NTRU格基密钥封装方案GPU高性能实现

李文倩[1)] 沈诗羽[1)] 赵运磊[1),2)]
[1)](复旦大学计算机科学技术学院 上海 200433)
[2)](密码科学技术全国重点实验室 北京 100878)

摘 要 随着量子计算技术的发展,传统加密算法受到的威胁日益严重.为应对量子计算时代的挑战,各国正积极加强后量子密码算法的实现和迁移部署工作.由于NTRU密码方案具有结构简洁、计算效率高、尺寸较小、无专利风险等优点,因此NTRU格基密钥封装算法对于后量子时代的密码技术储备和应用具有重要意义.同时,图形处理器(Graphics Processing Unit,GPU)以其强大的并行计算能力、高吞吐量、低能耗等特性,已成为当前高并发密码工程实现的重要平台.本文给出后量子密码算法CTRU/CNTR的首个GPU高性能实现方案.对GPU主要资源占用进行分析,我们综合考虑并行计算、内存访问、数据布局和算法优化等多个方面,采用一系列计算和内存优化技术,旨在并行加速计算、优化访存、合理占用GPU资源以及减少I/O时延,从而提高本方案的计算能力和性能.本文的主要贡献在于以下几个方面:首先,针对模约减操作,使用NVIDIA并行指令集实现,有效减少所需指令条数;其次,针对耗时的多项式乘法模块,采用混合基NTT,并采用层融合、循环展开和延迟约减等方法,加快计算速度;此外,针对内存重复访问和冲突访问等问题,通过合并访存、核函数融合等优化技术,实现内存的高效访问;最后,为实现高并行的算法,设计恰当的线程块大小和数量,采用内存池机制,实现多任务的快速访存和高效处理.基于NVIDIA RTX4090平台,本方案CTRU768实现中密钥生成、封装和解封装的吞吐量分别为每秒1170.9万次、926.7万次和315.4万次.与参考实现相比,密钥生成、封装和解封装的吞吐量分别提高了336倍、174倍和128倍.本方案CNTR768实现中密钥生成、封装和解封装的吞吐量分别为每秒1117.3万次、971.8万次和322.2万次.与参考实现相比,密钥生成、封装和解封装的吞吐量分别提高了329倍、175倍和134倍;与开源Kyber实现相比,密钥生成、密钥封装和密钥解封装的吞吐量分别提升10.84～11.36倍、9.49～9.95倍和5.11～5.22倍.高性能的密钥封装实现在大规模任务处理场景下具有较大的应用潜力,对保障后量子时代的信息和数据安全具有重要意义.

关键词 后量子密码;格基密码;密钥封装方案;并行处理;图形处理器
中图法分类号 TP309 **DOI号** 10.11897/SP.J.1016.2024.02163

Efficient GPU Implementation of NTRU Lattice-Based Key Encapsulation Mechanism

LI Wen-Qian[1)] SHEN Shi-Yu[1)] ZHAO Yun-Lei[1),2)]
[1)](*School of Computer Science, Fudan University, Shanghai* 200433)
[2)](*State Key Laboratory of Cryptology, Beijing* 100878)

Abstract As quantum computing technology advances, the threats to traditional encryption algorithms are becoming more and more serious. This entails not only developing robust encryption techniques resilient to quantum attacks but also implementing comprehensive strategies to ensure the security of sensitive data and communication channels in the post-quantum era. In order to meet the challenges of the quantum computing era, countries are actively strengthening the

收稿日期:2023-09-04;在线发布日期:2024-05-13.本课题得到国家重点研发计划基金资助项目(No.2022YFB2701601)、密码科学技术国家重点实验室面上课题基金资助项目(No.MMKFKT202227)、上海市科委技术标准基金资助项目(No.21DZ2200500)、上海市协同创新基金资助项目(No.XTCX-KJ-2023-54)、上海市科委区块链关键技术攻关专项基金资助项目(No.23511100300)资助.**李文倩**,硕士研究生,主要研究领域为后量子密码、密码工程.E-mail:liwq22@m.fudan.edu.cn.**沈诗羽**,博士研究生,主要研究领域为后量子密码、同态加密和密码工程.**赵运磊**(通信作者),博士,复旦大学特聘教授,主要研究领域为后量子密码、密码协议、密码工程、计算理论.E-mail:ylzhao@fudan.edu.cn.

implementation, migration and deployment of post-quantum cryptographic algorithms. Since the NTRU cryptographic scheme has the advantages of simple structure, high computational efficiency, small size, and no patent risk, the NTRU lattice-based key encapsulation algorithm is of great significance to the reserve and application of cryptographic technology in the post-quantum era. At the same time, the Graphics Processing Unit (GPU), with its powerful parallel computing capabilities, high throughput, low energy consumption and other characteristics, has become an important platform in the implementation of current high-concurrency cryptographic engineering. We propose the first efficient GPU implementation of the post-quantum cryptographic algorithm CTRU/CNTR. Taking into account multiple aspects such as parallel computing, memory access, data layout and algorithm optimization, the main resource occupancy of the GPU is analyzed. We use a series of computation and memory optimization techniques to accelerate parallel computing, optimize memory access, reasonably occupy GPU resources, and reduce I/O delays, thereby improving the computing power and performance of this solution. The main contributions of our work are as follows: First, for the modular reduction operation, it is implemented using the NVIDIA parallel instruction set, which effectively reduces the number of instructions required. Besides, for the time-consuming polynomial multiplication module, we employ a mixed-base Number-Theoretic Transform (NTT) and utilize methods such as layer fusion, loop unrolling, and delayed reduction to speed up calculations. Additionally, for problems such as repeated memory access and conflict access, efficient memory access is achieved through optimization technologies such as memory coalescing and kernel fusion. Finally, to achieve high parallel algorithm computation, we design appropriate thread block size, and design a memory pool mechanism to achieve rapid memory access and efficient processing of multi-tasks. Based on the RTX 4090, our implementation of CTRU768 demonstrates throughputs of 11709k, 9267k and 3154k operations per second for key generation, encapsulation and decapsulation, respectively. Compared with the C language reference implementation, the throughput of key generation, encapsulation and decapsulation is increased by 336×, 174×and 128×, respectively. CNTR768 demonstrates throughputs of 11173k, 9718k and 3222k operations per second for key generation, encapsulation and decapsulation, respectively. Compared with the C language reference implementation, the throughput of key generation, encapsulation and decapsulation is increased by 329×, 175×and 134×, respectively. Compared with the latest open source Kyber implementation, the throughput of key generation, key encapsulation and key decapsulation is increased by 10.84～11.36×, 9.49～9.95×and 5.11～5.22× respectively. High-performance key encapsulation implementation by this work is the key to ensuring the smooth operation of large-capacity applications, and is helpful to ensure information and data security in the post-quantum era.

Keywords post-quantum cryptography; lattice-based cryptography; key encapsulation mechanism; parallel processing; Graphics Processing Units (GPU)

1 引 言

随着量子计算的不断发展,未来效率足够高、性能足够稳定的量子计算机可能会攻破绝大多数现有公钥密码体制,对现有的公钥密码体系造成颠覆性的威胁.为确保密码系统在量子计算时代保持安全,后量子密码[1](Post-Quantum Cryptography, PQC)应运而生.目前,后量子密码算法的标准化工作正在积极进行中.国际标准化组织(International Organization for Standardization,ISO)和美国国家标准与技术研究所(National Institute of Science and Technology,NIST)分别组织相关的标准化工作组,以推动后量子密码算法的标准制定.相关标准的制定将有助于确保后量子时代通信安全的可靠性和操作性.

研究可以抵抗量子计算攻击的、技术成熟的、高效稳定的密码算法是密码学研究的重要方向. 在2016 年,NIST 启动了在全球范围内征集后量子密码标准的项目. 2022 年,NIST 公布了 4 个拟标准化的算法[2],其中包括一个密钥封装方案(Key Encapsulation Mechanism, KEM):CRYSTALS-Kyber[3],三个数字签名算法:CRYSTALS-Dilithium[4]、Falcon[5]以及 SPHINCS+[6]. 在拟标准算法中,CRYSTALS-Kyber、CRYSTALS-Dilithium 和 Falcon 都是基于格的密码方案.

除了普通的整数格,目前基于格的实用后量子密码方案大部分都基于代数结构格,如理想格、模格和 NTRU 格. 这些方案主要通过以下两种假设进行实例化:一是带错误学习(LWE)[7]及其具有代数结构的变体,例如环上容错学习[8](RLWE)、模上的容错学习[9](MLWE)以及{R,M}LWE 的去随机化版本:舍入学习(LWR)及其变体,例如环上舍入学习(RLWR)和模上舍入学习(MLWR);二是 NTRU 假设[10],NIST 拟标准化的 Falcon 签名算法便是基于 NTRU 假设的方案.

NTRU 密码系统具有结构简洁、计算速度较快、尺寸较小等优点. NTRU 现已被 IEEE 1363.1 与 X9 标准所采用. NTRU KEM[11],包括 NTRU-HRSS 和 NTRUEncrypt 是 NIST 第三轮 PQC 标准化决赛入围算法之一. NTRU Prime KEM[12],包括 SNTRU Prime 和 NTRU LPRime 是 NIST 第三轮 PQC 标准化的候补候选者之一,目前已经在 OpenSSH 标准中默认强制应用,成为事实标准. 此外,基于 LWE 计算路线的密钥封装算法存在较为复杂的专利风险和制约因素. 因此,进一步发展自主可控的 NTRU 格基密钥封装算法对于我国后量子时代的密码技术储备和应用具有重要意义.

本文的研究对象 CTRU/CNTR[13]是由我国学者提出的 NTRU 格基的密钥封装方案,已在我国密标委正式立项[14]. 在综合安全性、带宽、错误概率和计算效率等方面,相较于 Kyber 和 NTRU-HRSS,CTRU/CNTR 具有显著的综合性优势. 该方案能够允许更大的密钥范围,并实现可扩展的密文压缩. 在相同维度上,CTRU/CNTR 比 Kyber、NTRU-HRSS 具有更高或相当的安全性和更小的带宽开销,比 Kyber 错误概率更低.

