SM3 算法实现与 SIMD 优化实验报告

SM3 算法概述

SM3 是中国国家密码管理局发布的一种密码杂凑算法，用于生成消息摘要（哈希值）。其输出长度为 256 位（32 字节），设计安全性与 SHA-256 相当。SM3 算法主要包括以下步骤：

消息填充、消息扩展、迭代压缩

基本运算

循环左移：rol(x, j) = (x << j) | (x >> (32 - j))

置换函数：

P0(x) = x ⊕ rol(x, 9) ⊕ rol(x, 17)

P1(x) = x ⊕ rol(x, 15) ⊕ rol(x, 23)

布尔函数：

FF0(x,y,z) = x ⊕ y ⊕ z (前 16 轮)

FF1(x,y,z) = (x & y) | (x & z) | (y & z) (后 48 轮)

GG0(x,y,z) = x ⊕ y ⊕ z (前 16 轮)

GG1(x,y,z) = (x & y) | ((~x) & z) (后 48 轮)

消息填充

1. 在消息末尾添加一个'1'比特（即字节 0x80）
2. 添加若干个 0 比特，直到消息长度满足（填充后总长度 ≡ 448 mod 512）
3. 在最后 64 位（8 字节）添加消息长度的二进制表示（大端序）

消息扩展

将每个 512 位的消息块扩展为 132 个字（每个字 32 位）：

前 16 个字直接取自消息块

后续的字通过以下公式计算：W[j] = P1(W[j-16] ⊕ W[j-9] ⊕ rol(W[j-3], 15)) ⊕ rol(W[j-13], 7) ⊕ W[j-6]

额外计算 64 个 W'值：W1[j] = W[j] ⊕ W[j+4]

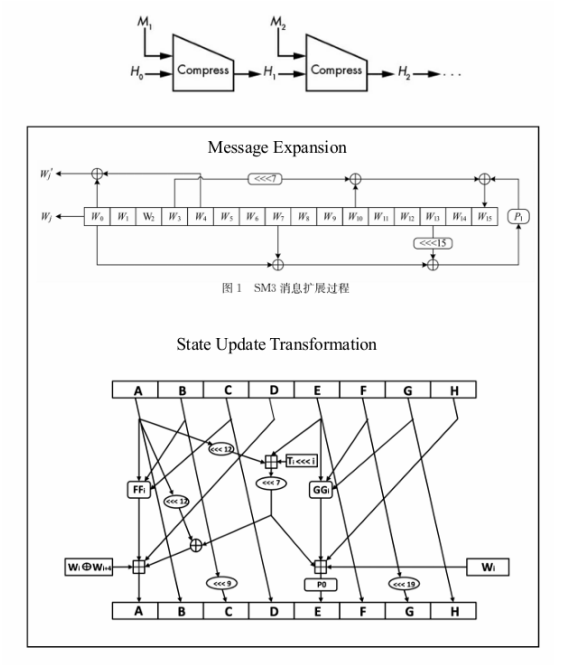

图 1　SM3 消息扩展过程

迭代压缩

SM3 使用 256 位的状态寄存器（8 个 32 位变量 A、B、C、D、E、F、G、H）进行迭代压缩。对于每个消息块，进行 64 轮迭代，每轮更新状态寄存器：

$SS1=rol(rol(A,12)+E+rol(T[j],j),7)\ SS2=SS1\oplus rol(A,12)\ TT1=FF(A,B,C)+D+SS2+W1[j]\ TT2=GG(E,F,G)+H+SS1+W[j]$

代码实现：

数据结构

```
typedef struct sm3_ctx_t {
    uint32_t digest[8];        // 8 个 32 位状态寄存器
    int nblocks;               // 已处理的块数
    uint8_t block[64];         // 当前处理的块
    int num;                   // 当前块中已填充的字节数
} sm3_ctx;
```

