计算机系统原理第三次实验

魏照林 202000460083
郭灿林 202000460092
李岱耕 202000460088
孙英皓 202022460109
李奕璇 202000460055

2022 年 5 月 4 日

目录

1 实验环境

1.1 任务一

硬件环境：

处理器	Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz
机带 RAM	16.0 GB (15.8 GB 可用)
设备 ID	9A1F2444-279B-471F-80AE-485BF17F0F25
产品 ID	00342-35806-97931-AAOEM
系统类型	64 位操作系统, 基于 x64 的处理器

软件环境：

Visual studio 2022

1.2 任务二

硬件环境：

基准速度:	2.90 GHz
插槽:	1
内核:	8
逻辑处理器:	16
虚拟化:	已启用
L1 缓存:	512 KB
L2 缓存:	4.0 MB
L3 缓存:	8.0 MB

软件环境：

linux(Windows 下的 wsl 子系统)

使用 clang 编译

2 实验要求

1. 实现一个基础的单线程 C 语言版本的 SM4 算法，并验证其正确性（对比开源库如 openssl 或 gmssl）。

2. 想尽各种办法在 CPU 上加速该算法（不许使用现成的库函数或硬件加密芯片）。可以考虑采用多线程、SIMD、流水线、循环展开等各种软件层面的加速优化方法，可参考网上的各种资料。

3. 设计一个实验，分析优化过的代码的延迟（latency）和吞吐量（throughput）相比第 1 部分的单线程基础版本以及开源库的加速比。

3 任务一

3.1 实验原理

3.1.1 加密算法

每一次轮函数的流程图如图所示：

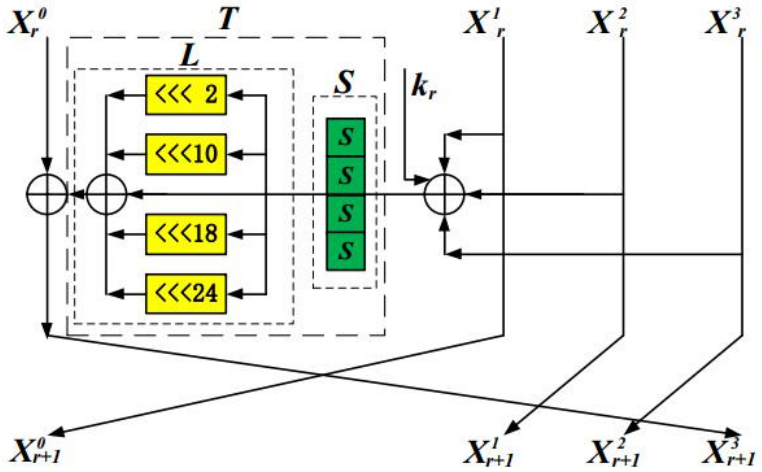

图 1: 轮函数流程

轮函数:

$$X^3_{\mathrm{r+1}} = X^0_r \oplus T(X^1_r \oplus X^2_r \oplus X^3_r \oplus k_r) = X^0_r \oplus L \circ S(X^1_r \oplus X^2_r \oplus X^3_r \oplus k_r),$$
$$X^2_{\mathrm{r+1}} = X^3_r, X^1_{\mathrm{r+1}} = X^2_r, X^0_{\mathrm{r+1}} = X^1_r, r = 0, 1, 2, \ldots, 31$$

合成置换 T:

$\mathrm{F}_2^{32} \to \mathrm{F}_2^{32}$ 的可逆转换,$T = L \circ S$

非线性变换 S:

由 4 个并行的 S 盒构成。输入为 $A = (a_0, a_1, a_2, a_3)$，则输出为 $B = (S(a_0), S(a_1), S(a_2), S(a_3))$

线性变换 L:

输入为 $B \in F_2^{32}$，输出为 $C \in F_2^{32}$，则 $C = L(B)$=$B \oplus (B <<< 2) \oplus (B <<< 10) \oplus (B <<< 18) \oplus (B <<< 24)$

反序变换 R:

$(Y_0, Y_1, Y_2, Y_3) = R(X_{32}, X_{33}, X_{34}, X_{35}) = (X_{35}, X_{34}, X_{33}, X_{32})$

3.1.2 密钥拓展算法

将 128 比特的加密密钥 K 扩展成 32 个 32 比特的轮子密钥，记为 rk_i(i=0,⋯,31)

