Vol.2019-HPC-171 No.9
2019/9/20

量子コンピューティングシミュレータにおける最適化に関する考察

高橋 ひとみ[1,a)]　土井 淳[1,b)]　堀井　洋[1,c)]

概要：量子コンピュータが実用化されつつある現在、古典コンピュータによる量子コンピューティングのシミュレーションはアルゴリズムの動作検証を行う上で有用である。しかし、計算が困難な処理を可能とする量子コンピュータを古典コンピュータでシミュレーションするには、小規模の量子回路を実行する際にも大量の計算時間が必要となる。そこで本研究では量子コンピューティングのシミュレーション時間を短縮するため、最適化の手法を量子コンピューティングシミュレータのオープンソフトウェアである Qiskit Aer に組み込み、その性能評価を行った。最適化の手法として、複数の量子ゲートを同時にシミュレーションすることによるキャッシュヒット率の向上を目的としたゲートフュージョン、複素数の計算速度を高速化するための SIMD 命令による最適化を Qiksit Aer に追加した。これらの最適化手法により、計測を行ったシナリオで最大 7.2 倍の性能向上がみられた。

A Study of Optimiziation Methods for A Quantum Computing Simulator

1. はじめに

量子コンピューティングの研究が進み、今まで論理的なものとして扱われていた量子アルゴリズムが実機上で検証可能となった。実機上で実行できる Qubit 数は増加しており、量子コンピュータが実用的に使用できるとされる 50qubit [1] 以上の量子アルゴリズムの実行は、既存のコンピュータではシミュレーションができないとされている。なぜならば量子シミュレーションでは多次元の複素ベクトルを扱うため、それらの状態を保持するためには莫大なメモリと計算コストが必要となるからである。例えば、複素数のデータ型に倍精度を用いた場合、n qubit のシミュレーションに必要なメモリ量は 2^{n+4} バイトとなる。つまり 50qubit のシミュレーションを倍精度を用い、既存の計算資源で行う場合には、16PB のメモリ量が必要となるためシミュレーションが困難と考えられる。

シミュレーションは qubit 数が小さい量子アルゴリズムしか実行できないが、簡単に何度も実行でき、ノイズの影響を受けないため新しいアルゴリズムの動作検証、パラメータによる性能の影響などのテストを行うには非常に有用なツールとなる。しかし、量子コンピューティングのシミュレーションでは前述した通り、小さい qubit 数でもメモリ量は多く、計算時間が長くなるため、繰り返しアルゴリズムを実行し検証するためには、計算時間が短い効率的なシミュレーションの実行が重要となる。

そこで本稿では、オープンソースのシミュレータである Qiskit Aer 上にゲートフュージョンによるキャッシュヒット率の向上、ゲート計算の SIMD 化による計算速度の向上を行い、量子コンピューティングシミュレーションの最適化を行った。これらの手法により最適化前のシミュレータと比較し、35 qubit、Quantum Volume のシナリオのおいてシミュレーション時間の性能がが 7.2 倍に向上した。

この後の本論文の構成として、まず第 2 節で本論文と同様に量子シミュレーションに関する関連研究を紹介し、第 3 節で本論文が取り扱う量子シミュレーションの動作を説明する。次に第 4 節で今回用いた最適化手法について述べ、それらの評価を第 5 節にて示す。最後に第 6 節にてまとめについて述べる。

[1] 日本 IBM 株式会社　東京基礎研究所
IBN Japan, Ltd., 19-21, Nihonbashi, Hakozaki-cho, Chuo-ku, Tokyo 103–8501
[a)] hitomi@jp.ibm.com
[b)] doichan@jp.ibm.com
[c)] horii@jp.ibm.com

2. 関連研究

この節では量子コンピューティングシミュレータおよびそこに使用されている最適化について述べる。マイクロソフト社が提供する量子コンピュータ向け開発 Kit [2] では Q#というプログラミング言語を用い量子アルゴリズムを実装する。この Q#で書かれたプログラムからは、開発キットに内包するローカルで実行するシミュレータ、クラウド上で実行するシミュレータ、または実機において実行可能なコードを生成できる。Q#プラグラムの構造は既存の C#と非常に近いが、量子コンピュータに特化した DSL であり、量子アルゴリズムを簡単かつ明確に記述できるよう設計されてある [3]。Q#で作成されたプログラムは C#または Python で作成されたプログラムから呼び出し可能であり、それにより外部との連携を行う。最近では Github にオープンソースとしてソースが公開された。

