5 Beyond Classical

The term *Beyond Classical* is now the preferred term over *Quantum Advantage*, which was the preferred term over the unfortunate term *Quantum Supremacy*. That term was originally coined by Prof. John Preskill to describe a computation that can be run efficiently on a quantum computer but would be intractable to run on a classical computer (Preskill, 2012; Harrow and Montanaro, 2017).

Computational complexity theory is a pillar of computer science. A good introduction, along with extensive literature references, can be found in Dean (2016). There exists a large set of complexity classes. The best known big categories are the following:

- Class P, the class of decision problems (with a yes or no answer) with problem size n that run in polynomial time (n^x).
- Class NP, decision problems with exponential run time (x^n) which can be *verified* in polynomial time.
- Class NP-complete, which is a somewhat technical construction. It is a class of NP problems that other NP-complete problems can be mapped to in polynomial time. Finding a single example from this class falling into P would mean that all members of this class are in P as well.
- Class NP-hard, the class of problems that are at least as hard as the hardest problems in NP. To simplify a little bit, this is the class of NP problems that may not be a decision problem, such as integer factorization, or for which there is no known polynomial-time algorithm for verification, such as the traveling salesman problem (Applegate et al., 2006).

There are dozens of complexity classes with various properties and interrelationships. The famous question of whether $P = NP$ remains one of the great challenges in computer science today (and can be answered jokingly with *yes* – if $N = 1$ or $P = 0$).

The interest in quantum computing arises from the belief that quantum algorithms fall into class BQP, the complexity class of algorithms that can be solved by a quantum Turing machine in polynomial time with an error probability of less than $1/3$. This group is believed to be more powerful than class BPP, the class of algorithms that can be solved in polynomial time by a probabilistic Turing machine with a similar error rate. Stated simply, there is a class of algorithms that can run exponentially faster on quantum machines than on classical machines.

From a complexity-theoretic point of view, since BQP *contains* BPP, this would mean that quantum computers can efficiently simulate classical computers. However, would we run a word processor or video game on a quantum computer? Classical and quantum computing appear complementary. The term *beyond* seems well chosen to indicate that there is a complexity class for algorithms that run tractably only on quantum computers.

To establish the quantum advantage, we will not take a complexity-theoretic approach in this book. Instead, we will try to estimate and validate the results of the quantum supremacy paper by Arute et al. (2019) to convince ourselves that quantum computers indeed reach capabilities beyond those of classical machines.

5.1 10,000 Years, 2 Days, or 200 Seconds

In 2019, Google published a seminal paper claiming to finally have reached quantum advantage on their 53-qubit Sycamore chip (Arute et al., 2019). A quantum random algorithm was computed and sampled 1,000,000 times in just 200 seconds, a result that was estimated to take the world's fastest supercomputer 10,000 years to simulate classically.

Shortly thereafter, IBM, a competitor in the field of quantum computing, followed up by estimating that a similar result could be achieved in just a few days, with higher accuracy, on a classical supercomputer (Pednault et al., 2019). A few days versus 200 seconds is a factor of about $1,000\times$. A few days versus 10,000 years is another factor of $1,000\times$. Disagreements of this magnitude are exciting. How is it that these two great companies disagree to the tune of a combined $1,000,000\times$?

5.2 Quantum Random Circuit Algorithm

In order to make claims about performance, you first need a proper benchmark. Typical benchmark sets are SPEC (`www.spec.org`) for CPU performance and the recent MLPerf benchmarks (`http://mlcommons.org`) for machine learning systems. It is also known that as soon as benchmarks are published, large groups of people embark on efforts to optimize and tune their various infrastructures towards the benchmarks. When these efforts cross into an area where optimizations *only* work for benchmarks, these efforts are called *benchmark gaming*.

