Intel® Quantum SDK

Developer Guide and Reference

August 1, 2023

Release Version 1.0

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).

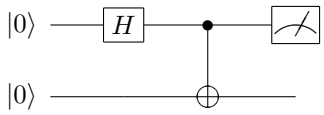
2.0 Overview

The Intel® Quantum Software Development Kit (SDK) is a high level programming environment that allows users to write software targeted for the Intel® quantum hardware. A host of backends including simulations and hardware are planned to support software development workflows.

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 to specifically target 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] [BELL964],



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

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

/* Quantum Runtime Library APIs */
#include <quantum.hpp>

#include <iostream>

/* Declare 2 qubits */
qbit q[2];

/* 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 setting of the simulator. */
    iqsdk::IqsConfig settings(2, "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** section 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.

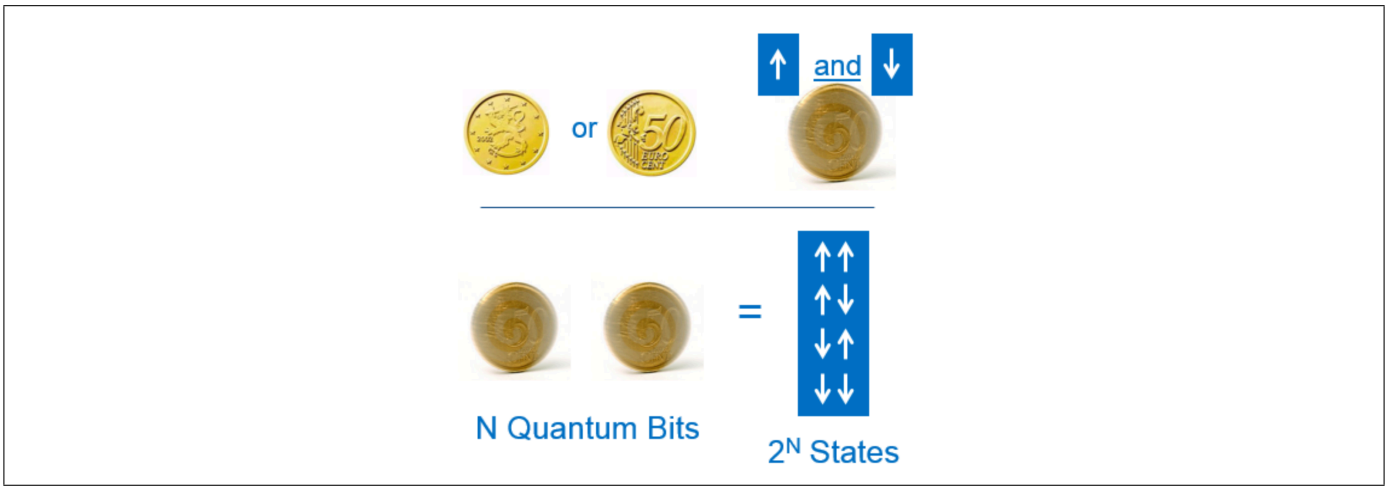
To summarize, the Intel® Quantum SDK includes:

1. an intuitive user interface based on the C++ programming language,
2. a sequence of optimizations and decompositions based on the LLVM compiler framework and specifically targeted at the Intel® Quantum Computing Stack,
3. a full compilation flow that produces an executable using the user's choice of backends compatible with Intel quantum hardware.

Authoring a quantum-accelerated application in the Intel® Quantum SDK closely follows the programming paradigm of other hardware accelerator development environments. Quantum programs are written in a C++ imperative programming environment that has been extended to allow the user to express quantum circuits at the level of primitive quantum logical operations. Depending on whether the user is targeting a simulation environment or qubit hardware, our quantum runtime library will direct the quantum workloads to the appropriate backend during runtime execution. There are currently two backends available to be built into the binary, the Intel® Quantum Simulator (IQS) and the Quantum Dot (QD) simulator. IQS provides a full-state-vector qubit simulation with a complete description of the quantum state of the qubits defined. The Quantum Dot simulator couples an equivalent state-vector simulation to a simulation of the physical quantum operations thereby offering a more realistic but computationally intense result.

3.0 Brief 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 in a classical computer 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.

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 somewhat 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 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 modulus (as in, $|\alpha|^2$ and $|\beta|^2$) represents the real-world probabilities 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 like 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**:

$$|\psi\rangle \;\text{——}\boxed{\nearrow}$$

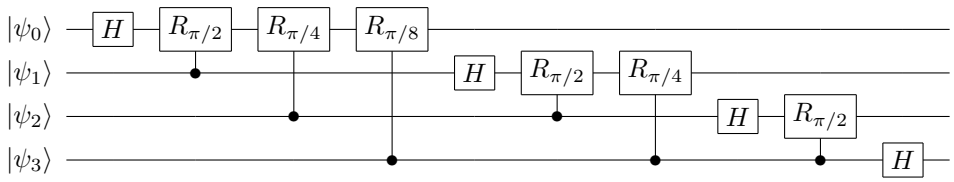
Immediately upon measuring the qubit's state, we will see that is it in either the $|0\rangle$ state or the $|1\rangle$ state. The probability the qubit is $|0\rangle$ is $|\alpha|^2$, and the probability the qubit is $|1\rangle$ is $|\beta|^2$.

Applying operations like a measurement 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, as represented by this circuit diagram:

$$|0\rangle \;\text{——}\boxed{X}\text{——}\; |1\rangle$$

Furthermore, qubits can be **entangled** with each other. That is, the combined state of multiple qubits contains more information than the qubits do independently. In our coin analogy above, 2 spinning coins can represent 4 possible states, 3 spinning coins can represent 8 states, and 4 coins can represent 16 states. This is also true for qubits; in general, N qubits can represent **2^N** possible states. With only 50 qubits, we can represent more states than any supercomputer could that is available today!

This massive amount of parallelism is what enables quantum computers to solve certain problems far more efficiently than a classical computer. Generally speaking, quantum computing is the exercise of manipulating arrays of qubits in parallel so as to solve these interesting problems. For instance, with only a $O(n^2)$ gates, specifically, the Hadamard and phase-shift gates (see the **quantum gates** and the **QFT sample** for more details), we can apply a discrete Fourier transform on $O(2^n)$ amplitudes. For example, the 4-qubit QFT circuit diagram looks like this:



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

4.0 Getting Started

From here, find information to get familiar with new tools and concepts, check on hardware **requirement**, and learn about **how to use** the SDK.

4.1 Fundamentals

To find out more about the core concepts of quantum computing, such as qubit mathematics and logic or quantum circuit representation, we recommend using a textbook such as

1. "Quantum Computation and Quantum Information" by Nielsen and Chuang [NICH2010]
2. "Quantum Computing for Computer Scientists" by Yanofsky and Mannucci [YAMA2008]
3. "Quantum Computing: An Applied Approach" by Hidary [HIDA2019]

To brush up on C++, we recommend the latest edition of "Tour of C++" by Bjarne Stroustrup [STRO2022]. In our code, we strive to adhere to the LLVM style standard to simplify working with our APIs.

4.2 Computing Ecosystem

The Intel® Quantum SDK is hosted on the Intel® DevCloud, a High-Performance Computing (HPC) cluster of Intel hardware. Working with this environment requires a set of skills separate from quantum computing knowledge. This knowledge base can be summarized as

- Connect to DevCloud with SSH
- Connect to DevCloud with Visual Studio Code
- Linux terminal
- HPC job submission

4.3 System Requirements

The Intel® Quantum SDK has the following memory requirements:

4.3.1 Memory Requirements

The state vector simulation requires an amount of memory increasing with the number of qubits. Representing n qubits requires 2 to the power of n complex numbers. Representing a complex `double` value requires 16 bytes, 2^4. So the single-node memory required to simulate n qubits is

$$\text{memory} = 2^{(4+n)} \text{ bytes}$$

As reference, it requires:

Table 1: Memory Requirements

qubits	memory
10	17 KB
20	17 MB
30	17.2 GB

and adding 1 additional qubit doubles the required memory. This memory requirement is in addition to the memory required by your data structures in the classical logic of your C++ application.

4.4 How to Use?

In the simplest terms, using the Intel® Quantum SDK follows the pattern:

1. Write your quantum algorithm with a **.cpp** extension as a separate file. In this file, you will define your `quantum_kernel` functions, initialize quantum hardware, and write and run your quantum and classical algorithms. (see **Writing New Algorithms** section for details) Example: `my_new_algo.cpp`
2. Build and Run. Invoke the **intel-quantum-compiler** script for using the SDK:

```
$ ./intel-quantum-compiler [compiler flags] my_new_algo.cpp
```

To see a list of the flag options, run

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

Run the executable produced in step 2

```
$ ./my_new_algo
```

When submitting a job to the DevCloud, the build and run steps can be performed individually or together in a job script, as shown below. See the DevCloud documentation website to find more details and examples for configuring your job submission script.

```
#PBS -S /bin/bash
#PBS -N my_first_job
#PBS -l walltime=01:00:00
## merge standard out and error
#PBS -j oe
## resource request
#PBS -l nodes=2:ppn=24
## change to submission directory
cd $PBS_O_WORKDIR

## compile the program; note "\" character at end to continue bash line
/glob/development-tools/intel-quantum-sdk/intel-quantum-compiler \
quantum_algorithm.cpp

## run the executable
./quantum_algorithm
```

See **Known Limitations and Issues** for the limitations.

5.0 Writing New Algorithms

Writing a new algorithm is as easy as writing C++ code. A basic template and example file can be found in `new_algo_start_here.cpp`. For a new `.cpp` file, the only additions needed to access quantum-specific functionality are:

1. Add the `#include <clang/Quantum/quintrinsics.h>` header which defines the quantum gates listed in **Supported Quantum Logic Gates**, and add the `<quantum.hpp>` header which defines the `FullStateSimulator` class.
2. Declare the variable(s) that represent your quantum algorithm's qubits as globally defined `qbit` data types.
3. Before invoking any function containing quantum instructions, prepare the backend that will serve as the quantum hardware by instantiating an object of the desired class. The `FullStateSimulator` class provides access to both IQS and QD Simulator as backends. For more details on using the class methods, see **Configuring the FullStateSimulator**, or see **Provided Classes & Methods** for a complete description of the class' methods. For details on using QD Simulator, see **Quantum Dot (QD) Simulator** section.
4. Place all calls to quantum gates inside a function or method, not raw in `main()`. Add the function specifier `quantum_kernel` to the definition of functions and methods including quantum gates or other `quantum_kernel` functions.

   These functions can declare and operate on classical (non-quantum) data types using typical scope rules. The compiler will move the classical calculation to have it available to a `quantum_kernel` while also preventing the classical instruction from being sent to quantum hardware. A `quantum_kernel` function can have parameters of `qbit` type as long as the logic ultimately resolves unambiguously to a globally defined `qbit` variable. See **In-lining & quantum_kernel functions** below for a more detailed description.
5. Just like any C++ program, use the `int main()` function to run your primary computation. Conceptually, your program is a hybrid of traditional or "classical" components and quantum components. The quantum components are sent to the quantum backend/device.

6.0 Supported Quantum Logic Gates

Below is a list of quantum logic gates and their signatures. To see the matrix definitions for these gates, refer the file

```
/<path>/<to>/<IntelQuantumSDK>/LLVM-10.0.0-Linux/include/clang/Quantum/quintrinsics.h
```

1. Hadamard (H)

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

2. Pauli X (X)

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

3. Pauli Y (Y)

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

4. Pauli Z (Z)

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

5. Phase (S)

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

   phase shift with phi = pi / 2.

6. Phase Inverse (Sdag)

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

   conjugate transpose of S.

