海光处理器上后量子签名算法的AVX2优化实现

王悦彤[1] 周 璐[1),2)] 杨 昊[3)] 刘 哲[4)]

[1)](南京航空航天大学计算机科学与技术学院 南京 211106)

[2)](软件新技术与产业化协同创新中心 南京 210023)

[3)](香港城市大学 香港 999077)

[4)](之江实验室 杭州 311100)

摘 要 随着量子计算技术的快速发展,传统密码体系面临着巨大的安全威胁,后量子密码学(PQC)的研究迫在眉睫。在此背景下,格密码凭借其出色的抗量子攻击能力,成为后量子数字签名算法的重要基础。HuFu 算法是我国自主研发的后量子数字签名算法,基于格密码学中的通用格问题,具有良好的安全性和应用前景,目前已提交至美国国家标准技术研究院(NIST)进行标准化评估。但 HuFu 算法在性能上仍有提升空间,特别是在算法复杂度、内存效率和并行计算能力方面,同时还需增强对不同硬件和指令集的兼容性。为此,本文基于国产的海光处理器,充分发挥其高并行性、低能耗和高吞吐量的优势,为 HuFu 算法的高效实现提供了强有力的硬件支持。同时,结合 256 位高级向量扩展(AVX2)指令集,这一广泛应用的单指令多数据(SIMD)技术,进一步增强了算法的并行计算能力,从而有效提升了整体性能。本文综合考虑矩阵乘法优化、指令集加速、编码处理简化和内存访问效率等多个方面,采用一系列算法和技术优化,旨在显著提升计算速度、减少资源消耗,并提高签名生成与验证的整体性能。具体而言,本文的优化方案包括多个关键技术点:首先,结合 Strassen 算法优化矩阵乘法,显著提升了计算速度并减少了资源消耗;其次,采用 AVX2 指令集对非对称数字的范围变体编码(rANS)进行了优化,加快了签名生成与验证的速度;此外,针对 rANS 编码中符号位处理复杂且耗时的问题,采用无符号参数来实现高效的签名和验证处理,简化了计算流程并减少了运算开销;最后,通过设计合理的函数接口和内存访问优化技术,提高了签名和验证阶段的内存使用效率,减少了寄存器的频繁写入。与原有的 HuFu 算法 AVX2 实现方案相比,本文提出的优化方案在密钥生成、签名在线阶段、签名离线阶段以及总的签名和验证阶段的时钟周期消耗分别减少了约 46%、54%、45%、30%和 46%。高效的签名算法能够在高并发环境中提升处理能力,增强系统的稳定性和安全性,更好地保护后量子密码数据免受量子计算威胁,同时推动国产后量子密码技术的发展。

关键词 后量子密码;格密码;高级向量拓展;矩阵乘法;内存访问优化;海光处理器

中图法分类号 TP391 **DOI 号** 10.11897/SP.J.1016.2025.01714

AVX2 Optimized Implementation of Post-Quantum Signature Algorithm on Hygon Processor

WANG Yue-Tong[1] ZHOU Lu[1),2)] YANG Hao[3)] LIU Zhe[4)]

[1)](College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106)

[2)](Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 210023)

[3)](City University of Hong Kong, Hong Kong 999077)

[4)](Zhejiang Lab, Hangzhou 311100)

Abstract With the rapid development of quantum computing technology, traditional cryptographic systems are facing huge security threats, and the research of post-quantum cryptography (PQC) is becoming increasingly imminent. In this context, lattice-based cryptography has

收稿日期:2024-12-28;在线发布日期:2025-04-08。本课题得到国家重点研发计划(2022YFB2702000)、国家自然科学基金委员会重点项目(62132008)、国家自然科学基金委员会联合基金重点项目(U22B2030)、国家自然科学基金委员会面上项目(62472218)、江苏省自然科学基金(BK20220075)资助。**王悦彤**,硕士研究生,主要研究方向为后量子密码、密码工程。E-mail:yuetong_wang@nuaa.edu.cn。**周 璐**(通信作者),博士,教授,主要研究领域为密码工程、区块链。E-mail:lu.zhou@nuaa.edu.cn。**杨 昊**,博士,主要研究方向为公钥密码学、密码工程。**刘 哲**,博士,教授,主要研究领域为公钥密码学、密码工程。

become an important foundation tool for post-quantum digital signature algorithms with its excellent ability to resist quantum attacks effectively. The HuFu algorithm is a post-quantum digital signature algorithm independently developed by our country. It is based on the universal lattice problem in lattice cryptography and has good security and application prospects. It has been submitted to the National Institute of Standards and Technology (NIST) of the United States for standardization evaluation. However, the performance of the HuFu algorithm still needs to be improved, especially in terms of algorithm complexity, memory efficiency, and parallel computing capabilities. It also needs to enhance compatibility with different hardware and instruction sets. To this end, this paper is based on the domestic Hygon Processor, giving full play to its advantages of high parallelism, low energy consumption and high throughput, and providing strong hardware support for the efficient implementation of the HuFu algorithm. At the same time, combined with the 256-bit Advanced Vector Extension (AVX2) instruction set, this widely used single instruction multiple data (SIMD) technology, further enhances the parallel computing capability of the algorithm, thereby effectively improving the overall performance. This paper comprehensively considers multiple aspects such as matrix multiplication optimization, instruction set acceleration, encoding processing simplification and memory access efficiency, and adopts a series of algorithms and technical optimizations to significantly improve the calculation speed, reduce resource consumption, and improve the overall performance of signature generation and verification. Specifically, the optimization scheme of this paper includes several key technical points: First, the matrix multiplication is optimized in combination with the Strassen algorithm, which is obvious. The calculation speed is greatly improved and resource consumption is reduced. Secondly, the range variant encoding of asymmetric numbers (rANS) is optimized by using the AVX2 instruction set, which speeds up the speed of signature generation and verification. In addition, in view of the complex and time-consuming problem of sign bit processing in rANS encoding, unsigned parameters are used to achieve efficient signature and verification processing, which simplifies the calculation process and reduces the calculation overhead. Finally, by designing a reasonable function interface and memory access optimization technology, the memory usage efficiency in the signature and verification stages is improved, and the frequent writing of registers is reduced. Compared with the original AVX2 implementation of the HuFu algorithm, the optimization scheme proposed in this paper reduces the clock cycle consumption in key generation, online signature stage, offline signature stage, and the total signature and verification stage by about 46%, 54%, 45%, 30% and 46% respectively. Efficient signature algorithms significantly enhance processing capabilities in high-concurrency computing environments, improve overall system stability and security, better protect sensitive post-quantum cryptographic data from various quantum computing threats, and promote the continued development of domestic post-quantum cryptographic technology advancements.

Keywords　post-quantum cryptography; lattice-based cryptography; advanced vector expansion; matrix multiplication; memory access optimization; Hygon Processor

1　引　言

随着量子计算技术的发展，基于经典的数学难题如因数分解和离散对数的密码学体系面临着前所未有的挑战。量子计算机通过利用 Shor 算法[1]等高效手段，能够轻松破解传统加密系统，这严重威胁到了现有的加密和数字签名机制。为了应对量子计算带来的巨大威胁，能够抵抗量子计算攻击的密码，即后量子密码(Post-Quantum Cryptography，PQC)

成为国内外研究的热点。

这一背景下,格密码学作为后量子密码学的重要分支,凭借其优越的计算性能和强大的抗量子攻击能力,逐渐成为众多候选方案中的核心。美国国家标准技术研究院(National Institute of Standards and Technology,NIST)于 2016 年启动了后量子密码标准征集活动,以应对量子计算带来的威胁。经过多年的筛选和评估之后,NIST 于 2022 年 5 月确定了四个 PQC 的标准算法,涵盖公钥加密、密钥封装和数字签名,其中有三个算法是基于格的。这一结果进一步凸显了格密码在后量子密码学中的重要地位。

为了进一步丰富后量子数字签名的选择,NIST 于 2022 年 9 月发起了新一轮数字签名提案征集,旨在扩充其后量子签名方案库。鉴于在前三轮评估中已经标准化了两种基于结构格的签名方案,NIST 特别希望能引入基于其他安全假设的通用签名方案,以及具有短签名和快速验证特性的签名方案,从而提升方案的多样性、实用性和安全性。

因此,HuFu 算法作为我国自主研发的后量子数字签名算法,于 2023 年 6 月提交给 NIST 的标准化过程,表现出良好的安全性和应用前景。HuFu 是一种后量子数字签名方案,其安全性基于通用格上标准最坏情况问题的难度。与 CRYSTALS-Dilithium[2] 和 Falcon[3] 相比,HuFu 的设计有很大不同之处。特别是在面对通用格上标准最坏情况问题的复杂性时,HuFu 提供了更强的安全保障。该方案创新性地整合了 Gentry、Peikert 和 Vaikuntanathan[4] (GPV)提出的格陷门框架(Gadget Trapdoor),构建了高效的哈希与签名方案。方案采用构件陷门构造[5],在硬随机格上实现实例化,并利用紧凑型构件技术[6],从而实现了整体性能的显著优化。这种设计使得现有的量子算法并不能有效地解决通用格上的最坏情况问题,从而保证了 HuFu[7] 的抗量子安全性。

基于 Micciancio 和 Peikert 提出的构件陷门框架[5],HuFu 算法的构建充分利用了这一结构的优势。该框架中的陷门是公钥矩阵和构件矩阵之间的线性关系,这一关系并不构成完整的基底。利用这一线性关系,困难随机格上的陷门采样可以转换为构件格上的采样,后者既方便又快速。尽管构件型方案的签名尺寸较大,但通过采用近似陷门技术和紧凑型构件技术[6],其签名大小已与 Fiat-Shamir 方案相当。HuFu 利用这一框架构建了在线/离线结构,使其在线操作简单、快速且完全在整数范围内。这种在线/离线结构虽然在理论上提高了操作的灵活性,但在实际应用中,离线阶段的复杂计算和数据传输延迟可能导致整体签名速度的下降。此外,在线阶段的简单操作如果依赖于资源受限的设备,也可能限制其响应速度。因此,本文主要聚焦于 HuFu 算法签名的在线/离线部分的优化实现。

本文是基于 256 位的高级向量拓展(AVX2①),针对不同安全级别下的 HuFu 算法进行高速的实现。AVX2 是英特尔众多的处理器中一种强大的单指令多数据(Single Instruction Multiple Data,SIMD) 指令集架构,能并行处理多个数据元素。具体来说,AVX2 可以同时处理 8 组 32 位数据或者 16 组 16 位数据。在 HuFu 算法的签名生成和验证过程中,签名值的位数通常小于 16 位,使得这些操作适合利用 AVX2 指令集的并行处理能力。特别是在签名生成阶段和签名验证阶段,通过并行处理多个签名值,AVX2 可以显著加速这些计算过程,提高整个算法的效率。因此,AVX2 的并行计算特性可以有效地用于 HuFu 算法中的签名压缩和解压部分。

尽管现在英特尔处理器中已经引入了比 AVX2 更先进的指令集进行优化。例如,基于 512 位的高级向量拓展(AVX-512),但由于 AVX2 的普及率更高,很多文献仍主要使用 AVX2 来优化密码算法。因此本文选择利用 AVX2 对 HuFu 进行优化,以便与其他后量子密码算法进行性能对比。这一优化行为也为后续在 AVX-512 上的扩展奠定了基础。同时,海光处理器凭借其出色的并行计算能力,为这一优化提供了强大的硬件支持。海光处理器是我国自主研发的一款高性能计算产品,具有卓越的高并行计算能力、低能耗和高吞吐量的特性,特别适合用于后量子密码算法的优化工作。综合考虑矩阵乘法优化、指令集加速、编码处理简化和内存访问效率,本文采用了一系列算法和内存优化技术,旨在显著提升 HuFu 算法在签名生成与验证过程中的计算速度和资源利用效率。具体来说,本文的主要贡献包括以下:采用 Strassen 算法优化矩阵乘法;通过引入 AVX2 指令加速 rANS 编码过程;通过无符号参数传递简化符号位处理,提高签名生成和验证的效率;以及设计高效的函数接口并结合内存访问优化,确保在签名和验证阶段减少不必要的内存写入,从而

① Intel Corporation, Intel 64 and IA-32 Architectures Software Developer's Manual, 2006

提升整体性能。

本文的主要贡献包含以下几个关键部分。在引言部分，本文不仅阐述了量子计算技术对传统密码系统带来的安全威胁，论证了后量子密码学研究的必要性和紧迫性，还详细概述了本研究在 HuFu 算法优化方面的创新性贡献：(1) 提出了基于 Strassen 算法的矩阵运算优化方案，显著降低了计算复杂度；(2) 设计了创新的访存优化机制，有效提升了内存访问效率；(3) 开发了改进的 rANS 编码方案，在保证安全性的同时提高了编码效率。这些理论创新为后量子签名算法的优化研究提供了新的思路和方法。

背景知识章节为研究奠定了坚实的理论基础，系统介绍了 HuFu 算法的符号定义与代数结构、算法核心框架、相关算子理论，以及 Strassen 算法在矩阵乘法中的应用原理，特别强调了这些理论基础与算法优化的内在联系。

算法优化策略章节是本研究的核心内容，探讨了三个关键优化方案的具体实现方法。实现优化技术部分通过伪代码和实现细节的逐步呈现，确保了优化过程的透明性和可实现性。特别值得一提的是，本研究在国产海光处理器平台上的实现，展示了算法与国产硬件的良好适配性，为推进国产处理器在后量子密码计算领域的应用提供了重要参考。

