|  |
| --- |
| **并行计算**  **2025年春季学期** |
| **期末报告** |
|  |
| 2025年 4月 17日 |

目录

[**摘要** 2](#_Toc195831576)

[**Highlights** 2](#_Toc195831577)

[**1.** **自选命题的背景和意义** 2](#_Toc195831578)

[**2.** **国内外研究现状** 3](#_Toc195831579)

[**3.** **串行算法设计与实现** 3](#_Toc195831580)

[**4.** **并行算法设计与实现** 4](#_Toc195831581)

[**4.1OpenMP并行** 4](#_Toc195831582)

[**4.1MPI并行** 6](#_Toc195831583)

[**4.3CUDA并行** 8](#_Toc195831584)

[**4.4OpenMP+MPI混合并行** 9](#_Toc195831585)

[**5.** **实验设置** 12](#_Toc195831586)

[**5.1 实验数据** 12](#_Toc195831587)

[**5.2 实验软硬件环境** 12](#_Toc195831588)

[**5.3 评价指标** 13](#_Toc195831589)

[**6.** **实验结果与分析** 13](#_Toc195831590)

[**6.1程序运行截图** 13](#_Toc195831591)

[**6.2实验数据分析** 16](#_Toc195831592)

[**OpenMP并行性能分析** 16](#_Toc195831593)

[**MPI并行分析** 17](#_Toc195831594)

[**CUDA并行分析** 18](#_Toc195831595)

[**OpenMP+MPI混合并行性能分析** 19](#_Toc195831596)

[**7.** **总结与讨论** 20](#_Toc195831597)

[**参考文献** 21](#_Toc195831598)

[**附件一：性能优化过程记录表（必备）** 22](#_Toc195831599)

[**附件二：关键源码说明（必备）** 23](#_Toc195831600)

[C++串行代码 23](#_Toc195831601)

[OpenMP并行代码 26](#_Toc195831602)

[MPI并行代码 29](#_Toc195831603)

[CUDA并行代码 32](#_Toc195831604)

[OpenMP+MPI混合并行代码 37](#_Toc195831605)

**摘要**

本报告围绕“图像灰度批量处理”的高性能并行化方案展开研究，选题旨在解决面对大规模图像数据时传统串行处理效率低下的问题。设计过程中，分别采用了四种主流并行计算模型：基于OpenMP的多线程并行、基于MPI的并行、基于CUDA的GPU加速并行、以及MPI与OpenMP结合的混合并行模式，实现了图像灰度处理任务在不同计算架构下的高效执行。本项目的创新点在于综合比较了多种并行策略在不同硬件资源上的性能表现，特别是通过混合并行方式，在保证计算正确性的基础上显著提升了整体处理速度。实验结果显示：OpenMP在当前任务环境下对性能的提升较弱；CUDA在GPU上实现了更优的核内并行加速；而MPI+OpenMP混合并行则在真实壁钟时间下表现出最优的总体性能，极大优化了用户实际等待时间。通过本项目验证了并行技术在图像处理中的实际应用效果，具有一定的参考价值和推广意义。

**Highlights**

多种并行模型实现与对比：本项目系统地实现并对比了OpenMP并行、MPI并行、CUDA并行及MPI+OpenMP混合并行四种方式，全面评估各模型在图像灰度批量处理中的性能优势与适用场景。

计时方式细致区分：分析中区分了CPU/GPU计算时间与Wall-clock时间，突出了MPI混合并行在用户真实感知性能上的显著优势，为实际部署提供合理依据。

混合并行性能最优：通过MPI与OpenMP协同利用多节点与多核资源，显著缩短处理总耗时，实现了性能与资源利用率的双提升。

实验数据详实、图表可视化清晰：提供了加速比、效率对比图等多维度实验结果，清晰展示了各方案在不同并行度下的扩展性与瓶颈。

具有现实应用推广价值：该项目适用于需要批量图像预处理的工业/科研场景（如医学影像、安防监控等），并具有较强的移植性和可扩展性。

1. **自选命题的背景和意义**

图像灰度批量处理是图像处理领域的基础性任务之一，广泛应用于医学图像分析、安防监控、遥感测绘、工业检测等场景。在现代信息社会中，随着图像采集设备的普及与分辨率的提升，单次处理的图像数量和尺寸呈指数级增长，传统串行图像处理方式在性能上已难以满足实时性和大规模任务的要求。因此，探索基于高性能计算的并行化图像灰度处理方法，不仅能够显著提升处理效率，更具备重要的实践意义。通过对 OpenMP、CUDA 和 MPI 等并行模型的综合应用，可以实现从多核 CPU 到 GPU 再到分布式集群的高效资源调度和负载均衡，推动图像处理从单机走向集群化、智能化，为人工智能、图像识别、大数据分析等领域提供坚实的技术基础。

1. **国内外研究现状**

在国际上，图像处理的并行化研究起步较早，NVIDIA 提出的 CUDA 架构极大推动了 GPU 在图像处理中的应用，国外诸如 OpenCV GPU 模块、TensorFlow 图像处理算子等已广泛集成高性能并行计算模型。在分布式方面，如谷歌的 MapReduce、Apache Spark 等框架也被用于大规模图像数据的分布式预处理与分析。国内在近年来也逐渐加大了在高性能计算与图像处理结合方向的研究力度，清华大学、中科院等单位在异构并行处理、GPU 加速图像处理算法、深度学习图像预处理优化等方面取得了显著成果。当前，随着国产高性能计算硬件的发展和人工智能应用的兴起，基于多种并行模型融合的图像处理系统正成为研究热点，未来发展趋势将聚焦于自适应调度、异构融合、自动并行优化等方向，以应对更复杂的应用需求和更庞大的数据规模。

1. **串行算法设计与实现**

**实现思路**

代码的串行化实现基于顺序处理原则，主要分为以下步骤：

1. **图像读取**：使用OpenCV库逐张读取输入图像。
2. **灰度转换**：通过双重循环遍历每个像素，按公式 **Gray = 0.299\*R + 0.587\*G + 0.114\*B** 计算灰度值。
3. **结果保存**：将灰度图写入磁盘，记录处理时间。

所有操作均按顺序执行，无并行优化。

**rgb2graySerial 的伪代码：**

Function rgb2graySerial(in, out, height, width):

for i = 0 to height-1 do

for j = 0 to width-1 do

idx = i \* width + j

R = in[idx \* 3]

G = in[idx \* 3 + 1]

B = in[idx \* 3 + 2]

out[idx] = 0.299\*R + 0.587\*G + 0.114\*B

end for

end for

end Function

**流程图**

开始

├─ 读取图片列表

├─ 逐张处理图片:

│ ├─ 读取原始图像 (imread)

│ ├─ 灰度转换 (双重循环逐像素计算)

│ └─ 保存灰度图 (imwrite)

└─ 统计总时间

结束

**性能瓶颈分析**

**计算瓶颈:** 逐像素串行计算，无法利用多核CPU 。

**复杂度分析：**单张图像处理时间为 O(H×W)，其中 H为高度，W为宽度。  
N张图像总时间为 O(N×H×W)，随数据量线性增长。

**关键耗时操作：**

浮点运算：每个像素需3次乘法、2次加法。

内存访问：每次循环需访问3个输入通道和1个输出通道。

1. **并行算法设计与实现**

**4.1OpenMP并行**

**主程序流程：**

开始

├─ 读取输入目录下所有图片路径

├─ 用户指定并行线程数

├─ 对每张图片循环处理:

│ ├─ 读取原始图像 (imread)

│ ├─ OpenMP并行灰度转换

│ └─ 保存灰度图 (imwrite)

└─ 统计总执行时间并输出

结束

**伪代码流程：**

主程序:

初始化文件列表

设置OpenMP线程数

for 每张图片 in 列表:

srcImg = 读取图像数据

创建grayImg缓冲区

start = 计时开始

#pragma omp parallel for collapse(2)

for i from 0 to height-1:

for j from 0 to width-1:

rgb = srcImg[i][j]

gray = 0.299\*r + 0.587\*g + 0.114\*b

grayImg[i][j] = gray

end\_time = 计时结束

保存grayImg

累加执行时间

输出总耗时

**并行化核心设计：**

**数据并行模式**：将图像划分为二维网格，各线程处理不同像素块

**任务分配策略**：OpenMP默认**static**调度，均匀分配行-列组合迭代

**并行域控制**：通过**#pragma omp parallel for collapse(2)**实现嵌套循环并行

**关键代码段：**

// 并行灰度转换函数

void rgb2grayOpenMP(unsigned char\* d\_in, unsigned char\* d\_out, int h, int w) {

#pragma omp parallel for collapse(2)

for (int i=0; i<h; ++i) {

for (int j=0; j<w; ++j) {

// 计算灰度值（略）

}

}

}

collapse(2)：将二维循环合并为单一迭代空间

内存访问模式：连续的行优先访问，提升缓存利用率

**性能优化：**

采用OpenMP并行化核心计算，通过#pragma omp parallel for collapse(2)将嵌套循环合并为单一迭代空间，利用多线程对图像像素进行分块处理，在保证内存连续访问（行优先模式）的同时实现线程级负载均衡,并通过内存访问优化，以行主序遍历像素，最大化缓存命中率，减少内存延迟。

**4.1MPI并行**

**整体流程：**

开始

├─ 主进程（Rank 0）收集图像路径列表

├─ 广播路径数据到所有进程

├─ 动态任务分配（余数优先法）

├─ 各进程独立处理分配到的图像

├─ 同步计时统计最大运行时间

└─ 主进程输出总耗时

结束结束

**伪代码流程：**

MPI\_Init();

MPI\_Comm\_rank(); // 获取进程号

MPI\_Comm\_size(); // 获取总进程数

if (rank == 0) {

遍历输入文件夹，收集所有图像路径;

打印总图像数量;

}

MPI\_Bcast(总图片数); // 广播图像总数

MPI\_Bcast(图像路径数组); // 广播路径字符串数据（固定长度格式）

// 块划分 + 余数优先任务分配

计算当前进程负责的图像编号范围 (start, end);

提取对应路径到本地路径列表;

// 确保输出目录存在

if (!输出目录存在) 创建输出目录;

MPI\_Barrier(); // 所有进程同步，开始计时

记录起始时间 = MPI\_Wtime();

for 每张本地图像:

读取图像;

调用手动灰度转换函数;

保存灰度图到输出目录;

MPI\_Barrier(); // 所有进程同步，结束计时

记录结束时间 = MPI\_Wtime();

计算本地耗时;

MPI\_Reduce(最大耗时收集到主进程);

if (rank == 0) {

打印总图像数量、进程数、最大耗时;

}

MPI\_Finalize();**性能优化实现：**

采用余数优先的任务分配策略，将总图像数按进程数均分，前N%P个进程多处理1张图像，确保最大负载差≤1，实现静态负载均衡；

优化数据通信，主进程通过固定长度字符数组（MAX\_PATH\_LEN=260）一次性打包所有路径，利用MPI\_Bcast广播避免多次通信，减少协议开销；

并行I/O设计上，各进程独立读写分配的图像，输出文件名嵌入进程号（如processed\_rankX）避免写入冲突；

精准耗时统计则通过MPI\_Barrier同步所有进程起止点，以MPI\_Reduce取最大本地耗时作为全局真实运行时间；

内存与计算优化包括预分配路径数组降低动态内存开销、本地路径列表缓存减少解析耗时。

**4.3CUDA并行**

**整体流程**

开始

├─ 主程序批量读取图像路径

├─ 对每张图片循环处理:

│ ├─ GPU内存分配与数据拷贝

│ ├─ 配置CUDA核函数线程块与网格

│ ├─ 启动核函数并行计算灰度值

│ ├─ 同步计时并回传结果

│ └─ 执行CPU串行版本对比

└─ 输出总耗时

结束

**CUDA核函数伪代码**

\_\_global\_\_ void rgb2grayInCuda(uchar3\* input, uchar\* output, int H, int W) {

int x = blockIdx.x \* blockDim.x + threadIdx.x;

int y = blockIdx.y \* blockDim.y + threadIdx.y;

if (x < W && y < H) {

uchar3 rgb = input[y \* W + x];

output[y \* W + x] = 0.299\*rgb.x + 0.587\*rgb.y + 0.114\*rgb.z;

