Cryptographic algorithms

Kishan H. Kavathiya
B.Tech ICT with minor in Computational Science
201701155@daiict.ac.in

Abstract — Cryptographic algorithms are very important to encrypt our important data. In many popular internet applications, Password-based Key derivation function-2(PBKDF2) is used as the encryption algorithm. In the derivation of PBKDFs, Secure hash functions are the key parameters. Due to GPU's highly parallel architecture, GPUs are ideal for performing brute force attacks. In this works, we understand the architecture of GPU and CUDA programming model to implement SHAs,PBKDFs.

I. Introduction

Encryption is a process that encodes a message or file so that it can be only be read by authorized parties. Encryption process uses a parameter – the **key**. The key is used to encrypt data and only known to authorize users such that data can be protected from unauthorized users. The Key is in the form of the password or passphrase (text). But, most cryptographic secure algorithms expect key in the form of the binary string. So, our first step is to convert Key in the binary form.

Key derivation functions are used to derive encryption keys from a secret value. The secret value can be another key or a password or passphrase. A KDF that is designed for deriving key from another key is called a Key Based Key Derivation Functions (KDKDF). A KDF that is designed for deriving key from another password is called a Password Based Key Derivation Functions (PBKDFs).

An attack on a cryptographic key is an attempt by an unauthorized party to determine the key from publicly known information or from certain partial information about the key. A **brute-force attack** consists of an attacker submitting many common passwords. In **rainbow table attack**, attacker first precomputes the hash of common passwords and stored in rainbow table (databases).

Generally, guess a password and enter into the login system if the guess is right then the password is cracked. If one is able to guess 1000 passwords per second (which is impossible) and password length is only 5 letters (40-bit) then one needs to consider 2^{40} possibilities. This password cracking process takes 34,865 years. Nowadays, all software industries prefer to store hashed passwords (hashing digest) rather than the direct password (text) into their databases. The main reason is that if any data breaches containing passwords are leak then unauthorized only will able to get encryption passwords. So, they have to first decrypt the hash. And leak of data breaches is common. Currently, Most Secure hashing algorithm ensures that the decryption of a password is impossible. So, try brute force attack or rainbow table attack for password cracking.

Here, we are implementing a cryptographic algorithm in GPU. First, understand the GPU. After, we understand the programming model of CUDA. Then, understand the cryptographic algorithm. Finally, implement the SHA256,PBKDF2 functions in GPU and CPU.

II. Understanding Of GPU

GPU is a Graphics Processing Unit. GPU is used to accelerate the task on a computer. GPU is originally designed for the acceleration of computer graphics and gaming industries. GPUs have recently also becomes a powerful platform for general-purpose parallel processing. Generally, the password cracking process takes very much time and memory. For example, the CPU needs 20 hours to crack the password and if GPU is able to achieve speedup 10 then the password cracking process complete in only 2 hours.

To optimally use of computational power of GPU, the task must be divisible into small tasks that all task performs the same but over a different piece of data.

Here, we discuss the architecture of GPU, the memory hierarchy of GPU, a basic understanding of the logical and programming model of CUDA. We are discussing the CUDA programming model, which is the heterogeneous model which means running on both the GPU as well as the CPU. The discussed architecture follows the CUDA model. GPU architectures that do not support CUDA may differ.

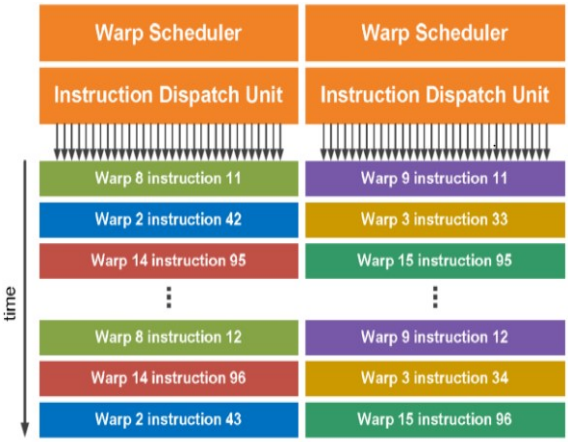
A. Architecure of GPU

A GPU consists of global memory and several **streaming multiprocessors** (SMs). Data can be transferred from the Host's main memory to device memory (global memory) via the PCI bus. In the case of the integrated GPU, the host's main memory is already shared with the GPU. There are also systems that consist of more than one GPU.

