CKKS Computation(N, Q, q)

(SEAL/native/examples/5_ckks_basics.cpp)

N, Q, q Setting

```
size_t poly_modulus_degree = 8192;
parms.set_poly_modulus_degree(poly_modulus_degree);
parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 60, 40, 40, 60 }));
```

Special prime

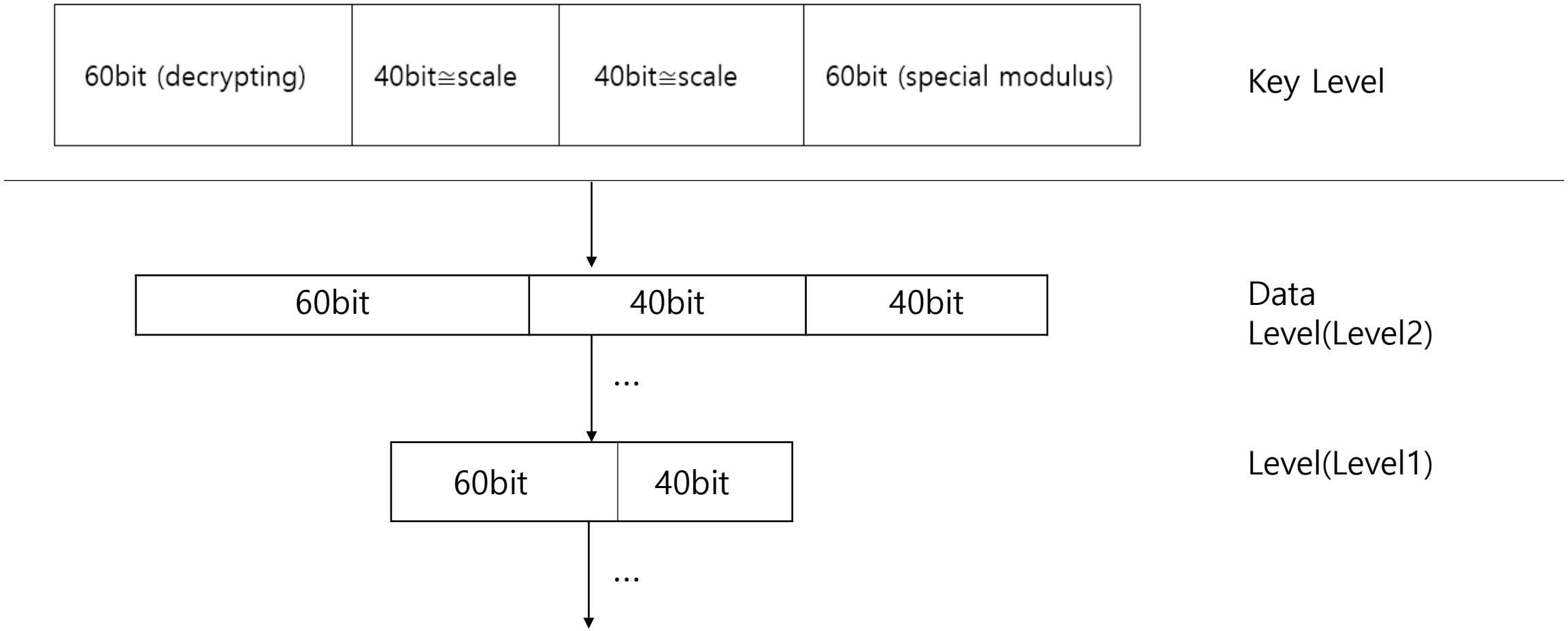
Relationship between poly_modulus(N) and max coeff_modulus(Q)

poly_modulus_degree	max coeff_modulus bit-length
1024	27
2048	54
4096	109
8192	218
16384	438
32768	881

Q : sum of the bit-lengths of its prime factors

N이 증가 -> max coeff_modulus bit-length가 증가 -> 더 많은 곱 연산 수행 가능

하지만, 연산 시 Runtime이 늘어나는 단점


60bit (decrypting)
40bit≅scale
40bit≅scale
60bit (special modulus)
Key Level
60bit
40bit
40bit
Data
Level(Level2)
...
60bit
40bit
Level(Level1)
...

