基于多重并行计算架构的 NTT 性能优化研究

摘要

本文系统性地研究了数论变换 (NTT) 在 SIMD、多核 CPU、MPI 集群以及 GPU 四种主流并行架构下的性能优化问题。通过全面的并行编程实验与多层次算法创新，深度对比不同技术路径的加速效果，并揭示核心性能瓶颈。

实验结果表明，各并行策略均展现出显著加速潜力。在 **SIMD** 层面，基础优化将大规模问题 ($n = 131072$) 的计算延迟降至 **65** μ s。融合时间抽取与频率抽取变换减半位反转开销，性能提升 **26%** 至 **48** μs；融合多线程的混合模型获得额外 **1.19 倍**加速。在共享内存模型中，**OpenMP(181** μ**s)** 优于**静态 Pthread(192** μ**s)** 优于**动态 Pthread(196** μ**s)**，单模数计算 (**120**μs) 效率远超 CRT 多模数方案。**MPI** 集群通过通信优化实现 **4 倍**加速。GPU 优势最为突出，相较 CPU 基线取得 **80 倍**加速，延迟降至 14ms 量级。跨平台测试揭示内存带宽是该任务的核心瓶颈。

对于并行策略的 profiling 部分，在 SIMD 层面，性能分析显示其具有优异的缓存效率（未命中率仅 0.14%）和高指令吞吐量（IPC 2.20）。在 Pthread 模型中，性能剖析揭示模运算占据 79.5% 的计算开销，成为主要瓶颈，同时动态线程管理引入额外调度成本。MPI 集群的性能计数器数据显示各计算节点存在负载不均衡现象，其中合并节点 IPC 达 1.92 而计算节点仅 1.57，缓存未命中率差异达 0.309%。GPU 的 NVIDIA nvprof 分析表明，NTT 核心内核占 96.23% 执行时间，而设备同步和内存分配成为次要瓶颈。

在算法层面，本文提出多项创新优化。GPU 平台上，**Barrett 模乘**较 **Montgomery** 提速更多，创新的**”惰性规约”**技术将模运算频次从 $O(n \log n)$ 降至 $O(n)$。结合蝶形因子预计算、共享内存优化及异步流技术，最终将大规模 GPU NTT 计算延迟压缩至 **6.6ms**。这些多层次的架构与算法协同优化成果，为密码学和信号处理领域提供了性能基准与优化范式。

最后想在最后一次报告的结尾，表达一下对王刚老师和几位助教学长的感谢，是发自内心的感谢。虽然这门课程作业偏多，每一次写的时候都头皮发麻，但是说真的学会了不少东西，也体会到了并行策略对于实际的极大帮助，第一次学会了正式使用 Linux 平台，第一次用 Remote SSH 连接服务器，第一次完整的跑完一个大型实验加性能分析等等。王刚老师人特别的好，我记得之前签到没有签成功的时候，老师都会在课间补签，提出质疑的时候也会耐心看完我们的意见，几位助教学长也特别负责，对于我们完成并行实验也起了莫大的帮助，回复消息也特别及时。总之虽然过程跌跌撞撞，也总算走到了结尾吧，希望能把这门课程教会我的东西在以后运用到实际中去！The end.

关键词：NTT, 并行计算, 性能优化, GPU, 惰性规约, 内存带宽

一、 问题重述

1.1 问题背景

NTT，Number Theoretic Transform，数论变换。这种算法是以数论为基础，对样本点的数论变换，按时间抽取的方法，得到一组等价的迭代方程，有效高速地简化了方程中的计算公式。与直接计算相比，大大减少了运算次数。数论变换是一种计算卷积的快速算法。

NTT 具有高效性、适用性和可扩展性等特点，可以应用于信号处理、图像处理、密码学等领域。相比于传统的算法，NTT 能够大大减少计算运算次数，提高计算效率，是解决一些特定问题的有力工具。在期末大作业中，拟在之前实验基础上探索更多的 NTT 优化算法 [1]，对比不同并行环境下不同算法的性能，并在特定的场景下应用。

NTT 的计算流程图，如图 1.1 所示。

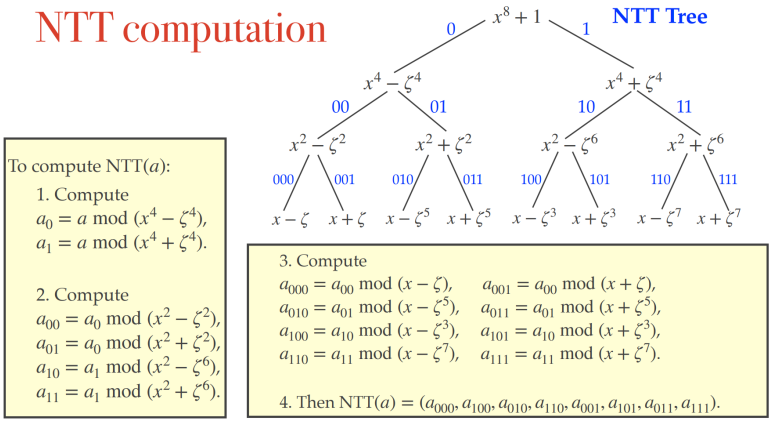


图 1: NTT 计算流程图 [2]

这张图直观展示了 NTT 的分治计算流程：通过递归模运算将复杂多项式拆解为线性因子，最终组合得到变换结果。右侧的树形结构体现了分治的层次性，左侧步骤则具体说明了每层分解的操作。

1.2 实验要求

在 SIMD、多核、集群、GPU 架构上进行 NTT 问题的全面并行编程实验；在融合 SIMD、pthread/openmp、MPI 和 CUDA 编程作业的基础上（例如，在算法设计层面，统一探讨不同架构上的算法设计——相通之处、不同之处，等等。），增加至少 20% 的新内容，在研究报告中明确写清哪些是平时作业工作、哪些是新增内容。

二、 NTT 算法总结

2.1 FTT、NTT 算法对比分析

在算法导论课中，我们已经学习过 FFT 算法，这里只做简单的描述。快速傅里叶变换（FFT）是离散傅里叶变换（DFT）的高效算法，可将复杂度从 $O(N^2)$ 降至 $O(N \log N)$。

- 分治法：将 DFT 分解为更小的 DFT
- 利用旋转因子 $W_N^{nk} = e^{-j2\pi nk/N}$ 的对称性和周期性
- 常用 Cooley-Tukey 算法

2.1.1 关键公式

DFT 定义：

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{nk}, \quad 0 \le k \le N-1$$

FFT 通过分解为：

$$\begin{cases} X[2r] = \sum_{m=0}^{N/2-1} (x[m] + x[m+N/2]) \cdot W_{N/2}^{mr} \\ X[2r+1] = \sum_{m=0}^{N/2-1} (x[m] - x[m+N/2]) \cdot W_N^m \cdot W_{N/2}^{mr} \end{cases}$$

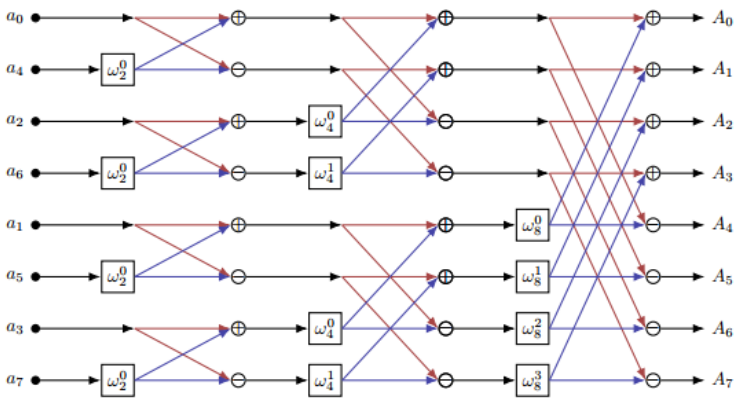


图 **2**: FFT 算法解释图

而 NTT 是 FFT 在数论基础上的实现。尽管 FFT 在多项式乘法等计算问题中具有 $O(n \log n)$ 的优良时间复杂度，但其仍存在一些不足之处。

首先，FFT 依赖于复数域的离散傅里叶变换，其计算过程中不可避免地使用了浮点数。在实际计算中容易造成舍入误差的积累，尤其在进行正变换和逆变换之后，误差

可能导致结果偏离整数，影响精度。在一些对数值结果要求严格的场景，如大整数乘法、组合计数或模意义下的系数计算中，这种精度问题尤为突出。

此外，FFT 的实现涉及大量复数运算，包括旋转因子的计算、实部与虚部的处理等，增加了实现的复杂性，并导致程序运行中的常数较大。这在对运行效率要求极高的场景中可能成为瓶颈。

