aws

デベロッパーガイド

Amazon Braket

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon Braket: デベロッパーガイド

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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。

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
コストの追跡と節約 10
ほぼリアルタイムのコスト追跡 11
コスト削減のベストプラクティス 13
API リファレンスとリポジトリ 14
コアリポジトリ 15
プラグイン 15
サポートされているリージョンとデバイス 16
リージョンとエンドポイント 20
入門 22
Amazon Braket を有効にする 22
前提条件 23
Amazon Braket を有効にする手順 23
Amazon Braket ノートブックインスタンスを作成する 24
(アドバンスト) CloudFormation を使用して Braket ノートブックを作成する 26
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 27
ステップ 2: Amazon SageMaker AI が引き受ける IAM ロールを作成する 28
ステップ 3: プレフィックスが付いた SageMaker AI ノートブックインスタンスを作成する amazon-braket- 29
ビルド 30
最初の回路の構築 30
最初の量子アルゴリズムの構築 35
SDK での回路の構築 35
回路の検査 46
結果タイプのリスト 48
エキスパートのアドバイスを受ける 52
(アドバンスト) Amazon Braket Hybrid Jobs の開始方法 53
ハイブリッドジョブとは 54
Amazon Braket Hybrid Jobsを使用する時期 55

入力、出力、環境変数、およびヘルパー関数 55
アルゴリズムスクリプトの環境を定義する 59
ハイパーパラメータの使用 61
(アドバンスト) OpenQASM 3.0 を使用して回路を実行する 63
OpenQASM 3.0 とは 64
OpenQASM 3.0 を使用するタイミング 64
OpenQASM 3.0 の仕組み 64
前提条件 65
Braket はどのような OpenQASM 機能をサポートしていますか？ 65
OpenQASM 3.0 量子タスクの例を作成して送信する 71
さまざまな Braket デバイスでの OpenQASM のサポート 73
ノイズをシミュレートする 84
Qubit 再ワイヤリング 85
逐語的なコンパイル 86
Braket コンソール 86
追加リソース 86
勾配の計算 87
特定の量子ビットの測定 88
(アドバンスト) 実験機能を調べる 89
QuEra Aquila でのローカルデチューニングへのアクセス 89
QuEra Aquila の背の高いジオメトリへのアクセス 91
QuEra Aquila でのタイトなジオメトリへのアクセス 92
Amazon Braket での (アドバンスト) パルス制御 93
[フレーム] 94
ポート 94
波形 95
フレームとポートのロール 96
Hello Pulse の使用 97
パルスを使用したネイティブゲートへのアクセス 101
(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 103
Hello AHS: 最初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 103
QuEra Aquila を使用してアナログプログラムを送信する 117
(アドバンスト) AWS Boto3 の使用 136
Amazon Braket Boto3 クライアントを有効にする 136
Boto3 と Braket SDK の AWS CLI プロファイルを設定する 140
テスト 142

シミュレーターへの量子タスクの送信 142
ローカル状態ベクトルシミュレーター (braket_sv) 143
局所密度行列シミュレーター (braket_dm) 144
ローカル AHS シミュレーター (braket_ahs) 145
ステートベクトルシミュレーター (SV1) 145
密度行列シミュレーター (DM1) 146
テンソルネットワークシミュレーター (TN1) 147
埋め込みシミュレーターについて 148
シミュレーターを比較する 149
Amazon Braket の量子タスクの例 153
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 158
量子タスクのバッチ処理 160
(アドバンスト) Amazon Braket Hybrid Jobs の使用 162
ローカルコードをハイブリッドジョブとして実行する 163
Amazon Braket Hybrid Jobs でハイブリッドジョブを実行する 171
最初のハイブリッドジョブを作成する 173
ジョブ結果の保存 183
チェックポイントを使用したハイブリッドジョブの保存と再起動 184
ローカルモードでのハイブリッドジョブの構築とデバッグ 186
実行 188
QPUs への量子タスクの送信 189
IonQ 190
IQM 191
Rigetti 191
QuEra 192
例: QPU への量子タスクの送信 192
コンパイルされた回路の検査 196
量子タスクはいつ実行されますか？ 196
QPU の可用性ウィンドウとステータス 197
キューの可視性 197
E メールまたは SMS 通知を設定する 199
(アドバンスト) Amazon Braket Hybrid Job の管理 199
スクリプトを実行するようにハイブリッドジョブインスタンスを設定する 200
ハイブリッドジョブをキャンセルする方法 204
パラメトリックコンパイルを使用してハイブリッドジョブを高速化する 205
Amazon Braket で PennyLane を使う 206

独自のコンテナ ... 221
Amazon Braket での CUDA-Q の使用 ... 229
を使用してハイブリッドジョブを直接操作する API ... 233
(アドバンスト) 予約の操作 ... 236
予約を作成する方法 ... 237
予約中の量子タスクの実行 ... 238
予約中のハイブリッドジョブの実行 ... 242
予約の最後に何が起こるか ... 243
既存の予約をキャンセルまたは再スケジュールする ... 243
(アドバンスト) エラー緩和手法 ... 244
IonQ デバイスのエラー軽減手法 ... 244
トラブルシューティング ... 247
AccessDeniedException ... 247
CreateQuantumTask オペレーションの呼び出し中にエラーが発生しました (ValidationException) ... 247
SDK 機能が動作しません ... 248
ServiceQuotaExceededException が原因でハイブリッドジョブが失敗する ... 248
ノートブックインスタンスでコンポーネントが動作を停止しました ... 249
OpenQASM のトラブルシューティング ... 249
Include ステートメントエラー ... 250
連続しないqubitsエラー ... 250
物理エラーqubitsと仮想qubitsエラーの混在 ... 251
同じプログラムエラーqubitsでの結果タイプのリクエストと測定 ... 251
Classical および qubit register limits exceeded エラー ... 251
逐語的なプラグマエラーが前にないボックス ... 252
ネイティブゲートが欠落している逐語的なボックスのエラー ... 252
逐語的なボックスで物理qubitsエラーが見つからない ... 252
逐語的なプラグマに「ブラケット」エラーがない ... 253
単一 はインデックス作成qubitsできないエラー ... 253
2 つのqubitゲートqubitsの物理 が接続されていないエラー ... 253
Local Simulator のサポートに関する警告 ... 254
セキュリティ ... 255
セキュリティの責任共有 ... 256
データ保護 ... 256
データ保持 ... 257
Amazon Braket へのアクセスを管理する ... 257

Amazon Braket のリソース 258
ノートブックとロール 258
AmazonBraketFullAccess ポリシーについて 259
AmazonBraketJobsExecutionPolicy ポリシーについて 265
特定のデバイスへのユーザーアクセスを制限する 268
AWS マネージドポリシーに対する Amazon Braket の更新 269
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 270
特定の S3 バケットへのユーザーアクセスを制限する 272
サービスリンクロール 272
Amazon Braket のサービスリンクロール許可 273
コンプライアンス検証 275
インフラストラクチャセキュリティ 276
サードパーティーのセキュリティ 277
VPC エンドポイント (PrivateLink) 277
Amazon Braket VPC エンドポイントに関する考慮事項 278
Braket と PrivateLink を設定する 278
エンドポイントの作成に関する追加情報 280
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 280
ログ記録とモニタリング 282
Amazon Braket SDK からの量子タスクの追跡 283
Amazon Braket コンソールを使用した量子タスクのモニタリング 285
リソースのタグ付け 287
タグの使用 288
Amazon Braket でのタグ付けでサポートされているリソース 288
Amazon Braket API を使用したタグ付け 289
タグ指定の制限 289
Amazon Braket でタグを管理する 290
Amazon Braket での AWS CLI タグ付けの例 291
EventBridge による量子タスクのモニタリング 292
EventBridge で量子タスクのステータスをモニタリングする 292
Amazon Braket EventBridge イベントの例 294
CloudWatch によるメトリクスのモニタリング 295
Amazon Braket のメトリクスとディメンション 295
CloudTrail を使用した量子タスクのログ記録 296
CloudTrail の Amazon Braket 情報 296
Amazon Braket ログファイルエントリについて 297

(アドバンスト) ログ記録 300
クォータ 303
追加のクォータと制限 348
ドキュメント履歴 349
..... ccclix

Amazon Braket とは

> **Tip**
>
> 量子コンピューティングの基盤について説明します AWS。Amazon Braket Digital Learning Plan に登録し、一連の学習コースとデジタル評価を完了した後に独自のデジタルバッジを獲得します。

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、量子力学の法則を活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できる可能性があります。

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、および最大のメリットを得るためにリソースをいつ投資するか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調べて設計します。
- 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します。
- 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します。
- 概念実証アプリケーションを作成します。

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て、ノートブックと呼ばれる一連のサンプル環境をすばやく使い始めることができます。

Braket 開発には 3 つのステージがあります。

- ビルド - Braket は、フルマネージド型の Jupyter ノートブック環境を提供し、簡単に開始できます。Braket ノートブックには、Amazon Braket SDK などのサンプルアルゴリズム、リソース、開発者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、1 行のコード

を変更することで、量子アルゴリズムを構築し、さまざまな量子コンピュータやシミュレーターでテストして実行できます。

- テスト - Braket は、フルマネージド型の高性能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、基盤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、従来の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。
- Run - Braket は、さまざまなタイプの量子コンピュータへの安全なオンデマンドアクセスを提供します。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、事前のコミットメントもないため、個々のプロバイダーを通じてアクセスを調達する必要はありません。

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開発の初期段階にあります。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規模量子 (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が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。より良い量子誤差補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、従来の (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使用に役立ちます。

ハイブリッド量子アルゴリズムでは、量子処理装置 (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。バリエーション量子アルゴリズムでは、従来の最適化ルーチンは、機械学習トレーニングセットのエラー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ほぼ同じ方法で、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、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、さまざまな量子アルゴリズムを支援します。

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で牽引しています。

- 数値理論 - 因数分解や暗号を含む (例えば、Shoor のアルゴリズムは数値理論計算の主要な量子メソッドです)
- 最適化 — 制約の満足度、線形システムの解決、機械学習を含む
- オラキュラーコンピューティング — 検索、非表示のサブグループ、順序の検出結果を含む (例えば、Grover のアルゴリズムは、オラキュラーコンピューティングの主要な量子メソッドです)
- 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、特定の実用的な問題に加えて、多くの概念実証問題に既に適用できる機能とサンプルノートブックを提供します。

このセクションの内容:

- Amazon Braket の仕組み
- Amazon Braket の用語と概念
- コストの追跡と節約
- Amazon Braket の API リファレンスとリポジトリ
- 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、一連の学習コースとデジタル評価を完了した後に独自のデジタルバッジを獲得します。

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 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 QC 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、操作フィールドの時間とスペースの依存関係が含まれます。

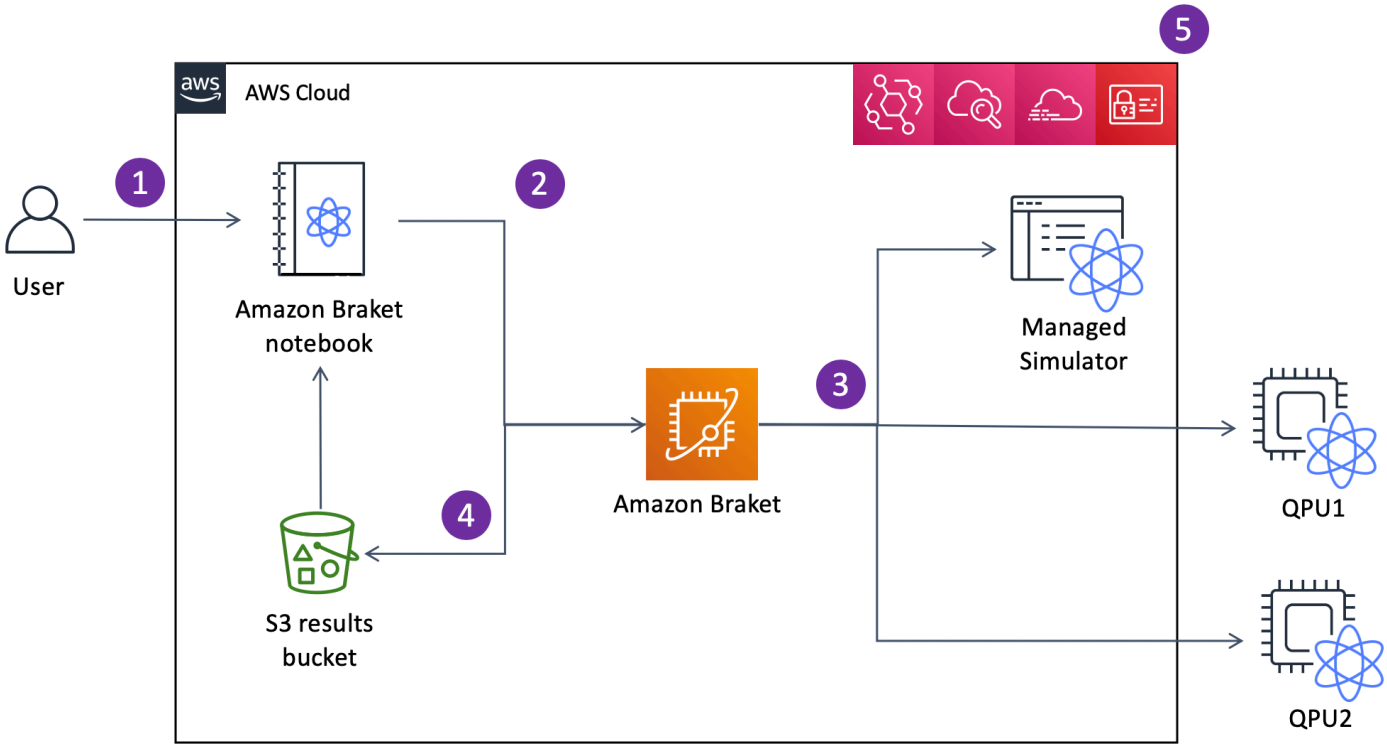
Braket Direct は、量子コンピューティングを探索する方法を拡張し AWS、研究とイノベーションを加速するプログラムです。さまざまな量子デバイスで専用容量を予約し、量子コンピューティングのスペシャリストと直接連携し、IonQ、Forte の最新のトラップオンデバイスを含む次世代機能に早期にアクセスできます。

このセクションでは、Amazon Braket で量子タスクを実行する大まかなフローについて説明します。

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、量子タスクを簡単に定義、送信Amazon Braketモニタリングできます。量子回路は SDK で直接構築できます。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

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 Amazon Braket API (2) に送信できます。選択したデバイスに応じて、量子タスクはデバイスが使用可能になるまでキューに入れられ、実装のためにタスクが QPU またはシミュレーターに送信されます (3)。Amazon Braket では、さまざまなタイプの QPUs (IonQ、IQM、QuEra、Rigetti) 、3 つのオンデマンドシミュレーター (SV1、DM1、TN1) 、2 つのローカルシミュレーター、1 つの埋め込みシミュレーターにアクセスできます。詳細については、「Amazon Braket がサポートされるデバイス」を参照してください。

量子タスクを処理すると、AmazonBraket は Amazon S3 バケットに結果を返します。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 は結果をバックグラウンドでポーリングし、量子タスクの完了時に Jupyter Notebook にロードします。Braket コンソールの量子タスクページで、または 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。 Amazon `GetQuantumTask` Amazon

Amazon Braket は AWS Identity and Access Management 、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 (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る量子タスクは、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、一連の学習コースとデジタル評価を完了した後に独自のデジタルバッジを獲得します。

Braket では、次の用語と概念が使用されます。

アナログハミルトニアシミュレーション

アナログハミルトンシミュレーション (AHS) は、多体システム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、ユーザーは時

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進化を直接エミュレートするように調整されます。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。これらは、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自然に実装されるため、AHS はアルゴリズムを回路として作成し、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

Braket サービスは、量子力学の標準表記であるブラケット表記にちなんで命名されました。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、量子コンピューティングスペシャリストに接続してワークロードのガイダンスを受け取り、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できます。

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 という機能があります。Braket ハイブリッドジョブは、次の 3 つのコンポーネントで構成されます。

1. アルゴリズムの定義。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ます。
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。 通常、1 つのハイブリッドジョブには、多くの量子タスクのコレクションが含まれています。

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 がサポートするデバイス」を参照してください。

エラーの軽減

エラーの軽減には、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が含まれます。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) では、回路ベースの QC とも呼ばれ、計算は基本オペレーション (ゲート) に分割されます。ゲートの特定のセットは汎用的です。つまり、すべての計算をそれらのゲートの有限シーケンス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、従来のデジタル回路のロジック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を指します。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)
```

ハミルトニア語

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まります。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の影響に関するすべての情報をエンコードします。N 量子ビットシステムのハミルトニアンは、一般的にクラシックマシン上の複雑な数値の 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ンシミュレーションを実行することで、これらの指数関数的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一時的な物理信号です。これは、キャリア信号のサポートとして機能し、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で再生される波形によって記述されます。お客様は、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、周波数と、量子ビットの |0# と |1# のエネルギーレベル間のエネルギー分離で、しばしば跳ね返るように選択されるフェーズ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、頻度、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、単一のサンプル解決で指定されます。

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、一連の量子ゲートであり、測定手順とともに、qubitレジスタ上の可逆的変換です。

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、量子サーキットの測定結果を計算するコンピュータプログラムです。一般的な回路の場合、量子シミュレーションのリソース要件は、シミュレートqubitsする 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 を介してアクセスAPI) 量子回路シミュレーターとローカル (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

量子コンピュータは、重ね合わせやもつれなどの量子力学現象を使用して計算を実行する物理デバイスです。ゲートベースの QC など、量子コンピューティング (QC) には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、量子タスクで実行できる物理量子コンピューティングデバイスです。QPUsは、ゲートベースの QC など、さまざまな QC パラダイムに基づくことができます。詳細については、Amazon Braket がサポートするデバイス」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 管理システムによってパルスを制御するため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[Devices] (デバイス) ページおよび Braket SDK から確認できます。

QPU でサポートされているゲート

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 で直接実行でき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。デバイスのサポートされているゲートは、Amazon Braket コンソールのデバイスページと 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 (測定手順と の数を含むshots) およびその他のリクエストメタデータが含まれます。量子タスクは、Amazon Braket SDK を通じて、または CreateQuantumTaskAPIオペレーションを直接使用して作成できます。量子タスクを作成すると、リクエストされたデバイスが使用可能になるまで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、または GetQuantumTask または SearchQuantumTasksAPIオペレーションを使用して表示できます。

Qubit

量子コンピュータの基本的な情報単位は、クラシックコンピューティングのビットと同様に qubit (量子ビット) と呼ばれます。qubit は、超伝導回路や個々のイオンや原子など、さまざまな物理実装によって実現できる 2 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、またはよりエキゾティックな量子システムに基づいています。

Queue depth

Queue depth 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を指します。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は、 Braket Software Development Kit (SDK)または からアクセスできますAmazon Braket Management Console。

1. タスクキューの深さとは、通常の優先度での実行を現在待機している量子タスクの合計数を指します。
2. 優先度タスクキューの深さは、 を通じて実行を待機している送信された量子タスクの合計数を指しますAmazon Braket Hybrid Jobs。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とは、デバイスで現在キューに入っているハイブリッドジョブの総数を指します。ハイブリッドジョブの一部としてQuantum tasks送信される には優先度があり、 に集約されますPriority Task Queue。

Queue position

Queue position は、それぞれのデバイスキュー内の量子タスクまたはハイブリッドジョブの現在の位置を指します。量子タスクまたはハイブリッドジョブの場合は、 Braket Software Development Kit (SDK)または を使用して取得できますAmazon Braket Management Console。

Shots

量子コンピューティングは本質的に確率的であるため、正確な結果を得るには、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。回路のショット数 (繰り返し実行) 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 および リソースへのアクセス許可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な以上のアクセス許可を持つのを防ぐことができます。たとえば、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、 内のすべてのバケットへのアクセス権を従業員に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 AWS アカウント。

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、ロールに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。イメージ、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 量子タスク結果など、任意の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。

コストの追跡と節約

> **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、一連の学習コースとデジタル評価を完了した後に独自のデジタルバッジを獲得します。

Amazon Braket を使用すると、事前のコミットメントなしに、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:

- 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

ほぼリアルタイムのコスト追跡

Braket SDK 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を提供します。各サンプルノートブックには、Braket の量子処理ユニット (QPUs) 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米ドルで表示され、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) タスクの使用状況に基づく見積もりです。表示される推定料金は、実際の料金とは異なる場合があります。推定料金は割引やクレジットを考慮しないため、Amazon Elastic Compute Cloud (Amazon EC2) などの他のサービスの使用に基づいて追加料金が発生する場合があります。

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、ベルステート回路を構築し、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、Bell 状態を定義して、 `Tracker()`関数を回路に追加します。

```
#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、ベルステートシミュレーションに次の出力が期待できます。トラッカー関数には、送信されたショットの数、完了した量子タスク、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は、最大しきい値がある場合があります。このようにして、コストトラッカーを使用して、実行コードにコスト管理ロジックを構築できます。次の例では、RigettiQPU で同じ回路を使用し、コストを 1 USD に制限します。コード内の回路の反復を 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を設定しました。したがって、コードスニペットは次の反復が 1 USD を超えるまで 3 回実行されます。通常、プログラムは希望する最大コストに達するまで反復し続けます。この場合、3 回反復します。

```
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}}}
0.9000000000 USD
```

> **Note**
>
> コストトラッカーは、失敗したTN1量子タスクの期間を追跡しません。TN1 シミュレーション中にリハーサルが完了し、収縮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微調整できます。
- 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。詳細については、「」を参照してくださいAWS Budgets。

ショット数が少ないTN1量子タスクをテストする

- シミュレーターのコストは QPUs よりも低くなりますが、量子タスクをショット数の多い状態で実行すると、特定のシミュレーターにコストがかかる場合があります。shot タスクを少数TN1でテストすることをお勧めします。 Shot カウントは、 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。

量子タスクのすべてのリージョンを確認する

- 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。送信された請求可能な量子タスクを検索する場合は、必ずすべてのリージョンを確認してください。
- 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 リファレンスとリポジトリ

> **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、一連の学習コースとデジタル評価を完了した後に独自のデジタルバッジを獲得します。

Amazon Braket には、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に使用できる APIs、SDKs、コマンドラインインターフェイスが用意されています。

- 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

Amazon Braket Tutorials GitHub リポジトリからコード例を取得することもできます。

- Braket チュートリアル GitHub

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Jupyterノートブックにコードを設定します。Jupyter ノートブックをセットアップしたら、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 (ステート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、Braket がサポートするプラグインや、以下に示すようにサードパーティーがサポートするプラグインが含まれます。

Amazon Braket がサポートされました。

- 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。それらをそのまま実行するか、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します。
- Braket-PennyLane プラグイン - Braket の QML フレームワークPennyLaneとして使用します。

サードパーティー (Braket チームによるモニタリングと貢献) :

- 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します。
- 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、一連の学習コースとデジタル評価を完了した後に独自のデジタルバッジを獲得します。

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 QPU またはシミュレーターを表します。Amazon Braket は、IonQ、、IQM、QuEraおよび から QPU デバイスへのアクセス、3 つのオンデマンドシミュレータRigetti一、3 つのローカルシミュレーター、1 つの埋め込みシミュレーターを提供します。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の追加のデバイスプロパティは、Amazon Braket コンソールのデバイスタブまたは `GetDevice` API を使用して確認できます。シミュレーターを使用して回路を構築する場合、Amazon Braket では現在、連続した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。Amazon Braket SDK を使用している場合は、次のコード例に示すように、デバイスプロパティにアクセスできます。

```
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:us-east-1::device/qpu/ionq/Aria-1')
 #IonQ Aria-1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Aria-2')
 #IonQ Aria-2
# 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: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
```

サポートされている量子ハードウェアプロバイダー

- IonQ
- IQM
- QuEra Computing
- Rigetti

サポートされているシミュレーター

- ローカル状態ベクトルシミュレーター (braket_sv) ('Default Simulator')
- 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
- ステート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)
- Tensor ネットワークシミュレーター (TN1)
- PennyLane の Lightning Simulators

量子タスクに最適なシミュレーターを選択する

- シミュレーターを比較する

Amazon Braket デバイス

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
IonQ	Aria-1	ゲートベース	QPU	arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1	us-east-1
IonQ	Aria-2	ゲートベース	QPU	arn:aws:braket:us-east-1::device/qpu/ionq/Aria-2	us-east-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
IQM	Garnet	ゲートベース	QPU	arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet	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

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	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	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

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 ハードウェアプロバイダー」を参照してください。

Amazon Braket のリージョンとエンドポイント

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

Amazon Braket のリージョンの可用性

リージョン名	リージョン	Braket エンドポイント	QPUsとシミュレーター
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)	IonQ、QuEra、SV1, DM1, TN1
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)	Rigetti、SV1, DM1
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)	SV1, DM1, TN1
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)	IQM

リージョン名	リージョン	Braket エンドポイント	QPUsとシミュレーター
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)	SV1, DM1, TN1

QPU デバイスで実行される量子タスクは、そのデバイスの リージョンの 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、量子タスクを任意の QPU デバイスに送信できます。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。

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

がリージョンとエンドポイントとどのように AWS 連携するかに関する一般的な情報については、AWS 「 全般のリファレンス」の「 AWS のサービス エンドポイント」を参照してください。

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、一連の学習コースとデジタル評価を完了した後に独自のデジタルバッジを獲得します。

Amazon Braket を有効にする」の手順に従ったら、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 AWS 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、一連の学習コースとデジタル評価を完了した後に独自のデジタルバッジを獲得します。

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。これらのアクセス許可は IAM AmazonBraketFullAccess 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 へのアクセスを許可するには、AmazonBraketFullAccess ポリシーをアタッチするか、作成したカスタムポリシーをアタッチして、ユーザーに許可を付与します。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から、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は、次の 3 つのステップを順を追って説明します。
   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 ノートブック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。契約の条項は、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 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、サードパーティーデバイスの有効化契約に同意せずに使用できます。

サードパーティーのハードウェアにアクセスする場合、これらの条件に同意してサードパーティーのデバイスを使用できるようにするには、アカウントごとに 1 回のみ実行する必要があります。

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、一連の学習コースとデジタル評価を完了した後に独自のデジタルバッジを獲得します。

Amazon Braket には、フルマネージド型の Jupyter ノートブックが用意されており、使用を開始できます。Amazon Braket ノートブックインスタンスはAmazon SageMakerノートブックインスタンスに基づいています。次の手順では、新規および既存のお客様の新しいノートブックインスタンスを作成する方法の概要を説明します。

Amazon Braket の新規のお客様：

1. Amazon Braket コンソールを開き、左側のペインのダッシュボード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」モーダルの開始方法をクリックします。 Amazon Braket ノートブック名を指定して、デフォルトの Jupyter ノートブックを作成します。
   a. ノートブックの作成には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表示され、ステータスは保留中です。
   c. ノートブックインスタンスを使用する準備ができると、ステータスは InService に変わります。
   d. ページを更新して、ノートブックの更新ステータスを表示します。

既存の Amazon Braket のお客様：

1. Amazon Braket コンソールを開き、左側のペインで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. ノートブックがゼロの場合は、標準セットアップを選択してデフォルトの Jupyter ノートブックを作成します。

3. 英数字とハイフンのみを使用してノートブックインスタンス名を入力し、任意のビジュアルモードを選択します。

4. ノートブックの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、ストレージ料金は続行されます。

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 タブを選択し、Inactivity Manager タブを選択します。

> **Note**
>
> 作業を保存するには、SageMaker AI ノートブックインスタンスを Git リポジトリと統合します。または、作業を `/Braket Algorithms`および `/Braket Examples`フォルダの外部に移動して、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. ノートブック設定で、インスタンスタイプを選択します。

i. 標準で費用対効果の高いインスタンスタイプ ml.t3.medium がデフォルトで選択されています。インスタンスの料金の詳細については、Amazon SageMakerの料金」を参照してください。

b. パブリック 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 からパブリック git リポジトリのクローンを作成します。 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とアクセス設定を構成します。

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識別します。起動 をクリックします。

> **Note**
>
> Amazon Braket および 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。追加の Amazon Braket ノートブック設定は、SageMaker コンソールから利用できます。

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 を実行するには、 コマンドを実行するとき、`pip install amazon-braket-sdk`または PennyLane プラグイン`pip install amazon-braket-pennylane-plugin`の コマンドを実行するときに SDK とプラグインをインストールします。

を使用して Braket ノートブックインスタンスを作成する AWS 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、一連の学習コースとデジタル評価を完了した後に独自のデジタルバッジを獲得します。

AWS 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 上に構築されています。CloudFormation を使用すると、意図した設定を記述す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 形式で記述されます。インスタンスは、順序的かつ繰り返し可能な方法で作成、更新、削除できます。これは、 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便利です AWS アカウント。

Braket ノートブックの CloudFormation テンプレートを作成したら、 AWS 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、「 AWS CloudFormation ユーザーガイド」の「 AWS 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します。
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. プレフィックスを持つ SageMaker AI ノートブックインスタンスを作成する **amazon-braket-**

ライフサイクル設定は、作成したすべての Braket ノートブックで再利用できます。同じ実行アクセス許可を割り当てる Braket ノートブックの IAM ロールを再利用することもできます。

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: プレフィックスが付いた SageMaker AI ノートブックインスタンスを作成する amazon-braket-

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、Braket 用の SageMaker AI ノートブックインスタンスをカスタマイズします。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、AWS 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、SageMaker AI はユーザーに代わってオペレーションを実行します。たとえば、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、SageMaker AI は Braket で オペレーションを実行します。ノートブック実行ロールは、SageMaker AI がユーザーに代わって実行できる正確なオペレーションを定義します。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。 Amazon SageMaker

次の例を使用して、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します。必要に応じてポリシーを変更できます。

> **Note**
>
> ロールに、 というプレフィックスが付いた Amazon S3 バケットに対する `s3:ListBucket`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ますbraketnotebookcdk-"。ライフサイクル設定スクリプトには、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のこれらのアクセス許可が必要です。

```
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: プレフィックスが付いた SageMaker AI ノートブックインスタンスを作成する **amazon-braket-**

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 ロールを使用して、SageMaker AI ノートブックインスタンスを作成します。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 コンソールからアクセスできます。この CloudFormation リソースの設定オプションの詳細については、 AWS 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 を使用して量子タスクを構築する

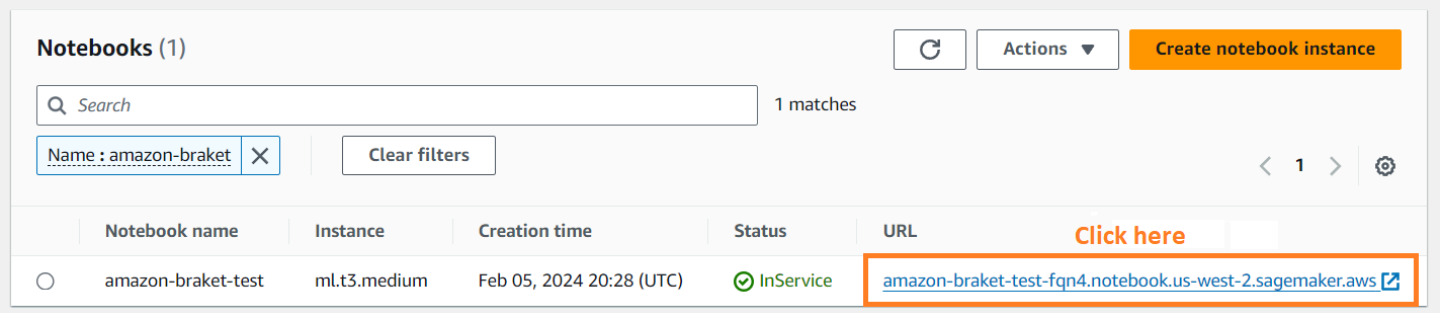
Braket は、フルマネージド型の Jupyter ノートブック環境を提供し、簡単に開始できます。Braket ノートブックには、Amazon Braket SDK などのサンプルアルゴリズム、リソース、開発者ツールがプリインストールされています。Amazon Braket SDK を使用すると、1 行のコードを変更することで、量子アルゴリズムを構築し、さまざまな量子コンピュータやシミュレーターでテストして実行できます。

このセクションの内容:

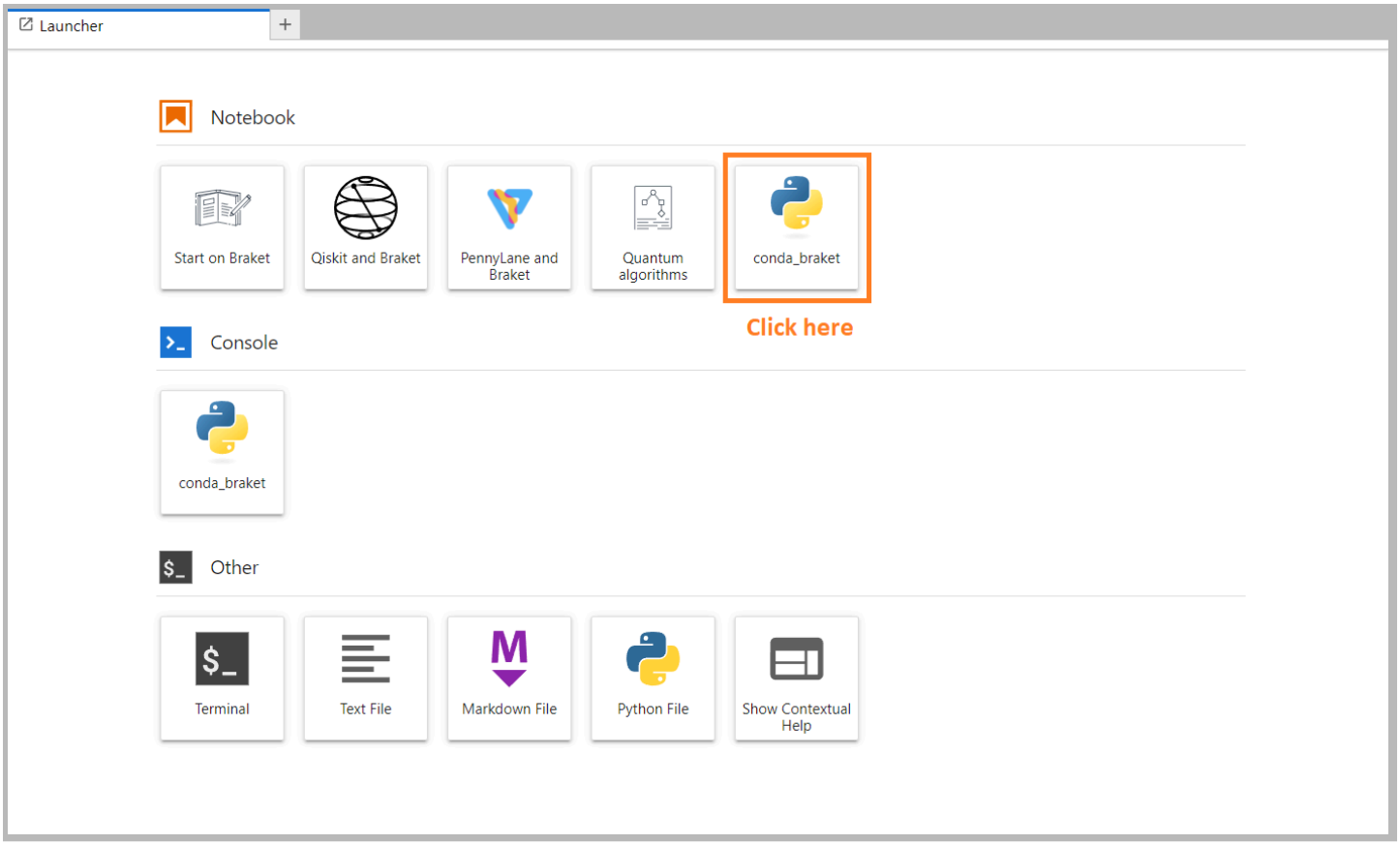
- 最初の回路の構築
- エキスパートのアドバイスを受ける
- Amazon Braket Hybrid Jobs の開始方法
- OpenQASM 3.0 で回路を実行する
- 実験機能を調べる
- Amazon Braket でのパルス制御
- アナログハミルトニアシミュレーション
- AWS Boto3 の使用

最初の回路の構築

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、単純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)
```

次のコマンドで回路を視覚化できます。

```
print(bell)
```

[Run your circuit on the local simulator] (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。より小さな回路にはローカルシミュレーターを使用することをお勧めします。これは最大 25 です qubits (ローカルハードウェアによって異なります) 。

ローカルシミュレーターをインスタンス化する方法は次のとおりです。

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

そして、回路を実行します。

```
# 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、予想どおり、測定結果として 00 と 11 のほぼ等しい (shotノイズまで) 分布が見つかります。

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高性能シミュレーター へのアクセスも提供します。 SV1はSV1、最大 34 個の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ステートベクトルシミュレーターですqubits。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。 (およびTN1任意の QPU SV1 で) 量子タスクを実行すると、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 によってデフォルトのバケットが作成されますamazon-braket-{region}-{accountID}。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 amzn-s3-demo-bucket が表示されます。Amazon Braket のバケット名は常に で始まり、その後に追加した

他の識別文字amazon-braket-が続きます。S3 バケットの設定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 = "amzn-s3-demo-bucket"
# the name of the folder in the bucket
my_prefix = "simulation-output"
s3_folder = (my_bucket, my_prefix)
```

で回路を実行するにはSV1、.run( )呼び出しで位置引数として以前に選択した 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。コンソールで量子タスクタブに移動し、量子タスクがリストの上部にある必要があります。または、一意の量子タスク ID または他の基準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たすべての量子タスク IDs と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 では、1 行のコードを変更するだけで、前の量子回路の例を物理量子コンピュータで実行できます。Amazon Braket は、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、デバイスタブの AWS コンソールで確認できます。次の例は、IQMデバイスをインスタンス化する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、次のコードのIonQデバイスを選択します。

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

デバイスを選択した後、ワークロードを実行する前に、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て、量子タスクまたはハイブリッドジョブの数を判断できます。さらに、お客様は のデバイスページでデバイス固有のキューの深さを表示できますAmazon Braket Management Console。

```
# 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)
'2' # returns the number of hybrid jobs queued on the device
```

タスクを実行すると、Amazon Braket SDK は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) 。次の例に示すように、 .run( ) コマンドで poll_timeout_secondsパラメータを変更することで、このデフォルトを変更できます。ポーリングタイムアウトが短すぎる場合、QPU が使用できず、ローカルタイムアウトエラーが返されるなど、ポーリング時間内に結果が返されない可能性があることに注意してください。task.result( ) 関数を呼び出して、ポーリングを再開できます。

```
# 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)
```

