中山大學 网络空间安全学院

SUN YAT-SEN UNIVERSITY SCHOOL OF CYBER SCIENCE AND TECHNOLOGY

# 全同态加密

**学院：网络空间安全学院**

**专业：网络空间安全**

# 一、场景设置及实验环境

- 场景设置:

由于不同的开源全同态加密库主要差别是底层语言的不同(主要为 C++和 python),故不同开源库的效率差距主要体现在 C++语言和 python 语言之间的性能差距,C++应比 python 速度快。

本实验采用基于 python 的 tenSEAL 库,分别使用 CKKS 和 BFV 算法。

对于 5 张图片(本实验的设置为 5 张,也可以添加任意图片在 input_images 文件夹中,进行不同数量图片的测试),分别使用 tenSEAL 库的 CKKS 算法和 BFV 算法,对上述 5 张图片进行加密并存储到相应的文件夹中(密态存储)。用户使用一张查询图片进行查询,在将该查询图片用相应加密算法加密后,在密态下计算查询图片与存储图片之间的余弦相似度(计算结果越接近 0,代表两个图片越相似),若余弦相似度小于一个阈值,则可以认为匹配成功,在存储图片中取出该匹配成功的图片,并进行解密,恢复原图像。

本实验计算的运行时间为:

1.将图片用相应算法加密并存储到文件夹中的用时

2.将加密后的查询图片与存储图片进行匹配的用时

3.将匹配的存储图片解密的用时

- 环境记录:

| 序号 | | |
|---|---|---|
| 1 | 操作系统 | Windows |
| 2 | CPU 配置 | AMD Ryzen5 5500U(6 核心 2.10GHz) |
| 3 | 内存 | 16GB（3200 MHz） |

- 实现代码：

**使用 CKKS 算法和 BFV 算法创建不同的加密环境：**

```
# 创建context加密环境
context = ts.context(
    ts.SCHEME_TYPE.CKKS,
    poly_modulus_degree=8192,
    coeff_mod_bit_sizes=[60, 40, 40, 60]
)
context.global_scale = 2**40
```

CKKS 算法环境中：

poly_modulus_degree：影响了同态加密系统中多项式环的大小和容量，越大代表着越高的安全性，越高的数据容量，也会消耗更多的计算时间。

coeff_mod_bit_sizes：用于设置同态加密中多项式系数模数的比特大小

global_scale：用于控制加密数据的范围，值越大，精度越大，计算时间越久。（只在 CKKS 方案中有该参数）

```
public_context = ts.context(
    ts.SCHEME_TYPE.BFV,
    poly_modulus_degree=8192,
    plain_modulus=1032193
)
```

BFV 算法环境中：

poly_modulus_degree：同上，但该算法的该值最大为 8192

plain_modulus：决定了明文空间的大小，它是一个质数，限制了加密数据的取值范围。官方文档给出的示例为 1032193，我尝试取更大或更小的值，但都报错，可能是不支持，官方文档没有给出具体说明。这也导致了在匹配图片时，进行向量内积操作可能会超出这个范围，导致产生精度不高的结果，但匹配结果的正确性是能保证的。

读取灰度形式图片：

```
def image_read(image_path):
    # 由于算法的密文长度限制，用灰度形式读取图片
    img = Image.open(image_path).convert('L')
    # 将Image对象转换为一个Numpy数组，并展开成一维向量
    np_img = np.asarray(img).flatten()
    return np_img
```

加密图片并存储：

```python
# 定义函数将图片加密并保存
def encrypt_and_save_image(img_array, context, output_folder, index):
    # 加密图像
    start_time = time.time()
    enc_image = ts.bfv_vector(context, img_array)
    end_time = time.time()
    time1 = end_time - start_time
    time_list_encrypt.append(time1)
    print(f"encrypt NO.{index} image cost: {time1} seconds")

    # 保存加密数据到文件
    file_path = f"{output_folder}/data_{index}.bin"
    with open(file_path, 'wb') as file:
        file.write(enc_image.serialize())

    return file_path
```

