JCST Journal of Computer Science and Technology

Indexed in: SCIE, EI, Scopus, INSPEC, DBLP,CSCD, etc.
Sponspored by: ICT, CAS & CCF

JCST Papers

Only for Academic and Non-Commercial Use

Thanks for Reading!

Survey

Computer Architecture and Systems

Artificial Intelligence and Pattern Recognition

Computer Graphics and Multimedia

Data Management and Data Mining

Software Systems

Computer Networks and Distributed Computing

Theory and Algorithms

Emerging Areas

JCST WeChat
Subscription Account

JCST URL: https://jcst.ict.ac.cn

SPRINGER URL: https://www.springer.com/journal/11390

E-mail: jcst@ict.ac.cn

Online Submission: https://mc03.manuscriptcentral.com/jcst

Twitter: JCST_Journal

LinkedIn: Journal of Computer Science and Technology

Dong JK, Wu W, Lu S *et al*. HI-SM3: High-performance implementation of SM3 hash function on heterogeneous GPUs. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY, 40(6): 1546-1562, Nov. 2025. DOI: 10.1007/s11390-025-4285-7, CSTR: 32374.14.s11390-025-4285-7

HI-SM3: High-Performance Implementation of SM3 Hash Function on Heterogeneous GPUs

Jian-Kuo Dong[1] (董建阔), *Member, CCF*, Wen Wu[1] (吴　雯), Sheng Lu[1] (陆　盛)
Le-Tian Sha[1,*] (沙乐天), *Member, CCF*, Fang-Yu Zheng[2] (郑昉昱)
Fu Xiao[1] (肖　甫), *Senior Member, CCF, IEEE, Member, ACM*, and Hua-Qun Wang[1] (王化群)

[1] *School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China*

[2] *School of Cryptology, University of Chinese Academy of Sciences, Beijing 101408, China*

E-mail: djiankuo@njupt.edu.cn; 2024040403@njupt.edu.cn; 1021041526@njupt.edu.cn; ltsha@njupt.edu.cn
zhengfangyu@ucas.ac.cn; xiaof@njupt.edu.cn; whq@njupt.edu.cn

Received March 18, 2024; accepted January 8, 2025.

Abstract Hash functions are essential in cryptographic primitives such as digital signatures, key exchanges, and blockchain technology. SM3, built upon the Merkle-Damgard structure, is a crucial element in Chinese commercial cryptographic schemes. Optimizing hash function performance is crucial given the growth of Internet of Things (IoT) devices and the rapid evolution of blockchain technology. In this paper, we introduce a high-performance implementation framework for accelerating the SM3 cryptography hash function, short for HI-SM3, using heterogeneous GPU (graphics processing unit) parallel computing devices. HI-SM3 enhances the implementation of hash functions across four dimensions: parallelism, register utilization, memory access, and instruction efficiency, resulting in significant performance gains across various GPU platforms. Leveraging the NVIDIA RTX 4090 GPU, HI-SM3 achieves a remarkable peak performance of 454.74 GB/s, surpassing OpenSSL on a high-end server CPU (E5-2699V3) with 16 cores by over 150 times. On the Hygon DCU accelerator, a Chinese domestic graphics card, it achieves 113.77 GB/s. Furthermore, compared with the fastest known GPU-based SM3 implementation, HI-SM3 on the same GPU platform exhibits a 3.12x performance improvement. Even on embedded GPUs consuming less than 40W, HI-SM3 attains a throughput of 5.90 GB/s, which is twice as high as that of a server-level CPU. In summary, HI-SM3 provides a significant performance advantage, positioning it as a compelling solution for accelerating hash operations.

Keywords SM3, heterogeneous GPU, CUDA, cryptographic engineering

1 Introduction

As one of the most basic cryptographic algorithms, hash function extracts a fixed-length message digest[1] from a message of any length, and this process is irreversible. In recent years, increasingly popular IoT[2] and blockchain technologies[3] use hash functions. Therefore, the efficient and high-throughput implementation of hash functions is also a hot field of cryptography research in recent years. The most well-known hash functions include MD5[4], SHA-1[5], SHA-2[6], and SM3[7]. Among them, SM3, a commercial hash algorithm independently developed by China, was announced by the State Cryptography Administration of China in 2010. The SM3 algorithm received widespread attention upon its release and offi-

Regular Paper

A preliminary version of the paper was published in the Proceedings of ICPADS 2022.

The work was supported by the National Natural Science Foundation of China under Grant Nos. U23B2002, 62302238, and 62372245, the Natural Science Foundation of Jiangsu Province of China under Grant No. BK20220388, the Natural Science Research Project of Colleges and Universities in Jiangsu Province of China under Grant No. 22KJB520004, the China Postdoctoral Science Foundation under Grant No. 2022M711689, and the CCF-Tencent Rhino-Bird Open Research Fund under Grant No. CCF-Tencent RAGR20240129.

*Corresponding Author

©Institute of Computing Technology, Chinese Academy of Sciences 2025

cially became an ISO/IEC international standard in 2018①.

As a computing platform, GPUs (graphics processing units) have extremely broad application areas, such as machine learning, deep learning, and artificial intelligence[8–12]. In the field of cryptographic engineering, the use of GPU general-purpose computing to accelerate cryptographic algorithms has significant computational advantages. As a new parallel computing architecture, GPUs can be regarded as parallel data processing devices. Thousands of stream processors perform massively parallel computing by efficiently utilizing various types of memory on GPUs to complete different kinds of computing tasks at high speeds. Similarly, China's domestically developed GPU platforms, such as DCUs (deep computing units)[13], excel in parallel computing tasks, making them ideal for hash function computations across various platforms, from servers to edge devices.

1.1 Related Work

Internationally recognized hash algorithms[14] include MD5, SHA-1, RIPEMD, HAVAL, SHA-2, and SHA-3. The SHA-1 algorithm was released in 1995, designed by the United States National Security Agency, followed by the development of the SHA-2 and SHA-3 algorithms. Zhang *et al.*[15] inserted and rearranged two expanders in the pipelined stages of the extended portion of SHA-2. This approach increases the ratio of IMD-REG to the total message word registers by over 20%. Choi and Seo[16] optimized the SHA-3 algorithm on GPUs by utilizing internal processes, inline parallel thread execution (PTX), memory management, and CUDA streams. Their optimizations achieved a throughput increase of 49.73%. Additionally, they suggested that these methods could be applied to SHAKE and post-quantum cryptography (PQC) algorithms.

The SM3② hash algorithm was released by the State Cryptography Administration of China in December 2010. The implementation of the SM3 algorithm across various platforms has been a research hotspot, with numerous articles currently addressing the efficiency and security issues of the SM3 algorithm on different platforms. Zang *et al.*[7] optimized both algorithmic and circuit aspects using the SMIC 40 nm high-performance technology, achieving a throughput of 6.4 Gbps. Zheng *et al.*[17] introduced a novel hybrid cipher framework for securing smart devices, employing SW/HW co-design to implement efficient SM2, SM3, and SM4 schemes, demonstrating superior efficiency and resilience against simple power analysis (SPA) attacks. Cai[18] divided a large number of plaintext messages into 64 groups and processed the data in parallel simultaneously. Sun *et al.*[19] were the first to implement the SM3 algorithm on GPU platforms. They fully leveraged the parallel capabilities of both the CPU and GPU through two dedicated computation modes. In experiments comparing the standard SM3 algorithm with the optimized version using fixed-length and arbitrary-length messages, they achieved a significant acceleration effect. Dong *et al.*[20] enhanced the performance of the SM3 algorithm by meticulously optimizing parallelism, memory access, and instruction sets. Based on the same GTX 1080, their performance is 1.12 times that of Sun's[19] implementation, and the latency is reduced by over 95%. Kuznetso *et al.*[21] used GPU platforms to evaluate various hash algorithms. In their experiments, different GPU devices such as GTX1050 Ti were used. Additionally, their experiment included various hash algorithms, such as SHA, STRIBOG[22], and KEECAK[23]. Tian *et al.*[24] proposed two optimization methods for the SM3 algorithm based on the OpenCL platform, namely password generation and instruction optimization. By simplifying the algorithm and reducing the number of instructions, the computational performance of a single SM3 implementation reached 13 958 MB/s on AMD Radeon R9-200.

Based on the analysis of data dependencies in the SM3 algorithm, Yue[25] reasonably partitioned tasks between the CPU and DCU on a heterogeneous CPU+DCU platform. The heterogeneous optimization of the SM3 algorithm was conducted in terms of computation, storage, data transfer, and communication concealment. Compared with sequential execution on the CPU, the SM3 algorithm achieved acceleration ratios of 1.095 1 and 1.224 9 for the standard implementation and the optimized version, respectively.

1.2 Our Contribution

The previously published conference version[20] optimized the implementation of the SM3 algorithm on

①https://www.iso.org/standard/67116.html, Oct. 2025.
②https://www.oscca.gov.cn/sca/xxgk/2010-12/17/content_1002389.shtml, Oct. 2025.

NVIDIA GPUs in terms of parallelism, memory access, and instruction usage. In contrast, this paper focuses on leveraging heterogeneous GPUs (expanding from NVIDIA GPUs to Chinese Hygon DCU) as accelerated computing platforms to enhance the high-performance implementation of the SM3 hash algorithm (HI-SM3). By employing a range of optimization techniques, particularly the newly proposed loop unrolling and register optimization techniques, our approach outperforms alternative methods in terms of computational acceleration performance. Notably, our method boasts a remarkable enhancement in computational acceleration performance compared with the conference version. This paper makes four key contributions as follows.