7. T

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

   phase shift with phi = 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);
```

Two controls Toffoli gate.

16. PrepX

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

initializes/resets qubit to the '+' computational state.

17. PrepY

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

initializes/resets qubit to the 'R' computational state.

18. PrepZ

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

initializes/resets qubit to the '0' computational state.

19. MeasX

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

measures the qubit q in the '+' or '-' computational states and stores the result in c.

20. MeasY

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

measures the qubit q in the 'R' or 'L' computational states and stores the result in c.

21. MeasZ

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

measures the qubit q in the '0' or '1' computational states and stores 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.

24. Swap Alpha

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

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

7.0 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 Provided Classes & Methods.

7.1 Built-in Types & Intrinsic Functions

`qbit` data type for variables to represent qubits.

`cbit` data type for variables to represent a classical bit returned by a quantum measurement.

`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`. Calling `release_quantum_state` facilitates optimizations when compiled using the `-O1` flag, which can lead to a 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().

7.1.1 Known Limitations

All `qbit` type variables must be declared in the global namespace.

Placing variable of `cbit` type in STL containers can create a bug. You can instead use a STL container of `bool` or `int` type and convert measured `cbit` values for storage as a workaround.

7.2 Namespace

`iqsdk` a namespace providing access to the classes and methods of the Intel® Quantum SDK.

7.3 Includes & Classes

`<clang/Quantum/quintrinsics.h>` This header file is required, and 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.

`<quantum.hpp>` This header file is required, and provides access to the `FullStateSimulator` class as well as auxiliary helper classes.

`IqsConfig` configuration data specifically used to configure 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 Provided Classes & Methods for a complete description of the class' methods.

`<qrt_errors.hpp>` This header file is included by quantum.hpp. 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`.

`<qrt_indexing.hpp>` This header file is included by quantum.hpp. It defines the data types for facilitating access to the data of the `FullStateSimulator`.

`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.

8.0 In-lining & quantum_kernel functions

When the compiler prepares a `quantum_kernel`, it separates all the quantum details from the classical details so that it can deliver a complete set of instructions to the quantum hardware.

Local declarations and operations with traditional C++ data types are supported to aid readability and preserve programming concepts. But the compiler does the work to pull these "classical" instructions out of the `quantum_kernel`. This has a consequence on the `cbit` data type: Any operation done with `cbit` types written inside a `quantum_kernel` will occur as though they are 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 my_kernel() {
  cbit c = 0;
  std::cout << "c has value 0 here after initialization: "
            << (int)c << "\n";
  PrepZ(q0);
  X(q0);
  MeasZ(q0,c);
  std::cout << "c still has value 0 here since the quantum_kernel is not complete: "
            << (int)c << "\n";
  PrepZ(q1);
  std::cout << "After all gates in quantum_kernel have executed, c has value 1: "
            << (int)c << "\n";
}
```

A `quantum_kernel` may be called from within another `quantum_kernel`. Here, too, the compiler does the work of in-lining 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 `cbit` means that for `quantum_kernel` functions containing a `cbit` which are called in the middle of other `quantum_kernel` function, the `cbit` operations will be moved to the beginning of the resulting set of instructions.

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

```
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
```

(continues on next page)

(continued from previous page)

```
quantum_kernel void top_level_bell() {
  bell(q[0],q[2]);
}

int main() {

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

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

9.0 Measurements & FullStateSimulator

The `FullStateSimulator` class provides three main ways for obtaining statistical measurement results from simulators:

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

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 through its methods regardless of which backend is selected to run the simulation.

9.1 Simulation Data

Working with the simulation data returned by methods of the `FullStateSimulator` such as `getProbabilities` is often the most computationally efficient route to simulating a quantum algorithm because the probability is usually the quantity of interest that a quantum circuit must be run many times to compute.

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 many times for as many groupings of samples as an application calls for.

9.2 Combining Simulation Data and Measurement Operations

IQS offers an additional capability to combine simulation data and measurement operations. However, this feature is not available on QD Simulator because it doesn't collapse the state (see the **Quantum Dot (QD) Simulator**). This means combining results of measurement operations (through `cbit` values) and sampling results with the QD Simulator can yield unexpected results.

When combining measurements with simulation data, it is important to remember that when calling an explicit measurement operation, i.e. one of `MeasX`, `MeasY`, or `MeasZ`, on a qubit, 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) applied to a qubit 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 to be determined, such as for a Bell pair or GHZ state.

9.3 Using Only Measurement Operations

A third alternative 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. In the absence of noise models, this approach will produce (statistically) the same results as the equivalent sampling, particularly when scaled up to large sample sizes. Because taking measurements is how a quantum algorithm must be implemented on quantum hardware, in this respect, the simulation data and sampling approaches can be similar to a debugging mode for the latter measurement approach. IQS can support using measurements anywhere in the quantum algorithm; in contrast, QD Simulator supports a readout of measurements at the end of the `quantum_kernel`.

10.0 Output

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

(a) <algo-name>.ll : Intermediated 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.

(b) <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).

(c) <algo-name> : Executable corresponding to <algo-name>

This is the binary and ultimate result of the Intel® Quantum SDK.

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 measurement outcome is dependent on another outcome or set of outcomes, then that `cbit` value can ultimately be determined by the classical part of the IR (in particular, 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.

11.0 Code Samples

See the **quantum-examples** directory for a set of sample algorithms supported by this toolchain.

Some algorithmic implementations are shown,

1. Deutch-Josza algorithm
2. Greenberger-Horne-Zeilinger state (GHZ)
3. Quantum Error Correction (QEC)
4. Quantum Fourier Transform (QFT)
5. Many Body Localization (MBL)
    i. Fixed parameters
    ii. Dynamic parameters
6. Thermal Field Double (TFD)
7. Quantum Teleportation

There are also some functional examples:

1. All gates usage
2. Dynamic parameters usage

as well as some teaching examples that demonstrate a few different workflows for using the simulators as quantum computers and useful idioms useful for working with the `FullStateSimulator`:

1. Working with a state (GHZ, for example) using
    i. idealized data from the state vector from IQS
    ii. using sampling as an efficient stand-in for ensemble results
    iii. QD Simulator as a backend
2. API demonstrations with
    i. IQS as the backend
    ii. QD Simulator as the backend

12.0 Compiling

The compiler provides a number of flags to specify search paths for including header files and linking libraries as well as to redirect output files. These options can be printed by running

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

The `-v` verbose flag provides a summary of each `quantum_kernel` in terms of both the supported gates set and the quantum dot qubit gates set.

12.1 Compiler Optimization

Just as with any classical program compilation, the quantum compiler can look for opportunities to reduce the required instructions and/or order and execute them more optimally. This optimization takes into account logical and physical constraints, and can be activated by passing the following optimization options:

- `-O0`:

  This optimization flag represents no optimization at this time. This is the default if no flag is provided.

- `-O1`:

  This optimization flag enables high-level quantum optimizations to be performed 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 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, or analogous elements for Clifford operations and non-unitary quantum operations such as measurement and qubit preparation. Optimizations are then performed on this representation of the quantum logic and then used to synthesize a new circuit directly using native gates for the target backend and taking into account the minimization of two-qubit gate entanglement and overall depth. The synthesis methods are adapted from arXiv: 2103.08602.

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.

12.2 Qubit Placement and Scheduling

The backends of the Intel® Quantum SDK provide features to simulate quantum hardware of different levels of idealization. When using the FullStateSimulator backend to provide fully idealized quantum computer with unlimited physical connectivity between simulated "hardware" qubits, no decision is required regarding the arrangement of the `qbit` type objects in the source code onto the resources provided by the simulated quantum computer.

However, when working with practical quantum hardware (real or simulated), one real constraint is that some of the qubits we use as a resource in the software won't be connected to another, i.e. no possible two-qubit operation exists for them. In these cases, some algorithms will necessarily require moving the data-qubits (objects in user's source) around over the resource-qubits (physical qubits).

The Intel® Quantum SDK integrates the solution to this constraint into the quantum basic block functions it builds from your `quantum_kernel`. The placement compiler pass assigns program qubits as declared in user's source code to the physical qubits as defined in the configuration .json. This is the initial placement of the program qubits which may or may not change once the circuit has passed through the scheduler compiler pass. The scheduler compiler pass schedules the sequence of quantum instructions/gates. It takes the connectivity into consideration and adds supplementary quantum instructions required to implement the algorithm.

12.2.1 Placement

By default, the placement pass assumes an all-to-all connection between all physical qubits and assign the program qubits to physical qubits trivially, meaning program qubit 0 is assigned to physical qubit 0, program 1 to physical 1, and so on. If using the default mode, the `-c` flag (configuration file) can be left out while invoking the compiler.

For example,

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

In this one case, `-c` is not required and the placement pass assumes all-to-all connectivity and maps the program qubits trivially. In general, if the user wishes to use the `-p` flag, the `-c` flag must be given as well.

When a configuration json is given, the placement currently offers three placement methods.

1. Trivial - this method maps program qubits to physical ones trivially
2. Dense - this method maps the program qubits in a cluster of the highest connected portion of the given connectivity as defined in the configuration .json
3. Custom - the user can provide the placement they wish to use in their source code

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
```

If the user wishes to use custom placement, use 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 0 to physical 2, and program 1 to physical 0, and␣
↪program 2 to physical 1
```

Again, invoke the placement pass with the `-p` custom flag:

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

12.2.2 Scheduling

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

Currently, there's one scheduling method, `greedy`. To invoke the scheduler pass, use the `-S` flag:

```
$ ./intel-quantum-compiler -c configuration_file -S greedy 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 connection 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`.

Combining the `-p` and `-S` flag

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
```

Configuration files

The SDK comes with four configuration files that each represents the details of a different implementation of quantum hardware. They are:

- 6 qubit: Linear connectivity targeting the quantum dot simulator backend. DevCloud is able to simulate up to 6 qubits.
- 34 qubit: Linear connectivity targeting the IQS backend. DevCloud is able to simulate up to 34 qubits.
- 54 qubit: Linear connectivity targeting the IQS backend. DevCloud is able to simulate up to 34 qubits. For circuits with more than 34 qubits, DevCloud only supports compilation without invoking the IQS backend.
- 255 qubit: All-to-all connectivity targeting the IQS backend. DevCloud is able to simulate up to 34 qubits. For circuits with more than 34 qubits, DevCloud only supports compilation without invoking the IQS backend.

For usage with the quantum dot simulator, use `intel-quantum-sdk-QDSIM.json` which points to the 6 qubit configuration file. For usage with the IQS as the backend or just the compiler, use `intel-quantum-sdk.json`. By default, **intel-quantum-sdk.json** points to the 255 qubit configuration file. If you wish to use the 34 or 54 qubit configuration file, 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 on DevCloud.

Known limitations with the configuration files

The default wall clock limit for a job on DevCloud is 6 hours. Generally speaking, algorithms with 4 qubits using QD Simulator backend should be able to complete within 6 hours. If you wish to specify a maximum duration larger than the default 6 hours, modify your job submission script by following the example here. Lastly, the maximum duration allowed on DevCloud is 24 hours. Algorithms with 7 or more qubits using QD Simulator will not finish before DevCloud's maximum wall clock limit.

13.0 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 **Provided Classes and Methods (APIs)** for the full list of APIs to find details about retrieving data.

13.1 Brief 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.

13.2 Brief Overview of IqsConfig

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

- 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 a `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".

`bool use_custom_seed` Whether custom seed for RNG is used.

`std::size_t` seed Custom seed for RNG.

`double depolarizing_rate` Depolarizing rate for noisy simulation.

- isValid

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

Returns whether the given config instance is valid.

14.0 Provided Classes & Methods (APIs)

Below is a complete description of the provided classes and methods. These classes and methods facilitate working with and retrieving results from the `FullStateSimulator` class for manipulation in your C++ code.

14.1 reference_wrapper

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

Used to specify orders and subsets of qubits for reporting data from a full state quantum simulator. Reference wrappers are constructed by calling `std::ref(q)` for qubit variables `q`.

Some software tools could differ on their convention in writing the associated basis ket or vector. We use the more common ordering, with the first qubit of a quantum register being on the left-most side of the tensor product and the last qubit on the right-most side. For instance, for a 2-qubit array `qbit qregister[2]`, if the first qubit, `qregister[0]` is in state $|0\rangle$ and the second qubit, `qregister[1]` is in state $|1\rangle$, their joint state would be $|01\rangle$.

While not provided by the Intel® Quantum SDK, this idiom is important because the relationship between the qbit type variable and the hardware qubit isn't guaranteed at compile time. Conceptually similar to an object instance that does not necessarily occupy the same memory address during each execution.

14.2 QRT_ERROR_T type

An error type used when interacting with quantum devices.

- `enum` type

