4 Scalable, Fast Simulation

The concepts and basic infrastructure consisting of tensors, states, and operators, implemented as big matrices and state vectors, are sufficient to implement many small-scale quantum algorithms. All the algorithms in Chapter 3 on simple algorithms were implemented this way.

This basic infrastructure is great for learning and experimenting with the basic concepts and mechanisms of quantum computing. However, for complex algorithms, which typically consist of much larger circuits with many more qubits, this matrix-based infrastructure becomes unwieldy, error-prone, and does not scale. In this chapter we address the scalability problem by developing an improved infrastructure that scales easily to much larger problems. We recommend at least skimming this content before exploring the complex algorithms in Chapter 6. Ultimately, we are building the foundation for a high-performance quantum simulator. You don't want to miss it!

In this chapter, we first give an overview of the various levels of infrastructure that will be developed, with the corresponding computational complexities and levels of performance. We introduce *quantum registers*, which are named groups of qubits. We describe a *quantum circuit model*, where most of the complexity of the base infrastructure is hidden away in an elegant way. To handle the larger circuits of advanced algorithms, we need faster simulation speeds. We detail an approach to apply an operator with linear complexity rather than the $O(n^2)$ method that we started with in Section 2.5.3. We then further accelerate this method with C++, attaining a performance improvement of up $100\times$ over the Python version. For some specific algorithms we can do even better. We describe a sparse state representation, which will be the best-performing implementation for many circuits.

4.1 Simulation Complexity

This book focuses on algorithms and how efficiently simulate them on a classical computer. Quantum simulation can be implemented in a multitude of ways. The key attributes of the various implementation strategies are computational complexity, the resulting performance, and the maximal number of qubits that can be simulated in a *reasonable* amount of time with *reasonable* resource requirements.

The size of the quantum state vector (as described so far) grows exponentially with the number of qubits. For a single qubit, we only need to store two complex

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

numbers, amounting to 8 bytes when using `float` or 16 bytes when using `double` as the underlying data type. Two qubits require four complex numbers; n qubits require 2^n complex numbers. Simulation speed, or the ability to fit a state into memory, is typically measured by the number of qubits at which a given methodology is still tractable. By tractable, we mean that a result that can be obtained in roughly less than an hour. At the time of this writing, the world record in storing and simulating a full wave function stood at 48 qubits (De Raedt et al., 2019).

Because of the exponential nature of the problem, improving performance by a factor of $8\times$ means that we can only handle three more qubits. If we see a speedup of $100\times$, this means we can handle six or seven additional qubits. The following are the five different approaches we describe in this book:

- **Worst**. We have seen this approach already. Implementing gates as potentially very large matrices and constructing operators via matrix-matrix products is of complexity $O(n^3)$. This is the worst case; avoid it if at all possible. It becomes intractable even at relatively small numbers of qubits; performance starts to suffer at $\mathrm{N} \sim 8$.
- **Bad**. Instead of constructing big-matrix operators, you can apply gates to the state vector one at a time as matrix-vector products. This leads to complexity $O(n^2)$, which is already a substantial improvement and reaches a number of qubits $N \sim 12$ before becoming intractable.
- **Good**. In Section 4.4, we will learn that one- and two-qubit gates can be applied with a linear traversal over the state, resulting in complexity $O(n)$, a massive improvement over the first two approaches, reaching $N \sim 16$ before becoming intractable.
- **Better**. We started our journey with Python, but we can go faster by using C++. With this approach, in Section 4.5 we will implement the previous `apply` functions in C++, extending Python with its foreign function interface (FFI). Although this approach still has complexity $O(n)$ as above, the performance gain of C++ over Python is about $100\times$ and can reach $N \sim 25$ qubits, depending on the problem.
- **Best**. Using this approach, in Section 4.6 we will change the underlying representation to a sparse one, saving memory and reducing iterations. Additionally, gate application is efficient. This approach is $O(n)$ in the worst case but can win over other implementations by a significant factor. The improvements are possible because for many circuits, the number of nonzero probability states is less than 3%, often even lower. With this, we may reach $N \sim 30$ qubits, or even more for some algorithms.

We could improve our proposed techniques (which are also called *Schröedinger full-state simulations*) further with vectorization (perhaps add one or two qubits), parallelization (64 cores might add $\log_2(64) = 6$ additional qubits), or even by using TPU SuperPODs with 4,096 TensorCores. We could employ machine clusters with 128 or more machines, and corresponding additional qubits, to reach simulation capability of around 45 qubits, utilizing 512 TB of memory. Today's supercomputers would add

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

another handful of qubits (if they were fully dedicated to a simulation job, including all their secondary storage).

These aforementioned techniques are mostly standard High-Performance Computing (HPC) techniques. Since they don't add much to the exposition here, we will not discuss them further. We list a range of open-source solutions in Section 8.5.10, several of which do support distributed simulation (the transpilation techniques detailed in Section 8.5 allow the targeting of several of these simulators). What these numbers demonstrate is how quickly simulation hits limits. Improving performance or scalability by $1,000\times$ only gains about 10 qubits. Adding 20 qubits results in $1,000,000\times$ higher resource requirements.

There are other important simulation techniques. For example, the Schrödinger–Feynman simulation technique, which is based on path history (Rudiak-Gould, 2006; Frank et al., 2009). This technique trades performance for reduced memory requirements. Other simulators work efficiently on restricted gate sets, such as the Clifford gates (Anders and Briegel, 2006; Aaronson and Gottesman, 2004). Furthermore, there is ongoing research on improving simulation of specific circuit types (Markov et al., 2018; Pan and Zhang, 2021). As exciting as these efforts are, they are beyond the scope of this book.

4.2 Quantum Registers

For larger and more complex circuits, we want to make the formulation of algorithms more readable by addressing qubits in named groups. For example, the circuit in Figure 4.1 has a total of eight qubits. We want to name the first four `data`, the next three `ancilla`, and the bottom one `control`. In the example, the gates are just random. On the right side, it shows the global qubit index as g_x, as well as the index into the named groups; for example, global qubit g_5 corresponds to register $ancilla_1$.

These named groups of adjacent qubits are called *quantum registers*. In classical machines, a register typically holds a single value (ignoring vector registers for a moment). A group of registers, or the full physical implementation of registers in hardware, is what is typically called a register file. In that sense, because a quantum register is a named group of qubits, it is more akin to a classical register file, like a group of pipes in a church organ register.

The state of the system is still the tensor product of all eight (global) qubits, numbered from 0 to 7 (g_0 to g_7). At the same time, we want to address `data` with indices ranging from 0 to 3, which should produce the global qubit indices 0 to 3 in the combined state; we want to address `ancilla` from 0 to 2, resulting in global qubit indices 4 to 6; and we want to address `control` with index 0, resulting in global qubit index 7. In code, a simple lookup table will do the trick.

