Project4 SM3算法的实现和优化

密码22.1 张靖陶 202200460011

1. SM3算法结构

消息填充（Padding）：使消息长度扩展为 len ≡ 448 mod 512，最后再追加 64 位的原始消息长度。保证最终消息长度为 512 的倍数。

消息分组（Grouping）：将填充后的消息按 512 位（64 字节）为一组，划分为 n 个分组：B(0), B(1), ..., B(n-1)。

消息扩展（Message Expansion）：每个分组扩展出 132 个字（W0W67, W′0W′63）使用两个置换函数 P0 和 P1，配合移位和异或等操作生成更多中间字。

压缩函数（Compression Function）：初始向量 IV（8 个 32 位字）参与压缩。每组消息都与前一组的中间值链式压缩，64 轮迭代。每轮计算用到了轮常数 Tj（前 16 轮与后 48 轮常数不同）。

最终输出:所有分组处理完后，输出 256 位摘要值。

1. 代码实现

|  |
| --- |
| #include <iostream> |
| #include <vector> |
| #include <cstring> |
| #include <iomanip> |
| #include <chrono> |
|  |
| using namespace std; |
| using namespace std::chrono; |
|  |
| typedef uint8\_t u8; |
| typedef uint32\_t u32; |
| typedef uint64\_t u64; |
|  |
| u32 Tj(int j) { |
| return (j < 16) ? 0x79CC4519 : 0x7A879D8A; |
| } |
|  |
| u32 left\_rotate(u32 x, int n) { |
| return ((x << n) | (x >> (32 - n))) & 0xFFFFFFFF; |
| } |
|  |
| u32 P0(u32 x) { |
| return x ^ left\_rotate(x, 9) ^ left\_rotate(x, 17); |
| } |
|  |
| u32 P1(u32 x) { |
| return x ^ left\_rotate(x, 15) ^ left\_rotate(x, 23); |
| } |
|  |
| u32 FF(u32 x, u32 y, u32 z, int j) { |
| return (j < 16) ? (x ^ y ^ z) : ((x & y) | (x & z) | (y & z)); |
| } |
|  |
| u32 GG(u32 x, u32 y, u32 z, int j) { |
| return (j < 16) ? (x ^ y ^ z) : ((x & y) | (~x & z)); |
| } |
|  |
| // 初始向量 |
| u32 IV[8] = { |
| 0x7380166F, 0x4914B2B9, 0x172442D7, 0xDA8A0600, |
| 0xA96F30BC, 0x163138AA, 0xE38DEE4D, 0xB0FB0E4E |
| }; |
|  |
| vector<u8> sm3\_padding(const u8\* msg, size\_t len) { |
| u64 bit\_len = len \* 8; |
| size\_t k = (448 - (bit\_len + 1) % 512 + 512) % 512; |
| size\_t total\_len = len + 1 + k / 8 + 8; |
|  |
| vector<u8> padded(total\_len); |
| memcpy(padded.data(), msg, len); |
| padded[len] = 0x80; |
|  |
| for (int i = 0; i < 8; i++) { |
| padded[total\_len - 1 - i] = (bit\_len >> (8 \* i)) & 0xFF; |
| } |
| return padded; |
| } |
|  |
| void sm3\_compress(u32 V[8], const u8\* block) { |
| u32 W[68], W1[64]; |
|  |
| for (int i = 0; i < 16; i++) { |
| W[i] = (block[4 \* i] << 24) | (block[4 \* i + 1] << 16) | |
| (block[4 \* i + 2] << 8) | (block[4 \* i + 3]); |
| } |
|  |
| for (int i = 16; i < 68; i++) { |
| W[i] = P1(W[i - 16] ^ W[i - 9] ^ left\_rotate(W[i - 3], 15)) |
| ^ left\_rotate(W[i - 13], 7) ^ W[i - 6]; |
| } |
|  |
| for (int i = 0; i < 64; i++) { |
| W1[i] = W[i] ^ W[i + 4]; |
| } |
|  |
