EasyChair Preprint

№ 4050

HpQC: a New Efficient Quantum Computing Simulator

Haodong Bian, Jianqiang Huang, Runting Dong, Yuluo Guo and Xiaoying Wang

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

August 17, 2020

HpQC: A new efficient quantum computing simulator

Haodong Bian[1][0000−0003−0907−288X], Jianqiang Huang[1,2,*][0000−0002−4454−7919], Runting Dong[1], Yuluo Guo[1], and Xiaoying Wang[1]

[1] Department of Computer Technology and Application, Qinghai University, Xining, China

[2] Department of Computer Science and Technology, Tsinghua University, Beijing, China

[*] corresponding author: Jianqiang Huang (Email: hjqxaly@163.com)

{HPC_BHD, hjqxaly, eleanordrt, qhugyl, Wangxiaofu163}@163.com

Abstract. With the continuous popularization of quantum computing, high-efficiency quantum computing simulators have attracted researchers' attention because the running time and memory overhead of quantum computing is increased exponentially, which means that it is challenging to be simulated on a traditional computer. The current mainstream work solves this problem by using multi-node clusters, and we find that its single-node performance has not been effectively exerted. This paper proposes HpQC (High-performance Quantum Computing), a simulator that can efficiently parallel quantum computing on a single-node multi-core processor. First, HpQC used AVX2 and FMA instruction sets to maximize the advantages of SIMD (Single Instruction Multiple Data) vectorizations; second, it reduced the CPU calculation cycle by using faster and more efficient bit operations; and finally, we designed innovation data structure to utilize spatial locality of cache effectively. Besides, this article selects the state-of-the-art quantum computing simulator, QuEST (the Quantum exact simulation toolkit), as the benchmark for performance evaluation. For the quantum fourier transform, experimental results show that HpQC can achieve an average acceleration of 2.20x (GNU compiler) and 1.91x (Intel compiler), respectively, compared to QuEST. As for the random quantum circuit program, HpQC can achieve an average speedup of 1.74x (GNU compiler) and 1.51x (Intel compiler), respectively, compared to QuEST.

Keywords: HpQC · QuEST · SIMD · Memory access optimization · Quantum fourier transform · Random quantum circuits

1 Introduction

As we all know, quantum computing [1] is a new type of computing model. Due to the characteristics of quantum mechanics, it has higher computing efficiency than traditional computing models. At least in the currently known quantum computing algorithms, such as quantum random walk [2], quantum

fourier transform [3], and other algorithms, its processing speed is much faster than traditional computing. Therefore, quantum computing is usually applied to large-scale and sophisticated data processing and calculation, such as cryptography [4-6], artificial intelligence [7-9], and other fields. With the development of science and technology in various fields, quantum computing with inherent time complexity has been unable to meet the needs of related applications. For this reason, many researchers in related fields try to obtain higher performance through parallel quantum computing.

The realization of quantum computing must depend on the corresponding computing equipment. Admittedly, many scholars have currently used quantum computers to complete some quantum simulations or achieve more efficient optimization[10-13], due to their expensive cost and immature technology, we need to find cheap and practical alternatives to simulate quantum computing. In this regard, many research scholars have developed efficient simulators for simulating quantum computing based on traditional computers, such as the simulation of random quantum circuit implemented on Sunway TaihuLight [14], the realization of efficiently implementing quantum computing simulations by using CUDA programming on GPUs [15], and the implementation of quantum circuits based on FPGA [16].

Due to the exponential growth of quantum computing runtime and memory overhead in traditional computers [17], it is challenging to perform quantum computing simulations on traditional computers. To further improve the performance of the simulator, some research scholars have made corresponding optimizations based on the characteristics of the processor architecture. Simulation may be difficult because of memory limitations, which follows that the simulation of quantum algorithms instead of single gate operations can give the simulator significant performance improvement [18]. It is also possible to use more efficient data compression to obtain the simulation of more qubits [19]. However, the ideal simulator needs to balance memory space and computing time. ProjectQ [20] is a quantum computing emulator developed based on a single-machine multi-core processor. It uses OpenMP, AVX instruction set, and cache blocking to obtain efficient parallelization. On distributed-memory machines, qHiPSTER(the Quantum High Performance Software Testing Environment) [21] improves performance by using multi-threading, vectorization, and cache partitioning on a single machine; it also efficiently performs quantum calculations on multiple machines by dividing the load equally, while reducing communication time as much as possible. Some researchers have also realized automatic code generation and calculation optimization on multi-machine multi-core processors with excellent portability[22].

Good vectorization and cache blocking operation can bring excellent performance to the simulator, and balanced load processing capacity is the key to large-scale parallel computing. At present, QuEST(High-performance Quantum Computing) [23] is the first open-source quantum computing simulation simulator that supports three different architectures of single-machine multi-threading, multi-machine multi-threading, and GPU. Due to its balanced load processing

capability, QuEST has comparable and better computing performance than ProjectQ on single-node multi-core processors, although it does not use SIMD(Single Instruction Multiple Data) vectorization and cache blocking optimization.

