Optimizing virtual qubit placement in sparse graph-like physical quantum hardware.

Gábor Funk

2026. April

Abstract

Physical quantum hardware is not perfect and in any such system, some qubits are guaranteed to have higher error rate than others. From another point, 2 qubit gate operations will also perform worse on some parts of the circuit. The underlying problem to optimize the virtual to physical qubit mapping is NP hard, which means that currently there is no polynomial algorithm which can supply an optimal placement. My approach offers a heuristic, polynomial algorithm which offers a sub-optimal placement which can reduce errorous outputs by up to $100\% - 2000\%$.

Contents

1 Introduction

Definition 1 *Circuit graph: A circuit graph is a special undirected graph where from each node, there exists a way to any other node and abides these rules:*

1. *Each node represents a physical qubit. The value of the node is the error rate of the qubit.*
2. *There is an edge between 2 nodes if and only if a 2 gate operation can be applied to those 2 qubits at once. The value of the edge is the error rate of the 2-qubit-operation.*

Definition 2 *Connection: There is a connection between 2 qubits if at any point of the circuit a 2 qubit operation is applied between them.*

Definition 3 *Weight of an operation: The weight of an operation is a real number which can be assigned to an operation. If the operation takes up more QCPU time then this number should be large. It can not be smaller than 0 or 0.*

Definition 4 *Importance of a qubit: The importance of a qubit is the sum of the weight of the operations which involves the current qubit.*

2 Simplification of the problem

The algorithm is optimized to find a good virtual qubit to physical qubit mapping for a known physical quantum circuit. The algorithm will provide a "good enough" placement and it will also present a number which indicates how good the placement was. The algorithm will get score if it can reduce the 2 qubit operation and single qubit error rates by moving the qubit to another place and it will also gain score if it can place virtual qubits to neighboring physical qubits if those virtual qubits had a 2 gate operation or data binding. The algorithm will get a penalty score for each SWAP operation it introduces.

3 Naive algorithm

The naive algorithm is to have an array from the physical qubits and apply this code:

```
get physicalQubits[];
get virtualQubits[];

for i = range(0,len(physicalQubits)) in physicalQubits:
    virtualQubits[i].physical = physicalQubits[i];
```

(Assuming that there are more physical qubits than virtual qubits) This algorithm will also serve as a benchmark.

Seeing the performance bottleneck in the algorithm is trivial because the code does not care about connections.

4 Algorithm in theory

The algorithm consists of other graph traversal algorithms like BFS, DFS and Neighbor search extended with sorting algorithms:

```
get physicalQubits[];
get virtualQubits[];
WorstMax = 5;

sort physicalQubits by error_rate;

worstQubitIds[] = get the ID-s of the WorstMax amount of the worst qubits.
//The WorstMax amount of qubits from the end of the sorted physicalQubits.
physicalQubits[]-= worstQubitIds[];

comparingFunction = (qb1,qb2) => return qb1.error_rate > qb2.error_rate;

visited[] = = BFS from the qubit with the best error rate in physicalQubits;
//Optionally, it is a min-heap based priority BFS based on comparingFunction.
ordered[] = visited[] based on comment;

//Optionally, add disconnected qubits based on logic.
...

//Add the worst qubits at the end
ordered[] +=worstQubitIds[];
visited[] +=worstQubitIds[];

physicalOrder[] = ordered[];

vConnections hashmap[qubitID,connectedTo[]] = get virtual connections;

vComparingFunction = max-heap based on qubit importance.

orderedVirtual[] = BFS and order using vComparingFunction;
//Start the upper BFS from the most important qubit in virtualQubits.

//Optionally, here the disconnected parts should be dealt with.

virtualOrder[] = orderedVirtual[];

let vqubit_to_pqubit hashmap[virtQ,physQ]; //virt. to phys. qubit mapping

for i = 0 to min(virtualOrder. count, physicalOrder count):
    vqubit_to_pqubit[physicalOrder[i]] = virtualOrder[i];
```

5 Testing the algorithm and metrics

For simplicity, I tested my algorithm on the *IBM Quantum Falcon processor* which has 27 qubits in a hex-lattice pattern.

I also made a C++ simulation environment where the qubit placement algorithms can be tested and will be used for further measurements: https://afghangoat.hu/imports/src/quantum2/qubit_error_opter.cpp

This section will showcase the metrics and measurements as well as the comparison in performance between the naive and the heuristic algorithm.

