https://crad.ict.ac.cn
63(2): 405−418, 2026

基于国产深度计算单元的 SPHINCS$^+$-SM3 高性能优化

宁祎静[1] 董建阔[1,2] 周思源[2] 林璟锵[1] 孙思维[3] 郑昉昱[3] 葛春鹏[4]
[1]（中国科学技术大学网络空间安全学院 合肥 230026）
[2]（南京邮电大学计算机学院 南京 210023）
[3]（中国科学院大学密码学院 北京 100049）
[4]（山东大学软件学院 济南 250100）
（truegeorge@mail.ustc.edu.cn）

High-Performance Optimization of SPHINCS$^+$-SM3 Based on Domestic Deep Computing Unit

Ning Yijing[1], Dong Jiankuo[1,2], Zhou Siyuan[2], Lin Jingqiang[1], Sun Siwei[3], Zheng Fangyu[3], and Ge Chunpeng[4]
[1]（*School of Cyber Science and Technology*, *University of Science and Technology of China*, *Hefei* 230026）
[2]（*School of Computer Science*, *Nanjing University of Posts and Telecommunications*, *Nanjing* 210023）
[3]（*School of Cryptology*, *University of Chinese Academy of Sciences*, *Beijing* 100049）
[4]（*School of Software*, *Shandong University*, *Jinan* 250100）

Abstract Digital signatures play a critical role in information security; however, traditional digital signature algorithms are at risk of becoming obsolete in the post-quantum era. SPHINCS$^+$, as a digital signature framework resistant to quantum computing attacks, is expected to become increasingly important in this new era. Nevertheless, the relatively slow computational speed of SPHINCS$^+$ poses challenges in meeting the high throughput and low latency demands of modern cryptographic applications, significantly limiting its practicality. An efficient optimization strategy based on a domestic DCU (Deep Computing Unit) is presented to accelerate the SPHINCS$^+$ algorithm instantiated with the domestic SM3 Hash function. By enhancing memory copy efficiency, optimizing the computational processes of SM3 and SPHINCS$^+$, and employing optimal computational parallelism, we implement the 128-f mode of SPHINCS$^+$-SM3 on DCU. Experimental results demonstrate that, compared with traditional CPU implementations, our DCU-based implementation achieves a significant increase in throughput, improving signature generation and verification by 2 603.87 times and 1 281.98 times, respectively. This substantial improvement in computational efficiency and practicality enhances the feasibility of SPHINCS$^+$ and advances the domestic adoption of post-quantum cryptographic algorithms. In scenarios involving high data traffic and large volumes of signature requests, the DCU implementation exhibits significant performance advantages over CPU implementations.

Key words SPHINCS$^+$; SM3; post quantum cryptography; parallel computing; DCU acceleration

摘 要 数字签名在信息安全中扮演着至关重要的角色，但传统的数字签名算法在后量子时代面临失效

收稿日期：2024-10-10；**修回日期**：2025-05-06
基金项目：国家自然科学基金项目（62572255, 62302238, 62372245, 62172258）；江苏省自然科学基金项目（BK20220388）；江苏省科技厅重点研发计划产业前瞻与关键核心技术项目（BE2022081）；中国博士后科学基金项目（2022M711689）；腾讯犀牛鸟基金项目（RAGR20240129）
This work was supported by the National Natural Science Foundation of China (62572255, 62302238, 62372245, 62172258), the Natural Science Foundation of Jiangsu Province (BK20220388), the Key Research and Development Program of Jiangsu Province Department of Science and Technology (BE2022081), the China Postdoctoral Science Foundation (2022M711689), and the CCF-Tencent Rhino-Bird Fundation (RAGR20240129).
通信作者：董建阔 (djiankuo@gmail.com)

的风险。SPHINCS$^+$作为一种能够抵抗量子计算攻击的数字签名框架,将在后量子时代发挥越来越重要的作用。然而,SPHINCS$^+$的计算速度较慢,难以满足现代密码算法对于高吞吐量和低延时的需求,极大地限制了其实用性。提出了一种基于国产深度计算单元(deep computing unit, DCU)的高效优化方案,以加速由国产哈希算法SM3实例化的SPHINCS$^+$算法。通过提高内存拷贝效率、优化SM3、改进SPHINCS$^+$的计算流程以及采用最佳计算并行度,在DCU上实现了SPHINCS$^+$-SM3的128-f模式。实验结果表明,与传统CPU实现相比,DCU上的实现显著提高了签名生成和验证的吞吐量,分别达到了2 603.87倍和1 281.98倍的提升,极大地增强了SPHINCS$^+$的计算效率和实用性,并推进了后量子密码算法的国产化进程。在数据流量和大量签名请求的场景下,DCU实现展现出显著优于CPU实现的性能优势。

关键词 SPHINCS$^+$; SM3; 后量子密码; 并行计算; DCU加速

中图法分类号 TP393

DOI: 10.7544/issn1000-1239.202440789 **CSTR:** 32373.14.issn1000-1239.202440789

在当今高度信息化的时代,密码学作为信息安全领域的基石,在保障网络空间安全方面发挥着至关重要的作用。然而,随着量子计算技术的进步和发展,传统的密码算法已经无法保证数据的安全性,基于Shor[1]和Grover[2]的量子计算能在多项式时间内破解目前广泛使用的公钥密码学方案,如RSA[3]和椭圆曲线密码(elliptic curve cryptography, ECC)。因此,后量子密码学(post quantum cryptography, PQC)逐渐成为了研究的热点。美国国家标准与技术研究院(National Institute of Standards and Technology, NIST)于2016年启动了全球范围内的后量子密码标准征集,以抵抗量子计算攻击。2022年,NIST公布了后量子密码学标准化流程的第3轮评审结果,包括4种选定的算法(CRYSTALS-KYBER[4], CRYSTALS-DILITHIUM[5], Falcon[6], SPHINCS$^+$[7])及4种候选算法(BIKE, Classic McEliece, HQC, SIKE)。这些算法均基于比传统公钥密码学更复杂的数学难题,并且没有已知的经典算法和量子算法可以有效攻克这些难题。目前的后量子密码算法有4种主流的技术路线:基于格的密码学(lattice-based cryptography, LBC)、基于编码的密码学(code-based cryptography, CBC)、基于哈希的密码学(Hash-based cryptography, HBC)和基于多变量的密码学(multivariate-based cryptography, MBC)。在NIST第3轮选定的4种算法中,不同于基于格构造的CRYSTALS-KYBER, CRYSTALS-DILITHIUM, Falcon, SPHINCS$^+$作为一种基于哈希的数字签名方案,成为唯一入选的HBC算法,其稳固的安全性源自哈希算法坚实的安全基础。

SPHINCS$^+$具有良好的可调节性,支持多种哈希算法(如SHA-256[8], SHAKE-256[9], SM3[10]等)的实例化,从而得到不同的SPHINCS$^+$算法实例。此外,SPHINCS$^+$作为一种无状态的数字签名算法框架,与其他需要维护状态信息的HBC算法(如XMSS[11]等)不同,它在实际应用中无需记录已使用的密钥等状态信息,因而具有更高的实用性。最后,SPHINCS$^+$还具备密钥短的优势,在128 b的安全强度下其公钥和私钥大小分别为32 B和64 B,与同等安全强度下的ECDSA[12]密钥长度接近,从而降低了密钥管理和传输的开销。然而,SPHINCS$^+$存在计算效率过低的缺陷,在x86-64处理器上,与基于格的数字签名算法CRYSTALS-DILITHIUM相比,SPHINCS$^+$的签名生成和验证速度分别慢约150倍和50倍[13],这极大地影响了其实际的应用性能。因此,提升SPHINCS$^+$的计算速度成为其实际部署的关键技术挑战。SPHINCS$^+$-SM3[14]是利用国产哈希算法SM3对SPHINCS$^+$进行实例化得到的算法实例,其特性与SPHINCS$^+$相似,过慢的计算速度限制了它的实际应用。为此,我们希望能采用CPU调配众核处理器的方式,通过异构计算模式来提升SPHINCS$^+$-SM3的性能。

GPU是一种专为图形处理和高性能计算(high performance computing, HPC)设计的处理器,适用于并行计算和大规模数据处理。GPU的应用不仅限于计算机图形学领域,还扩展到了各种需要大规模数据处理的应用场景。国际市场上主流的GPU品牌包括NVIDIA, AMD, Intel, ARM等,它们的产品同时面向游戏和高性能计算市场。在国内,海光信息、景嘉微和摩尔线程等公司也推出了面向高性能计算市场的GPU产品。随着科技的进步,GPU从早期的单一功能图形加速器发展成为今天的通用并行计算平台。在密码工程领域,GPU的应用极为广泛,其优势主要体现在大规模数据并行计算方面。基于GPU实现的密码算法能够显著提高计算吞吐量。当前,已有多

种密码学算法被成功移植到 GPU 平台上[15-17]，GPU 作为高性能的计算资源，在密码工程领域的作用日益突出。为了解决 SPHINCS$^+$计算速度过慢而难以实际应用的困境，推动后量子密码算法国产化的进展，本研究将 SPHINCS$^+$-SM3 算法实例在海光信息的深度计算单元（deep computing unit，DCU）上进行了高性能实现。我们采用了 CPU-DCU 异构计算架构，以实现高吞吐量的目标。

