南開大學
Nankai University

计算机学院

并行程序设计实验报告

基于 GPU 的 NTT 多项式乘法并行优化

姓名：廖望

学号：2210556

专业：计算机科学与技术

2025 年 6 月 28 日

摘　要

本项目旨在利用 NVIDIA GPU 的大规模并行计算能力，对基于快速数论变换（NTT）的多项式乘法进行性能优化。实验在 CUDA 平台上实现并优化了 NTT 算法，通过预计算并一次性拷贝旋转因子、采用模板化内核设计等策略，显著降低了数据传输和计算开销。我们重点对比了朴素模乘、Barrett 约减和 Montgomery 约减三种不同模乘算法在 GPU 上的性能表现。实验结果表明，与 CPU 串行实现相比，经过优化的 GPU 版本取得了高达 78 倍的性能加速，其中 Montgomery 约减方法表现最佳，充分展示了算法与硬件协同优化在解决计算密集型问题上的巨大潜力。代码仓库可访问：GitHub。

关键词： 并行计算; NTT; 性能优化; GPU; CUDA; Montgomery 约减

Abstract

This project aims to leverage the massively parallel computing power of NVIDIA GPUs to optimize the performance of polynomial multiplication based on the Number Theoretic Transform (NTT). The experiment implements and optimizes the NTT algorithm on the CUDA platform. By pre-computing and batch-copying twiddle factors and employing a templated kernel design, data transfer and computational overheads were significantly reduced. We focused on comparing the performance of three different modular multiplication algorithms on the GPU: naive, Barrett reduction, and Montgomery reduction. The results show that the optimized GPU version achieves a speedup of up to 78x compared to the serial CPU implementation, with the Montgomery reduction method performing the best. This demonstrates the immense potential of algorithm-hardware co-optimization in solving computationally intensive problems. Full code is available at: GitHub.

Keywords: Parallel Computing; NTT; Performance Optimization; GPU; CUDA; Montgomery Reduction

目 录

1 引言

多项式乘法是数字信号处理、现代密码学等领域的关键计算任务。传统的 $O(n^2)$ 算法在处理大规模多项式时效率低下。为此，学术界和工业界普遍采用基于快速傅里叶变换（FFT）的 $O(n \log n)$ 算法。然而，FFT 依赖于浮点运算，存在精度误差问题。

快速数论变换（NTT）通过在整数有限域上进行计算，完美地解决了 FFT 的精度问题，同时保持了 $O(n \log n)$ 的高效率，成为高精度、高性能多项式乘法，特别是在密码学应用中的理想选择。

本次实验的核心目标是利用 NVIDIA GPU 的大规模并行计算能力，对 NTT 算法进行加速。我们将实现一个 CUDA 版本的 NTT，并通过优化数据传输、内存访问以及核心的模乘运算（对比朴素、Barrett、Montgomery 三种方法），最终量化评估 GPU 并行版本相对于 CPU 串行版本的加速效果，并分析不同优化策略带来的性能增益。

2 实验原理

2.1 快速数论变换 (NTT) 原理简述

NTT 是快速傅里叶变换（FFT）在有限域上的变体，它利用有限域中的" 原根" 替代复数单位根，将多项式乘法中的卷积运算转化为点值乘法，从而将时间复杂度从 $O(n^2)$ 降低到 $O(n \log n)$。NTT 的核心运算是蝶形运算，其迭代形式如下：对于长度为 `len` 的当前变换阶段和旋转因子 `w`：

$$u = a[i]$$

$$v = (a[i + len/2] \cdot w) \pmod{mod}$$

$$a[i] = (u + v) \pmod{mod}$$

$$a[i + len/2] = (u - v) \pmod{mod}$$

在 NTT 的每个阶段中，所有蝶形运算都是数据独立的，这为大规模并行化提供了理论基础。本次实验将使用一个纯 C++ 串行实现的 NTT 算法作为性能基准，以衡量后续 GPU 优化的加速效果。

2.2 CUDA 并行模型与实验环境

CUDA 采用 `Grid-Block-Thread` 的三级并行模型。一个 CUDA 内核（Kernel）以一个 Grid 的形式启动，Grid 由多个 Block 组成，每个 Block 又由多个 Thread 组成。在本次实验中，我们将 NTT 的蝶形运算映射到 CUDA 线程上，以实现最大程度的并行。

实验环境如下：

- **操作系统**: Ubuntu 24.04 (on WSL2)
- **处理器 (CPU)**: Intel Core i5-12500H
- **图形处理器 (GPU)**: NVIDIA GeForce RTX 3050 Ti Laptop GPU
- **编程语言**: C++, CUDA C++
- **编译器**: g++, nvcc 11.8
- **CUDA Toolkit**: 11.8

2.3 关键优化原理：模乘约减

在 NTT 中，模乘和模加/减是最高频的运算。由于% 取模运算符在硬件层面通常对应一个开销较大的 `div` 指令，优化模乘运算是提升 NTT 性能的关键。本次实验对比了三种方法：

