计算机学院

并行程序设计第三次报告

SIMD 编程——以 NTT 选题为例

姓名：宋卓伦

学号：2311095

专业：计算机科学与技术

2025 年 4 月 29 日，南开大学计算机学院，天津

目录

1 实验目的及实验介绍

1.1 实验目的

经过这段时间 SIMD 架构的学习，我对于这方面变成有了一定的体会。这次我们以快速傅里叶变换为选题，展开 SIMD 编程，探究该方法对于 NTT 数论变换的优化程度。

1.2 SIMD

SIMD(Single Instruction Multiple Data) 即单指令流多数据流，是一种采用一个控制器来控制多个处理器，同时对一组数据（又称“数据向量”）中的每一个分别执行相同的操作从而实现空间上的并行性的技术。简单来说就是一个指令能够同时处理多个数据，这充分体现了并行化操作的特点。

1.3 实验硬件和环境

本次实验选用远程 WSL 连接 Ubuntu24.04 进行本地编程（表1，但是本人远程不支持 neon 头文件导入，所以仅用作正确性和可行性验证），同时采用助教学长们搭建的 OpenEuler 服务器进行代码设计和主要的性能查看。

属性	相关内容
内核版本	5.15.167.4-microsoft-standard-WSL2
硬件架构	x86_64（64 位处理器） 但是安装了模拟的 qemu-aarch64 架构 能够实现 user-static 模式

表 1: 硬件与环境信息

2 问题描述——NTT 算法简介

2.1 离散傅里叶变换

此处我们先引入离散傅里叶变换。工业界中，尤其是与时序数据相关的，具有周期性的数据，需要进行卷积（convolution）计算。

离散傅里叶变换（DFT）将有限长度的离散信号转换到频域的一种数学变换。对于一个长度为 N 的离散序列 $x[n]$，$n = 0, 1, , N-1$，其离散傅里叶变换 $X[k]$ 定义为：

$$\mathbf{X}[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j(2\pi/N)kn}, \ \ (0 \leq k \leq N-1) \tag{1}$$

DFT 将时域信号 $x[n]$ 转换为频域信号 $X[k]$，其中 $e^{-j(2\pi/N)kn}$ 是单位复根，表示信号在不同频率上的投影。

在这里我们可以发现，多项式的系数被写成了复数的形式，算法复杂度是 $O(n^2)$ 的形式。

2.2 快速傅里叶变换

由上所述，这样的复杂度是无法接受的，所以我们引入快速傅里叶变换（fast fourier transform, FFT），通过多次迭代递归，划分为递归子问题实现算法优化，时间复杂度来到了 $O(n\log n)$，实现了优化。设 $x = \left[x_0, x_1, \cdots, x_{n-1}\right]^{\mathrm{T}}$ 是时域输入向量，设单位根 $\omega_n = e^{-2\pi i/n}$，则 DFT 可以表示为矩阵乘法：

$$X = F_n \cdot x \tag{2}$$

其中 DFT 矩阵 F_n 定义为：

$$F_n = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \tag{3}$$

因此，输出频域向量为：

$$X_k = \sum_{j=0}^{n-1} x_j \cdot \omega_n^{jk}, \quad k = 0, 1, \ldots, n-1 \tag{4}$$

其中 $\omega_n = e^{-2\pi i/n}$ 是 n 阶单位根。

2.3 数论变换

但是新的问题：复数需要引入虚数 i，在计算过程中涉及到了许多高精度浮点数计算（每个复数系数的实部和虚部是一个正弦及余弦函数，π 的原因导致），在多次重复的情况下误差累加会导致计算错误。最浅显的一点，对于复数需要额外的存储结构（C++ 中需要手搓结构体或者调用 complex 库），浪费存储空间和运行时间。这里我们从微观的浮点数视角跳出，看宏观的大数视角。

我们想到引入离散数学中关于同构和模的概念，利用原根和欧拉函数的性质，在一个周期内相反（余数相加等于模数）的两个数可以在单位圆上标记的特点，我们将圆 n 等分，利用同构实现和虚数根 ω^k 相同的效果。自然我们引入了 NTT 的概念。**数论变换**（number-theoretic transform, NTT）是离散傅里叶变换（DFT）在数论基础上的实现；快速数论变换（fast number-theoretic transform, FNTT）是快速傅里叶变换（FFT）在数论基础上的实现。

$$\mathbf{X}[k] = \sum_{n=0}^{N-1} x[n] \cdot \omega^{nk} \mod p, \ \ (k = 0, 1, \ldots, N-1) \tag{5}$$