Streaming Multiprocessor (SM)

SMs are mainly designed for data parallelism. Each SM consists of one instruction unit (or known as the CUDA core which is similar to the CPU's Control Unit) with instruction cache, constant cache, shared memory, several SPs (execute arithmetic instructions) and several SFUs (execute special mathematical functions). SMs also contain multiple registers to load instructions.

Each SM executes a lot of threads. Here, thread refers to the number of execution and kernel refers to the single program (consist of a lot of instruction) because each thread works on similar instruction set over the different datasets. Threads are divided into blocks. Blocks are further divided into smaller wraps of 32 threads. All threads in a warp always execute the same instruction. SMs also contain two wrap schedules and two dispatch units. Also, SM can switch between currently running wraps to avoid memory latency (if any I/O requests occurring than all SPs and SFUs are free). Wrap scheduling is handled by hardware, which minimizes scheduling overhead.



Threads executing wraps.

GPU architecture follows the atomic scalability. GPU is built around an array of SMs. A multithreaded program is partitioned into blocks of threads that execute independently from each other so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

In CPU, the execution of instructions is out-of-order while, in GPU (in SMs), the execution of the instruction is always in-order. Memory latency is avoided by switching.

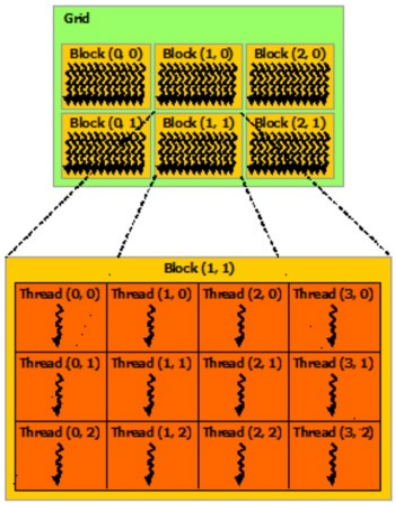
B. Logical Model

The kernel is a function that runs on the GPU (device) and invoked from the CPU (host). CUDA applications compose of multiple blocks of threads with each thread call a kernel once. In a kernel, many threads are invoked.

Here, Grid contains 6 blocks and each block contains 12 threads.

#no. of tasks = (#no. of threads) •(#no. of tasks in 1 thread)

#no. of threads = (grid size) • (block size)



If the number of threads is N and block size is n than the number of blocks in grid should be N/n.

Kernel can be defining using keyword **__global__**.

```
// Define a Kernel
__global__ void kernelFunction(){
    // Code
}
```

When Kernel is invoked, all threads invoked once independently. Kernel can be invoked using <<< >>> .

```
// Invocation of a kernel
kernelFunction <<< gridSize, blockSize>>>()
```

When a thread is invoked then it is given a unique block ID and unique thread ID. Block id is accessible through the keyword blockIdx. Thread id is accessible through the keyword threadIdx. They have properties like .x, .y, and .z . CUDA runtime launches a 3D grid of blocks of dimensions gX•gY•gZ. Each of those blocks will contain threads organized in a 3D structure of size tX•tY•tZ. They are using to determine which number of task we want to execute in the kernels.

$$Index_{task} = Index_{block} \bullet (\text{block size}) + Index_{thread}$$

As we discussed in architecture of GPU, in wrap all threads execute the same instruction. So, maximized efficiency achieved when all threads of a wrap follow the same execution paths. If any threads of a wrap diverge via a data-dependent conditional branch the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

Loop unrolling technique is used to avoid branching. No need to check conditions.

```
// Simple for loop
for(int i=0;i<4;i++){
    ans = ans + a[i]
}
// Loop unrolling
ans = a[0] + a[1] + a[2] + a[3]
```

C. Memory Model

Global Memory, **Constant Memory**, **Texture memory** is accessible to all threads of SM whereas each block has its own **shared memory** which is accessible by all threads of the blocks. And each thread has its own **registers** and **local memory**.

