Cybersecurity

RESEARCH

Open Access

GAPS: GPU-accelerated processing service for SM9

Wenhan Xu[1,2], Hui Ma[1,2] and Rui Zhang[1,2*]

Abstract

SM9 was established in 2016 as a Chinese official identity-based cryptographic (IBC) standard, and became an ISO standard in 2021. It is well-known that IBC is suitable for Internet of Things (IoT) applications, since a centralized processing of client data (e.g. IoT cloud) is often done by gateways. However, due to limited computation resources inside IoT devices, the performance of SM9 becomes a bottleneck in practical usage. The existing SM9 implementations are often CPU-based, with relatively low latency and low throughput. Consequently, a pivotal challenge for SM9 in large-scale applications is how to reduce the latency while maximizing throughput for numerous concurrent inputs. After a systematic analysis of the SM9 algorithms, we apply optimization techniques including precomputation, resource caching and parallelization to reduce the overhead of SM9. In this work, we introduce the first practical implementation of SM9 and its underlying `SM9_P256` curve on GPU. Our GPU implementation combines multiple algorithms and low-level optimizations tailored for GPU's single instruction, multiple threads architecture in order to achieve high throughput for SM9. Based on these, we propose `GAPS`, a high-performance Cryptography as a Service (CaaS) for SM9. `GAPS` adopts a heterogeneous computing architecture that flexibly schedules the inputs across two implementation platforms: a CPU for the low-latency processing of sporadic inputs, and a GPU for the high-throughput processing of batch inputs. According to our benchmark, `GAPS` only takes a few milliseconds to process a single SM9 request in idle mode. Moreover, when operating in its batch processing mode, `GAPS` can generate 2,038,071 private keys, 248,239 signatures or 238,001 ciphertexts per second. The results show that `GAPS` scales seamlessly across inputs of different sizes, preliminarily demonstrating the efficacy of our solution.

Keywords Identity-based cryptography, SM9, Cryptography as a service, Graphics processing units

Introduction

Identity-based cryptography (IBC) (Shamir 1984) is a special public-key cryptographic scheme where the public key can be an arbitrary string, e.g., email address or domain name. Such an attractive feature reduces the overhead of deploying public-key cryptosystems by eliminating the need for public-key infrastructure (PKI), and is considered a key technique to realize certificateless cryptography (Al-Riyami and Paterson 2003). Since its first introduction in 1984 (Shamir 1984), IBC has been widely studied and was standardized by various international organizations, e.g., RFC (2007), ISO/IEC (2021), IEEE (2013).

In the realm of IBC, the SM9 Cryptographic Schemes (Cheng 2017) are a series of cryptographic algorithms (digital signature, key agreement and encryption) specified in the Chinese National Cryptography Standard GM/T 0044-2016 (GM/T 2016a) and ISO's international standards (ISO/IEC 2018, 2021). As a set of lightweight and standardized algorithms, SM9 has been considered a practical solution for cloud and Internet of Things (IoT) security.

*Correspondence:
Rui Zhang
r-zhang@iie.ac.cn
[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, No.19 Shucun Road, Haidian District, Beijing 100085, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

Springer Open

© The Author(s) 2024. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

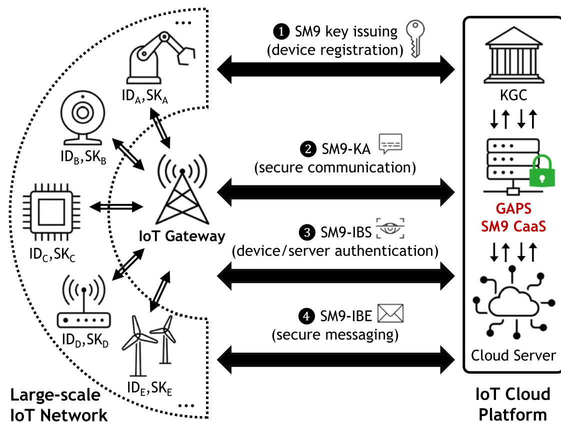

Fig. 1 Application of the SM9 cryptographic schemes in IoT cloud

Motivating scenario

As the demand for SM9-based security solutions continues to grow, the need for efficient implementation of SM9 becomes paramount. For example, Xiaomi (2023) reported that there are 654.5 million consumer devices connected to its IoT cloud platform. Assuming these devices connect to the cloud on a per-day basis, the server needs to handle on average $\frac{654.5*10^6}{24*60*60} \approx 7,575$ op/s. If SM9 is employed for secure communication, the workload would further escalate.

Figure 1 depicts an example of how SM9 can be applied to protect data and communication security in a large-scale IoT network. A key generation center (KGC) holds the system-wide master keys and is in charge of issuing SM9 private keys for registered devices. SM9-KA can be used to set up secure channels among devices, gateways and the cloud, SM9-IBS can be used for device authentication, and SM9-IBE can be used for encrypted messaging in the IoT network. As the IoT gateway and cloud are connected with many end devices, they need to handle the following compute-intensive tasks.

1. *Batch Key Generation.* During the device registration phase, the device manufacturer provides a list of device IDs, and the KGC should be able to handle the batch generation of private keys for the registered devices. The speed of key generation dominates the efficiency of this process.
2. *Concurrent Key Agreement.* For secure communication, the IoT gateway and the cloud should be able to handle large numbers of SM9-KA requests from end devices concurrently. The speed of SM9-KA greatly affects the efficiency of session establishment and the maximum number of secure connections that can be maintained by the gateway (or server).
3. *Batch signature processing.* The end IoT devices send signed data packets to the cloud server. In order to validate the authenticity of data packets, the server will have to perform batch verification of SM9-IBS signatures. Also, when the IoT cloud wants to issue signed commands to end devices, it needs to generate a batch of signatures. Therefore, the speed of SM9-IBS signature generation/verification is crucial to the server's overall performance.
4. *Batch ciphertext processing.* Similar to SM9-IBS, the cloud server should be able to handle batch encryption/decryption of SM9-IBE ciphertexts for efficiency considerations.

Unfortunately, such high requirement for operation throughput can be hardly met by today's SM9 implementations. Many solutions rely on the GmSSL toolkit (GmSSL 2023), which is the most popular software implementation of the Chinese national cryptographic algorithms. However, according to Sun et al. (2020a), deploying SM9-based solutions with GmSSL introduces significant computing latencies (over 300 ms), and would take thousands of CPU cores in order to reach our target throughput (10k+ op/s). Therefore, a pivotal challenge for SM9 applications is how to reduce the computing latency and improve the throughput for large number of requests.

Challenges and solutions

Inspired by the real-world deployment of cryptography using dedicated accelerators, e.g., Hardware Security Modules (Entrust 2023), we explore high-performance *SM9 Cryptography as a Service (CaaS)* for applications with substantial computational demands. Several technical challenges arise in our design of the service's architecture.

- *How to scale seamlessly across inputs of different sizes.* As a dedicated cryptography service, we expect SM9 CaaS to respond swiftly to *sporadic requests* while handling *batch inputs* with high throughput. Traditional CPU-based implementation using fast pairing libraries like the RELIC toolkit (Aranha et al. 2014) can satisfy our first expectation as its latency is at the milliseconds level. However, CPU fails to meet our second expectation as its throughput is limited by the number of available cores. Meanwhile, a high-throughput platform (GPU) is preferred for the efficient processing of batch inputs, but introduces significantly higher latency than CPU. Therefore, the challenge lies in the design of a *task scheduling strategy* that optimally leverages the advantages of multiple platforms.

- *How to optimize the computations in SM9.* Another challenge is the design of a systematic optimization strategy for SM9 algorithms. Although previous studies explored techniques like precomputation (Pan et al. 2017), they only focus on optimizing a single mathematical operation like point multiplication and lack a comprehensive and systematic analysis of optimizations in the entire SM9 cipher suite.
- *How to obtain a high-throughput implementation of SM9 on GPU.* We rely on GPU for the parallel processing of batch inputs. Here the challenge is how to maximize the throughput of SM9 and its underlying mathematical operations (bilinear pairing and elliptic curve). Although much progress has been made on their optimizations on CPU (Beuchat et al. 2010; Aranha et al. 2011), the actual details vary significantly on GPU's *Single Instruction, Multiple Threads (SIMT)* architecture. For example, traditional implementation of point multiplication uses the double-and-add algorithm with the scalar encoded in binary or non-adjacent form for acceleration, which is less efficient on GPU due to its data-dependent divergences that reduce the throughput of parallel threads. Additionally, the state-of-the-art implementations (Aranha et al. 2014; Shigeo 2015) are optimized using low-level CPU intrinsics like AVX2, which are not available on GPUs.

Our Contributions. In this paper, we propose `GAPS`, a GPU-Accelerated Processing Service for SM9. Our contributions are four-fold:

- First, we propose a scalable and heterogeneous architecture for SM9 CaaS. Our architecture combines the SM9 implementations on two different platforms: an efficient CPU implementation with low computing latencies, and a GPU implementation with high batch processing throughput. We employ a unique scheduling strategy that combines the strengths of both platforms for a scalable performance.
- Second, by systematically analyzing the workload of SM9 algorithms, we propose several optimization techniques that utilize precomputation, resource caching and parallelization to reduce the running time of SM9 algorithms.
- Third, we introduce the first practical implementation of the entire SM9 cipher suites and its underlying `SM9_P256` curve on the GPU platform. Our implementation combines multiple algorithm optimizations and low-level optimizations tailored for GPU's SIMT architecture, and is capable of evaluating 158,991 pairings and 2,585,630 point multiplications per second, effectively boosting the performance of SM9.
- Finally, we conduct an extensive benchmark of `GAPS`. We find that `GAPS` is capable of generating up to 2,038,071 private keys per second and can handle 77,797–1,137,015 SM9-KA/IBS/IBE requests per second in batch processing mode. Moreover, when operating in idle mode, `GAPS` processes a single SM9 request with less than 4 milliseconds of latency. The results show that `GAPS` scales well across sporadic and batch inputs, making it a practical solution for SM9 CaaS in large-scale IoT applications.

Related work

Design and implementation of SM9. Since its proposal in 2016, various efforts have been made to improve the functionality and efficiency of SM9. Sun et al. (2020a) proposed a server-aided user revocation mechanism for SM9, Zhang et al. (2020) proposed a distributed key generation scheme for SM9-based systems, while Lai et al. (2022) applied the online/offline methodology to SM9-IBS. Few works have looked into the high-performance implementation of SM9. Sun et al. (2020a) implemented their SM9 revocation scheme in GmSSL and used OpenMP to scale it to multiple cores, but the overall throughput is still far from `GAPS` 's goal. Jing et al. (2022) provided an efficient FPGA implementation of SM9 that takes 0.848 ms to perform on point multiplication. Compared with these works, we focus on designing a high-performance cryptographic service for the entire SM9 cipher suite that is capable of handling tens of thousands of operations per second.

Fast implementation of pairing. Due to its complexity, the optimization and implementation of pairing have attracted a line of research efforts. In Beuchat et al. (2010), the authors introduced the first high-speed implementation of the BN254 pairing with few million CPU cycles. Aranha et al. (2011) further reduced the cost to less than one million cycles on modern CPUs. Few works have explored the implementation of bilinear pairing on hardware platforms, as Cheung et al. (2011) and Pu and Liu (2013) introduced the first FPGA and GPU implementation of the BN254 pairing respectively, though their performance is less competitive than that of today's CPU implementations. Until recently, Hu et al. (2023) introduced a high-performance GPU implementation of the BN254 curve that can compute over 40k pairings per second, yet the algorithms and optimizations (e.g., point multiplication, exponentiation, miller loop) used in their work are somewhat outdated. Compared to these pure GPU-based solutions, `GAPS` adopts a heterogeneous CPU & GPU architecture and applies many

state-of-the-art optimization techniques. We report the highest throughput for pairings (158,991 op/s) on commodity processors.

