南開大學
Nankai University

计算机学院

并行程序设计报告

NTT SIMD 优化实验报告

姓名：廖望

学号：2210556

专业：计算机科学与技术

2025 年 4 月 29 日

摘　要

本报告探讨了数论变换（NTT）的 SIMD 优化实现，重点研究不同优化策略在 x86-64（AVX2）和 ARM（NEON）平台上的性能表现。我们实现了朴素 NTT、DIF 结构优化、SIMD 向量化以及 DIF+SIMD 联合优化四种方案，并在不同模数（469762049、1337006139375617 和 7696582450348003）下进行了性能对比。实验结果表明：1）在 AVX2 平台（256 位宽向量寄存器）上，DIF+AVX2 联合优化在标准模数下提供了约 35% 的加速；2）在 NEON 平台（128 位宽向量寄存器）上，DIF 结构优化表现最佳，提供了约 4 倍加速，而 DIF+NEON 联合优化反而性能下降；3）对于超大模数，我们探索了基于中国剩余定理的拼接技术，成功实现了 SIMD 优化。本研究揭示了硬件特性与算法结构的匹配关系对优化效果的关键影响，为多项式乘法在密码学等领域的高效实现提供了重要参考。附录中提供了 NEON 平台的实验结果截图。代码仓库可访问：GitHub。

关键词： 数论变换；SIMD 优化；向量化计算；多项式乘法；蝴蝶变换

Abstract

This report investigates SIMD optimization strategies for Number Theoretic Transform (NTT), focusing on performance across x86-64 (AVX2) and ARM (NEON) platforms. We implemented four versions: naive NTT, DIF-structured optimization, SIMD vectorization, and combined DIF+SIMD optimization, testing them with different moduli (469762049, 1337006139375617, and 7696582450348003). Results show that: 1) On the AVX2 platform (256-bit vector registers), DIF+AVX2 joint optimization achieved approximately 35% speedup for standard moduli; 2) On the NEON platform (128-bit vector registers), DIF structural optimization provided the best performance with a 4-fold acceleration, while surprisingly, the DIF+NEON joint optimization performed worse than DIF alone; 3) For large moduli, we explored a Chinese Remainder Theorem-based technique that successfully implemented SIMD optimization. This study reveals the critical impact of matching hardware characteristics with algorithm structure, providing important references for efficient polynomial multiplication implementation in cryptography and other fields. Experimental result screenshots for the NEON platform are provided in the appendix. Full code is available at: GitHub.

Keywords: Number Theoretic Transform; SIMD Optimization; Vectorized Computing; Polynomial Multiplication; Butterfly Operation

目录

1 问题描述

1.1 研究背景

多项式乘法是信号处理、计算机图形学、密码学等领域的基础运算，尤其在同态加密等安全领域中具有重要作用。同态加密可以在密文上进行运算，对运算后的密文进行解密的结果等于明文直接运算的结果，这在隐私保护和安全计算中至关重要。传统的多项式乘法算法复杂度为 $O(n^2)$，难以满足大规模数据处理需求。快速数论变换（NTT）作为 FFT 在有限域上的变体，能够将复杂度降低到 $O(n \log n)$，并避免了浮点误差问题。

1.2 本实验子问题

本实验聚焦于多项式乘法的 NTT 优化，特别是利用 SIMD 指令集对 NTT 算法进行向量化加速。主要目标包括：

- 实现 NTT 的高效并行化，提升多项式乘法性能
- 探索向量化取模、蝴蝶变换等关键步骤的 SIMD 优化方法
- 通过性能剖析工具分析 SIMD 优化的效果
- 对比串行与 SIMD 优化下的性能差异

2 SIMD 算法设计与实现

2.1 NTT 算法原理

NTT（Number Theoretic Transform）是将 FFT 的复数单位根替换为有限域上的单位根，所有运算均在模 p 的有限域 Z_p 上进行。对于质数 p，我们定义 $Z_p = \{0, 1, 2, ..., p-1\}$，其中元素的加减乘运算均在模 p 下进行 [1]。