Memory access time for Global Memory and Local Memory is 100 times more than the shared Memory, Cache (Constant Memory and Texture Memory), and registers. Among all memory, the register is a faster one.

The below table shows the scope, lifetime, latency (register is faster one. So, latency is in compare to register's latency) for each kind of memory.

Memory Type	Scope	Accessible from	Lifetime	Latency (Penalty)
Register	Thread	Device	Thread	×1
Local Memory	Thread	Device	Thread	×100
Shared Memory	Block	Device	Block	×1
Global Memory	SM	Host/ Device	Applicati on	×1
Constant Memory	SM	Host/ Device	Applicati on	×100

Global Memory

Global memory is the **main memory** of the GPU. GPU is type of DRAM memory. Global memory is used for transfers between the host and device as well as for the data input to and output from kernels. The size of the global memory is around 1GB-16GB.

In CUDA, global memory can be declared by **__device__** keyword outside any function or using function **cudaMalloc()**.**CudaMemCpy()** function is used to transfer data between host and device. If you want to transfer data from host to device (to initialize) use keyword **cudaMemcpyHostToDevice** and if you want to transfer data from device to host(to get the return back result) keyword **cudaMemcpyDeviceToHost** as an argument.

```
// Outside of the any function
__device__ int x[100];
// Inside Host function
int function(){
    int *data;
    cudaMalloc(&data,sizeof(int)*100)
}
```

Constant Memory

Constant memory is the part of global memory. Constant memory is **cached** and read-only memory. So, it has low latency than the global memory. Size of the constant memory in GPU is 64 KB. Each SM contains around 8-10 KB of cached (constant memory). Constant memory is optimized for two **temporal localities**.

Constant memory can be declared by **__constant__** keyword outside of any function. When we want to transfer data from global memory to constant memory use function cudaMemCpyTo Symbol() instead of CudaMemCpy().

```
// Global Scope
__constant__ int x[100]

// Host code
int function(){
    int y[100];
    cudaMemcpyToSymbol(y,x,sizeof(int)*100)
}
```

Texture Memory

Texture memory is part of the global memory which, is also **cached**. Texture memory is optimized for **two dimensional spatial localities**. Read input data through texture cache and CUDA array to take advantage of spatial caching.

Memory model

Shared Memory

Each blocks of SM contains shared memory. Shared memory is much faster than the Global and local memory. Shared memory has less latency due to access is restricted to single block.

Static Allocation (size known at compile time) can be declared by keyword **__shared__** in host code.

```
// Global scope
__shared__ int x[100]
```

Dynamic Allocation (size known at run time) can be declared only once using **extern** keyword. In the kernel launch specify the total shared memory needed.

```
extern __shared__ int x[]
int *intData = s
float *floatData =(float*)&intData[nI]

// Kernel Launch
myKernel<<<ng,nb, nI*sizeof(int)+nF*sizeof(float)>>>
(args);
```

Common pattern when using shared memory

1. Data copy into shared memory from global memory.
2. Use function **__syncthreads(),** to wait the transfer until completes.
3. Perform computation, incrementally storing output in shared memory, __syncthreads() as necessary.
4. Output copy from shared memory to global memory.

Local Memory

Local memory is used for data which are not fit into SM registers. Sometimes, if thread required more memory than global memory also allocates a block of memory as local memory. Local memory has high latency. So, the programmer should keep per-thread local data as small as possible.

