Spring 2020 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

MD5 Hash Attack

Chia-Wei Chen, Dax Chen, Pei-Hsuan Wu

May 6, 2020

Abstract

In this project, we aim to do step-by-step speedup in the MD5 hash cracker application. We tried the concept learned from the ME759 range from serial CPU, OpenMP threading parallelism, MPI process parallelism to GPU CUDA parallelism. The result shows that we increase the throughput of checking MD5 hashing from 0.356 Mhash/s using brute-force CPU to 6.778 Ghash/s using multiple GPUs, which is nearly 19,000x speedup.

Link to Final Project `git` repo: https://github.com/wu0607/2020-Spring-ME759-FinalProject

Contents

1. General information

- Computer Science
- MS student
- Chia-Wei Chen
- Dax Chen
- Pei-Hsuan Wu
- I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party.

2. Problem statement

Exhaustive search attack on the MD5-crypt password hashing scheme using modern CPU and GPU in parallel (using CUDA). Measure and compare **throughput** of different cracking methods in **#hash/second**. Identify performance bottlenecks and bounds using tools such as **NVIDIA Visual Profiler**. Expose an interface for the user to input a hash value (such as through command line argument), and have the program crack the password that generates the hash.

3. Solution description

3.0 Algorithms

3.0.1 MD5 Hashing Algorithm

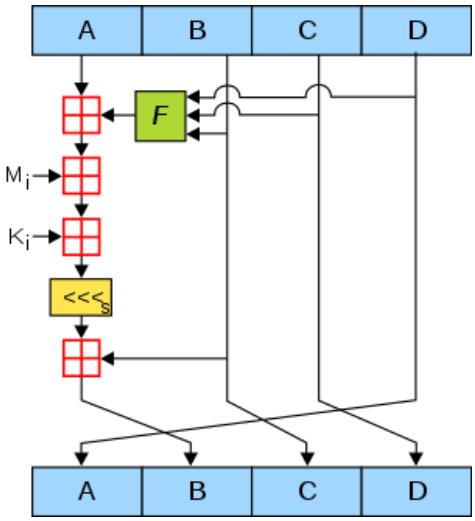


Figure 1: One MD5 operation.

We followed the Wikipedia MD5 page [1] with its pseudo-code. According to Wikipedia, MD5 algorithm takes a message of any length as input, and outputs a 128bit result. We summarize the high level MD5 algorithm and pseudo-code here:

1. Pad the input to be a multiple of 512 bits, because input is divided into **chunks of 512 bits**.

2. The MD5 operates on a **128-bit state**, so for each round we are essentially just modifying this 128-bit state with the 512-bit chunk. We separate the state into **4 32-bit parts**, named A, B, C, and D, as shown above in Figure [1].

3. For each round, a chunk of 512-bit message comes in to modify the 128-bit state. There are 4 different ways to modify the state, and there are 16 similar operations based on a nonlinear function F, modular addition, and left rotation. For simplicity we will skip the details here.

4. For each round, first **break the chunk into 16 32-bit words**, and go into the main loop for **64 iterations**. There are different ways to calculate the state based on the loop index, such as the first 16 iterations, it uses the first method, and the next 16 iterations, another method, so on and so forth.

5. After the main loop of 64 iterations, we **add the results to the state**, and process the next chunk.

Since our input message never exceeds 512 bits for cracking passwords, this algorithm **only runs for 1 round**.

3.0.2 Brute-Force Combination Generation Algorithm

For our brute-force attack, we need to generate all the possibilities of combinations for a character set. In our case, our character set consists of lowercase and uppercase alphabets and digits, so there are 62 characters in total.

Because we want to parallel the code including the text generation, we need to find a way that does not depend on other texts. The algorithm we used to generate password combinations is described as follows:

```
string ret;
while (index) {
    ret += alphabet[index % size];
    index /= size;
}
```

(`alphabet` is the character set; `size` is the size of the character set, which is `62` in our case here; `ret` is the resulting text input generated; and `index` is the sole thing needed to determine the generated text)

Notice with this method, we can exhaustively generate all possible combinations as **index goes from 1 to infinity**. Also notice with this method, **we only need to specify index to determine the input text**. This turns out to be important for parallelizing our attack.