Note: The simulation does not account for the Hadamard gate CNOT inversion where the connections are not two-way because the performance impact is negligible.

The are 4 test cases, 3 deterministic and 1 random virtual circuit where the mapping performance is being measured:

1. Sparse graph and local chain structures,
2. A random circuit with 50 % of 2 qubit gate operation. (Will be simulated using 1000 sample montecarlo approach),
3. A dense K>=5 graph,
4. A very simple circuit.

5.1 Sparse graph and local chain structures

This simulation aims to recreate a more formal and larger quantum circuit. It consists of:

Definition 5 *Crossing: A crossing is a part of the virtual qubit layout, where on one qubit multiple other qubits want to perform 2 gate operations.*

1. Local 2 qubit operation chains.
2. Cross-cluster entanglement which forces routing.
3. Star pattern utilizing crossings
4. Long range 2 qubit operations (or perceived long range operations)

5.1.1 The naive algorithm

```
Cost of each qubit operations (if cost is high, good):
Qubit number 26: 0.8125
Qubit number 25: 1.21429
Qubit number 24: 1.41667
Qubit number 23: 2
Qubit number 22: 1.75
Qubit number 21: 1.66667
Qubit number 20: 4.5
ERROR!
Qubit number 19: 8.5
Qubit number 18: 0.95
```

```
Qubit number 17: 1

SWAP Penalty Analysis:
...Consult the simulation program.

Total gain before penalty: 23.8101
Total SWAP penalty:        61.8146
Net cost effectivity:      -38.0044
```

5.1.2 The heuristic algorithm

```
Cost of each qubit operations (if cost is high, good):
Qubit number 26: 1.0625
Qubit number 25: 1.35714
Qubit number 24: 1.41667
Qubit number 23: 1.8
Qubit number 22: 2.5
Qubit number 21: 2.83333
Qubit number 20: 4.5
Qubit number 19: 6.5
ERROR!
Qubit number 16: 0.875
Qubit number 14: 0.833333

SWAP Penalty Analysis:
...Consult the simulation program.

Total gain before penalty: 23.678
Total SWAP penalty:        40.9095
Net cost effectivity:      -17.2316
```

It is trivial to see that based on the previously stated rules and conditions my algorithm outperforms the naive algorithm by 20.7728 points which is approximately 45 % better. It had more than twice better swap penalty ratio and negligibly worse initial placement.

5.2 Random circuit

Since this simulation is non-deterministic, I ran 10000 of them to avoid bias. The basic simulation involved N=15 qubits with G=50 number of gates and P=0.5 (50 %) of 2 qubit operations. Based on measurements, I can state that,

let S where $S = N/G * P$. If for a graph the S value is large, the heuristic algorithm will become more efficient.

Theorem 1 *The S value for sparse graphs will become larger for each fixed N and G value while the S value will become lesser for dense connected graphs.*

Proof: *This is trivial. Take the $S = N/G * P$ equation. By definition, for dense graphs P will be >0.5 and for sparse graphs P will be small so the S will be larger.*

5.2.1 The naive algorithm

```
AVG. Total gain before penalty: 29.5119
AVG. Total SWAP penalty:        96.7869
AVG. Net cost effectivity:      -67.275
```

5.2.2 The heuristic algorithm

```
AVG. Total gain before penalty: 31.9027
AVG. Total SWAP penalty:        77.5905
AVG. Net cost effectivity:      -45.6877
```

Measurements shown to me that for high S values, my heuristic algorithm performs way better, up to 200 % efficiency but for high P values it performs worse than the naive algorithm so for P=1.0 which will be a very densely connected graph, it becomes very inefficient but very few quantum circuits can be modelled with P=1.0 values or large P values.

5.3 Premade dense graph

The random graph placement makes this insignificant for the measurement, but I left it in the simulation as an example where the heuristic algorithm performs worse (by 40 %) than the naive one.

5.4 A small, premade circuit

This small circuit aims to show a real-life example. For this, the naive algorithm already performs quite good, but the heuristic algorithm improves the rate by up to 2000 %.

5.4.1 The naive algorithm

```
Total gain before penalty: 1.40595
Total SWAP penalty:        1.09091
Net cost effectivity:      0.315043
```

5.4.2 The heuristic algorithm

```
Total gain before penalty: 2.61905
Total SWAP penalty:        0
Net cost effectivity:      2.61905
```

Note: In this example, my algorithm found the optimal placement. Optimal placement means that there is no better placement and no SWAP gates are needed.

