SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementing an Intermediate Representation for Quantum Computing based on MLIR

Li-ming Bao

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementing an Intermediate Representation for Quantum Computing based on MLIR

Implementierung einer Intermediate Representation für Quantum Computing basierend auf MLIR

Author:	Li-ming Bao
Supervisor:	Prof. Dr. Helmut Seidl
Advisor:	M.Sc. Yannick Stade
Submission Date:	15.3.2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.3.2023 Li-ming Bao

Abstract

Due to the rising number of applications for quantum computing, many different ideas regarding quantum languages and tools have been proposed. In this thesis, we present and implement an IR in MLIR that is optimized for quantum computing. In the beginning, we explain the core concepts of quantum computing and MLIR and give a short overview of related work in this area of expertise. Afterwards, the details of the implementation and the different features of MLIR are explained. Furthermore, a variety of optimizations for quantum hybrid programs is discussed and implemented. We evaluate the efficiency of the optimization passes by counting the number of used gates during the execution of a test program. The results have shown that the optimization passes do not offer any improvements compared to the base program with no optimizations or other frameworks. Nevertheless, the execution time is nearly constant and independent of the input size. This results in faster execution in comparison to other frameworks that perform optimizations during runtime, depending on the input of the program.

Zusammenfassung

Aufgrund der steigenden Anzahl an Anwendungsmöglichkeiten für Quantencomputing, wurden viele Ideen bezüglich Quantum-Programmiersprachen und Werkzeuge vorgeschlagen. In dieser These präsentieren und implementieren wir eine IR in MLIR, welche optimiert für Quantencomputing ist. Am Anfang erklären wir die Grundkonzepte von Quantum Computing und MLIR, und geben einen kurzen Überblick über verwandte Arbeiten. Anschließend werden die Details der Implementierung vorgestellt und einige Bestandteile von MLIR werden erklärt. Zudem werden einige Vorschläge für Optimierungen in Quantum-Hybrid Programmen diskutiert und implementiert. Wir messen die Effizienz von den Optimierungspässen durch das Zählen der verwendeten R-Gates während der Ausführung eines Testprogramms. Die Ergebnisse zeigen, dass die Optimierungspässe keine Verbesserung gegenüber dem Standardprogramm ohne Optimierungen oder anderen Frameworks bringen. Nichtsdestotrotz ist die Ausführungszeit nahezu konstant und unabhängig von der Eingabegröße. Dadurch ist die Ausführung schneller im Vergleich zu anderen Frameworks, welche Optimierungen während der Laufzeit abhängig von der Eingabe durchführen.

Contents

1. Introduction

Quantum computing is a research field that combines classical computer science with quantum mechanics. By utilizing the quantum mechanical phenomena, quantum computers can compute problems that cannot be simulated efficiently with classical computations [8]. In recent years the number of applications for quantum computing has risen, such as in the areas of cryptography and machine learning [1]. In order to utilize quantum computers, many different quantum languages and tools have been proposed to translate code written by a programmer into instructions that can be understood and executed by quantum computers. Most quantum languages, such as Qiskit or ProjectQ, reuse the existing infrastructure of a widely used programming language and are embedded as a library. They leverage the existing components of the programming language they are embedded in to generate structures that represent quantum circuits. The results of these circuits can either be obtained by sending the data to a quantum computer or by running them through a simulation. Since the data presenting the quantum circuit is written in a classical programming language, these structures can be represented in an intermediate representation (IR), a flat data structure that is designed to be further processed. Since large-scale algorithms for quantum computing require a large number of gates, these algorithms cannot be represented in a simple IR as they would require too much space. For example, simulations in the domain of chemistry require up to 10^{15} gates [9]. Furthermore, the IR must adhere to the no-cloning theorem [10] in quantum computing in quantum mechanics.

In this thesis, we introduce the basic outline of an IR that is specialized in quantum computing using the Multi-Level-Intermediate-Representation framework (MLIR) from LLVM [4] that accommodates quantum operations and optimizations. In the beginning, we explain the basic concepts of quantum computing and MLIR. Then we present related work in this area of expertise. Afterwards, the IR and the requirements for it are described, followed by an explanation of the individual passes that are applied to the IR until execution. These implementation details can be found in Chapter 3. The performance of this IR is then evaluated in Chapter 4 by comparing the number of R-Gates and the controlled versions of them that are used during the execution of Shor's algorithm.

2. Background

We explain the basics of quantum computing and what that means for the development of an IR optimized for quantum computing. Furthermore, a short overview of MLIR, the framework which is used for the implementation of the IR, is given. The last section presents a variety of related work regarding MLIR and quantum computing.

2.1. Quantum Computing

Compared to standard computations that operate on classical bits, quantum computers utilize quantum bits or qubits for computations. A classical bit can only represent either a 1 or a 0 as its value whereas qubits can represent a 1, a 0 or any combination of both states, where either state has a certain probability. This is achieved with the quantum mechanical phenomena of superposition, where the qubit is expressed as a linear combination of both states. This allows qubits to hold more information compared to classical bits. The state of a qubit can be notated as follows

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2.1}$$

where α, $\beta \in \mathbb{C}$ are called probability amplitudes and $|\alpha|^2 + |\beta|^2 = 1$ must hold. The basic states that represent the classical 0 and 1 bit can be expressed as $|0\rangle$ and $|1\rangle$. Multi qubit systems can be represented through the tensor product of the individual qubit states.

In order to change the state of a qubit, a quantum operation has to be applied to the qubit. Quantum gates are unitary operators and are the basic blocks of quantum circuits. They are the equivalent of logical gates in classical computing. These operations can be represented as unitary matrices $M \in \mathbb{C}^{2^n \times 2^n}$ where n denotes the number of qubits the operation acts on. A quantum gate can operate on any number of qubits and returns the same number of quantum states as the number of inputs. This can be expressed as the multiplication of the input quantum state vector and the matrix of the gate:

$$M|\psi_1\rangle = |\psi_2\rangle \tag{2.2}$$

Furthermore, quantum gates are reversible which means that after applying an operation an inverse operation can be applied to reverse the input to its initial state. Gates that are their own inverse operation are known as Hermitian or self-adjoint operators. This property of gates can be used in different optimization steps during the compilation process of a quantum program.

Another peculiarity of quantum computing is the no-cloning theorem of quantum states [10]. The theorem states that it is impossible to create an independent and identical copy of an arbitrary unknown quantum state. Therefore, quantum programs that use multiple copies of the same value are prohibited and are not valid. This must be checked during the validation process of the input or during the runtime of the program.

2.2. MLIR

The Multi-Level Intermediate Representation (MLIR) is a static single assignment (SSA) based IR developed by LLVM and provides an interface to extend the existing compiler infrastructure for domain-specific problems. This allows the user to define their own types and operations customized for their own needs. The SSA property of the IR enforces that each variable must be defined and assigned only once before it is used. The user-defined operations and types are all defined in a dialect which is akin to a namespace and is a collection of operations and types. MLIR features a variety of built-in dialects, such as the *arith* dialect or the *complex* dialect, which support different mathematical operations. These dialects can be used in combination with any other dialects that are defined.

An operation is the equivalent of an instruction in LLVM and is the core unit of abstraction and computation. Every operation has its own semantics and can be used to represent constructs on any level of abstraction.

A program in MLIR is called a module. It consists of several functions that can be called inside the module, thus linking the functions. A function in MLIR is typically represented with the func operation of the *func* dialect and defines a region, which in turn can contain a number of blocks.

A block in MLIR represents a basic block in a compiler. It contains a list of operations that needs to be executed in sequential order. These operations can in turn define another region to support the nesting of operations. The last operation in a block must be a terminator operation that defines, where the control flow of the program goes next. An example operation can be seen below with a description of its components:

```
%result = test.opname %input : i32 -> !test.custom
```

- **%result** is the name of the result of the operation. This part is omitted when the operation provides no results. If the operation provides multiple results, multiple names are required to bind the values to the SSA values. The % symbol is added in front of a name to mark the name as an SSA value.
- **test.opname** is the name of the operation that is performed. The string *test* is the name of the dialect and *opname* is the name of the operation inside the dialect that

is used. The name of the dialect needs to be added as a prefix to the operation name in order to avoid name collisions.

- **%input** is the name of another value that is used as an operand for the current operation. If the operation has multiple inputs, the names of the operands are typically separated by commas.
- **: i32** is the list of the types of the operands. The : symbol marks the beginning and is followed by a list of types. In the case of standard types, such as i32, the type does not need a prefix. The number of types that needs to be listed typically matches the number of operands. Depending on the assembly format of the operation, some types can be inferred and do not need to be listed explicitly. This operation has exactly one operand called *%input* and the type of this value is i32.
- -> **!test.custom** lists the type of the results of the operation and begins with the -> symbol. In the case of user-defined types, a prefix in form of an ! symbol and the name of the dialect needs to be added in front of the name of the type. For this operation, the return type is a *custom* type from the *test* dialect. Similar to the operand type list, the number of types should typically match the number of results but depending on the assembly format of the operation, some result types can be inferred and do not need to be listed explicitly.

A complete function can be seen below that shows a modulo function written in MLIR.

Listing 2.1: Mod function in MLIR

```
1  func.func @mod(%a: i32, %N: i32) -> i32 {
2      %cond_0 = arith.cmpi "uge", %a, %N : i32
3      cf.cond_br %cond_0, ^while(%a:i32), ^ret(%a:i32)
4
5      ^while(%a_0 : i32):
6          %a_1 = arith.subi %a_0, %N : i32
7
8          %cond_1 = arith.cmpi "uge" , %a_1 , %N : i32
9          cf.cond_br %cond_1, ^while(%a_1: i32), ^ret(%a_1: i32)
10
11     ^ret(%res: i32):
12          return %res : i32
13 }
```

MLIR features a variety of tools to help the programmer ranging from transformation hooks and analysis tools to different debugging tools. The programmer can use the in-built hooks for canonicalization and folding for specific operations. Furthermore, general optimizations such as dead code elimination (dce) or common-sub-expression elimination (cse) can be applied to the program. The user can also implement custom optimizations or transformations in user-defined passes. The pattern rewriter of MLIR

allows the user to match any operation and rewrite it. The so-called def-use chains of operations are abstracted in MLIR as direct acyclic graph (DAG) patterns and connect the arguments and the definition of operations. The rewriter traverses these def-use chains through the DAG pattern. To improve the matching of operations, MLIR supports traits that can be attached to any operation. Traits are used as markers for operations, which can be utilized for further optimizations. For general transformation passes or analyses that operate on different dialects, MLIR offers interfaces. Interfaces provide hooks for general transformations and analyses without the knowledge of the involved operation, thus providing a generic way of interacting with the program. An example of an interface is the Inliner-interface that inlines operations which have the CallableOpInterface implemented.