1 背　景

由于数字化技术的广泛应用，我们每天都在产生和传输大量的个人隐私信息，如银行卡信息、各种应用程序的账户密码和医疗记录等，密码技术能确保这些个人信息的安全性。随着电子商务、在线支付、移动互联网等互联网应用的快速发展，面对海量数据即时安全传输的需求，高吞吐、低延时的高性能密码计算技术成为保障交易和通信安全的关键，确保敏感数据在传输过程中不被窃取或篡改。在国家安全层面，密码技术提供了可靠的保密机制。此外，密码技术还可用于实现数字签名和数字证书等认证机制，确保通信双方的合法性。随着数字化时代的不断演进，密码技术作为保障网络信息安全和数据安全的核心技术和基础支撑，变得愈发重要。

近年来，《中华人民共和国网络安全法》《中华人民共和国密码法》《中华人民共和国数据安全法》《中华人民共和国个人信息保护法》的相继颁布，标志着网络空间安全和数据安全已提升到新的高度。这些法规的制定和实施，为网络空间安全和数据信息保护提供了坚实的法律保障和指导，促进了网络空间安全的规范化和健康发展。2023 年 4 月，国务院常务会议审议通过《商用密码管理条例修订草案》[18]，表明商用密码在保障网络和信息安全、保障公民和企业利益等方面的重要性日益凸显。数字签名技术作为密码技术的重要组成部分，可以保障数据信息的完整性、可认证性和不可否认性，为网络空间安全的发展提供了坚实的技术保障。例如，基于数字签名、实体认证和数字证书等技术，可以构建公钥基础设施（public key infrastructure，PKI）[19]，通过双向认证机制，在云、边、端 3 侧实现身份认证和访问控制，避免遭受未授权访问或越权访问攻击。

然而，近年来，一些研究机构和企业在量子计算领域取得了重大进展[20]，量子计算机利用 Shor 算法和 Grover 算法可以在多项式时间内攻破椭圆曲线上的离散对数问题，这意味着当前广泛使用的 ECDSA 数字签名算法已不再安全。进入后量子时代，我们需要将原有的不具备量子安全性的数字签名算法替换为具备量子安全性的算法。SPHINCS$^+$是一种利用哈希算法的抗量子特性构造的具备量子安全性的无状态数字签名算法框架，已被 NIST 选为标准算法之一，具有很好的实用前景。然而，SPHINCS$^+$为了确保算法的无状态性，在计算流程中使用了大量的哈希函数，导致计算速度过慢，限制了其在 TLS[21]，DNSSEC[22] 等具体网络空间安全协议中的应用。

GPU 是一种专门用于高效执行数据处理任务的专用处理器，也是显卡的核心芯片，在深度学习、科学计算和游戏等领域被广泛应用。与 CPU 不同，GPU 配备了大量速度较慢的计算核心，其晶体管主要用于算术逻辑单元而非数据缓存和流程控制。因此，GPU 拥有远超 CPU 的并行计算能力，能够同时处理大量数据。在特定的应用场景下，GPU 可以提供比 CPU 更高的运算速度和性能。GPU 编程是指使用 GPU 进行并行计算的编程技术，在深度学习、图像处理、科学计算等领域，GPU 编程已经成为必不可少的技术手段，基于 CPU-GPU 的异构计算逐渐发展为高性能计算领域的主流模式。

2 基础知识

本节介绍与本文有关的相关概念和基本知识，主要包括对符号表达、SPHINCS$^+$-SM3 以及 DCU 的介绍。

2.1 符号表达

本节介绍本文中符号的具体含义，后续本文都将采用如下符号表达。

1）W_i, W_i'：SM3 算法消息扩展过程中生成的将参与迭代压缩的字。

2）sk, pk, sig：私钥、公钥和签名值。

3）B^x：长度为 x 字节的二进制字符串。

4）n：WOTS$^+$[23] 和 FORS（forest of random subsets）[24] 的安全参数，代表了 WOTS$^+$和 FORS 中密钥及单个签名块的字节数，同时也代表 Merkle 树[25] 中节点值的字节数。

5）w：WOTS$^+$的 Winternitz 参数，可选值为 4, 16, 256。

6）l：WOTS$^+$私钥块和签名块的数量，$l = l_1 + l_2$，$l_1 = \lceil 8n/\mathrm{lb}\, w \rceil$，$l_2 = \lfloor \mathrm{lb}(l_1(w-1))/\mathrm{lb}\, w \rfloor + 1$。

7）d：超树的层数。

8) h': 超树每一层中 Merkle 树的高度。

9) h: 超树的总高度, $h=dh'$。

10) k: FORS 中包含的 Merkle 树的数量。

11) a: FORS 中包含 k 棵高度相同的 Merkle 树, 每棵 Merkle 树的高度为 a。

12) t: FORS 中 Merkle 树的叶子节点数量, $t=2^a$。

13) m: 生成的字节长度。$m=\lfloor(ka+7)/8\rfloor+\lfloor(h-h'+7)/8\rfloor+\lfloor(h'+7)/8\rfloor$。在为消息生成签名时, SPHINCS$^+$首先使用哈希函数处理消息, 生成长度为 m 字节的临时消息摘要 tmp_m, 再将 tmp_m 拆分为长度分别为 $\lfloor(ka)+7)/8\rfloor$, $\lfloor(h-h'+7)/8\rfloor$, $\lfloor(h'+7)/8\rfloor$字节的 3 段。其中, 第 1 段的前 ka 比特被记作 md, 由 FORS 为其生成签名; 第 2 段的前 $h-h'$比特被记作 $tree\_index$, 用于选择超树最底层的 Merkle 树; 第 3 段的前 h'比被记作 $leaf\_index$, 用于从被选中的 Merkle 树中选择叶节点。

14) {128, 192, 256}-{s, f}: 代表了 SPHINCS$^+$的不同模式, 数字部分代表算法提供的安全强度, 分别对应于 NIST 定义的安全等级 1, 3, 5。例如, “128”表示算法能够提供 128 b 的安全强度, 安全等级为 1。后缀字母代表应用需求, “s”指代“small”, 算法的参数选择以尽量小的签名大小为目标; “f”指代“fast”, 算法的参数选择以尽量快的签名生成速度为目标。

2.2 SM3

SM3[10] 是一种由中国国家密码管理局制定的哈希算法, 其结构和功能类似于 SHA-256。SM3 接收变长的消息并为消息生成定长的哈希值, 通常用于数据完整性的验证。如图 1 所示, SM3 的计算流程主要包括消息填充、消息分组、消息扩展和迭代压缩。

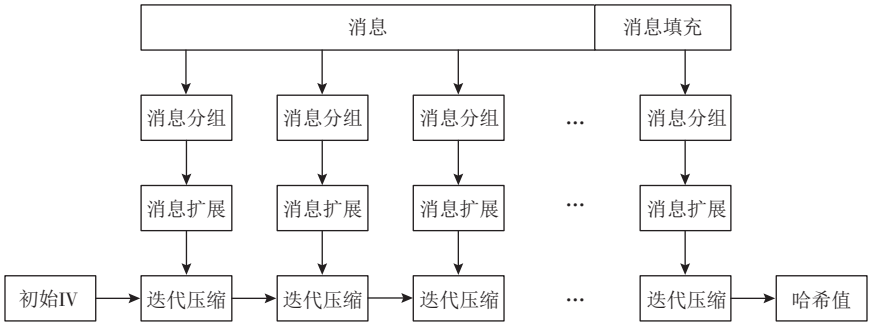


Fig. 1 Calculation process of SM3

图 1 SM3 计算流程

在消息填充阶段, 初始消息末端会被填充比特 0, 使其长度为 512 b 的整数倍。在消息分组阶段, 填充后的消息被划分为多个长 512 b 的消息块。每个消息块进行扩展后得到 132 个长 4 B 的字 W_0, W_1, …, W_{67}, W'_0, W'_1, …, W'_{63}。压缩函数应用包括加法、异或和循环移位在内的多种操作对扩展后的消息块进行处理, 每个消息块将历经 64 轮压缩。所有消息块被处理完毕后得到最终输出。

2.3 SPHINCS$^+$和 SPHINCS$^+$-SM3

SPHINCS$^+$[7] 算法使用多次签名算法 FORS 为消息摘要生成签名, 再借助单次签名算法 WOTS$^+$和 Merkle 树构建出超树对 FORS 的公钥进行认证。

2.3.1 哈希函数和地址字段

SPHINCS$^+$借助哈希函数对数据进行处理以得到签名值, 计算流程中使用的哈希函数有如下定义。

H_{msg}: $B^n\times B^n\times B^n\times B^n\to B^n$;

PRF_{msg}: $B^n\times B^n\times B^*\to B^n$(*表示任意长度);

PRF: $B^n\times B^{32}\to B^n$;

T_l: $B^n\times B^n\times B^{ln}\to B^n$($l$ 为任意正整数)。

在使用 SM3 算法对上述哈希函数进行实例化后, 就得到了 SPHINCS$^+$-SM3[14]。其具体的实例化方式如下:

$H_{msg}(R, PK.seed, PK.root, msg)=MGF1\text{-}SM3((R\|PK.seed\|PK.root\|msg), m)$;