3.1 CPU version

We start by implementing a basic CPU version of md5 hash algorithm in C++. After the basic algorithm finished, we added the **support for both brute-force attack** and **wordlist dictionary attack**. We use the famous **rockyou wordlist** to test our solution, and compared to brute-force attack, wordlist is

significantly faster when the target is in the dictionary. However, if the target password is not in the wordlist, it will never find it. We think later in the GPU version we want to compare the hash per second with the CPU version, so we only implement the dictionary attack here in CPU version, and **focus on enhancing the brute-force attack**.

3.1.1 Serial CPU

3.1.2 OpenMP

Since OpenMP doesn't allow explicitly **break** inside the for loop function, we use a flag - **find** for help. If the password has been cracked, set the shared variable **find** to **true.** At the very start of our for loop, **continue** if find is true.

#pragma omp for schedule([OPTION])
OpenMP schedule defines how iterations are split up, it decides the creation of **chunks**, and how chunks are mapped to threads.

We try all kinds of schedules in pure OpenMP mode and obtained following results:
static: 4.167 Mhash/sec
- Fix sized chunks (default size is about #iteration / #thread)
- Distributed in a round-robin fashion

dynamic: 4.219 Mhash/sec
- Fix sized chunks (default size is 1)
- Distributed one by one at runtime as chunks finish

guided: 4.197 Mhash/sec
- Start with large chunks, then exponentially decreasing size
- Distributed one by one at runtime as chunks finish

runtime: 4.439 Mhash/sec
- Controlled at runtime using control variable

auto: 4.417 Mhash/sec
- Compiler/Runtime can choose

In conclusion, auto & runtime performs the best, but we want a good and defined behavior, so we choose **dynamic** as our final schedule algorithm.

3.1.3 MPI

```
start = maxVal / size * rank
end = maxVal / size * (rank + 1)
for i in start ~ end:
      MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag)
      if find or flag:
            break
      // cracks out password
      if md5(customToString(i)) == hashcode:
            find = 1
            // MPI_Isend to all the other ranks except itself
            MPI_Isend(&find, 1, MPI_INT, r, tag, MPI_COMM_WORLD, &request)
```

Equally split workload(0~maxVal) for each rank to calculate candidate string from start to end, where start = maxVal / size * rank, end = maxVal / size * (rank + 1).

If a specific rank cracks out the password, it sends out a signal to all the other ranks, and after others receive the signal, they **continue** to bypass the for loop. To **improve throughput** and **avoid deadlock**, we use **non-blocking MPI functions to send/probe signals**.

3.2 GPU version

Next we moved on to implement the GPU version using CUDA.

In our GPU approach, we used the same method described in section 3.0.2 to generate input text for each iteration, so that **each kernel only needs to know the starting index and the ending index**, which helps to **decrease the memory usage** and **shift the problem to computation bound**.

Note that we implement a `next(int len, char* word, int increment)` that is used both in `__host__` and `__device__`, this function is for generating the word we want to crack for the given increment. We'll call this next() in three different part, (1) for each thread, after we calculate the global thread index, we can find out the starting word with this index (2) each thread will have its workload to crack, so we'll call next() with increment = 1 every time after a word is been checked (3) after launching the kernel function, host will call next() to increment the global index.

Below shows a series of optimizations we made, and the performance gain will be shown in section 4.

3.2.1 Basic CUDA

We implemented the basic CUDA version using the same approach we used in the CPU version. We deliberately implement this version without using shared memory and unrolling loops because we want to see how much performance gain we can get. This version serves as the baseline for the GPU version.

3.2.2 Shared Memory

Since we use char array "a-z0-9A-Z" in kernel code for generating words, we change it from global variable to share variable to have faster access time with shared memory in the GPU device.

3.2.3 Unrolling MD5 Function

In the rolling version, we need to run 64 iterations to generate MD5 hash, which gives us a hint to unroll this for loop and use a predefined constant number for each round.

3.2.4 Unrolling For Loop in Thread

In our kernel function, each single thread will crack `HASHES_PER_KERNEL(128 in our code)`. Because of constant iteration number, we can use `#pragma unroll` to take advantage of the compiler.

