SCIENCE CHINA
Information Sciences

• **Supplementary File** •

Hashing multiple messages with SM3 on GPU platforms

Shuzhou SUN[1,2], Rui ZHANG[1,2*] & Hui MA[1]

[1]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing* 100093*, China;*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, Beijing* 100049*, China*

Appendix A Related Work

We briefly review some known implementations of the SM3 hash algorithm. The internet draft [6] presented a reference software implementation for verifying the correctness. The first hardware implementation on FPGA [4] was accomplished by Yuan Ma et al., which proposed several optimizations and achieved a high throughput with low area. Concretely, their optimized implementations achieved 2.7 Gbps which is 1.68× improvement compared with a standard implementation. Later work [3] investigated compact implementations on resource-constrained security chips and presented reasonable throughput. The work [5] implemented SM3 on financial IC cards and showed that their design was correct and feasible. The peak reported throughput of the SM3 hash algorithm is owned to [7], which can achieve 6.54 Gbps data rate on ASIC platform (SMIC 65nm CMOS). They proposed a 4-round-in-1 structure to reduce the number of rounds, and a logical simplifying to move 3 adders and 3 XOR gates from critical path to the non-critical path. Recently, the designers of the SM3 hash algorithm summarized the design, properties, software and hardware implementations and cryptanalysis in [2] with x86, ASIC and FPGA platforms. There are also several commercial products (e.g., cryptographic chips) that support the SM3 hash algorithm. Targeting at different scenarios, these products vary in architecture and performance, e.g.,1392 Mbps on a 200MHz processor[1)] and 1.5 Gbps on a 600MHz processor[2)]. To the authors' knowledge, most of these products are slower than [7] when considering throughput.

We note that utilizing the computing power of dedicated hardwares to accelerate cryptographic algorithms has been intensively studied for years. The Advanced Encryption Standard (AES) has been parallelized on various platforms, e.g., FPGA [8–10] and GPU [11–15], which continuously improved the practical performance of AES. The well-known RSA was parallelized on GPUs [16, 17] and FPGA [18] and achieved a significantly high throughput. All second-round SHA-3 candidates, including Keccak-256, are evaluated in [28] on two exotic platforms: the Cell Broadband Engine (Cell) and the NVIDIA GPUs. Meanwhile, the work [28] set a performance record of Keccak-256 on GPUs, thus we compare our work with it. There also exist numerous parallel designs in Elliptic Curve Cryptography (ECC), e.g., point multiplication on FPGA [19] and ECDSA on GPU [20]. Recently, post-quantum cryptography receives a lot of attentions also their fast parallel implementations, e.g., NTRU based signature scheme on GPU [21] and isogeny-based key exchange on FPGA [22, 23]. On the other hand, a wide range of cryptanalytic problems were solved by using parallel collision search [24], for instance, the recent SHA-1 attack on multi-core CPUs and GPUs [1].

Appendix B Preliminary

We outline notations, the SM3 hash algorithm and the CUDA framework.

Appendix B.1 Notation

Let $[k]$ denote the integer set $[1, k]$. Let $||, \oplus, \wedge, \vee$ and $\neg$ denote bitwise operations concatenation, XOR, AND, OR and NOT, respectively. Let $\lll$ denote 32-bit cyclic left rotation. Let a word denote a 32-bit integer. Otherwise specified, all additions are computed in modulo 2^{32}.

* Corresponding author (email: r-zhang@iie.ac.cn)
1) http://www.china-core-tj.com/content/show.asp?m=1&d=347 (in Chinese)
2) http://www.trustsoc.com/index.php/Home/Product/view/id/6.html (in Chinese)

Appendix B.2 Review of SM3

We only give a brief review of SM3 here and refer to [2, 6] for detailed specifications and design considerations. The SM3 cryptographic hash algorithm takes input of a message m of length l ($l < 2^{64}$), and after padding and iterative compression, generates a hash value of 256 bits long.