- First, we select GPU/DCUs as platforms for cryptographic computing and fully implement the SM3 hash algorithm on these platforms. By leveraging the parallel computing capabilities of GPU/DCUs, we provide more efficient computational support for systems that require a large number of hash function calculations.
- Secondly, we merge the message expansion and compression iteration steps in the SM3 algorithm, performing message expansion in real-time during computation. By utilizing loop unrolling and register optimization, we reduce the required register size from 528 bytes to 128 bytes, significantly enhancing the performance of the SM3 algorithm.
- Thirdly, the PTX ISA, suitable for CUDA programming on GPUs, is utilized, while the AMD GPU ISA is employed within the Hygon DCU, replacing the original code to achieve improved computational speeds. Moreover, we leverage the CUDA/HIP stream technology in GPU/DCUs to effectively parallelize data copying and computational operations, thereby reducing the overall time costs of the computation process.
- Finally, we refer to coalesced access [16], which entails that the starting addresses of all threads are consecutive. In conjunction with INT4 instructions, we propose a new data storage method and utilize this method for parallel computation, thereby enhancing the computation speed of the hashing function and reducing time consumption.

The remaining sections are as follows. Section 2 introduces some background, including the SM3 hash function, heterogeneous GPU platforms such as Titan V, GTX 1080, Jetson Xavier, RTX 4090, Hygon DCU, as well as NVIDIA CUDA, HIP, and their instruction sets. Section 3 introduces our specific optimization implementation scheme. Section 4 shows the experimental results and compares them with results from the standard cryptographic algorithm library OpenSSL[③] and others. Section 5 concludes the paper.

2 Preliminary Knowledge

In this section, we briefly introduce some basic knowledge and platform technologies.

2.1 SM3

SM3 is a cryptographic hash function[7] developed in China. This algorithm was adopted as a standard by the Chinese cryptographic industry in 2012 and as the national hash algorithm standard in 2016. It is extensively utilized in China. Similar to SHA-256, SM3 generates a 256-bit hash value. Furthermore, as technology advances, the algorithm is expected to increasingly facilitate the processing of large-scale data. Therefore, researching high-performance implementation techniques for SM3 is significantly important.

Hash functions play a crucial role in cryptography and are widely employed in digital signatures, message authentication, and data integrity verification. To ensure hash security, the generated hash values must be sufficiently long, because a shorter length compromises security. Therefore, SM3 uses a 256-bit hash value to ensure security. The core algorithm steps of the SM3 compression function CF, illustrated in Fig.1, are executed 64 times in a loop.

Here, $\lll k$ denotes a 32-bit cyclic left shift operation, + indicates modulo 2^{32} arithmetic addition, and

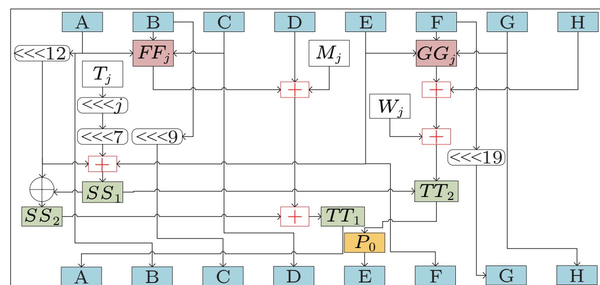

Fig.1. Core algorithm steps of compression function.

③https://www.openssl.org/, Oct. 2025.

P_0 represents a permutation function. FF_j and GG_j are Boolean functions. The formulas for these functions are as follows:

$$FF_j(X,Y,Z) = \begin{cases} X \oplus Y \oplus Z, & \text{if } j \in [0,15], \\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z), & \text{if } j \in [16,63]. \end{cases}$$

$$GG_j(X,Y,Z) = \begin{cases} X \oplus Y \oplus Z, & \text{if } j \in [0,15], \\ (X \wedge Y) \vee (\neg X \wedge Z), & \text{if } j \in [16,63]. \end{cases}$$

$$P_0 = X \oplus (X \lll 9) \oplus (X \lll 17).$$

Meanwhile, A, B, $\ldots$, H serve as word registers, while $SS1$, $SS2$, $TT1$, and $TT2$ function as intermediate variables. Additionally, the significance of W'_j and W_j should not be underestimated. These represent the 132 words derived from the message block expansion, as detailed in Algorithm 1. Consequently, W_0, W_1, $\ldots$, W_{67}, W'_0, W'_1, $\ldots$, W'_{63} are precomputed and stored in GPU/DCUs memory.

Algorithm 1. Message Block Expansion

Input: the 512-bit message block: msg;

Output: the 132 words after expansion: W_j and W'_j;

1: **for** j=0 **to** 15

2: $W_j = Divide(msg)$; /* Divide msg into 16 parts */

3: **endfor**

4: **for** j=16 **to** 67

5: $W_j = P_1(W_{j-16} \oplus W_{j-9} \oplus (W_{j-3} \lll 15)) \oplus (W_{j-13} \lll 7) \oplus W_{j-6}$;

6: **endfor**

7: **for** j=0 **to** 63

8: $W'_j = W_j \oplus W_{j+4}$;

9: **endfor**

2.2 Analysis of GPU/DCUs Architectures

Table 1 presents the specifications of the GPU/DCUs platforms employed in this study. The NVIDIA Titan V[26] is the flagship graphics card released in 2017, embodying NVIDIA's premier design standards. NVIDIA Titan series GPUs are designed as professional computing platforms with high energy efficiency, widely utilized in scientific applications, including large-scale AI, deep learning, and high-performance computing. NVIDIA announced that the Titan V employs a 12 nm Volta architecture, incorporating 21.1 billion transistors, 5 120 CUDA cores, 640 Tensor cores, and 320 texture units[④]. Additionally, the Titan V features 12 GB of memory with a 3 072-bit width, a frequency of 1 700 MHz, and a memory bandwidth of 653 GB/s. In terms of performance, the Titan V architecture combines the L1 data cache with shared memory units, enhancing performance and simplifying programming, achieving a peak tensor performance of up to 110 TFlops.

The GeForce GTX 1080[27] is a desktop GPU released by NVIDIA in 2016 and was the flagship GPU of that year. With the introduction of the GeForce GTX 1080, NVIDIA's GPU manufacturing process officially entered a new era utilizing a 16-nm process. The GTX 1080 utilizes the Pascal GP104 architecture and incorporates 2 560 CUDA cores. The GeForce GTX 1080 is the first to adopt the GDDR5X video memory subsystem, featuring lossless video memory compression technology. The enhancement of the video memory compression architecture, coupled with a video memory frequency of up to 10 Gbps, significantly increases the effective video memory bandwidth available to the Pascal GP104.

The NVIDIA Jetson Xavier[28] was released by NVIDIA in 2018 and has been specifically targeted at the robotics industry. It is a sophisticated minicomputer or embedded system based on the Xavier processor, offering high performance along with nearly all necessary interfaces, power supplies, and functional modules. The Jetson Xavier utilizes the NVIDIA Volta architecture, featuring 512 CUDA cores and 64 Tensor cores. The Jetson Xavier offers performance up to 32 TOPs, although it is less powerful compared with some high-end GPUs such as the Titan V. Its

Table 1. GPU/DCUs Architecture

Platform	Number of Multiprocessors	Number of CUDA Cores	Memory Clock (MHz)	Memory Bus (bit)	Global Memory (MB)	Max Power Consumption (W)
Titan V	80	5 120	850	3 072	12 056	250
GTX 1080	20	2 560	5 005	256	8 116	180
Jetson Xavier	8	512	1 377	256	7 764	30
RTX 4090	128	16 384	10 501	384	24 217	450
Hygon DCU	60	–	1 000	4 096	16 384	350

[④] https://www.techpowerup.com/gpu-specs/titan-v-ceo-edition.c3277, Oct. 2025.

key advantage as an embedded device is its low power consumption.

The NVIDIA GeForce RTX 4090[29] is based on the Ada Lovelace architecture with 16 384 CUDA cores and a clock frequency of 2.2 GHz. It also contains 512 tensor cores, 176 ROPs, and 128 RT cores. The memory dimension is 24 GB (GDDR6X), and the maximum bandwidth is 1 008 GB/s.

The DCU accelerator demonstrates exceptional computational and memory access performance, using a 7-nm process and incorporating 2.5D Interposer system-on-chip (SoC) technology. It features high bandwidth memory 2 (HBM2) on-chip, delivering a memory bandwidth of up to 1 TB/s. The peak single-precision performance is 13.3 TFLOPS, while the peak double-precision performance is 6.5 TFLOPS. The FP64 computational capability has been experimentally validated to achieve a peak performance of 6.5 TFLOPS. The architecture of the DCU acceleration device is depicted in Fig.2. The SIMD calculation group consists of 16 DCU cores and 16 384 32-bit registers, with one scheduler and four SIMD calculation groups forming a computing unit (CU). Each computing unit includes 64 KB of local data share (LDS), with LDS in separate computing units operating independently. Threads within a computing unit can access LDS more quickly than global memory.