3.2.5 Using More Threads

We double our thread from 256 to 512 and the performance only slightly increases at this point. (~1.03x)

3.2.6 Unrolling the Remaining Loop: Padding

After the above optimizations, we found that the only loop remaining we haven't unrolled yet is the padding function in the first step of MD5 algorithm. However, the padding function cannot be directly unrolled with unroll pragma automatically because it depends on the length of the input text. Because we know our length is always from 1 to around 8, we used a switch statement and unrolled every case with length 1 to 8, and this results in 1.06x performance increase.

3.2.7 NVIDIA Visual Profiler

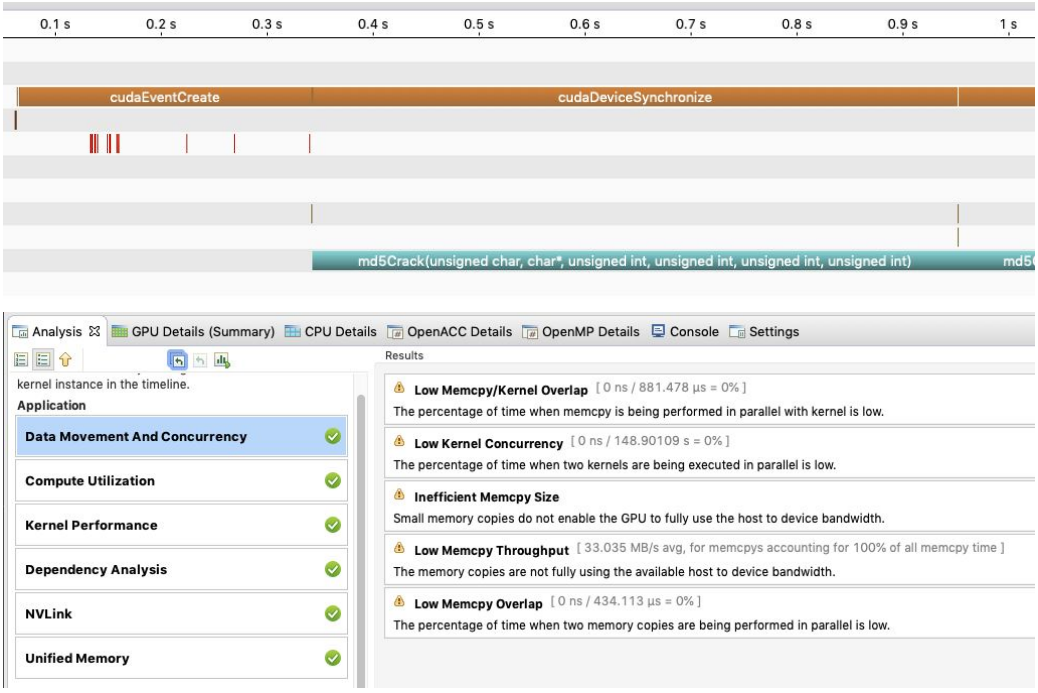


Figure 2: NVIDIA Visual Profiler results and its suggestions.

We used NVIDIA Visual Profiler [3] along the way as we optimized our performance. Figure 2 shows a screenshot of our final stage. As shown in the top red part, **very little time is spent in memory, and most of the time is in computation**, as our optimized workload is highly computation bound. We tried to follow its suggestions, but after several trials we cannot get any more performance improvements.

3.2.8 Using more GPUs

To further increase the performance, we experimented using 2 GPUs and 4 GPUs. Because there is very little overhead between host and kernel computations, and all the computations on the kernel are independent of each other and require no communication or synchronization, we observe a almost **linear gain in performance when increasing the number of GPUs**. In our case, 2 GPU is 1.96x faster than 1 GPU, and 4 GPU is 1.97x faster than 2 GPU.

4. Overview of results. Demonstration of your project

Our solution can crack passwords with **length from 1 to 7** using exhaustive brute-force attack. If using the final optimized 4 GPU setup, exhaustively searching over all 1 to 7 characters length takes less than 10 minutes.

