GPU-Based High Performance Password Recovery Technique for Hash Functions

WEIDONG QIU¹, ZHENG GONG²*, YIDONG GUO¹, BOZHONG LIU¹, XIAOMING TANG¹ AND YUHENG YUAN²
¹School of Electronic Information and Electrical Engineering
Shanghai Jiao Tong University
Shanghai, 200240 P.R. China
E-mail: qiuwd@sjtu.edu.cn
²School of Computer Science
South China Normal University
Guangzhou, 510631 P.R. China
E-mail: cis.gong@gmail.com

Abstract

Due to the development of GPGPU (General Purpose Graphic Processing Unit) technology, GPU has been applied in many computation tasks as accelerators. In this paper, a new password recovery technique for the standardized hash functions, MD5 and SHA1, are proposed by combining the optimization methods on GPU. The performance on AMD HD7970 is 2615 mc/s for SHA1 and 6877 mc/s for MD5, which is 10 times better than the original implementation. If the length of password is limited, our GPU-based technique makes it possible to recover password from hash values in a reasonable time.

Keywords: GPU, OpenCL, Password Recovery, Hash functions

1. INTRODUCTION

Stimulated by the great demand from the consuming market, GPU has been developed at an amazing speed over the last 10 years. Nowadays, the computing power of modern GPU can be 10 times faster than CPU. Thus, people are more and more interested in using GPU to do general purpose computation besides graphic processing. In 2007 Nvidia released CUDA (Compute Unified Device Architecture), which is a parallel computing platform that enables developers to use C or other high level language to utilize the computing power of GPU. OpenCL (Open Computing Language), which is another framework for writing programs that execute across platforms, has been released in 2008 by Khronos Group and then widely supported by the industry.

Due to its tremendous performance, GPGPU (General Purpose GPU) is now used in many areas to accelerate applications, such as medical, finance and aerospace engineering. Cryptography algorithms has been benefit from GPU computing as well.

* The corresponding author
For instance in [1][2], authors talked about improving AES performance by using GPU. [3] experimented with different computational granularity and memory strategies and found out the best implementation scheme for parallel AES encryption based on CUDA. In [4], authors found out the best implementation scheme for parallel AES encryption based on OpenCL. In [5], in addition to AES, authors also achieved parallel processing for Serpent and Twofish. [6] applied GPU to speed up the TEA/XTEA en/decryption. [7][8] discussed using GPU to accelerate elliptic curve cryptography. Moreover, GPU is also used to speed up RSA [9] and Lattice-based cryptography [10].

Currently password-based authentication is still the most widely used authentication method in information systems. To avoid the threat of malicious or compromised servers, it is widely-accepted that passwords cannot be transmitted or stored in plaintexts. Since hash functions have cryptographic properties such as the (second) preimage and collision resistances, it is a practical solution to them as one-way function to prove the knowledge of password in authentication protocols.

Table 1. MD5 cracking performances of the related works and our work

<table>
<thead>
<tr>
<th>Reference</th>
<th>Platform</th>
<th>Language</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weng et al. [12]</td>
<td>Nvidia GTX295</td>
<td>OpenCL</td>
<td>76.6 Mbps</td>
</tr>
<tr>
<td>Wu et al. [13]</td>
<td>Nvidia GTX9800+</td>
<td>CUDA</td>
<td>516 Mbps</td>
</tr>
<tr>
<td>Wang et al. [14]</td>
<td>Tianhe-1A¹</td>
<td>CUDA</td>
<td>670.55 Gbps</td>
</tr>
<tr>
<td>Lan et al. [15]</td>
<td>GPU cluster²</td>
<td>CUDA</td>
<td>Unknown³</td>
</tr>
<tr>
<td>Oclhashcat *</td>
<td>AMD HD7970(1G)</td>
<td>OpenCL</td>
<td>507.3 Gbps</td>
</tr>
<tr>
<td>Qiu et al. *</td>
<td>AMD HD7970(925M)</td>
<td>OpenCL</td>
<td>409.9 Gbps</td>
</tr>
</tbody>
</table>

* oclhashcat result: 8511 million passwords tries per second, the length of the password is 8.
* Our result: 6877 million passwords tries per second, the length of the password is 8.
¹ Tianhe-1A contains 7168 computing nodes. One node contains two Xeon X5670 and one Tesla M2050.
² Nvidia K20 * 4
³ Their platform can check 40 billion password per second, but the length of the password is not given and hence the performance can not be calculated.

