O'REILLY®

Программирование квантовых компьютеров

Базовые алгоритмы и примеры кода

Мерседес Химено-Сеговиа,
Ник Хэрриган, Эрик Джонстон

Programming Quantum Computers

Essential Algorithms and Code Samples

Eric R. Johnston, Nic Harrigan, and Mercedes Gimeno-Segovia

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Программирование квантовых компьютеров

Базовые алгоритмы и примеры кода

Мерседес Химено-Сеговиа,
Ник Хэрриган, Эрик Джонстон

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2021

ББК 32.973.2-018+22.31
УДК 004.4:530.145
Х46

Химено-Сеговиа Мерседес, Хэрриган Ник, Джонстон Эрик

Х46 Программирование квантовых компьютеров. Базовые алгоритмы и примеры кода. — СПб.: Питер, 2021. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1531-0

Квантовые компьютеры спровоцировали новую компьютерную революцию, и у вас есть прекрасный шанс присоединиться к технологическому прорыву прямо сейчас. Разработчики, специалисты по компьютерной графике и начинающие айтишники найдут в этой книге практическую информацию по квантовым вычислениям, нужную программистам. Вместо штудирования теории и формул вы сразу займетесь конкретными задачами, демонстрирующими уникальные возможности квантовой технологии.

Эрик Джонстон, Ник Хэрриган и Мерседес Химено-Сеговиа помогают развить необходимые навыки и интуицию, а также освоить инструментарий, необходимый для создания квантовых приложений. Вы поймете, на что способны квантовые компьютеры и как это применить в реальной жизни.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018+22.31
УДК 004.4:530.145

ISBN 978-1492039686 англ.

Authorized Russian translation of the English edition of Programming Quantum Computers
ISBN 9781492039686 © 2019 Eric R. Johnston, Nic Harrigan and Mercedes Gimeno-Segovia
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1531-0

Краткое содержание

Предисловие **15**

Глава 1. Введение 19

ЧАСТЬ I.
ПРОГРАММИРОВАНИЕ ДЛЯ QPU

Глава 2. Один кубит 32

Глава 3. Группы кубитов 58

Глава 4. Квантовая телепортация 88

ЧАСТЬ II.
ПРИМИТИВЫ QPU

Глава 5. Квантовая арифметика и логика 106

Глава 6. Усиление комплексной амплитуды 129

Глава 7. QFT: квантовое преобразование Фурье 147

Глава 8. Квантовая оценка фазы 177

ЧАСТЬ III.
ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ QPU

Глава 9. Реальные данные 194

Глава 10. Квантовый поиск 215

Глава 11. Квантовая избыточная выборка 236

Глава 12. Алгоритм Шора 260

Глава 13. Квантовое машинное обучение 284

ЧАСТЬ IV.
ПЕРСПЕКТИВЫ

Глава 14. Обзор литературы 318

Оглавление

От издательства **14**

Предисловие **15**

Структура книги 15

Типографские соглашения 17

Благодарности 17

Глава 1. Введение **19**

Необходимая подготовка 19

Что такое QPU? 21

Практический подход 22

Учебник QCEngine 22

Отладка 24

Низкоуровневые команды QPU 25

Ограничения моделирования 28

Аппаратные ограничения 28

QPU и GPU: некоторые общие характеристики 29

ЧАСТЬ I.
ПРОГРАММИРОВАНИЕ ДЛЯ QPU

Глава 2. Один кубит **32**

Краткий обзор физических кубитов 34

Круговая запись 37

Размер круга 38

Поворот 39

Первые операции QPU 41

Команда QPU: NOT 41

Команда QPU: HAD 42
Команда QPU: READ 43
Команда QPU: WRITE 43
Практический пример: идеально случайный бит 45
Пример кода 46
Пример кода 48
Команды QPU: PHASE(θ) 48
Команды QPU: ROTX(θ) и ROTY(θ) 49
COPY: недостающая операция 50
Объединение операций QPU 50
Команда QPU: ROOT-NOT 51
Пример кода 52
Практический пример: квантовая проверка защиты 53
Пример кода 54
Итоги 57

Глава 3. Группы кубитов 58

Круговая запись для многокубитных регистров 58
Пример кода 60
Изображение многокубитного регистра 61
Однокубитные операции в многокубитных регистрах 62
Чтение кубита в многокубитном регистре 64
Наглядное представление большего количества кубитов 65
Команды QPU: CNOT 67
Практический пример: использование пар Белла для реализации совместной случайности 70
Пример кода 71
Команды QPU: CPHASE и CZ 71
Приемы программирования QPU: фазовый откат 73
Пример кода 75
Команда QPU: CCNOT (вентиль Тоффоли) 75
Команды QPU: SWAP и CSWAP 76
Проверка обменом 77
Пример кода 78
Построение произвольной условной операции 81

Пример кода 82
Практический пример: дистанционно управляемая случайность 84
Пример кода 84
Итоги 87

Глава 4. Квантовая телепортация 88
Практический пример: первые шаги в телепортации 88
Пример кода 90
Анализ программы 94
Шаг 1: создание запутанной пары 95
Шаг 2: подготовка данных 95
Шаг 3.1: связывание данных с запутанной парой 96
Шаг 3.2: перевод кубита данных в суперпозицию 97
Шаг 3.3: чтение обоих кубитов Алисы 97
Шаг 4: получение и преобразование 98
Шаг 5: проверка результата 99
Интерпретация результатов 101
Как используется телепортация? 102
Известные проблемы при телепортации 102

ЧАСТЬ II.
ПРИМИТИВЫ QPU

Глава 5. Квантовая арифметика и логика 106
Странно и необычно 106
Арифметика с QPU 108
Практический пример: построение операторов инкремента и декремента 109
Пример кода 110
Сложение двух квантовых целых чисел 112
Пример кода 113
Отрицательные числа 114
Практический пример: более сложные вычисления 115
Пример кода 116
Переход на квантовый уровень 116
Квантовое условное выполнение 116
Пример кода 117

Фазовое кодирование результата ... 118
Пример кода ... 118
Обратимость и служебные кубиты ... 119
Отмена вычислений ... 122
Отображение булевой логики на операции QPU ... 125
Базовая квантовая логика ... 126
Пример кода ... 127
Итоги ... 128

Глава 6. Усиление комплексной амплитуды ... 129
Практический пример: преобразование между фазой и амплитудой ... 129
Пример кода ... 130
Итерация усиления комплексной амплитуды ... 132
Больше итераций? ... 133
Пример кода ... 134
Инвертирование нескольких элементов ... 136
Пример кода ... 137
Использование усиления комплексной амплитуды ... 142
AA и QFT при оценке суммы ... 142
Ускорение традиционных алгоритмов с применением AA ... 143
Внутри QPU ... 143
Интуитивное понимание ... 143
Итоги ... 146

Глава 7. QFT: квантовое преобразование Фурье ... 147
Скрытые закономерности ... 147
Пример кода ... 148
QFT, DFT и FFT ... 149
Частоты в регистре QPU ... 150
Пример кода ... 151
Пример кода ... 154
DFT ... 154
Вещественные и комплексные входные данные для DFT ... 156
Подробнее о DFT ... 158
Пример кода ... 160
Практическое применение QFT ... 163

Скорость QFT ... 163
Пример кода ... 167
Пример кода ... 167
Пример кода ... 168
Внутри QPU ... 169
Интуитивное объяснение ... 170
Последовательность операций ... 172
Пример кода ... 173
Итоги ... 176

Глава 8. Квантовая оценка фазы ... 177
Получение информации об операциях QPU ... 177
Собственные фазы предоставляют полезную информацию ... 178
Что делает оценка фазы ... 180
Как пользоваться оценкой фазы ... 180
Ввод ... 182
Пример кода ... 182
Вывод ... 184
Предостережения ... 184
Выбор размера выходного регистра ... 185
Сложность ... 186
Условные операции ... 186
Оценка фазы на практике ... 186
Внутри QPU ... 187
Пример кода ... 187
Интуитивное объяснение ... 188
Операция за операцией ... 190
Итоги ... 192

ЧАСТЬ III.
ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ QPU

Глава 9. Реальные данные ... 194
Нецелые данные ... 195
QRAM ... 196

Пример кода ... 198
Пример кода ... 200
Кодирование векторов ... 201
Пример кода ... 203
Ограничения комплексного амплитудного кодирования ... 204
Комплексное амплитудное кодирование и круговая запись ... 206
Кодирование матриц ... 207
Как матрицы могут представляться операциями QPU? ... 208
Квантовое моделирование ... 209

Глава 10. Квантовый поиск ... 215

Фазовая логика ... 216
Построение элементарных операций фазовой логики ... 218
Построение сложных команд фазовой логики ... 219
Пример кода ... 221
Решение логических головоломок ... 222
О котятах и тиграх ... 223
Пример кода ... 226
Общий рецепт для решения задач выполнимости булевых формул ... 227
Практический пример: задача выполнимости 3-SAT ... 229
Пример кода ... 229
Практический пример: невыполнимая задача 3-SAT ... 231
Пример кода ... 232
Ускорение традиционных алгоритмов ... 234

Глава 11. Квантовая избыточная выборка ... 236

Применение QPU в компьютерной графике ... 236
Традиционная избыточная выборка ... 237
Практический пример: вычисление изображений с фазовым кодированием ... 239
Пиксельный шейдер ... 240
Использование операции PHASE для рисования ... 241
Пример кода ... 242
Рисование кривых ... 243
Пример кода ... 244

Выборка в изображениях с фазовым кодированием ... 245
Пример кода ... 247
Более интересное изображение ... 247
Избыточная выборка ... 248
Пример кода ... 250
QSS и традиционная выборка методом Монте-Карло ... 251
Как работает QSS ... 252
Пример кода ... 255
Добавление цветов ... 257
Итоги ... 259

Глава 12. Алгоритм Шора ... 260
Практический пример: алгоритм Шора на QPU ... 261
Пример кода ... 262
Что делает алгоритм Шора ... 263
Для чего вообще нужен QPU? ... 264
Пример кода ... 264
Квантовое решение ... 266
Шаг за шагом: разложение числа 15 на простые множители ... 268
Пример кода ... 268
Шаг 1: инициализация регистров QPU ... 269
Шаг 2: перевод в квантовую суперпозицию ... 270
Шаг 3: условное умножение на 2 ... 272
Шаг 4: условное умножение на 4 ... 274
Шаг 5: квантовое преобразование Фурье ... 276
Шаг 6: чтение квантового результата ... 276
Шаг 7: цифровая логика ... 278
Пример кода ... 279
Шаг 8: проверка результата ... 281
Важные подробности ... 281
Вычисление остатка ... 281
Время и память ... 283
Другие значения переменной coprimes ... 283

Глава 13. Квантовое машинное обучение 284
Решение систем линейных уравнений 285
Описание и решение систем линейных уравнений 286
Решение линейных уравнений на QPU 288
Квантовый анализ главных компонент 298
Традиционный анализ главных компонент 299
PCA с QPU 301
Квантовый метод опорных векторов 305
Традиционный метод опорных векторов 306
SVM с QPU 310
Другие применения машинного обучения 315

ЧАСТЬ IV. ПЕРСПЕКТИВЫ

Глава 14. Обзор литературы 318
От круговой записи к комплексным векторам 318
Некоторые нюансы и примечания по терминологии 321
Измерительный базис 322
Разложение и компиляция вентилей 324
Телепортация вентилей 326
Зал славы QPU 326
Гонка между квантовыми и традиционными компьютерами 327
Алгоритмы на базе оракулов 328
Алгоритм Дойча—Джозы 329
Задача Бернштейна—Вазирани 329
Задача Саймона 330
Языки квантового программирования 330
Перспективы квантового моделирования 332
Исправление ошибок и устройства NISQ 332
Что дальше? 333
Книги 333
Конспекты лекций 334
Сетевые источники информации 334

От издательства

Актуальный список исправлений по книге вы можете найти по адресу: https://github.com/oreilly-qc/oreilly-qc.github.io/blob/master/errata/errata.md.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Предисловие

Квантовые компьютеры перестали быть чисто теоретическими устройствами.

Авторы этой книги полагают, что лучшее применение новым технологиям не всегда находят их изобретатели — чаще это делают эксперты в предметной области, экспериментирующие с технологией как с новым инструментом для своей работы. С учетом сказанного эта книга создавалась как практическое руководство по использованию технологии квантовых вычислений для программистов. В дальнейших главах вы освоите условные обозначения и операции вроде тех, что изображены на рис. П.1, и научитесь применять их в тех задачах, которые вас интересуют.

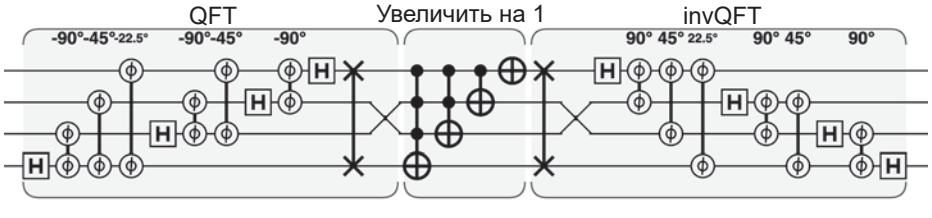


Рис. П.1. Квантовая программа немного похожа на нотную запись

Структура книги

Проверенный и надежный подход для практического освоения новых парадигм программирования основан на изучении набора концептуальных примитивов. Например, каждый программист, изучающий программирование для графических процессоров (GPU), должен сначала сосредоточиться на концепции параллелизма, а не на синтаксисе или специфике оборудования.

Главная задача этой книги заключается в формировании интуитивного понимания набора квантовых примитивов — концепций, формирующих инструментарий структурных элементов для решения задач с использованием QPU. Чтобы подготовить вас к этим примитивам, мы сначала представим основные концепции кубитов (*правила игры*, если хотите). Затем после краткого описания примитивов квантовых процессоров (QPU) мы покажем, как они могут использоваться в качестве структурных элементов в приложениях для QPU.

Книга разделена на три части. Мы рекомендуем читателю сначала ознакомиться с частью I и получить некоторый практический опыт, прежде чем переходить к более сложному материалу.

Часть I. Программирование для QPU

В этой части представлены основные концепции, необходимые для программирования QPU: кубиты, важнейшие команды, суперпозиция и даже квантовая телепортация. Приведенные примеры можно легко запустить в системе моделирования или на физическом QPU.

Часть II. Примитивы QPU

Во второй части книги приводятся описания некоторых алгоритмов и полезных приемов на более высоком уровне. В частности, здесь рассматривается усиление комплексной амплитуды, квантовое преобразование Фурье и оценка фазы. Их можно считать своего рода «библиотечными функциями», которые используются программистами для построения приложений. Без понимания того, как они работают, невозможно стать квалифицированным программистом QPU. Активное сообщество исследователей работает над разработкой новых примитивов QPU, поэтому следует ожидать, что библиотека будет расширяться в будущем.

Часть III. Практическое применение QPU

Мир практических применений QPU — объединяющих примитивы из части II для выполнения полезных реальных задач — развивается так же стремительно, как и сами QPU. В этой части представлены примеры существующих практических применений.

Надеемся, что к концу книги читатель будет понимать, на что способны квантовые приложения, почему они обладают столь мощными возможностями и как распознать типы задач, которые могут решаться с их помощью.

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсив

Используется для обозначения новых терминов.

`Моноширинный шрифт`

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

Так выделяются советы и предложения.

Так обозначаются советы, предложения и примечания общего характера.

Так обозначаются предупреждения и предостережения.

Благодарности

Эта книга не могла бы выйти без поддержки команды талантливых людей — энтузиастов квантовых вычислений. Авторы хотят поблагодарить Мишель, Майка, Ким, Ребекку, Криса и техническую группу издательства O'Reilly, которые разделили наш энтузиазм и усилили его. Хотя все ошибки и упущения лежат на совести авторов, книга сильно выиграла от невероятно полезной обратной связи и вдохновения от многих научных редакторов, среди которых были Конрад Килинг (Konrad Kieling), Кристиан Соммереггер (Christian Sommeregger), Миншен Инь (Mingsheng Ying), Рич Джонстон (Rich Johnston), Джеймс Уивер (James Weaver), Майк Шапиро (Mike Shapiro), Уайатт Берлиник (Wyatt Berlinic) и Айзек Ким (Isaac Kim).

И-Джей хочет поблагодарить Сью, свою музу. Для него квантовые вычисления начали складываться в осмысленную картину в ту неделю, когда они встретились. И-Джей также передает свою благодарность своим друзьям из Бристольского университета и из SolidAngle, которые уговорили его выйти за рамки традиционных представлений.

Ник также благодарит Дерека Хэрригана (Derek Harrigan), когда-то научившего его «говорить на двоичном языке», и остальных Хэрриганов за их любовь и поддержку, а также Шеннон Бернс (Shannon Burns) за желание присоединиться к кругу Хэрриганов.

Мерседес благодарит Хосе Марию Химено Блей (José María Gimeno Blay), который разжег ее ранний интерес к компьютерам, и Мехди Ахмади (Mehdi Ahmadi) — постоянного источника поддержки и вдохновения.

Но как бы банально это ни прозвучало, в первую очередь авторы хотят поблагодарить вас — читателей — за любознательность, которая заставила вас взять эту книгу и узнать что-то новое.

1 Введение

Кем бы вы ни были — экспертом в области разработки ПО, компьютерной графики, обработки данных или просто любознательным гиком, — в этой книге на конкретных практических примерах мы постараемся показать, почему тема квантовых вычислений может быть актуальной для вас.

В последующих главах *нет* подробных объяснений квантовой физики (законов, лежащих в основе квантовых вычислений) или даже квантовой теории информации (возможностей обработки информации, определяемых этими законами). Вместо этого мы приводим работоспособные примеры, формирующие представление о возможностях этой замечательной новой технологии. Что еще важнее, код, представленный в этих примерах, можно будет изменять и адаптировать. Это позволит вам учиться самым эффективным способом из всех возможных: накоплением практического опыта. Попутно базовые концепции поясняются по мере их использования, и только в той степени, в которой они формируют интуитивную основу для написания квантовых программ.

Авторы скромно надеются, что заинтересованные читатели смогут взять на вооружение эти принципы и расширить потенциальные применения квантовых вычислений в областях, о которых некоторые физики даже не слышали. Откровенно говоря, попытки поучаствовать в разжигании квантовой революции — не такая уж *скромная* задача, но быть среди первопроходцев безусловно интересно.

Необходимая подготовка

Физическая теория, лежащая в основе квантовых вычислений, насыщена сложной математикой. Впрочем, то же самое можно сказать и о физической

теории работы транзисторов, однако изучение C++ обходится без единого физического уравнения. В этой книге выбран похожий подход, ориентированный на программистов и не требующий сколько-нибудь значительной математической подготовки. Далее приведен краткий список знаний, которые могут пригодиться для освоения представленных концепций.

- Основные управляющие структуры программирования (`if`, `while` и т. д.). Для создания простых примеров, которые могут запускаться в интернете, используется язык JavaScript. Если вы не владеете JavaScript, но у вас есть предшествующий опыт программирования, необходимый уровень подготовки можно набрать менее чем за час. Более подробную информацию о JavaScript можно найти в книге «Learning JavaScript» Тодда Брауна (Todd Brown), опубликованной издательством O'Reilly.
- Некоторые математические концепции уровня программирования, прежде всего:
  - понимание использования математических функций;
  - знание тригонометрических функций;
  - умение работать с двоичными числами и выполнять преобразования между двоичным и десятичным представлением чисел;
  - понимание смысла комплексных чисел.
- Элементарное понимание принципов оценки вычислительной сложности алгоритмов (обозначение «O-большое»).

Одной из частей книги, выходящей за рамки этих требований, станет глава 13, в которой будет приведен обзор ряда применений квантовых вычислений в области машинного обучения. Из-за нехватки места в обзоре приводятся очень краткие вводные описания каждого примера машинного обучения, после которых мы показываем, чем в каждом случае вам поможет квантовый компьютер. И хотя мы постарались сделать материал понятным для массового читателя, тем, кто захочет поэкспериментировать с этими примерами, пригодится некоторая подготовка в области машинного обучения.

Книга посвящена программированию квантовых компьютеров (а не их построению или теоретическим обоснованиям), что и позволяет нам обойтись без расширенной математики и квантовой теории. Однако для читателей, пожелавших обратиться к более академической литературе по теме, глава 14 предоставляет полезные источники информации и связывает представленные концепции с математическими принципами, используемыми в научном сообществе квантовых вычислений.

Что такое QPU?

Несмотря на повсеместное использование, термин «квантовый компьютер» может ввести в заблуждение. Он вызывает в воображении образ совершенно нового, чуть ли не инопланетного устройства, заменяющего все существующие программные продукты футуристическими альтернативами.

На момент написания этой книги существовало распространенное, хотя и очень серьезное ошибочное представление. Перспективы квантовых компьютеров связаны не с тем, что они станут *«убийцами» традиционных компьютеров*, а, скорее, с их способностью радикально расширить набор задач, решаемых вычислительными средствами. Существуют важные вычислительные задачи, легко решаемые на квантовом компьютере, но которые буквально невозможно решить на любом гипотетическом вычислительном устройстве, которые мы когда-либо могли надеяться построить[1].

Но принципиально то, что подобное ускорение наблюдается только для определенных задач (многие из которых будут рассмотрены более подробно). И хотя ожидается, что со временем будут обнаружены новые типы таких задач, крайне маловероятно, что *все* вычисления будет разумно проводить на квантовых компьютерах. При решении большинства задач, на которые расходуются такты процессора вашего ноутбука, квантовый компьютер будет работать не лучше обычного.

Другими словами — с точки зрения программиста — квантовый компьютер в действительности является сопроцессором. В прошлом компьютеры использовали разные виды сопроцессоров, каждый из которых хорошо подходил для своей специализации: арифметических операций с плавающей точкой, обработки сигналов или графического моделирования в реальном времени. С учетом сказанного мы будем использовать термин *QPU* (*Quantum Processor Unit*, *квантовый процессор*) для обозначения устройства, на котором выполняются наши примеры кода. Мы считаем, что это подчеркивает важный контекст, в котором должны представляться квантовые вычисления.

---

[1] В целях демонстрации этого утверждения мы любим приводить следующий неформальный пример. Допустим, традиционные транзисторы можно было бы уменьшить до размера атома, и вы хотите построить традиционный компьютер размером с промышленный склад, способный сравниться с производительностью даже умеренного квантового компьютера по части разложения на простые множества. Тогда транзисторы пришлось бы упаковать настолько плотно, что это привело бы к образованию гравитационной сингулярности. Конечно, условия гравитационной сингулярности сильно затрудняют вычисления (и само существование).

Как и в случае с другими сопроцессорами — например, графическими процессорами (GPU), — программирование для QPU подразумевает написание кода, который будет в основном выполняться на центральном процессоре (CPU) нормального компьютера. Центральный процессор выдает команды QPU-сопроцессора только для инициирования задач, входящих в область его специализации.

Практический подход

Практические примеры образуют основу этой книги. Но на момент ее написания полноценный QPU общего назначения еще не существует — тогда как же вы будете запускать приведенный код? К счастью, даже во время написания книги существовали прототипы QPU, доступные в облаке. Кроме того, для меньших задач поведение QPU можно моделировать на традиционном вычислительном оборудовании. И хотя моделирование больших QPU-программ становится невозможным, для малых фрагментов кода моделирование помогает освоить управление настоящими QPU. Примеры кода в этой книге совместимы с обеими ситуациями, и они останутся полезными и поучительными даже с появлением более совершенных QPU.

Существует много разных библиотек, систем и средств моделирования QPU. Список ссылок на некоторые системы с хорошей поддержкой доступен по адресу http://oreilly-qc.github.io. На этой странице мы приводим примеры кода из книги там, где это возможно, на разных языках. Но чтобы примеры кода не забивали текст, мы приводим примеры только на JavaScript для QCEngine. QCEngine — бесплатная сетевая система моделирования квантовых вычислений, которая позволяет запускать примеры прямо в браузере, без установки какого-либо программного обеспечения. Эта система моделирования была разработана авторами: сначала для личного применения, затем как приложение к книге. Система QCEngine особенно полезна для нас по двум причинам: во-первых, она может работать без загрузки дополнительного кода и, во-вторых, в нее интегрирована система *круговых обозначений*, которая будет использоваться как средство визуализации в книге.

Учебник QCEngine

Так как материал книги в значительной мере зависит от QCEngine, мы не пожалеем времени и расскажем, как работать с системой моделирования, доступной по адресу http://oreilly-qc.github.io.

Запуск кода

Веб-интерфейс QCEngine, показанный на рис. 1.1, позволяет легко строить различные визуализации, которыми мы будем пользоваться. Чтобы создать эти визуализации, достаточно ввести код в редакторе кода QCEngine.

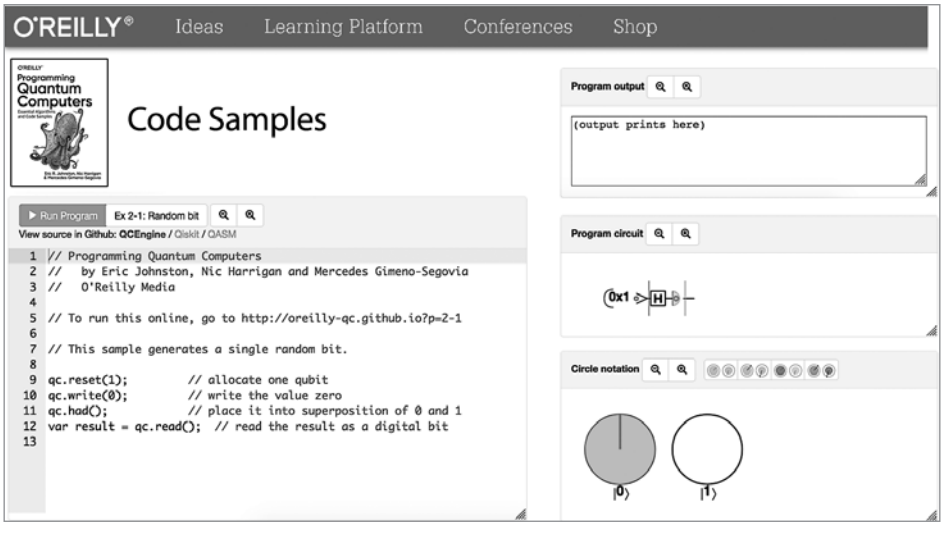


Рис. 1.1. Интерфейс QCEngine

Чтобы запустить один из примеров этой книги, выберите его в раскрывающемся списке в верхней части редактора и щелкните на кнопке Run Program. На экране появляются новые интерактивные элементы интерфейса для визуализации результатов выполнения кода (рис. 1.2).

- Визуализатор квантовой схемы (Quantum circuit visualizer) — элемент выдает визуальное представление схемы, представляющей ваш код. Условные обозначения, используемые в этих схемах, описаны в главах 2 и 3. Это представление также может использоваться для интерактивного пошагового выполнения программы (рис. 1.2).
- Визуализатор круговых обозначений (Circle-notation visualizer) — визуализация так называемой «круговой записи» регистра QPU (или системы моделирования). В главе 2 мы объясним, как читать и использовать эту запись.
- Консоль вывода QCEngine (QCEngine output console) — в этом поле появляется весь текст, который выводится вашим кодом (например, в процессе от-

ладки) командой `qc.print()`. Весь вывод стандартной функции JavaScript `console.log()` также направляется на консоль браузера JavaScript.

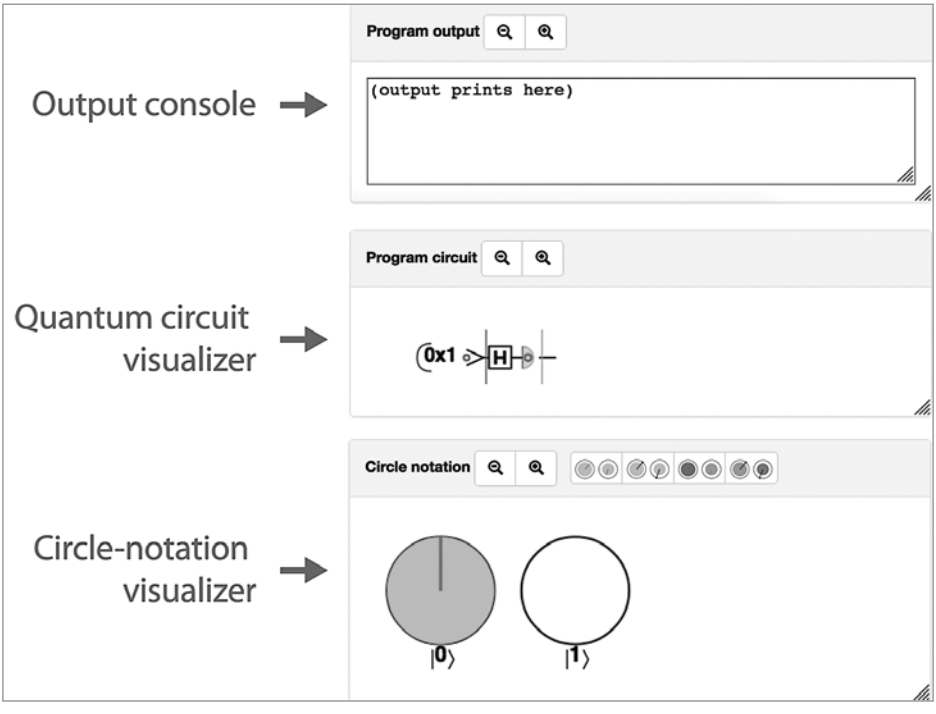


Рис. 1.2. Элементы пользовательского интерфейса QCEnging для визуализации результатов работы QPU

Отладка

Отладка программ для QPU может быть весьма непростым делом. Нередко оказывается, что единственный способ понять, как работает программа, — медленно пройти ее в пошаговом режиме, анализируя визуализации на каждом шаге. Наведя указатель мыши на визуализатор схемы, вы увидите вертикальную оранжевую черту в фиксированной позиции и серую вертикальную черту в той точке схемы, в которой находится ваш курсор. Оранжевая черта обозначает позицию схемы (а следовательно, и программы), представленную визуализатором круговой записи. По умолчанию это конец программы, но, щелкая на других частях схемы, вы сможете просмотреть в визуализаторе круговой записи конфигурацию QPU в соответствующих точках программы. Например, на рис. 1.3 показано, как визуализатор круговой записи изменяется при переходе между двумя шагами программы QCEngine.

Когда у вас появится доступ к системе моделирования QPU, вам наверняка захочется поэкспериментировать. И мы не собираемся вас останавливать! В главе 2 будет рассмотрен код QPU-программ возрастающей сложности.

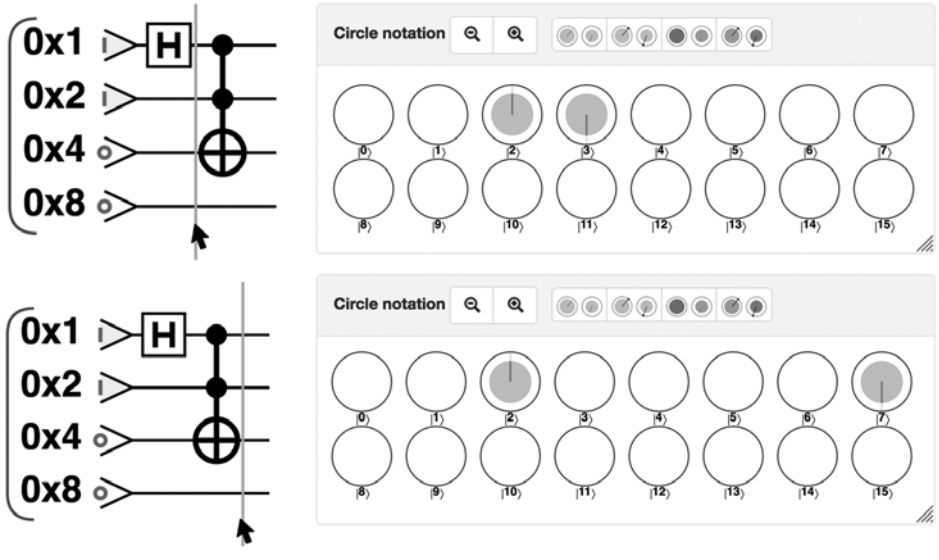


Рис. 1.3. Пошаговое выполнение программы с использованием визуализаторов круговой записи

Низкоуровневые команды QPU

QCEngine — один из нескольких инструментов для запуска и выполнения QPU-кода, но как выглядит сам код, выполняемый QPU? Для управления низкоуровневыми командами QPU обычно используются традиционные языки высокого уровня (как уже было показано с системой QCEngine на базе JavaScript). В этой книге мы будем регулярно пересекать границу между этими уровнями.

Описание программирования QPU с командами машинного уровня, имеющими специфическую квантовую природу, поможет вам освоить принципиально новую логику работы QPU, тогда как умение управлять этими операциями из традиционных высокоуровневых языков (таких как JavaScript, Python или C++) предоставляет более прагматическую парадигму для написания кода. Определение новых языков *квантового* программирования — одна из активно развивающихся областей. В книге эта

тема не описывается, но в главе 14 приведены ссылки для заинтересованных читателей.

Чтобы немного разжечь ваш интерес к теме, мы приведем сводку некоторых фундаментальных команд QPU в табл. 1.1. Все команды более подробно рассматриваются в последующих главах.

Таблица 1.1. Набор основных команд квантового процессора

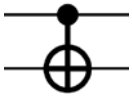
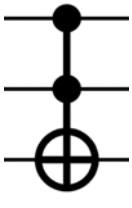
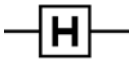
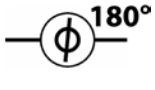
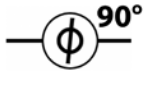
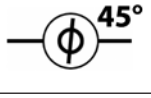
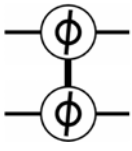
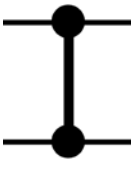
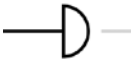
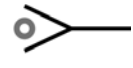
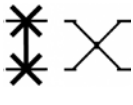
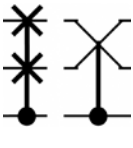
Условное обозначение	Название	Применение	Описание
	NOT (также X)	`qc.not(t)`	Логическое поразрядное отрицание (`NOT`)
	CNOT	`qc.cnot(t,c)`	Контролируемое отрицание: `if (c) then NOT(t)`
	CCNOT (вентиль Тоффоли)	`qc.cnot(t,c1| c2)`	`if (c1 AND c2) then NOT(t)`
	HAD (вентиль Адамара)	`qc.had(t)`	Вентиль Адамара
	PHASE	`qc.phase(angle,c)`	Относительный фазовый сдвиг
	Z	`qc.phase(180,c)`	Относительный фазовый сдвиг на 180°
	S	`qc.phase(90,c)`	Относительный фазовый сдвиг на 90°
	T	`qc.phase(45,c)`	Относительный фазовый сдвиг на 45°

Таблица 1.1 (*окончание*)

Условное обозначение	Название	Применение	Описание
	CPHASE	`qc.cphase(angle,c1| c2)`	Условный фазовый сдвиг
	CZ	`qc.cphase(180,c1| c2)`	Условный фазовый сдвиг на 180°
	READ	`val = qc.read(t)`	Чтение кубитов с возвращением числовых данных
	WRITE	`qc.write(t,val)`	Запись традиционных данных в кубиты
	ROOTNOT	`qc.rootnot(t)`	Корень из отрицания
	SWAP (EXCHANGE)	`qc.exchange(t1| t2)`	Перестановка двух кубитов
	CSWAP	`qc.exchange(t1| t2, c)`	Условная перестановка: `if (c) then SWAP(t1,t2)`

У каждой из этих операций конкретные команды и хронометраж зависят от производителя и архитектуры QPU. Тем не менее это основной набор базовых операций, которые, как предполагается, будут доступны на всех машинах. Эти операции образуют фундамент QPU-программирования, как команды MOV и ADD для программирования традиционных процессоров.

Ограничения моделирования

Хотя системы моделирования предоставляют потрясающие возможности для прототипизации малых QPU-программ, по сравнению с реальными QPU они безнадежно отстают по производительности. Одной из метрик мощности QPU является количество кубитов, доступных для работы[1] (квантовый эквивалент битов, о котором вскоре мы расскажем намного подробнее).

На момент издания книги мировой рекорд моделирования QPU составлял 51 кубит. На практике системы моделирования и оборудование, доступные для большинства читателей книги, справятся с 26 кубитами или около того, прежде чем замедлиться до полной остановки.

Примеры, приведенные в книге, были написаны с учетом этих ограничений. Они могут стать хорошей отправной точкой, но каждый добавляемый кубит удваивает объем памяти, необходимый для моделирования, и снижает скорость вдвое.

Аппаратные ограничения

С другой стороны, наибольшее реальное оборудование QPU, доступное на момент написания книги, составляло около 70 *физических* кубитов, тогда как наибольший QPU, доступный для массового пользователя через пакет разработки с открытым кодом Qiskit, содержит 16 физических кубитов[2]. Под физическими кубитами (в отличие от логических) подразумевается, что эти 70 кубитов не имеют механизма исправления ошибок, что делает их слишком зашумленными и ненадежными. Кубиты намного менее устойчивы, чем их традиционные аналоги; слабейшее взаимодействие с окружением может нарушить ход вычислений.

Взаимодействие с *логическими* кубитами позволяет программисту абстрагироваться от оборудования QPU и реализовать любой алгоритм из учебника, не беспокоясь о конкретных ограничениях физического оборудования. В этой книге мы сосредоточимся исключительно на программировании на уровне логических кубитов. И хотя приводимые примеры достаточно малы для запуска на меньших QPU (таких, как доступные на

---

[1] Несмотря на свою популярность в литературе как оценочной характеристики квантовых вычислений, количество кубитов, с которыми может работать устройство, в действительности является чрезмерно упрощенной метрикой, а для оценки истинной производительности QPU требуется учитывать намного более сложные аспекты.

[2] К моменту публикации книги этот показатель может уже устареть.

момент издания), абстрагирование подробностей физического оборудования означает, что навыки и интуитивные представления, которые будут выработаны, останутся исключительно полезными с развитием устройств в будущем.

QPU и GPU: некоторые общие характеристики

Идея программирования совершенно новой разновидности процессоров может показаться устрашающей, даже если она уже сформировала собственное сообщество на Stack Exchange. Ниже перечислены некоторые общие факты, относящиеся к программированию QPU.

- Программы, работающие исключительно на QPU, встречаются очень редко. Обычно программа, выполняемая на процессоре, выдает команды QPU, а потом получает результаты.
- Одни задачи очень хорошо подходят для QPU, а другие нет.
- QPU работает на отдельной тактовой частоте, нежели центральный процессор, и обычно имеет собственные специализированные аппаратные интерфейсы к внешним устройствам (например, оптическим выводам).
- Типичный QPU содержит собственную оперативную память, с которой центральный процессор не может работать эффективно.
- Простой QPU будет представлять собой одну микросхему, с которой работает портативный компьютер (а возможно, даже область внутри другой микросхемы). Более сложный QPU — большое и дорогостоящее устройство, всегда требующее специального охлаждения.
- Первые QPU, даже самые простые, достигали размеров холодильника и требовали специальных источников питания с повышенной мощностью.
- После завершения вычислений центральному процессору возвращается проекция результата, а большая часть внутренних рабочих данных QPU теряется.
- Отладка программ для QPU может быть чрезвычайно сложным делом, требующим специальных инструментов и методов. Пошаговое выполнение программ также усложняется, и часто лучшая тактика заключается во внесении изменений в программу и наблюдении за их влиянием на вывод.
- Оптимизации, ускоряющие один QPU, могут замедлять работу другого.

Звучит довольно пугающе. Но здесь есть один нюанс: замените квантовый процессор графическим, и каждое из этих утверждений останется абсолютно правильным!

Хотя квантовые процессоры представляют собой почти неземную технологию невероятной мощи, в том, что касается проблем, с которыми мы можем столкнуться при освоении их программирования, нет ничего принципиально нового — поколения программистов уже все это видели. Безусловно, в программировании QPU встречаются некоторые нюансы, которые действительно являются принципиально новыми (иначе для чего была бы нужна эта книга?), но очень многие моменты уже нам знакомы, и это должно вас приободрить. Эта задача нам по силам!

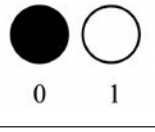
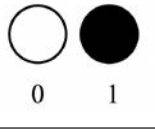
ЧАСТЬ I
Программирование для QPU

Что же это такое — кубит? Как наглядно представить его? Какая от него польза? Это короткие вопросы со сложными ответами. В первой части книги мы ответим на них с практической точки зрения. В главе 2 все начнется с описания и использования одного кубита. В главе 3 будет рассмотрена дополнительная сложность систем из многих кубитов. Попутно будут представлены многие одно- и многокубитные операции, а в главе 4 эти темы будут непосредственно использованы: мы опишем, как выполняется квантовая телепортация. Учтите, что примеры кода, демонстрирующие основные темы обсуждения, могут запускаться в системе моделирования QCEngine (см. главу 1) по предоставленным ссылкам.

2 Один кубит

Обычный бит имеет всего один двоичный параметр — бит можно инициализировать в состоянии 0 или в состоянии 1. Математика двоичной логики достаточно проста, а возможные значения 0/1 бита можно наглядно представить двумя кругами (один пустой, а другой заполненный), как показано в табл. 2.1.

Таблица 2.1. Возможные значения традиционного бита — графическое представление

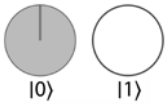
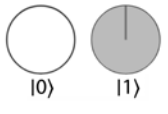
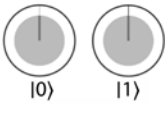
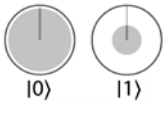
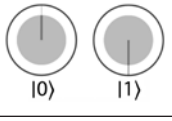
Возможные значения бита	Графическое представление
0	
1	

А теперь обратимся к кубитам. В каком-то отношении кубиты очень похожи на биты: при чтении значения кубита вы всегда получаете 0 или 1. Таким образом, *после чтения* кубита его всегда можно описать так, как показано в табл. 2.1. Однако охарактеризовать состояние кубитов *до* чтения — не столь тривиальная задача, требующая более сложного описания. Перед чтением кубиты могут существовать в *суперпозиции* состояний.

Вскоре мы попробуем объяснить, что такое суперпозиция. Но чтобы вы имели хотя бы общее представление о том, какие возможности таит эта

концепция, следует заметить, что существует бесконечное количество возможных суперпозиций, в которых отдельный кубит может существовать до чтения. В табл. 2.2 перечислены лишь некоторые суперпозиции, в которых может находиться кубит в результате подготовки. И хотя в конечном итоге вы всегда получаете 0 или 1, именно сама доступность этих дополнительных состояний открывает невероятные возможности при вычислениях, хотя для этого придется проявить некоторую изобретательность.

Таблица 2.2. Некоторые возможные значения кубита

Возможные значения кубита	Графическое представление
$\|0\rangle$	
$\|1\rangle$	
$0.707\|0\rangle + 0.707\|1\rangle$	
$0.95\|0\rangle + 0.35\|1\rangle$	
$0.707\|0\rangle - 0.707\|1\rangle$	

В обозначениях из табл. 2.2 метки 0 и 1 были заменены на $|0\rangle$ и $|1\rangle$. Эта система обозначений, называемая обозначениями Дирака, или обозначениями бра-кет, часто встречается в квантовых вычислениях. Как правило, числа, заключенные в обозначения бра-кет, представляют значения, в которых может находиться кубит при чтении. Когда речь идет о значении, которое было получено в результате чтения из кубита, мы просто используем число для представления полученного числового результата.

В первых двух строках табл. 2.2 изображены квантовые эквиваленты состояний традиционного бита без какой-либо суперпозиции. Кубит, подготовленный в состоянии $|0\rangle$, эквивалентен традиционному биту, содержащему 0 (то есть он всегда дает значение 0 при чтении); аналогичным образом дело обстоит с 1. Если бы используемые кубиты находились только в состояниях $|0\rangle$ или $|1\rangle$, то, по сути, это были бы самые обычные биты, только очень дорогие.

Но как приступить к освоению более экзотических возможностей суперпозиции, приведенных в других строках? Чтобы получить хотя бы интуитивное представление о запутанных вариантах из табл. 2.2, будет полезно кратко разобраться с тем, что же собой представляет кубит[1].

Краткий обзор физических кубитов

Объектом, который легко демонстрирует квантовую суперпозицию, является одиночный фотон. Чтобы продемонстрировать этот факт, отступим на шаг и допустим, что мы попытались использовать местоположение фотона для представления традиционного цифрового *бита*. В устройстве, показанном на рис. 2.1, переключаемое зеркало (которое может находиться в отражающем или прозрачном состоянии) позволяет управлять тем, на каком из двух путей находится фотон — соответствующему 0 или 1.

Такие устройства действительно существуют в современных технологиях цифровых коммуникаций. Бит из одиночного фотона получится очень капризным (хотя бы потому, что он не способен подолгу находиться на одном месте). Чтобы использовать эту конфигурацию для демонстрации некоторых свойств кубитов, допустим, что переключатель, переводящий фотон в состояние 0 или 1, заменен полупрозрачным зеркалом.

Полупрозрачное зеркало, изображенное на рис. 2.2 (также называемое *светоделителем*), представляет собой полуотражающую поверхность, которая с вероятностью 50% либо отражает свет на путь, связанный с состоянием 1, либо позволяет ему пройти напрямую на путь, связанный с состоянием 0. Других вариантов нет.

---

[1] В этой книге мы постараемся как можно меньше задумываться о том, чем в действительности являются кубиты. И хотя это может показаться противоестественным, стоит вспомнить, что учебники по программированию почти никогда не отвлекаются на физическую природу битов и байтов. Собственно, именно возможность абстрагироваться от физической природы информации позволяет нам справиться с написанием сложных программ.

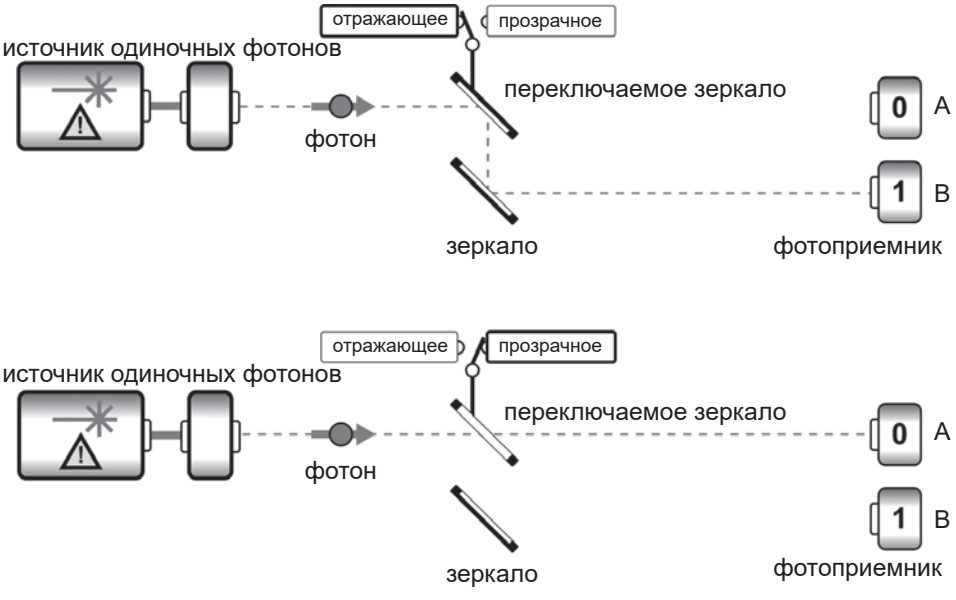


Рис. 2.1. Использование фотона в качестве традиционного бита

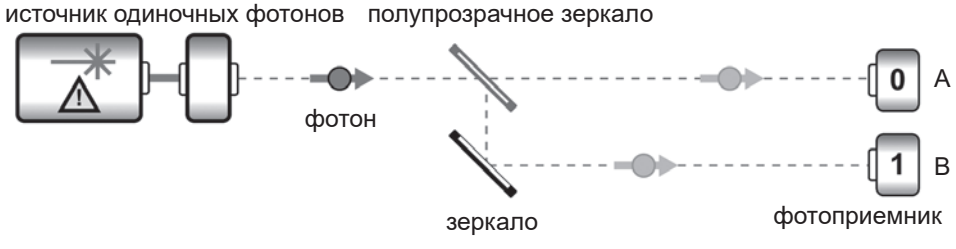


Рис. 2.2. Простая реализация одного фотонного кубита

Когда одиночный неделимый фотон сталкивается с такой поверхностью, он страдает от своеобразного «кризиса идентичности». В результате эффекта, у которого нет традиционных аналогов, он оказывается в состоянии, в котором на него могут воздействовать эффекты как пути 0, так и пути 1. Говорят, что фотон находится в *суперпозиции* перемещений по всем возможным путям. Другими словами, вместо традиционного бита появляется *кубит*, который может находиться в суперпозиции состояний, соответствующих значениям 0 и 1.

Природу суперпозиции очень легко понять неправильно (как происходит во многих популярных статьях о квантовых вычислениях). Было бы некор-

ректно утверждать, что фотон находится на путях 0 и 1 одновременно[1]. Фотон только один, и, если поставить датчики на каждом пути, как показано на рис. 2.2, сработает только один из них. Когда это произойдет, суперпозиция фотона редуцируется в числовой бит и дает определенный результат 0 или 1. Но как вскоре будет показано, существуют полезные (с вычислительной точки зрения) возможности взаимодействия QPU с кубитом в суперпозиции, прежде чем вам потребуется прочитать его.

Разновидность суперпозиции, показанная на рис. 2.2, играет центральную роль в использовании квантовых возможностей QPU. Вследствие этого к описанию и управлению квантовыми суперпозициями также следует подходить с квантовых позиций. Когда фотон находится в суперпозиции путей, мы говорим, что с каждым путем связана некоторая *комплексная амплитуда*. У комплексных амплитуд есть два важных аспекта — две «ручки переключателя», которые можно подкручивать для изменения конкретной конфигурации суперпозиции кубита.

- *Амплитуда* (magnitude), связанная с каждым путем суперпозиции фотона, представляет собой аналоговое значение, которое показывает, в какой степени фотон *распространился* на каждый путь. Амплитуда пути относится к вероятности обнаружения фотона на этом пути. А именно: *квадрат* амплитуды определяет вероятность того, что фотон будет наблюдаться на заданном пути. На рис. 2.2 можно корректировать комплексную амплитуду, связанную с каждым путем, изменяя степень отражения полупрозрачного зеркала.
- *Относительная фаза* (relative phase) между разными путями в суперпозиции фотона отражает величину задержки фотона на одном пути относительно другого. Это еще одно аналоговое значение, которым можно управлять за счет различий между перемещениями фотона на путях, соответствующих 0 и 1. Следует заметить, что относительную фазу можно изменить, не влияя на вероятность обнаружения фотона на каждом пути[2].

Стоит еще раз подчеркнуть, что термином «комплексная амплитуда» (amplitude) обозначается *совокупность* амплитуды и относительной фазы, связанных с некоторым значением из суперпозиции кубита.

---

[1] Утверждение спорно: в традиционных интерпретациях квантовой механики фотон одновременно находится на пути 0 и 1. В целом это открытый вопрос, но без дополнительных пояснений принято придерживаться стандартных интерпретаций. — *Примеч. науч. ред.*

[2] Хотя мы представляем идею относительной фазы в контексте относительных расстояний, преодолеваемых светом, это общая концепция, применимая к любым разновидностям кубитов: фотонов, электронов, сверхпроводников и т. д.

Для читателей, интересующихся математикой: комплексные амплитуды, связанные с разными путями в суперпозиции, обычно описываются комплексными числами. Амплитуда, связанная с комплексной амплитудой, соответствует модулю комплексного числа (квадратный корень числа, умноженного на комплексно-сопряженное число), тогда как относительная фаза соответствует углу при выражении комплексного числа в полярной форме. Для читателей, не интересующихся математикой, мы вскоре приведем описание визуальных обозначений, чтобы вам не приходилось отвлекаться на столь сложные материи.

Значения амплитуды и относительной фазы доступны для использования в процессе вычислений; можно считать, что они закодированы в кубите. Но если вы в какой-то момент захотите прочитать из него информацию, то фотон в конечном итоге должен столкнуться с каким-то из датчиков. И в этот момент оба аналоговых значения исчезают — квантовая природа кубита пропадает. И в этом заключается вся суть квантовых вычислений: вы должны каким-то образом использовать эти эфемерные величины, чтобы после разрушающего акта чтения получить некий полезный результат.

Конфигурация на рис. 2.2 эквивалентна примеру кода, который будет приведен в листинге 2.1 для случая, когда фотон используется в качестве кубита.

Впрочем, довольно возни с фотонами! Это руководство для программиста, а не учебник по физике. Давайте абстрагируемся от физики и посмотрим, каким образом описать и наглядно представить кубиты, отдалившись от фотонов и квантовой физики настолько, насколько двоичная логика отдаляется от электронов и физики полупроводников.

Круговая запись

Итак, вы примерно представляете, что такое суперпозиция, но это представление в значительной степени привязано к специфике поведения фотонов. Поищем абстрактный способ описания суперпозиций, который позволял бы сосредоточиться только на абстрактной информации.

Полноценное математическое обоснование квантовой физики предоставляет такую абстракцию, но, как видно из левого столбца табл. 2.2, эта математика куда менее интуитивна и менее удобна, чем простая двоичная логика традиционных битов.

К счастью, эквивалентная графическая система обозначений в правом столбце табл. 2.2 предоставляет более интуитивное описание. Так как мы стремимся сформировать свободное и прагматичное понимание того, что происходит внутри QPU, не углубляясь в малопонятную математику, с этого момента мы будем рассматривать кубиты исключительно в рамках этой круговой записи.

Эксперименты с фотонами показали, что у общего состояния кубита, которое необходимо отслеживать в QPU, есть два аспекта: *амплитуда* комплексных амплитуд ее суперпозиции и *относительная фаза* между ними. В круговой записи эти параметры представляются следующим образом:

- *величина к*омплексной амплитуды, связанной с каждым значением, которое может принимать кубит (пока что это $|0\rangle$ и $|1\rangle$), описывается радиусом заполненной области, изображенной для каждого из кругов $|0\rangle$ и $|1\rangle$;
- *относительная фаза* между комплексными амплитудами этих значений, обозначается углом поворота круга $|1\rangle$ относительно круга $|0\rangle$ (темная линия в кругах делает величину поворота более наглядной).

В этой книге круговая запись используется очень часто, поэтому мы выделим немного времени и разберемся в том, как размеры кругов и углы поворота отражают эти концепции.

Размер круга

Ранее мы упоминали о том, что *квадрат* амплитуды, связанный с $|0\rangle$ или $|1\rangle$, определяет вероятность получения этого значения при чтении. Так как заполненный *радиус* в круге представляет *амплитуду*, это означает, что площадь заполненной области в каждом круге (или в разговорной речи — ее размер) прямо пропорциональна *вероятности* получить значение этого круга (0 или 1) при чтении кубита. В примерах на рис. 2.3 представлена круговая запись для различных состояний кубитов и вероятность чтения 1 в каждом случае.

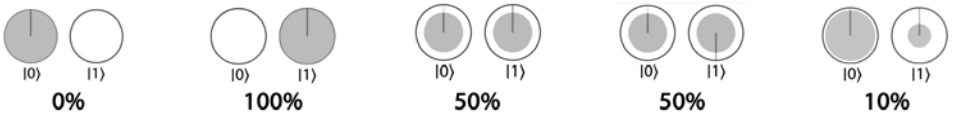


Рис. 2.3. Вероятность чтения значения 1 для разных суперпозиций, представленных в круговой записи

Чтение из кубита уничтожает информацию. Во всех случаях, представленных на рис. 2.3, при чтении кубита будет получено значение 0 или 1, и когда это произойдет, кубит изменит свое состояние в соответствии с наблюдаемым значением. Таким образом, даже если кубит изначально находился в более сложном состоянии, после того как из него будет прочитано значение 1, при немедленном повторном чтении из него вы всегда будете получать 1.

Следует заметить, что с увеличением заполненной части на круге $|0\rangle$ вероятность прочитать 0 увеличивается — а это, конечно, означает, что вероятность получить результат 1 уменьшается (до оставшейся величины). В последнем примере на рис. 2.3 существует 90%-ная вероятность получить 0 при чтении из кубита — а следовательно, вероятность прочитать 1 составляет оставшиеся 10%[1].

Мы будем часто говорить, что заполненная часть круга в круговой записи представляет амплитуду, связанную с этим значением в суперпозиции. И хотя на первый взгляд это может показаться раздражающей технической подробностью, важно помнить, что амплитуда, связанная с этим значением, в действительности соответствует *радиусу* круга, хотя часто эти две величины можно без особого вреда приравнять для визуального удобства.

Также легко забыть (хотя и важно помнить), что в круговой записи размер круга, связанный с конкретным результатом, *не* представляет полной *комплексной амплитуды* суперпозиции. Отсутствует важная информация: относительная фаза суперпозиции.

Поворот

Некоторые команды QPU также позволяют изменять относительные повороты кругов $|0\rangle$ и $|1\rangle$ кубита. Поворот представляет *относительную фазу* кубита. Относительная фаза состояния кубита может принимать любые значения из диапазона от 0° до 360°; некоторые примеры представлены на рис. 2.4.

По соглашениям, принятым в этой книге, положительный угол поворота соответствует вращению против часовой стрелки, как видно из рис. 2.4.

[1] Сумма амплитуд регистра всегда должна быть равна 1. Это требование называется нормировкой; за подробностями обращайтесь к разделу «Предупреждение 2: требование для нормированных векторов».

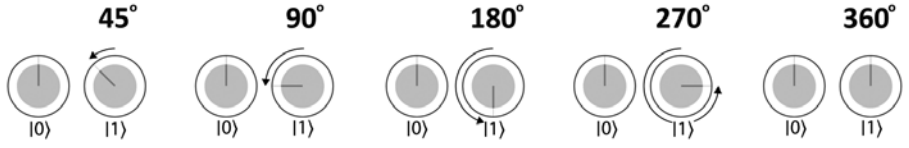


Рис. 2.4. Примеры относительных фаз в одном кубите

Во всех предыдущих примерах поворачивался только круг $|1\rangle$. Почему не поворачивается круг $|0\rangle$? Как следует из названия, важна лишь *относительная* фаза в суперпозиции кубита. Соответственно, нас интересует лишь *относительный* поворот между двумя кругами[1]. Если бы операция QPU должна была применить поворот к обоим кругам, то эффект всегда можно заново интерпретировать в эквивалентной форме так, словно повернут был только круг $|1\rangle$, что делает относительный поворот более очевидным. Пример показан на рис. 2.5.

Рис. 2.5. В круговой записи важны только относительные повороты — показанные два состояния эквивалентны, потому что относительные фазы в каждом случае совпадают

Заметим, что относительная фаза может изменяться независимо от амплитуды суперпозиции. Эта независимость также работает и в обратном направлении. При сравнении третьего и четвертого примера на рис. 2.3 становится видно, что относительная фаза между выходными значениями для одного кубита напрямую не влияет на вероятность того, какое именно значение будет прочитано.

Тот факт, что относительная фаза одного кубита не влияет на амплитуды в суперпозиции, означает, что она не оказывает *прямого* влияния на наблюдаемые результаты чтения. Может показаться, что из-за этого свойство относительной фазы становится несущественным, однако это впечатление невероятно далеко от истины! В квантовых вычислениях с участием *нескольких* кубитов можно воспользоваться поворотом, чтобы нетривиально

[1] Тот факт, что важны только относительные фазы, следует из законов квантовой механики, по которым действуют кубиты.

и косвенно повлиять на вероятность того, что со временем будут прочитаны разные значения. Более того, хорошо продуманные относительные фазы могут открыть невероятные вычислительные возможности. А теперь мы опишем операции, которые позволят нам это сделать (особенно операции, работающие только с одним кубитом), и наглядно представим их эффект при помощи круговой записи.

В отличие от традиционных битов, имеющих ярко выраженную цифровую природу, амплитуды и относительные фазы относятся к непрерывным степеням свободы. Этот факт породил широко распространенное заблуждение, будто квантовые вычисления чем-то похожи на злосчастные аналоговые вычисления. Однако стоит заметить, что несмотря на возможности манипуляций с разными степенями свободы, ошибки, возникающие в ходе работы QPU, могут исправляться цифровыми методами. Это одна из причин, по которой QPU превосходят аналоговые вычислительные устройства по устойчивости к ошибкам.

Первые операции QPU

Однокубитные операции QPU, как и их аналоги для традиционных процессоров, преобразуют входную информацию в нужный вывод. Только, конечно, в этом случае входные и выходные данные определяются кубитами, а не битами. Многие команды QPU имеют соответствующие обратные операции[1], о которых также полезно знать. В таких случаях операция QPU называется *обратимой* (reversible); по сути, это означает, что ее применение не приводит к потере информации. С другой стороны, некоторые операции QPU являются необратимыми и не имеют обратных операций (то есть они каким-либо образом приводят к потере информации). Как вы вскоре увидите, обратимость или необратимость операции может иметь важные последствия для ее использования.

На первый взгляд некоторые команды QPU выглядят странно, а их полезность далеко не очевидна, но лишь после изучения целой группы таких операций мы быстро найдем им практическое применение.

Команда QPU: NOT

NOT (отрицание) — квантовый эквивалент одноименной традиционной операции. Нуль переходит в единицу, и наоборот. Тем не менее, в отличие от своего традиционного аналога, опе-

[1] В этой книге термины «команда QPU» и «операция QPU» считаются равнозначными.

рация NOT у QPU также может работать с кубитом, находящимся в суперпозиции.

В круговой записи эти результаты представляются очень просто: круги $|0\rangle$ и $|1\rangle$ меняются местами, как показано на рис. 2.6.

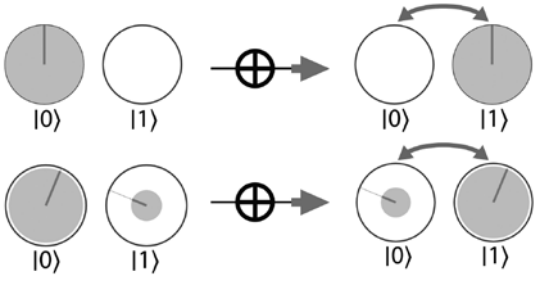
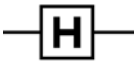


Рис. 2.6. Операция NOT в круговой записи

Обратимость: как и в цифровой логике, операция NOT обратима по отношению к самой себе; ее повторное применение возвращает кубиту исходное значение.

Команда QPU: HAD



Операция HAD (сокращение от фамилии Hadamard) фактически создает равновероятную суперпозицию при получении значения $|0\rangle$ или $|1\rangle$. Она открывает путь к использованию необычного и тонкого параллелизма квантовой суперпозиции. В отличие от NOT, операция HAD не имеет традиционного аналога.

В круговой записи операция HAD приводит к тому, что выходной кубит имеет одинаковые заполненные площади для $|0\rangle$ и $|1\rangle$, как показано на рис. 2.7.

Это позволяет использовать операцию HAD для получения равномерных суперпозиций выходных значений кубита, то есть суперпозиций, в которых оба результата имеют одинаковую вероятность. Также обратите внимание на то, что воздействие HAD на кубиты, изначально находящиеся в состояниях $|0\rangle$ и $|1\rangle$, несколько различается: при воздействии операции HAD на $|1\rangle$ происходит ненулевой поворот (добавляется относительная фаза) одного из кругов, тогда как при воздействии на $|0\rangle$ он отсутствует.

Разумно спросить, что произойдет, если применить операцию HAD к кубитам, уже находящимся в состоянии суперпозиции? Лучше всего провести эксперимент! А он показывает следующее:

- операция HAD воздействует на состояния $|0\rangle$ и $|1\rangle$ по отдельности в соответствии с правилами, изображенными на рис. 2.7;
- сгенерированные значения $|0\rangle$ и $|1\rangle$ объединяются с назначением весов по комплексным амплитудам исходных суперпозиций[1].

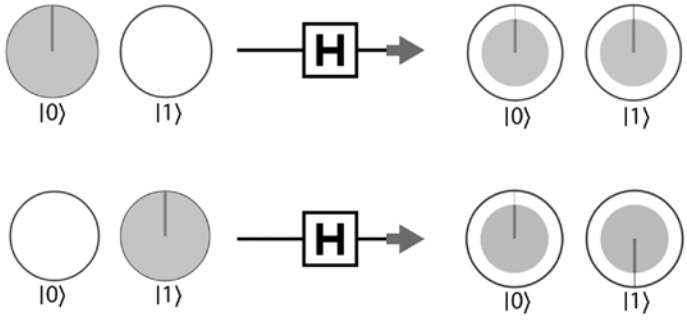
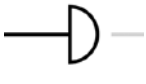


Рис. 2.7. Применение операции HAD к некоторым базовым состояниям

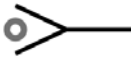
Обратимость: по аналогии с NOT, операция HAD обратима по отношению к самой себе; ее повторное применение возвращает кубиту исходное значение.

Команда QPU: READ



Операция READ является формальным выражением представленного ранее процесса *чтения*. Операция READ уникальна тем, что это единственная часть набора команд QPU, которая формально возвращает *случайный* результат.

Команда QPU: WRITE



Операция WRITE инициализирует регистр QPU перед выполнением с ним каких-либо действий. Процесс записи является детерминированным.

Применение READ к одному кубиту возвращает значение 0 или 1 с вероятностями, определяемыми соответствующими амплитудами (а точнее, их квадратами) в состоянии кубита (без учета относительной фазы). После

[1] За подробностями о математике, лежащей в основе HAD и других распространенных операций, обращайтесь к главе 14.

операции READ кубит остается в состоянии $|0\rangle$, если был получен результат 0, или в состоянии $|1\rangle$, если был получен результат 1. Другими словами, любая суперпозиция необратимо разрушается.

В круговой записи результат определяется вероятностью, определяемой заполненной площадью каждого круга. Затем происходит перераспределение заполненной области в кругах, отражающее полученный результат: круг, связанный с наблюдаемым результатом, становится полностью заполненным, а другой круг становится пустым. Эта ситуация показана на рис. 2.8 для операций READ, выполняемых с двумя разными суперпозициями.

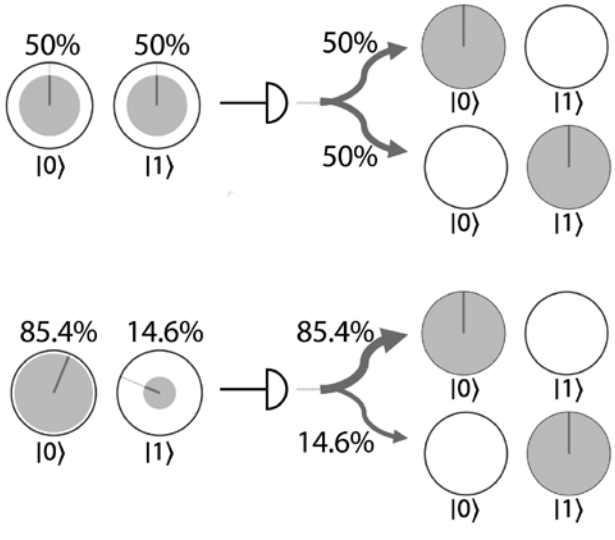


Рис. 2.8. Операция READ дает случайный результат

Во втором примере на рис. 2.8 операция READ уничтожает всю содержательную информацию относительной фазы. В результате состояние кубита переориентируется так, что радиус круга указывает вверх.

На базе READ и NOT также можно построить простую операцию WRITE, которая позволяет подготовить кубит в требуемом состоянии $|0\rangle$ или $|1\rangle$. Сначала состояние кубита читается операцией READ, а затем, если значение не соответствует тому, которое требуется записать, выполняется операция NOT. Обратите внимание: эта операция WRITE не позволяет подготовить кубит

в произвольной суперпозиции (с произвольной магнитудой и относительной фазой), а только в состоянии $|0\rangle$ или в состоянии $|1\rangle$[1].

Обратимость: операции READ и WRITE необратимы. Они уничтожают суперпозиции и приводят к потере информации. После их выполнения аналоговые значения кубита (магнитуда и фаза) безвозвратно теряются.

Практический пример: идеально случайный бит

Прежде чем переходить к описанию еще нескольких однокубитных операций, задержимся ненадолго и посмотрим, как при помощи операций HAD, READ и WRITE написать программу для выполнения операции, невозможной на любом традиционном компьютере: мы сгенерируем *действительно* случайный бит.

В истории вычислений колоссальные затраты времени и усилий были посвящены разработке систем генерирования псевдослучайных чисел (ГПСЧ), которые находят применение в различных областях, от криптографии до метеорологических прогнозов. ГПСЧ называются псевдослучайными в том смысле, что если вы знаете содержимое памяти компьютера и алгоритм ГПСЧ, возможно — теоретически — предсказать следующее генерируемое число.

В соответствии с известными законами физики, чтение кубита в суперпозиции фундаментально и идеально непредсказуемо. Это позволяет QPU создать лучший в мире генератор случайных чисел: следует просто подготовить кубит в состоянии $|0\rangle$, применить команду HAD и затем прочитать кубит. Эту комбинацию операций QPU можно продемонстрировать с использованием диаграммы *квантовой схемы*, на которой линия, проходящая слева направо, демонстрирует последовательность различных операций, выполняемых с (одним) кубитом, как показано на рис. 2.9.

Может, это и не произведет особого впечатления, но перед вами наша первая программа QPU: квантовый генератор случайных чисел (КГСЧ)! Его можно смоделировать фрагментом кода из листинга 2.1. Если снова и снова выполнять эти четыре строки кода в QCEngine, вы получите двоичную случайную строку. Конечно, системы моделирования на базе традиционных процессоров (такие как QCEngine) в действительности аппроксимируют квантовый генератор случайных чисел на базе генератора псевдослу-

[1] Вскоре вы увидите, что возможность подготовки произвольной суперпозиции — нетривиальная, но полезная задача, особенно в квантовых приложениях машинного обучения. Способы решения этой задачи представлены в главе 13.

чайных чисел, но при выполнении эквивалентного кода на реальном QPU будет получена идеальная двоичная строка.

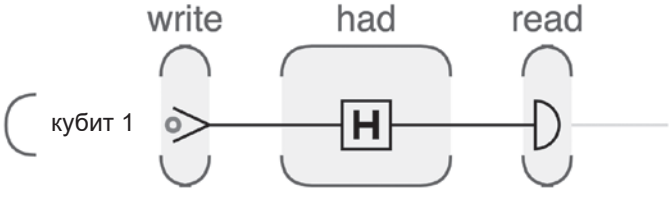


Рис. 2.9. Генерирование идеально случайного бита с использованием QPU

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=2-1.

Листинг 2.1. Один случайный бит

```
qc.reset(1);          // Выделить один кубит
qc.write(0);          // Записать нулевое значение
qc.had();             // Перевести в суперпозицию 0 и 1
var result = qc.read();  // Прочитать результат в виде цифрового бита
```

Все примеры кода из этой книги доступны онлайн по адресу http://oreilly-qc.github.io и могут выполняться как в системах моделирования QPU, так и на реальном физическом QPU. Выполнение этих примеров является существенной частью изучения программирования QPU. За дополнительной информацией обращайтесь к главе 1.

Так как эта программа может быть вашей первой квантовой программой (поздравляем!), мы разобьем ее на части, чтобы лучше понять каждую строку кода.

- `qc.reset(1)` готовит моделирование QPU, запрашивая один кубит. Все программы, которые мы напишем для QCEngine, будут инициализировать группу кубитов командами подобного вида.
- `qc.write(0)` просто инициализирует один кубит в состоянии $|0\rangle$ — эта операция эквивалентна инициализации традиционного бита значением 0.
- `qc.had()` применяет операцию HAD к кубиту, переводя его в суперпозицию состояний $|0\rangle$ и $|1\rangle$, как показано на рис. 2.7.

- `var result = qc.read()` читает значение нашего кубита в конце вычислений. Полученный случайный цифровой бит присваивается переменной `result`.

Казалось бы, в результате у нас всего лишь получился очень дорогой способ бросания монетки, но такая точка зрения недооценивает мощь операции HAD. Если бы вам удалось каким-то образом *заглянуть внутрь* операции HAD, вы бы не нашли там ни генератора псевдослучайных чисел, ни квантового генератора случайных чисел. Непредсказуемость операции HAD обеспечивается не этими генераторами, а законами квантовой физики. Никто и ничто в известной Вселенной не сможет предсказать, какое значение будет прочитано из кубита после HAD — 0 или 1 — даже при точном знании всех команд, которые использовались для генерирования случайных чисел.

Собственно, хотя в следующей главе будет рассказано о том, как работать сразу с несколькими кубитами, нашу программу случайного кубита можно легко запустить параллельно в восьми экземплярах для получения случайного байта. На рис. 2.10 показано, как это происходит.

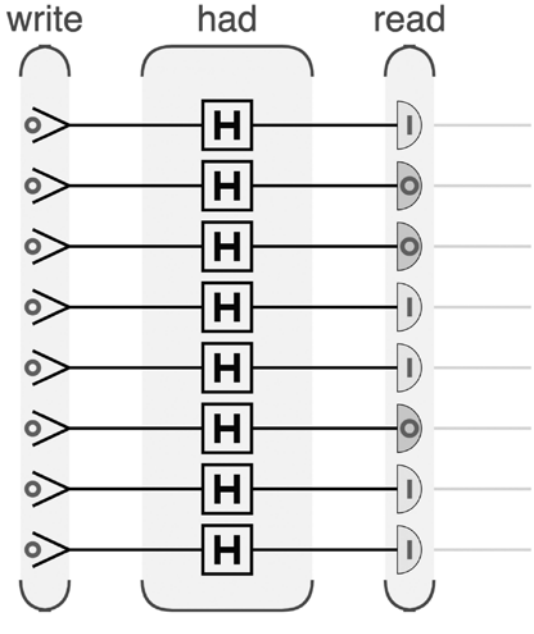


Рис. 2.10. Генерирование одного случайного байта

Код создания случайного байта из листинга 2.2 почти идентичен коду из листинга 2.1.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=2-2.

Листинг 2.2. Один случайный байт

```
qc.reset(8);
qc.write(0);
qc.had();
var result = qc.read();
qc.print(result);
```

Обратите внимание: мы используем тот факт, что такие операции QCEngine, как WRITE и HAD, по умолчанию работают со всеми инициализированными кубитами, если только им не будут явно переданы конкретные кубиты, с которыми они могут работать.

Хотя в листинге 2.2 используется целая группа кубитов, не существует многокубитных операций, которые получают на входе более одного кубита. Эту программу можно привести к последовательной форме, чтобы она работала только с одним кубитом.

Команды QPU: PHASE(θ)

Операция PHASE(θ) тоже не имеет традиционного аналога. Эта команда позволяет напрямую управлять относительной фазой кубита, изменяя ее до заданного угла. Таким образом, кроме кубита операция PHASE(θ) получает дополнительный (числовой) входной параметр — угол поворота. Например, PHASE(45) обозначает операцию PHASE, которая выполняет поворот фазы на 45°.

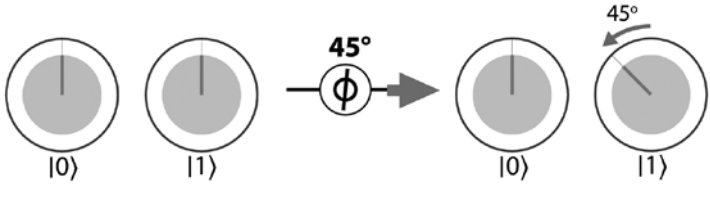


Рис. 2.11. Операция PHASE(45)

Обратите внимание: операция PHASE поворачивает только круг, связанный с состоянием $|1\rangle$, поэтому она никак не влияет на кубит в состоянии $|0\rangle$.

Обратимость: операции PHASE обратимы, хотя в общем случае не являются обратимыми к самим себе. Операция PHASE может быть обращена применением операции PHASE с углом, обратным к исходному. В круговой записи это соответствует возврату к исходному состоянию посредством поворота в обратном направлении.

При помощи операций HAD и PHASE можно получить однокубитные квантовые состояния, которые встречаются настолько часто, что им были присвоены собственные обозначения: $|+\rangle$, $|-\rangle$, $|+Y\rangle$ и $|-Y\rangle$ (рис. 2.12). Если вам захочется проверить, насколько хорошо вы осваиваете логику QPU, попробуйте определить, как получить эти состояния с использованием операций HAD и PHASE (все показанные суперпозиции имеют одинаковые магнитуды в каждом из состояний $|0\rangle$ и $|1\rangle$).

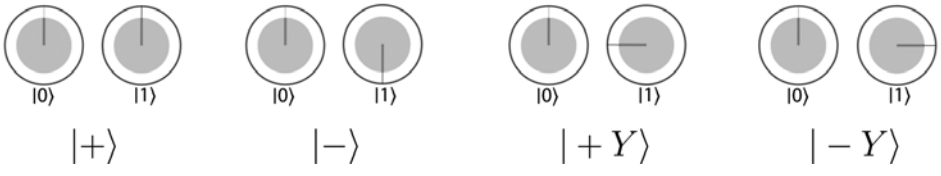


Рис. 2.12. Четыре распространенных однокубитных состояния

Эти четыре состояния будут использоваться в листинге 2.4. И, хотя они могут быть получены с использованием операций HAD и PHASE, они также могут рассматриваться как результаты так называемых однокубитных операций *поворота*.

Команды QPU: ROTX(θ) и ROTY(θ)

Как было показано ранее, операция PHASE поворачивает относительную фазу кубита, что в круговой записи соответствует повороту круга, связанного со значением $|1\rangle$. Две другие распространенные операции, связанные с PHASE — ROTX(θ) и ROTY(θ), — выполняют несколько отличающиеся разновидности поворота с кубитом.

На рис. 2.13 показан результат применения операций ROTX(45) и ROTY(45) к состояниям $|0\rangle$ и $|1\rangle$ в круговой записи.

Эти операции не соответствуют нашим интуитивным представлениям о поворотах — по крайней мере, не так очевидно, как для PHASE. Названия этих поворотов происходят от другого визуального представления состояния одного кубита, называемого *сферой Блоха*. В представлении сферы Блоха состояние кубита представляется точкой на поверхности трехмер-

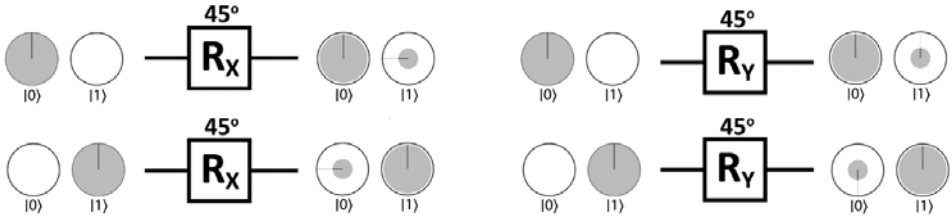


Рис. 2.13. Воздействие операций ROTX и ROTY на входные состояния 0 и 1

ной сферы. В этой книге вместо сферы Блоха используется круговая запись, так как сфера Блоха плохо расширяется на группы кубитов. Но для удовлетворения этимологической любознательности отметим, что при представлении кубита на сфере Блоха операции ROTY и ROTX соответствуют повороту точки кубита вокруг оси x и y сферы соответственно. Этот смысл теряется в круговой записи, так как вместо трехмерной сферы используются два двухмерных круга. Собственно, операция PHASE в действительности соответствует повороту по оси z при представлении кубитов на сфере Блоха, поэтому иногда ее обозначают ROTZ.

COPY: недостающая операция

Одна из операций традиционных компьютеров *не может быть* реализована на базе QPU. Хотя мы можем создать несколько копий известного состояния посредством его многократной инициализации (для состояний $|0\rangle$ или $|1\rangle$ это делается простым применением операции WRITE), невозможно скопировать промежуточное состояние квантовых вычислений без его конкретного определения. Это ограничение обусловлено фундаментальными законами физики, управляющими работой кубитов.

Это определенно создает неудобства, но, как будет показано в следующих главах, для QPU существуют другие возможности, которые могут компенсировать отсутствие команды COPY.

Объединение операций QPU

Итак, в нашем распоряжении появились операции NOT, HAD, PHASE, READ и WRITE. Стоит упомянуть о том, что в традиционной логике эти операции могут комбинироваться для реализации друг друга, и даже для создания совершенно новых операций. Например, предположим, что QPU поддер-

живает команды HAD и PHASE, но не NOT. Объединение операции PHASE(180) с двумя операциями HAD позволяет получить точный эквивалент NOT, как показано на рис. 2.14. Соответственно, команда PHASE(180) также может быть реализована на базе операций HAD и NOT.

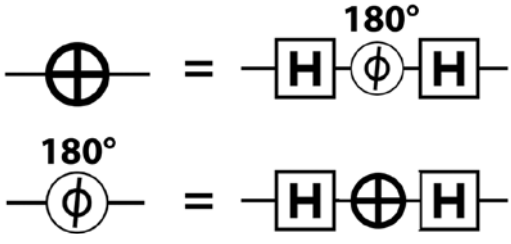


Рис. 2.14. Построение эквивалентных операций

Команда QPU: ROOT-NOT

Объединение команд также позволяет создавать интересные новые операции, не существующие в мире традиционной логики. Одним из примеров такого рода является операция ROOT-NOT (RNOT). Операция называется квадратным корнем из NOT в том смысле, что при двукратном применении она выполняет одну операцию NOT, как показано на рис. 2.15.

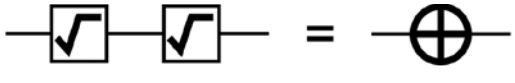


Рис. 2.15. Операция, невозможная для традиционных битов

Существуют разные способы построения таких операций, но одна из простых реализаций представлена на рис. 2.16.

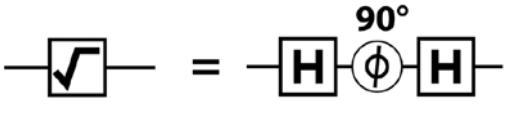


Рис. 2.16. Возможная реализация ROOT-NOT

Чтобы убедиться в том, что двукратное применение этого набора операций дает тот же результат, что и NOT, можно воспользоваться моделированием из листинга 2.3.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=2-3.

Листинг 2.3. Демонстрация результата RNOT

```
qc.reset(1);          // Выделить один кубит
qc.write(0);

// Root-not
qc.had();
qc.phase(90);
qc.had();

// Root-not
qc.had();
qc.phase(90);
qc.had();
```

В круговой записи можно наглядно представить каждый шаг, задействованный в реализации RNOT (операция PHASE(90) между двумя HAD). Полученная операция показана на рис. 2.17.

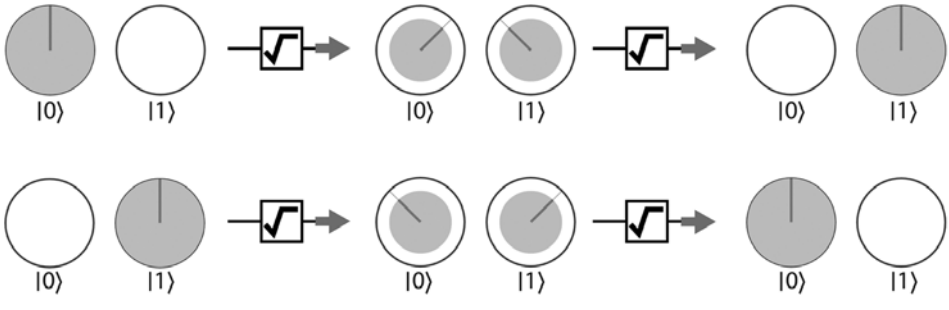


Рис. 2.17. Эффект операции ROOT-NOT

Наблюдая за изменением состояния кубита в круговой записи, можно увидеть, как RNOT проходит половину пути операции NOT. Как было показано на рис. 2.14, после применения операции HAD к кубиту, при повторном повороте его относительной фазы на 180°, другая операция HAD приведет к результату, эквивалентному операции NOT. RNOT выполняет половину этого поворота (PHASE(90)), поэтому два применения будут эквивалентны применению последовательности HAD-PHASE(180)-HAD, эквивалентной NOT. На первый взгляд все это выглядит довольно запутанно, но все же попробуйте разобраться

в том, что операция RNOT достигает этой цели при повторном применении (стоит напомнить, что операция HAD является обратимой по отношению к самой себе, так что серия из двух операций HAD эквивалентна пустой операции).

Обратимость: хотя операции RNOT никогда не являются обратимыми по отношению к самим себе, для построения операции, обратной к рис. 2.16, можно воспользоваться отрицательным значением фазы, как показано на рис. 2.18.

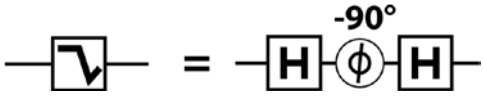


Рис. 2.18. Операция, обратная RNOT

И хотя операция RNOT может показаться экзотической странностью, она дает нам важный урок: тщательное кодирование информации в относительной фазе кубита позволяет реализовать совершенно новые разновидности вычислений.

Практический пример: квантовая проверка защиты

В завершение этой главы мы рассмотрим более сложную программу, которая нагляднее демонстрирует, какие возможности открывает использование относительной фазы в кубитах. В коде из листинга 2.4 простые однокубитные операции QPU, представленные ранее, используются для реализации упрощенной версии *квантового распределения ключей* (QKD, Quantum Key Distribution). Протокол QKD занимает центральное место в области квантовой криптографии, которая обеспечивает доказуемую криптостойкость при передаче информации.

Допустим, два QPU-программиста, Алиса (Alice) и Боб (Bob; обратите внимание, далее в схемах и примерах кода кубит Боб обозначается как «B». — *Примеч. ред.*), обмениваются данными по каналу связи, способному передавать кубиты. Время от времени они передают специально сконструированный «проверочный» кубит, описанный в листинге 2.4, который используется для проверки того, не был ли взломан канал связи.

Шпионская попытка прочитать один из таких кубитов будет обнаружена с вероятностью 25%. Таким образом, если Алиса и Боб используют всего

50 проверок во всей передаче, вероятность остаться необнаруженным для шпиона оказывается заметно ниже 1/1 000 000.

Чтобы выявить факт взлома ключа, Алиса и Боб могут обмениваться традиционными цифровыми данными, которые не нужно шифровать или скрывать. После обмена сообщениями они проверяют некоторые из своих кубитов; для этого они читают кубиты и убеждаются в том, что они согласуются друг с другом по некоторому заранее известному правилу. Если будет обнаружено несоответствие, значит, кто-то вмешался в передачу. Процесс изображен на рис. 2.19.

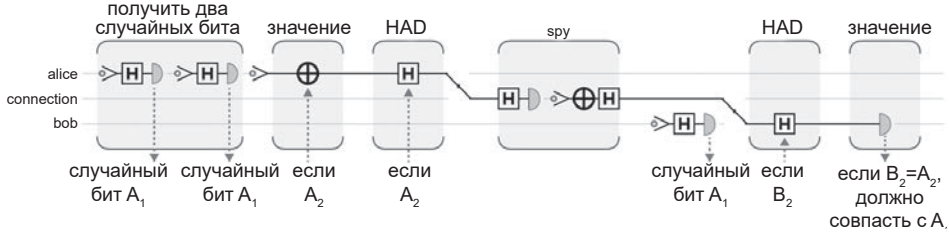


Рис. 2.19. Квантовая программа проверки защиты

Мы рекомендуем опробовать код в листинге 2.4 самостоятельно. Поэкспериментируйте и проверьте его так, как вы бы поступили с любым другим фрагментом кода.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=2-4.

