MD5 and SHA1 Bruteforcing using CUDA

Oliver Gräb

Johannes Gutenberg University Mainz

ograeb@students.uni-mainz.de

April 15, 2016

Overview

1. About Hashing
2. Bruteforcing
3. Implementation
4. Performance
5. Demonstration
6. Sources

General

Hash Function

A hash function is any function that can be used to map data of arbitrary size to data of fixed size.

Cryptographic Hash Function

A cryptographic hash function is a hash function which is considered practically impossible to invert

From `https://en.wikipedia.org/wiki/Hash_function`

MD5 message-digest algorithm

- 128 bit Hash value
- RFC 1321 from 1992
- “cryptographically broken and unsuitable for further use” according to the CMU Software Engineering Institute
- Broken in 2^{18} time, less than a second

SHA-1

- 160 bit Hash value
- Designed by NIST in 1995
- “No longer secure against well-funded opponents”
- Collision complexity $2^{60.3} - 2^{65.3}$
- Replaced by SHA-2 and SHA-3

Why Bruteforce?

Function not reversible
$\Rightarrow$ Try all the inputs

MD5:
$2^{128} = 340,282,366,920,938,463,463,374,607,431,768,211,456$ guesses

SHA1: $2^{160} =$
$1,461,501,637,330,902,918,203,684,832,716,283,019,655,932,542,9$
guesses

That is why we need massive parallelism

Enumeration

How do we try everything?

a

b

⋮

Z

aa

ba

⋮

Limit Inputs to $\{a - z, A - Z, 0 - 9\}$

Basic Implementation

1. Get serial Implementation without global Variables
2. Add CUDA annotations
3. Feed with Strings
4. ???????????
5. PROFIT!

MD5

Basic implementation is easy
md5.cu

```
GG (c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */
GG (b, c, d, a, x[ 0], S24, 0xe9b6c7aa); /* 20 */
GG (a, b, c, d, x[ 5], S21, 0xd62f105d); /* 21 */
GG (d, a, b, c, x[10], S22, 0x2441453); /* 22 */
GG (c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */
GG (b, c, d, a, x[ 4], S24, 0xe7d3fbc8); /* 24 */
GG (a, b, c, d, x[ 9], S21, 0x21e1cde6); /* 25 */
GG (d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */
GG (c, d, a, b, x[ 3], S23, 0xf4d50d87); /* 27 */
GG (b, c, d, a, x[ 8], S24, 0x455a14ed); /* 28 */
GG (a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */
GG (d, a, b, c, x[ 2], S22, 0xfcefa3f8); /* 30 */
GG (c, d, a, b, x[ 7], S23, 0x676f02d9); /* 31 */
GG (b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */

/* Round 3 */
HH (a, b, c, d, x[ 5], S31, 0xfffa3942); /* 33 */
HH (d, a, b, c, x[ 8], S32, 0x8771f681); /* 34 */
HH (c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */
HH (b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */
HH (a, b, c, d, x[ 1], S31, 0xa4beea44); /* 37 */
HH (d, a, b, c, x[ 4], S32, 0x4bdecfa9); /* 38 */
HH (c, d, a, b, x[ 7], S33, 0xf6bb4b60); /* 39 */
HH (b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */
HH (a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */
HH (d, a, b, c, x[ 0], S32, 0xeaa127fa); /* 42 */
HH (c, d, a, b, x[ 3], S33, 0xd4ef3085); /* 43 */
HH (b, c, d, a, x[ 6], S34, 0x4881d05); /* 44 */
HH (a, b, c, d, x[ 9], S31, 0xd9d4d039); /* 45 */
HH (d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */
HH (c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */
HH (b, c, d, a, x[ 2], S34, 0xc4ac5665); /* 48 */

/* Round 4 */
II (a, b, c, d, x[ 0], S41, 0xf4292244); /* 49 */
II (d, a, b, c, x[ 7], S42, 0x432aff97); /* 50 */
II (c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */
II (b, c, d, a, x[ 5], S44, 0xfc93a039); /* 52 */
II (a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */
II (d, a, b, c, x[ 3], S42, 0x8f0ccc92); /* 54 */
II (c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */
II (b, c, d, a, x[ 1], S44, 0x85845dd1); /* 56 */
II (a, b, c, d, x[ 8], S41, 0x6fa87e4f); /* 57 */
II (d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */
II (c, d, a, b, x[ 6], S43, 0xa3014314); /* 59 */
II (b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */
II (a, b, c, d, x[ 4], S41, 0xf7537e82); /* 61 */
```

```
}
__device__ inline void MD5::FF(uint32_t &a, uint32_t b, u
    a = rotate_left(a+ F(b,c,d) + x + ac, s) + b;
}
__device__ inline void MD5::GG(uint32_t &a, uint32_t b, u
    a = rotate_left(a + G(b,c,d) + x + ac, s) + b;
}
__device__ inline void MD5::HH(uint32_t &a, uint32_t b, u
    a = rotate_left(a + H(b,c,d) + x + ac, s) + b;
}
__device__ inline void MD5::II(uint32_t &a, uint32_t b, u
    a = rotate_left(a + I(b,c,d) + x + ac, s) + b;
}

__constant__ uint8_t padding[64]={
    0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

__device__ void initMd5(MD5* m, const uint8_t* text, int l
    m->count.val = 0;
    m->state.val64[0] = 0xefcdab8967452301ull;
    m->state.val64[1] = 0x1032547698badcfeull;

    m->update(text, len);

    uint64_u bits = m->count;

    uint32_t index = (m->count.val32[0] >> 3) & 0x3f;
    uint32_t padLen = index < 56 ?
                56-index : 120 - index;
    m->update(padding, padLen);
    m->update(bits.val8, 8);
}
```

