南　　开　　大　　学

计　算　机　学　院

并行程序设计 Lab2:SIMD 编程

NTT 快速数论变换

侯嘉栋

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 4 月 28 日

摘要

本文探讨了快速数论变换(NTT)的实现及其并行优化。首先介绍了快速傅里叶变换(FFT)的基础理论，进一步阐述了 NTT 在有限域上的扩展应用，分析了串行 NTT 算法的实现及性能瓶颈。随后，本文通过引入 Montgomery 规约、浮点数近似取模以及 ARM NEON SIMD 向量化技术，对 NTT 算法进行了性能优化，并设计了 DIF 与 DIT 混合结构以进一步提高计算效率。性能测试表明，采用 SIMD 优化后 NTT 算法的计算速度显著提升，尤其在大规模输入条件下性能优势更加明显。

关键字：快速数论变换（NTT）；SIMD；Montgomery 规约；NEON 优化；并行计算

Github 仓库地址:NKU-Parallel-Computing

目录

一、 概述

(一) 第一节：FFT 与 NTT 原理简介以及串行 NTT 实现

首先明确我们的问题：计算两个 n 次多项式的乘法，而朴素的算法复杂度为 $O(n^2)$, 快速傅里叶变化可以达到 $O(nlog(n))$, 其利用的单位根性质，而 NTT 将单位根换为原根在有限域上操作，完全避免了浮点数计算和舍入误差并且可以并行计算。

1. FFT

离散傅里叶变换（Discrete Fourier transform，DFT）:
$X_k = \sum_{n=0}^{N-1} x_n e^{-i\frac{2\pi}{N}kn}$ 可以记为：$\hat{x} = \mathcal{F}x$
逆离散傅里叶变换 (IDFT):
$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{i\frac{2\pi}{N}kn}$ 可以记为：$x = \mathcal{F}^{-1}\hat{x}$
我们可以将离散傅里叶变化用矩阵表示:

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \vdots \\ X_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \alpha & \alpha^2 & \cdots & \alpha^{N-1} \\ 1 & \alpha^2 & \alpha^4 & \cdots & \alpha^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{N-1} & \alpha^{2(N-1)} & \cdots & \alpha^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix} \text{ 其中 } \alpha = e^{-i\frac{2\pi}{N}} \text{。}$$

快速傅里叶变化: 核心思想是利用分治法。

将 $f(x)$ 用新函数表示为：$f(x) = G\left(x^2\right) + x \times H\left(x^2\right)$

利用偶数次单位根的性质 $\omega_n^i = -\omega_n^{i+n/2}$，和 $G\left(x^2\right)$ 和 $H\left(x^2\right)$ 是偶函数, 我们知道在复平面上 ω_n^i 和 $\omega_n^{i+n/2}$ 的 $G\left(x^2\right)$ 和 $H\left(x^2\right)$ 对应的值相同。得到:

$$f(\omega_n^k) = G((\omega_n^k)^2) + \omega_n^k \times H((\omega_n^k)^2) = G(\omega_n^{2k}) + \omega_n^k \times H(\omega_n^{2k}) = G(\omega_{n/2}^k) + \omega_n^k \times H(\omega_{n/2}^k)$$

$$f\left(\omega_n^{k+n/2}\right) = G\left(\omega_n^{2k+n}\right) + \omega_n^{k+n/2} \times H\left(\omega_n^{2k+n}\right) = G(\omega_n^{2k}) - \omega_n^k \times H(\omega_n^{2k}) = G(\omega_{n/2}^k) - \omega_n^k \times H(\omega_{n/2}^k)$$

因此我们求出了 $G\left(\omega_{n/2}^k\right)$ 和 $H\left(\omega_{n/2}^k\right)$ 后, 就可以同时求出 $f\left(\omega_n^k\right)$ 和 $f\left(\omega_n^{k+n/2}\right)$ 。于是对 G 和 H 分别递归 DFT 即可。FFT

而 FFT 的实现主要可以分为拆分和合并两个步骤，拆分可以通过位逆序置换实现，而合并可以通过蝴蝶变化实现。

位逆序置换: 可以通过递归实现。在求 R(x) 时，可以根据 $R(\lfloor \frac{x}{2} \rfloor)$ 的值确定。

位逆序置换

```
void get_rev(int *rev, int lim) {
  for (int i = 0; i < lim; ++i) {
      rev[i] = (rev[i >> 1] >> 1) | ((i & 1) ? (lim >> 1) : 0);
  }
}
```

蝴蝶变化1: 使用位逆序置换后, 对于给定的 n, k :
$G\left(\omega_{n/2}^k\right)$ 的值存储在数组下标为 k 的位置, $H\left(\omega_{n/2}^k\right)$ 的值存储在数组下标为 $k + \frac{n}{2}$ 的位置。
$f\left(\omega_n^k\right)$ 的值将存储在数组下标为 k 的位置, $f\left(\omega_n^{k+n/2}\right)$ 的值将存储在数组下标为 $k + \frac{n}{2}$ 的位置。
因此可以直接在数组下标为 k 和 $k + \frac{n}{2}$ 的位置进行覆写, 而不用开额外的数组保存值。(参考代码可以 oi-wiki 中找到)