伴随着后量子密码应用的研究和密码工程技术的进步,将这些算法在各类计算平台高效实现成为当前后量子密码领域研究的热点和重要方向. 其中,GPU(图形处理单元)是专门用于处理图形和并行计算任务的硬件设备. 它作为计算机系统中的一部分,用于处理复杂的数值计算. 与中央处理单元(CPU)相比,GPU 具备大规模并行处理能力,能够同时执行多项计算任务. 它包含许多小型的处理单元(称为流式多处理器或 CUDA 核心),能够同时处理多个数据流. 此外,GPU 通常配备大容量的显存,用于存储数据,可以快速读写数据,以满足复杂计算的需求. 因此 GPU 也是许多领域中的重要工具,如机器学习、密码学和科学计算[15]. 由于其并行计算和高吞吐的特性,GPU 适合加速计算密集型的任务,提供更快的计算速度和更高的执行效率. 目前,已经存在密钥封装、签名算法等基于 GPU 平台的加速实现[16-20]. 高吞吐的密钥封装对保障大容量应用程序的安全至关重要,因为这类应用往往需要快速响应时间,所以高吞吐的算法实现是保证程序流畅运行的关键因素.

本文提出一套针对 CTRU/CNTR 的高性能 GPU 加速设计方案. 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,达到低延时和高吞吐的目标. 与现有的 C 参考实现相比,本方案的效率有着百倍级的提升. 本文的主要贡献有以下几点:

(1)CTRU/CNTR 是首个高维 NTRU 格和低维 E_8 格连接起来的密钥封装方案,在带宽和安全性等方面有着综合性优势. 针对 CTRU/CNTR 提出首个 GPU 加速方案,以满足该方案面向大批量任务场景下高吞吐量的需求.

(2)本方案通过算法优化解决底层运算复杂和并行设计困难等问题. 对于底层复杂的多项式乘法模块,采用更加灵活的混合基数论变换进行计算. 在解密过程中,针对 Polydecode 模数非 NTT 友好情况,采用多模数 NTT 进行并行加速设计.

(3)本方案通过设置最优线程配置达到实验 GPU 平台的最大并行限度,以满足高吞吐量的标准. 同时,合理分配资源达到 100%的理论占用率,设计最优并行度,加快访存,提升计算速度. 该方案在并行设计和资源利用等方面具有着借鉴价值.

实验结果表明,基于 RTX 4090 平台,本方案中 CTRU768/CNTR768 每秒钟可以进行密钥生成 11173～11709 万次,密钥封装 926～971 万次,密钥解封装 315～322 万次,相较于 C 语言参考实现,密钥生成、密钥封装和密钥解封装的速度分别提升

329～336 倍、174～175 倍和 128～134 倍. 此外，与最新开源 Kyber 实现相比，密钥生成、密钥封装和密钥解封装的吞吐量分别提升 20.02～20.98 倍、9.50～9.96 倍和 6.26～6.39 倍.

2 相关工作

在密码学领域，算法性能是影响算法优劣的重要因素之一. 为了提升算法性能，学者们一直在多种平台上进行探索和优化，包括 AVX 并行实现、ARM Cortex-M4 嵌入式平台的轻量级实现等. GPU 作为一种优化平台，也逐渐支持越来越多算法的优化实现.

关于 NTRU 格基密码算法的 GPU 加速研究，早在 2010 年，Kamal 等人[21]及 Hermans 等人[22]就开始了基于 GPU 平台进行 NTRUEncrypt 加速的早期尝试. NTRUEncrypt 算法的操作具有良好的数据并行处理特性，使得 GPU 平台非常适合于加速 NTRU. 在 2013 年，Bai 等人[23]提出单 GPU 零拷贝、单 GPU 数据传输和多 GPU 版本三种策略加速 NTRU. 实验结果表明，通过应用流和零拷贝等 GPU 计算技术，实现了计算与通信的重叠，以及随着参与设备的增加，NTRU 加速方案的性能将得到显著提升. 在 2016 年，Dai 等人[24]对 NTRU 格基签名(NTRU-MLS)方案进行 GPU 加速. GPU 加速实现方案可以同时生成许多候选对象，从而减轻拒绝采样带来的性能下降. 此外，也针对 NTRU 格基中计算复杂的底层模块进行加速研究. 在 2018 年，Lee 等人[25]实现高吞吐量的 NTRUEncrypt GPU 加速实现，其中提出 Karatsuba 多项式乘法获得 20.6%的改进. Wan 等人[26]利用基于 NVIDIA AI 加速器 Tensor Core 进行环上的多项式乘法运算加速，性能相对于 AVX2 实现提升两个数量级. 这些工作展示了在 GPU 平台实现 NTRU 格基密码算法已经取得一定的进展与成果，但仍存在发展的空间. 对于算法底层计算复杂的多项式乘法计算，利用 Tensor Core 设计加速，需要对多项式乘法进行单独设计，不利于与其他模块融合操作. 此外，多项式乘法可以进行共享内存访问优化策略，提高加载存储时的并行度. 本方案兼顾资源占用和执行效率，采用最新优化技巧，并结合具体算法进行细致的设计.

截至目前，针对四个 NIST 拟标准化算法，均在 GPU 平台取得最新的研究成果. 2020 年，Sun 等人[27]基于 GeForce GTX 1080 GPU 平台提出了高度并行和优化的 SPHINCS 实现. 他们用三个原语 ChaCha、Haraka 和 Keccak 来实例化底层哈希函数，得到在吞吐量和时延方面 SPHINCS 的最先进实现，并且在多核和多 GPU 卡上具有可扩展性. 2022 年，Wan 等人[28]对 CRYSTALS-Kyber 进行带有 Ampere Tensor Core 的 RTX 3080 的案例研究. 选择利用 NVIDIA AI 加速器 Tensor Core 加速多项式乘法，并通过 CUDA(Compute Unified Device Architecture，统一计算架构)C++ WMMA API 调用高性能的 NTT 模块. Shen 等人[17]提出了 Dilithium 的高吞吐 GPU 实现，充分利用并行性实现任务级批处理，同时实现快速内存访问的内存池机制. 并提出一种动态任务调度机制，显著提高多处理器资源占用率并减少执行时间. 2023 年，Lee 等人[15]基于先进的 GPU 架构，包括 RTX 3080、A100、T4 和 V100，开发 Falcon 采样过程的迭代版本，无需依赖 GPU 上耗时的递归函数调用，并提出并行随机样本生成方法，以加速 Mitaka 在 GPU 上的性能.

上述研究工作展示了 GPU 在密码学领域的广泛应用和显著效果. GPU 作为一种广泛应用于云服务平台的大规模并行架构，在密码学领域具有诸多优势. 它可以加速密钥生成、加密和解密等过程，为密码算法的研究和应用提供了更强大的计算能力. 实现基于 GPU 平台的高吞吐密码算法在包括物联网在内的新兴应用中有不可忽视的应用价值.

3 预备知识

3.1 CTRU 算法描述

由我国学者 Liang 等人[13]提出的 CTRU 是基于分圆环 $\mathcal{R}_q=\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$ 和尺度化 E_8 格编码设计的 NTRU 格基密钥封装方案. CTRU 基于 NTRU 假设和 RLWE 假设，允许更大的密钥范围，实现可扩展的密文压缩，具有更优异的性能表现. 在安全性、带宽、错误率和实现效率等方面，CTRU 具有综合性的优势.

CTRU. PKE(Public Key Encryption，公钥加密)具有选择明文攻击下不可区分性(IND-CPA)安全，CTRU. KEM 具有选择密文攻击下不可区分性(IND-CCA)安全. 通过 FO 转换[29](Fujisaki-Okamoto Transformation)的变体和前缀哈希($FO_{ID(pk),m}$)[30]将 CCA 安全的 KEM 归约到 CPA 安全的 PKE. CTRU768 的参数设置如表 1 所示.

整个方案的实现包含三部分：密钥生成(KeyGen)、密钥封装(Encaps)和密钥解封装(Decaps).

CTRU 具体算法描述如下所示:

算法 1. CTRU. KEM. KeyGen.

输入:安全参数 1

输出: $pk' := pk, sk' := (sk, z)$

1. $(pk, sk) \leftarrow$ CTRU. PKE. KeyGen()
2. $z \xleftarrow{\$} \{0,1\}^l$
3. RETURN $(pk' := pk, sk' := (sk, z))$

算法 2. CTRU. KEM. Encaps.

输入: pk

输出: (c, K)

1. $m \xleftarrow{\$} \mathcal{M}$
2. $(K, coin) := \mathcal{H}(ID(pk), m)$
3. $c :=$ CTRU. PKE. Enc$(pk, m; coin)$
4. RETURN (c, K)

算法 3. CTRU. KEM. Decaps.

输入: sk, c

1. $m' :=$ CTRU. PKE. Dec(sk, c)
2. $(K', coin') := \mathcal{H}(ID(pk), m')$
3. $\widetilde{K} := \mathcal{H}_1(ID(pk), m')$
4. IF $m' \neq \perp$ and $c =$ CTRU. PKE. Enc$(pk, m'; coin')$ THEN
5. RETURN K'
6. ELSE
7. RETURN $\widetilde{K}$
8. END IF

PKE. KeyGen 模块生成一对密钥对 (pk, sk). 在该模块中多项式 f', g 的多项式采样通过中心二项分布(Centered Binomial Distribution, CBD) B_η 生成,其中 $\eta = 2$ 或 $\eta = 3$. 具体算法如算法 4 所示:

算法 4. CTRU. PKE. KeyGen.

输出: pk, sk

1. $f', g \leftarrow \Psi_1$
2. $f := pf' + 1$
3. IF f is not invertible in $\mathcal{R}_q$, RESTART.
4. $h := g/f$
5. RETURN $(pk := h, sk := f)$

PKE. Enc 模块输入公钥 pk 以及消息 m. 通过执行加密操作,得到密文 c. 具体算法如算法 5 所示:

算法 5. CTRU. PKE. Enc.

输入: $pk = h, m \in M$

输出: c

1. $r \leftarrow \Psi_2$
2. $\sigma := hr + e$
3. $c := \frac{q_2}{q}(\sigma + PolyEncode(m)) \bmod q_2$
4. RETURN c

PKE. Dec 模块输入私钥 sk 以及密文 c. 通过 $PolyDecode$ 进行解密计算,得到消息 m. 具体算法如算法 6 所示:

算法 6. CTRU. PKE. Dec.