MLIR supports the use of TableGen, which is a generic language that allows the user to maintain records of domain-specific information. It allows the user to define a dialect and its containing operations and types in a table-driven manner. Also, it generates the code for them based on the description inside the TableGen file. It generates all the necessary functions that are needed for the defined operation or type, such as getters/setters, building functions, or verifiers. This removes a lot of boilerplate code that needs to be written and reduces the number of potential errors in the code. Also, all the necessary information regarding a dialect and its operations, including the syntax, can be found in a single file. An example operation written in TableGen can be seen below with a description of the components.

Listing 2.2: Example implementation of an operation in TableGen

```
def HGateOp : Quantum_Op<"H",[HermitianTrait, QuantumTrait,AttrSizedOperandSegments]>{
    let description = [{
    Performs the H Gate Operation on the given input or returns the gate itself if no
     input is given.
    }];
    let arguments = (ins Optional<QuantumType>:$input, Optional<Index>:$index);
    let results = (outs GateType: $output);
    let assemblyFormat = "attr-dict (  $input^ (`[`$index^`]`)? `:` type($input) `->` )
     : (`->`)?  type($output)";

}
```

- **HGateOp** is the name of the C++ class that will be generated. Inside this class, all information and functions for this operation are stored.
- **Quantum_Op** is the name of the parent class of the operation.
- **"H"** is the name of the operation that is used in the syntax of the IR. Typically a prefix of the dialect name must be attached in front of it before the operation can be called.

- **[HermitianTrait,..., AttrSizedOperandsSegments]** are the names of the traits that are attached to the operation. They serve as a marker for certain properties of the operation. For example, the HermitianTrait indicates that the operation is a hermitian gate that can be used for further optimizations later on.
- **description** is used to give a short explanation regarding the operation for anyone who reads this file.
- **arguments** define the required input types for this operation. Some types can also be marked as optional or variadic. The input is usually bound to a name which is marked by the $ prefix.
- **results** define the return types for the operation. Same as arguments, the types can also be marked as optional or variable and are usually bound to a name.
- **assemblyformat** is used to define the syntax of the operation. The bound names of the arguments and the results are used to describe the syntax. Also, if any types are ambiguous or cannot be inferred, then the type must be explicitly stated. For example, in this case, the return type of the operation is GateType which could be either a u1 gate, a qubit, or a qubit register type. Since the return type is not exactly known, it must be explicitly written in the syntax of this operation. An explanation of each type is found in Section 3.2. Furthermore, if-else statements can be used to describe the existence of variadic arguments with the ? operator.

In addition to these building blocks, many more options and statements can be used to define an operation in TableGen. For example, custom builder functions and verifiers can also be defined in TableGen. The CmakeFile of the project can be modified to include the generated files in other files during the compilation process. Furthermore, rewriter patterns can also be written in TableGen and automatically included in the compilation phase of the optimizer. For further information, please refer to the official documentation of MLIR.[1]

2.3. Related Work

In recent years, few efforts have been made to utilize the MLIR framework for a quantum computing based IR. The Quantum Intermediate Representation for Optimization (QIRO) was introduced by David Ittah et al. [2] and was the main reference material for this thesis. They used the MLIR framework to define two different dialects, one for the translation of the input language and one for the optimization of the input dialect. The optimization dialect is specifically designed for quantum-classical co-optimization that utilizes the in-built rewriter patterns and transformation features in MLIR.

[1] MLIR documentation `https://mlir.llvm.org/docs/`

A similar IR was developed and introduced by Peduri et al. called QSSA [7]. They implemented their IR in MLIR based on SSA in order to leverage existing compiler optimizations and focused on verifying the physical constraints of quantum computing, such as the no-cloning theorem.

Another project that used the MLIR framework is presented by McCaskey et al. [5]. They focused on the translation of quantum programs written in QASM into their defined quantum dialect and then into LLVM IR. The output was done with the Quantum Intermediate Representation (QIR) developed by Microsoft [2] . This LLVM based IR supports Q# and serves as a common interface for quantum languages and leverages the existing LLVM compiler infrastructure.

[2]QIR `https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/`

3. Implementation

The focus of this chapter is the quantum dialect implementation in MLIR. In the beginning, we list the requirements for our proposed IR. Afterwards, the implemented types, operations, and passes until the execution of the program are described in detail.

3.1. Requirements

The proposed IR is implemented in a single dialect called *quantum* as opposed to the implementation of David Ittah et al. [2] who implemented their IR in two different dialects, one for the input and one for optimization. Also, it should fulfill the following requirements:

- Support common operations and types to construct valid quantum circuits, such as the allocation of qubits and qubit registers, and operations to manipulate them.
- All operations apart from the memory operations should be free from side effects and work on value-semantics
- Work in conjunction with classical programs
- Allow the use of classical and quantum-computing specific optimizations
- Provide a resource counter for resource estimation.

In order to ease the difficulty of lowering a quantum programming language to the IR, the dialect should feature an equivalent to the most common quantum operations. This includes a set of native gates that can be used. Furthermore, all operations aside from the memory operations must be free from side effects. In order to achieve that, the operations must consume the quantum state of the qubit and return a new quantum state. In addition, values should only be used once to enforce the no-cloning theorem of quantum computing. Also, the dataflow graph is made explicit in the IR, which allows the traversal of the use-def chain of the IR, thus allowing further optimizations. The user should be able to execute the program and get a resource estimation of the used operations. The quantum operations themselves, however, do not perform any computations. They are only used for demonstration purposes in this IR and are either replaced or eliminated before the execution of the program.

3.2. Types

The existing types in the quantum dialect are shown in table 3.1. The qubit type represents a single qubit in the dialect. It can be either created through an alloc operation or returned from a quantum operation, that manipulates the state of a qubit. The qubit register is similar to an array and holds n different qubits. The user can access the qubits inside the register with the subscript operator. The native single- and two-qubit gate types are given to the set of native quantum gates that are implemented in this dialect. The circ type represents the type of a circuit in this implementation. Lastly, the controlled operation type holds information about the base type of the operation it controls and the number of control bits that are used.

Table 3.1.: List of types defined in the quantum dialect

Type	Syntax
Qubit	!quantum.qubit
Qubit Register	!quantum.qureg<n>
Native 1-Qubit Gate	!quantum.u1
Native 2-Qubit Gate	!quantum.u2
Circuit	!quantum.circ
Controlled Operation	!quantum.cop<n, baseT>

3.3. Operations

The quantum operations are split up into four different categories, memory operations, gate operations, circuit operations, and meta operations. Table 3.2 shows all operations that we implemented and their syntax in the IR.

3.3.1. Memory Operations

The two alloc operations initialize and return the state of a qubit or a qubit register in the $|0\rangle$ state. The two free operations are used to free the resource of qubits and destroy the quantum states they hold. The user must explicitly call these operations to free the quantum resources. The measure operation takes either a qubit or a register as its input and returns a single bit for each qubit as its return value. The extract and combine operations are used internally to manipulate the quantum states of the qubits inside a register, that is accessed with the subscript operation. The qubits must first be extracted from the register, and then the given operation is performed on the extracted qubits returning new quantum state values. Finally, the new values are combined with the remaining qubit register to create a new qubit register with the updated quantum state values.

3.3.2. Gate Operations

The set of native single- and two-qubit gates implemented in this IR can be seen in table 3.2. All the gates except the SWAP-Gates are overloaded to take either a direct qubit, a qubit register, or a qubit from a register via the subscript accessor. In the case of the qubit register, the given gate operation is applied to all of the qubits inside the register as their inputs. Furthermore, if no qubit is given as an argument then the gate itself is returned as the result of the operation. The rotation gates must also take a float as input that denotes the degree of the rotation. The input can either be given as another SSA value or directly as a constant attribute.

3.3.3. Circuit Operations

The circuit operation is equivalent to a normal function but for quantum computing. They can take any number of arguments of any type and equally return any number of results of any type. The body of the circuit can hold a mix of classical and quantum operations, but they can only access the values created inside the circuit or the arguments. Furthermore, the last operation inside a circuit must be the return operation from the quantum dialect, which acts as the terminator operation. The circuit can either be called explicitly through the call operation of the quantum dialect, which uses the symbol name of the circuit or implicitly through the SSA value of the circuit. The SSA value of a circuit is obtained through the getval operation, which takes the symbol name of the circuit as its input. The returned SSA value of a circuit can be modified with meta operations and subsequently used with the apply operation.

3.3.4. Meta Operations

Meta operations are used to modify existing operations in the IR. The inputs of the meta operations are either a native gate or a circuit SSA value. The adjoint operation takes the input operation and reverses it. In the case of a native gate, the adjoint operation takes the input qubits that the reversed gate has been applied to as an additional input and applies the operation immediately. If the input is a circuit, then the reversed circuit has to be explicitly called with the apply operation that takes the input of the reversed circuit operation and returns the result of it. Similarly, the controlled meta operation creates a controlled variant of the input operation and takes a control bit as an additional argument. The operation will only be applied if the bit is in the $|1\rangle$ state, otherwise no operation is performed.

