UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Handy Kurniawan

Compiler for a Quantum Language

Master's Thesis (30 ECTS)

Supervisor: Dr. Dirk Oliver Theis

Tartu 2022

Compilers for Quantum Languages

Abstract:
With the increasing interest in experimenting with quantum computing in academia and industry, creating more versatile software tools, e.g., compilers, is needed. The compiler will transform the quantum assembly languages into either binary code for a quantum-accelerated classical CPU or a GPU-accelerated quantum-circuit simulator. The current inadequate supply of quantum computers delays the research and development of quantum algorithms. This reason causes the quantum circuit simulators to become explorative back-ends. The goal of the present master's thesis project was to develop a compiler for the commonly used quantum-circuit description language "OpenQASM2" into binary code for GPU-accelerated quantum-circuit simulation. For the quantum-circuit simulator (compiler back-end), the requirement was to use the state-of-the-art GPU-based "Quantum Exact Simulation Toolkit", QuEST. In the compiler front-end, the requirement was to develop a parser based on the venerable Flex/Bison combination. The result of this thesis project is a fast, dependable compiler written in C++. Along with the source code, a curated collection of OpenQASM2 source files serve as a test suite. In further work, the compiler will be integrated with Nordic-Estonian Quantum Computing e-Infrastructure Quest (NordIQuEst) project that will make it available to HPC users.

Tüübituletus neljandat järku loogikavalemitele

Lühikokkuvõte:

Kuna huvi kvantandmetöötlusega eksperimenteerimise vastu on kasvamas nii akadeemilises maailmas kui ka tööstuses, suureneb ka vajadus mitmekülgsemate mitmesuguste tarkvaratööriistade, nt kompilaatorite järele. Kompilaator muundab kvantassemblerkeeled kas klassikalise kvantkiirendatud keskprotsessori binaarkoodiks või keskprotsessori kiirendatud kvantahela simulaatoriks. Kvantarvutite praegune ebapiisav kättesaadavus takistab kvantalgoritmide uurimist ja arendamist ning seetõttu on kvantahela simulaatoreod hakatud kasutama põnevaid võimalusi pakkuvate tagasüsteemidena. Selle magistritöö eesmärk oli välja töötada kompilaator üldkasutatava kvantahela kirjelduskeele „OpenQASM2" muutmiseks binaarkoodiks, mida saab kasutada keskprotsessori kiirendatud kvantahela simulatsiooniks. Kvantahela simulaatori (kompilaatori põhiarvuti) puhul seati eelduseks kasutada tipptasemel keskprotsessoripõhist täpse matkimise tööriistakasti „Quantum Exact Simulation Toolkit", QuEST. Kompilaatori eessüsteemiga seoses seati eesmärgiks luua parser, mis põhineb Flex/Bisono kombinatsioonil. Selle magistritöö tulemusena loodi kiire, usaldusväärne C++ keeles kirjutatud kompilaator. Koos lähtekoodiga saab OpenQASM2 lähtefailide piiratud kogumit kasutada testkomplektina. Tulevikus integreeritakse loodud kompilaator Põhjamaade-Eesti ühisprojektiga „Quantum Computing e-Infrastructure Quest" (NordIQuEst), mis muudab selle kättesaadavaks kõrgjõudlustöötluse (HPC) kasutajatele.

Võtmesõnad:

kvantandmetöötlus, kompilaatori ehitus, OpenQASM

CERCS:

P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Acknowledgements

I would first like to thank my thesis supervisor, Assoc. Prof. Dr. Dirk Oliver Theis of the Theoretical Computer Science at the University of Tartu, for his excellent guidance and encouragement throughout working on this thesis. And last but not least, many thanks to my parents for their endless support.

Contents

1 Introduction

Research in quantum computing started to gain popularity at least two decades ago [Öm98, SEL04, LJL$^+$10, MMRP21]. Quantum computing is a computation term that harnesses quantum physics properties such as superposition, entanglement, and quantum interference to do the calculation [LJL$^+$10]. With this technique, it boosts the computational power for solving problems in chemistry [OCR$^+$17], finance [RGB18], machine learning [BWP$^+$17], optimization [SVPO$^+$17], and more [GAN14]. However, building a quantum computer requires a tremendous amount of money, which makes the number of quantum field researchers outweigh the number of the existing quantum computers. For this reason, a quantum simulator may boost development in quantum algorithms.

The challenge in using the quantum computer simulator is the need for specific skills, such as programming languages and coding. In contrast, quantum algorithms are usually described in a machine-independent quantum assembly language, e.g., OpenQASM2 [CBSG17] as the commonly used quantum-circuit description language. We use QuEST [JBBB19] as the compiler back-end and a Flex/Bison parser [LL09] as the compiler front-end in this thesis.

This thesis supports Nordic-Estonian Quantum Computing e-Infrastructure Quest (NordIQuEst), to make a compiler for the QuEST simulator. To make the compiler, we are required to integrate QuEST as our compiler back-end. However, our effort may not be the first and most likely not be the last in the myriad of existing open-source quantum full-stack libraries, such as staq [AG20], XACC [MLD$^+$20], Cirq [Dev21], ProjectQ [SHT18], and Strawberry Fields [KIQ$^+$19] just to name a few.

The protocol established in this thesis aims to guide a quick, dependable compiler written in C++. Along with the source code, a curated collection of OpenQASM2 source files serve as a test suite. The first part of the compiler, Flex/Bison, will produce an Abstract Syntax Tree (AST) processed into a C file. The latter part, QuEST, will generate an executable file that returns the quantum circuit result.

This thesis describes context behind the project and an overview (Section 1) of the components of a compiler OpenQASM (Section 2). Then it provides the requirements and specifications, an overview of the created compiler, the methodologies used in this work and the implementation (Section 3). This is then followed by the results, evaluation and discussion (Section 4). Finally, it ends with the conclusions and future directions (Section 5).

2 Literature Review

This section covers the basic principles and concepts required to develop the methodology. The first part describes the OpenQASM2 language, followed by a compiler's general structure. Afterwards, the quantum simulator and universal quantum gates are discussed. The chapter ends with a brief introduction of the state-of-the-art of existing open-source quantum full-stack libraries.

2.1 OpenQASM 2

Open Quantum Assembly Language (OpenQASM) combines C and assembly languages that describe quantum circuits built on earlier QASM dialects [CBSG17, CJAA$^+$22]. At the lowest level, OpenQASM2 only have two elementary gates: unitary $U(\theta, \phi, \lambda)$ and the two-qubit controlled X gate (CX). The rest of the gates are defined in the "qelib1.inc" file.

Programs in OpenQASM2 are structured as sequences of declarations (user-defined gate and quantum/classical) and commands (gate application, measurement and initialization of qubits, barrier, reset, and classically controlled gates). Figure 1 is the example of quantum error correction in OpenQASM2. Other examples can be seen in the appendix section (Figures 33, 34).

```
1  // Repetition code syndrome measurement
2  OPENQASM 2.0;
3  include "qelib1.inc";
4  qreg q[3];
5  qreg a[2];
6  creg c[3];
7  creg syn[2];
8  gate syndrome d1,d2,d3,a1,a2
9  {
10   cx d1,a1; cx d2,a1;
11   cx d2,a2; cx d3,a2;
12 }
13 x q[0]; // error
14 barrier q;
15 syndrome q[0],q[1],q[2],a[0],a[1];
16 measure a -> syn;
17 if(syn==1) x q[0];
18 if(syn==2) x q[2];
19 if(syn==3) x q[1];
20 measure q -> c;
```

Figure 1. Quantum error correction QASM file.

