Laboratory Course High Performance Programming with Graphics Cards: Computing SHA256 hashes on the GPU

Li-Wei Chen, Patrick Franczak, Debasish Mukherjee

September 18, 2018

1 Introduction

Passwords are stored in a form of a hash in many cases. A hash function is a one-way function transforming a plaintext string to an unique string - called the hash. This hash has a fixed length. In the case of SHA256, its length is 256 bit, i.e. 64 characters. Many hash functions are standardized, e.g. SHA256 is defined in the Federal Information Processing Standards Publication 180-2 [1]. For example, the SHA256 hash of the String `stuttgart` is

```
9DC5BAD7A55031B93AEB1DD8D76673D84582FB0FAEB4ECDB1B99423296447651
```

The reason for storing passwords as hashes is to prevent an attacker from reading the passwords in plaintext if a password database is leaked. Since hashes are unique and generated by a one-way function, it is not possible to reveal the plaintext just by knowing the hash. Hash function are a way to make it harder for an attacker to get valid login data. In some cases, there are theoretical attacks on hash algorithms but they are not all feasible in practice.

2 Challenge

Nowadays, passwords usually have a length of six characters and more. With increasing length, the number of possible passwords to guess increases exponentially as it can be seen in Figure 1. We look at permutations with repetitions, i.e. a character can be used multiple times on various positions in the password string. Assume, the possible alphabet per position is n and the password length is k. The number of possible permutations equals n^k. For lowercase letters $(a, b, \ldots, z)$ only $(n = 26)$ and password length 6 $(k = 6)$, There are $26^6 = 308915776$ permutations, which is a huge number. To determine the correct plaintext for a given hash, we have to compute the hash for every possible permutation, which can be computationally very intensive. Using the property of hash algorithm, each password has an unique hash, we can compute every hash independently, making it easy to parallelize the computation.

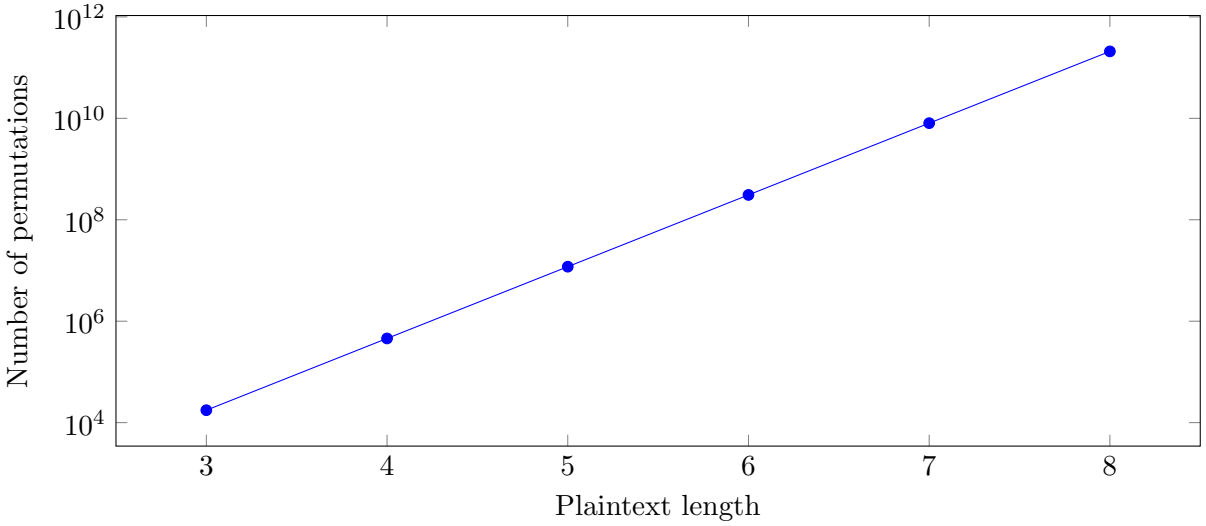

Figure 1: Computational complexity increasing with password length

3 Implementation

The project was implemented in C++11 and OpenCL, which was a subset of C99 without some header files and lacking support of recursion. This section provided an insight how the project was practically implemented. Technical details and challenges were addressed as well.