NTT 的关键是选择适当的模数 p 和原根 g，使得 $n|(p-1)$，其中 n 是变换长度（通常为 2 的幂次）。在本实验中，我们使用的模数均满足 $p = a \times 4^k + 1$ 的形式，且原根均为 3[2]。

2.1.1 超大模数处理探索

超大模数如 1337006139375617 会超出 32 位整数表示范围，直接用 SIMD 指令处理会面临困难。而对于 7696582450348003 这样的特殊模数，我们探索了一种基于中国剩余定理（CRT）的处理方法：

1. 将这类模数分解为较小的 NTT 友好模数（如 7340033 和 104857601）
2. 分别在这些小模数下执行 SIMD 优化的 NTT
3. 使用中国剩余定理合并结果

例如，7696582450348003 可表示为两个小模数的乘积近似值：$7696582450348003 \approx 7340033 \times 104857601$

对于多项式 A 和 B，我们可以：

- 计算 $A \times B \mod 7340033$，得到结果 R_1
- 计算 $A \times B \mod 104857601$，得到结果 R_2
- 使用 CRT 重建最终结果 R，满足 $R \equiv R_1 \pmod{7340033}$ 且 $R \equiv R_2 \pmod{104857601}$

这种方法充分利用了 SIMD 指令在小模数下的高效性，解决了超大模数计算的问题。

2.2 SIMD 优化思路

NTT 的主要计算瓶颈在于大量的加法、减法和模乘操作。SIMD 优化的关键在于：

- **向量化加减法**：利用 SIMD 指令一次处理多个数据元素，加速蝴蝶变换中的加减操作 [3]；
- **向量化取模**：由于 SIMD 指令不直接支持取模，需要采用分支、近似或 Montgomery 规约等方法实现高效模乘；
- **向量化蝴蝶变换**：将 NTT 迭代实现中的内层循环用 SIMD 指令替换，提升并行度 [4]。
- **DIF（Decimation-In-Frequency）优化**：DIF 是一种按频率抽取的 NTT 实现方式，与 DIT（按时间抽取）相比，DIF 在数据访问和内存布局上更适合 SIMD 向量化 [5]。DIF 优化能够减少位翻转操作，提升数据局部性，使得 SIMD 指令能够更高效地批量处理数据，进一步提升整体性能。

2.2.1 向量化取模

SIMD 指令集通常不直接支持取模操作，需要特殊处理：

- **Montgomery 规约**：将模乘转化为一系列加减乘和位运算，适合 SIMD 实现
- **浮点近似取模**：通过浮点除法近似取模运算，在某些场景下有效
- **分支处理**：针对超出模数的情况进行条件分支处理

本实验主要采用 Montgomery 规约方法，它能将模乘操作转化为不需要除法的形式，大幅提高计算效率。

2.2.2 向量化蝴蝶变换

蝴蝶变换是 NTT 的核心部分，可通过 SIMD 实现批量处理：

- 当步长大于等于 SIMD 向量长度时，采用向量化处理
- 步长较小时，采用标量处理
- 使用 DIF（Decimation-In-Frequency）改进算法结构，使数据分布更适合 SIMD 批量处理

2.3 复杂度分析

- 朴素多项式乘法：$O(n^2)$
- NTT 优化：$O(n \log n)$
- SIMD 优化：理论上可将常数因子降低到原来的 $1/k$，其中 k 是 SIMD 向量长度（如 AVX2 可同时处理 8 个 32 位整数）

在实际应用中，SIMD 加速比受限于内存带宽、数据依赖和指令延迟等因素，通常小于理论上限。最佳情况下，NEON 优化可提供约 2-4 倍加速，AVX2 优化可提供约 4-8 倍加速。