Let X, Y and Z be 32-bit words. SM3 has a few constants and functions. Equation (B1) lists 64 addition constants, which are used to provide randomness and reduce the linearity and probability of differential inheritance [25].

$$T_j = \begin{cases} 79cc4519, & 0 \leqslant j \leqslant 15, \\ 7a879d8a, & 16 \leqslant j \leqslant 63. \end{cases} \tag{B1}$$

Meanwhile, there are two permutation functions P_0 and P_1:

$$\begin{aligned} P_0(X) &= X \oplus (X \lll 9) \oplus (X \lll 17), \\ P_1(X) &= X \oplus (X \lll 15) \oplus (X \lll 23). \end{aligned} \tag{B2}$$

Lastly, to guard against bit-tracing cryptanalysis techniques, improve the nonlinearity and reduce differential image characteristics, SM3 has following boolean functions:

$$FF_j(X,Y,Z) = \begin{cases} X \oplus Y \oplus Z, & 0 \leqslant j \leqslant 15, \\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z), & 16 \leqslant j \leqslant 63. \end{cases} \tag{B3}$$

$$GG_j(X,Y,Z) = \begin{cases} X \oplus Y \oplus Z, & 0 \leqslant j \leqslant 15, \\ (X \wedge Y) \vee ((\neg X) \wedge Z), & 16 \leqslant j \leqslant 63. \end{cases} \tag{B4}$$

Message Padding and Parsing. On input a bit message m of length l, the SM3 algorithm appends following items to make the length of padded message m' a multiple of 512: 1) a single bit string "1", 2) a k-bit all zero string, where k is the smallest non-negative integer that satisfies $l + 1 + k \equiv 448 \pmod{512}$, and 3) the 64-bit binary representation of length l. Then the padded message m' is divided to n 512-bit blocks, denoted $B_1, B_2, \ldots, B_n$, where $n = \frac{l+k+65}{512}$.

Message Expansion. Each 512-bit block B_i ($i \in [n]$) is expanded to 132 words $W_0, \ldots, W_{67}, W'_0, \ldots, W'_{63}$. First, a block is viewed as sixteen 32-bit words denoted $W_0, \ldots, W_{15}$. Then other words are constructed as following equations:

$$\begin{aligned} W_j &= P_1(W_{j-16} \oplus W_{j-9} \oplus (W_{j-3} \lll 15)) \oplus (W_{j-13} \lll 7) \oplus W_{j-6}, 16 \leqslant j \leqslant 67, \\ W'_j &= W_j \oplus W_{j+4}, \qquad\qquad 0 \leqslant j \leqslant 63. \end{aligned} \tag{B5}$$

Message Compression. During this procedure, the working state contains eight 32-bit variables A, B, C, D, E, F, G, H which is initialized as $V_0^{(0)}, \ldots, V_7^{(0)}$ (see [6] for their concrete values). The words W and W' are compressed with 64 iterations of Equation B6, where $0 \leqslant j \leqslant 63$.

$$\begin{aligned} SS1 &= ((A \lll 12) + E + (T_j \lll j)) \lll 7 \\ SS2 &= SS1 \oplus (A \lll 12) \\ TT1 &= FF_j(A,B,C) + D + SS2 + W'_j \\ TT2 &= GG_j(E,F,G) + H + SS1 + W_j \\ D &= C \qquad C = B \lll 9 \\ B &= A \qquad A = TT1 \\ H &= G \qquad G = F \lll 19 \\ F &= E \qquad E = P_0(TT2). \end{aligned} \tag{B6}$$

After compressing one block with 64 iterations, the working state is updated:

$$V_0^{(i)} = A \oplus V_0^{(i-1)}, V_1^{(i)} = B \oplus V_1^{(i-1)}, \ldots, V_7^{(i)} = H \oplus V_7^{(i-1)}. \tag{B7}$$

In the SM3 hash algorithm, this expansion and compression procedure are repeated for each of n blocks. Finally, the hash digest is outputted as a 256-bit string:

$$digest = V_0^{(n)} || V_1^{(n)} || \ldots || V_7^{(n)}. \tag{B8}$$

Appendix B.3 The CUDA Framework

CUDA (Compute Unified Device Architecture) is a parallel computation framework and programming model that uses CUDA-supported GPU devices for general propose computing. It allows developers to solve an amount of tasks on a GPU device with their familiar high-level programming languages, e.g., C/C++, Python. Further, a parallel thread execution (PTX) instruction set is provided to facilitate more flexible optimizations. However, the GPU programming model is different from CPU. A succinct review is presented here and further detailed information can be found in [26, 27].