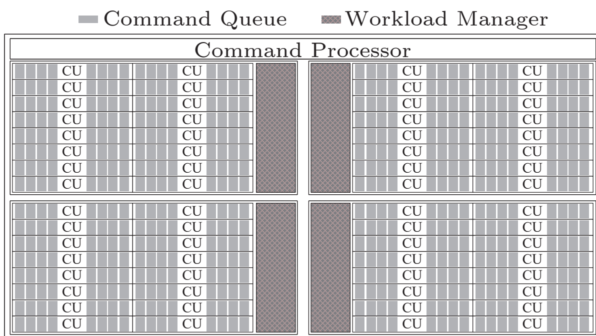

Fig.2. DCU acceleration device architecture.

2.3 CUDA, HIP, and ISA

CUDA⑤ is a computing platform and programming model introduced by NVIDIA, a leading graphics card manufacturer, in 2006. It is a general-purpose parallel computing architecture designed to leverage the extensive parallel computing capabilities of GPUs to address complex computational problems. Additionally, CUDA provides high-level programming language interfaces that allow developers to write programs optimized for the CUDA architecture. These programs can execute on CUDA-compatible processors with exceptional performance. This significantly simplifies CUDA programming, enabling users to quickly utilize CUDA technology and give higher priority to optimizing algorithms.

The DCU system employs the ROCm heterogeneous interface for portability (HIP)⑥ programming model, which is a C/C++-based framework that offers a header file and runtime library based on the hcc compiler. As illustrated in Fig.3, the HIP programming environment bears similarities to the CUDA programming environment in both hardware architectures and software implementations. HIP and CUDA employ analogous programming models, thread structures, memory organizations, and runtime API interfaces, providing a basis for the seamless migration of existing CUDA programs to HIP. Beyond the hardware architectures and runtime software layers, the organization of higher-level mathematical libraries and their function interfaces also reveal notable simi-

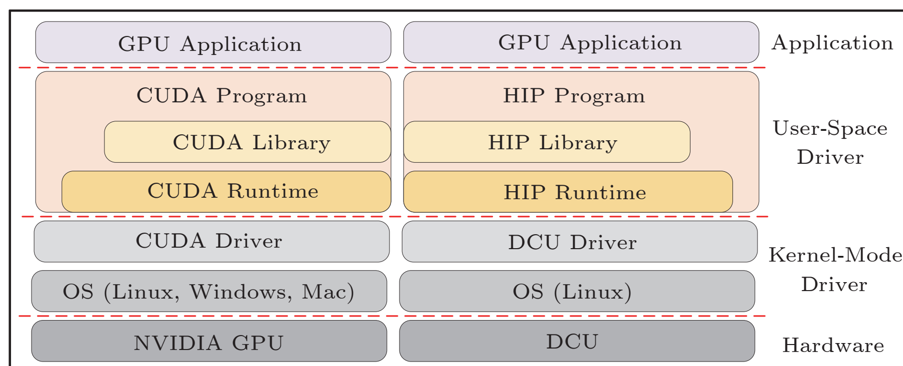

Fig.3. Comparison of HIP and CUDA software stack.

⑤https://docs.nvidia.com/cuda/, Oct. 2025.

⑥https://developer.sourcefind.cn/gitbook/dcu_developer/DeveloperGuide/dcu_programming/, Oct. 2025.

larities. This architectural resemblance simplifies the migration of existing programs and code using CUDA acceleration to the DCU platform.

NVIDIA introduced a low-level parallel thread execution (PTX) virtual machine and instruction set architecture (ISA)[7] to facilitate programmers who wish to develop applications using CUDA. PTX ISA is a stable and widely applicable instruction set architecture across different GPUs, utilizing a model similar to CUDA C++ but operating at a lower level. Some fundamental instructions, along with their corresponding explanations, are presented in Table 2. Listing 1 provides an example (32-bit integer remainder, $s = a\%b$) of inline PTX ISA by using inline compilation. Additionally, %0 refers to the first operand, %1 to the second operand, and %2 to the third operand.

Table 2. PTX ISA Instructions

PTX Instruction	Meanings
$s = add(a, b)$	$s = a + b$
$s = rem(a, b)$	$s = a\%b$
$s = shr(a, b)$	shift right $s = (a >> b)$
$s = shl(a, b)$	shift left $s = (a << b)$
$s = and(a, b)$	$s = (a\&b)$
$s = or(a, b)$	$s = (a \mid b)$
$s = xor(a, b)$	$s = (a \oplus b)$
$s = not(a)$	$s = \sim a$
$s = lop3(a, b, c, d)^*$	$s = (a, b, c, d)$

Note: * means arbitrary logical operation on 3 inputs. The bitwise logical operation on inputs a, b, and c is to be computed, and the result is to be stored in destination s. The logical operation is defined by a look-up table which, for 3 inputs, can be represented as an 8-bit value specified by operand d. d is an integer constant that can take values of 0 to 255, thereby allowing up to 256 distinct logical operations on inputs d, b, c.

Listing 1. Example of Inline PTX ISA

```
asm volatile(
    "template"   e.g., rem.u32 %0, %1, %2;
    :  "constraint"(output)   e.g.,   "= r"(s)
    :  "constraint"(input)    e.g.,   "= r"(a),
       "constraint"(input)    e.g.,   "= r"(b)
);
```

The AMD GPU ISA constitutes the basic instruction set specified for AMD GPU hardware. Its design is intended to support highly parallel computing tasks, encompassing a broad spectrum of instruction types, including arithmetic, logic, memory access, and flow control. This architecture features multiple register files for storing and processing various types of data. The memory model addresses various levels of memory, including local, private, shared, and global memory, to enhance memory access efficiency. The AMD GPU ISA supports debugging and performance analysis features and is integrated with programming models such as OpenCL and AMD ROCm, enabling developers to access powerful tools for optimizing parallel computing capabilities.

3 Proposed Optimization Techniques for SM3

In this section, we propose several optimized techniques for the SM3 algorithm. Specifically, according to the characteristics of GPU/DCUs, we propose several optimization methods of the SM3 algorithm and apply these optimization methods to GPU/DCUs.

3.1 Design Criteria for Proposed Software

In this subsection, we propose a CPU-GPU/DCUs joint computing technique, as detailed in Fig.4. Initially, the host (CPU) consolidates the input data to be encrypted and processes it in groups. It then uses the device (GPU/DCUs) copy engine to transfer data from CPU memory to GPU/DCUs memory. Subsequently, the kernels of the GPU/DCUs initiate a large number of threads and perform cryptographic parallel operations on the requests within each stream. Finally, the results of the parallel computations are copied back to CPU memory. The output data from different requests is then redistributed in CPU memory, and the final calculation results are returned.

Throughout the process, the host (CPU) can configure the number of threads and thread blocks required by the device (GPU/DCUs). The GPU/DCUs employs the configured blocks and threads for parallel computation. The entire implementation is executed by running a kernel function. In our experiment, we configure the device's kernel capabilities for SM3.

A specific number of threads are organized into blocks, and a specific number of these blocks form a grid, creating a hierarchical organizational structure known as the "thread-block-grid". The indexing of threads, blocks, and grids is automatically managed

[7] https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, Oct. 2025.

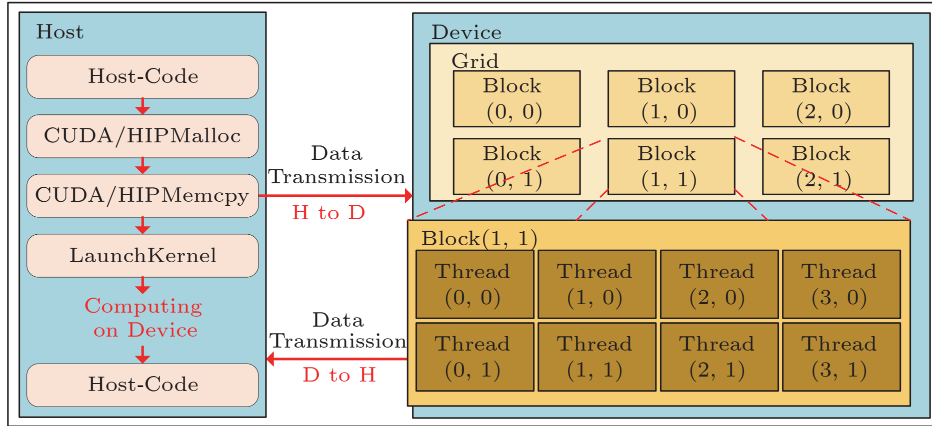

Fig.4. CPU-GPU/DCUs joint computing model.

by the CUDA/HIP runtime system, with thread IDs specified by *cuda/hipThreadIdx*. Each thread executing a kernel function is assigned a unique thread ID, with index conversion accomplished through computational methods. The kernel function executed on the device side manipulates threads within the GPU/DCUs to perform required data access and computational operations as needed.

3.2 Overall Structure of Optimization

In this subsection, we summarize the optimization methods implemented on GPU/DCUs for SM3. The overall optimization scheme is shown in Fig.5.

