Lecture 14: Quantum Program Compilation and Optimization

November 27, 2018

1 Quantum Compilation

In this lecture we are going to look at a bunch of compilation techniques and phrase them in terms of NISQ era machines, some of the work which is recent is targeted at NISQ era machines, older work is targeted at large machines. But techniques generally apply, we are using those techniques in the software toolchain that we have right now, so we will discuss them in the current context. In the intro we said, compilation for quantum machines is very similar to classical circuit synthesis. In the classical setting, there is an analogous technique, where we take some sort of high-level language (C-like, Verilog, etc) and compile it all the way down to transistors. The toolchain that we have looks very much like that because, in the end, we are basically dealing with bits and communication links – bits and wires, a lot like the classical circuit synthesis.

We are starting to see differences in the NISQ era because we are now looking at applications that e.g. cooperate with classical machines. So, the old model that we have for compiling quantum programs is very static. We take a program and go down to a static circuit, but in the newer applications where we have classical and quantum computation cooperating, we may get a result from quantum computation, that may affect the classical computation and that may, in turn, kick off a different quantum computation. That model almost does not fit well to the way we do things currently in our research group. We are trying to figure out how to be able to do things that are a little more dynamic. But for now, the basic model is that we are doing something that looks like classical circuit synthesis. The execution model (see fig. 1) includes some sort of classical control, like a microprocessor, controlling some quantum device (call it QPU, think of it like a GPU). You can send instructions to it or gates every cycle and tell it what to do. It’s a simplistic model, it’s not really implemented this way, but it’s a good enough abstraction for the compiler (compiler thinks of it this way). The compiler can emit instructions and assumes that it can control the machine every cycle. There is some amount of backward flow, where you can do measurements. In the simplest model, it’s completely static – we assume that the quantum program executes the way we envisioned it or as the compiler analyzed it to be statically and it does not change depending upon the dynamic behavior of the program.

Figure 1: Execution Model
Figure 2: Classical Control Resolution

We are mostly just compiling into the straight-line code, a whole bunch of quantum gates, executed in that order. Sometime in the future, need to be able to dynamically recompile that program when we get some new measurement. Not today though.

2 Compiler Optimization

Classical optimization steps:

- loop unrolling
- procedure cloning
- inter-procedural constant propagation

Standard ways of dealing with classical programs. We use the LLVM compiler framework that has all these except procedure cloning (non-standard).

In Fig. 2, we have some Scaffold code, C-like, in the first procedure, we have a conditional and we have a loop in the main procedure that calls the first procedure (twice). First, we do loop unrolling so that now instead of having just one instance of the procedure call, we call it twice. This shows an example where \( i < 2 \) but commonly these loops might have a non-constant bound, then the constant propagation will turn the variable into a static quantity. In other words, the reason constant propagation is so important is that most quantum programs are very static, in that we know all the inputs to the program. The difference between most classical analysis and quantum is that the quantum programs are pretty small and they are very specific and part of the reason they are so specific is because it’s so important to minimize the number of qubits and gates that we use in the quantum program that we are willing to compile the program for a specific input. This is often times the case for circuit synthesis as well. If you are building a piece of hardware that needs to be specialized, for example, to do a fast Fourier transform of a very specific size. It could be parametrized by size, but you know what size it will be. In the quantum case, we might even put in specific numbers that we want to deal with, e.g. factoring the product of two primes in Shor’s algorithm. We will compile a specific instance of Shor’s algorithm where we will take the specific key we want to break, we put it right in there and we compile a program in there. We will shortly see that because the runtime of the program you are compiling for on a classical machine tends to be intractable (days, years of execution time), so we are willing to spend a phenomenal amount of compilation time on a specific instance of a program that we want to execute. We may spend days or hours of compilation time to optimize the program so it can run on a quantum machine. This model works great when things are all static. In the quantum processing model, we run it once and immediately we have new parameters so we need to run it again then having days/hours of compilation time is not a good thing in that loop. We are still figuring this out – if we compile something very
optimally using hours of compute whether we can figure out if we can fix it up with just one parameter change instead of recompiling the whole thing (partial compilation techniques).

(Aside: An open question: What does a universal quantum assembly language look like? Easy to say that it will look like quantum gates but no one has ever tried to define an existing assembly language using quantum gates, we use variants of QASM (quantum assembly). Once you try adding classical control, what will that look like? Could we use some vendor’s language (like MIPS or Arm or x86), but there is no agreement on a universal control language.)