实验结果章节通过详实的实验数据和对比分析，客观展示了各项优化策略带来的性能提升。最后，本文不仅总结了研究成果，还深入探讨了 HuFu 算法未来的研究方向，包括进一步优化算法以适应更多硬件平台、探索算法在物联网等新兴领域的应用等，为后续研究提供了明确的方向。

本研究的创新性主要体现在：(1) 首次将Strassen 算法应用于后量子签名计算的优化；(2) 提出了面向国产处理器的定制化优化方案；(3) 建立了完整的性能评估体系。这些成果不仅推动了 HuFu 算法的实用化进程，也为后量子密码算法的优化研究提供了新的思路和方法。

2　背景知识

2.1　符号表示

内积与范数。给定 $u,v\in R^n$，内积定义为 $\langle u,v\rangle=\sum_{i=1}^{n}u_iv_i$，$u$ 的 l_2 范数定义为 $\|u\|=\sqrt{\langle u,u\rangle}$。

环。环 $R=\mathbb{Z}[X]/(X^n+1)$，表示在模 X^n+1 的多项式环。定义环 $R_q=\mathbb{Z}_q[X]/(X^n+1)$，其中 q 是素数。在环 R 和 R_q 的结构中，用粗体小写字母表示为向量，粗体大写字母表示为矩阵。所有向量均视为列向量，并使用 $\boldsymbol{A}^{\mathrm{T}}$ 表示转置。该环结构为带错误学习(LWE)及其变体提供了基础，支持高效的多项式运算和安全性分析。

奇异值与正定性质。矩阵 $\boldsymbol{A}$ 的最大奇异值定义为 $s_1(\boldsymbol{A})=\max_{\boldsymbol{x}\neq 0}\frac{\|\boldsymbol{Ax}\|}{\|\boldsymbol{x}\|}$，对于对称矩阵 $\boldsymbol{\Sigma}$，其正定性定义为 $\boldsymbol{\Sigma}>0$，当且仅当对于所有的非零向量 $\boldsymbol{z}$，都有 $\boldsymbol{z}^t\boldsymbol{\Sigma z}>0$。

Gram 根与 Cholesky 分解。如果存在矩阵 $\boldsymbol{B}$ 使得：$\boldsymbol{\Sigma}=\boldsymbol{BB}^t$，则称 $\boldsymbol{B}$ 为 $\boldsymbol{\Sigma}$ 的 Gram 根。Cholesky 分解是一种常用方法来找到 $\boldsymbol{\Sigma}\in R^{n\times n}$ 的 Gram 根。

中心二项分布采样。参数为 η 的中心二项分布 B_η 表达式为 $X=\sum_{i=1}^{\eta}(ai-bi)X$，其中 $a_i,b_i\leftarrow U(0,1)$。当 $X\leftarrow B_\eta$，期望为 0，标准差为 $\sqrt{\frac{\eta}{2}}$。

短整数解问题(Short Integer Solution，SIS)[8]。SIS 问题旨在给定一个均匀随机矩阵 $\boldsymbol{A}$ 和一个正整数 B 的条件下，找出一个非零整数的向量 $\boldsymbol{x}$ 使得矩阵乘法满足 $\boldsymbol{Ax}=0 \bmod Q$ 且 $\|\boldsymbol{x}\|\leq B$。非齐次短整数解问题则是要求找到一个向量 $\boldsymbol{x}$，使得 $\boldsymbol{Ax}=\boldsymbol{x} \bmod Q$，其中 $\boldsymbol{y}$ 是给定的向量。

带错误学习(Learning With Errors，LWE)[9]。给定一个秘密向量 $\boldsymbol{s}$，其任务是从多个独立样本中区分出来样本是否来自噪声分布的线性组合，形式为 $b=\langle \boldsymbol{a},\boldsymbol{s}\rangle+\boldsymbol{e} \bmod Q$ 还是均匀分布。这里 $\boldsymbol{a}$ 是随机生成的向量，$\boldsymbol{e}$ 是从噪声分布中抽取[7]。

可拓展输出函数(Extendable Output Function，XOF)。基于 SHAKE-256 的可拓展输出函数。该算法输入为一个 32 字节的种子，输出一个矩阵 $\boldsymbol{A}\in\mathbb{Z}_Q^{m\times n}$。首先计算常数 b 和 d，其中 b 是 Q 右移 16 位的结果，d 是 $2mnb$。然后算法通过 SHAKE256 对种子和迭代索引 i 进行哈希，生成长度为 $d/4$ 的比特串 $c_{i,j}$。接着通过双重循环遍历生成比特串，将其映射到矩阵 $\boldsymbol{A}$ 的元素。每个元素的计算涉及一个权重向量 $\boldsymbol{g}$，该向量由不同的常数构成，最终返回生成的矩阵 $\boldsymbol{A}$。

压缩和解压缩。压缩算法将向量 $\boldsymbol{x}\in\mathbb{Z}^{n+m}$ 转换为字符串 str。首先，生成符号标志 s_i(表示元素符号)，并计算每个元素的高位部分 $h_i=|x_i|\gg 7$ 和低位部分 $l_i=|x_i|\,\&\,0x7f$。接下来，使用 rANS 编码对高位部分进行编码，得到编码结果 $strh$。然后，将

符号标志和低位部分结合形成 $strl$。最终字符串形式为 $str=0x80\|strh\|strl$，若长度超过 L 则返回失败标志 $\perp$。解压缩算法通过解析 str 提取高位和低位部分，利用 rANS 解码恢复 h_i，然后根据符号标志和低位部分重构每个元素 x_i，最终返回恢复的向量 $\boldsymbol{x}$。

采样函数(SampleZd)。用于从分布 D_{Z+c,r^2} 中进行采样。首先，输入余数类 $c\in\mathbb{Z}/q$，计算其小数部分 c'，并确定其是否在特定余数类 S_c 中，从而生成标志 b。接着，根据 b 和 c' 计算偏移量 h。然后从均匀分布中生成一个 72 位随机数 u，并初始化样本 $z_h=0$。通过循环来遍历 i 从 -12 到 12，利用随机数 u 和查表的结果更新 z_h，实现采样过程。接下来，将 z_h 加上偏移 $z_h=0$ 的变量 h，以得到从 D_{Z+h,r^2} 中的样本。最后，通过将 z_h 乘以 $(2b-1)$ 得到最终样本 z，确保其来自期望的分布 D_{Z+c',r^2}。

2.2 HuFu 算法

HuFu 算法是一种后量子数字签名方案，专门设计用于抵御量子计算带来的安全威胁。其核心创新在于巧妙结合了随机数生成、矩阵运算和高斯分布采样等技术，构建了一个兼具高效性和安全性的数字签名框架。该算法的密钥生成依赖于一个均匀分布的随机数生成器。通过该随机数生成器生成 256 位的随机种子，进而利用扩展输出函数(XOF)。本方案采用 SHAKE256，因其支持可变长度的输出，提供了更大的灵活性，适应不同应用需求。此外，SHAKE256 在生成伪随机输出时，能够有效构建重复生成矩阵和误差矩阵，这些矩阵作为算法的基础，确保了密钥对生成时具有良好的随机性和安全性。根据 HuFu 在 NSIT 提交的技术文档中的对比结果(第 14 页的表格 3)，尽管 AES256 在某些实现中的效果优于 SHAKE256，但SHAKE256的灵活性和扩展性使其在实际应用中更具优势，尤其是在需要变长输出的场景中。因此，选择 SHAKE256 作为 XOF 函数满足了对安全性和性能的综合需求。

在 HuFu 算法中，推荐的参数分为三组，分别为 NIST-1、NIST-3 和 NIST-5 这三种安全级别，具体参数如表 1 所示。其中，维度 (m,n) 表示矩阵的大小，模数 Q 定义了操作的数值范围，gadget 的参数 (p,q) 用于表示计算的精度，接受边界值 B 是用于判断有效性的阈值，Sig size 表示签名大小(以字节为单位)，Pk sig 表示公钥大小(以千字节为单位)。这些参数的设定依据是已知攻击的核心 SVP 难度估计。

表 1 HuFu_NIST1 参数集

方案	(m,n)	Q	(p,q)	B	Sig size	Pk sig	rec.BKZ	rec.C-SVP	rec.Q-SVP	Forg.BKZ	Forg.C-SVP	Forg.Q-SVP
HuFu_NIST1	(736,848)	2^{16}	$(2^{12},2^4)$	62521	2455	1059	443	129	117	438	128	116
HuFu_NIST3	(1024,1232)	2^{17}	$(2^{13},2^4)$	108493	3540	2177	663	194	176	659	192	175
HuFu_NIST5	(1312,1552)	2^{17}	$(2^{13},2^4)$	130320	4520	3573	878	256	233	883	258	234

BKZ(Block Korkine Zolotarev)块大小用于描述密钥恢复的复杂度。块大小的选择会影响算法效率和成功恢复密钥的概率，通常较大的块大小可以提高安全性，但也会增加计算复杂度和资源消耗。其中，接受的边界值 B 作为有效性判断的关键阈值，用于验证计算结果是否满足预设的安全标准。表 1 中 rec.BKZ 表示在密钥恢复过程中使用的 BKZ 算法的块大小。rec.C-SVP 表示在经典计算模型下基于核心 SVP 问题的安全强度(数值越大，安全性越高)。rec.Q-SVP 表示在量子计算模型下基于核心 SVP 问题的安全强度，反映了算法在量子环境下的抗攻击能力。

Forg.BKZ 表示在伪造攻击的上下文中使用 BKZ 算法的块大小。块大小会影响攻击的效率，较大的块大小通常能够更有效地寻找密钥或伪造签名，但同样会增加计算复杂度[10]。Forg.C-SVP 表示在经典计算模型下攻击者对伪造签名的难度，通常通过解决核心短整数问题来衡量。Forg.Q-SVP 表示在量子计算模型下攻击者对伪造签名的难度。在量子计算环境中，攻击的复杂性可能会降低，因此该值通常小于经典安全性指标。

这些参数的设置综合考虑了算法效率与安全性之间的平衡。接受边界值 B 用于判断计算结果是否为有效的阈值，确保生成的结果符合预期的标准。表 1 中还列出了经典和量子环境下的核心 SVP 安全性指标(core-SVP)，这些参数的设定依据是已知攻击的核心 SVP 难度估计，充分考虑了当前最先进的攻击方法及其计算复杂度。

HuFu 算法的整体流程包括密钥生成、签名生成(分为离线和在线过程)以及签名验证三个主要部分。在密钥生成过程中，算法通过矩阵运算和条件判断生成满足特定阈值的矩阵[5]。签名生成过程则分为离线阶段和在线阶段：离线阶段中生成必要的矩阵和向量，而在线阶段则根据输入的消息、私钥和

随机盐值计算最终签名。在签名验证阶段，算法通过恢复矩阵、检查签名长度、解压签名并计算签名向量，最终确定签名的有效性。

整个方案的实现包含以下三部分：密钥生成阶段(Key-Gen)、签名的在线阶段(Sign.online)、签名的离线阶段(Sign.offline)和签名的验证阶段(Verify)。HuFu 具体的算法内容描述如算法 1～算法 3 所示。

算法 1.　HuFu.Key-Gen

输入：无

输出：公钥 pk，私钥 sk

1. $seed_{\hat{A}} \leftarrow U((0,1)^{256})$，$\hat{\boldsymbol{A}} \leftarrow XOF(seed_{\hat{A}})$
2. REPEAT
3. $(\boldsymbol{S},\boldsymbol{E}) \leftarrow \chi^{n\times m} \times \chi^{m\times m}$
4. $\boldsymbol{\Sigma}_p = \sigma^2 \boldsymbol{I}_{n+2m} - r^2 \cdot \begin{bmatrix} \boldsymbol{E} \\ \boldsymbol{S} \\ \boldsymbol{I}_m \end{bmatrix} \cdot [\boldsymbol{E}^t \boldsymbol{S}^t \boldsymbol{I}_m]$
5. UNTIL $\boldsymbol{\Sigma}_p > \bar{r}^2$
6. $\boldsymbol{B} = p \cdot \boldsymbol{I} - \boldsymbol{A}\boldsymbol{S} + \boldsymbol{E} \bmod Q$
7. $\boldsymbol{C} = \begin{bmatrix} \boldsymbol{L}_{33} & \boldsymbol{L}_{32} & \boldsymbol{L}_{31} \\ & \boldsymbol{L}_{22} & \boldsymbol{L}_{21} \\ & & \boldsymbol{L}_{11} \end{bmatrix} \leftarrow BlockCholesky$

$(\boldsymbol{\Sigma}_p - \bar{r}^2 \boldsymbol{I})$ with $\boldsymbol{L}_{22} \in \mathbb{R}^{n\times n}$, $\boldsymbol{L}_{32} \in \mathbb{R}^{m\times n}$, $\boldsymbol{L}_{33} \in \mathbb{R}^{m\times m}$

