CS 516: Programming Languages and Compilers II
Project 1: Qubit Mapping for Quantum Circuits

THIS IS NOT A GROUP PROJECT! You may talk about the project in general terms, but must not share your code. In this project, you will be asked to implement compiler transformations for quantum circuits. The program takes a quantum circuit (in QASM format) and coupling graph (i.e. physical architecture) as inputs, and returns an equivalent quantum circuit that can execute efficiently on the given architecture.

You will encounter considerable amount of design and implementation issues while you are working on this project. Identifying these issues is part of the project. You should start early, allowing enough time for debugging, testing, and improving of your code.

1 Background

1.1 Quantum Circuits and Coupling Graphs

A quantum circuit is a sequence of quantum gates, usually CNOT gates (also known as CX gates) and single-qubit gates. The CX gate uses two qubits, and has the effect that the "target" qubit will be flipped if the "control" qubit is true.

Quantum gates which utilize different qubits can be executed in parallel. For example, the following three gates can all be scheduled to start in the same cycle:

```
h q[0];
t q[1];
cx q[2], q[3];
```

Two-qubit gates, such as CX, can only execute if the physical architecture allows the two qubits to interact. The graph of possible qubit-interactions is called the **coupling map**. For example, Fig. 1 shows the coupling map of the IBM QX2 quantum architecture.

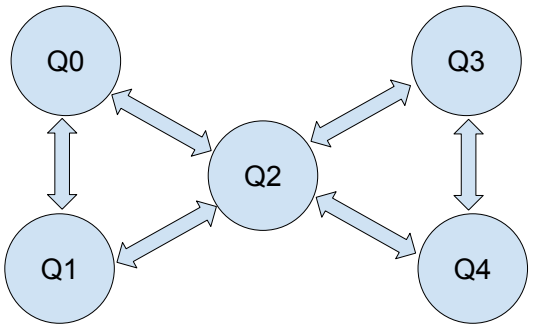

Figure 1: IBM QX2 coupling map, with five qubits and six bidirectional edges.

There is no guarantee that a given circuit will be able to execute as-is on a given architecture. For example if the circuit has a CX that uses qubits Q0 and Q3, then it will not be able to execute on the IBM QX2 architecture, since there is no edge connecting those qubits in the coupling map.

1.2 The Qubit Mapping Problem

The qubit mapping problem deals with transforming a quantum circuit so that it can execute on the desired architecture. There are two interrelated aspects to this problem: selecting an initial mapping of logical qubits to physical qubits, and deciding how to change the mapping (via Swap instructions) in the middle of the circuit.

For example, consider the following circuit which implements parrt of QFT (quantum Fourier transform) for 4 qubits:

```
cx q[0],q[1];
cx q[0],q[2];
cx q[0],q[3];
cx q[1],q[2];
cx q[1],q[3];
cx q[2],q[3];
```

Suppose we want to run this circuit on the architecture from Fig. 1. Even if we change the qubit IDs in order to effectively map them onto different physical qubits, it will be unable to execute. However, if we choose a good initial mapping and also insert a single SWP (swap) gate, then it becomes possible. The following, equivalent circuit satisfies the constraints of the coupling map:

```
cx q[2], q[0]; //originally cx q[0], q[1];
cx q[2], q[1]; //originally cx q[0], q[2];
cx q[2], q[3]; //originally cx q[0], q[3];
cx q[0], q[1]; //originally cx q[1], q[2];
swp q[2], q[3]; //swapping logical qubits 0 and 3
cx q[0], q[2]; //originally cx q[1], q[3];
cx q[1], q[2]; //originally cx q[2], q[3];
```

Merely generating an executable circuit is trivial – for example, you could use any arbitrary initial mapping, and handle the CX gates one-at-a-time by greedily swapping the relevant two qubits toward one another until the gate becomes executable.

The challenge lies in finding an efficient mapping. Each swap instruction will take time to execute, during which no other gates can use its qubits. The problem of finding the optimal mapping is NP-complete.

2 Implementation Details

We will provide you with code that parses the input files, producing a dependency graph for the quantum gates and a list of edges from the coupling map. You are not required to use this code, but your program must accept input files of the same format as ours. The program takes two file-paths as arguments – the circuit and the coupling map – as well as an optional argument to specify gate latencies.

2.1 Data Structure

The provided dependency graph for the circuit's gates is built using the following 'GateNode' class:

```
class GateNode {
    public:
        string name;
        int control;
        int target;
        GateNode * controlChild;
        GateNode * targetChild;
        GateNode * controlParent;
        GateNode * targetParent;
};
```

The **name** variable is the gate's name, such as "cx" or "h". The **control** and **target** variables are the (logical) qubits; target is always used, but control will be -1 for single-qubit gates. The remaining variables are points to the gate's parent(s) and child(ren) in the dependency graph. In cases where there is no such gate – i.e. a root gate with no parents, a leaf gate with no children, and/or a single-qubit gate with no control-qubit dependencies – the pointer will be set to 0 (NULL).

