Fall 2022 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

MD5 CUDA Brute Force Analysis

Pratham Baid

January 9, 2023

Abstract

The MD5 hash function is commonly used to verify the integrity of data, such as the integrity of downloaded files or messages transmitted over a network. However, the MD5 encryption scheme has been found to be vulnerable to a number of attacks and is no longer considered secure. This project aims to highlight the vulnerabilities of the MD5 encryption scheme by using a GPU (Graphics Processing Unit) to brute force it. The project involves creating a program that takes an MD5 hash value as input and uses a GPU to quickly try every possible combination of characters until it finds a match. The program could then be used to demonstrate the weakness of the MD5 encryption scheme by showing how quickly it can be broken using a brute-force attack. Overall, this project highlights the significance of using stronger and more secure encryption methods to protect sensitive data.

Contents

1 Problem Statement **4**

2 Solution Description **5**

3 Overview of results. Demonstration of your project **6**

4 Deliverables **7**

5 Conclusions and Future Work **8**

References **9**

1 Problem Statement

I proceeded with the default project.

The MD5 algorithm is a widely used cryptographic hashing algorithm that is used to verify the integrity of data. You may have seen an MD5 hash provided with a large file download in order to confirm package validity. In the past, MD5 has seen usage in database password encryption due to its relatively simple and fast algorithm. Unfortunately, its susceptibility to brute force attacks is a major security concern and one of the reasons it is discouraged from being used by Cryptographers.

A brute force attack is a type of attack that attempts to exhaustively try all possible combinations of characters until the correct one is found. Due to the sheer number of possible combinations, a brute-force attack on an MD5 hash can be extremely time-consuming. However, with the availability of powerful parallel GPU (Graphics Processing Unit) systems, attackers can perform brute force attacks at much faster speeds. By utilizing a large number of threads available on a GPU, attackers can drastically reduce the amount of time needed to complete a brute-force attack.

In addition to the increased speed of brute force attacks, the MD5 algorithm has also been found to be susceptible to collisions. This means that two different inputs can produce the same hash, giving attackers the ability to generate valid hashes for a given input without actually knowing the input itself.

The combination of increased attack speed and susceptibility to collisions has made MD5 an increasingly insecure hashing algorithm. As such, many developers and organizations have started to move away from using MD5. Alternatives such as SHA-256, SHA-3, and BLAKE2 are considered more secure and are recommended for use instead.[Wik22]

I hope to demonstrate how effective the brute-force approach is by utilizing the maximum performance from modern GPU computing. We will analyze the duration to determine the real-world feasibility of this approach.

2 Solution Description

In my implementation of this brute force algorithm, I had to make some design decisions. When approaching this project I had several questions and choices.

1. How can I ensure no two threads ever attempt the same string?
2. How can I select only alphanumeric characters for my string?
3. How do I generate all permutations of an n-digit password?
4. Is there some way to avoid creating a string before proceeding with the MD5 algorithm?
5. How can I leverage shared memory to reduce overall computation?
6. What are the optimal block and grid dimensions?
7. Is it better to convert an integer into a string with conditionals or to load from shared memory?

My conclusions were:

1. By using thread id + iteration * block * grid dimensions for successive computations.
2. & 3 by creating a character set of valid characters and accessing them through indices.
4. I believe it is not possible since MD5 depends on the correct character values for its algorithmic calculations.
5. Load character set and the secret hash into shared memory since those are constant across all threads. Potentially keep a shared block-wide substring (depending on kernel dimensions).
6. I found that working with multiples of 32 and close to the length of the character set (in our case length = 62) resulted in the best performance, but this is dependent on the answers to the other questions answered previously.
7. I was unable to test this due to time constraints, but it is something that I would consider experimenting with in the future.

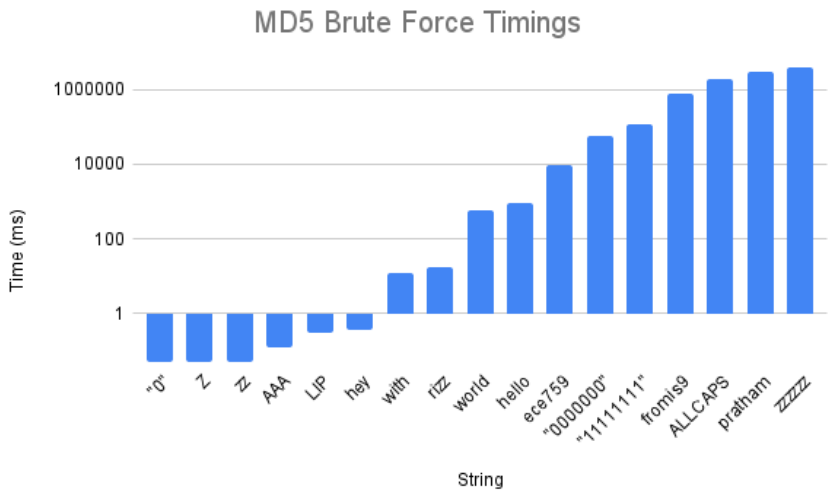
I implemented the MD5 algorithm twice. Once as a C++ library and second as a CUDA library. I didn't realize that using C++ classes in device code would cause the compiler to throw so many errors.