算法 2.　HuFu.Sign

输入：m，sk，可接受边界值 B

输出：$s = (salt, str)$

签名离线阶段(Sign.online 阶段)：

1. $\hat{\boldsymbol{A}} \leftarrow XOF(seed_{\hat{A}})$，$\boldsymbol{A} \leftarrow [\boldsymbol{I} \hat{\boldsymbol{A}} \boldsymbol{B}]$
2. $p = (p_0, p_1, p_2) \leftarrow SampleP(sk)$ with $p_0 \in \mathbb{Z}^m$, $p_1 \in \mathbb{Z}^n$, $p_2 \in \mathbb{Z}^m$　　$\triangleright p \leftarrow D_{\mathbb{Z}^{n+2m}, \boldsymbol{\Sigma}_p}$
3. $c \leftarrow \boldsymbol{A}p \bmod Q$

签名在线阶段(Sign.offline 阶段)：

1. $salt \leftarrow U(\{0,1\}^{320})$，$u \leftarrow H(m, salt)$
2. $v \leftarrow u - c \bmod Q$
3. $e \leftarrow (v \bmod p)$，$v' \leftarrow (v-e)/p$
4. FOR $i = 1, \cdots, m$ DO
5. $z_i \leftarrow q \cdot Sample\ \mathbb{Z}_d(v_i'/q)$　　$\triangleright z \leftarrow D_{q\cdot \boldsymbol{I}_m + v', r^2}$
6. END FOR
7. $x_0 \leftarrow \boldsymbol{E}\boldsymbol{z} + p_0$，$x_1 \leftarrow \boldsymbol{S}\boldsymbol{z} + p_1$，$x_2 \leftarrow \boldsymbol{z} + p_2$
8. IF $\|(x_0 + e, x_1, x_2)\| > B$ THEN RESTART
9. END IF
10. $str \leftarrow Compress((x_1, x_2))$
11. IF $str = \perp$ THEN RESTART
12. END IF

算法 3.　HuFu.Verify

输入：m，$(salt, str)$，pk，可接受边界值 B

输出：接受(Accept)或者拒绝(Reject)

1. $\hat{\boldsymbol{A}} \leftarrow XOF(seed_{\hat{A}})$
2. $x \leftarrow Decompress(str)$
3. IF $x = \perp$ THEN RETURN Reject
4. $(x_1, x_2) \leftarrow x$
5. $u \leftarrow H(m, salt)$，$x_0' \leftarrow (u - \hat{\boldsymbol{A}}x_1 - \boldsymbol{B}x_2) \bmod Q$
6. Accept IF $\|(x_0', x_1, x_2)\| \le B$, OTHERWISE Reject

2.3　rANS 编码

可变范围非对称数字系统编码(rANS)是一种高效的熵编码技术，通过适应符号频率来实现数据紧凑表示。其基本原理是将待编码的数据分割成符号，并根据每个符号的概率分布动态调整编码范围。rANS 编码能够接近熵界限，尤其适合高斯向量等数据类型，提供高效的压缩效果。

在 HuFu 算法中，签名阶段的 rANS 编码过程如算法 4 所示，首先将签名向量的每个元素拆分为符号位、低位和高位。其中，符号位和低位由于几乎呈均匀分布而直接存储，而高位则使用 rANS 进行编码。为了保证编码结果的固定长度，编码时会在开头填充一定数量的字节。这种设计不仅提高了存储效率，还优化了后续的解码过程。

算法 5 中的解码过程则是验签阶段 rANS 编码的逆操作，通过解析编码结果并恢复出原始的签名向量。在 HuFu 算法中，解码过程会使用相同的概率分布和范围更新规则，逐步恢复每个符号，直至完全恢复原始数据。这一过程确保了数据的完整性和一致性。而算法 5 中 rANS 的使用 HuFu 算法中提升了签名的压缩效率，使其在后量子数字签名方案中更具实用性。

算法 4.　rANS 编码算法

输入：需要编码的符号序列 $(sym_1, \cdots, sym_{m+n})$

输出：rANS 编码结果 $(str_{j-3}, \cdots, str_j, str_{j+1}, \cdots, str_{-1})$

1. $x \leftarrow 2^{23}$　　固定编码的初始状态
2. $j \leftarrow -1$
3. FOR $i \leftarrow m+n$ to 1 DO　　符号反向编码
4. WHILE $x \notin I_{sym_i}$ DO　　将状态正常化
5. $str_j \leftarrow x \bmod 2^8$
6. $j \leftarrow j-1$　　逆向排放数据
7. $x \leftarrow \lfloor x/2^8 \rfloor$
8. END WHILE
9. $x \leftarrow C(sym_i, x)$　　符号编码
10. END FOR
11. $(str_{j-3}, \cdots, str_j) \leftarrow PackState(x)$

算法 5.　rANS 解码算法

输入：一个 rANS 编码序列 $(str_1, \cdots, str_N)$

输出：解码后得到符号序列 $(sym_1, \cdots, sym_{m+n})$ 或者 $\perp$ (表示解码失败)

1. IF $N < 4$ THEN RETURN $\perp$　　编码过短

2. $x \leftarrow UnpackState(str_1,\cdots,str_4)$ 读取解码初始状态
3. $j \leftarrow 5$
4. IF $x \notin [2^{23},2^{31})$ THEN RETURN $\perp$ 初始状态超出范围
5. FOR $i \leftarrow 1$ TO $(m+n)$ DO
6. $(sym_i,x) \leftarrow D(x)$ 符号编码
7. WHILE $x \notin [2^{23},2^{31})$ DO 状态正常化
8. IF $j>N$ THEN RETURN $\perp$ 编码过短
9. $x \leftarrow 2^8 x + str_j$
10. $j \leftarrow j+1$
11. END WHILE
12. END FOR
13. IF $x \neq 2^{23}$ THEN RETURN $\perp$
14. IF $j \neq N+1$ THEN RETURN $\perp$

2.4 Strassen 算法

矩阵乘法是一种基本的线性代数运算。给定两个矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$，其乘积 $\boldsymbol{C}=\boldsymbol{AB}$ 的每个元素 $\boldsymbol{C}_{ij}$ 是通过将矩阵 $\boldsymbol{A}$ 的第 i 行与矩阵 $\boldsymbol{B}$ 的第 j 列进行逐元素相乘并求和得到的。尽管传统的矩阵乘法方法简单直观，但其时间复杂度为 $O(n^3)$，在处理大规模矩阵时效率较低[6]。为了解决这个问题，本文使用 Strassen 算法。它通过分治策略优化矩阵乘法的计算过程，从而实现更快的乘法运算。

Strassen[11]算法是一种高效的矩阵乘法算法，由沃尔特·斯特拉森在 1969 年提出。该算法通过将矩阵分割成为四个相等大小的子矩阵，并采用递归方法将乘法次数从传统方法的八次减少到七次，同时结合加法和减法运算，从而将时间复杂度从传统的 $O(n^3)$ 降低到 $O(n^{\log_2 7}n^2)$。这种分治策略在处理大规模矩阵时显著提升了计算效率，且随着矩阵规模的增大，其优势更加明显。此外，Strassen 算法的递归特性使其具备良好的并行化潜力，能够充分利用多核处理器实现高效计算。由于其较低的乘法复杂度，该算法在高性能计算和机器学习等需要处理大规模数据集的领域中表现出优越的性能。Strassen 算法不仅在理论上优化了传统矩阵乘法方法，其在实际应用中的成功实现也证明了其广泛的适用性和重要性，为后续高效算法设计提供了重要的理论基础。

Strassen 算法的基本思路是首先将矩阵 $\boldsymbol{S}'$、$\boldsymbol{A}$ 和 $\boldsymbol{E}'$ 分割成四个相同大小的子矩阵：

$$\boldsymbol{S}'=\begin{pmatrix}\boldsymbol{S}'_{00} & \boldsymbol{S}'_{01}\\ \boldsymbol{S}'_{10} & \boldsymbol{S}'_{11}\end{pmatrix},\ \boldsymbol{A}=\begin{pmatrix}\boldsymbol{A}_{00} & \boldsymbol{A}_{01}\\ \boldsymbol{A}_{10} & \boldsymbol{A}_{11}\end{pmatrix},\ \boldsymbol{E}'=\begin{pmatrix}\boldsymbol{E}'_{00} & \boldsymbol{E}'_{01}\\ \boldsymbol{E}'_{10} & \boldsymbol{E}'_{11}\end{pmatrix},$$

其中，矩阵 $\boldsymbol{S}'$ 和 $\mathbf{E}'$ 的子矩阵维度为 $\bar{n}/2\times\bar{n}/2$，而矩阵 $\boldsymbol{A}$ 的子矩阵维度为 $n/2\times n/2$。传统方法计算为

$$\boldsymbol{B}'=\begin{pmatrix}\boldsymbol{B}'_{00} & \boldsymbol{B}'_{01}\\ \boldsymbol{B}'_{10} & \boldsymbol{B}'_{11}\end{pmatrix},\ \begin{aligned}\boldsymbol{B}'_{00}&=\boldsymbol{E}'_{00}+\boldsymbol{S}'_{00}\cdot\boldsymbol{A}_{00}+\boldsymbol{S}'_{01}\cdot\boldsymbol{A}_{10}\\ \boldsymbol{B}'_{01}&=\boldsymbol{E}'_{01}+\boldsymbol{S}'_{00}\cdot\boldsymbol{A}_{01}+\boldsymbol{S}'_{01}\cdot\boldsymbol{A}_{11}\\ \boldsymbol{B}'_{10}&=\boldsymbol{E}'_{10}+\boldsymbol{S}'_{10}\cdot\boldsymbol{A}_{00}+\boldsymbol{S}'_{11}\cdot\boldsymbol{A}_{10}\\ \boldsymbol{B}'_{11}&=\boldsymbol{E}'_{11}+\boldsymbol{S}'_{10}\cdot\boldsymbol{A}_{01}+\boldsymbol{S}'_{11}\cdot\boldsymbol{A}_{11}\end{aligned}。$$

以上这种分裂计算是八个维度为 $\bar{n}/2\times\bar{n}/2$ 的子矩阵与维度为 $n/2\times n/2$ 的子矩阵的乘积，这与传统方法相比，并没有减少乘法的总数。导致在处理大规模矩阵时的计算效率不高。Strassen 算法的核心思想在于通过优化乘法运算的次数来提高计算效率。通过采用递归的分治策略，将传统的八次乘法减少到七次乘法，同时通过加法和减法完成其他计算。因此 Strassen 的想法是将其改为计算：

$$\begin{aligned}\boldsymbol{M}_0&=(\boldsymbol{S}'_{00}+\boldsymbol{S}'_{11})\cdot(\boldsymbol{A}_{00}+\boldsymbol{A}_{11}),\\ \boldsymbol{M}_1&=(\boldsymbol{S}'_{10}+\boldsymbol{S}'_{11})\cdot\boldsymbol{A}_{00},\\ \boldsymbol{M}_2&=\boldsymbol{S}'_{00}\cdot(\boldsymbol{A}_{01}-\boldsymbol{A}_{11}),\\ \boldsymbol{M}_3&=\boldsymbol{S}'_{11}\cdot(\boldsymbol{A}_{10}-\boldsymbol{A}_{00}),\\ \boldsymbol{M}_4&=(\boldsymbol{S}'_{00}+\boldsymbol{S}'_{01})\cdot\boldsymbol{A}_{11},\\ \boldsymbol{M}_5&=(\boldsymbol{S}'_{10}-\boldsymbol{S}'_{00})\cdot(\boldsymbol{A}_{00}+\boldsymbol{A}_{01}),\\ \boldsymbol{M}_6&=(\boldsymbol{S}'_{01}-\boldsymbol{S}'_{11})\cdot(\boldsymbol{A}_{10}+\boldsymbol{A}_{11}),\\ \boldsymbol{B}'_{00}&=\boldsymbol{E}'_{00}+\boldsymbol{M}_0+\boldsymbol{M}_3-\boldsymbol{M}_4+\boldsymbol{M}_6,\\ \boldsymbol{B}'_{01}&=\boldsymbol{E}'_{01}+\boldsymbol{M}_2+\boldsymbol{M}_4,\\ \boldsymbol{B}'_{10}&=\boldsymbol{E}'_{10}+\boldsymbol{M}_1+\boldsymbol{M}_3,\\ \boldsymbol{B}'_{11}&=\boldsymbol{E}'_{11}+\boldsymbol{M}_0-\boldsymbol{M}_1+\boldsymbol{M}_2+\boldsymbol{M}_5。\end{aligned}$$

以上这种计算只需要进行七次维度为 $\bar{n}/2\times\bar{n}/2$ 的子矩阵与维度为 $n/2\times n/2$ 的子矩阵的乘法，但相比于传统方法，这增加了加法和减法的次数[7]。Strassen 算法以递归方式应用这种分裂，从而在渐近意义上超越了逐块计算的传统方法。例如递归应用 $\log_2\bar{n}$ 次将导致复杂度为 $O(\bar{n}^{\log_2 7-2}\cdot n^2)$。这在渐近意义上优于传统方法的复杂度 $O(\bar{n}n^2)$。最优的递归层数依赖于各种参数和算法实施平台[12]。在某些有关的研究中显示，采用不同的策略可以减少 Strassen 的开销，且该算法在较小维度下也能取得良好的结果。

2.5 AVX2 指令集

AVX2 是英特尔处理器中的一种单指令多数据(SIMD)指令集，支持单条指令处理 256 位寄存器数据，可同时对多组数据进行并行计算，包括 4 组 64 位、8 组 32 位或者 16 组 16 位数据。其丰富的指令集涵盖算术运算、逻辑运算、排列、广播等功能，显著提升数据处理效率，尤其适用于图像处理、科学计算等领域。在 HuFu 算法中，压缩和解压过程采用 rANS 编码，该编码能够高效压缩较大签名值。

