《现代密码学》实验报告

实验名称：SHA-256	**实验时间：**2025 年 11 月 16 日
学生姓名：谢润文	**学号：**23336262
学生班级：23 计科	**成绩评定：**

一、实验目的

通过实现 SHA-256 的正向哈希过程，理解散列处理、填充和哈希函数和流式处理的思想，提高分块处理、缓冲区处理和对比调试（对拍）的能力。进一步地，通过指令集加速，更好地理解汇编逻辑到 c++语言的转换。

二、实验内容

用 C++实现 SHA-256 的正向哈希过程。

参数要求：

输入：需要计算 SHA-256 的数据，最大大小约为 128MB。

输出：以二进制格式输出它的 SHA-256。

三、实验原理

SHA-256 主要分为如下几步：信息填充分块（预处理），准备初始常数和轮常数（初始化），生成扩展消息块，以及最后的摘要计算（主循环）。

下面分步介绍。

1. 信息填充分块。对消息进行补码处理：假设消息 M 的二进制编码长度为 l 位（比特）。首先在消息末尾补上一位 1，然后再补上 k 个 0，其中 k 为下列方程的最小正整数：

$$l+k\equiv 447 \pmod{512}$$

需要注意的是若 $l\equiv 447 \pmod{512}$，仍需要填充 512 比特，即 512/8=64 个字节的 0。最后还需要在上述字节串后面继续进行补码，这个时候补的是原消息的二进制长度 l 的 64 位二进制表示形式。

这里以‘114514’的字符串的填充为例。它的 ASCII 码的二进制表示为 31 31 34 35 31 34，然后填上 1 位 1，相当于补一个 80，然后填充 56-6-1=49 个内容为 00 的字节作为填充。消息长度为 48bit，十六进制表示为 30，故最后补码为 00 00 00 00 00 00 00 30。

2. 准备初始常数和轮常数。相比于 SHA-1，在 SHA-256 中，初始哈希值和整数常量更加透明，它们来自素数的平方根和立方根的小数部分。以 h_0 为例，sqrt(2)≈1.41421356237，小数部分为 0.41421356237……，写成十六进制表示为 0.6a09e667f3……，故初始值为 6a09e667f3。

3. 生成扩展消息块。扩展消息块指的是基于原信息的拓展，记为 W[i]，对于 i=0~15

有：W[i]=M[i]，其中 M[i]为原消息以 32 字节（128 比特）为组合的数据块；对于 i=16~63，有： W[i]=sigma1(W[i-2])+W[i-7]+sigma0(W[i-15])+W[i-16] 。 其 中 sigma1(x) 表 示 S(x,17)^S(x,19)^(x>>10)，S 函数是循环右移：S(x,l)=(x>>l)|(x<<(32-l))。sigma0(x)表示 S(x,7)^S(x,18)^(x>>3)。

4. 摘要计算。首先，对于每一个消息块（长度为 512 比特），用 i-1 个哈希值对 a,b,c,d,e,f,g,h 这 8 个数进行初始化，i=1 时用初始常数 h 数组进行初始化。其次，进行更新（应用 SHA256 压缩函数）：for i:0~63：T1=H+Sigma1(E)+Ch(E,F,G)+k[i+1]+W[i]，T2=Sigma0(A)+Maj(A,B,C)，H=G，G=F，F=E，E=D+T1，D=C，C=B，B=A，A=T1+T2。其中 Sigma1(x)表示 S(x,6)^S(x,11)^S(x,25)，区别于 sigma1(x)。Sigma0(x)表示 S(x,2)^S(x,13)^S(x,22)。Ch(x,y,z)定义为(x&y)^(~x&z)。Maj(x,y,z)定义为(x&y)^(x&z)^(y&z)。最后计算第 i 个中间 hash 值，在 2^{32} 次方下做和溢位而不保留的加法。实现时 hash[i]+=a...h 即可。最后的 hash 值就是最终需要的哈希值_hash[]。

至此，SHA-256 的主算法就完成了。在 intel-sha-extensions-white-paper-402097.pdf 中给出了关于 SHA-256 用指令集（SHA-NI）完成的方法。在这里也一并介绍原理。需要事先声明的是，这里 xmm0/1/2 都是 128 位的数。首先是 SHA256RNDS2 指令。在 c++中对应 _mm_sha256rnds2_epu32(xmm1,xmm2,xmm0)的函数。这个指令完成了消息摘要的两轮计算。其次是 SHA256MSG1 指令。该指令对应扩展消息块的生成中 sigma0(W[i-15])+W[i-16]的部分。在 c++中对应_mm_sha256msg1_epu32(xmm1,xmm2)。然后是 SHA256MSG2 指令，该指令相当于完成整个 W[i]的生成（通过 SHA256MSG1 的部分与 sigma1(W[i-2])、W[i-7]相加，在 c++中对应_mm_sha256msg2_epu32(xmm1,xmm2)。

这里以 0~3 轮的消息摘要生成具体为例：

```
movdqu      MSG, [DATA_PTR + 0*16]
pshufb      MSG, SHUF_MASK
movdqa      MSGTMP0, MSG
   paddd       MSG, [SHA256CONSTANTS + 0*16]
   sha256rnds2 STATE1, STATE0
   pshufd      MSG, MSG, 0x0E
   sha256rnds2 STATE0, STATE1
```

图 1 0~3 轮的汇编指令

首先是数据的导入，由于大小端的原因，需要引入 SHUF_MASK 对源数据进行“洗牌”，也就是打乱顺序。具体来说对应到每一个字节的反转，可以用如下代码表示：

```
__m128i shuf_mask=_mm_set_epi32(
    0x0c0d0e0f,
    0x08090a0b,
    0x04050607,
    0x00010203
```

);

