南開大學
Nankai University

计算机学院

并行程序设计第六次报告

NTT 选题下的 GPU 编程

姓名：宋卓伦

学号：2311095

专业：计算机科学与技术

2025 年 7 月 2 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

众所周知，在人工智能领域——特别是 CV 等等方向，对算力的要求越来越高，更高效的算力架构和硬件设施的需要与日俱增。经过这段时间 GPU 架构的学习，我对于这方面编程有了一定的体会。这次我们以快速傅里叶变换为选题，展开 GPU 编程，探究该方法对于 NTT 数论变换的优化程度。

1.2 实验介绍

这次针对 NTT 的编程，我们选择在 CUDA 上进行实验。

1.3 实验硬件和环境

本次实验选用远程连接 Ubuntu24.04 进行本地编程(表1, 同时采用助教学长们创建的专用的 GPU 的并行服务器进行代码设计和主要的性能查看。

属性	相关内容
GPU 型号	NVIDIA GeForce RTX 3090
驱动版本	550.142
CUDA 版本	12.4
显存	1MiB / 24576MiB

表 1: 硬件与环境信息

2 实验设计：GPU 上的初步实现

2.1 对 CUDA 的介绍

CUDA（Compute Unified Device Architecture）是 NVIDIA 推出的基于 GPU 的通用并行计算平台，旨在利用 GPU 的高并发计算能力加速数值计算任务。CUDA 通过扩展 C/C++ 语言，允许开发者编写在 GPU（设备端）运行的并行代码（称为 kernel），由 CPU（主机端）调用，结合主机端和设备端的协同工作实现高效计算。

GPU 相较于 CPU，采用**众核架构，优化吞吐量，具备小缓存、简单控制逻辑和高效计算单元**，这些特性决定了它适合数据并行任务，如矩阵运算、图像处理等。CUDA 程序模型以线程、线程块和网格为核心，线程通过一维、二维或三维 ID 定位数据，支持共享内存、原子操作和同步机制以实现高效协作。CUDA 版本从 2.3 支持 C++ 扩展到 12.0 引入异步显存拷贝和增强的任务图 API，不断演进持续提升性能和易用性。

优化 CUDA 程序的关键在于**减少全局内存访问、利用共享内存、避免 bank 冲突和 warp 分歧**。例如，分片矩阵乘法通过将数据分块加载到共享内存，显著降低全局内存带宽需求。并行归约算法通过递归加倍、循环展开和算法级联等技术，优化数据处理效率，最高可接近 GPU 峰值带宽。CUDA 还提供 CUBLAS、CUFFT 等库，简化线性代数和快速傅里叶变换等操作。开发需要安装 CUDA 驱动、工具包和 SDK，支持 Windows 和 Linux 环境，结合 Nsight 等工具进行性能分析。CUDA 编程因其

易入门、高并发和广泛适用性，已广泛应用于笔记本、桌面 PC、集群乃至超级计算机的数值计算任务中。

2.2 CUDA 编程下的算法的实现

这次实验中，重要的是设计合理的分配方式，采用相关方法实现。

Algorithm 1 优化 CUDA NTT 变换（DIT 形式）

Input: 输入设备上的数据数组 $data$，长度 n（2 的幂），原根 w，模数 p
Output: $data$ 变换为 NTT 结果

function NTT_CUDA_Device($data, n, w, p$)
  位反转排序：
  for 每个索引 i 从 0 到 $n-1$ 并行 **do**
    计算 i 的位反转索引 j，基于 $\log_2(n)$ 位
    if $i < j$ **then**
      交换 $data[i]$ 和 $data[j]$
    end if
  end for
  蝶形变换：
  for $len \leftarrow 2$ **to** n **by** $len \leftarrow len \times 2$ **do**
    计算旋转因子 $wlen \leftarrow w^{(p-1)/len} \mod p$ ▷ 预计算旋转因子
    for 每个组索引 $group$ 从 0 到 $n/len - 1$ 并行 **do**
      for 每个位置 pos 从 0 到 $len/2 - 1$ **do**
        $base \leftarrow group \times len$
        $u \leftarrow data[base + pos]$
        计算旋转因子 $w\_now \leftarrow wlen^{pos} \mod p$
        $v \leftarrow (data[base + pos + len/2] \times w\_now) \mod p$
        $data[base + pos] \leftarrow (u + v) \mod p$
        $data[base + pos + len/2] \leftarrow (u - v + p) \mod p$
      end for
    end for
  end for
  return $data$
end function