2.2 Compiler

A *compiler* is a program that translates a high-level programming language (source) to a lower level language (target) so that the computer can process the latter [GvRB+12]. Nowadays, a compiler can be divided into front-end and back-end that the front-end handles the source language, while the back-end handles the target language. The front-end will produce Intermediate Representation (IR) to interface with the back-end for a particular target machine and separate between the two components. The separation between the front-end and back-end is vital to solving the $L \times M$ problem by creating L front-ends and M back-ends [GvRB+12]. Figure 2 is the overview of a compiler's phases

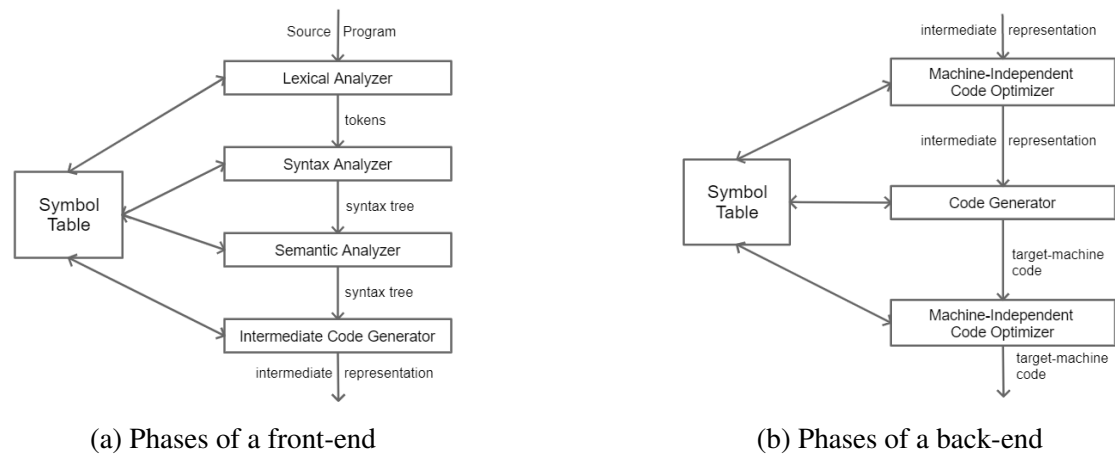(a) Phases of a front-end   (b) Phases of a back-end

Figure 2. Phases of a compiler [AA07].

2.2.1 Front-end

The compiler front-end usually analyzes the source program and generates an IR. The details of the source language are kept in the front end. As in figure 2a, the front-end phases are lexical analysis, syntax analysis, semantic analysis, and intermediate code generation.

Lexical Analysis   The task of the lexical analyzer is to read the input streams and group them into meaningful sequences, called *lexemes*. Each *lexeme* will represent a token containing a name and an attribute value kept in the symbol table. This token will be used in the syntax analysis. In this work, we use Flex which will be explained later [AA07].

Syntax Analysis   The task of the syntax analyzer is to use the first element of the tokens generated by the lexical analyzer to build a syntax tree which is an IR that depicts the

grammatical structure of the token stream. Each of the interior nodes in the syntax tree represents an operation, and the node's children represent the operation's arguments. This syntax tree will be used in the semantic analysis. In this work, we use Bison, which will be explained later.

Semantic Analysis   The task of the semantic analyzer is to make sure the input is correct semantically, like checking declaration before being used, static index range, scoping, and static type checking. It will process the syntax tree from the syntax analysis and get information from the symbol table to generate a more meaningful syntax tree (AST) used during intermediate code generation.

Intermediate Code Generation   The task of the intermediate code generator is to create an explicit low-level or machine-like IR that can be used for any compiler back-end. The generated IR should be easy to produce and translate into the target machine. In this work, we will produce the IR as a C file. Then, this IR will be compiled by the compiler back-end (QuEST).

2.2.2   Back-end

The compiler back-end usually generates target code from the IR produced by the front-end. The details of the target machine are kept in the back-end. As in figure 2b, the back-end phases are machine-independent optimization, code generation and machine-dependent optimization. However, we use QuEST as the compiler back-end because it is one of the requirements of this project. The QuEST compiler will be explained in the later section.

2.2.3   AST

The *abstract syntax tree* (AST) is a modified syntax tree that contains detailed information about the semantics in its nodes. This information is obtained through annotations by parsing a given grammar [GvRB+12]. Usually, the information is the data type and the location in the input text. Figure 3 is the example of an AST.
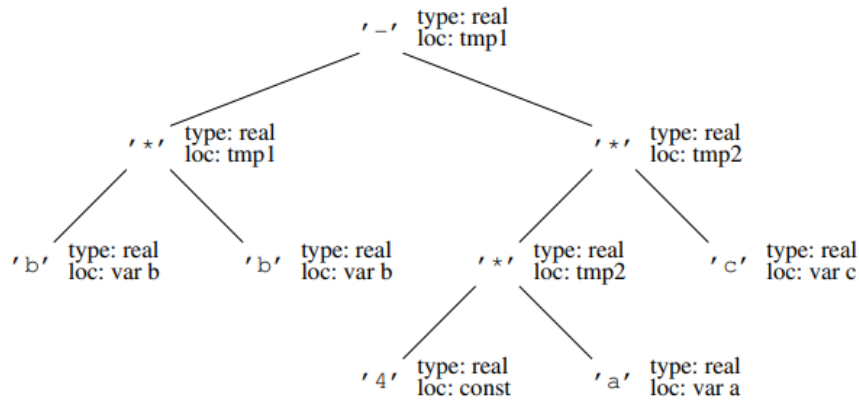
Figure 3. The expression of b*b - 4*a*c as an AST [GvRB⁺12] .

2.2.4 Flex and Bison
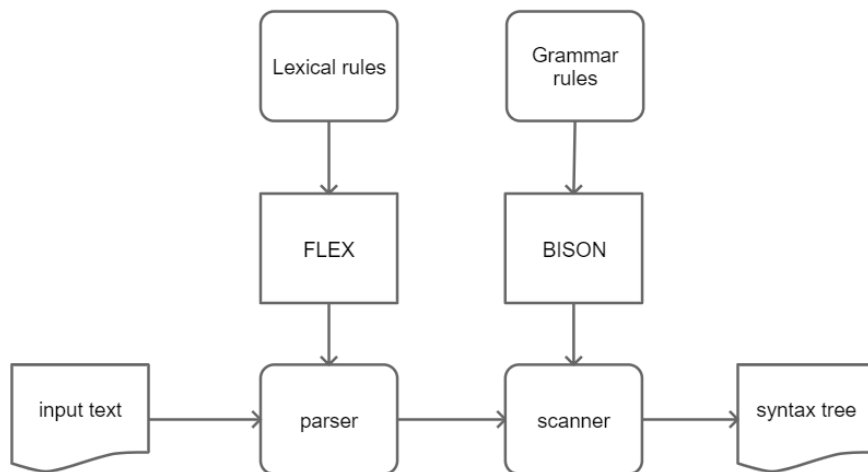
Figure 4. Overview of the Flex/Bison process.

Flex *Fast lexical analyzer generator* is a tool for generating programs that perform pattern-matching on text or can be called a lexical analyzer or scanner. As a lexical analyzer, Flex produces a token that the parser will use. [PEM16]

Bison *GNU Bison*, syntax analyzer generator, is a tool for generating programs that convert an annotated context-free grammar into a deterministic LR or Generalized LR (GLR) parser. Then the parser employs LALR(1), IELR(1) or canonical LR(1) parser

11

tables [Bis21]. In this work, we use LALR(1), a robust parser that can handle large grammar classes.

Flex/Bison When combined, Flex/Bison can generate a program that handles structured input, i.e., a scanner and parser. Since they can create "pure" reentrant code, this allows recursive calls to the scanner and parser and allows scanners and parsers to be used in multithreaded programs. They accommodate generating parsers in C/C++ languages. For a further introduction see [LL09]. Figure 4 is the overview of the Flex/Bison process.

2.3 Quantum Simulator

A *simulator* is a physical device that reveals information about a mathematical function interpreted as part of a physical model. Then, this model will be compared with a real physical system [JCJ14]. So, a quantum simulator is a device that actively uses quantum properties to solve problems about the model systems and the real systems [JCJ14]. Figure 5 shows the role of a quantum simulator.
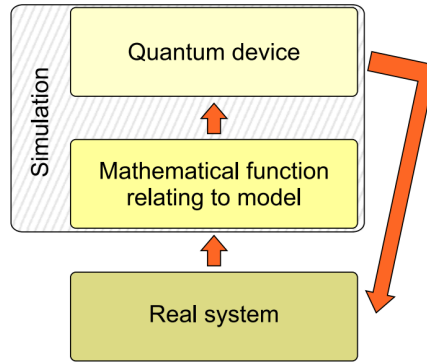
Figure 5. The role of a quantum simulator [JCJ14].

2.3.1 QuEST

Quantum Exact Simulation Toolkit(QuEST) is an open-source state-of-the-art GPU-based quantum-circuit simulator embodied as a stand-alone C library [JBBB19]. In this work, we are using QuEST because it is a part of the NordIQuEst project. Figure 6 is the example of entanglement in QuEST.

QuEST has functions to represent almost all quantum gates. However, since we are translating from OpenQASM2, which only has two elementary gates, we only use the unitary and CNOT functions from QuEST.

```
1  #include <QuEST.h>
2
3  int main() {
4    // load QuEST
5    QuESTEnv env = createQuESTEnv();
6
7    // create a 2 qubit register in the zero state
8    Qureg qubits = createQureg(2, env);
9    initZeroState(qubits);
10
11   // apply circuit
12   hadamard(qubits, 0);
13   controlledNot(qubits, 0, 1);
14
15   // unload QuEST
16   destroyQureg(qubits, env);
17   destroyQuESTEnv(env);
18   return 0;
19 }
```

Figure 6. Example of entanglement program in QuEST.

