IT-477 Introduction to GPU programming
Project Report

SHA message digest computation on GPU

by

Nachiket Trivedi   201401047
Maulik Trapasiya 201401051

Supervisor: Prof. Bhaskar Chaudhary

1 Introduction

Message digest algorithms (hash functions) are used on regular basis in cryptography. There are many applications of these functions, such as ciphering passwords, creating checksums for large data sets, providing message and file integrity, secure login, and authentication. When hash functions are invoked on long messages, it is very important for hash algorithm to be extremely fast. SHA(Secure Hash Algorithm-1) is one such cryptographic hash function which converts a message into digests of 160 bits. In this project we use CUDA programming to make it more fast. We intend to increase its speed in this project by not compromising security.

One way to improve the speed of hash function is to run it in parallel. That is to execute the code of the algorithm among multiple processes. This can be accomplished on CPU by using threads. However, since CPU is used by many other applications it might not be the most efficient way to run the algorithm on it. Recently GPUs are becoming more powerful than CPUs and general computing on GPU is becoming more popular.

The aim of the project is to implement SHA-1 Message Digest algorithm on GPU using CUDA and make performance comparisons of these algorithms between CPU and GPU versions. This project will give me better understanding of hash function and how they work.

Since all of the SHA family hash functions are iterative they are not used in a parallelized way, or it is very hard to parallelize them. Therefore, one workaround will be used to see if it is possible to use the power of GPU to speed up these algorithms.

2 Hash functions (Merkle–Damgård construction method)

Hash functions have become a salient part of today's security and cryptographic scenario. Most of the prominent hash functions including MD5 and the SHA family, are based on the Merkle–Damgård construction method. The Merkle–Damgård construction or Merkle–Damgård hash function is a method of building collision-resistant cryptographic hash functions from collision-resistant one-way compression functions.

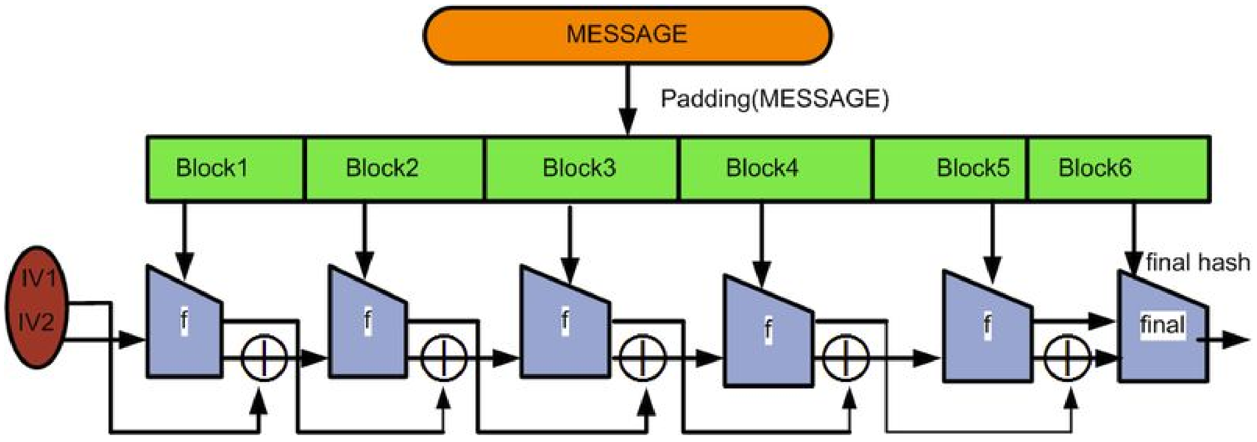


Fig 1: Hash computation using Merkle–Damgård method.