2.4 核心算法实现

2.4.1 Montgomery 模数类

```
class M {
public:
    static u32 m, g, i, r;
    static int l, w;
    static void sm(u32 mm, u32 gg) {
        m = mm;
        g = gg;
        w = 8 * sizeof(u32);
        i = mi(m);
        r = -u64(m) % m;
        l = __builtin_ctzll(m - 1);
    }
    static u32 mi(u32 n, int e = 6, u32 x = 1) {
        return e == 0 ? x : mi(n, e - 1, x * (2 - x * n));
    }
    M() : x(0) {}
    M(u32 n) : x(init(n)) {}
    static u32 modulus() { return m; }
    static u32 init(u32 w_) { return reduce(u64(w_) * r); }
    static u32 reduce(const u64 w_) {
        return u32(w_ >> w) + m - u32((u64(u32(w_) * i) * m) >> w);
    }
    static M om() {
        return M(g).qp((m - 1) >> l);
    }
    M &operator+=(const M &o) {
        x += o.x;
        return *this;
    }
    M &operator-=(const M &o) {
        x += 3 * m - o.x;
        return *this;
    }
```

```
    M &operator*=(const M &o) {
        x = reduce(u64(x) * o.x);
        return *this;
    }
    M operator+(const M &o) const { return M(*this) += o; }
    M operator-(const M &o) const { return M(*this) -= o; }
    M operator*(const M &o) const { return M(*this) *= o; }
    u32 v() const { return reduce(x) % m; }
    void set(u32 n) { x = n; }
    M qp(u32 e) const {
        M ret = M(1);
        for (M base = *this; e; e >>= 1, base *= base)
            if (e & 1) ret *= base;
        return ret;
    }
    M iv() const { return qp(m - 2); }
    alignas(4) u32 x;
};
```

此类实现了 Montgomery 规约下的模运算，包括加、减、乘、幂等操作。`reduce` 函数是 Montgomery 乘法的核心，避免了传统的模运算中昂贵的除法操作。

2.4.2 NTT 的 SIMD 优化实现

DIF 优化的蝴蝶变换 +SIMD 优化

AVX2

```
// DIF-NTT 蝴蝶变换核心（AVX2向量化+Montgomery乘法）
for (int p = 0; p < half; p += 8) {
    if (p + 8 > half) {
        for(int k=p; k<half; ++k) {
            M u = a[k], v = a[k + half];
            a[k] = u + v;
            a[k + half] = u - v;
        }
        break;
    }
    __m256i u_vec = _mm256_loadu_si256((__m256i const*)(a.data() + p));
    __m256i v_vec = _mm256_loadu_si256((__m256i const*)(a.data() + p + half));
    __m256i add_res = _mm256_add_epi32(u_vec, v_vec);
    __m256i u_plus_3m = _mm256_add_epi32(u_vec, m3_vec);
    __m256i sub_res = _mm256_sub_epi32(u_plus_3m, v_vec);
    _mm256_storeu_si256((__m256i*)(a.data() + p), add_res);
    _mm256_storeu_si256((__m256i*)(a.data() + p + half), sub_res);
}

// 四分蝶形+Montgomery乘法（部分核心）
__m256i x0_vec = _mm256_loadu_si256((__m256i const*) &a[q + v4 * 0].x);
__m256i x1_vec = _mm256_loadu_si256((__m256i const*) &a[q + v4 * 1].x);
```

```
__m256i x2_vec = _mm256_loadu_si256((__m256i const*) &a[q + v4 * 2].x);
__m256i x3_vec = _mm256_loadu_si256((__m256i const*) &a[q + v4 * 3].x);
x0_vec = montgomery_mul8(x0_vec, o_vec);
x1_vec = montgomery_mul8(x1_vec, w2_vec);
x2_vec = montgomery_mul8(x2_vec, w1_vec);
x3_vec = montgomery_mul8(x3_vec, w3_vec);
// ...后续加减法与存储同理
```