Листинг 2.4. Квантовая проверка защиты

```
qc.reset(3);
qc.discard();
var a = qint.new(1, 'alice');
var fiber = qint.new(1, 'fiber');
var b = qint.new(1, 'bob');

function random_bit(q) {
    q.write(0);
    q.had();
    return q.read();
}

// Сгенерировать два случайных бита
```

```
var send_had = random_bit(a);
var send_val = random_bit(a);

// Подготовить кубит Алисы
a.write(0);
if (send_val)  // Использовать случайный бит для задания значения
    a.not();
if (send_had)  // Использовать случайный бит для решения
               // о применении HAD
    a.had();

// Отправить кубит!
fiber.exchange(a);

// Активизировать шпиона
var spy_is_present = true;
if (spy_is_present) {
    var spy_had = 0;
    if (spy_had)
        fiber.had();
    var stolen_data = fiber.read();
    fiber.write(stolen_data);
    if (spy_had)
        fiber.had();
}

// Получить кубит!
var recv_had = random_bit(b);
fiber.exchange(b);
if (recv_had)
    b.had();
var recv_val = b.read();

// Алиса сообщает Бобу свой выбор операций и значения
// по электронной почте.
// Если выбор совпадает, а значения нет, значит,
// канал прослушивается шпионом!
if (send_had == recv_had)
    if (send_val != recv_val)
        qc.print('Caught a spy!\n');
```

В листинге 2.4 каждой из сторон (Алисе и Бобу) доступен простой QPU с одним кубитом; для отправки кубитов используется квантовый канал связи. Канал может прослушиваться шпионом; в приведенном примере кода вы можете управлять присутствием шпиона при помощи переменной `spy_is_present`.

Тот факт, что квантовая криптография может быть реализована на таких относительно малых QPU — одна из причин, по которым она стала находить коммерческие применения задолго до появления более мощных QPU общего назначения.

Разберем этот код шаг за шагом и посмотрим, как простые ресурсы Алисы и Боба позволили им решить эту задачу.

```
// Сгенерировать два случайных бита
```

Алиса использует свой однокубитный QPU как простой квантовый ГСЧ точно так же, как было сделано в листинге 2.2; она генерирует два секретных случайных бита, которые известны только ей. Эти биты обозначаются `send_val` и `send_had`.

```
// Подготовить кубит Алисы
```

Алиса готовит свой «проверочный» кубит с использованием двух своих случайных битов. Она записывает в него значение `value`, а затем использует `send_had` для принятия решения о том, нужно ли применять HAD. По сути она инициализирует свой кубит случайным образом в одном из состояний $|0\rangle$, $|1\rangle$, $|+\rangle$ или $|-\rangle$, но (пока) никому не сообщает, что это за состояние. Если Алиса решит применить операцию HAD, то если Боб захочет извлечь ожидаемое значение (0 или 1), он должен будет обратить операцию HAD (другой операцией HAD) перед выполнением READ.

```
// Отправить кубит!
```

Алиса отправляет свой кубит Бобу. Для наглядности в данном примере другой кубит используется для представления канала связи.

```
// Активизировать шпиона
```

Если бы Алиса передавала традиционные цифровые данные, то шпион просто скопирует бит и добьется своей цели. С кубитами это невозможно. Вспомните, что операции COPY не существует, поэтому шпион может сделать только одно: прочитать отправленный Алисой кубит операцией READ, а затем попытаться отправить Бобу точно такой же кубит, чтобы избежать обнаружения. Однако следует помнить, что чтение кубита приводит к необратимой потере информации, поэтому у шпиона останется только традиционный бит на выходе. Шпион не знает, выполняла Алиса операцию HAD или нет. В результате он не будет знать, нужно ли применять вторую операцию HAD перед выполнением своей команды READ. Если он просто выполнит READ, то не будет знать, получил ли он случайное значение из куби-

та в суперпозиции или значение, фактически закодированное Алисой. Это означает, что он не только не сможет надежно извлечь бит Алисы, но и не будет знать, какое состояние следует передать Бобу, чтобы не быть обнаруженным.

```
// Получить кубит!
```

Как и Алиса, Боб генерирует случайный бит `recv_had` и использует его для принятия решения о том, нужно ли применять операцию `HAD` перед применением операции `READ` к кубиту Алисы для получения значения. Это означает, что в одних случаях Боб будет (случайно) правильно декодировать двоичное значение, а в других — нет.

```
// Если выбор совпадает, а значения - нет, значит,
// канал прослушивается шпионом!
```

После получения кубита Алиса и Боб могут открыто сравнить случаи, в которых их решения о применении (или неприменении) операции `HAD` правильно совпали. Если они оба случайным образом согласились применять (или не применять) `HAD` (что будет происходить в половине случаев), биты значений должны совпасть, то есть Боб правильно расшифрует сообщение Алисы. Если в этих *правильно расшифрованных сообщениях* значения *не согласуются*, можно сделать вывод, что шпион прочитал сообщение операцией `READ` и отправил Бобу *неправильный* кубит замены, что привело к искажению расшифровки.

Итоги

В этой главе был представлен способ описания отдельных кубитов, а также различные команды QPU для работы с ними. Случайный характер операции `READ` был использован для построения квантового генератора случайных чисел, а возможность управления относительной фазой в кубите была применена для реализации простейшей схемы квантовой криптографии.

Круговая запись, использованная для визуализации состояния кубита, также широко применяется в последующих главах. В главе 3 будет представлена расширенная круговая запись для многокубитных систем, а также новые операции QPU, используемые для работы с ними.

3 Группы кубитов

Какими бы полезными ни были одиночные кубиты, при объединении в группы они становятся намного мощнее (и интереснее). В главе 2 уже было показано, как явление суперпозиции с его специфической квантовой природой вводит в вычисления новые параметры амплитуды и относительной фазы. Когда для вашего QPU доступно сразу несколько кубитов, появляется возможность использования другого замечательного квантового феномена, называемого *запутанностью* (entanglement). Квантовая запутанность — очень специфическая разновидность взаимодействия между кубитами. В этой главе мы применим ее на практике довольно сложными и нетривиальными способами.

Но чтобы исследовать возможности групп кубитов, сначала необходимо найти средства для их визуального представления.

Круговая запись для многокубитных регистров

Можно ли расширить круговую запись для нескольких кубитов? Если кубиты *не взаимодействуют* друг с другом, можно просто применить несколько экземпляров представления, используемого для одиночного кубита. Иначе говоря, можно использовать круги для обозначения состояний $|0\rangle$ и $|1\rangle$ каждого кубита. Хотя это наивное представление позволит описать суперпозицию каждого *отдельного* кубита, существуют суперпозиции групп кубитов, которые она представить не сможет.

Как еще круговая запись могла бы представить состояние регистра из нескольких кубитов? Как и в случае с традиционными битами, регистр из N кубитов может использоваться для представления одного из 2^N разных

значений. Например, регистр из трех кубитов в состояниях $|0\rangle|1\rangle|1\rangle$ может представлять десятичное значение 3. Говоря о многокубитных регистрах, мы часто описываем десятичное значение, представляемое регистром, в той же квантовой записи, которая используется для одиночного кубита; если отдельный кубит может кодировать состояния $|0\rangle$ и $|1\rangle$, регистр из двух кубитов способен кодировать значения $|0\rangle$, $|1\rangle$, $|2\rangle$ и $|3\rangle$. Используя квантовую природу кубитов, мы также можем создавать суперпозиции этих значений. Для представления таких суперпозиций N кубитов используется отдельный круг для каждого из 2^N разных значений, которые может принимать N-разрядное число (рис. 3.1).

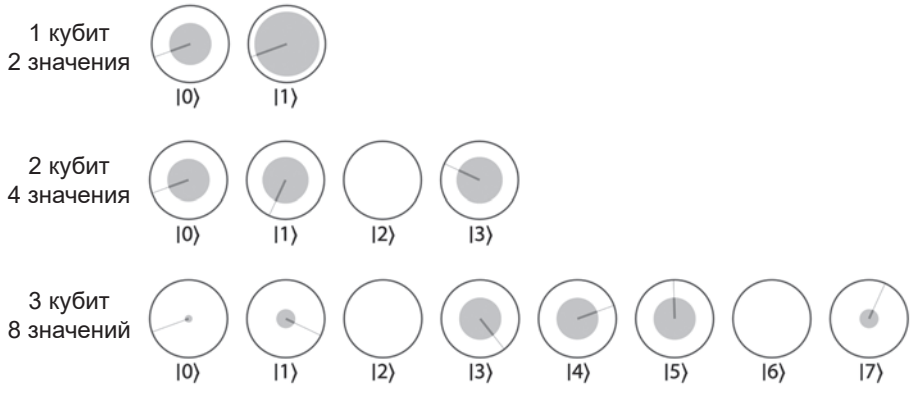


Рис. 3.1. Круговая запись для разного количества кубитов

На рис. 3.1 мы видим уже знакомое представление одного кубита из двух кругов $|0\rangle$, $|1\rangle$. Для двух кубитов используются круги $|0\rangle$, $|1\rangle$, $|2\rangle$, $|3\rangle$. Конечно, на этой иллюстрации представляется не пара кругов на кубит, а один круг на каждое возможное двухразрядное число, которое может быть получено при чтении этих кубитов. Для трех кубитов регистр QPU способен принимать значения $|0\rangle$, $|1\rangle$, $|2\rangle$, $|3\rangle$, $|4\rangle$, $|5\rangle$, $|6\rangle$, $|7\rangle$, так как при чтении может быть получено любое трехразрядное значение. На рис. 3.1 это означает, что с каждым из этих 2^N значений может быть связана амплитуда и относительная фаза. В трехкубитном примере амплитуда каждого круга определяет вероятность того, что при чтении всех трех кубитов будет получено конкретное трехразрядное значение.

Возможно, вас интересует, что собой представляет такая суперпозиция значения многокубитного регистра в отношении состояний отдельных образующих его кубитов. В некоторых случаях состояния отдельных кубитов легко выводятся. Например, суперпозиция трехкубитного регистра для со-

стояний $|0\rangle$, $|2\rangle$, $|4\rangle$, $|6\rangle$ на рис. 3.2 легко выражается состояниями всех отдельных кубитов.

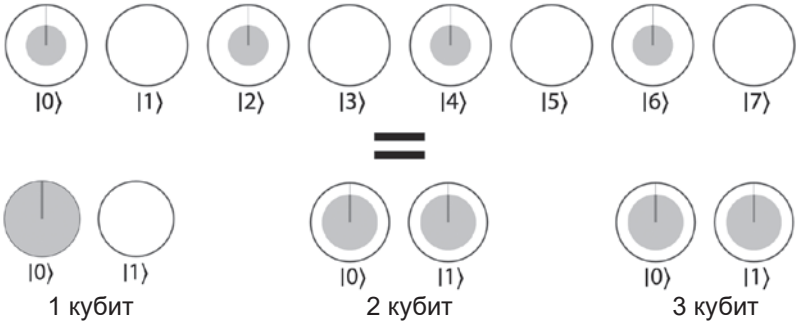


Рис. 3.2. Некоторые многокубитные квантовые состояния легко выражаются в однокубитных состояниях

Чтобы убедиться в том, что однокубитные и многокубитные представления эквивалентны, запишите каждое десятичное значение многокубитного состояния в форме трехразрядных двоичных значений. Заметим, что это многокубитное состояние может быть легко сгенерировано двумя однокубитными операциями HAD, как показано в листинге 3.1.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=3-1.

Листинг 3.1. Создание многокубитного состояния, которое может быть выражено составляющими кубитами

```
qc.reset(3);
qc.write(0);
var qubit1 = qint.new(1, 'qubit 1');
var qubit2 = qint.new(1, 'qubit 2');
var qubit3 = qint.new(1, 'qubit 3');
qubit2.had();
qubit3.had();
```

И хотя состояние на рис. 3.2 можно понять в выражении составляющих его кубитов, взгляните на трехкубитный регистр на рис. 3.3.

Рисунок представляет состояние трех кубитов в равной суперпозиции $|0\rangle$ и $|7\rangle$. Можно ли наглядно представить ситуацию в контексте того, что

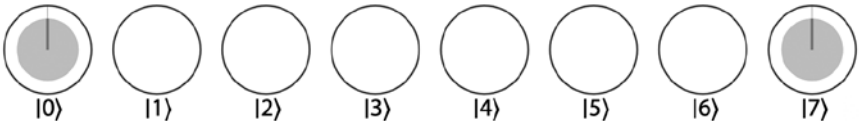


Рис. 3.3. Квантовые связи между несколькими кубитами

делает каждый отдельный бит, как на рис. 3.2? Так как значения 0 и 7 записываются в форме 000 и 111 в двоичном виде, значит, имеется суперпозиция трех кубитов в состояниях $|0\rangle|0\rangle|0\rangle$ и $|1\rangle|1\rangle|1\rangle$. Как ни странно, в данном случае записать круговые представления отдельных кубитов не удастся! Следует заметить, что при чтении этих трех кубитов они всегда будет иметь одинаковые значения (с 50%-ной вероятностью это будет значение 0, и с 50%-ной вероятностью — 1). Очевидно, между этими тремя кубитами должна существовать некоторая связь, которая обеспечивает равенство их результатов.

В листинге 3.1 представлены новые элементы синтаксиса QCEngine для отслеживания большего количества кубитов. Объект `qint` позволяет пометить кубиты и интерпретировать их таким образом, который в большей степени напоминает переменные из традиционного программирования. После того как мы используем `qc.reset()` для подготовки регистра с несколькими кубитами, `qint.new()` позволяет присвоить их объектам `qint`. Первый аргумент `qint.new()` указывает, сколько кубитов следует присвоить `qint` из стека, созданного вызовом `qc.reset()`. Второй аргумент получает метку, которая используется в схемной визуализации. Объекты `qint` содержат множество методов, позволяющих применять операции QPU непосредственно к группам кубитов. В листинге 3.1 используется метод `qint.had()`.

Эта связь — новый феномен квантовой *запутанности*. Запутанные многокубитные состояния невозможно определить через описания того, что делают отдельные составляющие кубиты (хотя можете попробовать!). Эта связь описывается только в конфигурации *всего* многокубитного регистра. Также выясняется, что запутанные состояния невозможно создать только из *однокубитных* операций. Чтобы исследовать запутанность более подробно, придется познакомиться с многокубитными операциями.

Изображение многокубитного регистра

Вы знаете, как описать конфигурацию N кубитов в круговой записи с использованием 2^N кругов, но как нарисовать многокубитные квантовые схемы? Наша многокубитная круговая запись предполагает, что каждый

кубит занимает свою позицию в битовой строке длины *N*, поэтому будет удобно пометить каждый кубит в соответствии с его двоичным значением.

Для примера снова взглянем на случайную восьмикубитную схему, представленную в главе 2. Регистр из восьми кубитов можно назвать *кубайтом* по аналогии с традиционным восьмиразрядным байтом. В нашем предыдущем знакомстве с кубайтом восемь кубитов были просто обозначены как кубит 1, кубит 2 и т. д. На рис. 3.4 показано, как будет выглядеть эта схема, если пометить каждый кубит тем двоичным значением, которое он представляет.

Рис. 3.4. Пометка кубитов в кубайте

На рис. 3.4 значения кубитов записаны в шестнадцатеричной системе, с использованием таких обозначений, как `0x1` и `0x2`. Это стандартный в среде программистов формат шестнадцатеричных значений, и мы будем использовать его в книге как удобный способ обозначений, поясняющий, когда речь идет о конкретном кубите — даже при большом количестве таких кубитов.

Однокубитные операции в многокубитных регистрах

Итак, теперь вы знаете, как рисовать многокубитные схемы и представлять их в круговой записи. Перейдем к их практическому применению. Что происходит (в круговой записи), когда мы применяем однокубитную операцию (например, NOT, HAD или PHASE) к многокубитному регистру?

Единственное отличие от однокубитного случая заключается в том, что операции с кругами выполняются в *операторных парах*, относящихся к кубиту, с которым выполняется операция.

Чтобы определить операторные пары для кубита, сопоставьте каждый круг с другим, значение которого отличается от него на битовую величину кубита, как показано на рис. 3.5. Например, если вы работаете с кубитом 0x4, то в каждую пару будут входить круги, значения которых отличаются ровно на 4.

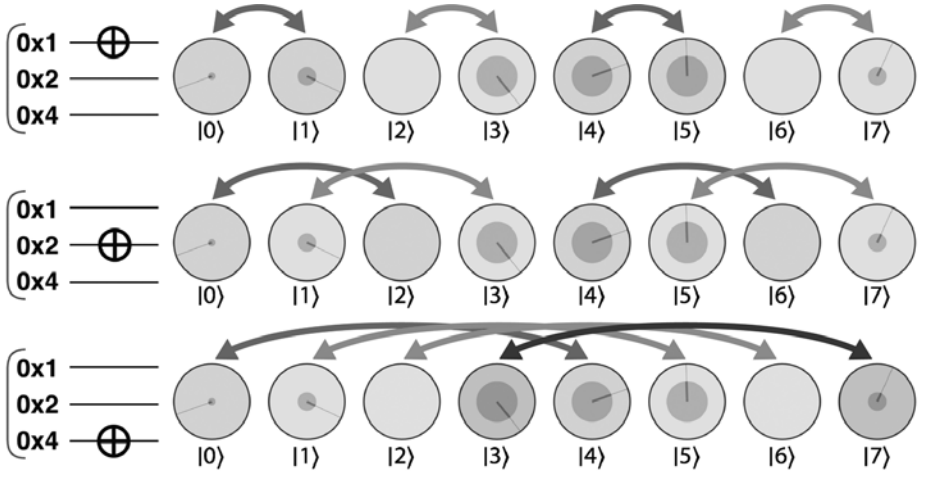


Рис. 3.5. Операция NOT меняет значения в каждой операторной паре кубитов

После того как вы определите операторные пары, операция выполняется с каждой парой так, как если бы компоненты пары были значениями $|0\rangle$ и $|1\rangle$ однокубитного регистра. Для операции NOT круги в паре просто меняются местами, как на рис. 3.5.

Для однокубитной операции PHASE правый круг каждой пары поворачивается на фазовый угол, как показано на рис. 3.6.

Рассматривая ситуацию в форме операторных пар, вы сможете быстро представить воздействие однокубитных операций на регистр. Чтобы лучше понять, почему этот способ работает, необходимо думать о том, как операция с заданным кубитом воздействует на двоичное представление всего регистра. Например, воздействие операции NOT (круги меняются местами) на второй кубит на рис. 3.5 соответствует простой смене состояния второго бита в двоичном представлении каждого значения. Аналогичным образом однокубитная операция PHASE, влияющая (например) на третий кубит,

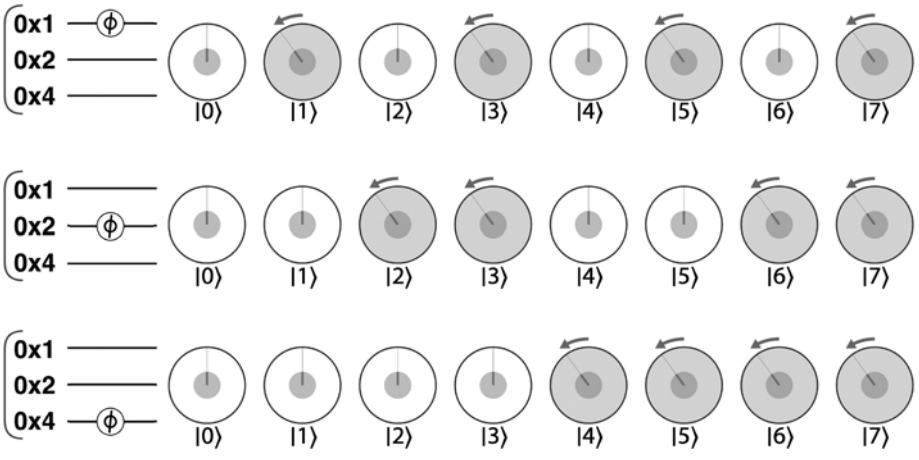


Рис. 3.6. Однокубитная фаза в многокубитном регистре

поворачивает каждый круг, у которого третий бит равен 1. Однокубитная операция PHASE всегда приводит к повороту ровно половины значений регистра, а какой именно половины — зависит от того, какой кубит был целью операции.

Аналогичные рассуждения помогают понять воздействие любой другой однокубитной операции на кубиты большего регистра.

Иногда мы будем выделять серым цветом некоторые круги в круговой записи, как на рис. 3.5 и 3.6. Это делается только для того, чтобы показать, какие состояния задействованы в операциях.

Чтение кубита в многокубитном регистре

Что произойдет при выполнении операции READ с одним кубитом из многокубитного регистра? Операция READ также работает с использованием операторных пар. Если вы используете многокубитное круговое представление, вероятность получения результата 0 для одного кубита можно вычислить суммированием квадратов магнитуд всех кругов в части $|0\rangle$ (левой) операторных пар этого кубита. Аналогичным образом вероятность получения 1 вычисляется суммированием квадратов магнитуд всех кругов в части $|1\rangle$ (правой) операторных пар этого кубита.

После выполнения операции READ состояние многокубитного регистра изменится и будет отражать полученный результат. Все круги, не согласующиеся с результатом, будут исключены, как показано на рис. 3.7.

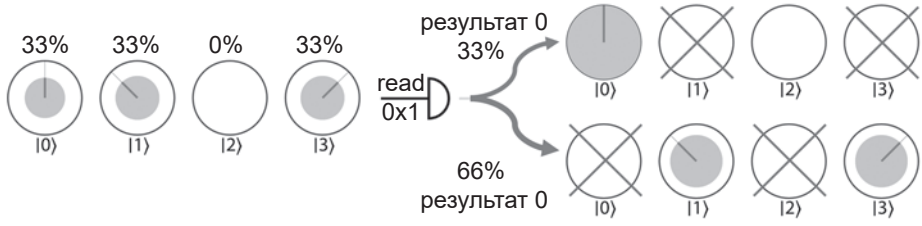


Рис. 3.7. Чтение одного кубита в многокубитном регистре

Заметим, что оба состояния $|1\rangle$ и $|3\rangle$ совместимы с чтением результата 1 в первом (0x1) кубите. Дело в том, что двоичные представления как 1, так и 3 содержат 1 в первом бите. Также обратите внимание на то, что после исключения состояние оставшихся значений перенормализируется, так что их площади (а следовательно, и соответствующие вероятности) в сумме дают 100%. Для чтения более одного кубита каждая однокубитная операция READ может быть выполнена отдельно в соответствии с правилом операторной пары.

Наглядное представление большего количества кубитов

Для N кубитов круговая запись должна содержать 2^N кругов; следовательно, каждый новый кубит, добавляемый в QPU, удваивает количество кругов, за которыми приходится следить. Как видно из рис. 3.8, это число стремительно растет до момента, когда круги в записи становятся едва заметными.

При таком количестве крошечных кругов представление круговой записи лучше подходит для выявления закономерностей, а не отдельных значений; любую область, которую потребуется изучить более подробно, можно увеличить. Тем не менее даже в таких ситуациях можно прояснить картину, выделив относительные фазы, увеличив толщину линии, обозначающей поворот круга, или выделяя различия в фазе разными цветами или интенсивностями, как на рис. 3.9.

В последующих главах мы будем пользоваться этими приемами. Но даже такие «визуальные трюки» работают только до определенного момента; в представлении 32-кубитной системы будет задействовано 4 294 967 296 кругов — слишком много информации как для большинства экранов, так и для человеческого глаза.

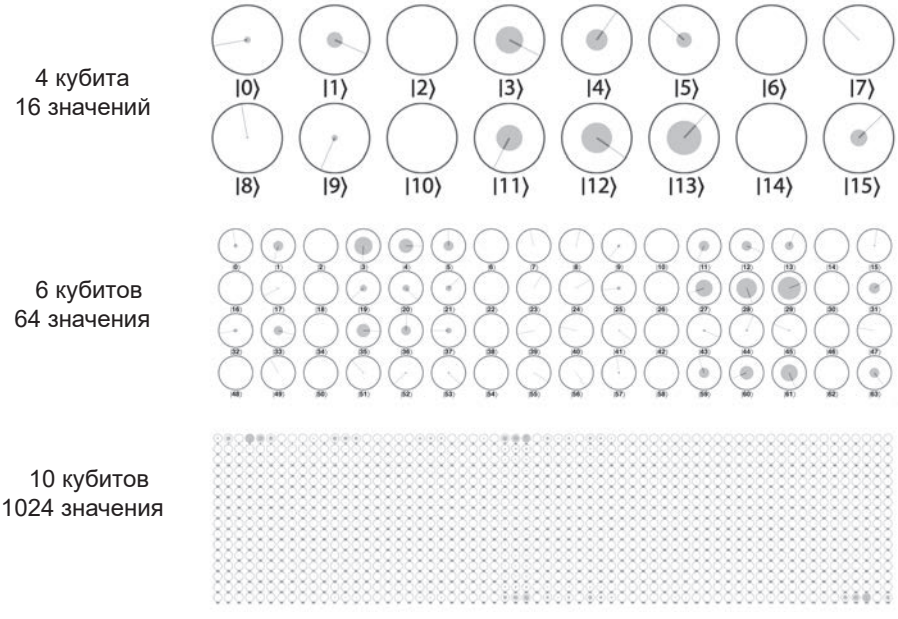


Рис. 3.8. Круговая запись при большом количестве кубитов

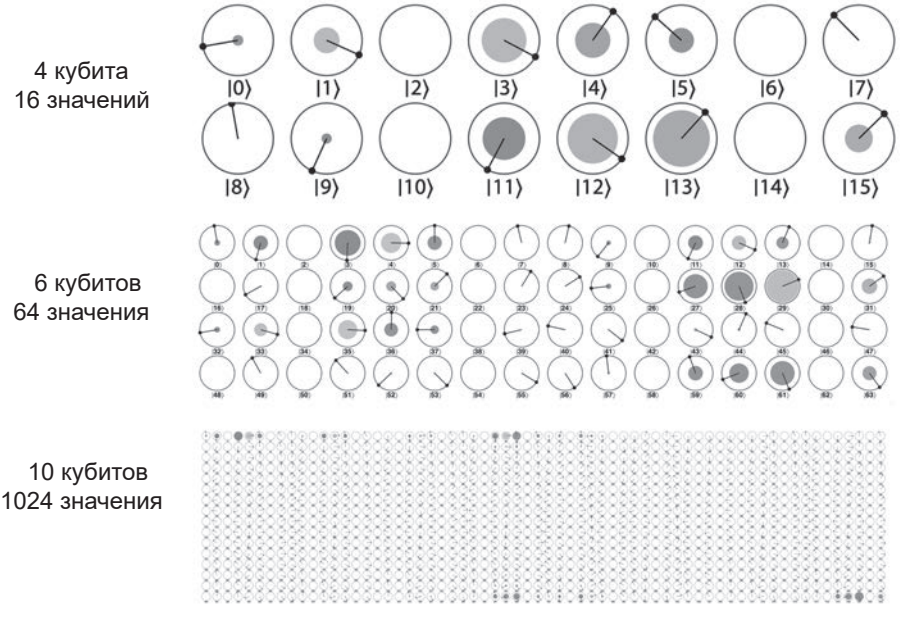
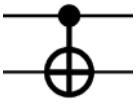


Рис. 3.9. Иногда без преувеличения не обойтись

Вы можете добавить в самое начало своих программ QCEngine команду `qc_options.color_by_phase = true;`, чтобы включить режим цветового выделения фазы на рис. 3.9. Команда `qc_options.book_render = true;` управляет режимом повышенной толщины фазовых линий.

Команды QPU: CNOT



Пришло время представить некоторые команды QPU, многокубитовые по своей природе. Под многокубитными операциями мы подразумеваем операции, для выполнения которых *требуется* более одного кубита. Начнем с замечательной операции CNOT. Операция CNOT работает с *двумя* кубитами и может рассматриваться как программная конструкция «if» со следующим условием: «*Применить операцию NOT к целевому кубиту, но только в том случае, если управляющий кубит содержит значение 1*». Для обозначения этой логики условный знак CNOT соединяет два кубита линией. Заполненная точка представляет управляющий кубит, а знак NOT обозначает целевой кубит, с которым выполняется условная операция[1].

Идея использования *управляющих кубитов* для избирательного применения действий встречается во многих операциях QPU, но CNOT, пожалуй, является классическим примером. На рис. 3.10 показаны различия между применением операции NOT к кубиту в регистре и применением операции CNOT (управляемой по другому кубиту).

Стрелки на рис. 3.10 показывают, у каких операторных пар круги меняются местами в круговой записи. Из рисунка видно, что основная операция

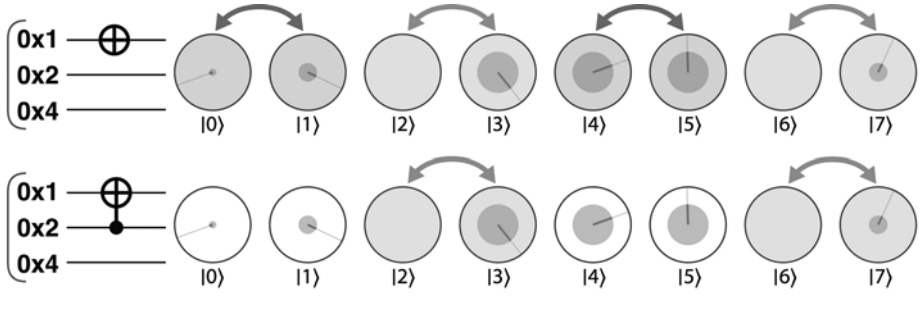


Рис. 3.10. Операции NOT и CNOT в действии

[1] Операциями также можно управлять по значению 0. Для этого достаточно добавить к управляющему регистру пару вентилей NOT: до и после операции.

CNOT идентична операции NOT, но выполняется более избирательно — операция NOT применяется только к значениям, двоичные представления которых (в данном примере) содержат 1 во втором бите (010=2, 011=3, 110=6 и 111=7).

Обратимость: операция CNOT, как и NOT, обратима по отношению к самой себе — повторное применение CNOT возвращает многокубитный регистр в исходное состояние.

В самой операции CNOT нет ничего особенно квантового; конечно, условная логика принадлежит к числу фундаментальных возможностей современных процессоров. Однако при помощи CNOT можно задать один интересный и определенно квантовый вопрос: что произойдет, если управляющий кубит в операции CNOT находится в суперпозиции, как на рис. 3.11?

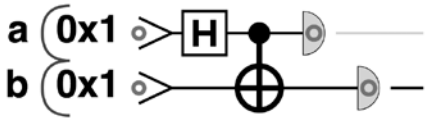


Рис. 3.11. CNOT с управляющим кубитом в суперпозиции

Для удобства пользования мы временно пометили два кубита, *a* и *b* (вместо шестнадцатеричных обозначений). Пройдем по схеме, начиная с регистра в состоянии $|0\rangle$, и посмотрим, что происходит. На рис. 3.12 изображена схема и ее состояние в начале программы, перед выполнением команд.

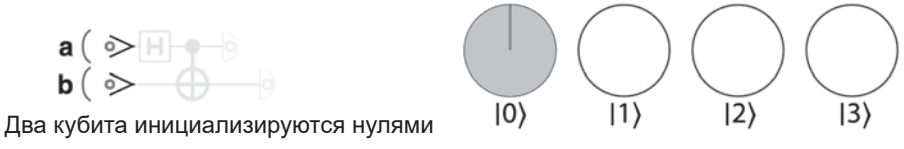


Рис. 3.12. Пара Белла, шаг 1

Сначала операция HAD применяется к кубиту *a*. Так как *a* является кубитом с наименьшим весом в регистре, будет создана суперпозиция значений $|0\rangle$ и $|1\rangle$, как показано на рис. 3.13 в круговой записи.

Затем применяется операция CNOT, чтобы состояние кубита *b* изменялось условно в зависимости от состояния кубита *a*, как показано на рис. 3.14.

Результатом является суперпозиция $|0\rangle$ и $|3\rangle$. И это логично, потому что если кубит *a* принял значение $|0\rangle$, то с кубитом *b* ничего не произойдет, и он останется в состоянии $|0\rangle$ — регистр остается в итоговом состоянии

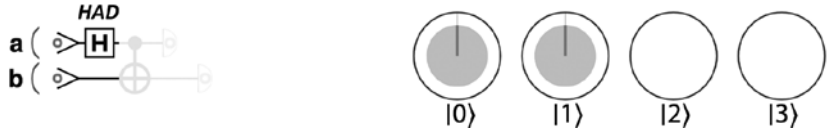


Рис. 3.13. Пара Белла, шаг 2

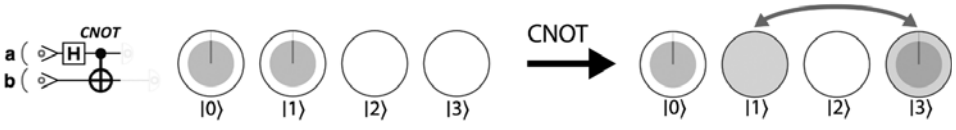


Рис. 3.14. Пара Белла, шаг 3

$|0\rangle|0\rangle=|0\rangle$. Но если кубит *a* находился в состоянии $|1\rangle$, то операция NOT будет применена к *b*, и регистр получит значение $|1\rangle|1\rangle = |3\rangle$.

Чтобы понять операцию CNOT на рис. 3.14, можно взглянуть на происходящее иначе: считайте, что она просто следует правилу круговой записи CNOT, которое подразумевает, что состояния $|1\rangle$ и $|3\rangle$ меняются местами (как показывают стрелки на рис. 3.14). Просто в данном случае один из этих кругов оказывается в суперпозиции.

Оказывается, результат на рис. 3.14 открывает исключительно полезные возможности. Собственно, он является однокубитным эквивалентом состояния квантовой запутанности, впервые представленного на рис. 3.3. Ранее уже упоминалось о том, что в запутанном состоянии проявляется своего рода взаимозависимость между кубитами — при чтении двух кубитов, показанных на рис. 3.14, результаты будут случайными, но они всегда будут согласовываться (то есть 00 или 11 с 50%-ной вероятностью каждого варианта).

Единственным исключением из правила «избегать физики любой ценой», которому мы следовали до настоящего момента, было чуть более глубокое описание суперпозиции. Явление квантовой запутанности не менее важно, поэтому мы — в последний раз — отвлечемся на физику, чтобы вы лучше поняли, *почему* квантовая запутанность открывает такие замечательные возможности[1].

Если вы предпочитаете примеры кода физической теории, вы можете спокойно пропустить следующую пару абзацев без особых последствий.

[1] Правда, это сто процентов последнее упоминание физики в книге. Честное слово.

Возможно, квантовая запутанность не покажется вам чем-то необычным. Конечно, согласованность между значениями традиционных битов не является причиной для беспокойства, даже если они *случайным образом* принимают коррелированные значения. В традиционных вычислениях, если два случайных бита всегда выдают согласующиеся значения при чтении, существует две совершенно заурядные возможности:

1. Некий механизм в прошлом приравнял их значения и наделил их общей отправной точкой. Тогда их «случайность» в действительности иллюзорна.
2. Два бита принимают случайные значения непосредственно в момент чтения, но они способны как-то взаимодействовать друг с другом и согласовывать свои значения.

Если немного поразмыслить, становится ясно, что *только* эти две причины могут объяснять согласованность случайности для двух традиционных битов.

Изящный эксперимент, предложенный великим ирландским физиком Джоном Беллом, убедительно демонстрирует, что с квантовой запутанностью такое согласование становится возможным и без этих двух разумных объяснений! Именно в этом смысле иногда говорят, что запутанность представляет собой сугубо квантовую форму связи между кубитами — *сильнее, чем можно было бы достичь традиционными средствами*. Когда вы начнете программировать более сложные приложения для QPU, запутанность будет проявляться сплошь и рядом. Чтобы пользоваться ею на практике, все эти философские размышления вам не понадобятся, однако всегда бывает полезно представить, что же происходит «под капотом».

Практический пример: использование пар Белла для реализации совместной случайности

Состояние запутанности, созданное на рис. 3.14, часто называют парой Белла[1]. Рассмотрим простой и быстрый способ практического применения связи, существующей в этих состояниях.

В главе 2 мы выяснили, что суперпозиция отдельного кубита позволяет реализовать квантовый генератор случайных чисел. Аналогичным образом

[1] Это состояние (а также ряд других) называется так потому, что оно было использовано Джоном Беллом в ходе демонстрации непостижимой корреляции запутанных состояний.

чтение пары Белла действует как КГСЧ, только на этот раз мы получаем *согласующиеся* случайные значения двух кубитов.

У квантовой запутанности есть одно удивительное свойство: кубиты сохраняют запутанность, на какое бы расстояние вы их ни разнесли. Таким образом, пары Белла могут легко использоваться для генерирования *согласованных случайных битов* в разных местах. Такие биты могут стать основой для установки безопасной совместной случайности — механизма, критичного для современного интернета.

Фрагмент кода в листинге 3.2 реализует эту идею: он генерирует совместную случайность, создавая пару Белла с последующим чтением значения из каждого кубита, как показано на рис. 3.15.

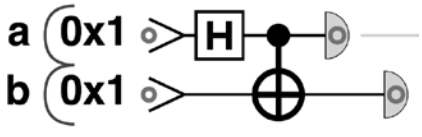


Рис. 3.15. Схема пары Белла

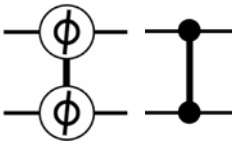
Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=3-2.

Листинг 3.2. Создание пары Белла

```
qc.reset(2);
var a = qint.new(1, 'a');
var b = qint.new(1, 'b');
qc.write(0);
a.had();           // Перевести в суперпозицию
b.cnot(a);         // Запутанность
var a_result = a.read();
var b_result = b.read();
qc.print(a_result);
qc.print(b_result);
```

Команды QPU: CPHASE и CZ



Другая чрезвычайно распространенная двухкубитная операция — CPHASE(θ). Как и CNOT, операция CPHASE задействует своего рода условную логику, порождающую квантовую запутанность. Как было

показано на рис. 3.6, однокубитная операция PHASE(θ) воздействует на регистр, поворачивая (на угол θ) значения $|1\rangle$ в операторных парах этого кубита. По аналогии с парой CNOT/NOT, CPHASE ограничивает воздействие на целевой кубит и выполняет его только в том случае, если другой управляющий кубит принимает значение $|1\rangle$. Следует заметить, что операция CPHASE действует только тогда, когда управляющий кубит содержит $|1\rangle$, и воздействует только на состояния целевого кубита со значением $|1\rangle$. Это означает, что применение CPHASE(θ), скажем, к кубитам 0x1 и 0x4 приведет к повороту (на угол θ) всех кругов, для которых эти оба кубита имеют значение $|1\rangle$. Из-за этой особенности CPHASE обладает симметрией между входными значениями, которая отсутствует у CNOT. В отличие от большинства других управляемых операций, для CPHASE не существенно, какой кубит считается целевым, а какой — управляющим.

На рис. 3.16 воздействие операции CPHASE на кубиты 0x1 и 0x4 сравнивается с воздействием отдельных операций PHASE с этими кубитами.

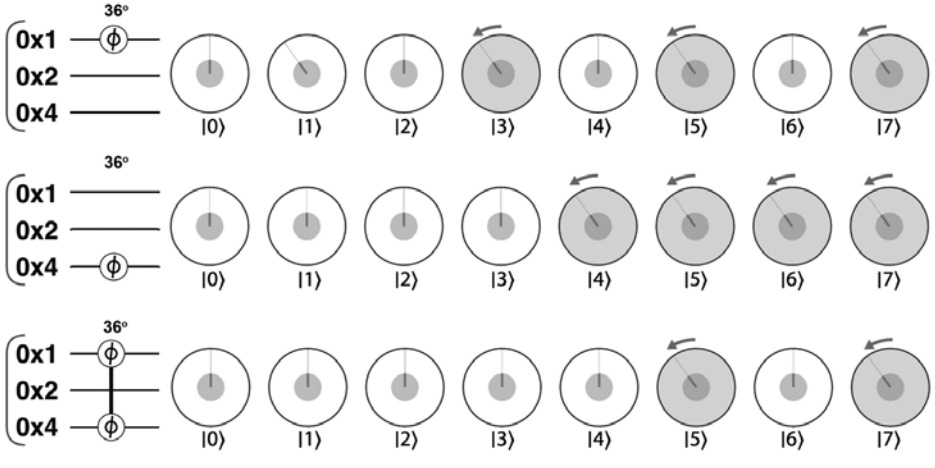


Рис. 3.16. Применение операции CPHASE в круговой записи

Сами по себе однокубитные операции PHASE поворачивают относительные фазы *половины* кругов, связанных с регистром QPU. Добавление условия сокращает количество повернутых кругов вдвое. Можно и дальше добавлять условные кубиты в CPHASE, каждый раз сокращая количество задействованных значений. В общем случае чем больше условий управляет операциями QPU, тем более избирательно можно подходить к тому, на какие значения регистра QPU она воздействует.

В программах QPU часто применяется операция CPHASE(θ) с фазой θ = 180°; эта конкретная реализация CPHASE получила собственное название CZ и упрощенное обозначение, показанное на рис. 3.17. Интересно, что операция CZ очень просто строится из операций HAD и CNOT. Как было показано на рис. 2.14, операция phase(180) (Z) может быть построена из двух операций HAD и NOT. Аналогичным образом операция CZ может быть построена из двух операций HAD и CNOT (рис. 3.17).

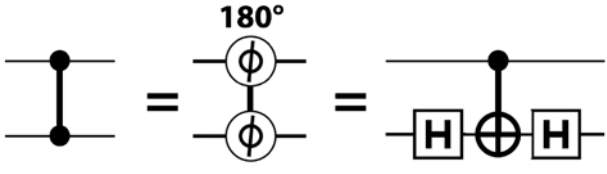


Рис. 3.17. Три представления CPHASE(180)

Приемы программирования QPU: фазовый откат

Если задуматься об изменении фазы одного регистра QPU, обусловленном значениями кубита некоторого другого регистра, можно прийти к удивительному и полезному эффекту, называемому *фазовой коррекцией*. Взгляните на схему на рис. 3.18.

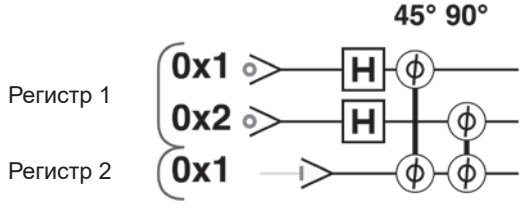


Рис. 3.18. Схема, демонстрирующая прием фазовой коррекции

Эту схему можно рассматривать следующим образом: после перевода регистра 1 в суперпозицию всех $2^2 = 4$ возможных значений мы выполняем поворот фазы регистра 2, который управляется значениями, принимаемыми кубитами регистра 1. Рассматривая полученные индивидуальные состояния обоих регистров в круговой записи, мы видим, что с состоянием регистра 1 произошло нечто интересное (рис. 3.19).

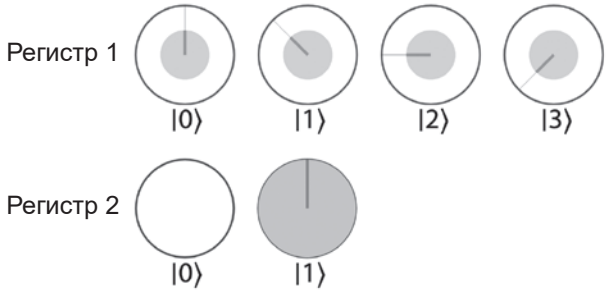


Рис. 3.19. Состояния обоих регистров, задействованных в фазовой коррекции

Повороты фазы, которые мы попытались применить ко второму регистру (управляемые кубитами первого регистра), также повлияли на различные значения первого регистра! А конкретнее, в приведенных представлениях круговой записи происходит следующее.

Поворот на 45° регистра 2, управляемый кубитом с наименьшим весом из регистра 1, также приводит к повороту каждого значения в регистре 1, для которого активируется этот кубит с наименьшим значением ($|1\rangle$ и $|3\rangle$).

Поворот на 90° регистра 2, управляемый кубитом с наибольшим весом из регистра 1, также отражается на всех значениях регистра 1, для которого активируется кубит с наибольшим весом (значения $|2\rangle$ и $|3\rangle$).

В результате комбинация этих поворотов фазы, выполненных с изначальной целью, то есть регистром 2, отражается на регистре 1. Обратите внимание: поскольку регистр 2 не находится в суперпозиции, его (глобальная) фаза остается неизменной.

Коррекция фазы — чрезвычайно полезная концепция, так как она может использоваться для применения поворота фазы к конкретным значениям в регистре (регистр 1 в приведенном примере). Это можно сделать выполнением поворота фазы с другим регистром, управляемого кубитами из регистра, который нас действительно интересует. Эти кубиты можно выбрать специально для выбора значений, которые требуется повернуть.

Чтобы фазовая коррекция работала, второй регистр всегда должен быть инициализирован значением $|1\rangle$. Хотя мы применяем двухкубитную операцию между двумя регистрами, никакая запутанность при этом не создается; следовательно, состояние может быть полностью представлено в виде отдельных регистров. Двухкубитные вентили не всегда порождают квантовую запутанность между регистрами; объяснения будут приведены в главе 14.

Если фазовая коррекция на первых порах кажется вам чем-то заумным, вы не одиноки — к ней привыкаешь не сразу. Чтобы разобраться в происходящем, лучше всего поэкспериментировать с примерами — это поможет сформировать интуитивное представление о сути дела. В листинге 3.3 приведен код QCEngine, воспроизводящий пример с двумя регистрами.

Пример кода

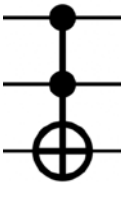
Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=3-3.

Листинг 3.3. Фазовая коррекция

```
qc.reset(3);
// Создать два регистра
var reg1 = qint.new(2, ‘Register 1’);
var reg2 = qint.new(1, 'Register 2');
reg1.write(0);
reg2.write(1);
// Перевести первый регистр в суперпозицию
reg1.had();
// Выполнить со вторым регистром поворот фазы,
// управляемый кубитами первого регистра
qc.phase(45, 0x4, 0x1);
qc.phase(90, 0x4, 0x2);
```

Коррекция фазы сильно пригодится нам в главе 8 для понимания внутреннего устройства QPU-примитива квантовой оценки фазы, а затем еще в главе 13 для объяснения того, как QPU применяется при решении систем линейных уравнений. Польза фазовой коррекции происходит от того факта, что она работает не только с операциями CPHASE, но и с любыми условными операциями, порождающими изменения в фазе регистра. Даже один этот факт поможет нам понять, как строятся более общие условные операции.

Команда QPU: CCNOT (вентиль Тоффоли)



Как упоминалось ранее, многокубитные условные операции можно сделать более избирательными за счет выполнения операций, управляемых несколькими кубитами. Давайте посмотрим, как это делается, и обобщим операцию CNOT добавлением новых условий. Операция CNOT с двумя управляющими кубитами часто называется операцией *CCNOT*. Также CCNOT иногда

называют *вентилем Тоффоли* по названию одноименного аналогичного вентиля из традиционных вычислений.

С добавлением каждого условия операция NOT остается прежней, но количество операторных пар, задействованных в круговой записи регистра, сокращается вдвое. Этот факт продемонстрирован на рис. 3.20, где операция NOT с первым кубитом трехкубитного регистра сравнивается с соответствующими операциями CNOT и CCNOT.

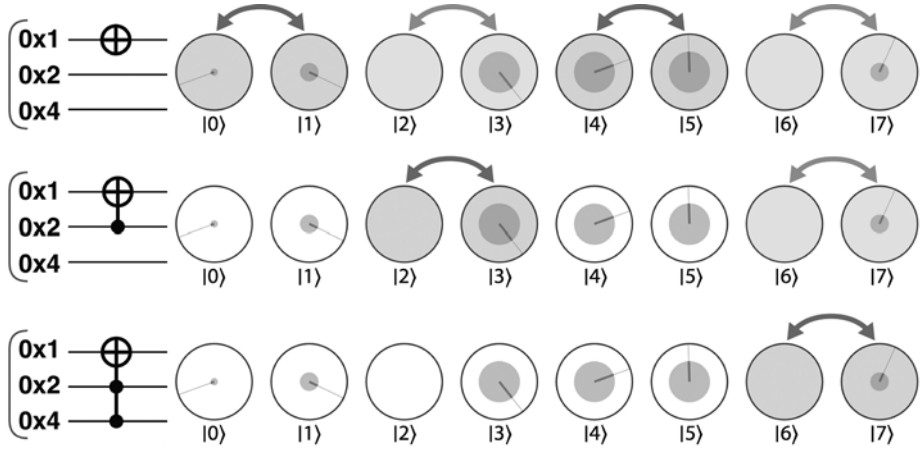
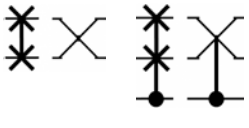


Рис. 3.20. Добавление условий повышает избирательность операций NOT

В определенном смысле операция CCNOT может интерпретироваться как операция, реализующая условие «Если A AND B, то переключить C». Пожалуй, для реализации базовой логики вентиль CCNOT является самой полезной операцией QPU. Объединение и каскадное связывание нескольких вентилей CCNOT позволяет реализовать широкий набор логических функций, как будет показано в главе 5.

Команды QPU: SWAP и CSWAP



Еще одна чрезвычайно распространенная операция в квантовых вычислениях — SWAP (также называемая *операцией обмена*) — просто меняет местами два кубита. Если архитектура QPU предоставляет такую возможность, операция SWAP может стать по-настоящему фундаментальной, в которой происходит физическое перемещение объектов, представляющих кубиты. Также операция SWAP может быть выполнена посредством

обмена информацией, содержащейся в двух кубитах (вместо самих кубитов), с использованием трех операций CNOT (рис. 3.21).

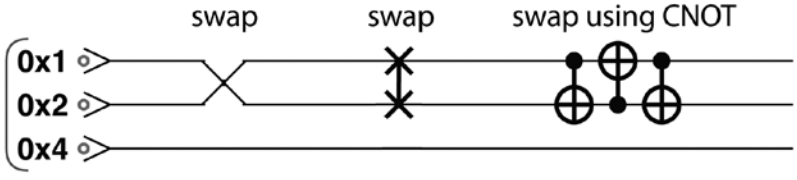


Рис. 3.21. Операция SWAP может быть реализована на базе CNOT

В этой книге мы используем два способа обозначения операций SWAP. В общем случае используется пара соединенных X. Если переставляемые кубиты соседствуют друг с другом, часто бывает проще и понятнее перекрестить линии кубитов. В обоих случаях операции идентичны.

Чем же так полезна операция SWAP, спросите вы? Почему бы просто не переименовать кубиты? В QPU потенциал операции SWAP проявляется тогда, когда мы рассматриваем ее обобщение в условную операцию CSWAP (условную операцию *обмена*). Операция CSWAP может быть реализована с использованием трех вентилей CCNOT (рис. 3.22).

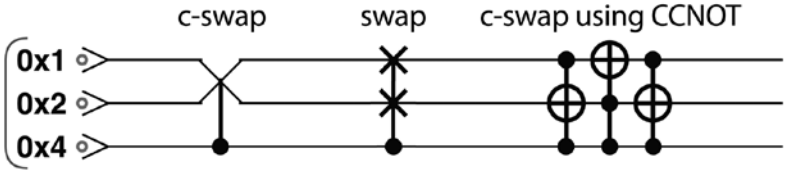


Рис. 3.22. Операция CSWAP, реализованная на базе CCNOT

Если управляющий кубит для операции CSWAP находится в суперпозиции, в результате меняются местами суперпозиции двух кубитов — а также не меняются. В главах 5 и 12 вы увидите, что эта особенность CSWAP позволяет выполнять операцию умножения на 2 в квантовой суперпозиции.

Проверка обменом

На базе операций SWAP строится очень полезная схема, называемая схемой *проверки обменом*. Схема проверки обменом решает следующую задачу: даны два кубитных регистра; как определить, находятся ли они в одном

состоянии? К настоящему моменту вы уже отлично знаете, что в общем случае (если хотя бы один регистр находится в суперпозиции) невозможно использовать разрушающую операцию READ для полного определения состояния каждого регистра с целью проведения сравнения. Операция SWAP действует хитрее. Не сообщая нам, что это за состояния, она просто позволяет определить, равны они или нет.

В ситуации, в которой невозможно точно определить содержимое выходного регистра, проверка обменом может стать бесценным инструментом. На рис. 3.23 изображена схема реализации проверки обменом из листинга 3.4.

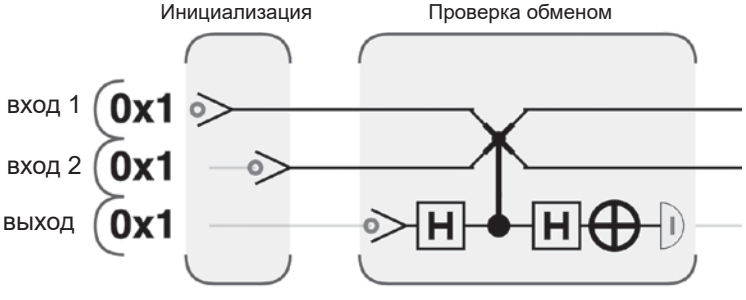


Рис. 3.23. Использование проверки обменом для определения того, находятся ли два регистра в одном состоянии

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=3-4.

Листинг 3.4. Проверка обменом

```
// В этом примере проверка обменом должна проверить
// равенство двух входных состояний
qc.reset(3);
var input1 = qint.new(1, 'input1');
var input2 = qint.new(1, 'input2');
var output = qint.new(1, 'output');

// Инициализировать любыми проверяемыми состояниями
input1.write(0);
input2.write(0);

// Сама проверка обменом
output.write(0);
```

```
output.had();
// Выполнить условный обмен двух входов, управляемый выходными кубитами
input1.exchange(input2, 0x1, output.bits());
output.had();
output.not();
var result = output.read();
// Результат равен 1, если входы равны
```

В листинге 3.4 проверка обменом используется для сравнения состояний двух однокубитных регистров, но та же схема легко расширяется для сравнения многокубитных регистров. Результат проверки обменом определяется при чтении дополнительного однокубитного выходного регистра (размеры входных регистров могут быть любыми; выходной регистр в любом случае остается однокубитным). Изменяя линии `input1.write(0)` и `input2.write(0)`, можно поэкспериментировать со входами, отличающимися в разных степенях. Как выясняется, в случае равенства двух входных состояний выходной регистр всегда находится в состоянии $|1\rangle$, поэтому при чтении этого регистра операцией READ всегда определенно будет получен результат 1. Однако по мере того, как два входных значения различаются все сильнее, вероятность чтения результата 1 из выходного регистра уменьшается и в конечном итоге достигает 50% в том случае, если `input1` содержит $|0\rangle$, а `input2` — $|1\rangle$. На рис. 3.24 точно показано, как изменяется вероятность результата в регистре `output` с ростом степени совпадения между двумя входными регистрами.

По оси x на рисунке измеряется числовая метрика расхождений между состояниями двух регистров, называемая *степенью совпадения* (fidelity). Мы не будем углубляться в математические обоснования того, как вычисляется степень совпадения между состояниями регистров QPU, но она математически инкапсулирует способ сравнения двух суперпозиций.

Вы можете несколько раз выполнить схему проверки обменом и проследить за полученными результатами. Чем для большего количества запусков будет наблюдаться результат 1, тем сильнее будет ваша уверенность в том, что два входных состояния идентичны. Точное количество повторений проверки обменом зависит от того, в какой степени вы хотите быть уверены в идентичности двух входов и до какой степени они должны быть близки для того, чтобы считаться идентичными. На рис. 3.25 показана нижняя граница количества проверок обменом, в которых необходимо наблюдать результат 1 для достижения 99% уверенности в идентичности входов. Ось y показывает, как это количество изменяется при ослаблении требований к идентичности входов (измеряемой по оси x). Обратите внимание: в момент получения результата 0 при проверке обменом вы мо-

жете быть полностью уверены в том, что два входных состояния не идентичны[1].

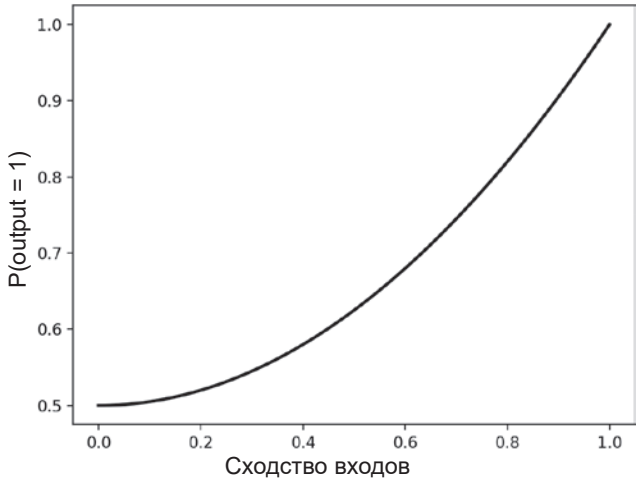


Рис. 3.24. Зависимость результата проверки обменом от степени совпадения входных состояний

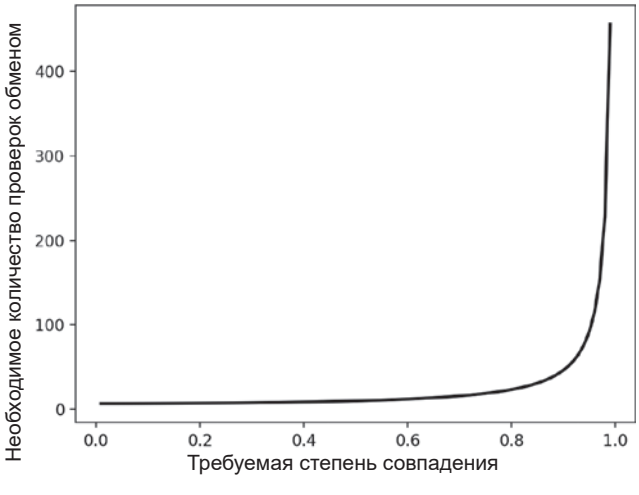


Рис. 3.25. Количество проверок обменом, которые должны вернуть результат 1 для достижения 99% уверенности в идентичности входов

[1] В реальности было бы желательно разрешить отдельные вхождения 0, если вас действительно устраивают близкие, но не идентичные состояния, а также, возможно, с учетом возможности ошибок при вычислениях. В приведенном простом анализе мы проигнорировали эти факторы.

Вместо того чтобы рассматривать проверку обменом как способ подтверждения равенства двух состояний с результатом «да/нет», также можно использовать другую полезную интерпретацию: обратите внимание на то, как на рис. 3.25 показано, что вероятность получения результата 1 является метрикой идентичности двух входов (то есть их степени сходства). Если повторить проверку обменом достаточное количество раз, можно оценить эту вероятность, а следовательно, и степень сходства — более количественную метрику близости двух состояний. Точная оценка того, насколько близки два квантовых состояния продемонстрированным способом, весьма полезна в квантовых приложениях машинного обучения.

Построение произвольной условной операции

Мы представили операции CNOT и CPHASE, но существуют ли такие операции, как CHAD (условная операция HAD) или CRNOT (условная операция RNOT)? Да, существуют! Даже если условная версия некоторой однокубитной операции не входит в набор команд конкретного QPU, существует процесс, позволяющий «преобразовать» однокубитные операции в многокубитные условные операции.

Общий рецепт преобразования однокубитных операций требует большего объема математики, чем мы хотим привести, но рассмотрение примера поможет вам более уверенно чувствовать себя в этом процессе. Ключевая идея заключается в том, что однокубитная операция разбивается на меньшие шаги. Как выясняется, однокубитную операцию всегда можно разбить на набор шагов так, чтобы использовать CNOT для условной *отмены* операций. В итоге вы можете условно выбирать, нужно ли применять ее эффект.

Сказанное гораздо проще пояснить на простом примере. Допустим, вы пишете программу для QPU, который может выполнять операции CNOT, CZ и PHASE, но не содержит команды для операции CPHASE. В действительности такая ситуация весьма типична для современных QPU, но, к счастью, мы можем легко построить собственную реализацию CPHASE из имеющихся ингредиентов.

Вспомните, что двухкубитная операция CPHASE должна поворачивать фазу любого значения регистра, в котором *оба* кубита принимают значение $|1\rangle$. Это показано на рис. 3.26 для случая PHASE(90).

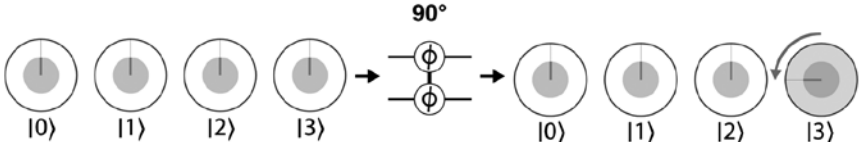


Рис. 3.26. Желательный результат операции CPHASE(90)

Операцию PHASE(90) можно легко разбить на меньшие составляющие с поворотами на меньшие углы. Например, PHASE(90)=PHASE(45)PHASE(45). Также можно «отменить» поворот, выполняя его в обратном направлении; например, операция PHASE(45)PHASE(–45) равносильна тому, что с кубитом не делается ничего. Вооружившись этими фактами, можно построить операцию CPHASE(90) на рис. 3.26 с использованием операций, показанных на рис. 3.27 и в листинге 3.5.

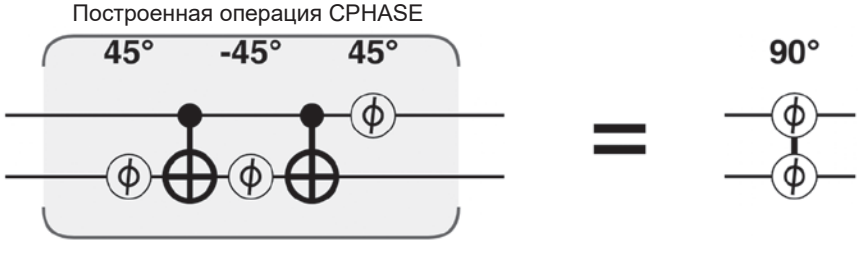


Рис. 3.27. Построение операции CPHASE

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=3-5.

Листинг 3.5. Самостоятельная реализация условного поворота фазы

```
var theta = 90;

qc.reset(2);
qc.write(0);
qc.hadamard();

// Использовать две операции CNOT и три PHASE...
qc.phase( theta / 2, 0x2);
qc.cnot(0x2, 0x1);
qc.phase(-theta / 2, 0x2);
qc.cnot(0x2, 0x1);
```

```
qc.phase( theta / 2, 0x1);

// Та же операция в виде 2-кубитной CPHASE
qc.phase(theta, 0x1, 0x2);
```

Анализируя эффект операции для разных входных значений, мы видим, что она применяет PHASE(90) только в том случае, когда оба кубита содержат $|1\rangle$ (на входе $|1\rangle|1\rangle$). Чтобы убедиться в этом, полезно вспомнить, что PHASE не влияет на кубит в состоянии $|0\rangle$. Также полезно рассмотреть работу схемы на рис. 3.27 в круговой записи (рис. 3.28).

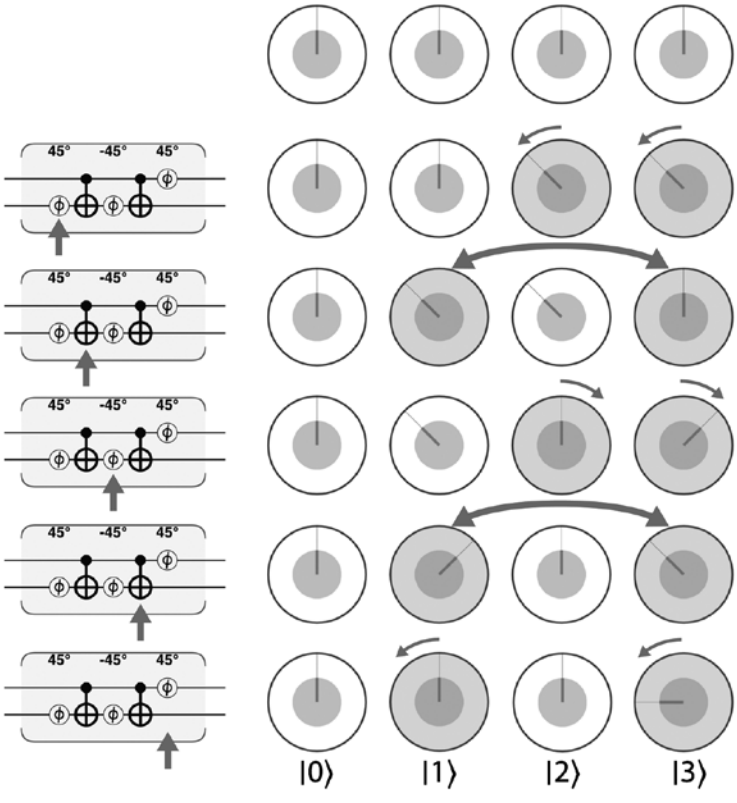


Рис. 3.28. Пошаговый анализ построенной операции CPHASE

Хотя эта схема правильно работает для реализации вентиля CPHASE(90), общий случай построения условной версии произвольной операции немного сложнее. За полным рецептом и объяснениями обращайтесь к главе 14.

Практический пример: дистанционно управляемая случайность

Взяв на вооружение многокубитные операции, можно исследовать некоторые интересные и неочевидные свойства квантовой запутанности на примере генерирования случайных чисел с дистанционным управлением. Программа генерирует два кубита, причем чтение одного влияет на вероятность получения случайного бита при чтении из другого. Более того, этот эффект проявляется при любых пространственных или временных промежутках. Эта невозможная на первый взгляд задача на удивление просто реализуется при использовании QPU.

Дистанционное управление работает следующим образом. Вы работаете с парой кубитов так, что чтение одного кубита (любого) возвращает случайный (50/50) бит с «измененной» вероятностью другого. Если результат равен 0, то при чтении другого кубита операцией `READ` с 15%-ной вероятностью будет получен результат 1. В противном случае, если прочитанный кубит возвращает 1, при чтении другого кубита с 85%-ной вероятностью будет получен результат 1.

Пример кода в листинге 3.6 демонстрирует, как реализуется такой «генератор случайных чисел с дистанционным управлением». Как и в листинге 3.1, для удобной работы с группами кубитов используются объекты QCEngine `qint`.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=3-6.

Листинг 3.6. Случайность с дистанционным управлением

```
qc.reset(2);
var a = qint.new(1, 'a');
var b = qint.new(1, 'b');
qc.write(0);
a.had();
// Вероятность для a составляет 50%
b.had();
b.phase(45);
b.had();
// Вероятность для b составляет 15%
b.cnot(a);
// Теперь вы можете прочитать *любой*
```

```
//  из кубитов с 50% вероятностью.
// Если результат равен 0, то вероятность
// для *другого* кубита равна 15%;
// в противном случае она составит 85%.
var a_result = a.read();
var b_result = b.read();
qc.print(a_result + ' ');
qc.print(b_result + '\n');
```

На рис. 3.29 эффект каждой операции в программе представлен в круговой записи.

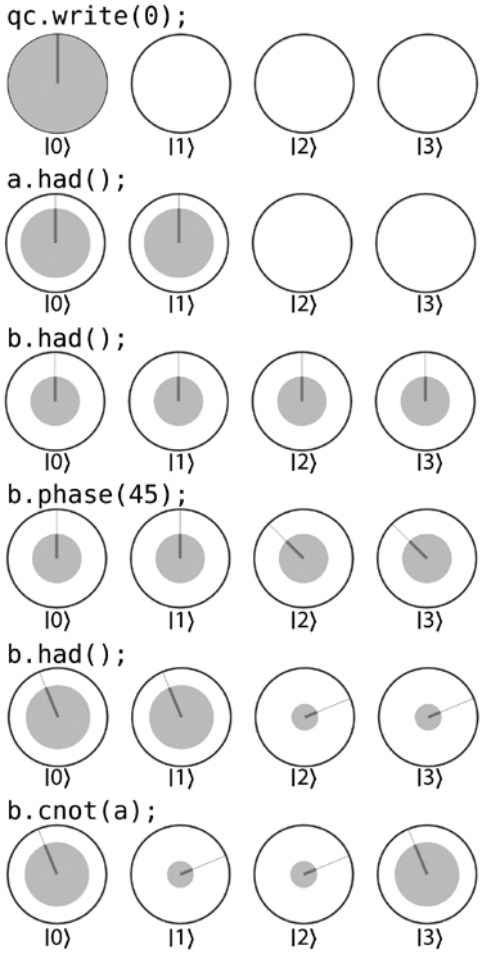


Рис. 3.29. Пошаговый анализ программы ГСЧ с дистанционным управлением в круговой записи

После завершения всех шагов мы получаем два кубита в состоянии запутанности. Рассмотрим, что произойдет при чтении кубита в состоянии на конец рис. 3.29. Если при чтении кубита *a* будет получено значение 0 (что происходит с 50%-ной вероятностью), то останутся только круги, совместимые с таким состоянием дел. Это значения $|0\rangle|0\rangle = |0\rangle$ и $|1\rangle|0\rangle = |2\rangle$, так что два кубита оказываются в состоянии, изображенном на рис. 3.30.

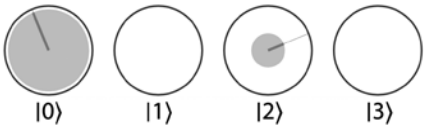


Рис. 3.30. Состояние одного кубита после чтения 0 из другого

Два значения в этом состоянии обладают ненулевыми вероятностями как для получения 0, так и для получения 1 при чтении кубита *b*. В частности, для кубита *b* вероятности прочитать 0/1 равны 70%/30%.

Предположим, что исходное чтение кубита *a* дало 1 (что также происходит с вероятностью 50%. В этом случае останутся только значения $|0\rangle|1\rangle=|1\rangle$ и $|1\rangle|1\rangle=|3\rangle$ (рис. 3.31).

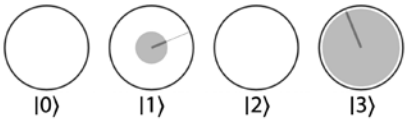


Рис. 3.31. Состояние одного кубита после чтения 1 из другого

Теперь вероятности чтения 0/1 для кубита *b* равны 30%/70%.

Хотя мы можем мгновенно изменить распределение вероятностей прочитанных значений для кубита *b*, сделать это намеренно невозможно (вы не можете выбрать распределение 70%/30% или 30%/70%), так как результат кубита чтения *a* измеряется случайным образом. И это тоже хорошо, потому что если бы такие изменения могли вноситься детерминировано, это означало бы возможность мгновенной (то есть превышающей скорость света) отправки сигналов с использованием квантовой запутанности. Хотя сама идея отправки сигналов со скоростью, превышающей скорость света, звучит заманчиво, это привело бы к самым неприятным последствиям[1].

[1] Последствиям типа «отправка информации обратно во времени с нарушением причинно-следственной связи». Авторитетные труды доктора Эммета Брауна свидетельствуют об опасности подобных выходок.

Одно из самых странных свойств квантовой запутанности заключается в том, что оно позволяет мгновенно изменять состояние кубитов на произвольных расстояниях, но всегда делает это так, чтобы исключить отправку содержательной, заранее определенной информации. Похоже, наша Вселенная не в восторге от научной фантастики.

Итоги

В этой главе было показано, что одно- и многокубитные операции позволяют манипулировать со свойствами суперпозиции и квантовой запутанности в QPU. Располагая такими средствами, вы сможете понять, какие новые и мощные способы вычислений становятся возможными с QPU. Как будет показано в главе 5, их появление приводит к переосмыслению фундаментальной цифровой логики. Тем не менее в начале следующей главы будет рассмотрено практическое исследование квантовой телепортации. Квантовая телепортация не только является фундаментальным компонентом многих квантовых приложений — ее исследование также объединяет все, что было сказано до сих пор об описании кубитов и работе с ними.

4 Квантовая телепортация

В этой главе будет представлена программа для QPU, позволяющая немедленно телепортировать объект[1] на расстояние 3,1 мм! А при наличии правильного оборудования тот же код будет работать и на межзвездных расстояниях.

Хотя слово «телепортация» может ассоциироваться с магией и фокусами, вы увидите, что разновидность квантовой телепортации, выполняемая при помощи QPU, выглядит столь же впечатляюще, но намного более практично — собственно, она является важнейшим концептуальным компонентом программирования QPU.

Практический пример: первые шаги в телепортации

Если вы хотите освоить телепортацию, лучше всего опробовать ее в деле. Учтите, что на протяжении всей человеческой истории до момента написания книги лишь несколько тысяч людей выполняло физическую телепортацию какого-либо рода, так что даже выполнение следующего кода ставит вас в число первопроходцев.

Для данного примера вместо системы моделирования мы воспользуемся реальным пятикубитным QPU компании IBM (рис. 4.1). Вы сможете скопировать пример кода из листинга 4.1 на сайт IBM Q Experience, щелкнуть на кнопке и убедиться в том, что телепортация прошла успешно.

---

[1] В данном случае телепортируется квантовое состояние объекта, а не сам объект. Квантовая телепортация — это обмен информацией, а не перемещение. — *Примеч. науч. ред.*

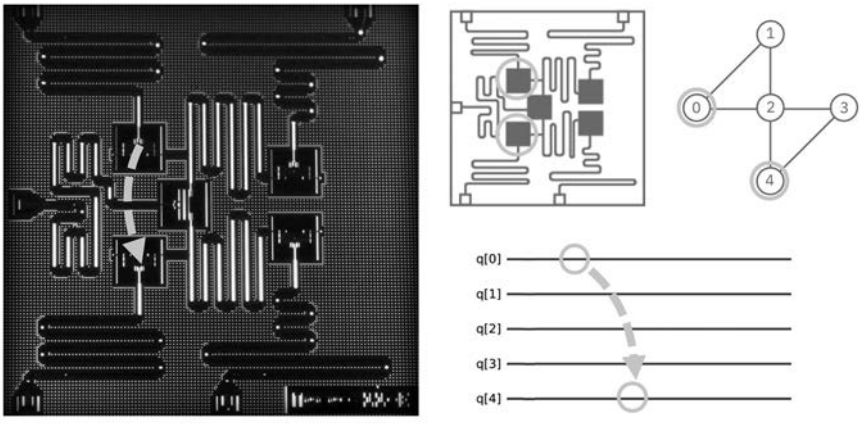


Рис. 4.1. Микросхема IBM очень мала, так что перемещение кубита будет довольно коротким; на иллюстрации и схеме выделены части QPU, между которыми будет происходить телепортация[1]

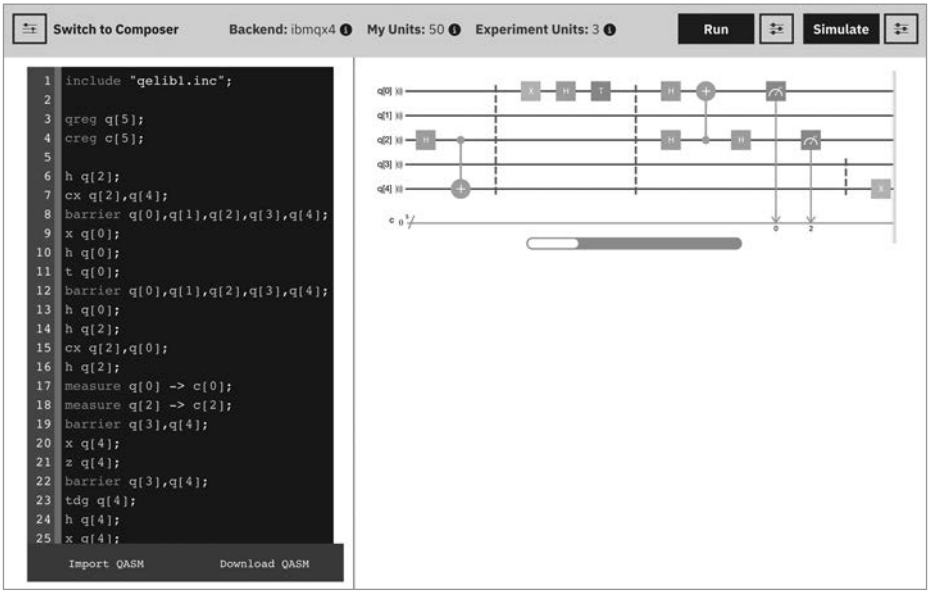


Рис. 4.2. Редактор IBM Q Experience (QX) OpenQASM[2]

[1] Публикуется с разрешения International Business Machines Corporation, © International Business Machines Corporation.

[2] Публикуется с разрешения International Business Machines Corporation, © International Business Machines Corporation.

Для программирования IBM Q Experience можно использовать OpenQASM[1] и Qiskit[2]. Обратите внимание: в листинге 4.1 содержится не код JavaScript, предназначенный для выполнения в QCEngine, а код OpenQASM, предназначенный для выполнения в облачном интерфейсе IBM (рис. 4.2). Выполнение этого кода позволит вам не смоделировать, а реально *выполнить* телепортацию кубита в исследовательском центре IBM в Йорктаун Хайтс (Нью-Йорк). Мы подробно опишем, как это делается. Подробное рассмотрение этого кода также поможет вам точно понять, как работает квантовая телепортация.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=4-1.

Листинг 4.1. Телепортация с проверкой

```
include "qelib1.inc";
qreg q[5];
creg c[5];

// Шаг 1: создать запутанную пару
h q[2];
cx q[2],q[4];
barrier q[0],q[1],q[2],q[3],q[4];

// Шаг 2: подготовить данные
x q[0];
h q[0];
t q[0];
barrier q[0],q[1],q[2],q[3],q[4];

// Шаг 3: отправить
h q[0];
h q[2];
cx q[2],q[0];
h q[2];
measure q[0] -> c[0];
measure q[2] -> c[2];
barrier q[3],q[4];

// Шаг 4: получить
x q[4];
z q[4];
barrier q[3],q[4];
```

[1] OpenQASM — квантовый язык ассемблера, поддерживаемый IBM Q Experience.

[2] Qiskit — пакет разработки с открытым кодом для работы с квантовыми процессорами. IBM Q.

```
// Шаг 5: проверить
tdg q[4];
h q[4];
x q[4];
measure q[4] -> c[4];
```

Прежде чем углубляться в подробности, сначала проясним ряд моментов. Под *квантовой телепортацией* мы подразумеваем возможность переноса точного состояния (то есть амплитуд и относительных фаз) из одного кубита в другой. Задача — взять всю информацию, содержащуюся в одном кубите, и поместить ее в другой кубит. Напомним, что квантовая информация не может реплицироваться; следовательно, информация в первом кубите обязательно уничтожается при телепортации во второй кубит. Так как квантовое описание является самым точным описанием физического объекта, в действительности именно такое явление мы называем телепортацией, только на квантовом уровне.

Разобравшись с этим, займемся телепортацией! В учебниках введение в квантовую телепортацию обычно начинается с истории, которая выглядит примерно так: пара кубитов, находящихся в *запутанном* состоянии, находится в распоряжении двух сторон, Алисы и Боба (физики обожают имена в алфавитном порядке). Запутанные кубиты — ресурс, который используется Алисой для телепортации состояния некоторого *другого* кубита Бобу. Таким образом, в телепортации задействованы три кубита: кубит *данных*, который хочет телепортировать Алиса, и запутанная пара кубитов, которые есть у нее и Боба (эти кубиты образуют некое подобие квантового кабеля Ethernet). Алиса готовит свои данные, а затем при помощи операций HAD и CNOT она запутывает кубит данных с другим кубитом (который, в свою очередь, уже запутан с кубитом Боба). Затем она разрушает свой кубит данных и запутанный кубит операциями READ. Результаты этих операций READ дают два традиционных бита информации, которые она отправляет Бобу. Так как на этот раз речь идет о *битах*, а не о кубитах, можно воспользоваться традиционным кабелем Ethernet. Используя эти два бита, Боб выполняет некие однокубитные операции со своей половиной запутанной пары, которая изначально принадлежала ему и Алисе — и получает кубит данных, который собиралась отправить Алиса.

Прежде чем перейти к описанному здесь[1] протоколу квантовых операций, необходимо разобраться с одной проблемой. *«Постойте-ка... эта часть рождена нашим воображением. Если Алиса отправляет Бобу традицион-*

[1] Полный исходный код версий этого примера для QASM и QCEngine доступен по адресу http://oreilly-qc.github.io?p=4-1.

ную информацию по кабелю Ethernet, — продолжаете вы, — то такая телепортация не особо впечатляет». Отличное наблюдение! Успех квантовой телепортации действительно зависит от передачи традиционных (цифровых) битов. Мы уже видели, что амплитуды и относительные фазы должны полностью описывать произвольное состояние, которое может принимать кубит в континууме значений. Принципиально здесь то, что протокол телепортации работает даже в том случае, когда Алиса не знает состояние своего кубита. И это особенно важно, потому что невозможно определить амплитуду и относительную фазу одного кубита в неизвестном состоянии. Тем не менее — с помощью запутанной пары кубитов — для эффективной передачи точной конфигурации кубита Алисы (независимо от его комплексных амплитуд) достаточно всего двух традиционных битов. Эта конфигурация будет верной при потенциально бесконечном количестве битов точности!

Как же добиться этого волшебного эффекта? Ниже описан весь протокол. На рис. 4.3 изображены операции, которые должны быть выполнены с тремя задействованными кубитами. Все эти операции были представлены в главах 2 и 3.

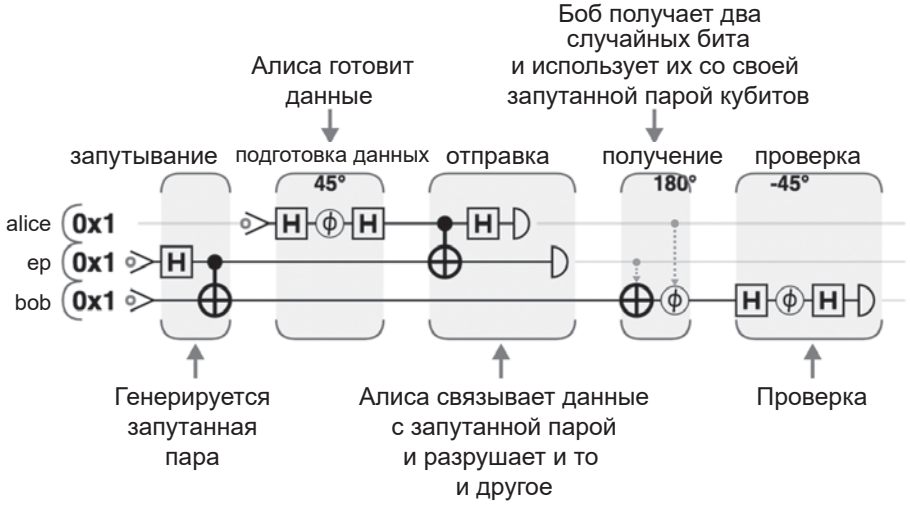


Рис. 4.3. Полная схема телепортации: на стороне Алисы находятся кубиты alice и ep, а на стороне Боба — кубит bob

Если вставить код из этой схемы из листинга 4.1 в систему IBM QX, пользовательский интерфейс IBM покажет схему на рис. 4.4. Обратите внимание: это точно такая же программа, как на рис. 4.3, только представленная несколько иным образом. Система обозначений с квантовыми вентилями,

которую мы использовали, считается стандартизированной; несомненно, она встретится вам за пределами книги[1].

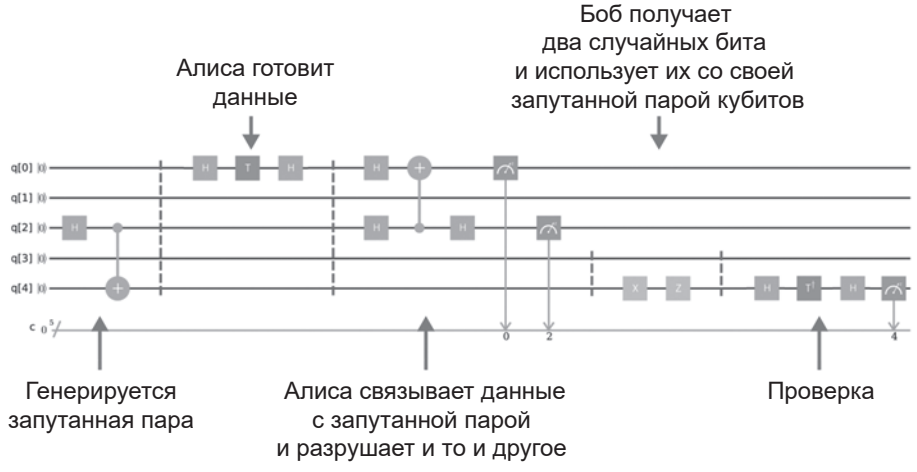


Рис. 4.4. Схема телепортации в IBM QX[2]

При нажатии кнопки Run IBM выполнит вашу программу 1024 раза (это число можно настроить), а затем выведет статистику по запуску. После выполнения программы результат будет выглядеть примерно (но не в точности) так, как показано на гистограмме на рис. 4.5.

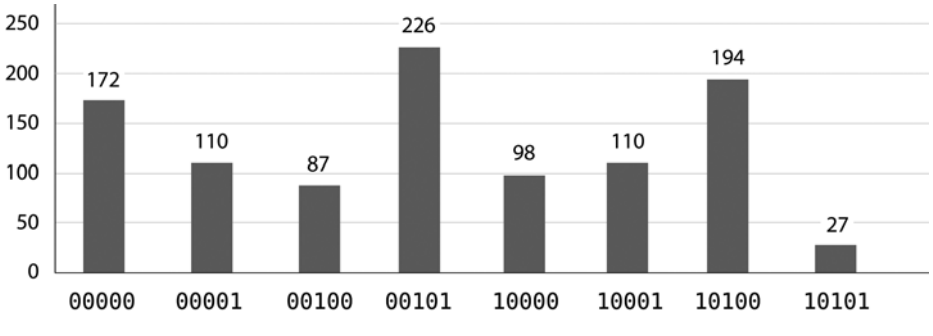


Рис. 4.5. Результаты запуска программы (успешная телепортация?)

[1] Такие вентили, как CNOT и HAD, могут комбинироваться разными способами для получения того же результата. Для некоторых операций в IBM QX и QCEngine используются разные разложения.

[2] Публикуется с разрешения International Business Machines Corporation, © International Business Machines Corporation.

Успех? Возможно! Чтобы продемонстрировать, как читать и интерпретировать эти результаты, мы подробнее разберем каждый шаг программы QPU, используя круговую запись для наглядного представления всего, что происходит с нашими кубитами[1].

На момент написания книги схемы и результаты, отображаемые IBM QX, показывают, что происходит со всеми пятью кубитами, доступными для QPU, даже если они не использовались. Это объясняет, почему в схеме на рис. 4.4 присутствуют две пустые линии кубитов и почему столбцы с результатами для каждого вывода на рис. 4.5 помечаются пятиразрядными двоичными числами (вместо трехразрядных) — несмотря на то что фактически показаны только столбцы, соответствующие $2^3 = 8$ возможным конфигурациям трех использованных кубитов.

Анализ программы

Так как в нашем примере телепортации используются три кубита, для их полного описания потребуются $2^3 = 8$ кругов (по одному для каждой возможной комбинации 3 битов). Мы расположим эти восемь кругов в две строки, что поможет вам понять, как операции влияют на три составляющих кубита. На рис. 4.6 все строки и столбцы кругов помечены конкретными значениями каждого кубита. Чтобы проверить правильность меток, обратитесь к двоичному значению соответствующего регистра.

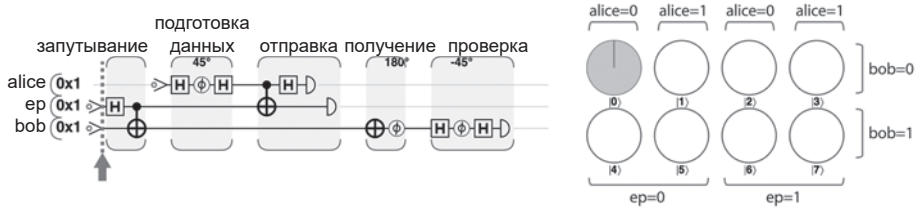


Рис. 4.6. Полная программа телепортации с проверкой

При работе с многокубитными регистрами круговая запись часто размещается по строкам и столбцам, как на рис. 4.6. Такое размещение всегда помогает быстрее выявить закономерности в поведении отдельных кубитов: вы всегда сразу замечаете важные строки или столбцы.