解释：这里以灰度形式读取图片，是因为”poly_modulus_degree”参数限制的，由于 BFV 算法的该参数最大只能取到 8192，如果待加密的数据大于这个值，就会无法完全加密数据，从而影响后续操作。为了控制变量，更好地比较两个方案的运行时间，我们将两个方案的该参数都设置为 8192。（CKKS 方案该值可以取到更大，从而可以处理 RGB 图像，也提高了安全性，但牺牲了性能）

关键步骤：ts.ckks(bfv)_vector(context, img_array) -->加密

enc_image.serialize( ) -->序列化存储

在读取时应有相应的反序列化操作：ts.ckks_vector_from(context, encrypted_image_data)

计算相似度：

```python
# 计算相似度
result1 = enc_query_image.dot(enc_image)
dis_a = enc_query_image.dot(enc_query_image)
dis_b = enc_image.dot(enc_image)

distance = abs(result1._decrypt()[0]/math.sqrt(abs(dis_a._decrypt()[0])))
distance = 1-distance/math.sqrt(abs(dis_b.decrypt()[0]))
```

解释：result1 表示查询向量和加密向量的内积；dis_a 表示查询向量的模长的平方。相似度定义如下：

$$d(a,b) = 1 - \cos < a,b >$$

说明相似度越接近 0，两个向量（两个图片）越相似。

由于在计算两个向量的余弦值时涉及到密态下的除法，现提供两种解决思路：

1.在明文上做归一化再进行加密存储，这样在后续进行相似度计算时就只用计算向量内积。但由于 BFV 算法主要用于处理整数值，在明文上归一化会出现浮点数，故该思路有待商榷。
2.在计算相似度时，涉及除法的部分转明文进行计算。但这种方案会影响安全性。但在权衡下本实验决定使用该方法。

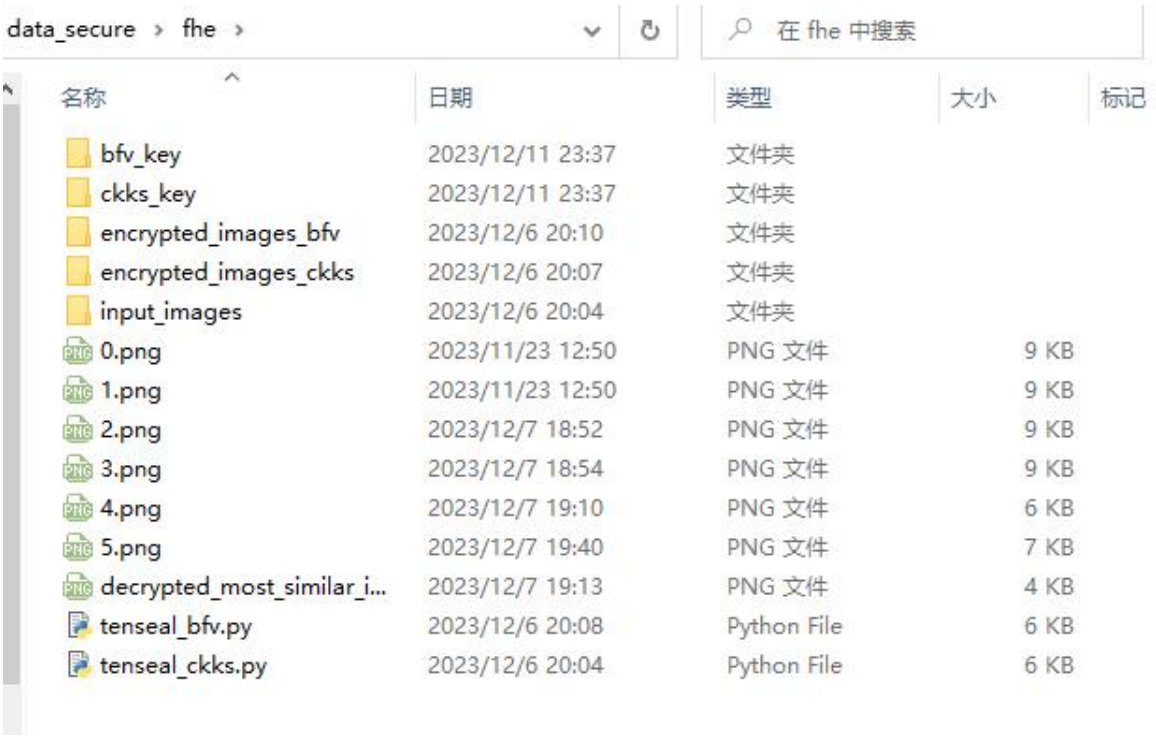
此外，由于 BFV 算法中 plain_modulus 参数的限制，在计算内积或模长时有可能会超出范围，经过算法处理后会出现导致精度不高的负数，因此使用了部分 abs() 函数。精度会缺失，但不影响匹配结果。（仅限于 BFV 算法，CKKS 算法不会出现）
计算完毕后，更新最大相似度和序号：

```
# 更新最大相似度和序号
if distance < max_similarity:
    max_similarity = distance
    max_similarity_index = index
```

