International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 9, Number 2, pages 370–389, July 2019

Implementation of high speed hash function Keccak on GPU

Thuong Nguyen Dat, Keisuke Iwai, Takashi Matsubara, and Takakazu Kurokawa
Department of Computer Science, National Defense Academy
1-10-20 Hashirimizu Yokosuka Kanagawa 239-8686, Japan
Email: {ed17004, iwai, matubara, kuro}@nda.ac.jp

Received: October 26, 2018
Revised: April 7, 2019
Accepted: May 7, 2019
Communicated by Yasuyuki Nogami

Abstract

Nowadays, a hash function is used for password management. The hash function is desired to possess the following three characteristics: Pre-Image Resistance, Second Pre-Image Resistance, and Collision Resistance. They are set on the assumption that it is computationally difficult to find the original message from a given hash value. However, the security level of the password management will be further reduced by implementing a high speed hash function on GPU. In this paper, the implementation of high speed hash function Keccak-512 using the integrated development environment CUDA for GPU is proposed. The following four techniques are used in order to speed up its implementation. The first one is reforming lookup tables from 2 dimensional arrays to 1 dimensional arrays at step ρ and π. The second is an investigation into the effect of using constant memory and shared memory for constant values. The third is the finding out the optimal configuration of blocks-threads, then evaluate the implementation according to the occupancy. And the last one is using CUDA streams with overlapping to hide the overhead of data transfer and GPU processing.

As the result, the throughput of implemented Keccak on GeForce GTX 1080 achieved up to maximum 64.58 GB/s. It is about 14.0 times faster than the previous research result. In addition, the safety level of Keccak is also discussed at the point of Pre-Image Resistance especially. In order to implement a high speed hash function for password cracking, we developed a special program for passwords up to 71 characters. Moreover, the throughputs of 2 times as well as 3 times hash are also evaluated. It is proved that multiple times hash is possible to greatly improved the security level of Keccak with password management. The throughput of hashing password with a large number of iterations confirmed the effect was about 90%. That is the time required to hash one password for 1000 times was almost the same as the total time to sequentially hash 900 passwords.

Keywords: Implementation, high speed, Keccak, GPU, CUDA, password management

1 Introduction

Opportunities to handle personal information and highly confidential information on the global network are increasing. It is necessary to use encryption techniques and the password management in order to protect them.

[0]This is a new paper based on the paper of Thuong Nguyen Dat, Keisuke Iwai, and Takakazu Kurokawa: Implementation of high speed hash function Keccak using CUDA on GTX 1080. 4th International Workshop on Information and Communication Security (WICS), CANDAR'17, 2017.

The hash function is a typical technique to be used for the password management. It has the feature that the same hashed value can always be obtained from the same input value, but a completely different value can be obtained from a different input value. Since it contains an irreversible one-way function, it is not easy to calculate the input value from the hashed value. However, it is becoming possible to find out an input value from the hashed value by performing an exhaustive search due to the improvements of processing speed of computers. As a result, the safety level of hash functions has been declining.

Keccak was selected as the winner of the competition on October 2, 2012 for the next-generation hash function SHA-3. The official version has been published as FIPS PUB 202 [1] on August 5, 2015. A hash function Keccak bases on the sponge construction, and is corresponding to the progress of research on the attack against MD5 and SHA-1.

There are many research results implementing Keccak on GPU [2-6]. This paper targeted the implementation result of [6] to compare our implementation result. [6] presented an implementation of Keccak on GPU with tree structures. As a result, their maximum throughput reached 1.183 GB/s on GeForce GTS 250 graphics card.

In this paper, an implementation of a high speed hash function Keccak-512 on GeForce GTX 1080 using integrated GPU development environment CUDA will be shown, and its processing speed will be compared with the previous research result. Moreover, security level of Keccak in password management will be discussed following our GPU implementation.

2 Keccak

The National Institute of Standards and Technology (NIST) of the United States selected Keccak from the SHA-3 candidates for the next generation cryptographic hash function standard on October 2, 2012. Keccak is a hash function with a sponge construction [7].

2.1 Hash function

The hash function receives an arbitrary length of message, and outputs a fixed number of bits as "fingerprint", or "input message digest". It is inferred to be impossible to create two messages with the same message digest or to prepare a message according to a message digest specified in advance. The hash function is applied to password management, authentication, and electronic signature. The hash function is desired to possess following three characteristics: Pre-Image Resistance, Second Pre-Image Resistance, and Collision Resistance. They are set on the assumption that it is computationally difficult to find a message outputting a given hash value.

The hash functions MD4, MD5, SHA-1, RIPEMD are based on Merkle-Damgard construction. The essential disadvantage of this construction is that it is vulnerable to length-extension attacks [8], and parallel processing cannot be done. The hash function Keccak is a family based on sponge construction, that generalize the concept of cryptographic hash function with infinite output and can perform quasi all symmetric cryptographic functions.

2.2 Sponge construction

The sponge construction bases on the fixed length permutation and padding. The sponge construction can be divided into the following two phases: "absorbing" and "squeezing".

Figure 1 shows absorbing phase as the first phase of sponge construction. The input message M is padded to generate a padded message Mp at the first step. The padded message Mp is then divided into multiple blocks consisting of r bits. And XOR operation is calculated with each block of Mp and the first r bits of *internal state (IV)* or the r-bit output of f. Here, f is the Keccak-f permutation function to update internal state. Symbols r and c are the bitrate and the capacity respectively. The initial values of r and c are set to 0. The internal state size is $1600 = r + c$ bits for SHA-3, and $r = 576bits, c = 1024bits$ in case of Keccak-512 and SHA3-512.

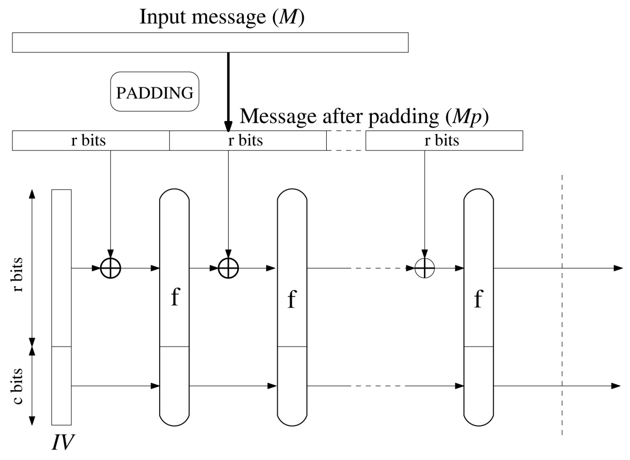

Figure 1: Absorbing phase [9].

The second phase of sponge construction is squeezing, which is shown in Figure 2. In squeezing phase, the Keccak-f permutation function is repeated up to generate the necessary length of output Z to be equal to the output bit length as the hashed value (512 bits in case of Keccak-512).