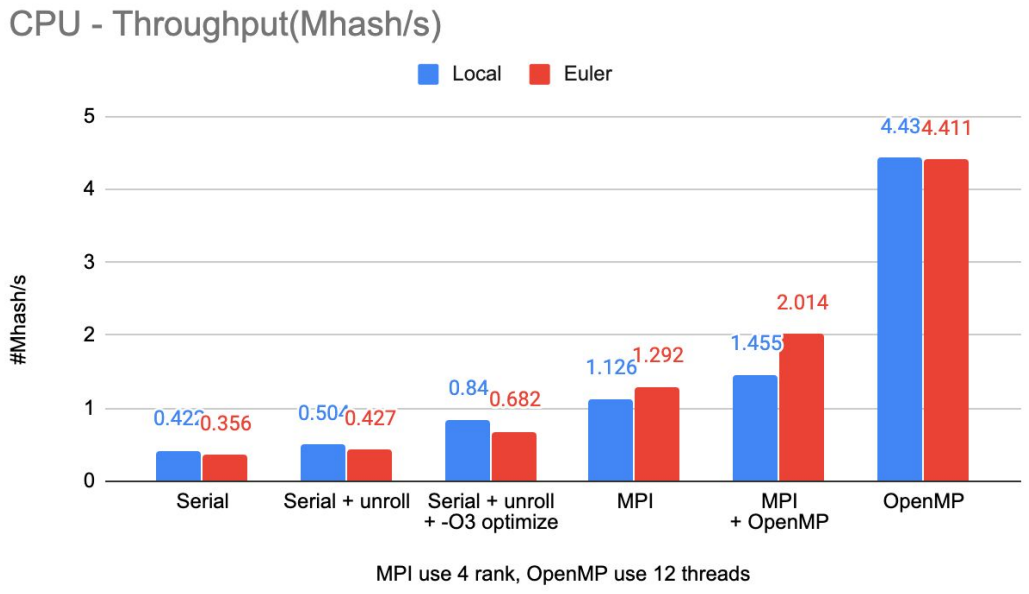
Compared to the baseline CPU serial version, the final optimized version with 4 GPU results in more than **19000x speedup**.

Below we discuss our experiment results and performance analysis for both CPU solution and GPU solution.

4.1 CPU solution

We tested our CPU solution both locally on a 15-inch Macbook Pro (2019) with Intel I9-9980HK CPU[4], and on Euler with Intel Xeon CPU E5-2650 v3 [5].

The figure below shows the throughput from the basic serial CPU version to optimized and parallelized versions. To make sure our local test result is scheduled on physical threads when using 12 threads, we checked that the I9-9980HK CPU has 8 cores and 16 threads, so **12 threads on OpenMP** should be safe.



From the tables below, we can see the individual speedup each optimization gives us, and the cumulated speedup after all optimizations compared to the baseline serial version.

Speedup	Device	Throughput	Method
1x	CPU I9-9980HK	0.422 Mhash/s	Basic Serial

1.19x	CPU I9-9980HK	0.504 Mhash/s	unroll MD5 function
1.67x	CPU I9-9980HK	0.840 Mhash/s	-O3 optimize
1.34x	CPU I9-9980HK	1.126 Mhash/s	Pure MPI
1.29x	CPU I9-9980HK	1.455 Mhash/s	MPI + OpenMP
3.04x	CPU I9-9980HK	4.431 Mhash/s	Pure OpenMP

15-inch Macbook Pro (2019) (MPI 2 rank, OpenMP 12 threads) CPU Core i9 (I9-9980HK) [4]
Total **10.5x** speedup in our CPU version on **local**

Speedup	Device	Throughput	Method
1x	CPU E5-2650 v3	0.356 Mhash/s	Basic Serial
1.19x	CPU E5-2650 v3	0.427 Mhash/s	unroll MD5 function
1.59x	CPU E5-2650 v3	0.682 Mhash/s	-O3 optimize
1.89x	CPU E5-2650 v3	1.292 Mhash/s	Pure MPI
1.56x	CPU E5-2650 v3	2.014 Mhash/s	MPI + OpenMP
2.19x	CPU E5-2650 v3	4.411 Mhash/s	Pure OpenMP

Euler (MPI 2 rank, OpenMP 12 threads) Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz [5]
Total **12.39x** speedup in our CPU version on **Euler**

