aws

デベロッパーガイド

Amazon Braket

Amazon Braket: デベロッパーガイド

Amazon の商標およびトレードドレスは Amazon 以外の製品およびサービスに使用することはできません。また、お客様に誤解を与える可能性がある形式で、または Amazon の信用を損なう形式で使用することもできません。Amazon が所有していないその他のすべての商標は Amazon との提携、関連、支援関係の有無にかかわらず、それら該当する所有者の資産です。

Table of Contents

Amazon Braket とは? 1
仕組み 3
Amazon Braket 量子タスクフロー 4
サードパーティーのデータ処理 5
Amazon Braket の用語と概念 5
AWS Amazon Braket の用語とヒント 9
コストの追跡と削減 10
Amazon Braket QPUsの使用制限の設定 11
ほぼリアルタイムのコスト追跡 15
コスト削減のベストプラクティス 17
API リファレンスおよびリポジトリ 18
コアリポジトリ 19
プラグイン 19
サポートされているリージョンおよびサービス 20
リージョンとエンドポイント 24
開始方法 26
Amazon Braket を有効にする 26
前提条件 27
Amazon Braket を有効にする手順 27
Amazon Braket ノートブックインスタンスを作成する 28
(アドバンスト) CloudFormation を使用して Braket ノートブックを作成する 30
ステップ 1: SageMaker AI ライフサイクル設定スクリプトを作成する 31
ステップ 2: Amazon SageMaker AI が引き受ける IAM ロールを作成する 32
ステップ 3: プレフィックス amazon-braket- を持つ SageMaker AI ノートブックインスタンスを作成する 33
構築 34
最初の回路を構築する 34
最初の量子アルゴリズムの構築 39
SDK での回路の構築 40
回路の検査 55
結果タイプのリスト 57
エキスパートのアドバイスを受ける 63
(アドバンスト) OpenQASM 3.0 での回路の実行 64
OpenQASM 3.0 とは? 65

OpenQASM 3.0 を使用するタイミング 65
OpenQASM 3.0 の仕組み 65
前提条件 66
Braket はどのような OpenQASM 機能をサポートしていますか? 66
OpenQASM 3.0 量子タスクの例を作成して送信する 72
さまざまな Braket デバイスでの OpenQASM のサポート 75
ノイズをシミュレートする 85
Qubitの再配線 86
逐語的なコンパイル 87
Braket コンソール 88
その他のリソース 88
勾配の計算 88
特定の量子ビットの測定 89
(アドバンスト) 実験機能を探索する 90
QuEra Aquila でのローカルデチューニングへのアクセス 91
QuEra Aquila のトールなジオメトリへのアクセス 93
QuEra Aquila でのタイトなジオメトリへのアクセス 94
IQM デバイスの動的回路 95
(アドバンスト) Amazon Braket でのパルス制御 98
フレーム 98
ポート 99
波形 99
Hello Pulse の使用方法 100
パルスを使用したネイティブゲートへのアクセス 104
(アドバンスト) アナログハミルトニアンシミュレーション 106
Hello AHS: 初めてのアナログハミルトニアンシミュレーションを実行する 106
QuEra Aquila を使用してアナログプログラムを送信する 120
(アドバンスト) AWS Boto3 の使用 140
Amazon Braket Boto3 クライアントを有効にする 140
Boto3 と Braket SDK の AWS CLI プロファイルを設定する 144
テスト 146
シミュレーターへの量子タスクの送信 146
ローカル状態ベクトルシミュレーター (braket_sv) 147
ローカル密度マトリックスシミュレーター (braket_dm) 148
ローカル AHS シミュレーター (braket_ahs) 149
状態ベクトルシミュレーター (SV1) 149

密度マトリックスシミュレーター (DM1) ... 150
テンソルネットワークシミュレーター (TN1) ... 151
埋め込みシミュレーターについて ... 152
シミュレーターを比較する ... 153
Amazon Braket での量子タスクの例 ... 157
ローカルシミュレーターを使用した量子タスクのテスト ... 162
ローカル量子デバイスエミュレーター ... 164
ローカルエミュレーションの利点 ... 164
ローカルエミュレーターを作成する ... 165
実行 ... 167
QPU に量子タスクを送信する ... 168
AQT ... 169
IonQ ... 170
IQM ... 171
Rigetti ... 171
QuEra ... 172
例: QPU に量子タスクを送信する ... 172
コンパイルされた回路を検査する ... 176
複数のプログラムを実行する ... 176
プログラムセットとコストについて ... 178
量子タスクのバッチ処理とコストについて ... 178
量子タスクのバッチ処理と PennyLane ... 179
タスクバッチ処理とパラメータ化された回路 ... 179
量子タスクはいつ実行されますか? ... 180
QPU の可用性ウィンドウとステータス ... 181
キューの可視性 ... 181
E メールまたは SMS 通知の設定 ... 183
(アドバンスト) 予約の使用 ... 183
予約を作成する方法 ... 184
予約期間での量子タスクの実行 ... 185
予約期間中にハイブリッドジョブを実行する ... 189
予約期間が終了するとどうなるか ... 190
既存の予約をキャンセルまたは再スケジュールする ... 191
(アドバンスト) エラー緩和手法 ... 191
IonQ デバイスのエラー緩和手法 ... 191
Amazon Braket Hybrid Jobs ... 194

Amazon Braket Hybrid Jobs を使用すべき場合 195
Amazon Braket Hybrid Jobs でハイブリッドジョブを実行する 195
主要なコンセプト 197
入力 198
アウトプット 199
環境変数 200
ヘルパー関数 200
前提条件 201
ハイブリッドジョブの作成 204
作成して実行 205
結果をモニタリングする 208
結果を保存する 210
チェックポイントの使用 212
ローカルコードをハイブリッドジョブとして実行する 213
ハイブリッドジョブでの API の使用方法 222
ローカルモードでのハイブリッドジョブの作成およびデバッグ 225
ハイブリッドジョブのキャンセル 226
ハイブリッドジョブのカスタマイズ 228
アルゴリズムスクリプトの環境を定義する 228
ハイパーパラメータの使用方法 239
ハイブリッドジョブインスタンスの設定 241
パラメトリックコンパイルを使用したハイブリッドジョブの高速化 244
(アドバンスト) PennyLane と Amazon Braket 246
PennyLane と Amazon Braket 247
Amazon Braket のハイブリッドアルゴリズムのサンプルノートブック 248
PennyLane シミュレーターが埋め込まれたハイブリッドアルゴリズム 249
Amazon Braket シミュレーターを使用した PennyLane による随伴勾配 249
Hybrid Jobs と PennyLane を使用した QAOA アルゴリズムの実行 250
PennyLane 埋め込みシミュレーターを使用したハイブリッドワークロードの実行 253
(アドバンスト) CUDA-Q と Amazon Braket 259
NBI の CUDA-Q 259
Hybrid Jobs での CUDA-Q 260
トラブルシューティング 264
AccessDeniedException 264
CreateQuantumTask オペレーションを呼び出したときにエラーが発生しました (ValidationException)。 265

SDK 機能が動作しません ... 265
ServiceQuotaExceededException が原因でハイブリッドジョブが失敗する ... 265
ノートブックインスタンスでコンポーネントが機能しなくなった場合 ... 266
Python 3.12 のアップグレードのトラブルシューティング ... 266
概要: ... 267
一般的なエラーメッセージ ... 267
Braket マネージドノートブック ... 268
ハイブリッドジョブデコレータ ... 268
Bring-Your-Own-Container (BYOC) ... 269
Braket ノートブックインスタンスのアップグレード ... 270
OpenQASM のトラブルシューティング ... 270
インクルードステートメントのエラー ... 271
連続しない qubits のエラー ... 271
物理 qubits と仮想 qubits を混在させていることを示すエラー ... 272
同一プログラムで結果タイプをリクエストするとともに qubits を測定することによるエラー ... 272
Classical and qubit register limits exceeded error ... 273
ボックスの前に逐語的なプラグマがないことを示すエラー ... 273
逐語的なボックスにネイティブゲートがないことを示すエラー ... 273
逐語的なボックスに物理 qubits がないことを示すエラー ... 274
逐語的なプラグマに「braket」がないことを示すエラー ... 274
単一 qubits にはインデックスを作成できないことを示すエラー ... 274
2 つの qubit ゲートの物理 qubits が接続されていないことを示すエラー ... 275
LocalSimulator のサポートに関する警告 ... 275
セキュリティ ... 276
セキュリティの責任共有 ... 277
データ保護 ... 277
データ保持 ... 278
Amazon Braket へのアクセスを管理する ... 278
Amazon Braket のリソース ... 279
ノートブックとロール ... 279
AWS 管理ポリシー ... 281
特定のデバイスへのユーザーアクセスを制限する ... 285
特定のノートブックインスタンスへのユーザーアクセスを制限する ... 287
特定の S3 バケットへのユーザーアクセスを制限する ... 288
サービスリンクロール ... 289

コンプライアンス検証 290
インフラストラクチャセキュリティ 290
サードパーティーのセキュリティ 291
VPC エンドポイント (PrivateLink) 291
Amazon Braket VPC エンドポイントに関する考慮事項 292
Braket と PrivateLink を設定する 292
エンドポイントの作成に関する追加情報 294
Amazon VPC エンドポイントポリシーによるアクセスのコントロール 294
ログ記録とモニタリング 296
Amazon Braket SDK を使用して量子タスクを追跡する 297
Amazon Braket コンソールを使用した量子タスクのモニタリング 299
リソースのタグ付け 301
タグの使用 302
Amazon Braket でタグ付けがサポートされるリソース 302
Amazon Braket API を使用したタグ付け 303
タグ指定の制限 303
Amazon Braket でタグを管理する 304
Amazon Braket での AWS CLI タグ付けの例 305
EventBridge を使用した量子タスクのモニタリング 306
EventBridge での量子タスクステータスのモニタリング 306
Amazon Braket EventBridge イベントの例 308
CloudWatch によるメトリクスのモニタリング 309
Amazon Braket のメトリクスとディメンション 309
CloudTrail を使用した量子タスクのログ記録 310
CloudTrail 内の Amazon Braket 情報 310
Amazon Braket ログファイルエントリの概要 311
(アドバンスト) ログ記録 314
クォータ 317
追加のクォータと制限 358
ドキュメント履歴 359
..... cccIxxiii

Amazon Braket とは?

> **Tip**
>
> 量子コンピューティングの基礎について説明します AWS。Amazon Braket Digital Learning Plan に登録し、一連のラーニングコースとデジタル評価を完了して自身にデジタルバッジを獲得してください。

Amazon Braket は、研究者、科学者、開発者 AWS のサービス が量子コンピューティングを開始するのに役立つフルマネージド型です。量子コンピューティングは、量子力学の法則を利用して、新しい方法で情報を処理するため、古典的なコンピュータの手の届かない計算問題を解決する可能性を秘めています。

量子コンピューティングハードウェアへのアクセスを得ることは、費用がかかり不便になる場合があります。アクセスが制限されているため、アルゴリズムの実行、設計の最適化、テクノロジーの現在の状態の評価、および最大限の利益を得るためにリソースをいつ投資するかについての計画が困難になります。Braket は、これらの課題を克服するのに役立ちます。

Braket は、さまざまな量子コンピューティングテクノロジーへの単一のアクセスポイントを提供します。Braket を使用すると、次のことができます。

- 量子アルゴリズムとハイブリッドアルゴリズムを探索し、設計する。
- さまざまな量子回路シミュレーターでアルゴリズムをテストする。
- さまざまなタイプの量子コンピュータでアルゴリズムを実行する。
- PoC (概念実証アプリケーション) を作成する。

量子問題を定義し、それを解決するための量子コンピュータのプログラミングには、新しいスキルのセットが必要です。これらのスキルを習得するために、Braket は、量子アルゴリズムをシミュレートして実行するためのさまざまな環境を提供しています。要件に最適なアプローチを見つけることができ、ノートブックと呼ばれる一連のサンプル環境を使用してすぐに作業を開始できます。

Braket 開発には 3 つのステージがあります。

- ビルド (構築) - Braket は、ビルド (構築) を簡単に開始できるフルマネージド Jupyter Notebook 環境を提供します。Braket ノートブックには、サンプルアルゴリズム、リソース、および Amazon Braket SDK を含むデベロッパーツールがプリインストールされています。Amazon Braket SDK を

使用すると、量子アルゴリズムを構築し、単一のコード行を変更することで、さまざまな量子コンピュータやシミュレーターで量子アルゴリズムをテストして実行できます。

- テスト - Braket は、フルマネージドでハイパフォーマンスの量子回路シミュレーターへのアクセスを提供します。回路をテストして検証できます。Braket は、基礎となるすべてのソフトウェアコンポーネントと Amazon Elastic Compute Cloud (Amazon EC2) クラスターを処理し、古典的なハイパフォーマンスコンピューティング (HPC) インフラストラクチャで量子回路をシミュレートする負担を取り除きます。
- 実行 - Braket は、さまざまな種類の量子コンピューターへの安全なオンデマンドアクセスを提供します。、AQT、IonQ、IQMおよび のゲートベースの量子コンピュータとRigetti、QuEra のアナログハミルトニアンシミュレーターにアクセスできます。また、前払いの義務はなく、個々のプロバイダーを介したアクセスを確保する必要もありません。

量子コンピューティングと Braket について

量子コンピューティングは初期の発達段階にあります。現在、普遍的でフォールトトレラントな量子コンピュータは存在しないことを理解することが重要です。したがって、特定のタイプの量子ハードウェアは各ユースケースに適しているため、さまざまなコンピューティングハードウェアにアクセスできることが重要です。Braket は、サードパーティープロバイダーを通じて、さまざまなハードウェアを提供しています。

既存の量子ハードウェアはノイズによって制限され、エラーが生じます。業界はノイズの多い中間スケール量子 (NISQ) の時代です。NISQ 時代には、量子コンピューティングデバイスでのノイズが多すぎてショアのアルゴリズムやグローバーのアルゴリズムなどの純粋な量子アルゴリズムを維持できません。より優れた量子エラー訂正が利用可能になるまで、最も実用的な量子コンピューティングでは、ハイブリッドアルゴリズムを作成するために、古典的な (従来の) コンピューティングリソースと量子コンピュータを組み合わせる必要があります。Braket は、ハイブリッド量子アルゴリズムの操作を支援します。

ハイブリッド量子アルゴリズムでは、量子処理ユニット (QPU) が CPU のコプロセッサとして使用されるため、古典的なアルゴリズムにおける特定の計算が高速化されます。これらのアルゴリズムは、計算が古典コンピュータと量子コンピュータ間で移動する反復処理を利用します。例えば、化学、最適化、機械学習における量子コンピューティングの現在の応用は、ハイブリッド量子アルゴリズムの一種である変分量子アルゴリズムに基づいています。変分量子アルゴリズムでは、古典的な最適化ルーチンは、機械学習のトレーニングセットの誤差に基づいて、ニューラルネットワークの重みが反復的に調整されるのと同じ方法で、パラメータ化された量子回路のパラメータを反復的に調整します。Braket は、変分量子アルゴリズムを支援する PennyLane オープンソースソフトウェアライブラリへのアクセスを提供します。

量子コンピューティングは、次の 4 つの主要な領域での計算を牽引しています。

- 数論 — 素因数分解と暗号を含む (例えば、ショアのアルゴリズムは数論の計算を行うための主要な量子メソッドです)
- 最適化 — 制約充足、線形システムの解法、機械学習を含む。
- Oracle を用いた計算 — 検索、非表示のサブグループ、位数発見問題を含む (Oracle を用いた計算を行うための主要な量子メソッドの例は、グローバーのアルゴリズムです)。
- シミュレーション — 直接シミュレーション、ノット不変量、量子近似最適化アルゴリズム (QAOA) アプリケーションを含む。

これらの計算力テゴリの用途は、金融サービス、バイオテクノロジー、製造、医薬品などが挙げられます。Braket は、今日の多くの概念実証問題、および特定の実用的な問題に既に適用されている機能とノートブックの例を提供します。

このセクションの内容:

- Amazon Braket の仕組み
- Amazon Braket の用語と概念
- コストの追跡と削減
- Amazon Braket の API リファレンスおよびリポジトリ
- Amazon Braket がサポートするリージョンとデバイス

Amazon Braket の仕組み

> **Tip**
>
> 量子コンピューティングの基礎について説明します AWS。Amazon Braket Digital Learning Plan に登録し、一連のラーニングコースとデジタル評価を完了して自身にデジタルバッジを獲得してください。

Amazon Braket は、オンデマンド回路シミュレーターやさまざまなタイプの量子処理ユニット (QPUs) など、量子コンピューティングデバイスへのオンデマンドアクセスを提供します。Amazon Braket では、デバイスへのアトミックリクエストは量子タスクです。ゲートベースのデバイスの場合、このリクエストには量子回路 (測定手順とショット数を含む) およびその他のリクエストメタ

データが含まれます。アナログハミルトニアンシミュレーターの場合、量子タスクには量子レジスターの物理レイアウトと、操作フィールド間の時間とスペースに関する依存関係が含まれます。

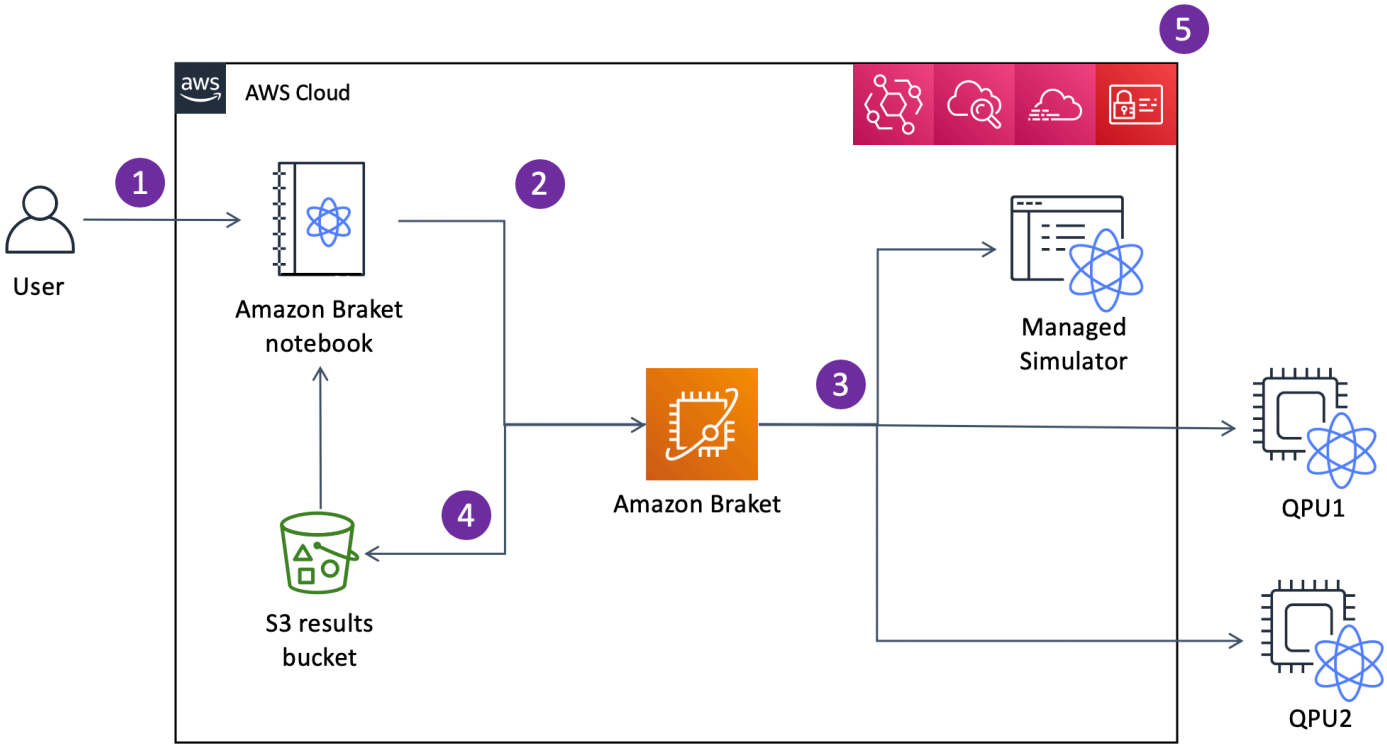
Braket Direct は、量子コンピューティングを探索する方法を拡張し AWS、研究とイノベーションを加速するプログラムです。さまざまな量子デバイスの専用の容量を予約したり、量子コンピューティングのスペシャリストと直接連携したり、あるいは IonQ の最新のトラップドイオンデバイスである Forte など、次世代の機能に早期にアクセスしたりできます。

このセクションでは、Amazon Braket で量子タスクを実行するフローの概略について学習します。

このセクションの内容:

- Amazon Braket 量子タスクフロー
- サードパーティーのデータ処理

Amazon Braket 量子タスクフロー



Jupyter ノートブックを使用すると、Amazon Amazon Braket Braket SDK を使用して量子タスクを定義、送信Amazon Braketモニタリングできます。量子回路は SDK で直接ビルドできます。ただし、アナログハミルトニアンシミュレーターでは、レジスタレイアウトと制御フィールド (1) を定義します。量子タスクを定義したら、実行するデバイスを選択して、量子タスクを Amazon Braket

API に送信することができます(2)。選択したデバイスに応じて、デバイスが利用可能になるまで量子タスクがキューに入れられ、タスクが実行のために QPU またはシミュレータに送信されます(3)。Amazon Braket では、QPUs、ローカルシミュレーター、埋め込みシミュレーターなど、サポートされているさまざまな量子デバイスにアクセスできます。

量子タスクを処理すると、Amazon Braket は結果を Amazon S3 バケットに返します。そこで、データは AWS アカウント (4) に保存されます。同時に、SDK はバックグラウンドで結果をポーリングし、量子タスク完了時に Jupyter Notebook にロードします。Amazon Braket コンソールの量子タスクページで、または Amazon Braket の オペレーションを使用して、量子タスクを表示および管理することもできますAPI。 `GetQuantumTask` Amazon Braket

Amazon Braket は、ユーザーアクセス管理、モニタリング AWS CloudTrail 、ログ記録、イベントベースの処理 AWS Identity and Access Management (5) のために、 (IAM)、Amazon CloudWatch、Amazon EventBridge と統合されています。

サードパーティーのデータ処理

QPU デバイスに送信された量子タスクは、サードパーティープロバイダーが運営する施設にある量子コンピュータで処理されます。Amazon Braket のセキュリティとサードパーティー処理の詳細については、「Amazon Braket ハードウェアプロバイダーのセキュリティ」を参照してください。

Amazon Braket の用語と概念

> **Tip**
>
> 量子コンピューティングの基礎について説明します AWS。Amazon Braket Digital Learning Plan に登録し、一連のラーニングコースとデジタル評価を完了して自身にデジタルバッジを獲得してください。

Braket では、次の概念と用語が使用されます。

アナログハミルトニアンシミュレーション

アナログハミルトニアンシミュレーション (AHS) は、多体系の時間依存量子力学を直接シミュレーションするための個別の量子コンピューティングパラダイムです。AHS では、ユーザーが時間依存のハミルトニアンを直接指定し、量子コンピュータは、このハミルトニアンの下での継続的な時間発展を直接エミュレートするようにチューニングされます。AHS デバイスは通常、専用

デバイスであり、ゲートベースのデバイスのような汎用量子コンピュータではありません。AHS デバイスは、シミュレートできるハミルトニアンのクラスに制限されます。ただし、これらのハミルトニアンはデバイスに元々実装されているため、AHS は、アルゴリズムを回路として作成してゲート操作を実装するために必要なオーバーヘッドに悩まされません。

Braket

私たちは量子力学における標準的な記法であるブラケット記法にちなんで Braket サービスと名付けました。量子系の状態を記述するために 1939 年に Paul Dirac によって導入され、ディラック記法とも呼ばれます。

Braket Direct

Braket Direct を使用すると、選択したさまざまな量子デバイスへの専用アクセスを予約したり、量子コンピューティングのスペシャリストと繋がってワークロードのガイダンスを受け取ったり、可用性が制限された新しい量子デバイスなどの次世代機能に早期にアクセスしたりできるようになります。

Braket ハイブリッドジョブ

Amazon Braket には、ハイブリッドアルゴリズムのフルマネージド実行を提供する Amazon Braket Hybrid Jobs と呼ばれる機能があります。Braket ハイブリッドジョブは、3 つのコンポーネントで構成されています。

1. スクリプト、Python モジュール、または Docker コンテナとして提供できるアルゴリズムの定義。
2. アルゴリズムを実行する Amazon EC2 に基づくハイブリッドジョブインスタンス。デフォルトは ml.m5.xlarge インスタンスです。
3. アルゴリズムの一部である量子タスクを実行する量子デバイス。1 つのジョブには、通常、多数の量子タスクから成る 1 つのコレクションが含まれます。

デバイス

Amazon Braket では、デバイスは量子タスクを実行できるバックエンドです。デバイスは QPU または量子回路シミュレーターである可能性があります。詳細については、「Amazon Braket supported devices」を参照してください。

エラー緩和

エラー緩和とは、複数の物理回路を実行し、その測定値を組み合わせて結果を改善することです。詳細については、「エラー緩和手法」を参照してください。

ゲートベースの量子コンピューティング

ゲートベースの量子コンピューティング (QC) (回路ベースの QC とも呼ばれる) では、計算は基本演算 (ゲート) に分割されます。ある種のゲートの集合は汎用的です。つまり、すべての計算がそれらのゲートの有限列として表現できます。ゲートは、量子回路の構成要素であり、古典的なデジタル回路の論理ゲートに似ています。

ゲートショットの制限

ゲートショットの制限とは、ショットあたりの合計ゲート数 (すべてのゲートタイプの合計) とタスクあたりのショット数に関するものです。数学的には、ゲートショット制限は次のように表現できます。

```
Gateshot limit = (Gate count per shot) * (Shot count per task)
```

ハミルトニアン

物理システムの量子力学は、そのハミルトニアンによって決定されます。ハミルトニアンは、システムの構成要素間の相互作用と外部駆動力による作用に関するすべての情報をエンコードしたものです。N 量子ビットシステムのハミルトニアンは、一般的に古典マシンにある複素数の 2^N x 2^N マトリックスとして表されます。量子デバイスでアナログハミルトニアンシミュレーションを実行することで、こういった大きなリソース要件を回避できます。

パルス

パルスは、量子ビットに送信される物理的な過渡信号です。パルスは、キャリア信号のサポートとして機能してハードウェアチャネルまたはポートにバインドされるフレーム内で再生される波形によって記述されます。カスタマーは、高周波正弦波キャリア信号を調整するアナログエンベロープを提供することで、独自のパルスを設計できます。フレームは、量子ビットの |0# と |1# のエネルギー準位間のエネルギー分離と共鳴するように頻繁に選択される周波数および位相によって一意に記述されます。したがって、ゲートは、事前に定義された形状と、その振幅、周波数、期間などのキャリブレーションされたパラメータを持つパルスとして作成されます。テンプレート波形でカバーされていないユースケースは、カスタム波形を介して有効になります。カスタム波形は、固定された物理サイクル時間で区切られた値のリストを提供することで、シングルサンプルの分解能で指定されます。

量子回路

量子回路は、ゲートベースの量子コンピュータ上の計算を定義する命令セットです。量子回路は、量子ゲート (qubit レジスタ上の可逆変換) と測定命令のシーケンスです。

量子回路シミュレーター

量子回路シミュレーターは、古典的なコンピュータ上で動作し、量子回路の測定結果を計算するコンピュータプログラムです。一般的な回路では、量子シミュレーションのリソース要件は、シミュレーションする qubits の数とともに指数関数的に増加します。Braket は、マネージド (Braket API を介してアクセス) とローカル (Amazon Braket SDK の一部) 量子回路シミュレーターの両方へのアクセスを提供します。

量子コンピュータ

量子コンピュータとは、重ね合わせやもつれなどの量子力学現象を用いて計算を行う物理デバイスです。量子コンピューティング (QC) には、ゲートベースの QC をはじめとするさまざまなパラダイムがあります。

量子処理ユニット (QPU)

QPU は、量子タスクを実行できる物理量子コンピューティングデバイスです。QPU は、ゲートベースの QC をはじめとするさまざまな QC パラダイムをベースにすることができます。詳細については、「Amazon Braket supported devices」を参照してください。

QPU ネイティブゲート

QPU ネイティブゲートは、QPU 制御システムによって制御パルスに直接マッピングできます。ネイティブゲートは、さらにコンパイルしなくても QPU デバイス上で実行できます。QPU がサポートするゲートのサブセット。デバイスのネイティブゲートは、Amazon Braket コンソールの [デバイス] ページおよび Braket SDK から確認できます。

QPU がサポートするゲート

QPU がサポートするゲートは、QPU デバイスで受け入れられるゲートです。これらのゲートは QPU で直接実行されない場合があります。つまり、ネイティブゲートに分解する必要がある可能性があります。デバイスサポートされるゲートは、Amazon Braket コンソールの [デバイス] ページおよび Amazon Braket SDK から確認できます。

量子タスク

Braket では、量子タスクはデバイスへのアトミックリクエストです。ゲートベースの QC デバイスの場合、これには、量子回路 (測定命令と shots 数を含む) 、およびその他の要求メタデータが含まれます。量子タスクを作成するには、Amazon Braket SDK を介するか、CreateQuantumTask API オペレーションを直接使用します。量子タスクを作成した後、リクエストされたデバイスが使用可能になるまで、タスクはキューに入れられます。量子タスクは、Amazon Braket コンソールの [量子タスク] ページで表示するか、GetQuantumTask または SearchQuantumTasks API オペレーションを使用して表示できます。

Qubit

量子コンピュータにおける情報の基本単位は、古典コンピューティングのビットに非常に似ているため、qubit (量子ビット) と呼ばれます。qubitは、超伝導回路、あるいは個々のイオンや原子など、さまざまな物理的実装によって実現できる 2 つのレベルの量子システムです。他の qubit タイプは、光子、電子スピンまたは核スピン、またはよりエキゾチックな量子システムに基づいています。

Queue depth

Queue depth とは、特定のデバイスに対してキューに入れられた量子タスクとハイブリッドジョブの数のことです。デバイスの量子タスクとハイブリッドジョブキューの数を知るには、Braket Software Development Kit (SDK) または Amazon Braket Management Console を使用します。

1. タスクキューの深さは、通常の優先度で実行を待っている量子タスクの合計数のことです。
2. 優先度タスクキューの深さは、Amazon Braket Hybrid Jobs での実行を待機している送信済み量子タスクの合計数のことです。これらのタスクは、ハイブリッドジョブが開始されると、スタンドアロンタスクよりも優先されます。
3. ハイブリッドジョブキューの深さは、現在デバイスのキューに入っているハイブリッドジョブの合計数のことです。ハイブリッドジョブの一部として送信された Quantum tasks は、Priority Task Queue に集約され、高い優先度が付けられます。

Queue position

Queue positionとは、量子タスクまたはハイブリッドジョブの、各デバイスキュー内での現在の位置のことです。量子タスクまたはハイブリッドジョブのキュー位置を取得するには、Braket Software Development Kit (SDK) または Amazon Braket Management Console を使用します。

Shots

量子コンピューティングは本質的に確率論的であるため、正確な結果を得るためには、回路を複数回評価する必要があります。単一の回路の実行と測定は、ショットと呼ばれます。回路のショット (繰り返し実行) の数は、結果の望ましい精度に基づいて選択されます。

AWS Amazon Braket の用語とヒント

IAM ポリシー

IAM ポリシーは、 AWS のサービス のサービスとリソースに対する権限を許可または拒否するドキュメントです。IAM ポリシーを使用すると、リソースへのユーザーのアクセスレベルをカスタ

マイズできます。たとえば、 内のすべての Amazon S3 バケットへのアクセスをユーザーに許可したり AWS アカウント、特定のバケットのみにアクセスを許可したりできます。

- ベストプラクティス: セキュリティ原則に従う最小特権アクセス許可を付与する場合。この原則に従うことで、ユーザーまたはロールが量子タスクの実行に必要とされるのよりも広範なアクセス許可を持つのを防ぐことができます。例えば、従業員が特定のバケットのみにアクセスする必要がある場合は、従業員には、 AWS アカウント内のすべてのバケットへのアクセス権を付与するのではなく、IAM ポリシーでバケットを指定します。

IAM ロール

IAM ロールは、アクセス許可に一時的にアクセスするために引き受けることができるアイデンティティです。ユーザー、アプリケーション、またはサービスが IAM ロールを引き受ける前に、ロールを切り替えるためのアクセス許可を付与する必要があります。IAM ロールを引き受けると、以前のロールで持っていた以前のすべてのアクセス許可を放棄し、新しいロールのアクセス許可を引き受けます。

- ベストプラクティス: IAM ロールは、長期的にではなく、サービスまたはリソースへのアクセスを一時的に付与する必要がある状況に最適です。

Amazon S3 バケット

Amazon Simple Storage Service (Amazon S3) は、データをオブジェクトとしてバケットに保存 AWS のサービス できる です。 Amazon S3 バケットは、無制限のストレージスペースを提供します。Amazon S3 バケット内のオブジェクトの最大サイズは 5 TB です。画像、動画、テキストファイル、バックアップファイル、ウェブサイトのメディアファイル、アーカイブされたドキュメント、Braket 量子タスクの結果など、あらゆるタイプのファイルデータを Amazon S3 バケットにアップロードできます。

- ベストプラクティス: S3 バケットへのアクセスを制御するためのアクセス許可を設定できます。詳細については、Amazon S3 ドキュメントの「バケットポリシー」を参照してください。

コストの追跡と削減

> **Tip**
>
> 量子コンピューティングの基礎について説明します AWS。Amazon Braket Digital Learning Plan に登録し、一連のラーニングコースとデジタル評価を完了して自身にデジタルバッジを獲得してください。

Amazon Braket を使用すると、前払いの義務なしで、オンデマンドで量子コンピューティングリソースにアクセスできます。お支払いいただくのは、使用分の料金だけです。料金の詳細については、料金ページをご覧ください。

このセクションの内容:

- Amazon Braket QPUsの使用制限の設定
- ほぼリアルタイムのコスト追跡
- コスト削減のベストプラクティス

Amazon Braket QPUsの使用制限の設定

Amazon Braket の使用制限では、量子処理ユニット (QPUs。

支出制限の仕組み: Amazon Braket は累積支出を追跡し、設定された制限に対してすべてのタスク作成リクエストを検証します。タスクの推定コストが残りの使用制限を超えると、Amazon Braket は検証エラーでタスクを直ちに拒否します。オプションで、使用制限の期間を設定できます。期間を設定することで、指定された期間にのみタスクを送信できます。期間外に送信されたタスクは拒否されます。

オプトイン設計: コントロールを明示的に有効にしない限り、既存のワークフローは影響を受けません。使用制限を削除することで、すべての制限を削除できます。

> **Note**
>
> 使用制限は、オンデマンドおよびハイブリッドジョブの QPU タスクにのみ適用されます。シミュレーター、マネージドノートブック、ハイブリッドジョブ EC2 インスタンスコスト、および Braket Direct 予約は除外されます。すべての AWS のサービスで包括的なコスト管理を行うには、引き続き を使用しますAWS Budgets。

使用制限アクションのリスト

検索

次の AWS CLI コマンドを使用すると、特定の AWS リージョンと特定の Braket デバイスの使用制限を検索して一覧表示できます。

```
aws --region {device_region} braket search-spending-limits --filters
 name=deviceArn,operator=EQUAL,values={device_arn}
```

作成

次の AWS CLI コマンドを使用すると、特定のリージョンで指定された量子デバイスの新しい使用制限を作成できます。デバイスの使用制限が既に存在する場合、リクエストは拒否されます。

```
aws --region {device_region} braket create-spending-limit --device-arn {device_arn}
 --spending-limit {max_spend}
```

更新

次の AWS CLI コマンドを使用すると、既存の使用制限を新しい最大使用値に更新できます。現在の支出とキューに入れられた支出の合計が、リクエストされた新しい最大支出をすでに上回っている場合、リクエストは拒否されます。

```
aws --region {device_region} braket update-spending-limit --spending-limit-arn
 {spending_limit_arn} --spending-limit {new_max_spend}
```

上記の例のように、新しい最大支出の代わりに、またはそれに加えて期間を指定できます。

削除

次の AWS CLI コマンドを使用すると、既存の使用制限を削除できます。

```
aws --region {device_region} braket delete-spending-limit --spending-limit-arn
 {spending_limit_arn}
```

上記の例のように、新しい最大支出の代わりに、またはそれに加えて期間を指定できます。

オプションですが、ベストプラクティスとして常に region パラメータを指定します。デバイスの とは異なるリージョンで実行されたコマンドは失敗するか、 の場合は誤った結果SearchSpendingLimitsを返します。

使用制限の使用方法のその他の例については、サンプルノートブックを参照してください。

タスク検証の仕組み

AWS アカウントがそれ以外の有効なCreateQuantumTaskリク工ストを送信すると、次のゲート動作が適用されます。注: 残りの予算は、使用制限とキューに入れられた支出と現在の支出の合計の差です。(次のセクションを参照)

- ケース 1: タスクデバイスの使用制限はありません: タスクが作成されます。
- ケース 2: ターゲットデバイスの使用制限があり、現在の時間は使用制限の期間内です。
  - タスクの推定コストが残りの予算以下である場合: CreateQuantumTask が成功すると、タスクが作成されます。
  - 推定コストが残りの予算よりも大きい場合: はCreateQuantumTask失敗し、タスクは作成されません。
- ケース 3: ターゲットデバイスの使用制限があり、現在の時刻が使用制限の期間外です。はCreateQuantumTask失敗し、タスクは作成されません。

残りの予算の計算方法

残りの予算は、支出制限と現在の支出とキューに入れられた支出の合計の差です。

使用制限があるデバイスに対してタスクが作成されると、キューに入れられた支出はタスクの推定コストによって増加します。このイベントは、次の表の最初の行に一覧表示されます。次の表は、タスクの進行状況に応じて、キューに入れられた支出と現在の支出がどうなるかを示しています。

古い量子タスクの状態	新しい量子タスクの状態	キューに入れられた支出に変更する	現在の支出に変更する
-	CREATED	推定コストの増加	変更なし
CREATED	QUEUED	変更なし	変更なし
いずれか	RUNNING	変更なし	変更なし
いずれか	キャンセル中	変更なし	変更なし
キャンセル中	CANCELLED	推定コストによる削減	チャネルなし

いずれか	FAILED	推定コストによる削減	変更なし
RUNNING	COMPLETED	推定コストによる削減	推定コストの増加 (部分的に完了したタスクに応じて調整)

エッジケース

Q: 使用制限を作成する場合、キューに入っているタスクはキューに入れられた使用にカウントされますか?

A: いいえ。作成済み、キューに入っている、または進行中のタスクは、新しく作成された使用制限のキューに入れられた支出にはカウントされません。

Q: 使用制限を小さくすると、更新によって、作成済み、キューに入っている、または進行中の量子タスクが早期に終了しますか?

A: いいえ。

Q: 使用制限の終了時間に達すると、作成済み、キューに入っている、または進行中の量子タスクが早期に終了しますか?

A: いいえ。作成済み、キューに入っている、または進行中のタスクは、使用制限のステータスに関係なく完了できます。

Q: 使用制限の不足と 0 USD の使用制限の違いは何ですか?

A: 使用制限がないと、制限なしで量子タスクを作成できます。ゼロドルの使用制限は、すべての量子タスクをブロックします。

Q: 過去または将来の使用制限は、すべての量子タスクの作成をブロックしますか?

A: はい。

Q: 使用制限を作成する場合、キューにすでにあるタスクの推定コストは、タスクが完了すると現在の支出にカウントされますか?

A: いいえ。支出制限がアクティブなときに送信されたタスクのみが、累積支出にカウントされます。

ほぼリアルタイムのコスト追跡

Braket SDK には、量子ワークロードにほぼリアルタイムのコスト追跡を追加するオプションが用意されています。各サンプルノートブックには、Braket の量子処理ユニット (QPU) とオンデマンドシミュレーターの最大コスト見積もりを提供するコスト追跡コードが含まれています。最大コスト見積もりは USD で表示され、クレジットや割引は含まれません。

> **Note**
>
> 表示される料金は、Amazon Braket シミュレーターと量子処理ユニット (QPU) のタスクの使用状況に基づいた見積もりです。表示される料金見積もりは、実際の料金とは異なる場合があります。料金見積もりには、割引やクレジットは勘案されていません。また、Amazon Elastic Compute Cloud (Amazon EC2) など他のサービスの利用状況によっては、追加の料金が発生する可能性もあります。

SV1 のコスト追跡

コスト追跡関数の使用方法を示すために、ベル状態回路を構築し、SV1 シミュレーターで実行します。まず、Braket SDK モジュールをインポートし、ベル状態を定義して、回路に `Tracker()` 関数を追加します。

```
#import any required modules
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.tracking import Tracker

#create our bell circuit
circ = Circuit().h(0).cnot(0,1)
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
with Tracker() as tracker:
    task = device.run(circ, shots=1000).result()

#Your results
print(task.measurement_counts)
```

```
Counter({'00': 500, '11': 500})
```

ノートブックを実行すると、ベル状態のシミュレーションで下記の出力が予想されます。トラッカー (コスト追跡) 関数により、送信されたショットの数、完了した量子タスク数、実行期間、請求された

実行期間、最大コストが USD で表示されます。実行時間はシミュレーションごとに異なる場合があります。

```
import datetime

tracker.quantum_tasks_statistics()
{'arn:aws:braket:::device/quantum-simulator/amazon/sv1':
 {'shots': 1000,
  'tasks': {'COMPLETED': 1},
  'execution_duration': datetime.timedelta(microseconds=4000),
  'billed_execution_duration': datetime.timedelta(seconds=3)}}

tracker.simulator_tasks_cost()
```

```
Decimal('0.0037500000')
```

コストトラッカーを使用して最大コストを設定する

コストトラッカーを使用して、プログラムの最大コストを設定できます。特定のプログラムに費やす金額に、最大しきい値を設定できるのです。そのように、コストトラッカーを使用して、実行コードにコスト制御ロジックを構築できます。次の例では、Rigetti QPU で同じ回路を実行し、コストを 1 USD に制限しています。コード内で回路の反復を 1 回実行するコストは 0.30 USD です。合計コストが 1 USD を超えるまで反復を繰り返すようにロジックが設定されています。したがって、このコードスニペットは次の反復が 1 USD を超えるまでの 3 回、実行されます。一般的に、プログラムは希望する最大コストに達するまで反復され続けます。この場合、反復は 3 回です。

```
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
with Tracker() as tracker:
    while tracker.qpu_tasks_cost() < 1:
        result = device.run(circ, shots=200).result()
print(tracker.quantum_tasks_statistics())
print(tracker.qpu_tasks_cost(), "USD")
```

```
{'arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3': {'shots': 600, 'tasks':
 {'COMPLETED': 3}}}
1.4400000000 USD
```

> **Note**
>
> コストトラッカーは、失敗した TN1 量子タスクの期間については追跡しません。TN1 のシミュレーション中にリハーサルが完了し、縮約ステップが失敗した場合、リハーサル料金はコストトラッカーに表示されません。

コスト削減のベストプラクティス

Amazon Braket を使用する際に次のベストプラクティスを考慮してください。時間を節約し、コストを最小限に抑え、一般的なエラーを回避します。

シミュレーターで検証する

- QPU で実行する前に、シミュレーターを使用して回路を検証します。これにより、QPU の使用料金が発生することなく回路をファインチューニングできます。
- シミュレーターで回路を実行した結果は、QPU で回路を実行した結果と同じではない可能性がありますが、シミュレーターを使用してコーディングエラーや構成の問題を特定できます。

特定のデバイスへのユーザーアクセスを制限する

- 権限のないユーザーが特定のデバイスで量子タスクを送信できないように制限を設定できます。アクセスを制限するには、IAM AWS を使用することをお勧めします。これを行う方法についての詳細は、アクセスの制限を参照してください。
- Amazon Braket デバイスへのユーザーアクセスを許可または制限する方法として、管理者アカウントを使用しないことをお勧めします。

請求アラームの設定

- 請求アラームを設定して、請求が事前設定された限度に達したときに通知を受けることもできます。アラームを設定する推奨方法は です AWS Budgets。カスタム予算を設定し、コストまたは使用量が予算額を超える可能性がある場合にアラートを受け取ることができます。情報は AWS Budgets で入手できます。

少ないショット数で TN1 量子タスクをテストする

- シミュレーターのコストは QPU より少なくなりますが、特定のシミュレーターは、高いショット数で量子タスクを実行するとコストが高くなる可能性があります。TN1 タスクは少ない shot 数でテストすることをお勧めします。Shot 数は、SV1 およびローカルシミュレータータスクのコストには影響しません。

量子タスクがないかすべてのリージョンをチェックする

- コンソールには、現在の量子タスクのみが表示されます AWS リージョン。提出された請求可能な量子タスクを探すときは、必ずすべてのリージョンをチェックしてください。
- サポートされるデバイスドキュメントページで、デバイスおよび関連するリージョンの一覧を表示できます。

Amazon Braket の API リファレンスおよびリポジトリ

> **Tip**
>
> 量子コンピューティングの基礎について説明します AWS。Amazon Braket Digital Learning Plan に登録し、一連のラーニングコースとデジタル評価を完了して自身にデジタルバッジを獲得してください。

Amazon Braket には、API、SDK、コマンドラインインターフェイスが用意されています。これらのインターフェイスを使用して、ノートブックインスタンスの作成と管理、モデルのトレーニングとデプロイを行うことができます。

- Amazon Braket Python SDK (推奨)
- Amazon Braket API リファレンス
- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for GoAPI Reference
- AWS SDK for Java
- AWS SDK for JavaScript
- AWS SDK for PHP
- AWS SDK for Python (Boto)

- AWS SDK for Ruby

また、コード例については、「Amazon Braket Tutorials」という GitHub リポジトリを参照してください。

- GitHub の「Braket Tutorials」

コアリポジトリ

Braket に使用されるキーパッケージを含むコアリポジトリのリストを次に示します。

- Braket Python SDK - Braket Python SDK を使用して、Python プログラミング言語で記述されたコードを Jupyter ノートブックに設定します。Jupyter ノートブックに設定したら、Braket デバイスとシミュレーターでコードを実行できます。
- Braket Schemas - Braket SDK と Braket サービス間の契約。
- Braket Default Simulator - Braket のすべてのローカル量子シミュレーター (状態ベクトルと密度マトリックス)。

プラグイン

次に、さまざまなデバイスやプログラミングツールとともに使用されるさまざまなプラグインがあります。これには、以下に示すように、Braket がサポートするプラグインや、サードパーティーがサポートするプラグインがあります。

Amazon Braket がサポートするプラグイン:

- Amazon Braket アルゴリズムライブラリ - Python で記述された構築済みの量子アルゴリズムのカタログ。これらのアルゴリズムはそのまま実行することも、より複雑なアルゴリズムを構築するための出発点として使用することもできます。
- Braket-PennyLane プラグイン - PennyLane を、Braket の QML フレームワークとして使用します。

サードパーティー (Braket チームがモニタリングし、支援):

- Qiskit-Braket プロバイダー - Qiskit SDK を使用して Braket リソースにアクセスできます。
- Braket-Julia SDK - (実験的) Braket SDK の Julia ネイティブバージョン。

Amazon Braket がサポートするリージョンとデバイス

> **Tip**
>
> 量子コンピューティングの基礎について説明します AWS。Amazon Braket Digital Learning Plan に登録し、一連のラーニングコースとデジタル評価を完了して自身にデジタルバッジを獲得してください。

Amazon Braket では、デバイスは量子タスクを実行するために呼び出すことができる量子処理ユニット (QPU) またはシミュレーターを表します。Amazon Braket は、、AQT、IonQ、IQM、QuEra および から QPU デバイスへのアクセスを提供しますRigetti。さらに、 はオンデマンド、ローカル、埋め込みシミュレーターへのアクセス AWS を提供します。埋め込みシミュレーターの詳細については、「埋め込みシミュレーターについて」を参照してください。

サポートされている量子ハードウェアプロバイダーの詳細については、「QPU に量子タスクを送信する」を参照してください。使用可能なシミュレーターの詳細については、「Submitting quantum tasks to simulators」を参照してください。次の表に、使用可能なデバイスとシミュレーターのリストを示します。

プロバイダー	デバイス名	パラダイム	タイプ	デバイス ARN	リージョン
AQT	IBEX-Q1	ゲートベース	QPU	arn:aws:braket:eu-north-1::device/qpu/aqt/Ibex-Q1	eu-north-1
IonQ	Forte-1	ゲートベース	QPU	arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1	us–east–1
IonQ	Forte-Enterprise-1	ゲートベース	QPU	arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1	us–east–1

プロバイダー	デバイス名	パラダイム	タイプ	デバイス ARN	リージョン
IQM	Garnet	ゲートベース	QPU	arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet	eu-north-1
IQM	Emerald	ゲートベース	QPU	arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald	eu-north-1
QuEra	Aquila	アナログハミルトニアンシミュレーション	QPU	arn:aws:braket:us-east-1::device/qpu/quera/Aquila	us–east–1
Rigetti	Ankaa-3	ゲートベース	QPU	arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3	us-west-1
AWS	braket_sv	ゲートベース	ローカルシミュレーター	該当なし (Braket SDK のローカルシミュレーター)	該当なし
AWS	braket_dm	ゲートベース	ローカルシミュレーター	該当なし (Braket SDK のローカルシミュレーター)	該当なし
AWS	braket_ahs	アナログハミルトニアンシミュレーション	ローカルシミュレーター	該当なし (Braket SDK のローカルシミュレーター)	該当なし

プロバイダー	デバイス名	パラダイム	タイプ	デバイス ARN	リージョン
AWS	SV1	ゲートベース	オンデマンドシミュレーター	arn:aws:braket:::device/quantum-simulator/amazon/sv1	us-east-1、us-west-1、us-west-2、eu-west-2
AWS	DM1	ゲートベース	オンデマンドシミュレーター	arn:aws:braket:::device/quantum-simulator/amazon/dm1	us-east-1、us-west-1、us-west-2、eu-west-2
AWS	TN1	ゲートベース	オンデマンドシミュレーター	arn:aws:braket:::device/quantum-simulator/amazon/tn1	us-east-1、us-west-2、eu-west-2

> **Note**
>
> デバイス ARNsでは大文字と小文字が区別されます。例えば、AQT IBEX-Q1デバイスを使用する場合は、デバイス ARN に が含まれていることを確認します'Ibex-Q1'。

Amazon Braket で使用できる QPU に関するその他の詳細を表示するには、「Amazon Braket Quantum Computers」を参照してください。

デバイスのプロパティ

すべてのデバイスについて、デバイストポロジ、キャリブレーションデータ、ネイティブゲートセットなど、デバイスの詳細なプロパティは、Amazon Braket コンソールの [デバイス] タブまたは `GetDevice` API を使用して見つけることができます。シミュレーターを使用して回路を構築する場合、Amazon Braket では、連続する量子ビットまたはインデックスを使用する必要があります。SDK を用いて作業を行う場合、以下のコード例を参照することで、使用可能な個々のデバイスとシミュレーターのデバイスプロパティへのアクセス方法を確認できます。

```
from braket.aws import AwsDevice
from braket.devices import LocalSimulator

device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/sv1')
 # SV1
# device = LocalSimulator()
 # Local State Vector Simulator
# device = LocalSimulator("default")
 # Local State Vector Simulator
# device = LocalSimulator(backend="default")
 # Local State Vector Simulator
# device = LocalSimulator(backend="braket_sv")
 # Local State Vector Simulator
# device = LocalSimulator(backend="braket_dm")
 # Local Density Matrix Simulator
# device = LocalSimulator(backend="braket_ahs")
 # Local Analog Hamiltonian Simulation
# device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/tn1')
 # TN1
# device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/dm1')
 # DM1
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/aqt/Ibex-Q1')
 # AQT IBEX-Q1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1')
 # IonQ Forte-1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1')
 # IonQ Forte-Enterprise-1
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet')
 # IQM Garnet
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald')
 # IQM Emerald
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/quera/Aquila')
 # QuEra Aquila
# device = AwsDevice('arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3')
 # Rigetti Ankaa-3
```

```
# Get device properties
device.properties
```

Amazon Braket のリージョンとエンドポイント

リージョンとエンドポイントの一覧については、「AWS 全般のリファレンス」を参照してください。

QPU デバイスで実行される量子タスクは、そのデバイスのリージョンにある Amazon Braket コンソールで表示できます。Amazon Braket SDK を使用している場合は、作業しているリージョンに関係なく、任意の QPU デバイスに量子タスクを送信できます。SDK は、指定された QPU のリージョンへのセッションを自動的に作成します。

Amazon Braket は、以下でご利用いただけます AWS リージョン。

リージョン名	リージョン	Braket エンドポイント
米国東部 (バージニア北部)	us–east–1	braket.us-east-1.amazonaws.com (IPv4 のみ) braket.us-east-1.api.aws (デュアルスタック)
米国西部 (北カリフォルニア)	us-west-1	braket.us-west-1.amazonaws.com (IPv4 のみ) braket.us-west-1.api.aws (デュアルスタック)
米国西部 2 (オレゴン)	us-west-2	braket.us-west-2.amazonaws.com (IPv4 のみ) braket.us-west-2.api.aws (デュアルスタック)
欧州北部 1 (ストックホルム)	eu-north-1	braket.eu-north-1.amazonaws.com (IPv4 のみ) braket.eu-north-1.api.aws (デュアルスタック)

リージョン名	リージョン	Braket エンドポイント
欧州西部 2 (ロンドン)	eu-west-2	braket.eu-west-2.amazonaws.com (IPv4 のみ) braket.eu-west-2.api.aws (デュアルスタック)

> **Note**
>
> Amazon Braket SDK は IPv6-only ネットワークをサポートしていません。

Amazon Braket の開始方法

> **Tip**
>
> 量子コンピューティングの基礎について説明します AWS。Amazon Braket Digital Learning Plan に登録し、一連のラーニングコースとデジタル評価を完了して自身にデジタルバッジを獲得してください。

[Amazon Braket の有効化] で表示される指示に従うと、Amazon Braket の使用を開始できます。

開始する手順は次のとおりです。

- Amazon Braket を有効にする
- Amazon Braket ノートブックインスタンスを作成する
- を使用して Braket ノートブックインスタンスを作成する CloudFormation

Amazon Braket を有効にする

> **Tip**
>
> 量子コンピューティングの基礎について説明します AWS。Amazon Braket Digital Learning Plan に登録し、一連のラーニングコースとデジタル評価を完了して自身にデジタルバッジを獲得してください。

アカウントで Amazon Braket を有効にするには、AWS コンソールを使用します。

このセクションの内容:

- 前提条件
- Amazon Braket を有効にする手順

前提条件

Amazon Braket を有効にして実行するには、Amazon Braket アクションを開始する権限を持つユーザーまたはロールが必要です。これらの権限は、 AmazonBraketFullAccess IAM ポリシー (arn:aws:iam::aws:policy/AmazonBraketFullAccess) に含まれています。

> **Note**
>
> 管理者の場合:
> 他のユーザーに Amazon Braket へのアクセス権を付与するには、ユーザーに AmazonBraketFullAccess ポリシー、または作成したカスタムポリシーをアタッチします。Amazon Braket の使用に必要なアクセス許可の詳細については、「Amazon Braket へのアクセスを管理する」を参照してください。

Amazon Braket を有効にする手順

1. を使用して Amazon Braket コンソールにサインインします AWS アカウント。
2. Amazon Braket コンソールを開きます。
3. Braket ランディングページで [開始方法] をクリックして [サービスダッシュボード] ページに移動します。サービスダッシュボードの上部にあるアラートに従うことで、次の 3 つのステップに誘導されます。
   a. サービスにリンクされたロール (SLR) の作成
   b. サードパーティーの量子コンピュータへのアクセスの有効化
   c. 新しい Jupyter Notebook インスタンスの作成

サードパーティーの量子デバイスを使用するには、自身 AWSとそれらのデバイス間のデータ転送に関する特定の条件に同意する必要があります。同意すべき条件は、Amazon Braket コンソールの [アクセス許可と設定] ページの [全般] タブに記載されています。

> **Note**
>
> Braket のローカルシミュレーターやオンデマンドシミュレーターなど、サードパーティーが関与していない量子デバイスは、「サードパーティーのデバイスを有効にする」契約に同意しなくても使用できます。

サードパーティーのハードウェアにアクセスするには、これらの条件をアカウントごとに 1 回同意するだけでよいのです。

Amazon Braket ノートブックインスタンスを作成する

> **Tip**
>
> 量子コンピューティングの基礎について説明します AWS。Amazon Braket Digital Learning Plan に登録し、一連のラーニングコースとデジタル評価を完了して自身にデジタルバッジを獲得してください。

Amazon Braket は、フルマネージドの Jupyter Notebook を提供し、作業を始めることができます。Amazon Braket ノートブックインスタンスは、Amazon SageMaker AI ノートブックインスタンスをベースにしています。新規および既存のカスタマー向けに新しいノートブックインスタンスを作成する方法・手順の概要は、次のとおりです。

Amazon Braket の新規のカスタマー:

1. Amazon Braket コンソールを開き、左のペインにある [ダッシュボード] ページを選択することで、その実際のページに移動します。
2. ダッシュボードページの中央にある [Amazon Braket へようこそ] モーダルで、[開始方法] をクリックします。ノートブック名を指定して、デフォルトの Jupyter Notebook を作成します。
   a. ノートブックが作成されるまでに数分かかる場合があります。
   b. ノートブックは、[ノートブック] ページに [保留中] のステータスで表示されます。
   c. ノートブックインスタンスが使用可能になると、ステータスは [実行中] に変わります。
   d. ノートブックの最新ステータスを表示するには、ページを更新する必要がある場合があります。

Amazon Braket の既存のカスタマー:

1. Amazon Braket コンソールを開き、左のペインにある [ノートブック] を選択します。
2. [ノートブックインスタンスの作成] を選択します。
   a. ノートブックがない場合は、[標準セットアップ] を選択してデフォルトの Jupyter Notebook を作成します。

3. [ノートブックインスタンス名] に、英数字とハイフンのみを使用した値を入力し、[ビジュアルモード] から希望の値を選択します。
4. お使いのノートブックに対し、[ノートブック非アクティブマネージャー] を有効または無効にします。
   a. 有効にした場合、ノートブックがリセットされるまでに必要なアイドル時間を選択します。ノートブックがリセットされると、コンピューティング料金は発生しなくなりますが、ストレージ料金は課金され続けます。
   b. ノートブックインスタンスのアイドル時間の残り時間を確認するには、コマンドバーに移動し、[Braket] タブ、[非アクティブマネージャー] タブの順に選択します。

> **Note**
>
> 作業を保存するには、お使いの SageMaker AI ノートブックインスタンスを Git リポジトリに組み込みます。または、作業がノートブックインスタンスの再起動によって上書きされないように、作業を `/Braket Algorithms` および `/Braket Examples` フォルダの外部に移動します。

5. (オプション) [詳細設定] では、アクセス許可、追加設定、ネットワークアクセス設定を使用してノートブックを作成できます。
   a. [ノートブックの設定] で、インスタンスタイプを選択します。
      i. 費用対効果の高い標準のインスタンスタイプ ml.t3.medium がデフォルトで選択されます。インスタンスの料金の詳細については、「Amazon SageMaker AI の料金」を参照してください。
   b. 公開 Github リポジトリをノートブックインスタンスに関連付けるには、[Git リポジトリ] ドロップダウンをクリックし、[リポジトリ] ドロップダウンメニューから [URL から公開 git リポジトリのクローンを作成] を選択します。[Git リポジトリ URL] テキストバーにリポジトリの URLを入力します。
   c. [アクセス許可] で、オプションの IAM ロール、ルートアクセス、および暗号化キーを設定します。
   d. [ネットワークアクセス] で、Jupyter Notebook インスタンスのカスタムネットワーク設定とアクセス設定を設定します。
6. 設定を確認し、タグを設定してノートブックインスタンスを決定します。[起動] をクリックします。

> **Note**
>
> Amazon Braket コンソールと Amazon SageMaker AI コンソールで、Amazon Braket ノートブックインスタンスを表示および管理します。Amazon Braket ノートブックの追加設定は SageMaker コンソールで行うことができます。

Amazon Braket SDK 内の AWS Amazon Braket コンソールで作業している場合、プラグインは作成したノートブックにプリロードされます。独自のマシンで実行を行う場合は、SDK または PennyLane プラグインをインストールするために、それぞれ、コマンド `pip install amazon-braket-sdk` またはコマンド `pip install amazon-braket-pennylane-plugin` を実行してください。

を使用して Braket ノートブックインスタンスを作成する CloudFormation

> **Tip**
>
> 量子コンピューティングの基礎について説明します AWS。Amazon Braket Digital Learning Plan に登録し、一連のラーニングコースとデジタル評価を完了して自身にデジタルバッジを獲得してください。

CloudFormation を使用して、Amazon Braket ノートブックインスタンスを管理できます。Braket ノートブックインスタンスは Amazon SageMaker AI で構築されています。CloudFormation では、目的の設定が記述されているテンプレートファイルを使用して、ノートブックインスタンスをプロビジョニングすることができます。テンプレートファイルは JSON または YAML 形式で記述されています。インスタンスは、順序正しい繰り返し可能な方法で作成、更新、削除できます。この方法は、AWS アカウントで複数の Braket ノートブックインスタンスを管理する場合に有益です。

Braket ノートブックの CloudFormation テンプレートを作成したら、 CloudFormation を使用してリソースをデプロイします。詳細については、 CloudFormation ユーザーガイドの「 CloudFormation コンソールでのスタックの作成」を参照してください。

CloudFormation を使用して Braket ノートブックインスタンスを作成するには、次の 3 つのステップを実行します。

1. SageMaker AI ライフサイクル設定スクリプトを作成する。

2. SageMaker AI が引き受ける AWS Identity and Access Management (IAM) ロールを作成します。

3. プレフィックス **amazon-braket-** を持つ SageMaker AI ノートブックインスタンスを作成する

ライフサイクル設定は、作成するすべての Braket ノートブック用に再利用できます。また、IAM ロールは、同じ実行許可を割り当てる Braket ノートブック用に再利用することができます。

このセクションの内容:

- ステップ 1: SageMaker AI ライフサイクル設定スクリプトを作成する
- ステップ 2: Amazon SageMaker AI が引き受ける IAM ロールを作成する
- ステップ 3: プレフィックス amazon-braket- を持つ SageMaker AI ノートブックインスタンスを作成する

ステップ 1: SageMaker AI ライフサイクル設定スクリプトを作成する

次のテンプレートを使用して、SageMaker AI ライフサイクル設定スクリプトを作成します。このスクリプトは、SageMaker AI ノートブックインスタンスを Braket 用にカスタマイズしています。ライフサイクル CloudFormation リソースの設定オプションについては、「CloudFormation ユーザーガイド」の「AWS::SageMaker::NotebookInstanceLifecycleConfig」を参照してください。

```
BraketNotebookInstanceLifecycleConfig:
    Type: "AWS::SageMaker::NotebookInstanceLifecycleConfig"
    Properties:
      NotebookInstanceLifecycleConfigName: BraketLifecycleConfig-${AWS::StackName}
      OnStart:
      - Content:
          Fn::Base64: |
            #!/usr/bin/env bash
            sudo -u ec2-user -i #EOS
            curl -o braket-notebook-lcc.zip https://d3ded4lzb1lnme.cloudfront.net/
notebook/braket-notebook-lcc.zip
            unzip braket-notebook-lcc.zip
            ./install.sh
            EOS

            exit 0
```

ステップ 2: Amazon SageMaker AI が引き受ける IAM ロールを作成する

Braket ノートブックインスタンスを使用すると、オペレーションがユーザーに代わって SageMaker AI により実行されます。例えば、サポートされているデバイスで回路を使用して Braket ノートブックを実行するとします。ノートブックインスタンス内で、ユーザーの代わりに SageMaker AI により、Braket に対するオペレーションが実行されます。SageMaker AI がユーザーに代わって実行できる詳細なオペレーションは、ノートブック実行ロールによって定義されています。詳細については、「Amazon SageMaker AIデベロッパーガイド」の「SageMaker AI ロール」を参照してください。

必要なアクセス許可を持つ Braket ノートブック実行ロールを作成するために、下記の例を使用します。ポリシーは必要に応じて変更できます。

> **Note**
>
> ロールに、`s3:ListBucket` というプレフィックスが付いた Amazon S3 バケットに対する `braketnotebookcdk-"` および `s3:GetObject` オペレーションのアクセス許可があることを確認してください。これらのアクセス許可は、ライフサイクル設定スクリプトが Braket ノートブックのインストールスクリプトをコピーするために必要です。

```
ExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      RoleName: !Sub AmazonBraketNotebookRole-${AWS::StackName}
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
        -
          Effect: "Allow"
          Principal:
            Service:
              - "sagemaker.amazonaws.com"
          Action:
          - "sts:AssumeRole"
      Path: "/service-role/"
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/AmazonBraketFullAccess
      Policies:
        -
          PolicyName: "AmazonBraketNotebookPolicy"
          PolicyDocument:
```

```
            Version: "2012-10-17"
            Statement:
              - Effect: Allow
                Action:
                  - s3:GetObject
                  - s3:PutObject
                  - s3:ListBucket
                Resource:
                  - arn:aws:s3:::amazon-braket-*
                  - arn:aws:s3:::braketnotebookcdk-*
              - Effect: "Allow"
                Action:
                  - "logs:CreateLogStream"
                  - "logs:PutLogEvents"
                  - "logs:CreateLogGroup"
                  - "logs:DescribeLogStreams"
                Resource:
                  - !Sub "arn:aws:logs:*:${AWS::AccountId}:log-group:/aws/sagemaker/*"
              - Effect: "Allow"
                Action:
                  - braket:*
                Resource: "*"
```

ステップ 3: プレフィックス **amazon-braket-** を持つ SageMaker AI ノートブックインスタンスを作成する

SageMaker AI ノートブックインスタンスを作成するには、SageMaker AI ライフサイクルスクリプトと、ステップ 1 とステップ 2 で作成した IAM ロールを使用します。このノートブックインスタンスは Braket 用にカスタマイズされており、Amazon Braket コンソールを使用してアクセスできます。この CloudFormation リソースの設定オプションの詳細については、「CloudFormation ユーザーガイド」の「AWS::SageMaker::NotebookInstance」を参照してください。

```
BraketNotebook:
    Type: AWS::SageMaker::NotebookInstance
    Properties:
      InstanceType: ml.t3.medium
      NotebookInstanceName: !Sub amazon-braket-notebook-${AWS::StackName}
      RoleArn: !GetAtt ExecutionRole.Arn
      VolumeSizeInGB: 30
      LifecycleConfigName: !GetAtt
 BraketNotebookInstanceLifecycleConfig.NotebookInstanceLifecycleConfigName
```

Amazon Braket で量子タスクを構築する

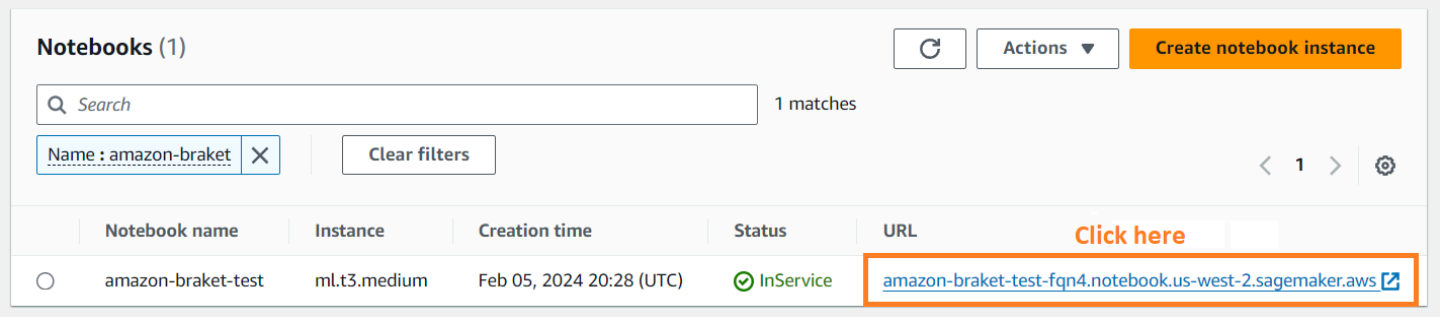
Braket は、ビルド (構築) を簡単に開始できるフルマネージド Jupyter Notebook 環境を提供します。Braket ノートブックには、サンプルアルゴリズム、リソース、および Amazon Braket SDK を含むデベロッパーツールがプリインストールされています。Amazon Braket SDK を使用すると、量子アルゴリズムを構築し、単一のコード行を変更することで、さまざまな量子コンピュータやシミュレーターで量子アルゴリズムをテストして実行できます。

このセクションの内容:

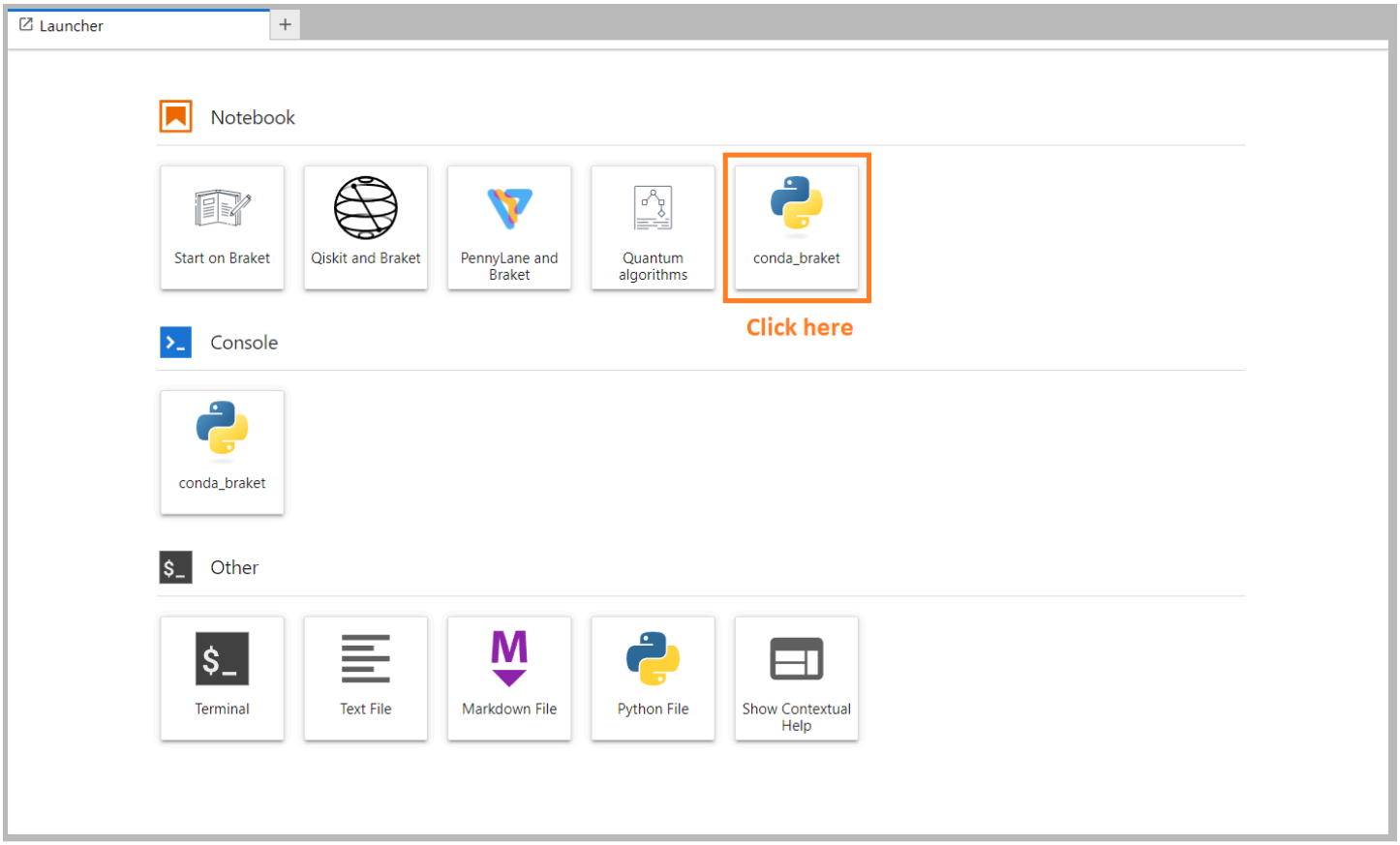
- 最初の回路を構築する
- エキスパートのアドバイスを受ける
- OpenQASM 3.0 での回路の実行
- 実験機能を探索する
- Amazon Braket でのパルス制御
- アナログハミルトニアンシミュレーション
- AWS Boto3 の使用

最初の回路を構築する

ノートブックインスタンスが起動したら、作成したノートブックを選択して、標準の Jupyter インターフェイスでインスタンスを開きます。



Amazon Braket ノートブックインスタンスは、Amazon Braket SDK とそのすべての依存関係があらかじめインストールされています。`conda_braket` カーネルを使用して新しいノートブックを作成することから始めます。



シンプルな「こんにちは、世界！」という 例から始めることができます。まず、ベル状態を準備する回路を構築し、その回路を異なるデバイスで実行して結果を取得します。

まず、Amazon Braket SDK モジュールをインポートし、simpleBRAKETlong SDK モジュールを定義して、基本的なベル状態回路を定義します。

```
import boto3
from braket.aws import AwsDevice
from braket.devices import LocalSimulator
from braket.circuits import Circuit

# Create the circuit
bell = Circuit().h(0).cnot(0, 1)
```

回路を視覚化するには、次のコマンドを使用します。

```
print(bell)
```

```
T  : #  0  #  1  #
```

```
      #####
q0 : ## H ########
      #####   #
            #####
q1 : ######## X ##
            #####
T  : #  0  #  1  #
```

[ローカルシミュレーターで回路を実行する]

次に、回路を実行する量子デバイスを選択します。Amazon Braket SDK には、ラピッドプロトタイピングとテスト用のローカルシミュレーターが付属しています。ローカルシミュレーターは、最大 25 qubits (ローカルハードウェアによって異なります) の小さい回路に使用することをお勧めします。

ローカルシミュレーターをインスタンス化するコードは次のとおりです。

```
# Instantiate the local simulator
local_sim = LocalSimulator()
```

次に、回路を実行するコードは次のとおりです。

```
# Run the circuit
result = local_sim.run(bell, shots=1000).result()
counts = result.measurement_counts
print(counts)
```

次のような結果が表示されます。

```
Counter({'11': 503, '00': 497})
```

準備した特定のベル状態は |00> と |11> の等しい重ね合わせであり、期待どおり、測定結果として 00 と 11 (shot ノイズまで含めて) がほぼ等しい分布となりました。

オンデマンドシミュレーターで回路を実行する

Amazon Braket は、より大きな回路を実行するためのオンデマンドのハイパフォーマンスシミュレーター SV1 へのアクセスも提供します。SV1 は、最大 34 qubits で構成される量子回路のシミュレーションを可能にするオンデマンドの状態ベクトルシミュレーターです。詳細について

は、SV1「サポートされているデバイス」セクションと「 AWS コンソール」を参照してください。SV1 (および TN1 または QPU) でタスクを実行する場合、量子タスクの結果はアカウントの S3 バケットに保存されます。バケットを指定しない場合、Braket SDK によりデフォルトのバケット amazon-braket-{region}-{accountID} が作成されます。詳細については、「Amazon Braket へのアクセスの管理」を参照してください。

> **Note**
>
> 実際の既存のバケット名を入力します。次の例では、バケット名として amazon-braket-s3-demo-bucket が表示されます。Amazon Braket のバケット名は常に amazon-braket- で始まり、追加した他の識別文字が続きます。S3 バケットを設定する方法に関する情報が必要な場合は、「Amazon S3 の開始方法」を参照してください。

```
# Get the account ID
aws_account_id = boto3.client("sts").get_caller_identity()["Account"]

# The name of the bucket
my_bucket = "amazon-braket-s3-demo-bucket"

# The name of the folder in the bucket
my_prefix = "simulation-output"
s3_folder = (my_bucket, my_prefix)
```

SV1 で回路を実行するには、.run( ) コールの位置引数として前に選択した S3 バケットの場所を指定する必要があります。

```
# Choose the cloud-based on-demand simulator to run your circuit
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")

# Run the circuit
task = device.run(bell, s3_folder, shots=100)

# Display the results
print(task.result().measurement_counts)
```