系统参数 FK = (0xA3B1BAC6, 0x56AA3350, 0x677D9197, 0xB27022DC)

k = $(k\_0, k\_1, k\_2, k\_3) = K \oplus FK$

For i=0,1,⋯,31//

$$rk\_i = k\_i{+}4 = k\_i \oplus L' \circ S(k\_i{+}1 \oplus k\_i{+}2 \oplus k\_i{+}3 \oplus ck\_i)$$

其中，

$$L'(B) = B \oplus (B <<< 13) \oplus (B <<< 23)$$

$$ck\_i = (ck\_i, 0, ck\_i, 1, ck\_i, 2, ck\_i, 3),$$

$$ck\_i, j = (4i + j) \times 7 \; mod \; 256$$

3.2 ECB 模式算法实现

对于普遍的遍历实现方式，无论是在密钥扩展算法中或是加密算法中，对于三十二轮中的每一轮来说都会存在一个 32 长的中间数组存储 K 或 X，而这将会增大空间开销，并且访问数组的速度慢于访问普通变量的速度，因此在具体的实现过程中将对此方面进行优化。

具体的优化实现方式为，区别于传统方式，对于密钥扩展算法以及加密算法的中间变量仅开辟四个内存空间使用强制类型转换即以四个变量进行承接。对于 32 轮的 for 循环，将其改变为仅进行 8 次 for 循环而对于每次内层循环而言，每次进行四次运算，每次将中间变量的值与 rk 或 x 进行替换，以此节省空间开销以及时间开销。核心代码实现如下：

```
uint32_t x0, x1, x2, x3;
uint32_t* temp;
temp = (uint32_t*)input;
x0 = temp[0];
x1 = temp[1];
x2 = temp[2];
x3 = temp[3];
for (int i = 0; i < 32; i += 4)
{
        uint32_t mid_x = x1 ^ x2 ^ x3 ^ rk[i];
        mid_x = ByteSub(mid_x);
        x0 ^= Lun_last(mid_x);
        mid_x = x2 ^ x3 ^ x0 ^ rk[i + 1];
        mid_x = ByteSub(mid_x);
        x1 ^= Lun_last(mid_x);
        mid_x = x3 ^ x0 ^ x1 ^ rk[i + 2];
        mid_x = ByteSub(mid_x);
        x2 ^= Lun_last(mid_x);
        mid_x = x2 ^ x0 ^ x1 ^ rk[i + 3];
        mid_x = ByteSub(mid_x);
        x3 ^= Lun_last(mid_x);
}
```

3.3 CBC 模式实现

在开销测试方面，考虑到不同的加密模式下算法的效率可能会有区别。因此本文在这里对 SM4 进行 CBC 加密模式的实现。以便于后续进行时间开销对比。其流程如下所示：