However, as the hardware developing according to More's law, the security of such hash-function-based algorithms are being questioned. In fact, many works have been done on using GPU to speed up the cracking of such algorithms. In [11], authors proposed a MD5 crack method based on CUDA and their method reached 223 Mbps when the data size was 256 MB. In [12], the authors used OpenCL to achieve parallel processing for MD5 cracking. And the result was 17 times faster than the decryption software, John the ripper. In [13], it also achieved GPU-based MD5 hash reverse implementation by using CUDA and their implementation reached 516 Mbps when the block size was 128 bits. In [14], authors implemented brute force attack algorithm of MD5 crypt on Tianhe-1A using CUDA. Their results proved that their system can verify 18 billion potential keys per seconds. In [15], the authors presented a general platform of GPU cluster for brute-force cracking based on CUDA and its experiment results shown that its platform can check 40 billion keys per seconds by using its platform. Oclhashcat,
as a non-opensource free password cracker, claim to be the world’s fastest and they do archive a very good result, which is 8511 mc/s for MD5 and 2722 mc/s for SHA-1 on 7970 (1 Ghz).

In addition to applying GPU to crack hash-function-based algorithms, there’re also works on using GPU to accelerate such algorithms. For instance in [16], authors implemented a high throughput md5 algorithm on GPU. In [17], the authors presented a MD5 decryption algorithm in parallel on GPU cluster. The performance was over 100 times higher than CPU. Moreover, the paper [18] also used CUDA to execute the parallel version of AES and MD5 on Nvidia GPUs. The performances of our work and the works mentioned above are shown in table 1.

After reviewing the related works, we can find out that although many researches on the fast implementation of MD5 on GPU are published in literature, less of them have discussed the SHA1 hash function (which is also an international standard). Moreover, nearly all of the existing articles implementing their algorithm on Nvidia platforms by using CUDA, but less jobs have been done on OpenCL. Oclhashcat did a great work and despite non-opensource, they disclosed some of their optimizations, including the pre-computing mentioned in subsection 4.3 and the meet-in-the-middle mentioned in subsection 4.4 (also referenced in [13]). However we cannot archive their results only using their revealed optimizations.

In this paper, we implement a brute-force password decryption algorithm for MD5 and SHA1 on AMD GPU by using OpenCL. We also optimize the algorithm by consider the special property of OpenCL computing and hash functions. Although the performance results from oclhashcat are marginally better than ours, the optimization methods that proposed in our algorithms might be considered as a possible choice for the improvement of OCLHashCat library.

The remainder of this paper is organized as follows. In Section 2, the overall of password recovery algorithm and the algorithms of MD5 and SHA1 will be introduced. Section 3 will briefly describe the GPU architecture and general optimization principles. Optimization methods for both algorithms with OpenCL will be proposed in Section 4. In Section 5, the experiment result will be demonstrated and the conclusion will be given in Section 6.

2. OVERVIEW OF PASSWORD RECOVERY ALGORITHM

Our GPU based password recovery algorithm is based on brute-force attack. The overall pseudo code is shown in Algorithm 1.

---

**Algorithm 1. Password Recovery Algorithm**

**Input:** hash: the hash to be recovered; charset[0..n]: (n+1)-character charset; length: password length

**Output:** password: the password recovered

**While** HasNextPassword(charset, length) **do**

  password = IterateNextPassword(charset, length)
  h = Hash(password)

  **if** h == hash **then** **return** password
This algorithm is very straightforward. In this section, we will briefly introduce the password generation algorithm as well as the MD5, SHA1 message digest algorithm.