SHA1

Basic implementation is easy
SHA1.cu

```
// 4 rounds of 20 operations each, loop
S_R0(a,b,c,d,e, 0); S_R0(e,a,b,c,d, 1);
S_R0(b,c,d,e,a, 4); S_R0(a,b,c,d,e, 5);
S_R0(c,d,e,a,b, 8); S_R0(b,c,d,e,a, 9);
S_R0(d,e,a,b,c,12); S_R0(c,d,e,a,b,13);
S_R1(e,a,b,c,d,16); S_R1(d,e,a,b,c,17);
S_R2(a,b,c,d,e,20); S_R2(e,a,b,c,d,21);
S_R2(b,c,d,e,a,24); S_R2(a,b,c,d,e,25);
S_R2(c,d,e,a,b,28); S_R2(b,c,d,e,a,29);
S_R2(d,e,a,b,c,32); S_R2(c,d,e,a,b,33);
S_R2(e,a,b,c,d,36); S_R2(d,e,a,b,c,37);
S_R3(a,b,c,d,e,40); S_R3(e,a,b,c,d,41);
S_R3(b,c,d,e,a,44); S_R3(a,b,c,d,e,45);
S_R3(c,d,e,a,b,48); S_R3(b,c,d,e,a,49);
S_R3(d,e,a,b,c,52); S_R3(c,d,e,a,b,53);
S_R3(e,a,b,c,d,56); S_R3(d,e,a,b,c,57);
S_R4(a,b,c,d,e,60); S_R4(e,a,b,c,d,61);
S_R4(b,c,d,e,a,64); S_R4(a,b,c,d,e,65);
S_R4(c,d,e,a,b,68); S_R4(b,c,d,e,a,69);
S_R4(d,e,a,b,c,72); S_R4(c,d,e,a,b,73);
S_R4(e,a,b,c,d,76); S_R4(d,e,a,b,c,77);

// Add the working vars back into state
pState[0] += a;
pState[1] += b;
pState[2] += c;
pState[3] += d;
```

```
__device__ void CSHA1::Update(const uint8_t* pbData, ui
    uint32_t j = ((m_count[0] >> 3) & 0x3F);
    if((m_count[0] += (uLen << 3)) < (uLen << 3))
        ++m_count[1]; // Overflow
    m_count[1] += (uLen >> 29);

    uint32_t i;
    if((j + uLen) > 63){
        i = 64 - j;
        memcpy(&m_buffer[j], pbData, i);
        Transform(m_state, m_buffer);
        for( ; (i + 63) < uLen; i += 64)
            Transform(m_state, &pbData[i]);
        j = 0;
    }
    else
        i = 0;

    if((uLen - i) != 0)
        memcpy(&m_buffer[j], &pbData[i], uLen - i);
}

__device__ void CSHA1::Final()
{
    uint32_t i;

    uint8_t pbFinalCount[8];
    for(i = 0; i < 8; ++i)
        pbFinalCount[i] =
            static_cast<uint8_t>((m_count[!(i >= 4)]
                >> ((3 - (i & 3)) * 8) ) & 0xFF);

    Update((uint8_t*)"\200", 1);
    while((m_count[0] & 504) != 448)
        Update((uint8_t*)"\0", 1);
    Update(pbFinalCount, 8); // Cause a Transform()
```