3.1 SHA256 Hash Algorithm

A hash function maps a message M of arbitrary length to n bits, which is called a hash. Such hash needs to be one-way and collision resistant. A collision occurs, if a hash function generates an identical hash for two distinct inputs. SHA256 provides a 256-bit hash and 128 bit security which implies it requires 2^{128} hash computations to find a message that results to a specific hash. In short the SHA256 processes the input in two distinct steps [1]:

1. Pre-Processing
2. Secure Hash Computation

3.1.1 Pre-processing

For a message M of length L bits, determine K, subject to the following equation:

$$L + 1 + K \equiv 448 \mod 512 \tag{1}$$

Next, pad the message M as follows:

1. Append bit '1' to the end of message.
2. Add K zero bits to the end.
3. Append the 64-bit binary representation of L.

This padded message is parsed into N 512-bit blocks as $M^{(1)}$,$M^{(2)}$...$M^{(N)}$. Each message block can be represented by sixteen 32-bit words. The first 32-bit of $M^{(i)}$ denotes $M_0^{(i)}$, next 32-bit of $M^{(i)}$ denotes $M_1^{(i)}$, and up to $M_{15}^{(i)}$.

3.1.2 Secure Hash Computation

The algorithm begins with initial hashes, $H_0^{(0)}$,$H_1^{(0)}$...$H_7^{(0)}$, each consists of 32-bit words.

$$
\begin{aligned}
H_0^{(0)} &= 6a09e667 \\
H_1^{(0)} &= bb67ae85 \\
H_2^{(0)} &= 3c6ef372 \\
H_3^{(0)} &= a54ff53a \\
H_4^{(0)} &= 510e527f \\
H_5^{(0)} &= 9b05688c \\
H_6^{(0)} &= 1f83d9ab \\
H_7^{(0)} &= 5be0cd19
\end{aligned}
$$

The used functions are described below. All of these functions are operating bitwise.

$$
\begin{aligned}
Ch(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
\sum\nolimits_0^{(256)}(x) &= S^2(x) \oplus S^{13}(x) \oplus S^{22}(x) \\
\sum\nolimits_1^{(256)}(x) &= S^6(x) \oplus S^{11}(x) \oplus S^{25}(x) \\
\sigma_0(x) &= S^7(x) \oplus S^{18}(x) \oplus R^3(x) \\
\sigma_1(x) &= S^17(x) \oplus S^{19}(x) \oplus R^{10}(x)
\end{aligned}
$$

The main SHA256 algorithm and the message schedule are depicted below. The message schedule algorithm leads to calculation of the expanded message blocks W_0, W_1, ...,W_{63}.

Algorithm 1 SHA256 main loop

for $i \leftarrow 1$ to N **do**
  $a \leftarrow H_1^{(i-1)}$
  $b \leftarrow H_2^{(i-1)}$
  $\vdots$
  $h \leftarrow H_8^{(i-1)}$
  for $j \leftarrow 0$ to 63 **do**
    $T_1 \leftarrow h + \sum_1(e) + Ch(e, f, g) + K_j + W_j$
    $T_2 \leftarrow \sum_0(a) + Maj(a, b, c)$
    $h \leftarrow g$
    $g \leftarrow f$
    $f \leftarrow e$
    $e \leftarrow d + T_1$
    $d \leftarrow c$
    $c \leftarrow b$
    $b \leftarrow a$
    $a \leftarrow T_1 + T_2$
  $H_1^{(i)} \leftarrow a + H_1^{(i-1)}$ ▷ Compute i_{th} intermediate hash value $H_1^{(i)}$
  $H_2^{(i)} \leftarrow b + H_2^{(i-1)}$
  $\vdots$
  $H_8^{(i)} \leftarrow$ a + $H_8^{(i-1)}$
$H^{(N)} = (H_1^{(N)}, H_2^{(N)}, \ldots, H_8^{(N)})$ ▷ The hash of M
return $H^{(N)}$

Algorithm 2 SHA 256 Message schedule

$W_j = M_j^{(i)}$ for j $\leftarrow$ 0 to 15
for j $\leftarrow$ 16 to 63 **do**
  $W_j \leftarrow \sigma_1(W_{j-2}) + W_{j-7} + \sigma_1(W_{j-15}) + W_{j-16}$

