Intel® Quantum SDK

Developer Guide and Reference

February 15, 2024

Release Version 1.1

Contents:

1.0 How to Cite

To cite the Intel® Quantum SDK, please reference:

Khalate, P., Wu, X.-C., Premaratne, S., Hogaboam, J., Holmes, A., Schmitz, A., Guerreschi, G. G., Zou, X. & Matsuura, A. Y., arXiv:2202.11142 (2022).

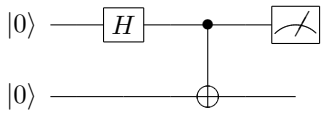
2.0 Overview

The Intel® Quantum Software Development Kit (SDK) is a high level programming environment that allows users to write software targeted to the Intel® quantum hardware. The Intel® Quantum SDK currently provides a choice of several simulation backends specialized for different tasks. When Intel® quantum hardware backends are available in the future, users will be able to seamlessly transition their simulations to execute on physical qubits with minimal software changes.

Developing applications that run on **quantum** computers involves considerable challenges whose solutions we often take for granted when programming the **classical** computers we use every day. The Intel® Quantum Computing Stack encapsulates these challenges as internal modules that include: quantum compilation (front-end and back-end), runtime mapping and scheduling, fault tolerance support, control electronics, and qubit management. The Intel® Quantum SDK is designed to fully integrate with these modules of the Intel® Quantum Computing Stack. It includes optimizations and decompositions based on the LLVM compiler framework targeting the Intel® Quantum Computing Stack.

The Intel® Quantum SDK provides an intuitive C++ based application programming interface (API). This API allows users to express quantum circuit diagrams using C++ code. At this point, readers new to quantum computing and interpreting quantum circuit diagrams may benefit from visiting the **Introduction to Quantum Computing** section and the collection of Tutorials.

Let's consider a simple example. The following quantum circuit, which represents the famous entangled EPR or Bell State [EIPR1935] [BELL1964],



is expressed with the Intel® Quantum SDK using the following C++ code:

Listing 1: Bell State Preparation & Measurement Example

```
/* Gate definitions and key words */
#include <clang/Quantum/quintrinsics.h>

/* Quantum Runtime APIs */
#include <quantum_full_state_simulator_backend.h>

#include <iostream>

const int num_qubits = 2;
/* Declare 2 qubits */
qbit q[num_qubits];

/* The quantum logic must be in a function with the keyword quantum_kernel */
/* pre-pended to the signature */
quantum_kernel void prep_and_meas_bell(cbit read_out) {
    /* Prepare both qubits in the |0> state */
    PrepZ(q[0]);
    PrepZ(q[1]);

    /* Apply a Hadamard gate to the top qubit */
    H(q[0]);

    /* Apply a Controlled-NOT gate with the top qubit as
     * the control and the bottom qubit as the target */
    CNOT(q[0], q[1]);

    /* Measure qubit 0 */
    MeasZ(q[0], read_out);
}

int main() {
    /* Configure the simulator */
    iqsdk::IqsConfig settings(num_qubits, "noiseless");
    iqsdk::FullStateSimulator quantum_8086(settings);
    if (iqsdk::QRT_ERROR_SUCCESS != quantum_8086.ready()) return 1;

    /* Declare 2 measurement readouts */
    /* Measurements are stored here as "classical bits" */
    cbit classical_bit;

    prep_and_meas_bell(classical_bit);

    /* Here we can use the FullStateSimulator APIs to get data */
    /* or we can write classical logic that interacts with our measurement */
    /* results, as below. */
    bool result = classical_bit;
    if (result) {
        std::cout << "True\n";
    }
    else {
        std::cout << "False\n";
    }

    return 0;
}
```

Ready to get started building quantum circuits? If so, feel free to jump straight to the Getting Started Guide to learn about the SDK's software requirements, installation, usage, and how to interpret the output. Otherwise, it may be helpful to brush up on the basics and investigate the resource material found in the **Introduction to Quantum Computing** section. The collection of Tutorials and **Samples** may also be of interest.

To summarize, the Intel® Quantum SDK includes:

- An intuitive user interface based on the C++ programming language.
- Optimizations and decompositions based on the LLVM compiler framework specifically targeted at the Intel® Quantum Computing Stack.
- A full compilation flow that produces an executable using a user's selected backend.

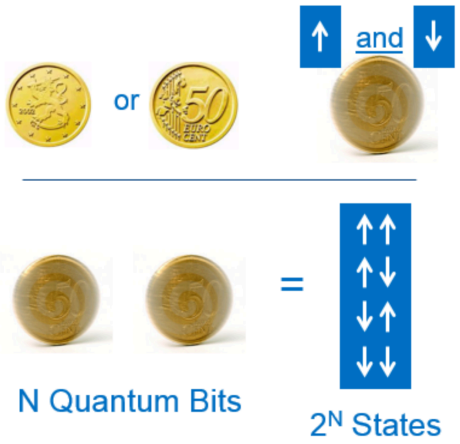
Authoring a quantum-accelerated application in the Intel® Quantum SDK follows the programming paradigm of other hardware accelerators. Quantum programs are written in a C++ programming environment that has been extended to allow the user to express quantum circuits as quantum logical operations. Depending on whether the user is targeting a simulation environment or qubit hardware, our quantum runtime library will direct the quantum workload to the appropriate backend during runtime execution.

Currently available backends:

- The **Intel Quantum Simulator (IQS) backend**, a full-state-vector qubit simulation with a complete description of the quantum state of the qubits defined.
- The **Quantum Dot (QD) simulator**, a physics-based simulation of the physical qubits paired with a state-vector frontend.
- The **Tensor Network backend**, a simulator that uses tensor contractions to evaluate quantum circuits.
- The **Clifford Circuit backend**, a simulator that provides extremely fast results for quantum algorithms implemented using only Clifford quantum gates.
- An interface for a **user-defined backend**; with this option, users can develop their own behavior for qubit simulators.

3.0 Introduction to Quantum Computing

Quantum computing is a new model of computation that solves problems by manipulating and measuring the properties of special systems that exhibit quantum mechanical phenomena. These special quantum mechanical systems are referred to as **quantum computers**. Quantum computers are particularly well-suited to certain kinds of computational problems, such as cryptography, Quantum Fourier Transforms (QFTs), optimization/search, physics/chemistry simulation, and many more.



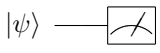
To better understand how a quantum computer works, it helps to compare the basic unit of quantum information, a **qubit**, with the well-known classical binary bit. A binary bit can be in only one of two possible states: a 0 or a 1. This is similar to how we consider the state of a standard coin lying on a table; it is **either** heads **or** tails.

Note: Here we focus on the paradigm of gate-based quantum computing using qubits. Other types of quantum systems are not directly supported by the Intel® Quantum SDK.

In this analogy we consider the state of a **quantum** coin to be "spinning" on the table top; that is, it is in neither the heads nor the tails state, it is in **both** states at the same time. In the quantum realm, this concept is called **superposition**: a qubit is simultaneously in both the 0 and the 1 state (in quantum mechanics, these are referred to as $|0\rangle$, $|1\rangle$). In fact, a qubit is in a linear combination of these two states. So in some sense, a qubit is more like a **weighted** spinning coin; it has some chance of being in the 0 state, and some chance of being in the 1 state. In quantum mechanics we often write this scenario as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

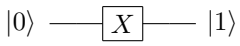
where $|\psi\rangle$ represents the state of the qubit and α and β represent the probability "amplitude" of the qubit being in the $|0\rangle$ or the $|1\rangle$ state, respectively. Actually, the α and β coefficients are complex numbers, and the square of their absolute values (as in, $|\alpha|^2$ and $|\beta|^2$) represents the real-world probability of that qubit being in the $|0\rangle$ or the $|1\rangle$ states, respectively. Since there are only 2 possible states, we know that $|\alpha|^2 + |\beta|^2 = 1$, which simply means there is a 100% chance that the qubit is in either the $|0\rangle$ or the $|1\rangle$ state. Just as we can stop a spinning coin at any time with our finger, we can also measure the state of a qubit $|\psi\rangle$ to see exactly which state it is in at a given time: we represent this measurement with a very simple **quantum circuit diagram**:



Note: This quantum circuit diagram represents measuring an arbitrary single-qubit state $|\psi\rangle$. Quantum circuit diagrams are read left-to-right. More operations on a single qubit extend the circuit horizontally, and more qubits are added vertically.

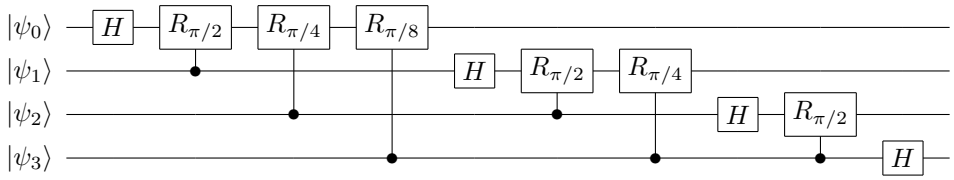
Immediately upon measuring the qubit's state, we will see that is it in either the $|0\rangle$ state or the $|1\rangle$ state. The probability the qubit is $|0\rangle$ is $|\alpha|^2$, and the probability the qubit is $|1\rangle$ is $|\beta|^2$. An important property of quantum measurements is that they change the state of the underlying qubits, irreversibly **collapsing** the qubit's state from a superposition to a single state.

Applying operations like measurements or other quantum **gates** (such as a qubit "flip") is fundamental to quantum computing. For example, we can flip the $|0\rangle$ state to the $|1\rangle$ state with an X gate, as represented by this circuit diagram:



Furthermore, qubits can be **entangled** with each other. That is, the combined state of multiple qubits can be correlated. In our coin analogy above, 2 spinning coins could represent 4 possible states via superposition. But if the two spinning coins are entangled, the result of one coin will necessarily inform us of the result of the other. In the similar case of measuring an entangled Bell pair of qubits, measuring the state of one qubit lets us know the state of the second qubit, even without measuring the second qubit.

These two properties, **superposition** and **entanglement**, enable quantum computers to solve certain problems far more efficiently than a classical computer. Generally speaking, quantum computing is performing quantum operations on qubits to solve these interesting problems. One example is applying a Quantum Fourier Transform (QFT); with only $O(n^2)$ gates, specifically the Hadamard and phase-shift gates (see the **quantum gates** and the **QFT sample** sections for more details), we can apply a Fourier transform on $O(2^n)$ amplitudes. The corresponding 4-qubit QFT circuit diagram looks like this:



To learn more about quantum computing and how to develop quantum algorithms like the one above, see suggestions in the Getting Started Guide.

4.0 Supported Quantum Logic Gates

Below is a list of quantum logic gates supported in the Intel® Quantum SDK and their signatures. To see the matrix definitions for these gates, refer to the file:

```
<path to Intel Quantum SDK>/iqc/include/clang/Quantum/quintrinsics.h
```

1. Hadamard (H)

   ```
   void H(qbit& q);
   ```

2. Pauli X (X)

   ```
   void X(qbit& q);
   ```

   This is equivalent to a rotation around the X-axis by π.

3. Pauli Y (Y)

   ```
   void Y(qbit& q);
   ```

   This is equivalent to a rotation around the Y-axis by π.

4. Pauli Z (Z)

   ```
   void Z(qbit& q);
   ```

   This is equivalent to a rotation around the Z-axis by π.

5. Phase (S)

   ```
   void S(qbit& q);
   ```

   Phase shift with half the rotation of Z (i.e. a $\pi/2$ rotation). Equivalent to $RZ(\pi/2)$.

6. Phase Inverse (Sdag)

   ```
   void Sdag(qbit& q);
   ```

   Conjugate transpose of S.

Note: Dag here is an abbreviation for "dagger", which denotes the conjugate transpose of a Hermitian matrix.

---

7. T

   ```
   void T(qbit& q);
   ```

   Phase shift, with one quarter the rotation of Z (i.e. a $\pi/4$ rotation). Equivalent to $RZ(\pi/4)$.

8. T Inverse (Tdag)

   ```
   void Tdag(qbit& q);
   ```

   Conjugate transpose of T.

9. X axis Rotation (RX)

   ```
   void RX(qbit& q, double angle);
   ```

10. Y axis Rotation (RY)

```
void RY(qbit& q, double angle);
```

11. Z axis rotation (RZ)

    ```
    void RZ(qbit& q, double angle);
    ```

12. Controlled Z (CZ)

    ```
    void CZ(qbit& ctrl, qbit& target);
    ```

13. CNOT

    ```
    void CNOT(qbit& ctrl, qbit& target);
    ```

14. SWAP

    ```
    void SWAP(qbit& ctrl, qbit& target);
    ```

15. Toffoli

    ```
    void Toffoli(qbit& ctrl0, qbit& ctrl1, qbit& tgt);
    ```

    Toffoli gate with two controls.

16. PrepZ

    ```
    void PrepZ(qbit& q);
    ```

    Initialize/reset qubit to the $|0\rangle$ computational state.

17. PrepX

    ```
    void PrepX(qbit& q);
    ```

    Initialize/reset qubit to the $|+\rangle$ computational state.

18. PrepY

    ```
    void PrepY(qbit& q);
    ```

    Initialize/reset qubit to the $|R\rangle$ computational state.

19. MeasZ

    ```
    void MeasZ(qbit& q, bool& c);
    void MeasZ(qbit& q, cbit& c);
    ```

    Measure the qubit q in the $|0\rangle$ or $|1\rangle$ computational states and store the result in c.

20. MeasX

    ```
    void MeasX(qbit& q, bool& c);
    void MeasX(qbit& q, cbit& c);
    ```

    Measure the qubit q in the $|+\rangle$ or $|-\rangle$ computational states and store the result in c.

21. MeasY

    ```
    void MeasY(qbit& q, bool& c);
    void MeasY(qbit& q, cbit& c);
    ```

    Measure the qubit q in the $|R\rangle$ or $|L\rangle$ computational states and store the result in c.

22. CPhase

```
void CPhase(qbit& ctrl, qbit& target, double angle);
```

Controlled Phase gate.

23. XY-plane Rotation

```
void RXY(qbit& q, double theta, double phi);
```

Define a rotation in the XY-plane of the Bloch sphere (RXY Matrix Representation).

24. Swap Alpha

```
void SwapA(qbit& q1, qbit& q2, double angle);
```

Rotation in the Span$\{|01\rangle\ |10\rangle\}$ subspace.

4.1 Quantum Dot Qubit Gates

Some physical systems will find it easier to implement certain quantum gates because of differences in the underlying quantum systems used to create hardware qubits. If two sets of quantum gates are each universal for quantum computing, then a quantum algorithm can be implemented in either set of quantum gates. Below is the list of the quantum gates that the Intel® Quantum SDK targets during compilation. The gates written in the `quantum_kernel` functions are decomposed by the compiler into the gates below, and the results can be found in the human-readable `.qs` file. This list is for reference.

