一次元アーキテクチャへの搭載に向けた量子回路設計支援アプリケーション

東京大学大学院　情報理工学系研究科　電子情報学専攻

長谷川研究室　修士1年　内藤壮俊

自己紹介

- プログラミング経験
  - 競技プログラミング (C++)
  - ゲーム開発 (Unity C#)
  - 研究，ウェブ開発 (Python, JavaScript)

- 量子コンピューティングの経験
  - 量子ゲート型：IBM Quantum Challenge 2020 に参加した程度
  - 量子アニーリング型：今回が初めて

参加したきっかけ

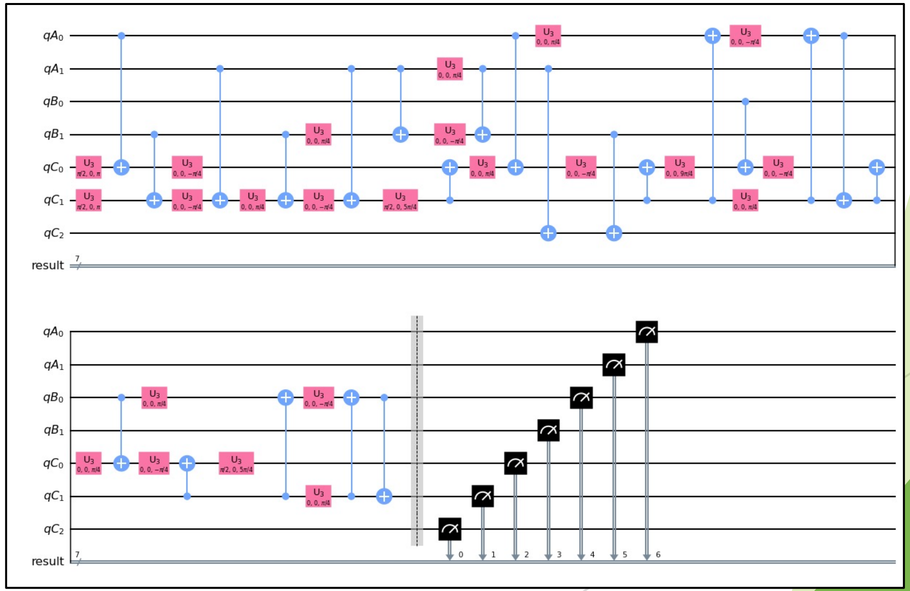
- 量子アニーリングを使って，量子ゲートの回路設計を支援できないか？
  - コラボって感じがしてカッコいい
- 現存するゲート型量子コンピュータは数十ビット程度の規模なので，扱う問題の大きさとしてちょうど良い…？
  - アニーリングが先行している分だけ，アドバンテージが活かせそう

背景説明

量子ゲート型計算機

- ゲート通過による状態変化　→　測定　により計算を行う
  - 量子ビットは複数の状態を重ね合わせることが可能

- 任意の量子回路はU3ゲート(1入力)とCXゲート(2入力)に展開可能
  - U3ゲート：1ビットの状態を任意に操作
  - CXゲート：2ビット間でXOR演算を行う
    - $(x,\ y) \rightarrow (x,\ x \oplus y)$

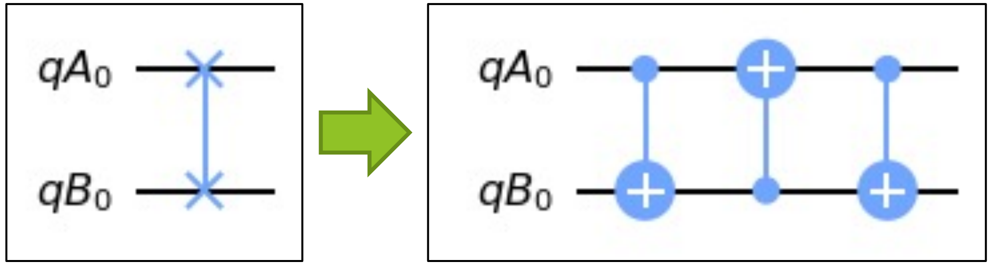


図：U3ゲートとCXゲートに展開された量子回路.