其次是取出对应的数据进行摘要计算。这里先是加法，经过两轮运算后再 shuffle，再经过两轮运算，实现 0~3 轮运算的等效。

其他轮数的运算同理，此处不再赘述。

四、实验步骤（源代码）

首先是不用指令集的：

```
#include<iostream>
#include<cstring>
#include<cstdint>
#include<fstream>
#include<fcntl.h>
using namespace std;
/*
FILE *fp=fopen("dump.bin","rb");
FILE *p=fopen("output.bin","wb");
*/

#define fp stdin
#define p stdout

const uint32_t h[9]={0,0x6a09e667,0xbb67ae85,0x3c6ef372,0xa54ff53a,
                     0x510e527f,0x9b05688c,0x1f83d9ab,0x5be0cd19};
const uint32_t k[65]={0,0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5,
                       0x3956c25b,0x59f111f1,0x923f82a4,0xab1c5ed5,
                       0xd807aa98,0x12835b01,0x243185be,0x550c7dc3,
                       0x72be5d74,0x80deb1fe,0x9bdc06a7,0xc19bf174,

                       0xe49b69c1,0xefbe4786,0x0fc19dc6,0x240ca1cc,
                       0x2de92c6f,0x4a7484aa,0x5cb0a9dc,0x76f988da,
                       0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7,
                       0xc6e00bf3,0xd5a79147,0x06ca6351,0x14292967,
                       0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13,
                       0x650a7354,0x766a0abb,0x81c2c92e,0x92722c85,
                       0xa2bfe8a1,0xa81a664b,0xc24b8b70,0xc76c51a3,
```

```
                        0xd192e819,0xd6990624,0xf40e3585,0x106aa070,

                        0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5,
                        0x391c0cb3,0x4ed8aa4a,0x5b9cca4f,0x682e6ff3,
                        0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208,
                        0x90befffa,0xa4506ceb,0xbef9a3f7,0xc67178f2};

    uint32_t
A=h[1],B=h[2],C=h[3],D=h[4],E=h[5],F=h[6],G=h[7],H=h[8],W[65],T1,T2;
    uint32_t _hash[10]={0};
    uint32_t S(uint32_t x,uint8_t l){
        return (x>>l)|(x<<(32-l));
    }
    uint32_t Ch(uint32_t x,uint32_t y,uint32_t z){
        return (x&y)^(~x&z);
    }
    uint32_t Maj(uint32_t x,uint32_t y,uint32_t z){
        return (x&y)^(x&z)^(y&z);
    }
    uint32_t Sigma0(uint32_t x){
        return S(x,2)^S(x,13)^S(x,22);
    }
    uint32_t Sigma1(uint32_t x){
        return S(x,6)^S(x,11)^S(x,25);
    }
    uint32_t sigma0(uint32_t x){
        return S(x,7)^S(x,18)^(x>>3);
    }
    uint32_t sigma1(uint32_t x){
        return S(x,17)^S(x,19)^(x>>10);
    }
    void sha_256(uint8_t tmp[],uint32_t _hash[],int start){
        if(start){
            A=_hash[0];
```

```
            B=_hash[1];
            C=_hash[2];
            D=_hash[3];
            E=_hash[4];
            F=_hash[5];
            G=_hash[6];
            H=_hash[7];
        }
        for(int i=0;i<=15;++i){

    W[i]=((uint32_t)tmp[i*4]<<24)|((uint32_t)tmp[i*4+1]<<16)|((uint32_t)
tmp[i*4+2]<<8)|((uint32_t)tmp[i*4+3]);
        }
        for(int i=16;i<=63;++i){
            W[i]=sigma1(W[i-2])+W[i-7]+sigma0(W[i-15])+W[i-16];
        }
        //cout<<hex<<(int)A<<'  '<<(int)B<<'  '<<(int)C<<'  '<<(int)D<<'
'<<(int)E<<' '<<(int)F<<' '<<(int)G<<' '<<(int)H<<endl;
        for(int i=0;i<=63;++i){
            T1=H+Sigma1(E)+Ch(E,F,G)+k[i+1]+W[i];
            /*
            cout<<"Ch:"<<hex<<(int)Ch(E,F,G)<<endl;
            cout<<"Sigma1:"<<hex<<(int)Sigma1(E)<<endl;
            cout<<"W[i]:"<<hex<<W[i]<<endl;
            cout<<"T1:"<<hex<<(int)T1<<endl;
            */
            T2=Sigma0(A)+Maj(A,B,C);
            H=G;
            G=F;
            F=E;
            E=D+T1;
            D=C;
            C=B;
            B=A;
```

```
            A=T1+T2;
            cout<<hex<<(int)A<<' '<<(int)B<<' '<<(int)C<<' '<<(int)D<<'
'<<(int)E<<' '<<(int)F<<' '<<(int)G<<' '<<(int)H<<endl;
        }
        _hash[0]+=A;
        _hash[1]+=B;
        _hash[2]+=C;
        _hash[3]+=D;
        _hash[4]+=E;
        _hash[5]+=F;
        _hash[6]+=G;
        _hash[7]+=H;
    }

    int main(){
        #ifdef _WIN32
        setmode(fileno(stdin), O_BINARY);
        setmode(fileno(stdout), O_BINARY);
        #endif
        uint8_t c,tmp[70],k;
        uint64_t cnt=0;
        int start=0;
        for(int i=0;i<8;++i) _hash[i]=h[i+1];
        while(!feof(fp)){
            fread(&c,sizeof(uint8_t),1,fp);
            if(feof(fp)){
                tmp[cnt%64]=0x80;
                if(cnt%64<=55){
                    k=55-cnt%64;
                    for(int i=1;i<=k;++i) tmp[cnt%64+i]=0x00;//padding
                    cnt<<=3;
                    for(int i=0;i<8;++i){
                        tmp[63-i]=cnt&0xFF;
                        cnt>>=8;
```

```
                }
                sha_256(tmp,_hash,start);
            }
            else{
                for(int i=cnt%64+1;i<=63;++i) tmp[i]=0x00;
                sha_256(tmp,_hash,start);
                //for(int i=0;i<=7;++i) cout<<hex<<(int)_hash[i]<<' ';
                //cout<<endl;
                cnt<<=3;
                for(int i=0;i<8;++i){
                    tmp[63-i]=cnt&0xFF;
                    cnt>>=8;
                }
                for(int i=0;i<56;++i) tmp[i]=0x00;
                sha_256(tmp,_hash,1);
            }
            for(int i=0;i<=7;++i){
                //cout<<hex<<(int)_hash[i]<<' ';
                uint32_t ttmp=_hash[i];
                uint8_t tttmp[4];
                for(int j=0;j<=3;++j){
                    tttmp[3-j]=ttmp;
                    ttmp>>=8;
                }
                fwrite(&tttmp,sizeof(uint8_t),4,p);
            }
            break;
        }
        tmp[cnt%64]=c;
        //cout<<cnt<<endl;
        if(cnt%64==63){
            sha_256(tmp,_hash,start);
            /*
            for(int i=0;i<=7;++i){
```

```
                cout<<hex<<(int)_hash[i]<<' ';
            }
            */
        }
        cnt++;
        if(cnt>64) start=1;
    }
}
```