  ```
  iqsdk::QRT_ERROR_T
  ```

  `QRT_ERROR_T` is an `enum` which is either `QRT_ERROR_SUCCESS`, `QRT_ERROR_WARNING`, or `QRT_ERROR_FAIL`.

14.3 `FullStateSimulator` class

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

- Constructor

```
FullStateSimulator();
```

```
FullStateSimulator(Device_Config &device_config);
```

- isValid

```
bool FullStateSimulator::isValid();
```

- initialize

```
virtual QRT_ERROR_T FullStateSimulator::initialize(
    IqsConfig &device_config);
```

Initializes the simulator instance to the settings in `device_config`.

- ready

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

Specifies that the next `quantum_kernel` called will be run on this device.

- printVerbose

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

Specifies that device will be put or not put in verbose mode.

- wait

```
iqsdk::QRT_ERROR_T iqsdk::FullStateSimulator::wait();
```

Waits for device to stop running before continuing.

Useful for asynchronous mode, to ensure all `quantum_kernel` functions that are queued have finished running, and the appropriate `cbit` variables have been set. Note: APIs that get data from the device are synchronous. APIs that set device properties are asynchronous so users may not always need to call this.

- getProbabilities

```
std::vector<double> FullStateSimulator::getProbabilities(
    std::vector<std::reference_wrapper<qbit>>& qids);
```

Returns the conditional probabilities of the qubits given in `qids`. Expects `qids` to be a non-repeating list of qubit reference wrappers. Not all qubits need be included in `qids` vector, in which case `getProbabilities()` will return conditional probabilities. If using Intel Quantum Simulator as the back-end, expects `qids` to be used in a previously run `quantum_kernel`.

Useful for returning the conditional probabilities of certain qubits. For example, in a 4-qubit system $\{q_0, q_1, q_2, q_3\}$, the probability associated with `QssIndex("|11>")` with `qids={ref(q0); ref(q1)}` is the sum of the probabilities of being in state $|1100\rangle$, $|1101\rangle$, $|1110\rangle$, or $|1111\rangle$.

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

Returns a `QssMap` of `QssIndex` to `double`. The same conditions on `qids` apply as above. The `bases` argument is used to specify a subset of the states whose probability you are interested in. If the `bases` argument is nonempty, `getProbabilities()` will only return the probabilities associated with the given indices; if empty, it will return probability associated with each quantum state. The `threshold` argument is used to omit state,double pairs where the probability falls below the argument value. If a `threshold` is given, `getProbabilities()` will only return results whose probabilities are greater than or equal to the threshold.

Consider, for example, retrieving 2^N probabilities for large N. It can become a cumbersome operation. Specifying a subset of states in the bases argument will cause speed up in the execution time of `getProbabilities()` by reducing the total number of operations. Specifying a threshold argument will increase the computational overhead by performing a comparison of the probability and threshold.

- getProbability

```
double FullStateSimulator::getProbability(
    std::vector<std::reference_wrapper<qbit>> &qids,
    QssIndex basis);
```

Returns a single probability corresponding to the basis element given.

- amplitudesToProbabilities

```
static QssMap<double> FullStateSimulator::amplitudesToProbabilities(
    QssMap<std::complex<double>> &amplitudes);
```

Calling `amplitudesToProbabilities(getAmplitudes(...))` is equivalent to `getProbabilities(...)` but if a user wants to obtain both probabilities and amplitudes, this helper function can be useful.

- displayProbabilities

```
static void FullStateSimulator::displayProbabilities(
    std::vector<double> &probability,
    std::vector<std::reference_wrapper<qbit>> &qids);
```

Display to standard output the probability of observing the states of the qubits in `qids`.

```
static void displayProbabilities(QssMap<double> &probability);
```

Display to standard output the probability of observing each state represented by a `QssIndex` in `probability`.

- getSamples

```
std::vector<std::vector<bool>> FullStateSimulator::getSamples(
    unsigned int num_shots,
    std::vector<std::reference_wrapper<qbit>> &qids);
```

Returns a vector with `num_shots` different samples of measurement results for the given `qubits_ids`. Generates samples from the full state vector at the end of the last run quantum basic block. Each sample is a vector of length `qids.size()`, whose value at index `i` corresponds to the measurment result of the qubit `qids[i]`. Each `qbit` produces `false` for the measurement result of the `0` state, and `true` for the measurement result of the 1 state.

Efficiently sample from the full-state data in the simulator without having to repeat the gates to prepare that state `num_shots` number of times. Note that measurement with a MeasX, MeasY, or MeasZ collapses the quantum state, so if a qubit that has recently been measured is included in `qids`, this will always return the same value that previously resulted from measurement.

- getSingleQubitProbs

```
std::vector<double> FullStateSimulator::getSingleQubitProbs(
    std::vector<std::reference_wrapper<qbit>>& qids);
```

Returns a vector of single qubit probabilities of length `qids.size()`. The order the `single_qubit_probs` is given by the order of the qubits in the `qids` vector.

Useful if you want only the probabilities of each qubit being in state $|0\rangle$ or $|1\rangle$, ignoring entanglement with other qubits.

- getAmplitudes

```
std::vector<std::complex<double>> FullStateSimulator::getAmplitudes(
    std::vector<std::reference_wrapper<qbit>>& qids)
```

Returns the complex amplitudes of the state space with respect to the order given by `qids`. Expects `qids` to be a non-repeating list of all qubit ids in scope.

```
QssMap<std::complex<double>> FullStateSimulator::getAmplitudes(
    std::vector<std::reference_wrapper<qbit>>& qids,
    std::vector<QssIndex> bases,
    double threshold = -1)
```

As above; if the `bases` argument is nonempty, `getAmplitudes` will only return the amplitudes associated with the given indices. If a `` `threshold `` is given, `getAmplitudes` will only return results $a + bi$ such that $a^2 + b^2 >=$ threshold.

The optional arguments `bases` and `threshold` are useful for querying subsets of a state space. If a user only wants to query the amplitudes of a subset of basis elements, specifying those bases will be significantly more efficient in time and space. If a user only wants amplitudes above a threshold, specifying a threshold will significantly reduce the space required.

```
std::complex<double> FullStateSimulator::getAmplitude(
    std::vector<std::reference_wrapper<qbit>> &qids,
    QssIndex basis)
```

Returns a single amplitude corresponding to the basis element given.

- displayAmplitudes

```
static void displayAmplitudes(
    std::vector<std::complex<double>> &amplitudes,
    std::vector<std::reference_wrapper<qbit>> &qids)
```

Display to standard output the amplitude of the states of the qubits in `qids`.

```
static void displayAmplitudes(QssMap<std::complex<double>> &amplitudes);
```

Display to standard output a summary of the amplitudes for each state represented by a `QssIndex` in `amplitudes`.

14.4 QssIndex

The QssIndex (Quantum State Space Index) class is used to index into state spaces (std::vectors and QssMaps) returned by getProbabilities() and getAmplitudes().

- Constructor

```
QssIndex(std::string);
```

The input string basis has the form |b_0...b_n> or b0...bn where each b is either 0 or 1 e.g. "|100>" or "100".

Useful for specifying a state when you want to request data from a large state-space.

```
QssIndex(std::vector<bool> basis);
```

```
QssIndex(size_t num_qubits, size_t i);
```

- getIndex

```
size_t QssIndex::getIndex();
```

- getNumQubits

```
size_t QssIndex::getNumQubits();
```

- wellFormed

```
bool QssIndex::wellFormed();
```

Returns whether or not the QssIndex is correctly formed.

- toString

```
std::string QssIndex::toString();
```

Converts idx to a binary representation using num_bits bits, printed in reverse order. This relates the index idx into a probability or amplitude vector to its corresponding basis element.

- bitAt

```
bool QssIndex::bitAt(size_t i);
```

- setBitAt

```
bool QssIndex::setBitAt(size_t i, bool b);
```

- patternToIndices

```
static std::vector<QssIndex> QssIndex::patternToIndices(
    std::string pattern);
```

Given a pattern, produce a vector of QssIndex objects obtained by replacing X, ?, or Wildcard in the pattern by every combination of 0 and 1.

The patterns |X1X>, ?1? and {Wildcard, 1, Wildcard} all represent the same set of indices: {|010>, |110>, |011>, |111>}.

```
static std::vector<QssIndex> QssIndex::patternToIndices(
    std::vector<int> pattern);
```

Given a pattern of a `std::vector` of `ints` returns a set of `QssIndex` consistent with the pattern. Similar to above, integers other than 0 or 1 will be interpreted as a wildcard.

- operators

```
bool operator==(const QssIndex &idx) const;
bool operator!=(const QssIndex &idx) const;
bool operator<(const QssIndex &idx) const;
```

For comparing the total number of elements in two `QssIndex` variables.

14.5 QssMap

A wrapper around a C++ map indexed by a `QssIndex`, to double or complex. Returned by `getProbabilities()` and `getAmplitudes()`. Useful for holding a state space or partial state space of probabilities or amplitudes.

14.5.1 QssMap helper functions

Convert a description of qubit states between `std::vector` and `QssMap` for either `doubles` for probabilities or `std::complex` for amplitudes.

- qssMapToVector

```
template <class T>
std::vector<T> qssMapToVector(QssMap<T> &map,
                              size_t num_qubits,
                              T default_val);
```

- qssVectorToMap

```
template <class T>
QssMap<T> qssVectorToMap(std::vector<T> &vec,
                         size_t num_qubits);
```

15.0 Running With MPI

15.1 MPI Support

Intel® Quantum SDK supports Message Passing Interface (MPI) to allow users to run IQS on multiple nodes to increase the simulation size. We recommend using MPI for large-scale simulation. Using MPI for small-scale circuits might cause performance degradation due to the overhead of `MPI_Init()` and MPI communications.

15.2 Compilation with MPI-Enabled Libraries

To compile a quantum program with MPI-enabled libraries, use the `-m` flag to ask the compiler to look for Intel® MPI in its default install location and use it if available. Alternatively, to use a custom installation of MPI, use `-I`, `-L`, `-l` flags to pass the relevant info.

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

15.3 Execution with MPI Feature

To run the quantum algorithm with MPI with multiple ranks, 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 our program with 2 ranks.

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

15.4 Job Submission Script Example

Here is an example job submission script using mpi.