Amazon Braket コンソールには、量子タスクに関する詳細情報が表示されます。コンソールの [量子タスク] タブに移動します。量子タスクはリストの上部にあるはずです。または、一意の量子タスク ID またはその他の条件を使用して量子タスクを検索することもできます。

> **Note**
>
> 90 日後、Amazon Braket は、量子タスクに関連付けられているすべての量子タスク ID およびその他のメタデータを自動的に削除します。詳細については、「データ保持期間」を参照してください。

QPU で実行する

Amazon Braket を使用すると、1 行のコードを変更するだけで、物理量子コンピュータで前の量子回路の例を実行できます。Amazon Braket は、さまざまな量子処理ユニット (QPU) デバイスへのアクセスを提供します。さまざまなデバイスと可用性ウィンドウに関する情報は、サポートされているデバイスセクションと、コンソールの AWS デバイスタブにあります。次の例は、IQM デバイスのインスタンス化方法を示しています。

```
# Choose the IQM hardware to run your circuit
device = AwsDevice("arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet")
```

または、次のコードで IonQ デバイスを選択するという方法もあります。

```
# Choose the Ionq device to run your circuit
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1")
```

デバイスを選択してからワークロードを実行するまでの間に、次のコードを使用してデバイスキューの深さをクエリし、量子タスクまたはハイブリッドジョブの数を確認できます。また、カスタマーは Amazon Braket Management Console の [デバイス] ページでもデバイス固有のキューの深さを確認できます。

```
# Print your queue depth
print(device.queue_depth().quantum_tasks)
# Returns the number of quantum tasks queued on the device
# {<QueueType.NORMAL: 'Normal'>: '0', <QueueType.PRIORITY: 'Priority'>: '0'}

print(device.queue_depth().jobs)
# Returns the number of hybrid jobs queued on the device
# '2'
```

タスクを実行すると、Amazon Braket SDK が結果をポーリングします (デフォルトのタイムアウトは 5 日後です)。このデフォルトを変更するには、以下の例に示すように .run( ) コマンドの

poll_timeout_seconds パラメータを変更します。ポーリングタイムアウトが短すぎると、QPU が利用できない場合など、ポーリング時間内に結果が返されない場合があり、ローカルタイムアウトエラーが返されることに注意してください。ポーリングを再開するには、task.result() 関数を呼び出します。

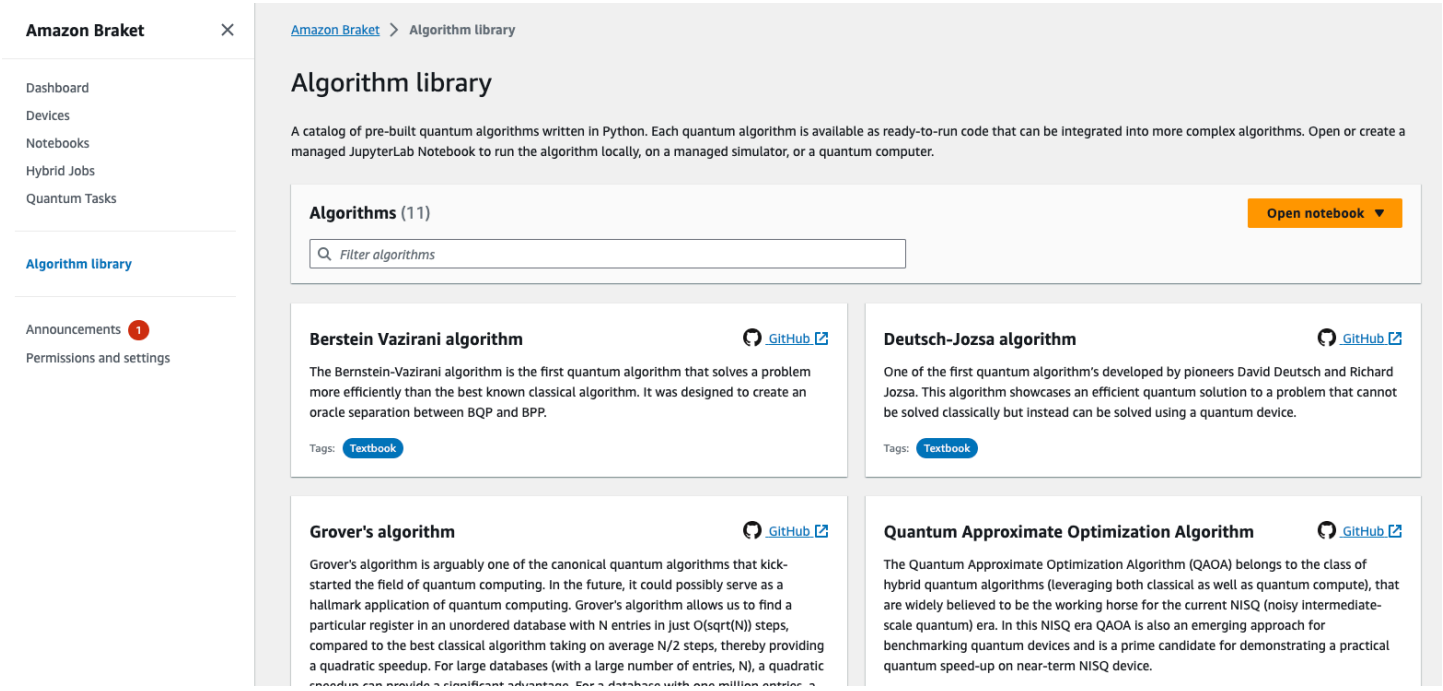
```
# Define quantum task with 1 day polling timeout
task = device.run(bell, s3_folder, poll_timeout_seconds=24*60*60)
print(task.result().measurement_counts)
```

また、キューの位置を確認するには、量子タスクまたはハイブリッドジョブを送信した後で queue_position() 関数を呼び出します。

```
print(task.queue_position().queue_position)
# Return the number of quantum tasks queued ahead of you
# '2'
```

最初の量子アルゴリズムの構築

Amazon Braket アルゴリズムライブラリは、Python で記述された構築済みの量子アルゴリズムのカタログです。これらのアルゴリズムはそのまま実行することも、より複雑なアルゴリズムを構築するための出発点として使用することもできます。アルゴリズムライブラリにアクセスするには、Braket コンソールを使用できます。詳細については、Github の「Braket algorithm library」を参照してください。



Braket コンソールには、アルゴリズムライブラリで使用可能な各アルゴリズムの概要が表示されます。GitHub リンクを選択して各アルゴリズムの詳細を表示するか、[ノートブックを開く] を選択して使用可能なすべてのアルゴリズムを含むノートブックを開くか作成します。ノートブックに関するオプションの方を選択すると、ノートブックのルートフォルダに Braket アルゴリズムライブラリが表示されます。

SDK での回路の構築

このセクションでは、回路の定義、使用可能なゲートの表示、回路の拡張、および各デバイスがサポートするゲートの表示の例を示します。また、qubits を手動で割り当てる方法、定義されたとおりに回路を実行するようにコンパイラーに指示する方法、ノイズシミュレーターを使用してノイズの多い回路を構築する方法についても説明します。

QPU によっては、Braket でさまざまなゲートのためにパルスレベルで作業することもできます。詳細については、「Pulse Control on Amazon Braket」を参照してください。

このセクションの内容:

- ゲートと回路
- プログラムセット
- 部分測定
- 手動 qubit 割り当て
- 逐語的なコンパイル
- ノイズシミュレーション

ゲートと回路

量子ゲートと回路は、Amazon Braket Python SDK の `braket.circuits` クラスで定義されています。SDK から、`Circuit()` を呼び出して、新しい回路オブジェクトをインスタンス化できます。

例: 回路を定義する

この例ではまず、標準の、1 量子ビットのアダマールゲートと 2 量子ビットの CNOT ゲートから成る 4 つのqubits (ラベルは `q0`、`q1`、`q2`、`q3`) のサンプル回路を定義しています。この回路を可視化するには、次の例に示されているように `print` 関数を呼び出します。

```
# Import the circuit module
from braket.circuits import Circuit
```

```
# Define circuit with 4 qubits
my_circuit = Circuit().h(range(4)).cnot(control=0, target=2).cnot(control=1, target=3)
print(my_circuit)
```

```
T  : #  0  #     1     #
      #####
q0 : ## H ##############
      #####   #
      #####   #
q1 : ## H ##############
      #####   #     #
      ##### #####   #
q2 : ## H ### X ########
      ##### #####   #
      #####       #####
q3 : ## H ######### X ##
      #####       #####
T  : #  0  #     1     #
```

例: パラメータ化された回路を定義する

この例では、自由パラメータに依存するゲートを持つ回路を定義します。これらのパラメータの値を指定することで、新しい回路を作成することもできれば、回路を送信する際に特定のデバイスで量子タスクとして実行することもできます。

```
from braket.circuits import Circuit, FreeParameter

# Define a FreeParameter to represent the angle of a gate
alpha = FreeParameter("alpha")

# Define a circuit with three qubits
my_circuit = Circuit().h(range(3)).cnot(control=0, target=2).rx(0, alpha).rx(1, alpha)
print(my_circuit)
```

パラメータ化された回路からパラメータ化されていない新しい回路を作成するには、次のように、単一の `float` 引数 (すべての自由パラメータが取る値)か、各パラメータの値を指定するキーワード引数を回路に指定します。

```
my_fixed_circuit = my_circuit(1.2)
my_fixed_circuit = my_circuit(alpha=1.2)
print(my_fixed_circuit)
```

my_circuit は変更されていないため、それを固定パラメータ値で使用して多くの新しい回路をインスタンス化できます。

例: 回路のゲートを変更する

次の例では、control 修飾子と power 修飾子を使用するゲートを持つ回路を定義しています。これらの修飾を使用して、制御された Ry ゲートなどの新しいゲートを作成できます。

```
from braket.circuits import Circuit

# Create a bell circuit with a controlled x gate
my_circuit = Circuit().h(0).x(control=0, target=1)

# Add a multi-controlled Ry gate of angle .13
my_circuit.ry(angle=.13, target=2, control=(0, 1))

# Add a 1/5 root of X gate
my_circuit.x(0, power=1/5)

print(my_circuit)
```

ゲート修飾子はローカルシミュレーターでのみサポートされています。

例: 使用可能なすべてのゲートを見る

次の例は、Amazon Braket で使用可能なすべてのゲートを確認する方法を示しています。

```
from braket.circuits import Gate
# Print all available gates in Amazon Braket
gate_set = [attr for attr in dir(Gate) if attr[0].isupper()]
print(gate_set)
```

このコードからの出力には、すべてのゲートが一覧表示されます。

```
['CCNot', 'CNot', 'CPhaseShift', 'CPhaseShift00', 'CPhaseShift01', 'CPhaseShift10',
 'CSwap', 'CV', 'CY', 'CZ', 'ECR', 'GPhase', 'GPi', 'GPi2', 'H', 'I', 'ISwap', 'MS',
 'PRx', 'PSwap', 'PhaseShift', 'PulseGate', 'Rx', 'Ry', 'Rz', 'S', 'Si', 'Swap', 'T',
 'Ti', 'U', 'Unitary', 'V', 'Vi', 'X', 'XX', 'XY', 'Y', 'YY', 'Z', 'ZZ']
```

これらのゲートは、そのタイプの回路のメソッドを呼び出して、回路に追加できます。例えば、circ.h(0) を呼び出し、最初の qubit にアダマールゲートを加えます。

> **Note**
>
> ゲートが所定の位置に追加され、以下の例では、前の例に挙げたすべてのゲートを同じ回路に追加しています。

```
circ = Circuit()
# toffoli gate with q0, q1 the control qubits and q2 the target.
circ.ccnot(0, 1, 2)
# cnot gate
circ.cnot(0, 1)
# controlled-phase gate that phases the |11> state, cphaseshift(phi) =
 diag((1,1,1,exp(1j*phi))), where phi=0.15 in the examples below
circ.cphaseshift(0, 1, 0.15)
# controlled-phase gate that phases the |00> state, cphaseshift00(phi) =
 diag([exp(1j*phi),1,1,1])
circ.cphaseshift00(0, 1, 0.15)
# controlled-phase gate that phases the |01> state, cphaseshift01(phi) =
 diag([1,exp(1j*phi),1,1])
circ.cphaseshift01(0, 1, 0.15)
# controlled-phase gate that phases the |10> state, cphaseshift10(phi) =
 diag([1,1,exp(1j*phi),1])
circ.cphaseshift10(0, 1, 0.15)
# controlled swap gate
circ.cswap(0, 1, 2)
# swap gate
circ.swap(0,1)
# phaseshift(phi)= diag([1,exp(1j*phi)])
circ.phaseshift(0,0.15)
# controlled Y gate
circ.cy(0, 1)
# controlled phase gate
circ.cz(0, 1)
# Echoed cross-resonance gate applied to q0, q1
circ = Circuit().ecr(0,1)
# X rotation with angle 0.15
circ.rx(0, 0.15)
# Y rotation with angle 0.15
circ.ry(0, 0.15)
# Z rotation with angle 0.15
circ.rz(0, 0.15)
# Hadamard gates applied to q0, q1, q2
```

```
circ.h(range(3))
# identity gates applied to q0, q1, q2
circ.i([0, 1, 2])
# iswap gate, iswap = [[1,0,0,0],[0,0,1j,0],[0,1j,0,0],[0,0,0,1]]
circ.iswap(0, 1)
# pswap gate, PSWAP(phi) = [[1,0,0,0],[0,0,exp(1j*phi),0],[0,exp(1j*phi),0,0],
[0,0,0,1]]
circ.pswap(0, 1, 0.15)
# X gate applied to q1, q2
circ.x([1, 2])
# Y gate applied to q1, q2
circ.y([1, 2])
# Z gate applied to q1, q2
circ.z([1, 2])
# S gate applied to q0, q1, q2
circ.s([0, 1, 2])
# conjugate transpose of S gate applied to q0, q1
circ.si([0, 1])
# T gate applied to q0, q1
circ.t([0, 1])
# conjugate transpose of T gate applied to q0, q1
circ.ti([0, 1])
# square root of not gate applied to q0, q1, q2
circ.v([0, 1, 2])
# conjugate transpose of square root of not gate applied to q0, q1, q2
circ.vi([0, 1, 2])
# exp(-iXX theta/2)
circ.xx(0, 1, 0.15)
# exp(i(XX+YY) theta/4), where theta=0.15 in the examples below
circ.xy(0, 1, 0.15)
# exp(-iYY theta/2)
circ.yy(0, 1, 0.15)
# exp(-iZZ theta/2)
circ.zz(0, 1, 0.15)
# IonQ native gate GPi with angle 0.15 applied to q0
circ.gpi(0, 0.15)
# IonQ native gate GPi2 with angle 0.15 applied to q0
circ.gpi2(0, 0.15)
# IonQ native gate MS with angles 0.15, 0.15, 0.15 applied to q0, q1
circ.ms(0, 1, 0.15, 0.15, 0.15)
```

定義済みのゲートセットとは別に、自己定義のユニタリゲートを回路に適用することもできます。これらは、単一量子ビットゲート (次のソースコードに示すとおり) とすることも、targets パラメータにより定義される qubits に適用されるマルチ量子ビットゲートとすることもできます。

```
import numpy as np

# Apply a general unitary
my_unitary = np.array([[0, 1],[1, 0]])
circ.unitary(matrix=my_unitary, targets=[0])
```

例: 既存の回路を拡張する

命令を追加することで、既存の回路を拡張できます。Instruction は、量子デバイス上で実行する量子タスクを記述する量子指令です。Instruction 作用素には、Gate のみのタイプのオブジェクトが含まれます。

```
# Import the Gate and Instruction modules
from braket.circuits import Gate, Instruction

# Add instructions directly.
circ = Circuit([Instruction(Gate.H(), 4), Instruction(Gate.CNot(), [4, 5])])

# Or with add_instruction/add functions
instr = Instruction(Gate.CNot(), [0, 1])
circ.add_instruction(instr)
circ.add(instr)

# Specify where the circuit is appended
circ.add_instruction(instr, target=[3, 4])
circ.add_instruction(instr, target_mapping={0: 3, 1: 4})

# Print the instructions
print(circ.instructions)
# If there are multiple instructions, you can print them in a for loop
for instr in circ.instructions:
     print(instr)

# Instructions can be copied
new_instr = instr.copy()
# Appoint the instruction to target
new_instr = instr.copy(target=[5, 6])
new_instr = instr.copy(target_mapping={0: 5, 1: 6})
```

例: 各デバイスがサポートするゲートの表示

シミュレーターは Braket SDK のすべてのゲートをサポートしますが、QPU デバイスは小さなサブセットをサポートします。デバイスのサポートされているゲートは、デバイスのプロパティで確認できます。IonQ デバイスの例を次に示します。

```
# Import the device module
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1")

# Get device name
device_name = device.name
# Show supportedQuantumOperations (supported gates for a device)
device_operations = device.properties.dict()['action']['braket.ir.openqasm.program']
['supportedOperations']
print('Quantum Gates supported by {}:\n {}'.format(device_name, device_operations))
```

```
Quantum Gates supported by Aria 1:
 ['x', 'y', 'z', 'h', 's', 'si', 't', 'ti', 'v', 'vi', 'rx', 'ry', 'rz', 'cnot',
 'swap', 'xx', 'yy', 'zz']
```

サポートされているゲートは、量子ハードウェアで実行する前に、ネイティブゲートにコンパイルする必要があります。回路を送信すると、Amazon Braket はそのコンパイルを自動的に実行します。

例: デバイスでサポートされているネイティブゲートの忠実度をプログラムで取得する

忠実度情報は、Braket コンソールの [デバイス] ページで確認できます。同じ情報にプログラムでアクセスできると役立つ場合があります。次のコードは、QPU の 2 つの量子ビット間における 2 つの qubit ゲートの忠実度を抽出する方法を示しています。

```
# Import the device module
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

# Specify the qubits
a=10
b=11
edge_properties_entry =
 device.properties.standardized.twoQubitProperties['10-11'].twoQubitGateFidelity
gate_name = edge_properties_entry[0].gateName
```

```
fidelity = edge_properties_entry[0].fidelity
print(f"Fidelity of the {gate_name} gate between qubits {a} and {b}: {fidelity}")
```

プログラムセット

プログラムセットは、1 つの量子タスクで複数の量子回路を効率的に実行します。その 1 つのタスクでは、最大 100 個の量子回路を送信することも、最大 100 個の異なるパラメータセットを持つ 1 つのパラメトリック回路を送信することもできます。このようなオペレーションにより、後続の回路実行間の時間が最短に抑えられ、量子タスク処理のオーバーヘッドが削減されます。現在、プログラムセットは Amazon Braket Local Simulatorと AQT、IQMおよび Rigettiデバイスでサポートされています。

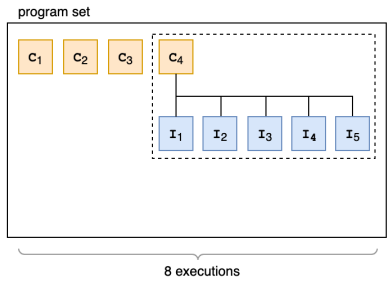
ProgramSet の定義

次の最初のコード例は、パラメータ化された回路を使用する場合とパラメータのない回路を使用する場合での ProgramSet の構築方法を示しています。

```
from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter
from braket.program_sets.circuit_binding import CircuitBinding
from braket.program_sets import ProgramSet

# Initialize the quantum device
device = AwsDevice("arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet")

# Define circuits
circ1 = Circuit().h(0).cnot(0, 1)
circ2 = Circuit().rx(0, 0.785).ry(1, 0.393).cnot(1, 0)
circ3 = Circuit().t(0).t(1).cz(0, 1).s(0).cz(1, 2).s(1).s(2)
parameterize_circuit = Circuit().rx(0, FreeParameter("alpha")).cnot(0, 1).ry(1,
 FreeParameter("beta"))

# Create circuit bindings with different parameters
circuit_binding = CircuitBinding(
    circuit=parameterize_circuit,
    input_sets={
            'alpha': (0.10, 0.11, 0.22, 0.34, 0.45),
            'beta': (1.01, 1.01, 1.03, 1.04, 1.04),
    })

# Creating the program set
program_set_1 = ProgramSet([
```

```
    circ1,
    circ2,
    circ3,
    circuit_binding,
])
```

このプログラムセットには circ1、circ2、circ3、および circuit_binding の、4 つの一意のプログラムが含まれています。circuit_binding プログラムは 5 つの異なるパラメータバインディングで実行され、5 つの実行可能回路を作成します。パラメータを使用しない他の 3 つのプログラムは、それぞれ 1 つの実行可能回路を作成します。したがって、合計では 8 つの実行可能回路が生成されることになります。これを次の図に示します。



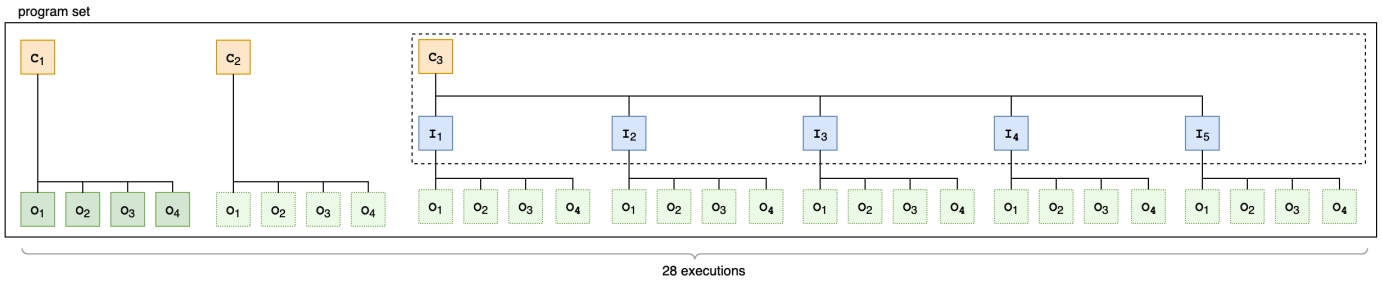

```
ProgramSet(
    [c1,
     c2,
     c3,
     CircuitBinding(c4, [i1, i2, i3, i4, i5])]
)
```

次の、2 番目のコード例は、product() メソッドを使用して、プログラムセットの各実行可能回路に同じオブザーバブルのセットをアタッチする方法を示しています。

```
from braket.circuits.observables import I, X, Y, Z

observables = [Z(0) @ Z(1), X(0) @ X(1), Z(0) @ X(1), X(0) @ Z(1)]

program_set_2 = ProgramSet.product(
    circuits=[circ1, circ2, circuit_binding],
    observables=observables
)
```

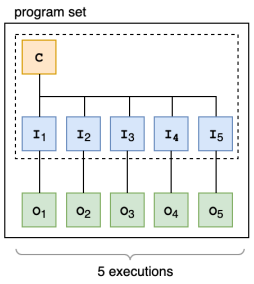
パラメータを使用しないプログラムの場合、各オブザーバブルは回路ごとに測定されます。パラメータを使用するプログラムの場合、次の図に示すように、各オブザーバブルは入力セットごとに測定されます。

```
ProgramSet.product(
    circuits=[c1, c2, CircuitBinding(c3, [i1, i2, i3, i4, i5])],
    observables=[o1, o2, o3, o4]
)
```



次の、3 番目のコード例は、zip() メソッドを使用して、個々のオブザーバブルを ProgramSet 内の特定のパラメータセットとペアリングする方法を示しています。

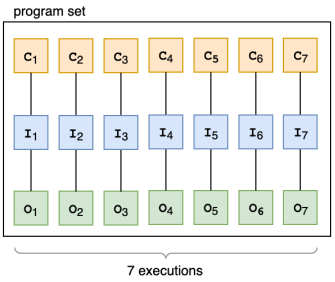
```
program_set_3 = ProgramSet.zip(
    circuits=circuit_binding,
    observables=observables + [Y(0) @ Y(1)]
)
```




```
ProgramSet.zip(
    circuits=CicruitBinding(
        circuit=c,
        input_sets=[i1, i2, i3, i4, i5]
    ),
    observables=[o1, o2, o3, o4, o5]
)
```

CircuitBinding() の代わりに、回路と入力セットのリストを使用して、オブザーバブルのリストを直接 zip できます。

```
program_set_4 = ProgramSet.zip(
    circuits=[circ1, circ2, circ3],
    input_sets=[{}, {}, {}],
    observables=observables[:3]
)
```




```
ProgramSet.zip(
    circuits=[c1, c2, c3, c4, c5, c6, c7],
    input_sets=[i1, i2, i3, i4, i5, i6, i7],
    observables=[o1, o2, o3, o4, o5, o6, o7]
)
```

プログラムセットの詳細と例については、amazon-braket-examples Github の「プログラムセットフォルダー」を参照してください。

デバイスでプログラムセットを検査して実行する

プログラムセットの実行可能回路数は、一意のパラメータ付き回路の数と等しくなります。実行可能回路数と回路のショット数を合算するには、次のコード例を使用します。

```
# Number of shots per executable
shots = 10
num_executables = program_set_1.total_executables

# Calculate total number of shots across all executables
total_num_shots = shots*num_executables
```

> **Note**
>
> プログラムセットを使用して、プログラムセット内のすべての回路の合計ショット数に基づいて、タスクごとの料金とショットごとの料金を支払います。

プログラムセットを実行するには、次のコード例を使用します。

```
# Run the program set
task = device.run(
   program_set_1, shots=total_num_shots,
)
```

Rigetti デバイスを使用する場合、タスクの一部のみが終了して残りがまだキューに入っている間、プログラムセットは RUNNING 状態のままになることがあります。より迅速な結果を得るには、プログラムセットをハイブリッドジョブとして送信することを検討してください。

結果の分析

ProgramSet 内の実行可能回路の結果を分析および測定するには、次のコードを実行します。

```
# Get the results from a program set
result = task.result()

# Get the first executbable
first_program = result[0]
first_executable = first_program[0]

# Inspect the results of the first executable
measurements_from_first_executable = first_executable.measurements
print(measurements_from_first_executable)
```

部分測定

部分測定を使用することで、量子回路内のすべての量子ビットを測定する代わりに量子ビットのサブセットまたは個々の量子ビットを測定できます。

> **Note**
>
> 実験機能として、中間回路測定およびフィードフォワードオペレーションなどの追加機能が利用できるようになりました。「Access to dynamic circuits on IQM devices」を参照。

例: 量子ビットのサブセットを測定する

次のコード例は、ベル状態回路で量子ビット 0 のみを測定する部分測定を示しています。

```
from braket.devices import LocalSimulator
from braket.circuits import Circuit

# Use the local state vector simulator
device = LocalSimulator()

# Define an example bell circuit and measure qubit 0
circuit = Circuit().h(0).cnot(0, 1).measure(0)

# Run the circuit
task = device.run(circuit, shots=10)
```

```
# Get the results
result = task.result()

# Print the circuit and measured qubits
print(circuit)
print()
print("Measured qubits: ", result.measured_qubits)
```

手動 qubit 割り当て

Rigetti の量子コンピュータで量子回路を実行する場合、任意で手動 qubit 割り当てを使用して、アルゴリズムに使用される qubits を制御できます。Amazon Braket コンソールと Amazon Braket SDK は、選択した量子処理装置 (QPU) デバイスの最新のキャリブレーションデータを検査するのに役立ち、実験に最適な qubits を選択できます。

手動qubit割り当てを使用すると、回路をより正確に実行し、個々のqubitのプロパティを調べることができます。研究者および上級ユーザーは、最新のデバイスキャリブレーションデータに基づいて回路設計を最適化し、より正確な結果を得ることができます。

次の例は、明示的に qubits を割り当てる方法を示しています。

```
# Import the device module
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
circ = Circuit().h(0).cnot(0, 7)  # Indices of actual qubits in the QPU

# Set up S3 bucket (where results are stored)
my_bucket = "amazon-braket-s3-demo-bucket" # The name of the bucket
my_prefix = "your-folder-name" # The name of the folder in the bucket
s3_location = (my_bucket, my_prefix)

my_task = device.run(circ, s3_location, shots=100, disable_qubit_rewiring=True)
```

詳細については、「GitHub での Amazon Braket の例」、より具体的には、ノートブック:「QPU デバイスでの量子ビットの割り当て」を参照してください。

逐語的なコンパイル

ゲートベースの量子コンピュータで量子回路を実行する場合、変更を加えることなく、定義したとおりに正確に回路を実行するようにコンパイラーに指示することができます。逐語的なコンパイルを使用して、回路全体を指定どおりに正確に保持するか、または回路の特定の部分のみを保持する

(Rigetti でのみ可能) ように指定できます。ハードウェアベンチマークまたはエラー緩和プロトコルのアルゴリズムを開発する場合、ハードウェアで実行するゲートと回路レイアウトを正確に指定できる必要があります。逐語的なコンパイルでは、特定の最適化ステップを無効にすることで、コンパイルプロセスを直接制御できます。これにより、回路が設計どおりに実行されるようになります。

逐語的なコンパイルは、AQT、、IonQIQM、および Rigetti デバイスでサポートされており、ネイティブゲートを使用する必要があります。逐語的なコンパイルを使用する場合は、デバイスのトポロジをチェックして、接続された qubits でゲートが呼び出され、回路がハードウェアでサポートされているネイティブゲートを使用していることを確認することをお勧めします。次に、デバイスでサポートされるネイティブゲートのリストにプログラムでアクセスする例を示します。

```
device.properties.paradigm.nativeGateSet
```

Rigetti 向けには、逐語的コンパイルと併用するために disableQubitRewiring=True を設定することで、qubit の再配線を無効にする必要があります。disableQubitRewiring=False がコンパイルで逐語的ボックスを使用するときに設定されると、量子回路は検証に失敗し、実行されません。

回路に対して逐語的なコンパイルが有効で、それをサポートしていない QPU で実行すると、サポートされていないオペレーションによってタスクが失敗したことを示すエラーが生成されます。コンパイラー関数をネイティブにサポートする量子ハードウェアが増えるにつれて、この機能は拡張され、これらのデバイスを含めるようになります。逐語的なコンパイルをサポートするデバイスは、次のコードで照会されたときに、サポートされているオペレーションとしてそれを含めます。

```
from braket.aws import AwsDevice
from braket.device_schema.device_action_properties import DeviceActionType
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
device.properties.action[DeviceActionType.OPENQASM].supportedPragmas
```

逐語的なコンパイルを使用しても追加コストは発生しません。Amazon Braket の料金ページで指定されている現在の料金に基づいて、Braket QPU デバイス、ノートブックインスタンス、オンデマンドシミュレーターで実行された量子タスクに対して、引き続き課金されます。詳細については、逐語的なコンパイルサンプルノートブックを参照してください。

> **Note**
>
> OpenQASM を使用して AQTおよび IonQデバイスの回路を書き込み、回路を物理量子ビットに直接マッピングする場合は、OpenQASM によって #pragma braket verbatim disableQubitRewiringフラグが無視されるため、 を使用する必要があります。

ノイズシミュレーション

ローカルノイズシミュレーターをインスタンス化するために、バックエンドを次のように変更できます。

```
# Import the device module
from braket.aws import AwsDevice

device = LocalSimulator(backend="braket_dm")
```

ノイズの多い回路は、次の 2 つの方法で構築できます。

1. ノイズの多い回路を最初から構築する。
2. 既存のノイズのない回路を利用し、全体にノイズを挿入する。

以下の例は、脱分極ノイズとカスタム Kraus チャネルを備えた基本的な回路を使用したアプローチを示しています。

```
import scipy.stats
import numpy as np

# Bottom up approach
# Apply depolarizing noise to qubit 0 with probability of 0.1
circ = Circuit().x(0).x(1).depolarizing(0, probability=0.1)

# Create an arbitrary 2-qubit Kraus channel
E0 = scipy.stats.unitary_group.rvs(4) * np.sqrt(0.8)
E1 = scipy.stats.unitary_group.rvs(4) * np.sqrt(0.2)
K = [E0, E1]

# Apply a two-qubit Kraus channel to qubits 0 and 2
circ = circ.kraus([0, 2], K)
```

```
from braket.circuits import Noise

# Inject noise approach
# Define phase damping noise
noise = Noise.PhaseDamping(gamma=0.1)
# The noise channel is applied to all the X gates in the circuit
circ = Circuit().x(0).y(1).cnot(0, 2).x(1).z(2)
circ_noise = circ.copy()
```

```
circ_noise.apply_gate_noise(noise, target_gates=Gate.X)
```

回路を実行するユーザーエクスペリエンスは、次の 2 つの例に示されているように前のユーザーエクスペリエンスと変わりません。

例 1

```
task = device.run(circ, shots=100)
```

または

例 2

```
task = device.run(circ_noise, shots=100)
```

その他の例については、「Braket 入門ノイズシミュレーターの例」を参照してください。

回路の検査

Amazon Braket の量子回路には、Moments という擬似時間の概念があります。各 qubit に対し、Moment あたり 1 つのゲートを実行できます。Moments の目的は、回路とそのゲートに対処しやすくし、時間的な構造を提供することです。

> **Note**
>
> モーメントは通常、QPU でゲートが実行されるリアルタイムに対応していません。

回路の深さは、その回路内のモーメントの総数によって与えられます。回路の深さを表示するには、次の例に示すようにメソッド circuit.depth を呼び出します。

```
from braket.circuits import Circuit

# Define a circuit with parametrized gates
circ = Circuit().rx(0, 0.15).ry(1, 0.2).cnot(0, 2).zz(1, 3, 0.15).x(0)
print(circ)
print('Total circuit depth:', circ.depth)
```

```
T  : #     0      #        1         #  2  #
      ############                    #####
q0 : ## Rx(0.15) ###################### X ##
```

```
      ############   #                #####
      ############   #   ############
q1 : ## Ry(0.20) ######### ZZ(0.15) ########
      ############   #   ############
                  #####      #
q2 : ############## X ######################
                  #####      #
                       ############
q3 : ###################### ZZ(0.15) ########
                       ############
T  : #     0      #        1         #  2  #
Total circuit depth: 3
```

上記の回路の回路全体の深度は 3 です (モーメント 0、1、および 2 で示されます)。各モーメントのゲート動作を確認することができます。

Moments は、キー、バリューのペアのディクショナリとして機能します。

- このキーは、MomentsKey() です。これには擬似時間とqubitの情報が含まれす。
- この値は、Instructions() のタイプで割り当てられます。

```
moments = circ.moments
for key, value in moments.items():
    print(key)
    print(value, "\n")
```

```
MomentsKey(time=0, qubits=QubitSet([Qubit(0)]), moment_type=<MomentType.GATE: 'gate'>,
 noise_index=0, subindex=0)
Instruction('operator': Rx('angle': 0.15, 'qubit_count': 1), 'target':
 QubitSet([Qubit(0)]), 'control': QubitSet([]), 'control_state': (), 'power': 1)

MomentsKey(time=0, qubits=QubitSet([Qubit(1)]), moment_type=<MomentType.GATE: 'gate'>,
 noise_index=0, subindex=0)
Instruction('operator': Ry('angle': 0.2, 'qubit_count': 1), 'target':
 QubitSet([Qubit(1)]), 'control': QubitSet([]), 'control_state': (), 'power': 1)

MomentsKey(time=1, qubits=QubitSet([Qubit(0), Qubit(2)]), moment_type=<MomentType.GATE:
 'gate'>, noise_index=0, subindex=0)
Instruction('operator': CNot('qubit_count': 2), 'target': QubitSet([Qubit(0),
 Qubit(2)]), 'control': QubitSet([]), 'control_state': (), 'power': 1)
```

```
MomentsKey(time=1, qubits=QubitSet([Qubit(1), Qubit(3)]), moment_type=<MomentType.GATE:
 'gate'>, noise_index=0, subindex=0)
Instruction('operator': ZZ('angle': 0.15, 'qubit_count': 2), 'target':
 QubitSet([Qubit(1), Qubit(3)]), 'control': QubitSet([]), 'control_state': (), 'power':
 1)

MomentsKey(time=2, qubits=QubitSet([Qubit(0)]), moment_type=<MomentType.GATE: 'gate'>,
 noise_index=0, subindex=0)
Instruction('operator': X('qubit_count': 1), 'target': QubitSet([Qubit(0)]), 'control':
 QubitSet([]), 'control_state': (), 'power': 1)
```

また、Moments を介して回路にゲートを追加することもできます。

```
from braket.circuits import Instruction, Gate

new_circ = Circuit()
instructions = [Instruction(Gate.S(), 0),
                Instruction(Gate.CZ(), [1, 0]),
                Instruction(Gate.H(), 1)
                ]

new_circ.moments.add(instructions)
print(new_circ)
```

```
T  : #  0  #  1  #  2  #
      ##### #####
q0 : ## S ### Z ########
      ##### #####
               #   #####
q1 : ############## H ##
                  #####
T  : #  0  #  1  #  2  #
```

結果タイプのリスト

Amazon Braket は、ResultType を使用して回路を測定すると、異なるタイプの結果を返すことができます。回路が返すことができる結果のタイプは次のとおりです。

- AdjointGradient は、指定されたオブザーバブルの期待値の勾配 (ベクトルの導関数) を返します。このオブザーバブルは、指定されたパラメータに関して随伴微分法を用いて、提供されたターゲットに作用します。この微分法は、shots=0 の場合にのみ使用できます。

- `Amplitude` は、出力波動関数の指定された量子状態の振幅を返します。これは、SV1 およびローカルシミュレーターでのみ使用できます。
- `Expectation` は、指定されたオブザーバブルの期待値を返します。この値は後ほどこの章内で紹介する `Observable` クラスで指定できます。オブザーバブルの測定に使用されるターゲット qubits を指定する必要があります。また、指定された qubits の数は、オブザーバブルが作用する量子ビットの数と等しくなければなりません。ターゲットが指定されていない場合、オブザーバブルは 1 qubit でのみ動作し、すべての qubits に並列に適用されます。
- `Probability` は、計算基底状態を測定する確率を返します。ターゲットが指定されていない場合、`Probability` はすべての基底状態を測定する確率を返します。ターゲットが指定されている場合、指定された qubits の基底ベクトルの周辺確率のみが返されます。マネージドシミュレーターと QPU は最大 15 量子ビットに制限され、ローカルシミュレーターはシステムのメモリサイズに制限されます。
- `Reduced density matrix` は、qubits のシステムから、指定されたターゲット qubits のサブシステムの密度マトリックスを返します。この結果タイプのサイズを制限するため、Braket はターゲット qubits の数を最大 8 に制限しています。
- `StateVector` は、完全な状態ベクトルを返します。ローカルシミュレーターで利用できます。
- `Sample` は、指定されたターゲット qubit セットおよびオブザーバブルの測定カウントを返します。ターゲットが指定されていない場合、オブザーバブルは 1 qubit でのみ動作し、すべての qubits に並列に適用されます。ターゲットが指定されている場合、指定されたターゲットの数は、オブザーバブルが作用する qubits の数と等しくなければなりません。
- `Variance` は、指定されたターゲット qubit セットおよび要求された結果タイプとしてのオブザーバブルの分散 (`mean([x-mean(x)]`^{2})) を返します。ターゲットが指定されていない場合、オブザーバブルは 1 qubit でのみ動作し、すべての qubits に並列に適用されます。それ以外の場合、指定するターゲットの数は、オブザーバブルを適用できるqubitsの数と等しくなければなりません。

さまざまなプロバイダーでサポートされている結果タイプ:

	ローカルシム	SV1	DM1	TN1	AQT	IonQ	IQM	Rigetti
随伴勾配	いいえ	はい	N	N	N	N	N	いいえ
Amplitud	はい	Y	N	N	N	N	N	いいえ

期待値	はい	Y	Y	Y	Y	Y	Y	はい
確率:	はい	Y	Y	いいえ	Y	Y	Y	はい
縮約密度マトリックス	はい	いいえ	はい	N	N	N	N	いいえ
状態ベクトル	はい	N	N	N	N	N	N	いいえ
サンプル	はい	Y	Y	Y	Y	Y	Y	はい
分散	はい	Y	Y	Y	Y	Y	Y	はい

サポートされている結果タイプを確認するには、次の例に示すように、デバイスのプロパティを調べます。

```
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

# Print the result types supported by this device
for iter in
 device.properties.action['braket.ir.openqasm.program'].supportedResultTypes:
    print(iter)
```

```
name='Sample' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=50000
name='Expectation' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=50000
name='Variance' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=50000
name='Probability' observables=None minShots=10 maxShots=50000
```

ResultType を呼び出すには、次の例に示すように、結果タイプを回路に追加します。

```
from braket.circuits import Circuit, Observable

circ = Circuit().h(0).cnot(0, 1).amplitude(state=["01", "10"])
```

```
circ.probability(target=[0, 1])
circ.probability(target=0)
circ.expectation(observable=Observable.Z(), target=0)
circ.sample(observable=Observable.X(), target=0)
circ.state_vector()
circ.variance(observable=Observable.Z(), target=0)

# Print one of the result types assigned to the circuit
print(circ.result_types[0])
```

> **Note**
>
> 結果は、量子デバイスによって異なる形式で提供されます。例えば、Rigetti デバイスは測定値を返すのに対し、IonQ デバイスは確率を提供します。Amazon Braket SDK は、どの結果についても測定値プロパティを提供します。ただし、確率を返すデバイスの場合、ショットごとの測定値は得られないため、これらの測定値は確率に基づいて事後計算されます。結果が事後計算されたかどうかを判断するには、結果オブジェクトの `measurements_copied_from_device` を確認します。このオペレーションの詳細については、Amazon Braket SDK GitHub リポジトリの gate_model_quantum_task_result.py ファイルを参照してください。

オブザーバブル

Amazon Braket の `Observable` クラスでは、特定のオブザーバブルを測定できます。

各 qubit には、一意の非同一性オブザーバブルを 1 つのみ適用できます。同じ qubit に 2 つ以上の異なる非同一性オブザーバブルを指定すると、エラーが表示されます。この目的のために、テンソル積の各因子は個々のオブザーバブルとしてカウントされます。つまり、qubitに作用する因子が同じである限り、同じ qubit に複数のテンソル積を持つことができます。

オブザーバブルはスケールでき、他のオブザーバブル (スケールするかどうかは問わない) を追加できます。これにより、`AdjointGradient` 結果タイプで使用できる `Sum` が作成されます。

`Observable` クラスには次のオブザーバブルが含まれています。

```
import numpy as np

Observable.I()
Observable.H()
```

```
Observable.X()
Observable.Y()
Observable.Z()

# Get the eigenvalues of the observable
print("Eigenvalue:", Observable.H().eigenvalues)
# Or rotate the basis to be computational basis
print("Basis rotation gates:", Observable.H().basis_rotation_gates)

# Get the tensor product of the observable for the multi-qubit case
tensor_product = Observable.Y() @ Observable.Z()
# View the matrix form of an observable by using
print("The matrix form of the observable:\n", Observable.Z().to_matrix())
print("The matrix form of the tensor product:\n", tensor_product.to_matrix())

# Factorize an observable in the tensor form
print("Factorize an observable:", tensor_product.factors)

# Self-define observables, given it is a Hermitian
print("Self-defined Hermitian:", Observable.Hermitian(matrix=np.array([[0, 1], [1,
 0]])))

print("Sum of other (scaled) observables:", 2.0 * Observable.X() @ Observable.X() + 4.0
 * Observable.Z() @ Observable.Z())
```

```
Eigenvalue: [ 1. -1.]
Basis rotation gates: (Ry('angle': -0.7853981633974483, 'qubit_count': 1),)
The matrix form of the observable:
 [[ 1.+0.j  0.+0.j]
 [ 0.+0.j -1.+0.j]]
The matrix form of the tensor product:
 [[ 0.+0.j  0.+0.j  0.-1.j  0.+0.j]
 [ 0.+0.j -0.+0.j  0.+0.j  0.+1.j]
 [ 0.+1.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.-1.j  0.+0.j -0.+0.j]]
Factorize an observable: (Y('qubit_count': 1), Z('qubit_count': 1))
Self-defined Hermitian: Hermitian('qubit_count': 1, 'matrix': [[0.+0.j 1.+0.j], [1.+0.j
 0.+0.j]])
Sum of other (scaled) observables: Sum(TensorProduct(X('qubit_count': 1),
 X('qubit_count': 1)), TensorProduct(Z('qubit_count': 1), Z('qubit_count': 1)))
```

パラメータ

回路には自由パラメータを組み込むことができます。これらの自由パラメータは、勾配の計算に使用でき、1 回だけ構築すれば何回でも実行できます。

各自由パラメータが使用する文字列エンコードされた名前は、以下の目的に使用されます。

- パラメータ値を設定する
- 使用するパラメータを特定する

```
from braket.circuits import Circuit, FreeParameter, observables
from braket.parametric import FreeParameter

theta = FreeParameter("theta")
phi = FreeParameter("phi")
circ = Circuit().h(0).rx(0, phi).ry(0, phi).cnot(0, 1).xx(0, 1, theta)
```

随伴勾配

SV1 デバイスは、多項ハミルトニアンを含む、オブザーバブルの期待値の随伴勾配を計算します。パラメータを区別するには、名前 (文字列形式) または直接参照で指定します。

```
from braket.aws import AwsDevice
from braket.devices import Devices

device = AwsDevice(Devices.Amazon.SV1)

circ.adjoint_gradient(observable=3 * Observable.Z(0) @ Observable.Z(1) - 0.5 *
 observables.X(0), parameters = ["phi", theta])
```

固定パラメータ値を引数としてパラメータ化された回路に渡すと、自由パラメータが削除されます。この回路を AdjointGradient で実行すると、自由パラメータが存在しなくなっているため、エラーが発生します。次のコード例は、正しい使用方法と誤った使用方法を示しています。

```
# Will error, as no free parameters will be present
#device.run(circ(0.2), shots=0)

# Will succeed
device.run(circ, shots=0, inputs={'phi': 0.2, 'theta': 0.2})
```

エキスパートのアドバイスを受ける

Braket マネジメントコンソールで量子コンピューティングのエキスパートに直接繋がることで、ワークロードに関する追加のガイダンスを受けることができます。

Braket Direct を使用してエキスパートのアドバイスオプションを確認するには、Braket コンソールを開き、左側のペインで [Braket Direct] を選択し、[エキスパートのアドバイス] セクションに移動します。また、以下のエキスパートのアドバイスオプションも用意されています。

- Braket オフィスアワー: Braket オフィスアワーは 1:1 のセッションであり、先着順で毎月行われます。オフィスアワーで利用可能な個々のスロットは 30 分であり、無料です。Braket のエキスパートに相談することで、ユースケースとデバイスの適合度を調べ、アルゴリズムに Braket を最適に使用するオプションを特定し、Amazon Braket Hybrid Jobs、Braket Pulse、アナログハミルトニアンシミュレーションなど特定の Braket 機能の使用方法に関する推奨事項を得ることができるため、コンセプト立案から実行までの時間を短縮できます。
  - Braket オフィスアワーにサインアップするには、[サインアップ] を選択し、連絡先情報、ワークロードの詳細、必要なディスカッショントピックを入力します。
  - 空いている次のスロットへのカレンダーによる招待が E メールで届きます。

  > **Note**
  >
  > 緊急の問題やクイックトラブルシューティングの質問については、 AWS サポートに問い合わせることをお勧めします。緊急でない質問には、AWS re:Post forum または Quantum Computing Stack Exchange を使用することもできます。後者では、以前に回答された質問を参照したり、新しい質問をしたりできます。

- 量子ハードウェアプロバイダーのサービス: IonQ、QuEra、および Rigetti はそれぞれ、 AWS Marketplaceを通じてプロフェッショナルサービスを提供しています。
  - サービスを調べるには、[接続] を選択してリストを参照します。
  - のプロフェッショナルサービスの詳細については AWS Marketplace、「プロフェッショナルサービス製品」を参照してください。
- Amazon Quantum Solutions Lab (QSL): QSL は、量子コンピューティングを効果的に学習し、この技術の現在の性能を評価するお手伝いができる量子コンピューティングのエキスパートがいる、共同研究およびプロフェッショナルサービスのチームです。
  - QSL に問い合わせるには、[お問い合わせ] を選択し、連絡先情報とユースケースの詳細を入力します。

- QSL チームから、後続のステップについて E メールで連絡されます。

OpenQASM 3.0 での回路の実行

Amazon Braket は、ゲートベースの量子デバイスとシミュレーターの OpenQASM 3.0 をサポートするようになりました。このユーザーガイドでは、Braket でサポートされている OpenQASM 3.0 のサブセットについて説明します。Braket のお客様は、Braket 回路を送信するのに、SDK を使用か、Amazon Braket API と Amazon Braket Python SDK を使用して OpenQASM 3.0 文字列をすべてのゲートベースのデバイスに直接提供するかを選択できるようになりました。

このガイドのトピックでは、以下の量子タスクの完了方法のさまざまな例について説明します。

- さまざまな Braket デバイスで OpenQASM 量子タスクを作成して送信する
- サポートされているオペレーションと結果タイプにアクセスする
- OpenQASM でノイズをシミュレートする
- OpenQASM で逐語的なコンパイルを使用する
- OpenQASM の問題をトラブルシューティングする

また、このガイドでは、Braket の OpenQASM 3.0 で実装できる特定のハードウェア固有機能の概要と、その他のリソースへのリンクについても示します。

このセクションの内容:

- OpenQASM 3.0 とは?
- OpenQASM 3.0 を使用するタイミング
- OpenQASM 3.0 の仕組み
- 前提条件
- Braket はどのような OpenQASM 機能をサポートしていますか?
- OpenQASM 3.0 量子タスクの例を作成して送信する
- さまざまな Braket デバイスでの OpenQASM のサポート
- OpenQASM 3.0 でノイズをシミュレートする
- OpenQASM 3.0 を使用したQubitの再配線
- OpenQASM 3.0 を使用した逐語的なコンパイル
- Braket コンソール

- その他のリソース
- OpenQASM 3.0 を使用した勾配の計算
- OpenQASM 3.0 を使用した特定の量子ビットの測定

OpenQASM 3.0 とは?

Open Quantum Assembly Language (OpenQASM) は、量子命令の中間表現です。OpenQASM はオープンソースフレームワークであり、ゲートベースのデバイス用の量子プログラムの仕様に広く使用されています。OpenQASM を使用すると、ユーザーは量子計算の構成要素を形成する量子ゲートと測定操作をプログラムできます。以前のバージョンの OpenQASM (2.0) は、基本的なプログラムを記述するために多くの量子プログラミングライブラリで使用されました。

OpenQASM (3.0) の新しいバージョンでは、以前のバージョンを拡張して、パルスレベルの制御、ゲートのタイミング、古典的な制御フローなどの機能を追加することで、エンドユーザーインターフェイスとハードウェア記述言語間のギャップを埋めています。現在のバージョン 3.0 の詳細と仕様については、GitHub の「OpenQASM 3.x Live Specification」を参照してください。OpenQASM の将来の開発はOpenQASM 3.0 テクニカルステアリング委員会によって管理されます。この委員会 AWS は、IBM、Microsoft、およびインスブルック大学のメンバーです。

OpenQASM 3.0 を使用するタイミング

OpenQASM は、アーキテクチャ固有ではない低レベルな制御機能を通じて量子プログラムを指定する表現力豊かなフレームワークを提供するため、複数のゲートベースのデバイスにわたる表現として最適です。Braket による OpenQASM のサポートにより、OpenQASM がゲートベースの量子アルゴリズムの開発に対する一貫したアプローチとしてますます採用されるため、ユーザーが複数のフレームワークでライブラリを学習して維持する必要性が減ります。

OpenQASM 3.0 に既存のプログラムライブラリをお持ちの場合は、それらの回路を完全に書き換えるのではなく、Braket で使用するように調整できます。また、研究者やデベロッパーは、OpenQASM でのアルゴリズム開発をサポートする利用可能なサードパーティーライブラリの数が増えていることからもメリットを得られます。

OpenQASM 3.0 の仕組み

Braket による OpenQASM 3.0 のサポートにより、現在の中間表現と同等の機能が提供されます。つまり、Braket を使用してハードウェアデバイスやオンデマンドシミュレーターで現在できていることが、すべて Braket API を使用して OpenQASM でも実行できるようになったということで

す。OpenQASM 3.0 プログラムを実行するには、OpenQASM 文字列をすべてのゲートベースのデバイスに直接供給します。これは、回路が現在 Braket でデバイスに供給されている方法に似ています。また、Braket ユーザーは、OpenQASM 3.0 をサポートするサードパーティーライブラリを統合することもできます。このガイドの残りの部分では、Braket で使用する OpenQASM 表現を開発する方法について説明します。

前提条件

Amazon Braket で OpenQASM 3.0 を使用するには、Amazon Braket Python スキーマのバージョン v1.8.0 および Amazon Braket Python SDK のバージョン v1.17.0 以降が必要です。

Amazon Braket を初めて使用する場合は、Amazon Braket を有効にする必要があります。手順については、「Amazon Braket を有効にする」を参照してください。

Braket はどのような OpenQASM 機能をサポートしていますか?

以下のセクションでは、Braket でサポートされている OpenQASM 3.0 のデータ型、ステートメント、およびプラグマ命令を示します。

このセクションの内容:

- サポートされている OpenQASM データ型
- サポートされている OpenQASM ステートメント
- Braket OpenQASM プラグマ
- LocalSimulator での OpenQASM の高度な機能のサポート
- OpenPulse でサポートされているオペレーションと文法

サポートされている OpenQASM データ型

Amazon Braket では以下の OpenQASM データ型がサポートされています。

- 非負の整数は (仮想および物理) 量子ビットのインデックスに使用されます。
  - `cnot q[0], q[1];`
  - `h $0;`
- 浮動小数点数や定数は、ゲート回転角度に使用できます。
  - `rx(-0.314) $0;`
  - `rx(pi/4) $0;`

> **Note**
>
> pi は OpenQASM の組み込み定数であり、パラメータ名として使用することはできません。

- 複素数の配列 (虚部については OpenQASM では「im」と表記) は、一般エルミートオブザーバブルを定義するための結果タイププラグマ、およびユニタリプラグマで使用できます。
  - `#pragma braket unitary [[0, -1im], [1im, 0]] q[0]`
  - `#pragma braket result expectation hermitian([[0, -1im], [1im, 0]]) q[0]`

サポートされている OpenQASM ステートメント

Amazon Braket では、以下の OpenQASM ステートメントがサポートされています。

- `Header: OPENQASM 3;`
- 古典ビット宣言:
  - `bit b1;` (等価: `creg b1;`)
  - `bit[10] b2;` (等価: `creg b2[10];`)
- 量子ビット宣言:
  - `qubit b1;` (等価: `qreg b1;`)
  - `qubit[10] b2;` (等価: `qreg b2[10];`)
- 配列内のインデックス作成: `q[0]`
- 入力: `input float alpha;`
- 物理qubitsの指定: `$0`
- デバイスでサポートされているゲートとオペレーション:
  - `h $0;`
  - `iswap q[0], q[1];`

> **Note**
>
> デバイスでサポートされているゲートは、OpenQASM アクションのデバイスプロパティにあります。これらのゲートを使用するのにゲート定義は必要ありません。

- 逐語的なボックスステートメント。現在、ボックス期間表記はサポートされていません。逐語的なボックスには、ネイティブゲートと物理 qubits が必要です。

```
#pragma braket verbatim
box{
    rx(0.314) $0;
}
```

- qubitsまたはqubitレジスタ全体の測定と測定の割り当て。
  - `measure $0;`
  - `measure q;`
  - `measure q[0];`
  - `b = measure q;`
  - `measure q # b;`
- 障壁ステートメントは、障壁の境界を越えたゲートの順序変更と最適化を防止することで、回路のコンパイルと実行を明示的に制御します。また、実行中に厳密な時間順序を適用し、後続のオペレーションを開始する前に障壁が完了する前にすべてのオペレーションを確実に実行します。
  - `barrier;`
  - `barrier q[0], q[1];`
  - `barrier $3, $6;`

Braket OpenQASM プラグマ

Amazon Braket でサポートされている OpenQASM プラグマ命令は、以下のとおりです。

- ノイズプラグマ
  - `#pragma braket noise bit_flip(0.2) q[0]`
  - `#pragma braket noise phase_flip(0.1) q[0]`
  - `#pragma braket noise pauli_channel`
- 逐語的なプラグマ
  - `#pragma braket verbatim`
- 結果タイププラグマ
  - 基底不変な結果タイプ

    - 状態ベクトル: `#pragma braket result state_vector`
    - 密度マトリックス: <code>#pragma braket result density_matrix</code>
- 勾配計算プラグマ:
    - 随伴勾配: <code>#pragma braket result adjoint_gradient expectation(2.2 * x[0] @ x[1]) all</code>
- Z基底による結果タイプ:
    - 振幅: `#pragma braket result amplitude "01"`
    - 確率: `#pragma braket result probability q[0], q[1]`
- 基底回転結果タイプ
    - 期待値: `#pragma braket result expectation x(q[0]) @ y([q1])`
    - 分散: `#pragma braket result variance hermitian([[0, -1im], [1im, 0]]) $0`
    - 標本: `#pragma braket result sample h($1)`

> **Note**
>
> OpenQASM 3.0 は OpenQASM 2.0 と下位互換性があるため、2.0 を使用して記述されたプログラムは Braket で実行できます。ただし、Braket でサポートされている OpenQASM 3.0 の機能には、`qreg` と `creg` の違いや `qubit` と `bit` の違いなど、構文の小さな違いがあります。測定構文にも違いがあり、これらの違いには正しい構文で対応する必要があります。

LocalSimulator での OpenQASM の高度な機能のサポート

`LocalSimulator` は、Braket の QPU またはオンデマンドシミュレーターの構成要素として提供されていない高度な OpenQASM 機能をサポートしています。`LocalSimulator` でのみサポートされている機能を次にリストします。

- ゲート修飾子
- OpenQASM 組み込みゲート
- クラシカル変数
- クラシカルオペレーション
- カスタムゲート
- クラシカル制御

- QASM ファイル
- サブルーチン

高度な各機能の例については、「sample notebook」を参照してください。OpenQASM の完全な仕様については、OpenQASM のウェブサイトを参照してください。

OpenPulse でサポートされているオペレーションと文法

サポートされている OpenPulse データ型

Cal ブロック:

```
cal {
    ...
}
```

Defcal ブロック:

```
// 1 qubit
defcal x $0 {
...
}

// 1 qubit w. input parameters as constants
defcal my_rx(pi) $0 {
...
}

// 1 qubit w. input parameters as free parameters
defcal my_rz(angle theta) $0 {
...
}

// 2 qubit (above gate args are also valid)
defcal cz $1, $0 {
...
}
```

フレーム (frame):

```
frame my_frame = newframe(port_0, 4.5e9, 0.0);
```

波形 (waveform):

```
// prebuilt
waveform my_waveform_1 = constant(1e-6, 1.0);

//arbitrary
waveform my_waveform_2 = {0.1 + 0.1im, 0.1 + 0.1im, 0.1, 0.1};
```

カスタムゲートキャリブレーションの例:

```
cal {
    waveform wf1 = constant(1e-6, 0.25);
}

defcal my_x $0 {
   play(wf1, q0_rf_frame);
}

defcal my_cz $1, $0 {
    barrier q0_q1_cz_frame, q0_rf_frame;
    play(q0_q1_cz_frame, wf1);
    delay[300ns] q0_rf_frame
    shift_phase(q0_rf_frame, 4.366186381749424);
    delay[300ns] q0_rf_frame;
    shift_phase(q0_rf_frame.phase, 5.916747563126659);
    barrier q0_q1_cz_frame, q0_rf_frame;
    shift_phase(q0_q1_cz_frame, 2.183093190874712);
}

bit[2] ro;
my_x $0;
my_cz $1,$0;
c[0] = measure $0;
```

任意パルスの例:

```
bit[2] ro;
cal {
    waveform wf1 = {0.1 + 0.1im, 0.1 + 0.1im, 0.1, 0.1};
    barrier q0_drive, q0_q1_cross_resonance;
    play(q0_q1_cross_resonance, wf1);
    delay[300ns] q0_drive;
```

```
    shift_phase(q0_drive, 4.366186381749424);
    delay[300dt] q0_drive;
   barrier q0_drive, q0_q1_cross_resonance;
   play(q0_q1_cross_resonance, wf1);
    ro[0] = capture_v0(r0_measure);
    ro[1] = capture_v0(r1_measure);
}
```

OpenQASM 3.0 量子タスクの例を作成して送信する

Amazon Braket Python SDK、Boto3、または を使用してOpenQASM 3.0 量子タスクを Amazon Braket デバイス AWS CLI に送信できます。

このセクションの内容:

- OpenQASM 3.0 プログラムの例
- Python SDK を使用して OpenQASM 3.0 量子タスクを作成する
- Boto3 を使用して OpenQASM 3.0 量子タスクを作成する
- を使用して OpenQASM 3.0 タスク AWS CLI を作成する

OpenQASM 3.0 プログラムの例

OpenQASM 3.0 タスクを作成するには、まず、次の例に示されている、GHZ 状態を作成する基本的な OpenQASM 3.0 プログラム (ghz.qasm) を使用します。

```
// ghz.qasm
// Prepare a GHZ state
OPENQASM 3;

qubit[3] q;
bit[3] c;

h q[0];
cnot q[0], q[1];
cnot q[1], q[2];

c = measure q;
```

Python SDK を使用して OpenQASM 3.0 量子タスクを作成する

上記のプログラムを Amazon Braket デバイスに送信するには、Amazon Braket Python SDK と次のコードを使用します。例内の Amazon S3 バケットロケーション内の「amzn-s3-demo-bucket」を独自の Amazon S3 バケット名に置き換えてください。

```
with open("ghz.qasm", "r") as ghz:
    ghz_qasm_string = ghz.read()

# Import the device module
from braket.aws import AwsDevice
# Choose the Rigetti device
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
from braket.ir.openqasm import Program

program = Program(source=ghz_qasm_string)
my_task = device.run(program)

# Specify an optional s3 bucket location and number of shots
s3_location = ("amzn-s3-demo-bucket", "openqasm-tasks")
my_task = device.run(
    program,
    s3_location,
    shots=100,
)
```

Boto3 を使用して OpenQASM 3.0 量子タスクを作成する

量子タスクを作成するには、次の例に示すように AWS Python SDK for Braket (Boto3) と OpenQASM 3.0 文字列を使用する方法もあります。次のコードスニペットは、上に示した、GHZ 状態を作成する ghz.qasm を参照しています。

```
import boto3
import json

my_bucket = "amzn-s3-demo-bucket"
s3_prefix = "openqasm-tasks"

with open("ghz.qasm") as f:
    source = f.read()

action = {
```

```
    "braketSchemaHeader": {
        "name": "braket.ir.openqasm.program",
        "version": "1"
    },
    "source": source
}
device_parameters = {}
device_arn = "arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3"
shots = 100

braket_client = boto3.client('braket', region_name='us-west-1')
rsp = braket_client.create_quantum_task(
    action=json.dumps(
        action
    ),
    deviceParameters=json.dumps(
        device_parameters
    ),
    deviceArn=device_arn,
    shots=shots,
    outputS3Bucket=my_bucket,
    outputS3KeyPrefix=s3_prefix,
)
```

を使用して OpenQASM 3.0 タスク AWS CLI を作成する

OpenQASM 3.0 プログラムを送信するには、次の例に示すように AWS Command Line Interface (CLI) を使用する方法もあります。

```
aws braket create-quantum-task \
    --region "us-west-1" \
    --device-arn "arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3" \
    --shots 100 \
    --output-s3-bucket "amzn-s3-demo-bucket" \
    --output-s3-key-prefix "openqasm-tasks" \
    --action '{
        "braketSchemaHeader": {
            "name": "braket.ir.openqasm.program",
            "version": "1"
        },
        "source": $(cat ghz.qasm)
    }'
```

さまざまな Braket デバイスでの OpenQASM のサポート

OpenQASM 3.0 をサポートするデバイスの場合、[action] フィールドは、 Rigetti および IonQ デバイスの次の例に示すように、GetDevice レスポンスによって新しいアクションをサポートします。

```
//OpenQASM as available with the Rigetti device capabilities
{
    "braketSchemaHeader": {
        "name": "braket.device_schema.rigetti.rigetti_device_capabilities",
        "version": "1"
    },
    "service": {...},
    "action": {
        "braket.ir.jaqcd.program": {...},
        "braket.ir.openqasm.program": {
            "actionType": "braket.ir.openqasm.program",
            "version": [
                "1"
            ],
            ….
        }
    }
}

//OpenQASM as available with the IonQ device capabilities
{
    "braketSchemaHeader": {
        "name": "braket.device_schema.ionq.ionq_device_capabilities",
        "version": "1"
    },
    "service": {...},
    "action": {
        "braket.ir.jaqcd.program": {...},
        "braket.ir.openqasm.program": {
            "actionType": "braket.ir.openqasm.program",
            "version": [
                "1"
            ],
            ….
        }
    }
}
```

パルス制御をサポートするデバイスの場合、[pulse] フィールドが [GetDevice] レスポンスに表示されます。次の例は、Rigetti デバイスの [pulse] フィールドを示しています。

```
// Rigetti
{
  "pulse": {
    "braketSchemaHeader": {
      "name": "braket.device_schema.pulse.pulse_device_action_properties",
      "version": "1"
    },
    "supportedQhpTemplateWaveforms": {
      "constant": {
        "functionName": "constant",
        "arguments": [
          {
            "name": "length",
            "type": "float",
            "optional": false
          },
          {
            "name": "iq",
            "type": "complex",
            "optional": false
          }
        ]
      },
      ...
    },
    "ports": {
      "q0_ff": {
        "portId": "q0_ff",
        "direction": "tx",
        "portType": "ff",
        "dt": 1e-9,
        "centerFrequencies": [
          375000000
        ]
      },
      ...
    },
    "supportedFunctions": {
      "shift_phase": {
        "functionName": "shift_phase",
```

```
        "arguments": [
          {
            "name": "frame",
            "type": "frame",
            "optional": false
          },
          {
            "name": "phase",
            "type": "float",
            "optional": false
          }
        ]
      },
     ...
    },
    "frames": {
      "q0_q1_cphase_frame": {
        "frameId": "q0_q1_cphase_frame",
        "portId": "q0_ff",
        "frequency": 462475694.24460185,
        "centerFrequency": 375000000,
        "phase": 0,
        "associatedGate": "cphase",
        "qubitMappings": [
          0,
          1
        ]
      },
      ...
    },
    "supportsLocalPulseElements": false,
    "supportsDynamicFrames": false,
    "supportsNonNativeGatesWithPulses": false,
    "validationParameters": {
      "MAX_SCALE": 4,
      "MAX_AMPLITUDE": 1,
      "PERMITTED_FREQUENCY_DIFFERENCE": 400000000
    }
  }
}
```

以下のようなフィールドが先行しています。

ポート (ports):

指定したポートの関連プロパティに加えて、QPU で宣言されている事前作成の外部 (extern) デバイスポートを記述するものです。この構造にリストされているすべてのポートは、ユーザーが送信した OpenQASM 3.0 プログラム内で有効な識別子として事前に宣言されています。ポートの追加プロパティは次のとおりです。

- ポート ID (portId)
  - OpenQASM 3.0 で識別子として宣言されたポート名。
- 方向 (direction)
  - ポートの方向。ドライブポートはパルスを送信し (方向は「tx」)、測定ポートはパルスを受信します (方向は「rx」)。
- ポートタイプ (portType)
  - このポートが担当するアクションのタイプ (ドライブ、キャプチャ、ff [高速磁束制御] など)。
- Dt (dt)
  - 指定したポートの 1 つのサンプル時間ステップを表す秒単位の時間。
- 量子ビットマッピング (qubitMappings)
  - 指定したポートに関連付ける量子ビット。
- 中心周波数 (centerFrequencies)
  - ポート上のすべての事前宣言済みフレームまたはユーザー定義済みのフレームに関連する中心周波数のリスト。詳細については、「フレーム」を参照してください。
- QHP 固有のプロパティ (qhpSpecificProperties)
  - QHP に固有のポートに関する既存のプロパティの詳細を示す、オプションのマップ。

フレーム (frames):

QPU で宣言された事前作成済みの外部フレームと、それらのフレームのプロパティを記述します。この構造にリストされているすべてのフレームは、ユーザーが送信した OpenQASM 3.0 プログラム内で有効な識別子として事前に宣言されています。フレームの追加プロパティは次のとおりです。

- フレーム ID (frameId)
  - OpenQASM 3.0 で識別子として宣言されたフレーム名。
- ポート ID (portId)
  - フレームに関連付けられたハードウェアポート。
- 周波数 (frequency)
  - フレームのデフォルトの初期周波数。

- 中心周波数 (centerFrequency)
  - フレームの周波数帯域幅の中心。通常、フレームは中心周波数の周りの特定の帯域幅にのみ調整できます。この結果、周波数調整は中心周波数からの所定のデルタ内にとどまる必要があります。帯域幅の値は検証パラメータで確認できます。
- 位相 (phase)
  - フレームのデフォルトの初期位相。
- 関連付けるゲート (associatedGate)
  - 指定したフレームに関連付けるゲート。
- 量子ビットマッピング (qubitMappings)
  - 指定したフレームに関連付ける量子ビット。
- QHP 固有のプロパティ (qhpSpecificProperties)
  - QHP に固有のフレームに関する既存のプロパティの詳細を示す、オプションのマップ。

SupportsDynamicFrames:

OpenPulse `newframe` 関数を通じてフレームを `cal` または `defcal` ブロックで宣言できるかどうかを記述するものです。false の場合、フレーム構造にリストされているフレームのみをプログラム内で使用できます。

SupportedFunctions:

デバイスでサポートされている OpenPulse 関数に加えて、それらの関数に関連する引数、引数タイプ、戻り値タイプを記述します。OpenPulse 関数の使用例については、「OpenPulse specification」を参照してください。現時点では、Braket は以下をサポートしています。

- shift_phase
  - フレームの位相を指定した値だけシフトします。
- set_phase
  - フレームの位相を指定した値に設定します。
- swap_phases
  - 2 つのフレーム間で位相をスワップします。
- shift_frequency
  - フレームの周波数を指定した値だけシフトします。
- set_frequency

  - フレームの周波数を指定した値に設定します。
- play
  - 波形をスケジュールします。
- capture_v0
  - キャプチャフレームの値をビットレジスタに返します。

SupportedQhpTemplateWaveforms:

デバイスで使用できる構築済みの波形関数、およびそれらに関連する引数とタイプを記述します。デフォルトでは、Braket Pulse はすべてのデバイスで構築済みの波形ルーチンを提供します。それらのルーチンは以下のとおりです。

Constant

$$Constant(t, \tau, iq) = iq$$

τ は波形の長さで、iq は複素数です。

```
def constant(length, iq)
```

ガウシアン

$$Gaussian(t, \tau, \sigma, A = 1, ZaE = 0) = \frac{A}{1 - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)}\left[\exp\left(-\frac{1}{2}\left(\frac{t-\frac{\tau}{2}}{\sigma}\right)^2\right) - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)\right]$$

τは波形の長さ、σはガウシアンの幅、A は振幅です。ZaE を True に設定すると、ガウシアンにオフセットが生じ、波形の開始時と終了時にゼロに等しくなるように再スケールされ、最大で A になります。

```
def gaussian(length, sigma, amplitude=1, zero_at_edges=False)
```

DRAG ガウシアン

$$DRAG\_Gaussian(t, \tau, \sigma, \beta, A = 1, ZaE = 0) = \frac{A}{1 - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)}\left(1 - i\beta\frac{t-\frac{\tau}{2}}{\sigma^2}\right)\left[\exp\left(-\frac{1}{2}\left(\frac{t-\frac{\tau}{2}}{\sigma}\right)^2\right) - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)\right]$$

τは波形の長さ、σはガウシアンの幅、βは自由パラメータ、A は振幅です。ZaE を True に設定すると、断熱ゲートによる微分除去 (DRAG) ガウシアンにオフセットが生じ、波形の開始時と終了時にゼロに等しくなるように再スケールされ、実部が最大で A になります。DRAG 波形の詳細については、文書「Simple Pulses for Elimination of Leakage in Weakly Nonlinear Qubits」を参照してください。

```
def drag_gaussian(length, sigma, beta, amplitude=1, zero_at_edges=False)
```

誤差関数平滑化方形パルス

$$\text{Erf\_Square}(t, L, W, \sigma, A = 1, ZaE = 0) =$$

$$A \times \frac{\text{erf}((t - t_1)/\sigma) + \text{erf}(-(t - t_2)/\sigma)}{2 \times \text{erf}(W/2\sigma))}$$

ここで、L と W はそれぞれ波形の長さと幅、σはエッジの立ち上がりと立ち下がりの速度、t_1=(L−W)/2、$t_2$2=(L+W)/2、A は振幅です。ZaE を True に設定すると、ガウシアンにオフセットが生じ、波形の開始時と終了時にゼロに等しくなるように再スケールされ、最大で A になります。次の式は、波形の再スケールされたバージョンです。

$$\text{Erf\_Square}(..., ZaE = 1) = (a \times \text{Erf\_Square}(..., ZaE = 0) - bA)/(a - b)$$

ここで、a=erf(W/2σ) であり、b=erf(-t_1/σ)/2+erf(t $_2$/σ)/2 です。

```
def erf_square(length, width, sigma, amplitude=1, zero_at_edges=False)
```

SupportsLocalPulseElements:

ポート、フレーム、波形などのパルス要素をdefcal ブロックでローカルに定義できるかどうかを記述します。値が false の場合、要素は cal ブロックで定義する必要があります。

SupportsNonNativeGatesWithPulses:

パルスプログラムと組み合わせて非ネイティブゲートを使用できるかどうかを記述します。例えば、最初に使用済み量子ビットに対して defcal を使用してゲートを定義することなく、プログラ

ム内で H ゲートのような非ネイティブゲートを使用することはできません。ネイティブゲートの nativeGateSet キーのリストは、[デバイス機能] にあります。

ValidationParameters:

以下を含む、パルス要素の検証境界を記述します。

- 波形の最大スケール/最大振幅値 (任意および構築済み)
- Hz 単位で指定された中心周波数の最大周波数帯域幅
- 最小パルス長/持続時間 (秒単位)
- 最小パルス長/持続時間 (秒単位)

OpenQASM でサポートされているオペレーション、結果、結果タイプ

各デバイスがサポートする OpenQASM 3.0 の機能を確認するには、デバイス機能の出力で [action] フィールド内の braket.ir.openqasm.program キーを参照してください。例えば、Braket 状態ベクトルシミュレーター SV1 でサポートされているオペレーションと結果タイプを次に示します。

```
...
  "action": {
    "braket.ir.jaqcd.program": {
      ...
    },
 "braket.ir.openqasm.program": {
      "version": [
        "1.0"
      ],
      "actionType": "braket.ir.openqasm.program",
      "supportedOperations": [
        "ccnot",
        "cnot",
        "cphaseshift",
        "cphaseshift00",
        "cphaseshift01",
        "cphaseshift10",
        "cswap",
        "cy",
        "cz",
        "h",
        "i",
```

```
        "iswap",
        "pswap",
        "phaseshift",
        "rx",
        "ry",
        "rz",
        "s",
        "si",
        "swap",
        "t",
        "ti",
        "v",
        "vi",
        "x",
        "xx",
        "xy",
        "y",
        "yy",
        "z",
        "zz"
      ],
      "supportedPragmas": [
        "braket_unitary_matrix"
      ],
      "forbiddenPragmas": [],
      "maximumQubitArrays": 1,
      "maximumClassicalArrays": 1,
      "forbiddenArrayOperations": [
        "concatenation",
        "negativeIndex",
        "range",
        "rangeWithStep",
        "slicing",
        "selection"
      ],
      "requiresAllQubitsMeasurement": true,
      "supportsPhysicalQubits": false,
      "requiresContiguousQubitIndices": true,
      "disabledQubitRewiringSupported": false,
      "supportedResultTypes": [
        {
          "name": "Sample",
          "observables": [
            "x",
```

```
            "y",
            "z",
            "h",
            "i",
            "hermitian"
          ],
          "minShots": 1,
          "maxShots": 100000
        },
        {
          "name": "Expectation",
          "observables": [
            "x",
            "y",
            "z",
            "h",
            "i",
            "hermitian"
          ],
          "minShots": 0,
          "maxShots": 100000
        },
        {
          "name": "Variance",
          "observables": [
            "x",
            "y",
            "z",
            "h",
            "i",
            "hermitian"
          ],
          "minShots": 0,
          "maxShots": 100000
        },
        {
          "name": "Probability",
          "minShots": 1,
          "maxShots": 100000
        },
        {
          "name": "Amplitude",
          "minShots": 0,
          "maxShots": 0
```

```
        }
        {
          "name": "AdjointGradient",
          "minShots": 0,
          "maxShots": 0
        }
      ]
    }
  },
...
```

