EECS598 Applied Parallel Programming With GPUs

Cryptohash Recovery

Liu, Xueshen
liuxs@umich.edu

He, Chenxi
hechenxi@umich.edu

December 7, 2021

Abstract

MD5 is a one-way hash function. To find the original password given its MD5 hash, the only method is to compute the encrypted MD5 code of all possible passwords and check if there is a match.

This project explores the GPU implementation of such a cryptohash recovery algorithm that finds a simple password given its MD5 hash. The algorithm first iterates through a dictionary of 10,000 commonly used passwords and their mutations, before trying to break the hash by brute force.

The GPU baseline of the algorithm was first implemented, before different optimizations were added to the algorithm: a better memory access pattern was designed for memory coalescing and bank conflict avoidance, streaming was used to reach a higher computing power utilization, and password encoding was used for balancing the loads in different kernel launches.

All implementations were tested on different test sets, each consisting of 5000 passwords generated in different ways. It is shown that the performance of the baseline GPU cryptohash recovery algorithm can be improved by 2.4 times to 3.8 times, depending on the portion of meaningful passwords in a test set. The improvement is most significant when mutations of meaningful passwords dominate the test set.

1 Background and Motivation

Computer security has been a heated topic ever since the introduction of the first personal computers. One of the most common security features that every computer user encounters everyday is the shibboleth used to log into web services, which usually comes in the form of passwords and/or two-factor-authentications.

To verify a user, the web servers must store the username and the corresponding password of that user. However, it is never recommended to store the passwords in plaintext: when there is a security breach and the server is compromised, all passwords will be easily leaked. A better approach will be first encrypting the password with a cryptographic hash function (cryptohash), and then storing the encrypted password. One of the major feature of a cryptohash is that it is a one-way function, a function for which it is practically infeasible to invert or reverse the computation [1]. Thus, even if a hacker succeeds in hacking into the server, it is still impossible for that hacker to acquire the plaintext of the passwords. In this report, we will be focusing on MD5, a hash function that accepts an arbitrary long input and produces a 128-bit hash value [2]. Even though MD5 has been proven to be cryptographically broken [3], it is still widely used today. Without losing generosity, the approaches discussed in this report can be applied to other more secure cryptohashes such as SHA-256 [4] or Whirlpool [5]. More sophisticated password protection includes salting the password before hashing [6], so that even if two passwords are the same, their stored hash remains different. Salting is beyond the scope of this report.

Due to the one-way property of cryptohash, the only method of cracking an encrypted password is to iterate through all possible passwords, and compare their hash values with the encrypted value. Historically, tools that are used to perform this task executes sequentially and are CPU-based. The performance of such tools degrades dramatically as the search space gets larger and the complexity of the password gets higher. In this project, we will be exploring the possibility of implementing a cryptohash recovery tool that runs on GPUs. By utilizing the parallel computing capability of GPUs, the tool is expected to have a significant performance boost compared to its CPU-based counterpart.

2 Baseline Algorithm

2.1 MD5 Cryptohash

The MD5 cryptohash algorithm accepts input messages of arbitrary lengths and works on chunks of 512-bit blocks (sixteen 32-bit words) iteratively. Thus, the first step of MD5 algorithm is padding the message so that its length is divisible by 512. The padding works as follows: first, a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, modulo 2^{64} [2].

After padding, the MD5 algorithm maintains a 128-bit state, divided into four 32-bit words. The states are initialized to a fixed constant, before being modified by each 512-bit chunk of the padded message. The processing of a message block consists of four similar stages, termed rounds; each round is composed of 16 similar operations based on a non-linear function F, modular addition, and left rotation, as shown in Figure 1.

In Figure 1, F is a nonlinear function; one function is used in each round. M_i denotes a 32-bit block of the message input, and K_i denotes a 32-bit constant, different for each operation. $<<<_s$ denotes a left bit rotation by s places; s varies for each operation. $\boxplus$ denotes addition modulo 2^{32} [2].

For simplicity, this report reuses the MD5 implementation provided.

