The help document of quantum compiler

Qsoftware Team in ISCAS

1 Introduction

This is a brief introduction to our quantum compiler, a part of quantum software toolkit of Institute of Software, Chinese Academy of Science. The main function of this tool is compiling the high-level source language in to a low-level intermediate representation (IR). The high-level language, named *isQ*, is based on [1] and similar to qWhile [2], besides we add some new features such as recursion and local variables. The IR is similar to Rigetti's Quil [3], but we make some moderate changes to enable recursion stacks for recursion and local variables. As we have not connected it to a real quantum hardware device, we write a simulator to execute the current IR instructions. Further applying the IR (and modifying it if necessary) to hardware like superconducting or ion-trap will be a following step.

As the compiler in a rather raw version, if you find any bugs, please feel free to inform us and we will be very grateful. We also appreciate any kind suggestions. :)

2 Grammar

In [1], the high-level language supports several quantum instructions: initialization, unitary operation, measurement. Classical control like if-statement and loop-statement is also involved. To make the language more portable to users and more feasible to existing quantum architectures, we permit explicitly declaration of both classical and quantum variables in the source language. This indicates that in the IR both quantum registers and classical registers are needed, as Quil does.

2.1 Structure

A program written in our source language usually composes of three parts: gate definition, global variable definition and procedure definition. For readability and convenience, we require these three parts in strict order for the moment,

that is:

$$Gate\ Definition$$
$$...$$
$$Global\ Variable\ Definition$$
$$...$$
$$Procedure\ Definition$$
$$...$$

And they can not overlap with each other. Only when previous part is finished could the users write the next part.

2.2 Gate Definition

We permit the users to define their own desired unitary operators at the very beginning of programs. Note that some basic gates are automatically defined and need no definition by users: H,T,X,Y,Z,CX(CNOT),CZ. They are reserved words and cannot be used as user-defined gate names, variable names or procedure names. Another reserve word is M, which acts as the projection measurement operator.

To define a new gate the user should input a complex matrix and name the gate like below:

$$Defgate\ U1 = [c_{1,1}, c_{1,2}, \cdots, c_{1,n};$$
$$c_{2,1}, c_{2,2}, \cdots, c_{2,n};$$
$$\cdots$$
$$c_{n,1}, c_{n,2}, \cdots, c_{n,n}];$$

where $U1$ is the gate name and n must be a power of 2. $c_{i,j}$ is a complex number, such as 0 or $-0.5 + 0.5j$.

2.3 Global Variable Definition

There are two types of variables are supported currently: *int* and *qbit*. Define a new *int* or *qbit* is like:

$$int\ a;$$
$$qbit\ b;$$

By the way, the user can define multiple variable of same type in one clause:

$$qbit\ q0, q1, q2, q3;$$

We also allow definition of variable arrays now, like:

$$qbit\ q[3];$$

2.4 Procedure Definition

For flexibility, we permit the user to define custom procedures which can recursively call each other. But a main procedure must exist and can not be called by other procedures:

```
procedure A() {
     PROCEDURE_BODY_A
}

procedure main() {
     PROCEDURE_BODY_MAIN
}
```

A is a proc name. PROCEDURE_BODY_A and PROCEDURE_BODY_MAIN can contain five kinds of statements, which will be introduced in the following.

2.5 Statement

the permitted statements include: while statement, if statement, initialization, assignment statement, procedure call statement. For classical control, the guard in while and if statement can only contains classical variables. Initialization is used to reset a quantum variable. Assignment statements include assignment of arithmetic value, unitary operation and measurement operation.

2.5.1 while statement

while statement provides the loop structure, it has the form:

```
while (GUARD) do
     LOOPBODY
od
```

Currently the GUARD only permits the comparison of two expressions composed of integers and int variables. LOOPBODY has no restrictions, with all five statements permitted. Note that brackets are necessary for the GUARD. See an example below:

```
while (x<2) do
          x = x + 1;
     od
```

2.5.2 if statement

Similar to while statement, the structure is below:

```
if (GUARD) then
     IFBODY
fi
```

or

```
if (GUARD) then
    IFBODY_1
else
    IFBODY_2
fi
```

in while and if statements, 'do','od','if','fi' are reserved words to indicate the scope of such statements.

2.5.3 initialization

Initialization statement are used to reset a qubit to state $|0\rangle$. See an example below:

```
q0 = |0>;
q1 = |0>;
```

where $q0, q1$ are *qbit* variables. The structure for this statement is fixed, and reset a qubit to state $|1\rangle$ is not permitted.