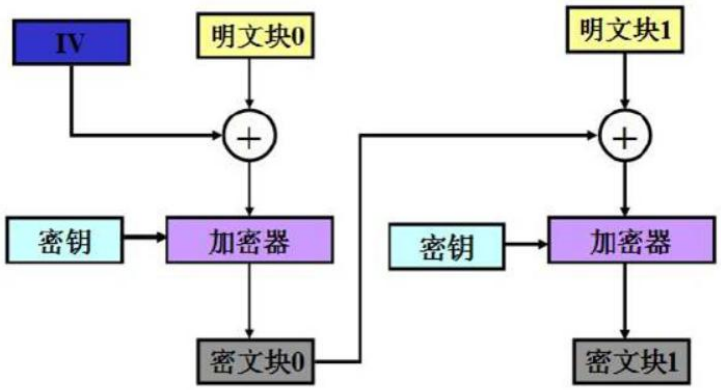

图 2: CBC 模式流程

CBC 模式主要基于两种思想，第一，所有分组的加密都链接在一起，其中各分组所用的密钥相同。加密时输入的是当前的明文分组和上一个密文分组的异或，这样使得密文分组不仅依赖当前明文分组，而且还依赖前面所有的明文分组。因此，加密算法的输入不会显示出与这次的明文分组之间的固定关系，所以重复的明文分组不会再密文中暴露出这种重复关系。第二，加密过程使用初始量进行了随机化。基于在 1.2 中所实现的 SM4 加解密算法，便可很容易的实现 CBC 模式加密，具体核心代码如下所示：

```
1  void SM4_CBC(uint8_t* In, uint8_t* Out, uint8_t* iv, uint32_t
       length,int flag, uint32_t* rk)
2  {
3          uint8_t* temp_in = In;
4          uint8_t* temp_out = Out;
5          uint8_t temp_temp_in[SM4_BLOCK_SIZE] = { 0 };
6          uint8_t temp_temp_out[SM4_BLOCK_SIZE] = { 0 };
7          uint8_t temp_iv[SM4_BLOCK_SIZE] = {0};
8          memcpy(temp_iv, iv, SM4_BLOCK_SIZE);
9          if (flag == ENC)
10         {
11                 while (length >= SM4_BLOCK_SIZE)
```

```
12                      {
13                              for (int i = 0; i < SM4_BLOCK_SIZE; i++)
14                              temp_temp_in[i] = temp_in[i] ^ temp_iv[i];
15                              SM4_Enc_Dec(temp_temp_in, temp_out, rk);
16                              memcpy(temp_iv, temp_out, SM4_BLOCK_SIZE);
17                              length -= SM4_BLOCK_SIZE;
18                              temp_in += SM4_BLOCK_SIZE;
19                              temp_out += SM4_BLOCK_SIZE;
20                      }
21              }
22              else
23              {
24                      while (length >= SM4_BLOCK_SIZE)
25                      {
26                              SM4_Enc_Dec(temp_in, temp_temp_out, rk);
27
28                              for (int i = 0; i < SM4_BLOCK_SIZE; i++)
29                              temp_out[i] = temp_temp_out[i] ^ temp_iv[i
                                    ];
30
31                              memcpy(temp_iv, temp_in, SM4_BLOCK_SIZE);
32                              length -= SM4_BLOCK_SIZE;
33                              temp_in += SM4_BLOCK_SIZE;
34                              temp_out += SM4_BLOCK_SIZE;
35                      }
36              }
37      }
```

3.4 ECB 模式与 CBC 模式效率对比

当数据规模较小时，无论使用何种模式其时间开销均较小，因此本文采用适当的数据规模对两种加密模式进行对比。加密时间结果对比如下：

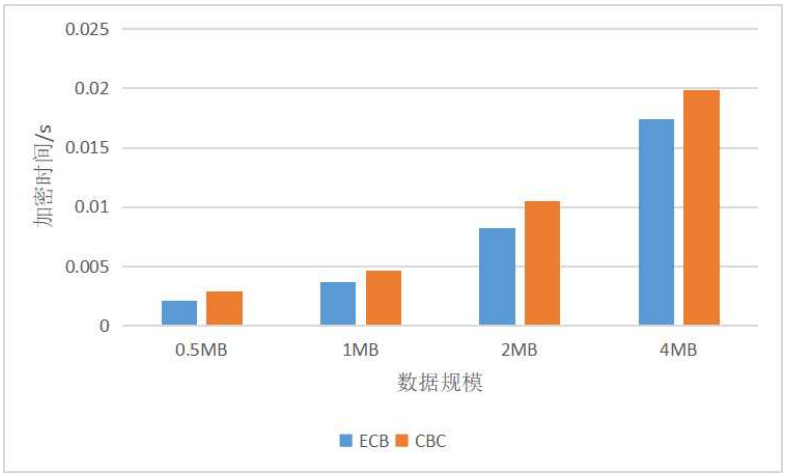

图 3: 不同数据规模下两种加密模式的加密时间对比

由上图可知，两种加密模式效率基本一致，CBC 模式效率会略低于 ECB 模式，这是由于 iv 的更新以及 iv 与明密文的异或操作会增加时间开销，但影响并不太大。

3.5 正确性检验

本文以验证 ECB 模式为例,将明文与密钥均设置为 0123456789abcdeffedcba9876543210 进行测试。通过与标准库加密结果进行对比得出其加密皆为 681edf34d26965e86b3e94f536e4246。并且解密后能够得到与明文相同的结果，这也再次验证了密码方案的正确性。

4 任务二

4.1 查表优化

查表方法的核心思想是将密码算法轮函数中尽可能多的变换操作制成表。SM4 加/解密轮函数中的 T 变换由非线性变换 和线性变换 L 构成。将非线性变换 的输入记为 X=(x0,x1,x2,x3)∈(Z28)4，输