QuEST(Quantum Exact Simulation Toolkit) [4] は Jones らにより提案されたシミュレーションである。C 言語で記述されており、ソースコードはオープンソースとして git 上に公開されている [5]。OpenMP を用いた並列化、MPI による複数ノードへの対応もしており、かつシングル GPU のサポートもしている。シミュレーションを動作させるホストとして、通常のラップトップからスーパーコンピュータまで同一のプログラムで実行できるよう設計されている。量子アルゴリズムを記述するために必要なゲートの関数群は全て、C のライブラリとして提供されており、QuEST を用いて量子アルゴリズムを作成する場合は、それらの関数が使用された C プログラムとなる。

Intel 社 Parallel ComputingLab で開発された qHiPSTER(The Quantum High Performance Software Testing Environment) [6] は、マルチノード、マルチコアをサポートし、大規模な分散環境で多くの量子ビット数がシミュレーション可能なシミュレータである。コードは C 言語で記述されており、ソースコードも git 上に公開されている [7]。論文 [6] では 1000 ノード、32TB RAM の実験環境を用い、40qubit までのシミュレーションを実行した。SIMD、マルチスレッド、ゲートフュージョンなどの最適化を用い、オーバーヘッドの削減を実現している。

ProjectQ [8] は Thomas らが開発した Python ベースのシミュレーションであり、オープンソフトウェアとして公開されている [9]。ただし一部のコアとなる処理は C++ のモジュールになっており、ゲートフュージョン、SIMD などの最適化を用い高速化を実現している。また、シングルノードだけではなく MPI を使用したマルチノードでも動作可能なよう実装されており、論文では 8192 ノード、0.5PB のメモリを用い 45qubit までのシミュレーションを実現している。量子プログラム用に SDK も公開しており、Python コードに埋め込む形で DSL として記述できる。

Forest [10] は Rigetti 社が提供している開発環境である。量子プログラミングには Quil という独自言語を用いて行われるが、pyQuil という Python ライブラリを用い、Python プログラムとしても記述可能である。量子シミュレータはサービスとして API が公開されており、26qubit まで無料でシミュレーションが実行可能である。

3. 量子コンピューティングシミュレーション

本節では最適化の目的とする量子コンピューティングの基礎について説明し、その後、量子コンピューティングのシミュレータの基本動作および Qiskit Aer について説明をする。

3.1 量子コンピューティング

本稿は、1 量子ビットの回転と 2 量子ビットからなる CNOT ゲートの量子回路を実行する、ユニバーサル量子コンピューティングのシミュレーションに焦点を当てる。このユニバーサル量子コンピューティングでは、1 量子ビットの回転と 2 量子ビットの CNOT を用い、全ての回路が表現できる。量子回路では量子ゲートと呼ばれる量子計算を各量子ビットに適応させ、量子プログラムを表現する。最小となる単位である 1 量子ビットは式 1 のようにベクトルで表現される。

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{1}$$

既存のコンピュータの場合、0 または 1 で表現されるビットだが、量子ビットの場合、重ね合わせを用い、式 2 に示すように、確率振幅と呼ばれる式 3 を満たす複素数で示される。

$$|\varphi\rangle = a_0 |0\rangle + a_1 |1\rangle \tag{2}$$

$$|a_0|^2 + |a_1|^2 = 1 \tag{3}$$

計測 (Measure) を行いビットを確定する際、0 もしくは 1 はそれぞれ $|a_0|^2$、$|a_1|^2$ の確率に従い決定される。そのため 1 量子ビットの状態を表現する複素数が必要となる。また、量子ビットが複数となる場合はテンソル積が使用され、式 4 のように表現でき十進数での表現を行う場合もある。そして n 量子ビットでは式 5 と表現できる。ただし、各状態である $|i\rangle$ は a_i の確率振幅をそれぞれ持つことになる。量子回路のシミュレーションには n 量子ビットのレジスタの状態を表現するためには、これらの 2^n 個の複素数を格納する必要がある。

$$|2\rangle = |1\rangle \otimes |0\rangle = |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \tag{4}$$