First, this paper extends the 64-cycle compression function CF by implementing a simultaneous computation and compression method, which replaces the previously precomputed 132 words and significantly reduces register usage. Second, in GPU programming, we utilize the PTX ISA instruction sets to supersede the original code implementations on other platforms, thereby achieving faster execution speeds. Similarly, on the DCU accelerator, the AMD GPU ISA is used to replace the original code. Third, we employ CUDA/HIP streams to minimize the latency of data transfer between the GPU/DCUs and CPU. Specifically, our implementation leverages CUDA/HIP streams to concurrently execute kernel functions and data transfers. In our implementation, each thread computes the hash digest of the message using the hash algorithm, and these threads are invoked via CUDA/HIP streams. Finally, due to the high memory access latency inherent in the GPU/DCUs architecture, which often exceeds the runtime of some kernel functions, optimizing memory access times is crucial for enhancing the hash algorithm's computational speed. Therefore, we also employ optimized coalesced memory access to further enhance hash function performance.

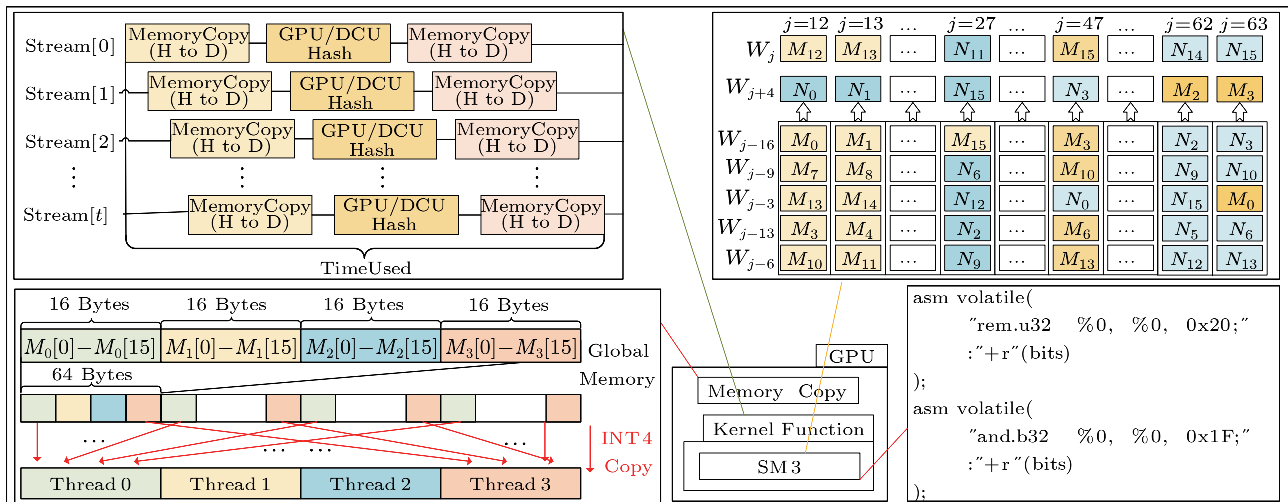

Fig.5. Overall structure of SM3 optimization.

3.3 Register Optimization

As described in Algorithm 1, two loops are utilized during the expansion of grouped messages. The output 132-word messages are stored in W_j and W'_j, and kept in the GPU/DCUs registers until the compression is completed, at which point they are released. However, in the compression function CF, as the number of iterations increases, it becomes evident that only six words at most are utilized in each loop iteration. This result in register wastage consequently impacts the execution speed of the SM3 algorithm. Therefore, this subsection proposes a fusion scheme that combines expansion and compression concurrently, enhancing register reuse and effectively reducing register usage.

The method proposed in this paper only requires 32 words (M_i, N_i, $i \in [0, 15]$) to replace W_j and W'_j, respectively, thereby achieving both message expansion and compression. As a result, the actual register size employed decreases from 528 bytes to 128 bytes. The specific implementation is as follows.

3.3.1 Loops 0–11

Algorithm 2 illustrates the specific implementation of this process, in which $CF(M_j, N_j)$ represents the SM3 compression function. This process utilizes only 16 blocks of 512-bit messages.

Algorithm 2. Expansion & Compression for Loops 0–11

Input: the 512-bit message block: msg;
Output: the result after 12 rounds of compression;
1: **for** j=0 **to** 15
2: $M_j = Divide(msg)$; /* Divide msg into 16 parts */
3: **endfor**
4: **for** j=0 **to** 11
5: $N_j = M_j \oplus M_{j+4}$;
6: $CF(M_j, N_j)$;
7: **endfor**

In this scenario, the optimization for calculating the intermediate values $TT1$ and $TT2$ is as follows:

$$TT1 = FF_j(A, B, C) + D + SS2 + (M_i \oplus M_{i+4}),$$
$$TT2 = GG_j(E, F, G) + H + SS1 + M_i,$$

where $i \in [0, 11]$. Thus, it can be observed that the implementation of the first 12 rounds of loops is relatively straightforward, as the required values of M_i are precomputed, and the calculation process only involves reading the corresponding data.

3.3.2 Loops 12–63

According to Algorithm 1, the calculation method for W_{16} to W_{67} in the SM3 expansion algorithm is as follows:

$$W_j = P_1(W_{j-16} \oplus W_{j-9} \oplus (W_{j-3} \lll 15)) \oplus (W_{j-13} \lll 7) \oplus W_{j-6}.$$

In programming, the 68 words in the above equation can be compressed into 16 words, as detailed below:

$$W_{j\bmod 16} = P_1(W_{(j-16)\bmod 16} \oplus W_{(j-9)\bmod 16} \oplus (W_{(j-3)\bmod 16} \lll 15)) \oplus (W_{(j-13)\bmod 16} \lll 7) \oplus W_{(j-6)\bmod 16}.$$

The calculation method for W'_0 to W'_{63} is $W'_j = W_j \oplus W_{j+4}$, similar to the optimization method described above, which reuses 16 words of the register space for 64 words. In computation, it only requires calculating with a delay of four words compared with W_j. Its expansion can be represented by the following formula:

$$W'_{j\bmod 16} = W_{j\bmod 16} \oplus W_{(j+4)\bmod 16}.$$

Therefore, it is possible to reuse M_i and N_i to store calculation results. The specific implementation method is illustrated in Fig.6. The described approach is based on loop unrolling, and in this paper, this portion of the code is implemented through macros. The macro code is shown in Listing 2, where

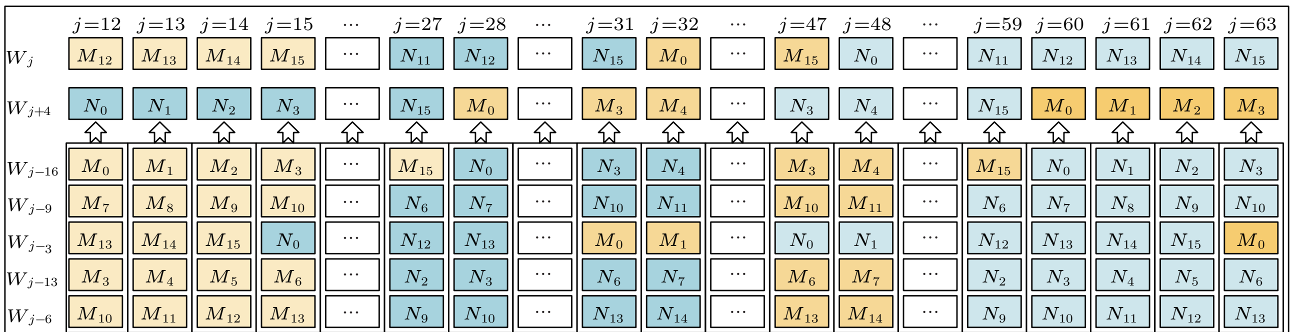

Fig.6. Implementation details of simultaneous expansion and compression.

Listing 2. Macro Code for Loops 12–63

```
#defineLoop_12_63(j, X, Y, R, S, T, U, V)                                   \
    SS1 = LeftRotate((LeftRotate(A, 12) + E + LeftRotate(T(j), j)), 7);     \
    SS2 = SS1^LeftRotate(A, 12);                                            \
    Y = P1(R^S^LeftRotate(T, 15))^LeftRotate(U, 7)^V;                       \
    TT1 = FF(A, B, C, j) + D + SS2 + (X^Y);                                 \
    TT2 = GG(E, F, G, j) + H + SS1 + X;                                     \
    D = C;    tRotate(B, 9);          B = A;    A = TT1;                    \
    H = G;    G = LeftRotate(F, 19);  F = E;    E = P0(TT2);                \
```

LeftRotate$(A, 12)$ represents cyclically shifting A to the left by 12 positions, i.e., $A \lll 12$.

Through this macro, relevant parameters can be passed to it for each iteration, thereby achieving the functionality of the loop during each execution. Simultaneously, the use of this macro simplifies the code structure after loop unrolling. Specifically, in the j-th iteration (where $j \in [12, 63]$), substitute the values corresponding to this loop in Fig.6 for the placeholders X, Y, R, S, T, U, V in the macro.

3.4 Optimizing SM3 with Instruction Set

For the internal optimization of the SM3 algorithm on a GPU, we employ the PTX ISA instruction set to replace specific internal computing operations within SM3 during our experiments. The PTX instruction set represents the assembly language instruction set used in CUDA. Assembly language is a low-level programming language that interfaces directly with a hardware. Compared with high-level programming languages such as C++ and Java, assembly languages (e.g., PTX ISA) typically offer faster execution. Utilizing assembly language can significantly reduce the time required for instruction transmission and machine language conversion, thereby enhancing the overall efficiency of the SM3 algorithm. For instance, the rotate-left operation in SM3 is optimized through instruction substitution, as detailed in Algorithm 3. The rotate-left operation is a frequent operation in the SM3 algorithm. We employ the PTX ISA instruction set to replace the original C code, including instructions for remainder, left shift, right shift, and bitwise OR operations. The test shows that replacing the instruction set significantly enhances the performance of the SM3 hash algorithm.