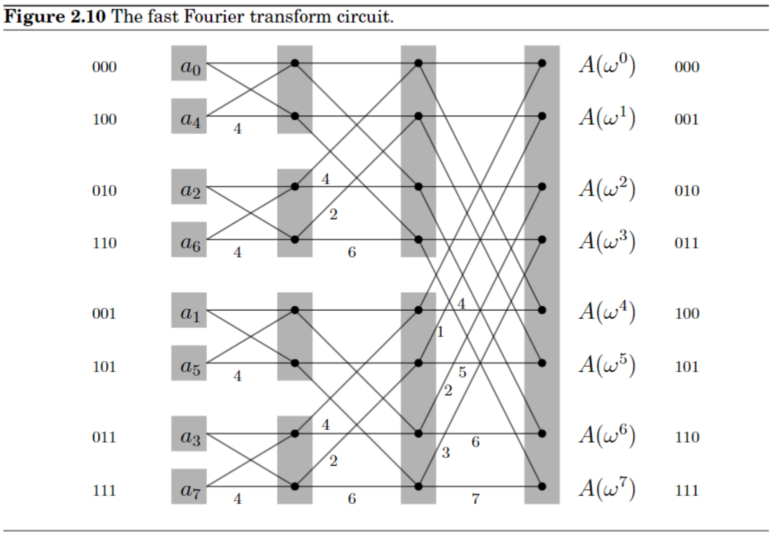

图 1: n=8 时蝴蝶变化示例

2. NTT

FFT 之所以能用是因为它在复数域上选定了 n 个特殊的点，也就是 n 个单位根。满足了某类特殊性质，才使得 FFT 可用。所以我们只需要在取模域上，找到与单位根等价的数，也就使得 NTT 可用。(具体满足的性质和证明可以参考算法学习笔记 (19)：NTT(快速数论变换)和快速数论变换)

NTT 中设计到模数的选取，快速数论变换（FNTT）中提供了常见的模数和原根。

3. 串行 NTT 的实现

串行 NTT 的实现

```
void ntt(int *a, int lim, int opt, int p) {
    int rev[lim];
    memset(rev, 0, sizeof(int) * lim);
    get_rev(rev, lim);
    for (int i = 0; i < lim; ++i) {
        if (i < rev[i]) std::swap(a[i], a[rev[i]]);}
    for (int len = 2; len <= lim; len <<= 1) {
        int m = len >> 1;
        int wn = qpow(3, (p - 1) / len, p);
        if (opt == -1) wn = qpow(wn, p - 2, p);
        for (int i = 0; i < lim; i += len) {
            int w = 1;
            for (int j = 0; j < m; ++j) {
                int u = a[i + j];
                int v = 1LL * a[i + j + m] * w % p;
                a[i + j] = (u + v) % p;
                a[i + j + m] = (u - v + p) % p;
                w = 1LL * w * wn % p;}}}
    if (opt == -1) {
```

```
        int inv = qpow(lim, p - 2, p);
        for (int i = 0; i < lim; ++i) {
            a[i] = 1LL * a[i] * inv % p;}}}
```

我们测试串行 NTT 的性能:

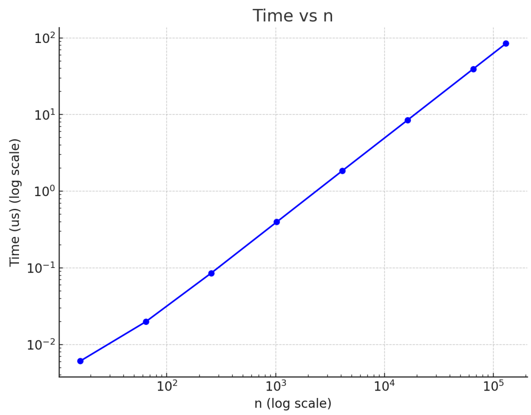

图 2: 测试不同输入规模下的性能 (p=7340033)

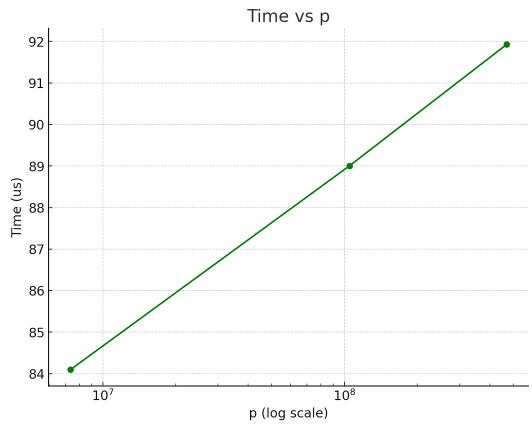

图 3: 测试不同模数下的性能 (n=131072)

步骤	时间 (us)	占比
位反转置换	1.80379	2.1795%
正向 NTT (A)	25.4885	30.7975%
正向 NTT (B)	26.8416	32.4324%
点乘	0.930219	1.12397%
逆 NTT	27.6971	33.4662%
总时间	82.7616	100%

表 1: NTT 各阶段性能 (n=131072, p=7340033)

模数	时间 (us)
7340033	84.0962
104857601	89.0015
469762049	91.925

表 2: 不同模数下的 NTT 性能

(二) 第二节：向量化取模

1. Montgomery 规约

优化取模的几种方法Montgomery 规约是一种高效的模乘算法，用于加速大整数模运算。其核心思想是通过引入一个辅助数 R（通常选择为 2^n，便于计算机进行位运算），将模运算转换为一系列更高效的运算。