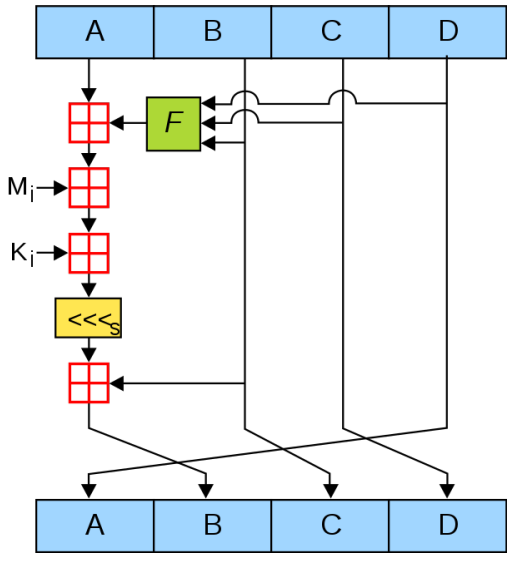

Figure 1: One MD5 operation. MD5 consists of 64 of these operations, grouped in four rounds of 16 operations.

2.2 Single Thread Implementation

Due to the one-way property of MD5, the only method of cracking an encrypted password is to iterate through all possible passwords, and compare their hash values with the encrypted value. However, if we consider all possible characters in a password, the search space is gigantic and grows exponentially as the length of the password grows. Furthermore, it is more likely for a user's password to be "password123", rather than a completely random "F6cW,S-M.ekta>;mR6t2X-lxsnb". Thus, the high-level guideline of the algorithm will be first trying on some common passwords and their mutations, before trying to brute force the answer.

2.2.1 Basic Dictionary Attack

First, we iterate through a dictionary of 10,000 passwords that are meaningful and commonly used by users. Their MD5 cryptohash values are calculated and compared to the hash value provided. Since 90% of the passwords used today can be found from this 10,000-password dictionary, we have a very high chance of finding a match in this first step.

2.2.2 Advanced Dictionary Attack

For someone who takes cyber security more seriously but doesn't want to memorize a completely random password, he or she might use a password that is a mutation of a commonly used human-readable password. For example, instead of using "password" and "matthew", "password123" and "mattheW" are used. Thus, we iterate through 252 mutation methods and calculate the MD5 cryptohash values of the mutated 10,000-password dictionary. Similarly, we compare the cryptohash values to the hash value provided. The mutation methods include making the last letter uppercase and adding various number sequences to the end of the password. It is expected that 95% of all the passwords used can be found by the above two methods.

2.2.3 Brute Force Attack

For the rest 5% of the password, there is no other better option than breaking them using brute force. For this part, the search space is limited to passwords that consist of only lower case letters, and are no longer than 6 letters. These limitations are due to the fact that the search space grows exponentially as the length of the password increases. The algorithm iterates through all possible combinations in the search space, and compare their MD5 cryptohash values to the hash value provided.

2.3 GPU Adaption

The computation speed of the above algorithm can be greatly increased by adopting parallelism.

2.3.1 Parallel Dictionary Attack

In a paralleled dictionary attack, different mutation methods (no-mutation is also counted as one mutation method) are still executed in series. For each mutation iteration, the MD5 hashes of all mutated passwords are computed and compared with the target hash in parallel. Each thread of each block will be in charge of computing and comparing the MD5 of only one password.

2.3.2 Parallel Brute Force Attack

In each iteration, the passwords of a certain lengths are inspected. The length ranges from 1 to 6, and each iteration are executed in series. Similar to the paralleled dictionary attack, the MD5 hashes of all combinations of letters in each iteration are computed and compared with the target hash in parallel. Similarly, each thread will be in charge of computing and comparing the MD5 of only one password. This limits the maximum length of the password to be tested, since there will not be enough blocks for all passwords with length 7.

2.3.3 Termination

If a match has already been found, then it is meaningless to go through the rest of the search space. Each thread in each block calculates and compares one mutated password or combination. If there is a match, the corresponding thread atomically adds to an integer indicator in GPU which is initialized to 0, and prints out the original password. After each iteration, the indicator is checked. The search will be terminated early when the indicator is not 0.

3 Optimizations

Even though the GPU version of the algorithm already runs faster compared to the CPU version, it can still be optimized in terms of better memory access pattern, higher parallelism and more efficient thread utilization.

3.1 Better Memory Access Pattern in Dictionary Attack

The first optimization focuses on the dictionary attack part. In the original version, each thread copies one password from the global memory, mutates the password and then computes its MD5 value. Since each password is 64 Bytes long, the accesses to the global memory are not coalesced, which might slow down the entire process.

Thus, optimization 1 coalesces this access to global memory. More specifically, each block has its own shared memory that stores all the passwords used by all the threads in the block. To load passwords from the global memory to the shared memory, each thread loads a 4-byte-word, and consecutive threads load consecutive words in the global memory. This process is executed $64/4 = 16$ times to load all the data needed.