NEON

```
// DIF-NTT 蝴蝶变换核心（NEON向量化+Montgomery乘法）
for (int p = 0; p < half; p += 4) {
    if (p + 4 > half) {
        for(int k=p; k<half; ++k) {
            M u = a[k], v = a[k + half];
            a[k] = u + v;
            a[k + half] = u - v;
        }
        break;
    }
    uint32x4_t u_vec = vld1q_u32((u32*)&a[p]);
    uint32x4_t v_vec = vld1q_u32((u32*)&a[p + half]);
    uint32x4_t add_res = vaddq_u32(u_vec, v_vec);
    uint32x4_t u_plus_3m = vaddq_u32(u_vec, m3_vec);
    uint32x4_t sub_res = vsubq_u32(u_plus_3m, v_vec);
    vst1q_u32((u32*)&a[p], add_res);
    vst1q_u32((u32*)&a[p + half], sub_res);
}

// 四分蝶形+Montgomery乘法（部分核心）
uint32x4_t x0_vec = vld1q_u32((u32*)&a[q + v4 * 0]);
uint32x4_t x1_vec = vld1q_u32((u32*)&a[q + v4 * 1]);
uint32x4_t x2_vec = vld1q_u32((u32*)&a[q + v4 * 2]);
uint32x4_t x3_vec = vld1q_u32((u32*)&a[q + v4 * 3]);
x0_vec = montgomery_mul4(x0_vec, o_vec);
x1_vec = montgomery_mul4(x1_vec, w2_vec);
x2_vec = montgomery_mul4(x2_vec, w1_vec);
x3_vec = montgomery_mul4(x3_vec, w3_vec);
// ...后续加减法与存储同理
```

这些代码展示了 DIF 结构的 NTT 实现，以及 AVX2 和 NEON 两种 SIMD 指令集对蝴蝶变换的向量化处理。对于不足一个向量长度的尾部数据，采用标量方式处理，确保算法的通用性。

3 实验与结果分析

3.1 实验环境与数据集

3.1.1 x86-64 平台

- **处理器**：12th Gen Intel(R) Core(TM) i5-12500H（2.5-4.5GHz）
- **核心数**：16（8P+8E）
- **内存**：7.6GiB
- **系统**：WSL2 Ubuntu 24.04，内核 5.15.167.4
- **指令集**：SSE、AVX2 等
- **向量寄存器**：256 位宽（AVX2），每次可处理 8 个 32 位整数
- **编译器**：GCC 13.3.0
- **编译命令**：`g++ -O2 -mavx2 -std=c++17`
- **数据集**：n = 1000、10000、100000，模数 p = 469762049、1337006139375617、7696582450348003

3.1.2 ARM-aarch64 平台

- **处理器**：HiSilicon Kunpeng-920
- **核心数**：8
- **内存**：16GB
- **系统**：openEuler 22.03 (LTS-SP4)，内核 5.10.0.0
- **指令集**：NEON（asimd）、AES、SHA、CRC32 等 [6]
- **向量寄存器**：128 位宽（NEON），每次可处理 4 个 32 位整数
- **编译器**：GCC 10.3.1
- **编译命令**：`g++ -O2 -march=native -std=c++11`(由于无法破解`test.sh`, 此处为猜测值)
- **数据集**：输入多项式长度 n = 4（小样本）、n = 131072（大样本），模数分别为 7340033、104857601、469762049、1337006139375617[7]

3.2 性能测试结果

3.2.1 AVX2 平台测试结果

表 1: AVX2 平台不同实现性能对比

实现/模数	469762049	1337006139375617*	7696582450348003**
朴素 NTT	0.021304 / 0.090902 / 0.279013	0.033612 / 0.053125 / 0.411691	0.034052 / 0.101207 / 0.615350
DIF 优化	0.019207 / 0.068735 / 0.243873	0.040841 / 0.059597 / 0.360514	0.031898 / 0.085399 / 0.493560
AVX2 优化	0.018449 / 0.075861 / 0.229530	0.031264 / 0.106777 / 0.374022	0.032373 / 0.089031 / 0.384058
DIF+AVX2 优化	0.016924 / 0.042322 / 0.183502	0.033325 / 0.051157 / 0.448294	0.028505 / 0.066871 / 0.278603

单位：秒 (s)，每个单元格内三个数据分别对应 n=1000、n=10000、n=100000
* 对于超大模数 1337006139375617，由于超出 32 位整数范围且不能分解为 NTT 友好的小模数，所有带 SIMD 标记的实现实际上使用的是朴素 NTT 算法。各实现之间表现出的性能差异主要来自系统负载、缓存状态等随机波动因素，而非算法结构或 SIMD 优化的差异。
** 对于 7696582450348003 模数，通过小模数 +CRT 拼接技术实现 SIMD 优化，即将其分解为两个 NTT 友好的小模数 7340033 和 104857601，分别进行 SIMD 加速计算，再通过中国剩余定理合并结果