The initial implementation is a bit rough. No worries, we will wrap this up nicely in the next section. We introduce a Python class `Reg` (for "Register") and initialize it by passing the size of the register file we want to create *and* the current global offset,

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

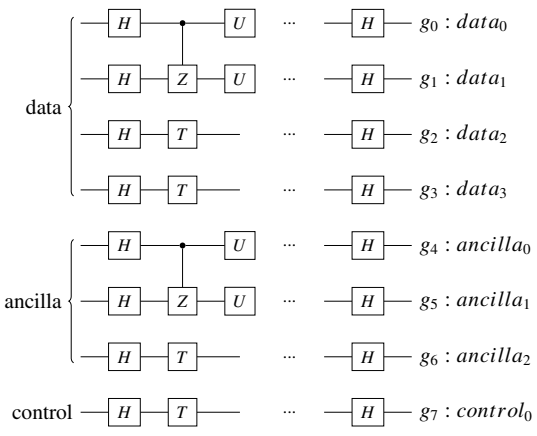

Figure 4.1 Quantum registers *data*, *ancilla*, and *control*.

which has to be manually maintained for this interface. In the example above, the first global offset is 0, for the second register it is 4, and for the last register it is 7.

By default, the states are assumed to be all $|0\rangle$, but an initializer, `it`, can be passed as well. If `it` is an integer, it is converted to a string with the number's binary representation. If `it` is a string (including after the previous step), tuple, or list, the lookup table is initialized with 0s and 1s according to the binary numbers passed. Again, the ordering is from most significant to least significant qubit.

```
class Reg():
  def __init__(self, size: int, it=0, global_reg: int = None):
    self.size = size
    self.global_idx = list(range(global_reg,
                                 global_reg + size))
    self.val = [0] * size

    if it:
      if isinstance(it, int):
        it = format(it, '0{}b'.format(size))
      if isinstance(it, (str, tuple, list)):
        for idx, val in enumerate(it):
          if val == '1' or val == 1:
            self.val[idx] = 1
```

For example, to create and initialize `data` with $|1011\rangle$ and `ancilla` with $|111\rangle$, which is the binary representation of decimal 7, and to access global qubit 5, we write:

```
data = state.Reg(4, (1, 0, 1, 1), 0)    # 0b1011
ancilla = state.Reg(3, 7, 4)    # 0b111

# Access global qubit 5 as:
. . . = ancilla[1]
```

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

To give a textual representation of the register with the initial state, we write a short dumper function to print the register in state notation:

```
def __str__(self) -> str:
    s = '|'
    for _, val in enumerate(self.val):
        s += f'{val}'
   return s + '>'
```

The code so far is simple but good enough for our next use cases. It does not, for example, allow the initialization of individual registers in superposition. To get the global qubit index from a register's index, we use this function, which allows getting a global register by simply indexing into the register, for example, `greg = ancilla[1]`:

```
 def __getitem__(self, idx: int) -> int:
   return self.global_idx[idx]
```

To initialize a specific qubit (with 0 or 1) at a register index:

```
 def __setitem__(self, idx: int, val: int) -> None:
   self.val[idx] = val
```

To get the size of a register, we also add this standard function:

```
@property
 def nbits(self) -> int:
   return self.size
```

After all this setup, we still have to create an actual state from the register, with:

```
 def psi(self) -> State:
   return bitstring(*self.val)
```

We must only call this function once per initialized register, as a final step. Modifying the initialization value of a register after the state has been created has no effect on the created state. This may not be the most elegant way to do this, but it is compact and sufficient for our purposes. We do not show any code examples here, because we are going to develop a much nicer interface next.

4.3 Circuits

So far we have used full state vectors and operator matrices to implement the initial algorithms. This infrastructure is easy to understand and works quite well for algo-

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

rithms with a small number of qubits. It is helpful for learning, but the representation is explicit. It exposes the underlying data structures, and that can cause problems:

- Describing states and operators explicitly at a very low level of abstraction requires a lot of typing, which is error-prone.
- The representation exposes the implementation details. Changing aspects of the implementation would be difficult – all users of the base infrastructure would have to be updated.
- A minor point to make is that this style of representation differs from that commonly found in existing frameworks such as Qiskit (Gambetta et al., 2019) or Cirq (Google, 2021c).

The second problem is especially important in our context, as we want to develop faster ways to apply gates, both in Python and C++-accelerated Python. We might want to change the representation of states themselves from storing a full state vector to a sparse representation. The current level of abstraction does not allow that without changing all dependent client code.

In order to remedy these problems, we create a data structure we call a quantum circuit `qc`. It wraps up and nicely packages all the functions we have discussed so far. The naming convention is all lowercase to distinguish from the explicit representation discussed in Chapters 2 and 3.

A circuit's constructor accepts a string argument to assign a name to a circuit. This name is consequently used in printouts. The circuit has an internally stored quantum state that we initialize to the scalar value of 1.0, indicating that there is no qubit in this circuit right after creation. Here is the constructor:

```
class qc:
 """Wrapper class to maintain state + operators."""

 def __init__(self, name=None):
   self.name = name
   self.psi = 1.0
   state.reset()
```

4.3.1 Qubits

The circuit class supports quantum registers, which immediately add a register's qubits to the circuit's full state. This is the place where the global register count is maintained, hiding away the rough earlier interface of the underlying `Reg` class:

```
def reg(self, size: int, it, *, name: str = None):
   ret = state.Reg(size, it, self.global_reg)
   self.global_reg = self.global_reg + size
   self.psi = self.psi * ret.psi()
   return ret
```

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

To add individual qubits to the circuit, we wrap the various constructor functions discussed earlier with corresponding member functions of `qc`. Each of these generator functions immediately combines the newly generated qubits with the internal state. In order to allow mixing of qubits and registers, we have to update the global register count as well.

```
def qubit(self,
          alpha: np.complexfloating = None,
          beta: np.complexfloating = None) -> None:
    self.psi = self.psi * state.qubit(alpha, beta)
    self.global_reg = self.global_reg + 1

def zeros(self, n: int) -> None:
    self.psi = self.psi * state.zeros(n)
    self.global_reg = self.global_reg + n

def ones(self, n: int) -> None:
    self.psi = self.psi * state.ones(n)
    self.global_reg = self.global_reg + n

def bitstring(self, *bits) -> None:
    self.psi = self.psi * state.bitstring(*bits)
    self.global_reg = self.global_reg + len(bits)

def arange(self, n: int) -> None:
    self.zeros(n)
    for i in range(0, 2**n):
        self.psi[i] = float(i)
    self.global_reg = self.global_reg + n

def rand(self, n: int) -> None:
    self.psi = self.psi * state.rand(n)
    self.global_reg = self.global_reg + n
```

Of course, we have to have the ubiquitous `nbits` property, which we forward to the state's function of the same name:

```
@property
 def nbits(self) -> int:
   return self.psi.nbits
```

4.3.2 Gate Application

In order to apply gates to qubits, assume for now that there are two functions to do that, one each for single gates and for controlled gates. We will detail their implementation in the following sections. For now, let us pretend that these functions will apply gates at index `idx`, while updating the internal state.