3.2 Program flow

First, the user has to generate a wordlist with a fixed word length. The length has to be changed once in the main cpp-file and once in the kernel code. The program flow is visualized in Figure 2. Our program begins by asking the user for three inputs: hash to search, which platform (CPU or GPU or both) to compute the hash on, and whether to write the generated hashes to a file or not. The open source Linux tool, `crunch`, is used to generate an exhaustive list of combinations of predefined length characters/symbols for the chosen character set. These generated potential passwords are then sliced into chunks of size, dependent on the work-group size of the graphics platform the target system has. Depending on the input from the user, these slices are fed to the CPU, or GPU, or both to generate the hash according to the SHA 256 algorithm implemented on these platforms. These generated hashes are then either saved to a file or not depending on the input choice provided by the user. There are two modes of comparison, depending on whether the generated hashes are saved or not.

1. For the case of not saving the hashes, the input hash (the hash from the user to be broken) is searched in the output hashes of each slice. If the hash is found on the current slice, the plaintext-hash pair is displayed in the console leading to the successful recovery of the password/plaintext and the program stops generating more hashes. If the hash is not found, the program continues with the next slice until the input hash is found in one of the slices. If the input hash is never found, an appropriate message is displayed.

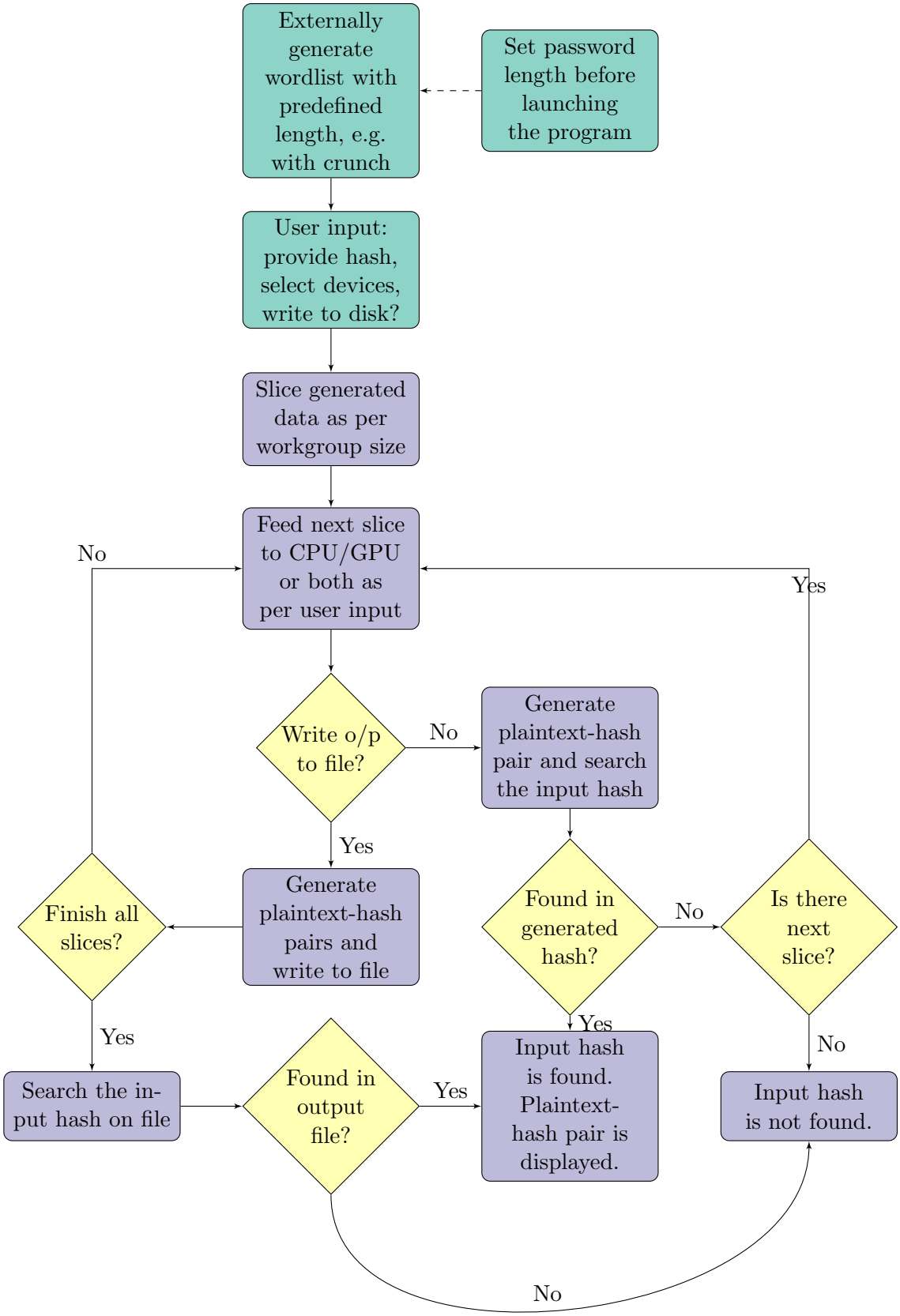

Figure 2: Flow chart of the program

2. For the case of saving the hashes, the program generates all the plaintext-hash pairs from all the slices and then compares the input hash with the generated hashes in the file. If the hash is found, the plaintext-hash pair is displayed in the console leading to the successful recovery of the password/plaintext. If the input hash is never found, an appropriate message is displayed.

It must be noted, that the writing to a file requires significant processing time. If the character/symbol set is fixed for a particular application, then, it is advisable to write the generated hashes to a file while running the program for the first time. On subsequent runs, the hash to be cracked can be simply searched from the stored hashes thereby reducing processing time. However, for the demonstration of the efficiency of our kernel code, we have focused on generating the hashes every-time the tool is run.