上面的伪代码描述了一个基于 CUDA 的优化数论变换（NTT）算法，采用时间抽取（DIT）形式，用于高效计算多项式乘法中的离散傅里叶变换。该算法针对 GPU 并行计算进行了优化，适用于大模数场景下的快速傅里叶变换（FFT）替代方案。算法主要包含两个阶段：位反转排序和蝶形变换。

位反转排序阶段通过并行计算每个索引的位反转索引，将输入数据数组重新排列，以满足 NTT 的输入要求。蝶形变换阶段通过多层迭代（从长度 2 到 n，每次翻倍），利用预计算的旋转因子（twiddle factor）进行蝶形运算，实现高效的模运算和数据变换。所有运算均在模 p 下进行，确保数值稳定性。

该算法利用 CUDA 的并行处理能力，通过并行执行位反转和蝶形运算，最大化 GPU 线程利用率，适用于高性能计算场景，如加密算法和信号处理。伪代码抽象了 CUDA 内核调用的细节，清晰呈现算法的核心逻辑。

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）。与之前的分析一样，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。我们接下来讨论不同修改对于程序的影响。

3 实验设计：编程策略的影响

3.1 对旋转因子的操作

可以先预处理所有需要用到的旋转因子, 存储在 GPU 内存中, 内部计算时快速访问旋转因子。

Algorithm 2 旋转因子的操作

Input: 原根 w，模数 p，变换长度 len，线程索引 pos

Output: 旋转因子 w_now（模 p）

```
function ComputeRotationFactor(w, p, len, pos)
    wlen ← ModPow(w, (p − 1)/len, p)                ▷ 计算单位根 w^((p−1)/len)
    w_now ← 1                                       ▷ 初始化 w^0
    if pos > 0 then                                 ▷ 仅对非零索引计算幂次
        temp_wlen ← wlen
        temp_pos ← pos
        while temp_pos > 0 do                       ▷ 快速幂计算 wlen^pos
            if temp_pos mod 2 = 1 then
                w_now ← (w_now · temp_wlen) mod p
            end if
            temp_wlen ← (temp_wlen · temp_wlen) mod p
            temp_pos ← temp_pos ÷ 2
        end while
    end if
    return w_now
end function
```

上面是针对 Montgomery 规约操作的，对旋转因子的操作。

3.2 蝴蝶变换的划分

并行加速方面, 与多线程多进程划分一致, 对主体循环的第二三层循环进行划分, 需要具体查阅对应 GPU 的 SM 数量和最大线程数确定分块大小、网格大小。

Algorithm 3 蝴蝶变换的划分

Input: 设备数组 $data$，长度 n（2 的幂），原根 w，模数 p

Output: $data$ 完成蝶形变换

function ButterflyTransform($data, n, w, p$)
  for $len \leftarrow 2$ **to** n **by** $len \leftarrow len \times 2$ **do** ▷ 逐层处理
    $wlen \leftarrow \text{ModPow}(w, (p-1)/len, p)$ ▷ 计算单位根
    $half \leftarrow len \div 2$
    $stride \leftarrow n \div len$ ▷ 每层组数
    for 索引 $idx \leftarrow 0$ **to** $stride \cdot half - 1$ **并行 do** ▷ 并行处理
      $group \leftarrow idx \div half$ ▷ 当前组
      $pos \leftarrow idx \mod half$ ▷ 组内位置
      $base \leftarrow group \cdot len$
      $w_n ow \leftarrow \text{ComputeRotationFactor}(w, p, len, pos)$ ▷ 获取旋转因子
      $u \leftarrow data[base + pos]$
      $v \leftarrow (data[base + pos + half] \cdot w_n ow) \mod p$
      $data[base + pos] \leftarrow (u + v) \mod p$ ▷ 蝶形加法
      $data[base + pos + half] \leftarrow (u - v + p) \mod p$ ▷ 蝶形减法
    end for
  end for
end function