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

```
def apply1(self, gate: ops.Operator, idx: int,
           name: str  = None, *, val: float = None):
  [...]

def applyc(self, gate: ops.Operator, ctl: int, idx: int,
           name: str = None, *, val: float = None):
  [...]
```

The function `apply1` applies the single-qubit gate `gate` to the qubit at `idx`. The gate to be applied may get a name. Some gates require parameters, e.g., the rotation gates, which can be specified with the named parameter `val`.

The function `applyc` operates the same way, but it additionally gets the index of the controlling qubit `ctl`.

4.3.3 Gates

With these two apply functions in place, we can now wrap all standard gates as member functions of the circuit. This is mostly a straightforward wrapping, except for the double-controlled X-gate and the corresponding `ccx` member function, which uses the special construction we saw earlier in Section 3.2.7.

```
def cv(self, idx0: int, idx1: int) -> None:
  self.applyc(ops.Vgate(), idx0, idx1, 'cv')

def cv_adj(self, idx0: int, idx1: int) -> None:
  self.applyc(ops.Vgate().adjoint(), idx0, idx1, 'cv_adj')

def cx(self, idx0: int, idx1: int) -> None:
  self.applyc(ops.PauliX(), idx0, idx1, 'cx')

def cu1(self, idx0: int, idx1: int, value) -> None:
  self.applyc(ops.U1(value), idx0, idx1, 'cu1', val=value)

# [... similar for cy, cz, crk]

def ccx(self, idx0: int, idx1: int, idx2: int) -> None:
  """Sleator-Weinfurter Construction."""

    self.cv(idx0, idx2)
    self.cx(idx0, idx1)
    self.cv_adj(idx1, idx2)
    self.cx(idx0, idx1)
    self.cv(idx1, idx2)

def toffoli(self, idx0: int, idx1: int, idx2: int) -> None:
  self.ccx(idx0, idx1, idx2)

def h(self, idx: int) -> None:
  self.apply1(ops.Hadamard(), idx, 'h')
```

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

```
def t(self, idx: int) -> None:
  self.apply1(ops.Tgate(), idx, 't')

# [... similar for u1, v, x, y, z, s, yroot]

def rx(self, idx: int, theta: float) -> None:
  self.apply1(ops.RotationX(theta), idx, 'rx')

# [... similar for ry, rz]

# This doesn't go through the apply functions. Don't use.
def unitary(self, op, idx: int) -> None:
  self.psi = ops.Operator(op)(self.psi, idx, 'u')
```

All these gates can be applied with our still-hypothetical two apply functions, except the `unitary` function. This function allows the application of an arbitrarily sized operator, falling back to the full matrix implementation. In the context of `qc`, this function is an abomination. As a matter of fact, we don't use it for any of the examples and algorithms in this book. We only added it for generality. Don't use it.

4.3.4 Adjoint Gates

We have wrapped operators as member functions of the `qc` class. But what about adjoints? There appear to be two obvious design choices.

Explicit wrapping. For each gate, offer the apply function as above, as well as a function to apply the adjoint. For example, for the CV-gate, we would offer a member function `qc.cv(...)` as well as `qc.cv_adj(...)`.

Alternatively, we could *add two static functions*:

```
    def id(gate: ops.Operator) -> ops.Operator:
        return gate
    def adjoint(gate: ops.Operator) -> ops.Operator:
        return gate.adjoint()
```

And add the `id` function as a default parameter to each gate application function. For example, for the S-gate:

```
    def s(self, idx: int, trans: Callable = id)  -> None:
        self.apply1(trans(ops.Sgate()), idx, 's')
```

And to apply the adjoint function, call the function with the `adjoint` function as parameter:

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

```
    qc.s(0, circuit.adjoint)
```

This is certainly elegant, especially for compiled languages, which can optimize away the overhead from this construction. Python is relatively slow as is, and we don't want to further slow it down, so we go with the first alternative – we add individual apply functions for adjoint gates, as needed for the code examples.

4.3.5 Measurement

We wrap the measurement operator in a straightforward way:

```
def measure_bit(self, idx: int, tostate: int = 0,
                collapse: bool = True) -> (float, state.State):
  return ops.Measure(self.psi, idx, tostate, collapse)
```

Note that we construct a full-matrix measurement operator, which means this way of measuring won't scale. Fortunately, in many cases we don't have to perform an actual measurement to determine a most likely measurement outcome. We can just look at the state vector and find the one state with the highest probability – we can do *measurement by peek-a-boo*.

For convenience, we also add a statistical sampling function. Its parameter is the probability of measuring $|0\rangle$. For example, we could provide the value 0.25. The function picks a random number in the range of 0.0 to 1.0. If the probability of measuring $|0\rangle$ is lower than this random number, it means that we happened to measure a state $|1\rangle$.

```
def sample_state(self, prob_state0: float):
   if prob_state0 < random.random():
     return 1  # corresponds to |1>
   return 0  # corresponds to |0>
```

To a degree this is silly, as in our infrastructure we know the probabilities for any given state. We don't have to sample over the probabilities to obtain probabilities we already know. Nevertheless, some code might be written in a way as if it would run on an actual quantum computer, and that would make sampling necessary. To mimic this, and also to mirror code that can be found in other infrastructures, we offer this function.

4.3.6 Swap Operations

We also add implementations of the Swap gate (`swap`) and Controlled Swap gate (`cswap`), as described earlier in Sections 2.7.3 and 2.7.4. The `cswap` gate, specifically,

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

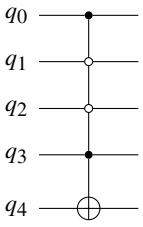

Figure 4.2 A multi-controlled X-gate.

will be used later in Shor's algorithm (Section 6.6). It is easy to implement by simply changing the `cx` gates in a Swap gate to double-controlled `ccx` gates:

```
def swap(self, idx0: int, idx1: int) -> None:
    self.cx(idx1, idx0)
    self.cx(idx0, idx1)
    self.cx(idx1, idx0)

def cswap(self, ctl: int, idx0: int, idx1: int) -> None:
    self.ccx(ctl, idx1, idx0)
    self.ccx(ctl, idx0, idx1)
    self.ccx(ctl, idx1, idx0)
```

4.3.7 Multi-Controlled Gates

To build multi-controlled gates as outlined in Section 3.2.8, we use the following code. We make it quite fancy:

- For the controlling gates, we allow 0, 1, or 2 or more controllers. This makes this implementation quite versatile in several scenarios.
- We allow for Controlled-By-1 gates and Controlled-By-0 gates. To mark a gate as Controlled-By-0, the index `idx` of the controller is passed as a single element list item `[idx]`.

