Qsimulation V2.0: An Optimized Quantum Simulator[⋆]

Hua Wu[1], Yuxin Deng[1], Ming Xu[1], and Wenjie Du[2]

[1] Shanghai Key Laboratory of Trustworthy Computing,
MOE International Joint Lab of Trustworthy Software,
and International Research Center of Trustworthy Software,
East China Normal University, Shanghai, China
coconuteva@qq.com, yxdeng@sei.ecnu.edu.cn, mxu@cs.ecnu.edu.cn
[2] Shanghai Normal University, Shanghai, China
wenjiedu@shnu.edu.cn

Abstract. Qsimulation is a tool for simulating quantum computation on classical computers, which allows a user to write quantum programs in a simple quantum programming language, draw quantum circuits, and view the results of executing them. Similar to many other quantum simulation tools, the performance of Qsimulation largely depends on its capacity of dealing with matrix operations. In this paper we present Qsimulation V2.0, an optimized quantum simulator that implements a new algorithm for accelerating matrix-vector multiplications. The algorithm is based on matrix decomposition using tensor products and suitable for simulating the execution of quantum circuits. Experimental results show that Qsimulation V2.0 outperforms the open source frameworks Qiskit and ProjectQ.

Keywords: Quantum computation · Quantum simulator · Optimization

1 Introduction

Quantum computation has been a topic of great interest for the last two decades [10]. Benefiting from the superposition of quantum states, quantum computers can accelerate computation remarkably compared with classical computers [7, 9, 12]. While a lot of experimental physicists and computer scientists are currently trying to build scalable quantum computers, e.g. [1], it appears that simulation of quantum computation will be at least as critical as circuit simulation in classical VLSI design [16].

As early as in 1980s Richard Feynman observed that simulating quantum processes on classical hardware seems to require super-polynomial (in the number of qubits) memory and time [16]. With the rapid development of hardware, the simulation of more and more qubits becomes gradually possible. Using quantum

[⋆] Supported by the National Natural Science Foundation of China (61672229, 61832015, 62072176, 1187122), the Inria-CAS joint project Quasar, and the Fundamental Research Funds for the Central Universities.

circuits to simulate quantum computation has become the mainstream. Up to now, a large number of quantum languages are springing up to simulate quantum computing, such as Quipper [6], Q# [15], Qiskit [3], and ProjectQ [14].

State-of-the-art algorithms for simulating quantum computation on classical computers can be mainly divided into two categories: (i) the state-vector approach stores a quantum state as a vector and lets it evolve during quantum computation; (ii) the tensor-based approach represents quantum states as tensors and specifies the input and output states as rank-1 Kronecker projectors [8]. The first approach is limited by the number of qubits due to the exponential growth of the Hilbert space, and the second one is less sensitive to the number of qubits and has been pursued more actively. However, if a quantum state is entangled, it is impossible to decompose the state into the form of a product of states of individual subsystems. Therefore, an entangled state still needs to be represented by a vector. When the number of qubits in an entangled state is very large, the consumption of space and time grows exponentially. For example, in the cases of major interest — Shor's and Grover's algorithms — quantum simulation is still performed with straightforward linear-algebraic tools and requires astronomic resources [16].

Qsimulation [4] is a quantum simulator designed with the first approach. The bottleneck of its performance lies in manipulating large matrices. We observe that a quantum circuit consists of many steps, and in each step the common pattern is to apply some quantum gates in parallel to a number of qubits. Very often, a relatively small number of qubits are involved in the computation of each step while many other qubits remain idle, though the whole circuit could be large in both width and depth. Since the computation in each step can be viewed as the multiplication of a unitary matrix with a state vector, the above observation inspires us to decompose the unitary matrix as a tensor product of smaller matrices representing the parallel gates applied to some qubits together with identity matrices representing the idleness of other qubits. To some extent, we combine the state-vector and the tensor-based approaches. We implement this idea in an optimized algorithm for matrix-vector multiplication, exploit multithreaded programming, and incorporate the new algorithm in Qsimulation V2.0. We have conducted experiments on various quantum algorithms, such as Grover's algorithm [7, 10], Simon's algorithm [13], Bernstein-Vazirani algorithm [2]. It turns out that Qsimulation V2.0 has almost doubled the number of qubits that can be handled by Qsimulation V1.0. It outperforms Qiskit in most cases by taking almost one-third the time, and it is nearly 30 times faster than ProjectQ when the number of qubits is large.

