Optimization of Quantum Computing Simulation with Gate Fusion

Hiroshi Horii[1,a)] Jun Doi[1,b)]

Abstract: Memory to simulate quantum computing is exponentially increased based on qubits of a circuit and the entire memory is updated for simulation of each gate in a circuit to be simulated. For example, 32 GB memory is necessary to represent all the probability amplitudes with double-precision and all of them is updated for each gate. Aggregating multiple gates into a single unitary-matrix gate reduces load and store of memory. However, if an aggregated gate updates many qubits, memory access and calculation of intermediate state of matrix multiplication can become the bottleneck. We propose a method to efficiently aggregate gates with pattern-matchings, greedy algorithms, and a graph algorithm. Our gate fusion reduced gates of various quantum circuits of Qiskit and improved performance of their simulation.

Keywords: Quantum Computing Simulation, Optimization, Transpilation

1. Introduction

Quantum computing simulation is an important tool for the study and development of quantum algorithms. Real quantum computers are now available as cloud service, but their noise sometimes prevent users from validating that running quantum circuits are written correctly. Therefore, popular toolkits for quantum software, such as Qiskit[1], Q#[3], Qulacs[11], and Cirq[2], [6], provide their simulators that calculate quantum states of quantum computers. The most naive way to represent quantum state in simulator is storing all the probability amplitudes as a complex vector and iterating gates in a quantum circuit while updating the vector. Probability amplitudes are increased based on a number of qubits, thus, simulators need huge memory and computation resources to simulate quantum computers of large qubit. For example, a simulator updates 64GB memory for each gate to simulate 30-qubit quantum computer with double precision. In this paper, we study a method to optimize such updates of probability amplitudes in simulation of quantum computers.

Gate fusion [2], [7], [10] is an optimization technique to reduce gates in a circuit by replacing multiple gates with a single gate while guaranteeing generation of the same quantum state. To simulate a gate that updates m qubits, a simulator iterates load of 2^m probability amplitudes, update them with multiplying a $2^m \times 2^m$ matrix, and store the updated amplitudes until all of the amplitudes are updated. By reducing a number of gates in a circuit, gate fusion can reduce such load, store and multiplication of complex values.

Gate fusion does not always improve simulator performance. Consider aggregation into a single m-qubit gate and its application to a vector of 2^n complex numbers. First, aggregations of gates also requires computation and memory resources because the aggregated gate is represented as a $2^m \times 2^m$ complex matrix. If m is large, generation of $2^m \times 2^m$ matrix requires more resources than simulation of original gates. Second, a $2^m \times 2^m$ matrix exceeds capacity of registers in the current CPU architecture if m is more than two, then, its multiplication with a complex vector leads additional overheads. Increased load and store instructions for spilled memory become more than reduced instructions and then total simulation time can be worse. Moreover, based on m, multiplication and summation of complex numbers are exponentially increased in multiplication of a $2^m \times 2^m$ matrix and 2^n vector. Therefore, gate fusion is effective if aggregated gates repeatedly updates the same qubits and a generated gate updates small qubits.

In this paper, we propose a method to aggregate gates to reduce total simulation time. Though existing simulators generate gates that update fixed-number qubits in their gate fusion, our gate fusion generates gates that update various numbers of qubits. In addition, we specialize gate fusion for diagonal matrices. Applying a diagonal matrix needs only 2^n multiplication of complex numbers. We use pattern matching to find a sequence of gates in a quantum circuit, and then generate a diagonal-matrix gate to replace them. After aggregating neighbor gates if they updates same qubits, we finally find the best pattern of gate fusion for a quantum circuit while estimating total simulation time.

2. Quantum Computing Simulation

We focus on simulating universal quantum computers based on quantum circuits consisting of one-qubit rotation gates and two-qubit CNOT gates. These gates are universal; i.e., any quantum circuit (for realizing some quantum algorithm) can be constructed

[1] IBM Quantum, IBM Research Tokyo, 19-21, Nihonbashi Hakozaki-cho Chuo-ku, Tokyo 103-8510 Japan
[a)] horii@jp.ibm.com
[b)] doichanj@jp.ibm.com

from the CNOT and one-qubit rotation gates. This section briefly overviews quantum computing and its simulation, and then summarizes optimization implemented in existing simulators.

2.1 Overview

A qubit has two basis states $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$, and the quantum state of an n-qubit register is a linear superposition of 2^n basis states, each of the form of the tensor product of n qubits. For example, the basis state of digit 2 (or 10 in binary) can be represented as $|2\rangle = |1\rangle \otimes |0\rangle = |10\rangle = (0, 0, 1, 0)^T$, where T denotes the transpose of a vector. Thus, the quantum state of the n-qubit register can be written as $|\psi\rangle = \sum_{i=0}^{2^n-1} a_i|i\rangle$. Note that each state $|i\rangle$ has its own probability amplitude a_i in a complex number. Simulating quantum circuits requires that these 2^n complex numbers (statevector) be stored to enable tracking of the evolution of the quantum state of an n-qubit register.

Quantum gates transform the quantum state of the n-qubit register by rotating its complex vector. The OpenQASM specification [4] provides the gate set: U an arbitrary single-qubit (rotation) gate and CX a two-qubit gate. The U gates are rotations of the 1-qubit state and are mathematically defined as

$$U(\theta, \psi, \lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ -e^{i\psi}\sin(\theta/2) & e^{\psi+\lambda}\cos(\theta/2) \end{pmatrix}. \quad (1)$$

This matrix m transforms the state of a q-th qubit in a statevector qv with Listing 1. Two elements at p and p + (1 ≪ q) in qv are updated together, and each element is updated once.