输入: $sk = f, c$

输出: m

1. $m := PolyDecode(cf \bmod \pm q_2)$
2. RETURN m

3.2　CNTR 算法描述

CNTR 是 CTRU 的简化版本,其设计实现与 CTRU 基本一致,主要区别在于以下几点:

(1)由 Ψ_1, Ψ_2 随机生成系数范围不同,CTRU 的系数范围为 $[-2, 2]$,由 B_2 生成. CNTR 的系数范围为 $[-3, 3]$,由 B_3 生成.

(2)PKE. Enc 中,CNTR 消除了噪声多项式 e. 并减少了对 $PolyEncode(m)$ 的四舍五入.

CNTR768 的参数设置如表 2 所示.

CNTR. PKE 具体的加密操作如算法 7 所示:

算法 7. CNTR. PKE. Enc.

输入: $pk = h, m \in M$

输出: c

1. $r \leftarrow \Psi_3$
2. $\sigma := hr$
3. $c := \frac{q_2}{q}(\sigma + PolyEncode(m)) \bmod q_2$
4. RETURN c

表 1　CTRU768 参数集

方案	n	q	q_2	(Ψ_1, Ψ_2)	$\lvert pk \rvert$	$\lvert ct \rvert$	B. W.	NTRU (Sec. C, Sec. Q)	LWE, primal (Sec. C, Sec. Q)	LWE, dual (Sec. C, Sec. Q)	δ
CTRU-768	768	3457	2^{10}	(B_2, B_2)	1152	960	2112	(181,164)	(181,164)	(180,163)	2^{-184}

表 2　CNTR768 参数集

方案	n	q	q_2	(Ψ_1, Ψ_2)	$\lvert pk \rvert$	$\lvert ct \rvert$	B. W.	NTRU (Sec. C, Sec. Q)	RLWR (Sec. C, Sec. Q)	δ
CNTR-768	768	3457	2^{10}	(B_3, B_3)	1152	960	2112	(192,174)	(191,173)	2^{-230}

3.3 NTT 前置知识

对于 CTRU/CNTR，运算是在环 $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n - x^{n/2} + 1)$ 上进行的.为了加速多项式乘法运算，采用灵活的混合基数论变换(Number Theoretic Transform，NTT)进行运算.对于多项式的乘法实现，如 $h = f \cdot g$，可以通过 INVNTT(NTT(f)∘NTT(g))来完成计算.其中，∘为点乘操作，NTT 表示正向 NTT，INVNTT 表示逆向 NTT.即对于两个多项式的乘法运算，首先对两个多项式进行 NTT 操作，将多项式的系数转变到 NTT 域内，然后执行点乘操作.最后，再通过 INVNTT 将 NTT 域内的点乘结果还原为正常域内的值.正向 NTT 的计算过程如下：

首先，先进行一层分解.由于 $\frac{3}{2}n \mid (q-1)$，因此在 $\mathbb{Z}_q$ 中计算得到 $\frac{3}{2}n$ 次的本原单位根 $\zeta = 5$.则可 $\mathbb{Z}_q[x]/(x^n - x^{n/2} + 1) \cong \mathbb{Z}_q[x]/(x^{n/2} - \zeta_1) \times \mathbb{Z}_q[x]/(x^{n/2} - \zeta_2)$，$\zeta_1 + \zeta_2 = 1$ 且 $\zeta_1 \cdot \zeta_2 = 1$.其中，$\zeta_1 = \zeta^{\frac{n}{4}} \bmod q$，$\zeta_2 = \zeta_1^5 = \zeta^{\frac{5n}{4}} \bmod q$.其次，$\mathbb{Z}_q[x]/(x^{n/2} - \zeta_1)$ 和 $\mathbb{Z}_q[x]/(x^{n/2} - \zeta_2)$ 可以通过六层基 2NTT 分解，得到环 $\mathbb{Z}_q[x]/(x^6 \pm \zeta^3)$ 上的结果.最后，进行一层基 3NTT 变换，以 $\mathbb{Z}_q[x]/(x^6 - \zeta^3)$ 为例，可以得到 $\mathbb{Z}_q[x]/(x^6 - \zeta^3) \cong \mathbb{Z}_q[x]/(x^2 - \zeta) \times \mathbb{Z}_q[x]/(x^2 - \rho\zeta) \times \mathbb{Z}_q[x]/(x^2 - \rho^2\zeta)$，其中 $\rho = \zeta^{n/2} \bmod q$.综上所述，$\mathbb{Z}_q[x]/(x^n - x^{n/2} + 1)$ 可以分解为 $\prod_{i=0}^{n/2-1} \mathbb{Z}_q[x]/(x^2 - \zeta^{\tau(i)})$.通过正向的混合基 NTT 得到 $\hat{f} = (\widehat{f_0}, \widehat{f_1}, \ldots, \widehat{f_{\frac{n}{2}-1}})$，其中，$\widehat{f_i} \in \mathbb{Z}_q[x]/(x^2 - \zeta^{\tau(i)})$ 是一个线性多项式，$i = 0, 1, \ldots, \frac{n}{2} - 1$.

在 CTRU/CNTR 的 NTT 实际实现中，采用一层分解、六层基 2NTT 以及一层基 3NTT.其中，基 2NTT 结合 Cooley-Tukey(CT)和 Gentleman-Sande(GS)算法进行实现.该方法减少原本所需乘法次数，通过运算速度更快的加法操作来替换乘法运算，以提高多项式乘法运算的速度.

3.4 模约减

在 CTRU/CNTR 算法的 GPU 实现中，由于多项式的计算结果都需要归约到 $\mathbb{Z}_q$ 域中，因此需要对数据不断地进行取模运算.为了提高约减效率并减少约减时间，我们采用延迟约减策略，仅在数据溢出之前进行约减运算.而取模运算的本质是一种除法运算，运算效率较低，耗时较长，并且容易受到侧信道攻击的威胁.故引入常数时间实现的模约减算法来改善计算过程中取模运算的低效性和安全性问题.常见的模约减算法[31]有 Montgomery 约减、Barrett 约减等.

3.4.1 Montgomery 约减

Montgomery 约减[32]适用于大模数情况下进行有效的模约减，一般用于乘法结果的约减.在 CTRU/CNTR 中，Montgomery 约减算法用于将 32bit 的数据约减至 16bit，约减后的结果在蒙哥马利域(Montgomery Domain)内.具体实现如算法 8 所示：

算法 8. 有符号的 Montgomery 约减.

输入：$0 < q < \frac{\beta}{2}$，$-\frac{\beta}{2}q \leqslant a = a_1\beta + a_0 < \frac{\beta}{2}q$.

其中，int32_ta，$0 < a_0 \leqslant \beta$，$\beta = 2^{16}$，q 为奇数

输出：$r' \equiv \beta^{-1}a \pmod q$，$-q < r' < q$，int16_$tr'$

1. $m \leftarrow a_0 q^{-1} \bmod \pm \beta$
2. $t_1 \leftarrow \left[\frac{mq}{\beta}\right]$
3. $r' \leftarrow a_1 - t_1$

3.4.2 Barrett 约减

Barrett 约减[33]能够消除计算中的除法运算，通常用于加减法的约减.在 CTRU/CNTR 中，Barrett 约减算法用于 16bit 数据加减运算后进行，以避免下一次计算后的结果溢出 16bit，如算法 9 所示：

算法 9. Barrett 约减.

输入：$0 < q < \frac{\beta}{2}$，$-\frac{\beta}{2} \leqslant a < \frac{\beta}{2}$.

其中，int16_ta，$0 < a_0 \leqslant \beta$，$\beta = 2^{16}$

输出：$r \equiv a \pmod q$，$0 \leqslant r \leqslant q$，int16_$tr$

1. $v \leftarrow \left[\frac{2^{\log(q)-1}\beta}{q}\frac{1}{1}\right]$
2. $t \leftarrow \left[\frac{av}{2^{\log(q)-1}\beta}\right]$
3. $t \leftarrow tq \bmod \beta$
4. $r \leftarrow a - t$

在本方案的具体实现中，预计算 v 和 t 以减少计算时间，从而简化为：$r \equiv a - \left[\frac{av}{2^k\beta}\right]q \bmod \beta$.其中，$k = \log(q) - 1$.

3.5 GPU 基础知识

GPU 是当今主流计算系统的重要组成部分，具有出色的性能和功能表现[34].它不仅是一个强大的图形引擎，还是一个高度并行的可编程处理器，其计算性能和内存带宽远超 CPU.本文旨在利用 GPU 良好的计算特性，对 CTRU/CNTR 中高计算强度的复杂问题进行优化解决.

3.5.1　线程层次结构

GPU 的可编程单元遵循单程序多数据(Single Program Multiple Dataflow,SPMD)的编程模型. 在 GPU 中,线程的层次结构主要包括线程、线程块(block)和网格(grid)三个层次. 线程是 GPU 中最基本的执行单元,线程块是由多个线程组成的逻辑单元,网格是由多个线程块组成的逻辑单元. 每个线程都有一个独立的 ID,用于计算内存地址和做出控制决策. 其中,一个线程束(warp)包括 32 个线程,是 GPU 中最小的执行单元[35]. GPU 可以同时运行多个线程块和网格,从而实现任务处理的高度并行化,呈现出卓越的性能. 善用 GPU 的线程层次结构可以更好地发挥其并行计算能力,线程块和网格的大小和数量可以根据具体的计算需求进行调整,从而提高程序的性能.