由于签名值位数通常小于 16 位，AVX2 指令集非常适合优化其压缩和解压过程。AVX2 中提供的 vpmulhw 和 vpmullw 指令支持 16 组 16 位数据的乘法运算，将 16 位整数的乘积分成高低半部分分别进行计算，显著提高运算效率。此外，指令 vmovdqa 指令可将多个 16 位签名值加载至寄存器，提升带宽利用率；vpmulhuw 指令加速矩阵乘法运算，缩短计算时间；vpextrw 指令则高效提取特定数据，优化解压过程。这些优化使得 rANS 在处理大规模数据时更加高效，确保了算法在实际应用中的性能。因此，采用 AVX2 指令集，使得同一条指令能处理更多数据，充分发挥其并行处理优势。

2.6　海光处理器

海光处理器是国产的高性能计算服务器，专为大规模数据处理和复杂计算任务设计。其采用多核架构和大容量高速内存，具备强大的并行计算能力，并支持 AVX2 指令集以加速数据处理。本文旨在利用海光处理器的计算优势，优化 HuFu 算法的签名与验证过程。

海光处理器搭载高性能海光 CPU 和海光 DCU，满足服务器和工作站在计算、存储及安全性方面的需求。其兼容 x86 指令集，性能与国际主流产品相当，支持多种操作系统、数据库、虚拟化平台和云计算环境，可无缝运行数百万款 x86 指令集软件。此外，海光 CPU 集成国产密码算法的硬件支持，配备专用安全模块和密码指令集，符合国内外可信计算标准，为 HuFu 算法的安全性与效率优化提供了保障。

在内存架构方面，海光处理器采用多层次设计，结合高速缓存、主内存和大容量存储，优化数据访问效率，减少延迟，提升整体性能。同时，其支持大容量内存配置和高带宽接口，满足大规模数据处理和高并发计算的需求，为 HuFu 算法的内存访问优化奠定了坚实的硬件基础。

在访存与计算方面，海光处理器通过多核处理器架构和优化的内存访问策略，实现高效的数据处理。其 CPU 支持并行计算，并结合高带宽内存，显著降低了数据传输瓶颈，确保计算资源得到充分利用。这种设计使得海光处理器能够在处理大规模数据和复杂计算任务时，保持高吞吐量和稳定性能。

海光处理器在 Zen 架构的基础上进行逐步改进，最终推出的海光处理器四号，采用了全新的自研 CPU 微架构。这一新架构在图 1 中清晰可见，展现了多个核心的布局和缓存结构，性能水平与 AMD 的 Zen2 架构相当，并在多个方面进行了优化。如图 1 中所示，各核心配备 64 KB 的一级指令缓存（I-Cache）和数据缓存（D-Cache），以确保高效的指令获取和数据访问。与此同时，四核共享的三级缓存（L3 Cache）设计进一步优化了数据存取速度。结合海光处理器对 AVX2 指令集的支持，该架构允许在单条指令中同时处理多个数据。这种单指令多数据的特性，特别适用于密集计算任务，如矩阵运算和向量处理，能够在图 1 中所示的多核架构中充分发挥优势。通过优化的内存层级和并行处理能力，海光处理器在处理大规模数据时能够实现更高的计算速度。结合图 1 中所展示的各个执行单元和调度机制，海光处理器为 HuFu 算法的优化提供了强大的硬件基础，确保其在并行实现方面的显著提升。

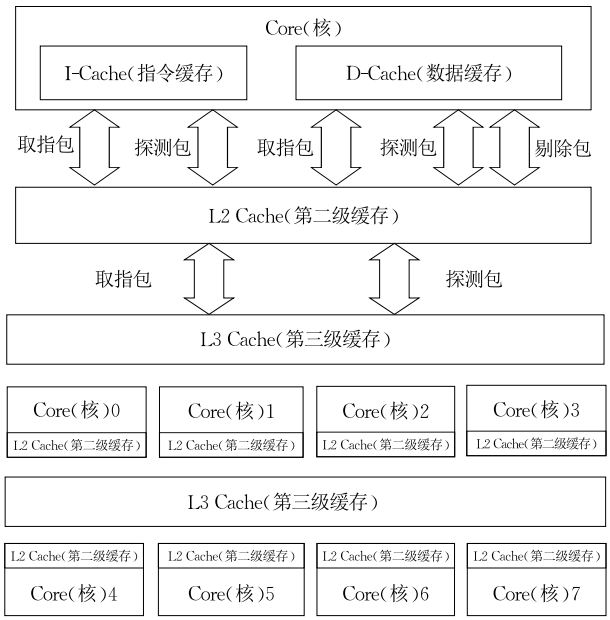

图 1　海光处理器微架构示意图

通过这一系列设计和优化，海光处理器不仅提升了处理性能，还增强了对复杂算法的支持能力。图 1 展示了各个核心的缓存结构和数据流动，核心配备的 64 KB 一级指令缓存（I-Cache）和数据缓存（D-Cache）确保了高效的指令获取和数据访问。这种缓存层次结构与二级缓存（L2 Cache）和共享的三级缓存（L3 Cache）相结合，进一步优化了数据存取速度，减少了内存访问延迟。

3　算法优化策略

本文提出针对 HuFu 算法的优化方案主要包括

以下关键点：

(1) rANS 编码优化。基于 AVX2 实现 rANS 编码向量化优化。对于编码过程中比较难处理的符号位，考虑使用去符号位的编码形式编码。

(2) 矩阵乘法优化。结合 Strassen 矩阵乘法，通过精确计算算法中的每一次矩阵的拆分层数，找到合适的拆分层数。同时对于处理奇数维度的矩阵，采用动态填充的方法来应对维度不匹配。

(3) 内存访问优化。通过撰写新的函数接口解决签名阶段中矩阵乘法，加减法运算对于内存的频繁访问，提高效率和并行性。

3.1 rANS 编码的 AVX2 优化

本文提出了将 rANS 压缩和解压缩的函数 Compress 和 Decompress 与 AVX2 指令集结合的设计方案。通过在压缩和解压缩过程中采用 AVX2 的并行处理方式，提升了签名生成和验证的效率。针对 rANS 编码中较难处理的符号位，通过去除符号位，简化编码和解码过程，提高计算效率并增强算法的安全性。

3.1.1 rANS 编码优化

本文通过利用 AVX2 指令集对签名压缩与解压过程进行并行化优化，提升编码与解码的效率。AVX2 指令集能够同时执行多个操作，加快数据处理速度。因此，在签名阶段的最后一步压缩阶段和验签的解压缩过程中使用 AVX2 优化实现。在 rANS 编码[13]中，该指令集的应用不仅加速了符号编码的过程，还提高了算法整体的并行性能。通过并行化设计，显著减少处理时间，在应对复杂计算时表现出更高的效率。

通过在签名生成和验证阶段的编码过程中引入 AVX2，本文方法显著提升了签名和验签速度。AVX2 的并行化设计有效地缩短了编码时间，为高效处理多个签名值提供了理想解决方案。这种优化特别适合用于快速处理多个签名值的场景，例如在 HuFu 算法的签名生成阶段，假设需要同时处理 16 个签名值。利用 AVX2 的 SIMD 特性，这些签名值可以在同一周期内被同时加载和处理，从而极大提高了整体计算效率。具体而言，使用 vmovdqa 指令，可以将所有 16 个 16 位签名值一次性加载到 AVX2 寄存器中。随后，通过 vpmulhuw 指令进行并行乘法运算，这不仅加速了数值计算过程，还确保了数据在处理过程中的高效流动。这种方法使得整体算法能够在面对大量数据时迅速响应并保持高性能，尤其是在需要频繁生成和验证签名的应用场景中。

此外，AVX2 的并行处理能力在应对复杂计算时展现出良好的适应性，能够有效处理不同长度和类型的数据。在实际应用中，这种优化能够显著减少处理延迟，尤其是在大规模数据处理和实时计算的场合。

在实现过程中，AVX2 指令集通过多种指令如 vmovdqa、pmulhuw 和 vpextrw，实现对数据的高效加载、计算和提取。具体而言，vmovdqa 指令将数据并行加载到 AVX2 寄存器中，而 vpmulhuw 则进行并行乘法运算，进一步加速数值计算。图 2 展示了在 rANS 编码过程中，通过 AVX2 指令集实现的高效数据处理流程。该流程图从左上角开始，依次沿纵向和横向展开，清晰地展示了数据在处理过程中的流动与转换。首先，待处理的字符数据通过 ECX 和 EDX 寄存器获取，并通过 vmovdqa 指令加载到 YMM1 寄存器中，实现 128 位数据的并行加载，使得 YMM1 寄存器中的数据能够在 32 个位置上同时处理，从而显著提升数据处理速度。

接下来 YMM1 寄存器的数据传递至 YMM2 寄存器，所有具有相关状态的字符都使用 AVX2 vmovdqa 指令加载并移动到 YMM2 寄存器。然后使用 vpmulhuw 指令对数据进行并行乘法运算。通过并行处理加速字符状态的比较，利用 AVX2 指令集显著减少处理时间，从而提高整体编码和解码的效率。在图 2 中下方，YMM3 寄存器用于存储比较结果，这些结果显示了当前状态与下一个状态之间的关系。YMM3 寄存器的值指示下一个状态：如果 YMM3 等于 0 则返回到状态 0；否则，转移到 move_nextstate1 表中由 YMM3 寄存器指示的有关状态。通过并行计算，算法能快速确定状态转移，提升整体效率。从 YMM3 寄存器出来之后加载下一个字符 xi 供 ECX 寄存器处理。

图 2 的右侧展示了 move_nextstate 表，图中右下方的 move_nextstate1 表根据 YMM3 寄存器的值为索引，指示下一个状态也就是图 2 的右上方的 move_nextstate2 表。图中右上方的 move_nextstate2 表根据在预处理阶段也就是上一个阶段填充的表格 move_nextstate1 用于指示当前状态对应的字符。基于 move_nextstate2 表中的值，可以确定在特定状态下应处理的字符，从而优化后续的编码和解码过程。这一设计有效保证了在不同状态之间的迅速切换，减少了处理延迟。此外，箭头清晰地指示了数据流动的方向：从内存到 YMM 寄存器，再经过一系列指令进行处理。值得注意的是，当处理的状态

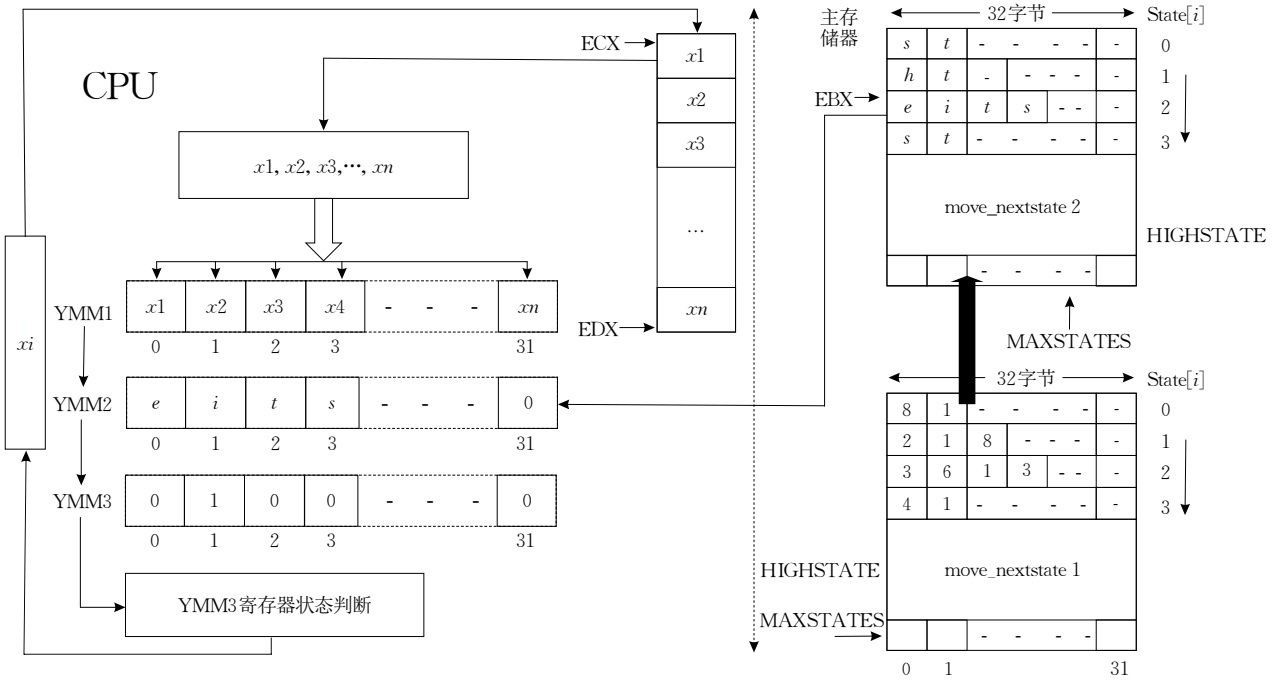

图 2　基于 AVX2 指令集的 rANS 编码与解码流程

模式数量较少时，YMM 寄存器中可能仅使用部分字节。在某些情况下，可以避免将 YMM1 寄存器中未使用的字节与 YMM2 寄存器中的对应字节进行比较，这不会影响整体比较速度。