1. `quprepz(qbit q)`

   An incoherent reset to computational $|0\rangle$ state.

2. `qumeasz(qbit q)`

   Measurement in the Z basis. This collapses the qubit to the measured state, either $|0\rangle$ or $|1\rangle$.

3. `qurotxy (qbit q, double theta, double phi)`

   A rotation of `theta` around arbitrary axis in X-Y plane of the Bloch sphere as characterized by angle `phi`, i.e. the operator exp $\left\{-i\,\theta/2\left(\cos(\phi)\hat{\mathrm{X}}+\sin(\phi)\hat{\mathrm{Y}}\right)\right\}$.

4. `qucphase(qbit q1, qbit q2, double theta)`

   An arbitrary phase of exp $(-i\,\theta)$ on the $|11\rangle$ state of the given qubits.

5. `quswapalp(qbit q1, qbit q2, double theta)`

   An arbitrary rotation of `theta` in the $\{|01\rangle\,,|10\rangle\}$ state subspace.

6. `qurotz(qbit q, double theta)`

   An arbitrary rotation of `theta` about the Z-axis of the Bloch sphere. Optimizers should minimize occurrences.

5.0 Language Extensions

The Intel® Quantum SDK defines a number of data types, keywords, and classes to facilitate expressing quantum algorithms as well as some common tasks associated with working with quantum qubit simulators. A complete list of the methods is provided in API Reference. This section summarizes the key concepts developers should know when writing C++-based programs in the Intel® Quantum SDK.

5.1 Built-in Types & Intrinsic Functions

`qbit`:

> Data type for variables representing qubits. `qbit` variables can be declared either globally in the global namespace or locally within a `quantum_kernel` function (see **Local variables**). A `qbit` variable cannot be used as a member variable of any class.

`cbit`:

> Data type for variables to represent a classical bit returned by a quantum measurement. Equivalent to `bool`.

`quantum_kernel`:

> Attribute for a function that will contain quantum logic, e.g. a gate acting on a qubit or another `quantum_kernel`.

`release_quantum_state()`:

> An intrinsic function that once invoked in a `quantum_kernel`, indicates that the quantum state is unconstrained from that point onwards. Quantum variables can be re-used in a new `quantum_kernel`, but they must be re-initialized using `PrepX`, `PrepY`, or `PrepZ` since the quantum states are unspecified after being released. Calling `release_quantum_state()` facilitates optimizations when compiling with the `-O1` flag, which can lead to more-efficient execution of the quantum algorithm. For an example of the effects of optimization on a `quantum_kernel` using `release_quantum_state()`, see Using release_quantum_state() in the Tutorial document.

`std::vector<std::reference_wrapper<qbit>>`

> Data structure used to specify the order of qubits or a subset of qubits for reporting data from a full state quantum simulator. Reference wrappers are constructed by calling `std::ref(q)` for qubit variables `q`. This idiom is important because there is no a priori relationship between qbit type variables and hardware qubits. Conceptually, this is similar to an object instance that does not necessarily occupy the same memory address during each execution.
>
> For example, suppose a user has two qubits: `q1` in state $|0\rangle$ and `q2` in state $|1\rangle$. If `getProbabilities` is invoked with the vector `{std::ref(q1), std::ref(q2)}` of reference wrappers, the reported state will be $|01\rangle$; if it is invoked with `{std::ref(q2), std::ref(q1)}`, the state will be: $|10\rangle$.

5.1.1 Known Limitations

Top-level `quantum_kernel` functions can only refer to global `qbit` variables or local `qbit` variables defined inside a `quantum_kernel` function. In other words, the following is not a valid top-level `quantum_kernel` function signature: `quantum_kernel void my_single_qubit_function(qbit &q);`. This restriction does not apply to quantum kernel expressions, which are described in FLEQ Guide and Reference (Local qubits).

5.2 Namespaces

`iqsdk`

Namespace providing access to the classes and methods of the Intel® Quantum SDK.

`qexpr`

Namespace providing access to quantum kernel expressions. Part of the Functional Language Extension for Quantum (FLEQ). See FLEQ Guide and Reference (Quantum kernel expressions).

`qlist`

Namespace providing access to compile-time quantum lists (see FLEQ Guide and Reference (`QList`)).

`datalist`

Namespace providing access to compile-time strings (see FLEQ Guide and Reference (`DataList`)).

5.3 Includes & Classes

`<clang/Quantum/quintrinsics.h>`**:** Required header file that provides access to the supported quantum gates as well as the instructions to prepare the state of a qubit and perform a measurement on a qubit. See **Supported Quantum Logic Gates** and **Intel Quantum Dot Qubit Gates** for additional details about the gates.

5.3.1 Quantum Backends

`<quantum_full_state_simulator_backend.h>`:

This header file is needed for full state simulators, and provides access to the `FullStateSimulator` class as well as auxiliary helper classes.

`IqsConfig`:

Configuration data for the `FullStateSimulator` class.

`FullStateSimulator`:

Class with API calls to both set up a quantum simulator device and access the underlying quantum state during simulation. See **Configuring the FullStateSimulator** for a quick explanation or API Reference for a complete description of the class's methods.

`<quantum_clifford_simulator_backend.h>`:

Header file needed for using the Clifford Simulator.

`ErrSpec1Q`:

Configuration struct for single qubit gate errors.

`ErrSpec2Q`:

Configuration struct for two qubit gate errors.

`ErrSpecIdle`:

Configuration struct for idling.

`ErrorRates`:

Configuration struct for error rates of each gate type.

`GateTimes`:

Configuration struct for duration of each gate type.

`CliffordSimulatorConfig`:

Configuration struct for setting up a Clifford Simulator Device.

`CliffordSimulator`:

Backend interface for using the Clifford Simulator.

`<quantum_tensor_network_backend.h>`:

Header file needed for using the Tensor Network Backend.

`TensorNetworkConfig`:

Configuration struct for setting up a Tensor Network Device.

`TensorNetworkSimulator`:

Backend interface for using Tensor Network Simulation.

`<quantum_custom_backend.h>`:

Header file needed for using or developing a Custom Backend.

`CustomInterface`:

Abstract base class for user to implement their own simulator.

`CustomSimulator`:

Backend interface for using the custom backend.

`<qrt_errors.hpp>`:

This header file is included by any quantum backend. It defines the data type for communicating success or failure from the quantum runtime.

`QRT_ERROR_T`:

Data type representing potential errors in setting up a quantum device. Either `QRT_ERROR_SUCCESS`, `QRT_ERROR_WARNING`, or `QRT_ERROR_FAIL`.

`<quantum_backend.h>`:

This header file contains base classes for simulation devices. It is included by any needed simulation interface.

`<quantum.hpp>`:

Deprecated. Includes headers for all backends.

5.3.2 Accessing Results

`<qrt_indexing.hpp>`:

This header file is included by any quantum backend. It defines the data types for accessing backend results.

`QssIndex`:

Data type for representing quantum basis elements. Used for indexing into data structures representing quantum state spaces (QSS).

`QssMap<T>`:

A map from `QssIndex` values to type T values. Used for representing total or partial quantum state spaces where T is `double` for probabilities or `complex` for amplitudes.

5.3.3 Functional Language Extension for Quantum (FLEQ)

`<clang/Quantum/qexpr.h>`:

Header file that provides resources for building quantum kernel expressions. See FLEQ Guide and Reference (Quantum kernel expressions).

`QExpr`:

Data type of quantum kernel expressions, a representation of quantum kernel functions provided by FLEQ. See FLEQ Guide and Reference (Quantum kernel expressions) for more information.

`<qlist-utils.h>`:

Header file that provides useful utilities for working with quantum kernel expressions.

`<clang/Quantum/qlist.h>`:

Header file that provides access to compile-time qubit lists. See FLEQ Guide and Reference (`QList`).

`qlist::QList`:

Data type for compile-time qubit lists.

`<clang/Quantum/datalist.h>`:

Header file that provides access to compile-time strings. See FLEQ Guide and Reference (`DataList`).

`datalist::DataList`:

Data type for compile-time strings.

6.0 Programming with the Intel® Quantum SDK

6.1 In-lining & `quantum_kernel` functions

When the compiler prepares a `quantum_kernel` function, it separates all the quantum instructions (as Intermediate Representation (IR)) from the classical IR so that it can deliver a complete set of instructions to the quantum backend.

Local declarations and operations with traditional C++ data types **are** supported inside a `quantum_kernel` function, which aids readability and preserves programming concepts. At compile time, these "classical" instructions are pulled out of the `quantum_kernel`. This has a consequence on classical instructions, especially `bool` and `cbit` measurement results: any operations on classical variables written inside a `quantum_kernel` will be executed at the beginning of that `quantum_kernel`, unless they are written after the final quantum gate in the `quantum_kernel`.

```
qbit q0;
qbit q1;

quantum_kernel void myKernel() {
  bool b = false;
  std::cout << "b has value false (0) here after initialization: "
            << b << "\n";
  PrepZ(q0);
  X(q0);
  MeasZ(q0, b);
  std::cout << "b still has value 0 here since the quantum gates are not complete: "
            << (int)c << "\n";
  PrepZ(q1);
  std::cout << "After all gates in quantum_kernel have executed, b has value true (1): "
            << b << "\n";
}
```

A `quantum_kernel` may be called from within another `quantum_kernel`. Here, too, the compiler in-lines the quantum instructions from the innermost `quantum_kernel` and continues until it produces one sequence of instructions that corresponds to the "top-level" `quantum_kernel` call that begins the quantum algorithm.

In-lining combined with the earlier rule on rearranging operations on measurement results means that for `quantum_kernel` functions containing a measurement which are called in the middle of another `quantum_kernel` function, the operations on those `cbit` and `bool` measurement results will be moved to the beginning of the resulting set of instructions. This means that the following code:

If a user needs classical instructions to be executed strictly in the middle of a quantum algorithm, they should break up the algorithm into multiple top-level quantum kernel functions. Alternatively, they can use the `bind` operator on quantum kernel expressions (see FLEQ Guide and Reference (Barriers and binding)).

The restriction that the entire `quantum_kernel` be known at compile time together with the in-lining behavior means that the top-level kernel cannot accept an arbitrary variable of type `qbit` as a parameter. The variables of `qbit` type that will be operated on must be explicitly defined in the "top-level" kernel's instructions; however, inner `quantum_kernel` functions may be written to accept `qbit` type variables as parameters.

Note: This restriction applies primarily to `quantum_kernel` functions, and **not** to FLEQ. See FLEQ Guide and Reference if you need this feature.

```
qbit qs[3];

// A nested quantum_kernel may take either classical or quantum arguments
quantum_kernel void bell(qbit &a, qbit &b) {
  PrepZ(a);
  PrepZ(b);
  H(a);
  CNOT(a,b);
}

// A top level quantum_kernel may take classical arguments, but not quantum
// arguments
quantum_kernel void topLevelBell() {
  bell(q[0],q[2]);
}

int main() {

  // may call top-level quantum_kernel
  topLevelBell();

  // may not call quantum_kernel with quantum arguments
  // invalid: bell(q[0], q[2]);
}
```

6.2 Measurements using Simulated Quantum Backends

A typical quantum program using the Intel® Quantum SDK will do effectively the following sequence: 1. Submit `quantum_kernel` functions to a quantum backend. 2. Execute `quantum_kernel` on the backend. 3. Retrieve results. 4. Repeat 1-3 as needed.

After the `quantum_kernel` has finished executing, users will need to retrieve results from the backend. This section describes the result retrieval and aggregation process, using the example of the `FullStateSimulator` backend.

The `FullStateSimulator` class provides three main approaches to obtain statistical measurements:

1. `getProbabilities()` (and/or other simulation data)
2. `getSamples()`
3. Repeated execution of explicit measurement operations e.g. `MeasZ` (sampling).

These methods will be elaborated in the following sections.

Both Intel® Quantum Simulator (IQS) and Quantum Dot (QD) Simulator backends support collecting the simulation details, such as the quantum amplitudes, conditional probabilities, or single-qubit probabilities. The `FullStateSimulator` class provides these data regardless of which backend is selected to run the simulation.

6.2.1 Simulation Data

Table 1: Simulation Method Comparison

Method	Returned object	Efficiency (with IQS)	Recommended?	Other Notes
`getProbabilities()`	`vector<double>` or `QssMap<double>`	Best	Yes	
`getSingleQubitProbs()`	`vector<double>`	Best	Yes	
`getSamples()`	`vector<vector<bool>>`	Good	Yes	
`getAmplitudes()`	`vector<complex<double>>` or `QssMap<complex<double>>`	Good	No	Accurate up to global phase
Repeated sampling calls	User-defined	Worst	No	Complexity scales with number of samples

Working with the simulation data returned by `FullStateSimulator` methods such as `getProbabilities()` is often the most computationally efficient route to simulating a quantum algorithm. This is because quantum algorithms often encode their results as probabilities of different states. If the entire algorithm needed to run many times to sample the probability, as required on a hardware quantum backend, the simulation time would increase significantly.

For applications that need a set of measurement outcomes, both backends of the `FullStateSimulator` offer a second route to obtain the simulation data, which avoids the need for repeated executions of a given `quantum_kernel` function. This route consists of calling `getSamples()` to get sequences of outcomes as if measurements were applied to the qubit register. This sampling of results doesn't affect the state and can even be applied as many times as an application calls for.

6.2.2 Combining Simulation Data and Measurement Operations

IQS offers the ability to retrieve simulation results (i.e. from `getProbabilities()` or `getSamples()`) when `quantum_kernel` functions include measurement gates (e.g. `MeasZ()`).

Note: This feature is not available in QD Simulator because it doesn't collapse the state (see the Quantum Dot (QD) Simulator). This means combining results of measurement operations and sampling results with the QD Simulator can yield unexpected results.

When using probability measurement **and** explicit measurement gates on a qubit in simulations, IQS will cause a 'partial collapse' of the state in the simulator to a sub-space. You can combine such operations with a sampling technique like `getProbability` or `getSamples` to compute data or collect statistics on the sub-space. To support combining measurement operations and simulation data, IQS will always collapse the quantum state of the simulator when it encounters a measurement operation in a `quantum_kernel`. Any subsequent querying of the `FullStateSimulator` after measurement will always give the same result on the qubits that had one of `MeasX`, `MeasY`, or `MeasZ` applied, and other qubits will have any correlated effects on their probabilities present.