CUDA programming involves the *host* side (CPUs) and the *device* side (GPUs). GPU devices are worked as coprocessors to the host. In the fashion of *heterogeneous programming*, intensive computations are offloaded to the devices by calling an asynchronous and configurable *kernel* function while the rest serial code executes on the host side.

From the hardware aspect, a GPU is built on several Streaming Multiprocessors (SMs). Each SM is designed to run thousands of threads concurrently in a SIMT (Single-Instruction, Multiple-Thread) fashion. The schedule unit in a SM is a *warp* which contains 32 threads. From the software aspect, a kernel function is partitioned into *grids* of *blocks* of threads. The number of blocks per grid and the number of threads per block are limited by the hardware.

A GPU device has a multi-level memory layout with each memory type varying in size and access latency. Global memory is a sufficiently large off-chip space that can be used to transfer data between the host and the device. It has the highest latency. As pointed out by [26], global memory favors coalesced access pattern, in which memory access will be merged to a single cache line transaction when certain access patterns are met. Local memory is another off-chip space. It has the same latency as global memory. Registers are the fastest storage on GPUs. Though there are thousands of 32-bit registers (typically 65,536) in a SM, they are partitioned among the running threads. As a result, using too much registers in a thread will make registers spill to local memory. Shared memory has smaller latency than global memory and is shared among all threads in the same block. Two additional read-only memory types are constant memory and texture memory. They are both cacheable.

One of the attractive property of the CUDA framework is asynchronous concurrent execution which allows multiple kernels to execute on one GPU device concurrently by using different streams to copy data. The maximum number of resident kernels per device depends on the specific device, e.g., 32 for GeForce GTX 1080. Meanwhile, current GPU devices typically have more than one data copy engines. Therefore, overlapping the data transfer with computation stage is feasible.

Appendix C Optimization Techniques

Appendix C.1 Optimizing the Data Channel

Pipeline Execution. The above procedure is actually a sequential execution between CPUs and GPUs, which has two major issues. The first is the waste of computing power. For instance, when GPUs hash multiple messages, CPUs have nothing to do but wait. The opposite case also holds. Even worse, having one kernel running on a GPU device merely uses part of the device's ability. The second is that all later messages' latency are increased. This is because that later messages can only be processed after computing previous digests. With these considerations, we adopt the pipeline execution technique, which utilize the computing power of both CPUs and GPUs as much as possible. Notice that the GPU kernel invocation is actually an asynchronous call. Therefore, when the messages are offloaded to GPU sides, the CPUs can simultaneously prepare next computation, including accepting more messages (and padding messages). After preparation, if we wait for the previous kernel explicitly, thousands of CPU cycles are still wasted. As introduced in Section Appendix B.3, the GPU devices support loading multiple kernels at the same time. Therefore, after accepting (and padding) the new arrived messages, CPUs can directly invoke the SM3 kernel to compute. Overall, the pipeline technique avoids wasting of resources and decreases the response time.

Building a Thread Pool. In the arbitrary size case, the message padding stage on the CPU side uses multiple threads. During our benchmark, we notice that the overall performance does not stabilize and sometimes decreases. The reason behind this weird behavior is that creating and destroying threads bring additional overhead. To remove this penalty, a thread pool is created and maintained using the OpenMP framework[3)]. Concretely, a group of threads will be created in the initialization step and terminated during a finalization step. This simplifies the control logic for checking for failures in thread creation midway through the application and amortizes the cost of thread management over the entire application.