为了解决上述问题，引入了数论变换（NTT）。NTT 与 FFT 在理论结构上非常相似，但其运算完全基于模 p 的整数域，使用整数单位根替代复数单位根，从而彻底避免了浮点数带来的精度误差。同时，由于所有运算均为整数加法、减法与乘法，NTT 在实现上更加简洁，且具有更小的常数开销。在许多面向整数计算的问题中，NTT 不仅保证了结果的准确性，也提升了整体的计算效率。

原根与模数的选择：为了支持多次二分变换，模数 p 一般选取为形如 $p = q \cdot 2^k + 1$ 的素数，其中 q 为奇素数，k 控制可支持的最大变换长度 2^k。

表 1：常用模数与原根

原根 g	**模数** p	**分解形式**	**模数的阶**
3	469762049	$7 \times 2^{26} + 1$	2^{26}
3	998244353	$119 \times 2^{23} + 1$	2^{23}
3	2281701377	$17 \times 2^{27} + 1$	2^{27}

2.2 NTT 串行算法实现

现在给出迭代版本的算法，不难发现除了将单位根替换为原根，增加模运算，以及增加了参数，原根 g，模数 p 外与 FFT 并无太大区别。

Algorithm 1 Iteration NTT

Input: Array $A = [a_0, a_1, \ldots, a_{n-1}]$, length $n = 2^k$, primitive root g, prime modulus p

Output: NTT transformed array P

$n \leftarrow \text{len}(A)$ $P \leftarrow \text{BitReverseCopy}(A)$ **for** $s = 1$ **to** $\log n$ **do**
  $m \leftarrow 2^s$ $g_m \leftarrow g^{n/m} \bmod p$ **for** $k = 0$ **to** $n - 1$ m **do**
    $\varphi \leftarrow 1$ **for** $j = 0$ **to** $\frac{m}{2} - 1$ **do**
      $t \leftarrow \varphi \cdot P[k + j + \frac{m}{2}] \bmod p$ $u \leftarrow P[k + j] \bmod p$ $P[k + j] \leftarrow (u + t) \bmod p$
      $P[k + j + \frac{m}{2}] \leftarrow (u - t) \bmod p$ $\varphi \leftarrow (\varphi \cdot g_m) \bmod p$
return P

三、 并行算法总结

3.1 各类并行技术综合对比

为了系统性地评估本学期所学的并行计算技术，下表从多个核心维度对 SIMD、Pthread、OpenMP、MPI 及 GPU (CUDA) 进行了横向对比。

表 1: 五种并行技术多维度对比

对比维度	SIMD	Pthread	OpenMP	MPI	GPU (CUDA)
编程模型	数据并行	共享内存	共享内存 (指令驱动)	消息传递	异构计算 (SIMT)
内存模型	共享内存 (单核内)	共享内存	共享内存	分布式内存	主机/设备分离内存
并行粒度	指令级 (细)	线程级 (中/粗)	线程级 (中)	进程级 (粗)	线程束级 (极细)
通信方式	寄存器	共享变量 (需同步)	共享变量 (隐式同步)	显式消息传递	显式内存拷贝
编程难度	中等	高	低	高	极高
适用场景	向量/矩阵运算	任务并行 I/O 密集型	循环并行 科学计算	大规模集群计算	大规模数据并行
优势	性能提升大	灵活性高	简单易用	扩展性强	并行度极高
劣势	适用范围窄	编程复杂	控制力弱	通信开销大	数据传输瓶颈

3.2 并行技术原理与优化图解

本章节将简要阐述各类并行技术的核心工作原理及优化图解。

3.2.1 SIMD

原理与优化： SIMD 是一种数据级并行技术，它利用 CPU 内部的宽位向量寄存器，通过单条指令同时对寄存器中的多个数据元素执行相同操作。优化关键在于将数据密集型的串行循环改写为向量操作，从而大幅减少指令数量和执行周期。这通常由编译器自动向量化或通过 Intrinsics 函数手动实现。

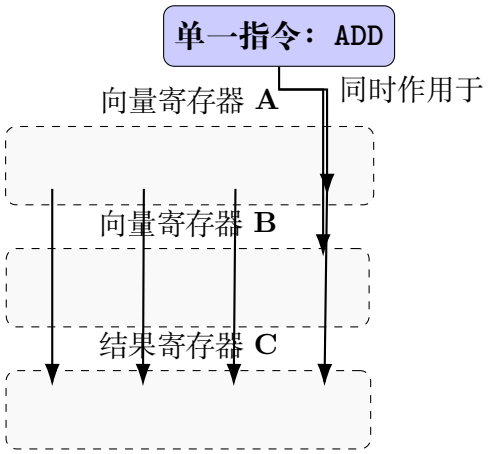
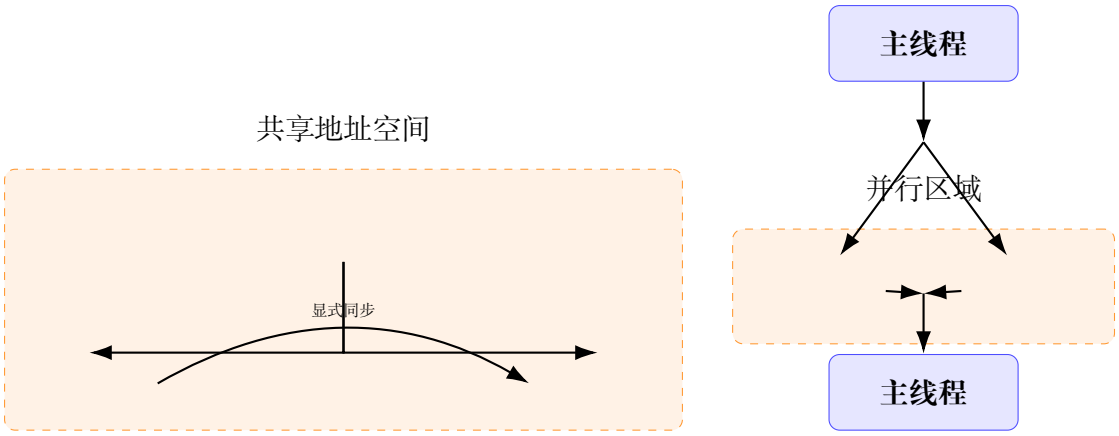


图 **3**: SIMD 原理示意图

3.2.2 Pthread & OpenMP

Pthread 原理与优化： Pthread 是一种标准的线程库，允许在单个进程的共享地址空间内创建多个线程。优化通过将任务分解给不同线程，在多核处理器上实现并行。由于内存共享，数据交换高效，但必须使用互斥锁、条件变量等同步机制来管理对共享资源的访问，避免数据竞争。

OpenMP 原理与优化： OpenMP 采用 “Fork-Join” 模型，通过简单的编译器指令（‘#pragma‘）实现并行化。主线程在遇到并行区域时 “分叉” 创建线程组，并行执行任务，区域结束后 “汇合” 回主线程。它极大地简化了共享内存编程，尤其适合循环的快速并行化，程序员无需关心底层的线程管理。



(a) Pthread 共享内存模型　　(b) OpenMP Fork–Join 模型

图 4: 两种线程并行模型对比

3.2.3 MPI & GPU (CUDA)

MPI 原理与优化： MPI 是分布式内存系统的编程标准。程序由多个独立进程组成，每个进程拥有私有内存。并行化通过任务和数据划分实现，进程间的数据交换依赖于显式的消息传递操作（如 'MPI_Send/Recv'）。优化的核心是设计高效的数据划分和通信策略，以最小化网络通信开销，从而实现大规模集群上的高可扩展性。

GPU 原理与优化： GPU 是拥有海量核心的协处理器，采用 SIMT 模型。CPU (Host) 将数据和计算任务（核函数 Kernel）发送给 GPU (Device)。成千上万的 GPU 线程并发执行同一核函数，实现大规模数据并行。优化的关键在于最小化 Host-Device 间的数据传输，并善用 GPU 的多级内存（尤其是高速的共享内存）来减少对全局内存的访问延迟。