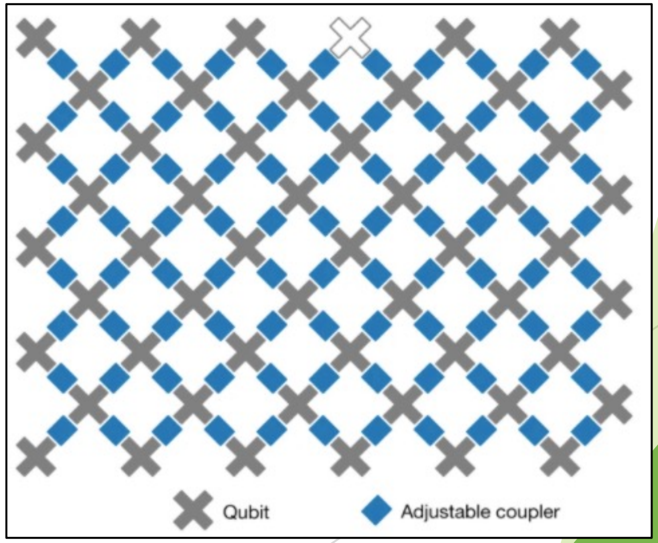
Qubit Allocation Problem
(量子ビット割り当て問題)

- 設計図上の「論理ビット」とデバイス上の「物理ビット」を対応させる問題
- CXゲートを作用させる物理ビットは隣り合っている必要がある
  - 離れている場合は？　→　SWAPゲートを使って物理ビットの中身を交換

- NISQデバイスでは，ゲート操作によるエラーが重要
  - SWAPゲートはCXゲート3つ分
  - エラー率は U3ゲート << CXゲート << SWAPゲート



図：SWAPゲートの構成.



図：物理ビットの配置の例.

先行研究

- “A QUBO formulation for qubit allocation" (2020)
  Bryan Dury and Olivia Di Matteo [1]
  - エラー率と回路の深さを考慮して，論理ビットの初期配置を決定する
    - $x_{ij} = 1$：「i 番目の論理ビットは j 番目の物理ビットに対応する」
    - $cost = \sum_{i,j,k,l} Q_{ijkl} \cdot x_{ij} x_{kl} + \sum_{i,j} b_{ij} \cdot x_{ij}$
    - $\underbrace{Q_{ijkl} = -\ln(p_{jl}) \cdot g_{ik} \cdot d_{jl}^3}_{\text{CXゲートのコスト}}$　$\underbrace{b_{ij} = -\ln(p_j) \cdot g_i}_{\text{U3ゲートのコスト}}$

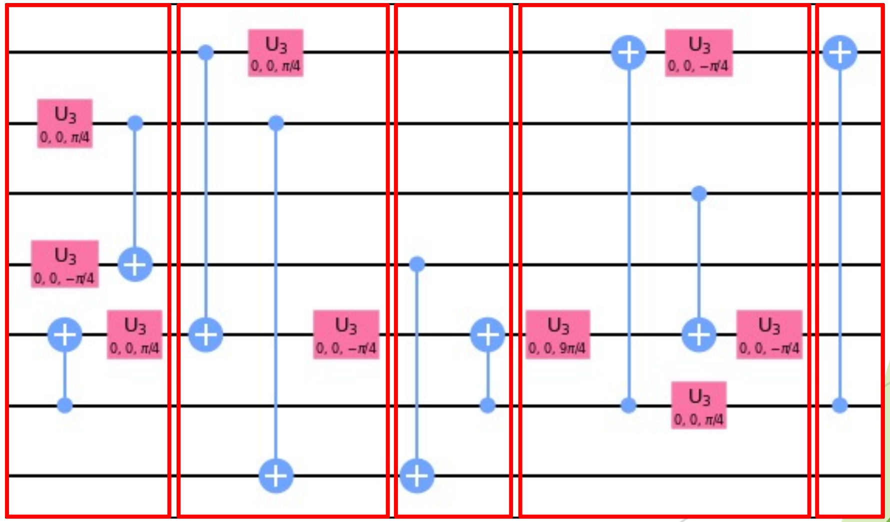
      CXゲートのコスト
      [エラー]・[個数]・[距離]

      U3ゲートのコスト
      [エラー]・[個数]
  - ダイナミックな並び替えは考慮せず
    - CXゲートごとに，「2ビットが隣り合うように並び替え → 配置を戻す」の繰り返し

[1] https://arxiv.org/abs/2009.00140

今回扱いたい問題

- 簡単のため，物理ビットが一列に並んだアーキテクチャを考える
- CXゲートを含むそれぞれのレイヤーに対して，論理ビットの配置を決定する
  - CXゲートが論理ビットを共有しないように，左からレイヤーを構成
  - レイヤー間にSWAPゲートを挿入し，物理ビット (の中身) を並び替える
- コスト ＝ 用いるSWAPゲートの個数
  - 1次元アーキテクチャの場合，転倒数で定式化可能



図：レイヤーの構成例.