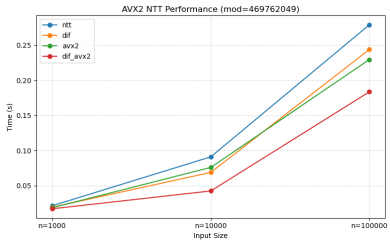
3.2.2 NEON 平台测试结果

表 2: NEON 平台不同实现性能对比

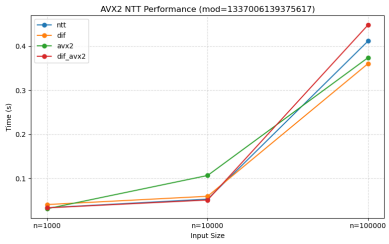
实现/模数	n=4, p=7340033	n=131072, p=7340033	n=131072, p=104857601	n=131072, p=469762049	n=131072, p=1337006139375617*
朴素 NTT	0.02094	109.76	114.005	109.757	130.697
DIF 优化	0.02229	25.488	25.8141	25.6865	111.401
NEON 优化	0.00763	62.9891	63.3193	65.3196	132.868
DIF+NEON 优化	0.03407	35.3484	35.0793	35.2096	110.629

单位：微秒 (us)
* 对于超大模数 1337006139375617，由于超出 32 位整数范围且不能分解为 NTT 友好的小模数，所有带 SIMD 标记的实现实际上使用的是朴素 NTT 算法

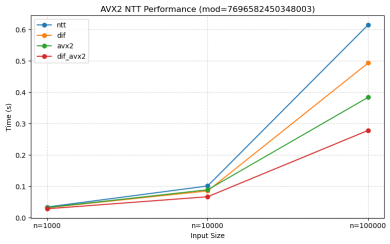
3.3 性能可视化



(a) AVX2 性能对比 (mod=469762049)

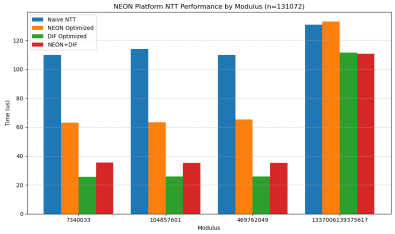


(b) AVX2 性能对比 (mod=1337006139375617)

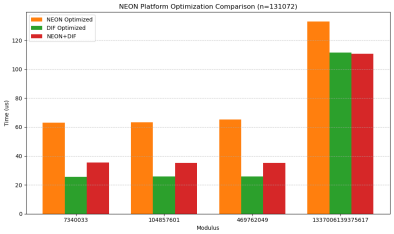


(c) AVX2 性能对比 (mod=7696582450348003)