The rest of the paper is organized as follows. In Section 2, we give the necessary notations about quantum bits and gates. A brief introduction to Qsimulation is given in Section 3. In Section 4 we explain the main idea for our new algorithm of computing matrix-vector multiplication. The algorithm itself is presented in Section 5. In Section 6 we use several typical quantum algorithms to show the performance of Qsimulation before and after the optimization, and we compare Qsimulation V2.0 with Qiskit and ProjectQ. In Section 7, we discuss further optimization with multithreaded programming. Finally, we conclude in Section 8.

All the Java code and supplementary materials are available at the link https://github.com/coconutoe/quantum.

2 Notations about Quantum Bits and Gates

Bit is a fundamental concept of classical computation and classical information. Quantum computation and quantum information are built upon an analogous concept, the quantum bit, or qubit for short [10]. For a qubit, there are two possible states, which correspond to the states 0 and 1 for a classical bit. The qubit states are expressed as $|0\rangle$ and $|1\rangle$, in which notation like '$|\ \rangle$' is called the Dirac notation and it is the standard notation for states in quantum mechanics. The difference between bits and qubits is that a qubit can be in a state other than $|0\rangle$ or $|1\rangle$. The state of a qubit can be linear combinations of other states, often called superpositions: $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where α and β are complex numbers and the normalization condition $|\alpha|^2 + |\beta|^2 = 1$ must be satisfied. Put another way, the state of a qubit is a unit vector in a two-dimensional complex vector space. The special states $|0\rangle$ and $|1\rangle$ form an orthonormal basis for this vector space. Multiple qubits can be similarly denoted. For example, we write $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$ for the state of a two-qubit system such that both qubits are in the same states $|0\rangle$ and $|1\rangle$ with equal chance. More detailed account can be found in [10].

Quantum gates are also analogous to gates in classical circuits, in which logic gates are used to process classical bits. The function of a quantum gate in quantum circuits is to convert the states of qubits. For example, quantum gate NOT can change quantum state $|0\rangle$ to $|1\rangle$. In fact, a quantum gate is essentially a unitary matrix. For example, the $Hadamard$ gate is defined as $H \equiv \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$, and the NOT gate mentioned earlier is $X \equiv \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$.

3 A Brief Introduction to Qsimulation

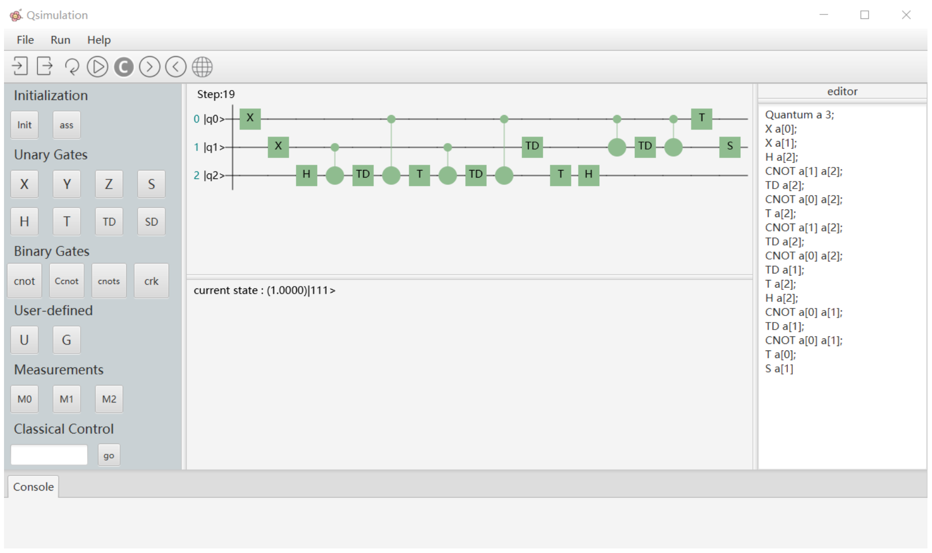
Qsimulation [4] is a lightweight quantum simulator intended to be used by instructors and novices to design and test simple quantum programs and circuits. It contains three main ingredients:

- an imperative language for writing simple quantum programs;
- an interpreter that can translate a quantum program into a quantum circuit;
- an interactive user interface that allows a user to simulate the execution of a quantum program or a circuit.