В начале программы все три кубита инициализируются в состоянии $|0\rangle$, как видно из рис. 4.6, — единственным возможным значением является значение, при котором `alice=0`, `ep=0` и `bob=0`.

[1] Полный исходный код этого примера доступен по адресу http://oreilly-qc.github.io?p=4-1.

Шаг 1: создание запутанной пары

Первым шагом телепортации должна стать установка связи через запутанность. Комбинация HAD и CNOT, решающая эту задачу, была использована в главе 3 для создания запутанного состояния двух кубитов — так называемой *пары Белла*. Из круговой записи на рис. 4.7 видно, что при чтении `bob` и `ep` значения будут случайными с распределением 50/50, но они будут гарантированно совпадать друг с другом.

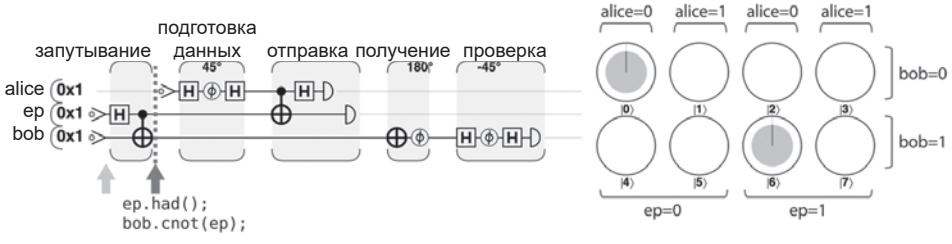


Рис. 4.7. Шаг 1: создание запутанной пары

Шаг 2: подготовка данных

После того как связь квантовой запутанности будет установлена, Алиса может подготовить данные для отправки. Конечно, способ подготовки зависит от природы (квантовой) информации, которую Алиса хочет отправить Бобу. Она может записать значение в кубит данных, ввести в состояние запутанности с другими данными QPU или даже получить из предыдущих вычислений в совершенно отдельной части QPU.

Для нашего примера мы разочаруем Алису и предложим ей подготовить особенно простой кубит данных с использованием операций HAD и PHASE. Преимуществом такого решения является получение данных с легко расшифровываемой закономерностью круговой записи (рис. 4.8).

Мы видим, что кубиты `bob` и `ep` продолжают зависеть друг от друга (только круги, соответствующие кубитам `bob` и `ep` с одинаковыми значениями, имеют ненулевые амплитуды).

Также видно, что значение `alice` не зависит ни от одного из двух других кубитов; более того, в результате подготовки данных был получен кубит с вероятностями 85,4% $|0\rangle$ и 14,6% $|1\rangle$ и относительной фазой –90° (круги, соответствующие `alice=1`, повернуты на 90° по часовой стрелке относительно кругов `alice=0`, что соответствует отрицательным значениям по нашим соглашениям).

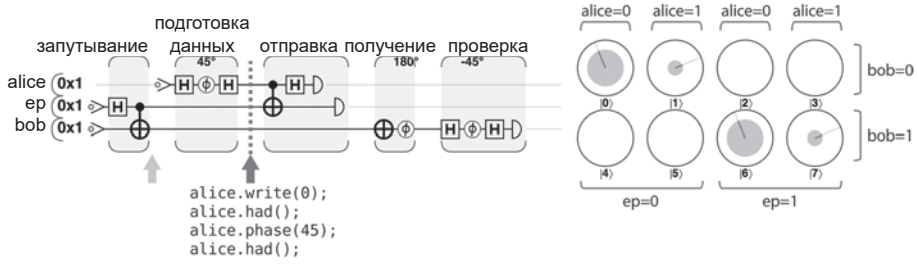


Рис. 4.8. Шаг 2: подготовка данных

Шаг 3.1: связывание данных с запутанной парой

В главе 2 было показано, что условная природа операции CNOT позволяет задействовать состояния двух кубитов. Теперь Алиса использует этот факт, чтобы запутать кубит данных со своей половиной запутанной пары (другая половина находится у Боба). В круговой записи это действие меняет круги местами, как показано на рис. 4.9.

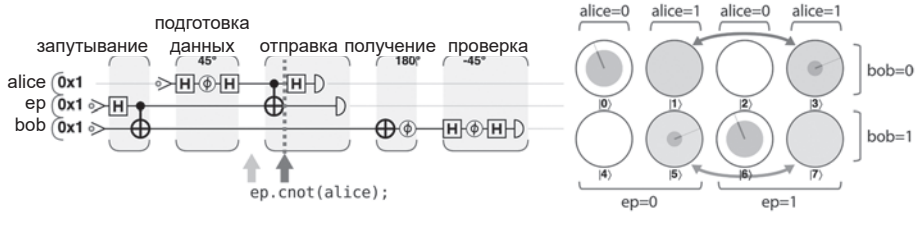


Рис. 4.9. Шаг 3.1: связывание данных с запутанной парой

Появление *множественных* запутанных состояний усложняет ситуацию, поэтому стоит пояснить происходящее. И у Алисы, и у Боба уже имеется один из двух запутанных кубитов (полученных на шаге 1). Теперь Алиса запутывает другой кубит (данные) со своей половиной (уже запутанной) пары. На интуитивном уровне понятно, что Алиса в каком-то смысле косвенно связала свои данные с половиной запутанной пары, принадлежащей Бобу, хотя ее кубит данных остался неизменным. Результаты READ для ее данных будут логически связаны с этими двумя другими кубитами. Эта связь видна в круговой записи, поскольку состояние регистра QPU на рис. 4.9 содержит только те конфигурации, у которых результат XOR всех трех кубитов равен 0. Ранее это условие выполнялось для ep и bob, но теперь оно выполняется для всех трех кубитов, образующих *трехкубитную* запутанную группу.

Шаг 3.2: перевод кубита данных в суперпозицию

Чтобы связь, созданная Алисой для ее набора данных, была действительно полезной, Алиса должна сделать завершающий шаг и выполнить операцию HAD со своими данными (рис. 4.10).

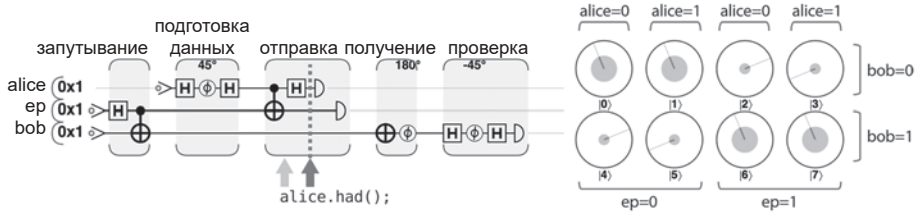


Рис. 4.10. Шаг 3.2: перевод данных в суперпозицию

Чтобы понять, для чего Алисе нужна эта операция HAD, взгляните на представление состояния трех кубитов в круговой записи на рис. 4.10. В каждом столбце находится пара кругов, обозначающих кубит, который может получить Боб (вскоре будет показано, что какой именно кубит он получит, зависит от результатов операций READ, выполненных Алисой). Интересно, что четыре потенциальных состояния, которые может получить Боб, являются разными вариациями исходного кубита данных Алисы.

- В первом столбце (для `alice=0` и `ep=0`) содержатся данные Алисы точно в таком виде, в котором она их подготовила.
- Во втором столбце содержатся те же данные, но с применением операции PHASE(180).
- В третьем столбце содержатся правильные данные, но с применением операции NOT ($|0\rangle$ и $|1\rangle$ меняются местами).
- Наконец, в последнем столбце применены операции сдвига фазы и отрицания (то есть PHASE(180) и NOT).

Если бы Алиса не применила операцию HAD, то это привело бы к разрушению информации магнитуды и фазы при применении операций READ, которые она вскоре будет использовать (попробуйте сами!). Применяя операцию HAD, Алиса смогла приблизить состояние кубита Боба к своим данным.

Шаг 3.3: чтение обоих кубитов Алисы

Затем Алиса выполняет операцию READ со своими двумя кубитами (данными и своей половиной запутанной парой, вторая половина которой на-

ходится у Боба). Операция READ необратимо разрушает оба кубита. Резонно спросить, для чего Алиса это делает? Как вы увидите, результаты этой неизбежно разрушительной операции READ играют важную роль в работе протокола телепортации. Копирование квантовых состояний невозможно даже при использовании запутанности. Единственный способ передачи квантовых состояний — их телепортация, а при телепортации оригинал должен быть уничтожен.

На рис. 4.11 Алиса выполняет предписанные операции READ со своими данными и своей половиной запутанной пары. Эта операция возвращает два бита.

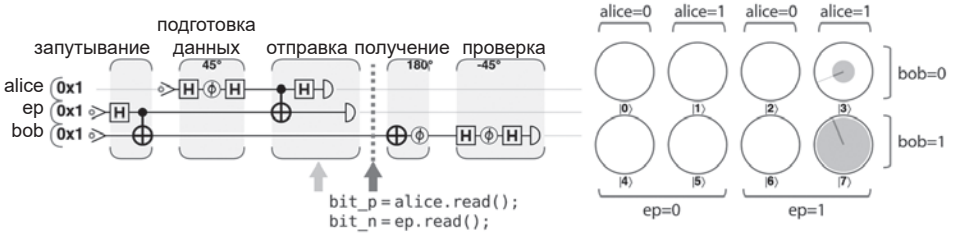


Рис. 4.11. Шаг 3.3: чтение обоих кубитов Алисы

В контексте круговой записи рис. 4.11 показывает, что при чтении значений своих кубитов Алиса выбрала один столбец кругов (какой именно — зависит от случайных результатов READ), в результате чего круги за пределами этого столбца имеют нулевую магнитуду.

Шаг 4: получение и преобразование

На шаге 3.2 («перевод кубита данных в суперпозицию») было показано, что кубит Боба может оказаться в одном из четырех состояний, каждое из которых связано с данными Алисы операциями HAD и/или PHASE(180). Если бы Боб мог узнать, каким из этих четырех состояний он обладает, он мог бы применить необходимые обратные операции, чтобы вернуться к исходному кубиту данных Алисы. И два бита, полученные Алисой от операций READ, — именно та информация, которая необходима Бобу! Итак, на этой стадии Алиса *берет телефон и передает Бобу два бита традиционной информации*.

На основании двух полученных битов Боб знает, какой столбец из представления круговой записи представляет его кубит. Если первый бит, полученный от Алисы, равен 1, он выполняет операцию NOT с кубитом. Затем, если второй бит равен 1, он также выполняет операцию PHASE(180), как показано на рис. 4.12.

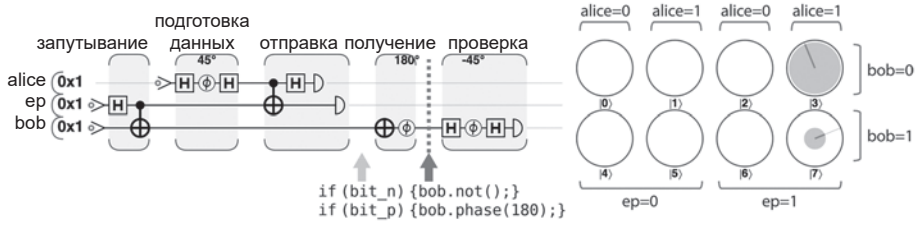


Рис. 4.12. Шаг 4: получение и преобразование

Протокол телепортации на этом завершен — в распоряжении Боба появился кубит, неотличимый от исходных данных Алисы.

Существующее оборудование IBM QX не поддерживает операции прямой связи, необходимые для того, чтобы (совершенно случайные) биты операции Алисы READ управляли действиями Боба. Этот недостаток можно обойти при помощи пост-селекции. Боб всегда выполняет одну и ту же операцию независимо от того, что отправила Алиса. Такое поведение может вывести Алису из себя (рис. 4.13). После этого мы рассматриваем все варианты вывода и отбрасываем все результаты, кроме тех, которые Боб получил бы при наличии информации от Алисы.

Рис. 4.13. Вместо прямой передачи можно считать, что Боб заснул на рабочем месте, и отбросить те случаи, в которых он принял неправильные решения

Шаг 5: проверка результата

Если Алиса и Боб используют схему телепортации в серьезной работе, их задача на этом решена. Боб берет телепортированный Алисой кубит и продолжает использовать его в большом квантовом приложении, с ко-

торым они работают. Пока они доверяют своему оборудованию QPU, они могут быть уверены, что Боб получил именно тот кубит, который отправила Алиса.

Но что делать в тех случаях, когда стороны хотят *проверить*, что оборудование телепортировало кубит правильно (даже если вас не пугает, что телепортированный кубит будет уничтожен в процессе проверки)?

Единственно возможный вариант — чтение итогового кубита Боба операцией READ. Конечно, нельзя рассчитывать на то, что вы узнаете (а следовательно, сможете проверить) состояние его кубита одной операцией READ, но при повторении всего процесса телепортации и многократном применении READ вы начнете получать представление о состоянии кубита Боба.

Собственно, простейший способ проверки успешности протокола телепортации на физическом устройстве заключается в том, чтобы Боб выполнил шаги подготовки данных, которые выполнялись Алисой с состоянием $|0\rangle$ для создания ее данных, со своим итоговым кубитом, только в обратном порядке. Если кубит, имеющийся у Боба, действительно совпадает с отправленным Алисой, то Боб должен получить состояние $|0\rangle$, и если Боб после этого выполнит последнюю проверочную операцию READ, то она должна всегда возвращать 0. Если Боб при чтении из тестового кубита когда-либо получит ненулевое значение, значит, телепортация завершилась неудачей. Этот дополнительный проверочный шаг показан на рис. 4.14.

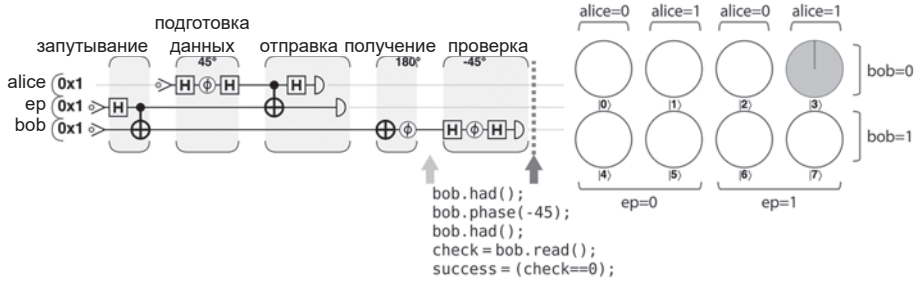


Рис. 4.14. Шаг 5: проверка результата

Даже если Алиса и Боб занимаются серьезной работой по телепортации, вероятно, они будут чередовать реальные телепортации с многочисленными проверками вроде показанной — просто для того, чтобы быть уверенными в правильности работы QPU.

Интерпретация результатов

Вооружившись более полным пониманием протокола телепортации и его тонкостей, вернемся к результатам, полученным в ходе физического эксперимента по телепортации на IBM QX. Теперь мы располагаем всем необходимым для их расшифровки.

В протоколе задействованы три операции READ: две выполняются Алисой как составляющие телепортации, и одна — Бобом для проверочных целей. На гистограмме на рис. 4.5 показано, сколько раз каждая из 8 возможных комбинаций этих результатов встречалась за 1024 попытки телепортации.

Как упоминалось ранее, мы будем применять пост-селекцию с IBM QX для поиска случаев, в которых Бобу повезло выполнить правильные операции (так, как если бы он работал с результатами READ, полученными от Алисы). В схеме, которая была передана IBM QX на рис. 4.4, Боб всегда выполнял операции HAD и PHASE(180) со своим кубитом, поэтому мы должны провести ретроспективный выбор тех случаев, в которых две операции READ, выполненные Алисой, дали результат 11. В результате остаются два набора полезных результатов, в которых действия Боба по случайности правильно совпали с результатами READ Алисы (рис. 4.15).

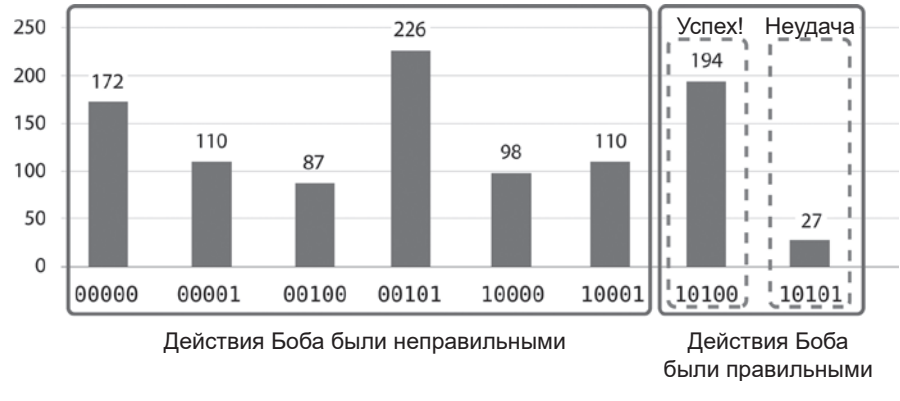


Рис. 4.15. Интерпретированные результаты телепортации

Из 221 раза, когда Боб выполнил правильные операции, телепортация была выполнена успешно, когда проверочная операция READ Боба дала значение 0 (так как он использует шаг проверки, о котором говорилось ранее). Это означает, что телепортация была выполнена успешно 194 раза и завершилась неудачей 27 раз. Вероятность успеха 87,8% — не так уж плохо, если учесть, что Алиса и Боб использовали лучшее оборудование, существую-

щее в 2019 году. Им стоит подумать дважды перед тем, как передавать что-нибудь важное.

Если Боб получает ошибочно телепортированный кубит, который почти не отличается от отправленного Алисой, проверка с большой вероятностью сообщит об успехе. Только при многократном выполнении этой проверки мы можем быть уверены в том, что устройство работает нормально.

Как используется телепортация?

Телепортация является одним из фундаментальных принципов работы QPU — даже в чисто вычислительных приложениях, не имеющих очевидного «коммуникационного» аспекта. Она позволяет передавать информацию между кубитами, обходя ограничение запрета на копирование. Большинство практических применений телепортации на очень короткие расстояния в QPU является неотъемлемой частью квантовых приложений. В последующих главах вы увидите, что многие квантовые операции с двумя и более кубитами основаны на формировании различных типов запутанности. Использование таких квантовых связей в ходе вычислений обычно может рассматриваться как практическое применение общей концепции телепортации. И хотя мы можем не сознавать, что телепортация задействована в рассматриваемых нами алгоритмах и приложениях, она играет в них основополагающую роль.

Известные проблемы при телепортации

Мы большие любители научной фантастики, и наш любимый пример использования телепортации встречается в классическом фильме «Муха». Как в оригинале 1958 года, так и в более современной версии 1986 года с Джеффом Голдблюмом изображен эксперимент телепортации, в ходе которого происходит ошибка. После того как главному герою не удается телепортировать кошку, он решает, что следующим логичным шагом будет эксперимент на самом себе, но он не подозревает, что вместе с ним в камеру телепортации залетела муха.

Нам немножко неудобно преподносить вам тяжелую новость: реальная квантовая телепортация не способна на то, что показано в «Мухе». Чтобы вам не было так горько, мы написали код для телепортации квантового изо-

бражения мухи — даже с небольшой погрешностью, обусловленной сюжетом фильма. Пример кода можно запустить онлайн по адресу http://oreilly-qc.github.io?p=4-*2*; ужасный результат показан на рис. 4.16.

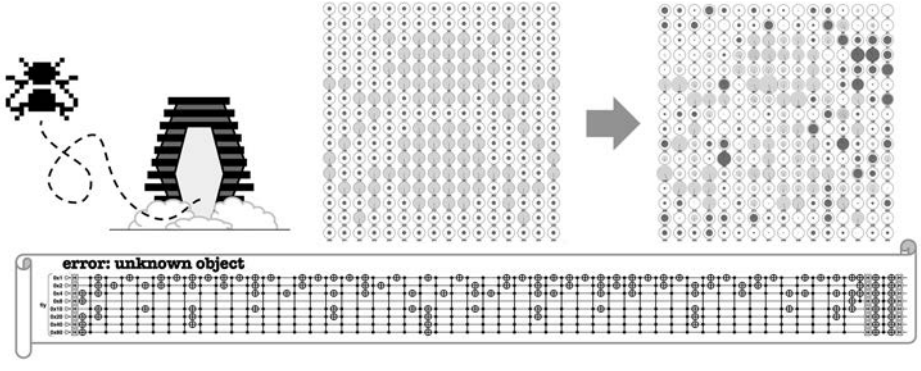


Рис. 4.16. Бойтесь!

ЧАСТЬ II
Примитивы QPU

Вы научились описывать кубиты и выполнять операции с ними на простейшем уровне, и мы можем перейти к *примитивам QPU* более высокого уровня. Эти примитивы образуют инструментарий, который в конечном итоге позволяет строить полноценные приложения.

В очень общих чертах квантовые приложения часто имеют структуру, показанную на рис. II.1.

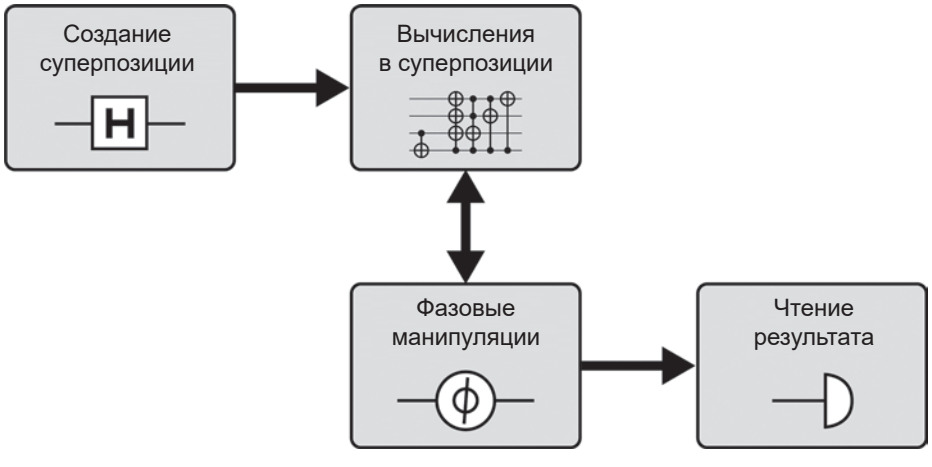


Рис. II.1. Высокоуровневая структура квантового приложения

Примитивы QPU помогают заполнить эту структуру. Примитивы, связанные со вторым из этих четырех шагов (вычисления в суперпозиции), позволяют проводить вычисления с использованием косвенного параллелизма суперпозиции, тогда как примитивы, реализующие третий шаг (фазовые манипуляции), гарантируют, что полученные результаты можно будет прочитать для практического использования.

Эти шаги обычно реализуются совместно и многократно повторяются так, как того требует конкретное приложение. Вместо одного универсального

примитива для каждого шага нам понадобится арсенал. В следующих пяти главах будут представлены примитивы, перечисленные в табл. II.1.

Таблица II.1. Квантовые примитивы

Примитив	Тип	Глава
Цифровая логика	Вычисления в суперпозиции	5
Усиление комплексной амплитуды	Фазовые манипуляции	6
Квантовое преобразование Фурье	Фазовые манипуляции	7
Оценка фазы	Фазовые манипуляции	8
Квантовые типы данных	Создание суперпозиции	9

В каждой главе мы сначала приводим практическое знакомство с примитивом, а затем описываем возможности использования этого примитива в реальных приложениях. Наконец, каждая глава завершается разделом «Внутри QPU», который дает интуитивное представление о работе примитива, часто с разбиением на фундаментальные операции QPU, представленные в главах 2 и 3.

Искусство программирования QPU заключается в том, чтобы определить, какая комбинация примитивов из табл. II.1 позволит сформировать структуру с рис. II.1 для конкретного приложения. В части III будут приведены примеры таких конструкций.

Итак, займемся изучением примитивов QPU!

5 Квантовая арифметика и логика

Преимущества приложений QPU перед их традиционными аналогами часто связаны с возможностью выполнения большого количества логических операций в суперпозиции[1].

В этом отношении очень важно иметь возможность применять простые арифметические операции к кубитным регистрам в суперпозиции. В этой главе подробно рассказано, как это делается. Изначально арифметические операции будут рассмотрены на более абстрактном уровне, типичном для традиционного программирования: мы будем работать с целыми числами и переменными вместо кубитов и операций. Но ближе к концу главы также будут более подробно описаны логические операции, на базе которых они строятся (аналоги элементарных вентилей в цифровой логике).

Странно и необычно

В традиционной цифровой логике хорошо известны оптимизированные методы выполнения арифметических операций — почему нельзя просто заменить биты кубитами и использовать их в QPU?

Конечно, проблема в том, что традиционные операции работают с *одним* набором входных данных, тогда как в квантовых приложениях входные регистры часто находятся в суперпозиции, и *квантовые* арифметические операции должны воздействовать на *все* значения в суперпозиции.

---

[1] Как было сказано во введении к этой части книги, одной суперпозиции недостаточно. Еще один (исключительно важный!) шаг должен гарантировать, что результаты (по сути) параллельных вычислений, выполняемых нами, можно будет прочитать, и они не останутся навечно скрытыми в квантовых состояниях кубитов.

Начнем с простого примера, который демонстрирует, как должна работать логика с суперпозициями. Допустим, имеются три однокубитных регистра QPU: `a`, `b` и `c`; требуется реализовать следующую логику: `if (a and b) then инвертировать c`. Таким образом, если `a` содержит $|1\rangle$, и `b` содержит $|1\rangle$, то значение `c` будет изменено на противоположное. На рис. 3.20 было показано, что эта логика напрямую реализуется одной операцией Тоффоли. В круговой записи вентиль Тоффоли меняет местами значения $|3\rangle$ и $|7\rangle$ независимо от их содержимого. Если `b` содержит $|0\rangle$, а `a` содержит $|1\rangle$, как на рис. 5.1, то переставляемые круги пусты, так что вентиль не влияет на `c`.

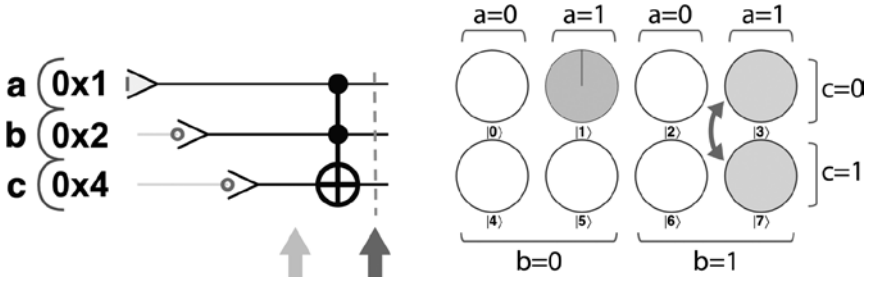


Рис. 5.1. Если b=0, то операция не имеет последствий

Теперь предположим, что операция HAD (см. главу 2) используется для подготовки `b` в суперпозиции $|0\rangle$ и $|1\rangle$ — что должно происходить с `c`? При правильной реализации эта операция должна одновременно инвертировать и не инвертировать `c` в суперпозиции (рис. 5.2). Вентили Тоффоли существуют и для традиционных вычислений, но они, безусловно, не справятся с этой задачей. Нам понадобится квантовая реализация вентиля Тоффоли, работающая с квантовыми регистрами. Круговая запись на рис. 5.2 поможет увидеть, что должен сделать правильно реализованный вентиль Тоффоли с входными данными в суперозиции.

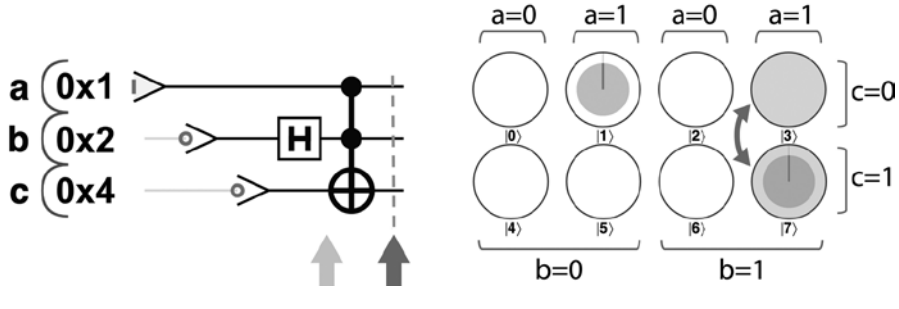


Рис. 5.2. Один вентиль выполняет две операции одновременно

Также есть несколько других требований, которым должна удовлетворять операция для того, чтобы она правильно работала с кубитами в QPU. И хотя некоторые из них могут быть не столь очевидны, как в приведенном примере, необходимо учитывать их при построении различных арифметических и логических операций для QPU.

- *Перемещение и копирование данных*. Как вы уже знаете, кубиты не могут быть скопированы. В этом отношении квантовая логика принципиально отличается от традиционной. Перемещение или копирование битов — самая распространенная операция, выполняемая традиционным процессором. QPU может перемещать кубиты с использованием команд, меняющих их местами с другими кубитами, но ни один QPU никогда не сможет реализовать команду `COPY`. В результате оператор =, так часто встречающийся при работе с цифровыми значениями, не может использоваться для присваивания одного кубитного значения другому.
- *Обратимость и потеря данных*. В отличие от многих традиционных логических операций, основные операции `QPU` (кроме `READ`) обратимы (из-за законов квантовой механики). Этот факт устанавливает серьезные ограничения на логические и арифметические операции, которые могут выполняться с QPU, и часто заставляет нас творчески мыслить при попытках замены традиционных арифметических операций. `READ` — единственная необратимая операция; возможно, у вас появится искушение интенсивно пользоваться ею для построения необратимых операций. Не надо! Ваши вычисления станут настолько традиционными, что, скорее всего, вы лишитесь всех преимуществ квантовых вычислений. Простейший способ реализации *любой* традиционной схемы в QPU заключается в замене ее эквивалентной традиционной схемой, использующей только *обратимые* операции — например, операции Тоффоли. Такая операция может быть реализована на квантовом регистре практически в неизменном виде[1].

Арифметика с QPU

В традиционном программировании разработчик редко использует отдельные логические вентили для написания программ. Вместо этого мы поручаем компилятору и процессору преобразовать нашу программу в операции с вентилями, необходимые для реализации нужного поведения.

---

[1] Любая традиционная операция, реализованная на базе обратимой логики с использованием *N* вентилей Тоффоли, может быть реализована с использованием O(N) одно- и двухкубитных операций.

С квантовыми вычислениями дело обстоит примерно так же. Чтобы написать серьезную программу для QPU, прежде всего необходимо научиться работать с кубайтами и квантовыми целыми числами вместо кубитов. В этом разделе будут описаны нюансы выполнения арифметических операций на QPU. Подобно тому как классическая цифровая логика может быть построена из вентилей NAND (вентиль, выполняющий операцию NOT(a AND b)), квантовые целочисленные операции могут быть построены из элементарных операций QPU, рассмотренных в главах 2 и 3.

Для простоты мы можем построить диаграмму и продемонстрировать арифметические операции на целых числах из четырех кубитов (*квантовые полубайты*, или *квантовые тетрады*, для тех, кто помнит начало эпохи микрокомпьютеров). Все эти примеры расширяются до регистров QPU большего размера. Размеры целых чисел, с которыми смогут работать арифметические операции, зависят от количества доступных кубитов в QPU или в системе моделирования.

Практический пример: построение операторов инкремента и декремента

Две простейшие целочисленные арифметические операции предназначены для увеличения на 1 (инкремента) и уменьшения на 1 (декремента) числа. Попробуйте выполнить листинг 5.1 и последовательно пройдите по вентилям, чтобы понаблюдать за операциями, показанными на рис. 5.3.

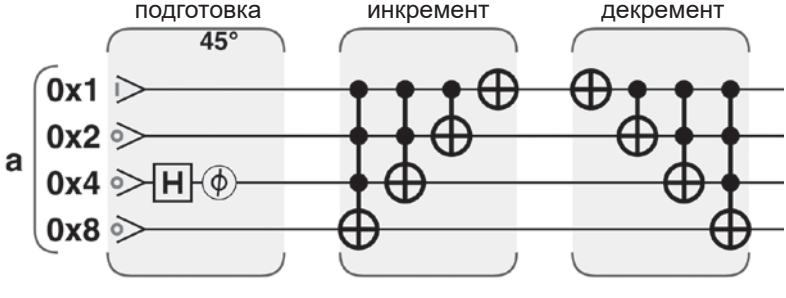


Рис. 5.3. Операции инкремента и декремента

Шаг подготовки и значение инициализации на рис. 5.3 не являются частью операции инкремента; они выбраны просто для того, чтобы предоставить ненулевое входное состояние для наблюдения за ходом событий.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=5-1.

Листинг 5.1. Целочисленный инкремент

```
// Инициализация
var num_qubits = 4;
qc.reset(num_qubits);
var a = qint.new(num_qubits, 'a');

// Подготовка
a.write(1);
a.hadamard(0x4);
a.phase(45, 0x4);

// Инкремент
a.add(1);

// Декремент
a.subtract(1);
```

В листинге 5.1 операции инкремента и декремента реализуются функциями QCEngine `add()` и `subtract()` с аргументом 1.

Эти реализации удовлетворяют всем требованиям, необходимым для того, чтобы они могли считаться квантовыми, в частности:

- *Обратимость*. В первую очередь операция декремента представляет собой операцию инкремента, выполненную в обратном порядке составляющих операций. Это вполне разумно, но может быть не так очевидно, если вы привыкли к традиционной логике. В традиционной логике у устройств обычно есть выделенные входы и выходы, и попытки инициировать работу устройства в обратном направлении либо приведет к его повреждению, либо по крайней мере не позволит получить полезный результат. Как упоминалось ранее, для квантовых операций обратимость является критичным требованием.
- *Операция с суперпозицией*. Реализация инкремента также должна работать со входными данными в суперпозиции. В листинге 5.1 фаза подготовки записывает значение $|1\rangle$ в квантовое целое число, а затем выполняет HAD и PHASE(45) в кубит 0x4, в результате чего регистр содержит суперпозицию[1] $|1\rangle$ и $|5\rangle$ (рис. 5.4).

[1] Мы включили изменение фазы на 45° между двумя значениями для того, чтобы их было проще различить.

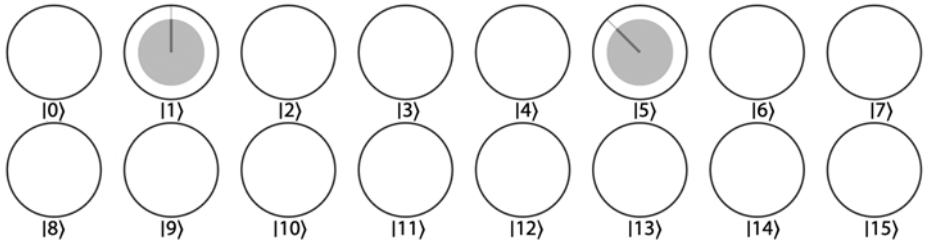


Рис. 5.4. Подготовленная суперпозиция перед инкрементом

А теперь попробуем выполнить операцию инкремента с этими входными данными. При этом состояние преобразуется в суперпозицию $|2\rangle$ и $|6\rangle$, в которой фаза каждого значения соответствует своему аналогу перед инкрементом (рис. 5.5).

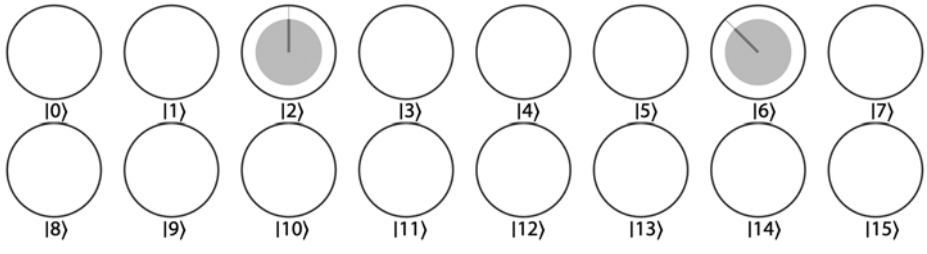


Рис. 5.5. Суперпозиция, полученная в результате инкремента

Как работает инкремент? Внимательно рассмотрев задействованные операции, вы увидите, что все начинается с применения вентиля CNOT с тремя управляющими условиями для реализации операции «если все младшие биты целого числа равны 1, изменить состояние старшего бита». По сути выполняется традиционная арифметическая операция переноса. Затем процесс повторяется для каждого бита в целом числе; в результате вы получаете полноценную операцию «сложения с переносом» для всех кубитов, но выполненную с использованием только вентилей CNOT с несколькими управляющими условиями.

Мы можем выйти за пределы простого инкремента и декремента. Попробуйте изменить целые значения, передаваемые `add()` и `subtract()` в листинге 5.1. Подойдет любое целое число, хотя разные варианты приведут к разным конфигурациям операций QPU. Например, `add(12)` создаст схему на рис. 5.6.

В данном случае из рис. 5.7 видно, что входные значения $|1\rangle$ и $|5\rangle$ преобразуются в $|13\rangle$ и $|1\rangle$ из-за переполнения (как и в традиционных целочисленных вычислениях).

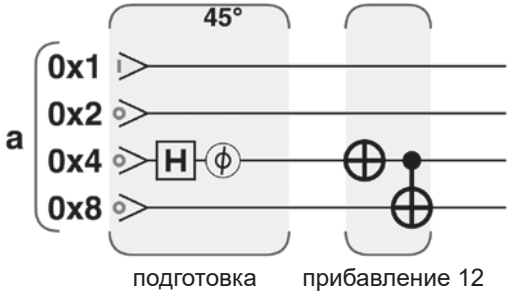


Рис. 5.6. Программа для прибавления 12 к квантовому целому числу

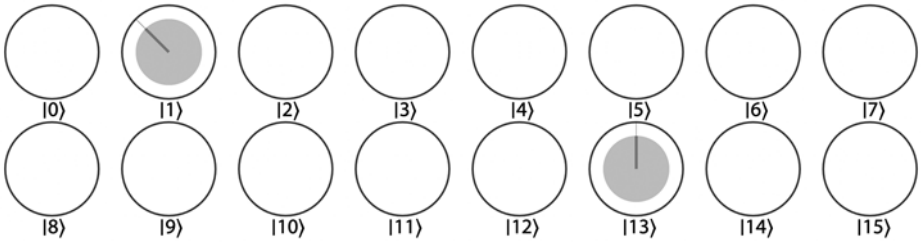


Рис. 5.7. Функция add(12) применяется к суперпозиции состояний 1 и 5

Из того факта, что эти функции могут получать любое целое число, следует интересный вывод: эта программа выполняет арифметические вычисления с традиционным цифровым и с квантовым значением. Мы всегда прибавляем *фиксированное* целое число к квантовому регистру, и нам приходится изменять набор используемых вентилей и приводить их в соответствие с конкретным прибавляемым целым числом. Но нельзя ли пойти еще дальше и выполнить операцию сложения между двумя *квантовыми* значениями?

Сложение двух квантовых целых чисел

Допустим, имеются два регистра `a` и `b` (помните, что в каждом из них теоретически может храниться суперпозиция целых значений), и вы хотите выполнить простую операцию `+`, которая сохранит результат сложения в новом регистре `c`. Происходящее аналогично тому, как процессор выполняет сложение с традиционными цифровыми регистрами. Однако тут возникает одна проблема — такой подход нарушает как обратимость, так и ограничения с запретом на копирование в логике QPU.

- Операция `c=a+b` нарушает обратимость, потому что предыдущее содержимое `c` теряется.

- Невозможность копирования нарушается тем, что мы можем схитрить и выполнить операцию `b=c-a`, чтобы в итоге получить две копии суперпозиции, которая могла храниться в `a`.

Чтобы обойти это ограничение, можно реализовать оператор `+=` и прибавить одно число непосредственно к другому. Пример кода в листинге 5.2, показанном на рис. 5.8, выполняет обратимое сложение двух квантовых целых чисел, в какой бы суперпозиции они ни находились. В отличие от предыдущего способа прибавления традиционного цифрового числа к квантовому регистру здесь присутствуют вентили, которые не нужно заново настраивать при каждом изменении входных значений.

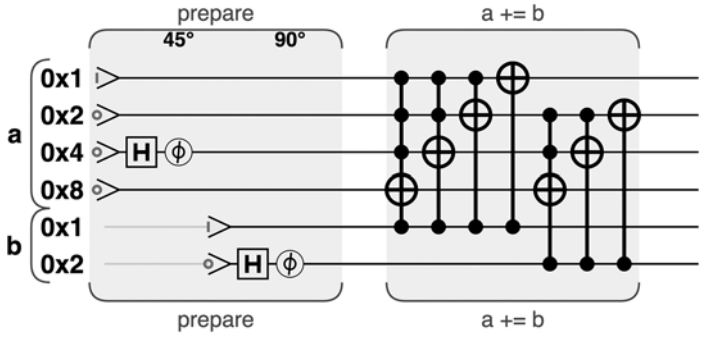


Рис. 5.8. Набор операций для выполнения операции +=

Как работает схема на рис. 5.8? Если более внимательно рассмотреть вентили этой программы, вы увидите, что они представляют обычные операции целочисленного сложения с рис. 5.3 и 5.6, примененные к *a*, но выполняемые условно в зависимости от состояния соответствующих кубитов *b*. Это позволяет значениям *b*, даже находящимся в суперпозиции, определять эффект сложения.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=5-2.

Листинг 5.2. Сложение двух квантовых чисел

```
// a += b
a.add(b);
```

Как и в примере из листинга 5.1, шаг подготовки на рис. 5.8 существует только для того, чтобы предоставить тестовые данные, и не является частью операции. Кроме того, операцию `a -= b` можно реализовать простым выполнением операции `a += b` в обратном порядке.

Отрицательные числа

До настоящего момента мы работали только со сложением и вычитанием положительных целых чисел. Но как представить отрицательные числа в регистре QPU и работать с ними? К счастью, мы можем воспользоваться двоичным представлением *дополнительного кода*, используемым всеми современными процессорами и языками программирования. Ниже приводится краткий обзор дополнительного кода, ориентированного на работу с кубитами.

Для заданного количества битов можно просто запутать одну половину значений с отрицательными числами, а другую — с положительными. Например, трехразрядный регистр позволяет представить целые числа –4, –3, –2, –1, 0, +1, +2 и +3 в соответствии с табл. 5.1.

Таблица 5.1. Двоичное представление отрицательных чисел в дополнительном коде

0	1	2	3	–4	–3	–2	–1
000	001	010	011	100	101	110	111

Если вы еще никогда не сталкивались с дополнительным кодом, связь между отрицательными и двоичными значениями может показаться несколько хаотичной, но у этого конкретного выбора есть неожиданное преимущество: способы выполнения элементарных арифметических операций, разработанные для положительных чисел, также будут работать с представлениями в дополнительном коде. Также из табл. 5.1 видно, что старший бит удобно использовать для представления знака целого числа.

Дополнительный код работает с регистрами, состоящими из кубитов, точно так же, как и с регистрами, состоящими из битов. Таким образом, все примеры этой главы нормально работают с отрицательными значениями, представленными в дополнительном коде. Конечно, необходимо внимательно следить за тем, кодируются данные в регистрах QPU в дополнительном коде или нет, чтобы правильно интерпретировать их двоичные значения.

Чтобы изменить знак числа в дополнительном коде, следует просто инвертировать всего его биты, а затем прибавить 1[1]. Квантовые операции для решения этой задачи показаны на рис. 5.9, который сильно напоминает реализацию оператора инкремента на рис. 5.3.

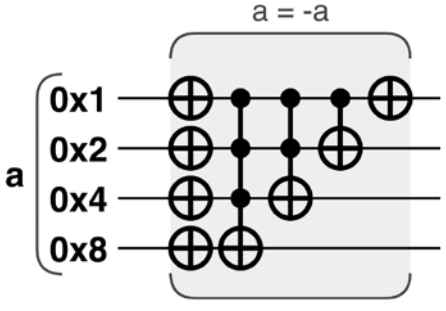


Рис. 5.9. Изменение знака в дополнительном коде: все биты инвертируются, и к числу прибавляется 1

Практический пример: более сложные вычисления

Не все арифметические операции легко адаптируются к требованиям, выдвигаемым к операциям QPU (в частности, обратимости и невозможности копирования). Например, умножение трудно реализовать в обратимом виде. Код в листинге 5.3, представленный на рис. 5.10, демонстрирует

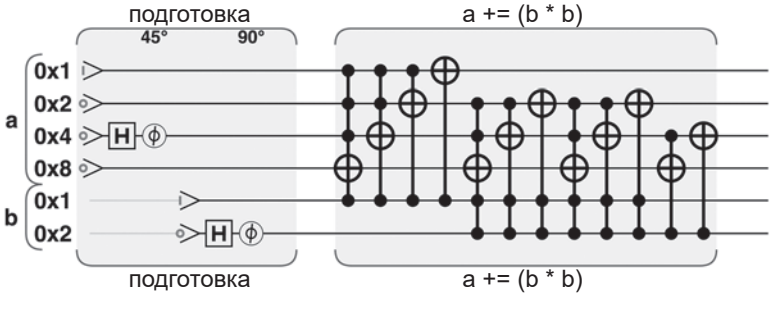


Рис. 5.10. Прибавление квадрата b к a

[1] В трехразрядном примере этот способ работает для всех значений, кроме –4. Как показано в табл. 5.1, представления для 4 нет, поэтому при изменении знака число остается без изменений.

связанную операцию, которая может быть построена как обратимая. А если говорить по сути, мы возводим в квадрат одно значение и прибавляем результат к другому.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=5-3.

Листинг 5.3. Интересные арифметические операции

```
// a += b * b
a.addSquared(b);
```

В результате выполнения операции `a+=b*b` (рис. 5.10) квадрат `b` прибавляется к `a`.

Как работает схема на рис. 5.10? Эта реализация выполняет умножение многократным сложением, управляемым битами *b*.

Обратимость — особенно интересный аспект этого примера. Если вы попытаетесь наивно реализовать `b=b*b`, то быстро обнаружите, что подходящей комбинации обратимых операций не существует, и знаковый бит всегда будет теряться. Однако с реализацией `a+=b*b` все нормально, так как для ее обращения достаточно операции `a-=b*b`.

Переход на квантовый уровень

Итак, наш инструментарий пополнился квантовыми версиями арифметических схем, которые открывают совершенно новые возможности.

Квантовое условное выполнение

На традиционных компьютерах под *условным выполнением* понимается выполнение логики только в том случае, если установлено некоторое управляющее значение. В ходе выполнения процессор читает значение и решает, должна ли выполняться логика. В QPU значение в суперпозиции одновременно установлено и не установлено, поэтому операция, зависящая от этого значения, должна одновременно выполняться и не выполняться. Мы

можем поднять эту идею на более высокий уровень и условно выполнять большие блоки цифровой логики в суперпозиции.

Для примера возьмем программу из листинга 5.4, которая инкрементирует регистр из трех кубитов с меткой `b` (рис. 5.11) — но только в том случае, если другой трехкубитный регистр `a` содержит целочисленное значение в определенном диапазоне. Если `a` инициализируется в суперпозиции $|1\rangle$ и $|5\rangle$, `b` оказывается в суперпозиции выполнения и невыполнения инкремента.

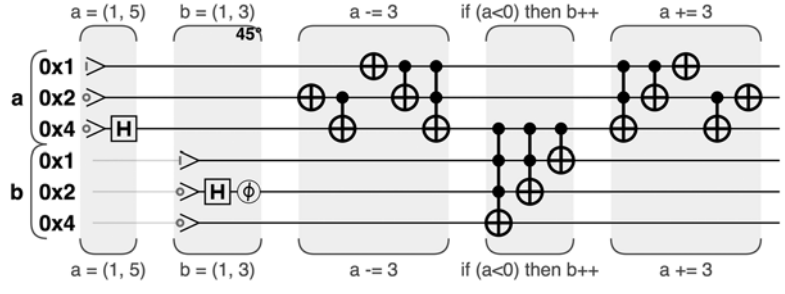


Рис. 5.11. Условное выполнение

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=5-4.

Листинг 5.4. Условное выполнение

```
a.subtract(3);
// Если старший бит a установлен, то b += 1
b.add(1, a.bits(0x4));
a.add(3);
```

Для некоторых значений `a` вычитание 3 из `a` приведет к установке кубита с наименьшим весом. Мы можем использовать старший кубит в качестве управляющего для операции инкремента. После инкремента `b`, управляемого значением этого кубита, необходимо снова прибавить 3 к `a`, чтобы восстановить исходное состояние.

При выполнении листинга 5.4 круговая запись показывает, что инкрементируются только части квантового состояния, в которых `a` меньше 3 или больше 6.

Логика инкремента влияет только на круги в столбцах 0, 1, 2 и 7 на рис. 5.12.

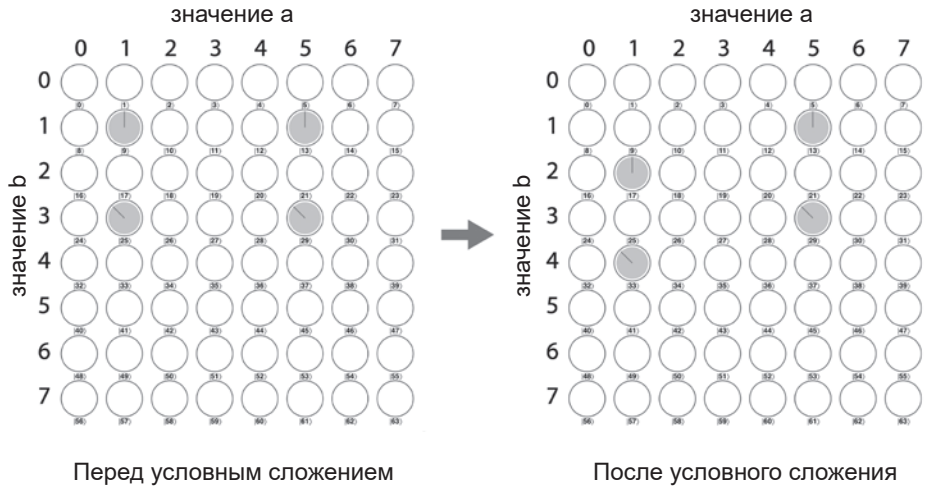


Рис. 5.12. Условное сложение

Фазовое кодирование результата

Наши QPU-версии арифметических операций также можно модифицировать так, чтобы результат вычислений кодировался в относительных фазах исходного входного кубитного регистра — с традиционными битами это абсолютно невозможно. Кодирование результатов вычислений в относительных фазах регистра является одним из важнейших навыков программирования QPU; оно поможет получать ответы, способные пережить операции READ.

В листинге 5.5 (рис. 5.13) приведена измененная версия листинга 5.4, которая изменяет *фазу* во входном регистре, если `a` меньше 3 и `b` равно 1.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=5-5.

Листинг 5.5. Фазовое кодирование результата

```
// Инвертирование фазы, если a меньше 3, а b равно 1
a.subtract(3);
b.not(~1);
qc.phase(180, a.bits(0x4), b.bits());
b.not(~1);
a.add(3);
```

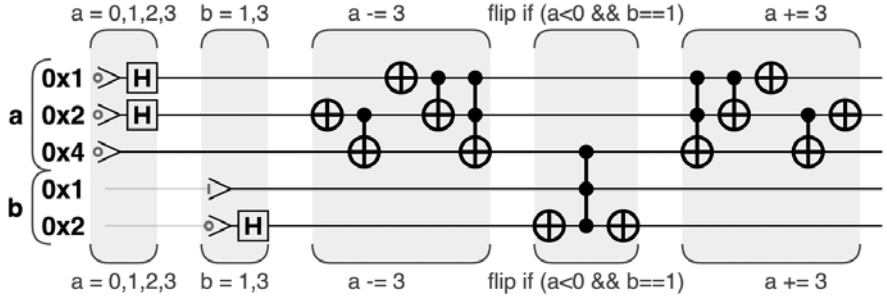


Рис. 5.13. Фазовое кодирование результата

После завершения этой операции амплитуды регистра не изменяются. Вероятности чтения каждого значения операцией READ остаются теми же, что и прежде; по ним невозможно определить, что операция когда-либо выполнялась. Однако из рис. 5.14 видно, что фазы входного регистра были использованы для «пометки» конкретных состояний суперпозиции как результата вычислений. Этот эффект проще всего рассмотреть в круговой записи.

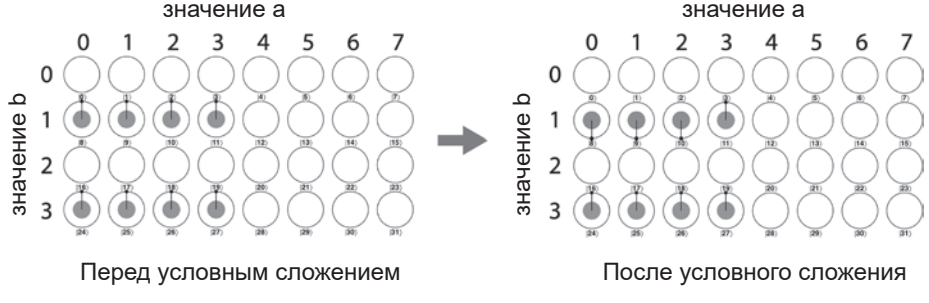


Рис. 5.14. Эффект фазового кодирования

Эта возможность будет интенсивно использоваться для вычислений в фазах регистра в главе 10.

Обратимость и служебные кубиты

В этой главе мы снова и снова напоминаем, что операции QPU должны быть обратимыми. Естественно спросить: «Как гарантировать, что арифметические операции, которые я хочу выполнить, обратимы?» И хотя не

существует четко определенных механизмов преобразования арифметической операции в обратимую форму (которая с большей вероятностью будет работать на QPU), один из полезных методов основан на использовании *служебных кубитов*.

Служебные кубиты не нужны для кодирования ввода или вывода, который вас интересует; они выполняют временную функцию и делают возможной квантовую логику, которая их связывает.

Ниже приведен конкретный пример необратимой операции, которую можно сделать обратимой при помощи служебного кубита. Допустим, вы пытаетесь создать QPU-реализацию `abs(a)` — функцию, вычисляющую абсолютное значение целого числа со знаком (рис. 5.15).

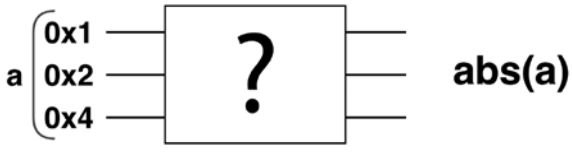


Рис. 5.15. Какие операции QPU могут вычислить абсолютное значение?

На рис. 5.9 уже было показано, как легко изменить знак целого числа в регистрах QPU. Казалось бы, реализовать `abs(a)` будет так же легко — достаточно изменить знак регистра QPU в зависимости от его собственного знакового бита. Но любые попытки такого рода не будут обратимыми (что вполне ожидаемо, так как математическая функция `abs(a)` разрушает информацию о знаке входных данных). Это не означает, что вы получите ошибку компиляции QPU или ошибку времени выполнения; проблема в том, что, как бы вы ни старались, вам не удастся найти конфигурацию обратимых операций QPU, которая могла бы произвести нужные результаты.

На помощь приходит служебный кубит! Он используется для хранения знака целого числа в `a`. Сначала он инициализируется $|0\rangle$, а затем операция `CNOT` используется для переключения служебного кубита в зависимости от кубита с наибольшим весом в регистре `a`. Затем выполняется изменение знака, управляемое значением служебного кубита (а не знаком самого регистра `a`), как показано на рис. 5.16.

С использованием служебного кубита мы теперь *можем* найти набор вентилей, который в конечном итоге реализует операцию `abs` с регистром `a`. Но прежде чем досконально убедиться, что схема на рис. 5.16 решает эту задачу, стоит заметить, что операция `CNOT`, используемая между служебным кубитом и кубитом `0x4` регистра `a` (который сообщает знак целого числа в до-

полнительном коде), не выполняет точного *копирования* знакового кубита. Вместо этого CNOT запутывает служебный кубит со знаковым кубитом. Это весьма важное различие, так как мы уже знаем, что операции QPU не могут копировать кубиты.

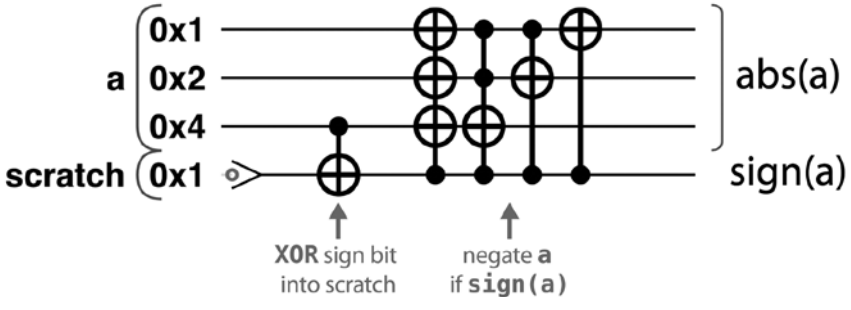


Рис. 5.16. Служебные кубиты могут сделать необратимую операцию обратимой

На рис. 5.17 продемонстрирован ход выполнения этих операций с использованием поучительного примера — состояния с различными относительными фазами просто для того, чтобы вам было проще следить за перемещениями кругов при различных операциях QPU.

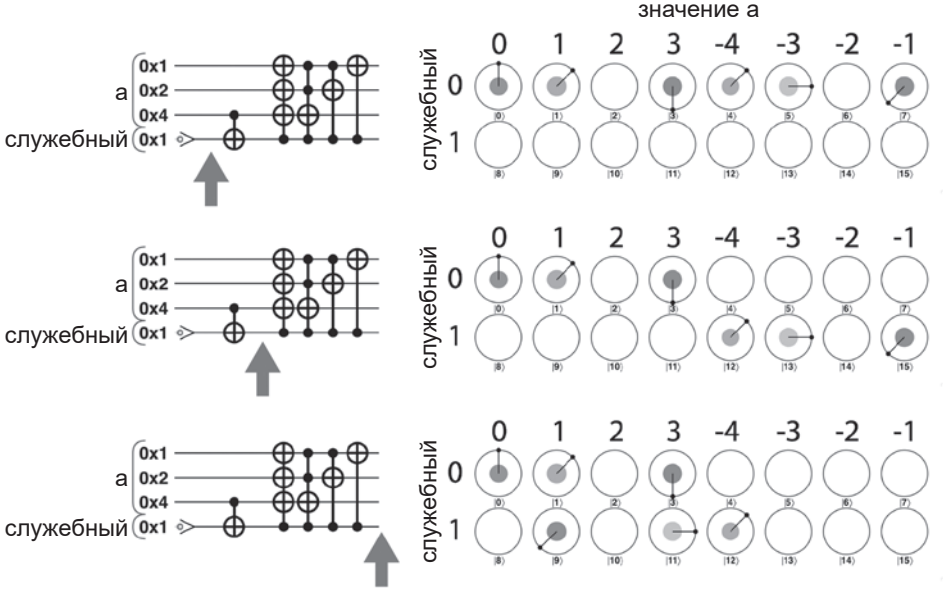


Рис. 5.17. Шаги круговой записи для вычисления модуля

Обратите внимание на то, что каждое значение в суперпозиции, соответствующее `a` в исходной положительным или нулевым значением, остается неизменным. Однако значения, для которых `a` имеет отрицательное значение, сначала перемещается во вторую строку кругов (в соответствии с условным переключением служебного кубита), а затем в соответствующие положительные значения (как требует `abs`), оставаясь в строке «служебный = 1»[1].

Отслеживание действия служебного кубита в круговой записи наглядно демонстрирует, как он решает наши проблемы необратимости. Если вы попытаетесь выполнить `abs` без служебного кубита, круговая запись на рис. 5.17 будет состоять только из одной строки, и попытки переместить круги отрицательных значений в соответствующие положительные значения приведут к разрушению амплитуд и фаз, уже хранящихся здесь, и позднее вы никак не сможете определить исходное состояние (то есть операция будет необратимой). Служебный кубит создает дополнительную строку, в которую можно перейти, а затем перемещаться внутри нее, оставляя все исходное состояние неизменным и восстановимым[2].

Отмена вычислений

Хотя служебные кубиты часто необходимы, они обычно связываются с регистрами QPU — а если говорить точнее, входят в состояние квантовой запутанности. При этом возникают две проблемы. Во-первых, служебные кубиты редко оказываются в чисто нулевом состоянии. И это плохо, потому что теперь эти служебные кубиты приходится сбрасывать, чтобы они могли снова использоваться в дальнейшей работе QPU.

Кто-то подумает: «В чем проблема? Я просто применю к ним `READ` и `NOT` при необходимости, как было показано в главе 1». Но тогда вы столкнетесь со второй проблемой. Поскольку использование служебных кубитов почти всегда приводит к их запутанности с кубитами выходного регистра (как, например, на рис. 5.17), выполнение операции `READ` с ними может иметь катастрофические последствия для выходного состояния, которое они помогли создать. Вспомните, о чем говорилось в главе 3: когда кубиты находятся в запутанном состоянии, любые операции, выполняемые с одним из них, неизбежно воздействуют на другие. В примере на рис. 5.17 попытка

---

[1] Обратите внимание: в этом случае значение –4 остается неизменным, что правильно для трехбитового числа в дополнительном коде.

[2] Это серьезное достижение. Отслеживание исходного состояния с использованием традиционных битов потребует экспоненциального количества битов, но добавление лишь одного служебного кубита идеально решает проблему.

сбросить служебный кубит в состояние $|0\rangle$ (что необходимо сделать, если вы захотите снова использовать его в QPU) разрушает половину квантового состояния `a`!

К счастью, существует прием, решающий эту проблему: он называется *отменой вычислений* (uncomputing). Концепция отмены вычислений заключается в обращении операций, приведших к запутанности служебного кубита, и возврате его в исходное незапутанное состояние $|0\rangle$. В примере с `abs` это означает обращение всей логики `abs(a)` с участием `a` и служебного кубита. Великолепно! Служебный кубит снова находится в состоянии $|0\rangle$. К сожалению, мы, конечно, полностью потеряли всю работу по вычислению абсолютного значения.

Какая польза от того, что мы получили обратно свой служебный кубит, если при этом также утратили свой ответ? Из-за невозможности копирования кубитов мы даже не можем скопировать значение, хранящееся в другом регистре, перед отменой вычислений. Однако причина, по которой отмену вычислений нельзя назвать полностью безрезультатной, заключается в том, что ответ часто используется в регистре `a` перед отменой. Обычно такие функции, как `abs`, используются как часть более крупных арифметических вычислений — например, в вычислении выражения «прибавить абсолютное значение `a` к `b`». Эту операцию можно сделать инвертируемой и сохранить служебный кубит при помощи схемы, изображенной на рис. 5.18.

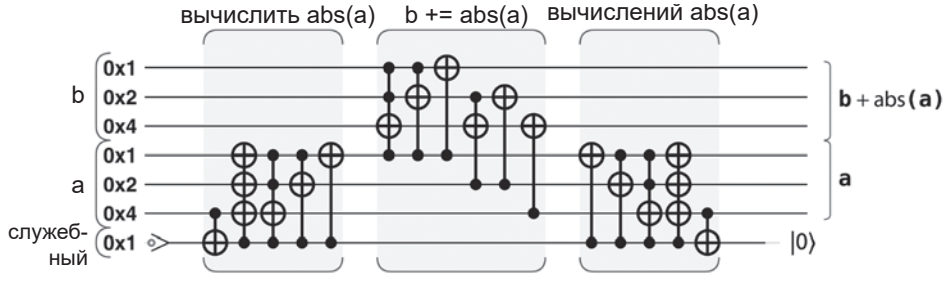


Рис. 5.18. Использование отмены вычислений для выполнения операции b+=abs(a)

После таких операций `a` и `b` с большой вероятностью будут находиться в состоянии запутанности; тем не менее служебный кубит вернулся в состояние $|0\rangle$ без запутанности и готов к использованию в другой операции. Большая операция *обратима* даже при том, что о самой операции `abs` этого сказать нельзя. В квантовых вычислениях этот прием встречается очень часто: временные служебные кубиты используются для необратимых

в остальном вычислений, затем выполняются другие операции, зависящие от результата, после чего происходит отмена вычислений.

Хотя мы не можем просто скопировать абсолютное значение в другой регистр перед отменой вычислений, можно добиться похожего эффекта, инициализируя b значением 0 перед сложением на рис. 5.18. В действительности того же результата можно достичь простым использованием операции CNOT вместо сложения, как на рис. 5.19.

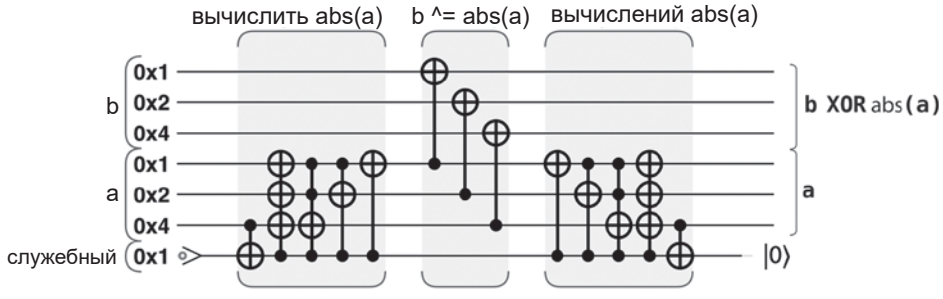


Рис. 5.19. Использование отмены вычислений для получения b xor abs(a)

В другом чрезвычайно распространенном применении отмены вычислений выполняются некоторые вычисления (возможно, с использованием служебных кубитов), результаты которых сохраняются в относительных фазах выходного регистра, после чего вычисленные результаты отменяются. При условии, что исходные вычисления (а следовательно, также итоговый шаг отмены вычислений) не нарушают относительные фазы выходных регистров, они переживут процесс без искажений. Этот прием будет использован в главе 6.

Например, схема на рис. 5.20 выполняет операцию «инвертировать фазу любого значения, для которого `abs(a)` равно 1». Абсолютное значение

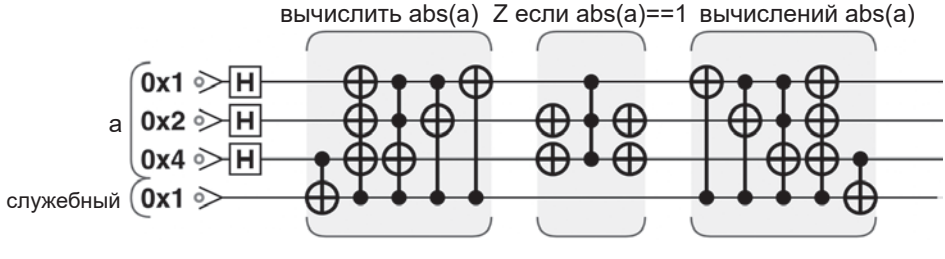


Рис. 5.20. Использование отмены вычислений для выполнения операции phase(180) при условии if abs(a)==1

вычисляется с использованием служебного кубита, фаза выходного регистра инвертируется только в том случае, если значение равно 1, после чего происходит отмена вычислений.

На рис. 5.21 эта программа проанализирована шаг за шагом в круговой записи.

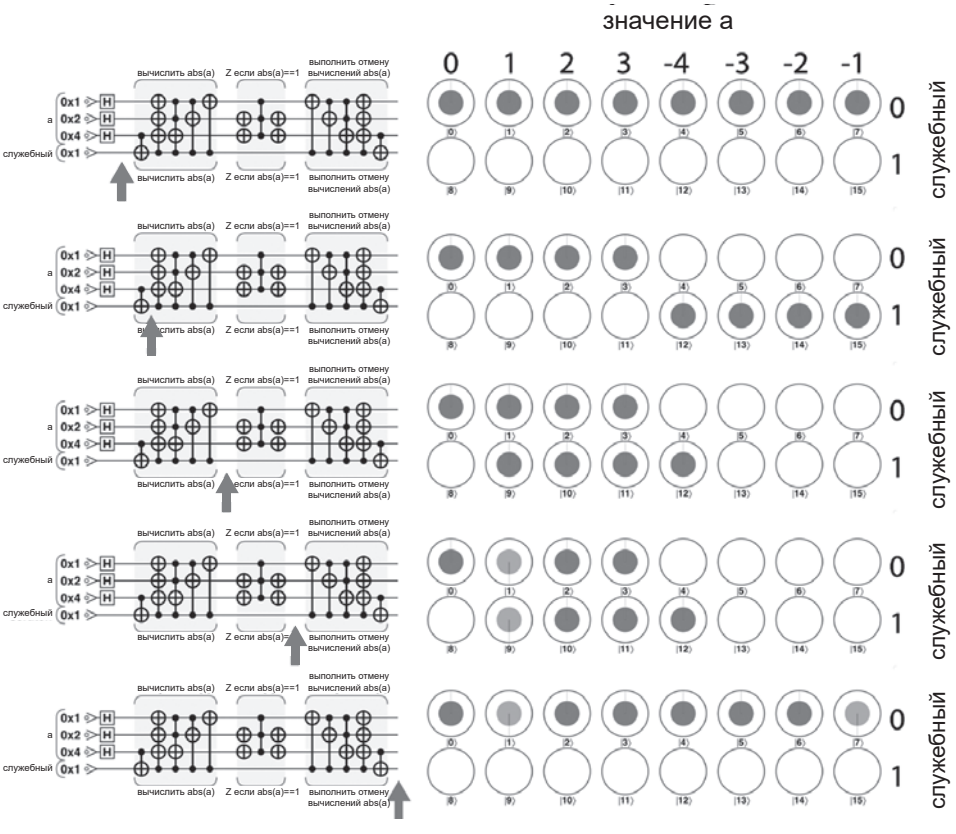


Рис. 5.21. Пошаговое применение отмены вычислений для условного инвертирования фазы

Отображение булевой логики на операции QPU

Как известно, цифровая арифметика строится из цифровых логических вентилей. Чтобы подробно разобраться в том, как работают схемы базовой

арифметики QPU, мы будем использовать инструмент программируемой квантовой логики. В этом разделе будут представлены квантовые аналогии некоторых низкоуровневых цифровых логических вентилей.

Базовая квантовая логика

В цифровой логике существуют базовые логические вентили, из которых могут строиться все остальные. Например, если у вас имеется вентиль NAND, вы можете использовать его для создания вентилей AND, OR, NOT или XOR, которые можно объединить в любую логическую функцию на ваше усмотрение.

Обратите внимание: вентили NAND на рис. 5.22 могут иметь любое количество входов. С одним входом (простейший случай) NAND выполняет операцию NOT.

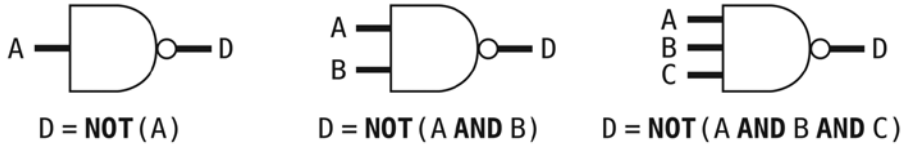


Рис. 5.22. Цифровые вентили NAND разных размеров

В квантовых вычислениях тоже можно начать с одного многоцелевого вентиля (рис. 5.23) и построить квантовую цифровую логику на этой базе. Для этого мы воспользуемся вентилем CNOT с несколькими условиями — вентилем Тоффоли. Как и в случае с вентилем NAND, количество условных входов можно менять для расширения выполняемой логики, как показано в листинге 5.6.

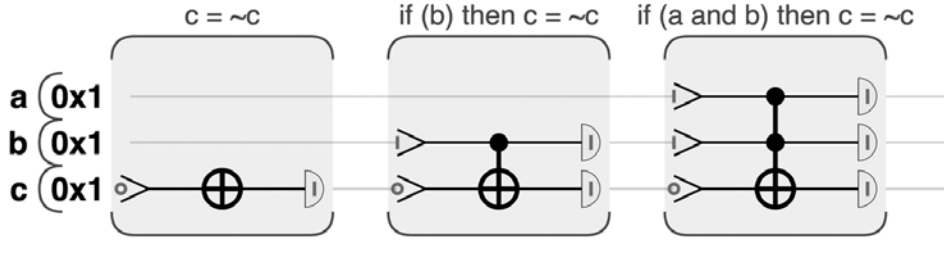


Рис. 5.23. Квантовые вентили CNOT разных размеров

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=5-6.

Листинг 5.6. Логика, основанная на вентилях CNOT

```
// c = ~c
c.write(0);
c.not();
c.read();

// if (b) then c = ~c
qc.write(2, 2|4);
c.cnot(b);
qc.read(2|4);

// if (a and b) then c = ~c
qc.write(1|2);
qc.cnot(4, 1|2);
qc.read(1|2|4);
```

Стоит заметить, что в данном случае выполняется почти та же операция, что и NAND. Эквивалентную цифровую логику можно выразить с использованием операций AND и XOR, как показано на рис. 5.24.

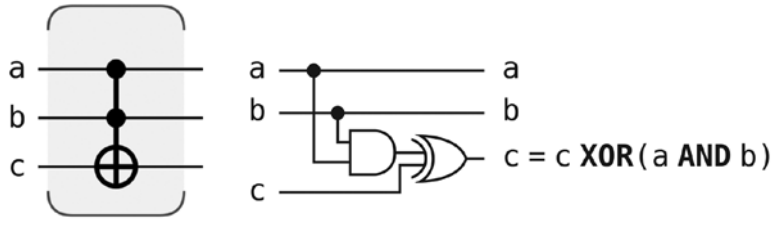


Рис. 5.24. Точный эквивалент вентиля CNOT с несколькими условиями в цифровой логике

Вооружившись этим вентилем, мы сможем создавать QPU-эквиваленты для широкого спектра логических функций (рис. 5.25).

В некоторых случаях для реализации нужной логической функции необходимо добавить дополнительный служебный кубит, инициализированный состоянием $|0\rangle$. Одной из серьезных проблем в квантовых вычислениях является сокращение количества служебных кубитов, необходимых для выполнения конкретного вычисления.

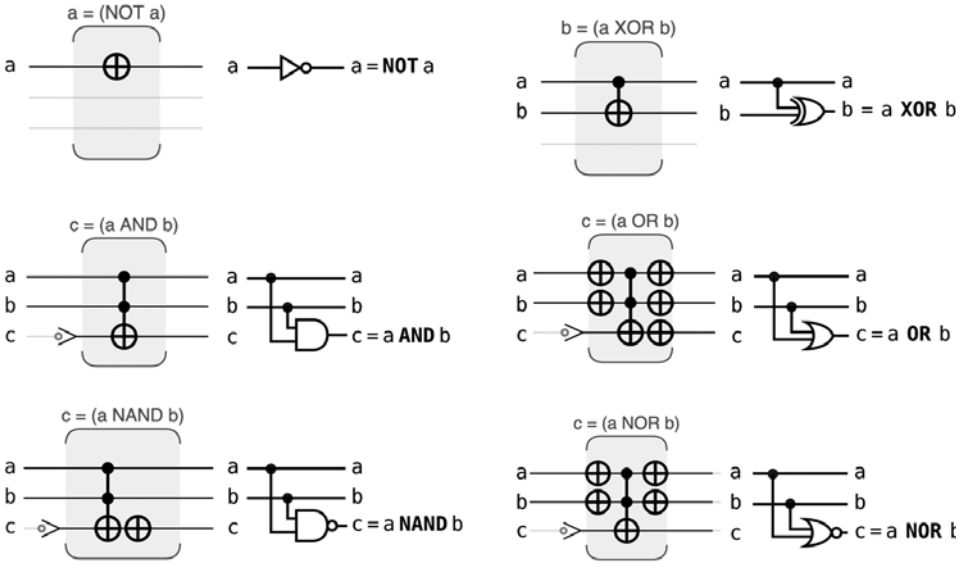


Рис. 5.25. Некоторые базовые вентили цифровой логики, построенные из вентилей CNOT с несколькими условиями

Итоги

Возможность выполнения цифровой логики в суперпозиции занимает центральное место во многих алгоритмах QPU. В этой главе были рассмотрены некоторые возможности работы с квантовыми данными и даже выполнения условных операций с суперпозициями.

Польза от выполнения цифровой логики в суперпозиции будет довольно ограниченной, если только мы не научимся извлекать полезную информацию из полученного состояния. Вспомните, что при попытке применить READ к суперпозиции решений арифметической задачи мы случайным образом получим только одно из них. В следующей главе будет рассмотрен примитив QPU, позволяющий надежно извлекать выходные данные из квантовых суперпозиций — *усиление комплексной амплитуды*.

6 Усиление комплексной амплитуды

В предыдущей главе мы показали, как строятся традиционные арифметические и логические операции, использующие мощь суперпозиции. Но при использовании QPU возможность выполнения вычислений в суперпозиции бесполезна, если у вас в запасе нет какого-нибудь хитрого трюка, который бы позволил прочитать решение операцией READ.

В этой главе представлен первый квантовый примитив, позволяющий манипулировать суперпозициями в форме, пригодной для надежного выполнения READ. Мы уже упоминали о том, что существует много таких примитивов, подходящих для различных типов задач. Начнем мы с рассмотрения *усиления комплексной амплитуды*[1].

Практический пример: преобразование между фазой и амплитудой

В очень простой формулировке усиление комплексной амплитуды представляет собой инструмент, преобразующий недоступные разности *фазы* в регистре QPU в разности *амплитуды*, которые могут читаться операцией READ (и наоборот). Это простой, элегантный, мощный и очень полезный инструмент QPU.

Допустим, у вас имеется четырехкубитный регистр QPU, который содержит одно из трех квантовых состояний (A, B или C), как на рис. 6.1, но мы не знаем, какое именно.

---

[1] В этой книге термин «усиление комплексной амплитуды» используется в несколько ином контексте, чем в академической литературе. Различия будут более подробно описаны в главе 14.

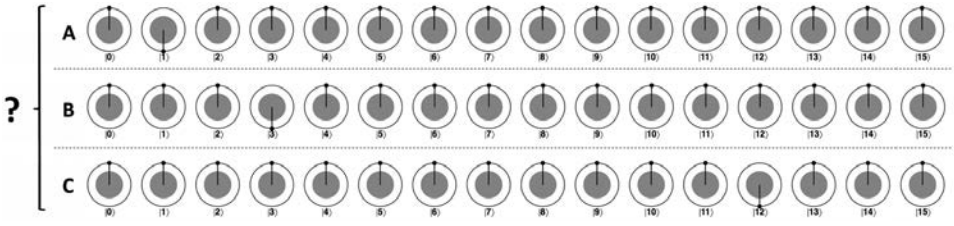


Рис. 6.1. Каждое состояние содержит одно значение с инвертированной фазой

Эти три состояния очевидно выделяются тем, что в каждом из них присутствует некоторое значение с инвертированной фазой. Назовем его *помеченным значением*. Но поскольку все значения в регистре имеют одинаковую амплитуду, при чтении регистра QPU в любом из этих состояний будет получено случайное число с равномерным распределением, и вы ничего не узнаете о том, какое из трех состояний было исходным. В то же время такие операции READ разрушают информацию фазы в регистре.

Но всего одна процедура QPU может раскрыть информацию скрытой фазы. Назовем эту процедуру *зеркальной операцией* (вскоре вы поймете почему); чтобы увидеть ее в действии, выполните пример кода из листинга 6.1 (рис. 6.2).

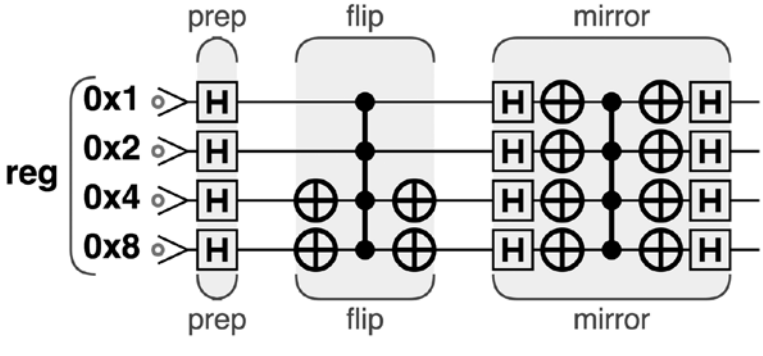


Рис. 6.2. Применение зеркальной операции к инвертированной фазе

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=6-1.

Листинг 6.1. Применение зеркальной операции к инвертированной фазе

```
var number_to_flip = 3; // 'Помеченное' значение

var num_qubits = 4;
qc.reset(num_qubits);
var reg = qint.new(num_qubits, 'reg')
reg.write(0);
reg.hadamard();

// Помеченное значение инвертируется
reg.not(~number_to_flip);
reg.cphase(180);
reg.not(~number_to_flip);
reg.Grover();
```

Обратите внимание: перед применением зеркальной операции сначала выполняется инвертирование фазы; оно берет регистр, изначально находящийся в состоянии $|0\rangle$, и *помечает* одно из его значений. Чтобы изменить, какое именно значение будет инвертировано в этом коде, измените переменную `number_to_flip`.

При применении зеркальной операции к состояниям A, B и C на рис. 6.1 будут получены результаты, показанные на рис. 6.3.

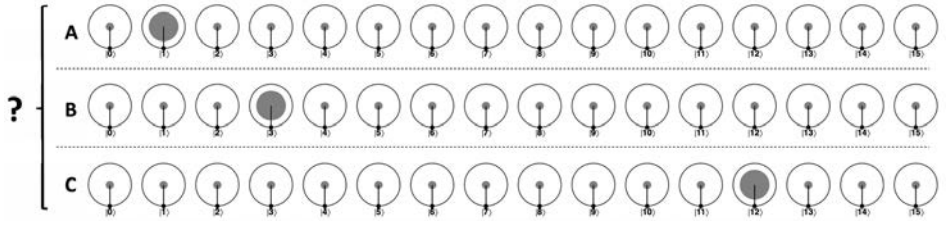


Рис. 6.3. После применения зеркальной операции разности фазы преобразуются в разности амплитуды

Теперь амплитуды в каждом состоянии очень сильно различаются, и выполнение операции READ с регистром QPU с очень большой вероятностью (хотя и не 100%-ной) покажет, у какого значения была инвертирована фаза — а следовательно, в каком из трех состояний находился регистр. Вместо одинаковой вероятности 6,25% у всех значений помеченное значение теперь обладает вероятностью READ около 47,3%, тогда как у непомеченных значений она равна приблизительно 3,5%. В этот момент чтение регистра дает почти 50%-ную вероятность получения значения, у которого была инвертирована фаза. Впрочем, результат все еще не идеален.

Хотя зеркальная операция изменила амплитуду помеченного значения, его фаза остается такой же, как у остальных значений регистра. В каком-то смысле зеркальная операция *преобразовала* разности фазы в разности амплитуды.

В литературе, посвященной квантовым вычислениям, зеркальная операция обычно называется оператором итерации Гровера. Алгоритм Гровера поиска в неструктурированной базе данных стал первым алгоритмом, реализующим операцию зеркального переключения, а примитив усиления комплексной амплитуды, рассмотренный здесь, является обобщением исходного алгоритма Гровера. Мы решили назвать операцию зеркальной, чтобы читателю было проще запомнить ее эффект.

Можно ли повторить операцию, чтобы повысить вероятность успеха? Возьмем состояние B на рис. 6.1 (то есть инвертирование воздействует на значение $|3\rangle$). Повторное применение зеркальной операции просто возвращает нас к исходному состоянию: разности амплитуд снова преобразуются в разности фазы. Но допустим, перед повторным применением зеркальной операции также будет повторно применена операция инвертирования фазы (для повторного инвертирования помеченного значения). В результате перед вторым применением зеркальной операции появляется еще одна разность фазы. На рис. 6.4 показано, что мы получим при двукратном применении *всей* комбинации инвертирования/зеркальной операции.

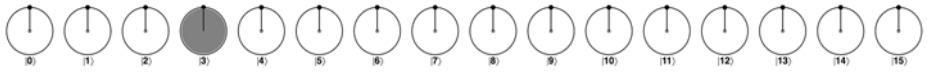


Рис. 6.4. После повторного применения инвертирования/зеркальной операции к состоянию B

После двух применений инвертирования/зеркальной операции вероятность успешного обнаружения помеченного значения повысилось с 47,3% до 90,8%!

Итерация усиления комплексной амплитуды

Инвертирование фазы и зеркальная операция образуют мощную комбинацию. Инвертирование позволяет выбрать значение регистра и выделить его фазу на фоне остальных. Затем зеркальная операция преобразует эту разность фазы в разность амплитуды. Эту комбинированную операцию, показанную на рис. 6.5, мы будем называть итерацией усиления комплексной амплитуды (AA).

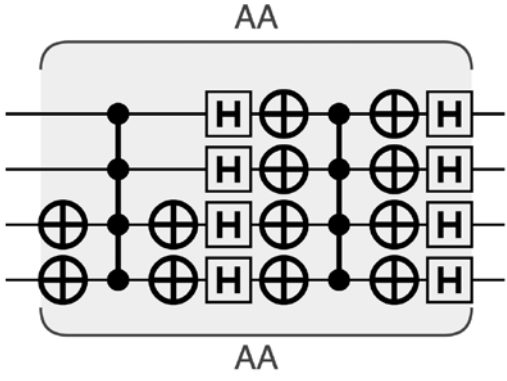


Рис. 6.5. Одна итерация AA

Вероятно, вы заметили: операция AA предполагает, что усиливаемое значение нам уже известно — оно жестко связано со значением, на которое влияет операция инвертирования. Казалось бы, это противоречит самой сути усиления комплексной амплитуды — если вы уже знаете, какие значения нужно усиливать, зачем тогда искать их? Мы использовали операцию инвертирования как простейший возможный прием операции, инвертирующей фазу выбранных значений в регистре QPU. В реальном приложении инвертирование будет заменено более сложной операцией, выполняющей комбинацию переключений фазы, которая представляет логику данного приложения. В главе 10 мы покажем подобнее, как вычисления могут выполняться исключительно с фазами регистра QPU. Когда приложения используют *фазовую логику* такого рода, они могут занять место инвертирования, а усиление комплексной амплитуды становится исключительно полезным инструментом.

Ключевой момент заключается в том, что, хотя мы обсуждаем выполнение инвертирования наряду с зеркальной операцией, комбинация более сложных операций изменения фазы с зеркальной операцией тоже преобразует изменение фазы в разность амплитуды.

Больше итераций?

На рис. 6.4 две итерации AA были применены к состоянию B, что дало 90,8% вероятность успеха наблюдения за помеченным значением. Можно ли продолжить применять итерации AA, чтобы вероятность стала еще ближе к 100%? Это несложно определить. Пример кода в листинге 6.2 повторяет операцию AA заданное количество раз. Изменяя переменную `number_`

of_iterations в коде, можно выполнить столько итераций AA, сколько потребуется.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=6-2.

Листинг 6.2. Повторение итераций усиления комплексной амплитуды

```
var number_to_flip = 3;
var number_of_iterations = 4;

var num_qubits = 4;
qc.reset(num_qubits);
var reg = qint.new(num_qubits, 'reg')
reg.write(0);
reg.hadamard();

for (var i = 0; i < number_of_iterations; ++i)
{
    // Инвертировать помеченное значение
    reg.not(~number_to_flip);
    reg.cphase(180);
    reg.not(~number_to_flip);

    reg.Grover();

    // Проверить вероятность
    var prob = reg.peekProbability(number_to_flip);
}
```

На рис. 6.6 показан результат выполнения этого кода для number_of_iterations=4 (таким образом, операция инвертирования и зеркальная операция выполняются последовательно четыре раза).

Рассмотрим результаты каждой последовательной итерации AA. После инвертирования первой итерации AA работа начинается с состояния B на рис. 6.1. Как уже было показано выше, в состоянии B1 вероятность успеха равна 47,3%, тогда как после двух итераций в состоянии B2 вероятность повышается до 90,8%.