$PRF_{msg}(SK.prf, opt, msg)=HMAC\text{-}SM3(SK.prf, opt\|msg)$;

$PRF(SK.seed, ADRS)=HMAC\text{-}SM3(SK.prf, opt\|M)$;

$T_l(PK.seed, ADRS, M)=SM3(BlockPad(PK.seed)\|ADRS^C\|M)$。

使用哈希函数对数据进行处理时, 为了标识子算法的类别以及当前待处理的数据在 SPHINCS$^+$结构中所处的位置, SPHINCS$^+$引入一个长度为 32 B 的地址字段 $ADRS$, $ADRS$ 与待处理数据一同作为哈希函数的输入, 每次调用哈希函数后, $ADRS$ 都将被更新。为了减少 SM3 压缩函数的调用次数, SPHINCS$^+$-SM3 采用长为 22 B 的压缩地址 $ADRS^C$ 来替代 $ADRS$。

2.3.2 计算流程

SPHINCS$^+$将超树和 FORS 结合在一起构建了一

个无状态的数字签名方案，其计算流程涵盖了密钥生成、签名生成和签名验证，算法流程的详细介绍参见官方文档[26]。算法的整体架构如图 2 所示。

2.3.3 参数选择

由于使用场景的需求不同，在实际应用过程中选择不同的参数得到 SPHINCS⁺的模式不同，表 1 是不同模式下 SPHINCS⁺的参数选择。

Table 1 Parameter Set of SPHINCS⁺

表 1 SPHINCS⁺参数集

模式	n	w	h	d	k	a	签名大小/B
128-s	16	16	63	7	14	12	7 856
128-f	16	16	66	22	33	6	17 088
192-s	24	16	63	2	17	14	16 224
192-f	24	16	66	22	33	8	35 664
256-s	32	16	64	8	22	14	29 792
256-f	32	16	68	17	35	9	49 856

2.4 DCU 和 HIP

DCU[27] 是由中国海光公司基于 AMD 指令集研发的一款 GPU，采用了通用 GPU 计算（general-purpose computing on GPU, GPGPU）[28] 架构。该设备通过 PCI-E 总线与 CPU 相连，由多个计算单元（compute unit, CU）、缓存系统（L1 和 L2 缓存）和全局内存组成。每个 CU 包含 4 个 SIMD 单元和 64 KB 共享内存 LDS（local data share），而每个 SIMD 计算单元中有 16 个 DCU 核和 16 384 个 32 b 的寄存器。寄存器在 DCU 中是极其珍贵的存储资源，由于它的访存时间极短，在使用 DCU 执行计算任务时，将频繁使用的数据存储在寄存器中可以有效降低数据读取的开销。共享内存是为线程之间的数据交互而设计的，同一个 CU 内的线程可以通过共享内存交换数据或存储常用的数据以减少对全局内存的访问频率。寄存器和共享内存均为计算单元内重要而有限的资源，而全局内存虽然存储空间大，但由于距离计算单元较远，访存速度慢。

HIP（heterogeneous interface for portability）[27] 是 DCU 采用的基于 C/C++的编程模型，提供了基于 hcc 编译器的头文件和运行库。HIP 编程环境与统一计算设备架构（compute unified device architecture, CUDA）[29] 编程环境在硬件架构和软件实现方面有诸多相似之处。HIP 和 CUDA 共享类似的编程模型、线程结构、内存组织和运行时 API 函数接口。除硬件架构和运行时软件层之外，二者在高级数学库的组织和函数接口方面也极为相似。这种架构上的高度相似性有助于将使用 CUDA 加速的现有程序和代码直接移植到 DCU 平台上运行。

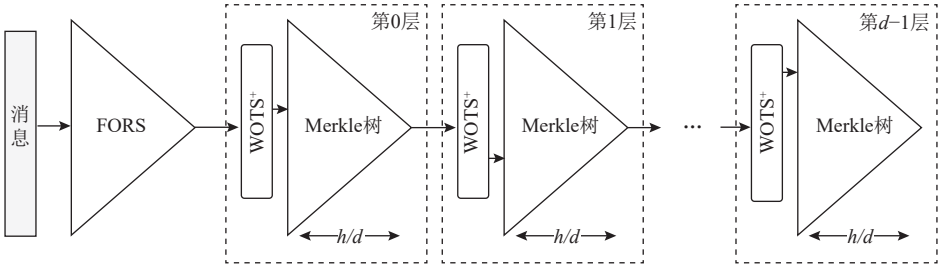


Fig. 2 Architecture of SPHINCS⁺

图 2 SPHINCS⁺架构

3 基于 DCU 的 SPHINCS⁺-SM3 加速

3.1 加速架构概览

图 3 展示了我们采用的基于 CPU-DCU 的异构计算模式：CPU 负责生成密钥并将密钥及待处理的数据（消息或签名）传输至 DCU。接收到密钥和待处理数据后，DCU 内核唤起大量线程进行并行计算。计算完成后 DCU 将计算结果传回 CPU。

SPHINCS⁺-SM3 的算法主要由 2 个子算法组成：FORS 和超树。其中，超树的计算时间约占整个计算过程总时间的 90%。FORS 和超树都是以 Merkle 树为基础组成的结构。Merkle 树是一棵满二叉树，每个节点的值由其 2 个子节点的值通过哈希函数计算得出。在签名生成（或签名验证）过程中，Merkle 树的计算包含 2 步：1）选择某个叶子节点对消息进行签名生成（或签名验证）。2）计算（或利用）验证路径，自底向上计算出根节点的值。

如图 3 所示，我们从多个层面对 SPHINCS⁺-SM3 进行了加速：1）采用新的内存访问方式，提高了内存访问效率。2）基于 DCU 的特性，对 SM3 的计算流程进行了优化，减少了计算过程中使用的寄存器数量。

3)简化了签名生成过程中超树的计算流程,减少了子算法的签名生成次数。4)从叶子节点内、Merkle树内以及不同Merkle树之间3个不同的角度分析了SPHINCS$^+$-SM3的算法流程,并讨论了并行计算的可能性,从而确定了吞吐量最高的并行模式。

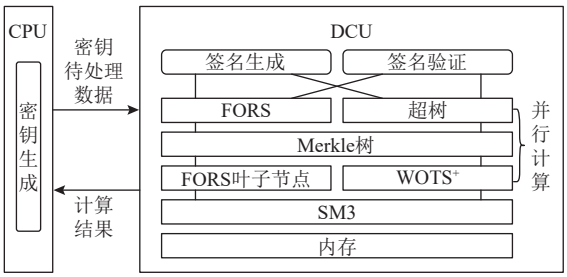


Fig. 3 Overall computing framework

图3 整体计算框架

3.2 SM3优化

SM3是SPHINCS+-SM3的基础组件。根据前文所述的SM3计算流程,SM3首先将消息扩展为132个字,随后使用扩展出的132个字进行64轮迭代压缩,这意味着需要使用132个寄存器对字进行存储直至迭代压缩结束。然而,通过对SM3迭代压缩过程进行深入分析,可以发现并非所有的字都参与每一轮的迭代压缩。实际上,与每轮迭代压缩直接相关的字仅有6个,这意味着原有的存储模式导致了大量的寄存器资源浪费。由于寄存器在DCU中是宝贵的存储资源,我们通过对消息扩展和迭代压缩进行结合以最大化寄存器利用率,从而提升性能。

图4展示了SM3迭代压缩过程中使用的压缩函数在第j轮的计算流程。其中T_j是常量,$A, B, \cdots, H$是储存每一轮计算结果的寄存器。消息扩展得到的字作用于上一轮的计算结果,经过复杂的数学操作后得出本轮的计算结果。可以观察到,每轮迭代压缩仅与W_j, W'_j(其中$W'_j = W_j \oplus W_{j+4}$)有关。由于$W_0$~$W_{15}$是由消息直接分块得出的,因此在前12轮计算中仅需要直接从寄存器读取W_j和W_{j+4}进行计算即可。当进入第13轮计算时,压缩函数中使用的W_{j+4}与W_{j-16}, W_{j-9}, W_{j-3}, W_{j-13}, W_{j-6}有关,无法再通过直接读取寄存器中的数据来获取W_{j+4}。原有的SM3计算流程预先计算所有的W_{j+4}值并将其存储在寄存器中,在计算时根据需求对寄存器进行读取,这种方法虽然简单直观,但并未充分利用寄存器资源。本文通过循环使用32个寄存器M_i和N_i(i=0, 1, ⋯, 15)来最大化地利用寄存器。

具体实现方案如图5所示。在进行第13轮计算(即j=12)时,仍可通过读取M_{12}直接得到W_{12}。但是,W_{12+4}不能被直接读取,需要通过计算得出。此时,我们开始使用数组N_i来存储W_{j+4}。可以观察到,N_0(存储W_{12+4})值是由W_0, W_7, W_{13}, W_3, W_{10}计算得出的,这些W_i可以直接从数组M_i中读取。随着迭代次数的增加,计算出的W_{j+4}值会陆续存储到N_i中。在第16轮计算(即j=15)时,W_{15+4}的计算与W_{16}有关。而W_{16}存储在N_0中,因此可以直接读取N_0来获取W_{16}。同样,在下一轮迭代中,可以通过读取N_1的值直接获取W_{17}。进入第28轮计算(j=27)时,可以发现16个

