Writing Variational Algorithms with the Intel Quantum SDK

Shavindra P. Premaratne[1]

[1]*Intel Labs, Intel Corporation, 2111 NE 25th Ave, Hillsboro, OR 97124*[*]

(Dated: April 4, 2023)

The Intel Quantum SDK (IQSDK) has many features that are geared for coding variational algorithms. The Hybrid Quantum Classical Library (HQCL) [1] is a collection of tools that will help a user increase productivity when programming variational algorithms. Here, we will explore the way to code a reasonably advanced algorithm for the IQSDK using HQCL and dlib (a popular C++ library for solving optimization problems) [2].

I. INTRODUCTION

Variational algorithms are considered to be one of the most promising applications to allow quantum advantage using near-term systems [3]. The IQSDK has many features available that allows users to code and execute variational algorithms with a focus on performance. In this manuscript we will look at coding a variational algorithm for generation of thermofield double (TFD) states [4, 5]. Our previous work [6] included the latter workload with a hard-coded version of the cost function expression, which was calculated elsewhere. Fig. 1 shows the full circuit that is used for the variational algorithm execution.

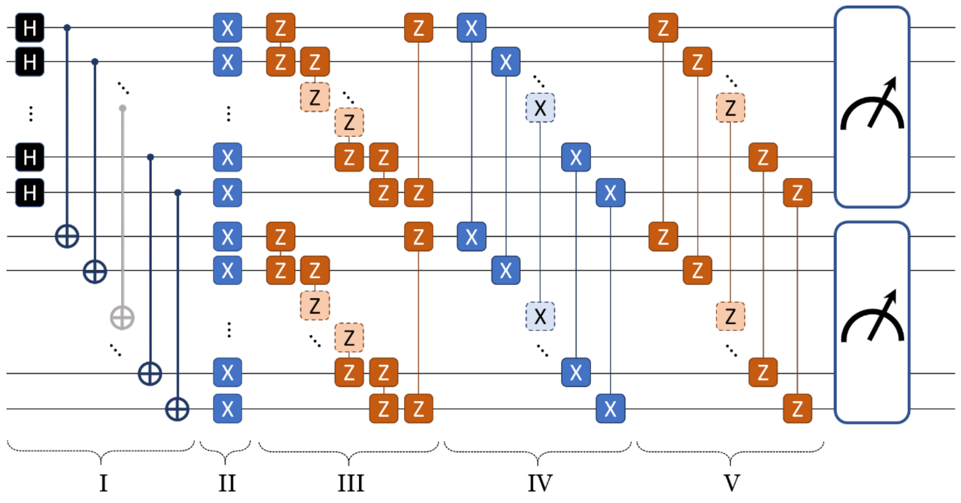

FIG. 1. The quantum circuit for single-step TFD state generation. Stages: (I) preparing infinite temperature TFD state, (II) Intra-system R_X operation, (III) Intra-system ZZ operation, (IV) Inter-system XX operation, (V) Inter-system ZZ operation. A and B represent the two subsystems, each containing N_{sub} qubits (reproduced from Khalate *et al.* [6]).

Here we will use HQCL to symbolically construct the cost function expression. We will also employ the available feature of qubit-wise commutation (QWC) [7, 8] to let HQCL 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, we will use the `find_min_bobyqa` function for the minimization of the cost function, since it 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 (IQS) [9] will be used as the backend in this example.

* shavindra.premaratne@intel.com

II. CODE

A. Preamble

In the preamble of the source file (program code 1), header files for the IQSDK, dlib, and HQCL are included. A data structure within dlib is defined (using a `typedef`) 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).

