PRE week6 _ TraceCipherText

날짜	@2023년 8월 10일

▼ **week6 task**

- OpenFHE 코드에서 TraceCipherText class를 추가적으로 만들어서 scale 및 decryption vector 추적
  ex. Cipertext ct
  ct.showDetail

> scale : ~~
> decryption : ~~
> original : ~~
> 암호화한 시점에 보여주고, 연산할수록 얼마나 차이가 나는지 확인할 수 있도록 한다. + 오차도 보여주면 좋다.

▼ **TraceCipherText 클래스**

암호문 연산 시 scale, original value, decryption value 등을 출력해주는 새로운 타입의 class

```
class TraceCipherText {
private:
    std::vector<double> original;  //Plaintext로 받았더니 packing되지 않았다는 오류가 나서 벡터로 받았습니다.
    Ciphertext<DCRTPoly> cipher;
    CryptoContext<DCRTPoly> cc;
    PrivateKey<DCRTPoly> secretKey;

public:
    TraceCipherText(std::vector<double> original, const Ciphertext<DCRTPoly> &cipher, CryptoContext<DCRTPoly> cc, const PrivateKey
        : original(original), cipher(cipher), cc(cc), secretKey(secretKey) {}

    void ShowDetail() {
        std::cout << " =========== Show Detail ===========" << std::endl;
        double scale = cipher->GetScalingFactor();

        Plaintext result;
        cc->Decrypt(cipher, secretKey, &result);

        std::cout << "   +  Scale: " << log2(scale) << std::endl;
        std::cout << "   +  Decrypted Result: " << result << std::endl;

        std::cout << "   +  Original : ";
        for (auto i : original) {
            std::cout << i << ", ";
        }
        std::cout << std::endl;
    }

    void Error(){

    }

    TraceCipherText tradd(const TraceCipherText &other) {
        std::cout << " =========== Add =========== " << std::endl;

        auto resultCipher = cc -> EvalAdd(cipher,other.cipher); //암호문끼리 덧셈 후 resultCipher에 저장

        double scale = resultCipher -> GetScalingFactor(); //resultCipher의 scale

        Plaintext add_result; //덧셈결과의 plaintext 타입
        cc->Decrypt(resultCipher, secretKey, &add_result); //암호문 resultCipher을 복호화해서 평문 add_result에 저장
        std::cout << "   +  덧셈 후 Scale  : " << log2(scale) << std::endl;
        std::cout << "   +  Computed Result  : " << add_result << std::endl;

        //암호화하지 않고 계산했을 때 나와야 하는 값
        std::vector<double> result_vector(original.size(),0);
        std::cout << "   +  Expected result  : ";
        for (size_t i = 0; i < original.size(); ++i) {
            result_vector[i] = original[i] + other.original[i];
            std::cout << result_vector[i] << ", ";
        }
        std::cout << std::endl;

        return TraceCipherText(result_vector, resultCipher, cc, secretKey); //암호화된 덧셈결과 반환
    }

    TraceCipherText trmult(const TraceCipherText &other) {
        std::cout << " =========== Multiply =========== " << std::endl;

        auto resultCipher = cc -> EvalMult(cipher,other.cipher); //암호문끼리 덧셈 후 resultCipher에 저장

        double scale = resultCipher -> GetScalingFactor(); //resultCipher의 scale

        Plaintext mult_result; //곱셈결과의 plaintext 타입
        cc->Decrypt(resultCipher, secretKey, &mult_result); //암호문 resultCipher을 복호화해서 평문 add_result에 저장
        std::cout << "   +  곱셈 후 Scale  : " << log2(scale) << std::endl;
        std::cout << "   +  Computed Result  : " << mult_result << std::endl;

        //암호화하지 않고 계산했을 때 나와야 하는 값
        std::vector<double> result_vector(original.size(),0);
        std::cout << "   +  Expected result : ";
        for (size_t i = 0; i < original.size(); ++i) {
            result_vector[i] = original[i] * other.original[i];
            std::cout << result_vector[i] << ", ";
        }
        std::cout << std::endl;

        return TraceCipherText(result_vector, resultCipher, cc, secretKey); //암호화된 곱셈결과 반환
    }
};
```