核心函数实现

```
void sm3_init(sm3_ctx* ctx) {
    ctx->digest[0] = 0x7380166F;
    ctx->digest[1] = 0x4914B2B9;
```

```
    // ... 其他初始值
    ctx->nblocks = 0;
    ctx->num = 0;
}
void sm3_update(sm3_ctx* ctx, const uint8_t* data, size_t dlen) {
    // 处理当前块中的剩余空间
    if (ctx->num) {
        unsigned int left = 64 - ctx->num;
        if (dlen < left) {
            memcpy(ctx->block + ctx->num, data, dlen);
            ctx->num += dlen;
            return;
        } else {
            memcpy(ctx->block + ctx->num, data, left);
            sm3_compress(ctx->digest, ctx->block);
            ctx->nblocks++;
            data += left;
            dlen -= left;
        }
    }

    // 处理完整块
    while (dlen >= 64) {
        sm3_compress(ctx->digest, data);
        ctx->nblocks++;
        data += 64;
        dlen -= 64;
```

```
    }

    // 保存剩余数据
    ctx->num = dlen;
    if (dlen) {
        memcpy(ctx->block, data, dlen);
    }
}
void sm3_final(sm3_ctx* ctx, uint8_t* digest) {
    // 添加填充位
    ctx->block[ctx->num] = 0x80;

    // 填充 0
    if (ctx->num + 9 <= 64) {
        memset(ctx->block + ctx->num + 1, 0, 64 - ctx->num - 9);
    } else {
        memset(ctx->block + ctx->num + 1, 0, 64 - ctx->num - 1);
        sm3_compress(ctx->digest, ctx->block);
        memset(ctx->block, 0, 64 - 8);
    }

    // 添加长度
    uint64_t bit_len = (ctx->nblocks * 512) + (ctx->num * 8);
    uint64_t* count = (uint64_t*)(ctx->block + 56);
    *count = byte_swap64(bit_len);

    // 处理最后一个块
```

```
        sm3_compress(ctx->digest, ctx->block);

        // 输出结果
        for (int i = 0; i < 8; i++) {
            ((uint32_t*)digest)[i] = byte_swap32(ctx->digest[i]);
        }
}
static void sm3_compress(uint32_t digest[8], const uint8_t block[64]) {
        uint32_t W[68], W1[64];
        const uint32_t* pblock = (const uint32_t*)(block);

        // 消息扩展
        for (int j = 0; j < 16; j++) {
            W[j] = byte_swap32(pblock[j]);
        }
        for (int j = 16; j < 68; j++) {
            W[j] = P1(W[j-16] ^ W[j-9] ^ rol(W[j-3], 15)) ^ rol(W[j-13], 7) ^ W[j-6];
        }
        for (int j = 0; j < 64; j++) {
            W1[j] = W[j] ^ W[j+4];
        }

        // 初始化状态
        uint32_t A = digest[0], B = digest[1], C = digest[2], D = digest[3];
        uint32_t E = digest[4], F = digest[5], G = digest[6], H = digest[7];

        // 迭代计算
```

```
for (int j = 0; j < 64; j++) {
        uint32_t T_val = (j < 16) ? 0x79CC4519 : 0x7A879D8A;
        uint32_t SS1 = rol(rol(A, 12) + E + rol(T_val, j), 7);
        uint32_t SS2 = SS1 ^ rol(A, 12);
        uint32_t TT1 = (j < 16) ?
            FF0(A, B, C) + D + SS2 + W1[j] :
            FF1(A, B, C) + D + SS2 + W1[j];
        uint32_t TT2 = (j < 16) ?
            GG0(E, F, G) + H + SS1 + W[j] :
            GG1(E, F, G) + H + SS1 + W[j];

        // 更新状态
        D = C;
        C = rol(B, 9);
        B = A;
        A = TT1;
        H = G;
        G = rol(F, 19);
        F = E;
        E = P0(TT2);
}

// 更新摘要
digest[0] ^= A;
digest[1] ^= B;
digest[2] ^= C;
digest[3] ^= D;
```

```
    digest[4] ^= E;
    digest[5] ^= F;
    digest[6] ^= G;
    digest[7] ^= H;
}
```