As the quantum simulators developed quantum simulators, it is becoming more and more common to use multi-node clusters to solve simulation problems. However, its single-node performance has not been effectively exerted. First, the simulator fails to utilize the advantages of SIMD vectorization fully; second, a large number of redundant calculations lead to reduced performance; third, the traditional data structure can not give full play to the advantages of spatial locality of cache.

This paper proposes HpQC (High-performance Quantum Computing), which is a simulator that can provide efficient parallel quantum computing on a multi-core processor. First, HpQC has better vectorization performance; second, it replaces the time-consuming multiplication and division operations by introducing low-overhead bit operations; third, it uses innovative data structures to maximize the spatial advantages locality of the cache. This paper selects the state-of-the-art QuEST quantum computing simulator as the benchmark for performance evaluation. Experimental results show that HpQC has better performance than QuEST. When executing the program of QFT, HpQC can achieve an average speedup of 2.20x (GNU compiler) and 1.91x (Intel compiler), respectively, compared to QuEST. As for the implementation of random quantum circuits, HpQC can achieve an average speedup of 1.74x (GNU compiler) and 1.51x (Intel compiler), respectively, compared to QuEST.

Specifically, our contributions:

(1) A parallel computing method that efficiently uses AVX2 and FMA instruction sets is designed to optimize its vectorization operation.

(2) An innovative and fast shift operation method is designed to optimize the calculation of qubits.

(3) An creative data structure is proposed so that the advantages of spatial locality of cache can be maximized and the memory access optimization can be achieved.

2 Background

2.1 Basic operations of quantum computing

The basis of quantum computing is qubit. Not only can qubits realize the classic $|0\rangle$ and $|1\rangle$ states, but they can also express the superposition state through linear combination. For an m-qubit, it can have 2^m states, which is also the key to quantum parallelism [12].

Large-scale quantum computing programs are usually composed of many quantum logic gates. CNOT gate, Hadamard gate, Unitary Transformation often appear in many programs, because they all play the essential role in state transition.

CNOT gate CNOT gate can usually control the controlled quantum object and is generally used to entangle the quantum. The conversion results of $|00\rangle$,

$|01\rangle$, $|10\rangle$, $|11\rangle$ by using the CNOT gate are demonstrated in Fig.1 (a). If the control bit is 0, the target bit remains unchanged; otherwise, the control bit is 1 and the target bit is inverted.

Hadamard gate In quantum computing, it is usually possible to perform a Hadamard gate operation on $|0\rangle$ or $|1\rangle$ to make it in a superimposed state. Fig.1 (b) shows the Hadamard gate and demonstrates the conversion process of $|0\rangle$ and $|1\rangle$.

Unitary Transformation Quantum states are not immutable. One quantum state can also evolve into another quantum state through the unitary matrix, and the process of transformation is called unitary transformation. In Fig.1 (c), the results of $|0\rangle$ obtained after Unitary Transformation are demonstrated.

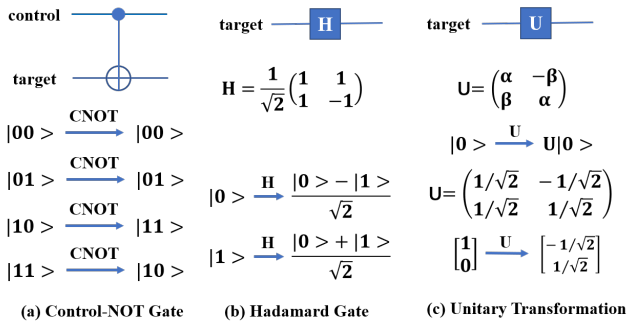

Fig. 1. CNOT Gate, Hadamard Gate and Unitary Transformation.

2.2 Classic quantum computing program

Quantum Fourier Transform The QFT(quantum fourier transform) [24] is the foundation of quantum logic gates, and it is also the core component of various quantum algorithms. Because of its extremely low time complexity [3], it is often used to solve complex problems such as large prime factorization. The calculation process of QFT is periodic, and the program is commonly composed of basic quantum gates such as CNOT and Hadamard, which made it easier to master and understand. Fig.2 shows the QFT operation of 3 qubits.

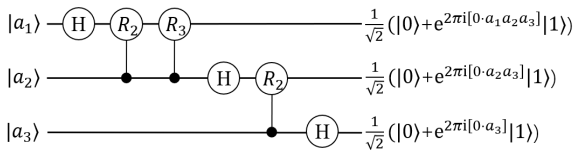

Fig. 2. The QFT operation of 3 qubits.

Random quantum circuits Random quantum circuits are outstanding candidates for proving quantum supremacy [25]. Normally, the simulation of quantum computing is random and irregular, which lead that the user cannot accurately determine the final result of the calculation in advance, so we cannot perform special optimization operations by searching for regularity or other similar methods. It can be seen that many logic gates present in the random quantum circuit must be executed in order. Therefore, it is vital to improving the performance of the single-door operation because this is the key to the overall performance improvement.

