Quantum Look-up Table (QLUT)

https://youtu.be/FTgtRGGvpTE

Reference 논문

- **A Q# Implementation of a Quantum Lookup Table for Quantum Arithmetic Functions**

- 현재 NTU와 진행 중인 연구는 **QLUT를 이용한 딥러닝 활성화 함수 연산**

- **향후 ATOM에서 키 스케줄링을 위해 LUT를 사용하는 부분에 적용해보자는 이야기가 나온 상태고 진행 X**
  - 아마도 경배님, 저, NTU 사람1, 박시, 박시 친구

A Q# Implementation of a Quantum Lookup Table for Quantum Arithmetic Functions

Rajiv Krishnakumar
Core R&D
Goldman Sachs
Geneva, Switzerland
rajiv.krishnakumar@gs.com

Mathias Soeken
Microsoft Quantum
Microsoft
Zürich, Switzerland
mathias.soeken@microsoft.com

Martin Roetteler
Microsoft Quantum
Microsoft
Redmond, WA, USA
martin.roetteler@microsoft.com

William Zeng
Core R&D
Goldman Sachs
New York, NY, USA
william.zeng@gs.com

QLUT의 사용이 이득이 되는 경우

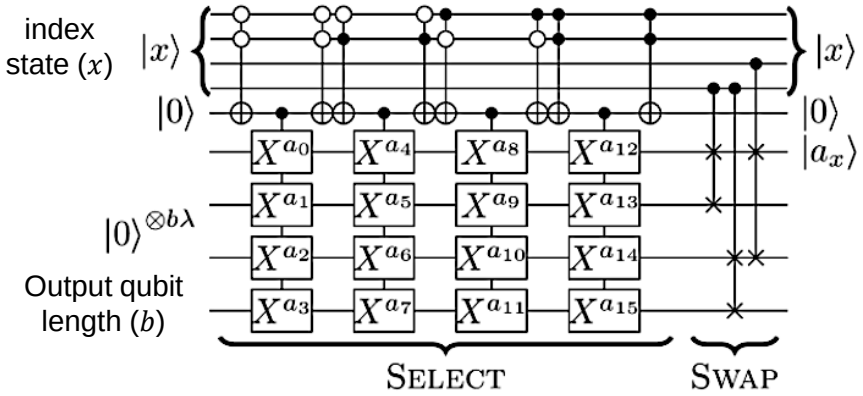
- **양자회로의 reverse 연산 보다 LUT 구현이 효율적인 경우가 존재**
  - **Output에 약간의 오차가 허용**되는 경우
    - 정확한 값이 아니라 근사해도 되는 경우
    - **DL의 활성화 함수** : 근사화 할 수 있는 산술 연산
  - **입력의 범위가 제한된 경우**
    - DL의 경우 **활성화 함수 입력이 -4 ~ 4 사이 정도로 봐도 된다고**

      2023 암호연구회에서 동형암호 관련 발표하신 교수님이 말씀하심

- **따라서, 딥러닝의 활성화 함수에 적용하기 적합**
  - **복잡한 활성화 함수일수록 계산 효율성 향상될 것**

QLUT 구성 방식

- **SELECT-SWAP 양자 회로 사용**



- **SELECT**
  - 입력 값에 따라 unitary 연산 수행
  - 입력 값에 해당하는 Data select 하기 위한 회로

$$\textsc{Select} = \sum_{x=0}^{N-1} |x\rangle\langle x| \otimes U_x.$$

- **SWAP**
  - 양자 자원 절약이 가능하다는데 왜 필요한지는 모르겠습니다.
  - 구현 상 필요한 모듈이라 사용은 하는데, 수학적으로 이해는 X..