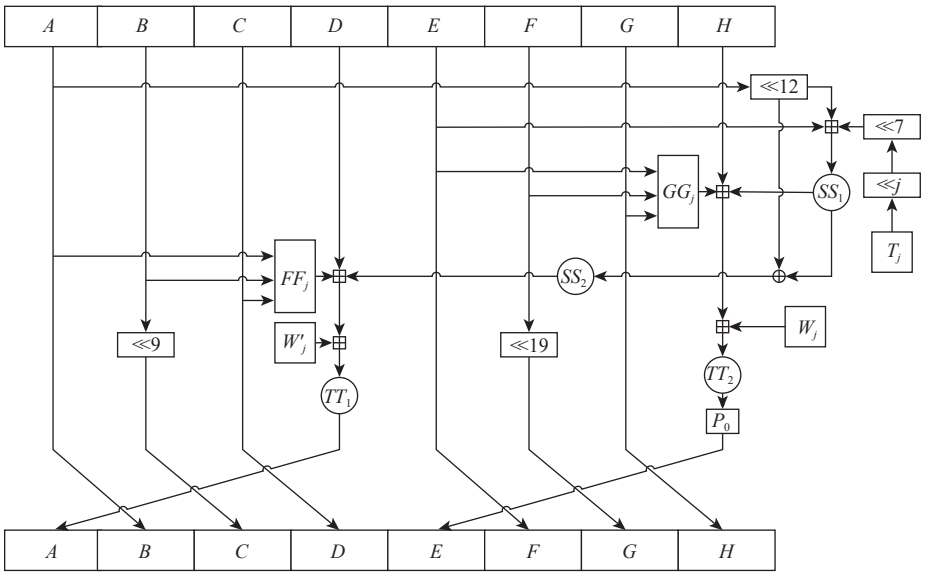


Fig. 4 Computation process of compress function

图4 压缩函数计算流程

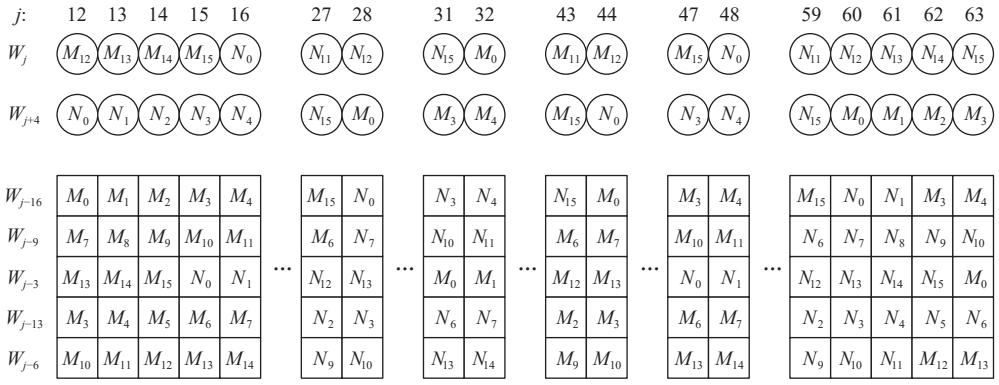


Fig. 5 Register optimization

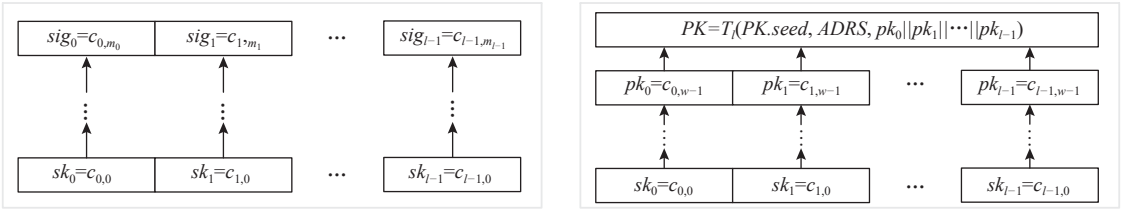
图 5 寄存器优化

寄存器 N_i(i=0, 1, …, 15)已被完全填满(对应于 W_{16}~W_{31}),而在后续的计算轮次中将不再使用 M_i 中存储的值。因此,我们接着使用寄存器 M_i,从而实现了寄存器的重用。在接下来的计算轮次中,W_{j+4} 的值都将存储在 M_i 中。例如,当 j=28 时,W_{28+4} 的值计算完成后,会被存储到 M_0 中。随后的循环需要获取 W_{32} 的值时将访问 M_0,以此类推。同样,当 j=44 和 j=60 时,也采用同样的方法重复使用 N_i 和 M_i。总之,随着迭代轮次的增加,M_i 或 N_i 的值也会依次增加。在下一次迭代中,任何数据一旦达到 M_{15} 或 N_{15},就会分别过渡到 N_0 或 M_0,并继续迭代。简单来说,随着迭代轮次的增加,与迭代压缩相关的 6 个字 W_j, W_{j-16}, W_{j-9}, W_{j-3}, W_{j-13}, W_{j-6} 中某一个的存储位置达到 M_{15}(或 N_{15})时,从下一次迭代开始,将继续从 N_0(或 M_0)中开始读取该值。

3.3 叶子节点内的计算

在 FORS 签名生成方案中,叶子节点内的计算包括 3 个步骤:1)使用伪随机函数为每个叶子节点生成私钥 sk_i=PRF($SK.seed$, $ADRS$)(i=0, 1, …, kt−1)。2)根据私钥计算出叶子节点的值 T_1($PK.seed$, $ADRS$, sk_i)。3)对于长度为 ka 比特的消息 md, FORS 将其拆分为 k 个长度为 a 比特的块 md_i(i=0, 1, …, k−1),md_i 代表了一个介于 0 和 t−1 之间的整数,从左至右选择第 it+md_i 个叶子节点的私钥作为块 md_i 的签名值。由于哈希函数的计算属于串行过程,这一计算过程不具备并行计算的可能性。在 FORS 的签名验证方案中,叶子节点内的计算涉及利用哈希函数 T_1 对签名值进行处理,得到叶子节点的值 T_1($PK.seed$, $ADRS$, sig_i)。如前文所述,这个过程同样不具备并行计算的可能性。

超树的每个叶子节点都是一个 WOTS⁺实例。如图 6 所示,在进行签名生成时,叶子节点内的计算包括 3 个步骤:1)为叶子节点生成 WOTS⁺私钥。每个 WOTS⁺实例的私钥包含 l 个私钥块,每个私钥块 sk_i=PRF($SK.seed$, $ADRS$)(i=0, 1, …, l−1),l 个私钥块构成私钥 sk=(sk_0, sk_1, …, sk_{l-1})。2)计算 WOTS⁺公钥作为叶子节点的值。借助哈希函数 T_1 构造出 l 条哈希链,每条哈希链上有 w 个元素。其中,第 i 条哈希链上的第 0 个元素 $c_{i,0}$=sk_i,第 j(j=1, 2, …, w−1)个元素 $c_{i,j}$=T_1($PK.seed$, $ADRS$, $c_{i,j-1}$)。使用 T_l 函数对哈希链末端的元素进行处理,得到叶子节点的值 T_l($PK.seed$, $ADRS$, $c_{0,w-1}||c_{1,w-1}||\cdots||c_{l-1,w-1}$)。3)选择某个叶子节点为消息生成签名。长度为 8n b 的消息 msg 会被拆分为 l_1 个长度为 lb w 的块 msg_i(i=0, 1, …, l_1−1)。计算消息校验和 $csum$,并将 $csum$ 拆分为 l_2 个长度为 lb w 的块 $csum_j$(j=0, 1, …, l_2−1),通过将消息与校验和分块,可以得到 l 个介于 0 和 w−1 之间的整数,记作 m_k(k=0, 1, …, l−1)。选取第 k 条哈希链上的第 m_k 个元素的值



(a) 签名生成 (b) 公钥生成

Fig. 6 The computation process of WOTS⁺

图 6 WOTS⁺计算流程

作为签名块 sig_k, 选取 l 个签名块组成签名 sig。在进行签名验证时, 叶子节点内的计算包括利用消息和签名值沿着哈希链继续进行计算得出哈希链的末端元素值, 再使用函数 T_l 对得出的 l 个元素值进行处理得到叶子节点的值, 并与公钥进行比对。由于每个叶子节点的私钥包含了 l 个私钥块, 代表各个块的哈希链之间的计算没有数据依赖, 因此可以进行并行计算。

