南開大學
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

作业六：NTT 算法 GPU 加速

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 7 月 2 日

摘要

本文研究了基于 **GPU 并行架构**的 NTT 加速方法。通过 CPU 串行 NTT 实现基准，在 NVIDIA GPU 上开发了多层次优化方案：首先依据 **SIMT 执行模型**设计线程分级策略，将蝶形运算映射至 CUDA 线程层次，实验确定 **128 线程/块**为最优配置，实现 **80.6**× 加速比；其次引入 **Barrett 模乘**算法替代传统取模运算，避免 Montgomery 算法的数据域转换开销，在 GPU 上获得 **1.83**× 加速；进一步通过预计算蝶形因子消除实时幂运算、异步内核执行消除同步等待、共享内存混合策略聚合全局访问等深度优化，带来额外 **1.18**× 性能提升。跨平台测试（Tesla T4/RTX 3090/RTX 4090）显示性能差异仅 **6%**，结合 **nvprof 剖析**证实计算内核占比 **96.23%** 且内存带宽是核心瓶颈。实验代码及图片已全部上传至：

https://github.com/eprogressing/NKU_COSC0025_Parallel

关键词：GPU 并行计算，NTT，CUDA 优化，Barrett 模乘，内存带宽

目录

1 GPU 并行计算原理与架构

1.1 CPU 与 GPU 的架构差异

CPU（中央处理器）和 GPU（图形处理器）在设计哲学上存在根本性的差异，这直接决定了它们各自擅长的计算任务类型。

1.1.1 面向延迟的 CPU 架构

CPU 的设计目标是**最小化单任务延迟**。它拥有少量（几个到几十个）功能强大且复杂的计算核心。这些核心配备了庞大的缓存和复杂的控制单元，能够高效地处理复杂的逻辑分支、指令级并行和乱序执行，从而以最快的速度完成单个任务流。如图 1.1 (a) 所示，其大部分芯片面积被用于控制和缓存。

1.1.2 面向吞吐量的 GPU 架构

与此相反，GPU 的设计目标是**最大化并行任务吞吐量**。它集成了数以百计甚至数以千计的、结构相对简单的计算核心。这些核心共享控制单元和缓存，牺牲了单核的复杂性和处理分支的能力，以换取在芯片上集成更多的计算单元。这种设计使得 GPU 能够同时处理海量的、彼此独立的简单计算任务。如图 1.1 (b) 所示，GPU 的绝大部分芯片面积都用于计算。

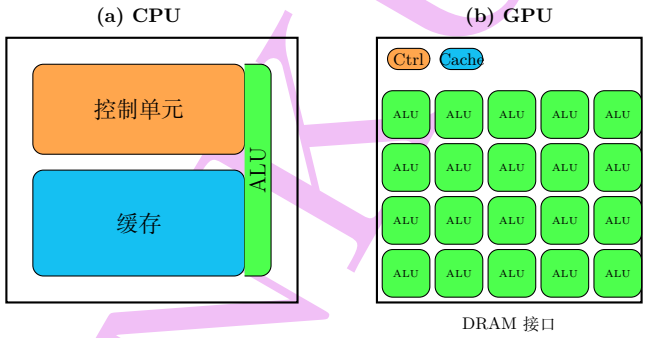

图 1.1: CPU 与 GPU 架构设计理念对比示意图

NTT 算法中的蝶形运算具有高度的数据并行性：每一层的上万个蝶形运算都可以独立进行，没有复杂的逻辑依赖。这种特性与 GPU 的大规模并行架构完美契合，是 GPU 能够显著加速 NTT 计算的根本原因。

1.2 SIMT 执行模型

NVIDIA GPU 采用一种称为**单指令多线程（SIMT）**的执行模型。这是其并行能力的核心。

- **Warp**: 在 CUDA 中，32 个线程会被组合成一个线程束（Warp）。Warp 是 GPU 上最基本的调度和执行单元。
- **统一指令**: 在一个 Warp 内，所有 32 个线程在同一时刻执行相同的指令，但处理的是不同的数据。
- **线程分化**: 如果 Warp 内的线程遇到分支语句（如 'if-else'）并选择了不同的路径，硬件会串行地执行每一个分支路径，并屏蔽掉该路径下不活跃的线程。这会导致部分计算资源闲置，降低性能。因此，在编写 CUDA 核函数时，应尽量避免 Warp 内的线程分化。