具体来说，Montgomery 规约的工作原理如下：

1. 选择模数 m，且 $\gcd(m, R) = 1$，通常 $R = 2^n$ 且 m 为奇数。
2. 预计算 $R^2 \bmod m$ 和 m'，其中 m' 满足 $R \cdot m' \equiv -1 \pmod{R}$。
3. 对于需要计算的 $a \cdot b \bmod m$，首先将 a 和 b 转换到 Montgomery 域：$\tilde{a} = a \cdot R \bmod m$ 和 $\tilde{b} = b \cdot R \bmod m$。
4. 在 Montgomery 域中计算乘积：$\tilde{c} = \tilde{a} \cdot \tilde{b} \cdot R^{-1} \bmod m$。

5. 最后，将结果转换回原始域：$c = \tilde{c} \cdot R^{-1} \bmod m$。

关于算法的正确性可以参考博客，写的很详细 [1] 算法涉及到拓展欧几里得求逆元的内容，下面简单说一下

欧几里得算法即通过递归求出最大公约数。

拓展欧几里得算法常用于求 $ax + by = gcd(a, b)$ 的一组可行解。主要是利用了

$$
\begin{align}
ax_1 + by_1 &= gcd(a, b) \tag{1}\\
bx_2 + (a\ mod\ b)y_2 &= gcd(b, a\ mod\ b) \tag{2}\\
gcd(a, b) &= gcd(b, a\ mod\ b) \tag{3}\\
ax_1 + by_1 &= bx_2 + (a\ mod\ b)y_2 \tag{4}\\
ax_1 + by_1 &= ay_2 + bx_2 - \lfloor \frac{a}{b} \rfloor \times by_2 = ay_2 + b(x_2 - \lfloor \frac{a}{b} \rfloor y_2) \tag{5}\\
& \tag{6}
\end{align}
$$

讲 x_2, y_2 不断带入递归直至为 0. 递归 x=1,y=0 回去求解

同时，我们可以将递归改为迭代。主要是根据：

$$
\begin{cases}
ax + by & = a \\
ax_1 + by_1 & = b
\end{cases}
$$

$$
\begin{cases}
ax_1 + by_1 & = b \\
a(x - qx_1) + b(y - qy_1) & = a - qb
\end{cases}
$$

gcd and exgcd

```
int gcd(int a, int b) {
  if (b == 0) return a;
  return gcd(b, a % b);}
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
int Exgcd(int a, int b, int &x, int &y) {
  if (!b) {
    x = 1;
    y = 0;
    return a;}
  int d = Exgcd(b, a % b, x, y);
  int t = x;
  x = y;
  y = t - (a / b) * y;
  return d;}
int gcd(int a, int b, int& x, int& y) {
  x = 1, y = 0;
  int x1 = 0, y1 = 1, a1 = a, b1 = b;
  while (b1) {
    int q = a1 / b1;
    tie(x, x1) = make_tuple(x1, x - q * x1);
    tie(y, y1) = make_tuple(y1, y - q * y1);
```

```
    tie(a1, b1) = make_tuple(b1, a1 - q * b1);}
  return a1;}
```

求逆元

实际上我们可以求解形如

$$ax \equiv b \mod n$$

的线性同余方程组:

1. $ax \equiv b \mod n$ 可以写为 $ax+nk=b$ 的线性不定方程。可以看出 $a\frac{b}{gcd(a,n)}x_0+n\frac{b}{gcd(a,n)}k_0 = b$ 解出的 x_0, k_0 乘以 $\frac{b}{gcd(a,n)}$ 是我们想要的一个特解。
2. 定理 2: 若gcd$(a,n)=1$, 且x_0,k_0 为方程 ax+nk=b 的一组解, 则该方程的任意解可表示为: $x=x_0+nt$ 和 $k=k_0-at$（对任意的整数 t 成立）

exgcd 求逆元

```
void exgcd(int a, int b, int& x, int& y) {
  if (b == 0) {
    x = 1, y = 0;
    return;}
  exgcd(b, a % b, y, x);
  y -= a / b * x;}
```

Montgomery 规约

```
  // Montgomery规约的向量化实现
  class MontgomeryReducer {
  private:
      u32 mod;          // 模数
      u32 mod_inv;      // -mod^(-1) mod 2^32
      u32 r2;           // (2^64) % mod
  public:
      MontgomeryReducer(u32 mod) : mod(mod) {
          // 计算 -mod^(-1) mod 2^32
          mod_inv = inverse(mod, 1U << 31);  // 使用2^31而不是2^32避免溢出
          // 计算 (2^64) % mod
          r2 = 0;
          u64 r = 1;
          for (int i = 0; i < 64; ++i) {
              r = (r << 1) % mod;}
          r2 = r;}
      // 单个数值的Montgomery规约
      u32 reduce(u64 x) {
          u32 q = (u32)x * mod_inv;  // q = x * mod' mod 2^32
          u64 m = (u64)q * mod;      // m = q * mod
          u32 y = (x - m) >> 32;     // y = (x - m) / 2^32
          return x < m ? y + mod : y; // 保证结果非负}
      // 将数值转换到Montgomery域
      u32 to_montgomery(u32 x) {
```

```
        return reduce((u64)x * r2);}
    // 将数值从Montgomery域转换回普通域
    u32 from_montgomery(u32 x) {
        return reduce((u64)x);}
    // 在Montgomery域中进行乘法
    u32 mul(u32 a, u32 b) {
        return reduce((u64)a * b);}
    // 向量化的Montgomery乘法
    void mul_vector(u32* a, u32* b, u32* result, int n) {
        for (int i = 0; i < n; i += 4) {
            // 加载4个元素到NEON寄存器
            uint32x4_t va = vld1q_u32(a + i);
            uint32x4_t vb = vld1q_u32(b + i);
            // 存储结果
            uint32x4_t vresult;
            // 逐个处理每个元素（因为NEON不直接支持64位乘法）
            for (int j = 0; j < 4; ++j) {
                u32 ai = vgetq_lane_u32(va, j);
                u32 bi = vgetq_lane_u32(vb, j);
                u32 res = mul(ai, bi);
                vresult = vsetq_lane_u32(res, vresult, j);}
            vst1q_u32(result + i, vresult);}}};
```