Table 3.2.: List of operations defined in the quantum dialect

Operation	Syntax
Alloc Qubit	quantum.alloc -> !quantum.qb
Alloc register	quantum.allocreg<n> -> !quantum.qureg<>
Free Qubit	quantum.free %qb
Free register	quantum.freereg %r
Measurement	quantum.meas %qb -> u1 quantum.meas %r -> memref<?xu1>
Extract	quantum.extract %r [i...] : !quantum.qureg<> ->!quantum.qureg<>, !quantum.qb...
Combine	quantum.combine %r[i...], %qb ... :!quantum.qureg<>, !quantum.qb... -> !quantum.qureg<>
Native u1 gates	quantum.H/X/Y/Z/S/T %qb : !quantum.qb -> !quantum.qb quantum.H/X/Y/Z/S/T %r : !quantum.qureg<> -> !quantum.qureg<> quantum.H/X/Y/Z/S/T -> !quantum.u1
Native u1 rotation gates	quantum.R/Rx/Ry/Rz (ϕ) %qb : !quantum.qb -> !quantum.qb quantum.R/Rx/Ry/Rz (ϕ) %r : !quantum.qureg<> -> !quantum.qureg<> quantum.R/Rx/Ry/Rz (ϕ) -> !quantum.u1
CNOT-Gate	quantum.CX %qb, %q : !quantum.qb , !quantum.qb -> !quantum.qb quantum.CX %qb, %q : !quantum.qb , !quantum.qureg<> -> !quantum.qureg<>
Swap-Gate	quantum.SWAP %qb, %qb : !quantum.qb, !quantum.qb
Controlled op	quantum.ctrl %op , %qb , %qb : !quantum.qb, !quantum.qb, !quantum.u1 -> !quantum.qb quantum.ctrl %circ , %qb : !quantum.circ, !quantum.qb -> !quantum.cop<1, !quantum.circ>
Adjoint op	quantum.adj %op : !quantum.u1, !quantum.qb -> !quantum.qb quantum.adj %circ : !quantum.circ -> !quantum.circ
Circuit op	quantum.circ @name (%arg : type(%arg)...) ...
Call op	quantum.call @name (%arg...) : type(arg)...
Getval op	quantum.getval @name -> !quantum.circ
Apply op	quantum.apply %circ (%arg...) : type(%arg)...

3.4. Transformation Passes

Before we can apply the optimization passes to the IR, we must decompose the meta operations into a simpler set of operations that are supported by the target machine. Furthermore, operations with a register access to a qubit register must be modified with a combination of extract and combine operations. In order to apply the changes to the IR, the pattern rewrite infrastructure from MLIR is used. This infrastructure allows the user to create static optimization and transformation passes for a given input program. The pattern rewriter walks through the DAG of the IR and matches any operations that are named as the root operations of the rewriter pattern. If an operation is successfully matched, then the changes specified in the rewrite function are applied to modify the operation and subsequently to the DAG of the IR.

3.4.1. Register Access Lowering

In order to lower register accesses to qubit registers for side effect free operations, the pattern rewriter is set to match any operation that can take a qubit as an input. This applies to the measure operation, most of the native gates, meta operations and call operations. After an operation is successfully matched, the root operation is inspected

whether the input includes a qubit register with an index or not. If it contains a register access, then the operation has to be changed. A new extract operation has to be inserted before the matched operation. The extract operation takes the qubit register and the indices of the qubits that need to be extracted as its inputs. As its result, it returns the requested qubits and the remaining qubit register from where the qubits were extracted. The matched operation itself must be replaced by the same operation but instead of using the qubits from the register access, the qubits obtained from the extract operation are used as the input. Afterwards, a combine operation has to be inserted after the modified operation that takes the result qubits of the previous operation and the remaining qubit register from the extract operation to complete the register again with the modified qubits. The subsequent uses of the register must be changed to use the result of the combine operation.

Listing 3.1: Example of register access lowering

```
quantum.circ @lowerRegisterAccess() {
    %c0 = arith.constant 2 : index
    %c1 = arith.constant 5 : index
    %qr = quantum.allocreg(%c1) -> !quantum.qureg<>
    %ret0 = quantum.H %qr [%c0] : !quantum.qureg<> -> !quantum.qb
    quantum.return
}
```

⇓

```
quantum.circ @lowerRegisterAccess() {
    %c0 = arith.constant 2 : index
    %c1 = arith.constant 5 : index
    %qr = quantum.allocreg(%c1) -> !quantum.qureg<>
    %remainder, %ex0 = quantum.extract %qr[%c0] : !quantum.qureg<> -> !quantum.qureg<>,
     !quantum.qb
    %ret0 = quantum.H %ex0 : !quantum.qb -> !quantum.qb
    %combined = quantum.combine %remainder[%c0], %ret0 : !quantum.qureg<>, !quantum.qb
     -> !quantum.qureg<>
    quantum.return
}
```

3.4.2. Adjoint Operation Lowering

The adjoint operation is lowered by matching two different operations, the adjoint operation and the apply operation. In the case of the adjoint operation, the input must be checked to see if it is a native gate that has the hermitian trait. If this is the case, then no changes need to be applied as gates with the hermitian trait are self-inverse and are not modified by the adjoint operation. They are replaced by the native gate operation itself. Otherwise, no changes need to be done when matching the adjoint operation.

If an apply operation is matched, then the input of the operation has to be checked whether it is the result of an adjoint operation on a circuit or not. In such cases, a new circuit has to be generated and the apply operation needs to be replaced with a call operation to the newly generated circuit. The new circuit has the same function signature and operations as the circuit that the adjoint operation is applied to. Also, all quantum operations inside the newly generated circuit need to be reversed. Special care has to be taken by operations that are dependent on each other. For example, the extract and combine operation surrounding a root operation are bound to the operation. They do not need to be reversed and are only moved together with the root operation. Furthermore, if and for loops from the *scf* dialect that contain any quantum operations are counted as quantum operations and need to be reversed. Loops also have to iterate from the upper bound to the lower bound of their range. Lastly, the sign of all degrees of the rotation gates needs to be flipped as the rotation is applied to the inverse direction. If the degree is given as a constant attribute to the rotation, then the change will be applied directly. Otherwise, a new constant operation with the value -1 has to be inserted before the rotate operation. The new SSA value from the newly created constant operation is used in combination with the SSA value of the degree to create a new multiplication operation that takes both values as input and returns the result of the multiplication. This result is used to construct a new rotation gate with the negated degree. The usage of the previous rotation gate needs to be replaced with the new gate and the old gate is removed afterwards.

Listing 3.2: Example of adjoint op lowering

```
quantum.circ @example ( %qb0 : !quantum.qb) -> !quantum.qb {
    %c0 = arith.constant 13.5 : f64
    %q0 = quantum.R(%c0 : f64) %qb0 : !quantum.qb -> !quantum.qb
    %q1 = quantum.H %q0 : !quantum.qb -> !quantum.qb
    quantum.return %q1 : !quantum.qb
}

quantum.circ @main() {
    %qb = quantum.alloc -> !quantum.qb
    %circ = quantum.getval @example -> !quantum.circ
    %adj = quantum.adj %circ : !quantum.circ -> !quantum.circ
    %ret = quantum.apply %adj(%qb) : !quantum.circ (!quantum.qb) -> !quantum.qb
    quantum.return
}
```

⇓

```
quantum.circ @example ( %qb0 : !quantum.qb) -> !quantum.qb {
    %c0 = arith.constant 13.5 : f64
    %q0 = quantum.R(%c0 : f64) %qb0 : !quantum.qb -> !quantum.qb
    %q1 = quantum.H %q0 : !quantum.qb -> !quantum.qb
    quantum.return %q1 : !quantum.qb
```

```
}

quantum.circ @exampleAdj ( %qb0 : !quantum.qb) -> !quantum.qb {
    %c0 = arith.constant 13.5 : f64
    %c1 = arith.constant -1.0 : f64
    %c2 = arith.mulf %c0, %c1 : f64
    %q0 = quantum.H %qb0 : !quantum.qb -> !quantum.qb
    %q1 = quantum.R(%c2 : f64) %q0 : !quantum.qb -> !quantum.qb
    quantum.return %q1 : !quantum.qb
}

quantum.circ @main() {
    %qb = quantum.alloc -> !quantum.qb
    %ret = quantum.call @exampleAdj (%qb0) : (!quantum.qb) -> !quantum.qb
    quantum.return
}
```

3.4.3. Controlled Operation Lowering

Similar to the adjoint operation lowering, the controlled operation is lowered by matching two different operations, the controlled operation and the apply operation. If the controlled operation takes a native gate as input, the operation is replaced by itself and a special attribute is attached to the operation to indicate the number of control bits that are used to control the operation.

If an apply operation is matched, then the input of the operation has to be checked if it is the result of a controlled operation on a circuit or a basic gate. If the check is successful, a new circuit needs to be generated, containing the same operations as the circuit that the controlled operation is applied to. The signature of the new circuit operation needs to be modified to include the control bit as an additional parameter. Also, a special attribute is attached to the circuit function to indicate the number of control bits for the circuit. This counter is later used for resource estimation. The initial apply operation is then replaced by a call operation for the new circuit including the control bit as an additional argument. The control bit must then be propagated to the operations inside the controlled circuit. All call operations and meta operations inside the controlled operations need to be replaced to take the control bit as an additional argument. In some cases, new circuits need to be created. This must be done recursively to propagate the control operations. In the case of a controlled operation inside a controlled circuit, the counter attribute of the newly generated circuit needs to be increased accordingly, including the subsequent operations inside the circuit.