出记为 Y=(y0,y1,y2,y3)∈(Z28)4。可将非线性变换 τ 的操作定义如下。

$$y_i = Sbox(x_i), 0 \leq i < 4$$

将线性变换 L 的输入记 $P = (p_0, p_1, ..., p_{n-1})) \in (\mathbb{Z}_2^m)^4$，输出记为 $Q = (q_0, q_1, ..., q_{n-1})) \in (\mathbb{Z}_2^m)^4$。其中，m 大小需为 SM4 使用 S 盒规模的倍数，m 与 n 的关系满足 n=32/m。由于 L 中仅包含循环移位和异或操作。因此，可将线性变换 L 的操作定义为下式：

$$Q = L(B) = L(p_0 \lll (n-1)m) \oplus L(p_1 \lll (n-2)m) \oplus \cdots \oplus L(p_{n-1}).$$
$$Q = T(X) = L(Sbox(x_0) \lll (n-1)m) \oplus L$$
$$(Sbox(x_1) \lll (n-2)m) \oplus \cdots \oplus L(Sbox(x_{n-1})).$$

m 和 n 有多种取法，由参考文献结果可知，当 m=8，n=8 时，SM4 性能最佳。制表操作如下：

$$\mathbf{T3}[d_0] = L(Sbox(d_0) \lll 24);$$
$$\mathbf{T2}[d_1] = L(Sbox(d_1) \lll 16);$$
$$\mathbf{T1}[d_2] = L(Sbox(d_2) \lll 8);$$
$$\mathbf{T0}[d_3] = L(Sbox(d_3)); d_i \in [0, 255], 0 \leq i < 4。$$

核心代码：

Create Table

```
1 void S_boxes_init() {
2     for (int i = 0; i < 256; i++) {
3         uint32_t b = S_box[i] ^ S_box[i] << 2 ^ S_box[i] << 10 ^
              S_box[i] << 18 ^ S_box[i] << 24;
4         S_boxes_0[i] = b;
5         S_boxes_1[i] = ROL(b, 010);
```

```
        S_boxes_2[i] = ROL(b, 020);
        S_boxes_3[i] = ROL(b, 030);
    }
    S_boxes_inited = 1;
}
```

find Table

```
void SM4_encrypt(const uint32_t rk[32], const uint32_t X[4],
    uint32_t Y[4]) {
        uint32_t T[36] = {X[0], X[1], X[2], X[3]};
        for (int i = 0; i < 32; i++) {
                uint32_t a = T[i + 1] ^ T[i + 2] ^ T[i + 3] ^ rk[i
                    ];
                T[i + 4] = T[i] ^ S_boxes_0[a & 0xff] ^ S_boxes_1[
                    a >> 010 & 0xff] ^ S_boxes_2[a >> 020 & 0xff] ^
                     S_boxes_3[a >> 030 & 0xff];
        }
        Y[0] = T[35];
        Y[1] = T[34];
        Y[2] = T[33];
        Y[3] = T[32];
}
```

4.2 SIMD 指令集加速

由于本文使用的是 AVX2 指令集，支持 256 位寄存器，但是加密一次只用 128 位，因此用 SIMD 指令集去优化的话，最小加密长度是 128 * 8 位，即 1024 位，这样才能把寄存器填满，充分利用 SIMD 寄存器。利用以下方式装载，每一列是每一组 128bit 明文，然后每一列分别进行迭代，这样就可以利用寄存器并行处理 8 组加密了。

```
__m256i mmT[4] = {
```

```
        _mm256_set_epi32(X[0x00], X[0x04], X[0x08], X[0x0c], X[0
            x10], X[0x14], X[0x18], X[0x1c]),
        _mm256_set_epi32(X[0x01], X[0x05], X[0x09], X[0x0d], X[0
            x11], X[0x15], X[0x19], X[0x1d]),
        _mm256_set_epi32(X[0x02], X[0x06], X[0x0a], X[0x0e], X[0
            x12], X[0x16], X[0x1a], X[0x1e]),
        _mm256_set_epi32(X[0x03], X[0x07], X[0x0b], X[0x0f], X[0
            x13], X[0x17], X[0x1b], X[0x1f]),
};
```

