计算机组成原理实验

实验名称 **国密 SM4 的软件实现和优化**

院　　系 网络空间安全学院 (研究院)

李卓群　学号:202000460041

刘力豪　学号:202000460095

杨文涛　学号:202000460052

孔伟骁　学号:202000460096

马洋　学号:20200460146

实验日期 2022 年 7 月 31 日

摘要

本次实验是对国密 SM4 算法进行实现，首先是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，我们通过官方文档对 SM4 算法进行了实现，并对其加密和解密的正确性同时进行了验证。

然后我们在基本实现的基础上对单线程的实现采用了多种手段进行优化，比如采用移位代替模运算的方式，将密钥生成与加密过程分离，我们还利用了 SIMD 指令集和查表方法对算法进行了进一步的优化，并和库函数的实现版本进行了对比，比较了代码的延迟特性

最后我们采用长文本进行加密，通过 ecb 模式并行处理长文本，并利用多线程进行加速，计算相应算法的吞吐量。

目录

1 SM4 国密算法介绍

1.1 SM4 介绍

SM4 算法于 2012 年被国家密码管理局确定为国家密码行业标准，SM4 算法的出现为将我国商用产品上的密码算法由国际标准替换为国家标准提供了强有力的支撑。随后，SM4 算法被广泛应用于政府办公、公安、银行、税务、电力等信息系统中，其在我国密码行业中占据着及其重要的位置。类似于 DES、AES 算法，SM4 算法也是一种分组密码算法。

1.2 原理介绍

SM4 算法主要包括异或、移位以及盒变换操作。它分为密钥拓展和加/解密两个模块，其中移位变换是指循环左移；盒变换是将 8bit 输入映射到 8bit 输出的变换，是一个固定的变换。

	acknowledged data
ACK 1	566
ACK 2	1460
ACK 3	1460
ACK 4	1460
ACK 5	1460
ACK 6	1460
ACK 7	1147
ACK 8	1460
ACK 9	1460
ACK 10	1460
ACK 11	1460
ACK 12	1460

	No	Seq	RTT
6	566	566	
9	2026	1460	
12	3486	1460	
14 4	4946	1460	
15	6406	1460	
16	7866	1460	

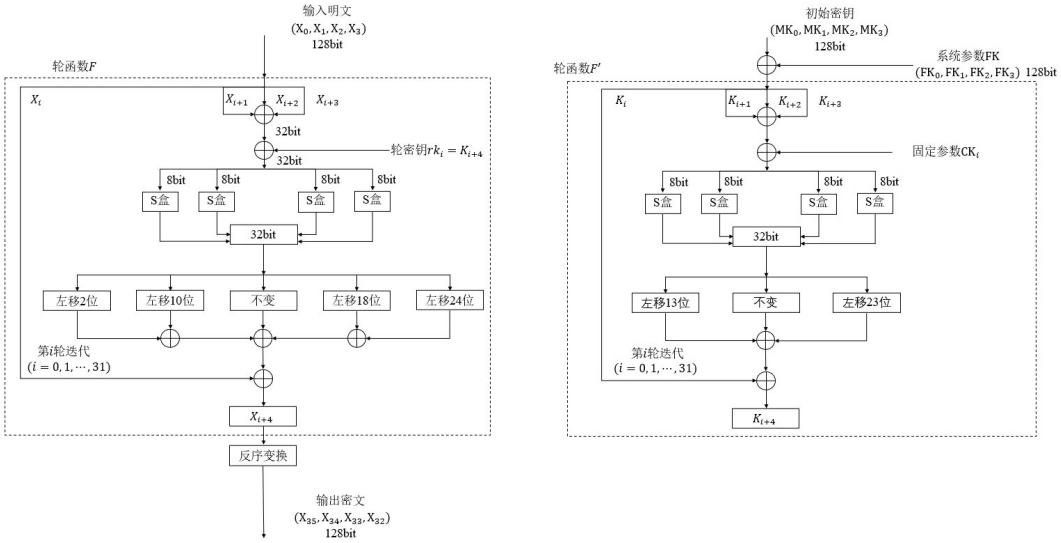

图 1: 算法流程

加解密运算

1. 首先将输入的 128bit 明文分成 4 个 32bit 的数据 x_0, x_1, x_2, x_3, 并作 32 轮轮变换.

2.x_i 暂时不做处理，将 $x_{i+1}, x_{i+2}, x_{i+3}$ 和轮密钥 rk_i 异或得到一个 32bit 都数据，作为 S 盒变换的输入.

3. 下面进行 S 盒变换的过程，每个 S 盒的输入都是 8 位的一个字节，将这 8 位的前 4 位对应的 16 进制数作为行编号，后 4 位作为列编号，然后用 S 盒中对应位置的数进行代替.

4. 然后将刚才 Sbox 的结果分别循环左移 2，10，18，24 位，得到的数与 Sbox 的输出进行异或然后得到 x_{i+4}.

5. 将最后输出的 $x_{35}, x_{34}, x_{33}, x_{32}$ 合并成一个 128bit 都数据，最为最后的结果进行输出.

6. 解密的过程和加密的过程其实没有本质区别，由于是对称密码方案，在解密时我们只需要将 32 轮的轮密钥反序使用即可。

密钥拓展：

1. 将初始的原始密钥 key 拆分成 4 个 32bit 的数据 K_0, K_1, K_2, K_3 , 然后将它们分别和异或固定参数 FK_0, FK_1, FK_2, FK_3 异或得到用于循环的密钥. 2. 然后进入轮密钥 rk 的生成，k_i 暂时不做处理，将 $k_{i+1}, k_{i+2}, k_{i+3}$ 和固定参数 CK_i 异或得到一个 32bit 都数据，作为 S 盒变换的输入.