}

}

**性能优化实现：**

线程配置上采用二维布局（AxA线程块，网格尺寸按(W+A-1)/A x (H+A)/A-1动态计算），确保每个像素由独立线程处理且完全覆盖图像；

内存访问优化方面，通过uchar3数据类型匹配OpenCV的BGR内存布局实现合并访问（线程按行连续读取RGB像素），单次cudaMemcpy批量传输整图数据减少PCIe通信开销；

计算层面将灰度系数预定义为常量（0.299f/0.587f/0.114f）降低指令数，并通过x < W && y < H边界判断避免无效线程计算，减少Warp分支分化；

资源管理上严格同步设备流（cudaDeviceSynchronize）确保计时准确，并集中释放GPU内存降低碎片化。

**4.4OpenMP+MPI混合并行**

**执行流程**

开始

├─ MPI初始化，确定进程数/rank

├─ 主进程收集图像路径列表

├─ MPI\_Bcast广播路径数据

├─ 任务分配（余数优先法）

├─ 各进程并行处理分配到的图像：

│ ├─ OpenMP并行灰度转换（collapse(2)）

│ └─ 保存结果（文件名含进程号）

└─ 同步计时并输出统计信息

结束

**混合并行程序伪代码：**

开始:

MPI\_Init() // 初始化MPI环境

获取当前进程rank和总进程数size

定义输入/输出目录路径

检查输入目录是否存在（所有进程）

主进程（rank=0）:

遍历输入目录，收集所有图像路径到imagePaths列表

计算总图像数totalImages = imagePaths.size()

MPI\_Bcast(totalImages) // 广播总图像数给所有进程

主进程:

分配固定长度路径数组allPaths[totalImages][MAX\_PATH\_LEN]

将imagePaths复制到allPaths

MPI\_Bcast(allPaths) // 广播路径数据

// 任务分配（余数优先法）

chunk = totalImages / size

remainder = totalImages % size

start = rank \* chunk + min(rank, remainder)

end = start + chunk + (rank < remainder ? 1 : 0)

end = min(end, totalImages)

提取本地路径列表localPaths = allPaths[start...end-1]

创建输出目录（同步所有进程）

设置OpenMP线程数（通过命令行参数）

omp\_set\_num\_threads(omp\_threads)

MPI\_Barrier() // 同步计时起点

startTime = MPI\_Wtime()

对每个路径 in localPaths: // MPI进程级并行

读取图像srcImg

创建灰度图像grayImg

#pragma omp parallel for collapse(2) // OpenMP线程级并行

for (i=0 to srcImg.height-1):

for (j=0 to srcImg.width-1):

idx = i\*width + j

grayImg[idx] = 0.299\*srcImg[idx].R + 0.587\*srcImg[idx].G + 0.114\*srcImg[idx].B

生成含进程号的文件名: hybrid\_rank{rank}\_filename

保存grayImg到输出目录

MPI\_Barrier() // 同步计时终点

totalTime = MPI\_Wtime() - startTime

主进程（rank=0）:

输出统计信息:

- 总图像数

- MPI进程数

- OpenMP线程数/进程

- 总耗时

MPI\_Finalize()

结束

**性能优化实现**

混合并行代码通过MPI与OpenMP的多层次优化实现高效处理：MPI层级采用余数优先任务分配策略（start = rank\*chunk + min(rank, remainder)），确保进程间最大负载差≤1，并通过固定长度数组（MAX\_PATH\_LEN=260）单次广播全量路径，将通信复杂度从O(N)降至O(1)；

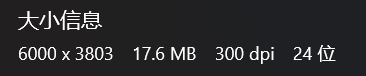
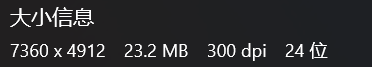
OpenMP层级使用collapse(2)合并行列循环，将二维图像展开为线程网格（如4K图像生成4096x2160线程任务），按行主序访问内存（idx = i\*imgWidth + j）提升缓存命中率，支持命令行参数动态设置线程（omp\_set\_num\_threads(atoi(argv[1]))）；

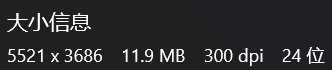
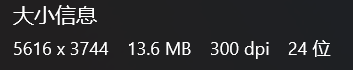
混合架构中MPI进程处理图像间并行，各进程内部通过OpenMP实现像素级并行，输出文件嵌入进程号（hybrid\_rankX）避免跨节点写入冲突。

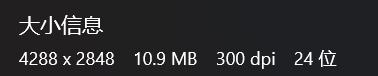
1. **实验设置**

**5.1 实验数据**

共20张大尺寸图片，分为四组，每组为5张重复的相同图片。







**5.2 实验软硬件环境**

**硬件：**



**串行环境配置：**VS2022，OpenCV

**OpenMP并行环境：**VS2022，OpenCV

**MPI并行环境：**VS2022，OpenCV，MPI，通过命令行运行

**CUDA并行：**VS2022，OpenCV，CUDA12.2.0

**OpenMP+MPI混合并行：**VS2022，OpenCV，MPI，通过命令行运行

**5.3 评价指标**

由于灰度图的公式确定为：*Gray*=0.299×*R*+0.587×*G*+0.114×*B*，不同并行方法（OpenMP、MPI、CUDA等）生成的灰度图**精度差异极小**，理论上应完全一致。实际可能因浮点运算顺序、数据类型转换或编译器优化产生微小误差（如MSE≈1e-7），但通过统一使用单精度浮点（**0.299f**）、确保全覆盖计算、禁用激进优化等措施，差异可控制在视觉不可见范围（PSNR>80dB）。规范实现的并行算法间差异像素数通常为0，**实际应用中可认为各方法精度等价**。