Cryptography implementation on GPUs. With the rapid development of GPU's computing power, cryptographers have been looking to exploit GPUs to accelerate cryptography implementations. This idea was first put forward by Cook et al. (2005) in 2005, where the authors reported an optimized AES implementation over graphic cards. Following that, Szerwinski and Güneysu (2008) further explored the implementation of asymmetric ciphers on GPUs. Jang et al. (2011) later designed a GPU-based cryptography accelerator for SSL, while Wei et al. (2021) proposed a GPU-based heterogeneous protocol stack for PAKE. Unlike previous works, we explore the GPU acceleration of SM9 for the first time.

Background

Notations

Let $\|$ denote the bitwise concatenation operation. An algorithm is efficient if it runs in probabilistic polynomial time (PPT) in the length of its input. $y \leftarrow F(x)$ denotes running an algorithm F with input x and output y. The following functions are used in SM9. One may refer to ISO/IEC 18033-5 (ISO/IEC 2021) for detailed definitions.

1. $\mathsf{KDF2}(\mathsf{H_v}, \mathsf{m}, \ell)$: Given a hash function H_v with v-bit output, a bit string m, and a non-negative integer ℓ, this key derivation function outputs an ℓ-bit octet key string.
2. $\mathsf{H2RF_i}(\mathsf{H_v}, \mathsf{m}, \mathsf{n})$: Given a hash function H_v, a bit string m, two non-negative integers n and i, it outputs an integer $h_i \in [1, n-1]$.

Bilinear pairing

SM9 is defined over an elliptic curve with bilinear pairings. Suppose $\exists$ a bilinear group generator $\mathcal{G} \leftarrow \mathsf{GroupGen}(1^\lambda)$, where a PPT algorithm $\mathsf{GroupGen}$ takes as input a security parameter λ and returns a group description $\mathcal{G} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, P_1, P_2, e)$, where p is a prime of $\Theta(\lambda)$ bits, $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are cyclic groups of order p, and P_1, P_2 are generators of $\mathbb{G}_1, \mathbb{G}_2$, respectively. The efficiently-computable map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ should satisfy the following properties:

1. Bilinearity: for all $(P, Q) \in \mathbb{G}_1 \times \mathbb{G}_2$ and all $a, b \in \mathbb{Z}$, $e([a]P, [b]Q) = e(P, Q)^{ab}$.
2. Non-degeneracy: $e(P_1, P_2) \neq 1$.

The most efficient construction of a pairing relies on particular family of curves. Particularly, GM/T 0044-2016.5 (GM/T 2016b) specifies a 256-bit Barreto-Naehrig (BN) curve (Barreto and Naehrig 2005) for SM9 (denoted `SM9_P256`). Below we review the related concepts.

Definition 1 (*BN Curves*) The BN curves are a family of elliptic curves $E : y^2 = x^3 + b, b \neq 0$ parameterized by an arbitrary integer $x \in \mathbb{Z}$. It is defined over a prime field $\mathbb{F}_p$, where the prime q, the prime group order p of the pairing groups and the trace t are polynomials given as:

$$q(x) = 36x^4 + 36x^3 + 24x^2 + 6x + 1$$
$$p(x) = 36x^4 + 36x^3 + 18x^2 + 6x + 1$$
$$t(x) = 6x^2 + 1$$

Definition 2 (*Optimal Ate Pairing over BN Curves*) Let $E[p]$ be the subgroup of p-torsion points of E and $E' : y^2 = x^3 + b/\xi$ be the sextic twist of E with ξ not a cube nor a square in $\mathbb{F}_{q^2}$. The Optimal Ate Pairing (Vercauteren 2008) over the BN curves is defined as:

$$a_{opt} : \mathbb{G}_2 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$$
$$(Q, P) \rightarrow (f_{\ell,Q}(P) \cdot l_{[\ell]Q, \pi_q(Q)}(P) \cdot l_{[\ell]Q+\pi_q(Q), -\pi_q^2(Q)}(P))^{\frac{q^{12}-1}{p}}$$

where $\ell = 6x + 2$; $\pi_q(x, y) = (x^q, y^q)$ is the Frobenius endomorphism; groups $\mathbb{G}_1$, $\mathbb{G}_2$ are determined by the eigenspaces of π_q as $\mathbb{G}_1 = E[p] \cap Ker(\pi_q - [1]) = E(\mathbb{F}_q)[p]$ and $\mathbb{G}_2$ as the preimage $E'(\mathbb{F}_{q^2})[p]$ of $E[p] \cap Ker(\pi_q - [q]) \subseteq E(\mathbb{F}_{q^{12}})[p]$ under the twisting isomorphism $\psi : E' \rightarrow E$; the group $\mathbb{G}_T$ is the subgroup of p-th roots of unity $\mu_p \subset \mathbb{F}^*_{q^{12}}$; $f_{\ell,Q}(P)$ is a normalized rational function with divisor $(f_{\ell,Q}) = \ell(Q) - ([\ell]Q) - (\ell - 1)(\mathcal{O})$ and $l_{Q_1,Q_2}(P)$ is the line equation corresponding to $Q_1 + Q_2 \in \mathbb{G}_2$ evaluated at $P \in \mathbb{G}_1$.

Definition 3 (`SM9_P256` *Curve*) The `SM9_P256` curve is a 256-bit prime order curve instantiated in the BN curve family parameterized as follows:

```
x = 60000000 0058F98A, b = 5
q = B6400000 02A3A6F1 D603AB4F F58EC745
    21F2934B 1A7AEEDB E56F9B27 E351457D
p = B6400000 02A3A6F1 D603AB4F F58EC744
    49F2934B 18EA8BEE E56EE19C D69ECF25
t = D8000000 019062ED 0000B98B 0CB27659
```

Note that the `SM9_P256` curve admits an M-type twist, i.e., $E' : y^2 = x^3 + b \cdot \xi$.

Algorithms in SM9

Definition 4 (*SM9-IBS*) The SM9-IBS signature scheme consists of following four PPT algorithms $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$.

$\mathsf{Setup}(1^\kappa) \rightarrow (\mathsf{msk}, \mathsf{mpk})$. On input the security parameter κ, the algorithm runs as follows:

1. Generate bilinear pairing groups $\mathcal{G} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, P_1, P_2, e) \leftarrow \mathsf{GroupGen}(1^\lambda)$.
2. Pick a random $s \in \mathbb{Z}_p^*$, compute $P_{pub\text{-}s} = [s]P_2$.
3. Compute $g = e(P_1, P_{pub\text{-}s})$.
4. Pick a cryptographic hash function H_v and a one byte identifier *hid*. GM/T 0044-2016.5 (GM/T 2016b) requires SM3 (GM/T 2012) as the hash function and $hid = 1$.
5. Output a master secret key $msk = s$ and a master public key $mpk = (\mathcal{G}, P_{pub\text{-}s}, H_v, hid)$.

$\mathsf{KeyGen}(\mathsf{mpk}, \mathsf{msk}, \mathsf{ID_A}) \rightarrow ds_A$. On input a pair of master keys (*mpk*, *msk*) and an identity $\mathsf{ID_A}$, this algorithm generates a signing key ds_A as follows:

1. Compute $t_1 = \mathsf{H2RF}_1(H_v, \mathsf{ID_A} \| hid, p) + s$.
2. Compute $t_2 = s \cdot t_1^{-1}$.
3. Compute a signing key as $ds_A = [t_2]P_1$.

$\mathsf{Sign}(\mathsf{mpk}, \mathsf{ds_A}, \mathsf{M}) \rightarrow (\mathsf{h}, \mathsf{S})$. On input a master public key *mpk*, a signing key ds_A and a message M, it generates a signature (h, S) as follows:

1. Pick a random $z \in \mathbb{Z}_p^*$.
2. Compute $w = g^z$.
3. Compute $h = \mathsf{H2RF}_2(H_v, M \| w, p)$.
4. Compute $l = (z - h) \mod p$.
5. Compute $S = [l]ds_A$.
6. Output a signature (h, S).

$\mathsf{Verify}(\mathsf{mpk}, \mathsf{ID_A}, \mathsf{M}, (\mathsf{h}, \mathsf{S})) \rightarrow \{0, 1\}$. On input a master public key *mpk*, a signer's identity $\mathsf{ID_A}$, a message M and a signature (h, S), this algorithm verifies the signature as follows:

1. Compute $h_1 = \mathsf{H2RF}_1(\mathsf{H_v}, \mathsf{ID_A} \| \mathsf{hid}, \mathsf{p})$.
2. Compute $P = [h_1]P_2 + P_{pub\text{-}s}$.
3. Compute $u = e(S, P)$.
4. Compute $t = g^h$.
5. Compute $w' = u \cdot t$.
6. Compute $h_2 = \mathsf{H2RF}_2(\mathsf{H_v}, \mathsf{M} \| \mathsf{w}, \mathsf{p})$.
7. If $h = h_2$, return 1. Otherwise, return 0.

Definition 5 (*SM9-IBE-KEM*) The SM9-IBE-KEM is an identity-based key encapsulation mechanism (KEM) consists of following four PPT algorithms $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Encap}, \mathsf{Decap})$.

$\mathsf{Setup}(1^\kappa) \rightarrow (\mathsf{msk}, \mathsf{mpk})$. Similar to Definition 4, it outputs $msk = s$ and $mpk = (\mathcal{G}, P_{pub-e}, H_v, hid)$. Note that $P_{pub-e} = [s]P_1 \in \mathbb{G}_1$ and $hid = 3$.

$\mathsf{KeyGen}(\mathsf{mpk}, \mathsf{msk}, \mathsf{ID_A}) \rightarrow \mathsf{de_A}$. Similar to Definition 4, it outputs a private key $de_A = [t_2]P_2 = [s \cdot t_1^{-1}]P_2 = [s \cdot (\mathsf{H2RF}_1(H_v, \mathsf{ID_A} \| hid, p) + s)^{-1}]P_2$.

$\mathsf{Encap}(\mathsf{mpk}, \mathsf{ID_A}) \rightarrow (\mathsf{K}, \mathsf{C})$. On input a master public key *mpk* and a user identity $\mathsf{ID_A}$, this algorithm runs as follows:

1. Pick a random $z \in \mathbb{Z}_p^*$.
2. Compute $w = g^z$.
3. Compute $h_1 = \mathsf{H2RF}_1(H_v, \mathsf{ID_A} \| hid, p)$.
4. Compute $P = [h_1]P_1 + P_{pub-e}$.
5. Compute $C = [z]P$.
6. Derive a data encapsulation (DEM) key $K = \mathsf{KDF2}(H_v, C \| w \| \mathsf{ID_A}, \ell)$.
7. Output (K, C), where K is the derived key and C is the KEM ciphertext.

$\mathsf{Decap}(\mathsf{mpk}, \mathsf{ID_A}, \mathsf{de_A}, \mathsf{C}) \rightarrow \mathsf{K}$. On input a master public key *mpk*, a user identity $\mathsf{ID_A}$, the corresponding key de_A and a KEM ciphertext C, this algorithm runs as follows:

1. Compute $w = e(C, de_A) = g^z$.
2. Recover a data encapsulation (DEM) key $K = \mathsf{KDF2}(H_v, C \| w \| \mathsf{ID_A}, \ell)$.
3. Output the derived key K.

Definition 6 (*SM9-KA*) The SM9-KA is an identity-based key agreement and consists of the following four operations (Setup, KeyGen, Message Exchange, Session Key Generation) and an optional operation Session Key Confirmation.

$\mathsf{Setup}(1^\kappa)$. This algorithm runs the same as in Definition 5, except that it sets $hid = 2$.

$\mathsf{KeyGen}(\mathsf{mpk}, \mathsf{msk}, \mathsf{ID_A})$. Same as in Definition 5.