The hash function takes the message as input and gives a digest of a fixed length as the output(eg: 128 bits, 160 bits etc.). First of all, before the message is given to the compression function, it is converted into an input which is multiple of a pre-defined by block size. This conversion is done by padding along with appending the message length. As shown in the figure, the input is then divided into blocks of fixed size and given to the main part of the algorithm. The MD based construction is an iterative process as the output of first algorithm acts as initialization input of the next one. The initialization vectors(shown in figure as IV) are given input to the first step, and are continuously replaced till the end. The last state of these vectors serves as the digest of the message.The main part is done by the compression function f which takes IV and the block as input and after a fixed number of rounds gives the output the same size as the IV. Now, depending on the algorithm, the block size, the compression function f and the number of rounds change. By the end of the algorithm, we get the message digest.

3 SHA-1 message digest algorithm

The SHA-1 message digest algorithm works on the Merkle–Damgård principle as mentioned above. Here, the message is first converted into a multiple of 512 bits by padding with 100….0 and message length. The converted message is then divided into blocks of 512 bits length. The first padding is applied on the last block till the length of that block is 448 bits. The remaining 64 bits are left for appending message length. This done to prevent different messages having same output after the first padding. Now, these blocks are sequentially given to a compression function which takes 5 buffer variables of 32 bits each as input shown in the figure as IV. These functions(f) also take the 512 bits block as input and gives a 160 bit output which acts as the input for the next step.

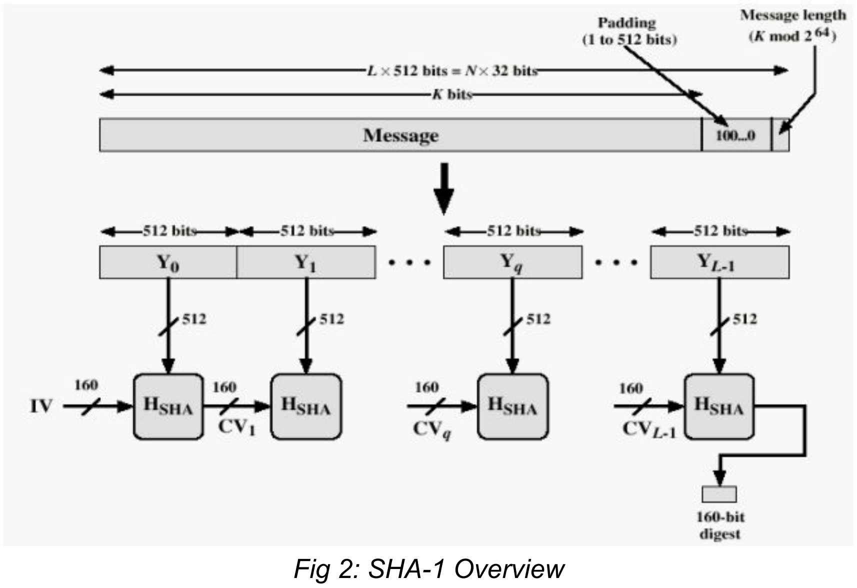


Fig 2: SHA-1 Overview

Inside the function f, there are four steps with 20 rounds each thereby making a total of 80 rounds. The 512-bit block is the 16 32-bit sub-blocks are expanded to 80 32-bit sub-blocks, on sub-block for each round. The 160 bit buffer vectors represented as A,B,C,D,E in the Fig-3 operates with the 32-bit sub-block to form the 160 bit output. This output replaces the buffer vectors which in turn acts as input for next step. This goes on for 80 rounds until the final output is obtained. As shown in the figure, this output is added with the initial vectors to obtain the final 160-bit digest.

