



南开大学
Nankai University

南开大学

计算机学院和密码与网络空间安全学院

《并行程序设计》实验报告

作业三：NTT 算法 SIMD 并行优化

姓名：梁景铭

学号：2312632

专业：计算机科学与技术

指导教师：王刚

2025 年 4 月 26 日

摘要

本文聚焦 NTT 算法优化，基于 SIMD 并行技术实现向量化 Montgomery 模乘与蝴蝶变换，通过数据对齐和指令集加速降低计算开销，实验验证其显著提升多项式乘法效率。实验代码及图片已全部上传至：

https://github.com/eprogressing/NKU_COSC0025_Parallel

关键字：NTT，SIMD，Montgomery 模乘，蝴蝶变换

目录

1 问题描述	1
1.1 期末选题	1
1.2 本次题目选题	1
1.2.1 实验要求	1
2 NTT 算法	2
2.1 FFT、NTT 算法对比分析	2
2.1.1 关键公式	2
2.1.2 FFT，NTT 计算开销对比	3
2.2 NTT 算法分析	3
2.2.1 数论前置知识	3
2.3 NTT 串行算法实现	4
3 NTT 算法优化	7
3.1 Montgomery 模乘	7
3.1.1 理论分析	7
3.1.2 参数预计算	7
3.1.3 标量运算优化	7
3.1.4 数据格式转换流程	8
3.2 蝴蝶变换	8
3.3 四分 NTT	9
3.4 SIMD 并行优化分析	11
3.4.1 数据对齐	11
3.4.2 向量化 Montgomery 乘积	11
3.4.3 向量化蝴蝶变换	12
3.5 越界问题处理	12
4 实验和结果分析	13
4.1 Profiling	13
4.1.1 Perf 多项指标分析	13
4.1.2 和朴素多项式乘法性能对比	14
4.2 一些个人思考及未来优化部分	14

1 问题描述

1.1 期末选题

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。

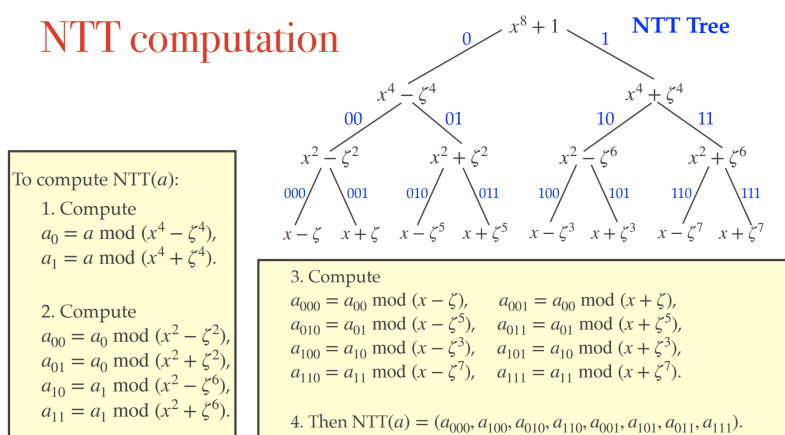


图 1.1: NTT 计算流程图 [2]

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 本次题目选题

1.2.1 实验要求

本实验主要探索在 SIMD 指令集下对 NTT 进行优化实现的可行性与效果。NTT 的 SIMD 优化需要解决两个主要难点:

1. NEON 等 SIMD 指令不支持取模运算, 因此需要手动实现模运算逻辑
2. 在递归实现 NTT 时, 循环主体涉及大量蝶形变换, 向量内部的数据需进行手动交换和重排。

实验的基础部分是完成蝶形运算的向量化实现, 这部分较为简单, 配合已有的 NTT 代码即可完成。相比之下, 模乘操作是 NTT 中最为耗时的一环, 但由于普通模乘难以直接向量化, 因此实验中引入 Montgomery 模乘算法来提升效率。该算法通过引入与模数相关的参数 r , 将 $a \times b \times r^{-1} \bmod p$ 的运算转化为一列加法、减法和位运算, 从而避免使用直接取模操作, 适合在 SIMD 环境中实现。