3 Optimization method

3.1 SIMD vectorization

SIMD vectorization has always existed in many calculation optimizations, and its existence can make parallel operations better perform on a single calculation of multiple data. For the QuEST quantum computing simulator with load balancing advantages, it still has comparable or even better performance than ProjectQ although without the SIMD optimization. However, a large amount of data in the QuEST for a single calculation operation does not perform well. After analyzing the specific program by using the Perf tool , we found that although the single calculation operation inthe program has been automatically vectorized by the compiler, it still occupied a sizeable overall overhead. To avoid this situation, we need to add the SIMD inline function for vectorization optimization manually in HpQC simulator.

Algorithm 1 QuEST compactUnitary

Input: sizeBlock, sizeHalfBlock, thisBlock, indexUp, indexLo, numTasks, RU, RL, IU, IL, aI, aR, bI, bR, numTasks, stateVecReal[], stateVecImag[]

1: $numTasks \leftarrow (qureg.numAmpsPerChunk >> 1)$
2: $sizeHalfBlock \leftarrow (1 << targetQubit)$
3: $sizeBlock \leftarrow 2 * sizeHalfBlock$
4: **for** $i \leftarrow 0$ to $numTasks$ **in parallel do**
5: $thisBlock \leftarrow i/sizeHalfBlock$
6: $indexUp \leftarrow thisBlock * sizeBlock + i\%sizeHalfBlock$
7: $indexLo \leftarrow indexUp + sizeHalfBlock$
8: $RL \leftarrow stateVecReal[indexUp]$, $IL \leftarrow stateVecImag[indexUp]$
9: $RU \leftarrow stateVecReal[indexLo]$, $IU \leftarrow stateVecImag[indexLo]$
10: $stateVecReal[indexUp] \leftarrow aR * RU - aI * IU - bR * RL - bI * IL$
11: $stateVecImag[indexUp] \leftarrow aR * IU + aI * RU - bR * IL + bI * RL$
12: $stateVecReal[indexLo] \leftarrow bR * RU - bI * IU + aR * RL + aI * IL$
13: $stateVecImag[indexLo] \leftarrow bR * IU + bI * RU + aR * IL - aI * RL$
14: **end for**

To embody the specific method of SIMD optimization in more detail, in Algorithm 1, we give the algorithm of the CompactUnitary function in the QuEST simulator. It is used to describe the evolution of a single qubit from one state to another under the action of the unitary matrix U. As can be seen from Algorithm 1, there is a large amount of data in the CompactUnitary function perform the operations of a single multiplication, addition, and subtraction operation. where SIMD vectorization, as a single-instruction multiple-data stream, is more suitable. Therefore, adding SIMD vectorization operations to these computationally intensive function that has similar operations to the CompactUnitary function can achieve higher calculation effect to get overall performance improvement.