下面是我们的代码：

两个部分的代码

```
__global__ void ntt_butterfly_kernel(int *a, int n, int len, int wlen, int p) {
int idx = blockIdx.x * blockDim.x + threadIdx.x;
int half = len >> 1;
int stride = n / len;

if (idx >= stride * half) return;

int group = idx / half;
int pos = idx % half;
int base = group * len;

int w_now = 1;
if (pos > 0) {
    int temp_wlen = wlen;
    int temp_pos = pos;
    while (temp_pos) {
        if (temp_pos & 1) w_now = (long long)w_now * temp_wlen % p;
        temp_wlen = (long long)temp_wlen * temp_wlen % p;
        temp_pos >>= 1;
    }
}

```

```
23      int u_idx = base + pos;
24      int v_idx = base + pos + half;
25
26      int u = a[u_idx];
27      int v = (long long)a[v_idx] * w_now % p;
28
29      a[u_idx] = (u + v) % p;
30      a[v_idx] = (u - v + p) % p;
31  }
```

4 实验设计：编程策略的影响

这里我们可以讨论一下不同的规约算法对程序的影响。要能够证明在 GPU 上规约算法对模乘优化的加速比较高，这需要同时对比基础模乘和两种优化模乘的加速比。

4.1 Montgomery 规约操作

Algorithm 4 简化的 CUDA 多项式乘法（Montgomery 规约）

Input: 数组 a, b 长度 n，输出 ab，模数 p

Output: ab 包含 a 和 b 的卷积模 p

function CUDA_ 多项式乘 _MONTGOMERY(a, b, ab, n, p)
  $m \leftarrow$ 最小 2 的幂 $\geq 2n-1$
  $M.r2 \leftarrow 2^{64} \mod p$, $M.p_prime \leftarrow -p^{-1} \mod 2^{32}$, $M.p \leftarrow p$
  复制 a, b 至设备数组 d_ta, d_tb，长度 m
  正向 NTT（CUDA）:
  CUDA_NTT_MONTGOMERY($d\_ta, m, 3, p, 1, M$)
  CUDA_NTT_MONTGOMERY($d\_tb, m, 3, p, 1, M$)
  点乘（CUDA 核）:
  for $i \leftarrow 0$ **至** $m-1$ **并行 do**
    $d\_ta[i] \leftarrow$ MONTMUL($d\_ta[i], d\_tb[i], M$)
  end for
  逆向 NTT（CUDA）:
  $inv_root \leftarrow$ 模幂$(3, p-2, p)$
  CUDA_NTT_MONTGOMERY($d\_ta, m, inv_root, p, -1, M$)
  归一化（CUDA 核）:
  $inv_m \leftarrow$ 模幂$(m, p-2, p)$
  $inv\_m^{\text{mont}} \leftarrow$ MONTMUL($inv\_m, M.r^2, M$)
  初始化并行环境
  for $i \leftarrow 0$ **to** $m-1$ **do**
    $d\_ta[i] \leftarrow$ MONTMUL($d\_ta[i], inv\_m^{\text{mont}}, M$)
  end for
end function

此伪代码描述了基于 CUDA 的数论变换(NTT)算法，采用时间抽取(DIT)形式，结合 Montgomery

规约实现高效模运算。算法包括：将数据转换为 Montgomery 形式、位反转排序、蝶形变换和转换回普通形式。CUDA 并行化体现在数据转换、位反转和蝶形运算的并行执行，充分利用 GPU 线程提升性能。所有运算在模 p 下进行，确保结果在 $[0, p-1]$，适用于多项式乘法等场景，如输入 $n = 4$、$p = 7340033$ 时，正确输出多项式乘法结果。