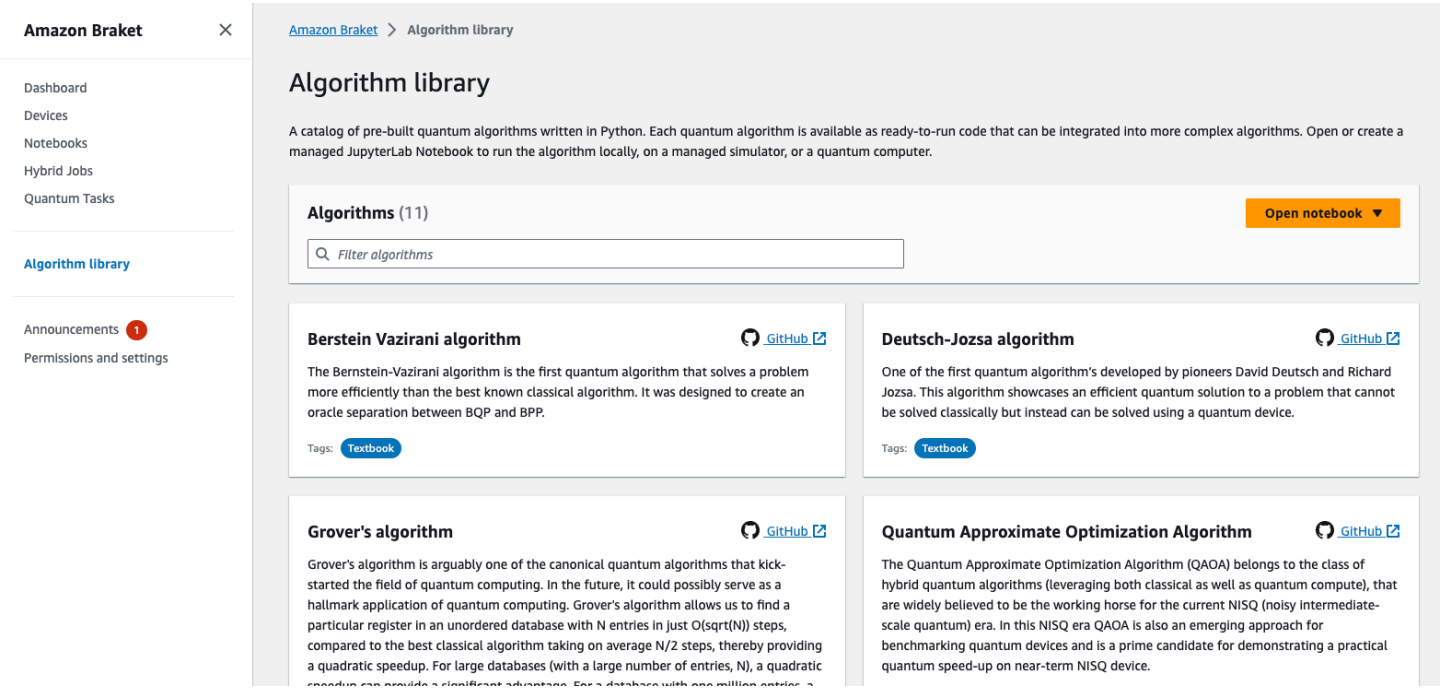
さらに、量子タスクまたはハイブリッドジョブを送信した後、 queue_position( )関数を呼び出してキューの位置を確認できます。

```
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をそのまま実行するか、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します。Braket コンソールからアルゴリズムライブラリにアクセスできます。詳細については、Braket Github アルゴリズムライブラリ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説明が表示されます。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。ノートブックオプションを選択すると、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、定義されたとおりに回路を実行するようにコンパイラに指示し、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の手順も含まれています。

特定の QPUs を持つさまざまなゲートに対して Braket の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、Amazon Braket でのパルス制御」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
- 部分的な測定
- 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、標準単一量子ビットのハダマードゲートと 2 量子ビットの CNOT ゲートで構成される 4 つの qubits (q0q1、q2、、および とラベル付けされたq3) サンプル回路を定義します。次の例に示すように、 `print`関数を呼び出すことで、この回路を視覚化できます。

```
# 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-C---
        |
q1 : -H-|-C-
        | |
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、空き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して新しい回路を作成するか、回路を送信するときに、特定のデバイスで量子タスクとして実行できます。

```
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回路を作成するには、次のように単一の float (すべての空きパラメータが取る値) または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)
```

my_circuit は変更されていないため、固定パラメータ値を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、制御修飾子と電力修飾子を使用するゲートを持つ回路を定義します。これらの変更を使用して、制御されたゲートなどの新しいRy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、AmazonBraket 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', 'GPi', 'GPi2', 'H', 'I', 'ISwap', 'MS', 'PSwap',
 'PhaseShift', 'PulseGate', 'Rx', 'Ry', 'Rz', 'S', 'Si', 'Swap', 'T', 'Ti', '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。たとえば、を呼び出して`circ.h(0)`、最初の にハダマードゲートを追加しますqubit。

> **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 (次のソースコードを参照) でも、 `targets`パラメータでqubits定義された に適用されるマルチ量子ビットゲートでもかまいません。

```
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、量子デバイスで実行する量子タスクを記述する量子ディレクティブです。 `Instruction`演算子には、 タイプのオブジェクト`Gate`のみが含まれます。

```
# 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])
new_instr = instr.copy(target_mapping={0: 5})
```

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 デバイスを使用した例を次に示します。

```
# 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 the Aria-1 device:
 ['x', 'y', 'z', 'rx', 'ry', 'rz', 'h', 'cnot', 's', 'si', 't', 'ti', 'v', 'vi', 'xx',
 'yy', 'zz', 'swap']
```

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、AmazonBraket はこ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

Braket コンソールのデバイスページで忠実度情報を表示できます。プログラムで同じ情報にアクセスすると役立つ場合があります。次のコードは、QPU の 2 つのqubitゲート間で 2 つのゲート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}")
```

部分的な測定

前の例に従って、量子回路内のすべての量子ビットを測定しました。ただし、個々の量子ビットまたは量子ビットのサブセットを測定できます。

例: 量子ビットのサブセットを測定する

この例では、ターゲット量子ビットを持つmeasure命令を回路の最後に追加することで、部分的な測定を示します。

```
# 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

から量子コンピュータで量子回路を実行する場合Rigetti、オプションで手動qubit割り当てを使用して、アルゴリズムqubitsに使用される を制御できます。Amazon Braket コンソールと Amazon Braket SDK を使用すると、選択した量子処理ユニット (QPU) デバイスの最新のキャリブレーションデータを検査できるため、実験qubitsに最適なものを選択できます。

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プロパティを調査できます。研究者と上級ユーザーは、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、qubits明示的に を割り当てる方法を示しています。

```
circ = Circuit().h(0).cnot(0, 7)  # Indices of actual qubits in the QPU
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、定義されたとおりに回路を実行するようにコンパイラに指示できます。逐語的なコンパイルを使用して、回路全体を指定されたとおりに正確に保持するか、その特定の部分のみを保持するか (Rigettiでのみサポート) を指定できます。ハードウェアベンチマークまたはエラー軽減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しているゲートと回路レイアウトを正確に指定するオプションが必要です。逐語的なコンパイルでは、特定の最適化ステップをオフにしてコンパイルプロセスを直接制御できるため、回路が設計どおりに動作します。

逐語的なコンパイルは現在、Rigetti、、および IQM デバイスでサポートされておりIonQ、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、接続時にゲートが呼び出され、回路qubits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することを確認することをお勧めします。次の例は、デバイスでサポートされてい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方法を示しています。

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

の場合Rigetti、逐語的なコンパイルで使用するdisableQubitRewiring=Trueように を設定して、qubit再ワイヤリングをオフにする必要があります。コンパイルで逐語的なボックスを使用するときに disableQubitRewiring=Falseが設定されている場合、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。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ついては、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IonQデバイスの回路を記述していて、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 verbatimdisableQubitRewiringフラグが完全に無視されるため、 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には、次のようにバックエンドを変更できます。

```
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下部から構築します。
2. 既存のノイズのない回路を取り、全体にノイズを挿入します。

次の例は、脱分極ノイズとカスタム Kraus チャネルを持つシンプルな回路を使用するアプローチを示しています。

```
# 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)
```

```
# 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)
```

回路の実行は、次の 2 つの例に示すように、前と同じユーザーエクスペリエンスです。

例 1

```
task = device.run(circ, s3_location)
```

Or

例 2

```
task = device.run(circ_noise, s3_location)
```

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 は、 ごとに 1 つのゲートを体験qubitできますMoment。の目的は、回路とそのゲートに対処しやすくし、一時的な構造を提供することMoments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。次の例circuit.depthに示すように、 メソッドを呼び出す回路深度を表示できます。

```
# 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)-C----------X-
                |
q1 : -Ry(0.2)--|-ZZ(0.15)---
                | |
q2 : ----------X-|----------
                  |
q3 : ------------ZZ(0.15)---

T  : |   0    |    1     |2|
Total circuit depth: 3
```

上記の回路の合計回路深度は 3 (モーメント 0、1、) です2。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、キーと値のペアのディクショナリとして機能します。

- キーは でMomentsKey()、擬似時間とqubit情報が含まれています。
- 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)]))
Instruction('operator': Rx('angle': 0.15, 'qubit_count': 1), 'target':
 QubitSet([Qubit(0)]))

MomentsKey(time=0, qubits=QubitSet([Qubit(1)]))
Instruction('operator': Ry('angle': 0.2, 'qubit_count': 1), 'target':
 QubitSet([Qubit(1)]))

MomentsKey(time=1, qubits=QubitSet([Qubit(0), Qubit(2)]))
Instruction('operator': CNot('qubit_count': 2), 'target': QubitSet([Qubit(0),
 Qubit(2)]))

MomentsKey(time=1, qubits=QubitSet([Qubit(1), Qubit(3)]))
Instruction('operator': ZZ('angle': 0.15, 'qubit_count': 2), 'target':
 QubitSet([Qubit(1), Qubit(3)]))

MomentsKey(time=2, qubits=QubitSet([Qubit(0)]))
Instruction('operator': X('qubit_count': 1), 'target': QubitSet([Qubit(0)]))
```

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。

```
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 : ---C-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。回路は次のタイプの結果を返すことができます。

- AdjointGradient は、指定されたオブザーバブルの期待値の勾配 (ベクトル派生) を返します。このオブザーバブルは、ジョイントの区別方法を使用して、指定されたパラメータに関して指定されたターゲットに対して動作します。このメソッドは、shots=0 の場合にのみ使用できます。
- Amplitude は、出力波関数で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、特定のオブザーバブルの期待値を返します。この値は、この章の後半で導入されたObservableクラスで指定できます。オブザーバブルの測定qubitsに使用されるターゲットを指定する必要があります。また、指定されたターゲットの数は、オブザーバブルqubitsが動作するの数と等しくなければなりません。ターゲットを指定しない場合、オブザーバブルは 1 でのみ動作qubitし、すべてqubitsに並行して適用されます。
- Probability は、計算基準の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、指定されたベースベクトルのわずかな確率のみqubitsが返されます。マネージドシミュレーターと QPUs 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、 のqubitsシステムから指定されたターゲットのサブシステムの密度マトリックスを返しますqubits。この結果タイプのサイズを制限するため、Braket はターゲットの数qubitsを最大 8 に制限します。
- StateVector はフルステート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セットとオブザーバブルの測定数を返します。ターゲットを指定しない場合、オブザーバブルは 1 でのみ動作qubitし、すべてqubitsに並行して適用されます。ターゲットを指定する場合、指定されたターゲットの数は、オブザーバブルqubitsが動作する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セットとオブザーバブルの分散 (mean([x-mean(x)]^2)) をリクエストされた結果タイプとして返します。ターゲットを指定しない場合、オブザーバブルは 1 でのみ動作qubitし、すべてqubitsに並行して適用されます。それ以外の場合、

指定されたターゲットの数は、qubitsオブザーバブルを適用できる の数と等しくなければなりません。

さまざまなデバイスでサポートされる結果タイプ:

	ローカルシム	SV1	DM1	TN1	Rigetti	IonQ	IQM
アドジョイントグラデーション	N	Y	N	N	N	N	N
Amplitude	Y	Y	N	N	N	N	N
期待	Y	Y	Y	Y	Y	Y	Y
見込み	Y	Y	Y	N	Y	Y	Y
低密度マトリックス	Y	N	Y	N	N	N	N
状態ベクトル	Y	N	N	N	N	N	N
サンプル	Y	Y	Y	Y	Y	Y	Y
分散	Y	Y	Y	Y	Y	Y	Y

次の例に示すように、デバイスのプロパティを調べることで、サポートされている結果タイプを確認できます。

```
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
```

を呼び出すにはResultType、次の例に示すように、回路に追加します。

```
from braket.circuits import 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**
>
> デバイスによっては、結果として測定値 (などRigetti) を提供し、結果として確率 (など) を提供するデバイスもありますIonQ。SDK は結果に測定プロパティを提供しますが、確率を返すデバイスの場合、事後計算されます。したがって、 が提供するようなデバイスはIonQ、ショットごとの測定値が返されないため、確率によって決定された測定値を持ちます。このファイルに示すように、結果オブジェクトmeasurements_copied_from_deviceで を表示することで、結果が事後計算されているかどうかを確認できます。

オブザーバブル

Amazon Braket には、測定するオブザーバブルを指定するために使用できる Observable クラスが含まれています。

各 には、最大 1 つの一意の非 ID オブザーバブルを適用できますqubit。同じ に 2 つ以上の異なる非 ID オブザーバブルを指定するとqubit、エラーが表示されます。この目的のために、テンソル製品の各要素は個々のオブザーバビリティとしてカウントされるため、同じ で動作する複数のテンソル製品を持つことは許可されます。ただしqubit、その で動作する要素が同じであることが条件qubitです。

オブザーバブルをスケーリングし、オブザーバブル (スケーリングの有無にかかわらず) を追加することもできます。これにより、AdjointGradient結果タイプSumで使用できる 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ます。

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

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

# get the tensor product of 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())

# also 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、「コンストラクト 1 回 - 複数回実行」方式で使用でき、勾配を計算するために使用できる無料のパラメータが含まれている場合があります。フリーパラメータには文字列エンコードされた名前があり、値を指定したり、値に関して区別するかどうかを決定したりできます。

```
from braket.circuits import Circuit, FreeParameter, Observable
theta = FreeParameter("theta")
phi = FreeParameter("phi")
circ = Circuit().h(0).rx(0, phi).ry(0, phi).cnot(0, 1).xx(0, 1, theta)
circ.adjoint_gradient(observable=Observable.Z() @ Observable.Z(), target=[0, 1],
 parameters = ["phi", theta]
```

区別するパラメータには、名前 (文字列として) または直接参照を使用して指定します。AdjointGradient 結果タイプを使用した勾配の計算は、観測可能な の期待値に関して行われることに注意してください。

注： パラメータ化された回路に引数として渡して空きパラメータの値を修正した場合、結果タイプAdjointGradientとして とパラメータを指定して回路を実行すると エラーが発生します。これは、 との差別化に使用しているパラメータが存在しないためです。次の例を参照してください。

```
device.run(circ(0.2), shots=0) # will error, as no free parameters will be present
device.run(circ, shots=0, inputs={'phi'=0.2, 'theta'=0.2) # will succeed
```

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専門家に直接接続して、ワークロードに関する追加のガイダンスを取得します。

Braket Direct を使用してエキスパートアドバイスオプションを調べるには、Braket コンソールを開き、左側のペインで Braket Direct を選択し、エキスパートアドバイスセクションに移動します。以下のエキスパートアドバイスオプションを使用できます。

- Braket の営業時間： Braket の営業時間は 1:1 セッションで、先着順で毎月行われます。利用可能な各オフィス時間スロットは 30 分で、無料です。Braket の専門家に相談すると、use-case-to-device適合を探索し、アルゴリズムに Braket を最大限に活用するための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の特定の Braket 機能の使用方法に関するレコメンデーションを取得することで、アイデアから実行までの時間を短縮できます。
  - Braket の営業時間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。
  - 利用可能な次のスロットへのカレンダーの招待が 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 フォーラムまたは Quantum Computing Stack Exchange を使用することもできます。ここでは、以前に回答し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提供： IonQ、QuEra、および Rigettiはそれぞれ、 を通じてプロフェッショナルサービスを提供します AWS Marketplace。
  - サービスを確認するには、接続を選択し、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 量子ソリューションラボ (QSL): QSL は、量子コンピューティングを効果的に探索し、このテクノロジーの現在のパフォーマンスを評価するのに役立つ量子コンピューティングの専門家がいる共同研究およびプロフェッショナルサービスチームです。
  - QSL に連絡するには、接続を選択し、連絡先情報とユースケースの詳細を入力します。
  - QSL チームは、次のステップについて E メールで連絡します。

Amazon Braket Hybrid Jobs の開始方法

このセクションでは、コンポーネントに関する情報と、Amazon Braket でハイブリッドジョブを設定する方法について説明します。

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:

- ハイブリッドジョブとは
- Amazon Braket Hybrid Jobsを使用する時期
- 入力、出力、環境変数、およびヘルパー関数
- 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

ハイブリッドジョブとは

Amazon Braket Hybrid Jobs は、従来の 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、リクエストされたクラシックリソースをスピンアップし、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、使用した分に対してのみ料金が発生します。

Hybrid Jobs は、従来の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 では、アルゴリズムを実行してから、Braket はスケーラブルでコンテナ化された環境でアルゴリズムを実行します。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 QPU デバイスへの優先度の高いキューイングの利点があります。この優先順位付けにより、量子計算が処理され、キューで待機している他のタスクよりも先に実行されます。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に特に便利です。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)、バリエーション量子固有ソルバー、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、トレーニング損失や期待値などのカスタムメトリクスを追跡できます。

Amazon Braket Hybrid Jobsを使用する時期

Amazon Braket Hybrid Jobs を使用すると、さまざまな量子固有ソルバー (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。このアルゴリズムは、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、現在の量子システムのパフォーマンスを最適化します。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。ジョブの実行中に、選択したターゲット QPU に優先的にアクセスできます。ジョブのタスクは、デバイスでキューに入れられた他のタスクの前に実行されます。これにより、ハイブリッドアルゴリズムのランタイムが短くなり、予測しやすくなります。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。フリーパラメータを使用して回路を送信でき、Braket は回路を 1 回コンパイルします。後続のパラメータ更新を同じ回路に再コンパイルする必要がないため、ランタイムがさらに短縮されます。
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指定し、実行する量子デバイス (量子処理ユニットまたはシミュレーター) を選択するだけです。Amazon Braket は、ターゲットデバイスが使用可能になるまで待機し、クラシックリソースをスピンアップして、構築済みのコンテナ環境でワークロードを実行し、結果を Amazon Simple Storage Service (Amazon S3) に返し、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。

入力、出力、環境変数、およびヘルパー関数

完全なアルゴリズムスクリプトを構成する ファイルに加えて、ハイブリッドジョブには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、AmazonBraket はハイブリッドジョブの作成の一部として提供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、アルゴリズム中に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。

このセクションでは、Amazon Braket Python SDK が提供する `AwsQuantumJob.create`関数の主要な概念と、コンテナファイル構造へのマッピングについて説明します。

このセクションの内容:

- 入力
- 出力
- 環境変数
- ヘルパー関数

入力

入力データ: 入力データは、 `input_data`引数でディクショナリとして設定された入力データファイルを指定することで、ハイブリッドアルゴリズムに提供できます。ユーザーは SDK の `AwsQuantumJob.create`関数内で引`input_data`数を定義します。これにより、環境変数 によって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`"AMZN_BRAKET_INPUT_DIR"`。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、Amazon Braket Hybrid Jobs を使用した QAOA」および「Amazon Braket Hybrid Jobs Jupyter ノートブック」のPennyLane and Quantum machine learning」を参照してください。Amazon Braket

> **Note**
>
> 入力データが大きい場合 (>1GB)、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、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、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、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ます。

ハイパーパラメータ: 通り過ぎたらhyperparametersの場合、環境変数で使用できます。"AMZN_BRAKET_HP_FILE"。

> **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を使用する」セクションとこの 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使用するチェックポイントjob-arnを持つ 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checkpoint_configs3Uriの にコピーし、ジョブの実行AMZN_BRAKET_CHECKPOINT_DIR中に環境変数によって指定されたパスで利用可能にします。デフォルトは です。つまりNone、別のハイブリッドジョブのチェックポイントデータは新しいハイブリッドジョブでは使用されません。

出力

量子タスク: 量子タスクの結果は S3 の場所 に保存されますs3://amazon-braket-<region>-<accountID>/jobs/<job-name>/tasks。

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。この値を指定していない場合は、デフォルトで s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/<timestamp>/data になります。SDK **save_job_result** ヘルパー関数 を提供しています。これを使用すると、アルゴリズムスクリプトから呼び出されたときに、ディクショナリの形式で結果を簡単に保存できます。

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: アルゴリズムメトリクスは、ハイブリッドジョブの実行中に Amazon CloudWatch に出力され、AmazonBraket 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として定義できます。アルゴリズムメトリクスの使用方法の例については、Amazon Braket Hybrid Jobs を使用して QAOA アルゴリズムを実行する」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を一覧表示します。

- 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含むファイル。
- 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 で指定されている出力 Amazon S3 バケットOutputDataConfig。
- AMZN_BRAKET_SCRIPT_ENTRY_POINT – CreateJobリクエストの で指定されたエントリポイントScriptModeConfig。
- AMZN_BRAKET_SCRIPT_COMPRESSION_TYPE – CreateJobリクエストの で指定された圧縮タイプScriptModeConfig。
- AMZN_BRAKET_SCRIPT_S3_URI – CreateJobリクエストの で指定されたユーザーのスクリプトの Amazon S3 の場所ScriptModeConfig。
- AMZN_BRAKET_TASK_RESULTS_S3_URI – SDK がジョブの量子タスク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 で指定されている、ジョブ結果が保存される Amazon S3 の場所OutputDataConfig。
- AMZN_BRAKET_JOB_TOKEN – ジョブコンテナで作成された量子タスクの 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。

```
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() # save hybrid job results
save_job_checkpoint() # save a checkpoint
load_job_checkpoint() # load a previously saved checkpoint
```

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 3 つの環境をサポートしています。

- ベースコンテナ (が指定されていない場合`image_uri`はデフォルト)
- Tensorflow と PennyLane を持つコンテナ
- PyTorch と PennyLane を含むコンテナ

次の表に、コンテナとそのコンテナに含まれるライブラリの詳細を示します。

Amazon Braket コンテナ

タイプ	TensorFlow を使用した PennyLane	PyTorch を使用した PennyLane	ペンニラン
基本	292282985366.dkr.ecr.us-east-1.amazonaws.com/amazon-braket-tensorflow-jobs:latest	292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:latest	292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:latest
継承ライブラリ	• awscli • numpy • pandas • scipy	• awscli • numpy • pandas • scipy	
追加のライブラリ	• amazon-braket-default-simulator • amazon-braket-PennyLane-plugin	• amazon-braket-default-simulator • amazon-braket-PennyLane-plugin	• amazon-braket-default-simulator • amazon-braket-PennyLane-plugin

タイプ	TensorFlow を使用した PennyLane	PyTorch を使用した PennyLane	ペンニラン
	• amazon-braket-schemas • amazon-braket-sdk • ipykernel • keras • matplotlib • networkx • openbabel • PennyLane • protobuf • psi4 • rsa • PennyLane-Lightning-gpu • cuQuantum	• amazon-braket-schemas • amazon-braket-sdk • ipykernel • keras • matplotlib • networkx • openbabel • PennyLane • protobuf • psi4 • rsa • PennyLane-Lightning-gpu • cuQuantum	• amazon-braket-schemas • amazon-braket-sdk • awscli • boto3 • ipykernel • matplotlib • networkx • numpy • openbabel • pandas • PennyLane • protobuf • psi4 • rsa • scipy

オープンソースのコンテナ定義は、aws/amazon-braket-containers で表示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。コンテナは、ハイブリッドジョブを呼び出す AWS リージョン 元の にある必要があります。ハイブリッドジョブの作成時にコンテナイメージを指定するには、ハイブリッドジョブスクリプトの`create(…)` 呼び出しに次の 3 つの引数のいずれかを追加します。Amazon Braket コンテナにはインターネット接続があるため、ランタイム時に (スタートアップまたはランタイムのコストで) 選択したコンテナに追加の依存関係をインストールできます。次の例は、us-west-2 リージョン用です。

- ベースイメージ image_uri"292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:1.0-cpu-py39-ubuntu22.04"
- Tensorflow Image Image_uri"292282985366.dkr.ecr.us-east-1.amazonaws.com/amazon-braket-tensorflow-jobs:2.11.0-gpu-py39-cu112-ubuntu20.04"
- PyTorch image image_uri"292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:1.13.1-gpu-py39-cu117-ubuntu20.04"

は、AmazonBraket SDK の retrieve_image( )関数を使用して取得image-uris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_tf = retrieve_image(Framework.PL_TENSORFLOW, "us-west-2")
image_uri_pytorch = retrieve_image(Framework.PL_PYTORCH, "us-west-2")
```

ハイパーパラメータの使用

ハイブリッドジョブを作成するときに、学習レート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、それら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。値は文字列データ型である必要があります。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ます。ハイパーパラメータを使用する最初のステップは、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。これは、次のコードで確認できます。

```
#defining the number of qubits used
n_qubits = 8
#defining the number of layers used
n_layers = 10
#defining the number of iterations used for your optimization algorithm
n_iterations = 10

hyperparams = {
    "n_qubits": n_qubits,
    "n_layers": n_layers,
    "n_iterations": n_iterations
}
```

次に、上記のコードスニペットで定義されたハイパーパラメータを渡して、選択したアルゴリズムで次のようになります。

```
import time
from braket.aws import AwsQuantumJob

#Name your job so that it can be later identified
job_name = f"qcbm-gaussian-training-{n_qubits}-{n_layers}-" + str(int(time.time()))
```

```
job = AwsQuantumJob.create(
    #Run this hybrid job on the SV1 simulator
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
    #The directory or single file containing the code to run.
    source_module="qcbm",
    #The main script or function the job will run.
    entry_point="qcbm.qcbm_job:main",
    #Set the job_name
    job_name=job_name,
    #Set the hyperparameters
    hyperparameters=hyperparams,
    #Define the file that contains the input data
    input_data="data.npy", # or input_data=s3_path
    # wait_until_complete=False,
)
```

> **Note**
>
> 入力データの詳細については、「入力」セクションを参照してください。

ハイパーパラメータは、次のコードを使用してハイブリッドジョブスクリプトにロードされます。

```
import json
import os

#Load the Hybrid Job hyperparameters
hp_file = os.environ["AMZN_BRAKET_HP_FILE"]
with open(hp_file, "r") as f:
    hyperparams = json.load(f)
```

> **Note**
>
> 入力データやデバイス ARN など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、この GitHub ページを参照してください。

Amazon Braket Hybrid Jobs チュートリアルのAmazon Braket Hybrid Jobs と PennyLane および Quantum 機械学習を使用した QAOA」では、ハイパーパラメータの使用方法を学ぶために非常に役立つガイド Amazon Braketがいくつか提供されています。

OpenQASM 3.0 で回路を実行する

Amazon Braket は、ゲートベースの量子デバイスとシミュレーターで 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、SDK を使用して Braket 回路を送信するか、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。 Amazon Braket

このガイドのトピックでは、以下の量子タスクを完了する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 の問題のトラブルシューティング

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の機能の概要と、その他のリソースへのリンクについても説明します。

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
- Qubit OpenQASM 3.0 を使用した再ワイヤリング
- OpenQASM 3.0 を使用した逐語的なコンパイル
- Braket コンソール
- 追加リソース
- 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、ユーザーは量子ゲートと量子計算の構成要素を形成する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、単純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 の新しいバージョン (3.0) では、以前のバージョンを拡張して、パルスレベルの制御、ゲートタイミング、クラシック制御フローなどの機能を追加し、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ます。現在のバージョン 3.0 の詳細と仕様は、GitHub OpenQASM 3.x Live Specification で入手できます。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コントロールを通じて量子プログラムを指定する表現的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 の OpenQASM のサポートにより、ゲートベースの量子アルゴリズムを開発するための一貫したアプローチとしての採用が促進され、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がなくなります。

OpenQASM 3.0 に既存のプログラムのライブラリがある場合は、こ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。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 の 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、ハードウェアデバイスや Braket を使用したオンデマンドシミュレーターで現在できることは、すべて Braket を使用して OpenQASM で実行できますAPI。OpenQASM 3.0 プログラムを実行するには、すべてのゲートベースのデバイスに OpenQASM 文字列を直接指定します。これは、回路が Braket 上のデバイスに現在提供されている方法と似ています。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 と 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、AmazonBraket 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？

次のセクションでは、Braket でサポートされている OpenQASM 3.0 データ型、ステートメント、およびプラグマの手順を示します。

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
- Local Simulator 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、次の OpenQASM データ型がサポートされています。

- 負以外の整数は、 (仮想および物理) 量子ビットインデックスに使用されます。
  - `cnot q[0], q[1];`
  - `h $0;`
- 浮動小数点数または定数は、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 に組み込まれた定数であり、パラメータ名として使用することはできません。

- 複雑な数値の配列 (架空の部分の OpenQASM `im`表記を使用) は、一般的なヘルミティアンオブザーバブルを定義するための結果タイプのプラグマと単一プラグマで許可されます。
  - `#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、次の OpenQASM ステートメントがサポートされています。

- `Header: OPENQASM 3;`
- クラシックビット宣言：
  - `bit b1;` (同等、`creg b1;`)
  - `bit[10] b2;` (同等、`creg b2[10];`)
- 量子ビット宣言：
  - `qubit b1;` (同等、`qreg b1;`)
  - `qubit[10] b2;` (同等、`qreg b2[10];`)
- 配列内のインデックス作成: `q[0]`
- 入力: `input float alpha;`
- 物理 の仕様qubits: `$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**
>
> デバイスの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には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ゲートqubitsが必要です。

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

- qubit またはレジスタqubits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;`

Braket OpenQASM プラグマ

次の OpenQASM プラグマ手順は、Amazon Braket でサポートされています。

- 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`
- 結果タイプのプラグマ
  - 基本不変の結果タイプ :
    - 状態ベクトル: `#pragma braket result state_vector`
    - 密度マトリックス: `#pragma braket result density_matrix`
  - グラデーション計算のプラグマ :
    - 結合勾配: `#pragma braket result adjoint_gradient expectation(2.2 * x[0] @ x[1]) all`

Zベースの結果タイプ

  - 振幅: `#pragma braket result amplitude "01"`
  - 確率: `#pragma braket result probability q[0], q[1]`
- ローテーションされた基本結果タイプ
  - 期待値: `#pragma braket result expectation x(q[0]) @ y([q1])`
  - 分散: `#pragma braket result variance hermitian([[0, -1im], [1im, 0]]) $0`
  - サンプル: `#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、`qregvs creg`や `qubitvs` など、構文の小さな違いがありますbit。測定構文にも違いがあり、これらは正しい構文でサポートする必要があります。

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

は、Braket の QPU またはオンデマンドシミュレーターの一部として提供されていない高度な OpenQASM 機能`LocalSimulator`をサポートしています。以下の機能のリストは、 でのみサポートされていますLocalSimulator。

- ゲート修飾子
- OpenQASM 組み込みゲート
- クラシック変数
- クラシックオペレーション
- カスタムゲート
- クラシックコントロール
- QASM ファイル
- サブルーチン

各高度な機能の例については、このサンプルノートブックを参照してください。OpenQASM の完全な仕様については、OpenQASM 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 {
    ...
}
```

デフォルトブロック：

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

波形：

```
// 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、次の例に示すように、GHZ 状態を準備するシンプルな OpenQASM 3.0 プログラム (ghz.qasm) から開始できます。

```
// 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

Amazon Braket Python SDK を使用して、次のコードを使用してこのプログラムを Amazon Braket デバイスに送信できます。Amazon S3 バケットの場所の例「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)

# You can also specify an optional s3 bucket location and number of shots,
 # if you so choose, when running the program
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

次の例に示すように、AWS Python SDK for Braket (Boto3) を使用して OpenQASM 3.0 文字列を使用して量子タスクを作成することもできます。次のコードスニペットは、上記のように GHZ 状態を準備する 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

次の例に示すように、 AWS Command Line Interface (CLI) を使用して OpenQASM 3.0 プログラムを送信することもできます。

```
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`レスポンスによる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この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
    }
  }
}
```

上記のフィールドでは、以下について詳しく説明します。

ポート：

特定のポートの関連プロパティに加えて、QPU で宣言された事前に作成された外部 (extern) デバイスポートについて説明します。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 (方向)
  - ポートの方向。ドライブポートはパルスを送信し (「tx」方向)、測定ポートはパルスを受信します (「rx」方向)。
- 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 - fast-flux など)。
- Dt (dt)
  - 指定されたポートの 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)
  - 指定されたポートに関連付けられた量子ビット。
- 中心頻度 (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フレームに関連する中心頻度のリスト。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：

QPU で宣言された事前に作成された外部フレームと、フレームに関連するプロパティについて説明します。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。
- 頻度 (頻度)
  - フレームのデフォルトの初期頻度。
- センターの頻度 (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。その結果、頻度調整は中心周波数の所定のデルタ内にとどまる必要があります。帯域幅値は検証パラメータで確認できます。

- フェーズ (フェーズ)
  - フレームのデフォルトの初期フェーズ。
- 関連付けられたゲート (associatedGate)
  - 指定されたフレームに関連付けられたゲート。
- 量子ビットマッピング (qubitMappings)
  - 指定されたフレームに関連付けられた量子ビット。
- 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します。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:

指定されたOpenPulse関数に関連する引数、引数タイプ、戻り値タイプに加えて、デバイスに対してサポートされている関数について説明します。OpenPulse 関数の使用例については、OpenPulse 仕様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
  - フレームのフェーズを指定された値だけシフトします。
- set_phase
  - フレームのフェーズを指定された値に設定します。
- swap_phases
  - 2 つのフレーム間でフェーズをスワップします。
- shift_frequency
  - フレームの頻度を指定された値だけシフトします。
- set_frequency
  - フレームの頻度を指定された値に設定します。
- play
  - 波形をスケジュールする
- capture_v0

- キャプチャフレームの値をビットレジスタに返す

SupportedQhpTemplateWaveforms:

デバイスで使用できる構築済みの波形関数、および関連する引数とタイプについて説明します。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。

定数

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

τ は波形の長さで、 iqは複雑な数値です。

```
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 に設定するとTrue、ガウシアンはオフセットされ、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リングされ、A最大値に達します。

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

ガウシアンのドラッグ

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

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 に設定するとTrue、Adiabatic Gate (DRAG) Gaussian による派生的削除はオフセットされ、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リングされ、実際の部分は最大に達しAます。DRAG 波形の詳細については、ホワイトペーパー「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)
```

Erf Square

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

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

ここで、 L は長さ、 Wは波形の幅、 はエッジの上下速度σを定義します。 t_1=(L-W)/2 $t_2$2=(L+W)/2Aは振幅です。ZaE を に設定するとTrue、ガウシアンはオフセットされ、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リングされ、A最大値に達します。次の式は、波形の再スケーリング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)と b=erf(-t_1/σ)/2+erf(t $_2$/σ)/2 。

```
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 の場合false、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について説明します。たとえば、defcal最初に使用済み量子ビットのゲートスルーを定義しないと、プログラム内のHゲートのように非ネイティブゲートを使用することはできません。デバイス機能の下にネイティブゲートnativeGateSetキーのリストがあります。

ValidationParameters:

以下を含むパルス要素の検証境界について説明します。

- 波形の最大スケール/最大振幅値 (任意および構築済み)
- Hz 単位の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/秒単位の持続時間
- 最大パルス長/秒単位の持続時間

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、デバイス機能出力の `action`フィールドの `braket.ir.openqasm.program`キーを参照してください。例えば、Braket State Vector Simulator 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SV1。

```
...
  "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演算子を追加します。たとえば、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 プログラムを送信できます。

```
// 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演算子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 演算子

Kraus 演算子を生成するには、行列のリストを反復処理し、行列の各要素を複雑な式として出力します。

Kraus 演算子を使用する場合は、次の点に注意してください。

- の数は 2 を超えるqubitsことはできません。スキーマの現在の定義により、この制限が設定されます。
- 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。
- 合計長は 2^{2*num_qubits} 行列を超えることはありません。つまり、1 には 4 つのマトリックス qubit、2 には 16 のマトリックスですqubits。
- 提供されたすべてのマトリックスは、完全正のトレース保存 (CPTP) です。
- 変換結合を持つ Kraus 演算子の積は、アイデンティティマトリックスに合計する必要があります。

Qubit OpenQASM 3.0 を使用した再ワイヤリング

Amazon Braket は、Rigettiデバイスの 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、このページを参照してください)。ナイーブ再ワイヤリング戦略qubitsで物理 を使用する場合は、 qubitsが選択したデバイスに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、PARTIAL 再ワイヤリング戦略はRigettiデバイスで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、、 などのベンダーが提供する量子コンピュータで量子回路を実行する場合IonQ、変更を加えることなく、定義されたとおりに回路を実行するようにコンパイラに指示できます。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など、保持されるものを正確に指定できます。回路の特定の部分のみを保持するには、保存された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、 は回路全体の逐語的なコンパイルIonQ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 を使用すると、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。このプラグマはそのままにしておき、ハードウェアの低レベルコンパイルルーチンでは最適化されません。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。

Braket コンソール

OpenQASM 3.0 タスクは Amazon Braket コンソールで管理できます。コンソールでは、OpenQASM 3.0 で量子タスクを送信した経験は、既存の量子タスクを送信した経験と同じです。

追加リソース

OpenQASM は、すべての Amazon Braket リージョンで使用できます。

Amazon Braket で 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 (完全) 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、結合区別方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、次の例のコードに示すように、適切なプラグマを指定できます。

```
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
```

個々のパラメータを明示的に一覧表示する代わりに、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。これは、パラメータの数が非常に多い場合に便利なオプションです。この場合、プラグマは次の例のコードのようになります。

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

Amazon Braket の OpenQASM 3.0 実装では、個々の演算子、テンソル製品、Hermitian オブザーバビリティ、Sumオブザーバビリティなど、すべてのオブザーバビリティ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演算子は expectation()関数内でラップする必要があり、観測可能な各用語が作用する量子ビットは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プログラムの送信をサポートします。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。たとえば、次のコードスニペットでは、2 量子ビット回路を作成し、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];
"""
```

この例では、2 つの量子ビット q[0]と を持つ量子回路がありますがq[1]、最初の量子ビットの状態の測定にのみ関心があります。これは、qubit[0] の状態を測定しb[0] = measure q[0]、その結果をクラシックビット b[0] に保存する行 によって実現されます。この部分的な測定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できます。

```
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)
```

