软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(7):3013−3032 [doi: 10.13328/j.cnki.jos.007559] [CSTR: 32375.14.jos.007559]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于 CUDA Core 和 Tensor Core 的 CTRU-Prime 高吞吐量实现*

胡晓雯[1], 邹恒川[1], 沈诗羽[2], 李文倩[1], 赵运磊[1,3]

[1](复旦大学 计算与智能创新学院, 上海 200433)

[2](香港城市大学 电气工程系, 香港 999077)

[3](密码科学技术国家重点实验室, 北京 100878)

通信作者: 赵运磊, E-mail: ylzhao@fudan.edu.cn

摘　要: 量子计算机的迅猛发展对现存密码体制造成了极大的威胁, 后量子密码算法的实现和迁移部署尤为重要. 其中, 基于 NTRU 格的密码方案因结构简洁、计算效率高等优点而备受瞩目. CTRU-Prime 方案基于 NTRU 格构造, 鉴于其在安全性、带宽和实现效率上的出色表现和 GPU 在大规模并行处理任务上的强大能力, 在 Tensor Core 和 CUDA (compute unified device architecture) Core 的基础上给出了 CTRU-Prime 的首个高吞吐量实现. CTRU-Prime 的底层代数结构为素阶数域, 在抵御针对分圆环攻击的同时, 也为多项式乘法的实现带来挑战. 首先, 提出两种素阶数域上多项式乘法的 GPU 实现方案. 基于 CUDA Core 的伪梅森数不完整 *NTT* 的多项式乘法使用层融合技术优化访存模式, 能够达到 256.98 倍吞吐量, 基于 Tensor Core 的教科书式多项式乘法, 将多项式乘法转化为矩阵操作, 利用低精度 MMA (matrix-multiply-and-accumulate) 操作实现, 能够达到 177.24 倍吞吐量. 接着, 结合批量模式和单一模式、多流技术和多线程技术, 给出了 GPU 平台上面向吞吐量的 CTRU-Prime 总体架构, 使用融合内核、合并全局内存访问、优化访存模式等优化策略, 加快各个核函数的访存和计算速度. 实验结果表明, 基于 RTX 3060 平台, CTRU-Prime-653、CTRU-Prime-761、CTRU-Prime-1277 每秒可以分别进行密钥生成 6.3 万、5.4 万、1.6 万次, 密钥封装 63.5 万、274.5 万、160.1 万次, 密钥解封装 35.1 万、262.2 万、152.4 万次, 是 C 实现版密钥生成吞吐量的 68.85、79.78、66.84 倍, 密钥封装吞吐量的 10.32、46.57、46.81 倍, 密钥解封装吞吐量的 11.43、89.19、90.32 倍. 与最新实现的 Kyber 相比, 密钥封装吞吐量达到 1.46 倍, 密钥解封装达到 1.74 倍, 是其他 NTRU 格基 GPU 高吞吐量实现的 26 倍.

关键词: 后量子密码; NTRU 格基密码; 密钥封装机制; 并行处理; 图形处理器

中图法分类号: TP309

中文引用格式: 胡晓雯, 邹恒川, 沈诗羽, 李文倩, 赵运磊. 基于CUDA Core和Tensor Core的CTRU-Prime高吞吐量实现. 软件学报, 2026, 37(7): 3013–3032. http://www.jos.org.cn/1000-9825/7559.htm

英文引用格式: Hu XW, Zou HC, Shen SY, Li WQ, Zhao YL. CTRU-Prime High-throughput Implementation Based on CUDA Core and Tensor Core. Ruan Jian Xue Bao/Journal of Software, 2026, 37(7): 3013–3032 (in Chinese). http://www.jos.org.cn/1000-9825/7559.htm

CTRU-Prime High-throughput Implementation Based on CUDA Core and Tensor Core

HU Xiao-Wen[1], ZOU Heng-Chuan[1], SHEN Shi-Yu[2], LI Wen-Qian[1], ZHAO Yun-Lei[1,3]

[1](College of Computer Science and Artificial Intelligence, Fudan University, Shanghai 200433, China)

[2](Department of Electrical Engineering, City University of Hong Kong, Hong Kong 999077, China)

[3](State Key Laboratory of Cryptology, Beijing 100878, China)

* 基金项目: 上海市协同创新基金 (XTCX-KJ-2023-54); 上海市科委区块链关键技术攻关专项基金 (23511100300)
收稿时间: 2024-11-26; 修改时间: 2025-03-17; 采用时间: 2025-09-26; jos 在线出版时间: 2026-01-14
CNKI 网络首发时间: 2026-01-15

Abstract: The rapid development of quantum computers poses significant threats to existing cryptographic systems. The implementation and migration of post-quantum cryptographic algorithms are therefore of utmost importance. Among these, NTRU lattice-based cryptographic schemes have gained attention due to their simplicity and computational efficiency. The CTRU-Prime scheme, based on NTRU lattices, stands out for its excellent performance in security, bandwidth, and implementation efficiency. Given the powerful capabilities of GPUs in handling large-scale parallel processing tasks, this study presents the first high-throughput implementation of CTRU-Prime using Tensor Core and compute unified device architecture (CUDA) Core. The underlying algebraic structure of CTRU-Prime is large-Galois-group prime-degree prime-ideal number field (LPPNF), which not only resists attacks targeting cyclotomic rings but also presents challenges for the implementation of polynomial multiplication. First, two GPU implementations of polynomial multiplication over LPPNF are proposed. The CUDA Core-based Pseudo-Mersenne incomplete *NTT* polynomial multiplication uses layer fusion techniques to optimize memory access patterns, achieving a throughput of 256.98 times. The Tensor Core-based schoolbook polynomial multiplication converts polynomial multiplication into matrix operations, leveraging low-precision matrix-multiply-and-accumulate (MMA) operations, achieving a throughput of 177.24 times. Next, an overall architecture for CTRU-Prime on the GPU platform is presented, focusing on throughput. This architecture combines batch mode and single mode, multi-stream technology, and multi-thread techniques. Optimization strategies such as fused kernels, coalesced global memory access, and optimized memory access patterns are employed to accelerate memory access and computation speeds of various kernel functions. Experimental results show that, on the RTX 3060 platform, CTRU-Prime-653, CTRU-Prime-761, and CTRU-Prime-1277 can perform key generation at rates of 63 000, 54 000, and 16 000 times per second, respectively; key encapsulation at rates of 635 000, 2 745 000, and 1 601 000 times per second, respectively; and key decapsulation at rates of 351 000, 2 622 000, and 1 524 000 times per second, respectively. These rates are 68.85, 79.78, and 66.84 times higher for key generation, 10.32, 46.57, and 46.81 times higher for key encapsulation, and 11.43, 89.19, and 90.32 times higher for key decapsulation compared to the C implementation. Compared to the latest Kyber implementation, the key encapsulation throughput is 1.46 times higher, and the key decapsulation throughput is 1.74 times higher, making it 26 times more efficient than other high-throughput NTRU lattice-based GPU implementations.

Key words: post-quantum cryptography; NTRU Lattice-based cryptography; key encapsulation mechanism (KEM); parallel processing; graphics processing unit (GPU)

随着量子技术的突飞猛进, 传统的公钥密码系统正面临重大威胁. 量子敌手能够利用 Shor 算法[1]在多项式时间破解当前公钥密码体制所基于的大整数分解和离散对数等经典困难问题. 因此, 研究能够抵抗量子攻击的新型公钥密码算法——后量子密码, 得到了学术界和工业界的广泛关注.

当前, 出现了许多后量子密钥封装方案 (key encapsulation mechanism, KEM). 美国国家标准与技术研究院 (National Institute of Standards and Technology, NIST) 在向社会各界发出的新密码体制征求提案中, 就包括 KEM. 在征集的方案中, 基于 NTRU 格的密码体制备受瞩目. Hoffstein 等人[2]于 1998 年正式提出了 NTRU 加密方案, 如今 NTRU 已成为各种密码学原语 (如提议标准化的数字签名方案 Falcon) 的基础组成部分. 基于分圆环的 CNTR[3]首次将高维 NTRU 格密码与底层格编码联系起来, 它使用单个密文多项式来压缩密文, 并针对 NTRU 型 KEM 在安全性、带宽、错误率和实现效率方面实现了均衡的性能. 此外, 基于 NTRU 格的方案早已走上标准化和商业化的道路. 早在 2009 年, IEEE 标准 1363.1 就包括了基于格的密码方案, 包括 NTRUEncrypt[4]. 2011 年, X9.98 标准采用了 NTRUEncrypt 用于金融服务[5]. 2016 年, Schanck 等人[6]尝试通过使用 NTRUEncrypt 生成额外的临时密钥与临时 ECDH 密钥一起使用, 以增强 Tor 协议在量子时代的前向保密性. 国际标准 OpenSSH 在 2022 年 4 月发布的 9.0 版[7]及以后版本中, 采用了 NTRU-Prime 与 X25519 ECDH 相结合的混合模式, 以抵御“解密后捕获”攻击. 然而, 大多数现有的基于格的密码方案, 包括 NTRU, 都是基于分圆环构造的. 如文献 [8–10] 中所强调的, 分圆环的丰富代数结构使这些方案容易受到攻击. Bernstein 等人[11]提出了一个具有“高安全性、素数阶、大 Galois 群和惰性模数”的基础代数结构, 本文将这类大 Galois 群、素数阶、基于素理想的数域 (large-Galois-group prime-degree prime-ideal number field, LPPNF) 简称为素阶数域. Bernstein 等人[11]还建议将现有的基础代数结构从分圆环迁移到素阶数域, 以抵御已知和潜在的针对分圆环的攻击, 从而提供更保守的安全性方案. 基于素阶数域设计各种基于格的密码方案是格密码学中安全、可靠、高效且实用的路线. 因此, NTRU-Prime 方案在这种背景下被提出[11], 并且其

SNTRU-Prime-761 参数已经在 OpenSSH 中默认应用, 成为 KEM 事实标准.

CTRU-Prime[12]在 CNTR 的基础上使用 LPPNF 代数结构, 在安全性、带宽、实现效率方面比 NTRU-Prime 更有优势, 其目前已在国家密码标准化委员会作为制定类标准进行了立项. 然而, 像 CTRU-Prime 这样基于格的方案存在高计算、内存开销大以及 IO 传输大的问题, 在查询量巨大的服务器端场景中, 这会导致其成为性能瓶颈. 此外, 同基于非素阶数域的代数结构相比, CTRU-Prime 提供了更高的安全性, 但由于素阶数域不具备 *NTT* 友好性其代码实现更为复杂.

此外, 图形处理单元 (graphics processing unit, GPU) 凭借其大规模并行计算能力和高吞吐量, 常被用于加速计算密集型任务[13]. 目前已经存在密钥封装、签名算法等基于 GPU 平台的优化实现, 例如, 文献 [14–16], 该类工作通过释放 GPU 的 CUDA (compute unified device architecture) Core 和 Tensor Core 的计算潜力以进一步提高格基密码的吞吐量. 目前, GPU 已广泛应用于需要后量子保护的云服务、物联网等场景, 因此基于 GPU 实现 CTRU-Prime 是高效性和适用性的综合考量.

本文提出 CTRU-Prime-653、CTRU-Prime-761、CTRU-Prime-1277 的 GPU 高吞吐量实现. 该方案综合考虑并行处理任务、高效访存、加速计算和优化算法等方面, 充分发挥 GPU 的计算能力, 加速算法的运行效率, 达到高吞吐量的目标. 本文的主要贡献有以下几点.

(1) 素阶数域由于非 *NTT* 友好性, 在 GPU 设计上存在挑战. 提出两种素阶数域上多项式乘法的 GPU 实现方案. 基于 CUDA Core 的伪梅森数不完整 *NTT* 的多项式乘法使用层融合技术优化访存模式, 在 n=653, 761, 1277 这 3 组参数下与 C 实现基线测试结果相比, 分别达到了 1.09–11.08 倍、2.02–256.13 倍和 2.86–256.98 倍的吞吐量. 基于 Tensor Core 的教科书式多项式乘法, 将多项式乘法转化为矩阵操作, 利用低精度 MMA (matrix-multiply-and-accumulate) 操作实现, 其中通用的 tensor 乘实现和面向相同公私钥的 tensor 乘实现分别达到了 1.19–2.04 倍和 10.36–177.24 倍的吞吐量.