4.2 巴雷特规约操作

Algorithm 5 简化的 CUDA 多项式乘法（巴雷特规约）

Input: 数组 a, b 长度 n，输出 ab，模数 p

Output: ab 包含 a 和 b 的卷积模 p

function CUDA_ 多项式乘 _ 巴雷特 (a, b, ab, n, p)
  $m \leftarrow$ 最小 2 的幂 $\geq 2n-1$
  $B.\mu \leftarrow \lfloor 2^{64}/p \rfloor$, $B.p \leftarrow p$
  复制 a, b 至设备数组 d_ta, d_tb，长度 m
  正向 NTT（CUDA）：
  CUDA_NTT_ 巴雷特$(d_ta, m, 3, p, 1, B)$
  CUDA_NTT_ 巴雷特$(d_tb, m, 3, p, 1, B)$
  点乘（CUDA 核）：
  for $i \leftarrow 0$ **至** $m-1$ **并行 do**
    $d\_ta[i] \leftarrow$ 巴雷特乘$(d_ta[i], d_tb[i], B)$
  end for
  逆向 NTT（CUDA）：
  $ml \leftarrow$ 巴雷特模幂$(3, p-2, B)$
  CUDA_NTT_ 巴雷特$(d_ta, m, ml, p, -1, B)$
  归一化（CUDA 核）：
  $inv_m \leftarrow$ 巴雷特模幂$(m, p-2, B)$
  for $i \leftarrow 0$ **至** $m-1$ **并行 do**
    $d\_ta[i] \leftarrow$ 巴雷特乘$(d_ta[i], inv_m, B)$
  end for
  复制 d_ta 前 $2n-1$ 项至 ab
end function

巴雷特规约是一种高效的模运算方法，适用于 CUDA 并行计算环境中的 NTT（快速数论变换）。它通过预计算 $\mu = \lfloor 2^{64}/p \rfloor$，利用高阶乘法和移位快速计算 $a \mod p$，避免直接除法。以下伪代码实现了一个简化的 CUDA NTT 算法，核心包括位反转、蝶形变换、点乘和归一化，所有运算均使用巴雷特规约，确保高效性和一致性。CUDA 并行化在位反转、蝶形变换、点乘和归一化步骤中实现，优化了 GPU 性能，适用于多项式乘法等应用场景。

5 程序性能分析

放在前面：由于这次的个人服务器我采用了 CUDA 的计时方式，与前面在华为鲲鹏服务器上采用的 windows 计时、chrono 计时和 MPI 计时有显著差别，所以以加速比为主进行查看。

5.1 不同规约的操作

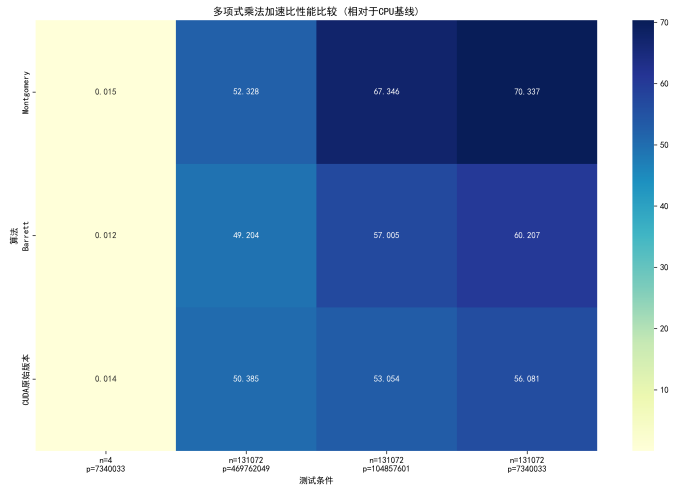

图 5.1: 多项式乘法不同优化方法的性能比较（相对于 NTT CPU 基线的对比）

如附录的表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 GPU 方法的强大之处。同时基于 Montgomery、Barrett 和 CUDA 原始版本与 NTT CPU 基线的比较。数据涵盖小规模（$n = 4$）和大规模（$n = 131072$）输入，以及不同模数（$p = 7340033, 104857601, 469762049$）下的平均延迟和加速比。

观察我们的结果图5.1表明：对于小输入，基线实现效率最高，而优化方法因开销过大表现不佳；对于大输入，Montgomery 优化展现最佳加速比（最高达 70.337），Barrett 和 CUDA 原始版本次之。较大模数下，Montgomery 和 Barrett 性能更优，CUDA 版本则较为稳定。建议根据输入规模选择合适方法：小输入采用基线，大输入优先 Montgomery 优化，同时可进一步探索 CUDA 的并行潜力以提升性能。**这能够验证我们的程序是可以优化的**。