Третья итерация достигает вероятности 96,1%, но обратите внимание: в B3 помеченное состояние отличается по фазе от других, поэтому после следующего инвертирования все фазы будут *синхронизированы*. В этой точке будет присутствовать разность амплитуды, но разности фаз не будет, так

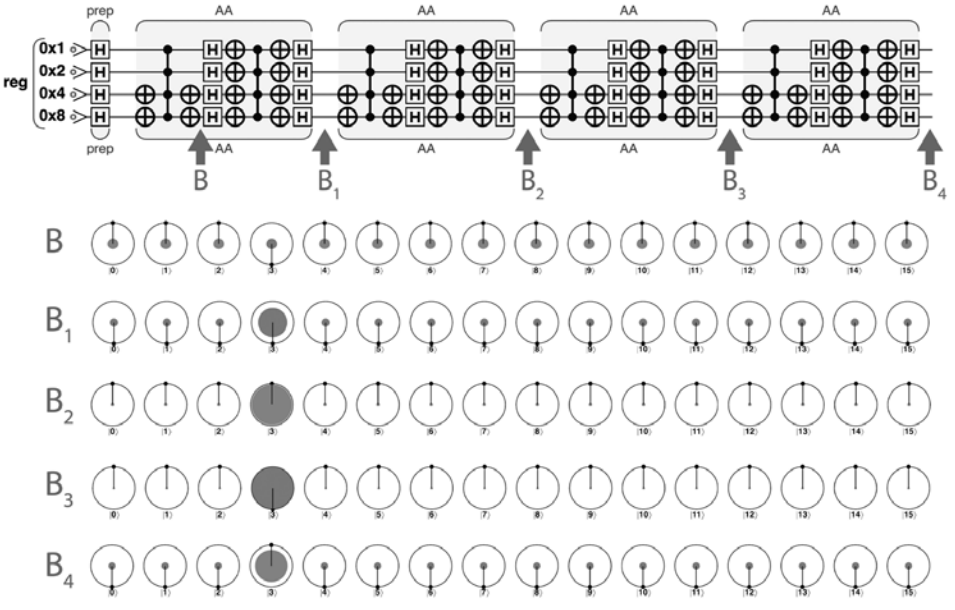


Рис. 6.6. Результат применения AA 1, 2, 3 и 4 раза в состоянии B — в рядах кругов показано состояние регистра QPU в обозначенных точках схемы

что дальнейшие итерации AA начнут уменьшать разности амплитуд, пока мы не вернемся к исходному состоянию.

К тому моменту, когда мы добираемся до B4, вероятность успешного чтения помеченного состояния падает до 58,2%, и продолжит уменьшаться при применении новых итераций AA.

Сколько же итераций AA следует применить для достижения максимальной вероятности правильного чтения помеченного значения операцией READ?

Диаграмма на рис. 6.7 показывает, что при продолжении итераций вероятность чтения помеченного значения колеблется предсказуемым образом. Мы видим, что для достижения максимальной вероятности получения правильного результата лучше дождаться 9-й или 15-й итерации, чтобы вероятность нахождения помеченного значения составила 99,9563%.

С другой стороны, если выполнение каждой итерации обходится дорого (некоторые примеры будут приведены позже), можно остановиться после трех итераций и попытаться получить ответ с вероятностью 96,1%. Даже если попытка будет неудачной и всю программу QPU придется повторять заново, с вероятностью 99,848% одна из двух попыток будет успешной, и для этого потребуется не более 6 итераций.

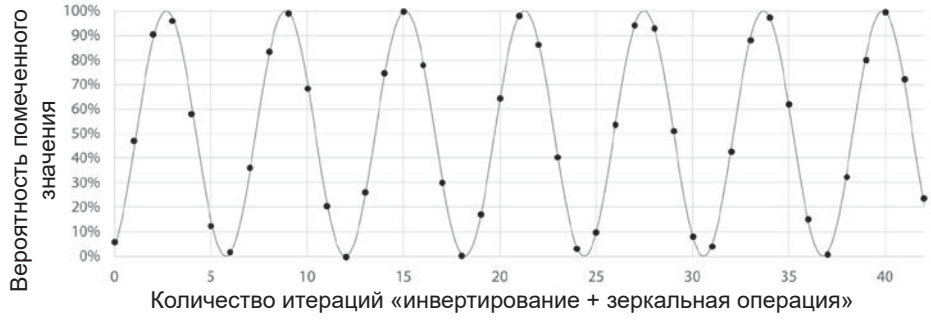


Рис. 6.7. Зависимость вероятности чтения помеченного значения от количества итераций AA

В общем случае существует простая и полезная формула, позволяющая определить количество итераций AA (N_{AA}), которые следует выполнить для обеспечения наивысшей вероятности в пределах первого периода колебания (например, вероятность успеха 96,1% при $N_{AA} = 3$ в предыдущем примере). В формуле 6.1 параметр *n* представляет количество кубитов.

Формула 6.1. Оптимальное количество итераций при усилении комплексной амплитуды

$$N_{AA} = \left\lfloor \frac{\pi\sqrt{2^n}}{4} \right\rfloor.$$

В вашем распоряжении появился инструмент, который способен преобразовать одну разность фазы в регистре QPU в разность амплитуд, которая может быть обнаружена в ходе выполнения. Но что, если регистр содержит несколько значений с разными фазами? Легко представить, что операции более сложные, чем инвертирование, могут изменить значения нескольких значений в регистре. К счастью, итерации AA справятся и с более сложным случаем.

Инвертирование нескольких элементов

При небольшом изменении схемы из листинга 6.1 можно попробовать выполнить несколько итераций AA с регистром, имеющим несколько значений с измененной фазой. В листинге 6.3 можно использовать переменную `n2f` для определения значений регистра, которые должны инвертироваться внутри каждой операции AA. Как и прежде, вы также можете скорректи-

ровать количество выполняемых итераций AA при помощи переменной `number_of_iterations`.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=6-3.

Листинг 6.3. Итерации усиления комплексной амплитуды с инвертированием нескольких значений

```
var n2f = [0,1,2];              // Инвертируемые значения
var number_of_iterations = 50; // Количество итераций Гровера

var num_qubits = 4;
qc.reset(num_qubits);
var reg = qint.new(num_qubits, 'reg')
reg.write(0);
reg.hadamard();

for (var i = 0; i < number_of_iterations; ++i)
{
    // Инвертировать помеченное значение
    for (var j = 0; j < n2f.length; ++j)
    {
        var marked_term = n2f[j];
        reg.not(~marked_term);
        reg.cphase(180);
        reg.not(~marked_term);
    }

    reg.Grover();

    var prob = 0;
    for (var j = 0; j < n2f.length; ++j)
    {
        var marked_term = n2f[j];
        prob += reg.peekProbability(marked_term);
    }
    qc.print('iters: '+i+' prob: '+prob);
}
```

Выполняя этот пример с одним инвертируемым значением (например, `n2f = [4]`, как на рис. 6.8), можно воспроизвести предшествующие результаты, когда повышение количества итераций AA приводило к синусоидальному изменению вероятности чтения «помеченного» значения, как показано на рис. 6.9.

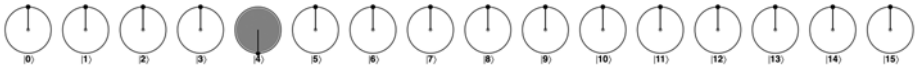


Рис. 6.8. Одно инвертированное значение

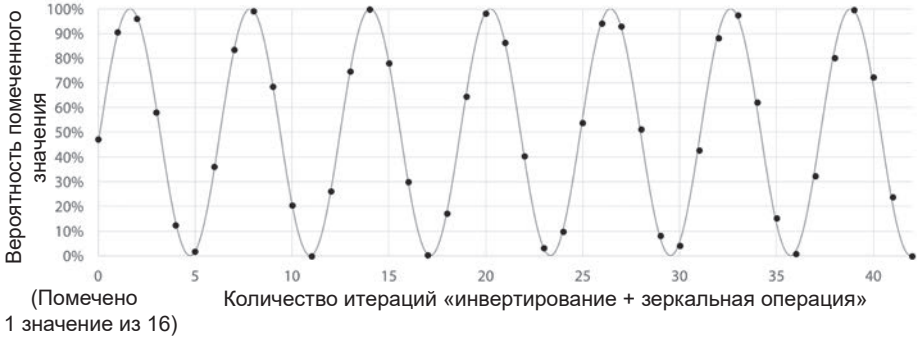


Рис. 6.9. Повторение итераций AA с инвертированием 1 значения из 16

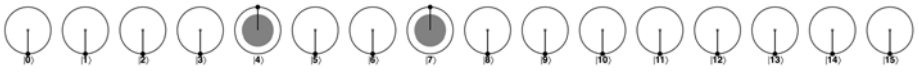


Рис. 6.10. Инвертирование фазы двух значений

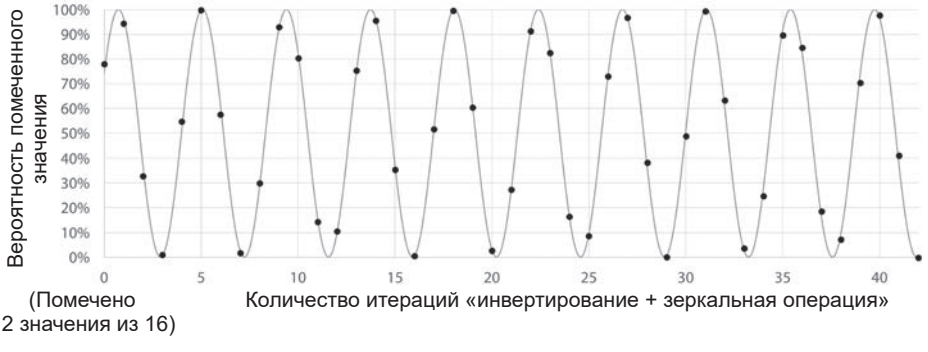


Рис. 6.11. Повторение итераций AA с инвертированием 2 значений из 16

А теперь попробуем инвертировать два значения (например, `n2f=[4,7]`, как на рис. 6.10). На этот раз в идеале регистр QPU должен быть настроен таким образом, чтобы операция READ читала одно из двух значений с инвертированной фазой, при нулевой вероятности чтения любых других. Применение множественных итераций AA, как было сделано для помечен-

ного состояния (хотя и с выполнением двух операций инвертирования при каждой итерации — по одному для каждого помеченного состояния), дает результаты, показанные на рис. 6.11.

Следует заметить, что на рис. 6.11 по оси *y* измеряется вероятность получения любого из (двух) помеченных значений при чтении регистра операцией READ.

Хотя мы и в этом случае получаем синусоидальное изменение вероятности успеха, сравнивая с аналогичным графиком для одного инвертирования фазы на рис. 6.7, можно заметить, что частота синусоиды увеличилась.

С тремя инвертируемыми значениями (например, `n2f=[4,7,8]`, как на рис. 6.12), частота синусоиды продолжает возрастать, как видно из рис. 6.13.

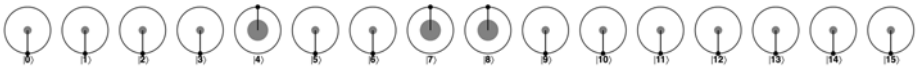


Рис. 6.12. Три инвертированных значения

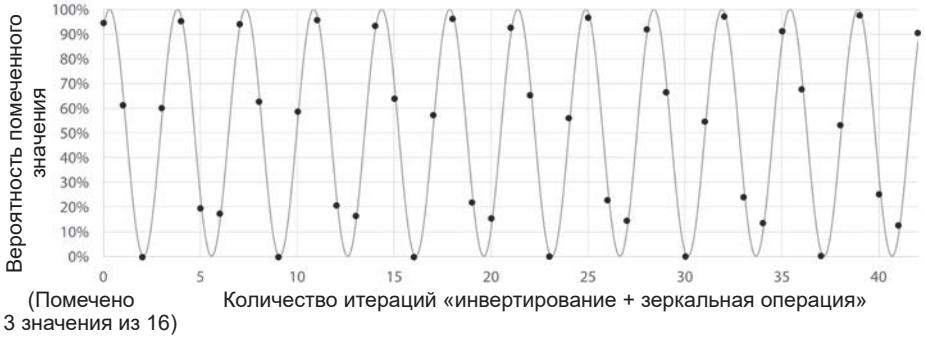


Рис. 6.13. Повторение итераций AA с инвертированием 3 значений из 16

Если инвертируются 4 из 16 значений, как показано на рис. 6.14, происходит нечто интересное. Как видно из рис. 6.15, частота волны становится такой, что вероятность чтения операцией READ одного из помеченных значений повторяется с каждой третьей примененной итерацией AA. В конечном итоге это означает, что самая первая операция дает 100%-ную вероятность успеха.

Эта тенденция продолжается до семи инвертируемых значений, как видно из рис. 6.16 и 6.17.

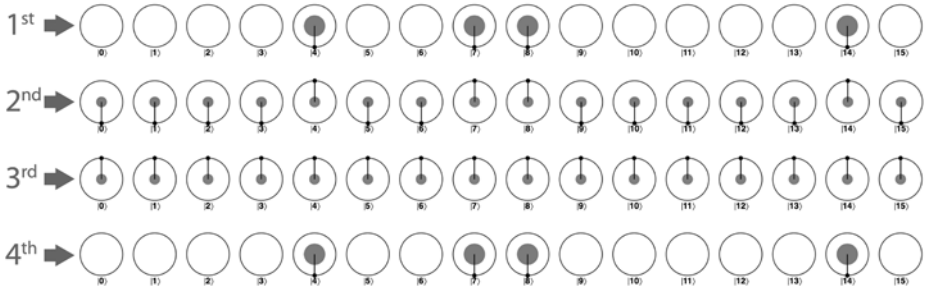


Рис. 6.14. Четыре инвертированных значения

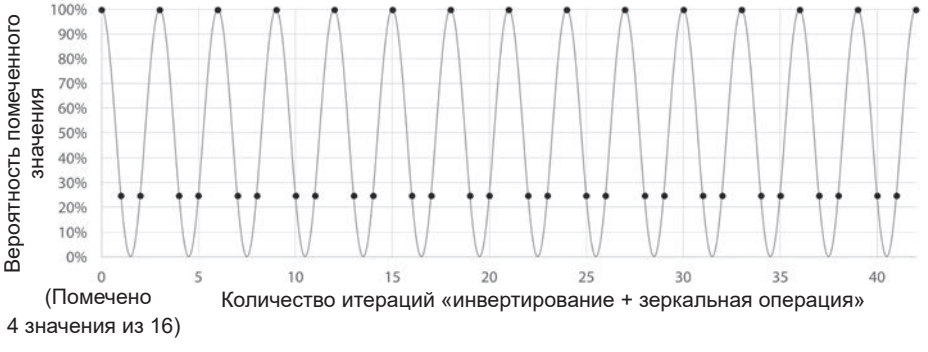


Рис. 6.15. Повторение итераций AA с инвертированием 4 значений из 16

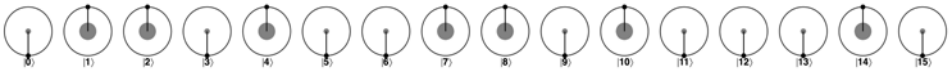


Рис. 6.16. Семь инвертированных значений

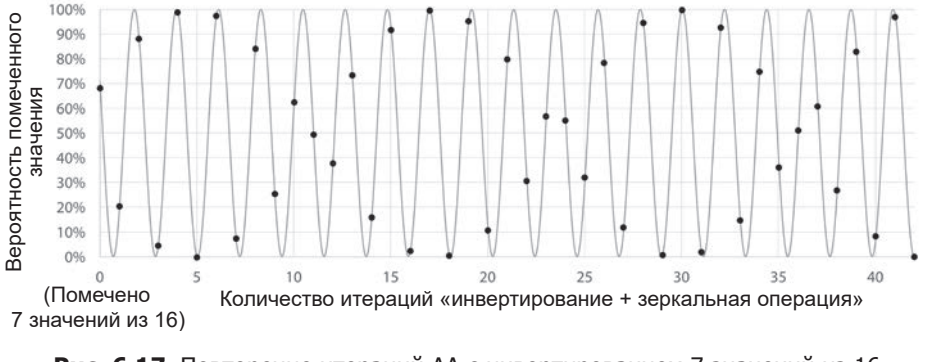


Рис. 6.17. Повторение итераций AA с инвертированием 7 значений из 16

Конечно, с 7 помеченными значениями из 16 даже само получение правильного значения при операции READ может не предоставить сколько-нибудь полезной информации.

Ситуация заходит в тупик на рис. 6.19, где инвертируются 8 из 16 значений (рис. 6.18). Как упоминалось в предыдущих главах, для квантовых состояний важна только *относительная* фаза. По этой причине инвертирование половины значений для их пометки физически не отличается от инвертирования другой половины. В этот момент итерации AA перестают работать, и вероятности чтения любого значения в регистре выравниваются.

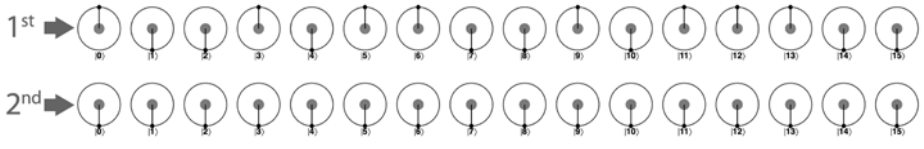


Рис. 6.18. Восемь инвертированных значений

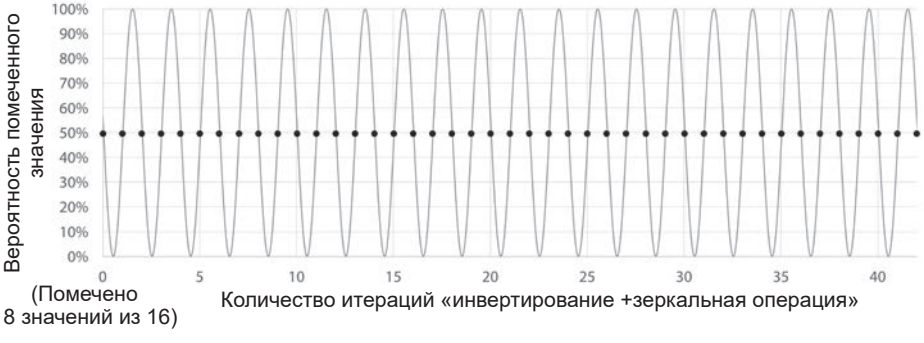


Рис. 6.19. Повторение итераций AA с инвертированием 8 значений из 16

Если помечено 50% значений регистра, то операции AA можно применять сколько угодно — вам никогда не удастся добиться лучших результатов, чем при чтении случайных чисел.

Из-за симметрии нам вообще не нужно задумываться над тем, что произойдет, если фаза инвертирована более чем у 50% значений регистра. Например, если помечены 12 из 16 значений, то при попытке прочитать «успех» получится то же, что наблюдалось при пометке только 4 значений, но с противоположными определениями успеха и неудачи.

Интересно, что наши исследования показывают, что частота колебания вероятности успеха зависит только от количества инвертируемых значений,

а не от того, какие именно значения инвертируются. Формулу 6.1 можно расширить так, чтобы она оставалась справедливой при нескольких помеченных значениях; результат приведен в формуле 6.2 (n — количество кубитов, m — количество помеченных элементов).

Формула 6.2. Оптимальное количество итераций с несколькими инвертированными фазами

$$N_{AA} = \left\lfloor \frac{\pi}{4}\sqrt{\frac{2^n}{m}} \right\rfloor.$$

Если значение m известно, то это выражение может использоваться для определения количества итераций AA, необходимых для усиления амплитуд инвертированных значений с целью обеспечения высокой вероятности успеха. И тут возникает интересный момент: если вы заранее не знаете, сколько состояний инвертировано, как определить, сколько итераций AA потребуется для достижения максимальной вероятности успеха? Когда в главе 10 речь пойдет о применении усиления комплексной амплитуды на практике, мы вернемся к этому вопросу и посмотрим, какую помощь могут оказать другие примитивы.

Использование усиления комплексной амплитуды

Вероятно, вы получили некоторое представление о возможностях усиления комплексной амплитуды. Возможность преобразования фаз, которые не читаются операцией READ, в читаемые амплитуды безусловно выглядит полезно, но как применить ее на практике? Усиление комплексной амплитуды может применяться по-разному, но одним из чрезвычайно практичных применений является метод *квантовой оценки суммы* (QSE, Quantum Sum Estimation).

AA и QFT при оценке суммы

Вы уже видели, что частота колебаний вероятности в примерах итераций AA зависит от количества инвертированных значений. В следующей главе будет представлено квантовое преобразование Фурье (QFT, Quantum Fourier Transform) — примитив QPU, позволяющий прочитать частоту изменения значений в квантовом регистре.

Оказывается, объединение примитивов AA и QFT позволяет создать схему, позволяющую прочитать не одно из помеченных значений, а значение, соответствующее количеству инвертированных помеченных значений в исходном состоянии регистра. Схема является разновидностью квантовой оценки суммы. Квантовая оценка суммы будет подробно рассмотрена в главе 11, но мы упоминаем ее здесь, чтобы дать представление о практической полезности усиления комплексной амплитуды.

Ускорение традиционных алгоритмов с применением AA

Оказывается, использование AA в качестве вспомогательной процедуры во многих традиционных алгоритмах обеспечивает квадратичный прирост скорости. Круг задач, в которых может эффективно применяться AA, составляют задачи с вызовом подпрограммы, которая многократно проверяет правильность решения. Примерами такого рода служат *задачи выполнимости булевых формул* и *поиска глобальных и локальных минимумов*.

Как было показано ранее, примитив AA состоит из двух частей: инвертирования и зеркальной операции. В первой части кодируется эквивалент классической процедуры, проверяющей правильность решения, тогда как вторая часть остается неизменной для всех приложений. В главе 14 этот аспект AA будет рассмотрен более подробно, а также показано, как кодировать классические процедуры в первой части AA.

Внутри QPU

Как же операции QPU, формирующие каждую итерацию AA, способствуют выполнению ее задачи? Вместо того чтобы разбираться с функционированием каждой отдельной операции, мы попробуем сформировать интуитивное понимание эффекта каждой итерации AA. Оказывается, существует полезное представление усиления комплексной амплитуды в контексте его геометрического воздействия на круговую запись регистра QPU.

Интуитивное понимание

Усиление комплексной амплитуды состоит из двух этапов: инвертирования фазы и зеркальной операции. Инвертирование переключает фазу помеченного элемента суперпозиции, который в конечном итоге будет извлечен из QPU.

Зеркальная операция, которая остается неизменной в примитиве AA, преобразует разности фазы в разности амплитуд. Однако зеркальную операцию также можно описать другим способом: она *отражает* каждое значение, входящее в состояние, относительно среднего всех значений.

Помимо того что эта альтернативная интерпретация объясняет, почему операция называется зеркальной, она также предоставляет более точное пошаговое описание того, как работает зеркальная операция.

Предположим, зеркальной операции передается двухкубитное входное состояние, которое является суперпозицией $|0\rangle$ и $|3\rangle$ (рис. 6.20).

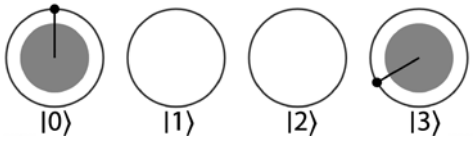


Рис. 6.20. Начальное состояние

Если рассматривать происходящее в круговой записи, зеркальная операция выполняет следующие действия:

1. Вычисляется среднее по всем значениям (кругам). Это может быть сделано числовым усреднением координат x и y точек внутри кругов[1]. При вычислении среднего нулевые значения (пустые круги) включаются в виде [0.0, 0.0], как на рис. 6.21 и 6.22.
2. Каждое значение отражается относительно общего среднего. В визуальном представлении происходит обычное симметричное отображение (рис. 6.23).

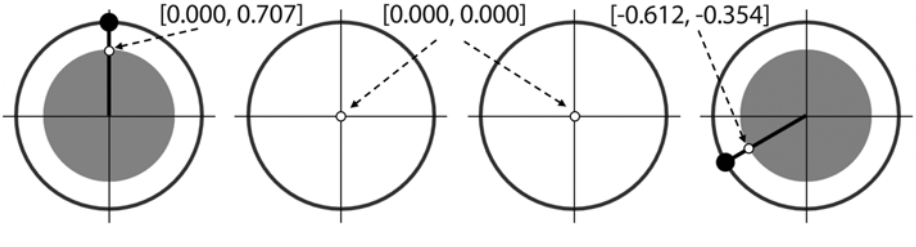


Рис. 6.21. Вычисление среднего

[1] В контексте полноценного математического описания состояния регистра QPU как комплексного вектора это соответствует усреднению вещественных и мнимых составляющих его компонентов.

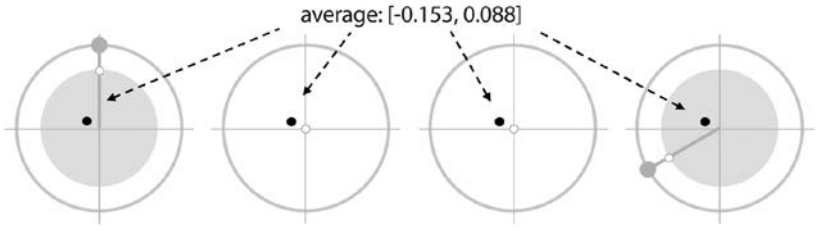


Рис. 6.22. Графическое представление среднего

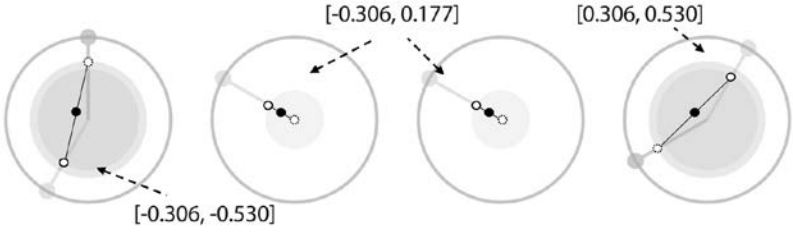


Рис. 6.23. Симметричное отображение относительно среднего

Собственно, это все. В результате (рис. 6.24) разности фаз в исходном состоянии были преобразованы в разности амплитуд. Стоит заметить, что общее среднее по всем кругам остается неизменным. Это означает, что при повторном применении преобразования просто произойдет возврат к исходному состоянию.

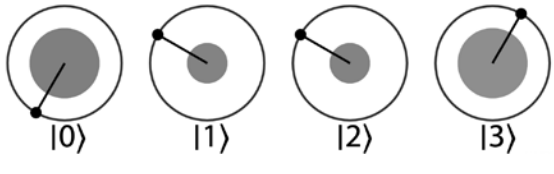


Рис. 6.24. Полученное состояние

Как получается, что операция «отражения относительно среднего» приводит к преобразованию разностей фазы и амплитуды? Представьте, что существует много состояний с одинаковыми фазами, а также одно аномальное состояние с сильно отличающейся фазой, как показано на рис. 6.25.

Учитывая, что большинство значений совпадает, среднее значение будет располагаться ближе к значению большинства состояний и очень далеко

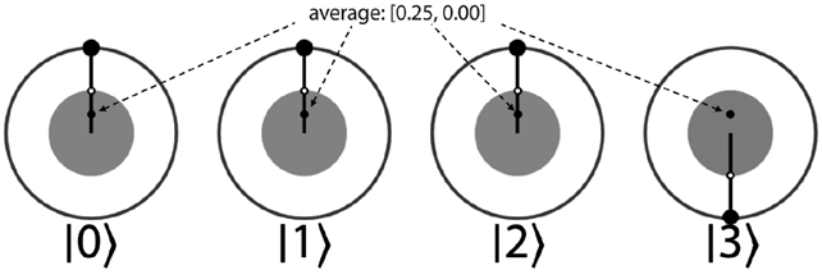


Рис. 6.25. Состояние с одной сильно отличающейся фазой

от состояния с противоположной фазой. Это означает, что при отражении относительно среднего значение с другой фазой «перелетит» через среднее и будет выделяться на общем фоне (рис. 6.26).

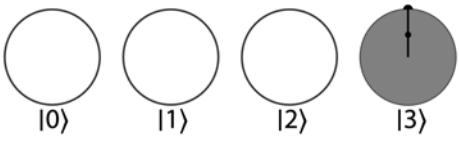


Рис. 6.26. Итоговое состояние

На практике обычно бывает проще реализовать отражение относительно среднего + инвертирование всех фаз вместо простого отражения относительно среднего. Так как на самом деле важны только относительные фазы в регистре, такая операция будет полностью эквивалентной.

Итоги

В этой главе представлена одна из базовых операций для многих практических применений QPU. Посредством преобразования разности фаз в разности амплитуд усиление комплексной амплитуды позволяет программам QPU предоставлять полезные выходные данные, связанные с информацией фазы из состояния, которая в противном случае осталась бы невидимой. Истинная мощь этого примитива будет рассматриваться в главах 11 и 14.

7 QFT: квантовое преобразование Фурье

Квантовое преобразование Фурье (QFT) — примитив, позволяющий обращаться к скрытой информации, хранящейся в относительных фазах и амплитудах регистра QPU. Хотя усиление комплексной амплитуды позволяет преобразовать разности относительных фаз в различия амплитуд, которые могут читаться операцией READ, вскоре вы увидите, что примитив QFT предоставляет собственный механизм манипуляций с фазами. Также будет показано, что кроме *фазовых манипуляций*, примитив QFT упрощает *вычисления в суперпозиции*, упрощая подготовку сложных суперпозиций регистра. Эта глава начинается с нескольких прямолинейных примеров QFT, а затем переходит к нетривиальным аспектам этой операции. Для любознательного читателя во врезке «Внутри QPU» преобразование QFT рассматривается по отдельным операциям.

Скрытые закономерности

Немного усложним игру «угадай состояние» из главы 6. Допустим, имеется четырехкубитный квантовый регистр, содержащий одно из трех состояний (A, B или C), как на рис. 7.1, но мы не знаем, какое именно.

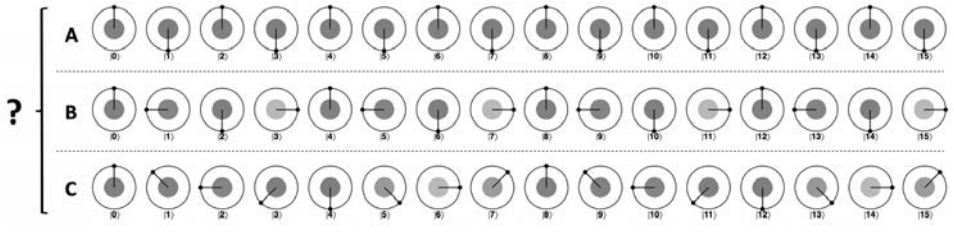


Рис. 7.1. Три разных состояния до применения QFT

Обратите внимание: это *не те* состояния A, B и C, которые рассматривались в предыдущей главе.

Визуально можно определить, какие состояния отличаются друг от друга, но поскольку амплитуды всех значений в каждом состоянии одинаковы, при чтении регистра будет получено случайное значение с равномерным распределением (независимо от того, в каком состоянии он фактически находился).

Даже усиление комплексной амплитуды здесь особой пользы не принесет, так как нет какой-то одной фазы, которая бы выделялась на фоне других в каждом состоянии. На помощь приходит примитив QFT! (Звучит драматическая музыка.) Применение QFT к регистру перед чтением преобразует каждое из состояний A, B и C в результаты, показанные на рис. 7.2.

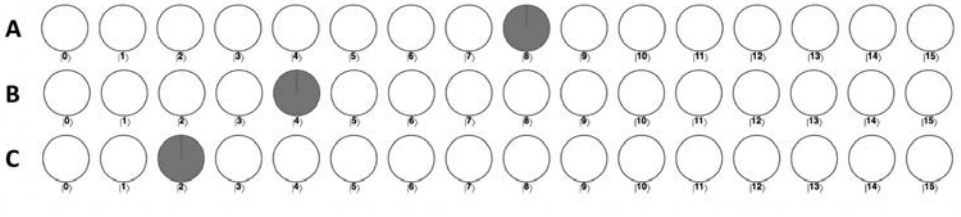


Рис. 7.2. Те же три состояния после применения QFT

Теперь чтение регистра позволит немедленно и однозначно определить, какое состояние было исходным, с использованием только одного набора операций READ. Между результатами QFT и входными состояниями на рис. 7.1 существует любопытная связь. В состоянии A относительная фаза входного состояния возвращается в 0 восемь раз, а QFT позволяет прочитать значение 8. В состоянии B относительная фаза возвращается к своему начальному состоянию четыре раза, а QFT позволяет прочитать значение 4. Состояние C подчиняется той же закономерности. В каждом случае QFT успешно открывает *частоту сигнала*, содержащуюся в регистре QPU.

Пример кода на рис. 7.1 позволит вам сгенерировать состояния A, B и C для самостоятельных экспериментов. Изменяя значение `which_signal`, можно выбрать состояние для подготовки.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=7-1.

Листинг 7.1. Применение QFT для распознавания трех состояний

```
var num_qubits = 4;
qc.reset(num_qubits);
var signal = qint.new(num_qubits, 'signal')
var which_signal = 'A';

// Подготовка сигнала
signal.write(0);
signal.hadamard();
if (which_signal == 'A') {
    signal.phase(180, 1);
} else if (which_signal == 'B') {
    signal.phase(90, 1);
    signal.phase(180, 2);
} else if (which_signal == 'C') {
    signal.phase(45, 1);
    signal.phase(90, 2);
    signal.phase(180, 4);
}

signal.QFT()
```

Если вы захотите применить примитив QFT к состояниям, сгенерированным в листинге 7.1, это можно сделать в QCEngine при помощи встроенной функции `QFT()`. Операция может быть реализована как глобальным методом `qc.QFT()`, получающим набор кубитов в аргументе, так и методом объекта `qint` — `qint.QFT()`, применяющим QFT ко всем кубитам в объекте `qint`.

QFT, DFT и FFT

Чтобы лучше понять, как преобразование QFT раскрывает частоты сигналов, лучше всего проанализировать его очень сильное сходство с классическим механизмом обработки сигналов, называемым дискретным преобразованием Фурье (DFT, Discrete Fourier Transform). Если вы когда-либо возились с графическим эквалайзером в акустической системе, то общая идея DFT вам уже знакома — как и QFT, это преобразование позволяет проанализировать различные частоты, содержащиеся в сигнале. Хотя механизм DFT используется для анализа более традиционных сигналов, применяемое им преобразование практически идентично математическим алгоритмам QFT. Более традиционное преобразование DFT лучше подходит для формирования интуитивных представлений, поэтому его основные концепции будут представлены по мере изложения материала главы.

Одной из популярных разновидностей DFT является быстрое преобразование Фурье (FFT, Fast Fourier Transform). Это самый быстрый из известных методов определения дискретных преобразований Фурье, поэтому FFT часто будет использоваться для сравнений при рассмотрении квантовой разновидности. Быстрое преобразование Фурье также похоже на QFT тем, что оно ограничивается длинами сигналов, равными степени 2.

Сокращений -FT набралось уже немало, поэтому ниже приводится краткий глоссарий.

- *DFT* — традиционное дискретное преобразование Фурье; используется для извлечения частотной информации из традиционных сигналов.
- *FFT* — быстрое преобразование Фурье; конкретный алгоритм для реализации DFT. Выполняет точно такое же преобразование, что и DFT, но работает значительно быстрее. Именно на него мы будем ориентироваться в описаниях QFT.
- *QFT* — квантовое преобразование Фурье. Выполняет то же преобразование, что и DFT, но работает не с традиционными потоками информации, а с сигналами, закодированными в квантовых регистрах.

Частоты в регистре QPU

Представление об амплитудах и относительных фазах в квантовых регистрах как о сигналах — исключительно богатая идея, но она заслуживает пояснений. С учетом специфических особенностей QPU стоит более четко оговорить, что же понимается под содержанием частот в квантовом регистре. Допустим, имеется состояние четырехкубитного регистра, изображенное на рис. 7.3.

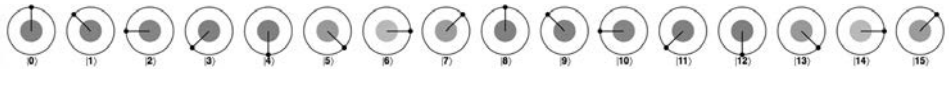


Рис. 7.3. Пример входного состояния QFT

Перед вами состояние C из предыдущего примера. Обратите внимание: при проходе слева направо относительные фазы $2^4 = 16$ значений проходят два полных поворота против часовой стрелки. Таким образом, можно считать, что относительные фазы в регистре представляют *сигнал*, повторяемый с частотой *2 раза за регистр*.

Это дает некоторое представление о том, как квантовый регистр может кодировать сигнал, с которым связана некоторая частота, но поскольку сигнал связан с относительными фазами кубитов, его заведомо не удастся извлечь простым чтением регистра. Как и в примере, рассмотренном в начале главы, информация *скрыта* в фазах, но может быть выявлена применением QFT (рис. 7.4).

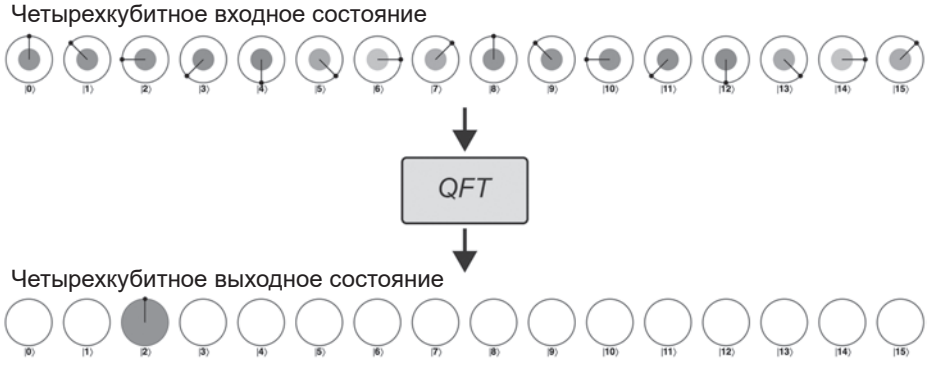


Рис. 7.4. Второй пример QFT

В этом простом случае чтение регистра после QFT позволяет определить, что частота повторения была равна 2 (то есть относительная фаза делает два полных поворота за регистр). Это одна из ключевых концепций QFT, и вскоре вы научитесь точнее использовать и интерпретировать ее. Она воспроизведена в листинге 7.2 (рис. 7.5).

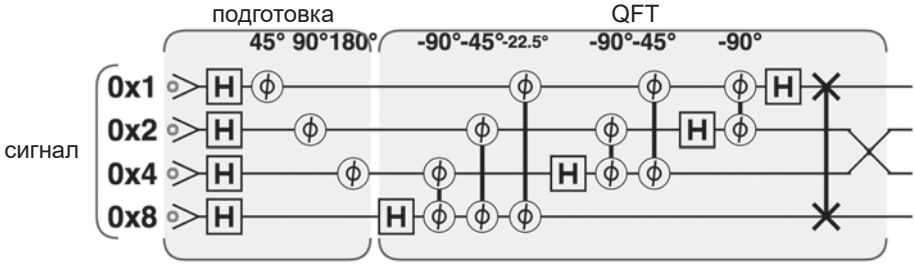


Рис. 7.5. QFT для простого сигнала в регистре QPU

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=7-2.

Листинг 7.2. QFT для простого сигнала в регистре QPU

```
var num_qubits = 4;
qc.reset(num_qubits);
var signal = qint.new(num_qubits, 'signal')

// Подготовить сигнал
signal.write(0);
signal.hadamard();
signal.phase(45, 1);
signal.phase(90, 2);
signal.phase(180, 4);

// Выполнить QFT
signal.QFT()
```

Листинг 7.2 демонстрирует важное свойство QFT. И хотя в главе 5 было показано, что операции, реализованные QPU, часто должны использовать разные входные и выходные регистры (для обеспечения обратимости), вывод QFT оказывается в том же регистре, в котором содержался ввод, — отсюда термин «преобразование». QFT может работать «на месте», потому что эта операция обратима по своей природе.

Казалось бы, почему нас вообще интересует инструмент, помогающий находить частоты в регистре QPU? Как это может помочь в решении практических задач? Как ни странно, QFT оказывается разносторонним инструментом программирования QPU, и ближе к концу главы будут приведены конкретные примеры.

Так что, вся суть QFT сводится именно к этому? Подать на вход регистр, получить обратно частоту? И да и нет. До настоящего момента все рассматривавшиеся сигналы после применения QFT давали одну четко определенную частоту (потому что мы специально отбирали их). Но для многих других периодических сигналов QFT не получаются настолько простыми. Для примера рассмотрим четыре сигнала, показанных на рис. 7.6.

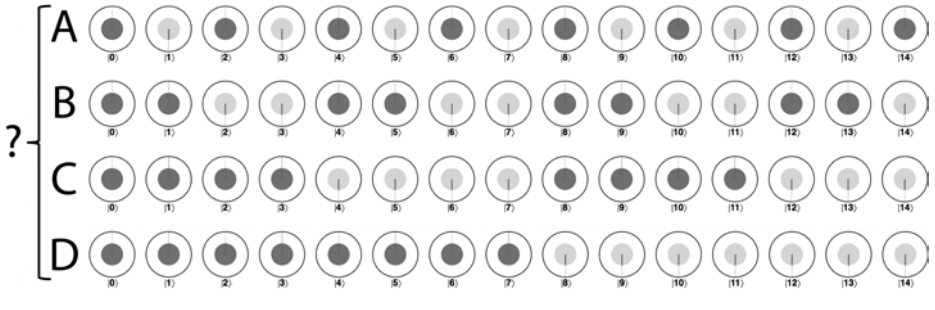


Рис. 7.6. Четыре прямоугольных сигнала перед применением QFT

Значение в состоянии регистра QPU, имеющее фазу 180° (обозначается в круговой записи линией, направленной на юг), эквивалентно отрицательному значению комплексной амплитуды.

Перед вами эквиваленты примеров, впервые представленных на рис. 7.1, из области прямоугольных волн. Хотя колебания имеют ту же частоту, что и в двух предыдущих примерах, их относительные фазы резко переключаются между двумя положительными и отрицательными значениями вместо непрерывного изменения. Соответственно для них QFT интерпретировать немного сложнее, как видно из рис. 7.7.

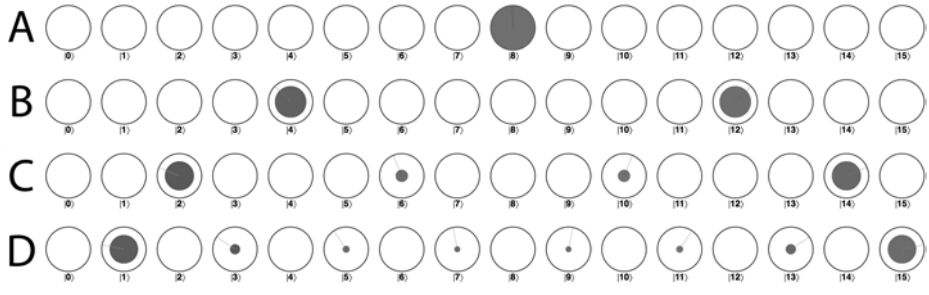


Рис. 7.7. Те же прямоугольные сигналы после применения QFT

Следует заметить, что входные состояния, показанные в этих примерах, можно сгенерировать операцией HAD и тщательно подобранными операциями PHASE. Пример кода в листинге 7.3 генерирует эти состояния прямоугольной волны, а затем применяет QFT (рис. 7.8). Чтобы выбрать, какое из состояний на рис. 7.6 должно быть сгенерировано, используйте переменную `wave_period`.

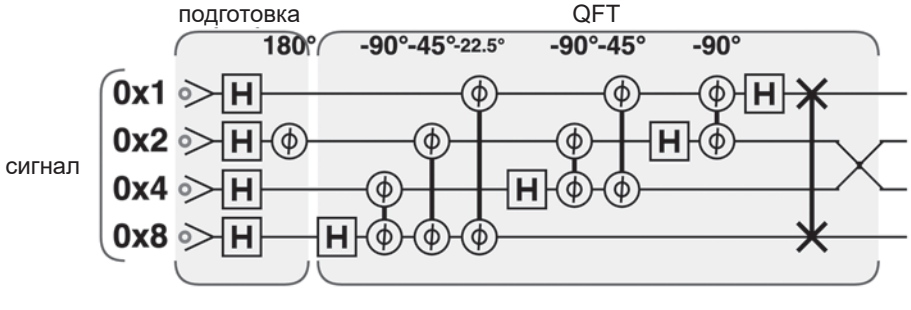


Рис. 7.8. Квантовые операции, используемые QFT

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=7-3.

Листинг 7.3. QFT для прямоугольной волны

```
var num_qubits = 4;
qc.reset(num_qubits);
var signal = qint.new(num_qubits, 'signal')
var wave_period = 2; // A:1 B:2 C:4 D:8

// Подготовить сигнал
signal.write(0);
signal.hadamard();
signal.phase(180, wave_period);

signal.QFT()
```

Если вы уже знакомы с традиционным дискретным преобразованием Фурье, возможно, результаты на рис. 7.7 не покажутся столь загадочными. Так как QFT в действительности просто применяет DFT к сигналам в регистрах QPU, для начала вспомним, как работает это более привычное преобразование.

DFT

DFT работает с дискретными выборками, полученными на базе сигнала, будь то форма волны музыкального ролика или цифровое представление изображения. Хотя традиционные сигналы обычно рассматриваются как списки вещественных значений, более понятные на интуитивном уровне, DFT также работает с комплексными значениями. Это обнадеживает, потому что полное представление состояния регистра QPU в наиболее общей форме описывается списком комплексных чисел (хотя мы всеми силами постараемся обойтись без этого).

Мы использовали круговую запись для интуитивного представления математических операций с комплексными числами там, где это было возможно. Каждый раз, когда вы видите упоминание комплексного числа, всегда можете представить круг из нашей записи, с амплитудой (размер круга) и фазой (угол поворота круга).

Применим традиционное представление DFT к простым синусоидальным сигналам: с точками дискретизации, которые являются целыми числами, и с четко определенной частотой (если бы речь шла о звуковом сигнале, это был бы чистый тон). Предположим, что мы получили 256 значений сигнала. Каждый образец хранится в виде комплексного числа с плавающей точкой (с амплитудой и фазой, как в круговой записи). В нашем конкретном примере мнимая часть будет равна нулю для всех образцов. Пример такого сигнала показан на рис. 7.9.

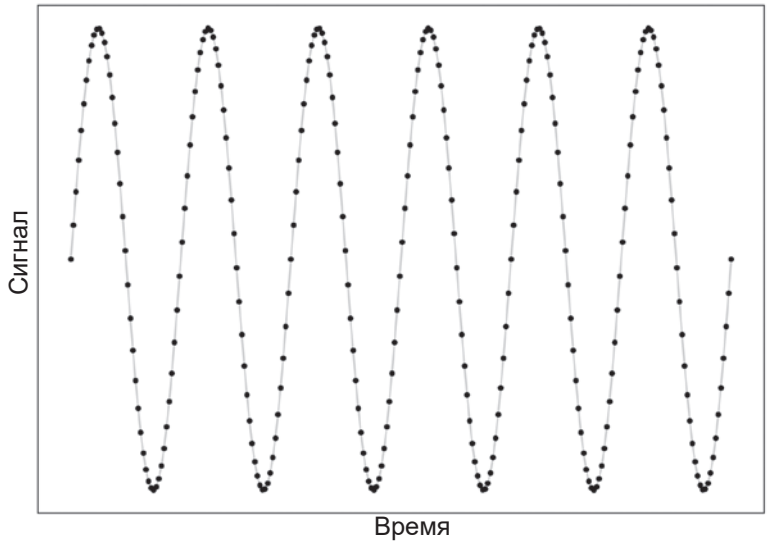


Рис. 7.9. Дискретизация 256 точек для простой синусоидальной волны

Если каждый образец хранится в виде 16-байтового комплексного вещественного числа (8 байт для вещественной и 8 байт для мнимой части), этот сигнал поместится в 4096-байтовом буфере. Обратите внимание: мнимые части все равно необходимо отслеживать (при том что в данном примере используются только вещественные входные значения), так как *результатом* работы DFT будут 256 комплексных значений.

Амплитуды комплексных выходных значений DFT сообщают, насколько значимый вклад вносит заданная частота в формирование сигнала. *Фазы* сообщают, насколько эти частоты *смещены* относительно друг друга во входном сигнале. На рис. 7.10 показано, как выглядят амплитуды DFT для простого вещественного синусоидального сигнала.

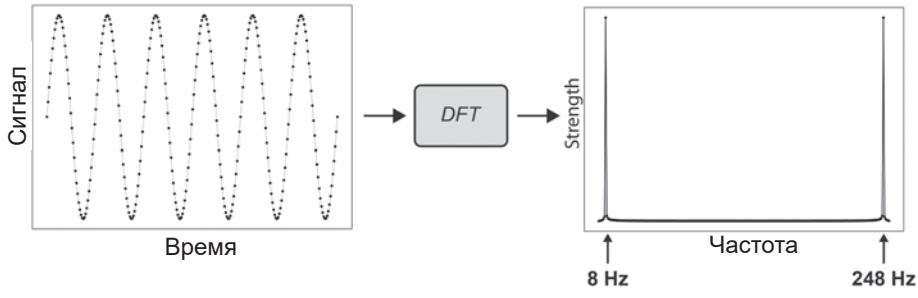


Рис. 7.10. Результаты DFT для простой одночастотной синусоидальной волны

Присмотревшись к основаниям двух пиков на рис. 7.10, вы увидите, что они окружены малыми ненулевыми значениями. Поскольку вывод DFT ограничен конечным количеством битов, пики имеют ненулевую ширину даже при том, что сигнал в действительности содержит только одну частоту. Этот эффект может проявляться и для QFT.

DFT преобразует сигнал в *диапазон частот*, в котором видны все частотные составляющие, присутствующие в сигнале. Так как входной сигнал совершает восемь полных колебаний за время дискретизации (1 с), мы считаем, что он имеет частоту 8 Гц — и именно это значение DFT возвращает нам в выходном регистре.

Вещественные и комплексные входные данные для DFT

Из результатов DFT в нашем примере можно заметить крайне неприятную проблему в виде частоты 248 Гц. Наряду с ожидаемой частотой 8 Гц DFT также выдает второй очевидный *зеркальный* пик в диапазоне частот.

Это свойство присуще DFT любого вещественного сигнала (то есть сигнала, у которого все образцы являются вещественными числами, как у большинства традиционных сигналов). В таких случаях полезна только первая половина результата DFT. Таким образом, в нашем примере будут использоваться только первые 256/2 = 128 точек, возвращаемых DFT. Все остальные данные в DFT после этой точки будут зеркальным отражением первой половины (отсюда и появление второго симметричного пика на частоте 256 – 8 = 248 Гц). На рис. 7.11 приведены другие примеры DFT реальных сигналов, которые подчеркивают эту симметрию.

Если бы использовался комплексный входной сигнал, то эффект симметрии не был бы заметен, и каждая из 256 точек данных в выходных данных DFT содержала бы самостоятельную информацию.

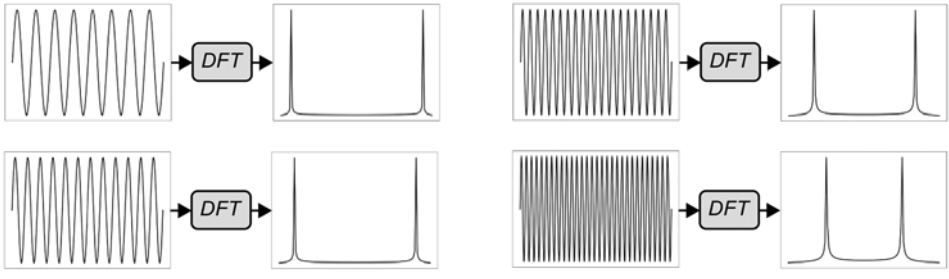


Рис. 7.11. Другие примеры DFT реальных сигналов (точки данных не показаны)

Все, что было сказано об интерпретации результатов вещественных и комплексных входных сигналов, также относится к QFT. Возвращаясь к примерам QFT, приведенным на рис. 7.2 и 7.4, вы также заметите, что эффект симметрии отсутствует — в выходных регистрах виден только один «пик», точно соответствующий частоте входного сигнала. Это объясняется тем, что сигнал, закодированный в относительных фазах состояний входного регистра, был комплексным, поэтому симметрическая избыточность отсутствует.

В регистре QFT возможно подготовить чисто вещественные сигналы. Предположим, сигнал, запланированный для применения QFT, был закодирован в амплитудах входного регистра QPU вместо относительных фаз, как показано на рис. 7.12.

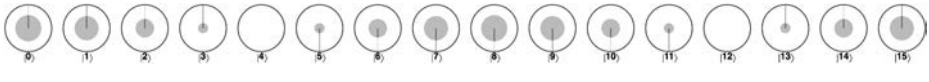


Рис. 7.12. Квантовые регистры с сигналом, закодированным в амплитудах

Это чисто вещественный входной сигнал. Пусть вас не смущает тот факт, что относительные фазы изменяются в пределах состояния, — напомним, что фаза 180° эквивалентна отрицательному значению, поэтому здесь присутствуют только вещественные положительные и отрицательные числа.

При применении QFT к вещественному входному состоянию выясняется, что в выходных регистрах проявляется точно такой же зеркальный эффект, который наблюдался с традиционным DFT (рис. 7.13).

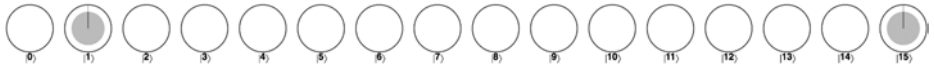


Рис. 7.13. Выходной регистр после применения QFT к сигналу, закодированному исключительно в амплитудах

Следовательно, необходимо тщательно следить за интерпретацией результатов QFT в зависимости от способа кодирования информации во входном регистре (фаза или амплитуда).

Подробнее о DFT

До сих пор мы довольно туманно говорили о том, что DFT (и QFT) сообщают, какие частоты содержит сигнал. Если говорить чуть более конкретно, эти преобразования возвращают информацию о частотах, пропорциях и смещениях простых *синусоидальных* составляющих, которые следует объединить для получения входного сигнала.

Многие примеры входных сигналов, рассмотренные выше, представляли собой простые синусоиды с одной четко определенной частотой. Как следствие, DFT и QFT давали один четко определенный пик в диапазоне частот (что фактически означает: «Этот сигнал можно построить всего из одной синусоидальной функции с конкретной частотой!»). Одно из самых полезных свойств представления DFT заключается в том, что его можно применять к более сложным сигналам, форма которых отлична от простой синусоиды. На рис. 7.14 показаны амплитуды от традиционного преобразования DFT, выполненного для входного сигнала, содержащего синусоидальные колебания с тремя разными частотами.

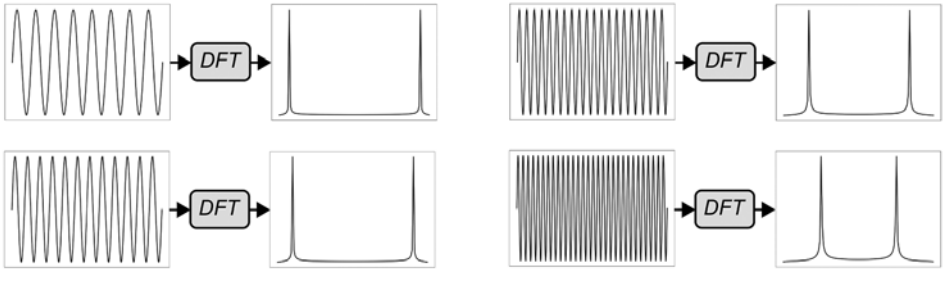


Рис. 7.14. Применение DFT к сигналу, содержащему смешанные частоты

Всегда стоит принимать во внимание значение DFT (или QFT), соответствующее нулевой частоте. Иногда оно называется постоянной составляющей (DC bias) — это базовая линия, выше и ниже которой колеблется сигнал. Поскольку во всех наших примерах колебания происходили относительно нуля, постоянная составляющая в их DFT отсутствовала.

DFT не только показывает, что сигнал состоит из трех синусоидальных частот; относительные высоты пиков в диапазоне частот также сообщают значимость вклада каждой частоты в сигнал (а анализ фаз комплексных значений, возвращаемых DFT, также может дать информацию о том, насколько синусоиды смещены относительно друг друга). Конечно, не стоит забывать, что поскольку наш входной сигнал является вещественным, вторую (зеркальную) половину DFT можно игнорировать.

Один особенно распространенный и полезный класс входных сигналов, совершенно не похожих на синусоиды, составляют прямоугольные волны. Примеры QFT прямоугольных волн уже приводились ранее на рис. 7.7. На рис. 7.15 показано, как выглядят амплитуды для традиционного преобразования DFT прямоугольной волны.

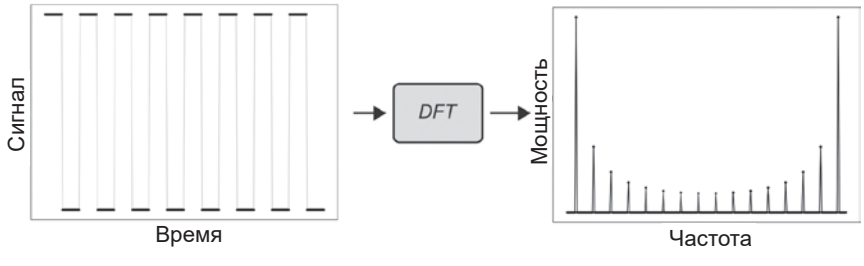


Рис. 7.15. DFT прямоугольной волны

Хотя результаты QFT для прямоугольных волн уже приводились на рис. 7.7, разберем пример более подробно. Чтобы задача была более интересной, возьмем восьмикубитный регистр (то есть кубайт), предоставляющий 256 значений состояний для экспериментов. Пример кода из листинга 7.4 (рис. 7.16) подготавливает прямоугольную волну, которая повторяется восемь раз, — регистр QPU, эквивалентный этому сигналу, показан на рис. 7.15.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=7-4.

Листинг 7.4. QFT для прямоугольной волны

```
// Подготовка
qc.reset(8);

// Создать равную суперпозицию
qc.write(0);
qc.had();

// Ввести отрицательное значение с некоторой частотой
// (изменяя фазу для других кубитов, можно изменить
// частоту прямоугольной волны)
qc.phase(180, 16);

// Применить QFT
qc.QFT();
```

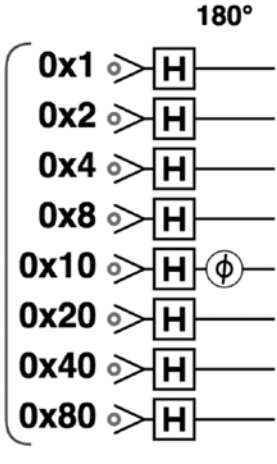


Рис. 7.16. Квантовые вентили, необходимые для подготовки восьмикубитного прямоугольного сигнала

Код, предшествующий QFT, оставляет входной регистр QPU в состоянии, показанном на рис. 7.17.

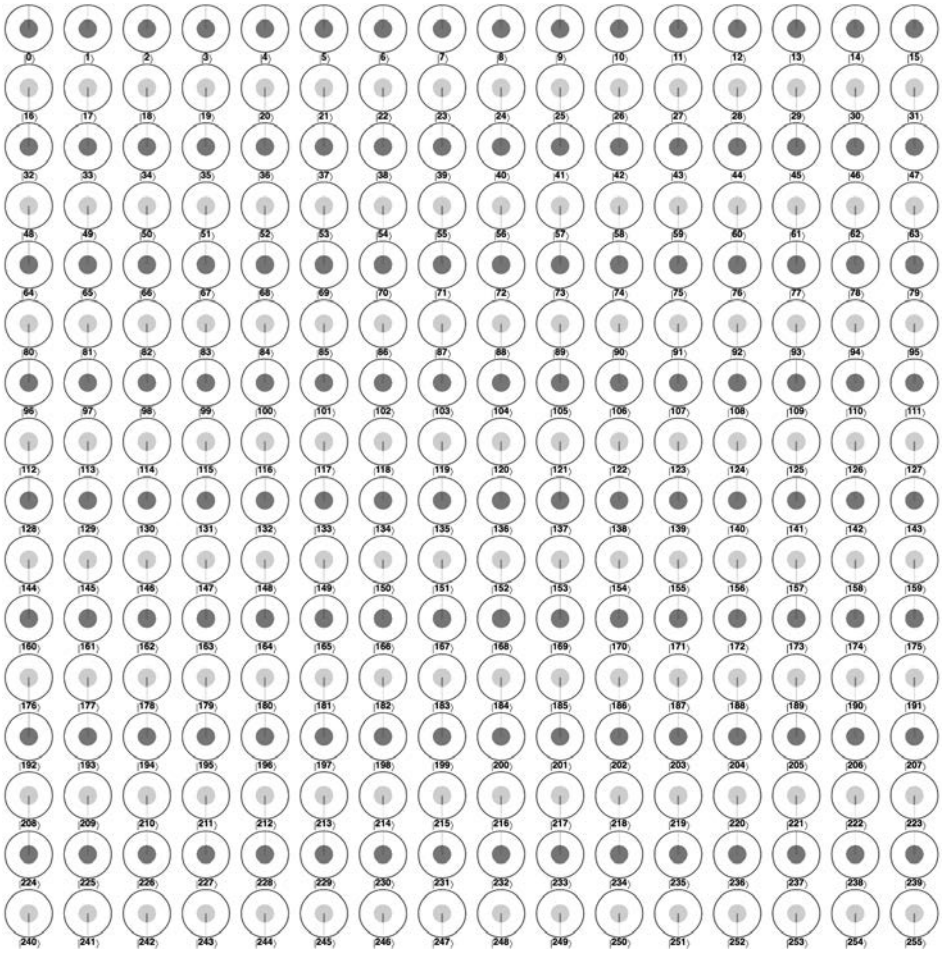


Рис. 7.17. Сигнал прямоугольной формы в кубайте

Если вспомнить, что зеленые круги (с относительной фазой 180°) представляют отрицательные значения, вероятно, вы сможете убедиться в том, что этот сигнал является чисто вещественным и полностью аналогичным входному сигналу DFT на рис. 7.15.

В результате применения QFT к регистру в этом состоянии (вызовом `qc.QFT()` в QCEngine) будет получено выходное состояние, показанное на рис. 7.18.

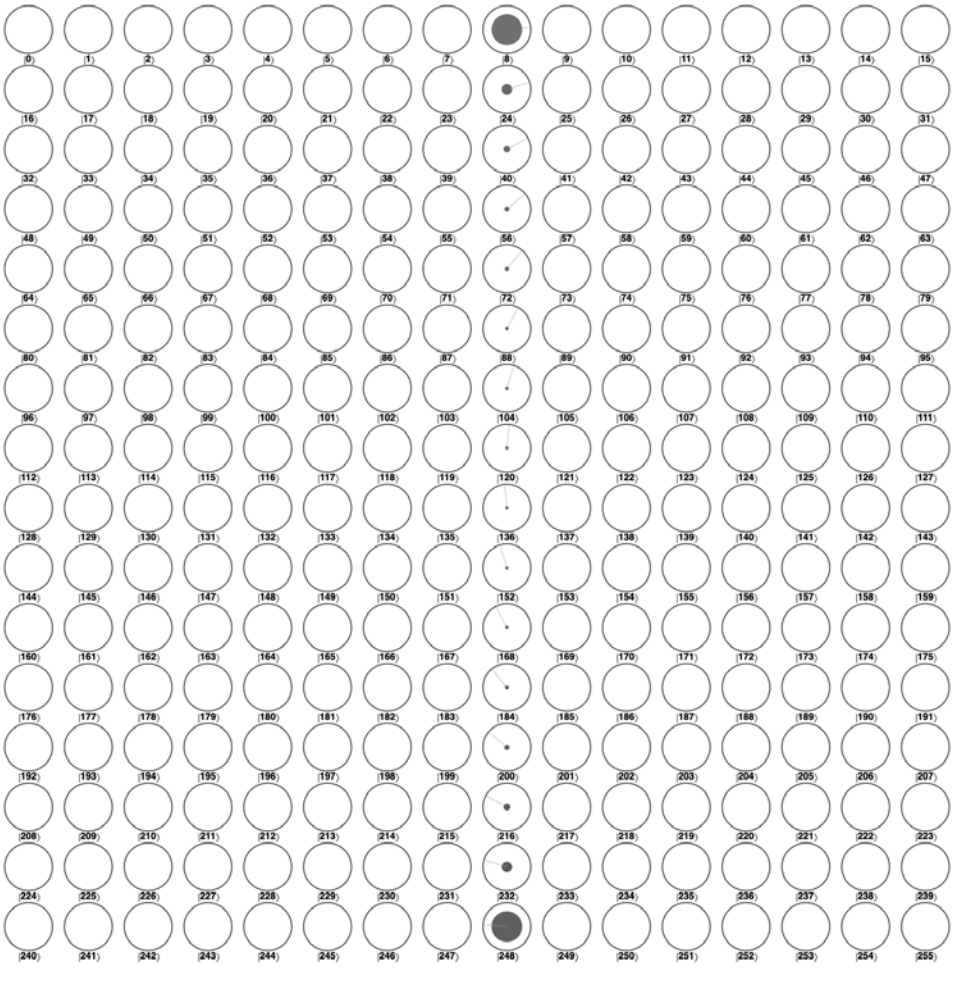


Рис. 7.18. Выходные данные QFT для прямоугольного входного сигнала

Этот регистр в точности соответствует DFT прямоугольной волны на рис. 7.15, при этом каждая составляющая в диапазоне частот кодируется амплитудами и относительными фазами состояний регистра QPU. Это означает, что вероятность чтения заданной конфигурации регистра после QFT теперь определяется тем вкладом, который заданная частота вносит в сигнал.

Если нанести эти вероятности на график на рис. 7.19, можно заметить сходство с результатами традиционного преобразования DFC, полученными ранее для прямоугольной волны на рис. 7.15.

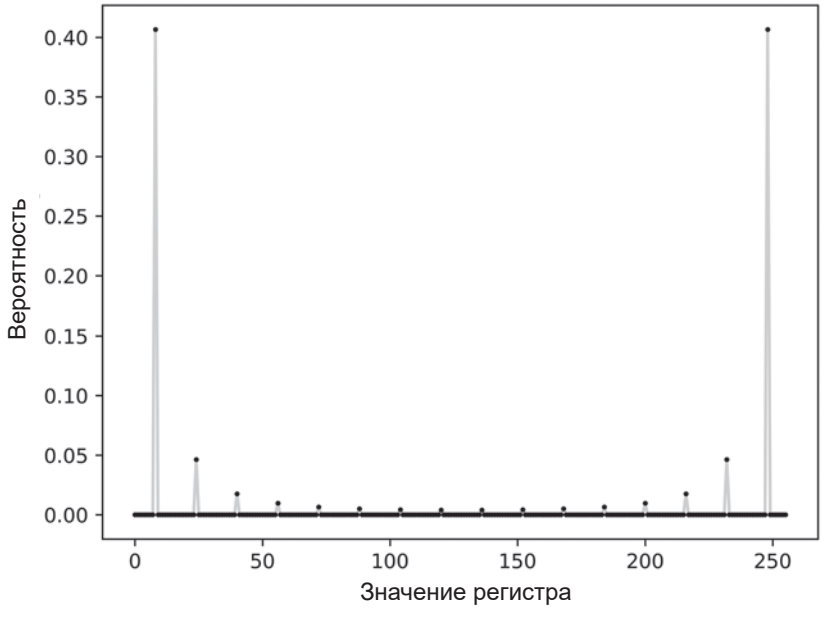


Рис. 7.19. Вероятности чтения результатов QFT для входного кубайта с прямоугольным сигналом

Практическое применение QFT

Мы приложили немало усилий, чтобы показать: преобразование QFT можно рассматривать как реализацию DFT для сигналов в регистрах QPU. Учитывая, насколько капризным и дорогостоящим устройством является квантовый процессор, может показаться, что все хлопоты привели к простому повторению традиционного алгоритма. Важно помнить, что главной причиной для использования примитива QFT для нас была обработка результатов вычислений, закодированных в фазе, а не разработка полноценной замены FFT.

С нашей стороны было бы небрежностью не оценить вычислительную сложность QFT по сравнению с FFT — особенно из-за того, что QFT работает намного, *намного* быстрее любого традиционного аналога.

Скорость QFT

Когда мы говорим о скорости алгоритмов FFT и QFT, нас прежде всего интересует скорость роста времени выполнения алгоритма с ростом размера входного сигнала (в общем количестве битов, необходимых для его пред-

ставления). С практической точки зрения можно считать, что количество фундаментальных операций, используемых алгоритмом, эквивалентно времени его выполнения, поскольку выполнение каждой операции с кубитами QPU занимает фиксированное время.

С ростом количества входных битов n количество операций FFT возрастает со скоростью $O(n2^n)$. Однако QFT — за счет использования 2^m состояний, доступных в m-кубитном регистре, — использует количество операций, возрастающее со скоростью всего $O(m^2)$.

На практике это означает, что для малых сигналов (менее 22 бит, или около 4 миллионов образцов) традиционная реализация FFT будет быстрее, даже если она выполняется на ноутбуке. Но с ростом размера входной задачи преимущество QFT становится все более очевидным.

На рис. 7.20 сравнивается эффективность этих двух методов с ростом размера сигнала. По оси x измеряется количество битов (в случае FFT) или кубитов (для QFT) во входном регистре.

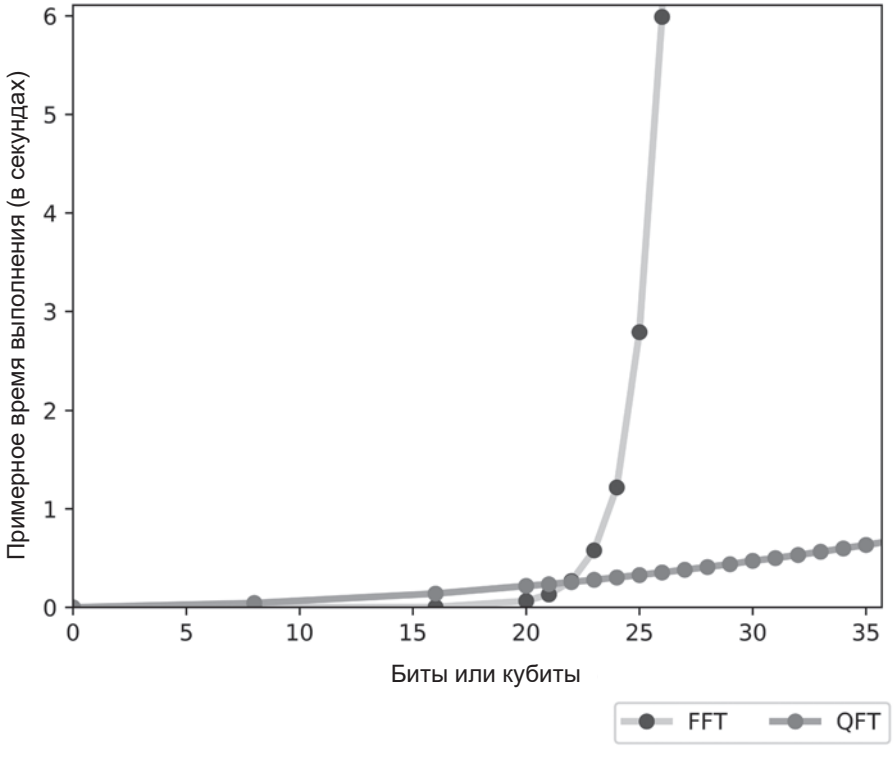


Рис. 7.20. Время выполнения QFT и FFT по линейной шкале

Обработка сигналов с QFT

С учетом важной роли FFT в обширной области обработки сигналов появляется соблазнительная мысль думать, что главная область применения QFT — реализация этого механизма обработки сигналов с экспоненциальной скоростью. Но если результатом DFT является массив цифровых значений, которые можно анализировать по своему усмотрению, вывод QFT заблокирован в выходном регистре QPU.

Попробуем чуть точнее сформулировать ограничения, с которыми мы сталкиваемся при попытке применить на практике результаты QFT. Кодирование входного сигнала и результатов его преобразования QFT в комплексных амплитудах квантового состояния создает две проблемы:

1. Как с самого начала поместить сигнал во входной квантовый регистр?
2. Как получить доступ к результату QFT после завершения вычислений?

Первая проблема — размещение входного сигнала в квантовом регистре — не всегда решается просто. Пример с прямоугольным сигналом на рис. 7.18 можно сгенерировать относительно простой квантовой программой, но если вам нужно ввести в квантовый регистр произвольные традиционные данные, такая простая схема может не существовать. Для некоторых сигналов затраты на инициализацию входного регистра могут нейтрализовать выигрыш от применения QFT.

Существуют методы, упрощающие подготовку некоторых видов суперпозиций в регистрах QPU, хотя часто они требуют расширения стандартного оборудования QPU. Один из таких методов будет представлен в главе 9.

Конечно, вторая упомянутая проблема является фундаментальной для всего программирования QPU — как прочитать ответ из регистра? Для простого одночастотного преобразования QFT, вроде показанного на рис. 7.2, при чтении из регистра будет получен однозначный ответ. Но для более сложных сигналов (вроде изображенных на рис. 7.18) QFT дает суперпозицию частот. При чтении вы получите всего лишь одну из задействованных частот.

Итоговое состояние после QFT может оказаться полезным при некоторых обстоятельствах. Возьмем результат, полученный для прямоугольного входного сигнала на рис. 7.17:

1. Если приложение (то, в котором было вызвано преобразование QFT) устраивает ответ типа «доминирующая частота или кратная ей», тогда все нормально. В нашем примере с прямоугольным сигналом есть восемь правильных ответов, и вы всегда получите один из них.

2. Если приложение может проверить ответ, то вы можете быстро определить, подходит ли вам случайный результат чтения выходного состояния QFT. Если он не подходит, то QFT можно выполнить снова.

При таком подходе функциональность обработки сигналов QFT может предоставить ценный примитив для манипуляций с фазой; в частности, они сыграют эту роль в алгоритме факторизации Шора в главе 12.

Подготовка суперпозиций с использованием обратного преобразования QFT

До настоящего момента преобразование QFT рассматривалось как примитив фазовой манипуляции. Также QFT может использоваться для подготовки (или изменения) периодически изменяющихся суперпозиций, которые было бы очень трудно реализовать другими способами. Как и все операции QPU, кроме READ, операция QFT является обратимой. Обратное преобразование QFT (invQFT) получает на входе кубитный регистр, представляющий диапазон частот, и возвращает на выходе регистр с представлением сигнала, которому соответствует заданное состояние.

invQFT можно использовать для простой подготовки периодически изменяющихся суперпозиций следующим образом:

1. Подготовить квантовый регистр, представляющий нужное состояние в диапазоне частот. Часто это оказывается проще, чем напрямую подготовить состояние с использованием более сложной схемы.
2. Применить преобразование invQFT, которое вернет нужный сигнал в выходном регистре QPU.

В листинге 7.5 (рис. 7.21) показано, как создать кубайтовый регистр с относительной фазой, совершающей три колебания.

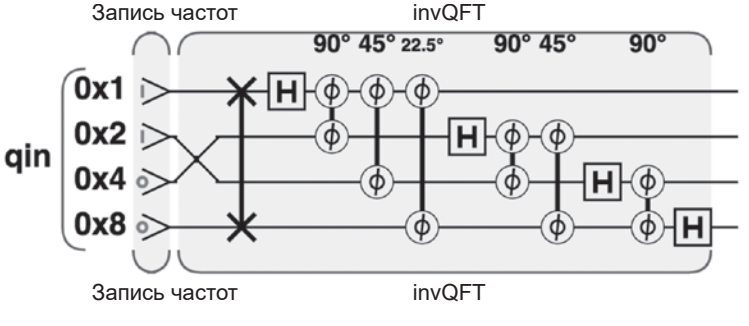


Рис. 7.21. Квантовые операции, необходимые для генерирования сигнала с периодически изменяемой фазой

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=7-5.

Листинг 7.5. Преобразование частот в состояние

```
// Подготовка
var num_qubits = 4;
qc.reset(num_qubits);
var qin = qint.new(num_qubits, 'qin');

// Записать нужную частоту в регистр
qin.write(3);

// Выполнить обратное преобразование QFT для перевода в сигнал
qin.invQFT()
```

invQFT также может использоваться для подготовки регистра QPU, содержащего периодические изменения амплитуды (вместо относительной фазы), как в примере на рис. 7.12. Для этого необходимо лишь вспомнить, что регистр с периодически изменяемыми амплитудами представляет вещественный сигнал, поэтому в диапазоне частот понадобится симметричное представление, которое станет входным для invQFT. Эта возможность продемонстрирована в листинге 7.6 (рис. 7.22).

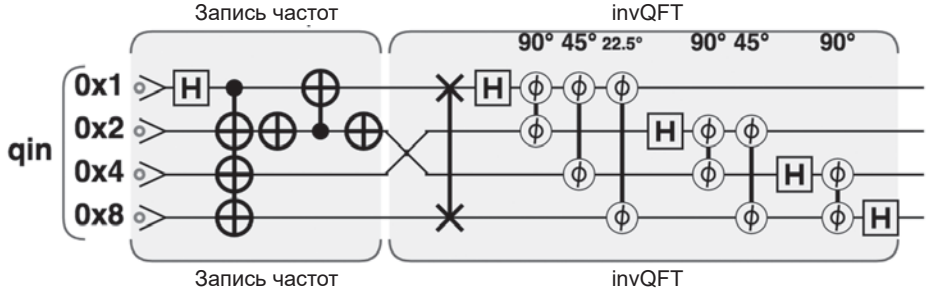


Рис. 7.22. Квантовые операции, необходимые для генерирования сигнала с периодически изменяющейся амплитудой

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=7-6.

Листинг 7.6. Подготовка состояния операцией invQFT

```
// Подготовка
var num_qubits = 4;
qc.reset(num_qubits);
var qin = qint.new(num_qubits, 'qin');
qin.write(0);

// Записать нужные частоты в регистр
qin.had(1);
qc.cnot(14,1);
qin.not(2);
qc.cnot(1,2);
qin.not(2);

// Выполнить обратное преобразование QFT для получения сигнала
qin.invQFT()
```

Кроме подготовки состояний с заданными частотами, invQFT также позволяет легко изменять их информацию о частотах. Допустим, в какой-то момент выполнения алгоритма потребовалось увеличить частоту колебания относительных фаз в регистре QPU. Для этого можно выполнить следующие действия:

1. Применить QFT к регистру, чтобы получить представление сигнала в диапазоне частот.
2. Увеличить значение, хранящееся в регистре, на 1. Так как входной сигнал является комплексным, это приведет к увеличению значения каждой частотной составляющей.
3. Применить invQFT, чтобы вернуться к исходному сигналу (с увеличенными частотами).

Простой пример приведен в листинге 7.7 (рис. 7.23).

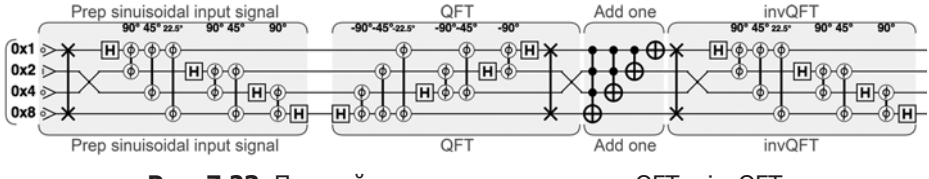


Рис. 7.23. Простой пример использования QFT и invQFT для изменения частот сигнала

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=7-7.

Листинг 7.7. Управление частотами с использованием QFT

```
// Подготовка входного регистра
var n = 4;

// Подготовить комплексный синусоидальный сигнал
qc.reset(n);
var freq = 2;
qc.write(freq);
var signal = qint.new(n, 'signal');
signal.invQFT();

// Перейти к диапазону частот при помощи QFT
signal.QFT();

// Повысить частоту сигнала
signal.add(1)

// Вернуться к диапазону частот
signal.invQFT();
```

Переход к диапазону частот позволяет выполнять манипуляции с состоянием, которые трудно выполнять другими способами, но в некоторых ситуациях необходима осторожность. Например, с вещественными входными сигналами работать труднее.

Если регистр QPU содержит состояние, представляющее отдельную частоту, или несимметричную суперпозицию частот, после применения invQFT будет получен регистр QPU, периодически изменяющийся по относительной фазе. С другой стороны, если до применения invQFT регистр содержит симметричную суперпозицию частот, выходной регистр QPU содержит состояние, периодически изменяющееся по амплитуде.

Внутри QPU

На рис. 7.24 показаны фундаментальные операции QPU, используемые для применения QFT к сигналу в кубайте.

Теперь нужно понять, как схема из простых операций QPU на рис. 7.24 извлекает частотные составляющие из сигнала во входном регистре. Чтобы понять логику QFT, необходимо подумать о том, как эта схема будет работать с входным состоянием на рис. 7.6, с периодическим изменением фазы в регистре. Это весьма непростая задача. Мы пойдем по более простому пути и попробуем объяснить, как работает обратное преобразование QFT.

Если вы сможете понять, как работает операция invQFT, то QFT — просто обратная операция.

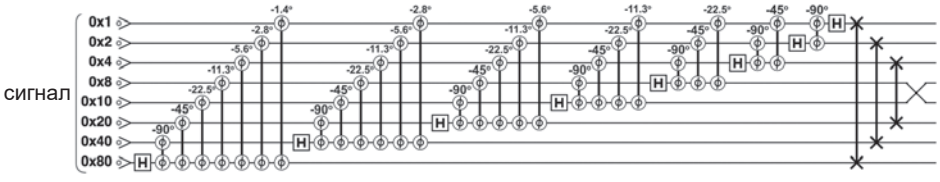


Рис. 7.24. Квантовое преобразование Фурье на уровне операций

В простом случае четырехкубитного ввода invQFT содержит разложение, показанное на рис. 7.25.

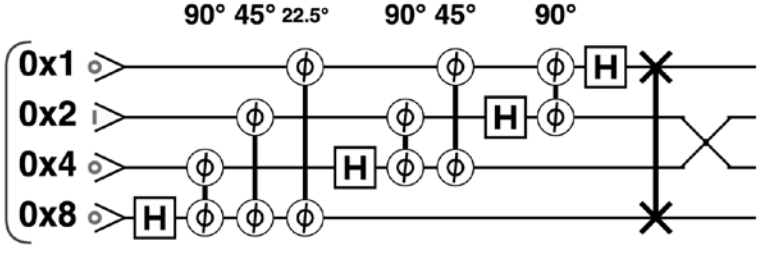


Рис. 7.25. Четырехкубитное обратное преобразование QFT

Хотя эта схема очень напоминает схему QFT, фазы принципиально различны. На рис. 7.26 показано, что вы получите при применении этой схемы к регистру, содержащему значение 2.

Схемы для QFT и invQFT могут быть записаны разными способами. Мы выбрали те формы, которые особенно наглядно объясняют их действие. Если вам встретятся другие схемы для этих примитивов, вам стоит проверить их на эквивалентность.

Интуитивное объяснение

Допустим, invQFT передается N-кубитный входной регистр, в котором закодировано некоторое число n. Если представить последовательный перебор 2^N значений выходного регистра, относительная фаза каждого последующего значения должна поворачиваться на такой угол, чтобы каждые $2^N/n$ кругов происходил полный поворот на 360°. Это означает, что для

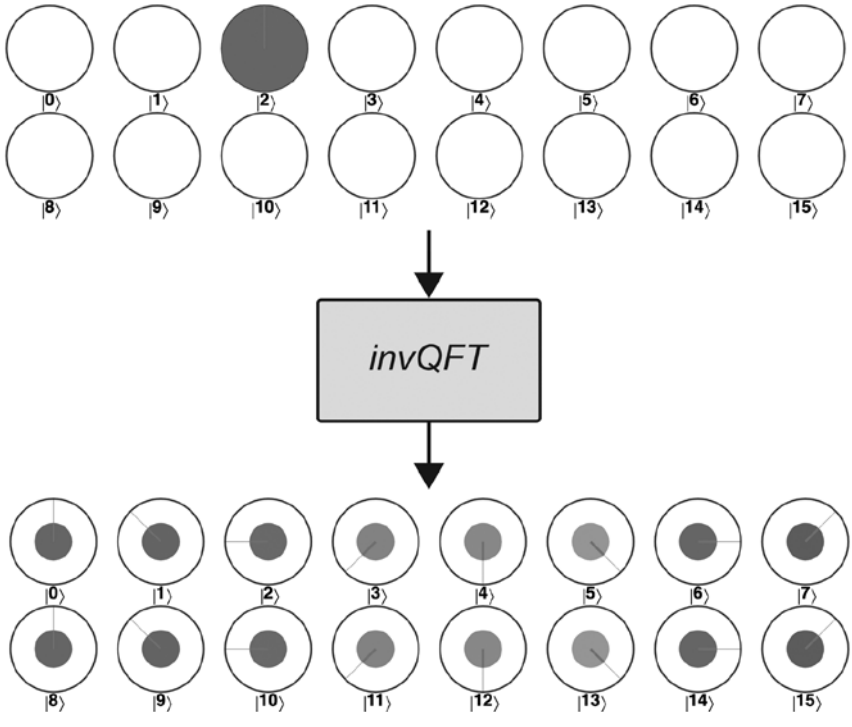


Рис. 7.26. Обратное преобразование QFT подготавливает регистр с заданной частотой

получения нужного выходного состояния относительная фаза каждого круга должна поворачиваться на дополнительный угол $360° \times n/2^N$, как на рис. 7.27.

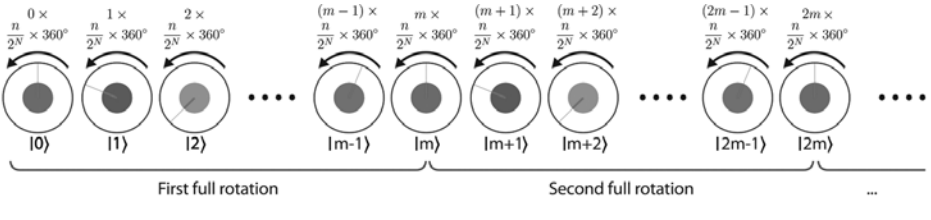


Рис. 7.27. Получение обратного преобразования QFT последовательным поворотом фазы для каждой амплитуды в регистре

Возможно, будет проще понять происходящее на конкретном примере, показанном на рис. 7.26. Для четырехкубитного регистра с входным значением 2 каждое последовательное значение должно поворачиваться на угол

$360° \times n/2^N = 360° \times 2/2^4 = 45°$ для получения желаемого выходного состояния (рис. 7.28).

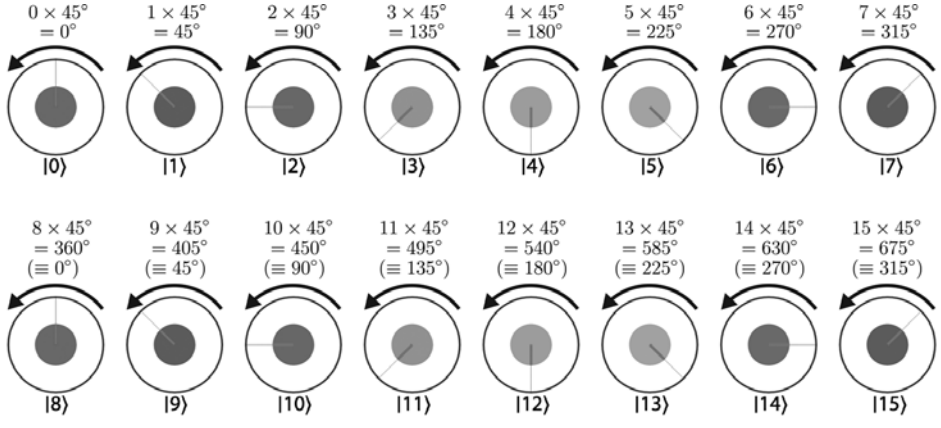


Рис. 7.28. Последовательные повороты для получения вывода QFT — конкретный пример

Повороты по этому простому правилу обеспечивают требуемую периодичность в относительных фазах регистра (конечно, $360° \equiv 0°$, $405° \equiv 45°$, и т. д).