其次是使用指令集的：

```
#include<iostream>
#include<cstring>
#include<cstdint>
#include<fstream>
#include<fcntl.h>
#include<wmmintrin.h>
#include<immintrin.h>
#include<emmintrin.h>
using namespace std;
/*
FILE *fp=fopen("dump.bin","rb");
FILE *p=fopen("output.bin","wb");
*/

#define fp stdin
#define p stdout

const uint32_t h[9]={0,0x6a09e667,0xbb67ae85,0x3c6ef372,0xa54ff53a,
                     0x510e527f,0x9b05688c,0x1f83d9ab,0x5be0cd19};
uint32_t _hash[10]={0};
__m128i
state0,state1,msg,msgtmp0,msgtmp1,msgtmp2,msgtmp3,msgtmp4,abef,cdgh;
__m128i shuf_mask=_mm_set_epi32(
    0x0c0d0e0f,
    0x08090a0b,
```

```
        0x04050607,
        0x00010203
    );
    void sha_256(uint8_t tmp[],uint32_t _hash[],int start){
        if(!start){
            state0=_mm_set_epi64x(0x6a09e667bb67ae85,0x510e527f9b05688c);
            state1=_mm_set_epi64x(0x3c6ef372a54ff53a,0x1f83d9ab5be0cd19);
        }
        abef=state0;
        cdgh=state1;
        //round 0~3
        msg=_mm_loadu_si128((__m128i*)(tmp+0));
        msgtmp0=_mm_shuffle_epi8(msg,shuf_mask);
        msg=_mm_add_epi32(msgtmp0,_mm_set_epi64x(0xe9b5dba5b5c0fbcf,0x71
374491428a2f98));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        //cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
        //round 4~7
        msg=_mm_loadu_si128((__m128i*)(tmp+16));
        msgtmp1=_mm_shuffle_epi8(msg,shuf_mask);
        msg=_mm_add_epi32(msgtmp1,_mm_set_epi64x(0xab1c5ed5923f82a4,0x59
f111f13956c25b));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp0=_mm_sha256msg1_epu32(msgtmp0,msgtmp1);
        //cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
        //round 8~11
        msg=_mm_loadu_si128((__m128i*)(tmp+32));
        msgtmp2=_mm_shuffle_epi8(msg,shuf_mask);
```

```
	msg=_mm_add_epi32(msgtmp2,_mm_set_epi64x(0x550c7dc3243185be,0x12
835b01d807aa98));
	state1=_mm_sha256rnds2_epu32(state1,state0,msg);
	msg=_mm_shuffle_epi32(msg,0x0e);
	state0=_mm_sha256rnds2_epu32(state0,state1,msg);
	msgtmp1=_mm_sha256msg1_epu32(msgtmp1,msgtmp2);
	//cout<<hex<<state0[0]<<"	"<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
	//round 12~15
	msg=_mm_loadu_si128((__m128i*)(tmp+48));
	msgtmp3=_mm_shuffle_epi8(msg,shuf_mask);
	msg=_mm_add_epi32(msgtmp3,_mm_set_epi64x(0xc19bf1749bdc06a7,0x80
deb1fe72be5d74));
	state1=_mm_sha256rnds2_epu32(state1,state0,msg);
	msgtmp4=_mm_alignr_epi8(msgtmp3,msgtmp2,4);
	msgtmp0=_mm_add_epi32(msgtmp0,msgtmp4);
	msgtmp0=_mm_sha256msg2_epu32(msgtmp0,msgtmp3);
	msg=_mm_shuffle_epi32(msg,0x0e);
	state0=_mm_sha256rnds2_epu32(state0,state1,msg);
	msgtmp2=_mm_sha256msg1_epu32(msgtmp2,msgtmp3);
	//cout<<hex<<state0[0]<<"	"<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
	//round 16~19

msg=_mm_add_epi32(msgtmp0,_mm_set_epi64x(0x240ca1cc0fc19dc6,0xefbe4786e
49b69c1));
	state1=_mm_sha256rnds2_epu32(state1,state0,msg);
	msgtmp4=_mm_alignr_epi8(msgtmp0,msgtmp3,4);
	msgtmp1=_mm_add_epi32(msgtmp1,msgtmp4);
	msgtmp1=_mm_sha256msg2_epu32(msgtmp1,msgtmp0);
	msg=_mm_shuffle_epi32(msg,0x0e);
	state0=_mm_sha256rnds2_epu32(state0,state1,msg);
	msgtmp3=_mm_sha256msg1_epu32(msgtmp3,msgtmp0);
	//cout<<hex<<state0[0]<<"	"<<state0[1]<<endl<<state1[0]<<"
```

```
"<<state1[1]<<endl;
	//round 20~23
	msg=_mm_add_epi32(msgtmp1,_mm_set_epi64x(0x76f988da5cb0a9dc,0x4a
7484aa2de92c6f));
	state1=_mm_sha256rnds2_epu32(state1,state0,msg);
	msgtmp4=_mm_alignr_epi8(msgtmp1,msgtmp0,4);
	msgtmp2=_mm_add_epi32(msgtmp2,msgtmp4);
	msgtmp2=_mm_sha256msg2_epu32(msgtmp2,msgtmp1);
	msg=_mm_shuffle_epi32(msg,0x0e);
