Stevens Institute of Technology
Department of Computer Science

Project report

SHA message digest computation on GPU

by

Tadas Vilkeliskis

Supervisor: prof. Sven Dietrich

Hoboken, 2008

Contents

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 files and checking their integrity. When hash functions are invoked on long messages, it is very important for hash algorithm to be extremely fast.

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.

2 Project summary

The aim of the project is to implement SHA-1 [1] and PARSHA-256 [2] Message Digest algorithms on GPU using CUDA [4] 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. On the other hand, PARSHA-256 [2] is specially designed to be run among multiple processes and we will see how much faster it is against multi-threaded implementation of PARSHA-256 on CPU.

3 Hash functions

Most of the hash functions available today are based on Merkle-Damgård construction method [3].

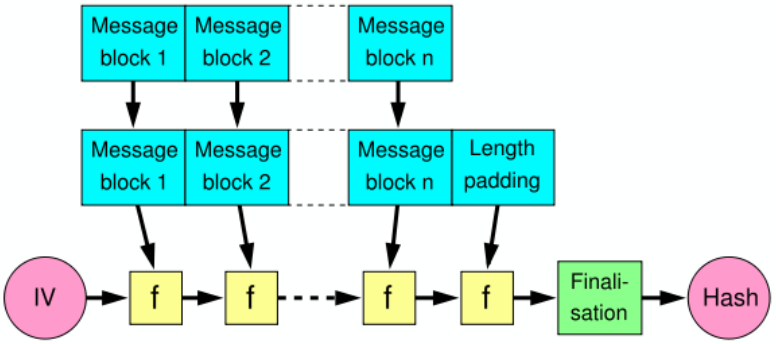

Figure 1: Hash computation using Merkle-Damgård method.

The input message is split into blocks of particular size and additional padding block is added to the end of the message. After message was prepared it is given to the hash algorithm to produce hash of the message. Hash algorithm basically has three states: initialization, compression and finalization. In the initialization state starting values of the hash are set from the initialization vector IV. Then, each message block is compressed using compression function f. To make hash algorithm stronger against collisions, compression function f uses intermediate hash value returned by the previous compression function. Hence, intermediate hash value for each message block depends on the intermediate hash value of the previous block. Finally, in finalization state, for security purposes, the memory used in partial hash computation is cleared and final hash value is returned to user.

4 SHA message digest algorithms

In this section I will shortly cover the specifications of SHA-1, SHA-256 and PARSHA-256.

4.1 SHA-1

SHA-1 hash function returns 160 bit hash value for given message and it is based on Merkle-Damgård construction method.

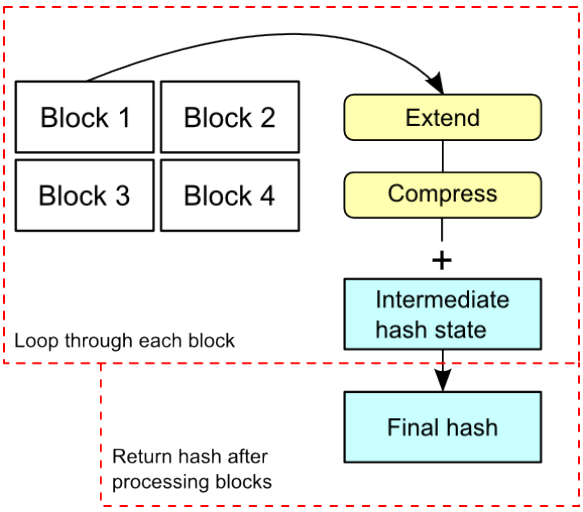

Figure 2: Computing hash value with SHA-1 of four block message.

The message is padded to be multiple of 512 bits and it is split in 512-bit blocks (sixteen 32-bit words). Initial hash state is set. Later each block is processed in such way:

1. Sixteen 32-bit words are extended into eighty 32-bit words.
2. Eighty 32-bit words are compressed to 160 bits (five 32-bit words).
3. Intermediate hash state is updated.

