SM3优化

李瑞涵　202000161037

一.利用SIMD和循环展开进行优化

1.SIMD指令介绍

在本次实验中，由于主机配置为intel CPU，所以SIMD选用了intel架构下的指令集。综合加速效果以及api接口的调用复杂度，最终选用了AVX指令集，下面是对AVX指令集的简单介绍。

（1）数据结构

__m256i 类型： 256bit的整型数据。一个__m256i类型的数据其中可以存储8个int型或者4个long long型，因此，使用SIMD指令可以同时对8个int型或4个long long 类型的数据进行计算，在局部运算中达到并行性。

（2）使用的基本格式

首先把内存中的数据移动到寄存器中，在寄存器中调用运算指令。当运算完成后，把结果从寄存器中重新移动到内存。

（3）接口命名方式

在intel架构下的SIMD指令的接口命名格式为：_<指令集类型>_<运算类型>_<参与运算的数据类型>

（4）常用接口

```
__m256i  _mm256_setr_epi32(int a,int b,int c,int d,int e,int f,int g,int h)
                                                    赋值（内存到寄存器）

void      _mm256_storeu_epi32(int* a, __m256i b)    把b存到a中(寄存器到内存)

__m256i  _mm256_and_si256(__m256i a,__m256i b)      逻辑与

__m256i  _mm256_or_si256(__m256i a,__m256i b)       逻辑或

__m256i  _mm256_xor_si256(__m256i a,__m256i b)      异或

__m256i  _mm256_slli_epi32(__m256i a, int count)    左移count位

__m256i  _mm256_srli_epi32(__m256i a,int count)     右移count位
```

2.使用SIMD指令进行优化

在本次实验中，我使用SIMD主要对两个地方进行了优化：SM3消息扩展部分，SM3迭代压缩部分。

（1）使用SIMD对消息扩展部分进行优化

a. 消息扩展的基本过程

消息扩展将512bit的一个分组分为16个字，记做W_0...W_{15}。将这十六个字扩展为132个字，即W_0...W_{67}与W'_0....W'_{63}。

其中$W_j = P_1(W_{j-16} \oplus W_{j-9} \oplus (W_{j-3} <<< 15)) \oplus (W_{j-13} <<< 7) \oplus W_{j-6}$；$W'_j = W_j \oplus W_{j+4}$

b. 对W_j的生成进行优化

在之前写的SM3算法中，数据类型全部为int型，所以可以将8个int型打包成一个__m256i型。但是W_j的运算不可以全用__m256i类型计算，因为其中有W_{j-3}和W_{j-6}的存在。如果以8个一组，那么W_{j-3}和W_{j-6}的数据缺失，无法进行计算。

因此可以使用SIMD指令先计算$W_{j-16} \oplus W_{j-9}$和$W_{j-13} <<< 7$，将计算结果转为int型。再使用int型完成剩余W_j的计算。

首先先定义SIMD的循环左移:将a循环左移b位。代码如下

```
__m256i _mm256_rl_epi32(__m256i a, int b) {
    return _mm256_xor_si256(_mm256_slli_epi32(a, b), _mm256_srli_epi32(a, 32 -
b))
}
```

然后可以进行$W_{j-16} \oplus W_{j-9}$和$W_{j-13} <<< 7$的运算。注意W一共有68个，而SIMD以8个为一组。因此使用SIMD计算前64个，最后4个使用int类型计算即可。代码如下，其中tmp1是$W_{j-16} \oplus W_{j-9}$，tmp2是$W_{j-13} <<< 7$，即W_j的计算变成$P_1(tmp1 \oplus (W_{j-3} <<< 15)) \oplus tmp2 \oplus W_{j-6}$ 其中tmp1和tmp2使用SIMD完成。

```
for (int i = 16; i < 64; i++) {
    if (i%8 == 0) {
        __m256i a = _mm256_setr_epi32(w[i - 16], w[i - 15], w[i - 14], w[i -
13],
                    w[i - 12], w[i - 11], w[i - 10], w[i - 9]);
        __m256i b = _mm256_setr_epi32(w[i - 9], w[i - 8], w[i - 7], w[i - 6],
                    w[i - 5], w[i - 4], w[i - 3], w[i - 2]);
        __m256i c = _mm256_setr_epi32(w[i - 13], w[i - 12], w[i - 11], w[i -
10],
                    w[i - 9], w[i - 8], w[i - 7], w[i - 6]);

        __m256i d = _mm256_xor_si256(a, b);
        __m256i e = _mm256_rl_epi32(c, 7);

        _mm256_storeu_epi32(tmp1[count], d);
        _mm256_storeu_epi32(tmp2[count], e);
        count++;
    }
    int index = i - i / 8 * 8;
    int count1 = count - 1;
    w[i] = (int)P1(tmp1[count1][index] ^ rol(w[i - 3], 15, 'l')) ^ tmp2[count1]
[index] ^ w[i - 6];
}
```

```
for (int i = 64; i < 68; i++) {
     W[i] = (int)P1(W[i - 16] ^ W[i - 9] ^ rol(W[i - 3], 15, 'l')) ^ rol(W[i -
13], 7, 'l') ^ W[i - 6];
}
```