For the example in Figure 4.2, for the X-gate on qubit q_4, which is controlled by By-1 and By-0 control qubits, we make the following function call. Of course, we have to make sure we have reserved enough space for the ancillae in the `aux` register:

```
qc.multi_control([0, [1], [2], 3], 4, aux, ops.PauliX(), 'multi-X'))
```

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

Here is the full implementation. We also modified the function `applyc` to enable Controlled-By-0 gates by emitting an X-gate before and after the controller qubit (not shown here).

```
def multi_control(self, ctl, idx1, aux, gate, desc: str):
  """Multi-Controlled gate, using aux as ancilla."""

  # This is a simpler version that requires n-1 ancillae, instead
  # of n-2. The benefit is that the gate can be used as a
  # single-controlled gate, which means we don't need to take the
  # root (no need to include scipy). This construction also makes
  # the Controlled-By-0 gates a little bit easier, those controllers
  # are being passed as single-element lists, eg.:
  #   ctl = [1, 2, [3], [4], 5]
  #
  # This can be optimized (later) to turn into a space-optimized
  # n-2 version.
  #
  # We also generalize to the case where ctl is empty or only has 1
  # control qubit. This is very flexible and practically any gate
  # could be expressed this way. This would make bulk control of
  # whole gate sequences straightforward, but changes the trivial
  # IR we're working with here. Something to keep in mind.
  with self.scope(self.ir, f'multi({ctl}, {idx1}) # {desc})'):
    if len(ctl) == 0:
      self.apply1(gate, idx1, desc)
      return
    if len(ctl) == 1:
      self.applyc(gate, ctl[0], idx1, desc)
      return

    # Compute the predicate.
    self.ccx(ctl[0], ctl[1], aux[0])
    aux_idx = 0
    for i in range(2, len(ctl)):
      self.ccx(ctl[i], aux[aux_idx], aux[aux_idx+1])
      aux_idx = aux_idx + 1

    # Use predicate to single-control qubit at idx1.
    self.applyc(gate, aux[aux_idx], idx1, desc)

    # Uncompute predicate.
    aux_idx = aux_idx - 1
    for i in range(len(ctl)-1, 1, -1):
      self.ccx(ctl[i], aux[aux_idx], aux[aux_idx+1])
      aux_idx = aux_idx - 1
    self.ccx(ctl[0], ctl[1], aux[0])
```

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

4.3.8 Example

To show an example of how to use the circuit model, here is the classical arithmetic adder circuit using the matrix-based infrastructure:

```
def fulladder_matrix(psi: state.State) -> state.State:
 """Non-quantum-exploiting, classic full adder."""

 psi = ops.Cnot(0, 3)(psi, 0)
 psi = ops.Cnot(1, 3)(psi, 1)
 psi = ops.ControlledU(0, 1, ops.Cnot(1, 4))(psi, 0)
 psi = ops.ControlledU(0, 2, ops.Cnot(2, 4))(psi, 0)
 psi = ops.ControlledU(1, 2, ops.Cnot(2, 4))(psi, 1)
 psi = ops.Cnot(2, 3)(psi, 2)
 return psi
```

And here is the formulation using the quantum circuit. It is considerably more compact. It also hides the implementation details, which means we will be able to accelerate the circuit later (in Sections 4.4 and 4.5) by providing fast implementations of the `apply` functions.

```
def fulladder_qc(qc: circuit.qc) -> None:
 """Non-quantum-exploiting, classic full adder."""

 qc.cx(0, 3)
 qc.cx(1, 3)
 qc.ccx(0, 1, 4)
 qc.ccx(0, 2, 4)
 qc.ccx(1, 2, 4)
 qc.cx(2, 3)
```

As we will see later in Section 8.5, this wrapper class makes it easy to augment the implementations of the gate application functions to add functionality for transpilation of a circuit to other forms, e.g., QASM (Cross et al., 2017), Cirq (Google, 2021c), or Qiskit (Gambetta et al., 2019).

But wait – we have not yet detailed the `apply` functions! They will be the topic of the next sections.

4.4 Fast Gate Application

Up to this point, we have applied a gate by first tensoring it with identity matrices and then applying the resulting large matrix to a full state vector. As described in the introductory notes in Section 4.1, this does not scale well beyond a small number

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

of qubits. For ten qubits, the augmented matrix is already a 1024×1024 matrix, requiring 1024^2 multiplications and additions. Can we devise a more efficient way to apply gates? Yes, we can.

Let's analyze what happens during gate application. To start the analysis, we create a pseudo state vector which is *not* normalized, but allows the visualization of what happens to it as gates are applied to individual qubits.

```
    qc = circuit.qc('test')
    qc.arange(4)
    print(qc.psi)
>>
4-qubit state. Tensor:
[ 0.+0.j  1.+0.j  2.+0.j  3.+0.j  4.+0.j  5.+0.j  6.+0.j  7.+0.j  8.+0.j
  9.+0.j 10.+0.j 11.+0.j 12.+0.j 13.+0.j 14.+0.j 15.+0.j]
```

Now we apply the X-gate to qubits 0 to 3, one by one, always starting with a freshly created vector. The X-gate is interesting in that it multiplies state vector entries by 0 and 1, causing values to swap. This is similar to how applying the X-gate to a regular qubit "flips" $|0\rangle$ and $|1\rangle$.

```
    # Let's try this for qubits 0 to 3.
    for idx in range(4):
      qc = circuit.qc('test')

      # Populate vector with values 0 to 15.
      qc.arange(4)

      # Apply X-gate to qubit at index `idx`.
      qc.x(idx)
      print('Applied X to qubit {}: {}'.format(idx, qc.psi))
```

First we apply the X-gate to qubit 0 to get this result:

```
Applied X to qubit 0: 4-qubit state. Tensor:
[ 8.+0.j  9.+0.j 10.+0.j 11.+0.j 12.+0.j 13.+0.j 14.+0.j 15.+0.j
  0.+0.j  1.+0.j  2.+0.j  3.+0.j  4.+0.j  5.+0.j  6.+0.j  7.+0.j]
```