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

#include <vector>
#include <cassert>

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

// Libraries for automating hybrid algorithm
#include <armadillo>
#include "SymbolicOperator.hpp"
#include "SymbolicOperatorUtils.hpp"

// Define the number of qubits needed for compilation
const int N_sub = 2;  // Number of qubits in subsystem (thermal state size)
const int N_ss = 2; // Number of subsystems (Not a general parameter to be changed)
const int N = N_ss * N_sub; // Total number of qubits (TFD state size)

qbit QReg[N];
cbit CReg[N];

const int N_var_angles = 4;
const int N_map_angles = 2 * N;

double QVarParams[N_var_angles]; // Array to hold dynamic parameters for quantum algorithm
double QMapParams[N_map_angles]; // Array to hold mapping parameters for HQCL

typedef dlib::matrix<double, N_var_angles, 1> column_vector;
namespace hqcl = hybrid::quantum::core;
```

Program Code 1. The preamble of the source file.

B. Construction of the Ansatz

Program Code 2 contains the operations corresponding to the stages II, III, IV, and V from Fig. 1 (stage I will be discussed later in program code 3). 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 [10, 11].

```
quantum_kernel void TFD_terms () {
  int index_intraX = 0, index_intraZ = 0, index_interX = 0, index_interZ = 0;

  // Single qubit variational terms
  for (index_intraX = 0; index_intraX < N; index_intraX++)
    RX(QReg[index_intraX], QVarParams[0]);

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

  if (N_sub > 2) {
    // 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]);
}
```

Program Code 2. The core set of operations to implement the TFD algorithm.

There are three supporting quantum kernels that are used (see program code 3), 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.

```
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]);
}

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]);
  }
}
```

Program Code 3. The supporting quantum kernels to implement the TFD algorithm.

We combine the supporting quantum kernels (program code 3) and the core TFD quantum kernel (program code 2) to form the full quantum circuit in program code 4.

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

Program Code 4. The full TFD quantum kernel to run during optimization loop.

C. Functions for Constructing the Cost Expression and the Cost Calculation

```cpp
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);
      }
  }

  // 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;
}
```

Program Code 5. Construction of the full symbolic operator for the cost function expression.

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 program code 5, 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

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

where the single subscript operators are acting within the corresponding subsystem and the double subscript operators represent the operators acting between the subsystems A and B [6].

```
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];

    // 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;
}
```

Program Code 6. A function to calculate the cost at each iteration during optimization.

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.

D. Main function