重点关注向量化的 Montgomery 乘法，我们每个循环加载 4 个元素到 NEON 寄存器，之后逐个处理每个每个元素，得到结果。以下是用 Montgomery 乘法优化的蝴蝶变换。

Montgomery 规约优化蝴蝶变化

```
    for (int j = 0; j < m; j += 4) {
                if (j + 4 <= m) {
                    uint32x4_t vw = {(u32)w,
                                    reducer.mul(w, wn),
                                    reducer.mul(reducer.mul(w, wn), wn),
                                    reducer.mul(reducer.mul(reducer.mul(w, wn), wn), wn)};
                    uint32x4_t vu = vld1q_u32((u32*)(a + i + j));
                    uint32x4_t vv = vld1q_u32((u32*)(a + i + j + m));
                    uint32x4_t vvw;
                    for (int k = 0; k < 4; ++k) {
                        u32 vk = vgetq_lane_u32(vv, k);
                        u32 wk = vgetq_lane_u32(vw, k);
                        vvw = vsetq_lane_u32(reducer.mul(vk, wk), vvw, k);
                    }
                    uint32x4_t new_u, new_v;
                    for (int k = 0; k < 4; ++k) {
                        u32 uk = vgetq_lane_u32(vu, k);
                        u32 vwk = vgetq_lane_u32(vvw, k);
                        u32 sum = uk + vwk;
                        if (sum >= p) sum -= p;
                        new_u = vsetq_lane_u32(sum, new_u, k);
```

```
                u32 diff = uk >= vwk ? uk - vwk : uk + p - vwk;
                new_v = vsetq_lane_u32(diff, new_v, k);
            }
            vst1q_u32((u32*)(a + i + j), new_u);
            vst1q_u32((u32*)(a + i + j + m), new_v);
            w = reducer.mul(reducer.mul(reducer.mul(reducer.mul(w, wn), wn), wn), wn);
```

重点说明：

1. 在进行完位反转后，将所有的数据首先转化到 Montgomery 域，在接下来的蝴蝶变化种，要将原根也首先转化到 Montgomery 域，在蝴蝶变化的第一层循环（分治部分），我们初始化 1 的原根时也要将其转化位 Montgomery 域中的 1
2. 之后处理每个蝴蝶操作时，并行加载 4 个 w 值，预处理得到向量化后的 vw. 再向量化加载 vu,vv。
3. 在之后计算新的 u 和新的 v 时，要逐个计算，后并行的存回到原数组中，在这个过程中需要注意如果新的 u(即做加运算的值) 的结果大于 p，要减去 p. 如果新的 v(做减运算的值) 小于 p，要加上 p。
4. 之后用 Montgomery 乘法更新 w.
5. 额外处理不足 4 个的剩余的元素。（虽然貌似不会涉及到）
6. 最后将结果从 montgomery 域转回普通域

具体代码实现可以参考 githubNKU-Parallel-Computing仓库。

我们简单的测速，得到结果：

表 3: 不同参数下的性能对比

参数 (n, p)	原始版本 (us)	SIMD 版本 (us)	性能提升 (%)
$n = 4, p = 7340033$	0.00398	0.02502	$\approx -500\%$
$n = 131072, p = 7340033$	85.7989	56.0619	$\approx 35\%$
$n = 131072, p = 104857601$	89.2863	54.7128	$\approx 39\%$
$n = 131072, p = 469762049$	92.1617	54.8806	$\approx 40\%$

测试用例	实现方式	CPU 周期	指令数	IPC	分支预测失败率	缓存缺失率
0	串行 NTT	1,116,320,285	1,830,062,373	1.63	%	0.68%
0	Montgomery NTT	850,939,132	2,203,234,236	2.58	%	0.66%
1	串行 NTT	1,114,959,411	1,830,062,370	1.64	%	0.67%
1	Montgomery NTT	851,821,608	2,203,234,288	2.58	%	0.64%
2	串行 NTT	1,113,726,531	1,830,062,364	1.64	%	0.67%
2	Montgomery NTT	870,690,533	2,203,234,272	2.53	%	0.63%
3	串行 NTT	1,114,038,651	1,830,062,338	1.64	%	0.66%
3	Montgomery NTT	853,243,571	2,203,234,132	2.58	%	0.62%