3.5.2　内存层级

GPU 具有相对于 CPU 的独立内存空间. 当 CPU 调用 GPU 进行运算时,首先需要将 GPU 所需的数据从 CPU 拷贝至 GPU. 在 GPU 上运算结束后,再将结果从 GPU 拷贝至 CPU. GPU 上的内存主要包括全局内存(Global Memory,GMEM)、共享内存(Shared Memory,SMEM)、寄存器以及本地内存等. 内存模型如图 1 所示:

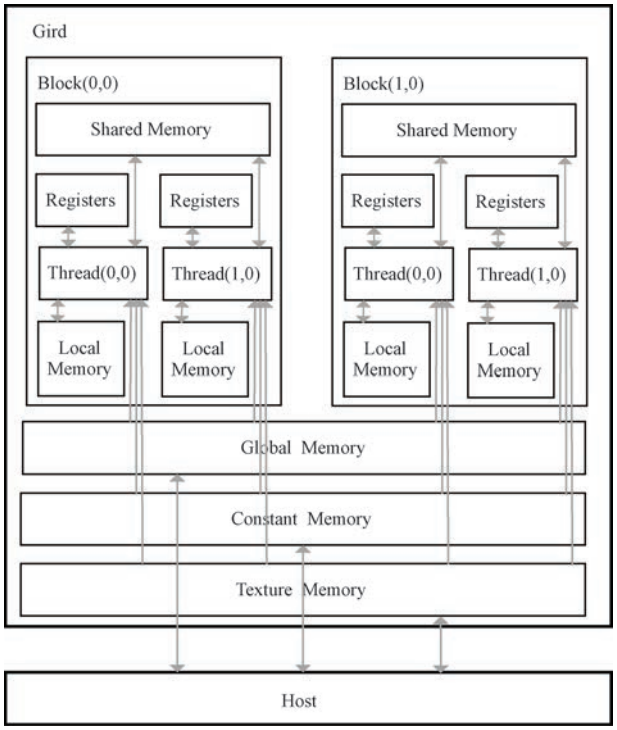

图 1　GPU 的内存模型

GPU 可以通过灵活地利用内存层次来实现低延迟的数据访问[17]. 每个执行线程都拥有私有的寄存器和本地内存. 寄存器提供最快的访问速度,而本地内存与全局内存具有相同的延迟. 线程块内的线程共同使用共享内存,共享内存比本地内存或全局内存拥有更高的带宽和更低的延迟. 全局内存可被所有线程访问,但具有最高的输入/输出(Input/Output,IO)延迟,并通过 L1 缓存和 L2 缓存进行交互[36]. 由于不同的内存层级具有不同的读写速度和容量,因此在 CUDA 编程中需要根据程序的需求选择合适的内存层级.

3.5.3　访存和计算

在 GPU 中,访存和计算是两个至关重要的方面. 不同类型的指令会被调度到不同的流水线(pipeline)中执行. 例如,算术逻辑单元(Arithmetic Logic Unit,ALU)主要执行算数操作和逻辑指令,加载存储单元(Load Storage Unit,LSU)主要负责处理存储器访问的加载和存储指令. 通过合理安排访存与计算的指令,并设计大量的线程并行,可以最大限度地减少线程等待时间.

3.5.4　NVIDIA 并行指令集

NVIDIA 并行指令集(Parallel Thread Execution,PTX)是一种稳定的编程模型和指令集,为通用并行编程提供了重要的支持[37],具有高效、灵活和可移植等特点. PTX 指令集涵盖多种工具和功能,包括对向量化、线程同步、内存访问和算术运算等基本操作的支持,提供灵活而丰富的工具集. 此外,PTX 指令集还具有对 GPU 硬件的更细粒度的控制和优化能力,能够更有效地利用 GPU 的并行计算资源. 编译器通常将 CUDA 和 C/C++等高级语言编译生成 PTX 指令,经过针对目标架构的优化处理,最终转换为相应的目标架构指令. 而直接编写 PTX 指令可以实现对指令的直观细致的设计,从而提高程序设计的效率和性能.

4　实现方案设计

在本章节中,主要阐述实现方案的总体设计架构,并详细说明 CTRU/CNTR 中涉及的主要模块的 GPU 实现方案. 通过仔细优化 CTRU/CNTR 各项操作,充分利用 GPU 高并行性的特点,以提高整体性能. 由于 CTRU 与 CNTR 的算法设计较为相似,在 GPU 具体实现上较为接近,因此本章的 GPU 具体实现以 CTRU768 为例.

4.1　设计概览

在 CTRU768 的 GPU 实现中,总体设计思路为线程块级并行,每个线程内执行一个 CTRU 任务. 在每个流式多处理器(Streaming Multiprocessors,

SM)上可以同时运行多个线程块,同时多个 SM 可以同时并行执行.这一设计可以充分利用 GPU 的层级架构,实现任务处理的高度并行化.此外,在每个线程块的任务处理中启动 3 个线程束,因为以算力 8.9 的 GPU 计算平台实现为例,由于硬件资源的限制,一个流式多处理器最多同时执行 16 个线程块,且最多同时执行 48 个线程束,所以每个线程块中同时执行 3 个线程束是资源利用最充分的状态.而 3 恰巧是多项式维度 768 的一个因子,因此 3 个线程束与 CTRU768 参数适配,每个线程可以得到充分的利用.然而,对于数据关联性较强的模块并不具有良好的并行特性,例如哈希函数,所以采用线程最小的执行单位线程束来实现.

在实现过程中,我们优化方案的资源利用,以达到较高的实际资源占有率.实际占用率与理论占用率接近,而理论占用率可以通过计算得到,因此我们先从理论上分析资源的分配.为达到 100%的理论占用率,在最大限度的活跃线程数量下,给每个线程块分配不超过 3072 字节的内存,每个线程占有不超过 42 个寄存器,以避免资源占用过多导致本地内存的使用.但在实际实现中,并非占用率越高越好.有时使用更多的寄存器占用率下降,但会呈现出更快的计算速度.因此,在综合考虑程序性能和资源占用后,在方案设计时需要灵活地进行 GPU 资源的分配.

此外,在 CTRU 方案中涉及大量数据的运算导致内存开销较高,因此访存时延也是影响性能表现的重要因素.为了优化访存性能,我们采取了以下措施:在线程的集中访问时,实现全局内存的合并访问、解决共享内存块访问冲突,并减少不必要的重复访问,尽可能地减少 IO 时间.同时,由于计算和访存是不同的流水线,我们调整计算和访存指令顺序,并设计尽可能多的线程并行,最大限度地实现多线程掩盖内存延迟.下面,将从主要算子并行设计和内存管理两方面详细介绍实现细节.

4.2 多项式采样设计

多项式采样主要包含两个过程,且采样过程中数据之间的计算相对独立,具有良好的并行特性.首先,将 4 个 8bit 的字节流依次拼接为 32bit 的无符号数.其次,利用拼接后的 32bit 无符号数计算固定比特数的汉明重量,生成 8 个多项式系数.基于 GPU 平台,设计了一种最大化寄存器利用率的并行方案.启动 3 个线程束,为每个线程分配 8 个寄存器,寄存器用于存储生成的多项式系数.在计算过程中,由于良好的并行性,数据对齐后每个线程只需完成一次字节拼接和一次无符号数转换为多项式系数的操作.此外,每个线程将拼接后的无符号数以及中间结果均存储在寄存器中,以加快访存.同时,使用循环展开、指令调整等方法提高流水线的利用率.

4.3 多项式乘法优化实现

模域内的多项式乘法是 CTRU 算法的底层复杂算数运算之一,是密钥生成、加密、解密中的关键步骤.其计算复杂度较高,计算速度影响整体密码算法的执行时间.因此,在密码学领域中,加速环上的乘法运算是一个重要的研究方向,而 NTT 则是实现其优化的一种重要技术.在本节中,将重点介绍本方案中多项式乘法使用 NTT 计算的具体实现方法.

4.3.1 NTT 实现

如前文所述,CTRU768 的 NTT 需要做一层分解,六层基 2NTT 和一层基 3NTT.我们使用层融合、延迟约减以及循环展开等技巧,提高计算速度.首先,采用层融合[38]的方法,读取一次数据.在寄存器中进行多层的 NTT 计算,减少访存开销.NTT 模块同样采用 3 个线程束实现.由于 $n=768$,因此每个线程需至少对 8 个系数进行计算,所以需要给每个线程最少设计分配 8 个寄存器.由于基 2NTT 的蝴蝶变换只涉及两个系数,且每层变换的数据之间相对独立,所以为了减少数据读取时间,我们可以同时处理 8 个系数,完成 3 层的基 2NTT 操作.每完成三层基 2NTT 后,进行一次寄存器与 SMEM 的数据交换,直至完成六层基 2NTT.CTRU768 的 NTT 第一层分解与六层基 2NTT 的操作均可以通过每个线程 8 个寄存器实现.但最后一层的基 3NTT 需同时对 $3^l, l>0$ 个系数进行变换,最终每个线程需分配 12 个寄存器.

此外,本方案还采用循环展开的技术.通过减少循环的迭代次数、提高指令级并行性、提高数据缓存效率和优化编译器优化等方面的优势,来提高程序的运行速度和效率.本方案还采用延迟约减的方法.在 NTT 以及 INVNTT 的实现中,约减算法也是较为耗时的操作,因此我们尽量减少约减次数.延迟约减核心思想是,并非对每层的结果都进行约减,而是通过具体的计算,仅对可能会超出数据范围的数据进行约减,以减少不必要的约减次数,提高运算速度.

综上所述,NTT 实现采用 3 个线程束并行计算,即 96 个线程,为每个线程分配 12 个寄存器.具体设计如下:首先,将系数读入线程的寄存器中,在完成一层分解后,寄存器与 SMEM 交换数据,以调整系数排列顺序.接着,完成三层基 2NTT,在与 SMEM 交换数据后,再次执行三层基 2NTT 计算.

最后,进行寄存器数据交换,完成最后一层的基 3NTT.在具体的实现过程中,采用层融合、循环展开以及延迟约减等优化技术,来提高计算速度.INVNTT 设计实现与 NTT 设计类似.

4.3.2　访问冲突解决方案

在 NTT 的 GPU 并行设计中,存在内存块访问冲突(bank conflicts)的性能瓶颈.在本节中主要介绍如何解决内存块访问冲突问题.

为了实现并行访问的高内存带宽,SMEM 被划分为 32 个可以同时访问的内存块(bank).多个内存请求的地址映射到同一个内存块中,则线程访问被序列化,访存的速度和效率有所降低.为提高内存访问的并行度,线程束中的 32 个线程应当对应访问不同的内存块.由此,我们提出在保证性能的前提下,通过填充空白单元来打乱原本的数据存储顺序,使得不同线程访问不同的内存块,从而避免访存冲突,增加内存访问的吞吐,减少访存时间.下面以 NTT 的实现为例,详细阐述如何解决内存块访问冲突问题.