Due to the architectural differences between the Hygon DCU and NVIDIA GPU, the PTX ISA is not applicable to the Hygon DCU. Instead, the AMD GPU ISA should be employed. The specific implementations mirror the PTX ISA.

Algorithm 3. Rotate Left Operation Based on PTX

Input: the 32-bit element: *word*, and the 32-bit element: *bits*;
Output: the 32-bit element: *word*;

```
1: temp = 0
   Step (1):
2: asm volatile(
3:     "rem.u32 %0, %0, 0x00000020;"
4:     :"+r"(bits)
5: )
   Step (2):
6: asm volatile(
7:     "shr.b32 %0, %1, %2;"
8:     :"+r"(temp)
9:     :"r"(word),"r"(32 − bits)
10: )
   Step (3):
11: asm volatile(
12:     "shl.b32 %0, %0, %1;"
13:     :"+r"(word)
14:     :"r"(bits)
15: )
   Step (4):
16: asm volatile(
17:     "or.b32 %0, %0, %1;"
18:     :"+r"(word)
19:     :"r"(temp)
20: )
```

3.5 Use of CUDA/HIP Streaming Technology

The CUDA/HIP operation generally refers to data transfer between the host (CPU) and the device (GPU/DCUs) and the execution of the kernel function. The running order for each operation in a CUDA/HIP stream is controlled by the host, and the operations are executed in sequence according to the commands from the host.

We utilize the CUDA/HIP streaming technology in our optimization, allowing different operations to overlap between the host and the device through asynchronous CUDA/HIP operations. This means that we can execute operations queued in the stream simultaneously with other useful operations, thereby hiding the overhead of some operations in the execution process. In many computing scenarios, executing

a kernel function takes more time than transferring data between the host and the device. Therefore, we can hide the transfer delay between the host and the device by using the CUDA/HIP stream technology. The specific execution architecture is shown in Fig.7. The entire process of our calculation is data copying, calculations performed by devices, and data returning. These three operations can be overlapped, which means that the data copying of the next set of data has already started during the calculations of the previous set of data, effectively reducing the overall execution time of the program.

3.6 Optimized Memory Access

In this subsection, we introduce another GPU/DCUs-based optimization technique, which is based on coalesced memory access. By using this optimization technique, the speed of data reading and writing can be effectively improved, thereby accelerating the calculation of the entire hash algorithm.

Global memory occupies a significant portion of the total memory in the GPU/DCUs. Before the GPU/DCUs device performs calculations, the data should be copied to device memory first, usually the global memory. When the GPU needs to access the memory to retrieve data, it accesses memory in units of warp. A warp here consists of 32 threads, and these threads in the same warp execute the same instruction. However, in the DCU accelerator, the number of wavefronts is 64. This implies that, within the same time frame, a SIMD in the DCU can concurrently execute a wavefront composed of 64 threads. There are two main access modes for global memory, which are coalesced memory access and non-coalesced memory access, respectively.

Taking the GPU as an example, coalesced memory access refers to a warp's request to access global memory that results in the least amount of data transfers. Otherwise, it is referred to as non-coalesced memory access. Specifically, when a warp accesses memory, each thread in the warp accesses the data it needs, and if the storage addresses of this data are consecutive, memory access can be implemented efficiently, which is called coalesced access. In the DCU accelerator, similar memory access patterns exist. Based on this method of coalesced memory access in the GPU/DCUs, a new data storage method is proposed[16]. Specifically, the original horizontal storage of data is changed to vertical storage. By using this new storage method, each element in data begins at a continuous address, meaning that the addresses accessed by each thread are consecutive, which can significantly enhance data access speed.

In addition, CUDA/HIP provides special instructions to facilitate data access, including the INT, INT2, and INT4 instructions. By using the INT, INT2, and INT4 instructions, the computing program can read or write 4, 8, and 16 bytes of data to the global memory once, respectively. These particular instructions that read multiple bytes at a time are faster than ordinary data access instructions. Based on these characteristics, we propose a new data storage method based on the vertical method[16]. In our experiment, we compare the INT, INT2, and INT4 instructions, and finally find that the INT4 instruction is the fastest. Therefore, we use the INT4 instruction in the experiment.

The new data storage structure we propose is shown in Fig.8. The upper part of the figure is a common calculation model. Taking the 64-byte plaintext and four threads as an example, the offset between

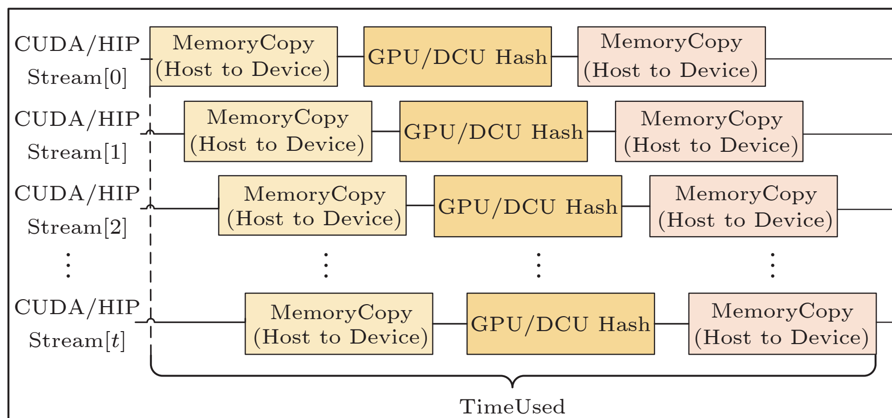

Fig.7. CUDA/HIP stream.

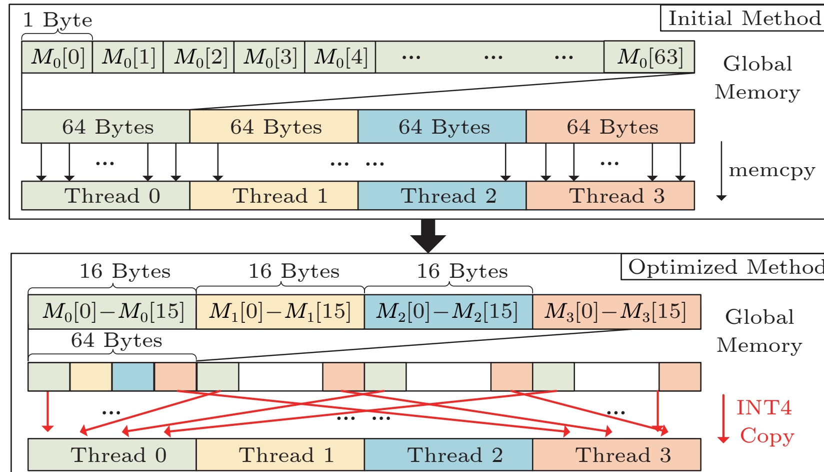

Fig.8. INT4-based coalesced memory access method.

each thread is 64 bytes. The second part is the calculation model we propose. Similarly, taking the 64-byte plaintext and four threads as an example, we split into 16-byte groups, and 64 bytes are divided into four groups. In this way, we can see from the figure that the offset between each thread is 16 bytes, which is less than 64 bytes. At the same time, because it is a group of 16 bytes, we can use the INT4 instruction in CUDA/HIP to copy data in batches, which can save more time than ordinary copy operations. In general, we use 16 bytes as a new data element to make these new data elements continuous, and we use the INT4 instruction to access when reading and writing. Compared with the initial memory access method, the speed of the SM3 algorithm is greatly improved by using our data access method.

4 Performance Analysis

We present our experimental setup, discuss the performance results of SM3 based on various optimization implementations on different platforms, and compare the results with related work. In addition, we show the performance differences between our implementation and OpenSSL.

4.1 Experimental Setup

In our experiments, performance measurements are performed in the environments of the Titan V, GTX 1080, Jetson Xavier, RTX 4090 and Hygon DCU, and the CPU is under Intel® Xeon® CPU E5-2699 v3 @ 2.30 GHz environment. The specific GPU hardware configurations for the experiments are detailed in Table 1. In addition, the CUDA version is 11.5.1, and the HIP version is 5.2.

4.2 Performance Evaluation

Latency and throughput serve as crucial metrics for evaluating the performance of our implementation, where throughput represents the amount of data in megabytes that can be hashed per second (MB/s). Tailored for high-concurrency scenarios, our objective is to achieve "as large a throughput with acceptable latency". Consequently, the latency metric serves as a practical gauge of the viability of our high-throughput solution. Making full use of the total available threads can keep the pipeline busy most of the time and take greater advantage of the computing power in GPU/DCUs. Each thread runs one SM3 instance. Thus, the batch size, defined as the product of the grid size and the block size, plays a critical role in the performance of GPU kernels.

Because the Titan V has 80 streaming multiprocessors (SMs), the GTX 1080 has 20 SMs, the Jetson Xavier has 8 SMs, the RTX 4090 has 128 SMs, and the Hygon DCU has 60 computing units, we set the number of thread blocks in our experiments to match the number of SMs or computing units on each platform. Additionally, the sizes of thread blocks are set to 256 512 768, and 1 024, respectively. Table 3 shows the results of the optimized SM3 algorithm on the Titan V, GTX 1080, Jetson Xavier, RTX 4090 and Hygon DCU. We test the effect of different numbers of threads on the throughput of SM3 when the plaintext sizes are 64 bytes and 8 192 bytes (8 KB), respectively.