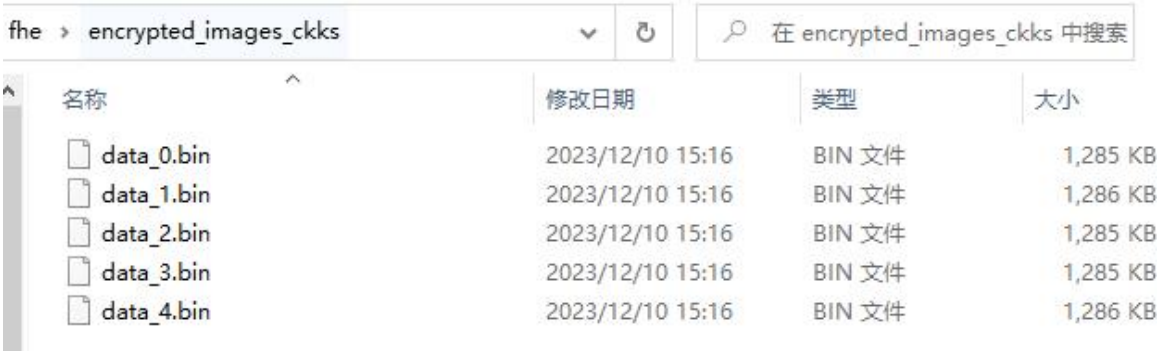
设置一个阈值，相似度小于该阈值则可以认为匹配成功，并从文件夹中读取相应图片密文进行解密：

```
print("-----------------decrypt image(if matching is successful)-----------
if(abs(max_similarity) < 10**-6):
    print("Match successfully!")
    print(f"The most similar image index: {max_similarity_index}, similarit

    # 从文件中读取并解密最相似的图片
    most_similar_image_path = f"{output_folder}/data_{max_similarity_index}

    with open(most_similar_image_path, 'rb') as file:
        encrypted_image_data = file.read()

    # 反序列化并解密
    enc_most_similar_image = ts.bfv_vector_from(public_context, encrypted_i
    start_time = time.time()
    decrypt_ = enc_most_similar_image.decrypt()
    end_time = time.time()
    time1 = end_time - start_time
    print(f"decrypt cost: {time1} seconds")
```