- **SELECT-SWAP**
  - QLUT 구현을 용이하게 하는 회로
  - N개의 상태 저장 가능 (해당 그림에서 $N = 16$)

QLUT 알고리즘

- **QLUT 전체적인 알고리즘**
  - **입력**: 타겟 함수 ($f(x)$), 입력 도메인(x_{min}, x_{max}), 에러 범위($\epsilon_{in}, \epsilon_{out}$), swap용 큐비트 수($l$)

    → **입력 도메인과 큐비트의 수로 정밀도를 정할 수 있음**

    → 해당 구현에서 **T-depth는 constant**함 ; 8,16,...,128 ; 이에 따라 안실라 큐비트 수 결정됨
  - **출력**: in/output으로 사용될 전체 큐비트 수, 고정 소수점을 위한 레지스터 수

- **동작 과정**

  1. $\hat{x_i}$ 계산 : 주어진 (x_{min}, x_{max}) 및 ϵ_{in}, (n, p)에서의 input list

  → input을 위한 ; n : 정수를 위한 큐비트의 수, p : 소수점을 위한 큐비트의 수

  2. $f(\hat{x_i})$ 계산: 주어진 정밀도 (m, q)안에서 $f(x)$ 계산

  → x의 근사 ($\hat{x_i}$)를 넣음 → $f(x)$ 의 근사 값 ($f(\hat{x_i})$) 생성

  → LUT를 위한 ; m : 정수를 위한 큐비트의 수, q : 소수점을 위한 큐비트의 수

  3. x_i , $f(\hat{x_i})$, l 사용하여 SELECT-SWAP 회로 생성

Algorithm 1 The wrapper function ApplyFunctionWithLookup that implements the quantum LUT for arithmetic operations

1: **Inputs**
$f(\cdot)$: Desired arithmetic function to implement
$(x_{\min}, x_{\max})$: Input domain (with inclusive bounds)
$(\epsilon_{\text{in}}, \epsilon_{\text{out}})$: Maximum allowed error for the input value and precision tolerance for the output value
l: Number of qubits to be used as swap qubits

2: **Outputs**
LUT: Lookup table Q# operation
(n, p): Number of total bits and integer bits respectively required for the input fixed-point register to the LUT
(m, q): Number of total bits and integer bits respectively required for the output fixed-point register of the LUT

3: **procedure** ApplyFunctionWithLookup $(f(\cdot), (x_{\min}, x_{\max}), (\epsilon_{\text{in}}, \epsilon_{\text{out}}), l)$

1) Compute the list of inputs $\hat{x}_i$ in fixed point representation as well as (n, p) given $(x_{\min}, x_{\max})$ and ϵ_{in}
2) Using the inputs $\hat{x}_i$, compute the list of outputs $\hat{f}(\hat{x}_i)$, the approximation of $f(\hat{x}_i)$ in fixed-point representation to within a precision of ϵ_{out}, and (m, q)
3) If $x_{\min} \neq 0$, create a Subtraction circuit that takes in a fixed-point register and subtracts $x_{\min}$ from it
4) Create the SelectSwap circuit using $\hat{x}_i$, $\hat{f}(\hat{x}_i)$ and l and append it to the Subtraction circuit

QLUT 실제 코드

- 저는 Q#으로 진행했었는데 1저자인 분이 알고보니 Qiskit으로 진행 중이어서 제 코드는 증발하였지만..

  이 세미나는 제가 공부한 **Q#** 코드로 진행하겠습니다.

- **간단한 overview**
  - LUT 입력 : 타겟 함수 $(f(x))$, 입력 도메인(x_{min}, x_{max}), 에러 범위$(\epsilon_{in}, \epsilon_{out})$, swap용 큐비트 수$(l)$
  - LUT 출력 : [[**00…00,ouput1**],[**00…01,ouput2**], …,[**11…11,last output**]]
  - 위와 같이 table 형태를 갖게 됨

QLUT 실제 코드 – QLUT (1)

- 연산 대상 함수 정의

```
function ExpInv(x:Double) : Double {
    return ExpD(-x);
}
```