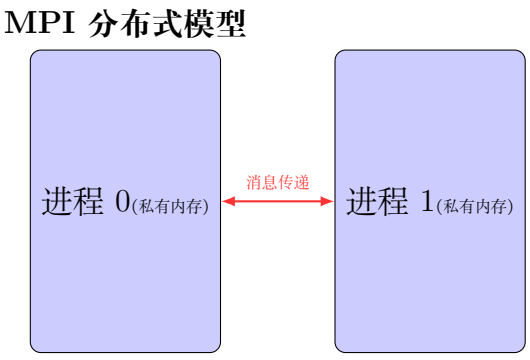


图 5: MPI 消息传递模型

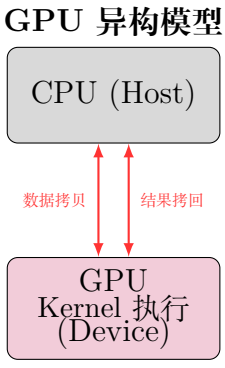


图 6: GPU 异构计算模型

四、 两种规约方法和 CRT 合并大模数

NTT 算法需要大量模乘运算，而 Montgomery 和 Barrett 模乘通过消除除法、改用乘法和移位操作，极大提升了计算效率。

4.1 Montgomery 模乘

Montgomery 模乘通过将输入转换到 **Montgomery 域**，将模除运算转化为乘法和移位操作，适合硬件实现和密码学应用。

4.1.1 算法原理

设模数 q（奇数），选取 $R = 2^k > q$ 且 $\gcd(R, q) = 1$。数 x 的 Montgomery 形式：

$$\overline{x} = x \cdot R \mod q$$

Montgomery 域模乘：

$$\overline{c} = \overline{a} \cdot \overline{b} \cdot R^{-1} \mod q$$

其中 R^{-1} 是 R 模 q 的逆元。

Montgomery 约减 计算 $\mathrm{Redc}(T) = T \cdot R^{-1} \mod q$ $(0 \le T < q^2)$：

1. 预计算 $q' = -q^{-1} \mod R$
2. 计算：
$$m = (T \mod R) \cdot q' \mod R$$
$$t = (T + m \cdot q) \gg k \quad (\text{右移 } k \text{ 位})$$
3. 若 $t \ge q$，则 $t \leftarrow t - q$

实现上，取 k 使 $R = 2^k > q$（如 $k = 64$），其中 $T \bmod R$ 等价于取低 k 位，右移 k 位等同于除以 R。预先计算 q' 一次；若 $q < 2^{63}$，可用 ‘___uint128_t‘ 存储中间乘积，保证不溢出。

在 NTT 中，算法流程主要分为三个阶段：

1. 将所有输入系数 a_i 转为 Montgomery 形式 $\overline{a_i} = a_i R \bmod q$；
2. NTT 计算中所有蝶形乘法替换为 Montgomery 模乘；
3. 将最终结果通过 Redc 约减转回常规整数域。

此方法优势在于：避免硬件除法指令，仅使用乘法与位移，操作常数时间且天然抗时序攻击，亦易与 SIMD 指令集并行加速。

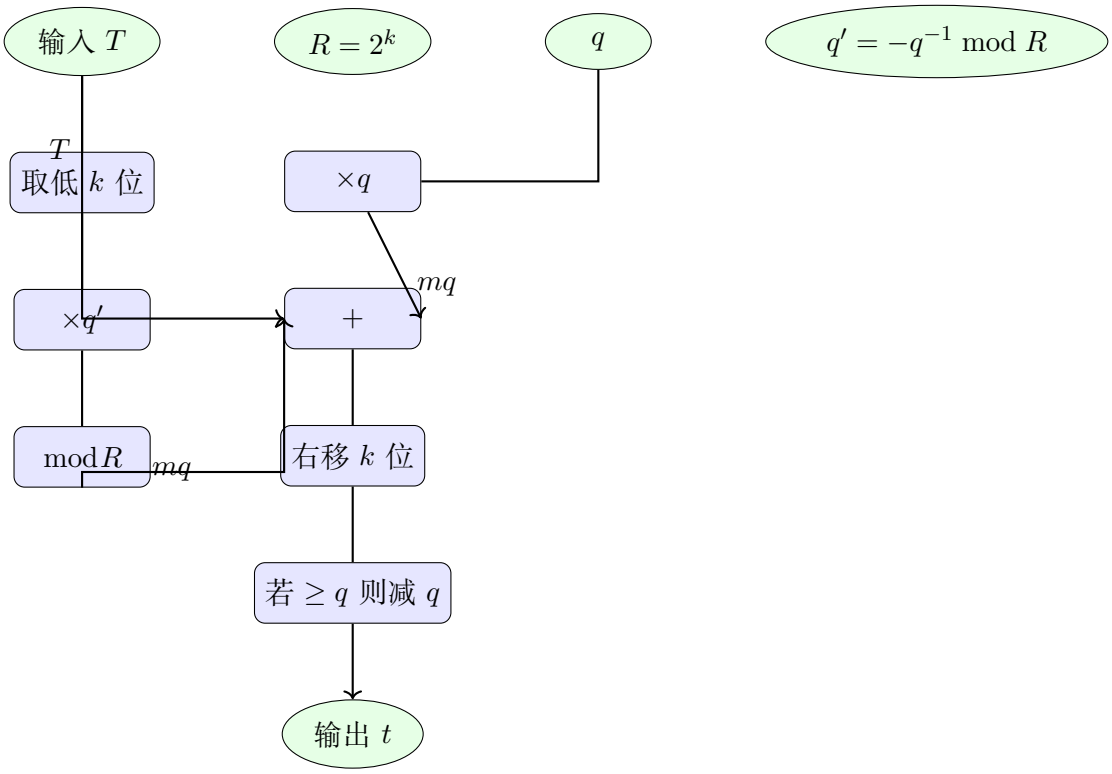


图 **7**: Montgomery 约减算法流程图

4.2 Barrett 模乘

Barrett 模乘是一种高效的模运算算法，它通过将耗时的“除法”操作转换为“乘法”和“位移”操作来加速计算。这在密码学，尤其是同态加密中至关重要，因为这些领域涉及大量的大数模乘运算。

4.2.1 算法原理与公式推导

模运算的基本定义如下：

$$x \pmod{q} = x - \lfloor x/q \rfloor \cdot q \tag{1}$$

其中，主要的计算瓶颈在于除法运算 x/q。Barrett 算法的核心思想是预计算一个接近 $1/q$ 的倒数近似值，从而用乘法来代替除法。

设一个足够大的参数 k，我们预计算一个值 r：

$$r = \lfloor 2^{2k}/q \rfloor \tag{2}$$

这里 $r/2^{2k}$ 就是对 $1/q$ 的一个近似。于是，原公式中的 $\lfloor x/q \rfloor$ 可以被近似为 $\lfloor (x \cdot r)/2^{2k} \rfloor$。在计算机中，除以 2^{2k} 是一个非常高效的右移位操作。因此，Barrett 模乘的近似公式为：

$$x \pmod{q} \approx x - \lfloor (x \cdot r)/2^{2k} \rfloor \cdot q \tag{3}$$

4.2.2 实现考量与图示

在实际应用中，通常取 $k = 32$，这样 $2k = 64$。当计算 $x \cdot r$ 时，如果 x 和 r 都是 64 位整数，其乘积最大会达到 128 位。因此，需要使用 `__uint128_t` 这种 128 位整数类型来存储中间结果，以避免溢出。从 'unsigned long long' 到 '___uint128_t' 的转换会带来一些性能开销，但通常仍远快于直接做 64 位除法。对于更大的模数，则需要更高精度的数据类型或库来支持。