Listing 1: Pseudocode to apply single qubit with m

```
1 void apply_matrix(int q, complex m[2][2]) {
2 #pragma omp parallel for collapse(2)
3  for (int i = 0; i < (1<<n); i += (1<<(q+1)) {
4    for (int j = 0; j < (1<<q); j++) {
5      auto p = i|j;
6      auto q0 = qv[p];
7      auto q1 = qv[p|(1<<q)];
8      qv[p] = m[0][0]*q0 + m[0][1]*q1;
9      qv[p|(1<<q)] = m[1][0]*q0 + m[1][1]*q1;
10 } } }
```

The CX gates are applied to two qubits: a control and a target qubit. If the control qubit is in state $|1\rangle$, the CX gate flips the target qubit. If the control qubit is in state $|0\rangle$, the CX gate does not change the target qubit. If we have two qubits and take the higher bit as the control qubit and the other as the target qubit, the CX gate is mathematically defined as

$$\mathrm{CX} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \quad (2)$$

This matrix m transforms the state of qubits q[2] in a statevector qv with Listing 2 assuming q[1] > q[0]. Four elements at p, p + (1 ≪ q[0]), p + (1 ≪ q[1]) and p + (1 ≪ q[0]) + (1 ≪ q[1]) in qv are updated together, and each element is updated once.

Listing 2: Pseudocode to apply two qubits with m

```
1 void apply_matrix(int q[2], complex m[4][4]) {
2  assert(q[0] < q[1]);
3  auto m0 = 1<<q[0]; auto m1 = q << q[1];
4 #pragma omp parallel for collapse(3)
5  for (int i=0; i<(1<<n); i+=(1<<(q[1]+1)) {
6    for (int j=0;j<(1<<q[1]);j+=(1<<(q[0]+1)){
7      for (int k=0; k<(1<<q[0]); k++) {
8        int p = i|j|k;
9        auto q0 = qv[p]; auto q1 = qv[p|m0];
10       auto q2 = qv[p|m1]; auto q3 = qv[p|m0|m1];
11       qv[p] = q0*m[0][0] + q1*m[0][1]
12                     + q2*m[0][2] + q3*m[0][3];
13       qv[p|m0] = q0*m[1][0] + q1*m[1][1]
14                     + q2*m[1][2] + q3*m[1][3];
15       qv[p|m1] = q0*m[2][0] + q1*m[2][1]
16                     + q2*m[2][2] + q3*m[2][3];
17       qv[p|m0|m1] = q0*m[3][0] + q1*m[3][1]
18                     + q2*m[3][2] + q3*m[3][3];
19  } } } }
```

Note that Lines 8-9 in Listing 1 and Lines 11-18 in Listing 2 perform multiplication of a matrix and a vector of complex numbers.

Main overheads are three parts in these Listings: Loading complex numbers from a state vector (Lines 6-7 and 9-10 in Listing 1 and 2), multiplying the complex numbers, and storing updated complex numbers to the state vector (Lines 8-9 and 11-18 in Listing 1 and 2). Because a complex number consists a real and imaginary numbers, instructions to load, multiply, add, and store floating-point numbers are frequently called.

2.2 Parallelization

Parallelization is typical optimization in quantum computing simulation. As shown in Listing 1 and 2, to simulate a q qubit gate, 2^q probability amplitudes are updated together and each probability amplitude is updated once. By making updates of 2^q probability amplitudes a unit of work, multiple levels of parallelization are effective for simulation of quantum computing.

Modern CPUs, such as Intel and POWER, support SIMD instructions that enable instruction-level parallelization. Especially, instructions of Intel's AVX2 and POWER's VSX load 256-bit data into a vector register (precisely two 128bit vector registers are combined to one 256bit) that can store two or four floating-point numbers with single and double precisions respectively. Instructions to multiply and add for such vector registers are processed with the mostly same clocks with for general-purpose registers of 64-bit. Though variations of instructions for vector registers are less than for general-purpose registers, AVX2 and VSX cover instructions to multipy complex numbers to improve performance of HPC workloads. In addition, memory controllers in Intel load (store) vector registers from (to) memory more efficiently than general-purpose registers. Consequently, SIMD instructions can improve its memory bandwidth.

Multi-threading is effective optimization in quantum computing simulation. As shown in Line 2 and 4 in Listing 1 and 2, OpenMP is frequently used in existing simulators to enable multithread parallelization. These directives parallelize their loop bod-

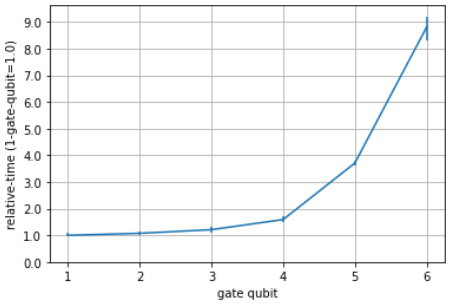

Fig. 1: Relative elapsed time to update 20-qubit state vector with a matrix based on simulation time

ies by using the same number of threads with hardware-threads in the system or configured variable OMP_NUM_THREADS.

GPU is also effective to improve performance if memory of GPU is enough to store all the complex numbers of a state vector. Though techniques to transfer probability amplitudes are necessary for performance [5], distributed memory in multi-GPUs and multi-nodes are also effective to enlarge qubits to be simulated with performance.