The provided encoding for the coupling map is an std::set where each element is a pair of integers. The coupling map is assumed to be bidirectional, and so every edge in this set will be of the format (x,y) where x is less than y.

2.2 Input Format

We have provided code for parsing the input files, but we nonetheless describe them below.

First, the quantum circuit is a QASM-format text file. For example, the qft-4 circuit from earlier would look like:

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

The qasm file starts with a one-line header, then typically a line that imports a file defining gate behavior, then one or more lines defining arrays of quantum and classical qubits that may be used in the rest of the circuit, and finally the quantum gates.

Next, the coupling map is a sequence of integers. The first line contains the number of nodes (qubits) in the graph and the number of edges. The next however-many lines each contain a pair of qubits representing the edge. For convenience, our coupling map files also include a drawing of the architecture, which our program will not attempt to parse. For example, the input file corresponding to Fig. 1 would look like:

```
5 6
0 1
0 2
1 2
2 3
2 4
3 4

4   0
|\ /|
| 2 |
|/ \|
3   1
```

2.3 Output

You must generate a circuit which is equivalent to the original circuit, but is able to run on the given architecture. You are expected to both select a good initial mapping, and insert SWP (swap) gates as necessary.

The output's format is QASM, same as the input format for the quantum circuit. Note that comments with double-slashes are allowed. We recommend utilizing comments to add useful information such as the original (logical) qubits for each gate, as well as any other info that might be useful for debugging and understanding your program's performance.

2.4 Gate Latency

We describe the latency of each gate as an integer number of cycles. The simplest case is to assume that every gate takes one cycle to complete, but in practice it is typically the case that two-qubit gates (such as CX) take longer than single-qubit gates, and that swap instructions take longer than ordinary two-qubit gates. As such, we specify three distinct latency values when running the program.

3 Grading

Your programs will be graded primarily on functionality. Grading will be done on *ilab* machines. You will receive 0 credit if we cannot compile and run your code on *ilab* machines.

4 How to Get Started

The code package for you to start with is provided on **Sakai**. Create your own directory on the ilab cluster, and copy the entire provided project folder to your home directory or any other one of your directories. Make sure that the read, write, and execute permissions for groups and others are disabled (`chmod go-rwx <directory_name>`).

4.1 Compilation and Execution

Compilation: We have provided a **Makefile** in the sample code package. Modifying the Makefile will likely be necessary as you add files to your project, but as it stands the Makefile offers the following commands:

- **make** should compile all code into an executables called **mapper**.
- **make clean** should remove all files that were generated by Make, including object files, executables, and the tar file.
- **make debug** compiles the executable with flags that disable optimizations and add debugging information. Note that you may need to run **make clean** prior to switching between the debug and default versions.

Execution: The mapper program transforms the provided quantum circuit by modifying the initial mapping and/or inserting swap instructions.

You must specify the file-paths for the quantum circuit and coupling map when you run the program. You can also specify the latencies (which otherwise default to all 1s). For example, to map qft4 onto the QX2 architecture such that single-qubit gates take 1 cycle, two-qubit gates take 2 cycles, and swaps take 6 cycles, you would use the command:
./mapper qft4.qasm qx2.txt -latency 1 2 6

4.2 Program Output:

The program should output the transformed circuit to standard out (e.g. using the **std::cout** or **printf** commands). You should also output the initial mapping that was determined by your program.

5 What to Submit

5.1 Basic Components

Given any qasm file and hardware coupling graph, return an initial mapping, as well as a transformed circuit with inserted swaps (if swaps are necessary).

More information will be given as we start talking about quantum compilation later this semester.

For now, a good warm-up exercise would be to check whether there is any initial mapping that does not require any swaps for a given circuit. In the test cases we provide to you, some of them have a perfect initial mapping, and some don't.

If you work on the warm-up component, we will test your assignments for you.

5.2 Project Report

You are also expected to submit a PDF report, at least six pages in length (at the end of the semester). This report should describe your algorithm(s), as well as document your results on each of the benchmarks including: number of gates in the produced circuit, depth (in cycles) of the produced circuit, and how long your program took to run. You may compare different algorithm choices and discuss their tradeoffs.

Lastly, submit a tar file containing your project's code. Please do not include any quantum circuits, coupling maps, or executables.

6 Questions

Questions regarding this project can be posted on Sakai forum. Good luck!