Measuring a qubit leaves it in one of the two states into which the measurement was projected; e.g. measuring a qubit along the Z-axis (in a Bloch sphere representation) leaves it in either a $|0\rangle$ or $|1\rangle$ state. Another perspective on this is that the post-measurement state of the entire set of qubits now occupies a sub-space of the Hilbert space previously occupied by the pre-measurement qubits. This can be qualitatively understood by noting that there is no uncertainty in the state of the measured qubit. A measurement also has consequences on the correlations arising from entanglement between qubits. More simply, measuring one qubit can affect the probabilities of the outcomes of measuring a different qubit (provided the two qubits were entangled). In the extreme case, a large amount of correlation present in the system could mean that a single measurement applied on one qubit results in the state of the entire set of qubits being determined, such as for a Bell pair or GHZ state.

6.2.3 Using Only Measurement Operations

A third option is to collect your own statistical results by executing the entire quantum algorithm with all the required measurement operations many times in a loop (or other control-flow structure) to direct execution flow. Each iteration of the quantum algorithm produces and then stores, analyzes, or accumulates the result of the measurements. Under ideal conditions (no noise), the sampling & measurement approaches will each produce statistically-equivalent results, especially with large sample sizes. Because quantum algorithms running on quantum hardware must use the measurement approach, the simulation data and sampling approaches can be seen as a debugging mode for the measurement approach. IQS supports using measurements anywhere in the quantum algorithm; in contrast, QD Simulator only supports reading measurements at the end of the `quantum_kernel`.

6.3 Local `qbit` Variables

`qbit` variables can be declared globally or locally. When the compiler maps the program qubits to physical qubits, each `qbit` variable will be assigned to a physical qubit. Since the compiler cannot guarantee the state that a local `qbit` variable is in, local `qbit` variables must be initialized using `PrepX`, `PrepY`, or `PrepZ` before being used. At the end of the `quantum_kernel`, the local `qbit` variables must be released. This can be achieved through measurements or `release_quantum_state()`.

Note that if using `release_quantum_state()`, the quantum states are unspecified after the function call (see **Language Extensions**). Without releasing the quantum states, the physical qubits assigned to the local `qbit` variables might be assigned to other local `qbit` variables in a new `quantum_kernel` function while still holding the quantum states of the out-of-scope variables. The out-of-scope variables' physical qubits will not be assigned to unreleased global `qbit` variables, however.

In the following example, a local `qbit` variable is declared, initialized, and measured.

```
quantum_kernel void kernel() {
   qbit q;
   bool b;  // can also be of type cbit

   PrepZ(q);    // prepare the qbit variable before applying gates
   H(q);

   MeasZ(q, b);  // release the qbit variable at the end of the quantum_kernel
}
```

If local `qbit` variables are entangled with global `qbit` variables, the entanglement persists after the local `qbit` variables go out of scope. The user must insert gates needed to disentangle the local `qbit` variables from the global ones before releasing the local variables' quantum states.

```
qbit global;

quantum_kernel void errorExampleEntangledQubits() {
   qbit local;

   PrepZ(local);    // Prep the qbit variable before applying gates
   H(local);
   CNOT(local, global);

   // After local goes out of scope, the physical qubit it was assigned to
   // is still entangled with global
}
```

The recommended best practice with regards to local `qbit` variables is therefore to prep them before they are used and insert gates to undo the entanglement between local and global `qbit` variables before releasing the quantum states at the end of `quantum_kernel` functions.

For information on how to use local `qbit` variables with quantum kernel expressions and FLEQ, refer to FLEQ Guide and Reference (Local qubits).

7.0 Compiling

The compiler's operation can be modified using command-line flags, allowing functionality such as specifying header include paths and library paths, redirecting output files, and specifying different qubit hardware or connectivity. These options can be printed by running

```
$ ./intel-quantum-compiler -h
```

7.1 Output of the Intel® Quantum SDK Compiler

Three files are generated from the compilation stage and written to the working or user-specified output directory. These files are:

`<algo-name>.ll`: Intermediate representation (IR) of source file.

> This file shows the LLVM IR of both quantum and classical parts of the code combined, with the `quantum_kernels` and operations represented as function calls.

`<algo-name>.qs`: Human-readable assembly file for Intel® Quantum backend target.

> This file shows the assembly for each `quantum_kernel` written by the user and mapped to the quantum backend. Thus, it will reflect some of the quantum target's attributes, such as its native gate set, and limited connectivity (in the form of additional swap gates).

`<algo-name>`: Executable corresponding to `<algo-name>`.

> This is the binary and final result of the Intel® Quantum SDK compiler.

The details in the `.ll` and `.qs` files can provide a better understanding of the program's low-level execution flow. When debugging or trying to understand the results of optimization, referring to both `.ll` and `.qs` can be informative. For example, in optimizing measurement operations, when the compiler can be sure that a given boolean or `cbit` measurement outcome is dependent on another outcome or set of outcomes, then that measurement outcome can ultimately be determined by the classical part of the IR (especially in conjunction with a call to `release_quantum_state()`) and the measurement that set it can be omitted. Similarly, the dynamic parameters passed to some quantum gates can sometimes be combined by the compiler, reducing the number of operations on dynamic variables. Inspecting the `.qs` file will reveal which measurements and operations will be executed.

7.2 Compiler Optimization

As in compilation for classical programs, the LLVM-based Intel® Quantum SDK quantum compiler can look for opportunities to reduce the required quantum instructions and/or order and execute them more optimally. This optimization accounts for logical and physical constraints, and can be activated by passing one of the following optimization options:

- `-O0`:

  > This optimization flag represents no optimization at this time. This is the default if no flag is provided. Certain compiler passes will still be applied, such as converting to native gates.

- `-O1`:

  > This optimization flag enables high-level quantum optimizations on the `quantum_kernel` functions. At this time, the `-O1` optimization converts all `quantum_kernel` functions to a high-level representation we refer to as a "product-of-Pauli-rotations" representation. The overall unitary (and more generally, the quantum channel) is converted to an abstract Pauli-based form, consisting of:

- A sequence of Pauli-operator-based elements of the form e^{-itP}, where P is a general Pauli operator (tensor product of single-qubit Pauli operators) and t is any real number.
- Analogous elements for Clifford operations and non-unitary quantum operations such as measurement and qubit preparation.

Optimizations are performed on this representation, which is then used to synthesize a new circuit directly using native gates for the target backend. The synthesis process minimizes entangling gates and overall depth. The synthesis methods are adapted from [Schmitz2021], [Paykin2023], [Schmitz2023].

For `quantum_kernel` functions that use many qubit preparation operations, i.e. significantly more than the number of qubits used, use of `-O1` flag is known to dramatically slow down the compilation due to the intense amount of computation needed.

7.3 Qubit Placement and Scheduling

Note: This section distinguishes between a **physical** qubit, and a **program** qubit, which is the model used in users' programs. A "program qubit" is sometimes referred to as a virtual qubit. For the purposes of this section, the primary constraint of a physical qubit is that it will not have all-to-all connectivity, meaning it is not possible to perform a two-qubit gate between every pair of qubits. A physical qubit does not need to be implemented in hardware, and can exist solely in simulation.

The backends of the Intel® Quantum SDK provide features to simulate quantum hardware at different levels of idealization. For example, the `FullStateSimulator` backend provides an idealized quantum computer with unlimited ("all-to-all") physical connectivity between simulated physical qubits, so there is no "placement" decision required to map program `qbit` objects in the source code onto the physical qubits.

When all-to-all physical qubit connectivity is not available, some algorithms will require moving the program qubits around over the physical qubits.

The Intel® Quantum SDK compiler integrates the solution to this constraint into the quantum basic block functions it constructs from `quantum_kernel` functions. The placement compiler pass assigns program qubits (as declared in user's source code) to physical qubits (as defined in a platform configuration `.json` file). This is the initial placement of the program qubits which may change once the circuit has been processed by the scheduler compiler pass.

The scheduler compiler pass sequences the quantum instructions & gates, accounting for physical qubit connectivity by adding quantum instructions required to implement the algorithm. These additional quantum instructions effectively "move" the quantum information across physical qubits to perform a quantum gate between program qubits whose physical qubits were not directly connected.

7.3.1 Placement

By default, the placement pass assumes an all-to-all connectivity between the physical qubits and assigns the program qubits to physical qubits trivially, meaning program qubit 0 is assigned to physical qubit 0, program qubit 1 to physical qubit 1, and so on. If using the default mode, the `-c` flag (specifying a configuration file) is optional when invoking the compiler.

For example,

```
$ ./intel-quantum-compiler quantum_algorithm.cpp
```

In this case, `-c` is not required and the placement pass uses the default trivial placement.

When a configuration file is provided, the compiler offers four placement methods:

1. Trivial (`-p trivial`): Map program qubits to physical ones trivially (see above).

2. Dense (`-p dense`): Map the program qubits in a cluster of the highest connected portion of the given connectivity as defined in the platform configuration file.
3. Local (`-p local`): Use a local search optimization technique to place qubits that occur in the same gate close to each other.
4. Custom (`-p custom`): User provides the desired placement in their source code (see below).

In general, if the user wishes to select a placement method, the `-c` flag must also be specified. To invoke the placement pass, use the `-p` flag. Only one `-p` flag is accepted at a time.

```
$ ./intel-quantum-compiler -c configuration_file -p trivial quantum_algorithm.cpp
$ ./intel-quantum-compiler -c configuration_file -p dense quantum_algorithm.cpp
$ ./intel-quantum-compiler -c configuration_file -p local quantum_algorithm.cpp
```

If using custom placement, both insert the following line to define the placement in the source code:

```
// When defining the global qubit register, provide the custom placement.
qbit qreg[3] = {2, 0, 1}
// This places program qubit qreg[0] to physical qubit 2, qreg[1] to physical qubit 0, and qreg[2] to␣
↪physical qubit 1.
```

And invoke the placement pass with the `-p custom` flag:

```
$ ./intel-quantum-compiler -c configuration_file -p custom quantum_algorithm.cpp
```

Details about Local Search Placement

The local search compares two graphs with sets of vertices and edges. The application graph has vertices of program qubits and edges defined between two vertices if their respective program qubits appear in the same gate. It also considers the qubit connectivity graph, where the vertices are the physical qubits and the edges are the pairs of qubits for which the qubit chip natively supports operations between them. It would be ideal to place the qubits such that the placement maps an edge on the application graph maps to an edge on the qubit connectivity graph. However, this is not always possible. The local search can be configured with a certain amount of resets and a certain amount of iterations per reset. This can be passed through `-i=n` or `-r=n` as a compilation option.

Each reset starts out at a random placement, and iteratively swaps qubits using two qubits connected in the qubit connectivity graph. Half the time, it greedily chooses an edge to minimize a heuristic, and the other half of the time it does a random move to get out of a local minima.

The heuristic is based on iterating over all edges on the application graph, and for each edge adding up the minimal path length on the qubit connectivity graph between the two physical qubits that the placement maps the two program qubits onto that make up the edge in the application graph. There are some optimizations related to the fact that many of the terms in these sums do not need to be recomputed each time.

Known Limitations with the Placement pass

Custom placement can only be used on global `qbit` variables, not local `qbit` variables.

7.3.2 Scheduling

By default, the scheduler pass is disabled and an all-to-all connection is assumed of the device.

When the qubit connectivity is constrained, the scheduler adds SWAP gates to dynamically change the map specifying the program-to-physical-qubit assignment. For simplicity, we refer to this map as the "qubit map".

Updating the qubit map is often referred as "routing" since it can be visualized as a movement of the program qubits onto the physical qubit graph. Routing consists of two parts, performed once per QBB:

1. Forward routing: needed to satisfy the connectivity constraints when 2-qubit gates in the QBB need to be scheduled for execution. This changes the program-to-physical-qubit map.
2. Backward routing: needed to re-schedule the program qubits to the qubit map expected at the end of the QBB. This is a requirement of advanced quantum programs in which the order of QBB execution is not known at compile time. No backward routing is needed when the qubits are released after execution, i.e. when the QBB contains the command `release_quantum_state()`.

The forward routing method can be set by using the `-S` flag:

- `none`: connectivity constraints are neglected.
- `greedy`: given gate G between program qubits (qA, qB) currently mapped to physical qubits (QA, QB), the method search for two physical qubits (QC, QD) such that:
    1. Gate G is available between (QC, QD).
    2. The duration of a SWAP chains from QA to QC and from QB to QD, plus the duration of gate G(QC, QD) is minimized.
    3. The SWAP chains in point 2 are computed via A* search with the SWAP chain duration as the cost function.
    4. Ties in point 2 are broken by favoring solutions with balanced durations of the SWAP chains QA to QC and QB to QD.

  This method works for any connectivity.

The backward routing method can be set by using the `-K` flag:

- `retrace`: perform all the SWAP gates added in the forward routing but in opposite order. Cancel consecutive SWAP gates on the same pair of physical qubits. This method is often inefficient, but the overhead is at most twice the forward routing cost. It works for any qubit connectivity.
- `bubble-sort`: for linear connectivity only, based on the bubble-sort algorithm. It considers the qubit map desired at the end of the QBB as defining the order among program qubits and the qubit map at the end of the forward routing as an unordered program qubit sequence (to be ordered via bubble sort). This works also for non-linear connectivity when a Hamiltonian path can be identified via a simple heuristic.
- `oddeven-sort`: as for `bubble-sort` but using the odd-even transposition sort algorithm.
- `grid`: for 2D grid connectivity only, based on the successive ordering along rows, columns, and rows again. The size of the 2D grid is identified automatically given that the physical qubits are ordered with the row-major ordering.

To invoke the scheduler pass, use the `-S` and `-K` flags:

```
$ ./intel-quantum-compiler -c configuration_file -S greedy -K bubble-sort quantum_algorithm.cpp
```

Known Limitations with the Scheduler pass

If the scheduler pass `-S` flag is not set or set to "none", the compiler assumes an all-to-all connectivity even if a non-all-to-all connectivity is given in the config .json. Conversely, to invoke the `-S` flag, the `-c` flag must be given.

If the `-p` flag is given, the scheduler will use the placement generated by the placement pass as an initial placement. If the `-p` flag is not given but the `-S` flag is set, the scheduler will assume a trivial initial placement.

When the `-S` flag is not set and `-O1` optimization is set, some `quantum_kernel` functions may see additional `quswapalp` gate operations at the end of the `quantum_kernel`.

When the `-K` flag is set to either `bubble-sort` or `oddeven-sort` but a Hamiltonian path cannot be found in the connectivity graph of physical qubits, the default `retrace` method is used.

When the `-K` flag is set to `grid` but the connectivity graph does not correspond to a row-major 2D array of qubits, the default `retrace` method is used.