3. 将经过 S 盒变换的数据分别左移 13 位和 23 位然后与 S 盒输出数据进行异或，然后在将异或结果与 k_i 异或得到迭代输出 k_{i+4} 4. 这样不断进行迭代，逐步得到 32 轮的轮密钥。

2 实现 SM4 加密算法

2.1 普通实现

在 S 盒实现我们采用基本移位运算对高 4 位和低 4 位的值进行提取，然后通过 x 乘行加列将替换的值在一维数组中提取出来.

```
uint8_t S_replace(uint8_t in){
    uint8_t xcoor = in >> 4;
```

```
    uint8_t ycoor = in << 4;
    ycoor = ycoor >> 4;
    int index = xcoor * 16 + ycoor;
    return SBOX[index];
}
```

下面是循环左移的操作，一种思路我们可以利用模余运算进行实现，但实现效率由于除法的加入可能比较低，因此我们直接使用优化以后的移位运算进行实现.

未进行优化的模实现

```
u32 shiftleft(u32 a, short length) {
        for (int i = 0; i < length; i++) {
                a = a * 2 + a / 0x80000000;
        }
        //the a shift left and catch the remainder
        return a;
}
```

优化后移位实现

```
 //shift left
uint32_t Cycle_Shift_lift(uint32_t num,int Shift_number) {
    return (num << Shift_number) ^ (num >> 32 - Shift_number);
}//the high level shift number use the 32-shiftnum
```

下面给出密钥生成时的 T 函数。

```
uint32_t T_for_get_key_func(uint32_t K_for_t) {
    uint8_t *A = (uint8_t*)&K_for_t;
    uint8_t B_8_t[4];
    B_8_t[0] = S_replace(A[0]);
    B_8_t[1] = S_replace(A[1]);
    B_8_t[2] = S_replace(A[2]);
    B_8_t[3] = S_replace(A[3]);
    uint32_t* B_32_t = (uint32_t*)B_8_t;
    uint32_t B_32_t_Shift_lift_13 = Cycle_Shift_lift(*B_32_t, 13);
    uint32_t B_32_t_Shift_lift_23 = Cycle_Shift_lift(*B_32_t, 23);
    return *B_32_t^ B_32_t_Shift_lift_13^ B_32_t_Shift_lift_23;
}
```

下面给出轮函数中的 T 函数。

```
uint32_t T_for_roundfunc_func(uint32_t K_for_t) {
    uint8_t* A = (uint8_t*)&K_for_t;
    uint8_t B_8_t[4];
    B_8_t[0] = S_replace(A[0]);
    B_8_t[1] = S_replace(A[1]);
    B_8_t[2] = S_replace(A[2]);
    B_8_t[3] = S_replace(A[3]);
    uint32_t* B_32_t = (uint32_t*)B_8_t;
    uint32_t B_32_t_Shift_lift_2 = Cycle_Shift_lift(*B_32_t, 2);
    uint32_t B_32_t_Shift_lift_10 = Cycle_Shift_lift(*B_32_t, 10);
    uint32_t B_32_t_Shift_lift_18 = Cycle_Shift_lift(*B_32_t, 18);
    uint32_t B_32_t_Shift_lift_24 = Cycle_Shift_lift(*B_32_t, 24);
    return *B_32_t ^ B_32_t_Shift_lift_2 ^ B_32_t_Shift_lift_10 ^
        B_32_t_Shift_lift_18 ^ B_32_t_Shift_lift_24;
}
```

在轮函数中，我们也同样利用密钥生成方案中的思路，先将输入进行分组过 S 盒，然后最后将对应输出移位后进行异或运算，下面是加密的过程.

```
void encryption(uint32_t* ciphertext) {
    uint32_t K[36];
    uint32_t X[36] = { 0 };
    X[0] = plaintext[0]; X[1] = plaintext[1]; X[2] = plaintext[2]; X[3] =
        plaintext[3];
    K[0] = MK[0] ^ FK[0]; K[1] = MK[1] ^ FK[1]; K[2] = MK[2] ^ FK[2]; K[3]
         = MK[3] ^ FK[3];//first xor with FK
    for (int i = 4; i < 36; i++) {
        rK[i-4]=K[i] = K[i-4] ^ T_for_get_key_func(K[i-3] ^ K[i-2] ^ K[i
            -1] ^ CK[i-4]);
        printf("x");
        dump_buf(temp, 4);*/
    }
    ciphertext[0] = X[35]; ciphertext[1] = X[34]; ciphertext[2] = X[33];
        ciphertext[3] = X[32];
}
```