我们将主要分析**加速比 (Speedup)和效率（Efficiency）**来衡量并行化带来的性能提升：

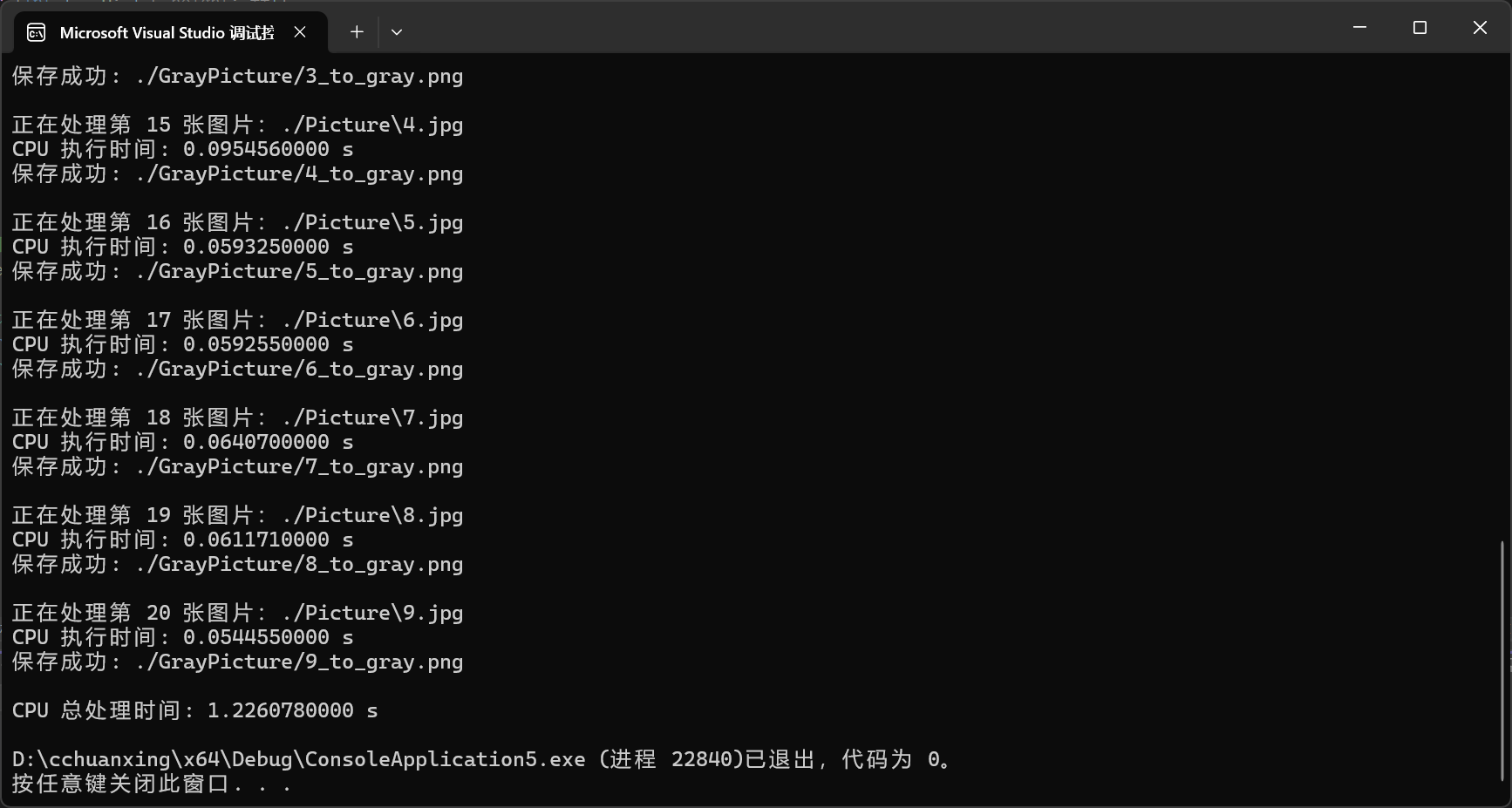
***S*= ​*T*serial​​ */T*parallel**

**Efficiency=Speedup */*使用的处理单元数**

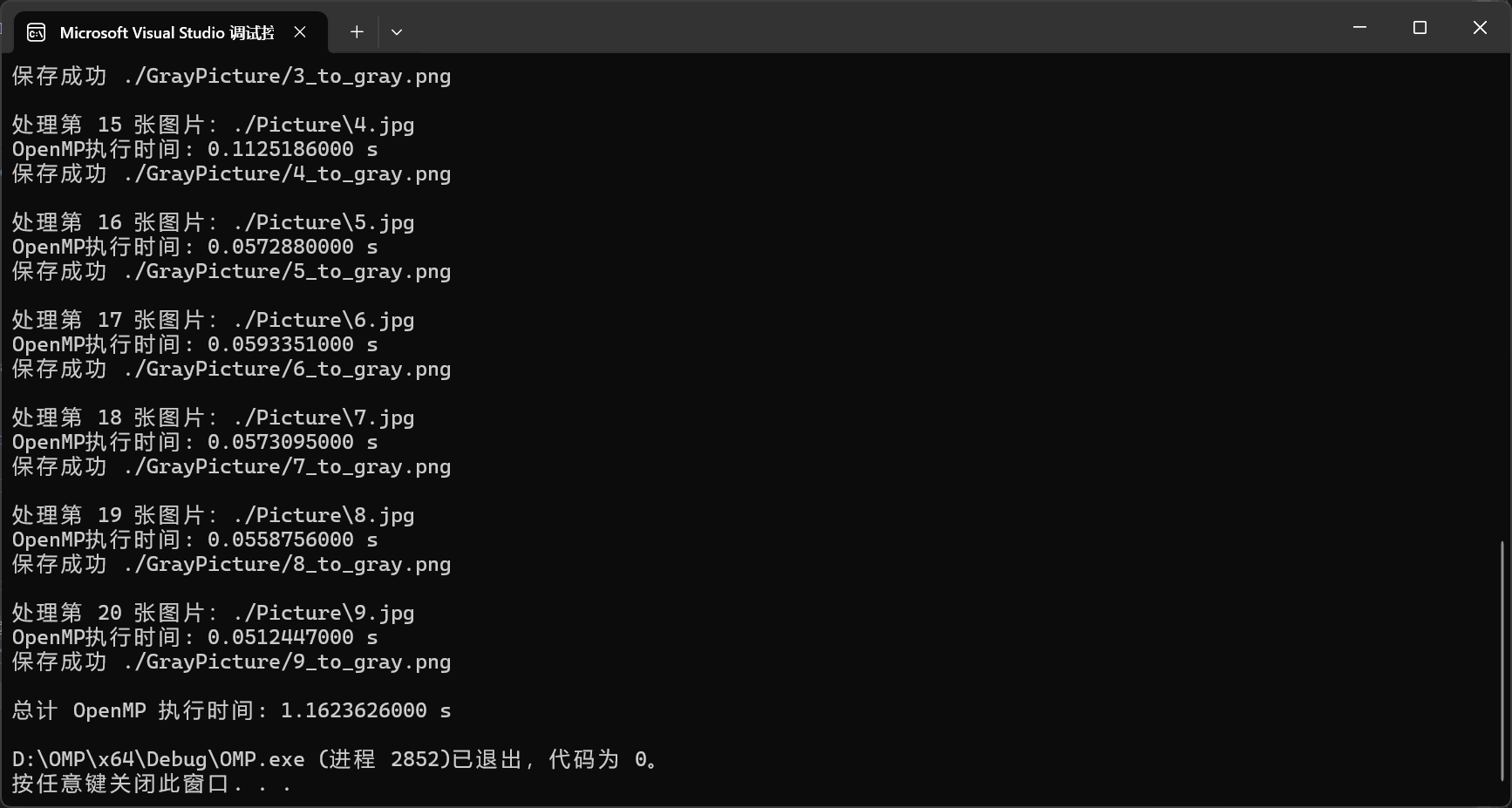
1. **实验结果与分析**

**6.1程序运行截图**

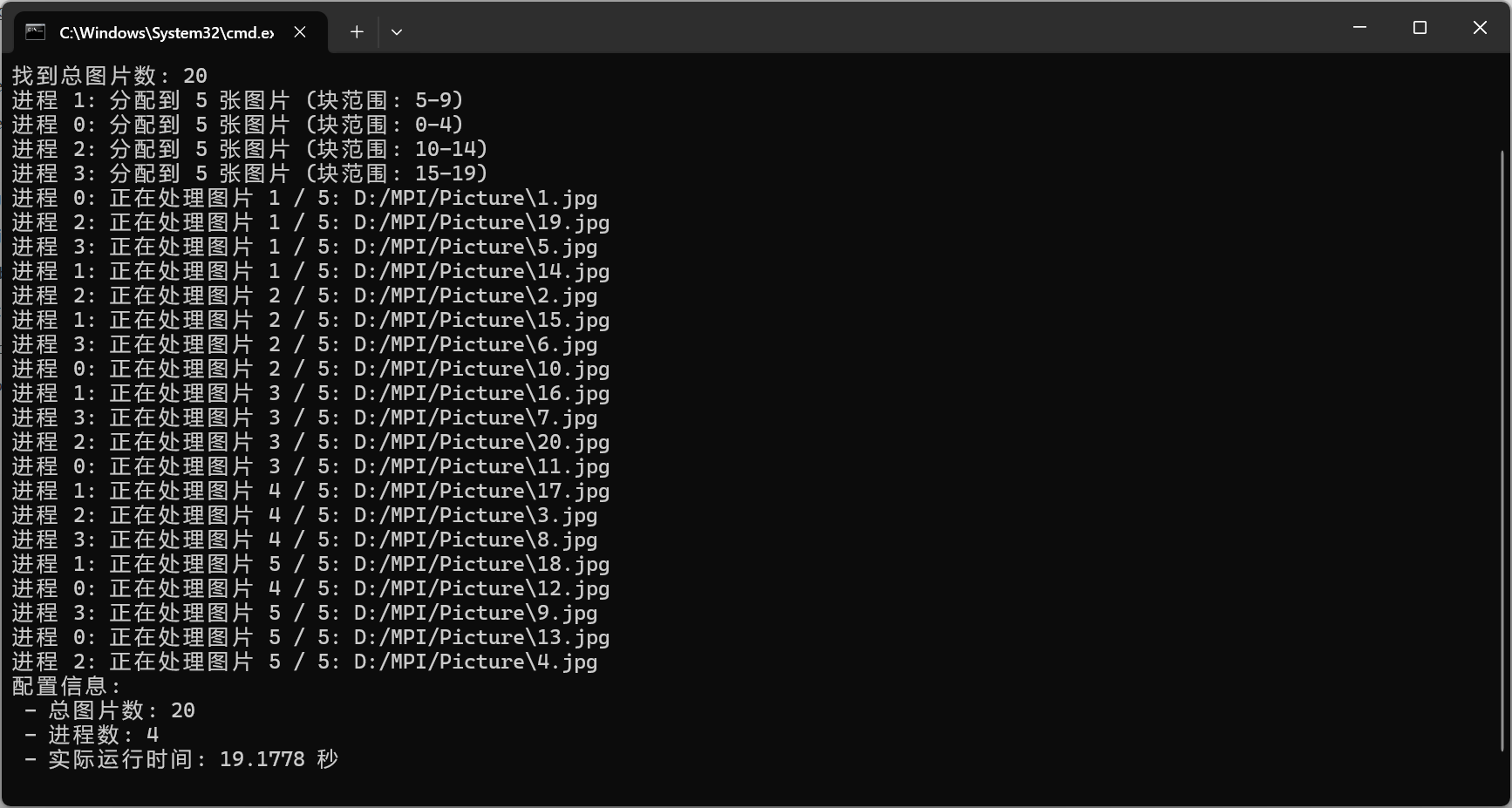
C++串行程序运行截图：



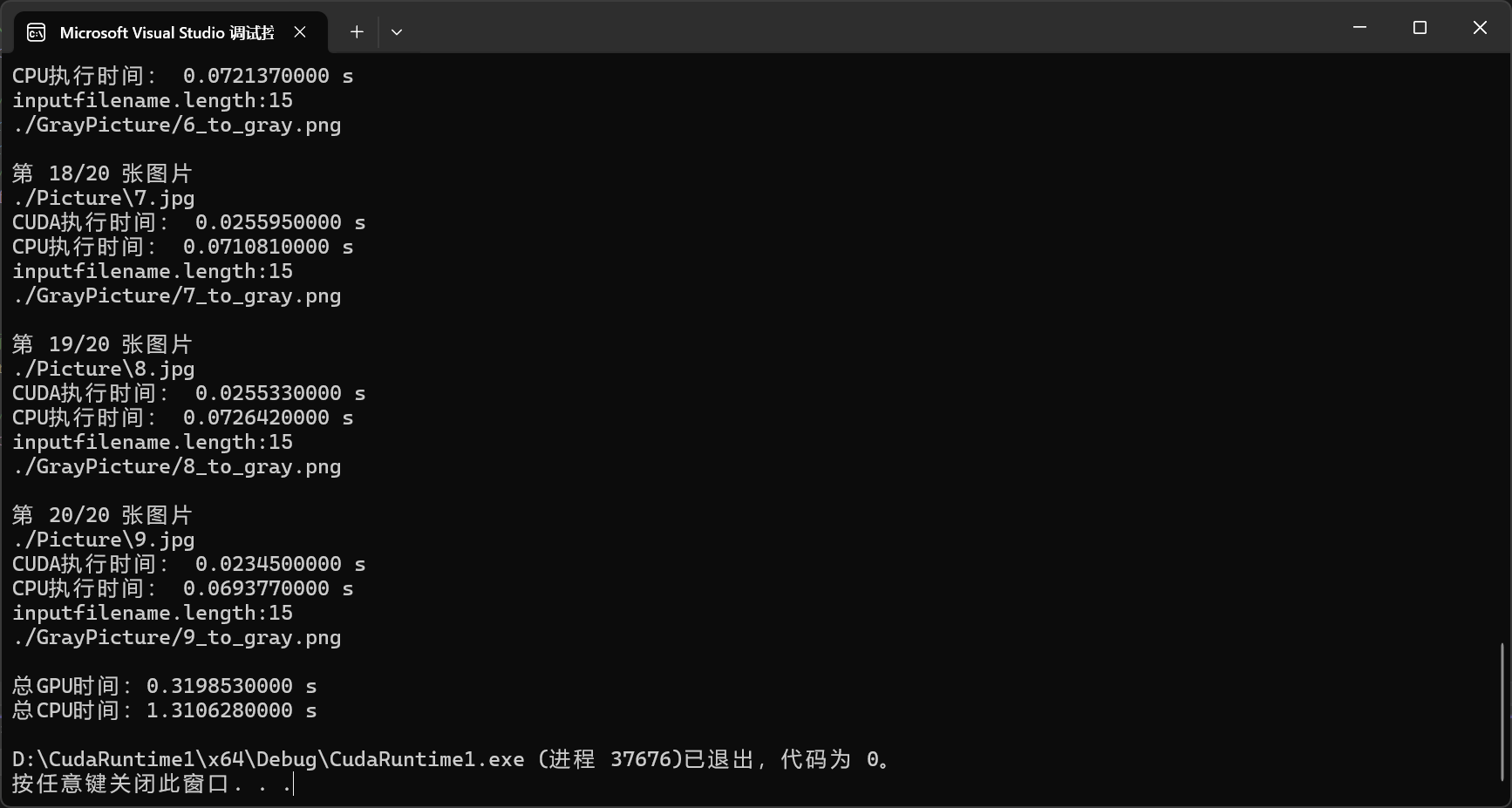
OpenMP并行程序运行截图：



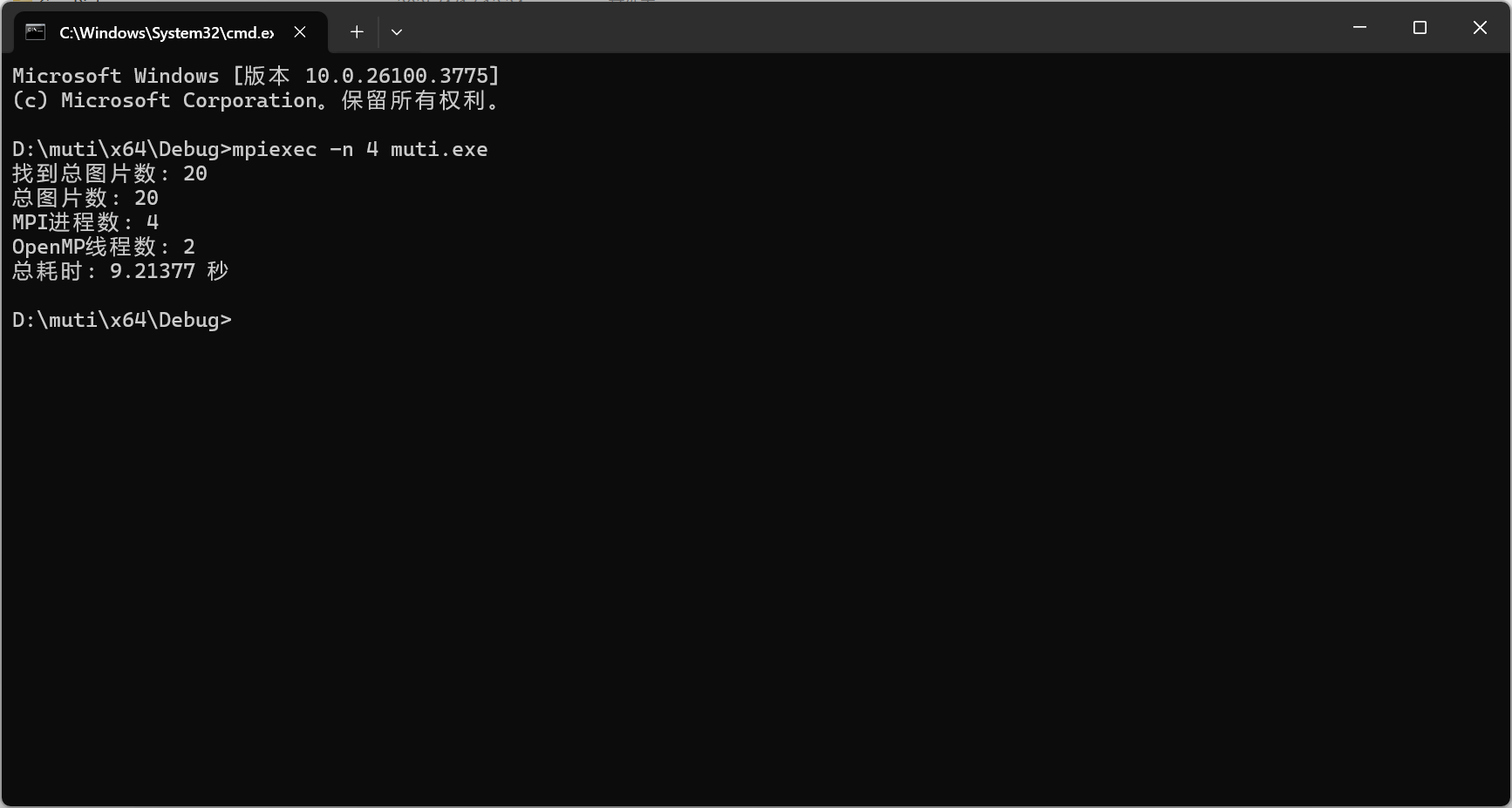
MPI并行程序运行截图：



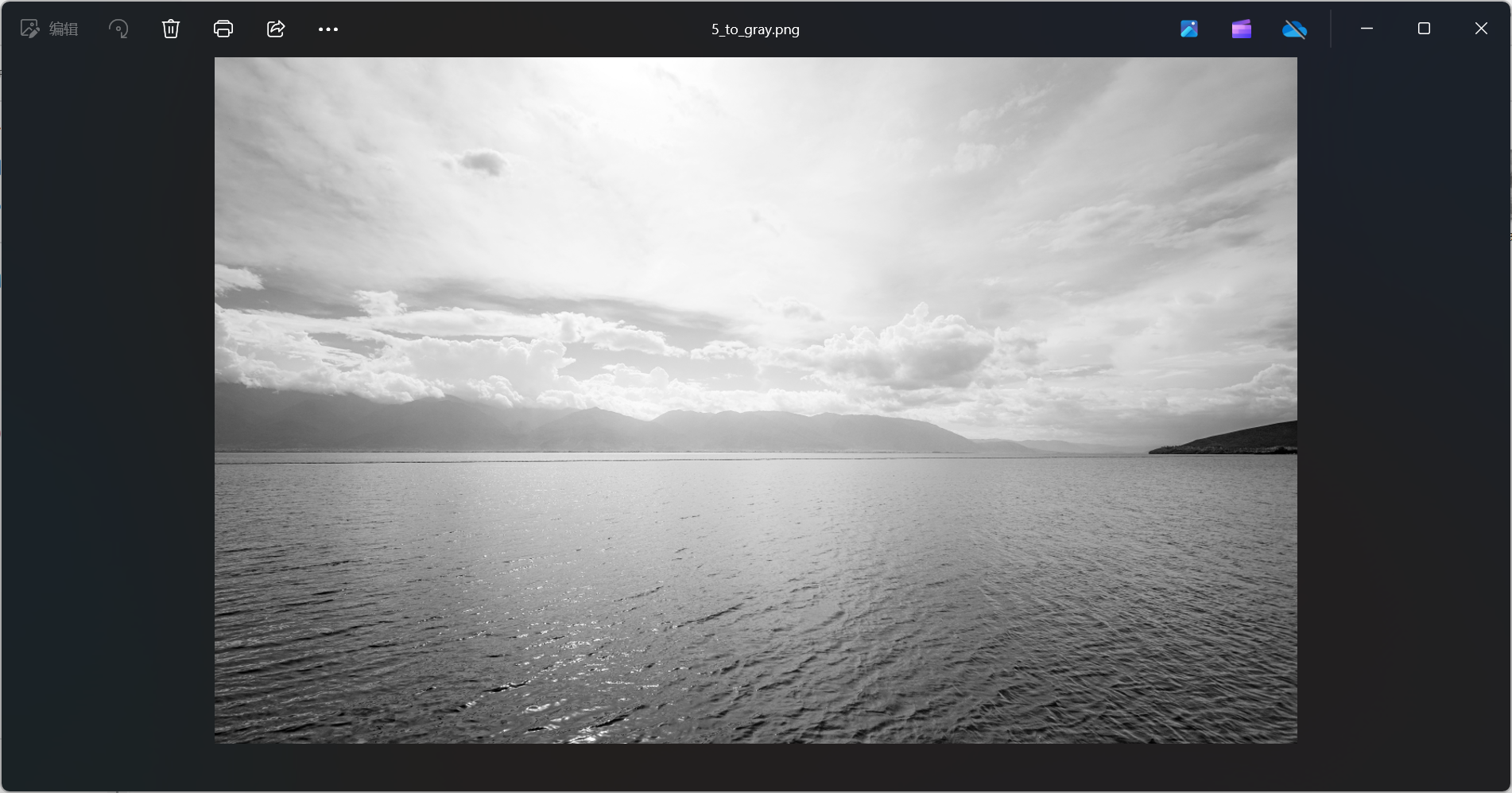
CUDA并行程序运行截图：



OpenMP+MPI混合并行程序运行截图：



灰度图：

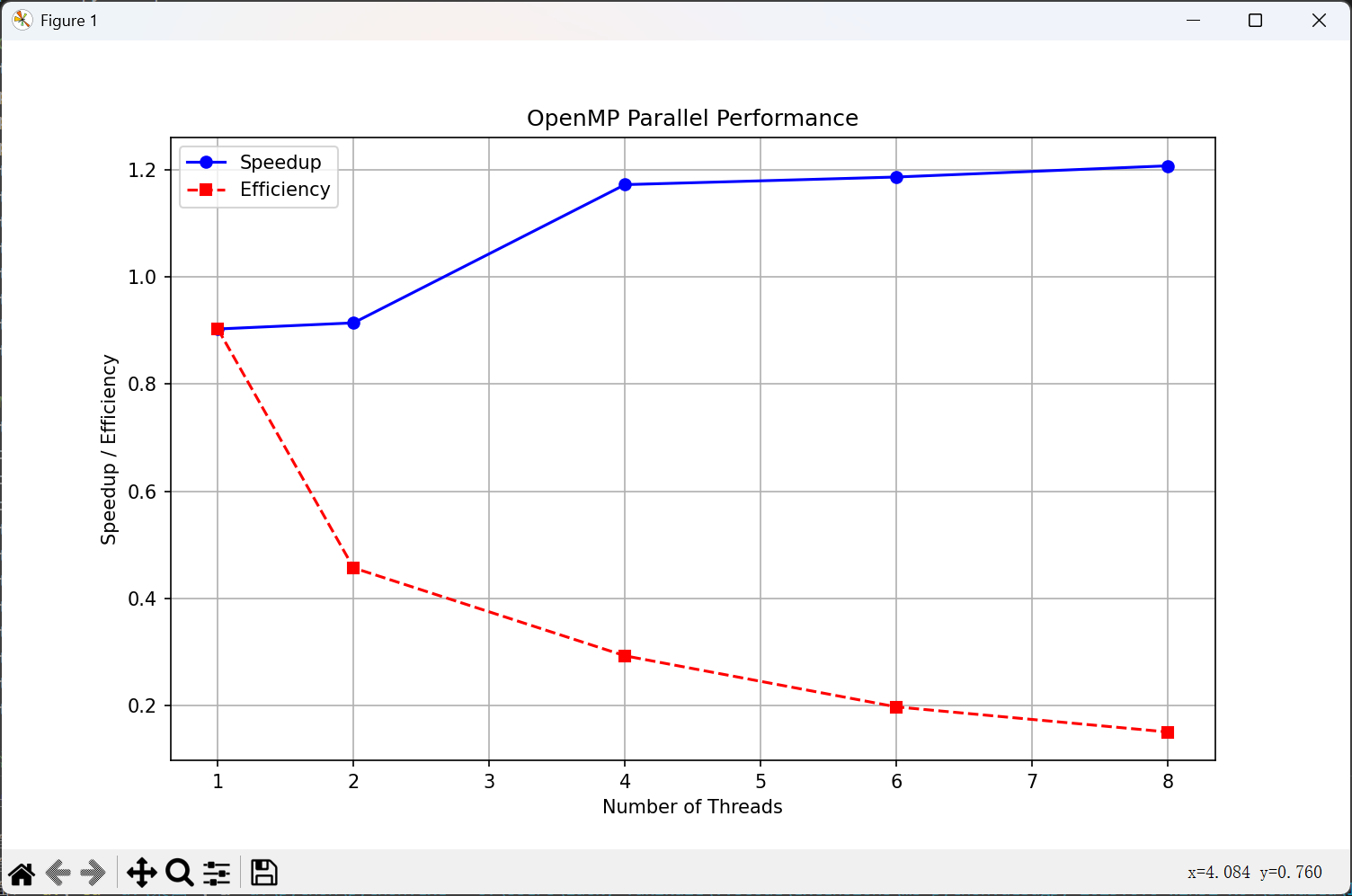