MessageExchange. Entities A and B exchange two (or three) rounds of messages:

$$
\begin{aligned}
A \rightarrow B : &R_A = [z_A]P_B \\
B \rightarrow A : &R_B = [z_B]P_A, S_B \\
A \rightarrow B : &S_A
\end{aligned}
$$

where $z_A, z_B \in \mathbb{Z}_p^*$ are two randoms by A and B respectively, $P_i = [\mathsf{H2RF}_1(H_v, \mathsf{ID_i}\|hid, p)]P_1 + P_{pub-e}$ for $i \in \{A, B\}$ and S_A, S_B are the optional key confirmation messages.

Session Key Generation. Upon receiving R_A, R_B, A and B can derive their session keys as follows:

1. Entity A computes three-tuple (g_1, g_2, g_3)
$$g_1 = e(R_B, de_A), g_2 = g^{z_A}, g_3 = g_1^{z_B}$$
2. Similarly, entity B computes (g_1', g_2', g_3')
$$g_1' = g^{z_B}, g_2' = e(R_A, de_B), g_3' = {g_1'}^{z_A}$$
3. ℓ-bit session key is:
$$SK = \mathsf{KDF2}(H_v, \mathsf{ID_A}\|\mathsf{ID_B}\|R_A\|R_B\|g_1\|g_2\|g_3, \ell)$$

Session Key Confirmation (optional). Entities A and B exchange S_A, S_B for key confirmation.

1. Entity B computes key confirmation $S_B = H_v(0x82\|g_2'\|H_v(g_1'\|g_3'\|\mathsf{ID_A}\|\mathsf{ID_B}\|R_A\|R_B))$.
2. Entity A computes key confirmation $S_A = H_v(0x83\|g_1'\|H_v(g_2'\|g_3'\|\mathsf{ID_A}\|\mathsf{ID_B}\|R_A\|R_B))$.

GPGPU and CUDA

Modern GPUs support general purpose computation (GPGPU) in the Single Instruction, Multiple Threads (SIMT) fashion, among which, NVIDIA's *Compute Unified Device Architecture (CUDA)* is the most popular with dozens of *streaming multiprocessors (SMs)* each capable of executing up to 256 threads in parallel. The threads are further grouped into *blocks*, and a GPU function (called *kernel*) may execute with multiple blocks across multiple SMs, effectively exploiting GPU's processing power. From a hardware perspective, CUDA SMs schedule and run thread blocks in a 32-thread unit called *warp*. A warp executes one common instruction per cycle, therefore full efficiency is achieved only when all threads of a warp agree on their execution paths. As a result, CUDA programs should reduce the use of data-dependent branches for the maximum concurrency.

Design of GAPS

In this section, we first introduce the architecture of GAPS, then present our concrete optimization ideas for the SM9 algorithms.

System architecture

The architecture of GAPS is presented in Fig. 2. Upon receiving some requests from clients, GAPS first preprocesses them and pushes the requests into several first in, first out (FIFO) task queues. Note that we maintain 9 individual queues for each of the 9 algorithms in SM9-KA/IBS/IBE[1] (except Setup, which are only executed once during the system's initialization). A dispatcher repetitively checks the status of the queues and retrieves active tasks from them. Depending on the scale of the tasks and the availability of hardware resources, the dispatcher thread adopt different strategies for work dispatching. Generally speaking, when the size of tasks is *lightweight* for a CPU thread, it directly invokes the CPU implementation of SM9 for fast processing. However, when the size of tasks becomes too *heavyweight* for CPU, it sends the workload to GPU for batch processing instead. By properly balancing the task scheduling between CPU and GPU, GAPS can seamlessly scale from zero to up to millions of operations per second.

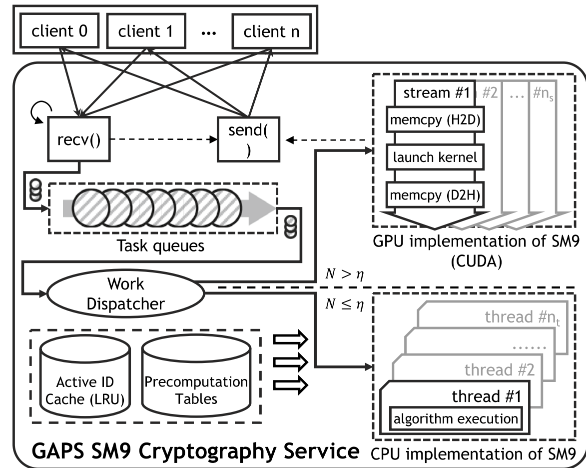

Fig. 2 The system architecture of GAPS

System optimizations. GAPS adopts two optimization techniques at the system level.

- **Active ID Cache.** Many algorithms in SM9 produce intermediate values that are dependent on users' IDs. This overhead can be reduced by pooling ID-related intermediate values in a Last Recently Used (LRU) cache. When a new ID is requested, GAPS computes fresh values for it and stores the intermediate results in the LRU cache. Subsequently, the next time the same ID is requested, GAPS directly loads it from the cache to avoid repetitive computations.
- **Precomputation Table.** For values that rely on fixed parameter in the system (e.g., master public key), we can precompute a table of intermediate values for

[1] Note that although the KeyGen algorithms in SM9-IBE & SM9-KA share similar structure, they cannot use the same queue since using the same set of master keys is insecure.

it. Specifically, GAPS generates multiple precomputation tables for the fixed parameters in its offline time and stores it locally. When evaluating the values online, GAPS loads the precomputed tables into memory to save the computations.

- **Parallelization.** If an SM9 algorithm computes independent intermediate values, we further apply parallelization to reduce its latency.

Heterogeneous implementation platforms. GAPS features GPU-based and CPU-based SM9 implementations for different types of workloads.

- **Batch workload.** The GPU implementation exploits its massive threads to process the batch workload in parallel, i.e., each GPU thread computes a full instance of the algorithm (data parallelism). The key of this implementation is to maximize the parallelism and throughput on the GPU platform. In "Implementation of GAPS " section, we show how we combine mulitple optimized algorithm and implementation techniques to maximize the GPU implementation's throughput.
- **Sporadic workload.** The CPU implementation serves as the fallback option for processing sporadic tasks. Concretely, we first implement the SM9_P256 curve in the RELIC toolkit (Aranha et al. 2014), then implement the SM9 algorithms by applying the same system optimizations as our GPU implementation.

Work dispatching strategy. The work scheduling strategy relies on a threshold paramter η. When the size of tasks N is not greater than η, the worker thread dispatches it to CPU implementation. When N exceeds η, the worker thread sends the data to GPU, waits for GPU's kernel execution, then fetches the results back from GPU. In "Performance of SM9 algorithms" section, we conduct experimental analysis to find out the optimal scheduling threshold η.

Concrete optimization ideas

In this section, we analyze the workload of the SM9 algorithms ("Algorithms in SM9" section) and discuss how the optimizations in "System architecture" section can be applied.

SM9 Key Generation. The private keys of SM9-IBS/IBE/KA share the same structure $[\frac{s}{h_1+s}]P$ so their $\mathsf{K}eyGen$ algorithms can adopt the same optimizations. Specifically, the $\mathsf{K}eyGen$ algorithms first evaluate a hash $h_1 = \mathsf{H2RF}_1(H_v, ID_A\|hid, p)$ for the input ID via the $\mathsf{H}2RF_1$ function and stores it in the active ID cache, then computes a value $t_2 = \frac{s}{h_1+s}$, which requires 1 addition, 1 inversion and 1 multiplication in the field $\mathbb{F}_p$. After that, the algorithm computes a single point multiplication with $P_1 \in \mathbb{G}_1$ or $P_2 \in \mathbb{G}_2$. As P_1, P_2 are part of the system-wide master public keys that are fixed before the execution of $\mathsf{K}eyGen$, we can generate two precomputation tables containing the intermediate point values (e.g., $2P, 3P, 4P, \cdots$) in the offline phase, then use the precomputed values in the online phase for acceleration.

SM9-IBS. It has the following computations:

- Signature generation: the $\mathsf{S}ign$ algorithm first generates a random in $\mathbb{F}_p^*$, then computes a fixed-base exponentiation $w = g^z$ in the extension field $\mathbb{F}_{q^{12}}$ ($\mathbb{G}_T$). Subsequently, it evaluates a hash h_2 for the message M and the $\mathbb{F}_{q^{12}}$ element w via $\mathsf{H}2RF_2$, then computes 1 subtraction in the field $\mathbb{F}_p$. Finally, it computes 1 point multiplication with ds_A in $\mathbb{G}_1$. If the user uses a fixed key for signatures, we can further generate precomputation table for it for acceleration.
- Signature verification: to verify a signature (h, S), Verify first evaluates the identity hash h_1 with $\mathsf{H}2RF_1$. It then computes $P = [h_1]P_2 + P_{pub\text{-}s}$, which involves 1 fixed-point multiplication and a point addition operation in $\mathbb{G}_2$. Note that since the value of P is completely dependent on the user's identity and the system's public key P_{pub-s}, we can store it in the active ID cache to save future computations. After that, the algorithm evaluates a pairing for (S, P), computes a fixed-base exponentiation $t = g^h$ in $\mathbb{F}_{q^{12}}$ and restores w' via a multiplication in $\mathbb{F}_{q^{12}}$. Finally, it evaluates the hash h_2 for M, w and compares it with the h part of the signature. Note that computation of the 4th step ($t = g^h$) can be performed in parallel with steps 1–3 to further reduce the latency.

SM9-IBE-KEM. Its computations include:

- Key encapsulation: the Encap algorithm first picks a random in $\mathbb{F}_p^*$, then computes a fixed-base exponentiation $w = g^z$. Next, it evaluates $P = [h_1]P_1 + P_{pub-e}$ which is also an intermediate value that can be stored in the LRU cache. After that, the algorithm computes a point multiplication in $\mathbb{G}_1$ and uses $\mathsf{K}DF2$ to derive the section key. Note that the evaluation of g^z in step 2 of Encap is independent with steps 3–4 and can be performed concurrently.
- Key decapsulation: the Decap algorithm evaluates paring for (C, de_A) then uses $\mathsf{K}DF2$ to derive the encapsulated key.

SM9-KA. For SM9-KA, we focus on the computation of entity B, as in client/server mode the server only accepts connections from clients.

Table 1 Overhead of the SM9 algorithms and the application of the optimizations in "System architecture" section

Scheme	Algo.	Total cost	Optimizations		
			Cache (I)	Precomp. (II)	Parallel. (III)
SM9-IBS	KeyGen	$1H_1 + 1PM^{fix}_{\mathbb{G}_1} + 1Add_{\mathbb{F}_q} + 1Inv_{\mathbb{F}_q} + 1Mul_{\mathbb{F}_q}$	×	P_1	×
	Sign	$1H_2 + 1Exp^{fix}_{\mathbb{G}_T} + 1PM_{\mathbb{G}_1} + 1Add_{\mathbb{F}_q}$	×	g	×
	Verify	$1H_1 + 1H_2 + 1BP + 1PM^{fix}_{\mathbb{G}_2} + 1Add_{\mathbb{G}_2} + 1Exp^{fix}_{\mathbb{G}_T} + 1PM_{\mathbb{G}_T}$	h_1, P	g, P_2	$h_1 \to P \to u \parallel t$
SM9-IBE (KEM)	KeyGen	$1H_1 + 1PM^{fix}_{\mathbb{G}_2} + 1Add_{\mathbb{F}_q} + 1Inv_{\mathbb{F}_q} + 1Mul_{\mathbb{F}_q}$	×	P_2	×
	Encap	$1H_1 + 1K_2 + 1PM^{fix}_{\mathbb{G}_1} + 1PM_{\mathbb{G}_1} + 1Add_{\mathbb{G}_1} + 1Exp^{fix}_{\mathbb{G}_T}$	h_1, P	g, P_1	$h_1 \to P \to C \parallel w$
	Decap	$1K_2 + 1BP$	×	×	×
SM9-KA	KeyGen	$1H_1 + 1PM^{fix}_{\mathbb{G}_2} + 1Add_{\mathbb{F}_q} + 1Inv_{\mathbb{F}_q} + 1Mul_{\mathbb{F}_q}$	×	P_2	×
	MsgExch	$1H_1 + 1PM_{\mathbb{G}_1} + 1Add_{\mathbb{G}_1}$	h_1, P	P_1	×
	SessKeyGen	$1K_2 + 1BP + 1Exp^{fix}_{\mathbb{G}_T} + 1Exp_{\mathbb{G}_T}$	×	g	×

†H_i = H2RF$_i$, $K_2 = KDF2$, PM=Point Multiplication, Exp=Exponentiation, BP=Bilinear Pairing, Inv=Inversion

- Message exchange: in this phase, the server picks a random $z_B \in \mathbb{F}_p^*$ and computes a point multiplication with P_A. Note that P_A is an intermediate value that depends on the client's ID_A, which can be stored in the LRU cache.
- Session key generation: in this phase, the server first evaluates a pairing with (R_A, de_B), then computes two exponentiations in $\mathbb{F}_{q^{12}}$. Note that the first exponentiation uses a fixed-based g and can be optimized with precomputation. After that, the server derives a key with $KDF2$.