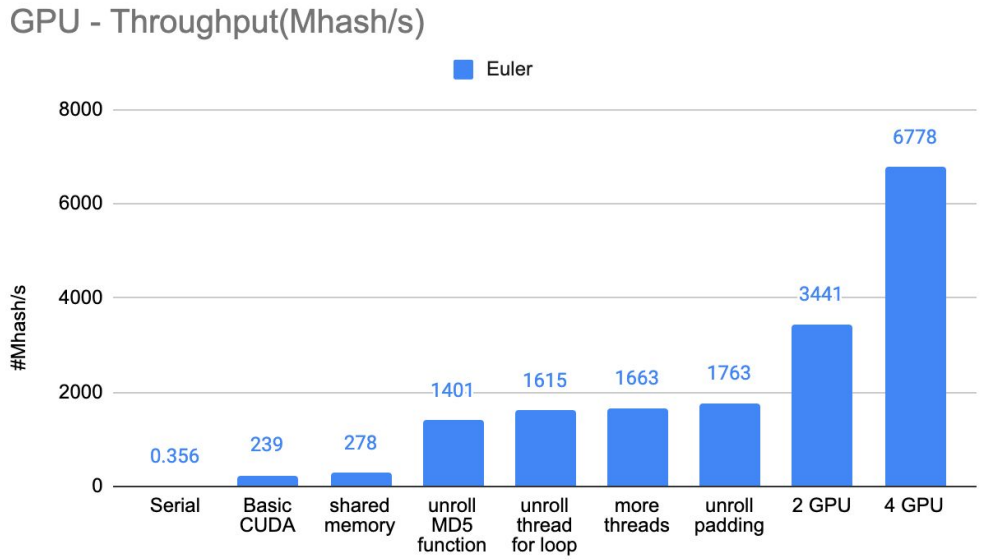
Some interesting observations and discussions:

- MPI is slower than OpenMP:
  Thread synchronisation is more compact than process synchronisation. For process synchronization, there are overheads in transferring messages from between processes; while threads share data, so no such communication overhead. OpenMP could suffer from memory limitations, but MPI acts well when a problem involves **large memory**. Here MD5's data transmission is little, the **bottleneck lies in huge computation**, so MPI's memory advantage is not utilized.
- MPI + OpenMP(hybrid) is not necessarily better than pure OpenMP:
  Hybrid = distributed memory MPI across nodes + shared memory OpenMP in single node. The hybrid parallelization is more suitable for **large applications**. In MD5, it is for-loop based to independently crack passwords. OpenMP threads distribute work and synchronize well, exploiting loop-level parallelism.

4.2 GPU solution

For all the GPU version experiments, we use Euler with GeForce GTX 1080.

The below figure shows the throughput from basic serial CPU version to the most optimized version using 4 GPUs.



The following table shows the individual speedup for each optimization, and the cumulated speedup after all optimizations compared to the baseline serial CPU version.

Speedup	Device	Throughput	Method
1x	CPU E5-2650 v3	0.356 Mhash/s	Basic Serial CPU
671x	GeForce GTX 1080	0.239 Ghash/s	Basic CUDA
1.16x	GeForce GTX 1080	0.278 Ghash/s	shared memory
5.03x	GeForce GTX 1080	1.401 Ghash/s	unrolling MD5 function
1.15x	GeForce GTX 1080	1.615 Ghash/s	unrolling for loop in Thread
1.03x	GeForce GTX 1080	1.663 Ghash/s	more thread 256 -> 512
1.06x	GeForce GTX 1080	1.763 Ghash/s	unrolling padding
1.96x	GeForce GTX 1080	3.441 Ghash/s	2 GPU
1.97x	GeForce GTX 1080	6.778 Ghash/s	4 GPU

Total **19039x** speedup in our multi-GPU version compare to Single CPU serial version

Some interesting observations and discussions:

- Basic CUDA is already more than **600x faster** than basic serial CPU. We think it is because the brute-force searching of all combinations is almost "**embarrassingly parallel**", and extremely suitable for using GPU. Furthermore, this workload is heavily computation bound, so without doing much optimization, the performance is not blocked by common drawbacks caused by bad memory transfer patterns.