古典的なアプローチ (動的計画法による高速化)

- 量子ビットの個数 N に対し，各レイヤーにおける配置は $N!$ 通り
- レイヤーの枚数 M に対して，全体の取りうる状態数は $(N!)^M$ 通り

- 配置に対する暫定的なコストを持っておくことで，
空間計算量 $O(M \cdot N!)$，時間計算量 $O(M \cdot (N!)^2)$ で解くことができる

- $N = 10$ で $(N!)^2 \approx 1.3 \times 10^{13}$ なので，小規模の回路にしか適用できない．

背景説明　まとめ

- CXゲートを作用させる物理ビットは隣り合っていなければならない
- 物理ビットの (中身の) 入れ替えはSWAPゲートによって実現できるが，
エラー率が高いためできるだけ使いたくない
- 先行研究では初期配置の最適化を行っていたが，並び替えの順番の最適化は行われていない
- 古典的解法は計算量が非常に大きく，小規模な回路にしか使えない

提案手法

バイナリ変数を用いた定式化

- 各レイヤーにおける論理ビットは $[0, 1, \cdots, N-1]$ の並び替えとなる
- $\boldsymbol{Q_{mnv}}$：「レイヤー m において，物理ビット n は論理ビット v に対応する」
  - MN^2 個のバイナリ変数が必要

- one-hot 制約
  - 「物理ビットは単一ビットのみに対応する」：$\sum_{v=0}^{N-1} \boldsymbol{Q_{mnv}} = 1$
  - 「論理ビットは単一ビットのみに対応する」：$\sum_{n=0}^{N-1} \boldsymbol{Q_{mnv}} = 1$

- CXゲートによる制約
  - 作用させる物理ビットは隣り合っていなければならない
  - ペナルティ関数：$\sum_{(a,b)\in[CX-gates]} \sum_{(i,j),|i-j|\geq 2} \boldsymbol{Q_{mia}}\, \boldsymbol{Q_{mjb}}$

コスト関数の定式化 (厳密解法)

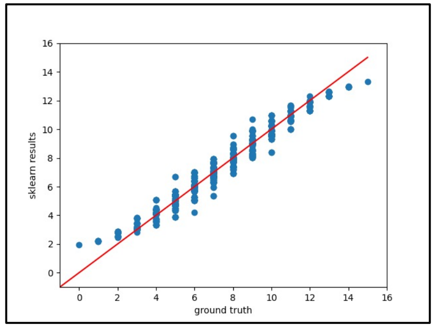
- 隣り合うレイヤー A, B 間における，論理ビットの移動を考える
  - $C_{ij} = 1 \;\leftrightarrow\; A[i] = B[j]$
- 補助変数+制約の追加による定式化
  - $S_{ijv} = A_{iv} \cdot B_{jv} \;\leftrightarrow\; penalty = 3S_{ijv} + A_{iv}B_{jv} - 2S_{ijv}(A_{iv} + B_{jv})$
  - $C_{ij} = \sum_v S_{ijv}$
- この時，転倒数 (= 論理ビットの入れ替わり) は厳密に定式化可能
  - $cost = \sum_{0 \le i_1 < i_2 < N} \sum_{0 \le j_2 < j_1 < N} C_{i_1 j_1} \cdot C_{i_2 j_2}$

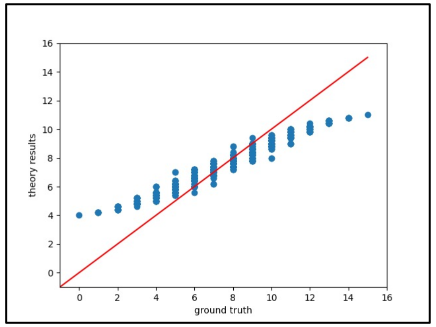
厳密解法の実行結果

- コストが非常に大きくなる + 実行可能解が見つからないという結果に．．．
- 探索空間の大きさに対して，制約を満たす組が少ないことが原因か
  - 用いたバイナリ変数の合計 $= \log_2(\text{探索空間の状態数}) = MN^3 + 2MN^2 - N^3 - N^2$
  - $\log_2(\text{解となる状態数}) < \log_2(N!)^M \approx \frac{1}{\log 2} M(N \log N - N)$
- ほとんど全てが「ハズレ」だった