The challenge, therefore, is to build a benchmark that is meaningful, general, yet hard to game. Google suggested the methodology of using quantum random circuits (QRC) and cross entropy benchmarking (XEB) (Boixo et al., 2018). XEB observes that the measurement probabilities of a random circuit follow certain patterns, which would be destroyed if there were errors or chaotic randomness in the system. XEB samples the resulting bitstrings and uses statistical modeling to confirm that the chip indeed performed a nonchaotic computation. The math is beyond the scope of this text, so we defer to Boixo et al. (2018) for further details.

How do you construct a random circuit? Initially Google used a set of 2×2 operators and Controlled-Z gates. The choice of this particular set of gates, as well as the corresponding constraints on connectivity, were influenced by the capabilities of the Sycamore chip (Google, 2019).

The problem size with a 53-qubit random circuit is very large. Assuming complex numbers of size 2^3 bytes, traditional Schrödinger full-state simulation would require 2^{56} bytes, or 72 PB of storage; twice that for 16-byte complex numbers. Assuming that a full-state simulation would not be realistic, the Google team used a hybrid simulation technique combining full-state simulation with a simulation technique based on Schrödinger–Feynman path history (Rudiak-Gould, 2006). This method trades exponential space requirements for exponential runtime. The hybrid technique breaks the circuit into two (or more) chunks. It simulates each half using the Schrödinger full-state method, and for gates spanning the divided hemispheres, it uses path history techniques. The performance overhead for those gates is very high, but their numbers are comparatively small. Based on benchmarking of the hybrid technique, as well as evaluation of full-state simulation on a supercomputer (Häner and Steiger, 2017), it was estimated that a full simulation for 53 qubits would take thousands of years, even when run on 1,000,000 server-class machines.

Soon after publication, ways were indeed found to game the benchmark with targeted simulation techniques for this specific circuit type, exploiting some unfortunate patterns in how the circuits were constructed. The benchmark needed to be refined. Fortunately, relatively simple changes, such as the introduction of new gate types, seemed to counter these techniques. Details can be found in Arute et al. (2020).

There are, of course, concerns that this choice of benchmark is a somewhat artificial proposition – an algorithm of no practical use for which no other classically equivalent algorithm exists other than quantum simulation. To play devil's advocate, let's take a pendulum with a magnetic weight and have it swing right over an opposite magnetic pole. The movement will be highly chaotic. Simulating this behavior from some assumed starting conditions can theoretically be done in polynomial time but an enormous amount of computing resources are required to model the motion with accuracy over a prolonged period of time. And even then, it is not possible to model *all* starting conditions – the proverbial flap of a butterfly wing on the other side of earth will eventually influence the motion. If we ran the simulations n times and sampled the final positions, the results would come out as chaotically random, and differ from equivalent physical experiments.

On the other hand, just letting the pendulum swing as a physical system "performs" the problem in real time, using practically no computational resources, and resulting in an equally chaotic, random outcome. Have we really proven the pendulum-swing computer advantage?

This is an intriguing argument, but it is flawed. The pendulum-swing computer is a chaotic, physical, analog, and, most importantly, a nonrepeatable process. The most insignificant changes in the initial conditions will lead to different, unpredictable, and nonrepeatable outcomes. As such, it does not perform a computation (which is why we used the term *performs* above).

A random quantum circuit, on the other hand, is a digital computation. A significant change in the setup, such as modified sequences of different gates or starting from a different initial state, will change the outcome chaotically. However, small changes to parameterized gates, different levels of noise, or modest exposure to errors will not cause the resulting probabilities to change meaningfully; the deviations are bounded. In future machines, quantum error correction will make outcomes even more robust and repeatable.

The key argument stands: A nonchaotic calculation can be computed efficiently on a quantum computer. It runs dramatically less efficiently on a classical machine to the tune of thousands of years, thus proving a quantum advantage.

In all cases, it is just a matter of time until we will be able to run something big *and* meaningful on a quantum computer – perhaps Shor's algorithm utilizing millions of qubits with error correction. In the meantime, let's take a closer look at Google's quantum circuit and estimate how long it would take to simulate it in *our* infrastructure.