其中，p 是一个素数，ω 是模 p 意义下的 N 次原根，满足 $\omega^N \equiv 1 \mod p$ 且 $\omega^{N/d} \not\equiv 1 \mod p$（对于 $d < N$ 的所有因子）。

数论变换是一种计算卷积的快速算法。主要应用在于计算多项式乘法，给定两个多项式 $A(x) = \sum_i a_i x^i$ 和 $B(x) = \sum_i b_i x^i$，其乘法结果 $C(x) = A(x) \cdot B(x)$ 的系数可以通过卷积计算：

$$c_k = \sum_{i+j=k} a_i b_j \tag{6}$$

使用 NTT：

1. 将 A 和 B 的系数向量进行 NTT，得到频域表示 A' 和 B'。
2. 计算点值乘法：$C'[k] = A'[k] \cdot B'[k] \bmod p$。
3. 对 C' 进行逆 NTT，得到卷积结果 c_k。

模数 p 限制了系数范围，需确保结果不超过 p。若结果超出，可使用多模数 NTT 结合中国剩余定理恢复。NTT 解决的是多项式乘法带模数的情况，可以说有些受模数的限制，数也比较大。目前最常见的模数是 998244353。

3 实验设计

3.1 朴素算法实现

朴素算法——逐个计算

```
void poly_multiply(int *a, int *b, int *ab, int n, int p){
    for(int i = 0; i < n; ++i){
        for(int j = 0; j < n; ++j){
            ab[i+j]=(1LL * a[i] * b[j] % p + ab[i+j]) % p;
        }
    }
}
```

这里朴素算法的实现比较简单：朴素算法描述了

$$f(x) = \sum_{i=0}^{n} a_i x^i \tag{7}$$

$$g(x) = \sum_{i=0}^{n} b_i x^i \tag{8}$$

两个式子进行卷积运算得到 $f \cdot g(x)$，将对应次幂的系数整合出来，$ab[i+j]$ 在循环过程中不断更新。最后得到整个结果系数向量。这样的算法时间复杂度 $O(n^2)$，是无法接受的。所以下面介绍 NTT 的代码实现。

3.2 NTT 初始化实现

我们利用前面提到的数论前置知识设计算法：

Algorithm 1 多项式乘法的 NTT 实现

Input: 多项式系数数组 a,b, 结果多项式系数数组 ab, 多项式次数 n, 模数 p

Output: 把多项式 a 和 b 相乘结果存储在 ab 中

function POLY_MULTIPLY(a, b, ab, n, p)
  $m \leftarrow 1$
  while $m < 2 * n - 1$ **do**
    $m \leftarrow m \times 2$ ▷ 将 m 变大至两数组之和-1
  end while
  $ta, tb \leftarrow length = m, initial = 0$ ▷ 分配长度为 m 的整数数组并初始化为 0
  $ta[i] \leftarrow a[i], \quad tb[i] \leftarrow b[i], \quad \forall\, i \in [0, n)$ ▷ 前 n 个赋初值
  $root \leftarrow 3$ ▷ 原根，依赖具体模数
  $inv\_root \leftarrow \text{mod\_pow}(root,\, p-2,\, p)$ ▷ 求取模逆
  $\text{ntt}(ta,\, m,\, root,\, p)$，$\text{ntt}(tb,\, m,\, root,\, p)$ ▷ 使用 NTT 获取单位根
  for i 0 to $m - 1$ **do**
    $ta[i] \leftarrow (ta[i] \times tb[i]) \bmod p$ ▷ 点积计算
  end for
  $\text{ntt}(ta,\, m,\, inv\_root,\, p)$ ▷ 通过标记进行 INTT，还原结果
  $inv\_m \leftarrow \text{mod\_pow}(m,\, p-2,\, p)$
  for i 0 to $2 * n - 2$ **do**
    $ab[i] \leftarrow (ta[i] \times inv\_m) \bmod p$
  end for
  delete ta，tb 数组的内存
  return
end function