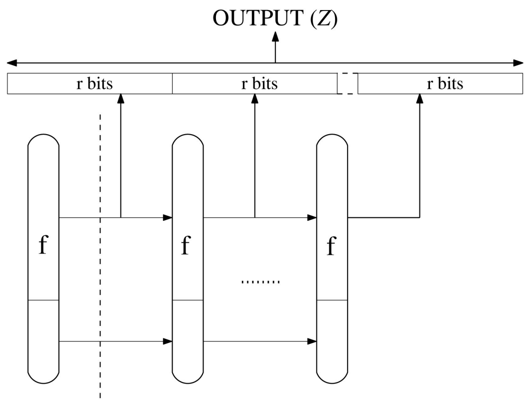

Figure 2: Squeezing phase [9].

2.3 Keccak-f permutation function

Keccak's basic stirring function is selected from the set of Keccak-f functions represented by seven Keccak-f [b] (b ∈ 25, 50, 100, 200, 400, 800, 1600). The bit number b is the stirring width, which corresponds to the magnitude of the internal state held by the function. The permutation function used by Keccak of Keccak-512 is Keccak-f [1600], and it performs 24 rounds of processing. The Keccak-f permutation function calculates the three-dimensional state by the step of performing the XOR operation of the four steps θ, ρ, π and χ with the round constant. Figure 3 shows the state array that is a representation of Keccak's three-dimensional state with labeling convention x, y, z in the case of Keccak-512, which have $b = 1600$ and quantity $b/25 = 64$ denoted by w. In this figure, as well as the following three figures, one small cube represents one bit.

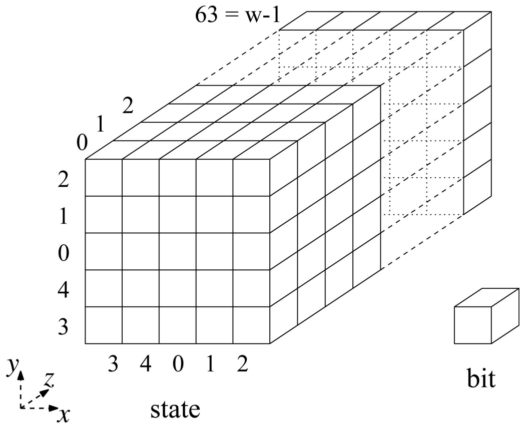

Figure 3: The state array with labeling [9].

In each round, the following processing is performed.
(See FIPS PUB 202[1] for details)

2.3.1 Step θ

In this step, an XOR operation is performed through the following steps ①, ②, ③ applied to a single bit as shown in Figure 4. The effect of θ is to XOR each bit in the state with the parities of two columns in the array.

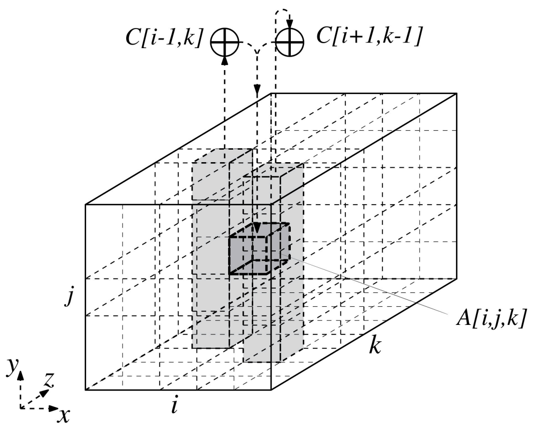

Figure 4: Illustration of step θ [9].

For all pairs (x, z) and triples (x, y, z) such that $0 \leq x < 5$, $0 \leq y < 5$, and $0 \leq z < w$, let

① $C[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$.

② $D[x, z] = C[(x - 1) \bmod 5, z] \oplus C[(x + 1) \bmod 5, (z - 1) \bmod w]$.

③ $A[x, y, z] = A[x, y, z] \oplus D[x, z]$.

Here, $C[x, z]$, $D[x, z]$ are intermediate variables, that store the x-coordinate of the column $\#z$ to calculate next status of a bit $A[x, y, z]$ in the array.

Pseudo-code description of step θ can be represented through the following (1), (2), (3). Note that y in these equations represent z in Figure 4 above. Here, $A[x, y]$ is a state status and $C[x]$, $D[x]$ are intermediate variables, $ROTL(C[i], r)$ is a cyclic right shift operation modifying the position of

$C[i]$ to $C[i+r]$, that means $ROTL(C[x+1],1)$ is equal to $C[x+1]$ with $z = z-1$ position in Figure 3. All operations with indices are performed with modulo 5.

$$C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4], \quad 0 \le x \le 4 \tag{1}$$

$$D[x] = C[x-1] \oplus ROTL(C[x+1],1), \quad 0 \le x \le 4 \tag{2}$$

$$A[x,y] = A[x,y] \oplus D[x], \quad 0 \le x,y \le 4 \tag{3}$$

2.3.2 Step ρ

Step ρ rote the bits by a length on the same z-axis called the offset, which depends on the fixed x and y coordinates. Equivalently, for each bit, the z coordinate is modified by adding the offset, module $w = b/25$. The algorithm of step ρ is performed through the following steps ① and ②.

① Let $(x,y) = (0,1)$.

② For t from 0 to 23:
for all z such that $0 \le z < w$, let $A[x,y,z] = A[x,y,(z-(t+1)(t+2)/2) \bmod w]$;
let $(x,y) = (y,(2x+3y) \bmod 5)$.

Figure 4 shows an illustration of ρ for the case $b = 200$, that means $w = 8$ and more simplicity than the case $b = 1600$, $w = 64$ of Keccak-512.

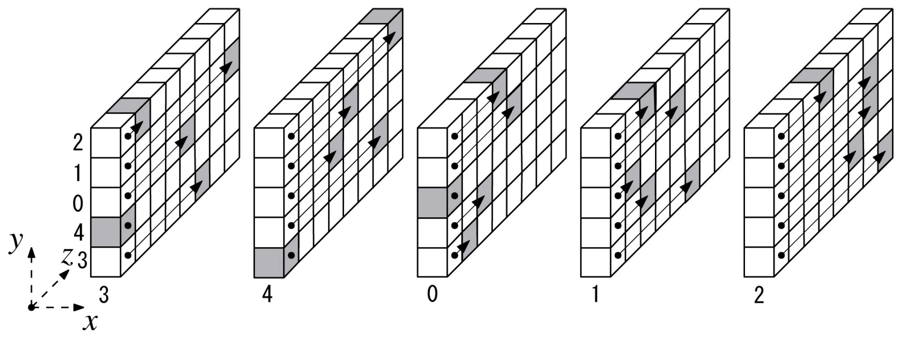

Figure 5: Illustration of step ρ for $b = 200, w = 8$ [9].

The offsets for each z that result from the computation in Step ② are listed in Table 1 as the case of Keccak-512, which $b = 1600, w = 64$.