Optimizations

- Shared Memory: Crypto objects, current Strings, possible characters.
  48kB is plenty...
- Constant Memory: Hash which is searched (copied by host), padding for algorithm. Constants in general faster as literals.
- Global Memory: local Memory

Lots of bithacks
Result: Not a single divergent branch

Best runtime with 64 threads x 64 blocks

String enumeration

Using 64 symbols for simplicity: [a-z],[A-Z],[0-9],' ', '-'

SHA1.cu / md5.cu

```
if(idx < 64){
    const char sym[] = {
        'a','b','c','d','e','f','g','h','i','j','k','l','m',
        'n','o','p','q','r','s','t','u','v','w','x','y','z',
        'A','B','C','D','E','F','G','H','I','J','K','L','M',
        'N','O','P','Q','R','S','T','U','V','W','X','Y','Z',
        '0','1','2','3','4','5','6','7','8','9',' ','-'};
     symbols[idx] = sym[idx];
}
__syncthreads();
```

utility.cu

```
//generate the string depending on ID and round
__device__ void genString(char* target, uint128_t it,
                          const int len, const char* symbols){
    #pragma unroll 4
    for(int i = 0; i < len-FIXED; it >>= 6, ++i)
        target[i] = symbols[it.val[0]&63];
}
```

Fixed suffix depending on the global thread Id

Needs 64^n total threads

128 Bit Integers

With more than 10 characters we need numbers larger than 2^{64} for the enumeration...

bigInt.h

```
class uint128_t{
public:
    __device__ uint128_t operator+(uint128_t const& rhs);
    __device__ uint128_t operator+(uint64_t  const  rhs);
    __device__ uint128_t operator-(uint128_t const& rhs);
    __device__ uint128_t operator*(const uint32_t rhs);
    __device__ uint128_t operator<< (const uint32_t rhs);
    __device__ uint128_t operator>> (const uint32_t rhs);
    __device__ uint128_t& operator<<=(const uint32_t rhs);
    __device__ uint128_t& operator>>=(const uint32_t rhs);

    __device__ bool      operator<  (const uint128_t& rhs);
    __device__ bool      operator>  (const uint128_t& rhs);

    __device__ uint128_t& operator++ ();

    __device__ void set (const uint64_t lower, const uint64_t upp
        {val[0]=lower; val[1]=upper;};
    __device__ uint128_t(const uint64_t lower, const uint64_t upp
        {val[0]=lower; val[1]=upper;};
    __device__ uint128_t(const uint128_t& src)
        {val[0]=src.val[0];val[1]=src.val[1];};
    __device__ uint128_t(const uint64_t v)
        {val[0]=v;          val[1]=0;};

    __device__ uint128_t()
        {val[0]=0;      val[1]=0;};
    uint64_t val[2];
};
__device__ uint128_t bigMul(const uint64_t a, const uint64_t b);
```

bigInt.cu

```
__device__ uint128_t uint128_t::operator+(uint128_t const& rhs){
    uint128_t x;
    //We use asm here to utilize the carry flag
    asm("add.cc.u64 %0, %2, %4;"          //Add lower with carry-ou
        "addc.u64   %1, %3, %5;"          //Add upper with carry-ir
        :"=l"(x.val[0])    , "=l"(x.val[1])   //0,1=result
        : "l"(val[0])      ,  "l"(val[1])      //2,3=lhs
        , "l"(rhs.val[0]) ,  "l"(rhs.val[1]));//4,5=rhs
    return x;
}
```

```
__device__ uint128_t uint128_t::operator-(uint128_t const& rhs){
    uint128_t x;
    //same as above
    asm("sub.cc.u64 %0, %2, %4;"
        "subc.u64   %1, %3, %5;"
        :"=l"(x.val[0])    , "=l"(x.val[1])   //0,1=result
        : "l"(val[0])      ,  "l"(val[1])      //2,3=lhs
        , "l"(rhs.val[0]) ,  "l"(rhs.val[1]));//4,5=rhs)
    return x;
}
```

Because assembly is FAST!
(and can use the carry flag)

Multi-GPU?