NTT 算法的结构相对规整，蝶形运算的模式统一，可以有效地避免线程分化，从而充分发挥 SIMT 模型的威力。

1.3 CUDA 编程模型与硬件映射

CUDA 提供了一个抽象的编程模型，让开发者能够组织和管理成千上万的线程。该模型主要包含三个层次，并与 GPU 硬件紧密对应。

- **线程**：最基本的执行单位，通常执行一个简单的计算任务，如一次蝶形运算中的一个点的计算。
- **线程块**: 由一组线程构成。同一个线程块内的线程可以相互协作，通过快速的片上共享内存交换数据，并且可以通过 '___syncthreads()'进行同步。线程块被作为一个整体调度到 GPU 的一个流式多处理器（SM）上执行。
- **网格**：由一组线程块构成。一个核函数的启动会生成一个网格。不同线程块之间默认无法直接通信和同步。

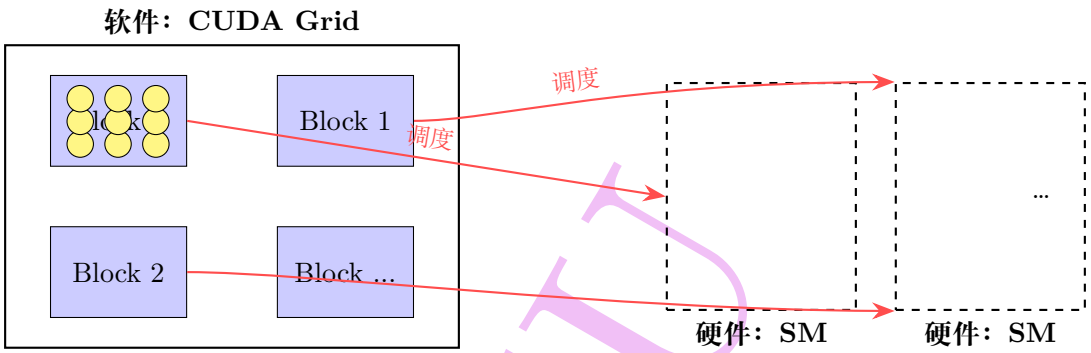

图 1.2: CUDA 编程模型（左）到 GPU 硬件（右）的映射关系

这种层次化的模型允许开发者根据问题的并行特性进行任务分解。对于 NTT，我们可以将整个变换看作一个 Grid，将其中一个计算阶段或一部分数据分配给一个 Block，再将具体的蝶形运算分配给 Block 内的 Threads。合理地组织 Grid 和 Block 的尺寸，并高效利用共享内存，是实现高性能 CUDA 程序的关键，也是本报告后续章节将要深入探讨的核心内容。

2 GPU 加速 NTT 的实现与优化

2.1 CPU 串行 NTT 实现与分析

为了准确评估 GPU 加速的效果，我们不能将性能与实验框架中 $O(n^2)$ 的朴素乘法作比较，因为这无法体现算法层面的改进。因此，我们首先需要实现一个高效的、时间复杂度为 $O(n \log n)$ 的 CPU 串行 NTT 算法。这个实现将作为我们后续所有 GPU 优化版本的“黄金标准”和 Baseline。

一个标准的迭代式 Cooley-Tukey NTT 算法主要包含以下两个步骤：

1. **位逆序置换**：在迭代式 NTT 中，输入数据需要先按照其索引的二进制位翻转后的顺序进行重新排列。例如，对于 8 点 NTT，索引为 2(010b) 的元素需要与索引为 4(100b) 的元素交换位置。这一步是为了保证后续的蝶形运算能够原址进行且数据依赖正确。

2. **迭代蝶形运算**：位逆序完成后，算法将进行 $\log_2(n)$ 个阶段的计算。在每个阶段，数据被分成若干组，组内进行蝶形运算。随着阶段的推进，蝶形运算的“跨度”不断增大，直到最后完成。

逆 NTT（INTT）的过程与正向 NTT 高度相似，只需将单位根 ω_n 替换为其逆元 ω_n^{-1}，并在最后将所有结果乘以 n 的逆元 n^{-1} 即可。两种模乘的具体推导已经在前面的实验详细说过了，这里就不详细赘述了。