```
#PBS -S /bin/bash
#PBS -N my_quantum_algorithm
#PBS -l walltime=01:00:00
#PBS -o my_quantum_algorithm.out
#PBS -e my_quantum_algorithm.err
#PBS -l nodes=2:ppn=24
## change to submission directory
cd $PBS_O_WORKDIR

## compile the program
/glob/development-tools/intel-quantum-sdk/intel-quantum-compiler \
-m quantum_algorithm.cpp

## run the executable
mpirun -n 2 ./quantum_algorithm
```

15.5 Known Limitations with MPI

Users should not call `MPI_Finalize()` in the user program. Otherwise, MPI functions will be called after `MPI_Finalize()`, which is not allowed.

Running a simulation with more than 35 qubits, the display and getAmplitudes APIs might not work properly if the user tries to get all amplitudes or probabilities.

16.0 Circuit Figures & LaTeX

The circuit printer feature produces a .tex file written for the LaTeX typesetting language for each `quantum_kernel`. A TeX distribution on your local machine with the qcircuit package (maintained at the Comprehensive TeX Archive Network) and its dependencies are all needed 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 will potentially be able to extend the diagram.

17.0 Intel Quantum Dot Qubit Gates

Because of differences between physical implementations of qubit systems, some physical systems will find it easier to implement certain quantum gates. 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 also. 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 state.

2. `qumeasz(qbit q)`

   Measurement in the Z basis (with collapse to measured state).

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 $|11>$ state of input qubits.

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

   An arbitrary rotation of `theta` in the $|01>, |10>$ state subspace.

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

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

18.0 Quantum Dot Simulator Backend

Quantum Dot Simulator (QD Simulator) is a simulator reproducing the physics of quantum processors 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 for Version 1.0 of the Intel® Quantum SDK [KWPH2022] [KPZC2022].

18.1 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 by [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.

With QD Simulator as the backend, the simulation goes through the a qubit control processor, the control electronics, and the 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. A simulator aware of the effects of multiplexed microwave drive pulses on electron spins is a key tool to aid the design of practical applications tailored for the quantum dot qubit technology.

In the current implementation these simulated qubits can be controlled via the single-qubit operations $R_{xy}(\theta, \phi)$

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

and the two-qubit operation CZ

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

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

18.2 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 these simulated quantum dots [KPZC2022], we are not in the rotating frame, nor using the rotating wave approximation. A future version of the Intel Quantum SDK should support these techniques. 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.

18.3 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 its full evolution history during algorithm execution. Since this detailed history is currently unavailable to users, the use of the latter function for full state characterization is discouraged.

This further highlights how close 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] should be required to reconstruct the full state when using QD Simulator.

18.4 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_SIM` `DeviceConfig`.

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

Once the simulator is configured, then you can call your `quantum kernel` functions to perform simulations on the QD Simulator.

18.5 Important Points on Quantum Dot Simulator

Because the quantum dot 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 group of 1-qubit and 2-qubit operations are applied
- the final probabilities or amplitudes for each basis state are retrieved.

There should be no continuity expected between `quantum_kernel` functions called within `main()`, since each time a `quantum_kernel` is called within `main()`, the QD Simulator history is reset and qubits will all 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 have an impact with QD Simulator, since the qubits will always be reset to `"|0>"` when starting a simulation. However, using `PrepZ` in the middle of simulating a `quantum_kernel` on QD Simulator will result in unexpected behavior (as is the case for `MeasZ`).

Note that Z rotations are currently not natively enabled for the hardware in simulation. Hence a user wishing to use RZ(θ), would implement it in one of two ways:

Recommended method. If using compiler optimization `O1`, then there is no extra step required. The compiler will absorb all RZ operations into other single-qubit operations.

If not using compiler optimization `O0`, then it is necessary to explicitly decompose the RZ operation (or related operations such as S, T, etc.) 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);
}
```

18.5.1 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.

Known limitations with the configuration files

The default wall clock limit for a job on DevCloud is 6 hours. Generally speaking, algorithms with 4 qubits using the QD Simulator backend should be able to complete within 6 hours. If you wish to specify a maximum duration larger than the default 6 hours, modify your job submission script by following the example here. Lastly, the maximum duration allowed on DevCloud is 24 hours. Algorithms with 7 or more qubits using the QD Simulator will not finish before DevCloud's maximum wall clock limit.

18.6 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. For example,

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

18.7 Package Installation for Quantum Dot Simulator on Intel® DevCloud

All the required packages are pre-installed on the Intel® DevCloud.

18.8 Sourcing compiler variables:

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

```
$ source /opt/intel/oneapi/mpi/latest/env/vars.sh
$ source /opt/intel/oneapi/setvars.sh
```

19.0 Support for OpenQASM 2.0

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

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

20.0 Python Interface

20.1 Introduction

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

1. Write quantum circuits in openqasm 2.0 – write a quantum circuit to a file, and convert that to a .cpp file that has `quantum_kernel` functions, compile, and use the `iqsdk 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 virtualenv placed alongside the compiler. To run your Python scripts using the **iqsdk** library, you can use

```
$ source /glob/development-tools/intel-quantum-sdk/virtualenv/bin/activate
```

or call your script with the python3 at

```
$ /glob/development-tools/intel-quantum-sdk/virtualenv/bin/python3
```

20.2 Python via OpenQASM 2.0

20.2.1 Procedure

Step 1: Write quantum programs

Using OpenQASM2.0, or alternatively, transpile your python program into OpenQASM2.0, with your choice of quantum programming package. As long as your 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.

Step 2: Write your python script

First, we will import several modules

```
import iqsdk
from openqasm_bridge.v2 import translate
```

Next, we will 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-compiler"
iqsdk.compileProgram(compiler_path, "example.cpp", "-s")
```

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

```
iqs_config = iqsdk.IqsConfig()
iqs_config.num_qubits = 5
iqs_config.simulation_type = "noiseless"
iqs_device = iqsdk.FullStateSimulator(iqs_config)
iqs_device.ready()
iqsdk.callCppFunction("my_kernel")
qbit_ref = iqsdk.RefVec()
for i in range(5):
    qbit_ref.append(iqsdk.QbitRef("q", i).get_ref())
probabilities = iqs_device.getProbabilities(qbit_ref)
iqsdk.FullStateSimulator.displayProbabilities(probabilities, qbit_ref)
```

20.3 Compiling quantum_kernel to .so

20.3.1 Procedure

Step 1: Write quantum_kernel functions in C++

Let's suppose that you have already written 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/IQSDK/intel-quantum-compiler -s quantum_algorithm.cpp
```

Alternatively, in your python script, compile and load the .so file. It is assumed that the output directory is the same as the directory to the C++ file when loading the .so file.

```
iqsdk.compileProgram("/path/to/compiler", "path/to/cpp_file", "flags")
```

Step 3: Write your python script which calls the APIs

At the beginning of your python script, include the following lines:

```
from iqsdk import *
loadSdk("/path/to/so/file")
```

You need to call `loadSdk` before calling functions or creating objects from `iqsdk` library.

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, we need to create python equivalent of the C++ objects used by `iqsdk`

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

Call APIs which form the quantum circuit

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

Call APIs to get the probabilities and measurement results

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

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 we have applied an identity
FullStateSimulator.displayProbabilities(probs, qids)

# No-op since device is synchronous
iqs_device.wait()
```

20.3.2 How to get cbit values after running quantum_kernel functions?

Create an `CbitRef` object. For example, if we have the following global variables in the C++ source

```
cbit c[3];
cbit c2;
```

then in Python script, we will need to declare the following two variables

```
cbit_c = iqsdk.CbitRef("c", 1) # This refers to c[1]
cbit_c2 = iqsdk.CbitRef("c2")  # This refers to c2
```

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

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

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

Create an `QbitRef` object. For example, if we have the following global variables in the C++ source

```
cbit c[3];
cbit c2;
```

In Python script, we will need to declare the following two variables

```
qbit_q = iqsdk.QbitRef("q", 2) # This refers to q[2]
qbit_q2 = iqsdk.QbitRef("q_2") # This refers to q_2
```

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

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

```
refvec = iqsdk.RefVec()
refvec.append(qbit_q.get_ref())
refvec.append(qbit_q2.get_ref())
```

20.3.4 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>
```

20.3.5 C++ classes that can be imported

```
QRT_ERROR_T
DeviceConfig
IqsConfig
QssIndex
Device
FullStateSimulator
```

20.3.6 C++ functions that can be imported

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

20.3.7 Usage examples

Suppose the user has an instance of `iqsdk.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 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())
```

20.3.8 Known Limitations with Python Interface

- Any variables of `cbit` type must be global in order to access them.
- Only one .so file may be loaded per program at this time.
- The C++ functions, including `quantum_kernel`, called from python must return `void` and either take no parameters or take an single array of `double`.

21.0 Summary of Known Limitations / Issues

- 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 **Memory Requirements**
- All `qbit` type variables must be declared in the global namespace.
- Any operation done with `cbit` types written inside a `quantum_kernel` will occur as though they are at the beginning of that `quantum_kernel`, unless they are written 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 `cbit` to an integer higher in scope will be moved to the front beginning of the result set of instructions. See **In-lining & quantum_kernel functions**.
- All source code must be located in a single cpp file.
- Placing variable of `cbit` type in STL containers can create a bug. You can instead use a STL container of `bool` or `int` type and convert measured `cbit` values for storage as a workaround.
- 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**.
- 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 config .json. 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**.
- Intel® DevCloud only allows 2 total nodes per job at this time.
- The default wall clock limit for a job on DevCloud is 6 hours. Generally speaking, algorithms with 4 qubits using QD Simulator backend should be able to complete within 6 hours. If you wish to specify a maximum duration larger than the default 6 hours, modify your job submission script by following the example here. Lastly, the maximum duration allowed on DevCloud is 24 hours. Algorithms with 7 or more qubits using QD Simulator will not finish before DevCloud's maximum wall clock limit.

22.0 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.

23.0 FAQ

23.1 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)
```

Basically, this is the 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 for the above gate in shows the instruction given is

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

The qurotxy quantum dot qubit gate was applied to the 0th qubit and the parameters given were 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.

23.2 What to do if I'm getting the "API called with qubits not yet used!" error?

This error is caused when the following scenario occurs:

```
N = 5;
qbit q[N];

// A quantum_kernel where q[0], q[1] and q[2] are used
quantum_kernel void example() {
    X(q[0]);
    H(q[1]);
    Y(q[2]);
}

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();

    // Run the quantum kernel where q[0], q[1] and q[2] are used
    example();

    std::vector<std::reference_wrapper<qbit>> qids;
    for (int qubit = 0; qubit < N; ++qubit) {
        qids.push_back(std::ref(QubitReg[qubit]));
    }

    std::vector<double> ProbabilityRegister;
    // This API call can trigger the error since q[3] and q[4]
    // haven't been used
    ProbabilityRegister = iqs_device.getProbabilities(qids);
}
```

To resolve this issue, ensure that the API is called on used qubits only. For example:

```
N = 5;
qbit q[N];

// A quantum_kernel where q[0], q[1] and q[2] are used
quantum_kernel void example() {
    X(q[0]);
    H(q[1]);
    Y(q[2]);
}

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();

    // Run the quantum kernel where q[0], q[1] and q[2] are used
    example();
```

(continues on next page)

(continued from previous page)

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

    // Add used qubits to the reference wrapper
    for (int qubit = 0; qubit < 3; ++qubit) {
        qids.push_back(std::ref(QubitReg[qubit]));
    }

    std::vector<double> ProbabilityRegister;
    // This API call is now only acting on q[0], q[1] and q[2]
    ProbabilityRegister = iqs_device.getProbabilities(qids);
}
```

23.3 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)};
```

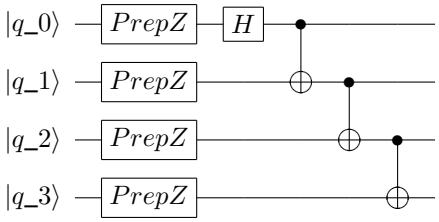
23.4 What to do if I'm getting the "1-qubit gate x on qubit x is not available in the platform" error?

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.

24.0 Tutorials

24.1 A Tour of GHZ examples

This tutorial provides some basic introduction to programming quantum algorithms and working with the `FullStateSimulator` through three implementations of the following circuit



and the accompanying classical logic. The executable will manage running the above quantum circuit and retrieving the data in the `FullStateSimulator` to confirm the instructions are producing what is intended. This circuit produces a Greenberger-Horne-Zeilinger state for 4 qubits. In other words, the qubits will be in a super-position of two states, one with all qubits in the $|0\rangle$ state and one with all qubits in the $|1\rangle$ state, and each with equal chance of being measured.

The source code for each implementation is available in the `quantum_examples/` directory of the Intel® Quantum SDK.

24.1.1 Ideal GHZ

To begin writing the program, get access to the quantum data types and methods by including the `quintrinsics.h` and the `quantum.hpp` headers into the source as shown in Listing 1.