Последовательность операций

Схема invQFT на рис. 7.25 — всего лишь умный и компактный способ реализации правила поворотов. Чтобы вы убедились в этом, задачи, выполняемые схемой на рис. 7.25, можно разбить на два разных требования:

Вычислить значение $\theta = n/2 \times 360°$ (где n — значение, изначально хранившееся в регистре).

Повернуть фазу каждого круга в регистре на кратный угол, вычисленный умножением угла θ, вычисленного на предыдущем шаге, на *десятичное значение круга*.

Преобразование invQFT выполняет оба шага одновременно чрезвычайно компактным и элегантным способом. Тем не менее мы разберем каждый отдельный шаг, чтобы вы лучше поняли принцип его действия. И начать будет проще со второго шага (даже если вам покажется, что мы двигаемся задом наперед). Как применить операцию PHASE, у которой угол поворота для каждого значения в регистре кратен этому значению?

Поворот фазы каждого круга на угол, кратный его значению

При выражении целого числа в регистре QPU значение 0/1 k-го кубита обозначает наличие или отсутствие вклада 2^k в значение целого числа — как и в обычном двоичном регистре. Таким образом, чтобы выполнить любую заданную операцию с регистром *в количестве, представленном значением регистра*, необходимо выполнить операцию один раз для кубита с весом 2^0, два раза для кубита с весом 2^1 и т. д.

Происходящее проще всего понять на конкретном примере. Предположим, вы хотите выполнить k поворотов по 20° для регистра, где k — значение, хранимое в регистре. Это можно сделать так, как показано на рис. 7.29.

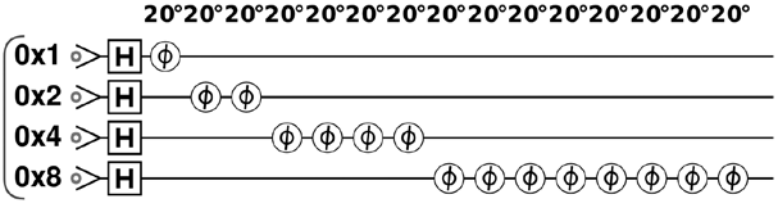


Рис. 7.29. Применение операции (PHASE) в количестве раз, заданном значением регистра

В самом начале также выполняются операции HAD, чтобы получить результат для каждого возможного значения k в суперпозиции. Код, использованный для генерирования этой схемы, приведен в листинге 7.8.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=7-8.

Листинг 7.8. QFT с поворотом фаз на кратные углы

```
// Повернуть k-е состояние в регистре k раз по 20 градусов
var phi = 20;

// Возьмем 4-кубитный регистр
qc.reset(4);
// Сначала выполнить операцию HAD, чтобы увидеть результат
// для всех k значений
qc.write(0);
qc.had();
// Применить 2^k операций phase к k-му кубиту
```

```
for (var i=0; i<4; i++) {
    var val = 1<<i;
    for (var j=0; j<val; j++) {
     qc.phase(phi, val);
    }
}
```

При выполнении листинга 7.8 будет получено k-е состояние, повернутое на угол $k \times 20^\circ$ (рис. 7.30). Именно то, что нужно!

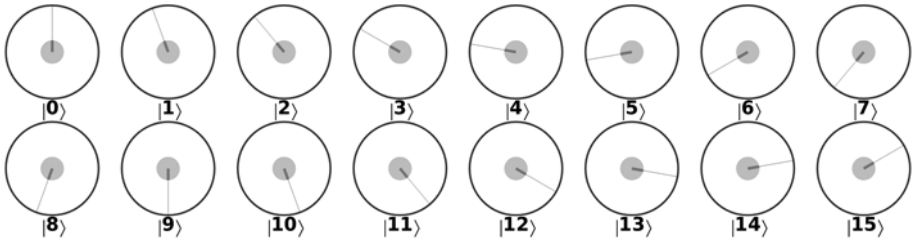


Рис. 7.30. Результат применения к каждому кубиту операций, определяемых двоичным весом кубита

Чтобы реализовать invQFT, необходимо выполнить этот прием с углом $n/2^N \times 360^\circ$ вместо 20° (где n — исходное значение в регистре).

Условный поворот на угол $n/2^N \times 360°$

Возвращаясь к рис. 7.25, мы видим, что схема invQFT состоит из четырех подсхем, показанных на рис. 7.31.

Каждая из этих подсхем выполняет множественные повороты с возрастающими весами, как показано на рис. 7.28. Вопрос в том, кратные какому углу

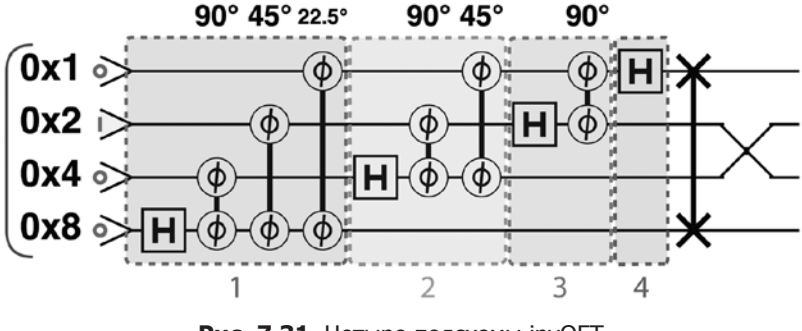


Рис. 7.31. Четыре подсхемы invQFT

повороты выполняются этими подсхемами? Мы видим, что первая подсхема поворачивает кубит с наибольшим весом на $n/2^N \times 360°$, тогда как вторая поворачивает второй по весу кубит на величину, большую в 2^1 раза, и т. д. (рис. 7.32) — в точности как требовалось на рис. 7.28.

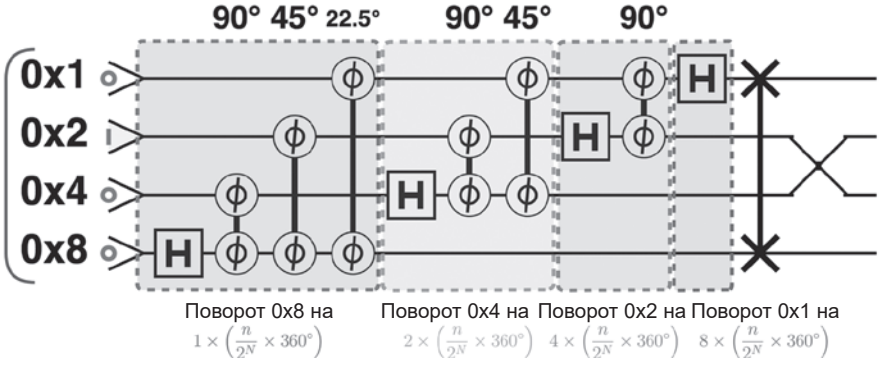


Рис. 7.32. Функция каждой подсхемы invQFT в реализации кратных поворотов фазы, определяемых входным регистром

Если вы внимательно следили за описанием, то заметили, что повороты на рис. 7.28 на самом деле применяются в обратном порядке, но вскоре мы покажем, как легко решить эту проблему.

Рассмотрим первую подсхему на рис. 7.32, изображенную более подробно на рис. 7.33. Вы можете убедиться в том, что она выполняет заявленный поворот на $n/2^N \times 360°$ с кубитом, имеющим наибольший вес (0x8).

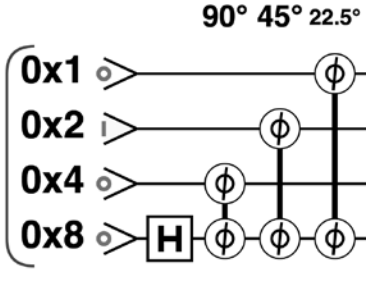


Рис. 7.33. Первая подсхема обратного преобразования QFT

Каждая операция CPHASE в этой подсхеме выполняет условный поворот кубита 0x8 на угол, который находится с 360° в той же пропорции, что и условный кубит к 2^N. Например, операция CPHASE, действующая между куби-

тами 0x4 и 0x8 на рис. 7.33, поворачивает кубит с наибольшим весом на 90°, а 4/24=90°/360°. Таким образом строится поворот на $n/2^N \times 360°$ для кубита 0x8 через все компоненты его двоичного расширения.

Но как насчет кубита с наибольшим весом? Это двоичное расширение также должно требовать выполнение условного поворота относительной фазы 180° для кубита с наибольшим весом, зависимого от значения кубита с наибольшим весом. Операция HAD на рис. 7.33 решает именно эту задачу (просто вспомните определяющее действие HAD на состояния $|0\rangle$ и $|1\rangle$). Эта операция HAD также служит другой цели: генерированию суперпозиции этого кубита из регистра, как того требует рис. 7.28. Видите, что мы имели в виду, говоря о компактности и элегантности этой схемы?

Каждая последующая подсхема на рис. 7.32 фактически сдвигает вверх вес угла, связанного с каждым кубитом (например, в подсхеме 2 кубит 0x2 связывается с поворотом 90° вместо 45°). В результате каждая подсхема умножает применяемую ей фазу $n/2^N \times 360°$ на 2 перед тем, как передавать ее конкретному кубиту, с которым она оперирует, — как и требовалось на рис. 7.32.

В совокупности эта схема выполняет условные повороты, необходимые для invQFT, но с одной проблемой. Все перевернуто вверх дном! Подсхема 1 поворачивает кубит 0x8 на однократную фазу, тогда как из рис. 7.28 видно, что поворот должен составлять ее восьмикратное значение. Именно для этого нужны обмены в самом конце рис. 7.32.

Таким образом, обратная схема QFT выполняет многочисленные действия, необходимые для получения периодически изменяющегося вывода, сжатые в один многоцелевой набор операций!

Чтобы упростить объяснение, мы ограничились одним целочисленным вводом для invQFT, но использованные операции QPU с таким же успехом работают с суперпозициями входных значений.

Итоги

В этой главе был рассмотрен один из самых мощных примитивов QPU — квантовое преобразование Фурье (QFT). Хотя примитив AA позволял извлечь информацию о дискретных значениях, закодированных в фазах регистра, примитив QFT позволяет извлечь информацию о *закономерностях* в информации, закодированной в регистре QPU. Как будет показано в главе 12, этот примитив лежит в основе самых мощных алгоритмов, выполняемых на QPU, включая алгоритм Шора, который изначально вызвал широкий интерес к квантовым вычислениям.

8 Квантовая оценка фазы

Квантовая оценка фазы (также называемая просто оценкой фазы) — еще один примитив QPU для вашего инструментария. Как и усиление комплексной амплитуды и QFT, оценка фазы представляет информацию, пригодную для чтения, из суперпозиций. Возможно, оценка фазы также станет самым сложным примитивом из рассмотренных до настоящего момента. Ее концептуальная сложность объясняется двумя причинами:

1. В отличие от примитивов AA и QFT, оценка фазы сообщает информацию об атрибуте *операции*, выполняемой с регистром QPU, а не об атрибуте состояния самого регистра QPU.
2. Конкретный атрибут операции QPU, информацию о котором предоставляет оценка фазы, на первый взгляд кажется бесполезным и субъективным, хотя он и играет исключительно важную роль во многих алгоритмах. Объяснить его практическую полезность без относительно сложной математики довольно трудно. И все же мы попробуем!

В этой главе мы расскажем, что такое оценка фазы, рассмотрим ряд практических примеров, а затем проанализируем этот примитив на уровне отдельных операций.

Получение информации об операциях QPU

Программирование задачи, которая должна быть решена на QPU, неизбежно подразумевает применение к регистрам QPU фундаментальных операций, описанных в главах 2 и 3. Сама мысль о примитиве, который нас чему-то научит, кажется странной — ведь если вы построили схему, то знаете о ней все, что только можно! Но некоторые виды входных данных могут быть закоди-

рованы в операциях QPU, так что получение более подробной информации о них может стать важным шагом на пути к искомому решению.

Например, в главе 13 будет показано, что алгоритм HHL для обращения некоторых матриц кодирует эти матрицы с использованием тщательно подобранных операций QPU. Свойства этих операций, раскрываемые квантовой оценкой фазы, сообщают критическую и нетривиальную информацию о матрице, которая должна быть обращена.

Собственные фазы предоставляют полезную информацию

Итак, какое же свойство операций QPU можно узнать при помощи оценки фазы? Пожалуй, на этот вопрос проще всего ответить конкретным примером. Обратимся к старому знакомому HAD. Вспомните, что при применении к однокубитному регистру HAD преобразует $|0\rangle$ и $|1\rangle$ в совершенно новые состояния (рис. 8.1).

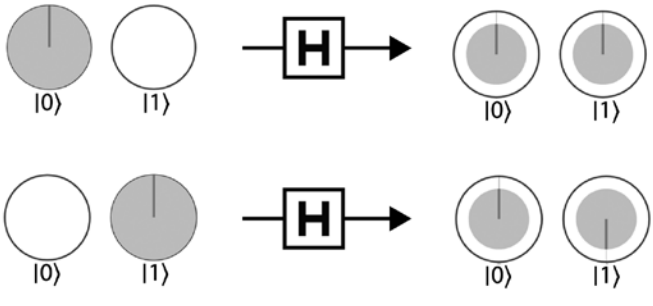


Рис. 8.1. Воздействие HAD на состояния $|0\rangle$ и $|1\rangle$

Для большинства других входных состояний HAD тоже производит совершенно новые выходные состояния. Но посмотрим, как HAD действует на два специальных входных состояния на рис. 8.2.

У первого входного состояния оба компонента имеют одинаковые фазы и 14,64%-ная вероятность получить при чтении состояние $|1\rangle$, тогда как у второго входного состояния компоненты расходятся по фазе на 180° и 14,64%-ная вероятность[1] получить при чтении $|0\rangle$.

[1] Более точное значение этой вероятности — $\sin^2(22{,}5)$.

Обратите внимание на то, как HAD работает с этими состояниями. Первое остается полностью неизменным, тогда как второе только приобретает *глобальную* фазу 180° (то есть относительная фаза остается неизменной). В главе 2 мы упоминали о том, что глобальные фазы не наблюдаемы, поэтому можно с таким же успехом сказать, что HAD фактически оставляет второе состояние неизменным.

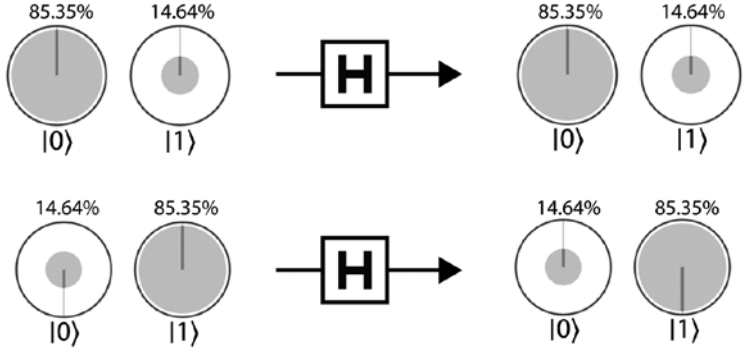


Рис. 8.2. Воздействие HAD на собственные состояния

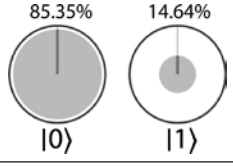
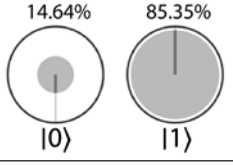
Состояния, невосприимчивые для некоторых операций QPU в этом отношении (не считая глобальных фаз), называются *собственными состояниями* (eigenstates) операции. У каждой операции QPU существует четко определенный уникальный набор таких состояний, для которой эта операция повлияет только на несущественную глобальную фазу.

Хотя глобальные фазы, которые могут приобретать собственные состояния, не наблюдаемы, они сообщают полезную информацию об операциях QPU, которые их произвели. Глобальная фаза, приобретаемая конкретным собственным состоянием, называется *собственной фазой*.

Как было показано ранее, операция HAD обладает двумя (и на самом деле только двумя) собственными состояниями с сопутствующими собственными фазами, перечисленными в табл. 8.1.

Стоит еще раз повторить, что конкретные собственные состояния (и связанные с ними собственные фазы) из табл. 8.1 относятся к HAD — у других операций QPU будут совершенно иные собственные состояния и собственные фазы. Вообще говоря, собственные состояния и собственные фазы операции QPU полностью определяют операцию в том смысле, что никакая другая операция QPU не обладает тем же набором.

Таблица 8.1. Собственные состояния и собственные фазы HAD

Собственное состояние	Собственная фаза
	0°
	180°

Что делает оценка фазы

Теперь, когда вы освоили терминологию собственных состояний и собственных фаз, можно описать, что делает примитив оценки фазы. Оценка фазы помогает определить собственные фазы, связанные с собственными состояниями операции QPU, возвращая суперпозицию всех собственных фаз. Это довольно выдающееся достижение, поскольку глобальные фазы обычно являются ненаблюдаемыми артефактами. Элегантность примитива оценки фазы заключается в том, что он находит способ перемещения информации о глобальной фазе в другой регистр — в форме, которая *может* читаться операцией READ.

Для чего бы нам понадобилось определять собственные фазы операций QPU? В последующих главах мы покажем, насколько полезными они могут быть, но предыдущее замечание о том, что они могут однозначно характеризовать операцию QPU, подсказывает, что такая информация может быть исключительно полезной.

Для читателей с подготовкой в области линейной алгебры: собственные состояния и собственные фазы являются собственными векторами и комплексными собственными фазами унитарных матриц, представляющих операции QPU в полной математической теории квантовых вычислений.

Как пользоваться оценкой фазы

Получив некоторое представление о том, что делает квантовая оценка фазы, перейдем от теории к практике и посмотрим, как ею пользоваться.

Предположим, имеется некоторая QPU-операция *U*, действующая на *n* кубитов и обладающая множеством собственных состояний, которые будут обозначаться u_1, u_2, ..., u_j. Методом оценки фазы можно узнать собственные фазы, связанные с каждым из собственных состояний. Не забудьте, что собственная фаза, связанная с j-м собственным состоянием, всегда является глобальной — а следовательно, может быть задана углом θ_j, на который глобальная фаза поворачивает состояние регистра. В этой системе обозначений можно сформулировать чуть более компактное описание задачи, выполняемой оценкой фазы:

«Для заданной QPU-операции *U* и одного из его собственных состояний u_j, оценка фазы возвращает (с определенной степенью точности) соответствующий угол собственной фазы θ_j».

В QCEngine для выполнения оценки фазы можно воспользоваться встроенной функцией `phase_est()` (реализация функции в элементарных операциях QPU приведена в листинге 8.2). Чтобы успешно вызвать этот примитив, необходимо понимать, какой ввод она рассчитывает получить и как должен интерпретироваться ее результат. Сводка входных и выходных данных приведена на рис. 8.3.

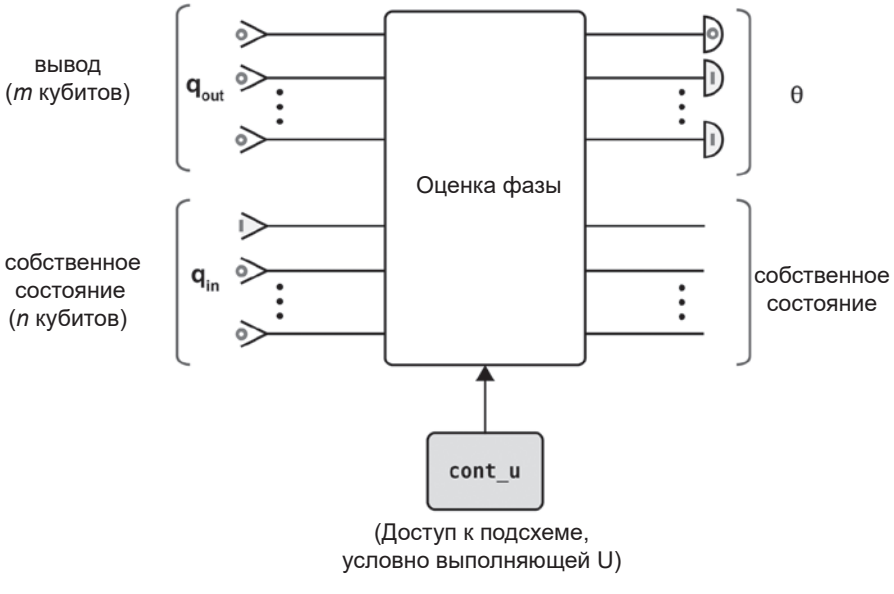


Рис. 8.3. Использование примитива оценки фазы

А теперь перейдем к более глубокому анализу аргументов `phase_est()`.

Ввод

Функция оценки фазы имеет следующую сигнатуру:

```
phase_est(qin, qout, cont_u)
```

`qin` и `qout` — регистры QPU, а аргумент `cont_u` должен содержать ссылку на функцию, которая выполняет интересующую вас операцию QPU (хотя и несколько специфическим образом, как вы вскоре увидите):

- `qin` — n-кубитный входной регистр, подготовленный в собственном состоянии u_j, для которого требуется получить собственную фазу;
- `qout` — второй регистр из m кубитов, инициализированный одними нулями. Примитив в конечном итоге использует этот регистр для двоичного представления нужного угла θ_j, соответствующего собственному состоянию, введенному в `qin`. В общем случае чем больше m, тем с большей точностью будет получено представление θ_j;
- `cont_u` — реализация управляемой версии QPU-операции U, для которой определяются собственные фазы. Должна передаваться в виде функции `cont_u(in, out)`, которая получает один кубит `in`, управляющий применением U к n-кубитному регистру `out`.

Чтобы привести конкретный пример использования оценки фазы, мы применим примитив для нахождения собственной фазы HAD. В табл. 8.1 было показано, что одно из собственных состояний HAD имеет собственную фазу 180° — посмотрим, сможет ли `phase_est()` воспроизвести этот результат (листинг 8.1).

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=8-1.

Листинг 8.1. Использование примитива оценки фазы

```
// Задать размер входного регистра - определяет точность
// ответа.
var m = 4;
// Задать размер входного регистра, который определяет
// собственное состояние.
var n = 1;
// Подготовка
```

```
qc.reset(m + n);
var qout = qint.new(m, 'output');
var qin = qint.new(n, 'input');
// Инициализировать выходной регистр нулями
qout.write(0);

// Инициализировать входной регистр собственным состоянием HAD
qin.write(0);
qin.roty(-135);
// Состояние имеет собственную фазу 180.
// Для собственной фазы 0 следует использовать qin.roty(45);

// Определить условную унитарную
function cont_u(qcontrol, qtarget, control_count) {
    // Для HAD достаточно знать, четно или нечетно значение control_
count
    // так как при применении HAD четное количество раз
    // ничего не происходит.
    if (control_count & 1)
        qtarget.chadamard(null, ~0, qcontrol.bits(control_count));
}
// Применить примитив оценки фазы к регистрам
phase_est(qin, qout, cont_u);
// Прочитать выходной регистр
qout.read();
```

Интересующее собственное состояние задается в `qin`, а `qout` инициализируется четырехкубитным регистром, заполненным нулями. Что касается `cont_u`, следует подчеркнуть, что передается не просто HAD, а функция, реализующая управляемую операцию HAD. Как будет показано позднее в этой главе, этого требуют внутренние механизмы оценки фазы. Так как генерирование управляемой версии любой операции QPU может быть довольно нетривиальной задачей, `phase_est()` предлагает пользователю задать функцию, которая удовлетворяет этому требованию. В этом конкретном примере используется `chadamard()` — управляемая версия HAD, встроенная в QCEngine.

На рис. 8.4 изображена схема, показывающая, чего следует ожидать от выполнения листинга 8.1.

Состояние входного регистра остается неизменным до и после применения `phase_est()`, как и ожидалось. Но что происходит с выходным регистром? Мы ожидали получить 180°, а получили 8!

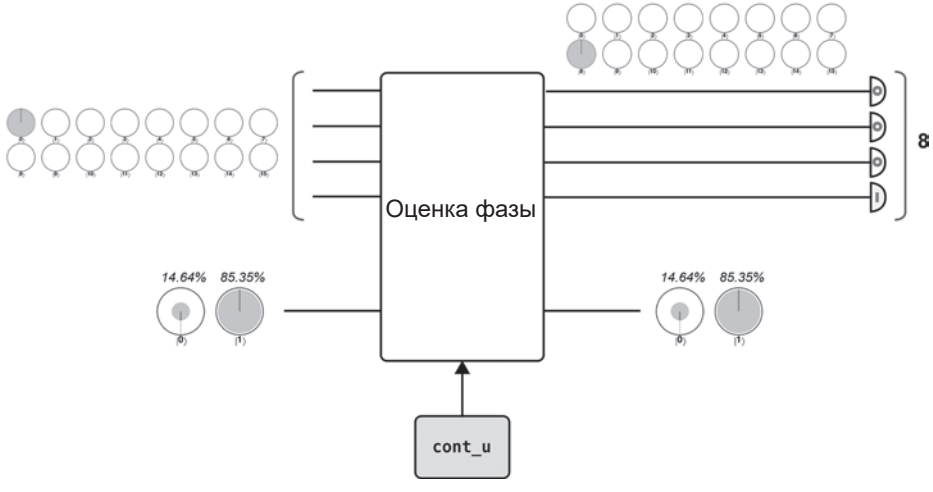


Рис. 8.4. Общая схема использования примитива оценки фазы

Вывод

Собственная фаза была получена применением операций READ к m кубитам выходного регистра. Важно заметить, что внутренние механизмы оценки фазы выражают θ_j как долю от 360°, что кодируется в выходном регистре как часть этого значения. Иначе говоря, если выходная собственная фаза была равна 90°, что составляет четверть от полного поворота, то следует ожидать, что для трехкубитного выходного регистра будет получено двоичное представление 2 — четверти от $2^3 = 8$ возможных значений регистра. Для случая на рис. 8.4 ожидалась фаза 180°, то есть половина полного поворота. Следовательно, в четырехкубитном регистре с $2^4 = 16$ значениями ожидается значение 8 — ровно половина размера регистра. Простое выражение, связывающее собственную фазу (θ_j) со значением регистра (R), как функция от размера регистра, приведено в формуле 8.1.

Формула 8.1. Связь между выходным регистром (R), собственной фазой (θ_j) и размером регистра m

$$R = \frac{\theta_j}{360°} \times 2^m .$$

Предостережения

Как это обычно бывает с программированием на QPU, необходимо учитывать ограничения, с которыми вы можете столкнуться. В том, что касается оценки фазы, следует помнить о некоторых неочевидных моментах.

Выбор размера выходного регистра

В листинге 8.1 собственная фаза, которую мы желали узнать, может быть идеально выражена в четырехкубитном представлении. Однако в общем случае точность определения собственной фазы зависит от размера выходного регистра. Например, трехкубитный выходной регистр позволяет точно представить следующие углы:

Двоичное представление	000	001	010	011	100	101	110	111
Часть регистра	0	1/8	1/4	3/8	1/2	5/8	3/4	7/8
Угол	0	45	90	135	180	225	270	315

Если попытаться использовать этот трехкубитный выходной регистр для определения собственной фазы со значением 150°, то разрешения регистра окажется недостаточно. Полное представление 150° потребует увеличения количества кубитов в выходном регистре.

Конечно, бесконечно увеличивать размер выходного регистра было бы нежелательно. В тех случаях, когда выходной регистр обладает недостаточным разрешением, регистр оказывается в суперпозиции, в центре которой находятся ближайшие возможные значения. Из-за этой суперпозиции лучшая оценка фазы с недостаточным разрешением будет возвращена только с некоторой вероятностью. Так, на рис. 8.5 изображено фактическое выходное состояние, полученное при выполнении оценки фазы при определении собственной фазы 150° для выходного регистра, состоящего только из трех кубитов. Полный пример доступен по адресу http://oreilly-qc.github.io?p=8-2.

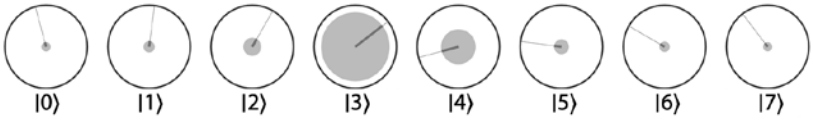


Рис. 8.5. Оценка фазы с превышением разрешения выходного регистра

Вероятность получения лучшей оценки всегда превышает 40%, и, конечно, эту вероятность можно повысить увеличением размера выходного регистра. Если вы хотите определить собственную фазу до p битов точности так, чтобы вероятность ошибки (то есть получения не лучшей оценки) не превышала ε, то необходимый размер выходного регистра m вычисляется по формуле:

$$m = p + \left\lceil \log\left(2 + \frac{1}{\varepsilon}\right) \right\rceil.$$

Сложность

Сложность примитива оценки фазы (в отношении количества необходимых операций) зависит от количества кубитов m в выходном регистре и описывается выражением $O(m^2)$. Очевидно, чем большая точность вам нужна, тем больше операций QPU потребуется. В разделе «Внутри QPU» показано, что эта зависимость в основном обусловлена зависимостью оценки фазы от примитива invQFT.

Условные операции

Возможно, самый неочевидный момент, связанный с оценкой фазы, — предположение о том, что вы можете обратиться к подсхеме, реализующей *управляемую* версию операции QPU, для которой нужно найти собственные фазы. Так как примитив оценки фазы многократно выполняет эту подсхему, очень важно, чтобы она выполнялась *эффективно*. Насколько эффективно — зависит от требований конкретного приложения, использующего оценку фазы. В общем случае, если подсхема `cont_u` имеет сложность выше $O(m^2)$, то общая эффективность примитива оценки фазы пострадает. Трудность нахождения таких эффективных подсхем зависит от конкретной операции QPU.

Оценка фазы на практике

Оценка фазы позволяет извлечь собственную фазу, связанную с конкретным собственным состоянием, для чего это собственное состояние должно быть задано во входном регистре. На первый взгляд это выглядит немного странно — часто ли требуется узнать собственную фазу при том, что собственное состояние вам уже известно?

Настоящая полезность оценки фазы заключается в том, что она — как и все хорошие операции QPU — может выполняться в суперпозиции! Если подать *суперпозицию собственных состояний* на вход примитива оценки фазы, мы получим суперпозицию сопутствующих собственных фаз. Амплитуда каждой собственной фазы в выходной суперпозиции будут в точности равна амплитуде, которую его собственное состояние имело во входном регистре.

Способность оценки фазы работать с суперпозициями собственных состояний делает этот примитив особенно полезным, так как выясняется, что *любое* состояние регистра QPU может рассматриваться как суперпозиция

собственных состояний любой операции QPU[1]. А это означает, что, если присвоить входному аргументу `cont_u` функции `phase_est()` некоторую операцию U, а `qin` — некоторое общее состояние регистра $|x\rangle$, то примитив вернет информацию о том, какие собственные фазы *характеризуют* действие U на $|x\rangle$. Такая информация будет полезной во многих математических вычислениях, в которых задействована линейная алгебра. Тот факт, что мы можем эффективно извлекать все эти собственные фазы в *суперпозиции*, расширяет возможности *параллелизации* этих математических вычислений на QPU (хотя, как обычно, не обходится без нюансов).

Внутри QPU

К внутренним механизмам оценки фазы определенно стоит присмотреться. Они не только расширяют примитив QFT, представленный в главе 7, но и играют центральную роль во многих приложениях QPU.

В листинге 8.2 приведена полная реализация функции `phase_est()`, которая впервые была использована в листинге 8.1.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=8-2.

Листинг 8.2. Реализация примитива оценки фазы

```
function phase_est(q_in, q_out, cont_u)
{
    // Один проход оценки фазы
    // Выполнить HAD с выходным регистром
    q_out.had();

    // Применить условные возможности u
    for (var j = 0; j < q_out.numBits; j++)
        cont_u(q_out, q_in, 1 << j);

    // Выполнить обратное преобразование QFT с выходным регистром
    q_out.invQFT();
}
```

[1] Этот факт не очевиден, но его можно продемонстрировать средствами полной математической теории квантовых вычислений. В главе 14 приводятся ссылки на технические ресурсы, в которых можно найти дополнительную информацию о том, почему это утверждение истинно.

Этот код реализует схему, изображенную на рис. 8.6.

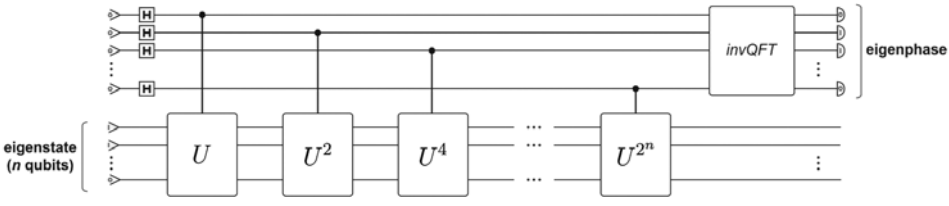


Рис. 8.6. Полная схема реализации оценки фазы

Весьма компактно! Остается понять, как эта схема умудряется извлечь собственные фазы операции QPU, переданной ей в параметре `cont_u`.

Интуитивное объяснение

Задача получения собственных фаз для операции QPU *кажется* довольно простой. Так как для `phase_est()` доступна анализируемая операция QPU и одно или несколько ее собственных состояний, почему бы просто не применить операцию QPU к собственному состоянию, как показано на рис. 8.7?

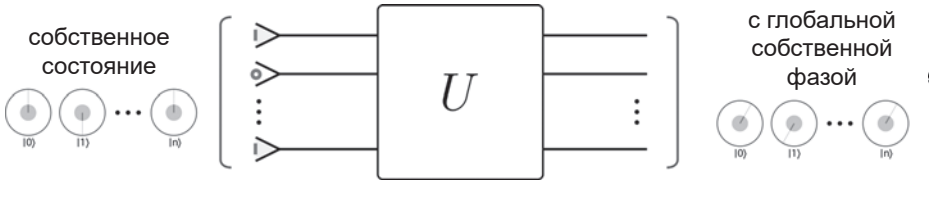


Рис. 8.7. Наивная схема для оценки фазы

Благодаря самому определению собственных состояний и фаз эта простая программа приводит к тому, что выходной регистр содержит то же входное собственное состояние, но с собственной фазой, примененной к нему как глобальная фаза. Хотя такое решение *представляет* собственную фазу θ в выходном регистре, ранее уже упоминалось о том, что *глобальная* фаза не может быть прочитана. Таким образом, эта простая идея сталкивается с отлично известной проблемой блокировки нужной информации в фазах регистра QPU.

Таким образом, нужно каким-то образом изменить рис. 8.7 для получения желаемой собственной фазы в свойстве регистра, доступном для чтения

операцией READ. Обратившись к растущему инструментарию примитивов, мы видим, что QFT выглядит перспективно.

Напомним, что QFT преобразует разности фаз в комплексные амплитуды, которые могут быть прочитаны. Таким образом, если мы найдем способ заставить относительные фазы в выходном регистре периодически изменяться с частотой, определяемой собственной фазой, успех гарантирован — нужно просто применить *обратное* преобразование QFT для чтения собственной фазы.

На рис. 8.8 показано, чего мы добиваемся, на примере двух разных собственных фаз.

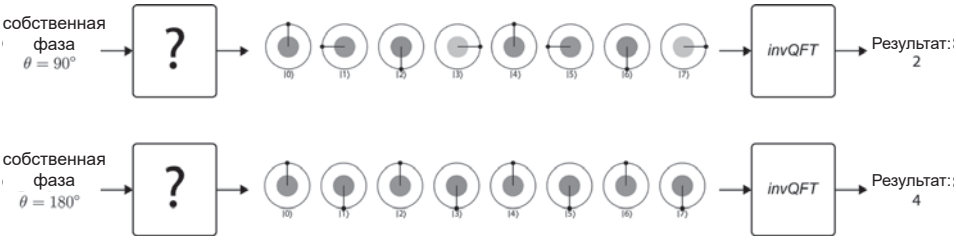


Рис. 8.8. Два примера представления собственных фаз в частотах регистров

Место вопросительных знаков должны занять группы операций QPU, которые производят эти результаты. Допустим, вы пытаетесь определить собственную фазу 90° (первый пример на рис. 8.8) и хотите закодировать ее в регистре, для чего относительные фазы в регистре поворачиваются с частотой $90/360 = 1/4$. Так как регистр имеет $2^3 = 8$ возможных состояний, это означает, что регистр должен выполнить два полных поворота по своей длине для получения частоты $2/8 = 1/4$. Конечно, при выполнении с этим регистром операции invQFT будет прочитано значение 2. Отсюда можно легко вывести собственную фазу:

$$2/8 = \theta/360° \rightarrow \theta = 90°.$$

Если поразмыслить над рис. 8.8, можно понять, что необходимые состояния можно получить при помощи нескольких тщательно подобранных поворотов. Вы просто берете равную суперпозицию всех возможных состояний регистра и поворачиваете k-е состояние k раз с любой нужной частотой. Таким образом, если вы хотите закодировать собственную фазу θ, k-е состояние будет повернуто на $k\theta$.

Эта процедура дает желаемые результаты на рис. 8.8, а на рис. 8.9 приведен пример кодирования собственной фазы 90° в восьми состояниях трехкубитного регистра.

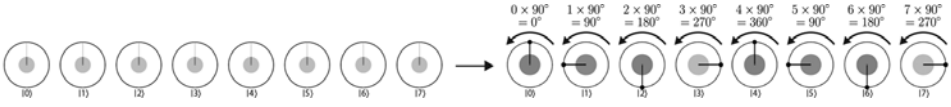


Рис. 8.9. Кодирование значения частоты регистра посредством условных поворотов

Итак, нам удалось переформулировать задачу следующим образом: *если k-е состояние регистра удастся повернуть k раз на собственную фазу, которая нас интересует, то мы сможем ее прочитать (при помощи обратного преобразования QFT).*

Идея поворота на углы, кратные значению регистра, выглядит знакомо? Вероятно, дело в том, что выбранный подход использовался для объяснения invQFT в главе 7. Впрочем, на этот раз частоты в регистре должны определяться собственной фазой операции.

Операция за операцией

Как вы сейчас увидите, схему для реализации подобных условных поворотов можно построить объединением подсхемы `cont_u` (предоставляющей условный доступ к U) с приемом фазовой коррекции, представленным в главе 3.

Каждый раз, когда операция U применяется к своему собственному состоянию, происходит глобальный поворот на ее собственную фазу. Глобальные фазы сами по себе особой пользы не приносят, но идею можно расширить и применить глобальную фазу, которая повернута на угол собственной фазы (заданный другим регистром QPU), умноженный на целый множитель k. Схема на рис. 8.10 решает эту задачу.

Каждый раз, когда операция U применяется к собственному состоянию в нижнем регистре, происходит поворот на собственную фазу θ. Выбранный набор условных операций просто выполняет количество поворотов, обусловленное каждым битом в двоичном представлении k, что приводит к применению U всего k раз в нижнем регистре. Таким образом выполняется поворот на $k\theta$.

Эта схема позволяет реализовать всего одну глобальную фазу для некоторого одиночного значения k. Чтобы выполнить трюк, требуемый для рис. 8.9, необходимо реализовать такой поворот для всех значений k реги-

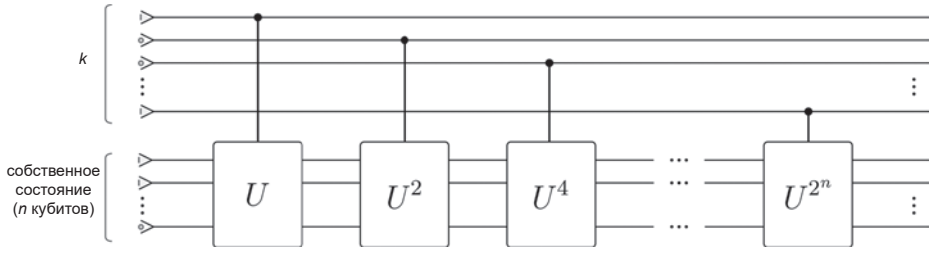


Рис. 8.10. Поворот на заданное количество повторений собственной фазы

стра в суперпозиции. Вместо того чтобы задавать одно значение k в верхнем регистре, воспользуемся равномерной суперпозицией всех 2^n возможных значений, как показано на рис. 8.11.

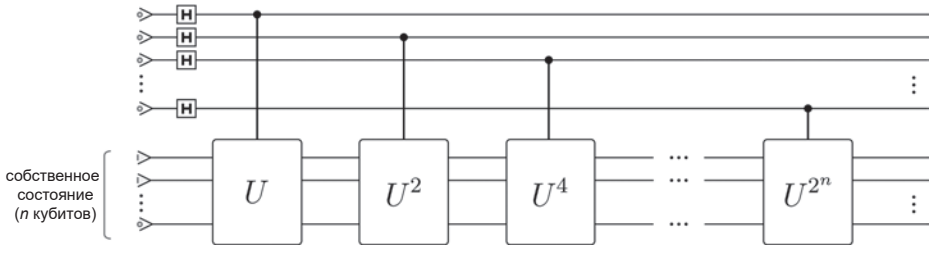


Рис. 8.11. Условный поворот всех состояний в регистре

Результат выполнения этой схемы для второго регистра: если первый регистр находился в состоянии $|0\rangle$, то глобальная фаза второго регистра поворачивается на угол $0 \times \theta = 0°$, а если первый регистр находился в состоянии $|1\rangle$, то второй регистр будет повернут на угол $1 \times \theta = \theta$ и т. д. Но и после всего возникает впечатление, что нам всего лишь удалось добиться довольно бесполезных (условных) глобальных фаз во втором регистре.

Что можно сказать о состоянии первого регистра, в котором изначально хранилась суперпозиция k значений? Вспомните, о чем говорилось в разделе «Приемы программирования QPU: фазовая коррекция»: после выполнения схемы можно эквивалентно представить, что каждое состояние в первом регистре обладает относительной фазой, повернутой на заданную величину. Другими словами, состояние $|0\rangle$ получает относительную фазу 0°, состояние $|1\rangle$ получает относительную фазу 90° и т. д. — то есть мы получаем именно то состояние, которое требуется на рис. 8.9. В точку!

Возможно, вам придется несколько раз перечитать это описание, чтобы понять, как мы использовали `cont_u` и фазовую коррекцию для извлечения

собственной фазы в частоту первого регистра. Но когда это будет сделано, останется лишь применить invQFT к регистру и прочитать его операцией READ, чтобы получить искомую собственную фазу. Полная схема реализации оценки фазы показана на рис. 8.12.

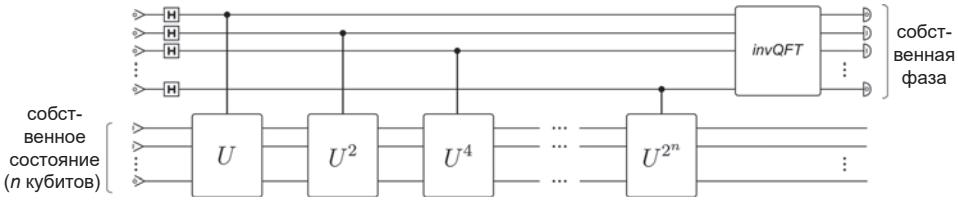


Рис. 8.12. Полная схема реализации оценки фазы

Теперь вы видите, почему необходимо иметь возможность предоставить подсхему для выполнения условной версии операции QPU. Также следует заметить, что нижний регистр останется в том же собственном состоянии в конце примитива. Хочется надеяться, что теперь вы понимаете, почему (благодаря invQFT) размер верхнего выходного регистра ограничивает точность примитива.

Способ выполнения степеней `cont_u` может оказать огромный эффект на эффективность оценки фазы. Наивное выполнение (допустим) U^4 четырехкратным последовательным вызовом `cont_u` чрезвычайно неэффективно. По этой причине можно передать `phase_est()` подсхему, которая может эффективно возвращать *n*-ю степень операции QPU (то есть `cont_u(n)`).

Итоги

В этой главе был рассмотрен новый примитив QPU — *оценка фазы*. Этот примитив использует три ранее представленные концепции (фазовая коррекция, построение управляемых унитарных операций и примитив invQFT) для решения нетривиальной задачи: извлечения информации, закодированной операциями QPU в *глобальных* фазах регистра. Для этого он преобразует информацию глобальной фазы в информацию относительной фазы во втором квантовом регистре, после чего применяет invQFT для извлечения этой информации в формат, пригодный для чтения операцией READ. Эта операция чрезвычайно важна для некоторых операций машинного обучения, которые будут представлены в главе 13.

ЧАСТЬ III
Практическое применение QPU

После знакомства с важнейшими примитивами QPU пришло время перейти от вопроса, как работают QPU, к вопросу, как использовать QPU, чтобы сделать что-то полезное.

Глава 9 начинается с демонстрации того, как полезные структуры данных (вместо простых целых чисел) представляются и хранятся в QPU. Затем в главе 10 приводятся рецепты, показывающие, как арифметические примитивы из главы 5 используются для решения общего класса задач, объединяемых общим термином «квантовый поиск». Затем в главе 11 мы перейдем к применениям QPU в компьютерной графике. В главе 12 будет представлен знаменитый алгоритм факторизации (разложения на простые множители), разработанный Питером Шором. Наконец, в главе 13 мы обратимся к практическим применениям наших примитивов QPU в области машинного обучения.

Поиск практических применений QPU идет полным ходом; этой теме посвящены серьезные научные исследования тысяч специалистов по всему миру. Будем надеяться, что эта часть книги даст необходимую подготовку и подтолкнет к тому, чтобы вы сделали свои первые шаги в поиске практических применений QPU в новых, еще не исследованных областях.

9 Реальные данные

Полноценные приложения для QPU строятся для работы с реальными, не учебными данными. Реальные данные далеко не всегда ограничиваются базовыми целыми числами, которыми мы обходились до сих пор. Следовательно, вопрос о том, как представить более сложные данные в QPU, стоит потраченных усилий, и хорошие структуры данных могут быть не менее важны, чем хорошие алгоритмы. В этой главе мы постараемся ответить на два вопроса, которые ранее обходили стороной:

1. *Как представить сложные типы данных в регистре QPU?* Положительное целое число можно представить в простой двоичной кодировке. Но что делать с иррациональными числами или даже с составными типами данных вроде векторов или матриц? Этот вопрос обретает новую глубину, если учесть, что суперпозиция и относительная фаза могут предоставить новые квантовые варианты кодирования таких типов данных.
2. *Как прочитать данные, хранящиеся в регистре QPU?* До сих пор мы инициализировали свои входные регистры вручную, используя операции WRITE для ручной инициализации кубитов регистра нужными двоичными числами. Если вы собираетесь применять квантовые приложения с большими массивами данных, эти данные нужно будет прочитать в регистры QPU из памяти. Это нетривиальное требование, так как, возможно, регистр QPU нужно будет инициализировать суперпозицией значений, а для этого традиционная оперативная память не приспособлена.

Начнем с первого вопроса. При описании представлений QPU для типов данных возрастающей сложности мы придем к введению полноценных квантовых структур данных и концепции квантовой оперативной памяти (QRAM). Квантовая оперативная память является критическим ресурсом для многих практических применений QPU.

Материал последующих глав будет сильно зависеть от структур данных, представленных в этой главе. Например, так называемое *комплексное амплитудное кодирование*, которое будет описано для векторных данных, занимает центральное место во всех квантовых приложениях машинного обучения, представленных в главе 13.

Нецелые данные

Как закодировать нецелые числовые данные в регистре QPU? Два стандартных способа представления таких значений в двоичной форме — представления *с фиксированной точкой* и *с плавающей точкой*[1]. Хотя представление с плавающей точкой обладает большей гибкостью (и способностью адаптироваться к диапазону значений, которые требуется представить с определенным количеством битов), из-за высокой ценности кубитов и нашего стремления к простоте представление с фиксированной точкой лучше подойдет для начала работы.

Числа с фиксированной точкой часто описываются в Q-записи (к сожалению, Q в данном случае не означает «квантовый»). Это помогает избавиться от неоднозначности относительно того, в какой позиции регистра кончаются биты дробной части и начинаются биты целой части. Запись *Qn.m* обозначает *n*-разрядный регистр, *m* разрядов которого предназначены для дробной части (а следовательно, оставшиеся ($n - m$) содержат целую часть). Конечно, при помощи той же записи можно указать, как регистр QPU должен использоваться для кодирования числа с фиксированной точкой. Например, на рис. 9.1 изображен восьмикубитный регистр QPU, в котором закодировано значение 3.640625 в представлении с фиксированной точкой Q8.6.

В приведенном примере выбранное число может быть точно закодировано в представлении с фиксированной точкой, потому что $3.640625 = 2^1+2^0+1/2^1+1/2^3+1/2^6$ (как удобно!). Конечно, такое везение встречается не всегда. Увеличение количества битов в целой части регистра с фиксированной точкой расширяет *диапазон* целочисленных значений, которые могут быть им представлены, а увеличение количества битов в дробной части повышает *точность* представления дробной части числа. Чем больше

[1] В русском языке допустимо использование записи десятичных чисел как через точку, так и через запятую. В компьютерной литературе мы осознанно используем десятичную точку, чтобы не допускать разночтения в фрагментах кода и формульной части. (Ср. вариант «0, 0.0001, 0.25, 0.784, 0.9995», который читается и воспринимается лучше, чем «0, 0,0001, 0,25, 0,784, 0,9995»). — *Примеч. ред.*

кубитов в дробной части, тем больше вероятность того, что некоторая комбинация $1/2^1$, $1/2^2$, $1/2^3$... сможет точно представить заданное число.

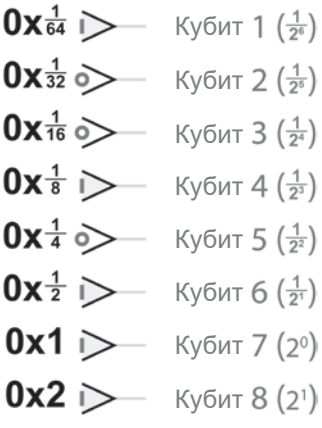


Рис. 9.1. Число 3.640625 (11101001 в двоичной записи), закодированное в представлении с фиксированной точкой Q8.6

И хотя в следующих главах мы будем кратко упоминать об использовании представления с фиксированной точкой, оно играет чрезвычайно важную роль в экспериментах с реальными данными в небольших регистрах QPU. Работая с разными способами кодировки, необходимо внимательно следить за тем, какая конкретная кодировка использовалась для данных в конкретном регистре QPU, чтобы правильно интерпретировать состояние его кубитов.

Будьте внимательны — в операциях, использующих дополнительную кодировку и представление с фиксированной точкой, часто возникает опасность переполнения, когда результат вычислений получается слишком большим для представления в регистре. Искаженный результат становится бессмысленным числом. К сожалению, у проблемы переполнения есть только одно реальное решение — добавление большего количества кубитов в регистры.

QRAM

В регистрах QPU можно хранить представления разных числовых значений, но как записать в них эти значения? Данные, инициализированные вручную, очень быстро устаревают. В действительности нам нужна возможность читать значения из памяти, с выборкой хранимых значений по двоичному адресу. Программист работает с традиционной оперативной

памятью при помощи двух регистров: один инициализируется адресом памяти, а другой остается неинициализированным. Оперативная память записывает во второй регистр двоичные данные, хранящиеся по адресу, заданному первым регистром, как показано на рис. 9.2.

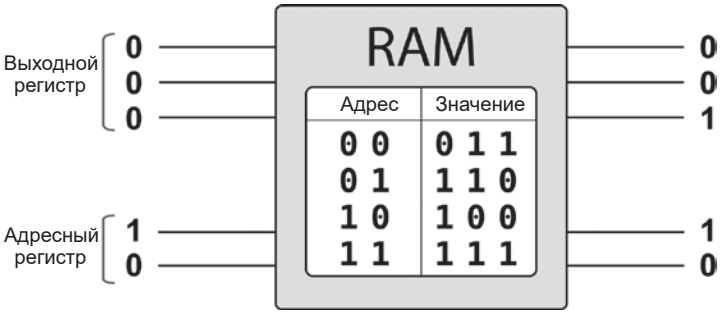


Рис. 9.2. Традиционная память — на диаграмме показаны хранимые значения и интерфейс для работы с ними

Можно ли использовать традиционную память для хранения значений, предназначенных для инициализации регистров QPU? Конечно, идея выглядит привлекательно.

Если вы хотите инициализировать регистр QPU только одним традиционным значением (в дополнительном коде, в представлении с фиксированной точкой или в простой двоичной кодировке), то оперативная память прекрасно подойдет. Нужное значение просто хранится в памяти, а для его записи или чтения из регистра QPU используются операции `write()` и `read()`. Именно этот ограниченный механизм использовался кодом QCEngine JavaScript для взаимодействия с регистрами QPU до настоящего момента.

Так, пример кода из листинга 9.1, который получает массив `a` и реализует операцию `a[2] += 1;`, неявно извлекает этот массив значений из оперативной памяти для инициализации регистра QPU. Схема изображена на рис. 9.3.

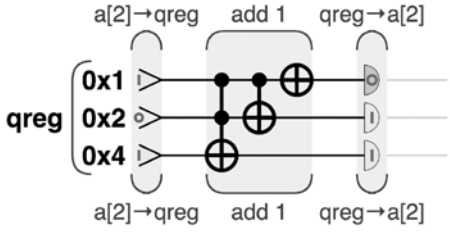


Рис. 9.3. Использование QPU для увеличения числа в памяти

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=9-1.

Листинг 9.1. Использование QPU для увеличения числа в памяти

```
var a = [4, 3, 5, 1];

qc.reset(3);
var qreg = qint.new(3, 'qreg');

qc.print(a);
increment(2, qreg);
qc.print(a);

function increment(index, qreg)
{
    qreg.write(a[index]);
    qreg.add(1);
    a[index] = qreg.read();
}
```

Стоит заметить, что в этом простом случае не только традиционная оперативная память используется для хранения целого числа, но и традиционный процессор выполняет индексирование массива для выбора и передачи QPU нужного значения.

Хотя такое использование оперативной памяти позволяет инициализировать регистры QPU простыми двоичными значениями, у него есть серьезные ограничения. Что, если потребуется инициализировать регистр QPU суперпозицией хранимых значений? Например, предположим, что в оперативной памяти значение 3 (110) хранится по адресу `0x01`, а значение 5 (111) — по адресу `0x11`. Как подготовить входной регистр в суперпозиции этих двух значений?

С традиционной оперативной памятью и неуклюжей традиционной операцией `write()` это сделать не удастся. Квантовым процессорам, как и когда-то их ламповым предкам, понадобится принципиально новое оборудование памяти — квантовое по своей природе. Знакомьтесь: квантовая оперативная память (QRAM) позволяет читать и записывать данные на квантовом уровне. Уже есть несколько идей относительно того, как физически строить QRAM, но стоит заметить, что история вполне может повториться, и невероятно мощные квантовые процессоры могут появиться задолго до того, как появится работоспособное оборудование квантовой памяти.

Стоит чуть точнее объяснить, что же делает QRAM. Как и традиционная память, QRAM получает на входе два регистра: *адресный* регистр QPU для адреса памяти и *выходной* регистр QPU, в котором возвращается значение, хранящееся по заданному адресу. Для QRAM оба регистра состоят из кубитов. Это означает, что в адресном регистре можно задать суперпозицию ячеек памяти и как следствие получить в выходном регистре суперпозицию соответствующих значений (рис. 9.4).

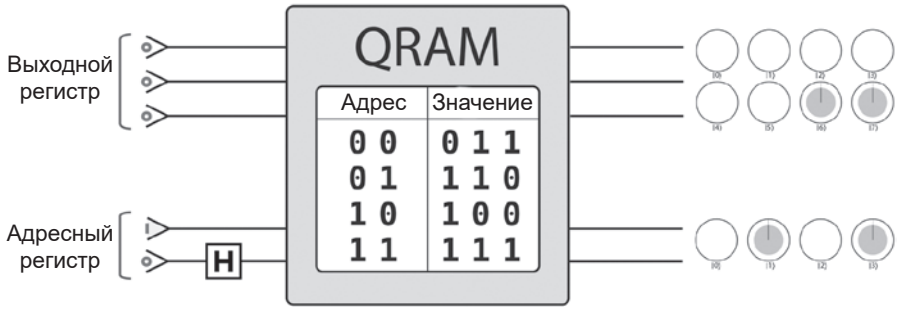


Рис. 9.4. QRAM — адресный регистр подготавливается в суперпозиции операцией HAD, а в результате будет получена суперпозиция хранимых значений (приводится в круговой записи)

Таким образом, QRAM фактически позволяет читать хранимые значения в суперпозиции. Точные комплексные амплитуды суперпозиции, которая будет получена в выходном регистре, определяются суперпозицией, предоставленной в адресном регистре. На рис. 9.2 показаны различия при выполнении той же операции инкремента в листинге 9.1 (рис. 9.5), но для обращения к данным используется QRAM вместо операций чтения/записи QPU. Буквой «A» обозначен регистр, в котором QRAM передается адрес (или суперпозиция). Буквой «D» обозначен регистр, в котором QRAM возвращает соответствующую суперпозицию хранимых значений (данных).

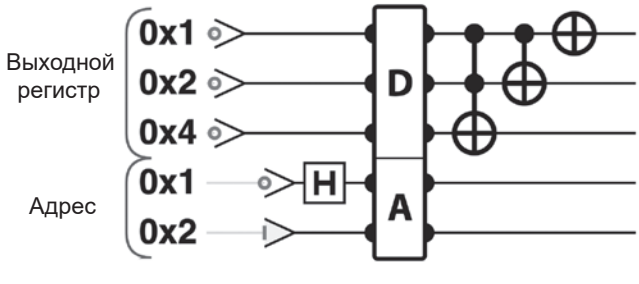


Рис. 9.5. Использование QRAM для реализации инкремента

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=9-2.

Листинг 9.2. Использование QPU для увеличения числа из QRAM — адресный регистр может содержать суперпозицию, что приведет к тому, что выходной регистр будет содержать суперпозицию хранимых значений

```
var a = [4, 3, 5, 1];
var reg_qubits = 3;
qc.reset(2 + reg_qubits + qram_qubits());
var qreg = qint.new(3, 'qreg');
var addr = qint.new(2, 'addr');
var qram = qram_initialize(a, reg_qubits);

qreg.write(0);
addr.write(2);
addr.hadamard(0x1);

qram_load(addr, qreg);
qreg.add(1);
```

Можно ли записывать суперпозиции обратно в QRAM? Квантовая память для этого не предназначена. Она позволяет обращаться к цифровым значениям, записанным традиционным образом, в суперпозиции. Долговременная квантовая память, способная хранить суперпозиции неограниченно долго, была бы совершенно другим устройством, которое было бы еще сложнее построить.

Такое описание QRAM может показаться слишком неопределенным — что же собой представляет оборудование квантовой памяти? В этой книге мы не станем приводить описание того, как построить QRAM на практике (как, скажем, в большинстве книг по C++ не приводится подробное описание того, как работает традиционная память). Примеры кода вроде приведенного в листинге 9.2 выполняются с использованием упрощенной модели, имитирующей поведение QRAM. Тем не менее прототипы технологий QRAM существуют[1].

Хотя квантовая память будет критическим компонентом любого серьезного QPU, подробности ее реализации с большой вероятностью изменятся, как у любого квантового вычислительного устройства. Для нас важна сама идея *фундаментального ресурса*, который ведет себя так, как показано на

[1] Например, см. Giovannetti et al., 2007, и Prakash, 2014.

рис. 9.4, и мощные приложения, которые могут быть построены на ее основе.

Имея в распоряжении квантовую память, можно переходить к построению сложных квантовых структур данных. Особый интерес представляют структуры, которые позволяют представлять векторные и матричные данные.

Кодирование векторов

Допустим, вы хотите инициализировать регистр QPU для представления простого вектора, вроде приведенного в формуле 9.1.

Формула 9.1. Пример вектора для инициализации регистра QPU

$$\vec{v} = [0,1,2,3].$$

Данные в такой форме часто встречаются в квантовых приложениях машинного обучения.

Пожалуй, самым очевидным методом кодирования векторных данных является представление каждого компонента в состоянии отдельного регистра QPU с подходящим двоичным представлением. Мы будем называть этот (вероятно, самый очевидный) метод *кодированием состояния* для векторов. Вектор из приведенного примера можно закодировать в четырех двухкубитных регистрах, как показано на рис. 9.6.

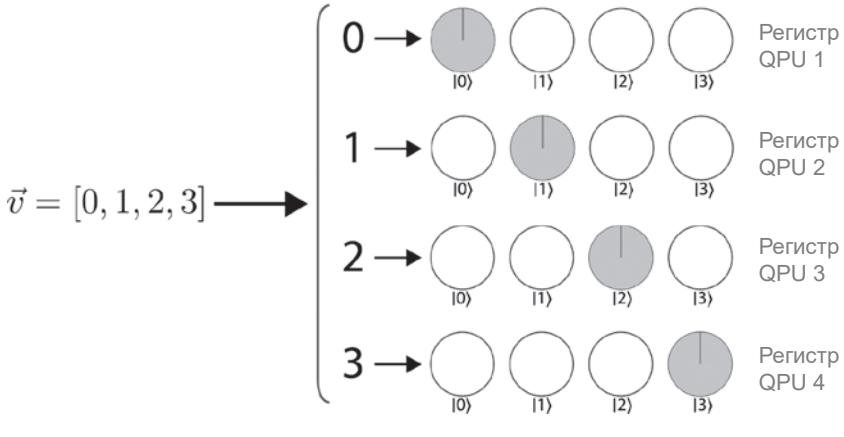


Рис. 9.6. Применение кодирования состояния для хранения данных вектора в регистрах QPU

Одна из проблем наивного кодирования состояния заключается в том, что оно неэффективно расходует кубиты — самый драгоценный ресурс QPU. Впрочем, к плюсам традиционных векторов с кодированием состояния следует отнести то, что они не требуют квантовой памяти. Компоненты вектора можно просто сохранить в стандартной памяти и использовать их отдельные значения для управления подготовкой каждого отдельного регистра QPU. Но это преимущество также лежит в основе самого серьезного недостатка кодирования состояния векторов: хранение векторных данных таким традиционным способом не позволяет нам использовать нетрадиционные возможности QPU. Чтобы пользоваться мощью QPU, необходимо уметь манипулировать относительными фазами суперпозиций, а это непросто сделать, если каждый компонент вектора фактически рассматривает ваш квантовый процессор как набор традиционных двоичных регистров!

Вместо этого необходимо спуститься на квантовый уровень. Предположим, компоненты вектора хранятся в суперпозиции амплитуд одного регистра QPU. Так как регистр QPU из n кубитов может существовать в суперпозиции с 2^n амплитудами (а следовательно, для экспериментов с круговой записью будет 2^n кругов), можно представить кодирование вектора с n компонентами в регистре QPU с ceil(log(n)) кубитами.

Для примера вектора из формулы 9.1 этот подход потребует двухкубитного регистра — идея заключается в том, чтобы найти подходящую квантовую схему для кодирования векторных данных на рис. 9.7.

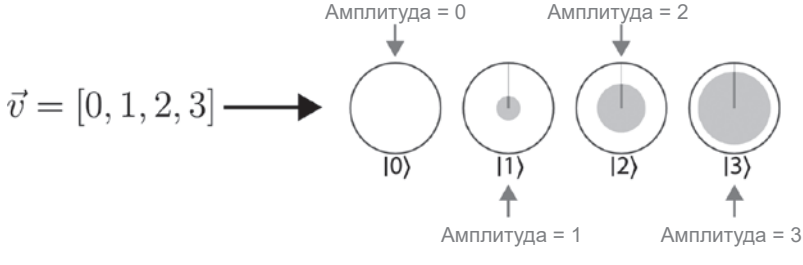
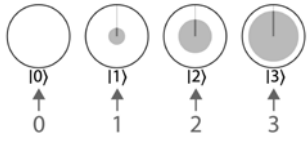
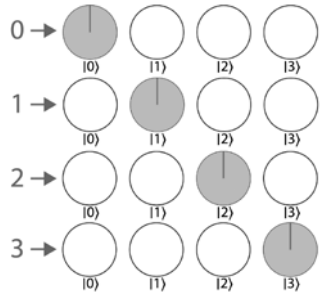
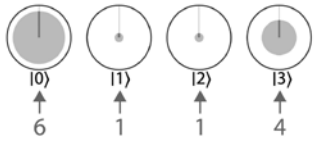
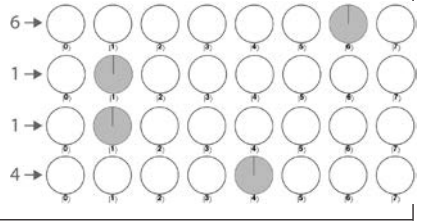
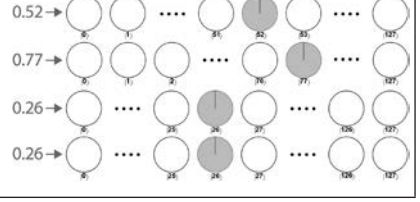


Рис. 9.7. Базовая идея комплексного амплитудного кодирования для векторов

Назовем этот уникальный квантовый способ кодирования векторных данных *комплексным амплитудным кодированием*. Важно оценить различия между комплексным амплитудным кодированием и более привычным кодированием состояния. В табл. 9.1 сравниваются эти два способа кодирования для разных векторных данных. В последнем примере кодирования со-

стояния понадобятся четыре семикубитных регистра, каждый из которых использует представление с фиксированной точкой Q7.7.

Таблица 9.1. Различия между способами кодирования векторных данных (комплексным амплитудным кодированием и кодированием состояния)

Вектор	Амплитудное кодирование	Кодирование состояния
`[0, 1, 2, 3]`		
`[6, 1, 1, 4]`		
`[0.52, 0.77, 0.26, 0.26]`		

Для получения векторов с комплексным амплитудным кодированием в QCEngine можно воспользоваться удобной функцией `amplitude_encode()`. Программа в листинге 9.3 получает вектор значений и ссылку на регистр QPU (который должен иметь достаточный размер) и подготавливает этот регистр, выполняя комплексное амплитудное кодирование вектора.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=9-3.

Листинг 9.3. Подготовка векторов с комплексным амплитудным кодированием в QCEngine

```
// Заранее следует убедиться в том, что длина входного вектора
// равна степени 2
var vector = [-1.0, 1.0, 1.0, 5.0, 5.0, 6.0, 6.0, 6.0];

// Создать регистр достаточного размера для вектора с комплексным
// амплитудным кодированием
var num_qubits = Math.log2(vector.length);
qc.reset(num_qubits);
var amp_enc_reg = qint.new(num_qubits, 'amp_enc_reg');

// Сгенерировать комплексное амплитудное кодирование в amp_enc_reg
amplitude_encode(vector, amp_enc_reg);
```

В этом примере вектор просто передается в виде массива JavaScript, хранящегося в традиционной памяти, — при том что мы указали, что комплексное амплитудное кодирование зависит от QRAM. Как QCEngine выполняет комплексное амплитудное кодирование, если для программы доступна только оперативная память вашего компьютера? Хотя возможно сгенерировать схему для комплексного амплитудного кодирования и без QRAM, сделать это эффективно, безусловно, не получится. QCEngine предоставляет медленную, но работоспособную модель того, что можно достигнуть с доступом к QRAM.

Ограничения комплексного амплитудного кодирования

Поначалу идея комплексного амплитудного кодирования выглядит отлично — оно использует меньше кубитов и предоставляет квантовые средства для работы с векторными данными. В любом приложении, в котором используется этот механизм, необходимо учитывать два важных фактора.

Проблема 1: квантовые результаты

Возможно, вы уже обратили внимание на первое из этих ограничений: *квантовые суперпозиции в общем случае не могут читаться операцией READ*. Снова наш главный противник! Если распределить компоненты вектора по квантовой суперпозиции, их не удастся прочитать снова. Естественно, это не создает особых проблем при передаче векторных данных на вход другой программы QPU из памяти. Но очень часто приложения для QPU, получающие векторные данные с комплексным амплитудным кодированием на входе, также генерируют векторные данные с комплексным амплитудным кодированием на выходе.

Таким образом, применение комплексного амплитудного кодирования жестко ограничивает нашу возможность чтения выходных данных приложений операцией READ. К счастью, из результатов с комплексным амплитудным кодированием часто возможно извлечь полезную информацию. Как будет показано в следующих главах, хотя вы не можете узнать отдельные компоненты, можно узнать глобальные свойства векторов, закодированных таким способом. Тем не менее комплексное амплитудное кодирование — не панацея, и его успешное применение требует внимания и изобретательности.

Если вы читали о квантовых приложениях машинного обучения, решающих традиционные задачи машинного обучения с невероятной скоростью, всегда обращайте внимание на то, возвращают ли они свои результаты в квантовой форме. Квантовые результаты, включая векторы с комплексным амплитудным кодированием, ограничивают возможности практического применения и требуют дополнительной спецификации, описывающей механизм извлечения полезных результатов.

Проблема 2: требование нормировки векторов

Вторая проблема, связанная с комплексным амплитудным кодированием, скрыта в табл. 9.1. Присмотритесь повнимательнее к комплексному амплитудному кодированию первых двух векторов в таблице: `[0,1,2,3]` и `[6,1,1,4]`. Могут ли комплексные амплитуды двухкубитного регистра QPU принять значения `[0,1,2,3]` или значения `[6,1,1,4]`? К сожалению, нет. В предыдущих главах мы обычно обходили обсуждение амплитуд и относительных фаз в пользу более интуитивной круговой записи. Хотя такой подход был более интуитивным, он оградил вас от одного важного числового правила, касающегося комплексных амплитуд: квадраты комплексных амплитуд регистра должны в сумме давать 1. Это требование, называемое *нормировкой,* выглядит логично, если вспомнить, что квадраты амплитуд в регистре соответствуют вероятностям чтения различных результатов. Так как должен быть получен один какой-то результат, эти вероятности (а следовательно, и квадраты всех комплексных амплитуд) должны в сумме давать 1. При использовании удобной круговой записи легко забыть о нормировке, но она устанавливает важное ограничение на то, к каким векторным данным может применяться комплексное амплитудное кодирование. Законы физики не позволяют создать регистр, находящийся в суперпозиции с комплексными амплитудами `[0,1,2,3]` или `[6,1,1,4]`.

Чтобы применить комплексное амплитудное кодирование к двум проблемным векторам из табл. 9.1, сначала необходимо их нормировать, разделив каждый компонент на сумму квадратов всех компонентов. Например, при

комплексном амплитудном кодировании вектора `[0,1,2,3]` сначала необходимо разделить все компоненты на 3,74 для получения нормированного вектора `[0.00, 0.27, 0.53, 0.80]`, который теперь подходит для кодирования в комплексных амплитудах суперпозиции.

Имеет ли нормировка векторных данных какие-либо нежелательные эффекты? Все выглядит так, словно данные полностью изменились! На самом деле нормировка оставляет большую часть важной информации без изменений (в геометрическом представлении она всего лишь масштабирует длину вектора, оставляя направление без изменений). Можно ли считать, что нормализованные данные полностью заменяют исходные? Это зависит от потребностей конкретного приложения QPU, в котором вы собираетесь их использовать. Помните, что при необходимости числовое значение коэффициента нормализации можно хранить в другом регистре.

Комплексное амплитудное кодирование и круговая запись

Когда вы начинаете более конкретно думать о числовых значениях комплексных амплитуд регистров, будет полезно напомнить себе, как комплексные амплитуды представляются в круговой записи, и заметить возможную ловушку. Заполненные области в круговой записи представляют *квадраты* амплитуд комплексных амплитуд квантового состояния. В таких ситуациях, как при комплексном амплитудном кодировании, когда комплексные амплитуды должны представлять компоненты вектора с вещественными значениями, это означает, что заполненные области определяются *квадратом* соответствующего компонента вектора, а не самим компонентом. На рис. 9.8 показано, как правильно интерпретировать представление вектора `[0,1,2,3]` после нормализации в круговой записи.

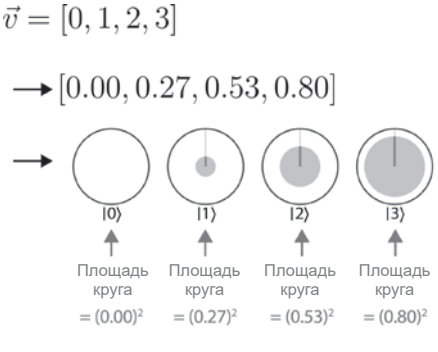


Рис. 9.8. Правильное комплексное амплитудное кодирование с нормализованным вектором

Оценивая числовые значения комплексных амплитуд из круговой записи, не забывайте о том, что заполненная часть круга определяется квадратом амплитуды состояния (а следовательно, и комплексные амплитуды для чисто вещественных случаев). Будьте внимательны и не забывайте о квадрате!

Теперь вы знаете о векторах с комплексным амплитудным кодированием достаточно, чтобы разобраться в приложениях для QPU, которые будут приведены в книге. Но для многих приложений, особенно относящихся к области квантового машинного обучения, необходимо пойти на шаг дальше и использовать QPU для манипуляций не только с векторами, но и с целыми *матрицами* данных. Как же закодировать двумерные массивы чисел?

Хотя во всех примерах векторов, встречавшихся до настоящего момента, использовались только вещественные компоненты, поскольку относительные фазы суперпозиции позволяют в общем случае рассматривать их комплексные амплитуды как комплексные числа, стоит заметить, что комплексное амплитудное кодирование позволяет легко представлять (нормализованные) комплексные векторы в регистре QPU. Именно по этой причине комплексное амплитудное кодирование не называется амплитудным кодированием — вы можете использовать полную комплексную амплитуду в суперпозиции для кодирования комплексных значений, хотя в этой главе данная возможность не используется.

Кодирование матриц

В самом очевидном варианте кодирования матрицы из $m \times n$ значений используются m регистров QPU, каждый из которых имеет длину $\log_2(n)$, с комплексным амплитудным кодированием каждой строки матрицы так, как если бы она была вектором. Хотя этот способ, несомненно, позволит передать матричные данные QPU, это не означает, что он очень хорошо представляет данные *в виде матрицы*. Например, совершенно не очевидно, насколько хорошо этот подход позволит выполнять такие фундаментальные операции, как транспонирование или умножение матрицы с векторами в комплексном амплитудном кодировании, хранящимися в других регистрах QPU.

Выбор лучшего способа кодирования матрицы в регистрах QPU зависит от того, насколько точно вы хотите позднее использовать эту матрицу в приложениях QPU. По крайней мере, на момент написания книги существовало несколько способов кодирования матриц, широко применявшихся на практике.

Существует одно универсальное требование, предъявляемое к кодированию матриц. Так как операции матриц с векторами данных (умножение) так

часто встречаются на практике, а данные векторов кодируются в регистрах QPU, будет разумно рассматривать кодирование матриц как операции QPU, способные работать с регистрами, в которых хранятся векторы. Осмысленное представление матрицы в виде операции QPU — непростая задача, и у каждого существующего метода для ее решения есть свои важные достоинства и недостатки. Мы сосредоточимся на одном очень популярном методе, который называется *квантовым моделированием*. Но прежде чем этим заниматься, стоит разобраться в том, чего же мы пытаемся добиться.

Как матрицы могут представляться операциями QPU?

Что же мы имеем в виду, говоря, что операция QPU правильно *представляет* некоторую матрицу данных? Допустим, вы узнали, как все возможные векторы, с которыми может работать матрица, могут быть закодированы в регистре QPU (по некоторой схеме — например, с применением комплексного амплитудного кодирования). Если в результате применения операции QPU к таким регистрам в выходных регистрах будут закодированы именно те векторы, которые должны быть получены при выполнении операции с *матрицей*, можно с уверенностью утверждать, что операция QPU отражает поведение матрицы.

При описании оценки фазы в главе 8 мы упоминали о том, что операция QPU полностью характеризуется собственными состояниями и собственными фазами. Аналогичным образом матрица полностью характеризуется своим *собственным спектральным разложением* (eigendecomposition). Таким образом, убедиться в том, что операция QPU точно воспроизводит матрицу данных, можно более простым способом: нужно проверить, имеют ли они одинаковое собственное разложение. Под термином «собственное разложение» мы понимаем то, что собственные состояния операции QPU являются собственными векторами исходной матрицы (в комплексном амплитудном кодировании), а собственные фазы связаны с собственными значениями матрицы. В таком случае можно утверждать, что операция QPU реализует поведение нужной матрицы с векторами в комплексном амплитудном кодировании. Анализ операции QPU, имеющей такое же собственное разложение, позволит получить достоверные ответы на вопросы, относящиеся к закодированной матрице.

Термином «собственное разложение» обозначается набор собственных значений и собственных векторов матрицы. Термин также можно применить к операции QPU; в этом случае он обозначает набор собственных состояний и собственных фаз, связанных с этой операцией.

Предположим, операция QPU с таким же собственным разложением, как у кодируемой матрицы, найдена. Это все, что необходимо? Почти. Когда вы запрашиваете *представление матрицы в виде операции QPU*, одного абстрактного математического описания подходящей операции QPU недостаточно. С практической точки зрения потребуется детальная спецификация того, как должна выполняться эта операция в простых одно- и многокубитных операциях, представленных в главах 2 и 3. Кроме того, эта спецификация должна быть эффективной в том смысле, чтобы из-за слишком большого количества операций с матричными данными приложение для QPU не стало слишком медленным. Таким образом, для наших целей можно сформулировать желаемый результат более конкретно:

> хорошее представление матрицы — процедура, связывающая матрицу с операцией QPU, которая может быть эффективно реализована на уровне базовых одно- и многокубитных операций.

Для некоторых типов матриц процедура *квантового моделирования* предоставляет хорошие представления матриц.

Квантовое моделирование

Термином «квантовое моделирование» в действительности обозначается целый класс процедур, способных находить эффективно реализуемые операции для представления эрмитовых матриц.

Эрмитовыми матрицами называются матрицы, для которых выполняется условие $H = H^{\dagger}$. Здесь $H^{\dagger}$ — эрмитово-сопряженная матрица, которая вычисляется транспонированием и вычислением комплексного сопряжения (потенциально комплексной) матрицы.

Методы квантового моделирования предоставляют операцию QPU, которая имеет такое же собственное разложение, как и исходная эрмитова матрица. Все многочисленные методы квантового моделирования строят схемы с различными ресурсными требованиями и могут даже устанавливать разные дополнительные ограничения для типов представляемых матриц. Впрочем, все методы квантового моделирования как минимум требуют того, чтобы кодируемая матрица была эрмитовой.

Но разве требование того, чтобы матрица реальных данных была эрмитовой, не сужает применимость методов квантового моделирования до минимума? Как выясняется, способность представления только эрмитовых матриц не создает таких серьезных ограничений, как может показаться на

первый взгляд. Кодирование не-эрмитовой матрицы X размером $m \times n$ может быть реализовано построением большей эрмитовой матрицы $2m \times 2n$ H следующим образом:

$$H = \begin{bmatrix} 0 & X \\ X^\dagger & 0 \end{bmatrix},$$

где 0 на диагонали представляют блоки $m \times n$, заполненные нулями. Постоянные затраты на создание большей матрицы обычно оказываются относительно незначительными.

Мы опишем общий высокоуровневый метод, используемый многими методами квантового моделирования, чуть подробнее остановившись на одном конкретном методе в качестве примера. При этом в тексте будет чуть больше математических выкладок, чем до настоящего момента, но это будет только линейная алгебра, что в порядке вещей при работе с матрицами.

Основная идея

В книге мы в основном используем круговую запись, но ранее упоминалось о том, что полностью квантово-механическое описание состояния регистра QPU представляет собой вектор с комплексными значениями. При этом полностью квантово-механическое описание операции QPU представляет собой матрицу. Может возникнуть впечатление, что наша цель по кодированию матриц в форме операций QPU достигается просто. Если по сути операции QPU описываются матрицами, нужно просто найти операцию с такой же матрицей, как у кодируемых данных! К сожалению, лишь некоторое подмножество матриц соответствует действительным (реально создаваемым) операциям QPU — а именно действительные операции QPU описываются *унитарными* матрицами.

Матрица U называется унитарной, если для нее выполняется условие $UU^\dagger = 1$, где 1 — единичная матрица (содержащая единицы на диагонали и нули в остальных ячейках) такого же размера, как U. Операции QPU должны описываться унитарными матрицами, так как это гарантирует, что реализующие их схемы будут обратимыми (это требование упоминалось в главе 5).

К счастью, для заданной эрмитовой матрицы H связанная с ней унитарная матрица U может быть построена возведением в степень: матрица $U = \exp(-iHt)$ унитарна, если матрица H является эрмитовой. Методы квантового моделирования используют этот факт (именно поэтому они ограничи-

ваются представлением эрмитовых матриц). Значение t в показателе степени — время применения операции QPU $U = \exp(-iHt)$. Для своих целей мы будем считать t подробностью реализации, которую можно пропустить для простоты объяснения.

Таким образом, задача квантового моделирования заключается в эффективном предоставлении схемы, выполняющей возведение в степень H.

Хотя мы используем квантовое моделирование для кодирования матричных данных, оно называется так потому, что предназначено в первую очередь для моделирования поведения квантовомеханических объектов (например, в молекулярном моделировании или моделировании материалов). При моделировании квантовых объектов некоторая эрмитова матрица (известная в физике как матрица гамильтониана) математически описывает выполняемое моделирование, а операция QPU $\exp(-iHt)$ прогнозирует изменение квантового объекта со временем. Этот метод интенсивно используется в алгоритмах QPU из области квантовой химии; за дополнительной информацией обращайтесь к разделу «Перспективы квантового моделирования».

Для некоторых особенно простых эрмитовых матриц H найти набор простых операций QPU, реализующих $\exp(-iHt)$, относительно просто. Например, если H содержит элементы только на главной диагонали или работает только с очень малым количеством кубитов, то найти такую схему несложно.

Впрочем, маловероятно, что эрмитова матрица данных будет сходу удовлетворять этим требованиям, поэтому квантовое моделирование дает возможность разбить такие трудно кодируемые матрицы на несколько матриц, которые кодируются проще. В следующем разделе мы в общих чертах расскажем, как это делается. И хотя мы не предоставляем подробных алгоритмов квантового моделирования, наше высокоуровневое описание, по крайней мере, поможет продемонстрировать важные ограничения этих методов.

Как это работает

Многочисленные методы квантового моделирования используют похожие процедуры. Для заданной эрмитовой матрицы H выполняется следующая последовательность действий.

1. *Деконструирование*. Ищется способ разбиения H на *сумму* некоторого числа (n) других, более простых эрмитовых матриц $H = H_1 + ... H_n$. Матрицы $H_1 ... H_n$ названы более простыми в том смысле, что их проще эффективно моделировать способом, описанным в предыдущем разделе.

2. *Моделирование компонентов*. Затем выполняется эффективный поиск квантовых схем (выраженных в фундаментальных операциях QPU) для этих более простых составляющих матриц.
3. *Реконструирование*. Схема реализации полной матрицы H воссоздается из меньших квантовых схем, найденных для ее деконструированных компонентов.

Чтобы этот план заработал, необходимо пояснить два шага: поиск способа разбиения эрмитовой матрицы на сумму легко моделируемых составляющих (шаг 1) и описание объединения моделей меньших составляющих в полноценную модель H (шаг 3). Методы квантового моделирования отличаются друг от друга по тому, как они подходят к решению этой задачи. Ниже приведена сводка одной группы таких методов, называемых *методами формулы произведения* (product formula methods).

На самом деле проще начать с конца и сначала понять, как методы формулы произведения выполняют завершающее реконструирование H (шаг 3 в приведенном списке).

Реконструирование

Предположим, вы нашли некоторый способ записи $H=H_1+...+H_n$, где H_1, ..., H_n — эрмитовы матрицы, для которых можно легко найти операции QPU. В таком случае саму матрицу H можно реконструировать благодаря математическому отношению, называемому *формулой произведения Ли*. Эта формула позволяет аппроксимировать унитарную матрицу $U=\exp(-iHt)$ последовательным выполнением всех составляющих операций QPU

$$U_1 = \exp(-iH_1\delta t), ..., U_n = \exp(-iH_n\delta t)$$

на очень коротких временных промежутках δt и повторением всей процедуры некоторое количество раз m, как показано на рис. 9.9.

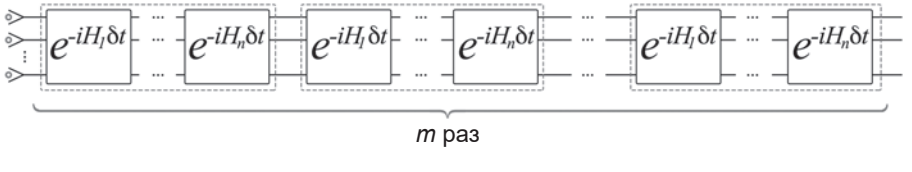


Рис. 9.9. Квантовое моделирование реконструирует трудно моделируемую эрмитову матрицу повторным моделированием серии эрмитовых матриц, которые проще моделируются

По сути, формула произведения Ли показывает, что если H можно деконструировать H на матрицы с эффективными схемами, то также возможно аппроксимировать $U = \exp(-iHt)$ с эффективным временем выполнения.

Деконструирование H

С шагом 3 мы разобрались, но как изначально деконструировать матрицу на сумму просто моделируемых эрмитовых матриц?

Разные методы квантового моделирования из категории формул произведений выполняют эту деконструкцию по-разному. Все методы имеют серьезную математическую базу и устанавливают разные дополнительные требования для H. Например, один из методов, подходящий для разреженных матриц P (с возможностью эффективного обращения к разреженным компонентам), основан на интерпретации H как матрицы смежности для графа. В результате решения определенной задачи раскраски графа в найденных цветах объединяются элементы H, которые образуют просто моделируемые составляющие H_1, ...H_n, полученные в результате деконструирования.

Здесь термин «граф» используется в математическом смысле — как структура из набора вершин, соединенных ребрами. В задаче раскраски графа требуется назначить каждой вершине один из нескольких возможных цветов с тем условием, что две вершины, соединенные ребром, не могут быть окрашены в один цвет. Возможно, связь деконструирования эрмитовых матриц с раскраской графа не столь очевидна, но она происходит от математической структуры, лежащей в основе задачи.

Стоимость в квантовом моделировании

Надеемся, это краткое описание методов формул произведений дает некоторое представление о тех усилиях, которые приходится предпринимать для представления матричных данных в форме операций QPU. Как упоминалось ранее, также существуют другие методы квантового моделирования, многие из которых обладают более высокой эффективностью, потому что они либо требуют меньших схем, либо меньшего количества обращений к кодируемой матрице. В табл. 9.2 сравнивается время выполнения некоторых распространенных методов квантового моделирования. В данном случае под «временем выполнения» подразумевается размер схемы, генерируемой методом для моделирования матрицы (размер схемы определяется количеством необходимых фундаментальных операций QPU). d — характеристика разреженности матрицы (максимальное количество

ненулевых элементов в строке), а ε — характеристика требуемой точности представления[1].

Таблица 9.2. Время выполнения некоторых методов квантового моделирования

Метод	Время выполнения схемы
Формула произведения [a]	$0\ (d^4)$
Квантовый обход	$0\left(d/\sqrt{\varepsilon}\right)$
Квантовая обработка сигнала	$0\left(d+\frac{\log(1/\varepsilon)}{\log\log(1/\varepsilon)}\right)$

[a] Время выполнения методов формул произведения можно немного улучшить за счет использования аппроксимаций «высшего порядка» в формуле произведения Ли.

[1] Также следует отметить, что время выполнения зависит от других важных параметров, таких как входная матрица и используемый метод квантового моделирования. Для простоты мы не стали включать их в сводку.