2.1.1 正确性验证

我们接下来对实现的 SM4 加密算法的正确性进行验证, 我们在官方说明文档上找到一个对应输入输出进行验证

输入明文:01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10

输入密钥:01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10

result:68 1E DF 34 D2 06 96 5E 86 B3 E9 4F 53 6E 42 46

图 2: 程序结果

2.1.2 移位优化和未优化时间对比

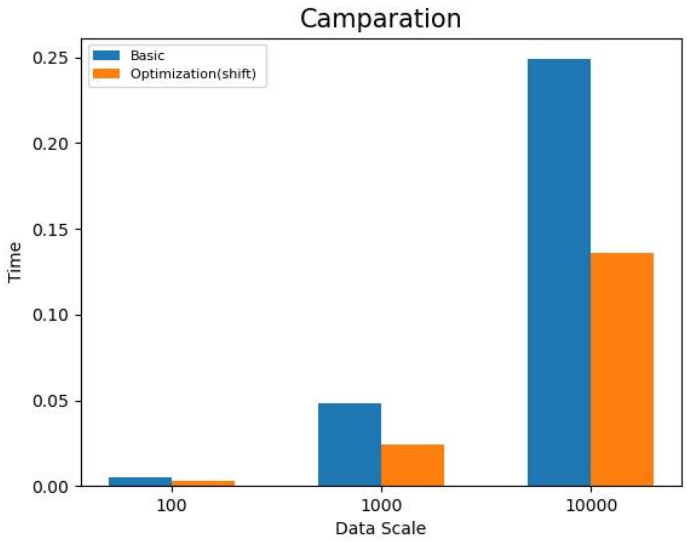

图 3: 时间对比

从图中我们可以看出，在减少数据除法运算以后，移位操作的速度有了很大提升，进而对整个程序效率有了 2 倍左右的提高。

2.2 利用库函数进行实现

2.2.1 代码实现

```
int sm4EvpEncrypt(unsigned char* key, unsigned char* in, int inl, unsigned char* out)
{
    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    EVP_CIPHER_CTX_init(ctx);
    EVP_EncryptInit_ex(ctx, EVP_sm4_ecb(), NULL, key, NULL);//设置ecb加密模式
    EVP_CIPHER_CTX_set_padding(ctx, 0);//设置不填充
    int len = 0;
    int outl = 0;
    EVP_EncryptUpdate(ctx, out, &outl, in, inl);
    len = outl;
    EVP_EncryptFinal_ex(ctx, out + len, &outl);
    len += outl;
    EVP_CIPHER_CTX_cleanup(ctx);
    return len;
}
```

2.2.2 加密结果

```
unsigned char in[] = {
0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef,
0xfe, 0xdc, 0xba, 0x98, 0x76, 0x54, 0x32, 0x10
};
unsigned char key[] = {
0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef,
0xfe, 0xdc, 0xba, 0x98, 0x76, 0x54, 0x32, 0x10
};
Microsoft Visual Studio 调试控制台
cipher:
68, 1e, df, 34, d2, 06, 96, 5e, 86, b3, e9, 4f, 53, 6e, 42, 46
```

图 4: 库函数加密结果

2.2.3 时间对比

数据规模 (轮)	1000	5000	10000	30000	50000	80000	100000
openssl(s)	0.001	0.006	0.011	0.034	0.058	0.09	0.114
普通实现 (s)	0.016	0.078	0.155	0.501	0.881	1.359	1.606

表 1: 时间对比

```
unsigned char plaintext[] = { ... }
unsigned char key[]
    unsigned char plaintext[160000]
    联机搜索
    0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef,
    0xfe, 0xdc, 0xba, 0x98, 0x76, 0x54, 0x32, 0x10
};
Microsoft Visual Studio 调试控制台
加解密长明文用时:0.004s
```

图 5:

如图所示, 这是 10000 组标准长度,openssl 库加解密使用了 0.004s, 那么加解密的延迟为 4×10^{-7}s, 吞吐量为 4×10^{7} 字节。