	state0=_mm_sha256rnds2_epu32(state0,state1,msg);
	msgtmp0=_mm_sha256msg1_epu32(msgtmp0,msgtmp1);
	//cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
	//round 24~27
	msg=_mm_add_epi32(msgtmp2,_mm_set_epi64x(0xbf597fc7b00327c8,0xa8
31c66d983e5152));
	state1=_mm_sha256rnds2_epu32(state1,state0,msg);
	msgtmp4=_mm_alignr_epi8(msgtmp2,msgtmp1,4);
	msgtmp3=_mm_add_epi32(msgtmp3,msgtmp4);
	msgtmp3=_mm_sha256msg2_epu32(msgtmp3,msgtmp2);
	msg=_mm_shuffle_epi32(msg,0x0e);
	state0=_mm_sha256rnds2_epu32(state0,state1,msg);
	msgtmp1=_mm_sha256msg1_epu32(msgtmp1,msgtmp2);
	//cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
	//round 28~31
	msg=_mm_add_epi32(msgtmp3,_mm_set_epi64x(0x1429296706ca6351,0xd5
a79147c6e00bf3));
	state1=_mm_sha256rnds2_epu32(state1,state0,msg);
	msgtmp4=_mm_alignr_epi8(msgtmp3,msgtmp2,4);
	msgtmp0=_mm_add_epi32(msgtmp0,msgtmp4);
	msgtmp0=_mm_sha256msg2_epu32(msgtmp0,msgtmp3);
	msg=_mm_shuffle_epi32(msg,0x0e);
	state0=_mm_sha256rnds2_epu32(state0,state1,msg);
```

```
	msgtmp2=_mm_sha256msg1_epu32(msgtmp2,msgtmp3);
	//cout<<hex<<state0[0]<<"	"<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
	//round 32~35
	msg=_mm_add_epi32(msgtmp0,_mm_set_epi64x(0x53380d134d2c6dfc,0x2e
1b213827b70a85));
	state1=_mm_sha256rnds2_epu32(state1,state0,msg);
	msgtmp4=_mm_alignr_epi8(msgtmp0,msgtmp3,4);
	msgtmp1=_mm_add_epi32(msgtmp1,msgtmp4);
	msgtmp1=_mm_sha256msg2_epu32(msgtmp1,msgtmp0);
	msg=_mm_shuffle_epi32(msg,0x0e);
	state0=_mm_sha256rnds2_epu32(state0,state1,msg);
	msgtmp3=_mm_sha256msg1_epu32(msgtmp3,msgtmp0);
	//cout<<hex<<state0[0]<<"	"<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
	//round 36~39
	msg=_mm_add_epi32(msgtmp1,_mm_set_epi64x(0x92722c8581c2c92e,0x76
6a0abb650a7354));
	state1=_mm_sha256rnds2_epu32(state1,state0,msg);
	msgtmp4=_mm_alignr_epi8(msgtmp1,msgtmp0,4);
	msgtmp2=_mm_add_epi32(msgtmp2,msgtmp4);
	msgtmp2=_mm_sha256msg2_epu32(msgtmp2,msgtmp1);
	msg=_mm_shuffle_epi32(msg,0x0e);
	state0=_mm_sha256rnds2_epu32(state0,state1,msg);
	msgtmp0=_mm_sha256msg1_epu32(msgtmp0,msgtmp1);
	//cout<<hex<<state0[0]<<"	"<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
	//round 40~43
	msg=_mm_add_epi32(msgtmp2,_mm_set_epi64x(0xc76c51a3c24b8b70,0xa8
1a664ba2bfe8a1));
	state1=_mm_sha256rnds2_epu32(state1,state0,msg);
	msgtmp4=_mm_alignr_epi8(msgtmp2,msgtmp1,4);
	msgtmp3=_mm_add_epi32(msgtmp3,msgtmp4);
	msgtmp3=_mm_sha256msg2_epu32(msgtmp3,msgtmp2);
```

```
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp1=_mm_sha256msg1_epu32(msgtmp1,msgtmp2);
        //cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
        //round 44~47
        msg=_mm_add_epi32(msgtmp3,_mm_set_epi64x(0x106aa070f40e3585,0xd6
990624d192e819));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp3,msgtmp2,4);
        msgtmp0=_mm_add_epi32(msgtmp0,msgtmp4);
        msgtmp0=_mm_sha256msg2_epu32(msgtmp0,msgtmp3);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp2=_mm_sha256msg1_epu32(msgtmp2,msgtmp3);
        //cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
        //round 48~51

msg=_mm_add_epi32(msgtmp0,_mm_set_epi64x(0x34b0bcb52748774c,0x1e376c081
9a4c116));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp0,msgtmp3,4);
        msgtmp1=_mm_add_epi32(msgtmp1,msgtmp4);
        msgtmp1=_mm_sha256msg2_epu32(msgtmp1,msgtmp0);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp3=_mm_sha256msg1_epu32(msgtmp3,msgtmp0);
        //cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
        //round 52~55