デバイスが部分的な測定をサポートしているかどうかは、アクションプロパティの requiresAllQubitsMeasurementフィールドを調べることで確認できます。 の場合False、部分的な測定がサポートされます。

```
from braket.devices import Devices

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

ここで、 requiresAllQubitsMeasurementは です。これはFalse、すべての量子ビットを測定する必要がないことを示します。

実験機能を調べる

研究ワークロードを進めるには、新しい革新的な機能にアクセスすることが重要です。Braket Direct を使用すると、可用性が制限された新しい量子デバイスなど、利用可能な実験的な機能へのアクセスを Braket コンソールで直接リクエストできます。

実験機能へのアクセスをリクエストするには：

1. Amazon Braket コンソールに移動し、左側のメニューで Braket Direct を選択し、実験機能セクションに移動します。
2. 「アクセスの取得」を選択し、必要な情報を入力します。
3. ワークロードの詳細と、この機能を使用する予定の場所を指定します。

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 の背の高い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

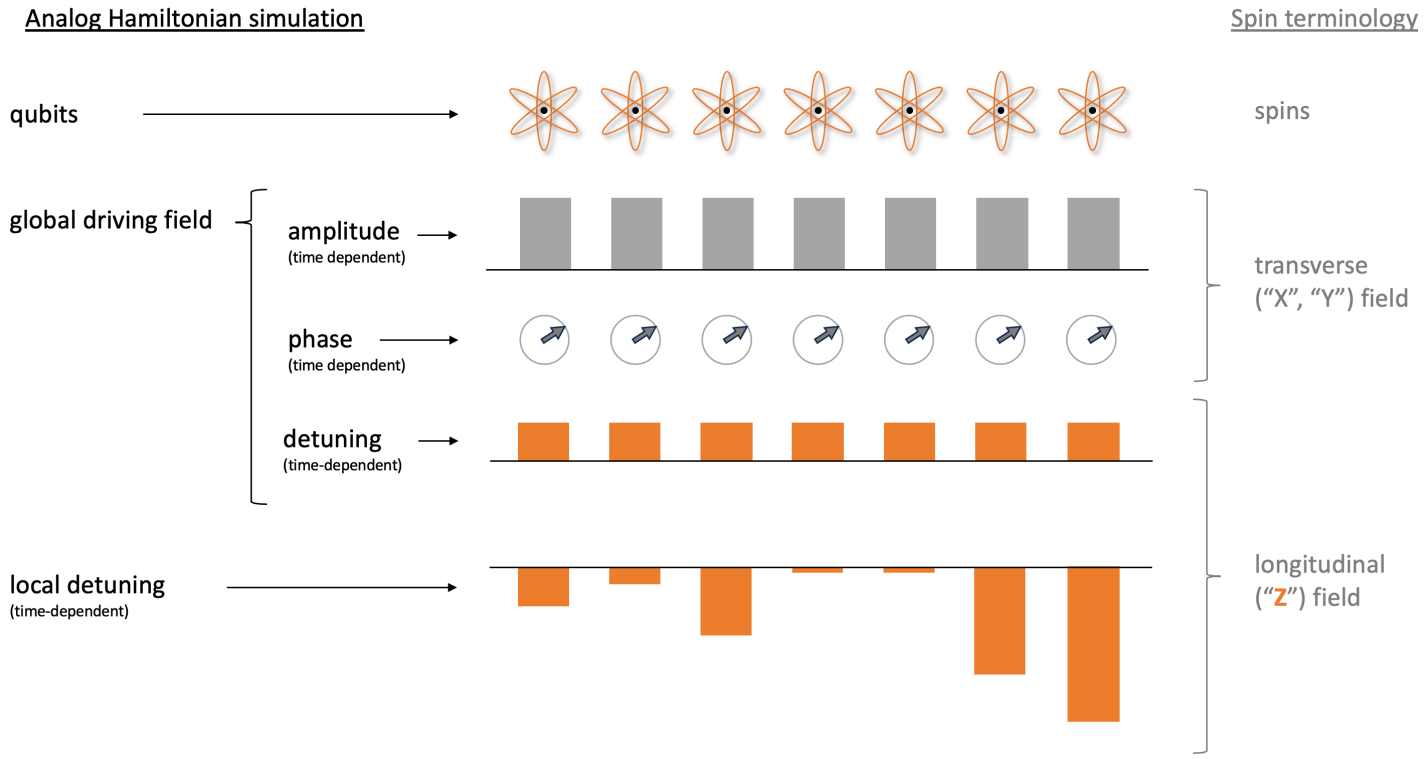
QuEra Aquila でのローカルデチューニングへのアクセス

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影響を与え、均一な駆動フィールドと Rydberg-Rydberg インタラクションが作成できる量子ビットを超えて、異なる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： ローカル調整フィールドの空間パターンは AHS プログラムごとにカスタマイズできますが、プログラムの過程で一定です。ローカル調整フィールドの時系列は、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、ローカル調整フィールド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、特定のデバイスプロパティセクション - の Braket SDK を通じて表示できますaquila_device.properties.paradigm.rydberg.rydbergLocal。

制限： ローカル調整フィールドを使用する量子プログラムを実行すると (その大きさがハミルトニアンで一定ゼロに設定されている場合でも) 、デバイスは Aquila プロパティのパフォーマンスセク

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。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調整フィールドを省略することをお勧めします。



例:

1. スピンシステムにおける不均一な長手磁界の影響をシミュレートします。

   駆動フィールドの振幅とフェーズは、回転時の横磁界と同じ影響を量子ビットにもたらしますが、駆動フィールドのチューニングとローカルチューニングの合計は、回転時の縦磁界と同じ影響を量子ビットにもたらします。ローカル調整フィールドを空間的に制御することで、より複雑なスピンシステムをシミュレートできます。

2. 非平衡初期状態の準備。

   サンプルノートブック Rydberg 原子を使用した格子ゲージ理論のシミュレーションは、システムを Z2 の順序付けられたフェーズにアニーリングするときに、9 原子の線形配置の中心原子が興奮しないようにする方法を示しています。準備ステップの後、ローカル調整フィールドはランプダウンされ、AHS プログラムはこの特定の非均衡状態から始まるシステムの時間進化をシミュレートし続けます。

3. 加重最適化の問題を解決します。

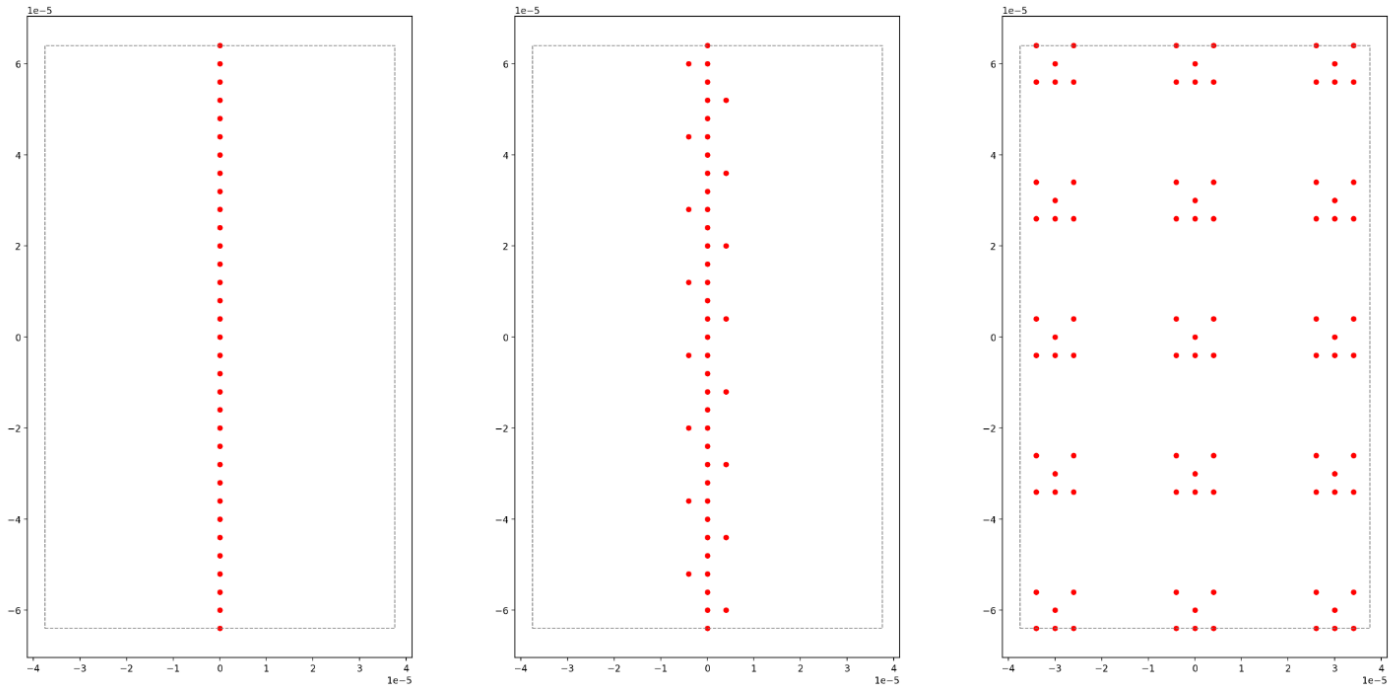
サンプルノートブック Maximum weight independent set (MWIS) は、Aquila で MWIS 問題を解決する方法を示しています。ローカル調整フィールドは、エッジが Rybderg-blockage 効果によって実現されるユニットディスクグラフのノードの重みを定義するために使用されます。均一な地上状態から開始し、ローカル調整フィールドを徐々に増やすと、システムは MWIS Hamiltonian の地上状態に移行し、問題の解決策を見つけます。

QuEra Aquila の背の高いジオメトリへのアクセス

高ジオメトリ機能を使用すると、高くなったジオメトリを指定できます。この機能を使用すると、AHS プログラムのアトム配置は、Aquila の通常の機能を超える y 方向の追加の長さにまたがることができます。

制約： 背の高いジオメトリの最大高さは 0.000128 m (128 um) です。

制限： アカウントでこの実験的な機能が有効になっている場合、デバイスのプロパティページとGetDevice呼び出しに表示される機能は、高さの通常の下限を引き続き反映します。AHS プログラムが通常の機能を超えるアトム配置を使用すると、フィルエラーが増加することが予想されます。タスク結果pre_sequenceの一部で予期しない 0 の数が増え、完全に初期化された配置を得る機会が減ります。この効果は、多くのアトムを持つ行で最も強力です。



例:

1. より大きな 1d および準 1d 配置。

   アトムチェーンとはしごのような配置は、より高いアトム数に拡張できます。y と平行に長い方向を向けることで、これらのモデルの長いインスタンスをプログラミングできます。

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が増えました。

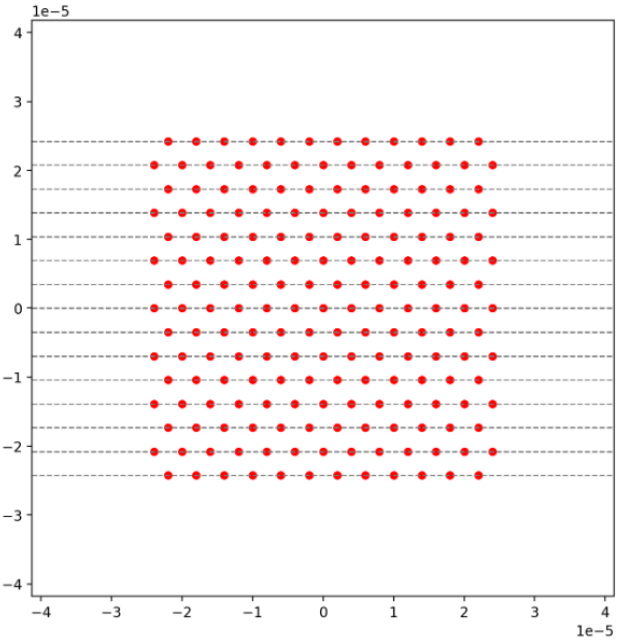
   Aquila のノートブック並列量子タスクの例は、問題のジオメトリの多重コピーを 1 つのアトム配置に配置することで、利用可能な領域を最大限に活用する方法を示しています。利用可能な領域が多いほど、より多くのコピーを配置できます。

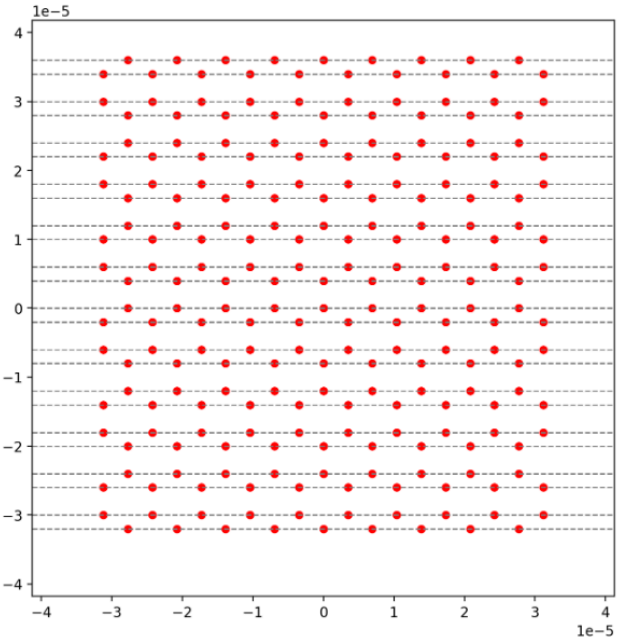
QuEra Aquila でのタイトなジオメトリへのアクセス

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、アトム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 つのアトムサイトの y 座標は、ゼロ (同じ行) であるか、最小行間隔 (異なる行) よりも大きく異なる必要があります。タイトジオメトリ機能により、行間隔が最小限に抑えられ、よりタイトなアトム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、離れた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形格子です。

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

制限： アカウントでこの実験的な機能を有効にすると、デバイスのプロパティページとGetDevice呼び出しに表示される機能は、高さの通常の下限を引き続き反映します。AHS プログラムが通常の機能を超えるアトム配置を使用すると、フィルエラーが増加することが予想されます。お客様は、タスク結果pre_sequenceの一部で予期しない 0 の数が増え、完全に初期化された配置を得る機会が減ります。この効果は、多くのアトムを持つ行で最も強力です。





例:

1. 小さな格子定数を持つ非長方形格子。

   行間隔を小さくすると、一部の原子に最も近い隣接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、三角形、六角形、および Kagome 格子といくつかの準暗号があります。
2. 調整可能な格子ファミリー。

   AHS プログラムでは、インタラクションは原子のペア間の距離を調整することで調整されます。行間隔を小さくすると、アトム構造を定義する角度と距離は最小行間隔の制約によって制限されなくなるため、互いに対して異なるアトムペアの相互作用をより自由に調整できます。注目すべき例は、異なるボンド長を持つ Shastry-Sutherland 格子のファミリーです。

Amazon Braket でのパルス制御

パルスは、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て、パルスを使用して回路を送信できます。Braket SDK、OpenQASM 3.0、または Braket APIs 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な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
- フレームとポートのロール
- Hello Pulse の使用
- 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フェーズの両方として機能するソフトウェア抽象化です。クロック時間は、各使用量と、周波数で定義されるステートフルキャリア信号に対して増分されます。キュービットに信号を送信する場合、フレームはキュービットのキャリア周波数、フェーズオフセット、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、フレームの構築はデバイス、頻度、フェーズ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、ユーザーが操作して量子ビットを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特徴付けられます。この文字列は、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公開します。

```
from braket.pulse import Port
Port0 = Port("channel_0", dt=1e-9)
```

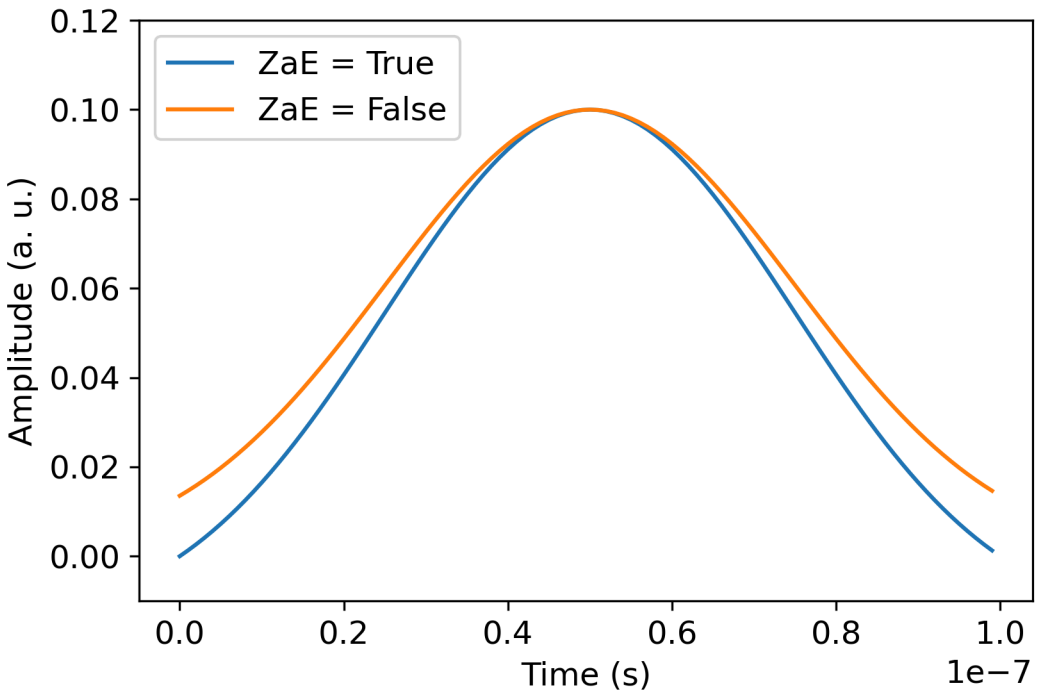
波形

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、複雑な数値のリストを使用するか、波形テンプレートを使用してハードウェアプロバイダーからリストを生成します。

```
from braket.pulse import ArbitraryWaveform, ConstantWaveform
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、離反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。

```
gaussian_waveform = GaussianWaveform(1e-7, 25e-9, 0.1)
x = np.arange(0, gaussian_waveform.length, drive_frame.port.dt)
plt.plot(x, gaussian_waveform.sample(drive_frame.port.dt))
```



上の画像は、 から作成されたガウス波形を示していますGaussianWaveform。パルス長は 100 ns、幅は 25 ns、振幅は 0.1 (任意の単位) を選択しました。波形はパルスウィンドウを中心としています。 はブール引数 `zero_at_edges` (凡例の ZaE) GaussianWaveformを受け入れます。に設

定するとTrue、この引数は t=0 と t=length のポイントがゼロになるようにガウシアン波形をオフセットし、最大値がamplitude引数に対応するように振幅を再スケーリングします。

パルスレベルのアクセスの基本概念を説明したので、次にゲートとパルスを使用して回路を構築する方法を説明します。

フレームとポートのロール

このセクションでは、各デバイスで使用可能な事前定義されたフレームとポートについて説明します。また、特定のフレームでパルスが再生されるときのメカニズムについても簡単に説明します。

Rigetti フレーム

Rigetti デバイスは、関連する量子ビットで擬似するように周波数とフェーズがキャリブレーションされた事前定義されたフレームをサポートします。命名規則は、 q{i}[_q{j}]_{role}_frame が{i}最初の量子ビット数{j}を参照し、フレームが 2 量子ビットインタラクションをアクティブ化する場合の 2 番目の量子ビット数を参照し、 がフレームのロール{role}を参照する です。ロールは以下のとおりです。

- rf は、量子ビットの 0-1 遷移を駆動するフレームです。パルスは、 setおよび shift関数を通じて以前に提供された頻度とフェーズのマイクロ波ー時信号として送信されます。信号の時間依存振幅は、フレームで再生される波形によって決まります。フレームは、単一量子ビットの対角線外インタラクションを差し込みます。詳細については、「Krantz et al.」および「Rahamim et al.」を参照してください。
- rf_f12 は に似rfており、そのパラメータは 1~2 遷移をターゲットとしています。
- ro_rx は、結合された同一平面導波路を介して量子ビットの分散読み取りを実現するために使用されます。読み取り波形の頻度、フェーズ、およびパラメータの完全なセットは事前にキャリブレーションされています。現在、フレーム識別子以外の引数capture_v0を必要としない を通じて使用されます。
- ro_tx は、レゾネーターからシグナルを送信するためのものです。現在使用されていません。
- cz は、2 量子ビットczゲートを有効にするようにキャリブレーションされたフレームです。ff ポートに関連付けられたすべてのフレームと同様に、隣接する と の擬似でペアの調整可能な量子ビットを調整することで、フラックスラインを介した交雑インタラクションを有効にします。エンタングルメカニズムの詳細については、「Reagor et al.」、「Caldwell et al.」、および「Didier et al.」を参照してください。
- cphase は、2 量子ビットcphaseshiftゲートを有効にするようにキャリブレーションされたフレームであり、ffポートにリンクされています。エンタングルメカニズムの詳細については、czフレームの説明を参照してください。

- xy は、2 量子ビット XY(θ) ゲートを有効にするようにキャリブレーションされたフレームであり、ffポートにリンクされています。エンタングルメカニズムと XY ゲートの達成方法の詳細については、cz「フレームの説明」および「Abrams et al.」を参照してください。

ff ポートに基づくフレームが調整可能な量子ビットの頻度をシフトすると、量子ビットに関連する他のすべての駆動フレームは、周波数シフトの振幅と期間に関連する量だけ減速されます。したがって、対応するフェーズシフトを隣接する量子ビットのフレームに追加して、この効果を補正する必要があります。

ポート

Rigetti デバイスは、デバイス機能を通じて検査できるポートのリストを提供します。ポート名は、q{i}_{type}が量子ビット番号{i}を参照し、 がポートのタイプ{type}を参照する規則に従います。すべての量子ビットにポートの完全なセットがあるわけではないことに注意してください。ポートのタイプは次のとおりです。

- rf は、単一量子ビット遷移を駆動するメインインターフェイスを表します。これは、 フレームrfと rf_f12フレームに関連付けられています。容量的に 量子ビットに結合されるため、ギガヘルツ範囲でのマイクロ波駆動が可能になります。
- ro_tx は、量子ビットに容量的に結合されたリードアウトレゾネーターに信号を送信する役割を果たします。読み取り信号配信は 8 倍に 8 倍になります。
- ro_rx は、量子ビットに結合されたリードアウトレゾネーターからシグナルを受信する役割を果たします。
- ff は、量子ビットに帰納的に結合された高速フラックス線を表します。これを使用して、トランスモンの頻度を調整できます。高度に調整可能なように設計された量子ビットのみがffポートを持ちます。このポートは、隣接するトランスモンの各ペア間に静的容量結合があるため、量子ビットと量子ビットの相互作用をアクティブ化する役割を果たします。

アーキテクチャの詳細については、「Valery et al.」を参照してください。

Hello Pulse の使用

このセクションでは、Rigettiデバイスのパルスを使用して 1 つの量子ビットゲートを直接特徴付けて構築する方法について説明します。量子ビットに磁気フィールドを適用すると、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。パルスの長さとフェーズがキャリブレーションされている場合、Rabi 振動は単一の量子ビット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 パルスを測定する最適なパルス長を決定します。

まず、パルスシーケンスを構築するには、 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 Amazon Resource Name (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。次のコードブロックは を使用しますRigetti Ankaa-3。

```
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の 2 つのコンポーネントが含まれています。通常、パルスシーケンスはフレームに適用できます。障壁や遅延などの一部の例外は、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。ドライブフレームは Rabi 振動のパルスを適用するために使用され、読み取りフレームは量子ビット状態を測定するために使用されます。この例では、 は量子ビット 25 のフレームを使用します。フレームの詳細については、「フレームとポートのロール」を参照してください。

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

次に、ドライブフレームで再生される波形を作成します。目標は、さまざまなパルス長の量子ビットの動作を特徴付けることです。毎回異なる長さの波形を再生します。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 をパルスシーケンスFreeParameterで使用します。波形とパルスシーケンスをフリーパラメータで一度作成し、同じパルスシーケンスを異なる入力値で実行できます。

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

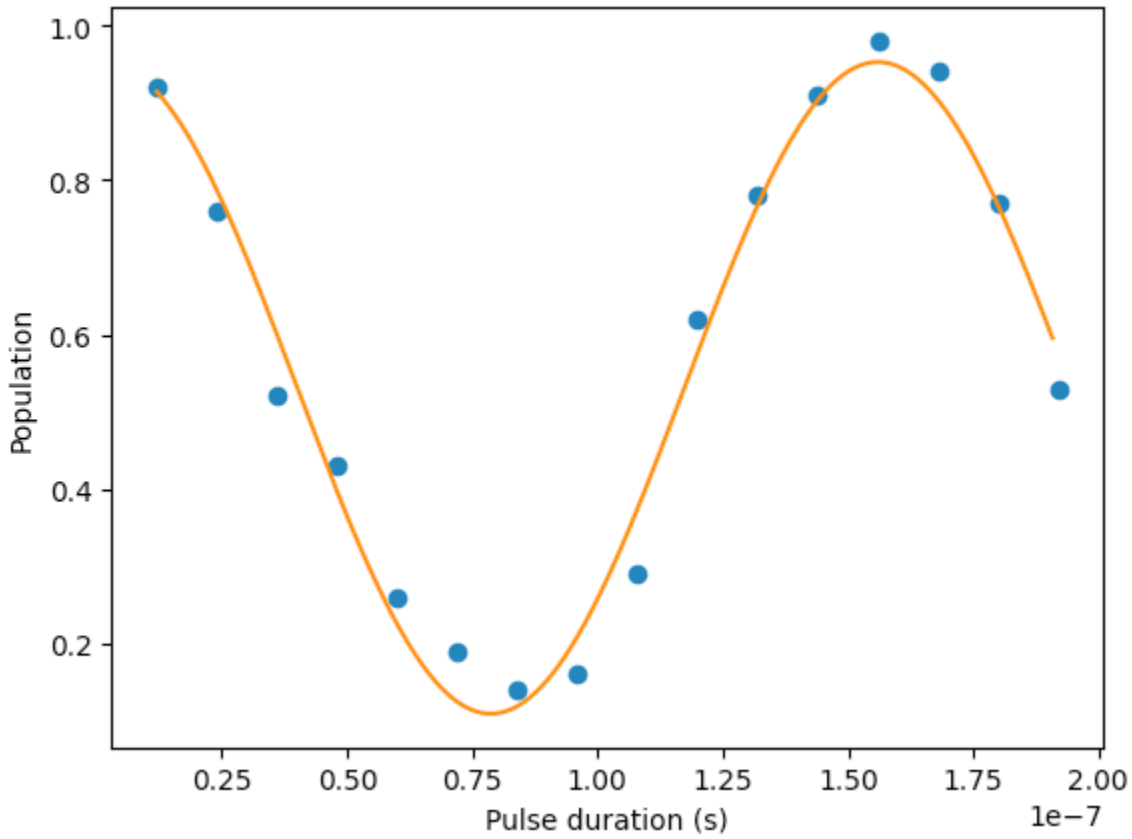
最後に、それらをパルスシーケンスとしてまとめます。パルスシーケンスでは、 はドライブフレームで指定された波形をplay再生し、 は読み取りフレームの状態capture_v0を測定します。

```
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。

```
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
]
```

量子ビット測定値の統計は、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、パルスの長さを微調整して、特定の 1 量子ビットゲートを実装できます。たとえば、以下の図のデータから、周期性は約 154 ns です。したがって、pi/2 ローテーションゲートは、長さが 38.5 ns のパルスシーケンスに対応します。



OpenPulse を使用した Hello OpenPulse

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。Amazon Braket はOpenPulse、OpenQASM 3.0 表現を使用してパルスを直接プログラミングするための をサポートしています。

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OpenPulseとして を使用します。 は、 `defcal` (「キャリブレーションの定義」の略) 宣言の形式で命令キャリブレーションの追加OpenPulseをサポートしています。これらの宣言では、低レベルの制御文法内でゲート命令の実装を指定できます。

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

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

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がパルスとして実装され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ますが、それらにアクセスすると、研究者はより良いゲートを設計したり、特定のゲートのパルスを伸張してゼロノイズ外挿などのエラー緩和のためのプロトコルを探索したりできます。

Amazon Braket は、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 を定期的にキャリブレーションし、多くの場合 1 日に 1 回以上キャリブレーションします。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。たとえば、0 に適用された RX(π/2) qubit のパルス実装を検査できます。

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

pulse_sequence_rx_pi_2_q0 = calibrations.pulse_sequences[rx_pi_2_q0]
```

`filter` 関数を使用して、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できます。ゲートのリストまたは のリストを渡しますQubitSet。次のコードは、RX (π/2) と 0 のすべてのキャリブレーションを含む 2 qubit つの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、次の回路を考えてみましょう。

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

オブジェクトのディクショナリをgate_definitionsキーワード引数PulseSequenceに渡すqubit 0ことで、ゲートのカスタムrxゲートキャリブレーションで実行できます。GateCalibrations オブジェクトpulse_sequencesの 属性からディクショナリを作成できます。指定されていないすべてのゲートは、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シミュレーション

アナログハミルト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。一連のゲートの代わりに、各回路は一度に数量子ビットでのみ動作します。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。システムのハミルトニアンは、そのエネルギーレベルと外部力の影響をエンコードし、一緒にその状態の時間進化を管理します。N 量子ビットシステムの場合、ハミルトニアンは複雑な数値の $2^N X 2^N$ 二乗行列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システムの時間進化に近づけるように設計されています。コヒーレント駆動フィールドの振幅の調整やパラメータの調整など。AHS パラダイムは、縮合物物理学や量子化学など、相互作用するパーティクルが多い量子システム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。の Aquila デバイスのような専用の量子処理ユニット (QPUs) はQuEra、AHS の能力を活用し、従来のデジタル量子コンピューティングアプローチの手の届かない問題に革新的な方法で対処するために開発されました。

このセクションの内容:

- Hello AHS: 最初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: 最初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

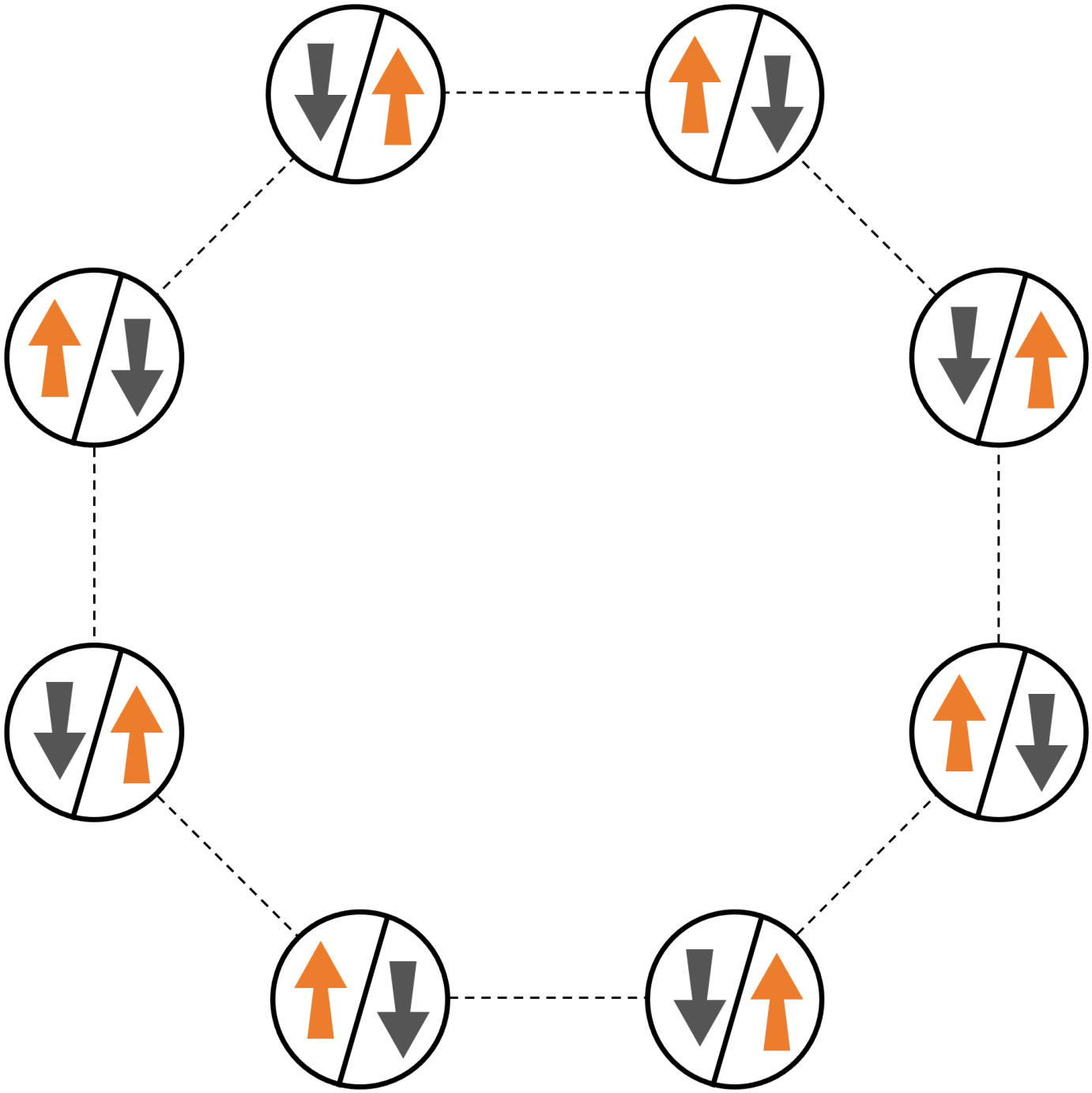
このセクションでは、最初のアナログハミルトニアンシミュレーションの実行について説明します。

このセクションの内容:

- インタラクションスピンチェーン
- 配置
- インタラクション
- 運転フィールド
- AHS プログラム
- ローカルシミュレーターでの実行
- シミュレーターの結果の分析
- QuEra の Aquila QPU での実行
- QPU 結果の分析
- 次のステップ

インタラクションスピンチェーン

多くの相互作用するパーティクルのシステムの正規例については、8 つのスピンのリング (それぞれが「up」|↑# および「down」|↓# 状態になる可能性があります) を考えてみましょう。小さくても、このモデルシステムは、自然に発生する磁気マテリアルのいくつかの興味深い現象をすでに示しています。この例では、連続するスピンが反対方向に向く、いわゆる強磁化防止の順序を準備する方法を示します。



配置

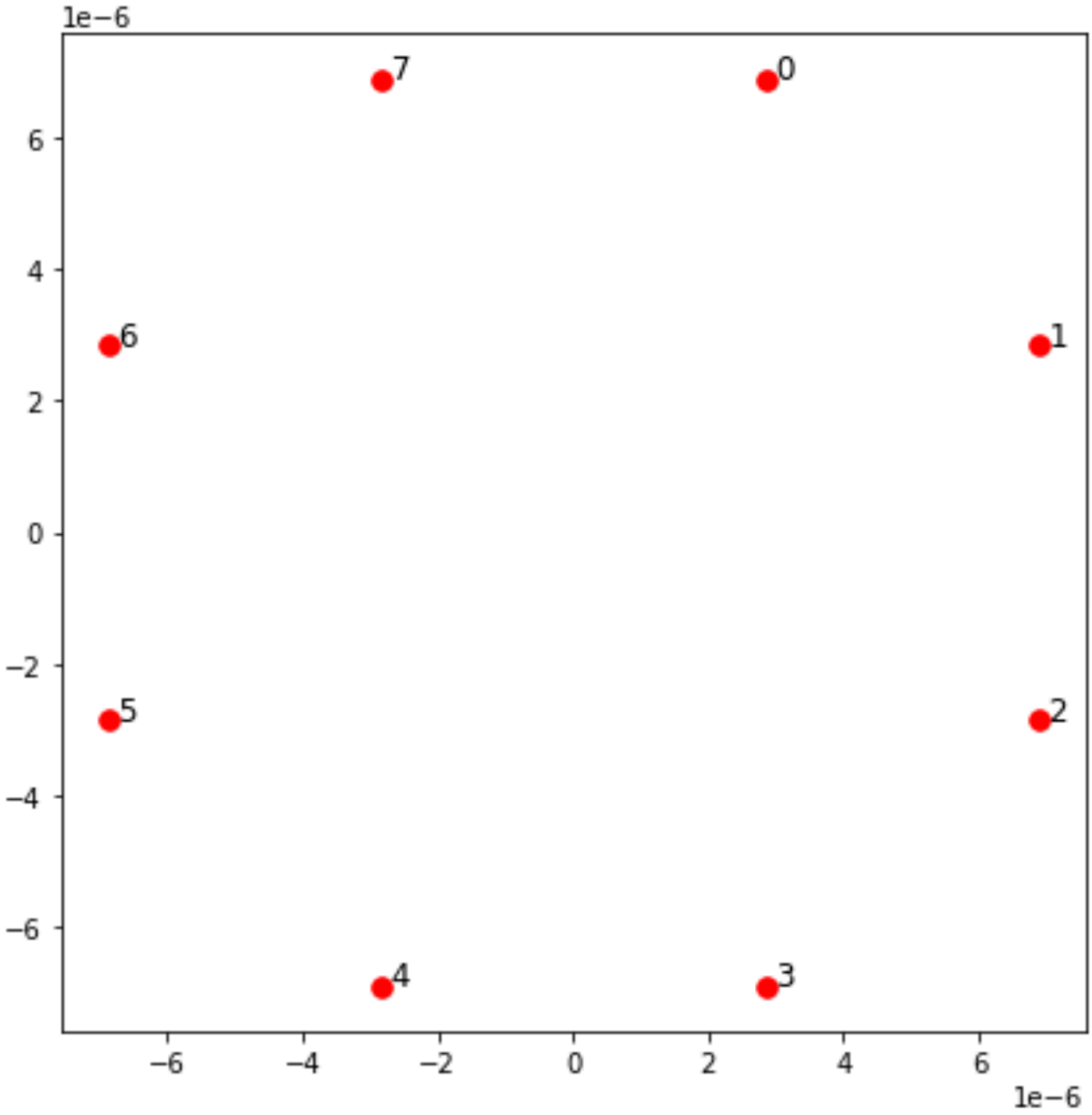
スピンごとに 1 つの中立原子を使用し、「上」スピン状態と「下」スピン状態は、それぞれ原子の興奮した Rydberg 状態と地面状態でエンコードされます。まず、2-d 配置を作成します。上記のスピンリングは、次のコードでプログラムできます。

前提条件: Braket SDK を pip 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している場合、この SDK にはノートブックがプリインストールされています) 。プロットを再現するには、シェルコマンド を使用して matplotlib を個別にインストールする必要がありますpip install matplotlib。

```
import numpy as np
   import matplotlib.pyplot as plt  # required for plotting

   from braket.ahs.atom_arrangement import AtomArrangement

   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)
```

また、 を使用してプロットすることもできます。

```
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
```



インタラクション

抗強磁気フェーズを準備するには、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 van der Waals インタラクションを使用します。これは、ニュートラルアトムデバイス (のAquilaデバイスなど) によってネイティブに実装されますQuEra。スピン表現を使用すると、このインタラクションのハミルトニア語用語は、すべてのスピンペア (j,k) の合計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 =$_{jj}$|↑##↑| はスピン j が「up」状態である場合にのみ 1 の値を受け取り、それ以外の場合は 0 の値を取る演算子です。強度は $V_{j,k}$=C_6/($d_{j,k}$) ^{6} で、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。この相互作用項の即時効果は、スピン j とスピン k の両方が「上」であるすべての状態がエネルギーを (V の量だけ) 増加させることです$_{j,k}$。AHS プログラムの残りの部分を慎重に設計することで、この相互作用により、隣接するスピンが両方とも「アップ」状態になるのを防ぐことができます。これは、一般的に「Rydberg ブロック」と呼ばれる効果です。

運転フィールド

AHS プログラムの開始時、すべてのスピン (デフォルトでは) は「ダウン」状態で始まり、いわゆる強磁気フェーズになります。抗強磁気フェーズを準備するという目標を念頭に置いて、スピンをこの状態から「アップ」状態が優先される多体状態にスムーズに移行する時間依存のコヒーレント駆動フィールドを指定します。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、"(t)、"(t)、"(t) は、すべてのスピンに均一に影響する駆動フィールドの時間依存、グローバル振幅 (Rabi 周波数) 、フェーズ、およびデチューニングです。ここで、$S_{-,k}$=$_{kk}$|↓##↑| および $S_{+,k}$=($S_{-,k}$) †=$_{kk}$|↑##↓| 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=$_{kk}$|↑↑��#↑| は以前と同じ演算子です。駆動フィールドの IAM 部分は、すべてのスピンの「ダウン」状態と「アップ」状態を同時に一貫して結合し、Δ 部分は「アップ」状態のエネルギー報酬を制御します。

強磁フェーズから非強磁フェーズへのスムーズな移行をプログラムするには、次のコードで駆動フィールドを指定します。

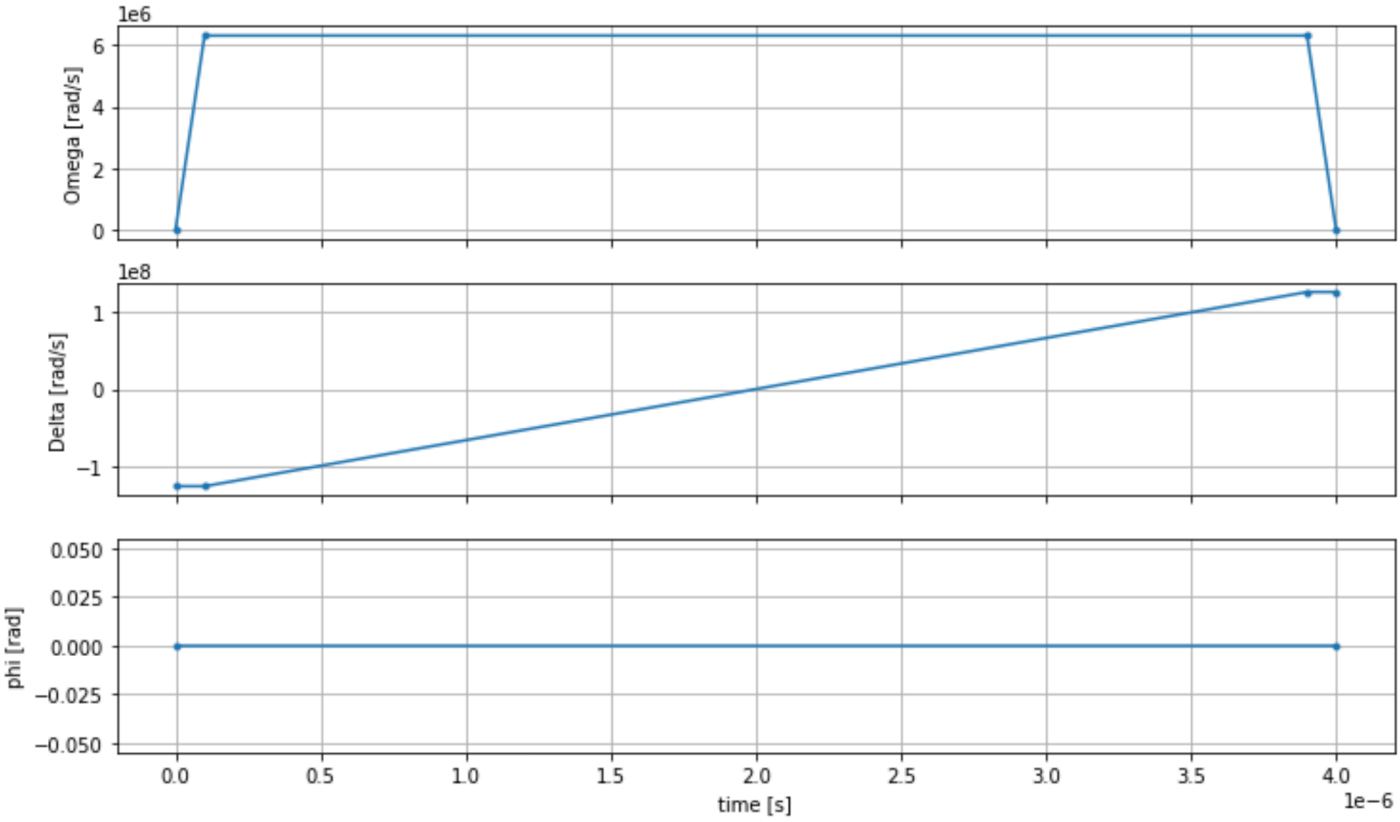
```
from braket.timings.time_series import TimeSeries
   from braket.ahs.driving_field import DrivingField