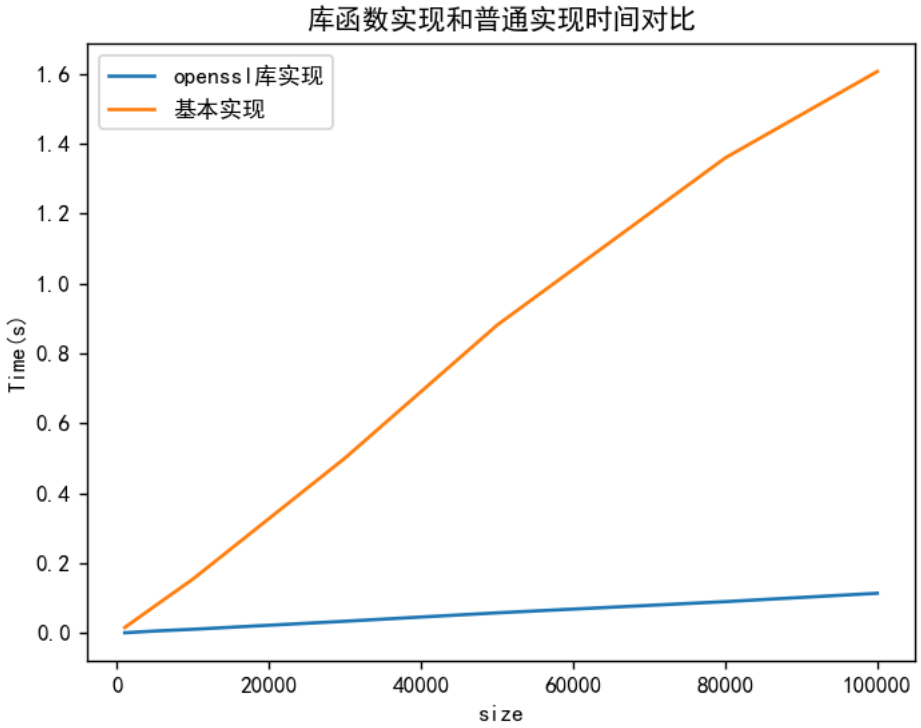

图 6: 库函数与普通实现对比

可见直接调用库函数是极其快速的, 比普通实现要稳定快 10 ~20 倍。

以上数据均为 for 循环加密同一组数据。这样相当于人为为明文分组，考虑到 for 循环会增加指令数，所以将长明文直接放在加密数组里。

图 7: for 加密

图 8: 长文本加密

可以看出, 在放弃 for 循环后, 在实际应用场景下,10000 组标准长度, 库函数的加密速度快了五倍。

3 SM4 算法优化

3.1 循环展开优化

由于在轮函数中，for 循环的存在会增加轮函数的延迟。所以我们将函数展开。并进行测时。

代码如下：

```
void encryption(uint32_t* ciphertext, uint32_t* plaintext_for_one_round) {
    uint32_t K[36];
    uint32_t X[36] = { 0 };
    X[0] = plaintext_for_one_round[0]; X[1] = plaintext_for_one_round[1];
        X[2] = plaintext_for_one_round[2]; X[3] = plaintext_for_one_round
        [3];
    for (int i = 4; i < 36; i++) {
        rK[i - 4] = K[i] = K[i - 4] ^ T_for_get_key_func(K[i - 3] ^ K[i -
            2] ^ K[i - 1] ^ CK[i - 4]);
    }
    X[4] = X[0] ^ T_for_roundfunc_func(X[1] ^ X[2] ^ X[3] ^ rK[0]);
    X[5] = X[1] ^ T_for_roundfunc_func(X[2] ^ X[3] ^ X[4] ^ rK[1]);
    X[6] = X[2] ^ T_for_roundfunc_func(X[3] ^ X[4] ^ X[5] ^ rK[2]);
    /*.............................略.......................*/
    ciphertext[3] = X[32] = X[28] ^ T_for_roundfunc_func(X[29] ^ X[30] ^ X
        [31] ^ rK[28]);
    ciphertext[2] = X[33] = X[29] ^ T_for_roundfunc_func(X[30] ^ X[31] ^ X
        [32] ^ rK[29]);
    ciphertext[1] = X[34] = X[30] ^ T_for_roundfunc_func(X[31] ^ X[32] ^ X
        [33] ^ rK[30]);
    ciphertext[0] = X[35] = X[31] ^ T_for_roundfunc_func(X[32] ^ X[33] ^ X
        [34] ^ rK[31]);
}
```

3.2 查表优化

S 盒操作为 x0,x1,x2,x3→S(x0),S(x1),S(x2),S(x3)，其中 xi 为 8bit 字。为了提升效率，可将 S 盒与后续的循环移位变换 L 合并，即:

L(S(x0),S(x1),S(x2),S(x3))=L(S(x0)«24) L(S(x1)«16) L(S(x2)«8) L(S(x3))

可定义 4 个 8bit→32bit 查找表 Ti:

T0(x)=L(S(x)«24)

T1(x)=L(S(x)«16)

T2(x)=L(S(x)«8)

T3(x)=L(S(x)) 节省后续的循环移位操作，大致操作如下:

通过移位取出 x0,x1,x2,x3

返回 $T0(x0) \oplus T1(x1) \oplus T2(x2) \oplus T3(x3)$

表项生成:

首先进行的操作是对 S 盒运算和 L 线性运算操作后的表项进行生成，这里首先要将 S 盒输出的 8bit 扩展为 32bit，然后直接进行移位而不是循环移位。由于 L 运算是线性扩展运算，因此 L 运算可以将移位和异或扩展到 4 个 S 盒的 8bit 输出，在线性扩展运算时我们采用循环移位的策略。

```
uint32_t L(uint32_t num){
    return num = Cycle_Shift_lift(num, 2) ^ Cycle_Shift_lift(num, 10) ^
       Cycle_Shift_lift(num, 18) ^ Cycle_Shift_lift(num, 24) ^ num;
    //using the cycle shift to extend the operation
}
//to generate a table
void generate_table()
{
    for (uint8_t i = 0; i < 256; i++){
        table0[i] = L(((uint32_t)S_replace(i)) << 24);
        table1[i] = L(((uint32_t)S_replace(i)) << 16);
        table2[i] = L(((uint32_t)S_replace(i)) << 8);
        table3[i] = L(((uint32_t)S_replace(i)));
    }//extend the uint8 with uint32 and do the L linear operation
```

```
}
```

接下来是利用生成的表进行加密运算的操作，这里将 S 盒和 L 运算合并为查表运算，其实体现了利用空间换取时间的思想。

```
//轮函数中的T函数查表优化
uint32_t T_for_roundfunc_use_Tbox_func(uint32_t K_for_t) {
    uint8_t* A = (uint8_t*)&K_for_t;
    return  Table0[A[3]] ^ Table1[A[2]] ^ Table2[A[1]] ^ Table3[A[0]];
}
//T表查询的多线程方法
void encryption_use_t_box(uint32_t* ciphertext, uint32_t*
    plaintext_for_one_round) {
    uint32_t K[36];
    uint32_t X[36] = { 0 };
    X[0] = plaintext_for_one_round[0]; X[1] = plaintext_for_one_round[1];
        X[2] = plaintext_for_one_round[2]; X[3] = plaintext_for_one_round
        [3];
    K[0] = MK[0] ^ FK[0]; K[1] = MK[1] ^ FK[1]; K[2] = MK[2] ^ FK[2]; K[3]
         = MK[3] ^ FK[3];
    for (int i = 4; i < 36; i++) {
        rK[i - 4] = K[i] = K[i - 4] ^ T_for_get_key_func(K[i - 3] ^ K[i -
            2] ^ K[i - 1] ^ CK[i - 4]);
        X[i] = X[i - 4] ^ T_for_roundfunc_use_Tbox_func(X[i - 1] ^ X[i -
            2] ^ X[i - 3] ^ rK[i - 4]);
    }
    ciphertext[0] = X[35]; ciphertext[1] = X[34]; ciphertext[2] = X[33];
        ciphertext[3] = X[32];
}
```