However, if passwords are stored in the shared memory consecutively, there will be bank conflict during the access of the shared memory, since each password is 16 words long. Thus, there will be a 4-byte-word padding between each two adjacent passwords in the shared memory. Because the maximum shared memory size per block is 49152 Bytes, and each thread will use up $16 \times 4 + 1 \times 4 = 68$ Bytes of shared memory, we set the number of threads per block as 512.

An illustration of this optimization is shown in Figure 2.

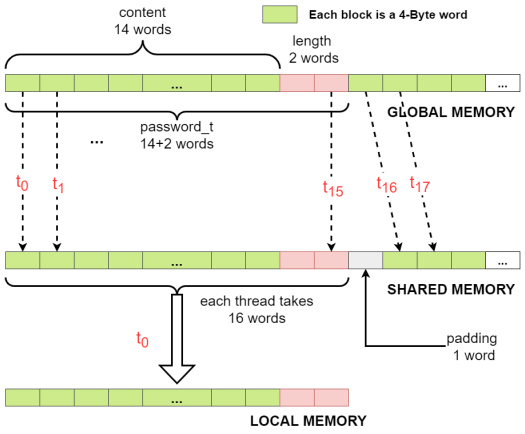

Figure 2: Optimization 1.

3.2 Higher Parallelism in Dictionary Attack

Even though optimization 1 optimizes the memory access pattern in the dictionary attack kernel, all dictionary attack kernels are still launched sequentially after the password data are completely copied from the CPU to the GPUs. Since the dictionary is comparatively large, this process is quite time consuming, and all GPUs are completely idle when passwords are being copied. Massive computing power is wasted in this process. Thus, optimization 2

adopts streaming to utilize the wasted computing power by paralleling memory copying and computing.

More specifically, the kernel in each stream has 253 blocks, with each block of the kernel calculating one mutation of 512 passwords. Thus, there will be $\lceil 10000 \ / \ 512 \rceil = 20$ streams, as shown in Figure 3. Each stream first asynchronously copies the passwords it needs into a chunk of global memory, then launches the kernel, before being synchronized. If a match is already found in one stream, all other streams that have not been synchronized will be destroyed.

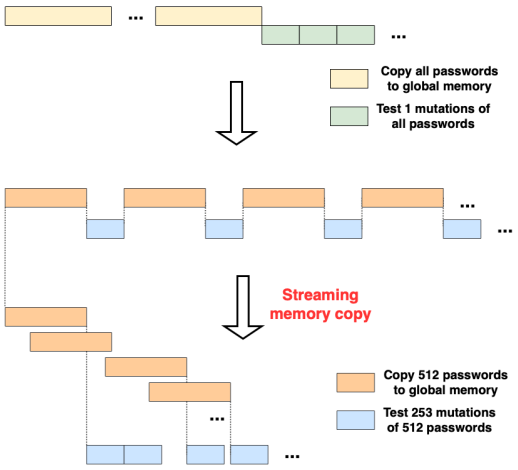

Figure 3: Optimization 2.

3.3 More Efficient Threads Utilization in Brute Force Attack

Previous two optimizations focus on the dictionary attack. The brute force attack can also be optimized. The original algorithm launches one kernel per password length sequentially. Since the search space grows exponentially as the password length grows, computing power utilization is low when the length is small, and too much computing power is needed when the length is large. This causes imbalance between different kernel launches, as shown in figure 4. Further more, the maximum number of blocks that can be launched in a kernel limits the maximum password length.

In this optimization, each combination of password is sequentially encoded by an integer in alphabetical order, starting with "a" encoded as 0, "b" encoded as 1, "z" encoded as 25 and "aa" encoded as 26. The encoding was done in such a way that decoding a password from an integer will not cause control divergence within a warp, as shown in Algorithm 1.

In each iteration, 8192 blocks of 1024 threads are launched. Each threads finds the encoded integer it is in charge of, then decodes the integer

Algorithm 1: Non-control Divergence Password Decoding Algorithm

```
Initialize ctn to be true.
Initialize pwd to be an empty password.
Initialize index to be the global thread id.
for i in [0, MAX_LEN) do
    pwd.word[i] = ctn·'a' + index%26
    pwd.length = pwd.length + ctn
    index = index/26
    ctn = (index != 0)
    index = index - ctn
end for
```

to find the password. Similar to the baseline dictionary attack, the result is checked after each iteration. There is no need to adopt streaming here, since there is no memory access in brute force attack.

In this way, theoretically there will be no limit to the length of the password to be tested, as long as the total number of combinations fit into a 64-bit integer. Moreover, computing power is more balanced and better utilized when testing passwords of smaller lengths.