(2) 结合批量模式、单一模式、多流技术和多线程技术, 给出了 GPU 平台上面向吞吐量的 CTRU-Prime 总体架构. 进一步地, 使用融合内核、合并全局内存访问、优化访存模式等优化策略, 加快各个核函数的访存和计算速度. 通过实验验证上述优化思路的合理性: 在不同批处理大小上, 融合内核策略有效减少了 31.25%–73.17% 的内核执行时间; 当并行度为 768, CUDA 流数量为 8 时, 使用多流技术能够减少 72.83% 的 CTRU-Prime-761 密钥解封装算法的执行时间.

(3) 实验结果表明, 基于 RTX 3060 平台, CTRU-Prime-653、CTRU-Prime-761、CTRU-Prime-1277 每秒可以分别进行密钥生成 6.3 万、5.4 万、1.6 万次, 密钥封装 63.5 万、274.5 万、160.1 万次, 密钥解封装 35.1 万、262.2 万、152.4 万次, 是 C 实现版密钥生成吞吐量的 68.85、79.78、66.84 倍, 密钥封装吞吐量的 10.32、46.57、46.81 倍, 密钥解封装吞吐量的 11.43、89.19、90.32 倍. 同最新实现的 Kyber 相比, 密钥封装吞吐量达到 1.46 倍, 密钥解封装达到 1.74 倍, 是其他 NTRU 格基 GPU 高吞吐实现的 26 倍.

1 相关工作

GPU 平台上的后量子密码加速可大致分为基于 CUDA Core 和基于 Tensor Core 两类.

基于 CUDA Core 设计并实现后量子密码是 GPU 后量子密码加速的主流方向. 例如, Gupta 等人[15]探索了后量子密钥交换算法的 GPU 实现, 提出了 FrodoKEM-976、NewHope-1024 和 Kyber-1024 的 GPU 实现, 并设计了简单模式和批处理模式以适应不同应用场景. Sun 等人[17]研究了基于多核平台的 SPHINCS 优化方法, 依据 SPHINCS 内部结构设计并行化版本, 并结合多种优化技术将其应用于 GPU 平台. Gao 等人[18]提出并比较了 NewHope 的基准实现、细粒度实现和多流实现, 探讨了不同优化策略在降低整体延迟和提高吞吐量方面的效果. Shen 等人[19]提出了第 1 个面向服务器的高吞吐量 ML-DSA 签名设计, 通过稀疏三元多项式乘法技术实现深度优先且提早拒绝的采样过程, 实现了显著的性能提升.

Tensor Core 主要用于 MMA 运算, 不少工作探讨了 Tensor Core 在后量子密码上的加速效果. Lee 等人[16]提出

使 Tensor Core 能够处理灵活的矩阵大小和临时密钥对的几种并行算法以加速格基密码的多项式卷积操作, 并将该技术应用于 NTRU. Wan 等人[20]探讨了 Tensor Core 在精度和性能之间的权衡, 提出了密码原语到 AI 加速器运算的转化框架和基于 Tensor Core 的 *NTT* 盒, 最终在 Kyber 上实施与评估. ConvKyber[21]进一步优化了 Kyber 基于 Tensor Core 的实现, 提出了两种将 Kyber *NTT* 转化为迭代矩阵乘法的创新方法, 使用汇编级代码精确操作 Tensor Core 内部资源. Hafeez 等人[22]引入单周期多公钥处理技术和乘法与约减分离技术, 利用 Tensor Core 加速多项式乘法以提高密钥封装的性能, 并将该技术应用于 Sable 和 Florete. Hafeez 等人[13]提出了一种并行 Toeplitz 矩阵向量积版本以加速 GPU 上实现的多项式乘法, 在 CUDA Core 和 Tensor Core 上进行比较, 方案的有效性在 Saber 和 Sable 上得到验证.

NTRU 格基密码算法的 GPU 加速研究发展如下. 2010 年, Kamal 等人[23]使用 GPU 加速具有良好数据并行特性的 NTRU-Encrypt. 2010 年, Hermans 等人[24]针对 CUDA 平台详细分析了确切的实现, 通过查看对分支、内存访问、块和线程的潜在影响来解释每个选择的影响. 2014 年, Bai 等人[25]针对 NTRU 的大数据问题, 提出了单 GPU 零拷贝、单 GPU 数据传输和多 GPU 版本这 3 种策略来分别实现设备、块和线程级别的数据并行, 并使用流优化技术来实现设备计算重叠. 2016 年, Dai 等人[14]利用 GPU 在大型向量处理上的优势对签名方案 NTRU-MLS 的底层操作算子进行研究并加速, 通过 GPU 同时生成大量候选来进一步提高拒绝采样的效率. 2018 年, Akleylek 等人[26]提出了 NTRU-encrypt 的高吞吐量实现, 其提出的 Karatsuba 多项式乘技巧与通用实现相比具有 20.6% 的提升, 该研究能够为 IoP 的两实体间通信提供有效安全保护.

2 基础知识

2.1 符号与定义

在本文中, $\mathbb{R}$ 表示实数集, $\mathbb{Z}$ 表示整数集, n,q 表示某些正整数, $\mathbb{Z}_q=\mathbb{Z}/q\mathbb{Z}\cong\{0,1,\ldots,q-1\}$ 表示模 q 剩余系. 算法实现中的多项式运算均在多项式环 $\mathcal{R}_q=\mathbb{Z}_q[x]/(x^n-x-1)$ 上进行, 其中的元素均为最高 $n-1$ 次多项式且系数都在 $\mathbb{Z}_q$ 中. 一般使用 $f=\sum_{i=0}^{n-1}f_ix^i$ 来表示这个多项式环中的元素, 其中 $f_i\in\mathbb{Z}_q$.

设 D 为一个集合, 则符号 $x\leftarrow D$ 表示从集合中均匀随机选取一个元素 x, 若 D 为一个概率分布, 则符号 $x\leftarrow D$ 表示根据这一概率分布选取元素 x. 以整数 η 为参数的中心二项分布 B_η 的定义为: 从 $\{0,1\}^{2\eta}$ 中均匀随机采样得到 $(a_1,a_2,\ldots,a_\eta,b_1,b_2,\ldots,b_\eta)$, 输出 $\sum_{i=1}^{\eta}(a_i-b_i)$. 而采样多项式 $f\leftarrow B_\eta$ 表示对每个系数进行采样.

2.2 数论变换

数论变换 (number theoretic transform, *NTT*)[6]是计算有限域上多项式乘法的高效实现, 它是快速傅里叶变换 (fast Fourier transform, FFT) 在有限域上的特殊版本. 考虑多项式环 $\mathcal{R}_q=\mathbb{Z}_q[x]/(x^n-1)$ 上的多项式乘法, 其中 q 为素数且满足 $n|q-1$, 那么可以在 $\mathbb{Z}_q$ 中找到 n 次本原单位根 ω. 设 f 为 $\mathcal{R}_q$ 上任意多项式, 则数论变换为 $NTT(f)=\hat{f}=(\hat{f}_0,\hat{f}_1,\ldots,\hat{f}_{n-1})$, 其中 $\hat{f}_i=\sum_{j=0}^{n-1}f_j\omega^{ij}\bmod q$, 这本质上是将多项式转化成点值表示, 从而将多项式乘法转化为向量对应位置元素的乘积, 本文使用“$\circ$”表示这一乘积. 则 *NTT* 将 $h=f\cdot g$ 转化为 $NTT(h)=NTT(f)\circ NTT(g)$. 数论变换的逆变换定义为 $f=INTT(\hat{f})$, 其中 $f_i=n^{-1}\sum_{j=0}^{n-1}\hat{f}_j\omega^{-ij}$, 注意 $f=INTT(NTT(f))$, 因此 $h=INTT(NTT(f)\circ NTT(g))$.

基于中国剩余定理 (Chinese remainder theorem, CRT), 可以利用 FFT-trick 进一步加速 *NTT*. 如果多项式 f和g互素, 则可以得到同构关系 $\mathbb{Z}_q[x]/(fg)\cong\mathbb{Z}_q[x]/(f)\times\mathbb{Z}_q[x]/(g)$, 这一结论可以推广至任意多个两两互素的多项式. 一般的基 *N-NTT* 对应的同构为: $\mathbb{Z}_q[x]/(x^{Nm}-\zeta^N)\cong\mathbb{Z}_q[x]/(x^m-\zeta)\times\mathbb{Z}_q[x]/(x^m-\rho\zeta)\times\ldots\times\mathbb{Z}_q[x]/(x^m-\rho^{N-1}\zeta)$, 其中 ρ 为 $\mathbb{Z}_q$ 上的 N 次本原单位根. 正向 FFT trick 使用 CT (Cooley-Tukey) 蝴蝶操作[6], 逆向 FFT trick 使用 GS (Gentleman-Sande) 蝴蝶操作[27].

2.3 CTRU-Prime 算法描述

由梁志闯等人[12]提出的 CTRU-Prime 是基于素阶数域和尺度化 E_8 格编码设计的 NTRU 格基密钥封装方案.

CTRU-Prime 基于 NTRU 假设和 RLWR 假设, 在安全性、带宽、实现效率方面和 SNTRU-Prime 相比具有优势, 表 1 给出了 CTRU-Prime 的 3 组参数集.

表 1 CTRU-Prime 参数集

n	q	q_2	n'	(Ψ_1,Ψ_2,Ψ_3)	$\lvert pk\rvert$	$\lvert ct\rvert$	$B.W.$	$NTRU(C, Q)$	$RLWR(C, Q)$	δ
653	4621	2^{11}	320	(B_3, B_3, B_3)	994	898	1892	(152, 137)	(151, 136)	2^{-166}
761	4591	2^{10}	376	(B_3, B_2, B_3)	1158	952	2110	(178, 161)	(181, 164)	2^{-170}
1277	7879	2^{10}	632	(B_2, B_2, B_2)	2067	1597	3664	(299, 271)	(297, 269)	2^{-279}

CTRU-Prime 的公钥加密方案如算法 1–3 所示, 密钥封装方案如算法 4–6 所示.

算法 1. CTRU-Prime.PKE.KeyGen.

输入: 安全参数 1^k;

输出: 公钥加密公私钥对 (pk, sk).

1. $g \leftarrow \Psi_1$, $f' \leftarrow \Psi_2$
2. $f = pf' + 1$
3. $h = g/f$
4. return $(pk = h, sk = f)$

算法 2. CTRU-Prime.PKE.Enc.

输入: 公钥 pk, 明文 m;

输出: 密文 c.

1. $r \leftarrow \Psi_3$
2. $\sigma = hr$
3. $c = \left\lfloor \frac{q_2}{q}(\sigma + PolyEncode(m)) \right\rceil \bmod q_2$
4. return c

算法 3. CTRU-Prime.PKE.Dec.

输入: 私钥 sk, 密文 c;

输出: 明文 m.

1. $m' = cf \bmod^{\pm} q_2$
2. $m = PolyDecode(m')$
3. return m

算法 4. CTRU-Prime.KEM.KeyGen.

输入: 安全参数 1^k;

输出: 密钥封装公私钥对 (pk', sk').

1. $(pk, sk) \leftarrow$ CTRU-Prime.PKE.KeyGen(1^k)
2. $z \leftarrow \{0,1\}^{\iota}$
3. return $(pk' = pk, sk' = (sk, z))$

算法 5. CTRU-Prime.KEM.Encaps.

输入: 公钥 pk';
输出: 密文 c, 共享密钥 K.

1. $m \leftarrow \mathcal{M}$
2. $(K, coin) = \mathcal{H}(ID(pk), m)$
3. c = CTRU-Prime.PKE.Enc$(pk, m; coin)$
4. return (c, K)

算法 6. CTRU-Prime.KEM.Decaps.

输入: 私钥 $sk' = (sk, z)$, 密文 c;
输出: 共享密钥 K.