3.4 Merkle 树内的计算

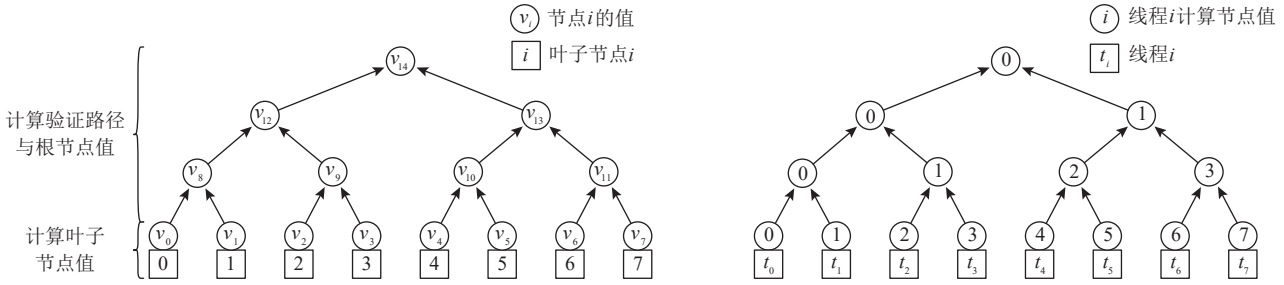
FORS 和超树都是基于 Merkle 树构建的。在签名生成过程中, FORS 和超树中的每一棵 Merkle 树都需要利用叶子节点的值自底向上计算验证路径和根节点值。图 7(a)展示了 1 棵高度为 3 的 Merkle 树的计算流程。父节点的值由 T_2 对 2 个子节点值进行处理得出。例如, 设节点 i 的值为 v_i, 则有 $v_8=T_2(PK.seed, ADRS, v_0\|v_1)$, 依此类推可计算出根节点值。验证路径是指被选中的叶子节点到根节点的路径上所有节点(不包括根节点)的兄弟节点值, 验证路径可以在计算根节点值的过程中得出。例如, 在图 7(a)中, 若被选中的叶子节点为 1, 则验证路径为(v_0, v_9, v_{13})。从上述分析可以得出结论: 同一高度下节点值的计算是相互独立的, 因此可以进行并行计算, 图 7(b)展示了 Merkle 树的并行计算方案, 在自底向上的计算过程中, 参与计算的线程数量逐渐减少。一般地, 对于 1 棵高度为 h' 的 Merkle 树, 自底向上的计算任务数量从 $2h'$ 降为 1, 这意味着若采用多个线程并行计算, 将导致部分线程在计算后期处于闲置状态, 从而降低线程利用率。

在进行签名验证时, Merkle 树内的计算包括根据签名值导出被选中的叶子节点值, 并自底向上借助验证路径求出根节点值。例如, 在图 7(a)中, 若被选中的叶子节点为 1, 验证路径为(v_0, v_9, v_{13}), 则可由 v_0 和 v_1 计算 v_8, 由 v_8 和 v_9 计算 v_{12}, 由 v_{12} 和 v_{13} 计算根节点值 v_{14}。可以看到, 这是一个串行过程, 不具备并行计算的可能性。

3.5 Merkle 树之间的计算

FORS 由 k 棵相互独立的 Merkle 树组成。图 8 展示了一个 k=3 的 FORS 实例, 在 FORS 中, 各棵 Merkle 树分别计算出各自的根节点值 r_0, r_1, r_2, 进而可计算得出 FORS 的公钥值 $T_3(PK.seed, ADRS, r_0\|r_1\|r_2)$。由于各棵 Merkle 树的计算互不影响, 不同的树之间可以进行并行计算。

如图 8 所示, 超树分为 d 层, 计算过程中会在每层选择 1 棵 Merkle 树参与计算。在签名生成过程中, 各层 Merkle 树需要使用选定的叶子节点为下层 Merkle 树的根节点生成签名, 随后自底向上计算根



(a) 计算流程 (b) 并行方案

Fig. 7 The computation process of Merkle tree

图 7 Merkle 树计算流程

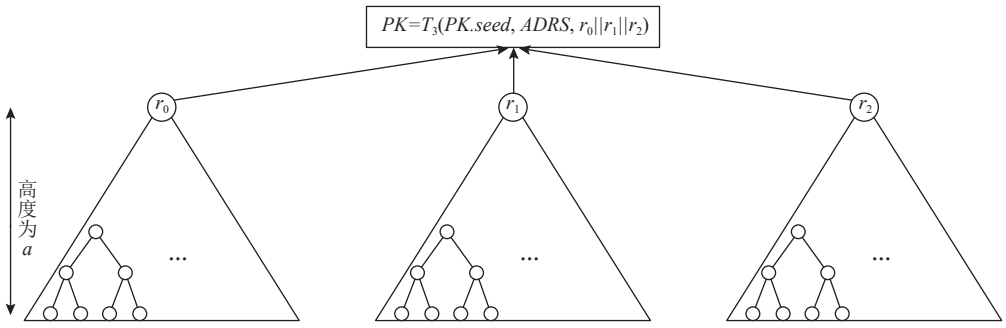


Fig. 8 Computation process of FORS

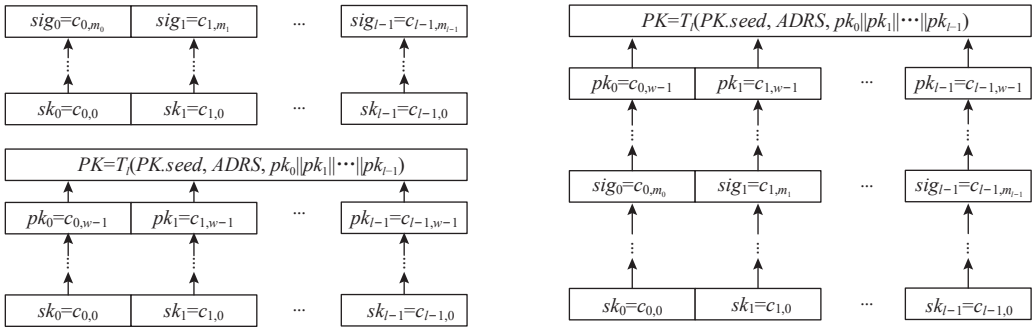
图 8 FORS 计算流程

节点的值供上层 Merkle 树进行签名生成。各层的计算存在数据依赖，但由于根节点值的计算不依赖于其他层，我们可以通过调整计算顺序，各层先计算出验证路径及根节点的值，再使用选定的叶子节点为下层的根节点生成签名，这样就可以并行地对超树不同层的 Merkle 树进行计算。在签名验证过程中，各层 Merkle 树需要根据下层 Merkle 树的根节点值导出被选中叶子节点的值，再根据被选中叶子节点的值和验证路径求出根节点值，这个过程与签名生成不同，根节点值的计算与其他层之间存在数据依赖，因而我们无法通过调整计算顺序来实现各层之间的并行计算，只能采用自底向上逐层计算的方式。

3.6 其他优化方法

3.6.1 签名生成过程中超树的优化

在 SPHINCS⁺-SM3 的签名生成过程中，超树中每棵 Merkle 树的计算包括 2 部分：1）选定叶子节点使用 WOTS⁺签名算法为消息生成签名。2）在生成每个叶子节点的公钥后，自底向上计算验证路径以及根节点值。图 9（a）展示了 WOTS⁺的初始计算流程，被选中的叶子节点首先为消息生成签名，随后生成公钥以便于计算验证路径与根节点的值。WOTS⁺的计算流程可被视作一系列哈希链，共有 l 条哈希链，每条哈希链以私钥块为起点值，通过 $w-1$ 次哈希函数迭代得出终点值。在签名生成过程中，根据消息块的不同，从哈希链中选取相应的元素作为签名值的一部分，由此可知 WOTS⁺的签名生成是公钥生成的中间过程。一旦确定待签名消息后，我们可以在生成公钥的同时完成签名生成，从而省去单独的签名生成步骤。图 9（b）展示了优化后的计算流程。在初始的计算流程中，被选中的叶子节点在生成签名时平均需要进行 $wl/2$ 次哈希运算，生成公钥时需要进行 $wl+1$ 次哈希运算。经过优化后，被选中的叶子节点所需的哈希运算次数从平均 $3wl/2+1$ 次下降至 $wl+1$ 次。在 128-f, 192-f 和 256-f 模式下，优化前的哈希次数分别为 841, 1 837 和 3 217，而优化后的哈希次数降低为 561, 1 225 和 2 145，减少了约 33% 的哈希次数。值得一提的是，这是一个通用的优化策略，不仅适用于 DCU，同样适用于 CPU。



(a) 原计算流程　　(b) 现计算流程

Fig. 9 Comparison of WOTS⁺ before and after optimization

图 9 WOTS⁺优化前后对比

3.6.2 内存优化

DCU 中内存采用分级组织的方式，内存逐级分为寄存器、共享内存和全局内存。内存容量依次增大，而访存速度相应递减。DCU 在执行计算任务时，需要频繁从全局内存读取数据至寄存器中进行处理，并在计算结束后将寄存器中的数据写入全局内存以便于将计算结果传输回 CPU。而传统的内存读写方式每次仅传输单个字节的数据，这导致内存访存效率低下。例如，在 SPHINCS⁺-SM3 的 128-f 模式中，每个基本数据块的大小为 16 B，在传统的内存读写方式下读写 1 个数据块需要进行 16 次内存操作，未能高效利用总线的带宽。通过将数据块转为 HIP 支持的 INT4 格式（大小为 16 B），达到了合并内存访问的目的，使得每次可以读取或写入 16 B 的数据，从而仅需 1 次内存操作即可完成整个数据块的读写，显著提高了内存访问的效率。