2.3 Gate Fusion

Gate fusion is a technique to aggregate multiple gates into a single gate while guaranteeing that the final quantum state is equivalent before and after gate fusion. For example, to aggregate two gates that update different a-th and b-th qubits with matrices $M^a = \begin{pmatrix} m^a_{00} & m^a_{01} \\ m^a_{10} & m^a_{11} \end{pmatrix}$ and $M^b = \begin{pmatrix} m^b_{00} & m^b_{01} \\ m^b_{10} & m^b_{11} \end{pmatrix}$, a generated gate with gate fusion uses a matrix M^{fused} to update the qubits, which is calculated with the following equation:

$$M^{fused} = \begin{pmatrix} m^a_{00} & m^a_{01} & 0 & 0 \\ m^a_{10} & m^a_{11} & 0 & 0 \\ 0 & 0 & m^a_{00} & m^a_{01} \\ 0 & 0 & m^a_{10} & m^a_{11} \end{pmatrix} \cdot \begin{pmatrix} m^b_{00} & 0 & m^b_{01} & 0 \\ 0 & m^b_{00} & 0 & m^b_{01} \\ m^b_{10} & 0 & m^b_{11} & 0 \\ 0 & m^b_{10} & 0 & m^b_{11} \end{pmatrix}. \tag{3}$$

Note that if a and b are the same, M^{fused} is calculated by just multiplying M^a and M^b. We call gate aggregation that generates q-qubit gate *q-qubit fusion*.

If all of the gates in a circuit is merged into a single gate, the gate updates all the n qubits of the circuit with a $2^n \times 2^n$ matrix. In general, if q becomes large in q-qubit fuxion, time to update a statevector is increased. Figure 1 shows relative elapsed time to update one to six qubits of a 20-qubit state vector with configuration written in Section 4. Update of two qubits took only 40% longer time than of one qubit. This mean that two 1-qubit gates always should be aggregated to a gate. On the other hand, update of 5 and 6-qubits took more than 11 and 31 times longer time than update of 1-qubit respectively. This mean that 5-qubit and 6-qubit gate fusion is effective only if a number of aggregated gates is at least more than 11 and 31. A number of vector registers in any CPUs are limited and not enough to store all complex numbers in M^{fused} if q is more than two. Therefore, while multiplying with M^{fused}, their elements move between registers and memory with additional load and store instructions, and then overheads are increased.

3. New Gate Fusion

Given a quantum circuit, ways to apply gate fusion to its gates are varied and finding the best to shorten simulation time is a challenge . This section overviews gate fusion for Qiskit-Aer [9], which provides quantum computing simulation for Qiskit [1]. Qiskit is an open-source framework for working with noisy quantum computers at the level of pulses, circuits, and algorithms. Qiskit-Aer runs quantum circuits written in Qiskit through the same interface with real devices in IBM Quantum Experience [8]. Qiskit-Aer has various methods to simulate quantum computers, such as state vector, density matrix, MPS and stabilizer. In this paper, we focus performance of state vector method that maintains probability amplitudes with configuration that does not use a noise model.

3.1 Transpilation

Before calculating state vector, Qiskit-Aer processes transpilation, conversion of a quantum circuit to equivalent circuit, to truncates unnecessary gates (such as Identity and Barrier) and not-referred qubits, and applying gate fusion optimization. We enhance the gate fusion of Qiskit-Aer by adding several phases in this gate fusion:

(1) Generate diagonal-matrix gates,
(2) Generate special gates,
(3) Apply one and two-qubit optimization, and
(4) Apply cost-based gate fusion.

Note that the current Qiskit-Aer uses cost-based gate fusion, which we implemented in 2018.

All of the phases are independent and sequentially applied: each gat fusion takes a list of gates from the previous gate fusion, replace gates in the list with new generated gates, and then pass the list to the next gate fusion.

Transpilation of gate fusion incurs additional overheads. We assume that transpilation is a part of simulation time and designed the above transpilation without significant performance degradation. However, for small qubit simulation, transpilation occupies relatively large portion in simulation time. That is, Qiskit-Aer does not enable gate fusion for simulation of fifteen and less qubits by default.

Gate fusion is parallelized if a list of gates is longer than a threshfold (default is 1K). This parallelization is simply designed: divide a list of gates into lists and perform gate fusion for them in parallel. Side effects of this parallelization is that gate fusion may not work efficiently for gates around beginning and ending of each list. However, we believe that such side effects give relatively small impacts on simulation time for quantum circuits that exceeds the threshold.

3.2 Pattern Matching

A diagonal matrix is multiplied with a state vector more efficiently than a normal matrix is. Listing 3 describes that a diagonal matrix d is applied to two qubit qv[2] in a state vector qv.