After all blocks have been processed intermediate hash is returned to the user as a value for the final hash.

4.2 SHA-256 and PARSHA-256

SHA-256 hash function works almost in the same way as SHA-1 hash function. The main differences are longer hash value (256 bits) also compression and extension functions are implemented differently than in SHA-1, finally 512-bit block is extended only into sixty-four 32-bit words.

PARSHA-256, on the other hand, is a new function which uses SHA-256 compression and extension functions to obtain hash value of the message. Everything else including message padding, block processing is accomplished differently than in SHA-256.

The algorithm utilizes binary tree of processors to compute hash value of the given message. Where each processor takes one block of a message and computes the hash value for that block. In next parallel rounds parent processors takes a combination of hash values from their children and input message. The process is repeated until all message is hashed.

5 Methodology

5.1 Hardware

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

- Intel(R) Core(TM)2 Quad CPU Q9300 @ 2.50GHz
- NVIDIA(R) Quadro FX 570 GPU

5.2 SHA-1 implementation on GPU

In Section 4.1 it was mentioned that SHA-1 hash function is based on Merkle-Damgård construction method. This is a huge obstacle when you want to implement SHA-1 in a multi-threaded way. From the Figure 2 we can observe that there are only two steps in the algorithm that are used to create hash value. The compression step depends on the extension step. However, extension step does not depend on anything but the input message. Therefore, we can run code of extension function among multiple threads and later use single thread to invoke compression function on extended data. This method is illustrated in Figure 3.

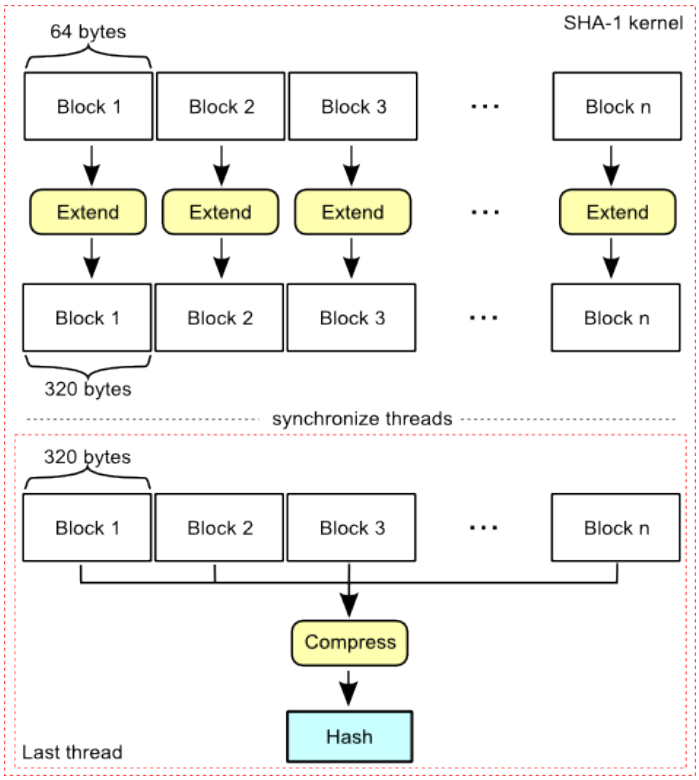

Figure 3: Semi-parallel implementation of SHA-1.

My implementation of the method defined earlier uses two data buffers on GPU. One is used to hold a copy of input message that is available on CPU. And another one is big enough to hold as much extended blocks of the message as there are threads in the execution grid.

There is one drawback with GPU in this particular case. That is a number of threads allowed per thread block. The number of threads depend on resources consumed per thread and also is limited by hardware itself. In my implementation one thread uses 60 registers and total number of registers available on Quadro FX 570 GPU in one thread block is 8192. Thus, I could successfully utilize only

128 threads per block. There is also another limitation by GPU. Since I have to synchronize threads so that the last thread could correctly compute hash value, I can only use one thread block because GPU does not allow thread synchronization among multiple blocks. Because of reasons mentioned above, the SHA-1 kernel is called multiple times in order to process longer message.