Since the problem is embarrassingly parallel and I had some time left. . .

md5.cu / SHA1.cu

```
void md5_auto(hash_t h, int len){
    if(len < 2){
        printf("Less than 2 chars not yet supported!");
        exit(-1);
    }

    int devNum;
    cudaGetDeviceCount(&devNum);
    cudaStream_t streams[devNum];
    for(int i = 0; i < devNum; i++){
        cudaSetDevice(i);
        cudaStreamCreate(&streams[i]);

        //Copy hash into constant memory
        cudaMemcpyToSymbol(d_h, &h, sizeof(hash_t));
        checkCUDAError("cudaMemcpyToSymbol");


        //We allocate shared memory for each threads string
        md5_autoGpu<<<BLOCKNUM/devNum, THREADNUM,
                      (len+sizeof(MD5))*THREADNUM,streams[i]
                      (len, i, devNum);
    }
    for(int i = 0; i < devNum; i++){
        cudaSetDevice(i);
        cudaDeviceSynchronize();
        checkCUDAError("md5_autoGpu");
    }
}
```

Performance of serial and parallel implementation

MD5, 4 characters
i3–3110M @ 2.40GHz: 37.5 s
GTX 480: 0.14 s

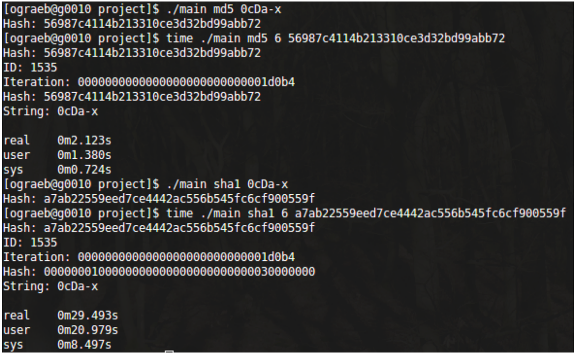
Speedup around 270 on an older GPU

Tests on GTX 680, 5 letters:
MD5: 4.710s
SHA1: 53.0s

MD5: 228 MH/s SHA1: 20.3 MH/s

Demo!

GTX 480



```
[ograeb@g0010 project]$ ./main md5 0cDa-x
Hash: 56987c4114b213310ce3d32bd99abb72
[ograeb@g0010 project]$ time ./main md5 6 56987c4114b213310ce3d32bd99abb72
Hash: 56987c4114b213310ce3d32bd99abb72
ID: 1535
Iteration: 000000000000000000000000001d0b4
Hash: 56987c4114b213310ce3d32bd99abb72
String: 0cDa-x

real    0m2.123s
user    0m1.380s
sys     0m0.724s
[ograeb@g0010 project]$ ./main sha1 0cDa-x
Hash: a7ab22559eed7ce4442ac556b545fc6cf900559f
[ograeb@g0010 project]$ time ./main sha1 6 a7ab22559eed7ce4442ac556b545fc6cf900559f
Hash: a7ab22559eed7ce4442ac556b545fc6cf900559f
ID: 1535
Iteration: 000000000000000000000000001d0b4
Hash: 0000000100000000000000000000000030000000
String: 0cDa-x

real    0m29.493s
user    0m20.979s
sys     0m8.497s
```

Disregard wrong display in the picture

Sources of Sample code etc.

- **Wikipedia**
  Information about MD5, SHA1, Hash functions
  https://en.wikipedia.org/wiki/

- **MD5 implementation**
  zedwood.com — C++ MD5 function
  http://www.zedwood.com/article/cpp-md5-function

- **SHA1 implementation**
  Code Project — CSHA1 - A C++ Class Implementation of the SHA-1 Hash Algorithm
  http://www.codeproject.com/Articles/2463/CSHA-A-C-Class-Implementation-of-the-SHA-Hash-A

- **Latex Beamer template**
  LaTeX Templates — Beamer Presentation
  http://www.latextemplates.com/template/beamer-presentation

The End