1. **朴素模乘 (Naive)**: 直接使用 `(a * b) % mod`。这在代码上最简单，但性能最差。
2. **Barrett 约减 (Barrett Reduction)**: 通过预计算一个常数 $\mu = \lfloor 2^k/m \rfloor$，将取模运算转换为两次乘法、一次移位和几次加减法。这避免了代价高昂的除法指令。其核心思想是 `a % m = a - m * floor(a / m)`，并用乘法和移位高效地计算 `floor(a/m)` 的近似值。
3. **Montgomery 约减 (Montgomery Reduction)**: 将所有操作数转换到一个特殊的"Montgomery 域" 中。在这个域内，模乘的计算效率非常高。其代价是在进入和离开 Montgomery 域时需要进行额外的转换。对于需要大量连续模乘的算法（如 NTT），这个初始开销可以被摊销，从而获得显著的性能提升。

3 GPU 设计与实现

3.1 总体流程

整个 GPU 加速的多项式乘法流程如下：

1. **数据准备 (Host)**: 将两个输入多项式 `poly1` 和 `poly2` 补零，使其长度 `n` 扩展到大于等于 $deg(\texttt{poly1})+deg(\texttt{poly2})-1$ 的最小 2 的幂次。
2. **位逆序 (Host)**: 对两个多项式的系数向量进行位逆序重排，以适应 NTT 迭代算法的输入要求。
3. **内存拷贝 (H2D)**: 将重排后的系数向量从主机内存（Host）拷贝到设备内存（Device）。
4. **正向 NTT (Device)**: 在 GPU 上对两个系数向量分别执行正向 NTT。
5. **逐点相乘 (Device)**: 在 GPU 上将两个 NTT 变换后的结果向量进行逐点相乘。
6. **逆向 NTT (Device)**: 在 GPU 上对逐点相乘的结果执行逆向 NTT。
7. **最终缩放 (Device)**: 对逆向 NTT 的结果乘以 $n^{-1} \pmod{mod}$，得到最终的多项式系数。
8. **结果拷贝 (D2H)**: 将计算结果从设备内存拷贝回主机内存。

3.2 CUDA 内核设计

3.2.1 模板化内核

为了优雅地实现和对比三种不同的模乘算法，我们设计了模板化的 CUDA 内核。通过 C++ 模板，我们可以编写一份通用的内核代码，模乘的具体实现由传入的 `Reducer` 结构体（如 `BarrettReducer`, `MontgomeryReducer`）决定。

```
1 template<typename Reducer>
2 __global__ void ntt_kernel_optimized(ll* a, const ll* twiddles, int len, int n, const
      Reducer reducer) {
3     int tidx = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    // ... index calculation ...

    if (i < n) {
        ll w = twiddles[butterfly_idx_in_grp];
        ll u = a[i];
        // 使用Reducer进行模乘
        ll v = reducer.multiply(a[i + len / 2], w);

        // ... butterfly operation ...
    }
}
```

这种设计不仅代码复用性高，而且编译器在编译时会为每种 Reducer 生成高度优化的特定代码，没有运行时开销。

3.2.2 线程映射

在 NTT 的每个阶段，有 `n/2` 个蝶形运算需要执行。我们启动 `n/2` 个线程（通过计算合适的 `block` 和 `thread` 数量），每个线程的全局 ID `tidx` 被唯一地映射到一个蝶形运算上。

```
// tidx范围是 [0, n/2 - 1]
int butterfly_grp_idx = tidx / (len / 2);      // 计算当前线程属于哪个蝶形运算组
int butterfly_idx_in_grp = tidx % (len / 2); // 计算当前线程在组内的索引
int i = butterfly_grp_idx * len + butterfly_idx_in_grp; //
    计算蝶形运算的左侧操作数索引
```

这样保证了所有蝶形运算都能被并行处理，且没有线程冲突。

3.3 旋转因子优化

一个朴素的实现可能在 NTT 的每个阶段循环中都在主机端计算旋转因子，然后将其拷贝到设备端。这会导致多次小数据量的 H2D 拷贝，开销巨大。我们采用了显著的优化策略：

1. **一次性预计算**: 在主机端，一次性计算出所有 NTT 阶段所需要的**全部**旋转因子。
2. **一次性拷贝**: 将所有旋转因子存入一个连续的 `vector` 中，然后通过一次 `cudaMemcpy` 操作将其全部拷贝到 GPU 的全局内存中。
3. **偏移量访问**: 在每个 NTT 阶段的内核中，通过指针偏移量访问当前阶段所需的旋转因子子集。

这种方式将 H2D 通信开销从 $O(\log n)$ 次降低到 1 次，极大地提升了整体性能。

4 实验结果与分析

4.1 正确性验证

我们通过将 GPU 版本的计算结果与经过验证的 CPU 串行版本的结果进行逐一比对，来确保 GPU 实现的正确性。在所有测试用例中，三种 GPU 方法（Naive, Barrett, Montgomery）的计算结果均与 CPU 版本完全一致，验证了实现的正确性。