5.3 Circuit Construction

There are specific constraints for the gates on the Google chip, and they cannot be placed at random. We follow the original construction rules from Boixo et al. (2018).

The supremacy experiment uses three types of gates, each a rotation by $\pi/2$ around an axis on a Bloch sphere's equator. Note that the definitions of the gates is slightly different from those we presented earlier:

$$X^{1/2} \equiv R_X(\pi/2) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & -i \\ -i & 1 \end{bmatrix},$$

$$Y^{1/2} \equiv R_Y(\pi/2) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix},$$

$$W^{1/2} \equiv R_{X+Y}(\pi/2) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & -\sqrt{i} \\ \sqrt{-i} & 1 \end{bmatrix}.$$

There is a list of specific constraints for circuits:

- For each qubit, the very first and last gates must be Hadamard gates. This is reflected in a notation for circuit depth as *1-n-1*, indicating that n steps, or gate levels, are to be sandwiched between Hadamard gates.
- Apply *CZ* gates in the patterns shown in Figure 5.1, alternating between horizontal and vertical layouts.
- Apply single-qubit operators $X^{1/2}$, $Y^{1/2}$, and T (or $W^{1/2}$) to qubits that are not affected by the *CZ* gates, using the criteria below. For our simulation (using our infrastructure, which does not specialize for specific gates), the choice of gates actually does *not* matter in regards to computational complexity: they are all 2×2 gates, and we can use any of the standard gates. For more sophisticated methodology, like tensor networks, the choice of gates can make a difference.

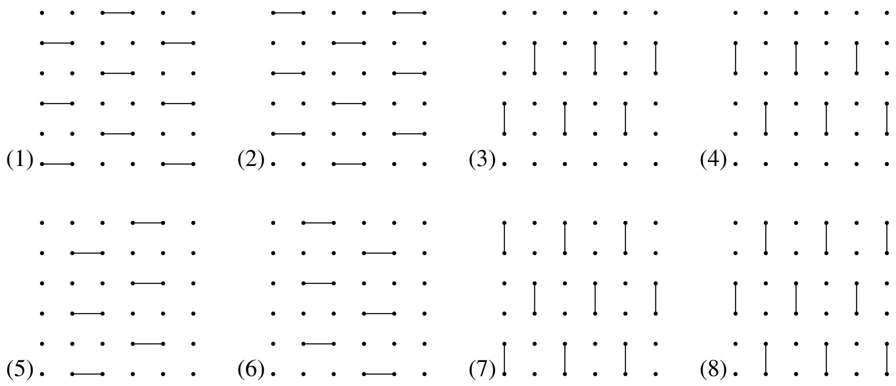

Figure 5.1 Patterns for applying controlled gates on the Sycamore chip.

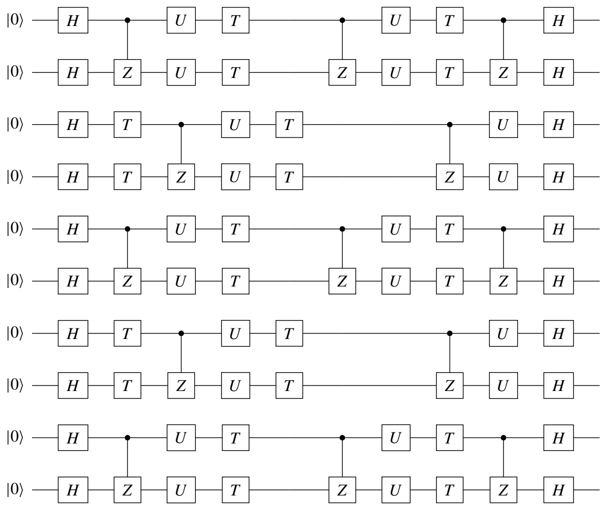

Figure 5.2 A smaller-scale, semi-random supremacy circuit.