3.3 Computation of a SHA256 hash

On the CPU side, the hash is computed by the SHA module of the OpenSSL library. To make use of the library, the parameter `-lcrypto` has to be added to the g++ Linker. On the GPU side, we developed a kernel which is capable of computing SHA256 hashes based on the work [2] and [3]. Thanks to C++11, we could create two structures called *wordStruct* and *outStruct* and put them into a std::vector. This makes it possible to feed the GPU directly with the desired input for each kernel run and have an output, which is easy to handle.

4 Results

We benchmarked our program on two independent systems.

System	1	2
CPU	Intel Core i5-2500K (Sandy Bridge) 4.3GHz	Intel i7-5820K (Haswell) 3.3GHz
RAM	16GB DDR3	32GB DDR3
GPU	Nvidia Geforce GTX 1060 6GB of VRAM	AMD Radeon R290 8GB of VRAM
OS	Ubuntu 18.04 LTS	Ubuntu 16.04 LTS
GPU Driver	Nvidia Driver Version 390.48	AMD Driver Version 16.60.3
G++ version	7.3.0	5.4.0

Table 1: Systems' specifications.

For benchmarking, we chose four wordlists generated by the open-source tool *crunch*. Every wordlist had just lower letter alphabet and a fixed password length from 3 to 6. The wordlists were generated by the following command:

```
$ crunch k k -f charset.lst lalpha -o lalphak.txt
```

Where `k` is the password length and the `charset.lst` file resides in the same directory as the executable file of crunch. It lists the different types of character/symbol lists. The wordlists had the following structure. Every possible permutation had its own line, followed by a linebreak character. Thus, each wordlist had a size of

$$\textit{Number of permutations} \cdot (\textit{password length} + 1)$$

in bytes. The smallest wordlist had a size of 70KB. The largest one had a size of 2.2GB.

4.1 Comparing CPU and GPU Time and Speed-up

Although both systems used different CPUs, the CPU times were very similar. This allowed us to compare time and speed-ups between both systems. Specific speed-up on the system 2 was shown in table 2.

System 2		
Plaintext length	Speed-up w/o memory copy time	Speed-up w/ memory copy time
3	51.81	31.75
4	95.94	45.35
5	191.77	29.47
6	192.80	29.19

Table 2: System 2 w/o #pragma unroll.