Next thing after loop unrolling, we do is procedure cloning which is that we can specialize the procedures, because we know the inputs, e.g. instead of having a single procedure with the conditional in it, we get rid of the conditional by making two versions of the procedures which now do one or the other thing depending upon the input. We rename the procedures accordingly. We also see inter-procedural constant propagation happening here as one of the inputs is eliminated after being moved from the main procedure to the two cloned procedures and being used up in their names. The whole goal is to turn the given code into one big flat line code, once we have that, what’s nice is that there is no classical control. Then we can think of it as a big computational graph. Then we can apply a few other optimizations such as:

- Commutativity
- Circuit template optimizations
- scheduling and mapping (can be done in any order)

These all can be done at the logical level or the physical level. In NISQ machines, we just have the physical level as there is no error correction. While in error-corrected machines, we have both logical (fault-tolerant) and physical level. In the latter, it makes sense to apply these optimizations at both levels. We will see that the early work was all logical and the later work is all physical, but they can be mixed up. In LLVM, these optimizations are applied serially in a single pass, but it’s not always clear which optimization should go first and which later, sometimes you need to redo the same optimization. They are serial (one pass-through) because it is streaming, but the complexity goes up if one goes back to redo optimizations but one might get a better-optimized code.

What we saw above are all pass-driven approaches to optimization. There is another approach called instrument-driven is somewhat strange in that you generate a program that generates your program. Take a C program with quantum code, turn into an executable classical program which has all the classical control such as conditionals and loops and then print out the program as it runs. There are some advantages to this approach, especially in the past, to make the analysis more tractable. With really large programs and really large machines, the static version would blow up in memory usage. This is more scalable. For NISQ machines, scalability is not our main concern, making it optimal is more important.

We mainly focus on pass-driven approaches these days, but sometimes a mix and match approach between the two approaches is useful.
2.1 Case Studies

2.1.1 Large Machines

In Fig. 3, we can see that, e.g., for Binary Welded Tree (BWT) benchmark instrumentation-driven approach is much smaller in compilation time.

In Fig. 4, we see that the more you collapse the structure of the program (flattening inlined procedures), the more parallelism can be obtained. The critical path length is reducing with more flattened code. The flatter the code, the more the compiler sees and more parallelism is discovered. The red line shows the analysis time increases with more flattening. This becomes intractable very quickly.

**Template Optimization** is basically recognizing circuit patterns and coming up with more efficient substitutions. Fig. 5 shows a template in the form of rings, where each ring is a template of circuit patterns where you can go from one pattern to another. What you can do is look for a start state and an end state and then look for the shortest path between those two states. This is implemented at the logical level in the Scaffold compiler. You can see that for classical arithmetic, it can reduce the number of classical gates required by a substantial amount.

2.1.2 NISQ Machines

The NISQ type optimizations are much more expensive. Two examples from recent work submitted to ASPLOS.

1. **Optimal Control** This is an interesting approach where we want more optimization for NISQ machines and we can do that by breaking the abstraction of nice passes and layers in the software (ISA, for example). The old flow looks like what we see on the left side of fig. 6. In the old flow, we go from quantum gates (quantum assembly in the figure) to microwave pulses one at a time vs in this approach, we take assembly instructions and group them together into blocks of pulses and we find that it is way more efficient this way (5-10x).

**QAOA Example**, see fig. 7: Here we see grouping into blocks in the bottom circuit. The reason we group is because it is too expensive to do gradient descent on the start state and the end state of the whole circuit. Even at 10 qubits, it is very difficult without the use of heuristics. But it is so much easier to do for small groups. For $G_3$, we see that for one gate at a time, we get all these pulses that take a long time (the top graph), while if we run the $G_3$ block with the optimal control gradient descent algorithm, we get many fewer pulses and much shorter time.
Figure 6: Aggregated Instructions for Optimal Control

Figure 7: QAOA Example

See fig. 8. We can see unrolling, flattening, optimizations at the logical level and then at the physical level and then we do the mapping to the gates that we have. This old flow is technology specific depending on what gates are available on the particular machine. In the new flow, we abstract over the specific gates on the machine so we come up with a block of code that doesn’t take a position on what gates we use and match them directly to the physical machine. In this ”aggregation”, there are some things we worry about like maintaining parallelism, keeping the critical path low while deciding which instructions to group together.