在完成基本优化的基础上, 探索更高级的 NTT 优化方案, 本次实验我采用四分 NTT SIMD 指令优势的变体算法, 进一步提高性能。

2 NTT 算法

2.1 FFT、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：

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

FFT 通过分解为：

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

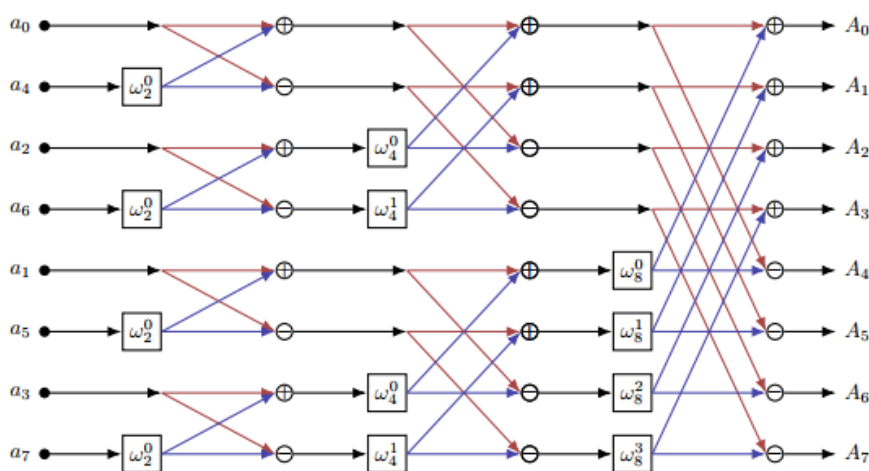


图 2.2: FFT 算法解释图

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。

2.1.2 FFT, NTT 计算开销对比

这里我通过 Leetcode 第 43 题字符串相乘运行程序，对比 FFT 和 NTT 之间的开销，结果如下：



图 2.3: FFT 和 NTT 程序性能对比

可以看到在执行用时分布和内存消耗分布上，NTT 都比 FFT 有更加明显的优势。

2.2 NTT 算法分析

2.2.1 数论前置知识

欧拉函数和欧拉定理 欧拉函数 $\varphi(n)$ 表示小于等于 n 且与 n 互素的正整数的个数。特别地，当 n 是素数时，有：

$$\varphi(n) = n - 1.$$

欧拉定理指出：对于任意整数 $a \in \mathbb{Z}$ 和正整数 $m \in \mathbb{N}^*$ ，若 $\gcd(a, m) = 1$ ，则有：

$$a^{\varphi(m)} \equiv 1 \pmod{m}.$$

在 NTT 中，通常需要在模一个质数 p 的意义下进行运算。此时， $\varphi(p) = p - 1$ ，欧拉定理保证了对于任意与 p 互素的 a ，有 $a^{p-1} \equiv 1 \pmod{p}$ 。这一性质使得我们可以选取一个原根 g ，通过 $g^{(p-1)/n}$ 构造出 n 次单位根，从而实现模 p 的快速数论变换。

费马小定理 费马小定理指出：若 p 为素数，且 $\gcd(a, p) = 1$ ，则有

$$a^{p-1} \equiv 1 \pmod{p}.$$

等价地，对于任意整数 a ，有

$$a^p \equiv a \pmod{p}.$$

在数论变换中，为了进行模 p 意义下的除法操作，常常要求某个数的逆元。由费马小定理可得，当 a 与 p 互素时，其模逆元为

$$a^{-1} \equiv a^{p-2} \pmod{p}.$$

这使得我们可以通过快速幂算法高效地求出逆元，从而完成 NTT 中对多项式系数的归一化处理。

阶 由欧拉定理可知，若 $\gcd(a, m) = 1$ ，则存在最小的正整数 n 使得

$$a^n \equiv 1 \pmod{m}.$$

这个最小的 n 称为 a 模 m 的**阶**，记作 $\delta_m(a)$ 或 $\text{ord}_m(a)$ 。

阶具有以下两个重要性质：

- ▶ **性质 1:** $a, a^2, \dots, a^{\delta_m(a)}$ 在模 m 意义下两两不同余，之后将进入周期性重复；
- ▶ **性质 2:** 若 $a^n \equiv 1 \pmod{m}$ ，则 $\delta_m(a) \mid n$ ，从而可推得若 $a^p \equiv a^q \pmod{m}$ ，则 $p \equiv q \pmod{\delta_m(a)}$ 。

在 NTT 中，为了构造 n 次单位根 ω ，需选择一个模 p 的原根 g ，使得 $\text{ord}_p(g) = p - 1$ ，再令 $\omega = g^{(p-1)/n}$ ，确保 ω 的阶为 n ，从而能够生成 n 个不同的单位根，满足数论变换的需求。