   # 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
    )
```

次のスクリプトを使用して、運転フィールドの時系列を視覚化できます。

```
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

登録、運転フィールド (および暗黙的な van der Waals インタラクション) は、アナログハミルトンシミュレーションプログラム を構成しますahs_program。

```
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

この例は小さい (15 スピン未満) ため、AHS 互換の 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。ローカルシミュレーターは Braket SDK で無料で利用できるため、コードが正しく実行されるようにするのがベストプラクティスです。

ここでは、ローカルシミュレーターが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ため、ショットの数を高い値 (つまり、100 万) に設定することができます。したがって、ショットの数を増やしながら、合計ランタイムをわずかに増やすだけです。

```
from braket.devices import LocalSimulator
   device = LocalSimulator("braket_ahs")

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

シミュレーターの結果の分析

各スピンの状態を推測する次の関数 (「ダウン」の場合は「d」、「アップ」の場合は「u」、または空のサイトの場合は「e」) 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
```

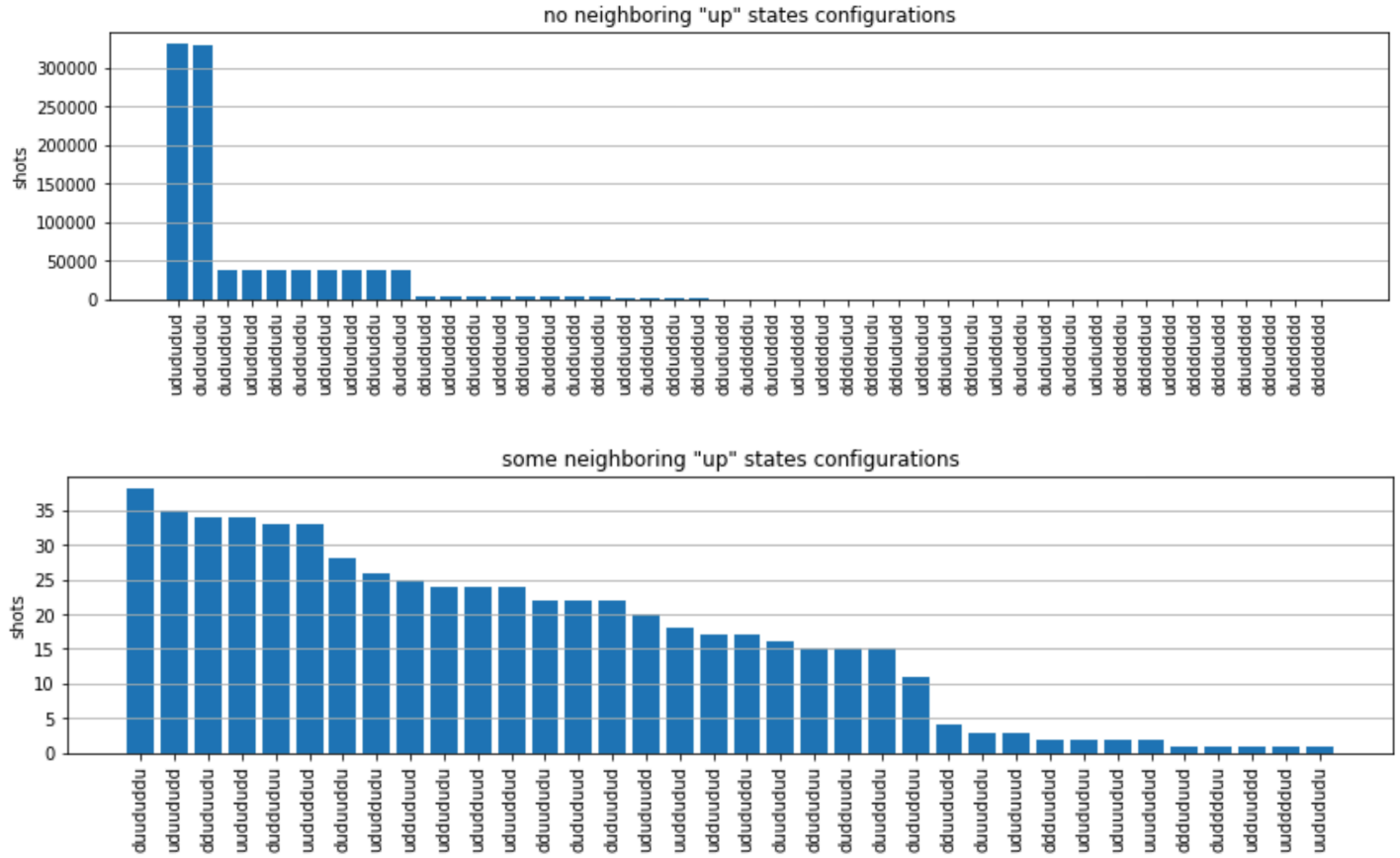
```
            state_idx = np.array(pre) * (1 + np.array(post))
            state = "".join(map(lambda s_idx: states[s_idx], state_idx))
            state_counts.update((state,))
        return dict(state_counts)

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

```
{'udududud': 330944, 'dudududu': 329576, 'dudud dud': 38033, ...}
```

以下はcounts、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、次のコードで視覚化することもできます。

```
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、次の観測値を読み、抗強磁気フェーズを正常に準備したことを確認できます。

1. 一般に、ブロックされていない状態 (2 つの隣接するスピンが「アップ」状態ではない状態) は、少なくとも 1 つの隣接するスピンのペアが両方とも「アップ」状態である状態よりも一般的です。
2. 一般的に、設定がブロックされない限り、興奮が「アップ」する状態が優先されます。
3. 最も一般的な状態は、実際には完全な強磁化防止状態 "dudududu"と です"udududud"。

4. 2 番目に一般的な状態は、連続する分離が 1、2、2 の「アップ」興奮が 3 つしかない状態です。これは、van der Waals のインタラクションが、近傍にも影響する (ただし、はるかに小さい) ことを示しています。

QuEra の Aquila QPU での実行

前提条件: Braket SDK のインストールとは別に、Amazon Braket を初めて使用する場合は、必要な開始手順が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している場合、Braket SDK にはインスタンスが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、AquilaQPU 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、AquilaQPU 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。(これらの要件は、名前に「解決」が付いたデバイスパラメータによって管理されます。 ノートブックaquila_qpu.properties.dict()で を実行すると、それらを確認できます。 Aquila の機能と要件の詳細については、「Aquila の概要ノートブック」を参照してください。) これを行うには、 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 (現在 100 ショットのみを実行) を Aquila QPU で実行できるようになりました。

> **Note**
>
> プロセッサでこのプログラムを実行すると、コストAquilaが発生します。Amazon Braket SDK には、お客様が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る Cost Tracker が含まれています。

```
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}")
```

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

量子タスクの実行にかかる時間は大きく異なるため (可用性ウィンドウと QPU 使用率によって異なります)、量子タスク ARN を書き留めておくことをお勧めします。このため、後で次のコードスニペットを使用してステータスを確認できます。

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

ステータスが完了すると (Amazon Braket コンソールの量子タスクページから確認することもできます)、次の方法で結果をクエリできます。

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

QPU 結果の分析

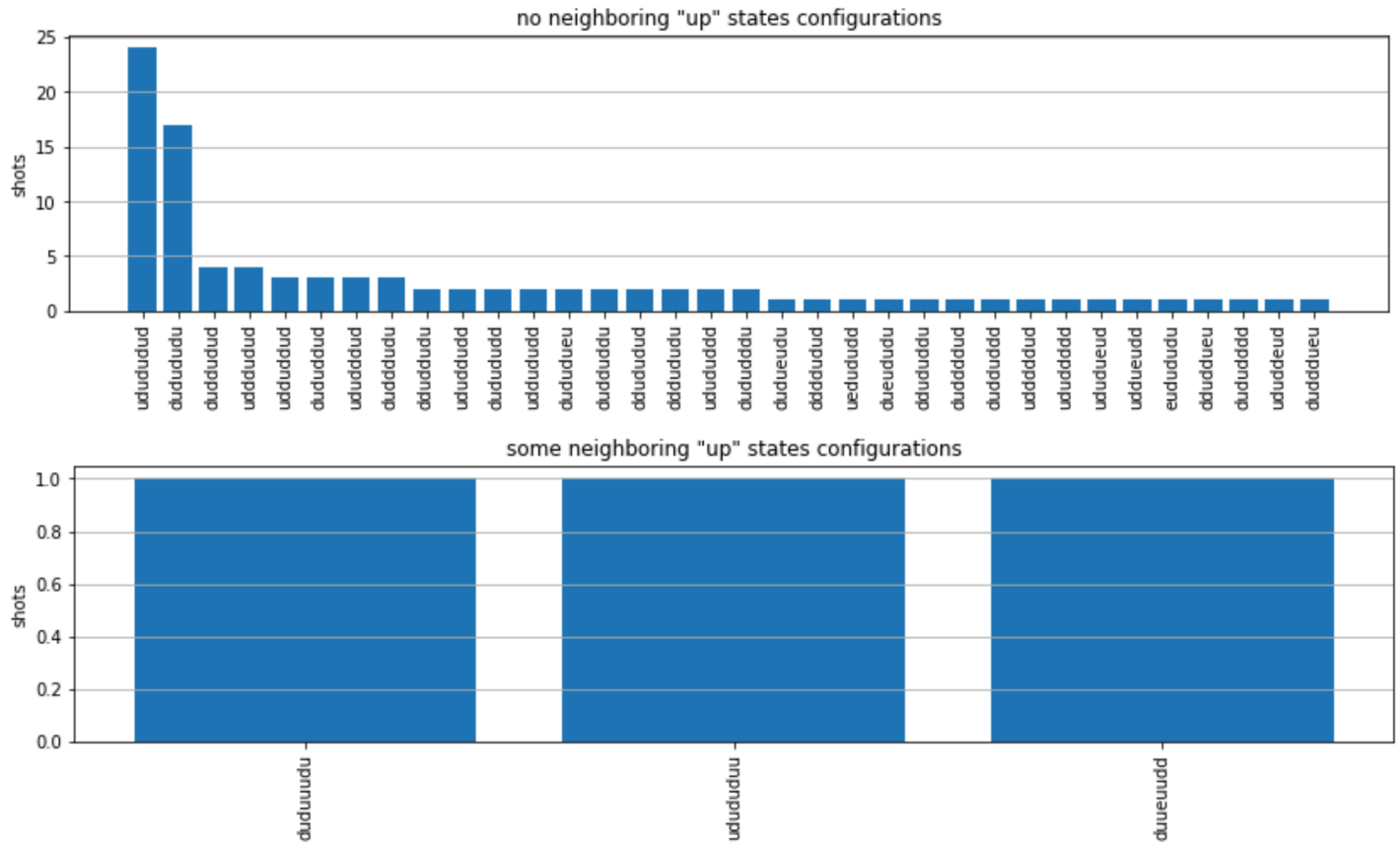
以前と同じget_counts関数を使用して、カウントを計算できます。

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

```
*[Output]*
   {'udududud': 24, 'dudududu': 17, 'dududdud': 3, ...}
```

と を使用してプロットしますplot_counts。

```
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、AquilaQPU の原子ごとの準備の不完全性が 1~2% であるためです。これとは別に、ショット数が少ないため、結果は予想される統計的変動内でシミュレーションと一致します。

次のステップ

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 で最初の AHS ワークロードを実行できました。

Rydberg 物理、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、サンプル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、 の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QuEra。ここで説明する詳細は次のとおりです。1) によってシミュレートされたパラメータ化されたハミルトニアンAquila、2) AHS プログラムパラメータ、3) AHS 結果コンテンツ、4) Aquila機能パラメータ。Ctrl +F テキスト検索を使用して、質問に関連するパラメータを検索することをお勧めします。

このセクションの内容:

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

ハミルトニア語

のAquilaマシンは、次の (時間依存) ハミルトニアンをネイティブにQuEra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的な機能であり、Braket Direct からのリクエストで利用できます。

この場合、次のようになります。

- $H_{drive,k}(t)=(\ ^{1}/_{2}$ Gbps(t)$^{ei\varphi(t)}S_{-,k}+\ ^{1}/_{2}$ Gbps(t)$e^{-i\varphi(t)}\ S_{+,k}) + (-_{global}\Delta(t)n_{k}\ )$ 、
  - 「(t)」は、時間依存的なグローバル走行振幅 (Rabi 周波数) を (rad / s) 単位で表したものです。
  - (t)��
  - $S_{-,k}$ と $S_{+,k}$は、原子 k のスピン下降および上昇演算子です (ベース =|↓#|g#, |↑#=|r#, S_{-}=|g##r|, $S_{+}=(S_{-}\ )^{\dagger}$=|r##g|)。
  - $_{global}\Delta(t)$ は時間依存的でグローバルな調整です
  - n_{k} は、原子 k の Rydberg 状態の射影演算子です (例: n=|r##r|)
- $H_{local\ detuning,k}(t)=-_{local}\Delta(t)h\ _{k}n_{k}$
  - $_{local}\Delta(t)$ は、ローカル周波数シフトの時間依存係数で、単位は (rad / s) です。
  - h_{k} はサイト依存係数で、0.0 から 1.0 の間のディメンションレス数です。
- $V_{vdw,k,l}=C_{6}/(d_{k,l}\ )^{6}n_{k}n_{l}$、
  - $_{6}$は、(rad / s) * (m)^6 の単位の van der Waals 係数です。
  - $d_{k,l}$ は、原子 k と l の間のユークリッド距離で、メートル単位で測定されます。

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

- 2 次元の原子配置 (各原子 k の $_{k}$xand $_{k}$ycoordinates、um 単位) 。k、l=1、2、...N で対の原子距離 $d_{k,l}$ を制御します。
- 時間依存、グローバル Rabi 頻度、単位 (rad / s) の "(t)
- 時間依存のグローバルフェーズ、(rad) 単位の φ(t)
- $_{global}\Delta(t)$、時間依存、グローバル調整、単位 (rad / s)
- $_{local}\Delta(t)$、ローカル調整の大きさの時間依存 (グローバル) 係数、 (rad / s) の単位
- h_{k}、ローカル調整の大きさの (静的) サイト依存係数、0.0 ~ 1.0 のディメンションレス数

> **Note**
>
> ユーザーは、関係するレベル (S_{-}、S_{+}、n 演算子が固定されている) や Rydberg-Rydberg 相互作用係数 (C) の強度を制御できません$_{6}$。

Braket AHS プログラムスキーマ

braket.ir.ahs.program_v1.Program オブジェクト (例)

> **Note**
>
> アカウントでローカル調整機能が有効になっていない場合は、次の例localDetuning=[]で 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 (例)

> **Note**
>
> アカウントでローカル調整機能が有効になっていない場合は、次の例"localDetuning": []の 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]
                }
            }
```

```
        ]
    }
}
```

メインフィールド

プログラムフィールド	type	description
setup.ahs_register.sites	List[List[10 進数]]	ピンセットが原子をトラップする 2 次元座標のリスト
setup.ahs_register.filling	List[int]	トラップサイトを占有する原子を 1、空のサイトを 0 でマークします
hamiltonian.drivingFields[].amplitude.time_series.times	List[10 進数〕	走行振幅の時点、Omega(t)
hamiltonian.drivingFields[].amplitude.time_series.values	List[10 進数〕	走行振幅の値、Omega(t)
hamiltonian.drivingFields[].amplitude.pattern	str	走行振幅の空間パターン、Omega(t)。「均一」である必要があります
hamiltonian.drivingFields[].phase.time_series.times	List[10 進数〕	走行フェーズの時点、phi(t)
hamiltonian.drivingFields[].phase.time_series.values	List[10 進数〕	走行フェーズの値、phi(t)
hamiltonian.drivingFields[].phase.pattern	str	走行フェーズの空間パターン、phi(t)。「均一」である必要があります

プログラムフィールド	type	description
hamiltonian.drivingFields[].detuning.time_series.times	List[10 進数〕	運転調整の時点、Delta_global(t)
hamiltonian.drivingFields[].detuning.time_series.values	List[10 進数〕	運転調整の値、Delta_global(t)
hamiltonian.drivingFields[].detuning.pattern	str	Delta_global(t) 運転調整の空間パターン。「均一」である必要があります
hamiltonian.localDetuning[].magnitude.time_series.times	List[10 進数〕	ローカル調整の大きさの時間依存係数の時点、Delta_local(t)
hamiltonian.localDetuning[].magnitude.time_series.values	List[10 進数〕	ローカル調整の大きさの時間依存係数 Delta_local(t) の値
hamiltonian.localDetuning[].magnitude.pattern	List[10 進数〕	ローカル調整の大きさのサイト依存係数 h_k (値は setup.ahs_register.sites のサイトに対応します)

メタデータフィールド

プログラムフィールド	type	description
braketSchemaHeader.name	str	スキーマの名前。'braket.ir.ahs.program' である必要があります

プログラムフィールド	type	description
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
(例)

```
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 (例)

```
{
    "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
        }
    }
}
```

メインフィールド

タスク結果フィールド	type	description
measurements[].shotResult.preSequence	List[int]	各ショットのシーケンス前測定ビット (アトミックサイトごとに 1 つ): サイトが空の場合は 0、サイトが満たされている場合は 1、量子進化を実行するパルスのシーケンスの前に測定
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 Rydberg 状態またはサイトが空の場合は 0、原子が地面状態の場合は 1、量子進化を実行するパルスシーケンスの最後に測定

メタデータフィールド

タスク結果フィールド	type	description
braketSchemaHeader.name	str	スキーマの名前。'braket.task_result.analog_hamiltonian_simulation_task_result' である必要があります

タスク結果フィールド	type	description
braketSchemaHeader.version	str	スキーマのバージョン
taskMetadata.braketSchemaHeader.name	str	スキーマの名前。「braket.task_result.task_metadata」である必要があります。
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	type	description
taskMetadata.shots.createdAt	str	作成のタイムスタンプ。形式は ISO-8601/RFC3339 文字列形式 YYYY-MM-DDTHH:mm:ss.sssZ である必要があります。デフォルトは None です。
taskMetadata.shots.endedAt	str	量子タスクが終了したときのタイムスタンプ。形式は ISO-8601/RFC3339 文字列形式 YYYY-MM-DDTHH:mm:ss.sssZ である必要があります。デフォルトは None です。

タスク結果フィールド	type	description
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' である必要があります
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	type	description
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の数と同じである必要があります
measurements[].shotMetadata.shotStatus	int	ショットのステータス (成功、部分成功、失敗) 。「成功」である必要があります

QuEra デバイスプロパティスキーマ

braket.device_schema.quera.quera_device_capabilities_v1.QueraDeviceCapabilities (例)

```
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 (例)

```
{
    "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フィールド

サービスプロパティフィールド	type	description
service.executionWindows[].executionDay	ExecutionDay	実行ウィンドウの日数。「毎日」、「平日」、「週末」、「月曜日」、「火曜日」、「水曜日」、木曜日「金曜日」、「土曜日」、または「日曜日」である必要があります
service.executionWindows[].windowStartHour	datetime.time	実行ウィンドウが開始される時刻の UTC 24 時間形式
service.executionWindows[].windowEndHour	datetime.time	実行ウィンドウが終了する時刻の 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	フロート	米ドル単位のデバイスの料金
service.qpu_capabilities.service.deviceCost.unit	str	料金の請求単位。例:「分」、「時間」、「ショット」、「タスク」

メタデータフィールド

メタデータフィールド	type	description
action[].version	str	AHS プログラム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' である必要があります
service.braketSchemaHeader.name	str	スキーマの名前。'braket.device_schema.device_service_properties' である必要があります
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 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時刻

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、オブジェクト指向の とAPI、Amazon Braket への低レベルアクセスを提供します。

「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。

たとえば、 を呼び出すと`CreateQuantumTask`、Amazon Braket SDK は Boto3 にリクエストを送信し、 を呼び出します AWS API。

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 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API。次の例では、Boto3 クライアントの名前は ですbraket。

```
import boto3
import botocore

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

> **Note**
>
> Braket は IPv6 をサポートしています。IPv6-onlyネットワークを使用している場合、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する場合は、デュアルスタックおよび FIPS エンドポイントガイドで説明されているように、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
- 量子タスクの検索
- 量子タスクのキャンセル

デバイスを検索します。

- search_devices(**kwargs)

指定され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 で使用できる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)

指定され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'])
```

量子タスクの検索

- search_quantum_tasks(**kwargs)

指定され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']}")
```

量子タスクのキャンセル

- cancel_quantum_task(quantumTaskArn)

指定され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 を実行するときは、デフォルトのままにすることをお勧めします。ノートブックインスタンスを起動するアクセス許可を持つ IAM ロールを指定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、このような CLI を設定する方法`profile`と、そのプロファイルを Amazon Braket に組み込む方法を簡単に説明し、そのプロファイルからのアクセス許可でAPI呼び出しが行われるようにします。

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確立する
- 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とデフォルト以外のプロファイルの設定方法については、このドキュメントの範囲外です。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。

- 入門
- を使用する AWS CLI ための の設定 AWS IAM Identity Center

Amazon Braket を使用するには、このユーザー、および関連する CLI `profile`に必要な Braket アクセス許可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確立する

Boto3 セッションオブジェクトを確立するには、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**
>
> 予想されるAPI呼び出しにprofileデフォルトのリージョンと一致しないリージョンベースの制限がある場合は、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、us-east-1、 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-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、量子タスクをインスタンス化されたAwsDeviceオブジェクトに送信できます (たとえば、 device.run(…) コマンドを呼び出します)。そのデバイスによって行われたすべてのAPI呼び出しは、以前に として指定した CLI プロファイルに関連付けられた IAM 認証情報を活用できますprofile。

Amazon Braket による量子タスクのテスト

Amazon Braket には、実際の量子ハードウェアで実行する前に量子アルゴリズムをテストおよび検証するのに役立つ、さまざまな高性能量子回路シミュレーターが用意されています。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) クラスターを処理し、従来の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と、量子アプリケーションの開発と最適化に集中できます。

Braket のシミュレーターを使用すると、物理量子デバイスの制約や制限なしに、量子回路とアルゴリズムを徹底的にテストできます。これにより、基本的な量子ゲートや回路から、より高度な量子アルゴリズムやエラー軽減技術まで、幅広い量子コンピューティングの概念を探索できます。

Braket SDK を使用すると、シミュレーターに量子タスクを簡単に送信できるため、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して、量子アルゴリズムの動作をよりよく理解できます。Amazon Braket Hybrid Job の機能を活用して、古典的な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て、テストと検証の範囲をさらに拡大することもできます。

Braket のシミュレーターで量子タスクを徹底的にテストすることで、実際の量子ハードウェアにデプロイする前に、貴重なインサイトを得て、アルゴリズムを改良し、その正確性を確認できます。これにより、開発時間を短縮し、エラーを最小限に抑え、最終的に量子コンピューティングの分野での進捗状況を加速できます。

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
- Amazon Braket Hybrid Jobs の使用

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いくつかのシミュレーターへのアクセスを提供します。量子タスクは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。

シミュレーター

- 密度行列シミュレーター、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`
- Tensor Network Simulator、TN1: `arn:aws:braket:::device/quantum-simulator/amazon/tn1`
- ローカルシミュレーター : `LocalSimulator()`

> **Note**
>
> QPUs およびオンデマンドシミュレーターの CREATED状態の量子タスクをキャンセルできます。状態の量子タスクは、オンデマンドシミュレーターと QPUs に対してベストエフォートベースQUEUEDでキャンセルできます。QPU 量子タスクは、QPU QUEUED の可用性ウィンドウ中に正常にキャンセルされる可能性は低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)
- 局所密度行列シミュレーター (braket_dm)
- ローカル AHS シミュレーター (braket_ahs)
- ステートベクトルシミュレーター (SV1)
- 密度行列シミュレーター (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 シミュレーターを比較する
- Amazon Braket 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
- 量子タスクのバッチ処理

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、 環境でローカルに実行される Amazon Braket SDK の一部です。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、小さな回路 (最大 25 個qubits) でのラピッド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 Amazon Braket SDK のすべてのゲートをサポートしますが、QPU デバイスはより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。

> **Note**
>
> ローカルシミュレーターは、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可能性がある高度な OpenQASM 機能をサポートしています。サポートされている機能の詳細については、OpenQASM Local Simulator ノートブックで提供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。

局所密度行列シミュレーター (`braket_dm`)

ローカル密度行列シミュレーター (`braket_dm`) は、 環境でローカルに実行される Amazon Braket SDK の一部です。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 個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、既存の回路の特定の qubitsおよびゲートにノイズオペレーション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は、指定された数の がある場合、次の結果を提供できます shots。

- 低密度マトリックス: Shots = 0

> **Note**
>
> 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、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 ノートブックで提供されている例を参照してください。

局所密度行列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 (Analog Hamiltonian Simulation) シミュレーター (braket_ahs) は、 環境でローカルに実行される Amazon Braket SDK の一部です。AHS プログラムの結果をシミュレートするために使用できます。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、小さなレジスタ (最大 10 ~ 12 個のアトム) でのプロトタイプ作成に適しています。

ローカルシミュレーターは、1 つの均一な運転フィールド、1 つの (不均一な) シフトフィールド、および任意の原子配置を持つ AHS プログラムをサポートします。詳細については、Braket AHS クラスと Braket AHS プログラムスキーマ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: Run your first=" Hamiltonian Simulation ページと、Achemal Hamiltonian Simulation サンプルノートブックを参照してください。

ステートベクトルシミュレーター (SV1)

SV1 は、オンデマンドで高性能なユニバーサルステートベクトルシミュレーターです。最大 34 個のの回路をシミュレートできますqubits。34-qubit、高密度、および正方形の回路 (回路深度 = 34) は、使用するゲートのタイプやその他の要因に応じて、完了までに約 1 ~ 2 時間かかることが予想されます。all-to-allゲートを持つ回路は、 に適していますSV1。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 の最大ランタイムは 6 時間です。デフォルトでは、35 個の同時量子タスクがあり、最大 100 個の同時量子タスク (us-west-1 および eu-west-2 では 50 個) があります。

SV1 結果

SV1 は、指定された数の がある場合、次の結果を提供できますshots。

- サンプル: Shots > 0
- 期待値: Shots >= 0
- 分散: Shots >= 0
- 確率: Shots > 0
- 振幅: Shots = 0
- 結合グラデーション: 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、複数の回路を並行して実行できます。ランタイムは、オペレーションの数に応じて直線的にスケールされ、 の数に応じて指数関数的にスケールされますqubits。の数shotsはランタイムにわずかな影響を与えます。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。

密度行列シミュレーター (DM1)

DM1 は、オンデマンドで高性能な密度行列シミュレーターです。最大 17 の回路をシミュレートできますqubits。

DM1 の最大ランタイムは 6 時間、デフォルトは 35 個の同時量子タスク、最大 50 個の同時量子タスクです。

DM1 結果

DM1 は、指定された数の がある場合、次の結果を提供できますshots。

- サンプル: Shots > 0
- 期待値: Shots >= 0
- 分散: Shots >= 0
- 確率: Shots > 0
- 低密度マトリックス: Shots = 0、最大 8 qubits

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、複数の回路を並行して実行できます。ランタイムは、オペレーションの数に応じて直線的にスケールされ、 の数に応じて指数関数的にスケールされますqubits。の数shotsはランタイムにわずかな影響を与えます。詳細については、「Compare Simulators」を参照してください。

ノイズゲートと制限

```
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、高性能、テンソルネットワークシミュレーターです。 は、最大 50 個、qubits回路深度が 1,000 個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パスを識別しようとします。そのため、 TN1は収縮フェーズと呼ばれる次のステージのランタイムを推定できます。推定収縮時間がTN1シミュレーションランタイム制限を超える場合、TN1は収縮を試みません。

TN1 のランタイム制限は 6 時間です。最大 10 (eu-west-2 では 5) の同時量子タスクに制限されています。

TN1 結果

収縮段階は、一連の行列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続行されます。

注: は > 0 Shots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、複数の回路を並行して実行できます。詳細については、「Compare Simulators」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 サンプルノートブックの Amazon Braket GitHub リポジトリにアクセスして、 の使用を開始してくださいTN1。

を使用するためのベストプラクティス TN1

- オールツーオール回路は避けてください。
- 少数の で新しい回路または回路のクラスをテストしてshots、 の回路の「剛性」を学習します TN1。
- 大規模なshotシミュレーションを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によって動作します。また、同じコンテナ内に含まれ、ハイブリッドジョブインスタンスで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に通常関連するボトルネックを取り除くのに役立ちます。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、組み込みシミュレーターはメモリ要件を大幅に削減し、ターゲット結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セットアップと比較して、多くの場合 10 倍以上のパフォーマンスが大幅に向上する可能性があります。組み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有効にする方法の詳細については、Amazon Braket Hybrid Jobs でハイブリッドジョブを実行する」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、結合の区別などの高度な勾配計算方法を活用

して、勾配をより速く評価できます。Lightning.qubit シミュレーターは Braket NBIs を介してデバイスとして、また埋め込みシミュレーターとして使用できますが、Lightning.gpu シミュレーターは GPU インスタンスを備えた埋め込みシミュレーターとして実行する必要があります。Lightning.gpu の使用例については、Braket Hybrid Jobs ノートブックの「埋め込みシミュレーター」を参照してください。

Amazon Braket シミュレーターを比較する

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することで、量子タスクに最適な 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の選択 (SV1、TN1、DM1)

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される Braket ノートブックインスタンスなど、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を追加します。これは、多くの量子タスクが関与する場合、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、以下のガイダンスは、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：

- 18 未満の を使用する場合はqubits、ローカルシミュレーターを使用します。
- 18～24 の を使用する場合はqubits、ワークロードに基づいてシミュレーターを選択します。
- 24 個を超える を使用する場合はqubits、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：

- 9 未満の を使用する場合はqubits、ローカルシミュレーターを使用します。
- 9～12 個の を使用する場合はqubits、ワークロードに基づいてシミュレーターを選択します。
- 12 個を超える を使用する場合はqubits、 を使用しますDM1。

状態ベクトルシミュレーターとは何ですか?

SV1 はユニバーサルステート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。シミュレータSV1ーの量子タスクの実行時間は、回路内のゲートの数に応じて直線的に増加します。

密度行列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。システムの全密度行列を保存し、回路のゲートとノイズ操作を順番に適用します。最終的な密度行列には、回路実行後の量子状態に関する完全な情報が含まれています。ランタイムは通常、オペレーションの数に応じて直線的にスケールされ、の数に応じて指数関数的にスケールされますqubits。

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 または で構成されますqubits。
- グラフのエッジは、ゲート間の接続を表します。

この構造の結果として、 TN1は比較的大きく複雑な量子回路のシミュレートされたソリューション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です

通常、 は量子計算をシミュレートするための 2 フェーズアプローチでTN1動作します。

- リハーサルフェーズ : このフェーズでは、 はグラフを効率的にトラバースする方法TN1を考え出します。これには、必要な測定値を取得できるようにすべてのノードを訪問することが含まれます。は両方のフェーズを一緒にTN1実行するため、顧客にはこのフェーズは表示されません。最初のフェーズを完了し、実用的な制約に基づいて 2 番目のフェーズを単独で実行するかどうかを決定します。シミュレーションの開始後、その決定への入力はありません。
- 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。フェーズは、一連の行列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が実行されます。リハーサルフェーズ中に収縮パスが見つかったら、 は回路のゲートをまとめてTN1シミュレーションの結果を生成します。

TN1 グラフはマップに似ています

比喩的に、基盤となるTN1グラフを都市の通りと比較できます。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実行しなかった場合は、最初に地図を見るのではなく、都市の通りを歩いて目的地を見つけるように見えます。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、トラバースする基盤となる回路の構造に特定の「認識」があると言えます。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、主に一定数の qubitsおよび ゲートを持つことに依存する問題のクラスに最適です。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、 の数には依存しませんshots。 SV1は一般的に 28 未満のTN1回路の場合よりも高速ですqubits。

SV1 は、非常に可能性の低いものであっても、実際にはすべての可能性をシミュレートするため、qubit数値が大きいほど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評価のために、 は 2^30 設定を計算するSV1必要があります。qubits Amazon Braket SV1 Simulator の 34 の制限は、メモリとストレージの制限による実用的な制約です。これを次のように考えることができます。 qubitを に追加するたびにSV1、問題は 2 倍の困難になります。

多くのクラスの問題では、 TN1はグラフの構造を利用するSV1ため、 よりもはるかに大きな回路を現実的な時間で評価TN1できます。基本的には、最初からソリューションの進化を追跡し、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。別の方法として、行列乗算の順序を作成するための設定を保存し、評価プロセスを簡素化します。

ではTN1、 qubitsゲートと ゲートの数は重要ですが、グラフの構造はさらに重要です。例えば、TN1は、ゲートが短い (つまり、それぞれqubitがゲートによって最も近い隣接する にのみ接続されている) 回路 (グラフqubits) と、接続 (またはゲート) の範囲が似ている回路 (グラフ) を評価するのに非常に適しています。の一般的な範囲TN1は、それぞれが 5 qubitsの距離にある他の とのみqubit通信qubitsすることです。構造の大部分をより単純な関係に分解できる場合、このような関係は、より詳細、より詳細、またはより均一なマトリックスで表すことができます。 は評価を簡単にTN1実行します。

の制限 TN1

TN1 は、グラフの構造の複雑さSV1に応じて遅くなる可能性があります。特定のグラフでは、 はリハーサルステージの後にシミュレーションTN1を終了し、次の 2 つの理由のいずれかで FAILEDのステータスを表示します。

- パスを見つけることができない - グラフが複雑すぎる場合、正常なトラバーサルパスを見つけるのは難しすぎ、シミュレーターは計算をあきらめます。 は収縮を実行TN1できません。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、 はトラバーサルパスを見つけるTN1ことができますが、評価に非常に長く、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膨大なものになり、代わりにリハーサルフェーズの後にTN1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**
>
> 収縮が実行されず、 FAILEDステータスが表示されるTN1場合でも、 のリハーサルステージに対して課金されます。

予測ランタイムは、shotカウントにも依存します。最悪のシナリオでは、TN1収縮時間はshotカウントに直線的に依存します。回路は、より少ない で収縮できる場合がありますshots。例えば、100 ので量子タスクを送信できます。これはshots、 が契約不可能であるTN1と判断しますが、10 の のみで再送信すると、収縮が続行されます。このような状況では、100 個のサンプルを得るために、同じ回路shotsに対して 10 個の量子タスクを 10 個送信し、その結果を最後に組み合わせることができます。

ベストプラクティスとして、数個 shots (10 など) の回路または回路クラスで常にテストし、 の数を増やす前にTN1、 の回路のハードさを調べることをお勧めしますshots。

> **Note**
>
> 収縮フェーズを形成する一連の乗算は、小さな NxN マトリックスから始まります。例えば、2-qubit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。そのため、AmazonBraket は非常に複雑な収縮を試みません。

同時実行

Braket シミュレーターはすべて、複数の回路を同時に実行することができます。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。同時実行数の制限の詳細については、「クォータ」ページを参照してください。

Amazon Braket 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、量子タスクの例を実行する段階について説明します。Amazon Braket のベストプラクティスとして、まず 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SV1。

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
- サンプル結果の表示

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 を選択する方法を示しますSV1。

```
# choose the on-demand simulator to run the circuit
from braket.aws import AwsDevice
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
```

このデバイスのプロパティの一部は、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', <DeviceActionType.JAQCD: 'braket.ir.jaqcd.program'>)
('supportedOperations', ['ccnot', 'cnot', 'cphaseshift', 'cphaseshift00',
 'cphaseshift01', 'cphaseshift10', 'cswap', 'cy', 'cz', '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)])
```

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。

```
# 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 = "amzn-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()` コマンドは、 `CreateQuantumTask` API を使用して量子タスクを作成します。初期化時間が短いと、量子タスクは、デバイス上で量子タスクを実行する容量が存在するまでキューに入れられます。この場合、デバイスは ですSV1。デバイスが計算を完了すると、AmazonBraket 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` は口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 オンデマンドシミュレーターとローカルシミュレーター、QPUsもサポートしています。これを行うには、次の例に示すようにdevice.run()、 の inputs引数を使用します。は文字列と浮動小数点のペアのディクショナリinputsである必要があります。キーはパラメータ名です。

パラメトリックコンパイルは、特定の QPUs。サポートされている QPU に量子タスクとしてパラメータ回路を送信すると、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、最高品質の結果を確保します。

> **Note**
>
> パラメトリックコンパイルは、パルスレベルプログラムRigetti Computingを除き、 からのすべての超電導ゲートベースの QPUs でサポートされています。

```
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 を指定する

shots 引数は、必要な測定 の数を参照しますshots。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SV1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。(では使用できません) TN1。
- ゼロ以外の値の場合shots、シミュレーターは出力分布からサンプリングして実際の QPU のshotノイズをエミュレートします。 QPUs QPU デバイスは shots > 0 のみを許可します。

量子タスクあたりの最大ショット数については、「Braket Quotas」を参照してください。

結果のポーリング

を実行するとmy_task.result()、SDK は量子タスクの作成時に定義したパラメータを使用して結果のポーリングを開始します。

- poll_timeout_seconds は、オンデマンドシミュレーターや QPU デバイスで量子タスクを実行するときにタイムアウトする前に量子タスクをポーリングする秒数です。デフォルト値は 432,000 秒 (5 日) です。
- 注： Rigettiや などの QPU デバイスの場合はIonQ、数日かかることをお勧めします。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。オンデマンドシミュレーターおよび QPU デバイスで量子タスクが実行されたときに、Braket を呼び出してステータスAPIを取得する頻度を指定します。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:
```