2.2 测试结果与数据呈现

为了探究并行化对 NTT 算法的加速效果，并找到最优的 GPU 执行参数，本实验进行了一系列 GPU 加速测试。实验在不同 'threadsPerBlock' 配置下运行 NTT 多项式乘法代码，测试结果整理如表 1。该性能趋势由图 2.3 直观展示。

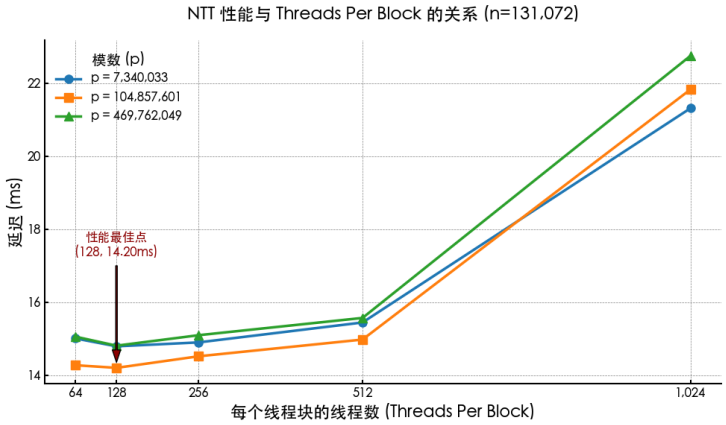

图 2.3: NTT 性能与 Threads Per Block 的关系 ($n = 131072$)

表 1: 不同 'threadsPerBlock' 配置下的性能测试结果

Threads Per Block	问题规模 (n)	模数 (p)	延迟 (ms)
测试案例 *1: $n = 4$, $p = 7340033$*			
64	4	7340033	309.450
128	4	7340033	306.647
256	4	7340033	305.911
512	4	7340033	**304.336**
1024	4	7340033	308.767
测试案例 *2: $n = 131072$, $p = 7340033$*			
64	131072	7340033	15.0086
128	131072	7340033	**14.7928**
256	131072	7340033	14.9005
512	131072	7340033	15.4459
1024	131072	7340033	**21.3131**
测试案例 *3: $n = 131072$, $p = 104857601$*			
64	131072	104857601	14.2761
128	131072	104857601	**14.2035**
256	131072	104857601	14.5199
512	131072	104857601	14.9784
1024	131072	104857601	**21.8278**
测试案例 *4: $n = 131072$, $p = 469762049$*			
64	131072	469762049	15.0572
128	131072	469762049	**14.8107**
256	131072	469762049	15.0973
512	131072	469762049	15.5707
1024	131072	469762049	**22.7487**

2.2.1 结果分析

根据上表数据，可得出以下关键结论：

1. **小规模问题 ($n = 4$) 的特殊性：** 对于 $n = 4$ 的情况，计算量极小，测得的延迟主要由 CUDA 内核启动、数据传输等固定开销主导，而非实际计算时间。因此，该案例的性能变化不具代表性，分析重点应置于大规模问题。

2. **大规模问题 ($n = 131072$) 的性能趋势：** 对于所有大规模测试案例，均呈现出一致的模式：
   - **性能最佳点：** 性能在 `threadsPerBlock = 128` 时达到**最佳**（延迟最低）。从 128 增加到 256 和 512，性能有轻微下降，但 128 线程/块无疑是最高效的配置。
   - **性能断崖式下跌：** 当 `threadsPerBlock` 增加到 **1024** 时，性能出现显著恶化，延迟大幅增加了约 50%。

3. **模数 (p) 的影响：** 仔细观察三个大规模测试案例可以发现，在相同的 'threadsPerBlock' 配置下，**模数 'p' 越大，计算延迟也越长**。例如，在最佳配置（128 线程/块）下，延迟从 p=104M 时的 14.20ms 增长到 p=469M 时的 14.81ms。虽然增幅微小，但这符合预期，因为更大的模数意味着取模运算涉及的数字更大，计算开销会略有增加。

2.2.2 与 CPU 基线的性能对比

为了量化 GPU 并行计算带来的优化效果，我们将最佳 GPU 性能（'threadsPerBlock=128'）与在相同问题规模下运行的 CPU 基线代码进行对比。

