基于 IQS 的编译和通信优化

(2021 年 3 月 14 日)

摘要：在经典计算机上模拟量子系统结合了量子计算以及经典优化程序的优势，将继续在量子信息科学中发挥重要作用。在这个工作中，我们基于 IQS，尝试由给定的量子线路构建有向无环图，利用 PermuteQubit(σ) 优化有向无环图上的算法，进而缩短量子算法的模拟时间。进一步地，我们考虑更深层级的优化，如不同量子门完成时间的差异、最优交换策略带来的性能增益与 qubit 数目的关系等，并尝试实现。我们将在简单的量子门以及更复杂（如 Shor 算法、VQE 等）的算法上测试运行时间。我们期望测试结果表明该工作显著缩短了量子算法的模拟时间。

关键词： IQS; 量子模拟器; 分布式; MPI; DAG

1 研究动机

量子计算是通过量子信息单元进行计算的一种新型计算方式，它在某些特定情形下应用精心设计的量子算法可以达到指数级的加速效果。由于量子计算机硬件的发展还处在起步阶段，在传统计算机上模拟量子计算对量子算法及其应用设计、优化、验证和性能评估仍然有着举足轻重的地位。实际上，目前任何应用于量子计算的软件框架都提供了可以在量子模拟器上运行的算法。在此背景下，如何减少量子算法的模拟时间就成为一个关键的问题。

2 Intel Quantum Simulator 介绍

Intel Quantum Simulator (IQS，又称作 qHiPSTER[1]) 是一个在经典计算机上运行的高性能的、分布式的量子模拟器。软件的高性能计算 (HPC) 能力允许用户可以利用超级计算机提供的可用硬件资源以及可用的硬件资源公共云计算基础设施。要利用后一种平台，与之相结合分布式模拟每个独立的量子态，IQS 允许细分计算并行模拟相关电路池的资源。IQS 在 HPC 基础设施获得的基准测试多达 42 量子位。

英特尔量子模拟器，无论是在其初始版本和最新版本，都利用了 HPC 系统，由共享和分布式内存实现。第一种情况是当几个处理器，或具有多个计算核心的处理器，可以访问相同的内存和操作需要在没有特定顺序的情况下执行。可以很好地利用这种并行性 OpenMP。当相对较少的内存需要大量内存时，或者在存储量子态的情况下，一个机器无法容纳大量的内存或节点，就会出现并行性的第二个方式在这种情况下，需要显式地考虑不同流程之间的通信模式采用消息传递接口。在新版本的 IQS 中，它使用 OpenMP+MPI 来作为进程间通信的工具。

IQS 的开源版本 (`https://github.com/iqusoft/intel-qs`) 已经发布到 GitHub，使用 C++ 作为宿主语言。

n 个 qubit 的状态对应 2^n 维的 Hilbert 空间中的矢量，IQS 使用 2^n 个双精度复数记录 n 个 qubit 的全振幅，从而模拟量子态在各种环境下的演化。储存量子态的全振幅对内存开销很大，对于 30 个 qubit 就需要约 17GB 的内存. 为了实现快速模拟量子线路，需要对量子态进行划分并分别分配内存。

2.1 local qubit and global qubit

IQS 假设进程数 P 是 2 的幂：$P = 2^p$，每个进程储存 $2^{n-p} = 2^m$ 个复数。如果 P 不是 2 的幂，取 $p = \lfloor \log_2 P \rfloor$。IQS 默认前 m 个 qubits 为 local qubit，存储在本地内存中, 只涉及 local qubit 的操作不需要线程间的通信；后 $n-m$ 个为 global qubits，分布式地存储在其余节点上，节点间使用 MPI 通信。

考虑作用在第 q 个比特上的单比特门 U（$0 \le q < n$）

$$U = \begin{pmatrix} U_{00} & U_{01} \\ U_{10} & U_{11} \end{pmatrix} \tag{1}$$

U 的作用为

$$\begin{aligned} \alpha'_{\ldots 0_q \ldots} &= U_{00}\alpha_{\ldots *0_q \ldots} + U_{01}\alpha_{\ldots 1_q \ldots} \\ \alpha'_{\ldots 1_q \ldots} &= U_{10}\alpha_{\ldots 0_q \ldots} + U_{11}\alpha_{\ldots 1_q \ldots} \end{aligned} \tag{2}$$