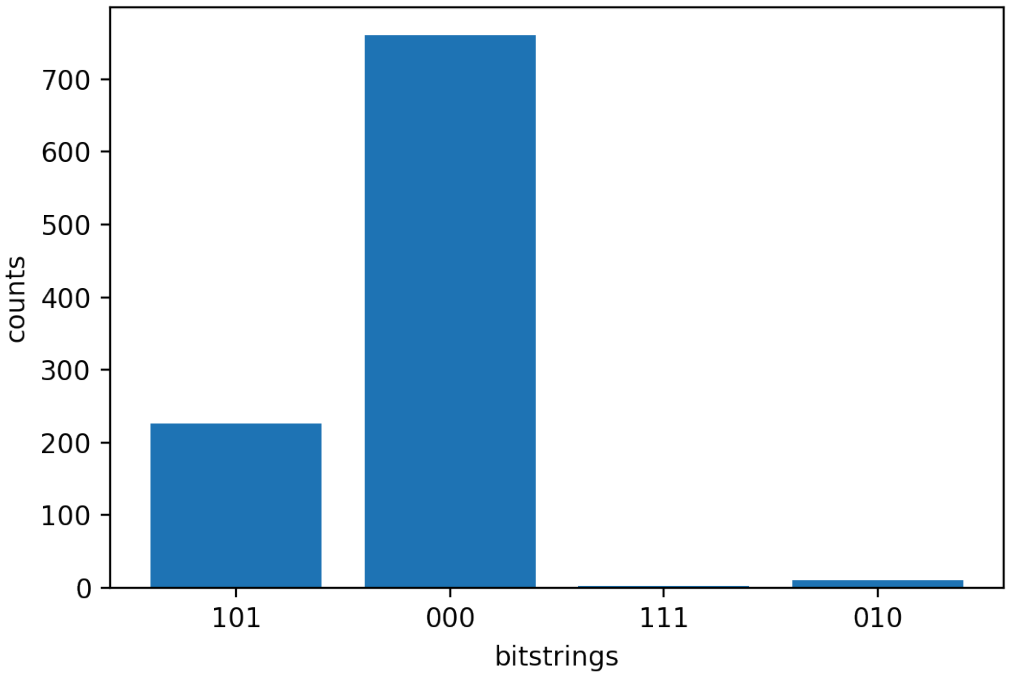
```
 [[1 0 1]
 [0 0 0]
 [1 0 1]
 ...
 [0 0 0]
 [0 0 0]
 [0 0 0]]
Counts for collapsed states:
 Counter({'000': 761, '101': 226, '010': 10, '111': 3})
Probabilities for collapsed states:
 {'101': 0.226, '000': 0.761, '111': 0.003, '010': 0.01}
```

サンプル結果の表示

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])

# you can 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={'observable': ['z'], 'targets': [0], 'type': 'variance'},
 value=0.7062359999999999), ResultTypeValue(type={'targets': [0, 2], 'type':
 'probability'}, value=array([0.771, 0.   , 0.   , 0.229]))]
Variance= 0.7062359999999999
Probability= [0.771 0.    0.    0.229]
```



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

量子タスクをローカルシミュレーターに直接送信して、ラピッド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qubitsーが処理できる実行速度と最大数は、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仕様によって異なります。

以下のコマンドはすべて同一で、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)
```

ローカル密度行列 (ノイズ) シミュレーターをインスタンス化するために、お客様はバックエンドを次のように変更します。

```
# 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の特定の量子ビットの測定

ローカル状態ベクトルシミュレーターとローカル密度行列シミュレーターは、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。これは、多くの場合、部分測定と呼ばれます。

例えば、次のコードでは、2 量子ビット回路を作成し、ターゲット量子ビットを持つmeasure命令を回路の末尾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)
```

量子タスクのバッチ処理

量子タスクバッチ処理は、ローカルシミュレーターを除くすべての 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 (TN1 または SV1) で実行する量子タスクに特に役立ちます。さまざまな量子タスクのセットアップに役立つように、Amazon Braket にはサンプルノートブックが用意されています。

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、10 個の量子タスクを必要とする計算を行い、それらの量子タスクの回路が互いに独立している場合は、バッチ処理を使用することをお勧めします。これにより、ある量子タスクが完了するのを待ってから別のタスクを開始する必要がなくなります。

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。

```
circuits = [bell for _ in range(5)]
batch = device.run_batch(circuits, s3_folder, shots=100)
print(batch.results()[0].measurement_counts)  # The result of the first quantum task in
 the batch
```

詳細については、GitHub の Amazon Braket の例またはバッチ処理に関するより具体的な情報を含む量子タスクバッチ処理を参照してください。

このセクションの内容:

- 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

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のコストについては、いくつかの注意点があります。

- デフォルトでは、量子タスクバッチ処理はすべてのタイムアウトを再試行するか、量子タスクを 3 回失敗させます。
- qubits の 34 など、長時間実行される量子タスクのバッチにはSV1、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`割り当て値を注意深く再確認してください。TN1 で を使用することはお勧めしません`run_batch`。
- 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 の説明を参照してください)。自動再試行によってコストが増加する可能性があるた

め、 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 TN1 (「量子タスクバッチ処理」、186 行目を参照)。

量子タスクのバッチ処理と PennyLane

次の例に示すように、Amazon Braket で PennyLane を使用している場合は、Amazon Braket デバイスをインスタンス化parallel = Trueするときに を設定して、バッチ処理を活用します。

```
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 によるバッチ処理の詳細については、「量子回路の並列最適化」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ときは、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listの を指定できます。この場合、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

# 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、単一のパラメトリック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)
```

Amazon Braket Hybrid Jobs の使用

このセクションでは、Amazon Braket でハイブリッドジョブを作成および実行する基本的な手順について説明します。

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:

- 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 でハイブリッドジョブを実行する
- 最初のハイブリッドジョブを作成する
- ジョブ結果の保存
- チェックポイントを使用したハイブリッドジョブの保存と再起動
- ローカルモードでのハイブリッドジョブの構築とデバッグ

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、Amazon EC2 コンピューティングリソースと Amazon Braket Quantum Processing Unit (QPU) アクセスを組み合わせた、ハイブリッド量子クラシック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キューイングされるため、量子タスクキュー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 は個別のハイブリッドジョブキューを維持し、一度に実行できるハイブリッドジョブは 1 つだけです。

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データを保存してロードする
- ハイブリッドジョブデコレータの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 Job として実行できます。これを行うには、次のコード例に示すように、コードにデ@hybrid_jobコレータで注釈を付けます。カスタム環境では、Amazon Elastic Container Registry (ECR) のカスタムコンテナを使用することを選択できます。

> **Note**
>
> デフォルトでは、Python 3.10 のみがサポートされています。

デ@hybrid_job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 インスタンスの デコレータ内で 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、 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State Vector Simulator (SV1) device。

```
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 関数を呼び出します。ただし、デコレータ関数は、関数の結果ではなくハイブリッドジョブハンドルを返します。完了後に結果を取得するには、 を使用しますjob.result()。

```
job = run_hybrid_job(num_tasks=1)
result = job.result()
```

@hybrid_job デコレータの device 引数は、ハイブリッドジョブが優先的にアクセスできるデバイスを指定します。この場合はシミュレータSV1ーです。QPU の優先度を取得するには、 関数内で使用されるデバイス ARN が、 デコレータで指定されたものと一致することを確認する必要があります。便宜上、 ヘルパー関数を使用してget_job_device_arn()、 で宣言されたデバイス ARN をキャプチャできます@hybrid_job。

> **Note**
>
> 各ハイブリッドジョブは、Amazon EC2 にコンテナ化された環境を作成しているため、起動時間が少なくとも 1 分あります。したがって、1 つの回路や 1 つの回路のバッチなど、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()` 関数は 引数`num_tasks`を使用して、作成された量子タスクの数を制御します。ハイブリッドジョブは、これ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 Braket コンソールに文字列として表示され、2500 文字に制限されます。

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は で記録されます`log_metrics`。メトリクスは、ハイブリッドジョブタブの Braket コンソールページ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して、ハイブリッドジョブ実行中の量子タスク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、結果タイプから最初の確率を記録するメトリクス名「確率」を使用しています。

結果の取得

ハイブリッドジョブが完了したら、 `job.result()` を使用してハイブリッドジョブの結果を取得します。return ステートメントのすべてのオブジェクトは Braket によって自動的にキャプチャされます。関数によって返されるオブジェクトは、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、次のコードは動作中の例と失敗の例を示しています。

```
@hybrid_job(device=Devices.Amazon.SV1)
def passing():
    np_array = np.random.rand(5)
    return np_array  # serializable

@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から推測されます。最大 50 文字のカスタム名を指定することもできます。たとえば、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、 引数をデlocal=Trueコレータに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。ローカルジョブには、量子タスクの優先度キューイングはありません。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、デバイスを SV1 シミュレーターとして使用してローカルハイブリッドジョブを作成します。

```
@hybrid_job(device=Devices.Amazon.SV1, local=True)
def run_hybrid_job(num_tasks = 1):
    return ...
```

その他のハイブリッドジョブオプションはすべて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 モジュール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

任意の Python パッケージを使用するようにランタイム環境をカスタマイズできます。requirements.txt ファイル、パッケージ名のリスト、または独自のコンテナ (BYOC) を使用できます。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、次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 ...
```

たとえば、 requirements.txt ファイルにはインストールする他のパッケージが含まれている場合があります。

```
qiskit
pennylane >= 0.31
mitiq == 0.29
```

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データを保存してロードする

入力トレーニングデータの指定

ハイブリッドジョブを作成するときは、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指定できます。ローカルパスを指定し、Braket はで Amazon S3 s3://<default_bucket_name>/jobs/<job_name>/<timestamp>/data/<channel_name> にデータを自動的にアップロードすることもできます。ローカルパスを指定すると、チャネル名はデフォルトで「入力」になります。次のコードは、ローカルパス からの numpy ファイルを示していますdata/file.npy。

```
@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。

```
s3_path = "s3://amazon-braket-us-west-1-961591465522/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 URIs またはローカルパスのディクショナリを指定することで、複数の入力データソースを指定できます。

```
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場合 (>1GB)、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、S3 バケットに最初にアップロードされたときのローカル入力データが原因で、S3 パスがジョブリクエストに追加されます。最後に、ジョブリクエストが Braket サービスに送信されます。

S3 への結果の保存

修飾された関数の return ステートメントに含ま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。

```
@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、 という名前のファイルに圧縮されますmodel.tar.gz。結果は、Python 関数 job.result() でダウンロードするか、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から結果フォルダに移動することでダウンロードできます。

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。組み込みへsave_job_checkpoint()ルパー関数を使用するか、AMZN_BRAKET_JOB_RESULTS_DIRパスにファイルを保存できます。ヘルパー関数 では、後でを使用できますget_job_results_dir()。

以下は、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ための最小限の実例です。

```
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()は、保存するデータを含むディクショナリで呼び出されます。デフォルトでは、すべての値をテキストとしてシリアル化する必要があ

ります。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を作成するには、を設定できますdata_format = PersistedJobDataFormat.PICKLED_V4。このコードは、「checkpoints」というサブフォルダのハイブリッドジョブアーティファクト<jobname>.jsonにデフォルト名でチェックポイントファイルを作成して上書きします。

チェックポイントから続行する新しいハイブリッドジョブを作成するには、 が前のジョブのハイブリッドジョブ ARN copy_checkpoints_from_job=job_arnjob_arnである を渡す必要があります。次に、 load_job_checkpoint(job_name)を使用してチェックポイントからロードします。

ハイブリッドジョブデコレータのベストプラクティス

非同期性を採用する

デコレータ注釈で作成されたハイブリッドジョブは非同期です。古典リソースと量子リソースが利用可能になると実行されます。アルゴリズムの進行状況は、 Braket Management Consoleまたは Amazon CloudWatch を使用してモニタリングします。アルゴリズムを実行して送信すると、Braket はスケーラブルなコンテナ化された環境でアルゴリズムを実行し、アルゴリズムが完了すると結果を取得します。

反復バリエーションアルゴリズムを実行する

ハイブリッドジョブは、反復量子古典アルゴリズムを実行するツールを提供します。純粋な量子問題の場合は、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 QPUsは、古典的な処理で QPUs への複数回の反復呼び出しを必要とする長時間実行されるバリエーションアルゴリズムに最も有益です。

ローカルモードを使用したデバッグ

QPU でハイブリッドジョブを実行する前に、まずシミュレーター SV1 で を実行して、期待どおりに実行されること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 を実行して、迅速な反復とデバッグを行うことができます。

Bring your own container (BYOC) による再現性の向上

コンテナ化された環境内でソフトウェアとその依存関係をカプセル化して、再現可能な実験を作成します。すべてのコード、依存関係、および設定をコンテナにパッケージ化することで、競合やバージョニングの問題が発生するのを防ぐことができます。

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内の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、「埋め込みシミュレーター」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 QPUs で自動的にコンパイルされ、アルゴリズムのランタイムが向上します。 ???

定期的にチェックポイントする

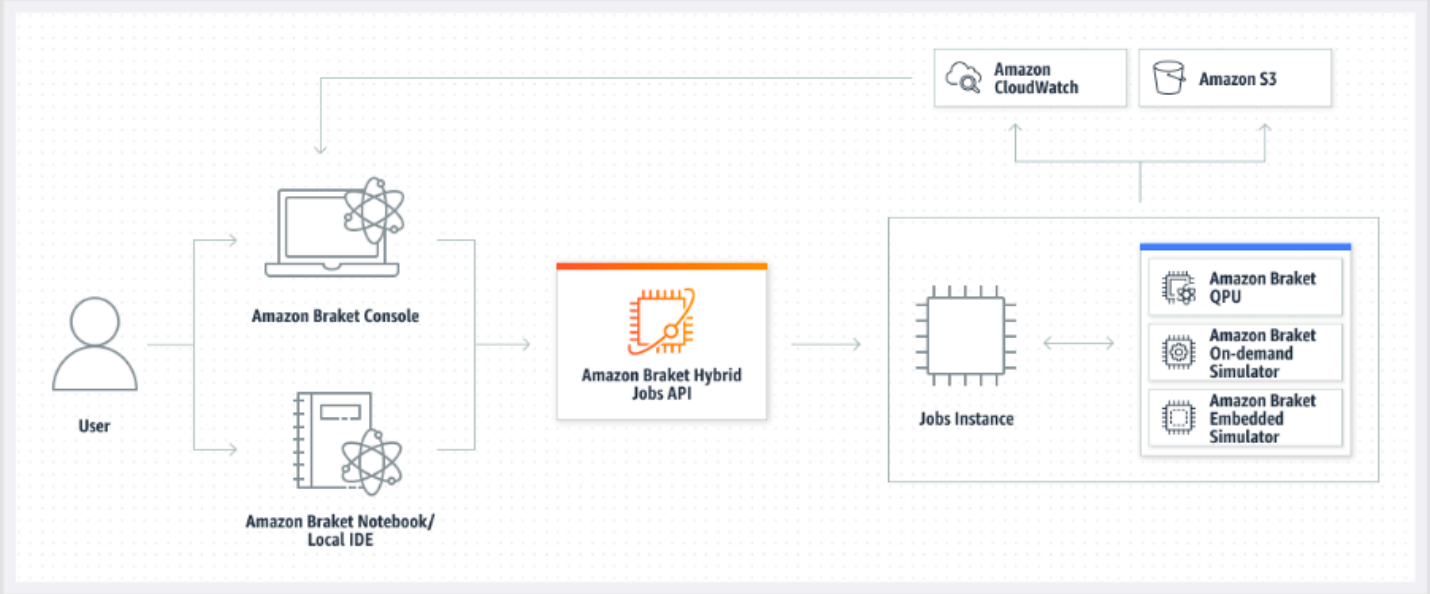
長時間実行されるハイブリッドジョブの場合、アルゴリズムの中間状態を定期的に保存することをお勧めします。

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、Amazon Braket examples GitHub」を参照してください。

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独自の) ライブラリを使用する場合は、これらのライブラリを含む Docker を使用して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 を使用してハイブリッドジョブを作成します。ここでAPI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がハイブリッドジョブを実行するには、QPU、オンデマンドシミュレーター (、、 などDM1TN1) SV1、または従来の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 を使用すると、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。組み込みシミュレーターを使用すると、シミュレーターは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るため、使用できます。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と使用するインスタンスの数を指定できます。各選択肢に関連するコストについては、Amazon Braket の料金表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を起動し (API呼び出しでインスタンスタイプをカスタマイズできます) 、アルゴリズムを実行し、結果を Amazon S3 に書き込み、リソースを解放します。このリリースのリソースを使用すると、使用した分に対してのみお支払いいただくことができます。

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 つだけです。キューは、許可される制限を超えないように、実行できるハイブリッドジョブの数を制御するために使用されます。ターゲットデバイスが QPU の場合、ハイブリッドジョブは最初に選択した QPU のジョブキューに入ります。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、デバイスでハイブリッドジョブを実行します。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、ジョブ量子タスクが数分に 1 回 QPU に送信されていれば、ハイブリッド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 量子タスクの前に実行されます。ハイブリッドジョブが完了すると、リソースが解放されます。つまり、使用した分だけ料金が発生します。

> **Note**
>
> デバイスはリージョナルであり、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。

シミュレーターと QPU ターゲットシナリオの両方で、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 の組み込みシミュレーターで使用可能な GPU ベースのシミュレーターのいずれかを使用してください (例: `lightning.gpu`) 。CPU ベースの埋め込みシミュレーター (、 など`braket:default-simulator`) のいずれかを選択した場合`lightning.qubit`、GPU は使用されず、不要なコストが発生する可能性があります。

最初のハイブリッドジョブを作成する

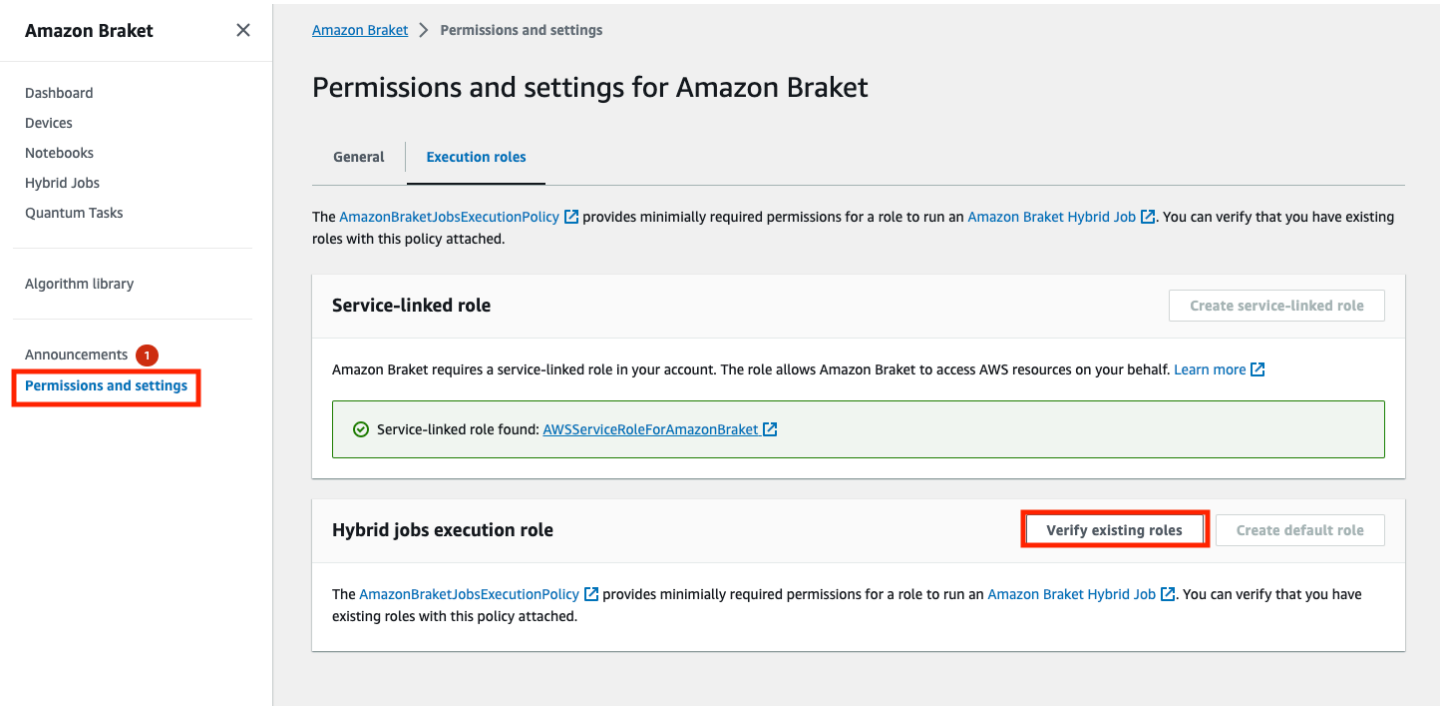
このセクションでは、Python スクリプトを使用してハイブリッドジョブを作成する方法について説明します。または、任意の統合開発環境 (IDE) や Braket ノートブックなどのローカル Python コードからハイブリッドジョブを作成するには、「」を参照してくださいローカルコードをハイブリッドジョブとして実行する。

このセクションの内容:

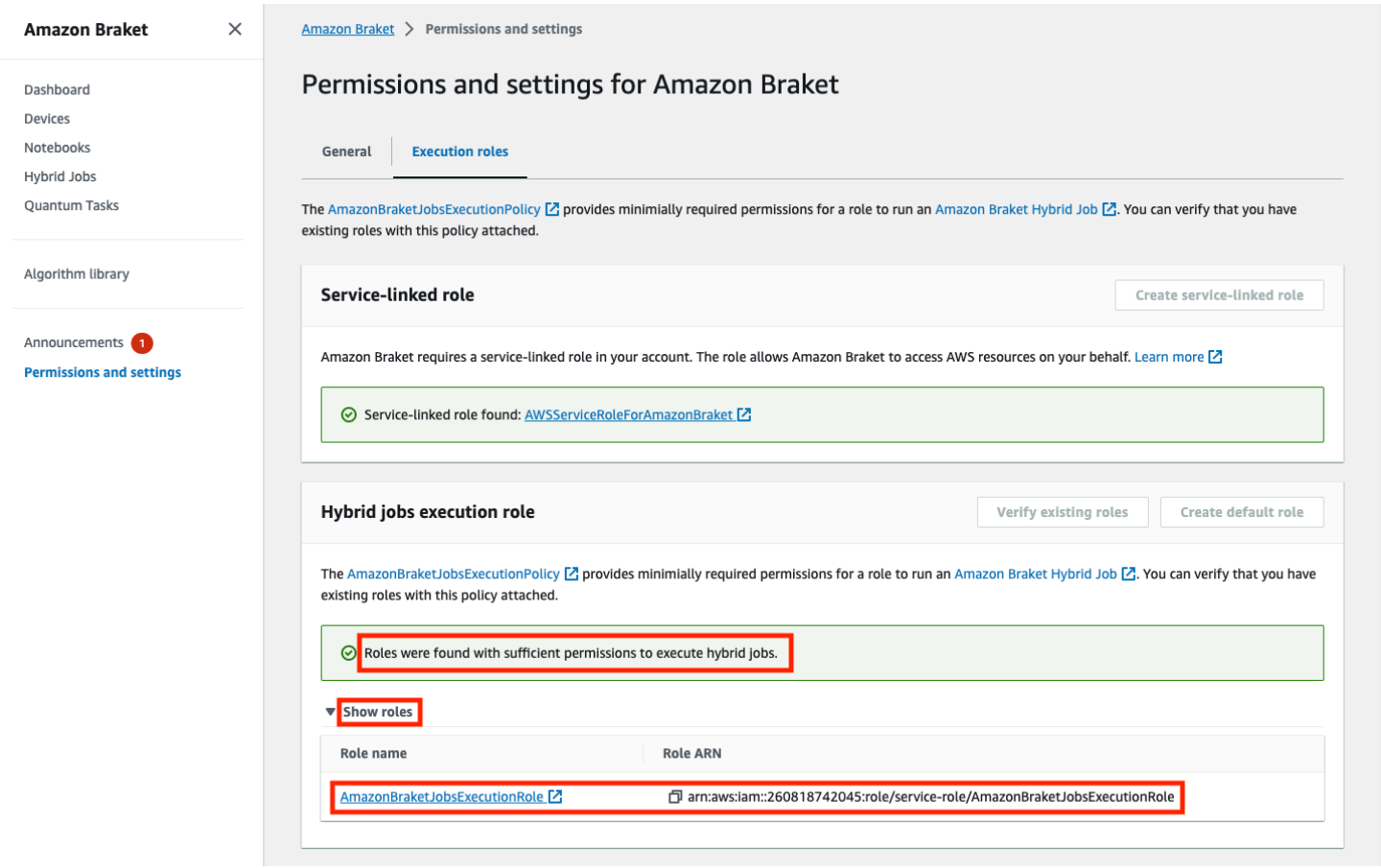
- アクセス許可の設定
- 作成と実行
- 結果をモニタリングする

アクセス許可の設定

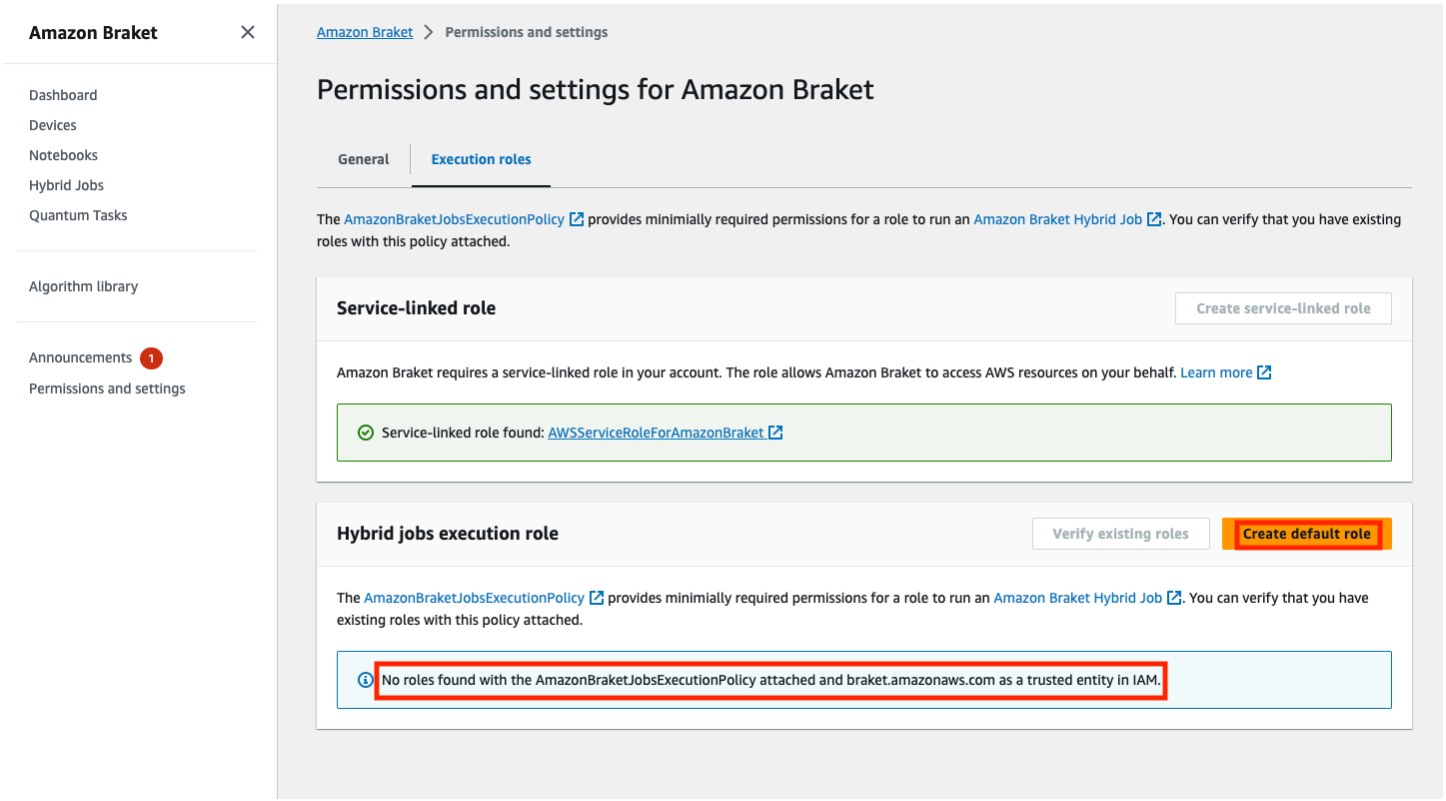
最初のハイブリッドジョブを実行する前に、このタスクを続行するのに十分なアクセス許可があることを確認する必要があります。適切なアクセス許可があることを確認するには、Braket コンソールの左側にあるメニューから[Permissions] (アクセス許可) を選択します。Amazon Braket のアクセス許可管理ページでは、既存のロールの 1 つにハイブリッドジョブを実行するのに十分なアクセス許可があるかどうかを検証するか、そのようなロールがない場合はハイブリッドジョブを実行するために使用できるデフォルトロールの作成をガイドします。



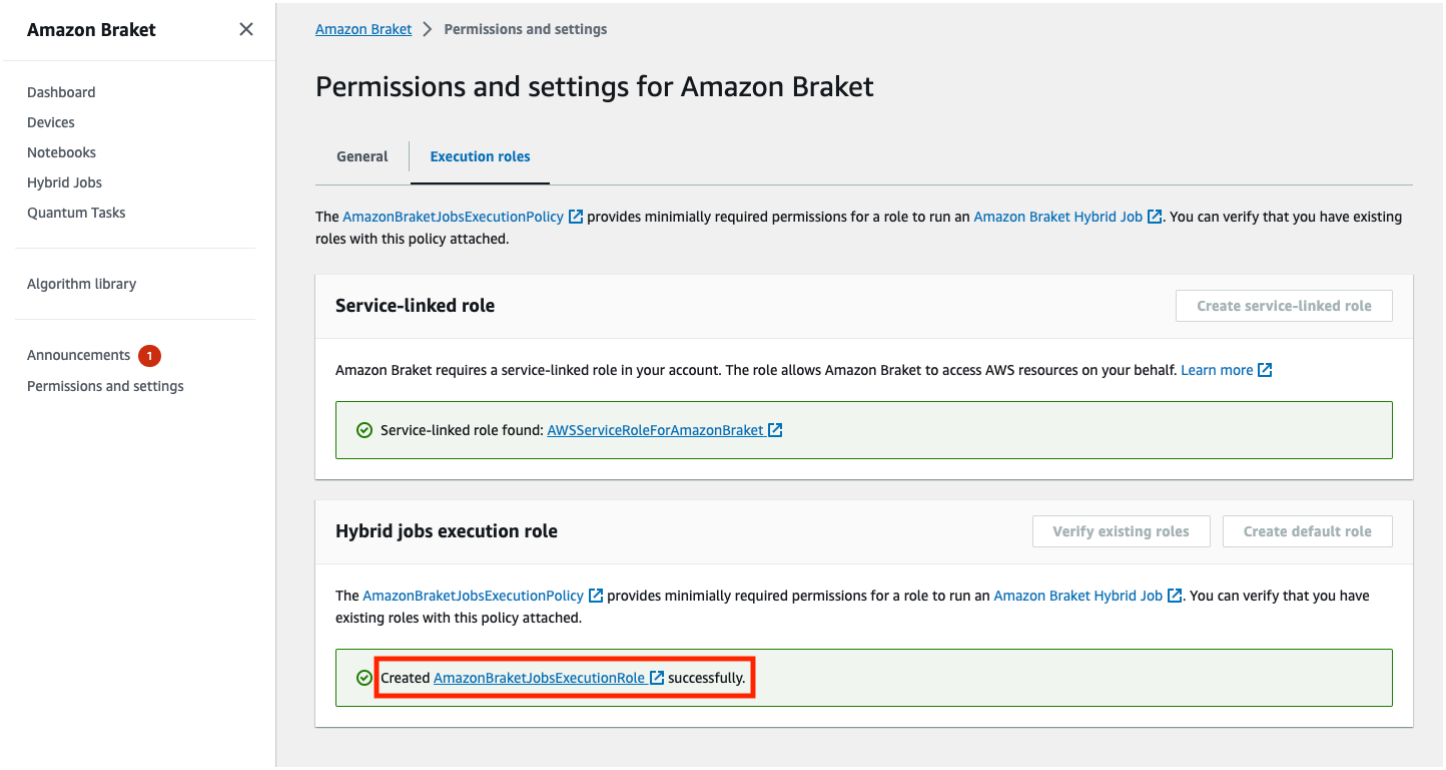
ハイブリッドジョブを実行するのに十分なアクセス許可を持つロールがあることを確認するには、既存のロールの検証ボタンを選択します。使用すると、ロールが見つかったというメッセージが表示されます。ロールの名前とそのロール ARN を表示するには、[Show roles] (ロールを表示する) ボタンを選択します。



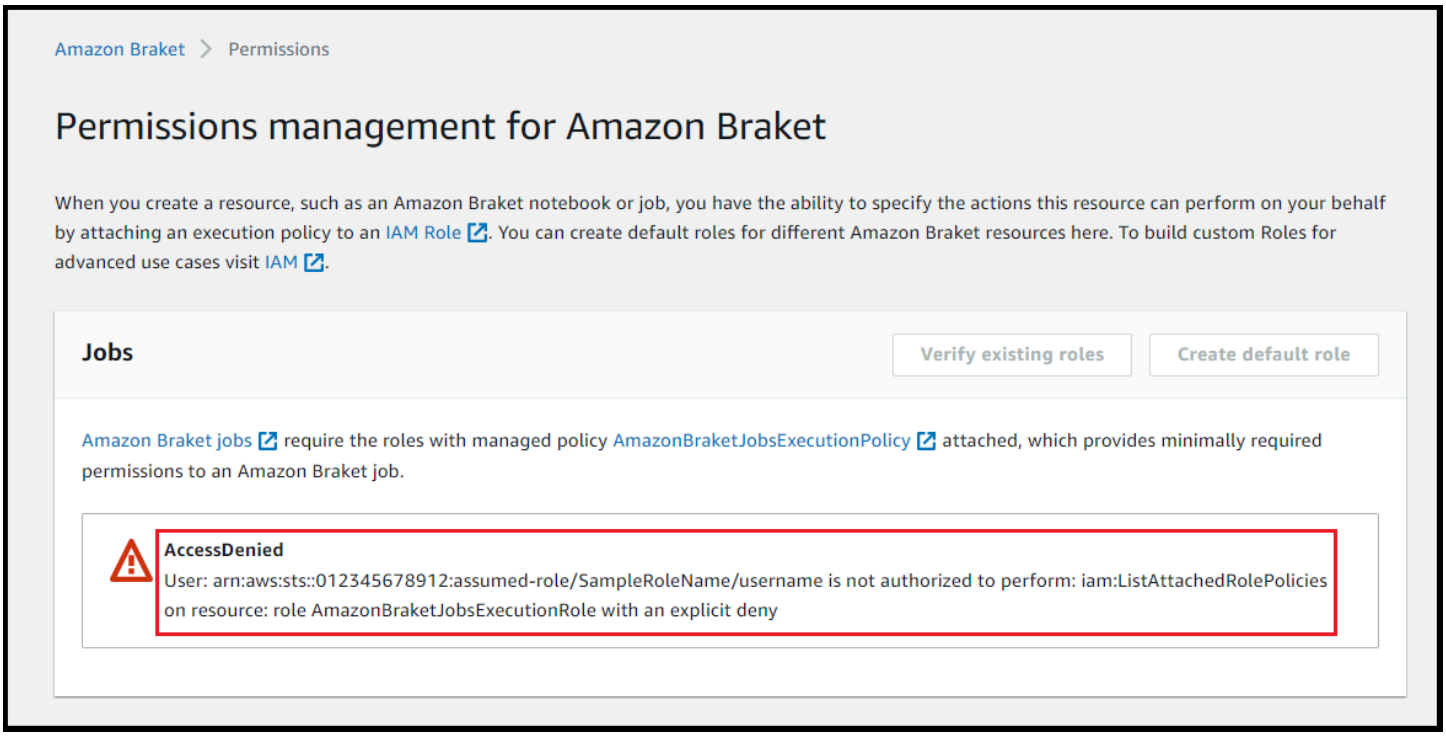
ハイブリッドジョブを実行するのに十分なアクセス許可を持つロールがない場合は、そのようなロールが見つからなかったというメッセージが表示されます。[Create default role] (デフォルトのロールの作成) ボタンを選択して、十分な権限を持つロールを取得します。



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。

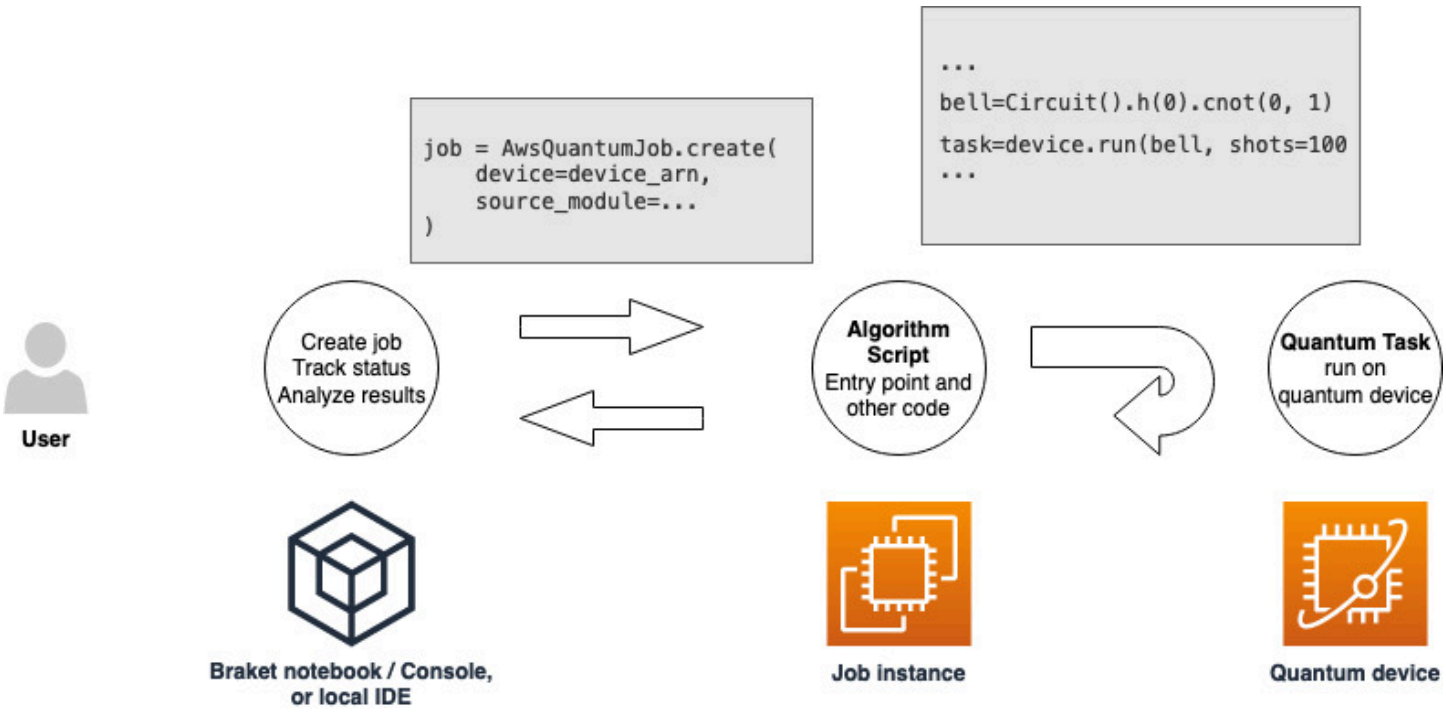


この問い合わせを行う権限がない場合、アクセスは拒否されます。この場合、内部 AWS 管理者に連絡してください。



作成と実行

ハイブリッドジョブを実行するアクセス許可を持つロールを取得したら、続行する準備が整います。最初の Braket ハイブリッドジョブのキー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す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、関心のあるステータスや主要な結果の印刷などの非同期処理を処理します。少なくとも、このスクリプトは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、Amazon Braket Python SDK を使用して最初の回路を実行する」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ユニット (QPU) のタイプにかかわらず、次のスクリプト`device`で で指定したデバイスはハイブリッドジョブのスケジュールに使用され、アルゴリズムスクリプトで環境変数 として使用できます`AMZN_BRAKET_DEVICE_ARN`。

> **Note**
>
> ハイブリッドジョブ AWS リージョン の で使用可能なデバイスのみを使用できます。Amazon Braket SDK がこれを自動的に選択します AWS リージョン。例えば、us-east-1 のハイブリッドジョブでは、IonQ、SV1、DM1、および デバイスを使用できますがTN1、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 はハイブリッドジョブがすべての量子タスクを優先アクセスで実行するように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
)
```

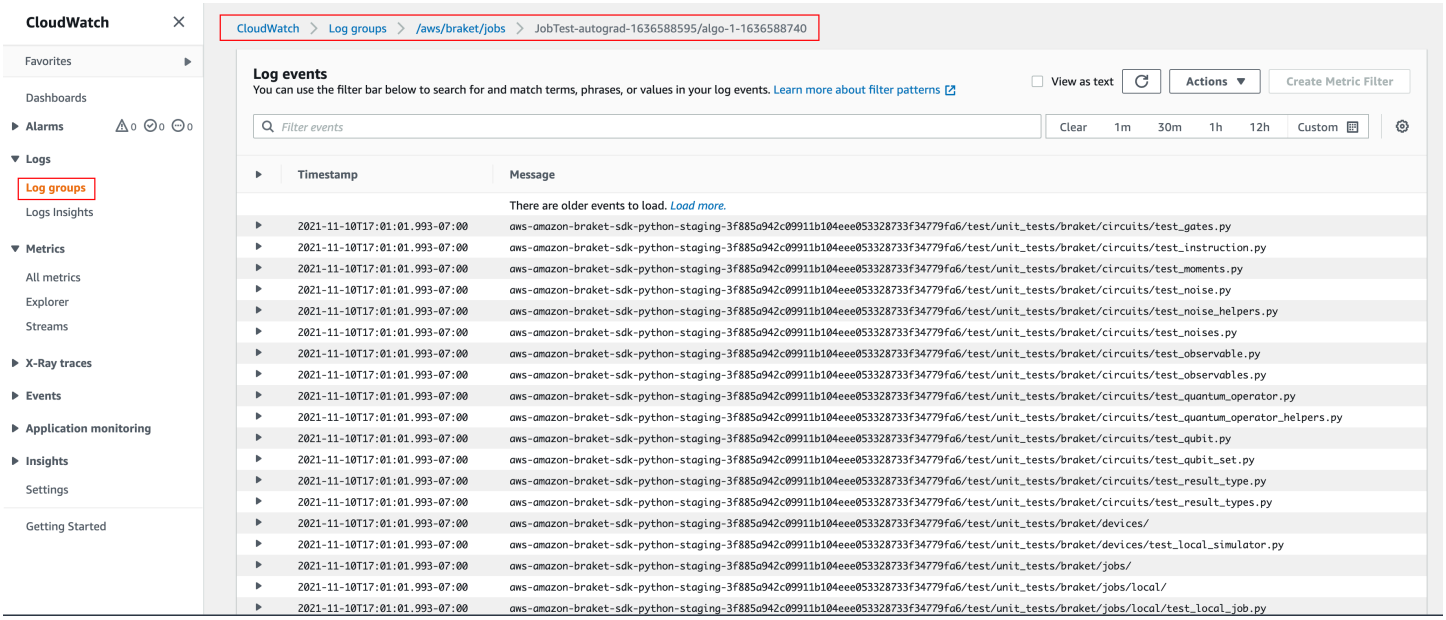
パラメータ wait_until_complete=True は、冗長モードを設定して、ジョブが実行中に実際のジョブからの出力を出力するようにします。次の例のような出力が表示されます。

```
job = AwsQuantumJob.create(
    Devices.Amazon.SV1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    wait_until_complete=True,
)
Initializing Braket Job: arn:aws:braket:us-west-2:<accountid>:job/<UUID>
.....................................
.
.
.