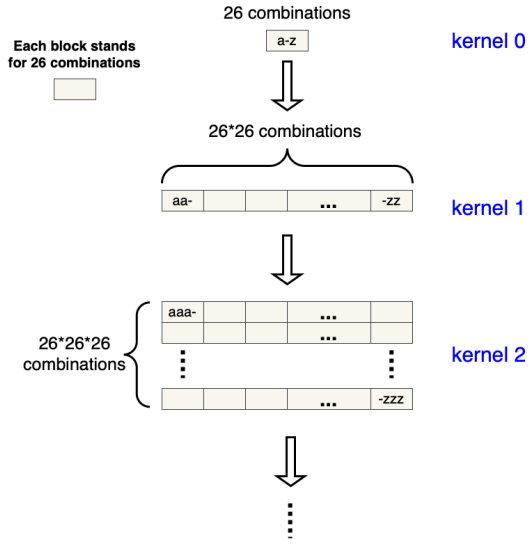

Figure 4: Imbalance in the baseline brute force attack.

4 Evaluation Methodology

The GPU algorithm will be run on a testing data set consisting of the MD5 hash of 5000 different passwords. The passwords should all fall into the search space, i.e., should be successfully recoverable by the baseline algorithm. The total computation time will be compared between different optimizations and the baseline.

Since different passwords will be recovered in different phases (dictionary attack and brute

Device Property	Device Configuration
Maximum global memory size, in Byte	16945512448
Maximum constant memory size, in Byte	65536
Maximum shared memory size per block, in Byte	49152
Maximum block dimensions	1024 x 1024 x 64
Maximum grid dimensions	2147483647 x 65535 x 65535
Warp size	32

Table 1: Great Lakes GPU configuration.

force attack), and different optimization methods focuses on different phases, constructing an extensive test set will be tricky. What makes it even trickier is that it is hard to make the test set reflect real-word situation, since different groups of people have different habits of setting their passwords.

Thus, on the premise that people prefer to use a meaningful password rather than a completely random password, multiple test sets are constructed. Different test sets have different combinations of mutated dictionary passwords and completely random passwords. The percentage of mutated dictionary passwords in the test set ranges from 70% to 100%. The mutation method and the dictionary password to be mutated from will all be picked at random evenly. The lengths of random combination passwords are evenly chosen from 4, 5 and 6. After the length is chosen, each alphabet of the random password will be evenly chosen from 'a' to 'z'.

These data set will be tested on The Great Lakes Slurm cluster. The configuration of the GPUs used in Great Lakes is shown in Table 1. Some aspects of the configuration, especially the maximum shared memory size per block and the maximum grid dimension, pose some limitations to the program. Some parameters of the program have been tailored so that the resources of the GPU can be maximally utilized.

5 Speed-up Analysis

The experimental results are shown in Figure 5 , Figure 6, and Figure 7.

Figure 5 shows the average time used by the baseline algorithm and each version of the optimized algorithm to recovery the passwords in each test set. Each experiment is conducted several times, and the average is taken to minimize uncertainty. Figure 6 illustrates the improvement of different optimizations compared to the baseline algorithm, and Figure 7 shows the improvement of different optimizations compared to their previous version.

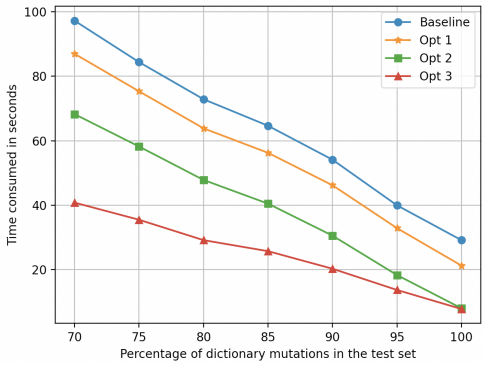

Figure 5: Average time consumed by the baseline and different optimization versions on different test cases.

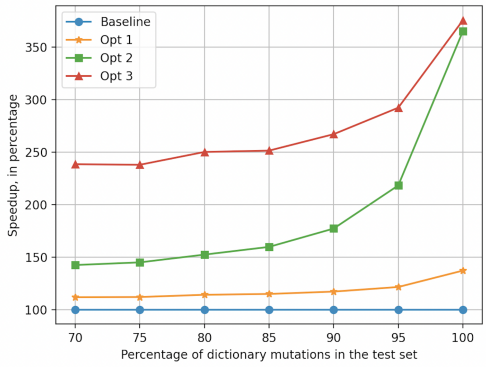

Figure 6: Improvement of different optimization versions compared to the baseline, tested on different test cases.