| u32 A = V[0], B = V[1], C = V[2], D = V[3]; |
| u32 E = V[4], F = V[5], G = V[6], H = V[7]; |
|  |
| for (int j = 0; j < 64; j++) { |
| u32 SS1 = left\_rotate((left\_rotate(A, 12) + E + left\_rotate(Tj(j), j % 32)) & 0xFFFFFFFF, 7); |
| u32 SS2 = SS1 ^ left\_rotate(A, 12); |
| u32 TT1 = (FF(A, B, C, j) + D + SS2 + W1[j]) & 0xFFFFFFFF; |
| u32 TT2 = (GG(E, F, G, j) + H + SS1 + W[j]) & 0xFFFFFFFF; |
|  |
| D = C; |
| C = left\_rotate(B, 9); |
| B = A; |
| A = TT1; |
| H = G; |
| G = left\_rotate(F, 19); |
| F = E; |
| E = P0(TT2); |
| } |
|  |
| V[0] ^= A; V[1] ^= B; V[2] ^= C; V[3] ^= D; |
| V[4] ^= E; V[5] ^= F; V[6] ^= G; V[7] ^= H; |
| } |
|  |
| vector<u8> sm3(const u8\* msg, size\_t len) { |
| vector<u8> padded = sm3\_padding(msg, len); |
| u32 V[8]; |
| memcpy(V, IV, sizeof(IV)); |
|  |
| size\_t block\_count = padded.size() / 64; |
| for (size\_t i = 0; i < block\_count; i++) { |
| sm3\_compress(V, &padded[i \* 64]); |
| } |
|  |
| vector<u8> digest(32); |
| for (int i = 0; i < 8; i++) { |
| digest[4 \* i] = (V[i] >> 24) & 0xFF; |
| digest[4 \* i + 1] = (V[i] >> 16) & 0xFF; |
| digest[4 \* i + 2] = (V[i] >> 8) & 0xFF; |
| digest[4 \* i + 3] = V[i] & 0xFF; |
| } |
| return digest; |
| } |
|  |
| void print\_hex(const vector<u8>& data) { |
| for (u8 b : data) |
| cout << hex << setw(2) << setfill('0') << (int)b; |
| cout << dec << endl; |
| } |
|  |
| int main() { |
| string msg = "abc"; |
|  |
| cout << "SM3(\"abc\") = "; |
| auto hash = sm3((const u8\*)msg.c\_str(), msg.size()); |
| print\_hex(hash); |
|  |
| // 运行效率测试 |
| const int N = 10000; // 测试次数 |
| vector<u8> test\_data(64, 0x61); // 64 字节全是 'a' |
|  |
| auto start = high\_resolution\_clock::now(); |
|  |
| for (int i = 0; i < N; ++i) { |
| sm3(test\_data.data(), test\_data.size()); |
| } |
|  |
| auto end = high\_resolution\_clock::now(); |
| auto duration = duration\_cast<milliseconds>(end - start); |
|  |
| cout << "运行 " << N << " 次 SM3（64字节输入）总耗时: " |
| << duration.count() << " 毫秒" << endl; |
| cout << "平均每次耗时: " << (double)duration.count() / N << " 毫秒" << endl; |
|  |
| return 0; |
| } |

三、优化思路

四路并行设计：使用 \_\_m256i 向量寄存器打包四个 32 位状态或中间字（W）数据，使得一轮运算同时处理 4 条 SM3 消息。

消息扩展并行化：W[0..67] 和 W1[0..63] 由 SIMD 并行扩展生成，节省重复计算负载。

压缩函数并行执行：64 轮压缩中的逻辑函数（FF/GG）、中间变量 SS1, SS2, TT1, TT2 等都通过向量指令并行计算。

IV 和最终状态写出：初始 IV 用 set1\_epi32 批量复制，每条 lane 有自己的 IV；压缩结束后将并行状态存储并分离为每条消息的最终 hash 值。