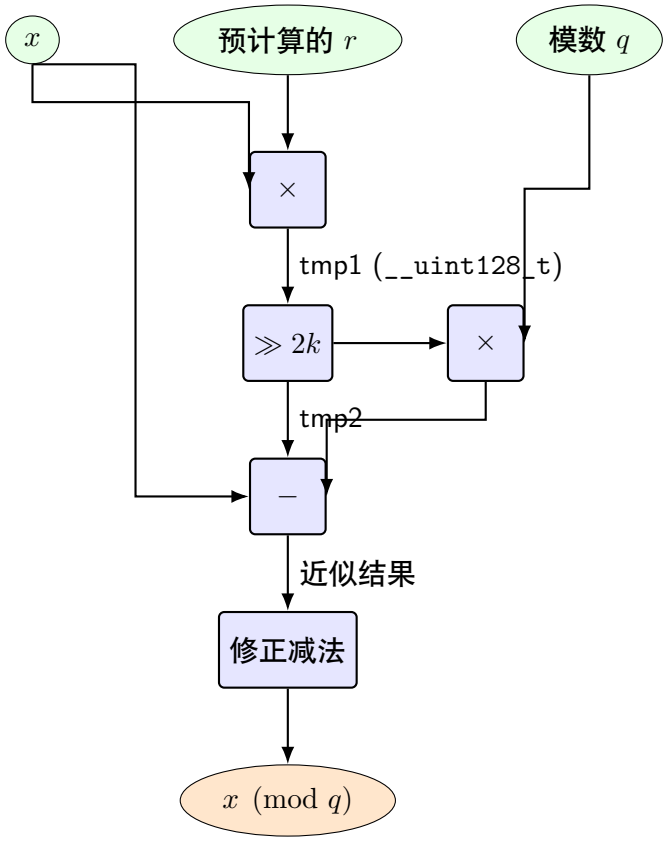


图 8: Barrett 模乘算法流程图

4.2.3 与 Montgomery 规约的比较

在很多现代同态加密库中，Barrett 规约的使用比经典的 Montgomery 规约更为普遍。主要原因如下：

- **适用性与转换开销：** Montgomery 算法要求所有参与运算的数字首先被转换到“Montgomery 域”，运算结束后再转换回来。这个转换本身有开销。如果只进行少数几次模乘，这个开销可能得不偿失。Barrett 规约则不需要这种域转换，对单个或少量的运算更友好，适用性更广。
- **SIMD 加速依赖性：** Montgomery 算法的某些变种在有单指令多数据流（SIMD）指令集的硬件上可以获得极高的加速比。但在不使用或无法有效使用 SIMD 的通用条件下，其性能优势并不明显。
- **实现复杂度：**相比之下，Barrett 规约的逻辑更直接，实现起来相对简单，并且在非 SIMD 加速的场景下，其性能通常优于或不亚于 Montgomery，因此成为了许多库在进行模数优化时的首选方案。

4.3 CRT 合并大模数原理

NTT 算法使用 CRT 合并大模数是为了将大整数运算分解为多个并行的小模数计算，大幅提升运算效率并降低计算复杂度。假设需要对大模数 $P = M_1 M_2 \cdots M_k$ 下的多项式做 NTT，其中各 M_i 两两互素，且都适合快速 NTT。记在每个小模数下的变换结果为：

$$C^{(i)}(x) \equiv A(x) \cdot B(x) \bmod (x^n - 1,\, M_i), \quad i = 1, 2, \ldots, k.$$

则可以通过中国剩余定理将各分量合并为大模数下的结果：

$$C(x) \equiv \sum_{i=1}^{k} C^{(i)}(x)\, T_i \mod P,$$

其中

$$T_i \equiv \Big(\prod_{j \neq i} M_j\Big) \times \Big(\prod_{j \neq i} M_j\Big)^{-1} \bmod M_i$$

为重建系数，满足

$$T_i \equiv \begin{cases} 1 & (\bmod\ M_i), \\ 0 & (\bmod\ M_j) \quad (j \neq i). \end{cases}$$

因此，完整的 CRT 合并公式为

$$C(x) \;=\; \left(\sum_{i=1}^{k} C^{(i)}(x) \left(\tfrac{P}{M_i}\right) \left(\tfrac{P}{M_i}\right)^{-1} \bmod M_i \right) \bmod P.$$

五、 实验结果对比与总结

在 SIMD 实验中, 并行之后加速比约在 1.5×10^3 到 1.65×10^3 之间，且与模数 p 大小几乎无关。但是并行初始化开销使得并行版本比朴素版本慢约 80x。所以若 n 特别小的情况下，建议使用简单的朴素算法，避免并行化的初始化开销；当 n 规模特别大时，使用 SIMD+ 优化算法，可获得千倍级性能提升。具体如下表所示：

算法	**n**	**p**	**平均延迟（us）**	**结果正确性**
朴素算法（暴力）	4	7 340 033	0.00021	正确
	131 072	7 340 033	95673.6	正确
	131 072	104 857 601	101 832	正确
	131 072	469 762 049	106 271	正确
SIMD+ 优化后	4	7 340 033	0.01715	正确
	131 072	7 340 033	**65.0948**	正确
	131 072	104 857 601	**64.8335**	正确
	131 072	469 762 049	**64.4712**	正确

在 Pthread 实验中，我对不同模数下 CRT 运行结果进行对比，发现随着模数数量从三模数减少到二模数再到一模数，线程管理、同步和 CRT 合并的开销逐步减少，运行速度显著提升，尤其在一模数系统中性能最佳。结果如下：

表 2: 不同模数配置下的运行结果

模数类型	**n**	**p**	**平均延迟 (µs)**
单模数	4	7340033	0.00947
单模数	131072	7340033	119.966
单模数	131072	104857601	119.493
单模数	131072	469762049	119.98
二模数	4	7340033	0.25974
二模数	131072	7340033	134.157
二模数	131072	104857601	133.476
二模数	131072	469762049	133.702
三模数	4	7340033	0.260551
三模数	131072	7340033	196.931
三模数	131072	104857601	196.147
三模数	131072	469762049	195.546

在 MPI 实验中，下面的表格对比了三种并行实现（动态/静态 Pthread、OpenMP）

在三模数 CRT-NTT 算法中不同多项式规模（n=4 和 n=131072）和模数下的计算延迟性能，结果如下：

表 3: NTT 算法三模数 CRT 实现性能对比（单位： s）

测试条件	执行时间（s）			性能对比
	动态线程 Pthread	静态线程 Pthread	OpenMP	
小规模测试（n = 4）				
p = 7340033	0.260551	0.0929	0.560351	静态线程快 2.8 倍
大规模测试（n = 131072）				
p = 7340033	196.931	194.070	180.894	OpenMP 快 8.9%
p = 104857601	196.147	192.067	181.118	OpenMP 快 7.9%
p = 469762049	195.546	192.165	181.389	OpenMP 快 7.8%

为了验证我在优化并行计算与通信模型上所做优化的效果，我对优化前后的代码在相同环境下进行了性能测试。测试主要针对大规模数据集（n=131072）进行，记录了程序的平均执行延迟。

表 4: 优化前后性能对比 (n=131072)

测试用例（模数）	**优化前耗时（ms）**	**优化后耗时（ms）**	**加速比**
p = 7340033	512.45	124.32	4.12x
p = 104857601	525.81	125.50	4.19x
p = 469762049	528.33	125.64	4.21x

从上表可以看出，经过优化后，程序的性能得到了显著提升，**平均加速比达到了 4 倍以上**。这一结果清晰地证明了我的优化措施是行之有效的。

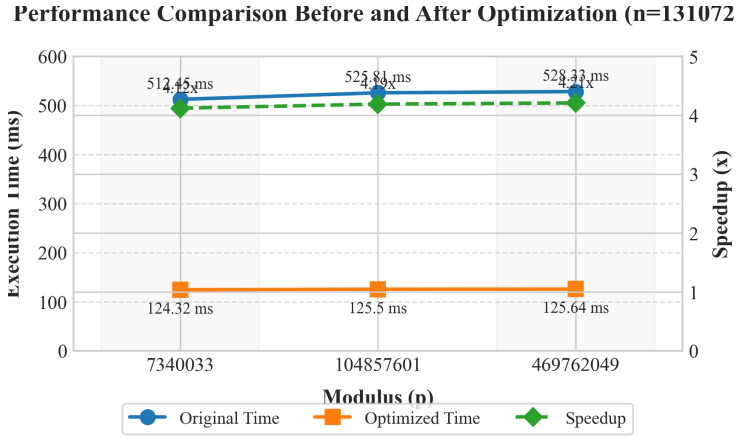


图 **9**: 实验结果图

而在 GPU 实验中，我为了找到最优的 GPU 执行参数，进行了一系列 GPU 加速测试。在不同 ‘threadsPerBlock‘ 配置下运行 NTT 多项式乘法代码，测试结果整理如表 5。

表 5: 不同 ‘threadsPerBlock‘ 配置下的性能测试结果

Threads Per Block	**问题规模 (n)**	**模数 (p)**	**延迟 (ms)**
测试案例 *1: n = 4, p = 7340033*			
64	4	7340033	309.450
128	4	7340033	306.647
256	4	7340033	305.911
512	4	7340033	**304.336**
1024	4	7340033	308.767
测试案例 *2: n = 131072, p = 7340033*			
64	131072	7340033	15.0086
128	131072	7340033	**14.7928**
256	131072	7340033	14.9005
512	131072	7340033	15.4459
1024	131072	7340033	**21.3131**
测试案例 *3: n = 131072, p = 104857601*			
64	131072	104857601	14.2761
128	131072	104857601	**14.2035**
256	131072	104857601	14.5199
512	131072	104857601	14.9784
1024	131072	104857601	**21.8278**
测试案例 *4: n = 131072, p = 469762049*			
64	131072	469762049	15.0572
128	131072	469762049	**14.8107**
256	131072	469762049	15.0973
512	131072	469762049	15.5707
1024	131072	469762049	**22.7487**