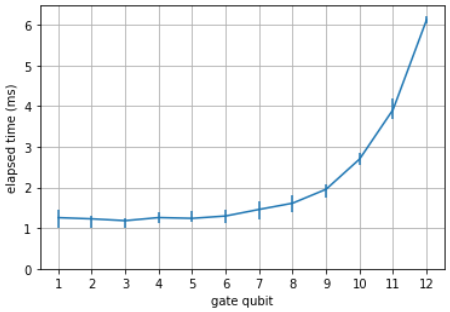

Fig. 2: Elapsed time to update 20-qubit state vector with diagonal matrices

Listing 3: Transformation of two qubits with a diagonal matrix

```
1 void apply_diagonal_matrix(int q[2], complex d
      [4][4]) {
2  assert(q[0] < q[1]);
3  auto m0 = 1<<q[0]; auto m1 = q << q[1];
4 #pragma omp parallel for collapse(3)
5  for (int i=0; i<(1<<n); i+=(1<<(q[1]+1)) {
6    for (int j=0;j<(1<<q[1]);j+=(1<<(q[0]+1)){
7      for (int k=0; k<(1<<q[0]); k++) {
8        int p = i|j|k;
9        qv[p] = qv[p] * m[0][0];
10       qv[p|m0] = qv[p|m0] * m[1][1];
11       qv[p|m1] = qv[p|m1] * m[2][2];
12       qv[p|m0|m1] = qv[p|m0|m1] * m[3][3];
13 } } } }
```

Each element of qv is updated once only by referring itself and an element in d. Comparing with Listing 2, computation resource to multiplication of complex numbers in Listing 3 is relatively small. In addition, matrix multiplication refers only diagonal elements. Vector registers store fewer complex numbers than Listing 3, and then spill is reduced in assembled binaries. Figure 2 shows elapsed time to transform 20-qubit state vector with diagonal matrices varying gate qubits with configuration written in Section 4. By 10-qubits, relative elapsed time based on 1-qubit was lower than 2.0.

Though Qiskit-Aer supports a diagonal-matrix gate as a basis gate, real devices do not support the gate. If a diagonal-matrix gate is decomposed to universal gates, optimized multiplication of a diagonal-matrix shown in Listing 3 is not called. Therefore, we finds sequences of gates with pattern-matching to generate diagonal-matrix gates from a given quantum circuit. This pattern-matching recursively work with following two rules:

(1) If a neighbor of a diagonal-matrix gate is a diagonal-matrix gate, aggregate them as a diagonal-matrix gate (left in Figure 3), and

(2) If both of the neighbors of the diagonal-matrix gate are CX gates with the same control and target qubits, fuse them as a diagonal-matrix gate (right in Figure 3)

In addition to a diagonal-matrix gate, we specializes simulation of a list of CX gates. CX gate swaps the half of probability amplitudes in a state vector. Therefore, after simulating a list of CX gates, a complex number of a probability amplitude (source) moves to the another (destination).

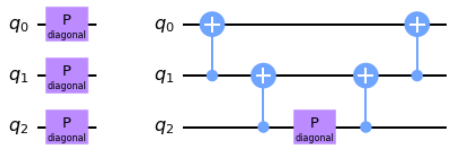

Fig. 3: Patterns to convert to a diagonal-matrix gate

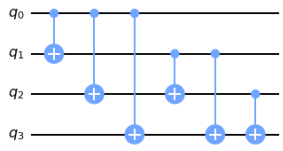

Fig. 4: Patterns to convert to a full-entanglement gate

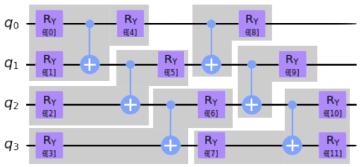

Fig. 5: RealAmplitudes with linear entanglement

Listing 4: Transformation of a list of CX gates (control and target bits are ctrls and tgts respectively)

```
1 void apply_cx_list(int ctrls[n], int tgts[n]) {
2   complex next[1 << n];
3   ctrl_masks = [1 << ctrl for ctrl in ctrls];
4   tgt_masks = [1 << tgt for ctrl in tgts];
5 #pragma omp parallel for collapse(3)
6   for (int i = 0; i < (1<<n); i++) {
7     int idx = i;
8     for (int j = n -1; n >= 0; --n)
9       if (idx & ctrl_masks[j])
10        idx ^= tgt_masks[j];
11    next[i] = qv[i];
12  }
13  qv = copy; }
```

Listing 4 shows pseudocode to efficiently simulate a list of CX. In the loop, a destination index of probability amplitude is calculated with the source index and masks of control and target bits, and then the probability amplitude at source index in qv is copied to at destination index in next. In Listing 4, qv is loaded once and no multiplication of matrices exists.

Note that Listing 4 copies all the probability amplitudes to next that occupies the same size of memory with a state vector. Therefore, this optimization is not enabled if the system does not have enough memory.

3.3 Two-Qubit optimization

As shown in Listing 2, simulators can apply an arbitrary matrix to a state vector, though real devices support only one specific two-qubit operation. If gates around a two-qubit gate update only the two qubits without any entanglements to other qubits, they can be aggregated to a two-qubit gate. For example, CX and Ry gates of a quantum circuit writ-

ten in Figure 5, which is generated with Qiskit circuit library (RealAmplitudes(4, reps = 2, entanglement = 'linear')) are aggregated with gray rectangles. Consequently, a number of gates is reduced from eighteen to six.

Listing 5 describes a pseudocode to identify whether two gates at left and right in a gate sequence gates can be aggregated. Each gate in gates updates its qubits and this qubits has contains and remove methods to check inclusion and remove intersection respectively.

Listing 5: Check whether gates at left and right in gates can be aggregated

```
bool can_aggregate(int left, int right) {
  auto qubits = gates[idx].qubits;
  for (int i = left + 1; i < right; i++)
    if (!qubits.contains(gates[i].qubits))
      qubits.remove(gates[i].qubits);
  return qubits.contains(gates[tgt].qubits);
}
```

Note that Listing 5 is available for aggregation to any-qubit gate. However, because decomposed gates for real devices update at most two qubit, we apply this optimization to generate one and two-qubit gates.

