Lab3 实验报告

1 项目结构

```
lab10/
├── src/
│   ├── gemm.cu           # 主程序文件，包含矩阵乘法的CUDA实现
│   ├── gemm_naive.cu     # 朴素实现版本
│   ├── gemm_shared.cu    # 使用共享内存优化版本
│   ├── gemm_tiled.cu     # 使用分块优化版本
│   └── utils.h           # 工具函数（矩阵初始化、验证等）
├── results/
│   └── performance.csv   # 性能测试结果
└── README.md             # 实验说明和结果分析
```

2 实验背景

本实验旨在通过不同的优化策略提高CUDA矩阵乘法的性能。我们实现了朴素版本、共享内存优化版本和分块优化版本，并对比了它们在不同矩阵大小和线程块大小下的性能表现。

3 实验方法

起初拿到这个实验的时候，说实话我是有点发怵的。CUDA 对我而言一直像是一块“黑魔法”领域——性能强大但晦涩难懂。但也正是因为这种神秘感，我暗暗下定决心，哪怕这次绕点远路，也要摸清楚它到底是怎么一回事。

实验第一步是实现最基础的 CUDA 版本矩阵乘法，也就是所谓的“朴素实现”。我们要做的事情其实不复杂：每个线程计算结果矩阵中的一个元素，通过遍历对应行和列实现乘法累加。听起来不难，但当我真正开始写代码的时候，才意识到 CUDA 和普通 C 不一样的地方太多了，线程的调度、块的组织、索引的计算……每一步都让我怀疑自己是不是“打开方式”不对。

在不断阅读资料、参考 slides 和教材的过程中，我逐渐理解了 CUDA 的核心设计哲学：将大量的小任务交给成千上万的线程去并行完成。于是，我开始尝试自己写 kernel 函数，并小心翼翼地编译、运行，再用 CPU 实现的结果来比对正确性。

调试的过程一开始异常痛苦。最初版本的代码，不是 kernel 不工作就是矩阵尺寸搞错了，有一次我甚至直接访问越界导致 GPU 程序直接崩溃。那种“黑盒调试”的感觉非常挫败。但也正是在这些错误中，我开始慢慢理出 CUDA 的内存分配和线程索引之间的逻辑关系。

当第一个朴素实现终于跑通，并且正确输出结果的时候，我真的感到一种踏实的成就感。虽然这个版本性能并不理想，但它就像是我和 CUDA 之间建立起的第一座桥梁。从这里出发，我才更有信心进入下一步的优化世界。

为了更好地理解代码的行为，我特地在每一部分实现中都加入了详细的注释和 print 输出。其中，朴素实现的核心 kernel 函数大致如下：

```cpp
__global__ void gemm_naive(float* A, float* B, float* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; ++k) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

这段代码的逻辑非常直观，每个线程根据其全局位置 `(row, col)` 来对应结果矩阵 `C` 中的某一个元素，然后通过遍历矩阵 `A` 的一行与矩阵 `B` 的一列来进行乘加累积，最终得到该位置的值。

值得注意的是：我们需要检查 `row` 和 `col` 是否越界，否则在较小的矩阵尺寸或线程块尺寸下可能会访问非法内存，导致程序崩溃。

在进一步的优化版本（如 `gemm_shared.cu`）中，我们引入了 `__shared__` 变量来存放每次 block 内需要复用的数据块，极大地减少了全局内存访问次数，示例结构如下：

```cpp
__shared__ float Asub[TILE_SIZE][TILE_SIZE];
__shared__ float Bsub[TILE_SIZE][TILE_SIZE];
```

我们每个 block 分别加载矩阵 `A` 和 `B` 的一小块进来，然后通过线程协作完成该子块的计算。这种方式让 GPU 的带宽得到了更充分的利用，也显著提升了性能。

在写这些优化代码的时候，我第一次真正感受到“高性能计算”并不仅仅是让程序能跑起来，而是在算力与资源的约束下尽可能高效地跑好。每一个线程访问哪块内存、共享内存何时同步、如何避免 bank conflict……这些都成了我必须面对的问题。

2. **优化实现**：
    - 使用共享内存优化：减少全局内存访问。
    - 分块计算优化：将矩阵分成小块进行计算，提高缓存命中率。
    - 结合多种优化技术的版本。
3. **性能测试**：
    - 测试不同线程块大小 (8x8, 16x16, 32x32)。
    - 测试不同矩阵规模 (128x128, 512x512, 1024x1024, 2048x2048)。
    - 测试不同访存模式和优化策略。
    - 记录并比较各种实现的执行时间。

4 实验结果

实验结果记录在 `results/performance.csv` 文件中，并通过 `analysis.py` 脚本生成性能对比图表 `performance_plots.png`。图表展示了不同实现方式在不同矩阵大小和线程块大小下的执行时间和加速比。

5 结果分析

在实验结果中，我们观察到以下几点：

1. **朴素实现 (naive)**：
    - 在较小的矩阵大小（如128x128）下，执行时间显著较长，尤其是在较小的线程块大小（8x8）时。
    - 随着线程块大小的增加，执行时间显著减少，表明较大的线程块有助于提高性能。
    - 在较大的矩阵（如2048x2048）下，执行时间显著增加，并且在8x8线程块大小下未通过验证。
2. **共享内存优化 (shared)**：
    - 在所有测试条件下，执行时间普遍低于朴素实现，表明共享内存优化在减少执行时间方面非常有效。
    - 在较小的矩阵大小下，线程块大小对性能的影响较小，但在较大的矩阵下，较大的线程块（如32x32）提供了更好的性能。
    - 在2048x2048矩阵大小下，8x8线程块大小未通过验证，可能需要进一步优化。
3. **验证结果**：
    - 大多数情况下，结果通过验证，表明实现的正确性。
    - 在某些情况下（如2048x2048矩阵的8x8线程块），结果未通过验证，可能需要检查实现的稳定性和准确性。

总体而言，实验结果表明，使用共享内存优化和较大的线程块可以显著提高CUDA矩阵乘法的性能。建议在实际应用中优先考虑这些优化策略。

6 结论

本实验验证了通过优化内存访问模式和计算策略可以显著提高CUDA矩阵乘法的性能。共享内存和分块计算是有效的优化策略，建议在实际应用中结合使用以获得最佳性能。实验结果为CUDA编程提供了有价值的参考。