$$|\varphi\rangle = \sum_{i=0}^{2^n-1} a_i \, |i\rangle \tag{5}$$

3.2 Qiskit Aer

Qiskit Aer [11] は Github 上で公開されているオープンソースの量子コンピューティングシミュレータである。Python の API をサポートし、実際のシミュレーション計算部分は C++を用い実装されている。Qiskit Aer では Qiskit Terra と呼ばれる SDK が存在し、これにより作成された量子プログラミングが Qiskit Aer のシミュレータもしくは実機上で実行される。Qiskit Terra によるサンプルプログラムを以下に示す。

```
1  from qiskit import QuantumRegister,
       ClassicalRegister
2  from qiskit import QuantumCircuit, execute,
       Aer
3
4  q = QuantumRegister(3)
5  c = ClassicalRegister(3)
6  qc = QuantumCircuit(q, c)
7
8  qc.u3(pi/2, pi/2, pi/2, q[0])
9  qc.measure(q, c)
10
11 backend = Aer.get_backend('qasm_simulator')
12 job_sim = execute(qc, backend, shots=1)
13 sim_result = job_sim.result()
```

Qiskit Terra における量子プログラミングの基本的な処理は、まず、量子コンピュータのレジスタを定義し、各量子ビットに行うゲート操作を定義する流れとなる。サンプルプログラムでは、L.1、L.2 で量子レジスタ、クラシカルレジスタを確保し、量子回路を Qiskit Aer 上で実行するためのモジュールをインポートしている。L.4 で 3 量子ビットの量子レジスタを作成し、L.5 で 3 ビットのクラシカルレジスタ、L.6 では、それらのレジスタを含む量子回路を作成している。L.8 で U3 のゲートを量子ビットの 0 ビット目に適応する。この U3 は X, Y, Z の軸に対し、任意に回転を行うゲートである。L.9 では全ての量子ビットにおいて Measure を行い、それらの結果をクラシカルレジスタに保存する。L.11 で作成した量子回路を実行するバックエンドを指定する。このサンプルコードはシミュレーションである Qiskit Aer での実行を選択しているが、実機での実行も指定できる。L.12 で量子回路の実行回数を shots の引数で指定し、シミュレーションを実行する。最後に L.13 でシミュレーションの結果を取得する。

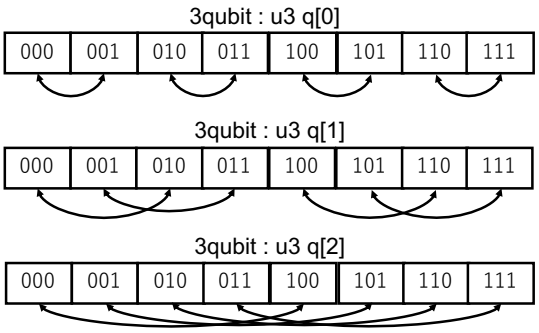

図 1 U3 ゲートの計算

n 量子ビットレジスタが宣言される際、シミュレータ上では 2^n 個の状態を保持することになる。そのため、2^n 個の配列が確保され、各ビットの状態を示す確率振幅（複素数）が各メンバに保持される。各ゲートの計算は確率振幅が保持されている配列から、2 つの要素を取り出し引数で渡された数値と計算をした後、更新後の二数の値が再び配列へ格納される。二数を取得した際の具体的な計算は、配列から取り出した二個の複素数をそれぞれ 1 列の行列 A とし、回転を与えられた場合の引数を 2x2 の行列 B とすると、式 6 で示すように A と B の内積である AB が、ゲート操作を適応した新しい状態 C となる。

$$C = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \tag{6}$$

図 1 に U3 ゲートの計算に発生する配列のアクセス例を示す。ゲートの計算を適応するために指定されている量子ビットは図の上より、0、1、2 ビットを示す。量子ビットが 0 である場合、0 ビット目が 0 と 1 のメンバが取得される。2 である場合は 2 ビット目が 0 と 1 のメンバが取得される。このようにシミュレータ上では、各ゲートの計算毎に全配列のメンバの要素を参照、更新しながらシミュレーションが実行される特質から、シミュレーションの最適化には計算スピードの向上に加え、キャッシュの効率化が大きく影響する。

また、サンプルプログラムの L.9 の Measure という操作は、実機上では観測を行い量子状態を確定する操作である。ゲートの計算が終了し、Measure が呼び出されると、シミュレータは確率振幅が格納されている配列より、それぞれの確率を算出し、サンプリングによって結果を確定する。ノイズを含まないシミュレーションである場合、この確率振幅の値は何度計算を行っても等しくなるため、同じ回路を複数回実行する場合はこのサンプリングのみを行う。