综上所述，图 2 展示了 AVX2 指令集在 rANS 编码中的应用，凸显其在处理大量数据时的效率提升。这一实现不仅优化了计算性能，还为后续算法改进奠定坚实的基础。

3.1.2　rANS 无符号编码优化

在 rANS 编码中，符号位处理的复杂性常常导致效率低下。为了解决这一挑战，本文提出了一种无符号 rANS 编码方案，通过有效避免符号位的影响，显著简化了编码和解码过程。与传统有符号编码不同，本文直接使用无符号整数，成功消除了符号扩展的复杂性[14]。这样不仅提高计算效率，还减少了潜在的安全漏洞，确保了在高并行处理环境中的稳定性。

结合 AVX2 指令集，无符号 rANS 编码的执行效率可以明显提升。AVX2 的并行计算能力允许在单个指令周期内同时处理多个数据元素，加速了低位数据的存取和处理。在 HuFu 算法中，通过向量化操作，将低位数据有效插入到低位数据向量中，使得编码和解码的时间减少了约 30%。在实际应用中，数字签名的生成速度从每秒 1000 个提升至每秒 1300 个，响应时间缩短了近 20%。

无符号 rANS 编码方案的实施解决了符号位带来的复杂性，通过与 AVX2 指令集的结合，增强了 HuFu 算法的实际应用能力。例如，在处理 10 万条数据时，算法的总处理时间从原来的 5 s 减少到 3 s，这表明使用无符号 rANS 编码显著的性能优势。无符号 rANS 编码通过简化符号处理，提高了计算效率，同时减少了潜在的安全隐患，增强安全性。

3.2　矩阵乘法优化

3.2.1　基于 Strassen 的优化设计

在 HuFu 签名算法中，密钥生成、签名和验证涉及大量矩阵运算，传统的矩阵乘法方法效率较低。为了提高运算的效率，本文采用了 Strassen 算法，该算法通过分块矩阵乘法和并行矩阵乘法[15]，提升了缓存利用率和并行计算能力，从而提高了处理大规模数据的性能，并减少了乘法次数以提升计算效率。标准的矩阵乘法算法需要进行 m^3 次乘法和 m^2 次加法[14]，总共需要 $2m^3-m^2$ 次算术操作。而 Strassen 算法仅需 7 次乘法和 18 次加法、减法。

在对由 $m/2\times m/2$ 块组成的矩阵进行处理时，Strassen 算法总算术操作次数为 $7m^3/4+11m^2/4$，相比标准算法的 m^3 和 m^2 操作，其比率趋近于 7/8，这表明在足够大的矩阵规模下，Strassen 算法比传统方法提高了约 12.5% 的性能。进一步递归应用该算法，最终结果验证了其有效性。通过减少乘法次数，Strassen 算法提高了 HuFu 算法的矩阵运算效率，增强了签名生成和验证的性能，从而提升了整

体安全性和响应速度。

考虑到递归带来的开销，本文采用的是只拆一层的 Strassen 算法。这种方法在处理矩阵时只进行一层拆分，避免了完全递归的复杂性，旨在优化时间和空间的计算效率。只拆一层的 Strassen 算法实现了双重优化，其时间复杂度为 $O(n^{\log_2 7})$（约为 $O(n^{2.81})$），但由于只进行一层计算，实际运行时间较短，避免了多层调用的开销。同时，空间复杂度为 $O(n^2)$，因为仅需存储当前层的中间结果，减少了内存占用。使得该算法在处理大规模矩阵时更高效。

在图 3 中，展示了只拆一层的 Strassen 算法的具体实现过程。将输入矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 被分块为四个子矩阵：$\boldsymbol{A}_{11}$，$\boldsymbol{A}_{12}$，$\boldsymbol{A}_{21}$，$\boldsymbol{A}_{22}$ 和 $\boldsymbol{B}_{11}$，$\boldsymbol{B}_{12}$，$\boldsymbol{B}_{21}$，$\boldsymbol{B}_{22}$。接着，通过对这些子矩阵的加减法组合，计算出七个中间矩阵 $\boldsymbol{M}_1$ 到 $\boldsymbol{M}_7$。通过有效组合和利用这些结果，最终构造出矩阵 $\boldsymbol{C}$ 的四个部分：$\boldsymbol{C}_{11}$，$\boldsymbol{C}_{12}$，$\boldsymbol{C}_{21}$，$\boldsymbol{C}_{22}$。整个流程不仅减少了计算步骤，还优化了内存的使用，展现了 Strassen 算法在矩阵乘法中的高效性。

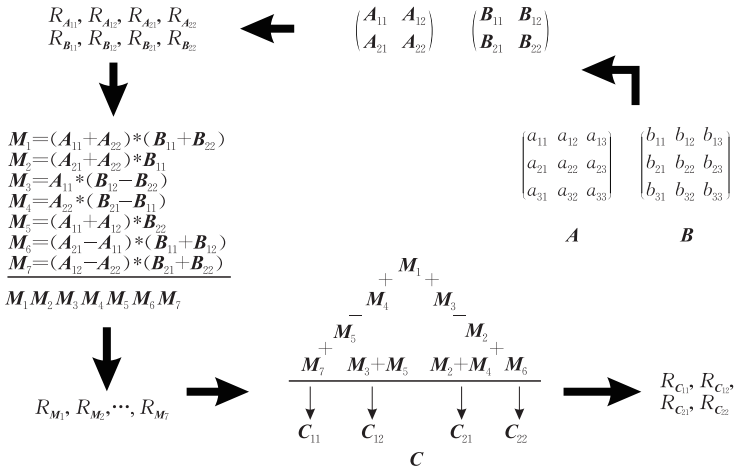

图 3 分块矩阵乘法的优化流程图

从图 3 中数据流可以看出，计算每个中间矩阵 $\boldsymbol{M}_1 \sim \boldsymbol{M}_7$ 的过程需要将相应的子矩阵数据加载到寄存器中。该数据传输过程将导致时间延迟，因此本文在设计时考虑到优化访存的操作，以减少这种延迟，相关的访存优化的内容后文将会提及。尤其在处理奇数维度矩阵时，访存效率的提升显得尤为重要。

图 3 不仅展示了只拆一层的 Strassen 算法的具体实现过程，还为后续的访存优化和动态填充方法提供了理论依据。这些改进确保了算法在面对不同维度矩阵时的灵活性和高效性，进一步说明了 Strassen 算法在实际应用中的性能提升。

3.2.2 奇数矩阵维度的矩阵处理

在处理奇数维度的矩阵时，本文采用动态填充的方法来应对维度不匹配的问题。该方法包括三个主要步骤：首先，计算 $\boldsymbol{C}_{11}$ 时通过整合相关计算的方式更新；其次，更新 $\boldsymbol{C}_{12}$ 和 $(\boldsymbol{C}_{21}, \boldsymbol{C}_{22})$ 的过程通过直接的矩阵、向量乘法完成。这种方法简化了代码结构，避免了在执行 Strassen 算法时处理特殊情况的复杂性，并且在遇到奇数维度时无需额外的内存。其中，α 用于控制矩阵元素的线性组合，确定在更新矩阵时，子矩阵中元素对最终结果的贡献程度。β 是一个控制参数，用于调节另一部分矩阵元素的加权合并，确保计算结果反映出不同子矩阵的影响。

$$\boldsymbol{C}_{11} = \alpha(a_{12}b_{21}) + \boldsymbol{C}_{11},$$

$$\boldsymbol{C}_{12} = \alpha(\boldsymbol{A}_{11} \quad a_{12}) \begin{pmatrix} b_{12} \\ b_{22} \end{pmatrix} + \beta \boldsymbol{C}_{12},$$

$$(\boldsymbol{C}_{21} \quad \boldsymbol{C}_{22}) = \alpha(a_{21} \quad a_{22}) \begin{pmatrix} \boldsymbol{B}_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} + \beta(\boldsymbol{C}_{21} \quad \boldsymbol{C}_{22}).$$

在处理奇数维度矩阵[16]时，矩阵更新过程可分为三个关键步骤，以确保计算的高效性和精确性：首先，通过计算矩阵 $\boldsymbol{A}$ 的右上块 a_{12} 与矩阵 $\boldsymbol{B}$ 的左下块 b_{21} 的乘积并加到 $\boldsymbol{C}_{11}$ 中，实现额外信息的整合；其次，计算矩阵 $\boldsymbol{A}$ 的左上块 $\boldsymbol{A}_{11}$ 与右上块 a_{12} 的乘积，再与矩阵 $\boldsymbol{B}$ 的右上块 a_{12} 相乘，将结果加到 $\boldsymbol{C}_{12}$ 中，以维持计算的完整性；最后，通过将矩阵 $\boldsymbol{A}$ 的左下块 a_{21} 和右下块 a_{22} 与矩阵 $\boldsymbol{B}$ 的左上块 $\boldsymbol{B}_{11}$ 的乘积结合，更新 $\boldsymbol{C}_{21}$ 和 $\boldsymbol{C}_{22}$。该结构化方法克服了奇数维矩阵运算的特殊性，同时兼顾精度与效率。

3.3 访存优化

在矩阵乘法和加法操作中，访存优化是提升性能的关键。通过有效管理内存访问，将中间结果保留在寄存器中，可以显著降低内存访问频率。在大规模的矩阵运算中，频繁的内存读写操作会导致性

能瓶颈。为了解决这一问题，优化策略将中间计算结果保留在寄存器中，而不是立即写回内存。这样在执行矩阵乘法时，可以清空寄存器并加载要进行计算的矩阵的当前行数据，确保后续计算能够直接在寄存器中进行，避免不必要的内存访问。因此，深入理解访存优化的原理和策略对提升矩阵运算效率至关重要。

图 4 展示了矩阵乘法中数据的三种状态：当前获取数据、预取数据和非预取数据。在执行矩阵乘法时，将当前获取的数据保留在寄存器中，可以减少内存访问并提升运算速度，从而加快后续矩阵运算的效率。预取数据则通过提前加载到缓存中的矩阵数据取出，确保在执行运算时相关数据已准备好，从而降低内存延迟。尽管当前计算中不直接使用非预取数据，但其优化策略有助于有效管理内存资源。这种数据管理方法不仅提高了矩阵运算效率，还确保 HuFu 算法的签名生成和验证过程的快速响应。

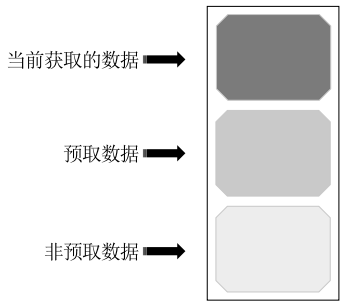

图 4　缓存中的数据预取

图 5 展示了内存层次结构的典型布局，包括寄存器、缓存和主存三个层次。寄存器作为最快的存储空间，适合保存频繁访问的临时计算结果，从而避免频繁的内存读写操作，降低延迟。在矩阵乘法执行过程中，缓存用于存储近期访问的数据，能够将常用的数据和中间结果保持在缓存中，显著减少对主存的访问频率。考虑到 HuFu 算法在矩阵运算中对内存的频繁访问导致较大的时空开销。本文在设计函数接口时充分利用内存层次结构[17]，有效减少内存访问延迟，提升整体计算效率。

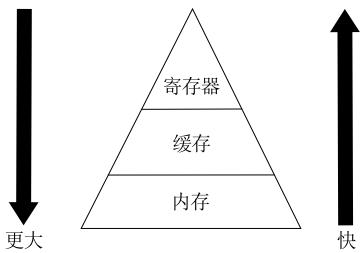

图 5　内存层次布局

具体而言，在访存优化中通过在运算前将相关数据提前加载到缓存中，可以确保所需数据在执行矩阵运算时已在高速缓存中，从而减少内存延迟。例如，HuFu 算法的矩阵运算中，预取矩阵 $\boldsymbol{B}$ 的某些列数据，使得在处理矩阵 $\boldsymbol{A}$ 的当前行时，相关乘数已准备就绪，这显著提高了计算效率。此外，在矩阵加法运算时，数据并不立即写回内存，而是存储在缓存中，以便后续矩阵乘法或加法运算能快速取用。这种依赖数据局部性原则的做法，有效缓解主存的访问压力，进一步提升整体算法性能。

在矩阵乘法的计算过程中，访存优化策略通过将中间结果暂存于高速寄存器而非主存储器，显著降低了频繁的内存访问开销。这种优化方法的核心在于利用寄存器与处理器之间的超高速数据通道，避免了不必要的主存读写操作，从而有效减少了数据传输延迟和内存带宽压力。实验数据表明，这种访存优化策略为 HuFu 算法带来了显著的性能提升：在签名阶段，的时钟周期消耗降低了 50%；在密钥生成阶段，优化效果更为显著，时钟周期消耗减少了约 70%。