As a domain-specific high-level language, the imperative language of Qsimulation follows Selinger's slogan of "quantum data, classical control" [11]. Besides assignments, sequential composition, and conditionals, it has quantum initialization, unitary operations, and measurements to deal with quantum data. It also has auxiliary assemblies, including quantum types and a series of reentrant encapsulated functions to simplify programming.

The interpreter accepts a quantum program and translates it into a quantum circuit. Furthermore, the interpreter is also designed as an execution engine. It conforms to the rules of quantum mechanics [10], represents a quantum state as a complex vector and a quantum gate as a complex matrix. In the current work, we propose an algorithm about decomposing matrices into blocks so to accelerate matrix-vector multiplication and an engineering method to optimize the linear algebra calculation inside the simulator.

The graphic user interface facilitates a user to design quantum programs and circuits. In Figure 1, we show the interface with a quantum circuit whose input quantum state is $|110\rangle$. By making use of the buttons in the left part of the panel, we can easily drag and drop built-in gates to the quantum circuit in the middle of the panel. A user can also write a quantum program in the coding editor and then run it. The results are displayed in the lower part of the middle panel. By clicking the icons above the panel, we can perform some general operations such as importing and saving files. The graphic user interface is convenient and intuitive for a user to better understand the circuit model of a quantum algorithm. It is a useful assistant for an instructor to teach simple quantum algorithms. With the help of Qsimulation, students can have a more intuitive understanding of quantum circuits.



current state : (1.0000)|111>

```
Quantum a 3;
X a[0];
X a[1];
H a[2];
CNOT a[1] a[2];
TD a[2];
CNOT a[0] a[2];
T a[2];
CNOT a[1] a[2];
TD a[2];
CNOT a[0] a[2];
TD a[1];
T a[2];
H a[2];
CNOT a[0] a[1];
TD a[1];
CNOT a[0] a[1];
T a[0];
S a[1]
```

Fig. 1: Interface of Qsimulation V2.0

4 Main Idea of the Optimization Algorithm

The evolution process of a quantum state can be described by a quantum circuit. A unitary gate in the circuit corresponds to a unitary matrix, say U, and a quantum state to a unit vector, say $|\psi\rangle$. After applying the unitary matrix to the state, we obtain a new state given by the vector $U|\psi\rangle$. For example, consider a quantum circuit made up of n qubits. The size of the unitary matrix U is $N \times N$, where $N = 2^n$. The size of the unit vector $|\psi\rangle$ is N, so is the size of the resulting vector $U|\psi\rangle$. It is easy to see that the time complexity of the matrix-vector multiplication is $\mathcal{O}(N^2)$, when addition and multiplication are considered as basic operations of cost $\mathcal{O}(1)$. In practice, however, a quantum algorithm consists of a series of quantum gates, most of which are $Pauli$ gates or $CNOT$ gates in typical quantum algorithms such as Deutsch-Jozsa algorithm [5, 10], Grover's algorithm [7, 10], Simon's algorithm [13], and Bernstein-Vazirani algorithm [2]. Fortunately, the dimension of the matrix corresponding to each gate is usually no more than four. That is, most quantum operations may just be applied to a few qubits, resulting in a large number of zero entries in the unitary matrix U. We will decompose matrix U into some blocks, and then perform matrix multiplication for blocks. We aim to reduce the time complexity to nearly $\mathcal{O}(N)$ in the optimized approach of matrix multiplication. The discussion will be carried out in three cases: (1) $U = I \otimes U_e$, (2) $U = U_e \otimes I$, (3) $U = I \otimes U_e \otimes I$, where each I represents an identity matrix and $\otimes$ stands for the tensor product operation of matrices, according to different positions of the quantum operation represented by the matrix U_e. The first two cases are special forms of the last case. We separate them out for the convenience of presentation.