2.5.4 assignment statement

There are three types of assignment. The first type is assigning expression value to a int variable. For example:

```
a = a + 2;
```

The second type is assigning measurement outcome of a qubit to a int variable:

```
a = M [q];
```

where q must be a *qbit* variable and a must be a *int* variable. The third type is unitary operation:

```
H <q>;
CX <q0,q1>;
uGate <q0,q1,q2>;
```

H is Harmard gate and CX is controll-NOT gate. The common basic gates include $H, X, Y, Z, T, S, \mathrm{CX(CNOT)}, \mathrm{CZ}$. Here *uGate* is a user-defined gate, it needs to be defined in the gate definition stage. Note that *uGate* must be 8×8-dimension unitary or the compiler will report error message.

2.5.5 procedure call statement

This statement has one line:

```
A();
```

where A is name of a procedure. The parameters are permitted.

```
A( para1 , para2 , . . . , paraM );
```

For example:

```
procedure A( int m, int b [ ] , qbit q)
```

3   An Quil-like IR

For demonstration, we currently adopt an IR similar to Rigetti's Quil. The major difference lays the type of classical registers. Our IR permits an stack besides with static memory. We use registers $C[0], C[1], ...$ to denote static memory, and use registers $s[0], s[1], ...$ each of these registers could store a 32-bit signed integer. Such approach make it convenient to separate the dynamic and static memory consumption and relatively more readable to programmers. Though whether Quil or our IR, it cannot be directly applied to a real hardware device[1].

Example 3.1 (quantum-while) *See below a program with a while loop:*

```
qbit  q , p ;
int  x , y ;

procedure  main ( )  {
        q  =  | 0 >;
        p  =  | 0 >;
        x  =  0 ;
        while  ( x  ==  0 )  do
                H <p>;
                CNOT <p , q >;
                x  =  M[ p ] ;
        od
        y  =  M[ p ] ;
        print  x ;
        print  y ;
}
```

The compiling results:

```
__main :
        MOV  REGS  0
        MOV  SPEC1  2
        MOV  SPEC2  2
        Measure  q [ 0 ]  c [ SPEC1 ]
        JUMP–IF  @IF0  c [ SPEC1 ]==0
```

---

[1]In fact, there is an instruction set called QPU-executable Quil for Rigetti's current quantum devices, but in this doc we discuss the original Quil

```
X  q [ 0 ]
LABEL  @IF0
 Measure  q [ 1 ]  c [ SPEC1 ]
JUMP–IF  @IF1  c [ SPEC1 ]==0
X  q [ 1 ]
LABEL  @IF1
MOV  c [ 0 ]  0
LABEL  @LOOPGUARD0
MOV  SPEC3  c [ 0 ]
MOV  SPEC4  0
JUMP–UNLESS  @LOOPEND0  SPEC3==SPEC4
H  q [ 1 ]
CNOT  q [ 1 ]  q [ 0 ]
 Measure  q [ 1 ]  c [ 0 ]
JUMP  @LOOPGUARD0
LABEL  @LOOPEND0
 Measure  q [ 1 ]  c [ 1 ]
PRINT  c [ 0 ]
PRINT  c [ 1 ]
```

Example 3.2 (Grover's fixed-point quantum search[4]) *initial state* $|s\rangle$, *target state* $|t\rangle$, *where* $\langle s|t\rangle = 0$. *To search* $|t\rangle$, *we use following three gates* U, R_s, R_t, *which satisfies:*

$$||\langle t|U|s\rangle||^2 = (1 - \epsilon) < 1$$

$$R_s = I - [1 - exp(i\frac{\pi}{3})]|s\rangle\langle s|$$

$$R_t = I - [1 - exp(i\frac{\pi}{3})]|t\rangle\langle t|$$

define U_m *as following:*

- *when* $m = 0$, $U_m = U$;

- $\forall m \geq 0, U_{m+1} = U_m R_s U_m^\dagger R_t U_m.$

it can be proved by induction:

$$\forall m > 0, \ ||\langle t|U_m|s\rangle||^2 = 1 - \epsilon^{3^m}$$

program code:

```
Defgate  Rs  =[0.5+0.8660254 j ,0 ,0 ,0;
            0 ,1 ,0 ,0;
            0 ,0 ,1 ,0;
          0 ,0 ,0 ,1];
Defgate  Rs2  =[0.5−0.8660254 j ,0 ,0 ,0;
            0 ,1 ,0 ,0;
```