We present the overhead and our optimizations of SM9-KA/IBS/IBE-KEM in Table 1. Concretely, GAPS maintains a $\langle h_1, P\rangle$ pair for each active ID in the LRU cache. Note that for the three SM9 schemes we maintain different h_1 values as their *hid*-s are different. For precomputation, GAPS should precompute tables for P_1, P_2, g, which are all part of the system's public keys.

Implementation of GAPS

Element representations

Prime field. We store prime field elements in radix-R form, i.e., $a = \sum_{i=0}^{\ell} a_i R^i$. Since CUDA GPUs are 32-bit machines, we select $R = 2^{32}$, so we have $\ell = \lceil log_2(q/R)\rceil = 8$ and each $\mathbb{F}_q$ (also $\mathbb{F}_p$) element in the SM9_P256 curve takes 8 words of storage. Before the computations, we convert all prime field elements to the Montgomery domain (i.e. $\tilde{a} = a \cdot R^{\lceil log_2(q/R)\rceil} \ mod\ q$), so that efficient Montgomery reduction and product algorithms can be applied, then convert the results back to normal form in the end.

Extension Field. The Optimal Ate Paring over BN curve produces final result in $\mathbb{F}_{q^{12}}$, which is constructed through tower extension (Benger and Scott 2010). Specifically, we adopt the $\mathbb{F}_q \to \mathbb{F}_{q^2} \to \mathbb{F}_{q^6} \to \mathbb{F}_{q^{12}}$ towering scheme, where

$$\mathbb{F}_{q^2} = \mathbb{F}_q[u]/(u^2 - \beta), \text{with } \beta = -2,$$
$$\mathbb{F}_{q^6} = \mathbb{F}_{q^2}[v]/(v^3 - \xi), \text{with } \xi = u,$$
$$\mathbb{F}_{q^{12}} = \mathbb{F}_{q^{12}}[w]/(w^2 - v).$$

With this construction, an element in $\mathbb{F}_{q^2}$ is represented as $a + bu$, where a, b are two elements in $\mathbb{F}_q$. An element in $\mathbb{F}_{q^6}$ is represented as $a + bv + cv^2$ with $a, b, c \in \mathbb{F}_{q^2}$. An element in $\mathbb{F}_{q^{12}}$ is represented as $a + bw$ with $a, b \in \mathbb{F}_{q^6}$. For the SM9_P256 curve, the cost for storing the $\mathbb{F}_{q^2}, \mathbb{F}_{q^6}, \mathbb{F}_{q^{12}}$ elements are 16, 48, 96 words, respectively.

Elliptic Curve. We represent elements in $\mathbb{G}_1, \mathbb{G}_2$ using Jacobian coordinates, which support fast formulas for elliptic curve operations. Specifically, each point is represented as a three-tuple (X, Y, Z) with $Z \neq 0$. After the computations, the points can be converted back to projective coordinates $(X, Y, Z) \to (X/Z^2, Y/Z^3)$ for storage. For the SM9_P256 curve, a $\mathbb{G}_1$ element is defined over $E: Y^2 = X^3 + 5Z^6$, with $X, Y, Z \in \mathbb{F}_q$, which takes 24 words of storage. A $\mathbb{G}_2$ element is defined over the twisted curve $E': Y^2 = X^3 + 5\xi \cdot Z^6$, with $X, Y, Z \in \mathbb{F}_{q^2}$, taking 48 words of storage.

Optimal ate pairing

We apply the following optimizations to the evaluation of a pairing (Algorithm 1).

Signed representation of the loop parameter. Observe that in the SM9_P256 curve, we have $\log_2(\ell) = 66$ and hamming weight $w = 16$, so the traditional binary double-and-add execution of the loop would require 66 point doublings and 16 point additions. To further reduce the complexity, we adopt the signed

binary representation of ℓ, i.e., $\ell = \sum_{i=0}^{\log_2(\ell)} k_i 2^i$ where $k_i \in \{-1, 0, 1\}$ with hamming weight $w = 11$, saving 5 point additions. Additionally, we unroll the first iteration of the loop to avoid trivial computations (steps 1–5).

Algorithm 1 Revised algorithm for Optimal Ate Pairing on SM9_P256 curve.

```
   Input: P ∈ G1, Q ∈ G2, ℓ = |6x+2| = Σ_{i=0}^{log2(ℓ)} k_i·2^i,
          k_i ∈ {−1, 0, 1}
   Output: f = a_opt(Q, P)
1  d ← l_{T,T}(P), T ← 2Q
2  if k_i = 1 then
3  |  e ← f · l_{T,Q}(P), T ← T + Q
4  end
5  f ← d · e
6  for i = log2(ℓ) − 2 downto 0 do
7  |  f ← f^2 · l_{T,T}(P), T ← 2T
8  |  if k_i = 1 then
9  |  |  f ← f · l_{T,Q}(P), T ← T + Q
10 |  else if k_i = −1 then
11 |  |  f ← f · l_{T,−Q}(P), T ← T − Q
12 |  end
13 end
14 Q1 ← π_q(Q), Q2 ← π_{q^2}(Q)
15 if x < 0 then
16 |  T ← −T, f ← f̄
17 end
18 d ← f · l_{T,Q1}(P), T ← T + Q1
19 e ← f · l_{T,−Q2}(P), T ← T − Q2
20 f ← f · (d · e)
21 f ← f^{(q^6−1)(q^2+1)(q^4−q^2+1)/p}
22 return f
```

Evaluating the line functions and points. As the curve equation of SM9_P256 is $y^2 = x^3 + 5$, we adopt homogeneous projective coordinates (Costello et al. 2010) for the best performance. For the SM9_P256 curve, the homogeneous curve equation becomes $Y^2Z = x^3 + 5Z^3$. As the SM9_P256 curve admits an M-type sextic twist, the line functions are evaluated at the twisting point $\psi(P) = (x_P w^2, y_p w^3)$. Particularly, the formula for point doubling, addition and line computations for the SM9_P256 curve can be derived as follows.

1. Point Doubling and line evaluation: for $T = (X_1, Y_1, Z_1) \in E'(\mathbb{F}_{q^2})$, one can compute the point doubling $2T = (X_3, Y_3, Z_3)$ with the following formula:

$$X_3 = \frac{X_1 Y_1}{2}(Y_1^2 - 9b'Z_1^2),$$
$$Y_3 = (\frac{1}{2}(Y_1^2 + 9b'Z_1^2))^2 - 27b'^2 Z_1^4,$$
$$Z_3 = 2Y_1^3 Z_1,$$

where $b' = 5\xi$. The line function $l_{T,T}$ evaluated at the twisting point $\psi(P)$ can be computed with the following formula:

$$\begin{aligned} l_{T,T}(P) &= y_P w^3 - \lambda x_P w^2 + (\lambda x_3 - x_1) \\ &= -2Y_1 Z_1 y_p \cdot w^3 + 3X_1^2 x_p \cdot w^2 + (3b'Z_1^2 - Y_1^2) \\ &= -2Y_1 Z_1 y_p \cdot vw + 3X_1^2 x_p \cdot v + (3b'Z_1^2 - Y_1^2) \end{aligned}$$

The above steps cost 3 multiplications, 2 squarings and a few additions in $\mathbb{F}_{q^2}$. Observe that by precomputing $\bar{y}_P = -y_P, x'_p = 3x_P$, we can further save 3 $\mathbb{F}_{q^2}$ additions.

2. Point Addition and line evaluation: for $T = (X_1, Y_1, Z_1)$ and $Q = (x_2, y_2) \in E'(\mathbb{F}_{q^2})$, one can compute the mixed point addition $T + Q = (X_3, Y_3, Z_3)$ with the following formula:

$$X_3 = \lambda(\lambda^3 + Z_1\theta^2 - 2X_1\lambda^2),$$
$$Y_3 = \theta(3X_1\lambda^2 - \lambda^3 - Z_1\theta^2) - Y_1\lambda^3,$$
$$Z_3 = Z_1\lambda^3,$$

where $\theta = Y_1 - y_2 Z_1$ and $\lambda = X_1 - x_2 Z_1$. The line function $l_{T,Q}$ evaluated at the twisting point $\psi(P)$ can be computed with

$$\begin{aligned} l_{T,Q}(P) &= -\lambda y_p \cdot w^3 - \theta x_p \cdot w^2 + (\theta X_2 - \lambda Y_2) \\ &= -\lambda y_p \cdot vw - \theta x_p \cdot v + (\theta X_2 - \lambda Y_2) \end{aligned}$$

The complete formula can be evaluated with 11 multiplications, 2 squarings and a few additions in $\mathbb{F}_{q^2}$. By precomputing $\bar{x}_P = -x_P, \bar{y}_P = -y_P$, 2 $\mathbb{F}_{q^2}$ additions can be saved.

Sparse Multiplications. The results of the line functions are sparse elements in $\mathbb{F}_{q^{12}}$. Actually, by rewriting $l = l_{11}w^3 + l_{01}w^2 + l_{00} = (l_{00} + l_{01}v + \underline{0}v^2) + (\underline{0} + l_{11}v + \underline{0}v^2)w$, one can see that half of its $\mathbb{F}_{q^2}$ elements are zeros. Therefore, we can apply the revised algorithms in Algorithm 2 for the multiplication between a dense element and a sparse element (saving 5 $\mathbb{F}_{q^2}$ multiplications), and the multiplication between two sparse elements (saving 11 $\mathbb{F}_{q^2}$ multiplications).

Computing the Final Exponentiation. The power $\frac{q^{12}-1}{p}$ can be further decomposed into two parts: an easy part $f^{(q^6-1)(q^2+1)}$ that can be computed with cheap multiplications, conjugations and applications of the Frobenius map π_q, and a hard part $f^{(q^4-q^2+1)/p}$ that is computed with the addition chain method in Scott et al. (2009).