OpenQASM 3.0 でノイズをシミュレートする

OpenQASM3 でノイズをシミュレートするには、プラグマ命令を使用してノイズ作用素を追加します。例えば、前述の GHZ プログラムの、ノイズの多いバージョンをシミュレートするには、次の OpenQASM プログラムを送信します。

```
// ghz.qasm
// Prepare a GHZ state
OPENQASM 3;

qubit[3] q;
bit[3] c;

h q[0];
#pragma braket noise depolarizing(0.75) q[0] cnot q[0], q[1];
#pragma braket noise depolarizing(0.75) q[0]
#pragma braket noise depolarizing(0.75) q[1] cnot q[1], q[2];
#pragma braket noise depolarizing(0.75) q[0]
#pragma braket noise depolarizing(0.75) q[1]

c = measure q;
```

サポートされているすべてのプラグマノイズ作用素の仕様を次のリストに示します。

```
#pragma braket noise bit_flip(<float in [0,1/2]>) <qubit>
#pragma braket noise phase_flip(<float in [0,1/2]>) <qubit>
#pragma braket noise pauli_channel(<float>, <float>, <float>)  <qubit>
#pragma braket noise depolarizing(<float in [0,3/4]>) <qubit>
#pragma braket noise two_qubit_depolarizing(<float in [0,15/16]>) <qubit>, <qubit>
#pragma braket noise two_qubit_dephasing(<float in [0,3/4]>) <qubit>, <qubit>
#pragma braket noise amplitude_damping(<float in [0,1]>) <qubit>
```

```
#pragma braket noise generalized_amplitude_damping(<float in [0,1]> <float in [0,1]>)
 <qubit>
#pragma braket noise phase_damping(<float in [0,1]>) <qubit>
#pragma braket noise kraus([[<complex m0_00>, ], ...], [[<complex m1_00>, ], ...], ...)
 <qubit>[, <qubit>]     // maximum of 2 qubits and maximum of 4 matrices for 1 qubit,
 16 for 2
```

Kraus 作用素

Kraus 作用素を生成するには、マトリックスのリストを繰り返し実行し、マトリックスの各要素を複素数表現として出力します。

Kraus 作用素を使用する場合は、次の点に注意してください。

- qubitsの数は 2 を超えることはできません。この制限は、スキーマの現在の定義によって設定されています。
- 引数リストの長さは 8 の倍数である必要があります。つまり、2x2 マトリックスのみで構成する必要があります。
- 作用素マトリックス数の合計が 2^{2*num_qubits} 個を超えることはありません。つまり、1 qubitには 4 個のマトリックス、2 qubitsには 16 個のマトリックスという制限があります。
- 提供されたすべてのマトリックスは、完全に正なトレース保存 (CPTP) の条件を満たしています。
- Kraus 作用素と転置共役の積は、単位マトリックスになる必要があります。

OpenQASM 3.0 を使用したQubitの再配線

Amazon Braket は、Rigettiデバイスでの OpenQASM 内の物理qubit表記をサポートしています (詳細については、こちらのページを参照してください)。ナイーブ再配線戦略で物理qubitsを使用する場合は、選択したデバイスでqubitsが接続されていることを確認します。または、代わりにqubitレジスタを使用する場合は、Rigetti デバイスでは、PARTIAL (部分的) 再配線戦略がデフォルトで有効になります。

```
// ghz.qasm
// Prepare a GHZ state
OPENQASM 3;

h $0;
cnot $0, $1;
cnot $1, $2;
```

```
measure $0;
measure $1;
measure $2;
```

OpenQASM 3.0 を使用した逐語的なコンパイル

Rigetti や IonQ などのベンダーで提供される量子コンピュータで量子回路を実行する場合、変更を加えることなく、回路を定義したとおりに正確に実行するようにコンパイラーに指示することができます。この機能は逐語的なコンパイルと呼ばれます。Rigetti デバイスを使用すると、回路全体または回路の特定部分のみで、保持されるものを正確に指定できます。回路の特定部分のみを保持するには、保持対象のリージョン内でネイティブゲートを使用する必要があります。現在、IonQ は回路全体の逐語的なコンパイルのみをサポートしているため、回路内のすべての命令を逐語的なボックスで囲む必要があります。

OpenQASM では、コードボックスの周囲に逐語的なプラグマを明示的に指定できます。このプラグマは、ハードウェアの低レベルコンパイルルーチンでは変更されず、最適化されません。次のコード例は、 `#pragma braket verbatim` ディレクティブを使用してこれを実現する方法を示しています。

```
OPENQASM 3;

bit[2] c;

#pragma braket verbatim
box{
    rx(0.314159) $0;
    rz(0.628318) $0, $1;
    cz $0, $1;
}

c[0] = measure $0;
c[1] = measure $1;
```

例やベストプラクティスなど、逐語的なコンパイルプロセスの詳細については、amazon-braket-examples github リポジトリにある Verbatim compilation サンプルノートブックを参照してください。

Braket コンソール

OpenQASM 3.0 タスクが Amazon Braket コンソールで利用、管理できます。このコンソールでは、既存の量子タスクを送信する際に持ったのと同じ量子タスク送信経験を OpenQASM 3.0 で持つことができます。

その他のリソース

OpenQASM はすべての Amazon Braket リージョンで利用できます。

Amazon Braket での OpenQASM の使用を開始するためのノートブックの例については、「Braket Tutorials GitHub」を参照してください。

OpenQASM 3.0 を使用した勾配の計算

Amazon Braket は、[shots=0] (正確) モードで実行する場合、オンデマンドシミュレーターとローカルシミュレーターの両方で勾配の計算をサポートします。これは、随伴微分法を使用して実現されます。計算する勾配を指定するには、次の例内のコードに示されているような、適切なプラグマを指定します。

```
OPENQASM 3.0;
input float alpha;

bit[2] b;
qubit[2] q;

h q[0];
h q[1];
rx(alpha) q[0];
rx(alpha) q[1];
b[0] = measure q[0];
b[1] = measure q[1];

#pragma braket result adjoint_gradient h(q[0]) @ i(q[1]) alpha
```

また、個々のパラメータを明示的にリストする代わりに、プラグマ内で all キーワードを指定することもできます。これにより、リストされているすべての input パラメータに関する勾配が計算されます。つまり、all は、パラメータの数が非常に多い場合に便利なキーワードです。この場合、次の例内のコードのようなプラグマを指定することになります。

```
#pragma braket result adjoint_gradient h(q[0]) @ i(q[1]) all
```

Amazon Braket の OpenQASM 3.0 実装では、個々の作用素、テンソル積、エルミートオブザーバブル、Sum オブザーバブルなど、すべてのオブザーバブルタイプがサポートされています。勾配を計算するときに使用する特定の作用素は expectation() 関数内にラッピングする必要があり、オブザーバブルの各項が作用する量子ビットを明示的に指定する必要があります。

OpenQASM 3.0 を使用した特定の量子ビットの測定

Amazon Braket が提供するローカル状態ベクトルシミュレーターとローカル密度マトリックスシミュレーターは、回路の量子ビットのサブセットを選択的に測定できる OpenQASM プログラムを送信することができます。この機能は部分測定とも呼ばれ、よりターゲットを絞った効率的な量子計算を可能にします。例えば、次のコードス二ペットでは、2 量子ビットの回路を作成し、2 番目の量子ビットを測定せずに最初の量子ビットのみを測定できます。

```
partial_measure_qasm = """
OPENQASM 3.0;
bit[1] b;
qubit[2] q;
h q[0];
cnot q[0], q[1];
b[0] = measure q[0];
"""
```

この例には q[0] と q[1] の 2 つの量子ビットを持つ量子回路がありますが、ここでは最初の量子ビットの状態のみを測定するとします。これは、qubit[0] の状態を測定し、その結果を古典ビット b[0] に保存する b[0] = measure q[0] の行によって実現されます。この部分測定のシナリオを実行するには、Amazon Braket が提供するローカル状態ベクトルシミュレーターで次のコードを実行します。

```
from braket.devices import LocalSimulator

local_sim = LocalSimulator()
partial_measure_local_sim_task =
 local_sim.run(OpenQASMProgram(source=partial_measure_qasm), shots = 10)
partial_measure_local_sim_result = partial_measure_local_sim_task.result()
print(partial_measure_local_sim_result.measurement_counts)
print("Measured qubits: ", partial_measure_local_sim_result.measured_qubits)
```

デバイスで部分測定がサポートされているかどうかは、[requiresAllQubitsMeasurement] フィールドの action のプロパティを調べることで確認できます。そのプロパティが False の場合は、部分測定がサポートされています。

```
from braket.devices import Devices

AwsDevice(Devices.Rigetti.Ankaa3).properties.action['braket.ir.openqasm.program'].requiresAllQu
```

この場合、 requiresAllQubitsMeasurement は False です。これは、必ずしもすべての量子ビットを測定する必要がないことを示しています。

実験機能を探索する

実験的な機能により、可用性が制限され、新しいソフトウェア機能が新たに導入されたハードウェアにアクセスできます。これらの機能は、標準仕様を超えるデバイスのパフォーマンスに影響を与える可能性があります。Amazon Braket SDK を使用して、タスクごとに実験的なソフトウェア機能を自動的に有効にできます。

実験機能を使用するには、量子タスクを作成するときに experimental_capabilitiesパラメータを指定します。このパラメータを に設定して"ALL"、そのタスクで使用可能なすべての実験機能を有効にします。次の例は、デバイスで回路を実行するときに実験機能を有効にする方法を示しています。

```
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/quera/Aquila")

task = device.run(
   circuit,
   shots=1000,
   experimental_capabilities="ALL"
)
```

> **Note**
>
> これらの機能は実験的なものであり、予告なしに変更される可能性があります。デバイスのパフォーマンスは公開されている仕様とは異なる場合があり、結果は標準オペレーションとは異なる場合があります。タスクごとに実験機能を明示的に有効にする必要があります。このパラメータのないタスクでは、標準のデバイス機能のみが使用されます。

このセクションの内容:

- QuEra Aquila でのローカルデチューニングへのアクセス
- QuEra Aquila のトールなジオメトリへのアクセス
- QuEra Aquila でのタイトなジオメトリへのアクセス
- IQM デバイスの動的回路

QuEra Aquila でのローカルデチューニングへのアクセス

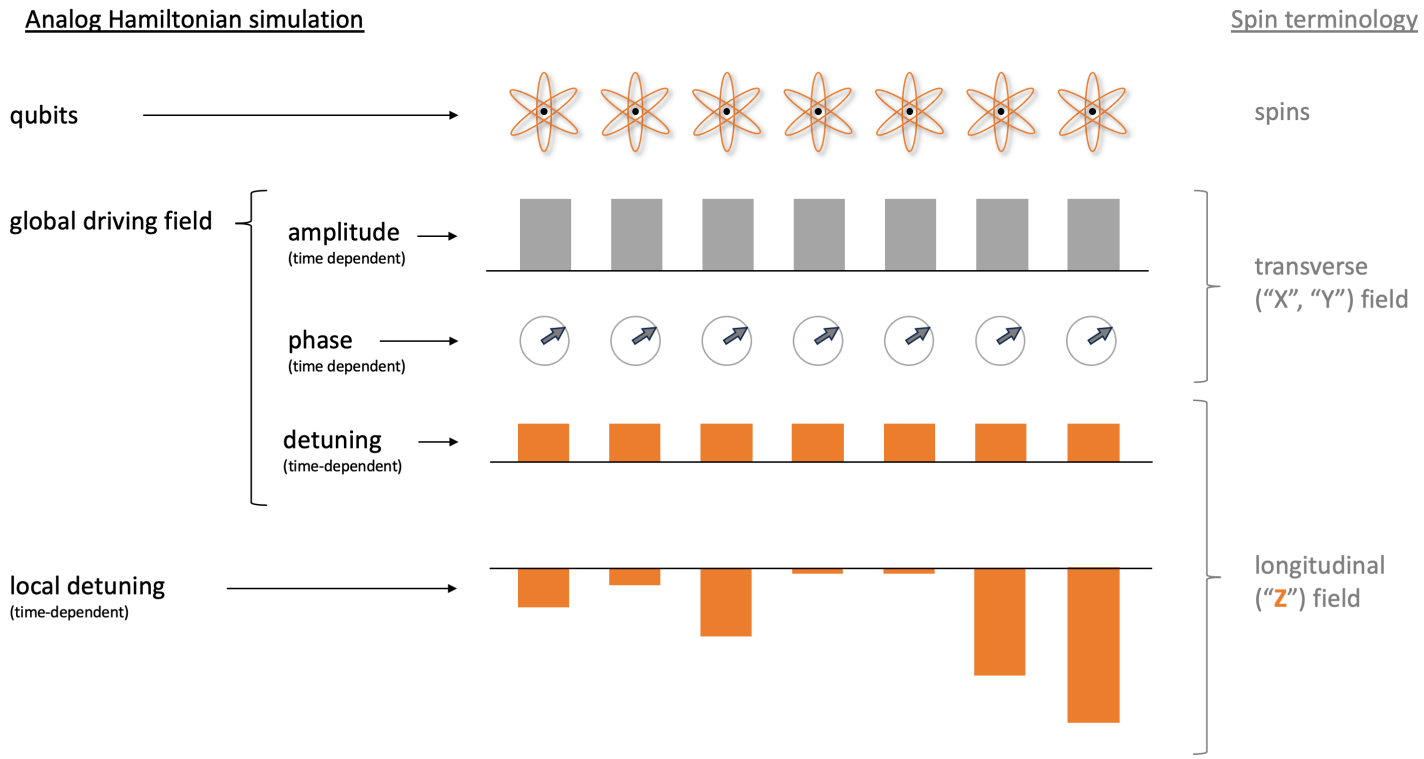
ローカルデチューニング (LD) は、カスタマイズ可能な空間パターンを持つ新しい時間依存制御フィールドです。LD フィールドは、カスタマイズ可能な空間パターンに従って量子ビットに作用します。また、均一な駆動場とリュードベリ-リュードベリ相互作用が作成できる量子ビットを超えるさまざまな量子ビットに対して、異なるハミルトニアンを実現します。

制約:

ローカルデチューニング磁場の空間パターンは AHS プログラムごとにカスタマイズできますが、プログラムの実行中においては不変です。ローカルデチューニング磁場の時系列は、ゼロで開始および終了し、すべての値がゼロ以下である必要があります。さらに、ローカルデチューニング磁場のパラメータは数値制約によって制限されます。数値制約は、Braket SDK の特定のデバイスプロパティセクション `aquila_device.properties.paradigm.rydberg.rydbergLocal` に表示されます。

機能制限:

ローカルデチューニング磁場の大きさがハミルトニアンで定数ゼロに設定されている場合でも、ローカルデチューニング磁場を使用する量子プログラムを実行する場合は、デバイスでは、Aquila のプロパティの [performance] セクションに記載されている T2 時間よりも高速なデコヒーレンスが発生します。ローカルデチューニング磁場が不要な場合は、AHS プログラムのハミルトニアンからローカルデチューニング磁場を除外することをお勧めします。



例:

1. スピン系で不均一な縦磁場の作用をシミュレートする

   駆動場の振幅と位相はスピンへの横磁場の作用と同じ作用を量子ビットに与えるのに対し、駆動場のデチューニングとローカルデチューニングの和はスピンへの縦磁場の影響と同様の作用を量子ビットに与えます。ローカルデチューニング磁場を空間的に制御することで、より複雑なスピン系をシミュレートできます。

2. 非平衡初期状態を準備する

   サンプルノートブック「Simulating lattice gauge theory with Rydberg atoms」には、系を Z2 秩序相にアニーリングするときに 9 原子線形配置の中心原子が振動しないようにする方法が示されています。この準備ステップの後、ローカルデチューニング磁場は減少し、AHS プログラムはこの特定の非均衡状態から始まる、系の時間進化をシミュレートし続けます。

3. 加重最適化問題を解決する

   Aquila での MWIS 問題を解決する方法が、サンプルノートブック「Maximum weight independent set」(MWIS) に示されています。単位円板グラフのノードの重みを定義するために、ローカルデチューニング磁場が使用されます。グラフのエッジは、リュードベリ遮断効果によって作成さ

れます。ローカルデチューニング磁場を均一な基底状態から始めて徐々に大きくすると、系が MWIS ハミルトニアンの基底状態に移行することで、問題の解決策が見つかります。

QuEra Aquila のトールなジオメトリへのアクセス

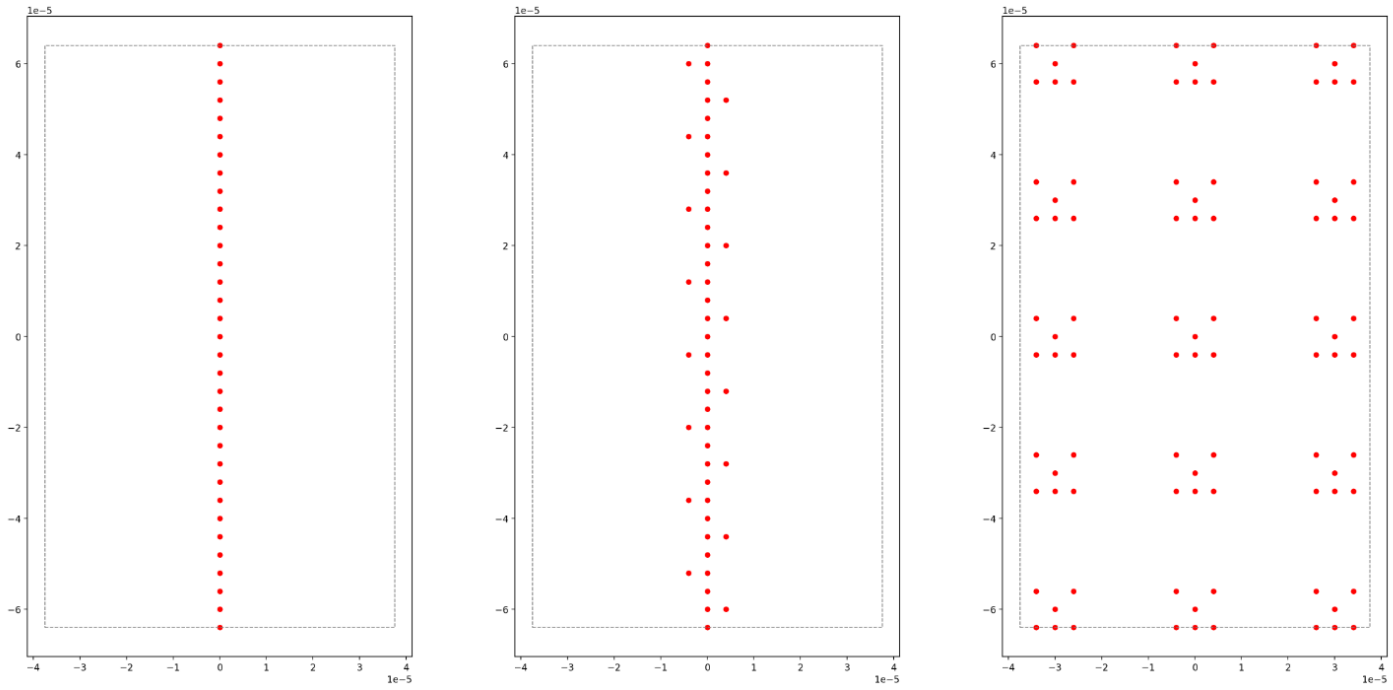
トールなジオメトリの機能を使用すると、高さを大きくしたジオメトリを指定できます。この機能を使用すると、AHS プログラムの原子配置を、Aquila の通常の機能を超える y 軸方向の追加の長さにまたがらせることができます。

制約:

トールなジオメトリの最大高さは 0.000128 m (128 um) です。

機能制限:

アカウントでこの実験機能を有効にすると、デバイスプロパティページと `GetDevice` コールに表示される機能に、高さの通常の下限が引き続き反映されます。AHS プログラムが通常の機能を超える原子配置を使用すると、充填誤差が増大することが予想されます。タスク結果の `pre_sequence` の部分で予期しない 0 の数が増えるため、完全に初期化された配置を得る機会が減ります。このような影響は、多くの原子を持つ行に最も強く現れます。



例:

1. より大きな 1d および準 1d 配置

原子鎖と、はしごのような配置は、より大きな原子数に拡張できます。これらのモデルのより長いインスタンスをプログラミングするには、長い方向を y 軸と平行に配置します。

2. 小さなジオメトリでタスクの実行を多重化するスペースを増やす

   ノートブック例「Parallel quantum tasks on Aquila」は、問題のジオメトリの多重コピーを 1 つの原子配置に配置することで、利用可能な領域を最大限に活用する方法を示しています。利用可能な領域が多いほど、より多くのコピーを配置できます。

QuEra Aquila でのタイトなジオメトリへのアクセス

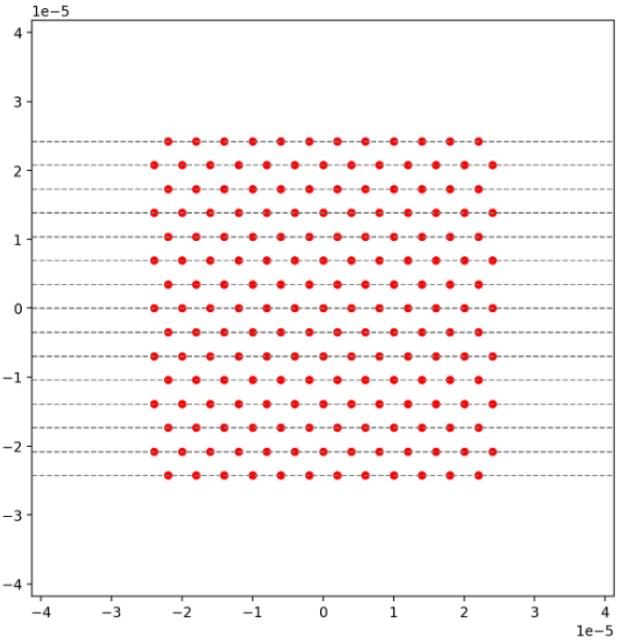
タイトなジオメトリ機能を使用すると、隣接する行間の間隔を短くしてジオメトリを指定できます。AHS プログラムでは、原子は最小の垂直間隔で区切られて行に配置されます。任意の 2 つの原子サイトの y 座標は、ゼロ (同じ行) であるか、最小行間隔よりも大きく異なる (異なる行である)必要があります。タイトなジオメトリ機能により、最小行間隔が小さくなるため、よりタイトな原子配置を作成できます。この拡張機能は、原子間のユークリッド距離の最小要件を変更しませんが、遠くの原子が互いに近い隣接する行を占める格子を作成できます。注目すべき例は三角格子です。

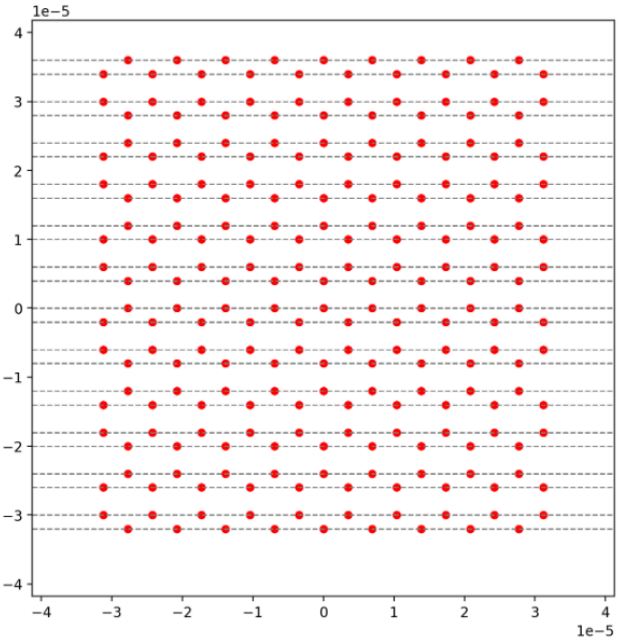
制約:

タイトなジオメトリの最小行間隔は 0.000002 m (2 um) です。

機能制限:

アカウントでこの実験機能を有効にすると、デバイスプロパティページと `GetDevice` コールに表示される機能に、高さの通常の下限が引き続き反映されます。AHS プログラムが通常の機能を超える原子配置を使用すると、充填誤差が増大することが予想されます。タスク結果の `pre_sequence` の部分で予期しない 0 の数が増えるため、完全に初期化された配置を得る機会が減ります。このような影響は、多くの原子を持つ行に最も強く現れます。





例:

1. 小さな格子定数を持つ非矩形格子

   行間隔をタイトにすると、一部の原子に最も近い隣接部分が対角線方向にある格子を作成できます。注目すべき例としては、三角格子、六角格子、カゴメ格子と、いくつかの準結晶があります。

2. 調整可能な格子ファミリー

   AHS プログラムでは、相互作用は原子ペア間の距離を調整することで調整されます。行間隔をタイトにすると、最小行間隔の制約によって原子構造を定義する角度と距離に関する制限が減るため、異なる原子ペア同士の相互作用をより自由に調整できます。注目すべき例は、異なる結合距離を持つ Shastry-Sutherland 格子のファミリーです。

IQM デバイスの動的回路

IQM デバイスの動的回路は、中間回路測定 (MCM) とフィードフォワードオペレーションを可能にします。これらの機能により、量子の研究者や開発者は、条件付きロジックと量子ビット再利用機能を備えた高度な量子アルゴリズムを実装できます。この実験機能により、リソース効率を向上させた量子アルゴリズムを探求し、量子エラー緩和およびエラー修正スキームを研究できます。

重要な命令:

- measure_ff: フィードフォワード制御の測定を実装することで、量子ビットを測定し、結果をフィードバックキーに保存します。
- cc_prx: 指定されたフィードバックキーに関連付けられた結果が |1# 状態を測定した場合にのみ適用される、古典的な方法で制御された回転を実装します。

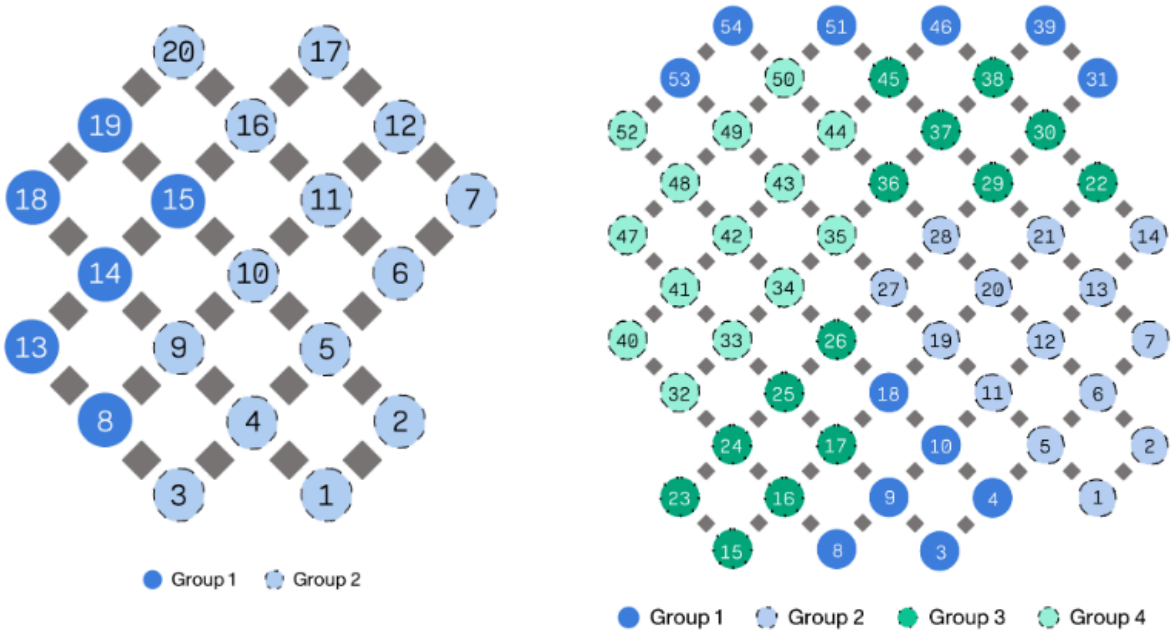
Amazon Braket は、OpenQASM、Amazon Braket SDK、および Amazon Braket Qiskit Provider を使用して動的回路をサポートしています。

制約:

1. measure_ff 命令のフィードバックキーは一意である必要があります。
2. cc_prx は、measure_ff の後で、同じフィードバックキーを使用して実行する必要があります。
3. 単一の回路では、量子ビットに対するフィードフォワードは、その量子ビット自身または別の量子ビットである 1 つの量子ビットによってのみ制御できます。複数の回路では、異なる制御ペアを持つことができます。
   a. 例えば、量子ビット 1 が量子ビット 2 によって制御されている場合、量子ビット 1 は同じ回路の量子ビット 3 によって制御することができません。量子ビット 1 と量子ビット 2 の間に制御が適用される回数に制約 (制限) はありません。量子ビット 2 は、量子ビット 2 に対するアクティブなリセットが実行されない限り、量子ビット 3 (または量子ビット 1) で制御できます。
4. 制御は、同じグループ内の量子ビットにのみ適用できます。IQM Garnet および Emerald デバイスの量子ビットグループは、下記の画像にあります。
5. これらの機能を備えたプログラムは、逐語的なプログラムとして送信する必要があります。逐語的なプログラムの詳細については、「OpenQASM 3.0 を使用した逐語的なコンパイル」を参照してください。

機能制限:

MCM は、プログラム内のフィードフォワード制御にのみ使用できます。MCM の結果 (0 または 1) がタスク結果の一部として返されることはありません。



これらの画像には、両方の IQM デバイスの量子ビットグループが表示されています。Garnet の 20 量子ビットデバイスには 2 つの量子ビットグループが含まれ、Emerald の 54 量子ビットデバイスには 4 つの量子ビットグループが含まれています。

例:

1. アクティブなリセットによる量子ビットの再利用

   条件付きリセットオペレーションを使用する MCM は、1 つの回路実行内で量子ビットを再利用できます。これにより、回路の深さ要件が軽減され、量子デバイスのリソース使用率が向上します。

2. アクティブなビットフリップ保護

   動的回路は、測定結果に基づいて、ビットフリップ誤差を検出して修正オペレーションを適用します。この実装により、量子誤差検出実験を行うことができます。

3. テレポーテーション実験

   状態テレポーテーションは、ローカル量子オペレーションと MCM からの古典情報を使用して量子ビットの状態を転送します。ゲートテレポーテーションでは、量子オペレーションを直接行わずに、量子ビット間のゲートを実装します。この実験には、量子誤差補正、測定ベースの量子コンピューティング、量子通信の 3 つの主要領域における基本的なサブルーチンが含まれています。

4. 開放量子系のシミュレーション

   動的回路は、データ量子ビット、環境のもつれ、環境測定を通じて量子系のノイズをモデル化します。このアプローチでは、データ要素と環境要素を表すのに、特定の量子ビットを使用します。ノイズチャネルは、環境に適用されたゲートと測定値によって設計できます。

動的回路の使用の詳細については、他の例を Amazon Braket notebook repository で参照してください。

Amazon Braket でのパルス制御

パルスとは、量子コンピュータ内の量子ビットを制御するアナログ信号です。Amazon Braket の特定のデバイスでは、パルス制御機能にアクセスし、パルスを使用して回路を送信できます。Braket SDK または OpenQASM 3.0 を使用して、あるいは Braket API から直接、パルス制御にアクセスできます。まず、Braket でのパルス制御の主要概念をいくつか紹介します。

このセクションの内容:

- フレーム
- ポート
- 波形
- Hello Pulse の使用方法
- パルスを使用したネイティブゲートへのアクセス

フレーム

フレームは、量子プログラム内のクロックと位相の両方として機能するソフトウェア抽象化です。操作が特定の周波数で定義され、ステートフルキャリア信号によって行われます。操作するたびに、それにかかる時間がクロック時間として加算されます。量子ビットに信号を送信すると、フレームは量子ビットのキャリア周波数、位相オフセット、および、波形エンベロープが放出される時間を決定します。Braket Pulse では、フレームの構築方法はデバイス、周波数、位相によって異なります。デバイスに応じて、事前定義されたフレームを選択するか、ポートを指定して新しいフレームをインスタンス化できます。

```
from braket.aws import AwsDevice
from braket.pulse import Frame, Port
```

```
# Predefined frame from a device
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
drive_frame = device.frames["Transmon_5_charge_tx"]

# Create a custom frame
readout_frame = Frame(frame_id="r0_measure", port=Port("channel_0", dt=1e-9),
 frequency=5e9, phase=0)
```

ポート

ポートは、量子ビットを制御する入出力ハードウェアコンポーネントを表すソフトウェア抽象化です。これにより、ハードウェアベンダーは、ユーザーが量子ビットを操作して監視できるインターフェイスを提供できます。ポートは、コネクタの名前を表す単一の文字列によって決定されます。また、この文字列には、波形をどれだけ細かく定義できるかを指定する最小時間増分も定義できます。

```
from braket.pulse import Port

Port0 = Port("channel_0", dt=1e-9)
```

波形

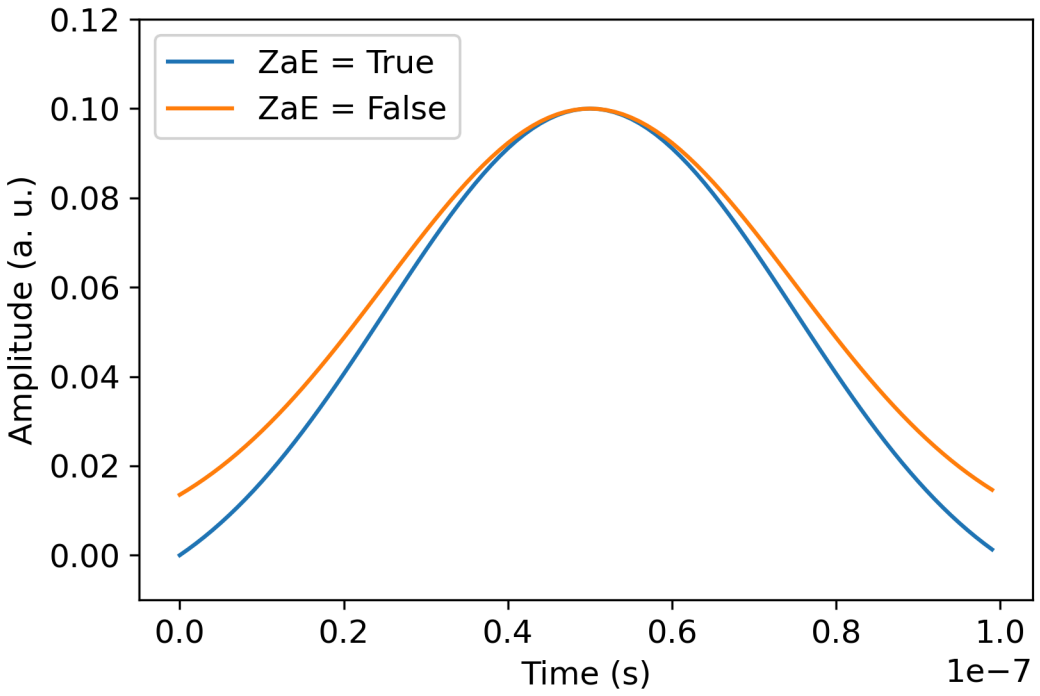
波形は、出力ポートで信号を出力したり、入力ポートを介して信号をキャプチャしたりするために使用できる時間依存エンベロープです。波形を直接指定するには、複素数のリストを使用するか、ハードウェアプロバイダーからリストを生成する波形テンプレートを使用します。

```
from braket.pulse import ArbitraryWaveform, ConstantWaveform
import numpy as np

cst_wfm = ConstantWaveform(length=1e-7, iq=0.1)
arb_wf = ArbitraryWaveform(amplitudes=np.linspace(0, 100))
```

Braket Pulse は、一定の波形、ガウシアン波形、断熱ゲートによる微分除去 (DRAG) 波形を含む、波形の標準ライブラリを提供します。次の例に示すように、sample 関数を使用して波形データを取得し、波形の形状を描画できます。

```
from braket.pulse import GaussianWaveform
import numpy as np
import matplotlib.pyplot as plt

zero_at_edge1 = GaussianWaveform(1e-7, 25e-9, 0.1, True)
```

```
# or zero_at_edge1 = GaussianWaveform(1e-7, 25e-9, 0.1)
zero_at_edge2 = GaussianWaveform(1e-7, 25e-9, 0.1, False)

times_1 = np.arange(0, zero_at_edge1.length, drive_frame.port.dt)
times_2 = np.arange(0, zero_at_edge2.length, drive_frame.port.dt)

plt.plot(times_1, zero_at_edge1.sample(drive_frame.port.dt))
plt.plot(times_2, zero_at_edge2.sample(drive_frame.port.dt))
```



上の画像は、GaussianWaveform から作成されたガウス波形を示しています。パルス長 100 ns、幅 25 ns、振幅 0.1 (任意単位) を選択しました。波形はパルスウィンドウの中央に配置されます。GaussianWaveform はブール引数 zero_at_edges (凡例の ZaE) を入力として取ります。この引数を True に設定すると、t=0 と t=length のポイントがゼロになるようにガウス波形がオフセットされ、最大値が amplitude 引数の値になるように振幅が再スケールされます。

Hello Pulse の使用方法

このセクションでは、Rigetti デバイスでパルスを使用して 1 つの量子ビットゲートを特性評価して直接構築する方法について説明します。量子ビットに電磁場を適用すると、Rabi 振動が発生し、量子ビットが 0 状態と 1 状態の間で切り替わります。パルスの長さと位相がキャリブレーションされている場合、Rabi 振動により 1 量子ビットのゲートを計算できます。ここでは、より複雑なパルス

シーケンスの構築に使用される基本ブロックである pi/2 パルスを測定するために、最適なパルス長を特定します。

まず、パルスシーケンスを構築するために、PulseSequence クラスをインポートします。

```
from braket.aws import AwsDevice
from braket.circuits import FreeParameter
from braket.devices import Devices
from braket.pulse import PulseSequence, GaussianWaveform

import numpy as np
```

次に、QPU の <noloc>Amazon リソースネーム</noloc> (ARN) を使用して新しい Braket デバイスをインスタンス化します。次のコマンドブロックでは、Rigetti Ankaa-3 が使用されています。

```
device = AwsDevice(Devices.Rigetti.Ankaa3)
```

次のパルスシーケンスには、波形の再生と量子ビットの測定という 2 つの構成要素が含まれています。通常、パルスシーケンスはフレームに適用できます。障害や遅延など、一部の例外は、量子ビットに適用できます。パルスシーケンスを構築する前に、使用可能なフレームを取得する必要があります。Rabi 振動のパルスを適用するために駆動フレームが使用され、量子ビット状態を測定するのには読み出しフレームが使用されます。次の例は、量子ビット 25 に対し、それらのフレームを使用しています。

```
drive_frame = device.frames["Transmon_25_charge_tx"]
readout_frame = device.frames["Transmon_25_readout_rx"]
```

次に、駆動フレームで再生される波形を作成します。目的は、さまざまなパルス長の量子ビットの動作を特性評価することです。毎回異なる長さの波形を再生することになります。毎回新しい波形をインスタンス化する代わりに、Braket でサポートされている FreeParameter をパルスシーケンスで使用します。波形とパルスシーケンスを自由パラメータで一度作成しておけば、このパルスシーケンスをさまざまな入力値を指定して実行できるのです。

```
waveform = GaussianWaveform(FreeParameter("length"), FreeParameter("length") * 0.25,
 0.2, False)
```

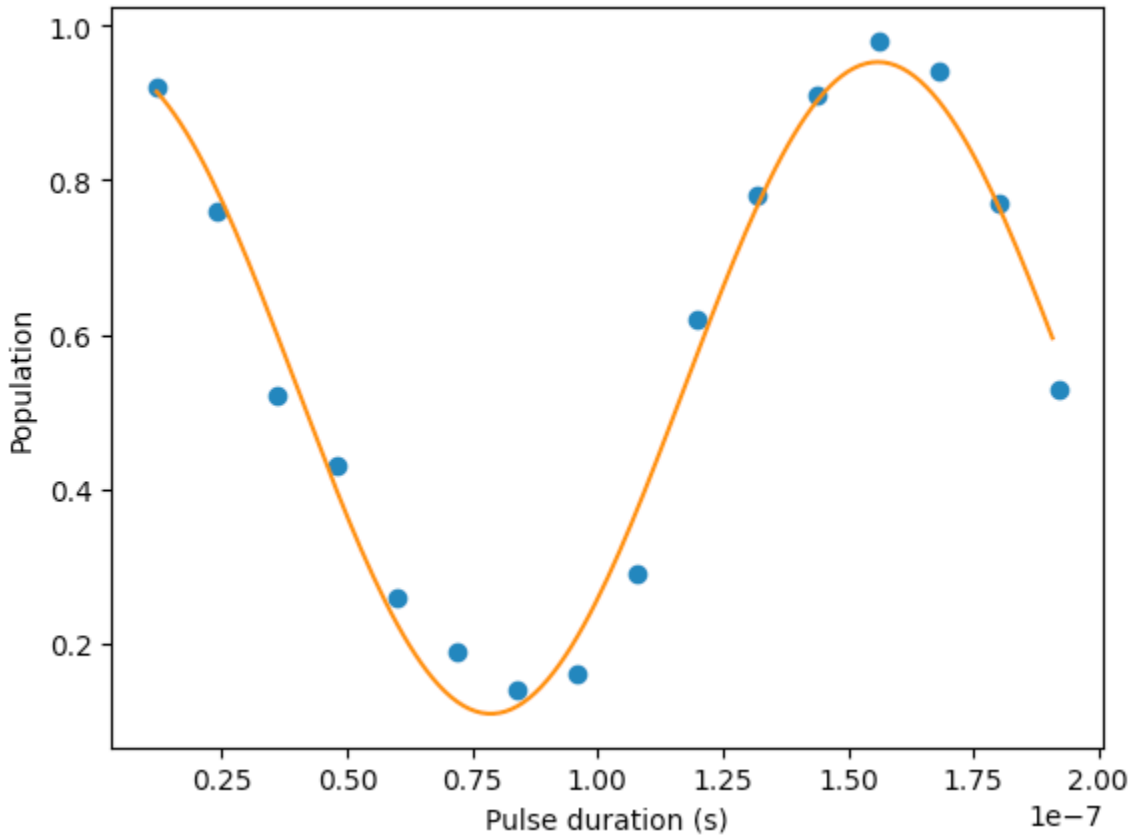
最後に、これらを1つのパルスシーケンスとしてまとめます。パルスシーケンスにおいて、play は駆動フレームで指定された波形を再生し、capture_v0 は読み出しフレームの状態を測定します。

```
pulse_sequence = (
    PulseSequence()
    .play(drive_frame, waveform)
    .capture_v0(readout_frame)
)
```

パルス長の範囲をスキャンし、QPU に送信します。QPU でパルスシーケンスを実行する前に、自由パラメータの値をバインドします。

```
start_length = 12e-9
end_length = 2e-7
lengths = np.arange(start_length, end_length, 12e-9)
N_shots = 100

tasks = [
    device.run(pulse_sequence(length=length), shots=N_shots)
    for length in lengths
]

probability_of_zero = [
    task.result().measurement_counts['0']/N_shots
    for task in tasks
]
```

量子ビット測定の統計は、0 状態と 1 状態の間で振動する量子ビットの振動力学を示します。測定データから Rabi 周波数を抽出し、パルスの長さをファインチューニングして、特別な 1 量子ビットゲートを実装できます。以下の図のデータから、例えば、周期性は約 154 ns です。したがって、pi/2 回転ゲートは、長さが 38.5 ns のパルスシーケンスに対応します。



OpenPulse によるパルス入門

OpenPulse は、一般的な量子デバイスのパルスレベルの制御を指定するための言語であり、OpenQASM 3.0 仕様の一部です。OpenPulse は OpenQASM 3.0 の表現を用いてパルスを直接プログラミングするための機能であり、Amazon Braket によってサポートされています。

Braket は、ネイティブ命令でパルスを表現するための基盤となる中間表現として OpenPulse を使用します。OpenPulse は、`defcal` (「define calibration」の略) 宣言の形式で命令のキャリブレーション結果を追加できます。この宣言により、低レベルの制御文法を使用してゲート命令の実装を指定できます。

OpenPulse の Braket `PulseSequence` プログラムを表示するには、次のコマンドを使用します。

```
print(pulse_sequence.to_ir())
```

また、OpenPulse プログラムを直接構築することもできます。

```
from braket.ir.openqasm import Program

openpulse_script = """
```

```
OPENQASM 3.0;
cal {
    bit[1] psb;
    waveform my_waveform = gaussian(12.0ns, 3.0ns, 0.2, false);
    play(Transmon_25_charge_tx, my_waveform);
    psb[0] = capture_v0(Transmon_25_readout_rx);
}
"""
```

スクリプトを使用して Program オブジェクトを作成します。次に、プログラムを QPU に送信します。

```
from braket.aws import AwsDevice
from braket.devices import Devices
from braket.ir.openqasm import Program

program = Program(source=openpulse_script)

device = AwsDevice(Devices.Rigetti.Ankaa3)
task = device.run(program, shots=100)
```

パルスを使用したネイティブゲートへのアクセス

多くの場合、研究者は、特定の QPU でサポートされているネイティブゲートをパルスとして実装する方法を正確に把握する必要があります。パルスシーケンスはハードウェアプロバイダーによって慎重にキャリブレーションされていますが、それらを利用することで、研究者はより良いゲートを設計したり、特定ゲートのパルスの伸張によるゼロノイズ外挿など、エラー緩和のための規定を探求したりすることができます。

Amazon Braket は、Rigetti のネイティブゲートへのプログラムによるアクセスをサポートしています。

```
import math
from braket.aws import AwsDevice
from braket.circuits import Circuit, GateCalibrations, QubitSet
from braket.circuits.gates import Rx

device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

calibrations = device.gate_calibrations
print(f"Downloaded {len(calibrations)} calibrations.")
```

> **Note**
>
> ハードウェアプロバイダーは QPU を定期的に (通常、1 日に 2 回以上) キャリブレーションしています。Braket SDK を使用すると、最新のゲートキャリブレーション結果を取得できます。

```
device.refresh_gate_calibrations()
```

RX ゲートや XY ゲートなど、特定のネイティブゲートを取得するには、Gate オブジェクトと目的の量子ビットを渡す必要があります。例えば、qubit 0 に適用された RX(π/2) のパルス実装を検査してみましょう。

```
rx_pi_2_q0 = (Rx(math.pi/2), QubitSet(0))

pulse_sequence_rx_pi_2_q0 = calibrations.pulse_sequences[rx_pi_2_q0]
```

フィルタリングされたキャリブレーションのセットを作成するには、filter 関数を使用します。ゲートのリストまたは QubitSet のリストを渡します。次のコードは、RX(π/2) のすべてのキャリブレーション結果を含むセットと、qubit0 のすべてのキャリブレーション結果を含むセットを作成します。

```
rx_calibrations = calibrations.filter(gates=[Rx(math.pi/2)])
q0_calibrations = calibrations.filter(qubits=QubitSet([0]))
```

カスタムキャリブレーションセットをアタッチすることで、ネイティブゲートのアクションを提供または変更できるようになりました。例えば、次の回路について考えてみましょう。

```
bell_circuit = (
    Circuit()
    .rx(0, math.pi/2)
    .rx(1, math.pi/2)
    .iswap(0, 1)
    .rx(1, -math.pi/2)
)
```

qubit 0 で rx ゲートのカスタムゲートキャリブレーション結果を使用してこの回路を実行するには、PulseSequence オブジェクトのディクショナリを gate_definitions キーワード引数に渡します。ディクショナリを作成するには、GateCalibrations オブジェクトの

pulse_sequences 属性を使用します。指定しなかったすべてのゲートは、量子ハードウェアプロバイダーのパルスキャリブレーション結果に置き換えられます。

```
nb_shots = 50
custom_calibration = GateCalibrations({rx_pi_2_q0: pulse_sequence_rx_pi_2_q0})
task = device.run(bell_circuit, gate_definitions=custom_calibration.pulse_sequences,
 shots=nb_shots)
```

アナログハミルトニアンシミュレーション

アナログハミルトニアンシミュレーション (AHS) は、量子コンピューティングにおける新たなパラダイムであり、従来の量子回路モデルとは大きく異なります。各回路は一度に、一連のゲートでなく、数量子ビットにのみ作用します。AHS プログラムは、問題のハミルトニアンの時間依存パラメータとスペース依存パラメータによって定義されます。系のハミルトニアンは、系のエネルギー準位と外力の作用をエンコードします。これらが連携して系の状態の時間進化を管理します。N 量子ビットの系の場合、ハミルトニアンは複素数で構成される $2^N X 2^N$ の正方マトリックスで表すことができます。

AHS を実行できる量子デバイスは、内部制御パラメータを慎重に調整することで、カスタムハミルトニアンにおける量子系の時間進化をほぼ近似できるように設計されています。調整できるパラメータの例は、コヒーレント駆動場の振幅やパラメータなどです。AHS パラダイムは、凝縮系物理学や量子化学など、相互作用パーティクルが多い量子系の静的および動的特性をシミュレートするのに適しています。AHS のパワーを利用し、従来のデジタル量子コンピューティングアプローチでは手の届かなかった問題に革新的な方法で対処するため、QuEra の Aquila デバイスなど、専用の量子処理装置 (QPU) が開発されました。

このセクションの内容:

- Hello AHS: 初めてのアナログハミルトニアンシミュレーションを実行する
- QuEra Aquila を使用してアナログプログラムを送信する

Hello AHS: 初めてのアナログハミルトニアンシミュレーションを実行する

このセクションでは、初めてのアナログハミルトニアンシミュレーションの実行について説明します。

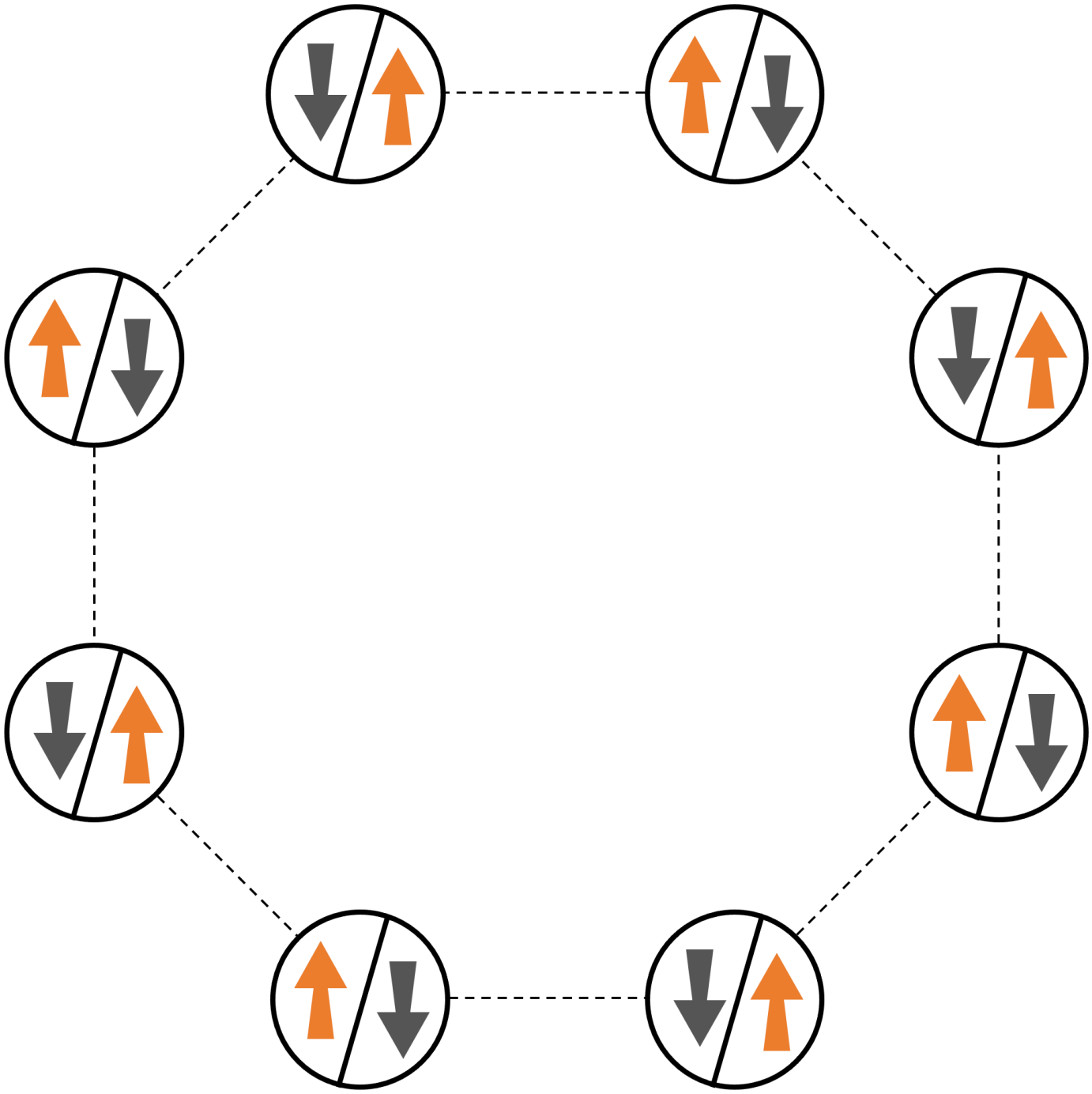
このセクションの内容:

- スピン鎖の操作

- 配置
- 相互作用
- 駆動場
- AHS プログラム
- ローカルシミュレーターでの実行
- シミュレーター結果の分析
- QuEra Aquila QPU での実行
- QPU の結果を分析する
- 次の手順

スピン鎖の操作

多くの相互作用パーティクルで構成される系の正規例については、8 つのスピンのリング (それぞれが「上向き」|↑# 状態と「下向き」|↓# 状態になる場合があります) を考えてみましょう。次のモデル系は小さいながらも、自然に発生する磁性物質によるいくつかの興味深い現象を最初から示しています。この例では、連続するスピンが反対方向に向く、いわゆる反強磁性秩序を作成する方法を示します。



配置

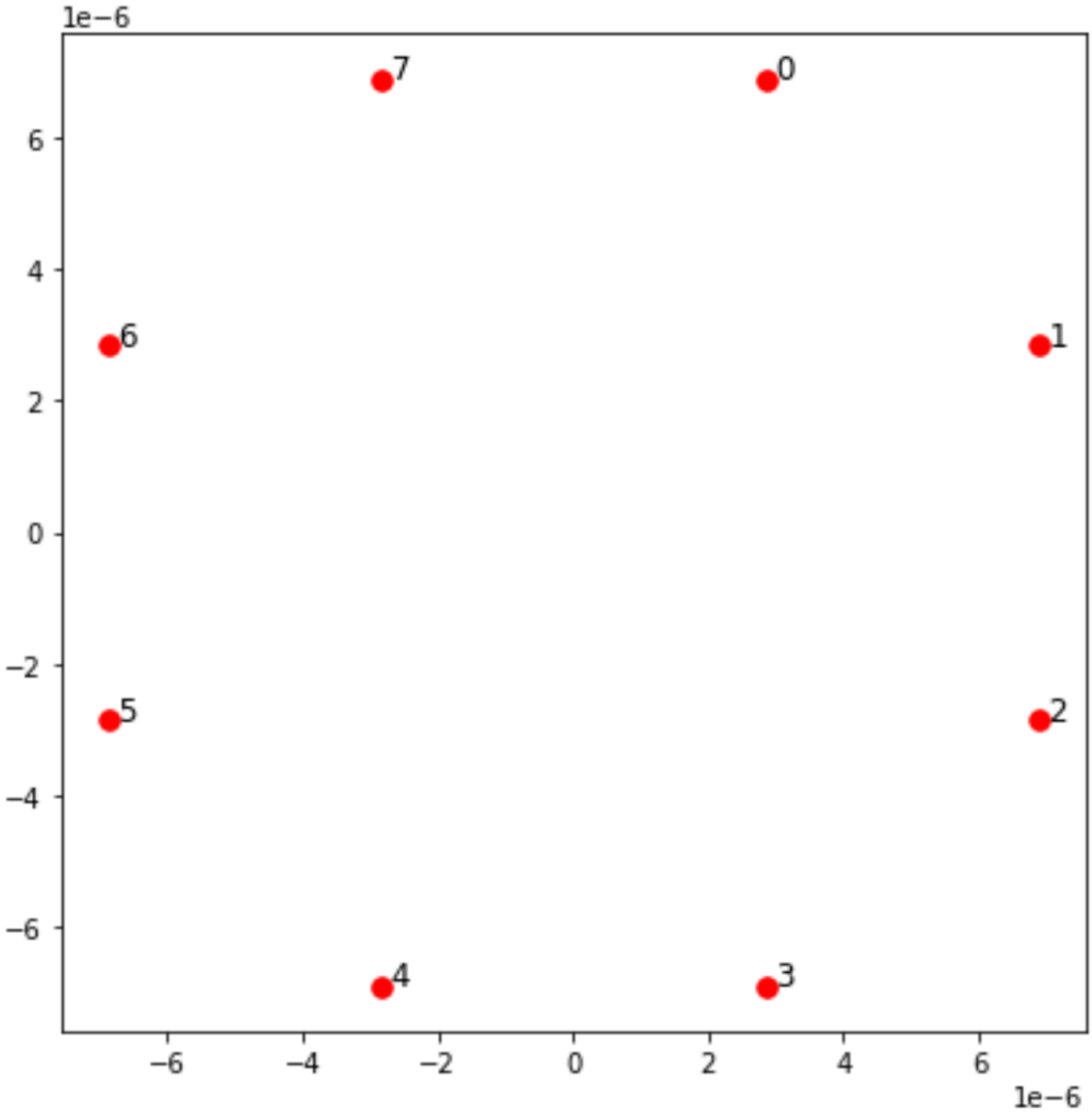
スピンごとに 1 つの中性原子を使用し、「上向き」スピン状態と「下向き」スピン状態をそれぞれ、原子の励起リュードベリ状態と基底状態でエンコードします。まず、2-d 配置を作成します。上記のスピンリングは、下記のコードでプログラミングできます。

前提条件: Braket SDK を pip install コマンドでインストールする必要があります。(Braket がホストするノートブックインスタンスを使用する場合には、そのノートブックはこの SDK にプリインストールされています)。プロットを再現する場合は、シェルコマンド `pip install matplotlib` を使用して matplotlib を別途インストールすることも必要です。

```
from braket.ahs.atom_arrangement import AtomArrangement
import numpy as np
import matplotlib.pyplot as plt  # Required for plotting

a = 5.7e-6  # Nearest-neighbor separation (in meters)

register = AtomArrangement()
register.add(np.array([0.5, 0.5 + 1/np.sqrt(2)]) * a)
register.add(np.array([0.5 + 1/np.sqrt(2), 0.5]) * a)
register.add(np.array([0.5 + 1/np.sqrt(2), - 0.5]) * a)
register.add(np.array([0.5, - 0.5 - 1/np.sqrt(2)]) * a)
register.add(np.array([-0.5, - 0.5 - 1/np.sqrt(2)]) * a)
register.add(np.array([-0.5 - 1/np.sqrt(2), - 0.5]) * a)
register.add(np.array([-0.5 - 1/np.sqrt(2), 0.5]) * a)
register.add(np.array([-0.5, 0.5 + 1/np.sqrt(2)]) * a)
```

これをプロットするには、以下を使用します。

```
fig, ax = plt.subplots(1, 1, figsize=(7, 7))
xs, ys = [register.coordinate_list(dim) for dim in (0, 1)]
ax.plot(xs, ys, 'r.', ms=15)

for idx, (x, y) in enumerate(zip(xs, ys)):
    ax.text(x, y, f" {idx}", fontsize=12)

plt.show()  # This will show the plot below in an ipython or jupyter session
```



相互作用

反強磁性相を作成するには、隣接するスピン間の相互作用を誘発する必要があります。これには[ファンデルワールス相互作用](#)を使用します。これは、中性原子デバイス (QuEra の Aquila デバイスなど) でネイティブに実装されています。この相互作用のハミルトニアン項は、スピン表現を使用することで、すべてのスピンペア (j,k) の和として表現できます。

$$H_{\text{interaction}} = \sum_{j=1}^{N-1} \sum_{k=j+1}^{N} V_{j,k} n_j n_k$$

ここで、「nj=|↑$_j$##↑$_j$| 」は、スピン j が「上向き」状態である場合にのみ値 1 を取り、それ以外の場合は 0 を取る演算子です。強度は $V_{j,k}=C_6/(d_{j,k})^6$ です。C_6は固定係数、$d_{j,k}$ はスピン j と k の間のユークリッド距離です。この相互作用項には、「スピン j とスピン k の両方が『上向き』であるすべての状態が、エネルギーを ($V_{j,k}$ の量だけ) 増加させる」という即時効果があります。この相互作用に関する AHS プログラムの残りの部分を綿密に設計することで、隣接するスピンが両方とも「上向き」状態になるのを防ぐことができます。これは、一般的に「リュードベリ遮断」と呼ばれる効果です。

駆動場

AHS プログラムの開始時、すべてのスピン (デフォルト) は「下向き」状態で始まり、いわゆる強磁性相にあります。強磁性相を作成するという目的に留意し、スピンをこの状態から、「上向き」状態が望ましい多体状態にスムーズに遷移させる、時間依存のコヒーレント駆動場を指定します。対応するハミルトニアンは次のように記述できます。

$$H_{\text{drive}}(t) = \sum_{k=1}^{N} \frac{1}{2}\Omega(t)\left[e^{i\phi(t)} S_{-,k} + e^{-i\phi(t)} S_{+,k}\right] - \sum_{k=1}^{N} \Delta(t) n_k$$

ここで、Ω(t)、ϕ(t)、Δ(t) はそれぞれ、すべてのスピンに均一に作用する駆動場に対する、時間依存、グローバル振幅 (別名: Rabi 周波数)、位相、およびデチューニングです。ここで、$S_{-,k}$=|↓$_k$##↑$_k$| および $S_{+,k}=(S_{-,k})^{\dagger}$=|↑$_k$##↓$_k$| はそれぞれ、スピン k の下降演算子と上昇演算子であり、n_k=|↑$_k$##↑$_k$| は以前と同じ演算子です。駆動場において、Ω の項は、すべてのスピンの「下向き」状態と「上向き」状態を同時にコヒーレントに結合し、Δ の項は「上向き」状態のエネルギー報酬を制御します。

強磁性相から反強磁性相へのスムーズな遷移をプログラミングするには、駆動場を次のコードで指定します。

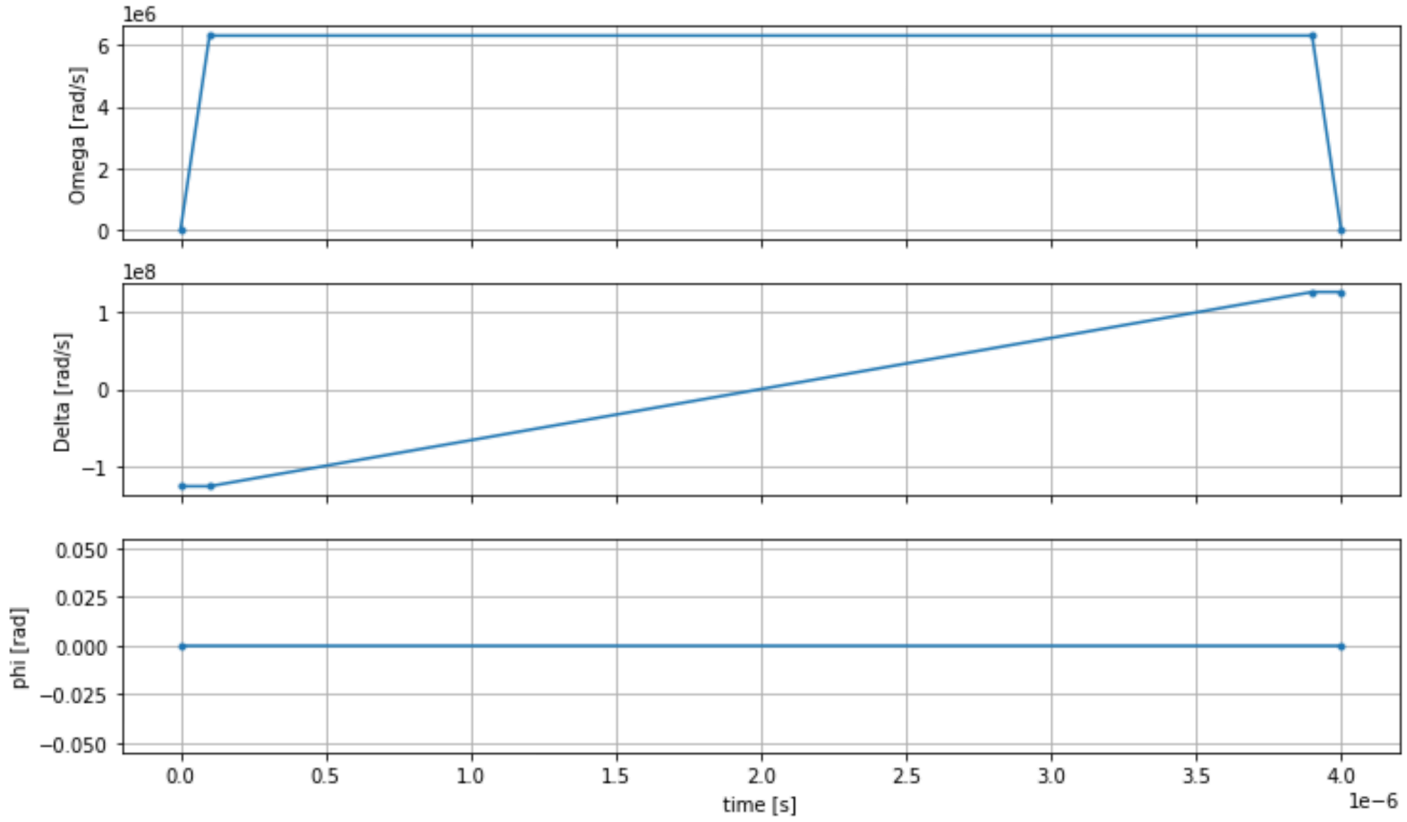
```
from braket.timings.time_series import TimeSeries
from braket.ahs.driving_field import DrivingField

# Smooth transition from "down" to "up" state
time_max = 4e-6  # seconds
time_ramp = 1e-7  # seconds
omega_max = 6300000.0  # rad / sec
delta_start = -5 * omega_max
```

```
delta_end = 5 * omega_max

omega = TimeSeries()
omega.put(0.0, 0.0)
omega.put(time_ramp, omega_max)
omega.put(time_max - time_ramp, omega_max)
omega.put(time_max, 0.0)

delta = TimeSeries()
delta.put(0.0, delta_start)
delta.put(time_ramp, delta_start)
delta.put(time_max - time_ramp, delta_end)
delta.put(time_max, delta_end)

phi = TimeSeries().put(0.0, 0.0).put(time_max, 0.0)

drive = DrivingField(
   amplitude=omega,
   phase=phi,
   detuning=delta
)
```

駆動場を時系列で図示するには、次のスクリプトを使用します。

```
fig, axes = plt.subplots(3, 1, figsize=(12, 7), sharex=True)

ax = axes[0]
time_series = drive.amplitude.time_series
ax.plot(time_series.times(), time_series.values(), '.-')
ax.grid()
ax.set_ylabel('Omega [rad/s]')

ax = axes[1]
time_series = drive.detuning.time_series
ax.plot(time_series.times(), time_series.values(), '.-')
ax.grid()
ax.set_ylabel('Delta [rad/s]')

ax = axes[2]
time_series = drive.phase.time_series
# Note: time series of phase is understood as a piecewise constant function
ax.step(time_series.times(), time_series.values(), '.-', where='post')
ax.set_ylabel('phi [rad]')
```

```
ax.grid()
ax.set_xlabel('time [s]')

plt.show()  # This will show the plot below in an ipython or jupyter session
```



AHS プログラム

アナログハミルトニアンシミュレーションプログラム ahs_program は、レジスタと駆動場 (および暗黙のファンデルワールス相互作用) で構成されています。

```
from braket.ahs.analog_hamiltonian_simulation import AnalogHamiltonianSimulation

ahs_program = AnalogHamiltonianSimulation(
   register=register,
   hamiltonian=drive
)
```

ローカルシミュレーターでの実行

この例は小規模 (15 スピン未満) のため、AHS 互換 QPU で実行する前に、Braket SDK に付属のローカル AHS シミュレーターで実行できます。このローカルシミュレーターは、Braket SDK で無償で利用できるため、コードが正しく実行されるかを確認するためのベストプラクティスです。

ここでは、ショットの数を高い値 (例えば、100 万) に設定できます。量子状態の時間進化を追跡し、最終状態からサンプルを描画するのが、ローカルシミュレーターであるためです。つまり、ショットの数を増やしても、全体のランタイムはわずかに長くなるだけで済みます。

```
from braket.devices import LocalSimulator

device = LocalSimulator("braket_ahs")

result_simulator = device.run(
   ahs_program,
   shots=1_000_000
).result()  # Takes about 5 seconds
```

シミュレーター結果の分析

各スピンの状態 (「下向き」の場合は「d」、「上向き」の場合は「u」、空のサイトの場合は「e」) を推測する次の関数を使用して、ショット結果を集計し、各設定がショット全体で発生した回数をカウントできます。

```
from collections import Counter


def get_counts(result):
    """Aggregate state counts from AHS shot results

    A count of strings (of length = # of spins) are returned, where
    each character denotes the state of a spin (site):
      e: empty site
      u: up state spin
      d: down state spin

    Args:
      result
 (braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationQua
```

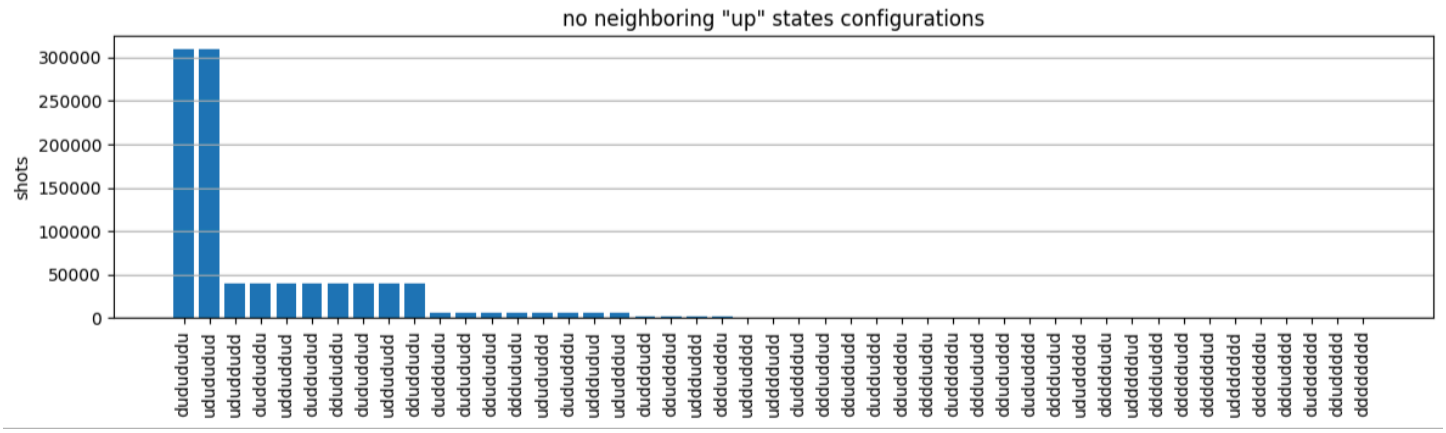
```
    Returns
       dict: number of times each state configuration is measured

    """
    state_counts = Counter()
    states = ['e', 'u', 'd']
    for shot in result.measurements:
        pre = shot.pre_sequence
        post = shot.post_sequence
        state_idx = np.array(pre) * (1 + np.array(post))
        state = "".join(map(lambda s_idx: states[s_idx], state_idx))
        state_counts.update((state,))
    return dict(state_counts)


counts_simulator = get_counts(result_simulator)  # Takes about 5 seconds
print(counts_simulator)
```

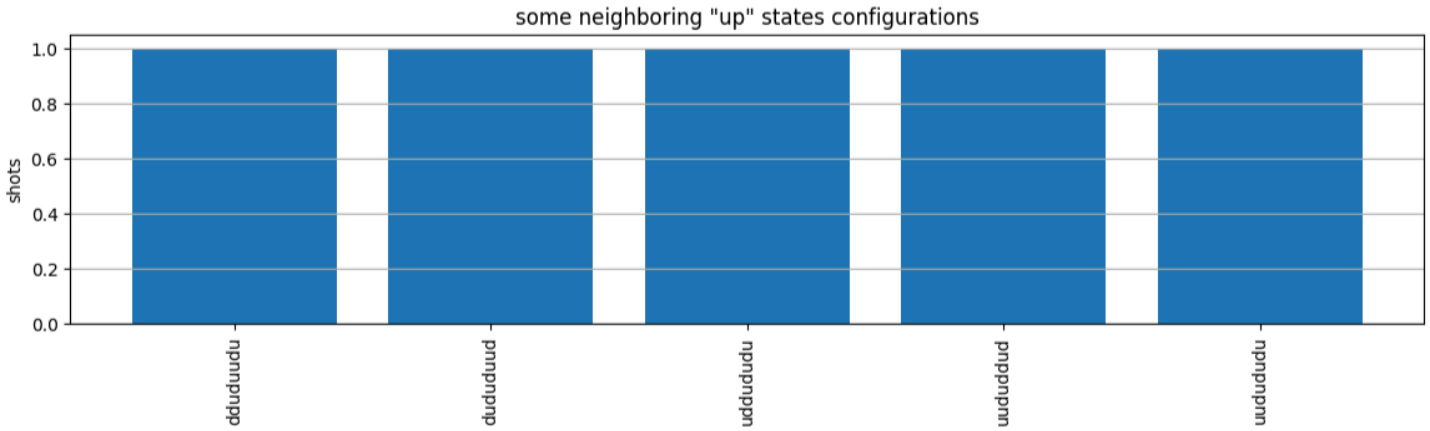
```
*[Output]*
{'dddddddd': 5, 'ddddddddu': 12, 'ddddddud': 15, ...}
```

ここで、counts は、ショット全体で各状態設定が観測された回数をカウントするディクショナリです。また、この回数を視覚化するには、次のコードを使用します。

```
from collections import Counter


def has_neighboring_up_states(state):
    if 'uu' in state:
        return True
    if state[0] == 'u' and state[-1] == 'u':
        return True
    return False


def number_of_up_states(state):
    return Counter(state)['u']


def plot_counts(counts):
    non_blockaded = []
    blockaded = []
    for state, count in counts.items():
```

```
        if not has_neighboring_up_states(state):
            collection = non_blockaded
        else:
            collection = blockaded
        collection.append((state, count, number_of_up_states(state)))

    blockaded.sort(key=lambda _: _[1], reverse=True)
    non_blockaded.sort(key=lambda _: _[1], reverse=True)

    for configurations, name in zip((non_blockaded,
                                     blockaded),
                                    ('no neighboring "up" states',
                                     'some neighboring "up" states')):
        plt.figure(figsize=(14, 3))
        plt.bar(range(len(configurations)), [item[1] for item in configurations])
        plt.xticks(range(len(configurations)))
        plt.gca().set_xticklabels([item[0] for item in configurations], rotation=90)
        plt.ylabel('shots')
        plt.grid(axis='y')
        plt.title(f'{name} configurations')
        plt.show()


plot_counts(counts_simulator)
```





プロットから、以下の観測値を読み取ることで、反強磁性相が作成されたことを確認できます。

1. 通常、非遮断状態 (いかなる 2 つの隣接するスピンも「上向き」状態ではない状態) は、少なくとも 1 つの隣接スピンペアが両方とも「上向き」状態である状態よりも一般的です。
2. 一般に、設定が遮断状態でない限り、より多くの「上向き」の励起がある状態が優先されます。
3. 実際には、最も一般的な状態は、完全な反強磁性状態 "dudududu" と "udududud" です。
4. 2 番目に一般的な状態は、連続する間隔が 1、2、2 である 3 つの「上向き」励起のみを持つ状態です。これは、ファンデルワールス相互作用が、次の最近傍にも作用する (ただし、はるかに小さなものです) ことを示しています。