**6.2实验数据分析**

**OpenMP并行性能分析**

其中 T\_serial = 1.226078s（C++串行程序总CPU时间）

| 线程数 | OpenMP时间 (s) | 加速比（Speedup） |
| --- | --- | --- |
| 1 | 1.358167 | 0.9027 |
| 2 | 1.3406714 | 0.9141 |
| 4 | 1.0462797 | 1.1720 |
| 6 | 1.0334276 | 1.1862 |
| 8 | 1.0158813 | 1.2071 |



**观察发现，多线程带来的加速 非常有限（最高仅提升约 20%）并且使用 1 线程的 OpenMP 实际反而比串行还慢**

**OpenMP 并行效果不佳的原因分析：**

**1.任务粒度小，线程创建/调度开销大：**

图像灰度处理的单张任务计算量较小，线程创建和调度的开销抵消了并行带来的收益。

**2.** **OpenMP 单线程时间比串行程序还慢：**串行程序的循环更“紧凑”、更直接，而 OpenMP 版本中增加了线程管理、任务切分，反而拖慢了进程。

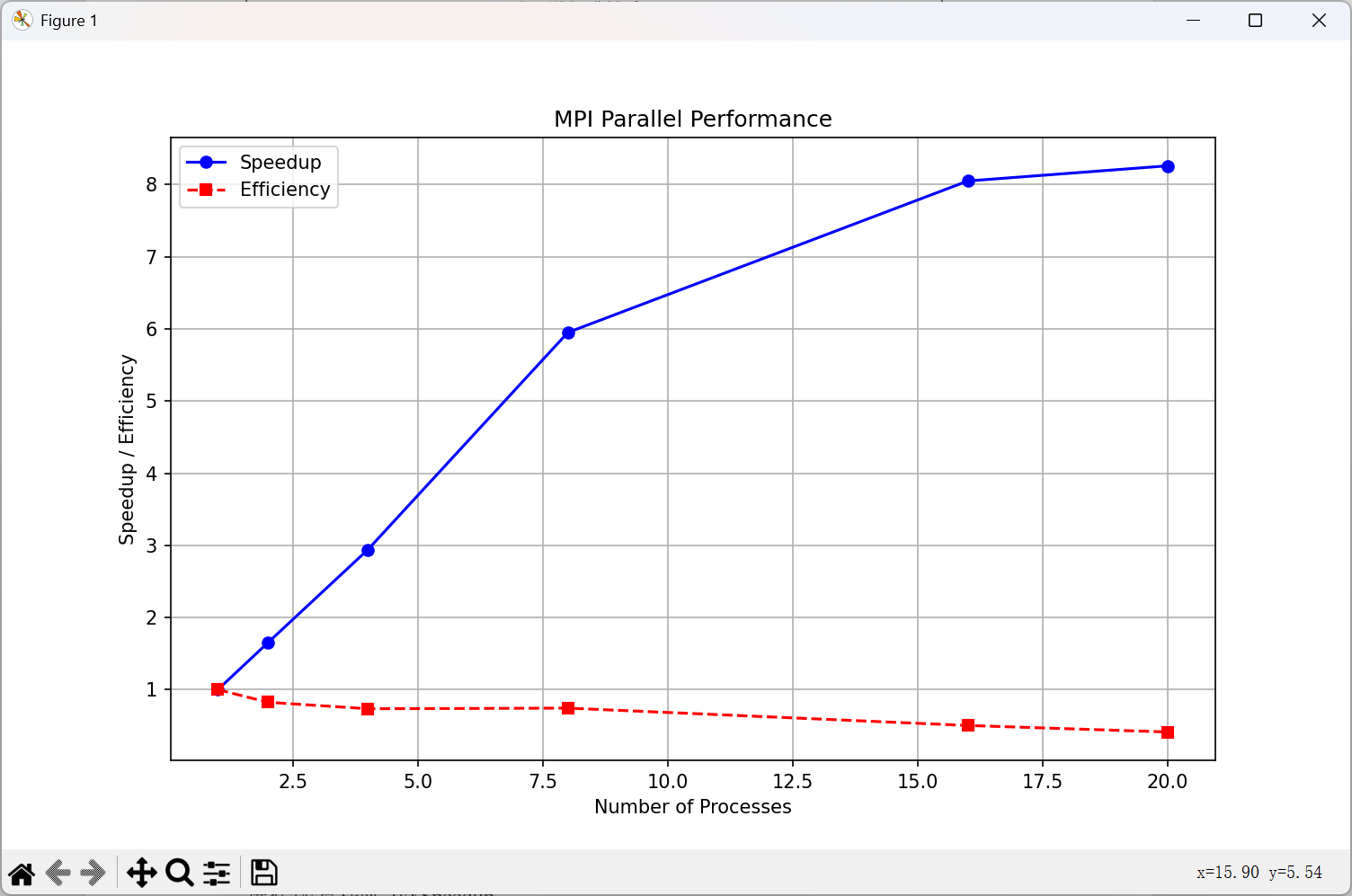
**3. OpenMP适合CPU密集、重计算、低I/O的小块任务，**而图像灰度处理偏向轻计算、重I/O，因此OpenMP效果不显著。