Algorithm 2 Multiplications between Dense, Sparse elements in $\mathbb{F}_{q^{12}}$.

```
1  Function Fq12DenseMulSparse(a, b):
     Input: a = a_0 + a_1 w ∈ F_{q^12},
            b = b_0 + b_1 w ∈ F_{q^12}. Particularly,
            a_0 = a_00 + a_01 v, a_1 = a_11 v;
     Output: c = a × b = c_0 + c_1 w ∈ F_{q^12}
2    a_0 b_0 ← Fq6DMul01(b_0, (a_00, a_01, 0))
3    t_0 ← b_10 × a_11, t_1 ← b_11 × a_11, t_2 ← b_12 × a_11
4    a_1 b_1 ← t_1 + t_2 v + t_0 v^2
5    T_0 ← b_0 + b_1, T_1 ← a_0 + a_1
     /* T_1 = a_00 + (a_01 + a_11) v + 0 v^2 */
6    T_0 T_1 ← Fq6DMul01(T_0, T_1)
7    c_1 ← T_0 T_1 − a_0 b_0 − a_1 b_1
8    c_0 ← a_0 b_0 + a_1 b_1 v
9    return c = c_0 + c_1 w
10 return
11 Function Fq12SparseMulSparse(a, b):
     Input: a = a_0 + a_1 w ∈ F_{q^12},
            b = b_0 + b_1 w ∈ F_{q^12}. Particularly,
            a_0 = a_00 + a_01 v, a_1 = a_11 v,
            b_0 = b_00 + b_01 v, b_1 = b_11 v;
     Output: c = a × b = c_0 + c_1 w ∈ F_{q^12}
12   a_0 b_0 ← Fq6SMul01(a_0, b_0)
13   t_1 ← b_11 × a_11
14   a_1 b_1 ← t_1 + 0 + 0 v^2
15   T_0 ← b_0 + b_1, T_1 ← a_0 + a_1
     /* Both T_0, T_1 are sparse F_{q^6} elements */
16   T_0 T_1 ← Fq6SMul01(T_0, T_1)
17   c_1 ← T_0 T_1 − a_0 b_0 − a_1 b_1
18   c_0 ← a_0 b_0 + a_1 b_1 v
19   return c = c_0 + c_1 w
20 return
21 Function Fq6DMul01(a, b):
     Input: a = a_0 + a_1 v + a_2 v^2, b = b_0 + b_1 v
     Output: c = a × b = c_0 + c_1 v + c_2 v^2 ∈ F_{q^6}
22   t_0 ← a_0 × b_0
23   t_1 ← a_1 × b_1
24   c_0 ← ((a_1 + a_2) × (b_1) − t_1) ξ + t_0
25   c_1 ← (a_0 + a_1) × (b_0 + b_1) − t_0 − t_1
26   c_2 ← a_2 × b_0 + t_1
27   return c = c_0 + c_1 v + c_2 v^2
28 return
29 Function Fq6SMul01(a, b):
     Input: a = a_0 + a_1 v, b = b_0 + b_1 v
     Output: c = a × b = c_0 + c_1 v + c_2 v^2 ∈ F_{q^6}
30   t_0 ← a_0 × b_0
31   t_1 ← a_1 × b_1
32   c_0 ← (a_1 b_1 − t_1) ξ + t_0
33   c_1 ← (a_0 + a_1) × (b_0 + b_1) − t_0 − t_1
34   c_2 ← t_1
35   return c = c_0 + c_1 v + c_2 v^2
36 return
```

Scalar multiplication and exponentiation

Another major workload in SM9 is the scalar multiplication in the two source groups $\mathbb{G}_1, \mathbb{G}_2$, and the exponentiation in the target group $\mathbb{G}_T$.

The case of unknown scalars

Endomorphism and scalar decompositions. We exploit efficient endomorphisms for acceleration. Specifically, since in pairing-friendly curves we have $E(\mathbb{F}_q): y^2 = x^3 + b$ and $p \equiv 1 \pmod 3$, we can use the GLV endomorphism (Gallant et al. 2001) $\phi: (x, y) \mapsto (\xi x, y)$ in $\mathbb{G}_1$ where $\xi^3 = 1$ and $\xi \in \mathbb{F}_q \backslash \{1\}$. Such endomorphism corresponds to scalar multiplication by a small factor λ_ϕ that satisfies $\lambda_\phi^2 + \lambda_\phi + 1 \equiv 0 \pmod p$. As a result, by applying the GLV endomorphism we can decompose a scalar $k \in \mathbb{F}_p$ into two mini-scalars k_1, k_2 such that $[k_1]P + [k_2]\phi(P) = [k]P$ and $|k_i| \approx |p|/2$. Similar methods can be applied to the scalar multiplication in $\mathbb{G}_2$, by applying the GLS endomorphism (Galbraith et al. 2009) $\psi = \Psi \circ \pi_q \circ \Psi^{-1}$, which gives a 4-dimensional decomposition $[k]Q = [k_1]Q + [k_2]\psi(Q) + [k_2]\psi(Q)^2 + [k_2]\psi(Q)^3$ with $|k_i| \approx |p|/4$, turning the single scalar multiplication $[k]Q$ into a multi-scalar multiplication problem that only has 1/4 the size of $|k|$.

Multi-scalar Multiplications. To compute $\sum_{i=1}^{n}[k_i]P$, a widely used method is to use the Straus-Shamir trick (Ciet et al. 2003) for simultaneous multi-scalar multiplication. However, according to our experiment, such method has extremely low throughput on GPU. This is because the standard binary-and-add algorithm for scalar multiplication introduces data-dependent branches, which significantly reduces the concurrency of GPU's SIMT threads. To overcome this problem, we adopt the Sign-Aligned Column (SAC) representation (Faz-Hernández et al. 2014) using Algorithm 3. It recodes a set of binary scalars $(a_1, \cdots, a_n)$ into SAC forms $(b_1, \cdots, b_n)$, where $b_i \in \{-1, -, 1\}^{\mu+1}$ are signed bit sequences of length $\mu + 1$ that satisfies

$$(1)\ a_i = \sum_{j=0}^{\mu} b_i[j]2^j, \quad \text{for } i \in [1, n],$$

$$(2)\ b_1[j] \in \{-1, 1\}, \quad \text{for } j \in [0, \mu],$$

$$(3)\ b_i[j] \in \{0, b_1[j]\}, \quad \text{for } j \in [0, \mu], i \in [2, n].$$

The first condition (1) guarantees the correctness of the encoding, while conditions (2–3) ensures that the bits of scalars $b_2, \ldots, b_n$ are aligned with b_1. Therefore, in Algorithm 4, each iteration of the loop only computes an addition with a point in the precomputation table, which

removes the data-dependent divergences and ensures full concurrency of GPU's warp execution.

Algorithm 3 Sign-Aligned Column (SAC) recoding of n-dimension scalars.

Input: n non-negative scalars $a_j = \sum_{i=0}^{\mu-1} a_j[i] \cdot 2^i$ where $j \in [1, n]$, $a_j[i] \in \{0, 1\}$ and a_1 is odd; μ denote the scalars' maximum bitlength
Output: n non-negative scalars $b_j = \sum_{i=0}^{\mu} b_j[i] \cdot 2^i$ where $b_1[i] \in \{-1, 1\}, b_k[i] \in \{0, b_1[i]\}$ for $i \in [0, \mu]$ and $k \in [2, n]$ such that $b_j = a_i$

1 $b_1[\mu] \leftarrow 1$
2 **for** $i = 0$ **to** $\mu - 1$ **do**
3 $\quad b_1[i] \leftarrow 2a_1[i+1] - 1$
4 $\quad$ **for** $j = 2$ **to** n **do**
5 $\quad\quad b_j[i] \leftarrow b_1[i] \cdot a_j[0]$
6 $\quad\quad a_j \leftarrow \lceil a_j/2 \rceil - \lceil b_j[i]/2 \rceil$
7 $\quad$ **end**
8 **end**
9 **for** $j = 2$ **to** n **do**
10 $\quad b_j[\mu] \leftarrow a_j[0]$
11 **end**
12 **return** $(b_j[\mu], \cdots, b_j[0])$ *for* $j \in [1, n]$

Algorithm 4 Unknown Point Multiplication using degreen-n endomorphism ψ and SAC scalar encoding.

Input: A point P and an integer scalar $k \in \mathbb{F}_p$
Output: $Q = [k]P$

1 Compute $\psi^i(P)$ for $i \in [1, n-1]$.
2 Precompute $T[u] \leftarrow P + \sum_{i=1}^{n-1}[u_i]\psi^i(P)$ for all $0 \leq u < 2^{n-1}$ where $u = \sum_{i=1}^{n-1} u_i \cdot 2^i$ and $u_i \in \{0, 1\}$.
3 Decompose k into n mini-scalars $(k_1, \ldots, k_n)$.
4 For $k_i < 0$, convert $k_i, \psi^{i-1}(P)$ to $-k_i, -\psi^{i-1}(P)$.
5 Recode $(k_1, \ldots, k_n)$ to $(b_1, \cdots, b_n)$ using Algorithm 3.
6 Compute $d_i = \sum_{j=2}^{n} |b_j[i]| \cdot 2^{j-2}$ for $i \in [0, \mu]$.
7 $Q \leftarrow b_1[\mu] \cdot T[d_\mu]$
8 **for** $i = \mu - 1$ **downto** 0 **do**
9 $\quad Q \leftarrow [2]Q$
10 $\quad Q \leftarrow Q + b_1[i] \cdot T[d_i]$
11 **end**
12 **return** Q

Note that Algorithm 4 can be further accelerated with sliding-window method in steps 6–11. Specifically, for a window width w, we use it to partition the recoded scalars and precompute $T[u] = u'P_0 = u_0\psi(P) + \cdots + u_{n-2}\psi(P)^{n-1}$ for all $u \in [0, 2^{wn-1}]$ and $u' \in \{1, 3, \cdots, 2^w - 1\}$. The loop in steps 8–11 can be then performed by scanning w-bit of d_i. In our experiment, we find that $w = 2$ provides the best performance for GLV in $\mathbb{G}_1$, while $w = 3$ performs the best for GLS in $\mathbb{G}_2$.

Applying to the exponentiation in $\mathbb{G}_T$. The GLS-based scalar decomposition and multi-scalar multiplication method can be easily applied to accelerate the exponentiation g^k in the extension field $\mathbb{G}_T$, where the Frobenius map π_q serves as the endomorphism for acceleration. Particularly, in Algorithm 4, the point additions should be replaced with finite field multiplications, and the point doubling $[2]Q$ should be replaced by a squaring, i.e., g^2. Other steps of the algorithm can be straightforwardly applied to the context of $\mathbb{G}_T$.

The case of known scalars

When the point P is known in advance, we can adopt a large precomputation table for acceleration. Specifically, for a w-width window, we can rewrite $[k]P = \sum_{i=0}^{l-1} k_i 2^{w \times i} \cdot P$ where $l = \lceil \log_2(p)/w \rceil$. With this representation, we then compute and store the points $k_i 2^{w \times i} \cdot P$ for each $k_i \in \{1, \cdots, 2^w - 1\}$, and $i \in [0, l-1]$. When evaluating $[k]P$ in the online phase, the intermediate point values $P_i = k_i 2^{w \times i} \cdot P$ can be obtained from the look-up table, and the scalar multiplication is reduced to l successive additions, i.e., $[k]P = \sum_{i=0}^{l-1} P_i$, significantly accelerating the process.

For the `SM9_P256` curve, we select $w = 8$, so $l = \lceil \log_2(p)/w \rceil = 32$. Each $\mathbb{G}_1$ point contains two $\mathbb{F}_q$ elements, taking 64 B storage. Therefore, the storage cost for precomputation in $\mathbb{G}_1$ is $64\text{ B} \times 32 \times (2^8 - 1) \approx 0.498\ MB$. Similarly, the elements in $\mathbb{G}_2, \mathbb{G}_T$ takes 128 B, 384 B storage, and the costs for storing their precomputation tables are 0.996 MB and 2.988 MB, respectively. Note that our method is similar to Pan et al. (2017), but differs in the choice of the window size w. We revised the choice of w such that the precomputation table can be effectively loaded into GPU's L2 Cache for faster read access, while the choice of w in Pan et al. (2017) yields a large precopmutation table that takes hundreds of megabytes and can cause significant global memory accessing delays.

Low-level implementation

Optimizations. Below we outline several optimizations applied at the hardware and the software level to reduce the latency and improve the throughput of pairing operations. With GPU's SIMT architecture, full throughput is achieved when all threads in an execution unit (warp) agrees on the same execution path. Therefore, the key is to reduce the data-dependent divergence across threads for maximum parallelization.