c. 对W'j的生成进行优化

注意到W'一共有64个，正好是8的倍数，而且使用已经全部计算完成的W，不需要考虑数据依赖问题，因此可以全部使用SIMD指令。大概思路如下：把W_j到W_{j+8}存在一个__m256i数据中，把W_{j+4}到W_{j+12}存在另一个__m256i数据中。而后对这两个数据进行异或操作，最后得到的结果为W'_j到W'_{j+8}。

代码如下

```
__m256i Wi[8], Wi_4[8], W_another_tmp[8];
for (i = 0; i < 8; i++) {
    int a = 8 * i;
    Wi[i] = _mm256_setr_epi32(W[a], W[a + 1], W[a + 2], W[a + 3],
                                      W[a + 4], W[a + 5], W[a + 6], W[a + 7]);

    Wi_4[i] = _mm256_setr_epi32(W[a + 4], W[a + 5], W[a + 6], W[a + 7],
                                        W[a + 8], W[a + 9], W[a + 10], W[a +
11]);
    W_another_tmp[i] = _mm256_xor_si256(Wi[i], Wi_4[i]);
    _mm256_storeu_epi32(W_another + a, W_another_tmp[i]);
}
```

(2) 使用SIMD对迭代压缩部分进行优化

迭代压缩部分过程如下

5.3.3 压缩函数

令A,B,C,D,E,F,G,H为字寄存器,$SS1,SS2,TT1,TT2$为中间变量,压缩函数$V^{i+1} = CF(V^{(i)}, B^{(i)}), 0 \leq i \leq n-1$。计算过程描述如下：

$ABCDEFGH \leftarrow V^{(i)}$

FOR j=0 TO 63

$SS1 \leftarrow ((A \lll 12) + E + (T_j \lll j)) \lll 7$

$SS2 \leftarrow SS1 \oplus (A \lll 12)$

$TT1 \leftarrow FF_j(A,B,C) + D + SS2 + W'_j$

$TT2 \leftarrow GG_j(E,F,G) + H + SS1 + W_j$

$D \leftarrow C$

$C \leftarrow B \lll 9$

$B \leftarrow A$

$A \leftarrow TT1$

$H \leftarrow G$

$G \leftarrow F \lll 19$

$F \leftarrow E$

$E \leftarrow P_0(TT2)$

ENDFOR

$V^{(i+1)} \leftarrow ABCDEFGH \oplus V^{(i)}$

其中，字的存储为大端(big-endian)格式。

其中ABCDEFG的计算比较复杂，依赖度较高，因此只对结尾的$V^{(i+1)}$ = ABCDEFGH ⊕ $V^{(i)}$进行优化。使用SIMD指令，相当于把八次循环(ABCDEFGH和对应的$V^{(i)}$部分)变为一次SIMD指令。代码如下

```
__m256i iv_tmp = _mm256_setr_epi32(IV[0], IV[1], IV[2], IV[3], IV[4], IV[5],
IV[6], IV[7]);
__m256i letter_tmp = _mm256_setr_epi32(A, B, C, D, E, F, G, H);
__m256i result_tmp = _mm256_xor_si256(letter_tmp, iv_tmp);
_mm256_storeu_epi32(result, result_tmp);
```