- **TraceCipherText 클래스의 멤버변수들**
    - `std::vector<double> original` : 암호화되지 않은 형태의 실수형 벡터. 평문을 Plaintext로 받으려 했으나 packing되지 않았다는 오류가 나서 벡터로 받았습니다.
    - `Ciphertext<DCRTPoly> cipher` : 암호화된 Ciphertext
    - `CryptoContext<DCRTPoly> cc` : 암호 컨텍스트를 설정하고 생성하는 역할. 암호화, 복호화, 키 생성 등 다양한 암호 연산 수행
    - `PrivateKey<DCRTPoly> secretKey` : 암호문 decription에 필요한 비밀키

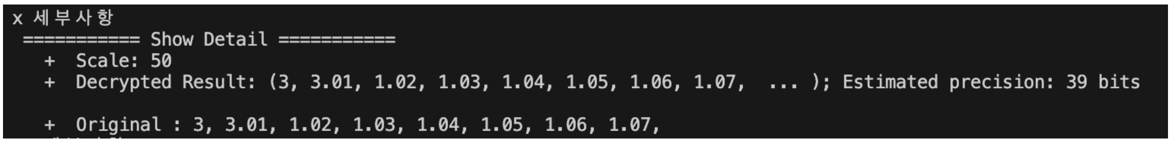
- **`ShowDetail` 함수**

TraceCipherText 타입으로 선언된 변수에 대해서,

scale, decrypted result, original value 등을 보여줍니다.

📃 실행예시

```
TraceCipherText ct1(x, c, cc, keys.secretKey);
std::cout << "x 세부사항" << std::endl;
ct1.ShowDetail();
```



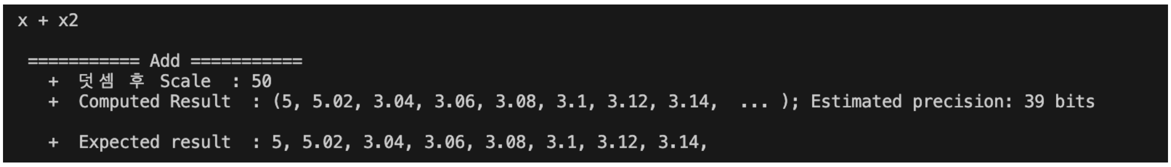
```
x 세부사항
 =========== Show Detail ===========
   +  Scale: 50
   +  Decrypted Result: (3, 3.01, 1.02, 1.03, 1.04, 1.05, 1.06, 1.07,  ... ); Estimated precision: 39 bits

   +  Original : 3, 3.01, 1.02, 1.03, 1.04, 1.05, 1.06, 1.07,
```

- **`tradd` 함수**

두 TraceCipherText 타입 변수의 덧셈 지원, 덧셈 이후의 scale, 계산 결과 및 Expected value 등을 출력

📃 실행예시

```
TraceCipherText ct1(x, c, cc, keys.secretKey);
TraceCipherText ct2(x2, c2, cc, keys.secretKey);

std::cout << "\nx + x2\n" << std::endl;
TraceCipherText add_ct1_ct2 = ct1.tradd(ct2);
```



```
x + x2

 =========== Add ===========
   +  덧셈 후 Scale  : 50
   +  Computed Result  : (5, 5.02, 3.04, 3.06, 3.08, 3.1, 3.12, 3.14,  ... ); Estimated precision: 39 bits

   +  Expected result  : 5, 5.02, 3.04, 3.06, 3.08, 3.1, 3.12, 3.14,
```

👉 tradd는 덧셈 결과를 TraceCipherText 타입으로 반환하므로 덧셈 결과를 새로 정의한 TraceCipherText 변수에 할당할 수 있습니다.