后面则是把原来的异或、移位、与、查表操作别分替换成 AVX2 指令集中的 _mm256_xor_si256(),_mm256_srli_epi32(),_mm256_and_si256(),_

```
void encrypt(const uint32_t X[32], uint32_t Y[32]){
    const __m256i mm0xff = _mm256_set_epi32(0xff, 0xff, 0xff, 0xff
        , 0xff, 0xff, 0xff, 0xff);
    __m256i mmT[4] = {
        _mm256_set_epi32(X[0x00], X[0x04], X[0x08], X[0x0c], X[0
            x10], X[0x14], X[0x18], X[0x1c]),
        _mm256_set_epi32(X[0x01], X[0x05], X[0x09], X[0x0d], X[0
            x11], X[0x15], X[0x19], X[0x1d]),
        _mm256_set_epi32(X[0x02], X[0x06], X[0x0a], X[0x0e], X[0
            x12], X[0x16], X[0x1a], X[0x1e]),
        _mm256_set_epi32(X[0x03], X[0x07], X[0x0b], X[0x0f], X[0
            x13], X[0x17], X[0x1b], X[0x1f]),
    };
    for (int i = 0; i < 32;) {
        __m256i mma;
        mma = _mm256_xor_si256(_mm256_xor_si256(mmT[1], mmT[2]),
            _mm256_xor_si256(mmT[3], mmrk[i++]));
        mmT[0] = _mm256_xor_si256(mmT[0], _mm256_xor_si256(
            _mm256_xor_si256(
                _mm256_i32gather_epi32(S_boxes_0, _mm256_and_si256
                    (mma, mm0xff), 4),
                _mm256_i32gather_epi32(S_boxes_1, _mm256_and_si256
                    (_mm256_srli_epi32(mma, 010), mm0xff), 4)
            ),
```

```
17                 _mm256_xor_si256(
18                     _mm256_i32gather_epi32(S_boxes_2, _mm256_and_si256
                           (_mm256_srli_epi32(mma, 020), mm0xff), 4),
19                     _mm256_i32gather_epi32(S_boxes_3,
                           _mm256_srli_epi32(mma, 030), 4)
20                 )
21             ));
22             mma = _mm256_xor_si256(_mm256_xor_si256(mmT[2], mmT[3]),
                   _mm256_xor_si256(mmT[0], mmrk[i++]));
23             mmT[1] = _mm256_xor_si256(mmT[1], _mm256_xor_si256(
24                 _mm256_xor_si256(
25                     _mm256_i32gather_epi32(S_boxes_0, _mm256_and_si256
                           (mma, mm0xff), 4),
26                     _mm256_i32gather_epi32(S_boxes_1, _mm256_and_si256
                           (_mm256_srli_epi32(mma, 010), mm0xff), 4)
27                 ),
28                 _mm256_xor_si256(
29                     _mm256_i32gather_epi32(S_boxes_2, _mm256_and_si256
                           (_mm256_srli_epi32(mma, 020), mm0xff), 4),
30                     _mm256_i32gather_epi32(S_boxes_3,
                           _mm256_srli_epi32(mma, 030), 4)
31                 )
32             ));
33             mma = _mm256_xor_si256(_mm256_xor_si256(mmT[3], mmT[0]),
                   _mm256_xor_si256(mmT[1], mmrk[i++]));
34             mmT[2] = _mm256_xor_si256(mmT[2], _mm256_xor_si256(
35                 _mm256_xor_si256(
36                     _mm256_i32gather_epi32(S_boxes_0, _mm256_and_si256
                           (mma, mm0xff), 4),
37                     _mm256_i32gather_epi32(S_boxes_1, _mm256_and_si256
                           (_mm256_srli_epi32(mma, 010), mm0xff), 4)
38                 ),
39                 _mm256_xor_si256(
40                     _mm256_i32gather_epi32(S_boxes_2, _mm256_and_si256
                           (_mm256_srli_epi32(mma, 020), mm0xff), 4),
41                     _mm256_i32gather_epi32(S_boxes_3,
                           _mm256_srli_epi32(mma, 030), 4)
42                 )
```

```
43            ));
44            mma = _mm256_xor_si256(_mm256_xor_si256(mmT[0], mmT[1]),
                  _mm256_xor_si256(mmT[2], mmrk[i++]));
45            mmT[3] = _mm256_xor_si256(mmT[3], _mm256_xor_si256(
46                  _mm256_xor_si256(
47                        _mm256_i32gather_epi32(S_boxes_0, _mm256_and_si256
                              (mma, mm0xff), 4),
48                        _mm256_i32gather_epi32(S_boxes_1, _mm256_and_si256
                              (_mm256_srli_epi32(mma, 010), mm0xff), 4)
49                  ),
50                  _mm256_xor_si256(
51                        _mm256_i32gather_epi32(S_boxes_2, _mm256_and_si256
                              (_mm256_srli_epi32(mma, 020), mm0xff), 4),
52                        _mm256_i32gather_epi32(S_boxes_3,
                              _mm256_srli_epi32(mma, 030), 4)
53                  )
54            ));
55      }
56      uint32_t T_0[8], T_1[8], T_2[8], T_3[8];
57      _mm256_storeu_si256((__m256i *)T_0, mmT[0]);
58      _mm256_storeu_si256((__m256i *)T_1, mmT[1]);
59      _mm256_storeu_si256((__m256i *)T_2, mmT[2]);
60      _mm256_storeu_si256((__m256i *)T_3, mmT[3]);
61      Y[0x00] = T_3[7];Y[0x04] = T_3[6];Y[0x08] = T_3[5];Y[0x0c] =
            T_3[4];Y[0x10] = T_3[3];Y[0x14] = T_3[2]; Y[0x18] = T_3[1];
            Y[0x1c] = T_3[0];Y[0x01] = T_2[7];Y[0x05] = T_2[6];Y[0x09]
            = T_2[5];Y[0x0d] = T_2[4];Y[0x11] = T_2[3];Y[0x15] = T_2
            [2];Y[0x19] = T_2[1]; Y[0x1d] = T_2[0];Y[0x02] = T_1[7];Y[0
            x06] = T_1[6]; Y[0x0a] = T_1[5];Y[0x0e] = T_1[4];Y[0x12] =
            T_1[3];Y[0x16] = T_1[2]; Y[0x1a] = T_1[1]; Y[0x1e] = T_1
            [0];Y[0x03] = T_0[7];Y[0x07] = T_0[6];Y[0x0b] = T_0[5];Y[0
            x0f] = T_0[4];Y[0x13] = T_0[3]; Y[0x17] = T_0[2];Y[0x1b] =
            T_0[1];Y[0x1f] = T_0[0];
62  }
```