6

```
                  0 , 0 , 1 , 0;
             0 , 0 , 0 , 1];


Defgate  Rt  =  [1 , 0 , 0 , 0;
                  0 , 1 , 0 , 0;
                  0 , 0 , 1 , 0;
             0 , 0 , 0 , 0.5 + 0.8660254 j ];
Defgate  Rt2  =  [1 , 0 , 0 , 0;
                  0 , 1 , 0 , 0;
                  0 , 0 , 1 , 0;
             0 , 0 , 0 , 0.5 − 0.8660254 j ];



qbit  p , q , r;
qbit  t [5];
int  a , x , y;

procedure  A1( int  a )  {
          if  (a==0)  then
                    H<p>;
                    H<q>;
          fi
          if  (a>0)  then
                    a  =  a  −  1;
                    A1(a );
                    Rt<p , q>;
                    B1(a );
                    Rs  <p , q>;
                    A1(a );
                    a  =  a+1;
          fi
}

procedure  B1( int  a )  {
          if  (a==0)  then
                    H<p>;
                    H<q>;
          fi
          if  (a>0)  then
                    a  =  a  −  1;
                    B1(a );
                    Rs2<p , q>;
                    A1(a );
                    Rt2<p , q>;
                    B1(a );
                    a  =  a  +  1;
```

```
          fi
}

procedure   main ( ) {
          a  =  2;
          A1 ( a );

          x  =  M [ p ];
          y  =  M [ q ];
          print  x;
          print  y;
}
```

the compiling result (omitted lines represented by "..."):

```
LABEL  @PROC_A1
          MOV  SPEC3  REGS
          SUB  SPEC3  2
          MOV  SPEC3  s [SPEC3]
          MOV  SPEC3  c [SPEC3]
          MOV  SPEC4  0
          JUMP–UNLESS  @IF0  SPEC3==SPEC4
          H  q [0]
          H  q [1]
          . . .
          MOV  c [SPEC4]  c [SPEC3]
          ADD  c [SPEC4]  1
          LABEL  @IF1
          SUB  REGS  1
          JUMP–IF  @CALL_END_A10  s [REGS]==0
          JUMP–IF  @CALL_END_A11  s [REGS]==1
          JUMP–IF  @CALL_END_A12  s [REGS]==2
          JUMP–IF  @CALL_END_A13  s [REGS]==3
LABEL  @PROC_B1
          MOV  SPEC3  REGS
          SUB  SPEC3  2
          MOV  SPEC3  s [SPEC3]
          MOV  SPEC3  c [SPEC3]
          MOV  SPEC4  0
          JUMP–UNLESS  @IF2  SPEC3==SPEC4
          H  q [0]
          H  q [1]
          LABEL  @IF2
          . . .
          LABEL  @IF3
          SUB  REGS  1
          JUMP–IF  @CALL_END_B10  s [REGS]==0
```

```
        JUMP-IF  @CALL_END_B11  s[REGS]==1
        JUMP-IF  @CALL_END_B12  s[REGS]==2
__main :
        MOV  REGS  0
        MOV  SPEC1  3
        MOV  SPEC2  8
        MOV  c[0]  2
        MOV  SPEC3  REGS
        MOV  s[SPEC3]  0
        ADD  REGS  1
        MOV  s[REGS]  3
        ADD  REGS  1
        JUMP  @PROC_A1
        LABEL  @CALL_END_A13
        SUB  REGS  1
        Measure  q[0]  c[1]
        Measure  q[1]  c[2]
        PRINT  c[1]
        PRINT  c[2]
```

4 Discussion

As our compiler might adjust the target IR, we expect to apply our compiler on real classic-quantum hybrid systems in the future. And the optimization of compilers, which plays a pivotal role in classical computer science, has tremendous potential in quantum computing. We will keep on updating the compiler.

References

[1] M. S. Ying, *Foundations of Quantum Programming*, Morgan-Kaufmann, 2016.

[2] Liu, Shusen, Wang, Xin, Zhou, Li, et al. $Q|SI\rangle$: A Quantum Programming Environment[J]. Scientia Sinica, 2017.

[3] Smith R S, Curtis M J, Zeng W, et al. A Practical Quantum Instruction Set Architecture.[J]. arXiv: Quantum Physics, 2016.

[4] Grover L K. Fixed-point quantum search.[J]. Physical Review Letters, 2005, 95(15): 150501.Grover L K. Fixed-point quantum search.[J]. Physical Review Letters, 2005, 95(15): 150501.