Listing 1: The headers required to use the quantum gates or `FullStateSimulator`.

```
16 #include <clang/Quantum/quintrinsics.h>
17
18 /// Quantum Runtime Library APIs
19 #include <quantum.hpp>
```

As described in **Writing New Algorithms**, next declare an array of qubits (`qbit`) in global namespace as show in Listing 2

Listing 2: Declaration of the `qbit type variables in global namespace.

```
21 const int total_qubits = 4;
22 qbit qubit_register[total_qubits];
```

Next, implement the quantum algorithm by writing the functions or classes that contain the quantum logic. The `quantum_kernel` keyword should be in front of each function that builds the algorithm. Shown in Listing 3, all the quantum instructions are placed in a single `quantum_kernel`. The first instruction is to initialize the state of each qubit by utilizing a `for` loop to call `PrepZ()` on the elements of the `qbit` array. Next, apply a Hadamard gate (`H()`) on the first qubit to set it into a superposition. And lastly, apply Controlled Not gates (`CNOT()`) on pairs of qubits to create the entanglement between them.

Listing 3: `quantum_kernel` to prepare a Greenberger-Horne-Zeilinger state.

```
24 quantum_kernel void ghz_total_qubits() {
25   for (int i = 0; i < total_qubits; i++) {
26     PrepZ(qubit_register[i]);
27   }
28
29   H(qubit_register[0]);
30
31   for (int i = 0; i < total_qubits - 1; i++) {
32     CNOT(qubit_register[i], qubit_register[i + 1]);
33   }
34 }
```

In the `main` function of the program, instantiate a `FullStateSimulator` object and use its `ready()` method, as shown in Listing 4. Do that as described in **Configuring the FullStateSimulator**, by first creating an `IqsConfig` object, changing it to be verbose (not required), and then using it as an argument to the `FullStateSimulator` constructor. Then use the return of the `ready()` method to trigger an early exit if something is wrong.

Listing 4: Setup of the FullStateSimulator. See **Configuring the FullStateSimulator** for a more detailed explanation.

```
36 int main() {
37   iqsdk::IqsConfig settings(total_qubits, "noiseless");
38   settings.verbose = true;
39   iqsdk::FullStateSimulator quantum_8086(settings);
40   if (iqsdk::QRT_ERROR_SUCCESS != quantum_8086.ready())
41     return 1;
```

Next, prepare the classical data structures for working with simulation data. Create a set of `id` values to refer to the qubits of interest as shown in Listing 5. This is important because `qbit` variables do not necessarily specify a constant physical qubit in hardware; this mapping can be created and changed at various stages during program execution according to the compiler optimizations.

Listing 5: Declare and fill a `std::vector` with references to the elements of the `qbit` array.

```
43  // get references to qbits
44  std::vector<std::reference_wrapper<qbit>> qids;
45  for (int id = 0; id < total_qubits; ++id) {
46    qids.push_back(std::ref(qubit_register[id]));
47  }
```

Now that the backend simulator is ready, call the `quantum_kernel`.

Listing 6: Call the `quantum_kernel`.

```
49  ghz_total_qubits();
```

After this line during execution, the `FullStateSimulator` contains the state-vector data describing the qubits. Although there are facilities to conveniently specify every possible state for a set of qubits, this could be an overwhelming amount of data. Where possible, use the available constructors described in **Provided Classes & Methods** to configure a `QssIndex` object to refer to only the states of interest and store them in a vector. In Listing 7, two explicit strings formatted for 4 qubits are used to specify the two states.

Listing 7: Specifying the states of interest; this will be an argument used to collect data from the simulation.

```
51  // use string constructor of Quantum State Space index to choose which
52  // basis states' data is retrieved
53  iqsdk::QssIndex state_a("|0000>");
54  iqsdk::QssIndex state_b("|1111>");
55  std::vector<iqsdk::QssIndex> bases;
56  bases.push_back(state_a);
57  bases.push_back(state_b);
```

Now access the probabilities for these two states through `FullStateSimulator`'s `getProbabilities()` method. Remember that ideally the GHZ state would only be in $|0000\rangle$ or $|1111\rangle$. So for this simulated circuit, when summing the probabilities of measuring in either $|0000\rangle$ or $|1111\rangle$, the sum should be 1.0.

Listing 8: The classical logic that checks a property of the state in the qubit register. This gets quantum results from simulation data at runtime.

```
59  iqsdk::QssMap<double> probability_map;
60  probability_map = quantum_8086.getProbabilities(qids, bases);
61
62  double total_probability = 0.0;
63  for (auto const & key_value : probability_map) {
64    total_probability += key_value.second;
65  }
66  std::cout << "Sum of probability to measure fully entangled state: "
67            << total_probability << std::endl;
```

Alternatively, display a formatted summary of the states to the console with `displayProbabilities()` or `displayAmplitudes()`.

Listing 9: Generate a quick, formatted display of the probabilities for the two states of interest printed to console.

```
68  quantum_8086.displayProbabilities(probability_map);
```

This program can be compiled with the default options. Configuring the qubit simulation in the Intel Quantum Simulator creates a set of qubits with no limitation on their connectivity, in other words two-qubit operations (gates) can be applied between any pair of qubits.

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

Running the executable produces a line describing each quantum instruction because the verbose option was set to true when configuring the `FullStateSimulator`. That is followed by the line showing that `total_probability` is equal to 1 and the summary produced by `displayProbabilities()`.

```
$ ./ideal_GHZ

- Run noiseless simulation (verbose mode)
Instruction #4 : ROTXY(phi = 1.5707, gamma = 1.57075)  on phys Q 0
Instruction #5 : ROTXY(phi = 0, gamma = 3.14154)  on phys Q 0
Instruction #6 : ROTXY(phi = 4.71229, gamma = 1.57075)  on phys Q 1
Instruction #7 : CPHASE(gamma = 3.1415) on phys Q0 and phys Q1
Instruction #8 : ROTXY(phi = 1.5707, gamma = 1.57075)  on phys Q 1
Instruction #9 : ROTXY(phi = 4.71229, gamma = 1.57075)  on phys Q 2
Instruction #10 : CPHASE(gamma = 3.1415) on phys Q1 and phys Q2
Instruction #11 : ROTXY(phi = 1.5707, gamma = 1.57075)  on phys Q 2
Instruction #12 : ROTXY(phi = 4.71229, gamma = 1.57075)  on phys Q 3
Instruction #13 : CPHASE(gamma = 3.1415) on phys Q2 and phys Q3
Instruction #14 : ROTXY(phi = 1.5707, gamma = 1.57075)  on phys Q 3
Sum of probability to measure fully entangled state: 1
Printing probability map of size 2
|0000>  : 0.5                          |1111>  : 0.5
```

The complete code is available as `ideal_GHZ.cpp` in the `quantum_examples/` directory of the Intel® Quantum SDK.

24.1.2 Sampled GHZ

As discussed in Measurements & FullStateSimulator, the state-vector probabilities that the above program uses are not data that quantum hardware is capable of returning. Consider the hypothetical scenario in which you now need to know how many times the quantum circuit must be run and evaluated in order to find the probability with the desired numerical accuracy. This can be done efficiently by using the simulation data.

Only the non-`quantum_kernel` sections of the `ideal_GHZ.cpp` program need changed to accomplish this. This can be done by using a different `FullStateSimulator` method after calling the `quantum_kernel`, as shown in Listing 10

Listing 10: code_samples/sample_GHZ.cpp

```
71  // use sampling technique to simulate the results of many runs
72  std::vector<std::vector<bool>> measurement_samples;
73  unsigned total_samples = 1000;
74  measurement_samples = quantum_8086.getSamples(total_samples, qids);
```

Each `std::vector<bool>` represents the observation of a state, each individual value represents the outcome of that qubit as arranged in the `qids` vector. The `FullStateSimulator` has a helper method to conveniently calculate the number of times a given state appears in an ensemble of observations.

Listing 11: code_samples/sample_GHZ.cpp

```
76  // build a distribution of states
77  iqsdk::QssMap<unsigned int> distribution =
78          iqsdk::FullStateSimulator::samplesToHistogram(measurement_samples);
```

From this `QssMap`, calculate the estimate of the probability of observing each state.

Listing 12: The classical logic inspecting the results of sampling many simulated measurements.

```
80  // print out the results
81  std::cout << "Using " << total_samples
82            << " samples, the distribution of states is:" << std::endl;
83  for (const auto & entry : distribution) {
84    double weight = entry.second;
85    weight = weight / total_samples;
86
87    std::cout << entry.first << " : " << weight << std::endl;
88  }
```

This program can be compiled with the same command as the previous program was above.

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

In addition to the same output from `ideal_GHZ.cpp`, the result of the `samplesToHistogram()` method is also printed out:

```
Using 1000 samples, the distribution of states is:
|0000> : 0.496
|1111> : 0.504
```

Because the `IqsConfig` used the default setting for the random seed (by not giving one), this simulation will produce a different sequence of samples on subsequent executions and thus a slightly different estimate of the probability of observing each state.

The complete code is available as `sample_GHZ.cpp` in the `quantum_examples/` directory of the Intel® Quantum SDK.

24.1.3 GHZ state on Quantum Dot Simulator

It takes the change of only a few lines of code to target another backend, as you can see by using the `diff` tool to compare `sample_GHZ.cpp` and `qd_GHZ.cpp`. As described in **Quantum Dot Simulator Backend**, the Quantum Dot Simulator (`QD_Sim`) adds to the state vector simulation by including details from the control signals that interact with quantum dot qubits. This creates additional computational overhead when compare to the Intel® Quantum Simulator. The first change in this program is to reduce the size of the circuit to just a pair of qubits so the execution stays around a few seconds.

`QD_Sim` treats running sequential `quantum_kernels` and measurement gates differently than the Intel® Quantum Simulator (see **Important Points on Quantum Dot Simulator**). Although some programs could require alteration of their logic, the above program's `quantum_kernel` and simulator usage is compatible with either backend. To change the qubit simulator, instead of the `IqsConfig` object switch to a `DeviceConfig` constructed with the input argument `QD_SIM` as shown in Listing 13 (note the all capitals for the argument string).

Listing 13: Configuring the `FullStateSimulator` to use the Quantum Dot Simulator backend.

```
36 int main() {
37   iqsdk::DeviceConfig qd_config("QD_SIM");
38   iqsdk::FullStateSimulator quantum_8086(qd_config);
```

With this change of backend qubit simulation, the program must now be compiled with options other than the default. The file `intel-quantum-sdk-QDSIM.json` points to a file describing a 6-qubit linear array. Use the `-c` flag to give the file's location to the compiler, and specify placement and scheduling options with `-p` and `-S`.

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

`QD_Sim` produces its own output in addition to the output explicitly designed into the behavior of the program.

```
$ ./qd_GHZ.cpp

1688766838 time dependent evolution start time
sweep progress: calculation point=0 0%
sweep progress: calculation point=18199 10%
sweep progress: calculation point=36398 20%
sweep progress: calculation point=54597 30%
sweep progress: calculation point=72796 40%
sweep progress: calculation point=90994 50%
sweep progress: calculation point=109193 60%
sweep progress: calculation point=127392 70%
sweep progress: calculation point=145591 80%
sweep progress: calculation point=163790 90%
Time evolution took 2.568378 seconds
Fri Jul  7 14:54:01 2023
1688766841 time dependent evolution end time
Sum of probability to measure fully entangled state: 0.999983
Printing probability map of size 2
|00>    : 0.4998                        |11>    : 0.5002

Using 1000 samples, the distribution of states is:
|00> : 0.519
|11> : 0.481
```

The output starts with the backend's simulation progress, followed by the familiar output in the program's implementation. The probabilities reported by the `FullStateSimulator` are slightly different from the exact 0.50 because the Quantum Dot Simulator includes the simulation of the electronics controlling the quantum dots.

The complete code is available as `qd_GHZ.cpp` in the `quantum_examples/` directory of the Intel® Quantum SDK.