  - If the previous cycle had a *CZ* gate, apply any of three single-qubit unitary gates.
  - If the previous cycle had a nondiagonal unitary gate, apply the T-gate.
  - If the previous cycle had no unitary gate (except Hadamard), apply the T-gate.
  - Else, don't apply a gate.
- Repeat above steps for a given number of steps (which we call *depth* in our implementation).
- Apply the final Hadamard and measure.

This interpretation of the rules produces a circuit like the one shown in Figure 5.2. Note that there have been refinements since first publication; Arute et al. (2020) has the details. The main motivation for making changes was to make it harder for the new circuits to be simulated by tensor networks, the most efficient simulation technique for this type of network (Pan and Zhang, 2021). In our case, we are looking for orders of magnitude differences. We stick with this original definition and make sure to apply corresponding fudge factors in the final estimation.

Let's implement this interpretation. Again, it doesn't matter which gates to apply specifically; the simulation time is the same for each gate in our infrastructure. As long as the gate types and density are roughly aligned with the Google circuit, our estimation should be reasonably accurate. Note that other infrastructures, including Google's `qsimh`, do apply a range of optimizations to improve simulation performance.

We encode the patterns as lists of indices, where a nonzero element indicates a *CZ* gate from the current index to the index with the offset found at that location. The eight patterns are then encoded like this:

```
# The paper suggests 8 patterns of size 6*6 of CZ gates. To fully
# encode the patterns one would need at least 36 qubits, but that's
# hard to simulate. We make a compromise and try to apply as many
# gates as possible. The patterns are encoded as simple lists, where
# a non-zero element at index i serves as the control and has
# the offset to the target qubit. To go right, the offset is 1, to
# go down the offset is 6.
#
pattern1 = [0, 0, 1, 0, 0, 0,
            1, 0, 0, 0, 1, 0] * 3

pattern2 = [1, 0, 0, 0, 1, 0,
            0, 0, 1, 0, 0, 0] * 3

[...] # similar for the remaining patterns

patterns = [pattern1, pattern2, pattern3, pattern4,
            pattern5, pattern6, pattern7, pattern8]
```

Gates are represented by a simple enumeration (`H`, `T`, `U`, `CZ`). With this, we are ready to build the circuit. We start from the horizontal and vertical patterns and then proceed to apply the rules as stated above. Note, again, for our simulation the actual gates do not matter:

```
def build_circuit(nbits, depth):
  """Construct the full circuit."""

  def apply_pattern(pattern):
    bits_touched = []
    for i in range(min(nbits, len(pattern))):
      if pattern[i] != 0 and i + pattern[i] < nbits:
        bits_touched.append((i, i + pattern[i]))
    return bits_touched

  print('\nBuild smaller circuit ({} qubits, depth {})\n'.
        format(nbits, depth))
  state0 = [Gate.H] * nbits
  states = []
  states.append(state0)

  for _ in range(depth - 1):
    state1 = [Gate.UNK] * nbits
    touched = apply_pattern(patterns[random.randint(0, 7)])
    for idx1, idx2 in touched:
      state1[idx1] = Gate.CZ
      state1[idx2] = Gate.CZ
    for i in range(len(state0)):
      if state0[i] == Gate.CZ and state1[i] != Gate.CZ:
        state1[i] = Gate.U
      if state0[i] == Gate.U and state1[i] != Gate.CZ:
        state1[i] = Gate.T
      if state0[i] == Gate.H and state1[i] != Gate.CZ:
        state1[i] = Gate.T
    state0 = state1
    states.append(state0)

  state0 = [Gate.H] * nbits
  states.append(state0)
  return states
```

Let's print a sample circuit with 12 qubits and depth of *1-10-1*:

```
def print_state(states, nbits, depth):
  [...]
>>
    0  1  2  3  4  5  6  7  8  9 10 11 12
 0: h  cz cz u  t                       h
 1: h  t  cz u  t                 cz cz h
 2: h  cz u  t              cz u  t  cz h
 3: h  t     cz u  t  cz cz cz cz cz u  h
 4: h  cz cz cz u  t  cz cz u  cz u  t  h
 5: h  t  cz u  t                 cz u  h
 6: h  cz u  t              cz u  t     h
 7: h  t     cz u  t  cz cz cz cz cz u  h
 8: h  cz cz cz u  t  cz cz u  cz u  t  h
 9: h  t  cz u  t                 cz cz h
10: h  cz u  t              cz u  t  cz h
11: h  t                    cz u  cz u  h
```