QuEra Aquila QPU での実行

前提条件: Amazon Braket を初めて使用する場合 (Braket SDK を pip install コマンドでインストールすることは除く) は、必要な開始手順を完了していることを確認してください。

> **Note**
>
> Braket がホストするノートブックインスタンスを使用する場合には、そのインスタンスが Braket SDK にプリインストールされています。

すべての依存関係をインストールすると、Aquila QPU に接続できます。

```
from braket.aws import AwsDevice

aquila_qpu = AwsDevice("arn:aws:braket:us-east-1::device/qpu/quera/Aquila")
```

AHS プログラムを QuEra マシンに適したものにするには、Aquila QPU で許容される精度レベルに準拠するようにすべての値を丸める必要があります。(丸める必要があるかどうかは、名前に「Resolution」が含まれるデバイスパラメータで管理されています。そのようなデバイスパラメータを確認するには、ノートブックで `aquila_qpu.properties.dict()` を実行します。Aquila の機能と要件の詳細については、「Introduction to Aquila」ノートブックを参照してください)。丸めるには、`discretize` メソッドを呼び出します。

```
discretized_ahs_program = ahs_program.discretize(aquila_qpu)
```

これで、プログラムを Aquila QPU で実行 (現在、100 ショットのみを実行) できるようになりました。

> **Note**
>
> このプログラムを Aquila プロセッサで実行すると、コストが発生します。Amazon Braket SDK に搭載されているコストトラッカーにより、お客様はコスト制限を設定し、ほぼリアルタイムでコストを追跡できます。

```
task = aquila_qpu.run(discretized_ahs_program, shots=100)

metadata = task.metadata()
task_arn = metadata['quantumTaskArn']
task_status = metadata['status']

print(f"ARN: {task_arn}")
print(f"status: {task_status}")
```

```
*[Output]*
ARN: arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef
status: CREATED
```

量子タスクの実行にかかる時間は可用性ウィンドウと QPU 使用率によって大きなばらつきがあるため、後で次のコードスニペットを使用して量子タスク ARN の状態を確認できるように、ARN を書き留めておくことをお勧めします。

```
# Optionally, in a new python session
```

```
from braket.aws import AwsQuantumTask

SAVED_TASK_ARN = "arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef"

task = AwsQuantumTask(arn=SAVED_TASK_ARN)
metadata = task.metadata()
task_arn = metadata['quantumTaskArn']
task_status = metadata['status']

print(f"ARN: {task_arn}")
print(f"status: {task_status}")
```

```
*[Output]*
ARN: arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef
status: COMPLETED
```

状態が [完了] になる (Amazon Braket コンソールの量子タスクページから確認できます) と、以下を使用してタスクの結果をクエリできます。

```
result_aquila = task.result()
```

QPU の結果を分析する

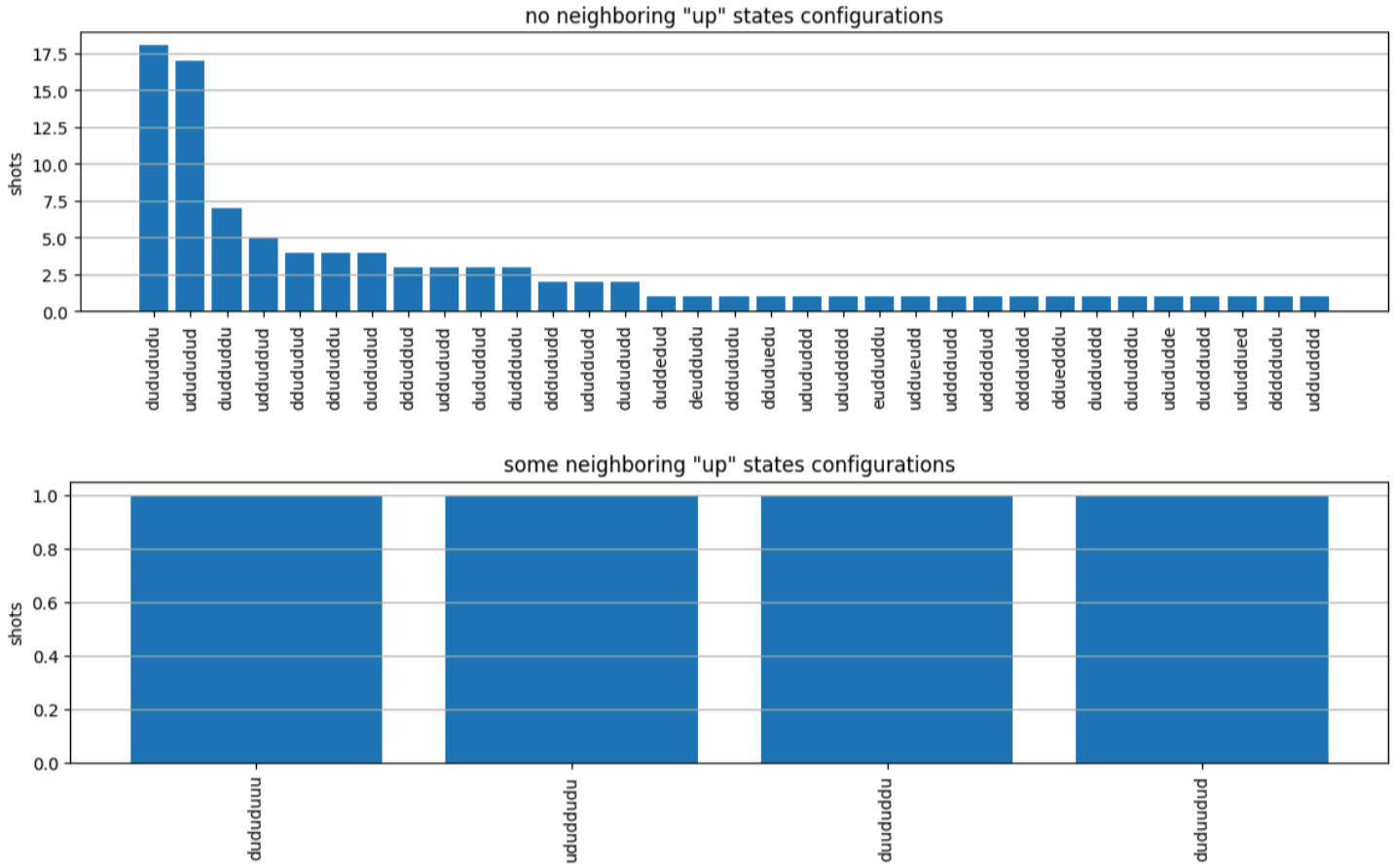
カウントを計算するには、以前と同じ get_counts 関数を使用します。

```
counts_aquila = get_counts(result_aquila)
   print(counts_aquila)
```

```
*[Output]*
{'ddddudud': 2, 'dudududu': 18, 'ddududud': 4, ...}
```

次に、カウントをプロットするには、plot_counts を使用します。

```
plot_counts(counts_aquila)
```



ショットのごく一部に空のサイトがあることに注意してください (「e」でマークされています)。これは、Aquila QPU の原子ごとの準備の不完全性が 1～2% であるためです。これを例外とすれば、結果は、シミュレーションと、ショット数が少ないことによる予想される統計的変動の範囲内で一致しています。

次の手順

これで、ローカル AHS シミュレーターと Aquila QPU を使用して Amazon Braket で初めての AHS ワークロードを実行できました。

リュードベリ物理学、アナログハミルトニアンシミュレーション、Aquila デバイスの詳細については、example ノートブックを参照してください。

QuEra Aquila を使用してアナログプログラムを送信する

このページでは、QuEra Aquila マシンの機能に関する包括的なドキュメントを提供します。ここで説明する細目は次のとおりです。

1. Aquila によってシミュレートされる、パラメータ化されたハミルトニアン

2. AHS プログラムのパラメータ
3. AHS の結果の内容
4. Aquila の機能のパラメータ

このセクションの内容:

- ハミルトニアン
- Braket AHS プログラムのスキーマ
- Braket AHS タスク結果スキーマ
- QuEra デバイスプロパティのスキーマ

ハミルトニアン

QuEra Aquila マシンは、次の (時間依存) ハミルトニアンをネイティブでシミュレートします。

$$H(t) = \sum_{k=1}^{N} H_{\text{drive},k}(t) + \sum_{k=1}^{N} H_{\text{local detuning},k}(t) + \sum_{k=1}^{N-1} \sum_{l=k+1}^{N} V_{\text{vdw},k,l}$$

> **Note**
>
> ローカルデチューニングへのアクセスは実験機能であり、Braket Direct を通じたリクエストによって利用可能になります。

各一般項の意味は次のとおりです。

- $H_{\text{drive},k}(t) = (\frac{1}{2}\,\Omega(t)e^{i\phi(t)}S_{-,k} + \frac{1}{2}\,\Omega(t)e^{-i\phi(t)}\,S_{+,k}) + (-\Delta_{\text{global}}(t)n_k)$,
  - Ω(t) は、時間依存のグローバル駆動振幅 (別名: Rabi 周波数) を rad/s 数として表したものです
  - ϕ(t) は時間依存のグローバル位相であり、ラジアン数として測定されます。
  - $S_{-,k}$ と $S_{+,k}$ は、原子 k のスピン下降演算子および上昇演算子です (基底 |↓#=|g#、|↑#=|r# において、これらは S_{-}=|g##r|、$S_{+}=(S_{-})^{\dagger}$=|r##g|) となります)
  - $\Delta_{\text{global}}(t)$ は時間依存のグローバルデチューニングです
  - n_k は、原子 k の リュードベリ状態への射影演算子です (つまり、n=|r##r|)。
- $H_{\text{local detuning},k}(t) = -\Delta_{\text{local}}(t)h_k n_k$
  - $\Delta_{\text{local}}(t)$ は、ローカル周波数シフトの時間依存係数であり、単位は rad/s です。

  - h_k はサイト依存因子であり、0.0 から 1.0 の間の無次元数です。
- $V_{vdw,k,l}=C_6/(d_{k,l})^6 n_k n_l$
  - $_6$ はファンデルワールス係数であり、単位は「rad/s * m^6」です。
  - $d_{k,l}$ は、原子 k と l の間のユークリッド距離であり、メートル数として測定されます。

ユーザーは Braket AHS プログラムスキーマを使用して以下のパラメータを制御できます。

- 2次元での原子配置 (各原子 k の x_k および y_k 座標、μm 数): k,l=1、2、...N までの原子間の距離 $d_{k,l}$ を制御する
- Ω(t): 時間依存、グローバル Rabi 周波数、単位 rad/s
- ϕ(t): 時間依存、グローバル位相、単位 rad
- Δ_{global}(time): 時間依存、グローバル調整、単位 rad/s
- Δ_{local}(t): ローカルデチューニングの大きさの時間依存 (グローバル) 因子、単位 rad/s
- h_k: ローカルデチューニングの大きさの (静的) サイト依存因子、0.0 ~ 1.0 の無次元数

> **Note**
>
> ユーザーは、関係するレベルを制御できず (つまり、S_-、S_+、n 演算子は固定されている)、リュードベリ-リュードベリ相互作用係数 (C_6) の強度も制御できません。

Braket AHS プログラムのスキーマ

braket.ir.ahs.program_v1.Program オブジェクト (サンプル)

> **Note**
>
> アカウントでローカルデチューニング機能が有効になっていない場合は、次のサンプルに含まれている `localDetuning=[]` を使用します。

```
Program(
    braketSchemaHeader=BraketSchemaHeader(
        name='braket.ir.ahs.program',
        version='1'
    ),
```

```
    setup=Setup(
        ahs_register=AtomArrangement(
            sites=[
                [Decimal('0'), Decimal('0')],
                [Decimal('0'), Decimal('4e-6')],
                [Decimal('4e-6'), Decimal('0')]
            ],
            filling=[1, 1, 1]
        )
    ),
    hamiltonian=Hamiltonian(
        drivingFields=[
            DrivingField(
                amplitude=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('0'), Decimal('15700000.0'),
Decimal('15700000.0'), Decimal('0')],
                        times=[Decimal('0'), Decimal('0.000001'), Decimal('0.000002'),
Decimal('0.000003')]
                    ),
                    pattern='uniform'
                ),
                phase=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('0'), Decimal('0')],
                        times=[Decimal('0'), Decimal('0.000003')]
                    ),
                    pattern='uniform'
                ),
                detuning=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('-54000000.0'), Decimal('54000000.0')],
                        times=[Decimal('0'), Decimal('0.000003')]
                    ),
                    pattern='uniform'
                )
            )
        ],
        localDetuning=[
            LocalDetuning(
                magnitude=PhysicalField(
                    times_series=TimeSeries(
                        values=[Decimal('0'), Decimal('25000000.0'),
Decimal('25000000.0'), Decimal('0')],
```

```
                        times=[Decimal('0'), Decimal('0.000001'), Decimal('0.000002'),
 Decimal('0.000003')]
                    ),
                    pattern=Pattern([Decimal('0.8'), Decimal('1.0'), Decimal('0.9')])
                )
            )
        ]
    )
)
```

JSON (サンプル)

> **Note**
>
> アカウントでローカルデチューニング機能が有効になっていない場合は、次のサンプルに含まれている `"localDetuning": []` を使用します。

```
{
    "braketSchemaHeader": {
        "name": "braket.ir.ahs.program",
        "version": "1"
    },
    "setup": {
        "ahs_register": {
            "sites": [
                [0E-7, 0E-7],
                [0E-7, 4E-6],
                [4E-6, 0E-7]
            ],
            "filling": [1, 1, 1]
        }
    },
    "hamiltonian": {
        "drivingFields": [
            {
                "amplitude": {
                    "time_series": {
                        "values": [0.0, 15700000.0, 15700000.0, 0.0],
                        "times": [0E-9, 0.000001000, 0.000002000, 0.000003000]
                    },
                    "pattern": "uniform"
```

```
                },
                "phase": {
                    "time_series": {
                        "values": [0E-7, 0E-7],
                        "times": [0E-9, 0.000003000]
                    },
                    "pattern": "uniform"
                },
                "detuning": {
                    "time_series": {
                        "values": [-54000000.0, 54000000.0],
                        "times": [0E-9, 0.000003000]
                    },
                    "pattern": "uniform"
                }
            }
        ],
        "localDetuning": [
            {
                "magnitude": {
                    "time_series": {
                        "values": [0.0, 25000000.0, 25000000.0, 0.0],
                        "times": [0E-9, 0.000001000, 0.000002000, 0.000003000]
                    },
                    "pattern": [0.8, 1.0, 0.9]
                }
            }
        ]
    }
}
```

主なフィールド

プログラムフィールド	型	説明
setup.ahs_register.sites	List[List[Decimal]]	光ピンセットが原子を捕捉する 2 次元座標のリスト
setup.ahs_register.filling	List[int]	捕捉サイトを占有する原子を 1 でマークし、空のサ

プログラムフィールド	型	説明
		イトを 0 でマークします
hamiltonian.drivingFields[].amplitude.time_series.times	List[Decimal]	駆動振幅のタイムポイント、Omega(t)
hamiltonian.drivingFields[].amplitude.time_series.values	List[Decimal]	駆動振幅の値、Omega(t)
hamiltonian.drivingFields[].amplitude.pattern	str	駆動振幅の空間パターン、Omega(t)。「uniform」を指定する必要があります
hamiltonian.drivingFields[].phase.time_series.times	List[Decimal]	駆動位相のタイムポイント、phi(t)
hamiltonian.drivingFields[].phase.time_series.values	List[Decimal]	駆動位相の値、phi(t)
hamiltonian.drivingFields[].phase.pattern	str	駆動位相の空間パターン、phi(t)。「uniform」を指定する必要があります
hamiltonian.drivingFields[].detuning.time_series.times	List[Decimal]	駆動デチューニングのタイムポイント、Delta_global(t)
hamiltonian.drivingFields[].detuning.time_series.values	List[Decimal]	駆動デチューニングの値、Delta_global(t)

プログラムフィールド	型	説明
hamiltonian.drivingFields[].detuning.pattern	str	駆動デチューニングの空間パターン、Delta_global(t)。「均一」である必要があります
hamiltonian.localDetuning[].magnitude.time_series.times	List[Decimal]	ローカルデチューニングの大きさの時間依存因子のタイムポイント、Delta_local(t)
hamiltonian.localDetuning[].magnitude.time_series.values	List[Decimal]	ローカルデチューニングの大きさの時間依存因子の値、Delta_local(t)
hamiltonian.localDetuning[].magnitude.pattern	List[Decimal]	ローカルデチューニングの大きさのサイト依存係数 h_k (値は setup.ahs_register.sites の sites に対応)

メタデータフィールド

プログラムフィールド	型	説明
braketSchemaHeader.name	str	スキーマの名前。「braket.ir.ahs.program」を指定する必要があります
braketSchemaHeader.version	str	スキーマのバージョン

Braket AHS タスク結果スキーマ

braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationQuantumTas (サンプル)

```
AnalogHamiltonianSimulationQuantumTaskResult(
    task_metadata=TaskMetadata(
        braketSchemaHeader=BraketSchemaHeader(
            name='braket.task_result.task_metadata',
            version='1'
        ),
        id='arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef',
        shots=2,
        deviceId='arn:aws:braket:us-east-1::device/qpu/quera/Aquila',
        deviceParameters=None,
        createdAt='2022-10-25T20:59:10.788Z',
        endedAt='2022-10-25T21:00:58.218Z',
        status='COMPLETED',
        failureReason=None
    ),
    measurements=[
        ShotResult(
            status=<AnalogHamiltonianSimulationShotStatus.SUCCESS: 'Success'>,

            pre_sequence=array([1, 1, 1, 1]),
            post_sequence=array([0, 1, 1, 1])
        ),

        ShotResult(
            status=<AnalogHamiltonianSimulationShotStatus.SUCCESS: 'Success'>,

            pre_sequence=array([1, 1, 0, 1]),
            post_sequence=array([1, 0, 0, 0])
        )
    ]
)
```

JSON (サンプル)

```
{
    "braketSchemaHeader": {
        "name": "braket.task_result.analog_hamiltonian_simulation_task_result",
```

```
        "version": "1"
    },
    "taskMetadata": {
        "braketSchemaHeader": {
            "name": "braket.task_result.task_metadata",
            "version": "1"
        },
        "id": "arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef",
        "shots": 2,
        "deviceId": "arn:aws:braket:us-east-1::device/qpu/quera/Aquila",

        "createdAt": "2022-10-25T20:59:10.788Z",
        "endedAt": "2022-10-25T21:00:58.218Z",
        "status": "COMPLETED"

    },
    "measurements": [
        {
            "shotMetadata": {"shotStatus": "Success"},
            "shotResult": {
                "preSequence": [1, 1, 1, 1],
                "postSequence": [0, 1, 1, 1]
            }
        },
        {
            "shotMetadata": {"shotStatus": "Success"},
            "shotResult": {
                "preSequence": [1, 1, 0, 1],
                "postSequence": [1, 0, 0, 0]
            }
        }
    ],
    "additionalMetadata": {
        "action": {...}
        "queraMetadata": {
            "braketSchemaHeader": {
                "name": "braket.task_result.quera_metadata",
                "version": "1"
            },
            "numSuccessfulShots": 100
        }
    }
```

```
}
```

主なフィールド

タスク結果フィールド	型	説明
measurements[].shotResult.preSequence	List[int]	各ショットのシーケンス前測定ビット (原子サイトごとに 1 つ): サイトが空の場合は 0、サイトが占有されている場合は 1。測定は、量子進化を実行するパルスのシーケンスの前に行います
measurements[].shotResult.postSequence	List[int]	各ショットのシーケンス後測定ビット: 原子がリュードベリ状態か空の場合は 0、原子が基底状態の場合は 1。測定は、量子進化を実行するパルスシーケンスの終了後に行います

メタデータフィールド

タスク結果フィールド	型	説明
braketSchemaHeader.name	str	スキーマの名前。「braket.task_result.analog_hamiltonian_simulation_task_result」を指定する必要があります
braketSchemaHeader.version	str	スキーマのバージョン

タスク結果フィールド	型	説明
taskMetadata.braketSchemaHeader.name	str	スキーマの名前。「braket.task_result.task_metadata」を指定する必要があります
taskMetadata.braketSchemaHeader.version	str	スキーマのバージョン
taskMetadata.id	str	量子タスクの ID。 AWS 量子タスクの場合、これは量子タスク ARN です。
taskMetadata.shots	int	量子タスクのショット数
taskMetadata.shots.deviceId	str	量子タスクが実行されたデバイスの ID。 AWS デバイスの場合、これはデバイス ARN です。

タスク結果フィールド	型	説明
taskMetadata.shots.createdAt	str	作成のタイムスタンプ。形式は、ISO-8601/RFC 3339 文字列形式 YYYY-MM-DDTHH:mm:ss.sssZ にする必要があります。デフォルトは「None」です。
taskMetadata.shots.endedAt	str	量子タスクが終了した日時のタイムスタンプ。形式は ISO-8601/RFC3339 文字列形式 YYYY-MM-DDTHH:mm:ss.sssZ にする必要があります。デフォルトは「None」です。

タスク結果フィールド	型	説明
taskMetadata.shots.status	str	量子タスクのステータス (CREATED、QUEUED、RUNNING、COMPLETED、FAILED)。デフォルトは「None」です。
taskMetadata.shots.failureReason	str	量子タスクの失敗の理由。デフォルトは「None」です。
additionalMetadata.action	braket.ir.ahs.program_v1.Program	(「Braket AHS プログラムのスキーマ」セクションを参照)
additionalMetadata.action.braketSchemaHeader.queraMetadata.name	str	スキーマの名前。「braket.task_result.quera_metadata」を指定する必要があります
additionalMetadata.action.braketSchemaHeader.queraMetadata.version	str	スキーマのバージョン

タスク結果フィールド	型	説明
additionalMetadata.action.numSuccessfulShots	int	完全に成功したショットの数。リクエストされたショット数と同じになる必要があります
measurements[].shotMetadata.shotStatus	int	ショットの状態 (「成功」、「部分的成功」、「失敗」)、「成功] になる必要があります

QuEra デバイスプロパティのスキーマ

braket.device_schema.quera.quera_device_capabilities_v1.QueraDeviceCapabilities (サンプル)

```
QueraDeviceCapabilities(
    service=DeviceServiceProperties(
        braketSchemaHeader=BraketSchemaHeader(
            name='braket.device_schema.device_service_properties',
            version='1'
            ),
            executionWindows=[
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.MONDAY: 'Monday'>,
                    windowStartHour=datetime.time(1, 0),
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.TUESDAY: 'Tuesday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                ),
```

```
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.WEDNESDAY: 'Wednesday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.FRIDAY: 'Friday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.SATURDAY: 'Saturday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.SUNDAY: 'Sunday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                )
            ],
            shotsRange=(1, 1000),
            deviceCost=DeviceCost(
                price=0.01,
                unit='shot'
            ),
            deviceDocumentation=
                DeviceDocumentation(
                    imageUrl='https://
a.b.cdn.console.awsstatic.com/59534b58c709fc239521ef866db9ea3f1aba73ad3ebcf60c23914ad8c5c5c878/
a6cfc6fca26cf1c2e1c6.png',
                    summary='Analog quantum processor based on neutral atom arrays',
                    externalDocumentationUrl='https://www.quera.com/aquila'
                ),
                deviceLocation='Boston, USA',
                updatedAt=datetime.datetime(2024, 1, 22, 12, 0,
 tzinfo=datetime.timezone.utc),
                getTaskPollIntervalMillis=None
    ),
    action={
        <DeviceActionType.AHS: 'braket.ir.ahs.program'>: DeviceActionProperties(
                version=['1'],
                actionType=<DeviceActionType.AHS: 'braket.ir.ahs.program'>
            )
```

```
    },
    deviceParameters={},
    braketSchemaHeader=BraketSchemaHeader(
        name='braket.device_schema.quera.quera_device_capabilities',
        version='1'
    ),
    paradigm=QueraAhsParadigmProperties(
        ...
        # See https://github.com/amazon-braket/amazon-braket-schemas-python/blob/main/
src/braket/device_schema/quera/quera_ahs_paradigm_properties_v1.py
        ...
    )
)
```

JSON (サンプル)

```
{
    "service": {
        "braketSchemaHeader": {
            "name": "braket.device_schema.device_service_properties",
            "version": "1"
        },
        "executionWindows": [
            {
                "executionDay": "Monday",
                "windowStartHour": "01:00:00",
                "windowEndHour": "23:59:59"
            },
            {
                "executionDay": "Tuesday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "12:00:00"
            },
            {
                "executionDay": "Wednesday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "12:00:00"
            },
            {
                "executionDay": "Friday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "23:59:59"
            },
```

```
            {
                "executionDay": "Saturday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "23:59:59"
            },
            {
                "executionDay": "Sunday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "12:00:00"
            }
        ],
        "shotsRange": [
            1,
            1000
        ],
        "deviceCost": {
            "price": 0.01,
            "unit": "shot"
        },
        "deviceDocumentation": {
            "imageUrl": "https://
a.b.cdn.console.awsstatic.com/59534b58c709fc239521ef866db9ea3f1aba73ad3ebcf60c23914ad8c5c5c878/
a6cfc6fca26cf1c2e1c6.png",
            "summary": "Analog quantum processor based on neutral atom arrays",
            "externalDocumentationUrl": "https://www.quera.com/aquila"
        },
        "deviceLocation": "Boston, USA",
        "updatedAt": "2024-01-22T12:00:00+00:00"
    },
    "action": {
        "braket.ir.ahs.program": {
            "version": [
                "1"
            ],
            "actionType": "braket.ir.ahs.program"
        }
    },
    "deviceParameters": {},
    "braketSchemaHeader": {
        "name": "braket.device_schema.quera.quera_device_capabilities",
        "version": "1"
    },
    "paradigm": {
        ...
```

```
        # See Aquila device page > "Calibration" tab > "JSON" page
        ...
    }
}
```

サービスプロパティのフィールド

サービスプロパティのフィールド	型	説明
service.executionWindows[].executionDay	ExecutionDay	実行ウィンドウの曜日。「Everyday」、「Weekdays」、「Weekend」、「Monday」、「Tuesday」、「Wednesday」、「Thursday」、「Friday」、「Saturday」、または「Sunday」を指定する必要があります
service.executionWindows[].windowStartHour	datetime.time	実行ウィンドウを開始する時刻 (UTC 24 時間形式)
service.executionWindows[].windowEndHour	datetime.time	実行ウィンドウを終了する時刻 (UTC 24 時間形式)
service.qpu_capabilities.service.shotsRange	Tuple[int, int]	デバイスの最小ショット数と最大ショット数
service.qpu_capabilities.service.deviceCost.price	float	デバイスの価格 (単位: USD)
service.qpu_capabilities.service.deviceCost.unit	str	料金の請求単位。例：「minute」(分)、「hour」(時間)、「shot」(ショット)、「task」(タスク)

メタデータフィールド

メタデータフィールド	型	説明
action[].version	str	AHS プログラムのスキーマのバージョン
action[].actionType	ActionType	AHS プログラムスキーマ名。「braket.ir.ahs.program」を指定する必要があります
service.braketSchemaHeader.name	str	スキーマの名前。「braket.device_schema.device_service_properties」を指定する必要があります
service.braketSchemaHeader.version	str	スキーマのバージョン
service.deviceDocumentation.imageUrl	str	デバイスのイメージ (画像) の URL
service.deviceDocumentation.summary	str	デバイスに関する簡単な説明
service.deviceDocumentation.externalDocumentationUrl	str	外部ドキュメントの URL
service.deviceLocation	str	デバイスの地理的位置
service.updatedAt	datetime	デバイスプロパティが最後に更新された日時

AWS Boto3 の使用

Boto3 は AWS SDK for Python です。Boto3 を使用すると、Python 開発者は Amazon Braket などのを作成、設定 AWS のサービス、管理できます。Boto3 は、Amazon Braket への低レベルのアクセスだけでなく、オブジェクト指向の API を提供します。

「Boto3 クイックスタートガイド」の指示に従って、Boto3 をインストールして設定する方法を確認してください。

Boto3 は Amazon Braket Python SDK と連携して機能するコア機能を提供し、量子タスクの設定と実行に役立ちます。Python のお客様は常に Boto3 をインストールする必要があります。これがコア実装だからです。追加のヘルパーメソッドを使用する場合は、Amazon Braket SDK もインストールする必要があります。

例えば、`CreateQuantumTask` を呼び出すと、Amazon Braket SDK がリクエストを Boto3 に送信することで、Boto3 が AWS API を呼び出します。

このセクションの内容:

- Amazon Braket Boto3 クライアントを有効にする
- Boto3 と Braket SDK の AWS CLI プロファイルを設定する

Amazon Braket Boto3 クライアントを有効にする

Amazon Braket で Boto3 を使用するには、Boto3 をインポートし、Amazon Braket API への接続に使用するクライアントを定義する必要があります。次の例では、Boto3 クライアントに `braket` という名前が付けられています。

```
import boto3
import botocore

braket = boto3.client("braket")
```

> **Note**
>
> Braket は IPv6 をサポートしています。IPv6 専用ネットワークを使用する場合、またはワークロードが IPv6 トラフィックを使用していることを確認したい場合は、「デュアルスタックと FIPS エンドポイント」というガイドに示されているように、デュアルスタックエンドポイントを使用します。

braket クライアントが確立されたら、Amazon Braket サービスからリクエストを行い、応答を処理することができます。リクエストとレスポンスのデータの詳細については、API リファレンスをご覧ください。

次の例は、デバイスと量子タスクの操作方法を示しています。

- デバイスを検索します。
- デバイスを取得する
- 量子タスクを作成する
- 量子タスクを取得する
- 量子タスクを検索する
- 量子タスクをキャンセルする

デバイスを検索します。

- search_devices(**kwargs)

指定したフィルターを使用してデバイスを検索します。

```
# Pass search filters and optional parameters when sending the
# request and capture the response
response = braket.search_devices(filters=[{
    'name': 'deviceArn',
    'values': ['arn:aws:braket:::device/quantum-simulator/amazon/sv1']
}], maxResults=10)

print(f"Found {len(response['devices'])} devices")

for i in range(len(response['devices'])):
    device = response['devices'][i]
    print(device['deviceArn'])
```

デバイスを取得する

- get_device(deviceArn)

Amazon Braket で利用可能なデバイスを取得します。

```
# Pass the device ARN when sending the request and capture the repsonse
response = braket.get_device(deviceArn='arn:aws:braket:::device/quantum-simulator/
amazon/sv1')

print(f"Device {response['deviceName']} is {response['deviceStatus']}")
```

量子タスクを作成する

- create_quantum_task(**kwargs)

量子タスクを作成します。

```
# Create parameters to pass into create_quantum_task()
kwargs = {
    # Create a Bell pair
    'action': '{"braketSchemaHeader": {"name": "braket.ir.jaqcd.program", "version":
 "1"}, "results": [], "basis_rotation_instructions": [], "instructions": [{"type": "h",
 "target": 0}, {"type": "cnot", "control": 0, "target": 1}]}',
    # Specify the SV1 Device ARN
    'deviceArn': 'arn:aws:braket:::device/quantum-simulator/amazon/sv1',
    # Specify 2 qubits for the Bell pair
    'deviceParameters': '{"braketSchemaHeader": {"name":
 "braket.device_schema.simulators.gate_model_simulator_device_parameters",
 "version": "1"}, "paradigmParameters": {"braketSchemaHeader": {"name":
 "braket.device_schema.gate_model_parameters", "version": "1"}, "qubitCount": 2}}',
    # Specify where results should be placed when the quantum task completes.
    # You must ensure the S3 Bucket exists before calling create_quantum_task()
    'outputS3Bucket': 'amazon-braket-examples',
    'outputS3KeyPrefix': 'boto-examples',
    # Specify number of shots for the quantum task
    'shots': 100
}

# Send the request and capture the response
response = braket.create_quantum_task(**kwargs)

print(f"Quantum task {response['quantumTaskArn']} created")
```

量子タスクを取得する

- get_quantum_task(quantumTaskArn)

指定した量子タスクを取得します。

```
# Pass the quantum task ARN when sending the request and capture the response
response = braket.get_quantum_task(quantumTaskArn='arn:aws:braket:us-
west-1:123456789012:quantum-task/ce78c429-cef5-45f2-88da-123456789012')

print(response['status'])
```

量子タスクを検索する

- search_quantum_tasks(**kwargs)

指定したフィルター値に一致する量子タスクを検索します。

```
# Pass search filters and optional parameters when sending the
# request and capture the response
response = braket.search_quantum_tasks(filters=[{
    'name': 'deviceArn',
    'operator': 'EQUAL',
    'values': ['arn:aws:braket:::device/quantum-simulator/amazon/sv1']
}], maxResults=25)

print(f"Found {len(response['quantumTasks'])} quantum tasks")

for n in range(len(response['quantumTasks'])):
    task = response['quantumTasks'][n]
    print(f"Quantum task {task['quantumTaskArn']} for {task['deviceArn']} is
 {task['status']}")
```

量子タスクをキャンセルする

- cancel_quantum_task(quantumTaskArn)

指定した量子タスクをキャンセルします。

```
# Pass the quantum task ARN when sending the request and capture the response
response = braket.cancel_quantum_task(quantumTaskArn='arn:aws:braket:us-
west-1:123456789012:quantum-task/ce78c429-cef5-45f2-88da-123456789012')

print(f"Quantum task {response['quantumTaskArn']} is {response['cancellationStatus']}")
```

Boto3 と Braket SDK の AWS CLI プロファイルを設定する

Amazon Braket SDK は、明示的に指定しない限り、デフォルトの AWS CLI 認証情報に依存します。マネージド Amazon Braket ノートブックで実行する場合は、デフォルトのままにすることをお勧めします。これは、ノートブックインスタンスを起動する権限を持つ IAM ロールを提供する必要があるためです。

オプションで、コードをローカルで (Amazon EC2 インスタンスなどで) 実行する場合、名前付き AWS CLI プロファイルを確立できます。デフォルトのプロファイルを定期的に上書きするのではなく、各プロファイルに異なる権限セットを与えることができます。

このセクションでは、そのような CLI `profile`を設定する方法と、そのプロファイルを Amazon Braket に組み込み、そのプロファイルからのアクセス許可で API コールが行われるようにする方法を簡潔に説明します。

このセクションの内容:

- ステップ 1: ローカル CLI AWS を設定する profile
- ステップ 2: Boto3 セッションオブジェクトを作成する
- ステップ 3: Boto3 セッションを Braket AwsSession に組み込む

ステップ 1: ローカル CLI AWS を設定する **`profile`**

ユーザーの作成方法およびデフォルト以外のプロファイルの設定方法について説明することは、このドキュメントの範囲を超えています。これらのトピックの詳細については、以下を参照してください。

- 開始方法
- を使用する AWS CLI ように を設定する AWS IAM アイデンティティセンター

Amazon Braket を使用するには、このユーザーおよび関連する CLI `profile`に対し、必要な Braket 権限を付与する必要があります。例えば、AmazonBraketFullAccessポリシーをアタッチできます。

ステップ 2: Boto3 セッションオブジェクトを作成する

Boto3 セッションオブジェクトを作成するには、次のコード例を使用します。

```
from boto3 import Session
```

```
# Insert CLI profile name here
boto_sess = Session(profile_name=`profile`)
```

> **Note**
>
> 必要な API コールにリージョンベースの制限があり、`profile` のデフォルトのリージョンに一致しない場合、次の例に示すように、Boto3 セッションのリージョンを指定できます。

```
# Insert CLI profile name _and_ region
boto_sess = Session(profile_name=`profile`, region_name=`region`)
```

として指定された引数で`region`、、 など Amazon Braket AWS リージョン が利用可能な のいずれかに対応する値を置き換え`us-east-1`、`us-west-1`ます。

ステップ 3: Boto3 セッションを Braket AwsSession に組み込む

次の例は、Boto3 Braket セッションを初期化し、そのセッションでデバイスをインスタンス化する方法を示しています。

```
from braket.aws import AwsSession, AwsDevice

# Initialize Braket session with Boto3 Session credentials
aws_session = AwsSession(boto_session=boto_sess)

# Instantiate any Braket QPU device with the previously initiated AwsSession
sim_arn = 'arn:aws:braket:::device/quantum-simulator/amazon/sv1'
device = AwsDevice(sim_arn, aws_session=aws_session)
```

この設定が完了したら、インスタンス化された `AwsDevice` オブジェクトに量子タスクを送信できます (それには、例えば `device.run(…)` コマンドを呼び出します)。そのデバイスによって行うすべての API コールには、以前に `profile` と指定した CLI プロファイルに関連付けられている IAM 認証情報を活用できます。

Amazon Braket を使用した量子タスクのテスト

Amazon Braket は、実際の量子ハードウェアで量子アルゴリズムを実行する前にテストおよび検証するのに役立つ、ハイパフォーマンスのさまざまな量子回路シミュレーターを提供しています。これらのシミュレーターは、基盤となる複雑なソフトウェアとインフラストラクチャ、および Amazon Elastic Compute Cloud (Amazon EC2) クラスターを処理できるため、古典的なハイパフォーマンスコンピューティング (HPC) インフラストラクチャで量子回路をシミュレートする負担を取り除きます。これらのリソースを使用することで、量子アプリケーションの開発と最適化に専念できます。

Braket のシミュレーターを使用することで、物理量子デバイスの制約や制限を受けずに、量子回路と量子アルゴリズムを徹底的にテストすることができます。これにより、基本的な量子ゲートや量子回路から、より高度な量子アルゴリズムやエラー緩和手法まで、量子コンピューティングのさまざまな概念を探求できます。

Braket SDK は、ショット数やノイズモデルなどのシミュレーションパラメータを制御できるようにしてシミュレーターへの量子タスクの送信を簡素化することで、量子アルゴリズムの動作をよりよく理解できるようにしています。また、Amazon Braket Hybrid Jobs の機能を使用して古典コンピューティング要素と量子コンピューティング要素を組み合わせることで、テストと検証の範囲をさらに拡大することもできます。

量子タスクを実際の量子ハードウェアにデプロイする前に、Braket のシミュレーターで徹底的にテストすることで、貴重なインサイトを得てアルゴリズムを改良し、正しいかどうかを確認することができます。これにより、開発時間を短縮し、エラーを最小限に抑え、量子コンピューティング分野の進歩を最終的に加速できます。

このセクションの内容:

- シミュレーターへの量子タスクの送信
- ローカル量子デバイスエミュレーター

シミュレーターへの量子タスクの送信

Amazon Braket は、量子タスクをテストできる複数のシミュレーターへのアクセスを提供します。単一の量子タスクを個別に送信・実行することも、複数のプログラムを実行することもできます。

シミュレーター

- 密度マトリックスシミュレーター、DM1: `arn:aws:braket:::device/quantum-simulator/amazon/dm1`

- 状態ベクトルシミュレーター、SV1: `arn:aws:braket:::device/quantum-simulator/amazon/sv1`
- テンソルネットワークシミュレーター、TN1: `arn:aws:braket:::device/quantum-simulator/amazon/tn1`
- ローカルシミュレーター : `LocalSimulator()`

> **Note**
>
> QPU およびオンデマンドシミュレーターでは、CREATED 状態の量子タスクをキャンセルできます。QPU およびオンデマンドシミュレーターでは、QUEUED 状態の量子タスクをベストエフォートベースでキャンセルできます。QPU の QUEUED 状態の量子タスクは、QPU の可用性ウィンドウ中は正常にキャンセルされないことに注意してください。

このセクションの内容:

- ローカル状態ベクトルシミュレーター (braket_sv)
- ローカル密度マトリックスシミュレーター (braket_dm)
- ローカル AHS シミュレーター (braket_ahs)
- 状態ベクトルシミュレーター (SV1)
- 密度マトリックスシミュレーター (DM1)
- テンソルネットワークシミュレーター (TN1)
- 埋め込みシミュレーターについて
- Amazon Braket シミュレーターの比較
- Amazon Braket での量子タスクの例
- ローカルシミュレーターを使用した量子タスクのテスト

ローカル状態ベクトルシミュレーター (`braket_sv`)

ローカル状態ベクトルシミュレーター (`braket_sv`) は、お客様のローカル環境で実行される Amazon Braket SDK の構成要素です。Braket ノートブックインスタンスまたはローカル環境のハードウェア仕様に応じた 25 qubits までの小型回路でのラピッドプロトタイピングに適しています。

このローカルシミュレーターは Amazon Braket SDK のすべてのゲートをサポートしていますが、QPU デバイスではより小さなサブセットがサポートされています。デバイスのサポートされているゲートは、デバイスのプロパティで確認できます。

> **Note**
>
> このローカルシミュレーターは高度な OpenQASM 機能をサポートしていますが、高度な OpenQASM 機能は QPU デバイスやその他のシミュレーターではサポートされていない場合があります。サポートされている機能の詳細については、「OpenQASM Local Simulator notebook」に記載されている例を参照してください。

シミュレーターを使用する方法については、「Amazon Braket の例」を参照してください。

ローカル密度マトリックスシミュレーター (`braket_dm`)

ローカル密度マトリックスシミュレーター (`braket_dm`) は、お客様のローカル環境で実行される Amazon Braket SDK の構成要素です。Braketノートブックインスタンスまたはローカル環境のハードウェア仕様に応じて、ノイズ (最大 12 qubits) の小さな回路でのラピッドプロトタイピングに適しています。

ビットフリップや脱分極誤差などのゲートノイズ演算を使用して、一般的なノイズの多い回路をゼロから構築できます。また、ノイズの有無にかかわらず動作することを意図した既存の回路の特定の qubits およびゲートにノイズ演算を適用することもできます。

`braket_dm` ローカルシミュレーターでは、指定した shots 数を指定すると、次の結果が得られます。

- 縮約密度マトリックス: Shots = 0

> **Note**
>
> このローカルシミュレーターは高度な OpenQASM 機能をサポートしていますが、高度な OpenQASM 機能は QPU デバイスやその他のシミュレーターではサポートされていない場合があります。サポートされている機能の詳細については、「OpenQASM Local Simulator notebook」に記載されている例を参照してください。

ローカル密度マトリックスシミュレーターの詳細については、「Braket 入門ノイズシミュレーターの例」を参照してください。

ローカル AHS シミュレーター (**braket_ahs**)

ローカル AHS (アナログハミルトニアンシミュレーション) シミュレーター (braket_ahs) は、お客様のローカル環境で実行される Amazon Braket SDK の構成要素です。これを使用することで、AHS プログラムの結果をシミュレートできます。ローカル AHS シミュレーターは、Braket ノートブックインスタンスまたはローカル環境のハードウェア仕様に応じて、最大 10～12 原子の小型レジスターでのプロトタイピングに適しています。

ローカルシミュレーターは、1 つの均一な駆動場、1 つの (不均一な) 移動磁場、および任意の原子配置について、AHS プログラムをサポートしています。詳細については、Braket の「AHS class」と Braket の「AHS program schema」を参照してください。

ローカル AHS シミュレーターの詳細については、「Hello AHS: 初めてのアナログハミルトニアンシミュレーションを実行する」ページと「Analog Hamiltonian Simulation example notebooks」を参照してください。

状態ベクトルシミュレーター (SV1)

SV1 は、オンデマンド、ハイパフォーマンスのユニバーサル状態ベクトルシミュレーターです。最大で 34 qubits の回路をシミュレートできます。34-qubit、高密度、正方形の回路 (回路の深さ = 34) の完了までは、使用するゲートのタイプやその他の要因に応じて約 1～2 時間かかることが予想されます。オールツーオールゲートを備えた回路は、SV1 に適しています。完全な状態ベクトルや振幅の配列などの形式で結果を返します。

SV1 の最大実行時間は 6 時間です。デフォルトでは、35 個の同時量子タスクを持ち、最大では 100 個の同時量子タスクを持てます (us-west-1 と eu-west-2 で 50 個ずつ)。

SV1 の結果

SV1 で以下の左の各結果を得るには、以下の右の shots 数を指定する必要があります。

- 標本: Shots > 0
- 期待値: Shots >= 0
- 分散: Shots >= 0
- 確率: Shots > 0

- 振幅: Shots = 0
- 随伴勾配: Shots = 0

結果の詳細については、「結果タイプ」を参照してください。

SV1 は常に利用可能で、オンデマンドで回路を実行し、複数の回路を並列に実行できます。実行時間は、オペレーションの数に応じて線形にスケールされ、qubits 数とともに指数関数的にスケールされます。shots 数は、実行時間へは少ししか影響しません。詳細については、「シミュレーターを比較する」を参照してください。

シミュレーターは Braket SDK のすべてのゲートをサポートしますが、QPU デバイスは小さなサブセットをサポートします。デバイスのサポートされているゲートは、デバイスのプロパティで確認できます。

密度マトリックスシミュレーター (DM1)

DM1 は、オンデマンド、ハイパフォーマンスの密度マトリックスシミュレーターです。最大で 17 qubits の回路をシミュレートできます。

DM1 は、最大実行時間が 6 時間、同時量子タスク数のデフォルトは 35 個、同時量子タスク数の最大値は 50 個です。

DM1 の結果

DM1 で以下の左の各結果を得るには、以下の右の shots 数を指定する必要があります。

- 標本: Shots > 0
- 期待値: Shots >= 0
- 分散: Shots >= 0
- 確率: Shots > 0
- 縮約密度マトリックス: Shots = 0、最大 qubits 数 = 8

結果の詳細については、「結果タイプ」を参照してください。

DM1 は常に利用可能で、オンデマンドで回路を実行し、複数の回路を並列に実行できます。実行時間は、オペレーションの数に応じて線形にスケールされ、qubits 数とともに指数関数的にスケールされます。shots 数は、実行時間へは少ししか影響しません。詳細については、「シミュレーターを比較する」を参照してください。

ノイズゲートと制限事項

```
AmplitudeDamping
    Probability has to be within [0,1]
BitFlip
    Probability has to be within [0,0.5]
Depolarizing
    Probability has to be within [0,0.75]
GeneralizedAmplitudeDamping
    Probability has to be within [0,1]
PauliChannel
    The sum of the probabilities has to be within [0,1]
Kraus
    At most 2 qubits
    At most 4 (16) Kraus matrices for 1 (2) qubit
PhaseDamping
    Probability has to be within [0,1]
PhaseFlip
    Probability has to be within [0,0.5]
TwoQubitDephasing
    Probability has to be within [0,0.75]
TwoQubitDepolarizing
    Probability has to be within [0,0.9375]
```

テンソルネットワークシミュレーター (TN1)

TN1 は、オンデマンドで高性能なテンソルネットワークシミュレーターです。 は、最大 50 で回路深度qubitsが 100 以下の特定の回路タイプをシミュレートTN1できます。 TN1は、スパース回路、ローカルゲートを持つ回路、量子フーリエ変換 (QFT) 回路などの特殊な構造を持つ回路に特に強力です。 は 2 つのフェーズでTN1動作します。まず、リハーサルフェーズは回路の効率的な計算経路を特定しようとします。この特定結果により、TN1 が収縮フェーズと呼ばれる次の段階の実行時間を推定できます。推定収縮時間が TN1 シミュレーションの実行時間制限を超える場合、TN1 は収縮を試みません。

TN1 の実行時間制限は 6 時間です。同時量子タスク数の上限は、最大 10 個 (eu-west-2 では 5 個) です。

TN1 の結果

収縮フェーズは、マトリックスの一連の乗算で構成されます。一連の乗算は、結果に達するまで、または結果に到達できないと判断されるまで継続されます。

注: Shotsは > 0 である必要があります。

結果タイプは次のとおりです。

- サンプル
- 期待
- 分散

結果の詳細については、「結果タイプ」を参照してください。

TN1 は常に利用可能で、オンデマンドで回路を実行し、複数の回路を並列に実行できます。詳細については、「シミュレーターを比較する」を参照してください。

シミュレーターは Braket SDK のすべてのゲートをサポートしますが、QPU デバイスは小さなサブセットをサポートします。デバイスのサポートされているゲートは、デバイスのプロパティで確認できます。

TN1 を開始するには、Amazon Braket GitHub リポジトリの「TN1 example notebook」を参照してください。

TN1 を使用するためのベストプラクティス

- オールツーオール回路は避けてください。
- TN1 の回路の「難易度」を学ぶために、shots数が少ない回路クラスまたは新しい回路をテストします。
- 大きな shot のシミュレーションは複数の量子タスクに分割します。

埋め込みシミュレーターについて

埋め込みシミュレーターは、シミュレーションをアルゴリズムコードに直接埋め込むことで動作します。また、同じコンテナ内に格納されており、ハイブリッドジョブインスタンスに対するシミュレーションを直接実行します。このアプローチは、シミュレーションとリモートデバイス間の通信で通常発生するボトルネックを排除するのに役立ちます。埋め込みシミュレーターは、すべての計算を 1 つのまとまりのある環境で維持することで、必要なメモリを大幅に削減し、目標となる成果を達成するために必要な回路実行の数を減らすことができます。これにより、リモートシミュレーションに依存する従来の環境と比較して、多くの場合、10 倍以上の大幅なパフォーマンス向上につながる可能性があります。埋め込みシミュレーターがパフォーマンスを向上させ、効率的なハイブリッドジョブ

を可能にする方法の詳細については、「Run a hybrid job with Amazon Braket Hybrid Jobs」のドキュメントページを参照してください。

PennyLane の稲妻シミュレーター

PennyLane の稲妻シミュレーターを Braket の埋め込みシミュレーターとして使用できます。PennyLane の稲妻シミュレーターを使用すると、随伴微分法などの高度な勾配計算方法を使用して、勾配をより迅速に評価できます。lightning.qubit シミュレーターは、Braket NBI を使用してデバイスとして使用できますが、埋め込みシミュレーターとしては GPU インスタンスを使用して実行する必要があります。lightning.gpu の使用例については、「Embedded simulators in Braket Hybrid Jobs」ノートブックを参照してください。

Amazon Braket シミュレーターの比較

このセクションでは、いくつかの概念、制限、ユースケースを説明して、量子タスクに最も適した Amazon Braket シミュレーターを選択するのに役立ちます。

ローカルシミュレーターとオンデマンドシミュレーター (SV1、TN1、DM1) からの選択

ローカルシミュレーターのパフォーマンスは、シミュレーターの実行に使用する Braket ノートブックインスタンスなどのローカル環境をホストするハードウェアによって異なります。オンデマンドシミュレーターは AWS クラウドで実行され、一般的なローカル環境を超えてスケールするように設計されています。オンデマンドシミュレーターは、より大きな回路用に最適化されていますが、量子タスクまたは量子タスクのバッチごとにレイテンシーオーバーヘッドが掛かります。これは、多くの量子タスクが関与する場合にはトレードオフを意味する可能性があります。これらの一般的なパフォーマンス特性を考慮すると、次のガイダンスは、ノイズのあるシミュレーションを含むシミュレーションの実行方法を選択するのに役立ちます。

シミュレーションの場合:

- 使用する qubits が 18 未満の場合は、ローカルシミュレーターを使用します。
- 使用する qubits が 18 ~ 24 の場合は、ワークロードに応じたシミュレーターを選択します。
- 使用する qubits が 24 を超える場合は、オンデマンドシミュレーターを使用します。

ノイズシミュレーションの場合:

- 使用する qubits が 9 未満の場合は、ローカルシミュレーターを使用します。
- 使用する qubits 9 ~ 12 の場合は、ワークロードに応じたシミュレーターを選択します。

- 使用する qubits が 12 を超える場合は、DM1 を使用します。

状態ベクトルシミュレーターとは何ですか?

SV1 はユニバーサル状態ベクトルシミュレーターです。量子状態の全波動関数を格納し、ゲート演算を状態に順次適用します。それは、非常にありそうもないものであっても、すべての可能性を格納します。SV1 シミュレーターによる量子タスクの実行時間は、回路内のゲート数に比例して増大します。

密度マトリックスシミュレーターとは何ですか?

DM1 はノイズのある量子回路をシミュレートします。システムの全密度マトリックスを保存し、回路のゲートとノイズ演算を順次適用します。最終的な密度マトリックスには、回路の実行後の量子状態の完全な情報が含まれています。通常、実行時間は操作数に応じて直線的にスケールされ、qubits 数に応じて指数関数的にスケールされます。

テンソルネットワークシミュレーターとは何ですか?

TN1 は量子回路を構造化グラフにエンコードします。

- グラフのノードは、量子ゲート、つまりqubitsで構成されます。
- グラフのエッジは、ゲート間の接続を表します。

この構造の結果、TN1 は比較的大規模で複雑な量子回路のシミュレーション解を見つけることができます。

TN1 には 2 つのフェーズが必要

通常、TN1 は量子計算をシミュレートする 2 フェーズアプローチで動作します。

- リハーサルフェーズ: このフェーズでは、TN1 は効率的な方法でグラフをトラバースする方法を考え出します。これには、すべてのノードにアクセスして、目的の測定値を取得できます。TN1 が両方のフェーズを一緒に実行するため、ユーザーにはこのフェーズが表示されません。TN1 は、第 1 フェーズを完了し、実用的な制約に基づいて、第 2 フェーズを実行するかどうかを TN1 自身で決定します。シミュレーションが開始すれば、この決定のための入力をユーザーが行うことはありません。
- 収縮フェーズ: このフェーズは、古典的なコンピュータにおける計算の実行フェーズに似ています。このフェーズは、一連のマトリックス乗算で構成されます。これらの乗算の順序は、計算の難しさに大きな影響を与えます。したがって、グラフ全体で最も効果的な計算パスを見つけるた

めに、最初にリハーサルフェーズを実行します。リハーサルフェーズ中に収縮経路が検出されると、TN1 は回路のゲートを集約してシミュレーションの結果を生成します。

TN1 グラフは地図に似ている

比喩的に、基礎となる TN1 グラフを都市の道路と比較できます。計画されたグリッドがある都市では、地図を使用して目的地までのルートを簡単に見つけることができます。計画外の道路や道路名が重複している都市では、地図を見て目的地までのルートを見つけるのが難しい場合があります。

TN1 がリハーサルフェーズを行わなかった場合、最初に地図を見る代わりに、街の通りを歩いて目的地を見つけるようなものになります。地図を見るのにより多くの時間を費やすことは、歩く時間という点で本当に報われるでしょう。同様に、リハーサルフェーズは貴重な情報を提供します。

TN1 は通過する基礎回路の構造について特定の「認識」を持っていると言えるかもしれません。この意識はリハーサルフェーズ中に高まります。

これらのタイプのシミュレーターに最も適した問題の種類

SV1 は、主に特定の数の qubits とゲートを持つことに依存する問題のクラスに適しています。一般的に、必要な時間はゲートの数に応じて直線的に増加しますが、shots 数には依存しません。SV1 は 28 未満のTN1の回路の qubits よりも通常、高速です。

SV1 は、非常にありそうもないものであっても、実際にはすべての可能性をシミュレートするため、より高い qubit 数では遅くなる可能性があります。どの結果が出そうなのかを判断する方法はありません。したがって、30-qubit の評価では、SV1 は 2^30 の設定を計算する必要があります。Amazon Braket SV1 シミュレーターの 34 qubitsの制限は、メモリとストレージの制約による実際的な制約です。これは、「SV1 に 1 qubit を追加するたびに、問題は 2 倍難しくなる」と考えることができます。

多くのクラスの問題について、TN1 は SV1 よりもはるかに大きな回路を現実的な時間で評価できます。TN1 はグラフの構造を利用するためです。TN1 は、基本的には、最初から解の進化を追跡し、効率的なトラバーサルに寄与する設定のみを保持します。別の言い方をすると、TN1 はマトリックス乗算の順序を作成する設定を保存することで、評価プロセスをよりシンプルにします。

TN1 の場合、qubits とゲートの数は重要ですが、グラフの構造はもっと重要です。例えば、TN1 は、ゲートが短距離である回路 (グラフ) を評価するのに非常に優れており (つまり、各 qubit はその近傍 qubits にのみゲートによって接続される) 、接続 (またはゲート) が類似の範囲を持つ回路 (グラフ) を評価します。TN1 にとって典型的な範囲は、各 qubit が他の qubits と 5 qubits 離れている位置でのみ相互作用することです。構造の大部分がこれらのようなより単純な関係に分解できれば、そう

いった単純な関係はより大きい、またはより小さい、あるいはより均一なマトリックスで表すことができるため、TN1 は評価を効率的に実行するようになります。

TN1 の制約事項

グラフの構造的複雑さSV1によっては、TN1 は よりも遅くなる場合があります。いくつかのグラフの場合、TN1 はリハーサルステージの後にシミュレーションを終了し、次の 2 つの理由のいずれかで FAILED ステータスを表示します。

- パスが見つかりません — グラフが複雑すぎると、良好なトラバーサル経路を見つけるのが難しくなりすぎ、シミュレーターが計算をあきらめます。TN1 は収縮を実行できなくなります。以下のようなエラーメッセージが表示されることがあります: `No viable contraction path found.`
- 収縮段階が難しすぎる — 一部のグラフでは、TN1 はトラバーサルパスを見つけることができますが、非常に長く、評価に非常に時間がかかります。この場合、収縮は非常に高価であるため、コストは法外になります。このため、TN1 は収縮せずにリハーサルフェーズの後に終了します。以下のようなエラーメッセージが表示されることがあります: `Predicted runtime based on best contraction path found exceeds TN1 limit.`

> **Note**
>
> 収縮を行わない場合でも、TN1 のリハーサルステージの料金が請求され、FAILED ステータスが表示されます。

また、予測される実行時間は shot 数に依存します。最悪のシナリオでは、TN1 収縮時間は shot 数に線形に依存します。shots 数が少ない場合、回路が収縮する可能性があります。例えば、ユーザーが 100 shotsで量子タスクを送信すると、TN1 が収縮不可能として判断しますが、10 ショットだけで再送信すると、収縮が進行します。この状況では、100 個の標本を取得するために、同じ回路に対して10 shotsの 10 個の量子タスクを送信し、最終的に複数の結果を合成することができます。

ベストプラクティスとして、より多くのshotsを実行する前に、回路または回路クラスを数shots (例えば 10) でテストし、TN1 の回路の難易度を調べることをお勧めします。

> **Note**
>
> 収縮フェーズを形成する一連の乗算は、小さな NxN マトリックスで始まります。例えば、2-qubitのゲートには 4x4 マトリックスが必要です。難しすぎると判断される収縮中に必要

な中間行列は巨大です。このような計算には、完了までに数日かかるでしょう。だからこそ、Amazon Braket は極端に複雑な収縮を試みないのです。

同時実行

すべての Braket シミュレーターを使用すると、複数の回路を同時に実行できます。同時実行数の制限は、シミュレーターとリージョンによって異なります。同時実行数量の制限の詳細については、「Quotas」ページを参照してください。

Amazon Braket での量子タスクの例

このセクションでは、デバイスの選択から結果の表示まで、量子タスク例の実行段階を順を追って説明します。Amazon Braket のベストプラクティスとして、まず SV1 などのシミュレーターで回路を実行することをお勧めします。

このセクションの内容:

- デバイスを指定する
- 量子タスクの例を送信する
- パラメータ化されたタスクを送信する
- shots を指定する
- 結果のポーリング
- 結果の例の表示

デバイスを指定する

まず、量子タスクのデバイスを選択して指定します。この例では、シミュレーター SV1 を選択する方法を示します。

```
from braket.aws import AwsDevice

# Choose the on-demand simulator to run the circuit
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
```

このデバイスのプロパティは、次のように表示できます。

```
print(device.name)
```

```
for iter in device.properties.action['braket.ir.jaqcd.program']:
    print(iter)
```

```
SV1
('version', ['1.0', '1.1'])
('actionType', 'braket.ir.jaqcd.program')
('supportedOperations', ['ccnot', 'cnot', 'cphaseshift', 'cphaseshift00',
 'cphaseshift01', 'cphaseshift10', 'cswap', 'cy', 'cz', 'ecr', 'h', 'i', 'iswap',
 'pswap', 'phaseshift', 'rx', 'ry', 'rz', 's', 'si', 'swap', 't', 'ti', 'unitary', 'v',
 'vi', 'x', 'xx', 'xy', 'y', 'yy', 'z', 'zz'])
('supportedResultTypes', [ResultType(name='Sample', observables=['x', 'y', 'z', 'h',
 'i', 'hermitian'], minShots=1, maxShots=100000), ResultType(name='Expectation',
 observables=['x', 'y', 'z', 'h', 'i', 'hermitian'], minShots=0, maxShots=100000),
 ResultType(name='Variance', observables=['x', 'y', 'z', 'h', 'i', 'hermitian'],
 minShots=0, maxShots=100000), ResultType(name='Probability', observables=None,
 minShots=1, maxShots=100000), ResultType(name='Amplitude', observables=None,
 minShots=0, maxShots=0)])
('disabledQubitRewiringSupported', None)
```

量子タスクの例を送信する

オンデマンドシミュレーターで実行する量子タスクの例を送信します。

```
from braket.circuits import Circuit, Observable

# Create a circuit with a result type
circ = Circuit().rx(0, 1).ry(1, 0.2).cnot(0, 2).variance(observable=Observable.Z(),
 target=0)
# Add another result type
circ.probability(target=[0, 2])

# Set up S3 bucket (where results are stored)
my_bucket = "amazon-braket-s3-demo-bucket"  # The name of the bucket
my_prefix = "your-folder-name"  # The name of the folder in the bucket
s3_location = (my_bucket, my_prefix)

# Submit the quantum task to run
my_task = device.run(circ, s3_location, shots=1000, poll_timeout_seconds=100,
 poll_interval_seconds=10)
# The positional argument for the S3 bucket is optional if you want to specify a bucket
 other than the default

# Get results of the quantum task
```

```
result = my_task.result()
```

device.run() コマンドを使用すると、CreateQuantumTask APIを介してタスクが作成されます。短い初期化時間が経過すると、量子タスクはデバイス上で実行するための容量ができるまでキューに入れられます。この場合、デバイスは SV1 です。デバイスが計算を完了すると、Amazon Braket は呼び出しで指定された Amazon S3 の場所に結果を書き込みます。位置引数 s3_location はローカルシミュレーターを除くすべてのデバイスで必要です。

> **Note**
>
> Braket 量子タスクアクションのサイズは 3MB に制限されています。

パラメータ化されたタスクを送信する

Amazon Braket オンデマンドおよびローカルシミュレーターと QPU では、タスクの送信時に自由パラメータの値を指定することも可能です。これを行うには、下記の例に示すように device.run() の inputs 引数を使用します。inputs には、文字列と浮動小数点のペアのディクショナリを指定する必要があります。ここで、キーはパラメータ名です。

パラメトリックコンパイルにより、特定の QPU 上でパラメトリック回路を実行する際のパフォーマンスが向上します。サポートされている QPU に量子タスクとしてパラメトリック回路を送信すると、Braket は回路を 1 回コンパイルし、結果をキャッシュします。同じ回路への後続のパラメータ更新の再コンパイルは行われないため、同じ回路を使用するタスクの実行時間が短縮されます。Braket は、回路をコンパイルするときに、ハードウェアプロバイダーからの更新されたキャリブレーションデータを自動的に使用して、最高品質の結果を実現します。

> **Note**
>
> パラメトリックコンパイルは、Rigetti Computing 製のあらゆる超伝導ゲートベース QPU でサポートされています。ただし、パルスレベルプログラムは例外です。

```
from braket.circuits import Circuit, FreeParameter, Observable

# Create the free parameters
alpha = FreeParameter('alpha')
beta = FreeParameter('beta')
```

```
# Create a circuit with a result type
circ = Circuit().rx(0, alpha).ry(1, alpha).cnot(0, 2).xx(0, 2, beta)
circ.variance(observable=Observable.Z(), target=0)

# Add another result type
circ.probability(target=[0, 2])

# Submit the quantum task to run
my_task = device.run(circ, inputs={'alpha': 0.1, 'beta': 0.2}, shots=100)
```

shots を指定する

shots 引数は、必要な測定 shots の数を示します。SV1 などのシミュレーターは、2 つのシミュレーションモードをサポートしています。

- shots = 0 の場合、シミュレーターは正確なシミュレーションを実行し、すべての結果タイプの真の値を返します。(このモードは TN1 にはありません。)
- shots がゼロ以外の値の場合、シミュレーターは出力分布からサンプリングし、実際の QPU の shot ノイズをエミュレートします。QPU デバイスでは 0 を超える shots 数しか許可されません。

量子タスクあたりの最大ショット数の詳細については、「Braket Quotas」を参照してください。

結果のポーリング

`my_task.result()` の実行時、SDK は結果のポーリングを開始し、量子タスクの作成時に定義したパラメータを使用します。

- `poll_timeout_seconds` は、オンデマンドシミュレーターまたは QPU デバイスで量子タスクを実行するときに、量子タスクがタイムアウトするまでのポーリングにかかる秒数です。デフォルト値は 432,000 秒 (5 日) です。
- 注意: Rigetti や IonQ などの QPU デバイスの場合、数日かけることを推奨します。ポーリングタイムアウトが短すぎると、ポーリング時間内に結果が返されないことがあります。例えば、QPU が利用できない場合、ローカルタイムアウトエラーが返されます。
- `poll_interval_seconds` は、量子タスクがポーリングされる頻度です。これには、量子タスクがオンデマンドシミュレーターおよび QPU デバイスで実行されるときに、ステータスを取得するために Braket API を呼び出す頻度を指定します。デフォルト値は 1 秒です。

この非同期実行により、常に使用できるとは限らない QPU デバイスの操作が容易になります。例えば、通常のメンテナンス期間中はデバイスを使用できない場合があります。

返される結果には、量子タスクに関連付けられたメタデータの範囲が含まれています。測定結果は、次のコマンドで確認できます。

```
print('Measurement results:\n', result.measurements)
print('Counts for collapsed states:\n', result.measurement_counts)
print('Probabilities for collapsed states:\n', result.measurement_probabilities)
```

```
Measurement results:
 [[1 0 1]
 [0 0 0]
 [0 0 0]
 ...
 [0 0 0]
 [0 0 0]
 [1 0 1]]
Counts for collapsed states:
 Counter({'000': 766, '101': 220, '010': 11, '111': 3})
Probabilities for collapsed states:
 {'101': 0.22, '000': 0.766, '010': 0.011, '111': 0.003}
```

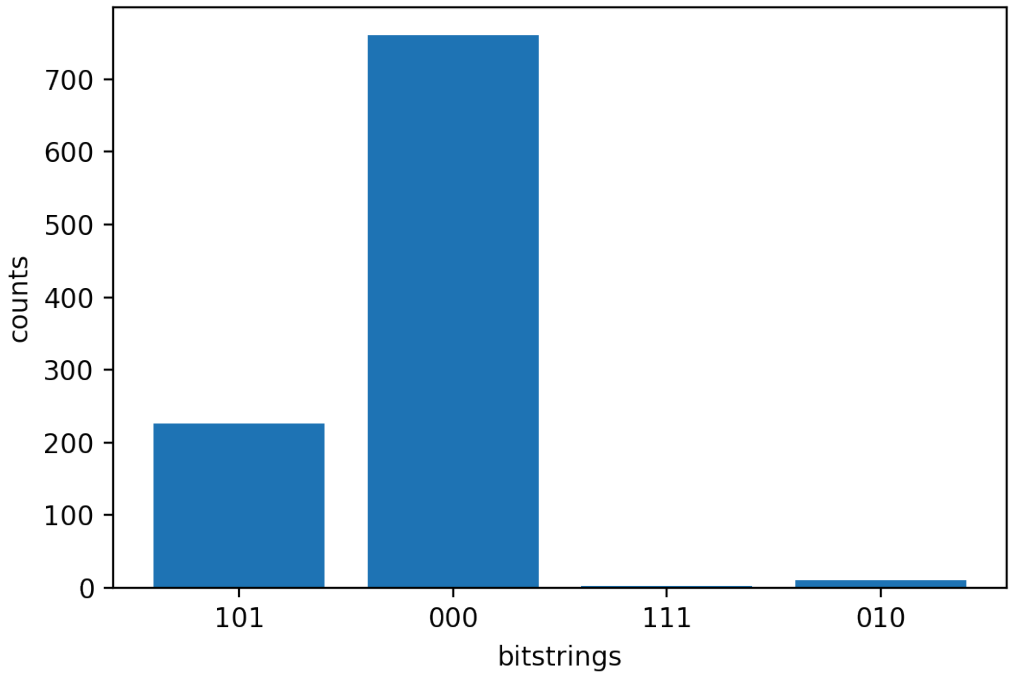
結果の例の表示

ResultType も指定したので、返される結果を表示できます。結果タイプは、回路に追加された順に表示されます。

```
print('Result types include:\n', result.result_types)
print('Variance=', result.values[0])
print('Probability=', result.values[1])

# Plot the result and do some analysis
import matplotlib.pyplot as plt
plt.bar(result.measurement_counts.keys(), result.measurement_counts.values())
plt.xlabel('bitstrings')
plt.ylabel('counts')
```

```
Result types include:
```

```
 [ResultTypeValue(type=Variance(observable=['z'], targets=[0], type=<Type.variance:
 'variance'>), value=0.693084), ResultTypeValue(type=Probability(targets=[0, 2],
 type=<Type.probability: 'probability'>), value=array([0.777, 0.   , 0.   , 0.223]))]
Variance= 0.693084
Probability= [0.777 0.    0.    0.223]
Text(0, 0.5, 'counts')
```



ローカルシミュレーターを使用した量子タスクのテスト

ローカルシミュレーターに量子タスクを直接送信して、迅速なプロトタイピングとテストを行うことができます。このシミュレーターはローカル環境で実行されるため、Amazon S3 の場所を指定する必要はありません。結果はセッションで直接計算されます。ローカルシミュレーターで量子タスクを実行するには、shots パラメータのみを指定する必要があります。

> **Note**
>
> ローカルシミュレーターが処理できる実行速度と最大qubits数は、Amazon Braket ノートブックインスタンスタイプ、またはローカルハードウェアの仕様によって異なります。

下記のコマンドはすべて等価であり、状態ベクトル (ノイズフリー) ローカルシミュレーターをインスタンス化しています。

```
# Import the LocalSimulator module
from braket.devices import LocalSimulator

# The following are identical commands
device = LocalSimulator()
device = LocalSimulator("default")
device = LocalSimulator(backend="default")
device = LocalSimulator(backend="braket_sv")
```

次に、以下を使用して量子タスクを実行します。

```
my_task = device.run(circ, shots=1000)
```

ローカル密度マトリックス (ノイズ) シミュレーターをインスタンス化するためには、お客様がバックエンドを次のように変更するものとします。

```
# Import the LocalSimulator module
from braket.devices import LocalSimulator

device = LocalSimulator(backend="braket_dm")
```

ローカルシミュレーターで特定の量子ビットを測定する

ローカル状態ベクトルシミュレーターとローカル密度マトリックスシミュレーターは、回路の量子ビットのサブセットを測定できる回路の実行をサポートします。このような測定は、多くの場合、部分測定と呼ばれています。

例えば、次のコードでは、2 量子ビット回路を作成しており、ターゲット量子ビットを指定した measure 命令を回路の最後に追加することによってのみ、最初の量子ビットを測定できます。

```
# Import the LocalSimulator module
from braket.devices import LocalSimulator

# Use the local simulator device
device = LocalSimulator()

# Define a bell circuit and only measure
```

```
circuit = Circuit().h(0).cnot(0, 1).measure(0)

# Run the circuit
task = device.run(circuit, shots=10)

# Get the results
result = task.result()

# Print the measurement counts for qubit 0
print(result.measurement_counts)
```

ローカル量子デバイスエミュレーター

Amazon Braket のローカルエミュレーターツールを使用すると、逐語的な量子プログラムを実際の量子ハードウェアで実行する前に、ローカルでエミュレートできます。このエミュレータはデバイスキャリブレーションデータを使用して逐語的な回路を検証し、互換性の問題を早期に捕捉できるようにします。

また、ローカルエミュレーターは、次のプロセスを通じて量子ハードウェアノイズをシミュレートします。

- デバイスキャリブレーションデータを使用してノイズモデルを構築する
- 回路内の各ゲートに脱分極ノイズを適用する
- 回路の最後に読み出しエラーを適用する
- ローカル密度マトリックスシミュレーターを使用してノイズの多い回路をシミュレートする

ローカルエミュレーターの使用の詳細については、amazon-braket-examples GitHub リポジトリの「Local emulation for verbatim circuits on Amazon Braket」を参照してください。

このセクションの内容:

- ローカルエミュレーションの利点
- ローカルエミュレーターを作成する

ローカルエミュレーションの利点

- リアルタイムまたは過去のキャリブレーションデータを使用して、逐語的な回路をデバイスの制約と照らし合わせて検証できます。

- タスクを量子ハードウェアに送信する前に、問題をデバッグできます。
- ノイズのないエミュレーション、ノイズの多いエミュレーション、ハードウェアの結果を比較することで、ノイズの影響を理解できます。
- ノイズ対応量子アルゴリズムを開発するワークフローを効率化できます。

ローカルエミュレーターを作成する

ローカル量子デバイスエミュレーターは、量子デバイスまたは一連のデバイスプロパティから直接作成できます。デバイスを直接エミュレートする場合、エミュレーターはインスタンス化されたデバイスにある最新のキャリブレーションデータを使用します。次のコード例は、Rigetti's Ankaa-3 デバイスを直接エミュレートする方法を示しています。

```
from braket.aws.aws_device import AwsDevice

ankaa3 = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
ankaa3_emulator = ankaa3.emulator()
```

次の例は、Ankaa-3 デバイスの、JSON 形式の一連のプロパティからローカルデバイスエミュレーターを作成する方法を示しています。

```
from braket.aws import AwsDevice
from braket.emulation.local_emulator import LocalEmulator
import json

# Instantiate the device
ankaa3 = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
ankaa3_properties = ankaa3.properties

# Put the Ankaa-3 properties in a file named ankaa3_device_properties.json
with open("ankaa3_device_properties.json", "w") as f:
    json.dump(ankaa3_properties.json(), f)

# Load the json into the ankaa3_data_json variable
with open("ankaa3_device_properties.json", "r") as json_file:
    ankaa3_data_json = json.load(json_file)

# Create the Ankaa-3 local emulator from the json file you created
ankaa3_emulator = LocalEmulator.from_json(ankaa3_data_json)
```

上記の例をカスタマイズするには、以下の方法を使用します。

- 別の QPU デバイスのプロパティを使用する
- エミュレーターに別の JSON ファイルを指定する
- エミュレーターをインスタンス化する前にデバイスプロパティの値を変更する

Amazon Braket で量子タスクを実行する

Braket は、さまざまな種類の量子コンピューターへの安全なオンデマンドアクセスを提供します。、AQT、IonQ、IQMおよび のゲートベースの量子コンピュータとRigetti、QuEra のアナログハミルトニアンシミュレーターにアクセスできます。また、前払いの義務はなく、個々のプロバイダーを介したアクセスを確保する必要もありません。

- Amazon Braket コンソールは、リソースと量子タスクの作成、管理、モニタリングに役立つデバイス情報とステータスを提供します。
- Amazon Braket Python SDK およびコンソールを通じて、量子タスクを送信および実行できます。SDK は、事前設定された Amazon Braket ノートブックからアクセスできます。
- Amazon Braket API は、Amazon Braket Python SDK およびノートブックからアクセスできます。プログラムにより量子コンピューティングで動作するアプリケーションを構築している場合は、API を直接呼び出すことができます。

このセクションの例では、Amazon Braket Python SDK と AWS Python SDK for Braket (Boto3) を合わせて使用して Amazon Braket API を直接操作する方法を説明します。

Amazon Braket Python SDK の詳細

Amazon Braket Python SDK を使用するには、 と通信できるように、まず AWS Python SDK for Braket (Boto3) をインストールします AWS API。Amazon Braket Python SDK は、量子を取り扱うカスタマーにとって Boto3 の便利なラッパーと考えることができます。

- Boto3 には、 を利用するために必要なインターフェイスが含まれています AWS API。(Boto3 は と通信する大規模な Python SDK であることに注意してください AWS API。 ほとんどの は Boto3 インターフェイス AWS のサービス をサポートしています) 。
- Amazon Braket Python SDK には、回路、ゲート、デバイス、結果タイプ、および量子タスクの他部分用のソフトウェアモジュールが含まれています。プログラムを作成するたびに、その量子タスクに必要なモジュールをインポートします。
- Amazon Braket Python SDK は、量子タスクの実行に必要なすべてのモジュールと依存関係が事前にロードされているノートブックからアクセスできます。
- ノートブックで作業したくない場合は、Amazon Braket Python SDK から任意の Python スクリプトにモジュールをインポートできます。

Boto3 のインストール後に Amazon Braket Python SDK を使用して量子タスクを作成する手順の概要は、次のようになります。

1. (オプション) ノートブックを開く。
2. 回路に必要な SDK モジュールをインポートする。
3. QPU またはシミュレータを指定する。
4. 回路をインスタンス化する。
5. 回路を実行する。
6. 結果を収集する。