Documentation and Tutorial   The documentation of QuEST can be found online at https://quest.qtechtheory.org/docs/. This contains instructions on installation and setup, tutorials on how to implement and example programs.

2.4   Universal Quantum Gates

A set of *universal quantum gates* is a small set of gates that can be used to decompose any operation possible in a quantum computer [NC10]. Based on [NC10], these gates are Hadamard (H), phase, controlled-NOT (CNOT), and $\pi/8$ gates. However, from the definition of the OpenQASM language, these gates can also be reduced by two built-in universal gate basis: a single-qubit unitary gate called U and a two-qubit entangling gate called CNOT gate [CBSG17].

2.4.1   Unitary Gate

The single-qubit unitary gate is a built-in gate and parameterized as

$$U(\theta, \phi, \lambda) := R_z(\phi)R_y(\theta)R_z(\lambda) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\phi}\sin\left(\frac{\theta}{2}\right) & e^{i(\lambda+\phi)}\cos\left(\frac{\theta}{2}\right) \end{pmatrix} \tag{1}$$

where $R_y(\theta) = \exp(-i\theta Y/2))$ and $R_z(\phi) = \exp(-i\theta Z/2)$. The command for this gate is **U(theta, phi, lambda) q;**. When q is a qubit, it will apply U to the specific

13

register, but when q are quantum registers, U will be applied to all the registries. See figure 7 for the details.

$$U(\theta,\phi,\lambda)\ q[0];$$

$$U(\theta,\phi,\lambda)\ q;$$

Figure 7. Unitary gate in QASM [CBSG17].

2.4.2 Controlled-NOT Gate

The CNOT gate is a built-in two-qubit entangling gate.

$$\text{CNOT} := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2}$$

The command for CNOT gate is **CX q, r;**. This gate flips the target qubit r if and only if the control qubit q is one. When q and r is a qubit, it will apply CNOT from register q into register r. When q and r are quantum registers, CNOT will be applied to each index accordingly. See figure 8 for details and other cases.
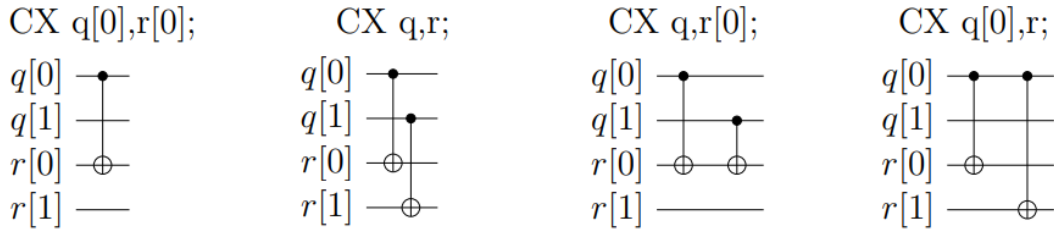
Figure 8. controlled-NOT gate in QASM [CBSG17].

2.5 State of the Art

One of the ways to measure quality is by comparing it to other compilers. Here are some of the open-stack quantum full-stack libraries.

14

2.5.1 Staq

Staq is an open-source compiler and software toolkit for the OpenQASM language written in C++. The primary purpose of staq is cross-compilation between quantum assembly languages. It also provides transformation, optimization and compilation tools that can operate on OpenQASM[AG20]. Figure 9 is an overview of staq.
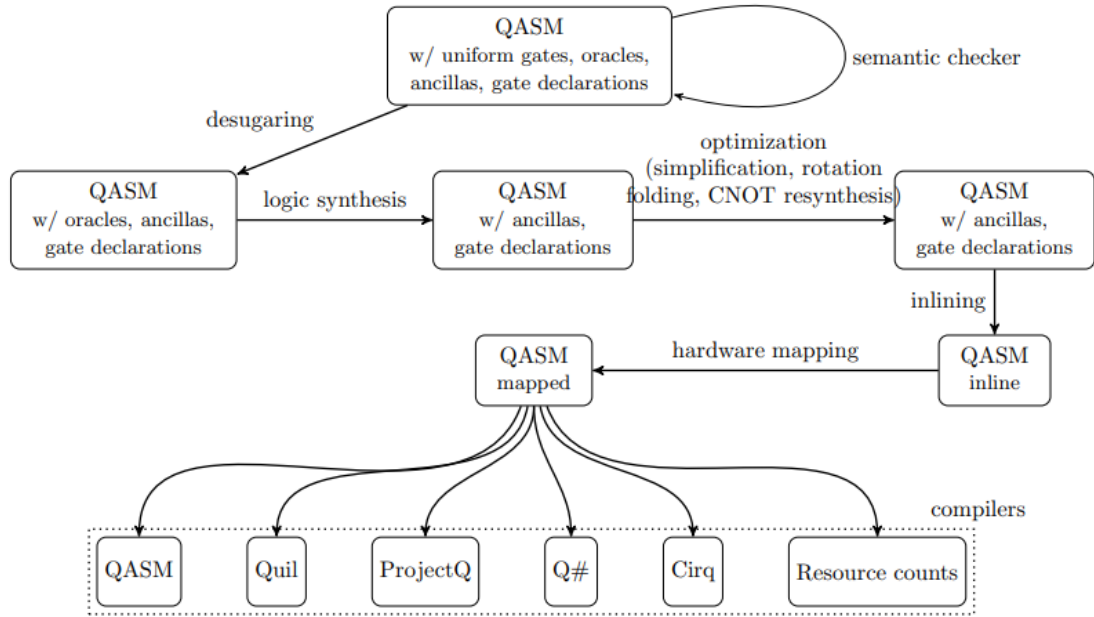
Figure 9. Overview of the staq toolchain [AG20].

2.5.2 XACC

XACC is a system-level software infrastructure for quantum-classical computing built-in C++. The main purpose of XACC is a full-cycle process of quantum compilation from parsing the language until the execution in the quantum hardware/simulation. They currently support quantum-classical programming and execution on IBM, Rigetti, D-Wave QPUs, and many more. Figure 10 is an overview of XACC.
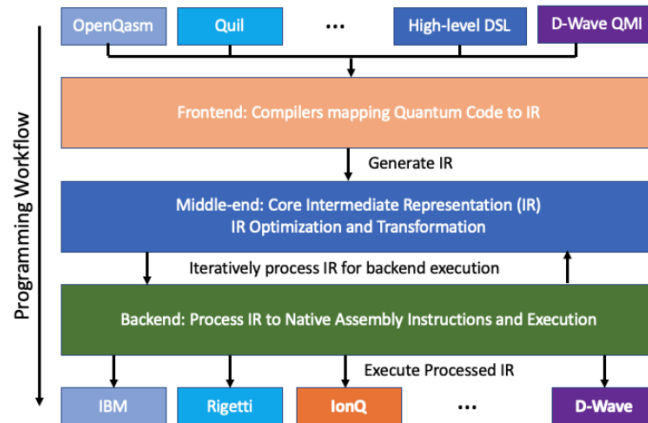
Figure 10. Overview of the xacc toolchain [MLD^{+}20].

2.5.3 Cirq

Cirq is an open-source quantum full-stack library for programming quantum computers written in Python for NISQ devices. It has the features to write, manipulate, and optimize quantum circuits and then execute them in quantum computers and quantum simulators. Figure 11 is an overview of Cirq.
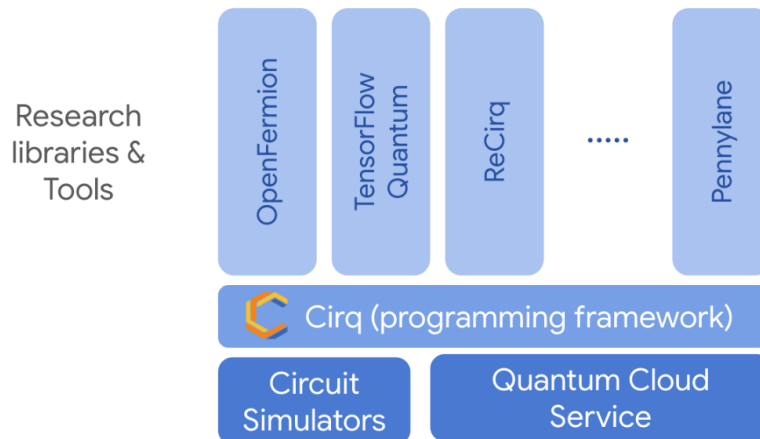
Figure 11. Overview of the Circ toolchain [Dev21].