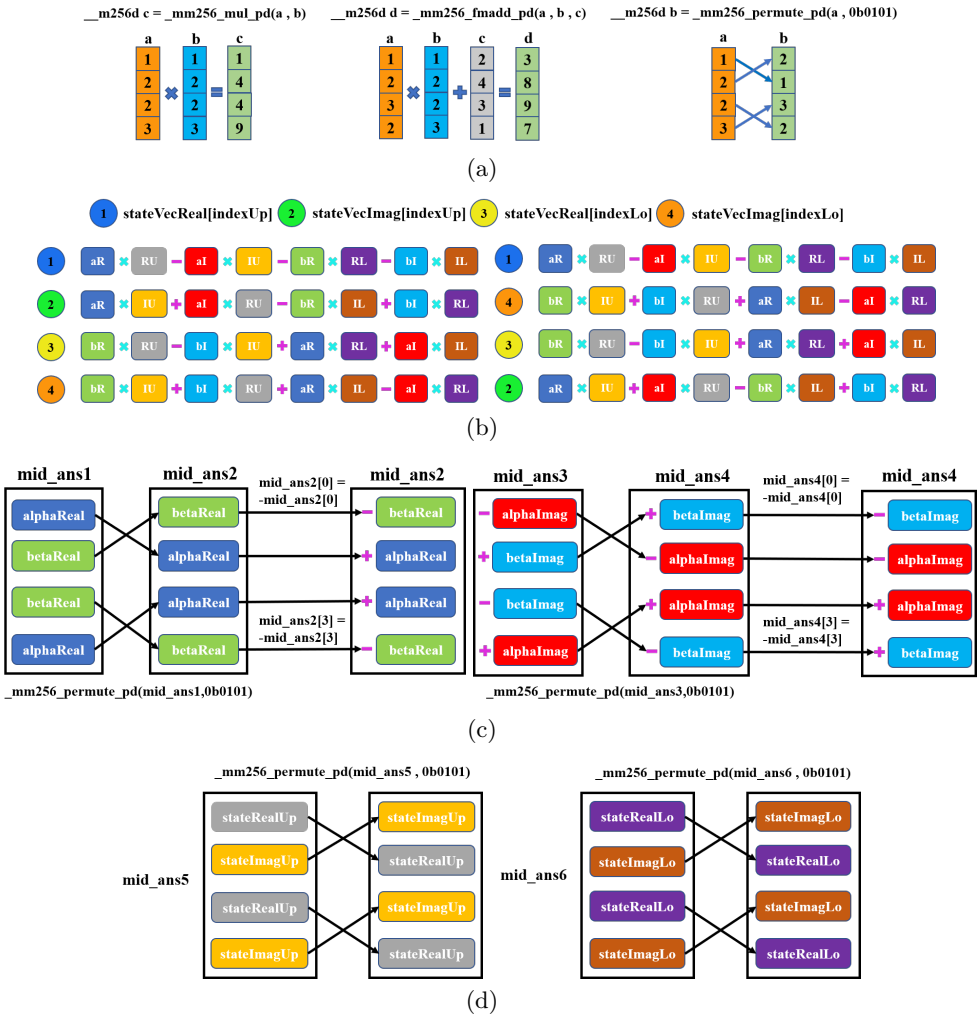

Fig. 3. SIMD vectorization.

In selecting the SIMD instruction set, we used INTEL's AVX2 and FMA instruction sets and gave detailed steps for **CompactUnitary** function optimization. As shown in Fig.3, Fig.3 (a) shows the three inline functions appearing in the vectorization operation, which are used to deal with multiplication, multiplication and addition, and data exchange. The left side of Fig.3 (b) is the calculation operation that needs to be optimized. The right side of Fig.3 (b) demonstrates the swap operation of positions 2 and 4. This operation needs to be performed because it is very time-consuming to update the four variables, such as **stateRealUp** multiple times in the loop, to perform the _mm_set_pd operation. Besides, It is also essential to use data exchange for multiple updated variables and get different operations, just as shown in Fig.3 (d). The values of four constants, such as **alphaReal** in the loop are always fixed, so it is very cost-effective to perform a simple data inversion operation before the start of the loop, and Fig.3 (c) shows the constant data exchange and inversion operations.

3.2 Bit computing optimization

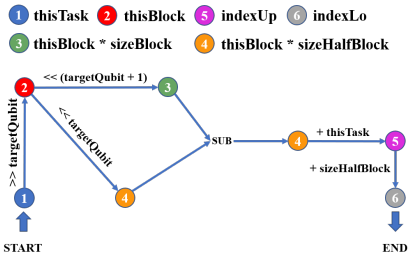

Fig. 4. Bit computing optimization.

QuEST has a very balanced load processing capability and has good scalability. However, After analyzing using the **Perf** tool of the Linux system, we found that most functions will perform **idiv** operation after being compiled into assembly language occupies nearly 50% of the total function overhead. The result of the analysis shows that the time-consuming **idiv** operations are concentrated in lines 5 to 7 of Algorithm 1 (the rest of the functions with the same operations are also applicable). It can be seen in Algorithm 1 that this part of the operation contains division and modular remainder operations. As we all know, performing a division operation in the CPU often requires more than ten clock cycles, and performing a modular remainder operation requires tens or even hundreds of clock cycles, which has a significant impact on the overall performance of the program. In contrast, bitwise operations of the integers in the CPU takes at most one clock cycle. In the HpQC, to better improve the program's performance, it is essential to convert the division and modular remainder operations into bitwise operations as much as possible. Since the compiler does not automatically

complete the conversion of this part of the operation, we need to optimize it manually.

For **sizeHalfBlock** inline 2, it is equal to the result of the number 1 shifted **targetQubit** to the left. Then **thisBlock** is equal to the value of **thisTask** shifted **targetQubit** to the right. For **sizeBlock**, it is equal to the value of number 1 shifted left by **targetQubit + 1** bit. Therefore, multiplying **thisBlock** by **sizeBlock** equals the value of **thisBlock** shifted left by **targetQubit + 1** bit. For **thisTask** modulus residual **sizeHalfBlock** operation. First, divide **thisTask** by **sizeHalfBlock** and record the result as **a**. Then, multiply a by **sizeHalfBlock** and record the result as **b**. Finally, use **thisTask** to subtract **b** and get the final result. The operation of calculating the values of **a** and **b** can be performed by bitwise operations. After the above simplification steps, the **idiv** operation can be successfully eliminated. The specific operation conversion is shown in Fig.4, starting from **START** following the direction of the arrow to **END** to complete the entire calculation process.

3.3 Memory access optimization

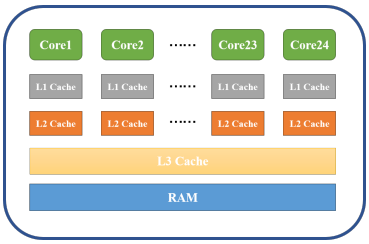

Fig. 5. A architecture of CPU processor with 24 cores.

With the continuous acceleration of CPU computing, slow memory bandwidth has become the main bottleneck limiting the system's overall speed. In CPU architectures, increasing the cache between the CPU and memory to obtain higher data access speed is the key to improving system performance. As shown in Fig.5, this is a CPU processor with 24 cores, each of which has its L1 cache and L2 cache, and all cores share L3 cache and memory. In this CPU architecture, the closer to the CPU, the less storage space the cache has, and the higher the speed of the CPU accessing memory. However, under normal circumstances, most programs cannot improve the system performance due to inadequate cache statuses such as insufficient cache data reuse and frequent cache line replacement. Therefore, solving the problem of the efficient and reasonable use of limited cache space is the key to improving system performance.