项目文件结构：

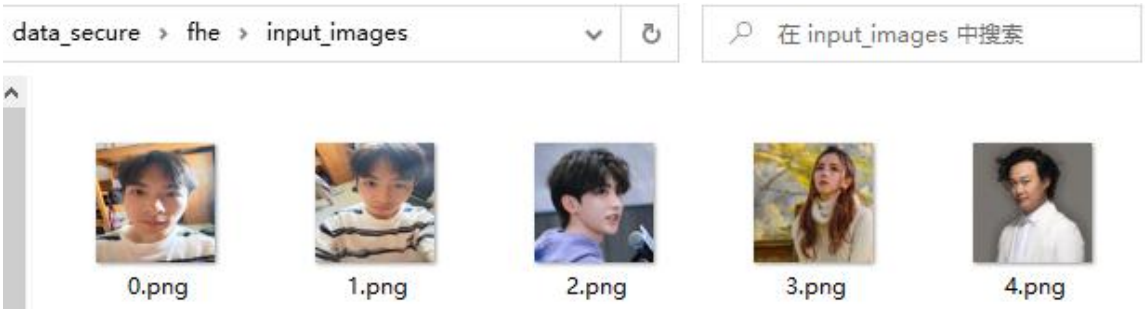



加密存储的文件夹：




分发密钥的文件夹：

待加密的图像：




实验结果中的查询图片示例为：4.png

# 二、GPU 加速 【加分项】

- 场景设置：

使用 https://github.com/vernamlab/cuFHE 项目进行 GPU 加速。

测试方法：

生成若干个随机数，先验证加密解密算法是否生效。若成功验证，则在使用 CPU 和 GPU 的条件下测试密态下的各种运算，并比较运行时间。

- 环境记录：

| 序号 | | |
|---|---|---|
| 1 | 操作系统 | Ubuntu 22.04 64 位 |
| 2 | CPU 配置 | Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz*32 |
| 3 | 内存 | 64G |
| 4 | GPU 配置 | 英伟达 GeForce RTX 3090 |

- 代码实现：

cpu：

验证加密解密：

```
cout<< "------ Test Encryption/Decryption ------" <<endl;
cout<< "Number of tests:\t" << kNumTests <<endl;
std::chrono::steady_clock::time_point begin2 = std::chrono::steady_clock:
correct = true;
for (int i = 0; i < kNumTests; i ++) {
  pt[0].message_ = rand() % Ptxt::kPtxtSpace;
  Encrypt(ct[0], pt[0], pri_key);
  Decrypt(pt[1], ct[0], pri_key);
  if (pt[1].message_ != pt[0].message_) {
    correct = false;
    break;
  }
}
if (correct)
  cout<< "PASS" <<endl;
else
  cout<< "FAIL" <<endl;
```

测试密态下的与非算法：

```
cout<< "------ Test NAND Gate ------" <<endl;
std::chrono::steady_clock::time_point begin1 = std::chrono::steady_clock:
kNumTests = 4;
cout<< "Number of tests:\t" << kNumTests <<endl;
correct = true;
for (int i = 0; i < kNumTests; i ++) {
  pt[0].message_ = rand() % Ptxt::kPtxtSpace;
  pt[1].message_ = rand() % Ptxt::kPtxtSpace;
  Encrypt(ct[0], pt[0], pri_key);
  Encrypt(ct[1], pt[1], pri_key);
  Nand(ct[0], ct[0], ct[1], pub_key);
  NandCheck(pt[1], pt[0], pt[1]);
  Decrypt(pt[0], ct[0], pri_key);
  if (pt[1].message_ != pt[0].message_) {
    correct = false;
    break;
  }
}
if (correct)
  cout<< "PASS" <<endl;
else
  cout<< "FAIL" <<endl;
```

gpu：

验证加密解密：

```
cout<< "------ Test Encryption/Decryption ------" <<endl;
cout<< "Number of tests:\t" << kNumTests <<endl;
std::chrono::steady_clock::time_point begin1 = std::chrono::steady_clock:
correct = true;
for (int i = 0; i < kNumTests; i ++) {
  pt[i].message_ = rand() % Ptxt::kPtxtSpace;
  Encrypt(ct[i], pt[i], pri_key);
  Decrypt(pt[kNumTests + i], ct[i], pri_key);
  if (pt[kNumTests + i].message_ != pt[i].message_) {
    correct = false;
    break;
  }
}
if (correct)
  cout<< "PASS" <<endl;
else
  cout<< "FAIL" <<endl;
```