転倒数の近似によるアプローチ

- 性能向上のため，近似解法を採用して状態数を削減することに
  - 転倒数を2次で表現できれば，用いるバイナリ変数は MN^2個で済む
- 重回帰分析によるフィッティング，期待値による推定を行った





図：重回帰分析，期待値による転倒数の推定結果.
横軸が正しい値，縦軸が推定値.

- 2つの推定結果は，互いに一次関数の関係になっていた
  - [重回帰分析による推定結果] = $\frac{2N-2}{\mathrm{N}}$ [期待値による推定結果] − $\frac{(N-1)(N-2)}{4}$

コスト関数の定式化 (近似解法)

- 期待値による推定結果 $\approx \left\{\sum_{0\le i,j<N}\left(\frac{i+j}{2}-\frac{ij}{N-1}\right)\cdot C_{ij}\right\}$
- 重回帰分析による推定結果へ変換： $\mathrm{y}=\frac{2N-2}{\mathrm{N}}x-\frac{(N-1)(N-2)}{4}$

- → 転倒数 $\approx \frac{2N-2}{\mathrm{N}}\left\{\sum_{0\le i,j<N}\left(\frac{i+j}{2}-\frac{ij}{N-1}\right)\left(\sum_{v=0}^{N-1} A_{iv}\, B_{jv}\right)\right\}-\frac{(N-1)(N-2)}{4}$ と書ける
  - レイヤー $m,\ m+1$ 間においては，$A_{iv}=\boldsymbol{Q}_{\boldsymbol{miv}}$，$B_{jv}=\boldsymbol{Q}_{(\boldsymbol{m}+\boldsymbol{1})\boldsymbol{jv}}$

- 代入すると，以下のように整理できる
  - $\underbrace{\sum_{m=0}^{M-2}\left\{\sum_{0\le i,j<N}\frac{(N-1)(i+j)-2ij}{N}\left(\sum_{v=0}^{N-1}\boldsymbol{Q}_{\boldsymbol{miv}}\,\boldsymbol{Q}_{(\boldsymbol{m}+\boldsymbol{1})\boldsymbol{jv}}\right)\right\}}_{\text{この部分を最小化したい.}}-\underbrace{(M-1)\frac{(N-1)(N-2)}{4}}_{\text{定数項. 最小化においては無視される.}}$

Amplify解法モデルの構成

- $cost = \sum_{m=0}^{M-2}\left\{\sum_{0\le i,j<N}\frac{(N-1)(i+j)-2ij}{N}\left(\sum_{v=0}^{N-1}\boldsymbol{Q_{miv}}\,\boldsymbol{Q_{(m+1)jv}}\right)\right\}$
- $constraint = \sum_{m=0}^{M-1}\left\{\underbrace{\sum_{v=0}^{N-1}\left(1-\sum_{n=0}^{N-1}\boldsymbol{Q_{mnv}}\right)^2+\sum_{n=0}^{N-1}\left(1-\sum_{v=0}^{N-1}\boldsymbol{Q_{mnv}}\right)^2}_{\text{one-hot 制約}} + \underbrace{\sum_{(a,b)\in[CX-gates]}\sum_{(i,j),|i-j|\ge 2}\boldsymbol{Q_{mia}}\,\boldsymbol{Q_{mjb}}}_{\text{CXゲートによる制約}}\right\}$

- $model = constraint \times \lambda + cost$ として構成した
  - $model$ の項数(= モデルの規模)は $O(MN^3)$ 個
  - 制約 >> コストとするために，とりあえず $\lambda = 100$ と設定

パフォーマンスの評価 (コスト最小化の性能比較)

- ランダムに生成したデータ10個に対してコストを計算した
  - 古典的解法は最適解を出力するので，必ず 古典的解法 ≤ Amplify解法 となる
  - Amplify解法においては，$\lambda = 100$, $timeout$ = 1秒 として実行

- N, M の大きいケースで誤差が大きくなった
  - 制約の重み λ を小さくする + $timeout$ を伸ばすことでコスト抑制が可能

N	3	4	5	6
古典的解法 (M = 5)	0.6	1.4	2.1	4.1
Amplify解法 (M = 5)	0.6	1.4	2.1	4.5
古典的解法 (M = 20)	3.8	13.0	15.0	24.9
Amplify解法 (M = 20)	3.8	13.1	18.9	34.0

表：それぞれの解法における
コストの平均値.