**MPI并行分析**

MPI加速比（相对1进程）：

| 进程数 | 总耗时 (s) | Speedup |
| --- | --- | --- |
| 1 | 30.2116 | 1.00x |
| 2 | 18.3566 | 1.65x |
| 4 | 10.2615 | 2.94x |
| 8 | 5.07981 | 5.95x |
| 16 | 3.75089 | 8.05x |
| 20 | 3.65798 | 8.26x |

\* MPI的计时从任务开始到全部结束，体现了真实用户感受（wall-clock time），体感上反而更快

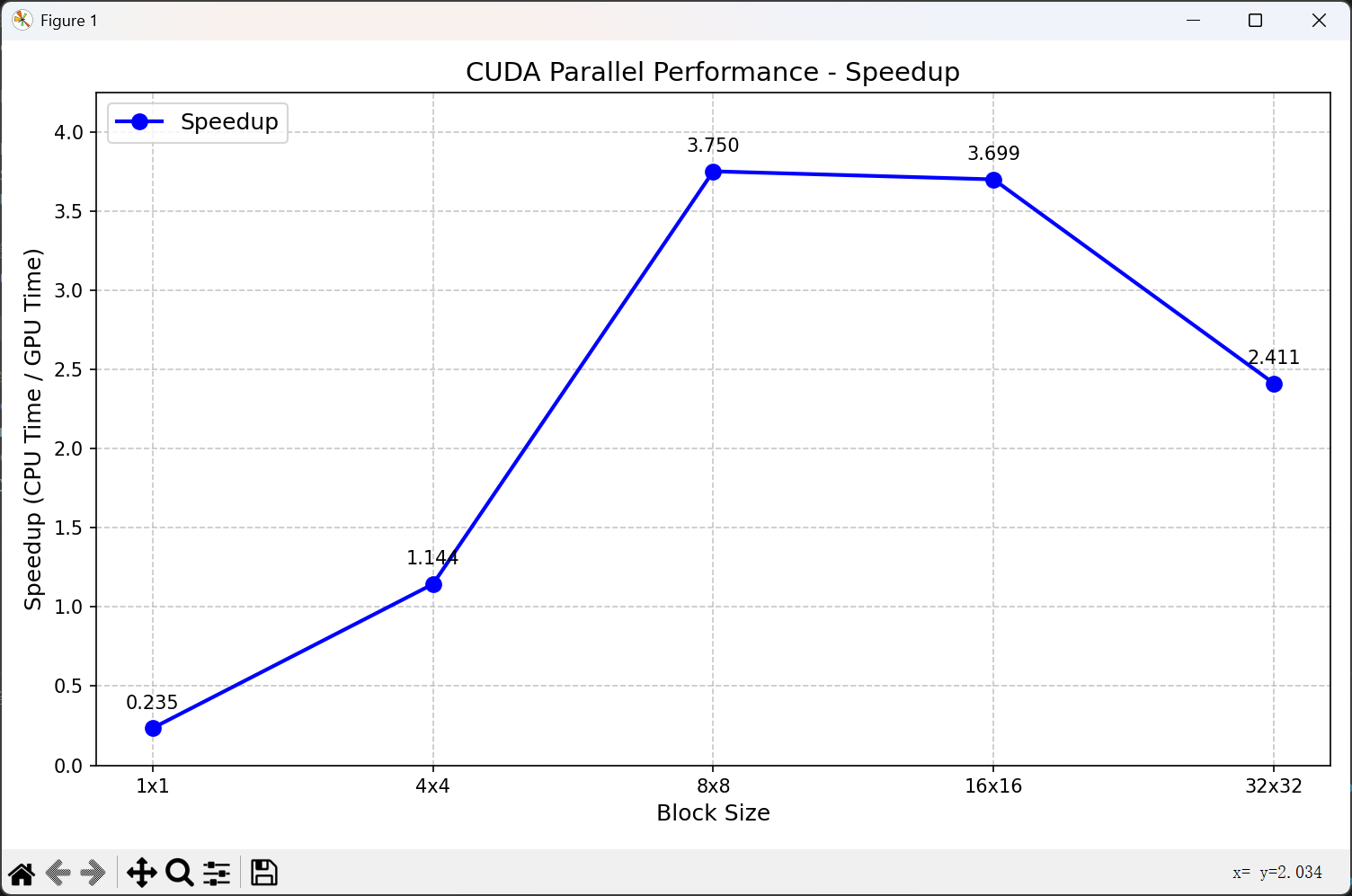


**观察发现：加速效果随进程数近似线性提升（较理想）超过8进程后，提升趋缓但仍有提升，即使包括I/O时间，MPI仍然表现出优秀加速性能。**

**OpenMP在灰度图像处理任务中并未显著提升速度，主要由于任务粒度小、I/O比重高、线程调度开销高。相比之下，MPI采用独立进程、合理任务划分、并行I/O处理，实现了更优秀的加速比和实际体验，特别适合高I/O、高并发的图像处理场景。**

**CUDA并行分析**

| Block Size | Speedup = CPU / GPU |
| --- | --- |
| 1x1 | 1.1615 / 4.9457 ≈ 0.235 |
| 4x4 | 1.1709 / 1.0242 ≈ 1.144 |
| 8x8 | 1.1735 / 0.3129 ≈ 3.750 |
| 16x16 | 1.1713 / 0.3167 ≈ 3.699 |
| 32x32 | 1.1216 / 0.4652 ≈ 2.411 |



**结论：**8x8 和 16x16 提供最高加速比，达到约 3.7～3.75 倍，说明这两种配置下 GPU 使用最为高效。

1x1 每个线程块仅含1线程（dim3 threadsPerBlock(1,1)），导致GPU计算单元利用率极低

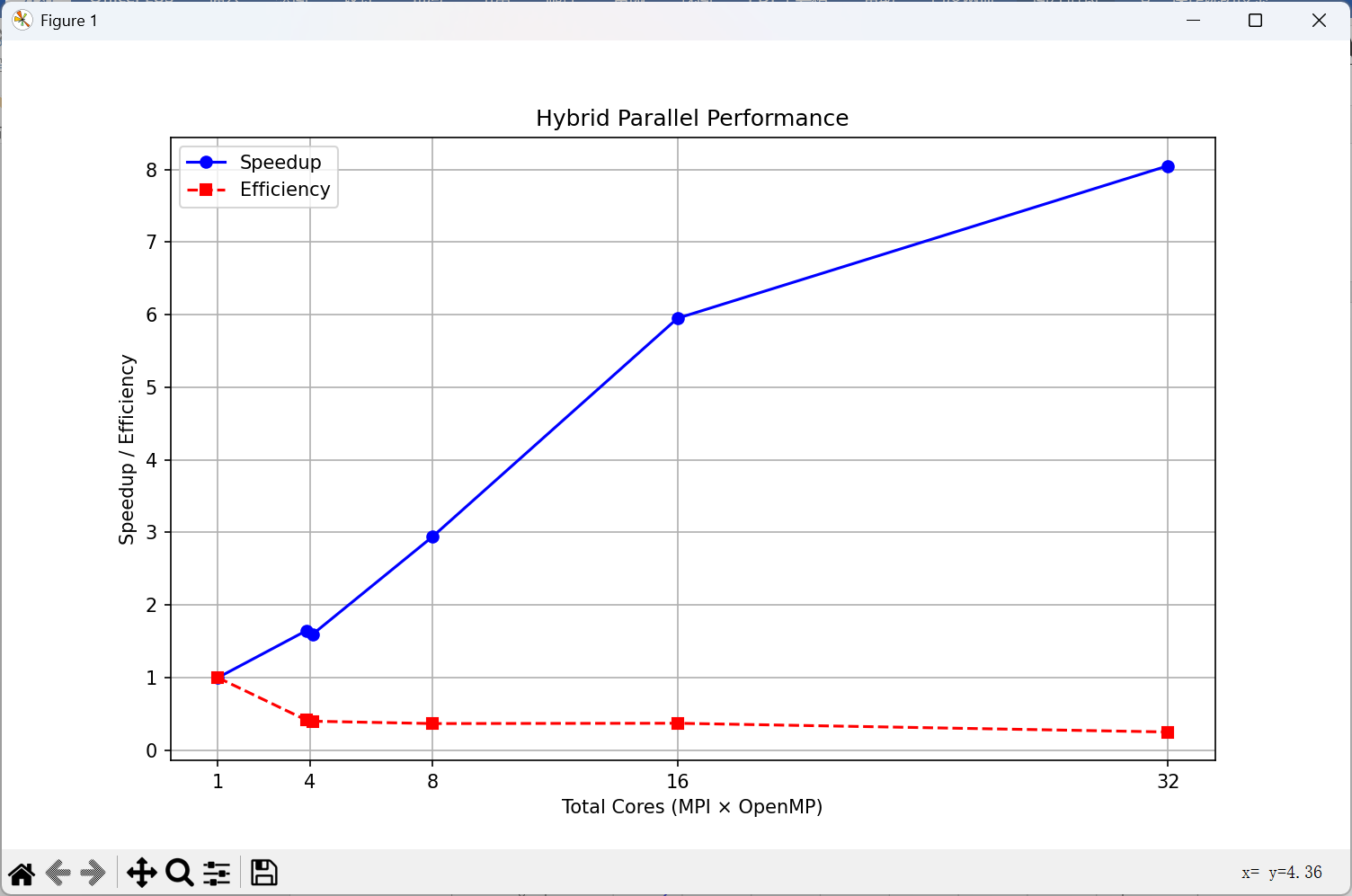
4x4 为转折点，开始表现出超越CPU的优势，但并未显著加速。

32x32 性能下降，说明线程块过大可能导致资源不足（如共享内存或寄存器压力），降低了并行效率。

**OpenMP+MPI混合并行性能分析**

| 实验编号 | MPI进程数 | OpenMP线程数/进程 | 总核数（估算） | 总耗时（秒） |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 30.2116 |
| 2 | 2 | 2 | 4 | 18.3566 |
| 3 | 2 | 2 | 4 | 18.8626 |
| 4 | 4 | 2 | 8 | 10.2615 |
| 5 | 8 | 2 | 16 | 5.07981 |
| 6 | 16 | 2 | 32 | 3.75089 |

| 实验编号 | 总耗时(s) | 加速比（Speedup） |
| --- | --- | --- |
| 1 | 30.2116 | 1.00 |
| 2 | 18.3566 | ≈ 1.65 |
| 3 | 18.8626 | ≈ 1.60 |
| 4 | 10.2615 | ≈ 2.94 |
| 5 | 5.07981 | ≈ 5.95 |
| 6 | 3.75089 | ≈ 8.05 |