Encoding

```
double scale = pow(2.0, 40);
```

```
CKKSEncoder encoder(context);
size_t slot_count = encoder.slot_count();
cout << "Number of slots: " << slot_count << endl;

vector<double> input;
input.reserve(slot_count);
double curr_point = 0;
double step_size = 1.0 / (static_cast<double>(slot_count) - 1);
for (size_t i = 0; i < slot_count; i++)
{
    input.push_back(curr_point);
    curr_point += step_size;
}
```

Slot count는 $\frac{N}{2}$으로 만들어 진다.

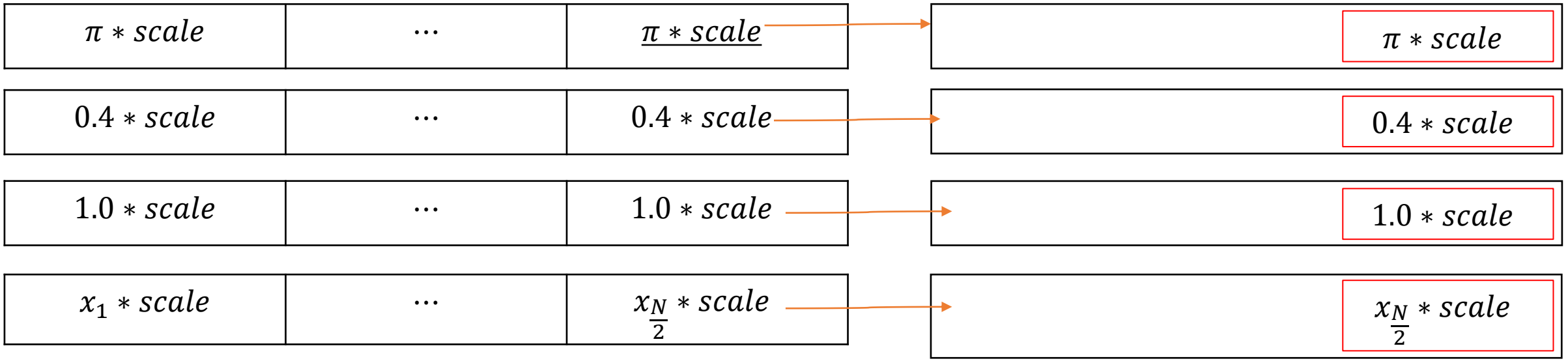
x_1	x_2	x_3	⋯	$x_{\frac{N}{2}}$

Input vector

```
Plaintext plain_coeff3, plain_coeff1, plain_coeff0;
encoder.encode(3.14159265, scale, plain_coeff3);
encoder.encode(0.4, scale, plain_coeff1);
encoder.encode(1.0, scale, plain_coeff0);

Plaintext x_plain;
print_line(__LINE__);
cout << "Encode input vectors." << endl;
encoder.encode(input, scale, x_plain);
Ciphertext x1_encrypted;
encryptor.encrypt(x_plain, x1_encrypted);
```

