Enfield: An OpenQASM Compiler

Marcos Yukio Siraichi[1], **Caio Henrique Segawa Tonetti**[1]

[1]Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

{yukio.siraichi,caiotonetti}@dcc.ufmg.br

Abstract. *In 2016, IBM made quantum computing accessible to everyone by deploying a cloud-based quantum computer. It elicited much enthusiasm for quantum computing research and testing. When writing programs for executing in this environment, the programmer had to manually tune the program so that it complied to some resource restrictions specific to IBM's machine. Hence, we generalized and automated this process, building Enfield: a source-to-source compiler for transforming quantum programs, allocating resources and optimizing. It features state-of-the-art implementations and the extensibility necessary for implementing and comparing new solutions. We illustrate its effectiveness with benchmarks from IBM that were used in previous research and show that our tool performs at least* 22% *better.*

Video Link: https://youtu.be/u5OgQAlB3L0

1. Introduction

With the recent popularization of quantum computing and the introduction of experimental prototypes available for the general public, the possibility of building and programming quantum experiments has elicited much enthusiasm [Devitt 2016]. In this platform, tools that aid the creation of programs are available, such as visual circuit representation and a gate-level syntax based on the Open Quantum Assembly (OpenQASM) [Cross et al. 2017]. Visual circuits are good for learning the basics of quantum programming, but it quickly becomes a tiresome activity for larger programs. OpenQASM, on the other hand, offers a low level of abstraction, demanding a deeper understanding of each gate and the specific constraints for the machine to be executed on [Häner et al. 2018].

Much of this happens because quantum programming still lacks the compiler support that modern languages take advantage of. To use a quantum computer, OpenQASM programmers must tune their code to fit tight resource constraints, while minimizing decoherence issues [Lidar and Brun 2013] and keeping the logic sound. For instance, the *ibmqx2* and *ibmqx3* computers support 5 and 16 qubits respectively, each of them connected by a partial network. There are plans to add 20-qubit and 50-qubit architectures [Dario 2017], but the connectivity in these computers remains restrictive. Moreover, reducing runtime and complexity of the code is essential to assert the accuracy of the solution itself, since quantum gates are not self-stabilizing and accumulate noise. Although quantum error-correction codes exist, the sheer complexity of the techniques restrict what can be done today [Svore et al. 2006]. These problems asks for a tool that is able to take care and optimize quantum programs automatically.

In this context, we propose *Enfield*: an OpenQASM source-to-source compiler written in C++, created to ease the development process of quantum programs and help

the programmer focus on the logic of the algorithm while the compiler carries the burden of adapting and optimizing the code for the target architecture. Furthermore, *Enfield* offers an organized set of data structures and functions to help quantum developers create new optimizations.

2. Related Work

Several design and implementation proposals are found in [Svore et al. 2006, Javadi Abhari et al. 2014, Chong et al. 2017, Häner et al. 2018]. [Svore et al. 2006] and [Chong et al. 2017] introduced fundamentals on the compilation process as well as some design remarks. [Javadi Abhari et al. 2014] implemented a compiler for a high-level language [Abhari et al. 2012] and evaluated some compilation methods for large quantum programs. In contrast, *Enfield* is a source-to-source compiler designed for solving machine-dependent problems in the low-level OpenQASM language [Cross et al. 2017] , which is used by IBM in its cloud quantum computer [Devitt 2016].

To the best of our knowledge, only `qubiter` [Tucci 2004] and *QISKit*[1] work on tools for solving resource allocation in quantum computers. The tool presented in [Tucci 2004] not only uses another representation for quantum programs, but also its algorithm for allocating qubits is simple and not the main focus of the tool. On the other hand, *QISKit*, the python SDK (Software Development Kit) that IBM created for programming in their quantum computers, is also an OpenQASM source-to-source compiler. Although it is convenient for running experiments, up to this moment, they have implemented only one kind of allocator. *Enfield* has an extensible set of qubit allocators already implemented (including the one used in *QISKit*), and has already been used for resource allocation research in the literature [Siraichi et al. 2018].

3. The Tool: Enfield

As mentioned before, *Enfield* is an OpenQASM source-to-source compiler that aims to ease the task of adapting a machine-independent OpenQASM program to a program that complies to all resource restrictions of the machine the program will run on. Our tool is open source (GNU GPLv2) and was implemented with a modular architecture, aiming for low coupling and high extensibility. Briefly describing the modules, we have:

1. **Parsing:** uses a combination of GNU Flex and Bison for parsing and creating an AST (Abstract Syntax Tree). The driver functions are also implemented here;
2. **Transformations:** all program manipulation code is written as an inheritance of the class `Pass`, such as module analysis, and program transformations;
3. **Architecture:** every architecture related structures and methods, such as the architecture's layout (currently the *ibmqx2* and *ibmqx3*);
4. **Support:** some useful functions and classes that do not strictly belong anywhere, such as: command line options; `Graph` class; etc.