4. シミュレーションにおける最適化手法

ここでは、本論文で適応した量子コンピューティングシミュレーションにおける最適化について説明する。まず初

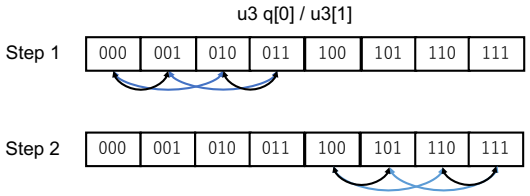

図 2　ゲートフュージョンによるメモリアクセス

めに複数のゲート操作を 1 つにまとめ計算することで、配列からメンバを取得、更新するためのロード、ストア命令を削減するゲートフュージョンという手法を説明する。次に、ゲートフュージョンを行い、配列に格納されてある状態の更新を行う際、複素数の積および和を計算する処理となる。この計算を SIMD 命令を用い、計算の高速化を行った。

4.1　ゲートフュージョンによる最適化

量子シミュレーションではゲートの操作毎に配列のメンバから、二数の複素数をロードしゲート操作の計算を行い、その値を再び配列へ格納する。この配列の大きさは n 量子ビットのシミュレーションに対し、2^n のサイズとなり、ゲートの計算毎にすべての値のロードとストアを繰り返す。さらに、量子シミュレーションのゲート操作で行う値の更新のための計算処理は少ないため、メモリの転送速度がシミュレーションの速度に大きく影響を与える。そこで、一度ロードした変数を効率的に使えるよう、1 回のループで複数のゲート操作を行う最適化を追加した。これをゲートフュージョンと呼ぶ [6]。

通常のゲート操作は 1 で示すように、二数を配列よりロードしストアした後、次のループにより再び違う二つのメンバである複素数をロード、ストアするという操作を繰り返すが、ゲートフュージョンを行うことで、複数のゲート操作を 1 回のループで処理でき、ロードストアの回数を減少できる。これによりメモリのキャッシュ効率が高くなり、シミュレーション速度が向上する。

ゲートフュージョンでは複数のゲートを一つの処理としてまとめる必要がある。各ゲートの操作は行列で示すことができ、これらの行列を一つにまとめる場合、各行列の積、もしくはテンソル積を用い合成することで、一つの行列として処理できる。例えば 2 量子ビットの量子回路において、1 ビット目に Pauli-X(NOT) ゲート、2 ビット目に Pauli-Y ゲートの操作を行った場合、2 つのゲートを適応した回路は式 7 で示すように、各ゲート操作を示す行列のテンソル積となる。この合成した行列と各ビットの状態との内積により、2 ゲートを適応した状態に更新できる。このように複数のゲートを 1 つの行列に変換することで、ゲートフュージョンを実現する。

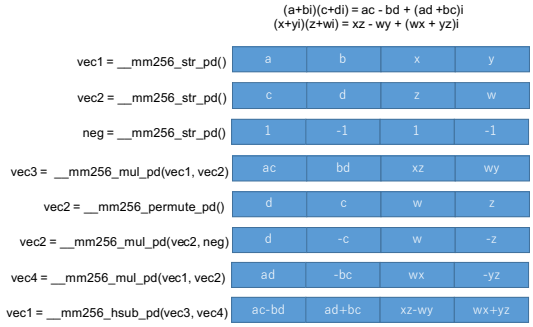

図 3　SIMD 命令による複素数の掛け算

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \end{bmatrix} \tag{7}$$

4.2　SIMD 命令による最適化

SIMD 命令とは 1 命令で複数のデータを演算できるベクトル演算である。POWER や X86 などの一般的な CPU はこれらの命令をサポートしており、通常の命令より少ない数で複数のデータを処理できる。本稿では、この SIMD 命令をゲート処理の計算に用い、複素数の計算を高速化した。ゲートフュージョンを行わない通常の処理となる、1 ゲートの操作では式 6 が計算処理となり、更新される値はループ毎に二変数づつ更新される。具体的な式は $a_1 * b_{11} + a2 * b_{12}$, $a_1 * b_{21} + a_2 * b_{22}$ となり、全ての変数は複素数となる。式 6 は通常のゲート操作であり、前節で説明したゲートフュージョンを行うと、この行列式が変更される。例えば 2 つのゲートを 1 つの行列に合成した場合、4x4 の行列となり、計算は式 8 となる。3 つのゲートをフュージョンした場合は同様に 8x8 の行列となる。

$$C = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \tag{8}$$

最適化では先に示したこれらの行列計算部分を SIMD 命令を用い高速化した。具体的な SIMD 命令による計算方法を図 3 に示す。このように、4 つの複素数をレジスタにロードし、積や加算などの SIMD 命令を用い、更新される二数の複素数を算出できる。この例では通常の 1 ゲート処理となるが、ゲートフュージョンを行い行列のサイズが大きくなった場合もほぼ図 3 の計算を繰り返すことで値を算出できる。