Completed 36.1 KiB/36.1 KiB (692.1 KiB/s) with 1 file(s) remaining#015download:
 s3://braket-external-assets-preview-us-west-2/HybridJobsAccess/models/
braket-2019-09-01.normal.json to ../../braket/additional_lib/original/
braket-2019-09-01.normal.json
Running Code As Process
Test job started!!!!!
Counter({'00': 55, '11': 45})
Counter({'11': 59, '00': 41})
Counter({'00': 55, '11': 45})
Counter({'00': 58, '11': 42})
Counter({'00': 55, '11': 45})
Test job completed!!!!!
Code Run Finished
2021-09-17 21:48:05,544 sagemaker-training-toolkit INFO     Reporting training SUCCESS
```

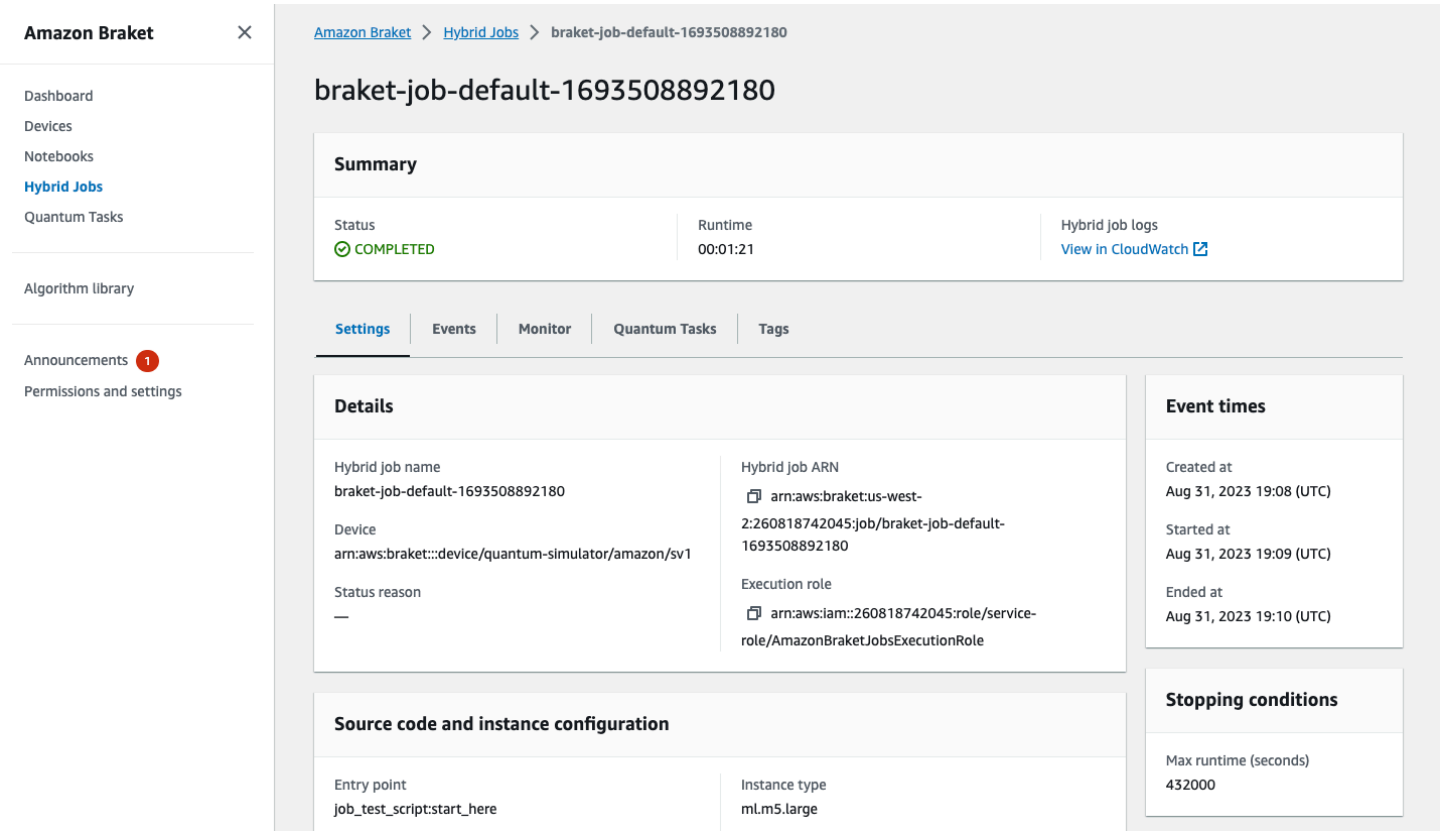
> **Note**
>
> また、AwsQuantumJob.create メソッドでカスタム作成モジュールを使用するには、その場所 (ローカルディレクトリまたはファイルへのパス、または tar.gz ファイルの S3 URI) を渡します。実例については、Amazon Braket サンプル Github リポジトリのハイブリッドジョブフォルダにある Parallelize_training_for_="L.ipynb ファイルを参照してください。

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、ジョブの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 を選択しaws/braket/jobs、ジョブ名を含むログストリームを選択します。上記の例で、これは braket-job-default-1631915042705/algo-1-1631915190 です。



ハイブリッドジョブページを選択し、設定を選択して、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こともできます。



ハイブリッドジョブは、実行中に Amazon S3 でいくつかのアーティファクトを生成します。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code_locationときに別の 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 にある必要があります。

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。また、 というディレクトリもあります。scriptこのディレクトリには、アルゴリズムスクリプトのアーティファクトがsource.tar.gzファイルに含まれています。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。

ジョブ結果の保存

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して、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 圧縮ファイル) から結果を利用できるようにします。

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、ピクルドデータ形式を使用してシリアル化するオプションを渡すことができます。詳細については、braket.jobs.data_persistence モジュール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アルゴリズムスクリプトに追加します。

```
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="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 が出力されjob-arn、ハイブリッドジョブの状態が になるまで 10 秒ごとにハイブリッドジョブの状態が出力されます。その後COMPLETED、ベル回路の結果が表示されます。次の例を参照してください。

```
arn:aws:braket:us-west-2:111122223333:job/braket-job-default-1234567890123
INITIALIZED
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
...
RUNNING
RUNNING
COMPLETED
{'measurement_counts': [{'11': 53, '00': 47},..., {'00': 51, '11': 49}]}
```

チェックポイントを使用したハイブリッドジョブの保存と再起動

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/opt/jobs/checkpointsパスの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 <jobname>-checkpoint-1.json ファイルが作成されます。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 を使用しますfrom braket.jobs import load_job_checkpoint。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。

```
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は、前のハイブリッドジョブjob-arnの を指定し、#ADD でコメントされた行を指定する必要があります。

```
job = AwsQuantumJob.create(
    source_module="source_dir",
    entry_point="source_dir.algorithm_script:start_here",
    device_arn="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
    copy_checkpoints_from_job="<previous-job-ARN>", #ADD
    )
```

ローカルモードでのハイブリッドジョブの構築とデバッグ

新しいハイブリッドアルゴリズムを構築する場合、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ますが、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 Notebook インスタンスまたはラップトップ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) に対して実行してもパフォーマンス上の利点はありません。

ローカルモードを使用するには、プログラム内で発生したLocalQuantumJob場所をAwsQuantumJobに変更します。例えば、最初のハイブリッドジョブを作成するの例を実行するには、次のようにコード内のハイブリッドジョブスクリプトを編集します。

```
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、「Docker の取得」ページを参照してください。さらに、すべてのパラメータがローカルモードでサポートされているわけではありません。

Amazon Braket で量子タスクを実行する

Braket は、さまざまなタイプの量子コンピュータへの安全なオンデマンドアクセスを提供します。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、事前のコミットメントもないため、個々のプロバイダーを通じてアクセスを調達する必要はありません。

- 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 コンソールを使用して量子タスクを送信して実行します。SDK には、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 とノートブックからアクセスできます。量子コンピューティングをプログラムで機能させるアプリケーションを構築APIしている場合は、 を直接呼び出すことができます。

このセクション全体の例は、Amazon Braket Python SDK と Python SDK for Braket (Boto3) を使用して Amazon Braket APIを直接操作する方法を示しています。 AWS

Amazon Braket Python SDK の詳細

Amazon Braket Python SDK を使用するには、最初に AWS Python SDK for Braket (Boto3) をインストールして、 と通信できるようにします AWS API。Amazon Braket Python SDK は、量子のお客様にとって 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 にはノートブックからアクセスできます。ノートブックには、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プリロードされています。
- ノートブックを使用しない場合は、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 をインストールした後、AmazonBraket 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) ノートブックを開きます。
2. 回路に必要な SDK モジュールをインポートします。
3. QPU またはシミュレーターを指定します。
4. 回路をインスタンス化します。
5. 回路を実行します。
6. 結果を収集します。

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 Examples」リポジトリを参照してください。

このセクションの内容:

- QPUs への量子タスクの送信
- 量子タスクはいつ実行されますか？
- Amazon Braket Hybrid Job の管理
- 予約の操作
- エラー緩和手法

QPUs への量子タスクの送信

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。量子タスクは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。

QPU

量子タスクはいつでも QPUs 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。

- IonQ Aria-1 ：`arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1`
- IonQ Aria-2 ：`arn:aws:braket:us-east-1::device/qpu/ionq/Aria-2`

- 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`
- 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**
>
> QPUs およびオンデマンドシミュレーターの CREATED状態の量子タスクをキャンセルできます。オンデマンドシミュレーターと QPUs では、ベストエフォートベースで QUEUED状態の量子タスクをキャンセルできます。QPU 量子タスクは、QPU QUEUED の可用性ウィンドウ中に正常にキャンセルされる可能性は低いことに注意してください。

このセクションの内容:

- IonQ
- IQM
- Rigetti
- QuEra
- 例: QPU への量子タスクの送信
- コンパイルされた回路の検査

IonQ

IonQ は、イオントラップテクノロジーに基づくゲートベースの QPUs を提供します。 IonQ's トラップされたイオン QPUs は、バキュームチェンバー内のマイクロハブ表面電着トラップによって空間的に閉じ込められた、トラップされた 171Yb 以上のイオンのチェーン上に構築されています。

IonQ デバイスは、次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、IonQQPUsは次のネイティブゲートをサポートします。

```
'gpi', 'gpi2', 'ms'
```

ネイティブ MS ゲートを使用するときに 2 つのフェーズパラメータのみを指定すると、完全にエンタングルな MS ゲートが実行されます。完全に交差する MS ゲートは常に π/2 回転を実行します。別の角度を指定し、部分的に絞る 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の詳細については、「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、超伝導トランスモン量子ビットに基づくユニバーサルおよびゲートモデルデバイスです。IQM Garnet デバイスは、正方形の格子トポロジを備えた 20 量子ビットデバイスです。

IQM 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"
```

逐語的なコンパイルでは、IQMデバイスは次のネイティブゲートをサポートします。

```
'cz', 'prx'
```

Rigetti

Rigetti 量子プロセッサは、調整可能な超伝導 に基づくユニバーサルゲートモデルマシンです qubits。

- Ankaa-3 システムは、スケーラブルなマルチチップテクノロジーを利用する 84 量子ビットデバイスです。

Rigetti デバイスは、次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、 は次のネイティブゲートAnkaa-3をサポートします。

```
'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」の角度でのみ実行できます。

パルスレベルの制御は、Rigetti デバイスで使用でき、Ankaa-3システムの次のタイプの事前定義されたフレームのセットをサポートします。

```
`flux_tx`, `charge_tx`, `readout_rx`, `readout_tx`
```

これらのフレームの詳細については、「フレームとポートのロール」を参照してください。

QuEra

QuEra は、アナログハミルトンシミュレーション (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。

量子ビットレジスタのレイアウトと、操作フィールドの時間的および空間的な依存関係を記述することで、アナログハミルトンシミュレーションのパラダイムでこれらのデバイスをプログラムできます。Amazon Braket は、Python SDK の AHS モジュール を通じ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ますbraket.ahs。

詳細については、「Analog Hamiltonian Simulation example notebooks」または「Submit an analog program using QuEra's Aquila」ページを参照してください。

例: QPU への量子タスクの送信

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、量子タスクを Rigettiまたは IonQ デバイスに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は、Rigettiデバイス内の各量子ビットの隣接する量子ビットをconnectivityGraph一覧表示します。

IonQ Aria-1 デバイスを選択する

デバイスの場合IonQ Aria-1、次の例に示すように、 connectivityGraphは空です。これは、デバイスがall-to-all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 Aria-1 device
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-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場合は、 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) を調整するオプションがあります。

```
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 qubits の物理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 (24/7) 送信できます。次のセクションに示すように、 AwsQuantumTaskと一意の量子タスク ID を使用して量子タスクを復元できます。

コンパイルされた回路の検査

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。たとえば、高レベルの量子回路をターゲット 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変換します。量子回路の実際のコンパイル済み出力を検査することは、デバッグと最適化の目的で非常に役立ちます。この知識は、量子アプリケーションのパフォーマンスと効率を向上させる潜在的な問題、ボトルネック、または機会を特定するのに役立ちます。および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IQM、量子回路のコンパイルされた出力を表示Rigetti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デバイスのコンパイルされた回路出力の表示はサポートされていません。

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) とオンデマンドシミュレーターの量子タスクはキューに入れられ、受信した順序で処理されます。量子タスクを送信した後の処理に必要な時間は、他の Amazon Braket のお客様が送信したタスクの数と複雑さ、および選択した QPU の可用性によって異なります。

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 通知を設定する

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で、現在および今後の可用性ウィンドウとデバイスのステータスを確認できます。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が表示されます。

可用性ウィンドウに関係なく、お客様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、目の前にある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 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を指します。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は、 Braket Software Development Kit (SDK)または からアクセスできますAmazon Braket Management Console。

1. タスクキューの深さとは、現在通常の優先度で実行を待っている量子タスクの合計数を指します。
2. 優先度タスクキューの深さとは、 での実行を待機している送信された量子タスクの合計数を指しますAmazon Braket Hybrid Jobs。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とは、デバイスで現在キューに入っているハイブリッドジョブの総数を指します。ハイブリッドジョブの一部としてQuantum tasks送信される には優先度があり、 に集約されますPriority Task Queue。

を使用してキューの深さを表示したいお客様は、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Braket SDKできます。

```
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-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キューの位置を可視化します。

キューの位置

Queue position は、それぞれのデバイスキュー内の量子タスクまたはハイブリッドジョブの現在の位置を指します。量子タスクまたはハイブリッドジョブの場合は、 Braket Software Development Kit (SDK)または を使用して取得できますAmazon Braket Management Console。

を使用してキューの位置を表示したいお客様は、次のコードス二ペットを変更して、量子タスクまたはハイブリッドジョブのキューの位置を取得Braket SDK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 通知を設定する

Amazon Braket は、QPU の可用性が変化したとき、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。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 メッセージで受信するには、次の手順に従います。

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」および「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 による Amazon Braket のモニタリング EventBridge」を参照してください。

(オプション) SNS 通知を設定する

Amazon Simple Notification Service (SNS) を使用して通知を設定し、Amazon Braket 量子タスクが完了すると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外で量子タスクを送信する場合など、長い待機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まで待機しない場合は、SNS 通知を設定できます。

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 の Amazon Braket の例、特に通知を設定するためのノートブックの例を参照してください。

Amazon Braket Hybrid Job の管理

このセクションでは、Amazon Braket でハイブリッドジョブを管理する方法について説明します。

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:

- スクリプトを実行するようにハイブリッドジョブインスタンスを設定する
- ハイブリッドジョブをキャンセルする方法
- パラメトリックコンパイルを使用してハイブリッドジョブを高速化する
- Amazon Braket で PennyLane を使う
- 独自のコンテナ
- Amazon Braket での CUDA-Q の使用
- を使用してハイブリッドジョブを直接操作する API

スクリプトを実行するようにハイブリッドジョブインスタンスを設定する

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設定引数を使用してハイブリッドジョブを作成するときに、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.p3.8xlarge"), # Use NVIDIA Tesla
 V100 instance with 4 GPUs.
    ...
    ),
```

埋め込みシミュレーションを実行していて、デバイス設定でローカルデバイスを指定している場合は、 を指定`instanceCount`して 1 つ以上のインスタンスを追加`InstanceConfig`でリクエストできます。上限は 5 です。たとえば、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.p3.8xlarge", instanceCount=3), #
 Use 3 NVIDIA Tesla V100
    ...
    ),
```

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。この QML の並列化トレーニングの例を表示する方法の詳細については、次のノートブックの例を参照してください。

次の 3 つの表に、標準インスタンス、高性能インスタンス、GPU アクセラレーションインスタンスで使用できるインスタンスタイプと仕様を示します。

> **Note**
>
> ハイブリッドジョブのデフォルトの従来の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

高性能インスタンス	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

高性能インスタンス	vCPU	メモリ (GiB)
ml.c5n.xlarge	4	10.5
ml.c5n.2xlarge	8	21
ml.c5d.4xlarge	16	32
ml.c5d.9xlarge	36	72
ml.c5d.18xlarge	72	192

GPU アクセラレーションインスタンス	GPU	vCPU	メモリ (GiB)	GPU メモリ (GiB)
ml.p3.2xlarge	1	8	61	16
ml.p3.8xlarge	4	32	244	64
ml.p3.16xlarge	8	64	488	128
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

> **Note**
>
> p3 インスタンスは us-west-1 では使用できません。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キャパシティ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ます。

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.p3.8xlarge",
        volumeSizeInGb=50,
    ),
    ...
    ),
```

AwsSession でデフォルトのバケットを設定します。

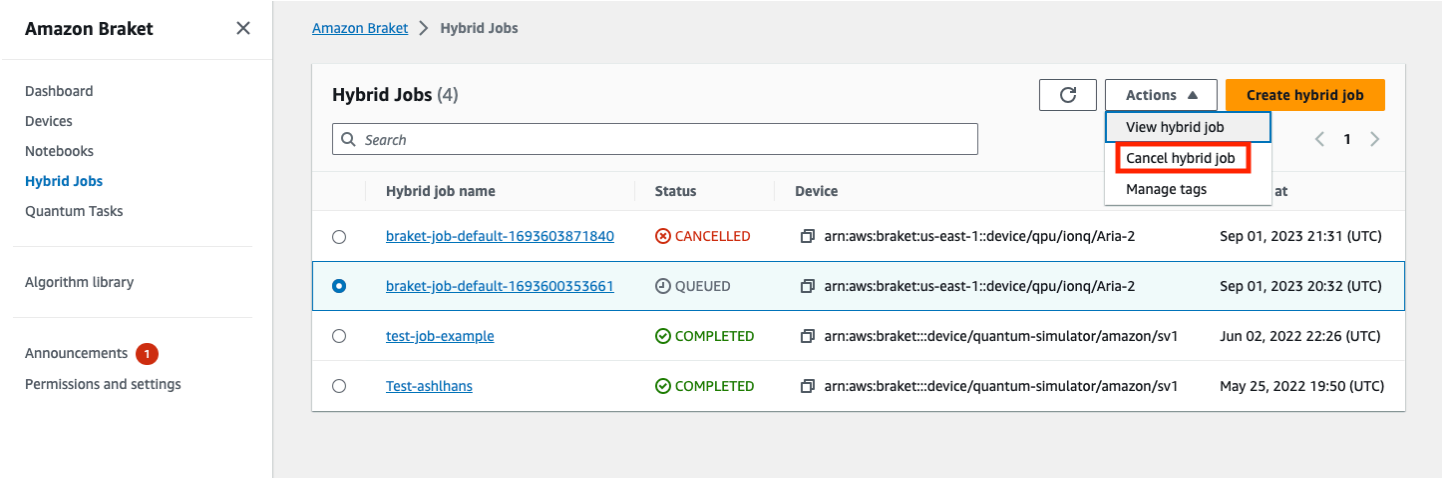
独自のAwsSessionインスタンスを使用すると、デフォルトの Amazon S3 バケットのカスタムロケーションを指定する機能など、柔軟性が向上します。デフォルトでは、 の Amazon S3 バケットの場所AwsSessionは事前設定されています"amazon-braket-{id}-{region}"。ただし、の作成時にデフォルトの Amazon S3 バケットの場所を上書きすることもできますAwsSession。ユーザーは、次のコード例に示すように aws_sessionパラメータを指定することで、オプションで AwsSession オブジェクトを AwsQuantumJob.create()メソッドに渡すことができます。

```
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
)
```

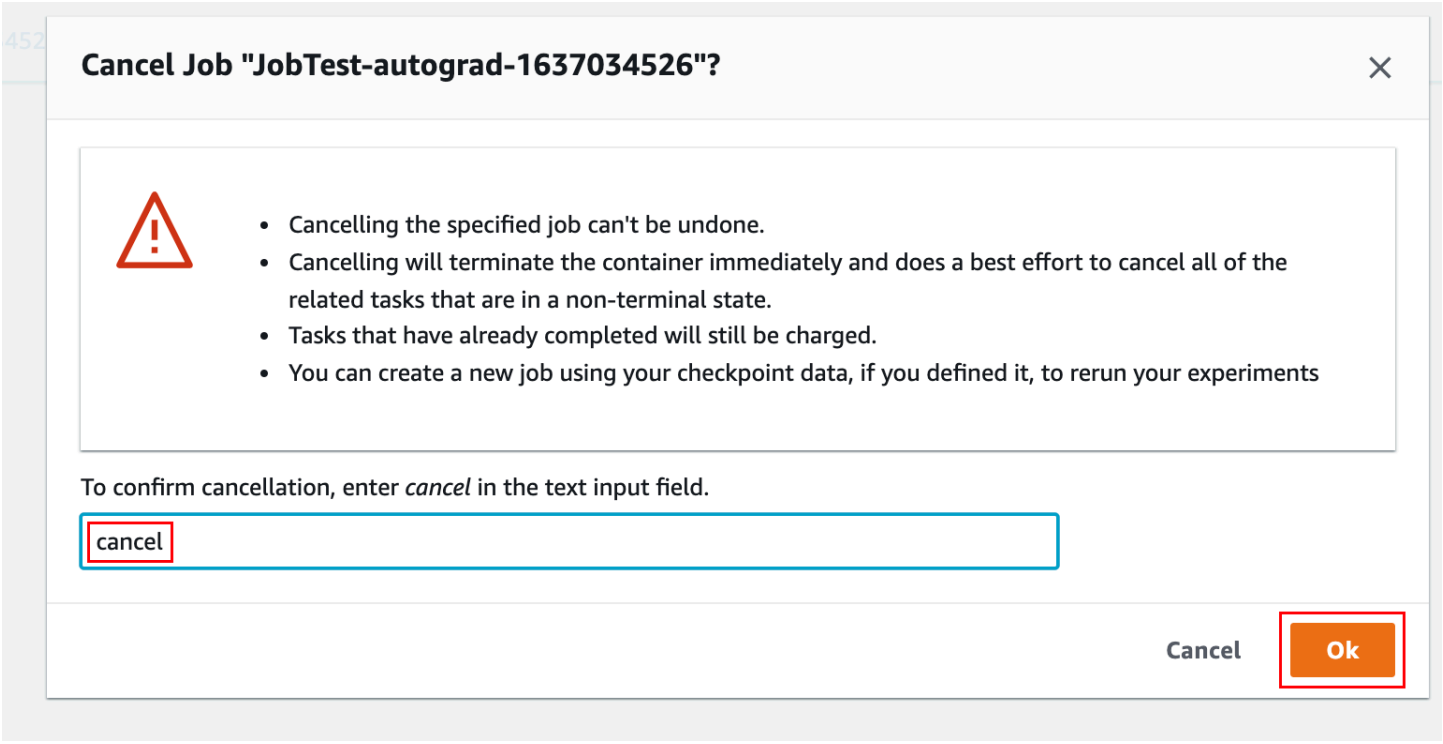
ハイブリッドジョブをキャンセルする方法

非ターミナル状態のハイブリッドジョブをキャンセルする必要がある場合があります。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。



キャンセルを確認するには、プロンプトが表示されたら、と入力フィールドにキャンセルと入力し、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識別し、次のコードに示すようにそのジョブで 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、従来のハイブリッドジョブコンテナを直ちに終了し、非終了状態のままの関連する量子タスクをすべてキャンセルするよう最善を尽くします。

パラメトリックコンパイルを使用してハイブリッドジョブを高速化する

Amazon Braket は、特定の QPUs でのパラメトリックコンパイルをサポートしています。これにより、ハイブリッドアルゴリズムの反復ごとにではなく、回路を 1 回だけコンパイルすることで、計算コストの高いコンパイルステップに関連す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ランタイムが大幅に向上します。Braket Hybrid Job として、サポートされている QPUs のいずれかにパラメータ化された回路を送信するだけです。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、最高品質の結果を確保します。

パラメトリック回路を作成するには、まずアルゴリズムスクリプトの入力としてパラメータを指定する必要があります。この例では、小さなパラメータ回路を使用し、各反復間の従来の処理を無視します。一般的なワークロードでは、多数の回路をバッチで送信し、各反復でパラメータを更新するなどの従来の処理を実行します。

```
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、回路は初回実行時にのみコンパイルされます。次の実行では、コンパイルされた回路が再利用され、追加のコード行なしでハイブリッドジョブのランタイムパフォーマンスが向上します。

```
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、パルスレベルプログラムRigetti Computingを除き、 からのすべての超伝導ゲートベースの QPUs でサポートされています。

Amazon Braket で PennyLane を使う

ハイブリッドアルゴリズムは、古典的指示と量子指示の両方を含むアルゴリズムです。クラシック命令はクラシックハードウェア (EC2 インスタンスまたはラップトップ) 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で実行されます。Hybrid Jobs 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 Jobs を使用するタイミング」を参照してください。

Amazon Braket では、Amazon Amazon Braket PennyLane プラグイン、または Amazon Amazon Braket Python SDK とサンプルノートブックリポジトリを使用して、ハイブリッド量子アルゴリズムをセットアップして実行できます。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。ただし、PennyLane は豊富なエクスペリエンスを提供するため、お勧めします。

ハイブリッド量子アルゴリズムについて

ハイブリッド量子アルゴリズムは、現代の量子コンピューティングデバイスが一般的にノイズを生成するため、今日の業界にとって重要です。計算に追加された量子ゲートはすべてノイズを追加する可能性を高めます。したがって、長時間実行されるアルゴリズムはノイズに圧倒され、計算が失敗する可能性があります。

Shor's （量子フェーズ推定の例) や Grover's (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、量子処理装置 (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の手の届く範囲内で、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 での 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 の結合勾配
- Hybrid Jobs と PennyLane を使用して QAOA アルゴリズムを実行する
- PennyLane 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

PennyLane での 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を使用して、量子化学、量子機械学習、最適化における計算問題の解決策を見つけるために二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できます。

PennyLane ライブラリには、PyTorch や TensorFlow などの使い慣れた機械学習ツールへのインターフェイスが用意されており、量子回路のトレーニングを迅速かつ直感的に行うことができます。

- PennyLane ライブラリ -- PennyLane は Amazon Braket ノートブックにプリ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、選択した 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、AmazonBraket 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 に接続するため、PennyLane で Amazon Braket デバイスで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
```

次の例は、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 に関するチュートリアルの例と詳細については、Amazon Braket サンプル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 の埋め込み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 と連携する 2 つの Amazon Braket 量子デバイスを提供します。

- `braket.aws.qubit` QPUs やシミュレーターなど、AmazonBraket サービスの量子デバイスで実行するための
- `braket.local.qubit` Amazon Braket SDK のローカルシミュレーターで実行するための

Amazon Braket PennyLane プラグインはオープンソースです。PennyLane Plugin GitHub リポジトリからインストールできます。

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、AmazonBraket を介して量子コンピューティングハードウェアデバイスとやり取り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ように設計されたオープンソースライブラリです。

Amazon Braket は、サンプル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 には、PennyLane の高性能 CPU および GPU ベースの組み込みシミュレーターが付属するようになりました。この組み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、高速ステートベクトル`lightning.qubit`シミュレーター、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ます。これらの組み込みシミュレーターは、結合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 つ以上の CPU または GPU インスタンスで実行できます。

Hybrid Jobs では、従来のコプロセッサと QPU の組み合わせ、 などの Amazon Braket オンデマンドシミュレーター、または PennyLane の埋め込みシミュレーターを直接使用してSV1、バリエーションアルゴリズムコードを実行できるようになりました。

組み込みシミュレーターは Hybrid Jobs コンテナで既に使用できます。メインの Python 関数をデコレータで`@hybrid_job`デコレートするだけで済みます。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InstanceConfigに示すように、 で 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.p3.8xlarge"))
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、サンプル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 の結合勾配

Amazon Braket 用PennyLaneプラグインを使用すると、ローカルステートベクトルシミュレーターまたは SV1 で を実行するときに、ジョイントの区別方法を使用して勾配を計算できます。

注： アドジョイントの区別方法を使用するには、 ではなく qnodediff_method='device'で を指定する必要がありますdiff_method='adjoint'。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**
>
> 現在、 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。から QAOA を実行するときに SV1 の結合差別化機能を使用する場合はPennyLane、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再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 を使用して QAOA アルゴリズムを実行する

このセクションでは、学習した内容を使用して、PennyLane とパラメトリックコンパイルを使用して実際のハイブリッドプログラムを記述します。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。プログラムは、従来の Max Cut 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、単純な勾配降下法を使用してパラメータを最適化して、コスト関数を最小化します。この例では、わかりやすくする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、入力データ設定の専用チャネルを介して問題仕様を提供することがベストプラクティスです。フラグの`parametrize_differentiable`デフォルトは `True`であるため、サポートされている QPUs。

```
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、パルスレベルプログラムRigetti Computingを除き、 からのすべての超伝導ゲートベースの QPUs でサポートされています。

PennyLane 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

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説明します。Pennylane の GPU ベースの埋め込みシミュレーターは`lightning.gpu`、Nvidia cuQuantum ライブラリを使用して回路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**`lightning.gpu`**での の使用

このノートブックの量子近似最適化アルゴリズム (QAOA) の例を考えてみましょう。埋め込みシミュレーターを選択するには、 という形式の文字列に引`device`数を指定します`"local:<provider>/<simulator_name>"`。たとえば、 `"local:pennylane/lightning.gpu"`に を設定します`lightning.gpu`。起動時にハイブリッドジョブに渡すデバイス文字列は、環境変数 としてジョブに渡されます`"AMZN_BRAKET_DEVICE_ARN"`。

```
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 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。さまざまな勾配を計算するには、シミュレーターにいくつかのカスタムゲート分解を提供する必要があります。

これで、ハイブリッドジョブ起動スクリプトを準備する準備ができました。QAOA アルゴリズムは、 `m5.2xlarge`と の 2 つのインスタンスタイプを使用して実行します`p3.2xlarge`。`m5.2xlarge` インスタンスタイプは、標準のデベロッパーラップトップと同等です。`p3.2xlarge` は、16GB のメモリを持つ単一の NVIDIA Volta GPU を持つ高速コンピューティングインスタンスです。

すべてのハイブリッドジョブ`hyperparameters`の は同じになります。さまざまなインスタンスとシミュレーターを試すために必要なことは、次のように 2 行変更することです。

```
# Specify device that the hybrid job will primarily be targeting
device = "local:pennylane/lightning.qubit"
# Run on a CPU based instance with about as much power as a laptop
instance_config = InstanceConfig(instanceType='ml.m5.2xlarge')
```

または

```
# Specify device that the hybrid job will primarily be targeting
device = "local:pennylane/lightning.gpu"
# Run on an inexpensive GPU based instance
instance_config = InstanceConfig(instanceType='ml.p3.2xlarge')
```

> **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、必ず組み込み GPU ベースのシミュレーターを使用してください。

まず、2 つのハイブリッドジョブを作成し、18 個の頂点を持つグラフで QAOA を使用して Max-Cut を解決できます。これは 18 量子ビット回路に変換されます。比較的小さく、ラップトップや`m5.2xlarge`インスタンスですばやく実行できます。

```
num_nodes = 18
num_edges = 24
seed = 1967

graph = nx.gnm_random_graph(num_nodes, num_edges, seed=seed)

# And similarly for the p3 job
m5_job = AwsQuantumJob.create(
    device=device,
    source_module="qaoa_source",
    job_name="qaoa-m5-" + str(int(time.time())),
    image_uri=image_uri,
    # Relative to the source_module
    entry_point="qaoa_source.qaoa_algorithm_script",
    copy_checkpoints_from_job=None,
    instance_config=instance_config,
```

```
    # general parameters
    hyperparameters=hyperparameters,
    input_data={"input-graph": input_file_path},
    wait_until_complete=True,
)
```

m5.2xlarge インスタンスの平均反復時間は約 25 秒で、p3.2xlargeインスタンスの平均反復時間は約 12 秒です。この 18 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 2 倍の高速化が提供されます。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、p3.2xlargeインスタンスのコストは 0.06375 USD であることがわかります。ここで行ったように、合計 5 回の反復を実行すると、CPU インスタンスでは 0.016 USD、GPU インスタンスでは 0.06375 USD のコストがかかります。どちらもかなり安価です。

次に、問題をより難しくし、24 量子ビットに変換される 24 頂点グラフで Max-Cut の問題を解決してみましょう。同じ 2 つのインスタンスでハイブリッドジョブを再度実行し、コストを比較します。

> **Note**
>
> CPU インスタンスでこのハイブリッドジョブを実行するのに約 5 時間かかる場合があります。

```
num_nodes = 24
num_edges = 36
seed = 1967

graph = nx.gnm_random_graph(num_nodes, num_edges, seed=seed)

# And similarly for the p3 job
m5_big_job = AwsQuantumJob.create(
    device=device,
    source_module="qaoa_source",
    job_name="qaoa-m5-big-" + str(int(time.time())),
    image_uri=image_uri,
    # Relative to the source_module
    entry_point="qaoa_source.qaoa_algorithm_script",
    copy_checkpoints_from_job=None,
    instance_config=instance_config,
```

```
    # general parameters
    hyperparameters=hyperparameters,
    input_data={"input-graph": input_file_path},
    wait_until_complete=True,
)
```

m5.2xlarge インスタンスの平均反復時間は約 1 時間ですが、p3.2xlargeインスタンスの平均反復時間は約 2 分です。このより大きな問題では、GPU インスタンスは 1 桁速くなります。この高速化のメリットを得るために必要なのは、2 行のコードを変更し、インスタンスタイプと使用するローカルシミュレーターを交換することだけでした。ここで行ったように、合計 5 回の反復を実行するには、CPU インスタンスを使用して約 2.27072 USD、または GPU インスタンスを使用して約 0.775625 USD のコストがかかります。CPU 使用率は高価になるだけでなく、実行に時間がかかります。NVIDIA CuQuantum にバックアップされた PennyLane の埋め込みシミュレーターを使用して AWS、 で利用可能な GPU インスタンスを使用してこのワークフローを加速することで、中間量子ビット数 (20 から 30 の間) のワークフローを実行して、総コストと時間を削減できます。つまり、大きすぎてラップトップや同様のサイズのインスタンスですぐに実行できない問題でも、量子コンピューティングを試すことができます。

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てワークロードをさらに高速化できます。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、モデルのパラメータが更新され、損失関数が最小限に抑えられます。損失関数は通常、単一のデータポイントと、データセット全体の平均損失の合計に対して定義されます。QML では、勾配計算の合計損失を平均する前に、損失は通常シリアルで計算されます。この手順は、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、損失は並行して評価できます。異なるデータポイントに関連付けられた損失と勾配を同時に評価できます。これはデータ並列処理と呼ばれます。SageMaker の分散データ並列ライブラリを使用すると、Amazon Braket Hybrid Jobs はデータ並列処理を活用してトレーニングを高速化することを容易にします。