首先,需要将待计算的数据存储于 SMEM 中以进行 NTT 运算.在 NTT 的 2～4 层,每个线程间隔 48 个 SMEM 单元将数据读取到寄存器,此时会存在内存访问 2 路冲突.为了解决此访问冲突,可以通过每 48 个 SMEM 单元增加 1 个空白单元的方式,确保同一线程束中的线程访问不同的内存块,具体实现如图 2 所示.其中,P 表示填充的空白内存单元,块中的数字表示系数索引.在 NTT 的 5～7 层,每个线程间隔 6 个 SMEM 单元读取数据至寄存器中,此时导致 3 路冲突.同样地,可以通过每 48 个 SMEM 单元增加 3 个空白单元解决访问冲突.具体实现如图 3 所示.在最后一层基 3NTT 中,每个线程间隔 1 个 SMEM 单元读取数据至寄存器中,这会导致 4 路冲突,可以通过每 96 个 SMEM 单元增加 1 个空白单元的方式解决,具体实现如图 4 所示.

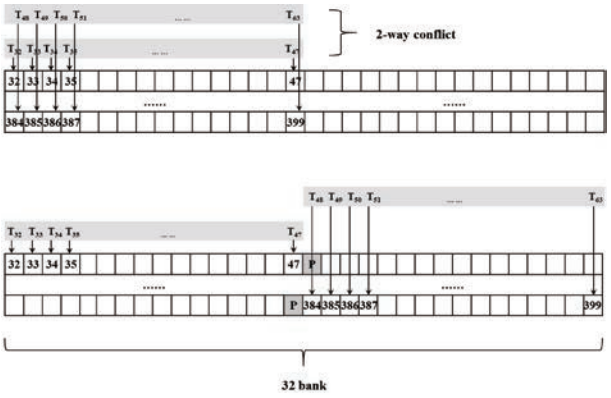

图 2　解决 NTT 2～4 层的 bank conflicts

通过上述的操作,在 NTT 阶段最多只需要增加 48 个 SMEM 空白单元,来解决内存块访问冲突问题.经测试,在解决内存块访问冲突之后,单次 NTT 的执行速度可以提升 8.83%.

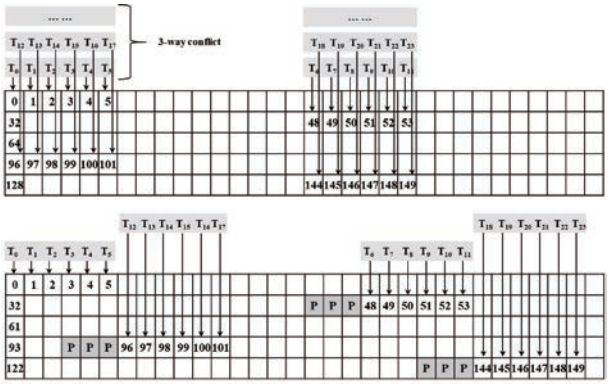

图 3　解决 NTT 5～7 层的 bank conflicts

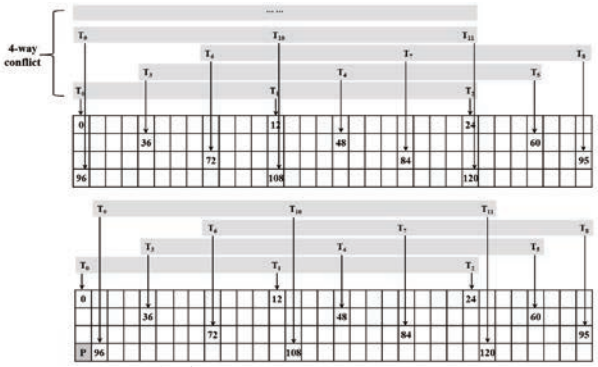

图 4　解决 NTT 基 3 层的 bank conflicts

4.3.3　核函数融合优化

核函数融合[39](kernel fusion)是提升多线程 GPU 计算效率的一种高效方法.其基本思想是将两个或者更多的核函数转换为一个核函数,相关操作也可以合并[40].核函数融合的优势主要体现在两方面:一是可以重用已经填充到寄存器或者 SMEM 中的数据,减少 IO 时间;二是可以减少“冗余”的负载和存储,实现内存空间的复用.

以多项式乘法 $res=\mathrm{INVNTT}(\mathrm{NTT}(c)\circ\mathrm{NTT}(f))$ 为例,考察核函数融合前后的优化效果,如图 5 所示.在未进行核函数融合前,需计算 $\hat{c}=\mathrm{NTT}(c)$,$\hat{f}=\mathrm{NTT}(f)$ 以及 $res=\mathrm{INVNTT}(\hat{c}\circ\hat{f})$ 三个核函数.其中,每个线程需要执行以下操作:

1. 将全局变量 c 加载至寄存器组 $regs$ 中.

2. 在寄存器组 $regs$ 中,完成 NTT 的操作.此计算过程中涉及 $regs$ 与 SMEM 的数据交换.

3. 将存储在 $regs$ 中的结果,存回全局变量 $\hat{c}$ 中.

4. 将全局变量 f 加载至寄存器组 $regs$ 中.

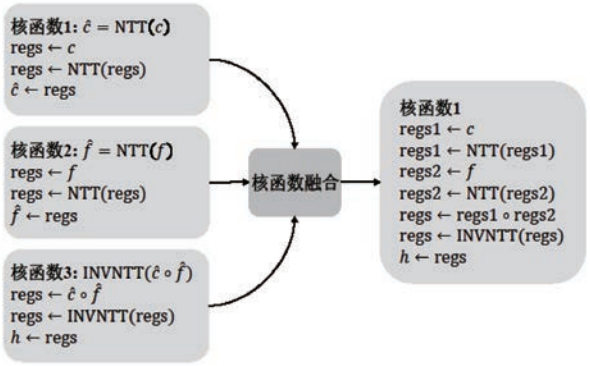

图 5 多项式乘法模块核融合操作

5. 在寄存器组 *regs* 中，完成 NTT 的操作. 此计算过程中涉及 *regs* 与 SMEM 的数据交换.

6. 将存储在 *regs* 中的结果存回全局变量 $\hat{f}$ 中.

7. 计算全局变量 $\hat{c}\circ\hat{f}$，计算结果存在 *regs* 中.

8. 在存储点乘结果的寄存器组 *regs* 中，做 INVNTT 操作. 在 INVNTT 计算过程中同样会涉及 *regs* 与 SMEM 的数据交换.

9. 将存储在 *regs* 中的结果，存回全局变量 *res* .

对以上 $res=\mathrm{INVNTT}(\mathrm{NTT}(c)\circ\mathrm{NTT}(f))$ 操作进行核函数融合，则仅需一个核函数完成操作. 对于核函数融合后，每个线程需要进行如下操作：

1. 将全局变量 c 加载至寄存器组 $regs1$ 中.

2. 在寄存器组 $regs1$ 中，完成 NTT 的操作. 此计算过程会涉及 $regs1$ 与 SMEM 的数据交换. 最终将 NTT 计算结果 $\hat{c}$ 存储在 $regs1$ 中.

3. 将全局变量 f 加载至寄存器组 $regs2$ 中.

4. 在寄存器组 $regs2$ 中，完成 NTT 的操作. 此计算过程中会涉及 $regs2$ 与 SMEM 的数据交换，这时占用的 SMEM 可以复用 $\mathrm{NTT}(c)$ 时的 SMEM. 最终，将 NTT 计算结果 $\hat{f}$ 存储在 $regs2$ 中.

5. 将存储在 $regs1$ 和 $regs2$ 的结果，直接在寄存器中完成乘法计算. 乘法结果存储在寄存器 *regs* 中，此时的 *regs* 复用 $regs1$ 或 $regs2$.

6. 在寄存器组 *regs* 中，完成 INVNTT 的操作. INVNTT 计算过程中同样会涉及 *regs* 与 SMEM 的数据交换，此时也可复用 SMEM.

7. 将存储在 *regs* 中的结果，存回全局变量 *res* 中. 也可以继续与其他核函数融合.

经过对多项式乘法的比较分析可知，核函数融合前后得到的计算结果相同. 然而，核函数融合后，通过复用已存储在寄存器中的数据，减少了两次 GMEM/SMEM 存储操作，一次 GMEM/SMEM 加载操作. 核函数融合通过复用数据，减少“冗余”的加载存储等访存操作等，有效地减少不必要的执行时间. 针对操作 $res=\mathrm{INVNTT}(\mathrm{NTT}(c)\circ\mathrm{NTT}(f))$，经测试，核函数融合后性能提升 7.80%，表现出较为有效的提升.

4.4 模约减算法加速方案

利用 PTX 指令集加速实现模约减算法，包括 Montgomery 约减和 Barrett 约减. Montgomery 约减 PTX 伪代码如算法 10 所示，首先通过 mul 指令进行 32bit 数的乘法操作，得到 32bit 的 t，并提供 cvt 指令进行 t 的数据类型转换，最后将 $t=a-(int32\_t)t*q$ 转化为 $t=a+(-(int32_t)t*q)$，利用一条 mad 乘加指令完成该计算，最后通过 shr 指令完成取高 16bit 的操作：

算法 10. Montgomery 约减汇编伪代码.

输入：$int32\_t$ a

输出：$int16\_t$ t

1. $mul.lo.s32$ t,q^{-1},a ▷$t=a*q^{-1}$
2. $cvt.s16.s32$ tmp,t ▷$t=(int16\_t)t$
3. $cvt.s32.s16$ t,tmp ▷$t=(int32\_t)t$
4. $mad.lo.s32$ $t,t,-q,a$ ▷$t=a-(int32\_t)t*q$
5. $shr.s32$ $t,t,16$ ▷$t\gg=16$

Barrett 约减 PTX 伪代码如算法 11 所示，首先通过 cvt 指令将 16bit 的数据扩展为 32bit，存在 32bit 的 tmp 中，使用 mul 指令进行乘法运算，利用 shr 指令进行取高位的操作，最后采用一条 mad 乘加指令完成 $t=a-t*q^{-1}$ 的计算：

算法 11. Barrett 约减汇编伪代码.