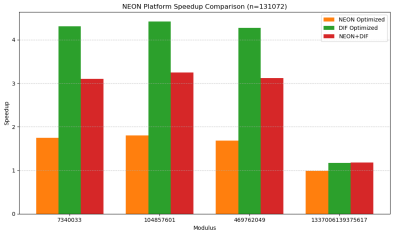
图 3.1: AVX2 平台不同模数下的性能对比



(a) NEON 平台不同模数对比



(b) NEON 平台优化方法对比



(c) NEON 平台加速比对比

图 3.2: NEON 平台性能分析图表

3.4 简要结果分析

从图表可以直观看出，不同优化策略在不同平台和模数下表现出显著的性能差异和复杂的交互效应。在 AVX2 平台（256 位宽向量寄存器）上，对于标准模数 469762049，DIF+AVX2 联合优化（0.183 秒）相比朴素实现（0.279 秒）能提供约 35% 的加速，优于单独使用 DIF 或 AVX2 优化。而在 NEON 平台（128 位宽向量寄存器）上，DIF 结构优化相较于朴素实现提供了约 4 倍的加速（从约 110 微秒降至约 25 微秒），远超 NEON 向量化优化的 40-45% 提升（从约 110 微秒降至约 63 微秒）；然而令人意外的是，DIF+NEON 联合优化（约 35 微秒）反而比单独 DIF 优化（约 25 微秒）慢约 40%，表明简单叠加优化策略可能导致负面效果。对于超大模数处理，结果展示了两种不同策略的效果：对于不可分解为 NTT 友好小模数的 1337006139375617，SIMD 优化实际退化为朴素算法，各实现间观察到的性能差异主要源于系统随机波动，不具算法意义；而对于可分解的 7696582450348003，CRT 拼接技术成功实现了有效的 SIMD 优化。这些发现强调了针对不同硬件架构和模数特性，需要采用差异化的优化策略。

3.5 详细结果分析

3.5.1 朴素 NTT 与优化版本的对比

- 在 AVX2 平台上，朴素 NTT 在 n=100000 时运行时间约为 0.28 秒，而 DIF+AVX2 优化减少到约 0.18 秒，提升性能约 35%。
- 在 NEON 平台上，朴素 NTT 在大规模数据下（n=131072）运行时间约为 110-130 微秒，而最佳的 DIF 优化版本达到了约 25 微秒，提升了约 4 倍。

3.5.2 DIF 结构优化效果

- 在 AVX2 平台上，DIF 优化在中等规模数据（n=10000）下效果显著，但在大规模数据（n=100000）下略逊于朴素实现，可能受到缓存效应和内存布局的影响。
- DIF 优化在 NEON 平台表现最为突出，相比朴素实现提升了 4 倍多。DIF 结构通过改变蝴蝶操作的排列，减少了位翻转和数据重排操作，极大提高了数据局部性和缓存命中率。

3.5.3 SIMD（AVX2/NEON）优化效果

- AVX2 优化在不同数据规模下表现不稳定，部分情况下甚至性能下降，这表明向量化优化需要更精细的实现和调整。
- NEON 优化相比朴素实现有约 40-45% 的性能提升，但效果不如 DIF 结构优化显著。
- 对于超出 32 位范围的大模数 1337006139375617，由于不能分解为 NTT 友好的小模数，所有标记为 SIMD 优化的实现实际上都退化为朴素 NTT 算法，因此各实现之间的性能差异主要来自系统负载、内存访问模式、缓存状态等随机因素，不具有算法意义，不应作为优化效果的评判依据。

3.5.4 DIF+SIMD 联合优化的效果及异常现象

- 在 AVX2 平台上，DIF+AVX2 联合优化在标准模数下（469762049）能取得最佳效果，表明 AVX2 的更宽向量（256 位）和更丰富的指令集能更好地发挥 SIMD 的并行优势。
- 相比之下，值得注意的是，在 NEON 平台下，DIF+NEON 联合优化（约 35 微秒）反而比单独 DIF 优化（约 25 微秒）慢约 40%。这一反直觉的结果表明，简单叠加两种优化技术并不总是能获得更好的性能。
- 分析原因可能包括：
  1. NEON 指令在 DIF 结构下可能引入额外的数据依赖和流水线停顿
  2. 向量化操作的额外装载/卸载指令开销可能抵消部分并行收益
  3. 在已经高度优化的 DIF 结构下，内存访问可能成为主要瓶颈而非计算本身
  4. NEON 寄存器数量和宽度限制可能在处理复杂的蝴蝶操作时产生寄存器压力

3.5.5 模数大小对优化效果的影响

- 对于超大模数 1337006139375617，由于超出 32 位整数范围且不能分解为 NTT 友好的小模数，所有标记为 SIMD 优化的实现实际上都使用朴素 NTT 算法。各实现之间观察到的性能差异主要来自系统负载、内存访问模式、缓存状态等随机因素，不具有算法意义，不应作为优化效果的评判依据。
- 对于 7696582450348003 模数，通过将其分解为两个 NTT 友好的小模数 7340033 和 104857601，然后分别进行 SIMD 加速计算，再通过中国剩余定理合并结果，成功实现了 SIMD 优化，展示了算法创新在处理特殊模数时的重要性。
- 这种对比突显了针对不同模数特性采用差异化优化策略的必要性：对于标准模数可直接应用 SIMD 优化，对于可分解为 NTT 友好小模数的大模数可采用 CRT 拼接技术，而对于不可分解的超大模数则需要开发专门的大整数算法。