5.4 Estimation

So far, so good: We have implemented functionality to construct circuits for a given number of qubits and circuit depths. For a bottom-up estimation, we construct a circuit with a smaller, tractable dimension, simulate it, and from the simulation results we extrapolate to a 53-qubit circuit.

We make several simplifying assumptions, most notably, that communication between machines is free. At the end of the estimation, we should apply appropriate factors to account for such overheads.

Simulation is done with an eager execution function, which iterates over the depth of the circuit, simulating each gate one by one:

```
def sim_circuit(states, nbits, depth, target_nbits, target_depth):
  """Simulate the generated circuit."""

  [...]
  qc = circuit.qc('Supremacy Circuit')
  qc.reg(nbits)

  for d in range(depth):
    s = states[d]
    for i in range(nbits):
      if s[i] == Gate.UNK:
        continue
      ngates += 1
      if s[i] == Gate.T:
        qc.t(i)
      [... similar for H, U/V, U/Yroot]
      if s[i] == Gate.CZ:
        ngates += 1  # This is just an estimate of the overhead
        if i < nbits - 1 and s[i + 1] == Gate.CZ:
          qc.cz(i, i+1)
          s[i+1] = Gate.UNK
        if i < nbits - 6 and s[i + 6] == Gate.CZ:
          qc.cz(i, i+6)
          s[i+6] = Gate.UNK
    [...]
```

To estimate the time it would take to execute this circuit at 53 qubits, we make the following assumptions:

- We assume that one-qubit and two-qubit gate application time is linear over the size of the state vector.
- Performance is memory-bound.
- We know we'd have to distribute the computation over multiple machines, but we ignore the communication cost.
- We assume a number of machines and a number of cores on those machines. We know that a small number of cores on a high-core machine can saturate the available memory bandwidth, so we take a guess on what the number of reasonably utilized cores would be (16, but this number can be adjusted).

With these assumptions, the metric *Time per gate per byte in the state vector* is the one we'll use to extrapolate results. It is remarkably stable across qubits and depth and thus can be used to estimate approximate performance of bigger circuits. In order to estimate how many gates there would be in a larger circuit, we compute a gate ratio, which is the number of gates found in a circuit divided by `(nbits * depth)`. In code:

```
print('\nEstimate simulation time on larger circuit:\n')
gate_ratio = ngates / nbits / depth
print('Simulated circuit:')
print('  Qubits                   : {:d}'.format(nbits))
print('  Circuit Depth            : {:d}'.format(depth))
print('  Gates                    : {:.2f}'.format(ngates))
print('  State Memory             : {:.4f} MB'.format(
    2 ** (nbits-1) * 16 / (1024 ** 2)))
print('Estimated Circuit Qubits  : {}'.format(target_nbits))
print('Estimated Circuit Depth   : {}'.format(target_depth))
print('Estimated State Memory    : {:.5f} TB'.format(
    2 ** (target_nbits-1) * 16 / (1024 ** 4)))
print('Machines used             : {}'.format(flags.FLAGS.machines))
print('Estimated cores per server: {}'.format(flags.FLAGS.cores))
print('Estimated gate density    : {:.2f}'.format(gate_ratio))

estimated_sim_time_secs = (
    # time per gate per byte
    (duration / ngates / (2**(nbits-1) * 16))
    # gates
    * target_nbits
    # gate ratio scaling factor to circuit size
    * gate_ratio
    # depth
    * target_depth
    # memory
    * 2**(target_nbits-1) * 16
    # number of machines
    / flags.FLAGS.machines
    # Active core per machine
    / flags.FLAGS.cores)

print('Estimated for {} qbits: {:.2f} y or {:.2f} d or ({:.0f} sec)'
      .format(target_nbits,
              estimated_sim_time_secs / 3600 / 24 / 365,
              estimated_sim_time_secs / 3600 / 24,
              estimated_sim_time_secs))
```