        msg=_mm_add_epi32(msgtmp1,_mm_set_epi64x(0x682e6ff35b9cca4f,0x4e
d8aa4a391c0cb3));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
```

```
        msgtmp4=_mm_alignr_epi8(msgtmp1,msgtmp0,4);
        msgtmp2=_mm_add_epi32(msgtmp2,msgtmp4);
        msgtmp2=_mm_sha256msg2_epu32(msgtmp2,msgtmp1);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        //cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
        //round 56~59
        msg=_mm_add_epi32(msgtmp2,_mm_set_epi64x(0x8cc7020884c87814,0x78
a5636f748f82ee));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp2,msgtmp1,4);
        msgtmp3=_mm_add_epi32(msgtmp3,msgtmp4);
        msgtmp3=_mm_sha256msg2_epu32(msgtmp3,msgtmp2);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        //cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
        //round 60~63
        msg=_mm_add_epi32(msgtmp3,_mm_set_epi64x(0xc67178f2bef9a3f7,0xa4
506ceb90befffa));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        //cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
        state0=_mm_add_epi32(state0,abef);
        state1=_mm_add_epi32(state1,cdgh);
        //cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
        _hash[0]=(uint32_t)(state0[1]>>32);
        _hash[1]=(uint32_t)(state0[1]);
        _hash[4]=(uint32_t)(state0[0]>>32);
        _hash[5]=(uint32_t)(state0[0]);
```

```
    _hash[2]=(uint32_t)(state1[1]>>32);
    _hash[3]=(uint32_t)(state1[1]);
    _hash[6]=(uint32_t)(state1[0]>>32);
    _hash[7]=(uint32_t)(state1[0]);
    //for(int i=0;i<=7;++i) cout<<hex<<(int)_hash[i]<<" ";
    //cout<<endl;
}
int main(){
    #ifdef _WIN32
    setmode(fileno(stdin), O_BINARY);
    setmode(fileno(stdout), O_BINARY);
    #endif
    uint8_t tmp[70];
    uint64_t cnt=0;
    int start=0;
    for(int i=0;i<8;++i) _hash[i]=h[i+1];
    while(!feof(fp)){
        int b_cnt=fread(tmp,sizeof(uint8_t),64,fp);
        if(b_cnt==64){
            sha_256(tmp,_hash,start);
            cnt+=64;
            start=1;
        }
        else if(b_cnt){
            cnt+=b_cnt;
            uint64_t len=cnt<<3;
            if(b_cnt<=55){
                tmp[b_cnt]=0x80;
                for(int i=b_cnt+1;i<56;++i) tmp[i]=0x00;
                for(int i=0;i<8;++i) tmp[63-i]=(len>>(i*8))&0xFF;
                sha_256(tmp,_hash,start);
            }
            else{
                tmp[b_cnt]=0x80;
```

```
                for(int i=b_cnt+1;i<64;++i) tmp[i]=0x00;
                sha_256(tmp,_hash,start);
                for(int i=0;i<56;++i) tmp[i]=0x00;
                for(int i=0;i<8;++i) tmp[63-i]=(len>>(i*8))&0xFF;
                sha_256(tmp,_hash,1);
            }
            for(int i=0;i<=7;++i){
                uint32_t ttmp=_hash[i];
                uint8_t tttmp[4];
                for(int j=0;j<=3;++j){
                    tttmp[3-j]=ttmp;
                    ttmp>>=8;
                }
                fwrite(tttmp,sizeof(uint8_t),4,p);
            }
            break;
        }
        else if(feof(fp)){
            if(cnt==0){
                for(int i=0;i<64;++i) tmp[i]=(i==0)?0x80:0x00;
                sha_256(tmp,_hash,start);
                for(int i=0;i<=7;++i){
                    uint32_t ttmp=_hash[i];
                    uint8_t tttmp[4];
                    for(int j=0;j<=3;++j){
                        tttmp[3-j]=ttmp;
                        ttmp>>=8;
                    }
                    fwrite(tttmp,sizeof(uint8_t),4,p);
                }
            }
            break;
        }
    }
```

```
}
```