Let us start with a general discussion of the mathematical tools needed in the above three cases, regardless of concrete quantum circuits. In the following discussion, we assume that the matrix U_e has dimension M and is expressed as $U_e = (u_{i,j})_{M\times M}$ with $0 \le i, j \le M-1$. Analogously, the dimension of U is assumed to be N, and the vector $|\psi\rangle$ is $(c_0, c_1, \ldots, c_{N-1})^{\mathrm{T}}$.

4.1 Case $U = I \otimes U_e$

The dimension of I is $L = N/M$ in this case. Then, we have

$$U|\psi\rangle = \begin{pmatrix} U_e & & & \\ & U_e & & \\ & & \ddots & \\ & & & U_e \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{pmatrix} = \begin{pmatrix} U_e|\psi_0\rangle \\ U_e|\psi_1\rangle \\ \vdots \\ U_e|\psi_{L-1}\rangle \end{pmatrix} \tag{1}$$

where $|\psi_i\rangle = \left(c_{M\cdot i}, c_{M\cdot i+1}, \ldots, c_{M\cdot(i+1)-1}\right)^{\mathrm{T}}$ is a segment of $|\psi\rangle$, for $i = 0, \ldots, L-1$. We can see that the multiplication of the $N \times N$ matrix by the N-dimensional vector is decomposed into L multiplications of $M \times M$ matrices by M-dimensional vectors. The overall time complexity becomes $\mathcal{O}(M^2 \cdot L) = \mathcal{O}(M \cdot N)$. In many applications, the dimension of U_e is far smaller than that of U so that M can be regarded as a fairly small constant. Thus the final time complexity is nearly $\mathcal{O}(N)$.

4.2 Case $\boldsymbol{U = U_e \otimes I}$

The dimension of I is also $L = N/M$ in this case. Then, we have

$$U|\psi\rangle = \begin{pmatrix} u_{0,0}I & u_{0,1}I & \cdots & u_{0,M-1}I \\ u_{1,0}I & u_{1,1}I & \cdots & u_{1,M-1}I \\ \vdots & \vdots & \ddots & \vdots \\ u_{M-1,0}I & u_{M-1,1}I & \cdots & u_{M-1,M-1}I \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{pmatrix}. \tag{2}$$

For any $0 \le i, j \le M-1$, the $(i \cdot L + j)$-th element of the resulting $U|\psi\rangle$ is $\sum_{k=0}^{M-1} u_{i,k} \cdot c_{j+k\cdot L}$. As in the first case, we regard M as a fairly small constant. Thus the time complexity of the matrix multiplication becomes $\mathcal{O}(N)$, which is far less than the original $\mathcal{O}(N^2)$.

4.3 Case $\boldsymbol{U = I \otimes U_e \otimes I}$

Let L_l be the dimension of I on the left of U_e, and L_r the dimension of I on the right. Again, the dimension of U_e is M. That is, the dimension of U is $N = L_l \cdot M \cdot L_r$. Then, we have

$$U|\psi\rangle = \begin{pmatrix} U_e \otimes I & & & \\ & U_e \otimes I & & \\ & & \ddots & \\ & & & U_e \otimes I \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{pmatrix} = \begin{pmatrix} (U_e \otimes I)|\psi_0\rangle \\ (U_e \otimes I)|\psi_1\rangle \\ \vdots \\ (U_e \otimes I)|\psi_{L_l-1}\rangle \end{pmatrix} \tag{3}$$

where $|\psi_i\rangle = \left(c_{M\cdot L_r \cdot i}, c_{M\cdot L_r\cdot i+1}, \ldots, c_{M\cdot L_r\cdot(i+1)-1}\right)^{\mathrm{T}}$ is a segment of $|\psi\rangle$, for $i = 0, \ldots, L_l - 1$. Here $(U_l \otimes I)|\psi_i\rangle$ can be obtained through the method in the second case. As in the previous cases, M is usually a small constant. So the time complexity is $\mathcal{O}(N)$ in total, far less than the original $\mathcal{O}(N^2)$.