3.7 SPHINCS⁺-SM3 签名生成和签名验证的高速实现

基于对 SM3 计算流程的优化，我们利用高性能的 SM3 算法实现了高性能的哈希函数。以高效的内存访问方式和高性能的哈希函数为基础，我们确定了与签名生成和签名验证最适配的并行模式和相应的并行参数，整体的并行计算流程如图 10 所示。

SPHINCS⁺-SM3 的签名生成流程大致可以分为 2 个主要阶段：在第 1 阶段，FORS 用于为消息生成签

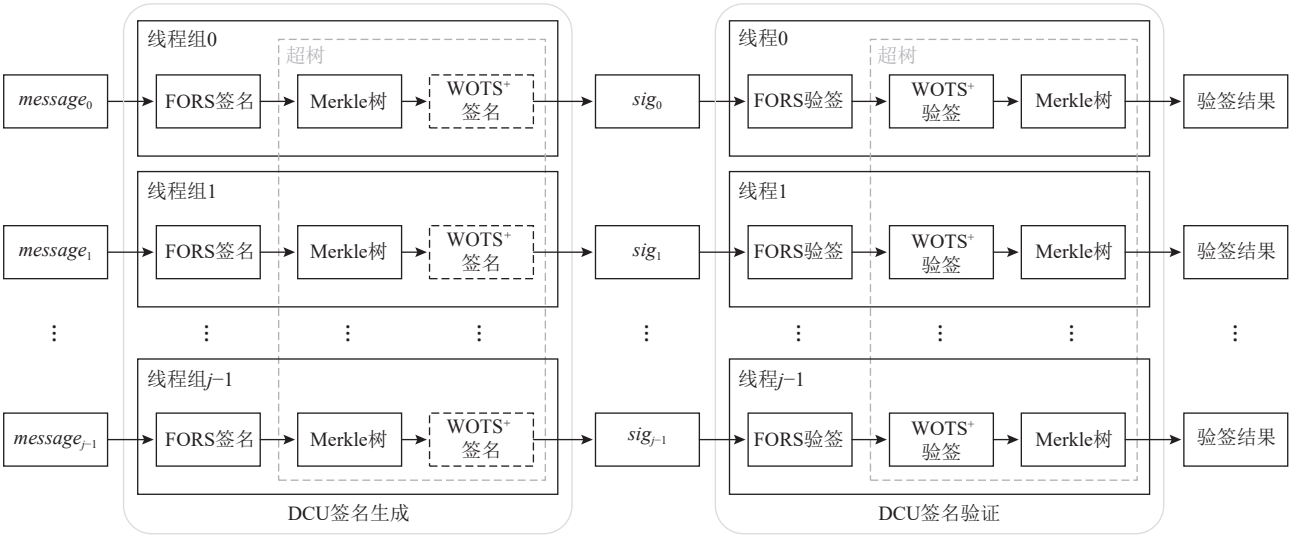


Fig. 10 Signature generation and verification computation processes based on DCU

图 10 基于 DCU 的签名生成和验证计算流程

名; 在第 2 阶段, 超树为 FORS 的根节点值生成签名。根据 3.3~3.5 节中的分析, 在签名生成过程中, FORS 可以选择树内并行或树间并行的方式进行计算。然而, 树内并行会导致线程的浪费, 因此对于 FORS 我们采用树间并行的方式进行计算。对于超树, 理论上在 3 个层面上均可以采用并行计算。若选择叶子内并行的方式, 即使用多个线程同时计算 WOTS$^+$中的不同数据块, 那么在后续计算过程中, 每棵 Merkle 树都将被分配给多个线程进行处理, 然而如 3.4 节所述, 在 Merkle 树内进行并行计算将导致线程的闲置。因此, 我们通过调整超树的计算顺序实现了树间并行, 在每一层中先计算 Merkle 树, 随后使用叶子节点为下层 Merkle 树的根节点生成签名, 从而避免线程的浪费。综上所述, 对于 FORS 和超树, 均采用 Merkle 树间并行的方式进行计算。为了最大化线程利用率, 我们需要选择最优的线程配置。假设我们使用包含 e 个线程的线程组为单个消息生成签名, 每个线程在每轮中负责对 FORS 中的某一棵 Merkle 树和超树中的某一层进行计算, 那么 FORS 和超树的计算将分别进行$\lceil k/e\rceil$和$\lceil d/e\rceil$轮, 在最后 1 轮将分别导致$(e\times\lceil k/e\rceil-k)$和$(e\times\lceil d/e\rceil-d)$个线程闲置, 为了最大化吞吐量, 需要减少闲置线程的数量, k/e 和 d/e 应当分别接近$\lceil k/e\rceil$和$\lceil d/e\rceil$。如表 1 所示, 在 128-f 和 192-f 模式下, k 和 d 分别为 33 和 22, e 的值应该为 k 和 d 的最大公因数 11, 意味着每 11 个线程构成线程组为单个消息生成签名。这样, 在对 FORS 和超树进行计算时, 我们需要分别进行 3 轮和 2 轮计算。而在 256-f 模式下, k 和 d 分别为 35 和 17, 综合考量后, e 的取值调整为 18, 意味着此时的线程组由 18 个线程构成。在此设置下, 对 FORS 和超树进行计算时, 分别需要进行 1 轮和 2 轮计算, 在 FORS 的第 1 轮计算和超树的第 2 轮计算中, 仅会导致 1 个线程的浪费。

SPHINCS$^+$-SM3 的签名验证流程也可以分为 2 个阶段, 第 1 阶段为借助消息和签名值计算出 FORS 的根节点值, 第 2 阶段为根据 FORS 根节点值与签名值计算出超树的根节点值与公钥进行比对。由 3.3~3.5 节的分析, 在签名验证过程中, FORS 可采用树间并行的方式, 不会出现线程浪费。对于超树, 虽然可采用叶子内并行的方式, 即使用多个线程并行计算 WOTS$^+$的不同数据块, 然而由于后续根据验证路径计算根节点值的过程是一个仅能由单个线程完成的串行过程, 叶子内并行的方式会导致在线程计算根节点值时产生闲置。考虑到在整个签名验证过程中超树的计算是最耗时的部分, 因而, 使用单个线程对每个签名进行验证是最佳的策略。

在进行签名生成时, 我们将并行计算与超树的优化结合在一起。如 3.6.1 节所述, 在为超树中被选中的叶子节点生成公钥时, 若待叶子节点进行签名的消息已知, 可以将签名生成与公钥生成相结合, 从而省略单独的签名生成步骤。在我们的并行方案中, 128-f 模式下, 超树需要使用 11 个线程进行 2 轮计算, 线程 i(i=0, 1, …, 10)负责计算超树的第 $2i$ 层和 $2i$+1 层。在线程 i 进行第 1 轮计算时, 它对第 $2i$ 层进行处理, 此时待签名的消息为线程 i−1 在第 2 轮计算中得出的第 $2i$−1 层 Merkle 树的根节点值, 这对于线程

i 来说是未知的(i=0 时，待签名消息为已知的 FORS 根节点值)。然而，在线程 i 进行第 2 轮计算时，它对第 $2i$+1 层进行处理，此时待签名的消息为线程 i 在第 1 轮计算得出的第 $2i$ 层 Merkle 树的根节点值，这对于线程 i 来说是已知的。因此，在对超树进行计算时，从第 2 轮计算开始可以采用我们的优化策略，将 WOTS⁺签名生成次数从 22 次降为 10 次(线程 0 第 1 轮计算即可采用该优化策略)。一般地，若采用 e(e≤d)个线程为单个消息生成签名，则采用我们的优化方案，可以使超树中额外的 WOTS⁺签名生成次数由 d 次减少为 e−1 次。而对于 CPU 上逐层进行的超树计算，可以采用上述方法将额外的 WOTS⁺签名生成次数降为 0。

4 实验结果及对比分析

通过采用第 3 节提出的优化策略，我们在 DCU 上实现了 SPHINCS⁺-SM3 的 128-f 模式，并进行了性能测试。性能评估的指标主要有 2 项：1)延时，即程序从开始执行到结束所耗费的时间。2)吞吐量，即单位时间内可以进行的操作数量，在本实验中代表单位时间内能生成或验证的签名数量。每项性能指标都是程序执行 100 次后取平均值得到的。

4.1 实验结果

在 DCU 中，若干个线程被组织为 1 个块，而若干个块则构成 1 个网格。在运行 HIP 程序时，需要指定网格和块的大小。在签名生成过程中，每个线程组包含 11 个线程，不是 2 的幂次，此时存在 2 种处理策略：1)将块的大小设置为 11 的倍数，确保线程组在同一个块内并避免了线程闲置。2)将块的大小设置为 2 的幂次，虽然会导致最后 1 个块中出现少量线程闲置，但能够最大化利用 DCU 的计算资源。实验表明，第 2 种策略展现出了更为优越的性能。因而在我们的实现中，将网格的大小固定为 128，块的大小依次从 64 增至 512，得出了不同块大小下的签名生成和签名验证的性能，如表 2 和图 11 所示。

Table 2 Performance with Different Block Sizes

表 2 不同块大小下的性能

性能参数	签名生成				签名验证			
	64	128	256	512	64	128	256	512
吞吐量/OPS	9 276.81	18 351.00	22 328.86	22 653.63	69 500.30	133 019.40	185 161.40	195 425.67
延时/ms	80.20	81.14	133.37	262.96	117.87	123.17	176.97	335.35