原根 设 $n \in \mathbb{N}^*$ ， $g \in \mathbb{Z}$ ，若 $\gcd(g, m) = 1$ 且 $\delta_m(g) = \varphi(m)$ ，则称 g 为模 m 的**原根**。

当 m 为素数时，有 $\varphi(m) = m - 1$ ，此时 $g^i \pmod{m}$ 对于 $0 < i < m$ 两两不同。

原根个数: 若模数 m 存在原根，则原根的个数为 $\varphi(\varphi(m))$ 。

原根存在定理: 模数 m 存在原根当且仅当 $m = 2, 4, p^a, 2p^a$ ，其中 p 为奇素数， $a \in \mathbb{N}^*$ 。

原根的性质:

- ▶ **不重性:** $\forall 0 \leq i < j < \varphi(p)$ ，有 $g^i \not\equiv g^j \pmod{p}$ ；
- ▶ **折半性:** 定义 $g_n = g^{\frac{p-1}{n}}$ ，则有 $g_{an}^{ak} \equiv g_n^k \pmod{p}$ ；
- ▶ **对称性:** $g_{2n}^{k+n} \equiv -g_{2n}^k \pmod{p}$ ；
- ▶ **求和性:** $\sum_{i=0}^{n-1} (g_n)^{ki} \equiv n[k=0] \pmod{p}$ ，其中 $[k=0] = 1$ ，否则为 0。

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}

2.3 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 $A = [a_0, a_1, \dots, a_{n-1}]$, 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**

$m \leftarrow 2^s$ $g_m \leftarrow g^{n/m} \pmod{p}$ **for** $k = 0$ **to** $n - 1$ m **do**

$\varphi \leftarrow 1$ **for** $j = 0$ **to** $\frac{m}{2} - 1$ **do**

$t \leftarrow \varphi \cdot P[k + j + \frac{m}{2}] \pmod{p}$ $u \leftarrow P[k + j] \pmod{p}$ $P[k + j] \leftarrow (u + t) \pmod{p}$ $P[k + j + \frac{m}{2}] \leftarrow (u - t) \pmod{p}$ $\varphi \leftarrow (\varphi \cdot g_m) \pmod{p}$

return P

NTT 串行算法的整体代码如下：

```
1 #include <iostream>
2 #include <cstring>
3 using namespace std;
4 using int64 = long long;
5 const int MAXN = 1 << 23; // 8,388,608, 足以为 lim 达到 2^23
6 const int64 p = 998244353; // NTT友好的模数: 998244353 = 2^23 * 119 + 1
7 int64 A[MAXN], B[MAXN];
8 int rev[MAXN], ans[MAXN];
9 int lim = 1, log2_lim = 0;
10 inline int64 pow(int64 b, int64 e) {
11     int64 res = 1;
12     while (e) {
13         if (e & 1) res = res * b % p; // 模乘法
14         b = b * b % p; // 基数平方
15         e >>= 1;
16     }
17     return res;
18 }
19 inline void swap(int64& c1, int64& c2) {
20     int64 t = c1; c1 = c2; c2 = t;
21 }
22 void NTT(int64* c, int flag) {
23     for (int i = 0; i < lim; i++)
24         if (i < rev[i])
25             swap(c[i], c[rev[i]]); // 位反转置换
26     for (int j = 1; j < lim; j <= 1) {
27         int j2 = j << 1;
28         int64 w = pow(3, (p - 1) / j2); // 原根的幂
29         if (flag == -1) w = pow(w, p - 2); // 逆变换的逆元
30         for (int k = 0; k < lim; k += j2) {
31             int64 t = 1;
32             for (int l = 0; l < j; l++, t = t * w % p) {
33                 int k1 = k + l;
34                 int64 Nx = c[k1], Ny = t * c[k1 + j] % p;
35                 c[k1] = (Nx + Ny) % p; // 蝶形运算: 偶数部分
36                 c[k1 + j] = (Nx - Ny + p) % p; // 蝶形运算: 奇数部分, 确保 >= 0
37             }
38         }
39     }
40     if (flag == -1) {
```

```
41     int64 inv = pow(lim, p - 2); // lim 的乘法逆元
42     for (int i = 0; i < lim; i++)
43         c[i] = c[i] * inv % p; // 归一化逆 NTT
44 }
45 }
46 int main() {
47     string s1_str, s2_str;
48     cin >> s1_str >> s2_str;
49     int len_A = s1_str.length(), len_B = s2_str.length();
50     for (int i = 0; i < len_A; i++)
51         A[i] = s1_str[len_A - 1 - i] - '0'; // 反转并转换为数字
52     for (int i = 0; i < len_B; i++)
53         B[i] = s2_str[len_B - 1 - i] - '0'; // 反转并转换为数字
54     int len_AB = len_A + len_B;
55     while (lim < len_AB) lim <= 1, log2_lim++; // 将 lim 设置为下一个 2 的幂
56     for (int i = 1; i < lim; i++)
57         rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (log2_lim - 1)); // 预计算 rev
58     NTT(A, 1); NTT(B, 1); // 正向 NTT
59     for (int i = 0; i < lim; i++)
60         A[i] = A[i] * B[i] % p; // 点对点乘法
61     NTT(A, -1); // 逆 NTT
62     for (int i = 0; i < lim; i++) {
63         ans[i] += A[i];
64         if (ans[i] >= 10) {
65             ans[i + 1] += ans[i] / 10; // 进位传播
66             ans[i] %= 10; // 保留 0-9 的数字
67         }
68     }
69     int max_lim = lim - 1;
70     while (max_lim > 0 && ans[max_lim] == 0) max_lim--; // 找到最高的非零数字
71     if (max_lim == -1) cout << "0"; // 处理零乘积
72     else {
73         for (int i = max_lim; i >= 0; i--)
74             cout << ans[i]; // 从最高有效位到最低有效位打印结果
75     }
76     return 0;
77 }
```