Concerning these limitations in some cases when input message is large the SHA-1 kernel is executed multiple times.

5.3 PARSHA-256 implementation on GPU

The PARSHA-256 GPU code that comes with the project is based on multi-threaded implementation of PARSHA-256 [5] which uses POSIX threads.

Binary tree of input blocks to the processors is represented as an array. Where each i^{th} element of the array has children data in $2i$ and $2i + 1$ elements.

My implementation uses two data buffers on GPU. One is used as an input to the kernel. While second one is used for output from the processors. The size of memory used depends on the height of the tree used in the computations. I have implemented tree of height sixteen. Such tree in the worst case will consume $2^{16} \cdot 96 + 2^{16} \cdot 32$ bytes of memory for storing only input data and intermediate hash states.

One thread uses 50 registers and maximum number of threads available in the thread block is 128. Each thread represents a processor in the processor tree.

The process for computing hash of the given message is split in four stages: initialization, first round, steady state and flushing. In the initialization stage the parameters crucial for algorithm execution are set. First round is used to prepare the algorithm for steady state. Basically, the first round returns intermediate hashes that were computed by every processor in the tree. In the steady state each available processor in the tree uses a combination of intermediate hash states provided by its children and input message to compute subsequent intermediate hashes while leaf processors use combination of input message and initialization vector. This stage is executed only when the values of certain parameters defined in initialization stage are satisfiable. In the next stage same steps are repeated as in previous stage, the only difference is that number of processors is eventually decreasing (leaf processors are becoming dead processors). Thus PARSHA-256 kernel is executed multiple times with different parameters.

The designers of PARSHA-256 give us three options to implement their algorithm. We can use IV of different lengths: 0, 128 and 256 bits. My implementation allows only IV of 256 bits.

NVIDIA's NVCC compiler has one great feature. The code that has to be executed on GPU can be compiled with device emulation mode turned on and therefore executed on CPU. I am using the same GPU implementation to do both CPU and GPU performance test of PARSHA-256.

6 Performance improvements

Project includes two algorithm benchmarking programs to test performance of both SHA-1 and PARSHA-256. Whether performance improvement goals were met or not depends on the perspective you are looking from: either from CPU or GPU.

Table 1 shows the results of performance tests for SHA-1 algorithm. If we look only at time spent on hash computation ("Kernel" column) we can notice that GPU version is much faster than CPU version when input data is reasonably small. When we reach threshold of 10MB the amount of time spent on hash computation on GPU increases significantly. This again, seems to be reasonable result, since the program is using only limited amount of threads and when size of input increases the number of kernel calls also increases and furthermore lots of system time is consumed by thread synchronization.

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 can be result of bus (AGP or PCI bus) used by GPU and front-side bus which is used for data exchange with CPU. Also system calls are significantly slower than user space calls.

Unit	Size, bytes	Kernel, ms	cudaMemcpy, ms	cudaMalloc, ms	cudaFree, ms
CPU	1000	0.012000			
GPU	1000	0.005000	1.664000	0.104000	0.095000
CPU	10000	0.104000			
GPU	10000	0.005000	0.322000	0.189000	0.186000
CPU	100000	1.036000			
GPU	100000	0.045000	141.783005	0.186000	0.190000
CPU	1000000	10.611000			
GPU	1000000	0.383000	1415.932007	0.238000	0.233000
CPU	10000000	106.547997			
GPU	10000000	12372.243164	1787.130981	0.495000	0.434000