2.5.4 ProjectQ

ProjectQ is an open-source compilation framework written in Python with capabilities of targeting various types of hardware, a high-performance quantum computer simulator with emulation capabilities, and various compiler plug-ins. Figure 12 is an overview of ProjectQ.
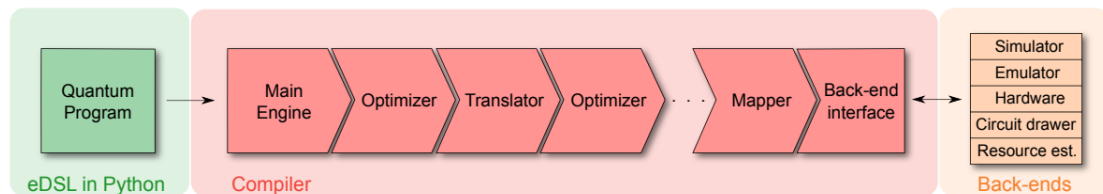
Figure 12. Overview of the ProjectQ toolchain [SHT18] .

2.5.5 Strawberry Fields

Strawberry Fields is an open-source quantum full-stack library written in Python for designing, simulating, optimizing, and quantum machine learning continuous-variable circuits. Its primary purpose is to solve practical problems, including graph and network optimization, machine learning and chemistry. Figure 13 is an overview of Strawberry Fields.
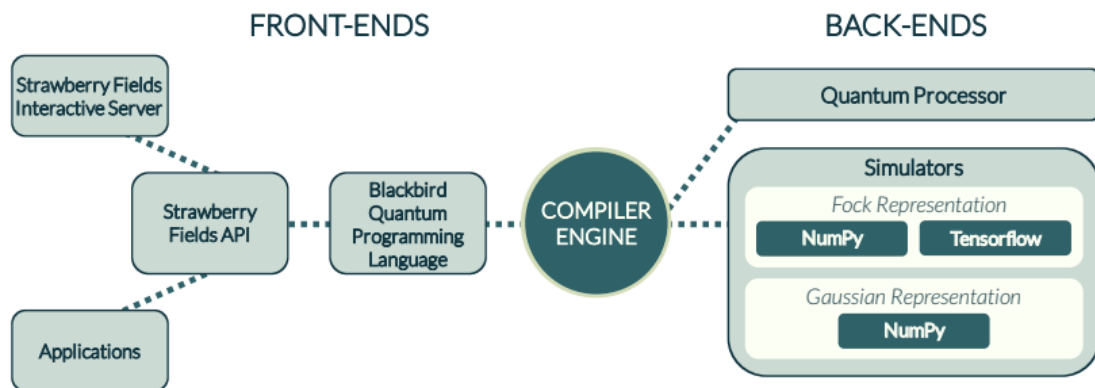
Figure 13. Overview of the Strawberry Fields toolchain [KIQ$^+$19] .

3 OpenQASM2-Compiler

OpenQASM2-Compiler is a new compiler written in C++, aiming to "translate" the OpenQASM2 language into C based code file, and we will run it with the QuEST simulator. This section gives a requirements and specifications, an overview of the architecture, usage, and algorithmic methods of *OpenQASM2-Compiler*.

3.1 Requirements and Specifications

3.1.1 Requirements

The primary purpose of *OpenQASM2-Compiler* is to create an extendable, user-interaction-free compiler, designed according to standard compiler engineering standards (e.g., Grune et all [GvRB$^+$12]). The compiler should return an accurate and stable result with a given quantum circuit in the OpenQASM2 language.

The other requirement of this project is to use QuEST as the compiler back-end because this project is a part of the NordIQuest project.

3.1.2 Specifications

To achieve the extendable feature, we decided to use Flex/Bison because they are good at handling structured inputs [LL09] because the grammar rules of OpenQASM2 are structured inputs [CBSG17]. It is also proven [DD13] that bison can generate a complex bottom-up parser that we use as the syntax analyzer. Then the semantic analysis will be done by our compiler, which comes with a top-down approach.

With a user-interaction-free compiler, the program should process the QASM file as an input and then produce a binary code or error message if the compilation fails. Also, this work comes with a test suite to ensure the compiler's accuracy and stability.

3.2 Overview

The *OpenQASM2-Compiler* processes the input files in two steps. First is the lexical analysis (Flex) and bottom-up syntax analysis (Bison), followed by the top-down semantic analysis. The semantic analyzer will also optimize the file. Figure 14 is the overview of the compiler.

OpenQASM2-Compiler was designed to help researchers run their quantum algorithm without programming skills, only the QASM. After the whole process from scanning to compiling a QASM file, the compiler should output the same result with any quantum computer simulator in a stable environment, i.e., no noise.

Figure 14 gives an overview of how the *OpenQASM2-Compiler* operates. The compiler front-end will scan the QASM file and then parse it into an AST. The semantic
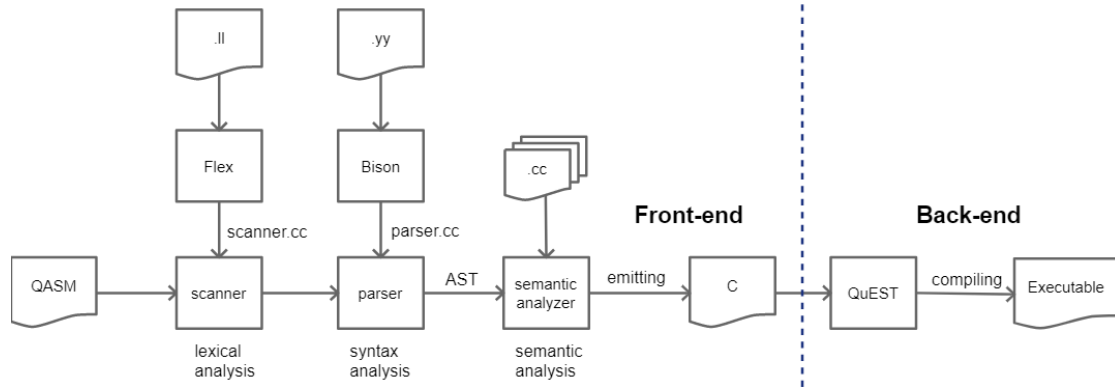
Figure 14. Overview of the *OpenQASM2-Compiler*.

analyzer will process the AST, ensuring if there is an error with the QASM file. Then it will optimize the QASM by inlining, desugaring and decompose (Figure 17). After optimization, the compiler will emit the AST into a C file code compiled by the compiler back-end, QuEST, to produce an executable file.

3.3 Scanner and Parser

Flex/Bison has been widely used to create compiler where Flex is a scanner and Bison is a parser. They are good at handling structured input. In this work, we are using Flex version 2.6.4 [LL09] and Bison version 3.8.1 [Bis21].

3.3.1 Scanner

We use a C++ scanner since Flex is compatible with creating a C++ scanner. Instead of creating a c file, it generates a cc file from the lexical rules files (.ll) based on OpenQASM2 grammar [CBSG17]. The cc file will be compiled into a scanner. This scanner reads the input stream (QASM) and compiles it into a token. The scanner will interact with the parser and then keep the information in the Symbol Table. Figure 15 shows the process of the scanner.

3.3.2 Parser

We use an LALR(1) parser generated from the Bison since the LALR parser is very fast and small. Bison processes the grammar rules and generates a parser based on OpenQASM2 rules. Then, the parser processes the scanner's token and keeps the information in the Symbol Table. At this phase, the parser also maintains the accuracy

Figure 15. Process of the scanner.

of the syntax. Once everything is done, the parser will generate an AST for semantic analysis. Figure 16 shows the process of the parser.

Figure 16. Process of the parser.