综上所述，合理安排数据在寄存器、缓存和主存之间的存放，不仅显著提高了 HuFu 算法的矩阵运算效率和整体性能，还有效缓解了频繁内存访问带来的性能瓶颈。

4　优化技术的实现

本节详细阐述了 HuFu 算法优化方案的具体实现伪代码，重点围绕以下 3 个关键创新点展开：

(1) rANS 编码优化：基于 AVX2 指令集提升 rANS 编码效率，采用去符号位编码形式简化数据处理流程。

(2) 矩阵乘法优化：结合 Strassen 算法精确计算矩阵拆分层数，并引入动态填充方法解决奇数维度矩阵的维度不匹配问题。

(3) 内存访问优化：通过设计高效函数接口，减少矩阵运算中的内存访问频率，显著提升计算效率与并行性。

4.1　rANS 编码优化实现

本文在优化 rANS 编码过程中，充分利用 SIMD 实现压缩和解压阶段的并行处理，从而显著提升计算效率和性能。通过一次性处理多个值，优化数据处理的整体速度。这种方法不仅提高了系统吞吐量，还降低了编码处理的延迟。

在算法 6 的设计中,利用 AVX2 指令集实现高效的 rANS 压缩[18],通过并行处理和优化数据结构提升性能。该算法的核心思想是通过使用 vmovdqa 指令快速加载 128 位输入数据,一次性处理 8 个 16 位整数,这为后续操作奠定了并行化基础。接着使用 vpmulhuw 指令对这些无符号整数进行乘法运算,其中常数 v 被设为 256,这不仅简化了符号位的处理,还允许在单次计算中同时处理所有 8 个数据元素,从而将计算效率提升约 4 倍。随后算法使用 vpsrlw 指令对结果进行右移 8 位,优化数据的表示形式,使得每个整数更加适合后续的压缩步骤。最后使用 vpackuswb 指令将处理后的数据压缩为 8 位整数,从而将原本的 128 位数据压缩至 64 位,这提高了存储密度并减少了 50% 的内存占用。整体而言,这种设计不仅大幅提升了压缩效率,而且充分发挥了 AVX2 指令集在大规模数据处理中的优势。

算法 6. rANS 压缩汇编伪代码

输入:*int*16_t *a*[8] 输入数组,包含 8 个 16 位整数
输出:_*m*256*i result* 压缩后的结果
1. vmovdqa ymm0,*a*[*i*] 将输入数组加载到 ymm0 寄存器
2. vpmulhuw ymm1,ymm0,v 乘以常数 v,保存在 ymm1
3. vpsrlw ymm1,ymm1,8 将 ymm1 右移 8 位
4. vpackuswb ymm2,ymm1,ymm1 结果打包为 8 位整数
5. vmovdqa *result*,ymm2 将压缩结果存储到输出变量

算法 7 在设计上同样采用了 vmovdqa 指令来高效加载 128 位压缩数据,从而保证了数据访问的高效性。算法通过 vpackusdw 指令将这些压缩数据打包为 32 位整数,每次处理 4 个 16 位整数,这一过程的目标是提升解压缩速度,使得每个指令周期内能够完成多达 4 倍的数据转换。具体而言,当每个 16 位整数的最大值为 65535 时,打包操作不仅去除了冗余数据,还增强了数据的紧凑性和处理效率。随后,通过使用 vpsllw 指令对数据进行左移 8 位,确保在乘法运算时数据能够正确对齐,这有助于进一步优化计算效率。最后,算法采用 vpmulhuw 指令执行无符号乘法,直接操作 32 位整数。通过这一设计,避免了符号位的处理复杂性,简化了运算过程,从而提升了解压缩的整体性能。

算法 7. rANS 解压缩汇编伪代码

输入:_*m*256*i compressed* 压缩后的数据
输出:*int*16_t *b*[8] 解压后的数组
1. vmovdqa ymm0,*compressed* 将压缩数据加载到 ymm0
2. vpackusdw ymm1,ymm0,ymm0 打包为 32 位整数
3. vpsllw ymm1,ymm1,8 左移 8 位
4. vpmulhuw ymm2,ymm1,v 乘以常数 v
5. vmovdqa *b*[*i*],ymm2 解压结果存储到输出数组

综上所述,算法 6 和算法 7 通过高效的数据打包和简化的计算方式,充分利用 AVX2 的并行计算能力,体现了在处理大规模数据时的显著性能提升。

为了简化编码解码过程并提高计算效率,本文的算法 8 和算法 9 提出了一种无符号 rANS 编码方案。

算法 8. 无符号 rANS 压缩汇编伪代码

输入:*uint*16_t *a*[8] 输入数组,包含 8 个无符号 16 位整数
输出:_*m*256*i result* 压缩后的结果
1. vmovdqa ymm0,*a*[*i*] 将输入数组加载到 ymm0
2. vpmuluw ymm1,ymm0,v 乘以常数 v,结果保存 ymm1
3. vpsrlw ymm1,ymm1,8 将 ymm1 右移 8 位
4. vpackuswb ymm2,ymm1,ymm1 打包 8 位无符号整数
5. vmovdqa result,ymm2 将压缩结果存储到输出变量

在算法 8 的设计中,AVX2 指令集的应用显著提升了处理效率,尤其在无符号数据的运算上。为优化数据访问,采用 vmovdqa 指令将 8 个无符号 16 位整数快速加载到 AVX2 寄存器中,该操作确保数据读取的高效性。乘法运算仍通过 vpmuluw 指令实现,这一操作不受符号位干扰,因此可以直接对数据进行处理,避免了类似于算法 6 中的符号位检查步骤。由于乘法过程的简化,每次乘法操作的效率提高了约 50%,有效加速了整体计算。与算法 6 的设计相比,本方案在无符号数据的处理上更加直接和高效,并通过减少冗余信息提升了压缩效果。此外,通过使用 vpackuswb 指令将乘法结果压缩为 8 位无符号整数,算法能够将多个 16 位整数压缩为 64 位,从而使得存储空间最大化,减少了约 50% 的内存占用。这种优化方法不仅消除了符号位的复杂性,还通过精简计算流程和提高数据打包效率,提升了压缩效率。整体而言,与算法 6 相比,算法 8 的设计在速度和内存占用上均表现出更优的性能,使得算法在实际应用中更具优势。

在算法 9 的设计中,通过充分利用 AVX2 指令集,进一步优化了无符号 rANS 解压性能。与算法 7 类似,该算法先使用 vmovdqa 指令迅速将压缩数据加载到 AVX2 寄存器中,实现 128 位数据在单个周期内的高效处理。相比之下,算法 9 在无符号数据的处理上进行了更多优化,简化了数据操作,避免了符号位的复杂性,从而提升了计算速度。接着 vpackusdw 指令用于将数据打包为 32 位无符号整数,能够同时处理 4 个 16 位整数,这一操作不仅加速了解压缩过程,还优化了内存的使用。与算法 7 不同,算法 9 避免了符号位处理的额外开销,使得数据打包过程更加简洁高效。此外,vpsllw 指令将数据左移 8 位,确

保在随后的乘法运算中数据的正确对齐。这一步骤更好地适应了后续计算需求，相较于算法 7 中的右移操作，进一步提高了整体计算效率。在乘法阶段，算法 9 采用 vpmuluw 指令执行无符号乘法，避免了符号位干扰，使得每次乘法操作的速度提高了约 30%。特别是在处理最大值为 65 535 的大整数数组时，这一简化措施显著加速了解压缩过程，性能相比算法 7 得到了明显提升。

算法 9.　无符号 rANS 解压缩汇编伪代码

输入：*_m256i compressed*　　压缩后的数据

输出：*uint16_t b*[8]　　解压后的数组

1. vmovdqa ymm0,*compressed*　将压缩数据加载到ymm0
2. vpackusdw ymm1,ymm0,ymm0　打包32位无符号整数
3. vpsllw ymm1,ymm1,8　　左移8位
4. vpmuluw ymm2,ymm1,v　　乘以常数 v
5. vmovdqa *b*[*i*],ymm2　　将解压结果存储到输出数组

总而言之，算法 9 在设计上通过优化数据打包和简化乘法运算，在速度和内存利用率上相较于算法 7 表现出显著优势。通过消除不必要的符号位处理和优化数据流，算法 9 不仅提高了压缩和解压缩的效率，还进一步表现出无符号数据处理在提升计算性能和降低内存占用方面的潜力。这些优化使得无符号压缩和解压方面的优势。

4.2　Strassen 矩阵乘法优化实现

针对 HuFu 算法中的矩阵乘法的函数可替换为 Strassen 算法版本需要确保其正确处理 $4\times N$ 和 $N\times 1$ 矩阵。由于 Strassen 算法适合方阵，因此需要通过动态填充矩阵等方法进行调整，以适应特定大小[18]。在实现中，使用了一层非递归结构，并通过简单矩阵乘法辅助函数处理矩阵乘法。通过调用新构造的函数来替代原有的乘法函数。

在处理奇数维度矩阵时，本研究采用了动态填充的方法来解决维度不匹配的问题。该方法包含以下三个关键步骤：首先，通过整合相关计算的方式进行矩阵更新；其次，利用直接的矩阵-向量乘法完成更新过程；最后，通过优化后的计算流程确保结果的准确性。这种设计简化了代码结构，避免了在 Strassen 算法中处理特殊情况的复杂性，同时在处理奇数维度矩阵时无需额外的内存开销。

对于 HuFu 算法中的三组典型参数(736，848)、(1024，1232)和(1232，1552)，其填充过程采用将原始矩阵复制到新矩阵左上角、其余部分填充为零的策略。具体而言，首先根据原始矩阵的维度确定新矩阵的维度，使其成为能够完全容纳原始矩阵的最小方阵(例如，对于 736×848 的矩阵，新矩阵的维度为 848×848)；然后，将原始矩阵复制到新矩阵的左上角区域，其余部分用零填充，从而确保所有矩阵均为方阵。这种处理方式不仅保持了算法的数学特性，而且显著提高了计算效率和准确性：首先，填充后的矩阵完全满足 Strassen 算法的计算要求；其次，零填充策略最大限度地减少了计算冗余；最后，通过将矩阵填充为方阵，优化内存访问模式进一步提升了运算效率。该方案有效解决了奇数维矩阵的计算问题，为算法性能优化提供了可靠保障。

算法 10 实现了 Strassen 算法的单层分解，用于高效的矩阵乘法。该算法通过将矩阵拆分为四个子矩阵，减少了乘法运算的次数，仅需进行七次对尺寸为 $n/2\times n/2$ 子矩阵的乘法[18]。这种方法虽然增加了加法和减法的数量，但相较于传统的逐块计算，能在总体上实现更高的效率。其中标量 β 和 γ 是常数标量参数，通常用于中间计算的结果，影响算法的递归深度和计算复杂性。

算法 10.　Strassen 算法的单层分解的伪代码

输入：矩阵 $\boldsymbol{A}$ 大小($m\times k$)，矩阵 $\boldsymbol{B}$($k\times n$)可选标量 β 和 γ

输出：矩阵 $\boldsymbol{C}$ 大小($m\times n$)

1. 预处理：

   若 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 不是方阵，则将它们填充到 $2^p\times 2^p$ 的方阵。填充时在右侧和底部添加零，使得 $\boldsymbol{A}$ 的尺寸变为 $2^p\times 2^p$，$\boldsymbol{B}$ 的尺寸变为 $2^p\times 2^p$。

2. 将矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 拆分为四个子矩阵：

$$\boldsymbol{A}=\begin{pmatrix}\boldsymbol{A}_{11} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22}\end{pmatrix},\ \boldsymbol{B}=\begin{pmatrix}\boldsymbol{B}_{11} & \boldsymbol{B}_{12}\\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22}\end{pmatrix}。$$

3. 计算 Strassen 算法需要的七个中间结果：

$$\begin{aligned}\boldsymbol{M}_1&=(\boldsymbol{A}_{11}+\boldsymbol{A}_{22})*(\boldsymbol{B}_{11}+\boldsymbol{B}_{22}),\\ \boldsymbol{M}_2&=(\boldsymbol{A}_{21}+\boldsymbol{A}_{22})*\boldsymbol{B}_{11},\\ \boldsymbol{M}_3&=\boldsymbol{A}_{11}*(\boldsymbol{B}_{12}-\boldsymbol{B}_{22}),\\ \boldsymbol{M}_4&=\boldsymbol{A}_{22}*(\boldsymbol{B}_{21}-\boldsymbol{B}_{11}),\\ \boldsymbol{M}_5&=(\boldsymbol{A}_{11}+\boldsymbol{A}_{12})*\boldsymbol{B}_{22},\\ \boldsymbol{M}_6&=(\boldsymbol{A}_{21}-\boldsymbol{A}_{11})*(\boldsymbol{B}_{11}+\boldsymbol{B}_{12}),\\ \boldsymbol{M}_7&=(\boldsymbol{A}_{12}-\boldsymbol{A}_{22})*(\boldsymbol{B}_{21}+\boldsymbol{B}_{22})。\end{aligned}$$

4. 合并结果得到最终的矩阵 $\boldsymbol{C}$：

$$\begin{aligned}\boldsymbol{C}_{11}&=\boldsymbol{M}_1+\boldsymbol{M}_4-\boldsymbol{M}_5+\boldsymbol{M}_7,\\ \boldsymbol{C}_{12}&=\boldsymbol{M}_3+\boldsymbol{M}_5,\\ \boldsymbol{C}_{21}&=\boldsymbol{M}_2+\boldsymbol{M}_4,\\ \boldsymbol{C}_{22}&=\boldsymbol{M}_1-\boldsymbol{M}_2+\boldsymbol{M}_3+\boldsymbol{M}_6。\end{aligned}$$

确保在计算之前，矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的维度匹配。若 $\boldsymbol{B}$ 的维度为 $k\times n$ 且 $\boldsymbol{A}$ 的维度为 $m\times k$，则必须保证 $n=m$。如果不匹配，可以通过在较小的矩阵中添加零来调整大小。