Registers

It is the fastest memory. The registers memory is allocated until all threads in a block finish their execution. Most stack variables declared into registers.

D. Programming model of CUDA

CUDA allows one to execute functions on the GPU using many threads in parallel. The CUDA programming model is a **heterogeneous model** in which both the CPU and GPU are used. There are two terms are associated with CUDA programming model. The Host refers to the CPU and its memory and the device refers to the GPU and its memory. Executed functions on device are known as the kernel. These kernels are executed by many GPU threads in parallel.

CUDA is implemented in C,C++ and Fortran. Here, we are discussing about CUDA C++.

A sequence of operations for CUDA program:-

1. Define a Kernel
2. Declare and allocate host and device memory.
3. Initialize host data.
4. Transfer data from the host to the device.
5. Execute one or more kernels.
6. Transfer results from the device to the host.
7. Cleanup Kernels.

III. Cryptographic Algorithms

PBKDF2 is very popular variant of the PBKDF. PBKDF2 uses secure hash functions (like SHA1, SHA224, SHA256, SHA512 etc.) to construct more secure encryption. So, first understand common Secure Hash algorithms.

Hash function is an algorithm that consists of bitwise operation, modular operations, additions operations, and compression functions. Secure Hash Algorithms also known as SHA, are a family of cryptographic functions designed to keep data secured. SHA works by transforming the data using Hash functions.

There are many variants of SHA like SHA-0, SHA-1, SHA-2, and SHA-3. Nowadays, many applications use hash functions (SHA-256, SHA-512) of family SHA-2. The only differences between SHA-256 and SHA-512 are output size, block size, number of iterations. So, here we discuss about algorithm of SHA-256, implementation of SHA-256 in CUDA and results of SHA-256.

There are many open-source libraries for SHA256 and similar cryptographic functions in C, C++, Python. But, as we discussed in the CUDA programming model host function can't be called from the device functions or the kernel. So that we can't use predefined libraries like OpenCL or CUDA. So, we have to create our device functions that can be invoked from the kernel (device functions). Here, first, understand algorithms for SHA256 after that we discuss some notes for the implementation.

A. SHA256

SHA-256 is a 256-bit hash and is meant to provide security against collision attacks. In SHA-256, the size of the message block is 512-bit and the size of intermediate hash value is 256- bit.

Algorithm:-

I. Preprocessing the message

Convert text (password, message) into the binary string which is known as "Hashed message". Padded extra "0" and length of the hashed message is a hashed message such that the length of the resulted string is multiple of 512. Divided into 512-bit message blocks $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$.

II. Initialize Hash values (intermediate 32 bit).

SHA256 uses hash value as the first 32 bits of the **fractional parts of the square root** of the first 8 primes 2…19.

H_1 = 0x6a09e667
H_2 = 0xbb67ae85
H_3 = 0x3c6ef372
H_4 = 0x a54ff53a
H_5 = 0x510e527f
H_6 = 0x9b05688c
H_7 = 0x1f83d9ab
H_8 = 0x5be0cd19

Similarly, SHA256 uses array of round constants as first 32 bits of the **fractional parts of the cube root** of the first 64 primes 2,3,…,311.

Other SHA-2 family algorithms use different hash values. For an example, SHA224 uses hash values as the fractional part of the square root of the 9th prime to 16th prime 23…53.

III. For each message block,
   a. Initialize hash value in different registers variables.
   b. Apply SHA256 compression functions. (Iterate 64 times). For SHA224 compression function is same but it is iterated for 80 times. Implementation of Compression function is given here. (https://www.movable-type.co.uk/scripts/sha256.html)
   c. Add the compressed result value into the current hash values.

IV. Calculate final digest values. (Digest = $H_0 + H_1 + \ldots + H_8$)

SHA-256 Compression Function:-

- Compute Ch (e, f, g), Maj (a, b, c), $\sum_0(a)$, $\sum_1(a)$ and W_j.