- 우측 코드는 QLUT 세부 코드
  - 입력: 함수, 도메인, 에러 범위
  - 출력: LUT 출력
  - 사용된 내부 함수는 아래와 같음

- FunctionWithLookupTable
- SignedLittleEndian
- FixedPoint
- FixedPointAsBoolArray
- DoubleAsFixedPoint
- BoolArrayAsFixedPoint
- PrepareFxP
- SubtractFxP
- InvertFxP
- AddFxP
- AddConstantFxP
- Invert2sSI
- AddI
- IdenticalPointPosFactFxP
- LookupOperationWrapper
- WriteBits
- MakeWriteBitsUnitary
- Select
- ApplyFunctionWithLookupTable
- ExpInv

```
// LUT
function ApplyFunctionWithLookupTable(func: Double -> Double, domain: (Double, Double), epsIn: Double, epsOut: Double): FunctionWithLookupTable {

    let (minIn, maxIn) = domain;

    let pLower = BitSizeI(Ceiling(AbsD(minIn)));
    let pUpper = BitSizeI(Ceiling(AbsD(maxIn)));
    let pIn = MaxI(pLower, pUpper) + 1; // The +1 is for the sign bit

    let qIn = Ceiling(Lg(1.0/epsIn));
    let qOut = Ceiling(Lg(1.0/epsOut));

    // Compute approximations of minIn and maxIn
    let minInFxP = DoubleAsFixedPoint(pIn, qIn, minIn);
    let maxInFxP = DoubleAsFixedPoint(pIn, qIn, maxIn);

    // Compute number of values in between minIn and maxIn
    let deltaIn = 1.0/PowD(2.0, IntAsDouble(qIn));
    let numValues = Truncate((maxInFxP - minInFxP) / deltaIn) + 1;

    mutable outValues = [0.0, size=numValues]; // List to store all the output values (initialized at all 0s)
    mutable inValueFxP = minInFxP; // Starting input value
    mutable pOut = 0; // Set initial pOut value which will be updated in loop below
    for i in 0..numValues-1 {
        // First a quick check to see that the enumaration is going correctly, i.e. that we are hitting all the values in order
        let inAddress = BoolArrayAsInt(FixedPointAsBoolArray(pIn, qIn, inValueFxP - minInFxP));
        EqualityFactI(inAddress, i, $"Unexpected address in enumeration");

        // Now we compute the output value, compute the number of integer bits it has and see if it is bigger than our current pOut
        let outValue = func(inValueFxP);
        set outValues w/= i <- outValue; // this is the syntax to say "outValues = outValues but with the ith index as outValue"
        set pOut = MaxI(pOut, BitSizeI(Ceiling(AbsD(outValue)))+1); //the +1 is for the sign bit
        set inValueFxP += deltaIn;
    }

    let outValuesFxP = Mapped(DoubleAsFixedPoint(pOut, qOut, _), outValues);
    let outBits = Mapped(FixedPointAsBoolArray(pOut, qOut, _), outValues);
    let lookupOperation = LookupOperationWrapper(minInFxP, outBits, _, _);


    return FunctionWithLookupTable(
        pIn, qIn,
        pOut, qOut,
        lookupOperation
    );
}
```

QLUT 실제 코드 - main

- FunctionWithLookupTable(pIn, qIn, pOut, qOut,lookupOperation)
  → ApplyFunctionWithLookupTable를 통해 integer랑 fraction 비트 수를 반환

- **pIn + qIn :** $\hat{x_i}$: (입력값(x)에 에러 범위 포함)
  → based on input list

- **pOut+qOut :** $f(\hat{x_i})$ **:** $(f(x)$에 에러 범위 포함)
  → output list based on input list