下面是复杂度分析：设输入多项式的次数为 n，变换长度 $m = O(n)$（通常为最接近 $2n - 1$ 的 2 的幂）。

1. 初始化长度：通过循环将 m 调整为 2 的幂，时间复杂度为 $O(\log n)$。
2. 数组分配与赋值：分配和初始化数组 ta 和 tb 需要 $O(m) = O(n)$ 时间，复制 a 和 b 的系数需要 $O(n)$ 时间。
3. NTT 变换：每次 NTT 或 INTT 的时间复杂度为 $O(m \log m) = O(n \log n)$。算法中执行了两次 NTT（对 ta 和 tb）和一次 INTT，总时间复杂度为 $O(n \log n)$。
4. 点值乘法：对 m 个元素逐一相乘，时间复杂度为 $O(m) = O(n)$。
5. 模逆计算：使用快速模幂算法计算 $root$ 和 m 的模逆，每次时间复杂度为 $O(\log p)$，共两次，复杂度为 $O(\log p)$。
6. 归一化：对 $2n - 1$ 个元素进行乘法和取模，时间复杂度为 $O(n)$。
7. 内存管理：释放数组内存的时间复杂度为 $O(1)$。

综合以上步骤，总时间复杂度为：

$$O(n \log n) + O(n) + O(\log p) = O(n \log n),$$

其中 $O(n\log n)$ 是主导项，假设模数 p 的位数较小，$O(\log p)$ 可忽略。基于 NTT 的多项式乘法算法通过将系数表示转换为点值表示，显著降低了多项式乘法的时间复杂度，从朴素算法的 $O(n^2)$ 优化到 $O(n\log n)$。该算法在模运算环境下表现出色，广泛应用于高性能计算领域，如大整数乘法和卷积运算等。

由于篇幅问题我们将代码放在后面的 GitHub 网站上面：我的 GitHub。但是，虽然实现了 NTT，是否还能够使用前面提到的 SIMD 技术优化程序呢？下面我们给出尝试。

4 实验并行化改进

这里我们提出几种方法来基于 SIMD 方法实现并行化操作，实现内存友好的计算：

4.1 向量化模乘

Montgomery 规约是专门用于取模优化的算法，利用进制表示简化除法运算，转化成位运算。

首先进行初始化操作：

- 模数逆元计算（模 $r = 2^k$）：

$$ir = -m^{-1} \mod r$$

  使用牛顿迭代法：

$$x_{i+1} = x_i \cdot (2 - x_i \cdot m) \mod r, \quad x_0 = 1$$

- Montgomery 转换因子：

$$r2 = r^2 \mod m = (2^k \mod m) \cdot (2^k \mod m) \mod m$$

接下来转换为 Montgomery 形式：将整数 a 转换为 Montgomery 形式 $\bar{a}$：

$$\bar{a} = (a \cdot r2) \mod m$$

Montgomery 模乘（REDC 算法）：计算 Montgomery 形式下的模乘 $\bar{a} \cdot \bar{b} \mod m$：

$$\text{REDC}(T) = \frac{T + (T \cdot ir \mod r) \cdot m}{r}$$

其中 $T = \bar{a} \cdot \bar{b}$。具体步骤：

- 计算辅助值：

$$u = (T \cdot ir) \mod r$$

- 计算结果：

$$R = \frac{T + u \cdot m}{r}$$

  若 $R \geq m$，则 $R = R - m$。

接下来，将 Montgomery 形式 $\bar{a} = a \cdot r \mod m$ 转换回普通形式：

$$a = \text{REDC}(\bar{a} \cdot 1) = \frac{\bar{a} + (\bar{a} \cdot ir \mod r) \cdot m}{r}$$

下面是一段简单的伪代码：

Algorithm 2 简化的 Montgomery 算法实现

Input: 模数 m，整数 a，b ▷ 输入模数 m 和待操作的整数 a、b
Output: 返回 Montgomery 形式下的乘积 ▷ 输出为 Montgomery 形式的模乘结果