Table 1: Rotation offsets $r[x,y]$ for $b = 1600, w = 64$ [10].

y \ x	3	4	0	1	2
2	25	39	3	10	43
1	55	20	36	44	6
0	28	27	0	1	62
4	56	14	18	2	61
3	21	8	41	45	15

2.3.3 Step π

Step π illustrates for any bit on the xy-plane as shown in Figure 6 respectively following step ①:
① For all triples (x,y,z) such that $0 \leq x < 5, 0 \leq y < 5$, and $0 \leq z < w$, let
$A[x, y, z] = A[(x + 3y) \bmod 5, x, z]$.

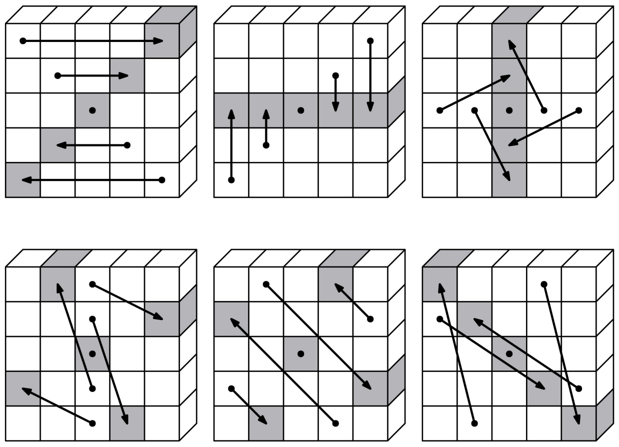

Figure 6: Illustration of step π [9].

Pseudo-code for two steps ρ and π that move bit positions of internal state can be written as following (4).

$$B[y, 2x + 3y] = ROTL(A[x, y], r[x, y]), \quad 0 \leq x, y \leq 4 \tag{4}$$

Here, $B[x, y]$ is an intermediate variable and is same as $C[x]$ and $D[x]$. $r[x, y]$ is the rotation offset, which is given in Table 1 above.

2.3.4 Step χ

Step χ effects to XOR each bit with a non-linear function of two other bits in its row, as illustrated in Figure 7 and step ① below.

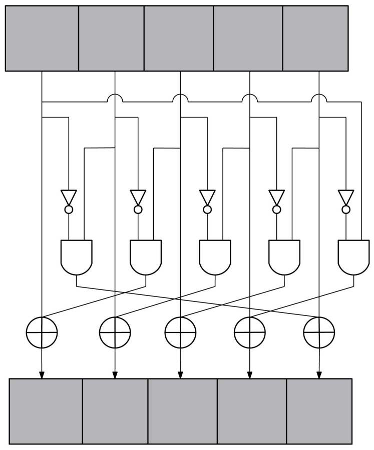

Figure 7: step χ [9].

① For all triples (x,y,z) such that $0 \leq x < 5, 0 \leq y < 5$, and $0 \leq z < w$, let $A[x, y, z] = A[x, y, z] \oplus ((A[x + 1) \bmod 5, y, z] \oplus 1) \cdot A[(x + 2) \bmod 5, y, z])$.

Pseudo-code for step χ can be represented as following (5).

$$A[x, y] = B[x, y] \oplus (\overline{B[x + 1, y]} \cdot B[x + 2, y]), \quad 0 \leq x, y \leq 4 \tag{5}$$

2.3.5 Step ι

Finally, in step ι, XOR operations are performed with the round constant $RC[i]$ and the first bit of the whole state as shown in (6). Here, i is the round index.

$$A[0, 0] = A[0, 0] \oplus RC[i] \tag{6}$$

Table 2 shows the round constant $RC[i]$ for 24 round of Keccak-f permutation function.

Table 2: Round constant $RC[i]$ [10].

$RC[0]$	$0x0000000000000001$	$RC[12]$	$0x000000008000808B$
$RC[1]$	$0x0000000000008082$	$RC[13]$	$0x800000000000008B$
$RC[2]$	$0x800000000000808A$	$RC[14]$	$0x8000000000008089$
$RC[3]$	$0x8000000080008000$	$RC[15]$	$0x8000000000008003$
$RC[4]$	$0x000000000000808B$	$RC[16]$	$0x8000000000008002$
$RC[5]$	$0x0000000080000001$	$RC[17]$	$0x8000000000000080$
$RC[6]$	$0x8000000080008081$	$RC[18]$	$0x000000000000800A$
$RC[7]$	$0x8000000000008009$	$RC[19]$	$0x800000008000000A$
$RC[8]$	$0x000000000000008A$	$RC[20]$	$0x8000000080008081$
$RC[9]$	$0x0000000000000088$	$RC[21]$	$0x8000000000008080$
$RC[10]$	$0x0000000080008009$	$RC[22]$	$0x0000000080000001$
$RC[11]$	$0x000000008000000A$	$RC[23]$	$0x8000000080008008$

2.4 Keccak and SHA3

Given a message M, the SHA3-512 hash function is defined from the KECCAK[c] function [1] by appending a two-bit suffix to M and by specifying the length of the output, as follows:

SHA3-512(M) = KECCAK[1024](M||01, 512).

The difference with Keccak is that a 2-bit string 01 is added to the message before padding in SHA-3. So, SHA-3 is a hash function based on Keccak, and almost the same with Keccak, but their output values are different for the same input message.

3 GPGPU and CUDA Programming

3.1 Overview

Initially, CPU in a computer performs all of the arithmetic operations and instructions. However, in recent years, the use of the Graphics Processing Unit (GPU) specifically developed for graphic processing has advanced. Unlike the CPU, the GPU has thousands of cores and has high computing function. Due to its characteristics, research on GPGPU (General Purpose Computation on Graphics Processing Unit) for numerical calculation has been actively conducting.

Since GPGPU was originally programmed using a graphics API (Application Programming Interface) such as OpenGL, Direct X and shader language, GPGPU programming was not easy. However, in 2006 NVIDIA released CUDA (Compute Unied Device Architecture) [11], a development environment using C language, GPGPU programming became easier and is widespread. Furthermore, accelerator board such as Tesla, which is not for the graphics processing, but for the GPGPU, has been adopted in many supercomputers. Now, GPGPU is one of the fastest device for numerical calculations.

3.2 CUDA Programming and GPU construction

The hierarchical structure of GPU consists of threads, blocks, and grids as shown in Figure 8. A thread is a minimum unit for executing a program, and a group of threads becomes a block. In addition, the gathered blocks become a grid.

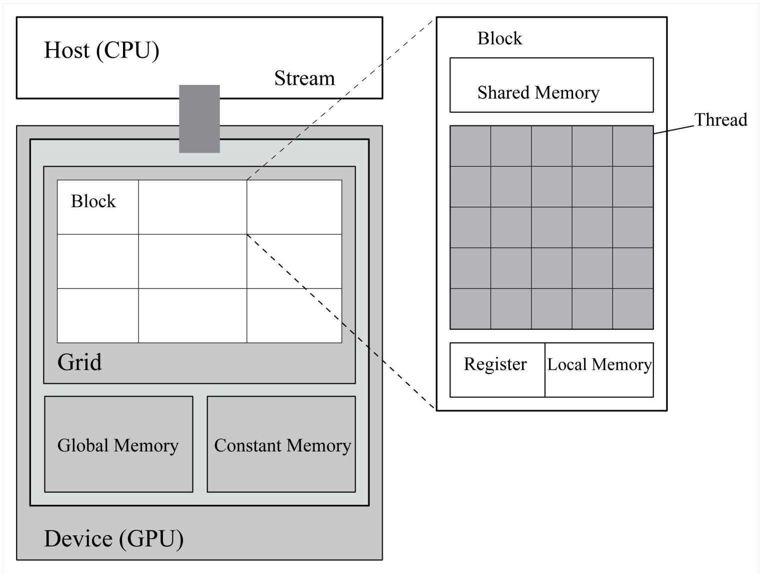

Figure 8: Hierarchical structure of GPU.