- 입력 : func, domain, epsIn, epsOut 모두 **Double type**
- 출력 : **Table**

```
    return FunctionWithLookupTable(
        pIn, qIn,
        pOut, qOut,
        lookupOperation
    );
}
```

```
@EntryPoint()
operation Main() : Unit{

    let lookup = ApplyFunctionWithLookupTable(ExpInv, (0.0,1.0), 0.125 , 0.01); // f, (x_min, x_max), e_in, e_out -> return (

    Message($"lookup::IntegerBitsIn: {lookup::IntegerBitsIn}");
    Message($"lookup::FractionalBitsIn: {lookup::FractionalBitsIn}");

    Message($"lookup::IntegerBitsOut: {lookup::IntegerBitsOut}");
    Message($"lookup::FractionalBitsOut: {lookup::FractionalBitsOut}");


    use input = Qubit[lookup::IntegerBitsIn+lookup::FractionalBitsIn];

    use output = Qubit[lookup::IntegerBitsOut+lookup::FractionalBitsOut]; // 여기가  큐비트가 너무 많으면 할당이 안 됨

    lookup::Apply(FixedPoint(lookup::IntegerBitsIn,input),FixedPoint(lookup::IntegerBitsOut,output)); // fx_{hat}
    mutable cnt_zero = 0;
    mutable cnt_one = 0;

    for idx in 0 .. lookup::IntegerBitsOut+lookup::FractionalBitsOut-1 { // lookup::IntegerBitsOut+lookup::FractionalBitsOut

        Message($"idx: {idx}");
        // 측정 결과에 따라 인코딩된 비트를 확인
        if Measure([PauliZ], [output[idx]]) == Zero {
            set cnt_zero = cnt_zero + 1;
            Message($"[output[idx]]: {[output[idx]]}");
        }
        else{
            set cnt_one = cnt_one + 1;
            Message($"[output[idx]]: {[output[idx]]}");
            }
    } // EndFor
    Message($"cnt_zero: {cnt_zero}");
    Message($"cnt_one: {cnt_one}");
}
```

큐비트 할당

Lookup에 저장된 입/출력을 위한 큐비트 길이를 큐비트 할당한 것에 저장 후, QLUT 적용 (함수 호출)

======= 여기 아래는 그냥 아웃풋 보려고 만들었습니다. =======

QLUT 실제 코드 – QLUT (2)

```
// LUT
function ApplyFunctionWithLookupTable(func: Double -> Double, domain: (Double, Double), epsIn: Double, epsOut: Double): FunctionWithLookupTable {

    let (minIn, maxIn) = domain;

    let pLower = BitSizeI(Ceiling(AbsD(minIn)));
    let pUpper = BitSizeI(Ceiling(AbsD(maxIn)));
    let pIn = MaxI(pLower, pUpper) + 1; // The +1 is for the sign bit

    let qIn = Ceiling(Lg(1.0/epsIn));
    let qOut = Ceiling(Lg(1.0/epsOut));

    // Compute approximations of minIn and maxIn
    let minInFxP = DoubleAsFixedPoint(pIn, qIn, minIn);
    let maxInFxP = DoubleAsFixedPoint(pIn, qIn, maxIn);

    // Compute number of values in between minIn and maxIn
    let deltaIn = 1.0/PowD(2.0, IntAsDouble(qIn));
    let numValues = Truncate((maxInFxP - minInFxP) / deltaIn) + 1;

    mutable outValues = [0.0, size=numValues]; // List to store all the output values (initialized at all 0s)
    mutable inValueFxP = minInFxP; // Starting input value
    mutable pOut = 0; // Set initial pOut value which will be updated in loop below
    for i in 0..numValues-1 {
        // First a quick check to see that the enumaration is going correctly, i.e. that we are hitting all the values in order
        let inAddress = BoolArrayAsInt(FixedPointAsBoolArray(pIn, qIn, inValueFxP - minInFxP));
        EqualityFactI(inAddress, i, $"Unexpected address in enumeration");

        // Now we compute the output value, compute the number of integer bits it has and see if it is bigger than our current pOut
        let outValue = func(inValueFxP);
        set outValues w/= i <- outValue; // this is the syntax to say "outValues = outValues but with the ith index as outValue"
        set pOut = MaxI(pOut, BitSizeI(Ceiling(AbsD(outValue)))+1); //the +1 is for the sign bit
        set inValueFxP += deltaIn;
    }
    let outValuesFxP = Mapped(DoubleAsFixedPoint(pOut, qOut, _), outValues);
    let outBits = Mapped(FixedPointAsBoolArray(pOut, qOut, _), outValues);
    let lookupOperation = LookupOperationWrapper(minInFxP, outBits, _, _);

    return FunctionWithLookupTable(
        pIn, qIn,
        pOut, qOut,
        lookupOperation
    );
}
```

입력 인자들 처리
오차범위 허용 위함

OutValues에 모든 아웃풋이 list형태로 저장
FP→ Bool으로 mapping된 값이 outBits에 저장
이를 활용하여 최종 반환 값은 lookupOperation