表 4: NTT 性能测试结果对比

测试用例	周期数减少	指令数增加	IPC 提升
0	23.77%	20.39%	58.28%
1	23.60%	20.39%	57.31%
2	21.82%	20.39%	54.26%
3	23.40%	20.39%	57.31%

表 5: Montgomery NTT 相对于串行 NTT 的性能提升百分比

2. 浮点数取模

根据提示，还可以实现浮点数取模. 由于浮点数精度的误差导致结果不准确，很容易造成误差累积。

$$a\times b\%p = a\times b - p\times\ (a\times b)/p\ = a\times b - p\times\ (1/p)\times a\times b$$

具体实现的如下:

浮点数近似优化

```
    for (int j = 0; j < m; j += 4) {
            if (j + 4 <= m) {
                int32x4_t vw = {w,
                              (int)(1LL * w * wn % p),
                              (int)(1LL * w * wn % p * wn % p),
                              (int)(1LL * w * wn % p * wn % p * wn % p)
                                  };
                int32x4_t vu = vld1q_s32(a + i + j);
                int32x4_t vv = vld1q_s32(a + i + j + m);
                int32x4_t vvw;
                float32x4_t vf_p = vdupq_n_f32(p);
                float32x4_t vf_inv_p = vdupq_n_f32(inv_p);
                for (int k = 0; k < 4; ++k) {
                    int v_val = vgetq_lane_s32(vv, k);
                    int w_val = vgetq_lane_s32(vw, k);
                    long long prod = 1LL * v_val * w_val;
                    float f_prod = prod;
                    float f_q = f_prod * inv_p;
                    int q = (int)f_q;
                    int mod = prod - q * p;
                    if (mod >= p) mod -= p;
                    if (mod < 0) mod += p;
                    vvw = vsetq_lane_s32(mod, vvw, k);}
                int32x4_t new_u, new_v;
                int32x4_t vp = vdupq_n_s32(p);
                for (int k = 0; k < 4; ++k) {
                    int u_val = vgetq_lane_s32(vu, k);
                    int vw_val = vgetq_lane_s32(vvw, k);
                    int sum = u_val + vw_val;
                    if (sum >= p) sum -= p;
                    new_u = vsetq_lane_s32(sum, new_u, k);
```

```
                int diff = u_val - vw_val;
                if (diff < 0) diff += p;
                new_v = vsetq_lane_s32(diff, new_v, k);}
            vst1q_s32(a + i + j, new_u);
            vst1q_s32(a + i + j + m, new_v);
            w = 1LL * w * wn % p * wn % p * wn % p * wn % p;}
```

表 6: 不同参数下的性能对比

参数 (n, p)	原始版本 (us)	SIMD 版本 (us)	性能提升 (%)
$n = 4, p = 7340033$	0.00398	0.02502	$\approx -528\%$
$n = 131072, p = 7340033$	85.7989	60.2603	$\approx 29\%$
$n = 131072, p = 104857601$	89.2863	87.928	$\approx 2\%$
$n = 131072, p = 469762049$	92.1617	90.0183	$\approx 2\%$

测试用例	实现方式	CPU 周期	指令数	IPC	分支预测失败率	缓存缺失率
0	串行 NTT	1,135,066,507	1,830,062,526	1.61	%	0.68%
0	浮点数优化 NTT	926,931,961	1,641,690,705	1.77	%	0.95%
1	串行 NTT	1,130,784,096	1,830,062,409	1.61	%	0.67%
1	浮点数优化 NTT	912,781,729	1,641,690,633	1.79	%	0.97%
2	串行 NTT	1,123,548,661	1,830,062,364	1.62	%	0.67%
2	浮点数优化 NTT	910,043,384	1,641,690,633	1.80	%	0.94%
3	串行 NTT	1,115,423,876	1,830,062,468	1.64	%	0.67%
3	浮点数优化 NTT	923,894,090	1,641,690,644	1.77	%	0.96%

表 7: NTT 性能测试结果对比

测试用例	**周期数减少**	**指令数增加**	**IPC 提升**
0	18.33%	-10.29%	9.93%
1	19.27%	-10.29%	11.18%
2	19.00%	-10.29%	11.11%
3	17.17%	-10.29%	7.92%

表 8: 测试结果表格

算法的重点和上一个相似，$\frac{1}{p}$, 可以提前预处理出来, 所有操作均可由 SIMD 提供的函数实现

3. neon 向量化取模

这里探索 neon 几个向量化取模的实现，具体应用会用在 DIF-DIT 的 neon 并行实现中.

mod_add_vec: 向量模加

mod_add_vec: 向量模加

```
inline int32x4_t mod_add_vec(int32x4_t a, int32x4_t b, int32x4_t P) {
```

```
    int32x4_t s  = vaddq_s32(a, b);
    int32x4_t ge = vreinterpretq_s32_u32(vcgeq_s32(s, P));
    return vsubq_s32(s, vandq_s32(ge, P));}
```