5.　評価

本節ではゲートフュージョンおよび SIMD 命令による最適化を量子コンピューティングシミュレータに追加し、実

行時間を測定した。まず、始めに予備実験として、第 4 節で解説した SIMD 命令による最適化によって、計算速度が向上しているかを調査するため計算速度のみを測定した。その後、二つの最適化を追加したシミュレータをサーバ上で実行し、量子ビットを変化させ、各量子回路の実行時間を測定した。

5.1 予備実験

予備実験として SIMD 命令を用いた複素数演算の計算速度を測定した。シミュレータと同様な複素数による行列計算を行う計算処理のみを抽出し、通常の処理と SIMD 命令による計算処理を測定した。測定環境は MacBook Pro、CPU: Intel Core i7(4 コア)、メモリ 16GB のラップトップを用い、28 量子ビットにおける計算処理を行った。実験結果を図 4 に示す。X 軸は計算に用いた行列のサイズ、Y 軸は実行時間（ミリ秒）を示す。SIMD 命令を用いた計算では速度が通常の命令と比較し、すべての行列サイズにおいて速度が向上した。特に行列サイズが 5 である場合、SIMD 命令の計算速度は通常の命令と比較し計算速度が 2.3 倍となった。この予備実験により、SIMD 命令によって、量子コンピューティングシミュレータの計算処理のみの速度が向上することが示せた。

5.2 評価環境

本節では X86 のサーバ上で量子コンピューティングシミュレータを実行し、実行時間を測定した。評価に使用したサーバのスペックは、CPU: Intel Xeon Gold 6140 (Skylake)、36 コア、72 ハードウェアスレッド、メモリ: 1.5TB となっている。このホストに Ubuntu18.04.2 をインストールし、シミュレーションを実行した。

量子シミュレーションは Qiskit Aer 0.2.2 に第 4 節で説明した、ゲートフュージョンおよび SIMD 命令の最適化を追加したソフトウェアを使用した。

量子回路として QFT (Quantum Fourier transform) およびランダム回路となる QV(Quantum Volume、Depth:10) を用い、それぞれの実行時間を測定した。測定はすべて 1 ショットで実行し、ランダムのシードはすべて同一に設定している。また、シミュレーションは全てノイズなしとなっている。

5.3 ゲートフュージョンおよび SIMD 命令による実行時間への効果

Qiskit Aer にゲートジューションおよび SIMD 命令による最適化を追加し、QFT、QV のシミュレーション速度を計測した。図 5、6 に qubit 毎の実行時間を示す。X 軸は実行した量子回路の量子ビット数、Y 軸は実行時間のログスケールとなっている。Base は最適化を追加していないオリジナルの Qiskit Aer、Gate Fusion はゲートフュージョンのみを追加、Fusion+SIMD ではゲートフュージョンに加え、SIMD 命令による最適化を追加した実行時間をそれぞれ示している。

QFT、QV ともに最適化を追加することで実行時間に対する性能の向上が見られた。図 7 では、QV、35 qubit の実行時間のみを示している。ゲートフュージョンのみでは Orignal の実行速度と比較し 6.8 倍、ゲートフュージョンに加え SIMD 命令による最適化を追加した場合は 7.2 倍の性能向上が見られた。

これらの評価によって、2 つの最適化を追加することで量子シミュレーションの実行速度における性能向上を示すことができた。特にゲートフュージョンによる性能向上の効果は非常に大きく、量子シミュレーションでは計算速度の効率化よりも、メモリキャッシュの効率化が性能へ非常に大きく影響することがわかった。これは量子シミュレータの処理は、配列のメンバをレジスタにロードした後の計算処理が少なく、すぐにレジスタからメモリをストアするという特性から、シミュレーションの実行速度は計算処理による律速ではなく、メモリ転送速度による律速になるからである。