パフォーマンスの評価 (実行時間の比較)

- $M = 5$ にて，N を動かした時の実行時間(秒)を比較した

N	3	4	5	6	7	8	9	10	15	20
古典的解法	0.0007	0.0084	0.4551	8.9532	566.95	—	—	—	—	—
Amplify解法	1.8336	1.8184	1.4685	1.1751	1.2766	1.3620	1.3654	1.6483	6.1014	20.047

表： N を動かした時の実行時間の比較.
$6 \leq N$ においてAmplify解法の方が高速となっている.

- N が大きくなると，Amplify解法でも時間がかかる傾向に
  - $O(MN^3)$ サイズのモデル構築に時間がかかっていた
- といっても，古典的解法は $O(M \cdot (N!)^2)$ なので飛躍的向上と言える
  - $N = 10$ のとき，$(10!)^2 \div 10^3 \approx 132$億倍の高速化に成功

提案手法　まとめ

- 厳密解法は使い物にならなかったため，近似解法を採用した
- N が小さい場合は古典的解法が強く，N が大きい場合はAmplify解法が強い

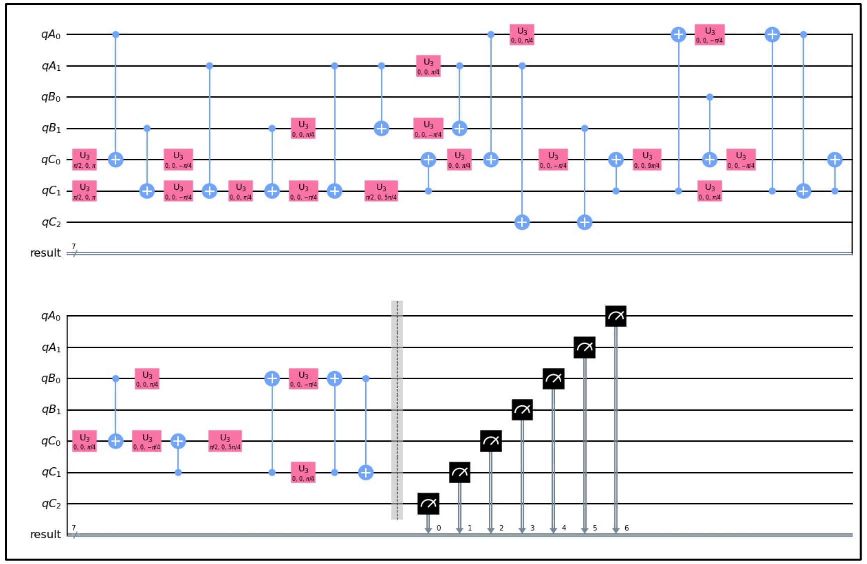
	古典的解法	Amplify解法	
		厳密解法	近似解法
大域最適解	計算可能	計算可能	近似解のみ
実用的な範囲	$N \leq 6$ $M \leq 100$	N, M ともに 小さい場合	$N \leq 20$ $\mathrm{M} \leq 100$