4.3 多线程实现

SM4 内部算法是迭代型的，因此在单组加密算法里实现多线程意义不大，因此我们考虑的是加密多组明文的情况，即从提高吞吐量的角度优化。多线程思路和之前向量求和是类似的，出于简化的角度考虑，我们采用 ECB 加密模式，加解密都可以并行处理。核心代码：

```
1  void *threading(void *pbuf) {
2          uint32_t *plaintext = ((args_t *)pbuf)->plaintext, *rk =
               ((args_t *)pbuf)->rk, *ciphertext = ((args_t *)pbuf)->
               ciphertext, *end = ((args_t *)pbuf)->end;
3          char mode = ((args_t *)pbuf)->mode;
4          while(plaintext < end){
5                  if (mode == 1) SM4_encrypt(rk, plaintext,
                       ciphertext);
6                  else SM4_decrypt(rk, ciphertext, plaintext);
7                  plaintext += 4;
8                  ciphertext += 4;
9          }
10         return NULL;
11 }
```

5 任务三

由于 SM4 本身数据很快，加密较小的数据结果不明显，因此我们这里考虑加密 1GB 的数据；另外，在加密算法实现中，一组轮密钥可以加密多组明文，密钥扩展可以提前做好并存储下来，因此我们不考虑密钥扩展的时间。