四、优化后代码

|  |
| --- |
| #include <iostream> |
| #include <vector> |
| #include <cstring> |
| #include <iomanip> |
| #include <chrono> |
| #include <immintrin.h> |
|  |
| using namespace std; |
| using namespace std::chrono; |
|  |
| typedef uint8\_t u8; |
| typedef uint32\_t u32; |
| typedef uint64\_t u64; |
|  |
| static const u32 IV[8] = { |
| 0x7380166F,0x4914B2B9,0x172442D7,0xDA8A0600, |
| 0xA96F30BC,0x163138AA,0xE38DEE4D,0xB0FB0E4E |
| }; |
|  |
| inline u32 Tj\_scalar(int j) { return (j < 16) ? 0x79CC4519 : 0x7A879D8A; } |
|  |
| inline \_\_m256i Tj\_vec(int j) { |
| u32 t = Tj\_scalar(j); |
| return \_mm256\_set1\_epi32((int)t); |
| } |
|  |
| inline \_\_m256i rotl32(\_\_m256i x, int n) { |
| return \_mm256\_or\_si256(\_mm256\_slli\_epi32(x, n), |
| \_mm256\_srli\_epi32(x, 32 - n)); |
| } |
|  |
| inline \_\_m256i P0\_vec(\_\_m256i x) { |
| return \_mm256\_xor\_si256(\_mm256\_xor\_si256(x, rotl32(x, 9)), rotl32(x, 17)); |
| } |
|  |
| inline \_\_m256i P1\_vec(\_\_m256i x) { |
| return \_mm256\_xor\_si256(\_mm256\_xor\_si256(x, rotl32(x, 15)), rotl32(x, 23)); |
| } |
|  |
| inline \_\_m256i FF\_vec(\_\_m256i x, \_\_m256i y, \_\_m256i z, int j) { |
| if (j < 16) return \_mm256\_xor\_si256(\_mm256\_xor\_si256(x, y), z); |
| \_\_m256i t1 = \_mm256\_and\_si256(x, y), t2 = \_mm256\_and\_si256(x, z), t3 = \_mm256\_and\_si256(y, z); |
| return \_mm256\_or\_si256(\_mm256\_or\_si256(t1, t2), t3); |
| } |
|  |
| inline \_\_m256i GG\_vec(\_\_m256i x, \_\_m256i y, \_\_m256i z, int j) { |
| if (j < 16) return \_mm256\_xor\_si256(\_mm256\_xor\_si256(x, y), z); |
| \_\_m256i t1 = \_mm256\_and\_si256(x, y); |
| \_\_m256i t2 = \_mm256\_andnot\_si256(x, z); |
| return \_mm256\_or\_si256(t1, t2); |
| } |
|  |
| void sm3\_4way\_compress(\_\_m256i V[8], const u8\* blocks[4]) { |
| \_\_m256i W[68], W1[64]; |
| // load W0-15 for each lane |
| for (int j = 0; j < 16; j++) { |
| u32 w0 = (blocks[0][4 \* j] << 24) | (blocks[0][4 \* j + 1] << 16) | (blocks[0][4 \* j + 2] << 8) | (blocks[0][4 \* j + 3]); |
| u32 w1 = (blocks[1][4 \* j] << 24) | (blocks[1][4 \* j + 1] << 16) | (blocks[1][4 \* j + 2] << 8) | (blocks[1][4 \* j + 3]); |
| u32 w2 = (blocks[2][4 \* j] << 24) | (blocks[2][4 \* j + 1] << 16) | (blocks[2][4 \* j + 2] << 8) | (blocks[2][4 \* j + 3]); |
| u32 w3 = (blocks[3][4 \* j] << 24) | (blocks[3][4 \* j + 1] << 16) | (blocks[3][4 \* j + 2] << 8) | (blocks[3][4 \* j + 3]); |
| W[j] = \_mm256\_setr\_epi32(w0, w1, w2, w3); |
| } |
| for (int j = 16; j < 68; j++) { |
| \_\_m256i wj16 = W[j - 16], wj9 = W[j - 9], wj3 = rotl32(W[j - 3], 15); |