2.1 Password Generation Algorithm

The password generation algorithm should generate each candidate password in the password space once and only once. A straightforward and widely used algorithm is Index-based password generation algorithm, which can be described in Algorithm 2.

Algorithm 2. Index-based Password Generation Algorithm

Input: charset[0..n]: (n+1)-character charset; length: password length; index: an integer from 0 to (n+1)^length-1 mapping to a password;

Output: password: the password generated

Input: charset[0..n]: (n+1)-character charset; length: password length; index: an integer from 0 to (n+1)^length-1 mapping to a password;

Output: password: the password generated

<table>
<thead>
<tr>
<th>Input: charset[0..n]: (n+1)-character charset; length: password length; index: an integer from 0 to (n+1)^length-1 mapping to a password;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: password: the password generated</td>
</tr>
</tbody>
</table>

| | clear password[0..length-1] |
| | for i from 0 to length-1 do |
| | password[i] = charset[index % (n+1)] |
| | index = (n+1) |
| | end for |
| | return password |

Despite of the simplicity of the index-based password generation algorithm, the direct implementation of it is not quite suitable for GPU to archive best performance. The reason and the optimization for this algorithm will be discussed in the next section.

2.2 MD5

MD5 [19] is one in a series of message digest algorithms designed by Professor Ronald Rivest of MIT (Rivest, 1992). When analytic work indicated that MD5's predecessor MD4 was likely to be insecure, MD5 was designed in 1991 to be a secure replacement.

MD5 is an irreversible transformation transforming a set of data of length less than $2^{64}$ into a hash value of 128-bit length. The origin message is first padded with one bit ‘1’, several bit ‘0’ and a 64-bit integer with regarding to the length of origin bit string so that the length of the padded message is a multiple of 512. Then the padded message is broken up into chunks of 512-bit blocks and computations are performed on each of the block orderly.

The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted A, B, C, and D. These are initialized to certain constants. The main algorithm then operated on each 512-bit message block in turn, each block modifying the state. The processing of a message block consists of four similar stages, termed rounds; each round is composed of 16 similar operations based on non-linear function F, G, H, I and modular addition as well as left rotation.

Since recovery of a password, the length of which is longer than 55 characters, is
hopeless, we only focus on the first chunk of MD5.

2.3 SHA1

SHA1(Secure Hash Algorithm 1) [20] is one of the most popular message digest algorithms applied in information security applications and protocols, such as SSH, PGP, SSL and etc. It was designed by the United States National Security Agency and published by the United States National Institute of Standards and Technology.

The input message of SHA1 can be a bit string the length of which is less than \(2^{64}\). The algorithm will produce a fix 160 bit hash.

Just like MD5 and other hash algorithms, in pre-processing stage the bit '1' is appended to the origin message and no more than 512 bits '0' is appended until the MessageLength \(\equiv 448 \pmod{512}\). After that message length is appended in a 64-bit big-endian integer, which makes the message length a multiple of 512.

The padded message is broken into several 512-bit chunks and then each chunk is processed successively. Since the length of password is less than 55 characters generally, we only focus on the first chunk of SHA1.

3. GENERAL GPU OPTIMIZATION PRINCIPLES

GPU is quite from GPU in hardware architecture. Though using OpenCL or CUDA can easily transfer C or C++ code to GPU, it still needs to understand GPU architecture and modify the algorithm to archive the maximum performance. Thus, in this section we discuss general optimization principles on AMD platforms, which is the basic to optimize specific hash password recovery algorithms.

3.1 Branch Removing

GPU is designed to have overwhelming parallel computing power compared with CPU at the price of low branch processing ability.

In execution level, the 64 work-items within a wavefront (wave), which is the minimum schedule unit on AMD GPU, must be executed in the same lock step in a computing unit.