五、实验结果

先以不用指令集的 SHA-256 为例，测试样例为字符串“abc”。输入数据保存在了 dump2.bin 中（图 2）。

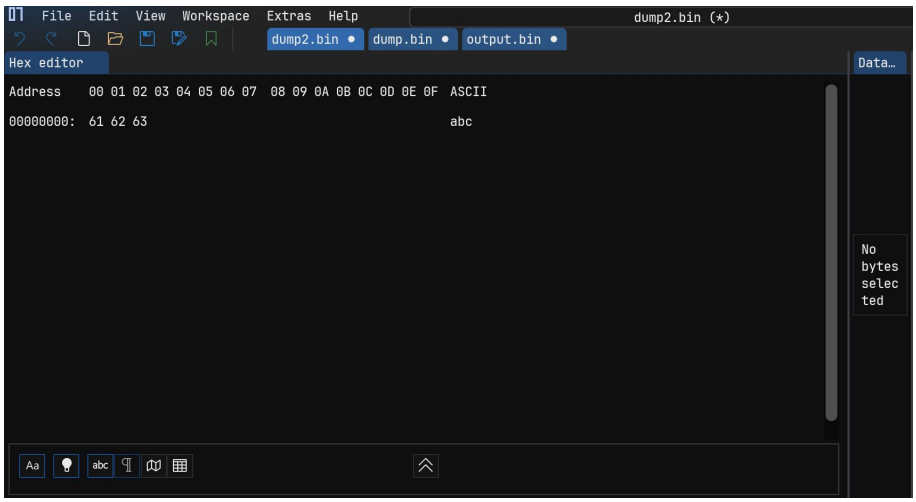

图 2 测试数据 1

运行，得到：

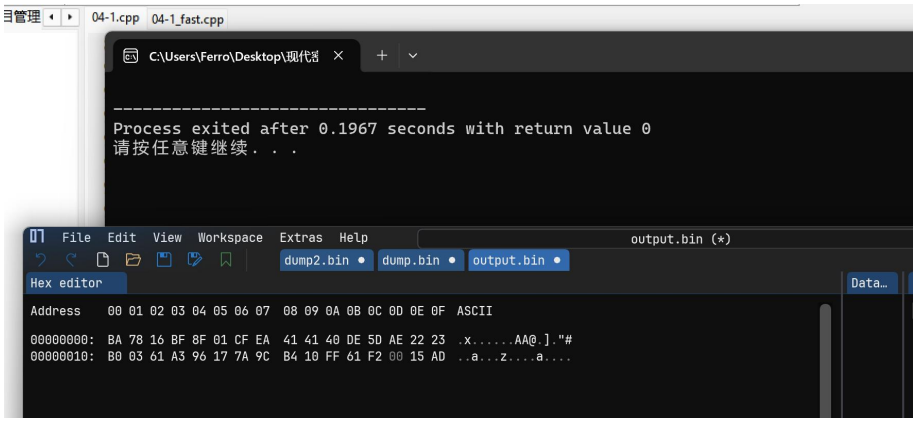

图 3 数据 1 的 hash 结果

在终端执行指令：certutil -hashfile dump2.bin sha256，得到结果如下：

```
C:\Users\Ferro\Desktop\现代密码学>certutil -hashfile dump2.bin sha256
SHA256 的 dump2.bin 哈希：
ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
CertUtil: -hashfile 命令成功完成。
```

图 4 根据系统自带的命令检验 hash 值

可以看到二者无差别。这一结果验证了 SHA-256 实现的正确性。

在 OJ 上评测结果如下（提交编号 1754）：

图 5 不用指令集的评测结果

再以用指令集的程序为例，这一次数据在 dump.bin 中：

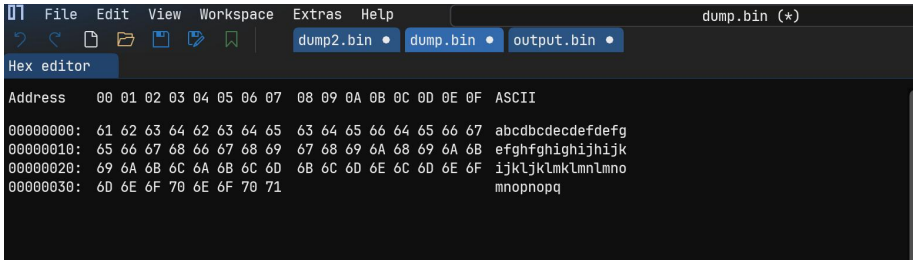

图 6 超出 55 字节需要分成两块的测试数据

运行程序，得到结果如下：

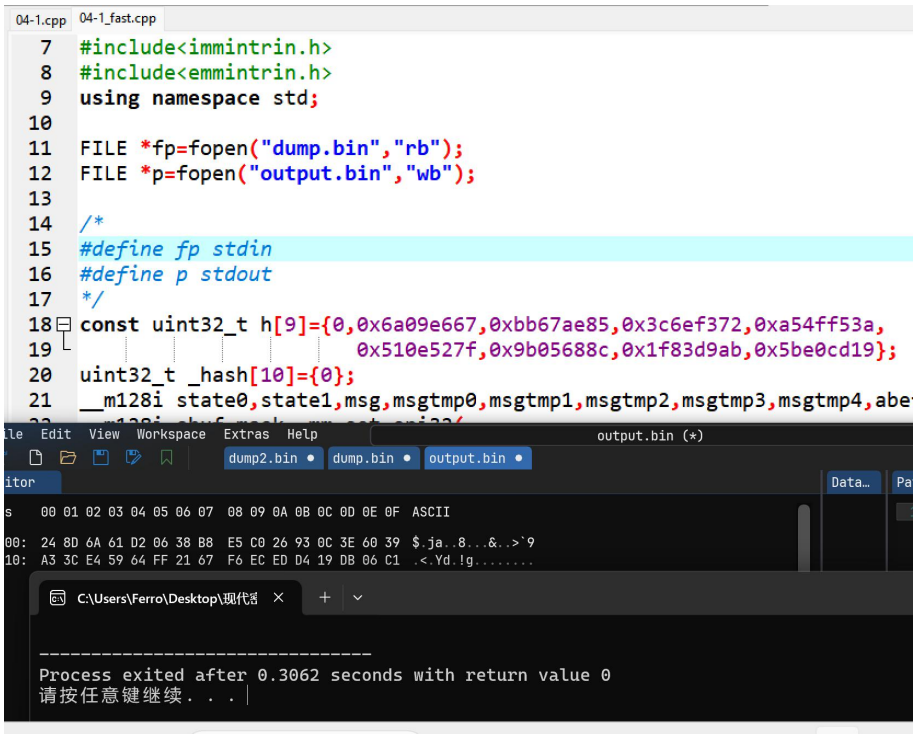

图 7 测试数据 2 的 SHA-256 结果

再在命令提示符中用系统自带的 SHA-256 验证：

```
C:\Users\Ferro\Desktop\现代密码学>certutil -hashfile dump.bin sha256
SHA256 的 dump.bin 哈希：
248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1
CertUtil: -hashfile 命令成功完成。
```