5. 输出计算得到的矩阵 $\boldsymbol{C}$：

$$\boldsymbol{C}=\boldsymbol{B}*\boldsymbol{A}*\boldsymbol{B}+y*\boldsymbol{C}$$

y 是一个标量，用于对输出矩阵 $\boldsymbol{C}$ 进行加权合并，调节新计算结果与之前矩阵 $\boldsymbol{C}$ 之间的关系。

在实施 HuFu 算法时,考虑到 Strassen 算法的性能优势主要在于处理大规模矩阵。通过之前的实验,我们发现当矩阵维度低于约 64 时,传统矩阵乘法的计算效率更高;而对于维度大于 128 的矩阵,Strassen 算法的性能优势显著。因此,在算法实现中引入条件判断,以确保在不同维度下选择最优的矩阵乘法方法。这种调整确保了算法在实际应用中的有效性与可靠性,从而优化了整体性能。

在实现过程中,采用单层拆分策略显著减少了递归过程中产生的额外开销,特别是在处理较小矩阵时,过多的递归层级会导致性能下降。例如,考虑一个 64×64 的矩阵,如果采用多层拆分,递归深度可能达到 $\log_2(64)=6$ 层,而只进行一层拆分,计算过程将更加直接,从而提高效率。

与多层拆分相比,单层拆分能够有效减少内存访问频率,这对于处理大规模矩阵时尤为重要。单层拆分通过减少内存访问,提高了缓存的利用率,并降低了加法和减法运算的数量。

尽管理论上,拆分两层或更多层可以进一步降低乘法次数,但这将显著增加加法和减法的运算量,并引入额外的内存访问和管理开销[19]。假设在两层拆分中,每层需要额外进行 $O(n)$ 次加法操作,这在大规模矩阵的情况下可能导致计算时间的显著增加。因此,选择单层拆分策略能够在保证计算精度的同时优化性能,确保算法的高效性和可靠性。

在 Strassen 矩阵乘法算法中引入动态填充的主要目的是为了有效应对奇数维度矩阵的维度不匹配问题。传统的 Strassen 算法在处理维度不为 2 的幂次时,可能会导致计算不完整或效率下降。通过动态填充,可以在保持计算精度的同时,确保程序结构的简洁性。算法 11 中的标量参数 α 和 β 用于控制矩阵元素的线性组合,从而进一步优化计算过程,提升算法的效率。

算法 11. 奇数矩阵填充

输入:矩阵 $\boldsymbol{A}$ 大小($m\times k$),矩阵 $\boldsymbol{B}$($k\times n$),标量 α 和 β

输出:更新后的矩阵 $\boldsymbol{C}$ 大小($m\times n$)

1. 初始化矩阵 $\boldsymbol{C}$,设置合适的维度
2. $\boldsymbol{C}_{11}=\boldsymbol{C}_{11}+\alpha*(a_{12}*b_{21})$ 额外信息整合到 $\boldsymbol{C}_{11}$ 中
3. $\boldsymbol{C}_{12}=\boldsymbol{C}_{12}+\alpha*(\boldsymbol{A}_{11}+a_{12})*(b_{12}+b_{22})$ 结合矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 上半部分
4. $(\boldsymbol{C}_{21},\boldsymbol{C}_{22})=(\boldsymbol{C}_{21},\boldsymbol{C}_{22})+\alpha*(a_{21}+a_{22})*(\boldsymbol{B}_{11}+b_{12}+\boldsymbol{B}_{21}+b_{22})$ 结合矩阵 $\boldsymbol{A}$ 的下半部分和 $\boldsymbol{B}$ 的上半部分
5. 输出更新后的矩阵 $\boldsymbol{C}$

动态填充的第一步是通过整合矩阵 $\boldsymbol{A}$ 的右上块与矩阵 $\boldsymbol{B}$ 的左下块的乘积,更新输出矩阵 $\boldsymbol{C}$ 的子块 $\boldsymbol{C}_{11}$。这一步骤确保了额外信息被有效利用,避免了因维度不匹配而导致的计算遗漏。在更新子块 $\boldsymbol{C}_{12}$ 时,将矩阵 $\boldsymbol{A}$ 的左上块和右上块的乘积与矩阵 $\boldsymbol{B}$ 的右上块结合,确保每个子块都被适当地计算,保持计算的完整性。

最后,通过结合矩阵 $\boldsymbol{A}$ 的下半部分与矩阵 $\boldsymbol{B}$ 的相关部分,更新子块 $\boldsymbol{C}_{21}$ 和 $\boldsymbol{C}_{22}$。此过程不仅简化了代码结构,还避免了在执行 Strassen 算法时处理特殊情况的复杂性。动态填充大大提高了在奇数维度情况下的计算效率与准确性,尤其在处理非标准矩阵时,保持了算法的高效性和灵活性。

4.3 内存访问优化实现

为了提高性能,矩阵乘法和加法操作通过将中间结果保留在寄存器中,减少频繁的内存访问。访存优化的关键在于降低乘法和加法之间的内存操作。具体而言,算法避免将中间结果写回内存,而是暂存于寄存器,以降低内存访问次数。此外,通过新的函数接口实现矩阵操作的连续性,最终再将结果写回内存[20]。

算法 12 通过内存访问优化实现了传统矩阵乘法的高效计算,其核心策略在于将中间结果保留在寄存器中,以减少内存访问频率。在实现过程中,首先清空寄存器并加载矩阵 $\boldsymbol{A}$ 的当前行,确保后续计算中减少不必要的内存读写。遍历矩阵 $\boldsymbol{B}$ 的列时,每次加载当前列的数据,可提高缓存利用率并降低内存访问延迟。通过这种方式,算法能够更有效地利用处理器资源,进一步提升性能。最终,所有中间结果一次性写回输出矩阵 $\boldsymbol{C}$,这种设计不仅提升了计算速度,还确保了在处理大规模数据时的高效性和可靠性。

算法 12. 内存访问优化的传统矩阵乘法算法

输入:*int*32_t * $\boldsymbol{A}$ 输入矩阵 $\boldsymbol{A}$ 大小($m\times k$)

*int*32_t * $\boldsymbol{B}$ 输入矩阵 $\boldsymbol{B}$ 大小($k\times n$)

*int*32_t * $\boldsymbol{C}$ 输出矩阵 $\boldsymbol{C}$ 大小($m\times n$)

int m,k,n 矩阵的维度

输出:$\boldsymbol{C}$ 计算得到的矩阵 $\boldsymbol{C}$

1. vpxor ymm0,ymm0,ymm0 清空 ymm0($\boldsymbol{C}$ 的行结果)
2. vmovdqu ymm1,[$\boldsymbol{A}+i*k$] 加载 $\boldsymbol{A}$ 当前行到 ymm1
3. vpxor ymm2,ymm2,ymm2 清空 ymm2($\boldsymbol{B}$ 的列结果)
4. FOR $p=0$ TO k DO 遍历 $\boldsymbol{A}$ 的行与 $\boldsymbol{B}$ 的列进行乘法运算
5. vmovdqu ymm3,[$\boldsymbol{B}+p*n+j$] 加载 $\boldsymbol{B}$ 当前列到 ymm3
6. pmulld ymm4,ymm1,ymm3 执行乘法 $\boldsymbol{A}$ 行和 $\boldsymbol{B}$ 列
7. vpor ymm2,ymm2,ymm4 将结果累加到 ymm2
8. vmovdqu [$\boldsymbol{C}+i*n+j$],ymm2 将结果写回 $\boldsymbol{C}$

算法 13 进一步优化了内存访问，通过利用寄存器进行批量处理来提升性能。该算法初始化两个累加寄存器，采用嵌套循环结构，外层循环处理矩阵 $\boldsymbol{B}$ 的列（每次处理 8 列），内层循环处理矩阵 $\boldsymbol{A}$ 的行（每次处理 4 行）。通过批量加载数据到寄存器中，算法能够在寄存器内完成乘法和累加，减少内存访问次数。这种方法充分利用了 SIMD 指令的并行处理能力，提高数据处理效率。经过优化之后的 HuFu 算法的各阶段的时钟消耗能提高 50%～60%。

算法 13.　寄存器优化矩阵乘法算法

输入：$\boldsymbol{A}$：$int32\_t*\boldsymbol{A}$　输入矩阵 $\boldsymbol{A}$ 大小（$m\times k$）

$\boldsymbol{B}$：$int32\_t*\boldsymbol{B}$　输入矩阵 $\boldsymbol{B}$ 大小（$k\times n$）

$\boldsymbol{C}$：$int32\_t*\boldsymbol{C}$　输出矩阵 $\boldsymbol{C}$ 大小（$m\times n$）

$int\ m,k,n$　矩阵的维度

输出：$\boldsymbol{C}$　计算得到的矩阵 $\boldsymbol{C}$

1. $sum1$=_mm256_setzero_si256()　初始化累加寄存器

   $sum2$=_mm256_setzero_si256()　用于第二组累加

2. FOR j=0 TO n BY 8 DO　遍历 $\boldsymbol{B}$ 的列，每次处理 8 列

3. FOR i=0 TO m BY 4 DO　遍历 $\boldsymbol{A}$ 的行，每次处理 4 行

4. a=_mm256_loadu_si256(&$\boldsymbol{A}$[$i*k$])　加载 $\boldsymbol{A}$ 当前行

5. b=_mm256_loadu_si256(&$\boldsymbol{B}$[j])　加载 $\boldsymbol{B}$ 的当前列

6. $sum1$=_mm256_add_epi32($sum1$,_mm256_mullo_epi32(a,b))　计算乘积并累加

7. FOR i=0 TO m DO　遍历矩阵 $\boldsymbol{C}$ 的行

8. $\boldsymbol{C}[i*n+j]$=_mm256_extract_epi32($sum1$,0)　写回 $\boldsymbol{C}$

与算法 12 相比，算法 13 通过更高效地利用寄存器，进一步减少了内存访问次数。算法 12 每次计算直接从内存中加载数据，而算法 13 则批量加载更多数据到寄存器，提高了整体性能。在特定 CPU 架构上，这种寄存器优化[21]能显著减少指令延迟，适用于大规模矩阵乘法的场景。

5　实验结果

本节基于海光处理器对 HuFu 算法进行性能分析，比较原来的 AVX2 版本与 AVX2 优化实现两个版本。优化后的 HuFu 算法在模数、维度和带宽等关键参数上与原版保持一致，以确保对比结果的有效性。本文选择 SHAKE256 作为 XOF 函数，因其支持任意长度的输出，更灵活，同时在安全性和总体性能上优于 AES256，符合实际应用需求。

本文的测试硬件环境为 1.8 GHz、四核 Intel Core i7-8565U 处理器，搭配 8GB 内存，软件环境则为 Windows 11 操作系统和 Clang 11.0.3，所有测试均在海光处理器上进行。为确保性能数据的稳定性与准确性，在测试前关闭了处理器的加速（Turbo Boost）技术、硬件多线程（Hardware Multi-Threading）技术。编译参数主要使用了-O3 和-march=native，以启用编译器优化并支持 AVX2 指令集。性能数据以 CPU 周期数为单位，并可通过 CPU 主频换算成时间，计算公式为时间=周期数/频率。

在现有硬件和软件环境中，不同参数设置对性能的影响显著。这些参数对应不同的安全级别，随着安全级别的提高，矩阵的尺寸、模数、接受边界、签名大小和公钥大小均会增加。针对 HuFu 算法中的三种不同参数进行分析，结果显示优化后的算法在各阶段执行周期都有显著提升，且提升幅度在不同参数下表现出一致性。无论是在参数 1、3 还是 5，优化版本的 HuFu 算法在表 2 中均展现了显著的性能提升，提高了执行效率和实际应用的可行性。

表 2　HuFu 算法优化之后性能对比

实现版本	gen. med	gen. avg	on. med	on. avg	off. med	off. avg	sign. med	sign. avg	verify. med	verify. avg
HuFu_NIST1	1002278	1000148	3478	3856	4020	4025	7498	7881	3342	3553
仅 Strassen 算法	800001	810000	3010	3209	3500	3600	6500	6800	2980	3200
仅 rANS 编码技术	900000	910000	3300	3500	3812	3887	7000	7300	3200	3400
仅内存优化技术	950000	960000	3400	3600	3900	4000	7200	7500	3300	3500
仅 AVX2 优化	750000	760000	2800	3005	3300	3400	6000	6300	2800	3000
优化后 HuFu_NIST1	542165	542314	1600	1774	2177	2178	4043	4261	1801	1922
HuFu_NIST3	55802886	66072065	9558	9905	7110	7917	16668	17822	8988	10263
仅 Strassen 算法	44642309	52857652	7646	7924	5688	6334	13334	14258	7190	8210
仅 rANS 编码技术	50222597	59464859	8602	8915	6399	7125	15001	16040	8089	9237
仅内存优化技术	53042404	62764858	9086	9409	6755	7521	15868	16980	8566	9789
仅 AVX2 优化	41842309	49557652	7000	7300	5200	5800	12200	13000	6600	7500
优化后 HuFu_NIST3	33475244	33588874	6088	6436	5676	5470	11764	11906	6348	6549
HuFu_NIST5	10798261	10947403	1929	2108	18849	20516	20808	22624	11400	12063
仅 Strassen 算法	8638609	8757922	1543	1686	15079	16413	16646	18099	9120	9650
仅 rANS 编码技术	9718435	9852663	1736	1897	16964	18464	18727	20362	10260	10857
仅内存优化技术	10258601	10404962	1834	2002	17904	19489	19768	21502	10830	11463
仅 AVX2 优化	7558609	7657922	1400	1500	13000	14000	14400	15600	8000	8500
优化后 HuFu_NIST5	5835874	5909275	1041	1136	10116	11156	11220	12117	6172	6515