自此，关于 NTT 算法的介绍和分析就结束了，下面的部分将根据实验要求和上面的叙述，进行代码的优化和 SIMD 并行操作。

3 NTT 算法优化

3.1 Montgomery 模乘

3.1.1 理论分析

由于 Neon 的向量化指令不支持直接取模运算，本实验采用 Montgomery 规约方法，将模乘转化为适合 SIMD 优化的操作。Montgomery 规约通过引入一个大于模数 n 且与之互质的参数 r ，将数值映射到 Montgomery 空间。在此空间中，加减法与常规一致，而乘法则需经过特殊处理。

具体而言，若 x, y 在 Montgomery 空间中表示为 $\bar{x} = x \cdot r \bmod n$ 和 $\bar{y} = y \cdot r \bmod n$ ，则它们的乘积可以通过下式计算：

$$\bar{x} * \bar{y} = (\bar{x} \cdot \bar{y} \cdot r^{-1}) \bmod n$$

为了避免显式除法操作，使用预先计算好的参数 n' ，将模乘规约简化为加减、乘法和移位运算。例如，对于 32 位情况，规约过程可以表示为：

$$\begin{aligned} q &= (x \times n') \bmod r \\ m &= q \times n \\ y &= (x - m) \gg 32 \end{aligned}$$

若 $x < m$ ，则输出 $y + n$ ，否则输出 y 。这一过程中的所有操作均可使用 SIMD 指令优化，从而大幅提升大规模 NTT 计算的性能。

3.1.2 参数预计算

在算法初始化阶段，系统预计算关键参数以支持快速 Montgomery 运算。选定 $R = 2^{32}$ 作为规约基数，该值匹配处理器的 32 位字长特性。通过扩展欧几里得算法计算模数 p 的逆元 $p^{-1} \bmod R$ ，随后生成核心参数 $\text{mont_factor} = -p^{-1} \bmod R$ 。该参数将用于后续乘法运算中的快速约简计算，其预计算过程如代码所示：

```
1 const u64 R = 1ULL << 32;
2 const u32 R_mod = R % p;
3 const u64 inv = mod_inverse(p, R);
4 const u32 mont_factor = (u32)(R - inv);
```

参数间的数学关系确保了后续运算的正确性。其中 $R \cdot R^{-1} \equiv 1 \bmod p$ 的成立性通过逆元算法保证，而 mont_factor 的构造使得乘法运算中的中间值约简无需实际除法操作。

3.1.3 标量运算优化

基础标量运算采用经典的 Montgomery 乘法四步法实现。对于输入值 $a, b \in \mathbb{Z}_p$ ，其 Montgomery 乘积计算过程为：

$$t = a \cdot b \tag{1}$$

$$m = (t \cdot \text{mont_factor}) \bmod R$$

$$u = \left\lfloor \frac{t + m \cdot p}{R} \right\rfloor$$

$$\text{Result} = \begin{cases} u - p, & \text{if } u \geq p \\ u, & \text{otherwise} \end{cases}$$

