SM4 算法 AES-NI 指令集优化实验报告

实验目的：

本实验旨在探索利用现代 CPU 的 AES-NI 指令集优化 SM4 分组密码算法的实现。使用查表法优化，存在缓存侧信道攻击风险。AES-NI 指令集提供硬件级加密加速，通过数学变换可将 SM4 的 S 盒操作映射到 AES-NI 指令，实现高效安全的加密加速。

关键技术原理

2.1 AES-NI 指令集基础

AES-NI 是 Intel 和 AMD 处理器上的指令集扩展，包含 6 条专用指令：

AESENC/AESENCLAST：AES 加密轮函数

AESDEC/AESDECLAST：AES 解密轮函数

AESKEYGENASSIST：AES 密钥扩展

AESIMC：逆列混合变换

这些指令直接在硬件层面实现 AES 的 S 盒和列混合操作，单条指令只需 1-3 个时钟周期。

2.2 SM4 与 AES 的数学关联

SM4 的 S 盒可以表示为复合变换：

$SM4\_SBox(x) = L^{-1}(AES\_SBox(L(x)))$

使用 AES-NI 指令实现 AES_SBox

用矩阵乘法实现 L 和 L^{-1}

线性变换优化

SM4 的线性变换：

$\tau(x)=x\oplus(x\lll2)\oplus(x\lll10)\oplus(x\lll18)\oplus(x\lll24)$

在代码中可以通过 SIMD 指令并行计算。

具体实现：

1：头文件：

包含标准 I/O、内存管理、时间处理等必要库

```
#include <stdio.h>          // 标准输入输出
#include <stdlib.h>         // 标准库函数
```

```
#include <iostream>        // C++输入输出流
#include <iomanip>         // 格式化输出
#include <stdint.h>        // 标准整数类型
#include <chrono>          // 时间库
#include <immintrin.h>     // SIMD 指令集
```

2：计时器类实现

```
void UpDate() {
        _begin = std::chrono::steady_clock::now();
    }

    double GetSecond() {
        _end = std::chrono::steady_clock::now();
        std::chrono::duration<double> temp =
            std::chrono::duration_cast<std::chrono::duration<double>>(_end - _begin);
        return temp.count();
    }
```

UpDate()：记录当前时间作为开始时间点

GetSecond()：计算从上次 UpDate()到当前的时间差（秒）

使用 std::chrono::steady_clock 保证计时稳定不受系统时间调整影响

3：SM4 算法核心定义：

```
typedef struct _SM4_Key {
    uint32_t rk[32]; // 32 轮密钥
} SM4_Key;

static uint32_t FK[4] = { 0xa3b1bac6, 0x56aa3350, 0x677d9197, 0xb27022dc };
```

rk[32]：存储 32 轮轮密钥

FK：系统参数，用于密钥扩展的初始异或值

4：轮常数 CK 定义

每轮使用不同的 CK 值，基于特定算法生成，提供非线性特性

```
static uint32_t CK[32] = {
    0x00070e15, 0x1c232a31, 0x383f464d, 0x545b6269, 0x70777e85, 0x8c939aa1,
    // … 共 32 个常数
};
```

5：循环左移宏，实现 32 位数的循环左移

```
#define shift32(value, shift) ((value << shift) | value >> (32 - shift))
```

value << shift：左移指定位数

value >> (32 - shift)：右移补位

使用位或|组合两部分

用于密钥扩展和轮函数中的线性变换

6：密钥初始化函数

将 128 位密钥扩展为 32 个轮密钥，

初始化：将 16 字节密钥分为 4 个 32 位字，每个与 FK 异或

轮迭代：

计算临时值：tmp = k[1] ^ k[2] ^ k[3] ^ CK[i]

S 盒变换：对 tmp 的 4 个字节分别查表

线性变换：k[0] ^ tmp ^ rotl(tmp,13) ^ rotl(tmp,23)

状态更新：移位寄存器

7：数字打包：

重组 SIMD 寄存器中的数据，将 4 个 128 位寄存器(a,b,c,d)中的数据重组

每个寄存器包含 4 个 32 位整数，结果寄存器包含：

a[0], b[0], c[0], d[0]（最低 32 位到最高 32 位）

```
#define MM_PACK0_EPI32(a, b, c, d) \
    _mm_unpacklo_epi64(_mm_unpacklo_epi32(a, b), _mm_unpacklo_epi32(c, d))
```