On both systems, the total time for computing all hashes increased linearly according to the number of permutations, whereas the complexity of the input grew exponentially with length of password/plaintext. This can be seen due to the logarithmic y-axis in Figure 3 and Figure 4. System 1 achieved a speed-up of ≈ 23 without loop unrolling and ≈ 27 compared to the CPU. The speed-up curve of System 2 was not as smooth as the curve of System 1. Still, the speed-up factor was around 29 compared to the CPU. On system 2, compiler assisted loop unrolling (`#pragma unroll`) was not possible due to missing compiler support. Hence, all times and speed-ups of system 2 did not include compiler assisted loop unrolling.

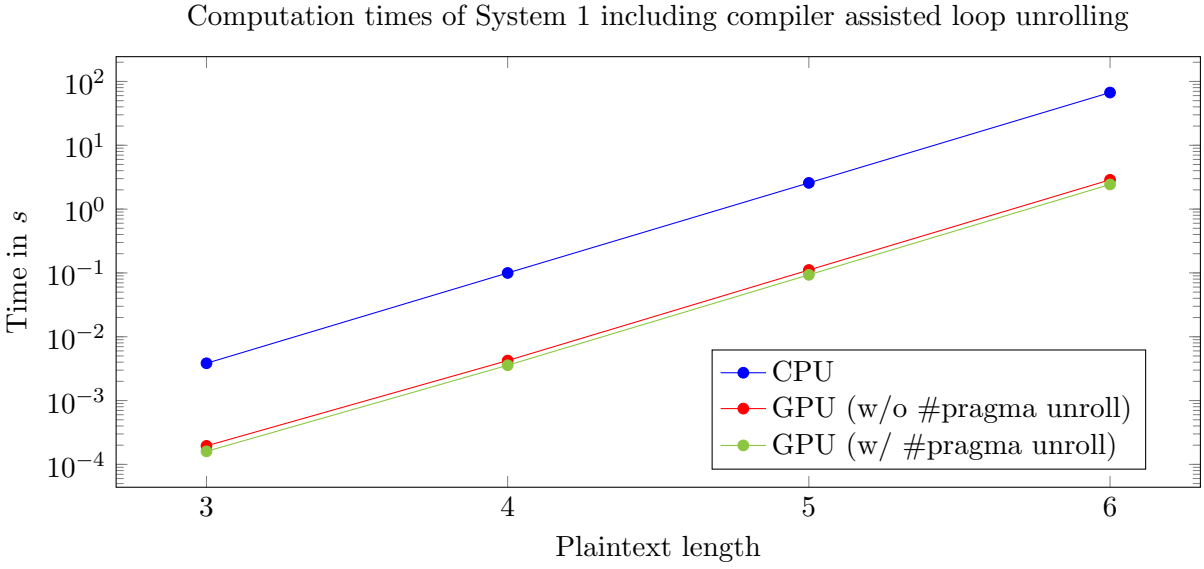

Figure 3: System 1 with compiler assisted loop unrolling (#pragma unroll). Displayed GPU times include memory copy time.