6. まとめ

本稿では量子コンピューティングシミュレータのソフトウェアである Qiskit Aer にゲートフュージョンおよび SIMD 命令による最適化を追加し、シミュレーションの実行速度を評価した。ゲートフュージョンは複数のゲートを 1 つのゲートに合成することで、配列に格納されてある変数のロードストア回数を削減し、メモリキャッシュの効率化を行う最適化である。SIMD 命令による最適化では、変数のロード後、ゲート操作となる行列と配列のメンバの積を求める計算処理を SIMD 命令に置き換え、計算処理速度の向上を行った。これらの最適化により QFT、QV ともに最適化による性能向上が見られた。具体的には QV 35qubit において、最適化なしのシミュレータとの実行速度と比較しゲートフュージョンのみでは 6.8 倍、ゲートフュージョンに加え SIMD 命令の最適化を追加した場合は 7.2 倍の性能向上が見られた。

参考文献

[1] Knight, W.: IBM Raises the Bar with a 50-Qubit Quantum Computer, *MIT Technology Review* (2017).
[2] Microsoft: Microsoft Quantum Development Kit, `https://www.microsoft.com/en-us/quantum/development-kit`.
[3] Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A. and Roetteler, M.: Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL, *Proceedings of ACM RWDSL 2018* (2018).
[4] Jones, T., Brown, A., Bush, I. and Benjamin, S.: QuEST

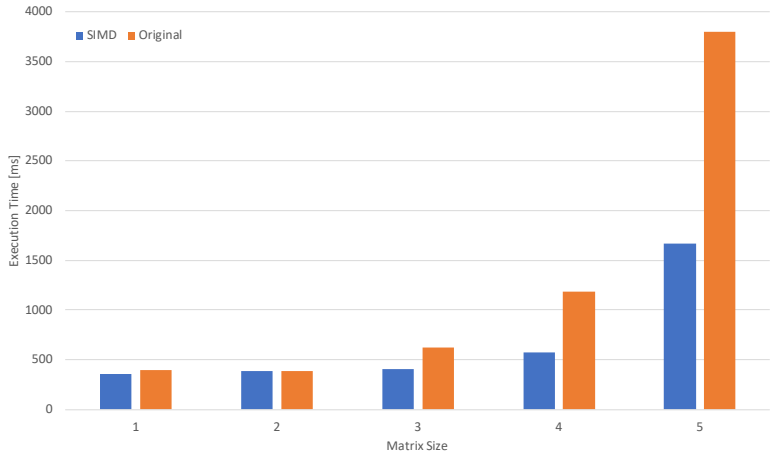

図 4　SIMD 命令による計算速度の結果

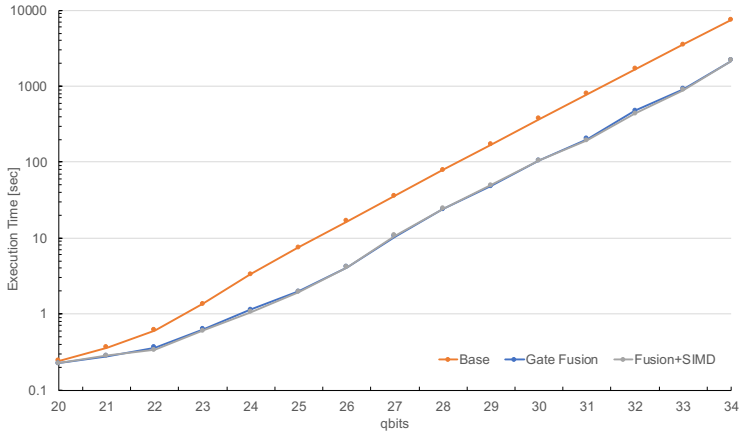

図 5　QFT の実行時

and High Performance Simulation of Quantum Computers, *arXiv:1802.08032* (2018).

[5] QuEST (Source Code): `https://github.com/aniabrown/QuEST`.

[6] Smelyanskiy, M., Sawaya, N. P. D. and Aspuru-Guzik, A.: qHiPSTER: The Quantum High Performance Software Testing Environment, *arXiv:1601.07195* (2016).

[7] qHiPSTER (Source Code): `https://github.com/intel/Intel-QS`.

[8] Häner, T. and Steiger, D. S.: 0.5 Petabyte Simulation of a 45-qubit Quantum Circuit, *Proceedings of ACM SC 2017* (2017).

[9] ProjectQ (Source Code): `https://github.com/ProjectQ-Framework/ProjectQ`.