1. m' = CTRU-Prime.PKE.Dec(sk, c)
2. $(K', coin') = \mathcal{H}(ID(pk), m')$
3. $\hat{K} = \mathcal{H}_1(ID(pk), z, c)$
4. if $m' \neq \perp$ and c = CTRU-Prime.PKE.Enc$(pk, m'; coin)$ then
5. return K'
6. else
7. return $\hat{K}$

2.4 GPU 简介

GPU 的线程组织形式是其强大计算能力的关键. 与 CPU 的少数几个复杂核心不同, GPU 拥有成千上万个较简单的核心, 这些核心可以并行执行大量的线程. 在 CUDA 中, 线程被组织成线程块 (block) 和线程网格 (grid). 一个线程块包含多个线程, 这些线程共享一个小的、快速的本地存储器 (称为共享内存), 可以高效地进行线程间的数据交换和协作. 而多个线程块则组成一个线程网格, 线程网格可以覆盖整个 GPU 的计算资源. 这样的组织方式允许 GPU 同时处理大量的数据块, 极大地提升了并行处理能力. 此外, GPU 还采用了一种称为 SIMT (single instruction multiple threads) 的执行模式, 即多个线程同时执行相同的指令, 但操作的数据不同, 这种模式进一步提高了计算效率. GPU 的内存结构也是其高效运算的重要因素之一. GPU 的内存层次结构包括全局内存、常量内存、纹理内存、共享内存和寄存器等. 全局内存是容量最大但访问速度较慢的内存, 所有线程都可以访问. 常量内存和纹理内存是只读的, 访问速度较快, 适合存储不变的数据. 共享内存则是一个线程块内的所有线程共享的内存, 具有非常快的访问速度, 非常适合需要频繁访问的数据. 寄存器是每个线程独享的高速存储器, 用于存储临时变量和数据. 通过这种层次化的内存结构, GPU 能够在不同的存储需求下优化数据访问效率, 从而提高整体计算性能. GPU 拥有多个流式处理器 (streaming multiprocessor, SM). 在 SM 上, 寄存器、共享内存等资源有限, 且规定了最大并行线程束数和最大并行线程块数. 基于此, 占用率 (occupancy) 是评估 SM 上活跃线程束比例的关键指标, 用于衡量线程对 GPU 资源的利用是否合理. 具体来说, 核函数的占用率指的是在任意给定周期内, 每个 SM 上活跃的线程束平均数量与该处理器支持的最大线程束数量的比率. 占用率分为理论占用率和实际占用率两种类型. 通常情况下, 实际占用率应尽可能接近理论占用率, 而理论占用率则是基于核函数理论上所需的资源进行计算得出的. 通过优化核函数设计以提高实际占用率, 可以更有效地利用 GPU 的并行处理能力.

3 素阶数域上多项式乘法的 GPU 设计与实现

多项式乘法是格基密码中最为耗时的操作之一, 一个通用的优化方式是使用 *NTT* 来加速这个过程, *NTT* 对于

密码方案的底层代数结构具有一定要求, 一般将支持 *NTT* 操作的环称为 *NTT* 友好环. CTRU-Prime 的底层代数结构为素阶数域 $\mathbb{Z}_q[x]/(x^n-x-1)$, 具备不可分解性, 不支持一般的 *NTT* 操作. 因此, 优化加速素阶数域上的多项式乘法存在一定的挑战. 本节首先基于 CUDA Core 设计了素阶数域拓展环上的伪梅森数不完整 *NTT*, 此外, 在完成素阶数域多项式乘法的矩阵数学推导的基础上, 利用 GPU Tensor Core 的矩阵计算优势实现教科书式的多项式乘法.

3.1 基于 CUDA Core 的伪梅森数不完整 *NTT* 的多项式乘法

3.1.1 伪梅森数不完整 *NTT* 简介

文献 [12] 使用伪梅森数不完整 *NTT* 完成素阶数域上的多项式乘法, 表 2 为 CTRU-Prime 的伪梅森数不完整 *NTT* 参数表. 具体来说, 基于域扩张技术, 将素阶数域 $\mathbb{Z}_q[x]/(x^n-x-1)$ 上的多项式乘法转移到 *NTT* 友好的拓展环 $\mathbb{Z}_{q'}[x]/(x^N-1)$ 上 (q' 为具有 2^x-2^y+1 形式的伪梅森数), 且拓展环上的多项式乘法结果仅需进行简单的多项式模运算即可回到原先的素阶数域上. 下面以 $\mathcal{R}_q$ 上的多项式乘法 $h=f\cdot g$ 为例, 介绍主体流程.

(1) 域扩张. 将 n 维多项式 f 和 g 的高次项补 0 扩充至 N 维, 得到多项式 $f',g'\in\mathbb{Z}_{q'}[x]/(x^N-1)$, 其中 $N\geqslant 2n$, q' 为某个足够大的素数使得它大于计算过程中多项式系数在 $\mathbb{Z}$ 上的最大值.

(2) 计算 f' 和 g' 的正向 *NTT* 变换、点乘和逆向 *NTT* 变换, 得到 $h'=f'\cdot g'\in\mathbb{Z}_{q'}[x]/(x^N-1)$. 需要注意的是, 由于拓展环不具备完全可分解性, 在计算 N 维正向 *NTT* 变换和逆向 *NTT* 变换时, 使用伪梅森数不完整 *NTT*.

(3) 域收缩. 计算 $h=(h' \bmod (x^n-x-1)) \bmod q$, 即为 f 和 q 在 $\mathcal{R}_q$ 中相乘的结果. 需要注意的是, 域收缩过程的正确性由 N,q' 的取值保证.

表 2 伪梅森数不完整 *NTT* 参数集

方案	参数 (n,q,q_2,η)	N,q' 的取值	FFT trick顺序
CTRU-Prime-653	$(653,4621,2^{11},3)$	$(1344,16777153=2^{24}-2^6+1)$	5层基2、1层基3、1层基2
CTRU-Prime-761	$(761,4591,2^{10},2)$	$(1536,33550337=2^{25}-2^{12}+1)$	9层基2
CTRU-Prime-1277	$(1277,7879,2^{10},2)$	$(1344,16777153=2^{25}-2^{12}+1)$	9层基2

3.1.2 基于 CUDA Core 的优化设计与实现

如上所述, 基于伪梅森数不完整 *NTT* 的多项式乘法主要包括 5 步: 域扩张、正向 *NTT* 变换、点乘、逆向 *NTT* 变换和域收缩, 需从内存模式和线程组织形式两个角度考虑, 设计融合 5 个操作的基于 CUDA Core 的多项式乘法方案.

从内存延迟角度来看, 全局内存的访问延迟最高, 共享内存次之, 寄存器最少. 基于上述访存特点, 本文设计了如图 1 所示的内存模式. 在该内存模式中, 仅有域扩张和域收缩需要和全局内存进行交互, 其余操作被转移到寄存器和共享内存上, 从而有效提高了访存的效率. 进一步地, 在域扩张和域收缩的过程中, 合并对全局内存的访问, 以提高 DRAM (dynamic random access memory) 的带宽利用率. 在执行正向和逆向 *NTT* 的过程中, 使用层融合技术[28], 如图 1 所示, 线程以特定步长读取共享内存上的数据到自身寄存器后, 无需再与共享内存交互, 直接基于寄存器完成多层 *NTT* 计算, 从而进一步减少访存带来的开销.

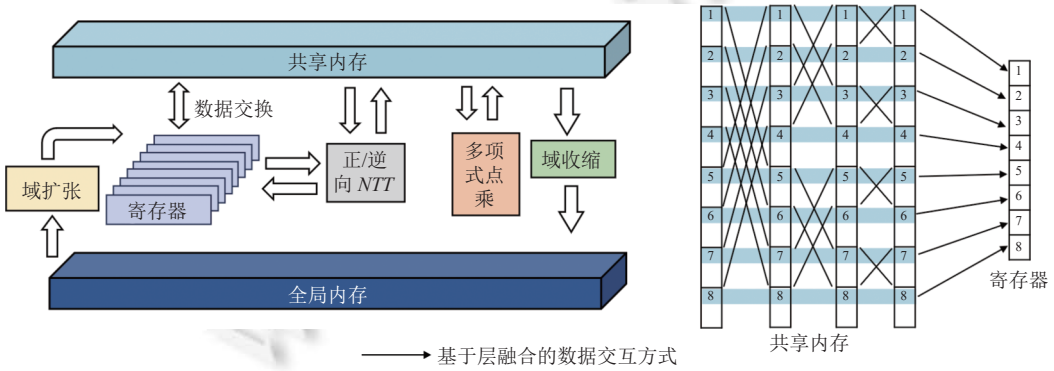

图 1 基于伪梅森数不完整 *NTT* 的素阶数域多项式乘法的内存模式

从线程组织形式的角度来看, 上述 5 个操作主要呈现 3 种并行关系: 第 1 种为 n 并行的域扩张和域收缩; 第 2 种为 $N/2$ 并行的正向和逆向 *NTT* 变换; 第 3 种为受拓展环 *NTT* 分解影响的多项式点乘. 由于并行关系的不同, 使用多个核函数是自然的想法, 但是这显然会带来和全局内存过于频繁交互的效率降低, 因此本文使用单一核函数实现上述 5 个操作. 鉴于上述 5 个操作中, 正向和逆向 *NTT* 变换占比最高且耗时最多, 线程组织形式的设计在优先减少上述两个操作分支执行的基础上, 力求尽可能多地满足更多操作的并行特性.

从伪梅森数不完整 *NTT* 的并行特性出发, 由于 CTRU-Prime 的 3 组参数的拓展环分解中, 都含 $3k$ ($k=2,3$) 个基 2-*NTT*, 因此本文中, 每个线程使用 8 个寄存器以完成层数为 3 的层融合, 这既是适应基 2-*NTT* 数量的考量, 也是对 SM 理论占用率的核函数线程数量和寄存器使用量的平衡. 具体来说, 每个线程块包含 $N/8$ 个线程, 每个线程拥有 8 个寄存器以存储不同位置的多项式系数. 为成功实现层融合, 需要正确设计在共享内存中的存取位置. 具体来说, 假如当前位于第 i 层, 则根据其使用旋转因子的异同, 线程被均匀地分到 2^i 组中, 即 $N/(8\times 2^i)$ 个连续线程的存取位置被限制在对应组内. 为实现层融合, 各线程以 $N/(8\times 2^i)$ 的间隔从组内存取数据. 上述方式将 3 次存取共享内存的操作减少为 1 次, 从而进一步提高了存储效率. 为在 CTRU-Prime-653 中应用层融合技术, 本文将伪梅森数不完整 *NTT* 的 FFT-trick 计算顺序调整为先计算 6 层基-2 FFT trick, 再计算 1 层基-3 FFT trick.

在上述线程组织形式下, 每个线程块完成一次多项式乘法, 使用大小为 3 个拓展环上多项式的共享内存空间. 对于除正向和逆向 *NTT* 以外的其他操作, $N/8$ 的线程束数量仅引起一次单个线程束的分支执行, 并未带来较大的性能惩罚. 除点乘外, 域扩张和域收缩过程的线程下标对齐是简单且自然的. 多项式点乘的实现方式为在 CRT 同构图的叶子节点上完成低维的教科书式多项式乘法, 其下标对齐需考虑旋转因子表的访问. 例如, 在 CTRU-Prime-653 中, 经过正向 *NTT* 后, 需在 192 个形如 $\mathbb{Z}_{16777153}[x]/(x^7-\zeta^{64k+br_6(j)+1})$ ($k=0,1,2;0\leqslant j\leqslant 63$) 的域上完成多项式点乘. CRT 同构树形图的第 i 个叶子节点对应旋转因子表中的第 $64+2\times i$ 个位置, 其对应的域为 $\mathbb{Z}_{16777153}[x]/(x^7-\omega\zeta^{64+2i})$, 其中 ω 为 3 次本原单位根. 本文使用 Karatsuba 技巧进一步加速点乘过程.

本文引入了循环展开技术, 以降低循环迭代次数、增强指令级并行性、提升数据缓存效率, 从而提高整体效率. 同时, 本文实施延迟约减策略, 并非在每一层级的计算后立即执行约减, 而是基于具体计算结果, 仅约减那些可能超出数据表示范围的情况, 从而有效减少不必要的约减次数以加快计算速度. 此外, 基于伪梅森数不完整 *NTT* 旋转因子中存在大量−1 和 1 的情况, 为 s 进一步减少约减次数, 对于旋转因子为 1 的蝴蝶操作, 省略乘法直接进行加减, 对于旋转因子为−1 的蝴蝶操作, 将 $a[i]+a[j]\times twiddle$ (*twiddle* 代表旋转因子) 变为 $a[i]-a[j]$, 将 $a[i]-a[j]\times twiddle$ 变为 $a[i]+a[j]$.