3.4 Cost-based Fusion

As shown in Figure 1, if we apply a gate fusion that aggregates a five-qubit gate, this gate fusion is effective to aggregate at least eleven gates. In most case, use of combination of two and three-qubit gates can achieve more efficient simulation than of five-qubit gate. Because possible combination of gate fusion is varied, we develop an algorithm to apply gate fusion for various qubits while estimating total simulation time.

In our algorithm, we generate a *fusion graph* a DAG that represents possible patterns of gate fusion. A node means a gate in a circuit and each node has a sequence number that represents the position of corresponding gate in a circuit. An edge represents gate fusion that aggregates from the source to the destination gates. If an edge connects from gate 2 to 4, gate fusion aggregates gates 2 and 3. Each edge has a weight that relatively represent estimated time to simulate the generated gate. Figure 6 shows a way to generate a fusion graph with maximum fusion qubit as five.

In Figure 6, we list nodes of all the gate in a circuit and then create edges when 1-qubit gate fusion is applied (Figure 6-1). Next, we create edges when 2-qubit gate fusion is applied (Figure 6-2) and continue this step until when 5-qubit gate fusion is applied (Figure 6-3, 4, and 5). Note that each node has only one edge for each qubit gate fusion and the destination gate of the edge has the largest sequence number in possible destination. For example, in Figure 6-2, we can create edges of 2-qubit gate fusion from gate-2, to gate-4, or 5. Because 5 is the largest sequence number, we create an edge from gate-2 to gate-5 for 2-qubit gate fusion.

Once a fusion graph is generated, we find the shortest path from the start node to the end node. Because weight of edges represents estimated time to simulate generated gates, edges in the shortest path are the best combination of gate fusion to shorten simulation time. In Figure 6, because the path of edges from 1 to 3, from 3 to 4, and from 4 to the end is the shortest, gate fusion generate three gates that produce equivalent quantum state with the original.

4. Evaluation

In this section, we evaluate our gate fusion with various quantum circuits generated with Qiskit Circuit Library. We use Qiskit 0.23.1 to generate quantum circuits and enable our gate fusion on Qiskit-Aer 0.7.2 with MacBook Pro (15-inch, 2018) that consists of Intel Core i7 (6-Core, 2.6GHz), 16GB 2400 MHz DDR4, and macOS Catalina (Version 10.15.5).

4.1 Quantum Circuits

We use twelve types of quantum circuits listed in Table 1. All of them are generated with Qiskit Circuit Library.

Table 1: Quantum Circuits generated with Qiskit Circuit Library

name	qubit	class name
adder	23	WeightedAdder
ansatz_ry	25	RealAmplitudes with full entanglement
ansatz_ry_l	25	RealAmplitudes with linear entanglement
ansatz_ryrz	23	EfficientSU2 with full entanglement
ansatz_ryrz_l	23	EfficientSU2 with linear entanglement
graph_state	25	GraphState
int_cmp	24	IntegerComparator
iqp	25	IQP
pe	20	PhaseEstimation with QuantumVolume as unitary
qft	25	QFT
quad_form	25	QuadraticForm
qv	25	QuantumVolume

adder is a circuit to compute the weighted sum of qubit registers. This quantum circuit consists of 60 CCCX (4-qubit gate), 75 CCX (3-qubit gate), 15 CX (2-qubit gate), and 90 X (1-qubit gate). These gates transform three carry and four sum qubits by referring fifteen state and one control qubits. Each type of gates is iteratively formed while keeping neighbours different types.

ansatz_ry_l is an ansatz circuit for VQE as written in Figure 5. Every qubit is transformed with a Ry gate following a list of CX gates to establish linear entanglements. This pattern of Ry and CX gates is repeated ten times in our configuration. ansatz_ryrz_l uses a Ry and Rz gate instead of a Ry gate of ansatz_ry_l. ansatz_ry and ansatz_ryrz use fewer CX gates to establish full entanglements of Figure 4.

graph_state is a circuit to prepare a graph state. A qubit represents a node and a CZ gate represents an edge. We randamly generate twenty-five edges that connect two of twenty-five nodes. Therefore, a circuit consists of 25 qubits for nodes with 25 CZ gates for edges and 25 H gates for initialization.

int_cmp is a circuit for integer comparator that validates whether quantum state is a given integer number. Several X gates and a CX gate or a CCX gate appear alternatively and the total numbers of X, CX, and CCX gates are 105, 2, 21 in a 24-qubit circuit.

iqp is an instantaneous quantum polynomial (IQP) circuit. Interactions between qubits are defined with a matrix and CP and P gates are generated based on the matrix. We randomly generate interactions for 25 qubits and construct a IQP circuit that consists