Appendix C.2 Optimizing the SM3 Kernel

As stated before, a plain implementation on GPU platforms wastes the computing resources and degrades the overall performance. Our implementations follow the *per-thread-per-instance* way. Each thread will compute a SM3 instance on a message. To obtain an efficient SM3 kernel, we present a few general and hardware-specific optimization techniques.

Unrolled Loop. A quite powerful optimization strategy is loop unrolling. It can be done manually by the developer or automatically by the compiler. The goal is to remove the loop overhead, save registers and help the compiler to do branch prediction. Meanwhile, this technique is beneficial to overlap the memory accesses and computations. But sometimes it will bring too much register pressure which means it is not a panacea for all situations. The SM3 kernel has loops in the iterative expansion and compression stage. Each loop in our implementation is tested in both two cases: with and without unrolling. It turns out that an improved performance is obtained when all loops are fully unrolled.

Storing Constants in Macros. The SM3 hash algorithm has a few constants: initialization value $V_0^{(0)}, V_1^{(0)}, \ldots, V_7^{(0)}$ and addition constants T_j $(0 \leqslant j \leqslant 63)$. Typically, storing these values in variables results more memory accesses and thus increases the latency. During message compression (cf. Equation B6), T_j is rotated left. These computation can be saved by unrolling the message compression procedure and storing all left circular values in macros. Therefore, $V_0^{(0)}, V_1^{(0)}, \ldots, V_7^{(0)}$, T_j $(0 \leqslant j \leqslant 63)$ and their left circular values are all embedded in macros.

Carefully Scheduled Code. The iterative compression stage (cf. Equation B6) is responsible for most of the computations in SM3, which has high data-dependence. To weaken such dependence and allow instruction-level parallelism (ILP), we

3) Available at http://www.openmp.org/.

iterate the computation sequences and test the related performance. After iteration, we derive new equations which are listed in Equation C1. We first remove the variables $TT1$ and $TT2$ and introduce two more registers A' and E' to store A and E temporarily. The updated variables A and E depend on $SS1$, and we put their calculations just after the evaluation of $SS1$. Later equations update other variables merely involve memory access operations. In such a manner, there is a high probability that the computation is overlapped with memory access.

$$\begin{aligned} SS1 &= ((A \lll 12) + E + (T_j \lll j)) \lll 7 \\ A' &= A \qquad A = FF_j(A,B,C) + D + SS1 \oplus (A \lll 12) + W'_j \\ E' &= E \qquad E = P_0(GG_j(E,F,G) + H + SS1 + W_j) \\ D &= C \qquad C = B \lll 9 \\ B &= A' \qquad H = G \\ G &= F \lll 19 \qquad F = E'. \end{aligned} \tag{C1}$$

Inline PTX Assembly. The CUDA framework supports to embed PTX instructions directly which would bring more opportunities for optimizations. During the iterative expansion and compression stage, bitwise operations are all computed with PTX instructions directly. Concretely, instructions `xor.b32`, `and.b32`, `or.b32` and `not.b32` are used for operations XOR, AND, OR and NOT, respectively. The left circular shift operation is more complicated and accomplished with three instructions: `shl.b32`, `shr.b32` and `or.b32`.