5 The Optimization Algorithm

We introduce the sub-procedure **UTensorI** given in Algorithm 1 that will be called in the main procedure **ITensorUTensorI** given in Algorithm 2.

The essence of **UTensorI** is actually the decomposition of matrices according to the formula (2) in Section 4.2. Regard the whole matrix I as a block. Then $U = U_e \otimes I$ becomes a matrix of M^2 unit matrices with different coefficients. In order to calculate $U|\psi\rangle$, we divide $|\psi\rangle$ into M blocks, each of which has the same dimension as matrix I and is multiplied by a coefficient. In this way, the calculation of a large complex matrix multiplied by a vector becomes the calculation of many smaller unit matrices multiplied by vectors.

The pseudo code in Algorithm 1 describes the details of the procedure. The outermost for loop traverses the block matrix of $U = U_e \otimes I$ by block lines. The middle for loop traverses each row of the decomposed matrix and records the final result. The innermost for loop accumulates the values of the entries in

Algorithm 1: Procedure **UTensorI**

```
Input: (1) $IDimension$: the dimension of matrix $I$;
(2) $uGate$: the matrix $U_e$;
(3) $qsIn$: a vector representing the quantum state before the evolution.
Output: $qsOut$: a vector representing the quantum state after the evolution.
1 for $i = 0 \to (uGate.dimension - 1)$ do
2     for $j = 0 \to (IDimension - 1)$ do
3         $temp \leftarrow Complex.ZERO$;
4         for $k = 0 \to (uGate.dimension - 1)$ do
5             $temp \leftarrow temp + uGate[i][k] * qsIn[j + k * IDimension]$;
6         end
7         $qsOut[i * IDimension + j] \leftarrow temp$;
8     end
9 end
```

Algorithm 2: Procedure **ITensorUTensorI**

```
Input: (1) $IlDimension$: the dimension of the left matrix $I$;
(2) $IrDimension$: the dimension of the right matrix $I$;
(3) $uGate$: the matrix $U_e$;
(4) $qsIn$: a vector representing the quantum state before the evolution.
Output: $qsOut$: a vector representing the quantum state after the evolution.
1 $i \leftarrow 0$;
2 $m \leftarrow IrDimension * uGate.dimension$;
3 while $i < IlDimension$ do
4     $vectorTemp[0 : m] \leftarrow qsIn[i * m : (i + 1) * m]$;
5     $qsOut[i * m : (i + 1) * m] \leftarrow$ UTensorI($IrDimension, uGate, vectorTemp$);
6     $i \leftarrow (i + 1)$;
7 end
8 Notation: $v_2[i_2 : i_2 + j_1 - i_1] \leftarrow v_1[i_1 : j_1]$ denotes the assignment of the $i_2$-th
  to the $(i_2 + j_1 - i_1)$-th elements of $v_2$ by the $i_1$-th to the $j_1$-th elements of $v_1$.
```

the result vector. We regard $uGate.dimension$ as a small constant, so the time complexity of Algorithm 1 is $\mathcal{O}(IDimension)$.

The main procedure **ITensorUTensorI** is described by the pseudo code in Algorithm 2. Since $U_e \otimes I$ has already been solved by Algorithm 1, we can treat $U_e \otimes I$ as a larger unitary matrix with a calculation cost of $\mathcal{O}(IrDimension)$ without caring about its internal details. The following discussion will only focus on $I \otimes U_e$. Note that $I \otimes U_e$ can be partitioned into diagonally identical matrices U_e, and other elements are all zero as discussed in Section 4.1. Then the calculation is simplified to the form of $IlDimension$ U_e's right multiplied by vectors. The procedure **ITensorUTensorI** has only one layer of while loop, which is used to control the number of times that U_e is multiplied by a vector on the right, and puts the corresponding results into the final position of the result vector. The total time complexity of Algorithm 2 is $\mathcal{O}(IlDimension \cdot IrDimension)$.