3.4 Semantic Analyzer

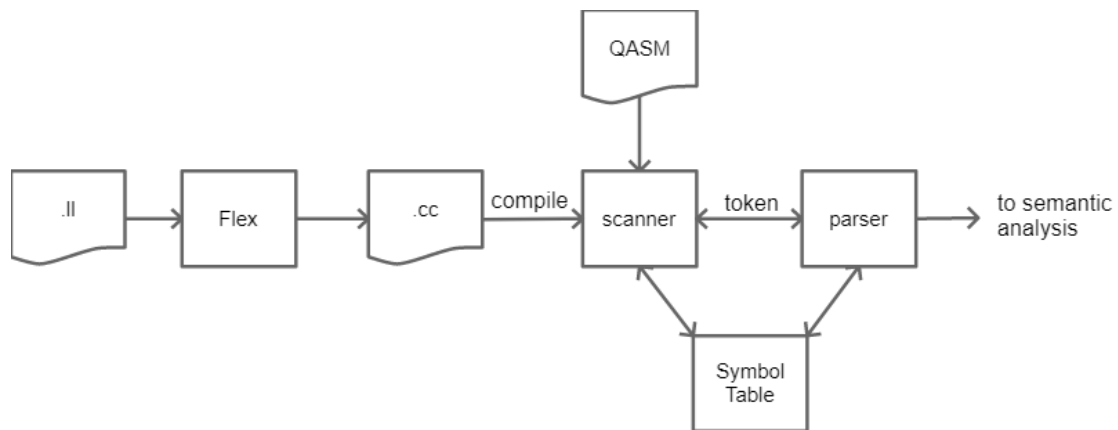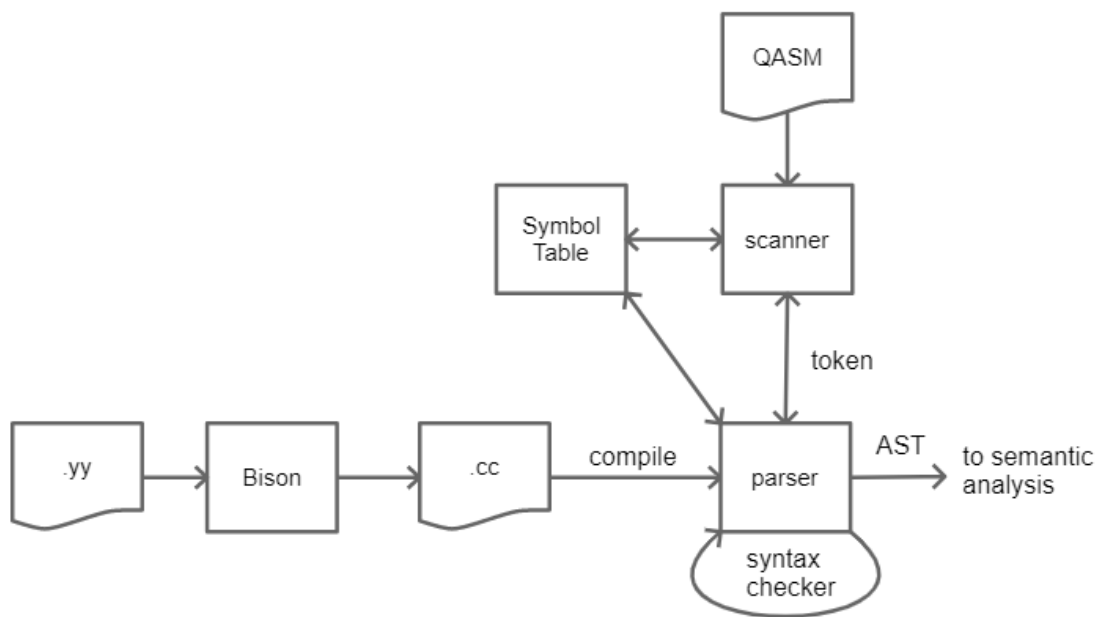In this work, the semantic analyzer consists of three parts: semantic checker, desugaring, and decomposing. This part will ensure the syntax is semantically correct. Figure 17 is the process of the semantic analyzer.
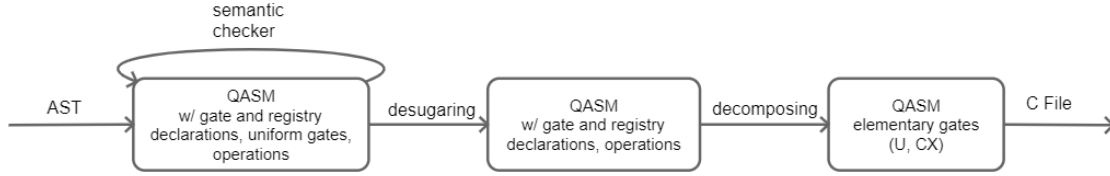
Figure 17. Process of the semantic analyzer.

3.4.1 Semantic Checker

The semantic checker ensures that the uniform gates are well-formed according to the grammar rules in [CBSG17] and other semantic properties such as those used gate/register before the declaration and the suitable argument types. Figure 30 shows the example of syntax errors, and figure 31 shows the example of semantic errors.

3.4.2 Desugaring

After the semantic checker, desugaring is needed to convert the uniform gates (gates applied to registers) to a sequence of gates applied to individual qubits. For example, q is a quantum register of length 2, and the desugar-er will replace x q; with x[0]; and x [1];

3.4.3 Decomposing

After the desugaring, we want to make all the operations be the elementary gate in the QuEST (U and CX). In QuEST, applying a unitary to a qubits needs to be in ComplexMatrix form, i.e., matrix transformation. So, all the gates states in [CBSG17] which have the base of $U(\theta, \phi, \lambda)$, will be converted using this formula:

$$U(\theta, \phi, \lambda) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\phi}\sin\left(\frac{\theta}{2}\right) & e^{i(\lambda+\phi)}\cos\left(\frac{\theta}{2}\right) \end{pmatrix} \tag{3}$$

For example, based on "qelib1.inc", x gate is $u3(\pi, 0, \pi)$, which is a $U(\pi, 0, \pi)$, so:

$$U(\pi, 0, \pi) = \begin{pmatrix} \cos\left(\frac{0}{2}\right) & -e^{i\pi}\sin\left(\frac{0}{2}\right) \\ e^{i\phi}\sin\left(\frac{\pi}{2}\right) & e^{i(\pi+0)}\cos\left(\frac{0}{2}\right) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{4}$$

We break it down into the elementary gates to make it flexible for different compiler back-ends for future development.

3.5 Emitting

Before being compiled by the QuEST, the compiler front-end will emit the AST into a C file using the function from the QuEST library. Figure 35 is an example of the C file that originated from Figure 1. This section will explain the QuEST functions that we are using.

3.5.1 Initialization

The initialization of the QASM program starts with the "OPENQASM M.m;" indicating a major version M and minor version m. After that, it follows with the "include 'qelib1.inc'" where the definitions of other gates are. Then, this code will be translated into "createQuESTEnv ( )" for environment initialization. Figure 18 is the comparison between QASM and QuEST for the initialization.

```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3
```

```
1 // initialize the env for QuEST
2 QuESTEnv env = createQuESTEnv();
3
```

(a) Code in QASM.   (b) Code in C.

Figure 18. Initialization code.

3.5.2 Qubit Declaration

Register declaration (qreg/creg) is pretty straightforward in QASM. It is to specify the type, name and index, e.g., qreg q[2]. However, in QuEST, the register will not be differentiated by the name but by the index, i.e., two registers created in QASM will be considered one register in the QuEST environment. Also, the classical register will come together with the quantum register. See figure 19 for an example.

```
1 qreg q[3];
2 qreg a[3];
3 creg c[3];
4 creg syn[3];
5
```

```
1 // create the qubits into the env
2 Qureg qubits = createQureg(6, env);
3 initZeroState(qubits);
4 ...
5
```

(a) Code in QASM.   (b) Code in C.

Figure 19. Register declaration code.

3.5.3 Unitary Gate

All gates in QASM are declared in the "qelib1.inc" file. We converted all the declared gates in QASM into ComplexMatrix transformation applied to the qubits described in 3. However, in QASM, a gate can be applied to a register or a single qubit. We are desugaring it to handle it in the QuEST. Figure 20 is the conversion from x gate into ComplexMatrix transformation.

```
1 x q;
2
```

```
1 ComplexMatrix2 m = {
2     .real = {{0,1},{1,0}},
3     .imag = {{0,0},{0,0}}};
4 unitary(qubits, 0, m);
5 unitary(qubits, 1, m);
6 unitary(qubits, 2, m);
7
```

(a) Code in QASM.   (b) Code in C.

Figure 20. Applying unitary gate.

3.5.4 CNOT Gate

The *controlled-NOT* (CNOT) gate is obvious in both languages, cx for QASM and controlledNot for QuEST. The first qubit is the control, and the latter is the target qubit. CNOT gate can be applied to a register or a single qubit in the same case as the unitary gate where desugaring is needed. See figure 21 for an example.

```
1 cx q,a;
2
```

```
1 controlledNot(qubits, 0, 3);
2 controlledNot(qubits, 1, 4);
3 controlledNot(qubits, 2, 5);
4
```

(a) Code in QASM.   (b) Code in C.

Figure 21. Applying CNOT gate.

3.5.5 Measurement

In QASM measurement, the source qubit needs to be paired with a classical bit. On the other hand, in QuEST, the classical bit comes together with the qubit. Measurement can be applied to a register or a single qubit as well. See figure 22 for an example.