Transformations in *Enfield* are implemented as passes through the modules. We have them divided in three different groups: (I) the preparation phase, which prepares the modules to be allocated; (II) the allocators, which solve the "Qubit Allocation Problem"; and (III) the verifiers, which are passes that verify whether we allocated the program

[1] https://github.com/QISKit/qiskit-sdk-py

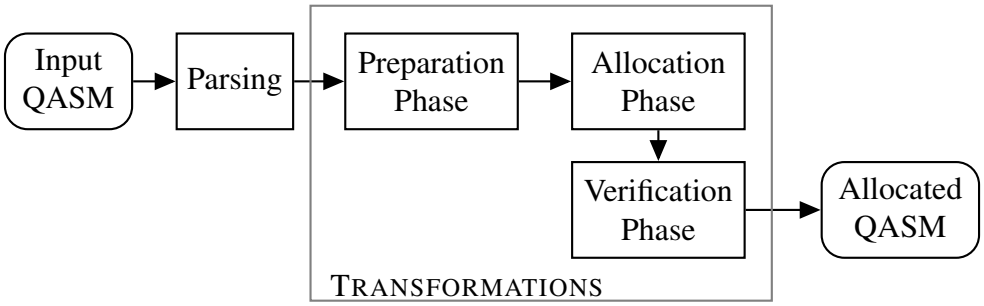

Figure 1. Overview of the transformations applied to a quantum program. First, it goes through parsing, then it is prepared for the allocation phase, and finally its correctness is verified before being outputted.

correctly. Figure 1 illustrates the path that each allocated OpenQASM program goes through: it is parsed, prepared and then allocated for the chosen architecture. Finally, the program is verified and outputted.

As the main passes of the preparation phase, there are: the `FlattenPass`, that replicates the gates that use quantum registers instead of qubits; and the `InlineAllPass`, which inlines all gates into basic operations (CNOTs, Haddamards, etc.). The allocation phase transforms the program in order to comply with the described architecture restrictions. Lastly, the verification phase has two passes: `ArchVerifierPass`, responsible for checking if all restrictions imposed by the architecture were satisfied; and `SemanticVerifierPass`, that checks whether the input program has the same behavior as the generated one.

The following sections describe the solutions to the two most important problems in *Enfield* up to this moment: the "Qubit Allocation Problem", in Section 3.1; and the "Token Swapping Problem", in Section 3.2.

3.1. Qubit Allocation Problem

[Siraichi et al. 2018] formally described and showed that this is an NP-Hard problem. *Enfield*, focuses exactly on this problem. Given a program that is unaware of the architecture that will execute it, we have to generate a second program that complies to the architecture description (represented as an undirected graph) while maintaining the semantics of the original program. We implemented the state-of-the-art solutions for this problem mentioned before as "allocators", namely: `dynprog`, which is the exact dynamic programming solution; `wpm`, a heuristic created in our tool; `ibmmapper`, the IBM solution to this problem; etc.

3.2. Token Swapping Problem

The previous problem discussed in Section 3.1 makes use of the "Token Swapping Problem" [Yamanaka et al. 2014], which is also shown to be NP-Hard. Our tool implements two solutions for this problem: an exact solution that takes exponential time and space (it is roughly factorial on the number of qubits); and a 4-approximative solution of the colored version (general version of the problem) [Miltzow et al. 2016], which can be solved in polynomial time.

4. Working Example

Enfield receives a program written in OpenQASM as input, giving the user the possibility to choose the target architecture and one of the available algorithms for qubit allocation. Figure 2 shows the effects of our tool. On the left there is an example of an input OpenQASM program. Although it is not an actual algorithm, since it does not compute anything, it is brief and serves our purposes. The right hand side of the figure illustrates the same input program on the left, after the last allocation and verification phase.

```
1  qreg q[2];
2  qreg r[1];
3  gate round_trip a, b {
4    cx a, b; cx b, a;
5  }
6  h q;
7  round_trip q, r[0];
```

```
 8  qreg q[5];
 9  h q[0];
10  h q[1];
11  cx q[0], q[2];
12  reverse_cx q[2], q[0];
13  cx q[1], q[2];
14  reverse_cx q[2], q[1];
```

Figure 2. On the left, we have the input OpenQASM program that applies the gate `h` and the custom gate `round_trip` to the quantum registers `q` and `r`. On the right, the allocated OpenQASM translates all qubits to the ones defined by the architecture, replicates the uses of each gate based on the number of qubits in `q` and inlines the resulting gates.