7.3.3 Combining the `-p` and `-S` flags

Placement and scheduling passes can be invoked together with the `-p` and the `-S` flags:

```
$ ./intel-quantum-compiler -c configuration_file -p custom -S greedy quantum_algorithm.cpp
```

7.3.4 Sample Platform Configuration files

The SDK comes with example platform configuration files representing the details of a different implementation of quantum hardware. They are:

- 8 qubits: Linear connectivity targeting the quantum dot simulator backend.
- 9 qubits: Square grid connectivity targeting non quantum dot simulation backends.
- 34 qubits: Linear connectivity targeting non quantum dot simulation backends.
- 256 qubits: Square grid connectivity targeting non quantum dot simulator backends.
- 256 qubits: Ladder connectivity targeting non quantum dot simulator backends.
- 256 qubits: Linear connectivity targeting non quantum dot simulator backends.

For usage with the quantum dot simulator, use `intel-quantum-sdk-QDSIM.json` which points to the 8 qubit configuration file. For usage with the non quantum dot simulator backends or just the compiler, use `intel-quantum-sdk.json`. By default, `intel-quantum-sdk.json` points to the 256 qubit configuration file. If you wish to use other configuration files, please copy `intel-quantum-sdk.json` to your own directory, modify the pointed to configuration file and use the `-c` flag to point to your copy.

7.4 Circuit Printing & LaTeX

To invoke the circuit printer, user the `-P` flag:

```
$ ./intel-quantum-compiler -P console quantum_algorithm.cpp
$ ./intel-quantum-compiler -P tex quantum_algorithm.cpp
$ ./intel-quantum-compiler -P json quantum_algorithm.cpp
```

For each `quantum_kernel` function in the source code compiled, the circuit printer feature will output a representation of the quantum kernel to the target specified. The `console` target will result in ascii-style circuits being displayed to the console. The `tex` and `json` targets will output, for each `quantum_kernel` function, a separate `.tex` or `.json` file.

A TeX distribution on your local machine with the `qcircuit` package (maintained at the Comprehensive TeX Archive Network (CTAN)) and its dependencies are required to produce an image or PDF file from the `.tex` file. Many options for TeX distributions exist for each platform. Those familiar with the LaTeX typesetting language will be able to incorporate the `.tex` file or part of its contents into their projects. Those familiar with the commands of the `qcircuit` package may customize and extend the diagram at will.

7.5 Support for OpenQASM 2.0

The Intel® Quantum SDK provides a source-to-source converter which takes OpenQASM code and converts it into C++ for use with the Intel® Quantum SDK. This converter requires Python 3; see Getting Started Guide (System Requirements) section for specifics and recommendations. Currently, it processes OpenQASM 2.0 compliant code as described by the Open Quantum Assembly Language paper (arXiv:1707.03429 [quant-ph]).

To translate an OpenQASM file to C++ file, you can run the compiler with the `-B` flag to generate the corresponding `quantum_kernel` functions in C++ format.

```
$ ./intel-quantum-compiler -B example.qasm
```

7.6 Other Compiler Flags

Verbosity - `-v`:

Provides a summary of each `quantum_kernel` in terms of both the supported gates set and the quantum dot qubit gates set.

8.0 Configuring the `FullStateSimulator`

Before a `quantum_kernel` can be called, a properly configured instance of the `FullStateSimulator` class is required. This can be done by creating an `IqsConfig` object with the desired values and passing it to the constructor or initializer of the `FullStateSimulator`. The type `QRT_ERROR_T` is used to check-on the status of simulator instance. For example,

```
// configure to use N qubits; accepts defaults for remaining
iqsdk::IqsConfig iqs_config(/*num_qubits*/ N);

// setup quantum device
iqsdk::FullStateSimulator iqs_device(iqs_config);
iqs_device.printVerbose(true);

// ensure setup was successful
if (iqsdk::QRT_ERROR_SUCCESS != iqs_device.ready()) return 1;
```

The essential classes and methods for configuration are detailed below. See API Reference for the full list of APIs to find details about retrieving data.

8.1 Overview of `FullStateSimulator`

Class with API calls to both set up a quantum simulator device and access the underlying quantum state during simulation.

- Constructor

  ```
  FullStateSimulator(IqsConfig &device_config);
  ```

  Instantiates a simulator object that is initialized to the settings in `device_config`.

- `printVerbose()`

  ```
  QRT_ERROR_T FullStateSimulator::printVerbose(bool printVerbose);
  ```

  Sets the status of the simulator's verbose output.

- `ready()`

  ```
  QRT_ERROR_T FullStateSimulator::ready();
  ```

  Returns an `enum` of `QRT_ERROR_T`; `QRT_ERROR_SUCCESS` if the simulator is ready to run a `quantum_kernel`, else returns `QRT_ERROR_FAIL`. Ensure the simulator is ready before executing `quantum_kernel` functions or making any queries.

  Provides a trigger for opportunities to define error handling logic.

8.2 Execution Options

The Intel® Quantum SDK backends have two execution modes:

- Synchronous (default): pauses the execution of the program whenever a QBB is called. Execution resumes once the QBB is done running.
- Asynchronous: the host puts the QBB into a queue of QBBs to be run.

Prior to using the results of any measurements, the user should call `wait()` on the device to ensure that the device has finished running and set the appropriate `cbit(s)`. Any API that sets a device property (e.g. setting contraction path method) is put on the queue, while any API that gets simulation data from the device blocks until the device has finished running.

The `synchronous` parameter in the `DeviceConfig` specifies whether the backend will run in synchronous or asynchronous mode. Other backends such as **Clifford Simulator**, **Tensor Network Simulator**, and a user-defined **Custom Backend** can also utilize the `asynchronous` execution mode for faster simulations.

8.3 Overview of `IqsConfig`

Class to hold configuration data used to configure the `FullStateSimulator` or user-defined qubit simulator backend.

- Constructor

```
IqsConfig(int num_qubits = 1,
          std::string simulation_type = "noiseless",
          bool verbose = false,
          std::size_t seed = time(NULL),
          bool synchronous = true,
          double depolarizing_rate = 0.01);
```

Specify configuration data for the IQS. Creates an `IqsConfig` which has the following properties:

`int num_qubits`**:** Number of qubits in simulation.

`std::string simulation_type`**:** Type of simulation to be run. Valid simulation types are: `"noiseless"`, `"depolarizing"`, and `"custom"`. See **Customizable Noise Modeling** for details on the `"custom"` option.

`std::size_t seed`**:** Custom seed for RNG. If no seed is provided, the current time will be used as the seed.

`double depolarizing_rate`**:** Depolarizing rate for noisy simulation.

- `isValid()`

```
bool IqsConfig::isValid();
```

Returns whether the given config instance is valid.

9.0 Intel® Quantum Simulator Backend

Intel® Quantum Simulator (IQS) is a full-state simulator working at the qubit level, abstracting the physics of the specific implementaion. It is available as a standalone open-source project, but it also comes fully integrated as one of the backends of the Intel® Quantum SDK [KWPH2022]. IQS is designed to take full advantage of High Performance Computing (HPC) infrastructure and allows both multi-thread (shared memory, using OpenMP) and multi-process parallelization (distributed memory, using MPI) [GHBS2020].

The API has already being described in the context of full-state simulators (see **Configuring the FullStateSimulator**). Here we focus on the possibility of adding a customizable noise model in the simulation. The programmer does not need to be familiar with IQS, and no IQS code or APIs need to be used.

9.1 Customizable noise modeling

The user can customize the action of every quantum operation within the template provided below by defining appropriate functions. The action of each operation is divided in three parts:

- **Pre-operation:** Apply one or more of the following phenomenological noise channels:
  - Dephasing channel
  - Depolarizing channel
  - Amplitude damping
  - Bitflip channel

  Each effect is characterized by an intensity parameter.
- **Operation itself:** The choice here is whether to apply the ideal operation or a user-provided process matrix (also known as the χ matrix). In the latter case, the user can include all noise effects directly in the process matrix, and thus avoid pre- or post-operation actions. However, we find it convenient to provide the pre- and post-operation templates to facilitate writing standard noise models quickly.
- **Post-operation:** Similar to the pre-operation case, the user can apply one or more of the following phenomenological noise models:
  - Dephasing channel
  - Depolarizing channel
  - Amplitude damping
  - Bitflip channel

  Each effect is characterized by an intensity parameter.

9.1.1 Custom operation definition

The definition of a custom operation is provided by means of objects of type `iqsdk::IqsCustomOp`, which can be initialized as follows:

```
IqsCustomOp op = {pre_dephasing, pre_depolarizing, pre_amplitude_damping, pre_bitflip,
                  process_matrix, label,
                  post_dephasing, post_depolarizing, post_amplitude_damping, post_bitflip};
```

where:

- `pre_dephasing, pre_depolarizing, pre_amplitude_damping, pre_bitflip` are scalar values representing the intensity of pre-operations.
- `process_matrix` is a `std::vector<std::complex<double>>` in row-major format. When the operation is ideal, one can simply use an empty vector as `process_matrix`.
- `label` is a string used as unique tag for the process matrix. If multiple operations use the same process matrix (for example, the CZ gate on different pairs of simulated physical qubits), assigning the same label reduces the memory and computation by using a single process matrix.
- `post_dephasing, post_depolarizing, post_amplitude_damping, post_bitflip` are scalar values representing the intensity of post-operations.

If the complete operation is noiseless, one can simply use the global object:

```
iqsdk::k_iqs_ideal_op = {0, 0, 0, 0, {}, "ideal", 0, 0, 0, 0}
```

already defined in the header `quantum_full_state_simulator_backend.h`.

9.1.2 Custom operation specification

While the subsection above explained how to define a single custom operation, we still need to specify the behavior of a custom action. For example, one may want to return different `IqsCustomOp` objects for the same gate type depending on the simulated physical qubit as a way of having the noise reflect that of a realistic, inhomogeneous device.

The user needs to write a function for every quantum operation returning the appropriate `IqsCustomOp` object for the given parameters of the quantum operation. For example, one may want to use a simplified noise model for the one-qubit gates by expressing them as ideal gates followed by depolarization. At the same time, they may want to use a process matrix describing the action of the two-qubit CZ gates. One may even use different process matrices depending on the qubits involved in the gate.

In a simple example, a custom CPhase operation with a 10% chance of a dephasing error prior to the gate executing would be defined as:

```
iqsdk::IqsCustomOp CustomCPhaseRot(unsigned q1, unsigned q2, double g) {
  return {0.1, 0, 0, 0, {}, "cphase_dephasing", 0, 0, 0, 0};
}
```

9.2 Using Custom IQS Noise Models in a Program

To enable the IQS with a customizable noise model, an `IqsConfig` should be declared with `"custom"`.

```
iqsdk::IqsConfig custom_iqs_config(N, "custom");
```

where `N` is the number of qubits. Then, associate the desired functions to the customizable actions:

```
custom_iqs_config.PrepZ = CustomPrepZ;
custom_iqs_config.RotationXY = CustomRotXY;
custom_iqs_config.CPhaseRotation = CustomCPhaseRot;
```

Here, `custom_iqs_config.<name>` are set to user-defined functions with the following signatures:

```
iqsdk::IqsCustomOp PrepZ(unsigned qubit);
iqsdk::IqsCustomOp MeasZ(unsigned qubit);
iqsdk::IqsCustomOp RotationXY(unsigned qubit, double phi, double gamma);
iqsdk::IqsCustomOp RotationZ(unsigned qubit, double gamma);
iqsdk::IqsCustomOp ISwapRotation(unsigned qubit_1, unsigned qubit_2, double gamma);
iqsdk::IqsCustomOp CPhaseRotation(unsigned qubit_1, unsigned qubit_2, double gamma);
```

Not all of the functions need to be defined. If they are not defined, they will default to the ideal operation. Since the customizable noise model is compatible with full-state simulators, the IqsConfig is passed to an instantiation of a full-state simulator.

```
iqsdk::FullStateSimulator custom_iqs_device(custom_iqs_config);

if (iqsdk::QRT_ERROR_SUCCESS != custom_iqs_device.ready()) return 1;
```

A complete code example can be found in the `custom_backend.cpp` sample described in Code Samples.

9.3 Important Points on Performing Noisy Simulations with IQS

The Intel® Quantum SDK allows noisy simulations of qubits with the Intel® Quantum Simulator as the backend. Noise is incorporated during a simulation via stochastic injection of noise based on the specified noise intensity parameter. Thus, it is necessary to aggregate results from multiple simulations to accurately simulate a noisy qubit system. Given that the stochastic nature is realized via the initial seed, it is imperative that the user instantiates the backend with a different seed each time the same quantum circuit is run during sample collection. By processing the resulting samples, the probabilities from noisy simulations can be reconstructed. This sampling process can be made efficient by using the asynchronous mode of simulations, whereby multiple simulator backends initialized with different seeds are used to simultaneously perform simulations depending on the available memory and processing capability.

10.0 Quantum Dot Simulator Backend

Quantum Dot Simulator (QD Simulator) is a simulator reproducing the physics of a Quantum Dot (QD) qubit chip in software. Simulation of quantum systems is a field of great importance [GEAN2014]. In quantum computing, there are benefits in accurately simulating quantum systems for the purpose of evaluating their strengths and weaknesses for use as qubits. Simulations help drive design decisions on the critical characteristics for physical realizations [KPZC2022]. Though there are many ways of performing quantum simulations, here we focus on Schrodinger evolution for simulating quantum dot qubits. This QD Simulator is used as the realistic qubit simulation backend of the Intel® Quantum SDK [KWPH2022] [KPZC2022].

10.1 Simulation of Qubits

Qubits are quantum mechanical systems with two distinct states, typically labeled $|0\rangle$ and $|1\rangle$ [BaSR2021], [NICH2010]. The current backend for quantum dot qubits utilizes qubit states encoded in the spin degree of freedom of single electrons [ZKWL2022]. These qubits are typically referred to as Loss-DiVincenzo qubits [LODI1998]. Abstract qubits are simple systems with only two isolated levels. However, practical quantum systems are never quite as simple, with careful consideration required for selection of a suitable system to form a qubit [DIVI2000]. These requirements and the thought process behind the selection of some currently favored types of qubits were reviewed in [LJLN2010]. One important fact common to all of these qubits is the presence of a characteristic resonance frequency or natural frequency. The frequency usually refers to the energy difference (expressed as a frequency) between the qubit levels (computational states) of the quantum system being considered for digital gate-based quantum computing. Resonance frequencies for most types of qubits are 1 GHz to 30 GHz, though there are exceptions with much higher or lower frequencies.

When using the QD Simulator backend, the simulation goes through the qubit control processor, the control electronics, to the simulated quantum dot qubits. The qubit control processor takes the compiled instruction sequence and the platform configuration files to generate the corresponding micro-instructions for the control electronics. The control electronics generate the RF and DC pulses with the correct parameters to interact with the quantum dot qubit chip. All the control flow and operations are modeled in simulation.