表：古典的解法とAmplify解法の比較．
Amplify解法の方が実用的と言える．

アプリケーションの作成

OpenQASMとの連携

- 「OpenQASM」という言語で書かれた回路を入力できるようにしたい
- U3ゲートとCXゲートに分解済みの回路を入力に用いる



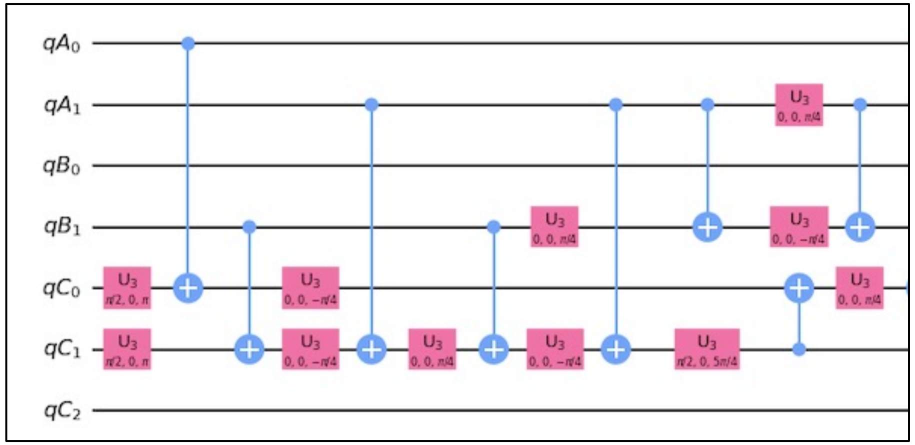
図：展開した後の回路.

```
・・・
u3(pi/2,0,pi) qC[0];
cx qA[0],qC[0];
u3(0,0,-pi/4) qC[0];
u3(pi/2,0,pi) qC[1];
cx qB[1],qC[1];
u3(0,0,-pi/4) qC[1];
cx qA[1],qC[1];
u3(0,0,pi/4) qC[1];
cx qB[1],qC[1];
u3(0,0,pi/4) qB[1];
u3(0,0,-pi/4) qC[1];
cx qA[1],qC[1];
cx qA[1],qB[1];
u3(0,0,pi/4) qA[1];
u3(0,0,-pi/4) qB[1];
cx qA[1],qB[1];
u3(pi/2,0,5*pi/4) qC[1];
cx qC[1],qC[0];
u3(0,0,pi/4) qC[0];
cx qA[0],qC[0];
u3(0,0,pi/4) qA[0];
u3(0,0,-pi/4) qC[0];
cx qC[1],qC[0];
u3(0,0,9*pi/4) qC[0];
cx qB[0],qC[0];
u3(0,0,-pi/4) qC[0];
cx qC[1],qA[0];
u3(0,0,-pi/4) qA[0];
u3(0,0,pi/4) qC[1];
cx qC[1],qA[0];
cx qA[0],qC[1];
cx qC[1],qC[0];
u3(0,0,pi/4) qC[0];
cx qB[0],qC[0];
u3(0,0,pi/4) qB[0];
u3(0,0,-pi/4) qC[0];
cx qC[1],qC[0];
u3(pi/2,0,5*pi/4) qC[0];
cx qC[1],qB[0];
u3(0,0,-pi/4) qB[0];
u3(0,0,pi/4) qC[1];
cx qC[1],qB[0];
cx qB[0],qC[1];
cx qA[1],qC[2];
cx qB[1],qC[2];
・・・
```

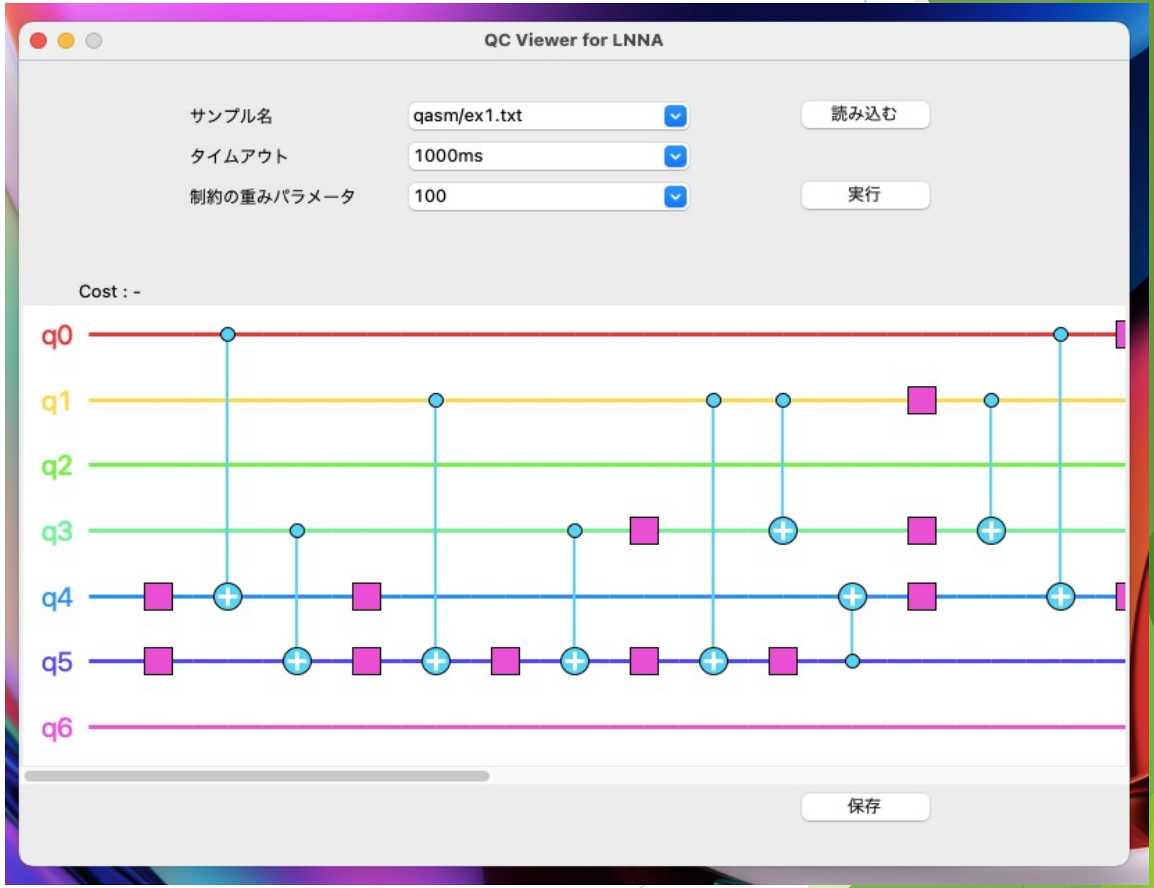
図：出力されるOpenQASMプログラム.