Table 1: Benchmark test of SHA-1

Unit	Size, bytes	Kernel, ms	cudaMemcpy, ms	cudaMalloc, ms	cudaFree, ms
CPU	1000	25.032000	0.026000	2.565000	0.017000
GPU	1000	0.283000	2.404000	143.300003	0.010000
CPU	10000	42.157997	0.040000	0.004000	0.002000
GPU	10000	0.041000	10.058999	0.110000	0.095000
CPU	100000	71.198006	0.355000	0.022000	0.024000
GPU	100000	0.054000	76.107979	0.207000	0.187000
CPU	1000000	322.458954	3.275999	0.024000	0.118000
GPU	1000000	0.086000	731.846802	0.223000	0.225000
CPU	10000000	1992.895874	30.029993	0.011000	1.038000
GPU	10000000	0.148000	7337.381348	0.375000	0.421000
CPU	100000000	17592.455078	259.789185	0.012000	0.005000
GPU	100000000	0.668000	73513.625000	0.377000	0.410000

Table 2: Benchmark test of PARSHA-256

Table 2 lists results of performance tests for PARSHA-256 algorithm. CPU row in the table represents results of PARSHA-256 GPU version compiled in device emulation mode. As we can see from the results the GPU version is significantly faster no matter what input size we use. The highest test vector used in PARSHA-256 performance test was roughly 100MB in size. Such input size forced to use processor tree of height 16, which utilizes 2^{16} threads and 4MB of data is parsed in first and few other parallel rounds. Here again, most of the time is spent on data exchange between host and device. PARSHA-256 implementation spends even more time in data exchange than in implementation of SHA-1. Since more memory copy system calls are called in PARSHA-256 implementation the more total time is spent in exchanging data.

Lets look at PARSHA-256 performance on CPU. Because CPU row represents GPU code compiled in device emulation mode the memory copy, allocation and deallocation columns can be omitted. The results of performance tests in emulation mode is something that I have been expecting to be. The thread blocks on GPU are executed concurrently on separate multiprocessors [6]. Four multiprocessors are available in Quadro FX 570 GPU and it means that four thread blocks are executed on a real hardware while the rest are waiting in the queue for their turn, and furthermore each thread block equips 128 threads. CPU in our case has only four processors. I am not very familiar with CPU architecture that was used in test, however I think that this was the reason why GPU outperformed CPU. Also I have no good reasoning why the performance of benchmark program was so bad when it was run in emulation mode. Here again we should take many factors into account that could reduce performance: number of threads allowed per one process, operating system (threads are implemented differently in Linux and Windows), how NVCC optimizes code when it is compiled with emulation mode turned on, and whether all of the CPU cores were used when multiple threads were executed.

7 Conclusion

As seen from the results, the semi-parallel implementation of SHA-1 provided with the project should not be used with large input. This is because the time spent on both hash computation and data exchange between host and device increases significantly. On the other hand, PARSHA-256 performs very well on GPU even when large message is given as an input.

By my understanding both GPU implementations of hash algorithms have same performance bottleneck - copying memory from host to device and vice versa. Algorithm performance depend on how memory management is implemented. Better memory management should make perform these algorithms better. Another way to increase performance of SHA-1 could be a search for parallelism in other places, such as extension or compression functions. Also, one could modify my implementation and use shared memory that I was not using in my code because I lacked some knowledge.

GPU should be a great tool when recovery of hashed passwords (or other short messages) is needed. In this case it would be possible to run thousands of threads to search for one particular hash. And results have shown that hash computation on GPU is faster than on CPU.

References

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

[2] PARSHA-256 - A New Parallelizable Hash Function and a Multithreaded Implementation,
`http://www.iacr.org/archive/fse2003/28870366/28870366.ps`

[3] Merkle-Damgård Revisited : how to Construct a Hash Function,
`http://cs.nyu.edu/~puniya/papers/merkle.pdf`

[4] Download CUDA,
`http://www.nvidia.com/object/cuda_get.html`

[5] Multi-Threaded implementation of PARSHA-256,
`http://www.isical.ac.in/~crg/software/parsha256.c`

[6] CUDA Programming Guide 2.0,
`http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`