Consider the following code. Suppose work-group size is 128, which means 2 wavefronts in a work-group. Work-item 0, 2, 4, ..., 62 in the first wavefront go to branch1 while work-item 1, 3, 5...63 go to branch2. To finish the wavefront it costs time T1+T2 instead of \(\max(T1, T2)\). T1 refers to the time consumed by the first branch and T2 refers to the time consumed by the second. The work-group executes in time \(2*(T1+T2)\). Even only one work-item goes to divergent paths, it still costs \(2*(T1+T2)\) time to finish the work-group.

```c
__kernel void foo1()
{
    int id = get_local_id(0);
    if(id % 2 == 0)
        Branch1(); //consume time T1
```
else
    Branch2(); //consume time T2
}  

If the condition is slightly changed, for example, the first wavefront, work-item 0 to 63, all go into branch1 while the second wavefront, work-item 0 to 63, go into branch2. It will only cost (T1+T2) time to finish the work-group. That's a big difference.

if(id / 64 == 0)

Thus, GPU is suitable for computing-intensive tasks but not for branch-intensive ones. Removing as many branches as possible in kernel contributes to higher performance. Moving these branch-intensive parts to CPU is another option.

3.2 Memory Access Pattern

There are four different kinds of memory in GPU architecture and OpenCL memory model. Global memory can be accessed by every work-items and the capacity is the largest. But the bandwidth is the lowest. Constant memory can also be accessed by all work-items. It's a bit faster than global memory but the capacity is smaller and it's readonly by work-items. Local memory is only visible to work-items within a work-group. It's over 10 times faster than global memory. Private memory can only be accessed by one work-item. It provides the highest bandwidth but the capacity is the smallest.

<table>
<thead>
<tr>
<th>Table 2. Memory Parameters of AMD HD7970.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OpenCL Memory</strong></td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Global Memory</td>
</tr>
<tr>
<td>Constant Memory</td>
</tr>
<tr>
<td>Local Memory</td>
</tr>
<tr>
<td>Private Memory</td>
</tr>
</tbody>
</table>

Table 2 shows the memory parameters of AMD 7970 GPU. Memory access patterns always have great impact on performance. As for global memory, coalesced read/write is a good optimization method on Nvidia platform, which is also practicable on old AMD architecture before GCN. The latest 7900 series AMD GPU no longer support coalesced read/write any more. However, continuous addresses within work-groups still provide maximum performance.

For local memory, Bank Conflicts Avoidance is an important way to improve performance.
Take HD7970 for example, each compute unit contains a 64kb local data share, which is composed of 32 4-bytes wide and 512-bytes deep banks. Bank conflicts happen when work-items within half wavefront, that is work-item 0 to 31 or work-item 32 to 63, access slots in the same bank, as shown in Figure 1.

In that case, accesses are serialized instead of stalling the current wavefront and switching other active wavefronts in a given CU by the resource scheduler to hide memory access latencies. The compiler inserts wait operations prior to issuing operations that depend on the results. One exception is that when all work-items access the same address, data can be broadcast to all requestors, which doesn't result in bank conflicts.

Hence, bank conflict avoidance is necessary for performance sensitive applications. Local memory access patterns should be carefully designed so that different banks are accessed on the same circle.

### 3.3 Occupancy Rising

Occupancy refers to the ratio of the in-flight wavefront count of a certain kernel per CU to the maximum wavefront count limited by hardware. Take HD7970 for example, which may have a maximum of 40 wavefronts per CU, the occupancy is 50% if kernel can run 20 wavefronts per CU.

When a wavefront generates a long latency instruction, such as global memory access, the CU will schedule and process other independent wavefronts if available. Therefore, generally the occupancy is the higher the better to hide ALU and memory access latency and this improves performance.