对应的代码实现通过 64 位中间变量避免溢出，并利用位运算替代除法。关键代码段展示了对乘积的高效处理：

```

1 u32 montgomery_mul_scalar(u32 a, u32 b, u32 mod, u32 mont_factor) {
2     u64 t = (u64)a * b;
3     u32 m = (u32)t * mont_factor;
4     u64 u = (t + (u64)m * mod) >> 32;
5     return (u >= mod) ? u - mod : u;
6 }

```

此实现将传统模乘的 1 次除法、1 次乘法优化为 2 次乘法和 1 次移位操作。

3.1.4 数据格式转换流程

系统采用分层转换策略确保数据一致性：

- ▶ **输入阶段**：将多项式系数 a_i 转换为 Montgomery 形式 $a'_i = a_i \cdot R \bmod p$ ，消除后续运算中的重复转换开销
- ▶ **运算阶段**：所有 NTT 蝶形运算均在 Montgomery 数域内进行，避免传统实现中的频繁格式转换
- ▶ **输出阶段**：通过 $\text{mont_mul}(a'_i, 1)$ 将结果转换回标准数域，保证与其他系统的兼容性

完整的数据流可抽象为：

$$\mathbb{Z}_p \xrightarrow{\times R} \mathbb{M}_p \xrightarrow{\text{NTT}} \mathbb{M}_p \xrightarrow{\times R^{-1}} \mathbb{Z}_p$$

其中 \mathbb{M}_p 表示 Montgomery 数域，该设计使得运算在 \mathbb{M}_p 内完成，显著减少格式转换次数。

3.2 蝴蝶变换

Algorithm 2 Butterfly Operation with Montgomery Multiplication

Input: Array $A = [a_0, a_1, \dots, a_{n-1}]$, length n , modulus m , twiddle factors W_n , precomputed Montgomery factors wn_mont , Montgomery factor $mont_factor$

Output: Transformed array A after butterfly operation

for $\text{mid} = 1$ **to** limit $\text{mid} \ll= 1$ **do**

for $j = 0$ **to** $\text{limit} - 1$ 2 mid **do**

$w \leftarrow 1$ **for** $k = 0$ **to** $\text{mid} - 1$ **do**

$x \leftarrow a_{j+k}$ $y \leftarrow a_{j+\text{mid}+k}$ $y_w \leftarrow \text{montgomery_mul_scalar}(y, wn_mont[\text{offset} + k], m, mont_factor)$ $u \leftarrow (x + y_w) \bmod m$ $v \leftarrow (x - y_w + m) \bmod m$ $a_{j+k} \leftarrow u$

$a_{j+\text{mid}+k} \leftarrow v$ $w \leftarrow (w \cdot W_n) \bmod m$

return A

在整个蝶形计算过程中，我们按照如下步骤进行：

首先，使用分层分治的方法逐步处理数据。外层循环中，变量 mid 从 $2^0, 2^1, 2^2, \dots$ 开始，每次翻倍，直到覆盖整个数组长度 limit 。在每一层中，将数组划分为若干长度为 2mid 的小段，分别对每段执行蝶形变换。在每个小段处理前，旋转因子 w 初始化为单位元 1，其中 W_n 是模 m 意义下的本原单位根。随着内层循环 k 推进，旋转因子更新为

$$w \leftarrow (w \cdot W_n) \bmod m,$$

以保证不同位置对应的旋转角度正确。

接下来进行数据加载与旋转：对于索引对 (j, k) ，取出原数组元素

$$x = a[j + k], \quad y = a[j + k + \text{mid}],$$

并调用

$$y_w = \text{montgomery_mul_scalar}(y, \text{wn_mont}[\text{offset} + k], m, \text{mont_factor})$$

蝶形合并操作更新公式为

$$\begin{cases} u = (x + y_w) \bmod m, \\ v = (x - y_w + m) \bmod m, \end{cases}$$

确保结果始终落在 $[0, m)$ 范围内。计算结果写回数组：

$$a[j + k] \leftarrow u, \quad a[j + k + \text{mid}] \leftarrow v.$$

每层蝶形操作结束后，更新旋转因子表偏移量：

$$\text{offset} \leftarrow \text{offset} + \text{mid}.$$

3.3 四分 NTT

在函数 `ntt_butterfly_neon` 中，当每个阶段的步长 $\text{mid} \geq 4$ 时，使用 NEON 向量指令一次处理 4 个蝶形对：

```

1 if (mid >= 4) {
2     for (int k = 0; k < mid; k += 4) {
3         // 加载 4 个元素向量 X, Y 和扭转因子 W
4         uint32x4_t x = vld1q_u32(a + j + k);
5         uint32x4_t y = vld1q_u32(a + j + mid + k);
6         uint32x4_t w = vld1q_u32(wn_mont + offset + k);
7
8         // 并行 Montgomery 乘法 YW = Y * W mod p
9         uint32x4_t yw = montgomery_mul_neon(y, w, mod, mont_factor);
10
11        // 计算 u = (x + yw) mod p
12        uint32x4_t sum = vaddq_u32(x, yw);
13        sum = vsubq_u32(sum, vdupq_n_u32(mod));