二項分類の例として、よく知られている UCI リポジトリの Sonar データセットデータセットを使用するデータ並列処理については、次の QML ワークロードを検討してください。Sonar データセットには 208 個のデータポイントがあり、それぞれに 60 個の特徴があり、ソナーシグナルから収集され、マテリアルから跳ね返ります。各データポイントは、地雷の場合は「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ます。入出力レイヤーは、PyTorch に実装されたクラシック

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、サンプル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 などの組み込み GPU ベースのシミュレーターを使用して GPU の能力を活用してlightning.gpu、組み込み 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、 AwsQuantumJob.createを呼び出し、キーワード引数を使用して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指定できます。

```
instance_config = InstanceConfig(instanceType='ml.p3.2xlarge')

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、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して、トレーニングを正しく並列化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。パッケージは、ワークロードを複数の GPUs と複数のインスタンスに分散するために、ほとんどの負荷が高くなります。このパッケージは Braket PyTorch コンテナと TensorFlow コンテナで事前設定されています。このdistモジュールは、トレーニング (world_size) の GPUs の合計数と GPU コアlocal_rankの rankと をアルゴリズムスクリプトに指示します。 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あり、 local_rank はインスタンス内の GPU のインデックスです。たとえば、トレーニングにそれぞれ 8 つの GPUs が割り当てられる 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 DistributedSamplerに従って を定義しrank、それをデータローダーに渡します。このサンプラーは、GPUsデータセットの同じスライスにアクセスするのを回避します。

```
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、多くの場合、結果を保存し、トレーニングの進行状況を出力します。各 GPU が 保存コマンドと印刷コマンドを実行すると、ログは繰り返される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、0 の GPU rank からのみ保存および出力できます。

```
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.p3.16xlargeインスタンスタイプをサポートしています。ハイブリッドジョブの 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、 "sagemaker_distributed_dataparallel_enabled"を に設定"true"し、 を使用中のインスタンスタイプ"sagemaker_instance_type"に設定して、2 つのハイパーパラメータを追加する必要があります。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。アルゴリズムスクリプトは明示的に使用する必要はありません。Amazon Braket SDK では、便利なキーワード引数 を提供しますdistribution。ハイブリッドジョブの作成distribution="data_parallel"時に、Amazon Braket SDK は 2 つのハイパーパラメータを自動的に挿入します。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含める必要があります。

インスタンスとデータ並列処理を設定して、ハイブリッドジョブを送信できるようになりました。ml.p3.16xlarge インスタンスには 8 GPUs があります。を設定するとinstanceCount=1、ワークロードはインスタンス内の 8 GPUs に分散されます。instanceCount 複数の を設定すると、ワークロードはすべてのインスタンスで使用できる GPUs に分散されます。複数のインスタンスを使用する場合、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。たとえば、4 つのインスタンスを使用する場合、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.p3.16xlarge',
                                 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では、 `train_dp.py`はデータ並列処理を使用するための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する場合にのみ正しく機能することに注意してください。正しく変更されたアルゴリズムスクリプトなしでデータ並列処理オプションが有効になっている場合、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。これは非効率です。

上記の二項分類問題の 26 量子ビット量子回路でモデルをトレーニングする場合の例で、実行時間とコストを比較してみましょう。この例で使用されている`ml.p3.16xlarge`インスタンスの料金は 1 分あたり 0.4692 USD です。データ並列処理がないと、シミュレーターがモデルを 1 エポック (つまり、208 データポイント以上) でトレーニングするのに約 45 分かかり、コストは約 20 USD になります。1 つのインスタンスと 4 つのインスタンスのデータ並列処理では、それぞれ 6 分と 1.5 分しかかからず、どちらも約 2.8 USD になります。4 つのインスタンスでデータ並列処理を使用することで、実行時間を 30 倍向上させるだけでなく、コストを 1 桁削減できます。

独自のコンテナ

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いずれか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提供するだけで済みます。欠落している軽微な依存関係は、アルゴリズムスクリプトまたは を使用する`requirements.txt`ファイルから追加できます`pip`。

これらのコンテナのいずれもユースケースをサポートしていない場合、またはコンテナを拡張する場合、Braket Hybrid Jobs は独自のカスタムDockerコンテナイメージを使用したハイブリッドジョブの実行、または独自のコンテナの持ち込み (BYOC) をサポートします。ただし、先に進む前に、実際に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:

- 自分のコンテナを持ち込むのはどのような場合が適切ですか？
- 独自のコンテナを持ち込むためのレシピ
- 独自のコンテナで Braket ハイブリッドジョブを実行する

自分のコンテナを持ち込むのはどのような場合が適切ですか？

Braket Hybrid Jobs に独自のコンテナ (BYOC) を導入すると、パッケージ化された環境にインストールすることで、独自のソフトウェアを柔軟に使用できます。特定のニーズに応じて、完全なBYOC Dockerビルド - Amazon ECR アップロード - カスタムイメージ URI サイクルを経ることなく、同じ柔軟性を実現する方法がある場合があります。

> **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 が適していない可能性があります。例えば、PyPi を使用している場合です。

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。ファイルは自動的に読み込まれ、指定されたバージョンで通常どおりパッケージ`pip`が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、ジョブのランタイムが大幅に増加する可能性があります。Python を確認し、該当する場合は、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 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、または Braket の事前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に必要です。また、次の場合にも適しています。

- ライセンスキーでソフトウェアを使用しているため、ソフトウェアを実行するにはライセンスサーバーに対してそのキーを認証する必要があります。BYOC を使用すると、ライセンスキーをDockerイメージに埋め込み、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している。たとえば、ソフトウェアは、特定の SSH キーが必要なプライベート GitLab または GitHub リポジトリでホストされます。
- 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ります。BYOC を使用すると、ソフトウェアのインストールによるハイブリッドジョブコンテナの長い起動時間を排除できます。

BYOC では、ソフトウェアでDockerコンテナを構築し、ユーザーが利用できるようにすることで、カスタム SDK またはアルゴリズムを顧客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に従う必要があります。

独自のコンテナを持ち込むためのレシピ

このセクションでは、ハイブリッドジョブを Braket bring your own container (BYOC)するために必要なもの、つまりカスタムDockerイメージを起動して実行するためにそれらを組み合わせるスクリプト、ファイル、手順のstep-by-stepステップガイドを提供します。2 つの一般的なケースのレシピを提供しています。

1. Docker イメージに追加のソフトウェアをインストールし、ジョブで Python アルゴリズムスクリプトのみを使用します。
2. Hybrid Jobs で Python 以外の言語で記述されたアルゴリズムスクリプト、または x86 以外の CPU アーキテクチャを使用します。

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 がハイブリッドジョブを実行すると、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、イメージ URI 入力でDocker指定されたイメージを実行してジョブを作成します。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、そのカスタムイメージを使用してジョブを実行します。

ハイブリッドジョブで使用できるDockerイメージを構築するために必要な特定のコンポーネント。の記述と構築に慣れていない場合はDockerfiles、これらの手順を読みながら、必要に応じてDockerfile ドキュメントと Amazon ECR CLIドキュメントを参照することをお勧めします。

必要なものの概要を以下に示します。

- 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
- で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

Dockerfile のベースイメージ

Python を使用していて、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、ベースイメージのオプションは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 つです。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。たとえば、BYOC Docker ファイルの最初の行は次のようになります。 `FROM [IMAGE_URI_HERE]`

次に、 の残りの部分に記入Dockerfileして、コンテナに追加するソフトウェアをインストールしてセットアップします。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、これを含める必要はありません。

C++、Rust、Julia などの Python 以外の言語を使用する場合、または ARM などの 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ビルドする必要がある場合があります。このようなイメージの多くは、Amazon Elastic Container Registry Public Gallery にあります。CPU アーキテクチャに適したものを選択し、必要に応じて使用する GPU を選択します。

(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 イメージにのみソフトウェアを追加する場合は、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 スクリプトを変更する必要があります。例えば、`braket_container.py`Amazon Braket Github の Python スクリプトです。これは、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するスクリプトです。コンテナエントリポイントスクリプト自体は Python にある必要がありますが、Python 以外のスクリプトを起動できます。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を確認できます。このロジックを変更することで、エントリポイントスクリプトを有効にして Python 以外のアルゴリズムスクリプトを起動できます。たとえば、ファイル拡張子の終了に応じて 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 を記述することもできますbraket_container.py。入力データ、ソースアーカイブ、および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。

で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**

> **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している場合、コンテナスクリプトは既に存在します。

前のステップで変更され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、環境変数を SAGEMAKER_PROGRAMに定義するかbraket_container.py、新しいコンテナエントリポイントスクリプトに名前を付けました。

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Dockerfileできるようにする 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。例えば、 によって管理されるインストール済みコードを適切に帰属させることですMIT license。

プライベート GitHub または GitLab リポジトリでホストされているコードなど、非パブリックコードを含める必要がある場合は、Dockerイメージに SSH キーを埋め込んでアクセスしないでください。代わりに、ビルドDocker Compose時に を使用して、ビルドされているホストマシン上の SSH Dockerへのアクセスを に許可します。詳細については、Docker で SSH キーを安全に使用してプライベート Github リポジトリにアクセスするガイドを参照してください。

Dockerイメージの構築とアップロード

適切に定義された ではDockerfile、プライベート Amazon ECR リポジトリがまだ存在しない場合、そのリポジトリを作成するステップに従う準備が整いました。コンテナイメージを構築、タグ付け、リポジトリにアップロードすることもできます。

イメージを構築、タグ付け、プッシュする準備が整いました。のオプションの完全な説明docker buildといくつかの例については、Docker ビルドドキュメントを参照してください。

上記のサンプルファイルでは、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 アクセス許可の割り当て

Braket Hybrid Jobs Docker イメージはプライベート Amazon ECR リポジトリでホストする必要があります。デフォルトでは、プライベート Amazon ECR リポジトリは、 または共同作業者Braket Hybrid Jobs IAM roleや学生など、イメージを使用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には、リポジトリポリシーを設定する必要があります。一般的に、イメージにアクセスする特定のユーザーとIAMロールにのみアクセス許可を付与し、 を持つすべてのユーザーにイメージimage URIのプルを許可しないでください。

独自のコンテナで Braket ハイブリッドジョブを実行する

独自のコンテナを使用してハイブリッドジョブを作成するには、image_uri指定された 引数AwsQuantumJob.create()で を呼び出します。QPU、オンデマンドシミュレーターを使用する

か、Braket Hybrid Jobs で使用できるクラシックプロセッサでコードをローカルで実行できます。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。

クラシックプロセッサでコードを実行するには、 を更新して instanceCount instanceTypeとを指定しますInstanceConfig。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する必要があります。選択できるインスタンスの数の上限は 5 です。例:

```
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.p3.8xlarge", 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。許容値は の形式に従う必要がありますdevice = "local:<provider>/<simulator_name>"。<provider> および は、文字、数字、_、-、および . のみで構成<simulator_name>する必要があります。文字列は 256 文字に制限されています。
> BYOC を使用する予定で、Braket SDK を使用して量子タスクを作成していない場合は、環境変数の値をCreateQuantumTaskリクエストの jobTokenパラメータAMZN_BRAKET_JOB_TOKEN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。

Amazon Braket での CUDA-Q の使用

NVIDIA's CUDA-Q は、CPUs、GPUs、量子処理ユニット (QPUs。統合されたプログラミングモデルを提供するため、デベロッパーは従来の命令と量子命令の両方を 1 つのプログラムで表現できるため、ワークフローが合理化されます。 は、組み込みの CPU および GPU シミュレーターを使用して量子プログラムのシミュレーションとランタイムCUDA-Qを加速します。

Amazon Braket Hybrid Jobs CUDA-Qで を使用すると、柔軟でオンデマンドの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、使用した分だけ料金が発生します。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、完全な実験の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できます。

Amazon Braket Hybrid Jobs はGPUs をサポートしています。 CUDA-QGPUs CPU ベースのシミュレーターと比較して量子プログラムシミュレーションを大幅に高速化します。特に、高量子ビット数回路を使用する場合に便利です。Amazon Braket Hybrid Jobs CUDA-Qで を使用すると、並列化が簡単になります。Hybrid Jobs は、回路サンプリングと観測可能な評価の複数の計算ノードへの分散を簡素化します。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ためのインフラストラクチャを設定するのではなく、ワークロードの開発に集中できます。

開始するには、Amazon Braket CUDA-Q の例 Github のスターターの例を参照して、独自のコンテナの持ち込み (BYOC) CUDA-Qを通じて がサポートするジョブコンテナを作成します。CUDA-Q コンテナを構築して Amazon ECR リポジトリに公開するための適切な IAM アクセス許可があることを確認してください。

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例です。

```
image_uri = "<ecr-image-uri>"

@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ます。この例では、ラップトップまたは Braket Jupyter ノートブックでローカルに実行されます。local=True 設定のため、このスクリプトを実行すると、ローカル環境でコンテナが開始され、テストとデバッグのために 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フラグを削除してジョブを実行できます AWS。詳細については、Amazon Braket Hybrid Jobs の開始方法」を参照してください。

ワークロードに高い量子ビット数、多数の回路、または多数の反復がある場合は、instance_config設定を指定することで、より強力な 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デコレータで instance_config 設定を構成する方法を示しています。サポートされてい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を設定してスクリプトを実行する」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-QGPU シミュレーターでワークロードを実行できます。GPU シミュレーターを有効にするには、バックエンド名 を使用しますnvidia。nvidia バックエンドは CUDA-Q GPU シミュレーターとして動作します。次に、NVIDIAGPU 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 設定の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.p3.2xlarge"),
)
def my_job_script():
    ...
```

Amazon Braket Hybrid Jobs は、 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 CUDA-Q。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して、ワークロードのパフォーマンスを向上させることができます。複数のオブザーバビリティを並列化するには、アルゴリズムスクリプトに次の変更を加えます。

nvidia バックエンドの mgpuオプションを設定します。これは、オブザーバビリティを並列化するために必要です。並列化は GPUs 間の通信に MPI を使用するため、MPI は実行前に初期化し、実行後に確定する必要があります。

次に、 を設定して実行モードを指定しますexecution=cudaq.parallel.mpi。次のコードスニペットは、これらの変更を示しています。

```
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()
```

デhybrid_jobコレータで、次のコードスニペットに示すようにGPUs 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.p3.8xlarge", instanceCount=1),
    image_uri=image_uri,
)
def parallel_observables_gpu_job(sagemaker_mpi_enabled=True):
    ...
```

Amazon Braket サンプル Github の並列シミュレーションノートブックは、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し、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すend-to-endの例を提供します。

量子コンピュータでのワークロードの実行

シミュレーターのテストが完了したら、QPUs での実験の実行に移行できます。ターゲットを 、、IonQまたは Rigetti デバイスなどの Amazon Braket QPU IQMに切り替えるだけです。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 QPUsAmazon Braket コンソール」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)
```

Amazon Braket Hybrid Jobs 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」を参照してください。CUDA-Q の詳細については、CUDA-Q ドキュメントを参照してください。

を使用してハイブリッドジョブを直接操作する API

を使用して、Amazon Braket Hybrid Jobs に直接アクセスして操作できますAPI。ただし、 APIを直接使用する場合、デフォルトと便利なメソッドは使用できません。

> **Note**
>
> Amazon Braket Python SDK を使用して Amazon Braket Hybrid Jobs を操作することを強くお勧めします。ハイブリッドジョブを正常に実行するのに役立つ便利なデフォルトと保護を提供します。

このトピックでは、 の使用の基本について説明しますAPI。API を使用する場合は、このアプローチがより複雑になり、ハイブリッドジョブを実行するための反復処理の準備が整う可能性があることに注意してください。

API を使用するには、アカウントに `AmazonBraketFullAccess`管理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。

さらに、実行ロールが必要です。このロールは サービスに渡されます。 Amazon Braket コンソールを使用してロールを作成できます。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します。

`CreateJob` API では、ハイブリッドジョブに必要なすべてのパラメータを指定する必要があります。Python を使用するには、アルゴリズムスクリプトファイルを input.tar.gz ファイルなどの tar バンドルに圧縮し、次のスクリプトを実行します。コードの一部を角括弧 (<>) で更新して、ハイブリッドジョブを開始するパス、ファイル、方法を指定するアカウント情報とエントリポイントと一致させます。

```
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を作成したら、 GetJobAPIまたは コンソールからハイブリッドジョブの詳細にアクセスできます。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 () Amazon Resource Name の CancelJobAPIを使用して を呼び出します'JobArn'。

```
cancelJob = client.cancel_job(jobArn=job["jobArn"])
```

checkpointConfig パラメータcreateJobAPIを使用して、 の一部としてチェックポイントを指定できます。

```
    checkpointConfig = {
        "localPath" : "/opt/omega/checkpoints",
        "s3Uri": f"s3://{bucket}/{s3_prefix}/checkpoints"
    },
```

> **Note**
>
> のローカルパス checkpointConfig は、予約済みパスの /opt/ml、/opt/braket、/tmp、または /usr/local/nvidia のいずれかで開始することはできません。

予約の操作

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。都合の良いときに予約をスケジュールできるため、ワークロードの実行の開始時刻と終了時刻を正確に把握できます。予約は 1 時間単位で利用可能で、最大 48 時間前にキャンセルでき、追加料金はかかりません。今後の予約のために量子タスクとハイブリッドジョブをキューに入れるか、予約中にワークロードを送信するかを選択できます。

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。

予約には、次の量子コンピュータを使用できます。

- IonQ の Aria と Forte
- Rigetti の Ankaa-3
- IQM のガーネット
- QuEra の Aquila

> **Note**
>
> IonQ デバイスで直接予約を使用する場合、ゲートショットの制限はなく、エラー緩和タスクでは最低 500 ショットです。

予約を使用するタイミング

予約で専用デバイスアクセスを利用すると、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比較して、他のカスタマータスクでキューに待機することはありません。予約中にデバイスへの排他的なアクセス権があるため、予約中は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い、迅速かつコスト効率の高いアルゴリズムの反復を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しているなど、特定の時間帯にタスクを実行する場合は、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
- 予約中の量子タスクの実行
- 予約中のハイブリッドジョブの実行
- 予約の最後に何が起こるか
- 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

予約を作成するには、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の予約 を選択します。
3. 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を入力します。定期的にチェックする有効な E メールアドレスを必ず指定してください。

5. 「ワークロードについて教えてください」で、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。
6. 予約の確認後に予約準備セッションのために Braket の専門家と接続する場合は、オプションで準備セッションに関心がある を選択します。

以下の手順に従って、当社に連絡して予約を作成することもできます。

1. Amazon Braket コンソールを開きます。
2. 左側のペインでデバイスを選択し、予約するデバイスを選択します。
3. 概要セクションで、デバイスを予約を選択します。
4. 前の手順のステップ 4 ~ 6 に従います。

フォームを送信すると、Braket チームから予約を作成するための次のステップが記載された E メールが届きます。予約が確認されると、E メールで予約 ARN が送信されます。

> **Note**
>
> 予約は、予約 ARN を受け取った後にのみ確認されます。

予約は最低 1 時間単位で利用可能で、特定のデバイスに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) がある場合があります。Braket チームは、予約を確認する前に関連情報をすべて共有します。

予約準備セッションに関心を示した場合、Braket チームは E メールでお客様に連絡し、Braket エキスパートとの 30 分間のセッションを手配します。

予約中の量子タスクの実行

予約の作成から有効な予約 ARN https://docs.aws.amazon.com/braket/latest/developerguide/braket-reservations.html#braket-create-a-reservationを取得したら、予約中に実行する量子タスクを作成できます。これらのタスクは、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。

> **Note**
>
> 予約 ARN で送信されたタスクとジョブのキューの可視性はありません。これは、予約中にタスクのみが実行されるためです。

Python SDKs Braket、、Qiskit、または boto3 (Boto3 の使用) を使用してPennyLane直接量子タスクを作成できます。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。次のコードを使用して、最新の Braket SDK、Qiskitプロバイダー、PennyLaneプラグインに更新できます。

```
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内で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は、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で実行されるようにします。

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。

```
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.Aria1)

# 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る限り、 PennyLaneおよび 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。たとえば、 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)

aria = BraketProvider().get_backend("Aria 1")

# run the circuit in a reservation
with DirectReservation(Devices.IonQ.Aria1, reservation_arn="<my_reservation_arn>"):
    aria_task = aria.run(qc, shots=10)
```

同様に、次のコードは、 Braket-PennyLaneプラグインを使用して予約中に回路を実行します。

```
from braket.devices import Devices
from braket.aws import DirectReservation
import pennylane as qml


dev = qml.device("braket.aws.qubit", device_arn=Devices.IonQ.Aria1.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.Aria1, 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、次のコードを使用して予約コンテキストを手動で設定できます。

```
# set reservation context
reservation = 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で実行される Jupyter ノートブックに最適です。

> **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 ノートブックを再起動するか、以下を呼び出してコンテキストをオンデマンドモードに戻します。

```
reservation.stop()  # unset reservation context
```

> **Note**
>
> 予約には開始時刻と終了時刻がスケジュールされています (「予約の作成」を参照)。`reservation.start()` および `reservation.stop()`メソッドは予約を開始または終了しません。これらは、予約中に実行する後続のすべての量子タスクを変更する方法です。これらの方法は、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

予約中にタスクを作成するもう 1 つの方法は、 を呼び出すときに予約 ARN を明示的に渡すことですdevice.run()。

```
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。このオプションでは、予約中に実行する予定の各タスクに予約 ARN を追加します。さらに、 Qiskitまたは で作成されたタスクPennyLaneが正しい予約 ARN を使用していることを確認します。これらの追加の考慮事項により、前の 2 つの方法をお勧めします。

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)
```

予約中のハイブリッドジョブの実行

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、予約でハイブリッドジョブを実行できます。ハイブリッドジョブ内のすべてのタスクは予約 ARN を使用します。重要なのは、 のハイブリッドジョブは、予約が開始された後reservation_arnにのみクラシックコンピューティングを起動することです。

> **Note**
>
> 予約中に実行されるハイブリッドジョブは、リザーブドデバイスで量子タスクのみ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、DirectReservation代わりに を使用します。

次のコードは、予約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.Aria1, 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「最初のハイブリッドジョブの作成」のセクションを参照) 、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内で実行できます。

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

job = AwsQuantumJob.create(
    Devices.IonQ.Aria1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    reservation_arn="<my_reservation_arn>"
)
```

予約の最後に何が起こるか

予約が終了すると、デバイスへの専用アクセスはできなくなります。この予約でキューに入っている残りのワークロードは自動的にキャンセルされます。

> **Note**
>
> 予約終了時にRUNNINGステータスにあったジョブはすべてキャンセルされます。チェックポイントを使用して、都合の良いときにジョブを保存および再起動することをお勧めします。

予約開始後や予約終了前などの継続的な予約は、各予約がスタンドアロンの専用デバイスアクセスを表すため、延長できません。例えば、2 つのback-to-back予約は個別と見なされ、最初の予約から保留中のタスクは自動的にキャンセルされます。2 番目の予約では再開されません。

> **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、他のお客様は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、スケジュールされた予約開始時刻の 48 時間以上前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で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

量子エラーの軽減は、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って制限されています。短期的にこれに対処するために、研究者はノイズの多い量子計算の精度を向上させる方法を調査しています。このアプローチは量子誤差緩和と呼ばれ、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 デバイスのエラー軽減手法

IonQ デバイスのエラー軽減手法

エラーの軽減には、複数の物理回路を実行し、測定値を組み合わせて結果を向上させる必要があります。

> **Note**
>
> のすべての IonQデバイスの場合: オンデマンドモデルを使用する場合、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 2500 ショットがあります。直接予約の場合、ゲートショット制限はなく、エラー緩和タスクでは最低 500 ショットです。

バイアスの排除

IonQ デバイスには、バイアス解除と呼ばれるエラー緩和方法があります。

Debiasing は、回路を、異なる量子ビットの置換または異なるゲート分解で動作する複数のバリアントにマッピングします。これにより、測定結果をバイアスする可能性のある回路の異なる実装を使用することで、ゲートのオーバーローテーションや単一の欠陥量子ビットなどの体系的なエラーの影響が軽減されます。これは、複数の量子ビットとゲートをキャリブレーションするための余分なオーバーヘッドを犠牲にします。

バイアス排除の詳細については、「対称化による量子コンピュータのパフォーマンスの向上」を参照してください。

> **Note**
>
> バイアス解除を使用するには、少なくとも 2500 ショットが必要です。

量子タスクは、次のコードを使用してIonQデバイス上でバイアスを取り除きながら実行できます。

```
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/Aria-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任意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とカウントは 1 つのディストリビューションに集約されます。期待値など、回路で指定された結果タイプは、総測定数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、バリアント間で最も可能性の高い測定結果を優先します。詳細については、「対称化による量子コンピュータのパフォーマンスの向上」を参照してください。

重要なのは、シャープニングでは、出力分布の形式が疎で、確率の高い状態はほとんどなく、確率の低い状態が多いことを前提としていることです。この仮定が有効でない場合、確率分布が歪む可能性があります。

`GateModelTaskResult` Braket Python SDK の `additional_metadata`フィールドで、シャープ化されたディストリビューションから確率にアクセスできます。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、シャープニング後にディストリビューション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 のトラブルシューティング

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して、Amazon Braket の問題を解決します。

このセクションの内容:

- AccessDeniedException
- CreateQuantumTask オペレーションの呼び出し中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動作を停止しました
- OpenQASM のトラブルシューティング

AccessDeniedException

Braket を有効化または使用するときに AccessDeniedException を受け取った場合、制限されたロールに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、次の条件のうちどれが当てはまるかを理解する必要があります。

- リージョンへのアクセスを妨げるロール制限がある場合。
- 使用しようとしているロールに Braket の使用が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 オペレーションの呼び出し中にエラーが発生しました (ValidationException)

次のようなエラーが表示された場合は、既存の `An error occurred (ValidationException) when calling the CreateQuantumTask operation: Caller doesn't have access to`

amazon-braket-… 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的に作成しません。

API に直接アクセスしていて、次のような エラーが表示される場合は、Amazon Failed to create quantum task: Caller doesn't have access to s3://MY_BUCKET S3 バケットパスs3://に を含めていないことを確認します。 Amazon S3

SDK 機能が動作しません

Python のバージョンは 3.9 以降である必要があります。Amazon Braket Hybrid Jobs では、Python 3.10 をお勧めします。

SDK とスキーマがup-to-date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、次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
```

独自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ます。

ServiceQuotaExceededException が原因でハイブリッドジョブが失敗する

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は、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た場合、作成に失敗することがあります。サービスの制限の詳細については、「クォータ」トピックを参照してください。

アカウントから複数のハイブリッドジョブでシミュレーターデバイスに対して同時タスクを実行している場合、このエラーが発生する可能性があります。

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API、 search-quantum-tasks 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
```

Amazon CloudWatch メトリクスを使用して、デバイスに対して作成された量子タスクを表示することもできます: Braket > Device 別。

これらのエラーが発生しないようにするには：

1. シミュレーターデバイスの同時量子タスクの数に対するサービスクォータの引き上げをリクエストします。これは 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動作を停止しました

ノートブックの一部のコンポーネントが機能しない場合は、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。

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ポインタについて説明します。

このセクションの内容:

- Include ステートメントエラー
- 連続しないqubitsエラー
- 物理エラーqubitsと仮想qubitsエラーの混在

- 同じプログラムエラーqubitsでの結果タイプのリクエストと測定
- Classical および qubit register limits exceeded エラー
- 逐語的なプラグマエラーが前にないボックス
- ネイティブゲートが欠落している逐語的なボックスのエラー
- 逐語的なボックスで物理qubitsエラーが見つからない
- 逐語的なプラグマに「ブラケット」エラーがない
- 単一 はインデックス作成qubitsできないエラー
- 2 つのqubitゲートqubitsの物理 が接続されていないエラー
- Local Simulator のサポートに関する警告

Include ステートメントエラー

Braket には、現在、OpenQASM プログラムに含め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ます。

```
OPENQASM 3;
include "standardlib.inc";
```

このコードは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エラー

デバイス機能qubitsで `requiresContiguousQubitIndices`に設定されているデバイスで非連続 `true` を使用すると、エラーが発生します。

シミュレーターと で量子タスクを実行するとIonQ、次のプログラムによって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].`

物理エラーqubitsと仮想qubitsエラーの混在

同じプログラムqubitsで物理qubitsと仮想を混在させることは許可されず、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;
cnot q[0], $1;
```

このコードはエラーメッセージを生成します。`[line 4] mixes physical qubits and qubits registers.`

同じプログラムエラーqubitsでの結果タイプのリクエストと測定

同じプログラムで明示的に測定qubitsされる結果タイプと をリクエストすると、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 および qubit register limits exceeded エラー

1 つのクラシックレジスターと 1 つのqubitレジスター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.`

逐語的なプラグマエラーが前にないボックス

すべてのボックスの前には、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.`

ネイティブゲートが欠落している逐語的なボックスのエラー

逐語的なボックスには、ネイティブゲートと物理 が必要ですqubits。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.`

逐語的なボックスで物理qubitsエラーが見つからない

逐語的なボックスには物理的な が必要ですqubits。次のコードは、欠落している物理qubits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「ブラケット」エラーがない

逐語的なプラグマには「ブラケット」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
```

このコードはエラーメッセージを生成します。 `You must include “braket” in the verbatim pragma`

単一 はインデックス作成qubitsできないエラー

シングル はインデックス作成qubits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];
```

このコードはエラーを生成します。 `[line 4] single qubit cannot be indexed.`

ただし、単一のqubit配列は次のように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qubitsの物理 が接続されていないエラー

物理 を使用するにはqubits、まずデバイスが物理 を使用していることを確認し qubitsdevice.properties.action[DeviceActionType.OPENQASM].supportPhysicalQubits、

次に device.properties.paradigm.connectivity.connectivityGraphまたは を確認して接続グラフを確認しますdevice.properties.paradigm.connectivity.fullyConnected。

```
OPENQASM 3;

cnot $0, $14;
```

このコードはエラーメッセージを生成します。 [line 3] has disconnected qubits 0 and 14

Local Simulator のサポートに関する警告

は、QPUs 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LocalSimulatorをサポートしています。次の例に示すようにLocalSimulator、プログラムに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、「このプログラムは LocalSimulator でのみサポートされている OpenQASM 言語機能を使用します。これらの機能の一部は、QPUs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場合があります。

サポートされている OpenQASM 機能の詳細については、ローカルシミュレーターの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

でのクラウドセキュリティが最優先事項 AWS です。 AWS のお客様は、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 とお客様の間で共有される責任です。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があります AWS クラウド。 AWS また、 は、お客様が安全に使用できるサービスも提供します。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、サードパーティーの監査者は定期的にセキュリティの有効性をテストおよび検証。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、ユーザーは、データの機密性、会社の要件、適用される法律や規制など、その他の要因についても責任を負います。

このドキュメントは、Braket を使用する際の責任共有モデルの適用方法を理解するのに役立ちます。以下のトピックでは、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ように Braket を設定する方法について説明します。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 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 とお客様の間で共有される責任です。責任共有モデルでは、この責任が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されています。

- 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があります AWS クラウド。 AWS また、 は、お客様が安全に使用できるサービスも提供します。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の詳細については、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の制御を維持する責任はお客様にあります。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

Amazon Braket でのデータ保護には、 AWS の責任共有モデルが適用されます。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 Identity Center 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 が必須で、TLS 1.3 をお勧めします。
- 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 を操作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するすべての量子タスク IDs と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 日以上保存されている過去の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付けられた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 を実行したり、特定のユーザーとロールのアクセスを制限したり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、次のセクションで説明するように、アカウントの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、必ず、(1) 管理者権限、または (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があ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
- AmazonBraketFullAccess ポリシーについて
- AmazonBraketJobsExecutionPolicy ポリシーについて
- 特定のデバイスへのユーザーアクセスを制限する
- AWS マネージドポリシーに対する Amazon Braket の更新
- 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、量子タスクリソースの 1 つのタイプの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 Noteboook リソースタイプを使用できます。ノートブックは、Braket が共有できる Amazon SageMaker AI リソースです。Braket でノートブックを使用するには、 で始まる名前の IAM ロールを指定する必要がありますAmazonBraketServiceSageMakerNotebook。

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。

```
{
   "Version": "2012-10-17",
   "Statement": [
       {
           "Effect": "Allow",
           "Action": "iam:CreateRole",
```

```
            "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole*"
        },
        {
            "Effect": "Allow",
            "Action": "iam:CreatePolicy",
            "Resource": [
                "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookAccess*",
                "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookRole*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iam:AttachRolePolicy",
            "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole*",
            "Condition": {
                "StringLike": {
                    "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、「ノートブックの作成」ページに記載されているステップに従うか、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。

AmazonBraketFullAccess ポリシーについて

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も含まれます。

- 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」の形式に従う必要があります。
- 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。
- ロールを使用して リソースを制御する – アカウントにサービスにリンクされたロールを作成します。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 に渡しCreateJobAPI、ロールを作成し、AmazonBraketFullAccess の対象となるポリシーをロールにアタッチします。
- アカウントの使用状況ログファイルを維持するために、ロググループ、ログイベント、クエリログループを作成する – 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、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 パスを囲み、ログデータを配置できるようにします。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 バケットを作成するには、アカウントの 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の任意のバケットにオブジェクトを配置して取得します。これらのアクセス許可は、Braket が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に配置し、バケットから取得するために必要です。
- IAM ロールを渡す – IAM ロールを CreateJob に渡しますAPI。
- 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のSageMakerノートブックインスタンスを作成および管理します。
- サービスクォータの検証 – SageMaker AI ノートブックと Amazon Braket Hybrid 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はできません。
- 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。

ポリシーの内容

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:ListBucket",
                "s3:CreateBucket",
                "s3:PutBucketPublicAccessBlock",
                "s3:PutBucketPolicy"
            ],
            "Resource": "arn:aws:s3:::amazon-braket-*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceAccount": "${aws:PrincipalAccount}"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:ListAllMyBuckets",
                "servicequotas:GetServiceQuota",
                "cloudwatch:GetMetricData",
                "pricing:GetProducts"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage",
                "ecr:BatchCheckLayerAvailability"
            ],
            "Resource": "arn:aws:ecr:*:*:repository/amazon-braket*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ecr:GetAuthorizationToken"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
```

```
            "Action": [
                "logs:Describe*",
                "logs:Get*",
                "logs:List*",
                "logs:StartQuery",
                "logs:StopQuery",
                "logs:TestMetricFilter",
                "logs:FilterLogEvents"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/braket*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:ListRoles",
                "iam:ListRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
                "iam:ListAttachedRolePolicies"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "sagemaker:ListNotebookInstances"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "sagemaker:CreatePresignedNotebookInstanceUrl",
                "sagemaker:CreateNotebookInstance",
                "sagemaker:DeleteNotebookInstance",
                "sagemaker:DescribeNotebookInstance",
                "sagemaker:StartNotebookInstance",
                "sagemaker:StopNotebookInstance",
                "sagemaker:UpdateNotebookInstance",
                "sagemaker:ListTags",
                "sagemaker:AddTags",
                "sagemaker:DeleteTags"
            ],
            "Resource": "arn:aws:sagemaker:*:*:notebook-instance/amazon-braket-*"
```

```
        },
        {
            "Effect": "Allow",
            "Action": [
                "sagemaker:DescribeNotebookInstanceLifecycleConfig",
                "sagemaker:CreateNotebookInstanceLifecycleConfig",
                "sagemaker:DeleteNotebookInstanceLifecycleConfig",
                "sagemaker:ListNotebookInstanceLifecycleConfigs",
                "sagemaker:UpdateNotebookInstanceLifecycleConfig"
            ],
            "Resource": "arn:aws:sagemaker:*:*:notebook-instance-lifecycle-config/
amazon-braket-*"
        },
        {
            "Effect": "Allow",
            "Action": "braket:*",
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-service-role/braket.amazonaws.com/
AWSServiceRoleForAmazonBraket*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "braket.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole*",
            "Condition": {
                "StringLike": {
                    "iam:PassedToService": [
                        "sagemaker.amazonaws.com"
                    ]
                }
            }
        },
```

```
        {
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketJobsExecutionRole*",
            "Condition": {
                "StringLike": {
                    "iam:PassedToService": [
                        "braket.amazonaws.com"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:GetQueryResults"
            ],
            "Resource": [
                "arn:aws:logs:*:*:log-group:*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogStream",
                "logs:CreateLogGroup"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/braket*"
        },
        {
            "Effect": "Allow",
            "Action": "cloudwatch:PutMetricData",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "cloudwatch:namespace": "/aws/braket"
                }
            }
        }
    ]
```

```
}
```

AmazonBraketJobsExecutionPolicy ポリシーについて

AmazonBraketJobsExecutionPolicy ポリシーは、Amazon Braket Hybrid Jobs で使用される実行ロールのアクセス許可を次のように付与します。

- 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の読み取りとダウンロードのアクセス許可。コンテナは「arn:aws:ecr:*:*:repository/amazon-braket*」の形式に従う必要があります。
- アカウントの使用状況ログファイルを維持するために、ロググループ、ログイベント、クエリロググループを作成する – 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、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 パスを囲み、ログデータを配置できるようにします。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 – アカウントの S3 バケットを一覧表示し、amazon-braket- で始まるアカウントの任意のバケットにオブジェクトを配置して、その名前でオブジェクトを取得します。これらのアクセス許可は、Braket が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に配置し、バケットから取得するために必要です。
- IAM ロールを渡す – IAM ロールを CreateJob に渡しますAPI。ロールは arn:aws:iam::*:role/service-role/AmazonBraketJobsExecutionRole* の形式に従う必要があります。

```
"Version": "2012-10-17",
"Statement": [
 {
  "Effect": "Allow",
  "Action": [
   "s3:GetObject",
   "s3:PutObject",
   "s3:ListBucket",
   "s3:CreateBucket",
   "s3:PutBucketPublicAccessBlock",
   "s3:PutBucketPolicy"
  ],
  "Resource": "arn:aws:s3:::amazon-braket-*"
 },
 {
  "Effect": "Allow",
  "Action": [
   "ecr:GetDownloadUrlForLayer",
```

```
    "ecr:BatchGetImage",
    "ecr:BatchCheckLayerAvailability"
   ],
   "Resource": "arn:aws:ecr:*:*:repository/amazon-braket*"
  },
  {
   "Effect": "Allow",
   "Action": [
    "ecr:GetAuthorizationToken"
   ],
   "Resource": "*"
  },
  {
   "Effect": "Allow",
   "Action": [
    "braket:CancelJob",
    "braket:CancelQuantumTask",
    "braket:CreateJob",
    "braket:CreateQuantumTask",
    "braket:GetDevice",
    "braket:GetJob",
    "braket:GetQuantumTask",
    "braket:SearchDevices",
    "braket:SearchJobs",
    "braket:SearchQuantumTasks",
    "braket:ListTagsForResource",
    "braket:TagResource",
    "braket:UntagResource"
   ],
   "Resource": "*"
  },
  {
   "Effect": "Allow",
   "Action": [
    "iam:PassRole"
   ],
   "Resource": "arn:aws:iam::*:role/service-role/AmazonBraketJobsExecutionRole*",
   "Condition": {
    "StringLike": {
     "iam:PassedToService": [
      "braket.amazonaws.com"
     ]
    }
   }
```

```
  },
  {
   "Effect": "Allow",
   "Action": [
    "iam:ListRoles"
   ],
   "Resource": "arn:aws:iam::*:role/*"
  },
  {
   "Effect": "Allow",
   "Action": [
    "logs:GetQueryResults"
   ],
   "Resource": [
    "arn:aws:logs:*:*:log-group:*"
   ]
  },
  {
   "Effect": "Allow",
   "Action": [
    "logs:PutLogEvents",
    "logs:CreateLogStream",
    "logs:CreateLogGroup",
    "logs:GetLogEvents",
    "logs:DescribeLogStreams",
    "logs:StartQuery",
    "logs:StopQuery"
   ],
   "Resource": "arn:aws:logs:*:*:log-group:/aws/braket*"
  },
  {
   "Effect": "Allow",
   "Action": "cloudwatch:PutMetricData",
   "Resource": "*",
   "Condition": {
    "StringEquals": {
     "cloudwatch:namespace": "/aws/braket"
    }
   }
  }
 ]
}
```

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 Braket デバイスのユーザーアクセスを制限するには、特定のIAMロールに拒否アクセス許可ポリシーを追加できます。

以下のアクションを制限できます。

- `CreateQuantumTask` - 指定され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の取得を拒否します。

次の例では、 のすべての QPUs へのアクセスを制限します AWS アカウント 123456789012。

