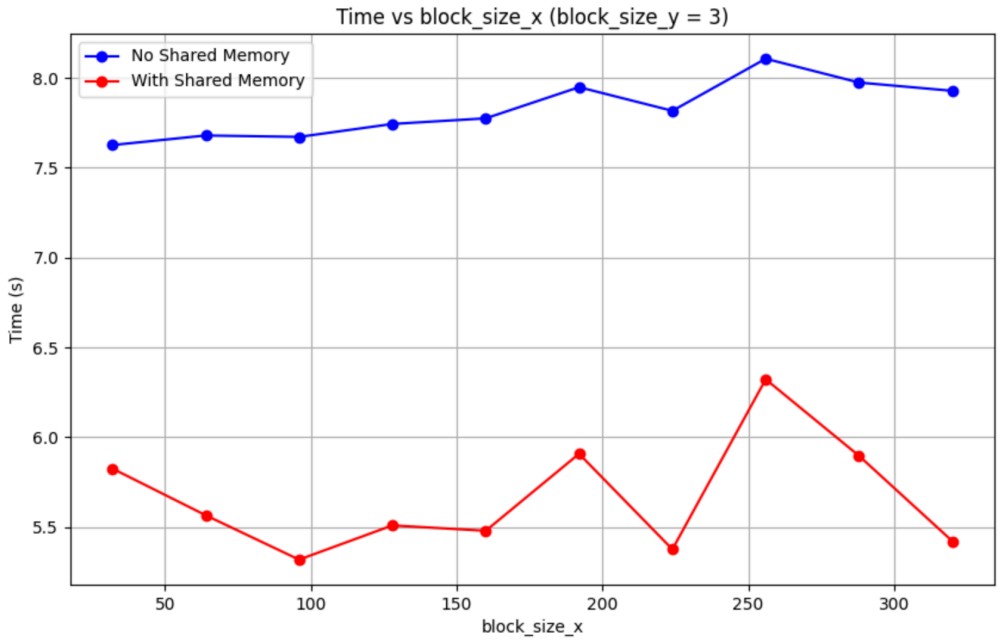
实验四报告

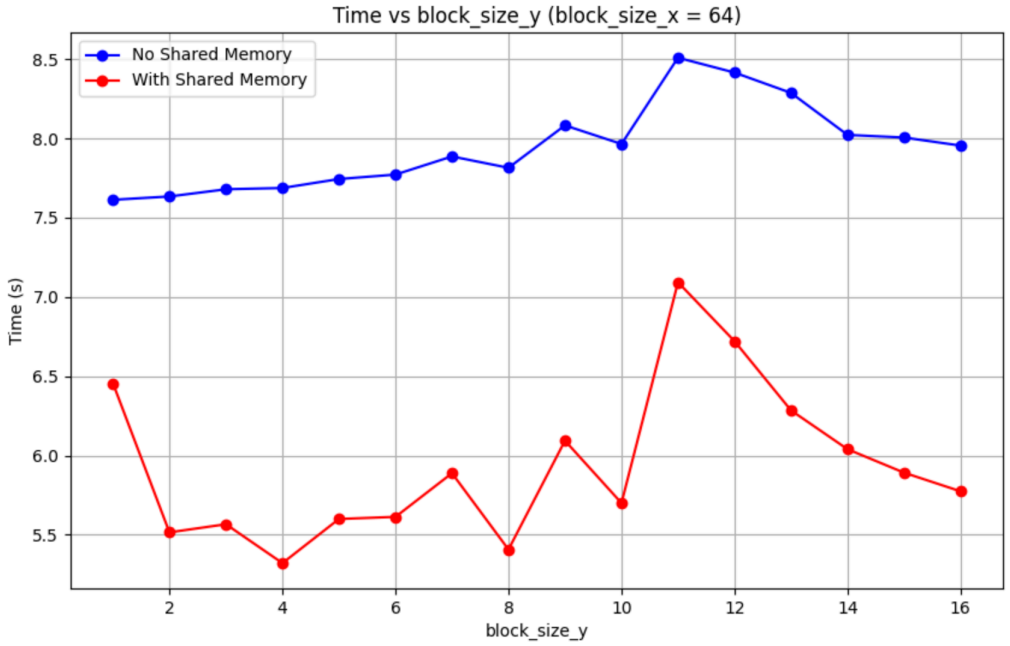
宋建昊 2022010853

实验结果

下面为 block_size_x=64时，耗时time与 block_size_y的关系



下面为 block_size_y=3时，耗时time与 block_size_x的关系



影响因素分析

- block_size较小时程序执行的效率较高，且在不选择shared memory时更明显。
- 随着 block_size变化，程序性能会出现波动，呈现出剧烈波动，结果不稳定，在使用 shared memory的情况下波动更明显。