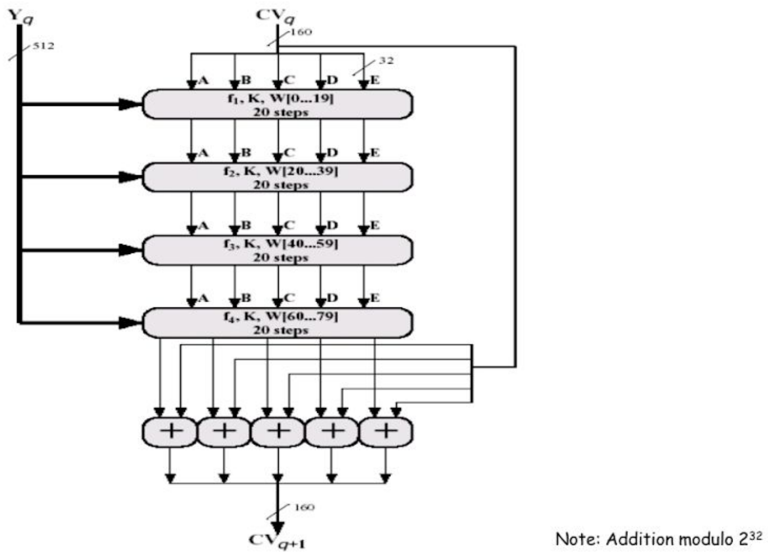


Fig 3: SHA compression function on a block

4 Parallelization

Hardware Specifications

All performance test will be executed on 64-bit Linux machine with the following technical parameters:

- **CPU Specs:**

model name : Intel(R) Xeon(R) CPU E5-2640 v2 @2.00GHz

cpu cores : 8

- **GPU Specs:**

Major revision number: 3

Minor revision number: 5

Name: Tesla K40c

Total global memory: 3489202176

Total shared memory per block: 49152

Total registers per block: 65536

Warp size: 32

Maximum memory pitch: 2147483647

Maximum threads per block: 1024

Maximum dimension 0 of block: 1024

Maximum dimension 1 of block: 1024

Maximum dimension 2 of block: 64

Maximum dimension 0 of grid: 2147483647

Maximum dimension 1 of grid: 65535

Maximum dimension 2 of grid: 65535

Clock rate: 875500

Total constant memory: 65536

Texture alignment: 512

Concurrent copy and execution: Yes

Number of multiprocessors: 15

Kernel execution timeout: No

GPU implementation of SHA-1

As discussed before, SHA-1 is an avalanche algorithm where the output of one round acts as an input for the next round. This property poses as a great challenge while parallelizing. If we look closely, the compression function on each 512-bit block is actually made of 2 functions: one is expansion of 16 32-bit sub-blocks to 80 32-bit sub-blocks and the other is compression to form 160 bit output. The former function does not depend on any pre-computed value and only takes the 512-bit block as input. We used this property to our advantage to parallelize the algorithm. To do this, we invoked the kernel where each thread will compute the expansion function of each block. After the parallel computation of expansion function is completed, it is necessary to sync them for our next step.

For syncing these parallel threads, we used __syncthreads() function. As the syncthreads() function only works within a block, we can’t have threads running parallely in two different blocks. This forced us to only 1 block per grid. The GPU of the server can accommodate 1024 threads per block and hence only 1024 threads can run at a time parallely. This means that only 1024 segments of 512-bits can be processed for expansion at a given time. If the input message length is greater than 1024*512=524288 bits, the kernel is re-launched and the same process is repeated again and again till all the sub-blocks are processed and synced.