OpenQASM syntax is similar to array-based operations. In this context, we call each array "registers", which may be "quantum" (`qreg`) or "classical" (`creg`), and its elements "bits" ("qubits" and "cbits"), allowing us to observe the following transformations:

1. `FlattenPass`: looks for register operations "`h q;`" (line 6) and turn them into N instructions (lines 9-10), where N is the number of qubits in the register `q`;
2. `InlineAllPass`: copies the body of the gates, e.g. "`gate round_trip`" (lines 3-5), into the gate applications, e.g. "`round_trip q, r[0];`" (lines 11-14);
3. `QbitAllocation`: translates the registers used in the program (lines 1-2) into the ones in the architecture (line 8), and inserts new operations for complying with the restrictions imposed by the architecture (lines 12-14).

Enfield is accessible at `http://cuda.dcc.ufmg.br/enfield/` as an online tool (Figure 3) and is open sourced, available on GitHub at `https://github.com/ysiraichi/enfield`.

5. Experiments

We gathered some OpenQASM programs released by IBM to be used as our benchmark and complemented them with randomly generated OpenQASM programs. These synthetic programs were generated using a uniform distribution of gate applications.

We evaluated the qubit allocators implemented in *Enfield* against `ibmmapper`, implemented within QISKit, and `qubiter` [Tucci 2004]. As the target architecture, we set *ibmqx2*. Figure 4 shows the resulting costs of the qubit allocation for each IBM program and Figure 5 shows the ratio of the costs over the optimal cost for the synthetic

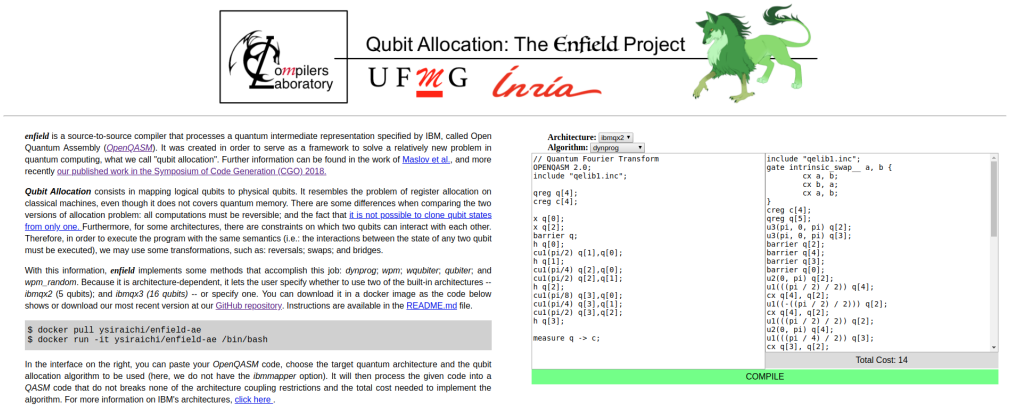

Figure 3. Screenshot of *Enfield*'s web interface, showing example of output..

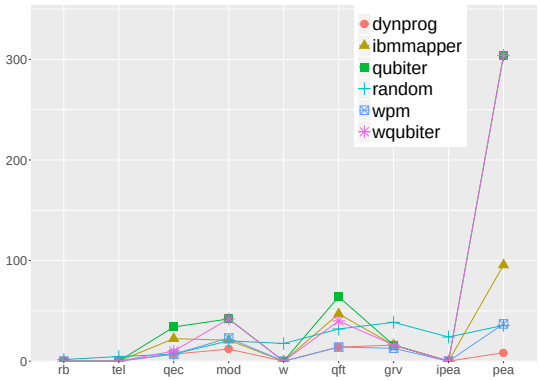

Figure 4. Costs (y-axis) of the output programs using each allocator on the IBM benchmarks (x-axis).