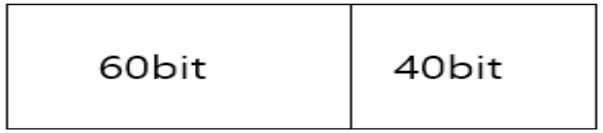
Implement polynomial
PI*x^3 + 0.4x + 1

logQ bits

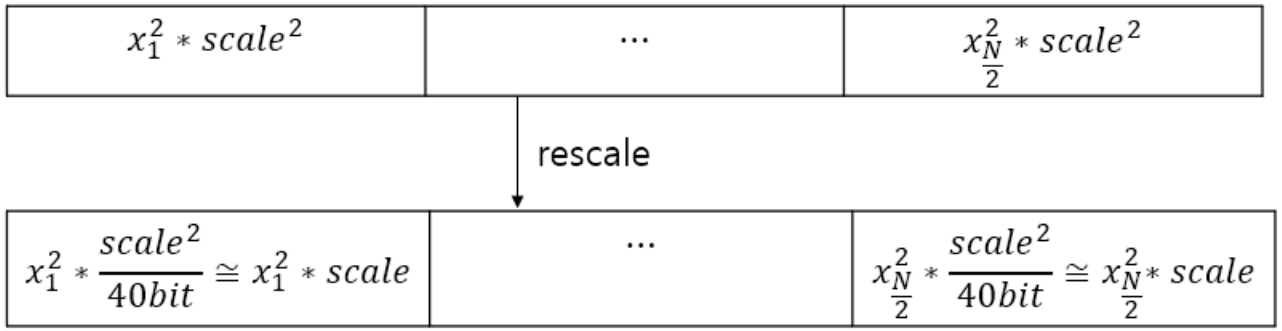


# of input slots

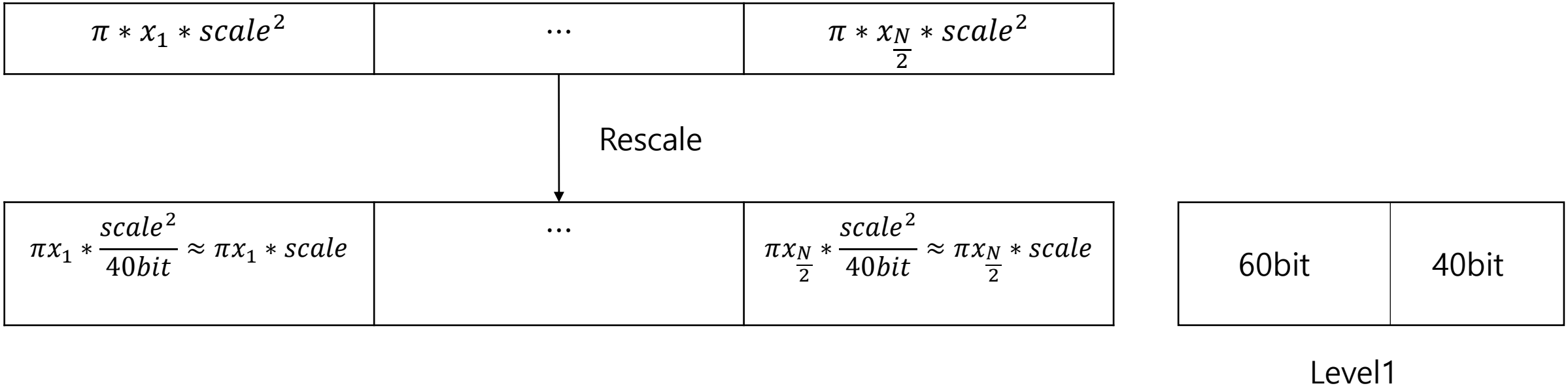
```
Ciphertext x3_encrypted;
print_line(__LINE__);
cout << "Compute x^2 and relinearize:" << endl;
evaluator.square(x1_encrypted, x3_encrypted);
evaluator.relinearize_inplace(x3_encrypted, relin_keys);
cout << "    + Scale of x^2 before rescale: " << log2(x3_encrypted.scale()) << " bits" << endl;

/*
Now rescale; in addition to a modulus switch, the scale is reduced down by
a factor equal to the prime that was switched away (40-bit prime). Hence, the
new scale should be close to 2^40. Note, however, that the scale is not equal
to 2^40: this is because the 40-bit prime is only close to 2^40.
*/
print_line(__LINE__);
cout << "Rescale x^2." << endl;
evaluator.rescale_to_next_inplace(x3_encrypted);
cout << "    + Scale of x^2 after rescale: " << log2(x3_encrypted.scale()) << " bits" << endl;
```