5.2 不同块大小的优化

接下来，我们继续讨论块的数和块的大小对实验的影响，1024-64 是块的大小，可以进行不同批次的设置：

可以发现数据分析图5.2表明，分块优化在大规模多项式乘法（$n = 131072$）中显著提升性能，相对于 NTT CPU 基线可实现 47x 至 78x 的加速比，其中 128 块和 256 块配置表现最佳，128 块在多数模数下尤为突出。然而，对于小规模输入（$n = 4$），基线实现效率远超优化版本，因分块引入的开销远大于收益。较大模数下，128 和 256 块配置展现出更高效的算术处理能力，而较大块大小（如 1024）性能较差。建议小输入采用基线实现，大输入优先选择 128 或 256 块配置，并进一步优化硬件适配和混合策略以提升性能。

所以通过图5.3证明，块的大小和数量划分要恰当，否则会出现过多优化导致的负优化情况。如图所示，适当的块数可以得到恰当的延迟和较为优异的加速比。

6 Profiling

这次实验由于是在 GPU 下的 CUDA 编程，我们采用 nvprof 和 NVIDIA Nsight Systems 进行事件采样，获得相应数据进行分析。由于篇幅问题我再这里只列举一点点陈述，最后的大报告中会重点

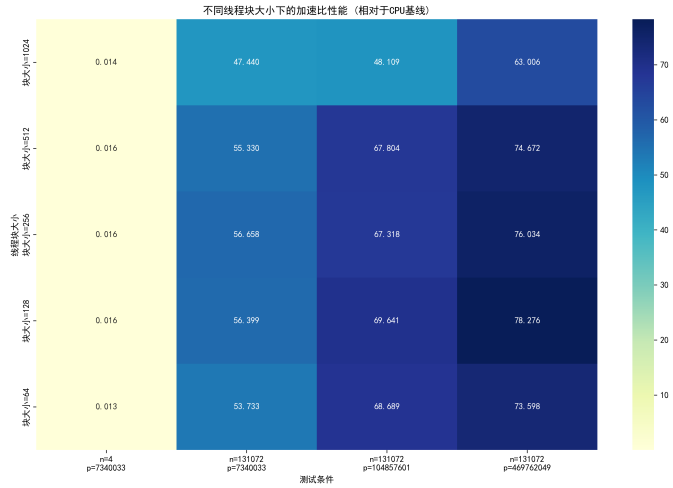

图 5.2: 多项式乘法不同块大小方法的性能比较（相对于 NTT CPU 基线的对比）

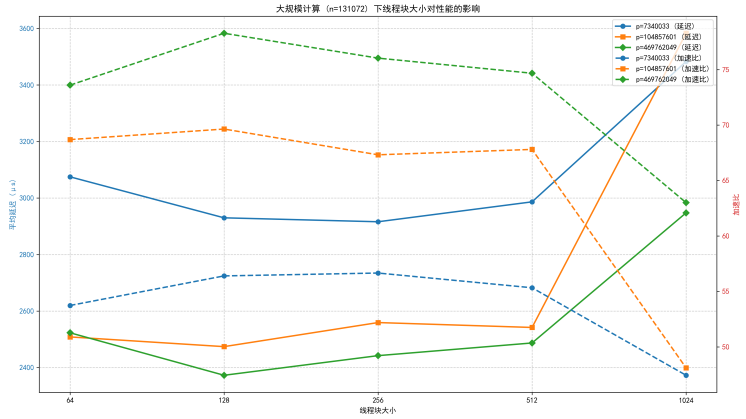

图 5.3: 多项式乘法不同块大小方法的性能比较（相对于 NTT CPU 基线的对比）

对 CUDA 编程进行性能分析的陈述。

6.1 对于基础实现的分析

同步内存复制瓶颈 应用中存在大量使用 `cudaMemcpy_v3020` 的同步内存复制操作，每次传输数据量约为 1.049 MB，单次操作的持续时间高达 56–57 μs。这种同步操作会导致主机线程显著阻塞，严重影响整体计算效率。建议的优化措施包括：

