3: A broader and deeper quantum assembly language,” arXiv preprint arXiv:2104.14722, 2021.
[4] C. Hoare, “Proof of a structured program: ‘The sieve of Eratosthenes’,” The Computer Journal, vol. 15, no. 4, pp. 321–325, 1972.
[5] A. Mandviwalla, K. Ohshiro, and B. Ji, “Implementing Grover’s algorithm on the IBM quantum computers,” in 2018 IEEE International Conference on Big Data (Big Data). IEEE, 2018, pp. 2531–2537.
[6] B. F. Lanyon, T. J. Weinhold, N. K. Langford, M. Barbieri, D. F. James, A. Gilchrist, and A. G. White, “Experimental demonstration of a compiled version of Shor’s algorithm with quantum entanglement,” Physical Review Letters, vol. 99, no. 25, p. 250505, 2007.
[7] T. Gabor, M. Lingsch Rosenfeld, and C. Linhoff-Popien, “Simple quantum state encodings for hybrid programming of quantum simulators,” in 2nd Workshop on Quantum Software Architecture (QSA), 2022.
[8] N. Tyagi, J. Lynch, and E. D. Demaine, “Toward an energy efficient language and compiler for (partially) reversible algorithms,” in International Conference on Reversible Computation. Springer, 2016, pp. 121–136.