如果 $q < m$，那么 U 门作用的元素均储存在同一个线程中，没有通信代价。如果 $q \ge m$，那么元素分别储存在不同的线程，需要线程间 MPI 通信。

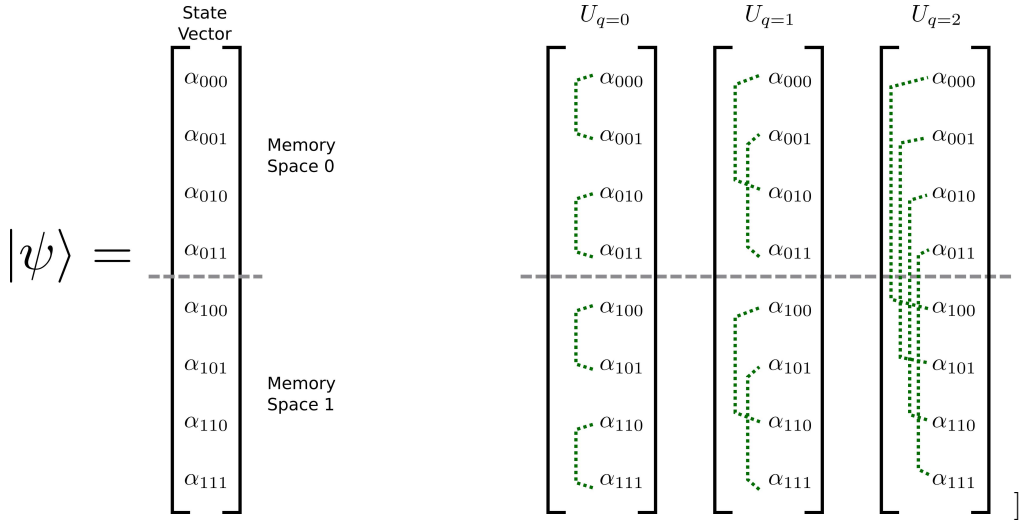

图 1: 3 qubit，2 线程的情形示意图。左：态矢量的八个复数，{000,001,010,011} 这四个元素储存在第一个线程中，其余元素储存在第二个线程中。右：单比特门的实现方案示意图，虚线连线对应于单比特门涉及的一对元素。

我们假设有 $n = 3$ 个 qubit，有 2 个线程，$p = 1$，$m = 2$，如图 1。对前两个 qubit 的单比特门没有通信代价，如 $q = 0$，{000,001},{010,011},{100,101},{110,111} 这四组元素，每组元素均在同一个线程中。而对于 $q = 2$，{000,100},{001,101},{010,110},{011,111} 均不在同一个线程。

目前已经有相关工作 [3] 在优化 local qubits 和 global qubits 的分配，通过重排 qubits 的顺序，以降低 MPI 通信的开销，减少模拟所需时间。我们的算法如下:

Algorithm1 Compiler pass specialized for IQS simulation

The quantum circuit is provided as a DAG describing the logical dependency of the gates. The initial qubit order corresponds to the identity permutation: $\forall q,\ \sigma(q) = q$. The compiled circuit C is represented by a sequential list of instructions (either gates or calls to `PermuteQubits`).

```
G ← DAG of circuit
σ ←identity permutation
C ← {}
while G.vertices≠ ∅ do
    for v in G.vertices do
        if (v has no incoming edges) ∧ (v acts on local qubits according to σ) then
            add ApplyGate(v) to C
            remove v from G
        end if
    end for
    σ̄ ←identity permutation
    ā ← 0
    for l in local qubits according to σ do
        for g in global qubits according to σ do
            σ′ ← σ ∘ (l g)
            a′ ← count G.vertices on local qubits according to σ′
            if a′ > ā then
                σ̄ ← σ′
                ā ← a′
            end if
        end for
    end for
    if ā > 0 then
        σ ← σ̄
        add PermuteQubits(σ) to C
    else
        v ← vertex of G without incoming edges
        add v to C
        remove v from G
    end if
end while
```

图 2: Pseudo code for our algorithm.