输入：$int16\_t$ a

输出：$int16\_t$ t

1. $cvt.s32.s16$ tmp,a ▷$tmp=(int32\_t)a$
2. $mul.lo.s32$ t,v,tmp ▷$t=v*tmp$
3. $shr.s32t,t,26$ ▷$t\gg=26$
4. $mad.lo.s32t,t,-q,tmp$ ▷$t=a-t*q$

由上述的 PTX 指令伪代码可以看出，通过对数据的处理，使用一条乘加指令 *mad* 执行两个操作，减少指令条数，在计算效率上有一定程度的提升.

4.5 哈希函数并行策略

哈希函数中数据之间关联性较强，并行性相对较弱. 因此，为了优化哈希函数的实现，采用缓解流水线阻塞、优化计算流程和内存访问模式的策略. 考虑到哈希函数的并行性较弱，设计时仅采用单线程

束的维度,以充分利用线程资源,避免资源的浪费.主要方案借鉴[17]中描述的方法,并加以指令级的优化调整.

首先,线程束中的每个线程加载预先计算的索引、轮常数和寄存器的偏移量,并在执行期间计算一些常量.将轮常数存储在常量内存中,以减少全局内存访问.这种方法平衡内存读写指令与算术运算指令,减少 IO 延迟和流程阻塞,从而提高流水线效率.其次,由于每个线程在吸收或挤压阶段需要加载或存储 8 字节值,因此提前对齐输入和输出流,用单次访问全局内存的内联函数替换字节加载和存储操作.这种优化减少了加载和存储的次数,从而减轻流水线的压力.此外,调整整体计算流程,特别是负责处理输入不足的分支,以减少线程分歧并提高整体性能.在用于内部状态更新的置换函数中,24 轮置换由 25 个线程执行,每个线程将 64 比特位的状态存储在寄存器中,每轮都采用线程束操作 warp shuffle 来使线程能够访问其他线程中寄存器的状态.哈希函数的并行实现显著提高算法整体性能、吞吐量和占用率,同时大幅减少了内存开销和访问.

4.6　解密过程优化设计

在 CTRU/CNTR 方案的解密过程中,涉及 PolyDecode($cf \bmod \pm q_2$),其中,$q_2=1024$ 是非 NTT 友好的素数,无法直接应用 NTT.求乘法 $cf \bmod \pm q_2$ 可以利用 Schoolbook 或者 Karatsuba 算法直接进行计算,但这两种方法并行度较差.鉴于 GPU 设计强调并行性,因此在 R_{q_2} 域上采用多模数 NTT 进行并行加速计算[41-42].

我们选择 $R_Q=\mathbb{Z}_Q[x]/(x^n-x^{n/2}+1)$ 的环,其中 Q 为正整数,且大于在 $\mathbb{Z}$ 上最大的系数绝对值.在具体实现中,设定 $Q=qq'$,选定 $q=3457$,$q'=7681$.由于 $q=3457$,$q'=7681$ 均为 NTT 友好的素数,我们分别计算在 $\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$ 和 $\mathbb{Z}_{q'}[x]/(x^n-x^{n/2}+1)$ 上的 NTT 变换.最后,根据中国剩余定理(Chinese Remainder Theorem,CRT)将这两个域同构 $\mathcal{R}_Q \cong \mathcal{R}_q \times \mathcal{R}'_q$ 恢复[41]到 $\mathcal{R}_Q$ 上.根据定理 1.得到在 $R_{q_2}=\mathbb{Z}_{q_2}[x]/(x^n-x^{n/2}+1)$ 上的结果.

定理 1.　令 q_i 为正奇数,并且两两互素($\gcd(p_i,p_j)=1, 1\leqslant i<j<s$).如果 $|u_i|<p_i/2$,$|u|<\prod_{i=1}^{s}p_i$,那么 $u\equiv u_i(mod\ p_i)$,$i=1,\cdots,s$ 的显式解 u 由下述式子可得

$$\begin{cases} y_1=u_1 \\ y_2=y_1+((u_2-y_1)m_2 \bmod \pm p_2)p_1 \\ y_3=y_2+((u_3-y_2)m_3 \bmod \pm p_3)p_1p_2 \\ \vdots \qquad \vdots \\ u=y_s=y_{s-1}+((u_s-y_{s-1})m_s \bmod \pm p_s)p_1\cdots p_{s-1} \end{cases}$$

其中,$m_i:=(p_1\cdots p_{i-1})^{-1} \bmod \pm p_i$.

将 $q=3457$,$q'=7681$ 代入上式计算,以 $q_2=1024$ 为模数进行约减,即可得到 $c*f$ 在 $R_{q_2}=\mathbb{Z}_{q_2}[x]/(x^n-x^{n/2}+1)$ 上的结果.由于 q_2 为非 NTT 友好的数,为了利用 GPU 进行并行加速计算,所以我们将问题转化为多次 NTT、INVNTT 和一次 CRT 进行求解,从而实现较高的并行度.经测试,此方案显著提升了计算效率.

4.7　内存管理

4.7.1　内存池设计

本方案使用内存池进行内存管理[43].内存池(memory pool),又称为固定大小的内存块分配,旨在确保并行任务处理中内存的高效访问.其基本原理是预先分配一些大小相同的内存块,为多组任务提供内存,来管理任务对内存资源的使用,并在使用完毕后释放.内存池分为设备内存池和固定内存池两种类型.设备内存池是指 GPU 设备内存,用于 GPU 内存分配.固定内存池是指不可交换的 CPU 内存,在 CPU 到 GPU 数据传输期间使用.

内存池的高效实现主要需要满足两个方面的要求:一是参数的存储顺序合理,二是内存的分配适当对齐[17].在分配内存池时,可以根据哈希函数参数的输入顺序,提前确定参数的内存排列顺序,以减少不必要的字节流拼接开销.密钥封装模块的参数分配顺序如图 6 所示,密钥解封装模块的参数分配顺序如图 7 所示.其次,由于对 L2 的存储器请求单位为 128 字节的高速缓存行,即 4 个连续的 32 字节扇区,所以尽可能地要求内存空间的对齐.但由于哈希函数的设计实现要求 256 字节对齐,所以最终的字节对齐按照 256 字节对齐,既可以满足 L2 高速的访存需求,又满足哈希函数的参数调用.

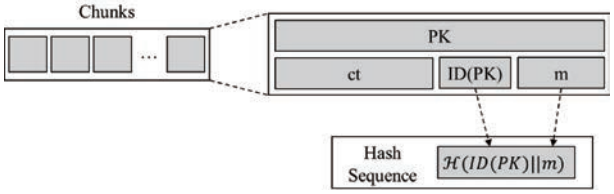

图 6　密钥封装模块的参数分配顺序

通过内存池的设计,本方案实现对内存的高效

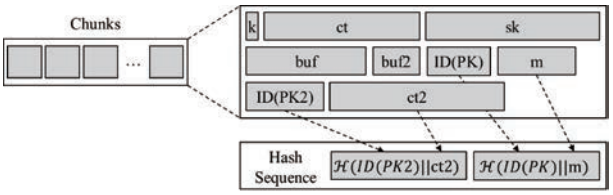

图 7 密钥解封装模块的参数分配顺序

管理,有效地利用系统资源,安全实现多个任务的并行处理.一次性的内存分配也减少了频繁内存分配与释放的开销,从而显著提升整体的性能表现.

4.7.2 优化占用率

单个 GPU 拥有多个流式多处理器,但每个流式多处理器上可分配的资源是有限的,包括寄存器、SMEM 等.此外,流式多处理器限制最大并行线程数,最大并行线程束数以及最大并行块数.占用率(Occupancy)是衡量流式多处理器上活跃线程束比例的重要参数,也是衡量是否合理分配线程的 GPU 资源的重要指标.核函数的占用率是指每周期内流式多处理器上活跃的线程束平均数量与支持的线程束最大数量的比值,包括理论占用率和实际占用率.一般情况下,实际占用率接近理论占用率,理论占用率可以根据理论上占用的资源进行计算.因此,如何合理利用有限的资源来达到更优的占用率,从而实现更高的效率,也是方案设计的关键.

对于本方案 8.9 算力的 GPU 实验平台而言,每个流式多处理器最多支持 16 个线程块,最多支持 48 个线程束,最多有 65532 个寄存器,最多有 65532 字节的 SMEM.若要达到 100%的理论占用率,则每个线程块最多分配 3072 字节,每个线程最多分配 42 个寄存器.核函数执行时的内存访问示意图如图 8 所示.一般而言,流式多处理器中活跃线程束数达到支持的最多线程束数,占用率为 100%.在此情况下,计算并行度最高,程序处理的效率最高.低的占用率会降低内存延迟隐藏的效果,表明程序存在性能瓶颈.但是高的占用率不一定性能最佳,可能会导致 GPU 资源的浪费.在某些情况下,增加寄存器的使用会导致占用率降低,但因为寄存器访问速度最快,GMEM 以及 L2 缓存访问延迟非常高,这反而是更优的选择.因此,需要根据具体应用场景,灵活地优化设计核函数的参数,以达到最优的 GPU 资源占用率.

4.7.3 合并访存策略

在不同的内存存储访问中,全局变量的访存是最为耗时的访存,所以,针对全局变量的访存进行优

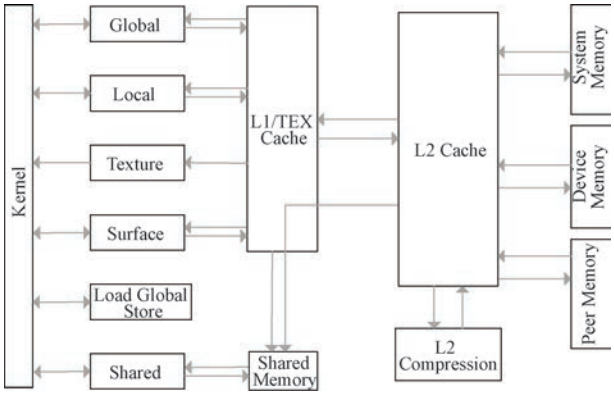

图 8 核函数内存访问示意图

化将会带来显著的效果.内存的合并访问(coalesced memory)是一种优化 GMEM 访问带宽的技术,通过将多线程的内存访问组织起来实现 GMEM 的高效访问.