$T_1 \leftarrow h + \sum_1(e) + Ch(e,f,g) + K_j + W_j$
$T_2 \leftarrow \sum_0(a) + Maj(a,b,c)$
$h \leftarrow g$
$g \leftarrow f$
$f \leftarrow e$
$e \leftarrow d + T_1$
$d \leftarrow c$
$c \leftarrow b$
$b \leftarrow a$
$a \leftarrow T_1 + T_2$

Where,

$Ch(x, y, z)=(x \wedge y) \text{ xor } (\neg x \wedge z)$

$M aj(x; y; z)=(x \wedge y) \text{ xor } (x \wedge z) \text{xor } (y \wedge z)$

$\sum_0(x) = S^2(x) \text{ xor } S^{13}(x) \text{ xor } S^{22}(x)$

$\sum_1(x) = S^6(x) \text{ xor } S^{11}(x) \text{ xor } S^{25}(x)$

$\sigma_0(x) = S^7(x) \text{ xor } S^{18}(x) \text{ xor } R^3(x)$

$\sigma_1(x) = S^{17}(x) \text{ xor } S^{19}(x) \text{ xor } R^{10}(x)$

$W_j = M_j \qquad 0 \le j \le 15$
$W_j = \sigma_1(W_{j-2}) + W_{j-7} + \sigma_2(W_{j-15}) + W_{j-16} \qquad 15 \le j \le 63$

PBKDF2

PBKDF2 is a generic password-based key derivation function – its definition depends on the choice of an underlying pseudorandom function (PRF).

PRF is an efficiency computability function which output is purely random and there are no algorithm exits to get the inputs back. The instantiations of PBKDF2 using specific PRFs are often denoted as PBKDF2-PRF, where PRF is the name of the PRF used. For example, PBKDF2 using HMAC-SHA1 would be denoted as PBKDF2-HMAC-SHA1.

In SHA256, attacker is able to attack rainbow table attack. But, In PBKDF2 there is also another input parameter called the salt. Salt is securely generated binary string. (Commonly 64-128 bits long). So, if an attacker tries to rainbow table attack then attacker needs so many memory and so much times. (minimum number of possible salts = 2^{64})

In PBKDF2, there is another important parameter – C (the number of iterations). As C increase the algorithm takes more time to compute the hash.

PBKDF2 can be defined as follow:

DK = PBKDF2 (PRF, Password, Salt, C, dkLen)

Where,

- dkLen:- dkLen is the length of the desired output key.
- DK:- DK is the output key

IV. Implimention Details and Results

All the code did for this project in the Google Colab platform. In Google Colab first, install NVIDIA CUDA which provides libraries to run CUDA code. Codes are uploaded on Github. (https://github.com/Kishan-143/Cryptographic-Algorithms).

Function time defined as the only calculation time for function. It's not including the memory allocation and initialization time. Elapsed time defined as the total time covering by the program which also includes file read/ write, memory allocation, and initialization. Avg. function time is defined as the average function time for single task. (Avg. function time = Total function time/Number of tasks). Avg. elapsed time is defined as the average elapsed time for single task.

CPU implementation of SHA256

For the CPU implementation, we use the OpenSSL library which provides the predefined function for the SHA256. (SHA256_Init(), SHA256_Update(), SHA256_Final()).

In project folder there is SHA256_kernel file in which, we implemented sha256() function using predefined library to get hash of the password and measure time to encrypt single password. This takes around 69 microseconds to calculate single hash password.