计算发现，加速比总体是递增的，说明混合并行有效，但有边际递减现象。

并行效率开始在40%左右，最大仅约37%，最后下降到25%左右，这是由于：图片数量固定（任务较小），核数过多导致资源浪费；OpenMP线程内开销；MPI进程间通信与调度开销上升。所以，4-16个核为性能提升的“性价比高区间”，后面性能增益变小；

最快运行时间为 3.75秒（32核），对比单核提升约 8倍，但使用了 32倍资源。

1. **总结与讨论**

值得注意的是，CUDA和OpenMP的计时仅反映核心计算时间，MPI/混合的 wall-clock 计时包括了分配任务、通信、同步、合并等实际运行中不可避免的过程。因此，即使耗时更高的OpenMP看似比MPI快，但用户真实等待时间是MPI类并行更短。

**结论：**

混合并行（MPI+OpenMP）在用户“真实等待体验”上最优，兼具通信效率与计算性能，尤其适合真实部署。

**CUDA**计算速度惊人，但受限于**内存拷贝开销未计入**，在真实工程中常伴有传输延迟，适合批量图像、矩阵运算等GPU友好型任务。

**OpenMP**和**CUDA**适合密集、数据局部性强的任务，但在需要大量文件读写或调度时，不敌MPI类。

本项目选题为**图像灰度批量处理**，具有较强的实用性和广泛的应用场景，例如图像预处理、机器学习数据准备、图像压缩等领域。选题本身具备数据量大、计算密集的特点，因此非常适合探索和比较多种并行计算策略。解决方案中，我们首先实现了串行版本以作为基准，然后引入了三类并行化手段：OpenMP 多线程并行、CUDA GPU 并行以及 MPI 分布式并行，并进一步探索了 MPI + OpenMP 的混合并行方式。在实验中，我们发现 CUDA 并行在纯计算阶段具有极高的加速比，而 MPI 混合并行在 wall-clock 时间上则体现了最优的用户体验，尤其适用于任务粒度较大或需要跨节点分布的情况。综合分析表明，不同并行方法适用于不同的应用场景，未来可以进一步延伸为**自适应并行策略选择系统**，根据图像数量、单图大小及硬件平台智能选择最优并行方式；同时也可拓展到更复杂的图像处理操作（如滤波、边缘检测），进一步增强系统的通用性和性能。

**参考文献**

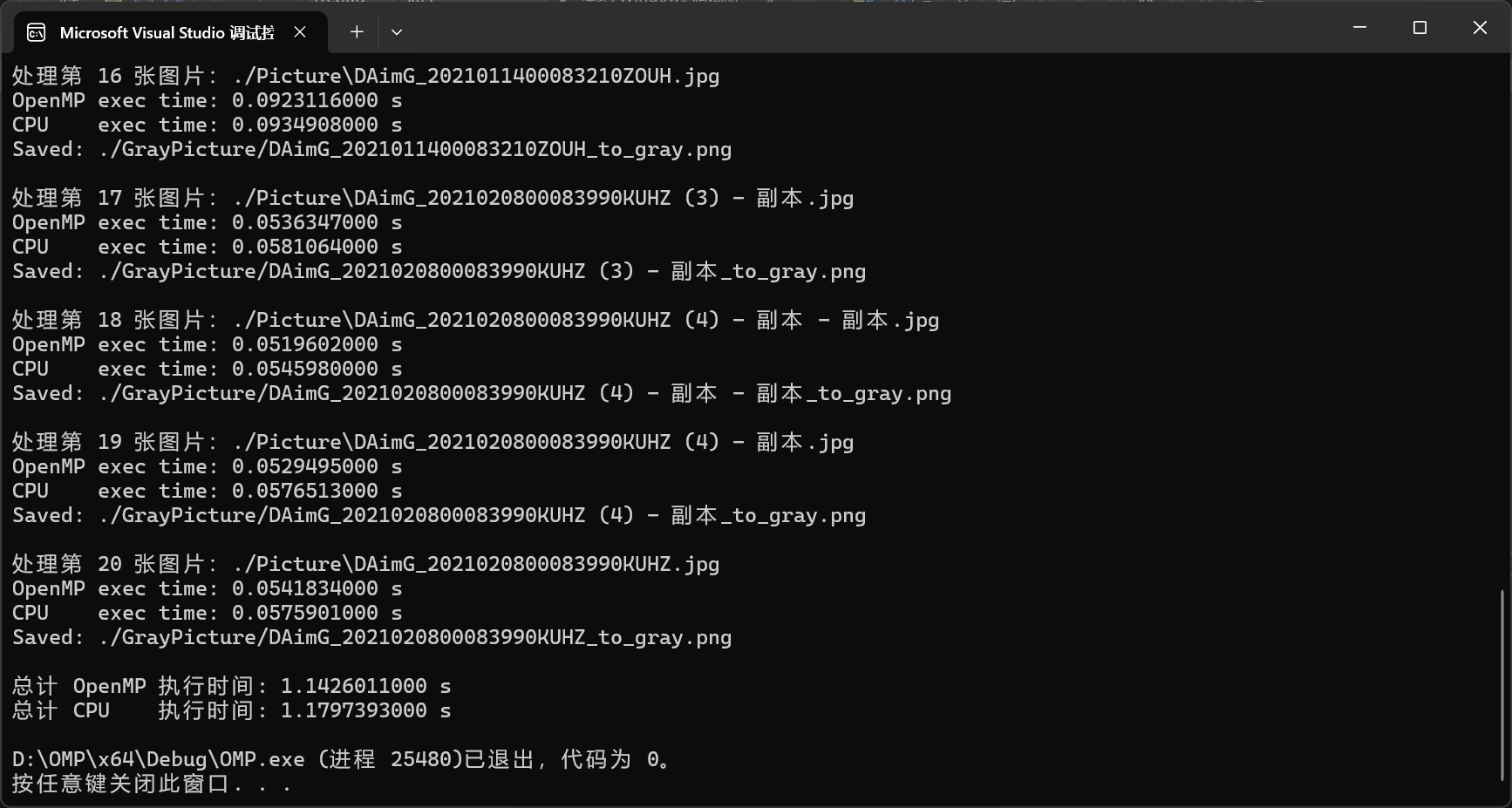
1.[并行计算程序设计案例（CUDA12.2+VS2022+opencv4.8）图像处理\_vs2022 opencv cuda-CSDN博客](https://blog.csdn.net/qq_45774917/article/details/135377802?ops_request_misc=%257B%2522request%255Fid%2522%253A%25225f8bf8f5aa0dc3bf1156e0db62a68e9c%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=5f8bf8f5aa0dc3bf1156e0db62a68e9c&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~baidu_landing_v2~default-5-135377802-null-null.142%5ev102%5epc_search_result_base3&utm_term=%E5%B9%B6%E8%A1%8C%E8%AE%A1%E7%AE%97%E8%AF%BE%E7%A8%8B%E8%AE%BE%E8%AE%A1&spm=1018.2226.3001.4187)

2.[VisualStudio2022下配置 OpenMP多线程编程环境与运行\_C 语言\_脚本之家](https://www.jb51.net/program/322932eub.htm)

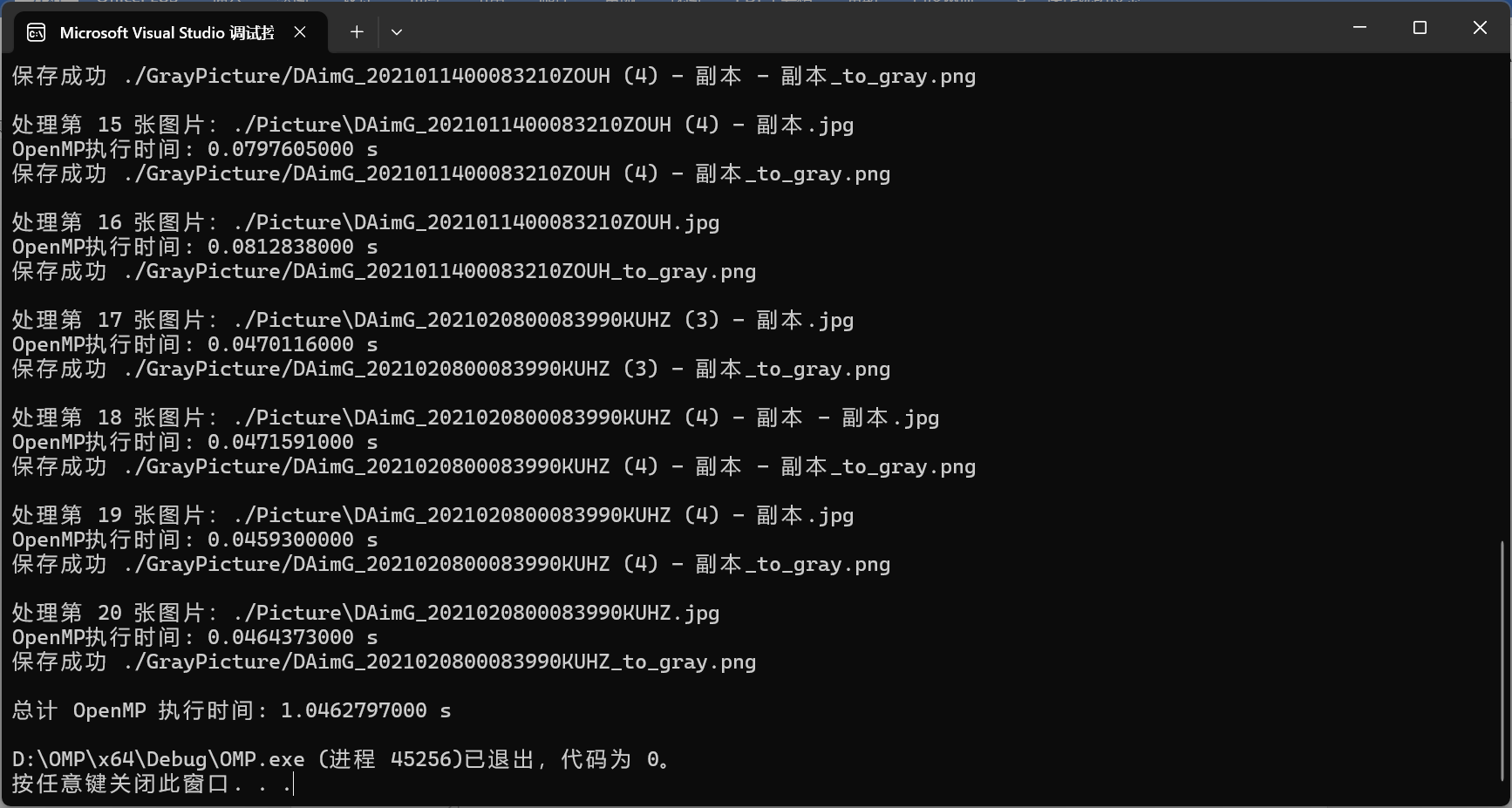
3.[VS2022配置MPI环境\_vs mpi-CSDN博客](https://blog.csdn.net/m0_58064058/article/details/129543203)

**附件一：性能优化过程记录表（必备）**

1.在编写OpenMP并行时，尝试采用并行和串行一同运行，分别统计运行时间来直观体现二者性能差异，后发现此举将消耗更多的资源，反而不利于性能优化，遂放弃此思路。



二者一起运行时（上图）所花费的时间略多于单独运行OpenMP（下图）（设置线程数为4）



**附件二：关键源码说明**

C++串行代码

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <vector>

#include <string>

#include <cstdio>

#include <cstdlib>

#include <cassert>

#include <chrono>

#include <io.h>

#include <opencv2/core/core.hpp>

#include <opencv2/highgui/highgui.hpp>

#include <opencv2/opencv.hpp>

using namespace cv;

using namespace std;

// 灰度转换（串行）

void rgb2graySerial(const unsigned char\* const in, unsigned char\* const out, int height, int width)

{

for (int i = 0; i < height; ++i)

{

for (int j = 0; j < width; ++j)