This resource [Riv92] provided much of the logic and implementation. It needed slight adjustments for C++ and CUDA 64-bit systems.

Click here for additional informal notes I took while approaching this project.

3 Overview of results. Demonstration of your project

Once I cleaned up all of the bugs, I ran the cracker for multiple sample hashes to test for correctness and get timing data. All data comes from the GTX 1080 provided by WACC on Euler. The y-axis of the graph is logarithmic to better represent the relationship between String and Time.



String	Time (ms)
0	0.052096
Z	0.049152
zz	0.050176
AAA	0.125952
LIP	0.301472
hey	0.3584
with	12.5133
rizz	17.4026
world	600.701
hello	868.04
ece759	9659.91
0000000	57478.9
1111111	119835
fromis9	740909
ALLCAPS	1.85e+06
pratham	2.83e+06
zzzzzzz	∼3.91e+06

Figure 1: Runtime table on GTX 1050 Ti.

Although I was unable to time test how long it takes to crack all 7-digit passwords (string 'zzzzzzz', due to Euler time limitation), with the knowledge it takes 1 minute to attempt all 6-digits password combinations we can estimate that it will take a little over 1 hour to attempt all permutations from 1 through 7 digits.

Considering that these hashes need to be calculated only once, if you save the generated hashes in a lookup table then this is a very viable crack, even for salted hash databases given you know the salt.

4 Deliverables

Alongside this write-up, important files in the repo are

1. cracker.cu – contains brute force thread implementation
2. cracker.cuh – contains brute force thread implementation
3. main.cu – runs cracker
4. makefile – compiles and runs the project
5. md5.cuh – MD5 algorithm CUDA implementation & thread id → string algorithm
6. md5.cpp – MD5 algorithm C++ implementation
7. md5.h – MD5 algorithm C++ implementation

There is only one executable produced from all of these files, which is *main* and it requires no additional data dependencies.

Compilation:

```
make compile
```

Run on local machine:

```
# To crack a specific hash
./main <your hash here>

# Or enter string to convert to hash to crack
./main
> string prompt: <your string here>

# Running predefined test hashes
make test
```

You can modify test.sh to change what gets run on Euler.

```
# Running predefined test hashes in test.sh on Euler
make test_euler
```

5 Conclusions and Future Work

I see a lot of room for optimization and improvements with my implementation.

In my implementation, I convert a thread's id into a string and then run the MD5 algorithm. If we pick our block dims correctly (blockDim = length of character set), we can share a block string and simply concatenate it with the thread id-specific character. I tried implementing this but I ran into a bug, so I reverted to my working algorithm.

Since this task scales very well horizontally, another improvement would be requesting multiple GPUs for a multi-device effort. No data would need to be exchanged between the devices, they would just need to know of each others' existence, basically outright multiplying the hash rate.

Another place of improvement can be with the conditionals. If I can reduce the if statements in the thread loop then I can reduce the accumulating warp divergence. Since there are millions of hashes being run, these seemingly small costs add up over time.

Finally, one issue with this project is that increasing the kernel dimensions causes the algorithm to get stuck. I am not sure why this occurs and would debug this issue in the future as plenty of computational power is wasted by not using all of the resources available on the device.

Some of the skills I acquired from the course material of ECE759 that were used in this project include shared memory, loading data in parallel, device synchronization, warp synchronization, and thread divergence. Not all of these were implemented, but they were considered in implementation decision-making.

One takeaway from this project is that when designing high-performance applications, one cannot rely on a theoretically better implementation for performance benefits. One of my number-to-string algorithm implementations was theoretically superior to the current implementation but it actually scaled relatively poorly. Another takeaway is that seemingly cheap operations can have a really big overhead cost attached to them. You need to consider all associated costs with an operation. For example, if I decided to do arithmetic offset to map an index to a valid character instead of a load instruction, the offset approach could be faster, but the overhead cost of the conditional statement required to pull off the offset could add unwanted divergence.

With strong advancements in GPU & ASIC computing, I believe many algorithms we currently think are unfeasible to brute force may actually be susceptible.

References

[Wik22] Wikipedia contributors. *MD5 — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-December-2022]. 2022.

[Riv92] R. Rivest. *The MD5 Message-Digest Algorithm*. [Online; accessed 14-December-2022]. April 1992.