The specific throughput peaks are shown in bold in Table 3. We can see from the table that when the

Table 3. Performance of SM3

Plaintext (byte)	TitanV		GTX 1080		Jetson Xavier		RTX 4090		Hygon DCU	
	Threads	Tput	Threads	Tput	Threads	Tput	Threads	Tput	Threads	Tput
64	80×256	28 952	20×256	7 606	8×256	512	128×256	96 169	60×256	18 246
	80×512	31 480	20×512	9 870	8×512	911	128×512	118 251	60×512	27 115
	80×768	33 312	20×768	10 666	8×768	1 215	128×768	127 204	60×768	35 122
	80×1 024	**34 679**	20×1 024	**12 800**	8×1 024	**1 442**	128×1 024	**132 979**	60×1 024	**35 363**
8 192	80×256	102 852	20×256	19 876	8×256	5 797	128×256	458 041	60×256	65 893
	80×512	116 570	20×512	39 399	8×512	6 003	128×512	465 655	60×512	94 038
	80×768	120 326	20×768	**39 777**	8×768	**6 042**	128×768	**469 640**	60×768	111 049
	80×1 024	**123 002**	20×1 024	31 848	8×1 024	5 470	128×1 024	463 512	60×1 024	**116 507**

Note: Threads: block size × grid size. Tput stands for throughput (MB/S).

plaintext size is 64 bytes, the peak throughput of SM3 on the Titan V reaches 34 679 MB/s, on the Jetson Xavier it attains 1 442 MB/s, on the GTX 1080 it attains 12 800 MB/s, on the RTX 4090 platform the throughput reaches its peak at 132 979 MB/s, and the Hygon DCU accelerator achieves a throughput of 35 363 MB/s. In the case of 8 192 bytes of plaintext data, SM3 achieves peak performance on the Titan V at 123 002 MB/s, on the GTX 1080 at 39 777 MB/s, on the Jetson Xavier with a peak throughput of 6 042 MB/s, on the RTX 4090 reaching an impressive 469 640 MB/s, and on the Hygon DCU accelerator demonstrating a throughput of 116 507 MB/s. The table data shows significantly higher throughput with an 8 192-byte plaintext compared with a 64-byte plaintext. We can see that the throughput performance, from high to low, is the RTX 4090, Titan V, Hygon DCU, GTX 1080, and Jetson Xavier. The reason is that the computing resources of these four platforms are not the same: the RTX 4090 has the strongest computing power, followed by the Titan V, Hygon DCU, and GTX 1080, while the Jetson Xavier, being an embedded GPU, exhibits comparatively lower computing capabilities.

As shown in Fig.9, we present the histogram of the SM3 algorithm on different platforms and thread counts. We especially provide the performance of the SM3 algorithm implemented by Dong *et al.*[20] on the same platform. The throughput performance of SM3 corresponds to the primary vertical axis (left), with units in GB/s, while latency corresponds to the secondary vertical axis (right), where the unit of latency is milliseconds (ms). Regarding latency, we can see that as the number of threads increases, the latency consumed by computation also increases. Furthermore, we observe from the experimental results that when the number of threads is different on the same platform, the throughput tends to change significantly. Generally, throughput improves as the number of threads increases. However, in some cases, within our implementation, increasing the number of threads results in a decrease in throughput. In fact, it is not the case that starting more threads necessarily leads to better throughput results, as this is also related to registers used in the calculation. If too many registers are utilized, using more threads in some cases will result in a decrease in throughput performance.

From Table 3, for a plaintext size of 64 KB, GPU compute resources are underutilized, leading to lower performance. However, with an increase in plaintext length to 8 192 KB, there is a significant performance improvement. To assess whether these results approach peak performance on each platform, we select optimal thread block sizes from Table 3 and test various thread block counts across platforms. The experimental findings are depicted in Fig.10. As the number of thread blocks increases, throughput stabilizes across platforms, while latency for completing computational requests gradually rises. This suggests that when the number of thread blocks matches the number of streaming multiprocessors, GPU compute performance is nearly maximized, with further increases potentially leading to unacceptable latencies.

4.3 Performance Comparison

In this subsection, we detail the peaks of our experimental results in Table 4. Additionally, we compare our work with the GPU implementations, DCU-based implementations, and implementations on CPU and FPGA, respectively. It can be seen that, compared with CPUs and FPGAs, GPU/DCUs typically have higher power consumption. This is because GPU/DCUs designs emphasize parallel computing capabilities, integrating numerous computing units and memory controllers to support high-throughput parallel computing tasks.

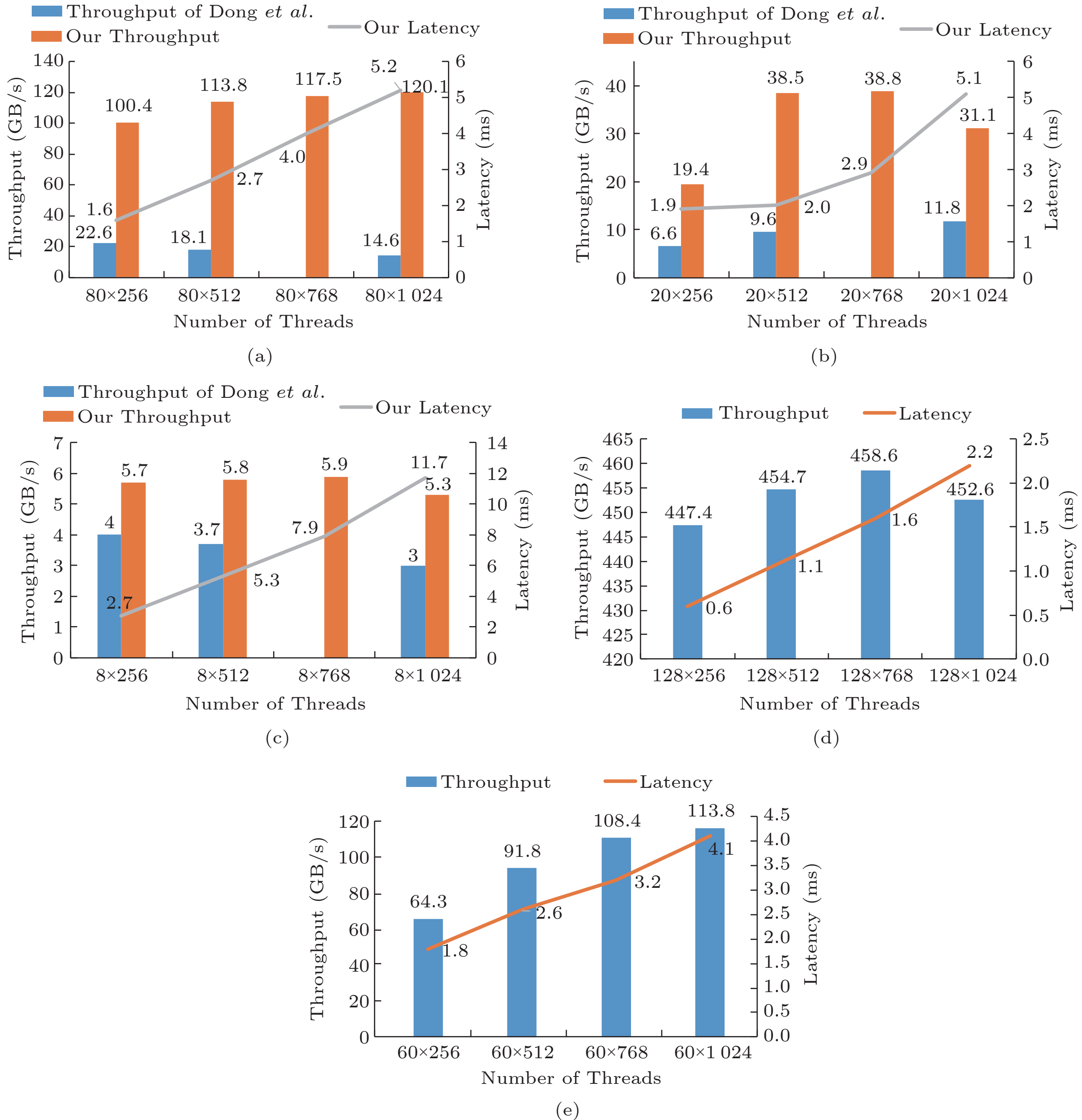

Fig.9. Performance of SM3 with 8 KB plaintext. Number of threads = the block size × the grid size. (a) Titan V. (b) GTX 1080. (c) Jetson Xavier. (d) RTX 4090. (e) Hygon DCU.