```
{
  "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**
>
> Braket コンソールを介してデバイスの可用性、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、制限されたデバイスのAmazonリソース番号 (ARN) を前の例に示す文字列に置き換えます。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 を選択するには次の手順を実行します。 `arn:aws:braket:*:*:device/qpu/*`
- us-west-2 リージョンのすべての QPU のみを選択するには次の手順を実行します。 `arn:aws:braket:us-west-2:*:device/qpu/*`
- 同様に、us-west-2 リージョンのみのすべての QPUs を選択するには (デバイスは顧客リソースではなくサービスリソースであるため) 。 `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には: `arn:aws:braket:*:*:device/quantum-simulator/*`
- 特定のプロバイダーからのデバイス (デバイスなどRigettiQPU) へのアクセスを制限するには: `arn:aws:braket:*:*:device/qpu/rigetti/*`
- TN1 デバイスへのアクセスを制限するには: `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には: `braket:Create*`

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てからの 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
AmazonBraketFullAccess - Braket のフルアクセスポリシー	「pricing:GetProducts」アクションを追加しました。	2025 年 4 月 14 日

変更	説明	日付
AmazonBraketFullAccess - Braket のフルアクセスポリシー	S3 アクションに「aws:ResourceAccount」：「${aws:PrincipalAccount}」条件スコープを追加しました。	2025 年 3 月 3 日
AmazonBraketFullAccess - Braket のフルアクセスポリシー	AmazonBraketFullAccess ポリシーに含める servicequotas:GetServiceQuota および cloudwatch:GetMetricData アクションを追加しました。 .	2023 年 3 月 24 日
AmazonBraketFullAccess - Braket のフルアクセスポリシー	Braket はAmazonBraketFullAccess にservice-role/ パスを含めるように iam:PassRole アクセス許可を調整しました。	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 は、ハイブリッドジョブ実行ロール ARN を更新してservice-role/ 、パスを含めました。	2021 年 11 月 29 日
Braket が変更の追跡を開始しました	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、またはグループに拒否アクセス許可ポリシーを追加できます。

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起動、停止、および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に応じて名前が付けられAliceます (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)。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "sagemaker:CreateNotebookInstance",
        "sagemaker:DeleteNotebookInstance",
        "sagemaker:UpdateNotebookInstance",
        "sagemaker:CreateNotebookInstanceLifecycleConfig",
        "sagemaker:DeleteNotebookInstanceLifecycleConfig",
        "sagemaker:UpdateNotebookInstanceLifecycleConfig"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Deny",
      "Action": [
        "sagemaker:DescribeNotebookInstance",
        "sagemaker:StartNotebookInstance",
        "sagemaker:StopNotebookInstance",
      ],
      "NotResource": [
        "arn:aws:sagemaker:*:123456789012:notebook-instance/amazon-braket-
${aws:username}"
      ]
    },
    {
      "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できます。

次の 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し、それらのオブジェクトのリストも制限します。

```
{
  "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 サービスにリンクされたロールは、 AWS のサービス ユーザーに代わって

他の 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める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 がセットアップするサービスにリンクされたロールは、 AWS Identity and Access Management (IAM) サービスにリンクされたロール機能の一部です。サービスにリンク AWS のサービス されたロールをサポートする他の の詳細については、AWS 「IAM と連携する のサービス」を参照し、「サービスにリンクされたロール」列で「はい」があるサービスを探します。サービスにリンクされたロールに関するドキュメントをサービスで表示するには、[Yes] (はい) リンクを選択します。

このセクションの内容:

- Amazon Braket のサービスリンクロール許可

Amazon Braket のサービスリンクロール許可

Amazon Braket は、braket.amazonaws.com エンティティを信頼するAWSServiceRoleForAmazonBraketサービスにリンクされたロールを使用してロールを引き受けます。

サービスにリンクされたロールの作成、編集、削除を IAM エンティティ (グループまたはロールなど) に許可するには、アクセス許可を設定する必要があります。詳細については、「サービスにリンクされたロールのアクセス許可」を参照してください。

Amazon Braket のサービスリンクロールには、デフォルトで次のアクセス許可が付与されます。

- Amazon S3 – アカウント内のバケットを一覧表示し、amazon-braket- で始まる名前のアカウント内の任意のバケットにオブジェクトを配置および取得するアクセス許可。
- Amazon CloudWatch Logs – ロググループを一覧表示および作成し、関連するログストリームを作成し、Amazon Braket 用に作成されたロググループにイベントを配置するアクセス許可。

次のポリシーが**AWSServiceRoleForAmazonBraket**サービスにリンクされたロールにアタッチされます。

```
{"Version": "2012-10-17",
    "Statement": [
        {"Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": "arn:aws:s3:::amazon-braket*"
        },
        {"Effect": "Allow",
            "Action": [
                "logs:Describe*",
                "logs:Get*",
                "logs:List*",
                "logs:StartQuery",
                "logs:StopQuery",
                "logs:TestMetricFilter",
                "logs:FilterLogEvents"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/braket/*"
        },
        {"Effect": "Allow",
            "Action": "braket:*",
            "Resource": "*"
        },
        {"Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-service-role/braket.amazonaws.com/
AWSServiceRoleForAmazonBraket*",
            "Condition": {"StringEquals": {"iam:AWSServiceName": "braket.amazonaws.com"
                }
            }
        }
    ]
}
```

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を選択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 QPUs には適用されません。

AWS のサービス が特定のコンプライアンスプログラムの範囲内にあるかどうかを確認するには、コンプライアンスAWS のサービス プログラムによる範囲内コンプライアンスを参照し、関心のあるコンプライアンスプログラムを選択します。一般的な情報については、AWS 「 Compliance ProgramsAssurance」を参照してください。

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 AWS Artifact。詳細については、「Downloading AWS Artifact reports 」を参照してください。

を使用する際のお客様のコンプライアンス責任 AWS のサービス は、お客様のデータの機密性、貴社のコンプライアンス目的、適用される法律および規制によって決まります。 では、コンプライアンスに役立つ以下のリソース AWS を提供しています。

- セキュリティのコンプライアンスとガバナンス – これらのソリューション実装ガイドでは、アーキテクチャ上の考慮事項について説明し、セキュリティとコンプライアンスの機能をデプロイする手順を示します。
- HIPAA 対応サービスのリファレンス – HIPAA 対応サービスの一覧が提供されています。すべてが HIPAA 対応 AWS のサービス であるわけではありません。
- AWS コンプライアンスリソース – このワークブックとガイドのコレクションは、お客様の業界や地域に適用される場合があります。
- AWS カスタマーコンプライアンスガイド – コンプライアンスの観点から責任共有モデルを理解します。このガイドは、複数のフレームワーク (米国国立標準技術研究所 (NIST)、Payment Card Industry Security Standards Council (PCI)、国際標準化機構 (ISO) を含む) のセキュリティコントロールを保護し、そのガイダンスに AWS のサービス マッピングするためのベストプラクティスをまとめたものです。
- 「 デベロッパーガイド」の「ルールによるリソースの評価」 – この AWS Config サービスは、リソース設定が社内プラクティス、業界ガイドライン、および規制にどの程度準拠しているかを評価します。 AWS Config
- AWS Security Hub – これにより AWS のサービス 、セキュリティ状態を包括的に把握できます AWS。Security Hub では、セキュリティコントロールを使用して AWS リソースを評価し、セ

キュリティ業界標準とベストプラクティスに対するコンプライアンスをチェックします。サポートされているサービスとコントロールの一覧については、Security Hub のコントロールリファレンスを参照してください。

- Amazon GuardDuty – 環境をモニタリングして AWS アカウント不審なアクティビティや悪意のあるアクティビティがないか調べることで、、ワークロード、コンテナ、データに対する潜在的な脅威 AWS のサービス を検出します。GuardDuty を使用すると、特定のコンプライアンスフレームワークで義務付けられている侵入検知要件を満たすことで、PCI DSS などのさまざまなコンプライアンス要件に対応できます。
- AWS Audit Manager – これにより AWS のサービス 、 AWS 使用状況を継続的に監査し、リスクの管理方法と規制や業界標準への準拠を簡素化できます。

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 グローバル AWS 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 がインフラストラクチャ AWS 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 Well-Architected Framework」の「Infrastructure Protection」を参照してください。

AWS が公開した API コールを使用して、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、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。

また、リクエストにはアクセスキー ID と、IAM プリンシパルに関連付けられているシークレットアクセスキーを使用して署名する必要があります。またはAWS Security Token Service (AWS STS) を使用して、一時的なセキュリティ認証情報を生成し、リクエストに署名することもできます。

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 はリソースベースのアクセスポリシーをサポートしており、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。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 VPCs からのアクセスを制御することもできます。これにより、

実質的にネットワーク内の特定の VPC からのみ、特定の Braket リソースへの AWS 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、処理のために回路を指定された QPU に送信するときに、識別子として DeviceARN を使用します。

Amazon Braket を使用して、サードパーティーのハードウェアプロバイダーのいずれかが運用する量子コンピューティングハードウェアにアクセスする場合、回路とその関連データは、 が運用する施設以外のハードウェアプロバイダーによって処理されます AWS。各 QPU が利用可能な物理的な場所と 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 の情報はサードパーティーに送信されません。

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 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 AWS 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 間のトラフィックは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され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 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「インターフェイスエンドポイントの前提条件」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、インターフェイスとして Amazon Virtual Private Cloud (Amazon VPC) エンドポイントを作成し、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
- エンドポイントを介して Braket 量子タスクを接続して実行する

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 ユーザーガイドのインターフェイス VPC エンドポイントを使用する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。Braket に対して特定の呼び出しを行う場合は、 `–endpoint-url`フラグの一部として必要ですAPI。

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: エンドポイントを介して 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したら、 `endpoint-url`パラメータを含む CLI コマンドを実行して、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 API と Amazon Braket SDK がデ

フォルトで使用する Braket DNS ホスト名が 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確立する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。

ブログ記事のステップに従っている場合は、AmazonBraket という名前を Amazon SageMaker AI に置き換えてください。リージョンが us-east-1 でない場合は、サービス名に正しい AWS リージョン名前を入力するcom.amazonaws.us-east-1.braketか、その文字列に置き換えます。

エンドポイントの作成に関する追加情報

- プライベートサブネットを持つ VPC を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 エンドポイントの作成」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 AWS CloudFormation、 AWS 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制御するには、 AWS Identity and Access Management (IAM) エンドポイントポリシーを Amazon VPC エンドポイントにアタッチします。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 VPC 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 サービスを通じて量子タスクを送信すると、Amazon Braket SDK と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細にモニタリングできます。これにより、ワークロードの実装を追跡し、潜在的なボトルネックや問題を特定し、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を実行するための一元化されたインターフェイスが提供されます。量子タスクが完了すると、Braket は指定された Amazon S3 の場所に結果を保存します。量子タスクの完了時間は、特に量子処理ユニット (QPU) デバイスで実行されているタスクでは異なる場合があります。これは、量子ハードウェアリソースが複数のユーザー間で共有されるため、主に実行キューの長さによるものです。

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りました。
- `QUEUED` – Amazon Braket が量子タスクを処理し、デバイスでの実行を待っています。
- `RUNNING` – 量子タスクは QPU またはオンデマンドシミュレーターで実行されています。
- `COMPLETED` – QPU またはオンデマンドシミュレーターでの量子タスクの実行が完了しました。
- `FAILED` – 量子タスクの実行が試行され、失敗しました。量子タスクが失敗した理由に応じて、量子タスクを再度送信してみてください。
- `CANCELLED` – 量子タスクをキャンセルしました。量子タスクは実行されませんでした。

このセクションの内容:

- Amazon Braket SDK からの量子タスクの追跡
- 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 による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 からの量子タスクの追跡

コマンドは、一意の量子タスク ID を持つ量子タスクdevice.run(…) を定義します。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: task = device.run()は非同期オペレーションです。つまり、システムが量子タスクをバックグラウンドで処理している間も作業を続けることができます。

結果を取得する

を呼び出すとtask.result()、SDK は Amazon Braket のポーリングを開始し、量子タスクが完了した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、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
```

メタデータの確認

次の例に示すように、完成した量子タスクのメタデータを確認できます。

```
# 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
```

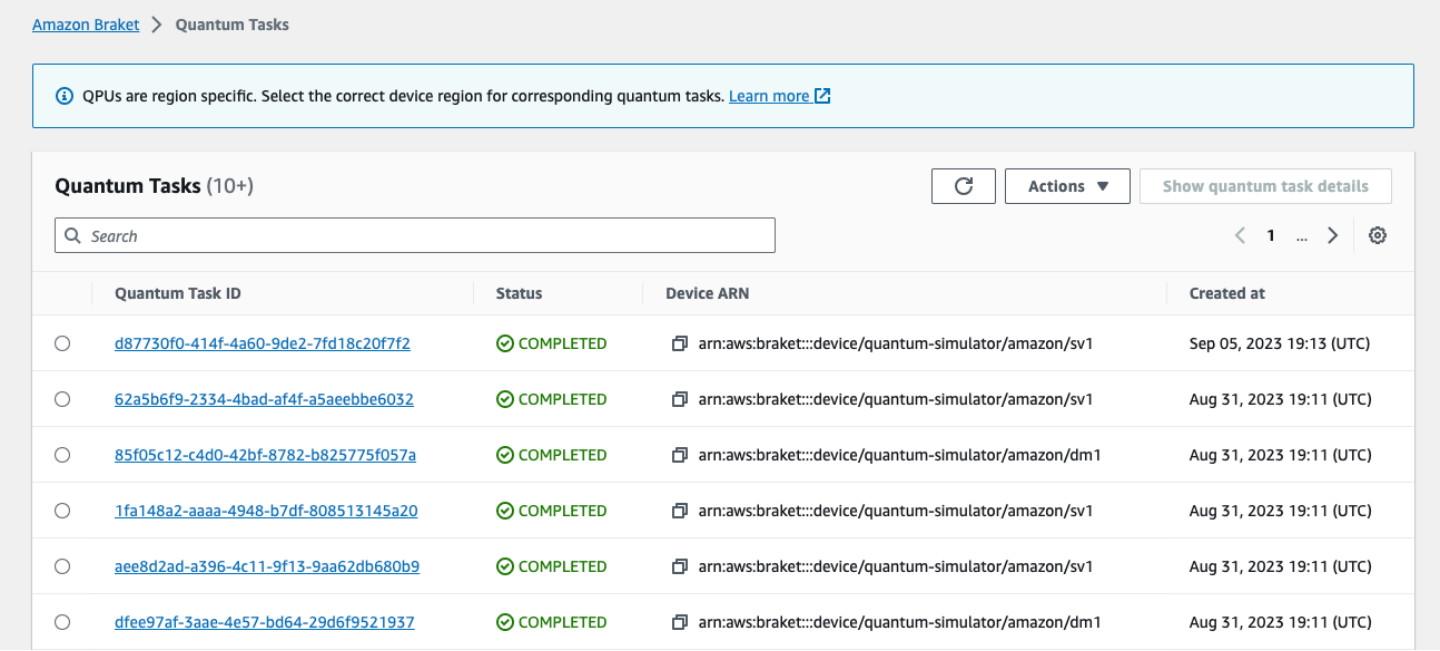
量子タスクまたは結果を取得する

量子タスクを送信した後にカーネルが死亡した場合、またはノートブックまたはコンピュータを閉じた場合、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できます。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。

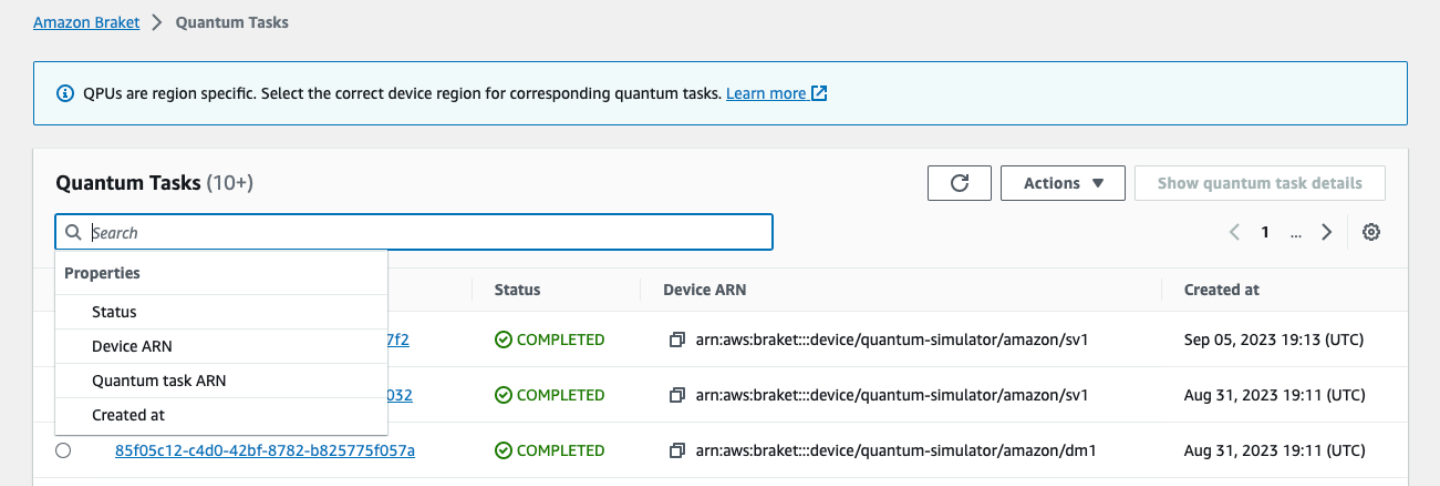
```
from braket.aws import AwsSession, AwsQuantumTask

# 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

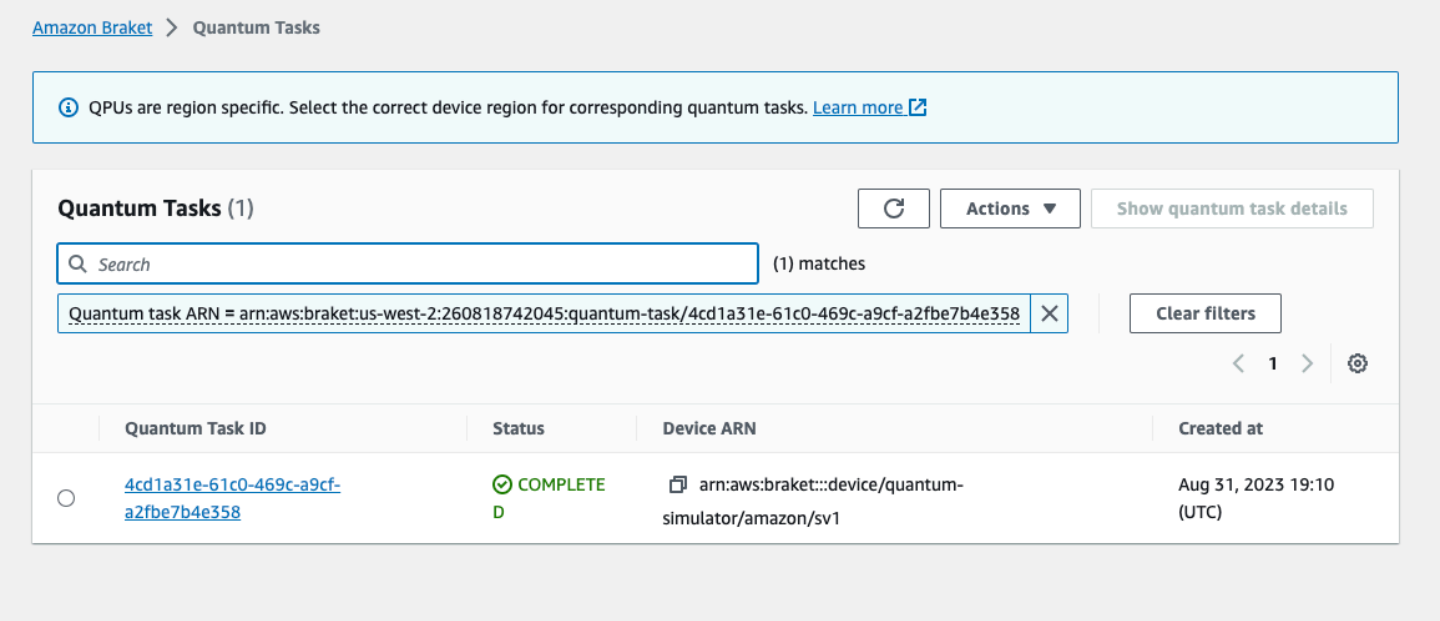
Amazon Braket は、Amazon Braket コンソールを使用して量子タスクをモニタリングする便利な方法を提供します。次の図に示すように、送信されたすべての量子タスクが量子タスクフィールドに一覧表示されます。このサービスはリージョン固有です。つまり、特定の で作成された量子タスクのみを表示できます AWS リージョン。



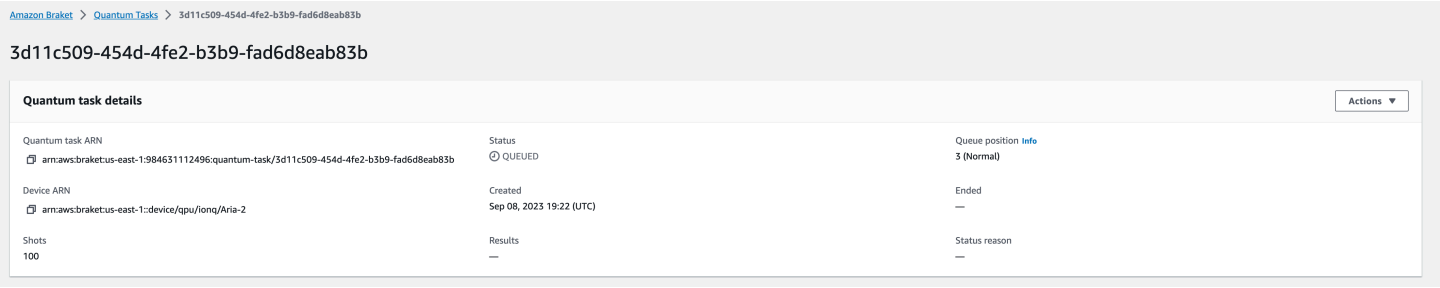
ナビゲーションバーから特定の量子タスクを検索できます。検索は、量子タスク ARN (ID)、ステータス、デバイス、作成時刻に基づいて行うことができます。次の例に示すように、ナビゲーションバーを選択すると、オプションが自動的に表示されます。



次の図は、 を呼び出すことで取得できる一意の量子タスク ID に基づいて量子タスクを検索する例を示していますtask.id。



さらに、次の図に示すように、量子タスクのステータスは QUEUED状態のとき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、処理す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、キューに入っているときに優先されます。ハイブリッドジョブの外部で送信される量子タスクは、通常のキューイング優先度になります。

Braket SDK をクエリしたいお客様は、量子タスクとハイブリッドジョブキューの位置をプログラムで取得できます。詳細については、「タスクの実行日時」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 AWS リソースに割り当てるカスタム属性ラベルです。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。ユーザーが割り当てるタグでは、ユーザーが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、それに関連付けられた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、「 Tagging Resources and Tag Editor User Guide」の「Tag 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 でのタグ付けでサポートされているリソース
- 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。
- タグ値と呼ばれるオプションのフィールド (111122223333 や Production など) 。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の制御」を参照してください。

Amazon Braket でのタグ付けでサポートされているリソース

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： 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、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が、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 を使用してリソースにタグAPIを設定している場合は、 を呼び出しますTagResourceAPI。

```
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、 を呼び出しますUntagResourceAPI。

```
aws braket list-tags-for-resource --resource-arn $YOUR_TASK_ARN
```

- 特定のリソースにアタッチされているすべてのタグを一覧表示するには、 を呼び出しますListTagsForResourceAPI。

```
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 、または Amazon Braket を使用して、タグを表示、追加、変更、一覧表示API、削除できます AWS CLI。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:

- タグの追加
- タグの表示
- タグの編集
- タグの削除

タグの追加

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`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リソースに移動するか、次のコマンドを使用してタグ可能なリソースにアタッチされたタグの値を変更することで、タグを編集できます。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) SV1 にパラメータ設定が指定されたRigetti量子シミュレーターでタスクが実行されています。サンプルコマンド内では、タグは、他のすべての必須パラメータの最後に指定されることは暗黙です。この場合、タグのキーは state、値は ですWashington。これらのタグは、この特定の量子タスクを分類または識別するために使用できます。

```
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\"}
```

この例では、 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ます。これは AWS CLI Braket リソースの整理と追跡に役立ちます。

EventBridge による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。簡単なルールを記述して、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。

- qauntum タスクの変更の状態

Amazon Braket は、量子タスクのステータス変更イベントの配信を保証します。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、「Amazon EventBridge のイベント」を参照してください。

このセクションの内容:

- EventBridge で量子タスクのステータスをモニタリングする
- Amazon Braket EventBridge イベントの例

EventBridge で量子タスクのステータスをモニタリングする

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。たとえば、量子タスクのステータスが変わるたびに 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。
   - [Event pattern] (イベントパターン) セクションで [Custom patterns (JSON editor)] (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、量子タスクの状態変更イベントが 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。

ルールの作成の詳細については、「Creating Amazon EventBridge rules that react to events」を参照してください。

Amazon Braket EventBridge イベントの例

Amazon Braket Quantum 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、「Amazon EventBridge のイベント」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[str]): 量子タスクが移行されたステータス。
- **deviceArn** (str): この量子タスクが作成されたユーザーによって指定されたデバイス。
- shots (int): ユーザーによってshotsリクエストされた の数。
- **outputS3Bucket** (str): ユーザーによって指定された出力バケット。
- **outputS3Directory** (str): ユーザーが指定した出力キープレフィックス。
- **createdAt** (str): ISO-8601 文字列としての量子タスクの作成時間。
- **endedAt** (オプション[str]): 量子タスクが終了状態になった時刻。このフィールドは、量子タスクが終了状態に移行した場合にのみ存在します。

次の JSON コードは、AmazonBraket Quantum 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 か月前に生成された履歴情報を表示するか、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 の動作をより的確に把握できます。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 Braket ノートブックの CloudWatch ログストリームを表示するには、Amazon SageMaker AI コンソールのノートブックの詳細ページに移動します。追加の Amazon Braket ノートブック設定は、SageMaker コンソールから利用できます。

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。

量子タスクメトリクス

量子タスクが存在する場合、メトリクスを使用できます。CloudWatch コンソールの AWS/Braket/By Device 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、量子タスクが完了すると出力されます。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、arn:aws:braket::device/xxx という形式を持つ `deviceArn`パラメータに基づくディメンションで発行されます。

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 のすべてのAPI呼び出し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 [Event history] (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 の Amazon Braket 情報
- Amazon Braket ログファイルエントリについて

CloudTrail 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 アクションは CloudTrail によって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について

「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。イベントは任意ソースからの単一リクエストを表し、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の情報を含みます。CloudTrail ログファイルは、パブリック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、 パラメータpoll_timeout_seconds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、量子タスクを長時間実行でき、量子タスクの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 ノートブックではなく 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、回路を作成し、実行するデバイスに送信して、この例に示すように何が起こるかを確認できます。

```
# 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

次のコマンドを入力して、ファイルに書き込まれる内容を確認できます。

```
# 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

前の例に示すように、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。ARN ID を使用すると、完了した量子タスクの結果を取得できます。

```
# 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。サービスクォータ (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 SV1 (DM1) 量子タスクの調整可能なクォータは、あたり最大 60 個まで可能です AWS リージョン。
- 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、クォータは調整可能です。

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	はい
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	いいえ
`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	はい

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	いいえ
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。	1 秒あたり 5	はい
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	いいえ
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	はい

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	いいえ
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	はい
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	いいえ
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。	1 秒あたり 5	はい

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	いいえ
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。	1 秒あたり 5	はい
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	いいえ
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。	毎秒 1 回	はい

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	いいえ
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。	1 秒あたり 5	はい
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	いいえ
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。	1 秒あたり 5	はい
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	いいえ

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	はい
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	いいえ
同時SV1量子タスクの数	現在のリージョンで状態ベクトルシミュレーター (SV1) で実行されてい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	いいえ
同時DM1量子タスクの数	現在のリージョンで密度マトリックスシミュレーター (DM1) で実行されてい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	いいえ
同時TN1量子タスクの数	現在のリージョンでテンソルネットワークシミュレーター (TN1) で実行されてい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の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のランタイム制限	ハイブリッドジョブを実行できる最大日数。	5	いいえ

ハイブリッドジョブのデフォルトのクラシックコンピューティングインスタンスクォータを次に示します。これらのクォータを引き上げるには、 にお問い合わせくださいサポート。さらに、使用可能なリージョンはインスタンスごとに指定されます。

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の最大数。	5	はい	あり	あり	あり	はい	いいえ

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.c4.2xlarge 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	あり	あり	あり	はい	いいえ

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	あり	あり	あり	はい	いいえ

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	あり	あり	はい	いいえ	いいえ

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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	はい	いいえ	いいえ

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	あり	あり	はい	いいえ	いいえ

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	あり	あり	はい	いいえ	いいえ

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	あり	あり	はい	いいえ	いいえ

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	あり	あり	はい	いいえ	いいえ

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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	はい	いいえ

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	あり	あり	あり	はい	いいえ

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	あり	あり	あり	はい	いいえ

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	あり	あり	あり	はい	いいえ

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	あり	あり	あり	はい	いいえ

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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	あり	あり	あり	あり	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	はい	なし	あり	いいえ	いいえ

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	はい	なし	あり	いいえ	いいえ

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	はい	なし	あり	いいえ	いいえ

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.p3.2xlarge 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.p3.2xlarge タイプのインスタンスの最大数。	0	はい	はい	なし	あり	いいえ	いいえ

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	はい	なし	あり	いいえ	いいえ

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.p3dn.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.p3dn.24xlarge タイプのインスタンスの最大数。	0	はい	はい	なし	あり	いいえ	いいえ

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.p3.8xlarge 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.p3.8xlarge タイプのインスタンスの最大数。	0	はい	はい	なし	はい	はい	いいえ

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.p3.16x 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.p3.16x large タイプのインスタンスの最大数。	0	はい	はい	なし	はい	はい	いいえ

制限の更新のリクエスト

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キャパシティ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ます。さらに、テーブルにインスタンスが表示されない場合は、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、最大 31 量子ビットの回路では 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。
- で許可されるタスクあたりの最大ショット数は 1000 TN1です。
- のすべての IonQデバイスの場合: オンデマンドモデルを使用する場合、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 2500 ショットがあります。直接予約の場合、ゲートショット制限はなく、エラー緩和タスクでは最低 500 ショットです。
- QuEraの Aquila デバイスの場合、最大はタスクあたり 1,000 ショットです。
- IQMの Garnet デバイスの場合、最大はタスクあたり 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

以下の表は、Amazon Braket の今回のリリースのドキュメント内容をまとめたものです。

- API バージョン： 2022 年 4 月 28 日
- API リファレンスの最終更新日： 2023 年 12 月 15 日
- ドキュメントの最終更新日： 2025 年 4 月 14 日

変更	説明	日付
デバイス料金AmazonBraketFullAccess へのアクセスを改善	include を更新AmazonBraketFullAccess pricing:GetProducts して、コンソールにハードウェアコストを表示します。	2025 年 4 月 14 日
新しいデバイス Forte-Enterprise-1	IonQ Forte-Enterprise-1 デバイスのサポートが追加されました。トラップされた ion テクノロジーを利用する 36 個の quibit デバイス。	2025 年 3 月 17 日
S3 条件のアクセス許可を改善	セキュリティを向上させるために、 は にのみs3:*アクションを提供するAmazonBraketFullAccess ようになりましたaws:PrincipalAccount 。これにより、リクエスタ自身のバケットへのアクセスのみが制限されます。	2025 年 3 月 7 日
新しいデバイス Rigetti Ankaa-3	Rigetti Ankaa-3 デバイスのサポートが追加されました。スケーラブルなマルチチップ	2025年1月14日

	テクノロジーを利用する 84 quibit デバイス。	
Rigetti Ankaa-2 デバイスの廃止	Rigetti Ankaa-2 デバイスのサポートを削除しました。	2025年1月14日
IPv6 トラフィックのサポート	Amazon Braket は、デュアルスタックエンドポイント を使用して IPv6 `braket.{region}.api.aws` 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
NVIDIA's CUDA-Q Amazon Braket での 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 デバイスはすぐに使用可能	IonQ Forte-1 デバイスは予約専用ではなくなり、お客様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削除しました。	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削除しました。	2024 年 8 月 29 日
新しいデバイス Rigetti Ankaa-2	Rigetti Ankaa-2 デバイスのサポートが追加されました。スケーラブルなマルチチップテクノロジーを利用する 84 quibit デバイス。	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、実行カスタマージャーニーを取り、Amazon Braket を使用してこのパスに沿ってユーザーをガイドします。	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削除しました。	2024 年 6 月 28 日
新しいデバイスIQM Garnetとリージョン Europe North 1	IQM ガーネットデバイスのサポートが追加されました。正方形の格子トポロジーを持つ 20 量子ビットデバイス。Braket がサポートするリージョンを欧州北部 1 (ストックホルム) に拡張しました。	2024 年 5 月 22 日
ローカル調整のリリース	実験機能には、QuEra の Aquila QPU のローカル調整機能が含まれるようになりました。	2024 年 4 月 11 日
ノートブック非アクティブマネージャーのリリース	ノートブックインスタンスを作成するときは、非アクティブマネージャーを有効にし、アイドル時間を設定して Braket ノートブックインスタンスを自動的にリセットします。	2024 年 3 月 27 日
目次の再作業	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ためのコンテンツの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 ダイレクトリリース	Braket direct 機能のサポートが追加されました。 • 予約の操作 • エキスパートのアドバイスを受ける • 実験機能を調べる	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 の更新	ドキュメントを更新して、新規および既存の Amazon Braket のお客様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しました。	2023 年 11 月 27 日
独自のコンテナ の更新	ドキュメントを更新して、BYOC のタイミング、BYOC のレシピ、コンテナでの Braket Hybrid Jobs の実行に関する情報を追加しました。	2023 年 10 月 18 日

ハイブリッドジョブデコレータのリリース	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。例を示します。 • 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データを保存してロードする • ハイブリッドジョブデコレータの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	開発者ガイドのドキュメントを更新して、 queue depth と を追加しましたqueue position。 キューを可視化するための新しい API の変更を反映するように API ドキュメンテーションを更新しました。	2023 年 9 月 25 日
ドキュメントで命名を標準化する	ドキュメントを更新して、「ジョブ」のインスタンスを「ハイブリッドジョブ」に変更し、「タスク」を「量子タスク」に変更しました。	2023 年 9 月 11 日
新しいデバイス IonQ Aria 2	IonQ Aria 2 デバイスのサポートを追加	2023 年 9 月 8 日

ネイティブゲートの更新	ドキュメントを更新して、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 出発	ドキュメントを更新してすべてのXanaduデバイスを削除	2023 年 6 月 2 日
新しいデバイス IonQ Aria	IonQ Aria デバイスのサポートを追加	2023 年 5 月 16 日
リタイアしたRigettiデバイス	のサポートを終了しました Rigetti Aspen-M-2	2023 年 5 月 2 日
AmazonBraketFullAccess ポリシー情報を更新しました	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、クォータに関する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の起動	Braket オンボーディングのより最新のシンプルな方法を反映するようにドキュメントを変更しました。	2023 年 4 月 5 日
新しいデバイス Rigetti Aspen-M-3	Rigetti Aspen-M-3 デバイスのサポートを追加	2023 年 1 月 17 日
新しい結合勾配機能	が提供する結合勾配機能に関する情報を追加しました SV1	2022 年 12 月 7 日

新しいアルゴリズムライブラリ機能	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 出発	すべてのD-Waveデバイスの削除に対応するためにドキュメントを更新しました	2022 年 11 月 17 日
新しいデバイス QuEra Aquila	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 のサポート	Braket Pulse のサポートが追加され、 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 ネイティブゲートのサポート	IonQ デバイスが提供するネイティブゲートセットのサポートを追加	2022 年 9 月 13 日
新しいインスタンスクォータ	Hybrid Jobs に関連付けられたデフォルトの従来の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
新しいサービスダッシュボード	サービスダッシュボードを含むようにコンソールのスクリーンショットを更新	2022 年 8 月 17 日
新しいデバイス Rigetti Aspen-M-2	Rigetti Aspen-M-2 デバイスのサポートを追加	2022 年 8 月 12 日
OpenQASM の新機能	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	2022 年 8 月 4 日

新しいコスト追跡手順	シミュレーターとハードウェアワークロードの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	2022 年 7 月 18 日
新しいXanadu Borealisデバイス	Xanadu Borealis デバイスのサポートを追加	2022 年 6 月 2 日
新しいオンボーディングの簡素化手順	新しく簡素化された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
新しいデバイス D-Wave Advantage_system6.1	D-Wave Advantage_system6.1 デバイスのサポートが追加されました	2022 年 5 月 12 日
埋め込みシミュレーターのサポート	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	s3:ListAllMyBuckets アクセス許可が追加され、ユーザーが Amazon Braket 用に作成および使用されたバケットを表示および検査できるようになりました。	2022 年 3 月 31 日
OpenQASM のサポート	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	2022 年 3 月 7 日
新しい量子ハードウェアプロバイダーOxford Quantum Circuits、新しいリージョン、eu-west-2	OQC と eu-west-2 のサポートを追加	2022 年 2 月 28 日

新しいRigettiデバイス	Rigetti Aspen M-1 のサポートの追加	2022 年 2 月 15 日
新しいリソース制限	同時タスクDM1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
新しいRigettiデバイス	Rigetti Aspen-11 のサポートの追加	2021 年 12 月 20 日
リタイアしたRigettiデバイス	Rigetti Aspen-10 デバイス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	ローカル密度行列シミュレーターとDM1デバイスでサポートされる低密度行列の結果タイプ	2021 年 12 月 20 日
更新されたポリシーの説明	Amazon Braket は、ロール ARN を更新して`servicerole/` パスを含めました。ポリシーの更新の詳細については、Amazon Braket マネージドポリシーの更新 AWS」表を参照してください。	2021 年 11 月 29 日
Amazon Braket Jobs	Amazon Braket Hybrid Jobs のユーザーガイドAPIを追加	2021 年 11 月 29 日
新しいRigettiデバイス	Rigetti Aspen-10 のサポートの追加	2021 年 11 月 20 日
リタイアしたD-Waveデバイス	D-Wave QPU のサポートを終了しました。 Advantage_system1	2021 年 11 月 4 日
新しいD-Waveデバイス	追加の D-Wave QPU のサポートが追加されました。Advantage_system4	2021 年 10 月 5 日

新しいノイズシミュレーター	最大 17 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) qubits 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	2021 年 5 月 25 日
PennyLane のサポート	Amazon Braket で PennyLane のサポートを追加	2020 年 12 月 8 日
新しいシミュレーター	より大きな回路を可能にする Tensor Network Simulator (TN1) のサポートを追加	2020 年 12 月 8 日
タスクバッチ処理	Braket はカスタマータスクのバッチ処理をサポート	2020 年 11 月 24 日
手動qubit割り当て	Braket がRigettiデバイスの手動qubit割り当てをサポート	2020 年 11 月 24 日
調整可能なクォータ	Braket は、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します。	2020 年 10 月 30 日
PrivateLink のサポート	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ます。	2020 年 10 月 30 日
タグのサポート	Braket が量子タスクリソースの APIベースのタグをサポート	2020 年 10 月 30 日
新しいD-Waveデバイス	追加の D-Wave QPU のサポートが追加されました。Advantage_system1	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	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。