23

```
1 measure q -> c;
2
```

```
1 int res;
2 res = measure(qubits, 0);
3 res = measure(qubits, 1);
4 res = measure(qubits, 2);
5
```

(a) Code in QASM.

(b) Code in C.

Figure 22. Measurement.

3.5.6 Reset

In QASM, the reset command is to revert the qubits into the ground state with high-fidelity [QT]. However, the reset may leave the remaining qubits in a mixed state in the actual quantum computer. These are some of the techniques for resetting [YT21, ZZY+21, NJ21]. However, we will use measurement and bit-flip in this work in a non-noise environment. If the result of the measurement is 0, then we do nothing. Otherwise, we will apply the X gate.

```
1 reset q[0];
2
```

```
1 res = measure(qubits, 0);
2 ComplexMatrix2 m = {
3     .real = {{0,1},{1,0}},
4     .imag = {{0,0},{0,0}}};
5 if (res == 1) unitary(qubits, 0, m);
6
```

(a) Code in QASM.

(b) Code in C.

Figure 23. Reset.

3.5.7 Barrier

In QASM, a barrier separates the gates to prevent optimizations from reordering gates across its source line. However, there is no optimization yet, and in QuEST, there is no barrier function. We do not implement barrier features in this work. Figure 24a is the example of barrier syntax.

```
1 barrier q;
2
```

```
1 // nothing
2
```

(a) Code in QASM.

(b) Code in C.

Figure 24. Termination.

3.5.8 Termination

In QASM, termination is not needed. However, in QuEST, the qubits and the environment need to be destroyed. See figure 25 for an example.

```
1 // nothing
2
```

```
1 destroyQureg(qubits, env);
2 destroyQuESTEnv(env);
3
```

(a) Code in QASM.　　　　　　　　　　　　　(b) Code in C.

Figure 25. Termination.

3.6 Compiling

Since QuEST is a stand-alone C library, it can be used without installation. To execute the QuEST function, we need to include the library while compiling the C file, such as

cc -o outputfile inputfile.c libQuEST.so -Wl,-rpath=. -lm

where "libQuEST.so" is the QuEST library. Also, since QuEST uses C++ math, we need to add "-lm" while compiling. Figure 26 is the result from the figure 1.

```
root@6d2735aaa12e:/home/OpenQASM/OpenQASM2-Compiler/build# cc -o qec qec.c  libQuEST.so -Wl,-rpath=. -lm
root@6d2735aaa12e:/home/OpenQASM/OpenQASM2-Compiler/build# ./qec
Qubit a[0] collapsed to 1 with probability 1
Qubit a[1] collapsed to 0 with probability 1
Qubit q[0] collapsed to 0 with probability 1
Qubit q[1] collapsed to 0 with probability 1
Qubit q[2] collapsed to 0 with probability 1
root@6d2735aaa12e:/home/OpenQASM/OpenQASM2-Compiler/build#
```

Figure 26. Example of QEC output.

4   Results, Evaluation, Discussion

This section consists of three parts. The first part talks about the results of this work. Then, it describes the evaluation of this project and is followed by the open-issue discussions.

4.1   Results

As described earlier, the *OpenQASM2-Compiler* is a quick, dependable compiler for running an "OpenQASM2" file with the GPU-based quantum simulator, QuEST, into an executable file. This file produces the result of a given quantum circuit in a closed environment.

The compiler front-end, Flex/Bison, generates a C file containing QuEST library elementary gates (U and CX). The compiler has been tested with a curated collection of OpenQASM2 and comes with a great result, i.e., stable and correct.

After the front-end compiling, the generated C files need to be compiled with QuEST shared library. In this work, we do not measure the performance of the QuEST since it has been mentioned here [JBBB19].

Section II will describe the requirements, installation and running procedure.

4.1.1   Run-time Analysis

One of the critical quality criteria is the run-time of the compiler. This section proves that *OpenQASM2-Compiler* is a quick compiler. The longest run-time is 0.19 sec. Table 1 shows the results of compiling a test suite.

Table 1. Results of compiling test suite.

QASM File	Run Time (sec)
Adder	0.01
BigAdder	0.19
InverseQft1	0.02
InverseQft2	0.01
Ipea3Pi8	0.01
Pea3Pi8	0.03
QEC	0.01
QFT	0.02
RB	0.01
Teleport	0.01
TeleportV2	0.02
W-state	0.2

4.1.2 Error Handling

Another essential quality criterion is the ability to handle the errors. This section describes that *OpenQASM2-Compiler* can detect if there is a syntax error or a semantic error. Figure 27 shows an example of a syntax error (source: figure 30), and figure 28 and 29 displays an example of a semantic error (source: figure 31 and 32). The error message tells where it needs to be fixed.

```
../examples/invalid_missing_semicolon.qasm:4.1-4 : syntax error, unexpected qreg, expecting ;
```

Figure 27. Example of an error: "invalid missing semicolon".

```
Error: Compile Error: ../examples/invalid_gate_no_found.qasm:5.1-4, Gate w not found
```

Figure 28. Example of an error: "invalid gate no found".

```
Error: ../examples/invalid_redefined_reg.qasm:3.1-10, symbol 'q' redefined. First defined ../examples/invalid_redefined_reg.qasm:2.1-10
```

Figure 29. Example of an error: "invalid redefined variable".

4.1.3 Testing

OpenQASM2-Compiler includes a unit test to ensure the translation is going well. Also, this test suite is needed to make sure the *OpenQuEST2-Compiler* is configured correctly. See Appendix II for the testing instructions.

4.2 Evaluation

The goal of the present master's thesis project was to develop a compiler for the commonly used quantum-circuit description language "OpenQASM2" into binary code for GPU-accelerated quantum-circuit simulation. Because of this goal, existing simulators were first analyzed. However, since this work is a part of the NordIQuEst project, we use QuEST as the compiler back-end. Another reason is that QuEST is a high-profile, high-quality academic open source project with GPU and MPI support.

From the test suite results, it can be said that the compiler has fulfilled all the requirements needed, i.e., it is quick, stable and correct. The speed is still arguable since the OpenQASM2 program is short. In contrast, the stability and accuracy have been proven due to the fact that it accepts all valid OpenQASM2 files and rejects all

bad OpenQASM2 files. However, there is still room for improvement to provide more explicit error messages.

The connection between the compiler front-end and back-end through the C file is smooth, i.e., no significant lag. However, the translation into C is a detour that was necessary only because deciding to use a Low Level Virtual Machine (LLVM)-based technique was difficult. Meanwhile, the direct pathway would use LLVM IR. LLVM IR may be a dead-end, as techniques such as Multi-Level Intermediate Representations (MLIR in LLVM language) may be a better choice; this point is argued in [MN21].

Compared with the existing state-of-the-art quantum full-stack libraries, *OpenQASM2-Compiler* is a good stepping stone, and it has much potential to grow as a quantum language cross-platform compiler. At the moment, it can only handle one language and one machine, while the others can support cross-platform compilations. At the very least, the compilation for the OpenQASM2 language has the same result.

4.3 Discussion

4.3.1 Restrictions

The compiler was successful in meeting the requirements mentioned at the beginning of the thesis. However, certain restrictions had to be made to avoid going beyond the scope of this work.

- The compiler is only able to handle one language (OpenQASM2) and one simulator (QuEST).

- The compiler was developed and tested only in Linux.

- Flex/Bison has a small memory leak. However, this does not interrupt the main functionality.

Despite these limitations, *OpenQASM2-Compiler* is a fully functional compiler to run quantum circuits in a simulator. Moreover, all of these restrictions can be overcome by further development.

4.3.2 Open Issues

This master's thesis does not treat some open issues. There are four aspects of the compiler that can be improved:

- It is hard for inexperienced users to understand the error messages that occur at this moment in time. This message could be improved by making them more explicit to guide the user when solving the highlighted issues.

- This work simulates the quantum circuit in a closed environment because the noise is defined through the elementary hardware gates. This reason is also why no quantum circuit optimization is performed here. For future improvement, simulating quantum noise may be added.

- A direct pathway using LLVM IR or MLIR(s) is way better than detouring with C. The C-detour must be replaced by using a new toolchain. However, building a new toolchain is a whole topic by itself, and it could be used for another master's thesis. Research between LLVM IR and MLIR(s) is also needed to decide which technique should be used.

- As mentioned in the abstract, this work must be integrated into the Nordic-Estonian Quantum Computing e-Infrastructure Quest (NordIQuEst) project. For that reason, the output format for the measurement statistics must be implemented. However, this only can be done once NordIQuEst has decided on the format.