3.使用循环展开进行优化

（1）对消息扩展中W_j的运算进行循环展开

代码如下

```
    __m256i Wi[8], Wi_4[8], W_another_tmp[8];
        for (i = 0; i < 7; i+=2) {
            int a = 8 * i;
            Wi[i] = _mm256_setr_epi32(w[a], w[a + 1], w[a + 2], w[a + 3],
                                      w[a + 4], w[a + 5], w[a + 6], w[a + 7]);

            Wi_4[i] = _mm256_setr_epi32(w[a + 4], w[a + 5], w[a + 6], w[a + 7],
                                        w[a + 8], w[a + 9], w[a + 10], w[a +
11]);
            W_another_tmp[i] = _mm256_xor_si256(Wi[i], Wi_4[i]);
            _mm256_storeu_epi32(W_another + a, W_another_tmp[i]);

            int b = 8 * i + 8;
            Wi[i + 1] = _mm256_setr_epi32(w[b], w[b + 1], w[b + 2], w[b + 3],
                w[b + 4], w[b + 5], w[b + 6], w[b + 7]);

            Wi_4[i + 1] = _mm256_setr_epi32(w[b + 4], w[b + 5], w[b + 6], w[b +
7],
                w[b + 8], w[b + 9], w[b + 10], w[b + 11]);
            W_another_tmp[i + 1] = _mm256_xor_si256(Wi[i + 1], Wi_4[i + 1]);
            _mm256_storeu_epi32(W_another + b, W_another_tmp[i + 1]);
        }

        for (i; i < 8; i++) {
            int a = 8 * i;
            Wi[i] = _mm256_setr_epi32(w[a], w[a + 1], w[a + 2], w[a + 3],
                w[a + 4], w[a + 5], w[a + 6], w[a + 7]);

            Wi_4[i] = _mm256_setr_epi32(w[a + 4], w[a + 5], w[a + 6], w[a + 7],
                w[a + 8], w[a + 9], w[a + 10], w[a + 11]);
            W_another_tmp[i] = _mm256_xor_si256(Wi[i], Wi_4[i]);
            _mm256_storeu_epi32(W_another + a, W_another_tmp[i]);
        }
```

（2）对消息扩展中W_j的运算进行循环展开

```
int tmp1[8][8], tmp2[8][8];
int count = 0;


for (int i = 16; i < 64; i+=2) {
    if (i%8 == 0) {
        __m256i a = _mm256_setr_epi32(w[i - 16], w[i - 15], w[i - 14], w[i -
13],
                    w[i - 12], w[i - 11], w[i - 10], w[i - 9]);
        __m256i b = _mm256_setr_epi32(w[i - 9], w[i - 8], w[i - 7], w[i - 6],
                    w[i - 5], w[i - 4], w[i - 3], w[i - 2]);
        __m256i c = _mm256_setr_epi32(w[i - 13], w[i - 12], w[i - 11], w[i -
10],
                    w[i - 9], w[i - 8], w[i - 7], w[i - 6]);

        __m256i d = _mm256_xor_si256(a, b);
        __m256i e = _mm256_rl_epi32(c, 7);

        _mm256_storeu_epi32(tmp1[count], d);
        _mm256_storeu_epi32(tmp2[count], e);
        count++;
    }
    int index = i - i / 8 * 8;
    int count1 = count - 1;
    w[i] = (int)P1(tmp1[count1][index] ^ rol(w[i - 3], 15, 'l')) ^ tmp2[count1]
[index] ^ w[i - 6];
    w[i+1] = (int)P1(tmp1[count1][index + 1] ^ rol(w[i - 2], 15, 'l')) ^
tmp2[count1][index + 1] ^ w[i - 5];
}

        //tmp1 = w[i - 16] ^ w[i - 9]
        //tmp2 = rol(w[i - 13], 7, 'l')

for (int i = 64; i < 68; i+=2) {
    w[i] = (int)P1(w[i - 16] ^ w[i - 9] ^ rol(w[i - 3], 15, 'l')) ^ rol(w[i -
13], 7, 'l') ^ w[i - 6];
    w[i + 1] = (int)P1(w[i - 15] ^ w[i - 8] ^ rol(w[i - 2], 15, 'l')) ^ rol(w[i
- 12], 7, 'l') ^ w[i - 5];

}
```

4.结果分析

测试程序代码如下。首先先随机生成1000个56字节的字符串，而后对这1000个字符串做1000次哈希，根据1000次哈希的时间，计算出每秒的吞吐量。

```
void test_thread(SM3_basic* test, string* str, int start, int end) {
    for (int i = start; i < end; i++) {
        test[i].update(str[i]);
```

```
    }
}
void test_() {
    SM3_basic test[1000];
    string str[1000] ;
    for (int i = 0; i < 1000; i++)
        str[i] = makeRandStr(56);
    thread first(test_thread, test,str,0,1000);
    clock_t start, end;
    start = clock();
    first.join();
    end = clock();
    double endtime = (double)(end - start) / CLOCKS_PER_SEC;
    cout << "吞吐量:" <<  1.0/endtime*1000*56/1000000  << "MB/s" << endl;
```