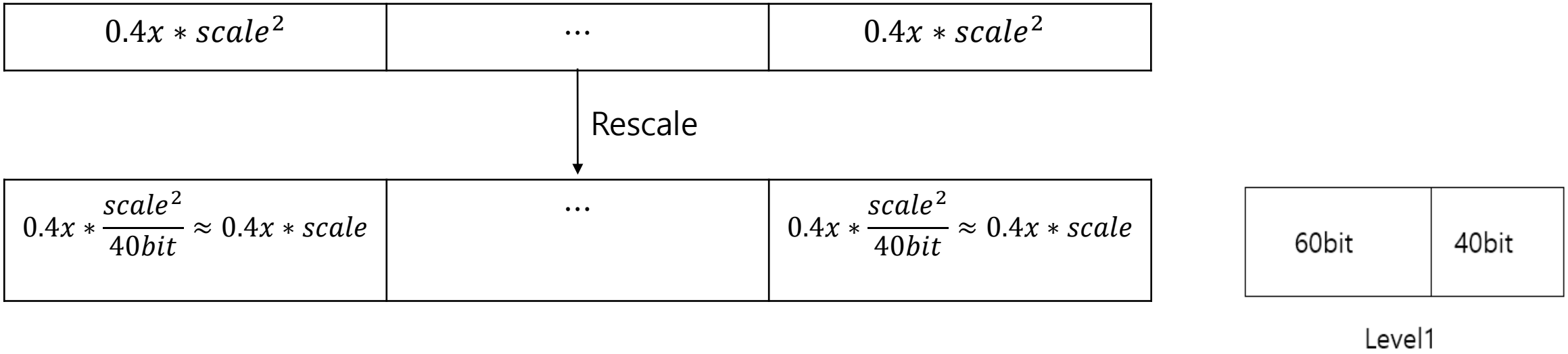
Level1



```
print_line(__LINE__);
cout << "Compute and rescale PI*x." << endl;
Ciphertext x1_encrypted_coeff3;
evaluator.multiply_plain(x1_encrypted, plain_coeff3, x1_encrypted_coeff3);
cout << "    + Scale of PI*x before rescale: " << log2(x1_encrypted_coeff3.scale()) << " bits" << endl;
evaluator.rescale_to_next_inplace(x1_encrypted_coeff3);
cout << "    + Scale of PI*x after rescale: " << log2(x1_encrypted_coeff3.scale()) << " bits" << endl;
```



```
print_line(__LINE__);
cout << "Compute and rescale 0.4*x." << endl;
evaluator.multiply_plain_inplace(x1_encrypted, plain_coeff1);
cout << "    + Scale of 0.4*x before rescale: " << log2(x1_encrypted.scale()) << " bits" << endl;
evaluator.rescale_to_next_inplace(x1_encrypted);
cout << "    + Scale of 0.4*x after rescale: " << log2(x1_encrypted.scale()) << " bits" << endl;
```



Level1

```
print_line(__LINE__);
cout << "Compute, relinearize, and rescale (PI*x)*x^2." << endl;
evaluator.multiply_inplace(x3_encrypted, x1_encrypted_coeff3);
evaluator.relinearize_inplace(x3_encrypted, relin_keys);
cout << "    + Scale of PI*x^3 before rescale: " << log2(x3_encrypted.scale()) << " bits" << endl;
evaluator.rescale_to_next_inplace(x3_encrypted);
cout << "    + Scale of PI*x^3 after rescale: " << log2(x3_encrypted.scale()) << " bits" << endl;
```

60 bit

Level 0

$\pi x_1 * scale$	⋯	$\pi x_{\frac{N}{2}} * scale$

X

$x_1^2 * scale$	⋯	$x_{\frac{N}{2}}^2 * scale$

↓

$\pi x_1^3 * scale^2$	⋯	$\pi x_{\frac{N}{2}}^3 * scale^2$

↓ Rescale

$\pi x_1^3 * \frac{scale^2}{40bit} \approx \pi x_1^3 * scale$	⋯	$\pi x_{\frac{N}{2}}^3 * \frac{scale^2}{40bit} \approx \pi x_{\frac{N}{2}}^3 * scale$