3.3 SIMD 指令集优化

上面我们实现了 32bit 的查找，将 SM4 的线性变换 L 和 S 盒合并为 4 个 $256\times32bit$ 的查找表。下面我们利用 intel 的 AVX 指令集可以实现并行查表。

3.3.1 _mm256_i32gather_epi32() 函数

在本节中最重要的就是 _mm256_i32gather_epi32() 函数的应用。下面给出官方定义:

Synopsis

```
__m256 _mm256_i32gather_ps (float const* base_addr, __m256i vindex, const int scale)
#include <immintrin.h>
Instruction: vgatherdps ymm, vm32x, ymm
CPUID Flags: AVX2
```

Description

Gather single-precision (32-bit) floating-point elements from memory using 32-bit indices. 32-bit elem
vindex (each index is scaled by the factor in scale). Gathered elements are merged into dst. scale

Operation

```
FOR j := 0 to 7
        i := j*32
        m := j*32
        addr := base_addr + SignExtend64(vindex[m+31:m]) * ZeroExtend64(scale) * 8
        dst[i+31:i] := MEM[addr+31:addr]
ENDFOR
dst[MAX:256] := 0
```

图 9: 官方定义

可以看出，该函数的作用是将 32 位元素从从 base_addr 开始的地址加载，并按 vindex 中的每个 32 位元素偏移（每个索引按比例因子缩放）。收集的元素被合并到 dst。所以我们可以通过该函数同时对八个 128 位的明文进行加密。

3.3.2 实现过程

首先将 n 组 128bit 明文消息记为 P_i，然后将 P_i 按照如下方式装载到 4 个 SIMD 寄存器 R_0 R_1 R_2 R_3 中，可以看到该过程就像发牌一样。

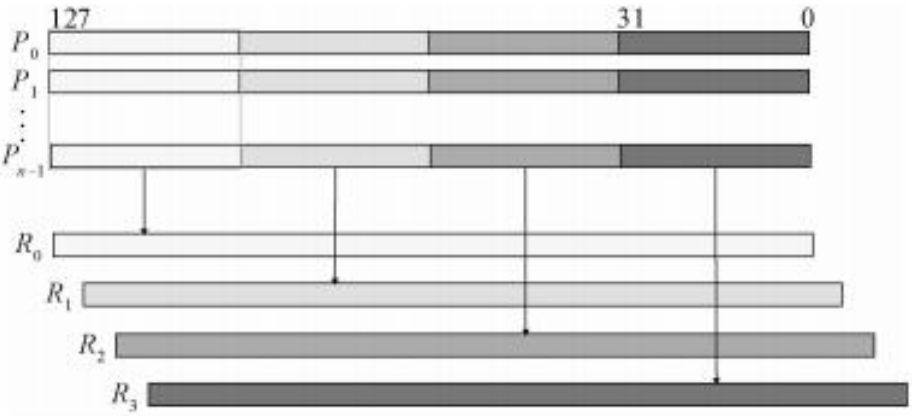

图 10: 装载方式

然记录将 n 组消息轮密钥装载到 SIMD 寄存器中. 代码如下:

```
    vindex = _mm256_setr_epi32(0, 4, 8, 12, 16, 20, 24, 28);
    R0 = _mm256_i32gather_epi32((int*)(plaintext_for_one_round), vindex,
        4);
    R1 = _mm256_i32gather_epi32((int*)(plaintext_for_one_round + 1),
        vindex, 4);
    R2 = _mm256_i32gather_epi32((int*)(plaintext_for_one_round + 2),
        vindex, 4);
    R3 = _mm256_i32gather_epi32((int*)(plaintext_for_one_round + 3),
        vindex, 4);
```

在轮函数 T 变换操作，我们可以通过掩码、移位、异或、查表等操作实现非线性变换 T.

$$\tau(R) = S0[R\&0\text{xff}] \oplus S1[(R \gg 8)\&0\text{xff}] \oplus S2[(R \gg 16)\&0\text{xff}] \oplus S3[R \gg 24]$$

生成表的方法和查表优化中表的生成相同。

下面我们给出 T 函数的 SIMD 代码

```
//simd中的T变换
__m256i SIMD_T_func(__m256i R) {
    __m256i mask, R_s0, R_s1, R_s2, R_s3;
    mask = _mm256_setr_epi32(0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0
        xff);
    R_s0 = _mm256_srli_epi32(R, 0);
    R_s1 = _mm256_srli_epi32(R, 8);
    R_s2 = _mm256_srli_epi32(R, 16);
    R_s3 = _mm256_srli_epi32(R, 24);
    R_s0 = _mm256_and_si256(R_s0, mask);
    R_s1 = _mm256_and_si256(R_s1, mask);
    R_s2 = _mm256_and_si256(R_s2, mask);
    R_s0 = _mm256_i32gather_epi32((int*)BOX0, R_s0, 4);
    R_s1 = _mm256_i32gather_epi32((int*)BOX1, R_s1, 4);
    R_s2 = _mm256_i32gather_epi32((int*)BOX2, R_s2, 4);
    R_s3 = _mm256_i32gather_epi32((int*)BOX3, R_s3, 4);
```

```
    R = _mm256_xor_si256(R_s0, R_s1);
    R = _mm256_xor_si256(R, R_s2);
    R = _mm256_xor_si256(R, R_s3);
    return R;
}
```