It appears the right half of the vector was swapped with the left half. Let's try the next qubit index. Applying the X-gate to qubit 1 results in:

```
Applied X to qubit 1: 4-qubit state. Tensor:
[  4.+0.j  5.+0.j  6.+0.j  7.+0.j  0.+0.j  1.+0.j  2.+0.j  3.+0.j
  12.+0.j 13.+0.j 14.+0.j 15.+0.j  8.+0.j  9.+0.j 10.+0.j 11.+0.j]
```

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

Now it appears that chunks of four vector elements are being swapped. The elements 4–7 swap position with elements 0–3, and the elements 12–15 swap position with elements 8–11. A pattern is emerging. Let us apply the X-gate to qubit 2:

```
Applied X to qubit 2: 4-qubit state. Tensor:
[ 2.+0.j  3.+0.j  0.+0.j  1.+0.j  6.+0.j  7.+0.j  4.+0.j  5.+0.j
 10.+0.j 11.+0.j  8.+0.j  9.+0.j 14.+0.j 15.+0.j 12.+0.j 13.+0.j]
```

The pattern continues: now groups of two elements are swapped. And finally, for qubit 3 we see that individual qubits are being swapped:

```
Applied X to qubit 3: 4-qubit state. Tensor:
[ 1.+0.j  0.+0.j  3.+0.j  2.+0.j  5.+0.j  4.+0.j  7.+0.j  6.+0.j  9.+0.j
  8.+0.j 11.+0.j 10.+0.j 13.+0.j 12.+0.j 15.+0.j 14.+0.j]
```

We recognize a clear "power-of-2" pattern. The vector has $2^4 = 16$ elements, corresponding to four qubits. To express numbers from 0–15, we need four classical bits: $b_3 b_2 b_1 b_0$. Remember that we enumerate qubits from left to right and classical bits from right to left. Also, remember that we are using the X-gate, which multiplies by 0s and 1s, leaving the impression of swapping elements.

- **Qubit 0**. Applying the X-gate to qubit 0 swaps the first half of the state vector with the second half.

  If we interpret vector indices in binary, the state elements with indices that had bit 3 set (most significant bit) switched position with the indices that did not have bit 3 set. Positions 8–15 had bit 3 set and switched positions with 0–7, which did not have bit 3 set. One block of eight elements got switched.
- **Qubit 1**. Applying the X-gate to qubit 1 swaps the second quarter of the state vector with the first, and the fourth quarter with the third.

  Correspondingly, the vector elements with indices that had bit 2 set switched with the ones that have bit 2 not set, "bracketed" by the bit pattern in bit 3. What does it mean that an index is bracketed by a higher-order bit? It simply means that the higher-order bit did not change, it remained 0 or 1. Only the lower order bits switch between 0 and 1. Blocks of four elements were switched at a time and there are two brackets for qubit index 1.
- **Qubit 2**. Applying the X-gate to qubit 2 swaps the second eighth of the state vector with the first, the fourth with the third, the sixth with the fifth, and so on.

  Similar to above, the vector elements with indices that had bit 1 set switched with the ones that didn't have bit 1 set. This swapping is bracketed by the bit pattern in bit 2 and further bracketed by the bit patterns of bit 3.
- **Qubit 3**. Finally, and again similar to above, applying the X-gate to qubit 3 now swaps single elements: element 0 swaps with element 1, element 3 swaps with element 2, and so on.

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

We can put this pattern in a closed form by looking at the binary bit pattern for the state vector indices (Smelyanskiy et al., 2016). Let us introduce this *bit index* notation for a state with a classical binary bit representation (where we omit the state brackets $|\cdot\rangle$ for ease of notation):

$$\psi_{\beta_{n-1}\beta_{n-2}\ldots\beta_0}$$

If we expect a specific 0 or 1 at a given bit position k, we specify this bit value in the notation:

$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 0_k\ldots\beta_0}$$
$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 1_k\ldots\beta_0}$$

As an example, the state $|01101\rangle$ can be written as decimal $|13\rangle$ or ψ_{01101} in this notation.

Applying a single-qubit gate on qubit k in an n-qubit state (qubits 0 to $n-1$) applies the gate to a pair of amplitudes whose indices differ in bit $(n-1-k)$ in binary representation. In our first example, we have four qubits. Qubit 0 translates to classical bit 3 in this notation, and qubit 3 corresponds to classical bit 0. We apply the X-gate to the probability amplitudes that correspond to the states where the bit index switches between 0 and 1, thus swapping chunks of the state vector. The swapping happens because we applied the X-gate. Again, the same approach will work for all gates; it just becomes visual and easy to understand for the X-gate.

In general, we want to apply a gate G to a qubit of a system in state $|\psi\rangle$, where G is a 2×2 matrix. Let us name the four matrix elements G_{00}, G_{01}, G_{10}, and G_{11}, corresponding to left-top, right-top, left-bottom, and right-bottom.

Applying a gate G to the kth qubit corresponds to the following recipe. This notation indicates looping over the full state vector. All vector elements whose indices match the specified bit patterns are being multiplied with the gate elements G_{00}, G_{01}, G_{10}, and G_{11}, as specified in this recipe.

$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 0_k\ldots\beta_0} = G_{00}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 0_k\ldots\beta_0} + G_{01}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_k\ldots\beta_0}$$
$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 1_k\ldots\beta_0} = G_{10}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 0_k\ldots\beta_0} + G_{11}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_k\ldots\beta_0}$$

For controlled gates, the pattern can be extended. We have to ensure that the control bit c is set to 1 and only apply the gates to states for which this is the case:

$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 0_k\ldots\beta_0} = G_{00}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 0_k\ldots\beta_0} + G_{01}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 1_k\ldots\beta_0}$$
$$\psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 1_k\ldots\beta_0} = G_{10}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 0_k\ldots\beta_0} + G_{11}\ \psi_{\beta_{n-1}\beta_{n-2}\ldots 1_c\ldots 1_k\ldots\beta_0}$$

In the implementation, we have to be mindful of the qubit ordering. Qubit 0 is the topmost qubit, but for the bit patterns, as is common, classical bit 0 is the least significant bit. So in our implementation, we have to reverse the bit indices.