A streaming multiprocessor and peripheral circuits such as memory are arranged in block shapes in GPU. One block is assigned to one streaming multiprocessor.

In Pascal architecture, 64 single-precision shader processors and also 32 double-precision processors are installed to one streaming multiprocessor, and parallel processing is performed by these streaming processors. In the program structure of CUDA, one thread is calculated by one streaming processor.

Shared memory and registers are equipped with each streaming multiprocessor. These memories have small capacity, but high access speed.

There is a global memory that can be accessed from all streaming processors. This memory has a slower access speed than shared memory, registers, etc. but has large capacity.

CUDA, provided by NVIDIA, is an integrated development environment of C language for GPUs. It consists of a compiler (nvcc), library and so on. CUDA program can be divided into CPU side (host) and GPU side (device) as shown in Figure 9.

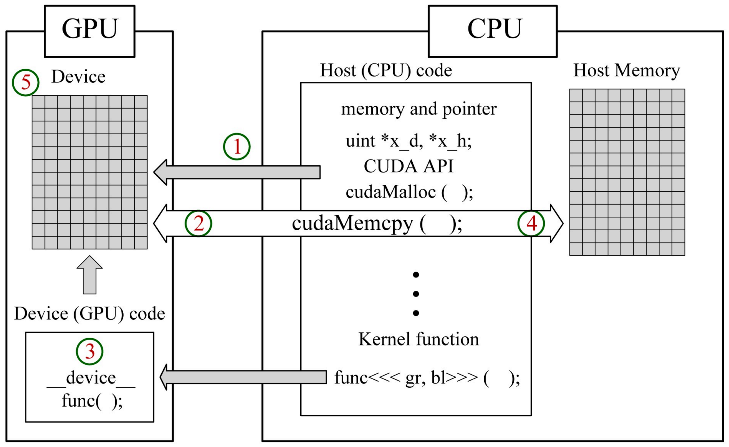

Figure 9: CUDA Programming.

The processing flow of the CUDA program is performed following five steps:

① Declare and reserve memory on the device side.

② Transfer data from the host side to the device side.

③ Call the kernel function from the host side and execute the kernel function on the device side.

④ Transfer the execution result on the device side to the host side.

⑤ The memory on the device side is released and the program is terminated.

The kernel function executed by the GPU is started on the host side.

4 GPU implementation of high speed hash function Keccak

This paper assumes the input of the hash function to be a plain text message which accords with a password for cracking. For an 8-character password, the input message length is equal to 64 bits. The length of the input message after padding should be fixed to 512 bits. As a result, 512 bits became one block, and it will be possible to remove the conditional branch to examine the number of remaining data blocks when executing the kernel function.

Table 3 shows the implementation environment.

Table 3: Implementation environment.

OS	Ubuntu 18.04.2 LTS
CPU	Intel Xeon E5-1620 v4 (3.50Ghz)
GPU	GeForce GTX 1080
CUDA Ver.	10.0
Compiler	gcc ver 7.3.0; nvcc ver 10.0 (CUDA)
Compiler Option	-O3

The guidelines for parallel high-speed implementation of hash functions in this research are summarized in Figure 10 and itemized as follows:

- For each password, hash processing is performed using one thread on the GPU.

- Multiple input messages (assuming passwords as plaintexts) are grouped into one array and copied to the memory on the GPU.
- Each thread performs a hashing process on the input message assigned to each thread in the GPU.

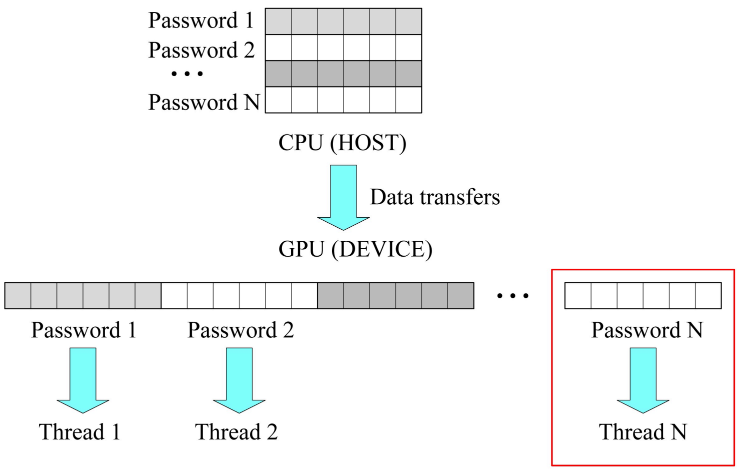

Figure 10: Parallels hashes for pasword cracking.

Although it is necessary to have a matching process of the given hashed value with the hashed value as the output of the GPU assuming brute force attacks, this research has not yet implemented this process. Below, our main three techniques to improve the performance of Keccak on GPU will be presented.

4.1 Reforming lookup tables

It is necessary to store constant values as the cyclic shift numbers as the form of 2-dimensional arrays in (4), at steps ρ and π. We prepared two tables #1 and #2 to store these arrays. $Index2X3YM5[y][x]$ is pre-caculation table as #1 to stores the calculation results of $(2x + 3y)$ *mod* 5. The second table #2 stores round offsets $r[y][x]$. In this case, a traditional implementation needs two pre-calculation tables, each of which has 25 elements (5x5 for 2-dimensional arrays). By reforming these steps with 1-dimensional array, we can reduce these tables' size to be 24 elements, which fact leads the decrease of the number of register variable.

The previous work [3] used the normal method with normal program using (4), that can be written as shown in Program 1.

Program 1: Source code for program using previous method.

```
for(y=0; y<5; y++){
  for(x=0; x<5; x++){
    B(y, Index2X3YM5[y][x]) = ROTL(A(x,y), r[y][x]);
  }
}
```

Here, $ROTL(A(i), r)$ is a cyclic right shift operation, modifying the position of $A[i]$ to $A[i + r]$. Tables #1 and #2 can be set to 1-dimensional arrays $index[i]$ and $r[i]$ by reforming as shown in Program 2.