When the message block size is 8 KB, Sun *et al.*[19] achieved a throughput of 10 786 MB/s with a latency of 398.18 ms for the SM3 algorithm on the GTX 1080. On the same platform, we measure a throughput of 39 777 MB/s, which is 3.7 times their performance, and the latency is significantly lower than Sun *et al.*[19] Similarly, the performance achieved in this study on the GTX 1080 is 3.1 times that of Dong *et al.*[20] result. Dong *et al.*[20] attained a peak SM3 performance of 23 113 MB/s on the Titan V platform, whereas our study demonstrates a throughput of 123 002 MB/s on the same platform, indicating a remarkable 5.3-fold increase. However, on the Jetson Xavier platform, Dong *et al.*[20] achieved an SM3 peak performance of 4 134 MB/s, while our study showcases a throughput of 6 042 MB/s, signifying a notable 1.46-fold improvement. This is because our study significantly reduces register usage in the compression function (CF) by employing loop unrolling and register optimization techniques. This fully harnesses the parallel computing capabilities of the GPU, preventing performance degradation due to resource con-

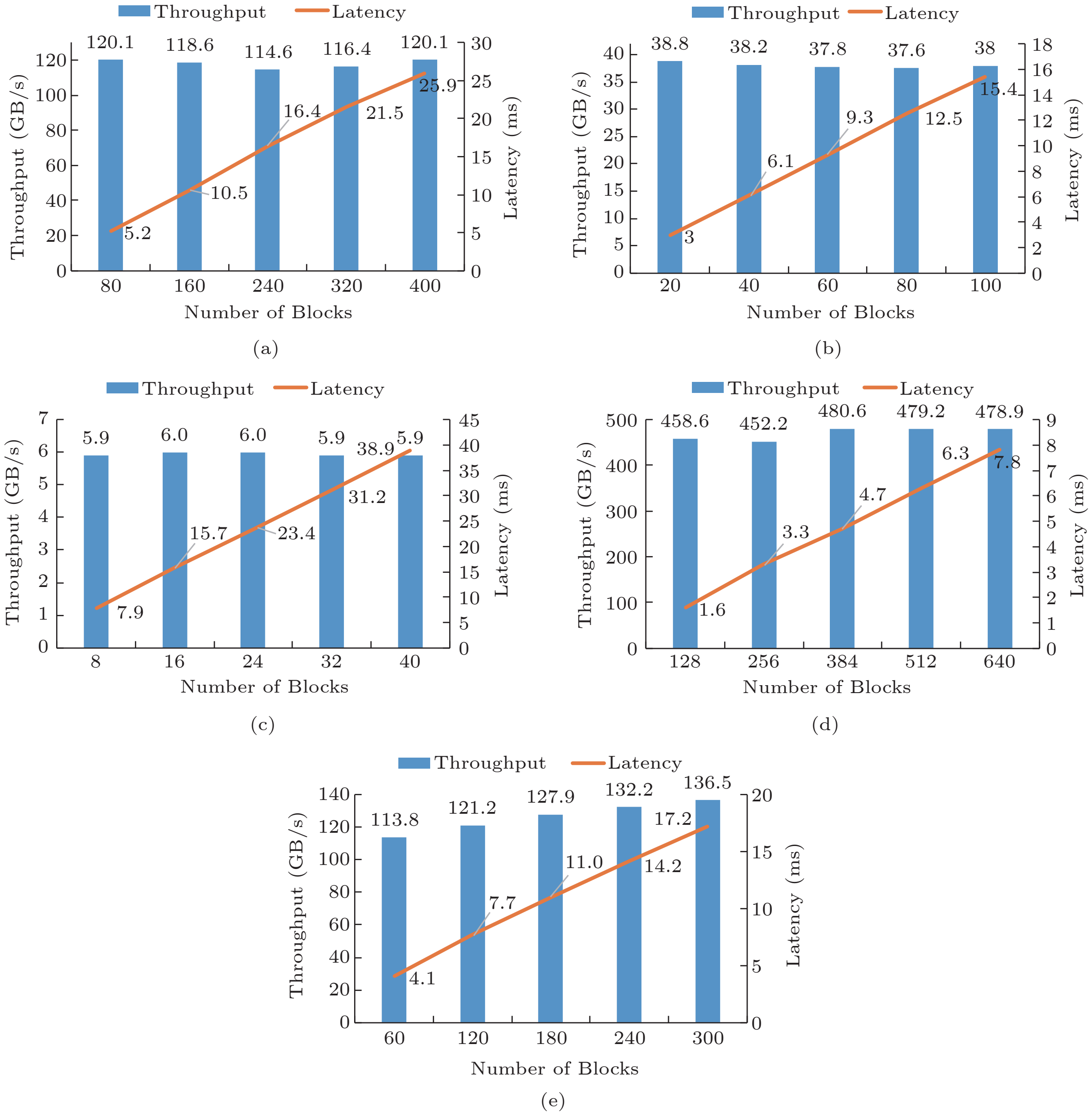

Fig.10. Performance of SM3 under different numbers of blocks. (a) Titan V under threads/block=1 024. (b) GTX 1080 under threads/block=768. (c) Jetson Xavier under threads/block=768. (d) RTX 4090 under threads/block=768. (e) Hygon DCU under threads/block=1 024.

straints and consequently enhancing the computational efficiency of the algorithm.

Yue *et al.*[25] utilized strategies such as thread organization, loop unrolling, on-chip caching, and register optimization on the Hygon DCU platform. In measurements on the same platform, the latency for the SM3 algorithm was found to be 54.7 ms when the message block size was 128 KB. However, our measured latency on the same platform only requires 38.3 ms, indicating a performance improvement of nearly 40%.

From Table 4, we can see that compared with OpenSSL, our method achieves higher throughput both in plaintext whether it is 64 bytes or 8 192 bytes. OpenSSL here uses multi-threaded speed measurement, which corresponds to the number of CPU cores available on the experimental platform. Even on the embedded GPU Jetson Xavier, the throughput reaches 6 042 MB/s, which is twice the OpenSSL performance on a server-level CPU. Huang *et al.*[30] attained a performance level of 286.33 MB/s for the SM3 algorithm on the Kintex-Ultrascale platform. Meanwhile,

Table 4. Performance Comparison of SM3

Method	Platform	Power Consumption (W)	Package Size (byte)	Throughput (MB/s)	Latency (ms)
Huang *et al.*[30]	Kintex-Ultrascale @ 286.33 MHz	30.00	–	286.33	–
Zheng *et al.*[17]	C-Sky CK802 @ 36 MHz	2.88	–	–	0.133
	ARM Cortex-A9 @ 666 MHz	–	–	–	0.032
Zang *et al.*[7]	SMIC 40nm	–	–	819.00	–
Hao *et al.*[24]	AMD 7970 GPU	250.00	–	13 958.00	–
Yue[25]	Hygon DCU	350.00	131 072	–	54.700
Sun *et al.*[19]	GTX 1080	180.00	8 192	10 786.00	398.180
	Titan Xp	250.00		10 255.00	418.810
OpenSSL[20]	Intel® Xeon® 1223 -E5-2699 v3 @ 2.30 GHz	145.00	64	1 223.00	–
			8 192	3 043.00	–
Dong *et al.*[20]	Titan V	250.00	8 192	23 113.00	–
	GTX 1080	180.00		12 134.00	–
	Jeston Xavier	30.00		4 134.00	–
Ours	Titan V	250.00	64	34 679.00	0.144
			8 192	123 002.00	5.198
	GTX 1080	180.00	64	12 800.00	0.097
			8 192	39 777.00	3.016
	Jeston Xavier	30.00	64	1 442.00	0.356
			8 192	6 042.00	7.944
	RTX 4090	450.00	64	132 979.00	0.061
			8 192	469 640.00	1.635
	Hygon DCU	350.00	64	35 363.00	0.106
			8 192	116 507.00	4.119
			131 072	100 353.00	38.272

Zang *et al.*[7] optimized both algorithmic and circuit aspects using SMIC's 40 nm high-performance technology, resulting in a throughput of 819 MB/s. The experimental results indicate that the throughput of our SM3 hash algorithm implementation far exceeds that of CPUs or FPGAs.

5 Conclusions

In response to the current demand for cryptographic computing performance, our research presented a high-performance implementation framework for the SM3 hash algorithm. By employing loop unrolling and register optimization techniques, we significantly reduced the register usage in the compression function CF, leading to a substantial improvement in the performance of SM3. In view of the characteristics of the GPU/DCUs architecture, we used the PTX ISA/AMD GPU ISA and CUDA/HIP streaming technology to accelerate hash calculation. Additionally, by combining coalesced memory access to global memory with the INT4 instruction, we proposed an optimized memory access method and implemented it in the SM3 hash algorithm. As a result, we achieved a peak performance of 454.74 GB/s on the NVIDIA RTX 4090, which is over 150 times faster than OpenSSL running on a 16-core server CPU.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Paar C, Pelzl J. Kryptografie Verständlich: Ein Lehrbuch für Studierende und Anwender. Springer, 2016. DOI: 10.1007/978-3-662-49297-0. (in German)

[2] Farhan L, Shukur S T, Alissa A E, Alrweg M, Raza U, Kharel R. A survey on the challenges and opportunities of the Internet of Things (IoT). In *Proc. the 11th International Conference on Sensing Technology*, Dec. 2017. DOI: 10.1109/ICSensT.2017.8304465.

[3] Zheng Z, Xie S, Dai H, Chen X, Wang H. An overview of blockchain technology: Architecture, consensus, and future trends. In *Proc. the 2017 IEEE International Congress on Big Data*, Jun. 2017, pp.557–564. DOI: 10.1109/BigDataCongress.2017.85.

[4] Rivest R. The MD5 message-digest algorithm. Technical Report, Google Research, 1992. https://www.rfc-editor.org/rfc/rfc1321, Oct. 2025.