- *PTX ISA.* We implement arithmetic operations in the prime field using the *extended-precision integer arithmetic instructions* provided in CUDA's PTX-ISA (NVIDIA 2023). Specifically, we use instructions like `addc, subc, madc` to implement multi-precision integer operations in the prime field, and utilize the carry/borrow flags as masks for performing *divergence-free* modular operations.

Table 2 The peak performance of **GAPS** 's GPU & CPU implementation of SM9

Scheme	Algo.	GAPS-GPU			GAPS-CPU (1-core)		
		$T^{\ddagger}$ (op/s)	L (ms)	N (op)	$T^{\ddagger}$ (op/s)	L (ms)	N (op)
SM9-IBS	KeyGen	2,038,070.59 ± 3.13%	8.04	16,384	4,508 ± 4.57%	0.22	1
	Sign	248,239.58 ± 0.48%	131.5	32,768	997 ± 1.81%	1.00	1
	Verify	88,024.53 ± 0.99%	372.3	32,768	326 ± 0.65%	3.06	1
SM9-IBE	KeyGen	550,718.24 ± 1.46%	29.75	16,384	2,062 ± 2.94%	0.48	1
	Encap	238,000.53 ± 2.63%	137.68	32,768	853 ± 1.46%	1.17	1
	Decap	148,260.89 ± 0.11%	110.51	16,384	569 ± 0.44%	1.76	1
SM9-KA	KeyGen	550,348.25 ± 0.93%	29.77	16,384	2,040 ± 2.51%	0.49	1
	MsgExch†	1,137,014.70 ± 1.13%	28.82	32,768	2,778 ± 1.77%	0.36	1
	SKeyGen†	77,996.80 ± 0.94%	420.12	32,768	259 ± 0.51%	3.86	1

† For SM9-KA, MsgExch refers to an entity's computation in Message Exchange, SKeyGen refers to an entity's computation in Session Key Generation + Confirmation

‡ The average peak throughput of the repeated experiments and the results' relative standard deviation

- *Loop Unrolling.* A powerful optimization strategy on GPU is loop unrolling, which reduces conditional branching and improves instruction's throughput. It can be achieved by prepending a `#pragma unroll` macro before a loop, which is automatically expanded during compilation. Note that the unrolling of a loop must be carefully conducted, otherwise it may incur high register pressure that slows down the access to thread local variables. Therefore, we investigated the resource usage of each algorithm and chose to unroll operations in $\mathbb{F}_q$, the Miller Loop and the power of x in $\mathbb{F}_{q^{12}}$.
- *Function Inlining.* Function invocation on GPU brings expensive overhead due to stack variable passing and code jumping, etc. Using the `__forceinline__` macro, we force the compiler to inline operations in $\mathbb{F}_p$ and other utility functions (e.g., data copy, assignment, comparison), thereby removing the penalties due to function invocations in low-level operations.

In "Performance of `SM9_P256` curve" section, we present a comprehensive analysis of how these optimizations have helped in reducing the latency of our GPU implementation.

Random Number Generation. We adopt the techniques used in Dai et al. (2016), Sun et al. (2020b). Specifically, we first load an initial seed from the host CPU (e.g., using `/dev/random`), then load the seed to GPU and implement the Chacha20 DRNG (Mueller 2017) to derive random numbers. The global (distinct) thread ID is used as the counter of Chacha20 DRNG to enable the generation of randoms in parallel (Table 2).

Performance evaluation

In this section, we first evaluate `GAPS` 's performance for SM9. We then evaluate the performance of the `SM9_P256` curve operations on GPU.

System configuration

We conduct the experiments on an Ubuntu 22.04 server, equipped with a 16-core Intel Xeon CPU running at 2.5 GHz, 64 GB RAM and an RTX 3080 GPU (see Table 3). Our GPU code is implemented with CUDA C++ and is compiled using CUDA Toolkit 11.8 with flags `-Xptxas -allow-expensive-optimizations=true, -O3`. The CPU code is implemented using the RELIC toolkit. In particular, we first configure it with `-DARITH=gmp -DFP_PRIME=256 -DWSIZE=64` to use its GMP implementation of the 256-bit prime field operations and set the word size to 64 bits, then compile it using clang-14 with `-O3 -funroll-loops -fomit-frame-pointer -finline-small-functions -march=native -mtune=native` for full optimizations.

Setup. We issue SM9 algorithm tasks to `GAPS` for evaluation. Specifically, during each run of the experiment, we send (batch) requests of size $N \in [2^0, 2^1, 2^2, \ldots, 2^{20}]$ to `GAPS`, then measure the processing latency (including the time for resource allocation, memory transfer and algorithm execution) of the requests in `GAPS`. For each request size N, we repeat the experiment for 10 times to obtain stable results.

Correctness. All SM9 implementations in `GAPS` have been checked against the test vectors in the SM9 standard GM/T (2016b). Additionally, during each run of the experiment, we check the correctness of `GAPS` 's results.

Table 3 Hardware specifications

Spec. GPU	NVIDIA RTX 3080
Core configuration	68 SMs, 8704 cores
Core frequency	1440 MHz - 1710 MHz
L2 Cache	5 MB
Global memory	12 GB (760 GB/s)
Compute capability	8.6
Monetary cost	$1,099.99[1]
Spec. GPU	**Intel Xeon Platinum 8269CY**
Core configuration	16 cores / 32 threads
Core frequency	2.50 GHz
Cache size	35.75 MB
Monetary cost	$1,086.00[2]

[1] Price obtained in USD from Amazon on Mar. 4, 2023

[2] Price obtained in USD from Ebay on Mar. 4, 2023

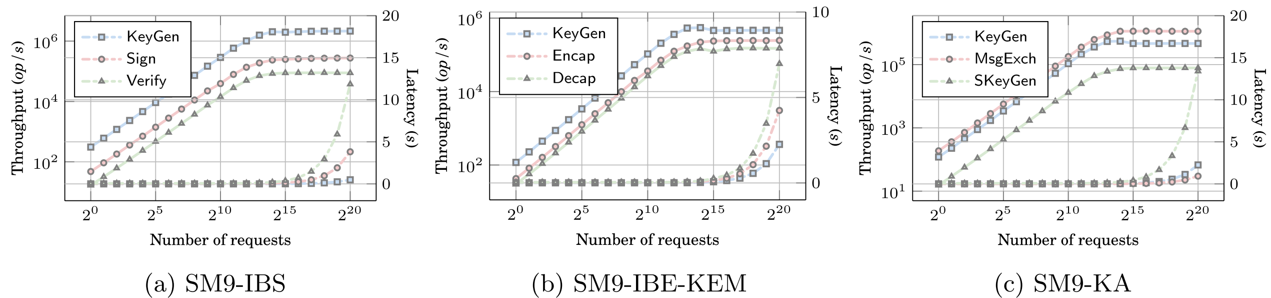

Fig. 3 The performance of `GAPS` 's GPU implementation of SM9 as the input size N grows

Concretely: ① For SM9-IBS, we require that Verify outputs 1 for all valid signatures. ② For SM9-IBE, we require that the keys derived from Encap and Decap are identical. ③ For SM9-KA, we require the produced session keys are identical. These checks guarantee the correctness of `GAPS`.

Metrics. Two performance metrics are considered: *throughput* and *latency*. Latency is the processing time (including the time for resource allocation and memory transfer) of a batch of requests, which is denoted with the symbol L. Throughput is the number of requests processed within a time unit and is denoted with the symbol T. Given batch size N and its latency L, the throughput is calculated with $T = N/L$.

Performance of SM9 algorithms

Throughput

Table 2 gives the peak throughput ($T_{\max}$) of `GAPS` implementations. For SM9-IBS, `GAPS` 's GPU implementation is capable of generating 2,038,070 signing keys, producing 248,239 message signatures or verifying 88,024 signatures per second. For SM9-IBE-KEM, `GAPS` can generate 550,718 user keys, produce 238,001 KEM ciphertexts or decapsulate 159,160 ciphertexts per second. For SM9-KA, `GAPS` can generate 550,348 user keys, generate 1,137,014 exchange messages and 77,996 session keys per second. The difference between the throughput of SM9 algorithms can be verified by our overhead analysis in Table 1. For example, the primary overhead of SM9-IBS's key generation algorithm is the point multiplication in $\mathbb{G}_1$, which is approximately 4 times faster than a point multiplication in $\mathbb{G}_2$ that is the primary overhead of SM9-IBE/KA's key generation. Also, as the Decap algorithm of SM9-IBS-KEM only requires 1 pairing and 1 KDF_2 operation, therefore its performance is significantly better than SM9-IBS's Verify, which requires 1 pairing, 1 point multiplication in $\mathbb{G}_2$ and 1 exponentiation in $\mathbb{G}_T$.

Through analyzing Fig. 3, we can learn how the size of the batch input N affects `GAPS` 's throughput on the GPU platform. Take SM9-IBS.*KeyGen* as an example, its

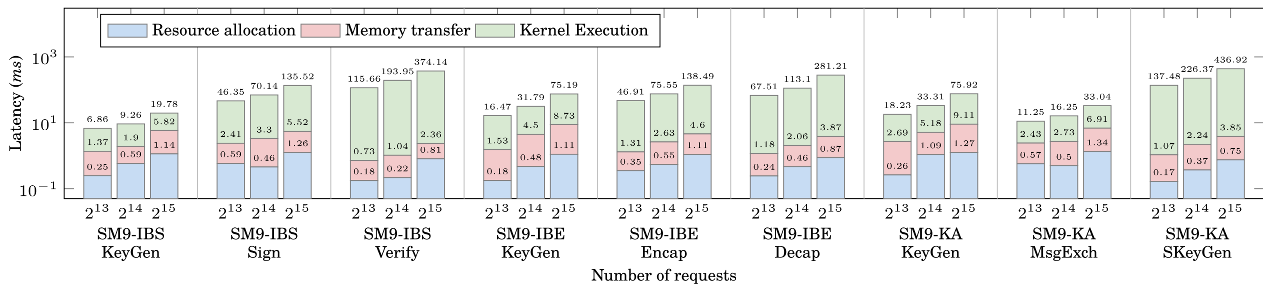

Fig. 4 Execution time breakdown of GAPS 's GPU implementation of SM9

throughput first grows linearly with N, then reaches the peak throughput when the number of batch inputs $N = 16,384$. At this point, the requests are processed with latency $L = 8.04$ ms, so the batch's throughput can be obtained through $T = N/L = 2,038,070$ op/s. The first time an implementation reaches its peak throughput also indicates the full utilization of GPU's resources. After that point, increasing the batch size N no longer raises GPU's throughput, but will make it fluctuate in a certain range. To understand this, suppose we increase the batch size to $N + 1$, which will introduce an additional round of processing as only N requests can be simultaneously processed, taking proportional time $\alpha \cdot L_k, \alpha \in (0, 1)$. Therefore, the throughput will fluctuate in the range $[\frac{N_k+1}{(1+\alpha)\cdot L_k}, \frac{N_k}{L_k}]$ as N keeps growing. Similar trend can be observed for other SM9 algorithms as shown in Fig. 3.

Additionally, we benchmarked the throughput of GAPS 's CPU implementation on a single core. As shown in Table 2, it only takes GAPS-CPU 0.22/1.00/3.06 ms to process one SM9-IBS key generation, signature generation and verification request, 0.48/1.17/1.76 ms to process one SM9-IBE-KEM key generation, key encapsulation and decapsulate request, or 0.49/0.36/3.86 ms to process one SM9-KA key generation, message exchange and session key generation request. The results show that GAPS-CPU can efficiently handle small number of requests in its idle mode.