3.2 基于 Tensor Core 的教科书式多项式乘法

本节首先给出了素阶数域 $\mathbb{Z}_q[x]/(x^n-x-1)$ 上多项式乘法矩阵表示的数学推导, 然后给出了基于 Tensor Core 的素阶数域上多项式乘法的 GPU 实现.

3.2.1 素阶数域上多项式乘法的矩阵转化

假设 $s=f\cdot g$ (注意这里并未进行多项式模运算). 教科书式的多项式乘法, 在得到 s 的基础上, 利用素阶数域上的特殊性质 $x^n\equiv x+1$ 进一步计算 $h=s \bmod (x^n-x-1)$. 图 2 以 $n=3$ 为例, 展示了性质 $x^n\equiv x+1$ 对多项式模运算的影响. 我们使用 p_i 代表多项式 p 的第 i 个多项式系数. 图 2 中 s_3 所在虚线框发射出两条箭头分别到达 s_0 和 s_1, 这两条箭头代表依据 $x^3\equiv x+1$, s_3 在多项式模运算中对第 0 位和第 1 位存在影响, 即 s_3 会被累加到 h_0 和 h_1 上. 进一步推广得到如下规律: 当 $i\geqslant n$ 时, s_i 会被累加到 $h_{i \bmod n}$ 和 $h_{(i \bmod n)+1}$ 上. 应用该规律, 得到矩阵 M_1, 对其每一列进行求和, 即可得到最终结果多项式 h 的各个系数. 进一步地, 将矩阵 M_1 拆分为矩阵和向量的乘积 $(M_2+M_3)\cdot v$, 其中 v 为多项式 g 各个系数构成的列向量, M_2 和 M_3 由多项式 f 导出, 分别对应了 s_i 对 $h_{i \bmod n}$ 和 $h_{(i \bmod n)+1}$ 的影响.

设 f和g 是 $n-1$ 次多项式, 其中 $f=\sum_{i=0}^{n-1}f_ix^i$, $g=\sum_{i=0}^{n-1}g_ix^i$, 二者的直接乘积可以写为 $s=\sum_{i=0}^{2n-2}s_ix^i$, 其中:

$$s_i=\begin{cases}\sum_{k=0}^{i}f_{i-k}g_k, & 0\leqslant i\leqslant n-1\\ \sum_{k=i-n+1}^{n-1}f_{i-k}g_k, & n\leqslant i\leqslant 2n-2\end{cases} \tag{1}$$

设二者在素阶数域上的乘积为 $h=\sum_{i=0}^{n-1}h_ix^i$, 则:

$$h_i=\begin{cases}s_0+s_n, & i=0\\ s_i+s_{i+n}+s_{i+n-1}, & 1\leqslant i\leqslant n-2\\ s_{n-1}+s_{2n-2}, & i=n-1\end{cases} \tag{2}$$

展开后得到一般公式为:

$$h_i=\begin{cases}f_0g_0+\sum_{k=1}^{n-1}f_{n-k}g_k, & i=0\\ \sum_{k=0}^{i}f_{i-k}g_k+\sum_{k=i+1}^{n-1}f_{n+i-k}g_k+\sum_{k=i}^{n-1}f_{n+i-1-k}g_k, & 1\leqslant i\leqslant n-2\\ \sum_{k=0}^{n-1}f_{n-1-k}g_k+f_{n-1}g_{n-1}, & i=n-1\end{cases} \tag{3}$$

进一步地, 用矩阵乘法来表示 h 的系数为:

$$\left(\begin{bmatrix}f_0 & f_{n-1} & f_{n-2} & \cdots & f_1\\ f_1 & f_0 & f_{n-1} & \cdots & f_2\\ f_2 & f_1 & f_0 & \cdots & f_3\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ f_{n-1} & f_{n-2} & f_{n-3} & \cdots & f_0\end{bmatrix}+\begin{bmatrix}0 & 0 & 0 & \cdots & 0\\ 0 & f_{n-1} & f_{n-2} & \cdots & f_1\\ 0 & 0 & f_{n-1} & \cdots & f_2\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & 0 & \cdots & f_{n-1}\end{bmatrix}\right)\begin{bmatrix}g_0\\ g_1\\ g_2\\ \vdots\\ g_{n-1}\end{bmatrix}=\begin{bmatrix}h_0\\ h_1\\ h_2\\ \vdots\\ h_{n-1}\end{bmatrix} \tag{4}$$

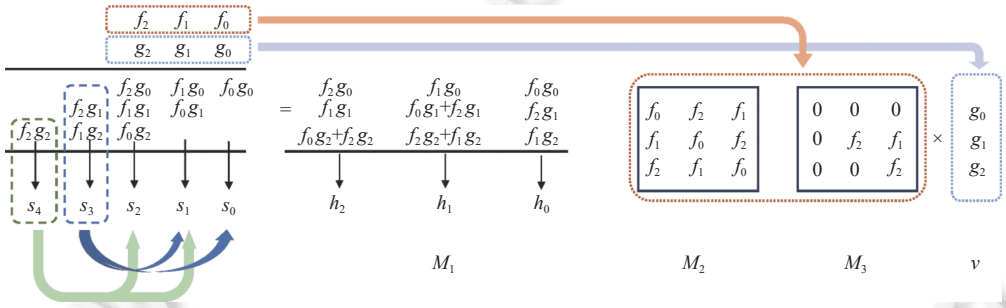

图 2 素阶数域上多项式乘法的矩阵转化

3.2.2 基于 Tensor Core 的实现方法与细节

在 GPU 的 C++编程中, 和传统的 CUDA Core 相比, Tensor Core 能进一步加速 MMA 运算, 即 $D=A\times B+C$, 其中 A 为 $m\times n$ 的矩阵, B 为 $n\times k$ 的矩阵, C 和 D 为 $m\times k$ 的矩阵. Tensor Core 使用 wmma::fragment 类来定义不同的矩阵类型, 其中 A 为矩阵乘法的左矩阵, B 为矩阵乘法的右矩阵, C 和 D 为 MMA 操作中的累加器. Tensor Core 支持多种混合精度的元素类型和矩阵大小, 即以低精度为输入, 高精度为输出, 表 3 给出了部分能够同时支持 3 种矩阵大小 (16×16×16、32×8×16、8×32×16) 的左矩阵、右矩阵、累加器的组合精度示例.

表 3 Tensor Core 支持的部分组合精度[29]

左矩阵	右矩阵	累加器
_half	_half	float
_half	_half	_half
unsigned char	unsigned char	int
signed char	signed char	int

Tensor Core 所支持的混合精度操作在 CUDA 中提供了相应的编程接口 wmma. 概括来说, 包括 1 个数据类型 (wmma::fragment) 和 3 个操作 (wmma::load_matrix_sync、wmma::store_matrix_sync 和 wmma::mma_sync), 它们都需要一个线程束协同完成. 例如, 当 m=16、n=16、k=16 时, 定义在核函数中的 wmma::fragment 对象, 运行时会以 8 个 8 位整型的寄存器出现在每一个线程的运行空间中. wmma::load_matrix_sync 操作实现从连续内存区中加载值到矩阵. wmma::store_matrix_sync 操作实现方向相反的操作, 即将矩阵中的值存储到连续的内存区域中. 上

述两个操作中, 连续内存空间和矩阵 (本质上是各个线程的寄存器) 间的映射关系对编程者是透明的. Zhou 等人[21]通过逆向的方式获取该种映射关系. wmma::mma_sync 操作完成线程同步的 MMA 操作. 对于不同的精度, Tensor Core 的性能是不同的. 概括来说, 精度越低, 运算速度越快. 根据文献 [20], 当使用 8 位整型作为计算精度时, MMA 操作表现出最短的延迟.

基于此, 在 RTX 3060 平台上本文选择 SM86 架构支持的 m=16、n=16、k=16, 左矩阵和右矩阵精度为 8 位整型, 累加器精度为 32 位整型的 MMA 操作来完成 CTRU-Prime 中的多项式乘法. 在 CTRU-Prime 中, 需要计算的多项式乘法包含 2 种, 第 1 种是 $\mathcal{R}_q$ 中的多项式乘法 $h=g\cdot f_{\text{inv}} \bmod q$ 和 $\sigma=hr \bmod q$; 第 2 种是 $\mathcal{R}_{q_2}$ 中的多项式乘法 $m'=cf=c(2f'+1) \bmod^{\pm} q_2$. 其中 g、r、f' 由参数为 2 或 3 的中心二项分布采样得到, 因此 g、r、f 只需一个 8 位整型表示. f_{inv}、h、c 是 $\mathcal{R}_q$ 上的多项式. 如表 1 所述, CTRU-Prime 的 3 组 q 都为 13 比特, 因此本文使用 2 个 8 位整型来表示 $\mathcal{R}_q$ 上的多项式系数.

图 3 展示了使用 Tensor Core 完成多项式乘法 $h=g\cdot f\in\mathcal{R}_q$, 其中 g 由两个 8 位整型表示, f 由一个 8 位整型表示. 首先, 将 g 和 f 依据公式 (4) 转化为 $A_x\times A_y$ 维矩阵 G 和 $B_x\times B_y$ 维矩阵 F. 需要注意的是, 由于矩阵乘法操作会对矩阵进行分块, 因此需要满足条件 $m|A_x$、$n|A_y$、$k|B_y$、$A_y=B_x$. 本文使用在空缺位置填充 0 的方式来满足上述条件. 例如, 对于 CTRU-Prime-653 首先依据公式 (4) 转化为 653×653 维矩阵和 653×1 维矩阵的乘法, 在填充 0 后, 变为 656×656 维矩阵和 653×16 维矩阵的乘法. 进一步地, 将矩阵 G 分为矩阵 G_h 和 G_l, 其中 G_h 对应 G 的高 7 比特, G_l 对应 G 的低 7 比特. 在完成多项式到矩阵的转化后, 基于分块矩阵乘法的思想, 使用 $m\times n\times k$ 维的 wmma 操作完成矩阵乘法 $H_h=G_h\cdot F$ 和 $H_l=G_l\cdot F$. 具体来说, 每一个线程束负责一个 $m\times k$ 维子矩阵的计算, 为此其需以 $m\times n$ 维子矩阵为单元遍历矩阵 A 的一行分块矩阵, 以 $n\times k$ 维子矩阵遍历矩阵 B 的一列分块矩阵, 在遍历过程中使用 wmma 操作完成一次矩阵乘法并累加计算结果到矩阵 C 中. 最后, 合并矩阵 H_h 和矩阵 H_l 得到矩阵 H, 并将矩阵 H 转化为多项式 h. 若 g 由一个 8 位整型表示, f 由两个 8 位整型表示, 大致过程和上述类似, 只是 f 而非 g 需要拆分为高、低比特两个矩阵, 在此不再赘述.