function MONTGOMERY_INV(m) ▷ 计算模 2^{32} 下 $-m^{-1}$ 的逆元
  $x \leftarrow 1$ ▷ 初始化逆元为 1
  for $i \leftarrow 0$ to 5 **do** ▷ 迭代 6 次以逼近逆元
    $x \leftarrow x \times (2 - x \times m)$ ▷ 牛顿迭代法更新 x，计算 $-m^{-1} \mod 2^{32}$
  end for
  return x ▷ 返回计算得到的逆元
end function
function MONTGOMERY_CONSTRUCTOR(m) ▷ 初始化 Montgomery 算法参数
  $ir \leftarrow -\text{Montgomery\_inv}(m)$ ▷ 调用 Montgomery_inv 计算 $-m^{-1} \mod 2^{32}$
  $r2 \leftarrow (2^{32} \mod m)^2 \mod m$ ▷ 计算 $r^2 \mod m$，其中 $r = 2^{32}$
  return $(m, ir, r2)$ ▷ 返回模数 m、逆元 ir 和 $r^2 \mod m$
end function
function MONTGOMERY_TO($a, m, r2$) ▷ 将整数 a 转换为 Montgomery 形式
  return $(a \times r2) \mod m$ ▷ 计算 $a \cdot r^2 \mod m$，转换为 Montgomery 形式
end function
function MONTGOMERY_MUL(a, b, m, ir) ▷ 在 Montgomery 形式下计算 $a \cdot b \mod m$
  $t \leftarrow a \times b$ ▷ 计算 a 和 b 的乘积
  $u \leftarrow (t \times ir) \mod 2^{32}$ ▷ 计算 $(t \cdot ir) \mod 2^{32}$，用于 REDC 算法
  $r \leftarrow (t + u \times m) \gg 32$ ▷ 执行 REDC 算法：$(t + u \cdot m)/2^{32}$
  if $r \geq m$ **then** ▷ 检查结果是否需要归约
    $r \leftarrow r - m$ ▷ 若 $r \geq m$，减去 m 确保结果在 $[0, m)$
  end if
  return r ▷ 返回 Montgomery 形式下的模乘结果
end function
function MONTGOMERY_VAL(a, m, ir) ▷ 将 Montgomery 形式的值还原为普通形式
  return $\text{Montgomery\_mul}(a, 1, m, ir)$ ▷ 通过与 1 相乘，计算 $a \cdot 1 \mod m$ 还原
end function

4.2 NTT 蝴蝶变换

蝴蝶变换（Butterfly Operation）是快速傅里叶变换（FFT）与数论变换（NTT）中的基本计算单元。它通过一对输入元素的加减与乘法，逐步将数据重组，达到高效计算多项式乘积或频域转换的目的。

在一次标准蝴蝶变换中，设输入为 u 和 v，旋转因子为 w，模数为 p，则更新规则如下：

$$\text{new}_u = (u + v \times w) \bmod p \tag{9}$$

$$\text{new}_v = (u - v \times w) \bmod p \tag{10}$$

其中乘积 $v \times w$ 体现了旋转因子的作用，不同层次的旋转因子用于捕捉不同频率成分。我此外蝴蝶变换具有以下特点：

- 计算量小，仅需加减乘除模；
- 数据访问模式规律，便于向量化（如 SIMD 优化）；
- 能以 $O(n \log n)$ 的时间复杂度完成整体变换，大幅优于直接计算。

在硬件优化中，蝴蝶变换往往被批量处理（如一次并行 4 或 8 对元素）以发挥指令集并行能力。

Algorithm 3 使用蝴蝶变换的 NTT

Input: 输入数组 a，长度 n，原根幂 w，模数 p
Output: 将数组 a 原位变换

function NTT_SIMD(a, n, w, p)
  位逆序置换 a
  for $len \leftarrow 2$ to n，每次翻倍 **do**
    $wlen \leftarrow w^{(p-1)/len} \bmod p$
    if $len/2 \geq 4$ **then**
      预计算旋转因子数组 $twiddles$
      for 每组长度为 len 的子区间 **do**
        for 每 4 个元素一组处理 **do**
          并行计算乘旋转因子、求和与差并模 p
        end for
        剩余不足 4 个元素时，退回标量计算
      end for
      释放 $twiddles$
    else
      for 每组长度为 len 的子区间 **do**
        逐个计算旋转因子乘积，更新 a
      end for
    end if
  end for
end function

4.3 其他优化操作

这里我们初步尝试使用 DIT 进行我们的处理。

Algorithm 4 简化版 NTT 变换（DIT 形式）

Input: 输入数据数组 $data$，数据长度 n，原根 g，模数 mod
Output: 将 $data$ 变换成 NTT 结果

function NTT_TRANSFORM$(data, n, g, mod)$
  位反转排序： 对 $data$ 数组进行位反转排序
  计算旋转因子表： 计算旋转因子 $wtable$
  for $len \leftarrow 2$ **to** n **by** 2 **do**

```
        for i ← 0 to n − 1, step len do
            for j ← 0 to len/2 − 1 do
                u ← data[i + j]
                v ← (data[i + j + len/2] × wtable[j]) mod mod
                data[i + j] ← (u + v) mod mod
                data[i + j + len/2] ← (u − v + mod) mod mod
            end for
        end for
    end for
    return data
end function
```