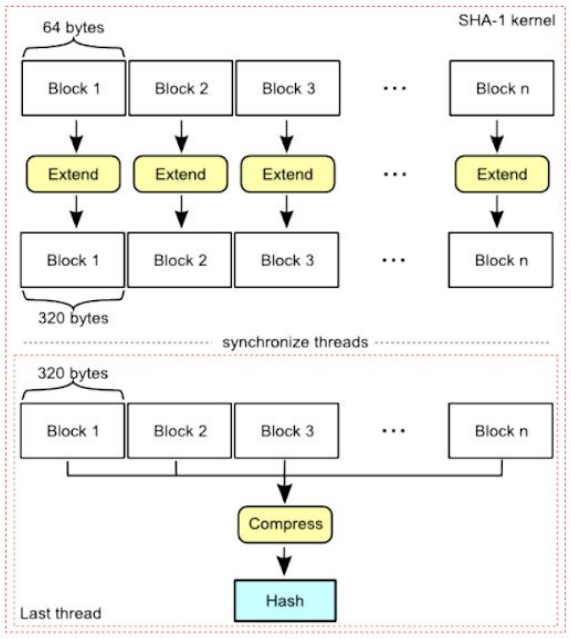


Fig 4: Parallelization of SHA-1

After the completion of the expansion function, the next step is the compression function, which is an eighty-round serial process which processes the 5 32-bit buffers(specified as A,B..,E in Fig 3), along with one 32 bit sub-block of the recently computed 80-bit sub-blocks. For doing this step, we used a single thread which serially computes 1024 segments (where each segment is the 512 bit block). The output of this thread will be stored in buffer values which were initially designated as A,B,..E in Fig 3. Let's say the kernel is re-launched for parallel computation of expansion function of the remaining segments, the whole process including the serial one is repeated again, as the parameter given to each kernel invoke contains the buffer values, which in this case is the output of the last kernel launch. This whole process goes on in loop until all the segments are processed. The final output is added with the initial buffer values, and we have the digest.

6 Performance Analysis

Thread per block : 1028

	SIZE	KERNEL	cudaMemcpy	cudaMalloc	cudaFree
CPU	1000	0.012			
GPU	1000	0.014	0.67	0.264	0.193
CPU	10000	0.101			
GPU	10000	0.015	5.114	0.406	0.269
CPU	100000	1.006			
GPU	100000	0.014	1.119	0.382	0.267
CPU	1000000	10.011			
GPU	1000000	0.1	488.700012	0.39	0.28
CPU	10000000	100.711998			
GPU	10000000	0.842	4885.374023	0.449	0.25
CPU	100000000	813.338989			
GPU	100000000	16178.08496	32745.14648	0.58	0.361