| \_\_m256i tmp = \_mm256\_xor\_si256(\_mm256\_xor\_si256(wj16, wj9), wj3); |
| \_\_m256i part = \_mm256\_xor\_si256(tmp, rotl32(W[j - 13], 7)); |
| W[j] = \_mm256\_xor\_si256(\_mm256\_xor\_si256(P1\_vec(tmp), rotl32(W[j - 13], 7)), W[j - 6]); |
| } |
| for (int j = 0; j < 64; j++) { |
| W1[j] = \_mm256\_xor\_si256(W[j], W[j + 4]); |
| } |
|  |
| \_\_m256i A = V[0], B = V[1], C = V[2], D = V[3], E = V[4], F = V[5], G = V[6], H = V[7]; |
| for (int j = 0; j < 64; j++) { |
| \_\_m256i SS1 = rotl32(\_mm256\_add\_epi32(\_mm256\_add\_epi32(rotl32(A, 12), E), \_mm256\_add\_epi32(Tj\_vec(j), rotl32(A, 12))), 7); |
| \_\_m256i SS2 = \_mm256\_xor\_si256(SS1, rotl32(A, 12)); |
| \_\_m256i TT1 = \_mm256\_add\_epi32(\_mm256\_add\_epi32(\_mm256\_add\_epi32(FF\_vec(A, B, C, j), D), SS2), W1[j]); |
| \_\_m256i TT2 = \_mm256\_add\_epi32(\_mm256\_add\_epi32(\_mm256\_add\_epi32(GG\_vec(E, F, G, j), H), SS1), W[j]); |
|  |
| D = C; |
| C = rotl32(B, 9); |
| B = A; |
| A = TT1; |
| H = G; |
| G = rotl32(F, 19); |
| F = E; |
| E = P0\_vec(TT2); |
| } |
| V[0] = \_mm256\_xor\_si256(V[0], A); |
| V[1] = \_mm256\_xor\_si256(V[1], B); |
| V[2] = \_mm256\_xor\_si256(V[2], C); |
| V[3] = \_mm256\_xor\_si256(V[3], D); |
| V[4] = \_mm256\_xor\_si256(V[4], E); |
| V[5] = \_mm256\_xor\_si256(V[5], F); |
| V[6] = \_mm256\_xor\_si256(V[6], G); |
| V[7] = \_mm256\_xor\_si256(V[7], H); |
| } |
|  |
| vector<vector<u8>> sm3\_4way(const vector<vector<u8>>& msgs) { |
| size\_t n = msgs.size(); |
| size\_t chunk = n / 4; |
| vector<vector<u8>> out(n, vector<u8>(32)); |
| for (size\_t i = 0; i < chunk; i++) { |
| const u8\* blocks[4] = { msgs[4 \* i + 0].data(), msgs[4 \* i + 1].data(), |
| msgs[4 \* i + 2].data(), msgs[4 \* i + 3].data() }; |
| \_\_m256i V[8]; |
| for (int k = 0; k < 8; k++) V[k] = \_mm256\_set1\_epi32((int)IV[k]); |
| sm3\_4way\_compress(V, blocks); |
| u32 tmp[4][8]; |
| for (int k = 0; k < 8; k++) \_mm256\_storeu\_si256((\_\_m256i\*)tmp[k], V[k]); |
| for (int lane = 0; lane < 4; lane++) { |
| for (int k = 0; k < 8; k++) { |
| u32 v = tmp[lane][k]; |
| out[4 \* i + lane][4 \* k + 0] = (v >> 24) & 0xFF; |
| out[4 \* i + lane][4 \* k + 1] = (v >> 16) & 0xFF; |
| out[4 \* i + lane][4 \* k + 2] = (v >> 8) & 0xFF; |
| out[4 \* i + lane][4 \* k + 3] = v & 0xFF; |
| } |
| } |
| } |
| return out; |
| } |
|  |
|  |
| int main() { |
| vector<vector<u8>> inputs(4, vector<u8>(64, 0)); |
| memcpy(inputs[0].data(), "abc", 3); |
| auto out = sm3\_4way(inputs); |
| for (int i = 0; i < 4; i++) { |
| for (u8 b : out[i]) cout << hex << setw(2) << setfill('0') << (int)b; |
| cout << dec << "\n"; |
| } |
| } |