6 Experimental Results

We have carried out experiments in Qsimulation on various quantum algorithms such as Deutsch-Jozsa algorithm, Bernstein-Vazirani algorithm, Grover's algorithm, and compared the running time of those algorithms before and after the optimization. As we can see from Table 1, the efficiency in Qsimulation V2.0 has been greatly improved. For example, the running time of Deutsch-Jozsa algorithm (11 qubits) is about 59 times less after the optimization. Since Qsimulation V1.0 can simulate quantum circuits up to 11 qubits in a laptop, our experimental settings are all within 11 qubits. In fact, Qsimulation V2.0 allows us to simulate quantum circuits up to 22 qubits in the same laptop. In addition, we run the same algorithms with two other software tools: Qiskit and ProjectQ, where Qiskit is an open source quantum programming library developed by IBM and ProjectQ was developed by ETH Zurich and has been open-sourced since 2006. When the running time exceeds 30000ms, we force the termination of the programs. It can be seen that Qsimulation V2.0 outperforms Qiskit by taking almost one-third the time, and it performs bettern than ProjectQ in most cases. As a matter of fact, ProjectQ merely performs well when the number of qubits is small. Its performance decreases drastically when the number of qubits increases.

Algorithms	qubits	Time(ms)			
		Qsimulation V1.0	Qsimulation V2.0	Qiskit	ProjectQ
Deutsch-Jozsa [5, 10]	11	8277	138	8811	≥ 30000
Simon [13]	10	635	7.5	15.1	9
Bernstein-Vazirani [2]	10	1148	9	15.5	9.3
Superdense Coding [10]	2	1.25	1.125	5.5	1
Teleportation [10]	3	3.5	1.625	5.875	1.125
Grover [7, 10]	9	5978	172	669	≥ 30000

Table 1: The running time of various algorithms

We then compare Qsimulation V2.0 with Qiskit and ProjectQ on quantum circuits with more qubits. We first prepare an entangled state with 18 qubits: $\frac{1}{\sqrt{2}}(|00\ldots0\rangle + |11\ldots1\rangle)$. More specifically, we apply the *Hadamard* gate H on the first qubit, and then use that qubit to control the *NOT* gate X acted on each of the other 17 qubits. Then in different cases we apply different gates on the 18 qubits. For example, in the case $18H$, the gate H is applied on each qubit; in the case $9H\_9Y$, we apply the H gate on 9 qubits and the *Pauli*-Y gate on 9 other qubits. We record the running time on Qsimulation V2.0, Qiskit and ProjectQ, respectively.

It can be seen from Table 2 that the running time of Qsimulation V2.0 is close to one-third of that of Qiskit in most cases, and the efficiency of ProjectQ is the worst, with the running time being about 10 times of Qiskit.

Gates	$18H$	$18X$	$18Y$	$18Z$	$9H\_9X$	$9H\_9Y$	$9H\_9Z$	$6X\_6Y\_6Z$
Qsimulation V2.0	830	782	824	790	786	831	812	823
Qiskit [3]	2229	2221	2198	2217	2298	2193	2201	2254
ProjectQ [14]	21198	20565	19725	27286	21016	22145	21803	18220

Table 2: Quantum gates acted on entangled states (ms)

7 Further Optimization by Multithreading

The mathematical description of a quantum evolution is essentially a matrix-vector multiplication. In the first stage of optimization, we divided the matrix into blocks for calculation. On this basis, we will introduce multi-threaded parallel computation. The reason why parallel computation is feasible is that calculating each element of the resulting matrix is independent. For example, for any matrices A and B, with the sizes $r \times s$ and $s \times t$, respectively, we can calculate the product of A and B like $C = AB = (c_{ij})$, where $c_{ij} = \sum_{k=1}^{s} a_{ik} \cdot b_{kj}$ $(1 \leq i \leq r,\ 1 \leq j \leq t)$. It can be seen that calculating each c_{ij} is mutually independent. That is, no matter which element is calculated first, the calculation of other elements will not be affected. So we can use multithreading to calculate each element in parallel. Theoretically, the time complexity will decrease from $\mathcal{O}(r \cdot s \cdot t)$ to $\mathcal{O}(s)$. If $r = s$ and $t = 1$, the matrices operation will be reduced to a matrix-vector multiplication, and the time complexity will drop from $\mathcal{O}(s^2)$ to $\mathcal{O}(s)$ by the multithreading optimization.