Program 2: Source code for program using proposed method.

```
temp = state[1];
for (i = 0; i < 24; i++) {
  j = index[i];
  C[0] = A[j];
  A[j] = ROTL(temp, r[i]);
  temp = C[0];
}
```

The computational performance using this technique can roughly be increased about 1.1 times as shown in Table 4. Here, the number of input message is 262,144, which matches with the total number of threads. The unit of throughput is GigaBytes per second (GB/s). Normal program uses 2-dimensional arrays with 25 elements and the proposed method uses 1-dimensional arrays each of which consists of 24 elements. We can confirm the access speed when using a 1-dimensional arrays to be 1.1 times faster than 2-dimensional arrays.

Table 4: Comparison of throughputs.

Number of blocks	Number of threads per block	Previous method [GB/s]	Proposed method [GB/s]
512	512	24.109	28.209
1,024	256	32.837	38.197
2,048	128	33.104	38.545
4,096	64	32.601	37.884
8,192	32	32.923	38.041
16,384	16	31.084	37.650

It is possible to think that the difference of arrays became the main reason for this speed up. Here, let's compare the access process to a 1-dimensional array $A[M*N]$, and a 2-dimensional array $B[M][N]$, both consist of $M*N$ elements. In order to access to the $[i][j]$-th element in $B[M][N]$, it is necessary to calculate $i*N+j$ in advance, and this calculation takes a finite time. By changing to 1-dimensional array in our implementation, this calculation time can be eliminated, so that our implementation could speed up throughput around 1.1 times.

4.2 Memory allocation of constant values

Our GPGPU implementation has three pre-calculated tables to store constant values. The previous section explained two tables used at step ρ and step π for index numbers and round offsets. The third table #3 stores the round constant values for step ι.

GPU has a global memory that can be accessed from all streaming processors, but its access time is slow (see in section 3.2). In order to speed up our implementation, we propose an adequate usage of constant memory and shared memory in this paper. A constant memory in threads should be a place for common variables, and high access time is possible. However, as the total number of threads increases, its access time decreases because of access collisions. So adequate usage of both constant memory and shared memory leads to a high computational performance.

The first table #1 stores the index numbers $i = (y, 2x+3y)$. The second table #2 stores the round offsets $r[i] = r(x, y)$. And the last table #3 stores the round constants $RC[i]$. By allocating these tables #1 to #3 to three types of memory on GPU device such as global memory, constant memory, and shared memory, it is possible to clarify which allocation has the highest computational performance as their throughputs. All cases are considered here, while previous work [3] stored all of these constant values in shared memory.

When hashing 512 blocks with 512 threads per block, the obtained results are summarized in Table 5. Here, G, C, and S indicate memory types out of global memory, constant memory, and shared memory.

Table 5: Throughput changed by memory type of tables.

Memory type			Throughput
Table #1	Table #2	Table #3	[GB/s]
G	G	G	3.514
C	C	C	41.203
C	C	S	43.056
C	S	C	35.201
C	S	S	35.157
S	C	C	3.511
S	C	S	3.474
S	S	C	3.463
S	S	S	3.491

As shown in Table 5, cases when table #1 is stored in global memory or shared memory, their throughputs became about one tenth of cases when table #1 is stored in constant memory. And the throughput became better if tables #1 and #2 are stored in constant memory. And furthermore, the maximum throughput could be obtained when table #3 is stored in shared memory. This result is obtained because there is a relationship of their access speeds depending on the access method to each memory. Usually, access to constant memory is faster than global memory and slower than shared memory. However, when a huge number of threads access to shared memory at the same time, and access to a lot of different values in random addresses in shared memory, it is almost as slow as global memory.

Here, tables #1, #2, and #3 have same 24 elements. SHA3-512 uses Keccak-f[1600] function with 24 rounds of processing. In each round, all elements of tables #1 and #2 are accessed once in random, but only one element of table #3 is accessed. That means 24 accesses per round for table #1 and table #2 are random accesses. Table #3 is accessed only 1 time per round.

It is fast to access a same address from a lot of threads to constant memory or shared memory in GPU program. However, when address of the access destinations are different, its access speed to shared memory becomes almost as slow as global memory. As our implementation's result, when tables #1 and #2 (random access) is stored in constant memory, we could obtain the better throughput. The cases when table #3 is stored in constant memory or shared memory are almost the same, but the maximum throughput of our implementation achieved up when table #3 is stored in shared memory. So, we decided that tables #1, and #2 should be stored in constant memory to achieve speed up for random access.

4.3 Occupancy rate and the optimal configuration of blocks-threads

Processing time of Kernel function running on the GPU depends on the grain size of threads. In order to find out the optimal grain size of threads, we measured the maximum throughput of Keccak on GPU by changing the number of input messages, blocks, and threads per block.

In our implementation, #blocks is the number of blocks per grid, that shows grid size. For example, the grid size is [128, 1, 1] when #blocks is equal to 128. And #threads is the number of threads per block, that shows block size. An implementation using CUDA environment can be evaluated with occupancy rate, that is the number of concurrent threads per Streaming Multiprocessor (SM). CUDA occupancy can be estimated by the numbers of threads per block, registers per thread, and shared memory per block with CUDA occupancy calculator [18]. Since each thread hashes 1 message, the number of register per thread, and shared memory per block are not changed in our implementation. As shown in Table 3, accelerator GeForce GTX 1080 is used in our implementation with CUDA 8.0 environment. That means the computing capability version is 6.1, and maximum registers per thread is 255, maximum threads block size is 1024. Our Kernel function uses 76 registers per thread, and 192 Bytes of shared memory per block. According to that registers per thread usage, the occupancy rate is not high. Table 6 shows the occupancy rate change depending on the block size (#threads).

Table 6: Relation of block size and occupancy rate.

Threads Per Block	8	16	32	64	128	256	512
Occupancy rate	37.5%	37.5%	37.5%	37.5%	37.5%	37.5%	25%

Note that for each GPU computing capability, the CUDA occupancy depends on threads per block, registers per thread, and shared memory per block. It is not depending on the number of blocks (grid size), but the throughput is also changed when the number of blocks is changed, even with the same occupancy rate actually. And 100% occupancy is not needed to reach maximum performance.

As shown in Table 6, the occupancy is almost the same when the number of threads per block is set to 8, 16, 32,64, 128, and 256. All of these cases reached 37.5%, that is better than the case when we use 512 threads per block. And the results in Table 6 show that the maximum throughput is obtained with the same 37.5% of occupancy, but the number of threads per block are different.

Processing time of Kernel function running on the GPU depends on the grain size of threads. In order to find out the optimal grain size of threads, we measured the maximum throughput of Keccak on GPU by changing the number of input messages, blocks, and threads per block. The measurement results are summarized in Table 7.

Table 7: Relation of blocks, threads per block to maximum throughput.