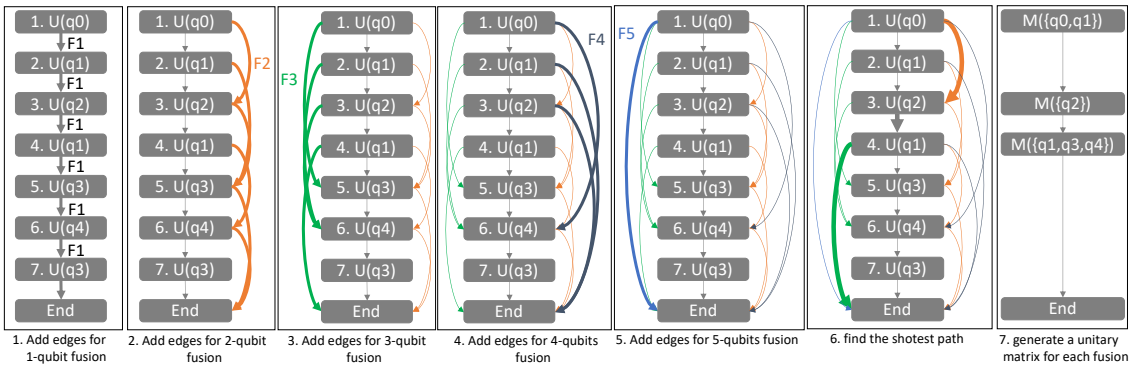

Fig. 6: Cost-based fusion. U(q) means a gate of U to update a qubit q and F*N* means an edge of *N* qubit gate fusion.

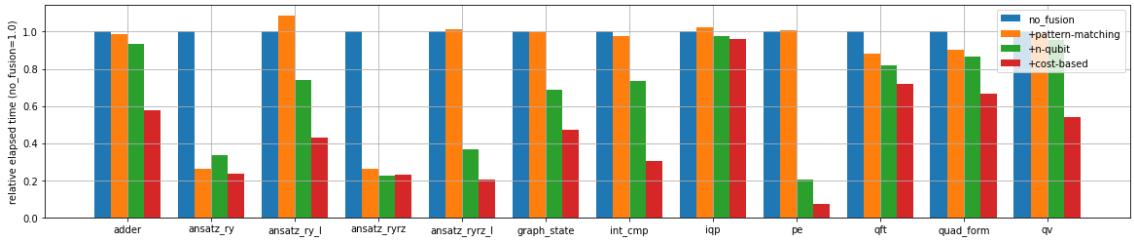

Fig. 7: Relative elapsed time of simulation

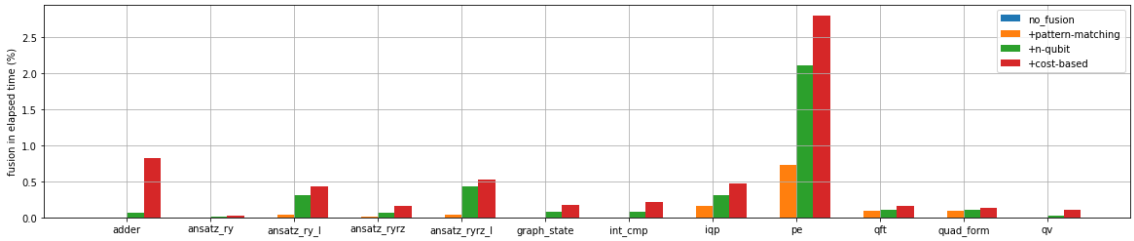

Fig. 8: Percentages of gate fusion in simulation

of around 227 CP gates, 22 P gates, and 50 H gates.

qft is a quantum fourier transform circuit that consists of 300 CP gates, 25 H gates, 12 Swap gates.

quad_form implements a quadratic form on binary variables encoded in qubit registers. We configured three qubits for coefficients and twenty-two qubits for result qubits. Majority gates in this quantum are 297 CP gates.

qv is a circuit to measure quantum volume. We generate 25-qubit circuits that consists of 300 unitary-matrix gates.

4.2 Performance

We evaluated following four configurations:

- no_fusion disabled all the optimization of gate fusion,
- +pattern_matching enables only gate fusion with pattern matching written in Subsection 3.2,
- +n_qubit adds gate fusion of one-qubit and two-qubit optimization written in Subsection 3.3,
- +cost – based enables all the gate fusion by adding cost-based optimization written in Subsection 3.4,

Because the system has enough memory to simulate the quantum circuits, pattern-matching for entanglements was enabled in any simulation. Figure 7 shows performance improvement of gate fusion by making no_fusion the baselines and Figure 8 shows percentages of gate fusion in total simulation time.

In adder, pattern-matching did not find any candidates of diagonal-matrix gates and entanglement gates, but two-qubit gate fusion reduced 16 gates and improved simulation time by 7.6 %. Cost-based gate fusion additionally reduced 148 gates and improved simulation time by 38.1 %. Consequently, our gate fusion reduced by 42.1 % from the baseline. Gates in adder frequently transform or refer the same two qubits and then two-qubit and cost-based gate-fusion worked performance. For further optimization, additional pattern-matching for X gates (with or without control qubits) will be possible because they transform qubits similarly as entanglements of Figure 4.

On the other hand, in ansatz_ry and ansatz_ryrz, pattern-matching improved simulation time by 73.6 % and 73.9 % while reducing 2991 and 2521 gates respectively. Our pattern-matching found lists of CX gates to establish full entanglements in these quantum circuits. Other gate fusion work also and consequently our gate fusion reduced simulation time of ansatz_ry and ansatz_ryrz by 76.3 % and 76.8 % while reducing 190 and 427 gates respectively.