图 8 在 cmd 的 hash 结果

可以验证实现的正确性。

OJ 上评测结果如下（提交编号 1792）：

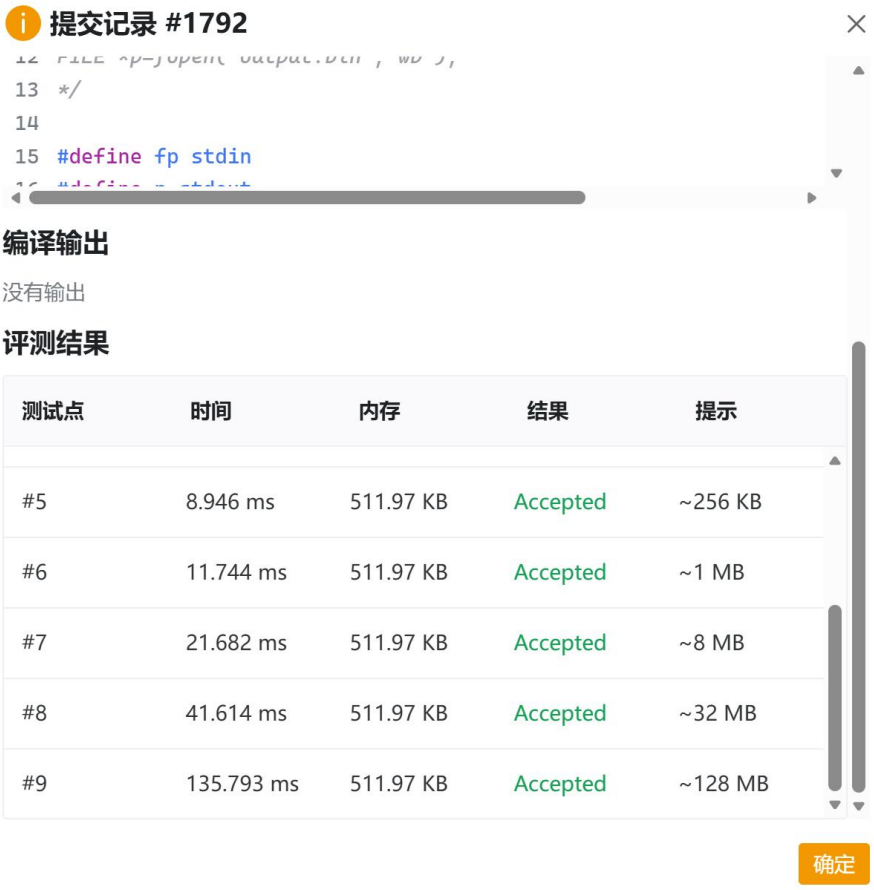

图 9 使用指令集的评测结果

可以看到明显的加速效果。

六、思考题

1. SHA-1 的初始常数是很有规律的 0x67452301,0xEFCDAB89,0x98BADCFE,0x10325476, 0xC3D2E1F0，SHA-256 的初始常数和轮常数分别来自于前几个素数的平方根/立方根的小数部分。为什么要这么设计？

这个问题本质上是说对于初始常数的构造没有隐藏任何嫌疑，也即没有后门。参见 https://en.wikipedia.org/wiki/Nothing-up-my-sleeve_number。假如密码使用者无法辨别是否留有后门，就只能糊涂地认为它们有所谓的“语义安全性”，这显然是把人当羔羊的行为。在 NSA 开发的 SHA-1 算法中，所有初始值均由 NSA 选择。所以人们想，万一 NSA 在这些常量中包含某种后门，以便能够通过查看哈希来获取有关消息的一些信息呢。为了避免这种情况，SHA-256 使用了更“透明”的设计方案，即选取 16 进制下表示的前几个素数的

平方根/立方根的小数部分。

该方案基于的数学思想如下：一般认为实数（例如 π、e 和无理根）在其进位制中的各个位上的数字被认为以相等的频率出现（即它们是正规数）。严格地说，目前还不知道 sqrt（n），n∈N+且 n≠k²，k∈N+的正规性，唯一查到的关于根号 2 的正规性论文（https://arxiv.org/abs/1807.07338，A note on normality of $\sqrt{2}$ in base 2）的方法似乎不能外推。那么我们其实不能完备地说它们各个数位在进制表示下一定是“均匀出现”的，只能“假设”各个数位上出现的数字以相等的频率出现，即这些数没有更好（更有效，柯氏复杂度更低）的表示方法。如果这一假设被数学验证的话，自然能排除后门的嫌疑。

2. 有一类 hash 算法被称为“Non-cryptographic hash function”，例如 Java (OpenJDK) 中 hashCode 的实现是非常简单的迭代计算 h=31*h+x，其他的此类算法包括 FNV、MurmurHash、xxHash 等。它们与 MD5、SHA-1、SHA-256 等 hash 算法的区别是什么？它们有哪些主要用途？为什么会比 SHA-256 等等更适合用于这些场景？

2.1 普通 hash 与密码学 hash 的区别

普通的 hash 是非加密哈希函数，目标是快速地将任意长度的数据映射成定长、分布均匀的摘要，这一生成过程不考虑抗碰撞；而密码学 hash（包括 MD5、SHA 系列）要求在对抗性环境（存在敌手攻击）下，生成一个独一无二的、不可伪造的“数字指纹”，它必须能抵御可能的密码学攻击。

2.2 普通 hash 的用途

大体应用就是数据去重、快速查找、校验和这些方面。

最经典的例子可能是 c++里的 map 了。普通 hash 用来计算键的存储位置。

然后 CRC32（循环冗余校验码）也用到了普通 hash。它用来检测在数据传输和存储中可

能发生的错误。

2.3 为何更适用

首先是性能的问题。哈希表的性能关键在于快速计算哈希值和均匀分布，以减少“哈希冲突”。普通哈希相较于密码学 hash 能更快地计算，不用复杂的填充、轮计算，往往就是一个简单的乘法、加法和模运算，运行速度非常快。

其次是需求不同，针对目标不同的问题。普通的 hash 函数不需要如此高的安全性。非加密哈希函数在“善意”输入下，其分布均匀性和碰撞率已经足够好，完全可以满足哈希表等数据结构的需求。

还有，普通哈希通常输出 32 位或 64 整数，对应寄存器、内存寻址，哈希表可以直接用这个整数取模来确定位置。而 SHA-256 的输出为 32 字节，对于哈希表来说太长了，内存开销较大。

3. 如果你需要设计一个用户系统，你也许知道数据库中不应该以明文存储用户的密码（遗憾的是，我国的知名计算机技术社区 CSDN 就犯过这样的错误），而应该存储密码的 hash。但是，在这种情况下直接使用 SHA-256 等等仍然是不推荐的。有哪些 hash 算法更适合用来处理密码？它们与 SHA-256 等等有什么区别？

3.1 hash 算法举例

比较著名的有 PBKDF2，bcrypt，scrypt 等，比较新的算法有 Argon2 等。这些算法都有一个共同的特点是抗 GPU 口令破解。

3.2 与 SHA-256 的区别

首先是盐值（salt）的引入，相当于与每个用户给出的密码唯一对应的随机值，即使两个用户的密码相同（比如都是 1919810），他们的 hash 值也会因为 salt 值不同而有差异。这样就预防了所谓的“彩虹表攻击”（https://en.wikipedia.org/wiki/Rainbow_table）