{

int idx = i \* width + j;

out[idx] = 0.299f \* in[idx \* 3] + 0.587f \* in[idx \* 3 + 1] + 0.114f \* in[idx \* 3 + 2];

}

}

}

// 递归获取文件

void getFiles(string path, vector<string>& files)

{

intptr\_t hFile = 0;

struct \_finddata\_t fileinfo;

string p;

if ((hFile = \_findfirst(p.assign(path).append("\\\*").c\_str(), &fileinfo)) != -1)

{

do

{

if (!(fileinfo.attrib & \_A\_SUBDIR))

files.push\_back(p.assign(path).append("\\").append(fileinfo.name));

else if (strcmp(fileinfo.name, ".") != 0 && strcmp(fileinfo.name, "..") != 0)

getFiles(p.assign(path).append("\\").append(fileinfo.name), files);

} while (\_findnext(hFile, &fileinfo) == 0);

\_findclose(hFile);

}

}

// 主处理函数（串行）

int SerialFunc(string inputfilename, double& cpusumtime)

{

Mat srcImg = imread(inputfilename);

if (srcImg.empty())

{

cerr << "无法打开图片: " << inputfilename << endl;

return -1;

}

int imgHeight = srcImg.rows;

int imgWidth = srcImg.cols;

Mat grayImg(imgHeight, imgWidth, CV\_8UC1, Scalar(0));

int hist[256] = { 0 };

auto cpu\_start = chrono::high\_resolution\_clock::now();

rgb2graySerial(srcImg.data, grayImg.data, imgHeight, imgWidth);

auto cpu\_end = chrono::high\_resolution\_clock::now();

auto cpu\_duration = chrono::duration\_cast<chrono::microseconds>(cpu\_end - cpu\_start);

double cpu\_time = cpu\_duration.count() / 1e6;

cpusumtime += cpu\_time;

cout << fixed << setprecision(10) << "CPU 执行时间: " << cpu\_time << " s" << endl;

// 保存灰度图

try

{

string outputname = "./GrayPicture/" + inputfilename.substr(10, inputfilename.length() - 14) + "\_to\_gray.png";

vector<int> compression\_params = { IMWRITE\_PNG\_COMPRESSION, 0 };

imwrite(outputname, grayImg, compression\_params);

cout << "保存成功: " << outputname << endl;

}

catch (runtime\_error& ex)

{

cerr << "图像保存失败: " << ex.what() << endl;

return 1;

}

return 0;

}

int main()

{

string filePath = "./Picture";

vector<string> files;

getFiles(filePath, files);

int size = files.size();

cout << "共检测到图片：" << size << " 张" << endl;

double cpusumtime = 0;

for (int i = 0; i < size; ++i)

{

cout << "\n正在处理第 " << i + 1 << " 张图片：" << files[i] << endl;

SerialFunc(files[i], cpusumtime);

}

cout << "\nCPU 总处理时间: " << cpusumtime << " s" << endl;

FILE\* fp = fopen("time.txt", "a");

if (fp)

{

fprintf(fp, "CPU总时间： %.10lf s\n", cpusumtime);

fclose(fp);

}

return 0;

}

OpenMP并行代码

#include <iostream>

#include <vector>

#include <string>

#include <cstdio>

#include <cstdlib>

#include <iomanip>

#include <cassert>

#include <chrono>

#include <io.h>

#include <omp.h>

#include <opencv2/core/core.hpp>

#include <opencv2/highgui/highgui.hpp>

#include <opencv2/opencv.hpp>

using namespace cv;

using namespace std;

void rgb2grayOpenMP(unsigned char\* const d\_in, unsigned char\* const d\_out, int imgHeight, int imgWidth)

{

#pragma omp parallel for collapse(2)

for (int i = 0; i < imgHeight; ++i)

{

for (int j = 0; j < imgWidth; ++j)

{

int idx = i \* imgWidth + j;

d\_out[idx] = static\_cast<unsigned char>(

0.299f \* d\_in[idx \* 3] +

0.587f \* d\_in[idx \* 3 + 1] +

0.114f \* d\_in[idx \* 3 + 2]);

}

}

}

void getFiles(string path, vector<string>& files)

{

intptr\_t hFile = 0;

struct \_finddata\_t fileinfo;

string p;

if ((hFile = \_findfirst(p.assign(path).append("\\\*").c\_str(), &fileinfo)) != -1)

{

do

{

if ((fileinfo.attrib & \_A\_SUBDIR))

{

if (strcmp(fileinfo.name, ".") != 0 && strcmp(fileinfo.name, "..") != 0)

getFiles(p.assign(path).append("\\").append(fileinfo.name), files);

}

else

{

files.push\_back(p.assign(path).append("\\").append(fileinfo.name));

}

} while (\_findnext(hFile, &fileinfo) == 0);

\_findclose(hFile);

}

}

int OpenMPFunc(string inputfilename, double& openmpsumtime)

{

Mat srcImg = imread(inputfilename);

if (srcImg.empty())

{

cerr << "无法打开图片: " << inputfilename << endl;

return -1;

}

int imgHeight = srcImg.rows;

int imgWidth = srcImg.cols;

Mat grayImg(imgHeight, imgWidth, CV\_8UC1, Scalar(0));

// OpenMP 并行部分

auto omp\_start = chrono::high\_resolution\_clock::now();

rgb2grayOpenMP(srcImg.data, grayImg.data, imgHeight, imgWidth);

auto omp\_end = chrono::high\_resolution\_clock::now();

double omp\_time = chrono::duration<double>(omp\_end - omp\_start).count();

openmpsumtime += omp\_time;

cout << fixed << setprecision(10) << "OpenMP执行时间: " << omp\_time << " s" << endl;

// 保存灰度图

try

{

string outputname = "./GrayPicture/" + inputfilename.substr(10, inputfilename.length() - 14) + "\_to\_gray.png";

vector<int> compression\_params = { IMWRITE\_PNG\_COMPRESSION, 0 };

imwrite(outputname, grayImg, compression\_params);

cout << "保存成功 " << outputname << endl;

}

catch (runtime\_error& ex)

{

cerr << "图像保存失败: " << ex.what() << endl;

return 1;

}

return 0;

}

int main()

{

string filePath = "./Picture";

vector<string> files;

getFiles(filePath, files);

int size = files.size();

cout << "图片数量：" << size << endl;

// 用户输入线程数

int thread\_num;

cout << "请输入使用的线程数：";

cin >> thread\_num;

omp\_set\_num\_threads(thread\_num);

double openmpsumtime = 0;

for (int i = 0; i < size; i++)

{

cout << "处理第 " << i + 1 << " 张图片：" << files[i] << endl;

OpenMPFunc(files[i], openmpsumtime);

cout << endl;

}

cout << "总计 OpenMP 执行时间: " << openmpsumtime << " s" << endl;

FILE\* fp = fopen("time.txt", "a");

if (fp)

{

fprintf(fp, "线程数：%d，OpenMP总时间：%.10lf s\n", thread\_num, openmpsumtime);

fclose(fp);

}

return 0;

}

MPI并行代码

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <mpi.h>

#include <opencv2/opencv.hpp>

#include <iostream>

#include <vector>

#include <string>

#include <filesystem>

namespace fs = std::filesystem;

// 手动实现灰度转换函数

cv::Mat manualBGR2Gray(const cv::Mat& img) {

CV\_Assert(img.type() == CV\_8UC3);

cv::Mat gray(img.rows, img.cols, CV\_8UC1);

for (int y = 0; y < img.rows; ++y) {

for (int x = 0; x < img.cols; ++x) {

cv::Vec3b pixel = img.at<cv::Vec3b>(y, x);

float grayValue = 0.299f \* pixel[2] +

0.587f \* pixel[1] +

0.114f \* pixel[0];

gray.at<uchar>(y, x) = static\_cast<uchar>(grayValue);

}

}

return gray;

}

int main(int argc, char\* argv[]) {

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// 设置输入和输出文件夹路径

std::string inputFolderPath = "D:/MPI/Picture";

std::string outputFolderPath = "D:/MPI/GrayPicture";

// 检查输入路径是否存在

if (!fs::exists(inputFolderPath)) {

std::cerr << "进程 " << rank << ": 错误 - 文件夹未找到: " << inputFolderPath << std::endl;

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

// 主进程收集图像路径

std::vector<std::string> imagePaths;

if (rank == 0) {

for (const auto& entry : fs::directory\_iterator(inputFolderPath)) {

if (entry.is\_regular\_file()) {

imagePaths.push\_back(entry.path().string());

}

}

std::cout << "找到总图片数: " << imagePaths.size() << std::endl;

}

// 广播图像总数

int totalImages = imagePaths.size();

MPI\_Bcast(&totalImages, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// 广播图像路径

const int MAX\_PATH\_LEN = 260;

char\* allPaths = new char[totalImages \* MAX\_PATH\_LEN];

if (rank == 0) {

for (int i = 0; i < totalImages; ++i) {

strncpy(&allPaths[i \* MAX\_PATH\_LEN], imagePaths[i].c\_str(), MAX\_PATH\_LEN);

}

}

MPI\_Bcast(allPaths, totalImages \* MAX\_PATH\_LEN, MPI\_CHAR, 0, MPI\_COMM\_WORLD);

//块分配逻辑

int chunk\_size = totalImages / size;

int remainder = totalImages % size;

int start, end;

if (rank < remainder) {

start = rank \* (chunk\_size + 1);

end = start + chunk\_size + 1;

}

else {

start = rank \* chunk\_size + remainder;

end = start + chunk\_size;

}

end = std::min(end, totalImages);

// 提取本进程分配到的路径

std::vector<std::string> localPaths;

for (int i = start; i < end; ++i) {

localPaths.emplace\_back(&allPaths[i \* MAX\_PATH\_LEN]);

}

delete[] allPaths;

std::cout << "进程 " << rank << ": 分配到 " << localPaths.size()

<< " 张图片 (块范围: " << start << "-" << end - 1 << ")" << std::endl;

// 确保输出目录存在

if (!fs::exists(outputFolderPath)) {

fs::create\_directories(outputFolderPath);

}

// 同步所有进程并开始计时

MPI\_Barrier(MPI\_COMM\_WORLD);

double startWallTime = MPI\_Wtime();

// 图像处理

for (int imgIdx = 0; imgIdx < localPaths.size(); ++imgIdx) {

const auto& path = localPaths[imgIdx];

std::cout << "进程 " << rank << ": 正在处理图片 " << imgIdx + 1

<< " / " << localPaths.size() << ": " << path << std::endl;

cv::Mat img = cv::imread(path);

if (img.empty()) {

std::cerr << "进程 " << rank << ": 读取图片失败: " << path << std::endl;

continue;

}

// 使用手动实现的灰度转换

cv::Mat gray = manualBGR2Gray(img);

fs::path p(path);

std::string outputName = "processed\_rank" + std::to\_string(rank) + "\_" + p.stem().string() + ".png";

std::string outputPath = outputFolderPath + "/" + outputName;

std::vector<int> compression\_params;

compression\_params.push\_back(cv::IMWRITE\_PNG\_COMPRESSION);

compression\_params.push\_back(0); // 设置为无压缩

if (!cv::imwrite(outputPath, gray, compression\_params)) {

std::cerr << "进程 " << rank << ": 保存图片失败: " << outputName << std::endl;

}

}

// 同步后结束计时

MPI\_Barrier(MPI\_COMM\_WORLD);

double endWallTime = MPI\_Wtime();

double localElapsed = endWallTime - startWallTime;

// 汇总最大时间（实际运行时间）

double globalElapsed;

MPI\_Reduce(&localElapsed, &globalElapsed, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);

// 主进程输出结果

if (rank == 0) {

std::cout << "配置信息:" << std::endl;

std::cout << " - 总图片数: " << totalImages << std::endl;

std::cout << " - 进程数: " << size << std::endl;

std::cout << " - 实际运行时间: " << globalElapsed << " 秒" << std::endl;

}

MPI\_Finalize();

return 0;

}

CUDA并行代码

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <iostream>

#include <string>

#include <cassert>

#include <vector>

#include <math.h>

#include <time.h>

#include <io.h>

#include <chrono>

#include <opencv2/core/core.hpp>

#include <opencv2/highgui/highgui.hpp>

#include <opencv2/opencv.hpp>

#include "opencv2/highgui.hpp"