```

```

14     uint32x4_t sum_mask =
15         vcgeq_u32(x, vsubq_u32(vdupq_n_u32(mod), yw));
16     sum = vbslq_u32(sum_mask, sum, vaddq_u32(x, yw));
17
18     // 计算  $v = (x - yw) \bmod p$ 
19     uint32x4_t diff = vsubq_u32(x, yw);
20     diff = vaddq_u32(diff, vdupq_n_u32(mod));
21     uint32x4_t diff_mask =
22         vcgeq_u32(diff, vdupq_n_u32(mod));
23     diff = vbslq_u32(diff_mask,
24                     vsubq_u32(diff, vdupq_n_u32(mod)),
25                     diff);
26
27     // 并行写回
28     vst1q_u32(a + j + k, sum);
29     vst1q_u32(a + j + mid + k, diff);
30 }
31 }

```

对应的数学表达如下。记向量

$$\mathbf{X} = (x_0, x_1, x_2, x_3), \quad \mathbf{Y} = (y_0, y_1, y_2, y_3), \quad \boldsymbol{\omega} = (\omega_0, \omega_1, \omega_2, \omega_3).$$

则对每个分量 $i = 0, 1, 2, 3$, 标准蝶形操作为

$$\begin{cases} u_i = x_i + y_i \omega_i \pmod{p}, \\ v_i = x_i - y_i \omega_i \pmod{p}. \end{cases}$$

在向量化形式中, 记逐分量的 Montgomery 乘积为

$$\mathbf{YW} = \mathbf{Y} \odot \boldsymbol{\omega},$$

则一次指令即可完成四个蝶形:

$$\mathbf{U} = (\mathbf{X} + \mathbf{YW}) \bmod p, \quad \mathbf{V} = (\mathbf{X} - \mathbf{YW}) \bmod p,$$

并通过条件选择指令保证结果落在区间 $[0, p)$ 内。

这种四路并行蝶形的优势在于:

- ▶ **减少循环开销**: 每次迭代处理 4 个元素, 循环次数减小为原来的四分之一。
- ▶ **高效内存访问**: 一次加载/存回 128 bit (4×32 bit) 的数据。
- ▶ **流水线并行**: 充分利用 ARM NEON 的 SIMD 单元, 同时执行多次乘加运算。

由此, 整体 NTT 时间复杂度依然为 $O(n \log n)$, 但常数因子显著下降, 实现了高效的“四分 NTT”优化。

3.4 SIMD 并行优化分析

在本实现中，我们利用 ARM NEON 的 128 位向量寄存器，通过 SIMD 指令对 NTT 的核心步骤进行加速。

3.4.1 数据对齐

首先，所有需要通过 NEON 加载和存储的数组都使用如下接口以 32 字节对齐方式分配，确保每次 `vld1q_u32` 和 `vst1q_u32` 均在对齐边界上完成，从而最大化 SIMD 内存带宽：

```
1 static void* aligned_malloc(size_t alignment, size_t size) {
2     void* ptr = nullptr;
3     if (posix_memalign(&ptr, alignment, size) != 0)
4         ptr = nullptr;
5     return ptr;
6 }
```