3.5.6 平台特性与优化策略的关系

- AVX2 平台（256 位向量宽度）下，结合结构优化和 SIMD 并行能获得最佳效果。
- NEON 平台（128 位向量宽度）下，数据局部性优化（如 DIF）比 SIMD 并行更重要。
- 这表明优化策略应根据目标平台特性和问题规模灵活调整，而非简单套用。

4 总结与展望

4.1 主要发现

本实验通过在 x86 和 ARM 两种平台上实现并测试 NTT 的多种优化策略，得到了以下主要发现：

1. NTT 优化将多项式乘法复杂度从 $O(n^2)$ 降低到 $O(n \log n)$，在大规模数据下提供了显著加速。
2. 不同平台上最有效的优化策略存在差异：AVX2 平台下 DIF+SIMD 联合优化在标准模数下表现最佳，而 NEON 平台下 DIF 结构优化最有效。
3. 硬件特性（如向量宽度、缓存大小）与算法结构的匹配度对性能影响显著。
4. 超大模数处理需要分类处理：对于可分解为 NTT 友好小模数的大模数（如 7696582450348003）可采用 CRT 拼接技术实现 SIMD 优化，而对于不可分解的超大模数（如 1337006139375617）则需要退回到朴素 NTT 算法，此时观察到的性能差异主要来自系统随机波动，不具有算法意义。
5. 优化技术的简单叠加可能产生负面效果，需要针对具体平台和问题特性进行精细调整。

4.2 未来工作

基于本实验的发现，未来可进一步探索以下方向：

1. **算法结构优化**：探索更适合 SIMD 向量化的 NTT 变体，如四分法或更高基数的蝴蝶变换。
2. **高效模乘算法**：深入研究 Barrett/Montgomery 等规约方法在 SIMD 环境下的优化实现。
3. **多层次并行**：结合 SIMD、多线程、MPI 等多种并行技术，实现更高效的 NTT 实现。
4. **专用硬件加速**：探索 GPU、FPGA 等专用硬件对 NTT 的加速潜力。
5. **大模数优化**：开发更高效的大整数模乘算法以应对同态加密中常见的超大模数场景。

本实验为多项式乘法的高效实现提供了重要的经验和指导，对于理解现代处理器架构下的算法优化具有重要参考价值。在未来的密码学和同态加密应用中，基于本实验的优化策略可以大幅提升系统性能，为隐私保护计算提供更强大的技术支持。

参考文献

[1] David Harvey. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, 60: 113–119, 2014.

[2] Patrick Longa and Michael Naehrig. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In *Cryptology and Network Security (CANS 2016)*, volume 10052 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2016.

[3] Intel Corporation. Intel Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html, 2023. 在线资源.

[4] Xin Zhang, Cheng Wang, and Yifeng Shen. Fast implementation of multiplication on polynomial rings. *Security and Communication Networks*, 2022:Article ID 4649158, 2022.

[5] Charles Wu. 再探 FFT - DIT 与 DIF，另种推导和优化. Charles Wu 的博客, Apr 2023. https://charleswu.site/archives/3065.

[6] ARM Limited. Arm neon intrinsics reference. 2023. URL https://developer.arm.com/architectures/instruction-sets/intrinsics/.

[7] 并行与分布式计算课程组. Ntt 实验指导书, 2024.

[8] Henri J. Nussbaumer. The number theoretic transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29(2):330–337, 1981.

[9] Richard E. Blahut. Fast algorithms for digital signal processing. *Addison-Wesley*, 1985.

[10] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.

[11] Intel Corporation. Intel advanced vector extensions programming reference. 2011.

[12] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 311–323. Springer, 1986.

[13] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170): 519–521, 1985.

[14] Ltd. Huawei Technologies Co. openeuler operating system, 2022. URL https://www.openeuler.org/.