Listing 3.3: Example of controlled op lowering

```
quantum.circ @example ( %qb0 : !quantum.qb) -> !quantum.qb {
    %c0 = arith.constant 13.5 : f64
    %q0 = quantum.R(%c0 : f64) %qb0 : !quantum.qb -> !quantum.qb
    %q1 = quantum.H %q0 : !quantum.qb -> !quantum.qb
```

```
    quantum.return %q1 : !quantum.qb
}

quantum.circ @main() {
    %qb = quantum.alloc -> !quantum.qb
    %cqb = quantum.alloc -> !quantum.qb
    %circ = quantum.getval @example -> !quantum.circ
    %ctrl = quantum.ctrl %circ, %qcb : !quantum.circ, !quantum.qb -> !quantum.cop<1, !quantum.circ>
    %ret = quantum.apply %ctrl(%qb) : !quantum.cop<1, !quantum.circ> (!quantum.qb) -> !quantum.qb
    quantum.return
}
```

```
quantum.circ @example ( %qb0 : !quantum.qb) -> !quantum.qb {
    %c0 = arith.constant 13.5 : f64
    %q0 = quantum.R(%c0 : f64) %qb0 : !quantum.qb -> !quantum.qb
    %q1 = quantum.H %q0 : !quantum.qb -> !quantum.qb
    quantum.return %q1 : !quantum.qb
}

quantum.circ @exampleCtrl ( %qb0 : !quantum.qb, %qcb : !quantum.qb) -> !quantum.qb {ctrlbit = 1} {
    %c0 = arith.constant 13.5 : f64
    %q0 = quantum.R(%c0 : f64) %qb0 : !quantum.qb -> !quantum.qb
    %q1 = quantum.H %q0 : !quantum.qb -> !quantum.qb
    quantum.return %q1 : !quantum.qb
}

quantum.circ @main() {
    %qb = quantum.alloc -> !quantum.qb
    %cqb = quantum.alloc -> !quantum.qb
    %ret = quantum.call @exampleCtrl (%qb, %cqb) : (!quantum.qb, !quantum.qb) -> !quantum.qb
    quantum.return
}
```

3.5. Optimization Passes

The MLIR infrastructure allows for static optimization of hybrid programs so that classical and quantum optimizations can be applied to the IR. The same procedure of matching and rewriting can be applied to perform optimizations. Furthermore, a number of classical optimizations, such as canonicalization and inlining, are already implemented for the classical parts of the IR and can be applied to the program. By reusing these optimization passes, we can also apply the optimizations to the quantum part of the program.

3.5.1. Classical Optimization

The canonicalization pass from MLIR performs several optimizations to the base program and is used to convert the IR into a canonical form. During the canonicalization pass, optimizations, such as constant folding or dead code elimination, are performed. We can apply the pass to user-defined operations by utilizing the canonicalization hook provided by the framework. The dead code elimination pass eliminates all unused operations where the result of the operation is not used and the operation itself is free from side effects. Due to the design of the IR, all quantum operations except the memory operations are free from side effects. The result is that some quantum operations will be eliminated when their results are not used.

Another classical optimization that we can apply to the quantum dialect is Inlining. Similar to normal functions that can be inlined, circuits can also be inlined to speed up the program by reducing the number of jumps that are needed. Furthermore, by inlining circuits, further optimizations can be applied to the new sequence of operations that were not possible earlier. The infrastructure offers the inlining interface so that the inlining feature can easily be extended to circuits and quantum calls.

3.5.2. Quantum Optimization

In combination with classical optimizations, certain optimizations can be applied to the quantum part of the program with the MLIR infrastructure. By utilizing the hermitian and unitary traits of native gates and circuits, we can perform a number of peephole optimizations.

Hermitian Gate Cancellation

Gates that have the hermitian trait are self-inverse and cancel out each other. This means that adjacent gates of the same type in the use-def chain can be removed from the IR. The pattern rewriter matches any operations that carry the hermitian trait in the IR. By following the use-def chain of the operations and the inputs, the rewriter verifies if the origin of the inputs is the result of an operation of the same kind as the operation that was matched. In the case of a successful match of two operations, the usage of the result of the second operation is replaced by the input of the first operation. Afterward, both operations will be erased from the IR.

Listing 3.4: Example of hermitian gate cancellation

```
1 quantum.circ @example ( %qb0 : !quantum.qb, %qb1 : !quantum.qb) -> !quantum.qb, !
      quantum.qb {
2     %q0, %q1 = quantum.SWAP %qb0, %qb1 : !quantum.qb, !quantum.qb -> !quantum.qb, !
      quantum.qb
3     %q2, %q3 = quantum.SWAP %q0, %q1 : !quantum.qb, !quantum.qb -> !quantum.qb, !
      quantum.qb
4     quantum.return %q2, %q3 : !quantum.qb, !quantum.qb
```

```
5 }
```

```
1 quantum.circ @example ( %qb0 : !quantum.qb, %qb1 : !quantum.qb) -> !quantum.qb, !
      quantum.qb {
2     quantum.return %qb0, %qb1 : !quantum.qb, !quantum.qb
3 }
```

Unitary Gate Cancellation

The same optimization can be applied to adjoint operations and unitary gates. For example, if the adjoint operation takes as input an unitary gate and a qubit that is the result of the same unitary gate operation, then both operations cancel out each other and will be erased. The same matching and verifying conditions apply as in the case above. The usage of the result of the adjoint operation is replaced with the input of the unitary gate operation that returned the result for the adjoint operation.

Listing 3.5: Example of unitary gate cancellation

```
1 quantum.circ @example ( %qb0 : !quantum.qb) -> !quantum.qb {
2     %c0 = quantum.S %qb0 : !quantum.qb -> !quantum.qb
3     %gate = quantum.S -> !quantum.u1
4     %adj = quantum.adj %gate, %q0 : !quantum.u1, !quantum.qb -> !quantum.qb
5     quantum.return %adj : !quantum.qb
6 }
```

```
1 quantum.circ @example ( %qb0 : !quantum.qb) -> !quantum.qb {
2     quantum.return %qb0 : !quantum.qb
3 }
```

Rotation Gate Folding

Another optimization that is equivalent to constant folding for classical programs is the merging of rotation gates. Two subsequent rotation gates of the same kind where the input of one gate is the result of the other gate can be combined into a single rotation gate. Since the rotation gates are overloaded so that they can either get the degree as a constant attribute or as an SSA value, we need to consider three different cases when merging the gates.

The first case is when the degree of both gates are stemming from SSA values. In this case, a new add operation has to be inserted that takes both SSA values of the degrees as its input. Afterwards, a new rotation gate of the same kind needs to be inserted that

takes the result of the add operation and the input of the first rotation gate as operands. The usage of the result of the second rotation is then replaced by the result of the new rotation gate operation and the two former gates are removed.

The second case is when the degree of one gate is an attribute and the other one is an SSA value. A new constant operation has to be inserted before the matched gates that uses the value of the attribute as an operand. Subsequently, the same steps as above are performed using the SSA value from the rotation gate and the newly created SSA value from the constant operation.

The last case happens when both rotation gates have attributes for their degrees. In this case, the value of the first rotation gate in the sequence is modified to be the sum of both degrees. The usage of the second rotation gate is replaced with the result of the first gate and the second gate is erased.

Listing 3.6: Example of rotation gate folding

```
quantum.circ @example1 ( %qb0 : !quantum.qb) -> !quantum.qb {
    %f0 = arith.constant 20.4 : f64
    %f1 = arith.constant 30.5 : f64
    %q0 = quantum.R (%f0 : f64) %qb0 : !quantum.qb -> !quantum.qb
    %q1 = quantum.R (%f1 : f64) %q0 : %quantum.qb -> !quantum.qb
    quantum.return %q1 : !quantum.qb
}
quantum.circ @example2 ( %qb0 : !quantum.qb) -> !quantum.qb {
    %q0 = quantum.R (20.4) %qb0 : !quantum.qb -> !quantum.qb
    %q1 = quantum.R (30.5) %q0 : %quantum.qb -> !quantum.qb
    quantum.return %q1 : !quantum.qb
}
```

```
quantum.circ @example1 ( %qb0 : !quantum.qb) -> !quantum.qb {
    %f0 = arith.constant 20.4 : f64
    %f1 = arith.constant 30.5 : f64
    %f2 = arith.addf %f0, %f1 : f64
    %q0 = quantum.R (%f2 : f64) %qb0 : %quantum.qb -> !quantum.qb
    quantum.return %q0 : !quantum.qb
}
quantum.circ @example2 ( %qb0 : !quantum.qb) -> !quantum.qb {
    %q0 = quantum.R (50.9) %qb0 : !quantum.qb -> !quantum.qb
    quantum.return %q0 : !quantum.qb
}
```

3.6. Resource Estimation

In order to evaluate the efficiency of the optimization passes, we implemented a resource counter to count the number of operations that are used during the execution

of the program. Similar to the evaluation process of Ittah et al. [2] the number of R-Gates and the controlled versions of them are used to compare the efficiency of the IR.

The first step of the resource estimation pass is to insert a global variable from the *memref* dialect at the beginning of the program. The variable holds the number of R-Gates and its controlled variants that are used during the execution. Afterwards, the pattern rewriter is utilized again to match all R-Gate operations. The rotation gates are then replaced by a combination of load-increment-store operation of the global variable to increase the gate count every time the operation is executed. Special attention has to be paid to controlled versions of the operation or operations that reside in a controlled operation. R-Gate operations that have a special attribute attached or are inside a controlled circuit need to increment the appropriate counter as the global variable stores the count of rotation gates separately, depending on the number of control bits. The number of rotation gates is printed during the execution of the program with the use of the print operation from the *vector* dialect.

3.7. Lowering Process

In order to transform the program into an executable, we need to lower the program to the LLVM IR. As the first step, all quantum operations need to be removed while maintaining the integrity of the program. The measuring operations need to be replaced with their estimated outcome. Normally, the measuring of a qubit returns a bit value of either 1 or 0. Each value is returned with a certain probability given by the probability amplitudes of the qubit. For this IR, the result of the measurement is always set to 1 so that all measure operations return an u1 value of 1 or an array of u1 values with 1 as their value. The circuits and the quantum call operations are replaced with the built-in func and call operations from the *func* dialect. The quantum call operations are matched and their matching circuits are found. An equivalent function operation of the circuit needs to be created with the func operation in the *func* dialect. The func operation should hold the same operations and arguments as the original circuit. Afterwards, all quantum operations are removed from the *func* operation in post-order so that the use-def chain of the operations is not violated. Then, the arguments of the newly created function need to be modified so that no argument has a quantum type. Lastly, the matched call operation is replaced with a standard call operation to the new function. An example of the lowering process can be seen in listing 3.7. The initial circuit may not be removed yet as multiple circuits can call a given circuit and an operation will only be removed when it is used by no other operation.