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

To apply a single gate, we add this function to our implementation of states in file `lib/state.py` (with `1<<n` as an optimized version of `2**n`):

```
def apply(self, gate: ops.Operator, index: int) -> None:
  """Apply single-qubit gate to this state."""

  # To maintain qubit ordering in this infrastructure,
  # index needs to be reversed.
  #
  index = self.nbits - index - 1
  pow_2_index = 1 << index
  g00 = gate[0, 0]
  g01 = gate[0, 1]
  g10 = gate[1, 0]
  g11 = gate[1, 1]
  for g in range(0, 1 << self.nbits, 1 << (index+1)):
    for i in range(g, g + pow_2_index):
      t1 = g00 * self[i] + g01 * self[i + pow_2_index]
      t2 = g10 * self[i] + g11 * self[i + pow_2_index]
      self[i] = t1
      self[i + pow_2_index] = t2
```

The implementation for controlled gates is very similar, but note the additional `if` statement in the code, checking whether or not the control bit is set:

```
def applyc(self, gate: ops.Operator, ctrl: int, target: int) -> None:
  """Apply a controlled 2-qubit gate via explicit indexing."""

  # To maintain qubit ordering in this infrastructure,
  # index needs to be reversed.
  qbit = self.nbits - target - 1
  pow_2_index = 2**qbit
  ctrl = self.nbits - ctrl - 1
  g00 = gate[0, 0]
  g01 = gate[0, 1]
  g10 = gate[1, 0]
  g11 = gate[1, 1]
  for g in range(0, 1 << self.nbits, 1 << (qbit+1)):
    idx_base = g * (1 << self.nbits)
    for i in range(g, g + pow_2_index):
      idx = idx_base + i
      if idx & (1 << ctrl):
        t1 = g00 * self[i] + g01 * self[i + pow_2_index]
        t2 = g10 * self[i] + g11 * self[i + pow_2_index]
        self[i] = t1
        self[i + pow_2_index] = t2
```

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

4.4.1 Benchmarking

The complexity of this method is now $O(n)$, compared to $O(n^2)$ for matrix-vector multiplication. To see how quickly one outperforms the other, we write a quick test. This is not "rocket surgery," but the effects are too pleasing to ignore.

```
def single_gate_complexity() -> None:
  """Compare times for full matmul vs single-gate."""

  nbits = 12
  qubit = random.randint(0, nbits-1)
  gate = ops.PauliX()

  def with_matmul():
    psi = state.zeros(nbits)
    op = ops.Identity(qubit) * gate * ops.Identity(nbits - qubit - 1)
    psi = op(psi)

  def apply_single():
    psi = state.zeros(nbits)
    psi = apply_single_gate(gate, qubit, psi)

  print('Time with full matmul: {:.3f} secs'
        .format(timeit.timeit(with_matmul, number=1)))
  print('Time with single gate: {:.3f} secs'
        .format(timeit.timeit(apply_single, number=1)))
```

Using mildly unscientific methodology, we see a significant performance difference of over 100× for 12 qubits already:

```
Time with full matmul: 0.627 secs
Time with single gate: 0.004 secs
```

We could now add these routines to the quantum circuit class, but wait – we can do even better and accelerate these routines with C++. This will be the topic of the next section.

4.5 Accelerated Gate Application

We now understand how to apply gates to a state vector with linear complexity, but the code was in Python, which is known to execute slower than C++. In order to add a few more qubits to our simulation capabilities and accelerate gate application, we

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

implement the gate application functions in C++ and import them into Python using standard extension techniques.

This section contains a lot of C++ code. The core principles were shown in Section 4.4; there is not much new here, except some fun observations about performance at the end. We still detail this code as it might be of value for readers with no experience extending Python with fast C++. The actual open-source code is about 150 lines long and available in the open-source repository.

The key routines are in a file `xgates.cc` for "accelerated gates". The `<path>` to `numpy` must be set correctly to point to a local setup. The open-source repository will have the latest instructions on how to compile and use this Python extension. We also want to support both `float` and `double` complex numbers, so we templatize the code accordingly.

```
// Make sure this header can be found:
#include <Python.h>

#include <stdio.h>
#include <stdlib.h>
#include <complex>

// Configure the path, likely in the BUILD file:
#include "<path>/numpy/core/include/numpy/ndarraytypes.h"
#include "<path>/numpy/core/include/numpy/ufuncobject.h"
#include "<path>/numpy/core/include/numpy/npy_3kcompat.h"

typedef std::complex<double> cmplxd;
typedef std::complex<float> cmplxf;

// apply1 applies a single gate to a state.
//
// Gates are typically 2x2 matrices, but in this implementation they
// are flattened to a 1x4 array:
//     a  b
//     c  d   ->  a b c d
//
template <typename cmplx_type>
void apply1(cmplx_type *psi, cmplx_type gate[4],
            int nbits, int tgt) {
  tgt = nbits - tgt - 1;
  int q2 = 1 << tgt;
  for (int g = 0; g < 1 << nbits; g += (1 << (tgt+1))) {
    for (int i = g; i < g + q2; ++i) {
      cmplx_type t1 = gate[0] * psi[i] + gate[1] * psi[i + q2];
      cmplx_type t2 = gate[2] * psi[i] + gate[3] * psi[i + q2];
      psi[i] = t1;
      psi[i + q2] = t2;
    }
  }
}
```

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

```
// applyc applies a controlled gate to a state.
template <typename cmplx_type>
void applyc(cmplx_type *psi, cmplx_type gate[4],
            int nbits, int ctl, int tgt) {
   //[... similar to above, but for controlled gates]
}
```

The code above mirrors the Python implementation very closely. To now extend Python and make this extension loadable as a shared module, we add standard Python bindings code for single-qubit gates:

```
template <typename cmplx_type, int npy_type>
void apply1_python(PyObject *param_psi, PyObject *param_gate,
                   int nbits, int tgt) {
  PyObject *psi_arr =
    PyArray_FROM_OTF(param_psi, npy_type, NPY_IN_ARRAY);
  cmplx_type *psi = ((cmplx_type *)PyArray_GETPTR1(psi_arr, 0));

  PyObject *gate_arr =
    PyArray_FROM_OTF(param_gate, npy_type, NPY_IN_ARRAY);
  cmplx_type *gate = ((cmplx_type *)PyArray_GETPTR1(gate_arr, 0));

  apply1<cmplx_type>(psi, gate, nbits, tgt);

  Py_DECREF(psi_arr);
  Py_DECREF(gate_arr);
}

static PyObject *apply1_c(PyObject *dummy, PyObject *args) {
  PyObject *param_psi = NULL;
  PyObject *param_gate = NULL;
  int nbits;
  int tgt;
  int bit_width;

  if (!PyArg_ParseTuple(args, "OOiii", &param_psi, &param_gate,
                        &nbits, &tgt, &bit_width))
    return NULL;
  if (bit_width == 128) {
    apply1_python<cmplxd, NPY_CDOUBLE>(param_psi,
                                       param_gate, nbits, tgt);
  } else {
    apply1_python<cmplxf, NPY_CFLOAT>(param_psi,
                                      param_gate, nbits, tgt);
  }
  Py_RETURN_NONE;
}
```

There is, of course, similar code for the controlled gates in the open-source repository. The following are the functions the Python interpreter will call when importing