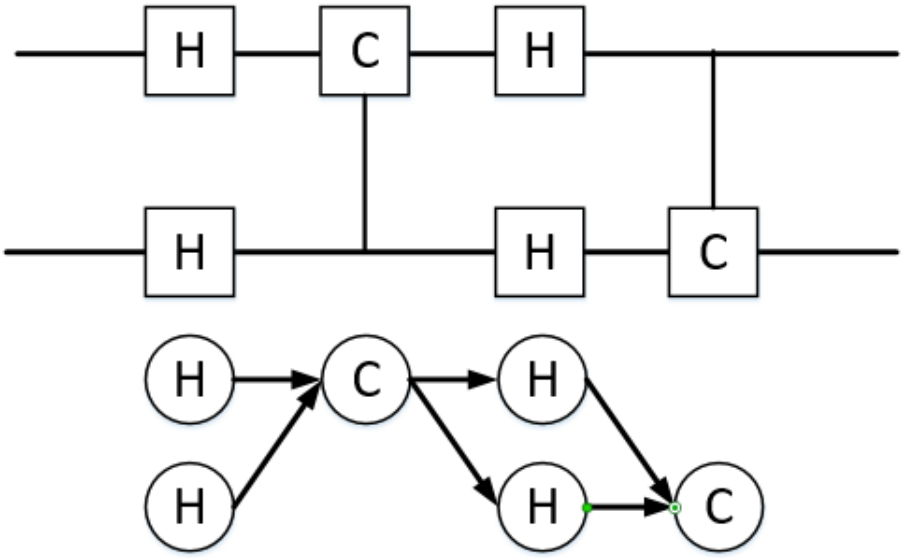

图 3: An example of the DAG for a quantum circuit.

该工作将一个给定的量子线路抽象为有向无环图的形式，图的每个顶点对应一个量子门，每条有向边描述了一个逻辑上的依赖关系，一个具体的例子如下图所示，一个量子门需要进行 MPI 通信时当且仅当至少有一个输入的 qubit 为 global qubit 时。接着引入 PermuteQubits(σ) 的操作完成 qubits 的交换并更新 IQS 中的量子态表示。新增加一个 compiler pass，通过有向无环图上的算法进行优化：如果交换一个 local qubit 和一个 global qubit 之后至少能够降低一个量子门上的通信，那么就执行相应的 PermuteQubits(σ) 指令。算法枚举 local qubit 和 global qubit 的组合寻找最优的交换策略。

2.2 部分代码

我们将量子线路转化为有向无环图的数据结构加入到了 IQS 的代码中，下面是引入图结构的主要的声明函数以及类的代码。

```
1
2  #ifndef DAG_HPP
3  #define DAG_HPP
4
5  #include <vector>
6  #include <set>
7  #include <map>
8  #include <algorithm>
9
10 enum class GateType
```

```
11 {
12     Hadamard,
13     PauliX,
14     PauliY,
15     PauliZ,
16 };
17
18 typedef struct DAGVertex
19 {
20     int id;
21     GateType type;
22     std::vector<unsigned> qubits;
23     bool is_removed;
24 } DAGVertex;
25
26 void hello();
27
28 class DAGCircuit
29 {
30 private:
31     int v_num;
32     int num_qubits;
33     std::map<int, std::set<int>> v2inedges;
34     std::map<int, std::set<int>> v2outedges;
35
36 public:
37     std::vector<DAGVertex> vertices;
38
39     // Constructors
40     DAGCircuit(int _num_qubits);
41
42     template <typename... T>
43     void AddVertex(GateType type, T... args)
44     {
45         std::vector<unsigned> qubits{std::forward<T>(args)...};
46         return AddVertex(type, qubits);
47     };
48     void AddVertex(GateType type, std::vector<unsigned> qubits);
49
```

```
    // remove a vertex and all its outcoming edges from the graph
    void RemoveVertex(DAGVertex &v);

    int getQubitNum();
    // num of vertices in current graph
    int getVertexNum();
    // calculate how many incoming edges does vertex v have currently
    int getInedgeNum(DAGVertex &v);

    // print graph for debug
    void printGraph();
};
#endif
```