The number of active wavefronts is limited by the size of private memory and local memory that a certain kernel used as well as the hardware limit. CU must have sufficient resources for each work-item in a work-group to run. Suppose \( W_{\text{max}} \) denote maximum number of wavefront per CU. \( W_{\text{wave}} \) denote the wavefront size which indicates maximum number of work-items in a wavefront. This number is 64 for most AMD GPUs. \( W_{\text{wg}} \) denote work group size, which should be multiple of \( W_{\text{wave}} \) to achieve best performance. \( R_{\text{cu}} \) denote number of registers in a computing unit. \( L_{\text{cu}} \) denote the capacity of local memory in a computing unit. \( R_{\text{wi}} \) denote the number of register requested by each work-item. \( L_{\text{wg}} \) denote the size of local memory shared by a work-group. Then
Occupancy can be expressed by

\[
\begin{align*}
\text{WAV}_{\text{reg}} &= \left( \frac{R_{cu}}{W_{wg}} \right) \times \left( \frac{W_{wg}}{W_{\text{warm}}} \right), \\
\text{WAV}_{\text{lm}} &= \left( \frac{L_{cu}}{L_{wg}} \right) \times \left( \frac{W_{wg}}{W_{\text{warm}}} \right)
\end{align*}
\]

\[\text{Occupancy} = \min(\text{WAV}_{\text{reg}}, \text{WAV}_{\text{lm}}, \text{WAV}_{\text{max}}) \times 100\% \tag{2}\]

Since registers and LDS are much faster than global memory, in some cases they can be used to buffer data from global memory. However, it still has limitation when using them because using more registers or LDS may lead to lower occupancy. They should be used the less the better when other conditions are the same. Usage of different kinds of memory should be balanced to achieve better performance.

### 4. OPTIMIZATIONS

In this section, we will discuss the optimization methods for the MD5 and SHA1 password recovery algorithm.

#### 4.1 Password Generation

Generally password can be either generated on CPU or GPU. The former one reduce the GPU computing workload but the passwords generated need to be transferred to GPU memory through PCI-E bus.

Pipeline technology can be used to hide the memory transfer latency if the time cost by transfer is less that by compute. However, passwords of high performance algorithm like MD5, SHA1, cannot be generated on CPU since the hash computing on GPU is very fast. Thus data transfer will become the bottleneck. Password must be generated, or partially, on GPU.

As mentioned in section 2, a straightforward and widely used algorithm is index-based password generation algorithm described in algorithm 2. In implementation, the index should be a 64-bit integer instead of a 32-bit integer or the password space is too small. However, since GPU is a 32-bit hardware, the 64-bit mod and divide instruction and the byte assignment is slow, which result in long time consumed by password generation compared with the hash computing. As the length of password grows up, the time consumed by password generation will be longer. Therefore, we exploit an improved password generation algorithm in our implementation.

Instead of generating password one by one, we choose to generate a batch of passwords each time. Password can be considered as the concat of password-prefix and password-suffix. Suppose the charset is \([a-z]\) and the length of password is 9. Firstly, we generate a prefix-table which contains password prefix from "aaaa", "baaa" to "zzzz". We
consider the 4-byte prefix as a 32-bit integer so that the prefix table is a 32-bit integer array with the length of array is $26^4$. Password suffix can be generated as the basic generation algorithm.

### Algorithm 3. Improved Password Generation Algorithm

| Input: charset[0..n]: (n+1)-character charset; length: password length; index: an integer from 0 to $(n+1)^{length-1}$ mapping to a password; ptLength: length of prefix table, which is $(n+1)^4$; prefix_table[0..(ptLength-1)]: 32-bit integer array which stores the prefix |
| Output: password: generated password |
| Clear password[0..(length-1)] |
| prefix = prefix_table[index % ptLength] |
| suffix = GeneratePassword(index / ptLength, charset, length - 4) |
| password = concat(prefix, suffix) |

In Algorithm 3, the Function GeneratePassword() can just use algorithm 2 mentioned before. Since a batch of passwords with the same suffix can be generated with one suffix, it's reasonable to generate the suffix on CPU and transfer it to GPU. And once the prefix table is generated, it doesn't change during the recovery process and thus can be generated and transferred to global or constant memory just one time. The cost of concat is also low because it's just a 32-bit integer XOR instruction.

### 4.2 Instruction

Reduction of total number of instructions in the process procedure always contributes to higher performance. AMD GPU provides some instructions can be used to speedup SHA1 implementation.