Listing 3.7: Example of the circ lowering process

```
quantum.circ @example ( %C : i64 , %r : !quantum.qureg<>, %n : index ) -> !quantum.
    qureg<> {
  %cm1 = arith.constant -1 : i64
  %mC = arith.muli %C , %cm1 : i64
  %r0 = quantum.H %r : !quantum.qureg<> -> !quantum.qureg<>
  %r1 = quantum.call @addConstant ( %mC , %r0 , %n ) : i64 , !quantum.qureg<>, index ->
      !quantum.qureg<>
  quantum.return %r1 -> !quantum.qureg<>
}
```

```
func.func @examplefunc ( %C : i64 , %n : index ) {
  %cm1 = arith.constant -1 : i64
  %mC = arith.muli %C , %cm1 : i64
  func.call @addConstantfunc ( %mC ,%n ) : i64 , index
  func.return
}
```

3.8. Execution

After the removal of all quantum operations, the program only consists of operations from built-in dialects including the resource counter for resource estimation. This program can be lowered with the built-in conversion patterns of MLIR to transform the program in the *llvm* dialect. We can then dump the *llvm* dialect into the LLVM IR. Finally, we can execute the LLVM IR with a LLVM interpreter such as lli. As the result of the execution, the number of R-Gates are printed, separated by the number of control bits that control the operation.

The graphic below illustrates the pass sequence that is applied to the base program written in MLIR until the execution of the program.

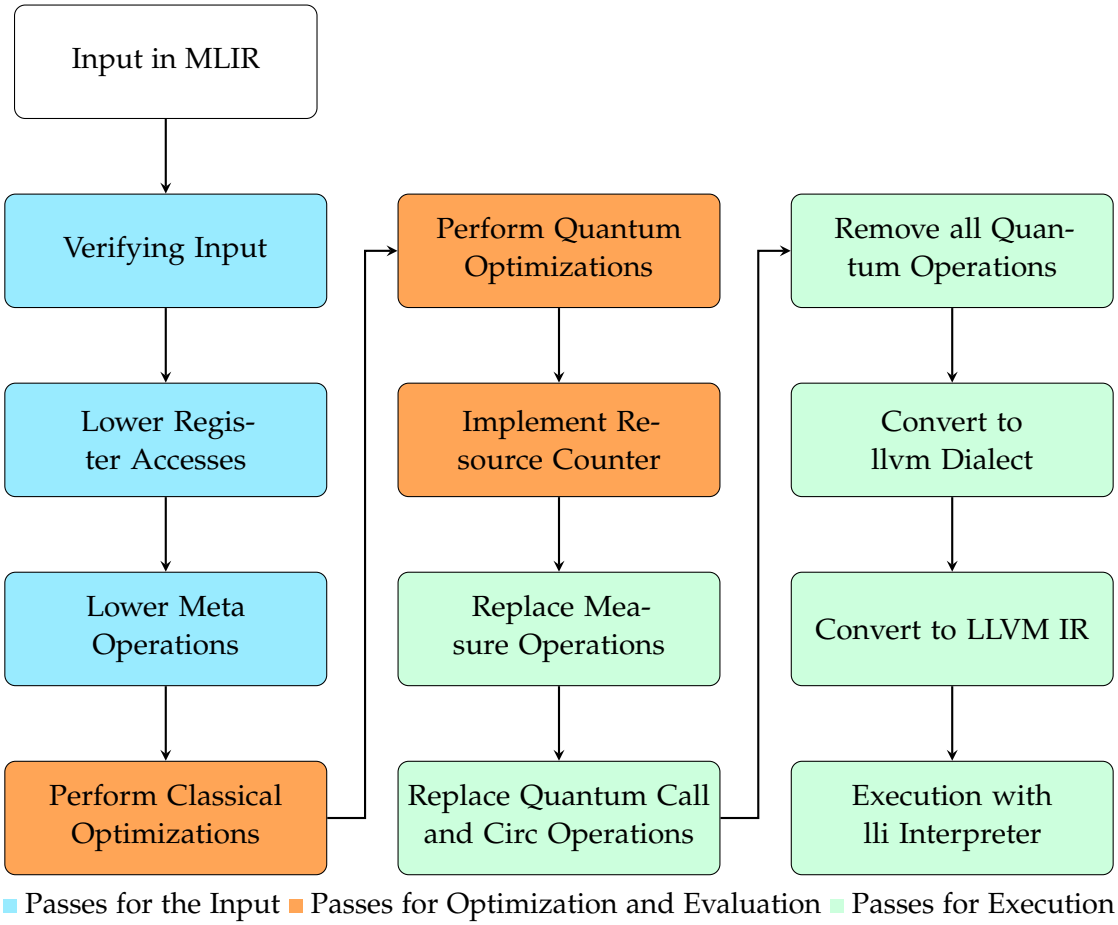

Figure 3.1.: Flowchart of the different passes in the IR

4. Evaluation

In this chapter, the evaluation process and the results of the proposed IR are presented. The IR is tested with an implementation of Shor's algorithm in MLIR. The code of the implementation in MLIR can be found in the appendix under A.1.

4.1. Testsuite

In order to test the implementation of the proposed quantum IR, we used the number of R-Gate that are used during the execution of Shor's algorithm and the controlled versions of them. We then used these numbers to measure the effectiveness of the optimizations. The implementation of Shor's algorithm can be found in the appendix of this work. We compared the optimized program against the unoptimized version of it to show the efficiency of the implemented optimization passes. Both programs used the same tools to compile the MLIR implementation of Shor's algorithm and are executed with the lli tool from LLVM. In contrast to the unoptimized program, the optimized one used the three quantum optimization passes mentioned in section 3.5. Furthermore, we used an implementation of Shor's algorithm in ProjectQ as an additional comparison. The ProjectQ framework was selected as it has a built-in resource counter for the number of used gates. The code for Shor's algorithm in ProjectQ stems from the ProjectQ repository on GitHub[1]. In addition, we had a look at the results obtained in QIRO as they were the main reference for this IR and therefore used a similar implementation as the one we proposed in this thesis.

[1]ProjectQ repository `https://github.com/ProjectQ-Framework/ProjectQ`

4.2. Results

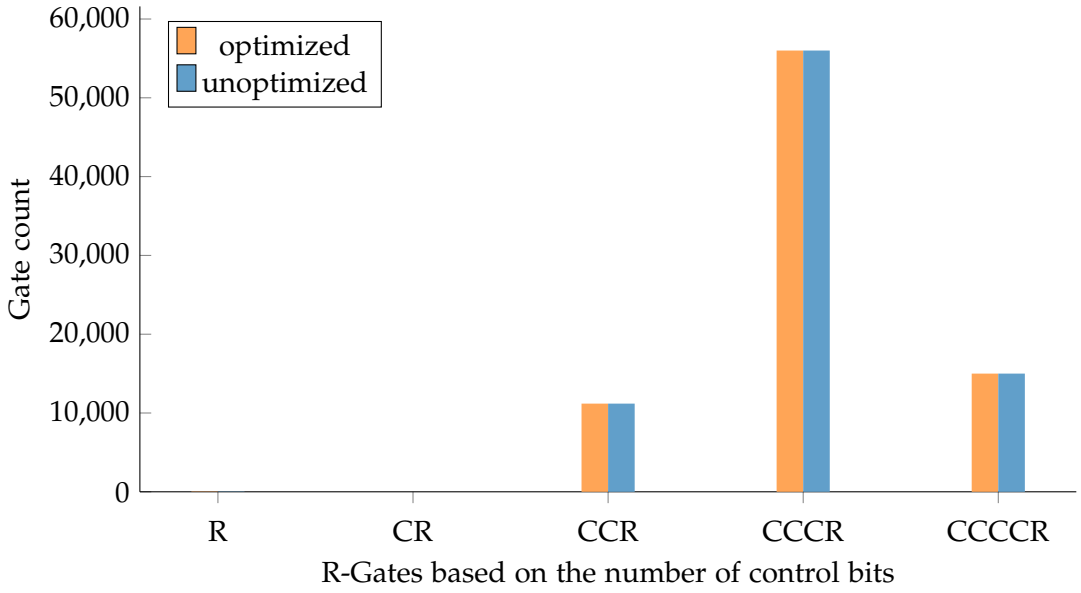

Figure 4.1.: R-Gate counts after executing Shor's algorithm with $N = 2^n - 1$ $n = 5$ and $a = 2$ for the optimized program and the unoptimized program

In figure 4.1 the number of R-Gates and the controlled versions of them are shown after executing Shor's algorithm with $N = 2^n - 1$, $n = 5$ as its factor, and $a = 2$ as its base. The x-axis labels indicate how many control bits are used to control the R-Gate where each C refers to one control bit. The graphic compares the results of the optimized program, highlighted in orange, and the unoptimized program highlighted in blue. The results reveal that the optimized program does not show any improvements compared to the base program without any optimizations. This can be explained by looking at the base program, which can be found in the Appendix of this thesis. After looking at the program and the possible optimizations, we can see that the source code does not offer any optimization possibilities for the implemented optimizations in this IR. The used optimization passes did not find any adjacent gates that could be eliminated or folded. This can be improved by adding more optimization passes that modify the IR. The resulting IR could potentially expose more optimization possibilities for the already implemented passes in addition to the newly introduced improvements. Especially an optimization pass regarding the controlled operations would be beneficial as controlled gates are the majority of the used gates. This would reduce the number of controlled operations and therefore reduce the number of controlled R-Gates that are used. Another issue lies in the decomposition of the controlled operations. We simplified the decomposition process of the controlled operations for this IR. Instead

of propagating the controlled operation to all operations inside the controlled circuit properly, a copy of the circuit was created and marked with an attribute to indicate that the circuit is controlled. This causes that some optimizations possibilities would not be exposed in the IR, resulting in the high number of controlled R-Gates where multiple control bits are needed.