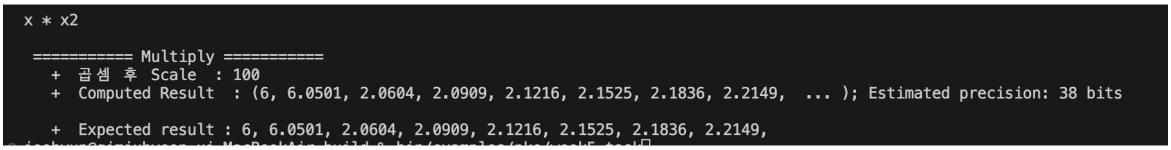
- 📌 `trmult` **함수**

두 TraceCipherText 타입 변수의 곱셈 지원

곱셈 이후의 scale, 계산 결과 및 Expected value 등을 출력

📃 실행예시

```
std::cout << "\nx * x2\n" << std::endl;
TraceCipherText mult_ct1_ct2 = ct1.trmult(ct2);
```



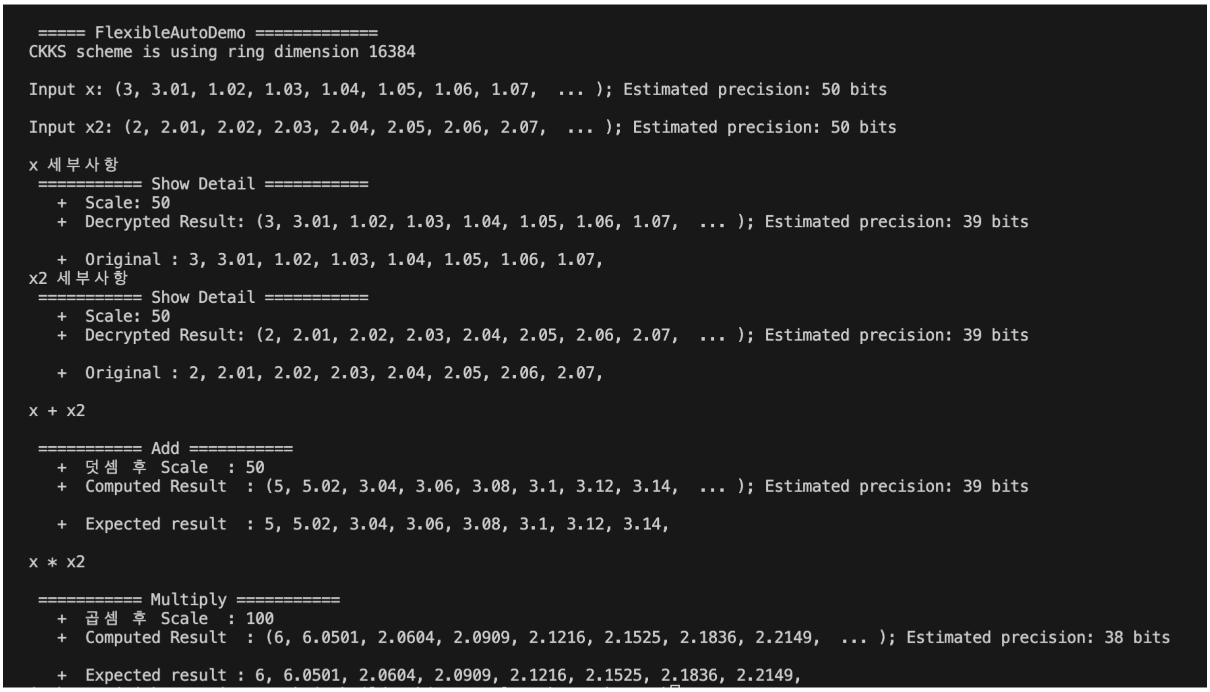
```
x * x2

 =========== Multiply ===========
   +  곱셈 후 Scale  : 100
   +  Computed Result  : (6, 6.0501, 2.0604, 2.0909, 2.1216, 2.1525, 2.1836, 2.2149,  ... ); Estimated precision: 38 bits

   +  Expected result : 6, 6.0501, 2.0604, 2.0909, 2.1216, 2.1525, 2.1836, 2.2149,
```

👉 마찬가지로 multadd는 곱셈 결과를 TraceCipherText 타입으로 반환하므로 곱셈 결과를 새로 정의한 TraceCipherText 변수에 할당할 수 있으며, showDetail 함수를 따로 실행하지 않아도 곱셈과 동시에 세부사항을 출력합니다.