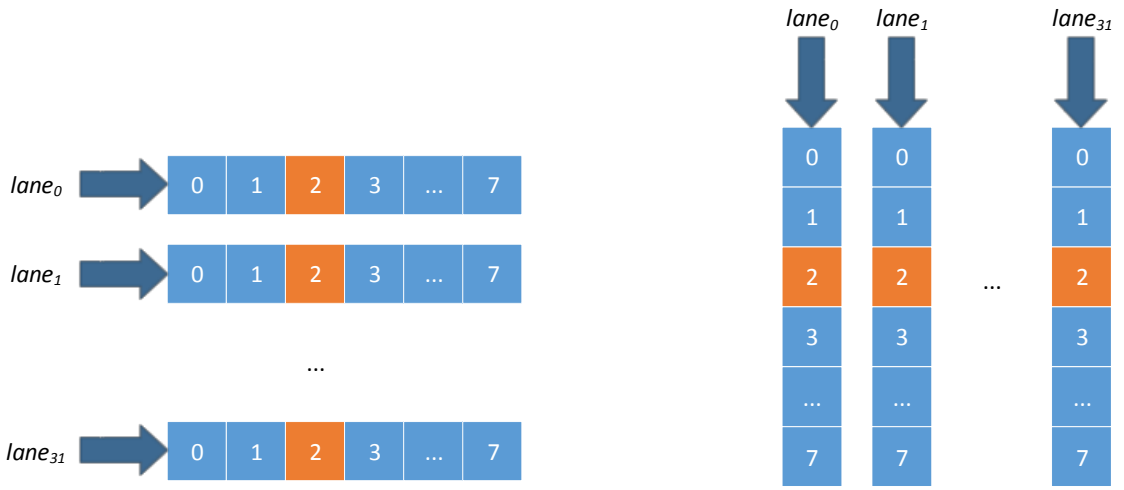
Another frequently used routine is swapping the endian of a 32-bit integer or a 64-bit integer which can utilize the `prmt` instruction. The 32-bit version function of this routine is presented in Listing 1.

Listing 1 Inline PTX example: swap a 32-bit integer's endian.

```
inline __device__ uint32_t swap32_S(uint32_t n) {
    uint32_t result;
    asm volatile ("prmt.b32 %0, %1, 0, 0x0123;" : "=r"(result): "r"(n));
    return result;
}
```

Coalesced Access. The global memory favors the coalesced access fashion [26]. To be specific, if 32 threads in a warp deal with the same 128-byte chunk and the chunk is aligned to 128-byte cache line, a single cache transaction is sufficient. Misaligned and non-coalesced (strided) access pattern will result more transactions in the cost of hundreds of times more cycles. The SM3 algorithm has many global memory access operations: reads the message input and writes the digest output. Therefore, the access pattern must be improved.

Take the output digests as an example. If the digests are stored in a row-major manner (Figure C1(a)), 32 cache transitions are occurred when we write each value of the digest. It means that 4,096-bytes data is transferred where actually only 128-byte data is required. Therefore, a column-major way is adopted (Figure C1(b)). In this case, only a single cache line transaction is needed which certainly is optimal. For the input messages, when they have the same length, the column-major manner is also adopted. When their lengths are different, the input messages are stored in row-major manner.



(a) Row-major storage. (b) Column-major storage.

Figure C1 Row-major storage vs column-major storage for 32 digests. Each box represents a 4-byte integer. Writing the 3-th values in digests results 32 cache line transactions for the row-major storage while only 1 cache transaction is required in the case of column-major storage.

Inline Function or Macro. Invocation of functions brings additional overhead: saving current registers to stack, jumping to the called function address, etc. Using inline function or macro will remove this penalty. For CUDA programming, the compiler nvcc will inline device functions when deemed appropriate. Meanwhile, the pragma __forceinline__ can be used to force the compiler to inline a device function. In our implementation, we adopt the macro way. The routines in SM3 hash algorithm are all written in macros.

Appendix C.3 Optimizing the Data Transfer

Using graphics cards as an accelerator requires data transfer between CPUs and GPUs through PCI Express bus, which introduces an increased latency. We thus optimize this procedure with certain techniques.

Pinned Memory. To improve the memory transfer bandwidth, the pinned memory [26] is used. Page-locked or pinned memory transfers attain the highest bandwidth between the host and the device. Concretely, a specific memory space is allocated and locked on the host side. This memory space is used to exchange data between CPUs and GPUs. Pinned memory is allocated using the `cudaHostAlloc()` function in the CUDA Runtime API. But excessive use of pinned memory will reduce the performance of operation system. In our servers, we notice that using about 50% of the whole memory does not affect the system performance dramatically, which is sufficient for our test data.