5.1 时间开销对比-O0 编译

不做优化的 SM4：time = 28.550354 s

查表优化：time = 10.619774 s

SIMD 查表优化：time = 21.716868 s

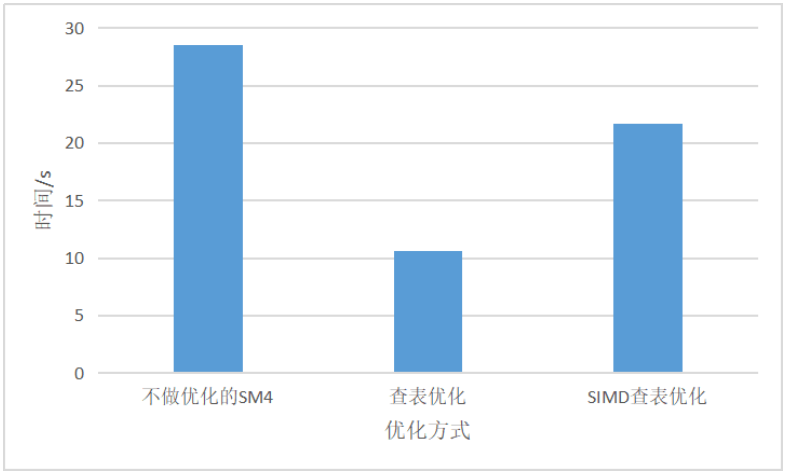

图 4: 时间开销对比-O0 编译

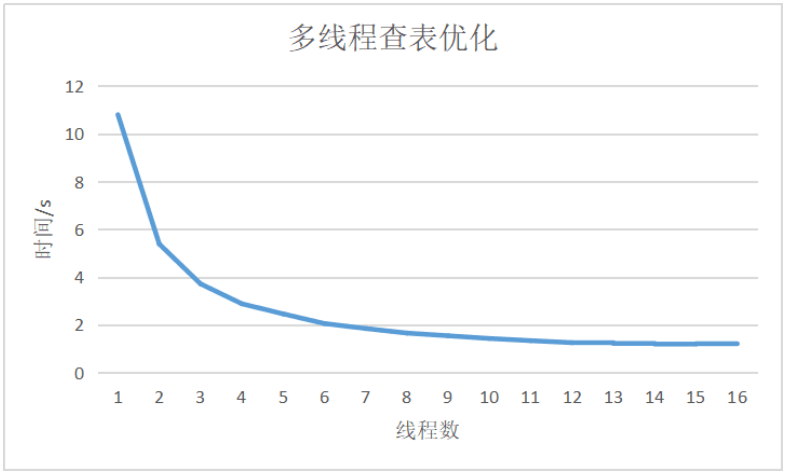

图 5: 多线程查表优化

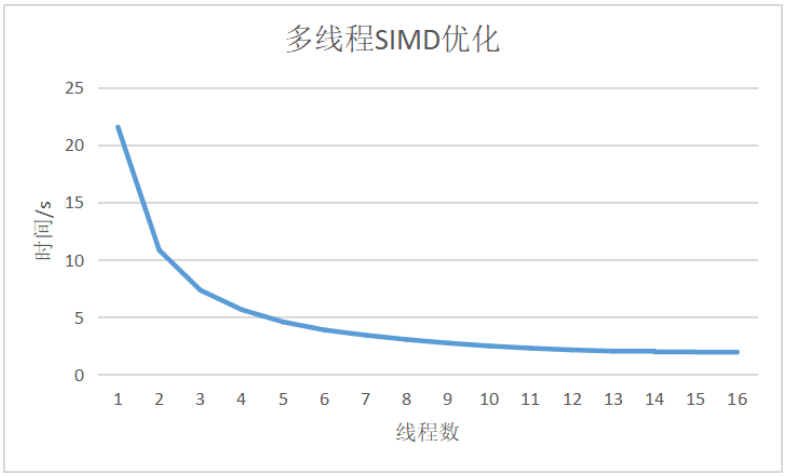

图 6: 多线程 SIMD 优化

利用 linux 下的 gprof 工具，生成 gmon.out 文件，可以分析程序运行过程各个函数调用的次数和时间。

```
fixed_point@LAPTOP-343L2PRM:/mnt/d/Codes/SM4$ gprof a.out gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 66.73     16.95    16.95 2147483680     7.89     7.89  tau
 33.35     25.42     8.47 67108864   126.21   378.79  SM4_encrypt
  0.28     25.49     0.07                             main
  0.00     25.49     0.00        1     0.00   252.57  key_expand
```

结果分析:

从图中可以看到 SM4 最占时间的是 tau 变化，这是因为 tau 变化中有四次查 S-box 表，我们知道查表操作通常是耗时最多的。

这里可以看到在开-O0 编译的情况下，用 SIMD 做查表优化反而速度变慢了，猜测是因为-O0 下对 SIMD 寄存器的使用可能有限制，或者在查表优化中编译器就自动利用了 SIMD 寄存器去做运算了，而 SIMD 开销比较大。下面用-O1 编译就可以得到预期的结果了。