消息扩展：将 16 个字扩展为 68 个字

W1 计算：计算 W1[j] = W[j] ^ W[j+4]

迭代计算：64 轮迭代更新状态寄存器

状态更新：每轮更新所有状态寄存器

更新摘要：将最终状态与原始状态异或

SIMD 优化实现分析

优化思路

1. **并行处理**：使用 SIMD 指令同时处理多个数据
2. **减少循环**：将串行操作转换为并行操作
3. **内存优化**：减少内存访问次数
4. **指令优化**：使用高效指令替代复杂操作

关键技术细节

SIMD 辅助宏

```
#define simd_rol(x, k) _mm_or_si128(_mm_slli_epi32(x, k), _mm_srli_epi32(x, 32 - k))
```

使用 SSE 指令实现 SIMD 循环左移

同时处理 4 个 32 位整数的循环左移

宏定义节省进入函数压栈所耗费的时间

消息扩展优化

在消息扩展时，我们可以依靠 SIMD 指令集在一个时钟周期内对多个数据进行处理。但由于 sm3 在计算时需要 W_i-3 一项，这就导致 SIMD 原有的效果不能完全实现。

```
// 使用 SIMD 计算 W1
for (int i = 0; i < 64; i += 4) {
```

```
        __m128i w0_i = _mm_loadu_si128((__m128i*)(W0 + i));
        __m128i w0_i4 = _mm_loadu_si128((__m128i*)(W0 + i + 4));
        __m128i w1 = _mm_xor_si128(w0_i, w0_i4);
        _mm_storeu_si128((__m128i*)(W1 + i), w1);
}
```

并行计算：同时计算 4 个 W1 值

内存优化：减少内存访问次数

指令优化：使用_mm_xor_si128 实现高效异或

内置函数优化

```
// 字节交换
#define byte_swap32(x) __builtin_bswap32(x)
#define byte_swap64(x) __builtin_bswap64(x)
```

使用编译器内置函数优化字节交换

运行时间统计：

```
// 性能测试代码
int main() {
    // ... 初始化

    // 预热缓存
    sm3(message, len, hash);

    // 性能测试
    int test_runs = 1000;
    double total_time = 0.0;

    for (int i = 0; i < test_runs; i++) {
        start_time = get_current_time();
```

```
        sm3(message, len, hash);
        end_time = get_current_time();
        total_time += (end_time - start_time);
    }

    // 输出结果
    printf("Average Time: %.9f seconds\n", total_time / test_runs);
    printf("Throughput: %.2f MB/s\n", (len / (total_time / test_runs)) / (1024 * 1024));
}
```

最终效果：

sm3：

```
SM3 Hash: c75d9e3bb13f9f2890e55d5419ec430482cdc9d3133a282b5f5de20349973c3b

Performance Metrics:
Test Runs: 1000
Total Time: 0.001588 seconds
Average Time: 0.000001588 seconds
Throughput: 7.21 MB/s
```

Sm3_simd:

```
SM3 Hash: c75d9e3bb13f9f2890e55d5419ec430482cdc9d3133a282b5f5de20349973c3b

Performance Metrics:
Test Runs: 1000
Total Time: 0.001118 seconds
Average Time: 0.000001118 seconds
Throughput: 10.24 MB/s
```

通过 SIMD 指令集优化，在保持算法正确性的前提下，显著提升了 SM3 哈希计算的性能。主要优化点包括：

消息扩展并行化：使用 SIMD 指令加速 W1 计算

高效指令使用：利用内置函数优化字节交换

内存访问优化：对齐加载提高缓存命中率

取得了较好的结果，平均执行时间减少百分之 30