Overlapping Memory Copies and Computations. A GPU device typically has more than one data copy engines, e.g., 2 for GeForce GTX 1080. To make full use of these copy engines, we build multiple data streams to transfer data in parallel. Operations in different streams can be interleaved and in some cases overlapped - a property that can be used to hide data transfers between the host and the device. Concretely, multiple data streams are created first. Then the input messages are fragmented to match the number of streams. Each segment of the messages are independently and asynchronously copied from host to device using different streams, and hashed with the SM3 kernel. Finally, the output digests are asynchronously copied back from GPUs to CPUs with corresponding stream. This computation strategy is depicted in Figure C2.

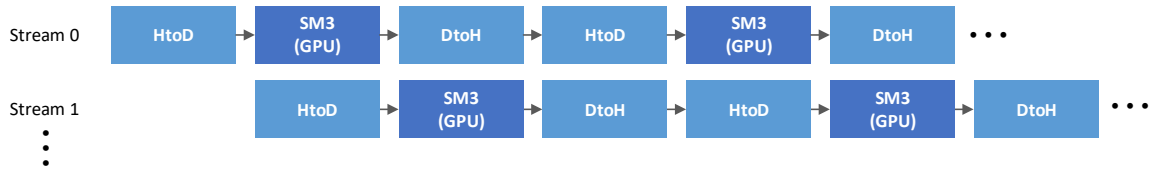


Figure C2 How to overlap data transfer with computation. The symbol HtoD indicates memory transfer from CPUs to GPUs, while the symbol DtoH denotes the reverse case.

After applying above two optimizations, the throughput between CPU and GPU reaches to 95.2 Gbps (resp. 80.0 Gbps) on GTX 1080 (resp. TITAN Xp). Meanwhile, the test tool provided by the NVIDIA CUDA Toolkit reports that the peak bandwidth of PCI-E on GTX 1080 (resp. TITAN Xp) is about 112 Gbps (resp. 90 Gbps). This shows that we make use of 85% (resp. 89%) of the bandwidth, which is a satisfactory result.

Appendix D Performance Evaluation

The SM3 kernel is invoked with a grid configuration that contains a number n_t indicates threads per block and a number n_b indicates blocks per grid. The hardware limitation for n_t is no more than 1,024. Let N be the number of input messages for a kernel which will be limited by the device (mainly the global memory). For n_b, it is computed as $\lceil \frac{N}{n_t} \rceil$. In our early experiments, we observe that n_t actually does not affect the performance as long as it is a multiple of 32. The reason behind this observation is that the schedule unit of SMs is actually a warp (32 threads). If there are enough warps to dispatch, all the SMs on a GPU device will be busy thus fully utilize the computing power. From now on, we fix the number of threads per block n_t to 32.

Fixed Length Data. Our benchmark starts with the fixed length data using only one GPU device on the two platforms. We first generate data files of sizes: 2^0 KB, 2^1 KB, ..., 2^{13} KB. To test the impact of N (the number of simultaneous hashing messages) for the SM3 kernel, we develop an automatic test routine. The routine grows N from 2^5 to 2^i at the rate of 2 until exceeding the hardware limitation. It turns out that the upper bound of N varies by the message size. For instance, in the case of 1 KB, N can be as large as 524,288. For 64 KB, the maximum N can be 32,768. In the case of 1 KB on GTX 1080, when $N = 262{,}144$, the latency is 27.09 ms and the related throughput is 79,272 Mbps. Increasing the N to 524,288 results 54.66 ms latency and decreases the throughput to 78,576 Mbps because of the excessive registers pressure. Akin results are also obtained for other fixed sizes. This means that when N is sufficiently large, the peak throughput is actually obtained. It also proves that a high throughput is obtainable with a low latency. Table D1 presents choices of N that achieve peak throughput and their related latency on the GPU platforms.

Arbitrary Length Data. We also test the case that multiple messages have arbitrary lengths. Our benchmark data is the code package[4)] from ECRYPT Benchmarking of Cryptographic Systems (eBACS). We first build a list of all files in

4) Downloaded from https://bench.cr.yp.to/supercop/supercop-20171218.tar.xz.

Table D1 Optimal throughput and latency of the SM3 hash algorithm (fixed length message).