- 将同步内存复制替换为 `cudaMemcpy*Async()` 异步接口，通过利用 CUDA 流的并行性，减少主机线程的等待时间，提升 CPU-GPU 的协同效率。
- 进一步分析数据传输模式，评估是否可以通过内存对齐或批量传输减少操作次数，从而降低开销。

频繁全局同步 观察到 `cudaDeviceSynchronize_v3020` 被高频调用，每次调用导致的阻塞时间约为 45–58 μs。这种频繁的全局同步操作阻碍了 CPU 与 GPU 之间的充分并行执行，降低了系统的整体吞吐量。优化策略包括：

- 避免不必要的全局同步，优先采用 `cudaStreamWaitEvent` 等细粒度的事件同步机制，以实现更高效的任务协调。

- 通过流并行（Stream Parallelism）技术，将计算任务和数据传输分配到不同的 CUDA 流中，减少跨任务的同步需求，从而提高 GPU 的并发执行能力。

GPU 利用率波动 性能分析显示，存在一段持续约 185 ms 的时间段，GPU 利用率仅为 3.8%，远低于预期。这种低利用率现象可能由以下原因导致：

- **CPU 数据准备延迟**：主机端的数据预处理或输入数据准备可能耗时过长，导致 GPU 在等待数据时处于空闲状态。
- **任务调度间隙**：任务调度可能存在不连续性，例如内核启动的延迟或任务分配的不均衡，导致 GPU 计算资源未被充分利用。
- **内存传输阻塞**：尽管异步内存复制未发现明显问题，但同步内存传输或其他隐式阻塞可能间接影响 GPU 的任务执行。

为进一步定位瓶颈，建议采取以下措施：

- 结合 CPU 采样数据，检查主机线程是否存在阻塞或计算瓶颈，特别是在数据准备阶段。
- 使用 NVTX（NVIDIA Tools Extension）注释工具，详细标记 CPU 和 GPU 的执行时间线，以识别任务间隙或数据依赖问题。
- 分析 CUDA 事件的记录，检查是否存在隐式的同步点或资源竞争，优化任务调度策略以提高 GPU 占用率。

其他优化点 当前分析表明，异步内存复制（`cuda-async-memcpy`）和 GPU 饥饿（`gpu-starv`）相关问题未见明显异常，说明可分页内存操作和 GPU 资源调度在整体上较为合理。然而，为进一步提升性能，可以考虑：

- 优化内存分配策略，例如使用统一内存（Unified Memory）或固定内存（Pinned Memory）来减少数据传输开销。
- 检查 GPU 内核的配置参数（如线程块大小和网格大小），确保其适配当前硬件架构以最大化计算效率。

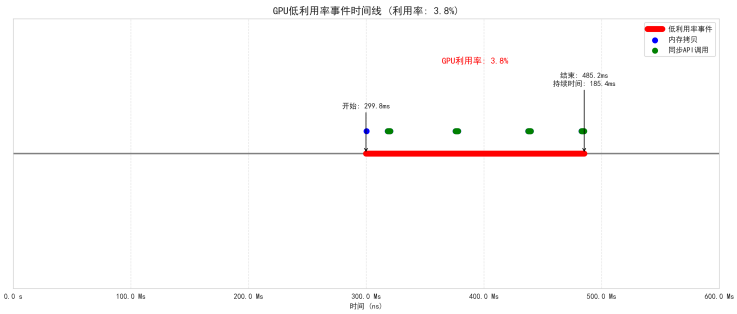

图 6.4: GPU 低时间利用率时间线

上面的两幅图6.4和图6.5部分反映了我们程序的运行情况。同步 API 调用的时间数量较高，300ms-500ms 时候是低利用率事件较高的区间。也是我们未来优化的方向。