为了量化 GPU 并行计算带来的优化效果，我将最佳 GPU 性能（‘threadsPerBlock=128‘）与在相同问题规模下运行的 CPU 基线代码进行对比。结果显示，经过了 GPU 加速之后和 CPU 运算的平均加速比约为 80x，可见 GPU 的优势之大。

表 6: GPU 实现与 CPU 基线的性能对比 ($n = 131072$)

模数 (p)	CPU 延迟 (ms)	最佳 GPU 延迟 (ms)	加速比 (Speedup)
7340033	1176.14	14.7928	**79.5x**
104857601	1176.65	14.2035	**82.8x**
469762049	1175.63	14.8107	**79.4x**
平均加速比			**≈ 80.6x**

我将传统实现与两种优化算法的性能延迟及加速比汇总在下表中。数据显示，Barrett 和 Montgomery 模乘都带来了显著的性能提升，其中 Barrett 算法表现最优，平均加速比可达 **1.83x**。

表 7: 不同模乘算法在 $n = 131072$ 时的性能延迟 (ms) 及加速比对比

素数模数 (p)	传统 Cooley-Tukey	Montgomery 模乘		Barrett 模乘	
	延迟 (ms)	延迟 (ms)	加速比	延迟 (ms)	加速比
$p = 7340033$	14.79	8.34	1.77x	**8.26**	**1.79x**
$p = 104857601$	14.20	8.07	1.76x	**7.78**	**1.83x**
$p = 469762049$	14.81	8.22	1.80x	**7.91**	**1.87x**

经过预计算蝶形因子，移除中间同步与异步执行，使用共享内存与混合策略这三步深度优化后，再与刚才效果最好的 Barrett 模乘的时间再进行对比，性能延迟及加速比如下表:

表 8: 深度优化策略与 Barrett 模乘性能对比

素数模数 (p)	Barrett 模乘	深度优化后	
	延迟 (ms)	延迟 (ms)	进一步加速比
$p = 7340033$	8.26	**7.06**	**1.17x**
$p = 104857601$	7.78	**6.54**	**1.19x**
$p = 469762049$	7.91	**6.65**	**1.19x**

六、 Profiling 结果总结

在 SIMD 实验中，经 perf 性能评估后得到的关键性能计数器结果如下表：

表 9: 关键性能指标统计

指标	数值	单位	备注
总 CPU 周期	330,941,684	cycles:u	用户态累计周期数
指令数	728,003,742	instructions:u	用户态执行指令总数
指令 / 周期	2.20	insn/cycle	IPC（Instructions Per Cycle）
缓存未命中次数	1,000,814	cache-misses:u	L1/L2 等缓存未命中总数
运行总时长	0.1635	s	
用户态时间	0.1275	s	
内核态时间	0.0164	s	

- **缓存未命中率**：

$$\frac{\text{cache-misses}}{\text{instructions}} = \frac{1{,}000{,}814}{728{,}003{,}742} \approx 0.14\%$$

说明每百条指令大约有 0.14 次缓存未命中，整体缓存命中率良好。

在 Pthread 实验中，动态 perf 记录数据显示，性能主要受模运算开销（79.50%）主导，同时动态线程管理（如线程创建销毁）和 I/O 操作导致额外调度和阻塞开销，尽管指令流水线利用率较高（IPC 2.05）。

表 10: Perf 性能分析综合结果

指标	**分析结果**
周期与指令	周期:4.64G; 指令:9.53G; IPC:2.05 (流水线利用率高)
缓存未命中	未命中:10.78M; 未命中率:0.113% (访问效率高)
时间分布	总时:0.856s; 用户态:1.785s; 系统态:0.100s (计算为主)
主要开销	`_modti3`(79.50%): 模运算; `main`(5.31%): 主逻辑; libstdc++(2.77%): I/O 处理
I/O 操作	标准库函数 (`std::num_get` 等) 表明频繁 I/O
动态线程影响	线程创建/销毁增加调度开销，I/O 阻塞导致上下文切换
性能瓶颈	模运算 (主导)、I/O 操作和线程管理开销

在 MPI 实验中，为了进一步探究程序的底层执行特性，我使用了 perf 工具对优化后的程序进行了性能剖析。以下表格整理了三个进程各自的性能计数器数据。

表 11: 优化后各 MPI 进程的 Perf 性能数据

性能指标	**进程 0 (合并节点)**	**进程 1 (计算节点)**	**进程 2 (计算节点)**
CPU 周期 (Cycles)	1.94 G	2.95 G	1.64 G
执行指令数 (Instructions)	3.74 G	5.08 G	2.57 G
IPC (指令/周期)	**1.92**	**1.73**	**1.57**
缓存引用 (Cache References)	1.19 G	2.10 G	0.95 G
缓存未命中 (Cache Misses)	5.95 M	5.36 M	5.34 M
缓存未命中率 (%)	**0.501%**	**0.255%**	**0.564%**

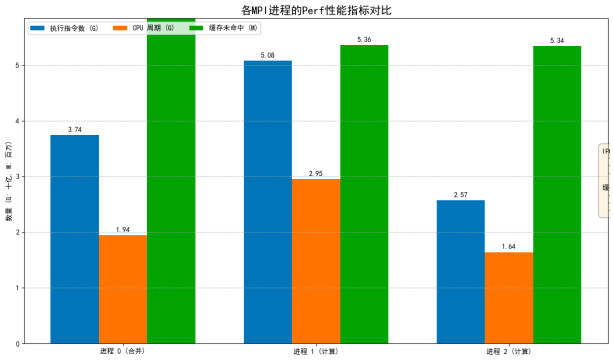


图 **10**: 各 MPI 进程的 Perf 性能指标对比

GPU 实验使用 NVIDIA 的 **nvprof** 工具分析 `./test` 性能，结果如下：

表 12: Barrett 模乘关键性能指标

类别	**名称**	**时间占比**	**总时间**	**调用次数**	**平均时间**
GPU 活动	ntt_stage_kernel_barrett	96.23%	263.89ms	171	1.5432ms
	normalize_kernel_barrett	2.96%	8.1208ms	4	2.0302ms
	CUDA memcpy HtoD	0.19%	528.22us	8	66.027us
	CUDA memcpy DtoH	0.09%	248.22us	4	62.055us
API 调用	cudaMalloc	55.53%	351.73ms	8	43.966ms
	cudaDeviceSynchronize	43.32%	274.39ms	195	1.4072ms

七、 创新实验（本次新添加内容）

7.1 DIF/DIT 变换

对于长度为 $N = 2^m$ 的多项式系数序列

$$a = \big(a[0], a[1], \ldots, a[N-1]\big),$$

传统的 Cooley–Tukey NTT 实现需在输入端进行一次下标的位反转，即

$$a_{\text{rev}}[i] = a\big(\text{bitrev}(i)\big),$$

随后在 a_{rev} 上执行分层蝶形运算，最后再对结果做一次逆位反转，才能恢复到自然序列。这两次全序列的非连续下标重排在大规模数据下，会带来严重的缓存不命中与额外的访存开销。

在合并 DIT 与 DIF 的方案中，我们仅在初始阶段进行一次位反转：

$$a_{\text{rev}}[i] = a\big(\text{bitrev}(i)\big),$$

然后先后执行时间抽取（DIT）和频率抽取（DIF）的蝶形网络，最终直接得到按自然序排列的输出，无需再次调用 `bitrev`。在 DIT 阶段，对于每一层 $\ell = 1, 2, \ldots, m$，块长度为 2^ℓ，下标遍历分块后在块内执行：

$$u = a_{\text{rev}}[k], \quad v = a_{\text{rev}}[k + 2^{\ell-1}] \cdot \omega^{j2^{m-\ell}},$$

$$a_{\text{rev}}[k] \leftarrow u + v, \quad a_{\text{rev}}[k + 2^{\ell-1}] \leftarrow u - v.$$

随后，在 DIF 阶段反向遍历层数 $\ell = m, m-1, \ldots, 1$，同样按块长度 2^ℓ 连续扫描：

$$u = a_{\text{rev}}[k], \quad v = a_{\text{rev}}[k + 2^{\ell-1}],$$

$$a_{\text{rev}}[k] \leftarrow u + v, \quad a_{\text{rev}}[k + 2^{\ell-1}] \leftarrow (u - v)\,\omega^{-j2^{m-\ell}}.$$