CUDA 设备的 GMEM 是通过动态随机存取存储器(Dynamic Random Access Memory,DRAM)实现的.实际上每次访问 DRAM 的某个位置时,都会访问包括请求位置在内的一系列连续位置.在 CUDA 编程中,线程束中线程遵循单指令多线程(Single Instruction Multiple Threads,SIMT)在任何给定时间内执行相同指令,当线程束中的所有线程都执行加载指令时,硬件会检测是否访问连续的 GMEM 地址.如果访问连续的 GMEM 地址,硬件会将所有访问合并为对连续 DRAM 地址的合并访问,可以以 DRAM 突发[44](DRAM Burst)的方式传送数据.因此可以实现通过将线程的内存访问组织成有利的模式来实现 GMEM 访问的高效率.即比如,一个线程束中,线程 0 访问 GMEM 位置 x 、线程 1 访问地址 $x+1$、线程 2 位置 $x+2$ 等,这时所有的访问就可以合并为一个请求,来访问 DRAM 时的连续位置,即合并访问.否则,则需要多次的访问请求,消耗数倍的时间.

在 CTRU/CNTR 的 GPU 实现中,尽可能多地采用了合并访问的方法,将多个内存事务合并为一个更大的内存事务减少全局变量访问耗时.此外,合并内存的技术保证访问的对齐,可以减少内存块的访问冲突,提高内存访问的效率和吞吐量.例如,在 NTT 过程中,将寄存器中的 NTT 计算结果直接存回全局内存的操作优化为:先将 NTT 计算结果存至共享内存中,再将结果由共享内存对齐复制到全局内存中.优化方法的速度会快于直接由寄存器未对齐复制到全局内存中.

5　性能分析比较

在本节中,基于 GPU 平台上对本方案 CTRU768 和 CNTR768 实现进行性能分析测试,分析本方案在资源利用方面的各项指标,验证本方案设计的合理性. 此外,将本 GPU 实现方案与其他平台的实现进行对比分析,包括 C、AVX 实现等. 此外,还与 LWE 路线代表方案 Kyber 和其他 NTRU 格基方案的最新实现进行比较. 最后,对实现方案的比较结果进行深入分析与讨论.

5.1　测试环境

测试实验的编译和执行均在 Ubuntu－22.04 进行,使用 g＋＋ 11.4.0 编译 C/C＋＋代码,使用 CUDA 12.0 编译 GPU 实现. 硬件测试 CPU 平台为 2.50 GHz 的十二核 Intel(R) Core(TM) i5－12400F,GPU 平台为 NVIDIA GeForce RTX 4090. RTX 4090 是桌面工作站中常见的消费级 GPU. 测试策略为批处理 1000 个任务,每个任务处理使用 96 个线程来完成计算.

5.2　CTRU/CNTR 实现设计分析

在本节中,我们展示实现中主要模块的测试数据,并从吞吐量、内存占用等方面进行分析. 设置批处理任务 10000 个,对 CTRU768 及 CNTR768 中 PKE. KeyGen、PKE. Enc 和 PKE. Dec 三个主要核函数所占用的 GPU 资源进行测试. 具体评测结果如表 3 所示.

表 3　CTRU/CTRU 的 GPU 实现主要操作的资源利用

算法	操作	执行时间(μs)	吞吐量(%)		占用率(%)		内存使用	
			计算	内存	理论	实际	寄存器	SMEM(byte)
CTRU768	PKE. KeyGen	921.60	50.02	85.94	50	48.57	78	3168
	PKE. Enc	877.63	25.09	93.59	100	98.38	40	1632
	PKE. Dec	901.78	25.78	93.44	75	74.13	53	4704
CNTR768	PKE. KeyGen	974.91	49.65	83.84	50	49.43	78	3168
	PKE. Enc	837.25	24.98	92.83	100	98.52	40	1632
	PKE. Dec	861.80	25.47	92.93	75	74.15	53	4704

如表 3 所示,在主要计算单元的执行时间方面,完成 10000 次任务的计算所需的时间均在 800～1000 微秒,每个任务处理的均摊时间不超过 0.1 微秒. 在吞吐量方面,内存吞吐量均高于 80%,属于高吞吐实现. 在 PKE. KeyGen、PKE. Enc 和 PKE. Dec 三个核心核函数中,内存吞吐高于计算吞吐. 这表明方案数据量较多,但计算复杂度较低,符合 CTRU/CNTR 算法计算简单,安全性较高的特点. 同时,GPU 本身为高效的计算平台,运算速度较快. 在占用率及内存使用方面,PKE. Enc 的寄存器与 SMEM 占用率最为理想,理论与实际占用率基本可达到 100%. 然而,PKE. KeyGen 和 PKE. Dec 处理的数据量较大,导致使用的寄存器数量较多,进而降低了其占用率. 在综合考虑计算性能、访存速度等因素后,本方案的设计是合理且高效的.

5.3　GPU 实现对比分析

在本小节中,对本方案与其他多平台上的实现方案进行对比分析,选取密钥生成、密钥封装和密钥解封装三个阶段进行测试. 在 CPU 上测试 C 实现和 AVX2 实现的吞吐,在 GPU 上测试本实现方案. 度量单位为每秒完成的操作数(OP/s). 其中,GPU 实现的提升效率是相对于 C 的参考实现[13]而言的,具体测试数据如表 4 所示.

表 4　CTRU/CTRU 的多平台实现对比

算法	操作	CPU		GPU
		C 参考实现	AVX2	RTX 4090
CTRU768	密钥生成	34810	306666	11709601(336×)
	密钥封装	53021	250000	9267840(174×)
	密钥解封装	24517	80419	3154574(128×)
CNTR768	密钥生成	33930	242105	11173184(329×)
	密钥封装	55286	273809	9718172(175×)
	密钥解封装	24048	85501	3222687(134×)

如表 4 所示,基于 RTX 4090 平台,本实现中的 CTRU768 方案每秒钟分别可以进行密钥生成 1170 万次,密钥封装 926 万次,密钥解封装 315 万次. 与 CTRU768 的 C 语言参考实现相比,密钥生成、密钥封装和密钥解封装的速度分别提升 336 倍、174 倍和 128 倍. CNTR768 每秒钟分别可以进行密钥生成 1117 万次,密钥封装 971 万次,密钥解封装 322 万次,与 CNTR768 的 C 语言参考实现相比,速度分别提升 329 倍、175 倍和 128 倍. 相对于 C 语言参考实现的提升,主要归因于 GPU 平台强大的并行计算能力. GPU 能够批处理多个任务,并且在每个任务的执行过程中可以进行更深层次的并行计算. CPU 方案适用于广泛场景,在小规模任务和通用计算方面表现较好. 而本方案则更适合处理大批量任

务的场景，例如云计算、服务器等，以填补 CPU 实现在面对多任务场景处理时的不足. 总体而言，本方案的 CTRU768 及 CNTR768 的 GPU 实现相对于 C 的基准实现，在密钥封装和解封装均有百倍级的效率提升，展现出优异的性能表现.

此外，将本方案与 Kyber、NTRU 格基方案等 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）. 其中，Kyber 是 NIST 拟标准化的后量子密码算法，也是 LWE 路线的代表性格基密钥封装方案. 因此，基于 RTX 4090 平台，将本方案与 Kyber 的 GPU 开源方案[45]进行性能对比. 同时，还将基于不同 GPU 平台的 Kyber 和 NTRU 格基等方案的最新 GPU 加速实现进行吞吐量的对比，并对测试结果分析与比较. 具体的测试数据如表 5 所示.

表 5 CTRU/CNTR 和其他 KEM 的 GPU 实现性能对比

方案	测试平台	吞吐量(kOP/s)		
		密钥生成	密钥封装	密钥解封装
CTRU768	RTX 4090	11709	9267	3154
CNTR768		11173	9718	3222
Kyber768[45]		1030	976	616
Kyber768[28]	RTX3080	2036	1880	1501
NTRUEncrypt[25]	GTX1080	—	—	508
NTRU-HRSS[46]	RTX 2080	—	105	114

由表 5 可以看出，基于 RTX 4090 平台，与开源 Kyber768[45]实现相比，GPU 方案的吞吐量在密钥生成阶段，CTRU768 提升 11.36 倍，CNTR768 提升 10.84 倍；密钥封装阶段，CTRU768 提升 9.49 倍，CNTR768 提升 9.95 倍；密钥解封装阶段，CTRU768 提升 5.11 倍，CNTR768 提升 5.22 倍. 工作[28]、[25]和[46]为闭源实现，性能结果引用自其文章中的测试数据. 对于最新闭源的 Kyber GPU 方案[28]，我们选取与本方案线程设计架构相似的(136,128)线程维度的测试结果. 文献[25][46]为其他 NTRU 格基方案的最新实现，文献[25]为基于 GPU 平台实现的高吞吐量 NTRU 加密方案，文献[46]提出 GPU 加速 NTRU-HRSS 的方法，并展示在需要同时处理大规模数据的服务器环境中的效率. 在 GPU 平台上，本方案实现高吞吐的主要原因在于本方案注重并行效率，并进行高效安全的内存管理，对 CTRU/CNTR 各算子进行详细的设计和优化，包括循环展开、合并访存和共享内存访问优化等技术，使得有着优于其他方案的性能表现.

经测试，本方案的 CTRU 和 CNTR 的 GPU 实现，相对于 C 参考方案实现均有百倍级的吞吐量提升，相对于 AVX2 实现有着数十倍的提升，相对于 Kyber768[45]实现也有着 5-11 倍的提升.

6 总 结

当前，随着 GPU 架构的不断演进和优化技术的不断提升，密码方案的 GPU 高性能实现领域正处于快速发展阶段. 然而，仍然存在一些挑战，如内存带宽瓶颈和任务并行设计等方面的限制. 本文给出了针对 NTRU 格基密钥封装方案 CTRU/CNTR 的首个 GPU 高性能实现. 针对内存带宽瓶颈，我们设计填充内存、合并访存和内存池等加速方法，针对任务并行处理，我们通过算子加速实现、解密并行优化和提高资源占用率等解决方案，克服内存访问和并行计算等现有技术的局限性.