このセクションの例では、各ステップについて詳しく説明します。

その他の例については、GitHub の Amazon Braket サンプルリポジトリを参照してください。

このセクションの内容:

- QPU に量子タスクを送信する
- 複数のプログラムを実行する
- 量子タスクはいつ実行されますか?
- 予約の使用
- エラー緩和手法

QPU に量子タスクを送信する

Amazon Braket は、量子タスクを実行できる複数のデバイスへのアクセスを提供します。量子タスクを個別に送信することも、量子タスクのバッチ処理を設定することもできます。

量子処理ユニット (QPU)

量子タスクはいつでも QPU に送信できますが、タスクは Amazon Braket コンソールの [デバイス] ページに表示される特定の可用性ウィンドウ内で実行されます。量子タスクの結果は、次のセクションで紹介する量子タスク ID を使用して取得できます。

- AQT IBEX-Q1 : `arn:aws:braket:eu-north-1::device/qpu/aqt/Ibex-Q1`
- IonQ Forte-1 : `arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1`

- IonQ Forte-Enterprise-1 : `arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1`
- IQM Garnet : `arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet`
- IQM Emerald : `arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald`
- QuEra Aquila : `arn:aws:braket:us-east-1::device/qpu/quera/Aquila`
- Rigetti Ankaa-3 : `arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3`

> **Note**
>
> QPU およびオンデマンドシミュレーターでは、CREATED 状態の量子タスクをキャンセルできます。QPU およびオンデマンドシミュレーターでは、QUEUED 状態の量子タスクをベストエフォートベースでキャンセルできます。QPU の QUEUED 状態の量子タスクは、QPU の可用性ウィンドウ中は正常にキャンセルされないことに注意してください。

このセクションの内容:

- AQT
- IonQ
- IQM
- Rigetti
- QuEra
- 例: QPU に量子タスクを送信する
- コンパイルされた回路を検査する

AQT

AQTの IBEX-Q1 QPU は、超高バキュームチェンバーに座る巨視的無線周波数トラップ内の $^{40}Ca^{+}$ の電離に基づいています。デバイスは室内温度で動作し、2 つの 19 インチのデータセンター互換ラックに収まります。

高忠実度ゲートは、トラップの低い加熱率と、量子ビット回転のための直接光遷移の使用によって有効になります。量子ビット遷移は、相対周波数の安定性が非常に高い狭線幅レーザーによって駆動されます。量子ビットは、光学シェルフによる効率的な状態の準備と読み取りも備えています。All-to-

all接続は、イオンマニフェスト内の長距離クーロン相互作用によって実現されます。単一イオンのアドレス指定と読み取りは、高い数値の冪定レンズを使用することで実現されます。

AQT デバイスは、次の量子ゲートをサポートしています。

```
'ccnot', 'cnot', 'cphaseshift', 'cphaseshift00', 'cphaseshift01', 'cphaseshift10',
 'cswap', 'swap', 'iswap', 'pswap', 'ecr', 'cy', 'cz', 'xy', 'xx', 'yy', 'zz', 'h',
 'i', 'phaseshift', 'rx', 'ry', 'rz', 's', 'si', 't', 'ti', 'v', 'vi', 'x', 'y', 'z',
 'prx'
```

逐語的なコンパイルでは、AQTデバイスは次のネイティブゲートをサポートします。

```
'prx', 'xx', 'rz'
```

> **Note**
>
> AQT ネイティブゲートと Amazon Braket の同等のゲートを以下に示します。
>
> - AQT Mølmer-Sørensen (MS または RXX) ゲートは Braket の `'xx'` ゲートに対応します
> - AQT R ゲートは Braket の `'prx'` ゲートに対応します
> - `'rz'` ゲートの命名は同じです

IonQ

IonQ は、イオントラップ技術に基づくゲートベースの QPU を提供しています。IonQ's のトラップされたイオン QPU は、真空チャンバ内の微細加工表面電極トラップによって空間的に閉じ込められた 171Yb+ イオンのチェーン上に構築されています。

IonQ デバイスは、以下の量子ゲートをサポートしています。

```
'x', 'y', 'z', 'rx', 'ry', 'rz', 'h', 'cnot', 's', 'si', 't', 'ti', 'v', 'vi', 'xx',
 'yy', 'zz', 'swap'
```

逐語的なコンパイルでは、IonQ QPU は以下のネイティブゲートをサポートしています。

```
'gpi', 'gpi2', 'ms'
```

ネイティブ MS ゲートを使用するときに 2 つの位相パラメータのみを指定すると、完全にもつれさせる MS ゲートが実行されます。完全にもつれた MS ゲートは常にπ/2 の回転を実行します。別の角度を指定して、部分的にもつれさせる MS ゲートを実行するには、3 番目のパラメータを追加して目的の角度を指定します。詳細については、「braket.circuits.gate モジュール」を参照してください。

これらのネイティブゲートは、逐語的なコンパイルでのみ使用できます。逐語的なコンパイルに関する詳細は、「逐語的なコンパイル」を参照してください。

IQM

IQM 量子プロセッサは、超電導トランズモン型量子ビットに基づく汎用ゲートモデルデバイスです。IQM Garnet は 20 量子ビットデバイスであり、IQM Emerald は 54 量子ビットデバイスです。これらのデバイスはいずれも、正方形の格子トポロジを使用します。このトポロジは「結晶格子トポロジ」とも呼ばれます。

IQM デバイスは、以下の量子ゲートをサポートしています。

```
"ccnot", "cnot", "cphaseshift", "cphaseshift00", "cphaseshift01", "cphaseshift10",
 "cswap", "swap", "iswap", "pswap", "ecr", "cy", "cz", "xy", "xx", "yy", "zz", "h",
 "i", "phaseshift", "rx", "ry", "rz", "s", "si", "t", "ti", "v", "vi", "x", "y", "z"
```

逐語的なコンパイルでは、IQM デバイスは以下のネイティブゲートをサポートしています。

```
'cz', 'prx'
```

Rigetti

Rigetti 量子プロセッサは、全調整可能な超伝導 qubits に基づく汎用ゲートモデルマシンです。

- Ankaa-3 システムは、スケーラブルなマルチチップ技術を利用する 84 量子ビットのデバイスです。

Rigetti デバイスは、以下の量子ゲートをサポートしています。

```
'cz', 'xy', 'ccnot', 'cnot', 'cphaseshift', 'cphaseshift00', 'cphaseshift01',
 'cphaseshift10', 'cswap', 'h', 'i', 'iswap', 'phaseshift', 'pswap', 'rx', 'ry', 'rz',
 's', 'si', 'swap', 't', 'ti', 'x', 'y', 'z'
```

逐語的なコンパイルでは、Ankaa-3 は以下のネイティブゲートをサポートしています。

```
'rx', 'rz', 'iswap'
```

Rigetti 超伝導量子プロセッサは、「rx」ゲートを「±π/2」または「±π」の角度のみを使用して実行できます。

Rigetti デバイスは、Ankaa-3 システムの、下記タイプの事前定義フレームのセットをサポートしているため、パルスレベルの制御を使用できます。

```
`flux_tx`, `charge_tx`, `readout_rx`, `readout_tx`
```

デバイスには、回路あたり 20,000 ゲートの最大制限Ankaa-3があります。この制限を超える回路は、検証エラーで拒否されます。これは、引き上げることができない固定制限です。ゲート数は、コンパイルされた回路を指します。これは、元のコンパイルされていない回路のゲート数とは異なる場合があります。QPU に送信する前にコンパイルされたゲート数を推定するには、逐語的なコンパイルをローカルで使用するか、回路をネイティブゲートセット (`rx`、`rz`、) にトランスパイルしますiswap。

QuEra

QuEra は、Analog Hamiltonian Simulation (AHS) 量子タスクを実行できる中性原子ベースのデバイスを提供しています。これらの専用デバイスは、同時に相互作用する数百の量子ビットの時間依存量子力学を忠実に再現します。

これらのデバイスをプログラムするには、アナログハミルトニアンシミュレーションのパラダイムで、量子ビットレジスタのレイアウトと、操作用フィールド間の時間的および空間的な依存関係を指定します。Amazon Braket は、Python SDK の AHS モジュール、`braket.ahs` を使用してこのようなプログラムを構築するためのユーティリティを提供しています。

詳細については、アナログハミルトニアンシミュレーションのサンプルノートブックまたは「QuEra Aquila を使用してアナログプログラムを送信する」を参照してください。

例: QPU に量子タスクを送信する

Amazon Braket では、QPU デバイスで量子回路を実行することができます。次の例は、Rigetti または IonQ デバイスに量子タスクを送信する方法を示しています。

Rigetti Ankaa-3 デバイスを選択し、関連する接続グラフを確認します。

```
# import the QPU module
from braket.aws import AwsDevice
# choose the Rigetti device
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

# take a look at the device connectivity graph
device.properties.dict()['paradigm']['connectivity']
```

```
{'fullyConnected': False,
 'connectivityGraph': {'0': ['1', '7'],
  '1': ['0', '2', '8'],
  '2': ['1', '3', '9'],
  '3': ['2', '4', '10'],
  '4': ['3', '5', '11'],
  '5': ['4', '6', '12'],
  '6': ['5', '13'],
  '7': ['0', '8', '14'],
  '8': ['1', '7', '9', '15'],
  '9': ['2', '8', '10', '16'],
  '10': ['3', '9', '11', '17'],
  '11': ['4', '10', '12', '18'],
  '12': ['5', '11', '13', '19'],
  '13': ['6', '12', '20'],
  '14': ['7', '15', '21'],
  '15': ['8', '14', '22'],
  '16': ['9', '17', '23'],
  '17': ['10', '16', '18', '24'],
  '18': ['11', '17', '19', '25'],
  '19': ['12', '18', '20', '26'],
  '20': ['13', '19', '27'],
  '21': ['14', '22', '28'],
  '22': ['15', '21', '23', '29'],
  '23': ['16', '22', '24', '30'],
  '24': ['17', '23', '25', '31'],
  '25': ['18', '24', '26', '32'],
  '26': ['19', '25', '33'],
  '27': ['20', '34'],
  '28': ['21', '29', '35'],
  '29': ['22', '28', '30', '36'],
  '30': ['23', '29', '31', '37'],
  '31': ['24', '30', '32', '38'],
  '32': ['25', '31', '33', '39'],
  '33': ['26', '32', '34', '40'],
```

```
    '34': ['27', '33', '41'],
    '35': ['28', '36', '42'],
    '36': ['29', '35', '37', '43'],
    '37': ['30', '36', '38', '44'],
    '38': ['31', '37', '39', '45'],
    '39': ['32', '38', '40', '46'],
    '40': ['33', '39', '41', '47'],
    '41': ['34', '40', '48'],
    '42': ['35', '43', '49'],
    '43': ['36', '42', '44', '50'],
    '44': ['37', '43', '45', '51'],
    '45': ['38', '44', '46', '52'],
    '46': ['39', '45', '47', '53'],
    '47': ['40', '46', '48', '54'],
    '48': ['41', '47', '55'],
    '49': ['42', '56'],
    '50': ['43', '51', '57'],
    '51': ['44', '50', '52', '58'],
    '52': ['45', '51', '53', '59'],
    '53': ['46', '52', '54'],
    '54': ['47', '53', '55', '61'],
    '55': ['48', '54', '62'],
    '56': ['49', '57', '63'],
    '57': ['50', '56', '58', '64'],
    '58': ['51', '57', '59', '65'],
    '59': ['52', '58', '60', '66'],
    '60': ['59'],
    '61': ['54', '62', '68'],
    '62': ['55', '61', '69'],
    '63': ['56', '64', '70'],
    '64': ['57', '63', '65', '71'],
    '65': ['58', '64', '66', '72'],
    '66': ['59', '65', '67'],
    '67': ['66', '68'],
    '68': ['61', '67', '69', '75'],
    '69': ['62', '68', '76'],
    '70': ['63', '71', '77'],
    '71': ['64', '70', '72', '78'],
    '72': ['65', '71', '73', '79'],
    '73': ['72', '80'],
    '75': ['68', '76', '82'],
    '76': ['69', '75', '83'],
    '77': ['70', '78'],
    '78': ['71', '77', '79'],
```

```
  '79': ['72', '78', '80'],
  '80': ['73', '79', '81'],
  '81': ['80', '82'],
  '82': ['75', '81', '83'],
  '83': ['76', '82']}}
```

前述のディクショナリ connectivityGraph には、Rigetti デバイス内の各量子ビットの隣接する量子ビットが一覧表示されています。

IonQ Forte-Enterprise-1 デバイスを選択します。

IonQ Forte-Enterprise-1 デバイスについては、以下の例に示すように、connectivityGraph は空です。これは、デバイスがオールツーオール接続を提供しているためです。したがって、詳しい connectivityGraph は必要ありません。

```
# or choose the IonQ Forte-Enterprise-1 device
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1")

# take a look at the device connectivity graph
device.properties.dict()['paradigm']['connectivity']
```

```
{'fullyConnected': True, 'connectivityGraph': {...}}
```

次の例に示すように、デフォルトバケット以外の場所を指定するには、shots (デフォルト = 1000)、poll_timeout_seconds (デフォルト = 432000 = 5 日)、poll_interval_seconds (デフォルト = 1)、および結果を保存する S3 バケットの場所 (s3_location) を変更できます。

```
my_task = device.run(circ, s3_location = 'amazon-braket-my-folder', shots=100,
 poll_timeout_seconds = 100, poll_interval_seconds = 10)
```

IonQ および Rigetti デバイスは、提供された回路をそれぞれのネイティブゲートセットに自動的にコンパイルし、抽象 qubit インデックスをそれぞれの QPU の物理 qubits にマッピングします。

> **Note**
>
> QPU デバイスの容量は限られています。容量に達すると、待機時間が長くなることが予想されます。

Amazon Braket は特定の可用性ウィンドウ内で QPU 量子タスクを実行できますが、対応するすべてのデータとメタデータが適切な S3 バケットに確実に保存されるため、量子タスクはいつでも送信できます。次のセクションに示すように、量子タスクをリカバリするには、AwsQuantumTask と一意の量子タスク ID を使用します。

コンパイルされた回路を検査する

量子回路を量子処理ユニット (QPU) などのハードウェアデバイスで実行する必要がある場合は、まず回路を、デバイスが理解して処理できる許容可能な形式にコンパイルする必要があります。例えば、高レベルの量子回路をターゲットの QPU ハードウェアでサポートされている特定のネイティブゲートにトランスパイルします。量子回路の実際のコンパイル出力を検査することは、デバッグと最適化の目的に非常に役立ちます。この検査の知見は、量子アプリケーションのパフォーマンスと効率を改善する上で、潜在的な問題やボトルネックを特定することで潜在的な改善機会を発見するのに役立ちます。Rigetti および IQM 量子コンピューティングデバイスの両方について、以下のコードを使用して、量子回路のコンパイル出力を表示および分析できます。

```
task = AwsQuantumTask(arn=task_id, aws_session=session)
# After the task has finished running
task_result = task.result()
compiled_circuit = task_result.get_compiled_circuit()
```

> **Note**
>
> 現在、IonQ デバイスの回路のコンパイル出力の表示はサポートされていません。

複数のプログラムを実行する

Amazon Braket は、複数の量子プログラムを効率的に実行するための 2 つのアプローチとして、プログラムセットと量子タスクバッチ処理を提供しています。

プログラムセットは、複数のプログラムでワークロードを実行する場合に推奨される方法であり、複数のプログラムを単一の Amazon Braket 量子タスクにパッケージ化できます。プログラムセットは、特にプログラム実行数が 100 に近づいた場合に、プログラムを個別に送信するよりもパフォーマンスの向上とコストの削減を実現します。

現在、プログラムセットは IQM および Rigetti デバイスでサポートされています。プログラムセットを QPU に送信する前に、まず Amazon Braket Local Simulator でテストすることをお勧めします。デバイスがプログラムセットをサポートしているかどうかを確認するには、Amazon Braket SDK を

使用してデバイスのプロパティを表示するか、Amazon Braket コンソールでデバイスページを表示します。

次の例は、プログラムセットの実行方法を示しています。

```
from math import pi
from braket.devices import LocalSimulator
from braket.program_sets import ProgramSet
from braket.circuits import Circuit

program_set = ProgramSet([
    Circuit().h(0).cnot(0,1),
    Circuit().rx(0, pi/4).ry(1, pi/8).cnot(1,0),
    Circuit().t(0).t(1).cz(0,1).s(0).cz(1,2).s(1).s(2),
])

device = LocalSimulator()
result = device.run(program_set, shots=300).result()
print(result[0][0].counts)  # The result of the first program in the program set
```

プログラムセットのさまざまな構築方法 (例えば、1 つのプログラムで多数のオブザーバブルまたはパラメータからプログラムセットを構築する方法) と、プログラムセットの結果を取得する方法に関する詳細は、「Amazon Braket デベロッパーガイド」の「プログラムセット」セクションと、Braket サンプル Github リポジトリのプログラムセットのフォルダーを参照してください。

量子タスクバッチ処理は、すべての Amazon Braket デバイスで使用できます。バッチ処理は、複数の量子タスクを並行して処理できるため、オンデマンドシミュレーター (SV1、DM1 または TN1) で実行する量子タスクで特に便利です。バッチ処理を使用すると、量子タスクを並行して起動できます。例えば、回路が互いに独立している10 個の量子タスクを必要とする計算を行う場合は、タスクのバッチ処理を使用することをお勧めします。プログラムセットをサポートしていないデバイスで、複数のプログラムを含むワークロードを実行する場合は、量子タスクバッチ処理を使用します。

次の例は、量子タスクのバッチを実行する方法を示しています。

```
from braket.circuits import Circuit
from braket.devices import LocalSimulator

bell = Circuit().h(0).cnot(0, 1)
circuits = [bell for _ in range(5)]

device = LocalSimulator()
```

```
batch = device.run_batch(circuits, shots=100)
print(batch.results()[0].measurement_counts)  # The result of the first quantum task in
 the batch
```

バッチ処理の具体的な詳細については、GitHub の「Amazon Braket examples」を参照してください。

このセクションの内容:

- プログラムセットとコストについて
- 量子タスクのバッチ処理とコストについて
- 量子タスクのバッチ処理と PennyLane
- タスクバッチ処理とパラメータ化された回路

プログラムセットとコストについて

プログラムセットは、最大 100 個のプログラムまたはパラメータセットを 1 つの量子タスクにパッケージ化することで、複数の量子プログラムを効率的に実行します。プログラムセットでは、タスクごとに 1 つの料金を支払うとともに、すべてのプログラムのショット総数に基づいてショットごとに 1 つの料金を支払うだけであり、プログラムを個別に送信するよりもコストを大幅に削減できます。このアプローチは、多くのプログラムがあってプログラムあたりのショット数が少ないワークロードに特に有益です。プログラムセットは現在、IQM および Rigetti デバイス、および Amazon Braket Local Simulator でサポートされています。

詳細な実装手順、ベストプラクティス、コード例など詳細については、「プログラムセット」セクションを参照してください。

量子タスクのバッチ処理とコストについて

量子タスクのバッチ処理と請求コストに関して留意すべきいくつかの注意点を以下に挙げます。

- デフォルトでは、量子タスクのバッチ処理はすべてのタイムアウトまたは失敗した量子タスクを 3 回再試行します。
- SV1 の 34 qubitsなど、長時間実行される量子タスクのバッチでは、大きなコストが発生する可能性があります。量子タスクのバッチを開始する前に、「`run_batch`」の割り当て値を入念にチェックしてください。TN1 を `run_batch` と一緒に使用することはお勧めしません。
- TN1 では、失敗したリハーサルフェーズタスクのコストが発生する可能性があります (詳細については TN1 の説明をご覧ください)。自動再試行によりコストが増加する可能性があるため、TN1 を

使用する場合は、バッチ処理の「max_retries」の数を 0 に設定することをお勧めします (量子タスクバッチ処理のページの 186 行目を参照)。

量子タスクのバッチ処理と PennyLane

Amazon Braket で PennyLane を使用する際にバッチ処理を利用するには、Amazon Braket デバイスをインスタンス化する際に、次の例に示すように `parallel = True` を設定します。

```
import pennylane as qml

# Define the number of wires (qubits) you want to use
wires = 2  # For example, using 2 qubits

# Define your S3 bucket
my_bucket = "amazon-braket-s3-demo-bucket"
my_prefix = "pennylane-batch-output"
s3_folder = (my_bucket, my_prefix)

device = qml.device("braket.aws.qubit",
                    device_arn="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
                    wires=wires,
                    s3_destination_folder=s3_folder,
                    parallel=True)
```

PennyLane を使用したバッチ処理の詳細については、「量子回路の並列化最適化」を参照してください。

タスクバッチ処理とパラメータ化された回路

パラメータ化された回路を含む量子タスクバッチを送信する場合、バッチ内のすべての量子タスクに使用される `inputs` ディクショナリ、または入力ディクショナリの `list` を提供することができます。後者の場合、次の例に示すように、`i` 番目のディクショナリは `i` 番目のタスクとペアになります。

```
from braket.circuits import Circuit, FreeParameter, Observable
from braket.aws import AwsQuantumTaskBatch, AwsDevice

# Define your quantum device
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")

# Create the free parameters
```

```
alpha = FreeParameter('alpha')
beta = FreeParameter('beta')

# Create two circuits
circ_a = Circuit().rx(0, alpha).ry(1, alpha).cnot(0, 2).xx(0, 2, beta)
circ_a.variance(observable=Observable.Z(), target=0)

circ_b = Circuit().rx(0, alpha).rz(1, alpha).cnot(0, 2).zz(0, 2, beta)
circ_b.expectation(observable=Observable.Z(), target=2)

# Use the same inputs for both circuits in one batch
tasks = device.run_batch([circ_a, circ_b], inputs={'alpha': 0.1, 'beta': 0.2})

# Or provide each task its own set of inputs
inputs_list = [{'alpha': 0.3, 'beta': 0.1}, {'alpha': 0.1, 'beta': 0.4}]

tasks = device.run_batch([circ_a, circ_b], inputs=inputs_list)
```

また、単一のパラメータ回路の入力ディクショナリのリストを準備し、量子タスクバッチとして送信することもできます。リストに N 個の入力ディクショナリがある場合、バッチには N 個の量子タスクが含まれます。i 番目の量子タスクは、i 番目の入力ディクショナリで実行される回路に対応します。

```
from braket.circuits import Circuit, FreeParameter

# Create a parametric circuit
circ = Circuit().rx(0, FreeParameter('alpha'))

# Provide a list of inputs to execute with the circuit
inputs_list = [{'alpha': 0.1}, {'alpha': 0.2}, {'alpha': 0.3}]

tasks = device.run_batch(circ, inputs=inputs_list, shots=100)
```

量子タスクはいつ実行されますか?

回路を送信すると、Amazon Braket は指定したデバイスに送信します。量子処理ユニット (QPU) およびオンデマンドシミュレーターの量子タスクは、受信された順序でキューに入れられ、処理されます。量子タスクを送信後に処理するために必要な時間は、他の Amazon Braket のカスタマーから送信されたタスクの数と複雑さ、および選択した QPU の可用性によって変わります。

このセクションの内容:

- QPU の可用性ウィンドウとステータス
- キューの可視性
- E メールまたは SMS 通知の設定

QPU の可用性ウィンドウとステータス

QPU の可用性はデバイスによって異なります。

Amazon Braket コンソールの [デバイス] ページには、各デバイスの現在および今後の可用性ウィンドウとデバイスステータスが表示されます。さらに、各デバイスページには、量子タスクとハイブリッドジョブの個々のキューの深さも表示されます。

可用性ウィンドウに関係なく、カスタマーが利用できない場合、デバイスはオフラインと見なされます。例えば、スケジュールされたメンテナンス、アップグレード、または運用上の問題により、オフラインになる可能性があります。

キューの可視性

量子タスクまたはハイブリッドジョブを送信する前に、デバイスキューの深さを確認することで、既存の量子タスクまたはハイブリッドジョブの数を確認できます。

キューの深さ

Queue depth とは、特定のデバイスに対してキューに入れられた量子タスクとハイブリッドジョブの数のことです。デバイスの量子タスクとハイブリッドジョブキューの数を知るには、Braket Software Development Kit (SDK) または Amazon Braket Management Console を使用します。

1. タスクキューの深さは、通常の優先度で現在実行を待っている量子タスクの合計数のことです。
2. 優先度タスクキューの深さは、Amazon Braket Hybrid Jobs での実行を待機している送信済み量子タスクの合計数のことです。これらのタスクは、スタンドアロンタスクの前に実行されます。
3. ハイブリッドジョブキューの深さは、現在デバイスのキューに入っているハイブリッドジョブの合計数のことです。ハイブリッドジョブの一部として送信された Quantum tasks は、Priority Task Queue に集約され、高い優先度が付けられます。

Braket SDK を使用してキューの深さを表示する場合は、次のコードスニペットを変更して、量子タスクまたはハイブリッドジョブのキュー位置を取得できます。

```
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1")
```

```
# returns the number of quantum tasks queued on the device
print(device.queue_depth().quantum_tasks)
{<QueueType.NORMAL: 'Normal'>: '0', <QueueType.PRIORITY: 'Priority'>: '0'}


# returns the number of hybrid jobs queued on the device
print(device.queue_depth().jobs)
'3'
```

QPU に量子タスクまたはハイブリッドジョブを送信すると、ワークロードが QUEUED 状態になる可能性があります。Amazon Braket は、量子タスクとハイブリッドジョブのキュー位置をカスタマーに表示します。

キュー位置

Queue position とは、量子タスクまたはハイブリッドジョブの、各デバイスキュー内での現在の位置のことです。量子タスクまたはハイブリッドジョブのキュー位置を取得するには、Braket Software Development Kit (SDK) または Amazon Braket Management Console を使用します。

Braket SDK を使用してキュー位置を表示したいカスタマーは、次のコードスニペットを変更して、量子タスクまたはハイブリッドジョブのキュー位置を取得できます。

```
# choose the device to run your circuit
device = AwsDevice("arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet")

#execute the circuit
task = device.run(bell, s3_folder, shots=100)

# retrieve the queue position information
print(task.queue_position().queue_position)

# Returns the number of Quantum Tasks queued ahead of you
'2'


from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    "arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet",
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
```

```
    wait_until_complete=False
)

# retrieve the queue position information
print(job.queue_position().queue_position)
'3' # returns the number of hybrid jobs queued ahead of you
```

E メールまたは SMS 通知の設定

Amazon Braket は、QPU の可用性が変更されたとき、または量子タスクの状態が変化したときに Amazon EventBridge にイベントを送信します。次の手順に従って、デバイスおよび量子タスクのステータス変更通知をEメールまたは SMS メッセージで受信します。

1. Amazon SNS トピックとサブスクリプションを E メールまたは SMS に作成します。メールまたは SMS の可用性は、リージョンによって異なります。詳細については、「Amazon SNS の開始方法」と「SMS メッセージの送信」を参照してください。
2. EventBridge で SNS トピックへの通知をトリガーするルールを作成します。詳細については「Amazon EventBridge を使用した Amazon Braket のモニタリング」を参照してください。

SNS 通知を設定する (オプション)

Amazon Simple Notification Service (SNS) を通じて通知を設定し、Amazon Braket の量子タスクが完了したときにアラートを受け取ることができます。アクティブな通知は、大きな量子タスクを送信する場合や、デバイスの可用性ウィンドウの外で量子タスクを送信する場合など、長い待ち時間が予想される場合に便利です。量子タスクが完了するのを待ちたくない場合は、SNS 通知を設定してください。

Amazon Braket ノートブックでは、セットアップ手順を順を追って説明します。詳細については、「GitHub での Amazon Braket の例」、特に「通知を設定するためのサンプルノートブック」を参照してください。

予約の使用

予約により、選択した量子デバイスへの排他的なアクセスが可能になります。都合の良いときでの予約をスケジュールできるため、ワークロードでの実行の開始と終了を正確に把握できます。予約は、すべての Braket デバイスに対して 1 時間単位で利用でき、最大 48 時間前に追加料金なしでキャンセルできます。Braket Direct Reservation ARN を使用するか、予約中にワークロードを送信して、今後の予約の量子タスクとハイブリッドジョブを事前にキューに入れることをお勧めします。

専用デバイスアクセスのコストは、量子処理ユニット (QPU) で実行する量子タスクとハイブリッドジョブの数に関係なく、予約の期間に基づきます。予約可能な量子コンピュータの最新リストは、料金ページまたは Amazon Braket マネジメントコンソールで確認できます。

> **Note**
>
> 予約には、ゲートショットの制限はありません。さらに、 IonQ デバイスの場合、エラー緩和タスクの最小ショット数は 500 に削減されます (オンデマンドの場合は 2500 に対して)。

予約を使用するタイミング

予約アクセスを活用すると、量子ワークロードの実行の開始と終了を正確に把握する利便性と予測可能性が得られます。タスクやハイブリッドジョブをオンデマンドで送信する場合との違いは、他のカスタマータスクが含まれるキューにおけるような待機を行う必要がないことです。予約した期間全体において、お客様はデバイスへの排他的なアクセス権を持っているため、お客様のワークロードのみがデバイスで実行されます。

オンデマンドアクセスを使用して研究の設計とプロトタイプ作成フェーズを行うことで、迅速かつコスト効率の高い、アルゴリズムの反復を使用可能にすることをお勧めします。最終的な実験結果を作成する準備ができたら、プロジェクトまたは公開の期限を確実に守るために、都合の良いときにデバイス予約をスケジュールすることを検討してください。また、量子コンピュータでライブデモやワークショップを実行するなど、特定の時間帯にタスクを実行する場合にも、予約を使用することをお勧めします。

このセクションの内容:

- 予約を作成する方法
- 予約期間での量子タスクの実行
- 予約期間中にハイブリッドジョブを実行する
- 予約期間が終了するとどうなるか
- 既存の予約をキャンセルまたは再スケジュールする

予約を作成する方法

予約を作成する方法については、以下の手順に従って Braket チームにお問い合わせください。

1. Amazon Braket コンソールを開きます。
2. 左ペインで [Braket Direct] を選択し、[予約] セクションで [デバイスを予約] を選択します。
3. [デバイス] で、予約するデバイスを選択します。
4. [名前] や [E メール] などの連絡先情報に値を入力します。E メールのアドレスとしては必ず、定期的にチェックする有効なものを指定してください。
5. [ワークロードについて教えてください] に、予約を使用して実行するワークロードに関する詳細を入力します。例えば、希望の予約期間、関連する制約、希望のスケジュールなどです。

フォームを送信すると、Braket チームから次のステップを記載した E メールが届きます。予約がチーム内で確認されると、予約 ARN が E メールで送信されます。予約 ARN を使用して予約タスクを作成するには、 が必要です。予約 ARN なしで作成されたタスクは、通常のオンデマンドキューに送信され、予約中は実行されません。

> **Note**
>
> お客様が予約 ARN を受け取って初めて、予約の確定となります。

予約は最低 1 時間単位で指定できます。デバイスによっては、追加の予約期間制約 (最小予約期間と最大予約期間を含む) があります。Braket チームは、予約を確認する前にすべての関連情報をお客様と共有します。

Braket チームは、Braket の専門家との 30 分間のセッションを手配するために、E メールでご連絡します。

予約期間での量子タスクの実行

[予約を作成] により有効な予約 ARN を取得したら、予約期間に実行する量子タスクを作成できます。予約 ARN で送信された量子タスクとハイブリッドジョブは、デバイスキューに表示されません。予約開始時刻より前に送信されたタスクは、予約が開始されるまで QUEUED状態のままになります。

> **Note**
>
> 予約は AWS アカウントとデバイスによって異なります。予約を作成した AWS アカウントのみが予約 ARN を使用できます。
> 予約中に、予約タスクと通常のタスクの両方を作成できます。作成された Braket 量子タスクが予約に関連付けられていることを確認するには、Braket コンソールの量子タスクのペー

ジの「予約 ARN」フィールドを確認するか、 SDK を使用してタスクメタデータ内の同じフィールドをクエリします。このページの残りの部分では、どのタスクをリザベーションに関連付けるかを指定する方法について説明します。

Braket、、CUDA-Q、PennyLane、 QiskitPython SDKsなどの量子タスクを boto3 (Boto3 の使用) で直接作成できます。予約を使用するには、Amazon Braket Python SDK のバージョン v1.79.0 以降が必要です。最新の Braket SDK、Qiskit プロバイダー、PennyLane プラグインに更新するには、次のコードを使用します。

```
pip install --upgrade amazon-braket-sdk amazon-braket-pennylane-plugin qiskit-braket-
provider
```

`DirectReservation` コンテキストマネージャーを使用してタスクを実行する

スケジュールされた予約期間内にタスクを実行するには、DirectReservation コンテキストマネージャーを使用することをお勧めします。ターゲットデバイスと予約 ARN を指定することで、コンテキストマネージャーにより、Python with ステートメント内で作成されたすべてのタスクがデバイスへの排他的アクセスによって実行されるようになります。

まず、量子回路とデバイスを定義します。次に、予約コンテキストを使用してタスクを実行します。ワークロード全体が **with**ブロック内で実行されていることを確認します。 **with**ブロックの範囲外で実行されたものは予約に関連付けられません。

```
from braket.aws import AwsDevice, DirectReservation
from braket.circuits import Circuit
from braket.devices import Devices

bell = Circuit().h(0).cnot(0, 1)
device = AwsDevice(Devices.IonQ.ForteEnterprise1)

# run the circuit in a reservation
with DirectReservation(device, reservation_arn="<my_reservation_arn>"):
    task = device.run(bell, shots=100)
```

量子タスクの作成中にDirectReservationコンテキストがアクティブであれば、CUDA-Q、PennyLane、 Qiskitプラグインを使用して予約に量子タスクを作成できます。例えば、Qiskit-Braket プロバイダーでは、次のようにタスクを実行できます。

```
from braket.devices import Devices
```

```
from braket.aws import DirectReservation
from qiskit import QuantumCircuit
from qiskit_braket_provider import BraketProvider


qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)

qpu = BraketProvider().get_backend("Forte Enterprise 1")

# run the circuit in a reservation
with DirectReservation(Devices.IonQ.ForteEnterprise1,
 reservation_arn="<my_reservation_arn>"):
    qpu_task = qpu.run(qc, shots=10)
```

また、次のコードは、Braket-PennyLane プラグインを使用して予約期間中に回路を実行するものです。

```
from braket.devices import Devices
from braket.aws import DirectReservation
import pennylane as qml


dev = qml.device("braket.aws.qubit", device_arn=Devices.IonQ.ForteEnterprise1.value,
 wires=2, shots=10)

@qml.qnode(dev)
def bell_state():
    qml.Hadamard(wires=0)
    qml.CNOT(wires=[0, 1])
    return qml.probs(wires=[0, 1])

# run the circuit in a reservation
with DirectReservation(Devices.IonQ.ForteEnterprise1,
 reservation_arn="<my_reservation_arn>"):
    probs = bell_state()
```

予約コンテキストを手動で設定する

または、予約コンテキストを手動で設定するには、次のコードを使用します。

```
# set reservation context
```

```
reservation_context = DirectReservation(device,
 reservation_arn="<my_reservation_arn>").start()

# run circuit during reservation
task = device.run(bell, shots=100)
```

これは、コンテキストを最初のセルで実行でき、後続のすべてのタスクが予約期間に実行される Jupyter Notebook に最適です。

> **Note**
>
> `.start()` 呼び出しを含むセルは 1 回だけ実行する必要があります。

オンデマンドモードに戻すには: Jupyter Notebook を再起動するか以下を呼び出すことで、コンテキストをオンデマンドモードに戻すことができます。

```
reservation_context.stop()  # unset reservation context
```

> **Note**
>
> 予約には、事前に定義された開始時刻と終了時刻があります (「予約の作成」を参照)。`reservation_context.start()` および `reservation_context.stop()` メソッドは予約を開始したり終了したりしません。代わりに、コンテキストがアクティブな間、作成した量子タスクは予約に関連付けられ、スケジュールされた予約中にのみ実行されます。予約コンテキストは、スケジュールされた予約時間には影響しません。

タスクの作成時に予約 ARN を明示的に渡す

予約期間におけるタスクを作成するもう 1 つの方法は、`device.run()` を呼び出すときに予約 ARN を明示的に渡すことです。

```
task = device.run(bell, shots=100, reservation_arn="<my_reservation_arn>")
```

このメソッドは、量子タスクを予約 ARN に直接関連付け、予約期間中に実行するようにします。この方法では、予約中に実行する予定の各タスクに予約 ARN を追加してください。ただし、 Qiskit や などのサードパーティーライブラリを使用する場合PennyLane、送信されたタスクが正しい予約

ARN を使用していることを確認するのは難しい場合があります。このため、DirectReservation コンテキストマネージャーを使用することをお勧めします。

boto3 を直接使用する場合は、タスクの作成時に予約 ARN を関連付けとして渡します。

```
import boto3

braket_client = boto3.client("braket")


kwargs["associations"] = [
    {
        "arn": "<my_reservation_arn>",
        "type": "RESERVATION_TIME_WINDOW_ARN",
    }
]

response = braket_client.create_quantum_task(**kwargs)
```

予約期間中にハイブリッドジョブを実行する

ハイブリッドジョブとして実行する Python 関数を作成したら、`reservation_arn` キーワード引数を渡すことで、予約期間内にそのハイブリッドジョブを実行できます。このハイブリッドジョブ内のすべてのタスクは当該の予約 ARN を使用します。重要なのは、`reservation_arn` を使用するハイブリッドジョブは、予約期間が開始されて初めて古典コンピューティングを起動することです。

> **Note**
>
> 予約期間中に実行されるハイブリッドジョブは、予約されたデバイスでのみ、量子タスクを正常に実行します。別のオンデマンド Braket デバイスを使用しようとすると、エラーが発生します。同じハイブリッドジョブ内でオンデマンドシミュレーターと予約済みデバイスの両方でタスクを実行する必要がある場合は、代わりに `DirectReservation` を使用してください。

次のコードは、予約期間中にハイブリッドジョブを実行する方法を示しています。

```
from braket.aws import AwsDevice
from braket.devices import Devices
from braket.jobs import get_job_device_arn, hybrid_job
```

```
@hybrid_job(device=Devices.IonQ.ForteEnterprise1,
 reservation_arn="<my_reservation_arn>")
def example_hybrid_job():
    # declare AwsDevice within the hybrid job
    device = AwsDevice(get_job_device_arn())
    bell = Circuit().h(0).cnot(0, 1)

    task = device.run(bell, shots=10)
```

Python スクリプトを使用するハイブリッドジョブの場合 (デベロッパーガイドの「Creating your first Hybrid Job」セクションを参照)、ジョブの作成時に reservation_arn キーワード引数を渡すことで予約期間内に実行できます。

```
from braket.aws import AwsQuantumJob
from braket.devices import Devices

job = AwsQuantumJob.create(
    Devices.IonQ.ForteEnterprise1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    reservation_arn="<my_reservation_arn>"
)
```

予約期間が終了するとどうなるか

予約期間が終了すると、デバイスへの専用アクセスはできなくなります。この予約期間を使用してキューに入れられていた残りのワークロードは自動的にキャンセルされます。

> **Note**
>
> 予約期間の終了時に RUNNING 状態だったジョブはすべてキャンセルされます。ジョブを保存しておいて、都合の良いときに再起動するためのチェックポイントを使用することをお勧めします。

予約開始後や予約終了前など、進行中の予約期間は、延長できません。各予約期間がデバイスへの単独の専用アクセスを表しているためです。例えば、連続する 2 つの予約期間は別々の予約期間と見なされるため、最初の予約期間において保留中だったタスクは自動的にキャンセルされます。2 番目の予約期間で再開されることはありません。

> **Note**
>
> 予約は、 AWS アカウントの専用デバイスアクセスを表します。デバイスがアイドル状態であっても、他のカスタマーはそのデバイスを使用できません。したがって、使用時間に関係なく、予約時間の長さに対して課金されます。

既存の予約をキャンセルまたは再スケジュールする

予約は、スケジュール済みの予約開始時刻の 48 時間前までにキャンセルできます。キャンセルするには、キャンセルリクエストとともに受け取った予約確認 E メールに返信します。

スケジュールを変更するには、既存の予約をキャンセルしてから、新しい予約を作成する必要があります。

エラー緩和手法

量子エラーの緩和は、量子コンピュータのエラーの影響を減らすことを目的とした一連の手法です。

量子デバイスは、実行される計算の品質を低下させる環境ノイズにさらされています。耐障害性量子コンピューティングはこの問題の解決策を約束していますが、現在の量子デバイスは量子ビットの数と比較的高いエラー率による限界があります。環境ノイズに短期的に対処するために、研究者は多くのノイズにさらされる量子計算の精度を向上させる方法を研究しています。このアプローチは量子エラー緩和とも呼ばれ、さまざまな手法を使用してノイズの多い測定データから最適なシグナルを抽出します。

このセクションの内容:

- IonQ デバイスのエラー緩和手法

IonQ デバイスのエラー緩和手法

エラー緩和とは、複数の物理回路を実行し、その測定値を組み合わせて結果を改善することです。

> **Note**
>
> IonQ の各デバイスの場合: オンデマンドモデルを使用するとき、100 万ゲートショットの制限があり、エラー緩和タスクには最低 2,500 ショットを実行する必要があります。直接予約

の場合、ゲートショットに制限はなく、エラー緩和タスクでは最低 500 ショットを実行する必要があります。

デバイアス処理

IonQ デバイスは、デバイアス処理と呼ばれるエラー緩和手法を備えています。

デバイアス処理は、量子ビットの複数の順列または複数のゲート分解で動作する、複数のバリアントに回路をマッピングします。これにより、測定結果にバイアスをかける可能性のある回路について、そのさまざまな実装を使用することで、ゲートの過回転や単一の故障した量子ビットなどの系統誤差の影響が緩和されます。これには、複数の量子ビットとゲートを較正するための余分なオーバーヘッドが掛かるという代償を伴います。

デバイアス処理の詳細については、「Enhancing quantum computer performance through symmetrization」を参照してください。

> **Note**
>
> デバイアス処理を使用するには、少なくとも 2500 ショットが必要です。

IonQ デバイスでデバイアス処理を行って量子タスクを実行するには、次のコードを使用します。

```
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.error_mitigation import Debias

# choose an IonQ device
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1")
circuit = Circuit().h(0).cnot(0, 1)

task = device.run(circuit, shots=2500, device_parameters={"errorMitigation": Debias()})

result = task.result()
print(result.measurement_counts)
>>> {"00": 1245, "01": 5, "10": 10 "11": 1240} # result from debiasing
```

量子タスクが完了すると、量子タスクの測定確率と何らかの結果タイプを確認できます。すべてのバリアントの測定確率および測定カウントは 1 つの分布に集約されます。期待値など、回路で指定された結果タイプは、総測定カウントを使用して計算されます。

シャープニング

また、シャープニングと呼ばれる別の後処理戦略で計算された測定確率にアクセスすることもできます。シャープニングとは、各バリアントの結果を比較し、一貫性のないショットを破棄して、バリアント間で最も頻度の高い測定結果を優先することです。詳細については、「Enhancing quantum computer performance through symmetrization」を参照してください。

重要なのは、シャープニングは出力分布の形式がスパースで、確率の高い状態がほとんどなく、確率が 0 の状態が多いことを前提としていることです。この前提が有効でない場合、確率分布がゆがむ可能性があります。

シャープニングされた分布から読み取れる確率に、Braket Python SDK の `GateModelTaskResult` の [`additional_metadata`] フィールドからアクセスできます。シャープニングは測定カウントを返さず、代わりに再正規化された確率分布を返すことに注意してください。次のコードスニペットは、シャープニング後の分布にアクセスする方法を示しています。

```
print(result.additional_metadata.ionqMetadata.sharpenedProbabilities)
>>> {"00": 0.51, "11": 0.549} # sharpened probabilities
```

Amazon Braket Hybrid Jobs の使用方法

Amazon Braket Hybrid Jobs は、古典的な AWS リソースと量子処理ユニット (QPUs) の両方を必要とするハイブリッド量子古典アルゴリズムを実行する方法を提供します。Hybrid Jobs は、リクエストされた古典的リソースを起動し、アルゴリズムを実行し、完了後にインスタンスを解放するように設計されているため、使用量に対してのみ料金が発生します。

Hybrid Jobs は、古典コンピューティングリソースと量子コンピューティングリソースの両方を使用する、長時間実行される反復アルゴリズムに最適です。Hybrid Jobs を使用する場合、実行するアルゴリズムを送信すると、アルゴリズムは、Braket により、スケーラブルでコンテナ化された環境で実行されます。アルゴリズムが完了したら、結果を取得できます。

さらに、ハイブリッドジョブから作成された量子タスクは、ターゲットとなる QPU デバイスへのキューイングにおいて優先順位が高くなるという利点があります。この優先順位付けにより、送信した量子コンピューティングが処理され、キューで待機している他のタスクよりも前に実行されます。これは、1 つの量子タスクの結果が以前の量子タスクの結果に依存する反復ハイブリッドアルゴリズムの場合に特に利点があります。このようなアルゴリズムの例としては、量子近似最適化アルゴリズム (QAOA)、変分量子固有値ソルバー、量子機械学習などがあります。また、アルゴリズムの進行状況をほぼリアルタイムでモニタリングできるため、コスト、予算、および、トレーニング損失やトレーニング期待値などのカスタムメトリクスを追跡できます。

Braket のハイブリッドジョブには、以下を使用してアクセスできます。

- Amazon Braket Python SDK。
- Amazon Braket コンソール。
- Amazon Braket API。

このセクションの内容:

- Amazon Braket Hybrid Jobs を使用すべき場合
- Amazon Braket Hybrid Jobs でハイブリッドジョブを実行する
- ハイブリッドジョブの主要なコンセプト
- 前提条件
- ハイブリッドジョブの作成
- ハイブリッドジョブのキャンセル

- ハイブリッドジョブのカスタマイズ
- PennyLane と Amazon Braket の併用
- CUDA-Q と Amazon Braket の併用方法

Amazon Braket Hybrid Jobs を使用すべき場合

Amazon Braket Hybrid Jobs を使用すると、古典的なコンピューティングリソースと量子コンピューティングデバイスを組み合わせて、今日の量子系のパフォーマンスを最適化する、変分量子固有ソルバー (VQE) や量子近似最適化アルゴリズム (QAOA) などのハイブリッド量子古典アルゴリズムを実行できます。Amazon Braket Hybrid Jobs には、主に次の 3 つの利点があります。

1. パフォーマンス: Amazon Braket Hybrid Jobs は、お客様の環境からハイブリッドアルゴリズムを実行するよりも優れたパフォーマンスを提供します。ハイブリッドジョブは、実行中に、選択したターゲット QPU に優先的にアクセスできます。ハイブリッドジョブ内のタスクは、デバイスでキューに入っている他のタスクよりも先に実行されます。これにより、ハイブリッドアルゴリズムの実行時間が短くなり、予測しやすくなります。また、Amazon Braket Hybrid Jobs は、パラメトリックコンパイルもサポートしています。自由パラメータを使用して回路を送信できるため、Braket は回路を 1 回コンパイルするだけです。同じ回路に対する後続のパラメータ更新を再コンパイルする必要がないため、実行時間がはるかに短縮されます。
2. 利便性: Amazon Braket Hybrid Jobs は、コンピューティング環境のセットアップと管理を簡素化し、ハイブリッドアルゴリズムの実行中も実行し続けることができます。アルゴリズムスクリプトを提供し、実行する量子デバイス (量子処理装置またはシミュレーター) を選択するだけでよいのです。Amazon Braket は、ターゲットデバイスが利用可能になるまで待機し、古典リソースを起動し、構築済みのコンテナ環境でワークロードを実行し、Amazon Simple Storage Service (Amazon S3) に結果を返し、コンピューティングリソースを解放します。
3. メトリクス: Amazon Braket Hybrid Jobs は、実行中のアルゴリズムに関するオンザフライのインサイトを提供し、カスタマイズ可能なアルゴリズムメトリクスをほぼリアルタイムで Amazon CloudWatch と Amazon Braket コンソールに配信することで、アルゴリズムの進行状況を追跡できます。

Amazon Braket Hybrid Jobs でハイブリッドジョブを実行する

Amazon Braket Hybrid Jobs でハイブリッドジョブを実行するには、まずアルゴリズムを定義する必要があります。Amazon Braket Python SDK または PennyLane を使用して、アルゴリズムスクリプトと、オプションで他の依存関係ファイルを作成することで定義できます。他の (オープンソースま

たはプロプライエタリ) ライブラリを使用する場合は、それらのライブラリが含まれている Docker を使用することで、独自のカスタムコンテナイメージを定義できます。詳細については、「自分のコンテナを持参 (BYOC)」を参照してください。

いずれの場合も、次に Amazon Braket API を使用してジョブを作成します。ここで、アルゴリズムスクリプトまたはコンテナを指定し、ハイブリッドジョブが使用するターゲット量子デバイスを選択し、さまざまなオプション設定から選択します。これらのオプション設定で提供されているデフォルト値は、ほとんどのユースケースで機能します。ターゲットデバイスにハイブリッドジョブを実行させる場合、QPU、オンデマンドシミュレーター (SV1、DM1、TN1 など)、または古典的なハイブリッドジョブインスタンス自体のいずれかを選択できます。オンデマンドシミュレーターまたは QPU を選択する場合は、ハイブリッドジョブコンテナはリモートデバイスに API コールを行います。埋め込みシミュレーターを選択する場合は、シミュレーターがアルゴリズムスクリプトと同じコンテナに埋め込まれます。PennyLane の稲妻シミュレーターには、デフォルトの構築済みハイブリッドジョブコンテナが埋め込まれています。埋め込み PennyLane シミュレーターまたはカスタムシミュレーターを使用してコードを実行する場合は、インスタンスタイプ、および使用するインスタンスの数を指定できます。各選択肢にかかるコストについては、Amazon Braket の料金ページを参照してください。

ターゲットデバイスがオンデマンドシミュレーターまたは埋め込みシミュレーターの場合、Amazon Braket はハイブリッドジョブの実行をすぐに開始します。ハイブリッドジョブインスタンスがスピンアップされて (API コールでインスタンスタイプをカスタマイズできます) アルゴリズムが実行され、結果が Amazon S3 に書き込まれてリソースが解放されます。このリソースの解放により、使用した分に対してのみお支払いいただくだけで済むようになります。

量子処理ユニット (QPU) あたりの同時ハイブリッドジョブの合計数は制限されています。現在、一度に QPU で実行できるハイブリッドジョブは 1 つのみです。許可される制限を超えないよう、実行できるハイブリッドジョブの数を制御するために、キューが使用されます。ターゲットデバイスが QPU の場合、ハイブリッドジョブは選択した QPU のジョブキューに最初に入ります。Amazon Braket は、必要なハイブリッドジョブインスタンスを起動し、ハイブリッドジョブをデバイスで実行します。アルゴリズムの期間中、ハイブリッドジョブには優先アクセスがあります。つまり、ハイブリッドジョブの量子タスクが数分に 1 回 QPU に送信されている場合、ジョブの量子タスクはデバイスでキューに入れられた他の Braket 量子タスクよりも先に実行されます。ハイブリッドジョブが完了すると、リソースが解放されるため、使用した分に対してのみ料金が発生します。

> **Note**
>
> デバイスはリージョン別であり、ハイブリッドジョブはプライマリデバイス AWS リージョン と同じ で実行されます。

シミュレーターおよび QPU の両方のターゲットシナリオで、アルゴリズムの一部としてハミルトニアンのエネルギーなどのカスタムアルゴリズムメトリクスを定義するオプションがあります。これらのメトリクスは Amazon CloudWatch に自動的にレポートされ、そこからほぼリアルタイムに Amazon Braket コンソールに表示されます。

> **Note**
>
> GPU ベースのインスタンスを使用する場合は、Braket の埋め込みシミュレーターとともに使用できるいずれかの GPU ベースシミュレーター (例えば `lightning.gpu` など) を使用してください。いずれかの CPU ベース埋め込みシミュレーター (例えば `lightning.qubit` や `braket:default-simulator` など) を選択した場合、GPU が使用されないため、不要なコストが発生する可能性があります。

ハイブリッドジョブの主要なコンセプト

このセクションでは、Amazon Braket Python SDK によって指定される `AwsQuantumJob.create` 関数の主要なコンセプトと、コンテナファイル構造へのマッピングを説明します。

ハイブリッドジョブには、アルゴリズムスクリプト全体を構成する 1 つまたは複数のファイルに加えて、追加の入力と出力を含めることができます。ハイブリッドジョブが開始されると、Amazon

Braket はハイブリッドジョブ作成の一部として指定された入力を、アルゴリズムスクリプトを実行するコンテナにコピーします。ハイブリッドジョブが完了すると、アルゴリズムで定義されたすべての出力が、指定された Amazon S3 の場所にコピーされます。

> **Note**
>
> アルゴリズムメトリクスはリアルタイムで報告されますので、この出力手順に従わないでください。

Amazon Braket では、コンテナの入力と出力とのやりとりを簡素化するために、いくつかの環境変数とヘルパー関数も指定します。詳細については、「Amazon Braket SDK」の「braket.jobs package」を参照してください。

このセクションの内容:

- 入力
- アウトプット
- 環境変数
- ヘルパー関数

入力

入力データ: 入力データをハイブリッドアルゴリズムに提供するには、ディクショナリとして設定されている入力データファイルを `input_data` 引数に指定します。ユーザーは引数 `input_data` を SDK の `AwsQuantumJob.create` 関数内に定義します。この引数により、環境変数 `"AMZN_BRAKET_INPUT_DIR"` で指定された場所にあるコンテナファイルシステムに入力データがコピーされます。ハイブリッドアルゴリズムでの入力データの使用方法のいくつかの例については、「QAOA with Amazon Braket Hybrid Jobs and PennyLane」および「Quantum machine learning in Amazon Braket Hybrid Jobs」Jupyter Notebook を参照してください。

> **Note**
>
> 入力データが大きい (>1GB) 場合、ハイブリッドジョブが送信されるまでの待機時間が長くなります。これは、最初にローカル入力データが S3 バケットにアップロードされ、次に S3 パスがハイブリッドジョブリクエストに追加されて、最後にハイブリッドジョブリクエストが Braket サービスに送信されるためです。

ハイパーパラメータ: hyperparameters を入力として渡した場合は、環境変数 "AMZN_BRAKET_HP_FILE" に設定されます。

> **Note**
>
> ハイパーパラメータと入力データを作成し、この情報をハイブリッドジョブスクリプトに渡す方法の詳細については、「ハイパーパラメータの使用」セクションとこちらの github ページを参照してください。

チェックポイント: 新しいハイブリッドジョブでチェックポイントを使用する job-arn を指定するには、copy_checkpoints_from_job コマンドを使用します。このコマンドは、チェックポイントデータを新しいハイブリッドジョブの checkpoint_configs3Uri にコピーし、ジョブの実行中に、環境変数 AMZN_BRAKET_CHECKPOINT_DIR で指定されたパスで使用できるようにします。デフォルトは、None であり、別のハイブリッドジョブからのチェックポイントデータが新しいハイブリッドジョブでは使用されないことを意味します。

アウトプット

量子タスク: 量子タスクの結果は S3 の場所 s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/tasksに 保存されます。

ジョブの結果: アルゴリズムスクリプトが環境変数 "AMZN_BRAKET_JOB_RESULTS_DIR" で指定されたディレクトリに保存するものはすべて、output_data_config で指定された S3 の場所にコピーされます。この場所の値が指定されない場合のデフォルト値は、s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/<timestamp>/data です。SDK ヘルパー関数 **save_job_result** が用意されています。これをアルゴリズムスクリプトから呼び出すと、結果を簡単にディクショナリ形式で保存できます。

チェックポイント: チェックポイントを使用する場合は、環境変数 "AMZN_BRAKET_CHECKPOINT_DIR" で指定されたディレクトリに保存できます。代わりに、SDK ヘルパー関数 save_job_checkpoint を使用することもできます。

アルゴリズムメトリクス: Amazon CloudWatch に放出され、ハイブリッドジョブの実行中に Amazon Braket コンソールにリアルタイムで表示されるアルゴリズムスクリプトの一部が、アルゴリズムメトリクスの定義です。アルゴリズムメトリクスの使用方法の例については、「Use Amazon Braket Hybrid Jobs to run a QAOA algorithm」を参照してください。

ジョブ出力の保存の詳細については、Hybrid Jobs ドキュメントの「Save your results」を参照してください。

環境変数

Amazon Braket は、コンテナの入力と出力の操作を簡素化するために、いくつかの環境変数を指定しています。次のコードは、Braket が使用する環境変数の一覧です。

- AMZN_BRAKET_INPUT_DIR – 入力データディレクトリ opt/braket/input/data。
- AMZN_BRAKET_JOB_RESULTS_DIR – ジョブ結果を書き込む出力ディレクトリ opt/braket/model。
- AMZN_BRAKET_JOB_NAME – ジョブの名前。
- AMZN_BRAKET_CHECKPOINT_DIR – チェックポイントディレクトリ。
- AMZN_BRAKET_HP_FILE – ハイパーパラメータを格納するファイル。
- AMZN_BRAKET_DEVICE_ARN – デバイス ARN (AWS リソース名) 。
- AMZN_BRAKET_OUT_S3_BUCKET – CreateJob リクエストの OutputDataConfig で指定する出力 Amazon S3 バケット。
- AMZN_BRAKET_SCRIPT_ENTRY_POINT – CreateJob リクエストの ScriptModeConfig で指定するエントリポイント。
- AMZN_BRAKET_SCRIPT_COMPRESSION_TYPE – CreateJobリクエストの ScriptModeConfig で指定する圧縮タイプ。
- AMZN_BRAKET_SCRIPT_S3_URI – CreateJobリクエストの ScriptModeConfig で指定する、ユーザースクリプトを保管する Amazon S3 の場所。
- AMZN_BRAKET_TASK_RESULTS_S3_URI – SDK がジョブの量子タスクの結果をデフォルトで保存する Amazon S3 の場所。
- AMZN_BRAKET_JOB_RESULTS_S3_PATH – CreateJob リクエストの OutputDataConfig で指定する、ジョブ結果を保存する Amazon S3 の場所。
- AMZN_BRAKET_JOB_TOKEN – ジョブコンテナ内に作成される量子タスクのための、CreateQuantumTask の jobToken パラメータに渡される文字列。

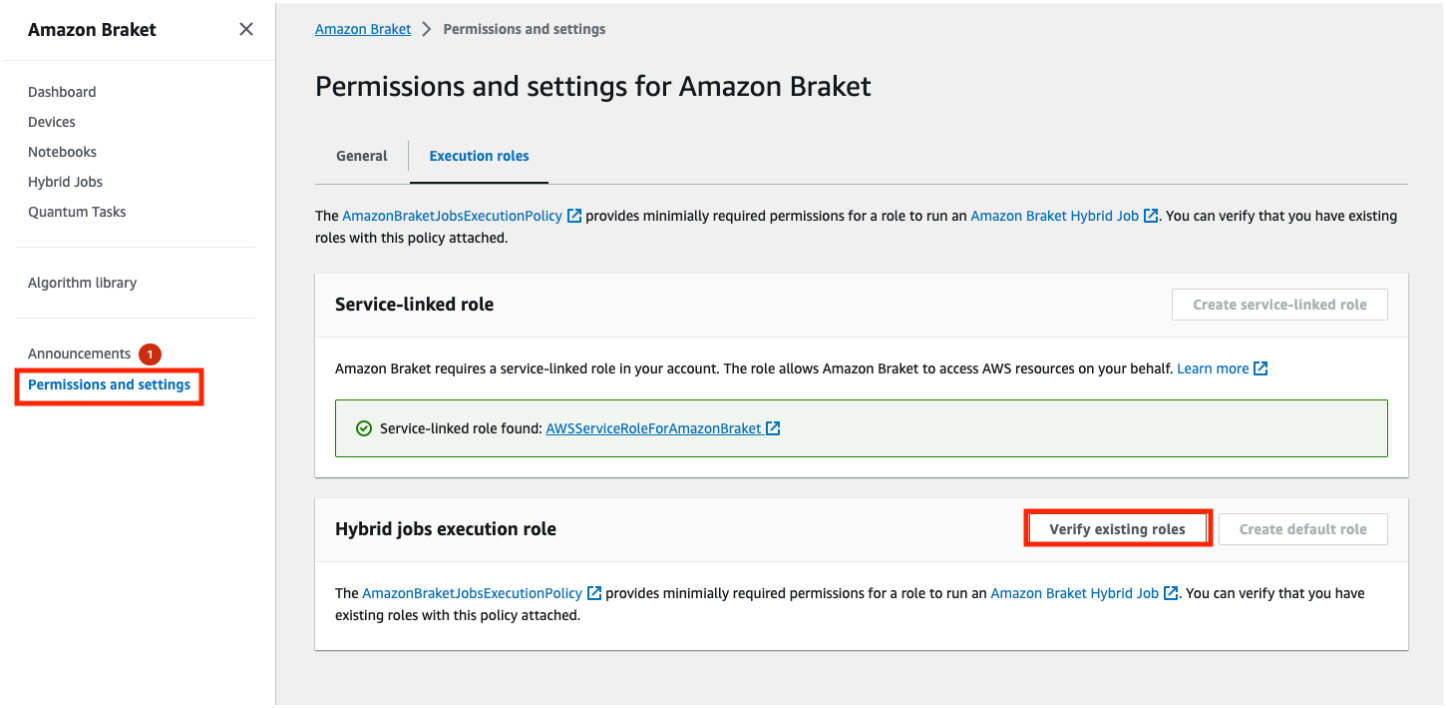
ヘルパー関数

Amazon Braket には、コンテナの入出力とのやりとりを簡素化するためのいくつかのヘルパー関数が指定されています。これらのヘルパー関数は、ハイブリッドジョブの実行に使用されるアルゴリズムスクリプト内から呼び出されます。次の例は、その使用方法を示しています。

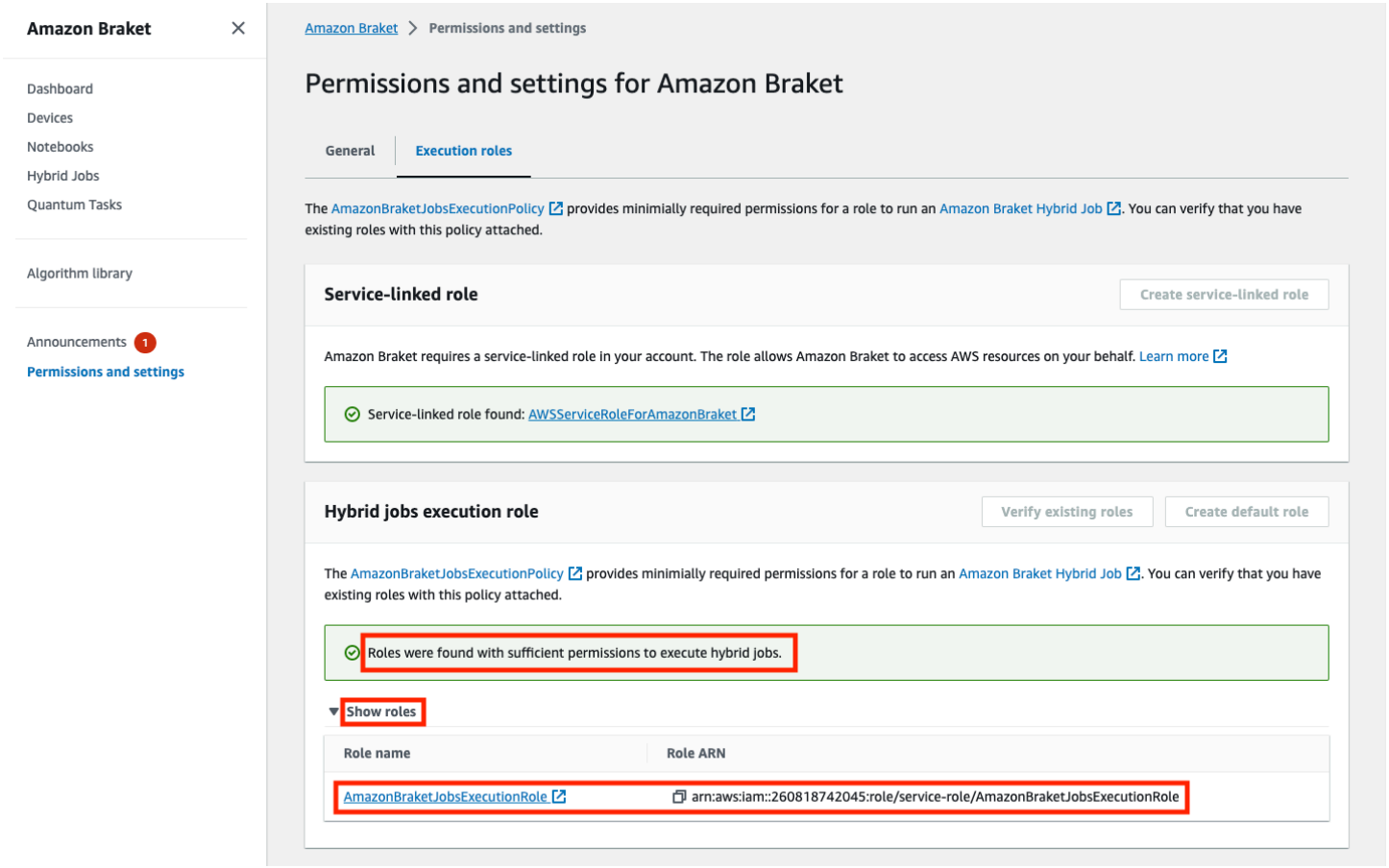
```
from braket.jobs import get_checkpoint_dir, get_hyperparameters, get_input_data_dir,
 get_job_device_arn, get_job_name, get_results_dir, save_job_result,
 save_job_checkpoint, load_job_checkpoint

get_checkpoint_dir() # Get the checkpoint directory
get_hyperparameters() # Get the hyperparameters as strings
get_input_data_dir() # Get the input data directory
get_job_device_arn() # Get the device specified by the hybrid job
get_job_name() # Get the name of the hybrid job.
get_results_dir() # Get the path to a results directory
save_job_result(result_data='data') # Save hybrid job results
save_job_checkpoint(checkpoint_data={'key': 'value'}) # Save a checkpoint
load_job_checkpoint() # Load a previously saved checkpoint
```

前提条件

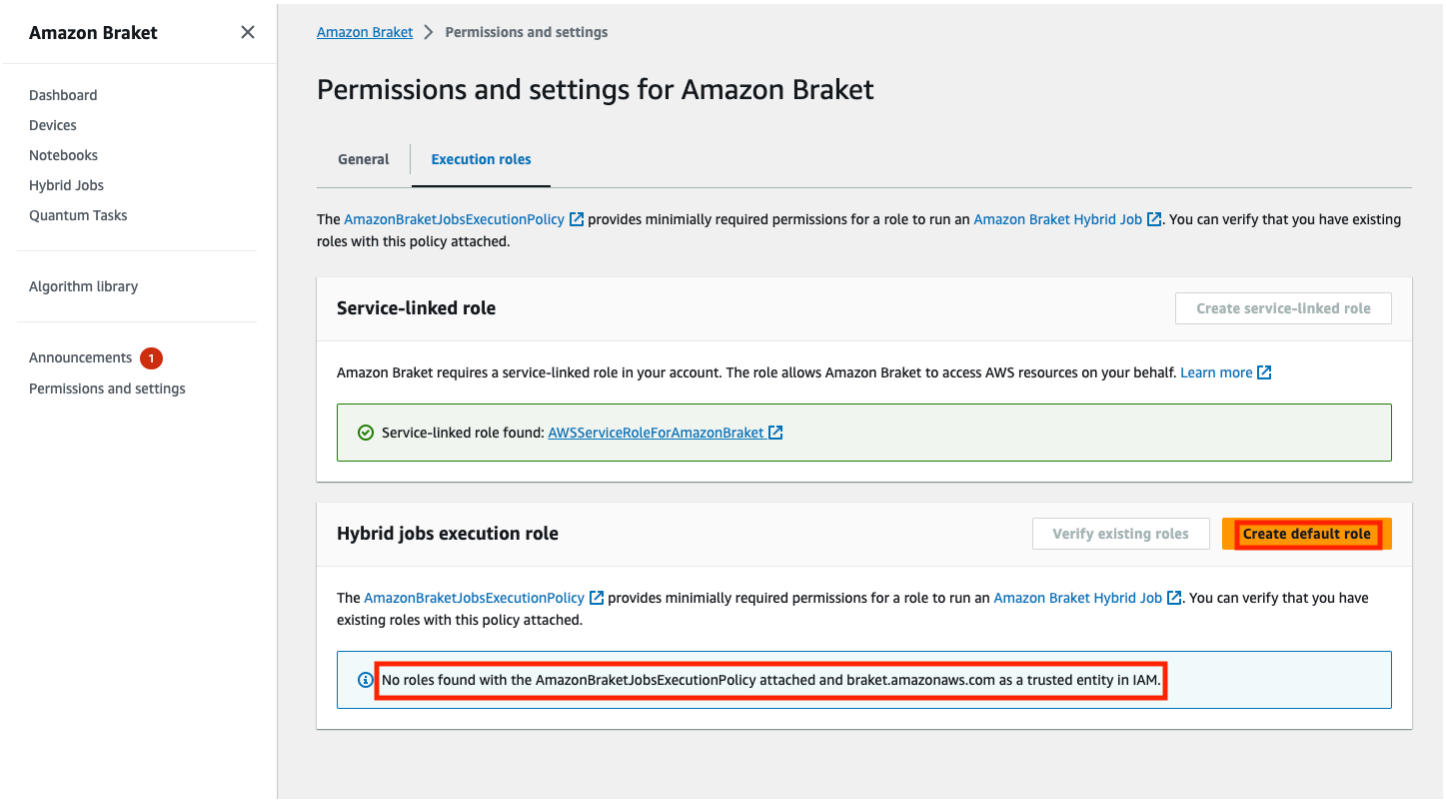
最初のハイブリッドジョブを実行する前に、このタスクを続行するための十分なアクセス許可があることを確認してください。適切なアクセス許可があることを確認するには、Braket コンソールの左側にあるメニューから [アクセス許可] を選択します。[Amazon Braket のアクセス許可管理] ページでは、既存のロールの 1 つでジョブの実行に十分な権限があるかどうかを確認したり、そのようなロールをまだ持っていない場合にジョブの実行に使用できるデフォルトロールの作成方法について説明しています。



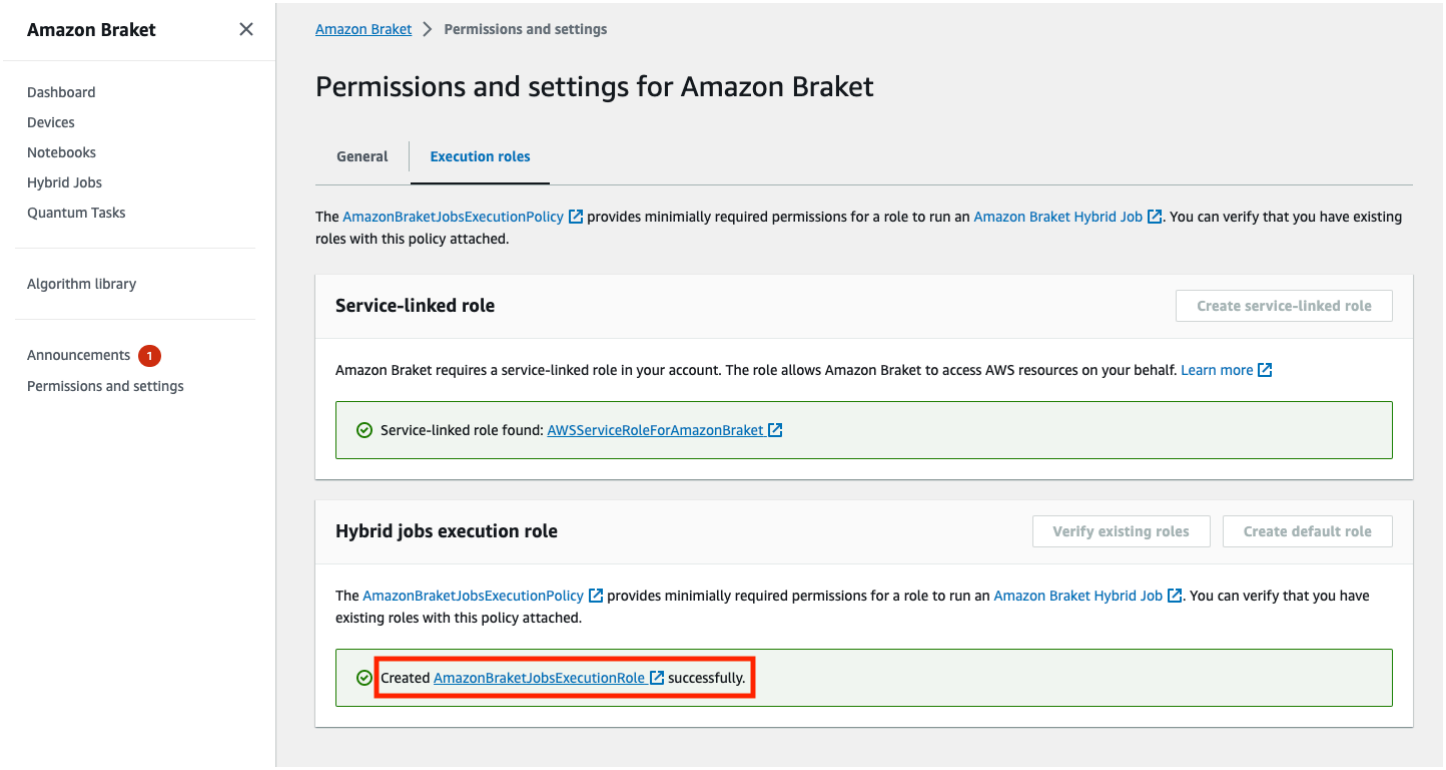
ハイブリッドジョブを実行するための十分なアクセス許可を持つロールがあることを確認するには、[既存のロールを検証] ボタンを選択します。選択すると、ロールが見つかったというメッセージが表示されます。ロールの名前とそのロール ARN を表示するには、[ロールを表示] ボタンを選択します。



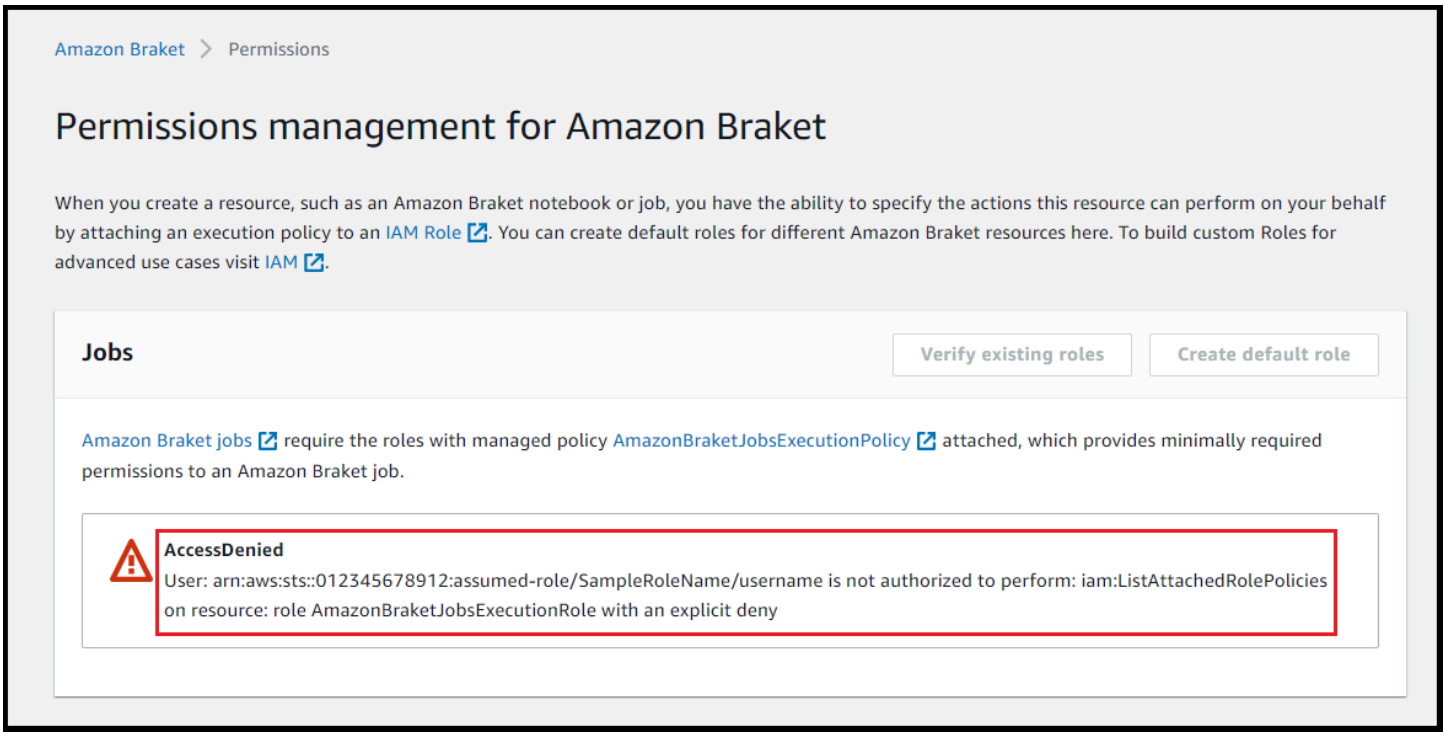
ハイブリッドジョブを実行するのに十分な権限を持つロールがない場合は、そのようなロールが見つからなかったというメッセージが表示されます。[デフォルトのロールの作成] ボタンを選択して、十分な権限を持つロールを取得します。