- 整体来看，使用shared memory之后程序用时会显著缩短。

综合分析

1. **block_size 较小时程序执行效率较高，且在不选择 shared memory 时更明显**

- **线程块大小（block_size）与硬件资源的匹配**： 在 CUDA 中，线程块（block）是调度和执行的基本单位。每个 SM（流多处理器）能同时处理有限数量的线程（受限于寄存器、共享内存和线程束 warp 的数量）。当 block_size 较小时（例如 32 或 64），每个线程块内的线程数较少，SM 可以同时调度更多的线程块（即更高的 occupancy，占用率），从而充分利用 GPU 的并行计算能力。
    - 不使用 shared memory 时，程序依赖全局内存（global memory）。较小的 block_size 意味着每个线程块的内存访问需求较小，可能更容易被缓存（L1/L2 cache）命中，减少全局内存访问的延迟。
    - 使用 shared memory 时，虽然可以减少全局内存访问，但小的 block_size 可能无法充分利用 shared memory 的容量，导致内存带宽优势不明显。
- **线程同步开销**： 不使用 shared memory 时，线程间通常没有显式的同步需求（例如 __syncthreads()），程序执行更接近"独立线程"的模式，效率较高。而使用 shared memory 时，线程块内的线程需要通过同步操作协作访问共享内存，较小的 block_size 会增加同步的相对开销（因为线程数少，同步的收益不足以抵消开销）。
- **warp 执行效率**： CUDA 中，线程以 warp（32 个线程）为单位执行。如果 block_size 是 32 的倍数（例如 32 或 64），warp 可以被完全填满，避免线程浪费（divergence 或 inactive threads）。你的数据中 block_size_x=32 或 64 时效率较高，可能与此相关。

---

2. **随着 block_size 变化，程序性能出现波动，呈现剧烈波动，结果不稳定，使用 shared memory 时更明显**

- **资源竞争与线程调度**： 随着 block_size_x 或 block_size_y 增大，线程块内的线程数增加，可能导致以下问题：
    - **寄存器和共享内存限制**：每个 SM 的寄存器和共享内存是有限的。当 block_size 过大时，单个线程块占用过多资源，可能减少并发的线程块数量（occupancy 下降），性能出现波动。
    - **内存带宽瓶颈**：较大的 block_size 会增加对全局内存或共享内存的访问需求。如果访问模式不优（例如非合并访问），会导致带宽利用率下降，性能不稳定。
- **使用 shared memory 时的波动更明显**：
    - **共享内存的容量限制**：每个 SM 的共享内存大小有限（例如 48KB 或 96KB，取决于 GPU 架构）。当 block_size 增大时，共享内存需求可能超过限制，导致编译器隐式减少并发的线程块数，或者程序运行时出现动态调整，引发性能剧烈波动。
    - **同步开销放大**：使用 shared memory 通常需要线程块内同步（__syncthreads()）。较大的 block_size 意味着更多线程参与同步，同步延迟随线程数增加而放大，性能波动更明显。
    - **内存访问冲突（Bank Conflict）**：如果你的代码中共享内存访问模式不佳（例如多个线程同时访问同一 bank），随着 block_size 变化，bank conflict 的影响会更显著，导致性能不稳定。
- **硬件调度特性**： GPU 的调度器在分配线程块时会受到 block_size 的影响。较大的 block_size 可能导致某些 SM 被过度占用，而其他 SM 空闲（负载不均衡），从而引起性能波动。

---

3. **整体来看，使用 shared memory 之后程序用时显著缩短**

- **减少全局内存访问延迟**： CUDA 中全局内存的访问延迟较高（几百个周期），而共享内存的延迟极低（接近寄存器级别）。通过将频繁访问的数据加载到共享内存中，可以显著减少全局内存的访问次数，从而提升性能。使用 shared memory 后时间普遍下降，说明共享内存有效减少了内存访问瓶颈。
- **线程协作优化**： 共享内存允许线程块内的线程共享数据，避免重复的全局内存读取。如果你的算法涉及线程间数据共享（例如矩阵计算、卷积等），使用 shared memory 可以大幅提升效率。
- **带宽利用率提升**： 共享内存的带宽远高于全局内存。合理使用 shared memory 可以将分散的全局内存访问转化为集中高效的共享内存访问，尤其在合并访问（coalesced access）的情况下效果更明显。