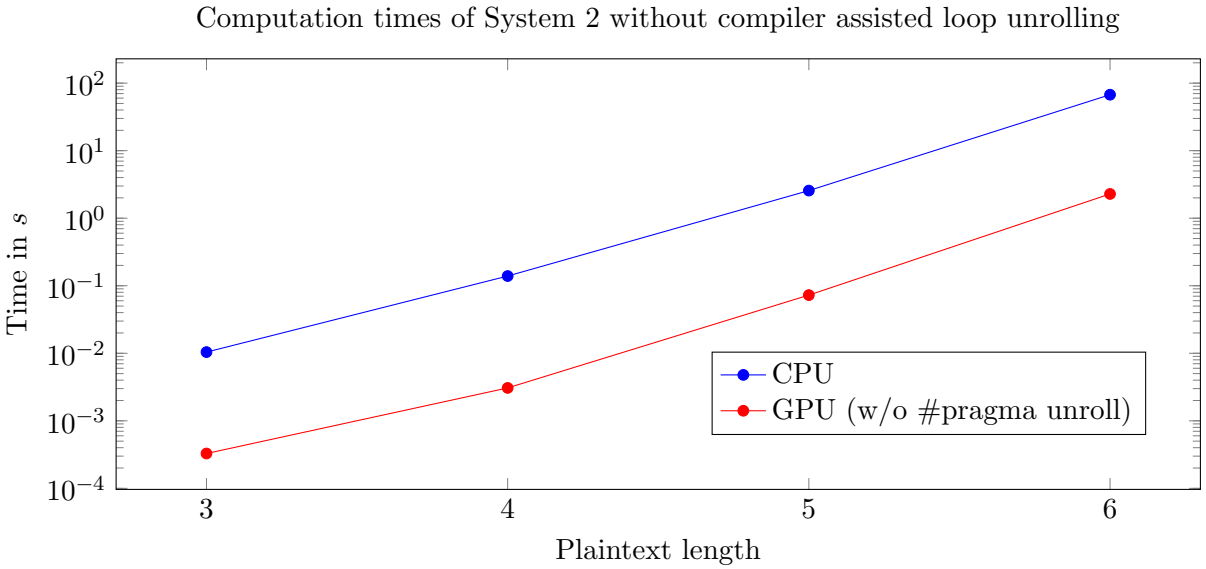

Figure 4: System 2 without compiler assisted loop unrolling (#pragma unroll). Displayed GPU times include memory copy time. Compiler-assisted loop unrolling was not possible on this system due to missing compiler support.

4.2 Comparing AMD and NVIDIA Time and Speed-up

From Figure 5, we observed that the smallest wordlist (`lalpha3.txt`) was too small to bring up the full performance of both GPUs for computation time only. If one focused on the total computation time including memory transfer time, both GPUs reached their peak even for the smallest wordlist. After feeding the GPUs with a plaintext length of 4, the NVIDIA GPU in System 1 reached its speed-up peak at ≈ 90 with loop unrolling enabled and ≈ 56 otherwise. The AMD GPU in System 2 behaved differently. Regarding computation only time, it reached its speed-up peak at ≈ 190 when the second largest wordlist (`lalpha5.txt`) was used. For the plaintext length of 4, the speed-up of AMD GPU was higher than the NVIDIA GPU with loop unrolling in system 1 (≈ 95 vs. ≈ 90). After the plaintext length 5 was used, the AMD GPU from System 2 outperformed the NVIDIA GPU from System 2. When considering the total computation time, the AMD GPU was still faster than the NVIDIA one.

System 1 offered support for compiler assisted loop unrolling. It was used by writing the compiler directive `#pragma unroll` in the line before every for loop used in the kernel code. Using this directive, the compiler unrolled the loop in compile time, meaning the loop iterations were written as individual statements. This technique saved some instructions since the for loop control had to check if the condition for the loop variable was fulfilled in every iteration. Specific speed-up on the system 1 was shown in table 3. For computation time only, an average speed-up of $\approx 40\%$ was achieved. Including memory time, the speed-up got smaller because the influence of the computation time was smaller. Hence, the speed-up here was around 15%. In fact, it was worthy to do compiler assisted loop unrolling, if available.

		System 1	
Plaintext length	Time	Speed-up w/o #pragma unroll	Speed-up w/ #pragma unroll
3	Execution	40.41	63.98
	Total Time	19.68	23.99
4	Execution	56.99	91.55
	Total Time	23.57	27.94
5	Execution	55.33	93.70
	Total Time	23.18	27.96
6	Execution	56.13	89.19
	Total Time	23.30	27.54

Table 3: System 1 w/ #pragma unroll. Displayed total times include memory transfer time as well.