ロールが正常に作成された場合は、これを確認するメッセージが表示されます。



この問い合わせを行う権限がない場合、アクセスは拒否されます。この場合、内部 AWS 管理者にお問い合わせください。



ハイブリッドジョブの作成

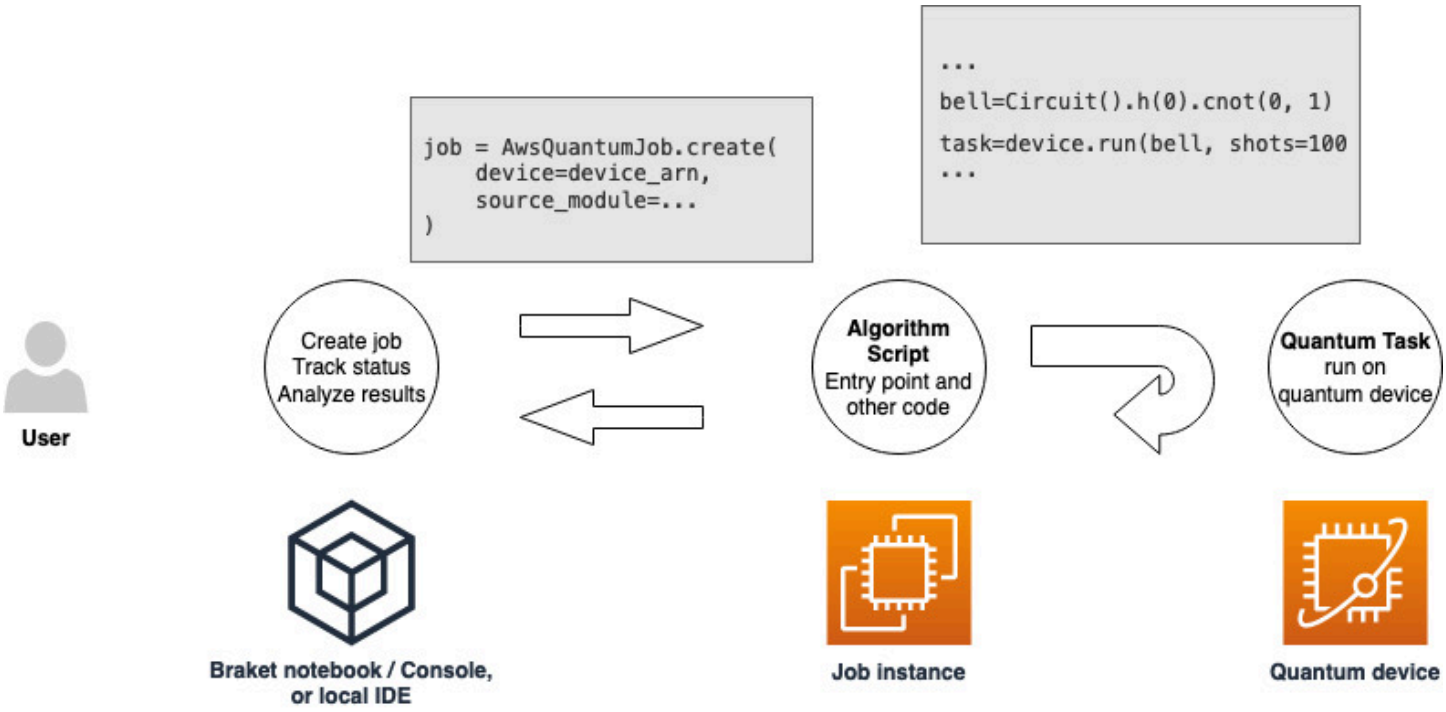
このセクションでは、Python スクリプトを使用してハイブリッドジョブを作成する方法を示します。または、ハイブリッドジョブを作成する別の方法として任意の統合開発環境 (IDE) や Braket ノートブックなどのローカル Python コードを使用する手順については、「ローカルコードをハイブリッドジョブとして実行する」を参照してください。

このセクションの内容:

- 作成して実行
- 結果をモニタリングする
- 結果を保存する
- チェックポイントの使用
- ローカルコードをハイブリッドジョブとして実行する
- ハイブリッドジョブでの API の使用方法
- ローカルモードでのハイブリッドジョブの作成およびデバッグ

作成して実行

ハイブリッドジョブを実行する権限を持つロールを取得したら、次に進む準備が整ったことになります。最初の Braket ハイブリッドジョブの重要部分は、アルゴリズムスクリプトです。実行するアルゴリズムを定義し、アルゴリズムの一部である古典的な論理と量子タスクが含まれています。アルゴリズムスクリプトに加えて、他の依存関係ファイルを指定することもできます。アルゴリズムスクリプトとその依存関係は、ソースモジュールと呼ばれます。エントリポイントは、ハイブリッドジョブの開始時にソースモジュールで実行される最初のファイルまたは関数を定義します。



まず、5 つのベル状態を作成し、対応する測定結果を出力するアルゴリズムスクリプトの基本的な例を考えてみましょう。

```
import os

from braket.aws import AwsDevice
from braket.circuits import Circuit


def start_here():

    print("Test job started!")

    # Use the device declared in the job script
    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])
```

```
bell = Circuit().h(0).cnot(0, 1)
for count in range(5):
    task = device.run(bell, shots=100)
    print(task.result().measurement_counts)

print("Test job completed!")
```

このファイルをalgorithm_script.py という名前で、Braket ノートブックまたはローカル環境の現在の作業ディレクトリに保存します。algorithm_script.py ファイルには計画されたエントリーポイントとして `start_here()` が含まれます。

次に、algorithm_script.py ファイルと同じディレクトリに Python ファイルまたは Python ノートブックを作成します。このスクリプトは、ハイブリッドジョブを開始し、必要なステータスや主要な結果の出力などの非同期処理を扱っています。このスクリプトでは、少なくともハイブリッドジョブスクリプトとプライマリデバイスを指定する必要があります。

> **Note**
>
> Braket ノートブックを作成する方法、または algorithm_script.py ファイルなどのファイルをノートブックと同じディレクトリにアップロードする方法の詳細については、「Run your first circuit using the Amazon Braket Python SDK」を参照してください。

この基本的な最初のケースでは、シミュレーターをターゲットにします。ターゲットとする量子デバイス、シミュレーター、または実際の量子処理装置 (QPU) のいずれのタイプであっても、次のスクリプトでは、`device` でハイブリッドジョブをスケジュールするために使用され、アルゴリズムスクリプトで環境変数 `AMZN_BRAKET_DEVICE_ARN` として使用できます。

> **Note**
>
> ハイブリッドジョブ AWS リージョン の で使用できるデバイスのみを使用できます。Amazon Braket SDK はこの AWS リージョンを自動的に選択します。例えば、us-east-1 のハイブリッドジョブでは、IonQ、SV1、DM1、および TN1 デバイスを使用できますが、Rigetti デバイスは使用できません。

シミュレーターの代わりに量子コンピュータを選択した場合、Braket は優先アクセスですべての量子タスクを実行するようにハイブリッドジョブをスケジュールします。

```
from braket.aws import AwsQuantumJob
from braket.devices import Devices

job = AwsQuantumJob.create(
    Devices.Amazon.SV1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    wait_until_complete=True
)
```

パラメータ wait_until_complete=True は、冗長モードを設定して、ジョブが実行中に実際のジョブからの出力を出力するようにします。以下の例のような出力が表示されます。

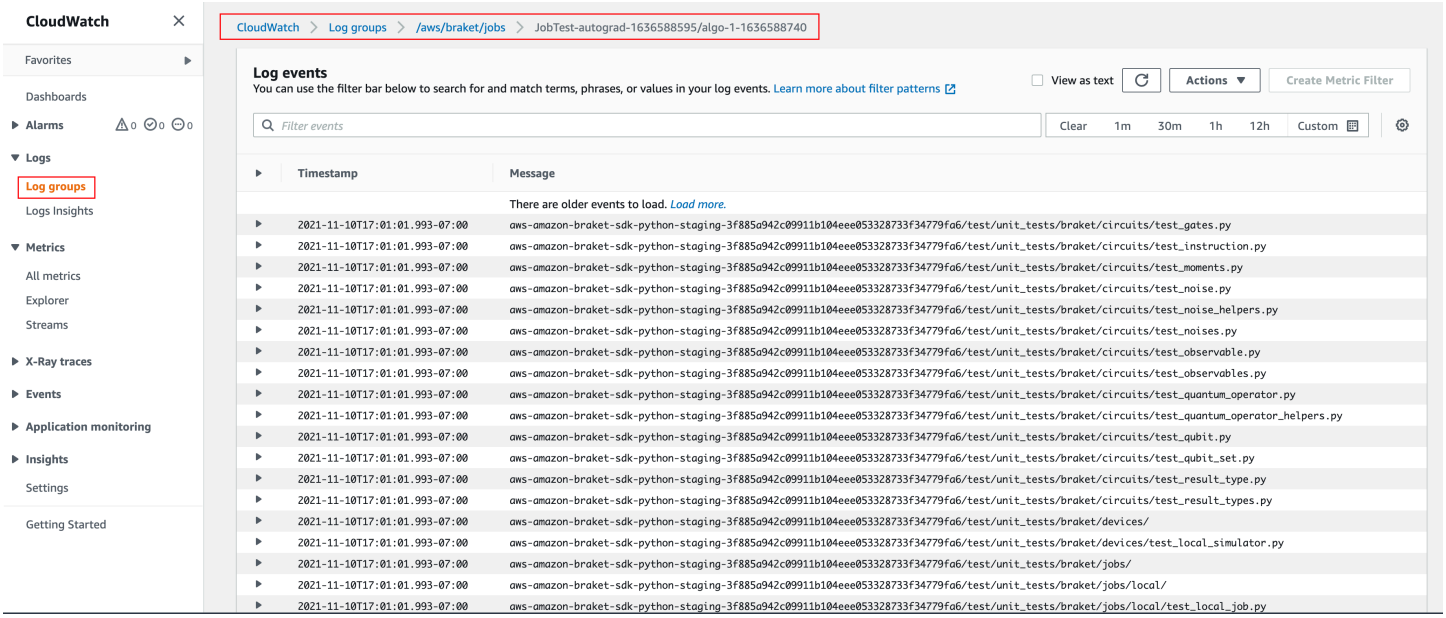
```
Initializing Braket Job: arn:aws:braket:us-west-2:111122223333:job/braket-job-
default-123456789012
Job queue position: 1
Job queue position: 1
Job queue position: 1
..............
.
.
.
Beginning Setup
Checking for Additional Requirements
Additional Requirements Check Finished
Running Code As Process
Test job started!
Counter({'00': 58, '11': 42})
Counter({'00': 55, '11': 45})
Counter({'11': 51, '00': 49})
Counter({'00': 56, '11': 44})
Counter({'11': 56, '00': 44})
Test job completed!
Code Run Finished
2025-09-24 23:13:40,962 sagemaker-training-toolkit INFO     Reporting training SUCCESS
```

> **Note**
>
> AwsQuantumJob.create メソッドを使用してカスタムメイドのモジュールを実行するには、その場所 (ローカルディレクトリまたはファイルへのパス、または tar.gz ファイルの S3 URI)

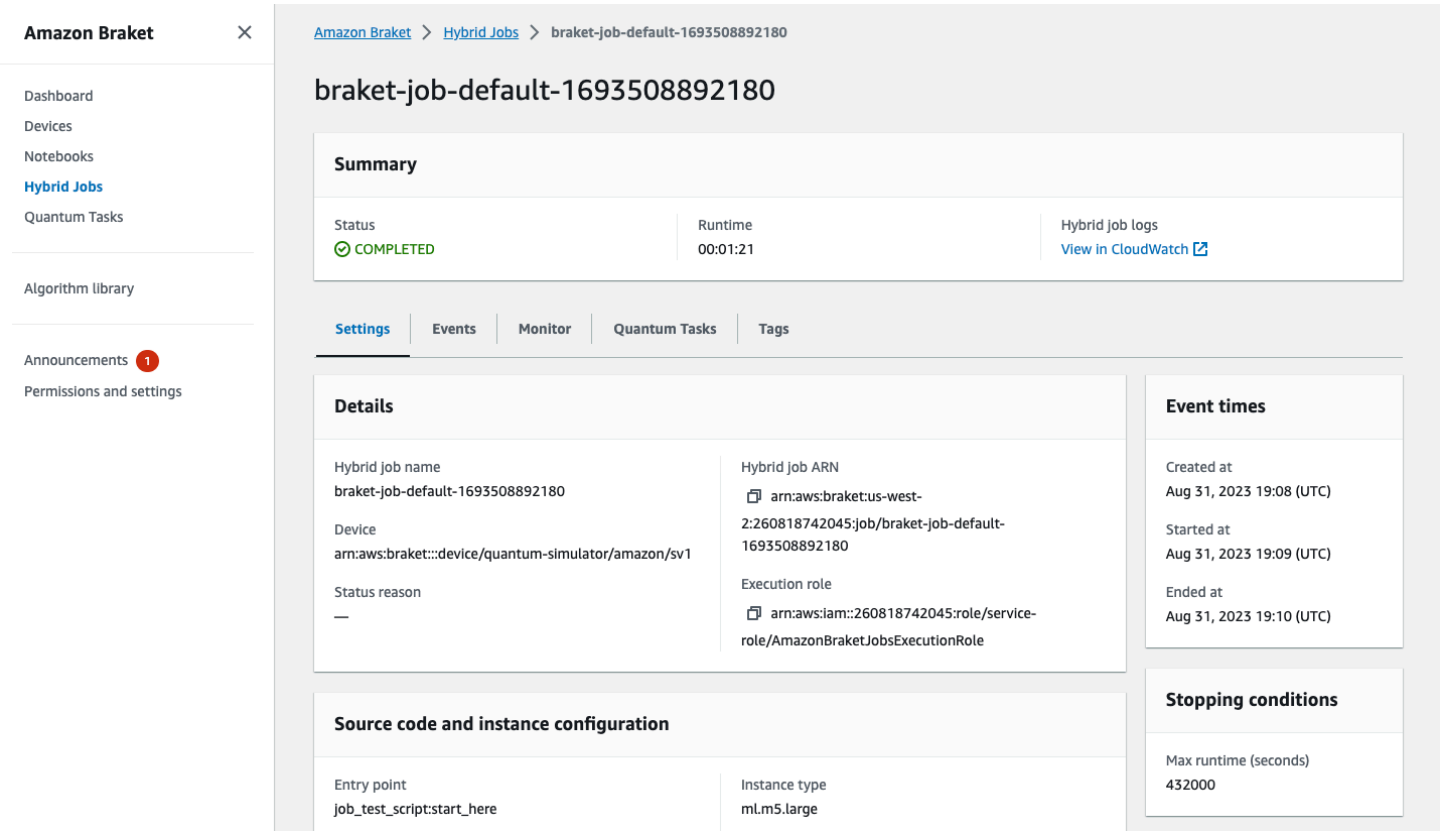
を渡します。実例については、「Amazon Braket examples」Github リポジトリの hybrid jobs フォルダにある Parallelize_training_for_QML.ipynb ファイルを参照してください。

結果をモニタリングする

または、Amazon CloudWatch からのログ出力にアクセスすることもできます。これを行うには、ジョブ詳細ページの左側のメニューにある [ロググループ] タブに移動し、ロググループ aws/braket/jobs をクリックして、ジョブ名を含むログストリームを選択します。これは、上記の例では braket-job-default-1631915042705/algo-1-1631915190 です。



また、コンソールでハイブリッドジョブのステータスを表示するには、[ハイブリッドジョブ] ページを選択して [設定] を選択します。



ハイブリッドジョブが実行されている間、Amazon S3 にアーティファクトがいくつか生成されます。デフォルトの S3 バケット名は `amazon-braket-<region>-<accountid>` で、コンテンツは `jobs/<jobname>/<timestamp>` ディレクトリにあります。Braket Python SDK でハイブリッドジョブを作成する際、別の `code_location` を指定することで、これらのアーティファクトが保存される S3 の場所を設定できます。

> **Note**
>
> この S3 バケットは、ジョブスクリプト AWS リージョン と同じ に配置する必要があります。

`jobs/<jobname>/<timestamp>` ディレクトリには、`model.tar.gz` ファイル内のエントリポイントスクリプトからの出力を含むサブフォルダが含まれています。`script` というディレクトリもあり、この中にある `source.tar.gz` ファイルにはアルゴリズムスクリプトのアーティファクトが含まれています。実際の量子タスクの結果は、`jobs/<jobname>/tasks` という名前のディレクトリにあります。

結果を保存する

アルゴリズムスクリプトによって生成された結果を保存することで、ハイブリッドジョブスクリプトのハイブリッドジョブオブジェクトと Amazon S3 の出力フォルダ (model.tar.gz という名前の tar zip ファイル内) からそれらを使用できるようにすることができます。

出力は JavaScript Object Notation (JSON) 形式を使用してファイルに保存する必要があります。numpy 配列の場合のように、データをテキストに簡単にシリアル化できない場合は、ピクル化データ形式を使用してシリアル化するオプションを入力として渡すことができます。詳細については、「braket.jobs.data_persistence module」を参照してください。

ハイブリッドジョブの結果を保存するには、#ADD でコメントされた次の行を algorithm_script.py ファイルに追加します。

```
import os

from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.jobs import save_job_result  # ADD


def start_here():

    print("Test job started!")

    device = AwsDevice(os.environ['AMZN_BRAKET_DEVICE_ARN'])

    results = []  # ADD

    bell = Circuit().h(0).cnot(0, 1)
    for count in range(5):
        task = device.run(bell, shots=100)
        print(task.result().measurement_counts)
        results.append(task.result().measurement_counts)  # ADD

        save_job_result({"measurement_counts": results})  # ADD

    print("Test job completed!")
```

次に、#ADD でコメントされた行 **`print(job.result())`** を追加することで、ジョブスクリプトのジョブの結果を表示できます。

```
import time
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
)

print(job.arn)
while job.state() not in AwsQuantumJob.TERMINAL_STATES:
    print(job.state())
    time.sleep(10)

print(job.state())
print(job.result())   # ADD
```

この例では、wait_until_complete=True を削除して冗長出力を抑制します。デバッグ用に再度追加できます。このハイブリッドジョブを実行すると、識別子と job-arn が出力され、その後、10秒ごとにハイブリッドジョブの状態が出力されて、ハイブリッドジョブが COMPLETED になったら、ベル回路の結果が示されます。次の例を参照してください。

```
arn:aws:braket:us-west-2:111122223333:job/braket-job-default-123456789012
INITIALIZED
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
...
RUNNING
RUNNING
COMPLETED
{'measurement_counts': [{'11': 53, '00': 47},..., {'00': 51, '11': 49}]}
```

チェックポイントの使用

チェックポイントを使用して、ハイブリッドジョブの中間イテレーションを保存できます。前のセクションのアルゴリズムスクリプトの例では、#ADD でコメントされた次の行を追加して、チェックポイントファイルを作成します。

```
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.jobs import save_job_checkpoint  # ADD
import os


def start_here():

    print("Test job starts!")

    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])

    # ADD the following code
    job_name = os.environ["AMZN_BRAKET_JOB_NAME"]
    save_job_checkpoint(checkpoint_data={"data": f"data for checkpoint from
 {job_name}"}, checkpoint_file_suffix="checkpoint-1")  # End of ADD

    bell = Circuit().h(0).cnot(0, 1)
    for count in range(5):
        task = device.run(bell, shots=100)
        print(task.result().measurement_counts)

    print("Test hybrid job completed!")
```

ハイブリッドジョブを実行すると、デフォルトの /opt/jobs/checkpoints パスを使用して、チェックポイントディレクトリのハイブリッドジョブアーティファクトにファイル <jobname>-checkpoint-1.json が作成されます。このデフォルトパスを変更しない限り、ハイブリッドジョブスクリプトは変更されません。

前のハイブリッドジョブによって生成されたチェックポイントからハイブリッドジョブをロードする場合、アルゴリズムスクリプトは `from braket.jobs import load_job_checkpoint` を使用します。アルゴリズムスクリプトにロードするロジックは次のとおりです。

```
from braket.jobs import load_job_checkpoint

checkpoint_1 = load_job_checkpoint(
```

```
    "previous_job_name",
    checkpoint_file_suffix="checkpoint-1",
)
```

このチェックポイントをロードした後、checkpoint-1 にロードされたコンテンツに基づいてロジックを続行できます。

> **Note**
>
> checkpoint_file_suffix は、チェックポイントの作成時に以前に指定した接尾辞と一致する必要があります。

オーケストレーションスクリプトでは、前のハイブリッドジョブ job-arn を、#ADD でコメントされた行で指定する必要があります。

```
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    source_module="source_dir",
    entry_point="source_dir.algorithm_script:start_here",
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
    copy_checkpoints_from_job="<previous-job-ARN>", #ADD
    )
```

ローカルコードをハイブリッドジョブとして実行する

Amazon Braket Hybrid Jobs は、ハイブリッド量子古典アルゴリズムのフルマネージドオーケストレーションを提供し、Amazon EC2 コンピューティングリソースと Amazon Braket 量子処理ユニット (QPU) アクセスを統合します。ハイブリッドジョブで作成された量子タスクは、個々の量子タスクよりも優先的にキューイングされるため、量子タスクキュー内の変動によってアルゴリズムが中断されることはありません。各 QPU は独自のハイブリッドジョブキューを維持することで、ハイブリッドジョブを一度に 1 つだけ実行するようにします。

このセクションの内容:

- ローカル Python コードからハイブリッドジョブを作成する
- 追加の Python パッケージとソースコードをインストールする
- ハイブリッドジョブインスタンスにおいてデータをロードしたり保存したりする
- ハイブリッドジョブデコレータに関するベストプラクティス

ローカル Python コードからハイブリッドジョブを作成する

ローカル Python コードを Amazon Braket Hybrid Jobs として実行できます。これは、次のコードサンプルに示すように、コードに @hybrid_job デコレータで注釈を付けることで実現できます。カスタム環境では、Amazon Elastic Container Registry (ECR) のカスタムコンテナの使用を選択できます。

> **Note**
>
> デフォルトでは、Python 3.12 のみがサポートされています。

@hybrid_job デコレータを使用して関数に注釈を付けることができます。Braket は、デコレータ内のコードを Braket ハイブリッドジョブのアルゴリズムスクリプトに変換します。次に、ハイブリッドジョブは Amazon EC2 インスタンス上で被デコレート関数を呼び出します。ジョブの進行状況は、`job.state()` または Braket コンソールでモニタリングできます。次のコード例は、State Vector Simulator (SV1) device で 5 つの状態のシーケンスを実行する方法を示しています。

```
from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter, Observable
from braket.devices import Devices
from braket.jobs.hybrid_job import hybrid_job
from braket.jobs.metrics import log_metric

device_arn = Devices.Amazon.SV1


@hybrid_job(device=device_arn)  # Choose priority device
def run_hybrid_job(num_tasks=1):
    device = AwsDevice(device_arn)  # Declare AwsDevice within the hybrid job

    # Create a parametric circuit
    circ = Circuit()
    circ.rx(0, FreeParameter("theta"))
    circ.cnot(0, 1)
    circ.expectation(observable=Observable.X(), target=0)

    theta = 0.0  # Initial parameter

    for i in range(num_tasks):
        task = device.run(circ, shots=100, inputs={"theta": theta})  # Input parameters
```

```
        exp_val = task.result().values[0]

        theta += exp_val  # Modify the parameter (possibly gradient descent)

        log_metric(metric_name="exp_val", value=exp_val, iteration_number=i)

    return {"final_theta": theta, "final_exp_val": exp_val}
```

ハイブリッドジョブを作成するには、通常の Python 関数と同様に被デコレート関数を呼び出します。ただし、デコレータ関数は、被デコレート関数の結果ではなくハイブリッドジョブのハンドルを返します。ジョブの完了後に結果を取得するには、`job.result()` を使用します。

```
job = run_hybrid_job(num_tasks=1)
result = job.result()
```

@hybrid_job デコレータのデバイス引数には、ハイブリッドジョブが優先アクセス権を持つデバイスを指定します。この場合はシミュレーター SV1 です。QPU の優先順位を取得するには、デコレータで指定されたものと一致するデバイス ARN を被デコレート関数内で使用する必要があります。@hybrid_job で宣言されたデバイス ARN をキャプチャするには、ヘルパー関数 `get_job_device_arn()` を使用するという便利な方法もあります。

> **Note**
>
> 各ハイブリッドジョブが Amazon EC2 上にコンテナ化された環境を作成してから起動するまでに、1 分以上かかります。このため、単一の回路や回路のバッチなど、非常に短いワークロードでは、量子タスクを使用するだけで十分です。

ハイパーパラメータ

`run_hybrid_job()` 関数が取る引数として、作成される量子タスクの数を制御できる `num_tasks` があります。ハイブリッドジョブは、この引数をハイパーパラメータとして自動的にキャプチャします。

> **Note**
>
> ハイパーパラメータは、文字列として Braket コンソールに表示されますが、2500 文字の制限があります。

メトリクスとログ記録

run_hybrid_job() 関数内では、反復アルゴリズムからのメトリクスが log_metrics を使用して記録されます。メトリクスは、Braket コンソールページのハイブリッドジョブタブに自動的にプロットされます。Braket コストトラッカーを使用することで、ハイブリッドジョブ実行中の量子タスクのコストをほぼリアルタイムで追跡できます。上記の例では、結果タイプにおける最初の確率を記録するメトリクス名「probability」(確率) を使用しています。

結果を取得する

ハイブリッドジョブが完了したら、 job.result() を使用することで、ハイブリッドジョブの結果を取得できます。return ステートメントに記述したすべてのオブジェクトが、Braket によって自動的にキャプチャされます。関数が返すオブジェクトは、各要素がシリアル化可能なタプルである必要があります。例えば、次のコードは成功している例と失敗している例を示しています。

```
import numpy as np


# Working example
@hybrid_job(device=Devices.Amazon.SV1)
def passing():
    np_array = np.random.rand(5)
    return np_array  # Serializable

# # Failing example
# @hybrid_job(device=Devices.Amazon.SV1)
# def failing():
#     return MyObject() # Not serializable
```

ジョブ名

デフォルトでは、このハイブリッドジョブの名前は関数名を使用して提案されます。また、50 文字までの長さのカスタム名を指定することもできます。例えば、次のコードでは、ジョブ名は「my-job-name」です。

```
@hybrid_job(device=Devices.Amazon.SV1, job_name="my-job-name")
def function():
    pass
```

ローカルモード

ローカルジョブは、デコレータに引数 `local=True` を追加することで作成されます。これにより、ラップトップなどのローカルコンピューティング環境のコンテナ化された環境でハイブリッドジョブが実行されます。ローカルジョブでは、量子タスクに対して優先的なキューイングが行われることはありません。マルチノードや MPI などの高度なケースでは、ローカルジョブが、必要な Braket 環境変数にアクセスできる場合があります。次のコードは、SV1 シミュレーターをデバイスとして使用してローカルハイブリッドジョブを作成しています。

```
@hybrid_job(device=Devices.Amazon.SV1, local=True)
def run_hybrid_job(num_tasks=1):
    return ...
```

この他にも、あらゆるハイブリッドジョブオプションがサポートされています。オプションのリストについては、「braket.jobs.quantum_job_creation module」を参照してください。

追加の Python パッケージとソースコードをインストールする

希望の Python パッケージを使用できるようにランタイム環境をカスタマイズできます。パッケージ名のリストである `requirements.txt` ファイルか、独自のコンテナ (BYOC) のいずれかを使用できます。例えば、`requirements.txt` ファイルには、インストールが必要な他のパッケージを含めることができます。

```
qiskit
pennylane >= 0.31
mitiq == 0.29
```

`requirements.txt` ファイルを使用してランタイム環境をカスタマイズするには、以下のコード例を参照してください。

```
@hybrid_job(device=Devices.Amazon.SV1, dependencies="requirements.txt")
def run_hybrid_job(num_tasks=1):
    return ...
```

または、次のようにパッケージ名を Python リストとして指定することもできます。

```
@hybrid_job(device=Devices.Amazon.SV1, dependencies=["qiskit", "pennylane>=0.31",
 "mitiq==0.29"])
def run_hybrid_job(num_tasks=1):
    return ...
```

追加のソースコードは、モジュールのリストとして指定することも、次のコード例のように単一のモジュールとして指定することもできます。

```
@hybrid_job(device=Devices.Amazon.SV1, include_modules=["my_module1", "my_module2"])
def run_hybrid_job(num_tasks=1):
    return ...
```

ハイブリッドジョブインスタンスにおいてデータをロードしたり保存したりする

入力トレーニングデータを指定する

ハイブリッドジョブを作成する場合は、Amazon Simple Storage Service (Amazon S3) バケットを指定して入力トレーニングデータセットを提供できます。また、ローカルパスを指定すれば、データが Braket によって Amazon S3 の `s3://<default_bucket_name>/jobs/<job_name>/<timestamp>/data/<channel_name>` に自動的にアップロードされます。ローカルパスを指定した場合、チャネル名はデフォルトで「input」になります。次のコードは、ローカルパス `data/file.npy` からアップロードされた numpy ファイルを示しています。

```
import numpy as np


@hybrid_job(device=Devices.Amazon.SV1, input_data="data/file.npy")
def run_hybrid_job(num_tasks=1):
    data = np.load("data/file.npy")
    return ...
```

S3 の場合は、`get_input_data_dir()` ヘルパー関数を使用する必要があります。

```
import numpy as np
from braket.jobs import get_input_data_dir

s3_path = "s3://amazon-braket-us-east-1-123456789012/job-data/file.npy"


@hybrid_job(device=None, input_data=s3_path)
def job_s3_input():
    np.load(get_input_data_dir() + "/file.npy")


@hybrid_job(device=None, input_data={"channel": s3_path})
def job_s3_input_channel():
```

```
    np.load(get_input_data_dir("channel") + "/file.npy")
```

チャネル値と S3 URI またはローカルパスで構成されるディクショナリを提供することで、複数の入力データソースを指定できます。

```
import numpy as np
from braket.jobs import get_input_data_dir

input_data = {
    "input": "data/file.npy",
    "input_2": "s3://amzn-s3-demo-bucket/data.json"
}


@hybrid_job(device=None, input_data=input_data)
def multiple_input_job():
    np.load(get_input_data_dir("input") + "/file.npy")
    np.load(get_input_data_dir("input_2") + "/data.json")
```

> **Note**
>
> 入力データが大きい (>1GB) 場合、ハイブリッドジョブが作成されるまでの待機時間が長くなります。これは、ローカル入力データが最初に S3 バケットにアップロードされた時点で S3 パスがジョブリクエストに追加されるためです。つまり、ジョブリクエストが Braket サービスに送信されのが最後であるためです。

結果を S3 に保存する

被デコレート関数の return ステートメントに指定されていない結果を保存するには、すべてのファイル書き込みオペレーションに正しいディレクトリを追加する必要があります。次の例は、numpy 配列と matplotlib 図の保存を示しています。

```
import matplotlib.pyplot as plt
import numpy as np


@hybrid_job(device=Devices.Amazon.SV1)
def run_hybrid_job(num_tasks=1):
    result = np.random.rand(5)
```

```
    # Save a numpy array
    np.save("result.npy", result)

    # Save a matplotlib figure
    plt.plot(result)
    plt.savefig("fig.png")
    return ...
```

すべての結果は model.tar.gz という名前のファイルに圧縮されます。この結果をダウンロードするには、Python 関数 job.result() を使用するか、または Braket マネジメントコンソールのハイブリッドジョブページにある結果フォルダに移動します。

チェックポイントからの保存と再開

長時間実行されるハイブリッドジョブの場合、アルゴリズムの中間状態を定期的に保存することをお勧めします。組み込みの save_job_checkpoint() ヘルパー関数を使用するか、ファイルを AMZN_BRAKET_JOB_RESULTS_DIR パスに保存できます。後者のパスはヘルパー関数 get_job_results_dir() で使用できます。

以下は、最小限の作業によって、ハイブリッドジョブデコレータを使用してチェックポイントを保存およびロードする例です。

```
from braket.jobs import save_job_checkpoint, load_job_checkpoint, hybrid_job


@hybrid_job(device=None, wait_until_complete=True)
def function():
    save_job_checkpoint({"a": 1})


job = function()
job_name = job.name
job_arn = job.arn


@hybrid_job(device=None, wait_until_complete=True, copy_checkpoints_from_job=job_arn)
def continued_function():
    load_job_checkpoint(job_name)


continued_job = continued_function()
```

最初のハイブリッドジョブでは、 save_job_checkpoint() が、保存するデータを含むディクショナリを指定して呼び出されています。デフォルトでは、すべての値がテキストとしてシリアル化されます。numpy 配列など、より複雑な Python オブジェクトのチェックポイントファイルを作成するには、data_format = PersistedJobDataFormat.PICKLED_V4 を設定します。このコードは、「checkpoints」というサブフォルダにあるハイブリッドジョブアーティファクトに、デフォルト名 <jobname>.json を持つチェックポイントファイルを作成および上書きします。

チェックポイントから再開する新しいハイブリッドジョブを作成するには、copy_checkpoints_from_job=job_arn を渡す必要があります。ここで、job_arn は直前のジョブのハイブリッドジョブ ARN です。次に、load_job_checkpoint(job_name) を使用してチェックポイントの内容をロードします。

ハイブリッドジョブデコレータに関するベストプラクティス

非同期性を採用する

デコレータ注釈を使用して作成されたハイブリッドジョブは、古典的リソースと量子リソースが利用可能になると実行され、非同期です。アルゴリズムの進行状況をモニタリングするには、Braket Management Console または Amazon CloudWatch を使用します。実行すべきアルゴリズムを送信すると、Braket により、スケーラブルなコンテナ化された環境で実行され、完了すると結果が取得されます。

反復変分アルゴリズムを実行する

ハイブリッドジョブでは、反復量子古典アルゴリズムを実行するツールを使用できます。純粋に量子力学的な問題には、量子タスクまたは量子タスクのバッチを使用します。特定の QPU への優先的なアクセスは、古典的な処理で QPU への複数の反復呼び出しを断続的に必要とする、長時間実行される変分アルゴリズムに最も有益です。

ローカルモードを使用してデバッグする

QPU でハイブリッドジョブを実行する場合は、まずシミュレーター SV1 で実行して、期待どおりに実行されるかを確認することをお勧めします。小規模なテストでは、ローカルモードで実行することで迅速な反復実行とデバッグを行うことができます。

独自のコンテナ (BYOC) により再現性を向上する

コンテナ化された環境内にソフトウェアとその依存関係をカプセル化することで、再現可能な実験を作成します。すべてのコード、依存関係、設定をコンテナにパッケージ化することで、潜在的な競合やバージョニングの問題を回避できます。

マルチインスタンス分散シミュレーター

多数の回路を実行するには、組み込みの MPI サポートを使用して、単一のハイブリッドジョブにおいて、複数のインスタンスでローカルシミュレーターを実行することを検討してください。詳細については、「embedded simulators」を参照してください。

パラメトリック回路を使用する

ハイブリッドジョブから送信したパラメトリック回路は、パラメトリックコンパイルを使用して特定の QPU で自動的にコンパイルされるため、アルゴリズムの実行時間が短縮されます。

定期的にチェックポイントを作成する

長時間実行されるハイブリッドジョブの場合、アルゴリズムの中間状態を定期的に保存することをお勧めします。

その他の例、ユースケース、ベストプラクティスについては、GitHub の「Amazon Braket examples」を参照してください。

ハイブリッドジョブでの API の使用方法

API を直接使用することで、Amazon Braket Hybrid Jobs にアクセスして操作できます。ただし、API を直接使用する場合にはデフォルトおよび便利なメソッドは使用できなくなります。

> **Note**
>
> Amazon Braket Hybrid Jobs の操作には Amazon Braket Python SDK を使用することを強くお勧めします。ハイブリッドジョブが正常に実行されるようにする便利なデフォルトと保護が提供されているためです。

このトピックでは、API の基本的な使用方法について説明します。API を使用するというアプローチは、SDK を使用する場合よりも複雑になることに注意してください。ハイブリッドジョブを実行できるようになるまでに、何度も試行錯誤する覚悟が必要です。

API を使用するには、アカウントに、`AmazonBraketFullAccess` マネージドポリシーを持つロールが必要です。

> **Note**
>
> AmazonBraketFullAccess マネージドポリシーでロールを取得する方法の詳細については、「Amazon Braket を有効にする」ページを参照してください。

また、実行ロールも必要です。このロールはサービスに渡されます。このロールを作成するには、Amazon Braket コンソールを使用します。[アクセス許可と設定] ページの [実行ロール] タブを使用して、ハイブリッドジョブのデフォルトロールを作成できます。

CreateJob API では、ハイブリッドジョブに必要なパラメータをすべて指定する必要があります。Python を使用するには、input.tar.gz ファイルなどのアルゴリズムスクリプトファイルを tar バンドルに圧縮し、次のスクリプトを実行します。コードのうち山括弧 (<>) で囲まれた部分を更新して、ハイブリッドジョブを開始するパス、ファイル、メソッドを指定するアカウント情報およびエントリポイントに一致させます。

```
from braket.aws import AwsDevice, AwsSession
import boto3
from datetime import datetime

s3_client = boto3.client("s3")
client = boto3.client("braket")

project_name = "job-test"
job_name = project_name + "-" + datetime.strftime(datetime.now(), "%Y%m%d%H%M%S")
bucket = "amazon-braket-<your_bucket>"
s3_prefix = job_name

job_script = "input.tar.gz"
job_object = f"{s3_prefix}/script/{job_script}"
s3_client.upload_file(job_script, bucket, job_object)

input_data = "inputdata.csv"
input_object = f"{s3_prefix}/input/{input_data}"
s3_client.upload_file(input_data, bucket, input_object)

job = client.create_job(
    jobName=job_name,
    roleArn="arn:aws:iam::<your_account>:role/service-role/
AmazonBraketJobsExecutionRole",  # https://docs.aws.amazon.com/braket/latest/
developerguide/braket-manage-access.html#about-amazonbraketjobsexecution
```

```
    algorithmSpecification={
        "scriptModeConfig": {
            "entryPoint": "<your_execution_module>:<your_execution_method>",
            "containerImage": {"uri": "292282985366.dkr.ecr.us-west-1.amazonaws.com/
amazon-braket-base-jobs:1.0-cpu-py37-ubuntu18.04"},   # Change to the specific region
 you are using
            "s3Uri": f"s3://{bucket}/{job_object}",
            "compressionType": "GZIP"
        }
    },
    inputDataConfig=[
        {
            "channelName": "hellothere",
            "compressionType": "NONE",
            "dataSource": {
                "s3DataSource": {
                    "s3Uri": f"s3://{bucket}/{s3_prefix}/input",
                    "s3DataType": "S3_PREFIX"
                }
            }
        }
    ],
    outputDataConfig={
        "s3Path": f"s3://{bucket}/{s3_prefix}/output"
    },
    instanceConfig={
        "instanceType": "ml.m5.large",
        "instanceCount": 1,
        "volumeSizeInGb": 1
    },
    checkpointConfig={
        "s3Uri":  f"s3://{bucket}/{s3_prefix}/checkpoints",
        "localPath": "/opt/omega/checkpoints"
    },
    deviceConfig={
        "priorityAccess": {
            "devices": [
                "arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3"
            ]
        }
    },
    hyperParameters={
        "hyperparameter key you wish to pass": "<hyperparameter value you wish to
 pass>",
```

```
    },
    stoppingCondition={
        "maxRuntimeInSeconds": 1200,
        "maximumTaskLimit": 10
    },
)
```

ハイブリッドジョブの作成後に、ハイブリッドジョブの詳細にアクセスするには、GetJob API またはコンソールを使用します。直前の例のように createJob コードを実行した Python セッションからハイブリッドジョブの詳細を取得するには、次の Python コマンドを使用します。

```
getJob = client.get_job(jobArn=job["jobArn"])
```

ハイブリッドジョブをキャンセルするには、ジョブ ('JobArn') の <noloc>Amazon リソースネーム</noloc>を指定して CancelJob API を呼び出します。

```
cancelJob = client.cancel_job(jobArn=job["jobArn"])
```

チェックポイントを指定するには、createJob API の一部として checkpointConfig パラメータを使用します。

```
    checkpointConfig = {
        "localPath" : "/opt/omega/checkpoints",
        "s3Uri": f"s3://{bucket}/{s3_prefix}/checkpoints"
    },
```

> **Note**
>
> checkpointConfig のローカルパスの先頭を、予約済みパスである /opt/ml、/opt/braket、/tmp、または /usr/local/nvidia のいずれかにすることはできません。

ローカルモードでのハイブリッドジョブの作成およびデバッグ

新しいハイブリッドアルゴリズムを構築する際、ローカルモードはアルゴリズムスクリプトのデバッグとテストに役立ちます。ローカルモードは、Amazon Braket Hybrid Jobs で使用する予定のコードを実行できる機能ですが、Braket がハイブリッドジョブを実行するためのインフラストラクチャを管理する必要はありません。代わりに、Amazon Braket ノートインスタンスまたはノートブックパ

ソコンやデスクトップコンピュータなどの優先クライアントでハイブリッドジョブをローカルで実行します。

ローカルモードでは、量子タスクを実際のデバイスに送信することはできますが、ローカルモードで実際の量子処理ユニット (QPU) に対して実行すると、パフォーマンス上の利点は得られません。

ローカルモードを使用するには、プログラム内で AwsQuantumJob が使用されている箇所で、AwsQuantumJob を LocalQuantumJob に変更します。例えば、「最初のジョブを作成する」に記載されている例を実行するには、コード内のハイブリッドジョブスクリプトを次のように編集します。

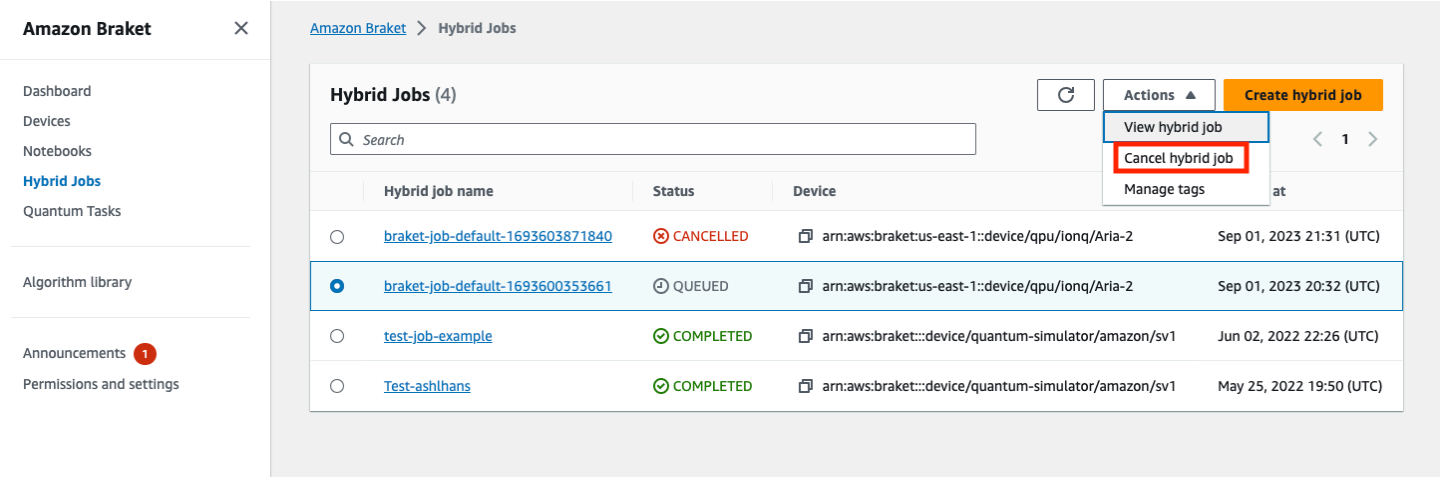
```
from braket.jobs.local import LocalQuantumJob

job = LocalQuantumJob.create(
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
)
```

> **Note**
>
> この機能を使用するには、Amazon Braket ノートブックに既にプリインストールされている Docker をローカル環境にインストールする必要があります。Docker のインストール手順については、「Get Docker」ページを参照してください。また、ローカルモードではすべてのパラメータがサポートされているわけではありません。

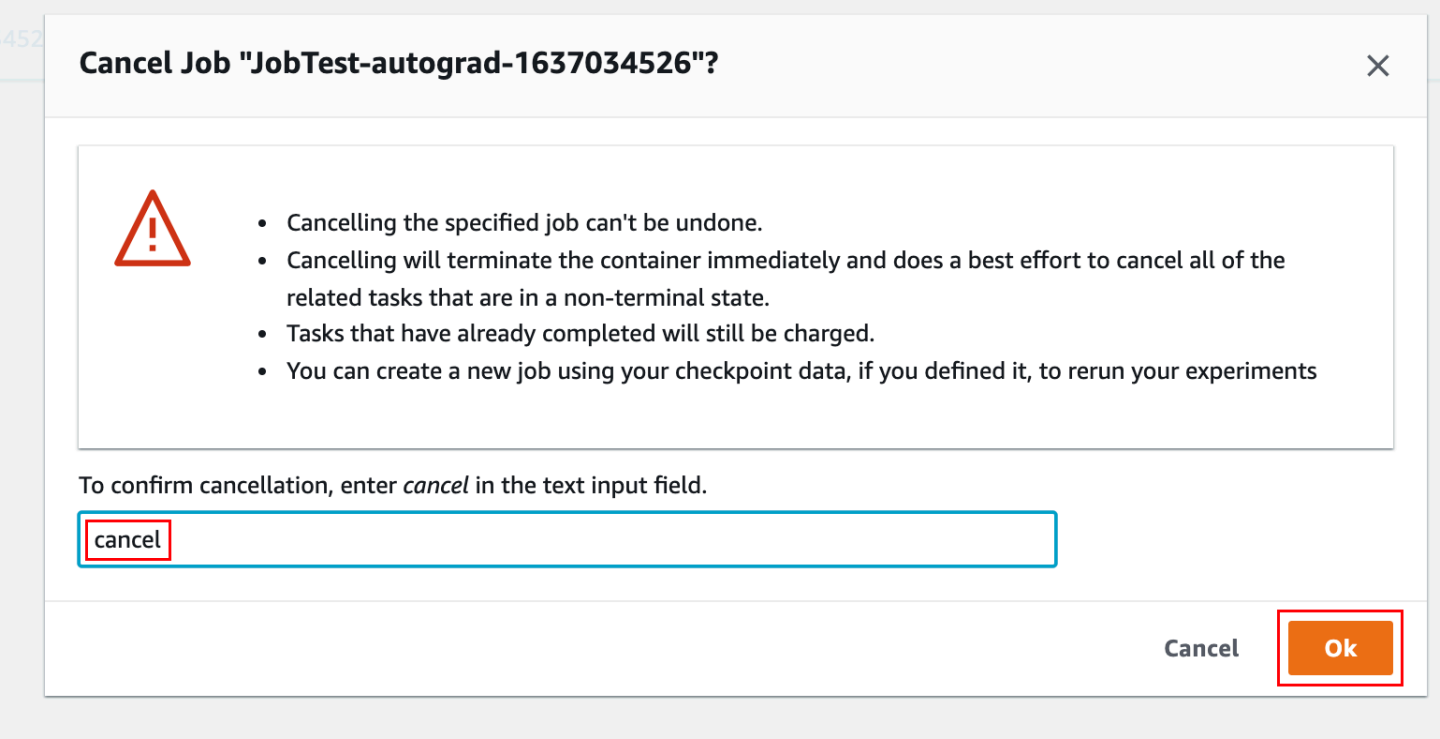
ハイブリッドジョブのキャンセル

最終ステータス以外のハイブリッドジョブをキャンセルすることが必要になる場合があります。これは、コンソールまたはコードで行うことができます。

コンソールでハイブリッドジョブをキャンセルするには、[ハイブリッドジョブ] ページからキャンセルするハイブリッドジョブを選択し、[アクション] ドロップダウンメニューから [ハイブリッドジョブをキャンセル] を選択します。



キャンセルを確定するには、プロンプトが表示されたら入力フィールドに「cancel」と入力し、[OK] を選択します。



Braket Python SDK のコードを使用してハイブリッドジョブをキャンセルするには、`job_arn` を使用してハイブリッドジョブを特定し、次のコードに示すように `cancel` コマンドを呼び出します。

```
job = AwsQuantumJob(arn=job_arn)
job.cancel()
```

`cancel` コマンドは、ハイブリッドジョブの古典側コンテナを直ちに終了し、まだ最終ステータスに到達していない関連量子タスクをすべて、ベストエフォート方式でキャンセルします。

ハイブリッドジョブのカスタマイズ

Amazon Braket には、ハイブリッドジョブの実行方法をカスタマイズするいくつかの方法が用意されており、特定のニーズに合わせて環境を調整できます。このセクションでは、アルゴリズムスクリプト環境の定義から独自のコンテナまで、ハイブリッドジョブをカスタマイズできるオプションについて説明します。ハイパーパラメータを使用したワークフローの最適化、ジョブインスタンスの設定、およびパラメトリックコンパイルを活用したパフォーマンスの向上を行う方法について説明します。これらのカスタマイズ手法は、Amazon Braket でのハイブリッド量子計算の潜在能力を最大限引き出すのに役立ちます。

このセクションの内容:

- アルゴリズムスクリプトの環境を定義する
- ハイパーパラメータの使用方法
- ハイブリッドジョブインスタンスの設定
- パラメトリックコンパイルを使用したハイブリッドジョブの高速化

アルゴリズムスクリプトの環境を定義する

Amazon Braket は、アルゴリズムスクリプトのコンテナによって定義された環境をサポートしています。

- ベースコンテナ (`image_uri` が指定されていない場合のデフォルト)
- CUDA-Q のコンテナ
- Tensorflow と PennyLane のコンテナ
- PyTorch、PennyLane、CUDA-Q のコンテナ

次の表に、コンテナとその付属ライブラリの詳細を示します。

Amazon Braket コンテナ

タイプ	ベース	CUDA-Q	TensorFlow	PyTorch
イメージ URI	292282985366.dkr.ecr.us-west-2.amazonaws.com/	292282985366.dkr.ecr.us-west-2.amazonaws.com/	292282985366.dkr.ecr.us-east-1.amazonaws.com/	292282985366.dkr.ecr.us-west-2.amazonaws.com/

タイプ	ベース	CUDA-Q	TensorFlow	PyTorch
	amazon-braket-base-jobs:latest	amazon-braket-cudaq-jobs:latest	amazon-braket-tensorflow-jobs:latest	amazon-braket-pytorch-jobs:latest
継承ライブラリ		• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • awscli • botocore • boto3 • dask • matplotlib • numpy • pandas • PennyLane • PennyLane-Lightning • qiskit-braket-provider • リクエスト • sagemaker-training • scikit-learn • scipy	• awscli • numpy • pandas • scipy	• awscli • numpy • pandas • scipy

タイプ	ベース	CUDA-Q	TensorFlow	PyTorch
追加のライブラリ	• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • awscli • boto3 • ipykernel • matplotlib • networkx • numpy • openbabel • pandas • PennyLane • protobuf • psi4 • rsa • scipy	• cudaq • cudaq-qec • cudaq-solvers	• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • ipykernel • keras • matplotlib • networkx • openbabel • PennyLane • protobuf • psi4 • rsa • PennyLane-Lightning-gpu • cuQuantum	• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • ipykernel • keras • matplotlib • networkx • openbabel • PennyLane • protobuf • psi4 • rsa • PennyLane-Lightning-gpu • cuQuantum • cudaq • cudaq-qec • cudaq-solvers

オープンソースのコンテナ定義は、aws/amazon-braket-containers で確認およびアクセスできます。ユースケースに一致するコンテナを選択します。Braket で使用可能な AWS リージョン (us-east-1、us-west-1、us-west-2、eu-north-1、eu-west-2) のいずれかを使用できますが、コンテナリージョンはハイブリッドジョブのリージョンと一致する必要があります。ハイブリッドジョブを作

成する際、次の 3 つの引数のいずれかをハイブリッドジョブスクリプトの `create(…)` コールに追加することでコンテナイメージを指定します。Amazon Braket コンテナはインターネット接続を備えているため、実行時に選択したコンテナに追加の依存関係を (起動時間または実行時間を犠牲にして) インストールできます。次の例は、us-west-2 リージョンの場合です。

- ベースイメージ: image_uri"292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:latest"
- CUDA-Q イメージ: image_uri"292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:latest"
- Tensorflow イメージ: image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-tensorflow-jobs:latest"
- PyTorch イメージ: image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:latest"

また、`image-uris` は、Amazon Braket SDK の `retrieve_image()` 関数を使用して取得することもできます。次の例は、us-west-2 からそれらを取得する方法を示しています AWS リージョン。

```
from braket.jobs.image_uris import retrieve_image, Framework

image_uri_base = retrieve_image(Framework.BASE, "us-west-2")
image_uri_cudaq = retrieve_image(Framework.CUDAQ, "us-west-2")
image_uri_tf = retrieve_image(Framework.PL_TENSORFLOW, "us-west-2")
image_uri_pytorch = retrieve_image(Framework.PL_PYTORCH, "us-west-2")
```

独自のコンテナ (BYOC)

Amazon Braket Hybrid Jobs には、さまざまな環境でコードを実行するための 3 つの構築済みコンテナが用意されています。これらのコンテナの 1 つがユースケースをサポートしている場合は、ハイブリッドジョブを作成するときにアルゴリズムスクリプトを指定するだけで済みます。`pip` を使用して `requirements.txt` ファイルから、またはアルゴリズムスクリプトから、欠落している小さな依存関係を追加できます。

これらのコンテナがユースケースをサポートしていない場合、またはそれらを拡張したい場合、Braket Hybrid Jobs は、独自のカスタム Docker コンテナイメージまたは独自のコンテナ (BYOC) を使用したハイブリッドジョブの実行をサポートします。それがユースケースに適した機能であることを確認してください。

このセクションの内容:

- 独自のコンテナは、どのような場合に適した選択肢となりますか?
- 独自のコンテナのレシピ
- 独自のコンテナでの Braket ハイブリッドジョブの実行

独自のコンテナは、どのような場合に適した選択肢となりますか?

Braket Hybrid Jobs で独自のコンテナ (BYOC) を使用すると、パッケージ化された環境に独自のソフトウェアをインストールして利用できる柔軟性が得られます。お客様のニーズによっては、同じ柔軟性を実現できる方法があります。そのような場合、「BYOC Docker を構築して Amazon ECR にアップロードし、カスタムイメージ URI を取得する」という完全なサイクルを経る必要がありません。

> **Note**
>
> 公開されている少数の Python パッケージ (通常は 10 未満) を追加する場合は、BYOC が適した選択肢にならない場合があります。例えば、PyPi を使用する場合です。

この場合、構築済みの Braket イメージのいずれかを使用し、ジョブ送信時にソースディレクトリに `requirements.txt` ファイルを含めることができます。このファイルが自動的に読み込まれ、`pip` により、パッケージが指定されたバージョンで正常にインストールされます。多数のパッケージをインストールする場合、ジョブの実行時間が大幅に長くなる可能性があります。Python を確認します。また、必要に応じて、ソフトウェアが動作するかどうかをテストするために使用する構築済みコンテナの CUDA バージョンを確認します。

BYOC が必要になるのは、ジョブスクリプトに Python 以外の言語 (C++ や Rust など) を使用する場合や、Braket の構築済みコンテナでは利用できない Python バージョンを使用する場合です。BYOC は、次の場合にも適した選択肢です。

- ソフトウェアをライセンスキーで使用するため、そのソフトウェアを実行するのにライセンスサーバーに対するそのキーの認証が必要となる場合。BYOC を使用を使用してライセンスキーを Docker イメージに埋め込むことで、ソフトウェアの認証コードを含めることができます。
- 公開されていないソフトウェアを使用する場合。例えば、ソフトウェアが、アクセスするのに特定の SSH キーが必要なプライベートな GitLab または GitHub リポジトリでホストされている場合などです。

- Braket が提供するコンテナにパッケージ化されていない大規模なソフトウェアスイートをインストールする必要がある場合。ソフトウェアのインストールによってハイブリッドジョブコンテナの起動時間が長くなるのを、BYOC の使用によって回避できます。

また、BYOC では、ソフトウェアを含んだ Docker コンテナを構築し、それを各ユーザーが利用できるようにすることで、カスタムの SDK またはアルゴリズムを各ユーザーが利用できるようにします。これを行うには、Amazon ECR で適切なアクセス許可を設定します。

> **Note**
>
> 該当するすべてのソフトウェアライセンスの条項を遵守する必要があります。

独自のコンテナのレシピ

このセクションでは、Braket Hybrid Jobs に bring your own container (BYOC) を持ち込むために必要なもの、つまりスクリプト、ファイル、および、それらを組み合わせてカスタム Docker イメージを起動して実行するための手順を順を追って説明します。2 つの代表的なケースに対するレシピ:

1. Docker イメージに追加のソフトウェアをインストールし、ジョブ内で Python アルゴリズムスクリプトのみを使用するケース。
2. Python 以外の言語で記述されたアルゴリズムスクリプトと Hybrid Jobs を使用するか、x86 以外の CPU アーキテクチャを使用するケース。

ケース 2 では、コンテナエントリスクリプトの定義がより複雑になります。

Braket は、ハイブリッドジョブを実行する際、リクエストされた数とタイプの Amazon EC2 インスタンスを起動し、それらのインスタンス上でジョブ作成へのイメージ URI 入力で指定された Docker イメージを実行します。BYOC 機能を使用する場合は、読み取りアクセス権がある[プライベート Amazon ECR リポジトリ]でホストされているイメージ URI を指定します。Braket Hybrid Jobs は、当該のカスタムイメージを使用してジョブを実行します。

Hybrid Jobs で使用できる Docker イメージを構築するには、特定のコンポーネントが必要です。`Dockerfiles` の記述と構築に慣れていない場合は、「Dockerfile documentation」と「Amazon ECR CLI documentation」を参照してください。

要件:

- Dockerfile のベースイメージ

- (オプション) 変更されたコンテナエントリポイントスクリプト
- Dockerfile を使用して必要なソフトウェアとコンテナスクリプトをインストールする

Dockerfile のベースイメージ

Python を使用し、Braket が提供するコンテナにソフトウェアをインストールする場合、ベースイメージの選択肢としては、GitHub リポジトリと Amazon ECR でホストされている Braket コンテナイメージの 1 つがあります。イメージをプルしてその上に構築するには、Amazon ECR に対して認証する必要があります。例えば、BYOC Docker ファイルの最初の行は次のように指定できます: `FROM [IMAGE_URI_HERE]`。

次に、Dockerfile の残りの部分に入力することで、コンテナに追加するソフトウェアをインストールして設定します。構築済みの Braket イメージには既に適切なコンテナエントリポイントスクリプトが含まれているため、あらためて含める必要はありません。

C++、Rust、Julia など Python 以外の言語を使用する場合、または ARM など x86 以外の CPU アーキテクチャ用のイメージを構築する場合は、ベアボーンパブリックイメージ上に構築する必要がある場合があります。このようなイメージを Amazon Elastic Container Registry Public Gallery で数多く見つけることができます。CPU アーキテクチャに適切 (であるとともに、必要に応じて使用する GPU にも適切) なイメージを必ず選択してください。

(オプション) 変更されたコンテナエントリポイントスクリプト

> **Note**
>
> 構築済みの Braket イメージにソフトウェアを追加するだけの場合は、このセクションをスキップできます。

ハイブリッドジョブの一部として Python 以外のコードを実行するには、コンテナエントリポイントを定義する Python スクリプトを変更します。例として、`braket_container.py`Amazon Braket Github の Python スクリプトを挙げます。このスクリプトは、Braket によって事前に構築されたイメージがアルゴリズムスクリプトを起動し、適切な環境変数を設定するために使用します。コンテナエントリポイントスクリプト自体は Python で記述される必要がありますが、Python 以外のスクリプトも起動できます。当該の構築済みの例では Python アルゴリズムスクリプトが Python サブプロセスまたはまったく新しいプロセスとして起動されていることが分かります。この口ジックを変更することで、Python 以外のアルゴリズムスクリプトをエントリポイントスクリプトが起動できるよう

にすることができます。例えば、末尾に付いているファイル拡張子に応じて Rust プロセスを起動するように thekick_off_customer_script() 関数を変更できます。

まったく新しい braket_container.py を記述することもできます。その Python スクリプトでは、入力データ、ソースアーカイブ、その他の必要なファイルを Amazon S3 からコンテナにコピーし、適切な環境変数を定義する必要があります。

Dockerfile を使用して必要なソフトウェアとコンテナスクリプトをインストールする

> **Note**
>
> 構築済みの Braket イメージを Docker ベースイメージとして使用する場合、コンテナスクリプトは既に存在しています。

直前のステップで、変更したコンテナスクリプトを作成した場合は、それをコンテナにコピーし、かつ環境変数 SAGEMAKER_PROGRAM を、braket_container.py か、新しいコンテナエントリポイントスクリプトに付けた名前に設定します。

以下は、GPU アクセラレーションジョブインスタンスで Julia を使用できるようにする Dockerfile の例です。

```
FROM nvidia/cuda:12.2.0-devel-ubuntu22.04


 ARG DEBIAN_FRONTEND=noninteractive
 ARG JULIA_RELEASE=1.8
 ARG JULIA_VERSION=1.8.3


 ARG PYTHON=python3.11
 ARG PYTHON_PIP=python3-pip
 ARG PIP=pip


 ARG JULIA_URL = https://julialang-s3.julialang.org/bin/linux/x64/${JULIA_RELEASE}/
 ARG TAR_NAME = julia-${JULIA_VERSION}-linux-x86_64.tar.gz


 ARG PYTHON_PKGS = # list your Python packages and versions here
```

```
RUN curl -s -L ${JULIA_URL}/${TAR_NAME} | tar -C /usr/local -x -z --strip-components=1
-f -


RUN apt-get update \

   && apt-get install -y --no-install-recommends \

   build-essential \

   tzdata \

   openssh-client \

   openssh-server \

   ca-certificates \

   curl \

   git \

   libtemplate-perl \

   libssl1.1 \

   openssl \

   unzip \

   wget \

   zlib1g-dev \

   ${PYTHON_PIP} \

   ${PYTHON}-dev \




RUN ${PIP} install --no-cache --upgrade ${PYTHON_PKGS}
```

```
 RUN ${PIP} install --no-cache --upgrade sagemaker-training==4.1.3


 # Add EFA and SMDDP to LD library path
 ENV LD_LIBRARY_PATH="/opt/conda/lib/python${PYTHON_SHORT_VERSION}/site-packages/
smdistributed/dataparallel/lib:$LD_LIBRARY_PATH"
 ENV LD_LIBRARY_PATH=/opt/amazon/efa/lib/:$LD_LIBRARY_PATH


 # Julia specific installation instructions
 COPY Project.toml /usr/local/share/julia/environments/v${JULIA_RELEASE}/
 RUN JULIA_DEPOT_PATH=/usr/local/share/julia \

    julia -e 'using Pkg; Pkg.instantiate(); Pkg.API.precompile()'
 # generate the device runtime library for all known and supported devices
 RUN JULIA_DEPOT_PATH=/usr/local/share/julia \

    julia -e 'using CUDA; CUDA.precompile_runtime()'


 # Open source compliance scripts
 RUN HOME_DIR=/root \

 && curl -o ${HOME_DIR}/oss_compliance.zip https://aws-dlinfra-
utilities.s3.amazonaws.com/oss_compliance.zip \

 && unzip ${HOME_DIR}/oss_compliance.zip -d ${HOME_DIR}/ \

 && cp ${HOME_DIR}/oss_compliance/test/testOSSCompliance /usr/local/bin/
testOSSCompliance \

 && chmod +x /usr/local/bin/testOSSCompliance \

 && chmod +x ${HOME_DIR}/oss_compliance/generate_oss_compliance.sh \

 && ${HOME_DIR}/oss_compliance/generate_oss_compliance.sh ${HOME_DIR} ${PYTHON} \

 && rm -rf ${HOME_DIR}/oss_compliance*


 # Copying the container entry point script
 COPY braket_container.py /opt/ml/code/braket_container.py
 ENV SAGEMAKER_PROGRAM braket_container.py
```

この例では、 は が提供するスクリプトをダウンロードして実行 AWS し、関連するすべてのオープンソースライセンスへの準拠を確保します。この遵守は例えば、MIT license が適用されるインストール済みコードに対し、適切な帰属表現を行う、といった方法によって行われます。

プライベートな GitHub または GitLab のリポジトリでホストされているコードなど、公開されていないコードを含める必要がある場合は、そういったコードにアクセスするための SSH キーを Docker イメージに埋め込まないでください。代わりに、構築時に Docker Compose を使用することで、構築のベースとするホストマシン上の SSH へのアクセスを Docker に許可します。詳細については、ガイド「Securely using SSH keys in Docker to access private Github repositories」を参照してください。

Docker イメージを構築し、アップロードする

`Dockerfile` を適切に定義できたため、プライベート Amazon ECR リポジトリ (がまだ存在しない場合に、それ) を作成する手順を実行する準備が整いました。また、コンテナイメージの構築、タグ付け、リポジトリへのアップロードも行うことができます。

イメージを構築、タグ付け、プッシュする準備ができたとします。`docker build` のオプションの完全な説明といくつかの例については、「Docker build documentation」を参照してください。

上記のサンプルファイルの場合は、以下を実行できます。

```
aws ecr get-login-password --region ${your_region} | docker login --username AWS --
password-stdin ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com
 docker build -t braket-julia .
 docker tag braket-julia:latest ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com/
braket-julia:latest
 docker push ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com/braket-julia:latest
```

適切な Amazon ECR アクセス許可を割り当てる

Braket Hybrid Jobs Docker イメージは、プライベートな Amazon ECR リポジトリでホストされている必要があります。デフォルトでは、プライベート Amazon ECR リポジトリは、Braket Hybrid Jobs IAM role への読み取りアクセスや、共同作業者や学生などイメージの使用を希望する他のユーザーへの読み取りアクセスを提供しません。適切なアクセス許可を付与するリポジトリポリシーを設定する必要があります。一般に、イメージにアクセスしてもらう必要がある特定のユーザーおよび IAM ロールにのみアクセス許可を付与し、当該の image URI を持つ任意のユーザーにイメージのプルを許可しないでください。

独自のコンテナでの Braket ハイブリッドジョブの実行

独自のコンテナでハイブリッドジョブを作成するには、指定された image_uri 引数を使用して、AwsQuantumJob.create() を呼び出します。QPU かオンデマンドシミュレーターを使用することも、Braket Hybrid Jobs で使用できる古典的プロセッサでコードをローカルに実行することもできます。実際の QPU で実行する前に、SV1、DM1、TN1 などのシミュレーターでコードをテストすることをお勧めします。

古典的プロセッサでコードを実行するには、InstanceConfig を更新して、instanceType および使用する instanceCount を指定します。instance_count > 1 を指定する場合は、コードが複数のホストで実行できることを確認しておく必要があります。選択できるインスタンスの数の上限は 5 です。例えば、次のようになります。

```
job = AwsQuantumJob.create(
    source_module="source_dir",
    entry_point="source_dir.algorithm_script:start_here",
    image_uri="111122223333.dkr.ecr.us-west-2.amazonaws.com/my-byoc-container:latest",
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge", instanceCount=3),
    device="local:braket/braket.local.qubit",
    # ...)
```

> **Note**
>
> デバイス ARN を使用して、ハイブリッドジョブメタデータとして使用したシミュレーターを追跡します。「device = "local:<provider>/<simulator_name>"」の形式に従った値が許容されます。<provider> および <simulator_name> は英字、数字、_、-、および . のみで構成する必要があります。文字列サイズは 256 文字に制限されています。BYOC を使用する予定で、Braket SDK を使用しないで量子タスクを作成する場合は、環境変数 AMZN_BRAKET_JOB_TOKEN の値をCreateQuantumTask リクエストのjobToken パラメータに渡す必要があります。そうしないと、量子タスクは優先されず、通常のスタンドアロン量子タスクとして請求されます。

ハイパーパラメータの使用方法

ハイブリッドジョブを作成するときに、学習率やステップサイズなど、アルゴリズムに必要なハイパーパラメータを定義できます。ハイパーパラメータ値は通常、アルゴリズムのさまざまな側面を制御するために使用され、多くの場合、アルゴリズムのパフォーマンスを最適化するために調整できま

す。Braket ハイブリッドジョブでハイパーパラメータを使用するには、ハイパーパラメータの名前と値をディクショナリとして明示的に指定する必要があります。最適な値のセットを検索するときにテストするハイパーパラメーター値を指定してください。ハイパーパラメータを使用するための最初のステップは、ハイパーパラメータをディクショナリとして設定して定義することです。このステップは、次のコードで確認できます。

```
from braket.devices import Devices

device_arn = Devices.Amazon.SV1

hyperparameters = {"shots": 1_000}
```

次に、上記のコードスニペットで定義されているハイパーパラメータを、選択したアルゴリズムに渡して使用されるようにします。次のコード例を実行するには、ハイパーパラメータファイルと同じパスに「src」という名前のディレクトリを作成します。「src」ディレクトリ内に、0_Getting_started_papermill.ipynb、notebook_runner.py、および requirements.txt コードファイルを追加します。

```
import time
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    device=device_arn,
    source_module="src",
    entry_point="src.notebook_runner:run_notebook",
    input_data="src/0_Getting_started_papermill.ipynb",
    hyperparameters=hyperparameters,
    job_name=f"papermill-job-demo-{int(time.time())}",
)

# Print job to record the ARN
print(job)
```

ハイブリッドジョブスクリプト内からハイパーパラメータにアクセスするには、notebook_runner.py Python ファイル内の `load_jobs_hyperparams()` 関数を参照してください。ハイブリッドジョブスクリプトの外部からハイパーパラメータにアクセスするには、次のコードを実行します。

```
from braket.aws import AwsQuantumJob

# Get the job using the ARN
```

```
job_arn = "arn:aws:braket:us-east-1:111122223333:job/5eabb790-d3ff-47cc-98ed-
b4025e9e296f"  # Replace with your job ARN
job = AwsQuantumJob(arn=job_arn)

# Access the hyperparameters
job_metadata = job.metadata()
hyperparameters = job_metadata.get("hyperParameters", {})
print(hyperparameters)
```

ハイパーパラメータの使用方法の詳細については、「QAOA with Amazon Braket Hybrid Jobs and PennyLane」および「Quantum machine learning in Amazon Braket Hybrid Jobs」を参照してください。

ハイブリッドジョブインスタンスの設定

アルゴリズムによっては、要件が異なる場合があります。デフォルトでは、Amazon Braket はアルゴリズムスクリプトを`ml.m5.large` インスタンスで実行します。ただし、次のインポートおよび構成引数を使用してハイブリッドジョブを作成する場合は、このインスタンスタイプをカスタマイズできます。

```
from braket.jobs.config import InstanceConfig

job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge"), # Use NVIDIA T4
 instance with 4 GPUs.
    ...
    ),
```

埋め込みシミュレーションを実行する場合に、デバイス設定でローカルデバイスが指定されているときは、`instanceCount` を指定して 1 より大きな値に設定することで、`InstanceConfig` で複数のインスタンスを追加でリクエストできます。上限は 5 です。例えば、次のように 3 つのインスタンスを選択できます。

```
from braket.jobs.config import InstanceConfig
job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge", instanceCount=3), #
 Use 3 NVIDIA T4 instances
    ...
    ),
```

複数のインスタンスを使用する場合は、データ並列機能を使用してハイブリッドジョブを分散することを検討してください。これを表示する方法の詳細については、ノートブックの例である「Parallelize training for QML」を参照してください。

次の 3 つの表は、スタンダードインスタンス、ハイパフォーマンスインスタンス、GPU 高速化インスタンスのぞれぞれで使用可能なインスタンスタイプと仕様の一覧です。

> **Note**
>
> Hybrid Jobs のデフォルトの古典コンピューティングインスタンスクォータを表示するには、「Amazon Braket Quotas」ページを参照してください。

スタンダードインスタンス	vCPU	メモリ (GiB)
ml.m5.large (デフォルト)	4	16
ml.m5.xlarge	4	16
ml.m5.2xlarge	8	32
ml.m5.4xlarge	16	64
ml.m5.12xlarge	48	192
ml.m5.24xlarge	96	384

ハイパフォーマンスインスタンス	vCPU	メモリ (GiB)
ml.c5.xlarge	4	8
ml.c5.2xlarge	8	16
ml.c5.4xlarge	16	32
ml.c5.9xlarge	36	72
ml.c5.18xlarge	72	144

ハイパフォーマンスインスタンス	vCPU	メモリ (GiB)
ml.c5n.xlarge	4	10.5
ml.c5n.2xlarge	8	21
ml.c5n.4xlarge	16	32
ml.c5n.9xlarge	36	72
ml.c5n.18xlarge	72	192

GPU 高速化インスタンス	GPUs	vCPU	メモリ (GiB)	GPU メモリ (GiB)
ml.p4d.24xlarge	8	96	1152	320
ml.g4dn.xlarge	1	4	16	16
ml.g4dn.2xlarge	1	8	32	16
ml.g4dn.4xlarge	1	16	64	16
ml.g4dn.8xlarge	1	32	128	16
ml.g4dn.12xlarge	4	48	192	64
ml.g4dn.16xlarge	1	64	256	16

各インスタンスは、30 GB のデータストレージ (SSD) のデフォルト設定を使用します。ただし、ストレージは、`instanceType` を設定するのと同じ方法で調整できます。次の例では、ストレージの合計を 50 GB に増やす方法を示します。

```
from braket.jobs.config import InstanceConfig

job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(
```

```
        instanceType="ml.g4dn.xlarge",
        volumeSizeInGb=50,
    ),
    ...
    ),
```

AwsSession でのデフォルトのバケットの設定

独自のAwsSession インスタンスを使用すると、カスタマイズしたデフォルトの Amazon S3 バケットの場所を指定する機能など、柔軟性が向上します。デフォルトでは、Amazon S3 バケットの場所 "amazon-braket-{id}-{region}" が AwsSession に事前設定されています。ただし、AwsSession の作成時に、デフォルトの Amazon S3 バケットの場所を上書きしておくこともできます。ユーザーはオプションで AwsSession オブジェクトを AwsQuantumJob.create() メソッドに渡すことができます。それには、次のコード例に示すように aws_session パラメータを指定します。

```
aws_session = AwsSession(default_bucket="amazon-braket-s3-demo-bucket")

# Then you can use that AwsSession when creating a hybrid job
job = AwsQuantumJob.create(
    ...
    aws_session=aws_session
)
```

パラメトリックコンパイルを使用したハイブリッドジョブの高速化

Amazon Braket は、特定の QPU でのパラメトリックコンパイルをサポートしています。このサポートにより、ハイブリッドアルゴリズムの各イテレーションをコンパイルするのではなく、回路を 1 回コンパイルするだけで済むようになるため、計算コストの高いコンパイルステップにかかるオーバーヘッドを削減できます。これにより、各ステップで回路を再コンパイルする必要がなくなるため、ハイブリッドジョブの実行時間が大幅に短縮されます。パラメータ化された回路を Braket ハイブリッドジョブとして、サポートされている QPU のいずれかに送信するだけでよいのです。長時間実行されるハイブリッドジョブの場合、Braket は、回路をコンパイルするときに、ハードウェアプロバイダーからの更新されたキャリブレーションデータを自動的に使用して、最高品質の結果を実現します。