之后我们按照普通实现中获取轮密钥的方式获得所有轮密钥。获得轮密钥后，我们为了使用轮密钥对明文进行加密，也需要将轮密钥装载到一个 ___ 型变量 R_K 中。给出代码：

```
装载轮密钥
__m256i index = _mm256_setzero_si256();
    vindex = _mm256_setr_epi32
    (0, 4, 8, 12, 16, 20, 24, 28);
R_K = _mm256_i32gather_epi32((int*)(rK + 4 * i + 0), index, 4);
R_K = _mm256_i32gather_epi32((int*)(rK + 4 * i + 1), index, 4);
R_K = _mm256_i32gather_epi32((int*)(rK + 4 * i + 2), index, 4);
R_K = _mm256_i32gather_epi32((int*)(rK + 4 * i + 3), index, 4);
```

然后由图 1 的轮函数可知，R1、R2、R3 和轮密钥 R_K 经过 T 变换后异或并可以得到新的 R0；R0、R2、R3 和轮密钥 R_K 经过 T 变换后异或并可以得到新的 R1；R0、R1、R3 和轮密钥 R_K 经过 T 变换后异或并可以得到新的 R2；R1、R2、R0 和轮密钥 R_K 经过 T 变换后异或并可以得到新的 R4, 经过八轮循环后我最后我们将最后一轮所得到的 R3，R2，R1，R0，按照收牌的顺序插入密文。按照该规律我们给出轮函数的代码：

```
//SIMD加密函数
void encryption_SIMD(uint32_t* ciphertext, uint32_t*
   plaintext_for_one_round) {
    __m256i vindex, R0, R1, R2, R3,R_K;
    __m256i index = _mm256_setzero_si256();
    vindex = _mm256_setr_epi32
    (0, 4, 8, 12, 16, 20, 24, 28);
    R0 = _mm256_i32gather_epi32((int*)(plaintext_for_one_round), vindex,
       4);
    R1 = _mm256_i32gather_epi32((int*)(plaintext_for_one_round + 1),
```

```
        vindex, 4);
    R2 = _mm256_i32gather_epi32((int*)(plaintext_for_one_round + 2),
        vindex, 4);
    R3 = _mm256_i32gather_epi32((int*)(plaintext_for_one_round + 3),
        vindex, 4);
    for (int i = 0; i < 8; ++i)
    {
        R_K = _mm256_i32gather_epi32((int*)
            (rK + 4 * i + 0), index, 4);
        R_K = _mm256_xor_si256(R_K, R1);
        R_K = _mm256_xor_si256(R_K, R2);
        R_K = _mm256_xor_si256(R_K, R3);
        R_K = SIMD_T_func(R_K);
        R0 = _mm256_xor_si256(R0, R_K);
        R_K = _mm256_i32gather_epi32((int*)
            (rK + 4 * i + 1), index, 4);
        R_K = _mm256_xor_si256(R_K, R0);
        R_K = _mm256_xor_si256(R_K, R2);
        R_K = _mm256_xor_si256(R_K, R3);
        R_K = SIMD_T_func(R_K);
        R1 = _mm256_xor_si256(R1, R_K);
        R_K = _mm256_i32gather_epi32((int*)
            (rK + 4 * i + 2), index, 4);
        R_K = _mm256_xor_si256(R_K, R1);
        R_K = _mm256_xor_si256(R_K, R0);
        R_K = _mm256_xor_si256(R_K, R3);
        R_K = SIMD_T_func(R_K);
        R2 = _mm256_xor_si256(R2, R_K);
        R_K = _mm256_i32gather_epi32((int*)
            (rK + 4 * i + 3), index, 4);
        R_K = _mm256_xor_si256(R_K, R2);
        R_K = _mm256_xor_si256(R_K, R1);
        R_K = _mm256_xor_si256(R_K, R0);
        R_K = SIMD_T_func(R_K);
        R3 = _mm256_xor_si256(R3, R_K);
```

```
41      }
42      _mm256_storeu_si256((__m256i*)ciphertext + 0, _mm256_unpackhi_epi64(
            _mm256_unpacklo_epi32(R3, R2), _mm256_unpacklo_epi32(R1, R0)));
43      _mm256_storeu_si256((__m256i*)ciphertext + 1, _mm256_unpackhi_epi64(
            _mm256_unpacklo_epi32(R3, R2), _mm256_unpacklo_epi32(R1, R0)));
44      _mm256_storeu_si256((__m256i*)ciphertext + 2, _mm256_unpackhi_epi64(
            _mm256_unpacklo_epi32(R3, R2), _mm256_unpacklo_epi32(R1, R0)));
45      _mm256_storeu_si256((__m256i*)ciphertext + 3, _mm256_unpackhi_epi64(
            _mm256_unpacklo_epi32(R3, R2), _mm256_unpacklo_epi32(R1, R0)));
46  }
```

可以看到我们实现了同时加密 8 组 128 位的明文。

3.4 时间对比

为了测量单轮加密的延迟，我们让三种优化方式只加密一个 128bit 的明文，让它加密 1000 次，记录时间后让所记录的时间除以 1000。得到下面结果。

	普通实现	循环展开	查表优化	simd 指令
延迟 (µs)	8	7.1	6.01	5.1

表 2: 延迟

通过延迟对比我们可以看出，经过查表优化后虽然牺牲了部分空间，但是换取了 20% 多的效率提升，在加上 SIMD 优化后加解密的效率更高.