6 Proving the correctness

6.1 Complexity

The algorithm consists of only operations with polynomial complexity, so its O value will be also polynomial.

To determine the precise Big-$\mathcal{O}$ time complexity of the `apply_circuit` algorithm, we first define variables for the sizes of the different input structures:

- P: Number of physical qubits (`qubits.size()`)
- E_p: Number of connections between physical qubits (edges in the physical graph)
- V: Number of virtual qubits (`vc.vqubits.size()`)
- G: Number of gates in the virtual circuit (`vc.gates.size()`)
- E_v: Number of unique 2-qubit virtual connections (where $E_v \leq G$)

Here is the step-by-step breakdown of the operations within the function:

1. Initialization

```
for (auto& [id, q] : qubits) // ...
for (auto& [id, vq] : vc.vqubits) // ...
```

Extracting the initial orders scales linearly with the number of qubits.
Complexity: $\mathcal{O}(P + V)$

2. Physical Qubit Sorting (`optimization > 0`)

```
std::sort(allQubits.begin(), allQubits.end(), [](const Qubit* a, const Qubit* b) {
    return a->error_rate < b->error_rate;
});
```

Copying P elements into `allQubits` and sorting them based on their error rate.
Complexity: $\mathcal{O}(P \log P)$

3. Isolating the Worst Qubits

Finding and inserting the worst 5 qubits into an `std::unordered_set`.
Complexity: $\mathcal{O}(1)$ (since it is capped at a constant maximum of 5).

4. Physical Qubit Priority BFS

```
while (!pq.empty()) {
    int current = pq.top();
    pq.pop();
    // ...
    for (int neighbor : qubits.at(current).connections) { ... }
}
```

Because a neighbor is inserted into the `visited` set *before* being pushed to the priority queue (`pq`), each physical qubit is pushed and popped at most once. Pushing/popping from `std::priority_queue` takes $\mathcal{O}(\log P)$ time, occurring up to P times, yielding $\mathcal{O}(P \log P)$. The inner loop iterates over the `connections`; across the entire `while` loop, this block executes proportional to the total number of physical edges, E_p. Lookups and insertions into `std::unordered_set` take $\mathcal{O}(1)$ on average.
Complexity: $\mathcal{O}(P \log P + E_p)$

5. Remaining Physical Assignments

Iterating through `allQubits` to append unvisited and worst-performing qubits.
Complexity: $\mathcal{O}(P)$

6. Building Virtual Connections

```
for (const auto& gate : vc.gates) {
    if (gate.qubits.size() == 2) { ... }
}
```

Iterating through all gates to build the `vConnections` map.
Complexity: $\mathcal{O}(G)$

7. Virtual Qubit Priority BFS

```
while (!vpq.empty()) {
    // ...
    for (int neighbor : vConnections[current]) { ... }
}
```

This mirrors the physical BFS logic. Each virtual qubit is pushed/popped at most once, and the inner loop processes the total virtual edges E_v.
Complexity: $\mathcal{O}(V \log V + E_v)$

8. Final Mapping Assignment

```
for (size_t i = 0; i < mappingCount; i++) {
    // ... assignment logic
}
```

Looping up to the minimum of P and V.
Complexity: $\mathcal{O}(\min(P, V))$

Final Evaluation

When we combine all these steps together, we get:

$$\mathcal{O}(P + V) + \mathcal{O}(P \log P) + \mathcal{O}(P \log P + E_p) + \mathcal{O}(P) + \mathcal{O}(G) + \mathcal{O}(V \log V + E_v) + \mathcal{O}(\min(P, V))$$

We can drop the less dominant terms (like the standalone $\mathcal{O}(P)$ and $\mathcal{O}(V)$, which are overpowered by the logarithmic terms). Furthermore, because E_v (the number of unique virtual edges) can never exceed G (the total number of gates), we can absorb E_v into G.

This gives us the **Exact Time Complexity**:

$$\mathcal{O}(P \log P + E_p + V \log V + G)$$

The algorithm successfully operates in polynomial time, avoiding the exponential blowup commonly associated with naive subgraph isomorphism approaches to qubit routing and mapping.

7 Outcome

Altough this is not the most efficient mapping algorithm it is cheap and easy to implement.

References

[1] C++ virtual qubit mapping algorithm framework,
https://afghangoat.hu/imports/src/quantum2/qubit_error_opter.cpp