Innovative data structure In the QuEST, the state vector is divided into the real part vector and imaginary part vector, and continuous address space is allocated according to specific needs, facilitating the sequential access to data.

However, in some quantum computing operations, a relatively long step-by-step data access operation is usually required, which is very unfriendly for a limited size cache. As shown in the left of Fig.6, for the real part vector and the imaginary part vector, we need to obtain the values at the **indexup** and **indexlo** positions in the real part and the imaginary part vector, and then perform the calculation. Suppose the step between **indexup** and **indexlo** is **X**, and the length of **X** is much larger than the length of the cache line in the current CPU. It can be seen from the left of Fig.6 that the cache line replacement operation needs to be performed four times to obtain the values at four different positions. As we all know, the cache in CPU accesses data in units of the length of a cache line, as for the data exceeding the length of the cache line, it has to perform replacement operations, which is very unfavorable for the performance of the cache.

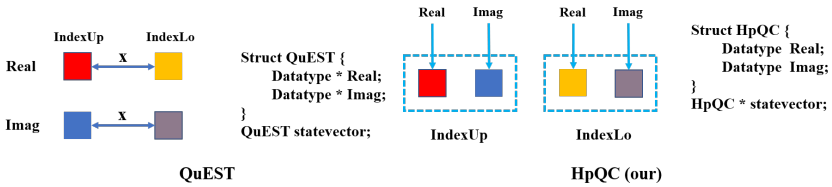

Fig. 6. Data structure of QuEST and HpQC.

To solve this problem, in the developed HpQC, we introduced a simple and efficient data structure. As shown in the right of Fig.6, the real part vector and the imaginary part vector, which were stored in two consecutive stages, are compressed into a structured vector containing the real part and the imaginary part. In this way, initial values stored respectively in the indexup position of the real vector and the imaginary vector can be merged into the indexup position of the new structure vector. Now, the indexup position only needs to be accessed once, and the real and imaginary values can be continuously obtained. Compared with QuEST, the merged data storage structure can better achieve cache data reuse. This is because a cache line access operation can simultaneously obtain the real and imaginary values at the same position, reducing replacements of the cache line.

Store byte alignment In modern CPU architecture, the cache plays a significant role between CPU and memory, which can speedup data transmission and minimize CPU waiting time. When accessing data, the cache is usually carried out with a cache line as the unit length. At present, the size of most cache lines is 64 bytes. Sequentially stored data can better utilize the advantages of cache data reuse when acquiring data in the cache line unit length. However, the situation shown in Fig.7 (a) will sometimes happen when the data that is not byte-aligned is acquired with the cache line. The figure shows that the **Double** data that was originally acquired only once needs to be operated twice without byte alignment. This is because a single cache line fails to obtain the required

data fully, so the cache has to obtain the remaining data through the cache line replacement operation. Fig.7 (b) shows the data alignment operation, and we can find that the data under byte alignment no longer has access across the cache line. This dramatically improves the efficiency of data access.

Algorithm 2 HpQC compactUnitary

Input: sizeHalfBlock, indexUp, indexLo, RU, RL, IU, IL, numTasks, stateVec[], mid_ans1, mid_ans2, mid_ans3, mid_ans4, mid_ans5, mid_ans6, mid_ans0, aI, aR, bI, bR

$mid\_ans1 \leftarrow \_mm256\_set\_pd(aR, bR, bR, aR)$
$mid\_ans2 \leftarrow \_mm256\_permute\_pd(mid\_ans1, 0b0101)$
$mid\_ans2[0] \leftarrow -mid\_ans2[0]$, $mid\_ans2[3] \leftarrow -mid\_ans2[3]$
$mid\_ans3 \leftarrow \_mm256\_set\_pd(aI, -bI, bI, -aI)$
$mid\_ans4 \leftarrow \_mm256\_permute\_pd(mid\_ans3, 0b0101)$
$mid\_ans4[0] \leftarrow -mid\_ans4[0]$, $mid\_ans4[3] \leftarrow -mid\_ans4[3]$
for $i \leftarrow 0$ to $numTasks$ **in parallel do**
  $indexUp \leftarrow ((i >> targetQubit) << targetQubit) + i$
  $indexLo \leftarrow indexUp + sizeHalfBlock$
  $\_mm\_prefetch((char*)\&stateVec[indexLo + 64], \_MM\_HINT\_T1)$
  $\_mm\_prefetch((char*)\&stateVec[indexUp + 64], \_MM\_HINT\_T1)$
  $RL \leftarrow stateVec[indexLo].real$, $IL \leftarrow stateVec[indexLo].imag$
  $RU \leftarrow stateVec[indexUp].real$, $IU \leftarrow stateVec[indexUp].imag$
  $mid\_ans5 \leftarrow \_mm256\_set\_pd(IU, RU, IU, RU)$
  $mid\_ans6 \leftarrow \_mm256\_set\_pd(IL, RL, IL, RL)$
  $mid\_ans0 \leftarrow \_mm256\_mul\_pd(mid\_ans1, mid\_ans5)$
  $mid\_ans0 \leftarrow \_mm256\_fmadd\_pd(mid\_ans3, \_mm256\_permute\_pd(mid\_ans5, 0b0101), mid\_ans0)$
  $mid\_ans0 \leftarrow \_mm256\_fmadd\_pd(mid\_ans2, mid\_ans6, mid\_ans0)$
  $mid\_ans0 \leftarrow \_mm256\_fmadd\_pd(mid\_ans4, \_mm256\_permute\_pd(mid\_ans6, 0b0101), mid\_ans0)$
  $stateVec[indexUp].real \leftarrow mid\_ans0[0]$,$stateVec[indexLo].imag \leftarrow mid\_ans0[1]$
  $stateVec[indexLo].real \leftarrow mid\_ans0[2]$, $stateVec[indexUp].imag \leftarrow mid\_ans0[3]$
end for