[5] Eastlake III D, Jones P. US secure hash algorithm 1 (SHA1). Technical Report, Google Research, 2001. https://www.rfc-editor.org/rfc/rfc3174, Oct. 2025.

[6] Chaves R, Sousa L, Sklavos N, Fournaris A P, Kalogeri-

dou G, Kitsos P, Sheikh F. Secure hashing: Sha-1, sha-2, and sha-3. *Circuits and systems for security and privacy*, 2016: 105–132.

[7] Zang S, Zhao D, Hu Y, Hu X, Gao Y, Du P, Cheng S. A high speed SM3 algorithm implementation for security chip. In *Proc. the 5th Advanced Information Technology, Electronic and Automation Control Conference*, Mar. 2021, pp.915–919. DOI: 10.1109/IAEAC50856.2021.9390790.

[8] Dong J, Zheng F, Lin J, Liu Z, Xiao F, Fan G. EC-ECC: Accelerating elliptic curve cryptography for edge computing on embedded GPU TX2. *ACM Trans. Embedded Computing Systems*, 2022, 21(2): Article No. 16. DOI: 10.1145/3492734.

[9] Lew J, Shah D A, Pati S, Cattell S, Zhang M, Sandhupatla A, Ng C, Goli N, Sinclair M D, Rogers T G, Aamodt T M. Analyzing machine learning workloads using a detailed GPU simulator. In *Proc. the 2019 IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2019, pp.494–504. DOI: 10.1109/ISPASS.2019.00028.

[10] Gao L, Zheng F, Emmart N, Dong J, Lin J, Weems C. DPF-ECC: Accelerating elliptic curve cryptography with floating-point computing power of GPUs. In *Proc. the 2020 IEEE International Parallel and Distributed Processing Symposium*, May 2020, pp.151–152. DOI: 10.1109/IPDPS47924.2020.00058.

[11] An S W, Seo S C. Highly efficient implementation of block ciphers on graphic processing units for massively large data. *Applied Sciences*, 2020, 10(11): Article No. 3711. DOI: 10.3390/app10113711.

[12] Gu J, Chowdhury M, Shin K G, Zhu Y, Jeon M, Qian J, Liu H, Guo C. Tiresias: A GPU cluster manager for distributed deep learning. In *Proc. the 16th USENIX Symposium on Networked Systems Design and Implementation*, Feb. 2019, pp.485–500.

[13] Tian Z, Yang S, Zhang C. Accelerating sparse general matrix-matrix multiplication for NVIDIA Volta GPU and Hygon DCU. In *Proc. the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, Aug. 2023, pp.329–330. DOI: 10.1145/3588195.3595936.

[14] Easttom W. Cryptographic hashes. In *Modern Cryptography: Applied Mathematics for Encryption and Information Security*. Springer International Publishing, 2021, pp. 205–224. DOI: 10.1007/978-3-030-63115-4_9.

[15] Zhang Y, He Z, Wan M, Zhan M, Zhang M, Peng K, Song M, Gu H S. A new message expansion structure for full pipeline SHA-2. *IEEE Trans. Circuits and Systems I: Regular Papers*, 2021, 68(4): 1553–1566. DOI: 10.1109/TCSI.2021.3054758.

[16] Choi H, Seo S C. Fast implementation of SHA-3 in GPU environment. *IEEE Access*, 2021, 9: 144574–144586. DOI: 10.1109/ACCESS.2021.3122466.

[17] Zheng X, Xu C, Hu X, Zhang Y, Xiong X. The software/hardware co-design and implementation of SM2/3/4 encryption/decryption and digital signature system. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 2020, 39(10): 2055–2066. DOI: 10.1109/TCAD.2019.2939330.

[18] Cai B, Bai G. Implementation of pipeline structure of SM3 algorithm on FPGA. *Microelectronics & Computer*, 2015, 32(1): 15–18 .(in Chinese)

[19] Sun S, Zhang R, Ma H. Hashing multiple messages with SM3 on GPU platforms. *Science China Information Sciences*, 2021, 64(9): 199103. DOI: 10.1007/s11432-018-9648-x.

[20] Dong J, Lu S, Zhang P, Zheng F, Xiao F. G-SM3: High-performance implementation of GPU-based SM3 hash function. In *Proc. the 28th IEEE International Conference on Parallel and Distributed Systems*, Jan. 2023, pp.201–208. DOI: 10.1109/ICPADS56603.2022.00034.

[21] Kuznetsov A, Shekhanin K, Kolhatin A, Kovalchuk D, Babenko V, Perevozova I. Performance of hash algorithms on GPUs for use in blockchain. In *Proc. the 2019 IEEE International Conference on Advanced Trends in Information Theory*, Dec. 2019, pp.166–170. DOI: 10.1109/ATIT49449.2019.9030442.

[22] Dolmatov V, Degtyarev A. GOST R 34.11-2012: Hash function. Technical Report, Google Research, 2013. https://www.rfc-editor.org/rfc/rfc6986, Oct. 2025.

[23] Bertoni G, Daemen J, Peeters M, Van Assche G. The making of KECCAK. *Cryptologia*, 2014, 38(1): 26–60. DOI: 10.1080/01611194.2013.856818.

[24] Tian H, Li Y, Wang Y, Peng T, Shi S, Qiu W. Optimized password recovery based on GPUs for SM3 algorithm. In *Proc. the 3rd International Conference on Computer Science and Application Engineering*, Oct. 2019, Article No. 148. DOI: 10.1145/3331453.3361632.

[25] Yue Y. Implementation and optimization of national secret algorithm for DCU accelerator [Master's Thesis]. Zhengzhou University, Zhengzhou, 2022. DOI: 10.27466/d.cnki.gzzdu.2022.005457. (in Chinese)

[26] Turner D, Andresen D, Hutson K, Tygart A. Application performance on the newest processors and GPUs. In *Proc. the 2018 Practice and Experience on Advanced Research Computing: Seamless Creativit*, Jul. 2018, Article No. 37. DOI: 10.1145/3219104.3219158.

[27] Dat T N, Iwai K, Kurokawa T. Implementation of high speed hash function Keccak using CUDA on GTX 1080. In *Proc. the 5th International Symposium on Computing and Networking*, Nov. 2017, pp.475–481. DOI: 10.1109/CANDAR.2017.47.

[28] Shin D J, Kim J J. A deep learning framework performance evaluation to use YOLO in Nvidia Jetson platform. *Applied Sciences*, 2022, 12(8): 3734. DOI: 10.3390/app12083734.

[29] Luo W, Fan R, Li Z, Du D, Wang Q, Chu X. Benchmarking and dissecting the NVIDIA hopper GPU architecture. In *Proc. the 2024 IEEE International Parallel and Distributed Processing Symposium*, May 2024, pp.656–667. DOI: 10.1109/IPDPS57955.2024.00064.

[30] Huang X, Guo Z, Song M, Zeng X. Accelerating the SM3 hash algorithm with CPU-FPGA Co-Designed architecture. *IET Computers & Digital Techniques*, 2021, 15(6): 427–436. DOI: 10.1049/cdt2.12034.

Jian-Kuo Dong received his B.E. degree from Xi'an Jiaotong University, Xi'an, in 2014, and his Ph.D. degree from the University of Chinese Academy of Sciences, Beijing, in 2019. He is currently an assistant professor at the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing. His research interests include cryptographic engineering, public key cryptography, and applied cryptography.

Wen Wu received his B.E. degree from Nanjing University of Posts and Telecommunications, Nanjing, in 2022, and is currently pursuing his Ph.D. degree at the School of Computer Science and Technology of the same university. His research interests include cryptographic engineering and public key cryptography.

Sheng Lu received his B.E. and M.E. degrees from Nanjing University of Posts and Telecommunications, Nanjing, in 2021 and 2024 respectively. He is currently employed at Nanjing Honor Software Technology Co., Ltd. His research interests include cryptographic engineering, public key cryptography, and high-performance computing.

Le-Tian Sha received his B.S. degree in information security from Southwest Jiaotong University, Chengdu, in 2007, his M.S. degree in information security from the University of Electronic Science and Technology of China, Chengdu, in 2010, and his Ph.D. degree in information security from Wuhan University, Wuhan, in 2014. He is currently a professor at the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing. His research interests include software security, cyber security, and defense in Internet of Things.

Fang-Yu Zheng received his B.E. degree from the University of Science and Technology of China, Hefei, in 2011, and his Ph.D. degree from the University of Chinese Academy of Sciences, Beijing, in 2016. He is currently an assistant professor at the School of Cryptology, University of Chinese Academy of Sciences, Beijing. His research interests include cryptographic engineering, high-performance computing, and applied cryptography.

Fu Xiao received his Ph.D. degree in computer science and technology from Nanjing University of Science and Technology, Nanjing, in 2007. He is currently a professor and a Ph.D. supervisor at the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing. His research interests include wireless sensor networks and mobile computing.

Hua-Qun Wang received his B.S. degree in mathematics education from Shandong Normal University, Jinan, in 1997, and his M.S. degree in applied mathematics from East China Normal University, Shanghai, in 2000. He received his Ph.D. degree in information security from Nanjing University of Posts and Telecommunications, Nanjing, in 2006. He is currently a professor of Nanjing University of Posts and Telecommunications, Nanjing. His research interests include applied cryptography, network security, and cloud computing security.