表 2: GPU 实现与 CPU 基线的性能对比 ($n = 131072$)

模数 (p)	CPU 延迟 (ms)	最佳 GPU 延迟 (ms)	加速比 (Speedup)
7340033	1176.14	14.7928	**79.5x**
104857601	1176.65	14.2035	**82.8x**
469762049	1175.63	14.8107	**79.4x**
平均加速比			**≈ 80.6x**

如上表所示，与 CPU 实现相比，经过优化的 CUDA NTT 代码展现了巨大的性能优势。在大规模问题上，GPU 实现的平均**加速比达到了约 80.6 倍**，这充分证明了将计算密集型的 NTT 算法移植到 GPU 平台进行并行化处理的显著有效性。

2.2.3 深入探讨：为何 1024 线程/块的效率显著下降？

此现象是典型的 GPU 资源限制问题，根本原因在于**占用率**降低，而占用率降低很可能是由**寄存器压力**过大导致。

核心理论：占用率与延迟隐藏 GPU 的 SM 通过在不同 Warp 之间快速切换来隐藏内存访问等操作带来的延迟。为了实现高效切换，SM 上必须同时驻留足够多的活跃 Warp。SM 能容纳的 Warp 总数受限于其物理资源，如寄存器文件、共享内存等。

寄存器成为瓶颈 一个线程块内的所有线程共享其所在 SM 的资源。假设 GPU 的每个 SM 拥有 65536 个 32-bit 寄存器，且内核编译后每个线程需使用 R 个寄存器。

- **当 `threadsPerBlock = 128` (效果最好) 时:** 一个块需 $128 \times R$ 个寄存器。若 $R = 40$，则需 5120 个寄存器。一个 SM 可同时容纳 $\lfloor 65536/5120 \rfloor = 12$ 个块。这 12 个块提供了 $12 \times (128/32) = 48$ 个活跃 Warp 供 SM 调度，占用率较高，能有效隐藏延迟。
- **当 `threadsPerBlock = 1024` (效果最差) 时:** 一个块需 $1024 \times R$ 个寄存器。同样假设 $R = 40$，则需 40960 个寄存器。此时，一个 SM **只能容纳** $\lfloor 65536/40960 \rfloor = 1$ **个块**，因为启动第二个块会导致寄存器需求（81920）超出 SM 物理上限。该 SM 上仅有 $1 \times (1024/32) = 32$ 个活跃 Warp。

低占用率导致性能下降 对比两种情况：128 线程/块配置下的 SM 拥有 48 个 Warp 可供调度，而 1024 线程/块配置下仅有 32 个。Warp 池规模的缩小，降低了 SM 在某个 Warp 暂停时找到其他可执行 Warp 的概率，导致计算单元闲置，无法有效隐藏延迟，最终致使整体性能大幅下降。

实验结果完美地印证了这一理论：从 128 增加到 1024，虽然理论上并行线程更多，但由于资源限制（很可能是寄存器）导致占用率降低，反而严重损害了性能。

2.2.4 统一参数后的验证性测试

为了确保实验的严谨性，后续进行了一组验证性测试，在该测试中，所有 CUDA 内核（包括点值乘法内核）的 'threadsPerBlock' 参数均被统一设置并同步调整。这组更严谨的测试结果证实了初步的结论，并提供了更精确的性能画像。性能最佳点从初步测试的 128 转移到了 `threadsPerBlock = 256`，这表明点值乘法内核的性能特征对整体最优配置有一定影响。同时，在 `1024` 时性能同样出现显著下降，这进一步说明，存在一个最优的块大小，过大或过小的配置都会因无法高效利用硬件资源而损害性能。(具体数据因实验篇幅这里省略)。

2.2.5 GPU 加速核心代码

Listing 1: 核心 CUDA C++ 代码

```cpp
// GPU 内核: 位反转置换， 将输入序列按照蝶形运算的要求进行重新排序
__global__ void bit_reverse_kernel(int* a, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= n) return;
    int logn = 0;
    if (n > 1) logn = __log2f(n);
    int rev_i = 0;
    for (int j = 0; j < logn; j++) {
        if ((i >> j) & 1) {
            rev_i |= 1 << (logn - 1 - j);
        }
    }
    if (i < rev_i) { // 防止重复交换
        int temp = a[i];
        a[i] = a[rev_i];
        a[rev_i] = temp;
```