パラメトリック回路を作成するには、まずパラメータをアルゴリズムスクリプトの入力として指定する必要があります。次の例では、小さなパラメトリック回路を使用し、各イテレーション間の古典的

な処理をすべて無視しています。標準的なワークロードなら、多数の回路をバッチで送信し、各イテレーションでパラメータを更新するなどの古典的な処理を実行するところです。

```
import os

from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter

def start_here():

    print("Test job started.")

    # Use the device declared in the job script
    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])

    circuit = Circuit().rx(0, FreeParameter("theta"))
    parameter_list = [0.1, 0.2, 0.3]

    for parameter in parameter_list:
        result = device.run(circuit, shots=1000, inputs={"theta": parameter})

    print("Test job completed.")
```

次のジョブスクリプトを使用して、ハイブリッドジョブとして実行するアルゴリズムスクリプトを送信できます。パラメトリックコンパイルをサポートする QPU でハイブリッドジョブを実行する場合、回路は最初の実行時にのみコンパイルされます。後続の実行では、コンパイルされた回路が再利用されるため、コード行を追加しなくてもハイブリッドジョブの実行時間が短縮されます。

```
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    device=device_arn,
    source_module="algorithm_script.py",
)
```

> **Note**
>
> パラメトリックコンパイルは、Rigetti Computing 製のあらゆる超伝導ゲートベース QPU でサポートされています。ただし、パルスレベルプログラムは例外です。

PennyLane と Amazon Braket の併用

ハイブリッドアルゴリズムは、古典命令と量子命令の両方を含むアルゴリズムです。古典命令は、古典的なハードウェア (EC2 インスタンスまたはノートパソコン) で実行され、量子命令はシミュレーターまたは量子コンピュータ上で実行されます。Hybrid Jobs の機能を使用してハイブリッドアルゴリズムを実行することをお勧めします。詳細については、「Amazon Braket ジョブを使用する時期」を参照してください。

Amazon Braket では、Amazon Braket PennyLane プラグインまたは Amazon Braket Python SDK およびサンプルノートブックリポジトリの助けを借りてハイブリッド量子アルゴリズムを設定および実行できます。SDK に基づく Amazon Braket のサンプルノートブックを使用すると、PennyLane プラグインを使用せずに特定のハイブリッドアルゴリズムを設定および実行できます。しかし、PennyLane を使用した方が、リッチな体験が提供されるため、PennyLane の使用をお勧めします。

ハイブリッド量子アルゴリズムについて

現代の量子コンピューティングデバイスは、一般的にノイズを生成し、このためエラーも生成するため、ハイブリッド量子アルゴリズムは今日の業界にとって重要です。コンピューティングに追加されるすべての量子ゲートにより、ノイズが増える可能性が高くなります。したがって、長時間実行されるアルゴリズムはノイズによって圧倒され、計算エラーが発生する可能性があります。

ショア (Quantum Phase Estimation [量子位相推定] の例) またはグローバー(Grover の例) などの純粋な量子アルゴリズムには、数千または数百万ものオペレーションが必要です。このため、これらは既存の量子デバイスでは実行できない可能性があります。これらの量子デバイスは、一般にノイズの多い中間スケール量子(NISQ) デバイスと呼ばれます。

ハイブリッド量子アルゴリズムでは、特に古典的なアルゴリズムにおける特定の計算を高速化するために、量子処理ユニット (QPU) は古典的 CPU のコプロセッサとして機能します。今日のデバイスの機能によって実現できる範囲において、回路の実行がはるかに短くなります。

このセクションの内容:

- PennyLane と Amazon Braket
- Amazon Braket のハイブリッドアルゴリズムのサンプルノートブック
- PennyLane シミュレーターが埋め込まれたハイブリッドアルゴリズム
- Amazon Braket シミュレーターを使用した PennyLane による随伴勾配
- Hybrid Jobs と PennyLane を使用した QAOA アルゴリズムの実行

- PennyLane 埋め込みシミュレーターを使用したハイブリッドワークロードの実行

PennyLane と Amazon Braket

Amazon Braket は、PennyLane のサポートを提供し、量子微分可能プログラミングのコンセプトを中心に構築されたオープンソースのソフトウェアフレームワークです。このフレームワークは、量子化学、量子機械学習、最適化におけるコンピューティング問題の解を見つけるために、ニューラルネットワークをトレーニングするのと同じ方法で量子回路をトレーニングするのに利用可能です。

PennyLane ライブラリは、PyTorch や TensorFlow など、使い慣れた機械学習ツールへのインターフェースを提供し、量子回路のトレーニングを直感的に簡単に行えます。

- PennyLane Library - PennyLane は Amazon Braket ノートブックに事前にインストールされています。PennyLane から Amazon Braket デバイスにアクセスするには、ノートブックを開き、次のコマンドを使用して PennyLane ライブラリをインポートします。

```
import pennylane as qml
```

チュートリアルノートブックで、すぐに開始できます。または、お好みの IDE から Amazon Braket で PennyLane を使用することもできます。

- Amazon Braket PennyLane プラグイン — 独自の IDE を使用するには、Amazon Braket PennyLane プラグインを手動でインストールできます。プラグインは PennyLane を Amazon Braket Python SDK に接続するので、Amazon Braket デバイスで PennyLane 内の回路を実行できます。PennyLane プラグインをインストールするには、次のコマンドを使用します。

```
pip install amazon-braket-pennylane-plugin
```

以下の例は、PennyLane で Amazon Braket デバイスへのアクセスを設定する方法を示しています。

```
# to use SV1
import pennylane as qml
sv1 = qml.device("braket.aws.qubit", device_arn="arn:aws:braket:::device/quantum-
simulator/amazon/sv1", wires=2)

# to run a circuit:
@qml.qnode(sv1)
def circuit(x):
```

```
    qml.RZ(x, wires=0)
    qml.CNOT(wires=[0,1])
    qml.RY(x, wires=1)
    return qml.expval(qml.PauliZ(1))

result = circuit(0.543)


#To use the local sim:
local = qml.device("braket.local.qubit", wires=2)
```

PennyLane のチュートリアルの例と詳細については、「Amazon Braket examples」リポジトリを参照してください。

Amazon Braket PennyLane プラグインを使用すると、1 行のコードで Amazon Braket QPU と PennyLane のシミュレーターデバイスを切り替えることができます。PennyLane プラグインは、それ自体とともに動作する 2 つの Amazon Braket 量子デバイスを提供しています。

- `braket.aws.qubit`: Amazon Braket サービスの量子デバイス (QPU やシミュレーターを含む) を使用して実行するデバイス
- `braket.local.qubit`: Amazon Braket SDK のローカルシミュレーターで実行するデバイス

Amazon Braket PennyLane プラグインはオープンソースです。インストールは PennyLane Plugin GitHub リポジトリから行えます。

PennyLane の詳細については、PennyLane ウェブサイトのドキュメントを参照してください。

Amazon Braket のハイブリッドアルゴリズムのサンプルノートブック

Amazon Braket では、ハイブリッドアルゴリズムの実行に PennyLane プラグインに依存しないさまざまなサンプルノートブックを提供しています。量子近似最適化アルゴリズム (QAOA) や変分量子固有値ソルバー (VQE) などの変分法を説明する Amazon Braket ハイブリッドサンプルノートブックのいずれからでも開始できます。

Amazon Braket のサンプルノートブックは、Amazon Braket Python SDK に依存しています。SDK は Amazon Braket を通じて量子コンピューティングハードウェアデバイスを操作するためのフレームワークを提供します。これは、ハイブリッドワークフローの量子部分を支援するために設計されたオープンソースライブラリです。

Amazon Braket は、「example のノートブックで詳しく調べることができます。

PennyLane シミュレーターが埋め込まれたハイブリッドアルゴリズム

Amazon Braket Hybrid Jobs に、PennyLane のハイパフォーマンスの CPU ベースおよび GPU ベースの埋め込みシミュレーターが追加されました。この埋め込みシミュレーターファミリーはハイブリッドジョブコンテナに直接埋め込むことができ、高速状態ベクトルの `lightning.qubit` シミュレーターと、NVIDIA の cuQuantum ライブラリを使用して高速化された `lightning.gpu` シミュレーターなどが含まれています。これらの埋め込みシミュレーターは、随伴区別方法などの高度な方法の利点を享受できる量子機械学習などのバリエーションアルゴリズムに最適です。これらの埋め込みシミュレーターは、1 つまたは複数の CPU または GPU インスタンスで実行できます。

Hybrid Jobs により、古典的なコプロセッサと QPU の組み合わせや SV1 などの Amazon Braket オンデマンドシミュレーターを使用したり、PennyLane の埋め込みシミュレーターを直接使用したりすることで、変分アルゴリズムコードを実行できるようになりました。

埋め込みシミュレーターは Hybrid Jobs コンテナでは既に利用可能になっています。利用するには、メインの Python 関数を `@hybrid_job` デコレータでデコレートしてください。PennyLane `lightning.gpu` シミュレーターを使用するには、次のコードスニペットに示すように `InstanceConfig` 内に GPU インスタンスを指定する必要があります。

```
import pennylane as qml
from braket.jobs import hybrid_job
from braket.jobs.config import InstanceConfig


@hybrid_job(device="local:pennylane/lightning.gpu",
 instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge"))
def function(wires):
    dev = qml.device("lightning.gpu", wires=wires)
    ...
```

ハイブリッドジョブで PennyLane 埋め込みシミュレーターの使用を開始するには、「example 」のノートブックを参照してください。

Amazon Braket シミュレーターを使用した PennyLane による随伴勾配

Amazon Braket 用 PennyLane プラグインを使用すると、ローカル状態ベクトルシミュレーターまたは SV1 で勾配計算を実行する際に、随伴微分法を使用できます。

注: 随伴微分法を使用するには、qnode 内に、`diff_method='adjoint'`ではなく `diff_method='device'` を指定する必要があります。次の例を参照してください。

```
device_arn = "arn:aws:braket:::device/quantum-simulator/amazon/sv1"
dev = qml.device("braket.aws.qubit", wires=wires, shots=0, device_arn=device_arn)

@qml.qnode(dev, diff_method="device")
def cost_function(params):
    circuit(params)
    return qml.expval(cost_h)

gradient = qml.grad(circuit)
initial_gradient = gradient(params0)
```

> **Note**
>
> 現行では、PennyLane は QAOA ハミルトニアンのグループ化インデックスを計算し、それらのインデックスを使用してハミルトニアンを複数の期待値に分割します。PennyLane から QAOA を実行するときに、SV1 の随伴微分機能を使用する場合は、次のようにグループ化インデックスを削除してコストハミルトニアンを新規に構築する必要があります: `cost_h, mixer_h = qml.qaoa.max_clique(g, constrained=False) cost_h = qml.Hamiltonian(cost_h.coeffs, cost_h.ops)`

Hybrid Jobs と PennyLane を使用した QAOA アルゴリズムの実行

このセクションでは、学習内容を活用し、パラメトリックコンパイルを指定して PennyLane を使用し、実際のハイブリッドプログラムを作成します。アルゴリズムスクリプトを使用して、量子近似最適化アルゴリズム (QAOA) 問題に対処します。このプログラムでは、古典的な最大カット最適化問題に対応するコスト関数を作成し、パラメータ化された量子回路を指定し、勾配降下法を使用してコスト関数が最小化されるようにパラメータを最適化します。この例では、単純化のためにアルゴリズムスクリプトで問題グラフを生成しますが、より一般的なユースケースでは、入力データ構成の専用チャネルを通じて問題の仕様を提供するのがベストプラクティスです。フラグ `parametrize_differentiable` のデフォルトは `True` であるため、サポートされている QPU でのパラメトリックコンパイルによる実行時間短縮のメリットが自動的に得られます。

```
import os
import json
import time

from braket.jobs import save_job_result
from braket.jobs.metrics import log_metric
```

```
import networkx as nx
import pennylane as qml
from pennylane import numpy as np
from matplotlib import pyplot as plt

def init_pl_device(device_arn, num_nodes, shots, max_parallel):
    return qml.device(
        "braket.aws.qubit",
        device_arn=device_arn,
        wires=num_nodes,
        shots=shots,
        # Set s3_destination_folder=None to output task results to a default folder
        s3_destination_folder=None,
        parallel=True,
        max_parallel=max_parallel,
        parametrize_differentiable=True, # This flag is True by default.
    )

def start_here():
    input_dir = os.environ["AMZN_BRAKET_INPUT_DIR"]
    output_dir = os.environ["AMZN_BRAKET_JOB_RESULTS_DIR"]
    job_name = os.environ["AMZN_BRAKET_JOB_NAME"]
    checkpoint_dir = os.environ["AMZN_BRAKET_CHECKPOINT_DIR"]
    hp_file = os.environ["AMZN_BRAKET_HP_FILE"]
    device_arn = os.environ["AMZN_BRAKET_DEVICE_ARN"]

    # Read the hyperparameters
    with open(hp_file, "r") as f:
        hyperparams = json.load(f)

    p = int(hyperparams["p"])
    seed = int(hyperparams["seed"])
    max_parallel = int(hyperparams["max_parallel"])
    num_iterations = int(hyperparams["num_iterations"])
    stepsize = float(hyperparams["stepsize"])
    shots = int(hyperparams["shots"])

    # Generate random graph
    num_nodes = 6
    num_edges = 8
    graph_seed = 1967
    g = nx.gnm_random_graph(num_nodes, num_edges, seed=graph_seed)
```

```
    # Output figure to file
    positions = nx.spring_layout(g, seed=seed)
    nx.draw(g, with_labels=True, pos=positions, node_size=600)
    plt.savefig(f"{output_dir}/graph.png")

    # Set up the QAOA problem
    cost_h, mixer_h = qml.qaoa.maxcut(g)

    def qaoa_layer(gamma, alpha):
        qml.qaoa.cost_layer(gamma, cost_h)
        qml.qaoa.mixer_layer(alpha, mixer_h)

    def circuit(params, **kwargs):
        for i in range(num_nodes):
            qml.Hadamard(wires=i)
        qml.layer(qaoa_layer, p, params[0], params[1])

    dev = init_pl_device(device_arn, num_nodes, shots, max_parallel)

    np.random.seed(seed)
    cost_function = qml.ExpvalCost(circuit, cost_h, dev, optimize=True)
    params = 0.01 * np.random.uniform(size=[2, p])

    optimizer = qml.GradientDescentOptimizer(stepsize=stepsize)
    print("Optimization start")

    for iteration in range(num_iterations):
        t0 = time.time()

        # Evaluates the cost, then does a gradient step to new params
        params, cost_before = optimizer.step_and_cost(cost_function, params)
        # Convert cost_before to a float so it's easier to handle
        cost_before = float(cost_before)

        t1 = time.time()

        if iteration == 0:
            print("Initial cost:", cost_before)
        else:
            print(f"Cost at step {iteration}:", cost_before)

        # Log the current loss as a metric
        log_metric(
            metric_name="Cost",
```

```
            value=cost_before,
            iteration_number=iteration,
        )

        print(f"Completed iteration {iteration + 1}")
        print(f"Time to complete iteration: {t1 - t0} seconds")

    final_cost = float(cost_function(params))
    log_metric(
        metric_name="Cost",
        value=final_cost,
        iteration_number=num_iterations,
    )

    # We're done with the hybrid job, so save the result.
    # This will be returned in job.result()
    save_job_result({"params": params.numpy().tolist(), "cost": final_cost})
```

> **Note**
>
> パラメトリックコンパイルは、Rigetti Computing 製のあらゆる超伝導ゲートベース QPU でサポートされています。ただし、パルスレベルプログラムは例外です。

PennyLane 埋め込みシミュレーターを使用したハイブリッドワークロードの実行

Amazon Braket Hybrid Jobs で PennyLane の埋め込みシミュレーターを使用してハイブリッドワークロードを実行する方法について見ていきましょう。Pennylane の GPU ベースの埋め込みシミュレーター `lightning.gpu` は、Nvidia cuQuantum ライブラリを使用して回路のシミュレーションを高速化します。埋め込み GPU シミュレーターは、ユーザーがすぐに使用できるすべての Braket ジョブコンテナに事前設定されています。このページでは、`lightning.gpu` を使用してハイブリッドワークロードを高速化する方法を示します。

QAOA ワークロードに **`lightning.gpu`** を使用する

こちらのノートブックにある量子近似最適化アルゴリズム (QAOA) の例について考えてみましょう。埋め込みシミュレーターを選択するには、引数 `device` を `"local:<provider>/<simulator_name>"` という形式の文字列として指定します。例えば、`lightning.gpu` の場合な

ら "local:pennylane/lightning.gpu" と設定します。ハイブリッドジョブの起動時にそれに渡すデバイス文字列は、環境変数 "AMZN_BRAKET_DEVICE_ARN" として渡されます。

```
device_string = os.environ["AMZN_BRAKET_DEVICE_ARN"]
prefix, device_name = device_string.split("/")
device = qml.device(simulator_name, wires=n_wires)
```

このページでは、2 つの埋め込み型 PennyLane 状態ベクトルシミュレーター lightning.qubit (CPU ベース) と lightning.gpu (GPU ベース) を比較します。各シミュレーターにカスタムゲート分解を提供して、さまざまな勾配を計算させます。

これで、ハイブリッドジョブ起動スクリプトを作成する準備ができました。ml.m5.2xlarge と ml.g4dn.xlarge の 2 つのインスタンスタイプを使用して QAOA アルゴリズムを実行します。ml.m5.2xlarge インスタンスタイプは、開発者用の標準的なラップトップに相当します。ml.g4dn.xlarge は、16GB のメモリを持つ単一の NVIDIA T4 GPU を持つ高速コンピューティングインスタンスです。

GPU を実行するには、まず互換性のあるイメージと正しいインスタンス (デフォルトでml.m5.2xlargeはインスタンス) を指定する必要があります。

```
from braket.aws import AwsSession
from braket.jobs.image_uris import Framework, retrieve_image

image_uri = retrieve_image(Framework.PL_PYTORCH, AwsSession().region)
instance_config = InstanceConfig(instanceType="ml.g4dn.xlarge")
```

次に、これらをハイブリッドジョブデコレータに入力し、システム引数とハイブリッドジョブ引数の両方で更新されたデバイスパラメータを入力する必要があります。

```
@hybrid_job(
        device="local:pennylane/lightning.gpu",
        input_data=input_file_path,
        image_uri=image_uri,
        instance_config=instance_config)
def run_qaoa_hybrid_job_gpu(p=1, steps=10):
    params = np.random.rand(2, p)

    braket_task_tracker = Tracker()

    graph = nx.read_adjlist(input_file_path, nodetype=int)
    wires = list(graph.nodes)
```

```
    cost_h, _mixer_h = qaoa.maxcut(graph)

    device_string = os.environ["AMZN_BRAKET_DEVICE_ARN"]
    prefix, device_name = device_string.split("/")
    dev= qml.device(simulator_name, wires=len(wires))
    ...
```

> **Note**
>
> GPU ベースのインスタンスを使用して `instance_config`として指定し、埋め込み CPU ベースのシミュレーター (`lightning.qubit`) `device`として を選択した場合、GPU は使用されません。GPU をターゲットにする場合は、必ず埋め込み 型の GPU ベースシミュレーターを使用してください。

`m5.2xlarge` インスタンスの平均反復時間は約 73 秒で、`ml.g4dn.xlarge`インスタンスの平均反復時間は約 0.6 秒です。この 21 量子ビットのワークフローでは、GPU インスタンスによって 100 倍高速になります。Amazon Braket Hybrid Jobs の料金ページを見ると、`m5.2xlarge`インスタンスの 1 分あたりのコストは 0.00768 USD で、`ml.g4dn.xlarge`インスタンスのコストは 0.01227 USD であることがわかります。この例では、GPU インスタンスで実行する方が高速で安価です。

量子機械学習とデータ並列処理

ワークロードタイプがデータセットでトレーニングする量子機械学習 (QML) である場合は、データ並列処理を使用した方がワークロードを高速化できます。QML では、モデルには 1 つ以上の量子回路が含まれています。モデルには古典ニューラルネットが含まれる場合と含まれない場合があります。データセットを使用してモデルをトレーニングすると、損失関数が最小限に抑えられるようにモデルのパラメータが更新されます。損失関数は通常、単一のデータポイントに対して定義されるのに対し、バッチ平均損失はデータセット全体に対する平均損失として定義されます。QML では、各損失が通常、逐次的に計算されてから、バッチ平均損失が勾配計算用に算出されます。このプロシージャは、特に数百のデータポイントがある場合に時間がかかります。

1 つのデータポイントからの損失は他のデータポイントに依存しないため、複数の損失を並列処理で評価できます。複数のデータポイントに関連する損失と勾配を同時に評価できるのです。これはデータ並列処理と呼ばれます。データ並列処理を使用してトレーニングを高速化しやすくするには、Amazon Braket Hybrid Jobs と SageMaker の分散データ並列ライブラリを使用します。

二項分類の例としてよく知られている UCI リポジトリの ソナーデータセットを使用するデータ並列処理については、以下の QML ワークロードを検討してください。ソナーデータセットには 208 個の

データポイントがあります。各データポイントには、対象物から反射したソナー信号から収集された 60 個の特徴量があります。各データポイントには、機雷の場合は「M」、岩の場合は「R」というラベルが付けられます。当社の QML モデルは、入力レイヤー、非表示レイヤーとしての量子回路、および出力レイヤーで構成されています。入力レイヤーと出力レイヤーは、PyTorch 内に実装されている古典ニューラルネットです。量子回路は、PennyLane の qml.qnn モジュールを使用して PyTorch ニューラルネットと統合されています。ワークロードの詳細については、「example」のノートブックを参照してください。上記の QAOA の例と同様に、PennyLane の `lightning.gpu` などの埋め込み GPU ベースのシミュレーターを使用して GPU の能力を活用することで、埋め込み CPU ベースのシミュレーターよりもパフォーマンスを向上させることができます。

ハイブリッドジョブを作成するには、アルゴリズムスクリプト、デバイス、およびその他の設定をキーワード引数に指定して `AwsQuantumJob.create` を呼び出します。

```
instance_config = InstanceConfig(instanceType='ml.g4dn.xlarge')

hyperparameters={"nwires": "10",
                 "ndata": "32",
                 ...
}

job = AwsQuantumJob.create(
    device="local:pennylane/lightning.gpu",
    source_module="qml_source",
    entry_point="qml_source.train_single",
    hyperparameters=hyperparameters,
    instance_config=instance_config,
    ...
)
```

データ並列処理を使用するには、トレーニングを正しく並列化するように、SageMaker 分散ライブラリのアルゴリズムスクリプトで数行のコードを変更する必要があります。まず、`smdistributed` パッケージをインポートします。このパッケージが、ワークロードを複数の GPU と複数のインスタンスに分散するための力仕事のほとんどを行います。このパッケージは Braket PyTorch コンテナと TensorFlow コンテナに事前設定されています。`dist` モジュールは、トレーニング (`world_size`) の GPU の合計数と GPU コアの `rank` と `local_rank` をアルゴリズムスクリプトに通知します。`rank` はすべてのインスタンスにおける GPU の絶対インデックスで、`local_rank` は特定のインスタンスにおける GPU のインデックスです。例えば、それぞれに 8 つの GPU が割り当てられた 4 つのインスタンスがある場合、`rank` の範囲は 0～31 で、`local_rank` の範囲は 0～7 です。

```
import smdistributed.dataparallel.torch.distributed as dist
```

```
dp_info = {
    "world_size": dist.get_world_size(),
    "rank": dist.get_rank(),
    "local_rank": dist.get_local_rank(),
}
batch_size //= dp_info["world_size"] // 8
batch_size = max(batch_size, 1)
```

次に、world_size と rank に従って DistributedSampler を定義し、それをデータローダーに渡します。次の例のサンプラー (sampler) は、データセットの同じスライスにアクセスする GPU を避けます。

```
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset,
    num_replicas=dp_info["world_size"],
    rank=dp_info["rank"]
)
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=False,
    num_workers=0,
    pin_memory=True,
    sampler=train_sampler,
)
```

次に、DistributedDataParallel クラスを使用してデータ並列処理を有効にします。

```
from smdistributed.dataparallel.torch.parallel.distributed import
 DistributedDataParallel as DDP

model = DressedQNN(qc_dev).to(device)
model = DDP(model)
torch.cuda.set_device(dp_info["local_rank"])
model.cuda(dp_info["local_rank"])
```

上記は、データ並列処理を使用するために必要な変更です。QML では、結果を保存し、トレーニングの進行状況を出力する必要が頻繁に生じます。各 GPU が保存および出力コマンドを実行すると、ログは繰り返しの情報でいっぱいになり、結果は互いに上書きされます。これを回避するには、rank が 0 の GPU においてのみ、保存および出力を行います。

```
if dp_info["rank"]==0:
    print('elapsed time: ', elapsed)
    torch.save(model.state_dict(), f"{output_dir}/test_local.pt")
    save_job_result({"last loss": loss_before})
```

Amazon Braket Hybrid Jobs は、SageMaker 分散データ並列ライブラリの ml.g4dn.12xlarge インスタンスタイプをサポートしています。Hybrid Jobs で InstanceConfig 引数を使用してインスタンスタイプを設定します。SageMaker 分散データ並列ライブラリがデータ並列処理が有効になっていることを知るには、2 つのハイパーパラメータを追加する必要があります ("sagemaker_distributed_dataparallel_enabled" を "true" に設定し、"sagemaker_instance_type" を使用予定のインスタンスタイプに設定)。これら 2 つのハイパーパラメータは smdistributed パッケージで使用されます。これらのハイパーパラメータは、アルゴリズムスクリプトで明示的に使用する必要はありません。Amazon Braket SDK のアルゴリズムスクリプトには、便利なキーワード引数 distribution が備わっているためです。ハイブリッドジョブの作成で distribution="data_parallel" を使用することで、これら 2 つのハイパーパラメータが Amazon Braket SDK により自動的に挿入されるのです。Amazon Braket API を使用する場合は、これら 2 つのハイパーパラメータをユーザーが指定する必要があります。

インスタンスおよびデータ並列処理を設定することで、ハイブリッドジョブを送信できるようになりました。ml.g4dn.12xlarge インスタンスには 4 つの GPUsがあります。instanceCount=1 と設定すると、ワークロードはインスタンス内の 8 つの GPU に分散されます。instanceCount を 1 より大きい値に設定すると、ワークロードはすべてのインスタンスに存在する GPU 間で分散されます。複数のインスタンスを使用する場合は、各インスタンスの使用時間に基づいて料金が発生します。例えば、4 つのインスタンスを使用する場合は、ワークロードを同時に実行しているインスタンスが 4 つあるため、請求対象時間はインスタンスあたりの実行時間の 4 倍になります。

```
instance_config = InstanceConfig(instanceType='ml.g4dn.12xlarge',
                                 instanceCount=1,
)

hyperparameters={"nwires": "10",
                 "ndata": "32",
                 ...,
}

job = AwsQuantumJob.create(
    device="local:pennylane/lightning.gpu",
    source_module="qml_source",
    entry_point="qml_source.train_dp",
    hyperparameters=hyperparameters,
```

```
    instance_config=instance_config,
    distribution="data_parallel",
    ...
)
```

> **Note**
>
> 上記のハイブリッドジョブの作成において、`train_dp.py` はデータ並列処理を使用できるように変更されたアルゴリズムスクリプトです。データ並列処理は、上記のセクションに従ってアルゴリズムスクリプトを変更した場合にのみ正しく動作することに注意してください。アルゴリズムスクリプトを正しく変更せずにデータ並列処理オプションを有効にしても、ハイブリッドジョブがエラーをスローしたり、各 GPU が同じデータスライスを繰り返し処理したりすることがあります。後者は非効率です。

正しく使用すると、複数のインスタンスを使用すると、時間とコストの両方が大幅に削減される可能性があります。詳細については、ノートブックの例を参照してください。

CUDA-Q と Amazon Braket の併用方法

NVIDIA's CUDA-Q は、CPU、GPU、量子処理ユニット (QPU) を組み合わせたハイブリッド量子アルゴリズムをプログラミングするために設計されたソフトウェアライブラリです。これは、開発者が古典命令と量子命令の両方を 1 つのプログラム内で表現できる統合されたプログラミングモデルを提供しているため、ワークフローを効率化できます。CUDA-Q は、埋め込みの CPU および GPU シミュレーターを使用することで、量子プログラムについてシミュレーションの高速化と実行時間の短縮を実現します。CUDA-Q は、ネイティブの Braket ノートブックインスタンス (NBI) と Amazon Braket Hybrid Jobs でご利用いただけます。

このセクションの内容:

- NBI の CUDA-Q
- Hybrid Jobs での CUDA-Q

NBI の CUDA-Q

CUDA-Q は、デフォルトで Braket NBI 環境にインストールされています。CUDA-Q のサンプルノートブックを開くには、Jupyter ランチャーページにアクセスし、[CUDA-Q と Braket タイル] を選択します。これにより、メインウィンドウでサンプルノートブック

0_Getting_started_with_CUDA-Q.ipynb が開きます。その他の CUDA-Q のサンプルについては、左側のパネルにある nvidia_cuda_q/ ディレクトリを参照してください。

また、CUDA-Q のバージョンや、NBI 内にインストールされているその他のサードパーティーパッケージを確認することもできます。例えば、環境にインストールされている CUDA-Q、Qiskit、PennyLane、および Braket パッケージのバージョンを確認するには、ノートブックのコードセルで次のコマンドを実行します。

```
%pip freeze | grep -i -e cudaq -e qiskit -e pennylane -e braket
```

Hybrid Jobs での CUDA-Q

Amazon Braket Hybrid Jobs で CUDA-Q を使用することで、オンデマンドの柔軟なコンピューティング環境が提供されます。計算インスタンスはワークロードの期間中のみ実行されるので、使用した分に対してしか料金が発生しないことが保証されます。また、Amazon Braket Hybrid Jobs はスケーラブルなエクスペリエンスも提供します。ユーザーは、プロトタイプ作成とテストのために小さなインスタンスから始めておいて、後で、完全な実験を行うために、より大きなワークロードを処理できる大きなインスタンスにスケールアップすることもできます。

Amazon Braket Hybrid Jobs は、CUDA-Q の潜在能力を最大限に引き出すために不可欠な GPU をサポートしています。GPU は、CPU ベースのシミュレーターと比較して、量子プログラムのシミュレーションを大幅に高速化します。量子ビット数の大きな回路を使用する場合は特にそうなります。Amazon Braket Hybrid Jobs で CUDA-Q を使用すると、並列化が簡単になります。Hybrid Jobs が、回路のサンプリングとオブザーバブルの評価を複数の計算ノードに簡単に分散できるようにしたためです。この CUDA-Q ワークロードのシームレスな並列化により、ユーザーは大規模な実験の場合に、インフラストラクチャを構築せずにワークロードの開発に専念することができます。

開発を始めるには、Braket が提供する CUDA-Q ハイブリッドジョブコンテナを使用するための、「Amazon Braket examples」Github 内の「CUDA-Q starter example」を参照してください。

次のコードスニペットは、Amazon Braket Hybrid Jobs で CUDA-Q プログラムを実行する hello-world サンプルです。

```
image_uri = retrieve_image(Framework.CUDAQ, AwsSession().region)

@hybrid_job(device='local:nvidia/qpp-cpu', image_uri=image_uri)
def hello_quantum():
    import cudaq
```

```
    # define the backend
    device=get_job_device_arn()
    cudaq.set_target(device.split('/')[-1])

    # define the Bell circuit
    kernel = cudaq.make_kernel()
    qubits = kernel.qalloc(2)
    kernel.h(qubits[0])
    kernel.cx(qubits[0], qubits[1])

    # sample the Bell circuit
    result = cudaq.sample(kernel, shots_count=1000)
    measurement_probabilities = dict(result.items())

    return measurement_probabilities
```

上記の例では、CPU シミュレーターのベル回路をシミュレートしています。この例は、ラップトップまたは Braket Jupyter Notebook でローカルに実行します。このスクリプトを実行すると、local=True という設定により、ローカル環境でコンテナが開始されるため、テストおよびデバッグ用の CUDA-Q プログラムが実行されます。テストが完了したら、local=True フラグを削除することで、 AWSでジョブを実行できます。詳細については、「Amazon Braket Hybrid Jobs の使用方法」を参照してください。

ワークロードに大きな量子ビット数、多数の回路、または多数の反復 (イテレーション) がある場合は、instance_config の設定値を指定することで、よりハイパフォーマンスの CPU コンピューティングリソースを使用できます。次のコードスニペットは、hybrid_job デコレータに instance_config の設定値を設定する方法を示しています。サポートされるインスタンスタイプの詳細については、「ハイブリッドジョブインスタンスの設定」を参照してください。インスタンスタイプのリストについては「Amazon EC2 インスタンスタイプ」を参照してください。

```
@hybrid_job(
    device="local:nvidia/qpp-cpu",
    image_uri=image_uri,
    instance_config=InstanceConfig(instanceType="ml.c5.2xlarge"),
)
def my_job_script():
    ...
```

より要求の厳しいワークロードの場合は、CUDA-Q GPU シミュレーターでワークロードを実行します。GPU シミュレーターを有効にするには、バックエンド名 nvidia を使用します。nvidia バックエンドは CUDA-Q GPU シミュレーターとして機能します。次に、NVIDIA GPU をサポート

する Amazon EC2 インスタンスタイプを選択します。次のコードスニペットは、GPU が設定された hybrid_job デコレータを示しています。

```
@hybrid_job(
    device="local:nvidia/nvidia",
    image_uri=image_uri,
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge"),
)
def my_job_script():
    ...
```

Amazon Braket Hybrid Jobs と NBI は、CUDA-Q を使用した並列 GPU シミュレーションをサポートしています。複数のオブザーバブルまたは複数の回路の評価を並列化することで、ワークロードのパフォーマンスを改善できます。複数のオブザーバブルを並列化するには、アルゴリズムスクリプトに以下の変更を加えます。

nvidia バックエンドの mgpu オプションを設定します。これは、オブザーバブルを並列化するために必要です。並列化は GPU 間の通信に MPI を使用するため、実行前に MPI を初期化しておき、実行後は MPI を確定する必要があります。

次に、execution=cudaq.parallel.mpi と設定することで、実行モードを指定します。以上の変更内容を示したのが、次のコードスニペットです。

```
cudaq.set_target("nvidia", option="mqpu")
cudaq.mpi.initialize()
result = cudaq.observe(
    kernel, hamiltonian, shots_count=n_shots, execution=cudaq.parallel.mpi
)
cudaq.mpi.finalize()
```

hybrid_job デコレータ内で、次のコードスニペットに示すように、複数の GPU をホストするインスタンスタイプを指定します。

```
@hybrid_job(
    device="local:nvidia/nvidia-mqpu",
    instance_config=InstanceConfig(instanceType="ml.g4dn.12xlarge", instanceCount=1),
    image_uri=image_uri,
)
def parallel_observables_gpu_job(sagemaker_mpi_enabled=True):
    ...
```

「Amazon Braket examples」Github の「parallel simulations」ノートブックは、GPU バックエンドで量子プログラムシミュレーションを実行するとともにオブザーバブルと回路バッチの並列シミュレーションを実行する方法を示したエンドツーエンドのサンプルを提供しています。

量子コンピュータでワークロードを実行する

シミュレーターのテストが完了したら、QPU での実験の実行に移ることができます。ターゲットを IQM、IonQ、または Rigetti デバイスなどの Amazon Braket QPU に切り替えるだけでよいのです。次のコードスニペットは、ターゲットを IQM Garnet デバイスに設定する方法を示しています。使用可能な QPU の一覧については、「Amazon Braket console」を参照してください。

```
device_arn = "arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet"
cudaq.set_target("braket", machine=device_arn)
```

ハイブリッドジョブの詳細については、「デベロッパーガイド」の「Amazon Braket Hybrid Jobs の使用方法」を参照してください。CUDA-Q の詳細については、NVIDIA CUDA-Q ドキュメントを参照してください。

Amazon Braket のトラブルシューティング

Amazon Braket の問題を解決するには、このセクションのトラブルシューティング情報と解決策を使用できます。

このセクションの内容:

- AccessDeniedException
- CreateQuantumTask オペレーションを呼び出したときにエラーが発生しました (ValidationException)。
- SDK 機能が動作しません
- ServiceQuotaExceededException が原因でハイブリッドジョブが失敗する
- ノートブックインスタンスでコンポーネントが機能しなくなった場合
- Python 3.12 のアップグレードのトラブルシューティング
- OpenQASM のトラブルシューティング

AccessDeniedException

Braket を有効化または使用しようとすると AccessDeniedException が出力される場合は、制限されたロールがアクセスできないリージョンで Braket を有効化または使用しようとしている可能性があります。

このような場合は、内部 AWS 管理者に連絡して、次のどの条件が適用されるかを確認してください。

- リージョンへのアクセスを禁止するロール制限がある場合。
- 使用しようとしているロールが Braket の使用を許可されている場合。

Braket の使用時にロールが特定のリージョンにアクセスできない場合、その特定のリージョンでデバイスを使用することはできません。

CreateQuantumTask オペレーションを呼び出したときにエラーが発生しました (ValidationException)。

「`An error occurred (ValidationException) when calling the CreateQuantumTask operation: Caller doesn't have access to amazon-braket-…`」というエラーが表示された場合は、既存の s3_folder を参照していることを確認します。Braket は、新しい Amazon S3 バケットとプレフィックスを自動作成しません。

API に直接アクセスしたときに「`Failed to create quantum task: Caller doesn't have access to s3://MY_BUCKET`」というエラーが表示される場合は、Amazon S3 バケットのパスから必ず `s3://` を除外します。

SDK 機能が動作しません

Python バージョンは 3.10 以降である必要があります。Amazon Braket Hybrid Jobs では、Python 3.12 をお勧めします。

SDK およびスキーマが最新であることを確認します。ノートブックまたは Python エディタから SDK を更新するには、以下のコマンドを実行します。

```
pip install amazon-braket-sdk --upgrade --upgrade-strategy eager
```

スキーマを更新するには、次のコマンドを実行します。

```
pip install amazon-braket-schemas --upgrade
```

自身のクライアントから Amazon Braket にアクセスする場合は、AWS リージョンが Amazon Braket でサポートされているリージョンに設定されていることを確認してください。

ServiceQuotaExceededException が原因でハイブリッドジョブが失敗する

ターゲットとするシミュレーターデバイスの同時量子タスク制限を超えると、Amazon Braket シミュレーターに対して量子タスクを実行するハイブリッドジョブが作成されないことがあります。サービス制限の詳細については、「クォータ」のトピックを参照してください。

このエラーは、アカウントからシミュレーターデバイスに対する同時タスクを複数のハイブリッドジョブで実行している場合に発生することがあります。

特定のシミュレーターデバイスに対する同時量子タスクの数を確認するには、次のコード例に示すように、search-quantum-tasks API を使用します。

```
DEVICE_ARN=arn:aws:braket:::device/quantum-simulator/amazon/sv1
task_list=""
for status_value in "CREATED" "QUEUED" "RUNNING" "CANCELLING"; do
    tasks=$(aws braket search-quantum-tasks --filters
 name=status,operator=EQUAL,values=${status_value}
 name=deviceArn,operator=EQUAL,values=$DEVICE_ARN --max-results 100 --query
 'quantumTasks[*].quantumTaskArn' --output text)
    task_list="$task_list $tasks"
done;
echo "$task_list" | tr -s ' \t' '[\n*]' | sort | uniq
```

Braket > デバイス別の順に操作することで、デバイスに対して作成されたタスクを Amazon CloudWatch メトリクスで表示することもできます。

これらのエラーが発生しないようにするには、以下を実行します。

1. シミュレーターデバイスの同時量子タスク数のサービスクォータの増加を要求します。この設定は SV1 デバイスにのみ適用されます。
2. コード内の ServiceQuotaExceeded の例外を処理し、再試行してください。

ノートブックインスタンスでコンポーネントが機能しなくなった場合

ノートブックの一部のコンポーネントが機能しなくなった場合は、以下を試してください。

1. 作成または変更したノートブックをローカルドライブにダウンロードします。
2. ノートブックインスタンスを停止します。
3. ノートブックインスタンスを削除します。
4. 別の名前で新しいノートブックインスタンスを作成します。
5. ノートブックを新しいインスタンスにアップロードします。

Python 3.12 のアップグレードのトラブルシューティング

発効日: 2026 年 1 月 21 日

概要:

2026 年 1 月 21 日以降、Amazon Braket はすべてのノートブックインスタンスとマネージドコンテナイメージ (Base、CUDA-Q、TensorFlow、PyTorch) の Python ランタイムを 3.10 から 3.12 にアップグレードします。このガイドでは、一般的な互換性の問題の解決策を示します。

このセクションの内容:

- 一般的なエラーメッセージ
- Braket マネージドノートブック
- ハイブリッドジョブデコレータ
- Bring-Your-Own-Container (BYOC)
- Braket ノートブックインスタンスのアップグレード

一般的なエラーメッセージ

SDK Python バージョンの不一致エラー

エラー:

```
RuntimeError: Python version must match between local environment and container. Client
 is running Python 3.10 locally, but container uses Python 3.12.
```

原因: Braket SDK はノートブックが Python 3.10 を実行していることを検出しましたが、ハイブリッドジョブコンテナは Python 3.12 を実行しています。

解決策: ノートブックを Python 3.12 にアップグレードするか、Python 3.10 コンテナに固定します。 ???

Cloudpickle シリアル化エラー

エラー:

```
TypeError: code() argument 13 must be str, not int
```

原因: SDK 検証をバイパスすると、Python 3.12 の CodeType コンストラクタの変更により、cloudpickle は Python 3.10 と 3.12 の間でコードをシリアル化できません。

解決策: ノートブックとコンテナが同じ Python バージョンを使用していることを確認します。

Braket マネージドノートブック

Python 3.10 で Braket ノートブックインスタンスを実行し、ハイブリッドジョブを送信すると、ジョブコンテナがデフォルトで Python 3.12 を使用するようになったため、バージョン不一致エラーが発生します。

これには 2 つのオプションがあります。

1. [推奨] Python 3.12 で新しいノートブックインスタンスを作成する - Braket Notebook Instance Upgrade を参照してください。
2. Python 3.10 コンテナへのピン留め - 「ハイブリッドジョブデコレータ」を参照してください。

ハイブリッドジョブデコレータ

@hybrid_job デコレータを使用するには、環境の Python バージョンがコンテナの Python バージョンと一致する必要があります。

オプション 1: Python 3.12 コンテナを使用する (推奨)

環境を Python 3.12 にアップグレードした場合、最新のタグ (デフォルトの動作) が使用されます。

オプション 2: Python 3.10 コンテナを使用する

Python 3.10 を維持する必要がある場合は、デ`@hybrid_job`コレータで `image_uri`パラメータを明示的に指定します。

Python 3.10 コンテナタグ:

イメージ名	タグ
Base	1.0-cpu-py310-ubuntu22.04
CUDA-Q	0.12.0-cpu-py310-0.12.0
PyTorch	2.2.0-gpu-py310-cu121-ubuntu20.04
TensorFlow	2.14.1-gpu-py310-cu118-ubuntu20.04

次の例は、us-west-2 リージョンの場合です。

フルイメージ URIs:

```
Base:       292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:1.0-cpu-py310-ubuntu22.04
CUDA-Q:     292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:0.12.0-cpu-py310-0.12.0
PyTorch:    292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:2.2.0-gpu-py310-cu121-ubuntu20.04
TensorFlow: 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-tensorflow-jobs:2.14.1-gpu-py310-cu118-ubuntu20.04
```

例:

```
from braket.jobs.hybrid_job import hybrid_job
from braket.devices import Devices

device_arn = Devices.Amazon.SV1

@hybrid_job(
    device=device_arn,
    image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:1.0-cpu-py310-ubuntu22.04"
)
def my_job():
    pass
```

> **Note**
>
> - Python 3.10 コンテナは引き続き使用できますが、更新は受信されません。
> - 「アルゴリズムスクリプトの環境を定義する」を参照してください。

Bring-Your-Own-Container (BYOC)

Dockerfile が最新のタグを持つ Braket マネージドイメージを使用している場合、2026 年 1 月 21 日以降に再構築すると、Python 3.12 でサポートされているイメージがプルされます。

Python 3.10 でサポートされている Braket マネージドイメージを維持するには、Dockerfile を更新します。

Before (アップグレード後に Python 3.12 を取得):

```
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:latest
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:latest
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-tensorflow-jobs:latest
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:latest
```

After (Python 3.10 で保存):

```
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:1.0-cpu-
py310-ubuntu22.04
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:0.12.0-cpu-
py310-0.12.0
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:2.2.0-gpu-
py310-cu121-ubuntu20.04
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-tensorflow-jobs:2.14.1-
gpu-py310-cu118-ubuntu20.04
```

Braket ノートブックインスタンスのアップグレード

Python 3.12 にアップグレードするには、次の手順に従います。

> **Important**
>
> ノートブックインスタンスを削除する前に、保持するすべてのノートブックとファイルをダウンロードしていることを確認してください。削除後にこのデータを復元することはできません。

1. 作成または変更したノートブックをローカルドライブにダウンロードします。
2. ノートブックインスタンスを停止します。
3. ノートブックインスタンスを削除します。
4. 別の名前で新しいノートブックインスタンスを作成します。
5. ノートブックを新しいインスタンスにアップロードします。

OpenQASM のトラブルシューティング

このセクションでは、OpenQASM 3.0 を使用している際にエラーが発生した場合に役立つトラブルシューティングのヒントについて説明します。

このセクションの内容:

- インクルードステートメントのエラー
- 連続しない qubits のエラー
- 物理 qubits と仮想 qubits を混在させていることを示すエラー
- 同一プログラムで結果タイプをリクエストするとともに qubits を測定することによるエラー
- Classical and qubit register limits exceeded error
- ボックスの前に逐語的なプラグマがないことを示すエラー
- 逐語的なボックスにネイティブゲートがないことを示すエラー
- 逐語的なボックスに物理 qubits がないことを示すエラー
- 逐語的なプラグマに「braket」がないことを示すエラー
- 単一 qubits にはインデックスを作成できないことを示すエラー
- 2 つの qubit ゲートの物理 qubits が接続されていないことを示すエラー
- LocalSimulator のサポートに関する警告

インクルードステートメントのエラー

Braket には現在、OpenQASM プログラムにインクルードできる標準ゲートライブラリファイルはありません。例えば、次の例ではパーサーエラーが発生しています。

```
OPENQASM 3;
include "standardlib.inc";
```

このコードは以下のエラーメッセージを生成します: `No terminal matches '"' in the current parser context, at line 2 col 17.`

連続しない qubits のエラー

デバイス機能で `requiresContiguousQubitIndices` が `true` に設定されているデバイスで、隣接していない qubits を使用すると、エラーが発生します。

シミュレーターと IonQ で量子タスクを実行しているときに、次のプログラムを実行すると、エラーがトリガーされます。

```
OPENQASM 3;
```

```
qubit[4] q;

h q[0];
cnot q[0], q[2];
cnot q[0], q[3];
```

このコードは以下のエラーメッセージを生成します: `Device requires contiguous qubits. Qubit register q has unused qubits q[1], q[4].`

物理 qubits と仮想 qubits を混在させていることを示すエラー

同じプログラムで物理 qubits と仮想 qubits を混在させることは許可されないため、エラーとなります。次のコードはエラーを生成します。

```
OPENQASM 3;

qubit[2] q;
cnot q[0], $1;
```

このコードは以下のエラーメッセージを生成します: `[line 4] mixes physical qubits and qubits registers.`

同一プログラムで結果タイプをリクエストするとともに qubits を測定することによるエラー

同一プログラムで結果タイプをリクエストするとともに qubits を明示的に測定すると、エラーが発生します。次のコードはエラーを生成します:

```
OPENQASM 3;

qubit[2] q;

h q[0];
cnot q[0], q[1];
measure q;

#pragma braket result expectation x(q[0]) @ z(q[1])
```

このコードは以下のエラーメッセージを生成します: `Qubits should not be explicitly measured when result types are requested.`

Classical and qubit register limits exceeded error

1 つの古典レジスタと 1 つの qubit レジスタのみが許可されます。次のコードはエラーを生成します。

```
OPENQASM 3;

qubit[2] q0;
qubit[2] q1;
```

このコードは以下のエラーメッセージを生成します: `[line 4] cannot declare a qubit register. Only 1 qubit register is supported.`

ボックスの前に逐語的なプラグマがないことを示すエラー

どのボックスの前にも、逐語的なプラグマを付ける必要があります。次のコードはエラーを生成します。

```
box{
rx(0.5) $0;
}
```

このコードは以下のエラーメッセージを生成します: `In verbatim boxes, native gates are required. x is not a device native gate.`

逐語的なボックスにネイティブゲートがないことを示すエラー

逐語的なボックスには、ネイティブゲートと物理 qubits が必要です。次のコードはネイティブゲートエラーを生成します。

```
#pragma braket verbatim
box{
x $0;
}
```

このコードは以下のエラーメッセージを生成します: `In verbatim boxes, native gates are required. x is not a device native gate.`

逐語的なボックスに物理 qubits がないことを示すエラー

逐語的なボックスには物理的な qubits が必要です。次のコードは、物理 qubits がないことを示すエラーを生成します。

```
qubit[2] q;

#pragma braket verbatim
box{
rx(0.1) q[0];
}
```

このコードは以下のエラーメッセージを生成します: `Physical qubits are required in verbatim box.`

逐語的なプラグマに「braket」がないことを示すエラー

逐語的なプラグマには「braket」を含める必要があります。次のコードはエラーを生成します:

```
#pragma braket verbatim          // Correct
#pragma verbatim                 // wrong
```

このコードは以下のエラーメッセージを生成します: `You must include “braket” in the verbatim pragma`

単一 qubits にはインデックスを作成できないことを示すエラー

単一 qubits にはインデックスを作成できません。次のコードはエラーを生成します:

```
OPENQASM 3;

qubit q;
h q[0];
```

このコードは以下のエラーを生成します: `[line 4] single qubit cannot be indexed.`

ただし、単一 qubit 配列には、次のようにしてインデックスを作成できます。

```
OPENQASM 3;
```

```
qubit[1] q;
h q[0];   // This is valid
```

2 つの qubit ゲートの物理 qubits が接続されていないことを示すエラー

物理 qubits を使用するには、まず `device.properties.action[DeviceActionType.OPENQASM].supportPhysicalQubits` をチェックすることでデバイスが物理 qubits を使用していることを確認し、次に `device.properties.paradigm.connectivity.connectivityGraph` または `device.properties.paradigm.connectivity.fullyConnected` をチェックして接続グラフを確認します。

```
OPENQASM 3;

cnot $0, $14;
```

このコードは以下のエラーメッセージを生成します: `[line 3] has disconnected qubits 0 and 14`

LocalSimulator のサポートに関する警告

`LocalSimulator` は、QPU またはオンデマンドシミュレーターでは利用できない OpenQASM の高度な機能をサポートしています。次の例に示すように、プログラムに、`LocalSimulator` にのみ固有の言語機能が含まれている場合は、警告が表示されます。

```
qasm_string = """
qubit[2] q;

h q[0];
ctrl @ x q[0], q[1];
"""
qasm_program = Program(source=qasm_string)
```

このコードは、「This program uses OpenQASM language features only supported in the LocalSimulator」という警告を生成します。これらの機能の一部は、QPU またはオンデマンドシミュレーターではサポートされていない可能性があります。

サポートされている OpenQASM 機能の詳細については、「Local Simulator での OpenQASM の高度な機能のサポート」ページを参照してください。

Amazon Braket のセキュリティ

のクラウドセキュリティが最優先事項 AWS です。お客様は AWS 、セキュリティを最も重視する組織の要件を満たすように構築されたデータセンターとネットワークアーキテクチャからメリットを得られます。

セキュリティは、 AWS お客様とお客様の間の責任共有です。責任共有モデルではこれをクラウドのセキュリティおよびクラウド内のセキュリティと説明しています。

- クラウドのセキュリティ – AWS は、 で AWS サービスを実行するインフラストラクチャを保護する責任を担います AWS クラウド。 は、お客様が安全に使用できるサービス AWS も提供します。サードパーティーの監査者は、AWS コンプライアンスプログラムコンプライアンスプログラムの一環として、当社のセキュリティの有効性を定期的にテストおよび検証。Amazon Braket に適用されるコンプライアンスプログラムの詳細については、「コンプライアンスプログラムAWS による対象範囲内のサービスコンプライアンスプログラム」を参照してください。
- クラウド内のセキュリティ – お客様の責任は、使用する AWS サービスによって決まります。また、お客様は、お客様のデータの機密性、企業の要件、および適用可能な法律および規制などの他の要因についても責任を担います。

このドキュメントは、Braket を使用する際に責任共有モデルを適用する方法を理解するのに役立ちます。セキュリティおよびコンプライアンスの目的を達成するために Braket を設定する方法を以下の各トピックで示します。また、Braket リソースのモニタリングや保護に役立つ他の AWS サービスの使用方法についても説明します。

このセクションの内容:

- セキュリティの責任共有
- データ保護
- データ保持
- Amazon Braket へのアクセスを管理する
- Amazon Braket のサービスリンクロール
- Amazon Braket のコンプライアンス検証
- Amazon Braket でのインフラストラクチャセキュリティ
- Amazon Braket ハードウェアプロバイダーのセキュリティ
- Amazon Braket 用の Amazon VPC エンドポイント

セキュリティの責任共有

セキュリティは、 AWS お客様とお客様の間の責任共有です。責任共有モデルでは、これをクラウドのセキュリティおよびクラウド内のセキュリティとして説明しています。

- クラウドのセキュリティ – AWS は、 AWS のサービス で実行されるインフラストラクチャを保護する責任を担います AWS クラウド。 は、安全に使用できるサービス AWS も提供します。サードパーティーの監査人は、AWS コンプライアンスプログラム の一環として、セキュリティの有効性を定期的にテストおよび検証します。Amazon Braket に適用されるコンプライアンスプログラムについては、「AWS コンプライアンスプログラムによる対象範囲内のサービス」を参照してください。
- クラウド内のセキュリティ – この AWS インフラストラクチャでホストされているコンテンツに対する制御を維持するのはお客様の責任です。このコンテンツには、 AWS のサービス 使用する のセキュリティ設定および管理タスクが含まれます。

データ保護

責任 AWS 共有モデル、Amazon Braket でのデータ保護に適用されます。このモデルで説明されているように、 AWS はすべての を実行するグローバルインフラストラクチャを保護する責任があります AWS クラウド。ユーザーは、このインフラストラクチャでホストされるコンテンツに対する管理を維持する責任があります。また、使用する「 AWS のサービス 」のセキュリティ設定と管理タスクもユーザーの責任となります。データプライバシーの詳細については、データプライバシーに関するよくある質問を参照してください。欧州でのデータ保護の詳細については、AWS セキュリティブログに投稿された「AWS 責任共有モデルおよび GDPR」のブログ記事を参照してください。

データ保護の目的で、認証情報を保護し AWS アカウント 、 AWS IAM アイデンティティセンター または AWS Identity and Access Management (IAM) を使用して個々のユーザーを設定することをお勧めします。この方法により、それぞれのジョブを遂行するために必要な権限のみが各ユーザーに付与されます。また、次の方法でデータを保護することもお勧めします:

- 各アカウントで多要素認証 (MFA) を使用します。
- SSL/TLS を使用して AWS リソースと通信します。TLS 1.2 は必須ですが、TLS 1.3 を推奨します。
- で API とユーザーアクティビティのログ記録を設定します AWS CloudTrail。CloudTrail 証跡を使用して AWS アクティビティをキャプチャする方法については、「 AWS CloudTrail ユーザーガイド」のCloudTrail 証跡の使用」を参照してください。

- AWS 暗号化ソリューションと、その中のすべてのデフォルトのセキュリティコントロールを使用します AWS のサービス。
- Amazon Macie などの高度な管理されたセキュリティサービスを使用します。これらは、Amazon S3 に保存されている機密データの検出と保護を支援します。
- コマンドラインインターフェイスまたは API AWS を介して にアクセスするときに FIPS 140-3 検証済み暗号化モジュールが必要な場合は、FIPS エンドポイントを使用します。利用可能な FIPS エンドポイントの詳細については、「連邦情報処理規格 (FIPS) 140-3」を参照してください。

お客様の E メールアドレスなどの極秘または機密情報を、タグ、または [名前] フィールドなどの自由形式のテキストフィールドに含めないことを強くお勧めします。これは、コンソール、API、または SDK を使用して Amazon Braket AWS CLIまたは他の AWS のサービス を使用する場合も同様です。 AWS SDKs タグ、または名前に使用される自由記述のテキストフィールドに入力したデータは、請求または診断ログに使用される場合があります。外部サーバーに URL を提供する場合、そのサーバーへのリクエストを検証できるように、認証情報を URL に含めないことを強くお勧めします。

データ保持

90 日後、Amazon Braket は、量子タスクに関連付けられているすべての量子タスク ID およびその他のメタデータを自動的に削除します。このデータ保持ポリシーの結果として、これらのタスクと結果は S3 バケットに保存されたままですが、Amazon Braket コンソールからの検索では取得できなくなります。

S3 バケットに 90 日以上保存される履歴量子タスクと結果にアクセスする必要がある場合は、タスク ID とそのデータに関連するその他のメタデータの個別のレコードを保持する必要があります。必ず 90 日前までに情報を保存してください。その保存された情報を使用して、履歴データを取得できます。

Amazon Braket へのアクセスを管理する

この章では、Amazon Braket の実行、または特定のユーザーおよびロールのアクセスを制限するために必要なアクセス許可について説明します。アカウント内の任意のユーザーまたはロールに必要なアクセス許可を付与 (または拒否) できます。これを行うには、後続のセクションで説明するように、アカウント内の該当するユーザーまたはロールに適切な Amazon Braket ポリシーをアタッチします。

前提条件として、Amazon Braket を有効にします。Braket を有効にするには (1) 管理者権限を持っているか、(2) AmazonBraketFullAccess ポリシーが割り当てられ、Amazon Simple Storage Service (Amazon S3) バケットを作成する権限を持っているユーザーまたはロールとしてサインインしてください。

このセクションの内容:

- Amazon Braket のリソース
- ノートブックとロール
- AWS Amazon Braket の マネージドポリシー
- 特定のデバイスへのユーザーアクセスを制限する
- 特定のノートブックインスタンスへのユーザーアクセスを制限する
- 特定の S3 バケットへのユーザーアクセスを制限する

Amazon Braket のリソース

Braket は、特定タイプのリソース、つまり量子タスクリソースを作成します。この AWS リソースタイプのリソースネーム (ARN) は次のとおりです。

- リソース名: AWS::Service::Braket
- ARN 正規表現: arn:${Partition}:braket:${Region}:${Account}:quantum-task/${RandomId}

ノートブックとロール

Braket ではノートブックリソースタイプを使用できます。ノートブックは Amazon SageMaker AI リソースであり、Braket が共有できます。Braket でノートブックを使用するには、`AmazonBraketServiceSageMakerNotebook` で始まる名前が付いた IAM ロールを指定する必要があります。

ノートブックを作成するには、管理者権限を持つロールを使用するか、次のインラインポリシーがアタッチされているロールを使用する必要があります。

JSON

```
{
  "Version":"2012-10-17",
```

```
  "Statement": [
    {
      "Sid": "CreateTheRole",
      "Effect": "Allow",
      "Action": "iam:CreateRole",
      "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole*"
    },
    {
      "Sid": "CreateThePolicy",
      "Effect": "Allow",
      "Action": "iam:CreatePolicy",
      "Resource": [
        "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookAccess*",
        "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookRole*"
      ]
    },
    {
      "Sid": "AttachTheRolePolicy",
      "Effect": "Allow",
      "Action": "iam:AttachRolePolicy",
      "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole*",
      "Condition": {
        "ArnLike": {
          "iam:PolicyARN": [
            "arn:aws:iam::aws:policy/AmazonBraketFullAccess",
            "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookAccess*",
            "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookRole*"
          ]
        }
      }
    }
  ]
}
```

ロールを作成するには、「ノートブックを作成する」ページに記載されている手順に従うか、管理者に作成してもらいます。AmazonBraketFullAccess ポリシーがロールに添付されていることを確認します。

ロールを作成したら、今後起動するすべてのノートブックでそのロールを再利用できます。

AWS Amazon Braket の マネージドポリシー

AWS 管理ポリシーは、 によって作成および管理されるスタンドアロンポリシーです AWS。 AWS 管理ポリシーは、ユーザー、グループ、ロールにアクセス許可の割り当てを開始できるように、多くの一般的なユースケースにアクセス許可を付与するように設計されています。

AWS 管理ポリシーは、すべての AWS お客様が使用できるため、特定のユースケースに対して最小特権のアクセス許可を付与しない場合があることに注意してください。ユースケースに固有のカスタマー管理ポリシーを定義して、アクセス許可を絞り込むことをお勧めします。

AWS 管理ポリシーで定義されているアクセス許可は変更できません。が AWS マネージドポリシーで定義されたアクセス許可 AWS を更新すると、ポリシーがアタッチされているすべてのプリンシパル ID (ユーザー、グループ、ロール) に影響します。 AWS は、新しい が起動されるか、新しい API オペレーション AWS のサービス が既存のサービスで使用できるようになったときに、 AWS マネージドポリシーを更新する可能性が高くなります。

詳細については、「IAM ユーザーガイド」の「AWS マネージドポリシー」を参照してください。

トピック

- AWS マネージドポリシー: AmazonBraketFullAccess
- AWS マネージドポリシー: AmazonBraketJobsExecutionPolicy
- AWS マネージドポリシー: AmazonBraketServiceRolePolicy
- AWS マネージドポリシーに対する Amazon Braket の更新

AWS マネージドポリシー: AmazonBraketFullAccess

AmazonBraketFullAccess ポリシーは、Amazon Braket オペレーションの権限を付与します。これには、次のタスクに対するアクセス権限が含まれます。

- Amazon Elastic Container Registry からコンテナをダウンロードする – Amazon Braket Hybrid Jobs 機能に使用されるコンテナイメージを読み取ってダウンロードします。コンテナは「arn:aws:ecr:::repository/amazon-braket」の形式に準拠している必要があります。

- AWS CloudTrail ログを保持する – クエリの開始と停止、メトリクスフィルターのテスト、ログイベントのフィルタリングに加えて、すべての説明、取得、および一覧表示アクション。 AWS CloudTrail ログファイルには、アカウントで発生したすべての Amazon Braket APIアクティビティの記録が含まれています。
- ロールを活用してリソースを制御する — サービスにリンクされたロールをアカウントに作成します。サービスにリンクされたロールは、ユーザーに代わって AWS リソースにアクセスできます。Amazon Braket サービスでのみ使用できます。また、IAM ロールを Amazon Braket `CreateJob` API に渡したり、ロールを作成して AmazonBraketFullAccess にスコープ設定されたポリシーをロールにアタッチしたりします。
- アカウントの使用状況ログファイルを管理するために、ロググループとログイベントを作成し、ロググループ内をクエリする – アカウントの Amazon Braket 使用状況に関するログ情報を作成、保存、および表示します。ハイブリッドジョブロググループのメトリクスをクエリします。適切な Braket パスを含んでおり、ログデータの配置を可能にします。CloudWatch にメトリクスデータを配置します。
- Amazon S3 バケットにデータを作成して保存し、すべてのバケットを一覧表示する — S3 バケットを作成したり、アカウント内の S3 バケットを一覧表示したり、名前が amazon-braket- で始まる、アカウント内の任意のバケットとの間でオブジェクトを出し入れしたりします。これらの権限は、Braket により、処理された量子タスクの結果を含むファイルをバケットとの間で出し入れできるようにするために必要です。
- IAM ロールを渡す – IAM ロールを `CreateJob` API に渡します。
- Amazon SageMaker AI ノートブック - 「arn:aws:sagemaker:::notebook-instance/amazon-braket-」からリソースの範囲に含まれる SageMaker ノートブックインスタンスを作成および管理します。
- サービスクォータを検証する – SageMaker AI ノートブックと Amazon Braket ハイブリッドジョブを作成するには、リソース数がアカウントのクォータを超えることができません。
- 製品の料金を表示する – ワークロードを送信する前に、量子ハードウェアコストを確認して計画します。

このポリシーのアクセス許可は、「AWS マネージドポリシーリファレンス」の「AmazonBraketFullAccess」で確認してください。

AWS マネージドポリシー: AmazonBraketJobsExecutionPolicy

AmazonBraketJobsExecutionPolicy ポリシーは、Amazon Braket Hybrid Jobs で使用される以下の実行ロールの権限を付与します。

- Amazon Elastic Container Registry からコンテナをダウンロードする – Amazon Braket Hybrid Jobs 機能に使用されるコンテナイメージを読み取ってダウンロードする権限。コンテナは「arn:aws:ecr:*:*:repository/amazon-braket*」の形式に準拠している必要があります。
- アカウントの使用状況ログファイルを管理するために、ロググループとログイベントを作成し、ロググループ内をクエリする – アカウントの Amazon Braket 使用状況に関するログ情報を作成、保存、および表示します。ハイブリッドジョブロググループのメトリクスをクエリします。適切な Braket パスを含んでおり、ログデータの配置を可能にします。CloudWatch にメトリクスデータを配置します。
- Amazon S3 バケットにデータを保存する — アカウント内の S3 バケットを一覧表示し、名前が amazon-braket- で始まるアカウント内の任意のバケットとの間でオブジェクトを出し入れします。これらの権限は、Braket により、処理された量子タスクの結果を含むファイルをバケットとの間で出し入れできるようにするために必要です。
- IAM ロールを渡す - IAM ロールを CreateJob API に渡します。ロールは arn:aws:iam::*:role/service-role/AmazonBraketJobsExecutionRole* の形式に従っている必要があります。

このポリシーのアクセス許可を表示するには、「AWS マネージドポリシーリファレンス」の「AmazonBraketJobsExecutionPolicy」を参照してください。

AWS マネージドポリシー: AmazonBraketServiceRolePolicy

AmazonBraketServiceRolePolicy ポリシーは、Amazon Braket オペレーションの権限を付与します。これには、次のタスクに対するアクセス権限が含まれます。

- Amazon S3 — アカウント内のバケットを一覧表示し、`amazon-braket-` で始まる名前で任意のバケットとの間でオブジェクトを出し入れする権限。
- Amazon CloudWatch Logs — ロググループを作成、一覧表示したり、関連するログストリームを作成したり、Amazon Braket 用に作成されたロググループにイベントを配置したりする権限。

サービスにリンクされたロールの詳細については、「Amazon Braket サービスリンクロール」を参照してください。

このポリシーのアクセス許可を表示するには、「AWS マネージドポリシーリファレンス」の「AmazonBraketServiceRolePolicy」を参照してください。

AWS マネージドポリシーに対する Amazon Braket の更新

次の表は、このサービスがこれらの変更の追跡を開始した時点からの Amazon Braket の AWS マネージドポリシーの更新に関する詳細を示しています。

変更	説明	日付
AmazonBraketServiceRolePolicy - リソース管理ポリシー	Amazon S3 および CloudWatch のログアクションに "aws:ResourceAccount" : "${aws:PrincipalAccount}" 条件スコープが追加されました。	2025 年 7 月 11 日
AmazonBraketFullAccess - Braket のフルアクセスポリシー	"pricing:GetProducts" アクションが追加されました。	2025 年 4 月 14 日
AmazonBraketFullAccess - Braket のフルアクセスポリシー	S3 のアクションに "aws:ResourceAccount": "${aws:PrincipalAccount}" 条件スコープが追加されました。	2025 年 3 月 7 日
AmazonBraketFullAccess - Braket のフルアクセスポリシー	servicequotas:GetServiceQuota および cloudwatch:GetMetricData アクションが追加されました。	2023 年 3 月 24 日
AmazonBraketFullAccess - Braket のフルアクセスポリシー	使用されている Amazon S3 バケットを表示および検査するための s3:ListAllMyBuckets アクセス許可が追加されました。	2022 年 3 月 31 日
AmazonBraketFullAccess - Braket のフルアクセスポリシー	Braket は、`service-role/` パスを含めるように AmazonBraketFullAccess の iam:PassRole アクセス許可を変更しました。	2021 年 11 月 29 日
AmazonBraketJobsExecutionPolicy - Amazon Braket Hybrid Jobs のハイブリッドジョブ実行ポリシー	Braket で、`service-role/` パスを含めるようにハイブリッ	2021 年 11 月 29 日

変更	説明	日付
	ドジョブ実行ロール ARN が更新されました。	
Braket、変更の追跡を開始	Braket は AWS 、管理ポリシーの変更の追跡を開始しました。	2021 年 11 月 29 日

特定のデバイスへのユーザーアクセスを制限する

ユーザーのアクセスを特定の Braket デバイスに制限するために、アクセス許可を拒否するポリシーを特定の IAM ロールに追加することができます。

以下のアクションを制限できます。

- `CreateQuantumTask` - 指定したデバイスでの量子タスクの作成を拒否します。
- `CreateJob` - 指定したデバイスでのハイブリッドジョブの作成を拒否します。
- `GetDevice` - 指定したデバイスの詳細が取得されることを拒否します。

以下は、 AWS アカウント `123456789012` に対し、すべての QPU へのアクセスを制限する例です。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
    "Effect": "Deny",
    "Action": [
      "braket:CreateQuantumTask",
      "braket:CreateJob",
      "braket:GetDevice"
    ],
    "Resource": [
    "arn:aws:braket:*:*:device/qpu/*"
    ],
    "Condition": {
             "StringEquals": {
```

```
                "aws:PrincipalAccount": "123456789012"
            }
        }
    }
  ]
}
```

> **Note**
>
> デバイスの可用性、キャリブレーションデータ、料金などのデバイスプロパティへのユーザーの読み取りアクセスを Braket コンソールで有効にするには、ポリシーから `braket:GetDevice` アクションを除外します。

このコードを適応させるには、前の例で示した文字列を、制限付きデバイスの Amazon リソース番号 (ARN) に置き換えます。この文字列は、リソース値を指定します。Braket では、デバイスは量子タスクを実行するために呼び出すことができる QPU またはシミュレーターを表します。使用可能なデバイスは、デバイスページに一覧表示されます。これらのデバイスへのアクセスを指定するために使用するスキーマは 2 つあります。

- `arn:aws:braket:<region>:*:device/qpu/<provider>/<device_id>`
- `arn:aws:braket:<region>:*:device/quantum-simulator/<provider>/<device_id>`

さまざまなタイプのデバイスアクセスの例を次に示します。

- すべてのリージョンですべての QPU を選択する: `arn:aws:braket:*:*:device/qpu/*`
- us-west-2 リージョンだけのすべての QPU を選択する: `arn:aws:braket:us-west-2:*:device/qpu/*`
- us-west-2 リージョンだけのすべての QPU を選択する (ただし、デバイスがカスタマーリソースではなくサービスリソースである場合): `arn:aws:braket:us-west-2:*:device/qpu/*`
- すべてのオンデマンドシミュレーターデバイスへのアクセスを制限する: `arn:aws:braket:*:*:device/quantum-simulator/*`
- 特定のプロバイダーからのデバイス (例えば Rigetti QPU デバイスなど) へのアクセスを制限する: `arn:aws:braket:*:*:device/qpu/rigetti/*`
- TN1 デバイスへのアクセスを制限する: `arn:aws:braket:*:*:device/quantum-simulator/amazon/tn1`

- すべての Create アクションへのアクセスを制限する: braket:Create*

特定のノートブックインスタンスへのユーザーアクセスを制限する

特定のユーザーのアクセスを特定の Braket ノートブックインスタンスに制限するには、特定のロール、ユーザー、またはグループに、「アクセス許可を拒否する」ポリシーを追加します。

次の例では、ポリシー変数を使用して、 内の特定のノートブックインスタンスを開始、停止、アクセスするアクセス許可を効率的に制限します。このインスタンスは AWS アカウント 123456789012、アクセス許可を持つ必要があるユーザー (たとえば、ユーザーは という名前のノートブックインスタンスにアクセスできますamazon-braket-Alice) に従って名前が付けられAliceます。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "DenyCreateDeleteUpdateNotebookInstances",
      "Effect": "Deny",
      "Action": [
        "sagemaker:CreateNotebookInstance",
        "sagemaker:DeleteNotebookInstance",
        "sagemaker:UpdateNotebookInstance",
        "sagemaker:CreateNotebookInstanceLifecycleConfig",
        "sagemaker:DeleteNotebookInstanceLifecycleConfig",
        "sagemaker:UpdateNotebookInstanceLifecycleConfig"
      ],
      "Resource": "*"
    },
    {
      "Sid": "DenyDescribeStartStopNotebookInstances",
      "Effect": "Deny",
      "Action": [
        "sagemaker:DescribeNotebookInstance",
        "sagemaker:StartNotebookInstance",
        "sagemaker:StopNotebookInstance"
      ],
      "NotResource": [
```

```
        "arn:aws:sagemaker:*:123456789012:notebook-instance/amazon-braket-
${aws:username}"
      ]
    },
    {
      "Sid": "DenyNotebookInstanceUrl",
      "Effect": "Deny",
      "Action": [
        "sagemaker:CreatePresignedNotebookInstanceUrl"
      ],
      "NotResource": [
        "arn:aws:sagemaker:*:123456789012:notebook-instance/amazon-braket-
${aws:username}*"
      ]
    }
  ]
}
```

特定の S3 バケットへのユーザーアクセスを制限する

特定のユーザーのアクセスを特定の Amazon S3 バケットに制限するには、特定のロール、ユーザー、またはグループに拒否ポリシーを追加します。

次の例では、オブジェクトを取得して特定の S3 バケット (`arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice`) に配置するアクセス許可を制限するだけでなく、それらのオブジェクトを一覧表示することも制限しています。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "s3:ListBucket"
      ],
      "NotResource": [
        "arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice"
      ]
    },
```