- From the figure above we can see that the bigger "jumps" are using GPU, unrolling MD5 function, and using multiple GPUs. For using GPU and using multiple GPUs, it is again due to our workload being almost "embarrassingly parallel". The interesting part is unrolling MD5. unrolling MD5 function itself yields more than 5x speedup, which was surprising. Because the main part of MD5 algorithm is a **for-loop with 64 iterations**, and each iteration has 4 if statements to determine what modification to use based on the iteration index. However, the whole loop with all the **if-statements can be unrolled** because the if condition depends on only the iteration index. Unrolling the whole function has the benefit of **enabling compiler optimization and pre-fetching**, and uses **more pipelining**.

- When we added more threads from 256 to 512, we were expecting a speedup close to 2x, as with the multiple GPU results. However, it only increased 1.03x. We think it might be that the kernel has already encountered some **resource wall**, so adding more threads could not help. More threads means higher occupancy, while occupancy is just one of many considerations that affect performance. Within one block, if there are more threads, we need more registers to hold intermediate values. Besides **register resources**, **the maximum number of active threads**, **number of warp schedulers**, **number of active blocks per streaming multiprocessor**... etc are also significant factors. Hence, max thread num doesn't imply the best experiment setting.

- After unrolling all the loops we possibly can find, including the padding function using **switch-statements** as described before, we cannot speed up the performance anymore. Although adding more GPUs can improve the overall performance, when **comparing hash per GPU per second**, it doesn't make sense. We think to further optimize the performance, we will need to change the way we generate text and move data between host and GPU. NVIDIA Visual Profiler suggests that our memory copy and kernel computation overlap is low, but we have almost no memory copy time. Therefore we think if we do some more pre-computation in the host and transfer more data to kernel memory concurrently, we may potentially have more speedups.

5. Deliverables:

This report is uploaded in Canvas.

- Our git repo folder is [2020-Spring-ME759-FinalProject]

- The git repo contains 2 subfolders, `cpu/` and `gpu/`

- `cpu/`
    - `vim run.sh    # modify md5 hash you want to crack`

    - `sbatch run.sh # this will compile & run (Basic, OpenMP, MPI, Hybrid)`
    - note that cpu approach is proper for password length <= 5
    - default password length is 5, if want to test longer length, modify `PASSWORD_LEN` in util.h line 3

- `gpu/`
    - `vim run.sh    # modify md5 hash you want to crack`
    - `sbatch run.sh # this will compile and run multiple gpu version`
    - note that gpu approach is proper for password length <= 7

6. Conclusions and Future Work

We have implemented an exhaustive brute-force attack to crack passwords. Our results show that if using a 4 GTX 1080 setup, exhaustively searching over all **1 to 7 characters length takes less than 10 minutes**. We compared the baseline CPU serial version with the final optimized version using 4 GPUs, and the performance gain is **more than 19000x**.

After all our optimizations on the GPU version, we compared our result with the state-of-the-art password cracking software: **hashcat** [6]. On the same GPU GeForce GTX 1080, hashcat achieves **24.809 GHash/s** for MD5 cracking, which is 14x faster than our single GPU approach. This shows that we still have a huge room of optimization. Therefore, one of the future work we can do is to look at their source code and try to investigate where the improvements come from.

Another future work we can try is to experiment with different hashing functions such as SHA-1, SHA-256, SHA-512 etc., or further experiment with cracking hash with salt and/or pepper, which seems like a more complex setup and will be less “embarrassingly parallel”.

References

[1] Wikipedia MD5: https://en.wikipedia.org/wiki/MD5
[2] IETF documentation on RFC 1321: https://tools.ietf.org/html/rfc1321
[3] Nvidia Profiler document: https://docs.nvidia.com/cuda/profiler-users-guide/index.html
[4] Intel® Core™ i9-9980HK Processor Spec
https://www.intel.com/content/www/us/en/products/processors/core/i9-processors/i9-9980hk.html
[5] Intel® Xeon® Processor E5-2650 v3
https://www.intel.com/content/www/us/en/products/processors/xeon/e5-processors/e5-2650-v3.html
[6] hashcat https://github.com/hashcat/hashcat