Size (KB)	GTX 1080			TITAN Xp		
	Optimal N	Latency	Throughput	Optimal N	Latency	Throughput
1	262,144	27.09	79,272	65,536	7.73	69,453
2	524,288	103.45	83,035	524,288	116.69	73,613
4	524,288	201.27	85,357	262,144	106.38	80,748
8	524,288	398.18	86,292	524,288	418.81	82,041
16	262,144	395.95	86,778	16,384	27.87	77,054
32	131,072	395.59	86,857	262,144	838.20	81,985
64	32,768	199.24	86,227	131,072	832.17	82,579
128	32,768	397.94	86,344	65,536	902.26	76,164
256	8,192	200.17	85,826	16,384	428.96	80,100
512	8,192	400.03	85,893	4,096	225.74	76,105
1,024	4,096	414.64	82,866	8,192	851.78	80,677
2,048	2,048	483.44	71,073	4,096	972.42	70,669
4,096	1,024	615.27	55,845	1,024	723.35	47,501
8,192	512	915.45	37,533	256	812.70	21,139

‡ The unit for latency is millisecond and the unit for throughput is Mbps.

all subdirectories. In total, the code package has 24,475 files. Meanwhile, the total size, maximum size and average size of files are 245.59 MB, 4.02 MB and 10.28 KB, respectively. We remark that for the arbitrary length data, the input messages are stored in a row-major manner.

The thread pool on CPU side contains multiple threads where the number of threads should be decided by the specific platforms. Both of our two platforms have 8 cores. When enabling the hyperthreading technique, we observe that using 16 threads obtains the optimal performance. Therefore, we fixed the CPU thread number to 16 in all later experiments.

Similar to the fixed size data, the quantity N is grew from 32 in the rate of 2. The performance result of arbitrary length data is described in Table D2. We observe that when $N = 16{,}384$, the smallest latency is 298.45 ms on GTX 1080, which gives 6,903 Mbps throughput. On the TITAN Xp device, the optimal throughput is 6,585 Mbps in the cost of 312.87 ms latency, which is also obtained when $N = 16{,}384$. Though the obtained throughput is high, it is not comparable with the fixed size case. For instance, the 8 KB fixed data on GTX 1080 obtains peak throughput 86,292 Mbps. The mainly reasons for this performance decrease are: 1) the input messages can not adopt coalesced access technique, which results misaligned access pattern in the cost of more cycles, and 2) different lengths result uneven computation in each thread thus wasting the computing resources of GPUs.

References

1 Stevens M, Bursztein E, Karpman P, et al. The first collision for full SHA-1. In: Katz J, Shacham H, eds. Proceedings of 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, 2017. 570–596

2 Wang X Y, Yu H B. SM3 cryptographic hash algorithm. In: Journal of Information Security Research, 2016. 983–994

3 Ao T Y, He Z Q, Dai K et al. A compact hardware implementation of SM3. In: 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications. 1–4

4 Ma Y, Xia L N, Lin J Q, et al. Hardware performance optimization and evaluation of SM3 hash algorithm on FPGA. In: Chim T W, Yuen T H, eds. Proceedings of 14th International Conference, ICICS 2012, Hong Kong, China, 2012. 105–118

5 Hu Y, Wu L J, Wang A, et al. Hardware design and implementation of SM3 hash algorithm for financial IC card. In: 2014 Tenth International Conference on Computational Intelligence and Security, 2014. 514–518

6 Shen S, Lee X D, Tse R H, et al. The SM3 cryptographic hash function. Internet Engineering Task Force. Internet-Draft, 2017. Work in Progress.

7 Du X J, Li S G. The ASIC implementation of SM3 hash algorithm for High Throughput. In: IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 2016. 1481–1487

8 Elbirt A J, Yip W, Chetwynd B, et al. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9: 4, 2001. 545–557

9 Zhang X M, Parhi K K. High-speed VLSI architectures for the AES algorithm. In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems 12: 9, 2004. 957–967

Table D2 Optimal throughput and latency of the SM3 hash algorithm (arbitrary length data).