首先使用 vaddq_s32(a,b), 对向量 a 和 b 中的四个 32 位有符号整数元素分别相加得到 s, 使用 vcgeq_s32(s,P) 对 s 和 P 做逐元素的 "signed greater-or-equal" 比较，返回一个无符号掩码向量 uint32x4_t。再使用 vreinterpretq_s32_u32(...) 把上一步得到的 uint32x4_t 掩码，按位 "原样" reinterpret 为 int32x4_t(后续操作要求), 再使用 vsubq_s32(s, vandq_s32(ge, P)) 对向量 s 做逐元素减法。达到了大于 P 的位置的值减去 P, 而不大于 P 的位置的值减去 0(不改变) 的效果。

mod_sub_vec: 向量模减

mod_sub_vec: 向量模减

```
inline int32x4_t mod_sub_vec(int32x4_t a, int32x4_t b, int32x4_t P) {
    int32x4_t d  = vsubq_s32(a, b);
    int32x4_t lt = vreinterpretq_s32_u32(vcltq_s32(d, vdupq_n_s32(0)));
    return vaddq_s32(d, vandq_s32(lt, P));}
```

原理基本同 add

路向量乘标量后取模

路向量乘标量后取模

```
#define VEC_WIDTH 4
static inline int mul_mod(int a, int b, int P) {
    return (int)((1LL * a * b) % P);}
inline int32x4_t mul_mod_vec(int32x4_t v, int w, int P) {
    int32x4_t res;
    for (int lane = 0; lane < VEC_WIDTH; ++lane) {
        int val = vgetq_lane_s32(v, lane);
        val     = mul_mod(val, w, P);
        res     = vsetq_lane_s32(val, res, lane);}
    return res;}
```

首先声明一个 NEON 向量类型 res，用来存放四个计算后的结果, 循环遍历向量的每个 "通道" 索引, 从向量 v 中提取第 lane 个 32 位有符号整数元素，赋值给 val。调用之前定义的标量模乘函数, 后将标量结果 val 写回到向量 res 的第 lane 个位置。

Barrett Reduction 的 NEON 向量化模乘函数 mod_mul_vec

为在 ARM NEON 向量化环境下高效地实现模乘运算，我们采用 Barrett 模约简方法，并行处理四个 32 位整数通道。算法定义如下：代码可见 githubNKU-Parallel-Computing仓库。

(三) 第三节 DIF+DIT 优化蝴蝶算法

DIF（Decimation in Frequency）算法是快速傅里叶变换（FFT）的一种实现方式，它通过按频域的方式将计算分解成多个较小的部分。DIF 与 DIT（按时域抽取）算法类似，但在分解时采用了不同的策略。

Algorithm 1 `mod_mul_vec`：四路并行 Barrett 模乘

Input: 向量寄存器 $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_{32}^4$，模数寄存器 $\mathbf{P} \in \mathbb{Z}_{32}^4$，预计算常数 $\boldsymbol{\mu} \in \mathbb{Z}_{32}^4$

Output: 输出 $\mathbf{r} = (\mathbf{a} \cdot \mathbf{b}) \bmod \mathbf{P}$

// 1. 并行计算 64 位乘积

$prod\_lo \leftarrow vmull\_s32(vget\_low\_s32(\mathbf{a}),\ vget\_low\_s32(\mathbf{b}))$

$prod\_hi \leftarrow vmull\_s32(vget\_high\_s32(\mathbf{a}),\ vget\_high\_s32(\mathbf{b}))$

// 2. 提取乘积高 32 位并近似商

$\mathbf{q\_lo} \leftarrow vshrn\_n\_s64(prod\_lo, 32);\quad \mathbf{q\_hi} \leftarrow vshrn\_n\_s64(prod\_hi, 32)$

$\mathbf{q} \leftarrow vcombine\_u32(\mathbf{q\_lo},\ \mathbf{q\_hi})$

$t\_lo64 \leftarrow vmull\_u32(vget\_low\_u32(\mathbf{q}),\ vget\_low\_u32(\boldsymbol{\mu}))$

$t\_hi64 \leftarrow vmull\_u32(vget\_high\_u32(\mathbf{q}),\ vget\_high\_u32(\boldsymbol{\mu}))$

$\mathbf{t\_lo} \leftarrow vshrn\_n\_u64(t\_lo64, 32);\quad \mathbf{t\_hi} \leftarrow vshrn\_n\_u64(t\_hi64, 32)$

$\mathbf{t} \leftarrow vreinterpretq\_s32\_u32(vcombine\_u32(\mathbf{t\_lo},\ \mathbf{t\_hi}))$

// 3. 计算候选余数

$tP\_lo \leftarrow vmull\_s32(vget\_low\_s32(\mathbf{t}),\ vget\_low\_s32(\mathbf{P}))$

$tP\_hi \leftarrow vmull\_s32(vget\_high\_s32(\mathbf{t}),\ vget\_high\_s32(\mathbf{P}))$

$r\_lo \leftarrow vsubq\_s64(prod\_lo,\ tP\_lo);\quad r\_hi \leftarrow vsubq\_s64(prod\_hi,\ tP\_hi)$

$\mathbf{r2\_lo} \leftarrow vmovn\_s64(r\_lo);\quad \mathbf{r2\_hi} \leftarrow vmovn\_s64(r\_hi)$