实验结果表明，本方案与 CTRU/CNTR 的 C 参考实现的效率相比，密钥生成阶段提升 329-336 倍，密钥封装阶段提升 174-175 倍，密钥解封装阶段提升 128-134 倍. 与 AVX2 实现相比有着数十倍的提升. 此外，本方案与 NIST 拟标准化算法（开源 Kyber768[45]）的 GPU 加速方案相比也有着显著的性能优势. 在资源占用、存储管理、吞吐表现方面，本方案均有着优异的性能表现，在大规模任务处理场景下具有较大的应用潜力，这也为后量子密码领域的高性能实现相关研究提供了新的方法和性能标杆.

参 考 文 献

[1] Bernstein D J, Lange T. Post-quantum cryptography. Nature, 2017, 549(7671): 188-194

[2] NIST. PQC Standardization Process: Announcing four candidates to be standardized, plus fourth round candidates. https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4

[3] Avanzi R, Bos J, Ducas L, et al. CRYSTALS-Kyber algorithm specifications and supporting documentation (version 3.02). Submission to round 3 of the NIST post-quantum project, 2021. https://pq-crystals.org/

[4] Shi B, Léo D, Eike K, et al. Crystals-dilithium: Algorithm specifications and supporting documentation. Submission to round 3 of the NIST post-quantum project, 2020. https://pq-crystals.org/

[5] 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/

[6] Bernstein D J, Hülsing A, Kölbl S, et al. The SPHINCS+ signature framework//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. London, UK, 2019: 2129-2146

[7] Regev O. On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM, 2009, 56(6): 1-40

[8] Lyubashevsky V, Peikert C, Regev O. On ideal lattices and learning with errors over rings. Journal of the ACM, 2013, 60(6): 1-35

[9] Langlois A, Stehlé D. Worst-case to average-case reductions for module lattices. Designs, Codes and Cryptography, 2015, 75(3): 565-599

[10] Hoffstein J, Pipher J, Silverman J H. NTRU: A ring-based public key cryptosystem//Proceedings of the International Algorithmic Number theory Symposium. Berlin, Germany, 1998: 267-288

[11] Chen C, Danba O, Hoffstein J, et al. NTRU algorithm specifications and supporting documentation. Submission to round 3 of the NIST post-quantum project, 2020. https://ntru.org/

[12] Bernstein D J, Brumley B B, Chen M S, et al. NTRU Prime: round 3. Submission to round 3 of the NIST post-quantum project, 2020. https://ntruprime.cr.yp.to/

[13] Liang Z, Fang B, Zheng J, et al. Compact and efficient KEMs over NTRU lattices. Computer Standards & Interfaces, 2024, 89: 103828

[14] Cryptography Standardization Technical Committee (CSTC). 2023 annual encryption industry standard formulation and revision tasks (commercial encryption field). Beijing: CSTC, 2023. (in Chinese)
(密码行业标准化技术委员会. 2023 年度密码行业标准制修订任务(商用密码领域). 北京: 密码行业标准化技术委员会, 2023)

[15] Lee W K, Zhao R K, Steinfeld R, et al. High throughput lattice-based signatures on gpus: Comparing falcon and mitaka. IEEE Transactions on Parallel and Distributed Systems, 2024, 35(4): 675-692

[16] Gupta N, Jati A, Chauhan A K, et al. Pqc acceleration using gpus: Frodokem, newhope, and kyber. IEEE Transactions on Parallel and Distributed Systems, 2020, 32(3): 575-586

[17] Shen S, Yang H, Dai W, et al. High-throughput gpu implementation of dilithium post-quantum digital signature. arXiv preprint arXiv:2211.12265, 2022. https://arxiv.org/abs/2211.12265

[18] Lee W K, Seo H, Zhang Z, et al. Tensorcrypto: High throughput acceleration of lattice-based cryptography using tensor core on gpu. IEEE Access, 2022, 10: 20616-20632

[19] Lee W K, Seo H, Hwang S O, et al. DPCrypto: Acceleration of post-quantum cryptography using dot-product instructions on GPUs. IEEE Transactions on Circuits and Systems I: Regular Papers, 2022, 69(9): 3591-3604

[20] Lee W K, Hwang S O. High throughput implementation of post-quantum key encapsulation and decapsulation on GPU for internet of things applications. IEEE Transactions on Services Computing, 2021, 15(6): 3275-3288

[21] Kamal A A, Youssef A M. Enhanced implementation of the NTRUEncrypt algorithm using graphics cards//2010 First International Conference On Parallel, Distributed and Grid Computing (PDGC 2010). Solan (HP), India, 2010: 168-174

[22] Hermans J, Vercauteren F, Preneel B. Speed records for NTRU//Proceedings of the Cryptographers' Track at the RSA Conference. San Francisco, USA, 2010: 73-88

[23] Bai T, Davis S, Li J, et al. Accelerating NTRU encryption with graphics processing units. International Journal of Networked and Distributed Computing, 2014, 2(4): 250-258

[24] Dai W, Sunar B, Schanck J, et al. NTRU modular lattice signature scheme on CUDA GPUs//Proceedings of the 2016 International Conference on High Performance Computing & Simulation (HPCS). Innsbruck, Austria, 2016: 501-508

[25] Akleylek S, Goi B M, Yap W S, et al. Fast NTRU encryption in GPU for secure IoP communication in post-quantum era//Proceedings 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). Guangzhou, China, 2018: 1923-1928

[26] Wan L, Zheng F, Lin J. TESLAC: Accelerating lattice-based cryptography with AI accelerator//Proceedings of the International Conference on Security and Privacy in Communication Systems. Canterbury, UK, 2021: 249-269

[27] Sun S, Zhang R, Ma H. Efficient parallelism of post-quantum signature scheme SPHINCS. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(11): 2542-2555

[28] Wan L, Zheng F, Fan G, et al. A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator//Proceedings of the European Symposium on Research in Computer Security. Copenhagen, Denmark, 2022: 514-534

[29] Fujisaki E, Okamoto T. Secure integration of asymmetric and symmetric encryption schemes//Proceedings of the Annual international cryptology conference. Santa Barbara, California, USA, 1999: 537-554

[30] Duman J, Hövelmanns K, Kiltz E, et al. Faster lattice-based KEMs via a generic Fujisaki-Okamoto transform using prefix hashing//Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. Virtual, 2021: 2722-2737

[31] Seiler G. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, 2018. https://eprint.iacr.org/2018/039

[32] Montgomery P L. Modular multiplication without trial division. Mathematics of Computation, 1985, 44(170): 519-521

[33] Barrett P. Implementing the rivest shamir and adleman pub-

lic key encryption algorithm on a standard digital signal processor//Proceedings of the Conference on the Theory and Application of Cryptographic Techniques. 1986: 311-323

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

[35] NVIDIA Corporation. CUDA C++ Programming Guide. NVIDIA Developer. 2023. https://docs.nvidia.com/cuda/cxx-compiler/index.html

[36] Lefohn A. Gpu memory model overview//Proceedings of the ACM SIGGRAPH 2005 Courses (SIGGRAPH'05). Los Angeles, USA, 2005: 127-es

[37] NVIDIA Corporation. Parallel thread execution ISA. NVIDIA Developer. 2023. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[38] Güneysu T, Oder T, Pöppelmann T, et al. Software speed records for lattice-based signatures//Post-Quantum Cryptography: 5th International Workshop. Limoges, France, 2013: 67-82

[39] Wu H, Diamos G, Wang J, et al. Optimizing data warehousing applications for GPUs using kernel fusion/fission//Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum. Shanghai, China, 2012: 2433-2442

[40] Wang G, Lin Y S, Yi W. Kernel fusion: An effective method for better power efficiency on multithreaded GPU//Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing. Hangzhou, China, 2010: 344-350

[41] Chung C M M, Hwang V, Kannwischer M J, et al. NTT multiplication for NTT-unfriendly rings: New speed records for Saber and NTRU on Cortex-M4 and AVX2. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021: 159-188

[42] Abdulrahman A, Chen J P, Chen Y J, et al. Multi-moduli NTTs for saber on Cortex-M3 and Cortex-M4. Cryptology ePrint Archive, 2021. https://eprint.iacr.org/2021/995

[43] Gelado I, Garland M. Throughput-oriented GPU memory allocation//Proceedings of the 24th symposium on principles and practice of parallel programming. Washington, USA, 2019: 27-37

[44] Chatterjee N, O'Connor M, Lee D, et al. Architecting an energy-efficient dram system for gpus//Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). Austin, USA, 2017: 73-84

[45] Ono T, Bian S, Sato T. Automatic parallelism tuning for module learning with errors based post-quantum key exchanges on GPUs//Proceedings of the 2021 IEEE International Symposium on Circuits and Systems (ISCAS). Daegu, Republic of Korea, 2021: 1-5

[46] Seong H, Kim Y, Yeom Y, et al. 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

LI Wen-Qian, master candidate. Her main research interests include post-quantum cryptography and cryptographic engineering.

SHEN Shi-Yu, Ph. D. candidate. Her main research interests include post-quantum cryptography, homomorphic encryption, and cryptographic engineering.

ZHAO Yun-Lei, Ph. D., distinguished professor at Fudan University. His main research interests include post-quantum cryptography, cryptographic protocols, and theory of computing.

Background

This paper focuses on the efficient implementation of the lattice-based CTRU/CNTR KEM based on GPU platform. This problem belongs to hardware optimization in cryptographic engineering. The implementation technology of cryptographic algorithms on software platforms such as AVX2, ARM Cortex-M4 is relatively mature, but there is still room for innovation on GPU platforms. The hardware implementation of lattice-based cryptography scheme is very important for prompting the application of post-quantum cryptography algorithms. This paper presents a high-speed GPU design of CTRU/CNTR, the main contributions include: (1) designing a reasonable size and number of thread blocks to achieve the maximum number of concurrent threads on the GPU; (2) adopting optimization techniques such as layer fusion, loop unrolling and lazy reduction to speed up polynomial multiplication which uses mixed-radix NTT; (3) coalescing memory and solving bank conflicts to reduce memory access conflicts and achieve efficient memory access; (4) using kernel fusion to avoid repeated memory access and calculation operations; (5) using PTX assembly instructions to improve module reduction by reducing the number of required instructions; (6) reasonably occupying GPU resources and adopting memory pool mechanism to realize fast memory access and efficient processing of concurrent multi-tasks. The previous research directions of our research group include software and hardware optimization of Aigis, Kyber, Dilithium and so on.