Execution Time Analysis. To find out which operation is the most expensive, we further conduct a breakdown analysis of GAPS-GPU's execution time. We split an execution into three stages: resource allocation, memory transfer and kernel execution, as illustrated in Fig. 4. Specifically, resource allocation spans a duration of 0.17∼1.34 ms across varying input sizes, while memory transfer between CPU & GPU ranges from 0.55∼7.85 ms. Nevertheless, these two stages collectively account for only 0.53%∼29.40% of the total execution time. Kernel execution on GPU, which takes around 70.60∼99.46% of the time, is still the most expensive stage in GAPS-GPU.

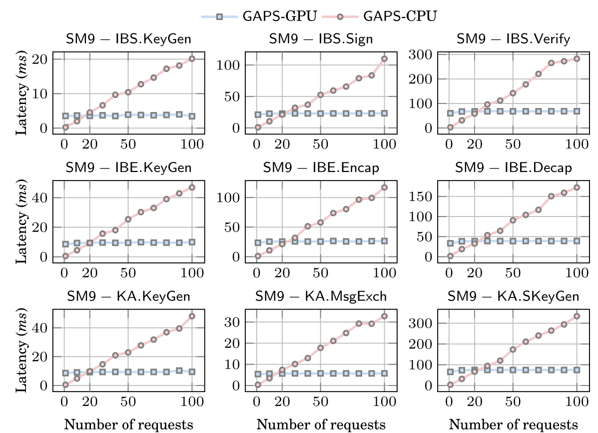

Fig. 5 The latency of GAPS 's GPU and CPU implementations for small request sizes

Finding the optimal scheduling threshold

According to "System architecture" section, GAPS relies on a threshold value η for its task scheduling. To determine the optimal threshold η, we measure the latency of GAPS 's CPU and GPU implementation for processing small amount of requests (i.e., $N \in [1, 10, 20, \cdots, 100]$). As shown in Fig. 5 the latency of GAPS-CPU grows linearly, while the latency of GAPS-GPU stays almost constant due to parallelization on the GPU platform. Therefore, we select the first batch size $N_t = 20$ that satisfies $L_{CPU} > L_{GPU}$ as the threshold. Starting from this point, processing $N \geq N_t$ requests with GPU becomes more efficient than CPU.

Performance of SM9_P256 curve

In this section, we evaluate the performance of GAPS 's GPU implementation of the SM9_P256 curve. Our goal is to verify the effectiveness of our optimization techniques in "Implementation of GAPS " section.

Throughput of curve operations. We benchmark the performance of SM9_P256 curve operations on

Table 4 The peak performance of the `SM9_P256` curve operations on GPU

Operation	Peak throughput		
	$T^{\dagger}$ (op/s)	L (ms)	N (op)
Pairing	158,991.15 ± 0.14%	103.05	16,384
$\mathbb{G}_1$ PM	2,585,630.99 ± 0.50%	25.35	65,536
$\mathbb{G}_1$ fixed PM	26,870,530.82 ± 1.15%	0.61	16,384
$\mathbb{G}_2$ PM	633,402.68 ± 0.26%	25.87	16,384
$\mathbb{G}_2$ fixed PM	5,189,675.14 ± 0.46%	3.16	16,384
$\mathbb{G}_T$ Exp.	313,821.32 ± 0.16%	52.40	65,536
$\mathbb{G}_T$ fixed Exp.	936,094.84 ± 0.61%	70.01	65,536

† The average peak throughput obtained from repeated experiments and the results' relative standard deviation

Table 5 Comparison of different algorithms for `SM9_P256` curve operations on GPU

Type	Operation	Latency	Speedup
e	Miller Loop (basic)	9.15 ms	–
	Miller Loop (Sec 4.2)	8.37 ms	1.09×
	Final Exp. (basic)	6.37 ms	–
	Final Exp. (Sec 4.2)	5.45 ms	1.16×
	Pairing	15.60 ms	–
$\mathbb{G}_1$	Scalar Mul. (basic)	5.33 ms	–
	Scalar Mul. (wNAF)	8.72 ms	0.61×
	Scalar Mul. (Sec 4.3)	2.73 ms	1.95×
$\mathbb{G}_2$	Scalar Mul. (basic)	12.12 ms	–
	Scalar Mul. (wNAF)	20.36 ms	0.60×
	Scalar Mul. (Sec 4.3)	3.54 ms	3.42×
$\mathbb{G}_T$	Scalar Exp. (basic)	36.78 ms	–
	Scalar Exp. (fast-sqr)	30.29 ms	1.21×
	Scalar Exp. (Sec 4.3)	9.60 ms	3.83×

GPU by issuing inputs of size $N \in [2^0, 2^1, \cdots, 2^{16}]$. As shown in Table 4, `GAPS` can compute 15.9k pairings, 2.6M/26.9M unknown/fixed point multiplications in $\mathbb{G}_1$, 633.4k/5.2M unknown/fixed point multiplications in $\mathbb{G}_2$, and 313.8k/936.2k unknown/fixed scalar exponentiations in $\mathbb{G}_T$. The results indicate that `GAPS` 's GPU implementation of the `SM9_P256` curve not only improves the performance of the SM9 cryptography schemes, but also shows huge potential in its application to many pairing-based protocols (e.g., attribute-based encryption) afflicted by bottlenecks in elliptic curve and pairing operations.

Latency of curve operations. In Table 5, we provide the latencies of running the curve operations with a single warp (32×1 threads), which is the minimal unit for thread scheduling on CUDA GPUs. As the table shows, `GAPS` can evaluate an optimal ate pairing on the `SM9_P256` curve in 15.60 ms. By studying its individual algorithms, we find that our optimization methods in "Optimal ate pairing" section provides 1.09×, 1.16× speedups for the Miller Loop and the Final Exponentiation. For the scalar multiplications in $\mathbb{G}_1, \mathbb{G}_2$, the GLV/GLS-based decomposition methods in "The case of unknown scalars" section provide 1.95×, 3.42× speedups over the basic binary-and-add algorithm. We also implemented the windowed-NAF (wNAF) algorithm on GPU, and found that its latency becomes 0.61×, 0.60× worse than the basic implementation, as wNAF introduces more data-dependent divergences. Finally, for the scalar exponentiation in $\mathbb{G}_T$, we first evaluated the basic square-and-multiply method with Granger-Scott's fast squaring formula (Granger and Scott 2010), and found that it is 1.21× faster than the basic method. Nevertheless, our GLS-based method is even faster, as it provides 3.83× speedup over basic method. Overall, the optimization techniques in "Optimal ate pairing" and "Scalar multiplication and exponentiation" sections have successfully reduced the latency of these operations to *milliseconds level* on GPU.

Effectiveness of the optimization methods. To figure out the effectiveness of our optimization techniques in "Low-level implementation" section, we conduct ablation studies on the `SM9_P256` curve. Specifically, we first reduce to a baseline implementation without any optimizations in "Low-level implementation"

Table 6 The latencies (microseconds) for performing low-level operations on `SM9_P256` curve on GPU

Optimization	$\mathbb{F}_q$			$\mathbb{F}_{q^2}$		$\mathbb{F}_{q^6}$		$\mathbb{F}_{q^{12}}$ ($\mathbb{G}_T$)		$\mathbb{G}_1$		$\mathbb{G}_2$	
	Add.	Mul.	Sqr.	Mul.	Sqr.	Mul.	Sqr.	Mul.	Sqr.	Add.	Dbl.	Add.	Dbl.
Original	3.26	94.46	39.97	254.30	148.80	1,399.34	921.63	3,929.60	3,113.22	1,003.32	367.19	3,315.69	1,334.51
+ PTX-ISA	2.1×	2.6×	2.3×	2.2×	2.0×	2.1×	2.1×	1.9×	2.0×	2.3×	2.3×	2.2×	2.3×
+ Unrolled loop	9.9×	8.4×	4.9×	7.0×	5.5×	5.3×	4.7×	5.1×	5.3×	6.8×	4.9×	6.3×	5.3×
+ Inline function	1.9×	2.1×	1.8×	2.2×	1.9×	2.4×	2.0×	2.2×	2.0×	2.1×	2.1×	2.1×	2.1×
Overall	39.5×	45.9×	20.3×	33.9×	20.9×	26.7×	19.7×	21.3×	21.2×	32.8×	23.7×	29.1×	25.6×
Result	0.08	2.08	1.99	4.67	4.11	53.25	42.83	178.04	142.60	30.60	15.23	77.25	40.90

Table 7 Performance comparison with related work

Scheme	Platform	Algorithm(s)	Curve	throughput (op/s)
Wang et al. (2019)	Xilinx FPGA Virtex-7	Pairing	SM9_P256	291
Xie et al. (2022)	ASIC 90 nm	Pairing	SM9_P256	10,000
Hu et al. (2022)	Intel Core i7-6500U	Pairing	SM9_P256	295
Aranha et al. (2011)	AMD Phenom II X4 940	Pairing	BN254	1,923
Cheung et al. (2011)	Xilinx FPGA Virtex-6	Pairing	BN254	1,745
Pu and Liu (2013)	GTX 680	Pairing	BN254	3351
Pan et al. (2017)	GTX 780 Ti	ECDSA-{Sign,Verify}	NIST_P256	8,710,000 / 929,000
Hu et al. (2023)	GTX 3060	Pairing	BN254	43,856
		IBS-{Sign,Verify}		322,773 / 40,643
GmSSL (2023)	Intel Xeon Platinum 8269CY	Pairing	SM9_P256	8
		SM9-IBS		65 / 5 / 3
		SM9-IBE		20 / 4 / 4
		SM9-KA		19 / 29 / 2
OLYM (2022)	SJJ1631-HSM	SM9-{KeyGen,Encap,Sign}	SM9_P256	90,000 / 27,000 / 30,000
GAPS (ours)	RTX 3080	Pairing	SM9_P256	158,991
		SM9-IBS		2,038,071 / 248,240 / 88,025
		SM9-IBE		550,718 / 238,002 / 148,262
		SM9-KA		550,348 / 1,137,015 / 77,997

section, then consecutively apply the optimizations to observe their effectiveness. According to Table 6, using PTX-ISA introduces 1.9×–2.6× speedups over the baseline, while loop unrolling contributes most to the speedups. The final optimized result shows 19.7×–45.9× speeupds over the baseline, performing low-level operations at the microseconds level.

Comparison with related work

We compare with three types of related works (Table 7). The first type of works optimizes the performance of bilinear pairing on particular platforms. Among them, Xie et al. (2022) reported the highest pairing throughput (10,000 op/s) for the SM9_P256 curve, while Aranha et al. (2011) reported the fastest CPU implementation (1,923 op/s) of the BN254 pairing. Compared with these works, GAPS-GPU's implementation can compute 158,991 pairings per second, which is at least 15.9× faster than previous works.

The second type of works implements other elliptic curve or pairing based algorithms on GPU. Pan et al. (2017) introduced a GPU-accelerated signature server and reported over millions of operations per second for ECDSA. Although their results are superior, it's important to know that ECDSA is based on non-pairing curves and thereby their work cannot be applied to SM9. Hu et al. (2023) proposed a GPU-based implementation of the identity-based signature scheme specified in IEEE (2013) that can generate 322,773 signatures or verify 40,643 signatures per second. Compared to them, GAPS focuses on the implementation of the entire SM9 cipher suites. It reports the highest throughput (158,991 op/s) for 256-bit pairings and can verify 88,025 SM9 signatures per second, outperforming existing implementations by 2.0×–3.6×.

The third type of works implements the entire SM9 cipher suites. GmSSL (2023) is the most popular open-source implementation. We compile GmSSL and run it locally in our environment. The results in Table 7 show that GmSSL can only process 19–65 key generation requests and a few signature generation & encryption requests per second. Another known implementation of SM9 is the commercial hardware security module (HSM) by OLYM (2022). According to its document, SJJ1631-HSM can handle 90,000 key generation requests, 27,000 signature generation requests and 30,000 encryption requests per second. Nevertheless, GAPS 's throughput is even higher, as GAPS-GPU outperforms them by at least 6.1×, showing the efficacy of GAPS 's design.