3.4.2 向量化 Montgomery 乘积

在蝴蝶运算中，当子块长度 $\text{mid} \geq 4$ 时，我们一次性加载四个相邻元素向量 \mathbf{x} 和 \mathbf{y} ，以及旋转因子向量 \mathbf{w} ，均为 `uint32x4_t` 类型，并调用下述 SIMD 版 Montgomery 乘积函数：

```
1 uint32x4_t montgomery_mul_neon(uint32x4_t a, uint32x4_t b,
2                                 u32 mod, u32 mont_factor) {
3     // 拆分低 / 高两对以做并行 64×32 位乘法
4     uint32x2_t a_lo = vget_low_u32(a), a_hi = vget_high_u32(a);
5     uint32x2_t b_lo = vget_low_u32(b), b_hi = vget_high_u32(b);
6     // vmull_u32 同时计算两对乘积，得到 uint64x2_t 向量
7     uint64x2_t t_lo = vmull_u32(a_lo, b_lo),
8         t_hi = vmull_u32(a_hi, b_hi);
9     // vmovn_u64 提取低 32 位，vcombine_u32 合并为四路中间值
10    uint32x4_t t_low = vcombine_u32(vmovn_u64(t_lo), vmovn_u64(t_hi));
11    // 并行计算 m_i = (t_i mod 2^32) * mont_factor
12    uint32x4_t m_vec = vmulq_n_u32(t_low, mont_factor);
13    // 并行计算 t_i + m_i * mod
14    uint64x2_t prod_lo = vmull_u32(vget_low_u32(m_vec), vdup_n_u32(mod));
15    uint64x2_t prod_hi = vmull_u32(vget_high_u32(m_vec), vdup_n_u32(mod));
16    uint64x2_t sum_lo = vaddq_u64(t_lo, prod_lo),
17        sum_hi = vaddq_u64(t_hi, prod_hi);
18    // vshrn_n_u64 右移 32 位提取高半字，再用 vbslq_u32 条件减法完成模约化
19    uint32x4_t u_vec = vcombine_u32(vshrn_n_u64(sum_lo, 32),
20                                    vshrn_n_u64(sum_hi, 32));
21    uint32x4_t mask = vcgeq_u32(u_vec, vdupq_n_u32(mod));
22    return vbslq_u32(mask,
```

```

23         vsubq_u32(u_vec, vdupq_n_u32(mod)),
24         u_vec);
25 }

```

该函数一次调用即可完成四对 (a_i, b_i) 的乘法、累加以及条件减模，从而获得很大程度的 SIMD 并行性能提升。

3.4.3 向量化蝴蝶变换

在 `ntt_butterfly_neon` 中，我们将四路 Montgomery 乘积结果与原向量并行求和与求差，并在同一循环体内通过 SIMD 指令完成模约化和写回：

```

1  for (int k = 0; k < mid; k += 4) {
2      uint32x4_t x = vld1q_u32(a + j + k);
3      uint32x4_t y = vld1q_u32(a + j + mid + k);
4      uint32x4_t w = vld1q_u32(wn_mont + offset + k);
5      uint32x4_t yw = montgomery_mul_neon(y, w, mod, mont_factor);
6      // 并行加法与模约化
7      uint32x4_t sum = vaddq_u32(x, yw);
8      uint32x4_t sum_mask = vcgeq_u32(sum, vdupq_n_u32(mod));
9      sum = vbslq_u32(sum_mask, vsubq_u32(sum, vdupq_n_u32(mod)), sum);
10     // 并行减法与模约化
11     uint32x4_t diff = vsubq_u32(x, yw);
12     diff = vaddq_u32(diff, vdupq_n_u32(mod));
13     uint32x4_t diff_mask = vcgeq_u32(diff, vdupq_n_u32(mod));
14     diff = vbslq_u32(diff_mask, vsubq_u32(diff, vdupq_n_u32(mod)), diff);
15     vst1q_u32(a + j + k, sum);
16     vst1q_u32(a + j + mid + k, diff);
17 }

```

这样，蝴蝶操作的循环次数被缩减为原来的 $1/4$ ，SIMD 指令在一次循环内完成四路并行计算。当 $\text{mid} < 4$ 时，出于安全考虑，算法会自动回退到标量版本，从而避免向量加载越界。通过以上对齐分配、Montgomery 乘法及蝴蝶运算的三大 SIMD 优化，NTT 主循环在理论上获得了很好的性能提升。

3.5 越界问题处理

在引入第四个模数 $p_4 = 263882790666241 > 2^{32}$ 后，原有基于 32 位整型的实现已无法正确表示模数及其中间乘积。我们的解决方案是在整个 NTT 流程中将 p_4 相关的变量全部升级为 64 位类型，并利用 128 位中间计算保证乘法精度。在后续期末大作业中，会实现大模数 NTT 更加准确的算法。

```

多项式乘法结果正确
average latency for n = 131072 p = 469762049 : 64.5254 (us)
多项式乘法结果错误
average latency for n = 131072 p = 2147483647 : 62.0255 (us)