接下来我们继续分析算法复杂度相关：

- **位反转排序**：通过交换数组元素实现位反转排序，时间复杂度为 $O(n)$。
- **计算旋转因子表**：预计算旋转因子表 $wtable$，通常需要计算 $\frac{n}{2}$ 个单位根的幂次。每次计算涉及模幂运算，假设模幂运算复杂度为 $O(\log mod)$，总时间复杂度为 $O(n\log mod)$。在实际实现中，旋转因子表可以缓存，降低开销。
- **蝶形运算**：
  - 外层循环迭代 $\log n$ 次（len 从 2 到 n）。
  - 中层循环对每个 len 执行 $\frac{n}{len}$ 次迭代。
  - 内层循环对每个子问题执行 $\frac{len}{2}$ 次蝶形运算，每次蝶形运算涉及常数次加法、减法、乘法和取模操作，复杂度为 $O(1)$。
  - 总计蝶形运算的复杂度为：

  $$\sum_{k=1}^{\log n}\left(\frac{n}{2^k}\cdot 2^{k-1}\right)=\sum_{k=1}^{\log n}\frac{n}{2}=\frac{n\log n}{2}.$$

  考虑每次蝶形运算的常数操作，时间复杂度为 $O(n\log n)$。
- **其他操作**：返回数组的复杂度为 $O(1)$。

综合以上步骤，总时间复杂度为：

$$O(n)+O(n\log mod)+O(n\log n)=O(n\log n),$$

其中 $O(n\log n)$ 是主导项，假设模数 mod 的位数较小，$O(n\log mod)$ 可忽略。

5 程序性能分析

如上表2所示，进行优化之后的算法当规模增大的时候，其表现远远比朴素算法好（$n=4$ 的时候规模很小其表现可以忽略不计），这正是体现了 SIMD 方法的强大之处。

算法	n	p	平均延迟（us）	结果正确性
朴素算法（暴力）	4	7340033	0.00021	正确
	131072	7340033	95664.8	正确
	131072	104857601	101744	正确
	131072	469762049	106164	正确
NTT 算法（优化）	4	7340033	0.00426	正确
	131072	7340033	83.9439	正确
	131072	104857601	88.5459	正确
	131072	469762049	90.6736	正确
Montgomery+ 蝴蝶变换	4	7340033	0.00516	正确
	131072	7340033	46.8303	正确
	131072	104857601	47.7473	正确
	131072	469762049	48.6792	正确

表 2: 多项式乘法算法性能比较

6 Profiling

这次实验，我们采用 Perf 进行事件采样，获得相应数据进行分析。

Perf 的测试

```
perf stat -g -e cpu-clock,cycles,instructions,cache-references,cache-misses,
L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses ./main

perf report > 文件名.txt
```

我们使用如上的指令进行性能分析，stat 的方式可以直接查看相应的数据，使用获取的数据从上述维度探究程序性能的影响（这里不提供普通算法）。

6.1 数据的总览

指标	Montgomery+ 蝴蝶变换	NTT 优化（neon）	NTT 普通
总耗时 (s)	**0.478**	0.479	0.630
CPU 时钟周期数	**1,112,273,425**	1,120,511,456	1,527,151,171
指令数	2,344,587,477	**2,340,706,549**	2,516,431,123
IPC（每周期指令数）	**2.11**	2.09	1.65
L1 缓存引用数	**688,826,674**	688,855,943	754,585,808
L1 缓存 Miss 率 (%)	0.63	0.66	**0.59**
LLC 加载次数	5,766,238	5,595,926	**5,496,173**
LLC Miss 率 (%)	**0.86**	0.96	1.28
用户/系统时间比	90%/10%	86%/14%	97%/3%

表 3: 三个程序的性能比较

在表3 中，列出了三个不同程序的性能统计数据。

- 从**总耗时**上来看，Montgomery+ 蝴蝶变换和 NTT 优化的执行时间**非常接近**，均约为 0.48 秒，而 NTT 普通算法明显较慢，耗时达到 0.63 秒。