One is the BFI_INT instruction. The three macro functions in the code block below all generate the same result. F1 is the origin expression which takes 4 instructions. As also referred in [16], F2 is another expression that reduces 1 instruction. F3 uses the bitselect function in OpenCL which utilizes the BFI_INT instruction provided by AMD GPU and only takes 1 instruction.

```c
#define F1(b, c, d) ((b&c) | (~b)&d)
#define F2(b, c, d) (d^(b&(c^d)))
#define F3(b, c, d) bitselect(d, c, b)
```

Another instruction can be used is rotate(). Though x86 support rol(rotate left) instruction, C and many other high-level programming language do not provide rotate operator. Therefore, generally in C language the implementation 32-bit integer rotate left can be expression like (1) in the following code block.

```c
#define R0TL(x, n) ((x<<n) | (x>>(32-n)))
#define R0TL(x, n) rotate(x, n)
```

The rotate function provided by OpenCL can be used to reduce the count of
instructions from 3 to 1.

4.3 Algorithm Modification

Since SHA1 and MD5 both have four or five different non-linear functions to use when processing a single block, it may expect several if-case to choose different non-linear function with regard to the i-th steps, as shown in the following pseudo code.

```
//Main loop for MD5
for i from 0 to 63
  if 0 ≤ i ≤ 15 then
    F := (B and C) or ((not B) and D)
    g := i
  else if 16 ≤ i ≤ 31
    F := (D and B) or ((not D) and C)
    g := (5 × i + 1) mod 16
  else if 32 ≤ i ≤ 47
    F := B xor C xor D
    g := (3 × i + 5) mod 16
  else if 48 ≤ i ≤ 63
    F := C xor (B or (not D))
    g := (7 × i) mod 16
  dTemp := D
  D := C
  C := B
  B := B + leftrotate((A + F + K[i] + M[g]), s[i])
A := dTemp
end for

//Main loop for SHA1
for i from 0 to 79
  if 0 ≤ i ≤ 19 then
    f = (b and c) or ((not b) and d)
    k = 0x5A827999
  else if 20 ≤ i ≤ 39
    f = b xor c xor d
    k = 0x6ED9EBA1
  else if 40 ≤ i ≤ 59
    f = (b and c) or (b and d) or (c and d)
    k = 0x8F1BBCDC
  else if 60 ≤ i ≤ 79
    f = b xor c xor d
    k = 0xCA62C1D6
  temp = (a leftrotate 5) + f + e + k + w[i]
  e = d
  d = c
  c = b leftrotate 30
  b = a
  a = temp
end for
```

GPU is weak to process switches. Fortunately, since these switches do not depend on runtime values, thus they can be removed by manually unrolling the main loop.

Another thing can be optimized is the register value swapping. As shown in the above codes, in each iteration the value of A, B, C, D, E (SHA1) is swapped. Instead of swapping register value by using five or size 32-bit assignment instruction, just swapping the register in code can achieve the same effect. The following codes show part of unrolled iterations of SHA1, MD5 can also be optimized in the same way.

```c
#define P(a,b,c,d,e,x) (e += rotate((uint)a,(uint)5)) + bitselect(d,c,b) + 0x5A827999 + x; b = rotate((uint)b,(uint)30);
......
P( c, d, e, a, b, W(18) ); //SHA1 round 18
P( b, c, d, e, a, W(19) ); //SHA1 round 19
#undef P
#define P(a,b,c,d,e,x) (e += rotate((uint)a,(uint)5)) + b^c^d + 0x6ED9EBA1 + x; b = rotate((uint)b,(uint)30);
P( a, b, c, d, e, W(20) ); //SHA1 round 20
```
For SHA1, the unrolled version is still not quite GPU friendly. The main problem is the word expansion. The word array pre-computed, which contains 80 32-bit integers, should be stored in registers for higher performance. However, registers are limited resources for GPU. Using such many registers will lead to low occupancy and performance. Therefore, the alternate method for computing SHA-1, proposed in [21], should be used in GPU implementation. Instead of using 80 registers, the words for round 16 to 79 is computed in-flight to save 64 registers as shown in the following macro.

```c
#define W(i) \  
( \  
   s = i & 0x0f, \  
   W[s] = W[(s+13)&0x0f] ^ W[(s+8)&0x0f] ^ W[(s+2)&0x0f] ^ W[(s)&0x0f], \  
   W[s] = rotate(W[s], 1) \  
)
```