The primary supported gates are $R_{xy}(\theta, \phi)$, referred to in code as RXY:

$$
\begin{aligned}
R_{xy}(\theta, \phi) &= \cos\left(\frac{\theta}{2}\right)\hat{I} - i\sin\left(\frac{\theta}{2}\right)\left[\hat{X}\cos\phi + \hat{Y}\sin\phi\right] \\
&= \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right)[\cos\phi - i\sin\phi] \\ -i\sin\left(\frac{\theta}{2}\right)[\cos\phi + i\sin\phi] & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}
\end{aligned}
$$

and the two-qubit operation CZ:

$$\mathrm{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

The physical implementation of CZ involves the use of a "Decoupled CZ operation" [WAPK2018]. All the other operations available via the Intel® Quantum SDK will be constructed using these operations.

10.2 Rotating vs. Laboratory Frame

Typically, if time dependence of the system can be set aside, simulation of quantum systems is convenient and fast. For certain quantum systems, it is possible to craft unitary transformations to analytically discard the overhead due to the resonance frequency of each qubit [SURI2015] [STEC2020] [NICH2010]. This is typically referred to as moving into the **rotating frame** of the qubit. This terminology is apt since the qubit is always precessing and incrementing its phase around the z-axis at a rate given by its resonance frequency. A further analytical approximation, known as the **rotating wave approximation** [ZHSD2020], is usually required to make the time-dependence fully transparent. These transformations and approximations usually have the effect of drastically reducing the burden on simulation resources, since evolution will then happen at kHz or MHz scales instead of GHz scales.

In the case of QD Simulator [KPZC2022], neither the rotating frame nor the rotating wave approximation is used. Currently, the evolution of the coupled multi-quantum-dot system (faithful to Intel®'s quantum hardware) is performed in the **laboratory frame**. The laboratory frame is the original environment of the quantum system, where the natural frequencies of the qubits are fully visible. This also means that the qubits are constantly accumulating Z-phases as is the case for real qubits.

10.3 Usage in conjunction with `getAmplitudes()`

The Schrodinger evolution is carried out in a Hilbert space that encompasses several energy levels per quantum dot, to ensure accurate modeling of the interactions. Since QD Simulator is performing a full quantum simulation, users have access to the fully evolved state vector (following truncation to the computational subspace) at the end of a simulation. As evolution is happening in the lab frame, the probability amplitude results returned from `FullStateSimulator::getAmplitudes` will include the extra Z-phases that were accumulated due to natural precession, and the extra phases will be dependent on the resonance frequencies as well as the full evolution history during algorithm execution. Since this detailed history is unavailable to users, the use of the latter function for full state characterization is discouraged.

This further highlights how closely the simulations with QD Simulator reflect actual quantum dot qubits. With physical qubits it is impossible to obtain actual probability amplitudes after evolution. Just as with physical qubits, techniques such as quantum state tomography [PARE2004] are required to reconstruct the full state when using QD Simulator.

10.4 Using Quantum Dot Simulator in a Program

To enable QD Simulator, a `DeviceConfig` should be declared with `"QD_SIM"`.

```
iqsdk::DeviceConfig qd_sim_config("QD_SIM");
```

Then, create a `FullStateSimulator` with the QD Simulator `DeviceConfig`:

```
iqsdk::FullStateSimulator qd_sim_device(qd_sim_config);
```

Once the simulator is configured, then the `quantum_kernel` functions can be called to perform simulations on the QD Simulator.

10.5 Important Points on Quantum Dot Simulator

Because the QD Simulator behaves more like realistic hardware, it carries a few limitations on the kinds of `quantum_kernel` functions that can be used in conjunction with it. Specifically, it expects that each `quantum_kernel` in `main()` will consist of a workload where

- All the qubits start in the $|0\rangle$ state
- A sequence of 1-qubit and 2-qubit operations are applied
- The final probabilities or amplitudes for each basis state are retrieved.

There is no continuity between `quantum_kernel` functions called within `main()`, because each time a `quantum_kernel` is called within `main()`, the QD Simulator history is reset and all qubits will start in the $|0\rangle$ state.

If sub `quantum_kernel` functions are to be used, they must be specified outside of `main()` and combined as desired within a single `quantum_kernel`, and then called in `main()`.

`MeasZ` operation is not advised to be used with the QD Simulator. This operation is designed to collapse the target qubit, and to store the result in a `cbit`. Using this operation will set the cbit according to the probability distribution associated with the quantum state at the end of the quantum_kernel, and will not collapse the state. In addition, `MeasX` and `MeasY` will likely give incorrect results.

Prepare operations (e.g. `PrepZ`) should be reserved for use either at the beginning of a `quantum_kernel`, or not used at all. Using `PrepZ` should provide benefits with compiler optimizations when using the `-O1` flag. Not using `PrepZ` at the beginning will not impact the QD Simulator, since the qubits will always be reset to $|0\rangle$ when starting a simulation. However, using `PrepZ` or `MeasZ` in the middle of simulating a `quantum_kernel` on QD Simulator will result in unexpected behavior.

Note that Z rotations are currently not natively enabled for the hardware in simulation. Hence a user wishing to use $RZ(\theta)$ can expect the compiler to implement it in one of two ways:

- If using compiler optimization (`-O1`), then the compiler will absorb all RZ operations into other single-qubit operations.
- If not using compiler optimization (`-O0`), the RZ operation (or related operations such as S, T, etc.) will be explicitly decomposed into RXY operations as follows:

```
quantum_kernel void rzDecomp (qbit qb, double angle) {
  RXY(qb, M_PI, 0.5 * M_PI);
  RXY(qb, M_PI, 0.5 * angle - 0.5 * M_PI);
}
```

10.5.1 Tip for Faster Simulations

Avoid all operations on qubits that have no gates applied. Any operations, including prepare (`PrepZ`), applied to a qubit causes it to be simulated. This means that even if a qubit only has `PrepZ` & `MeasZ` applied to it, it will still be simulated which adds overhead and increases runtime.

10.6 Compilation with Quantum Dot Simulator as the Computing Backend

To enable QD Simulator, a platform configuration file that describes the configuration of quantum operations and the connectivity of the qubits must be given to the compiler. Users also need to specify flags and arguments for placement and scheduling. The following example assumes the SDK location has already been added to the shell path,

```
$ intel-quantum-compiler -c /<path to config file>/intel-quantum-sdk-QDSIM.json -p trivial -S greedy qd_
↪GHZ.cpp
```

11.0 Clifford Simulator Backend

The Clifford Simulator is a specialized qubit simulator which can process and evaluate the outcome of quantum circuits composed only of Clifford gates and Pauli measurements. The Clifford group [HDER2006] can be broadly described as the group which transforms Pauli operators to Pauli operators. It is well known that Clifford operations are not universal for quantum computation, and that they are efficiently simulatable with classical computers [GOTT1998] [NEST2010]. In this sense, the Clifford Simulator is not a general purpose qubit simulator. However, Quantum Error Correction (QEC) is an application area that makes extensive use of Clifford operations. Thus for studying QEC or related applications involving only Clifford operations, the Clifford Simulator can serve as a powerful tool due to its scalability, low memory footprint, and focus on application of Clifford operations and Pauli measurements.

The Clifford Simulator adapts the methods of the **Pauli Tableau** [SCGO2004] using a sparse representation of the underlying Pauli operators to form the tableau. This means there is no cost to unused qubits in the tableau as the data structure expands as gates are applied.

11.1 Clifford Operations

The Clifford group is super-exponentially large in the number of qubits. However, it is possible to efficiently decompose any arbitrary Clifford unitary to the one-qubit gates **H**, **S**, and the two-qubit gate **CNOT** [HDER2006]. The gates in their matrix representations are given below for convenience.

$$\mathsf{H} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\mathsf{S} = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$

$$\mathsf{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The supported gates of the Intel® Quantum SDK which are allowed for the Clifford Simulator are:

- Hadamard (H)
- Pauli X (X)
- Pauli Y (Y)
- Pauli Z (Z)

- Phase (`S`)
- Phase Inverse (`Sdag`)
- X axis Rotation (`RX`) For angles $0, \pi/2, \pi, 3\pi/2$
- Y axis Rotation (`RY`) For angles $0, \pi/2, \pi, 3\pi/2$
- Z axis Rotation (`RZ`) For angles $0, \pi/2, \pi, 3\pi/2$
- Controlled Z (`CZ`)
- `CNOT`
- `SWAP`
- `PrepZ`
- `PrepX`
- `PrepY`
- `MeasZ`
- `MeasX`
- `MeasY`
- `CPhase` for angle π
- XY-plane Rotation (`RXY`) for both (`theta`, `phi`) angles in $\{0, \pi/2, \pi, 3\pi/2\}$ and the angle pairs $(\pi, \pi/4)$ and $(\pi, 3\pi/4)$
- `SwapA` for angle π

11.2 Using Clifford Simulator in a Program

To enable the Clifford Simulator, a `CliffordSimulatorConfig` should be declared as shown below. The seed is optional (defaults to a seed based on the time), but should be user-specified especially if modeling the effects of noise.

```
iqsdk::CliffordSimulatorConfig clifford_config(seed);
```

Then, create a `CliffordSimulator` with the `CliffordSimulatorConfig`, and call the `ready()` API for the simulator just before use; see API Reference for description.

```
iqsdk::CliffordSimulator clifford_device(clifford_config);
clifford_device.ready();
```

Once the simulator is configured and `ready()` is called, then the `quantum_kernel` functions can be called to perform simulations on the Clifford Simulator.

11.3 Important Points on Clifford Simulator

11.3.1 Using the Pauli Error Model

The Clifford Simulator includes a built-in Pauli error model which is **off** by default. To turn it **on**, set the flag for the passed `CliffordSimulatorConfig`:

```
iqsdk::CliffordSimulatorConfig clifford_config(seed);
clifford_config.use_errors = true;
```

All gate errors are based on the Pauli Twirling Approximation [GEZH2013] where the exact gate action is applied to the simulator, followed by a subset of Pauli operators with probability as defined by the parameters of the error model. The parameters of the error model are specified gate-by-gate. These are collectively held in `CliffordSimulatorConfig::error_rates` which is of type `struct iqsdk::ErrorRates`, and contains the members:

- `iqsdk::ErrSpec1Q prep`
- `iqsdk::ErrSpec1Q meas`
- `iqsdk::ErrSpec1Q xyrot`
- `iqsdk::ErrSpec1Q zrot`
- `iqsdk::ErrSpecIdle idle`
- `iqsdk::ErrSpec2Q cz`
- `iqsdk::ErrSpec2Q swap`

where `struct iqsdk::ErrSpec1Q`, `struct iqsdk::ErrSpec2Q` and `struct iqsdk::ErrSpecIdle` represent three Pauli twirling error models.

- `iqsdk::ErrSpec1Q` is a general single qubit error model where the probability for each of the single-qubit Pauli operators X, Y, Z can be individually specifed so long as their sum is less than 1. This can be set via the constructor `iqsdk::ErrSpec1Q(double x_rate, double y_rate, double z_rate)`.
- `iqsdk::ErrSpec2Q` is a more specific two-qubit error model based on the non-ideal CZ gate as described in [SJDL2003]. It is specified by an off-diagonal **switching** probability `e`, an off-diagonal phase `phi` and control-phase error angle `delta`. These parameters can be set via the constructor `iqsdk::ErrSpec2Q(double e, double phi, double delta)`.
- `iqsdk::ErrSpecIdle` is a single qubit decoherence model ($\alpha = 0$ from [GEZH2013]). Unlike the other two which are a fixed amount of error for each gate, this model is time-dependent and is used for idle error. It is specified by two time-like parameters, `T1` representing the depolarization rate and `T2` which represents the dephasing rate. These parameters can be specified via the constructor `iqsdk::ErrSpecIdle(double T1, double T2)`.

For the sake of generating idle error, the Clifford Simulator assumes as-soon-as-possible scheduling of the gates, and from this, applies idle error based on gaps in this scheduling. For this purpose, gate times can be independently specified as `CliffordSimulatorConfig::gate_times` which is of type `iqsdk::GateTimes` and contains the data:

- `double prep`
- `double meas`
- `double xyrot`
- `double zrot`
- `double idle`
- `double cz`
- `double swap`

11.3.2 Collecting State Information

Since the Clifford simulator is not a full state simulator, the primary API function to retrieve results of quantum circuit execution is `getExpectationValue`. With this function, the user can specify a Pauli string (e.g. `XX`, `YZ`) for their desired observable, and the simulator will directly return the expectation value. No state collapse is performed when this API is called.

Such observables are specified by passing a `std::vector` of `std::reference_wrapper<qbit>`, representing the operator's support and a `std::string` containing only characters from the set `{'I', 'X', 'Y', 'Z'}` representing the single-qubit operator type as matched to the qubit support. A detailed example for the usage can be found in the example `api_Simulator_clifford_test.cpp` (see Samples).

As with the Intel® Quantum Simulator, it is also possible to utilize individual measurements (`MeasZ`, `MeasX`, or `MeasY`) to simulate single-shot results when using the Clifford simulator. Use of these measurement gates **do** result in state collapse. This mode of collecting results is crucial in modeling applications such as Quantum Error Correction as it captures correlation not represented through the `getExpectationValue()` API. The single-shot results can then be aggregated and analyzed. This is the most straight forward way to compare with execution on the Intel® Quantum Simulator or Tensor Network simulator.

11.3.3 Tip for Faster Simulations

In the case of modeling noise, it is necessary to use a different seed when initializing the `CliffordSimulatorConfig`. See Important Points on Performing Noisy Simulations with IQS for the motivation behind this. A convenient method to follow could be to create configurations with different seeds, and then spawn simulator instances based on each of the uniquely seeded configurations, based on the number of samples required. If the running mode for the simulators is set to **asynchronous**, then multiple simulators can be executed in parallel, and results collected later. Using the `wait()` simulator API ensures that the given simulator has completed execution before moving on to the next part of the program. See Execution Options.

In the asynchronous operation mode, care should also be taken when writing to `cbit` variables during measurements. One possibility to avoid overwriting the same variables is to set up a multidimensional array of `cbit` type. With this technique, each simulator will have its own dedicated set of `cbit` variables that will be populated during execution. If multiple sets of cbits are required, the dimensionality of the array can be further extended. Another possibility is to use local `cbit` variables and have them be returned for further analysis from the `quantum_kernel` function upon completion.

A detailed example for the above usage scenarios can be found in the example `iqs_vs_clifford_comparison.cpp` (see Samples).

A detailed example of these methods for QEC scaling simulation can be found in the example `rep_code_clifford.cpp` (see Samples).