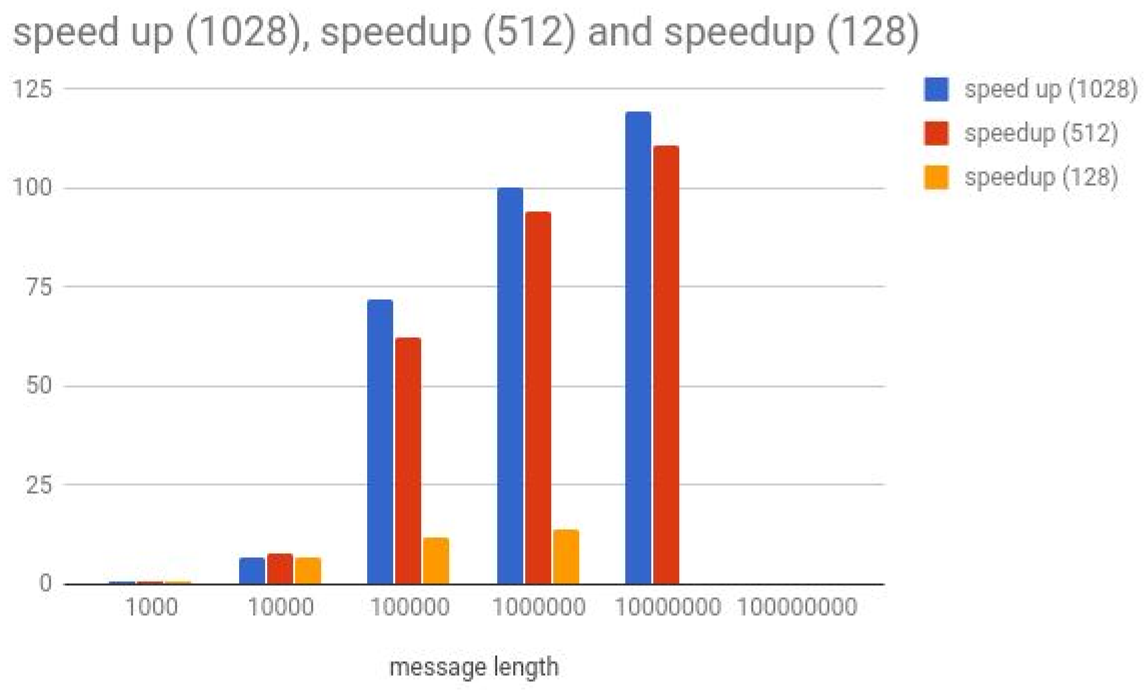
Thread per block : 512

	SIZE	KERNEL	cudaMemcpy	cudaMalloc	cudaFree
CPU	1000	0.01			
GPU	1000	0.013	0.653	0.213	0.202
CPU	10000	0.088			
GPU	10000	0.011	5.036	0.327	0.224
CPU	100000	0.872			
GPU	100000	0.014	1.072	0.347	0.239
CPU	1000000	8.487			
GPU	1000000	0.09	482.921997	0.331	0.347
CPU	10000000	85.875			
GPU	10000000	0.775	4923.70166	0.46	0.297
CPU	100000000	827.369995			
GPU	100000000	16880.20117	34043.53125	0.543	0.334

Thread per block 128:

	SIZE	KERNEL	cudaMemcpy	cudaMalloc	cudaFree
CPU	1000	0.012			
GPU	1000	0.016	0.66	0.287	0.194
CPU	10000	0.101			
GPU	10000	0.015	0.28	0.357	0.283
CPU	100000	0.999			
GPU	100000	0.083	50.091999	0.248	0.191
CPU	1000000	9.989			
GPU	1000000	0.725	498.389984	0.377	0.267
CPU	10000000	100.814003			
GPU	10000000	816.843994	4172.336914	0.402	0.239
CPU	100000000	812.926025			
GPU	100000000	46833.31641	4208.689941	0.461	0.381

Speed up :



message length	speed up (1028)	speedup (512)	speedup (128)
1000	0.8571428571	0.7692307692	0.75
10000	6.733333333	8	6.733333333
100000	71.85714286	62.28571429	12.03614458
1000000	100.11	94.3	13.77793103
10000000	119.6104489	110.8064516	0.1234189193
100000000	0.05027412027	0.04901422599	0.01735785734

We can clearly notice that GPU version of algorithm is much faster when input size is small. When we need to launch more than one kernel, GPU version performance decreases as next kernel lunch has to wait till previous kernel launch calculate its digest value and furthermore lots of system time is consumed by thread synchronization.

Each thread is converting 16 32 bit sub-block into 80 32 bit sub-block. here, only one block launch per grid is possible. So as we increase maximum_thread_per_block GPU performance increases. (GPU that we are using supports 1024 threads per block)

Another factor that we must consider is data exchange between host and device. Time spent on data transfer from host to device increases as data size increases. This is because of hardware limitation.

7 Conclusion

As seen in the results, when the GPU is used at its fullest of 1024 threads per block, we see significantly better performance. As for higher inputs, the GPU implementation doesn't work well as compared to CPU. This is because, due to high data, multiple kernel invokes take place. Thus, the total time spent on kernel along with the time spent on frequent memory allocation and memory copy increases. As seen in the analysis above, after 10^7 this change is visible.

This parallel implementation of SHA-1 works significantly well for not-so-large inputs. As SHA-1 is a hash function used primarily for signature based authentication and providing message integrity, security is a primary motive along with speed. Right now in market, SHA-1 is more secure than its competitor MD5 but MD5 is faster than SHA. Given the primary motive of this algorithm is to provide authentication and integrity, seldom arises a case of large inputs, as the messages for digital signatures of passwords are kept generally small. Thus, as our GPU implementation of SHA-1 gives significantly fast results for market-specific inputs in addition to non-compromisation of security, this can prove a better choice as compared to MD5.

For future improvements, we can use a technique to sync threads across the blocks. As we know, __syncthreads() only synchronizes the threads within a block, causing multiple kernel invokes. If we can find a technique that sync all the threads inside of a grid regardless of blocks, the limitation of one block per kernel is removed, making this algorithm significantly faster even for very large inputs. Another improvement can be the use of SHA-2 for parallelization. The implementation of SHA-2 is different and more complex than SHA-1, and is subject to complete parallelization by using binary tree for processors.

References

[1] Federal Information Processing Standards Publication 180-2,
http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[2] CUDA Programming Guide ,
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[3] Hash Project website
http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[4] Official NIST
https://csrc.nist.gov/projects/hash-functions

[5] RFC3174 sample SHA implementation
https://tools.ietf.org/html/rfc3174