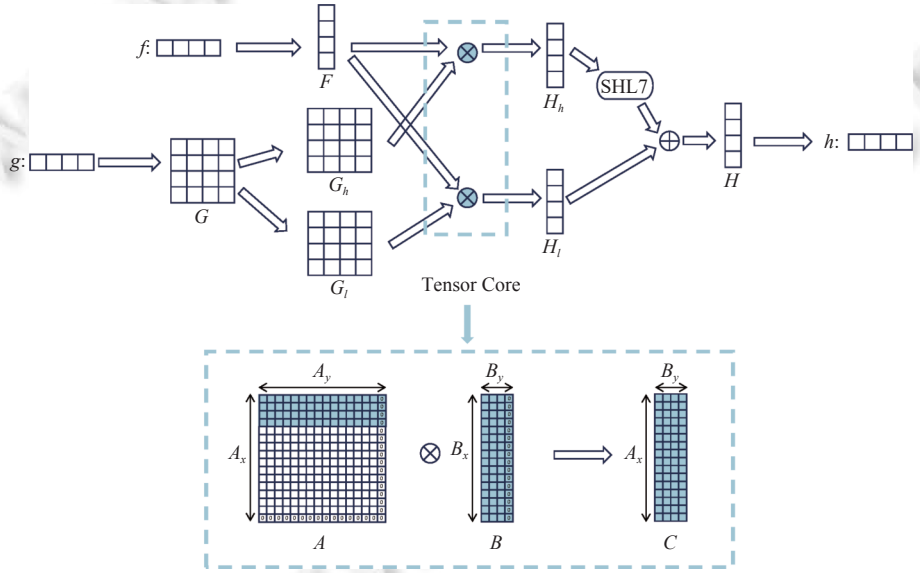

图 3 基于 Tensor Core 的素阶数域多项式乘法计算

在实现过程中, 核函数 *initMA* 完成多项式 g 到 G_h 和 G_l 的转变; 核函数 *initMB* 完成多项式 f 到 F 的转变; 核函数 *Twmma* 同时完成 $G_h\cdot F$ 和 $G_l\cdot F$; 核函数 *Merge* 完成 H_h 和 H_l 到 h 的转变. 内存模式上, 批量处理时, 以相邻的高位矩阵和低位矩阵为存储单元. 如算法 7 所示, 假设线程位于第 i 个线程束组, 其中每个线程束组处理一个多项式乘法, 含 *WRAP_NUMS* 个线程束. 在 *Twmma* 中, 如算法 7 的第 14、15 行所示, 该线程基于所在线程束编号

warpgroup 计算待访问 G_h、G_l 的位置偏移, 以 *warpgroup*/2 计算待访问 F 的位置偏移. 其中, G_h、G_l 以行主序存储于全局内存, F 则以列主序存储于全局内存.

算法 7. *Twmma*.

输入: 矩阵 A (大小为 $A_x \times A_y$) 对应的连续内存 *array_a*; 矩阵 B (大小为 $B_x \times B_y$) 对应的连续内存 *array_b*;
输出: 存储结果 $C = A \times B$ 的连续内存 *array_c*.

1. wmma::fragment<wmma::matrix_*a*, *m*, *n*, *k*, int8_t, wmma::row_major> *a_frag*;
2. wmma::fragment<wmma::matrix_*b*, *m*, *n*, *k*, int8_t, wmma::col_major> *b_frag*;
3. wmma::fragment<wmma::accumulator, *m*, *n*, *k*, int32_t> *c_frag*;
4. *tot_warpID* = (*blockIdx.x* × *blockDim.x* + *threadIdx.x*) / 32;
5. *warpgroup* = *tot_warpID* / *WRAP_NUMS*;
6. *warpID* = *tot_warpID* – *warpgroup* × *WRAP_NUMS*;
7. *row_idx* = *warpID* % ((B_y) / *k*) × *k*;
8. *col_idx* = *warpID* / ((B_y) / *k*) × *m*;
9. *st_offset* = *col_idx* + *row_idx* × A_x;
10. wmma::fill_fragment(*c_frag*, 0);
11. for *i* from 0 to (A_y) / *n* do
12. *ldA_offset* = *col_idx* × (A_y) + *i* × *n*;
13. *ldB_offset* = *row_idx* × (B_x) + *i* × *n*;
14. wmma::load_matrix_sync(*a_frag*, &*array_a*[$A_x \times A_y \times$ *warpgroup*] + *ldA_offset*,A_y);
15. wmma::load_matrix_sync(*b_frag*, &*array_b*[$B_x \times B_y \times$ (*warpgroup* / 2)] + *ldB_offset*,B_x);
16. wmma::mma_sync(*c_frag*, *a_frag*, *b_frag*, *c_frag*);
17. end for
18. wmma::store_matrix_sync(&*array_c*[$A_x \times B_y \times$ *warpgroup*] + *st_offset*, *c_frag*, A_x, wmma::mem_col_major);

在 IoT 场景下, 由于资源受限, 所有节点在特定会话中会使用同一公私钥对进行密钥封装与解封装, 这意味着在一次通信会话中使用相同的公私钥对执行数百到数千次 KEM 是很常见的[30]. 考虑到上述方法需填充 0, 为增大计算吞吐量, 提出面向相同公私钥的多项式乘法实现. 在该过程中, 相同的公私钥将展开为公式 (4) 中的矩阵, CTRU-Prime.PKE.Dec 中的密文 c 和 CTRU-Prime.PKE.Enc 中的随机多项式 r 将展开为公式 (4) 中的向量, 多个向量可以无缝拼接为矩阵而无需填充 0, 从而避免计算资源的浪费.

4 面向吞吐量的 CTRU-Prime 总体架构与优化实现

在本节中, 主要阐述面向吞吐量的 CTRU-Prime 实现方案的总体架构、优化策略以及多项式乘法以外的主要底层模块的 GPU 实现方案. 本文面向吞吐量设计, 基于 GPU 的不同特性, 提出不同的优化策略, 以最大化利用 GPU 的高并行性特点, 充分释放 GPU 的潜能.

4.1 设计概述

针对 CTRU-Prime 中的不同操作, 基于简单模式, 本文提取其并行度, 在最优数据划分的基础上, 使用多线程协同完成以减少操作延迟. 在核函数设计过程中, 以优化占用率为目标, 合理分配核函数使用的寄存器数、共享内存数和线程数, 从而最大化 SM 上活跃线程束数. 此外, 本文使用合并内存访问、转移数据交换到寄存器和共享内存、使用页锁定内存等内存管理技术, 进一步减少核函数的数据访问延迟, 提高核函数整体执行效率.

在此基础上, 本文充分考虑各操作之间的控制流和数据流关系. 鉴于 CTRU-Prime 无分支语句, 控制流较为简

单, 因此本文着重考虑连续操作间的数据依赖关系. 对于位于同一数据流上的操作, 本文使用融合内核技术, 将两个或多个独立的内核融合为一个内核, 有效降低了核函数启动和全局内存数据交换带来的开销.

面向吞吐量的 CTRU-Prime 总体架构如图 4 所示. 为进一步释放 GPU 在高吞吐量上的优势, 在单个核函数优化设计的基础上, 本文结合批量模式和单一模式, 引入批处理大小, 使得同时并发多个操作, 这充分利用了多 SM 的多核特点, 进而充分提高吞吐量. 进一步地, 实现过程中存在设备端和主机端的双向数据拷贝, 考虑 GPU 多流技术能够同步进行不同方向的数值拷贝和数值计算, 本文运用多流技术, 实现不同任务间的异步操作, 进一步提高了吞吐量. 为充分利用 CPU 端的多线程优势, 本文将多流技术和多线程技术相结合, CPU 端采用多线程模式, 使得 CPU 和 GPU 同时高效工作.

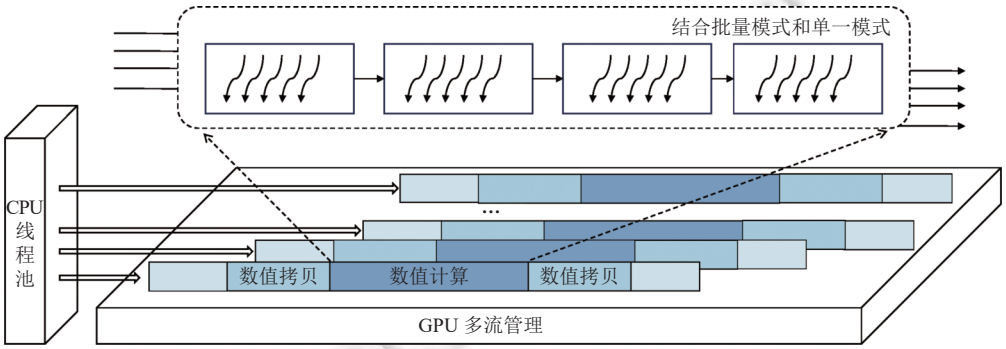

图 4 总计架构

4.2 多项式算子优化

除多项式乘法外, CTRU-Prime 中还包含多项式采样、多项式加法、多项式求逆等多项式算子. 本节对这些多项式算子的 GPU 优化方法进行说明.

4.2.1 多项式求逆

对于多项式求逆, 主要考虑了两种方法: 方法 1 基于 Bernstein 等人[31]提出的多项式求逆方法, 在合理数据划分的基础上利用多线程完成 GPU 实现; 方法 2 受 OpenSSLNTRU[32]的启发, 使用蒙哥马利求逆将 n 次多项式求逆转化为 1 次多项式求逆和 $n^2/2+3n/2-2$ 次多项式乘法. 具体来说, 为计算 $\mathbb{Z}_q[x]/f(x)$ 中的 n 个多项式 $(f_1,f_2,\ldots,f_n)$ 的逆, 即 $(f_1^{-1},f_2^{-1},\ldots,f_n^{-1})$, 首先计算 n 个多项式的累乘 $f_{\text{mul}}=\prod_{i=1}^{n}f_i$, 接着对 f_{mul} 进行一次多项式求逆运算得到 f_{mul}^{-1}, 最后只需利用乘法运算计算 $f_i^{-1}=f_{\text{mul}}^{-1}\cdot\prod_{i=1}^{k-1}f_i\cdot\prod_{i=k+1}^{n}f_i$. 本文分别实现了上述两种方式并进行测试. CTRU-Prime-653 的测试结果表明, 当批处理大小为 100 时, 方法 1 比方法 2 快了 56.01 倍, 这表明蒙哥马利求逆方法并不适合高并行度平台. 在传统平台上, 蒙哥马利求逆的优势在于转化更为耗时的多项式求逆为多项式乘法, 但在计算过程中引入了过多串行化的处理流程, 例如多个多项式相乘. 尽管多项式相乘的延迟较多项式求逆更小, 基于高并行性的特点, 相比串行化的多项式相乘, GPU 更适合并行化的多项式求逆. 此外, 单个多项式求逆无法充分利用 GPU 资源. 因此, 综合上述两个原因, 方法 1 比方法 2 展现出更优性能, 本文采用方法 1 完成多项式求逆操作.

4.2.2 多项式采样与多项式加法

在 CTRU-Prime 中, 使用中心二项分布完成多项式采样. 具体来说, CTRU-Prime-653 使用以整数 3 为参数的中心二项分布, CTRU-Prime-761 和 CTRU-Prime-1277 使用以整数 2 为参数的中心二项分布. 中心二项分布以随机值为输入, 以多项式为输出, 多项式系数的生成过程是相互独立的, 因此具备良好的并行特性. 在多项式系数的数据分块上, 本文综合考虑核函数的理论占用率和多项式系数对输入的依赖关系, 设计了一种基于寄存器的并行方案. 该方案将计算操作转移到高速寄存器中, 并在多个多项式系数之间共享寄存器值, 充分利用 GPU 资源.

多项式加法天然具备高并行度, 但对于 CTRU-Prime 而言, 直接将其实现为核函数并非最优解. 在 CTRU-Prime 中, 多项式加法主要包括两个操作: 其一为二倍乘操作 $g=f+f$; 其二为加一操作 $g=f+1$, 其中, f、g 为

$\mathbb{Z}_q[x]$ 中的多项式. 此外, 在 CTRU-Prime 中, 多项式加法是紧跟多项式采样后的操作, 因此, 本文将多项式加法融合到多项式采样核函数中. 具体来说, 设置两个控制变量分别控制是否进行二倍乘操作和加一操作, 某多项式系数的采样完成后, 在判断其是否为多项式第 0 位的基础上, 控制是否完成额外的多项式加法操作.

4.3 融合内核

内核融合 (kernel fusion)[33]是提升多线程 GPU 计算效率的一种高效方法. 该方法能够显著减少以特别耗时的全局内存访问为首的数据读写请求, 将大多数内存指令集中在指令延迟相对较低的寄存器和共享内存的读写上. 文献 [34] 提出的内核融合方法通过融合两个或多个独立的内核, 实现更高的利用率和对硬件资源更加均衡的需求, 为电源优化提供了更多潜力. 我们将内核融合技术应用到 CTRU-Prime 的 GPU 加速设计过程中.

将 CTRU-Prime 的每个操作转化为一个核函数是直接且自然的, 但这种方法忽略了操作之间可能存在的数据依赖性, 从而引入了大量的内核启动和数据传输开销. 例如, 在 CTRU-Prime.PKE.Enc 中, 需要接连完成多项式系数约减 (poly_subq)、多项式编码压缩 (poly_encode_compress) 和密文压缩 (pack_ct) 这 3 个操作. 由于这 3 个操作之间存在数据依赖关系, 即多项式编码压缩以多项式系数约减的输出为输入、密文压缩以多项式编码压缩的输出为输入, 将中间传递值从全局内存转移到共享内存或者寄存器能够有效减少访问延迟. 此外, 这 3 个核函数合并为 1 个也有效减少了内核启动带来的开销.