(https://github.com/Kishan-143/Cryptographic-Algorithms/blob/master/SHA256_kernel.cu)

```
Password: “sri”
Hash:“d5e96656c6f455d2b0d8da4930a4c744cd86b4e6d2915de179d1f92f49316a9”
Total elapsed time: 68980 microsecond
```

For the brute force attack, the above method is very trivial. We have to check manually for each password. Now, we implemented SHA256_file, in which you give input as the list of passwords and as result you get the list of encryption hash corresponding to each password (in the file). In the output, we get file result_sha_cpu.txt which contains list of a hash of password and file result_sha_cpu_details.txt which contains also a list of a hash of passwords with some more details. In the input give filename (input.txt) and in output you will get result_sha_cpu.txt and result_sha_cpu_details.txt.

```
Input: input.txt (file)
Number of tasks: 230450
Total function time: 56634 microseconds
Avg. function time: 0.245 microseconds
Total elapsed time: 131569 microseconds
Avg. elapsed time: 0.570 microsecond
```

In the second implementation, we can encrypt a single password with 0.245 microseconds. Here, we only consider functional time not considering the write time in the file. In one second someone can generate around 42,00,000 hash. So, if the password is only 32 bits long (4 byte character) and if we consider all possibilities then we can crack the password within 17 minutes in my i3-core CPU. Nowadays, hackers have very fast computers with multiprocessors and GPUs. So, it is better to use at least 8 byte long passwords. (if software is encrypted with SHA256).

GPU implementation of SHA256

For the GPU implementation, we have to write the functions of our devices because OpenSSL only provides host functions. In SHA256_gpu, the Cuda implementation of SHA256 is given.

```
Input: input.txt (file)
Number of tasks: 230450
Total function time: 56 microseconds
Avg. function time: 0.000243 microseconds
Total elapsed time: 9234370 microseconds
Avg. elapsed time: 40 microseconds
```

Here, the result is very unusually compare to the CPU implementation. Avg. functional time for GPU implementation of SHA256 is only 0.000243 microseconds while Avg. CPU implementation takes 0.245 microseconds. While Avg. elapsed time for GPU is 40 microseconds while Avg. elapsed time for CPU is only 0.570 microseconds.

In functional time we not consider memory allocation and initialization time. As we saw in the algorithm, an implementation needs to do lots of computations. In comparison to CPU, there are lots of ALUs which basically designed for the computation in GPUs. Multiple threads work on different ALUs. So, function time in GPU is very less than the CPU. In the CUDA programming model, first, we have to transfer data from host to device. Transferring large input from Host memory to Device memory requires much more time. Also, Transferring Device output to the host output also requires a long time. So, the total elapsed time in GPU is much more than the CPU. But, we can overcome this problem by using an integrated GPU.

CPU implementation of PBKDF2

For the CPU implementation of PBKDF2, we use OpenSSL library which provide predefined functions.

In project folder there is PBKDF2_kernel file in which, we implemented pbkdf2 using predefined library to get hash of the password and measure time to encrypt single password. This takes around 2 milliseconds to calculate single hash password.

```
Password: password
Iterations: 4096
Salt: 73616c74
Hash: 4b007901b765489abead49d926f721d065a429c1
Total elapsed time: 2006 microseconds
```

Now, we implemented PBKDF2_file, in which you give input as the list of passwords and as result you get the list of encryption hash corresponding to each password (in the file). In the output, we get file result_pbkdf2_cpu.txt which contains list of a hash of password.

Here, Avg. functional time is 3754 microseconds where as avg. elapsed time is 3755 microseconds. In PBKDF2, the computations time is very much less than the memory access time. So, implementation of PBKDF2 instead of SHA256 is very efficient on GPU.

```
Input: input1.txt
Number of tasks: 3706
Avg. functional time: 3754 microseconds
Total functional time: 1391 milliseconds
Avg. elapsed time: 3755 microseconds
Total elapsed time: 1392 micros
```

References

https://jhui.github.io/2017/03/06/CUDA/
https://en.wikipedia.org/wiki/SHA-2
http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf
https://medium.com/@smallfishbigsea/basic-concepts-in-gpu-computing-3388710e9239
https://developer.nvidia.com/
http://courses.cms.caltech.edu/cs179/2015_lectures/cs179_2015_lec05.pdf
https://www.movable-type.co.uk/scripts/sha256.html
https://en.wikipedia.org/wiki/PBKDF2
https://www.ce.jhu.edu/dalrymple/classes/602/Class13.pdf