Since our tool is developed in Java, our multithreaded parallel computation is implemented by the fork/join framework. In the actual experiments, parallel computation with three threads and quantum circuits with 21 qubits are used. When the number of qubits is less than 20, almost no optimization can be observed in that it takes time to start multithreading and the greater the amount of computation, the greater the advantage of multithreading. We just use a simple quantum circuit to verify the idea. We first initialize 21 qubits, and then apply quantum operations on the 21 qubits respectively. We record the running time of single-thread version and 3-thread version in Table 3. It can be seen that the running time of multithreaded programming indeed decreases.

Gates	$21H$	$21X$	$21Y$	$21Z$	$7X\_7Y\_7Z$
Single-th	22778	22499	21997	22179	22573
3-ths	13439	13329	13886	13919	13323

Table 3: The running time of single-thread and multi-threads (ms)

8 Conclusion

We have presented Qsimulation V2.0 that uses a new algorithm to compute matrix-vector multiplication. We have conducted experiments and found that it has almost doubled the number of qubits that can be handled by Qsimulation, and it outperforms Qiskit in most cases by taking almost one-third the time, let alone ProjectQ, which turns out to be 10 times slower than Qiskit when the number of qubits is large. From the engineering point of view, we have used multithreading to further optimize our algorithm and shown its effectiveness by experiments. Note that the space complexity is also reduced, which is reflected from the fact that more qubits can be simulated.

References

1. Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J.C., Barends, R., Biswas, R., et al.: Quantum supremacy using a programmable superconducting processor. Nature **574**, 505–510 (2019)
2. Bernstein, E., Vazirani, U.: Quantum complexity theory. SIAM Journal on Computing **26**(5), 1411–1473 (1997)
3. Cross, A.: The IBM Q experience and QISKit open-source quantum computing software. APS March Meeting 2018 **63**(1) (2018)
4. Deng, X., Deng, Y.: Qsimulation: A tool for simulating quantum computation (in Chinese). Computer Engineering and Science **041**, 843–850 (2019)
5. Deutsch, D., Jozsa, R.: Rapid solution of problems by quantum computation. Proceedings of the Royal Society of London Series A **439**(1907), 553–558 (1992)
6. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: a scalable quantum programming language. ACM SIGPLAN Notices **48**(6), 333–342 (2013)
7. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the 28th Annual ACM Symposium on the Theory of Computing. pp. 212–219 (1996)
8. Guo, C., Liu, Y., Xiong, M., Xue, S., Fu, X., Huang, A., Qiang, X., et al.: General-purpose quantum circuit simulator with projected entangled-pair states and the quantum supremacy frontier. Physical Review Letters **123**(19) (2019)
9. Harrow, A.W., Hassidim, A., Lloyd, S.: Quantum algorithm for solving linear systems of equations. Physics **103**(10) (2008)
10. Nielsen, M., Chuang, I.: Quantum Computation and Quantum Information. Cambridge University Press (2011)
11. Selinger, P.: Towards a quantum programming language. Mathematical Structures in Computer Science **14**(4), 527–586 (2004)
12. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. IEEE Computer Society pp. 124–134 (1994)
13. Simon, D.R.: On the power of quantum computation. SIAM Journal on Computing **26**(5), 1474–1483 (1997)
14. Steiger, D., Häner, T., Troyer, M.: ProjectQ: An open source software framework for quantum computing. Quantum **2** (12 2016)
15. Tolba, A.S., Rashad, M.Z., El-Dosuky, M.A.: Q#, a quantum computation package for the .net platform. ArXiv **abs/1302.5133** (2007)
16. Viamontes, G.F., Rajagopalan, M., Markov, I., Hayes, J.: Gate-level simulation of quantum circuits. In: Proceedings of the ASP-DAC Asia and South Pacific Design Automation Conference. pp. 295–301 (2003)