Discussion and future work

Security of GAPS **.** As a dedicated cryptography service, GAPS offers higher security and robustness guarantees. First, GAPS operates independently of the server's computational resources, ensuring the resilience of the system against potential Denial-of-Service (DoS) attacks. This also facilitates the service's scalable deployment, as more computing resources can be dynamically allocated to adapt to the system's workload. Moreover, the isolated nature of CaaS offers an additional layer of defense

against external threats, as the publicly exposed server no longer holds sensitive materials like private keys. This mitigates the security risks of many server-side vulnerabilities (e.g., OpenSSL Heartbleed Synopsys, Inc (2016)) and side-channel attacks like Spectre (Kocher et al. 2019) and Meltdown (Lipp et al. 2018). Finally, due to the application of the sign-aligned column recoding ("The case of unknown scalars" section), the multiplication and exponentiation with secret scalars are constant-time, protecting the implementations from side-channel timing attacks. In summary, GAPS is a secure and robust solution for SM9 in many large-scale applications.

Future extension of GAPS. Currently, GAPS only implements the SM9_P256 and the identity-based cryptographic schemes in SM9. It can be seamlessly extended to support other pairing-friendly curves in the BN family (Definition 1), or slightly modified to support other pairing curve families (e.g., Barreto et al. 2002). In the future, GAPS can be migrated to accelerate other pairing-based cryptography protocols such as attribute-based encryption (Sahai and Waters 2005), searchable encryption (Boneh et al. 2004) and zero knowledge proof systems (Groth 2016).

Conclusion

In this paper, we propose GAPS, a high-performance Cryptography as a Service for SM9. Combined with multiple optimization techniques, GAPS harnesses a heterogeneous computing architecture that dynamically balances the workload between a low-latency CPU implementation and a high-throughput GPU implementation, scaling seamlessly across sporadic inputs and batch inputs. Our evaluation shows that GAPS achieves a scalable performance, making it a practical solution for SM9 in large-scale applications.

Acknowledgements
The authors would like to thank the anonymous reviewers for their helpful comments.

Author contributions
WX and HM participated in the system design and drafted the manuscript. RZ participated in problem discussions and refinement of the manuscript. WX implemented and benchmarked the system. All authors have read and approved the submission of this manuscript.

Funding
This work is supported by National Natural Science Foundation of China (Nos. 62172411, 62172404, 61972094, and 62202458).

Availability of data and materials
The data used in this paper is available from the authors on a reasonable request.

Declarations

Competing interests
The authors declare that they have no competing interests.

Received: 30 November 2023 Accepted: 30 January 2024
Published online: 02 October 2024

References

Al-Riyami SS, Paterson KG (2003) Certificateless Public Key Cryptography. In: ASIACRYPT, Lecture Notes in Computer Science, vol 2894. Springer, pp 452–473

Aranha DF, Gouvêa CPL, Markmann T, et al (2014) The RELIC toolkit. https://github.com/relic-toolkit/relic

Aranha DF, Karabina K, Longa P, et al (2011) Faster Explicit Formulas for Computing Pairings over Ordinary Curves. In: EUROCRYPT, Lecture Notes in Computer Science, vol 6632. Springer, pp 48–68

Barreto PSLM, Lynn B, Scott M (2002) Constructing elliptic curves with prescribed embedding degrees. In: SCN, lecture notes in computer science, vol 2576. Springer, pp 257–267

Barreto PSLM, Naehrig M (2005) Pairing-Friendly Elliptic Curves of Prime Order. In: Selected Areas in Cryptography, Lecture Notes in Computer Science, vol 3897. Springer, pp 319–331

Benger N, Scott M (2010) Constructing tower extensions of finite fields for implementation of pairing-based cryptography. In: WAIFI, Lecture Notes in Computer Science, vol 6087. Springer, pp 180–195

Beuchat J, González-Díaz JE, Mitsunari S, et al (2010) High-speed software implementation of the optimal ate pairing over Barreto–Naehrig curves. In: Pairing, lecture notes in computer science, vol 6487. Springer, pp 21–39

Boneh D, Di Crescenzo G, Ostrovsky R, et al (2004) Public key encryption with keyword search. In: Advances in cryptology-EUROCRYPT 2004: international conference on the theory and applications of cryptographic techniques, Interlaken, Switzerland, May 2-6, 2004. Proceedings 23, Springer, pp 506–522

Cheng Z (2017) The SM9 cryptographic schemes. IACR Cryptol. ePrint Arch. 2017/117

Cheung RCC, Duquesne S, Fan J, et al (2011) FPGA implementation of pairings using residue number system and lazy reduction. In: CHES, lecture notes in computer science, vol 6917. Springer, pp 421–441

Ciet M, Joye M, Lauter KE, et al (2003) Trading inversions for multiplications in elliptic curve cryptography. IACR Cryptol. ePrint Arch. 2003/257

Cook DL, Ioannidis J, Keromytis AD, et al (2005) Cryptographics: secret key cryptography using graphics cards. In: CT-RSA, lecture notes in computer science, vol 3376. Springer, pp 334–350

Costello C, Lange T, Naehrig M (2010) Faster pairing computations on curves with high-degree twists. In: Public Key cryptography, lecture notes in computer science, vol 6056. Springer, pp 224–242

Dai W, Sunar B, Schanck JM, et al (2016) NTRU modular lattice signature scheme on CUDA GPUs. In: HPCS. IEEE, pp 501–508

Entrust (2023) Hardware security modules (HSMs). https://www.entrust.com/digital-security/hsm

Faz-Hernández A, Longa P, Sánchez AH (2014) Efficient and secure algorithms for glv-based scalar multiplication and their implementation on GLV-GLS curves. In: CT-RSA, lecture notes in computer science, vol 8366. Springer, pp 1–27

Galbraith SD, Lin X, Scott M (2009) Endomorphisms for faster elliptic curve cryptography on a large class of curves. In: EUROCRYPT, Lecture notes in computer science, vol 5479. Springer, pp 518–535

Gallant RP, Lambert RJ, Vanstone SA (2001) Faster point multiplication on elliptic curves with efficient endomorphisms. In: CRYPTO, lecture notes in computer science, vol 2139. Springer, pp 190–200

GM/T (2012) 0004-2012 SM3 Cryptogrpahic Hash Algorithm

GM/T (2016a) 0044.1-2016 Identity-Based Cryptographic Algorithms SM9 - Part 1. General

GM/T (2016b) 0044.1-2016 identity-based cryptographic algorithms SM9 - Part 5. Parameter Definition

GmSSL (2023) GmSSL - An open source cryptographic toolkit. https://github.com/guanzhi/GmSSL, accessed: 2023-11-06

Granger R, Scott M (2010) Faster squaring in the cyclotomic subgroup of sixth degree extensions. In: Public Key cryptography, lecture notes in computer science, vol 6056. Springer, pp 209–223

Groth J (2016) On the size of pairing-based non-interactive arguments. In: Fischlin M, Coron JS (eds) Advances in Cryptology - EUROCRYPT 2016. Springer, Berlin, pp 305–326
Hu X, He D, Peng C et al (2022) A fast implementation of Rate pairing in SM9 algorithm. J Cryptol Res 9(5):936–948
Hu X, He D, Luo M et al (2023) High-performance implementation of the identity-based signature scheme in IEEE P1363 on GPU. ACM Trans Embed Comput Syst 22(2):25:1-25:35
IEEE (2013) 1363.3-2013 - IEEE Standard for Identity-based cryptographic techniques using pairings
ISO/IEC (2018) ISO/IEC 14888-3:2018 - IT Security Techniques - Digital Signatures with Appendix - Part 3: discrete logarithm based mechanisms
ISO/IEC (2021) ISO/IEC 18033-5:2021 - Information technology - Security techniques - Encryption algorithms - Part 5: identity-based ciphers
Jang K, Han S, Han S, et al (2011) SSLShader: Cheap SSL acceleration with commodity processors. In: NSDI. USENIX Association
Jing S, Yang X, Feng Y, et al (2022) Hardware implementation of SM9 fast algorithm based on FPGA. In: Proceedings of the 2nd international conference on internet, education and information technology (IEIT 2022). Atlantis Press, pp 797–803
Kocher P, Horn J, Fogh A, et al (2019) Spectre attacks: exploiting speculative execution. In: 40th IEEE symposium on security and privacy (S &P'19)
Lai J, Huang X, He D et al (2022) Provably secure online/offline identity-based signature scheme based on SM9. Comput J 65(7):1692–1701
Lipp M, Schwarz M, Gruss D, et al (2018) Meltdown: reading kernel memory from user space. In: 27th USENIX security symposium (USENIX Security 18)
Mueller S (2017) ChaCha20 DRNG. https://www.chronox.de/chacha20_drng.html
NVIDIA (2023) CUDA PTX-ISA. https://docs.nvidia.com/cuda/parallel-thread-execution
OLYM (2022) GuoMi SJJ1631 Hardware Security Module (HSM). https://new.myibc.net/bsmmj
Pan W, Zheng F, Zhao Y et al (2017) An Efficient Elliptic Curve Cryptography Signature Server With GPU Acceleration. IEEE Trans Inf Forensics Secur 12(1):111–122
Pu S, Liu J (2013) EAGL: an elliptic curve arithmetic GPU-based library for bilinear pairing. In: Pairing, lecture notes in computer science, vol 8365. Springer, pp 1–19
RFC (2007) RFC 5091: Identity-Based Cryptography Standard (IBCS) #1: supersingular curve implementations of the BF and BB1 cryptosystems. https://www.rfc-editor.org/info/rfc5091
Sahai A, Waters B (2005) Fuzzy identity-based encryption. In: EUROCRYPT, lecture notes in computer science, vol 3494. Springer, pp 457–473
Scott M, Benger N, Charlemagne M, et al (2009) On the final exponentiation for calculating pairings on ordinary elliptic curves. In: Pairing, lecture notes in computer science, vol 5671. Springer, pp 78–88
Shamir A (1984) Identity-based cryptosystems and signature schemes. In: CRYPTO, lecture notes in computer science, vol 196. Springer, pp 47–53
Shigeo M (2015) MCL: a fast pairing-based cryptography library. https://github.com/herumi/mcl
Sun S, Ma H, Zhang R et al (2020a) Server-aided immediate and robust user revocation mechanism for SM9. Cybersecur 3(1):12
Sun S, Zhang R, Ma H (2020b) Efficient parallelism of post-quantum signature scheme SPHINCS. IEEE Trans Parallel Distrib Syst 31(11):2542–2555
Synopsys, Inc (2016) The heartbleed bug (cve-2014-0160). https://heartbleed.com/
Szerwinski R, Güneysu T (2008) Exploiting the power of gpus for asymmetric cryptography. In: CHES, lecture notes in computer science, vol 5154. Springer, pp 79–99
Vercauteren F (2008) Optimal pairings. Cryptol. ePrint Arch. 2008/96
Wang T, Guo W, Wei J (2019) Highly-parallel hardware implementation of optimal ate pairing over Barreto–Naehrig curves. Integr 64:13–21
Wei R, Zheng F, Gao L, et al (2021) Heterogeneous-PAKE: bridging the gap between PAKE protocols and their real-world deployment. In: ACSAC. ACM, pp 76–90
Xiaomi (2023) Xiaomi 2023 Q2 Adjusted Net Profit Surges 147 Billion. https://www.mi.com/global/discover/article?id=3008
Xie Y, Wang B, Zhang L et al (2022) A high-performance processor for optimal ate pairing computation over Barreto–Naehrig curves. IET Circuits Dev Syst 16(5):427–436
Zhang R, Zou H, Zhang C, et al (2020) Distributed key generation for SM9-based systems. In: Inscrypt, lecture notes in computer science, vol 12612. Springer, pp 113–129

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.