10 Квантовый поиск

В главе 6 было показано, как примитив усиления комплексной амплитуды (AA) преобразует разности фаз в регистре в обнаруживаемые разности амплитуды. Вспомните: при описании AA предполагалось, что приложения предоставляют процедуру для инвертирования фаз значений в регистре QPU. В упрощенном примере мы использовали схему, которая просто инвертировала фазу одного известного значения регистра. В этой главе подробно рассмотрены некоторые методы изменения фазы в квантовом состоянии, в зависимости от результатов нетривиальной логики.

Квантовый поиск (QS, Quantum Search) — метод, позволяющий AA надежно читать решения из регистра QPU для определенного класса задач. Иначе говоря, QS всего лишь является практическим применением AA, для которого предоставляется критически важная подсхема[1], помечающая решения некоторого класса задач в фазах регистра.

Класс задач, которые позволяет решать квантовый поиск, составляют задачи, многократно выполняющие подсхему с получением ответа «да/нет». Ответ «да/нет» этой подсхемы в общем случае является выводом традиционной логической команды[2].

Одна из очевидных задач, которые могут быть представлены в этой форме, — поиск конкретного значения в базе данных. Просто представьте логическую

---

[1] В литературе функция, изменяющая фазы в соответствии с некоторой логической функцией, называется оракулом (примерно в том же смысле, в котором этот термин используется в традиционной компьютерной теории). Мы остановились на более обыденной терминологии, но в главе 14 еще свяжем ее с популярным техническим жаргоном.

[2] Более подробное описание этого класса задач предоставляется в конце этой главы и в главе 14.

функцию, которая возвращает 1 в том и только в том случае, если входным значением является искомый элемент базы данных. Собственно, именно это применение квантового поиска было классическим; оно названо *алгоритмом Гровера* (по имени создателя). Применяя метод квантового поиска, алгоритм Гровера может найти элемент в базе данных всего за $O(\sqrt{N})$ запросов к базе данных вместо традиционных $O(N)$. Однако такой подход предполагает неструктурированную базу данных (довольно редкое явление), а его реализация сталкивается с существенными практическими препятствиями.

Хотя алгоритм Гровера является самым известным применением квантового поиска, существует много других применений, использующих квантовый поиск для повышения производительности, в самых разных областях — от искусственного интеллекта до верификации программного обеспечения.

В картине не хватает одного фрагмента: как же квантовый поиск позволяет находить подсхемы, кодирующие вывод любой логической команды в фазах регистра QPU (при поиске в базе данных алгоритмом Гровера или при другом применении квантового поиска)? Если вы будете знать, как это делается, AA сделает все остальное. Чтобы понять, как могут строиться такие подсхемы, нам понадобится сложный набор инструментов для манипуляций с фазами регистров QPU — инструментарий, который мы обозначим термином *фазовая логика*. В оставшейся части этой главы мы будем описывать фазовую логику и покажем, как она может быть задействована в квантовом поиске. В конце главы приводится общий рецепт применения методов квантового поиска в различных традиционных задачах.

Фазовая логика

В главе 5 была представлена разновидность квантовой логики — то есть способ выполнения логических функций, совместимых с квантовыми суперпозициями. Однако эти логические операции использовали в качестве входных данных значения регистров с ненулевыми *амплитудами* (например, $|2\rangle$ или $|5\rangle$) и выводили результаты в значениях регистров (возможно, в служебных кубитах).

С другой стороны, квантовая фазовая логика, необходимая для квантового поиска, должна выводить результаты логических операций в *относительных фазах* этих регистров.

А конкретнее, для выполнения фазовой логики мы ищем схемы QPU, которые могут реализовать квантовую фазовую логику, представляющую за-

данную логическую операцию (AND, OR и т. д.) инвертированием фаз значений регистра, для которых операция возвращает значение 1.

Это определение стоит пояснить. Идея в том, что схеме фазовой логики передается состояние (которое может находиться в суперпозиции), а схема инвертирует относительные фазы всех входных значений, для которых выполнима представляемая логическая операция. Если переданное состояние не находится в суперпозиции, то инвертирование фазы просто приведет к изменению бесполезной глобальной фазы, но с суперпозицией схема кодирует информацию в относительных фазах, к которым затем можно обратиться при помощи примитива усиления комплексной амплитуды.

Выражение «выполнимость» (satisfiability) часто используется для описания входных данных логической операции (или совокупности таких операций, образующих логическую команду), для которых будет получен результат 1. В этой терминологии фазовая логика инвертирует фазы всех значений регистра QPU, для которых логическая операция выполнима.

Вы уже видели пример такой операции, манипулирующей с фазами: это сама операция PHASE! Действие PHASE показано на рис. 5.13. При воздействии на одиночный кубит она просто записывает логическое значение кубита в его фазу (инвертируется фаза только значения $|1\rangle$, то есть когда результат чтения равен 1). Хотя до настоящего момента мы рассматривали PHASE просто как инструмент для поворота относительных фаз кубитов, операция удовлетворяет нашему определению фазовой логики, а следовательно, также может интерпретироваться как логическая операция, основанная на фазе.

Очень важно понимать, чем двоичная логика отличается от фазовой — этот аспект может породить путаницу. Краткая сводка отличий:

Традиционная двоичная логика применяет логические вентили к входным данным для получения результата.

Квантовая амплитудная логика применяет логические вентили к входным данным для получения суперпозиции результатов.

Квантовая фазовая логика инвертирует фазу каждого входного значения, которое дает 1 в результате; также работает, если регистр находится в суперпозиции.

Возможно, смысл фазовой логики будет проще понять на примерах в круговой записи. На рис. 10.1 показано, как версии операций OR, NOR, XOR, AND и NAND в фазовой логике влияют на однородную суперпозицию двухкубитного регистра.

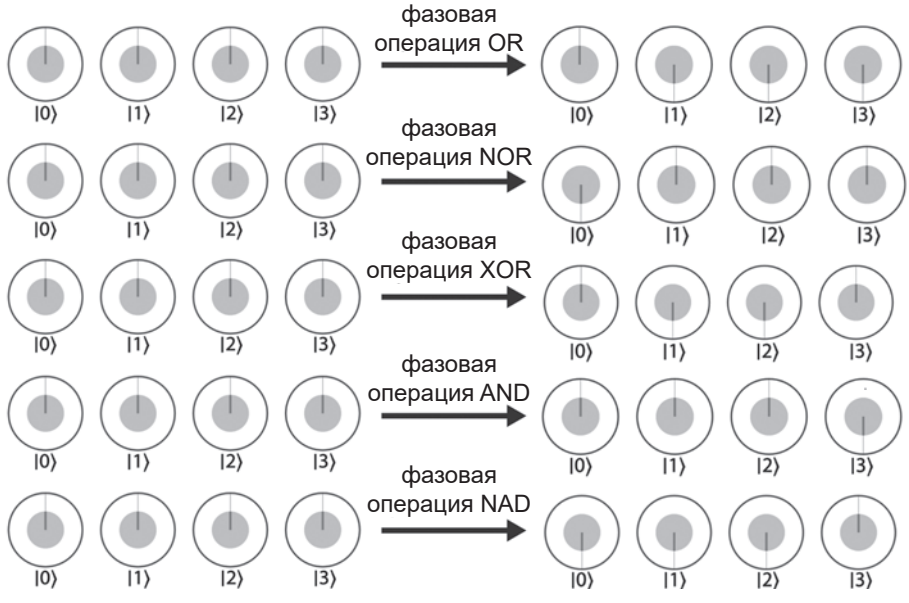


Рис. 10.1. Представления элементарных логических вентилей в фазовой логике для двухкубитного регистра

На рис. 10.1 мы решили представить воздействие этих операций фазовой логики в суперпозициях регистра, но вентили будут работать независимо от того, содержит ли регистр суперпозицию или отдельное значение. Например, традиционная двоичная логическая операция XOR выводит значение 1 для входных данных 10 и 01. Из рис. 10.1 видно, что инвертированы были только фазы $|1\rangle$ и $|2\rangle$.

Фазовая логика принципиально отличается от любой традиционной логики — результаты логических операций теперь скрыты в фазах, недоступных для READ. Но у нее есть одно важное преимущество: инвертируя фазы в суперпозиции, можно пометить *несколько* решений в *одном* регистре! Кроме того, хотя решения, содержащиеся в суперпозиции, обычно недоступны, вы уже знаете, что при использовании фазовой логики в качестве подсхемы инвертирования в усилении комплексной амплитуды *можно* создавать результаты, доступные для чтения READ.

Построение элементарных операций фазовой логики

Итак, теперь вы знаете, чего вы хотите добиться при помощи схем фазовой логики. Как же построить вентили фазовой логики вроде показанных на рис. 10.1 на базе фундаментальных операций QPU?

На рис. 10.2 показаны схемы, реализующие некоторые элементарные операции фазовой логики. Следует учесть, что некоторые из этих операций (как и в главе 5) используют дополнительный служебный кубит. В случае фазовой логики любые служебные кубиты всегда инициализируются[1] в состоянии $|-\rangle$. Важно заметить, что служебный кубит не вступает в состояние запутанности с входным регистром, а следовательно, отпадает необходимость в отмене вычислений для всего вентиля фазовой логики. Дело в том, что служебные кубиты реализуют вентили фазовой логики с использованием приема фазовой коррекции, описанного в главе 3.

Необходимо помнить, что входные значения для этих реализаций фазовой логики кодируются в состояниях регистров QPU (например, $|2\rangle$ или $|5\rangle$), но выходные значения кодируются в относительных фазах. Название «фазовая коррекция» может создать ошибочное впечатление, будто эти реализации также получают значения фазы на входе!

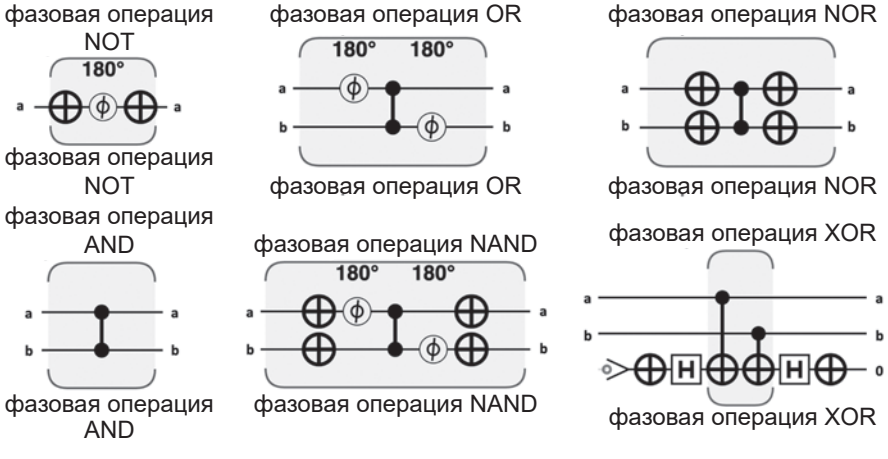


Рис. 10.2. Операции QPU, реализующие фазовую логику

Построение сложных команд фазовой логики

Логика, которую мы хотим исследовать средствами QS, объединяет много разных элементарных логических операций. Как найти схемы QPU для таких полномасштабных составных команд фазовой логики? Мы особо предупреждали о том, что реализации на рис. 10.2 выводят фазы,

[1] Например, операция XOR в фазовой логике на рис. 10.2 использует служебный кубит, который подготавливается в состоянии $|-\rangle$ при помощи операций NOT и HAD.

но требуют входных данных с амплитудными значениями. Таким образом, мы не можем просто запутать эти элементарные операции фазовой логики для построения более сложных команд; их входы и выходы несовместимы.

К счастью, существует один хитрый прием; мы берем полную команду, которая должна быть реализована средствами фазовой логики, и выполняем все, кроме завершающей логической операции с использованием амплитудной квантовой логики того типа, который был показан в главе 5. В результате будут выведены значения из предпоследней операции, закодированные в амплитудах регистра QPU. Результаты передаются реализации последней оставшейся логической операции, построенной на фазовой логике (с использованием одной из схем на рис. 10.2). Вуаля! Мы получаем завершающий вывод всей команды, закодированный в фазах.

Чтобы увидеть этот прием в действии, представьте, что вы хотите выполнить логическую команду (a OR NOT b) AND c (с тремя логическими переменными a, b и c) в фазовой логике. Набор традиционных логических вентилей для этой команды показаны на рис. 10.3.

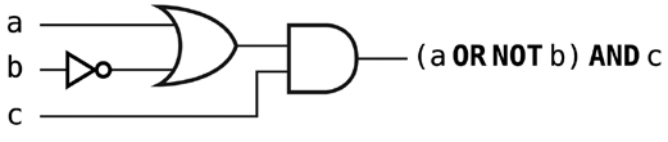


Рис. 10.3. Пример логической команды, выраженной набором традиционных логических вентилей

Для представления этой команды средствами фазовой логики в итоге мы хотим получить регистр QPU в однородной суперпозиции с фазами, инвертированными для всех входных значений, обеспечивающих выполнимость команды.

Наш план заключается в том, чтобы использовать квантовую логику на базе магнитуд для части команды (a OR NOT b) (все операции, кроме последней), а затем использовать схему фазовой логики для объединения этого результата c с операцией AND — в результате мы получим окончательный результат команды в фазах регистра. Схема на рис. 10.4 показывает, как это выражается в операциях QPU[1].

[1] Например, операция XOR в фазовой логике на рис. 10.2 использует служебный кубит, который подготавливается в состоянии $|-\rangle$ при помощи операций NOT и HAD.

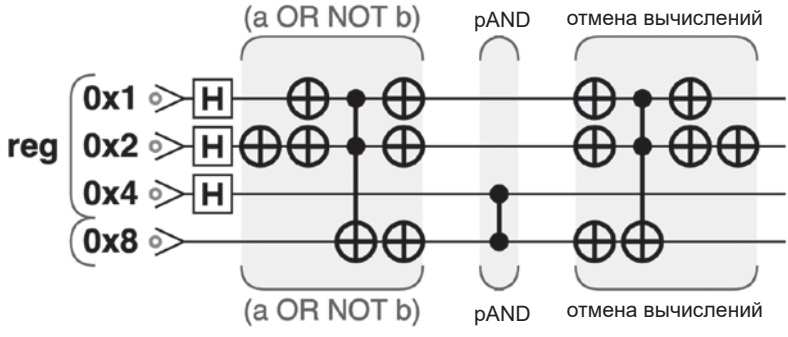


Рис. 10.4. Пример команды, выраженной в фазовой логике

Также обратите внимание на необходимость включения служебного кубита для логической операции с двоичным значением. Результат (a OR NOT b) записывается в служебный кубит, после чего выполняется операция AND в фазовой логике (которую мы обозначили pAND) между этим кубитом и кубитом c. В завершение происходит отмена вычисления этого служебного кубита, чтобы вернуть его в исходное незапутанное состояние $|0\rangle$.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=10-1.

Листинг 10.1. Кодирование в фазовой логике

```
qc.reset(4);
var reg = qint.new(4, 'reg');

qc.write(0);
reg.hadamard();

// (a OR NOT b)
qc.not(2);
bit_or(1,2,8);

// pAND
phase_and(4|8);

// отмена вычислений
inv_bit_or(1,2,8);
qc.not(2);

// Логические определения
```

```
// двоичная логическая операция OR с вспомогательным выводом
function bit_or(q1,q2,out) {
    qc.not(q1| q2);
    qc.cnot(out,q1| q2);
    qc.not(q1| q2| out);
}

// обращенная двоичная логическая операция OR (отмена вычислений)
function inv_bit_or(q1,q2,out) {
    qc.not(q1| q2| out);
    qc.cnot(out,q1| q2);
    qc.not(q1| q2);
}

// Операция AND в фазовой логике (pAND)
function phase_and(qubits) {
    qc.cz(qubits);
}
```

При выполнении приведенного кода схема инвертирует фазы значений $|4\rangle$, $|5\rangle$ и $|7\rangle$, как показано на рис. 10.5.

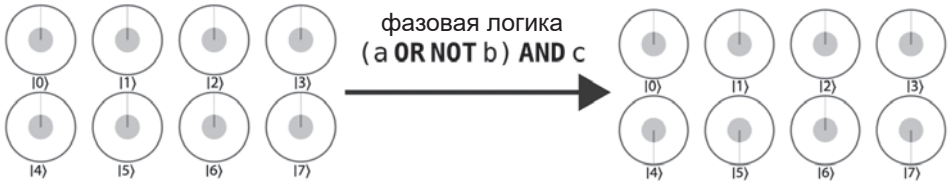


Рис. 10.5. Преобразование состояния

Эти состояния кодируют логические присваивания (`a=0`, `b=0`, `c=1`), (`a=1`, `b=0`, `c=1`) и (`a=1`, `b=1`, `c=1`) соответственно — это единственные логические входные данные, для которых обеспечивается выполнимость исходной логической команды на рис. 10.3.

Решение логических головоломок

С новоприобретенной способностью помечать фазы значений, для которых выполняются логические условия, мы можем воспользоваться AA для решения задач *выполнимости булевых формул*. В этих задачах требуется определить, существуют ли входные значения, для которых выполняется заданное логическое условие — в точности то, что мы научились делать с фазовой логикой! Задачи выполнимости булевых формул составляют

важную категорию задач в компьютерной теории[1] и находят разнообразные практические применения, включая проверку моделей, планирование искусственного интеллекта и верификацию программного обеспечения. Чтобы понять задачу (и способы ее решения), мы посмотрим, как QPU помогают при решении менее выгодного, но намного более интересного применения задачи выполнимости булевых формул: логических головоломок!

О котятах и тиграх

На одном далеком-предалеком острове жила принцесса, которая отчаянно хотела получить котенка на день рождения. Ее отец-король в принципе не возражал, но хотел убедиться в том, что дочь серьезно относится к своему решению, поэтому вместо котенка подарил ей на день рождения головоломку (рис. 10.6[2]).

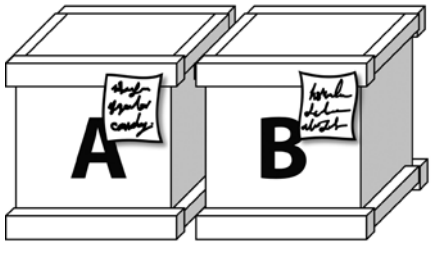


Рис. 10.6. Головоломка на день рождения

Принцесса получила два ящика, и ей было разрешено открыть только один. В каждом ящике мог находиться желанный котенок, но также мог оказаться и свирепый тигр. К счастью, ящики были помечены:

Надпись на ящике A

По крайней мере в одном из ящиков находится котенок.

Надпись на ящике B

В другом ящике находится тигр.

---

[1] В этой схеме присутствуют некоторые избыточные операции (например, отмена операций NOT). Мы оставили их для ясности.

[2] Задача выполнимости булевых формул была первой задачей, для которой была доказана NP-полнота. N-SAT, задача выполнимости булевых формул для логических условий, содержащих конструкции из N литералов, является NP-полной при $N > 2$. В главе 14 приводится более подробная информация об основных классах вычислительной сложности, а также ссылки на материалы с более глубоким изложением темы.

«Все просто!» — подумала принцесса и рассказала отцу решение.

«Но есть один нюанс, — добавил отец, знавший, что такая головоломка была бы слишком простой для нее. — Надписи на ящиках либо обе истинны, либо обе ложны».

«Вот как?» — сказала принцесса. После непродолжительной паузы она побежала в мастерскую и быстро собрала схему. Через минуту она вернулась с устройством, которое показала отцу. Схема имела два входа (рис. 10.7).

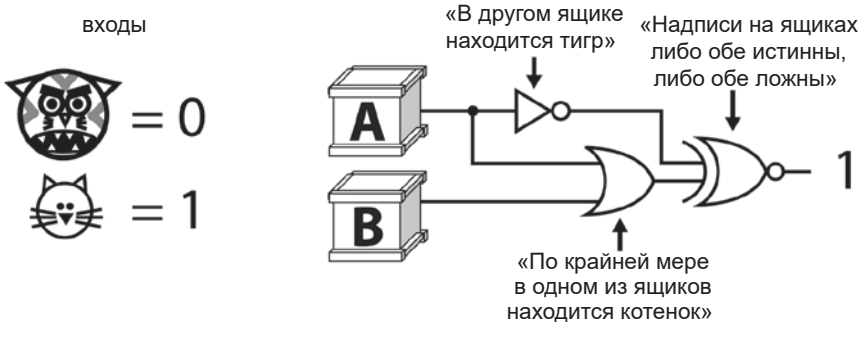


Рис. 10.7. Цифровое решение задачи

«Я настроила все так, что 0 обозначает тигра, а 1 — котенка, — гордо заявила принцесса. — Если задать возможное содержимое каждого ящика, то 1 на выходе появится только в том случае, если эта возможность удовлетворяет всем условиям».

Для каждого из трех условий (надписи на двух ящиках и дополнительное правило, сформулированное отцом) принцесса включила в свою схему логический вентиль:

- для надписи на ящике A она использовала вентиль OR, показывающий, что это ограничение будет выполняться только в том случае, если в ящике A *или* в ящике B находится котенок;
- для надписи на ящике B она использовала вентиль NOT, показывающий, что это ограничение будет выполняться только в том случае, если в ящике A *не* находится котенок;
- наконец, для отцовского дополнения она добавила в конец вентиль XNOR, ограничение которого будет выполняться (давать истинный результат) *только* в том случае, если результаты двух других вентилей одинаковы (истинны или ложны одновременно).

«А теперь остается выполнить эту схему для всех четырех возможных конфигураций котят и тигров и узнать, в каком случае выполняются все ограничения. Тогда я узнаю, какой ящик нужно открыть».

«Гм...» — сказал король.

Принцесса закатила глаза: «Что еще, папа?»

«И... ты можешь запустить свое устройство только один раз».

«Вот как», — задумалась принцесса. Возникла настоящая проблема. Если устройство отработает только один раз, она должна будет угадать, какую входную конфигурацию нужно проверить, и ей вряд ли удастся получить исчерпывающий ответ. Есть 25%-ная вероятность выбрать правильную конфигурацию, но если выбор окажется неверным, а схема выдаст 0, то ей придется просто выбрать случайный ящик и надеяться на лучшее. Нет, традиционная цифровая логика на этот раз не справится с задачей.

К счастью, принцесса недавно прочитала книгу издательства «Питер» о квантовых вычислениях. Горя от нетерпения, она снова бросилась в мастерскую. Через несколько часов принцесса вернулась — она построила новое устройство, на этот раз побольше. Она включила устройство, ввела свои данные в безопасном терминале, запустила программу и издала торжествующий клич. Она подбежала к правильному ящику, открыла его и обняла своего котенка.

Конец истории.

В листинге 10.2 приведена программа для QPU, использованная принцессой (рис. 10.8). На диаграмме также изображен традиционный логический вентиль, который в конечном счете реализуется каждой частью схемы QPU в фазовой логике. Посмотрим, как работает схема!

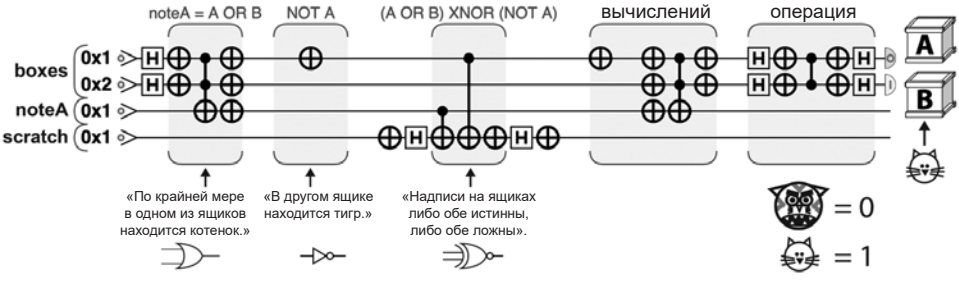


Рис. 10.8. Логическая задача: котенок или тигр?

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=10-2.

Листинг 10.2. Котята и тигры

```
qc.reset(4);
var boxes = qint.new(2, 'boxes')
var noteA = qint.new(1, 'noteA')
var anc = qint.new(1,'anc')
qc.write(0);

// Перевести оба ящика в квантовую суперпозицию "котенок/тигр"
boxes.hadamard();

// Выполнить условие на ящике A в двоичной логике
// noteA = A OR B
qc.not(0x1| 0x2);
qc.cnot(0x4,0x1| 0x2)
qc.not(0x1| 0x2| 0x4);

// Выполнить условие на ящике B в двоичной логике
// NOT A
qc.not(0x1);

// Перевести служебный кубит фазовой логики в состояние | +⟩
anc.not();
anc.hadamard();

// Выполнить последнее условие в фазовой логике
// (A OR B) XNOR (NOT A)
qc.cnot(0x8,0x4);
qc.cnot(0x8,0x1);
qc.not(0x8);

// Вернуть служебный кубит в состояние | 0⟩
anc.hadamard();
anc.not();

// Отмена вычислений для всей двоичной логики
qc.not(0x1);
qc.nop();
qc.not(0x1| 0x2| 0x4);
qc.cnot(0x4,0x1| 0x2)
qc.not(0x1| 0x2);
```

```
// Использовать зеркальную операцию для преобразования
// инвертированной фазы
boxes.Grover();

// Прочитать и интерпретировать результат!
var result = boxes.read();
var catA = result & 1 ? 'kitten' : 'tiger';
var catB = result & 2 ? 'kitten' : 'tiger';
qc.print('Box A contains a ' + catA + '\n');
qc.print('Box B contains a ' + catB + '\n');
```

Всего один запуск программы QPU из листинга 10.2 позволяет решить задачу! Как и в листинге 10.1, применяется амплитудная логика до завершающей операции, в которой применяется фазовая логика. Вывод однозначно показывает (со 100%-ной вероятностью), что если принцесса хочет получить котенка, то должна открыть ящик B. Наши подсхемы фазовой логики занимают место инвертирования в итерации AA, поэтому за ними следует зеркальная подсхема, определенная в листинге 6.1.

Как упоминалось в главе 6, количество применений полной итерации AA зависит от количества задействованных кубитов. К счастью, на этот раз хватит одного! Также нам повезло в том, что у задачи *было* решение (то есть *некоторый* набор входных данных обеспечивал выполнение булевой формулы). Вскоре вы увидите, что произойдет, если решения не существует.

Вспомните, что зеркальная подсхема увеличивает амплитуду значений с инвертированными фазами по отношению к другим. Это не обязательно означает, что фаза должна быть отрицательной для этого конкретного состояния и положительной для остальных — при условии, что она будет инвертирована по отношению к другим. В данном случае один из вариантов будет правильным (с положительной фазой), а остальные неправильны (отрицательная фаза). Но алгоритм все равно работает!

Общий рецепт для решения задач выполнимости булевых формул

Конечно, задача о котенке была разновидностью задачи выполнимости булевых формул. Тот метод, который использовался для решения задачи, хорошо обобщается для других задач выполнимости булевых формул с QPU:

1. Преобразовать булеву формулу из задачи выполнимости к форме, содержащей количество условий, которые должны выполняться одновре-

менно (то есть чтобы команда представляла собой операцию AND с набором независимых условий[1]).

2. Представить каждое отдельное условие с использованием амплитудной логики. Для этого потребуется некоторое число служебных кубитов. Практическое правило: так как большинство кубитов будет задействовано более чем в одном условии, будет полезно создать один служебный кубит на каждое логическое условие.
3. Инициализировать полный регистр QPU (содержащий кубиты, представляющие все входные переменные команды) в равномерной суперпозиции (с применением HAD) и инициализировать все служебные регистры в состоянии $|0\rangle$.
4. Использовать рецепты амплитудной логики, показанные на рис. 5.25, для последовательного построения логических вентилей в каждом условии, с сохранением выходного значения каждого логического условия в служебном кубите.
5. После того как все условия будут реализованы, выполнить операцию AND в фазовой логике между всеми служебными кубитами для объединения разных условий.
6. Выполнить отмену вычислений всех операций амплитудной логики, в результате чего служебные кубиты вернутся в исходные состояния.
7. Запустить зеркальную подсхему для регистра QPU для кодирования входных переменных.
8. Повторить предыдущие действия необходимое количество раз в соответствии с выражением усиления комплексной амплитуды из формулы 6.2.
9. Прочитать итоговый результат (с усилением комплексной амплитуды) из регистра QPU.

В следующих разделах приводятся два примера применения этого рецепта; второй из них демонстрирует, как эта процедура работает в тех случаях, когда булева формула невыполнима (то есть никакая комбинация входных значений не дает результат 1).

---

[1] Эта форма является самой желательной, потому что завершающая операция pAND, объединяющая все команды в фазовой логике, может быть реализована одной операцией CPHASE без обязательного использования дополнительных служебных кубитов. Тем не менее другие формы могут быть реализованы с использованием специально подготовленных служебных кубитов.

Практический пример: задача выполнимости 3-SAT

Рассмотрим следующую задачу 3-SAT:

```
(a OR b) AND (NOT a OR c) AND (NOT b OR NOT c) AND (a OR c)
```

Требуется определить, будет ли получен результат `1` на выходе этой команды для какой-либо комбинации логических входных значений `a`, `b`, `c`. К счастью, команда уже состоит из ряда условий, объединенных операцией AND (как удобно!). Мы воспользуемся регистром QPU из семи кубитов — три для представления переменных `a`, `b` и `c` и четыре служебных кубита для представления каждого из логических условий. Затем мы перейдем к реализации каждого логического условия в амплитудной логике, записав результат всех условий в служебные кубиты. После этого реализуется операция AND в фазовой логике между служебными кубитами, а затем происходит отмена вычислений каждого из условий амплитудной логики. В завершение зеркальная операция применяется к семикубитному регистру QPU, завершающая первую итерацию усиления комплексной амплитуды. Это решение реализуется кодом из листинга 10.3 (рис. 10.9).

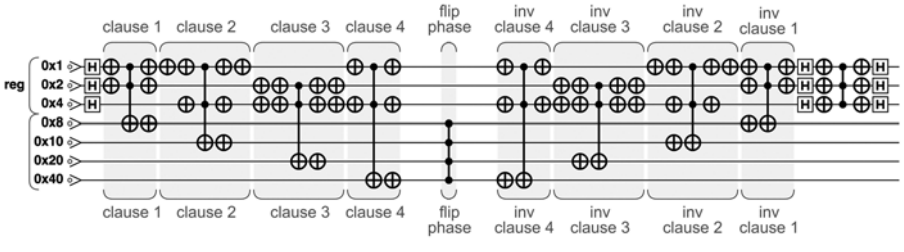


Рис. 10.9. Задача выполнимости 3-SAT

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=10-3.

Листинг 10.3. Задача выполнимости 3-SAT

```
var num_qubits = 3;
var num_ancilla = 4;

qc.reset(num_qubits+num_ancilla);
var reg = qint.new(num_qubits, 'reg');
qc.write(0);

reg.hadamard();
```

```
// Условие 1
bit_or(0x1,0x2,0x8);

// Условие 2
qc.not(0x1);
bit_or(0x1,0x4,0x10);
qc.not(0x1);

// Условие 3
qc.not(0x2| 0x4);
bit_or(0x2,0x4,0x20);
qc.not(0x2| 0x4);

// Условие 4
bit_or(0x1,0x4,0x40);

// Инвертирование фазы
phase_and(0x8| 0x10| 0x20| 0x40);

// Инвертирование условия 4
inv_bit_or(0x1,0x4,0x40);

// Инвертирование условия 3
qc.not(0x2| 0x4);

inv_bit_or(0x2,0x4,0x20);
qc.not(0x2| 0x4);

// Инвертирование условия 2
qc.not(0x1);
inv_bit_or(0x1,0x4,0x10);
qc.not(0x1);

// Инвертирование условия 1
inv_bit_or(0x1,0x2,0x8);
reg.Grover();

////////////// Определения

// Определение операции OR и инвертирование
function bit_or(q1, q2, out)
{
    qc.not(q1| q2);
    qc.cnot(out,q1| q2);
    qc.not(q1| q2| out);
}

function inv_bit_or(q1, q2, out)
```

```
{
    qc.not(q1| q2| out);
    qc.cnot(out,q1| q2);
    qc.not(q1| q2);
}

// Определение фазовой операции AND
function phase_and(qubits)
{
    qc.cz(qubits);
}
```

Используя круговую запись для обозначения изменений состояния трех кубитов, представляющих `a`, `b` и `c`, мы видим результаты, показанные на рис. 10.10.

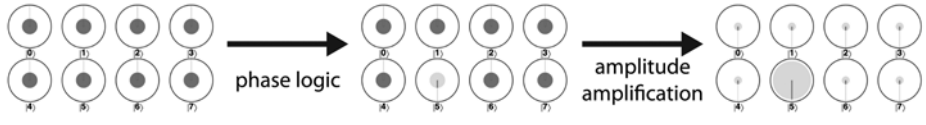


Рис. 10.10. Круговая запись для задачи выполнимости 3-SAT после одной итерации

Из рисунка видно, что булева формула выполнима при значениях `a=1`, `b=0` и `c=1`. В этом примере нам удалось найти набор входных значений, для которых выполнялась булева формула. Что происходит, если решения не существует? К счастью для нас, NP-задачи — такие, как задача выполнимости булевых формул, — обладают одним важным свойством: хотя поиск ответа является вычислительно затратным, проверка правильности ответа относительно дешева. Если булева формула невыполнима, никакая фаза в регистре не будет инвертирована и никакая амплитуда не изменится вследствие зеркальной операции. Поскольку мы начинаем с равной суперпозиции всех значений, завершающая операция READ выдаст одно случайное значение из регистра. Нужно просто проверить, удовлетворяет ли оно логической формуле, и если не удовлетворяет — значит, формула невыполнима. Этот случай будет рассмотрен в следующем разделе.

Практический пример: невыполнимая задача 3-SAT

Возьмем пример невыполнимой задачи 3-SAT:

```
(a OR b) AND (NOT a OR c) AND (NOT b OR NOT c) AND (a OR c) AND b
```

Зная, что эта формула невыполнима, мы ожидаем, что никакое присваивание значений переменных `a`, `b` и `c` не сможет дать результат `1`. Чтобы убедиться в этом, выполним программу для QPU из листинга 10.4 (рис. 10.11); она работает по тому же принципу, что и программа из предыдущего листинга.

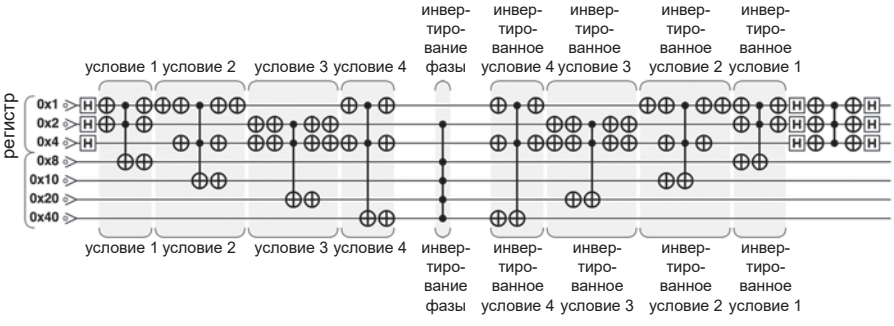


Рис. 10.11. Невыполнимая задача 3-SAT

Пример кода

Этот пример можно выполнить в онлайн по адресу http://oreilly-qc.github.io?p=10-4.

Листинг 10.4. Пример невыполнимой задачи 3-SAT

```
// --- 3-SAT - невыполнимая задача
var num_qubits = 3;
var num_ancilla = 4;

qc.reset(num_qubits+num_ancilla);
var reg = qint.new(num_qubits, 'reg');
qc.write(0);

reg.hadamard();

// Условие 1
bit_or(0x1,0x2,0x8);

// Условие 2
qc.not(0x1);
bit_or(0x1,0x4,0x10);
qc.not(0x1);

// Условие 3
qc.not(0x2| 0x4);
```

```
bit_or(0x2,0x4,0x20);
qc.not(0x2| 0x4);

// Условие 4
bit_or(0x1,0x4,0x40);

// Инвертирование фазы
phase_and(0x2| 0x8| 0x10| 0x20| 0x40);

// Инвертированное условие 4
inv_bit_or(0x1,0x4,0x40);

// Инвертированное условие 3
qc.not(0x2| 0x4);
inv_bit_or(0x2,0x4,0x20);
qc.not(0x2| 0x4);

// Инвертированное условие 2
qc.not(0x1);
inv_bit_or(0x1,0x4,0x10);
qc.not(0x1);

// Инвертированное условие 1
inv_bit_or(0x1,0x2,0x8);

reg.Grover();

////////////////// Определения

// Определение операции OR и инвертирование
function bit_or(q1, q2, out) {
    qc.not(q1| q2);
    qc.cnot(out,q1| q2);
    qc.not(q1| q2| out);
}

function inv_bit_or(q1, q2, out) {
    qc.not(q1| q2| out);
    qc.cnot(out,q1| q2);
    qc.not(q1| q2);
}

// Определение фазовой операции AND
function phase_and(qubits) {
    qc.cz(qubits);
}
```

Пока все хорошо! На рис. 10.12 показано, как три кубита, кодирующие входные значения `a`, `b` и `c`, преобразуются в ходе вычислений. Обратите

внимание: рассматривается только одна итерация AA, потому что (как видно из диаграммы) результат или его отсутствие становится очевидным после всего одной итерации.

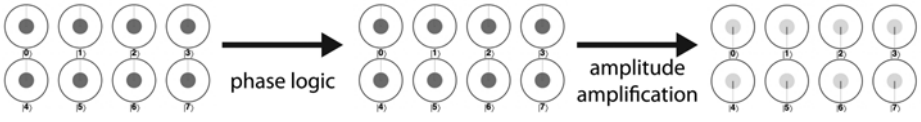


Рис. 10.12. Круговая запись для невыполнимой задачи 3-SAT

Ничего не произошло! Так как ни одно из восьми возможных значений регистра не выполняет булеву формулу, никакая фаза не была инвертирована по отношению к другим; соответственно, зеркальная часть итерации усиления комплексной амплитуды в равной степени повлияла на все значения. Сколько бы итераций AA мы ни выполняли, при чтении этих трех кубитов операцией READ одно из восьми значений будет получено совершенно случайно.

Итак, при чтении можно получить любое значение с одинаковой вероятностью; казалось бы, мы не узнали ничего. Но какие бы из трех значений `a`, `b` и `c` ни были прочитаны, мы можем попытаться ввести их в булеву формулу. Если будет получен результат 0, можно заключить (с высокой вероятностью), что булева формула невыполнима (в противном случае следовало бы ожидать, что будут прочитаны выполняющие значения).

Ускорение традиционных алгоритмов

Одна из самых замечательных особенностей усиления комплексной амплитуды заключается в том, что в некоторых случаях оно может обеспечить квадратичное ускорение не только относительно традиционных алгоритмов, действующих методом «грубой силы», но и относительно лучших традиционных реализаций.

К категории алгоритмов, которые могут ускоряться усилением комплексной амплитуды, относятся алгоритмы с односторонней ошибкой. Это алгоритмы для решения задач с процедурой вывода ответа «да/нет»; если правильный ответ на задачу «нет», то алгоритм всегда выдает ответ «нет», а если правильный ответ «да», то алгоритм выводит «да» с вероятностью $p > 0$. В задачах 3-SAT, которые встречались в предыдущих примерах, алгоритмы выдают ответ «да» (формула выполнима) только в том случае, если ему удастся найти выполняющее присваивание.

Чтобы найти решение с некоторой желательной вероятностью, традиционные алгоритмы должны повторить свою вероятностную процедуру несколько раз (k раз, если алгоритм имеет время выполнения $O(k^n \text{ poly}(n))$). Чтобы ускорить алгоритм для QPU и объединить их с квантовой процедурой, необходимо просто заменить повторяющуюся вероятностную процедуру шагом усиления комплексной амплитуды.

Любой традиционный алгоритм такого вида может быть объединен с усилением амплитуды для его ускорения. Например, наивный способ решения 3-SAT в предыдущих примерах выполняется за время $O(1.414^n \text{poly}(n))$, тогда как лучший традиционный алгоритм работает быстрее, за время $O(1.3(29^n \text{ poly}(n)))$. Объединяя этот традиционный результат с усилением амплитуды, можно достичь времени $O(1.1(53^n \text{ poly}(n)))$!

Существует ряд других алгоритмов, которые могут ускоряться этим методом.

- *Различающиеся элементы* — алгоритмы, которые для заданной функции `f`, работающей с регистром, могут определить, существуют ли в регистре два разных элемента i, j, для которых $f(i)=f(j)$. Эта задача имеет такие практические применения, как поиск треугольников на графах и вычисление произведений матриц.
- *Поиск глобальных минимумов* — алгоритмы, которые для целочисленной функции, работающей с регистром из N элементов, находит индекс i регистра, для которого $f(i)$ имеет наименьшее значение.

В главе 14 приведена сводка источников, в которых можно найти описание многих алгоритмов такого рода.

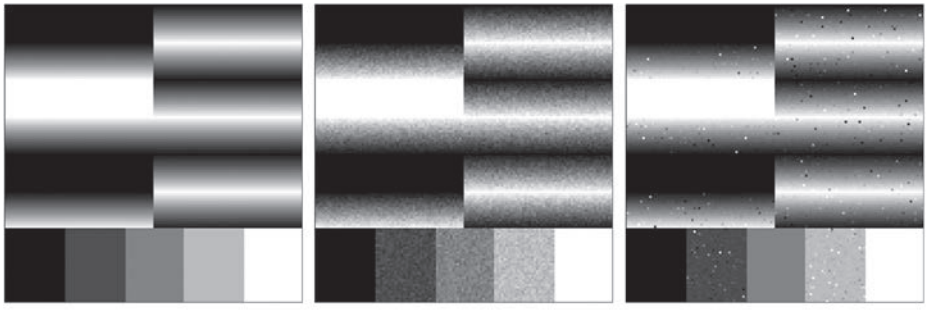
11 Квантовая избыточная выборка

Компьютерная графика всегда находилась на переднем крае компьютерных инноваций, от пиксельных приключенческих игр до фотореалистичных киноэффектов. Технология квантовой обработки изображений (QIP, Quantum Image Processing) использует QPU для расширения возможностей обработки графики. И хотя эта технология еще делает первые шаги, QIP уже предоставляет впечатляющие примеры того, как QPU может повлиять на область компьютерной графики.

Применение QPU в компьютерной графике

В этой главе рассматривается одно конкретное применение QIP — так называемая квантовая избыточная выборка (QSS, Quantum Supersampling). QSS использует QPU для серьезного усовершенствования задачи из области компьютерной графики, которая называется *избыточной выборкой* (supersampling) — рис. 11.1. Это метод традиционной компьютерной графики, при котором изображение, сгенерированное компьютером при высоком разрешении, сводится к изображению более низкого изображения посредством избирательной выборки пикселов. Избыточная выборка является важным шагом в процессе получения выходной графики по изображениям, сгенерированным компьютером.

Метод QSS изначально разрабатывался для *ускорения* избыточной выборки за счет применения QPU, и в этом отношении он потерпел неудачу. Однако численный анализ результатов открыл нечто интересное. Хотя итоговое качество QSS (оцениваемое по уровню погрешности на пиксел) примерно такое же, как у существующих традиционных методов, полученное изображение обладает другими преимуществами.



Эталонное изображение для выборки | Традиционная выборка по методу Монте-Карло. Средняя погрешность | Квантовая супервыборка. Средняя погрешность

Рис. 11.1. Результаты QSS (с идеальным и традиционным случаями для сравнения): изменения в характере шума в изображении, полученном в результате выборки

Как видно из рис. 11.1, средний уровень шума в изображениях, полученных в результате традиционной выборки и QSS, примерно одинаков, но *характер* шума очень сильно различается. В изображениях с традиционной выборкой все пикселы имеют небольшой уровень шума. В изображении с квантовой выборкой некоторые пикселы чрезвычайно зашумлены (черные и белые пятна), тогда как остальные идеальны.

Представьте, что вам выдали изображение и поручили вручную удалить весь видимый шум за 15 минут. Для шума, сгенерированного QSS, это сделать достаточно просто; для изображения с традиционной выборкой почти невозможно.

QSS объединяет несколько примитивов QPU: квантовую арифметику из главы 5, усиление комплексной амплитуды из главы 6 и квантовое преобразование Фурье из главы 7. Чтобы показать, как используются эти примитивы, сначала необходимо разобраться в избыточной выборке.

Традиционная избыточная выборка

Трассировка лучей — метод построения компьютерных изображений, позволяющий добиться повышения качества за счет повышения затрат вычислительных ресурсов. На рис. 11.2 изображено схематическое представление построения изображения на основе сцены, сгенерированной компьютером методом трассировки лучей.

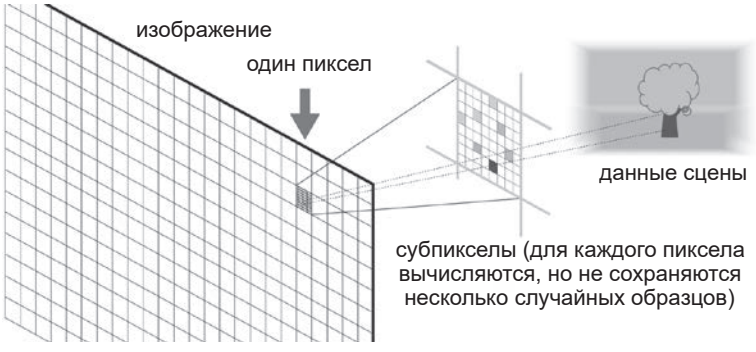


Рис. 11.2. В методе трассировки лучей для каждого пиксела, результирующего изображения, производится несколько выборок

Для каждого пиксела в итоговом изображении математический луч проецируется в трехмерном пространстве из камеры через пиксел по направлению к сцене, сгенерированной компьютером. Луч сталкивается с объектом сцены, и в простейшем варианте (не учитывающем отражения и прозрачность) цвет объекта определяет цвет соответствующего пиксела в изображении.

Хотя при отслеживании всего одного луча на пиксел будет построено правильное изображение, для удаленных подробностей (таких как дерево на рис. 11.2) края и мелкие детали сцены будут потеряны. Кроме того, при перемещении камеры объекта появляются неприятные шумовые эффекты на таких объектах, как листья и ограды.

Чтобы решить задачу без запредельного увеличения размеров изображения, программы трассировки лучей проецируют несколько лучей на пиксел (как правило, сотни или тысячи) с небольшим изменением направления для каждого луча. Сохраняется только *средний* цвет, а остальные детали отбрасываются. Этот процесс называется *избыточной выборкой*, или *выборкой по методу Монте-Карло*. Чем больше образцов будет получено, тем ниже уровень шума в итоговом изображении.

Избыточная выборка — задача, требующая параллельной обработки (с обсчетом результатов многих лучей, взаимодействующих со сценой), но в конечном итоге используются не отдельные результаты, а только их сумма. Похоже, это одна из тех задач, в которых может пригодиться QPU! Полный механизм трассировки лучей QPU потребует намного больше кубитов, чем доступно в данный момент. Но для демонстрации QSS мы можем воспользоваться QPU для построения изображений более высокого разрешения менее сложными методами (не требующими трассировки лучей!), чтобы проанализировать, как QPU влияет на завершающий шаг понижения разрешения.

Практический пример: вычисление изображений с фазовым кодированием

Чтобы применить QPU для избыточной выборки, нам понадобится способ представления изображений в регистрах QPU (пусть и не такой хитроумный, как при полноценной трассировке лучей!)

Существует много разных способов представления изображений, состоящих из пикселов, в регистрах QPU. В конце этой главы будет приведена сводка таких методов из литературы, посвященной квантовой обработке изображений. Тем не менее мы воспользуемся представлением, которое будет называться *фазовым кодированием*, при котором значения пикселов представляются в фазах суперпозиции. Важно, что с этим методом информация нашего изображения будет совместима с методом усиления комплексной амплитуды, представленным в главе 6.

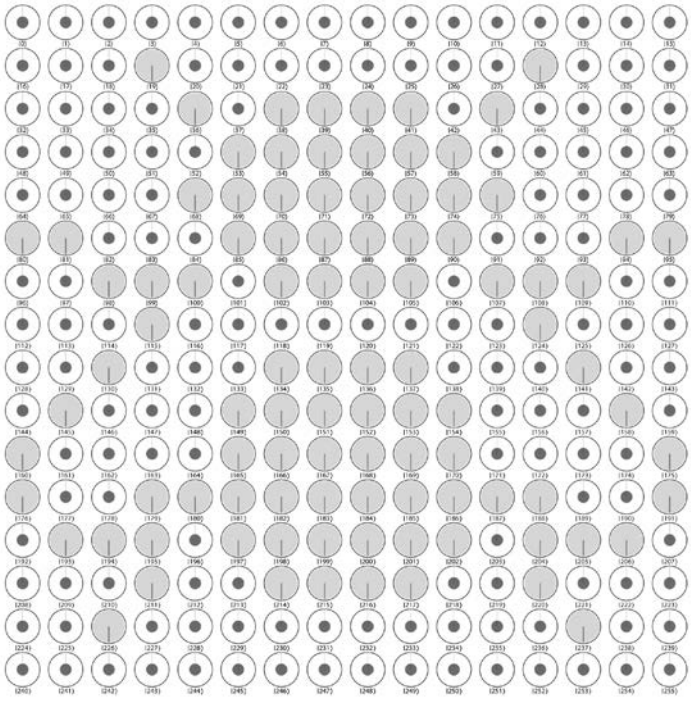


Рис. 11.3. Это не муха[1]

[1] Известная картина «Вероломство образов» сюрреалиста Рене Магритта подчеркивает мысль о том, что изображение трубки — это не трубка. Аналогичным образом закодированное по фазе изображение мухи — это не полное квантовое состояние мухи.

Кодирование изображений в фазе суперпозиции не является чем-то принципиально новым. В конце главы 4 восьмикубитный регистр использовался для фазового кодирования условного изображения мухи[1], как показано на рис. 11.3.

В этой главе вы узнаете, как создаются подобные изображения с фазовым кодированием, а затем мы используем их для демонстрации квантовой избыточной выборки.

Пиксельный шейдер

Пиксельный шейдер — программа (часто выполняемая на GPU), которая получает на входе координаты x и y и генерирует на выходе цвет (черный или белый в данном случае). Для демонстрации QSS мы построим квантовый пиксельный шейдер.

Для начала листинг 11.1 инициализирует два четырехкубитных регистра `qx` и `qy`. Они содержат входные значения x и y для нашего шейдера. Как было показано в предыдущих главах, выполнение операции HAD со всеми кубитами строит равномерную суперпозицию всех возможных значений (рис. 11.4). Перед вами пустой холст, готовый к рисованию.

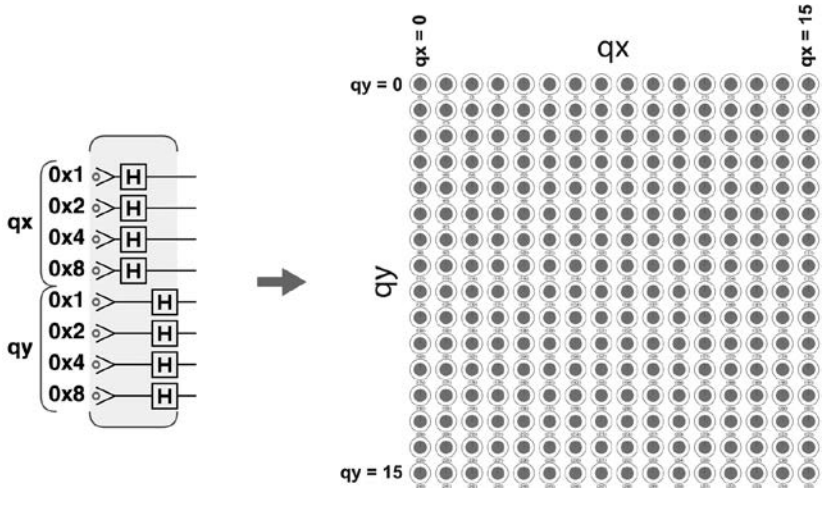


Рис. 11.4. Пустой квантовый холст

[1] Полный код эксперимента с мухой в телепортаторе доступен по адресу http://oreilly-qc.github.io?p=4-2. Чтобы изображение мухи было проще рассмотреть, кроме фазовой логики была применена зеркальная операция из главы 16.

Использование операции PHASE для рисования

Перейдем к рисованию. Поиск конкретных путей для рисования в фазах регистра может быть нетривиальным и сложным делом, но мы можем начать с заполнения правой половины холста, просто выполнив операцию `"if qx >= 8 then инвертировать фазу"`. Задача решается применением операции PHASE(180) к одиночному кубиту, как показано на рис. 11.5:

```
// Подготовка и очистка холста
qc.reset(8);
var qx = qint.new(4, 'qx');
var qy = qint.new(4, 'qy');
qc.write(0);
qx.hadamard();
qy.hadamard();

// Инвертировать, если qx >= 8
qc.phase(180, qx.bits(0x8));
```

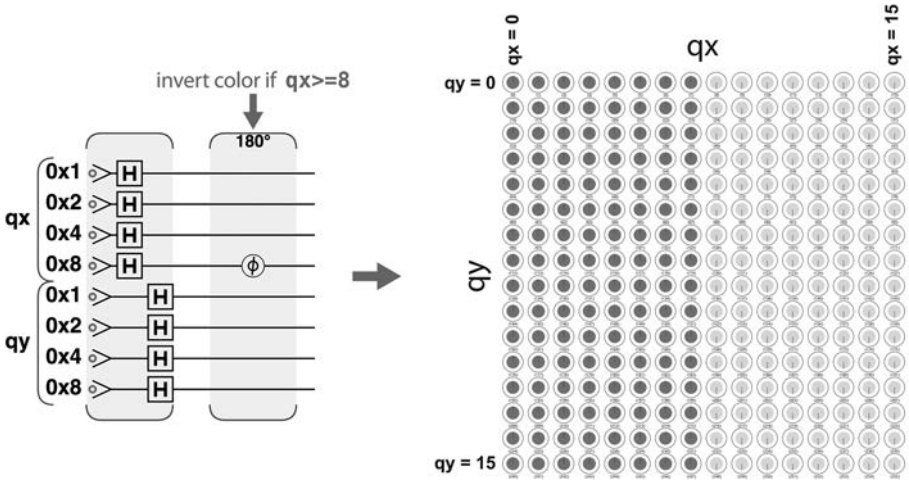


Рис. 11.5. Переключение фазы для половины изображения

В определенном смысле всего одна команда QPU была использована для заполнения 128 пикселов. На GPU для этого пришлось бы выполнить пиксельный шейдер 128 раз. Как вам отлично известно, проблема в том, что при попытке прочитать результат операцией READ вы получите только случайное значение qx и qy.

Слегка расширив логику, мы можем заполнить квадрат 50%-ным регулярным узором[1]. Для этого необходимо инвертировать любые пикселы, у которых оба значения qx и qy больше либо равны 8, а младший кубит qx не равен младшему кубиту qy. Это сделано в листинге 11.1, а результат показан на рис. 11.6.

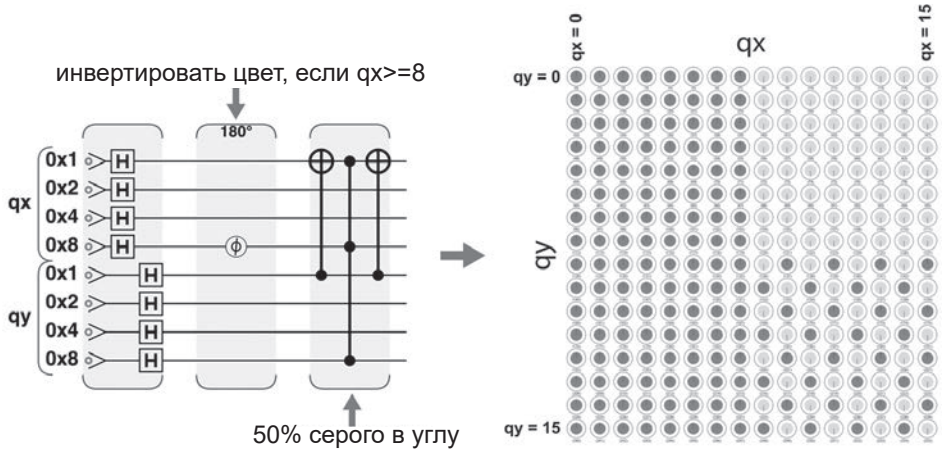


Рис. 11.6. Добавление регулярного узора

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=11-1.

Листинг 11.1. Простейший графический вывод

```
// Очистка холста
qc.reset(8);
var qx = qint.new(4, ‘qx’);
var qy = qint.new(4, 'qy');
qc.write(0);
qx.hadamard();
qy.hadamard();

// Инвертировать, если qx >= 8
qc.phase(180, qx.bits(0x8));

// 50% серого в углу
```

[1] Построение регулярного узора на изображении, обычно с целью аппроксимации недоступных цветов посредством визуального «смешивания» цветов ограниченной палитры.

```
qx.cnot(qy, 0x1);
qc.cphase(180, qy.bits(0x8, qx.bits(0x8| 0x1)));
qx.cnot(qy, 0x1);
```

Если добавить немного арифметики из главы 5, можно создать более интересные узоры, как показано на рис. 11.7.

```
// Очистка холста
qc.reset(8);
var qx = qint.new(4, ‘qx’);
var qy = qint.new(4, 'qy');
qc.write(0);
qx.hadamard();
qy.hadamard();

// Полосы
qx.subtractShifted(qy, 1);
qc.phase(180, qx.bits(0x8));
qx.addShifted(qy, 1);
```

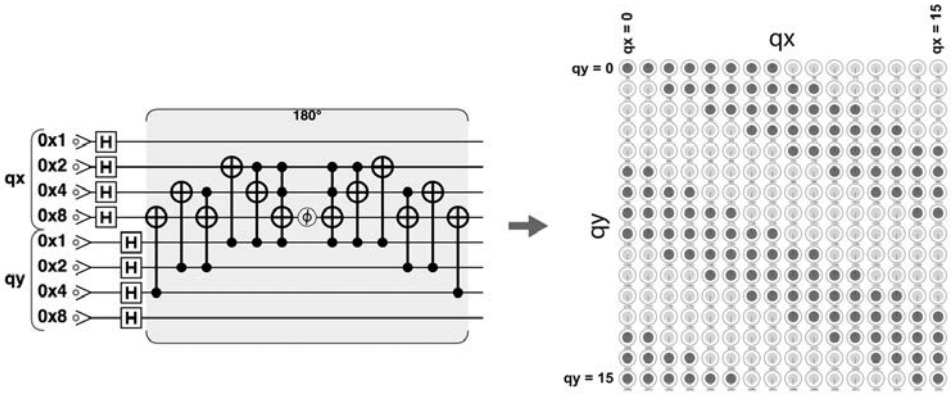


Рис. 11.7. Эксперименты с полосами

Рисование кривых

Для рисования более сложных фигур нам понадобится более сложная математика. В листинге 11.2 продемонстрировано использование функции QPU `addSquared()` из главы 5 для рисования четверти круга с радиусом 13 пикселов. Результат показан на рис. 11.8. На этот раз вычисления должны выполняться с большим 10-кубитным регистром для предотвращения возможного переполнения при возведении в квадрат и сложении `qx` и `qy`. Здесь мы воспользовались приемом, представленным в главе 10 для сохра-

нения значения логической (или математической) операции в фазе состояния с использованием комбинации амплитуды и операций фазовой логики.

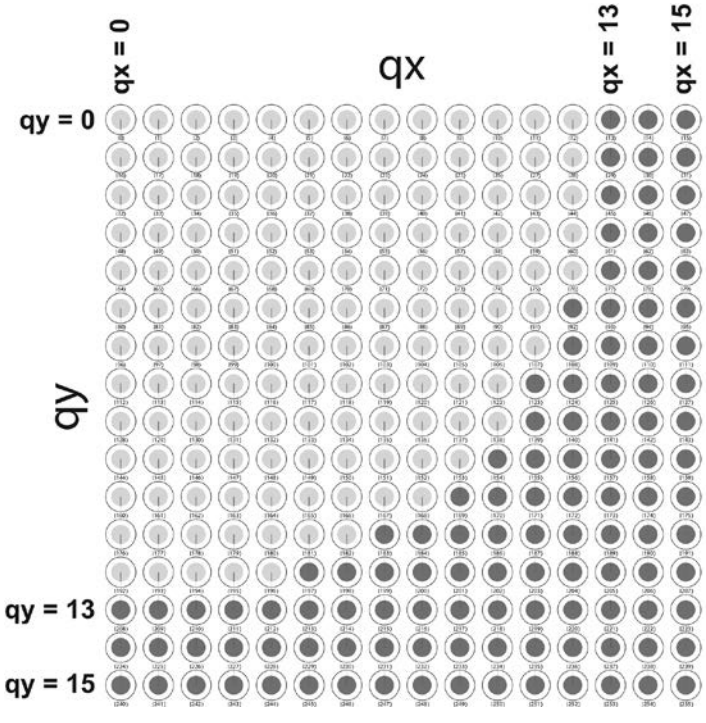


Рис. 11.8. Рисование кривых в гильбертовом пространстве

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=11-2.

Листинг 11.2. Рисование кривых в гильбертовом пространстве

```
var radius = 13;
var acc_bits = 10;

// Очистка холста
qc.reset(18);
var qx = qint.new(4, 'qx');
var qy = qint.new(4, 'qy');
var qacc = qint.new(10, 'qacc');
qc.write(0);
qx.hadamard();
```

```
qy.hadamard();

// Заполнить, если if x^2 + y^2 < r^2
qacc.addSquared(qx);
qacc.addSquared(qy);
qacc.subtract(radius * radius);
qacc.phase(180, 1 << (acc_bits - 1));
qacc.add(radius * radius);
qacc.subtractSquared(qy);
qacc.subtractSquared(qx);
```

Если регистр `qacc` слишком мал, то выполняемые в нем вычисления приведут к переполнению и появлению криволинейных полос. Эффект переполнения будет намеренно использован на рис. 11.11 позднее в этой главе.

Выборка в изображениях с фазовым кодированием

Итак, теперь вы знаете, как представляются изображения в фазах квантовых регистров, и мы можем вернуться к задаче избыточной выборки. Вспомните, что при избыточной выборке существуют различные блоки информации, вычисленные на основании сцены, сгенерированной компьютером (для разных отслеживаемых лучей), которые необходимо объединить для получения одного пиксела в итоговом изображении. Чтобы смоделировать этот процесс, необходимо рассмотреть массив 16 × 16 квантовых состояний, построенный на базе 16 блоков 4 × 4 (рис. 11.9).

Представим, что полное изображение 16 × 16 содержит данные в более высоком разрешении, которые затем требуется сократить до 16 результирующих пикселов.

Так как, по сути, они представляют собой субпикселы 4 × 4, регистры `qx` и `qy`, необходимые для рисования на этих блоках, могут быть сокращены до 2 кубитов каждый. Мы можем использовать регистр переполнения (который мы назовем `qacc`, потому что такие регистры часто называются *аккумуляторами* — accumulators) для выполнения любых необходимых логических операций, которые не помещаются в два кубита. На рис. 11.10 (листинг 11.3) показана схема для создания изображения, показанного на рис. 11.9, блок за блоком.

Обратите внимание: в этой программе переменные `tx` и `ty` — цифровые значения, указывающие, с каким блоком изображения работает шейдер. Чтобы получить абсолютное значение *x* субпиксела, который требуется прори-

совать, мы складываем `qx` и (`tx` × 4), легко вычисляемые благодаря тому, что `tx` и `ty` не являются квантовыми. Такое разбиение изображений на блоки упрощает последующее выполнение избыточной выборки.

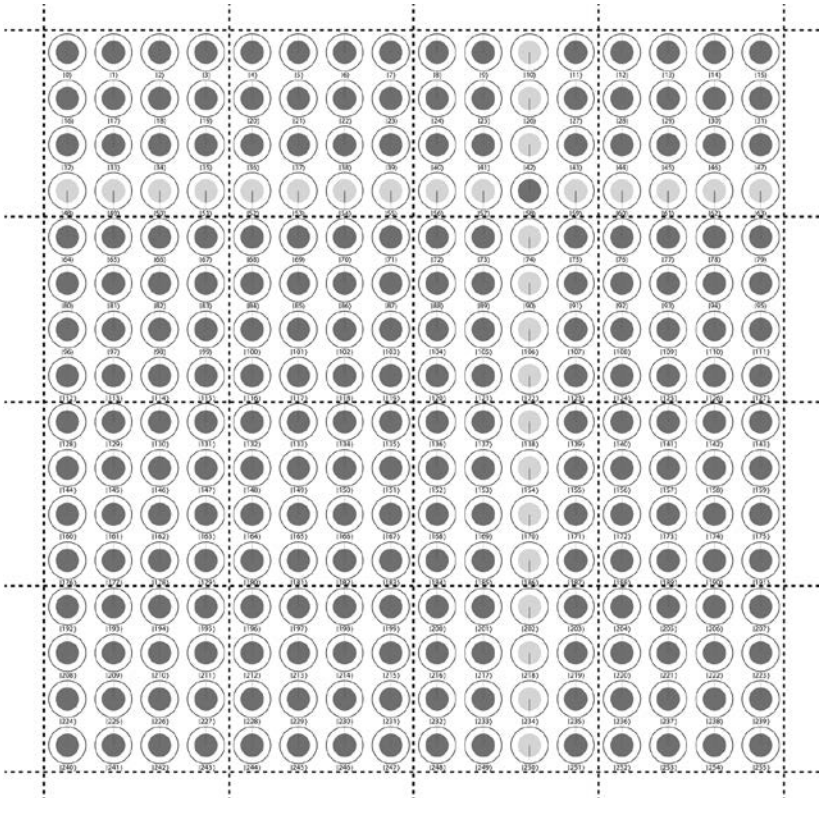


Рис. 11.9. Простое изображение, разделенное на субпикселы

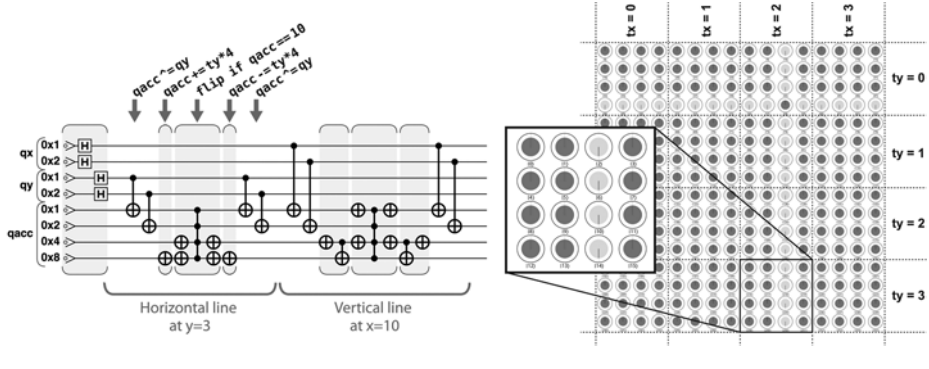


Рис. 11.10. Рисование субпикселов на одном блоке в суперпозиции

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=11-3.

Листинг 11.3. Рисование линий с использованием аккумулятора

```
// Подготовка и очистка холста
qc.reset(8);
var qx = qint.new(2, 'qx');
var qy = qint.new(2, 'qy');
var qacc = qint.new(4, 'qacc');
qc.write(0);
qx.hadamard();
qy.hadamard();

// Выбор блока для рисования
var tx = 2;  // Столбец
var ty = 1;  // Строка

// Горизонтальная линия y=3
qacc.cnot(qy)
qacc.add(ty * 4);
qacc.not(~3);
qacc.cphase(180);
qacc.not(~3);
qacc.subtract(ty * 4);
qacc.cnot(qy);

// Вертикальная линия x=10
qacc.cnot(qx)
qacc.add(tx * 4);
qacc.not(~10);
qacc.cphase(180);
qacc.not(~10);
qacc.subtract(tx * 4);
qacc.cnot(qx);
```

Более интересное изображение

С более сложной программой-шейдером можно построить более интересный тест для квантового алгоритма избыточной выборки. Чтобы протестировать и сравнить методы избыточной выборки, мы используем окружности для получения высокочастотных деталей. Полный исходный код для изображения на рис. 11.11, который также разбивает изображение с фазовым кодированием на блоки, как упоминалось ранее, может быть выполнен по адресу http://oreilly-qc.github.io?p=11-A.

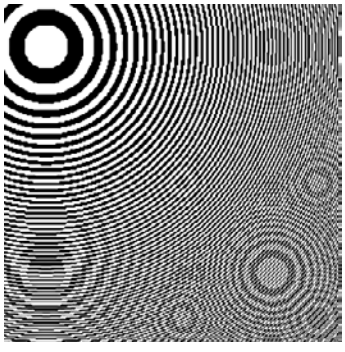


Рис. 11.11. Более сложное изображение с высокочастотными деталями, построенное традиционным методом при размере 256 × 256 пикселов

Может показаться, что для построения этих изображений придется использовать множество обходных путей и специальных трюков. Но происходящее не так уж сильно отличается от тех трюков и обходных путей, которые приходилось применять на ранней стадии традиционной компьютерной графики.

После того как изображение в высоком разрешении с фазовым кодированием будет разбито на блоки, все готово к применению алгоритма QSS. В данном случае полное изображение рисуется на области размером 256 × 256 пикселов. Мы используем 4096 блоков, каждый из которых состоит из 4 × 4 субпикселов, и проведем избыточную выборку всех субпикселов в одном блоке для генерирования одного пиксела, результирующего изображения по 16 субпикселам.

Избыточная выборка

Для каждого блока необходимо оценить количество субпикселов с инвертированной фазой. Для черных и белых субпикселов (представленных инвертированной или неинвертированной фазой) это позволит нам получить значение для каждого результирующего пиксела, характеризующее интенсивность исходных составляющих субпикселов. К счастью, эта задача в точности совпадает с задачей квантовой оценки суммы из раздела «Инвертирование нескольких элементов».

Чтобы воспользоваться квантовой оценкой суммы, мы просто берем квантовую программу, реализующую собственные команды вывода в качестве подсхемы инвертирования, используемой в усилении комплексной амплитуды из главы 6. Объединение его с квантовым преобразованием Фурье из

главы 7 позволяет аппроксимировать общее количество инвертированных субпикселов в каждом блоке. Это подразумевает многократный запуск нашей программы рисования для каждого блока.

Следует заметить, что без QPU преобразование изображения высокого разрешения к более низкому разрешению все равно потребует многократного выполнения процедуры рисования для каждого блока, с рандомизацией значений `qx` и `qy` для каждого образца. Каждый раз мы просто получаем образец «черный» или «белый», последующее суммирование которых позволяет приблизиться к аппроксимируемому изображению.

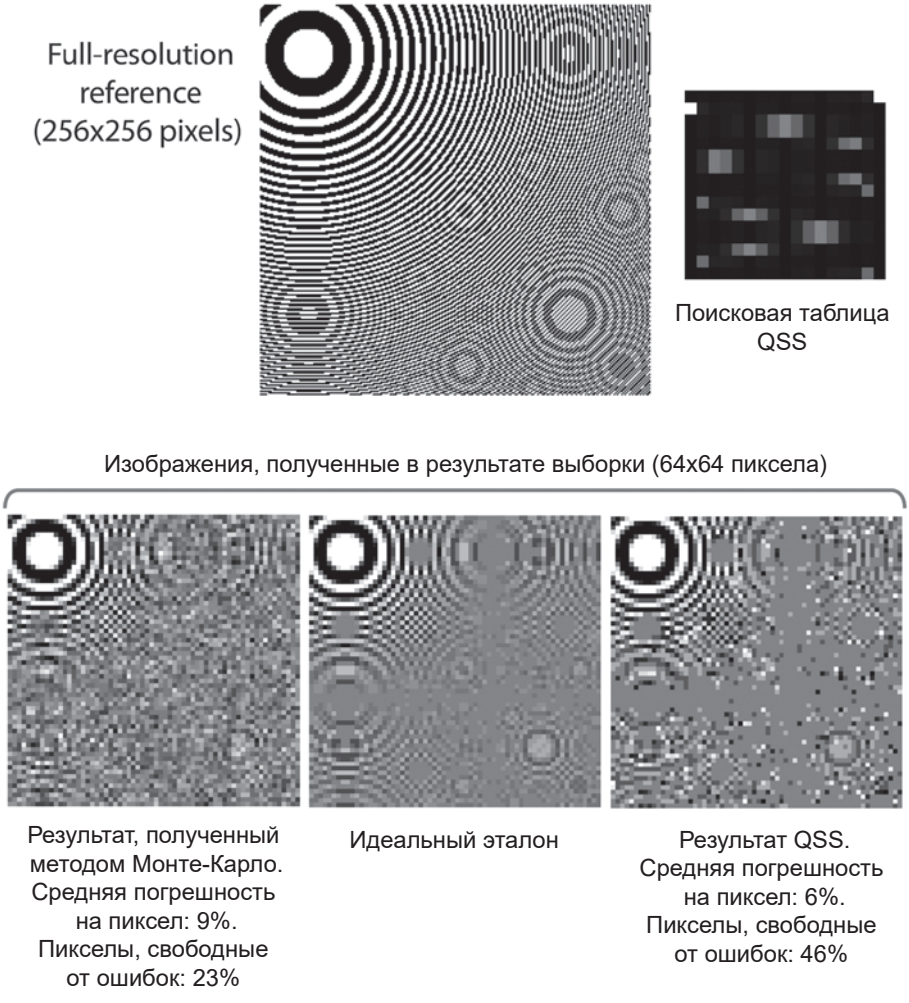


Рис. 11.12. Сравнение квантовой и традиционной избыточной выборки

Пример кода в листинге 11.4 демонстрирует результаты как квантовой, так и традиционной избыточной выборки (обычно выполняемой в традиционном случае методом, который называется выборкой методом Монте-Карло); эти результаты сравниваются на рис. 11.12. Эталон показывает, какой результат будет получен при идеальной выборке, а поисковая таблица QSS — инструмент, который будет подробно рассмотрен на следующих страницах.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=11-4.

Листинг 11.4. Избыточная выборка

```
function do_qss_image()
{
    var sp = {};
    var total_qubits = 2 * res_aa_bits + num_counter_bits
                       + accum_bits;

    // Подготовка квантовых регистров
    qc.reset(total_qubits);
    sp.qx = qint.new(res_aa_bits, 'qx');
    sp.qy = qint.new(res_aa_bits, 'qy');
    sp.counter = qint.new(num_counter_bits, 'counter');
    sp.qacc = qint.new(accum_bits, 'scratch');
    sp.qacc.write(0);

    // Для каждого блока в изображении выполнить функцию qss_tile()
    for (var sp.ty = 0; sp.ty < res_tiles; ++sp.ty) {
        for (var sp.tx = 0; sp.tx < res_tiles; ++sp.tx)
            qss_tile(sp);
    }
}

function qss_tile(sp)
{
    // Подготовка холста для блока
    sp.qx.write(0);
    sp.qy.write(0);
    sp.counter.write(0);
    sp.qx.hadamard();
    sp.qy.hadamard();
    sp.counter.hadamard();
```

```
    // Выполнить пиксельный шейдер несколько раз
    for (var cbit = 0; cbit < num_counter_bits; ++cbit) {
        var iters = 1 << cbit;
        var qxy_bits = sp.qx.bits().or(sp.qy.bits());
        var condition = sp.counter.bits(iters);
        var mask_with_condition = qxy_bits.or(condition);
        for (var i = 0; i < iters; ++i) {
            shader_quantum(sp.qx, sp.qy, sp.tx, sp.ty, sp.qacc,
                           condition, sp.qcolor);
            grover_iteration(qxy_bits, mask_with_condition);
        }
    }
    invQFT(sp.counter);

    // Прочитать и интерпретировать результат
    sp.readVal = sp.counter.read();
    sp.hits = qss_count_to_hits[sp.readVal];
    sp.color = sp.hits / (res_aa * res_aa);
    return sp.color;
}
```

Как упоминалось в начале главы, самые важные преимущества от использования QSS связаны не с количеством операций графического вывода, которые необходимо выполнить, а с различиями в *характере* наблюдаемого шума.

В этом примере при сравнении эквивалентного количества образцов QSS имеет среднюю погрешность на пиксел примерно на 33% ниже, чем у метода Монте-Карло. Что еще интереснее, количество пикселов с нулевой погрешностью (пикселов, у которых результат точно совпадает с идеальным) почти вдвое больше, чем у результата метода Монте-Карло.

QSS и традиционная выборка методом Монте-Карло

В отличие от традиционной выборки методом Монте-Карло, наш шейдер QSS никогда не выводит одиночные значения субпикселов. Вместо этого он использует суперпозицию возможных значений для оценки суммы, которую вы получили бы при вычислении и суммировании их всех. Если вам действительно нужно вычислить значение каждого субпиксела в процессе, традиционные вычислительные методы лучше подойдут для этой цели. Если же вам нужно знать сумму или другую характеристику группы субпикселов, QPU предоставляет интересный альтернативный подход.

Фундаментальное различие между QSS и традиционной избыточной выборкой можно описать следующим образом:

Традиционная избыточная выборка

С увеличением количества образцов результат сходится к точному ответу.

Квантовая избыточная выборка

С увеличением количества образцов вероятность получения точного ответа растет.

Итак, теперь вы примерно представляете возможности QSS. Давайте подробнее разберемся в том, как работает этот метод.

Как работает QSS

Основополагающая идея QSS заключается в применении метода, впервые представленного в разделе «Инвертирование нескольких элементов», в котором объединение итераций усиления комплексной амплитуды с квантовым преобразованием Фурье (QFT) позволяет оценить количество элементов, инвертированных логикой, используемой в подсхеме инвертирования каждой итерации AA.

В случае QSS подсхема инвертирования предоставляется нашей программой графического вывода, которая инвертирует фазу всех «белых» субпикселов.

Попробуем разобраться в том, как AA работает в сочетании с QFT для подсчета инвертированных элементов. Сначала выполняется одна итерация AA *в зависимости от значения регистра-«счетчика»*. Мы называем этот регистр «счетчиком» именно потому, что его значение определяет, сколько итераций AA выполнит наша схема. Если теперь использовать операции HAD для подготовки регистра-счетчика в суперпозиции, мы будем выполнять *суперпозицию* разного количества итераций AA. Как говорилось в разделе «Инвертирование нескольких элементов», вероятности чтения операцией READ нескольких инвертированных значений в регистре зависит от количества выполняемых итераций AA. В предыдущем обсуждении говорилось, что колебания вводятся в зависимости от количества инвертированных значений. Из-за этого при выполнении суперпозиции разного количества итераций AA вводятся периодические колебания по комплексным амплитудам регистра QPU с частотой, зависящей от количества инвертированных значений.

Что касается чтения частот, закодированных в регистрах QPU, мы знаем, что для этого применяется квантовое преобразование Фурье (QFT): при помощи QFT можно определить эту частоту и, соответственно, количество инвертируемых значений (то есть количество обработанных субпикселов). Зная количество субпикселов, подвергшихся супервыборке для одного пиксела в результирующем изображении в низком разрешении, мы можем использовать эту величину для определения яркости этого пиксела.

Хотя понять это утверждение в тексте непросто, с анализом кода в листинге 11.4 и полученными визуализациями в круговой записи действия, выполняемые QSS, станут более очевидными.

Чем больше кубитов будет использоваться для счетчиков, тем лучше будет качество выборки, но за счет того, что код графического вывода придется выполнять большее количество раз. Это соотношение истинно как для квантового, так и для традиционного подхода, как показано на рис. 11.13.

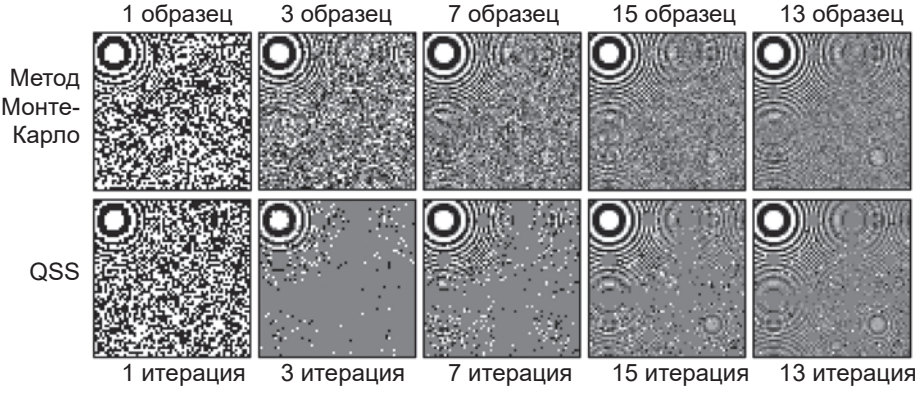


Рис. 11.13. Увеличение количества итераций

Поисковая таблица QSS

Когда мы запускаем алгоритм QSS и в завершение читаем регистр QPU операцией READ, полученное число будет связано с количеством белых пикселов в заданном блоке, но не будет точно равно ему.

Поисковая таблица QSS — инструмент для определения количества субпикселов в блоке, подразумеваемого заданным значением READ. Поисковая таблица, необходимая для конкретного изображения, не зависит от детализации изображения (а точнее, от детализации используемого квантового пиксельного шейдера). Можно сгенерировать и повторно использо-

вать одну поисковую таблицу QSS для любых приложений QSS с заданными размерами блоков и регистров-счетчиков.

Например, на рис. 11.14 изображена поисковая таблица для приложений QSS с размером блока 4 × 4 и регистром-счетчиком, состоящим из 4 кубитов.

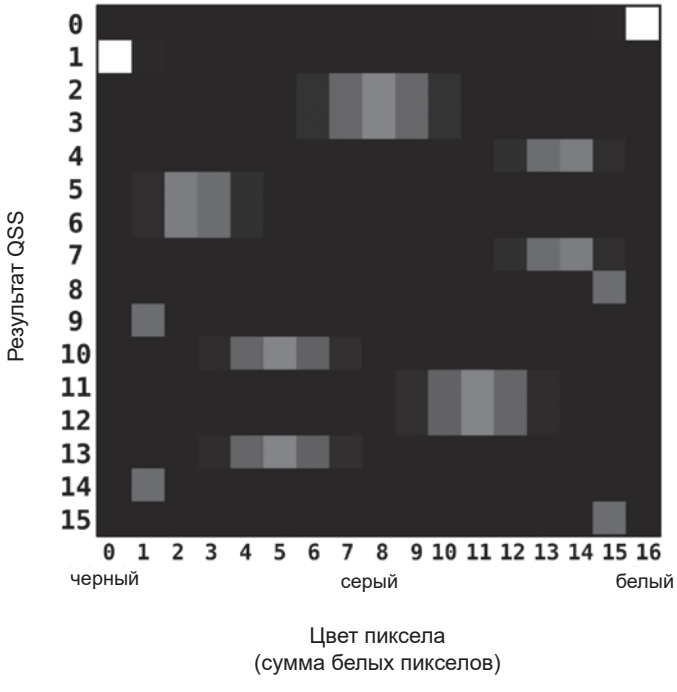


Рис. 11.14. Поисковая таблица QSS для преобразования результатов QSS в яркость пиксела

В строках (ось y) поисковой таблицы перечисляются возможные результаты, которые могут быть прочитаны из выходного регистра QPU при применении QSS. В столбцах (ось x) перечисляются различные возможные количества субпикселов в блоке, которые могут привести к таким значениям, прочитанным операцией READ. Оттенки серого в таблице графически представляют вероятности, связанные с разными возможными значениями READ (более светлые цвета обозначают более высокие вероятности). Рассмотрим пример использования поисковой таблицы. Допустим, был прочитан результат QSS, равный 10. Находим эту строку в поисковой таблице QSS и видим, что результат с наибольшей вероятностью указывает на то, что в супервыборке было задействовано пять белых пикселов. Однако существует ненулевая вероятность того, что субпикселов было четыре или шесть (и еще

меньшая вероятность того, что их было три или семь). При этом появляется некоторая погрешность, так как мы не всегда можем однозначно определить количество выбранных субпикселов по результату READ.

Такая поисковая таблица используется алгоритмом QSS для определения окончательного результата. Откуда взять поисковую таблицу для конкретной задачи QSS? Код в листинге 11.5 показывает, как вычисляется поисковая таблица QSS. Следует заметить, что этот код не полагается на конкретный квантовый пиксельный шейдер (то есть конкретное изображение). Так как поисковая таблица только связывает значение READ с заданным количеством белых субпикселов на блок (независимо от их конкретного местоположения), она может быть сгенерирована без информации о реальном изображении, к которому применяется избыточная выборка.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=11-5.

Листинг 11.5. Построение поисковой таблицы QSS

```
function create_table_column(color, qxy, qcount)
{
    var true_count = color;

    // Перевести в суперпозицию
    qc.write(0);
    qcount.hadamard();
    qxy.hadamard();

    for (var i = 0; i < num_counter_bits; ++i)
    {
        var reps = 1 << i;
        var condition = qcount.bits(reps);
        var mask_with_condition = qxy.bits().or(condition);
        for (var j = 0; j < reps; ++j)
        {
            flip_n_terms(qxy, true_count, condition);
            grover_iteration(qxy.bits(), mask_with_condition);
        }
    }
    invQFT(qcount);

    // Построить поисковую таблицу
    for (var i = 0; i < (1 << num_counter_bits); ++i)
        qss_lookup_table[color][i] = qcount.peekProbability(i);
}
```

Поисковая таблица QSS много говорит о том, как работает QSS для заданных размеров регистров-счетчиков и блоков, и становится важным характерным признаком алгоритма. На рис. 11.15 представлены примеры того, как изменяется поисковая таблица (для фиксированного размера регистра-счетчика из 4 кубитов) с увеличением размера блока (а следовательно, количества субпикселов), используемого в алгоритме QSS.

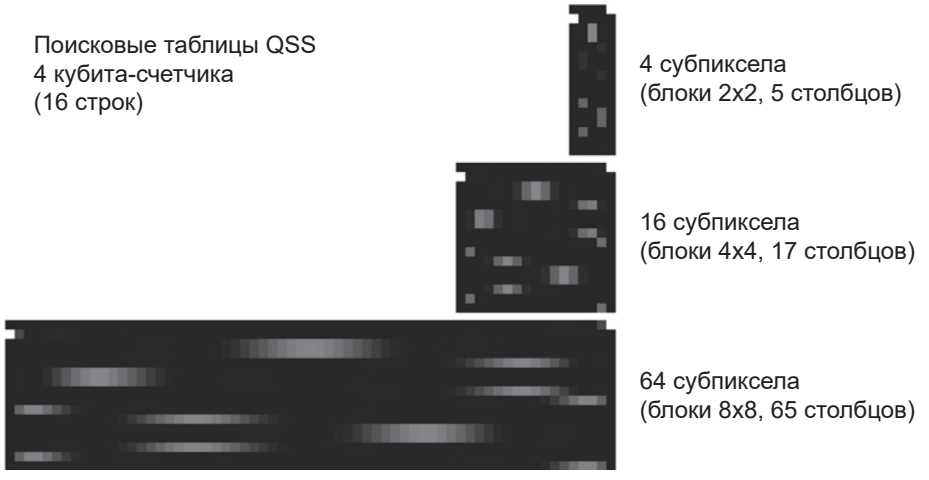


Рис. 11.15. С увеличением количества субпикселов добавляются новые столбцы

Аналогичным образом на рис. 11.16 показано, как поисковая таблица изменяется с увеличением количества используемых кубитов-счетчиков (для заданного размера блока).