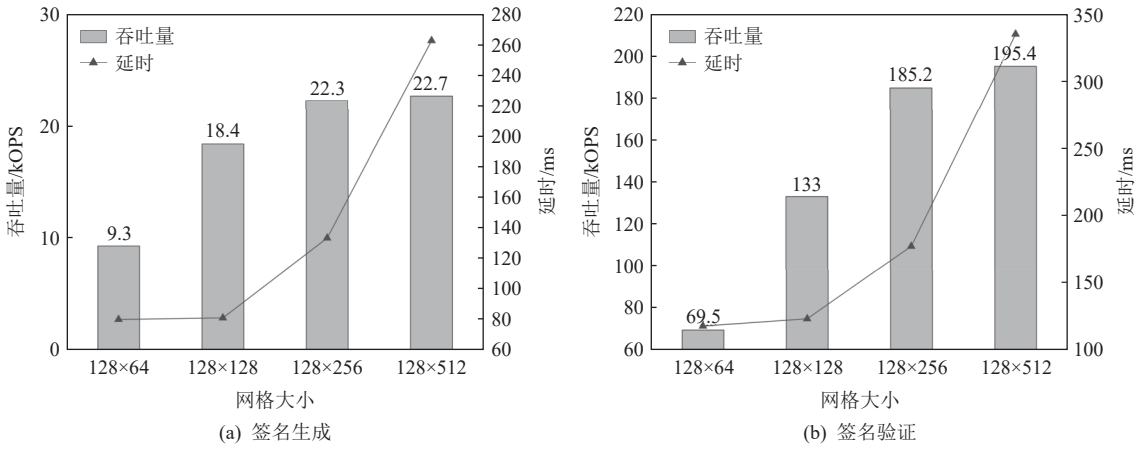


Fig. 11 Performance of signature generation and verification

图 11 签名生成与验证的性能

随着块的大小逐渐增加，线程数量也随之增长，可并行处理的消息数量增多，从表 2 可以观察到，程序运行所需的时间也逐渐增加。签名生成和签名验证的吞吐量在块的大小设置为 512 时达到峰值，分别为 22 653.63 OPS 和 195 425.67 OPS。用户可以根据待处理数据的规模并结合对系统吞吐量和延时的具体需求来选择合适的块大小进行计算。

4.2 结果对比与分析

表 3 展示了不同平台下多种后量子数字签名算法的峰值性能对比。相较于 CPU 实现[14]，基于 DCU

Table 3 Performance Comparison of Post Quantum Digital Signature Algorithms Under Different Platforms

表 3 不同平台下后量子数字签名算法的性能对比

实验平台	算法种类	安全等级	签名生成		签名验证	
			吞吐量/OPS	比值	吞吐量/OPS	比值
CPU[14]	SPHINCS$^+$-SM3	1	8.70	1.00	152.44	1.00
Xilinx XZU3EG[30]	SPHINCS$^+$-SHA256	1	15.54	1.79	398.41	2.61
Artix-7[31]	SPHINCS$^+$-SHAKE256	1	990.10	113.80	6 250.00	41.00
RTX 3090[32]	SPHINCS$^+$-SHA256	1	44 391.00	5 102.41	285 681.00	1 874.06
A100[33]	DILITHIUM	2	765 855.00	88 029.31	2 057 013.00	13 493.92
海光 DCU	SPHINCS$^+$-SM3	1	22 653.63	2 603.87	195 425.67	1 281.98

的 SPHINCS$^+$-SM3 实现在签名生成方面的吞吐量从 8.70 OPS 提升为 22 653.63 OPS，提高了约 2 603.87 倍；而在签名验证方面，吞吐量从 152.44 OPS 提升为 195 425.67 OPS，提高了约 1 281.98 倍。性能的巨大提升一方面得益于 DCU 的多个计算核心可以对海量数据同时进行处理，另一方面得益于我们在多个维度上所采用的优化策略。

许多研究者在不同于 DCU 的计算平台上对后量子数字签名算法的高性能实现进行了大量探索，各个实现的性能对比如表 3 所示。Berthet 等人[30]和 Amiet 等人[31]分别使用 SHA-256 和 SHAKE-256 对 SPHINCS$^+$进行了实例化，并在现场可编程门阵列(field-programmable gate array，FPGA)上进行了加速实现，在 128-f 模式下，与他们的实现相比，SPHINCS$^+$-SM3 的实现在签名生成吞吐量上分别提高了 1457.76 倍和 22.88 倍，在签名验证吞吐量上分别提高了 490.51 倍和 31.27 倍。Kim 等人[32]提出了一种使用不同 CUDA 内核对 SPHINCS$^+$不同子算法进行并行实现的方案。该方案通过采用不同的 CUDA 内核，能够根据子算法的特性定制线程配置，从而带来更灵活的并行计算模式。然而，为了在不同 CUDA 内核间进行数据传输，需要将中间结果存储至全局内存，这大大增加了内存访问开销。Kim 等人[32]在 NVIDIA RTX 3090(下称 RTX 3090)上对 SHA-256 实例化的 SPHINCS$^+$进行了加速实现，同样在 128-f 模式下，SPHINCS$^+$-SM3 的实现在签名生成和签名验证方面的吞吐量分别是 Kim 等人[32]的 0.51 倍和 0.68 倍，性能上的差距有 2 方面原因：1)SM3 相较于 SHA-256 来说不具性能优势。2)DCU 相较于 RTX 3090 存在较大的性能劣势。Shen 等人[33]在 NVIDIA A100(下称 A100)上对 CRYSTALS-DILITHIUM(下称 DILITHIUM)进行了加速实现。由于 DILITHIUM 官方未提供安全等级 1 下的具体参数，我们与安全等级 2 下的 DILITHIUM 实现进行性能比较。即便 SPHINCS$^+$-SM3 安全等级更低，但在签名生成和签名验证吞吐量上，SPHINCS$^+$-SM3 的实现分别是 Shen 等人[33]实现的 0.030 倍和 0.095 倍。性能上的巨大差距可以归结为 2 点：1)SPHINCS$^+$算法本身在性能上相较于 DILITHIUM 存在很大劣势。2)A100 作为 NVIDIA 专为高性能计算设计的 GPU，其计算能力远超 DCU。

综合上述分析，可以得出如下结论。一方面，SPHINCS$^+$-SM3 相较于 SPHINCS$^+$的其他算法实例在性能上不占优势，而相较于 DILITHIUM 更存在显著的性能劣势；另一方面，DCU 的计算能力相较于 NVIDIA 研发的 GPU 仍存在较大差距。这反映出在硬件层面，DCU 相对于高端 GPU 在计算性能上还存在较大的提升空间。因此，SPHINCS$^+$-SM3 在国产硬件平台上的加速实现还有很大的潜力可以挖掘，既包括对算法本身的进一步优化，也包括针对硬件平台的持续改进。

5 总 结

一直以来，数字签名在信息安全领域发挥着至关重要的作用，然而，在后量子时代，当前主流的数字签名算法面临失效风险。SPHINCS$^+$作为一种能够抵抗量子计算机攻击的数字签名框架，将在未来发挥越来越重要的作用。但 SPHINCS$^+$过慢的计算速度难以满足现代密码系统所需的高吞吐量需求，从而影响了其实际应用价值。随着 GPU 并行计算技术的不断发展进步，SPHINCS$^+$的性能优化迎来了新的机遇。本文重点研讨了如何利用国产并行计算平台 DCU 来加速基于国产哈希算法 SM3 实例化的 SPHINCS$^+$算法。从提升内存拷贝效率、优化 SM3 算法、改进 SPHINCS$^+$的计算流程以及选择最佳计算并行度 4 个方面入手，对 SPHINCS$^+$-SM3 进行了全面优化。本研

究对 SPHINCS$^+$-SM3 的 128-f 模式进行了实现(所提出的优化策略同样适用于其他模式)。实验结果表明,在同等安全强度下,与基于 CPU 的实现相比,我们的 DCU 实现在签名生成和签名验证方面吞吐量分别提高了 2 603.87 倍和 1 281.98 倍。这一成果不仅显著提升了 SPHINCS$^+$的计算效率,增强了其实用性,同时也加快了后量子密码算法的国产化进程。我们的 DCU 实现适用于数据流量较大以及需要在短时间内对大量签名生成或签名验证请求进行处理的场景,例如金融交易、区块链和在线服务平台等。

作者贡献声明:宁祎静提出实现方案、开展实验并完成论文撰写;董建阔指导实验方案的设计,并参与论文的撰写与修订;周思源参与研究背景的调查和论文的修订;林璟锵为研究方向提供了指导,并参与了论文的修订;孙思维在理论和方案实现上进行了指导;郑昉昱参与了实验结果的整理与分析;葛春鹏参与了论文的修订。

参 考 文 献

[1] Shor P W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer[J]. SIAM Review, 1999, 41(2): 303−332

[2] Grover L K. Quantum computers can search rapidly by using almost any transformation[J]. Physical Review Letters, 1998, 80(19): 4329−4332