5 Conclusion and Future Work

This section concludes the journey of this master's thesis. The first part is about the summary of what we have done, followed by the future direction of this project.

5.1 Conclusion

The main objective of the present master's thesis project was to develop a compiler for the commonly used quantum-circuit description language OpenQASM2 into binary code for GPU-accelerated quantum-circuit simulation. The secondary goal was to create an integrative compiler for the NordIQuEst project. In this work, we decided to use Flex/Bison as the compiler front-end because OpenQASM2 has a structured input program. Then, we used QuEST for the compiler back-end because QuEST has GPU and MPI support. The approach for achieving the objective was to divide the compiler into two parts: front-end and back-end, which can solve the $L \times M$ problem.

The first part, the front-end, is concerned mainly with translating the OpenQASM2 language into a simple and meaningful AST, which the back-end will process. This part is essential since it determines the accuracy (validation) and the compiler back-end's performance (desugaring and decomposing).

The second part, the back-end, is focused on transforming the optimized IR into a C-detour program, where it will be compiled using the QuEST library. The connection with QuEST is great because we only apply the elementary gates (U and CX). However, this part can be improved more by creating a new toolchain for making an LLVM.

The goals of this master's thesis were achieved. The created compiler is fully functional, as it is quick, stable, and accurate. It accepts all valid OpenQASM2 files and rejects all bad ones. Also, it produces the C-detour file in an negligible amount of time.

5.2 Future Work

The Quantum computing field has benefited from compiler engineering. With the advancement in this field, new tools like MLIR emerge. Exploring these new tools will replace the "shortcut" of compiler back-ends (quantum circuit simulators) by developing a new toolchain for compiling the circuits in the real quantum devices. This can be added into future development.

Also, in further work, the compiler project should be extended by adding a second stage: At this point, the compiler operates in a single-stage which implies, owing to the flex/bison requirement, that it cannot handle the "include" directive of OpenQASM2. For a future project, the successor language, OpenQASM3 should be addressed.

References

[AA07]   Alfred V. Aho and Alfred V. Aho, editors. *Compilers: principles, techniques, & tools*. Pearson/Addison Wesley, Boston, 2nd ed edition, 2007. OCLC: ocm70775643.

[AG20]   Matthew Amy and Vlad Gheorghiu. staq – A full-stack quantum processing toolkit. *arXiv:1912.06070 [quant-ph]*, August 2020. arXiv: 1912.06070.

[Bis21]   GNU Bison. Bison 3.8.1 documentation. `https://www.gnu.org/software/bison/manual/html_node/index.html`, 2021. Accessed: 2022-04-30.

[BWP+17]   Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, September 2017.

[CBSG17]   Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. *arXiv:1707.03429 [quant-ph]*, July 2017. arXiv: 1707.03429.

[CJAA+22]   Andrew W. Cross, Ali Javadi-Abhari, Thomas Alexander, Niel de Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, page 3505636, March 2022. arXiv: 2104.14722.

[DD13]   Wolfgang Dichler and Heinz Dobler. BoB: Best of Both in Compiler Construction – Bottom-up Parsing with Top-down Semantic Evaluation. *Research Gate*, page 11, 2013.

[Dev21]   Cirq Developers. Cirq, August 2021.

[GAN14]   I. M. Georgescu, S. Ashhab, and Franco Nori. Quantum simulation. *Reviews of Modern Physics*, 86(1):153–185, March 2014.

[GvRB+12]   Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. Springer New York, New York, NY, 2012.

[JBBB19]   Tyson Jones, Anna Brown, Ian Bush, and Simon C. Benjamin. QuEST and High Performance Simulation of Quantum Computers. *Scientific Reports*, 9(1):10736, July 2019.

[JCJ14]   Tomi H Johnson, Stephen R Clark, and Dieter Jaksch. What is a quantum simulator? *EPJ Quantum Technology*, 1(1):10, December 2014.

[KIQ+19]   Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. Strawberry Fields: A Software Platform for Photonic Quantum Computing. *Quantum*, 3:129, March 2019.

[LJL+10]   Thaddeus D. Ladd, Fedor Jelezko, Raymond Laflamme, Yasunobu Nakamura, Christopher Monroe, and Jeremy L. O'Brien. Quantum Computing. *Nature*, 464(7285):45–53, March 2010. arXiv: 1009.2267.

[LL09]   John R. Levine and John R. Levine. *Flex & bison*. O'Reilly, Sebastopol, CA, 1st ed edition, 2009. OCLC: ocn321016664.

[MLD+20]   Alexander J. McCaskey, Dmitry I. Lyakh, Eugene F. Dumitrescu, Sarah S. Powers, and Travis S. Humble. XACC: a system-level software infrastructure for heterogeneous quantum–classical computing. *Quantum Science and Technology*, 5(2):024002, February 2020. Publisher: IOP Publishing.

[MMRP21]   Marco Maronese, Lorenzo Moro, Lorenzo Rocutto, and Enrico Prati. Quantum Compiling. *arXiv:2112.00187 [quant-ph]*, November 2021. arXiv: 2112.00187.

[MN21]   Alexander McCaskey and Thien Nguyen. A MLIR Dialect for Quantum Assembly Languages. *arXiv:2101.11365 [quant-ph]*, January 2021. arXiv: 2101.11365.

[NC10]   Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, Cambridge ; New York, 10th anniversary ed edition, 2010.

[NJ21]   Paul Nation and Blake Johnson. How to measure and reset a qubit in the middle of a circuit execution, February 2021.

[OCR+17]   Jonathan Olson, Yudong Cao, Jonathan Romero, Peter Johnson, Pierre-Luc Dallaire-Demers, Nicolas Sawaya, Prineha Narang, Ian Kivlichan, Michael Wasielewski, and Alán Aspuru-Guzik. Quantum Information and Computation for Chemistry. *arXiv:1706.05413 [physics, physics:quant-ph]*, June 2017. arXiv: 1706.05413.

[PEM16]   Vern Paxon, Will Estes, and John Millaway. Lexical Analysis With Flex, for Flex 2.6.2. `https://westes.github.io/flex/manual/`, 2016. Accessed: 2022-04-30.

[QT]   IBM Q Team.   Conditional Reset on IBM Quantum Systems.
            `https://quantum-computing.ibm.com/lab/docs/iql/manage/`
            `systems/reset/backend_reset`. Accessed: 2022-04-30.

[RGB18]   Patrick Rebentrost, Brajesh Gupt, and Thomas R. Bromley. Quantum com-
            putational finance: Monte Carlo pricing of financial derivatives. *Physical
            Review A*, 98(2):022321, August 2018. arXiv: 1805.00109.

[SEL04]   PETER SELINGER. Towards a quantum programming language. *Mathe-
            matical Structures in Computer Science*, 14(4):527–586, 2004. Publisher:
            Cambridge University Press.

[SHT18]   Damian S. Steiger, Thomas Häner, and Matthias Troyer. ProjectQ: An
            Open Source Software Framework for Quantum Computing. *Quantum*,
            2:49, January 2018. arXiv: 1612.08091.

[SVPO⁺17]   Vadim N. Smelyanskiy, Davide Venturelli, Alejandro Perdomo-Ortiz,
            Sergey Knysh, and Mark I. Dykman. Quantum Annealing via Environment-
            Mediated Quantum Diffusion. *Physical Review Letters*, 118(6):066802,
            February 2017.

[YT21]   T. Yoshioka and J. S. Tsai. Fast unconditional initialization for supercon-
            ducting qubit and resonator using quantum-circuit refrigerator. *Applied
            Physics Letters*, 119(12):124003, September 2021. Publisher: American
            Institute of Physics.

[ZZY⁺21]   Yu Zhou, Zhenxing Zhang, Zelong Yin, Sainan Huai, Xiu Gu, Xiong
            Xu, Jonathan Allcock, Fuming Liu, Guanglei Xi, Qiaonian Yu, Hualiang
            Zhang, Mengyu Zhang, Hekang Li, Xiaohui Song, Zhan Wang, Dongning
            Zheng, Shuoming An, Yarui Zheng, and Shengyu Zhang. Rapid and
            unconditional parametric reset protocol for tunable superconducting qubits.
            Nature Communications, 12(1):5924, December 2021.

[Öm98]   Bernhard Ömer. A procedural formalism for quantum computing. Technical
            report, Department of Theoretical Physics, Technical University of Vienna,
            1998.