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

a module. We register the Python wrappers in a module named `xgates` with standard boilerplate code:

```
// Python boilerplate to expose above wrappers to programs.
static PyMethodDef xgates_methods[] = {
    {"apply1", apply1_c, METH_VARARGS,
     "Apply single-qubit gate, complex double"},
    {"applyc", applyc_c, METH_VARARGS,
     "Apply controlled qubit gate, complex double"},
    {NULL, NULL, 0, NULL}};

static struct PyModuleDef xgates_definition = {
  PyModuleDef_HEAD_INIT,
  "xgates",
  "Python extension to accelerate quantum simulation math",
  -1,
  xgates_methods
};

PyMODINIT_FUNC PyInit_xgates(void) {
  Py_Initialize();
  import_array();
  return PyModule_Create(&xgates_definition);
}
```

In order for Python to be able to find this extension, we typically set an environment variable. For example, on Linux:

```
    export PYTHONPATH=path_to_xgates.so
```

Alternatively, you can extend Python's module search path programmatically with code like this:

```
    import sys
    sys.path.append('/path/to/search')
```

4.5.1 Circuits Finally Finalized

With our accelerated implementation, we can finally finish the apply function in the quantum circuit `qc` class. Both single-qubit gates and controlled gates can be applied to qubits, but, for convenience, single-qubit gates can also be applied to whole registers:

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

```
import xgates

def apply1(self, gate: ops.Operator, idx: int,
           name: str = None, *, val: float = None):
  if isinstance(idx, state.Reg):
    for reg in range(idx.nbits):
        xgates.apply1(self.psi, gate.reshape(4), self.psi.nbits,
                      idx[reg], tensor.tensor_width)
    return
  xgates.apply1(self.psi, gate.reshape(4), self.psi.nbits, idx,
                tensor.tensor_width)

def applyc(self, gate: ops.Operator, ctl: int, idx: int,
           name: str = None, *, val: float = None):
  if isinstance(idx, state.Reg):
    raise AssertionError('controlled register not supported')
  xgates.applyc(self.psi, gate.reshape(4), self.psi.nbits, ctl,
                idx, tensor.tensor_width)
```

4.5.2 Premature Optimization, First Act

Looking at the standard gates, we find a lot of 0s and 1s, which means that several gate applications should run faster if we optimized for these special cases. Emphasis on *should*. Let us run an experiment to verify this assumption.

We construct a benchmark to compare the general gate application routines with ones that are specialized for the X-gate, which has two 0s and two 1s. This should enable us to save a total of four multiplies, two additions, and perhaps some memory accesses per single qubit. In other words, for this original inner loop:

```
  for (int i = g; i < g + q2; ++i) {
    cmplx t1 = gate[0][0] * psi[i] + gate[0][1] * psi[i + q2];
    cmplx t2 = gate[1][0] * psi[i] + gate[1][1] * psi[i + q2];
    psi[i] = t1;
    psi[i + q2] = t2;
  }
```

This is a modified and seemingly faster variant of the inner loop, which avoids at least four multiplications:

```
  for (int i = g; i < g + q2; ++i) {
    cmplx t1 = psi[i + q2];
    cmplx t2 = psi[i];
    psi[i] = t1;
    psi[i + q2] = t2;
  }
```

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

In the code below, we only show the implementation of the noncontrolled gates, but we benchmark both single gates and controlled gates. At the time of writing, the specific benchmarking infrastructure `BENCHMARK_BM` was not open-sourced, but there are countless other ways available in open-source for this type of benchmarking.

```
typedef std::complex<double> cmplx;
static const int nbits = 22;
static cmplx* psi;

void apply_single(cmplx* psi, cmplx gate[2][2], int nbits, int qubit) {
  int q2 = 1 << qubit;
  for (int g = 0; g < 1 << nbits; g += 1 << (qubit+1)) {
    for (int i = g; i < g + q2; ++i) {
      cmplx t1 = gate[0][0] * psi[i] + gate[0][1] * psi[i + q2];
      cmplx t2 = gate[1][0] * psi[i] + gate[1][1] * psi[i + q2];
      psi[i] = t1;
      psi[i + q2] = t2;
    }
  }
}

void apply_ctl(cmplx* psi, cmplx gate[2][2], int nbits,
               int ctl, int tgt) {
   [...]
}

// --- Benchmark full gates ---
void BM_apply_single(benchmark::State& state) { [...] }
BENCHMARK(BM_apply_single);

void BM_apply_controlled(benchmark::State& state) { [...] }
BENCHMARK(BM_apply_controlled);

// --- "Optimized Gates" ---
void apply_single_opt(cmplx* psi, int nbits, int qubit) {
  int q2 = 1 << qubit;
  for (int g = 0; g < 1 << nbits; g += 1 << (qubit + 1)) {
    for (int i = g; i < g + q2; ++i) {
      cmplx t1 = psi[i + q2];
      cmplx t2 = psi[i];
      psi[i] = t1;
      psi[i + q2] = t2;
    }
  }
}

void apply_ctl_opt(cmplx* psi, int nbits, int ctl, int tgt) {
   [... similar to apply_single_opt, but controlled version]
}
```

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

Table 4.1 Benchmark results (program output), comparing hand-optimized and nonoptimized gate application routines.

Benchmark	Time(ns)	CPU(ns)	Iterations
BM_apply_single	116403527	116413785	24
BM_apply_single_opt	132820169	132829412	21
BM_apply_controlled	81595871	81600200	34
BM_apply_controlled_opt	89064964	89072559	31

```
// --- Benchmark optimized gates ---
void BM_apply_single_opt(benchmark::State& state) { [...] }
BENCHMARK(BM_apply_single_opt);
void BM_apply_controlled_opt(benchmark::State& state) { [...] }
BENCHMARK(BM_apply_controlled_opt);
```

The performance results are shown in Table 4.1. Remember our hypothesis that the optimized version would be faster, because it executes fewer multiplications and additions. Column `Iterations` shows iterations per second; higher is better.

The specialized version runs about 10% *slower*! For the given x86 platform, the compiler was able to vectorize the nonspecialized version, leading to a slightly higher overall throughput. Intuition is good, validation is better.

In summary, we found a way to apply gates with linear complexity and accelerated it by another significant factor with C++. Performance comparison to the Python version shows a speedup of about 100×. This should add six or seven additional qubits to our simulation capabilities. This infrastructure is sufficient for all remaining algorithms in this book.

There are other ways to simulate quantum computing (Altman et al., 2021), as we discussed at the end of Section 4.1. There is one specific, interesting way to represent states *sparsely*. For many circuits this is a very efficient data structure. We give a brief overview of it in Section 4.6 and provide full implementation details in the Appendix.