在 GPU 上初始化数据，并行测试各种逻辑运算：

```
cout<< "------ Initilizating Data on GPU(s) ------" <<endl;
Initialize(pub_key); // essential for GPU computing

cout<< "------ Test NAND Gate ------" <<endl;
cout<< "Number of tests:\t" << kNumTests <<endl;
// Create CUDA streams for parallel gates.
Stream* st = new Stream[kNumSMs];
for (int i = 0; i < kNumSMs; i ++)
  st[i].Create();

correct = true;
for (int i = 0; i < 2 * kNumTests; i ++) {
  pt[i] = rand() % Ptxt::kPtxtSpace;
  Encrypt(ct[i], pt[i], pri_key);
}
Synchronize();

float et;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

// Here, pass streams to gates for parallel gates.
for (int i = 0; i < kNumTests; i ++)
  Nand(ct[i], ct[i], ct[i + kNumTests], st[i % kNumSMs]);
for (int i = 0; i < kNumTests; i ++)
  Or(ct[i], ct[i], ct[i + kNumTests], st[i % kNumSMs]);
for (int i = 0; i < kNumTests; i ++)
  And(ct[i], ct[i], ct[i + kNumTests], st[i % kNumSMs]);
```

其中，初始化和创建 CUDA 流是 GPU 加速的关键步骤。

# 三、实验结果分析

3.1 实验结果截图简要分析。

ckks 方案：

在相同的实验设置下运行三次，取平均值：

```
----------------encrypt images--------------------
encrypt NO.0 image cost: 0.016001462936401367 seconds
encrypt NO.1 image cost: 0.012015819549560547 seconds
encrypt NO.2 image cost: 0.012001276016235352 seconds
encrypt NO.3 image cost: 0.01202082633972168 seconds
encrypt NO.4 image cost: 0.012002944946289062 seconds
encrypt 5 images cost 0.06404232978820801 seconds
encrypt a single image cost: 0.012808465957641601 seconds on average
----------------matching images(close to 0 means similar)--------------------
No.0's similarity: 0.09543669559315959
No.1's similarity: 0.13506171847603854
No.2's similarity: 0.1574864810325547
No.3's similarity: 0.1823118878793435
No.4's similarity: 7.438494264988549e-15
compare cost: 1.307502269744873 seconds
compare a single image cost: 0.2615004539489746 seconds on average
----------------decrypt image(if matching is successful)--------------------
Match successfully!
The most similar image index: 4, similarity: 7.438494264988549e-15
decrypt cost: 0.004004716873168945 seconds
```

```
----------------encrypt images--------------------
encrypt NO.0 image cost: 0.015997886657714844 seconds
encrypt NO.1 image cost: 0.0160062313079834 seconds
encrypt NO.2 image cost: 0.01200556755065918 seconds
encrypt NO.3 image cost: 0.011749505996704102 seconds
encrypt NO.4 image cost: 0.016019582748413086 seconds
encrypt 5 images cost 0.07177877426147461 seconds
encrypt a single image cost: 0.014355754852294922 seconds on average
----------------matching images(close to 0 means similar)--------------------
No.0's similarity: 0.09543669559328516
No.1's similarity: 0.13506171847579196
No.2's similarity: 0.15748648103228147
No.3's similarity: 0.18231188787936192
No.4's similarity: 5.551115123125783e-16
compare cost: 1.297605276107788 seconds
compare a single image cost: 0.2595210552215576 seconds on average
----------------decrypt image(if matching is successful)--------------------
Match successfully!
The most similar image index: 4, similarity: 5.551115123125783e-16
decrypt cost: 0.004001617431640625 seconds
```

```
----------------encrypt images--------------------
encrypt NO.0 image cost: 0.016007423400878906 seconds
encrypt NO.1 image cost: 0.016000747680664062 seconds
encrypt NO.2 image cost: 0.012013673782348633 seconds
encrypt NO.3 image cost: 0.0160062313079834 seconds
encrypt NO.4 image cost: 0.016004323959350586 seconds
encrypt 5 images cost 0.07603240013122559 seconds
encrypt a single image cost: 0.015206480026245117 seconds on average
----------------matching images(close to 0 means similar)--------------------
No.0's similarity: 0.0954366955931284
No.1's similarity: 0.13506171847578674
No.2's similarity: 0.15748648103237473
No.3's similarity: 0.18231188787937036
No.4's similarity: -2.6645352591003757e-15
compare cost: 1.2957861423492432 seconds
compare a single image cost: 0.2591572284698486 seconds on average
----------------decrypt image(if matching is successful)--------------------
Match successfully!
The most similar image index: 4, similarity: -2.6645352591003757e-15
decrypt cost: 0.00439000129699707 seconds
```