- 从**指令吞吐率**（IPC）来看，Montgomery+ 蝴蝶变换达到了 2.11，由于 IPC 指标直接反映了 CPU 的指令执行效率，因此 Montgomery+ 蝴蝶变换从数据来看在指令层面上最为高效。
- **缓存**方面，三个程序的 L1 缓存 miss 率均较低，保持在 0.6% 左右，说明局部性良好；不过 NTT 普通尽管 L1 miss 率最低，但 LLC（最后一级缓存）miss 率却最高（1.28%），这表明未优化算法在更大粒度数据访问时**存在一定瓶颈**。
- 从**用户态和系统态的时间占比**来看，Montgomery+ 蝴蝶变换和 NTT 优化的系统开销略高于 NTT 普通，但考虑到其整体运行时间较长，这种偏差**并不足以抵消**性能下降带来的影响。

综合比较可得，实现了 **Montgomery+ 蝴蝶变换**的算法在各项指标上均表现最佳，**指令吞吐率高、缓存命中率优良，且整体运行时间最短**，因而是目前这些版本中性能较优的实现版本。

6.2 更进一步的解释

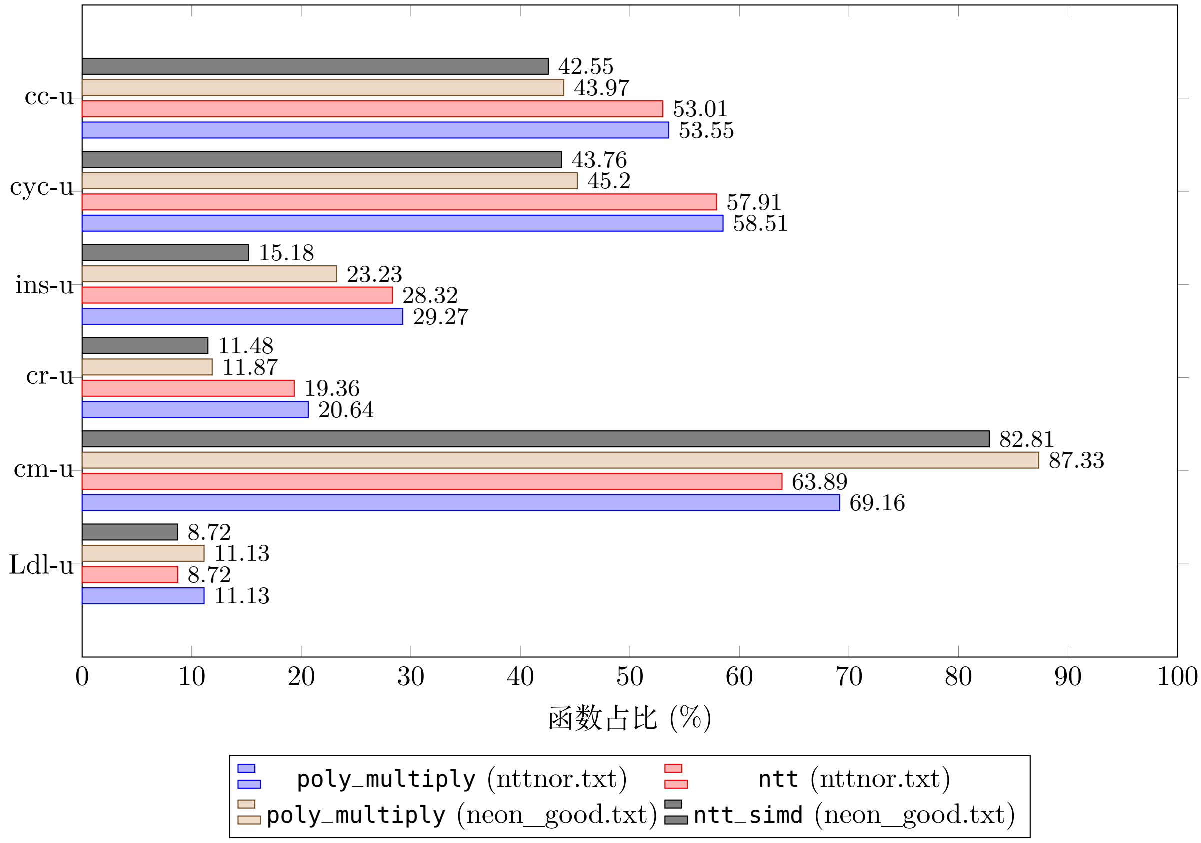

图 6.1：各函数在不同性能指标下的占比情况（横向绘制，带标签），其中六项指标从上到下依次是：cpu-clock-u，cycles-u，instructions-u，cache-references-u，cache-misses-u，L1-dcache-loads-u