In Fig. 9, we see the steps that we go through in this new flow. First, we flatten the program, so it doesn’t look modular anymore. Then we move things around using the commutativity rules to reduce the depth. Then, we schedule and map the instructions. The last step is aggregation where we make groups that are neighboring each other in such a way to avoid dependencies (to maintain parallelism).

In fig. 10, we see some heuristics used for scheduling. We are trying to cover all the edges in the least amount of rounds, so each round is trying to maximally cover as many edges as it can.

In fig. 11, we see lots of different options for aggregation. But we want to avoid dependencies, so for example, $G_2$ and $G_3$ cannot be merged together or it will become serialized. For the green colored gates, though it is not immediately visible from the figure, but $G_6$ can be moved before the other two green gates because it commutes with them.

In fig. 12, we see benchmarks for small circuits. Interesting things are on the right side of the table as parallelism, spatial locality and commutativity are important for the results. For example, if you have high parallelism, then this technique can mess you up.

In fig. 13, we see all the different benchmarks with commutativity analysis (CLS) and aggregation. We observe that mostly aggregation does pretty well against commutativity analysis and hand optimization, and further CLS + Aggregation is even better. This shows the limitation of hand optimization as no matter how smart you are, it is difficult to think about more than 1 and 2 qubit gates for translation to pulses. The automatic aggregation approach can look at much bigger space and get a better fit between the physical machine and the program.

In fig. 14, we see how if the applications are more parallel then, in general, it doesn’t scale as well (on the left). If there is not much parallelization, then aggregation works very well.

In fig. 15, we see more swaps are better (for this IBM machine). If the communication that you have to do involves a lot of swaps, those swaps can be well optimized if they are
grouped together with other operations. The more communication that you have to do, the better this technique does.

In summary, the main thing is that it is a new approach that we avoid the normal software layers and abstraction and by doing that we get really large gains.

2. **Noise-Adaptive Compiler Mappings**

Another interesting thing, using an SMT (Satisfiability modulo theories) solver which is a generalization of a SAT solver. It is basically a constraint solver, given some system with variables that you are optimizing for with a whole bunch of constraints, it uses those constraints to try to prune the search space. In general, it is exponentially complex, but pruning step makes it tractable. This work maps for specific properties of physical quantum machines. It looks at calibration data published for the IBM machine every day over the years to look at how you can map to the machine knowing certain qubits are going to be bad or the links between the qubits are going to be bad. The graph shows (fig. 16) how the coherence times for different qubits change every day. The bottom graph shows the error rate for CNOT gates for a specific pair of qubits. What we find is there are specifically bad qubits to map for the circuits.

In fig. 17, we see an example for Bernstein-Vazirani Algorithm that is mapped to two simple topologies. The links marked with red X’s are bad. The comparison shows the mapping without and with the knowledge of bad links.

Fig. 18 shows the tool flow using the Scaffold compiler that uses LLVM IR. We generate some constraints, e.g. CNOT and measurement centric and then they are fed to the solver. Lot of this work was to figure out how to express these constraints as logical constraints for the solver. Then the system spits out a mapping and a scheduler.

See fig. 19. There are interesting routing heuristics – when you have a CNOT between two qubits, it either only allows you to use a one bend path or reserve a particular rectangle area.

(Fig. 20 shows benchmarks used in this work primarily Bernstein-Vazirani (BV), Hidden Shift (HS) and logic gates and Adder and QFT. We again see a small number of gates, as they are run on the real machines.

Fig. 21 shows overall results for reliability.

Fig. 22 shows something interesting. We see how resilient is one mapping over a one week period.

In summary, we observe up to 18x better reliability (2.9x mean). But this approach may not scale beyond 500 qubits.
Figure 12: Benchmarks

Figure 13: Results

Figure 14: Results

Figure 15: Aggregation vs Locality

Figure 16: Noise-Adaptive Compiler Mappings

Figure 17: Example: Avoid Error-prone Links

Figure 18: Tool Flow

Figure 19: CNOT Routing

Figure 20: Benchmarks

Figure 21: Results

Figure 22: Resilience over a week