📌 전체 코드

```
#define PROFILE

#include "openfhe.h"

using namespace lbcrypto;

void AutomaticRescaleDemo(ScalingTechnique scalTech);


class TraceCipherText {
private:
    std::vector<double> original;  //Plaintext로 받았더니 packing되지 않았다는 오류가 나서 벡터로 받았습니다.
    Ciphertext<DCRTPoly> cipher;
    CryptoContext<DCRTPoly> cc;
    PrivateKey<DCRTPoly> secretKey;

public:
    TraceCipherText(std::vector<double> original, const Ciphertext<DCRTPoly> &cipher, CryptoContext<DCRTPoly> cc, const PrivateKey<DCR
        : original(original), cipher(cipher), cc(cc), secretKey(secretKey) {}

    void ShowDetail() {
        std::cout << " =========== Show Detail ===========" << std::endl;
        double scale = cipher->GetScalingFactor();
        Plaintext result;
        cc->Decrypt(cipher, secretKey, &result);

        std::cout << "   +  Scale: " << log2(scale) << std::endl;
        std::cout << "   +  Decrypted Result: " << result << std::endl;

        std::cout << "   +  Original : ";
        for (auto i : original) {
            std::cout << i << ", ";
        }
        std::cout << std::endl;
    }

    void Error(){

    }

    TraceCipherText tradd(const TraceCipherText &other) {
        std::cout << " =========== Add =========== " << std::endl;

        auto resultCipher = cc -> EvalAdd(cipher,other.cipher); //암호문끼리 덧셈 후 resultCipher에 저장

        double scale = resultCipher -> GetScalingFactor(); //resultCipher의 scale

        Plaintext add_result; //덧셈결과의 plaintext 타입
        cc->Decrypt(resultCipher, secretKey, &add_result); //암호문 resultCipher을 복호화해서 평문 add_result에 저장
        std::cout << "   +  덧셈 후 Scale  : " << log2(scale) << std::endl;
        std::cout << "   +  Computed Result  : " << add_result << std::endl;

        //암호화하지 않고 계산했을 때 나와야 하는 값
        std::vector<double> result_vector(original.size(),0);
        std::cout << "   +  Expected result  : ";
        for (size_t i = 0; i < original.size(); ++i) {
            result_vector[i] = original[i] + other.original[i];
            std::cout << result_vector[i] << ", ";
        }
        std::cout << std::endl;

        return TraceCipherText(result_vector, resultCipher, cc, secretKey); //암호화된 덧셈결과 반환
    }

    TraceCipherText trmult(const TraceCipherText &other) {
        std::cout << " =========== Multiply =========== " << std::endl;

        auto resultCipher = cc -> EvalMult(cipher,other.cipher); //암호문끼리 덧셈 후 resultCipher에 저장

        double scale = resultCipher -> GetScalingFactor(); //resultCipher의 scale

        Plaintext mult_result; //곱셈결과의 plaintext 타입
        cc->Decrypt(resultCipher, secretKey, &mult_result); //암호문 resultCipher을 복호화해서 평문 add_result에 저장
        std::cout << "   +  곱셈 후 Scale  : " << log2(scale) << std::endl;
        std::cout << "   +  Computed Result  : " << mult_result << std::endl;

        //암호화하지 않고 계산했을 때 나와야 하는 값
        std::vector<double> result_vector(original.size(),0);
        std::cout << "   +  Expected result : ";
        for (size_t i = 0; i < original.size(); ++i) {
            result_vector[i] = original[i] * other.original[i];
            std::cout << result_vector[i] << ", ";
        }
        std::cout << std::endl;

        return TraceCipherText(result_vector, resultCipher, cc, secretKey); //암호화된 곱셈결과 반환
    }
};


int main(int argc, char* argv[]) {

    AutomaticRescaleDemo(FLEXIBLEAUTO);


    return 0;
}

void AutomaticRescaleDemo(ScalingTechnique scalTech) {

    if (scalTech == FLEXIBLEAUTO) {
        std::cout << std::endl << std::endl << std::endl << " ===== FlexibleAutoDemo ============= " << std::endl;
    }
    else {
        std::cout << std::endl << std::endl << std::endl << " ===== FixedAutoDemo ============= " << std::endl;
    }

    uint32_t batchSize = 8;
    CCParams<CryptoContextCKKSRNS> parameters;
    parameters.SetMultiplicativeDepth(2);
    parameters.SetScalingModSize(50);
    parameters.SetScalingTechnique(scalTech);
    parameters.SetBatchSize(batchSize);

    CryptoContext<DCRTPoly> cc = GenCryptoContext(parameters); //암호 컨텍스트를 설정하고 생성하는 역할. 암호화, 복호화, 키 생성 등 다양한 암호 연산 수

    std::cout << "CKKS scheme is using ring dimension " << cc->GetRingDimension() << std::endl << std::endl;

    cc->Enable(PKE);
    cc->Enable(KEYSWITCH);
    cc->Enable(LEVELEDSHE);

    auto keys = cc->KeyGen();
    cc->EvalMultKeyGen(keys.secretKey);

    // Input
    std::vector<double> x = {3.0, 3.01, 1.02, 1.03, 1.04, 1.05, 1.06, 1.07};
    std::vector<double> x2 = {2.0, 2.01, 2.02, 2.03, 2.04, 2.05, 2.06, 2.07};

    Plaintext ptxt        = cc->MakeCKKSPackedPlaintext(x);
    Plaintext ptxt2        = cc->MakeCKKSPackedPlaintext(x2);

    std::cout << "Input x: " << ptxt << std::endl;
    std::cout << "Input x2: " << ptxt2 << std::endl;

    auto c = cc->Encrypt(ptxt, keys.publicKey);
    auto c2 = cc->Encrypt(ptxt2, keys.publicKey);

    TraceCipherText ct1(x, c, cc, keys.secretKey);
    TraceCipherText ct2(x2, c2, cc, keys.secretKey);


    std::cout << "x 세부사항" << std::endl;

    ct1.ShowDetail();

    std::cout << "x2 세부사항" << std::endl;
    ct2.ShowDetail();

    std::cout << "\nx + x2\n" << std::endl;
    TraceCipherText add_ct1_ct2 = ct1.tradd(ct2);

    std::cout << "\nx * x2\n" << std::endl;
    TraceCipherText mult_ct1_ct2 = ct1.trmult(ct2);



}
```