Our pattern-matching does not replace lists of CX gates in ansatz_ry_l and ansatz_ryrz_l because their length is less than the threshold (a number of qubits). However, two-qubit gate fusion significantly reduced their gates: 276 of 541 gates in ansatz_ry_l and 507 of 750 gates in ansatz_ryrz_l. In gates for linear en-

tanglements, a qubit becomes a target and control qubit of two CX gates. Two-qubit gate fusion aggregates all of the rotation gates for the qubit with CX gates as shown in Figure 5, and then simulation is optimized. In addition, cost-based gate fusion reduced their 121 gates and 132 gates and optimized simulation time of ansatz_ry_l and ansatz_ryrz_l became 43.3 % and 20.5 % respectively. Simulation time of graph_state and int_cmp were improved similarly by 53.0 % and 69.4 %, respectively.

qft and quad_form consist of many CP gates and our gate fusion replaces their lists of CP gates with diagonal-matrix gates. Other gate fusion also worked their simulation performance and consequently their 28.2 % and 33.3 % of simulation time were reduced.

Two-qubit and cost-based gate fusion worked well for pe: improving 79.8 % of simulation time while reducing gates from 60747 gates to 5911 gates. In pe, 13-qubit QuantumVolume is simulated three times by adding controlled qubits. Qiskit circuit library decomposes the target quantum circuits to U and CX, and then adds a control qubit for them. Two-qubit and cost-based gate fusion can re-composed such decomposed primitive gates in pe and improved performance of simulation.

In all quantum circuits, gate fusion took less than 1 % time in simulation excepting pe. In pe, pattern-matching, two-qubit, and cost-based gate fusion took 0.40 second, 0.76 second, and 0.38 second, respectively. pe consists of 63664 gates which exceeds a threshold to enable parallelization of gate fusion and 63178 gates are reduced. Therefore, parallelization of gate fusion and generation of new matrices in pe took relatively more time than in others.

5. Conclusion

Simulation of quantum computing needs huge computing resources if large qubits are simulated and gate fusion is an effective approach to shorten simulation time. We proposed a new method of gate fusion cost-based gate fusion that aggregates gates in a circuit with estimation of total simulation time. Our evaluation shows that our gate fusion improve any simulation types of quantum circuits.

References

[1] Abraham, H., AduOffei, Agarwal, R., Akhalwaya, I. Y., Aleksandrowicz, G., Alexander, T., Amy, M., Arbel, E., Arijit02, Asfaw, A., Avkhadiev, A., Azaustre, C., AzizNgoueya, Banerjee, A., Bansal, A., Barkoutsos, P., Barnawal, A., Barron, G., Barron, G. S., Bello, L., Ben-Haim, Y., Bevenius, D., Bhobe, A., Bishop, L. S., Blank, C., Bolos, S., Bosch, S., Brandon, Bravyi, S., Bryce-Fuller, Bucher, D., Burov, A., Cabrera, F., Calpin, P., Capelluto, L., Carballo, J., Carrascal, G., Chen, A., Chen, C.-F., Chen, E., Chen, J. C., Chen, R., Chow, J. M., Churchill, S., Claus, C., Clauss, C., Cocking, R., Correa, F., Cross, A. J., Cross, A. W., Cross, S., Cruz-Benito, J., Culver, C., Córcoles-Gonzales, A. D., Dague, S., Dandachi, T. E., Daniels, M., Dartiailh, M., DavideFrr, Davila, A. R., Dekusar, A., Ding, D., Doi, J., Drechsler, E., Drew, Dumitrescu, E., Dumon, K., Duran, I., EL-Safty, K., Eastman, E., Eberle, G., Eendebak, P., Egger, D., Everitt, M., Fernández, P. M., Ferrera, A. H., Fouilland, R., FranckChevallier, Frisch, A., Fuhrer, A., Fuller, B., GEORGE, M., Gacon, J., Gago, B. G., Gambella, C., Gambetta, J. M., Gammanpila, A., Garcia, L., Garg, T., Garion, S., Gilliam, A., Giridharan, A., Gomez-Mosquera, J., Gonzalo, de la Puente González, S., Gorzinski, J., Gould, I., Greenberg, D., Grinko, D., Guan, W., Gunnels, J. A., Haglund, M., Haide, I., Hamamura, I., Hamido, O. C., Harkins, F., Havlicek, V., Hellmers, J., Herok, Ł., Hillmich, S., Horii, H., Howington, C., Hu, S., Hu, W., Huang, J., Huisman, R., Imai, H., Imamichi, T., Ishizaki, K., Iten, R., Itoko, T., JamesSeaward, Javadi, A., Javadi-Abhari, A., Javed, W., Jessica, Jivrajani, M., Johns, K., Johnstun, S., Jonathan-Shoemaker, K, V., Kachmann, T., Kale, A., Kanazawa, N., Kang-Bae, Karazeev, A., Kassebaum, P., Kelso, J., King, S., Knabberjoe, Kobayashi, Y., Kovyrshin, A., Krishnakumar, R., Krishnan, V., Krsulich, K., Kumkar, P., Kus, G., LaRose, R., Lacal, E., Lambert, R., Lapeyre, J., Latone, J., Lawrence, S., Lee, C., Li, G., Liu, D., Liu, P., Maeng, Y., Majmudar, K., Malyshev, A., Manela, J., Marecek, J., Marques, M., Maslov, D., Mathews, D., Matsuo, A., McClure, D. T., McGarry, C., McKay, D., McPherson, D., Meesala, S., Metcalfe, T., Mevissen, M., Meyer, A., Mezzacapo, A., Midha, R., Minev, Z., Mitchell, A., Moll, N., Montanez, J., Monteiro, G., Mooring, M. D., Morales, R., Moran, N., Motta, M., MrF, Murali, P., Müggenburg, J., Nadlinger, D., Nakanishi, K., Nannicini, G., Nation, P., Navarro, E., Naveh, Y., Neagle, S. W., Neuweiler, P., Nicander, J., Niroula, P., Norlen, H., NuoWen-Lei, O'Riordan, L. J., Ogunbayo, O., Ollitrault, P., Otaolea, R., Oud, S., Padilha, D., Paik, H., Pal, S., Pang, Y., Pascuzzi, V. R., Perriello, S., Phan, A., Piro, F., Pistoia, M., Piveteau, C., Pocreau, P., Pozas-Kerstjens, A., Prokop, M., Prutyanov, V., Puzzuoli, D., Pérez, J., Quintiii, Rahman, R. I., Raja, A., Ramagiri, N., Rao, A., Raymond, R., Redondo, R. M.-C., Reuter, M., Rice, J., Riedemann, M., Rocca, M. L., Rodríguez, D. M., RohithKarur, Rossmannek, M., Ryu, M., SAPV, T., SamFerracin, Sandberg, M., Sandesara, H., Sapra, R., Sargsyan, H., Sarkar, A., Sathaye, N., Schmitt, B., Schnabel, C., Schoenfeld, Z., Scholten, T. L., Schoute, E., Schwarm, J., Sertage, I. F., Setia, K., Shammah, N., Shi, Y., Silva, A., Simonetto, A., Singstock, N., Siraichi, Y., Sitdikov, I., Sivarajah, S., Sletfjerding, M. B., Smolin, J. A., Soeken, M., Sokolov, I. O., Sokolov, I., SooluThomas, Starfish, Steenken, D., Stypulkoski, M., Sun, S., Sung, K. J., Takahashi, H., Takawale, T., Tavernelli, I., Taylor, C., Taylour, P., Thomas, S., Tillet, M., Tod, M., Tomasik, M., de la Torre, E., Trabing, K., Treinish, M., TrishaPe, Tulsi, D., Turner, W., Vaknin, Y., Valcarce, C. R., Varchon, F., Vazquez, A. C., Villar, V., Vogt-Lee, D., Vuillot, C., Weaver, J., Weidenfeller, J., Wieczorek, R., Wildstrom, J. A., Winston, E., Woehr, J. J., Woerner, S., Woo, R., Wood, C. J., Wood, R., Wood, S., Wood, S., Wootton, J., Yeralin, D., Yonge-Mallo, D., Young, R., Yu, J., Zachow, C., Zdanski, L., Zhang, H., Zoufal, C., Zoufalc, a kapila, a matsuo, bcamorrison, brandhsn, nick bronn, brosand, chlorophyll zz, csseifms, dekel.meirom, dekelmeirom, dekool, dime10, drholmie, dtrenev, ehchen, elfrocampeador, faisaldebouni, fanizzamarco, gabrieleagl, gadial, galeinston, georgios ts, gruu, hhorii, hykavitha, jagunther, jliu45, jscott2, kanejess, klinvill, krutik2966, kurarrr, lerongil, ma5x, merav aharoni, michelle4654, ordmoj, sagar pahwa, rmoyard, saswati qiskit, scottkelso, sethmerkel, shaashwat, sternparky, strickroman, sumitpuri, tigerjack, toural, tsura crisaldo, vvilpas, welien, willhbang, yang.luh, yotamvaknin ibm and Čepulkovskis, M.: Qiskit: An Open-source Framework for Quantum Computing (2019).