アプリ機能紹介：量子回路の描画

- tkinterというライブラリで実装
- 設計図段階の回路と，Amplify解法の実行結果を描画する



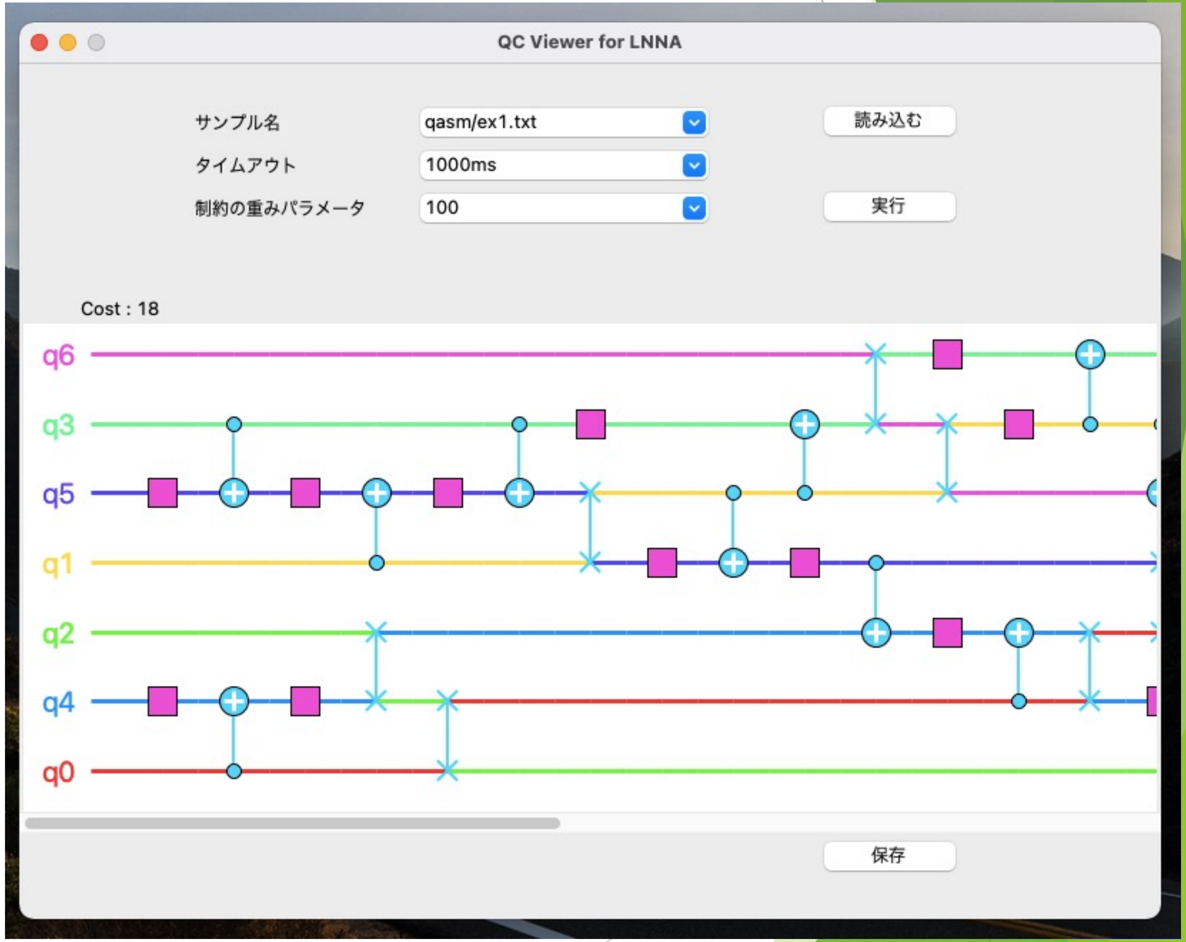
図：読み込んだ回路設計図の一部．
(IBM Q上で描画．)