In [22] the oclhashcat had revealed a Pre-Computing optimization which further reduce the count of instructions used in word expansion process.

### 4.4 Hash Reverse

In password recovery, computing the hash of candidate password is to verify whether it will generate the target hash we want. We do not need to fully compute hash of each candidate password if there exists a way to tell whether the password is correct or not. And that does exist.

Take SHA1 as the example, we consider the last 5 iterations of the main loop as shown in the following code.

```c
P( a, b, c, d, e, W(75) ); //SHA1 Iteration 75
P( e, a, b, c, d, W(76) ); //SHA1 Iteration 76
P( d, e, a, b, c, W(77) ); //SHA1 Iteration 77
P( c, d, e, a, b, W(78) ); //SHA1 Iteration 78
P( b, c, d, e, a, W(79) ); //SHA1 Iteration 79
ctx->h0 += a;
ctx->h1 += b;
ctx->h2 += c;
ctx->h3 += d;
ctx->h4 += e;
```

After iteration 75, the changing value of word e has nothing to do with the input password. It is left rotated 30 bits in iteration 77 and added the initial value 0xC3D2E1F0
\[ e_{\text{final}} = \text{rotate}(e_{75}, 30) + 0xC3D2E1F0 \]
\[ e_{75} = \text{rotate}((e_{\text{final}} - 0xC3D2E1F0), 2) \]

Since \( e_{\text{final}} \) is got from the SHA-1 hash, \( e_{75} \) can be easily pre-computed before the recovery process. Thus, it can be used to pre-check whether the password can be a candidate or not. For the collision probability for \( e_{75} \) pre-check is quite low, around \( 1/2^{32} \), it can be thought as saving 4 iteration and 5 additions for the password SHA1 computing.

The same method can also be applied on MD5 to reduce 3 iterations. In fact, authors in [13] find another way to reduce the 15 MD5 iterations and oclhashcat claims a closed method to reduce more. Combining these will result in about 28% performance gain of MD5.

5. EXPERIMENT RESULT

In this section the result of the experiment will be introduced. The raw MD5, SHA1 password recovery algorithm as well as the optimization methods are implemented and run on AMD HD7970 and 6470M. The former one is a high end desktop graphic card and uses latest GCN architecture while the latter one is a low end laptop graphic card and uses the old VLIW architecture. Table 3 shows the experiment environment.

<table>
<thead>
<tr>
<th>GPU</th>
<th>OS(64-bit)</th>
<th>IDE</th>
<th>SDK</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>HD 7970 (925Mhz)</td>
<td>Win 7</td>
<td>VS 2010</td>
<td>AMD APP v2.8</td>
<td>AMD CCC 14.9</td>
</tr>
<tr>
<td>HD 6470M</td>
<td>Win 8</td>
<td>VS 2010</td>
<td>AMD APP v2.8</td>
<td>AMD CCC 14.4</td>
</tr>
</tbody>
</table>

Table 4 demonstrates the performance of the raw MD5 and the performance after optimization using the method described in section 4 on both GPUs. The raw MD5 implements the algorithm described in the code block of subsection 4.3 and the password generation algorithm described in Algorithm 2.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>HD 7970(925Mhz)</th>
<th>HD 6470M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance (mc/s)</td>
<td>Acceleration Ratio</td>
</tr>
<tr>
<td>None</td>
<td>1710</td>
<td>1.00</td>
</tr>
<tr>
<td>Alg</td>
<td>2975</td>
<td>1.74</td>
</tr>
<tr>
<td>Pwd</td>
<td>3011</td>
<td>1.76</td>
</tr>
<tr>
<td>Alg+Pwd</td>
<td>3490</td>
<td>2.04</td>
</tr>
<tr>
<td>Alg+Pwd+Rev</td>
<td>5098</td>
<td>2.98</td>
</tr>
<tr>
<td>Alg+Pwd+Rev+Ins</td>
<td>6877</td>
<td>4.02</td>
</tr>
</tbody>
</table>