리턴하는 요소들
(정수 및 소수 부분을 위한 큐비트 길이 반환, LUT 적용한 output)

QLUT 실제 코드-결과

- 해당 결과는 input qubit을 5개로, output qubit를 9개로 설정한 결과임
- 이보다 많은 큐비트 할당 시 연산 불가 (in MS Azure)

- 각 인덱스에는 아래와 같은 값들이 들어있음
  - 0으로 측정된 것은 8번, 1로 측정된 것은 1번
    - Output이 0인 것은 8개, 1인 것은 1개

```
%simulate Main
```

✓

```
lookup::IntegerBitsIn: 2
lookup::FractionalBitsIn: 3
lookup::IntegerBitsOut: 2
lookup::FractionalBitsOut: 7
idx: 0
[output[idx]]: [q:5]
idx: 1
[output[idx]]: [q:6]
idx: 2
[output[idx]]: [q:7]
idx: 3
[output[idx]]: [q:8]
idx: 4
[output[idx]]: [q:9]
idx: 5
[output[idx]]: [q:10]
idx: 6
[output[idx]]: [q:11]
idx: 7
[output[idx]]: [q:12]
idx: 8
[output[idx]]: [q:13]
cnt_zero: 8
cnt_one: 1
```

QLUT 관련 논문 작성

- 알고보니 구현은 1저자가 다 해놔서 저는 중간에 논문 작성하는 쪽으로 빠졌습니다
- 2/29까지였어서 제 부분은 다 작성..
- 아누팜한테 연락이 안 와서
  끝난 프로젝트인지는 아직 모르겠습니다.

- **QLUT을 사용하여 복잡한 활성화함수 계산을 효율적으로 수행**

Efficient Quantum Circuits for Machine Learning Activation Functions including Constant T-depth ReLU

March 2, 2024

1 Introduction

Currently, machine learning is attracting a lot of attention as it is actively used in various industries. There are various models such as natural language processing models (NLP) such as GPT4, which are very popular recently, and image classification or generation. An important building block of machine learning algorithms are neural networks. Neural networks are collections of layers of activation functions and linear transformations. Many different activation functions have been discussed, such as Sigmoid, Softmax, and ReLU. In contrast to smooth functions such as Sigmoid, ReLU has some advantages (sparsity and fast calculation speed).

Quantum machine learning has shown progress to both adapt common machine learning algorithms to run on future quantum computers, and to use machine learning techniques to learn about quantum systems. Some examples include Quantum Neural Network (QNN), quantum support vector machine (QSVM), quantum principal component analysis (QPCA), variational quantum eigensolver (VQE) and parameterized quantum circuits (PQC), quantum approximate optimization algorithm (QAOA). As an example, quantum convolutional networks provide quantum circuits for the convolutional layers [1]. As such, some quantum neural network algorithms may require an activation function like classical neural networks. Quantum signal processing and QSVT quantum singular value transformation, which enacts function via polynomial transformations. However, polynomial approximations are in some cases of high degree, such as in the case of the ReLU [2]. Hence, it makes sense to study the arithmetic evaluation of the ReLU function and design quantum circuits for it.

In this work, we focus on the optimization of T-depth of quantum circuit. We proposed a quantum circuit whose T-depth is 4 to implement the ReLU function, even if the qubit connectivity is a constraint in a 2D grid. We also implement Leaky ReLU functions by a quantum circuit with T-depth of 8. For the activation functions that are not convenient to implement by arithmetic circuit, such as: Sigmoid, SoftMax, Tanh, Swish, Exponential Linear Unit (ELU), Gaussian Error Linear Unit (GELU) [2]. We implement these functions by using the quantum Lookup table (QLUT). We only could approximately implement these functions as their input and output is real number, it is important to trade off the number of qubits to represent the result and the accuracy of our implementation [3, 4]. We use the float number coded by $n \in \{8, 16, 32, 64, 128\}$ bits as input and output of these functions. We also show using different number of ancilla to reduce the T-depth of quantum circuit. The T-depth of quantum circuit that implement Sigmoid, SoftMax, Tanh, Swish,

감사합니다.