24.2 Using release_quantum_state()

Using `release_quantum_state()` to indicate the intention to effectively abandon operating on the qubits can lead to greater reduction of the total operations when combined with `-O1` optimization. Inconsequential operations can be removed, and in some cases this can include a measurement operation. Consider the following sample code. The first three `quantum_kernel` blocks build up the preparation and measurement of a state with three angles as input parameters.

```
#include <clang/Quantum/quintrinsics.h>
#include <quantum.hpp>

qbit q[2];

quantum_kernel void PrepAll() {
  PrepZ(q[0]);
  PrepZ(q[1]);
}

//nothing special about this ansatz
quantum_kernel void Ansatz_Heisenberg (double angle0, double angle1, double angle2) {
  RX(q[0], angle0);
  RY(q[1], angle1);
  S(q[1]);
  CNOT(q[0], q[1]);
  RZ(q[1], angle2);
  CNOT(q[0], q[1]);
  Sdag(q[1]);
}

quantum_kernel double Measure_Heisenberg(){
  cbit c[3];

  CNOT(q[0], q[1]);
  MeasX(q[0], c[0]); // XX term
  MeasZ(q[1], c[1]); // ZZ term
  CZ(q[0], q[1]);
  MeasX(q[0], c[2]); //-YY term
  CZ(q[0], q[1]);
  CNOT(q[0], q[1]);

  release_quantum_state();

  return (1.  -2. * (double) c[0]) + (1.  -2. * (double) c[1]) + (1.  + 2. * (double) c[2]);

}
```

The next `quantum_kernel` block calls the three blocks. This fourth block will contain the `release_quantum_state()` call from above. This `quantum_kernel` is called with some debug values in `main()` after a `FullStateSimulator` is setup.

```
quantum_kernel double VQE_Heisenberg(double angle0, double angle1, double angle2) {
  PrepAll();
  Ansatz_Heisenberg (angle0, angle1, angle2);

  return Measure_Heisenberg();  //note this QK will inherit the release for this function
}

int main(){
  iqsdk::IqsConfig settings(2, "noiseless");
  iqsdk::FullStateSimulator quantum_8086(settings);
  if (iqsdk::QRT_ERROR_SUCCESS != quantum_8086.ready())
    return 1;

  double debug1 = 1.570796;
  double debug2 = debug1 / 2;
  double debug3 = debug2 / 2;
```

(continues on next page)

(continued from previous page)

```
  VQE_Heisenberg(debug1, debug2, debug3);
}
```

If this is compiled with `-O0` flag and no optimization is performed, we expect each of the 3 measurement operations (`MeasX`, `MeasZ`, `MeasX`) to be called. Looking at the resulting `.qs` file confirms this.

```
        .globl        "_Z45VQE_Heisenberg(double, double, double).QBB.3.v.stub" // -- Begin function _
↪Z45VQE_Heisenberg(double, double, double).QBB.3.v.stub
        .type        "_Z45VQE_Heisenberg(double, double, double).QBB.3.v.stub",@function
"_Z45VQE_Heisenberg(double, double, double).QBB.3.v.stub": // @"_Z45VQE_Heisenberg(double, double, double).
↪QBB.3.v.stub"
// %bb.0:                                // %aqcc.quantum
        quprep QUBIT[0] (slice_idx=1)
        quprep QUBIT[1] (slice_idx=0)
        qurotxy QUBIT[0], @shared_variable_array[4], @shared_variable_array[0] (slice_idx=0)
        qurotxy QUBIT[1], @shared_variable_array[5], @shared_variable_array[1] (slice_idx=0)
        qurotz QUBIT[1], 1.570796e+00 (slice_idx=0)
        qurotxy QUBIT[1], 1.570796e+00, 4.712389e+00 (slice_idx=0)
        qucphase QUBIT[0], QUBIT[1], 3.141593e+00 (slice_idx=0)
        qurotxy QUBIT[1], 1.570796e+00, 1.570796e+00 (slice_idx=0)
        qurotz QUBIT[1], @shared_variable_array[6] (slice_idx=0)
        qurotxy QUBIT[1], 1.570796e+00, 4.712389e+00 (slice_idx=0)
        qucphase QUBIT[0], QUBIT[1], 3.141593e+00 (slice_idx=0)
        qurotxy QUBIT[1], 1.570796e+00, 1.570796e+00 (slice_idx=0)
        qurotz QUBIT[1], 4.712389e+00 (slice_idx=0)
        qurotxy QUBIT[1], 1.570796e+00, 4.712389e+00 (slice_idx=0)
        qucphase QUBIT[0], QUBIT[1], 3.141593e+00 (slice_idx=0)
        qurotxy QUBIT[1], 1.570796e+00, 1.570796e+00 (slice_idx=0)
        qurotxy QUBIT[0], 1.570796e+00, 4.712389e+00 (slice_idx=0)
        qumeasz QUBIT[0] @shared_cbit_array[0] (slice_idx=0)
        qurotxy QUBIT[0], 1.570796e+00, 1.570796e+00 (slice_idx=0)
        qumeasz QUBIT[1] @shared_cbit_array[1] (slice_idx=0)
        qucphase QUBIT[0], QUBIT[1], 3.141593e+00 (slice_idx=0)
        qurotxy QUBIT[0], 1.570796e+00, 4.712389e+00 (slice_idx=0)
        qumeasz QUBIT[0] @shared_cbit_array[2] (slice_idx=0)
        qurotxy QUBIT[0], 1.570796e+00, 1.570796e+00 (slice_idx=0)
        qucphase QUBIT[0], QUBIT[1], 3.141593e+00 (slice_idx=0)
        qurotxy QUBIT[1], 1.570796e+00, 4.712389e+00 (slice_idx=0)
        qucphase QUBIT[0], QUBIT[1], 3.141593e+00 (slice_idx=0)
        qurotxy QUBIT[1], 1.570796e+00, 1.570796e+00 (slice_idx=2)
        return
.Lfunc_end3:
        .size        "_Z45VQE_Heisenberg(double, double, double).QBB.3.v.stub", .Lfunc_end3-"_Z45VQE_
↪Heisenberg(double, double, double).QBB.3.v.stub"
                                        // -- End function
        .section        ".note.GNU-stack","",@progbits
```

However, compiling with `-O1` will cause the optimizer to remove operations. Here in addition to combining and removing gates, the optimizer will remove a measurement operation because its outcome can be deduced from the other two measurements' outcomes. The `cbit` used to store the now-missing measurement outcome will have its value correctly set by the Quantum Runtime.

```
        .globl        "_Z45VQE_Heisenberg(double, double, double).QBB.3.v.stub" // -- Begin function _
↪Z45VQE_Heisenberg(double, double, double).QBB.3.v.stub
        .type        "_Z45VQE_Heisenberg(double, double, double).QBB.3.v.stub",@function
"_Z45VQE_Heisenberg(double, double, double).QBB.3.v.stub": // @"_Z45VQE_Heisenberg(double, double, double).
↪QBB.3.v.stub"
```

(continues on next page)

(continued from previous page)

```
// %bb.0:                                // %aqcc.quantum
        quprep QUBIT[1] (slice_idx=1)
        quprep QUBIT[0] (slice_idx=0)
        qurotxy QUBIT[1], @shared_variable_array[4], @shared_variable_array[1] (slice_idx=0)
        qurotxy QUBIT[0], @shared_variable_array[5], @shared_variable_array[0] (slice_idx=0)
        qurotxy QUBIT[0], 1.570796e+00, 4.712389e+00 (slice_idx=0)
        qurotxy QUBIT[1], 1.570796e+00, 0.000000e+00 (slice_idx=0)
        qucphase QUBIT[0], QUBIT[1], 3.141593e+00 (slice_idx=0)
        qurotxy QUBIT[0], 1.570796e+00, 0.000000e+00 (slice_idx=0)
        qumeasz QUBIT[0] @shared_cbit_array[0] (slice_idx=0)
        qurotxy QUBIT[1], 1.570796e+00, 4.712389e+00 (slice_idx=0)
        qumeasz QUBIT[1] @shared_cbit_array[1] (slice_idx=2)
        return
.Lfunc_end3:
        .size        "_Z45VQE_Heisenberg(double, double, double).QBB.3.v.stub", .Lfunc_end3-"_Z45VQE_
↪Heisenberg(double, double, double).QBB.3.v.stub"
                                      // -- End function
        .section        ".note.GNU-stack","",@progbits
```

24.3 Writing Variational Algorithms

24.3.1 Introduction

Variational algorithms are considered to be one of the most promising applications to allow quantum advantage using near-term systems [CABB2021]. The Intel Quantum SDK has many features that are geared for coding variational algorithms with a focus on performance. The Hybrid Quantum Classical Library (HQCL) [HQCL2023] is a collection of tools that will help a user increase productivity when programming variational algorithms. This example combines these tools with `dlib` (a popular C++ library for solving optimization problems [DLIB2023]), to applying the variational algorithm to the generation of thermofield double (TFD) states [PRMA2020] [SPKR2021]. Previous implementations [KWPH2022] included the latter workload with a hard-coded version of the cost function expression, which was pre-calculated. shows the full circuit that is used for the variational algorithm execution. Fig. 1 (reproduced from [KWPH2022]) shows the full circuit that is used for the variational algorithm execution.

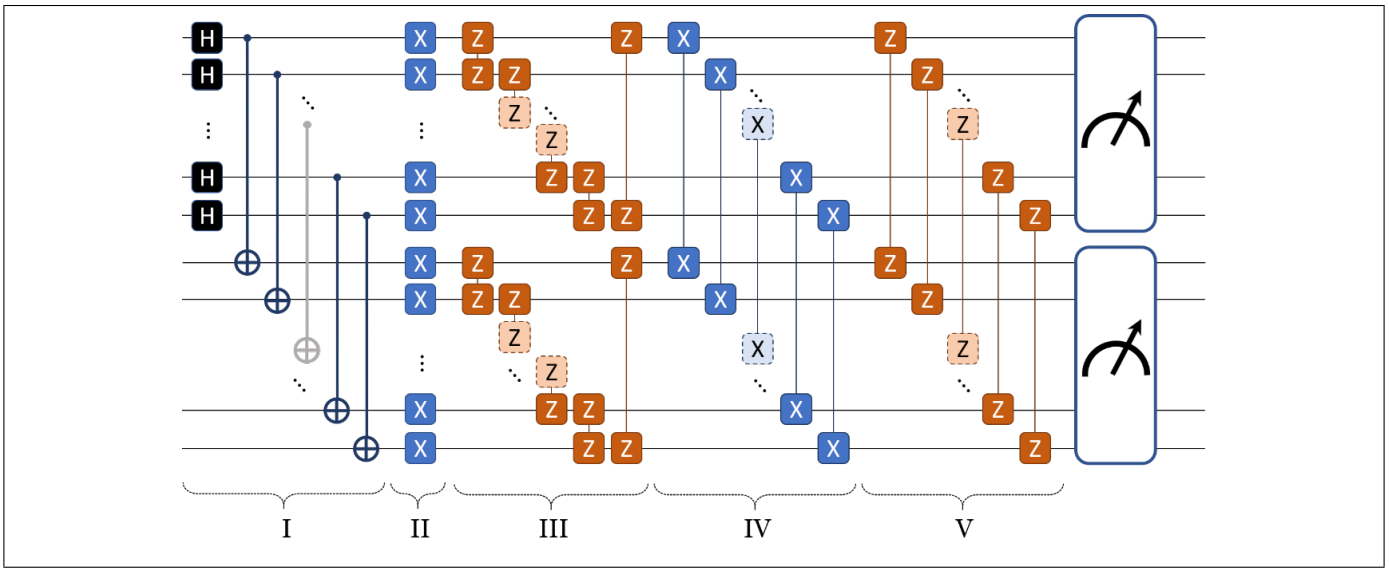


Fig. 1: The quantum circuit for single-step TFD state generation. Stages:(I) preparing infinite temperature TFD state, (II) Intra-system R_X operation, (III) Intra-system ZZ operation, (IV) Inter-system XX operation, (V) Inter-system ZZ operation. A and B represent the two subsystems, each containing N_{sub} qubits. (reproduced from [KWPH2022])

In this implementation, HQCL will symbolically construct the cost function expression, and use qubit-wise commutation (QWC) [VEYI2020] [YEVI2020] to group terms and reduce the number of circuit repetitions required to calculate the cost function. HQCL will also automatically populate the necessary mapping angles at runtime, to facilitate measurements of the system along different axes.