而后我们使用基于基于 permutation 的优化操作来减少量子通信门的通信开销，核心代码如下:

```
void scheduler(DAGCircuit &G, QubitRegister<> &psi)
{
    // num of local qubits.
    std::size_t m = G.getQubitNum() - qhipster::ilog2(qhipster::mpi
    ::Environment::GetStateSize());
    int myrank = qhipster::mpi::Environment::GetStateRank();
    if (myrank == 0)
        std::cout << m << " local qubits, " << G.getQubitNum() - m
        << " global qubits. \n";
    std::vector<std::size_t> sigma(G.getQubitNum());
    identity_permutation(sigma);

    while (G.getVertexNum() > 0)
    {
        for (auto &v : G.vertices)
        {
            if (!v.is_removed && G.getInedgeNum(v) == 0
            /* v has no incoming edges */ &&
                get_local_qubits_num(sigma, v.qubits, m) == v.qubits.size()
                /* v acts on local qubits */)
            {
                psi.ApplyGate(v);
                G.RemoveVertex(v);
            }
        }
```

```
26
27          int _a = 0;
28          int da = 0;
29          std::vector<std::size_t> _sigma(G.getQubitNum());
30          std::vector<std::size_t> dsigma(G.getQubitNum());
31          identity_permutation(_sigma);
32
33          // try all m(n-m) possible permutations
34          for (int l = 0; l < G.getQubitNum(); l++)
35          {
36              if (is_local_qubit(sigma, l, m))
37              {
38                  for (int g = 0; g < G.getQubitNum(); g++)
39                  {
40                      if (!is_local_qubit(sigma, g, m))
41                      {
42                          // std::cout << l << " " << g << "\n";
43                          //  ' =  ∘(lg)
44                          for (int i = 0; i < G.getQubitNum(); i++)
45                          {
46                              if (sigma[i] == l)
47                              {
48                                  dsigma[i] = g;
49                              }
50                              else if (sigma[i] == g)
51                              {
52                                  dsigma[i] = l;
53                              }
54                              else
55                              {
56                                  dsigma[i] = sigma[i];
57                              }
58                          }
59                          // print_permutation(dsigma);
60                          da = 0;
61                          for (auto &v : G.vertices)
62                          {
63                              if (!v.is_removed && get_local_qubits_num
64                              (dsigma, v.qubits, m) == v.qubits.size())
```

```
65                              {
66                                  da++;
67                              }
68                          }
69                          // std::cout << "da: " << da << "\n";
70                          if (da > _a)
71                          {
72                              _sigma = dsigma;
73                              _a = da;
74                          }
75                      }
76                  }
77              }
78          }
79          if (_a > 0)
80          {
81              sigma = _sigma;
82              // if (myrank == 0)
83              //     print_permutation(sigma);
84              psi.PermuteQubits(sigma, "direct");
85          }
86          else
87          {
88              // it seems we can not remove communication from any gate
89              //by permutating qubits,
90              // so we just apply gate directly.
91              for (auto &v : G.vertices)
92              {
93                  if (!v.is_removed && G.getInedgeNum(v) == 0
94                  /* v has no incoming edges */)
95                  {
96                      psi.ApplyGate(v);
97                      G.RemoveVertex(v);
98                  }
99              }