11.4 Compilation with Clifford Simulator as the Computing Backend

The Clifford Simulator can accommodate arbitrary qubits connectivities for compilation. The default connectivity is all-to-all (fully-connected) with support for up to 256 qubits. See the Configuration files section for other available configurations. Also, see the Scheduling section on how to enforce connectivity constraints during compilation.

12.0 Tensor Network Backend

The Tensor Network (TN) backend is a qubit simulator that represents a quantum circuit as a network of Tensors. Unlike some Tensor Networks which take on a specific form (e.g. Matrix Product State (MPS) or Tree Tensor Network (TTN)), the Tensor Network we use can take on an arbitrary geometry based on the simulated circuit. With the exception of calling State Preparation & Measurement (SPAM) operations, running a `quantum_kernel` will build up the Tensor Network which is an inexpensive operation. Since SPAM operations are treated as mid-circuit operations, it is recommended to avoid them unless you specifically are intending to see the effects of mid-circuit measurements since they have a high cost to perform.

The most cost-effective way to sample an algorithm is to call `getSamples()`; unlike explicit SPAM gates, `getSamples()` will not collapse the state.

The part of the Tensor Network simulation that requires the most computation is when the Tensor Network requires a contraction. Before a contraction, the TN simulator will do a search to try to find the best contraction path. Then, using the path found it will perform the contraction to the desired result. Finding a good contraction path can in some cases drastically reduce the amount of computation needed to do the contraction. Depending on which API is called (i.e. `getProbabilities()` or `getExpectationValue()`), a different contraction will be performed to get the result.

Tensor Networks are best at simulating circuits that have a low tree-depth.

When implementing variational quantum algorithms, such as a Quantum Approximate Optimization Algorithm (QAOA)-style algorithm, it is best to use a new instantiation of the Tensor Network object each time. Simulating a circuit twice with different parameters is inefficient:

1. The TN will need to reset the quantum state with expensive mid-circuit preparations.
2. During the second contraction, the tensors from both runs of the ansatz will be contracted.

In general, the user should take care in making sure that the result they are trying to get out of the Tensor Network simulator is reasonable. Depending on the quantum circuit, retrieving amplitudes of a few states in a 100-qubit algorithm may be possible, but retrieving all of them would not be.

12.1 Brief Overview of `TensorNetworkConfig`

Class to hold configuration data specifically used to configure the `TensorNetworkSimulator`.

- Constructor

```
TensorNetworkConfig(bool verbose = false,
                    bool synchronous = true);
```

  Specify configuration data for the TN backend. Creates a `TensorNetworkConfig` which has the following properties:

  `bool verbose`**:** Verbosity of the simulator.

  `bool synchronous`**:** Whether the simulator is synchronous.

- `isValid()`

```
bool TensorNetworkConfig::isValid();
```

  Return whether the given config instance is valid.

The `TensorNetworkSimulator` is the class used for doing Tensor Network simulation. The `TensorNetworkConfig` initializes the Tensor Network simulator.

The `TensorNetworkSimulator` can use any API from the `FullStateSimulator`.

As an example for Tensor Network Simulator-specific API details:

```
QssMap<double> TensorNetworkSimulator::getProbabilities(
    std::vector<std::reference_wrapper<qbit>>& qids,
    std::vector<QssIndex> bases,
    double threshold=-1);
```

`getProbabilities()` returns the conditional probabilities of a subset of the qubits (`qids`) used in the simulation. If `bases` is empty, then the Tensor Network will perform a single contraction directly to the tensor network of all conditional probabilities of the given qubits. Otherwise, for each `QssIndex` in the given `bases`, a contraction is done to compute the specific conditional probability of the specified basis.

Three additional APIs are available for the Tensor Network Simulator.

- `draw()`:

  ```
  void draw();
  ```

  Creates a graphical representation of the Tensor Network. The graphic will appear in a window.

- `getExpectationValue()`:

  ```
  double getExpectationValue(
    std::vector<std::reference_wrapper<qbit>> &qids, std::string pauli_string);
  ```

  Returns the expectation value of the given Pauli operator `pauli_string`.

- `setContractionPathOptimizer()`:

  ```
  void setContractionPathOptimizer(std::string optimizer_method);
  ```

  Sets the contraction path optimizer to `optimizer_method`. The default optimizer is `"greedy"`. Valid options are `"optimal"`, `"dynamic-programming"`, `"branch"`, `"greedy"`, `"random-greedy"`, `"random-greedy-128"`, `"auto"`, and `"auto-hq"`. See https://optimized-einsum.readthedocs.io/en/stable/path_finding.html for more details.

13.0 Custom Backend

The custom backend feature allows a user to use the Intel® Quantum Compiler and Quantum Runtime (QRT) with their own simulator.

Users will need to include the <quantum_custom_backend.h> header file to use the Custom Backend.

In the header file there is a `CustomInterface` class and a `CustomSimulator` class. The `CustomInterface` class is an abstract base class where the user can implement their own simulator. The `CustomSimulator` class is similar to `FullStateSimulator`, `TensorNetworkSimulator`, or `CliffordSimulator` classes in that this is the class representing the quantum device.

13.1 `CustomInterface`

The `CustomInterface` has the following abstract methods that must be implemented by the user in any derived class:

- `RXY()`

```
void RXY(qbit q, double theta, double phi) = 0;
```

  The function called by the QRT to apply a Rotation-in-the-XY-plane (`RXY`, **RXY Matrix Representation**) gate.

- `RZ()`

```
void RZ(qbit q, double angle) = 0;
```

  The function called by the QRT to apply a Rotation-around-Z-axis (`RZ`) gate.

- `CPhase()`

```
void CPhase(qbit ctrl, qbit target, double angle) = 0;
```

  The function called by the QRT to apply a Controlled-Phase (`CPhase`) gate.

- `SwapA()`

```
void SwapA(qbit q1, qbit q2, double angle) = 0;
```

  The function called by the QRT to apply a Swap-Alpha (`SwapA`) gate.

- `PrepZ()`

```
void PrepZ(qbit q) = 0;
```

  The function called by the QRT to prepare the specified qubit in the Z basis (`PrepZ`).

- `MeasZ()`

```
cbit MeasZ(qbit q) = 0;
```

  The function called by the runtime to measure the qubit in the Z basis (`MeasZ`). The return value is the result of the measurement. The QRT will map the measurement into the appropriate bool (or `cbit`) variable.

The user is free to implement any other functions that they may wish to call in this class as well. In addition, they can implement a constructor that takes in any number of arguments. Also, the base class does not include utilities to manage a state vector or other representation of the quantum state. The user will also need to manage this information if it is needed.

13.2 CustomSimulator

The user needs to register their simulator with the QRT. The following example assumes that the user made a class called `MyCustomBackend` that is publicly derived from `CustomInterface`.

```
class MyCustomBackend : public iqsdk::CustomInterface
```

The user will then need to create a `device_id` which is a string that will refer to the device type, which must not be an identifier for an already defined backend. Existing reserved identifiers include `"IQS"`, `"Tensor_Network"`, `"QD_SIM"`, and `"Clifford"`. Then they will need to call `iqsdk::CustomSimulator::registerCustomInterface<MyCustomBackend>(device_id, args...)`

The template parameter to `registerCustomInterface()` is the class for the simulator, the first parameter is the `device_id`, and the rest of the parameters get passed into the constructor for `MyCustomBackend` (can be zero parameters).

As an example, if `MyCustomBackend` has a constructor taking in a single integer, the following is possible:

```
std::string device_id = "my_custom_device";
iqsdk::QRT_ERROR_T status = iqsdk::CustomSimulator::registerCustomInterface<MyCustomBackend>(device_id, 3);
```

Then, the user can use `iqsdk::DeviceConfig` to make an instance of the device.

```
iqsdk::DeviceConfig new_device_config(device_id);
iqsdk::CustomSimulator generic_simulator(new_device_config);
```

As usual, you call `ready()` to indicate the next quantum kernel gets run on the custom backend.

```
status = generic_simulator.ready();
```

To get access to the custom simulator class, you can call `getCustomBackend()`.

```
iqsdk::CustomInterface *custom_interface = generic_simulator.getCustomBackend();
```

Then you can dynamic cast it to your class.

```
MyCustomBackend *custom_simulator_instance = dynamic_cast<MyCustomBackend *>(custom_interface);
```

Here, after running a quantum kernel, you can call any function you have implemented for the class.

Alternatively, for single use purposes, it is possible to register and get a generic simulator in a single call.

```
iqsdk::CustomSimulator *generic_simulator = iqsdk::CustomSimulator::createSimulator<MyCustomBackend>("my_
↪custom_device", 3);
```

13.3 Methods

- `getCustomBackend()`

  ```
  iqsdk::CustomInterface *getCustomBackend();
  ```

  Gets the custom backend stored in the `CustomSimulator` object.

- `registerCustomInterface()`

```
template <typename T, typename... Ts>
  static QRT_ERROR_T registerCustomInterface(std::string device_id,
                                             Ts... args);
```

Registers the custom backend with the QRT.

- `createSimulator()`

```
template <typename T, typename... Ts>
  static CustomSimulator *createSimulator(std::string device_id, Ts... args);
```

Registers and creates a custom backend.

14.0 Python Interface

14.1 Introduction

The Python Interface provides users a way to run the Intel® Quantum SDK using Python3, through the `intelqsdk.cbindings` library. There are two modes for interacting with Python:

1. Write quantum circuits in OpenQASM 2.0 – write a quantum circuit, and convert that to a .cpp file that has `quantum_kernel` functions, compile, and use the `intelqsdk.cbindings` library to run the `quantum_kernel` functions and call APIs, all from within Python.
2. Write `quantum_kernel` functions in C++, compile to a `.so` file, and call APIs from Python.

The Python Interface is installed in a virtual environment placed alongside the compiler in the `virtualenv` directory. To run Python scripts using the `intelqsdk.cbindings` library, use either

```
$ source <path to Intel Quantum SDK>/virtualenv/bin/activate
```

or call the script with `python3` located at

```
$ <path to Intel Quantum SDK>/virtualenv/bin/python3
```

14.2 Python via OpenQASM 2.0

14.2.1 Procedure

Step 1: Write quantum programs

Using OpenQASM2.0, or alternatively, transpile the Python program into OpenQASM2.0, with the user's choice of quantum programming package. As long as the program can be turned into a `.qasm` file, the Bridge library will be able to translate it to a C++ source file for the Intel® Quantum SDK.

At the beginning of the Python script, include the following lines:

```
from intelqsdk.cbindings import *
loadSdk("/path/to/file.so", sdk_name)
```

`loadSdk``needs to be called before calling other functions or creating objects from ``intelqsdk.cbindings` library.

The `sdk_name` is the user-created reference string given to this `.so` library. Later on, when calling functions from this library or referencing `cbit`/`qbit` variables, pass this identifier to indicate which library to use. Users can also access multiple `.so` libraries to call functions or reference `cbit`/`qbit` variables from each library.

Step 2: Write the Python script

First, import several modules:

```
import intelqsdk.cbindings
from openqasm_bridge.v2 import translate
```

Next, use Bridge to translate the OpenQASM file to C++:

```
with open('example.qasm', 'r', encoding='utf8') as input_file:
    input_string: str = input_file.read()

translated: list[str] = translate(input_string, kernel_name='my_kernel')

with open('example.cpp', 'w', encoding='utf8') as output_file:
    for line in translated:
        output_file.write(line + "\n")
```

Now, compile the translated C++ code:

```
compiler_path = "<path to Intel Quantum SDK>/intel-quantum-compiler"
intelqsdk.cbindings.compileProgram(compiler_path, "example.cpp", "-s", sdk_name)
```

From here, the user can start calling APIs to set up the simulator and run the quantum program. For example,

```
iqs_config = intelqsdk.cbindings.IqsConfig()
iqs_config.num_qubits = 5
iqs_config.simulation_type = "noiseless"
iqs_device = intelqsdk.cbindings.FullStateSimulator(iqs_config)
iqs_device.ready()
```

(continues on next page)

(continued from previous page)

```
intelqsdk.cbindings.callCppFunction("my_kernel", sdk_name)
qbit_ref = intelqsdk.cbindings.RefVec()
for i in range(5):
    qbit_ref.append(intelqsdk.cbindings.QbitRef("q", i, sdk_name).get_ref())
probabilities = iqs_device.getProbabilities(qbit_ref)
intelqsdk.cbindings.FullStateSimulator.displayProbabilities(probabilities, qbit_ref)
```

14.3 Compiling `quantum_kernel` to Shared Library (`.so`)

14.3.1 Procedure

Step 1: Write `quantum_kernel` functions in C++

Given a C++ source file, `quantum_algorithm.cpp`,

Step 2: Compile the source program to `.so`

Compile the source program using the `-s` flag to compile to `<source_program>.so`. For example,

```
$ <path to Intel Quantum SDK>/intel-quantum-compiler -s quantum_algorithm.cpp
```

Alternatively, in the Python script, compile and load the `.so` file. It is assumed that the output directory is the same as the directory of the C++ file when loading the `.so` file.

```
intelqsdk.cbindings.compileProgram("<path to Intel Quantum SDK>/intel-quantum-compiler", "path/to/cpp_file
↪", "flags", sdk_name)
```

Step 3: Write the Python script which calls the APIs

Next, set up a simulation device by using the following template:

```
# number of qubits
N = 4
iqs_config = IqsConfig()
# set the number of qubits for the simulation config
iqs_config.num_qubits = N
# choose the type of noise model
iqs_config.simulation_type = "noiseless"
iqs_config.synchronous = False
iqs_device = FullStateSimulator(iqs_config)
iqs_device.ready()
```

Then, create a Python equivalent of the C++ objects used by `intelqsdk.cbindings`:

```
qids = RefVec()
cbits = []
for i in range(N):
    cbits.append(CbitRef("CReg", i, sdk_name))
    qids.append(QbitRef("QubitReg", i, sdk_name).get_ref())
```

Call APIs which form the quantum circuit:

```
# Prepare all qubits in the 0 state
callCppFunction("prepZAll", sdk_name)
# Apply QFT
callCppFunction("qft", sdk_name)
# Apply the inverse of QFT, effectively applying an Identity
callCppFunction("qft_inverse", sdk_name)
```

Call APIs to get the probabilities and measurement results:

```
probs = iqs_device.getProbabilities(qids)
amplitudes = iqs_device.getAmplitudes(qids)
callCppFunction("measZAll", sdk_name)

print("\nMeasurements:")
for cbit in cbits:
    print(cbit.value())

print("\nProbabilities printed with QRT API")
# Expect to see |0000> to have a probability of 1
# since an identity has been applied
FullStateSimulator.displayProbabilities(probs, qids)

# Required wait since device is asynchronous
iqs_device.wait()
```

14.3.2 How to get cbit values after running `quantum_kernel` functions?

Create a `CbitRef` object. For example, if there are the following global variables in the C++ source:

```
cbit c0;
cbit c_array[3];
```

then in the Python script, declare the following two variables:

```
cbit_c0 = intelqsdk.cbindings.CbitRef("c0", sdk_name)  # This refers to c0
cbit_c_array = intelqsdk.cbindings.CbitRef("c_array", 1, sdk_name) # This refers to c_array[1]
```

To get the value of `cbit`, call the `value()` function on the `CbitRef` object:

```
bool_val = cbit_c0.value() # returns a bool representing the value of the cbit
```

14.3.3 How to get references to qbit variables to pass to runtime APIs?

Create an `QbitRef` object. For example, if there are the following global variables in the C++ source:

```
qbit q_array[3];
qbit q0;
```

In the Python script, declare the following two variables:

```
qbit_q0 = intelqsdk.cbindings.QbitRef("q0", sdk_name) # This refers to q0
qbit_q_array = intelqsdk.cbindings.QbitRef("q_array", 2, sdk_name) # This refers to q_array[2]
```

Then `qbit_q0.get_ref()` returns an Python object that can be used as a `std::reference_wrapper<qbit>`.