4.6 Sparse Representation

So far, our data structure for representing quantum states is a dense array holding all probability amplitudes for all superimposed states, where the amplitude for a specific state can be found via binary indexing. However, for many circuits and algorithms, there can be a high percentage of states with close to zero probability. Storing these 0-states and applying gates to them will have no effect and is wasteful. This fact can be exploited with a sparse representation. An excellent reference implementation of this principle can be found in the venerable, open source `libquantum` library (Butscher and Weimer, 2013).

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

We re-implement the core ideas of that library as they pertain to this book; `libquantum` addresses other aspects of quantum information, which we do not cover. We therefore name our implementation `libq` to distinguish it from the original. The original library is in plain C, but our implementation was moderately updated with C++ for improved readability and performance. We maintain some of the C naming conventions for key variables and functions to help with direct comparisons.

Here is the core idea: assume we have a state of N qubits, all initialized to be in the state $|0\rangle$. The dense representation stores 2^N complex numbers, where only the very first entry is a 1.0 and all other values are 0.0, corresponding to state $|00\ldots0\rangle$.

Our library `libq` turns this on its head. States are stored as bitmasks (currently up to 64 qubits, but this can be extended), where 0s and 1s correspond to states $|0\rangle$ and $|1\rangle$. Each of these bit combinations is then paired with a probability amplitude. In the above example, `libq` would store the tuple `(0x00...0, 1.0)`, indicating that the only state with nonzero probability is $|00\ldots0\rangle$. For 53 qubits, the full state representation would require 72 petabytes of memory, while the sparse representation only requires a total of 16 bytes if the amplitude is stored as a double precision value, 12 bytes if we use 4-byte floats.

Applying a Hadamard gate to the least significant qubit will put it in superposition. In `libq`, this means that there are now two states with nonzero probability:

- $|000\ldots00\rangle$ with probability 50%.
- $|000\ldots01\rangle$ with probability 50%.

Correspondingly, `libq` stores only two tuples, each with a probability amplitude of $1/\sqrt{2}$, using 32 bytes (or 24 bytes for 4-byte floats).

While a circuit is running, superposition is generated and destroyed. States become probable and no longer probable. A key aspect of `libquantum` is that gates are recognized as producing or destroying superposition and handled accordingly. Furthermore, it filters out all states with close to 0.0 probability after application of superpositioning gates. This reduces the number of stored tuples and accelerates future gate applications.

Gate application itself becomes very fast. For example, assume we need to apply the X-gate to the least significant qubit. In the dense case, the whole state vector needs to be traversed and modified, as outlined in Section 4.5 on accelerated gate application. In the `libq` case, only a bit-flip is needed. In the example above, assuming an initial state of all $|0\rangle$, applying the X-gate to the least significant qubit means we only have to flip the least significant bit in the bitmask; the tuple `(0x00...00, 1.0)` becomes `(0x00...01, 1.0)`. This is dramatically faster than having to traverse and modify a potentially very large state vector, especially if the number of nonzero probability states is low.

To maintain the state tuples, we need to support two main operations:

- Iterate over all available state tuples.
- Find or create a specific state tuple.

`libquantum` implements a hash table to manage the tuples, and, as we will later see, despite the favorable performance characteristics of hash tables, it ultimately

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

remains the performance bottleneck in the implementation. Our `libq` moderately improves this data structure.

The implementation is about 500 lines of C++ code. A detailed, annotated description, which also includes optimization wins and fails, can be found in the Appendix.

There are also downsides to this design, which may prevent it from scaling to very large numbers of qubits or circuits with a high percentage of nonzero probabilities. Individual states are encoded efficiently as tuples of a bitmask to encode a state and probability amplitude. But there are additional data structures, such as the hash table to maintain existing states. The memory requirement per state is higher than in a full-state representation. This means that there is a crossover point where the sparse representation becomes *less* efficient than the full-state representation. In particular, it does not appear to do well on the quantum random algorithms that we discuss in Section 5.2.

Another problem might arise from the way the hash table is used to store the states. At some size threshold, the hash table's random memory accesses will be outperformed by linear memory accesses, which benefit from caches and can be prefetched effectively. Furthermore, in a distributed computing environment, hash table entries might be distributed unpredictably across machines. Gate application might thus incur prohibitively high communication costs.

4.6.1 Benchmarking

We only provide anecdotal evidence for the efficiency of the sparse representation. A full performance evaluation is ill-advised in a book like this – the results will be out of date and no longer relevant by the time you read this.

The most complex algorithm in this book is Shor's integer factorization algorithm (Section 4.6). The quantum part of the algorithm is called order finding. To factor the number 15, it requires 18 qubits and 10,533 gates; to factor 21, it requires 22 qubits and 20,671 gates, and to factor 35, it requires 26 qubits and 36,373 gates. We run this circuit in two different ways:

- Run it as is, using the accelerated quantum circuit implementation.
- Transpile the circuit directly to `libq` without executing it. We describe transpilation in Section 8.5. The output is a C++ source file, which is compiled and linked with the `libq` library to produce an executable.

Both versions will compute the same result; the textual output only differs marginally. To factor 21 with 22 qubits, we get the following result. Note that a maximum of only 1.6% of all possible states ever obtain a nonzero probability at one point or the other during execution:

```
# of qubits        : 22
# of hash computes : 2736
Maximum # of states: 65536, theoretical: 4194304, 1.562%
States with nonzero probability:
 0.499966 +0.000000i|4> (2.499658e-01) (|00 0000 0000 0000 0000 0100>)
```

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press

```
 0.000001 -0.000000i|32772> (6.148556e-13) (|00 0000 1000 0000 0000
 ↪  0100>)
-0.499970 +0.000000i|65536> (2.499696e-01) (|00 0001 0000 0000 0000
↪  0000>)
 0.499966 +0.000000i|65540> (2.499658e-01) (|00 0001 0000 0000 0000
 ↪  0100>)
 0.000001 -0.000000i|98308> (6.148556e-13) (|00 0001 1000 0000 0000
 ↪  0100>)
 0.499970 -0.000000i|0> (2.499696e-01) (|00 0000 0000 0000 0000 0000>)
real    0m4.225s
```

The `libq` version runs in under five seconds on a modern workstation, while the circuit version takes about 2.5 minutes, a speedup of roughly 25×. To factor the number 35 with 26 qubits, the `libq` version runs for about 3 minutes, while the full state simulation takes about an hour. Again, a solid speedup, this time of about 20×. We do ignore the compile times for the generated C++ versions. We would have to include these in an actual scientific evaluation, which this is not.

https://doi.org/10.1017/9781009099974.006 Published online by Cambridge University Press