📌 실행 결과



```
 ===== FlexibleAutoDemo =============
CKKS scheme is using ring dimension 16384

Input x: (3, 3.01, 1.02, 1.03, 1.04, 1.05, 1.06, 1.07,  ... ); Estimated precision: 50 bits

Input x2: (2, 2.01, 2.02, 2.03, 2.04, 2.05, 2.06, 2.07,  ... ); Estimated precision: 50 bits

x 세부사항
 =========== Show Detail ===========
   +  Scale: 50
   +  Decrypted Result: (3, 3.01, 1.02, 1.03, 1.04, 1.05, 1.06, 1.07,  ... ); Estimated precision: 39 bits

   +  Original : 3, 3.01, 1.02, 1.03, 1.04, 1.05, 1.06, 1.07,
x2 세부사항
 =========== Show Detail ===========
   +  Scale: 50
   +  Decrypted Result: (2, 2.01, 2.02, 2.03, 2.04, 2.05, 2.06, 2.07,  ... ); Estimated precision: 39 bits

   +  Original : 2, 2.01, 2.02, 2.03, 2.04, 2.05, 2.06, 2.07,

x + x2

 =========== Add ===========
   +  덧셈 후 Scale  : 50
   +  Computed Result  : (5, 5.02, 3.04, 3.06, 3.08, 3.1, 3.12, 3.14,  ... ); Estimated precision: 39 bits

   +  Expected result  : 5, 5.02, 3.04, 3.06, 3.08, 3.1, 3.12, 3.14,

x * x2

 =========== Multiply ===========
   +  곱셈 후 Scale  : 100
   +  Computed Result  : (6, 6.0501, 2.0604, 2.0909, 2.1216, 2.1525, 2.1836, 2.2149,  ... ); Estimated precision: 38 bits

   +  Expected result : 6, 6.0501, 2.0604, 2.0909, 2.1216, 2.1525, 2.1836, 2.2149,
```