以上实验结果展现了详细的运行过程。

- 加密过程：展示了每个图片的加密时间、总的加密时间以及平均加密每个图片的时间；
- 匹配过程：展示了匹配数据库中每张图片的相似度结果，以及总的匹配时间、平均匹配一张图片所用时间。
- 解密过程：如果匹配成功，就从数据库中取出相应的图片密文进行解密，展示了解密时间。

平均加密时间：0.014123566945393s
平均匹配时间：0.260059579213460s
平均解密时间：0.004132111867268s

其中，平均加密时间是指加密一个展平的（64*64）向量所用时间；平均匹配时间是指查询图片（向量）与数据库中的一张图片计算相似度所用时间；平均解密时间是将一张图片的密文解密成明文的时间。
有时会出现解密时间为 0.0s 的情况，本实验视此情况为异常情况，不作考虑。

BFV 方案：

在相同的实验设置下运行三次，取平均值：

```
-----------------encrypt images--------------------
encrypt NO.0 image cost: 0.011983156204223633 seconds
encrypt NO.1 image cost: 0.011997699737548828 seconds
encrypt NO.2 image cost: 0.011989593505859375 seconds
encrypt NO.3 image cost: 0.01600360870361328 seconds
encrypt NO.4 image cost: 0.012011051177978516 seconds
encrypt 5 images cost 0.06398510932922363 seconds
encrypt a single image cost: 0.012797021865844726 seconds on average
-----------------matching images(close to 0 means similar)---------------------
No.0's similarity: -0.8044614765207911
No.1's similarity: 0.41716053087522986
No.2's similarity: -4.044809515829555
No.3's similarity: -3.3733025945952697
No.4's similarity: 1.1102230246251565e-16
compare cost: 3.625804901123047 seconds
compare a single image cost: 0.7251609802246094 seconds on average
-----------------decrypt image(if matching is successful)--------------------
Match successfully!
The most similar image index: 4, similarity: 1.1102230246251565e-16
decrypt cost: 0.005150794982910156 seconds
```

```
-----------------encrypt images--------------------
encrypt NO.0 image cost: 0.012271881103515625 seconds
encrypt NO.1 image cost: 0.012004375457763672 seconds
encrypt NO.2 image cost: 0.011996746063232422 seconds
encrypt NO.3 image cost: 0.012351751327514648 seconds
encrypt NO.4 image cost: 0.01599574089050293 seconds
encrypt 5 images cost 0.0646204948425293 seconds
encrypt a single image cost: 0.012924098968505859 seconds on average
-----------------matching images(close to 0 means similar)---------------------
No.0's similarity: -0.8044614765207911
No.1's similarity: 0.41716053087522986
No.2's similarity: -4.044809515829555
No.3's similarity: -3.3733025945952697
No.4's similarity: 1.1102230246251565e-16
compare cost: 3.499870777130127 seconds
compare a single image cost: 0.6999741554260254 seconds on average
-----------------decrypt image(if matching is successful)--------------------
Match successfully!
The most similar image index: 4, similarity: 1.1102230246251565e-16
decrypt cost: 0.004002094268798828 seconds
```

```
-----------------encrypt images--------------------
encrypt NO.0 image cost: 0.011518239974975586 seconds
encrypt NO.1 image cost: 0.011994361877441406 seconds
encrypt NO.2 image cost: 0.012595891952514648 seconds
encrypt NO.3 image cost: 0.016243934631347656 seconds
encrypt NO.4 image cost: 0.01200413703918457 seconds
encrypt 5 images cost 0.06435656547546387 seconds
encrypt a single image cost: 0.012871313095092773 seconds on average
-----------------matching images(close to 0 means similar)--------------------
No.0's similarity: -0.8044614765207911
No.1's similarity: 0.41716053087522986
No.2's similarity: -4.044809515829555
No.3's similarity: -3.3733025945952697
No.4's similarity: 1.1102230246251565e-16
compare cost: 3.544625997543335 seconds
compare a single image cost: 0.708925199508667 seconds on average
-----------------decrypt image(if matching is successful)---------------------
Match successfully!
The most similar image index: 4, similarity: 1.1102230246251565e-16
decrypt cost: 0.0040013790130615234 seconds
```