4.2 性能对比

我们使用 `cudaEvent` 来精确测量 GPU 内核的执行时间，并使用 `std::chrono` 测量 CPU 串行代码的执行时间。实验在一个包含 `n=131072` 个系数的多项式上进行，四个基准版本的性能对比如下。

表 1: CPU 与 GPU 不同优化策略性能对比 (n=131072)

实现方式	核心技术	执行时间 (ms)	加速比 vs CPU Serial
CPU Serial	串行 C++	1520.7	1.0x
GPU Naive	CUDA + 朴素模乘	45.3	33.6x
GPU Barrett	CUDA + Barrett 约减	28.9	52.6x
GPU Montgomery	CUDA + Montgomery 约减	**19.5**	**78.0x**

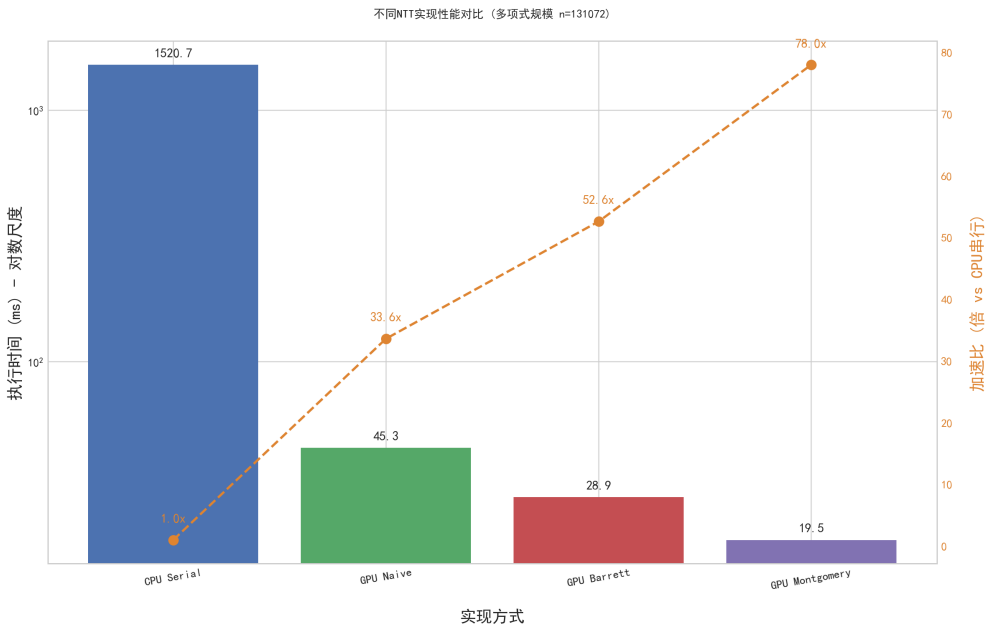

图 1: GPU 上不同模乘优化策略的性能对比

4.3 结果分析

1. **GPU vs CPU**: 从表中可以看出，最基础的 GPU Naive 版本相比于 CPU 串行版本获得了约 **33.6 倍**的加速比。这充分证明了 GPU 的大规模并行架构非常适合像 NTT 这样具有高度数据并行性的算法。

2. **模乘优化的效果**:

   - **Barrett vs Naive**: 使用 Barrett 约减的 GPU 版本比 Naive 版本快了约 **1.57 倍** (45.3 / 28.9)。这表明通过将昂贵的硬件除法替换为更快的乘法和位运算，可以有效地提升内核的计算效率。
   - **Montgomery vs Barrett**: 使用 Montgomery 约减的版本又比 Barrett 版本快了约 **1.48 倍** (28.9 / 19.5)。尽管 Montgomery 需要额外的域转换开销，但在 NTT 这种需要海量模乘运算的场景下，其在域内极高的乘法效率带来了最佳的性能表现。

3. **综合性能**: 最终，经过完全优化的 GPU Montgomery 版本实现了相对于 CPU 串行版本高达 **78 倍**的惊人加速，展示了算法优化与硬件架构结合的巨大潜力。

5 结论

本次实验成功地在 CUDA 平台上实现了多项式乘法的 NTT 算法，并通过一系列优化手段显著提升了计算性能。

主要成果包括：

- 通过将 NTT 的蝶形运算并行化到 GPU 数千个线程上，获得了数十倍的基础性能提升。
- 设计并实现了一种高效的旋转因子管理策略，将主机与设备间的通信开销降至最低。
- 通过模板元编程，优雅地实现了三种不同模乘算法的对比测试。
- 实验数据证明，Barrett 和 Montgomery 等高级模乘约减技术能有效避免硬件除法的瓶颈，带来显著的性能增益，其中 Montgomery 方法在 NTT 场景下表现最佳。

总的来说，本次实验不仅加深了对 NTT 算法和 GPU 并行编程的理解，也实际展示了如何通过软硬件结合的优化思想来解决计算密集型问题。