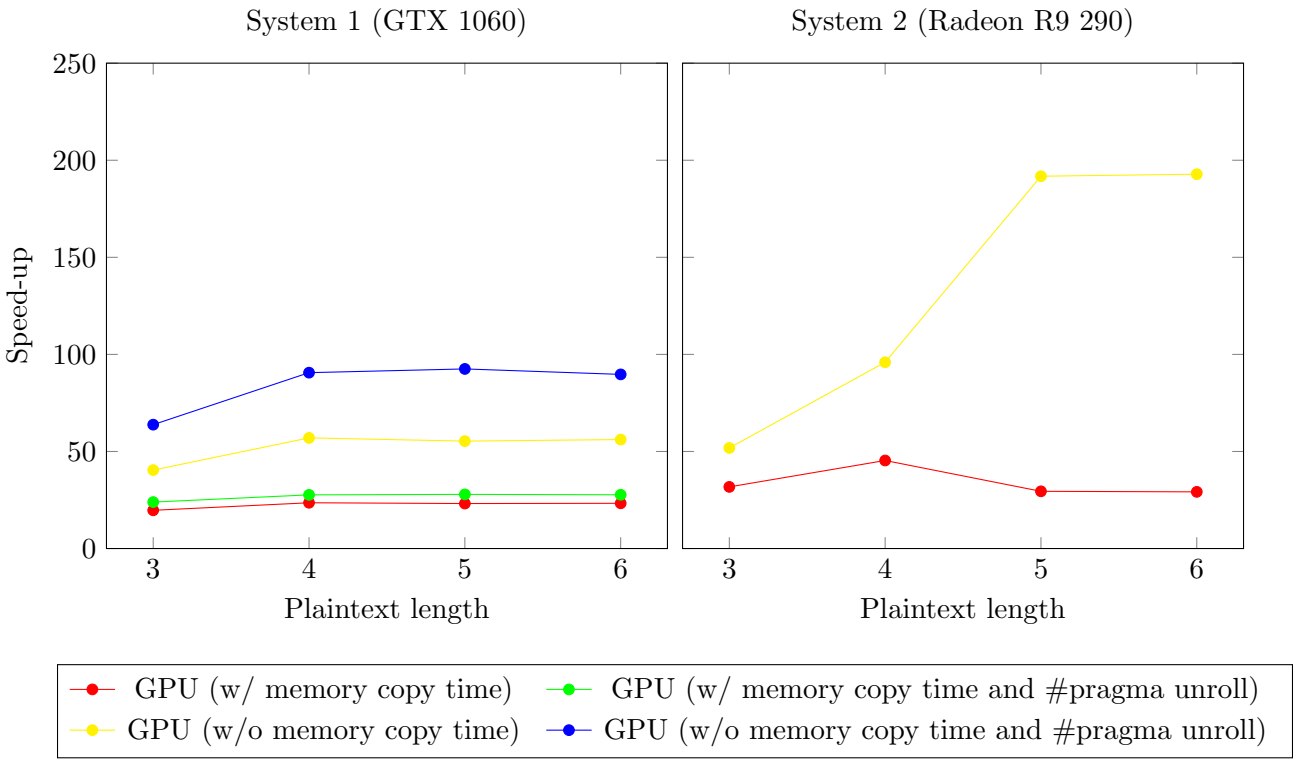

Figure 5: Comparison of the Speed-up between System 1 on the left and System 2 on the right. The GPU on System 2 is significantly faster than the GPU from System 1.

5 Conclusion and future outlook

Both systems achieved a significant speed-up when the GPU instead of the CPU. With compiler assisted loop unrolling, system 1 achieved an even higher speed-up compared to the CPU. The AMD GPU used in System 2 performs significantly better in OpenCL than the NVIDIA GPU does, although it generally performs worse in DirectX based video games. The giant ability of parallelization by graphic cards make them capable of exploiting data parallelism very easily. As shown in the previous section, even a modern multicore CPU was outclassed by a single GPU. When SHA256 was presented to the public in 2002, GPUs were not considered to have such a huge performance increase over the years. With powerful GPUs, more complex password combinations are getting feasible to be broken in a short time. This lowers the security level of particular hash functions, including SHA256. In order to make it harder for an attacker to break the hash, the successor of SHA256, named SHA3-256 increases difficulty for calculating the hash. This results in much higher computation time for breaking the hash on any device, even on powerful GPUs.

References

[1] Secretary of Commerce. "Secure Hash Signature Standard (SHS) (FIPS PUB 180-2)". In: *August* 1 (2002), p. 72.

[2] noryb009. *sha256*. `github.com/noryb009/sha256/blob/gpu/sha256.cl/`. 2016.

[3] zunceng. *opencl sha al im*. `github.com/Fruneng/opencl_sha_al_im/`. 2017.