100         }
101     }
102     if (myrank == 0)
103         print_permutation(sigma);
```

```
SQG(15)::           tot    1.7506 ncalls 4 ( 63.3%) s gflops    0.36 sn(   0.00 s   0.00 GB/s) dn(
 0.64 s   0.06 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(16)::           tot    1.7540 ncalls 4 ( 78.9%) s gflops    0.36 sn(   0.00 s   0.00 GB/s) dn(
 0.37 s   0.66 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(17)::           tot    3.2843 ncalls 4 ( 36.6%) s gflops    0.19 sn(   0.00 s   0.00 GB/s) dn(
 2.08 s   0.02 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(18)::           tot    2.1490 ncalls 4 ( 45.9%) s gflops    0.30 sn(   0.00 s   0.00 GB/s) dn(
 1.16 s   0.03 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(19)::           tot    1.6774 ncalls 4 ( 55.6%) s gflops    0.38 sn(   0.00 s   0.00 GB/s) dn(
 0.74 s   0.06 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(20)::           tot    2.0072 ncalls 4 ( 72.2%) s gflops    0.32 sn(   0.28 s   0.13 GB/s) dn(
 0.00 s   0.00 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.28 s   0.24 GB/s)
SQG(21)::           tot    1.6849 ncalls 4 ( 40.0%) s gflops    0.38 sn(   0.18 s   0.21 GB/s) dn(
 0.00 s   0.00 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.83 s   1.18 GB/s)
SQG(22)::           tot    1.7556 ncalls 4 ( 47.7%) s gflops    0.36 sn(   0.20 s   0.24 GB/s) dn(
 0.00 s   0.00 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.72 s   2.32 GB/s)
SQG(23)::           tot    1.7739 ncalls 4 ( 54.9%) s gflops    0.36 sn(   0.20 s   0.18 GB/s) dn(
 0.00 s   0.00 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.60 s   0.09 GB/s)
SQG(24)::           tot    1.6925 ncalls 4 ( 37.2%) s gflops    0.38 sn(   0.11 s   0.41 GB/s) dn(
 0.00 s   0.00 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.95 s   0.07 GB/s)
The quantum circuit is composed of 40 one-qubit gates and 0 two-qubitgates, for a total of 40 gates.
The greedy depth (all gates lasting one clock cycle) is 4.
SQG(0)::            tot    1.7640 ncalls 4 ( 65.3%) s gflops    0.36 sn(   0.00 s   0.00 GB/s) dn(
 0.61 s   0.07 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(1)::            tot    1.8441 ncalls 4 ( 69.4%) s gflops    0.35 sn(   0.00 s   0.00 GB/s) dn(
 0.56 s   0.08 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(15)::           tot    1.6304 ncalls 4 ( 49.5%) s gflops    0.39 sn(   0.00 s   0.00 GB/s) dn(
 0.82 s   0.05 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(16)::           tot    1.7759 ncalls 4 ( 61.9%) s gflops    0.36 sn(   0.00 s   0.00 GB/s) dn(
 0.68 s   0.06 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(17)::           tot    3.2798 ncalls 4 ( 48.0%) s gflops    0.19 sn(   0.00 s   0.00 GB/s) dn(
 1.71 s   0.46 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(18)::           tot    2.4039 ncalls 4 ( 53.4%) s gflops    0.27 sn(   0.00 s   0.00 GB/s) dn(
 1.12 s   0.03 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(19)::           tot    1.7915 ncalls 4 ( 69.0%) s gflops    0.36 sn(   0.00 s   0.00 GB/s) dn(
 0.55 s   0.08 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(2)::            tot    1.9061 ncalls 4 ( 75.0%) s gflops    0.33 sn(   0.00 s   0.00 GB/s) dn(
 0.48 s   0.07 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(3)::            tot    1.9016 ncalls 4 ( 61.4%) s gflops    0.34 sn(   0.00 s   0.00 GB/s) dn(
 0.73 s   0.05 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
SQG(4)::            tot    1.8786 ncalls 4 ( 59.2%) s gflops    0.34 sn(   0.00 s   0.00 GB/s) dn(
 0.77 s   0.05 GB/s) tn(   0.00 s   0.00 GB/s) cm(   0.00 s   0.00 GB/s)
The quantum circuit is composed of 40 one-qubit gates and 0 two-qubitgates, for a total of 40 gates.
The greedy depth (all gates lasting one clock cycle) is 4.
```