Data software prefetch In actual program processing, data transmission is also quite time-consuming. If the program can achieve in advance to the data to be used in the cache in the future, the data transmission will be reduced inevitably. In reality, CPU can prefetch part of the data into the cache at the same time as the calculation to mask the time-consuming data transmission. However, this requires the program itself to have high local accessibility. By modifying the state vector structure, our program is more suitable for using data prefetching to get better performance. The innovative structure puts the elements that need to be calculated in adjacent positions, which is very friendly to cache with the advantages of spatial locality.

In the HpQC, although the hardware prefetching method cannot achieve the expected effect, we use the software prefetching method for data prefetching. In terms of technology, we uses the **_mm_prefetch** function of the **SSE** instruction set to prefetch the data. According to the experimental results, we found that prefetching the data into the cache other than the L1 cache has a better effect.

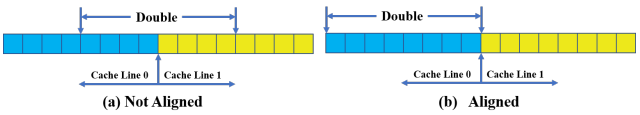

Fig. 7. Data alignment operation.

To further reflect the optimized program details, we have given an optimization algorithm, as shown in Algorithm 2.

4 Performance evaluation

4.1 Device Information

The experimental platform was intel's fourth-generation Haswell architecture CPU. The frequency of this CPU is 2.30GHz, 24 cores, and the L1 cache size is 32KB, the L2 cache size is 256KB, the L3 cache size is 30720KB, DRAM memory size is 128GB, Single max GFlops is 1766.4, Double max GFlops is 883.2. As for the SIMD instruction set, we selected the **AVX2** instruction set and **FMA** instruction set supported by the CPU. Second, we chose two different compilers, **Intel** and **GNU**, as the source code compilation environment, because the QuEST simulator also supports these two compilers. This can reflect the optimization effect of the HpQC simulator in different compilation environments in many aspects. Finally, the fast and efficient **OpenMP** parallel language is used for multi-thread implementation.

4.2 Test program selection

QFT is usually used as a benchmark program for the optimization of high-performance quantum computing. This is because it not only has an excellent structural law but also has a reasonable calculation regularity. Quantum random circuit simulation is usually more random and has no rules to find, so it is more suitable for testing the universality of a quantum computing simulator. In the experiment, the benchmark test programs consist of a QFT of 21 to 30 qubits and a random quantum circuit of 30 qubits. Besides, the experiment thoroughly considered the versatility of the test program. In the simulator's optimization process, we did not make any particular optimization for any specific program, so the optimization operation is more versatile in quantum computing.