NKU

```
    }
}
// GPU 内核: NTT 蝶形运算， 执行 NTT 的一个阶段，m 代表当前阶段的子问题大小
__global__ void ntt_stage_kernel(int* a, int n, int p, int m, bool is_inverse) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid >= n / 2) return;
    int g = 3; // 本原根，计算单位根 w_m
    long long wm_base = power_gpu(g, (p - 1) / m, p);
    if (is_inverse) { // 逆变换使用单位根的逆
        wm_base = power_gpu(wm_base, p - 2, p);
    }
    int j = tid % (m / 2);
    int k = (tid / (m / 2)) * m;
    int idx1 = k + j;
    int idx2 = idx1 + m / 2; // 蝶形运算: a[idx1]=u+t, a[idx2]=u-t
    long long w = power_gpu(wm_base, j, p);
    long long t = (w * a[idx2]) % p;
    long long u = a[idx1];
    a[idx1] = (u + t) % p;
    a[idx2] = (u - t + p) % p;
}
// GPU 内核: 点值乘法，将两个多项式在点值表示下进行相乘
__global__ void pointwise_mult_kernel(int* a, int* b, int* ab, int n, int p) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        ab[i] = (1LL * a[i] * b[i]) % p;
    }}
```

2.3 使用 Montgomery 模乘，Barrett 模乘 +GPU 加速

我们将传统实现与两种优化算法的性能延迟及加速比汇总在下表中。数据显示，Barrett 和 Montgomery 模乘都带来了显著的性能提升，其中 Barrett 算法表现最优，平均加速比可达 **1.83x**。具体的代码因为篇幅原因全部放在 github 里了，这里不再进行细说。

表 3: 不同模乘算法在 $n = 131072$ 时的性能延迟 (ms) 及加速比对比

素数模数 (p)	传统 Cooley-Tukey	Montgomery 模乘		Barrett 模乘	
	延迟 (ms)	延迟 (ms)	加速比	延迟 (ms)	加速比
$p = 7340033$	14.79	8.34	1.77x	**8.26**	**1.79x**
$p = 104857601$	14.20	8.07	1.76x	**7.78**	**1.83x**
$p = 469762049$	14.81	8.22	1.80x	**7.91**	**1.87x**

2.3.1 为什么 Barrett 模乘的效果更好

尽管 Barrett 和 Montgomery 模乘都通过将高成本的“取模”指令替换为更快的乘法和位移操作来优化计算，但 Barrett 算法在 NTT 场景下通常表现更优，其核心原因在于**避免了数据域的转换开销**。

Montgomery 算法要求所有操作数都必须首先转换到“Montgomery 域”中，运算结束后再转换回来。虽然对于固定常数（如 NTT 中的旋转因子）的预转换开销可以接受，但对于每次蝶形运算中不断变化的输入数据，这种反复的域转换会累积成不可忽视的性能负担。

相比之下，Barrett 算法直接对原始数据进行操作，无需任何域转换。它的计算流程更为直接，主要由乘法、位移和极少数的修正减法构成。这一系列操作能够非常高效地映射到现代 GPU 的计算流水线上，从而避免了 Montgomery 算法的额外开销，获得了微弱但稳定的性能优势。

2.4 加速代码优化

为了最大化 GPU 的计算潜力，我们在基础实现之上进行了三项关键的性能优化。

2.4.1 优化一：预计算蝶形因子

优化原理与实现 为消除内核中 ‘power_gpu‘ 函数实时计算蝶形因子的昂贵开销，我们采用了预计算策略。在计算开始前，所有蝶形因子在 CPU 端一次性生成并存入一个查找表，随后该表被拷贝至 GPU 全局内存。内核执行时，线程通过高效的内存查找替代了重复的实时计算，显著降低了计算延迟。

Listing 2: 优化后的蝶形运算内核

```
// 优化后的内核：从预计算表中读取蝶形因子
__global__ void ntt_stage_kernel_optimized(int* a, const int* wn, int n, int m, int
    stage_offset, int p) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid >= n / 2) return;
    int j = tid % (m / 2);
    // ... 直接从全局内存读取预计算好的蝶形因子
    long long w = wn[stage_offset + j];
    // ...}
```