$\mathbf{r2} \leftarrow vcombine\_s32(\mathbf{r2\_lo},\ \mathbf{r2\_hi})$

// 4. 条件减法保证结果模长

$\mathbf{mask} \leftarrow vcgeq\_s32(\mathbf{r2},\ \mathbf{P})$

return $vsubq\_s32(\mathbf{r2},\ vandq\_s32(\mathbf{mask},\ \mathbf{P}))$

1. DIT

$$\begin{aligned} DFT(a,n)_k &= \sum_{i=0}^{n-1} a_i w_n^{ik} \\ &= (\sum_{i=0}^{\frac{n}{2}-1} a_{2i} w_n^{2ik}) + (\sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} w_n^{2ik+k}) \\ &= (\sum_{i=0}^{\frac{n}{2}-1} a_{2i} w_{\frac{n}{2}}^{ik}) + (\sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} w_{\frac{n}{2}}^{ik}) w_n^k \end{aligned} \tag{7}$$

DIT 需要一开始对 a 做蝴蝶变换。

2. DIT

同理我们可以考虑按频域抽取，考虑 k 的奇偶性。

$$\begin{aligned} DFT(a,n)_k &= \sum_{i=0}^{n-1} a_i w_n^{ik} \\ &= \sum_{i=0}^{\frac{n}{2}-1} a_i w_n^{ik} + a_{i+\frac{n}{2}} w_n^{(i+\frac{n}{2})k} \\ &= \sum_{i=0}^{\frac{n}{2}-1} (a_i + a_{i+\frac{n}{2}} (-1)^k) w_n^{ik} \end{aligned} \tag{8}$$

如果输入序列 a，我们可以得到蝴蝶变化后的 DFT。

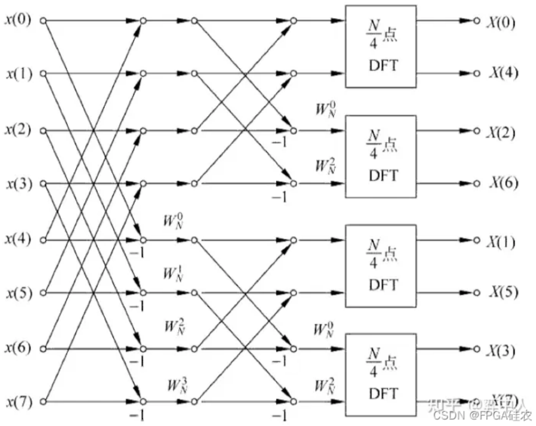

图 4: DIF 示意

3. 串行实现 DIF-DIT

首先，根据提供的资料实现一个串行的版本:

串行 DIF-DIT

```
    inline ll fpow(ll a, ll b, int P) {
    ll res = 1; a %= P;
    for (; b; b >>= 1) {
        if (b & 1) (res *= a) %= P;
        (a *= a) %= P;}
    return res;}
void calc_powg(int w[], int G, int P, int gen) {
    w[0] = 1; ll f;
    const int g = fpow(gen, (P-1)/G, P);
    for (int t = 0; (1<<(t+1)) < G; ++t) {
        f = w[1<<t] = fpow(g, G>>(t+2), P);
        for (int x = 1<<t; x < 1<<(t+1); ++x)
            w[x] = (ll)f * w[x - (1<<t)] % P;}}
void DIF(int f[], int l, int P, int w[]) {
    int lim = 1 << l;
    ll g, h;
    for (int len = lim; len > 1; len >>= 1) {
        for (int st = 0, t = 0; st < lim; st += len, ++t) {
            for (int i = st; i < st + len/2; ++i) {
                g = f[i];
                h = (ll)f[i + len/2] * w[t] % P;
                f[i] = (g + h) % P;
                f[i + len/2] = (P + g - h) % P;}}}}
void DIT(int f[], int l, int P, int w[]) {
    int lim = 1 << l;
    ll g, h;
    for (int len = 2; len <= lim; len <<= 1) {
        for (int st = 0, t = 0; st < lim; st += len, ++t) {
```

```
            for (int i = st; i < st + len/2; ++i) {
                g = f[i];
                h = f[i + len/2];
                f[i] = (g + h) % P;
                f[i + len/2] = (P + g - h) * w[t] % P;}}}
    const ll invl = fpow(lim, P-2, P);
    for (int i = 0; i < lim; ++i)
        f[i] = invl * f[i] % P;
    std::reverse(f + 1, f + lim);}
```

表 9: 不同参数下的性能对比

参数 (n, p)	起始版本 (us)	DIF-DIT 串行版本 (us)	性能提升 (%)
$n = 4, p = 7340033$	0.00398	0.00358	$\approx -500\%$
$n = 131072, p = 7340033$	85.7989	55.4273	$\approx 35\%$
$n = 131072, p = 104857601$	89.2863	57.6022	$\approx 39\%$
$n = 131072, p = 469762049$	92.1617	59.2388	$\approx 40\%$

4. 使用 omp simd 并行实现

相对于串行算法，omp simd 只需要在 DIF 的第三层循环前添加 #pragma omp simd, 于此同时，可以利用 omp 优化逐点乘法。

逐点乘法

```
void pointwise_mul(int A[], int B[], int lim, int P) {
    #pragma omp simd
    for (int i = 0; i < lim; ++i) {
        A[i] = (int)((ll)A[i] * B[i] % P);
    }
}
```