同一般核函数设计相似, 内核融合的设计首先需要考虑各个独立核函数内部的数据依赖图 (data dependency graph, DDG), 即结果值的生成对哪些值存在依赖关系, 这直接影响数据划分的方法. 减少对于全局内存的访问意味着数据的可交换性最大集中在线程块上, 这要求在函数执行过程中, 具有依赖关系的数据必须被划分到一个数据块上. 此外, 如上所述, 内核融合引入了两个独立函数之间的数据依赖关系, 这需要对齐多个函数的数据划分方法. 进一步地, 由于 SM 上限制了最大共享内存数量和最大线程数, 需要合理地设计线程组织形式以及内存分配策略以达到理想的占用率.

图 5 中展示了 3 个操作 (poly_subq、poly_encode_compress 和 pack_ct) 中的数据以及数据依赖关系. 其中, poly_subq 以 sigma 为输入和输出; poly_encode_compress 以 msg 和 sigma 为输入、以 c 为输出; pack_ct 以 ct 为输入和输出. 我们将 c 和 mh 转移到寄存器中, 从位于全局内存中的 msg 和 sigma 中读取对应输入分块, 完成运算后向全局内存 ct 中写入输出分块, 将 6 次对于全局内存的读写转化为 2 次全局内存的读写和对于寄存器的操作. 鉴于全局内存读取延迟比寄存器大 100 倍左右, 上述方法能够有效减少数据访问的延迟.

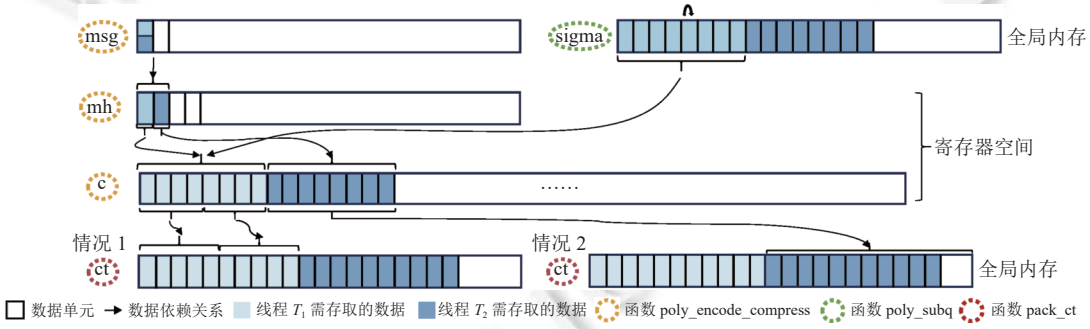

图 5 核函数融合

4.4 结合批量模式和单一模式

Gupta 等人[15]提出了 GPU 设计的两种模式: 批量模式 (batch mode) 和简单模式 (single mode). 本文的实现结合上述两种模式. 有些算法具有显著的固有并行性, 需要执行大量独立操作, 例如大型矩阵乘法、图像处理等, 大规模并行算法可以直接在 GPU 上实现, 并获得巨大的性能提升. 有些算法具有顺序性或半顺序性, 即算法的多个甚至全部步骤之间形成链式关系, 加快顺序算法速度的一种方法是执行多个并行实例, 而不是单个实例, 此类实现通常以批量方式处理数据, 在某些服务器应用程序中非常有用.

针对 CTRU-Prime 算法中的各个操作, 本文在提取其并行性的基础上, 基于单一模式设计核函数, 使用多线程执行单一操作, 从而有效提高执行速度. 此外, 为面向高吞吐量设计, 本文引入批处理大小, 基于批量模式使得单一核函数在正确数据划分后同时完成多次操作, 从而有效利用多 SM 的多核特点, 进一步提高吞吐量.

4.5 结合多流技术与多线程技术

在 GPU 中, 不同任务的数值计算和数值拷贝、不同方向的数值拷贝能够同步进行. 异步操作在提高 CPU 和 GPU 的整体性能方面起着至关重要的作用, 它允许任务同时执行, 而无需等待前面任务的完成. 这种方法有效地隐藏了数值拷贝和数值计算的延迟, 从而提高了吞吐量并减少了空闲时间. 在 CUDA 中, 有一个用于所有主机线程的默认流, 这会带来隐式同步. 例如, 考虑存在多个形如 (数值拷贝, 数值计算, 数值拷贝) 等任务的情况, 基于默认流的实现方式使得各个任务串行执行, 大大降低了资源的利用率. 而多流技术通过多个 CUDA 流, 使得不同操作的数值计算和数值拷贝、不同方向的数值拷贝能够同步进行.

本文使用 CPU 作为多流技术的调度器, 这需要 CPU 与 GPU 之间的同步. 为此, 在 CPU 端采用多线程模式, 每一个线程被分配一个唯一的 CUDA 流, 该流执行批处理大小个任务. 该种方法不仅能够让 CPU 和 GPU 同时高效工作, 还有效减少了数据传输的延迟, 提高了性能和资源利用率.

4.6 内存管理

GPU 的内存结构是分层的, 主要包括寄存器、本地内存、全局内存和共享内存这 4 种内存类型. 全局内存是层次化结构中的最高级别, 于片外内存中实现, 其大小比任何其他 GPU 内存类型都要大, 但访问延迟时间比共享内存和寄存器增加近 100 倍. 共享内存在 SM 的 L1 缓存 (片上内存) 中实现, 大小有限, 但是具有比全局内存更高的带宽和更低的延迟. 本文减少在全局内存中的数据交换操作以进一步提高访存效率, 通过合理分配线程组织结构和数据分块, 使得待交换数据仅局限在线程内或者线程块内, 从而可以分别使用寄存器和共享内存进行数据交换.

鉴于全局内存访问延迟较高, 本文在各个核函数的设计过程中, 控制线程束内线程访问连续内存地址从而形成合并内存访问, 以进一步提高访存效率. 具体来说, 由于 L1 缓存的缓存行大小为 128 字节, 因此每个内存访问事务都会导致读取 128 字节. 考虑线程束内的所有线程都以锁步方式执行, 如果其中一个线程出现延迟, 则所有剩余线程都必须等待访问完成. 因此, 如果内存访问是跨步的, 则大量获取的数据未被使用, 并且需要多个内存访问事务, 从而导致性能下降.

对 CUDA 架构而言, 主机端的内存被分为两种: 可分页内存和页锁定内存. 基于操作系统的分页存储机制, 分页内存会在物理内存和磁盘上交换, 而操作系统不会对页锁定内存进行分页和交换操作, 确保该内存始终驻留在物理内存中. 使用分页内存进行数据传输时, GPU 驱动会先将数据拷贝到主机中的临时页锁定内存中, 然后再由临时页锁定内存拷贝到 GPU 内存中, 从而完成数据从主机到设备的拷贝. 使用页锁定内存时, GPU 知道其物理地址, 通过直接内存访问 (direct memory access, DMA) 技术直接在主机和 GPU 之间复制数据, 和基于分页内存的数据传输相比, 减少了调度分页内存的开销, 从而提升了速度. 但是页锁定内存的不可交换性使得其消耗更多的内存空间. 因此, 本文通过使用适量的页锁定内存, 加速了数据传输, 从而提升方案的整体效率.

5 实验分析

5.1 实验设置

CPU 基线测试和 GPU 性能测试都在一台具有 6.5.0 kernel 的 Ubuntu 22.04.4 LTS 上完成, 其 CPU 型号为 Intel(R) Core(TM) i9-14900K. 本文使用 g++ 11.4.0 编译 C/C++代码, 使用 CUDA 11.5 完成 GPU 的实现, 代码实现在 NVIDIA GeForce RTX 3060 Ti 上部署, 其 CUDA 核心个数为 4 864, 显存带宽为 448 GB/s, 使用 O3 级别的优化. 本节测试结果由 1 000 次执行取平均获得, 需要注意的是, 本文将 CPU 和 GPU 之间的数据传输延迟也计算在内.

5.2 两种多项式乘法 GPU 实现比较

本节使用 CUDA 默认流, 在延迟和吞吐量两方面评估并比较第 3 节提出的两种多项式乘法的 GPU 实现方法.

为简化表述, 下文中使用 cuda 乘实现代表使用 CUDA Core 实现的基于伪梅森数不完整 *NTT* 的多项式乘法, 使用 tensor 乘实现代表基于 Tensor Core 的教科书式多项式乘法.

文献 [12] 采用伪梅森数不完整 *NTT* 完成多项式乘法, 并提供了表 2 所示 CTRU-Prime 的 3 组参数的多项式乘法的 C 语言实现. 如第 3.2.2 节所述, CTRU-Prime 在 $\mathcal{R}_q$ 和 $\mathcal{R}_{q_2}$ 上完成多项式乘法, 两者仅在域收缩阶段使用的约减算法上存在差异. 因此, 本文将文献 [12] 中在 $\mathcal{R}_q$ 上的多项式乘法 C 语言实现作为素阶数域上多项式乘法的基准测试结果. 具体而言, 当 CTRU-Prime 的参数 n 分别设置为 653、761 和 1 277 时, $\mathcal{R}_q$ 上的多项式乘法的平均运行时间分别为 13.98 μs、14.53 μs 和 26.32 μs.

图 6 展示了在不同批处理大小下, cuda 乘实现的测试结果. 得益于 GPU 的高并发处理能力, 随着批处理大小的增大, 多项式乘法的吞吐量显著提升. 当 n 分别为 653、761 和 1 277 时, 同基线测试结果相比, cuda 乘实现分别达到 1.09–11.08 倍、2.02–256.13 倍和 2.86–256.98 倍的吞吐量. 当 n=653 时, 伪梅森数不完整 *NTT* 由基 2-*NTT* 和基 3-*NTT* 混合而成, 这会在 GPU 实现过程中引入额外的分支执行, 因此同达到百倍加速级的两组其他参数相比, n=653 仅达到 10 倍级. 随着批处理大小的增加, GPU 任务负载也随之增加, 在一定范围内, 更大的批处理大小可以让 GPU 更好地利用其并行计算资源, 实际占用率也逐步增长. 当超过特定阈值后, GPU 处于饱和状态, 实际占用率趋于理论占用率, 例如, n=761 的理论占用率为 62.5%, 批处理大小为 64、128、256、512 和 1 024 时, 实际占用率分别为 20.06%、36.42%、49.62%、54.19% 和 56.72%. 当 GPU 处于饱和状态时, 核函数延迟与批处理大小呈线性增长关系, 吞吐量增速逐渐放缓. 因此受实际占用率影响, 如图 6 所示, n=653, 761, 1277 当批处理大小分别达到 32、256、256 后, 延迟开始与批处理大小成正比增加, 同时吞吐量的增长速率逐步下降. 此外, 将 cuda 乘实现延迟与基线结果进行比较: n=653 在批处理大小为 32 之前延迟分别为基线结果的 1.75–3.44 倍; n=761, 1277 在批处理大小小于 128 时, 在不同的批处理大小下, 延迟分别为基线结果的 42.31%–84.61% 和 35.02%–72.84%, 当批处理大小超过 128 时, 延迟分别为基线结果的 1.22–3.93 倍和 1.13–3.89 倍.