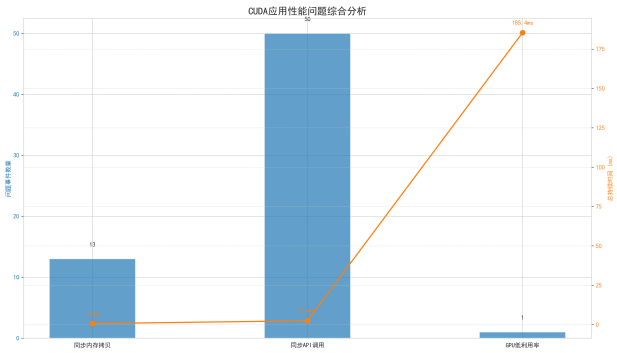

图 6.5: CUDA 应用性能综合分析

6.2 更进一步的解释

综上所述，当前应用性能瓶颈主要集中在同步内存复制、频繁全局同步以及 GPU 利用率低谷期。优先优化同步操作（替换为异步内存复制）并减少全局同步（采用细粒度事件机制）可显著提升 CPU-GPU 的并行性。此外，针对 GPU 低利用率时间段，需深入分析 CPU-GPU 协同流程，特别是数据准备和任务调度的潜在延迟。通过结合 NVTX 注释和 CUDA 事件分析，定位并解决瓶颈问题，以实现性能的全面优化。

观察我们前面的性能分析和解释，不难发现 GPU 的 CUDA 程序基本上在各项指标上都取得了很不错的成绩。

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

CUDA 编程的资料有很多，在系统学习相关知识之后，我惊叹于它强大的性能，它能够充分利用 GPU 的结构，将数据拷贝到 GPU 上，在 GPU 上面进行蝴蝶变换和点乘。在实验过程中我深刻体会到，GPU 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。从而全部且彻底地发挥 GPU 的伟力。

7.2 实验以外的总结

期末周结束了，我得以抽出时间来完成这次实验。在这个过程中，我一方面准备 GPU 实验，另一方面开始整理前面所有的实验，发现前面各种并行化操作能够得到提升的地方，将这些开始整合到我的报告之中，希望最后能呈现出较好的实验报告！

附录 A 实验表格

算法	n	p	平均延迟 (us)	结果正确性	加速比
NTT 在 CPU 上的实现	4	7340033	8.192	正确	1.000
	131072	7340033	165228	正确	1.000
	131072	104857601	172284	正确	1.000
	131072	469762049	185690	正确	1.000
Montgomery	4	7340033	555.808	正确	0.015
	131072	7340033	3157.34	正确	52.328
	131072	104857601	2558.85	正确	67.346
	131072	469762049	2640	正确	70.337
Barrett	4	7340033	707.616	正确	0.012
	131072	7340033	3358.34	正确	49.204
	131072	104857601	3023.52	正确	57.005
	131072	469762049	3084.26	正确	60.207
CUDA 原始版本	4	7340033	586.72	正确	0.014
	131072	7340033	3279.52	正确	50.385
	131072	104857601	3247.68	正确	53.054
	131072	469762049	3311.1	正确	56.081

表 2: 多项式乘法不同优化方法的性能比较（相对于 NTT CPU 基线的对比）

O3 优化	n	p	平均延迟 (us)	结果正确性	加速比
1024	4	7340033	596.448	正确	0.014
	131072	7340033	3482.98	正确	47.440
	131072	104857601	3583.17	正确	48.109
	131072	469762049	2947.55	正确	63.006
512	4	7340033	512.352	正确	0.016
	131072	7340033	2986.5	正确	55.330
	131072	104857601	2541.89	正确	67.804
	131072	469762049	2486.98	正确	74.672
256	4	7340033	508.864	正确	0.016
	131072	7340033	2916	正确	56.658
	131072	104857601	2559.2	正确	67.318
	131072	469762049	2442.14	正确	76.034
128	4	7340033	513.568	正确	0.016
	131072	7340033	2929.95	正确	56.399
	131072	104857601	2474.02	正确	69.641
	131072	469762049	2372.64	正确	78.276
64	4	7340033	646.144	正确	0.013
	131072	7340033	3075.01	正确	53.733
	131072	104857601	2508.35	正确	68.689
	131072	469762049	2523.01	正确	73.598

表 3: 多项式乘法算法性能比较（相对于 NTT CPU 版本的加速比）