#include "opencv2/imgcodecs/legacy/constants\_c.h"

#include <cuda.h>

#include <cuda\_runtime.h>

#include <cuda\_runtime\_api.h>

using namespace cv;

using namespace std;

#define THREAD\_NUM 32

//输入图像为BGR图，将其转化为gray图

\_\_global\_\_ void rgb2grayInCuda(uchar3\* dataIn, unsigned char\* dataOut, int imgHeight, int imgWidth)

{

//图片二维扫描，分别有x方向，y方向的像素点

int xIndex = threadIdx.x + blockIdx.x \* blockDim.x; //表示x方向上的ID

int yIndex = threadIdx.y + blockIdx.y \* blockDim.y; //表示y方向上的ID

//灰度变换操作

if (xIndex < imgWidth && yIndex < imgHeight)

{

uchar3 rgb = dataIn[yIndex \* imgWidth + xIndex];

dataOut[yIndex \* imgWidth + xIndex] = 0.299f \* rgb.x + 0.587f \* rgb.y + 0.114f \* rgb.z;

}

}

//串行转换灰度图像

void rgb2grayincpu(unsigned char\* const d\_in, unsigned char\* const d\_out, uint imgheight, uint imgwidth)

{

//使用两重循环嵌套实现x方向 y方向的变换

for (int i = 0; i < imgheight; i++)

{

for (int j = 0; j < imgwidth; j++)

{

d\_out[i \* imgwidth + j] = 0.299f \* d\_in[(i \* imgwidth + j) \* 3]

+ 0.587f \* d\_in[(i \* imgwidth + j) \* 3 + 1]

+ 0.114f \* d\_in[(i \* imgwidth + j) \* 3 + 2];

}

}

}

int CUDAfunc(string inputfilename, double& gpusumtime, double& cpusumtime) {

//传入图片

Mat srcImg = imread(inputfilename);

FILE\* fp;//创建运行时间文件

//读取图片像素值

int imgHeight = srcImg.rows;

int imgWidth = srcImg.cols;

Mat grayImg(imgHeight, imgWidth, CV\_8UC1, Scalar(0)); //输出灰度图

//在GPU中开辟输入输出空间

uchar3\* d\_in;

unsigned char\* d\_out;

int\* d\_hist;

//分配内存空间

cudaMalloc((void\*\*)&d\_in, imgHeight \* imgWidth \* sizeof(uchar3));

cudaMalloc((void\*\*)&d\_out, imgHeight \* imgWidth \* sizeof(unsigned char));

cudaMalloc((void\*\*)&d\_hist, 256 \* sizeof(int));

//将图像数据传入GPU中

cudaMemcpy(d\_in, srcImg.data, imgHeight \* imgWidth \* sizeof(uchar3), cudaMemcpyHostToDevice);

dim3 threadsPerBlock(THREAD\_NUM, THREAD\_NUM);

dim3 blocksPerGrid((imgWidth + threadsPerBlock.x - 1) / threadsPerBlock.x, (imgHeight + threadsPerBlock.y - 1) / threadsPerBlock.y);

//cuda灰度化

//计时开始

auto gpustart = chrono::system\_clock::now();

//调用核函数

rgb2grayInCuda << <blocksPerGrid, threadsPerBlock >> > (d\_in, d\_out, imgHeight, imgWidth);

//同步CPU和gpu，否则测速结果为cpu启动内核函数的速度

cudaDeviceSynchronize();

//计时结束

auto gpuend = chrono::system\_clock::now();

//计算时间差

auto gpuduration = chrono::duration\_cast<chrono::microseconds>(gpuend - gpustart);

double gput = gpuduration.count();

//微秒转化为秒

double gputime = gput / 1000000;

gpusumtime += gputime;

//打印cuda并行执行时间

cout << setiosflags(ios::fixed) << setprecision(10) << "CUDA执行时间： " << gputime << " s" << endl;

//将数据从GPU传回CPU

cudaMemcpy(grayImg.data, d\_out, imgHeight \* imgWidth \* sizeof(unsigned char), cudaMemcpyDeviceToHost);

vector<int> compression\_params;

compression\_params.push\_back(CV\_IMWRITE\_PNG\_COMPRESSION);

compression\_params.push\_back(0);

//释放内存

cudaFree(d\_in);

cudaFree(d\_out);

cudaFree(d\_hist);

/\*CPU串行开始\*/

//串行灰度化

//计时开始

auto cpustart = chrono::system\_clock::now();

//调用主函数

rgb2grayincpu(srcImg.data, grayImg.data, imgHeight, imgWidth);

//计时结束

auto cpuend = chrono::system\_clock::now();

//计算时间差

auto cpuduration = chrono::duration\_cast<chrono::microseconds>(cpuend - cpustart);

double cput = cpuduration.count();

//微秒转化为秒

double cputime = cput / 1000000;

cpusumtime += cputime;

//打印串行执行时间

cout << setiosflags(ios::fixed) << setprecision(10) << "CPU执行时间： " << cputime << " s" << endl;

/\*输出灰度图片\*/

try

{

int len = inputfilename.length();

cout << "inputfilename.length:" << len << endl;

string str = "./GrayPicture/";

imwrite(str + inputfilename.substr(10, len - 14) + "\_to\_gray.png", grayImg, compression\_params);

cout << str + inputfilename.substr(10, len - 14) + "\_to\_gray.png" << endl;

//在GrayPicture文件夹中，生成灰度变换后的结果图片

}

catch (runtime\_error& ex)

{

fprintf(stderr, "图像转换成PNG格式发生错误：%s\n", ex.what());

return 1;

}

return 0;

}

//批量读取图片

void getFiles(string path, vector<string>& files)

{

//文件句柄

intptr\_t hFile = 0;

//文件信息

struct \_finddata\_t fileinfo;

string p;

if ((hFile = \_findfirst(p.assign(path).append("\\\*").c\_str(), &fileinfo)) != -1)

{

do

{

//如果是目录,迭代之

//如果不是,加入列表

if ((fileinfo.attrib & \_A\_SUBDIR))

{

if (strcmp(fileinfo.name, ".") != 0 && strcmp(fileinfo.name, "..") != 0)

getFiles(p.assign(path).append("\\").append(fileinfo.name), files);

}

else

{

files.push\_back(p.assign(path).append("\\").append(fileinfo.name));

}

} while (\_findnext(hFile, &fileinfo) == 0);

\_findclose(hFile);

}

}

int main()

{

//图片文件路径，在项目文件下的Picture文件夹里面

string filePath = "./Picture";

vector<string> files;

//读取图片文件

getFiles(filePath, files);

//读取图片数量

int size = files.size();

//输出图片数量

cout << "图片数量：" << size << endl;

double gpusumtime = 0, cpusumtime = 0;

for (int i = 0; i < size; i++)

{

cout << "第 " << i + 1 << "/" << size << " 张图片" << endl;

cout << files[i].c\_str() << endl;

CUDAfunc(files[i].c\_str(), gpusumtime, cpusumtime);

cout << endl;

}

cout << "总GPU时间：" << gpusumtime << " s" << "\n" << "总CPU时间：" << cpusumtime << " s" << endl;

FILE\* fp;

fp = fopen("time.txt", "a");

fprintf(fp, "总CPU时间： %.10lf s ,总GPU时间： %.10lf s \n", cpusumtime, gpusumtime);

fclose(fp);

return 0;

}

OpenMP+MPI混合并行代码

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <mpi.h>

#include <opencv2/opencv.hpp>

#include <iostream>

#include <vector>

#include <string>

#include <filesystem>

#include <omp.h>

namespace fs = std::filesystem;

// OpenMP优化的灰度转换函数

void rgb2grayOpenMP(unsigned char\* const d\_in, unsigned char\* const d\_out, int imgHeight, int imgWidth) {

#pragma omp parallel for collapse(2)

for (int i = 0; i < imgHeight; ++i) {

for (int j = 0; j < imgWidth; ++j) {

int idx = i \* imgWidth + j;

d\_out[idx] = static\_cast<unsigned char>(

0.299f \* d\_in[idx \* 3] +

0.587f \* d\_in[idx \* 3 + 1] +

0.114f \* d\_in[idx \* 3 + 2]);

}

}

}

int main(int argc, char\* argv[]) {

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// 设置输入和输出文件夹路径

const std::string inputFolderPath = "D:/muti/x64/Debug/Picture";

const std::string outputFolderPath = "D:/muti/x64/Debug/GrayPicture";

// 检查输入路径

if (!fs::exists(inputFolderPath)) {

std::cerr << "进程 " << rank << ": 错误 - 文件夹未找到: " << inputFolderPath << std::endl;

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

// 主进程收集图像路径

std::vector<std::string> imagePaths;

if (rank == 0) {

for (const auto& entry : fs::directory\_iterator(inputFolderPath)) {

if (entry.is\_regular\_file()) {

imagePaths.push\_back(entry.path().string());

}

}

std::cout << "找到总图片数: " << imagePaths.size() << std::endl;

}

// 广播图像总数

int totalImages = 0;

if (rank == 0) totalImages = imagePaths.size();

MPI\_Bcast(&totalImages, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// 广播路径数据

const int MAX\_PATH\_LEN = 260;

char\* allPaths = new char[totalImages \* MAX\_PATH\_LEN]();

if (rank == 0) {

for (int i = 0; i < totalImages; ++i) {

strncpy(&allPaths[i \* MAX\_PATH\_LEN], imagePaths[i].c\_str(), MAX\_PATH\_LEN - 1);

}

}

MPI\_Bcast(allPaths, totalImages \* MAX\_PATH\_LEN, MPI\_CHAR, 0, MPI\_COMM\_WORLD);

// 任务分配

int chunk = totalImages / size;

int remainder = totalImages % size;

int start = rank \* chunk + std::min(rank, remainder);

int end = start + chunk + (rank < remainder ? 1 : 0);

end = std::min(end, totalImages);

// 提取本地任务

std::vector<std::string> localPaths;

for (int i = start; i < end; ++i) {

localPaths.emplace\_back(&allPaths[i \* MAX\_PATH\_LEN]);

}

delete[] allPaths;

// 创建输出目录

if (rank == 0 && !fs::exists(outputFolderPath)) {

fs::create\_directories(outputFolderPath);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

// 设置OpenMP线程数（可通过命令行参数控制）

int omp\_threads = 2; // 默认2线程

if (argc > 1) omp\_threads = atoi(argv[1]);

omp\_set\_num\_threads(omp\_threads);

// 同步计时

MPI\_Barrier(MPI\_COMM\_WORLD);

double startTime = MPI\_Wtime();

// 处理本地图片（MPI进程间并行 + OpenMP线程级并行）

for (const auto& path : localPaths) {

cv::Mat srcImg = cv::imread(path);

if (srcImg.empty()) {

std::cerr << "进程 " << rank << ": 读取失败: " << path << std::endl;

continue;

}

// 准备灰度图像

cv::Mat grayImg(srcImg.rows, srcImg.cols, CV\_8UC1);

// 使用混合并行处理

rgb2grayOpenMP(srcImg.data, grayImg.data, srcImg.rows, srcImg.cols);

// 保存结果

fs::path p(path);

std::string outputName = outputFolderPath + "/hybrid\_rank" +

std::to\_string(rank) + "\_" + p.filename().string();

std::vector<int> compression\_params;

compression\_params.push\_back(cv::IMWRITE\_PNG\_COMPRESSION); // PNG压缩参数

compression\_params.push\_back(0);

// 保存图像时传入压缩参数

if (!cv::imwrite(outputName, grayImg, compression\_params)) {

std::cerr << "进程 " << rank << ": 保存失败: " << outputName << std::endl;

}

}

// 计算总时间

MPI\_Barrier(MPI\_COMM\_WORLD);

double totalTime = MPI\_Wtime() - startTime;

// 汇总统计信息

if (rank == 0) {

std::cout

<< "总图片数: " << totalImages << "\n"

<< "MPI进程数: " << size << "\n"

<< "OpenMPOpenMP线程/进程: " << omp\_threads << "\n"

<< "总耗时: " << totalTime << " 秒\n";

}

MPI\_Finalize();

return 0;

}