Alternatively, make a `RefVec` to get a Python object that can be used as a `std::vector<std::reference_wrapper<qbit>>`. For example,

```
refvec = intelqsdk.cbindings.RefVec()
refvec.append(qbit_q0.get_ref())
refvec.append(qbit_q_array.get_ref())
```

Also, `qbit_q0.value()` returns the physical qubit mapped to by this program qubit.

```
print(qbit_q0.value())
```

14.3.4 Python objects and the corresponding C++ objects

```
DoubleVec - std::vector<double>
ComplexVec - std::vector<std::complex<double>>
SamplesVec - std::vector<std::vector<int>>
SampleVec - std::vector<int>
BoolVec - std::vector<bool>
IntVec - std::vector<int>
RefVec - std::vector<std::reference_wrapper<qbit>>
QssDoubleMap - QssMap<double>
QssComplexMap - QssMap<std::complex<double>>
QssIndexVec -  std::vector<QssIndex>
QssUnsignedIntMap - QssMap<std::unsigned int>
```

14.3.5 C++ classes that can be imported

```
QRT_ERROR_T
DeviceConfig
IqsConfig
TensorNetworkConfig
ErrSpec1Q
ErrSpec2Q
ErrSpecIdle
ErrorRates
GateTimes
CliffordSimulatorConfig
QssIndex
Device
FullStateSimulator
CustomInterface
TensorNetworkSimulator
CliffordSimulator
```

14.3.6 C++ functions that can be imported

```
qssMapToVector<double>
qssMapToVector<std::complex<double>>
qssMapVectorToMap<double>
qssMapVectorToMap<std::complex<double>>
```

14.3.7 Usage examples

Suppose the user has an instance of `intelqsdk.cbindings.FullStateSimulator` called `iqs_device`:

```
iqs_device.getProbabilities(qids) # returns a DoubleVec
iqs_device.getAmplitudes(qids) # returns a ComplexVec
iqs_device.getProbabilities(qids, QssIndexVec()) # returns a QssDoubleMap
iqs_device.getAmplitudes(qids, QssIndexVec()) # returns a QssComplexMap
iqs_device.getSamples(num_samples, qids) # returns a SamplesVec
iqs_device.getSingleQubitProbs(qids) # returns a DoubleVec
```

Example of using a map from C++:

```
#-- Iterating through a map in C++ gives a (key, value) pair --#
for prob in iqs_device.getProbabilities(qids, QssIndexVec()):
    print(prob.key().getIndex(), prob.data())
```

14.4 Using a Custom Backend with the Python Interface

To use a custom backend with the Python Interface, create a Python class that derives from the `CustomInterface` class in the `intelqsdk.cbindings` module. In this class, implement the `RXY`, `RZ`, `SwapA`, `CPhase`, `PrepZ`, and `MeasZ` methods, and add a constructor for the Python class.

The following example assumes the user has defined a Python class `MySimulator`:

```
custom_device_id = "custom device"
CustomSimulator.registerCustomInterface(MySimulator, custom_device_id, <args>) // args is optional,␣
↪depends on MySimulator's constructor
config = DeviceConfig(custom_device_id)
device = CustomSimulator(config)
```

Alternatively, if the user only intends to have one device, use the following shortcut:

```
device = CustomSimulator.createSimulator(MySimulator, "custom_device", <args>)
```

Note the difference relative to the C++ interface: instead of the class being a template argument of `registerCustomInterface` and `createSimulator`, it is the first parameter in the Python Interface.

Call the function `getCustomBackend()` to return the class that the user has created:

```
sim_object = device.getCustomBackend()
```

14.5 Known Limitations of the Python Interface

- Any variables of `cbit` type must be global in order to access them.
- The C++ functions, including `quantum_kernel`, called from Python must return `void` and either take no parameters or take an single array of `double`.

15.0 Running With MPI

15.1 MPI Support

The Intel® Quantum SDK leverages Message Passing Interface (MPI) in the qubit simulation backends for improved performance. It also provides users the option to run IQS simulations distributed across multiple compute nodes, enabling simulations involving larger numbers of qubits with the increased available memory.

15.2 Execution

To run the compiled executable, simply invoke it with

```
$ ./quantum_algorithm
```

If your program distributes IQS across multiple nodes of machines for distributed memory, launch the application with the `mpirun` command and use `-n` to specify the number of ranks. The total number of ranks must be a power of 2.

Here is an example command to run a program with 2 ranks.

```
$ mpirun -n 2 ./quantum_algorithm_iqs_distributed_mem
```

15.3 Sourcing compiler variables

This is required once per interactive session or once per job script for running the executable.

```
$ source /opt/intel/oneapi/setvars.sh
```

15.4 Known Limitations with MPI

Users can implement their own parallel code, but should not call `MPI_Finalize()`. Otherwise, Intel® Quantum SDK will call MPI functions after users' `MPI_Finalize()` call, which is not allowed.

While running a simulation with more than 35 qubits, the `displayAmplitudes()`, `getAmplitudes()`, `displayProbabilities()`, and `getProbabilities()` APIs might not work properly if the user tries to get all amplitudes or probabilities.

16.0 Running and Writing Custom Passes for the Intel® Quantum Compiler

16.1 Introduction

The Intel® Quantum SDK is built on the LLVM compiler's pass-based structure. The Intel® Quantum Compiler iteratively performs transformations of the program including optimization and lowering to hardware specific gates to compile a quantum program. This ordering can be changed, and extra transformations can be added to alter the compilation of the program. While the Intel® Quantum SDK has a defined set of transformations, there is room for extra passes to be added. The process of adding passes to the compilation flow from an external library are detailed below.

There are further details about how to access the development tools to create your own Intel® Quantum Passes as well.

16.2 Running Passes

As mentioned, the Intel® Quantum SDK provides a driver script with mechanisms to insert LLVM passes, passes from the Intel® Quantum SDK, or custom passes from external libraries, at specific points during the compilation flow of a quantum program.

16.2.1 Defining Custom Passes to Run

```
$ intel-quantum-compiler -E <user_defined_library> -e <compilation_stage_1> -a <pass_1>,...,<pass_n> -A
↪<pass_option> -A <pass_option> -e <compilation_stage_2>  -a <pass_1>,...,<pass_n> -A <pass_option> -A
↪<pass_option> <source_file>
```

Custom passes will be defined in libraries external to the Intel® Quantum Passes library. The `-E` flag with the path to the custom library as the argument, gives the compilation process access to the passes defined in the library. There are additional flags needed to define the compilation stage (when to run passes), pass lists (what passes to run), and a sequence's arguments that should be passed to those passes at each particular stage. First, a compilation stage is defined with the `-e` flag, followed by the name of one of the compilation stages listed below. Next, define the passes to be run during that compilation stage with the `-a` flag, followed by a comma-delimited list of pass names defining the passes that will be run during that stage. Then, optionally add the `-A` flag to pass each argument to the compiler invocation specified during the current `-e` specification. The arguments given to `-A` are concatenated together with a space between each argument. Compilation passes during other stages can be defined by using additional instances of the `-e` flag and its supporting options.

However, you do not have to specify an external library if the custom passes are already defined within the Intel® Quantum SDK or the LLVM Compiler. In this case, the `-E` flag and argument can be omitted. For example, to run the Dead Code Elimination pass from LLVM, the invocation would be:

```
$ intel-quantum-compiler -e <compilation_stage_1> -a dce <source_file>
```

16.2.2 Custom Pass Compilation Stages

There are five different stages where custom passes can be inserted:

- Preconditioning (`precond`)
- Presynthesis (`presynth`)
- Prelowering (`prelower`)
- Prescheduling (`presched`)
- Presplitting (`presplit`)

The first place that passes can be inserted into the pass pipeline is before the quantum program has been verified. At this point, you can expect there are no native quantum gates, and there should be no control flow constructs in the quantum kernels. Other optimizations expect a program with one execution within a quantum kernel. So, if additional control flow structures like loops or branching instructions have been added that cannot be handled by the native "flattening" optimizations provided by the SDK, they must be removed before this point with a custom pass. This is also a good place to replace your own custom functions with an intrinsic, or set of intrinsics if needed. This is called the "Precondition" section or `precond`.

The second stage where passes can be inserted is only available when using optimization level number 1. With this optimization there is extra synthesis of quantum programs. To insert a custom passes prior to synthesis, use the "Presynthesis" section or `presynth`.

The third stage where custom passes can be added is directly after the verification of the quantum program. If there are optimizations that act on the canonical gate set provided by the frontend, they should be performed here. At this point, you can expect that the remaining control flow structures will no quantum instructions. This is called the "Prelowered" section or `prelower`. If a custom placement pass needs to be written, it should be performed here.

The fourth stage stage where custom passes can be inserted is after the lowering of the quantum gates to the canonical gates for the device and the placement of the qubits on a device. At this point, there should be no canonical gates left, they should be replaced with native gate decompositions. If a gate was added that is not caught by the decomposition passes, this is the point to replace it with a native gate. Additionally, this is the stage before routing, and scheduling. This is called the Prescheduling section or `presched`. If a custom routing and scheduling pass needs to be written, it should be performed here.

The fifth and final opportunity is after routing, scheduling, but before the quantum kernels are separated from the rest of the program. At this point, the program acts on physical qubits and spin-native gates rather than the canonical gates. Any changes made at this point must honor the connectivity of a device. Passes that care about the physical qubits that the circuit is being run on should be made here. Or, if you only want to optimize what will be run, and need guarantees about which qubits you are acting on, or the kind of gate that is being used, this is the place to do it. This is the "Presplitting" section or `presplit`.

Putting all of this together, an example command line invocation from the an example external library, in this case from the open-source Intel® Quantum Compiler Passes repository, is:

```
$ intel-quantum-compiler -E path/to/libExampleMultiPass.so -e prelower -a print-all-gates,x-to-hzh,print-
↪all-gates -A -example-pass-opt -A testing -e presplit -a print-all-gates -A -example-pass-opt -A testing-
↪two <source_file>
```

This will run the sequence of passes "Print All Gates", "X to HZH" and "Print All Gates" during the prelowering stage of the compilation process along with the option `-example-pass-opt testing`. Then it will run the "Print All Gates" pass during the presplitting stage using the option `-example-pass-opt testing-two`.

16.3 The Open-Source Compiler Passes Repository

The Intel® Quantum Passes repository, found here, provide a mechanism for developers to add their own functionality and optimizations to the compilation process, or to modify the quantum passes to fit their needs.

This is not a necessary feature for most users of the Intel® Quantum SDK. Only developers looking to write and add their own features and optimizations to the quantum compilation process need to be aware of this section.

There are two main environments where building passes is supported, via the docker container or on the |Intel| Developer's Cloud.

The instructions for how to build the Quantum Passes can be found in the repositories themselves, but the basic steps are detailed here as well. This process is only necessary to change the operation of the passes themselves, or to add passes to the compilation process.

To build the repository you must have access to the Intel® Quantum SDK, CMake and either the Ninja or Make build systems.

The first step is to clone the Quantum Passes repository:

```
$ git clone https://github.com/intel/quantum-passes.git
$ cd quantum-passes
$ mkdir build
$ cd build
```

Then build the repository:

```
$ cmake -G Ninja -DLT_INTEL_QUANTUM_SDK_LOC=<sdk_install_dir> ../
$ ninja
```

The `sdk_install_dir` is the location where the Intel® Quantum SDK is installed on your system, this contains the necessary compiler tools and libraries to successfully build and run the Quantum Passes. If the Ninja build system is not available, use `-G "Unix Makefiles"` instead.

17.0 Code Samples

See the `quantum-examples` and `python-quantum-examples` directories in the Intel® Quantum SDK root directory for demonstrations of each of the included qubit simulators' APIs, demos, sample algorithm implementations and application simulations.

17.1 Algorithms and Simulations

`deutsch_jozsa_q7.cpp`:

An implementation of the Deutsch-Jozsa algorithm.

`qec_q5.cpp`:

An implementation of Quantum Error Correction (QEC) on 5 qubits.

`qft.cpp`:

An implementation of the Quantum Fourier Transform (QFT) and Inverse QFT algorithms.

`dynamic_mbl_q3.cpp`:

An implementation of Hamiltonian evolution simulating Many Body Localization (MBL).

`tfd_q4_hybrid_demo.cpp`:

A demonstration of generating Thermofield Double (TFD) state.

`teleport.cpp`:

A simulation of the procedure to teleport a quantum state.

`qkd_bb84.cpp`:

A simulation of establishing secure keys through Quantum Key Distribution (QKD) using the BB84 algorithm [BeBr1984].

`qnn_rus_n1.cpp`, `qnn_rus_nn1.cpp`:

Examples for simulating small Quantum Neural Networks (QNN).

17.2 Programming

`ghz.cpp`:

An implementation of creating a Greenberger-Horne-Zeilinger state (GHZ) using a template approach and compile time recursion to parameterize the number of qubits. The result is a `quantum_kernel` function that can be changed to simulate any number of qubits up to a predefined maximum number of qubits at compile time.

`dynamic_param.cpp`:

A demonstration of using dynamic parameters in `quantum_kernel` functions.

`custom_backend.cpp`:

An example of implementing a user-defined backend qubit simulator.

`iqs_custom_noise.cpp`:

An example for using the Intel® Quantum Simulator with a custom noise model.

`custom_backend_mimicking_iqs_custom_noise.cpp`:

An example implementing user-defined noise in qubits with the Intel® Quantum Simulator and comparing it to a user-defined backend that implements the same noise model.

`qexpr_ghz.cpp`, `qexpr_qft.cpp`, `qexpr_teleport.cpp`:

Re-implementations of preceding examples using FLEQ quantum kernel expressions (`QExpr`) to simplify and modularize the code.