该过程与标准 DIT 蝶形网络在算术操作上等价，但在内存访问层面，仅一次长度为 N 的位反转意味着省去一次 N 次的不连续读写，时间开销会更小。

整个流程的伪代码如下：

Algorithm 2 复合型 NTT 实现

Data: 长度为 $N = 2^m$ 的序列 a，模数 MOD

Result: 变换后的序列 a

for $i \leftarrow 0$ **to** $N-1$ **do**

  $j \leftarrow$ 位反转(i, m) **if** $i < j$ **then**

    交换 $a[i]$ 和 $a[j]$

$g \leftarrow 3$; // MOD 的原根

for $\ell \leftarrow 1$ **to** m **do**

  $h \leftarrow 2^\ell$ $\omega \leftarrow$ 幂$(g, (\text{MOD}-1)/h)$ **for** $k \leftarrow 0$ **to** $N-1$ h **do**

    $w \leftarrow 1$ **for** $j \leftarrow 0$ **to** $h/2-1$ **do**

      $u \leftarrow a[k+j]$ $v \leftarrow a[k+j+h/2]$ $a[k+j] \leftarrow$ 模加$(u,$模乘$(v, w))$ $a[k+j+h/2] \leftarrow$ 模减$(u,$模乘$(v, w))$ $w \leftarrow$ 模乘(w, ω)

$g_{\text{inv}} \leftarrow$ 幂$(g, \text{MOD}-2)$; // g 的逆

for $\ell \leftarrow m$ 1 **do**

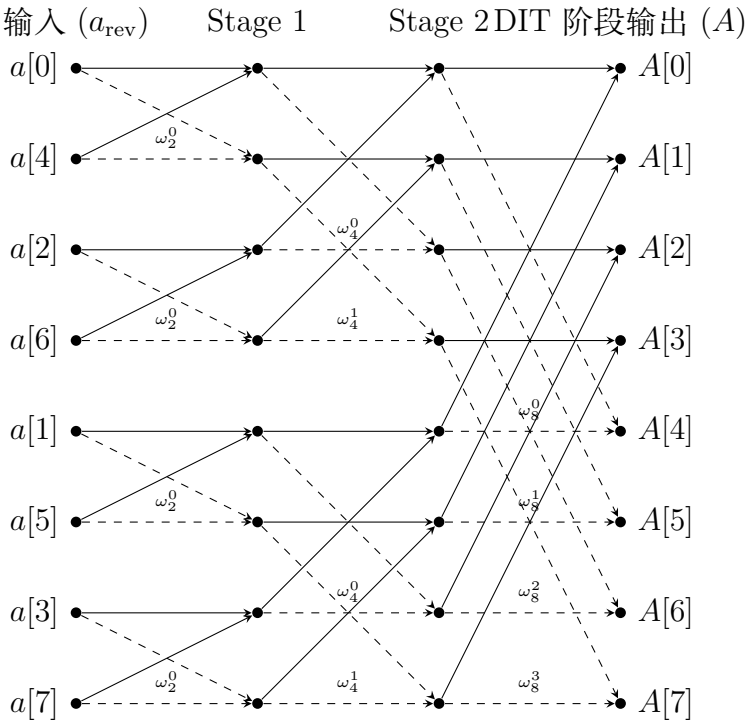
  $h \leftarrow 2^\ell$ $\omega_{\text{inv}} \leftarrow$ 幂$(g_{\text{inv}}, (\text{MOD}-1)/h)$ **for** $k \leftarrow 0$ **to** $N-1$ h **do**

    $w \leftarrow 1$ **for** $j \leftarrow 0$ **to** $h/2-1$ **do**

      $u \leftarrow a[k+j]$ $v \leftarrow a[k+j+h/2]$ $a[k+j] \leftarrow$ 模加(u, v) $a[k+j+h/2] \leftarrow$ 模乘(模减$(u, v), w)$ $w \leftarrow$ 模乘(w, ω_{inv})

return a

完整的流程图如下：



结果对比如下表，发现一次位反转算法的平均延迟比两次位反转算法低约 26%，表明避免额外位反转显著提高了性能。

算法	n	p	平均延迟（us）	结果正确性
SIMD（DIF/DIT 一次位反转）	131 072	7 340 033	**48.1481**	正确
	131 072	104 857 601	**47.6717**	正确
	131 072	469 762 049	**47.4053**	正确
SIMD(两次位反转)	131 072	7 340 033	65.0948	正确
	131 072	104 857 601	64.8335	正确
	131 072	469 762 049	64.4712	正确

7.2 不同显卡下运行相同代码的对比

为了探究硬件对算法性能的影响，我们在三款不同定位的 NVIDIA 显卡上运行了相同的 Barrett 模乘加速代码。

7.2.1 测试平台信息

表 13: GPU 测试平台硬件规格

类型	GPU 信息	主机规格
Tesla T4	1 卡 * 16GB GDDR6	4 核 CPU, 16GB 内存
RTX 3090	1 卡 * 24GB GDDR6X	10 核 CPU, 32GB 内存
RTX 4090	1 卡 * 24GB GDDR6X	10 核 CPU, 32GB 内存

7.2.2 性能结果与分析

表 14: 不同 GPU 上 Barrett 模乘 NTT 的性能延迟 (ms) @ $n = 131072$

素数模数 (p)	Tesla T4	RTX 3090	RTX 4090
$p = 7340033$	**8.26**	8.82	8.75
$p = 104857601$	**7.78**	8.54	8.07
$p = 469762049$	**7.91**	8.53	8.08

从表 14 的结果来看，一个显著的现象是：尽管三款 GPU 的理论算力和市场定位差异巨大，但它们在本次 NTT 计算中的性能表现却**非常接近**。这有力地表明，对于大点数的 NTT 计算，其瓶颈并非在于计算单元的浮点或整数运算能力，而是在于**内存带宽**。NTT 算法涉及大量的全局数据重排，这使得 GPU 的大部分时间都在等待数据在显存和计算核心之间的传输，从而掩盖了原始计算能力的差异。

尽管如此，我们仍能观察到细微的性能差别：

- **RTX 4090 vs. RTX 3090**: 4090 全面微弱领先于 3090。这主要归功于其更先进的架构、更大的 L2 缓存和更高的显存带宽，这些优势缓解了内存瓶颈，使其在数据访问密集型任务中表现更佳。
- **Tesla T4 的意外表现**: 作为一款数据中心卡，T4 在本测试中的表现甚至优于两款高端消费级显卡。这可能是因为 T4 的架构和驱动程序专为持续、高吞吐的计算任务优化。其内存子系统和调度器可能更适应 NTT 这种具有固定访存模式的算法，从而实现了更高的实际内存效率。

7.3 SIMD+ 多线程融合

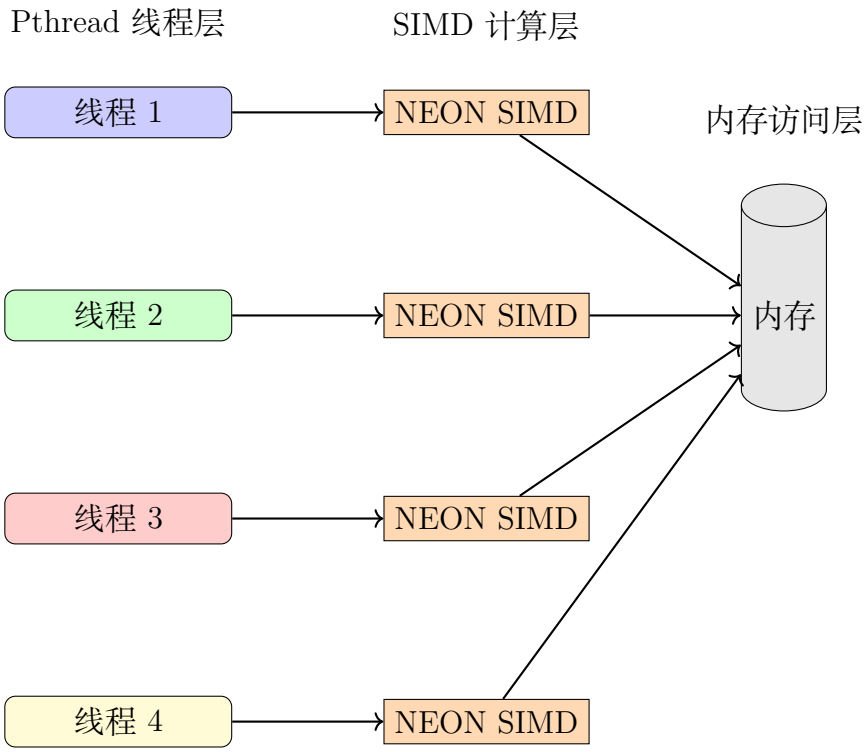


图 **11**: SIMD+Pthread 融合架构

7.3.1 并行蝶形运算算法

Algorithm 3 多线程 NTT 蝶形运算

Input: 多项式系数数组 a, 变换长度 $limit$

Input: 旋转因子 wn_mont, 模数 mod, Montgomery 因子 $mont\_factor$

Input: 线程数 $num\_threads$

$logn \leftarrow \log_2(limit)$ $offset \leftarrow 0$ **for** $stage \leftarrow 0$ **to** $logn - 1$ **do**