4 实现 SM4 的 ECB 模式加密

一个加密算法最终还是要实现对长文本的加密。所以我们在本节中用上面的几种加密算法 (1. 普通实现，2. 循环展开，3. 查表优化，4.simd 指令优化)。由于轮密钥用的都为同一套，所以我们把加密算法中的求轮密钥的步骤提出来放在加密算法的外面。

4.1 普通实现的 ECB 模式加密

代码如下：

```
void SM4_ECB() {
    //获取轮密钥
    K[0] = MK[0] ^ FK[0]; K[1] = MK[1] ^ FK[1]; K[2] = MK[2] ^ FK[2]; K[3]
        = MK[3] ^ FK[3];
    for (int i = 4; i < 36; i++) {
        rK[i-4]=K[i] = K[i-4] ^ T_for_get_key_func(K[i-3] ^ K[i-2] ^ K[i
            -1] ^ CK[i-4]);
    //每一轮的明文
    uint32_t* point_plaintext = new uint32_t(4);
    uint32_t* point_ciphertext = new uint32_t(4);
    for (int i = 0; i < sizeof(plaintext) / sizeof(int); i += 4) {
        uint32_t* point_plaintext = &plaintext[i];
        uint32_t* point_ciphertext =&ciphertext_32[i];
        //加密算法中无求轮密钥的步骤
        encryption(point_ciphertext, point_plaintext);
    }
}
```

4.2 循环展开的 ECB 模式加密

代码如下:

```
void SM4_ECB_open_cycle() {
    //获取轮密钥
    K[0] = MK[0] ^ FK[0]; K[1] = MK[1] ^ FK[1]; K[2] = MK[2] ^ FK[2]; K[3]
        = MK[3] ^ FK[3];
    for (int i = 4; i < 36; i++) {
        rK[i-4]=K[i] = K[i-4] ^ T_for_get_key_func(K[i-3] ^ K[i-2] ^ K[i
            -1] ^ CK[i-4]);
    //每一轮的明文
    uint32_t* point_plaintext = new uint32_t(4);
    uint32_t* point_ciphertext = new uint32_t(4);
    for (int i = 0; i < sizeof(plaintext) / sizeof(int); i += 4) {
        uint32_t* point_plaintext = &plaintext[i];
        uint32_t* point_ciphertext =&ciphertext_32[i];
        //加密算法中无求轮密钥的步骤
```

```
13          encryption_open_cycle(point_ciphertext, point_plaintext);
14      }
15  }
```

4.3 查表优化的 ECB 模式加密

代码如下：

```
1  void SM4_ECB_use_T_box() {
2      //每一轮的明文
3      uint32_t K[36];
4      K[0] = MK[0] ^ FK[0]; K[1] = MK[1] ^ FK[1]; K[2] = MK[2] ^ FK[2]; K[3]
           = MK[3] ^ FK[3];
5      for (int i = 4; i < 36; i++) {
6          rK[i - 4] = K[i] = K[i - 4] ^ T_for_get_key_func(K[i - 3] ^ K[i -
               2] ^ K[i - 1] ^ CK[i - 4]);
7      }
8      uint32_t* point_plaintext = new uint32_t(4);
9      uint32_t* point_ciphertext = new uint32_t(4);
10     for (int i = 0; i < sizeof(plaintext) / sizeof(int); i += 4) {
11         uint32_t* point_plaintext = &plaintext[i];
12         uint32_t* point_ciphertext = &ciphertext_32[i];
13         encryption_use_t_box(point_ciphertext, point_plaintext);
14     }
15 }
```

4.4 simd 的 ECB 模式加密

由于 simd 加速可以让八组 128bit 的明文进行并行加密，所以我们在加密长文本时跳步为 +32；

代码如下：

```
1  void SM4_ECB_for_SIMD() {
2      //每一轮的明文
3      uint32_t K[36];
```

```
    K[0] = MK[0] ^ FK[0]; K[1] = MK[1] ^ FK[1]; K[2] = MK[2] ^ FK[2]; K[3]
        = MK[3] ^ FK[3];
    //获取轮密钥
    for (int i = 4; i < 36; i++) {
        rK[i - 4] = K[i] = K[i - 4] ^ T_for_get_key_func(K[i - 3] ^ K[i -
            2] ^ K[i - 1] ^ CK[i - 4]);
    }
    uint32_t* point_plaintext = new uint32_t(32);
    uint32_t* point_ciphertext = new uint32_t(32);
    for (int i = 0; i < sizeof(plaintext) / sizeof(int); i += 32) {
        uint32_t* point_plaintext = &plaintext[i];
        uint32_t* point_ciphertext = &ciphertext_32[i];
        encryption_SIMD(point_ciphertext, point_plaintext);
    }
}
```

4.5 普通实现的多线程 ECB 模式加密

为了进一步加快加密的速度，我们引入了多线程来实现多组 128bit 明文的并行加密。其中我们定义线程数为 8。代码如下：

```
//SM4 ECB模式下的使用多线程的方法。
void SM4_ECB_for_thread(int num) {
    //每一轮的明文
    uint32_t* point_plaintext = new uint32_t(4);
    uint32_t* point_ciphertext = new uint32_t(4);
    for (int i = num*4; i < sizeof(plaintext) / sizeof(int); i += 4*4) {
        uint32_t* point_plaintext = &plaintext[i];
        uint32_t* point_ciphertext = &ciphertext_32[i];
        encryption(point_ciphertext, point_plaintext);
    }
}

//多线程main函数：
void SM4_ECB_for_thread_main() {
```

```
    uint32_t K[36];
    K[0] = MK[0] ^ FK[0]; K[1] = MK[1] ^ FK[1]; K[2] = MK[2] ^ FK[2]; K[3]
        = MK[3] ^ FK[3];
    for (int i = 4; i < 36; i++) {
        rK[i - 4] = K[i] = K[i - 4] ^ T_for_get_key_func(K[i - 3] ^ K[i -
            2] ^ K[i - 1] ^ CK[i - 4]);
    }
    thread t1(SM4_ECB_for_thread, 0);
    thread t2(SM4_ECB_for_thread, 1);
    thread t3(SM4_ECB_for_thread, 2);
    thread t4(SM4_ECB_for_thread, 3);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
}
```