由于 SM4 划分多线程是线性的，没有多余的影响因素，因此

不难预测到，线程数和电脑处理器数量一致时速度最快，前面提速比较明显，后面基本趋于一致。

5.2 时间开销对比-O1 编译

不做优化的 SM4：time = 10.227523 s
查表优化：time = 6.773726 s
SIMD 查表优化：time = 4.941447 s

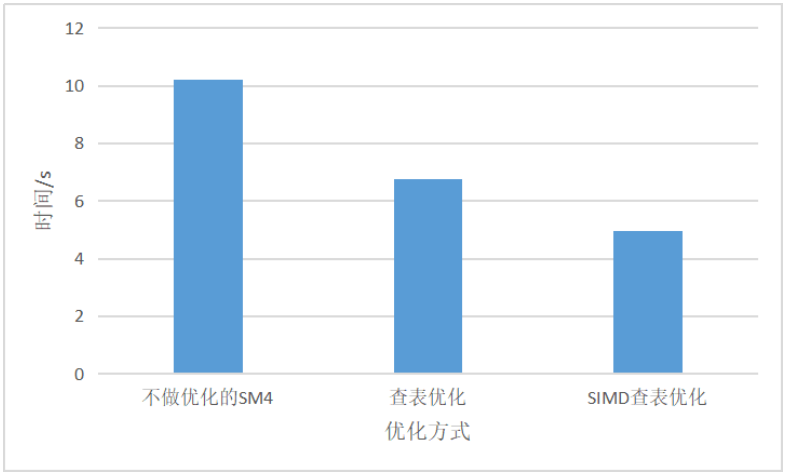

图 7: 时间开销对比-O1 编译

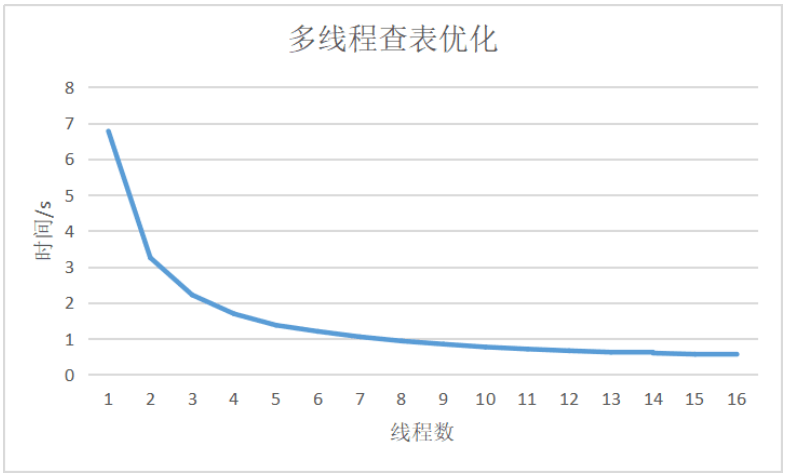

图 8: 多线程查表优化

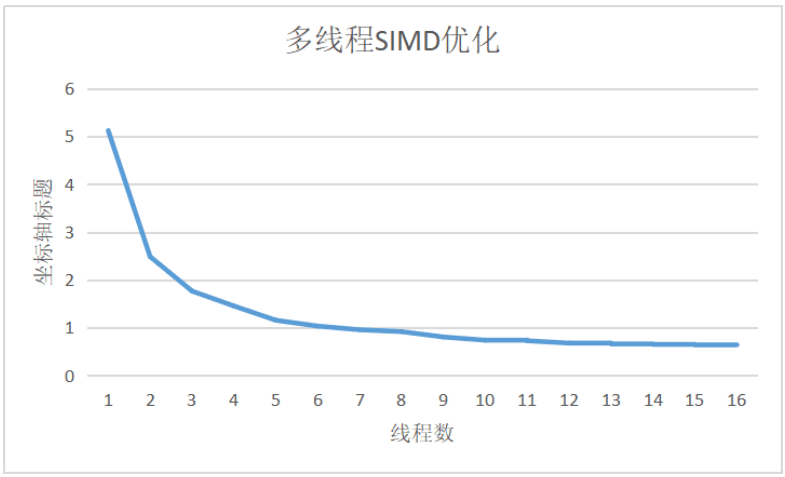

图 9: 多线程 SIMD 优化

结果分析:

可以看到，在开-O1 编译下，SIMD 查表优化就比普通的查表优化要快了将近两秒。多线程结果和-O0 编译基本一致。线程数越多，SIMD 提速效果反而不明显了，猜测是因为 CPU 中 SIMD 寄存器有限，线程数多的话会争用，不能全部利用起来。

5.3 总结--延迟与吞吐量对比

这里以-O1 编译为例，加密 1GB 的数据，普通的查表优化比不做优化快了大约 1.5 倍，应用 SIMD 查表优化比不做优化快了大约 2 倍，应用 SIMD 查表优化比普通的查表优化快了 1.3 倍。16 线程的普通查表优化比不做优化快了大约 17.2 倍。因为计算单次分组时间开销过小不具有代表性，以加密 1G 文件为例计算时间开销。

测试量	不做优化	查表优化	SIMD 查表	多线程 SIMD 查表
延迟	10.227523s	10.619774s	21.716868s	0.649150 s
吞吐量	107505172833.73502	162320062514.4861	222508028068.70132	1693771282101.2092

6 实验分工

成员	学号	分工
李奕璇	202000460055	代码编写，数据整理分析，论文排版
魏照林	202000460083	代码编写，任务一代码以及思路编写，论文排版
李岱耕	202000460088	代码编写以及任务二优化实现
郭灿林	202000460092	代码编写，任务二、三论文编写
孙英皓	202022460109	代码编写以及任务二优化实现