4.3 Performance comparison and analysis

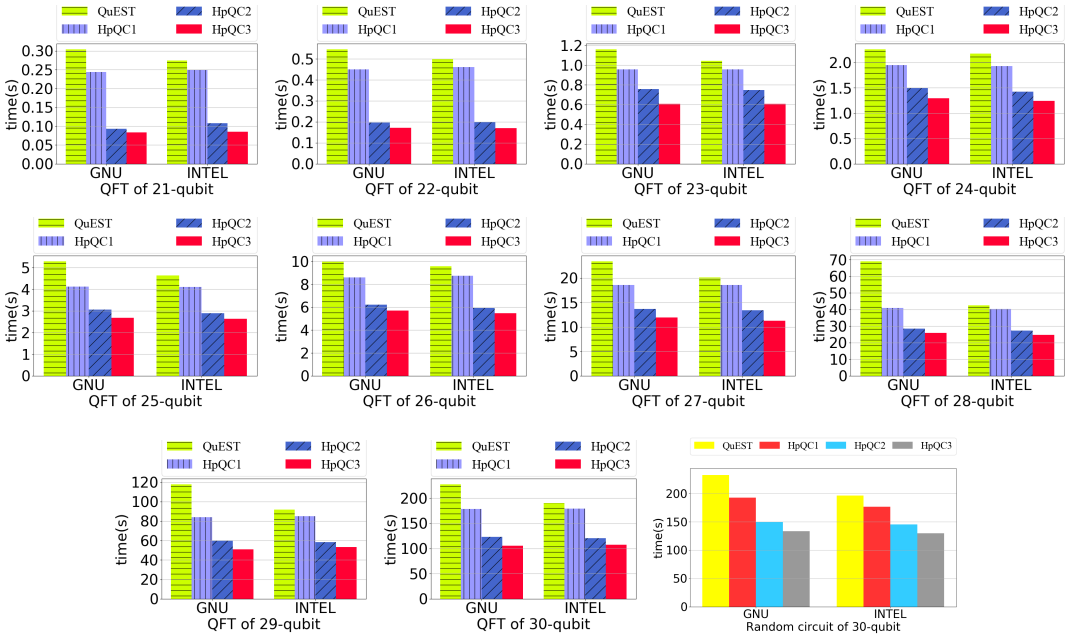

Fig. 8. QFT performance of 21 to 30 qubits and random circuit performance of 30 qubits.

As shown in Fig.8, we demonstrate the QFT performance of 21 to 30 qubits and compares the performance in four different states. In Fig.8, HpQC1 represents HpQC that only realizes SIMD vectorization; HpQC2 represents HpQC that realizes SIMD vectorization and calculation optimization; HpQC3 represents HpQC that implement all optimization methods.

In the same compilation environment, we can find that HpQC1 has a lower time overhead than QuEST. This is because sufficient vectorization can execute SIMD faster to achieve higher performance. Secondly, HpQC2, which achieves calculation optimization, has a significant performance improvement, because compared to the time-consuming multiplication and division operations, the use of fast and efficient bitwise operations can significantly reduce the CPU calculation time. Finally, HpQC3, which implements memory access optimization, has a higher calculation speed than HpQC2, because the innovative data structure can better utilize the spatial locality of the cache. For different compilation environments, the performance effects of HpQC vary since different compilers automatically optimize the code in different ways. What is exciting is that both of them have the same performance improvement trend. We can also find that the QFT performance of 21 and 22 qubits has excellent performance because

its space overhead approximates the size of the cache space, and smaller memory consumption is more suitable for taking advantage of spatial locality of the cache.

As shown in Fig.8 lower right corner, we demonstrated a 30-qubit random quantum circuit by using the GNU and INTEL compilers. First of all, HpQC1 has a pronounced speed increase compared to QuEST, which shows that SIMD vectorization plays a better role in stochastic quantum circuit optimization. Secondly, HpQC2 has a higher speedup than HpQC1, which means that computational optimization is also applicable to random quantum circuits. Finally, the performance brought by memory access optimization is also very significant, which means that the innovative data structure plays a better effect on random circuits.

Table 1. Speedup of HpQC compared with QuEST.

	SpeedUp(GNU)	SpeedUp(INTEL)
QFT-21	3.64x	3.22x
QFT-22	3.18x	2.93x
QFT-23	1.91x	1.71x
QFT-24	1.75x	1.75x
QFT-25	1.97x	1.77x
QFT-26	1.76x	1.75x
QFT-27	1.96x	1.78x
QFT-28	2.67x	1.73x
QFT-29	2.32x	1.72x
QFT-30	2.16x	1.78x
QFT Harmonic mean	2.20x	1.91x
Random-30	1.74x	1.51x

Note: SpeedUp = (QuEST runing times) / (HpQC runing times)

As shown in Table 1, the QFT of 21 and 22 qubits with different compilers have achieved performance improvements that approach or even exceed 3x acceleration ratios. Besides, QFT has achieved performance improvements of 2.20x and 1.91x harmonic average speedup with the GNU and INTEL compilers, respectively. With the INTEL compiler, we can also see that QFTs of more than 22 qubits are less affected by the number of qubits and have a stable performance improvement, which means that our optimization will not cause performance degradation with the increase of qubits. In a 30-qubit random quantum circuit, HpQC achieves performance improvements of 1.74x and 1.51x acceleration ratios with GNU and INTEL compilers compared to QuEST, respectively. This shows that the HpQC simulator is suitable for the simulation of random quantum circuits and maintains the best state.

5 Conclusion

This paper presents an HpQC simulator. On a single node, the methods of SIMD vectorization, calculation, and memory access optimization are used to achieve a more efficient performance. Moreover, it has more stable performance and will not cause performance loss with the increase of qubits. Besides, we also found that qubit operations close to the cache storage space can achieve higher performance. In the future, we will continue to conduct multi-node tests further to verify the scalability of HpQC.

Acknowledgment

This paper is partially supported by the National Natural Science Foundation of China (No.61762074, No.61962051), National Natural Science Foundation of Qinghai Province (No.2019-ZJ-7034). "Qinghai Province High-end Innovative Thousand Talents Program - Leading Talents" Project Support. The Open Project of State Key Laboratory of Plateau Ecology and Agriculture, Qinghai University (2020-ZZ-03).