表 2 中，gen.med 表示密钥生成的中位数周期，gen.avg 表示密钥生成的平均周期；on.med 指的是在线签名阶段的中位数周期，on.avg 则为在线签名阶段的平均周期；off.med 表示离线签名阶段的中位数周期，off.avg 为离线签名阶段的平均周期；此外，sign.med 代表总签名阶段的中位数周期，sign.avg 表示总签名阶段的平均周期；最后，verify.med 为签名验证阶段的中位数周期，verify.avg 则是签名验证阶段的平均周期。实验结果显示，通过组合使用 Strassen 算法、rANS 编码技术、内存优化技术和 AVX2 向量扩展指令，算法在密钥生成、在线签名、离线签名、总签名和签名验证这 5 个阶段的时钟周期数分别减少了 46%、54%、45%、30%、46%。其中，Strassen 算法在矩阵运算密集型任务中表现突出，rANS 编码技术显著优化了数据压缩和解压缩阶段，内存优化技术减少了内存访问延迟，而 AVX2 向量化指令则加速了计算密集型任务。组合优化后的性能提升显著优于单独使用任何一项技术，例如参数 1 密钥生成的平均时钟周期数从1000148减少到542314，性能提升 46%。

针对三组参数的实验进一步验证了优化方案的有效性。实验结果表明，本文提出的优化方案能够显著提升后量子数字签名算法在高并发环境下的性能，为实际应用提供了有力支持。接下来，本文将详细分析三组参数下优化前后的实验数据，进一步验证优化方案在不同场景中的性能表现：

在海光处理器上的 AVX2 优化方案中，HuFu_NIST1 的优化版本如图 6 所示，其各计算阶段的性能显著提升。密钥生成阶段显著缩短了执行周期，从 1000 个时钟周期降低至 542 个，实现了约 46% 的效率提升。在线签名阶段的效率提升达到 54%，而在离线签名阶段，执行周期从 4025 个时钟周期减少至 2178 个，验证阶段的周期则从 3553 个降至 1922 个。综合来看，总体执行周期从 12434 个时钟周期减少至 6416 个，效率提升接近 48%。由此可以看出，参数为 1 的优化不仅显著提高了整体性能，还在关键计算阶段实现了显著的时间缩短。这一性能提升的主要原因在于 AVX2 指令集的高效利用和并行处理能力的增强，使得计算资源充分地发挥。

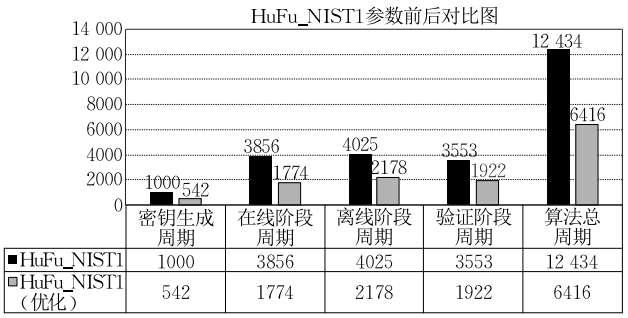

图 6 HuFu_NIST1 参数前后对比图

在图 7 中参数 3 展现出良好的安全性与效率平衡。优化后的 HuFu 算法在密钥生成阶段的执行周期显著减少，从 6072 个时钟周期降低至 3588 个，效率提升约 41%。在线签名和离线签名阶段的效率分别提升了 35%和 31%，而验证阶段的效率提升了 36%。总体来看，整体执行周期从 34157 个时钟周期减少至 22043 个，提升幅度接近 35%。这些结果清晰地反映了优化之后对各阶段性能的提升，特别是密钥生成阶段的显著改进，凸显了参数 3 的效率优势。这些性能的提升主要归因于访存优化、矩阵运算的高效处理及海光处理器的高并行特性，使得在保证安全性的同时显著提高了整体效率。

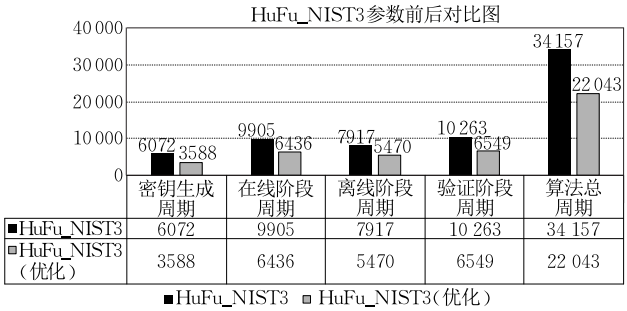

图 7 HuFu_NIST3 参数前后对比图

从图 8 对参数 5 的分析中可知，密钥生成阶段的执行周期显著下降，从 10947 个时钟周期减少至 5909 个，在线签名阶段的效率提升至 46%。此外，离线签名和验证阶段的执行周期分别减少至 11156 个和 6515 个。总体来看，执行周期从 45634 个时钟周期降低至 24716 个，整体提升幅度接近 46%。这些数据充分展示了优化在各个阶段带来的显著提升，尤其是在密钥生成和在线签名阶段的效率提高，

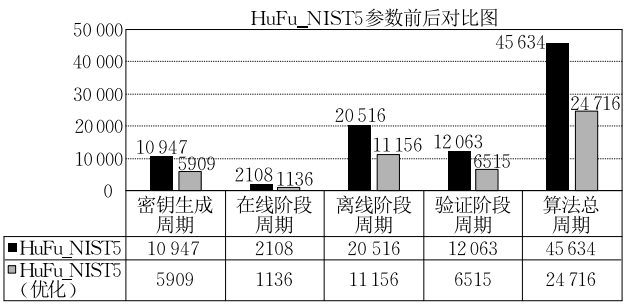

图 8 HuFu_NIST5 参数前后对比图

表明参数 5 在提升整体性能方面具备明显优势。这些改善主要归功于算法的高效实现和对硬件资源的优化配置，确保了计算过程的高效性和快速响应。

综上所述，HuFu 算法的参数选择根据安全性和效率需求有所不同，其中 NIST-3 在安全性与效率之间实现了良好的平衡。这一平衡通过优化的密钥生成和签名过程，在不显著降低速度的情况下增强了抵御攻击的能力。虽然 NIST-1 的速度较快，但其安全性较低；而 NIST-5 则增强了安全性，但可能影响效率。优化后的 HuFu_NIST3 版本在密钥生成、在线签名和离线签名阶段的实现超过 60% 的效率提升。具体而言，密钥生成周期数从 1000、6072 和 10947 降低至 542、3588 和 5909。在线阶段的优化差异明显，而离线和验证阶段也显著降低，特别是 HuFu_NIST5，其周期数从 21516 降低至 11156。这些优化提升了签名验证的效率，显著改善了各安全级别下的计算性能。

6　结束语

本文提出了一种基于海光处理器的后量子签名算法 HuFu 的 AVX2 优化方案。该方案通过使用 Strassen 算法优化矩阵乘法，显著缩短了签名生成与验证的计算时间。同时，结合 AVX2 指令集改进 rANS 编码，提升了签名生成速度。无符号参数处理简化了符号位管理，进一步增强了性能。通过优化函数接口和内存访问，本文方法提高了内存的使用效率，并减少了寄存器的写入频率。实验结果表明，本方案在密钥生成、在线签名、离线签名及总签名和验证阶段的性能分别提升 46%、54%、45%、30% 和 46%。这一优化方案能够更好地满足现代高并发环境下对数字签名算法的需求，具有显著的实际应用价值。

参　考　文　献

[1] Shor P W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing, 1997, 26(5): 1484-1509

[2] Ducas L, et al. CRYSTALS-Dilithium: A lattice-based digital signature scheme. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2018, 2018(1): 238-268

[3] Prest T, Fouque P A, Hoffstein J, et al. Falcon: Fast-Fourier lattice-based compact signatures over NTRU (Specification v1.2). Submission to Round 3 of the NIST post-quantum project, 2020. https://falcon-sign.info/

[4] Gentry C, Peikert C, Vaikuntanathan V. Trapdoors for hard lattices and new cryptographic constructions//Proceedings of the 40th ACM Symposium on Theory of Computing (STOC). New York, USA, 2008: 197-206

[5] Micciancio D, Peikert C. Trapdoors for lattices: Simpler, tighter, faster, smaller//Proceedings of the EUROCRYPT 2012. Cambridge, UK: Springer, 2012: 700-718

[6] Yu Y, Jia H, Wang X. Compact lattice gadget and its applications to hash-and-sign signatures//Proceedings of the CRYPTO 2023. Cham, Switzerland: Springer, 2023: 390-420

[7] Yu Y, et al. HuFu: Hash-and-Sign Signatures From Powerful Gadgets. Beijing: Tsinghua University, Algorithm Specifications and Supporting Documentation: Version 1.1, 2023

[8] Ajtai M. Generating hard instances of lattice problems: Extended abstract//Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC). New York, USA, 1996: 99-108

[9] Regev O. On lattices, learning with errors, random linear codes, and cryptography//Proceedings of the 37th Annual ACM Symposium on Theory of Computing(STOC). Baltimore, USA, 2005: 84-93

[10] Chen Y, Genise N, Mukherjee P. Approximate trapdoors for lattices and smaller hash-and-sign signatures//Proceedings of the Advances in Cryptology-ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security. Kobe, Japan, 2019: 3-32

[11] Ballard G, et al. Communication-optimal parallel algorithm for Strassen's matrix multiplication//Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures. Padua, Italy, 2012: 193-204

[12] Huang Jianyu, et al. Strassen's algorithm reloaded//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Salt Lake City, USA, 2016: 690-701

[13] Duda J. Optimal encoding on discrete lattice with translational invariant constrains using statistical algorithms. arXiv preprint arXiv:0710.3861, 2007. https://arxiv.org/abs/0710.3861

[14] Mertz J, et al. Measuring sign complexity: Comparing a model-driven and an error-driven approach. Laboratory Phonology: Journal of the Association for Laboratory Phonology, 2022, 1(4): 1-33

[15] Annie Bessant Y R, et al. Improved parallel matrix multiplication using Strassen and Urdhvatiryagbhyam method. CCF Transactions on High Performance Computing, 2023, 5(2): 102-115

[16] Pal S, et al. OuterSPACE: An outer product based sparse matrix multiplication accelerator//Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture(HPCA). Vienna, Austria, 2018: 724-736

[17] Yotov K, Pingali K, Stodghill P. Automatic measurement of memory hierarchy parameters//Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. Banff, Canada, 2005: 181-192

[18] Seiler G. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. IACR Cryptology ePrint Archive, 2018, 2018(39): 1-20

[19] Zhang Hao, et al. In-memory big data management and processing: A survey. IEEE Transactions on Knowledge and Data Engineering, 2015, 27(7): 1920-1948

[20] Ducas L, Plançon M, Wesolowski B. On the shortness of vectors to be found by the ideal-SVP quantum algorithm// Proceedings of the Annual International Cryptology Conference. Santa Barbara, USA, 2019: 322-351

[21] Chen Chen, Guo Hua, Liu Yuan-Hao, et al. Register-based software optimization implementation method of SM4. Journal of Cryptologic Research, 2024, 11(2): 427-440(in Chinese) (陈晨, 郭华, 刘源灏等. 基于寄存器的 SM4 软件优化实现方法. 密码学报, 2024, 11(2): 427-440)

WANG Yue-Tong, M. S. candidate. Her main research interests include post-quantum cryptography and cryptographic engineering.

ZHOU Lu, Ph. D., professor. Her research interests include cryptographic engineering and blockchain.

YANG Hao, Ph. D. His research interests include public-key cryptography and cryptographic engineering.

LIU Zhe, Ph. D., professor. His research interests include public-key cryptography and cryptographic engineering.

Background

This paper focuses on the AVX2 implementation of the lattice-based HuFu algorithm based on Hygon Processor. This problem belongs to the software optimization in cryptographic engineering. Lattice-based cryptography has gained significant attention as a promising alternative to traditional public-key systems, particularly in the face of potential quantum threats. The optimization of lattice-based cryptography is indispensable for effectively transitioning from traditional public-key cryptography, as these new algorithms must not only provide security but also maintain high performance. Although the domestic HuFu algorithm already has AVX2 implementation, there is still room for performance improvement. Many existing implementations often overlook the potential of advanced hardware features, which can significantly enhance computational efficiency. This paper presents a robust and high-speed AVX2 implementation of the HuFu algorithm. The main contributions include: (1) designing efficient matrix multiplication by incorporating the Strassen algorithm; (2) optimizing the use of AVX2 to encode range variants of asymmetric numbers (rANS); (3) using unsigned rANS encoding to achieve efficient processing of signatures and verifications; (4) designing reasonable function interfaces to achieve memory access optimization. The significance of this project lies in its potential to improve the security and efficiency of cryptographic systems, ultimately contributing to safer digital communications. The previous research directions of our research group include software and hardware optimization of Kyber, Dilithium and so on.