5.1 Analysis of Optimizations Combined

As shown in Figure 6, the optimized algorithm out-performs the baseline on every test set. The performance boost is the greatest when the test set includes only mutations of dictionary passwords: the running speed of the optimized algorithm is 375.62% of the baseline algorithm. The performance improvement drops as more and more passwords of random permutations are mixed into the test set, but the optimized version is still faster than the baseline by a large margin: the running speed of the optimized algo-

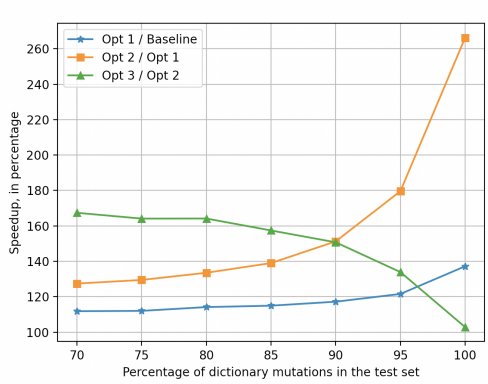

Figure 7: Improvement of different optimization versions compared to its previous version, tested on different test cases.

rithm is 238.52% of the baseline algorithm when 70% of the tested passwords are random permutation.

5.2 Analysis of Individual Optimization

Figure 7, which compares individual optimizations, shows some interesting results.

Optimization 3 focuses on improving the performance of brute force search on completely random passwords, and it is proven by the green "Opt3 / Opt2" curve. Optimization 3 contributes the greatest to the improved performance when the percentage of random permutation passwords is higher than 10%. The performance improvement drops down to 0 when the test set includes dictionary mutations only.

The orange "Opt2 / Opt1" and the blue "Opt1 / Baseline" curves compare the improvement of memory access coalescing and streaming. As illustrated, even though a better memory access pattern will indeed increase efficiency, a bigger improvement comes from utilizing wasted computing power when copying data.

6 Conclusion

By deploying the provided CPU algorithm of MD5 cryptohash recovery onto GPU, it is proved that GPU can be utilized for more efficient cryptohash recovery.

Furthermore, by designing a better memory access pattern, adopting streaming for higher computing power utilization and balancing the loads of kernels, we managed to improved the performance of GPU cryptohash recovery program by 2.4 times to 3.8 times, depending on the portion of meaningful passwords in the test set.

Compared with the baseline cryptohash recovery program, the improvement of our implementation is most significant when mutations of meaningful passwords dominate the test set. The higher level takeaway of this observation is that memory access takes up a significant amount of time in dictionary attack, making it crucial to increase parallelism and decrease memory access time.

Contribution

- Chenxi He was in charge of implementing the baseline algorithm, generating different test sets, conducting tests on all the codes and writing the report.
- Xueshen Liu was in charge of designing and implementing the three optimizations. He also helped editing the report by providing the pseudo codes and illustrations.

References

[1] Shai Halevi and Hugo Krawczyk. Randomized hashing and digital signatures. https://www.ee.technion.ac.il/~hugo/rhash/.

[2] Ron Rivest. The md5 message-digest algorithm. *IETF*, page 5, 1992.

[3] Md5 vulnerable to collision attacks, vulnerability note vu836068. https://www.kb.cert.org/vuls/id/836068, 2009.

[4] U.S. Department of Commerce, National Institute of Standards, and Technology. *Secure Hash Standard - SHS: Federal Information Processing Standards Publication 180-4*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2012.

[5] Paulo S. L. M. Barreto. The whirlpool hash function. https://web.archive.org/web/20171129084214/http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html, 2008.

[6] Salted password hashing - doing it right. https://crackstation.net/hashing-security.htm, 2021.

Appendix

For the source codes of this project, please refer to https://github.com/danielhe2000/EECS598_Final_Project.

"baseline" folder includes the provided CPU codes. "cuda_v1" folder includes the GPU baseline, and "cuda_v2", "cuda_v3", "cuda_v4"

are the codes corresponding to optimization 1, 2 and 3, respectively. "cuda_v5" and "cuda_v6" are some experimental codes that are not adopted. All folders ending in "test" are used for running the test cases.

To run the codes,

1. Build the executable code:

   ```
   make
   ```

2. Run the cryptohash kernel:

   ```
   sbatch run_test [32-character hash]
   ```

You can find the 32-bit character of your desired string by:

1. Build the executable code:

   ```
   make wiki
   ```

2. Run the wiki kernel:

   ```
   ./wiki [string]
   ```

For detail, please refer to the readme file.