Number of input messages	Number of blocks	Number of threads per block	Maximum throughput [GB/s]
256	8	32	2.346
512	16	32	4.591
1,024	16	64	8.936
2,048	16	128	16.783
4,096	256	16	24.097
8,192	512	16	31.558
16,384	1,024	16	40.221
32,768	1,024	32	44.526
65,536	1,024	64	48.562
131,072	1,024	128	50.428
262,144	2,048	128	51.477
524,288	4,096	128	59.931
1,048,576	8,192	128	60.357
2,097,152	16,384	128	60.586

The optimal configuration of blocks and thread's number depended on the number of input messages. However, almost cases of 128 threads per block gave high throughput. This result depends on GPU devices. Our implementation used GeForce GTX 1080, that has 20 Multiprocessors (MP) with 128 CUDA Cores per MP, so high throughput obtained when use 128 threads per block. As a result, the maximum throughput of this implementation achieved up to 20.5 GB/s.

4.4 CUDA Streams and Overlapping

A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code. A stream encapsulates these operations, maintains their ordering, permits operations to be queued in the stream to be executed after all preceding operations, and allows for querying the status of queued operations [13]. Device operations are host-device data transfer, kernel launches, and most other commands that are issued by the host but handled by the device. As default stream, all of these operations in CUDA run in a stream, also called the ”null stream” as no stream is specified. An operation in the default stream will not begin until all previously issued operations in any stream on the device have completed same as sequential processing. By using multiple streams to launch multiple simulltancous kernels, we can implement grid level concurrency. This non-default streams in CUDA are declared, created, destroyed, and launched in host code as the following Program 3.

Program 3: Source code for non-default streams [13].

```
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1);
result = cudaStreamDestroy(stream1);
.....
kernel<<<grid, block, sharedMemsize, stream>>>(argument list);
```

A timeline for the execution of default and non-default stream in our implementation are shown in the following Figure 11 and Figure 12. Here, ”make pwd” is making passwords process that executed on host (CPU). The data transfer from host to device (H2D) is a process of copying these generated passwords from the CPU to the GPU. By executing kernel on GPU, all passwords are hashed. These results are returned to the CPU side by data transfer from the device to the host (D2H).

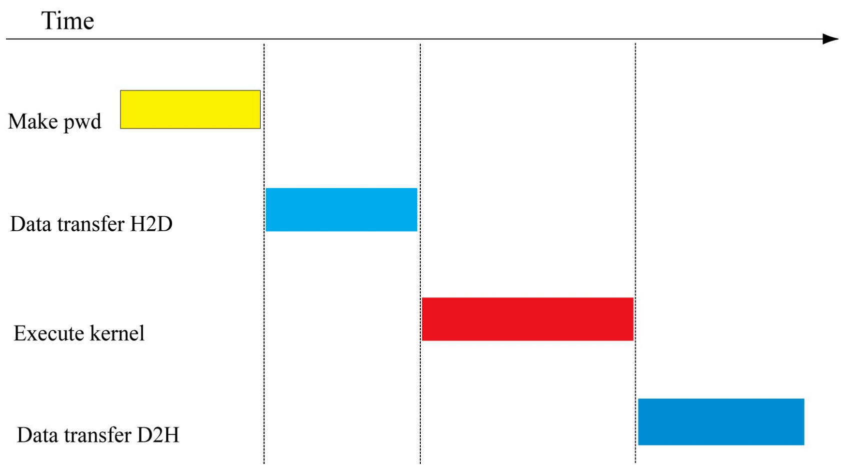

Figure 11: Default stream.

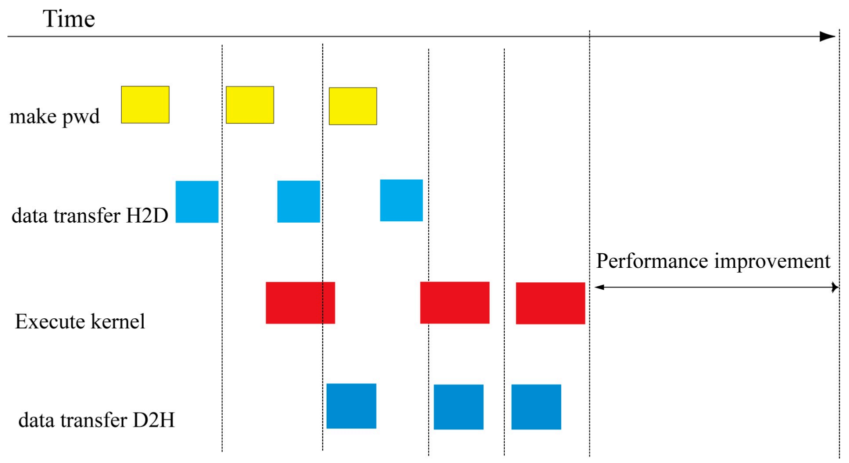

Figure 12: Execution time line using 3 streams.

It is neccessary to consider overhead attributable to the host-device data transfer to exploit effective performance. To hide this overhead, we can use overlapping data transfer and processing. In this implementation, we break up the total number of input messages N into chunks of streamSize elements. The number of (non-default) streams used is nStreams=N/streamSize. With nStreams=3, the measurement results are summarized in Table 8. The throughput improved by 1.07 times by using CUDA streams with Overlapping.

Table 8: Improved throughput by using CUDA streams with Overlapping.

Number of message	Default stream [GB/s]	3 streams [GB/s]	Rate
8,192	31.558	33.830	1.072
16,384	40.221	43.037	1.070
32,768	44.526	47.687	1.071
65,536	48.562	51.864	1.068
131,072	50.428	53.908	1.069
262,144	51.477	55.131	1.071
524,288	59.931	64.066	1.069
1,048,576	60.357	64.582	1.070

4.5 Related Work

Previous work [3] presented an implementation of the Keccak hash function family, that shown batch mode (normal hashing mode) and tree mode as the two way approach. Hashing more than one document at once in a batch mode, and hashing a single document in the tree mode. Due to time reasons, this previous work did not include the batch mode to their implementation [3]. Table 9 shows the experimental results of [3] for Keccak-f[1600] in tree mode on GTX 295, for different tree heights H.

Table 9: Throughput of [3] in tree mode.

File size [bytes]	H=0 [GB/s]	H =1 [GB/s]	H=2 [GB/s]	H=3 [GB/s]	H=4 [GB/s]
1,050,112	0.0025	0.0101	0.0525	0.0750	0.0553
10,500,096	0.0026	0.0106	0.0729	0.1522	0.1667
25,200,000	0.0026	0.0106	0.0759	0.1669	0.1953
50,400,000	0.0026	0.0106	0.0769	0.1732	0.2533

Moreover, the source code [14] shown that the authors of [3] used the previous method as mentioned in session 4.1, and stored all constant values in shared memory, that is one of the cases examined.

Previous work [6] presented an implementation of Keccak on GPU with tree structures, and their maximum throughput reached 1.183 GB/s on GeForce GTS 250 graphics card, better than the results of [3]. We executed the released source code [15] of [6] in our environment, and its measured throughput was 4.6 GB/s. Referred from techPowerUp GPU Database [16], the relative performance of GeForce GTS 250 is 42% of GeForce GTX 295, that used in [3], and the performance of GeForce GTX 1080 is 1020 % of GeForce GTS 250, based on TPU review data: "Performance Summary".