8：实现 4×4 矩阵乘法（在 GF(2)上）

```
__m128i tmp1, tmp2;
    __m128i andMask = _mm_set1_epi32(0x0f0f0f0f);
    tmp2 = _mm_srli_epi16(x, 4);
    tmp1 = _mm_and_si128(x, andMask);
    tmp2 = _mm_and_si128(tmp2, andMask);
    tmp1 = _mm_shuffle_epi8(lowerMask, tmp1);
    tmp2 = _mm_shuffle_epi8(higherMask, tmp2);
    tmp1 = _mm_xor_si128(tmp1, tmp2);
```

分离高低 4 位：

tmp1 = x & 0x0F（低 4 位）

tmp2 = (x >> 4) & 0x0F（高 4 位）

查表变换：

tmp1 = lowerMask[tmp1]

tmp2 = higherMask[tmp2]

合并结果：tmp1 ^ tmp2

9：实现特定矩阵 ATA 的乘法：

```
static __m128i MulMatrixATA(__m128i x) {
    __m128i higherMask = _mm_set_epi8(...);
    __m128i lowerMask = _mm_set_epi8(...);
    return MulMatrix(x, higherMask, lowerMask);
}
```

预定义掩码对应数学推导的变换矩阵，用于 SM4 S 盒的逆仿射变换

10：**S 盒实现（核心优化）**

```
static __m128i SM4_SBox(__m128i x) {
    __m128i MASK = _mm_set_epi8(...);
    x = _mm_shuffle_epi8(x, MASK);
    x = AddTC(MulMatrixTA(x));
```

```
    x = _mm_aesenclast_si128(x, _mm_setzero_si128());
    return AddATAC(MulMatrixATA(x));
}
```

字节重排：调整字节顺序适配 AES 指令

前变换：MulMatrixTA + AddTC

AES S 盒：_mm_aesenclast_si128

后变换：MulMatrixATA + AddATA

11：核心处理函数：

```
static void SM4_AESNI_do(uint8_t* in, uint8_t* out, SM4_Key* sm4_key, int enc) {
    // 加载 128 字节数据（8 个分组）
    __m128i Tmp[4] = {
        _mm_loadu_si128((const __m128i*)(in + 0)),
        // ... 其他分组
    };

    // 字节序调整掩码（大端转小端）
    __m128i vindex = _mm_setr_epi8(3,2,1,0,7,6,5,4,11,10,9,8,15,14,13,12);

    // 数据重组：将 8 个分组相同位置的字打包
    X[0] = MM_PACK0_EPI32(Tmp[0], Tmp[1], Tmp[2], Tmp[3]);
    // ... 其他位置

    // 调整字节序（大端转小端）
    X[0] = _mm_shuffle_epi8(X[0], vindex);
    // ... 其他寄存器

    // 32 轮迭代
```

```
    for (int i = 0; i < 32; i++) {
        // 轮密钥选择（加密/解密）
        __m128i k = _mm_set1_epi32(enc ? sm4_key->rk[31-i] : sm4_key->rk[i]);

        // 轮函数
        Tmp[0] = MM_XOR4(X[1], X[2], X[3], k); // 轮密钥加
        Tmp[0] = SM4_SBox(Tmp[0]);                    // S 盒变换
        Tmp[0] = MM_XOR6(X[0], Tmp[0],             // 线性变换
            MM_ROTL_EPI32(Tmp[0], 2),
            MM_ROTL_EPI32(Tmp[0], 10),
            MM_ROTL_EPI32(Tmp[0], 18),
            MM_ROTL_EPI32(Tmp[0], 24));

        // 状态更新
        X[0] = X[1];
        X[1] = X[2];
        X[2] = X[3];
        X[3] = Tmp[0];
    }

    // 结果重组和存储
    _mm_storeu_si128((__m128i*)(out + 0), MM_PACK0_EPI32(X[3], X[2], X[1], X[0]));
    // ... 其他位置
}
```

数据加载：加载 128 字节（8 个分组）

数据重组：将 8 个分组的相同位置字打包到 SIMD 寄存器

字节序调整：大端转小端（SM4 使用大端序）

轮迭代:

轮密钥加：异或轮密钥

S 盒变换：使用 AES-NI 加速

线性变换：异或和循环移位

结果重组：将 SIMD 数据重组为分组数据

存储结果：写回内存

运行结果：

```
Ciphertext:
        44 92 f0 a8 ae 3c cd b6 b5 d9 dc b8 21 af da f4
        37 50 a8 1d 9c 74 d9 06 a7 2d 53 4e e2 39 a3 ba
        37 50 a8 1d 9c 74 d9 06 a7 2d 53 4e e2 39 a3 ba
        37 50 a8 1d 9c 74 d9 06 a7 2d 53 4e e2 39 a3 ba

Plaintext:
        02 21 45 47 89 ab cd ef be de ba 08 70 58 11 ae
        00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00


Time for encryption of SM4 with AES-NI:  0.0000047 s
```