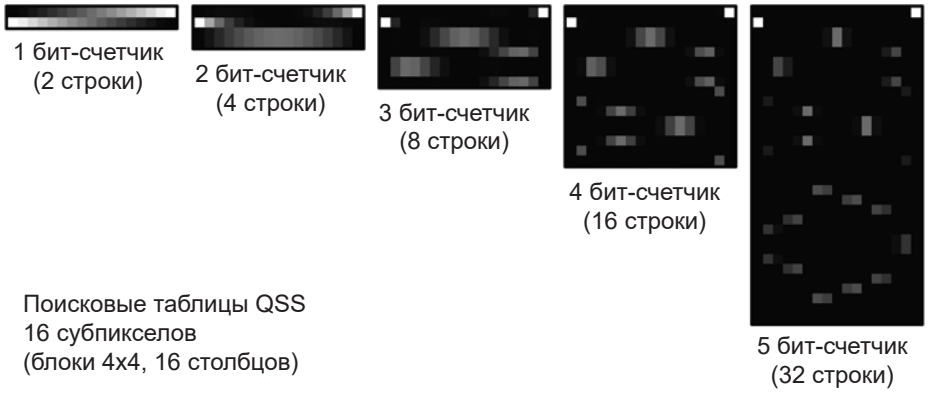


Рис. 11.16. При увеличении размера счетчика добавляются новые строки

Карты достоверности

Кроме интерпретации значений, прочитанных операцией READ, поисковая таблица QSS также может использоваться для оценки того, насколько вы уверены в итоговой яркости пиксела. По расположению READ-значения QSS в строке таблицы можно оценить вероятность того, что полученное значение было правильным. Например, в случае поисковой таблицы на рис. 11.14 можно быть уверенным в прочитанных значениях 0 или 1, но со значениями 2, 3 или 4 уверенность будет намного ниже. По полученным результатам можно построить «карту достоверности», обозначающую вероятное местоположение ошибок в изображениях, определенным по результатам QSS (рис. 11.17).

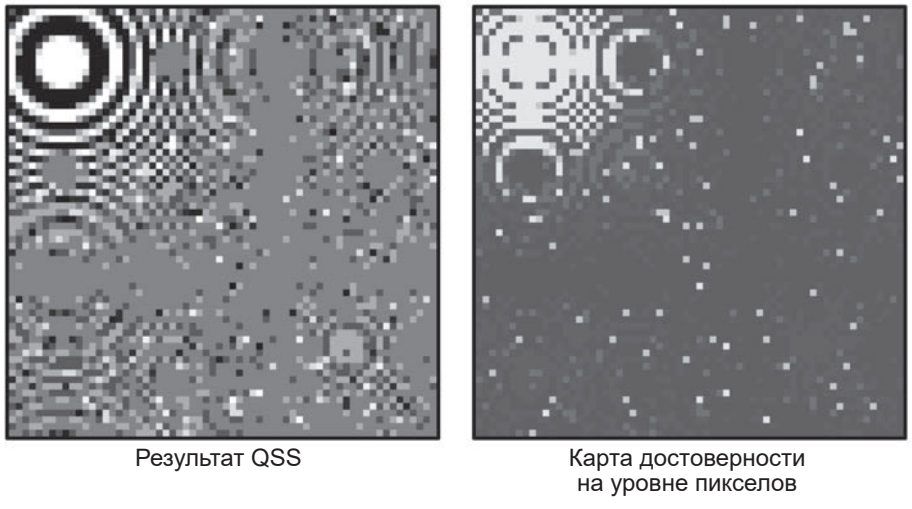


Рис. 11.17. Результат QSS и связанная с ним карта достоверности, сгенерированная на основании поисковой таблицы QSS, — на этой карте более яркие пикселы обозначают области с большей вероятностью правильного результата

Добавление цветов

Все изображения, которые были построены с использованием QSS в этой главе, были монохромными — инвертированные фазы в регистрах QPU использовались для представления черных и белых пикселов.

И хотя мы все любим видеоигры в стиле ретро, можно ли добавить к изображению немного цвета? Фазы и комплексные амплитуды регистра QPU

можно просто использовать для кодирования более широкого диапазона цветовых значений наших пикселов, но тогда метод квантовой оценки суммы, который используется QSS, работать не будет.

Тем не менее мы можем позаимствовать технологию *битовых слоев* (bitplanes), которая использовалась некоторыми ранними графическими картами. При таком решении квантовый пиксельный шейдер используется для построения отдельных монохромных изображений, каждое из которых представляет один бит изображения. Например, допустим, вы хотите запутать с каждым пикселом изображения три цвета (красный, зеленый и синий). Тогда наш пиксельный шейдер фактически может сгенерировать три разных монохромных изображения, каждое из которых представляет вклад одного из трех цветовых каналов. Эти три изображения могут пройти избыточную выборку по отдельности, после чего будут объединены в итоговое цветное изображение.

Такая схема позволяет использовать только восемь цветов (включая черный и белый); избыточная выборка позволяет смешивать *субпикселы* с фактическим созданием изображений с 12 битами на пиксел (рис. 11.18). Полный код доступен по адресу http://oreilly-qc.github.io?p=11-6.

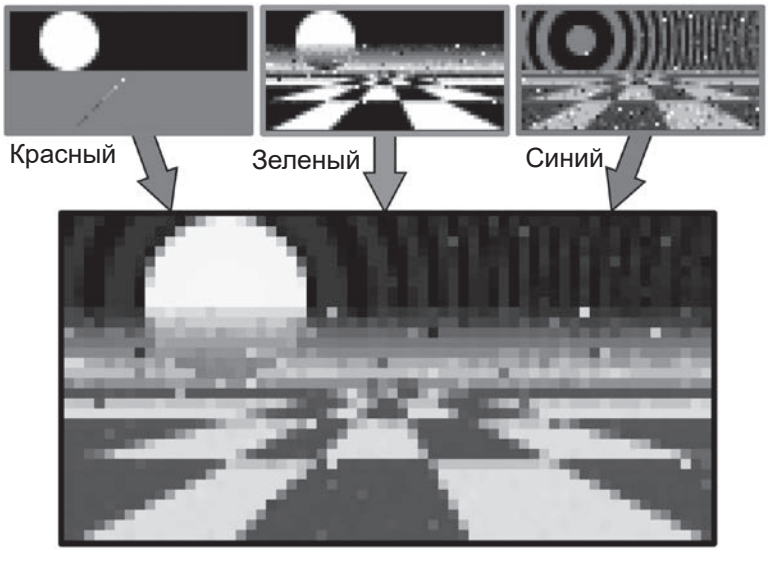


Рис. 11.18. Объединение цветовых слоев, полученных при избыточной выборке

Итоги

В этой главе мы показали, как исследование новых комбинаций примитивов QPU с некоторыми знаниями предметной области могут привести к новым практическим применениям QPU. Способность перераспределения шума выборки также наглядно показывает, что иногда преимущества QPU выходят за рамки простого ускорения.

Также стоит заметить, что способность квантовой избыточной выборки производить перераспределение шума может оказаться актуальной за пределами области компьютерной графики. Многие другие применения также интенсивно используют выборку методом Монте-Карло в таких областях, как искусственный интеллект, вычислительная гидродинамика и даже финансы.

Чтобы познакомить читателя с квантовой избыточной выборкой, мы воспользовались методом представления изображений в регистрах QPU методом фазового кодирования. Стоит заметить, что исследователи в области квантовой обработки изображений также предлагали много других представлений. К их числу относится так называемое представление кубитной решетки[1], гибкое представление квантовых изображений (FRQI[2], Flexible Representation of Quantum Images), новое расширенное квантовое представление (NEQR[3], Novel Enhanced Quantum Representation) и обобщенное квантовое представление изображений (GQIR[4], Generalized Quantum Image Representation). Эти представления[5] использовались для исследования других практических применений, включая поиск по шаблону[6], распознавание границ[7], классификацию изображений[8] и перевод изображений[9] — и это далеко не полный список[10].

---

[1] Venegas-Andraca et al, 2003.

[2] Le et al, 2011.

[3] Zhang et al, 2013.

[4] Jiang et al, 2015.

[5] Более подробный обзор этих квантовых представлений изображений приведен у Yan et al, 2015.

[6] Curtis et al, 2004.

[7] Yuan et al, 2013.

[8] Ostaszewski et al, 2015.

[9] Wang et al, 2015.

[10] За более подробными обзорами практических применений QIP обращайтесь к Cai et al, 2018 и Yan et al, 2017.

12 Алгоритм Шора

Если до этой книги вы слышали о практическом применении квантовых вычислений, то с большой вероятностью это был алгоритм факторизации (разложения на простые множители) Шора.

Когда-то считалось, что квантовые вычисления представляют в основном академический, а не практический интерес. Но в 1994 году Питер Шор открыл, что достаточно мощный квантовый компьютер может находить разложение числа на простые множители экспоненциально быстрее, чем любая традиционная машина. В этой главе будет рассмотрена одна конкретная реализация алгоритма Шора для QPU.

Возможность быстрого разложения больших чисел на простые множители представляет интерес не только для математиков: она позволит взломать криптографическую систему с открытым ключом RSA (Rivest–Shamir–Adleman). Например, вы пользуетесь RSA каждый раз, когда открываете сеанс `ssh`. Работа криптосистемы с открытым ключом — такой как RSA — основана на существовании общедоступного открытого ключа, который может использоваться любым желающим для *шифрования* информации. Но после того как информация будет зашифрована, *расшифровать* ее можно будет только при наличии секретного, закрытого ключа. Криптосистемы с открытым ключом часто сравнивают с электронной разновидностью почтового ящика. Представьте закрытый ящик с прорезью, в который кто угодно может бросить (но не достать!) письмо. В ящике есть дверца, ключ к которой имеется только у владельца. Оказывается, задача нахождения простых множителей большого числа N хорошо работает как часть криптосистемы с открытым ключом. Гарантия того, что посторонний сможет только использовать открытый ключ для шифрования, но не для расшифровки информации, основана на предположении о том, что поиск простых множителей N является задачей, неприемлемой с вычислительной точки зрения.

Полное объяснение RSA выходит за рамки книги, но ключевой момент заключается в том, что если алгоритм Шора предоставляет возможность нахождения простых множителей большого числа *N*, то это может иметь последствия для одного из основополагающих компонентов современного интернета.

Кроме криптографических последствий есть и другие веские причины для изучения алгоритма Шора, поскольку это самый известный пример алгоритмов для решения задач из категории *задач о скрытых подгруппах*. В задачах такого рода требуется определить периодичность заданной периодической функции, причем эта периодичность может быть достаточно сложной. К категории задач о скрытых подгруппах относится целый ряд задач дискретной математики: нахождение периода, нахождение порядка (трудная задача, лежащая в основе разложения на простые множители), нахождение дискретных логарифмов и т. д. Процедура, аналогичная описанной в этой главе, также может предоставить решения для некоторых из этих задач[1].

Алгоритм Шора также предоставляет еще один практический пример использования примитивов QPU. В главе 7 было показано, что квантовое преобразование Фурье (QFT) идеально подходит для исследования периодических сигналов, и в алгоритме Шора оно интенсивно используется.

Особенно поучительный аспект алгоритма Шора заключается в том, что он в конечном итоге также использует традиционную программу для получения искомых простых множителей на основании периодичности, полученной при помощи QFT. Алгоритм так эффективно работает именно потому, что он принимает роль QPU как сопроцессора и применяет квантовые идеи только в тех частях задачи, для которых они хорошо подходят.

Изучим поглубже общую идею и код, лежащий в основе алгоритма.

Практический пример: алгоритм Шора на QPU

Продолжим прагматический подход, выбранный для этой книги: пример кода в листинге 12.1 демонстрирует алгоритм Шора в деле с использованием встроенных функций из QCEngine.

---

[1] Следует учитывать, что не все задачи этого класса имеют решения в описанной форме. Например, задача изоморфизма графов (которая проверяет эквивалентность графов при изменении пометки вершин) также принадлежит к классу задач о скрытых подгруппах, но в настоящее время неизвестно, существует ли для этой задачи эффективный алгоритм для QPU.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=12-1.

Листинг 12.1. Разложение на простые множители на QPU

```
function shor_sample() {
    var N = 35;             // Число, раскладываемое на множители
    var precision_bits = 4; // См. описание в тексте
    var coprime = 2;        // Должно быть равно 2 в этой реализации

    var result = Shor(N, precision_bits, coprime);
}

function Shor(N, precision_bits, coprime) {
    // Квантовая часть
    var repeat_period = ShorQPU(N, precision_bits, coprime);
    var factors = ShorLogic(N, repeat_period, coprime);
    // Традиционная часть
    return factors;
}

function ShorLogic(N, repeat_period, coprime) {
    // Найти множители для заданного периода повторения
    var ar2 = Math.pow(coprime, repeat_period / 2.0);
    var factor1 = gcd(N, ar2 - 1);
    var factor2 = gcd(N, ar2 + 1);
    return [factor1, factor2];
}
```

Как упоминалось ранее, QPU здесь выполняет лишь часть работы. Функция `Shor()` вызывает две другие функции. Первая, `ShorQPU()`, использует QPU (или его модель) для поиска периодичности функции, тогда как остальная работа `ShorLogic()` выполняется традиционным кодом, выполняемым на обычном процессоре.

Обе функции более подробно рассматриваются в следующем разделе.

Реализация `ShorLogic()`, которую мы используем в своем примере, предназначена только для учебных целей. Она проще объясняется, но плохо работает с очень большими числами. В области полномасштабных алгоритмов Шора сейчас ведутся интенсивные исследования.

В оставшейся части этой главы рассматривается алгоритм Шора, представленный в стандартном и доступном виде. Однако следует учитывать, что представленная реализация не является самым эффективным вопло-

щением исходной идеи Шора, а существующие практические реализации с большой вероятностью будут зависеть от оборудования QPU.

Что делает алгоритм Шора

Начнем с функции `ShorQPU()`. Мы примем без доказательства один полезный факт из теории чисел[1] — а именно то, что задача нахождения простых множителей p и q числа $N = pq$ может быть решена в том случае, если нам удастся решить не связанную на первый взгляд задачу нахождения периода повторения функции $a^x \bmod(N)$ с изменением целочисленной переменной x. Здесь N — число, которое требуется разложить на простые множители; a — взаимно-простое число к N (переменная `coprime`). В качестве `coprime` может быть выбрано любое простое число на ваше усмотрение.

Если идея поиска периода функции $a^x \bmod(N)$ кажется невразумительной, не бойтесь. Это означает лишь то, что при изменении значения x последовательность чисел, возвращаемых функцией $a^x \bmod(N)$, будет повторяться. Период повторения — всего лишь количество значений x между повторениями, как показано на рис. 12.1.

Для простоты в качестве `coprime` было выбрано число 2. Кроме того что это наименьшее простое число, у этого варианта есть другое преимущество: наша реализация a^x для QPU может быть реализована простым сдвигом битов. Этот вариант хорошо подходит для ситуаций, рассмотренных в этой главе, но в других случаях он неуместен.

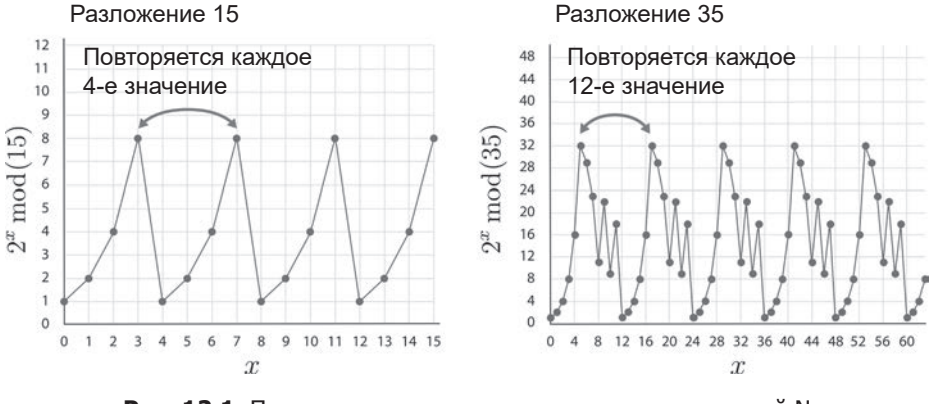


Рис. 12.1. Периоды повторения для двух разных значений N

[1] Более подробные описания приведены в главе 14.

Если вам известен период повторения p, то один из простых множителей N, *возможно*, определяется формулой $gcd(N, a^{p/2} + 1)$, а другой может определяться формулой $gcd(N, a^{p/2} - 1)$. И снова эти утверждения приводятся без доказательства, но они следуют из теории чисел. Здесь *gcd* — функция, возвращающая наибольший общий делитель двух ее аргументов. Хорошо известный *алгоритм Евклида* может быстро вычислить *gcd* на традиционном процессоре (реализация *gcd* доступна по адресу http://oreilly-qc.github.io?p=12-1).

Возможно, эти два выражения с *gcd* позволят найти простые множители, но это не гарантировано. Успех зависит от выбора значения `coprime` a. Как упоминалось ранее, мы выбрали 2 в качестве `coprime` для демонстрационных целей, и существуют числа, которые эта реализация разложить не сможет — например, 171 или 297.

Для чего вообще нужен QPU?

Задача разложения на простые множители была сведена к нахождению периодичности p функции $a^x \bmod(N)$. На самом деле возможно попытаться найти p программой для традиционного процессора. Все, что для этого необходимо, — многократно вычислять $a^x \bmod(N)$ для увеличивающихся значений x с подсчетом того, сколько значений было опробовано, и отслеживанием полученных возвращаемых значений. Как только возвращаемое значение повторится, можно остановиться и выбрать в качестве периода количество опробованных значений.

Метод нахождения периода $a^x \bmod(N)$ от «грубой силы» предполагает, что если от функции было получено то же значение, то был пройден один полный период повторения p. Хотя это и не очевидно, математическое свойство функции $a^x \bmod(N)$ заключается в том, что любое заданное значение может приниматься только один раз за один период. Следовательно, в тот момент, когда результат будет получен дважды, мы знаем, что был завершен полный период.

В листинге 12.2 реализовано это неквантовое решение по поиску p методом «грубой силы».

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=12-2.

Листинг 12.2. Разложение на множители без QPU

```
function ShorNoQPU(N, precision_bits, coprime) {
    // Традиционная замена для квантовой части алгоритма Шора
    varwork = 1;
    var max_loops = Math.pow(2, precision_bits);
    for (var iter = 0; iter < max_loops; ++iter) {
        work = (work * coprime) % N;
        if (work == 1) // Найдено повторение
            return iter + 1;
    }
    return 0;
}
```

Код в листинге 12.2 быстро выполняется для всего диапазона чисел. Цикл нахождения периода должен выполняться только до того момента, когда будет найдено первое повторение, после чего он останавливается. Зачем же тогда нужен QPU?

Хотя `ShorNoQPU()` в листинге 12.2 не кажется таким уж затратным, количество итераций, необходимых для нахождения периодичности, растет экспоненциально с количеством битов в N (рис. 12.2).

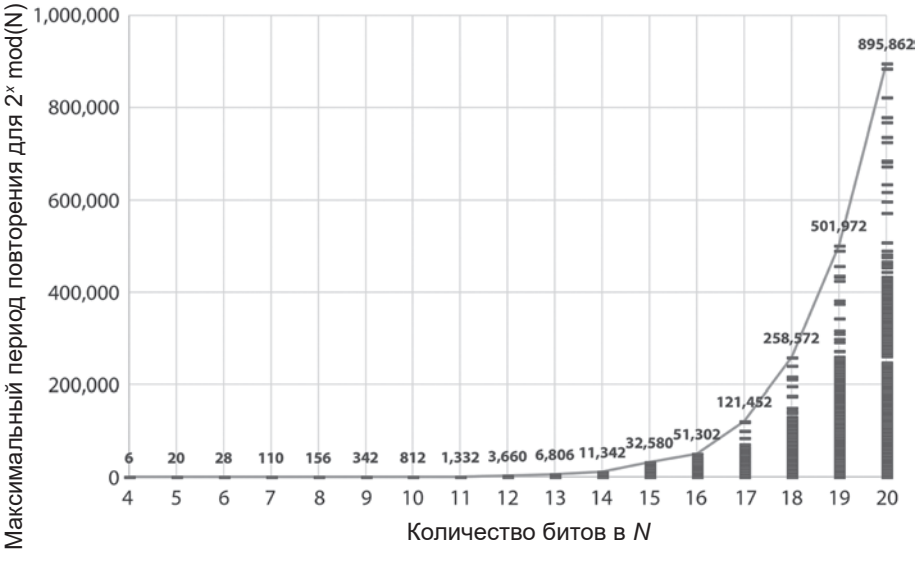


Рис. 12.2. Максимальное количество итераций, необходимых для нахождения периодичности для целого числа, представленного битовой последовательностью длины N. Каждый столбец также образует гистограмму, описывающую распределение периодичности для целых чисел, представленных битовыми последовательностями длины N

Существуют более эффективные методы нахождения простых множителей на традиционных процессорах (например, общий метод решета числового поля), но все они сталкиваются с аналогичными проблемами масштабирования при возрастании *N*. Время выполнения более эффективного алгоритма для разложения на традиционном процессоре возрастает экспоненциально от размера входных данных, тогда как время выполнения алгоритма Шора масштабируется полиномиально.

А теперь подключим к делу QPU.

Квантовое решение

Хотя в следующем разделе мы подробно объясним, как работает функция `ShorQPU()`, сначала стоит сказать несколько слов о нестандартном использовании QFT.

Как вскоре будет показано, благодаря нескольким начальным операциям `HAD`, `ShorQPU()`представляет $a^x \bmod(N)$ в амплитудах и относительных фазах нашего регистра QPU. Вспомните, о чем говорилось в главе 7: QFT реализует дискретное преобразование Фурье и оставляет выходной регистр QPU в суперпозиции разных частот, содержащихся во вводе.

На рис. 12.3 показано, как выглядит дискретное преобразование Фурье для $a^x \bmod(N)$.

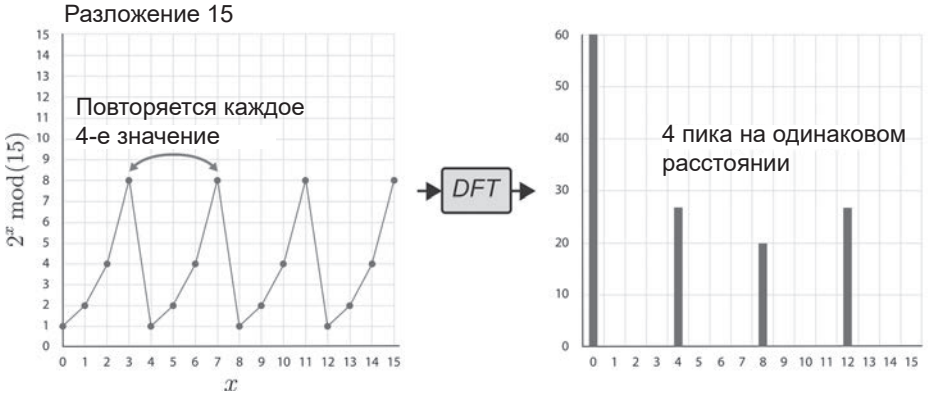


Рис. 12.3. Вычисления, выполненные при разложении на простые множители числа 15

Дискретное преобразование Фурье включает пик на правильной сигнальной частоте со значением 4, из чего можно легко вычислить *p*. Все это хорошо для традиционного дискретного преобразования Фурье, но вспомните,

о чем говорилось в главе 7: если выполнить QFT для сигнала, эти выходные пики будут проявляться в суперпозиции внутри выходного регистра QPU. Маловероятно, чтобы значение, полученное от QFT после операции READ, вернуло искомую частоту.

Может показаться, что наша идея об использовании QFT не оправдала ожиданий. Но если поэкспериментировать с разными значениями N, в результатах DFT для этой конкретной разновидности сигналов обнаруживается нечто интересное. На рис. 12.4 показано дискретное преобразование Фурье для $a^x \bmod(N)$ с $N = 35$.

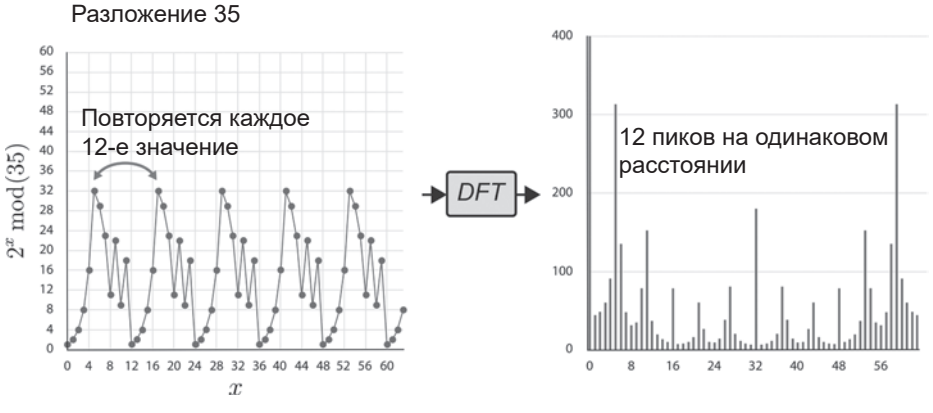


Рис. 12.4. Вычисления, выполненные при разложении на простые множители числа 35

В данном случае период повторения $a^x \bmod(N)$ равен 12, поэтому в дискретном преобразовании Фурье присутствуют 12 равномерно распределенных пиков с наибольшей высотой (значения, которые мы с большой вероятностью будем наблюдать при чтении после QFT). Экспериментируя с разными значениями N, мы начинаем замечать одну тенденцию: для закономерности с периодом повторения p амплитуда преобразования Фурье содержит ровно p равномерно распределенных пиков.

Так как пики распределены равномерно, а размер регистра известен, мы можем оценить количество пиков в QFT — при том что наблюдать сможем не более одного из них. (Вскоре будет приведен алгоритм для решения этой задачи.) Из экспериментальных результатов мы знаем, что количество пиков равно периоду повторения p нашего входного сигнала — именно то, что требовалось!

Это применение QFT менее прямолинейно, чем то, что рассматривалось в главе 7, но оно демонстрирует принципиальный момент — не бойтесь

использовать все инструменты, находящиеся в вашем распоряжении, для экспериментов с содержимым регистра QPU. Отрадно, что проверенная временем мантра «Изменяем код и смотрим, что получится» работает даже с переходом на QPU.

Шаг за шагом: разложение числа 15 на простые множители

Последовательно проанализируем, как QPU используется для разложения числа 15. (Спойлер: ответ 3×5.) Наша реализация алгоритма Шора усложняется при переходе к большим числам, но 15 станет хорошей отправной точкой. Приведенное ниже описание использует настройки из листинга 12.4 для демонстрации работы алгоритма.

Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=12-3.

Листинг 12.3. Разложение на простые множители числа 15

```
var N = 15;              // Число, раскладываемое на множители
var precision_bits = 4; // См. описание в тексте
var coprime = 2;         // Должно быть равно 2 в этой реализации

var result = Shor(N, precision_bits, coprime);
```

Первому аргументу N в этом примере присваивается значение 15 — число, которое требуется разложить на множители. Второй аргумент `precision_bits` равен 4. При большем количестве битов точности в общем случае вероятность получения правильного ответа повышается, но зато требует большего количества кубитов и значительно большего количества команд. Третий аргумент `coprime` остается равным 2 — это единственное значение, поддерживаемое нашей упрощенной реализацией алгоритма Шора для QPU.

Мы уже знаем, что главная функция `Shor()` выполняет две меньшие функции. Программа для QPU определяет период повторения, после чего передает результат второй функции, определяющей простые множители с использованием традиционной цифровой логики для традиционного процессора.

Составляющие функции `Shor()` выполняют следующие действия:

- шаги 1–4 создают суперпозицию результатов $a^x \bmod(N)$;

- шаги 5–6 реализуют описанный выше прием QFT для получения периода p этого сигнала;
- шаги 7–8 используют p в алгоритме для традиционного процессора с целью нахождения простых множителей.

Мы разберем последовательность этих шагов для восьмикубитного регистра. Четыре из этих кубитов будут использоваться для представления (суперпозиций) различных значений x, передаваемых $a^x \bmod(N)$, тогда как четыре других кубита будут обрабатывать и отслеживать значения, возвращаемые функцией.

Это означает, что в общем придется отслеживать $2^8 = 256$ значений — 256 кругов в круговой записи. Размещая круги в квадрате 16×16, мы можем наглядно представить 16 состояний каждого из 4-кубитных регистров по отдельности (о таком использовании круговой записи рассказано в главе 3). Для удобства мы также пометили эти структуры из сетки 16×16 кружками, показывающие вероятности каждого значения в каждом из двух регистров. Впрочем, довольно разговоров; займемся разложением на простые множители на QPU!

Шаг 1: инициализация регистров QPU

В начале квантовой части программы мы инициализируем регистры цифровыми значениями 1 и 0 (рис. 12.5). Вскоре мы покажем, что инициализация рабочего регистра значением 1 необходима для нашего способа вычисления $a^x \bmod(N)$.

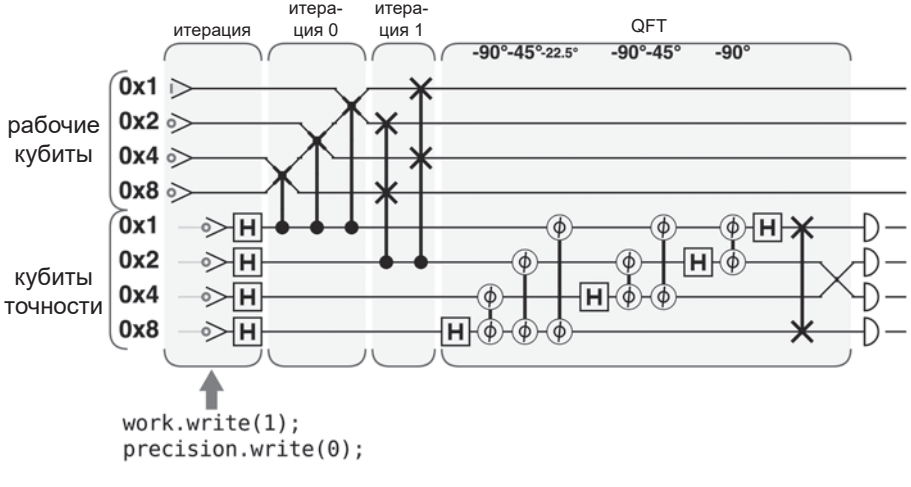


Рис. 12.5. Команды QPU для шага 1

На рис. 12.6 представлено состояние двух регистров в круговой записи после инициализации.

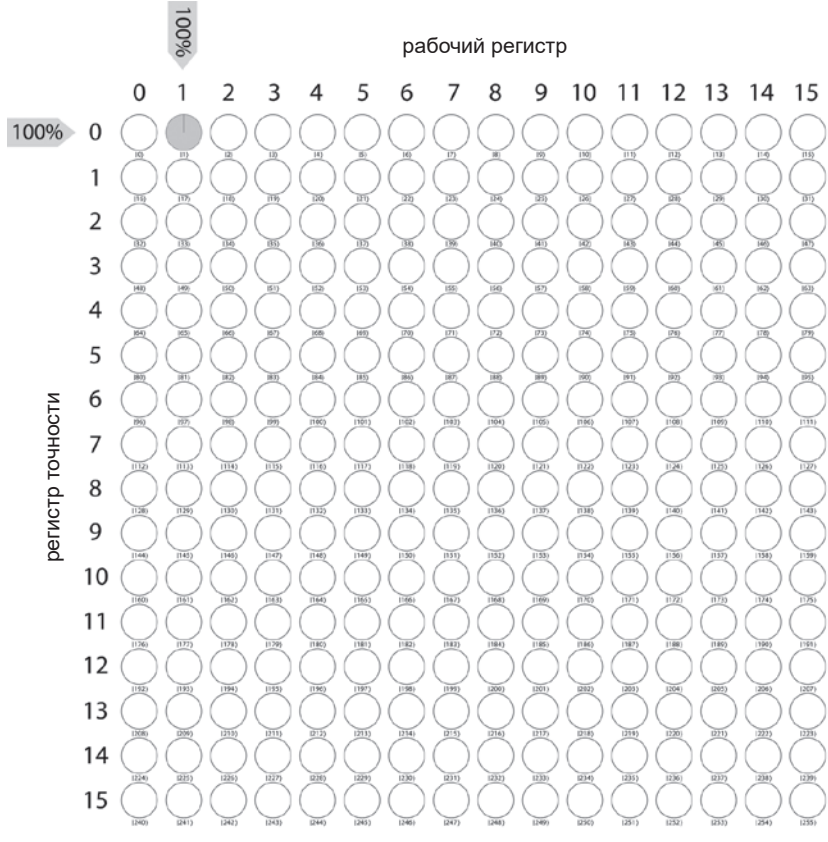


Рис. 12.6. Шаг 1: рабочий регистр и регистр точности инициализируются значениями 1 и 0 соответственно

Шаг 2: перевод в квантовую суперпозицию

Регистр точности используется для представления значений x, передаваемых функции $a^x \bmod(N)$. Квантовая суперпозиция будет использоваться для параллельного вычисления этой функции для нескольких значений x, поэтому мы применили операции HAD (рис. 12.7) для перевода регистра точности в суперпозицию всех возможных значений.

Каждая строка структуры кругов, показанной на рис. 12.8, готова к тому, чтобы рассматриваться как отдельное входное значение для параллельных вычислений.

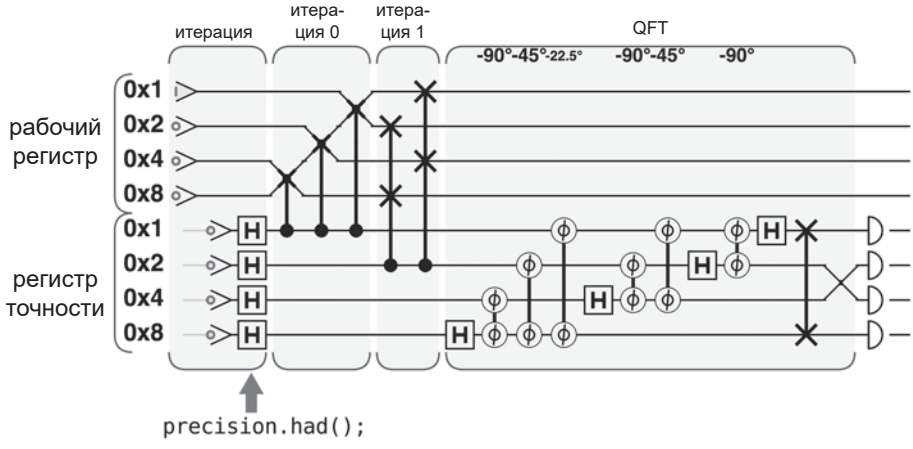


Рис. 12.7. Команды QPU для шага 2

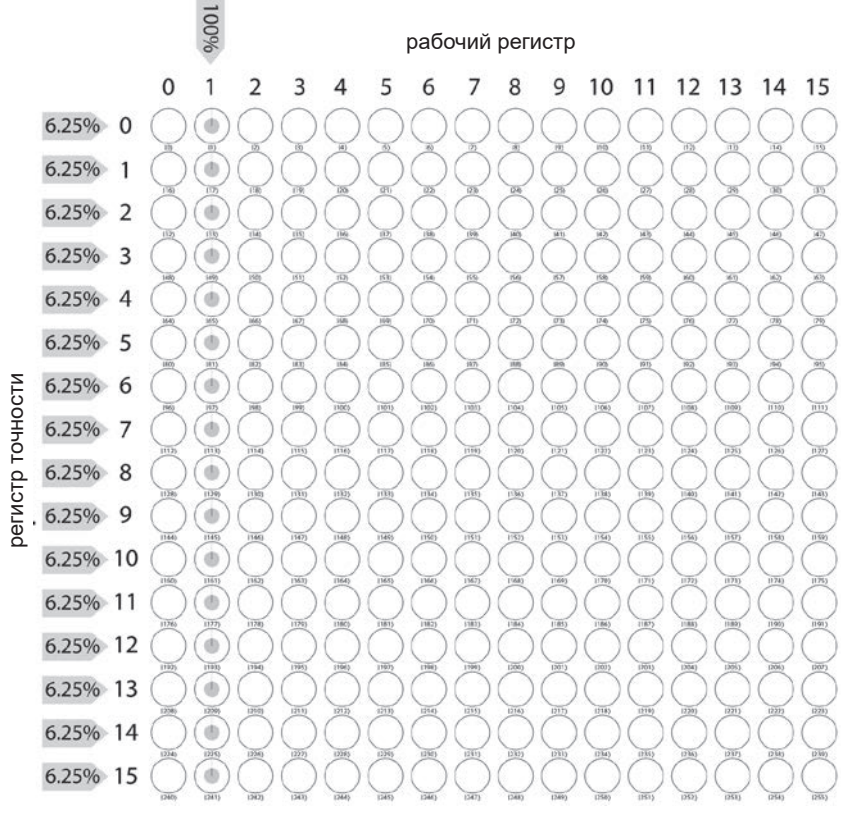


Рис. 12.8. Шаг 2: суперпозиция регистра точности готовит к вычислению $a^x \bmod(N)$ в суперпозиции

Шаг 3: условное умножение на 2

Теперь нужно выполнить функцию $a^x \bmod(N)$ для суперпозиции входных значений, содержащейся в регистре точности; рабочий регистр будет использоваться для хранения результатов. Вопрос в том, как выполнить $a^x \bmod(N)$ с кубитным регистром.

Вспомните, что мы выбрали $a = 2$ в качестве значения `coprime`; соответственно, часть a^x превращается в 2^x. Другими словами, для выполнения этой части функции необходимо умножить рабочий регистр на 2. Количество выполнений этого умножения равно x, где x — значение, представленное в двоичном виде в регистре точности.

Умножение на 2 (или любую степень 2) достигается в двоичном регистре простым сдвигом битов. В нашем случае каждый кубит переходит в позицию со следующим весом (при помощи метода QCEngine `rollLeft()`). Чтобы выполнить x умножений на 2, мы просто выполняем условное умножение по значениям кубитов, содержащимся в регистре точности. Учтите, что *для представления значений x будут использоваться только два кубита с наименьшим весом из регистра точности* (то есть x может принимать значения 0, 1, 2, 3). Соответственно, условное умножение будет управляться этими двумя кубитами.

Почему регистр точности является четырехкубитным, если для x необходимы только два кубита? Хотя дополнительные кубиты `0x4` и `0x8`, присутствующие в регистре точности, нигде не используются напрямую, их включение во все последующие вычисления обеспечивает эффективное расширение закономерностей круговой записи, наблюдаемых в регистрах QPU. Алгоритм Шора будет нормально работать и без них, но с учебной точки зрения нам будет немного труднее обозначить закономерности, объясняющие, как работает алгоритм.

Если кубит с наименьшим весом в регистре точности имеет значение 1, то с рабочим регистром необходимо будет выполнить только одно умножение ×2, поэтому мы выполняем одну операцию `rollLeft()` с рабочим регистром (рис. 12.9).

Результат показан на рис. 12.10.

В программировании QPU использование условных вентилей как эквивалента операций `if/then` чрезвычайно полезно, так как «условие» фактически проверяется для всех возможных значений сразу.

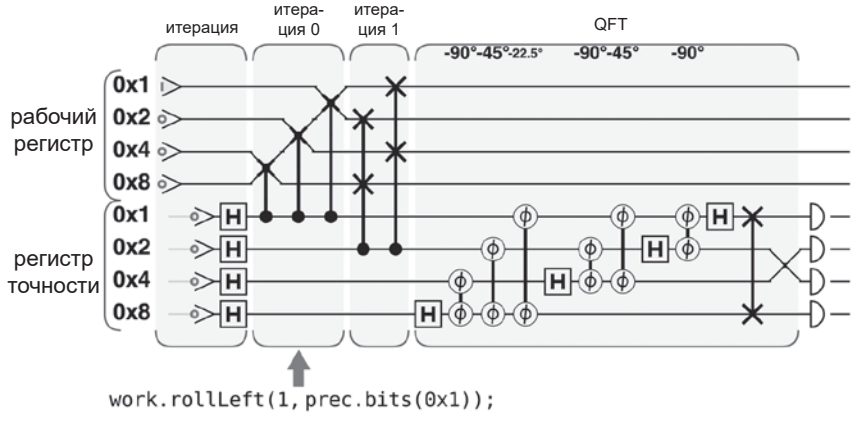


Рис. 12.9. Команды QPU для шага 3

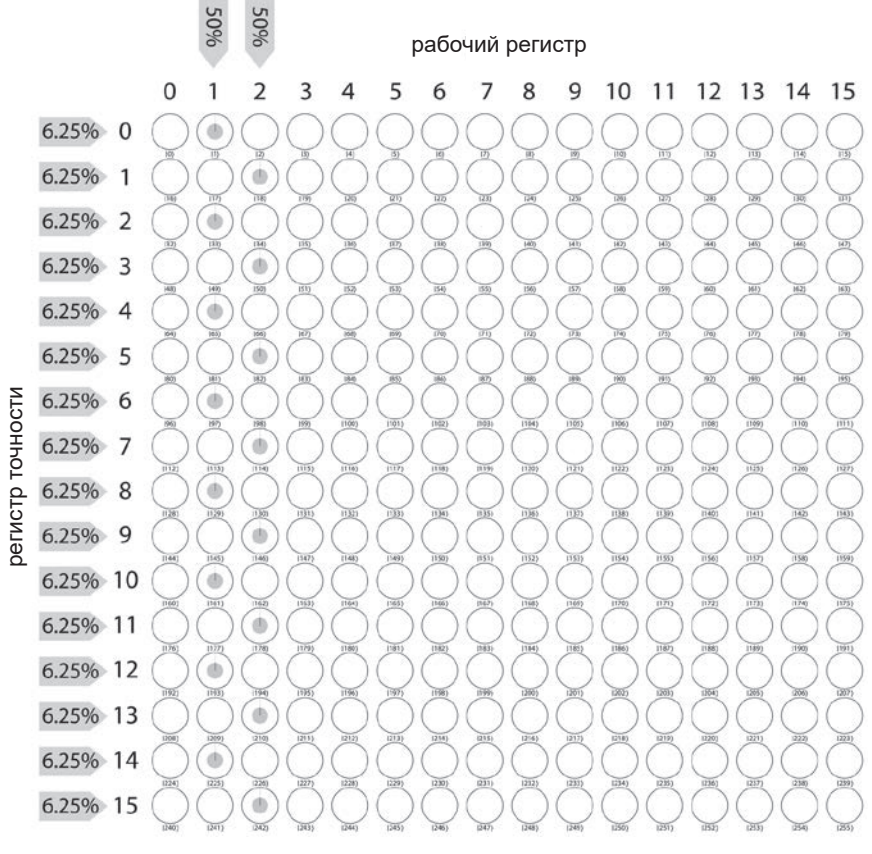


Рис. 12.10. Шаг 3: чтобы реализовать a^x, мы умножаем на 2 все кубиты в рабочем регистре при условии, что младший кубит регистра точности равен 1

Шаг 4: условное умножение на 4

Если следующий по весу кубит регистра точности равен 1, то это подразумевает, что двоичное значение x также требует еще *двух* умножений на 2 рабочего регистра. Следовательно, как показано на рис. 12.11, мы выполняем сдвиг на два кубита (то есть две операции `rollLeft()` — в соответствии со значением кубита `0x2` из регистра точности.

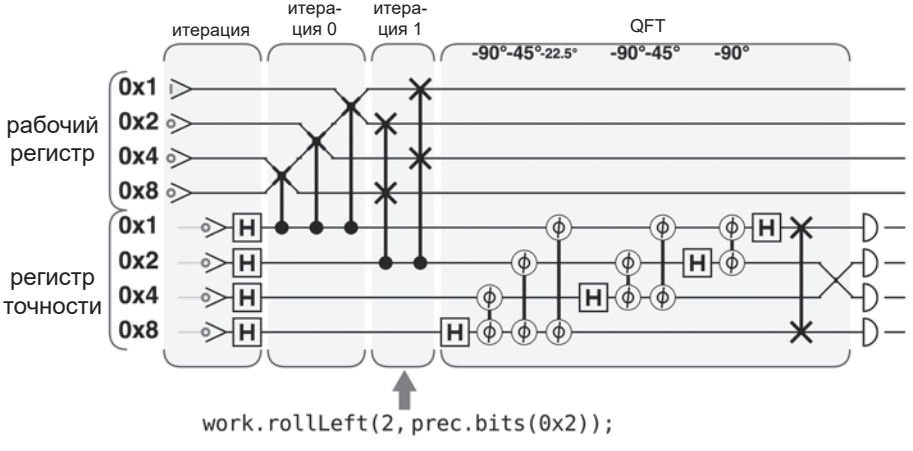


Рис. 12.11. Команды QPU для шага 4

Теперь в рабочем регистре находится значение a^x для значения x, закодированного в (первых двух) кубитах регистра точности. В нашем случае регистр точности находится в равномерной суперпозиции возможных значений x, чтобы получить соответствующую суперпозицию соответствующих значений a^x.

Хотя мы выполнили все необходимые умножения на 2, может показаться, что функция $a^x \bmod(N)$ не реализована, — ведь мы не сделали ничего для реализации части mod. На самом деле для нашего конкретного примера схема автоматически решает задачу вычисления остатка. В следующем разделе мы объясним, как это происходит.

На рис. 12.12 показано, как нам удалось вычислить $a^x \bmod(N)$ для каждого значения x из регистра точности в суперпозиции.

На предшествующей круговой записи видна знакомая закономерность. После выполнения функции $a^x \bmod(N)$ для регистра QPU амплитуды суперпозиции точно совпадают с графиком $a^x \bmod(N)$, который был впервые показан в начале этой главы на рис. 12.1 (хотя и с поворотом осей на 90°). Графики изображены на рис. 12.13.

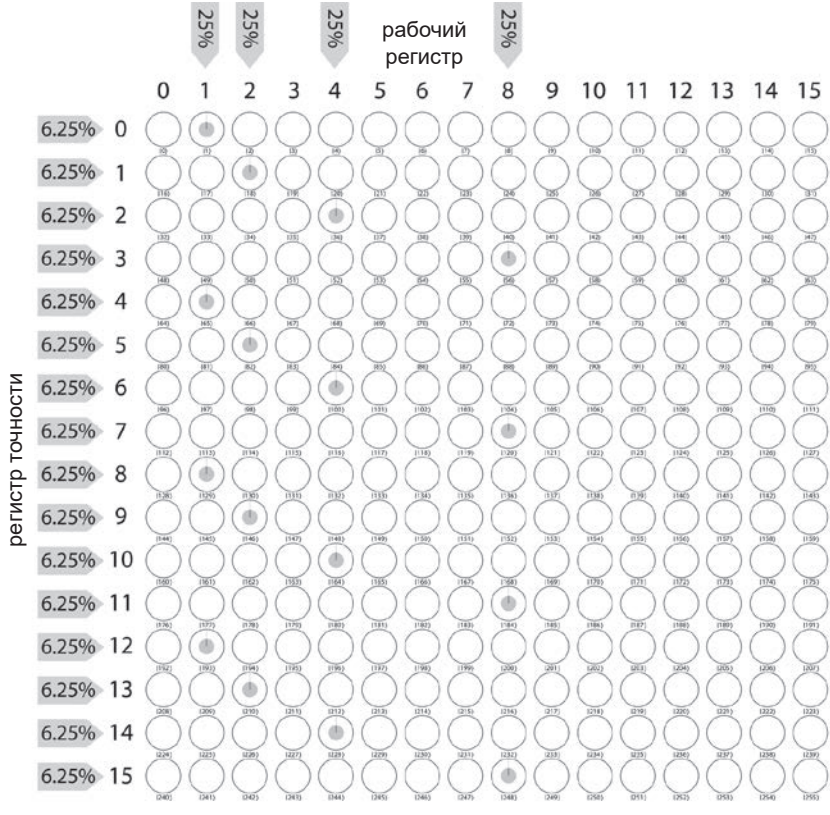


Рис. 12.12. Шаг 4: теперь рабочий регистр содержит суперпозицию 2^x mod(15) для всех возможных значений *x* в регистре точности

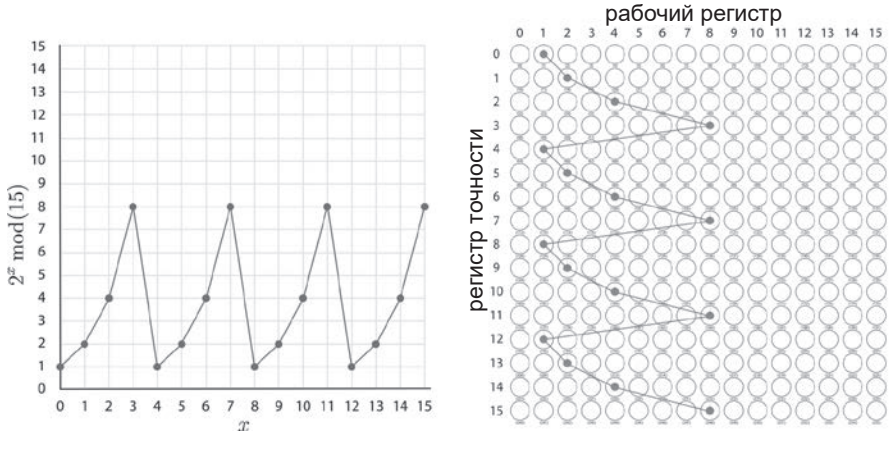


Рис. 12.13. Хм, что-то знакомое!

Теперь повторяющийся сигнал для $a^x \bmod(N)$ закодирован в регистрах QPU. Конечно, при попытке прочитать любой из регистров вы просто получите случайное значение для регистра точности с соответствующим рабочим результатом. К счастью, у нас в запасе имеется прием с подсчетом пиков дискретного преобразования Фурье для нахождения периода повторения этого сигнала. Пришло время применить QFT.

Шаг 5: квантовое преобразование Фурье

Применяя QFT к регистру точности, как показано на рис. 12.14, мы фактически выполняем DFT для каждого столбца данных, преобразуя состояния регистра точности (показанные как строки нашей структуры круговой записи) в суперпозицию составляющих частот периодического сигнала.

При виде периодической закономерности, проявляющейся в круговой записи на рис. 12.12, может возникнуть вопрос, почему QFT не нужно применять к обоим регистрам (в конце концов, функция $a^x \bmod(N)$ применялась к рабочему регистру!). Выберите значение рабочего регистра с ненулевой комплексной амплитудой на решетке круговой записи и просмотрите верхние и нижние круги этого столбца. Вы увидите, что с изменением значения регистра точности мы получаем периодическое изменение комплексной амплитуды (с периодом 4 в данном случае). Следовательно, в данном случае мы хотим найти QFT для этого регистра.

Рисунок 12.15 при просмотре сверху вниз теперь напоминает график DFT, который был показан в начале этой главы на рис. 12.3. Сходство продемонстрировано на рис. 12.16. Обратите внимание: на рис. 12.15 преобразование QFT также повлияло на относительные фазы комплексных амплитуд регистров.

Каждый столбец на рис. 12.16 теперь содержит правильное количество частотных пиков (4 в данном случае). Вспомните, о чем говорилось ранее: если мы можем подсчитать эти пики, то этой информации достаточно для нахождения искомого множителя с применением традиционной цифровой логики. Прочитаем регистр точности операцией `READ` для получения искомой информации.

Шаг 6: чтение квантового результата

Операция `READ`, используемая на рис. 12.17, возвращает случайное цифровое значение, взвешенное по вероятностям на круговой диаграмме. Кроме того, `READ` уничтожает все значения из суперпозиции, не согласующиеся с наблюдаемым цифровым результатом.

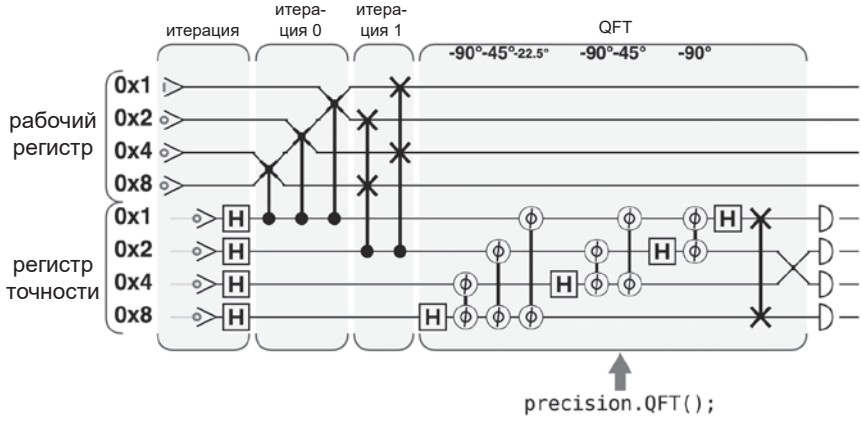


Рис. 12.14. Команды QPU для шага 5

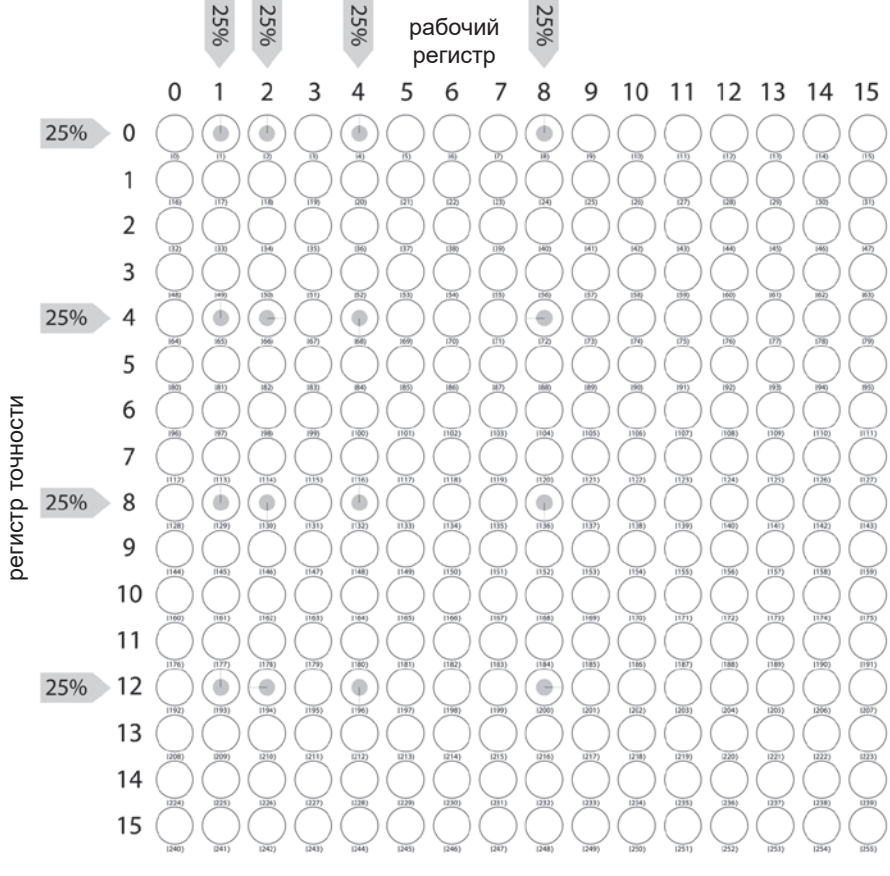


Рис. 12.15. Шаг 5: пики частот в регистре точности после применения QFT

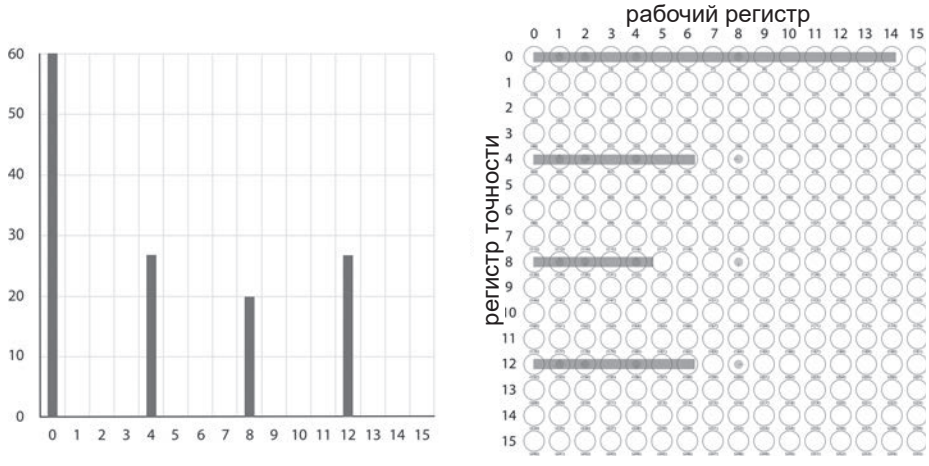


Рис. 12.16. И снова похоже на то, что вы уже видели

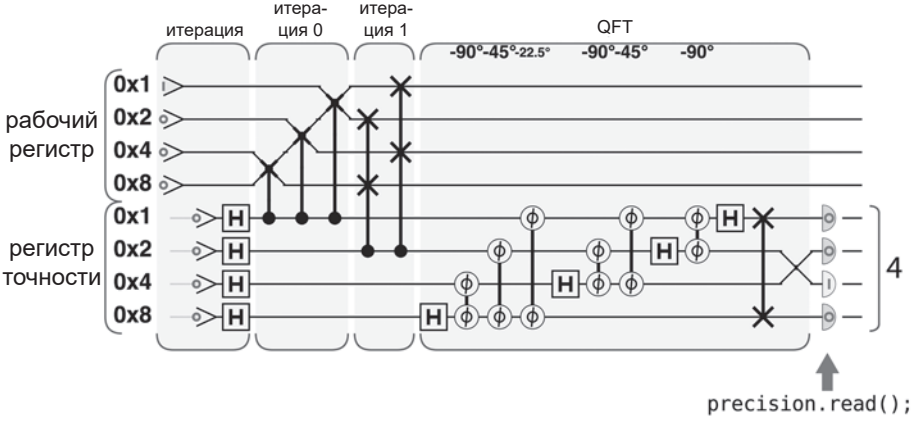


Рис. 12.17. Команды QPU для шага 6

В примере результата на рис. 12.18 число 4 было случайным образом получено из 4 самых вероятных вариантов. На этом квантовая часть алгоритма Шора завершена, и результат READ передается традиционной функции `ShorLogic()`, используемой на следующем шаге.

Шаг 7: цифровая логика

В результате работы, выполненной к настоящему моменту, было получено число 4, хотя по виду рис. 12.15 с таким же успехом можно было получить случайные результаты 0, 4, 8 или 12.

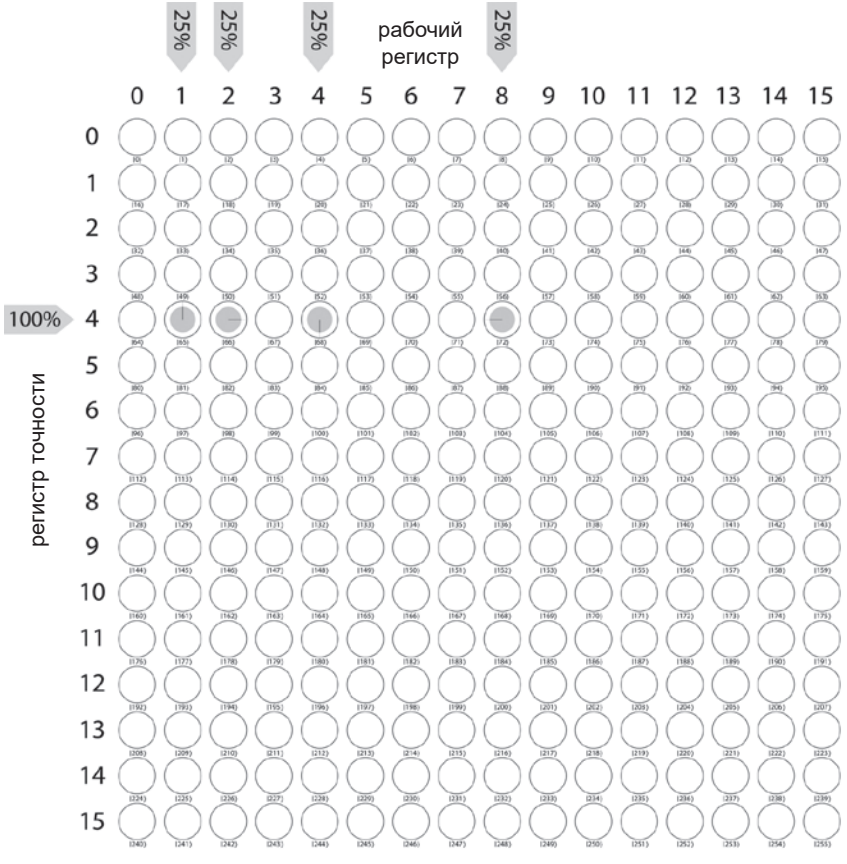


Рис. 12.18. Шаг 6: после чтения регистра точности операцией READ

Как упоминалось ранее, зная тот факт, что пики QFT равномерно распределяются в регистре, мы можем определить, что периоды соответствуют значению, прочитанному операцией READ с использованием традиционной цифровой логики. Функция `estimate_num_spikes()` в листинге 12.4 наглядно показывает логику, которая при этом используется. В некоторых случаях эта функция может вернуть более одного потенциального количества пиков для графика DFT. Например, если передать ей прочитанное значение 4, она вернет два значения 4 и 8; каждое из них представляет количество пиков в DFT, соответствующее прочитанному значению.

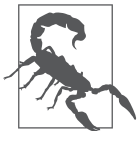
Пример кода

Этот пример можно выполнить онлайн по адресу http://oreilly-qc.github.io?p=12-4.

Листинг 12.4. Функция для оценки количества пиков на основании результата QPU

```
function estimate_num_spikes(spike, range)
{
    if (spike < range / 2)
        spike = range - spike;
    var best_error = 1.0;
    var e0 = 0, e1 = 0, e2 = 0;
    var actual = spike / range;
    var candidates = []
    for (var denom = 1.0; denom < spike; ++denom)
    {
        var numerator = Math.round(denom * actual);
        var estimated = numerator / denom;
        var error = Math.abs(estimated - actual);
        e0 = e1;
        e1 = e2;
        e2 = error;
        // Искать локальный минимум, который превосходит
        // текущую лучшую погрешность
        if (e1 <= best_error && e1 < e0 && e1 < e2)
        {
            var repeat_period = denom - 1;
            candidates.push(denom - 1);
            best_error = e1;
        }
    }
    return candidates;
}
```

Так как (в данном примере) мы получили два потенциальных результата (4 и 8), необходимо проверить оба значения и определить, приведут ли они к простым множителям числа 15. Мы представили метод `ShorLogic()`, реализующий формулы gcd, которые вычисляют простые множители по заданному количеству пиков DFT (о чем говорилось в начале главы, в листинге 12.1). Сначала в этом выражении опробуется значение 4, для которого возвращаются значения 3 и 5.



Не все доступные значения приводят к правильному ответу. Что произойдет, если вы получите значение 0? Этот вариант на 25% вероятен, в этом случае функция `estimate_num_spikes()` вообще не возвращает потенциальных кандидатов, и программа завершается неудачей. Такие ситуации часто встречаются с квантовыми алгоритмами и не создают проблем, если ответ можно быстро проверить на допустимость. В нашем случае мы выполняем такую проверку, а потом при необходимости выполняем программу с самого начала.

Шаг 8: проверка результата

Простые множители целого числа, может быть, сложно найти, но найденный ответ проверяется быстро. Вы можете легко убедиться в том, что числа 3 и 5 являются простыми, и при этом являются множителями 15 (а следовательно, нет необходимости даже проверять второе значение 8 в `ShorLogic()`). Успех!

Важные подробности

В этой главе представлена упрощенная версия алгоритма, который обычно бывает очень сложным. В нашей версии алгоритма Шора ряд аспектов был упрощен ради простоты изложения материала, хотя и за счет общности. Не углубляясь в подробности, в этом разделе кратко упоминаются некоторые необходимые упрощения. Дополнительную информацию также можно найти в коде, доступном в интернете.

Вычисление остатка

Ранее уже упоминалось, что в вычислениях $a^x \bmod(N)$ на QPU та часть функции, в которой вычисляется остаток, реализуется автоматически. Это было удачным совпадением, которое было связано с конкретным числом, раскладываемым на множители; к сожалению, в общем случае этого не происходит. Вспомните, что мы умножали рабочий регистр на степени 2 посредством сдвига битов. Если начать с 1 и выполнить сдвиг 4 раза, мы получим 2^4=16; но так как мы использовали только 4-разрядное число и допустили циклический перенос сдвинутых битов, вместо 16 мы получим 1 — именно то, что мы бы получили при выполнении умножений с последующим вычислением mod(15).

Вы можете убедиться в том, что этот прием работает и при попытке разложить на множители значение 21; тем не менее он терпит неудачу при попытке разложения большего (но при этом относительно малого) числа — такого как 35. Что еще можно сделать в этих более общих случаях?

Когда традиционный компьютер вычисляет остаток от деления числа (например, `1024 % 35`), он выполняет целочисленное деление и возвращает остаток. Количество традиционных логических вентилей, необходимых для выполнения целочисленного деления, очень (*очень!*) велико, а их реализация для QPU выходит за рамки книги.

Существует вариант решения задачи с использованием метода вычисления остатка — менее хитроумного, но лучше подходящего для операций QPU.

Допустим, вы хотите вычислить $y \bmod(N)$ для некоторого значения y. Это делается в следующем фрагменте:

```
y -= N;
if (y < 0) {
    y += N;
}
```

Мы просто вычитаем N из полученного значения и по знаку результата определяем, оставить результат или вернуться к исходному значению. На традиционном компьютере такое решение считалось бы нежелательным. Однако в данном случае он дает ровно то, что требуется: правильный ответ с использованием декрементов, инкрементов и сравнений — как мы знаем из главы 5, вся эта логика может быть реализована операциями QPU.

Схема для вычисления остатка этим способом (для некоторого значения `val`) показана на рис. 12.19.

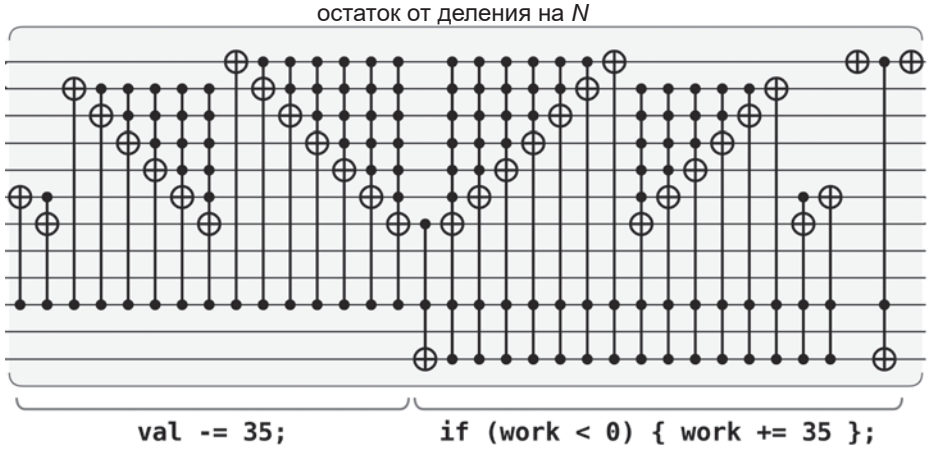


Рис. 12.19. Квантовые операции для умножения на 2 с вычислением остатка от деления на 35

Хотя в этом примере используется большое количество сложных операций для выполнения простого вычисления, он может выполнить операцию вычисления остатка в суперпозиции.

Реализация вычисления остатка на рис. 12.19 в действительности использует один служебный кубит для сохранения знакового бита рабочего регистра, используемого в условном сложении.

Время и память

Операция вычисления остатка, описанная в предыдущем разделе, существенно замедляет вычисления для общего случая — прежде всего потому, что она требует выполнять операции умножения на 2 по одной. Это увеличивает количество необходимых операций (а следовательно, и общее время выполнения) и сокращает преимущества от использования QPU. Проблема может быть решена увеличением количества используемых кубитов и последующим применением операции вычисления остатка логарифмическое число раз. За примером обращайтесь по адресу http://oreilly-qc.github.io?p=12-A. Значительная часть проблем при программировании QPU связана с поиском баланса между *глубиной* программы (количеством операций) и количеством необходимых кубитов.

Другие значения переменной coprimes

Рассмотренная реализация раскладывает на простые множители многие числа, но в ряде случаев возвращает нежелательные результаты. Например, при попытке разложить на простые множители число 407 вы получите `[407, 1]`. Хотя формально этот результат правилен, вы бы наверняка предпочли *нетривиальные* множители 407, то есть `[37, 11]`.

Проблема решается заменой `coprime=2` другим простым числом, хотя квантовые операции, требуемые для выполнения возведения в степень числа, отличного от 2, выходят за рамки книги. Выбор `coprime=2` — полезное упрощение для демонстрационных целей.

13 Квантовое машинное обучение

На момент написания книги квантовое машинное обучение (QML, Quantum Machine Learning) было самой эффектной комбинацией модных словечек. О квантовом машинном обучении написано много книг; эта тема часто вызывает хайп и в то же время не получает того внимания, которого она заслуживает. В этой главе мы попытаемся дать представление о том, как QPU могут преобразовать область машинного обучения, и в то же время указать на основные проблемы, присущие работе с квантовыми данными.

Полезные практические применения QML требуют очень большого количества кубитов. По этой причине материал в обзоре применений QML неизбежно излагается на очень высоком уровне. Такой стиль обзора также уместен с учетом стремительных изменений в этой области, которая еще только формируется. Хотя наше обсуждение будет скорее схематичным, чем прагматическим, в нем будет интенсивно использоваться наш практический опыт работы с примитивами из предыдущих глав.

Мы опишем три разных практических применения QML: решение систем линейных уравнений, квантовый анализ главных компонент и квантовый метод опорных векторов. Они были выбраны из-за своей связи с машинным обучением и простоты изложения. Кроме того, традиционные аналоги этих примеров знакомы практически каждому, кто когда-либо занимался машинным обучением. Мы ограничимся краткими описаниями традиционных «предков» каждого примера QML.

При обсуждении QML часто используются следующие термины из области машинного обучения:

Признаки (features)

Термин, используемый для описания измеримых свойств точек данных, доступных для построения прогнозов по моделям машинного обучения.

Часто все возможные значения этих признаков рассматриваются как определение *пространства признаков*.

Контролируемая

Так обозначаются модели машинного обучения, которые должны *обучаться* на наборе точек в пространстве признаков с уже известными правильными классами или откликами. Только после этого нормально обученная модель может использоваться для классификации (или прогнозирования) отклика для новых точек в пространстве признаков.

Неконтролируемая

Относится к моделям машинного обучения, способным выявлять закономерности и структуру в обучающих данных, для которых отклик неизвестен.

Классификация

Используется для описания контролируемых предиктивных моделей, которые относят заданную точку в пространстве признаков к одному из нескольких дискретных классов.

Регрессия

Используется для описания контролируемых моделей, прогнозирующих некоторую непрерывно изменяемую переменную отклика.

Снижение размерности

Одна из форм неконтролируемой предварительной обработки данных, которая может быть полезной для всех типов моделей машинного обучения. Снижение размерности направлено на сокращение количества признаков, необходимого для описания задачи.

Помимо этой терминологии, мы также будем использовать математические описания задач машинного обучения. По этой причине эта глава содержит чуть больше математических выкладок, чем предыдущие.

Наш первый практический пример QML учит тому, как QPU может помочь в решении систем линейных уравнений.

Решение систем линейных уравнений

Безусловно, системы линейных уравнений играют основополагающую роль в машинном обучении, но они также лежат в основе самых разных об-

ластей прикладной математики. *Алгоритм HHL*[1] (часто называемый просто HHL), который мы представим для эффективного решения таких систем на QPU, — фундаментальный и мощный инструмент; вы также увидите, что он является ключевым структурным элементом в других приложениях QML. Также рассматривались возможности применения HHL в разных областях, от моделирования электрических эффектов до оптимизации вычислений в компьютерной графике.

В самом начале обзора алгоритма HHL мы припомним математику, необходимую для описания традиционных систем линейных уравнений. Затем мы перейдем к специфической квантовой стороне HHL, опишем его преимущества по быстродействию и — что еще важнее — ограничения. В завершение будет приведено более подробное описание внутренних механизмов работы HHL.

Описание и решение систем линейных уравнений

Системы линейных уравнений наиболее компактно представляются в форме умножения матриц. Собственно, для специалиста по решению уравнений термины «матрицы» и «линейные уравнения» эквивалентны. Для примера рассмотрим систему из двух линейных уравнений: $3x_1 + 4x_2 = 3$ и $2x_1 + x_2 = 3$. Эти уравнения можно эквивалентно и намного более компактно представить в виде одной матричной формулы 13.1.

Формула 13.1. Использование матриц для описания систем линейных уравнений

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}.$$

Два линейных уравнения можно восстановить по правилам умножения матриц. В более общем виде система из n линейных уравнений для n переменных можно записать в виде матричного уравнения, содержащего матрицу $n \times n$ A и n-мерный вектор $\vec{b}$:

$$A\vec{x} = \vec{b}.$$

Здесь также вводится вектор $\vec{x} = [x_1, ..., x_n]$ из n переменных, для которых решается система уравнений.

[1] Harrow et al., 2009.

В матричном виде задача решения системы уравнений сводится к обратимости матрицы A. Если можно получить обратную матрицу A^{-1}, то n неизвестных переменных можно легко определить по формуле 13.2.

Формула 13.2. Решение систем линейных уравнений с использованием обратной матрицы

$$\vec{x} = A^{-1}\vec{b}.$$

Существует много традиционных алгоритмов для поиска обратной матрицы. Самый эффективный алгоритм зависит от того, что матрица обладает некоторыми полезными свойствами.

Следующие параметры матрицы могут влиять на быстродействие как традиционных, так и квантовых алгоритмов:

n

Размер системы линейных уравнений. Этот параметр эквивалентен размерности A — если вы хотите решить систему из n линейных уравнений для нахождения n переменных, то A будет матрицей $n \times n$.

k

Число обусловленности матрицы A, представляющее систему линейных уравнений. Для системы уравнений $A\vec{x} = \vec{b}$ число обусловленности сообщает, в какой мере погрешность в определении $\vec{b}$ влияет на погрешность, которую можно ожидать от решения $\vec{x}=A^{-1}\vec{b}$. k вычисляется как максимальное отношение между относительной погрешностью во вводе $\vec{b}$ и в выводе $\vec{x} = A^{-1}\vec{b}$. Как выясняется, значение k может быть эквивалентно вычислено[1] как отношение $|\lambda_{max}|/|\lambda_{min}|$ абсолютных значений максимального и минимального собственных значений A.

s

Разреженность матрицы A. Определяется как количество ненулевых элементов A.

ε

Точность, требуемая от решения. В случае HHL мы вскоре покажем, что в выводимом состоянии $|\vec{x}\rangle$ вектор решения $\vec{x}$ закодирован по ампли-

[1] Выражение для коэффициента переноса в контексте собственных значений справедливо только в том случае, если матрица A является нормальной. Все матрицы, используемые в HHL, заведомо нормальны из-за их зависимости от квантового моделирования и эрмитовых матриц.

туде. Увеличение ε означает повышение точности, с которой значения $\vec{x}$ представлены в этих амплитудах.

В нашей оценке того, насколько эффективно HHL может обратить матрицу, будут учитываться эти параметры. Под *эффективностью* имеется в виду время выполнения алгоритма (метрика того, сколько фундаментальных операций он должен выполнить). Для сравнения, на момент написания книги ведущим традиционным алгоритмом для решения систем линейных уравнений, вероятно, был *метод сопряженных градиентов*. Он обладает временем выполнения $O(nsk \log(1/\varepsilon))$.

Решение линейных уравнений на QPU

Алгоритм HHL (названный по фамилиям Хэрроу, Хассидима и Ллойда, открывшим его в 2009 году) использует примитивы, которые мы изучали ранее, для нахождения (в определенном смысле) обратной матрицы быстрее, чем было возможно при использовании метода сопряженных градиентов. Мы говорим «*в определенном смысле*», потому что HHL решает специфическую квантовую версию этой задачи. HHL предоставляет решения системы линейных уравнений, закодированных по комплексной амплитуде в регистре QPU, которые, соответственно, недоступны в квантовом виде. И хотя HHL не позволяет решать системы линейных уравнений в традиционном смысле, комплексные амплитудно-кодированные решения все равно могут быть чрезвычайно полезными; более того, они являются важнейшими структурными элементами других практических применений QML.

Как работает HHL

Прежде чем раскладывать HHL на примитивы, мы кратко опишем входные и выходные данные, а также быстродействие.

Входные и выходные данные HHL изображены на рис. 13.1.

Входы

Как видно из рисунка, HHL получает два набора входных регистров и матрицу a (посредством квантового моделирования).

- *Служебный регистр* — содержит служебные кубиты, используемые различными примитивами в HHL; все эти кубиты подготовлены в состоянии $|0\rangle$. Так как HHL работает с данными с фиксированной (или плавающей) точкой и включает нетривиальные арифметические операции (такие как извлечение квадратного корня), то требуется большое коли-

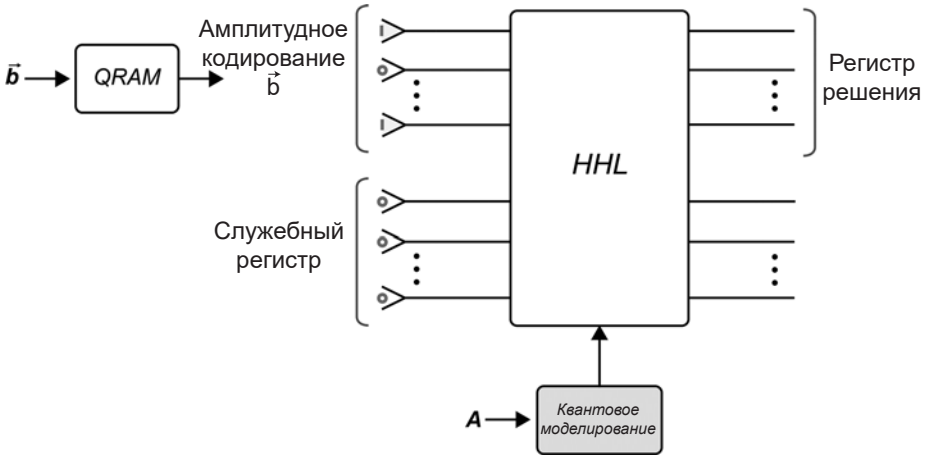


Рис. 13.1. Высокоуровневое представление входных и выходных данных, используемых алгоритмом HHL для решения систем линейных уравнений

чество служебных кубитов. Это усложняет моделирование HHL даже для простейших случаев.

- *Комплексное амплитудное кодирование* $b - \vec{b}$ — также необходимо предоставить HHL вектор $\vec{b}$ из формулы 13.2, амплитудно-кодированный в регистре QPU (в том смысле, который рассматривался в главе 9). Обозначим состояние амплитудного кодирования регистра $\vec{b}$ в виде $|\vec{b}\rangle$. Чтобы подготовить амплитудное кодирование $\vec{b}$, необходимо будет использовать QRAM. А значит, алгоритм HHL фундаментально зависит от существования QRAM.
- *Операция QPU, представляющая A,* — естественно, HHL также необходим доступ к матрице, представляющей систему линейных уравнений. В нижней части рис. 13.1 показано, что HHL требует представления A в виде операции QPU, что может быть достигнуто процессом квантового моделирования, описанным в главе 9. Следовательно, матрица A должна удовлетворять требованиям, которые были отмечены в тексте для процесса квантового моделирования.

Выходы

Из рис. 13.1 также видно, что HHL также имеет два выходных регистра.

- *Регистр решения* — вектор решения $\vec{x}$ амплитудно кодируется в одном выходном регистре QPU (обозначим это состояние $|\vec{x}\rangle$). Как уже упоминалось, из этого следует, что мы *не сможем обратиться к отдельным*

решениям, *скрытым* в комплексных амплитудах квантовой суперпозиции, которые не могут быть эффективно извлечены операциями READ.

- *Служебный регистр* — служебные кубиты возвращаются в их исходное состояние $|0\rangle$, благодаря чему мы можем продолжить пользоваться ими в QPU.

Ниже приведены примеры, которые показывают, что квантовые выходные данные HHL могут быть невероятно полезными, несмотря на их скрытую природу:

1. Вместо полной спецификации решений по всем n переменным в $\vec{x}$ нас может интересовать только некоторая производная характеристика — сумма, среднее значение или даже наличие/отсутствие некоторой частотной составляющей. Возможно, в такой ситуации удастся применить к $|\vec{x}\rangle$ подходящую квантовую схему, которая позволит прочитать производное значение.
2. Если вас интересует лишь проверка того, равен ли вектор решения $\vec{x}$ одному конкретному потенциальному вектору, то мы можем воспользоваться проверкой обменом (см. главу 3) между $|\vec{x}\rangle$ и другим регистром, в котором закодирован потенциальный вектор.
3. Если, например, вы намереваетесь использовать алгоритм HHL в качестве составляющей большего алгоритма, то $|\vec{x}\rangle$ может быть достаточно для ваших целей в неизменном виде.

Так как системы линейных уравнений играют фундаментальную роль во многих областях машинного обучения, HHL становится отправной точкой для ряда других практических применений QML — таких как регрессия[1] и аппроксимация данных[2].

Скорость и некоторые нюансы

Алгоритм HHL обладает временем выполнения[3] $O(k^2 s^2 \varepsilon^{-1} \log n)$.

По сравнению с традиционным методом сопряженных градиентов, обладающим временем выполнения $O(nsk \log(1/\varepsilon))$, HHL очевидным образом обеспечивает экспоненциальное улучшение в зависимости от размера задачи (n).

---

[1] Kerenidis and Prakash, 2017.

[2] Wiebe et al., 2012.

[3] В исходную версию алгоритма HHL были внесены некоторые улучшения и дополнения, которые изменили баланс между временем выполнения и другими параметрами. Здесь мы сосредоточимся на концептуально более простом исходном алгоритме.

На это можно возразить, что такое сравнение некорректно, поскольку традиционный метод сопряженных градиентов предоставляет полный набор решений, в отличие от квантового ответа, сгенерированного HHL. Вместо этого следовало бы сравнивать HHL с лучшими традиционными алгоритмами для определения производных статистик для решений систем линейных уравнений (сумма, среднее и т. д.), которые обладают зависимостями от n и k вида $O(n\sqrt{k})$, но HHL все равно обеспечивает экспоненциальное улучшение зависимости от n.

Было бы заманчиво сосредоточиться исключительно на масштабировании алгоритмов с размером задачи n, но другие параметры не менее важны. Хотя алгоритм HHL обладает впечатляющим экспоненциальным ускорением в отношении n, по общему быстродействию HHL уступает своим традиционным конкурентам, если принять во внимание задачи с плохим коэффициентом обусловленности или меньшей разреженностью (где играют важную роль k или s[1]). Быстродействие HHL также пострадает, если вы потребуете более высокой точности и уделите повышенное внимание параметру ε.

По этим причинам мы должны привести следующее предостережение:

> алгоритм HHL хорошо подходит для решения систем линейных уравнений, представленных разреженными, хорошо обусловленными матрицами.

Кроме того, поскольку HHL использует примитив квантового моделирования, следует принять во внимание все требования, специфические для применяемого метода квантового моделирования.

Мы постарались дать реалистичное представление об использовании HHL. А теперь разберемся, как работает этот алгоритм.

Что происходит внутри

Обоснование HHL зависит от одного конкретного метода нахождения обратной матрицы посредством ее собственного разложения. Каждая матрица обладает своими наборами собственных векторов и собственных значений. Так как эта глава посвящена машинному обучению, предполагается, что читатель уже отчасти знаком с этой концепцией. Если же сталкиваетесь с ней впервые — собственные векторы и собственные значения фактически являются матричными эквивалентами собственных состояний и собствен-

[1] В результате недавних исследований из работы Childs, et al., 2015 удалось улучшить зависимость HHL от ε до poly(log(1/ε)).

ных фаз операций QPU, упоминавшихся при обсуждении оценки фазы в главе 8. В разделе «Оценка фазы на практике» также упоминалось о том, что состояние любого регистра QPU может рассматриваться как некоторая суперпозиция собственных состояний любых операций QPU. Из-за своей зависимости от квантового моделирования HHL ограничивается решением систем линейных уравнений, представленных *эрмитовыми* матрицами[1]. Для таких матриц истинен аналогичный факт, относящийся к их собственному разложению. Любой вектор, к которому должна применяться эрмитова матрица A, может быть выражен (то есть записан в виде линейной комбинации) собственными векторами A.

Например, рассмотрим эрмитову матрицу A и вектор z, показанные в формуле 13.3.

Формула 13.3. Матрица и вектор для представления собственного разложения

$$A = \begin{bmatrix} 2 & 2 \\ 2 & 3 \end{bmatrix}, \ \vec{z} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Два собственных вектора конкретной матрицы A – $\vec{v}_1$=[−0.7882, 0.615] и $\vec{v}_2$ = [−0.615, −0.788], с которыми связаны собственные значения $\lambda_1 = 0.438$ и $\lambda_2 = 4{,}56$ (это можно проверить, убедившись в том, что $A\vec{v}_1 = \lambda_1\vec{v}_1$ и $A\vec{v}_2 = \lambda_2\vec{v}_2$). Так как матрица A из нашего примера считается эрмитовой, $\vec{z}$ можно записать в виде комбинации ее собственных векторов: $\vec{z} = -0.788\vec{v}_1 - 0.615\vec{v}_2$. Также можно использовать запись вида $\vec{z}$ = [−0.788, −0.615] (при этом следует понимать, что компоненты выражаются в собственных векторах A).

Матрицу A также можно записать в ее *собственном базисе*[2]. Как выясняется, записанная таким образом матрица всегда является диагональной, а ее главная диагональ состоит из ее собственных значений. Для нашего примера матрица A записывается в ее собственном базисе так, как показано в формуле 13.4.

Формула 13.4. Запись матрицы в ее собственном базисе

$$A = \begin{bmatrix} 0.438 & 0 \\ 0 & 4.56 \end{bmatrix}.$$

[1] Как упоминалось в главе 9, не-эрмитова матрица $n \times n$ всегда может быть расширена до эрмитовой матрицы $2n \times 2n$.

[2] Под этим подразумевается нахождение элементов, которые должна содержать матрица A для правильного выражения векторов, выраженных в ее собственном базисе.

Выражение A в собственном базисе чрезвычайно удобно для нахождения обратной матрицы, поскольку обращение диагональной матрицы выполняется элементарно. Для этого достаточно обратить ненулевые значения на ее главной диагонали. Например, A^{-1} вычисляется так, как показано в формуле 13.5.

Формула 13.5. Обращение матрицы, записанной в ее собственном базисе

$$A^{-1} = \begin{bmatrix} \frac{1}{0.438} & 0 \\ 0 & \frac{1}{4.56} \end{bmatrix} = \begin{bmatrix} 2.281 & 0 \\ 0 & 0.219 \end{bmatrix}.$$

Конечно, следует заметить, что в результате мы получаем матрицу A^{-1}, *выраженную в собственном базисе A*. Обратную матрицу можно оставить в таком виде (если вы хотите применять ее к векторам, также выраженным в собственном базисе *A*), либо переписать ее в исходном базисе.

Итак, если вы можете найти собственные значения некоторой обобщенной матрицы A, то в этом случае вы можете определить $\vec{x} = A^{-1}\vec{b}$ так, как показано в формуле 13.6.

Формула 13.6. Общий метод определения обратной матрицы через ее собственное разложение

$$\vec{x} = \begin{bmatrix} \frac{1}{\lambda_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\lambda_n} \end{bmatrix} \begin{bmatrix} \tilde{b}_1 \\ \vdots \\ \tilde{b}_n \end{bmatrix} = \begin{bmatrix} \frac{1}{\lambda_1}\tilde{b}_1 \\ \vdots \\ \frac{1}{\lambda_n}\tilde{b}_n \end{bmatrix}.$$

Здесь $\lambda_1, ..., \lambda_n$ — собственные значения A, а $\tilde{b}_1, ..., \tilde{b}_n$ используются для обозначения составляющих $\vec{b}$, выраженных в собственном базисе *A*.