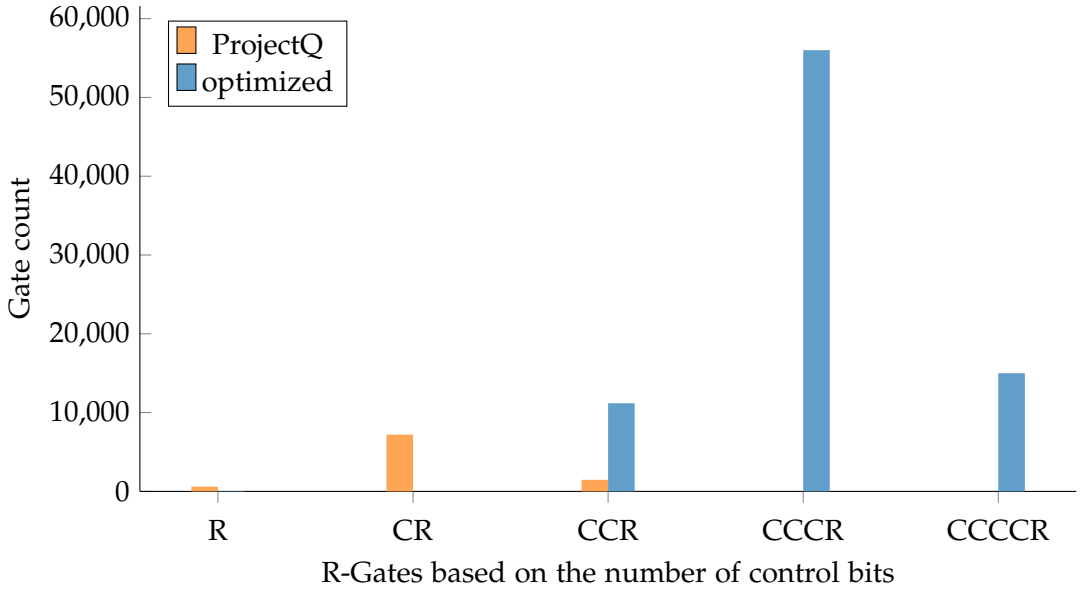

Figure 4.2.: R-Gate counts after executing Shor's algorithm with $N = 2^n - 1$, $n = 5$ and $a = 2$ for the optimized program and the ProjectQ implementation

The comparison between the optimized program and the ProjectQ implementation is seen in figure 4.2. We can see that the number of R-Gates and controlled R-Gates differ between the implementations. While the number of R-Gates and CR-Gates are lower in the MLIR implementation, the number of R-Gates with more than one control bit is significantly higher in the MLIR implementation and greatly overshadows the total number of R-Gates used in ProjectQ. Besides the higher total number of used gates, the ProjectQ implementation also used less controlled operations than our implementation. While our IR has R-Gates that are controlled by up to four control bits, the ProjectQ implementation uses at most two bits to control the R-Gates. As mentioned earlier, the main reason for that lies in the decomposition of the controlled operations and the missing optimizations for them.

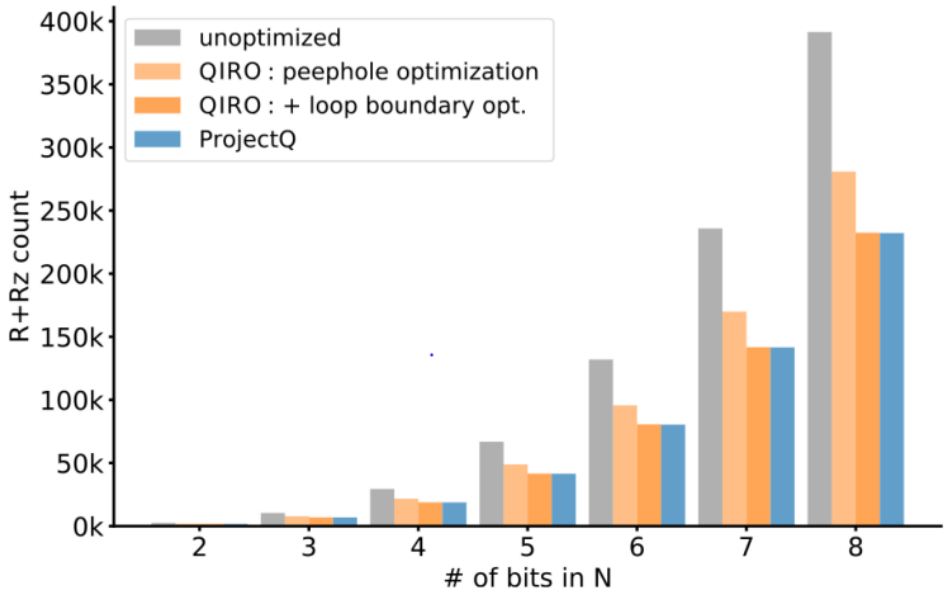

Figure 4.3.: Results obtained in QIRO compared to the results in ProjectQ for $N = 2^n - 1$, $n \in [2, 8]$; from Ittah et al. "QIRO: a static single assignment based quantum program representation for optimization" 2022

The importance of the controlled operations can also be seen by comparing the ProjectQ results to the results of QIRO. Their results can be seen in figure 4.3 (Ittah et al. 2022). They also compared the number of the R-Gates used during the execution of Shor's algorithm but they decomposed controlled operations differently and implemented the resource counter according to Meuli et al. [6]. While there exists a visible difference between their unoptimized program and ProjectQ, this gap is significantly smaller in comparison to our results. Their implementation of Shor's algorithm is nearly identical to the one we implemented for this thesis. The main difference is that their program is written in an input dialect where the operations are performed with side effects. This input is then translated into their optimization dialect where the operations work on value-semantics and are free from side effects. By applying the loop boundary optimizations in QIRO, the number of R-Gates used in QIRO are nearly equivalent to the numbers obtained in ProjectQ.

Nevertheless, a huge advantage compared to the implementation of ProjectQ is that the combined compile and execution time is a lot faster in the proposed IR. The optimizations in ProjectQ are performed at runtime and are dependent on the input size. This limits the size of the inputs that can be used in ProjectQ. For example, inputs of the size $N = 2^n - 1$, $n = 6$ can already take multiple minutes to compute depending on the hardware that is used. Since all optimizations in the proposed IR are performed statically during the compile time, the execution time is independent of the size of the input and stays below one second for inputs of the size of $N = 2^n - 1$, $n = 6$.

5. Conclusion

The goals and the results of the thesis are summarized, followed by a short discussion about the suitability of MLIR for optimizations. Lastly, an outlook for improvements in further iterations of this thesis is given.

5.1. Summary

In this thesis, we proposed and implemented an IR optimized for quantum computing with the MLIR framework from LLVM. We implemented a custom dialect that features types and operations that are needed for quantum computing. By utilizing this framework we introduce several passes to read, transform, and optimize quantum programs written in MLIR. Furthermore, quantum programs written in the implemented IR can be executed in order to get the resource count of the used gates. The tests have shown that the optimization passes do not reduce the count of R-Gates that are used during the execution of Shor's algorithm. One reason is that the implementation of Shor's algorithm that we used does not expose any possible optimization possibilities for the currently implemented optimization passes. Also, the decomposition of controlled operations was simplified so that no further optimization possibilities could happen. Nevertheless, the compilation of quantum programs in the proposed IR with the optimization passes can be applied in a manageable amount of time as compared to other frameworks that apply optimizations at run time and are dependent on the input size.

5.2. Suitability of MLIR for Optimizations

MLIR allows the user to create custom dialects for their IR that are specialized for their specific needs. Thanks to the many built-in tools such as the pattern rewriter and TableGen, different optimization and transformation passes can be implemented without a lot of effort. The results achieved with the MLIR framework are also remarkable as shown by Ittah et al. whose implementation in MLIR showed similar results as the ProjectQ implementation. In addition to that, most optimization passes are applied during the compilation phase which results in a significantly faster execution time. The main caveat of the framework is that the entry level is quite steep. While the official website offers a few tutorials, these tutorials only cover a fraction of the whole framework. Furthermore, the documentation of many functions is lacking in some aspects and does not offer enough information for the user. An example of that is the

description of the print operation in the *vector* dialect [1]. Although the documentation says that the user has to link another library to lower and execute the print operation, it does not state which library needs to be linked. Nevertheless, after working with the framework for a while and getting familiar with it, we can say that the framework itself is not hard to use and provides many helpful tools to optimize programs. Therefore, we would say that the framework is suitable for optimizations even though it is difficult to understand in the beginning.

5.3. Outlook

In future works of this quantum IR, a more detailed algorithm for the decomposition of controlled operations should be implemented to simulate a more realistic estimation of resources. Also, this gives more opportunities for optimization as the decomposition could potentially expose more adjacent gates that can be cancelled or folded. Furthermore, further optimizations such as loop-boundary optimization or adjoint circuit cancellation as shown by Ittah et al. [2] can be implemented. In addition, optimizations regarding quantum circuits, such as re-synthesizing them into smaller sub-circuits which are utilized in the ScaffCC framework [3], may be considered. In order to ease the use of the IR, a mapping from a higher-level quantum language, such as Q# or Qiskit, can be introduced, as the IR is not suitable for direct programming.

[1] Print operation in the vector dialect `https://mlir.llvm.org/docs/Dialects/Vector/#vectorprint-mlirvectorprintop`

List of Figures

List of Tables

Bibliography

[1] V. Hassija, V. Chamola, A. Goyal, S. S. Kanhere, and N. Guizani. "Forthcoming applications of quantum computing: Peeking into the future." In: *IET Quantum Communication* 1.2 (2020), pp. 35–41.

[2] D. Ittah, T. Häner, V. Kliuchnikov, and T. Hoefler. "QIRO: a static single assignment-based quantum program representation for optimization." In: *ACM Transactions on Quantum Computing* 3.3 (2022), pp. 1–32.

[3] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi. "ScaffCC: A framework for compilation and analysis of quantum computing programs." In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. 2014, pp. 1–10.

[4] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation." In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[5] A. McCaskey and T. Nguyen. "A MLIR dialect for quantum assembly languages." In: *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE. 2021, pp. 255–264.