[2] Broughton, M., Verdon, G., McCourt, T., Martinez, A. J., Yoo, J. H., Isakov, S. V., Massey, P., Niu, M. Y., Halavati, R., Peters, E., Leib, M., Skolik, A., Streif, M., Dollen, D. V., McClean, J. R., Boixo, S., Bacon, D., Ho, A. K., Neven, H. and Mohseni, M.: TensorFlow Quantum: A Software Framework for Quantum Machine Learning (2020).

[3] Corporation, M.: The Q# Programming Language (2018).

[4] Cross, A. W., Bishop, L. S., Smolin, J. A. and Gambetta, J. M.: Open quantum assembly language, *arXiv preprint arXiv:1707.03429* (2017).

[5] Doi, J. and Horii, H.: Cache Blocking Technique to Large Scale Quantum Computing Simulation on Supercomputers, *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, (online), DOI: 10.1109/qce49297.2020.00035 (2020).

[6] Google: Cirq: A python framework for creating, editing, and invoking noisy intermediate scale quantum circuits (2018).

[7] Häner, T. and Steiger, D. S.: 0.5 petabyte simulation of a 45-qubit quantum circuit, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, (online), DOI: 10.1145/3126908.3126947 (2017).

[8] IBM: IBM Quantum Experience.

[9] Qiskit: Qiskit Aer: A High Performance Simulator Framework for Quantum Circuits (2019).

[10] Smelyanskiy, M., Sawaya, N. P. D. and Aspuru-Guzik, A.: qHiPSTER: The Quantum High Performance Software Testing Environment (2016).

[11] Suzuki, Y., Kawase, Y., Masumura, Y., Hiraga, Y., Nakadai, M., Chen, J., Nakanishi, K. M., Mitarai, K., Imai, R., Tamiya, S., Yamamoto, T., Yan, T., Kawakubo, T., Nakagawa, Y. O., Ibe, Y., Zhang, Y., Yamashita, H., Yoshimura, H., Hayashi, A. and Fujii, K.: Qulacs: a fast and versatile quantum circuit simulator for research purpose (2020).