The classical optimization in this workload will be handled using the `dlib` C++ library. The `dlib` library contains powerful functions performing local as well as global optimizations. Here, the `find_min_bobyqa` function for the minimization of the cost function performs quite well for the chosen workload. The choice of the optimization technique can heavily influence the number of iterations required for convergence of certain variational algorithms. The Intel Quantum Simulator [GHBS2020] will be used as the backend in this example.

24.3.2 Code

Preamble

In the preamble of the source file shown in Listing 14 below, the header files for the IQSDK, `dlib`, and HQCL are included first.

Listing 14: The preamble of the source file.

```
// Intel Quantum SDK header files
#include <clang/Quantum/quintrinsics.h>
#include <quantum.hpp>

#include <vector>
#include <cassert>

// Library for optimizations
#include <dlib/optimization.h>
```

(continues on next page)

(continued from previous page)

```
10 #include <dlib/global_optimization.h>
11
12 // Libraries for automating hybrid algorithm
13 #include <armadillo>
14 #include "SymbolicOperator.hpp"
15 #include "SymbolicOperatorUtils.hpp"
16
17 // Define the number of qubits needed for compilation
18 const int N_sub = 2;  // Number of qubits in subsystem (thermal state size)
19 const int N_ss = 2; // Number of subsystems (Not a general parameter to be changed)
20 const int N = N_ss * N_sub; // Total number of qubits (TFD state size)
21
22 qbit QReg[N];
23 cbit CReg[N];
24
25 const int N_var_angles = 4;
26 const int N_map_angles = 2 * N;
27
28 double QVarParams[N_var_angles]; // Array to hold dynamic parameters for quantum algorithm
29 double QMapParams[N_map_angles]; // Array to hold mapping parameters for HQCL
30
31 typedef dlib::matrix<double, N_var_angles, 1> column_vector;
32 namespace hqcl = hybrid::quantum::core;
```

A data structure within `dlib` is defined (using a `typedef`) in line 31 for convenience in passing the set of parameters into the loop during optimization. This algorithm will use four variational parameters and two mapping angles per qubit (a total of eight).

Construction of the Ansatz

Listing 15 contains the operations corresponding to the stages II, III, IV, and V from Fig. 1 (stage I will be discussed later in Listing 16). These are the four core stages that define the operations for the TFD algorithm. A future version of the IQSDK will support `quantum kernel expressions` which can be used to conveniently construct quantum kernels in a modular way [PAMS2023] [SCHM2023].

Listing 15: The core set of operations to implement the TFD algorithm.

```
34 quantum_kernel void TFD_terms () {
35   int index_intraX = 0, index_intraZ = 0, index_interX = 0, index_interZ = 0;
36
37   // Single qubit variational terms
38   for (index_intraX = 0; index_intraX < N; index_intraX++)
39     RX(QReg[index_intraX], QVarParams[0]);
40
41   // Two-qubit intra-system variational terms (adjacent)
42   for (int grand_intraZ = 0; grand_intraZ < N_sub - 1; grand_intraZ++) {
43     for (index_intraZ = 0; index_intraZ < N_ss; index_intraZ++)
44       CNOT(QReg[grand_intraZ + N_sub * index_intraZ + 1], QReg[grand_intraZ + N_sub * index_intraZ]);
45     for (index_intraZ = 0; index_intraZ < N_ss; index_intraZ++)
46       RZ(QReg[grand_intraZ + N_sub * index_intraZ], QVarParams[1]);
47     for (index_intraZ = 0; index_intraZ < N_ss; index_intraZ++)
48       CNOT(QReg[grand_intraZ + N_sub * index_intraZ + 1], QReg[grand_intraZ + N_sub * index_intraZ]);
49   }
50
51   if (N_sub > 2) {
```

(continues on next page)

(continued from previous page)

```
    // Two-qubit intra-system variational terms (boundary term)
    for (index_intraZ = 0; index_intraZ < N_ss; index_intraZ++)
      CNOT(QReg[N_sub * index_intraZ], QReg[N_sub * index_intraZ + (N_sub - 1)]);
    for (index_intraZ = 0; index_intraZ < N_ss; index_intraZ++)
      RZ(QReg[N_sub * index_intraZ + (N_sub - 1)], QVarParams[1]);
    for (index_intraZ = 0; index_intraZ < N_ss; index_intraZ++)
      CNOT(QReg[N_sub * index_intraZ], QReg[N_sub * index_intraZ + (N_sub - 1)]);
  }

  // two-qubit inter-system XX variational terms
  for (index_interX = 0; index_interX < N_sub; index_interX++) {
    RY(QReg[index_interX + N_sub], -M_PI_2);
    RY(QReg[index_interX], -M_PI_2);
  }
  for (index_interX = 0; index_interX < N_sub; index_interX++)
    CNOT(QReg[index_interX + N_sub], QReg[index_interX]);
  for (index_interX = 0; index_interX < N_sub; index_interX++)
    RZ(QReg[index_interX], QVarParams[2]);
  for (index_interX = 0; index_interX < N_sub; index_interX++)
    CNOT(QReg[index_interX + N_sub], QReg[index_interX]);
  for (index_interX = 0; index_interX < N_sub; index_interX++) {
    RY(QReg[index_interX + N_sub], M_PI_2);
    RY(QReg[index_interX], M_PI_2);
  }

  // two-qubit inter-system ZZ variational terms
  for (index_interZ = 0; index_interZ < N_sub; index_interZ++)
    CNOT(QReg[index_interZ], QReg[index_interZ + N_sub]);
  for (index_interZ = 0; index_interZ < N_sub; index_interZ++)
    RZ(QReg[index_interZ + N_sub], QVarParams[3]);
  for (index_interZ = 0; index_interZ < N_sub; index_interZ++)
    CNOT(QReg[index_interZ], QReg[index_interZ + N_sub]);
```

There are three supporting quantum kernels that are used (see Listing 16), in addition to the core set of operations. `PrepZAll()` is used to prepare all the qubits in the ground state at the beginning of every iteration. `BellPrep()` is used to prepare Bell pairs between corresponding qubits between the two subsystems, effectively resulting in the infinite temperature TFD state. The `DynamicMapping()` quantum kernel is used to hold the mapping operations that HQCL will implement for basis changes during runtime.

Listing 16: The supporting quantum kernels to implement the TFD algorithm.

```
quantum_kernel void PrepZAll () {
  // Initialization of the qubits
  for (int Index = 0; Index < N; Index++)
    PrepZ(QReg[Index]);
}

quantum_kernel void BellPrep () {
  // preparation of Bell pairs (T -> Infinity)
  for (int Index = 0; Index < N_sub; Index++)
    H(QReg[Index]);
  for (int Index = 0; Index < N_sub; Index++)
    CNOT(QReg[Index], QReg[Index + N_sub]);
}

```

(continues on next page)

(continued from previous page)

```
quantum_kernel void DynamicMapping () {
  // Not part of the ansatz
  // Helper rotations to map X to Z or Y to Z
  for (int qubit_index = 0; qubit_index < N; qubit_index++) {
    int map_index = 2 * qubit_index;
    RY(QReg[qubit_index], QMapParams[map_index]);
    RX(QReg[qubit_index], QMapParams[map_index + 1]);
  }
}
```

The supporting quantum kernels (Listing 16) and the core TFD quantum kernel (Listing 15) are combined to form the full quantum circuit in Listing 17.

Listing 17: The full TFD quantum kernel to run during optimization loop.

```
quantum_kernel void TFD_full() {
  PrepZAll();
  BellPrep();
  TFD_terms();
  DynamicMapping();
}
```

Functions for Constructing the Cost Expression and the Cost Calculation

Listing 18: Construction of the full symbolic operator for the cost function expression.

```
hqcl::SymbolicOperator constructFullSymbOp(double inv_temp) {
  hqcl::SymbolicOperator symb_op;
  hqcl::pstring sym_term;

  // Single qubit variational terms
  for (int index_intraX = 0; index_intraX < N; index_intraX++) {
      sym_term = {std::make_pair(index_intraX, 'X')};
      symb_op.addTerm(sym_term, 1.00);
  }

  // Two-qubit intra-system variational terms (adjacent)
  for (int grand_intraZ = 0; grand_intraZ < N_sub - 1; grand_intraZ++) {
      for (int index_intraZ = 0; index_intraZ < N_ss; index_intraZ++) {
          int qIndex0 = grand_intraZ + N_sub * index_intraZ;
          int qIndex1 = grand_intraZ + N_sub * index_intraZ + 1;
          sym_term = {std::make_pair(qIndex0, 'Z'), std::make_pair(qIndex1, 'Z')};
          symb_op.addTerm(sym_term, 1.00);
      }
  }

  // Two-qubit intra-system variational terms (boundary term)
  if (N_sub > 2) {
      for (int index_intraZ = 0; index_intraZ < N_ss; index_intraZ++) {
          int qIndex0 = N_sub * index_intraZ;
          int qIndex1 = N_sub * index_intraZ + (N_sub - 1);
          sym_term = {std::make_pair(qIndex0, 'Z'), std::make_pair(qIndex1, 'Z')};
          symb_op.addTerm(sym_term, 1.00);
```

(continues on next page)

(continued from previous page)

```
      }
  }

  // two-qubit inter-system XX variational terms
  for (int index_interX = 0; index_interX < N_sub; index_interX++) {
      int qIndex0 = index_interX;
      int qIndex1 = index_interX + N_sub;
      sym_term = {std::make_pair(qIndex0, 'X'), std::make_pair(qIndex1, 'X')};
      symb_op.addTerm(sym_term, -pow(inv_temp, -1.00));
  }

  // two-qubit inter-system XX variational terms
  for (int index_interZ = 0; index_interZ < N_sub; index_interZ++) {
      int qIndex0 = index_interZ;
      int qIndex1 = index_interZ + N_sub;
      sym_term = {std::make_pair(qIndex0, 'Z'), std::make_pair(qIndex1, 'Z')};
      symb_op.addTerm(sym_term, -pow(inv_temp, -1.00));
  }

  return symb_op;
}
```

Programmatic construction of the cost expression is necessary for HQCL to form the QWC groups, to generate the necessary circuits to run per optimization iteration, and to correctly evaluate the cost at each iteration. In Listing 18, we generate symbolic terms (`sym_term`) for every expression present, and add it to the full symbolic operator `symb_op`. The full cost expression to be coded is given by

$$C(\beta) = \sum_{i=1}^{N_{\text{sub}}} X_i^A + \sum_{i=1}^{N_{\text{sub}}} X_i^B + \sum_{i=1}^{N_{\text{sub}}} Z_i^A Z_{i+1}^A + \sum_{i=1}^{N_{\text{sub}}} Z_i^B Z_{i+1}^B - \beta^{-1} \left(\sum_{i=1}^{N_{\text{sub}}} X_i^A X_i^B + \sum_{i=1}^{N_{\text{sub}}} Z_i^A Z_i^B \right)$$

where the subscript represents the qubit index and the superscript represents the subsystem the qubit belongs to (A or B) [KWPH2022].