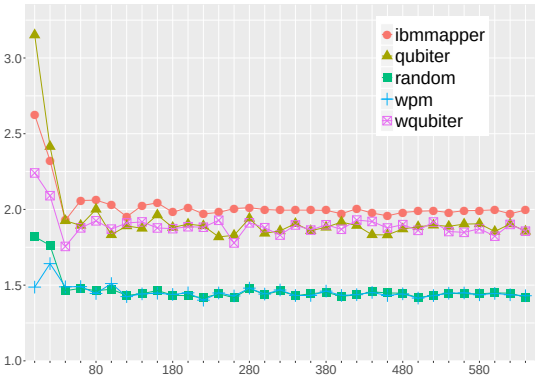

Figure 5. Ratio (y-axis) of the cost using each allocator over the optimal (*dynprog*) as we increase the size of the input (x-axis).

programs. The experiments were performed on an Intel Core i7-4700MQ computer with 8GB of RAM and a clock of 2.4GHz.

In general, `ibmmapper` was outperformed by our algorithms. We could find the exact solution in 7 out of 9 cases as opposed to 5 out of 9 cases from `ibmmapper` in the IBM programs. `qubiter` obtained similar results in most of these cases, but failed to deliver a good result on specific benchmarks. On the synthetic programs, we fared 27% better than `ibmmapper` and 22% better than `qubiter`.

6. Conclusion

This paper described an open source tool called *Enfield*, a compiler for the OpenQASM language which is used by the IBM quantum computers. We implemented the state-of-the-art solutions of the Qubit Allocation Problem that are used for generating better programs for real quantum computers. In our tool, we generalized these quantum computer problems and implemented them as an extensible framework for creating and testing new transformations. Summarizing, not only *Enfield* provides a compiler to help programmers create quantum programs without having to worry about architectural constraints, but also a framework to aid researchers in the implementation of new optimizations. *Enfield* is one

of the first steps towards better programming environments for quantum computers that, in the near future, will be capable of solving problems in the real world.

References

[Abhari et al. 2012] Abhari, A. J., Faruque, A., Dousti, M. J., Svec, L., Catu, O., Chakrabati, A., Chiang, C.-F., Vanderwilt, S., Black, J., Chong, F., Martonosi, M., Suchara, M., Brown, K., Pedram, M., and Brun, T. (2012). Scaffold: Quantum programming language. Technical report, Princeton University.

[Chong et al. 2017] Chong, F. T., Franklin, D., and Martonosi, M. (2017). Programming languages and compiler design for realistic quantum hardware. volume 549. Macmillan Publishers Limited, part of Springer Nature. All rights reserved.

[Cross et al. 2017] Cross, A. W., Bishop, L. S., Smolin, J. A., and Gambetta, J. M. (2017). Open quantum assembly language.

[Dario 2017] Dario, G. (2017). The future is quantum.

[Devitt 2016] Devitt, S. J. (2016). Performing quantum computing experiments in the cloud.

[Häner et al. 2018] Häner, T., Steiger, D. S., Svore, K., and Troyer, M. (2018). A software methodology for compiling quantum programs. *Quantum Science and Technology*, 3(2):020501.

[Javadi Abhari et al. 2014] Javadi Abhari, A., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F. T., and Martonosi, M. (2014). Scaffcc: A framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 1:1–1:10, New York, NY, USA. ACM.

[Lidar and Brun 2013] Lidar, D. and Brun, T. (2013). *Quantum Error Correction*. Cambridge University Press.

[Miltzow et al. 2016] Miltzow, T., Narins, L., Okamoto, Y., Rote, G., Thomas, A., and Uno, T. (2016). Approximation and hardness of token swapping. In Sankowski, P. and Zaroliagis, C., editors, *24th Annual European Symposium on Algorithms (ESA 2016)*, volume 57 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 66:1–66:15, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Siraichi et al. 2018] Siraichi, M. Y., Santos, V. F. d., Collange, S., and Pereira, F. M. Q. (2018). Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 113–125, New York, NY, USA. ACM.

[Svore et al. 2006] Svore, K. M., Aho, A. V., Cross, A. W., Chuang, I., and Markov, I. L. (2006). A layered software architecture for quantum computing design tools. *Computer*, 39(1):74–83.

[Tucci 2004] Tucci, R. R. (2004). Qubiter algorithm modification, expressing unstructured unitary matrices with fewer cnots.

[Yamanaka et al. 2014] Yamanaka, K., Demaine, E. D., Ito, T., Kawahara, J., Kiyomi, M., Okamoto, Y., Saitoh, T., Suzuki, A., Uchizawa, K., and Uno, T. (2014). Swapping labeled tokens on graphs. In Ferro, A., Luccio, F., and Widmayer, P., editors, *Fun with Algorithms*, pages 364–375, Cham. Springer International Publishing.