```

图 3.4: 模数为 263882790666241 发生错误

4 实验和结果分析

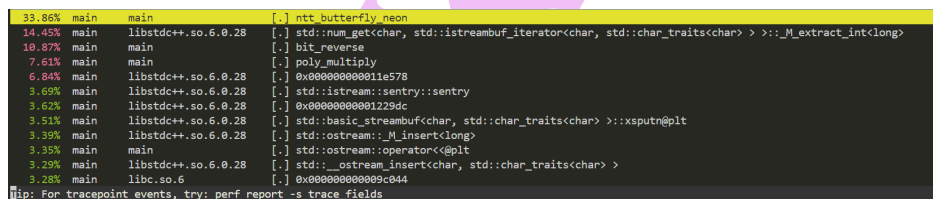
4.1 Profiling

4.1.1 Perf 多项指标分析

下表展示了使用 `perf record + perf report` 对程序 `main` 采样分析后的热点函数分布：

表 1: 热点函数采样结果

% Samples	进程名	对象	函数名 / 符号
33.86%	main	main	ntt_butterfly_neon
14.45%	main	libstdc++.so.6.0.28	std::num_get<char>::_M_extract_int<long>
10.87%	main	main	bit_reverse
7.61%	main	main	poly_multiply
6.84%	main	libstdc++.so.6.0.28	(地址符号)
3.69%	main	libstdc++.so.6.0.28	std::istream::sentry::sentry
3.62%	main	libstdc++.so.6.0.28	(地址符号)
3.51%	main	libstdc++.so.6.0.28	std::basic_streambuf<...>::xsputn@plt
3.39%	main	libstdc++.so.6.0.28	std::ostream::_M_insert<long>
3.35%	main	main	std::ostream::operator<<@plt
3.29%	main	libstdc++.so.6.0.28	std::_ostream_insert<char, ...>
3.28%	main	libc.so.6	(地址符号)



```

33.86% main main main [.] ntt_butterfly_neon
14.45% main libstdc++.so.6.0.28 [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
10.87% main main main [.] bit_reverse
7.61% main main main [.] poly_multiply
6.84% main libstdc++.so.6.0.28 [.] 0x00000000011e578
3.69% main libstdc++.so.6.0.28 [.] std::istream::sentry::sentry
3.62% main libstdc++.so.6.0.28 [.] 0x0000000001229dc
3.51% main libstdc++.so.6.0.28 [.] std::basic_streambuf<char, std::char_traits<char> >::xsputn@plt
3.39% main libstdc++.so.6.0.28 [.] std::ostream::_M_insert<long>
3.35% main main main [.] std::ostream::operator<<@plt
3.29% main libstdc++.so.6.0.28 [.] std::_ostream_insert<char, std::char_traits<char> >
3.28% main libc.so.6 [.] 0x000000000009c044

```

图 4.5: 热点函数分布

从表中可以看出，程序的时间主要消耗在蝴蝶变换和位反转上：

- ▶ `ntt_butterfly_neon`: 33.86%
- ▶ `bit_reverse`: 10.87%
- ▶ `poly_multiply`: 7.61%

这三者合计超过 50%，是优化的重点，原因我会在下面的个人思考中具体阐述。

下表展示了执行命令

```
1 perf stat -e cycles,instructions,cache-misses ./main
```

得到的关键性能计数器结果：

- ▶ **IPC = 2.20**：表示平均每个周期能执行约 2.2 条指令，说明流水线利用率较高。
- ▶ **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。

表 2: 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	

- **用户态 vs 内核态**: 用户态耗时约 0.1275 s, 内核态 (系统调用、上下文切换等) 仅约 0.0164 s, 占比不到 11%, 说明程序主要在用户态计算, 系统开销较低。

4.1.2 和朴素多项式乘法性能对比

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
并行 + 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

- **小规模输入 ($n = 4$):**

$$T_{\text{朴素}} = 0.00021 \text{ us}, \quad T_{\text{并行 + 优化}} = 0.01715 \text{ us}$$

并行初始化开销使得并行版本比朴素版本慢约 80 \times 。

- **大规模输入 ($n = 131\,072$):**

算法	$p = 7.34 \times 10^6$	$p = 1.048 \times 10^8$	$p = 4.6976 \times 10^8$
朴素	95,673.6 us	101,832 us	106,271 us
并行 + 优化	65.095 us	64.834 us	64.471 us

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。

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, 使用并行 + 优化算法, 可获得千倍级性能提升。

4.2 一些个人思考及未来优化部分

对于长度为 $N = 2^m$ 的多项式系数序列

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

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

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

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：

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

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 $\ell = 1, 2, \dots, m$ ，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 $\ell = m, m-1, \dots, 1$ ，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，后续将使用此方法进行进一步优化。

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.