Listing 19: A function to calculate the cost at each iteration during optimization.

```
double runQuantumKernel(iqsdk::FullStateSimulator &sim_device, const column_vector& params,
                  hqcl::SymbolicOperator &symb_op, hqcl::QWCMap &qwc_groups) {
  double total_cost = 0.0;

  for (auto &qwc_group : qwc_groups) {
    std::vector<double> variable_params;
    variable_params.reserve(N * 2);

    hqcl::SymbolicOperatorUtils::applyBasisChange(qwc_group.second, variable_params, N);

    std::vector<std::reference_wrapper<qbit>> qids;
    for (int qubit = 0; qubit < N; ++qubit)
      qids.push_back(std::ref(QReg[qubit]));

    // Setting all the mapping angles to the default of 0.
    for (int map_index = 0; map_index < N_map_angles; map_index++)
      QMapParams[map_index] = 0;

    for (auto indx = 0; indx < variable_params.size(); ++indx)
      QMapParams[indx] = variable_params[indx];
```

(continues on next page)

(continued from previous page)

```
    // Performing the experiment, Storing the data in ProbReg
    TFD_full();
    std::vector<double> ProbReg = sim_device.getProbabilities(qids);

    double current_pstr_val = hqcl::SymbolicOperatorUtils::getExpectValSetOfPaulis(
        symb_op, qwc_group.second, ProbReg, N);
    total_cost += current_pstr_val;
  }

  return total_cost;
}
```

The function `runQuantumKernel` encompasses all the functionality that is required to calculate the cost, when running a single optimization iteration with `dlib`. It will take in the already formulated set of QWC groups, and run the ansatz with the same set of variational parameters but with different basis mapping parameters (each time mapping from the different bases, as demanded by the cost expression). The full cost is then returned for consideration by the classical optimization loop within `dlib`.

Listing 20: The `main` function.

```
int main() {
    // Setup quantum device
    iqsdk::IqsConfig sim_config(N, "noiseless", false);
    iqsdk::FullStateSimulator sim_device(sim_config);
    assert(sim_device.ready() == iqsdk::QRT_ERROR_SUCCESS);

    // initial starting point. Defining it here means I will reuse the best result from
    // from previous temperature when starting the next temperature run
    column_vector starting_point = {0, 0, 0, 0};

    // calculating the actual inverse temperature that is used during calculations
    double inv_temp = 1.0;

    // Fully formulated Cost Function Expression
    hqcl::SymbolicOperator cost_expr = constructFullSymbOp(inv_temp);

    // Qubitwise Commutation (QWC) Groups Formation
    hqcl::QWCMap qwc_groups = hqcl::SymbolicOperatorUtils::getQubitwiseCommutationGroups(cost_expr, N);

    // Constructing a function to be used for a single optimization iteration
    // This function is directly called by the dlib optimization routine
    auto ansatz_run_lambda = [&](const column_vector& var_angs) {

      // Setting all the variational angles to input values.
      for (int q_index = 0; q_index < N_map_angles; q_index++)
          QVarParams[q_index] = var_angs(q_index);

      // runs the kernel to compute the total cost
      double total_cost = runQuantumKernel(sim_device, var_angs, cost_expr, qwc_groups);

      return total_cost;
    };

    // running the full optimization for a given temperature
    auto result = dlib::find_min_bobyqa(
```

(continues on next page)

(continued from previous page)

```
                        ansatz_run_lambda, starting_point,
                        2 * N_var_angles + 1, // number of interpolation points
                        dlib::uniform_matrix<double>(N_var_angles, 1, -7.0), // lower bound constraint
                        dlib::uniform_matrix<double>(N_var_angles, 1, 7.0), // upper bound constraint
                        1.5, // initial trust region radius
                        1e-5, // stopping trust region radius
                        10000 // max number of objective function evaluations
    );

    return 0;
}
```

The main function shown in Listing 20 will initialize the IQSDK backend, construct the cost expression, and kick off the optimization. The lambda function `ansatz_run_lambda` is used since `dlib` requires the function used during optimization to take a column vector as an input and to return a double. This function essentially wraps the `runQuantumKernel` function which was defined previously.

24.3.3 Results

The execution of the above program can be tracked with a log of the angles and the cost function at each iteration (by using appropriate functions). The summarized results are given in Fig. 2 and Fig. 3 and it is found that 95 steps are required for convergence to the requested tolerance level.

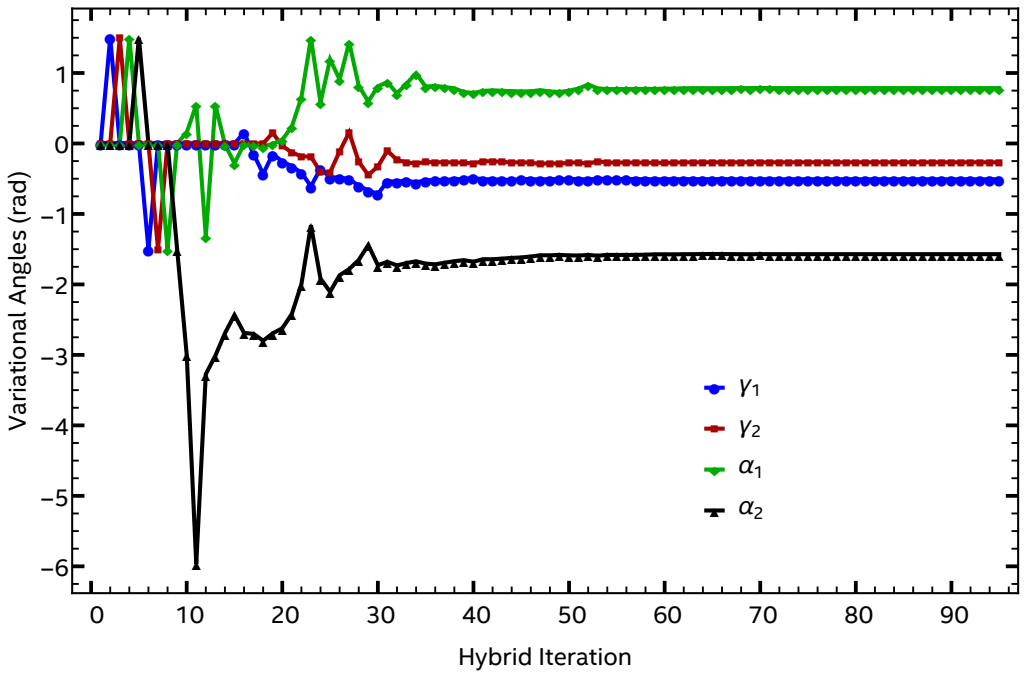


Fig. 2: Convergence behavior for the four variational angles (see [PRMA2020] [SPKR2021] for details on the notation).

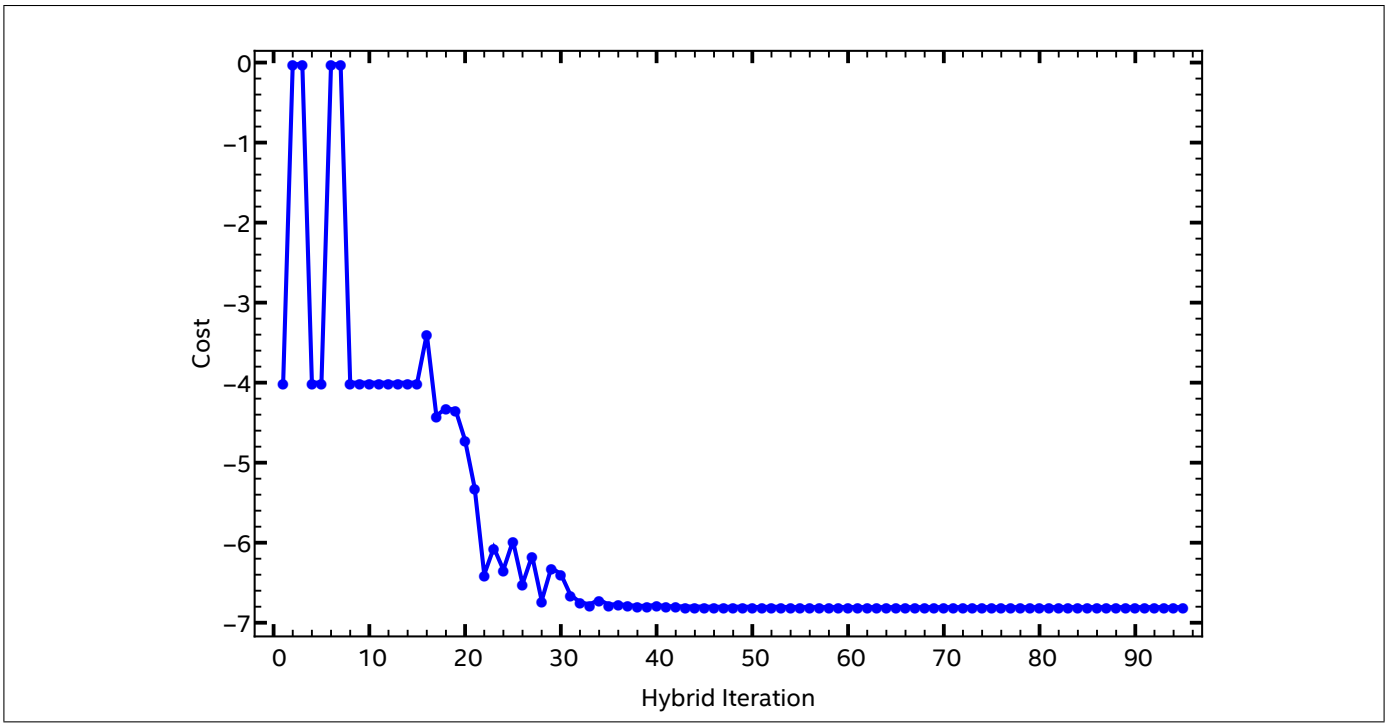


Fig. 3: Convergence of the evaluated cost during the variational algorithm execution.

Bibliography

[BELL964] 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

[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

[YAMA2008] Yanofsky, N., & Mannucci, M. (2008). Quantum Computing for Computer Scientists. Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9780511813887

[HIDA2019] Hidary, J.D. (2019). Quantum Computing: An Applied Approach. Springer, Cham. https://doi.org/10.1007/978-3-030-23922-0

[STRO2022] Stroustrup, B. (2022). A Tour of C++. Addison-Wesley. https://www.stroustrup.com/tour3.html

[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. Rehãcek, eds., ˇ Quantum State Estimation (Springer Berlin Heidelberg, 2004). https://doi.org/10.1007/b98673

[CABB2021] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio, and P. J. Coles, Nature Reviews Physics 3, 625 (2021). https://doi.org/10.1038/s42254-021-00348-9

[HQCL2023] Hybrid Quantum-Classical Library, https://github.com/IntelLabs/Hybrid-Quantum-Classical-Library, Accessed : 2023-03-26.

[DLIB2023] dlib C++ library, http://dlib.net/, Accessed : 2023-03-26.

[PRMA2020] S. P. Premaratne and A. Y. Matsuura, in 2020 IEEE International Conference on Quantum Computing and Engineering (QCE) (IEEE, 2020). https://doi.org/10.1109/QCE49297.2020.00042

[SPKR2021] R. Sagastizabal, S. P. Premaratne, B. A. Klaver, M. A. Rol, V. Neĝırneac, M. S. Moreira, X. Zou, S. Johri, N. Muthusubramanian, M. Beekman, C. Zachariadis, V. P. Ostroukh, N. Haider, A. Bruno, A. Y. Matsuura, and L. DiCarlo, npj Quantum Information 7, 10.1038/s41534-021-00468-1 (2021). https://doi.org/10.1038/s41534-021-00468-1

[VEYI2020] V. Verteletskyi, T.-C. Yen, and A. F. Izmaylov, The Journal of Chemical Physics 152, 124114 (2020). https://doi.org/10.1063/1.5141458

[YEVI2020] T.-C. Yen, V. Verteletskyi, and A. F. Izmaylov, Journal of Chemical Theory and Computation 16, 2400 (2020). https://doi.org/10.1021/acs.jctc.0c00008

[GHBS2020] Gian Giacomo Guerreschi, Justin Hogaboam, Fabio Baruffa and Nicolas P D Sawaya, 2020 Quantum Sci. Technol. 5 034007. (2020). https://dx.doi.org/10.1088/2058-9565/ab8505

[PAMS2023] J. Paykin, A. Y. Matsuura, and A. T. Schmitz, in 2023 APS March Meeting, RR08.00007 (2023). https://meetings.aps.org/Meeting/MAR23/Session/RR08.7

[SCHM2023] A. T. Schmitz, in 2023 APS March Meeting, RR08.00008 (2023). https://meetings.aps.org/Meeting/MAR23/Session/RR08.8