図：ビジュアライザ上での描画結果．

アプリ機能紹介：Amplifyの呼び出し

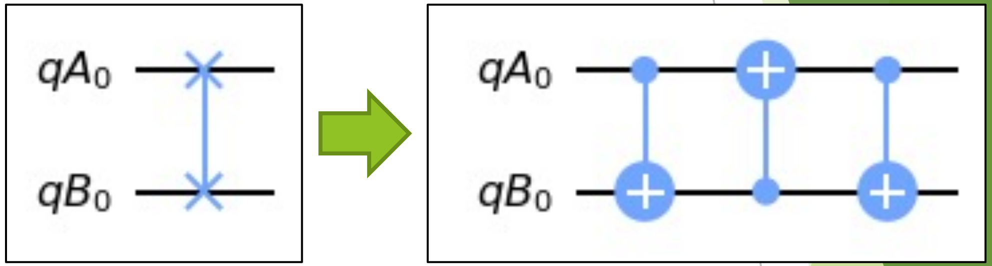
- Amplify解法を実行
  - タイムアウト ($timeout$) の調整可
  - 制約重みパラメータ (λ) の調整可
- 実行結果の描画
  - SWAPゲートは「X—X」で描画されている
- 良い解が見つかるまで調整・再試行が可能
  - 回路設計の効率化につながる



図：Amplify解法の実行結果.

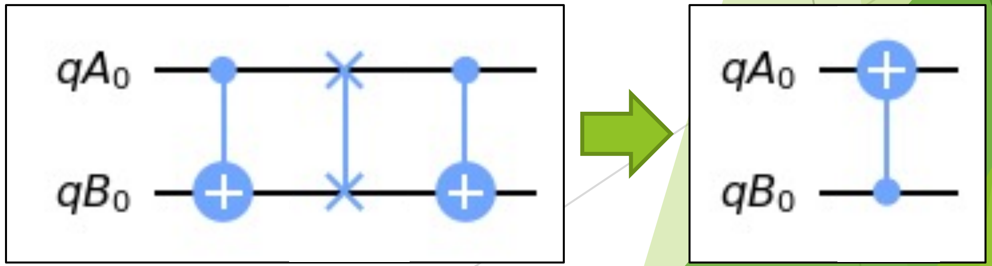
アプリ機能紹介：OpenQASM形式で出力

- SWAPゲートを3つのCXゲートに置き換えることでOpenQASM形式に書き換えることが可能
  - 設計図段階の回路を入力して，アプリで実機搭載可能な回路に変換して，その結果を出力する



図：SWAPゲートの展開．

- CXゲートの個数が最も少なくなるように置き換えを実行
  - 上下反転を考えると，置き換えは2通りある
  - CXゲートが相殺する場合がある



図：CXゲートの個数が減るケース．

アプリケーションの作成　まとめ

設計図段階の
量子回路の入力・描画

↓

$timeout, \lambda$ の調整

↓

Amplify解法の実行
実行結果の描画

↓

実機搭載可能な
量子回路の出力

- 一連の作業がアプリ内で完結
- フィードバックが高速
- 他アプリとの互換性

発表は以上になります.
ありがとうございました.