[10] Rigetti: Forest, `https://www.rigetti.com/forest`.

[11] Aleksandrowicz, G., Alexander, T., Barkoutsos, P., Bello, L., Ben-Haim, Y., Bucher, D., Cabrera-Hernádez, F. J., Carballo-Franquis, J., Chen, A., Chen, C.-F., Chow, J. M., Córcoles-Gonzales, A. D., Cross, A. J., Cross, A., Cruz-Benito, J., Culver, C., González, S. D. L. P., Torre, E. D. L., Ding, D., Dumitrescu, E., Duran, I., Eendebak, P., Everitt, M., Sertage, I. F., Frisch, A., Fuhrer, A., Gambetta, J., Gago, B. G., Gomez-Mosquera, J., Greenberg, D., Hamamura, I., Havlicek, V., Hellmers, J., Herok, Ł., Horii, H., Hu, S., Imamichi, T., Itoko, T., Javadi-Abhari, A., Kanazawa, N., Karazeev, A., Krsulich, K., Liu, P., Luh, Y., Maeng, Y., Marques, M., Martín-Fernández, F. J., McClure, D. T., McKay, D., Meesala, S., Mezzacapo, A., Moll, N., Rodríguez, D. M., Nannicini, G., Nation, P., Ollitrault, P., O'Riordan, L. J., Paik, H., Pérez, J., Phan, A., Pistoia, M., Prutyanov, V., Reuter, M., Rice, J., Davila, A. R., Rudy, R. H. P., Ryu, M., Sathaye, N., Schnabel, C., Schoute, E., Setia, K., Shi, Y., Silva, A., Siraichi, Y., Sivarajah, S., Smolin, J. A., Soeken, M., Takahashi, H., Tavernelli, I., Taylor, C., Taylour, P., Trabing, K., Treinish, M., Turner, W., Vogt-Lee, D., Vuillot, C., Wildstrom, J. A., Wilson, J., Winston, E., Wood, C., Wood, S., Wörner, S., Akhalwaya, I. Y. and Zoufal, C.: Qiskit: An Open-source Framework for Quantum Computing (2019).

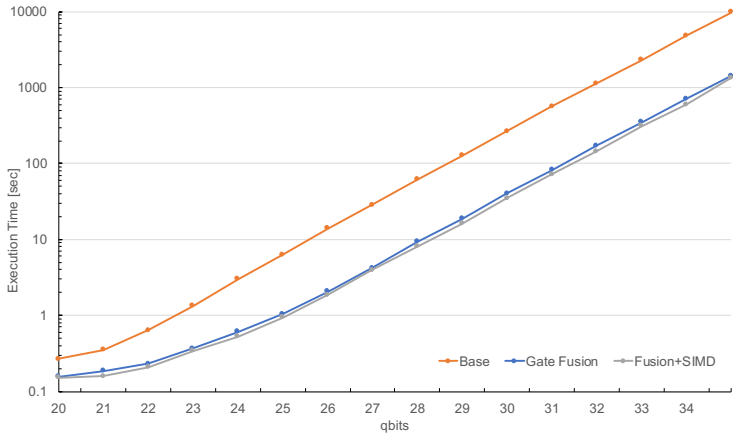

図 6 QV の実行時間

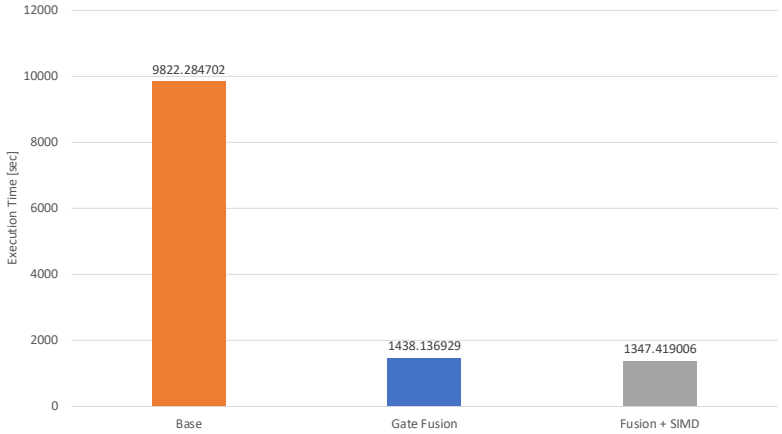

図 7 QV、35qubit の実行時間