图 4: 单比特门结果

```
104 }
```

2.3 实验结果

我们在操作系统 Ubuntu 18.04,16G 内存的笔记本电脑上测试了我们的算法, 使用的 cmake 版本为 3.19.6, mpich 使用的版本为 3.3a2, gcc 的版本为 7.5.0. 我们测试了 25 个 qubits(其中 5 个 global qubits,20 个为 local qubits), 使用 32 个进程来进行计算。以上为单比特门的结果. 所用的时间也在表格中列举出了。我们发现该算法确实能起到明显的加速作用，但是还可能存在的问题有:

- 进程数较少，MPI 通信的代价可能并不足以成为 simulation 的瓶颈
- 硬件原因：系统内存限制，只能使用较少的量子比特进行测试。没有使用高性能计算机
- 测试样例较为简单，没有使用随机电路或一些更加实用的线路上测试

3 our work

目前开源的 IQS 中已经实现了 PermuteQubits(σ) 的操作，可以参考 examples/目录下 communication_reduction_via_qubit_reordering.cpp 中给出的示例。但是我们并没有在代码中找到关于有向无

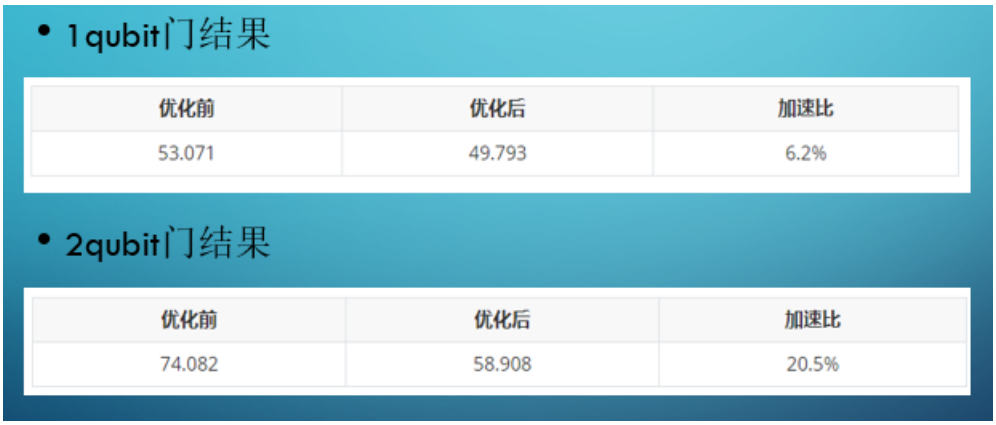

图 5: 计算用时比较

环图算法的部分，而且现在的优化还停留在较粗粒度上。本课题计划延续之前相关工作的思路，大致分为以下几个步骤开展：

1) 完成 IQS 的编译和安装，理解 examples/中的代码示例，能够熟练使用 IQS 编写量子程序。

2) 阅读并理解 IQS 源码实现（src/和 include/）。

3) 完成有向无环图的抽象。根据给定的量子线路构建有向无环图。

4) IQS 中已经提供了 PermuteQubits(σ) 的 API，我们不需要额外实现，熟悉怎么调用就好。

5) 尝试复现之前相关工作中有向无环图上的算法。

4 future work

1) 在此基础上可以考虑更细粒度或是算法层次上的优化。e.g.,

   – 原先的算法并没有考虑实际的通信开销，没有考虑量子门的差异，而是简单地假定应用在 global qubits 上的量子门比应用在 local qubits 的量子门花费更长的时间。我们可以考虑这些因素的影响，给有向无环图作加权处理。

   – 原先算法通过枚举所有 local qubit 和 global qubit 的组合搜索最优交换策略，这在量子比特数目较少时对性能影响不大，是否还存在更好的解决方案?

2) 测试优化前后模拟花费的时间。相关工作中只测试了由一系列单比特量子门 {H, Y} 和多比特量子门 CNOT 随机组合而成的量子线路的模拟时间，我们可以测试在更加实用的量子线路上运行的结果。

参考文献

[1] Mikhail Smelyanskiy, Nicolas P. D. Sawaya, and Alán Aspuru-Guzik. qHiPSTER: The Quantum High Performance Software Testing Environment. *arXiv:1601.07195*, 2016.

[2] Gian Giacomo Guerreschi, Justin Hogaboam, Fabio Baruffa, and Nicolas P. D. Sawaya. Intel Quantum Simulator: A cloud-ready high-performance simulator of quantum circuits. *arXiv:2001.10554*, 2020.

[3] Gian Giacomo Guerreschi. Fast simulation of quantum algorithms using circuit optimization. *arXiv:2010.09746*, 2020.