4.6 时间对比与结果分析

我们用上述算法加密了长度为 320000,3200000,32000000,320000000,3200000000bit 的长文本并记录了所用的时间。

数据规模 (bit)	320000	3200000	32000000	320000000	3200000000
普通实现 (s)	0.013	0.121	1.207	11.55	116.627
循环展开 (s)	0.012	0.114	1.172	10.351	111.112
查表优化 (s)	0.001	0.011	0.105	1.071	10.841
simd 指令 (s)	0.001	0.007	0.059	0.601	6.640
多线程 (s)	0.009	0.029	0.241	2.484	24.988

表 3: 时间对比

4.7 时间对比

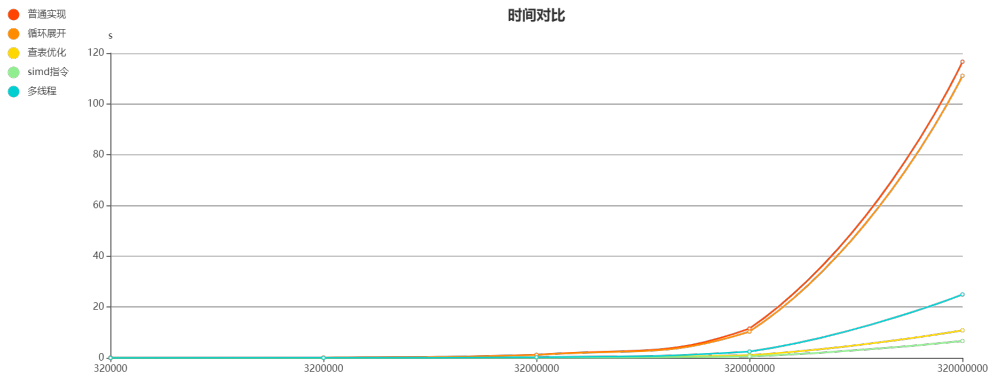

图 11: 时间对比图

可以发现大体的运行速度对比为 simd 优化 > 查表优化 > 多线程 > 循环展开 > 普通实现。

循环展开的速度虽然快于普通实现但是相差不大。对于这种现象我们认为虽然去除了 for 循环中增加的操作，但是这个操作对整个加密过程的影响不大。

simd 优化和多线程虽然同样是对 8 组 128bit 的明文并行加密，但是 simd 快于多线程，我们认为有如下两点原因：1.simd 优化采用了查表的方式进行初步加密。2. 但多线程中需要对线程进行管理，也使得速度差距进一步扩大。

总体来说，对优化结果的测量符合我们的预期。

下面给出吞吐量和加速比的对比图。

数据规模 (bit)	普通实现	循环展开	查表优化	simd 指令	多线程
吞吐量 (MB/s)	6.60	7.37	71.23	126.94	30.71
加速比	1	1.12	10.79	19.23	4.65

表 4: 吞吐量及加速比

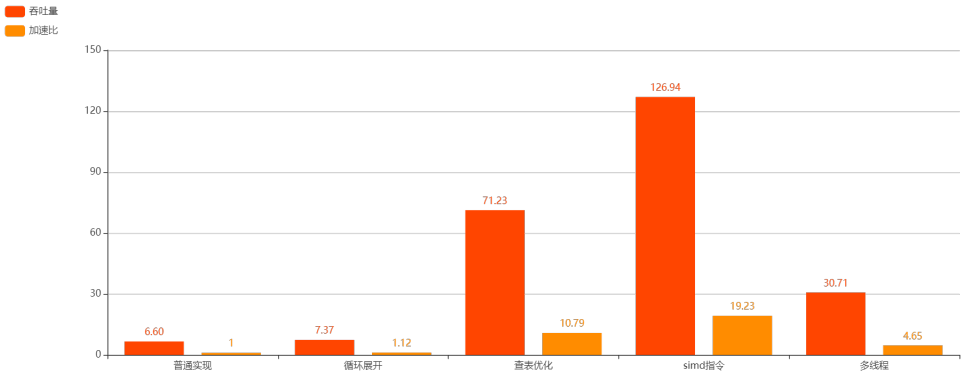

图 12: 对比图

A 小组分工

成员	分工
李卓群	部分实验报告撰写，单线程下 SM4 算法的优化和时间测定，优化方案对比
刘力豪	部分实验报告撰写，sm4 算法的一般优化、查表优化的实现、simd 指令优化。
马 洋	部分实验报告撰写，sm4 算法的普通实现，循环展开，simd 指令优化,ECB 模式多线程。
杨文涛	实验报告撰写,SM4 普通实现,openssl 库实现
孔伟骁	部分实验报告撰写,SM4 普通实现，SM4 查表优化，一般优化实现

参考文献

[1] http://www.gmbz.org.cn/main/viewfile/20180108015408199368.html

[2] [美]Bryant,R.E. Computer Systems:A Programmer's Perspective [M] 北京：机械工业出版社 2016

[3] http://html.rhhz.net/ZGKXYDXXB/20180205.htm [M] 北京：机械工业出版社 2016