References

1. Bennett, C. H., Divincenzo, D. P. (2000). Quantum information and computation. Nature, 404(6775), 247-255.
2. Douglas, B. L., Wang, J. (2009). Efficient quantum circuit implementation of quantum walks. Physical Review A, 79(5).
3. Cleve, R., Watrous, J. (2000). Fast parallel circuits for the quantum Fourier transform. foundations of computer science.
4. Nejatollahi, H., Dutt, N., Ray, S., Regazzoni, F., Banerjee, I., Cammarota, R. (2019). Post-Quantum Lattice-Based Cryptography Implementations: A Survey. ACM Computing Surveys, 51(6).
5. Bruss, D., Erdelyi, G., Meyer, T., Riege, T., Rothe, J. (2007). Quantum cryptography: A survey. ACM Computing Surveys, 39(2).
6. Elliott, C., Pearson, D., Troxel, G. D. (2003). Quantum cryptography in practice. acm special interest group on data communication.
7. Pudenz, K., Lidar, D. A. (2013). Quantum adiabatic machine learning. Quantum Information Processing, 12(5), 2027-2070.
8. Biamonte, J., Wittek, P., Pancotti, N., Rebentrost, P., Wiebe, N., Lloyd, S. (2017). Quantum Machine Learning. Nature, 549(7671), 195-202.
9. Ying, M. (2010). Quantum computation, quantum theory and AI. Artificial Intelligence, 174(2), 162-176.
10. Abdullah Ash-Saki, Mahabubul Alam, and Swaroop Ghosh. 2019. QURE: Qubit Re-allocation in Noisy Intermediate-Scale Quantum Computers. In Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19). Association for Computing Machinery, New York, NY, USA, Article 141, 1–6.
11. Prakash Murali, Norbert Matthias Linke, Margaret Martonosi, Ali Javadi Abhari, Nhung Hong Nguyen, and Cinthia Huerta Alderete. 2019. Full-stack, real-system quantum computer studies: architectural comparisons and design insights. In Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19). Association for Computing Machinery, New York, NY, USA, 527–540.

12. Ji Liu, Gregory T. Byrd, and Huiyang Zhou. 2020. Quantum Circuits for Dynamic Runtime Assertions in Quantum Computation. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20). Association for Computing Machinery, New York, NY, USA, 1017–1030.
13. Poulami Das, Swamit S. Tannu, Prashant J. Nair, and Moinuddin Qureshi. 2019. A Case for Multi-Programming Quantum Computers. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52). Association for Computing Machinery, New York, NY, USA, 291–303.
14. Li, R., Wu, B., Ying, M., Sun, X., Yang, G. (2020). Quantum Supremacy Circuit Simulation on Sunway TaihuLight. IEEE Transactions on Parallel and Distributed Systems, 31(4), 805-816.
15. Gutierrez, E., Romero, S., Trenas, M. A., Zapata, E. L. (2010). Quantum computer simulation using the CUDA programming model. Computer Physics Communications, 181(2), 283-300.
16. Mahdi Aminian, Mehdi Saeedi, Morteza Saheb Zamani, and Mehdi Sedighi. 2008. FPGA-Based Circuit Model Emulation of Quantum Algorithms. In Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI (ISVLSI '08). IEEE Computer Society, USA, 399–404.
17. Feynman, R. P. (1999). Simulating physics with computers. International Journal of Theoretical Physics, 21(6), 133-153.
18. Thomas Häner, Damian S. Steiger, Mikhail Smelyanskiy, and Matthias Troyer. 2016. High performance emulation of quantum circuits. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16). IEEE Press, Article 74, 1–9.
19. Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T. Chong. 2019. Full-state quantum circuit simulation by using data compression. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19). Association for Computing Machinery, New York, NY, USA, Article 80, 1–24.
20. Steiger, Damian S., Thomas Häner, and Matthias Troyer. "ProjectQ: An Open Source Software Framework for Quantum Computing." Quantum 2 (2018): 49. Crossref. Web.
21. Smelyanskiy, M., Sawaya, N. P., Aspuruguzik, A. (2016). qHiPSTER: The Quantum High Performance Software Testing Environment.. arXiv: Quantum Physics,.
22. Thomas Häner and Damian S. Steiger. 2017. 0.5 petabyte simulation of a 45-qubit quantum circuit. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17). Association for Computing Machinery, New York, NY, USA, Article 33, 1–10.
23. Jones, T., Brown, A., Bush, I., Benjamin, S. C. (2019). QuEST and High Performance Simulation of Quantum Computers. Scientific Reports, 9(1).
24. Weinstein, Y. S., Pravia, M. A., Fortunato, E. M., Lloyd, S., Cory, D. G. (2001). Implementation of the Quantum Fourier Transform. Physical Review Letters, 86(9), 1889-1891.
25. Guo, C., Liu, Y., Xiong, M., Xue, S., Fu, X., Huang, A., ... Wu, J. (2019). General-Purpose Quantum Circuit Simulator with Projected Entangled-Pair States and the Quantum Supremacy Frontier. Physical Review Letters, 123(19).