[3] Rivest R L, Shamir A, Adleman L. A method for obtaining digital signatures and public-key cryptosystems[J]. Communications of the ACM, 1978, 21(2): 120−126

[4] Schwabe P, Avanzi R. PQCRYSTALS[EB/OL]. 2017[2024-08-16]. https://pq-crystals.org/

[5] Lyubashevsky V, Ducas L. PQCRYSTALS[EB/OL]. 2017[2024-08-16]. https://pq-crystals.org/

[6] Prest T, Fouque P A. Falcon-sign[EB/OL]. 2017[2024-08-16]. https://falcon-sign.info/

[7] Hülsing A, Bernstein D J. SPHINCS$^+$[EB/OL]. 2019[2024-08-16]. https://sphincs.org/

[8] National Institute of Standards and Technology. Federal Information Processing Standards Publication 180−4[S]. Gaithersburg: National Institute of Standards and Technology, 2012

[9] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions[S]. Gaithersburg: National Institute of Standards and Technology, 2015

[10] National Cryptography Administration. Announcement of the National Cryptography Administration on the Release of SM3 Password Hash Algorithm[EB/OL]. 2010[2024-08-18]. https://www.oscca.gov.cn/sca/xxgk/2010-12/17/content_1002 389 .shtml (in Chinese)
(国家密码管理局. 国家密码管理局关于发布《SM3 密码杂凑算法》公告[EB/OL]. 2010[2024-08-18]. https: //www.oscca.gov.cn/sca/xxgk/2010-12/17/content_10023 89.shtml)

[11] Hülsing A, Butin D, Gazdag S, et al. XMSS: Extended Merkle Signature Scheme[S/OL]. 2018[2024-08-18]. https://www.rfc-editor.org/info/rfc8 391

[12] Kerry C F, Gallagher P D. FIPS 186−4 Digital Signature Standard (DSS)[S]. Gaithersburg: National Institute of Standards and Technology, 2013

[13] Alagic G, Apon D, Cooper D, et al. Status report on the third round of the NIST post-quantum cryptography standardization process[R/OL]. Gaithersburg: National Institute of Standards and Technology, 2022[2024-04-06]. https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf

[14] Sun Siwei, Liu Tianyu, Guan Zhi, et al. SPHINCS$^+$-SM3: SM3-based stateless digital signature scheme[J]. Journal of Cryptologic Research, 2023, 10(6): 1266−1278 (in Chinese)
(孙思维, 刘田雨, 关志, 等. SPHINCS$^+$-SM3: 基于 SM3 的无状态数字签名算法[J]. 密码学报, 2023, 10(6): 1266−1278)

[15] Dong Jiankuo, Lu Sheng, Zhang Pinchang, et al. G-SM3: High-performance implementation of GPU-based SM3 Hash function[C]// Proc of the 28th IEEE Int Conf on Parallel and Distributed Systems. Piscataway, NJ: IEEE, 2023: 201−208

[16] Eum S W, Kim H J, Kwon H D, et al. Implementation of SM4 block cipher on CUDA GPU and its analysis[C]//Proc of the 8th Int Conf on Platform Technology and Service. Piscataway, NJ: IEEE, 2022: 71−74

[17] Zhou Tian, Zheng Fangyu, Lin Jingqiang, et al. On software implementations of post-quantum cryptography[J]. Journal of Cryptologic Research, 2024, 11(2): 308−343 (in Chinese)
(周天, 郑昉昱, 林璟锵, 等. 后量子密码算法的软件实现研究[J]. 密码学报, 2024, 11(2): 308−343)

[18] National Cryptography Administration. The Executive meeting of The State Council Deliberated and Adopted the Regulations on the Administration of Commercial Passwords (Draft Revision)[EB/OL]. 2023[2024-08-22]. https://www.oscca.gov.cn/sca/xwdt/2023-04/20/content_10610 05.shtml (in Chinese)
(国家密码管理局. 国务院常务会议审议通过《商用密码管理条例(修订草案)》[EB/OL]. 2023[2024-08-22]. https: //www.oscca.gov.cn/sca/xwdt/2023-04/20/content_10610 05.shtml)

[19] Adams C, Lloyd S. Understanding Public-key Infrastructure: Concepts, Standards, and Deployment Considerations[M]. Indianapolis: Macmillan Technical Pub, 1999

[20] IBM. IBM quantum computing[EB/OL]. 2023[2024-08-22]. https://www.ibm.com/quantum/

[21] Dierks T, Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.2[S/OL]. 2008[2024-08-20]. https://www.rfc-editor.org/info/rfc5246

[22] Arends R, Austein R, Larson M, et al. DNS Security Introduction and Requirements[S/OL]. 2005[2024-08-20]. https://www.rfc-editor.org/info /rfc4033

[23] Hülsing A. WOTS$^+$−Shorter signatures for Hash-based signature schemes[C]//Proc of the 6th Progress in Cryptology—AFRICACRYPT. Berlin: Springer, 2013: 173−188

[24] Bernstein D J, Hülsing A, Kölbl S, et al. The SPHINCS$^+$ signature framework[C]//Proc of the 26th ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2019: 2129−2146

[25] Buchmann J, Erik D, Hülsing A. XMSS—A practical forward secure signature scheme based on minimal security assumptions[C]//Proc of the 4th Post-Quantum Cryptography—PQCrypto. Berlin: Springer, 2011: 117−129

[26] Hülsing A, Bernstein D J, Dobraunig C, et al. SPHINCS$^+$: Submission to the NIST post-quantum project, v3.1[R/OL]. Gaithersburg: National Institute of Standards and Technology, 2022[2023-11-10]. https://sphincs.org/data/sphincs+-r3.1-specification.pdf

[27] HYGON. DCU development and usage documentation[EB/OL]. 2023[2024-12-21]. https://developer.sourcefind.cn/gitbook/dcu_developer/DeveloperGuide/dcu_programming/DCU_programming_prefix.html (in Chinese)
(海光. DCU 开发与使用文档[EB/OL]. 2023[2024-12-21]. https://developer.sourcefind.cn/gitbook/dcu_developer/DeveloperGuide/dcu_programming/DCU_programming_prefix.html)

[28] Owens J D, Houston M, Luebke D, et al. GPU computing[J]. Proceedings of the IEEE, 2008, 96(5): 879−899

[29] NVIDIA. CUDA C++ programming guide 9.0[EB/OL]. 2017[2024-08-24]. https://docs.nvidia.com/cuda/cuda-c-progra mming-guide/

[30] Berthet Q, Upegui A, Gantel L, et al. An area efficient SPHINCS$^+$ post-quantum signature coprocessor[C]//Proc of the 35th IEEE Int Parallel and Distributed Processing Symp Workshops. Piscataway, NJ: IEEE, 2021: 180−187

[31] Amiet D, Leuenberger L, Curiger A, et al. FPGA-based SPHINCS$^+$ implementations: Mind the glitch[C]//Proc of the 23rd Euromicro Conf on Digital System Design. Piscataway, NJ: IEEE, 2020: 229−237

[32] Kim D C, Choi H J, Seo C. Parallel implementation of SPHINCS$^+$ with GPUs[J]. IEEE Transactions on Circuits and Systems I, 2024, 71(6): 2810−2823

[33] Shen Shiyu, Yang Hao, Dai Wangchen, et al. High-throughput GPU implementation of Dilithium post-quantum digital signature[J]. IEEE Transctions on Parallel and Distributed Systems, 2024, 35(11): 1964−1976

Ning Yijing, born in 2002. Master. His main research interests include post-quantum cryptography and high-performance computing.
宁祎静, 2002 年生。硕士。主要研究方向为后量子密码学、高性能计算。

Dong Jiankuo, born in 1992. PhD, associate professor. Member of CCF. His main research interests include public key cryptography, post-quantum cryptography, and parallel computing.
董建阔, 1992 年生。博士, 副教授。CCF 会员。主要研究方向为公钥密码学、后量子密码学、并行计算。

Zhou Siyuan, born in 1999. Master. His main research interests include post-quantum cryptography and parallel computing.
周思源, 1999 年生。硕士。主要研究方向为后量子密码学、并行计算。

Lin Jingqiang, born in 1978. PhD, professor. Member of CCF. His main research interests include cryptographic engineering and system security.
林璟锵, 1978 年生。博士, 教授。CCF 会员。主要研究方向为密码工程、系统安全。

Sun Siwei, born in 1985. PhD, professor. Member of CCF. His main research interests include the automation of symmetric cryptography algorithm design and analysis, optimization and secure implementation of cryptographic algorithms, and symmetric cryptography analysis based on quantum computing.
孙思维, 1985 年生。博士, 教授。CCF 会员。主要研究方向为自动化对称密码算法设计与分析、密码算法的优化与安全实现、基于量子计算的对称密码分析。

Zheng Fangyu, born in 1988. PhD, associate professor. Member of CCF. His main research interest includes applied cryptography.
郑昉昱, 1988 年生。博士, 副教授。CCF 会员。主要研究方向为应用密码学。

Ge Chunpeng, born in 1987. PhD, professor. Member of CCF. His main research interests include data security and privacy protection in cloud computing, blockchain, and privacy computing.
葛春鹏, 1987 年生。博士, 教授。CCF 会员。主要研究方向为云计算中的数据安全与隐私保护、区块链、隐私计算。