HHL реализует этот метод нахождения обратных матриц с использованием квантового параллелизма QPU. HHL реализует этот метод нахождения обратных матриц с использованием квантового параллелизма QPU. Выходной регистр HHL содержит амплитудное кодирование вектора, приведенного в формуле 13.6, то есть амплитуда состояния $|i\rangle$ равна ~ $\tilde{b}_i / \lambda_i$. Схема на рис. 13.2 показывает, что находится внутри блока HHL на рис. 13.1; по ней можно понять, как HHL использует знакомые примитивы QPU для получения выходного состояния из листинга 13.6.

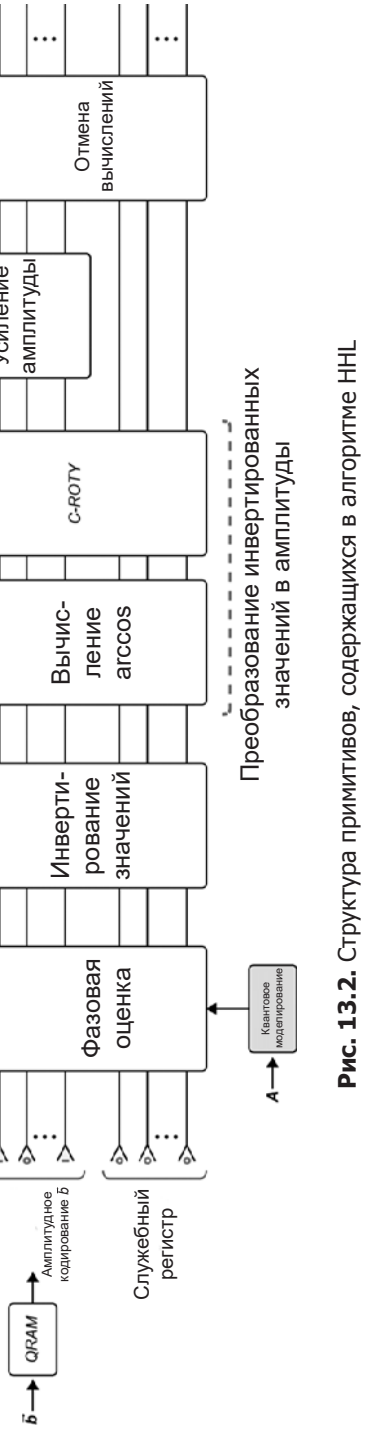


Рис. 13.2. Структура примитивов, содержащихся в алгоритме HHL

Хотя на рис. 13.2 это явно не обозначено, HHL также требует, чтобы другие входные регистры содержали определенные параметры конфигурации, необходимые для квантового моделирования.

Разберем каждый из составляющих примитивов на рис. 13.2.

1. Квантовое моделирование, QRAM и оценка фазы. Мы уже знаем, что примитив оценки фазы может эффективно находить собственные состояния и собственные фазы операции QPU. Резонно предположить, что это обстоятельство поможет использовать собственное разложение для обращения матрицы — и это действительно так!

На рис. 13.2 показано, что все начинается с использования QRAM для получения регистра с комплексного амплитудного кодирования b и квантового моделирования для получения операции QPU, представляющей A. Затем оба ресурса передаются примитиву оценки фазы, как показано на рис. 13.3.

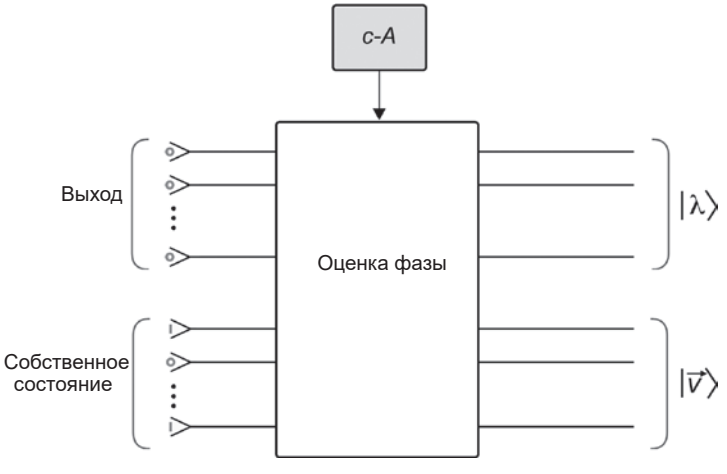


Рис. 13.3. Примитив оценки фазы

Рисунок 13.3 напоминает о том, что два входных регистра передаются примитиву оценки фазы. Нижний регистр собственного состояния получает входные данные, которые задают собственное состояние операции QPU, для которой вы хотели бы получить собственную фазу. Верхний выходной регистр (инициализированный в состоянии $|0\rangle$) производит представление собственной фазы, которая в нашем случае является собственным значением A.

Таким образом, оценка фазы дает собственное значение A. Но как получить все n собственных значений? Выполнение оценки фазы n раз сократит время выполнения HHL до $O(n)$ — не лучше, чем у традиционных аналогов. В регистре собственного состояния *можно* ввести равномерную суперпозицию и вычислить собственные значения параллельно, получая их равномерную суперпозицию в выходном регистре. Но предположим, мы пойдем по несколько иному пути и вместо этого передадим амплитудное кодирование $\vec{b}$ в регистре собственного состояния. Это приведет к тому, что в выходном регистре будет содержаться суперпозиция собственных значений A, но находящаяся в состоянии квантовой запутанности с амплитудным кодированием $\vec{b}$, производимым в выходном регистре.

Для нас это будет намного полезнее, чем равномерная суперпозиция собственных значений, поскольку теперь каждое состояние $|\lambda_i\rangle$ в запутанном состоянии собственного состояния и выходных регистров имеет амплитуду b_i (благодаря регистру собственного состояния). Но это не совсем то, что нужно для получения решения из формулы 13.6. Состояние выходного регистра представляет собственные значения A. На самом деле нужно инвертировать эти собственные значения, и — вместо сохранения их в другом (запутанном) регистре — переместить их в значения, умножая *комплексные амплитуды* регистра собственного состояния. Это инвертирование и перемещение реализуются следующими шагами HHL.

2. Инвертирование значений. Второй примитив на рис. 13.2 инвертирует каждое значение λ_i, хранящееся в (запутанной) суперпозиции выходного регистра и регистра собственного состояния.

В конце этого шага в выходном регистре кодируется суперпозиция состояний $|\lambda_i\rangle$, все еще находящаяся в состоянии запутанности регистра собственного состояния. Для инвертирования числовых значений, закодированных в квантовом состоянии, используются арифметические примитивы, представленные в главе 5. Существует много способов построения алгоритма числового инвертирования QPU из этих примитивов. Один из возможных вариантов — использование метода Ньютона для аппроксимации. Какой бы способ ни был выбран, как было замечено в главе 12, с делением возникают проблемы. Эта простая на первый взгляд операция требует значительных дополнительных ресурсов по числу кубитов. Не только составляющие операции требуют служебных кубитов, но и работа с инвертированием обозначает, что нам придется кодировать числовые значения в представлении с фиксированной или плавающей точкой (а также обрабатывать переполнение и т. д.). Собственно, именно дополнительные затраты, необходимые

для этого шага, отчасти объясняют, почему полный пример кода даже простейшей реализации HHL в настоящее время выходит за рамки того, что мы могли бы здесь представить в разумных пределах[1].

Тем не менее в конце этого шага регистр собственного состояния будет содержать суперпозицию значений $1/\lambda_i$.

3. Перемещение инвертированных значений в амплитуды. Теперь необходимо переместить закодированные в состоянии инвертированные собственные значения A в *амплитуды* этого состояния. Не забудьте, что амплитудно-кодированное состояние $\vec{b}$ остается в квантовой запутанности со всем этим, так что перемещение инвертированных собственных значений в амплитуды состояния дает последнюю строку в формуле 13.6 — а следовательно, амплитудное кодирование вектора решения $|\vec{x}\rangle$.

Ключом к решению этой задачи является применение операции C-ROTY (то есть условной операции ROTY; см. главу 2). А если конкретнее, мы назначаем приемником этой условной операции новый одиночный служебный кубит (помеченный «служебный кубит ROTY» на рис. 13.2), изначально находящийся в состоянии $|0\rangle$. Возможно показать (хотя для этого придется прибегнуть к гораздо более объемистым математическим выкладкам), что если управлять ROTY по обратному косинусу значений $1/\lambda_i$, хранящихся в выходном регистре, то амплитуды всех частей запутанных регистров (выходного и собственного состояния), для которых служебный кубит ROTY находится в состоянии $|1\rangle$, даст в точности те множители $1/\lambda_i$, которые нас интересуют.

Таким образом, этот шаг алгоритма состоит из двух частей:

1. *Вычисление арккосинуса $1/\lambda_i$ в суперпозиции каждого состояния в выходном регистре*. Эта задача может быть решена при помощи базовых арифметических примитивов[2], хотя и снова со значительными дополнительными затратами в количестве дополнительных кубитов.
2. *Выполнение* C-ROTY между первым регистром и служебным кубитом ROTY (подготовленным в состоянии $|0\rangle$).

В итоге вы получите искомое состояние, если служебный кубит ROTY находится в состоянии $|1\rangle$. Как это можно проверить? Прочитать служебный ку-

---

[1] Есть нечто романтическое (и возможно, фундаментальное) в том факте, что попытка воспроизведения традиционных арифметических операций на QPU может требовать столь огромных посторонних затрат.

[2] На самом деле в вычисления нужно включить константу и вычислять $\arccos(C/\lambda_i)$ вместо $\arccos(1/\lambda_i)$. Так как мы не приводим полной реализации алгоритма HHL, эта часть будет опущена для простоты.

бит ROTY операцией READ и получить результат 1. К сожалению, это происходит только с определенной вероятностью. Чтобы повысить вероятность того, что будет получен требуемый результат 1, мы можем воспользоваться другим примитивом QPU, выполнив усиление амплитуды со служебным кубитом ROTY.

4. Усиление комплексной амплитуды. Усиление комплексной амплитуды позволяет повысить вероятность получения результата 1 при чтении однокубитного регистра ROTY операцией READ. Как следствие, это повышает вероятность того, что в регистре собственного состояния будут содержаться нужные амплитуды $\tilde{b}_i/\lambda_i$.

Если, несмотря на усиление комплексной амплитуды, операция READ для служебного кубита ROTY все равно дает нежелательный результат 0, состояния регистров необходимо уничтожить и выполнить весь алгоритм HHL с самого начала.

5. Отмена вычислений. Если предположить, что предыдущая операция READ завершилась успехом, регистр собственного состояния теперь содержит комплексное амплитудное кодирование решений $\vec{x}$. Тем не менее работа еще не завершена. Регистр собственного состояния остается запутанным не только с выходным регистром, но также со многими другими служебными кубитами, которые были представлены в процессе. Как упоминалось в главе 5, запутанность желаемого состояния с другими регистрами создает проблемы по нескольким причинам. По этой причине мы применяем процедуру отмены вычислений (также описанную в главе 5) для разрыва запутанности регистра собственного состояния.

Есть! Регистр собственного состояния теперь содержит распутанное комплексное амплитудное кодирование $\vec{x}$, готовое к использованию любым из способов, предложенных в начале этого раздела.

HHL — весьма сложное и нетривиальное применение QPU. Не огорчайтесь, если вам не удастся разобраться в нем с первой попытки; очень полезно увидеть, как наши примитивы QPU используются в более серьезном алгоритме QPU.

Квантовый анализ главных компонент

Квантовый анализ главных компонент (QPCA, Quantum Principal Component Analysis) представляет собой реализацию одноименной процедуры обработки данных для QPU. QPCA не только предоставляет потен-

циально более эффективный подход к этой широко применяемой задаче машинного обучения, но и может послужить структурным элементом для других приложений QML. Как и HHL, работа QPCA зависит от оборудования квантовой памяти (QRAM). Прежде чем описывать, как QPU расширяет возможности анализа главных компонент (PCA), сначала рассмотрим сам метод PCA.

Традиционный анализ главных компонент

PCA является неоценимым инструментом data science, машинного обучения и т. д. Эта процедура, часто используемая как этап предварительной обработки данных, может преобразовать входной набор признаков в новый набор без корреляции. Некоррелированные признаки, производимые PCA, могут упорядочиваться по величине дисперсии закодированных данных. Так как PCA сохраняет лишь часть новых признаков, этот метод часто применяется для *снижения размерности*. Сохранение лишь нескольких первых главных компонент позволяет сократить количество используемых признаков при сохранении как можно большего объема интересных закономерностей в данных.

Процесс анализа главных компонент также известен под другими названиями в разных дисциплинах — например, преобразование Карунена — Лоэве или преобразование Хотеллинга. Он также эквивалентен сингулярному разложению.

Распространенная геометрическая аналогия для объяснения PCA заключается в представлении m точек данных, каждая из которых описывается n признаками, как группы из m точек в n-мерном пространстве признаков. В этой аналогии PCA строит список из n направлений в пространстве признаков, упорядоченных таким образом, что на первом месте находится направление с наибольшей дисперсией в данных, на втором — направление со второй по величине дисперсией и т. д. Эти направления представляют так называемые главные компоненты данных. Часто они изображаются в простом двухмерном пространстве признаков (то есть $n = 2$), как показано на рис. 13.4.

Один из недостатков главных компонент, сгенерированных PCA как нового набора признаков, заключается в том, что они могут не иметь физической интерпретации. Но если вы в конечном итоге заинтересованы в построении моделей, имеющих наибольший предиктивный потенциал, это может и не быть серьезной проблемой.

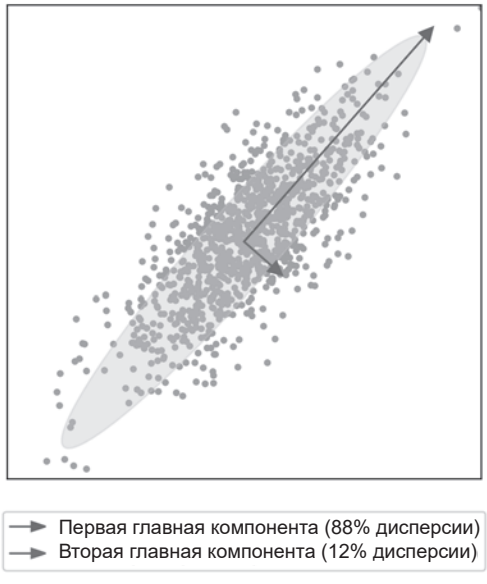


Рис. 13.4. Две главные компоненты 1000 точек данных в двухмерном пространстве признаков. Направления стрелок обозначают векторы главных компонент, тогда как их длины представляют дисперсию данных по этому направлению

Хотя геометрическое описание PCA полезно для формирования интуитивных представлений, для фактического вычисления главных компонент нам понадобится математическая процедура для их нахождения. Сначала вычисляется ковариационная матрица для заданного набора данных. Если данные расположены в матрице X размерами $m \times n$ (в которой каждая строка соответствует одной из m исходных точек данных, а каждый столбец содержит значения для одного из n различных признаков), то ковариационная матрица σ задается следующей формулой:

$$\sigma = \frac{1}{n-1} X^T X.$$

Удобно, что главные компоненты задаются нахождением собственного разложения этой ковариационной матрицы. Собственные векторы соответствуют направлениям главных компонент, а каждое ассоциированное собственное значение пропорционально дисперсии данных по этой главной компоненте. Если вы хотите использовать PCA для снижения размерности, собственные векторы можно переставить в порядке снижения собственного значения и выбрать только первые p в качестве нового, сокращенного набора признаков.

При применении PCA на практике важно *нормировать* данные перед вычислением ковариационной матрицы, так как процесс PCA чувствителен к масштабу данных. Один из общих методов нормализации заключается в нахождении отклонения каждого признака от своего среднего значения и масштабирования результата по стандартному отклонению данных.

Наиболее затратный в вычислительном отношении шаг PCA — выполнение собственного разложения ковариационной матрицы σ;. Как и с HHL, при необходимости определения собственных значений сразу вспоминается примитив оценки фазы в главе 8. Если действовать внимательно и осторожно, оценка фазы поможет вам выполнить PCA на QPU.

PCA с QPU

Казалось бы, для нахождения собственного разложения, необходимого для PCA, можно воспользоваться следующей процедурой:

1. Представить ковариационную матрицу данных в виде операции QPU.
2. Выполнить оценку фазы для этой операции QPU для определения ее собственных значений.

Однако у такого решения есть ряд проблем.

Проблема 1: квантовое моделирование с σ.

На первом шаге напрашивается предположение, что методы квантового моделирования помогут представить ковариационную матрицу в виде операции QPU по аналогии с матрицами, задействованными в HHL. К сожалению, ковариационные матрицы редко удовлетворяют требованиям методов квантового моделирования к разреженности, поэтому придется искать другой способ представления σ в форме операции QPU.

Проблема 2: оценка фазы имеет два входных регистра

Как на втором шаге этой процедуры узнать как собственные значения, так и собственные векторы? Вспомните по рис. 13.3, что оценка фазы имеет два входных регистра, один из которых должен использоваться для задания собственного состояния, для которого нужно узнать собственную фазу (а следовательно, и собственное значение). Но знание любых собственных векторов σ как раз и является частью задачи, которую требуется решить при помощи QPCA! Мы обошли эту вроде бы циклическую проблему при использовании оценки фазы в HHL, потому что могли использовать $|\vec{b}\rangle$ во входном регистре собственного состояния. Даже несмотря на то, что вы не знаете точно, какие собственные

состояния участвуют в суперпозиции $|\vec{b}\rangle$, оценка фазы действует на них параллельно — и вам даже не обязательно знать их. Нет ли каких-нибудь входных данных собственных состояний, которые можно было бы использовать для QPCA?

Примечательно, что всего один прием позволяет решить обе описанные проблемы. Суть в том, чтобы представить ковариационную матрицу σ в *регистре* QPU (а не в виде операции!). Вопрос о том, как этот прием решает описанные проблемы, весьма неочевиден (и требует серьезных математических обоснований), но мы приведем краткие пояснения.

Представление ковариационной матрицы в регистре QPU

Идея представления матрицы в регистре относительно нова — до сих пор мы прилагали определенные усилия для представления матриц в виде операций QPU.

Мы широко применяли круговую запись для описания регистров QPU, но время от времени намекали на то, что полноценное математическое описание требует использования векторов (с комплексными составляющими). Тем не менее, хотя мы тщательно избегали этой концепции, существует более общее математическое описание регистра QPU, использующее матрицу, которое называется *оператором плотности*[1]. Подробное описание операторов плотности выходит далеко за рамки этой книги (хотя в главе 14 приведены некоторые советы и ссылки на источники, по которым вы сможете начать их изучение). Для QPCA важно то, что при наличии QRAM-доступа к данным существует такой прием инициализации регистра QPU, что его описание оператором плотности в точности совпадает с ковариационной матрицей данных. Хотя такое кодирование матриц в описании оператора плотности регистра QPU обычно не приносит особой пользы, для QPCA оно позволяет решить обе проблемы, упомянутые выше.

Прием, используемый QPCA для представления ковариационной матрицы в форме оператора плотности, работает, потому что ковариационные матрицы всегда находятся в форме Грэма; это означает, что они могут быть записаны в форме V^TV для некоторой матрицы V. Для других матриц этот прием не столь полезен.

[1] Операторы плотности предоставляют более общее описание состояния регистра QPU, чем комплексные векторы (или круговая запись), потому что они допускают ситуации, при которых регистр не только находится в суперпозиции, но и существует некоторая статистическая неопределенность относительно этой суперпозиции. Квантовые состояния, содержащие статистическую неопределенность, обычно называются *смешанными состояниями*.

Решение проблемы 1

Хранение ковариационной матрицы в операторе плотности регистра QPU позволяет выполнить один прием, при котором за счет использования операции SWAP (см. главу 3) многократно выполняется некое подобное «частичного мини-SWAP» (частичного в смысле квантовой суперпозиции) между регистром, кодирующим σ, и вторым регистром. Хотя мы не станем углубляться в подробности того, как изменить SWAP для выполнения такой операции «мини-SWAP», выясняется, что ее эффективное применение приводит к квантовому моделированию реализации σ во втором регистре[1]. Именно этого результата мы бы обычно добивались с использованием более стандартных методов квантового моделирования[2], только этот метод «мини-SWAP» к генерированию операции QPU, представляющей σ, эффективно работает, даже если σ не является разреженной матрицей — важно лишь, чтобы она была низкоранговой. Несмотря на то что этот прием требует многократного применения операций SWAP (и, соответственно, многократного перекодирования σ в виде оператора плотности регистра QPU), он все равно оказывается эффективным.

Ранг матрицы определяется как число ее линейно-независимых столбцов. Так как столбцы σ представляют признаки данных, утверждение о том, что ковариационная матрица является низкоранговой, означает, что наши данные в действительности хорошо описываются меньшим подпространством полного пространства признаков. Количество признаков, которые потребуются для описания этого подпространства, равно рангу σ.

Решение проблемы 2

Как выясняется, представление σ в виде оператора плотности также дает именно то состояние, которое должно использоваться во входном регистре собственного состояния оценки фазы. Если вы так поступите, то в регистре собственного состояния на выходе примитива оценки фазы будет кодироваться собственный вектор σ (то есть одна из главных компонент), а в выходном регистре — связанное собственное значение (то есть величина дисперсии в направлении этой главной компоненты). Какая именно главная

[1] За более подробным описанием того, как строится эта операция и как она делает возможной такое квантовое моделирование, обращайтесь к Lloyd et al., 2013.

[2] Следует заметить, что метод «мини-SWAP» для представления матрицы в виде операции QPU не всегда будет лучше средств квантового моделирования. Он хорошо работает только из-за того, что ковариационные матрицы легко кодируются в операторе плотности регистра QPU благодаря тому, что они находятся в форме Грэма.

компонента, пара «собственное значение/собственный вектор», будет получена — определяется случайно, но вероятность получения заданной главной компоненты зависит от ее дисперсии.

Полная схема QPCA с предложенными исправлениями показана на рис. 13.5.

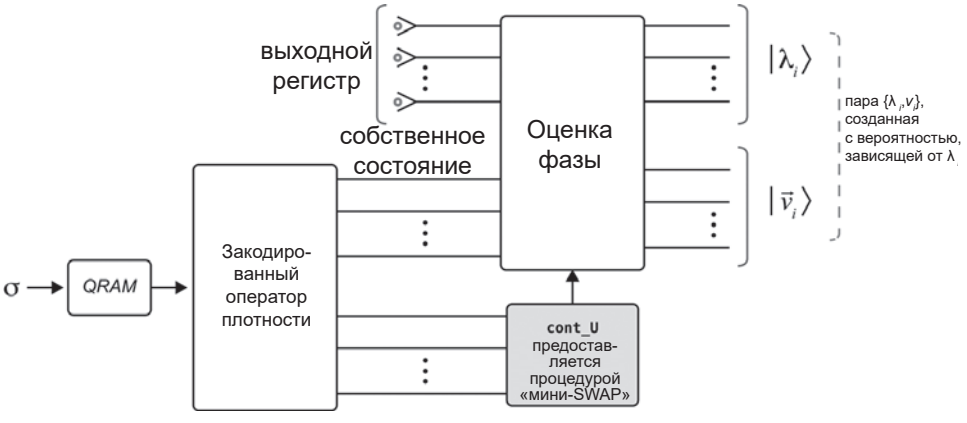


Рис. 13.5. Схема QPCA

На рис. 13.5 показано, как способность процедуры «мини-SWAP» по выполнению квантового моделирования с оператором плотности позволяет использовать ее на входе `cont_U` для оценки фазы. Учтите, что кодировщик оператора плотности должен выполняться многократно, а его вывод должен передаваться процедуре «мини-SWAP», которая предоставляет (условную) операцию QPU, необходимую для примитива оценки фазы (см. главу 8). Также следует заметить, что результат одного выполнения кодировщика оператора плотности вводится в регистр собственного состояния примитива оценки фазы.

Мы использовали термин «оператор плотности» для обозначения процесса, который позволяет представить ковариационную матрицу как оператор плотности регистра. Мы не будем углубляться в подробности работы кодировщика, но на рис. 13.5 в общих чертах показано, как эта возможность позволяет преобразовать примитив оценки фазы в задачу PCA.

Выход

После всего сказанного мы получаем алгоритм, возвращающий всю необходимую информацию об одной (случайно выбранной) главной

компоненте данных — вероятнее всего, это компонента с наибольшей дисперсией (именно эти компоненты чаще всего интересуют нас при использовании PCA). Но как главные компоненты, так и их дисперсии хранятся в регистрах QPU, и мы должны включить обычное предупреждение о том, что *решения выводятся в квантовой форме*. Тем не менее, как и в случае с HHL, возможно читать полезные производные свойства операцией READ. Более того, состояния регистров QPU можно передать другим приложениям QPU, где их квантовая природа скорее принесет пользу.

Быстродействие

Традиционные алгоритмы PCA обладают временем выполнения O(d), где d — количество признаков, по которым выполняется PCA.

С другой стороны, QPCA обладает временем выполнения $O(R \log d)$, где R — наименьший допустимый ранг приближения ковариационной матрицы (то есть наименьшее количество признаков, позволяющих представить данные с допустимым приближением). В тех случаях, когда $R < d$ (данные хорошо описываются низкоранговым приближением главных компонент), QPCA обеспечивает экспоненциальное улучшение времени выполнения по сравнению со своим традиционным аналогом.

Время выполнения QPCA подтверждает предшествующее утверждение о том, что улучшение обеспечивается только для низкоранговых ковариационных матриц. Это требование создает меньше ограничений, чем можно было бы ожидать. Обычно PCA применяется к данным, которые, как ожидается, хорошо подходят для такого низкорангового приближения — в противном случае пришлось бы рассмотреть их представление в виде подмножества главных компонент.

Квантовый метод опорных векторов

Квантовый метод опорных векторов (QSVM) демонстрирует, как QPU может использоваться для реализации контролируемых приложений машинного обучения. Как и в традиционной контролируемой модели, описанная здесь реализация QSVM должна пройти обучение на точках пространства признаков, обладающих заранее известной классификацией. Но QSVM присущи некоторые необычные ограничения.

Прежде всего QSVM требует, чтобы обучающие данные были доступны в суперпозиции с использованием QRAM. Кроме того, параметры, описы-

вающие обученную модель (используемую для дальнейшей классификации), строятся в комплексной амплитудно-кодированной форме в регистре QPU. Как обычно, это означает, что использование QSVM придется особенно тщательно планировать.

Традиционный метод опорных векторов

Метод опорных векторов (SVM) — популярная разновидность контролируемого классификатора с множеством разнообразных применений. Как и у других *линейных классификаторов*, в основе SVM лежит использование обучающих данных для поиска гиперплоскостей в пространстве признаков, разделяющих разные выходные классы в задаче. После того как SVM найдет такие гиперплоскости, новая точка данных в пространстве признаков может классифицироваться в зависимости от ее расположения относительно этих гиперплоскостей. Простой пример: допустим, имеются всего два признака (а следовательно, двухмерное пространство признаков); данные могут относиться всего к двум возможным выходным классам. В таком случае искомая гиперплоскость представляет собой линию (на рис. 13.6 оси x и y представляют значения двух признаков, а маркеры — обучающие данные двух выходных классов).

Как SVM вычисляет подходящую гиперплоскость по обучающим данным? С геометрической точки зрения можно представить процесс обучения SVM как рассмотрение двух параллельных гиперплоскостей, разделяющих классы обучающих данных в попытке максимизировать расстояние между этой парой гиперплоскостей. Если после этого выбрать третью гиперплоскость, находящуюся на половине расстояния между этими гиперплоскостями, в качестве классификатора вы получаете классификационную гиперплоскость с максимальным удалением от обучающих классов. Оптимизированная гиперплоскость, изученная SVM, с математической точки зрения описывается[1] вектором нормали $\vec{w}$ и смещением b. После получения этого описания гиперплоскости класс новой точки данных x определяется следующим правилом:

$$\text{class} = \text{sign}\left(\vec{w} \cdot \vec{x} - b\right).$$

Эта формула определяет на математическом уровне, с какой стороны гиперплоскости лежит новая точка.

[1] Эти параметры описывают гиперплоскость уравнением $\vec{w} \cdot \vec{x} - b = 0$, где $\vec{x}$ — вектор значений признаков.

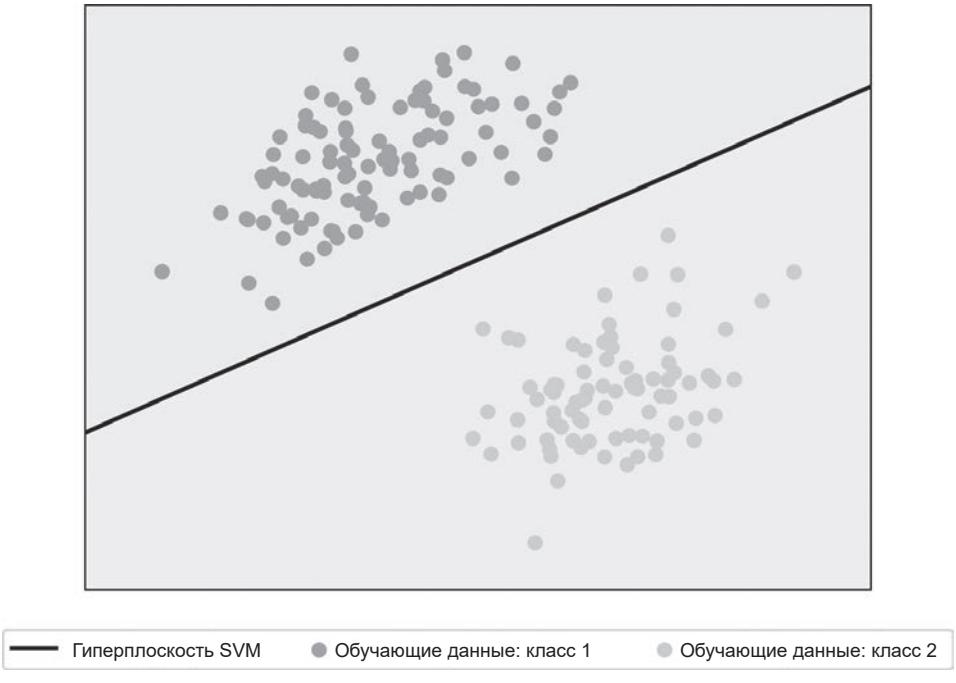


Рис. 13.6. Пример классификационной гиперплоскости, построенной методом SVM для задачи классификации с двумя классами и двумя признаками

Возможно, это описание нахождения оптимальных гиперплоскостей более наглядно, но для вычислительных целей обычно рассматривается так называемая *двойственная формулировка* процесса обучения SVM[1]. Двойственная форма более удобна для описания QSVM; она требует решения задачи квадратичного программирования для множества параметров $\vec{\alpha} = [\alpha_1, ..., \alpha_m]$. Конкретно нахождение оптимальной гиперплоскости SVM эквивалентно нахождению значения $\vec{\alpha}$, максимизирующего выражение в формуле 13.7.

Формула 13.7. Двойственное описание оптимизационной задачи SVM

$$\sum_i \alpha_i y_i - \frac{1}{2}\sum_i \sum_j \alpha_i \vec{x}_i \cdot \vec{x}_j \alpha_j .$$

Здесь $\vec{x}_i$ — i-я точка обучающих данных в пространстве признаков, а y_i — связанный с ней известный класс. Множество скалярных произведений

[1] Задачи оптимизации часто имеют двойственную форму, с которой проще работать. Стоит заметить, что иногда такие двойственные формы имеют нетривиальные отличия от исходной (первичной) задачи оптимизации.

$\vec{x}_i \cdot \vec{x}_j = K_{ij}$ между точками обучающих данных играет особую роль при обобщении SVM (о чем будет кратко рассказано в следующем разделе) и часто объединяется в матрицу, называемую *ядерной матрицей* (kernel matrix).

Нахождение $\vec{\alpha}$, удовлетворяющей этому выражению с учетом ограничений $\sum_i \alpha_i = 0$ и $y_i \alpha_i \geq 0 \ \forall \ i$, дает информацию, необходимую для определения параметров оптимальной гиперплоскости $\vec{w}$ и b. Собственно, новые точки данных можно классифицировать непосредственно в контексте $\vec{\alpha}$ по формуле 13.8, где точка пересечения b также может быть вычислена по обучающим данным[1].

Формула 13.8. Правило классификации для SVM в двойственном представлении

$$sign\left(\sum_i \alpha_i \vec{x}_i \cdot \vec{x} - b\right).$$

Здесь для нашего последующего обсуждения *квантового* метода SVM принципиально то, что классификация новой точки данных $\vec{x}$ требует вычисления ее скалярного произведения с каждой точкой обучающих данных $\vec{x} \cdot \vec{x}_i$.

Мы не будем углубляться в подробный вывод или использование этих уравнений; вместо этого будет показано, как доступность QPU позволяет намного эффективнее выполнить необходимые для них вычисления.

Обобщения SVM

Прежде всего стоит заметить, что разновидность SVM, описанная выше, ограничивается некоторыми задачами классификации. Для начала наше обсуждение предполагало, что моделируемые данные являются *линейно разделимыми*; то есть что определенно существует гиперплоскость, которая полностью и однозначно разделяет данные из двух классов. Конечно, часто это может быть не так, и хотя линейное разделение данных все равно может предоставить хорошую подгонку, данные из разных классов

[1] Более того, значение b может быть определено по одной точке обучающих данных, лежащей на одной из двух параллельных гиперплоскостей, определяющих границу. Точки, лежащие на этих определяющих гиперплоскостях, называются опорными векторами.

могут отчасти перекрываться в пространстве признаков, как показано на рис. 13.7.

Для того чтобы справиться с этой возможностью, SVM вводит в процесс обучения так называемые *мягкие зазоры* (soft margins). Хотя здесь мы не будем подробно рассматривать эту тему, стоит заметить, что ускорение работы QPU, о котором будет сказано ниже, сохраняются для SVM с мягким зазором.

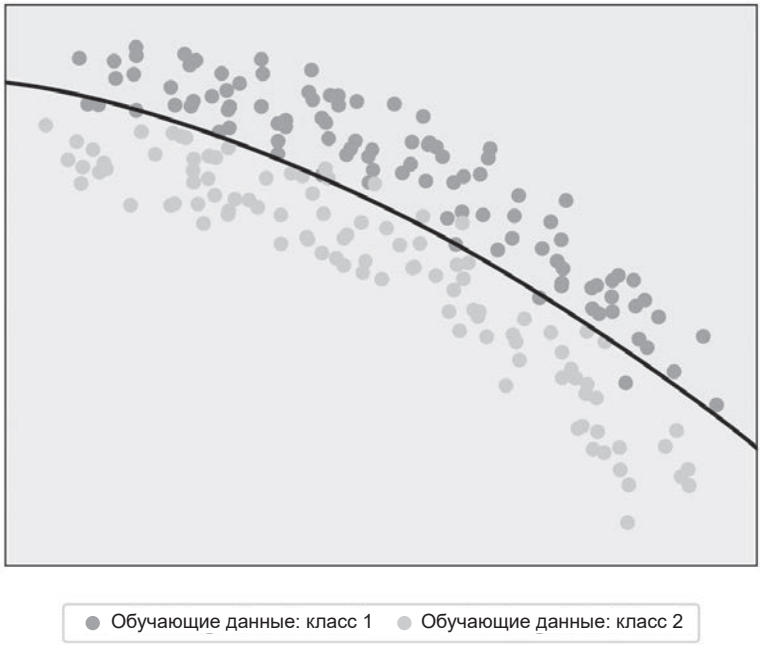


Рис. 13.7. Данные, которые не могут быть аппроксимированы линейной моделью SVM, — обучающие данные из разных классов перекрываются в пространстве признаков, а граница принятия решения очевидно нелинейна

Впрочем, даже SVM с мягким зазором ограничиваются линейным разделением данных. В некоторых случаях простая гиперплоскость плохо справляется с разделением классов. В такой ситуации можно попробовать внедрить пространство признаков в пространство еще более высокой размерности. Если процедура встраивания была выполнена достаточно тщательно, возможно, вам *удастся* найти в пространстве более высокой размерности

гиперплоскость, которая эффективно разделяет данные из разных классов. Иначе говоря, используется проекция $n + m$-мерной гиперплоскости в n-мерное пространство признаков, так как линейная $n + m$-мерная гиперплоскость может иметь нелинейную n-мерную проекцию. Такая нелинейная SVM изображена на рис. 13.7. Хотя это может показаться излишним усложнением процесса обучения SVM, оказывается, что такое нелинейное обобщение очень легко реализуется. Заменяя ядерную матрицу скалярных произведений из формулы 13.7 тщательно выбранной альтернативной ядерной матрицей, можно инкапсулировать нелинейные зазоры с более высокой размерностью. Это расширение часто называется обучением SVM на *нелинейном ядре*.

SVM с QPU

Существуют алгоритмы QPU, улучшающие быстродействие как обучения моделей SVM, так и классификации в обученной модели. Здесь мы только кратко обрисуем процесс применения QPU для эффективного обучения модели QSVM. Лучшие классические алгоритмы для обучения традиционных SVM обладают временем выполнения $O(\text{poly}(m, n))$, где m — количество точек обучающих данных, а n — количество признаков, описывающих каждую точку. С другой стороны, обучение QSVM выполняется с временем выполнения $O(\log(mn))$.

Использование QPU для обучения QSVM

Использование квантовых преимуществ для обучения SVM зависит от того, насколько вас устраивает полностью квантовая модель. Имеется в виду, что обученная модель QSVM представляет собой набор регистров QPU, содержащих *комплексные амплитудно-кодированные* параметры гиперплоскости $\vec{w}$ и b. Хотя эти параметры зафиксированы в суперпозициях, мы увидим, что они могут использоваться для классификации новых точек в пространстве признаков — при условии, что данные этих новых точек доступны в суперпозиции через QRAM.

Обучение SVM на множестве точек обучающих данных $\{\vec{x}_i, y_i\}_{i=1}^{m}$ требует нахождения оптимальных значений $\vec{\alpha}$, решающих задачу квадратичного программирования из формулы 13.7. Может показаться, что ускорить решение этой задачи при помощи QPU достаточно сложно. Тем не менее существует переформулировка задачи обучения SVM, называемая *методом опорных векторов по критерию наименьших квадратов* (LS-SVM), которая преобразует построение классификатора SVM в задачу оптимизации

методом наименьших квадратов[1]. Как следствие, для нахождения $\vec{\alpha}$, необходимой для двойственной формулировки SVM, мы теперь имеем линейную систему уравнений, решение которой задается матричным уравнением из формулы 13.9 (где y — вектор, содержащий классы обучающих данных).

Формула 13.9. Уравнение LS-SVM

$$\begin{bmatrix} b \\ \vec{a} \end{bmatrix} = F^{-1} \begin{bmatrix} 0 \\ \vec{y} \end{bmatrix}.$$

Эта формула уже больше похожа на то, что может быть ускорено средствами QPU с применением алгоритма HHL из раздела «Решение систем линейных уравнений». Матрица F строится из ядерной матрицы K скалярных произведений обучающих данных, как показано в формуле 13.10.

Формула 13.10. Построение матрицы F из ядерной матрицы

$$F = \begin{bmatrix} 0 & 1^T \\ 1 & K + \frac{1}{\gamma}\mathbf{1} \end{bmatrix}.$$

Здесь γ — вещественный гиперпараметр модели (который на практике будет определяться методом перекрестной проверки), $\mathbf{1}$ — единичная матрица, а 1 — обозначение вектора из m единиц. После того как матрица F была использована для получения значений $\vec{\alpha}$ и b, новые точки данных можно будет классифицировать при помощи LS-SVM (как это делалось со стандартной моделью SVM) по критерию из формулы 13.8.

Чтобы использовать алгоритм HHL для эффективного решения формулы 13.9 и получения амплитудного кодирования $|b\rangle$ и $|\vec{\alpha}\rangle$ в регистрах, требуется решение некоторых ключевых вопросов.

Вопрос 1: подходит ли F для HHL?

Другими словами, относится ли матрица F к типу, который может быть обращен алгоритмом HHL (то есть эрмитова, достаточно разреженная и т. д.)?

[1] Между моделями SVM и LS-SVM существуют тонкие различия, хотя в некоторых разумных условиях была доказана их эквивалентность — например, см. Ye and Xiong, 2007.

Вопрос 2: как применить F^{-1} к $[0, \vec{y}]$?

Если вы *можете* вычислить матрицу F^{-1} с использованием HHL, то как гарантировать, что она будет правильно воздействовать на вектор $[0, \vec{y}]$, как того требует формула 13.9?

Вопрос 3: как классифицировать данные?

Даже если вы ответите на вопросы 1 и 2, все равно нужно найти способ использования полученных квантовых представлений b и $\vec{\alpha}$ для обучения данных, которые появятся в будущем.

Ответим на каждый из этих вопросов поочередно, чтобы убедиться в том, что при обучении QSVM действительно можно воспользоваться алгоритмом HHL.

Вопрос 1: подходит ли F для HHL?

Как нетрудно увидеть, матрица F является эрмитовой, так что она потенциально может быть представлена в виде операции QPU с использованием методов квантового моделирования из главы 9. Как упоминалось ранее, принадлежность к категории эрмитовых матриц необходима, но недостаточна для того, чтобы матрица могла эффективно использоваться в квантовом моделировании. Тем не менее также возможно видеть, что матрица в форме F может быть разложена на сумму матриц, каждая из которых удовлетворяет всем требованиям методов квантового моделирования. Таким образом, выясняется, что квантовое моделирование может использоваться для нахождения операции QPU, представляющей F. Однако следует заметить, что нетривиальные элементы F состоят из скалярных произведений между точками обучающих данных. Для эффективного применения квантового моделирования с целью представления F в виде операции QPU также необходимо иметь доступ к обучающим данным через QRAM.

Вопрос 2: как применить F^{-1} к $[0, \vec{y}]$?

Если вы используете HHL для нахождения F^{-1}, то эту проблему можно решить на шаге оценки фазы алгоритма. Комплексное амплитудное кодирование вектора классов обучающих данных $|\vec{y}\rangle$ вводится в регистре собственного состояния оценки фазы. Здесь запись $|\vec{y}\rangle$ используется для обозначения состояния регистра QPU с амплитудно-кодированным вектором $[0, \vec{y}]$. Здесь снова предполагается, что классы обучающих данных доступны в QRAM. По той же логике, которая была описана при объ-

яснении HHL в разделе «Решение систем линейных уравнений», $|\vec{y}\rangle$ может рассматриваться как суперпозиция собственных состояний F. Как следствие, при рассмотрении алгоритма HHL оказывается, что $|F^{1}\vec{y}\rangle$ содержится в итоговом выходном регистре, что в точности совпадает с искомыми решениями: $|b, \vec{\alpha}\rangle$. Таким образом, ничего особенного делать не придется — HHL может вывести результат применения F^{-1} к нужному вектору точно так же, как это происходило при решении систем линейных уравнений на более общем уровне.

Вопрос 3: как классифицировать данные?

Предположим, вы получаете новую точку данных $\vec{x}$, которую нужно классифицировать обученной моделью QSVM. Вспомните, что «обученная модель QSVM» — это всего лишь доступ к состоянию $|b, \vec{\alpha}\rangle$. Классификация может выполняться эффективно при условии доступности к новой точке данных $\vec{x}$ в QRAM. Классификация новой точки данных требует вычисления знака по формуле 13.8. В свою очередь, для этого необходимо определить скалярные произведения $\vec{x}$ со всеми точками обучающих данных $\vec{x}_i$, взвешенными по параметрам двойственной гиперплоскости LS-SVM α_i. Вычисление скалярных произведений в суперпозиции и обращение к формуле 13.8 может происходить следующим образом: сначала обученное состояние LS-SVM $|b, \vec{\alpha}\rangle$ используется в адресном регистре запроса к памяти QRAM, содержащей обучающие данные. При этом вы получаете суперпозицию обучающих данных с амплитудами, содержащими α_i. В другом регистре выполняется запрос к содержимому памяти QRAM, содержащей новую точку данных $\vec{x}$.

При наличии обоих состояний можно выполнить специальную процедуру проверки обменом. Хотя мы не будем углубляться в подробности, эта процедура объединяет состояния, полученные от двух запросов QRAM, в запутанную суперпозицию, а затем выполняет специально сконструированную проверку обменом (см. главу 3). Напомним, что вы не только получаете информацию о том, равны или нет состояния двух регистров QPU — точная вероятность успеха p операции READ, задействованной в проверке обменом, зависит от *точности* двух состояний — количественной метрики их близости. Проверка обменом, которая используется в данном случае, специально строится таким образом, чтобы вероятность p чтения значения 1 отражала искомый знак из формулы 13.8. А именно, $p < 1/2$ для знака +1, и $p >= 1/2$ для знака –1. Повторяя проверку обме-

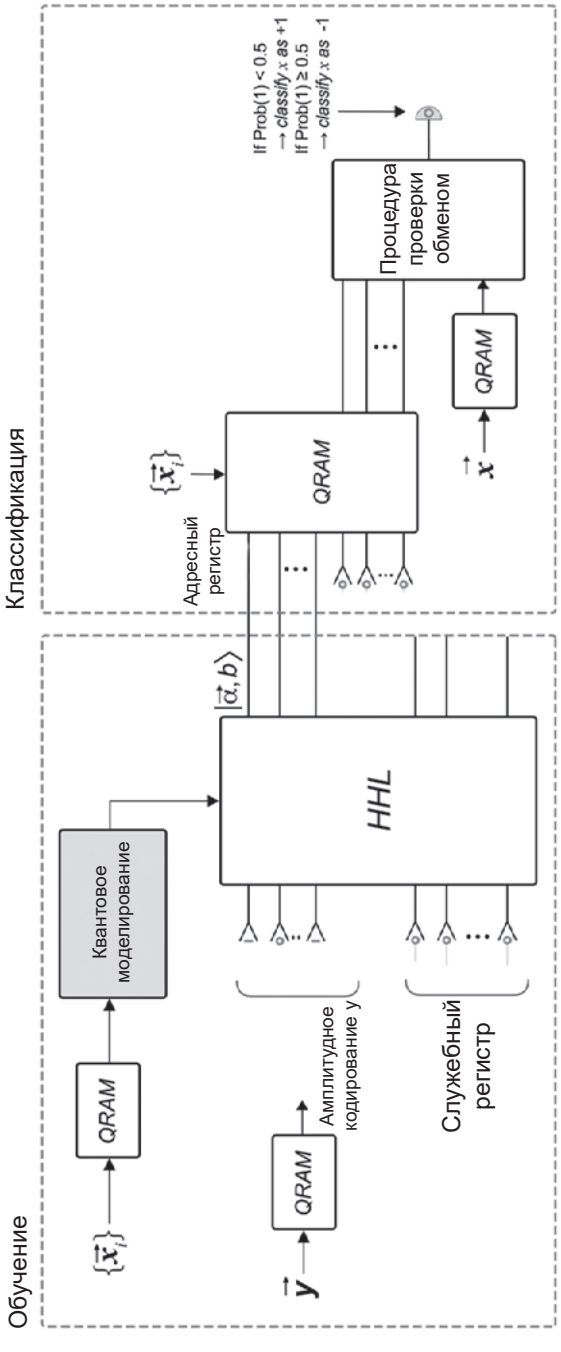


Рис. 13.8. Схема основных шагов обучения и классификации на базе модели QSVM

ном и подсчитывая результаты 0 и 1, можно оценить значение вероятности p до желаемой точности и классифицировать точку данных $\vec{x}$ соответствующим образом. Это позволяет провести обучение и использовать квантовую модель LS-SVM в соответствии со схемой, изображенной на рис. 13.8.

Не погружаясь в математические подробности (как у процедуры проверки обменом), рис. 13.8 дает только очень общее представление о том, как проходит обучение LS-SVM. Многие нюансы процедуры проверки обменом не показаны, хотя ключевые входные состояния выделены. Это дает некоторое представление о ключевых ролях некоторых примитивов QPU, представленных в книге.

Наша сводка QSVM также позволяет сделать важный вывод. Ее ключевым шагом было преобразование задачи SVM в формат, приспособленный для тех методов, для которых хорошо подходит QPU (в частности, обращение матриц). В этом воплощается центральный посыл этой книги — если руководствоваться информацией о том, *что* QPU умеет делать хорошо, знание предметной области позволяет находить новые применения QPU простым преобразованием существующих задач в QPU-совместимые формы.

Другие применения машинного обучения

Квантовое машинное обучение все еще остается исключительно динамически развивающейся областью исследований. В этой главе были представлены три канонических примера, но в области QML постоянно делаются новые открытия. В то же время квантовые методы машинного обучения становятся источником новых достижений в области традиционных методов — более того, за время написания этой книги все три применения QML, представленные в этой главе, вдохновили разработку традиционных алгоритмов с аналогичными улучшениями времени выполнения[1]. Это не должно поколебать вашей уверенности в потенциале QML; вряд эти результаты были бы получены без влияния со стороны аналогов QPU, что показывает, что приложения QML имеют далеко идущие и неожиданные последствия[2]. Также существует много других практических применений

[1] HHL: Chia et al., 2018; QPCA and QSVM: Tang, 2018.

[2] Пока не очевидно, насколько полезными окажутся эти алгоритмы, вдохновленные квантовыми аналогами, при реальном применении. См. пример у Arrazola et al., 2019.

QML, для нормального описания которых у нас просто не было места. К их числу относятся эффективные применения QPU для линейной регрессии[1], неконтролируемого обучения[2], машин Больцмана[3], полуопределенного программирования[4] и квантовых рекомендательных систем[5] (квантовые рекомендательные системы также вдохновили ряд усовершенствований в традиционных алгоритмах[6]).

[1] Chakraborty et al., 2018.
[2] Lloyd et al., 2013.
[3] Wiebe et al., 2014.
[4] Brandão et al., 2017.
[5] Kerenidis and Prakash, 2016.
[6] Tang, 2018.

ЧАСТЬ IV
Перспективы

14 Обзор литературы

Надеемся, вам понравилось возиться с вычислительными задачами, представленными в книге! Перед тем как закончить, мы кратко представим несколько тем, для которых не хватило здесь места, а также приведем ссылки на источники, в которых вы сможете больше узнать об этих и других темах квантового программирования. Мы не будем вдаваться в подробности; скорее постараемся связать то, что вы узнали к настоящему моменту, с материалом, выходящим за рамки книги. Пусть общие представления, которые вы получили, станут только первым шагом на вашем пути самостоятельных исследований квантового программирования!

От круговой записи к комплексным векторам

Запись $|x\rangle$, используемая в книге для обозначения состояний квантовых регистров, называется обозначениями бра-кет или иногда обозначениями Дирака в честь знаменитого физика, жившего в XX веке. В литературе по квантовым вычислениям именно эта запись (вместо круговой записи) используется для представления квантовых состояний. В главе 2 мы упоминали об эквивалентности двух систем записи, но об этом стоит сказать еще немного, чтобы направить вас на правильный путь. Обобщенная суперпозиция в однокубитном регистре может быть выражена в записи Дирака в форме $\alpha|0\rangle + \beta|1\rangle$, где α и β — *амплитуды* состояний, представленные в виде комплексных чисел, удовлетворяющих условию $\alpha^2 + \beta^2 = 1$. Магнитуда и относительная фаза каждого значения в круговой записи, которая использовалась в книге, определяются *модулем* и *аргументом* комплексного числа α и β соответственно. Вероятность результата операции READ для заданного двоичного выходного значения регистра QPU задается квадратом

модуля комплексного числа, описывающего амплитуду этого значения. Например, в предыдущем однокубитном случае $|\alpha|^2$ определяет вероятность чтения 0, а $|\beta|^2$ — вероятность чтения 1.

Комплексные векторы, описывающие состояния регистров QPU, обладают конкретными математическими свойствами — говорят, что они существуют в структуре, называемой *гильбертовым пространством*. Скорее всего, вам не понадобится много знать о гильбертовом пространстве, но вы часто будете слышать этот термин — чаще всего для обозначения совокупности возможных комплексных векторов, представляющих заданный регистр QPU.

В случае одиночных кубитов для параметризации α и β чаще всего используется выражение $\cos\theta|0\rangle+e^{i\varphi}\sin\theta|1\rangle$. В этом случае переменные θ и φ могут интерпретироваться как углы на сфере, во многих источниках называемой *сферой Блоха*. Как упоминалось в главе 2, сфера Блоха обеспечивает визуальное представление однокубитных состояний. Однако в отличие от круговой записи, сфера Блоха плохо подходит для визуализации регистров, состоящих из более чем одного кубита.

В книге также не упоминалась еще одна сложность, связанная с кубитными состояниями: речь идет о так называемых *смешанных состояниях*, которые математически представляются так называемыми *операторами плотности* (кратко упоминавшимися в главе 13). Они являются статистической смесью *чистых состояний*, с которыми мы имели дело при работе с квантовыми регистрами в этой книге (то есть способом описания кубита, когда вы не полностью уверены в том, в какой суперпозиции он находится[1]). До определенной степени смешанные состояния могут представляться и в круговой записи, но при использовании QPU с коррекцией ошибок (об этом позднее) чистых состояний будет вполне достаточно для изучения программирования QPU.

Так как визуализации многокубитных регистров редко встречаются в учебниках и справочниках, квантовые регистры чаще всего представляются исключительно комплексными векторами[2]; при этом длина вектора, необходимого для представления n кубитов, равна 2^n (по аналогии с тем, как количество кругов, необходимых для представления n кубитов в круговой записи, было равно 2^n). При записи комплексных амплитуд состоя-

[1] Данное уточнение относится к смешанным состояниям, а не чистым. — *Примеч. науч. ред.*

[2] Вспомните, о чем говорилось в главе 13: смешанные состояния представляются матрицами, называемыми «операторами плотности», вместо векторов.

ния n-кубитного регистра в столбцовом векторе комплексная амплитуда состояния $|00...0\rangle$ удобно размещается выше остальных возможных состояний, которые следуют далее по возрастанию двоичных значений.

Операции QPU описываются унитарными матрицами, оперирующими этими комплексными векторами. Матрицы записываются справа налево (в порядке, противоположном тому, в котором записывается квантовая круговая диаграмма — слева направо), так что первая матрица, оперирующая с комплексным вектором, соответствует первому (левому) вентилю в связанной круговой диаграмме. В формуле 14.1 приведен простой пример: вентиль NOT (в литературе он часто обозначается X) применяется к входному кубиту в состоянии $\alpha|0\rangle+\beta|1\rangle$. Вы видите, что он инвертирует значения α и β, как и предполагалось.

Формула 14.1. Воздействие вентиля NOT на кубит в стандартной записи комплексных векторов

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}.$$

Однокубитные вентили представляются матрицами 2×2, так как они преобразуют векторы с двумя элементами, соответствующими значениям однокубитного регистра 0 и 1. Двухкубитные вентили представляются матрицами 4×4; в общем случае n-кубитные вентили представляются матрицами $2^n \times 2^n$. На рис. 14.1 показаны матричные представления некоторых распространенных одно- и двухкубитных вентилей. Если вы понимаете, как работает матричное умножение, и хотите проверить свое понимание, попробуйте предсказать результат воздействия этих операторов на разные входные состояния и проверьте, совпадают ли ваши предсказания с результатами из круговой записи QCEngine.

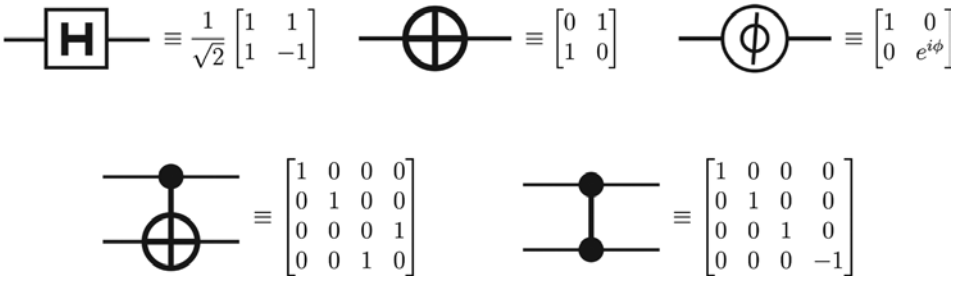


Рис. 14.1. Матричные представления распространенных одно- и двухкубитных вентилей

Некоторые нюансы и примечания по терминологии

В отношении терминологии, используемой в книге, существует ряд нюансов, о которых стоит упомянуть.

- В книге вычислительные технологии, существовавшие до появления квантовых, называются традиционными. Также традиционные двоичные вычисления, не работающие с квантовыми регистрами, часто называются классическими.
- Мы часто используем служебные кубиты для выполнения некоторых аспектов квантовых вычислений. Во многих ресурсах, посвященных квантовым вычислениям, они называются вспомогательными кубитами.
- В главе 2 был представлен вентиль PHASE, во входном параметре которого передается угол. В книге углы представлялись в *градусах* от 0° до 360°. В литературе по квантовым вычислениям углы также часто задаются в *радианах*. Радиан — угловая единица, соответствующая углу в центре круга, длина дуги которой равна радиусу круга. В следующей таблице приведены часто используемые углы в обеих единицах.

Градусы	0°	45°	90°	135°	180°	225°	270°	315°
Радианы	0	$\pi/4$	$\pi/2$	$3\pi/4$	π	$5\pi/4$	$3\pi/2$	$7\pi/4$

- В трех случаях вентиль PHASE обозначается специальным именем:

Угол (радианы)	$\pi/4$	$\pi/2$	π
Название	T	S	Z

- В главе 6 был представлен примитив AA. Этот примитив (а конкретно зеркальная операция) позволяет *усилить комплексную амплитуду* помеченных состояний в регистре с повышением вероятности чтения результата операцией READ из этого регистра. Хотя терминология выглядит достаточно прямолинейно, стоит заметить, что в научной литературе эти итерации обычно называются *итерациями Гровера*, тогда как выражение «усиление комплексной амплитуды» обычно предназначается для общего класса алгоритмов, использующих итерации Гровера для повышения вероятности успеха.

- Конкретная конфигурация регистра QPU (например, в какой суперпозиции или состоянии запутанности он должен находиться) обычно называется *состоянием регистра*. Эта терминология происходит от того факта, что конфигурация регистра в действительности описывается квантовым состоянием в математическом смысле, кратко упомянутом выше (даже если вы выбрали более удобную визуализацию регистра с использованием круговой записи).
- При описании N-кубитного регистра QPU в некоторой суперпозиции его 2^N возможных целочисленных значений все эти возможности (и связанные с ними комплексные амплитуды) часто называются *значениями в суперпозиции*. Эквивалентное и более распространенное выражение — «составляющая». Например, часто говорят о комплексной амплитуде «составляющей $|4\rangle$» в суперпозиции регистра QPU. Если рассматривать регистры QPU в контексте представления записи Дирака, эта терминология имеет смысл, так как $|4\rangle$ (и его комплексная амплитуда) в действительности является составляющей математического выражения. Мы избегали этого термина в книге просто из-за его неизбежных математических коннотаций.
- Операции QPU иногда называются *квантовыми вентилями* (по аналогии с логическими вентилями традиционных вычислений). Термины «операция QPU» и «квантовый вентиль» можно считать синонимами. Наборы квантовых вентилей образуют квантовые схемы.

Измерительный базис

В квантовых вычислениях существует распространенная концепция, которую мы старались не упоминать в книге, — это концепция *измерительного базиса*. Чтобы более полно понять концепцию измерений в квантовых вычислениях, необходимо взяться за изучение математических механизмов квантовой теории, а при столь ограниченном объеме нам не удастся сделать даже первые шаги. Эта книга была написана для того, чтобы дать читателю интуитивно понятный «вводный курс» в концепции, необходимый для программирования QPU, а за более глубоким обсуждением читателю следует обратиться к рекомендуемым ресурсам в конце главы. Тем не менее мы попробуем дать представление о том, как базовая идея базиса измерений связана с используемыми нами концепциями и терминологией.

Каждый раз, когда в книге использовалась операция `READ`, мы всегда предполагали, что она дает ответ 0 или 1. Вы видели, что эти два ответа соответ-

ствуют состояниям 0 и 1 в том смысле, что в этих состояниях всегда будет получен результат 0 или 1 соответственно. С технической точки зрения эти состояния являются *собственными состояниями*[1] операции PHASE(180) (также называемой вентилем *Z*). Каждый раз, когда мы выполняем READ в базисе *Z*, как неявно делалось в книге, комплексный вектор, описывающий регистр QPU, после READ окажется в одном из двух состояний. Регистр QPU *проецируется* в одно из этих состояний[2]. И хотя до настоящего момента мы думали только об изменениях в базисе *Z*, это не единственный вариант.

Разные базисы измерений можно сравнить с разными *вопросами*, которые мы задаем о состоянии QPU. Возможные вопросы, которые можно задавать при помощи квантовых операций READ, — в каком из собственных состояний для некоторых операций QPU находится система? Такой вопрос может показаться невероятно абстрактным, но в квантовой физике эти операции и их собственные состояния обладают физическим смыслом, а для понимания их точной природы потребуется более глубокое понимание физической основы. Так как до сих пор мы проводили измерения только по базису PHASE(180), в действительности мы всегда задавали вопрос: находится ли регистр QPU в собственном состоянии PHASE(180), соответствующем собственному значению +1, или же он находится в состоянии, соответствующему собственному значению –1? Даже если регистр QPU находится в суперпозиции этих возможностей, после операции READ он примет одну из них.

Выполнение операции READ в другом базисе означает вопрос о том, в каком из собственных состояний некоторой другой операции QPU — *U* — находится наш регистр QPU. После READ регистр QPU окажется в одном из собственных состояний *U* (то есть будет спроецирован в него).

Так как состояние регистра QPU может рассматриваться как суперпозиция собственных состояний любой операции QPU (как было показано в главе 13), запись представления состояния регистра QPU в собственном базисе некоторой операции *U* позволяет определить различные вероятности READ в базисе измерений *U*. Другой интересный аспект концепции базиса измерений заключается в том, что состояния, которые всегда обла-

---

[1] За информацией о собственных состояниях обращайтесь к главе 8.

[2] Термин «проекция» здесь используется в математическом смысле. Операторы проекции в математике «выбирают» некоторые векторы из линейной комбинации. В квантовой механике действие READ в базисе *Z* заключается в проецировании комплексного вектора, представляющего регистр QPU, на один из комплексных векторов, представляющих $|0\rangle$ или $|1\rangle$.

дают одинаковыми результатами измерений (с 100%-ной вероятностью) в одном базисе, могут не быть столь определенными в другом базисе — возможно, все результаты будут иметь только *вероятностный* характер. Например, вы видели, что при чтении состояния $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ существует 50%-ная вероятность получения 0 и 50%-ная вероятность получения 1 при измерении в базисе *Z*. Но если проводить измерения в базисе X(NOT), мы всегда будем получать 0; дело в том, что $|+\rangle$ оказывается собственным состоянием *X*, а следовательно, при рассмотрении в базисе измерений *X* состояние вообще не находится в суперпозиции.

Разложение и компиляция вентилей

Управляемые операции сыграли важную роль в книге. Возможно, время от времени у вас возникали вопросы относительно того, как реализовать эти операции. Особенно хотелось бы внести ясность в следующих случаях:

1. Управляемые операции, в которых на целевой кубит воздействует другая операция, отлична от NOT или PHASE.
2. Вентили, управляемые несколькими кубитами одновременно.

Ради компактности записи мы часто обозначали такие операции в виде одиночных блоков на диаграммах. Тем не менее в общем случае они не будут соответствовать машинным командам оборудования QPU, и их придется реализовать в виде совокупности более фундаментальных операций QPU.

К счастью, более сложные условные операции могут записываться в виде серии однокубитных и двухкубитных операций. На рис. 14.2 показано обобщенное разложение управляемой операции QPU (соответствующее обобщенной унитарной матрице в математике квантовых вычислений). Составляющие операции в этом разложении должны выбираться таким образом, чтобы *A*, *B* и *C* удовлетворяли условию $U = e^{i\alpha}AXBXC$ (где *X* — вентиль NOT), а выполнение всех трех операций сразу же друг за другом $A \cdot B \cdot C$ в целом не влияло на состояние регистра QPU.

Если вам удастся найти операции *A*, *B*, *C* и α, удовлетворяющие этим требованиям, то операция *U* может выполняться условно. Иногда может быть несколько возможных вариантов разложения условной операции по этой схеме.

Как насчет условных операций, управляемых сразу несколькими кубитами? Например, на рис. 14.3 показаны три разных разложения для реализаций вентиля CCNOT (Тоффоли), управляемые двумя кубитами.

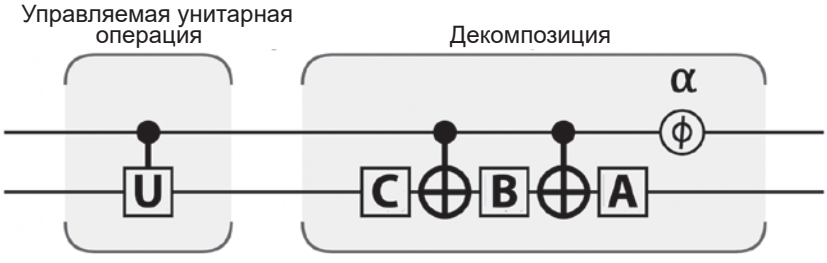


Рис. 14.2. Обобщенное разложение управляемой унитарной операции

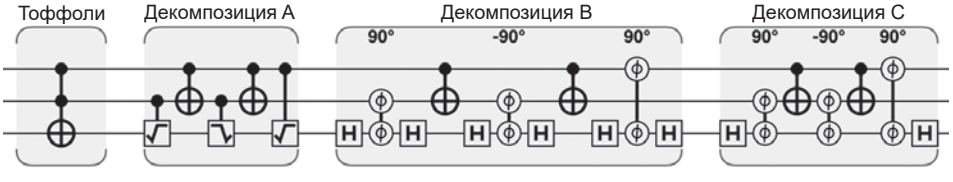


Рис. 14.3. Знакомый вентиль CCNOT можно разложить на более элементарные операции

Можно заметить, что все три разложения строятся по одной схеме. Это относится не только к CCNOT; любая условно-управляемая операция (то есть операция, управляемая двумя другими кубитами) будет иметь сходное разложение. На рис. 14.4 показан общий механизм реализации операции QPU, управляемой двумя кубитами, если вам удастся найти операцию QPU V, удовлетворяющую условию $V^2 = U$ (то есть двукратное применение V к регистру приводит к такому же результату, как при применении U).

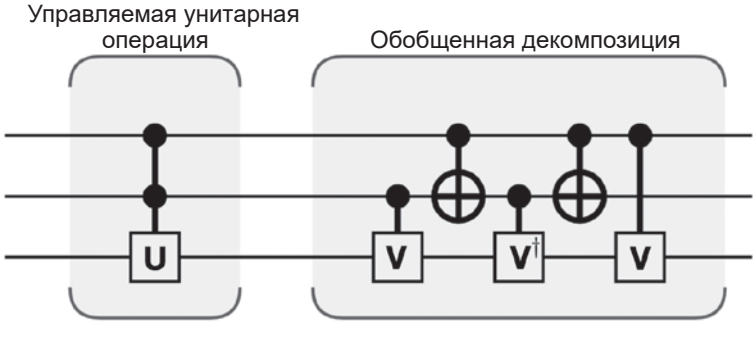


Рис. 14.4. Обобщенное разложение условно-управляемой унитарной операции

Если у вас возникнут проблемы с построением управляемых *V*-операций на рис. 14.4, вы всегда можете вернуться к рис. 14.3.

Найти оптимальные разложения алгоритмов QPU, выраженных простыми примитивами и операциями QPU, непросто. Дисциплина *квантовой компиляции* концентрируется на нахождении быстрых реализаций квантовых алгоритмов. Некоторые подробности, относящиеся к квантовой компиляции, будут упомянуты позднее в этой главе; примеры оптимизаций квантовой компиляции доступны по адресу http://oreilly-qc.github.io?p=14-GD.

Телепортация вентилей

В главе 4 протокол квантовой телепортации был использован для представления идей и обозначений, которые использовались позднее. Телепортация информации не находит применения в большинстве применений QPU, но с телепортацией операций QPU дело часто обстоит иначе. Это позволяет двум сторонам выполнить операцию с квантовым регистром, даже если ни одна из двух сторон не имеет доступа к состоянию и операции в одном месте. Как и в случае с протоколом телепортации, представленным в главе 4, этот прием требует использования пары запутанных кубитов.

Простой пример протокола телепортации вентилей доступен по адресу http://oreilly-qc.github.io?p=14-GT.

Зал славы QPU

В большей части этой книги мы воздерживались от отсылок к академическим источникам. Ниже приведен небольшой список источников, в которых были впервые представлены многие идеи и алгоритмы, которые более подробно обсуждались в книге:

- Feynman. Simulating Physics with Computers, 1982.
- Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer, 1985.
- Deutsch. Quantum Computational Networks, 1989.
- Shor. Algorithms for Quantum Computation: Discrete Log and Factoring, 1994.
- Barenco et al. Elementary gates for quantum computation, 1995.

- Grover. A fast quantum mechanical algorithm for database search, 1996.
- Brassard et al. Quantum Counting, 1998.
- Brassard et al. Quantum Amplitude Amplification and Estimation, 1998.
- Lloyd et al. Quantum Algorithm for Solving Linear Systems of Equations, 2009.

Другие ресурсы также приведены в списках книг и конспектов лекций в конце главы.

Гонка между квантовыми и традиционными компьютерами

При обсуждении различных практических применений QPU нас часто интересовало сравнение быстродействия недавно изученных алгоритмов QPU с их традиционными аналогами. И хотя мы проводили сравнение для каждого конкретного случая, также можно провести интересное сравнение возможностей квантовых и традиционных компьютеров в отношении вычислительной сложности. Вычислительная сложность задачи в компьютерной теории определяется (приближенно) ресурсами, необходимыми для выполнения самого эффективного алгоритма, решающего задачу. Теория вычислительной сложности изучает классификацию сложных задач в соответствии с их вычислительной сложностью[1]. Например, некоторые задачи относятся к классу P, что означает, что ресурсы, необходимые для поиска решения, растут в полиномиальной зависимости от размера задачи (пример — диагонализация матриц). К классу NP относятся задачи, имеющие правильное решение, которое может быть проверено за полиномиальное время, но при этом не гарантирована возможность нахождения решения таких задач за полиномиальное время. У этого класса есть «двойник» — co-NP; к нему относятся задачи, для которых неправильный ответ может быть проверен за полиномиальное время. Например, задача разложения числа на простые множители, описанная в главе 12, принадлежит одновременно к классам NP и co-NP — вы можете легко проверить, что пара чисел является (или не является) простыми множителями, но найти их не так просто. Пока остается открытой задача о равенстве классов P и NP, неоднократно упоминаемая в популярной культуре; того, кто сможет ее решить, ожидает приз в миллион долларов! Существует мнение — и притом обоснованное, —

[1] Полное описание 500+ классов доступно на сайте Complexity Zoo (https://complexityzoo.uwaterloo.ca/Complexity_Zoo).

что эти два класса не равны. NP-полный класс, упоминавшийся в главе 10, в определенном смысле соответствует самым сложным задачам класса NP. Любая задача в классе NP может быть сведена к одной из этих NP-полных задач, и если для любой NP-полной задачи удастся найти решение с полиномиальным временем, то все задачи из NP также станут решаемыми за полиномиальное время и перейдут в класс P.

Интерес, окружающий квантовые вычисления, в основном объясняется тем фактом, что они предположительно смогут сократить вычислительную сложность некоторых задач. Обратите внимание: речь идет не о всеобъемлющем ускорении любых традиционных вычислительных задач; в настоящее время квантовое ускорение распространяется только на отдельные классы алгоритмов. Сокращение может быть полиномиальным (например, переход от полинома высокого порядка к полиному более низкого порядка), как в случае с задачей 3-SAT из главы 10, или суперполиномиальным (например, переход от экспоненциальной сложности к полиномиальной), как в случае разложения на простые множители. Где квантовые компьютеры смогут обеспечить суперполиномиальное ускорение? Как выясняется, задачи, одновременно принадлежащие к классам NP и co-NP, становятся главными кандидатами для экспоненциального ускорения при использовании QPU. Более того, большинство алгоритмов, для которых было достигнуто такое ускорение, принадлежит пересечению этих двух классов.

Еще одно важное достижение, изначально упоминавшееся в главе 13, заслуживает повторного упоминания. Известно немало ситуаций, в которых после появления квантовых алгоритмов для некоторых задач появились алгоритмы полностью традиционные, но вдохновленные квантовыми аналогами. Таким образом, изучение и понимание квантовых алгоритмов также может привести к новым достижениям и в традиционных вычислениях!

Алгоритмы на базе оракулов

Три самых ранних квантовых алгоритма, у которых квантовая версия превосходила по скорости традиционные компьютеры, требовали вызова *оракула*. Оракул предоставляет информацию о переменной или функции без раскрытия самой переменной или функции. Задачей этих ранних алгоритмов было определение переменной или функции, используемой оракулом, с минимальным количеством вызовов. Необходимое количество вызова оракулов в таких задачах (и его масштабирование с размером задачи) обычно называется *сложностью запроса*.

Хотя эти алгоритмы не обеспечивали полезных вычислительных преимуществ, они сыграли критическую роль в разработке квантовых вычислений, так как помогали изучать возможности квантовых компьютеров, и со временем вдохновили исследователей (таких как Питер Шор) на разработку более полезных алгоритмов. Ввиду их педагогической и исторической важности мы кратко упомянем эти ранние алгоритмы, которые теперь известны по именам своих создателей.

Примеры кода QCEngine также доступны по адресу http://oreilly-qc.github.io, чтобы вы могли поэкспериментировать с этими первыми в своем роде квантовыми алгоритмами.

Алгоритм Дойча—Джозы

Оракул

Получает двоичную строку из n битов и выдает один бит, который является результатом применения функции f к двоичной строке. Гарантировано, что функция f является либо константной (в этом случае выходной бит всегда остается неизменным), либо сбалансированной (в этом случае количество 0 и 1 на выходе будет одинаковым).

Задача

Требуется решить с полной уверенностью, является ли функция f константной или сбалансированной, при минимальном количестве обращений к оракулу.

Сложность запроса

Традиционно для достижения полной уверенности в природе функции приходилось выдавать $2^{n-1} + 1$ запросов к оракулу. С QPU задача решается с нулевой вероятностью ошибки *всего одним квантовым запросом*!

Этот алгоритм доступен в по адресу http://oreilly-qc.github.io?p=14-DJ.

Задача Бернштейна—Вазирани

Оракул

Получает двоичную строку x, состоящую из n битов, и выдает одно двоичное число. Результат определяется по формуле $\sum_i x_i \cdot s_i$, где s — секретная строка, используемая оракулом.

Задача

Найти секретную строку s.

Сложность запроса

Традиционно требуется n запросов к оракулу, по одному для каждого входного бита. Тем не менее с QPU задача решается одним запросом.

Алгоритм доступен по адресу http://oreilly-qc.github.io?p=14-BV.

Задача Саймона

Оракул

Получает двоичную строку x, состоящую из n битов, и выдает одно целое число. Все возможные входные строки сопоставляются с секретной строкой s таким образом, что две строки (x, y) дают одинаковый результат в том и только в том случае, если $y = x \oplus s$ (где $\oplus$ — поразрядное суммирование по модулю 2).

Задача

Найти секретную строку s.

Сложность запроса

Традиционный детерминированный алгоритм требует как минимум $2^{n-1} + 1$ запросов к оракулу. С квантовым алгоритмом Саймона решение может быть найдено за количество вызовов, которое масштабируется линейно с ростом n (а не экспоненциально).

Алгоритм доступен по адресу http://oreilly-qc.github.io?p=14-S.

Языки квантового программирования

До настоящего момента мы не затрагивали тему языков квантового программирования — то есть языков программирования, разработанных с учетом особенностей квантовых вычислений. Применения и алгоритмы, рассмотренные в книге, описывались в базовых операциях QPU (по аналогии с традиционными двоичными логическими вентилями), которыми мы управляли из традиционного языка программирования. Этот подход был выбран по двум причинам. Во-первых, мы стремились дать вам практический опыт и помочь вам поэкспериментировать с возможностями QPU. На момент написания книги область квантового программирования все еще оставалась слишком молодой, и в ней не было никаких общепринятых стандартов. Во-вторых, многие доступные ресурсы по квантовым вычислениям, которые могут вывести вас за рамки этой книги (другие книги,

конспекты лекций и сетевые системы моделирования), излагают материал через призму тех же базовых операций QPU, занимающих центральное место в нашем обсуждении.

Одной из областей в сфере разработки стека квантового программирования, в которых в последнее время велись интенсивные разработки, была область *квантовой компиляции*. Из-за особенностей квантовых кодов исправления ошибок некоторые операции QPU (такие как `HAD`) намного проще реализуются в отказоустойчивой форме, чем другие (например, операция `PHASE(45)`, также называемая вентилем `T`). Задачей квантовой компиляции является поиск возможностей компиляции квантовых программ, которые учитывают такие ограничения реализации, но не оказывают отрицательного влияния на прирост скорости, обеспечиваемый QPU. В литературе по квантовой компиляции часто упоминается «метрика T» программы, то есть общее количество требуемых вентилей `T` с их затрудненной реализацией. Еще одной распространенной темой является подготовка и выделение так называемых магических состояний[1] — особых квантовых состояний, которые (при идеальной подготовке) позволяют реализовать проблемный вентиль `T`.

На момент написания книги исследования в области языков квантового программирования и инструментарии квантового программного обеспечения сильно выиграли бы от вклада экспертов по традиционным вычислениям — особенно в области отладки и проверки правильности. Ниже перечислены некоторые хорошие источники информации по теме.

- *Huang and Martonosi*. QDB: From Quantum Algorithms Towards Correct Quantum Programs, 2018.
- *Green et al. Quipper*. A Scalable Quantum Programming Language, 2013.
- *Altenkirch and Grattage*. A Functional Quantum Programming Language, 2005.
- *Svore*. Q#: Enabling Scalable Quantum Computing and Development with a High-Level Domain-Specific Language, 2018.
- *Hietala et al*. Verified Optimization in a Quantum Intermediate Representation, 2019.
- Qiskit: An open-source software development kit (SDK) for working with OpenQASM and the IBM Q quantum processors.

[1] В квантовых вычислениях определение «магический» является техническим термином. И немного волшебным тоже.

- *Aleksandrowicz et al.* Qiskit: An Open-source Framework for Quantum Computing, 2019.

Перспективы квантового моделирования

На момент написания книги (2019 г.) квантовое моделирование рассматривалось как самое перспективное направление квантовых вычислений. В настоящее время существуют предложения квантовых алгоритмов для решения задач квантовой химии, которые не решаются на классических компьютерах. Некоторые задачи, которые могут быть решены методами квантового моделирования:

Фиксация азота

Нахождение катализатора для преобразования азота в аммиак при комнатной температуре. Этот процесс может использоваться для снижения стоимости удобрений, что способствует решению проблемы нехватки продовольствия в странах третьего мира.

Сверхпроводимость при комнатной температуре

Нахождение материала, обладающего сверхпроводимостью при комнатной температуре. Такой материал позволит передавать энергию практически без потерь.

Катализатор для связывания углерода

Нахождение катализатора для поглощения углерода из атмосферы с целью сокращения выбросов CO_2 и замедления глобального потепления.

Очевидно, квантовые компьютеры обладают непревзойденным потенциалом социальных и экономических изменений по сравнению с большинством существующих технологий, что является одной из причин такого интереса к разработкам в этой области.

Исправление ошибок и устройства NISQ

Во всех обсуждениях операций QPU, примитивов и практических примеров в этой книге предполагалась (или моделировалась) доступность кубитов *с исправлением ошибок* (также называемых *логическими кубитами*). Как и в традиционных вычислениях, ошибки в регистрах и операциях могут быстро нарушить ход квантовых вычислений. В области квантовых кодов

с исправлением ошибок, которые должны противостоять этому эффекту, проводились серьезные исследования. Примечательным результатом в исследованиях квантового исправления ошибок стала *теорема о предельном значении*, которая показывает, что если частота ошибок QPU падает ниже определенного порога, то квантовые коды исправления ошибок позволят устранять ошибки ценой относительно малых затрат ресурсов. Применения QPU сохраняют свои преимущества перед традиционными алгоритмами только в таких условиях низкого шума; а следовательно, оно с большей вероятностью требует кубитов с исправлением ошибок.

Тем не менее во время написания книги существовала тенденция к поиску алгоритмов QPU, которые могут обеспечить более высокую скорость работы по сравнению с традиционными компьютерами даже на устройствах NISQ (Noisy Intermediate-Scale Quantum, то есть «шумные квантовые [компьютеры] промежуточного масштаба»). Предполагается, что эти устройства состоят из кубитов с высоким уровнем шума, не использующих исправления ошибок. Есть надежда, что могут существовать алгоритмы, которые по своей природе хорошо переносят шум QPU. В 2019 году не было известно о существовании таких алгоритмов, способных решать полезные задачи.

Что дальше?

В этой книге мы постарались дать читателю концептуальные инструменты и интуитивные представления о QPU, на основе которых он сможет приступить к дальнейшему исследованию увлекательной темы квантовых вычислений. С такой основой ссылки, приведенные в этом разделе, станут хорошей отправной точкой для того, чтобы вы могли вывести свое понимание QPU на новый уровень. Учтите, что в этих источниках часто используется более сложный уровень линейной алгебры и другие математические выкладки, которых мы старались избегать в тексте.

Книги

- *Nielsen and Chuang*. Quantum Computation and Quantum Information, 2011.
- *Mermin*. Quantum Computer Science: An Introduction, 2007.
- *Aaronson*. Quantum Computing Since Democritus, 2013.
- *Kitaev et al*. Classical and Quantum Computation, 2002.

- *Watrous*. The Theory of Quantum Information, 2018.
- *Kaye et al*. An Introduction to Quantum Computing, 2007.
- *Wilde*. Quantum Information Theory, 2013.

Конспекты лекций

- *Aaronson*, Quantum Information Science lecture notes (https://www.scottaaronson.com/qclec/combined.pdf).
- *Preskill*, lecture notes on Quantum Computation (http://www.theory.caltech.edu/people/preskill/ph229/#lecture).
- *Childs*, lecture notes on quantum algorithms (http://www.cs.umd.edu/~amchilds/qa/).
- *Watrous*, Quantum Computation, lecture notes (https://cs.uwaterloo.ca/~watrous/LectureNotes/CPSC519.Winter2006/all.pdf).

Сетевые источники информации

- *Vazirani*, Quantum Mechanics and Quantum Computation (https://www.edx.org/course/quantum-mechanics-and-quantum-computation).
- *Shor*, Quantum Computation (https://ocw.mit.edu/courses/mathematics/18-435j-quantum-computation-fall-2003/).
- Quantum Algorithm Zoo (http://quantumalgorithmzoo.org/).

Мерседес Химено-Сеговиа, Ник Хэрриган, Эрик Джонстон

Программирование квантовых компьютеров. Базовые алгоритмы и примеры кода

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Научный редактор	*М. Коробко*
Литературный редактор	*Д. Корунцева*
Художественный редактор	*В. Мостипан*
Корректоры	*Н. Сидорова, Г. Шкатова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214,
тел./факс: 208 80 01.

Подписано в печать 10.07.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 27,090. Тираж 700. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Крис Бернхард

КВАНТОВЫЕ ВЫЧИСЛЕНИЯ ДЛЯ НАСТОЯЩИХ АЙТИШНИКОВ

Квантовые вычисления часто упоминаются в новостях: Китай телепортировал кубит с Земли на спутник; алгоритм Шора поставил под угрозу ныне используемые методы шифрования; квантовое распределение ключей снова сделает шифрование надежным средством защиты; алгоритм Гровера увеличит скорость поиска данных.

Но что все это означает на самом деле? Как все это работает? Можно ли освоить эту тему без знания математики?

Нет, если вы хотите по-настоящему понять суть происходящего. Основные идеи берут начало в квантовой механике и часто противоречат здравому смыслу. Попытки описать их обычными словами обречены на провал, потому что эти явления не имеют отражения в обыденной жизни. Хуже того, словесные описания часто создают впечатление, что мы что-то поняли. Но на самом деле все не так плохо — нам не придется сильно углубляться в математику, достаточно того, что пытались вбить в наши головы в старших классах школы.

Квантовые вычисления — это удивительный сплав квантовой физики и информатики, объединяющий самые яркие идеи из физики двадцатого века и позволяющий по-новому взглянуть на компьютерные технологии.

КУПИТЬ

Владимир Силва

РАЗРАБОТКА С ИСПОЛЬЗОВАНИЕМ КВАНТОВЫХ КОМПЬЮТЕРОВ

Квантовые вычисления не просто меняют реальность! Совершенно новая отрасль рождается на наших глазах, чтобы создать немыслимое ранее и обесценить некоторые достижения прошлого.

В этой книге рассмотрены наиболее важные компоненты квантового компьютера: кубиты, логические вентили и квантовые схемы, а также объясняется отличие квантовой архитектуры от традиционной.

Вы сможете бесплатно экспериментировать с ними как в симуляторе, так и на реальном квантовом устройстве с применением IBM Q Experience. Вы узнаете, как выполняются квантовые вычисления с помощью QISKit (программный инструментарий для обработки квантовой информации), Python SDK и других API, в частности QASM. Наконец, вы изучите современные квантовые алгоритмы, реализующие запутанность, генерацию случайных чисел, линейный поиск, факторизацию целых чисел и др. Разберетесь с состояниями Белла, описывающими запутанность, алгоритмом Гровера для линейного поиска, алгоритмом Шора для факторизации целых чисел, алгоритмами оптимизации и многим другим.

Тим Рафгарден

СОВЕРШЕННЫЙ АЛГОРИТМ. ОСНОВЫ

Алгоритмы — это сердце и душа computer science. Без них не обойтись, они есть везде — от сетевой маршрутизации и расчетов по геномике до криптографии и машинного обучения. «Совершенный алгоритм» превратит вас в настоящего профи, который будет ставить задачи и мастерски их решать как в жизни, так и на собеседовании при приеме на работу в любую IT-компанию. В этой книге Тим Рафгарден — гуру алгоритмов — расскажет об асимптотическом анализе, нотации большое-О, алгоритмах «разделяй и властвуй», рандомизации, сортировки и отбора.

Книга «Совершенный алгоритм» адресована тем, у кого уже есть опыт программирования. Вы перейдете на новый уровень, чтобы увидеть общую картину, разобраться в низкоуровневых концепциях и математических нюансах. Познакомиться с дополнительными материалами и видеороликами автора (на английском языке) можно на сайте www.algorithmsilluminated.org.

Тим Рафгарден

СОВЕРШЕННЫЙ АЛГОРИТМ. ГРАФОВЫЕ АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Во второй книге Тим Рафгарден — гуру алгоритмов — расскажет о графовом поиске и его применении, алгоритме поиска кратчайшего пути, а также об использовании и реализации некоторых структур данных: куч, деревьев поиска, хеш-таблиц и фильтра Блума. Серия книг «Совершенный алгоритм» адресована тем, у кого уже есть опыт программирования, и основана на онлайн-курсах, которые регулярно проводятся с 2012 года.

Вы перейдете на новый уровень, чтобы увидеть общую картину, разобраться в низкоуровневых концепциях и математических нюансах. Познакомиться с дополнительными материалами и видеороликами автора (на английском языке) можно на сайте www.algorithmsilluminated.org.

КУПИТЬ

Тим Рафгарден

СОВЕРШЕННЫЙ АЛГОРИТМ. ЖАДНЫЕ АЛГОРИТМЫ И ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

В новой книге Тим Рафгарден расскажет о жадных алгоритмах (задача планирования, минимальные остовные деревья, кластеризация, коды Хаффмана) и динамическом программировании (задача о рюкзаке, выравнивание последовательностей, кратчайшие пути, оптимальные деревья поиска). Серия книг «Совершенный алгоритм» адресована тем, у кого уже есть опыт программирования, и основана на онлайн-курсах, которые регулярно проводятся с 2012 года. Вы перейдете на новый уровень, чтобы увидеть общую картину, разобраться в низкоуровневых концепциях и математических нюансах. Познакомиться с дополнительными материалами и видеороликами автора (на английском языке) можно на сайте www.algorithmsilluminated.org.

КУПИТЬ