无优化时（吞吐量为0.045MB/s）

```
吞吐量:0.0454177MB/s

C:\C++项目文件夹\SM3_optimize\Debug\SM3_optimize.exe (进程 12964)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

采用SIMD优化时(吞吐量为0.14MB/s)

```
吞吐量:0.143959MB/s

C:\C++项目文件夹\SM3_optimize\Debug\SM3_optimize.exe (进程 35648)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

采用循环展开后，吞吐量还在0.14到0.15MB/s之间，几乎没有什么变化。

由此我们可以看出，在采用了SIMD优化后，吞吐量提升了3倍，这个数据比网上使用SIMD的加速比低了不少，猜测是由于换进换出内存的次数太多，而导致加速比不高。但是由于SM3的特性，又使得哈希过程无法全程采用SIMD指令进行运算，因此并没有什么太好的方案来解决换入换出内存多的问题。而采用循环展开之前和采用循环展开之后，吞吐量的提升没有什么变化，猜测是循环次数太少（100次以内），因此采用循环展开的加速效果并不明显。

二.利用多线程进行优化

1. 优化代码

可以利用多线程将数据分块。例如在测试程序中，可以利用2个线程，一个线程处理500个字符串的哈希，这样在理论上可以得到近2倍的加速。

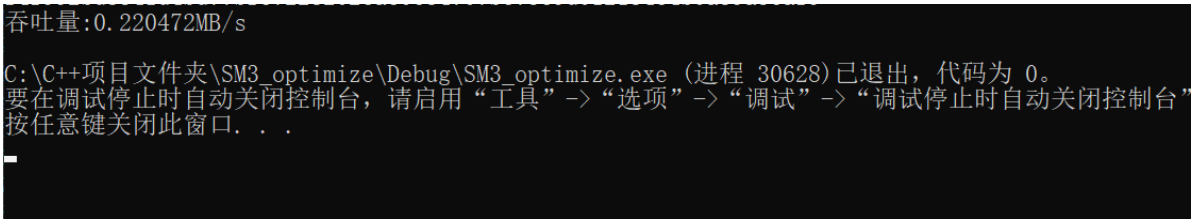
多线程测试程序的代码为

```
void test_thread(SM3_basic* test, string* str, int start, int end) {
    for (int i = start; i < end; i++) {
        test[i].update(str[i]);
    }
}

void test_() {
    SM3_basic test[1000];
```

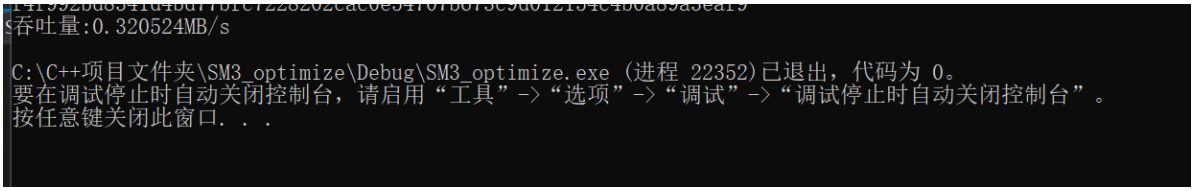
```
    string str[1000] ;
    for (int i = 0; i < 1000; i++)
        str[i] = makeRandStr(56);
    thread first(test_thread, test,str,0,250);
    thread second(test_thread, test, str, 250, 500);
    thread third(test_thread, test, str, 500, 750);
    thread fourth(test_thread, test, str, 750, 1000);
    clock_t start, end;
    start = clock();
    first.join();
    second.join();
    third.join();
    fourth.join();
    end = clock();
    double endtime = (double)(end - start) / CLOCKS_PER_SEC;

    cout << "吞吐量:" <<  1.0/endtime*1000*56/1000000  << "MB/s" << endl;
}
```

2.结果分析

采用双线程的吞吐量(0.22MB/s)。



采用四线程的吞吐量（0.32MB/s).



采用八线程的速度反而会变慢。猜测是由于IDE和电脑的限制，加上线程本身的开销过大，使得数据运算不再是时间的主要决定因素，因此线程太多反而会使吞吐率下降。

可以看到，在没有优化时的0.04MB/s到最终的3.2MB/s，吞吐量最终提升了8倍。最终每秒大概可以执行5万次到6万次SM3算法，这样的效率基本可以作为一个底层算法来实现更高级的其他算法。