即通过预先计算的散列破解，以空间换时间。

其次是更大的内存和时间消耗。比如 PBKDF2 简单而言就是将 salted hash 进行多次重复计算，这个次数是可选择的，导致建立彩虹表所需代价很大。scrypt 和 bcrypt 都是为密码存储设计的算法，有一个参数 work factor 可用于调整计算强度，而且 work factor 是包括在输出的摘要中的。随着攻击者计算能力的提高，使用者可以逐步增大 work factor，而且不会影响已有用户的登陆。在 Argon2d（https://en.wikipedia.org/wiki/Argon2）中以密码依赖的顺序访问内存数组，这减少了时间-内存权衡（TMTO）攻击的可能性。

4. 在比特币中，有一类节点的功能是将交易数据打包成区块附加到区块链上，而我们常说的“挖矿”，其基本原理就是不断变换区块中的 nonce 字段，使整个区块的 SHA-256（当成一个大整数）小于比特币网络当前规定的难度值。glminer 是一个由 EtherDream 编写的玩具版“挖矿”演示：给定 12 bytes 的数据 Q，要求找到另一串 4 bytes 的数据 A，使 Q+A 的 SHA-256 以 00000000 开头。尝试用你实现的 SHA-256 或各种标准库/第三方库来实现这一功能，并和 glminer 比较计算 SHA-256 的性能。查看在 V2EX 上的讨论以及 glminer 的源代码，为什么会存在这样的性能差距？它是如何实现 SHA-256 计算的？

首先找到一个 glminer 能很快挖出来的数据：（Q：4e c3 e6 f3 89 1c dc 67 c6 aa 34 f9）。

图 10 使用 glminer 示例

利用自己编写的代码得到结果如下：

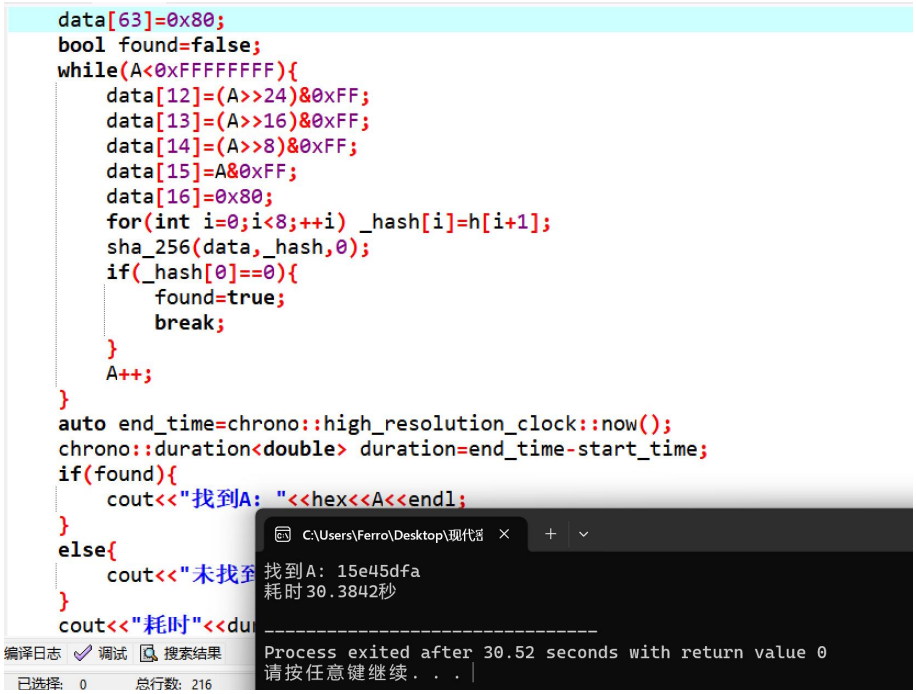

图 11 利用自己写的 sha_256 指令集版得到的结果

可以看到速度大约是 glminer 的 1/9。性能差距的来源是 glminer 利用了 GPU 的着色器进行加速。具体地说，它是这样实现 SHA-256 的：每个片段对应一个线程索引，由 gl_FragCoord.x 得到。CPU 通过 gl.uniformlui 函数把当前总体偏移传给 GPU 的着色器，表示这批 nonce 的起始值。然后就可以用多线程并行加速了。另外，着色器在每个片段内循环做多个 SHA-256 测试，以提高吞吐。

5. 其他的问题？

还是老生常谈，运行速度的问题。一开始我用的是一个一个字节地读取，速度奇慢无比，后面助教说可以多字节（比如 64 字节）读取，因为 fread 可以返回读取的数量。那么就有一个问题，为什么 fread 多字节读取要比单字节读取要快呢？假如我读更多的字节，是不是会更快？

事实上，fread 基于流读入，假如能利用局部缓存一次性多读一些，那么就可以降低

fread 的次数，从而降低系统调用的开销。但是太大的空间可能会导致 MLE，所以理论上应该控制开的数组的大小（缓冲区大小），可以实现有效加速。

七、实验总结

本实验通过完成 SHA-256 的普通实现和指令集实现，加深了对密码学哈希函数、预处理与填充及计算摘要的理解。代码运用了流式处理，以每 64 字节读入的方式分段处理，避免了不必要的空间占用。对于 SHA-NI 的指令集加速，实验结果表征约有 10 倍的加速，其主要通过降低重复运算、利用中间数据实现加速。此外，直接用 2 个 128 位整数进行处理而非 8 个 32 位整数也提高了运算效率。在实现过程中需要将汇编语言转化为 c++，在各个寄存器不同轮数的变化和轮换上容易出错，因此实现时利用了每 4 轮调试输出中间变量进行对比，从而完成了正确的指令集实现。