In fact, when we use the same released source code of the previous work, and executed it with our device-GTX 1080. The obtained throughput using GeForce GTX 1080 became 4.6 GB/s, it becomes only 3.9 times faster than GeForce GTS 250 actually.

The maximum throughput of this implementation achieved up to 64.58 GB/s on GeForce GTX 1080. Comparing with the throughput of the source code from the previous work [6] on our environment that shown in table 3, our implementation is about 14.0 times faster for password cracking. The previous work [6] targeted Keccak with tree structures, so it cannot be compared completely and fairly. This comparison is for reference only.

5 Approach to password cracking

Passwords are stored through hash functions using various methods in the password management system. Typical hash function can hashes all arbitrary length of input messages. However, length of passwords as input messages are only about a few characters actually. In this paper, in order to implement a high speed hash function for password cracking, we propose a special program developed for passwords up to 71 characters. In addition, the multiple times hash, as 2 times and 3 times hash as a simple example, against password cracking will be discussed.

5.1 Application to password cracking

In a typical implementation of Keccak, arbitrary length of input message can be its input. So that, it is necessary to process hashing for every block with 576 bits after padding. Here, we discuss the password cracking using GPU, so that the throughput of hash processing for every blocks with 576 bits, which accords with 71 characters, will be discussed.

Table 10 shows the measurement results of the throughput of the typical implementation of Keccak on GPU, the proposed method implemented on GPU, and the rate of performance improvement. The throughput of the proposed method improved by 7.49 times from the typical implementation. Here, because of password cracking, throughput is measured by Mega Hashed per second (MH/s).

Table 10: Improved throughput of program for password cracking.

Number of message	Typical implementation [MH/s]	Proposed method [MH/s]	Rate
8,192	94.26	476.49	5.06
16,384	107.25	606.15	5.65
32,768	113.55	671.65	5.92
65,536	113.56	730.48	6.43
131,072	113.69	759.27	6.68
262,144	113.67	776.50	6.83
524,288	127.15	902.34	7.10
1,048,576	121.42	909.60	7.49

5.2 Multiple times hash against password cracking

It is impossible to calculate the original input message from the hashed value. However, due to the high-speed implementation of the hashing process, password cracking can be done within a possible time by exhaustive search such as brute force attack, or rainbow tables. Especially short password is said to be very weak against this attacking method. One of the most important defending idea is the addition of "salt" after password in order to improve the security level of passwords. By adding different "salt", multiple users who use the same password will end up with a different hashed value. Another defending idea for the password management with hash function is applying the same algorithm by multiple times [17]. The resulting hashed value is simply resubmitted to the hash function as an input value in each time. A simple round function for hashing a password by multiple times could look like this: "*key = SHA3(password); key = SHA3(key) for n times* ". Actually, it is better to use both of the above ideas to protect user's information.

Multiple times hash is not suitable for many cases, but it is ineffective, and is being used in some fields. For example, in the process of creating Bitcoin Address [16], SHA-256 hashing is performed 3 times in total. Here, evaluation of implementation of high speed hash function Keccak, the multiple times hash against password cracking will be discussed as 2 and 3 times hash as a simple example. Copying input messages and padding are processed before 24 rounds of Keccak-f function as the initialization of internal state. In multiple hash program, new internal state can be set by previous hash result, and the initialization, padding are similar.

The throughput of 1 time, 2 times, and 3 times hash are measured with an assumption that the input message is equal to 262,144 passwords, which matches with the total number of threads. Table 11 shows the measured results.

Table 11: Throughput of 1 time, 2 times, and 3 times hash.

Block's number	Threads per block	1 time[MH/s]	2 times[MH/s]	3 times[MH/s]
512	512	584.590	336.319	243.181
1,024	256	725.023	365.359	260.196
2,048	128	719.553	366.273	252.329
4,096	64	718.710	365.285	250.417
8,192	32	717.652	364.148	247.447
16,384	16	425.648	252.591	183.941

Figure 13 shows maximum throughput of 1 time, 2 times, and 3 times hash changed by the number of input messages.

As results shown in Table 10 and Figure 13, the throughput of the hashing process didn't drop greatly due to multiple hash. Compared with the 1 time hash, the throughput of 2 times hash is about 50.9% slower, and the throughput of 3 times hash is about 34.8% slower as average. This results proves that using multiple hashes can increase the security level of password management with Keccak.

5.3 PBKDF2 and hashing password with a large number of iterations

PBKDF2 (Password-Based Key Derivation Function 2) is a key derivation function with a sliding computational cost, aimed to reduce the vulnerability of encrypted keys to brute force attacks. RFC 8018 [19], published in 2017, still recommends PBKDF2 for password hashing. In MySQL, the algorithm for password management looks like this: "*key = SHA1(SHA1(password, salt))* ". Here, it is only 2 times hashing while the iteration of hashing in PBKDF2 is about 1,000 to 10,000. In order to confirm the effects of this repetition of a huge number hashing, 1,000, 2,000, and 5,000 times hash are implemented. The throughput of 1time, 1000 times, 2000 times, and 5000 times hash are measured with an assumption that the input message is equal to 262,144 passwords, which matches with the total number of threads. Table 12 shows the measured results. The throughput of hashing passwords with a large number of iterations confirmed the effect was about 90%. That is the time

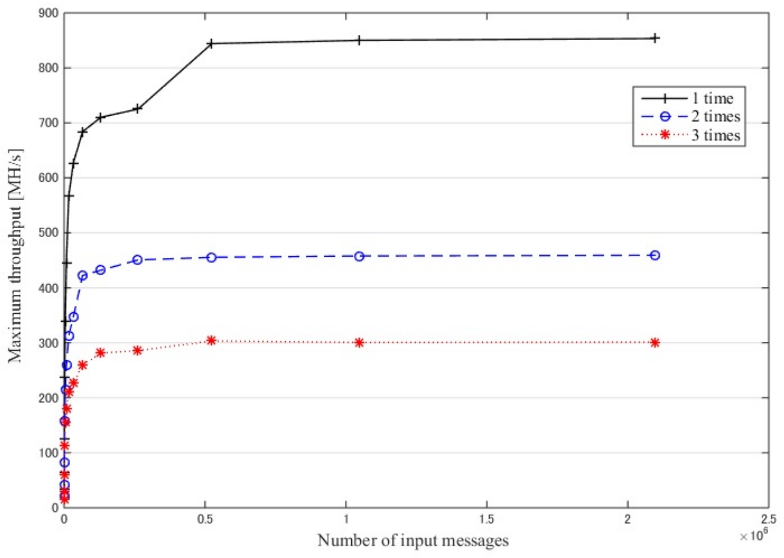

Figure 13: Comparison of maximum throughput of 1, 2, and 3 times hash.

required to hash one password for 1,000 times was almost the same as the total time to sequentially hash 900 passwords.

Table 12: Throughput of 1 time and hashing with large number of iterations.