```
    {
      "Effect": "Deny",
      "Action": [
        "s3:GetObject"
      ],
      "NotResource": [
        "arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice/*"
      ]
    }
  ]
}
```

特定のノートブックインスタンスのバケットへのアクセスを制限するには、前述のポリシーをノートブック実行ロールに追加します。

Amazon Braket のサービスリンクロール

Amazon Braket を有効にすると、サービスリンクロールがアカウント内に作成されます。

サービスリンクロールは、この場合、Amazon Braket に直接リンクされた特殊なタイプの IAM ロールです。Amazon Braket のサービスリンクロールは、ユーザーに代わって他の AWS のサービスを呼び出すときに、Braket が必要とするすべてのアクセス許可を含むように事前定義されています。

必要な許可を手動で追加する必要がないため、サービスリンクロールは Amazon Braket のセットアップを容易にします。Amazon Braket は、サービスリンクロールのアクセス許可を定義します。これらの定義を変更しない限り Amazon Braketのみがロールを引き受けることができます。定義されたアクセス許可には、信頼ポリシーとアクセス許可ポリシーが含まれます。アクセス許可ポリシーを他の IAM エンティティにアタッチすることはできません。

Amazon Braket が設定するサービスリンクロールは、AWS Identity and Access Management (IAM) サービスリンクロール機能の一部です。サービスリンクロールをサポートする他のAWS のサービスについては、「IAM と連携する AWS のサービス」を参照し、「サービスリンクロール」列が「はい」になっているサービスを探してください。サービスに対し、サービスリンクロールに関するドキュメントを表示するには、[はい] のリンクを選択します。

サービスリンクロールの AWS マネージドポリシーについては、
「AmazonBraketServiceRolePolicy」を参照してください。

Amazon Braket のコンプライアンス検証

> **Note**
>
> AWS コンプライアンスレポートは、独自の独立した監査に目を通すことができるサードパーティーのハードウェアプロバイダーからの QPU をカバーしていません。

AWS のサービス が特定のコンプライアンスプログラムの対象であるかどうかを確認するには、「コンプライアンスプログラムによる対象範囲内の AWS のサービス」をご覧いただき、関心のあるコンプライアンスプログラムを選択してください。一般的な情報については、「AWSコンプライアンスプログラム」を参照してください。

AWS Artifact を使用して、サードパーティーの監査レポートをダウンロードできます。詳細については、「AWS Artifact でレポートをダウンロードする」を参照してください。

AWS のサービス を使用する際のお客様のコンプライアンス責任は、お客様のデータの機密性や貴社のコンプライアンス目的、適用可能な法律および規制によって決定されます。AWS のサービスを使用する際のコンプライアンス責任の詳細については、「AWS セキュリティドキュメント」を参照してください。

Amazon Braket でのインフラストラクチャセキュリティ

マネージドサービスである Amazon Braket は AWS グローバルネットワークセキュリティで保護されています。AWS セキュリティサービスと AWS がインフラストラクチャを保護する方法については「AWS クラウドセキュリティ」を参照してください。インフラストラクチャセキュリティのベストプラクティスを使用して AWS 環境を設計するには「セキュリティの柱 - AWS 適切なアーキテクチャを備えたフレームワーク」の「インフラストラクチャの保護」を参照してください。

AWS が公開している API コールを使用し、ネットワーク経由で Amazon Braket にアクセスします。クライアントは以下をサポートする必要があります。

- Transport Layer Security (TLS)。TLS 1.2 が必須ですが、TLS 1.3 をお勧めします。
- DHE (楕円ディフィー・ヘルマン鍵共有) や ECDHE (楕円曲線ディフィー・ヘルマン鍵共有) などの完全前方秘匿性 (PFS) による暗号スイート。これらのモードは Java 7 以降など、ほとんどの最新システムでサポートされています。

これらの API オペレーションは任意のネットワークの場所から呼び出すことができますが、Braket ではリソースベースのアクセスポリシーがサポートされているため、ソース IP アドレスに基づく制限を含めることができます。また、Braket ポリシーを使用して、特定の Amazon Virtual Private Cloud (Amazon VPC) エンドポイントまたは特定の VPC からのアクセスを制御することもできます。これにより、実質的に AWS ネットワーク内の特定の VPC からの特定の Braket リソースへのネットワークアクセスが分離されます。

Amazon Braket ハードウェアプロバイダーのセキュリティ

Amazon Braket の QPU は、サードパーティーのハードウェアプロバイダーによってホストされています。QPU で量子タスクを実行すると、Amazon Braket は、処理のために指定された QPU に回路を送信するときに DeviceARN を識別子として使用します。

サードパーティーハードウェアプロバイダーの 1 つが運営する量子コンピューティングハードウェアへのアクセスに Amazon Braket を使用する場合、お客様の回路とその関連データは、AWS が運営する施設外のハードウェアプロバイダーによって処理されます。各 QPU が利用できる物理的な場所と AWS リージョンに関する情報は、Amazon Braket コンソールの [デバイスの詳細] セクションにあります。

コンテンツは匿名化されています。回路の処理に必要なコンテンツのみがサードパーティーに送信されます。 AWS アカウントの情報がサードパーティーに送信されることはありません。

すべてのデータは、保管時と転送時のいずれも暗号化されます。データは処理のためだけに復号されます。Amazon Braket サードパーティープロバイダーは、お客様の回路の処理以外の目的でお客様のコンテンツを保存または使用することは許可されていません。回路の完了後、結果は Amazon Braket に返され、S3 バケットに保存されます。

Amazon Braket サードパーティーの量子ハードウェアプロバイダーのセキュリティは、ネットワークセキュリティ、アクセスコントロール、データ保護、および物理的セキュリティの基準が満たされていることを確認するために、定期的に監査されます。

Amazon Braket 用の Amazon VPC エンドポイント

VPC と Amazon Braket とのプライベート接続を確立するには、インターフェイス VPC エンドポイントを作成します。インターフェイスエンドポイントは、インターネットゲートウェイAWS PrivateLink、NAT デバイス、VPN 接続、または Direct Connect 接続なしで Braket APIs へのアクセスを可能にするテクノロジーである を利用しています。VPC のインスタンスは、パブリック IP アドレスがなくても Braket API と通信できます。

各インターフェースエンドポイントは、サブネット内の 1 つ以上の Elastic Network Interface によって表されます。

では AWS PrivateLink、VPC と Braket 間のトラフィックがAmazonネットワークを離れないため、クラウドベースのアプリケーションと共有するデータのセキュリティが向上します。これは、データがパブリックインターネットに公開される可能性を減らすためです。詳細については、「Amazon VPC ユーザーガイド」の「インターフェイス VPC エンドポイントを使用して AWS サービスにアクセスする」を参照してください。

このセクションの内容:

- Amazon Braket VPC エンドポイントに関する考慮事項
- Braket と PrivateLink を設定する
- エンドポイントの作成に関する追加情報
- Amazon VPC エンドポイントポリシーによるアクセスのコントロール

Amazon Braket VPC エンドポイントに関する考慮事項

Braket 用のインターフェイス VPC エンドポイントを設定する前に、「Amazon VPC ユーザーガイド」の「Interface endpoint prerequisites」を確認してください。

Braket は、VPC からのすべての API アクションの呼び出しをサポートしています。

デフォルトでは、VPC エンドポイントを通じた Braket へのフルアクセスが許可されています。VPC エンドポイントポリシーを指定すれば、アクセスをコントロールできます。詳細については、「Amazon VPC ユーザーガイド」の「エンドポイントポリシーを使用して VPC エンドポイントへのアクセスを制御する」を参照してください。

Braket と PrivateLink を設定する

Amazon Braket AWS PrivateLink で を使用するには、Amazon Virtual Private Cloud (Amazon VPC) エンドポイントをインターフェイスとして作成し、Amazon Braket APIサービスを介してエンドポイントに接続する必要があります。

ここでは、このプロセスの一般的なステップを示します。これについては、後のセクションで詳しく説明します。

- AWS リソースをホストするように Amazon VPC を設定して起動します。VPC が既にある場合、このステップは省略できます。

- Braket 用の Amazon VPC エンドポイントを作成します
- エンドポイント経由で Braket 量子タスクを接続して実行します

ステップ 1: 必要に応じて Amazon VPC を起動する

アカウントに既に VPC が運用されている場合は、このステップを省略できることにご注意ください。

VPC は、IP アドレス範囲、サブネット、ルートテーブル、ネットワークゲートウェイなどのネットワーク設定をコントロールできます。基本的に、カスタム仮想ネットワークで AWS リソースを起動します。VPC の詳細については、「Amazon VPC ユーザーガイド」を参照してください。

Amazon VPC コンソールを開いて、サブネット、セキュリティグループ、ネットワークゲートウェイを含む新しい VPC を作成します。

ステップ 2: Braket のインターフェイス VPC エンドポイントの作成

Braket サービスの VPC エンドポイントは、Amazon VPC コンソールまたは AWS Command Line Interface () を使用して作成できますAWS CLI。詳細については、「Amazon VPC ユーザーガイド」の「VPC エンドポイントの作成」を参照してください。

コンソールで VPC エンドポイントを作成するには、Amazon VPC コンソールを開き、エンドポイントページを開いて、新しいエンドポイントの作成に進みます。後で参照できるように、エンドポイント ID を書きとめます。この ID は、Braket API に特定の呼び出しを行うときに –endpoint-url フラグの一部として必要になります。

Braket 用の VPC エンドポイントを作成するには、次のサービス名を使用します。

- `com.amazonaws.substitute_your_region.braket`

詳細については、「Amazon VPC ユーザーガイド」の「インターフェイス VPC エンドポイントを使用して AWS サービスにアクセスする」を参照してください。

ステップ 3: エンドポイント経由で Braket 量子タスクを接続して実行する

VPC エンドポイントを作成すると、 次の例のように、API またはランタイムへのインターフェイスエンドポイントを指定する `endpoint-url` パラメータを含む CLI コマンドを使用できます。

```
aws braket search-quantum-tasks --endpoint-url
 VPC_Endpoint_ID.braket.substituteYourRegionHere.vpce.amazonaws.com
```

VPC エンドポイントのプライベート DNS ホスト名を有効にした場合は、CLI コマンドで URL をエンドポイントとして指定する必要はありません。代わりに、CLI および Braket SDK がデフォルトで使用する Amazon Braket API DNS ホスト名は、ユーザーの VPC エンドポイントに解決されます。その形式は、次の例のようにします。

```
https://braket.substituteYourRegionHere.amazonaws.com
```

AWS PrivateLink エンドポイントを使用して Amazon VPC から Amazon SageMaker AI ノートブックへの直接アクセスというブログ記事では、AmazonBraket ノートブックに似た SageMaker ノートブックへの安全な接続を行うためにエンドポイントを設定する方法の例を示しています。

ブログ投稿の手順に従う場合は、Amazon SageMaker AI の代わりに Amazon Braket という名前を使用することを忘れないでください。サービス名 リージョンが us-east-1 でない場合は、その文字列に正しい AWS リージョン 名前を入力するcom.amazonaws.us-east-1.braketか、置き換えます。

エンドポイントの作成に関する追加情報

- VPC をプライベートサブネットで作成する方法については、「プライベートサブネットを持つ VPC を作成する」を参照してください。
- Amazon VPC コンソールまたは を使用してエンドポイントを作成および設定する方法については AWS CLI、「Amazon VPC ユーザーガイド」の「VPC エンドポイントを作成する」を参照してください。
- を使用してエンドポイントを作成および設定する方法については CloudFormation、CloudFormation ユーザーガイドのAWS「::EC2::VPCEndpoint リソース」を参照してください。

Amazon VPC エンドポイントポリシーによるアクセスのコントロール

Amazon Braket への接続アクセスをコントロールするには、Amazon VPC エンドポイントに AWS Identity and Access Management (IAM) エンドポイントポリシーをアタッチしてください。このポリシーでは、以下の情報を指定します。

- アクションを実行できるプリンシパル (ユーザーまたはロール)
- 実行可能なアクション。
- アクションを実行できるリソース。

詳細については、「Amazon VPC ユーザーガイド」の「エンドポイントポリシーを使用して VPC エンドポイントへのアクセスを制御する」を参照してください。

例: Braket アクションの VPC エンドポイントポリシー

Braket のエンドポイントポリシーの例を以下に示します。このポリシーは、エンドポイントに添付されると、すべてのリソースのすべてのプリンシパルに対して、登録されている Braket アクションへのアクセスを許可します。

```
{
 "Statement":[
 {
   "Principal":"*",
   "Effect":"Allow",
   "Action":[
     “braket:action-1",
     “braket:action-2",
     “braket:action-3”
     ],
   "Resource":"*"
   }
  ]
}
```

複数のエンドポイントポリシーを添付することで、複雑な IAM ルールを作成できます。詳細な説明と例については、以下を参照してください。

- Step Functions の Amazon Virtual Private Cloud エンドポイントポリシー
- 管理者以外のユーザー用の詳細な IAM アクセス許可の作成
- エンドポイントポリシーを使用して VPC エンドポイントへのアクセスを制御する

ログ記録とモニタリング

Amazon Braket サービスを使用して量子タスクを送信すると、Amazon Braket SDK およびコンソールを通じてそのタスクのステータスと進行状況を詳しくモニタリングできます。このようなモニタリングができるため、ワークロードの実行の追跡、潜在的なボトルネックや問題の発見、量子アプリケーションのパフォーマンスと信頼性を最適化するための適切なアクションの実行を 1 箇所で管理できます。Braket は、量子タスクを完了すると、結果を指定された Amazon S3 の場所に保存します。量子タスクの実行時間は変動します。特に、量子タスクを量子処理装置 (QPU) デバイスで実行する場合は、実行時間が大きく変動します。これは主に実行キューの長さによるものです。複数のユーザーが量子ハードウェアリソースを共用するためです。

ステータスタイプのリスト:

- `CREATED` – Amazon Braket が量子タスクを受け取った状態。
- `QUEUED` – Amazon Braket が量子タスクを処理済みで、量子タスクがデバイスで実行されるのを待機している状態。
- `RUNNING` – 量子タスクが QPU またはオンデマンドシミュレーターで実行されている状態。
- `COMPLETED` – 量子タスクが QPU またはオンデマンドシミュレーターで実行し終わった状態。
- `FAILED` – 量子タスクの実行が試みられたが、失敗した状態。量子タスクが失敗した理由によっては、量子タスクを再度送信してみてください。
- `CANCELLED` – 量子タスクがキャンセルされた状態。量子タスクは実行されなかった。

このセクションの内容:

- Amazon Braket SDK を使用して量子タスクを追跡する
- Amazon Braket コンソールを使用した量子タスクのモニタリング
- Amazon Braket リソースのタグ付け
- EventBridge を使用した量子タスクのモニタリング
- CloudWatch によるメトリクスのモニタリング
- CloudTrail を使用した量子タスクのログ記録
- Amazon Braket を使用した高度なログ記録

Amazon Braket SDK を使用して量子タスクを追跡する

コマンド device.run(…) は、一意の量子タスク ID を使用してタスクを定義するものです。次の例に示すように、task.state() を使用してステータスを照会および追跡できます。

注意: task = device.run() は非同期操作です。つまり、システムがバックグラウンドで量子タスクを処理している間も、ユーザーは作業を続けることができます。

結果を取得する

task.result() を呼び出すと、SDK は Amazon Braket のポーリングを開始し、量子タスクが完了しているかどうかを確認します。SDK は、.run() で定義したポーリングパラメータを使用します。量子タスクが完了すると、SDK は S3 バケットから結果を取得し、それを QuantumTaskResult オブジェクトとして返します。

```
# create a circuit, specify the device and run the circuit
circ = Circuit().rx(0, 0.15).ry(1, 0.2).cnot(0,2)
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
task = device.run(circ, s3_location, shots=1000)

# get ID and status of submitted task
task_id = task.id
status = task.state()
print('ID of task:', task_id)
print('Status of task:', status)
# wait for job to complete
while status != 'COMPLETED':
    status = task.state()
    print('Status:', status)
```

```
ID of task:
arn:aws:braket:us-west-2:123412341234:quantum-task/b68ae94b-1547-4d1d-aa92-1500b82c300d
Status of task: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: RUNNING
Status: RUNNING
```

```
Status: COMPLETED
```

量子タスクをキャンセルする

量子タスクをキャンセルするには、次の例に示すように、cancel() メソッドを呼び出します。

```
# cancel quantum task
task.cancel()
status = task.state()
print('Status of task:', status)
```

```
Status of task: CANCELLING
```

メタデータを確認する

完了した量子タスクのメタデータを確認できます。次に例を示します。

```
# get the metadata of the quantum task
metadata = task.metadata()
# example of metadata
shots = metadata['shots']
date = metadata['ResponseMetadata']['HTTPHeaders']['date']
# print example metadata
print("{} shots taken on {}.".format(shots, date))

# print name of the s3 bucket where the result is saved
results_bucket = metadata['outputS3Bucket']
print('Bucket where results are stored:', results_bucket)
# print the s3 object key (folder name)
results_object_key = metadata['outputS3Directory']
print('S3 object key:', results_object_key)

# the entire look-up string of the saved result data
look_up = 's3://'+results_bucket+'/'+results_object_key
print('S3 URI:', look_up)
```

```
1000 shots taken on Wed, 05 Aug 2020 14:44:22 GMT.
Bucket where results are stored: amazon-braket-123412341234
S3 object key: simulation-output/b68ae94b-1547-4d1d-aa92-1500b82c300d
S3 URI: s3://amazon-braket-123412341234/simulation-output/b68ae94b-1547-4d1d-
aa92-1500b82c300d
```

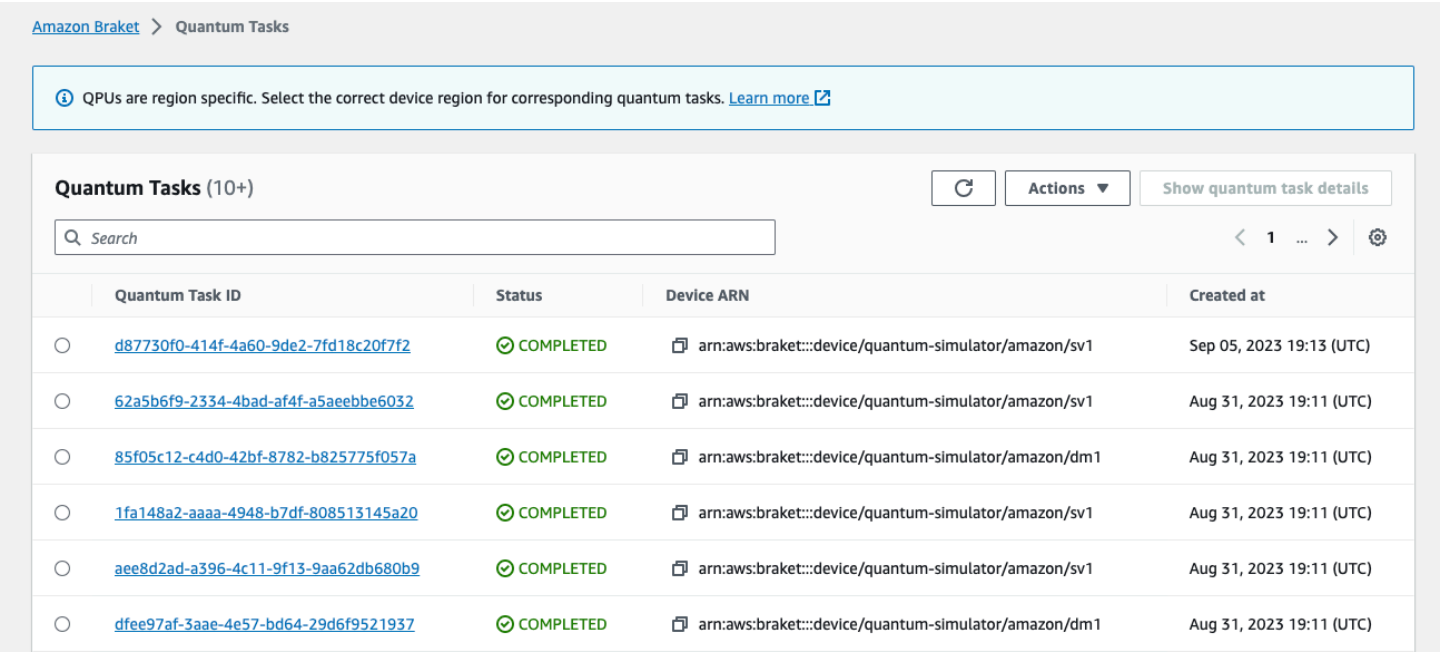
量子タスクまたは結果を取得する

量子タスクを送信した後にカーネルが停止した場合、またはノートブックまたはコンピュータを閉じた場合は、一意の ARN (量子タスク ID) を使用して `task` オブジェクトを再構築します。次に `task.result()` を呼び出して、保存されている S3 バケットから結果を取得します。

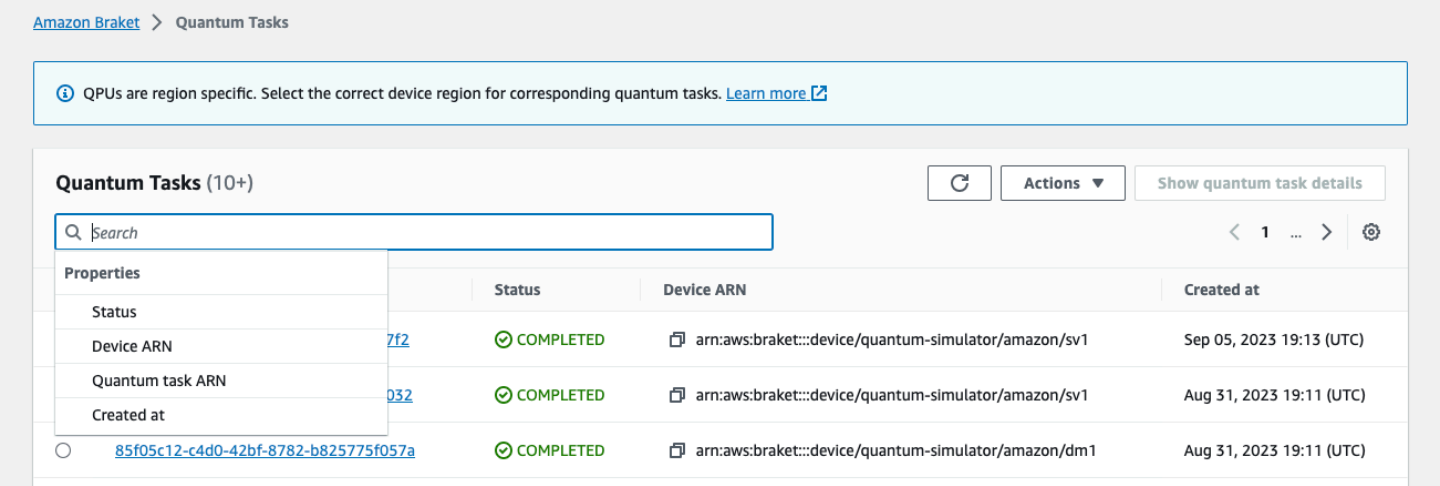
```
from braket.aws import AwsSession, AwsQuantumTask

# restore task with unique arn
task_load = AwsQuantumTask(arn=task_id)
# retrieve the result of the task
result = task_load.result()
```

Amazon Braket コンソールを使用した量子タスクのモニタリング

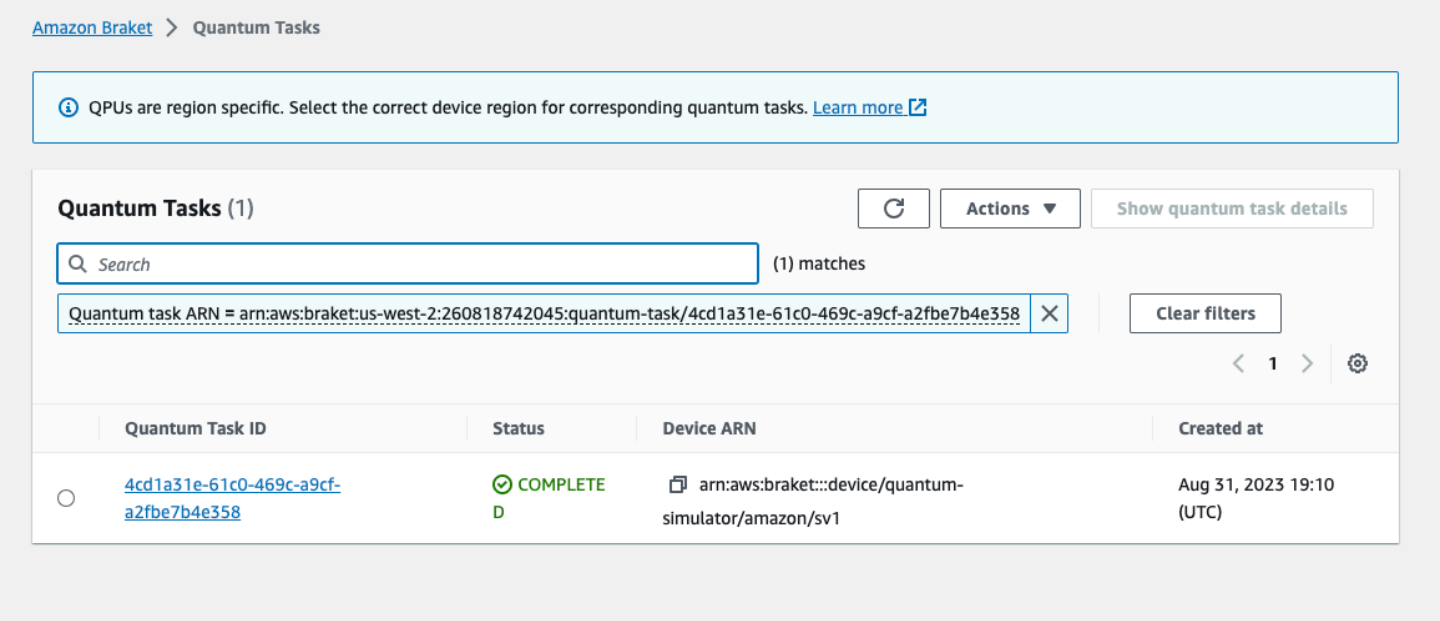
Amazon Braket では、Amazon Braket コンソールを使用して簡単に量子タスクをモニタリングできます。送信されたすべての量子タスクが [量子タスク] フィールドに一覧表示されます。その例を次の図に示します。このサービスはリージョン固有です。つまり、特定の で作成された量子タスクのみを表示できます AWS リージョン。



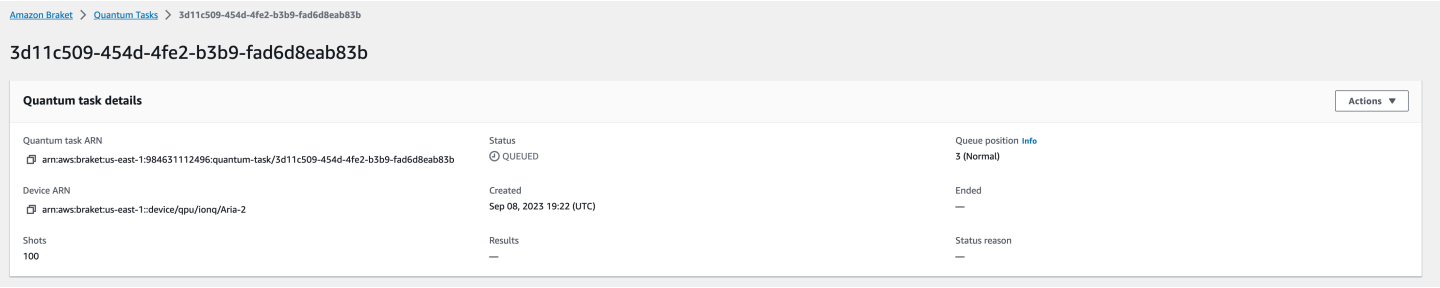
特定の量子タスクを検索するには、ナビゲーションバーを使用します。検索基準は、量子タスク ARN (ID)、ステータス、デバイス、および作成時刻です。これらのオプションは、次の例に示すように、ナビゲーションバーを選択すると自動的に表示されます。



次の画像は、一意のタスク ID に基づいて量子タスクを検索する例です。この ID を取得するには、`task.id` を呼び出します。



また、次の図に示すように、量子タスクのステータスは QUEUED 状態の間にモニタリングできます。量子タスク ID をクリックすると、詳細ページが表示されます。このページには、量子タスクが処理されるデバイスに関連する、量子タスクの動的キュー位置が表示されます。



ハイブリッドジョブの一部として送信される量子タスクは、キューに入れられると高い優先度を設定されます。ハイブリッドジョブに含まれずに送信される量子タスクは、キュー内で通常の優先度になります。

Braket SDK でのクエリを希望するカスタマーは、量子タスクとハイブリッドジョブキュー位置をプログラムで取得できます。詳細については、「タスクはいつ実行されますか」ページを参照してください。

Amazon Braket リソースのタグ付け

タグは、 AWS リソースに割り当てるカスタム属性ラベル AWS です。タグはリソースについて詳しく説明するメタデータです。各タグは、キー と 値 から構成されます。これらは共にキーと値のペアと呼ばれます。割り当てるタグには、キーと値を定義します。

Amazon Braket コンソールでは、量子タスクまたはノートブックに移動して、それに関連付けられているタグのリストを表示できます。タグの追加、タグの削除、またはタグの修正を行うことができます。量子タスクまたはノートブックの作成時にタグを付け、コンソール、 AWS CLI、または を使用して関連するタグを管理できますAPI。

AWS および タグの詳細

- 命名規則や使用規則など、タグ付けに関する一般的な情報については、「タグ付けリソースとタグエディタユーザーガイド」の「タグエディタとは」を参照してください。 AWS
- タグ付けの制限については、「 リソースのタグ付けとタグエディタユーザーガイド」の「タグの命名制限と要件」を参照してください。 AWS
- ベストプラクティスとタグ付け戦略については、「AWS リソースのタグ付けのベストプラクティス」を参照してください。
- タグの使用をサポートするサービスのリストについては、「リResource Groups のタグ付け API リファレンス」を参照してください。

以下のセクションでは、Amazon Braket のタグに関する詳細が説明されています。

このセクションの内容:

- タグの使用
- Amazon Braket でタグ付けがサポートされるリソース
- Amazon Braket API を使用したタグ付け

- タグ指定の制限
- Amazon Braket でタグを管理する
- Amazon Braket での AWS CLI タグ付けの例

タグの使用

タグを使用すると、リソースを自分に役立つカテゴリに整理できます。例えば、「部門」タグを割り当てて、このリソースを所有する部門を指定できます。

各 タグは 2 つの部分で構成されます:

- タグキー (CostCenter、Environment、Project など)。タグキーでは大文字と小文字が区別されます。
- タグ値 (111122223333 や Production など) として知られるオプションのフィールド。タグ値を省略すると、空の文字列を使用した場合と同じになります。タグキーと同様に、タグ値では大文字と小文字が区別されます。

タグは、以下のことに役立ちます。

- AWS リソースを特定して整理します。多くの はタグ付け AWS のサービス をサポートしているため、異なるサービスのリソースに同じタグを割り当てて、リソースが関連していることを示すことができます。
- AWS コストを追跡します。 AWS Billing and Cost Management ダッシュボードでこれらのタグをアクティブ化します。 はタグ AWS を使用してコストを分類し、毎月のコスト配分レポートを配信します。詳細については、「AWS Billing and Cost Management ユーザーガイド」の「コスト配分タグを使用する」を参照してください。
- AWS リソースへのアクセスを制御します。詳細については、「タグを使用したアクセス制御」を参照してください。

Amazon Braket でタグ付けがサポートされるリソース

Amazon Braket の次のリソースタイプは、タグ付けをサポートしています。

- `quantum-task` リソース:
- リソース名: `AWS::Service::Braket`

- ARN 正規表現: `arn:${Partition}:braket:${Region}:${Account}:quantum-task/${RandomId}`

注意: Amazon Braket コンソールで Amazon Braket ノートブックのタグを適用および管理するには、Amazon Braket コンソールを使用してノートブックリソースに移動します。ただし、ノートブックは実際には Amazon SageMaker AI リソースです。詳細については、SageMaker のドキュメントの「ノートブックインスタンスのメタデータ」を参照してください。

Amazon Braket API を使用したタグ付け

- Amazon Braket API を使用してリソースにタグを設定する場合は、TagResourceAPI を呼び出します。

```
aws braket tag-resource --resource-arn $YOUR_TASK_ARN --tags {\"city\": \"Seattle\"}
```

- リソースからタグを削除するには、UntagResourceAPI を呼び出します。

```
aws braket list-tags-for-resource --resource-arn $YOUR_TASK_ARN
```

- 特定のリソースに添付されているすべてのタグを表示するには、ListTagsForResourceAPI を呼び出します。

```
aws braket tag-resource --resource-arn $YOUR_TASK_ARN --tag-keys "[\"city\",\"state\"]"
```

タグ指定の制限

Amazon Braket リソースのタグには、以下の基本的な制限が適用されます。

- リソースに割り当てることができるタグの最大数: 50
- キーの最大長: 128 文字 (ユニコード)
- 値の最大長: 256 文字 (ユニコード)
- キーと値の有効な文字:a-z, A-Z, 0-9, space と、次の文字。_ . : / = + - と @
- キーと値は大文字と小文字が区別されます。
- キーのプレフィックスawsとして を使用しないでください。 AWS の使用は予約されています。

Amazon Braket でタグを管理する

リソースのプロパティとしてタグを設定します。Amazon Braket コンソール、Amazon Braket API、または AWS CLIを使用して、タグの表示、追加、変更、一覧表示、削除を行うことができます。詳細については、「Amazon Braket API リファレンス」を参照してください。

このセクションの内容:

- タグを追加する
- タグの表示
- タグの編集
- タグの削除

タグを追加する

タグ付きリソースには、次の場合にタグを追加できます。

- リソースを作成する場合: コンソールを使用するか、AWS API の `Create` オペレーションに `Tags` パラメータを含めます。
- リソースを作成した後: コンソールを使用して量子タスクまたはノートブックリソースに移動するか、AWS API で `TagResource` のオペレーションを呼び出します。

リソースの作成時にタグを追加するには、指定したタイプのリソースを作成する権限も必要です。

タグの表示

Amazon Braket のタグ付け可能なリソースのタグを表示するには、コンソールを使用してタスクまたはノートブックリソースに移動するか、 `ListTagsForResource`APIオペレーションを AWS 呼び出します。

次の AWS APIコマンドを使用して、リソースのタグを表示できます。

- AWS API: `ListTagsForResource`

タグの編集

コンソールを使用して量子タスクまたはノートブックのリソースに移動してタグを編集するか、次のコマンドを使用して、タグ付け可能なリソースに添付されたタグの値を変更できます。すでに存在するタグキーを指定すると、そのキーの値が上書きされます。

- AWS API: `TagResource`

タグの削除

リソースからタグを削除するには、削除するキーを指定するか、コンソールを使用して量子タスクまたはノートブックリソースに移動するか、`UntagResource` オペレーションを呼び出します。

- AWS API: `UntagResource`

Amazon Braket での AWS CLI タグ付けの例

AWS Command Line Interface (AWS CLI) を使用して Amazon Braket を操作する場合、次のコードは、作成する量子タスクに適用されるタグを作成する方法を示すコマンドの例です。この例では、量子処理ユニット (QPU) Rigetti にパラメータ設定が指定された SV1 量子シミュレーターでタスクを実行しようとしています。このコマンドの例において、タグが最後、つまり他のすべての必須パラメータの後に指定されている点は重要です。この場合、タグには、キー `state` と、値 `Washington` があります。これらのタグは、この特定の量子タスクの分類や識別に役立ちます。

```
aws braket create-quantum-task --action /
"{\"braketSchemaHeader\": {\"name\": \"braket.ir.jaqcd.program\", /
    \"version\": \"1\"}, /
    \"instructions\": [{\"angle\": 0.15, \"target\": 0, \"type\": \"rz\"}], /
    \"results\": null, /
    \"basis_rotation_instructions\": null}" /
  --device-arn "arn:aws:braket:::device/quantum-simulator/amazon/sv1" /
  --output-s3-bucket  "my-example-braket-bucket-name" /
  --output-s3-key-prefix "my-example-username"  /
  --shots 100   /
  --device-parameters /
  "{\"braketSchemaHeader\": /
     {\"name\": \"braket.device_schema.rigetti.rigetti_device_parameters\", /
      \"version\": \"1\"}, \"paradigmParameters\": /
       {\"braketSchemaHeader\": /
         {\"name\": \"braket.device_schema.gate_model_parameters\", /
```

```
        \"version\": \"1\"}, /
        \"qubitCount\": 2}}" /
        --tags {\"state\":\"Washington\"}
```

この例は、 AWS CLIを使用して量子タスクを実行するときにタグを適用する方法を示しています。AWS CLI は Braket リソースの整理と追跡に役立ちます。

EventBridge を使用した量子タスクのモニタリング

Amazon EventBridge は、Amazon Braket 量子タスクのステータス変更イベントをモニタリングします。Amazon Braket からのイベントは、ほぼリアルタイムに EventBridge に提供されます。ルールを記述して、注目するイベント (イベントがルールに一致した場合に自動的に実行するアクションを含む) を指定できます。トリガーできる自動アクションには、次が含まれます。

- AWS Lambda 関数の呼び出し
- AWS Step Functions ステートマシンのアクティブ化
- Amazon SNS トピックへの通知

EventBridge は、次の Amazon Braket ステータス変更イベントをモニタリングします。

- 量子タスクのステータスの変更

量子タスクステータス変更イベントは Amazon Braket によって必ず配信されます。これらのイベントは少なくとも 1 回配信されますが、順序が乱れている可能性があります。

詳細については、「Events in Amazon EventBridge」を参照してください。

このセクションの内容:

- EventBridge での量子タスクステータスのモニタリング
- Amazon Braket EventBridge イベントの例

EventBridge での量子タスクステータスのモニタリング

EventBridge を使用すると、Amazon Braket が Braket 量子タスクに関するステータス変更の通知を送信するときに実行するアクションを定義するルールを作成できます。例えば、量子タスクのステータスが変化するたびに E メールメッセージを送信するルールを作成できます。

1. EventBridge と Amazon Braket を使用するアクセス許可を持つアカウント AWS を使用して に口グインします。
2. Amazon EventBridge コンソールを開きます。
3. 次の値を使用して、EventBridge ルールを作成します。
   - [ルールタイプ] で、[イベントパターンを持つルール] を選択してください。
   - イベントソース では、[その他] を選択します。
   - [イベントパターン] セクションで [カスタムパターン (JSONエディター)] を選択し、次のイベントパターンをテキストエリアに貼り付けます。

     ```
     {
       "source": [
         "aws.braket"
       ],
       "detail-type": [
         "Braket Task State Change"
       ]
     }
     ```

     Amazon Braket からすべてのイベントをキャプチャするには、次のコードに示すように、`detail-type` セクションを除外します。

     ```
     {
       "source": [
         "aws.braket"
       ]
     }
     ```

   - ターゲットタイプで AWS のサービスを選択し、ターゲットの選択で Amazon SNS トピックや AWS Lambda 関数などのターゲットを選択します。ターゲットは、量子タスクのステータス変更イベントが Amazon Braket から受信されるとトリガーされます。

     例えば、Amazon Simple Notification Service (SNS) トピックを使用して、イベントが発生したときに E メールまたはテキストメッセージを送信できます。これを行うには、Amazon SNS コンソールを使用して Amazon SNS トピックを作成する必要があります。詳細については、「ユーザー通知に Amazon SNS を使用する」を参照してください。

ルールの作成に関する詳細については、「イベントに反応する Amazon EventBridge ルールの作成」を参照してください。

Amazon Braket EventBridge イベントの例

Amazon Braket 量子タスクステータス変更イベントのフィールドの詳細については、「Events in Amazon EventBridge」を参照してください。

JSON の「詳細」フィールドには、次の属性が表示されます。

- **quantumTaskArn** (str): このイベントが生成された量子タスク。
- **status** (オプションで [str]): 量子タスクの移行先のステータス。
- **deviceArn** (str): この量子タスクを作成したユーザーが指定したデバイス。
- shots (int): ユーザーがリクエストした shots の数。
- **outputS3Bucket** (str): ユーザーが指定した出力バケット。
- **outputS3Directory** (str): ユーザーが指定した出力キープレフィックス。
- **createdAt** (str): 量子タスクの作成時間 (形式: ISO-8601 文字列)。
- **endedAt** (オプションで [str]): タスクが最終ステータスに達した日時。このフィールドは、量子タスクが最終ステータスに移行した場合にのみ表示されます。

次の JSON コードは、Amazon Braket 量子タスクステータス変更イベントの例を示しています。

```
{
    "version":"0",
    "id":"6101452d-8caf-062b-6dbc-ceb5421334c5",
    "detail-type":"Braket Task State Change",
    "source":"aws.braket",
    "account":"012345678901",
    "time":"2021-10-28T01:17:45Z",
    "region":"us-east-1",
    "resources":[
        "arn:aws:braket:us-east-1:012345678901:quantum-task/834b21ed-77a7-4b36-a90c-
c776afc9a71e"
    ],
    "detail":{
        "quantumTaskArn":"arn:aws:braket:us-east-1:012345678901:quantum-
task/834b21ed-77a7-4b36-a90c-c776afc9a71e",
        "status":"COMPLETED",
        "deviceArn":"arn:aws:braket:::device/quantum-simulator/amazon/sv1",
        "shots":"100",
        "outputS3Bucket":"amazon-braket-0260a8bc871e",
        "outputS3Directory":"sns-testing/834b21ed-77a7-4b36-a90c-c776afc9a71e",
```

```
        "createdAt":"2021-10-28T01:17:42.898Z",
        "eventName":"MODIFY",
        "endedAt":"2021-10-28T01:17:44.735Z"
      }
}
```

CloudWatch によるメトリクスのモニタリング

Amazon CloudWatch を使用して Amazon Braket をモニタリングすることで、raw データを収集し、リアルタイムに近い読み取り可能なメトリクスに加工することができます。最大 15 か月前までに生成された履歴情報、または Amazon CloudWatch コンソールで過去 2 週間に更新されたメトリクスを検索して、Amazon Braket のパフォーマンスをよりよく把握できます。詳細については、「CloudWatch メトリクスの使用」 を参照してください。

> **Note**
>
> Amazon SageMaker AI コンソールの [ノートブックの詳細] ページに移動すると、Amazon Braket ノートブックの CloudWatch ログストリームを表示できます。Amazon Braket ノートブックの追加設定は SageMaker コンソールで行うことができます。

このセクションの内容:

- Amazon Braket のメトリクスとディメンション

Amazon Braket のメトリクスとディメンション

メトリクスは CloudWatch での基本的な概念です。メトリクスは、CloudWatch に発行された時系列のデータポイントのセットを表します。すべてのメトリクスは、一連のディメンションによって特徴付けられます。CloudWatch のメトリクスのディメンションの詳細については、「CloudWatch のディメンション」を参照してください。

Amazon Braket は、Amazon Braket に固有の以下のメトリクスデータを Amazon CloudWatch メトリクスに送信します。

量子タスクのメトリクス

量子タスクが存在する場合、そのメトリクスが存在します。これらは CloudWatch コンソールで [AWS]/[Braket]/[デバイス別] の下に表示されます。

メトリクス	説明
カウント	量子タスクの数。
レイテンシー	このメトリクスは、量子タスクが完了したときに発行されます。量子タスクの初期化から完了までの合計時間を表します。

量子タスクメトリクスのディメンション

量子タスクメトリクスは、deviceArn パラメータのディメンションに基づいて発行されます。これは、arn:aws:braket:::device/xxx という形式です。

CloudTrail を使用した量子タスクのログ記録

Amazon Braket は AWS CloudTrail、Amazon Braket のユーザー、ロール、または によって実行されたアクションを記録するサービスである と統合 AWS のサービス されています。CloudTrail は、Amazon Braket へのすべての API コールをイベントとしてキャプチャします。キャプチャされた呼び出しには、Amazon Braket コンソールからの呼び出しと、Amazon Braket オペレーションへのコード呼び出しが含まれます。証跡を作成する場合は、Amazon Braket のイベントなど、Amazon S3 バケットへの CloudTrail イベントの継続的な配信を有効にすることができます。証跡を設定しない場合でも、CloudTrail コンソールの [イベント履歴] で最新のイベントを表示できます。CloudTrail で収集された情報を使用して、Amazon Braket に対するリクエスト、リクエスト元の IP アドレス、リクエスト者、リクエスト日時などの詳細を確認できます。

CloudTrail の詳細については、「AWS CloudTrail ユーザーガイド」を参照してください。

このセクションの内容:

- CloudTrail 内の Amazon Braket 情報
- Amazon Braket ログファイルエントリの概要

CloudTrail 内の Amazon Braket 情報

CloudTrail は、アカウントの作成 AWS アカウント 時に で有効になります。Amazon Braket でアクティビティが発生すると、そのアクティビティはイベント履歴の他の AWS のサービス イベントとともに CloudTrail イベントに記録されます。 で最近のイベントを表示、検索、ダウンロードできま

す AWS アカウント。詳細については、CloudTrail イベント履歴でのイベントの表示を参照してください。

Amazon Braket のイベントなど AWS アカウント、 のイベントの継続的な記録については、証跡を作成します。証跡により、CloudTrail はログファイルを Amazon S3 バケットに配信できます。デフォルトでは、コンソールで証跡を作成するときに、証跡がすべての AWS リージョンに適用されます。証跡は、 AWS パーティション内のすべてのリージョンからのイベントをログに記録し、指定した Amazon S3 バケットにログファイルを配信します。さらに、CloudTrail ログで収集されたイベントデータをさらに分析して処理 AWS のサービス するように他の を設定できます。詳細については、次を参照してください:

- 証跡の作成のための概要
- CloudTrail がサポートするサービスと統合
- CloudTrail 用 Amazon SNS 通知の構成
- 複数のリージョンから CloudTrail ログファイルを受け取るおよび複数のアカウントから CloudTrail ログファイルを受け取る

すべての Amazon Braket アクションが CloudTrail によりログに記録されます。例えば、`GetQuantumTask` または `GetDevice` の各アクションを呼び出すと、CloudTrail ログファイルにエントリが生成されます。

各イベントまたはログエントリには、誰がリクエストを生成したかという情報が含まれます。アイデンティティ情報は、以下を判別するのに役立ちます。

- リクエストがロールまたはフェデレーションユーザーのテンポラリなセキュリティ認証情報を使用して行われたかどうか。
- リクエストが、別の AWS のサービスによって送信されたかどうか。

詳細については、「CloudTrail userIdentity 要素」を参照してください。

Amazon Braket ログファイルエントリの概要

「トレイル」は、指定した Amazon S3 バケットにイベントをログファイルとして配信するように設定できます。CloudTrail のログファイルは、単一か複数のログエントリを含みます。イベントは、任意の出典からの単一のリクエストを表し、リクエストされたアクション、アクションの日時、リクエストパラメータなどに関する情報が含まれます。CloudTrail ログファイルは、公開 API コールの順序付けられたスタックトレースではないため、特定の順序では表示されません。

次の例では、量子タスクの詳細を取得する GetQuantumTask アクションのログエントリを示します。


```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "foobar",
    "arn": "foobar",
    "accountId": "foobar",
    "accessKeyId": "foobar",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "foobar",
        "arn": "foobar",
        "accountId": "foobar",
        "userName": "foobar"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2020-08-07T00:56:57Z"
      }
    }
  },
  "eventTime": "2020-08-07T01:00:08Z",
  "eventSource": "braket.amazonaws.com",
  "eventName": "GetQuantumTask",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "foobar",
  "userAgent": "aws-cli/1.18.110 Python/3.6.10
 Linux/4.9.184-0.1.ac.235.83.329.metal1.x86_64 botocore/1.17.33",
  "requestParameters": {
    "quantumTaskArn": "foobar"
  },
  "responseElements": null,
  "requestID": "20e8000c-29b8-4137-9cbc-af77d1dd12f7",
  "eventID": "4a2fdb22-a73d-414a-b30f-c0797c088f7c",
  "readOnly": true,
  "eventType": "AwsApiCall",
  "recipientAccountId": "foobar"
}
```

以下に、デバイスイベントの詳細を返す GetDevice アクションのログエントリを示します。


```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "foobar",
    "arn": "foobar",
    "accountId": "foobar",
    "accessKeyId": "foobar",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "foobar",
        "arn": "foobar",
        "accountId": "foobar",
        "userName": "foobar"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2020-08-07T00:46:29Z"
      }
    }
  },
  "eventTime": "2020-08-07T00:46:32Z",
  "eventSource": "braket.amazonaws.com",
  "eventName": "GetDevice",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "foobar",
  "userAgent": "Boto3/1.14.33 Python/3.7.6 Linux/4.14.158-129.185.amzn2.x86_64 exec-
env/AWS_ECS_FARGATE Botocore/1.17.33",
  "errorCode": "404",
  "requestParameters": {
    "deviceArn": "foobar"
  },
  "responseElements": null,
  "requestID": "c614858b-4dcf-43bd-83c9-bcf9f17f522e",
  "eventID": "9642512a-478b-4e7b-9f34-75ba5a3408eb",
  "readOnly": true,
  "eventType": "AwsApiCall",
  "recipientAccountId": "foobar"
}
```

Amazon Braket を使用した高度なログ記録

ロガーを使用して、タスク処理プロセス全体を記録できます。これらの高度なロギング技術により、バックグラウンドポーリングを確認し、後でデバッグするためのレコードを作成できます。

ロガーを使用するには、poll_timeout_seconds パラメータと poll_interval_seconds パラメータを変更することをお勧めします。これにより、量子タスクが長時間実行され、量子タスクステータスが継続的にログに記録され、結果がファイルに保存されます。このコードを Jupyter Notebook ではなく Python スクリプトに転送して、スクリプトをバックグラウンドでプロセスとして実行できるようにします。

ロガーを設定する

まず、次の例の行に示すように、すべてのログがテキストファイルに自動的に書き込まれるように、ロガーを設定します。

```
# import the module
import logging
from datetime import datetime

# set filename for logs
log_file = 'device_logs-'+datetime.strftime(datetime.now(), '%Y%m%d%H%M%S')+'.txt'
print('Task info will be logged in:', log_file)

# create new logger object
logger = logging.getLogger("newLogger")

# configure to log to file device_logs.txt in the appending mode
logger.addHandler(logging.FileHandler(filename=log_file, mode='a'))

# add to file all log messages with level DEBUG or above
logger.setLevel(logging.DEBUG)
```

```
Task info will be logged in: device_logs-20200803203309.txt
```

回路を作成して実行する

これで、この例に示すように、回路を作成し、それをデバイスに送信して実行し、何が起こるかを確認することができます。

```
# define circuit
```

```
circ_log = Circuit().rx(0, 0.15).ry(1, 0.2).rz(2, 0.25).h(3).cnot(control=0,
 target=2).zz(1, 3, 0.15).x(4)
print(circ_log)
# define backend
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
# define what info to log
logger.info(
    device.run(circ_log, s3_location,
               poll_timeout_seconds=1200, poll_interval_seconds=0.25, logger=logger,
 shots=1000)
    .result().measurement_counts
)
```

ログファイルを確認する

ファイルに書き込まれた内容を確認するには、次のコマンドを入力します。

```
# print logs
! cat {log_file}
```

```
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: start polling for completion
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status CREATED
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status CREATED
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status QUEUED
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status RUNNING
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status RUNNING
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status COMPLETED
Counter({'00001': 493, '00011': 493, '01001': 5, '10111': 4, '01011': 3, '10101': 2})
```

ログファイルから ARN を取得する

前の例に示した、返されるログファイルの出力から、ARN 情報を取得できます。完了したタスクの結果を取得するには、ARN ID を使用します。

```
# parse log file for arn
```

```
with open(log_file) as openfile:
    for line in openfile:
        for part in line.split():
            if "arn:" in part:
                arn = part
                break
# remove final semicolon in logs
arn = arn[:-1]

# with this arn you can restore again task from unique arn
task_load = AwsQuantumTask(arn=arn, aws_session=AwsSession())

# get results of task
result = task_load.result()
```

Amazon Braket のクォータ

次のテーブルに Amazon Braket のサービスクォータの一覧を示します。Service Quotas (制限とも呼ばれます) は、AWS アカウントのサービスリソースまたはオペレーションの最大数です

一部のクォータは増やすことができます。詳細については、AWS のサービスクォータを参照してください。

- バーストレートクォータを増やすことはできません。
- 調整可能なクォータの最大レート増加 (調整できないバーストレートを除く) は、指定されたデフォルトのレート制限の 2 倍です。例えば、デフォルトのクォータの 60 個は最大 120 個に調整できます。
- 同時実行の SV1 (DM1) 量子タスクのための調整可能なクォータでは、AWS リージョンごとに最大 60 個を使用できます。
- ハイブリッドジョブで許可されるコンピューティングインスタンスの最大数は 1 で、クォータは引き上げ可能です。

[リソース]	説明	制限	調整可能
API リクエストのレート	現在のリージョンのこのアカウントで送信できる 1 秒あたりのリクエストの最大数。	140	可
API リクエストのバーストレート	現在のリージョンのこのアカウントで 1 回のバーストで送信できる 1 秒あたりの追加リクエストの最大数 (RPS) 。	600	不可
`CreateQuantumTask` リクエストのレート	現在のリージョンのこのアカウントで送信できる 1 秒あたりの `CreateQua`	20/秒	可

[リソース]	説明	制限	調整可能
	ntumTask リクエストの最大数。		
CreateQuantumTask リクエストのバーストレート	現在のリージョンのこのアカウントで 1 回のバーストで送信できる 1 秒あたりの追加の CreateQuantumTask リクエストの最大数 (RPS)。	40	不可
SearchQuantumTasks リクエストのレート	現在のリージョンのこのアカウントで送信できる 1 秒あたりの SearchQuantumTasks リクエストの最大数。	5/秒	可
SearchQuantumTasks リクエストのバーストレート	現在のリージョンのこのアカウントで 1 回のバーストで送信できる 1 秒あたりの追加の SearchQuantumTasks リクエストの最大数 (RPS)。	50	不可
GetQuantumTask リクエストのレート	現在のリージョンのこのアカウントで送信できる 1 秒あたりの GetQuantumTask リクエストの最大数。	100/秒	可

[リソース]	説明	制限	調整可能
GetQuantumTask リクエストのバーストレート	現在のリージョンのこのアカウントで 1 回のバーストで送信できる 1 秒あたりの追加の GetQuantumTask リクエストの最大数 (RPS)。	500	不可
CancelQuantumTask リクエストのレート	現在のリージョンのこのアカウントで送信できる 1 秒あたりの CancelQuantumTask リクエストの最大数。	2/秒	可
CancelQuantumTask リクエストのバーストレート	現在のリージョンのこのアカウントで 1 回のバーストで送信できる 1 秒あたりの追加の CancelQuantumTask リクエストの最大数 (RPS)。	20	不可
GetDevice リクエストのレート	現在のリージョンのこのアカウントで送信できる 1 秒あたりの GetDevice リクエストの最大数。	5/秒	可

[リソース]	説明	制限	調整可能
GetDevice リクエストのバーストレート	現在のリージョンのこのアカウントで 1 回のバーストで送信できる 1 秒あたりの追加の GetDevice リクエストの最大数 (RPS)。	50	不可
SearchDevices リクエストのレート	現在のリージョンのこのアカウントで送信できる 1 秒あたりの SearchDevices リクエストの最大数。	5/秒	可
SearchDevices リクエストのバーストレート	現在のリージョンのこのアカウントで 1 回のバーストで送信できる 1 秒あたりの追加の SearchDevices リクエストの最大数 (RPS)。	50	不可
CreateJob リクエストのレート	現在のリージョンのこのアカウントで送信できる 1 秒あたりの CreateJob リクエストの最大数。	1/秒	可

[リソース]	説明	制限	調整可能
CreateJob リクエストのバーストレート	現在のリージョンのこのアカウントで 1 回のバーストで送信できる 1 秒あたりの追加の CreateJob リクエストの最大数 (RPS)。	5	不可
SearchJobs リクエストのレート	現在のリージョンのこのアカウントで送信できる 1 秒あたりの SearchJob リクエストの最大数。	5/秒	可
SearchJobs リクエストのバーストレート	現在のリージョンのこのアカウントで 1 回のバーストで送信できる 1 秒あたりの追加の SearchJob リクエストの最大数 (RPS)。	50	不可
GetJob リクエストのレート	現在のリージョンのこのアカウントで送信できる 1 秒あたりの GetJob リクエストの最大数。	5/秒	可
GetJob リクエストのバーストレート	現在のリージョンのこのアカウントで 1 回のバーストで送信できる 1 秒あたりの追加の GetJob リクエストの最大数 (RPS)。	25	不可

[リソース]	説明	制限	調整可能
CancelJob リクエストのレート	現在のリージョンのこのアカウントで送信できる 1 秒あたりの CancelJob リクエストの最大数。	2/秒	可
CancelJob リクエストのバーストレート	現在のリージョンのこのアカウントで 1 回のバーストで送信できる 1 秒あたりの追加の CancelJob リクエストの最大数 (RPS)。	5	不可
同時 SV1 量子タスクの数	現在のリージョンの状態ベクトルシミュレーター (SV1) で実行される同時量子タスクの最大数。	100 (us-east-1)、 50 (us-west-1)、 100 (us-west-2)、 50 (eu-west-2)	不可
同時 DM1 量子タスクの数	現在のリージョンの密度マトリックスシミュレーター (DM1) で実行する同時タスクの最大数。	100 (us-east-1)、 50 (us-west-1)、 100 (us-west-2)、 50 (eu-west-2)	不可
同時 TN1 量子タスクの数	現在のリージョンのテンソルネットワークシミュレーター (TN1) で実行する同時量子タスクの最大数。	10 (us-east-1)、 10 (us-west-2)、 5 (eu-west-2)	はい

[リソース]	説明	制限	調整可能
同時ハイブリッドジョブの数	現在のリージョンで作成できる同時ハイブリッドジョブの最大数。	3	はい
ハイブリッドジョブ数の実行時制限	ハイブリッドジョブを実行できる最大日数。	5	不可

ハイブリッドジョブのデフォルトの古典コンピューティングインスタンスのクォータを次に示します。これらのクォータを引き上げたい場合は、サポート にご依頼ください。また、インスタンスごとに、適用可能なリージョンも指定されています。

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c4.xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c4.xlarge タイプのインスタンス	5	はい	はい	はい	はい	あり	なし

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
	の最大数。							
ハイブリッドジョブの ml.c4.2xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c4.2xlarge タイプのインスタンスの最大数。	5	はい	はい	はい	はい	あり	なし

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c4.4xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c4.4xlarge タイプのインスタンスの最大数。	5	はい	はい	はい	はい	あり	なし

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c4.8xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c4.8xlarge タイプのインスタンスの最大数。	5	はい	はい	はい	あり	なし	不可

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c5.xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c5.xlarge タイプのインスタンスの最大数。	5	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c5.2xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c5.2xlarge タイプのインスタンスの最大数。	5	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c5.4xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c5.4xlarge タイプのインスタンスの最大数。	1	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c5.9xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c5.9xlarge タイプのインスタンスの最大数。	1	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c5.18xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c5.18xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c5n.xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c5n.xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	あり	なし	不可

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c5n.2xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c5n.2xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	あり	なし	不可

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c5n.4xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c5n.4xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	あり	なし	不可

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c5n.9xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c5n.9xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	あり	なし	不可

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.c5n.18xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.c5n.18xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	あり	なし	不可

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.g4dn.xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.g4dn.xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.g4dn.2xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.g4dn.2xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.g4dn.4xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.g4dn.4xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.g4dn.8xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.g4dn.8xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.g4dn.12xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.g4dn.12xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.g4dn.16xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.g4dn.16xlarge タイプのインスタンスの最大数。	0	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.m4.xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.m4.xlarge タイプのインスタンスの最大数。	5	はい	はい	はい	はい	あり	なし

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.m4.2xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.m4.2xlarge タイプのインスタンスの最大数。	5	はい	はい	はい	はい	あり	なし

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.m4.4xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.m4.4xlarge タイプのインスタンスの最大数。	2	はい	はい	はい	はい	あり	なし

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.m4.10› large インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.m4.10› large タイプのインスタンスの最大数。	0	はい	はい	はい	はい	あり	なし

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.m4.16› large インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.m4.16› large タイプのインスタンスの最大数。	0	はい	はい	はい	はい	あり	なし

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.m5.large インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.m5.large タイプのインスタンスの最大数。	5	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.m5.xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.m5.xlarge タイプのインスタンスの最大数。	5	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.m5.2xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.m5.2xlarge タイプのインスタンスの最大数。	5	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.m5.4xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.m5.4xlarge タイプのインスタンスの最大数。	5	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.m5.12› large インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.m5.12› large タイプのインスタンスの最大数。	0	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.m5.24› large インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.m5.24› large タイプのインスタンスの最大数。	0	はい	はい	はい	はい	はい	はい

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.p2.xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.p2.xlarge タイプのインスタンスの最大数。	0	はい	あり	なし	あり	なし	不可

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.p2.8xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.p2.8xlarge タイプのインスタンスの最大数。	0	はい	あり	なし	あり	なし	不可

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.p2.16xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.p2.16xlarge タイプのインスタンスの最大数。	0	はい	あり	なし	あり	なし	不可

[リソース]	説明	制限	調整可能	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
ハイブリッドジョブの ml.p4d.24xlarge インスタンスの最大数	このアカウントとリージョンのすべての Amazon Braket Hybrid Jobs で許可される ml.p4d.24xlarge タイプのインスタンスの最大数。	0	はい	あり	なし	あり	なし	不可

制限の更新をリクエストする

インスタンスタイプの ServiceQuotaExceeded 例外を受け取り、十分なインスタンスが利用できない場合は、AWSコンソールの Service Quotas ページから制限の引き上げをリクエストし、AWSサービスで Amazon Braket を検索できます。

> **Note**
>
> ハイブリッドジョブがリクエストされた ML コンピューティング性能をプロビジョニングできない場合は、別のリージョンを使用してください。また、テーブルにないインスタンスは、ハイブリッドジョブでは使用できません。

追加のクォータと制限

- Amazon Braket 量子タスクアクションのサイズは 3MB に制限されています。
- SV1 の最大実行時間は、31 量子ビット以下の回路では 3 時間、31 量子ビットを超える回路では 11 時間です。
- SV1、DM1、および Rigetti デバイスで許可されるタスクあたりの最大ショット数は 50,000 です。
- TN1 で許可されるタスクあたりの最大ショット数は 1,000 です。
- AQTのIBEX-Q1デバイスの場合、最大はタスクあたり 2000 ショットです。
- のすべてのIonQデバイスの場合: タスクあたりの最小ショット数は 100 です。オンデマンドモデルを使用する場合、エラー緩和タスクには 100 万のゲートショット制限と最低 2500 ショットがあります。直接予約の場合、ゲートショットに制限はなく、エラー緩和タスクでは最低 500 ショットを実行する必要があります。
- QuEra の Aquila デバイスの場合、タスクあたりの制限は 1,000 ショットです。
- IQMGarnet および Emerald デバイスの場合、タスクあたり最大 20,000 ショットです。
- TN1 および QPU デバイスの場合、タスクあたりのショットは > 0 である必要があります。

Amazon Braket デベロッパーガイドのドキュメント履歴

次の表では、Amazon Braket のドキュメントリリースについて説明します。

- API リファレンスの最終更新日: 2025 年 11 月 20 日
- ドキュメントの最終更新日: 2026 年 3 月 2 日

変更	説明	日付
IonQ Aria-1 デバイスリタイア	IonQ Aria-1 デバイスのサポートを削除しました。	2026 年 3 月 2 日
「予約の操作」ページを更新する	「予約の操作」ページの明確性の向上	2026 年 2 月 3 日
Braket ノートブックとマネージドコンテナの Python バージョン 3.12 のサポート	Amazon Braket ノートブックとマネージドコンテナ (Base、CUDA-Q、Penny Lane、Tensorflow) の Python バージョン 3.12 サポートを追加しました。Python 3.12 アップグレードのトラブルシューティングガイドが含まれています。	2026 年 1 月 21 日
P3 の例を削除する	SageMaker は`ml.p3`インスタンスファミリーを廃止します。例を推奨される`ml.g4dn`インスタンスファミリーに置き換えました。	2025 年 12 月 19 日
新しい使用制限機能	Amazon Braket の使用制限機能のサポートが追加されました。これにより、設定された使用しきい値を超えるタスクを自動的に検証および拒否す	2025 年 11 月 20 日

	る個々の QPUs にオプションの予算上限を設定できます。	
新しい Braket デバイス AQT IBEX-Q1	AQT IBEX-Q1 デバイスのサポートが追加されました。このデバイスは、超高バキュームチェンバーにある巨視的無線周波数トラップ内の $^{40}Ca^{+}$ のイオンの反射に基づいています。	2025 年 11 月 18 日
Amazon Braket NBI での CUDA-Q の新しいネイティブサポート	Amazon Braket ノートブックインスタンスでの CUDA-Q のネイティブサポートが追加されました。詳細については、「CUDA-Q in NBIs」を参照してください。	2025 年 11 月 10 日
IonQ Aria-2 デバイスの廃止	IonQ Aria-2 デバイスのサポートを終了しました。	2025 年 10 月 27 日
ハイブリッドジョブのドキュメントの統合	ハイブリッドジョブの各セクションを統合し、「Working with Amazon Braket Hybrid Jobs」の下に表示されるようにしました。	2025 年 10 月 21 日
Braket、CUDA-Q コンテナを新しく提供	提供されている CUDA-Q ハイブリッドジョブコンテナのサポートを追加しました。詳細については、「Define the environment for your algorithm script」を参照してください。	2025 年 9 月 2 日

ローカルデバイスエミュレータ機能を追加	逐語的なプログラムを量子デバイスに送信する前にエミュレートするローカル量子デバイスエミュレータツールのサポートを追加しました。	2025/08/25
Pennylane と CUDA-Q のページを「ビルド」セクションに移動	Pennylane のページと CUDA-Q のページを、目次の「ビルド」セクションの下に移動しました。	2025 年 8 月 15 日
`ProgramSet` 機能を追加	1 つの量子タスクで複数の量子回路を実行するオペレーションであるプログラムセットのサポートを追加しました。	2025 年 8 月 14 日
デバイス IQM Emerald を追加	IQM Emerald デバイスのサポートを追加しました。このデバイスは、正方形 (結晶) 格子トポロジを持つ 54 量子ビットデバイスです。	2025 年 7 月 21 日
AmazonBraketServiceRolePolicy ポリシーを更新	AmazonBraketServiceRolePolicy は、aws:PrincipalAccount に対し、s3:* および logs:* アクションのみを提供するようになりました。これにより、リクエスタのバケットとロググループのみにアクセスが制限されます。	2025 年 7 月 11 日

実験的な機能として動的回路を追加	実験機能として、中間回路測定およびフィードフォワードオペレーションが利用できるようになりました。「Access to dynamic circuits on IQM devices」を参照。	2025 年 6 月 26 日
AmazonBraketFullAccess ポリシーを更新	AmazonBraketFullAccess に、ハードウェアコストをコンソールに表示するための pricing:GetProducts が含まれるようになりました。	2025 年 4 月 14 日
デバイス IonQ Forte-Enterprise-1 を追加	IonQ Forte-Enterprise-1 デバイスのサポートを追加しました。このデバイスは、トラップドイオン技術を利用する 36 個の量子ビットデバイスです。	2025 年 3 月 17 日
S3 のアクセス許可条件を改善	セキュリティを向上させるために、`AmazonBraketFullAccess` は `aws:PrincipalAccount` に対して `s3:*` アクションのみを提供するようになりました。これにより、アクセスがリクエスタ自身のバケットのみに制限されます。	2025 年 3 月 7 日
デバイス Rigetti Ankaa-3 を追加	Rigetti Ankaa-3 デバイスのサポートを追加しました。このデバイスは、スケーラブルなマルチチップ技術を利用する 84 量子ビットのデバイスです。	2025 年 1 月 14 日

Rigetti Ankaa-2 デバイスの廃止	Rigetti Ankaa-2 デバイスのサポートを終了しました。	2025 年 1 月 14 日
IPv6 トラフィックのサポート	Amazon Braket が、デュアルスタックエンドポイント `braket.{region}.api.aws` を使用して、IPv6 トラフィックをサポートするようになりました。	2024 年 12 月 12 日
Amazon Braket での NVIDIA's CUDA-Q のサポート	お客様は、Amazon Braket で NVIDIA's CUDA-Q 開発者フレームワークを使用して量子プログラムを実行できるようになりました。	2024 年 12 月 6 日
IonQ Forte-1 デバイスがすぐに利用可能に	IonQ Forte-1 デバイスは予約専用ではなくなり、カスタマーがすぐに利用できるようになりました。	2024 年 11 月 22 日
Rigetti Aspen-M-3 デバイスの廃止	Rigetti Aspen-M-3 デバイスのサポートを終了しました。	2024 年 9 月 27 日
IonQ Harmony デバイスの廃止	IonQ Harmony デバイスのサポートを終了しました。	2024 年 8 月 29 日
デバイス Rigetti Ankaa-2 を追加	Rigetti Ankaa-2 デバイスのサポートを追加しました。このデバイスは、スケーラブルなマルチチップ技術を利用する 84 量子ビットのデバイスです。	2024 年 8 月 26 日

デベロッパーガイドの再編成	新しいデベロッパーガイドでは、既存のビルド、テスト、実行を探求する旅にカスタマーを誘い、Amazon Braket を使用したこの旅程をカスタマーにご案内します。	2024 年 8 月 23 日
OQC Lucy デバイスの廃止	OQC Lucy デバイスのサポートを終了しました。	2024 年 6 月 28 日
デバイス IQM Garnet とリージョン Europe North 1 を追加	IQM Garnet デバイスのサポートを追加しました。このデバイスは、正方形格子トポロジを持つ 20 量子ビットデバイスです。Braket がサポートされるリージョンを欧州北部 1 (ストックホルム) にまで拡張しました。	2024 年 5 月 22 日
ローカル離調をリリース	実験的な機能として、QuEra Aquila QPU のローカル離調機能が追加されました。	2024 年 4 月 11 日
ノートブック非アクティブ状態マネージャーをリリース	ノートブックインスタンスを作成するときは、非アクティブ状態マネージャーを有効にし、Braket ノートブックインスタンスを自動的にリセットするアイドル時間を設定できるようになりました。	2024 年 3 月 27 日
目次を再編	AWS スタイルガイドの要件に従い、カスタマーエクスペリエンスのコンテンツフローを改善するために、Amazon Braket の目次を再編成しました。	2023 年 12 月 12 日

Braket Direct をリリース	以下の Braket Direct 機能のサポートを追加しました。 • 予約の使用 • エキスパートのアドバイスを受ける • 実験機能を探索する	2023 年 11 月 27 日
Amazon Braket ノートブックインスタンスを作成する を更新	新規および既存の Amazon Braket カスタマー向けの、ノートブックインスタンスを作成するための情報を追加することで、本ドキュメントを更新しました。	2023 年 11 月 27 日
独自のコンテナ (BYOC) を更新	BYOC を持ち込むタイミング、BYOC を持ち込むためのレシピ、BYOC での Braket Hybrid Jobs の実行に関する情報を追加することで、本ドキュメントを更新しました。	2023 年 10 月 18 日

ハイブリッドジョブデコレータをリリース	「ローカルコードをハイブリッドジョブとして実行する」ページを追加しました。以下の例が記載されています。 • ローカル Python コードからハイブリッドジョブを作成する • 追加の Python パッケージとソースコードをインストールする • ハイブリッドジョブインスタンスにおいてデータをロードしたり保存したりする • ハイブリッドジョブデコレータに関するベストプラクティス	2023 年 10 月 16 日
キューの可視性を追加	queue depth と queue position を追加して本デベロッパーガイドドキュメントを更新しました。 キューの可視性に関する新しい API の変更を反映するように API ドキュメントを更新しました。	2023 年 9 月 25 日
本ドキュメント内で名前を標準化	「ジョブ」のインスタンスを「ハイブリッドジョブ」に、「タスク」を「量子タスク」に変更するよう、本ドキュメントを更新しました。	2023 年 9 月 11 日

デバイス IonQ Aria 2 を追加	IonQ Aria 2 デバイスのサポートを追加しました。	2023 年 9 月 8 日
ネイティブゲートを更新	本ドキュメントを更新し、Rigetti からのネイティブゲートへのプログラムによるアクセスに関する情報を追加しました。	2023 年 8 月 16 日
Xanadu の離脱	本ドキュメントを更新し、すべての Xanadu デバイスに関する記述を削除しました。	2023 年 6 月 2 日
デバイス IonQ Aria を追加	IonQ Aria デバイスのサポートを追加しました。	2023 年 5 月 16 日
Rigetti デバイスを廃止	Rigetti Aspen-M-2 のサポートを終了しました。	2023 年 5 月 2 日
AmazonBraketFullAccess ポリシー情報を更新	AmazonBraketFullAccess ポリシーの内容を定義するスクリプトを更新し、servicequotas:GetServiceQuota および cloudwatch:GetMetricData アクションと、クォータの制限に関する情報を追加しました。	2023 年 4 月 19 日
ガイド付きジャーニーをローンチ	Braket に簡単にオンボーディングするための最新の方法を反映するよう、本ドキュメントを変更しました。	2023 年 4 月 5 日
デバイス Rigetti Aspen-M-3 を追加	Rigetti Aspen-M-3 デバイスのサポートを追加しました。	2023 年 1 月 17 日

随伴勾配機能を追加	SV1 によって提供される随伴勾配機能に関する情報を追加しました。	2022 年 12 月 7 日
アルゴリズムライブラリ機能を追加	構築済みの量子アルゴリズムのカタログを提供する Braket アルゴリズムライブラリに関する情報を追加しました。	2022 年 11 月 28 日
D-Wave の離脱	すべての D-Wave デバイスの廃止に対応するよう、本ドキュメントを更新しました。	2022 年 11 月 17 日
デバイス QuEra Aquila を追加	QuEra Aquila デバイスのサポートを追加しました。	2022 年 10 月 31 日
Braket Pulse をサポート	Braket Pulse のサポートを追加しました。これにより、Rigetti および OQC デバイスでパルス制御を使用できるようになりました。	2022 年 10 月 20 日
IonQ ネイティブゲートをサポート	IonQ デバイスによって提供されるネイティブゲートセットのサポートを追加しました。	2022 年 9 月 13 日
インスタンスクォータを追加	Hybrid Jobs に関連付けられたデフォルトの古典コンピューティングインスタンスクォータを更新しました。	2022 年 8 月 22 日
サービスダッシュボードを追加	サービスダッシュボードが含まれるよう、コンソールのスクリーンショットを更新しました	2022 年 8 月 17 日
デバイス Rigetti Aspen-M-2 を追加	Rigetti Aspen-M-2 デバイスのサポートを追加しました。	2022 年 8 月 12 日

OpenQASM の機能を追加	ローカルシミュレーター (braket_sv および braket_dm) の OpenQASM 機能のサポートを追加しました。	2022 年 8 月 4 日
コスト追跡手順を追加	シミュレーターとハードウェアのワークロードに関するほぼリアルタイムの最大コスト見積もりを取得する方法を追加しました。	2022 年 7 月 18 日
デバイス Xanadu Borealis を追加	Xanadu Borealis デバイスのサポートを追加しました。	2022 年 6 月 2 日
簡単なオンボーディング手順を追加	簡単な新しいオンボーディング手順の仕組みに関する情報を追加しました。	2022 年 5 月 16 日
デバイス D-Wave Advantage_system6.1 を追加	D-Wave Advantage_system6.1 デバイスのサポートを追加しました。	2022 年 5 月 12 日
埋め込みシミュレーターをサポート	ハイブリッドジョブで埋め込みシミュレーションを実行する方法と PennyLane 稲妻シミュレーターを使用する方法を追加しました	2022 年 5 月 4 日
AmazonBraketFullAccess - Amazon Braket のフルアクセスポリシー	ユーザーが Amazon Braket 用に作成および使用されているバケットを表示および検査できるようにする s3:ListAllMyBuckets アクセス許可を追加しました。	2022 年 3 月 31 日

OpenQASM をサポート	ゲートベースの量子デバイスとシミュレーターの OpenQASM 3.0 サポートを追加しました。	2022 年 3 月 7 日
量子ハードウェアプロバイダー Oxford Quantum Circuits とリージョン eu-west-2 を追加	OQC と eu-west-2 のサポートを追加しました。	2022 年 2 月 28 日
デバイス Rigetti を追加	Rigetti Aspen M-1 のサポートを追加しました。	2022 年 2 月 15 日
新しいリソース制限	同時の DM1 タスクと SV1 タスクの最大数を 55 から 100 に増やしました。	2022 年 1 月 5 日
デバイス Rigetti を追加	Rigetti Aspen-11 のサポートを追加しました。	2021 年 12 月 20 日
Rigetti デバイスを廃止	Rigetti Aspen-10 のサポートを終了しました。	2021 年 12 月 20 日
新しい結果タイプ	ローカル密度マトリックスシミュレーターと DM1 デバイスでサポートされる縮約密度マトリックス結果タイプを追加しました。	2021 年 12 月 20 日
ポリシーの説明を更新	Amazon Braket が、`servicerole/` のパスを含めるようにロール ARN を更新しました。ポリシーの更新内容の詳細については、「Amazon Braket updates to AWS managed policies」の表を参照してください。	2021 年 11 月 29 日

Amazon Braket Jobs	Amazon Braket Hybrid Jobs と API のユーザーガイドが追加されました。	2021 年 11 月 29 日
デバイス Rigetti を追加	Rigetti Aspen-10 のサポートを追加しました。	2021 年 11 月 20 日
D-Wave デバイスを廃止	D-Wave QPU、Advantage_system1 のサポートを終了しました。	2021 年 11 月 4 日
デバイス D-Wave を追加	追加の D-Wave QPU、Advantage_system4 のサポートを追加しました。	2021 年 10 月 5 日
新しいノイズシミュレーター	最大 17 qubitsの回路をシミュレートできる密度マトリックスシミュレーター (DM1) とローカルノイズシミュレーター braket_dm のサポートを追加しました。	2021 年 5 月 25 日
PennyLane をサポート	Amazon Braket での PennyLane のサポートを追加しました。	2020 年 12 月 8 日
シミュレーターを追加	Tensor Network Simulator (TN1) のサポートを追加しました。これにより、より大きな回路が可能になります。	2020 年 12 月 8 日
タスクバッチ処理	Braket がカスタマータスクのバッチ処理をサポートするようになりました。	2020 年 11 月 24 日

手動 qubit 割り当て	Braket が、Rigetti デバイスでの手動 qubit 割り当てをサポートするようになりました。	2020 年 11 月 24 日
調整可能なクォータ	Braket が、タスクリソースのセルフサービス調整可能クォータをサポートするようになりました。	2020 年 10 月 30 日
PrivateLink をサポート	Braket ジョブ用のプライベート VPC エンドポイントを設定できるようになりました。	2020 年 10 月 30 日
タグをサポート	Braket が、量子タスクリソースの API ベースのタグをサポートするようになりました。	2020 年 10 月 30 日
デバイス D-Wave を追加	追加の D-Wave QPU、Advantage_system1 のサポートを追加しました。	2020 年 9 月 29 日
初回リリース	Amazon Braket ドキュメントの初回リリースです。	2020 年 8 月 12 日

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。