平均加密时间：0.012864144643147s

平均匹配时间：0.711353445053100s

平均解密时间：0.004384756088256s

但 BFV 方案的解密时间有时候会出现异常波动，出现时间翻倍的情况：

```
-----------------encrypt images--------------------
encrypt NO.0 image cost: 0.012003183364868164 seconds
encrypt NO.1 image cost: 0.011977910995483398 seconds
encrypt NO.2 image cost: 0.016017913818359375 seconds
encrypt NO.3 image cost: 0.011994600296020508 seconds
encrypt NO.4 image cost: 0.012016534805297852 seconds
encrypt 5 images cost 0.0640101432800293 seconds
encrypt a single image cost: 0.012802028656005859 seconds on average
-----------------matching images(close to 0 means similar)--------------------
No.0's similarity: -0.8044614765207911
No.1's similarity: 0.41716053087522986
No.2's similarity: -4.044809515829555
No.3's similarity: -3.3733025945952697
No.4's similarity: 1.1102230246251565e-16
compare cost: 3.493642568588257 seconds
compare a single image cost: 0.6987285137176513 seconds on average
-----------------decrypt image(if matching is successful)---------------------
Match successfully!
The most similar image index: 4, similarity: 1.1102230246251565e-16
decrypt cost: 0.008213043212890625 seconds
```

由于出现次数较少，故不在考虑范围内。

总结分析：

| | 加密时间 | 匹配时间 | 解密时间 |
|---|---|---|---|
| CKKS 方案 | 0.014123566945393s | 0.260059579213460s | 0.004132111867268s |
| BFV 方案 | 0.012864144643147s | 0.711353445053100s | 0.004384756088256s |

设置相同的 poly_modulus_degree，并控制了一切可以控制的变量后，可以看到 CKKS 方案的加密时间略长但相差无几，但匹配时间相比 BFV 方案较快，解密时间相比 BFV 方案较快。

- CKKS 方案特点：

1.支持连续域数据：CKKS 是一种针对实数域的同态加密方案，主要用于对实数进行加密和同态计算。
2.支持更高精度计算：CKKS 方案允许在同态加密中对实数进行高精度的计算，能够处理更复杂的数学运算。
3.在设置了较高的 poly_modulus_degree 时需要更长的计算时间。

- BFV 方案特点：

1.支持整数域数据：BFV 最初设计用于整数域的同态加密方案。
2.较为简单：BFV 方案通常比 CKKS 方案更简单，他主要专注于对整数的同态加密。但精度不如 CKKS 方案。
3.有限的精度：BFV 方案对于加密整数的精度有一定限制，可能受到明文模数的限制，无法进行高精度的实数计算。
4.对于一些复杂运算效率较低。

这是两种有差异的算法，参数类型和参数设置也无法达到完全相同，CKKS 方案中的 coeff_mod_bit_sizes（系数模比特大小）可能影响了加密速度的表现。同时，CKKS 方案在处理实数域数据时具有优势，这也可能导致在处理图像数据时 CKKS 方案的密态匹配速度较快。
由于图像像素值是整数值，BFV 方案更适合于整数域，故在加密时间有略微优势。在计算向量的余弦相似度时，涉及较高精度的计算，这时 CKKS 方案具有优势，在匹配密文步骤消耗更少的时间。
综上，在不同的实际场景中应权衡不同算法的利弊，以达到最优的性能。