[6] G. Meuli, M. Soeken, M. Roetteler, and T. Häner. "Enabling accuracy-aware quantum compilers using symbolic resource estimation." In: *arXiv preprint arXiv:2003.08408* (2020).

[7] A. Peduri, S. Bhat, and T. Grosser. "QSSA: an SSA-based IR for Quantum computing." In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 2022, pp. 2–14.

[8] J. Preskill. "Quantum computing in the NISQ era and beyond." In: *Quantum* 2 (2018), p. 79.

[9] M. Reiher, N. Wiebe, K. M. Svore, D. Wecker, and M. Troyer. "Elucidating reaction mechanisms on quantum computers." In: *Proceedings of the national academy of sciences* 114.29 (2017), pp. 7555–7560.

[10] W. K. Wootters and W. H. Zurek. "The no-cloning theorem." In: *Physics Today* 62.2 (2009), pp. 76–77.

A. Appendix

Shor's algorithm implementation in MLIR

Listing A.1: Shor's algorithm in MLIR

```
module {
    func.func @mod ( %a : i64 , %N : i64 ) -> i64 {
        %0 = arith.divui %a , %N : i64
        %1 = arith.muli %N , %0 : i64
        %2 = arith.subi %a , %1 : i64
        return %2 : i64
    }

    func.func @mod_exp ( %b : i64 , %e : i64 , %N : i64 ) -> i64 {
        %c0 = arith.constant 0 : i64
        %c1 = arith.constant 1 : i64
        %c2 = arith.constant 2 : i64
        %cond = arith.cmpi "eq" , %N , %c1 : i64
        cf.cond_br %cond , ^ret ( %c0 : i64 ) , ^reduce

        ^reduce:
            %res = arith.constant 1 : i64
            %base = func.call @mod ( %b , %N ) : ( i64 , i64 ) -> i64
            %cond2 = arith.cmpi "ugt" , %e , %c0 : i64
            cf.cond_br %cond2 , ^while ( %base , %e , %res : i64 , i64 , i64 ) , ^ret (
    %res : i64 )

        ^while ( %base_0 : i64 , %exp_0 : i64 , %res_0 : i64 ):
            %0 = func.call @mod ( %exp_0 , %c2 ) : ( i64 , i64 ) -> i64
            %cond3 = arith.cmpi "eq" , %0 , %c1 : i64
            %res_1 = scf.if %cond3 -> i64 {
                %1 = arith.muli %res_0 , %base_0 : i64
                %2 = func.call @mod ( %1 , %N ) : ( i64 , i64 ) -> i64
                scf.yield %2 : i64
            } else {
                scf.yield %res_0 : i64
            }

            %exp_1 = arith.shrui %exp_0 , %c1 : i64
            %3 = arith.muli %base_0 , %base_0 : i64
            %base_1 = func.call @mod ( %3 , %N ) : ( i64 , i64 ) -> i64
            %cond4 = arith.cmpi "ugt" , %exp_1 , %c0 : i64
            cf.cond_br %cond4 , ^while ( %base_1 , %exp_1 , %res_1 : i64 , i64 , i64 ) ,
     ^ret ( %res_1 : i64 )
```

```
38
39          ^ret ( %r : i64 ):
40              return %r : i64
41      }
42
43      func.func @mod_inv ( %C : i64 , %N : i64 ) -> i64 {
44          %c0 = arith.constant 0 : i64
45          %c1 = arith.constant 1 : i64
46          cf.br ^while ( %N , %C , %c0 , %c1 : i64 , i64 , i64 , i64 )
47
48          ^while ( %r_0 : i64 , %old_r : i64 , %s_0 : i64 , %old_s : i64 ) :
49              %q = arith.divui %old_r , %r_0 : i64
50              %qr = arith.muli %q , %r_0 : i64
51              %r_1 = arith.subi %old_r , %qr : i64
52              %qs = arith.muli %q , %s_0 : i64
53              %s_1 = arith.subi %old_s , %qs : i64
54
55              %cond = arith.cmpi "ne" , %r_1 , %c0 : i64
56              cf.cond_br %cond , ^while ( %r_1 , %r_0 , %s_1 , %s_0 : i64 , i64 , i64 ,
    i64 ) , ^ret ( %s_0 : i64 )
57
58          ^ret ( %s : i64 ):
59              %0 = arith.addi %s , %N : i64
60              %1 = func.call @mod ( %0 , %N ) : ( i64 , i64 ) -> i64
61          return %1 : i64
62      }
63
64      func.func @calc_qft_angle ( %j : index ) -> f64 {
65          %pi = arith.constant 3.141592653589793238 : f64
66          %c1 = arith.constant 1 : index
67          %0 = arith.addi %c1 ,%j : index
68          %1 = index.shl %c1 ,%0
69          %2 = index.casts %1 : index to i64
70          %3 = arith.uitofp %2 : i64 to f64
71          %4 = arith.divf %pi , %3 : f64
72          return %4 : f64
73      }
74
75      func.func @calc_add_angle ( %i : index , %j : index ) -> f64 {
76          %pi = arith.constant 3.141592653589793238 : f64
77          %c1 = arith.constant 1 : index
78          %0 = arith.subi %i , %j : index
79          %1 = index.shl %c1 , %0
80          %2 = index.casts %1 : index to i64
81          %3 = arith.uitofp %2 : i64 to f64
82          %4 = arith.divf %pi , %3 : f64
83          return %4 : f64
84      }
85
86      func.func @calc_cur_a ( %N : i64 , %n : index , %a : i64 , %i : index ) -> i64 {
87          %c1 = arith.constant 1 : i64
```

```
88          %c2 = arith.constant 2 : i64
89          %k = index.casts %i : index to i64
90          %nbits = index.casts %n : index to i64
91          %0 = arith.muli %nbits , %c2 : i64
92          %1 = arith.subi %0 , %c1 : i64
93          %2 = arith.subi %1 , %k : i64
94          %3 = arith.shli %c1 , %2 : i64
95          %4 = func.call @mod_exp ( %a , %3 , %N ) : ( i64 , i64 , i64 ) -> i64
96          return %4 : i64
97      }
98
99      func.func @calc_shor_angle ( %i : index , %j : index ) -> f64 {
100         %mpi = arith.constant -3.141592653589793238 : f64
101         %c1 = arith.constant 1 : index
102         %0 = arith.subi %i , %j : index
103         %1 = index.shl %c1 , %0
104         %2 = index.casts %1 : index to i64
105         %3 = arith.uitofp %2 : i64 to f64
106         %4 = arith.divf %mpi , %3 : f64
107         return %4 : f64
108         }
109
110     quantum.circ @QFT ( %r : !quantum.qureg<>, %n : index ) -> !quantum.qureg<> {
111         %c0 = arith.constant 0 : index
112         %c1 = arith.constant 1 : index
113         %c2 = arith.constant 2 : index
114         %r1 = scf.for %i = %c0 to %n step %c1 iter_args(%qr = %r) -> !quantum.qureg<> {
115             %0 = arith.addi %i , %c1 : index
116             %k = arith.subi %n , %0 : index
117             %r_h = quantum.H %qr [%k] : !quantum.qureg<> -> !quantum.qb
118             %r2 = scf.for %j = %c0 to %k step %c1 iter_args (%qrr = %qr) -> !quantum.
    qureg<> {
119                 %phi = func.call @calc_qft_angle ( %j ) : ( index ) -> f64
120                 %rg = quantum.R( %phi : f64 ) -> !quantum.u1
121                 %1 = arith.addi %j , %c1 : index
122                 %h = arith.subi %k , %1 : index
123                 %qb2 = quantum.ctrl %rg , %qrr[%h], %qrr[%k] : !quantum.qureg<> , !
    quantum.qureg<>, !quantum.u1 -> !quantum.qb
124                 scf.yield %qrr : !quantum.qureg<>
125                 }
126             scf.yield %r2 : !quantum.qureg<>
127         }
128
129         %nd2 = index.divu %n , %c2
130         %r3 = scf.for %i = %c0 to %nd2 step %c1 iter_args(%qr= %r1) -> !quantum.qureg<>
    {
131             %0 = arith.addi %i , %c1 : index
132             %j = arith.subi %n , %0 : index
133             %qb1, %qb2 = quantum.SWAP %qr[%i], %qr[%j] : !quantum.qureg<>, !quantum.
    qureg<> -> !quantum.qb , !quantum.qb
134             scf.yield %qr : !quantum.qureg<>
```

```
135         }
136         quantum.return %r3 : !quantum.qureg<>
137     }
138
139     quantum.circ @addConstant ( %C : i64 , %r : !quantum.qureg<>, %n : index ) -> !
        quantum.qureg<> {
140         %c0 = arith.constant 0 : index
141         %s1 = arith.constant 1 : index
142         %c1 = arith.constant 1 : i64
143         %rr = quantum.call @QFT (%r , %n )  : (!quantum.qureg<>, index) -> !quantum.
        qureg<>
144         %r2 = scf.for %i = %c0 to %n step %s1 iter_args(%qr = %rr) -> !quantum.qureg<> {
145             %ip1 = arith.addi %i , %s1 : index
146             %qr0 = scf.for %j = %c0 to %ip1 step %s1 iter_args(%qrr = %qr) -> !quantum.
        qureg<>{
147                 %k = arith.subi %i , %j : index
148                 %0 = index.casts %k : index to i64
149                 %1 = arith.shrsi %C , %0 : i64
150                 %2 = arith.andi %1 , %c1 : i64
151                 %cond = arith.cmpi "eq" , %2 , %c1 : i64
152                 scf.if %cond  {
153                     %phi = func.call @calc_add_angle ( %i , %k ) : ( index , index ) ->
        f64