`state_preparation.cpp`:

Uses a FLEQ `DataList` to prepare a list of qubits according to a string specification of `n` basis states.

`pauli_rotations.cpp`:

Uses a FLEQ `DataList` to prepare multi-qubit Pauli rotations, preparations, and measurements given a Pauli string specification.

`ideal_GHZ.cpp`, `sampled_GHZ.cpp`, `qd_GHZ.cpp`:

Several teaching examples demonstrating a development workflow. See Tutorials.

`iqs_vs_clifford_comparison.cpp`, `rep_code_clifford.cpp`:

A basic example and an advanced example for using the Clifford Simulator backend.

`run_ghz.py`, `run_qft.py`, `run_tfd_demo.py`:

Several examples demonstrating how to use the Python Interface. Each interacts with one of the above examples.

`api_<backend>_test.cpp`:

A demonstration of the API for each qubit simulator `<backend>`.

18.0 Summary of Known Limitations / Issues

- The maximum number of qubits supported is bounded by the total memory available to the Intel® Quantum Simulator and is a machine and application dependent quantity. See Getting Started Guide (Memory Requirements). The Tensor Network and Clifford Simulator backends are limited to 256 qubits.
- All operations on classical variables inside a `quantum_kernel` function will be executed at the beginning of that `quantum_kernel`, unless placed after the final quantum gate in the `quantum_kernel.` This applies to `quantum_kernel` functions called in the middle of other `quantum_kernel` functions, i.e. adding the return value of the interior `quantum_kernel` to an integer inside the higher scope `quantum_kernel` will be moved to the beginning of the resulting set of instructions. See **In-lining & quantum_kernel functions**.
- All source code must be located in a single `.cpp` file or included through header files.
- Top-level `quantum_kernel` functions cannot support `qbit` arguments. See **In-lining & quantum_kernel functions**.
- For `quantum_kernel` functions that use many qubit preparation operations, i.e. significantly more than the number of qubits used, use of `-O1` flag is known to dramatically slow down the compilation. See **Compiling**.
- Custom placement can only be used on global `qbit` variables. See **Placement**.
- If the scheduler pass `-S` flag is not set, the compiler assumes an all-to-all connection even if a non-all-to-all connectivity is given in the platform configuration `.json` file. Conversely, to invoke the `-S` flag, the `-c` flag must be given. See **Scheduling**.
- When the `-S` flag is not set and `-O1` optimization is set, some `quantum_kernel` functions may see additional `quswapalp` gate operations at the end of the `quantum_kernel`. See **Scheduling**.
- Users should not call `MPI_Finalize()` in the user program. Otherwise, MPI functions will be called after `MPI_Finalize()`, which is not allowed. See **Running with MPI**.
- When running a simulation with more than 35 qubits, the `display_` and `get_` APIs for the `FullStateSimulator` might not work properly if the user tries these methods to retrieve or show all `amplitudes` or `probabilities`. See **Running with MPI**.
- A compilation failure could occur if code which supports exception handling is invoked within a `quantum_kernel` function. The compilation error will likely be reported as a result of **invalid branching**. One such case would be the initialization of a quantum simulator within a `quantum_kernel` function. To avoid undesirable behavior, it is recommended to initialize the simulator in the `main()` function.

19.0 Support and Bug reporting

You can get technical support and report any bugs encountered by visiting |Intel| Communities. This is also a great place to ask questions and share ideas.

20.0 FAQ

20.1 Why is the amplitude of this state not the same as my by-hand calculation?

The amplitude of a state may differ between the result you compute when you work the problem by hand, algebraic solver, or other quantum computing tool chain. Take for example, the quantum circuit:

$$|0\rangle \text{ ——} \boxed{X}$$

You may be surprised to find the amplitude of this qubit is $-i\,|1\rangle$

```
Printing amplitude register of size 2
|0>     : (0,0)                        |1>     : (0,-1)
```

This is a consequence of the compiler being designed to compute in terms of the gate set for quantum dot qubits. The decomposition of X into the native gates gives a different, but physically equivalent, global phase than we might write doing the math by hand (where we implicitly assume our qubits directly support the gates in the textbook). The global phase will have no effect on observable quantities; i.e., the probability is still guaranteed to be computed correctly. To wit: the only outcome of a measurement on the above qubit is $|1\rangle$.

Inspecting the corresponding line in the `.qs` (quantum assembly file generated by the compiler) for the above gate shows the instruction given is

```
qurotxy QUBIT[0], 3.141593e+00, 0.000000e+00
```

The qurotxy native quantum dot gate was applied to the 0th qubit with the parameters π and 0. The matrix elements of this gate are

$$\begin{aligned} R_{xy}(\theta,\phi) &= \cos\left(\frac{\theta}{2}\right)\hat{I} - i\sin\left(\frac{\theta}{2}\right)\left[\hat{X}\cos\phi + \hat{Y}\sin\phi\right] \\ &= \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right)[\cos\phi - i\sin\phi] \\ -i\sin\left(\frac{\theta}{2}\right)[\cos\phi + i\sin\phi] & \cos\left(\frac{\theta}{2}\right) \end{bmatrix} \end{aligned}$$

and substituting in $\theta = \pi$ and $\phi = 0$, we find

$$\begin{aligned} X = R_{xy}(\pi,0) &= \begin{bmatrix} 0 & -i \\ -i & 0 \end{bmatrix} \\ &= -i\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{aligned}$$

So the $-i$ becomes a global phase, and will not contribute to a change in the probability of observing a given state.

20.2 What to do if I'm getting the "API called with qubits that are duplicated!" error?

This error is caused when the following scenario occurs:

```
qbit a;
qbit b;
qbit c;

quantum_kernel void example() {
    X(a);
    H(b);
    Y(c);
}

int main() {
    using namespace iqsdk;
    // Set up IQS device
    IqsConfig iqs_config;
    iqs_config.num_qubits = N;
    FullStateSimulator iqs_device(iqs_config);
    iqs_device.ready();

    example();

    std::vector<std::reference_wrapper<qbit>> qids =
        // This line will trigger the above error since qubit a is added to qids twice
        {std::ref(a); std::ref(a); std::ref(c)};
    std::vector<double> ProbabilityRegister;
    ProbabilityRegister = iqs_device.getProbabilities(qids);
}
```

To resolve this issue, ensure that each qubit is added exactly once. For example, replace the `qids` definition with:

```
std::vector<std::reference_wrapper<qbit>> qids =
    {std::ref(a); std::ref(b); std::ref(c)};
```

20.3 What to do if I'm getting the "1-qubit gate X on qubit Y is not available in the platform" error?

This is likely caused by compiling the source code with a platform configuration file that is incompatible with the choice of compilation flags and/or simulation backend. One solution is to recompile the source code with `-O1` flag. Alternatively, the source code can be recompiled with a different platform configuration file.

20.4 Where can I find the Intel Quantum SDK?

Depending on what system you are using the location of the Intel® Quantum SDK can vary. Throughout this document we have refered to this location as a generalized `<path to Intel Quantum SDK>/`. Below is a table of common paths where the Intel® Quantum SDK can be found.

Table 2: Common Paths to the Intel Quantum SDK

System Name	`<path to Intel Quantum SDK>`
Intel Developers Cloud	`/opt/intel/quantum_sdk/`
Docker Container	`/opt/intel/quantum-sdk/latest/`

Note: For convenience, consider appending the SDK path to your shell's `$PATH` environment variable. The typical `bash` syntax for this is:

```
export PATH=$PATH:<path to Intel Quantum SDK>
```

Bibliography

[BELL1964] Bell, J.S. (1964) On the Einstein Podolsky Rosen Paradox. Physics, 1, 195-200. https://doi.org/10.1103/PhysicsPhysiqueFizika.1.195

[EIPR1935] Einstein, A., Podolsky, B., & Rosen, N. Can quantum-mechanical description of physical reality be considered complete? Physical Review, 47(10), 777–780 (1935). https://doi.org/10.1103/PhysRev.47.777

[Schmitz2021] Schmitz, A. T., Sawaya, N. P., Johri, S., & Matsuura, A. Y. (2021). Graph optimization perspective for low-depth Trotter-Suzuki decomposition. arXiv:2103.08602 [quant-ph]. https://doi.org/10.48550/arXiv.2103.08602

[Paykin2023] Paykin, J., Schmitz, A. T., Ibrahim, M., Wu, X. C., & Matsuura, A. Y. (2023). PCOAST: A Pauli-based Quantum Circuit Optimization Framework. arXiv:2305.10966 [quant-ph]. https://doi.org/10.48550/arXiv.2305.10966

[Schmitz2023] Schmitz, A. T., Ibrahim, M., Sawaya, N. P., Guerreschi, G. G., Paykin, J., Wu, X. C., & Matsuura, A. Y. (2023). Optimization at the Interface of Unitary and Non-unitary Quantum Operations in PCOAST. arXiv:2305.09843 [quant-ph]. https://doi.org/10.48550/arXiv.2305.09843

[NICH2010] M. A. Nielsen and I. L. Chuang, Quantum Computation and Quantum Information: 10th Anniversary Edition (Cambridge University Press, 2010). https://doi.org/10.1017/CBO9780511976667

[GEAN2014] I. M. Georgescu, S. Ashhab, and F. Nori, Quantum simulation, Rev. Mod. Phys. 86, 153 (2014). https://link.aps.org/doi/10.1103/RevModPhys.86.153

[KPZC2022] R. Kotlyar, S. Premaratne, G. Zheng, J. Corrigan, R. Pillarisetty, S. Neyens, O. Zietz, T. Watson, F. Luthi, F. Borjans, L. Lampert, E. Henry, H. George, S. Bojarski, J. Roberts, A. Y. Matsuura, and J. S. Clarke, Mitigating Impact of Defects On Performance with Classical Device Engineering of Scaled Si/SiGe Qubit Arrays, in 2022 International Electron Devices Meeting (IEDM) (2022) pp. 8.4.1–8.4.4 https://doi.org/10.1109/IEDM45625.2022.10019382

[KWPH2022] Khalate, P., Wu, X.-C., Premaratne, S., Hogaboam, J., Holmes, A., Schmitz, A., Guerreschi, G. G., Zou, X. & Matsuura, A. Y., arXiv:2202.11142 (2022). https://doi.org/10.48550/arXiv.2202.11142

[BaSR2021] J. C. Bardin, D. H. Slichter, and D. J. Reilly, Microwaves in Quantum Computing, IEEE Journal of Microwaves 1, 403 (2021). https://doi.org/10.1109/JMW.2020.3034071

[ZKWL2022] Zwerver, A.M.J., Krähenmann, T., Watson, T.F. et al. Qubits made by advanced semiconductor manufacturing. Nat Electron 5, 184–190 (2022). https://doi.org/10.1038/s41928-022-00727-9

[LODI1998] Loss D., DiVincenzo D.P. Quantum computation with quantum dots. Phys Rev A, 57 (1) (1998), pp. 120-126 https://doi.org/10.1103/PhysRevA.57.120

[DIVI2000] D. P. DiVincenzo, The Physical Implementation of Quantum Computation, Fortschritte der Physik 48, 771 (2000). https://doi.org/10.1002/1521-3978(200009)48:9/11<771::AID-PROP771>3.0.CO;2-E

[LJLN2010] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. O'Brien, Quantum computers, Nature 464, 45 (2010). https://doi.org/10.1038/nature08812

[WAPK2018] Watson, T., Philips, S., Kawakami, E. et al. A programmable two-qubit quantum processor in silicon. Nature 555, 633–637 (2018). https://doi.org/10.1038/nature25766

[SURI2015] B. Suri, Transmon qubits coupled to superconducting lumped element resonators, Ph.D. thesis, University of Maryland College Park (2015). https://www.proquest.com/dissertations-theses/transmon-qubits-coupled-superconducting-lumped/docview/1702138107/se-2

[STEC2020] D. A. Steck, Quantum and Atom Optics (2020), revision 0.13.1. Accessed 05/01/2020. https://atomoptics.uoregon.edu/~dsteck/teaching/

[ZHSD2020] D. Zeuch, F. Hassler, J. J. Slim, and D. P. DiVincenzo, Exact rotating wave approximation, Annals of Physics 423, 168327 (2020). https://doi.org/10.1016/j.aop.2020.168327

[PARE2004] M. Paris and J. Řeháček, eds., Quantum State Estimation (Springer Berlin Heidelberg, 2004). https://doi.org/10.1007/b98673

[GHBS2020] Guerreschi, G. G., Hogaboam, J., Baruffa, F., & Sawaya, N. P. D., Intel Quantum Simulator: A cloud-ready high-performance simulator of quantum circuits. Quantum Science and Technology, 5, 034007 (2020). https://doi.org/10.1088/2058-9565/ab8505

[NEST2010] Van den Nest, M., Classical simulation of quantum computation, the Gottesman-Knill theorem, and slightly beyond. Quantum Info. Comput. 10, 3 (2010). https://doi.org/10.5555/2011350.2011356

[GOTT1998] Gottesman, D., The Heisenberg Representation of Quantum Computers, Group22: Proceedings of the XXII International Colloquium on Group Theoretical Methods in Physics, eds. S. P. Corney, R. Delbourgo, and P. D. Jarvis, (1999). https://doi.org/10.48550/arXiv.quant-ph/9807006

[SCGO2004] Scott Aaronson, S., Daniel Gottesman, D., Improved simulation of stabilizer circuits, Phys. Rev. A 70, 052328 (2004). https://link.aps.org/doi/10.1103/PhysRevA.70.052328

[HDER2006] Hein, M., Dür, W., Eisert, J., Raussendorf, R., Van den Nest, M., Briegel, H. -J., Entanglement in Graph States and its Applications, (2006). https://doi.org/10.48550/arXiv.quant-ph/0602096

[GEZH2013] Geller, M. R., Zhou, Z., Efficient error models for fault-tolerant architectures and the Pauli twirling approximation, Phys. Rev. A 88, 012314 (2013). https://link.aps.org/doi/10.1103/PhysRevA.88.012314

[SJDL2003] Strauch, F. W., Johnson, P. R., Dragt, A. J., Lobb, C. J.,Anderson, J. R.,Wellstood, F. C., Quantum Logic Gates for Coupled Superconducting Phase Qubits, Phys. Rev. Lett. 91, 167005 (2003). https://link.aps.org/doi/10.1103/PhysRevLett.91.167005

[BeBr1984] C.H. Bennett and G. Brassard, Quantum cryptography: Public key distribution and coin tossing, Proceedings of IEEE International Conference on Computers, Systems and Signal Processing, vol. 175, pg. 8, (1984). https://doi.org/10.1016/j.tcs.2014.05.025