N	GTX 1080		TITAN Xp	
	Latency	Throughput	Latency	Throughput
32	7,341.57	281	8,651.28	238
64	4,210.36	489	4,518.81	456
128	2,327.92	885	2,461.89	837
256	1,390.93	1,481	1,515.63	1,359
512	734.19	2,806	827.25	2,490
1,024	416.54	4,946	527.16	3,908
2,048	320.57	6,427	480.12	4,291
4,096	310.99	6,625	402.99	5,112
8,192	306.50	6,722	378.26	5,447
16,384	298.45	6,903	312.87	6,585
32,768	307.94	6,690	314.46	6,552

‡ The unit for latency is millisecond and the unit for throughput is Mbps.

10 Good T, Benaissa M. AES on FPGA from the Fastest to the Smallest. In: Rao J R, Sunar B, eds. Proceedings of 7th International Workshop, Edinburgh, UK, 2005. 427–440

11 Cook D L, Ioannidis J, Keromytis A D, et al. CryptoGraphics: secret key cryptography using graphics cards. In: Menezes A, eds. Proceedings of The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, 2005. 334–350

12 Harrison O, Waldron J. AES encryption implementation and analysis on commodity graphics processing units. In: Paillier P, Verbauwhede I, eds. Proceedings of 9th International Workshop, Vienna, Austria, 2007. 209–226

13 Manavski A S. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: 2007 IEEE International Conference on Signal Processing and Communications, Dubai, 2007. 65–68

14 Harrison O, Waldron J. Practical symmetric key cryptography on modern graphics hardware. In: proceeding SS'08 Proceedings of the 17th conference on Security symposium, 2008. 195–209

15 Li Q J, Zhong C Wen, Zhao K Y, et al. Implementation and analysis of AES encryption on GPU. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication and 2012 IEEE 9th International Conference on Embedded Software and Systems, 2012. 843–848

16 Robert S, Tim G. Exploiting the power of GPUs for asymmetric cryptography. In: Elisabeth O, Pankaj R, eds. Cryptographic Hardware and Embedded Systems – CHES 2008: 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. 79–99

17 Harrison O, Waldron J. Efficient acceleration of asymmetric cryptography on graphics hardware. In: proceedings of the 2nd International Conference on Cryptology in Africa: Progress in Cryptology, 2009. 350–367

18 Eberle H, Gura N, Shantz S C, et al. A public-key cryptographic processor for RSA and ECC. In: proceedings of 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004. 98–110

19 Fan J F, Sakiyama K, Verbauwhede, Ingrid. Elliptic curve cryptography on embedded multicore systems. In: Design Automation for Embedded Systems 12: 3, 2008. 231–242

20 Pan W Q, Zheng F Y, Zhao Y et al. An efficient elliptic curve cryptography signature server with GPU acceleration. In: IEEE Transactions on Information Forensics and Security 12: 1, 2016. 111–122

21 Dai W, Schanck J, Sunar B, et al. NTRU modular lattice signature scheme on CUDA GPUs. In: Cryptology ePrint Archive, Report 2016/471, 2016.

22 Koziel B, Azarderakhsh R, Kermani M M, et al. Post-quantum cryptography on FPGA based on isogenies on elliptic curves. In: IEEE Transactions on Circuits and Systems 64: 1, 2017. 86–99

23 Kuo P C, Li W D, Chen Y W et al. High performance post-quantum key exchange on FPGAs. In: Cryptology ePrint Archive, Report 2017/690, 2017.

24 Oorschot C P, Wiener J M. Parallel collision search with cryptanalytic applications. In: Journal of Cryptology 12: 1, 1999. 1–28

25 Hiroshi, M. Addend dependency of differential/linear probability of addition. In: IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences (Vol. 81), 1998, 106–109

26 NVIDIA. CUDA C Best Practices Guide Version 9.1.85, 2018.

27 NVIDIA. CUDA C Programming Guide 9.1.85, 2018.

28 Bos J W, Stefan D. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In: Mangard S, Standaert FX, eds. Proceedings of 12th International Workshop, Santa Barbara, USA, 2010. 279–293