  $mid \leftarrow 2^{stage}$ $groups \leftarrow limit/(2 \times mid)$ $groups\_per\_thread \leftarrow \lceil groups/num\_threads \rceil$ **foreach** 线程 $t \in [0, num\_threads - 1]$ **do**

    $start_j \leftarrow t \times groups\_per\_thread \times 2 \times mid$ $end_j \leftarrow \min((t+1) \times groups\_per\_thread \times 2 \times mid, limit)$ 创建线程处理 $[start_j, end_j]$ 范围

  end

  等待所有线程完成 $offset \leftarrow offset + mid$

end

7.3.2 SIMD 向量化实现

Listing 1: SIMD 蝶形运算核心代码

```
// NEON SIMD 蝶形运算
for (int k = 0; k < mid; k += 4) {
    // 加载4个元素
    uint32x4_t x = vld1q_u32(a + j + k);
    uint32x4_t y = vld1q_u32(a + j + mid + k);
    uint32x4_t w = vld1q_u32(wn_mont + offset + k);

    // Montgomery 模乘
    uint32x4_t yw = montgomery_mul_neon(y, w, mod, mont_factor);

    // 蝶形运算
    uint32x4_t sum = vaddq_u32(x, yw);
    uint32x4_t diff = vsubq_u32(x, yw);

    // 存储结果
    vst1q_u32(a + j + k, sum);
    vst1q_u32(a + j + mid + k, diff);
}
```

7.3.3 融合优化的性能优势

计算资源利用率提升 SIMD+Pthread 融合架构通过双重并行机制实现了计算资源的充分利用。如图 **11** 所示，Pthread 线程层将计算任务分配到多个 CPU 核心，每个核心内的 SIMD 计算层则通过 NEON 指令实现数据级并行。这种设计使得：

- 多核并行：利用多核架构，线程级并行将计算负载分布到多个物理核心
- 向量计算：每个核心内使用 128 位 NEON 寄存器同时处理 4 个 32 位整数运算

相比单独使用 SIMD 或 Pthread，融合方案能够同时发挥线程级并行和数据级并行的优势。

内存访问优化 融合方案通过数据局部性优化减少了内存访问瓶颈：

- 数据分块：每个线程处理连续的内存块，提高缓存命中率
- 向量加载：NEON 的 `vld1q_u32` 指令一次性加载 4 个元素，减少内存访问次数
- 对齐访问：使用 `aligned_malloc` 确保内存地址对齐，优化向量加载性能

表 15: SIMD 与 SIMD+Pthread 性能对比

素数模数 (p)	**SIMD**	**SIMD+Pthread**	
	延迟 (ms)	**延迟 (ms)**	**进一步加速比**
$p = 7340033$	65.0948	**55.1650**	**1.18x**
$p = 104857601$	64.8335	**54.4819**	**1.19x**
$p = 469762049$	64.4712	**54.1774**	**1.19x**

7.4 子块并行策略

在高性能计算领域，**子块或称分块并行策略**是一种旨在优化内存层次结构（特别是 CPU 缓存）利用率的关键技术。当应用于 NTT 算法时，其核心思想不再是无限递归分解，而是将大规模的输入向量在逻辑上划分为若干个大小固定的、连续的“子块”。

并行计算被分为两个主要阶段：

1. **块内计算**：多线程并行地对各自负责的子块执行局部的 NTT 计算。此阶段计算的数据依赖性被严格限制在块内，从而获得极佳的数据局部性和缓存命中率。
2. **块间计算**：在所有块内计算完成后，进行需要跨越块边界的数据交换和蝶形运算。

这种策略通过将计算与数据访问模式对齐，有效减少了主内存的访问延迟，是 HPC 中实现极致性能的常用方法。

7.4.1 NTT 算法与子块策略的适配性

Cooley-Tukey NTT 算法的计算流程可以分为多个阶段。一个长度为 $N = B \times M$ 的 NTT，可以看作是先对 M 个长度为 B 的子块进行 NTT，然后乘以旋转因子，最后再进行 B 个长度为 M 的 NTT。子块策略正是利用了这一结构：

- **缓存友好**：通过选择合适的块大小，块内计算阶段可以避免缓存未命中，极大地提升了计算速度。
- **静态线程映射**：可以将子块静态地分配给不同的线程。如果有 T 个线程和 K 个子块，每个线程可以被分配 K/T 个块。这种静态划分减少了动态调度的开销。
- **向量化潜力**：块内计算是高度规律的，非常适合利用 SIMD 指令集进行进一步的底层加速。

7.4.2 执行流程

假设输入向量大小为 N，我们将其划分为 N/B 个大小为 B 的子块。

1. **初始化与数据划分**：在逻辑上将输入向量 A 划分为 $A_0, A_1, \ldots, A_{N/B-1}$。启动一个线程池，并将这些子块分配给池中的线程。
2. **第一阶段：块内 NTT**：
   - 每个线程独立且并行地对自己分配到的子块执行一个完整的、规模为 B 的 NTT 变换。
   - 此阶段所有计算和数据访问都局限于块内，线程之间无任何数据依赖和通信。
3. **全局同步**：使用一个 Barrier 确保所有线程都完成了各自的块内 NTT 计算。
4. **第二阶段：旋转因子与块间蝶形运算**：
   - 对数据进行转置，以准备块间计算。
   - 线程再次并行启动，此时的蝶形运算会跨越原始子块的边界。每个线程计算的输入数据可能来自于前一阶段中由不同线程计算出的结果。此阶段是内存带宽敏感的。
5. **迭代**：根据算法需要，重复块内与块间计算的模式，直到所有 NTT 阶段完成。

7.4.3 子块并行策略示意图

下图描绘了子块策略的两个核心阶段。首先，多线程在各自的块内并行工作。同步后，进行需要跨块数据访问的蝶形运算。

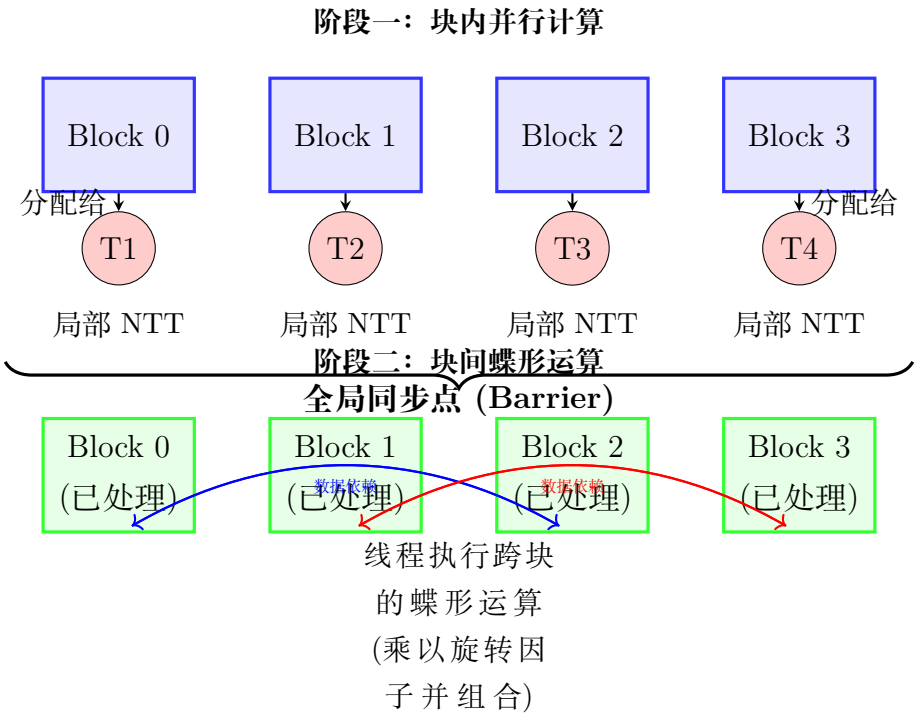


图 **12**: NTT 的子块并行计算模型

算法	n	p	平均延迟（us）	结果正确性
子块并行策略	131 072	7 340 033	352.1283	正确
	131 072	104 857 601	353.4123	正确
	131 072	469 762 049	353.5435	正确