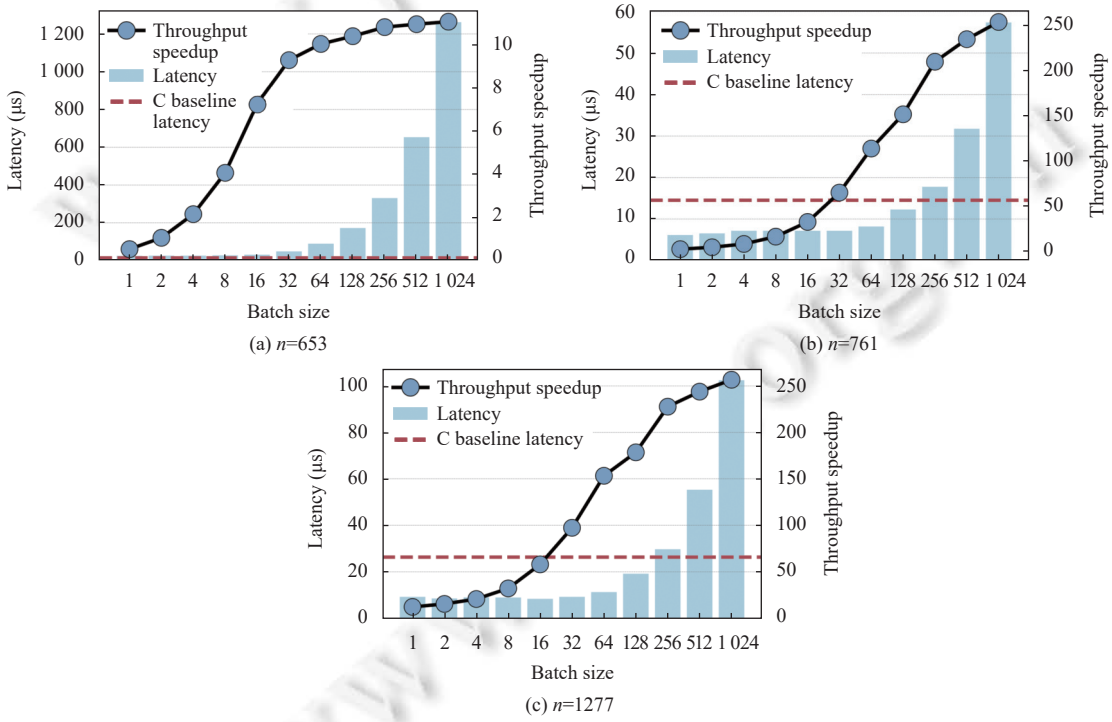

图 6 不同批处理大小的 cuda 乘实现吞吐量加速比与延迟

表 4 展示了 n=653 时两种 tensor 乘实现各个核函数的执行时间 (单位为 μs) 和相较于基线测试结果的加速比. 通用的 tensor 乘实现在不同的批处理大小上达到了 1.19–2.04 的加速比, 并在批处理大小为 8 时达到峰值. 横向比

较通用的 tensor 乘实现各个核函数的执行时间占比, *initMA* 和 *Twmma* 占比最多. *initMA* 需要频繁地和全局内存交互, 造成了较长的访存延迟; *Twmma* 使用 Tensor Core 完成矩阵乘法, 当批处理大小超过阈值 8 后, 执行时间呈线性增长, 这意味着在阈值处已充分利用 Tensor Core 的性能优势. 需要注意的是, 在通用的 tensor 乘实现中, 由于零填充操作会导致计算资源和内存资源的浪费, 因此同 cuda 乘实现相比, 在批处理大小逐步增大过程中, 性能表现逐渐落后. 面向相同公私钥的 tensor 乘实现, 针对 CTRU-Prime 中密钥封装算法, 使用相同的公钥封装不同的明文消息, 由于其仅需进行一次矩阵排列且向量排列无需填充 0, 充分利用了 Tensor Core 的计算资源, 达到了 10.36–177.24 的加速比, 其中, 当批处理大小为 512 时, 达到了 40.2 TOPS (Tera operations per second). 但是面向相同公私钥的 tensor 乘实现存在场景受限的缺点.

表 4 *n*=653 下 cuda 乘实现与两种 tensor 乘实现比较

批处理大小	通用的tensor乘实现					面向相同公私钥的tensor乘实现					cuda乘实现
	执行时间 (μs)				加速比	执行时间 (μs)				加速比	加速比
	initMA	*initMB*	*Twmma*	*Merge*		*initMA*	*initMB*	*Twmma*	*Merge*		
1	6.14	2.94	9.92	3.07	0.63	—	—	—	—	—	0.57
2	7.17	3.01	10.20	3.07	1.19	—	—	—	—	—	1.09
4	11.17	3.68	14.46	3.42	1.71	—	—	—	—	—	2.18
8	19.46	3.74	28.06	3.65	**2.04**	—	—	—	—	—	4.09
16	37.89	3.81	71.68	3.90	1.91	6.14	3.07	9.22	3.17	10.36	7.27
32	74.75	3.87	166.69	4.10	1.79	6.21	3.07	9.22	3.07	20.74	9.30
64	246.78	3.84	355.14	4.26	1.47	6.14	3.07	9.15	3.81	40.35	10.04
128	490.50	4.10	696.32	5.44	1.50	6.14	3.07	9.86	4.67	75.36	10.04
256	973.82	5.12	1 382.08	7.17	1.51	6.14	4.10	12.45	6.11	124.27	10.82
512	1 940.86	7.78	2 746.56	11.26	1.52	6.14	5.76	19.30	9.18	**177.24**	**10.96**

5.3 优化效果评估

5.3.1 融合内核

本节使用 CUDA 默认流, 以第 4.3 节中的示例为例, 在 *n*=653 下评估融合内核技术带来的性能提升. 表 5 展示了不同批处理大小下的核函数执行时间 (单位为 μs) 及节省的时间比率. 结果显示, 融合内核技术能显著减少核函数的执行时间, 且随着批处理大小的增加, 节省的时间比率进一步提高. 当批处理大小为 1 024 时, 相较于批处理大小为 2 时的 29.02%, 节省比率达到了 73.17%. 随着批处理大小的增大, 存储在全局内存中的待处理数据线性增长. 融合内核技术通过有效减少对全局内存的访问次数, 在更大批处理的情况下表现出更优的性能. 这表明, 融合内核技术在高吞吐量需求下尤为重要.

表 5 *n*=653 下融合内核和未融合内核的运行时间对比

批处理大小	延迟时间 (μs)		节省比率 (%)
	融合版延迟	未融合版延迟	
1	4.93	7.17	31.25
2	5.09	7.17	29.02
4	5.47	7.17	23.66
8	5.34	7.30	26.75
16	5.41	7.17	24.55
32	5.34	8.13	34.25
64	4.19	8.35	49.81
128	5.12	11.26	54.55
256	6.14	15.36	60.00
512	7.74	23.55	67.12
1 024	11.26	41.98	73.17

5.3.2 多流技术

多流技术通过覆盖数值计算和数值拷贝、不同方向的数值拷贝进一步提高吞吐量并减少空闲时间, 但是CPU 端使用多线程和 GPU 端多 CUDA 流会产生线程管理、流调用等性能开销. 本节以并行 768 个的 CTRU-Prime-761 的密钥生成函数、密钥封装函数和密钥解封装函数为例, 评估多流技术在不同 CUDA 流数量和数值拷贝任务量上带来的影响.

本节使用 3 组策略进行测量, 每组分别设置批处理大小为 384、192、96, 对比使用 k 个 CUDA 流和 k 次顺序执行两种方式 (k 依次为 2、4、8), 计算 k 个 CUDA 流相较于 k 次顺序执行节省的时间比率, 测试结果如图 7 所示. 3 个函数都在 $k=2$ 时出现了性能下降的情况, 这是由于 GPU 多流和 CPU 多线程产生的性能开销超过了多流技术带来的性能提升. 在相同函数下, 随着 CUDA 流数量的增加, 数值计算和数值拷贝以更细粒度进行, 节省的时间比率呈上升趋势. 密钥生成函数、密钥封装函数和密钥解封装函数在数值拷贝的任务量方面逐步递增, 在相同的 CUDA 流下比较 3 个函数, 任务量越多, 多流技术带来的性能提升更多.

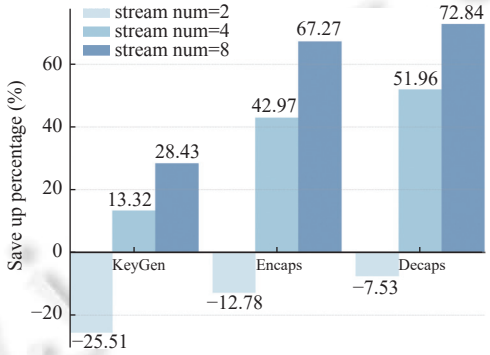

图 7 多流技术在不同的 CUDA 流数量和数值拷贝任务量上的性能比较

5.4 总体效果评估

CTRU-Prime 目前仅有 CPU 平台上的 C 实现[12], 在本节中, 使用具备普适性的 cuda 乘实现完成多项式乘法, 在 3 组参数下对密钥生成、密钥封装和密钥解封装进行测试, 并与 C 实现和其他相关工作进行对比.

同 C 实现相比, 基于 RTX 3060 平台, 本文在 CTRU-Prime 的 3 组参数上提供了 10.316–90.317 倍的吞吐量提升. 具体来说, CTRU-Prime-653、CTRU-Prime-761、CTRU-Prime-1277 每秒可以分别进行密钥生成 6.3 万、5.4 万、1.6 万次, 密钥封装 63.5 万、274.5 万、160.1 万次, 密钥解封装 35.1 万、262.2 万、152.4 万次, 分别是 C 实现版密钥生成吞吐量的 68.85、79.78、66.84 倍, 密钥封装吞吐量的 10.32、46.57、46.81 倍, 密钥解封装吞吐量的 11.43、89.19、90.32 倍. 相较于更适合小规模计算和通用任务的 C 实现, 本文更适合处理大批量任务处理的场景, 例如云计算、服务器等. 为进一步评估本方案的资源开销, 表 6 给出了本方案在 RTX 3060 平台的资源开销. 其中, 3 组参数功耗稳定在 80 W 左右, 低于其最大功耗限制 200 W, 显存占用峰值为 156 MiB, 占总显存的约 1.9%, 核心核函数单线程使用寄存器数量为 40 个, 线程块使用共享内存数量最多 30 KB.

表 6 CTRU-Prime 高吞吐量 GPU 实现的资源开销

方案	最高功耗 (W)	显存占用峰值 (MiB)	寄存器 (个)	共享内存 (KB)
CTRU-Prime-653	80	144	40	15.75
CTRU-Prime-761	82	146	40	18
CTRU-Prime-1277	81	156	40	30

此外, 将本文与相关工作中对 Kyber[20]和 NTRU 格基方案[26,35]等 GPU 加速实现进行比较, 主要衡量指标为吞吐量, 即每秒完成的操作数 (kOP/s), 测试与对比结果如表 7 所示. 上述工作均为闭源实现, 性能结果引用自其文章中的测试数据. 文献 [26,35] 为其他 NTRU 格基方案的最新实现, 本文的 3 组参数在不同函数上均展现出了

更优的性能, 提供了高达 26 倍的吞吐量. Kyber 是 LWE 路线的代表性格基密钥封装方案, 目前已被 NIST 标准化, 其底层代数结构为分圆环. 素阶数域使得 CTRU-Prime 能够抵御子域攻击等针对分圆环的攻击, 也引入了额外的性能开销, 例如多项式求逆无法被转移到子环上进行, 这使得无法有效提升最为耗时的多项式求逆的运算速度, 因此, 同 RTX 3080 平台上实现的 Kyber-768 相比, 具有相当安全性且基于 RTX 3060 平台实现的 CTRU-Prime-761 的密钥生成吞吐量较低. 除此之外, 在密钥封装和密钥解封装上同样展现出更优的性能, 与 Kyber-768 的 GPU 实现相比, 密钥封装吞吐量达到 1.46 倍, 密钥解封装达到 1.74 倍.