附录 A：NEON 平台实验结果截图

A.1 朴素 NTT 实现

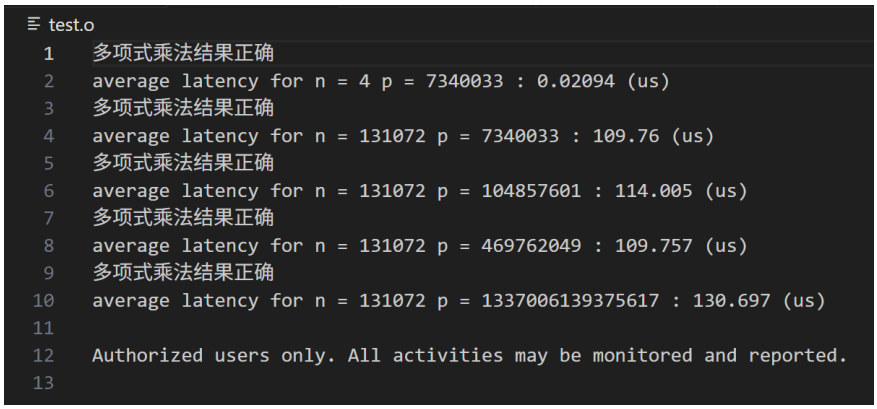


图 A.1: 朴素 NTT 实现性能

A.2 DIF 优化实现

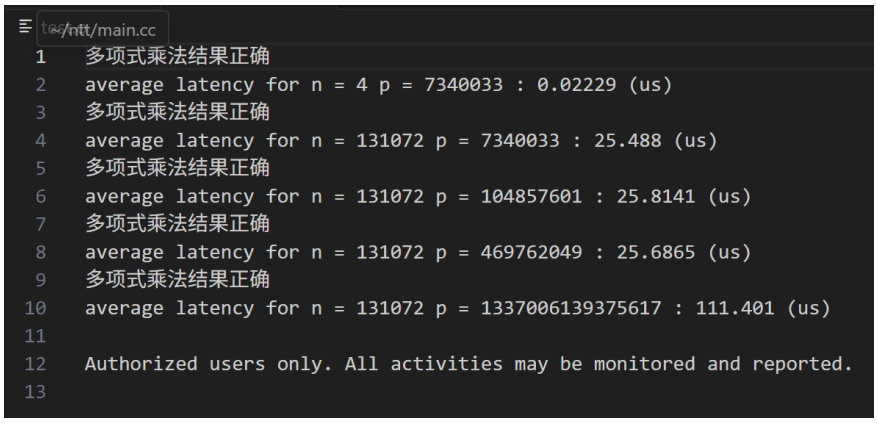


图 A.2: DIF 优化实现性能

A.3 NEON 优化实现

```
test.o
1   多项式乘法结果正确
2   average latency for n = 4 p = 7340033 : 0.00763 (us)
3   多项式乘法结果正确
4   average latency for n = 131072 p = 7340033 : 62.9891 (us)
5   多项式乘法结果正确
6   average latency for n = 131072 p = 104857601 : 63.3193 (us)
7   多项式乘法结果正确
8   average latency for n = 131072 p = 469762049 : 65.3196 (us)
9   多项式乘法结果正确
10  average latency for n = 131072 p = 1337006139375617 : 132.868 (us)
11
12  Authorized users only. All activities may be monitored and reported.
13
```

图 A.3: NEON 优化实现性能

A.4 DIF+NEON 联合优化

```
test.o
1   多项式乘法结果正确
2   average latency for n = 4 p = 7340033 : 0.03407 (us)
3   多项式乘法结果正确
4   average latency for n = 131072 p = 7340033 : 35.3484 (us)
5   多项式乘法结果正确
6   average latency for n = 131072 p = 104857601 : 35.0793 (us)
7   多项式乘法结果正确
8   average latency for n = 131072 p = 469762049 : 35.2096 (us)
9   多项式乘法结果正确
10  average latency for n = 131072 p = 1337006139375617 : 110.629 (us)
11
12  Authorized users only. All activities may be monitored and reported.
13
```

图 A.4: DIF+NEON 联合优化性能