Table 4. Performance of MD5 on AMD GPU

The Alg, Pwd, Rev and Ins in the table refers to algorithm optimization (including oclhashcat’s pre-computing optimization for SHA-1), password generation optimization,
hash reverse and instruction optimization. There exists 16 kinds of combinations of 4 different optimization methods and we implement 5 of them and list the result in Table 4. By applying all optimizations, we archive around 300% performance gain on both GPUs.

Table 5 shows the performance of SHA-1 on both GPUs. From table 5 we can see that algorithm and password generation optimizations mainly contribute to the final acceleration factor. Since both of these two optimization reduce the use of registers, especially the word expand optimization of SHA-1 reduce register from 80 to 16, the combination of them achieve much more performance gain because of higher occupancy. Hash reverse and instruction optimization do not enhance as much as MD5 do. The reverse of SHA-1 reduces 4 out of 80 iterations and that number is 18 out of 64 for MD5. As for instructions, 32 out of 46 iterations can use bitselect to optimize. However, only 16 out of 76 iterations of SHA-1 can be optimized after reverse.

### Table 5. Performance of SHA1 on AMD GPU

<table>
<thead>
<tr>
<th>Optimization</th>
<th>HD 7970(925Mhz)</th>
<th></th>
<th>HD 6470M</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance (mc/s)</td>
<td>Acceleration Ratio</td>
<td>Performance (mc/s)</td>
<td>Acceleration Ratio</td>
</tr>
<tr>
<td>None</td>
<td>180.9</td>
<td>1.00</td>
<td>8.02</td>
<td>1.00</td>
</tr>
<tr>
<td>Alg</td>
<td>654.6</td>
<td>3.61</td>
<td>35.3</td>
<td>4.40</td>
</tr>
<tr>
<td>Pwd</td>
<td>240.4</td>
<td>1.33</td>
<td>9.54</td>
<td>1.19</td>
</tr>
<tr>
<td>Alg+Pwd</td>
<td>2385</td>
<td>12.56</td>
<td>129.5</td>
<td>16.14</td>
</tr>
<tr>
<td>Alg+Pwd+Rev</td>
<td>2563</td>
<td>14.16</td>
<td>138.3</td>
<td>17.21</td>
</tr>
<tr>
<td>Alg+Pwd+Rev+Ins</td>
<td>2615</td>
<td>14.45</td>
<td>142.6</td>
<td>17.78</td>
</tr>
</tbody>
</table>

Ultimately we achieve 6877 million passwords tries per second for MD5 and 2615 million for SHA-1 on 7970. That means an 8-length MD5 encrypted password composed of highalpha, lowalpha and number can be recovered in 8.8 hour while 31.4 hour for SHA-1.

### 6. CONCLUSIONS

Modern GPU is capable of high performance general purpose computation in many areas besides graphic processing. In this paper, we implement a MD5 and SHA-1 password recovery algorithm on AMD GPUs and further optimize our implementation in password generation, instruction, algorithm and hash reverse.

The experiment shows that the combination of these optimizations finally achieves over 10 times performance gain comparing with the origin one. Though we only implemented MD5 and SHA-1 on GPU, these optimizations can also be applied in other password recoveries with similar hash functions.

### REFERENCES

1. O. Harrison and J. Waldron, “AES encryption implementation and analysis on commodity graphics processing units,” in 9th Workshop on Cryptographic
213

MD5 on GPU Cluster. In International Conference on Information Security and Artificial Intelligence (ISAI2010), Chengdu, China.