表 7　CTRU-Prime 的多平台实现对比和相关工作的吞吐量对比 (kOP/s)

方案	测试平台	吞吐量		
		密钥生成	密钥封装	密钥解封装
CTRU-Prime-653[12]	CPU	0.924	61.614	30.764
CTRU-Prime-761[12]	CPU	0.681	58.954	29.401
CTRU-Prime-1277[12]	CPU	0.243	34.204	16.883
CTRU-Prime-653	RTX 3060	63.639 (68.849×)	635.590 (10.316×)	351.532 (11.427×)
CTRU-Prime-761	RTX 3060	54.359 (79.779×)	2 745.392 (46.569×)	2 622.441 (89.197×)
CTRU-Prime-1277	RTX 3060	16.236 (66.843×)	1 601.174 (46.812×)	1 524.825 (90.317×)
Kyber-768[20]	RTX 3080	2 036	1 880	1 501
NTRUEncrypt[26]	GTX 1080	—	—	508
NTRU-HRSS[35]	RTX 2080	—	105	114

注: 括号内的数值为吞吐量相较于CPU平台基线的倍数

6 总　结

本文提出了一种基于 CUDA Core 和 Tensor Core 的 CTRU-Prime 高吞吐量实现. 针对素阶数域的非 *NTT* 友好性导致的 GPU 设计挑战, 本文提出两种素阶数域上多项式乘法的 GPU 实现方案. 基于 CUDA Core 的伪梅森数不完整 *NTT* 多项式乘法使用层融合技术优化访存模式, 在 n=653, 761, 1277 的 3 组参数下, 分别实现了 1.09–11.08 倍、2.02–256.13 倍和 2.86–256.98 倍的吞吐量提升; 基于 Tensor Core 的教科书式多项式乘法将多项式乘法转化为矩阵操作, 利用低精度 wmma 操作实现, 分别达到了 1.19–2.04 倍和 10.36–177.24 倍的吞吐量提升. 此外, 本文结合批量模式和单一模式、多流技术和多线程技术, 给出了 GPU 平台上面向吞吐量的 CTRU-Prime 总体架构, 并通过融合内核、合并全局内存访问、优化访存模式等优化策略, 显著加快了各核函数的访存和计算速度. 实验结果表明, 基于 RTX 3060 平台, CTRU-Prime-653、CTRU-Prime-761、CTRU-Prime-1277 每秒可以分别进行密钥生成 6.3 万、5.4 万、1.6 万次, 密钥封装 63.5 万、274.5 万、160.1 万次, 密钥解封装 35.1 万、262.2 万、152.4 万次, 分别是 C 实现版密钥生成吞吐量的 68.85、79.78、66.84 倍, 密钥封装吞吐量的 10.32、46.57、46.81 倍, 密钥解封装吞吐量的 11.43、89.19、90.32 倍. 与最新 Kyber 的 GPU 实现相比, 密钥封装吞吐量达到 1.46 倍, 密钥解封装达到 1.74 倍, 是其他 NTRU 格基 GPU 高吞吐量实现的 26 倍.

References

[1] Shor PW. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Review, 1999, 41(2): 303–332. [doi: 10.1137/S0036144598347011]

[2] Hoffstein J, Pipher J, Silverman JH. NTRU: A ring-based public key cryptosystem. In: Proc. of the 3rd Int'l Algorithmic Number Theory Symposium. Berlin: Springer, 1998. 267–288. [doi: 10.1007/BFB0054868]

[3] Liang ZC, Fang BY, Zheng JY, Zhao YL. Compact and efficient KEMs over NTRU lattices. Computer Standards & Interfaces, 2024, 89: 103828. [doi: 10.1016/J.CSI.2023.103828]

[4] IEEE standard specification for public key cryptographic techniques based on hard problems over lattices. IEEE Std 1363.1-2008, 2009. [doi: 10.1109/IEEESTD.2009.4800404]

[5] NTRUEncrypt history. 2011. https://en.wikipedia.org/wiki/NTRUEncrypt

[6] Schanck J, Whyte W, Zhang ZF. Circuit-extension handshakes for Tor achieving forward secrecy in a quantum world. Proc. on Privacy Enhancing Technologies, 2016, 2016(4): 219–236. [doi: 10.1515/POPETS-2016-0037]

[7] OpenSSH. OpenSSH release notes—OpenSSH 9.0/9.0p1. 2022. https://www.openssh.com/releasenotes.html [doi: 10.1007/s10623-013-9850-3]

[8] Augot D, Batina L, Bernstein DJ, Bos J, Buchmann J, Castryck W, Dunkelman O, GüneysuT, Gueron S, Hülsing A, Lange T, Mohamed MSE, Rechberger C, Schwabe P, Sendrier N, Vercauteren F, Yang BY. Initial recommendations of long-term secure post-quantum systems. 2015. https://pqcrypto.eu.org/docs/initial-recommendations.pdf

[9] Avanzi R, Bos J, Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, Schanck JM, Schwabe P, Seiler G, Stehlé D. CRYSTALS-kyber algorithm specifications and supporting documentation. NIST PQC Round, 2019, 2(4): 1–43.

[10] Weimerskirch A, Paar C. Generalizations of the Karatsuba algorithm for efficient implementations. IACR Cryptology ePrint Archive, 2006.224.

[11] Bernstein DJ, Chuengsatiansup C, Lange T, van Vredendaal C. NTRU prime: Reducing attack surface at low cost. In: Adams C, Camenisch J, eds. Proc. of the 24th Int'l Conf. on Selected Areas in Cryptography (SAC 2017). Cham: Springer, 2018. 235–260. [doi: 10.1007/978-3-319-72565-9_12]

[12] Liang ZC, Zhao XY, Fang BY, Zhao YL. Efficient and compact NTRU-based key encapsulation mechanism in large-galois-group prime-degree prime-ideal number field. Ruan Jian Xue Bao/Journal of Software, 2025, 36(2): 747–775 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7161.htm [doi: 10.13328/j.cnki.jos.007161]

[13] Hafeez MA, Lee WK, Karmakar A, Hwang SO. TMVP-based polynomial convolution for Saber and Sable on GPU using CUDA-cores and Tensor-cores. IACR Cryptology ePrint Archive, 2023.1541.

[14] Dai W, Sunar B, Schanck JM, Whyte W, Zhang Z. NTRU modular lattice signature scheme on CUDA GPUs. In: Proc. of the 2016 Int'l Conf. on High Performance Computing & Simulation (HPCS 2016). Innsbruck: IEEE, 2016. 501–508. [doi: 10.1109/HPCSIM.2016.7568376]

[15] Gupta N, Jati A, Chauhan AK, Chattopadhyay A. PQC acceleration using GPUs: FrodoKEM, NewHope, and Kyber. IEEE Trans. on Parallel and Distributed Systems, 2021, 32(3): 575–586. [doi: 10.1109/TPDS.2020.3025691]

[16] Lee WK, Seo H, Zhang ZF, Hwang SO. TensorCrypto: High throughput acceleration of lattice-based cryptography using tensor core on GPU. IEEE Access, 2022, 10: 20616–20632. [doi: 10.1109/ACCESS.2022.3152217]

[17] Sun SZ, Zhang R, Ma H. Efficient parallelism of post-quantum signature scheme SPHINCS. IEEE Trans. on Parallel and Distributed Systems, 2020, 31(11): 2542–2555. [doi: 10.1109/TPDS.2020.2995562]

[18] Gao YW, Xu J, Wang HB. cuNH: Efficient GPU implementations of post-quantum KEM NewHope. IEEE Trans. on Parallel and Distributed Systems, 2022, 33(3): 551–568. [doi: 10.1109/TPDS.2021.3097277]

[19] Shen SY, Yang H, Li WQ, Zhao YL. cuML-DSA: Optimized signing procedure and server-oriented GPU design for ML-DSA. IEEE Trans. on Dependable and Secure Computing, 2025, 22(3): 2295–2307 [doi: 10.1109/TDSC.2024.3494835]

[20] Wan LP, Zheng FY, Fan G, Wei R, Gao LL, Wang YW, Lin JQ, Dong JK. A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator. In: Proc. of the 27th European Symp. on Research in Computer Security. Copenhagen: Springer, 2022. 514–534. [doi: 10.1007/978-3-031-17143-7_25]

[21] Zhou T, Zheng FY, Fan G, Wan LP, Tang WX, Song YX, Bian Y, Lin JQ. ConvKyber: Unleashing the power of AI accelerators for faster Kyber with novel iteration-based approaches. IACR Trans. on Cryptographic Hardware and Embedded Systems, 2024, 2024(2): 25–63. [doi: 10.46586/TCHES.V2024.I2.25-63]

[22] Hafeez MA, Lee WK, Karmakar A, Hwang SO. High throughput acceleration of Scabbard key exchange and key encapsulation mechanism using tensor core on GPU for IoT applications. IEEE Internet of Things Journal, 2023, 10(22): 19765–19781. [doi: 10.1109/JIOT.2023.3282255]

[23] Kamal AA, Youssef AM. Enhanced implementation of the NTRUEncrypt algorithm using graphics cards. In: Proc. of the 1st Int'l Conf. on Parallel, Distributed and Grid Computing (PDGC 2010). Solan: IEEE, 2010. 168–174. [doi: 10.1109/PDGC.2010.5679887]

[24] Hermans J, Vercauteren F, Preneel B. Speed records for NTRU. In: Proc. of the 10th Cryptographers' Track at the RSA Conf. San Francisco: Springer, 2010. 73–88. [doi: 10.1007/978-3-642-11925-5_6]

[25] Bai TY, Davis S, Li JJ, Gu Y, Jiang H. Accelerating NTRU encryption with graphics processing units. Int'l Journal of Networked and Distributed Computing, 2014, 2(4): 250–258. [doi: 10.2991/IJNDC.2014.2.4.6]

[26] Akleylek S, Goi BM, Yap WS, Wong DCK, Lee WK. Fast NTRU encryption in GPU for secure IOP communication in post-quantum era. In: Proc. of the 2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/

CBDCom/IOP/SCI 2018). Guangzhou: IEEE, 2018. 1923–1928. [doi: 10.1109/SMARTWORLD.2018.00322]

[27] Gentleman WM, Sande G. Fast Fourier transforms: For fun and profit. In: Proc. of the 1966 Fall Joint Computer Conf. (AFIPS 1966). San Francisco: ACM, 1966. 563–578. [doi: 10.1145/1464291.1464352]

[28] Güneysu T, Oder T, Pöppelmann T, Schwabe P. Software speed records for lattice-based signatures. In: Proc. of the 5th Int'l Workshop Post-quantum Cryptography. Limoges: Springer, 2013. 67–82. [doi: 10.1007/978-3-642-38616-9_5]

[29] NVIDIA. CUDA C++ programming guide (legacy). 2024. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#element-types-and-matrix-sizes

[30] Lee WK, Hwang SO. High throughput implementation of post-quantum key encapsulation and decapsulation on GPU for Internet of things applications. IEEE Trans. on Services Computing, 2022, 15(6): 3275–3288. [doi: 10.1109/TSC.2021.3103956]

[31] Bernstein DJ, Yang BY. Fast constant-time GCD computation and modular inversion. IACR Trans. on Cryptographic Hardware and Embedded Systems, 2019, 2019(3): 340–398. [doi: 10.13154/TCHES.V2019.I3.340-398]

[32] Bernstein DJ, Brumley BB, Chen MS, Tuveri N. OpenSSLNTRU: Faster post-quantum TLS key exchange. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX Association, 2022. 845–862.

[33] Wu HC, Diamos G, Wang J, Cadambi S, Yalamanchili S, Chakradhar S. Optimizing data warehousing applications for GPUs using kernel fusion/fission. In: Proc. of the 26th IEEE Int'l Parallel and Distributed Processing Symp. Workshops & PhD Forum. Shanghai: IEEE, 2012. 2433–2442. [doi: 10.1109/IPDPSW.2012.300]

[34] Wang GB, Lin YS, Yi W. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In: Proc. of the 2010 IEEE/ACM Int'l Conf. on Green Computing and Communications & Int'l Conf. on Cyber, Physical and Social Computing. Hangzhou: IEEE, 2010. 344–350. [doi: 10.1109/GREENCOM-CPSCOM.2010.102]

[35] Seong H, Kim Y, Yeom Y, Kang JS. Accelerated implementation of NTRU on GPU for efficient key exchange in multi-client environment. Journal of the Korea Institute of Information Security & Cryptology, 2021, 31(3): 481–496. [doi: 10.13089/JKIISC.2021.31.3.481]

附中文参考文献

[12] 梁志闯, 赵旭阳, 方博越, 赵运磊. 素阶数域上的高效紧凑 NTRU 密钥封装方案. 软件学报, 2025, 36(2): 747–775. http://www.jos.org.cn/1000-9825/7161.htm [doi: 10.13328/j.cnki.jos.007161]

作者简介

胡晓雯, 硕士生, 主要研究领域为后量子密码, 密码工程.

邹恒川, 博士生, 主要研究领域为后量子密码.

沈诗羽, 博士, 主要研究领域为格密码, 同态加密, 密码工程.

李文倩, 博士生, 主要研究领域为后量子密码, 密码工程.

赵运磊, 博士, 特聘教授, 博士生导师, CCF 专业会员, 主要研究领域为后量子密码, 密码协议, 计算理论.