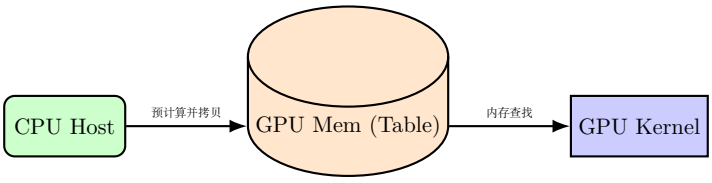

图 2.4: 蝶形因子计算优化示意图

2.4.2 优化二：移除中间同步与异步执行

优化原理与实现 为减少内核启动间的等待开销，我们移除了 ‘ntt_gpu‘ 函数中每个内核后的阻塞式同步调用 ‘cudaDeviceSynchronize()‘。优化后，所有内核被异步地提交至 CUDA 工作队列，仅在整个

计算流程末尾进行一次总同步。这使得 GPU 能够接收到一个连续的任务流并无间断执行，最大化了 CPU 与 GPU 的并行度，提升了整体吞吐量。

Listing 3: 移除中间同步调用

```
// 优化前的循环
for (int m = 2; m <= n; m <<= 1) {
    ntt_stage_kernel<<<...>>>(...);
    CUDA_CHECK(cudaDeviceSynchronize()); // 阻塞式同步，效率低
}
// 优化后的循环（在 poly_multiply 中调用）
for (int m = 2; m <= n; m <<= 1) {
    ntt_stage_kernel_optimized<<<...>>>(...); // 非阻塞提交
}
// 仅在顶层函数末尾同步一次
```

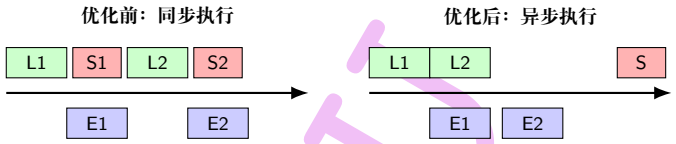

图 2.5: 同步与异步执行流程对比 (L: Launch, S: Sync, E: Execute)

2.4.3 优化三：使用共享内存与混合策略

优化原理与实现 为进一步减少访存延迟，我们引入了共享内存优化。共享内存是位于 SM 芯片上的高速缓存，其访问速度远快于全局内存。对于规模较小的 NTT 阶段，我们设计了专用的 'ntt_stage_kernel_shared' 内核。该内核首先由块内所有线程协作，将数据从全局内存一次性加载到共享内存中，然后在共享内存内完成所有蝶形运算，最后再将结果写回全局内存。这种方法将大量零散、高延迟的全局内存访问，聚合为少量的批量读写和大量高速的片上内存访问。对于无法使用共享内存的大规模阶段，则回退使用原有的全局内存内核。这种混合策略确保了在不同计算阶段都能选用最高效的执行路径。

Listing 4: 使用共享内存的蝶形运算内核

```
__global__ void ntt_stage_kernel_shared(int* a, const int* wn_global, int m, int
    stage_offset, int p) {
    extern __shared__ int s_mem[];
    int* s_a = s_mem;
    int* s_w = &s_mem[m];
    int tid = threadIdx.x;
    int k = blockIdx.x * m; // 1. 协作从全局内存加载到共享内存
    s_a[tid] = a[k + tid];
    s_a[tid + m/2] = a[k + tid + m/2];
    __syncthreads(); // 确保所有数据加载完毕
    // 2. 在共享内存中进行计算
```

NKU

```
    // ... 3. 将结果从共享内存写回全局内存
    a[k + tid] = s_a[tid];
    a[k + tid + m/2] = s_a[tid + m/2];
}
```