当然，在增加了 CKKS 方案的多项式模数后，加密时间、解密时间以及匹配时间都有一定程度的增加。但增加了密文容量以及安全性。

```
# 创建context加密环境
context = ts.context(
    ts.SCHEME_TYPE.CKKS,
    poly_modulus_degree=32768,
    coeff_mod_bit_sizes=[60, 40, 40, 60]
)
context.global_scale = 2**40
```

```
----------------encrypt images--------------------
encrypt NO.0 image cost: 0.06000542640686035 seconds
encrypt NO.1 image cost: 0.06001782417297363 seconds
encrypt NO.2 image cost: 0.0560154914855957 seconds
encrypt NO.3 image cost: 0.05631709098815918 seconds
encrypt NO.4 image cost: 0.06003403663635254 seconds
encrypt 5 images cost 0.2923898696899414 seconds
encrypt a single image cost: 0.05847797393798828 seconds on average
----------------matching images(close to 0 means similar)--------------------
No.0's similarity: 0.0954366955931274
No.1's similarity: 0.13506171847586212
No.2's similarity: 0.15748648103200624
No.3's similarity: 0.18231188787940256
No.4's similarity: -1.532107773982716e-14
compare cost: 5.542782783508301 seconds
compare a single image cost: 1.1085565567016602 seconds on average
----------------decrypt image(if matching is successful)--------------------
Match successfully!
The most similar image index: 4, similarity: -1.532107773982716e-14
decrypt cost: 0.00800013542175293 seconds
```

**匹配失败的情况(相似度大于阈值)：**

```
----------------encrypt images--------------------
encrypt NO.0 image cost: 0.060013771057128906 seconds
encrypt NO.1 image cost: 0.060022592544555664 seconds
encrypt NO.2 image cost: 0.0557801723480224 seconds
encrypt NO.3 image cost: 0.05952000617980957 seconds
encrypt NO.4 image cost: 0.06003093719482422 seconds
encrypt 5 images cost 0.2953674793243408 seconds
encrypt a single image cost: 0.05907349586486817 seconds on average
----------------matching images(close to 0 means similar)--------------------
No.0's similarity: 0.21665820848781958
No.1's similarity: 0.16961841016914259
No.2's similarity: 0.23935806315668318
No.3's similarity: 0.17463141561203677
No.4's similarity: 0.24933034582153035
compare cost: 5.214582681655884 seconds
compare a single image cost: 1.0429165363311768 seconds on average
----------------decrypt image(if matching is successful)--------------------
Match failure!
The most similar image index: 1, similarity: 0.16961841016914259
```

## 3.2 GPU 加速结果分析。

```
dell@dell:~/下载/cuFHE-master/cufhe/bin$ ./test_api_cpu
------ Key Generation ------
------ Test Encryption/Decryption ------
Number of tests:        1024
PASS
per gate time: 1.86191e-05 seconds
------ Test NAND Gate ------
Number of tests:        4
PASS
per gate time: 2.48396 seconds
Elapsed time: 14.2922 seconds
dell@dell:~/下载/cuFHE-master/cufhe/bin$ ./test_api_gpu
------ Key Generation ------
------ Test Encryption/Decryption ------
Number of tests:        2624
PASS
per test time: 1.88465e-05 seconds
------ Initilizating Data on GPU(s) ------
------ Test NAND Gate ------
Number of tests:        2624
0.180027 ms / gate
PASS
------ Cleaning Data on GPU(s) ------
Elapsed time: 3.86441 seconds
```

可以看到，两者的“Key Generation”和“Test Encryption/Decryption”的用时差距并不大，因为这两个步骤还没有使用 GPU 进行加速。

在测试密态下的逻辑门运算时，GPU 的优势就显现出来了，在 GPU 的支持下，每个逻辑门运算仅花费 0.180027ms；但在 CPU 条件下，则要花费 2s。

且 GPU 测试样例的数量是 CPU 测试样例数量的数倍，但总的用时还是比 CPU 快了 10s 左右。