由于 omp 的优化会涉及到比较底层，所以我们可以阅读其汇编代码。

对这个版本的代码进行测试：感觉优化并不显著

表 10: 不同参数下的性能对比

参数 (n, p)	DIF-DIT 串行版本 (us)	DIF-DIT omp 优化版本 (us)	性能提升 (%)
$n = 4, p = 7340033$	0.00358	0.00503	$\approx -40\%$
$n = 131072, p = 7340033$	55.4273	54.9665	$\approx 0.8\%$
$n = 131072, p = 104857601$	57.6022	57.0639	$\approx 0.9\%$
$n = 131072, p = 469762049$	59.2388	58.64	$\approx 1\%$

5. 使用 neon 并行优化

在第二节中我们探讨了 neon 向量取模的实现，根据之前定义的函数可以得到 neon 优化的 DIF-DIT 算法。

$\text{DIF}_{n}eon$

```
void DIF_neon(int *f, int l, int P, const int *w) {
    const int lim = 1 << l;
    const int32x4_t Pvec = vdupq_n_s32(P);
    for (int len = lim; len > 1; len >>= 1) {
        const int half = len >> 1;
        for (int st = 0, t = 0; st < lim; st += len, ++t) {
            const int wt = w[t]; // 同一块共用同一 w
            int i = st;
            // 4-way SIMD
            for (; i + VEC_WIDTH - 1 < st + half; i += VEC_WIDTH) {
                int32x4_t g = vld1q_s32(f + i);
                int32x4_t h = vld1q_s32(f + i + half);
                h             = mul_mod_vec(h, wt, P); // 标量乘模 + 向量装载
                int32x4_t sum = mod_add_vec(g, h, Pvec);
                int32x4_t dif = mod_sub_vec(g, h, Pvec);
                vst1q_s32(f + i, sum);
                vst1q_s32(f + i + half, dif);}
             for (; i < st + half; ++i) {
                ll g = f[i];
                ll h = 1LL * f[i + half] * wt % P;
                f[i]          = (g + h) % P;
                f[i + half]   = (P + g - h) % P;}}}}
void DIT_neon(int *f, int l, int P, const int *w) {
    const int lim = 1 << l;
    const int32x4_t Pvec = vdupq_n_s32(P);
    for (int len = 2; len <= lim; len <<= 1) {
        const int half = len >> 1;
        for (int st = 0, t = 0; st < lim; st += len, ++t) {
            const int wt = w[t];
            int i = st;
            for (; i + VEC_WIDTH - 1 < st + half; i += VEC_WIDTH) {
                int32x4_t g = vld1q_s32(f + i);
                int32x4_t h = vld1q_s32(f + i + half);
                int32x4_t sum = mod_add_vec(g, h, Pvec);
                int32x4_t dif = mod_sub_vec(g, h, Pvec);
                dif             = mul_mod_vec(dif, wt, P);
                vst1q_s32(f + i, sum);
                vst1q_s32(f + i + half, dif);}
            // scalar tail
            for (; i < st + half; ++i) {
                ll g = f[i];
                ll h = f[i + half];
                f[i]          = (g + h) % P;
                f[i + half]   = (P + g - h) * 1LL * wt % P;}}}
    const ll invl = fpow(lim, P - 2, P);
    int32x4_t inv_vec = vdupq_n_s32((int)invl);
    int i = 0;
```

```
    for (; i + VEC_WIDTH - 1 < lim; i += VEC_WIDTH) {
        int32x4_t v = vld1q_s32(f + i);
        v          = mul_mod_vec(v, (int)invl, P);
        vst1q_s32(f + i, v);}
    for (; i < lim; ++i) f[i] = (ll)f[i] * invl % P;
    std::reverse(f + 1, f + lim);}
```

测试他的性能:

表 11: 不同参数下的性能对比

参数 (n,p)	DIF-DIT 串行版本 (us)	neon SIMD 优化 (us)	性能提升 (%)
$n=4, p=7340033$	0.00358	0.0159	$\approx -344\%$
$n=131072, p=7340033$	55.4273	33.2878	$\approx 40\%$
$n=131072, p=104857601$	57.6022	35.4968	$\approx 38\%$
$n=131072, p=469762049$	59.2388	36.7921	$\approx 37\%$

二、 总结

本文完成了 NTT 算法的全面优化研究，具体工作包括以下几个方面:

1. 深入分析了 FFT 与 NTT 算法的数学原理，明确了算法实现细节和计算瓶颈;
2. 引入 Montgomery 规约方法，解决了模乘运算的效率问题，通过 ARM NEON 向量化实现了高效的模乘与模加减运算;
3. 提出了基于浮点数近似的取模优化方案，虽然性能提升不及 Montgomery 方法，但提供了另一种取模思路;
4. 实现了 DIF 和 DIT 两种 FFT 结构的混合优化，通过串行实现、OpenMP SIMD 并行优化以及 NEON SIMD 并行优化等多种手段，显著提升了计算性能;
5. 性能测试验证了 SIMD 优化的有效性，优化后的算法在较大规模的输入数据条件下相比串行算法表现出显著优势。

未来工作可以继续探索其他的并行优化技术，例如 GPU 加速，并针对实际应用场景进一步优化算法的实现，以满足更高性能需求。

参考文献

[1] J.E. Jia. Montgomery multiplication and modular operations. https://jia.je/crypto/2023/07/23/montgomery-mul-mod/#%E7%94%A8%E9%80%94. Accessed: 2025-04-18.