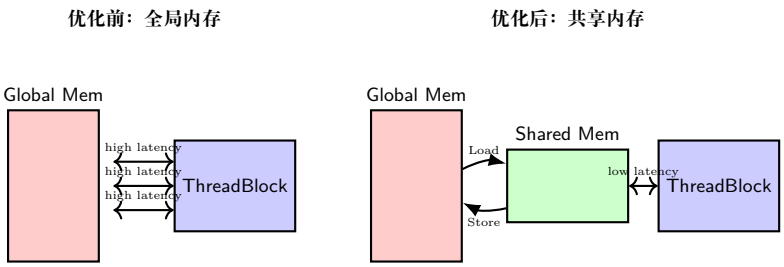

图 2.6: 共享内存优化示意图

表 4: 深度优化策略与 Barrett 模乘性能对比

素数模数 (p)	Barrett 模乘	深度优化后	
	延迟 (ms)	延迟 (ms)	进一步加速比
$p = 7340033$	8.26	**7.06**	**1.17x**
$p = 104857601$	7.78	**6.54**	**1.19x**
$p = 469762049$	7.91	**6.65**	**1.19x**

如表 4 所示，通过内存访问优化和执行策略调整，我们在 Barrett 模乘的基础上，再次获得了约 **1.18x** 的平均性能提升，将算法的计算效率推向了新的高度，代码同样见 github 仓库。

2.5 不同显卡下运行相同代码的对比

为了探究硬件对算法性能的影响，我们在三款不同定位的 NVIDIA 显卡上运行了相同的 Barrett 模乘加速代码。

2.5.1 测试平台信息

表 5: GPU 测试平台硬件规格

类型	GPU 信息	主机规格
Tesla T4	1 卡 * 16GB GDDR6	4 核 CPU, 16GB 内存
RTX 3090	1 卡 * 24GB GDDR6X	10 核 CPU, 32GB 内存
RTX 4090	1 卡 * 24GB GDDR6X	10 核 CPU, 32GB 内存

2.5.2 性能结果与分析

从表 6 的结果来看，一个显著的现象是：尽管三款 GPU 的理论算力和市场定位差异巨大，但它们在本次 NTT 计算中的性能表现却**非常接近**。这有力地表明，对于大点数的 NTT 计算，其瓶颈并非

表 6: 不同 GPU 上 Barrett 模乘 NTT 的性能延迟 (ms) @ $n = 131072$

素数模数 (p)	Tesla T4	RTX 3090	RTX 4090
$p = 7340033$	**8.26**	8.82	8.75
$p = 104857601$	**7.78**	8.54	8.07
$p = 469762049$	**7.91**	8.53	8.08

在于计算单元的浮点或整数运算能力，而是在于**内存带宽**。NTT 算法涉及大量的全局数据重排，这使得 GPU 的大部分时间都在等待数据在显存和计算核心之间的传输，从而掩盖了原始计算能力的差异。

尽管如此，我们仍能观察到细微的性能差别：

- **RTX 4090 vs. RTX 3090**: 4090 全面微弱领先于 3090。这主要归功于其更先进的架构、更大的 L2 缓存和更高的显存带宽，这些优势缓解了内存瓶颈，使其在数据访问密集型任务中表现更佳。
- **Tesla T4 的意外表现**: 作为一款数据中心卡，T4 在本测试中的表现甚至优于两款高端消费级显卡。这可能是因为 T4 的架构和驱动程序专为持续、高吞吐的计算任务优化。其内存子系统和调度器可能更适应 NTT 这种具有固定访存模式的算法，从而实现了更高的实际内存效率。

3 profiling

本报告使用 NVIDIA 的 **nvprof** 工具分析`./test` 性能。nvprof 是一个命令行工具，用于检测 CUDA 程序性能瓶颈，优化 GPU 资源。测试通过 `apt install` 安装后，运行 `nvprof ./test` 完成。

表 7: Barrett 模乘关键性能指标

类别	名称	时间占比	总时间	调用次数	平均时间
GPU 活动	ntt_stage_kernel_barrett	96.23%	263.89ms	171	1.5432ms
	normalize_kernel_barrett	2.96%	8.1208ms	4	2.0302ms
	CUDA memcpy HtoD	0.19%	528.22us	8	66.027us
	CUDA memcpy DtoH	0.09%	248.22us	4	62.055us
API 调用	cudaMalloc	55.53%	351.73ms	8	43.966ms
	cudaDeviceSynchronize	43.32%	274.39ms	195	1.4072ms

3.1 结果分析

表格显示：

- **GPU 活动**: `ntt_stage_kernel_barrett` 占 96.23%，是主要瓶颈。
- **API 调用**: `cudaMalloc`(55.53%) 和 `cudaDeviceSynchronize`(43.32%) 开销大，内存分配与同步需优化。

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.