$\pi x_{\frac{N}{2}} * scale$

X

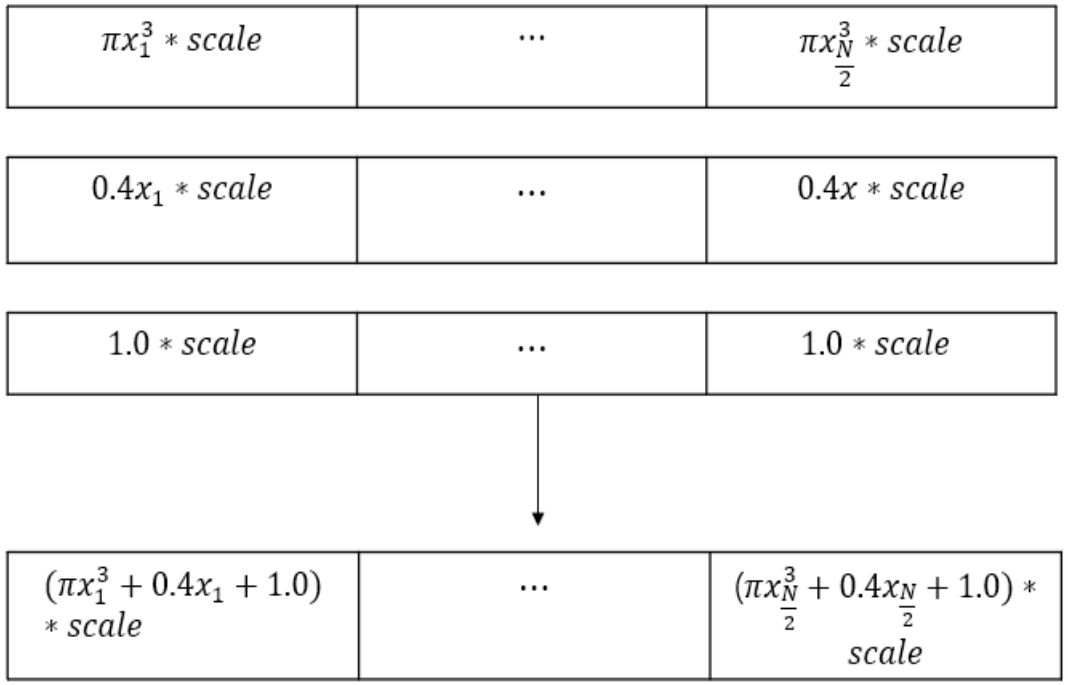
$x_{\frac{N}{2}}^2 * scale$

↓

$\pi x_{\frac{N}{2}}^3 * scale^2$

$\pi x_{\frac{N}{2}}^3 * scale$

```
print_line(__LINE__);
cout << "Normalize scales to 2^40." << endl;
x3_encrypted.scale() = pow(2.0, 40);
x1_encrypted.scale() = pow(2.0, 40);

/*
We still have a problem with mismatching encryption parameters. This is easy
to fix by using traditional modulus switching (no rescaling). CKKS supports
modulus switching just like the BFV scheme, allowing us to switch away parts
of the coefficient modulus when it is simply not needed.
*/
print_line(__LINE__);
cout << "Normalize encryption parameters to the lowest level." << endl;
parms_id_type last_parms_id = x3_encrypted.parms_id();
evaluator.mod_switch_to_inplace(x1_encrypted, last_parms_id);
evaluator.mod_switch_to_inplace(plain_coeff0, last_parms_id);

/*
All three ciphertexts are now compatible and can be added.
*/
print_line(__LINE__);
cout << "Compute PI*x^3 + 0.4*x + 1." << endl;
Ciphertext encrypted_result;
evaluator.add(x3_encrypted, x1_encrypted, encrypted_result);
evaluator.add_plain_inplace(encrypted_result, plain_coeff0);
```



각 cipher text마다 차지하는 Q가 서로 다르기 때문에 modular switch를 해준다.

또한, Add할 때 prime이 서로 다른 값이기 때문에, 오차를 줄이기 위해 동일한 scale로 맞춰야 한다.

```
decryptor.decrypt(encrypted_result, plain_result);

vector<double> result;

encoder.decode(plain_result, result);

cout << "    + Computed result ...... Correct." << endl;

print_vector(result, 3, 7);
```

$(\pi x_1^3 + \cdots) * scale/scale$	$\cdots$	$(\pi x_{\frac{N}{2}}^3 + \cdots) * scale/scale$