Blocks	Threads	1 time[MH/s]	1,000 times[MH/s]	2,000 times[MH/s]	5,000 times[MH/s]
512	512	584.590	0.874480	0.435341	0.171747
1,024	256	725.023	0.853247	0.423948	0.166117
2,048	128	719.553	0.837387	0.414416	0.158116
4,096	64	718.710	0.836849	0.413256	0.155060
8,192	32	717.652	0.843148	0.415734	0.154444
16,384	16	425.648	0.445586	0.219873	0.084713

5.4 Advanced password recovery utility Hashcat

Hashcat [20] is known as the world's fastest and most advanced password recovery utility, supporting five unique modes of attack for over 200 highly-optimized hashing algorithms. Hashcat currently supports CPUs, GPUs, and other hardware accelerators on Linux, Windows, and macOS, and has facilities to help enable distributed password cracking.

The benchmark results of executing Hashcat's published program [21] in our research's environment are shown in the following Table 13. Here, the version of hashcat is 5.1.0, the unit of throughput is Mega Hashes per second (MH/s).

Table 13: The benchmark results of Hashcat.

Hash algorithm	Throughput (MH/s)
SHA3-512	770.6
Keccak-512	769.6

As the benchmark results, the processing speed of SHA3-512 and Keccak-512 is almost the same because the algorithms are similar. This research implemented a high speed hash function Keccak-512, but the results of throughput can be used for SHA3-512.

Table 14 shows the measurement results of throughput of Hashcat and the maximum throughput of our proposed implementation can be obtained.

Table 14: Comparison with Hashcat.

Implementation	Maximum Throughput (MH/s)	Rate
Hashcat 5.1.0	769.6	—
Proposed implementation	909.6	1.18

The throughput of Hashcat 5.1.0 that built and executed in the same device GTX 1080 is 769.6 MH/s. Our implementation using the same environment, shown in Table 3, is 1.18 times faster than the Hashcat results. Since this was a comparison with Hashcat, which is the purpose of password cracking, this result did not include transfer time, and was calculated only from the time of hashing processing on the GPU. In addition, the input message is also a fixed condition of 6 characters. However, the number of characters in the input message can be changed before compilation and does not significantly affect the processing time. Here, the reason why our results were faster than hashcat can be considered from the following two things. The first is the effect of our proposed method, and the second is the difference between CUDA and OpenCL platforms. CUDA is considered to be a more comfortable environment for the used device GTX1080.

Hashcat supports various hash modes as crack mode. For example, I has several modes with hash functions MD5, SHA1, SHA2 as raw hash (one time hash), salted and/or Iterated, andPBKDF2 for HMAC. However, multiple iteration hashing for Keccak is not yet supported by Hashcat, so we can make the comparison with one time hash only this time.

6 Conclusion

Implementation of high speed hash function Keccak using the integrated development environment CUDA for GPU on GTX1080 is presented.

In order to improve the performance of Keccak on GPU, this paper proposed the following three techniques: Improvement constant values at steps ρ and π of the algorithm; using constant memory, shared memory for constant values. We found out the optimal configuration of blocks-threads. With the proposed method at steps ρ and π, the throughput improved about 1.1 times. By using constant memory and shared memory, processing speed was further increased about more than 10 times. We also found that the optimal configuration of block-thread's number depended on the number of input messages. And the throughput improved by 1.07 times by using CUDA streams with Overlapping.

As our implementation results, the throughput up to maximum 64.58 GB/s was obtained on GeForce GTX 1080. The throughput of 1 time hash, 2, 3 times and 1,000, 2,000, 5,000 times hash are compared. The results prove that multiple times hash is possible to greatly improved the security level of Keccak with password management.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number JP15K00113.

References

[1] "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," August 2015, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[2] Lowden Jason, Lukowiak Marcin, and Alarcon Sonia Lopez, "Design and performance analysis of efficient KECCAK tree hashing on GPU architectures," Journal of Computer Security, v 23, n 5, p 541-562, November 2015.

[3] Cayrel Pierre-Louis, Hoffmann Gerhard, and Schneider Michael, "GPU implementation of the Keccak hash function family," JCommunications in Computer and Information Science, v 200 CCIS, p 33-42, 2011, Information Security and Assurance - International Conference, ISA 2011, Proceedings.

[4] Allan Mariano de Souza, Fabio Dacencio Pereira, and Edward David Moreno, "Exploiting Heterogeneous Systems: Keccak on OpenCL," The 2013 International Conference on Parallel and Distributed, Processing Techniques and Applications (PDPTA'13), 2013.

[5] Kahri Fatma, Mestiri Hassen, Bouallegue Belgacem, and Machhout Mohsen, "High speed FPGA implementation of cryptographic KECCAK hash function crypto-processor," Journal of Circuits, Systems and Computers, v 25, n 4, April 1, 2016.

[6] Guillaume Sevestre, Keccak Tree hashing on GPU, using Nvidia Cuda API, 2010, http://sites.google.com/site/keccaktreegpu/.

[7] G.Bertoni, J.Daemen, M.Peeters, and G.Van Assche, "The Sponge Functions Corner," http://sponge.noekeon.org/.

[8] Thai Duong, and Juliano Rizzo, "Flickr's API Signature Forgery Vulnerabilit," http://netifera.com/research/flickr_api_signature_forgery.pdf.

[9] G.Bertoni, J.Daemen, M.Peeters, and G.Van Assche, "Keccak sponge function family main document," 2010, http://keccak.noekeon.org/Keccak-main-2.1.pdf.

[10] G.Bertoni, J.Daemen, M.Peeters, and G.Van Assche,, "The Keccak sponge function family," updated 2016, http://keccak.noekeon.org/specs_summary.html.

[11] CUDA Zone, https://developer.nvidia.com/category/zone/cuda-zone/.

[12] G.Bertoni, J.Daemen, M.Peeters, and G.Van Assche, "The Keccak reference," January 2011, http://keccak.noekeon.org/Keccak-reference-3.0.pdf.

[13] NVIDIA Developer Blog, https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/.

[14] Cayrel Pierre-Louis, Hoffmann Gerhard, and Schneider Michael, "GPU implementation of the Keccak hash function family," JCommunications in Computer and Information Science, v 200 CCIS, p 33-42, 2011, Information Security and Assurance - International Conference, ISA 2011, Proceedings, http://www.cayrel.net/?Keccak-implementation-on-GPU.

[15] Guillaume Sevestre, Keccak Tree hashing on GPU, using Nvidia Cuda API, 2010, http://sites.google.com/site/keccaktreegpu/KeccakTreeGpu.zip.

[16] "GPU Database," https://www.techpowerup.com/gpudb/.

[17] "Impediments - The H Security: News and Features," http://www.h-online.com/security/features/Storing-passwords-in-uncrackable-form-1255576.html.

[18] "CUDA Occupancy Calculator - Nvidia", https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

[19] "PKCS #5: Password-Based Cryptography Specification Version 2.1", https://tools.ietf.org/html/rfc8018.

[20] "hashcat - advance password recovery", https://hashcat.net/hashcat/.

[21] "hashcat/hashcat: World's fastest and most advanced password recovery utility", https://github.com/hashcat/hashcat.