```cpp
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(
                        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;
}
```

Program Code 7. The `main` function.

The main function 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 we defined previously.

III. 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 Figs. 2 and 3 and it is found that 95 steps are required for convergence to the requested tolerance level.

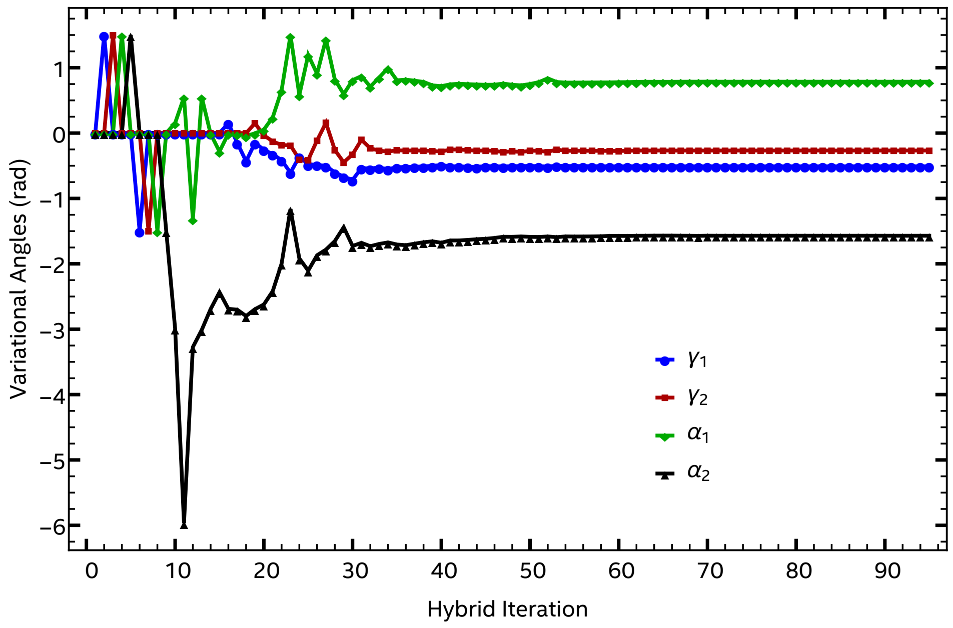

FIG. 2. Convergence behavior for the four variational angles (see [4, 5] for details on the notation).

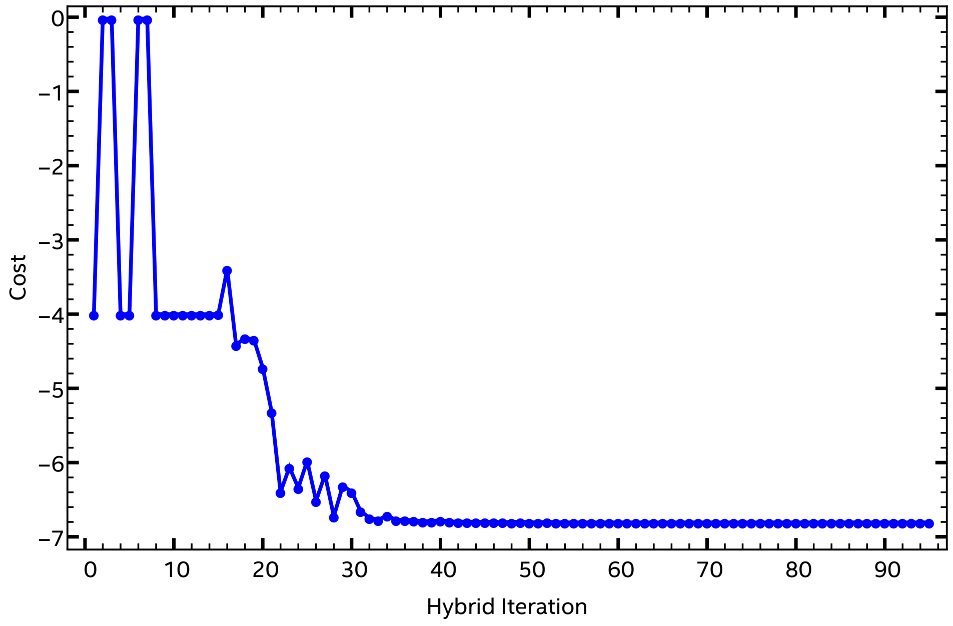

FIG. 3. Convergence of the evaluated cost during the variational algorithm execution.

ACKNOWLEDGMENTS

The author thanks Nicolas Sawaya, Kevin Rasch, and Atul Kulkarni for valuable comments on the manuscript.

[1] Hybrid Quantum-Classical Library, `https://github.com/IntelLabs/Hybrid-Quantum-Classical-Library`, Accessed : 2023-03-26.
[2] dlib C++ library, `http://dlib.net/`, Accessed : 2023-03-26.
[3] 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).
[4] S. P. Premaratne and A. Y. Matsuura, in *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)* (IEEE, 2020).
[5] R. Sagastizabal, S. P. Premaratne, B. A. Klaver, M. A. Rol, V. Negîrneac, 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).
[6] P. Khalate, X.-C. Wu, S. Premaratne, J. Hogaboam, A. Holmes, A. Schmitz, G. G. Guerreschi, X. Zou, and A. Y. Matsuura, An LLVM-based C++ Compiler Toolchain for Variational Hybrid Quantum-Classical Algorithms and Quantum Accelerators (2022), arXiv:2202.11142.
[7] V. Verteletskyi, T.-C. Yen, and A. F. Izmaylov, The Journal of Chemical Physics **152**, 124114 (2020).
[8] T.-C. Yen, V. Verteletskyi, and A. F. Izmaylov, Journal of Chemical Theory and Computation **16**, 2400 (2020).
[9] Intel Quantum Simulator, `https://github.com/intel/intel-qs`, Accessed : 2023-03-26.
[10] J. Paykin, A. Y. Matsuura, and A. T. Schmitz, in *2023 APS March Meeting*, RR08.00007 (2023).
[11] A. T. Schmitz, in *2023 APS March Meeting*, RR08.00008 (2023).