154                     %qb = quantum.R( %phi : f64 ) %qrr[%i] : !quantum.qureg<> -> !
        quantum.qb
155           }
156                 scf.yield %qrr : !quantum.qureg<>
157           }
158             scf.yield %qr0 : !quantum.qureg<>
159         }
160
161         %qft = quantum.getval @QFT -> !quantum.circ
162         %qft_inv = quantum.adj %qft : !quantum.circ -> !quantum.circ
163         %qar = quantum.apply %qft_inv(%r2,%n ) : !quantum.circ (!quantum.qureg<>, index
        ) -> !quantum.qureg<>
164         quantum.return %qar : !quantum.qureg<>
165     }
166
167     quantum.circ @subConstant ( %C : i64 , %r : !quantum.qureg<>, %n : index ) -> !
        quantum.qureg<>{
168         %cm1 = arith.constant -1 : i64
169         %mC = arith.muli %C , %cm1 : i64
170         %qt = quantum.call @addConstant ( %mC , %r , %n ) : (i64 , !quantum.qureg<>,
        index) -> !quantum.qureg<>
171         quantum.return %qt : !quantum.qureg<>
172     }
173
174     quantum.circ @addCmodN ( %C : i64 , %N : i64 , %r : !quantum.qureg<>, %n : index )
        -> !quantum.qureg<> {
175         %c1 = arith.constant 1 : index
176         %nm1 = arith.subi %n , %c1 : index
```

```
177         %r1 =quantum.call @addConstant ( %C , %r , %n ) : (i64 , !quantum.qureg<>, index
    ) -> !quantum.qureg<>
178         %r2 = quantum.call @subConstant ( %N , %r1 , %n) : (i64 , !quantum.qureg<>,
    index) -> !quantum.qureg<>
179         %anc = quantum.alloc -> !quantum.qb
180         %qb0, %qb01 = quantum.CX %r2 [ %nm1 ], %anc : !quantum.qureg<>, !quantum.qb -> !
    quantum.qb, !quantum.qb
181         %addOp = quantum.getval @addConstant -> !quantum.circ
182         %ctrlAdd = quantum.ctrl %addOp , %anc : !quantum.qb , !quantum.circ -> !quantum.
    cop<1, !quantum.circ>
183         %r3 = quantum.apply %ctrlAdd (%N , %r2 , %n ) : !quantum.cop<1 ,!quantum.circ >(
     i64 , !quantum.qureg<>, index ) -> !quantum.qureg<>
184         %r4 = quantum.call @subConstant ( %C , %r3 , %n ) : (i64 , !quantum.qureg<>,
    index) -> !quantum.qureg<>
185
186         %qb1 = quantum.X %r4 [ %nm1 ]  : !quantum.qureg<> -> !quantum.qb
187         %qb2, %qt1 = quantum.CX %qb1, %anc : !quantum.qb, !quantum.qb -> !quantum.qb, !
    quantum.qb
188
189         %qb3 = quantum.X %qb2 : !quantum.qb -> !quantum.qb
190         quantum.free %anc : !quantum.qb
191       %r5 = quantum.call @addConstant ( %C , %r4 , %n ) : (i64 , !quantum.qureg<>, index
    ) -> !quantum.qureg<>
192       quantum.return %r5 : !quantum.qureg<>
193     }
194
195     quantum.circ @subCmodN ( %C : i64 , %N : i64 , %r : !quantum.qureg<>, %n : index )
    -> !quantum.qureg<>{
196       %NmC = arith.subi %N , %C : i64
197       %r1 = quantum.call @addCmodN ( %NmC , %N , %r , %n ) : (i64 , i64 , !quantum.qureg
    <>, index) -> !quantum.qureg<>
198       quantum.return %r1 : !quantum.qureg<>
199     }
200
201     quantum.circ @mulCmodN ( %C : i64 , %N : i64 , %r : !quantum.qureg<>, %n : index )
    -> !quantum.qureg<>{
202       %c0 = arith.constant 0 : index
203       %c1 = arith.constant 1 : index
204       %np1 = arith.addi %n , %c1 : index
205       %anc = quantum.allocreg (%np1) -> !quantum.qureg<>
206       %Cinv = func.call @mod_inv ( %C , %N ) : ( i64 , i64 ) -> i64
207       %anc1, %r1 = scf.for %i = %c0 to %n step %c1 iter_args (%qanc=%anc , %qr = %r) ->
    (!quantum.qureg<>, !quantum.qureg<>) {
208         %addOp = quantum.getval @addCmodN -> !quantum.circ
209         %ctrlAdd = quantum.ctrl %addOp , %qr [ %i ] : !quantum.qureg<>, !quantum.circ ->
     !quantum.cop<1,!quantum.circ>
210         %0 = index.casts %i : index to i64
211         %1 = arith.shli %C , %0 : i64
212         %2 = func.call @mod ( %1 , %N ) : ( i64 , i64 ) -> i64
213         %qcr = quantum.apply %ctrlAdd (%2 , %N , %qanc , %np1 ) : !quantum.cop<1,!
    quantum.circ >( i64 , i64 , !quantum.qureg<>, index ) -> !quantum.qureg<>
```

```
214         scf.yield %qanc, %qcr : !quantum.qureg<>, !quantum.qureg<>
215       }
216       %anc2, %r2= scf.for %i = %c0 to %n step %c1 iter_args  (%qanc = %anc1, %qr=%r1)->
    (!quantum.qureg<>, !quantum.qureg<>) {
217         %qb1, %qb2 = quantum.SWAP %qanc [ %i ], %qr [ %i ] : !quantum.qureg<> , !quantum.
    qureg<> -> !quantum.qb , !quantum.qb
218         scf.yield %qanc, %qr : !quantum.qureg<>, !quantum.qureg<>
219       }
220       %anc3, %r3= scf.for %i = %c0 to %n step %c1 iter_args(%qanc = %anc2, %qr=%r2) ->
    (!quantum.qureg<>, !quantum.qureg<>) {
221         %subOp = quantum.getval @subCmodN -> !quantum.circ
222         %ctrlSub = quantum.ctrl %subOp , %qr[%i] : !quantum.qureg<> , !quantum.circ -> !
    quantum.cop<1 , !quantum.circ >
223         %3 = index.casts %i : index to i64
224         %4 = arith.shli %Cinv , %3 : i64
225         %5 = func.call @mod ( %4 , %N ) : ( i64 , i64 ) -> i64
226         %qcr = quantum.apply %ctrlSub (%5 , %N , %qanc , %np1 ) : !quantum.cop<1,!
    quantum.circ >( i64 , i64 , !quantum.qureg<>, index) -> !quantum.qureg<>
227         scf.yield %qanc, %qr : !quantum.qureg<>, !quantum.qureg<>
228       }
229       quantum.freereg %anc : !quantum.qureg<>
230       quantum.return %r3 : !quantum.qureg<>
231     }
232
233     quantum.circ @shor ( %N : i64 , %a : i64 )->() {
234       %c0 = arith.constant 0 : index
235       %c1 = arith.constant 1 : index
236       %c2 = arith.constant 2 : index
237       %0 = arith.uitofp %N : i64 to f64
238       %1 = math.log2 %0 : f64
239       %2 = math.ceil %1 : f64
240       %3 = arith.fptoui %2 : f64 to i64
241       %n = index.casts %3 : i64 to index
242       %n2 = arith.muli %n , %c2 : index
243       %m0 = arith.constant 0 : i1
244       %meas = memref.alloc ( %n2 ) : memref<?xi1>
245       scf.for %i = %c0 to %n2 step %c1 {
246         memref.store %m0 , %meas [ %i ] : memref<? xi1 >
247       }
248       %r = quantum.allocreg( %n ) -> !quantum.qureg<>
249       %cqb = quantum.alloc -> !quantum.qb
250       %c3 = arith.constant 0 : index
251       %qb = quantum.X %r[%c3] : !quantum.qureg<> -> !quantum.qb
252       %r1, %cqb1 =scf.for %i = %c0 to %n2 step %c1 iter_args (%qr =%r, %qcqb=%cqb) -> (!
    quantum.qureg<>, !quantum.qb) {
253         %cur_a = func.call @calc_cur_a ( %N , %n , %a , %i ) : ( i64 , index , i64 ,
    index ) -> i64
254         %qcqb1 = quantum.H %qcqb : !quantum.qb -> !quantum.qb
255         %mulOp = quantum.getval @mulCmodN -> !quantum.circ
256         %ctrlMul = quantum.ctrl %mulOp , %qcqb1 : !quantum.qb, !quantum.circ -> !quantum.
    cop<1 , !quantum.circ>
```

```
257         %qrr = quantum.apply %ctrlMul ( %cur_a , %N , %qr , %n ) : !quantum.cop<1 , !
        quantum.circ>( i64 , i64 , !quantum.qureg<>, index ) -> !quantum.qureg<>
258         %qcqb2 = scf.for %t = %c0 to %i step %c1 iter_args(%qqcqb=%qcqb1)-> !quantum.qb{
259                 %cond = memref.load %meas [ %t ] : memref<?xi1>
260           scf.if %cond {
261             %phi = func.call @calc_shor_angle ( %i , %t ) : ( index , index ) -> f64
262             %qqcqb1 = quantum.R ( %phi : f64 ) %qqcqb : !quantum.qb -> !quantum.qb
263
264           }
265           scf.yield %qqcqb : !quantum.qb
266         }
267         %qcqb3 = quantum.H %qcqb2 : !quantum.qb -> !quantum.qb
268         %m = quantum.meas %qcqb3 : !quantum.qb -> i1
269         memref.store %m , %meas [ %i ] : memref <?xi1 >
270         scf.if %m {
271           %qcqb4 = quantum.X %qcqb3 : !quantum.qb -> !quantum.qb
272         }
273         scf.yield %qrr, %qcqb3 : !quantum.qureg<>, !quantum.qb
274       }
275       %mres = quantum.meas %r1 : !quantum.qureg<> -> memref<?xi1 >
276       quantum.free %cqb1 : !quantum.qb
277       quantum.freereg %r1 : !quantum.qureg<>
278       quantum.return
279     }
280
281     quantum.circ @mlir_main ( %N : i64 , %a : i64 ) {
282       quantum.call @shor ( %N , %a ) : (i64 , i64) -> ()
283       quantum.return
284     }
285
286     func.func @main () {
287       %N = arith.constant 5: i64
288       quantum.call @mlir_main(%N,%N) : (i64, i64) -> ()
289         return
290     }
291 }
```