Appendix

I. Glossary

Acronyms

AST Abstract Syntax Tree. 7

CPU Central Processing Unit. 2

GLR Generalized LR. 11

GPU Graphics Processing Unit. 2

HPC High Performance Computing. 2

IELR Inadequacy Elimination LR. 11

IR Intermediate Representation. 9

LALR Look-Ahead LR. 11

LLVM Low Level Virtual Machine. 28

MLIR Multi-Level Intermediate Representations. 28

NordIQuEst Nordic-Estonian Quantum Computing e-Infrastructure Quest. 2, 12, 27, 29

OpenQASM Open Quantum Assembly Language. 2, 8

QuEST Quantum Exact Simulation Toolkit. 2, 12

II. User's Manual

Requirements and Installation

Requirements *OpenQASM2-Compiler* project is done in a Linux environment. To compile the project, C++ 9.4+ and Cmake 3.15+ are required. With Cmake, it will install all the necessary dependencies needed. Table 2 shows all the necessary dependencies.

Table 2. Software Requirement

Software	Min Version
C++	9.4+
Flex	2.6+
Bison	3.8+
CMake	3.15+
Make	4.2+

Installation   The easiest way to install *OpenQASM2-Compiler* is by using the Cmake command. First, download *OpenQASM2-Compiler* with git at the terminal

```
clone "https://github.com/HandyKurniawan/OpenQASM2-Compiler.git"
cd OpenQASM2-Compiler
```

Then, at a command line, compile them using the cmake. (Note that cmake automatically handles all dependencies.)

```
mkdir build
cd build
cmake ..
cmake --build .
```

Testing

Once compiled as above, the unit test can be run to test the functionality. We use the ctest feature from cmake:

```
ctest
```

The expected result from the unit test is in figure 36. The resulting message can also be seen by running the ctest with -V option.

Running

After running the unit test and there is no problem, the compiler is ready to use. The compiled executable can be run within the build directory. The command format is OpenQASM2 [file.qasm]. Below is the example for running the program:

```
./OpenQASM2 ../examples/example.qasm
```

The above command will give a C file that needs to be compiled using the QuEST library.

```
cc -o example example.c  libQuEST.so -Wl,-rpath=. -lm
```

Finally, the result from the QASM file can be obtained by running the file generated from the above command (example).

```
1  ./example
```

III. Code

Source code is available on GitHub (https://github.com/HandyKurniawan/OpenQASM2-Compiler). To gain access, please, send a request to handykhandy@gmail.com

IV. Figures

```
1  // name: missing semicolon
2  // section: TODO
3  OPENQASM 2.0
4  qreg q[1];
5  creg c[1];
6  measure q->c;
```

Figure 30. Invalid missing semicolon.

```
1  // name: Gate not found
2  // section: TODO
3  OPENQASM 2.0;
4  qreg q[1];
5  w q;
6  creg c[1];
7  measure q->c;
```

Figure 31. Invalid gate no found.

```
1  OPENQASM 2.0;
2  qreg q[1];
3  qreg q[2];
4  creg c[1];
5  measure q->c;
```

Figure 32. Invalid redefined variable.

36

```
1  OPENQASM 2.0;
2  include "qelib1.inc";
3  qreg q[2];
4  h q[0];
5  cx q[0],q[1];
```

Figure 33. Entanglement.

```
1   // quantum teleportation example
2   OPENQASM 2.0;
3   include "qelib1.inc";
4   qreg q[3];
5   creg c0[1];
6   creg c1[1];
7   creg c2[1];
8   // optional post-rotation for state tomography
9   gate post q { }
10  u3(0.3,0.2,0.1) q[0];
11  h q[1];
12  cx q[1],q[2];
13  barrier q;
14  cx q[0],q[1];
15  h q[0];
16  measure q[0] -> c0[0];
17  measure q[1] -> c1[0];
18  if(c0==1) z q[2];
19  if(c1==1) x q[2];
20  post q[2];
21  measure q[2] -> c2[0];
```

Figure 34. Teleportation.

```c
#include <stdio.h>
#include "../include/QuEST.h"

int main (int narg, char *varg[]) {
  QuESTEnv env = createQuESTEnv();
  Qureg qubits;
  ComplexMatrix2 m;
  int res;
  qreal prob;
  qubits = createQureg(5, env);
  initZeroState(qubits);
  m.real[0][0] = -0.00000004371139000186 ;
  m.imag[0][0] = 0.0000000000000000000 ;
  m.real[0][1] = 0.99999999999999522604 ;
  m.imag[0][1] = 0.0000000874227800372 ;
  m.real[1][0] = 0.99999999999999900080 ;
  m.imag[1][0] = 0.0000000000000000000 ;
  m.real[1][1] = 0.0000000437113900186 ;
  m.imag[1][1] = 0.00000000000000382137 ;
  unitary(qubits, 0, m);
  controlledNot(qubits, 0, 3);
  controlledNot(qubits, 1, 3);
  controlledNot(qubits, 1, 4);
  controlledNot(qubits, 2, 4);
  res = measureWithStats(qubits, 3, &prob);
  printf("Qubit %s collapsed to %d \w prob %g\n", "a[0]", res, prob);
  res = measureWithStats(qubits, 4, &prob);
  printf("Qubit %s collapsed to %d \w prob %g\n", "a[1]", res, prob);
  m.real[0][0] = -0.00000004371139000186 ;
  m.imag[0][0] = 0.0000000000000000000 ;
  m.real[0][1] = 0.99999999999999522604 ;
  m.imag[0][1] = 0.0000000874227800372 ;
  m.real[1][0] = 0.99999999999999900080 ;
  m.imag[1][0] = 0.0000000000000000000 ;
  m.real[1][1] = 0.0000000437113900186 ;
  m.imag[1][1] = 0.00000000000000382137 ;
  unitary(qubits, 0, m);
  res = measureWithStats(qubits, 0, &prob);
  printf("Qubit %s collapsed to %d \w prob %g\n", "q[0]", res, prob);
  res = measureWithStats(qubits, 1, &prob);
  printf("Qubit %s collapsed to %d \w prob %g\n", "q[1]", res, prob);
  res = measureWithStats(qubits, 2, &prob);
  printf("Qubit %s collapsed to %d \w prob %g\n", "q[2]", res, prob);
  destroyQureg(qubits, env);
  destroyQuESTEnv(env);
  return 0;
}
```

Figure 35. break down from QEC QASM.

```
root@6d2735aaa12e:/home/OpenQASM/OpenQASM2-Compiler/build# ctest
Test project /home/OpenQASM/OpenQASM2-Compiler/build
      Start  1: Runs
 1/16 Test  #1: Runs ...........................    Passed    0.02 sec
      Start  2: parseAdder
 2/16 Test  #2: parseAdder ......................    Passed    0.01 sec
      Start  3: parseBigAdder
 3/16 Test  #3: parseBigAdder ...................    Passed    0.19 sec
      Start  4: parseInverseQft1
 4/16 Test  #4: parseInverseQft1 ................    Passed    0.02 sec
      Start  5: parseInverseQft2
 5/16 Test  #5: parseInverseQft2 ................    Passed    0.01 sec
      Start  6: parseIpea3Pi8
 6/16 Test  #6: parseIpea3Pi8 ...................    Passed    0.01 sec
      Start  7: parsePea3Pi8
 7/16 Test  #7: parsePea3Pi8 ....................    Passed    0.03 sec
      Start  8: parseQEC
 8/16 Test  #8: parseQEC ........................    Passed    0.01 sec
      Start  9: parseQFT
 9/16 Test  #9: parseQFT ........................    Passed    0.02 sec
      Start 10: parseQPT
10/16 Test #10: parseQPT ........................    Passed    0.01 sec
      Start 11: parseRB
11/16 Test #11: parseRB .........................    Passed    0.01 sec
      Start 12: parseTeleport
12/16 Test #12: parseTeleport ...................    Passed    0.01 sec
      Start 13: parseTeleportv2
13/16 Test #13: parseTeleportv2 .................    Passed    0.02 sec
      Start 14: parseW-state
14/16 Test #14: parseW-state ....................    Passed    0.02 sec
      Start 15: parseInvalidGate
15/16 Test #15: parseInvalidGate ................    Passed    0.01 sec
      Start 16: parseInvalidSemicolon
16/16 Test #16: parseInvalidSemicolon ...........***Failed    0.00 sec

94% tests passed, 1 tests failed out of 16

Total Test time (real) =   0.42 sec

The following tests FAILED:
        16 - parseInvalidSemicolon (Failed)
Errors while running CTest
```

Figure 36. Unit test result.

39

V. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Handy Kurniawan**,
  *(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Compiler for Quantum Language,
     *(*title of thesis)

   supervised by Dr. Dirk Oliver Theis.
     *(*supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Handy Kurniawan
17/05/2022