Let's look at a specific result. We assume the target circuit has 53 qubits and is run on 100 machines, each one having 16 fully available cores. The number of gates in our simulation (475) seems to roughly align with the published number of gates from the Google publication, though not exactly. (Google quoted 1,200 gates, while extrapolating the 475 would yield about 2,000 gates.) For these parameters, the estimation results are:

```
Estimate simulation time on larger circuit:
Simulated circuit:
  Qubits                   : 25
  Circuit Depth            : 20
  Gates                    : 412.00
  State Memory             : 256.0000 MB
Estimated Circuit Qubits   : 53
Estimated Circuit Depth    : 20
Estimated State Memory     : 65536.00000 TB
Machines used              : 100
Estimated cores per server: 255
Estimated gate density     : 0.82
Estimated for 53 qbits: 0.01 y or 2.81 d or (242490 sec)
```

Of course, these parameters are hopelessly naive – how would we provision 72 PB of memory on just 100 machines? Assuming we can provision 1 TB per server, we'd need at least 72K hosts. At this scale, we can no longer ignore communication costs. You may want to experiment with more realistic settings.

5.5 Evaluation

For comparison, let's look at the massive Summit supercomputer (Oak Ridge National Laboratory, 2021). It is theoretically capable of performing up to 10^{17} single precision floating point operations per second. To compute 2^{53} equivalents of 2×2 matrix multiplications is of complexity 2^{56}. At 100 percent utilization, it would take Summit just a few seconds to compute a full simulation!

To store a full state of 53 qubits, we need 72 PB bytes of storage. Summit has an estimated 2.5 PB of RAM on all sockets, and 250 PB of secondary storage. This means we should expect that the simulation encounters high communication overhead moving data from permanent storage into RAM. Much of the permanent storage would have to be reserved for this experiment as well. The researchers at IBM found an impressive way to minimize data transfers, which is a major contribution by Pednault et al. (2019). With this technique, a slow-down of about $500\times$ was anticipated, which led to the estimate that a full simulation could run in about two days.

Now let's try to answer the question that started this section: Where does the discrepancy of 10,000 years versus days come from? This is a factor of about $1,000\times$, after all.

The Google Quantum X team based their estimations on a different simulator architecture (Markov et al., 2018), assuming that a full-state simulation is not realistic.

The simulation techniques were benchmarked at smaller scale. Full-state simulation results were evaluated on a supercomputer. From these data points, the computational costs were extrapolated to 1,000,000 machines, arriving at the estimate of 10,000 years of simulation time for 53 qubits and a circuit depth of 20.

The IBM researchers, on the other hand, found an elegant way to squeeze the problem onto one of the biggest supercomputers in the world. The results are only estimated—an experiment was not performed. It is difficult to determine how realistic the estimates are in practice because at petabyte scale, other factors have to be taken into account, for example, disk error rates. This also assumes that most of the machine's secondary storage was committed to the experiment.

Is there a right or wrong? The answer is *no* because we are comparing apples to oranges. The evaluated simulation techniques are different based on different assumptions of what can realistically run on a supercomputer. The supremacy experiment was physically run, the Summit paper was estimated. Even if physical simulation took just a day on Summit, adding a handful of additional qubits will exhaust its storage capacity. The simulation technique would have to change and trade storage requirements for simulation time, similar to the Schrödinger-Feynman path history technique (Rudiak-Gould, 2006). At that point, and only at that point, would we be able to make a more fair, apples-to-apples comparison.

It is safe to anticipate that other smart simulation techniques will emerge. However, as long as $\text{BPP} \subset \text{BQP}$ is true, it is also safe to assume that additional qubits or moderately modified benchmarks will again defeat attempts to simulate these circuits classically.