总结

- 如何设置 thread block size 才可以达到最好的效果？为什么？
- **最佳 thread block size**： 从你的数据来看，block_size_x 和 block_size_y 的组合对性能影响显著。综合分析：
  - 当 block_size_x=64, block_size_y=4 时，使用 shared memory 的时间为 **5.32051s**，是所有组合中最低的。
  - 未使用 shared memory 时，block_size_x=32, block_size_y=2 的时间为 **7.61406s**，表现较优。
  - 因此：
    - 如果使用 shared memory，推荐 block_size_x=64, block_size_y=4。
    - 如果不使用 shared memory，推荐 block_size_x=32, block_size_y=2。
- Shared memory 总是带来优化吗？如果不是，为什么？
- shared memory 不总是带来优化，block_size_x=32, block_size_y=1 时，未使用 shared memory 的时间为 **9.28377s**，而使用 shared memory 为 **10.1254s**，性能反而下降。
- Shared memory 在什么 thread block size 下有效果，什么时候没有？
- **有效情况**：
  - **中等 thread block size（如 128~256）**： 如 block_size_x=64, block_size_y=4（256 线程），shared memory 显著减少时间（从 7.68743s 到 5.32051s）。此时线程数足够多，共享内存能充分发挥数据共享和减少全局内存访问的优势。
  - **数据重用率高**：当线程块内需要多次访问同一数据（如矩阵计算中的 tile 操作），shared memory 效果更明显。
- **无效情况**：
  - **小的 thread block size（如 32）**： 如 block_size_x=32, block_size_y=1，时间反而增加（9.28377s -> 10.1254s）。线程数少时，共享内存的加载/同步开销占比高，收益不足以抵消。
  - **大的 thread block size（如 1024）**： 如 block_size_x=1024, block_size_y=1，未使用 shared memory 为 7.84245s，使用为 5.75355s，虽然有优化，但效果不如中等大小显著。原因是线程过多可能导致共享内存超限或 occupancy 下降。
- 还有哪些可以优化的地方？
- **内存访问合并（Coalesced Access）**： 确保全局内存访问是合并的（连续线程访问连续地址），减少带宽浪费。可以用 nvprof 检查是否存在非合并访问。

- **共享内存 Bank Conflict**： 检查 shared memory 访问是否引发 bank conflict（多个线程访问同一 bank），可以通过填充（padding）或调整数据布局优化。
- **寄存器使用**： 减少每个线程的寄存器占用（通过减少局部变量或编译选项 -maxrregcount），提高 occupancy。
- **指令级优化**： 使用快速数学函数（如 __fmul）或内联函数，减少指令开销。
- **异步操作**： 如果程序涉及大量数据传输，考虑使用 CUDA 流（Streams）实现计算与内存拷贝的 overlap。
- 应该如何设置 thread block size?

1. **计算 Occupancy**：
    - 使用 NVIDIA 的 Occupancy Calculator 或 API（如 cudaOccupancyMaxActiveBlocksPerMultiprocessor）计算不同 block_size 下的占用率。
    - 选择能最大化 occupancy 的值（通常 128、256、512）。
2. **考虑 Warp 大小**：
    - 确保 block_size 是 32 的倍数，避免 warp 内线程浪费。
3. **实验验证**：
    - 测试几个典型值（如 64、128、256、512），结合性能数据（如时间、吞吐量）选择最佳值。

- **原则**：
    - **计算密集型任务**：选择较大的 block_size（如 256~512），充分利用并行性。
    - **内存密集型任务**：选择中等 block_size（如 128~256），平衡内存访问和计算。
    - **动态调整**：根据问题规模（grid size）调整，确保线程总数覆盖所有数据。
- 应该如何决定 shared memory 的使用?
- **决策依据**：
    1. **数据重用率**：
        - 如果线程块内多次访问相同全局内存数据（例如矩阵乘法的 tile、卷积的窗口），使用 shared memory。
        - 如果每个线程独立访问全局内存且无重用（如流处理），无需使用。
    2. **全局内存访问模式**：
        - 非合并访问或随机访问时，shared memory 可优化为局部连续访问。
        - 已合并且缓存命中率高时（如使用 L2 缓存），shared memory 收益不大。
    3. **共享内存需求量**：
        - 计算每个线程块的共享内存需求，确保不超过 SM 限制（例如 48KB 或 96KB）。
        - 如果需求过大导致 occupancy 下降，考虑不使用或分块处理。
    4. **同步开销**：
        - 如果线程间协作频繁且同步成本可接受，使用 shared memory。
        - 如果任务简单或同步开销过高，避免使用。