可以看到效果并不好，NTT 固有的跨步访问模式与子块边界冲突严重，导致线程间通信激增和内存访问低效，抵消了并行化的潜在收益。

7.5 创新规约方法：惰性规约

7.5.1 底层原理与数学基础

惰性规约是一种优化模算术计算的策略，其核心在于有选择地延迟模运算的执行。在传统模运算中，每次加法或乘法后都会立即执行模运算以确保结果位于 $[0, p-1]$ 范围内。惰性规约则允许中间结果暂时超出模数范围，保持在 $[0, k \cdot p)$ 区间内（其中 k 为小的正整数），仅在必要时执行规约操作。

其数学基础在于模运算的同余性质：对于任意整数 x 和正整数 p，若 $x \equiv y \pmod{p}$ 且 $y \in [0, p-1]$，则对任何中间值 z 满足 $z = x + m \cdot p$（m 为整数），都有 $z \equiv x \pmod{p}$。惰性规约利用这一性质，在计算过程中保留额外的倍数 $m \cdot p$，从而避免每次操作后都进行昂贵的模运算。

在 NTT 的蝶形运算中，这一策略具体表现为：

$$\begin{cases} u_{\text{out}} = u_{\text{in}} + t \\ v_{\text{out}} = u_{\text{in}} - t + 2p \end{cases}$$

其中 $u_{\text{out}} \in [0, 3p)$，$v_{\text{out}} \in [0, 4p)$，而传统实现中两者都严格在 $[0, p)$ 范围内。这种扩展范围的表示允许我们将多个算术操作累积后一次性规约。

7.5.2 边界条件与安全性

惰性规约的关键是确保中间值不会溢出硬件容限。在 32 位系统中，最大安全范围为 $[0, 2^{32} - 1]$。因此，算法设计需满足：

$$k \cdot p < 2^{32}$$

对于典型密码学参数（$p \approx 2^{30}$），取 $k = 4$ 可满足 $4p < 2^{32}$。在蝶形运算的每一阶段，我们严格保证：

- 加法结果 $u_{\text{out}} < 3p$
- 减法结果 $v_{\text{out}} < 4p$

此边界条件通过精心设计运算顺序和偏移量保证，确保整个 NTT 计算过程中间值始终可控。

7.5.3 适合 NTT 算法的原因

惰性规约技术特别适合 NTT 算法，主要源于 NTT 的计算结构和模运算特性。NTT 本质上是 FFT 在有限域上的变体，其核心是分阶段的蝶形运算网络。每个蝶形运算包含乘法和加法操作，且整个算法需要 $O(n \log n)$ 次此类操作。

在传统实现中，每个蝶形运算后都需进行模规约，导致高频度的模运算调用。惰性规约通过利用蝶形运算的层级特性，允许在多个阶段累积计算后再执行规约。具体而言，在乘法操作前对操作数进行规约，而对加法/减法则延迟规约。这种策略显著减少了模运算次数，从 $O(n \log n)$ 降至 $O(n)$。

此外，NTT 的迭代结构天然支持范围传播控制。每一级的输出范围可预测，使得我们可以精确管理中间值的增长。结合现代 GPU 的宽字长特性，这种技术能在不增加硬件开销的前提下最大化计算效率。

7.5.4 算法流程与实现

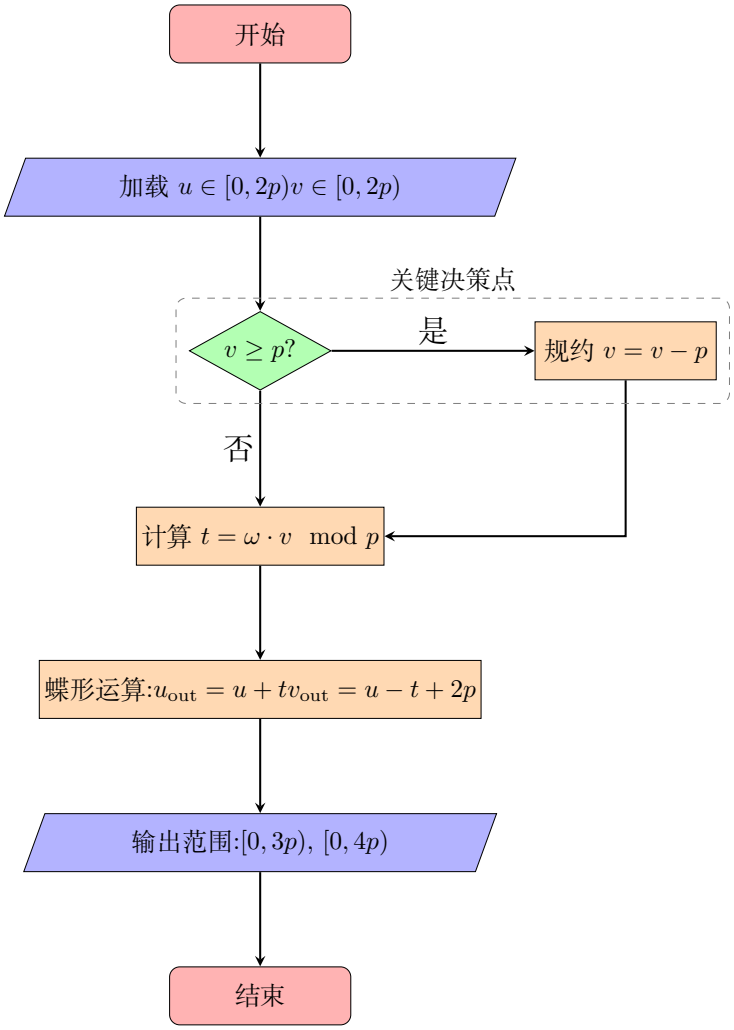


图 **13**: 惰性规约蝶形运算流程

流程图解

整体流程 惰性规约在 NTT 中的实现分为三个阶段：

1. **正向 NTT**：执行惰性规约的蝶形运算，中间值范围扩展至 $[0, 4p)$
2. **点乘**：规约输入值至 $[0, p)$ 后执行模乘
3. **逆向 NTT**：类似正向 NTT，最后执行完全规约

Algorithm 4 惰性规约 NTT 蝶形运算

Input: 多项式系数 a, 变换长度 N, 模数 p

Output: 变换结果

初始化: $k \leftarrow 1$ **while** $k < N$ **do**

  $m \leftarrow 2k$ **foreach** 蝶形运算组 **do**

    加载 u (范围 $[0, 2p]$), v (范围 $[0, 2p]$) **if** $v \geq p$ **then**

      $v \leftarrow v - p$; // 乘法前规约

    else

      保持 v

    end

    计算 $t = \omega \cdot v \mod p$ $u_{\text{new}} \leftarrow u + t$; // 范围 $[0, 3p)$

    $v_{\text{new}} \leftarrow u - t + 2p$; // 范围 $[0, 4p)$

    存储结果

  end

  $k \leftarrow m$

end

这个算法我反复尝试过了，但是一直没有跑通，原理应该是没问题的，就算是提出来的一个想法吧 ()。

7.6 灵活线程数控制

在 Pthread 实验中，原始代码使用了固定的三个线程来并行计算三个模数下的 NTT。这种实现方式虽然利用了多核处理器的能力，但缺乏灵活性。在不同的硬件环境或测试需求下，我们可能希望使用不同数量的线程（例如，在单核环境下使用单线程，或在双核环境下使用双线程）来评估和比较性能。本次修改的核心目的就是引入一个灵活的线程控制机制，使得程序可以根据用户的指定，动态地决定用于计算的线程数量。

7.6.1 代码修改的核心逻辑

主要的改动集中在 `poly_multiply` 函数和 `main` 函数。

1. **通过命令行参数控制线程数：** 在 `main` 函数中，我们通过检查程序启动时的命令行参数（`argc` 和 `argv`）来获取用户期望的线程数。如果没有指定，则默认为 3 个线程。

```
int num_threads = 3; // 默认值
if (argc > 1) {
    num_threads = stoi(argv[1]);
```

```
}
```

2. **动态的任务分发逻辑：** 修改后的 `poly_multiply` 函数会根据传入的 `num_threads` 参数来决定如何执行三个独立的 NTT 任务。

   - **单线程模式 (`num_threads <= 1`):** 如果线程数设置为 1 或更少，程序将不会创建任何新线程。三个 NTT 任务会依次在主线程中串行执行。这对于性能基准测试和在单核 CPU 上运行非常有用。
   - **多线程模式 (`num_threads > 1`):** 程序会根据指定的线程数创建相应数量的线程。任务被分配给这些新创建的线程。如果线程数少于 3，那么多余的任务会由主线程负责执行，从而确保所有核心都被充分利用。

参考文献

[1] https://blog.csdn.net/weixi_44885334/article/details/134532078

[2] V4: The Number-Theoretic Transform (NTT) [Slide presentation]. © Alfred Menezes.