由上图6.1我们可以发现，在第五行 cache-misses 里面，（结合上图我们已知优化后的命中率高于未优化）优化后的程序 cache-misses 更多是来自程序本身而不是其他项目，这大大提高了程序的性能。（占比只反映对应关系而不是绝对数量）。CPU 运行时间占比也大大降低了。

通过对比普通实现和 NEON SIMD 优化版本在不同性能指标下的表现，结果表明，NEON SIMD 优化显著提高了程序的执行效率。具体表现为：

1. **CPU 时钟消耗（cc-u）**：优化后的 `poly_multiply (neon_good.txt)` 和 `ntt_simd` 相比普通实现，CPU 时钟消耗显著降低，表明 SIMD 优化显著减少了 CPU 占用，提升了程序执行速度。

2. **CPU 周期数（cyc-u）**：SIMD 优化版本的周期数大幅减少，这进一步验证了 SIMD 指令级别的加速效果。
3. **指令数（ins-u）**： 优化后的 `poly_multiply (neon_good.txt)` 指令数仅为普通实现的一半左右（15.18%），显示了 SIMD 优化在降低指令数量方面的优势。
4. **缓存访问（cr-u）**：优化版本的缓存访问次数较普通版本显著减少，这表明 SIMD 优化有效减少了缓存访问负担，提高了内存访问的集中性与效率。
5. **缓存命中率（cm-u）**：在缓存命中率方面，SIMD 版本的缓存未命中率大幅降低，`poly_multiply (neon_good.txt)` 和 `ntt_simd` 的 miss 率为 63%-69%，远低于普通版本的 82%-87%。这表明 SIMD 优化显著提升了程序的缓存局部性，减少了不必要的缓存失效。
6. **L1 缓存加载次数（Ldl-u）**：在 L1 缓存加载次数方面，优化前后差异较小，均在 8.7%-11.1% 之间，说明优化主要体现在减少缓存未命中和降低访问开销，而非改变访问数量。

综上所述，NEON SIMD 优化极大地提升了程序性能，主要体现在减少 CPU 时钟消耗、降低指令数和周期数、提高缓存命中率等方面，特别是 `poly_multiply (neon_good.txt)` 版本的表现最为突出，展示了 SIMD 优化在实际应用中的显著优势。

7 实验总结

实验的代码在我的 GitHub 网站上：我的 GitHub

7.1 本实验的概括总结

在使用 SIMD（单指令多数据）进行算法优化时，对于各条指令的熟悉成为了我学习过程中的最大难点。经过一段时间的学习，我对该指令集有了一个初步的了解，进而一步一步实现自己的编程。

最初我发现使用 NTT 优化之后性能远远大于朴素算法，但是当我继续优化的时候碰到了一些瓶颈。我开始采取 Montgomery 算法继续探索优化路径，例如通过内联函数、选择合适的编译选项以及调整相应的变量类型来提升指令效率，或通过结构调整减少 cache miss，以实现更高的性能和功能优化。加速程序的并行化。

虽然最后还是没有实现较大模数的计算，但是后面我会继续发掘更多新的算法来实现这一特点。

在实验过程中，我意识到这让我深刻体会到，SIMD 优化的成功不仅依赖于指令级并行，还需要精准的实验设计和数据分析来捕捉性能提升的每一个细节。

7.2 实验以外的总结

这次实验对我来说是个极大的挑战。在事务繁忙的 4 月，我开始实验的时间相对较晚，在实验过程中也是遇到了一些困难，包括但不限于服务器使用不熟练，算法原理一知半解，甚至还有代码编写频频报错这样非常不应该的错误。但是我通过一系列的学习开始慢慢熟悉这套过程，能够熟悉原理和细节上的要点，最后能较为不错的完成实验。这对于我以后的学习很有帮助。

最后，由衷感谢助教学长们的辛勤付出，他们对于我们实验的配置和实施给予了很大帮助。希望未来我能够好好学习这门课程，尽可能多地学会很多知识，将我所学的知识传递下去帮助更多小同学们！再次衷心感谢我们的老师和助教们的辛勤付出！