# OpenMP实现卷积

## 1. 实验内容

本实验要求我们实现卷积操作，并且通过OpenMP编程来实现并行化。

## 2. 实验目的

1）掌握C++对图片的读取和写入技术

2）掌握通过C++编程实现对图片的卷积

3）熟悉使用OpenMP并行化来提高计算效率，降低时间成本

3. 实验环境

Windows操作系统下的opencv环境，vscode

## 4. 实验步骤

### 4.1搭建OpenMP运行环境

1）导入库<omp.h>

2）配置c++的opencv环境

### 4.2 创建源代码

#include <iostream>

#include <omp.h>

#define W 1920

#define H 1080

using namespace std;

int Limit(int num)

{

    if (num < 0)

        num = 0;

    else if (num > 255)

        num = 255;

    return num;

}

float imageKernel[3][3] = {0.111,0.111,0.111,0.111,0.111,0.111,0.111,0.111,0.111};

int main()

{   char \*name[3] = {"1.yuv","2.yuv","3.yuv"};

    char \*tar[3] = {"conv\_1.yuv","conv\_2.yuv","conv\_3.yuv"};

    FILE \*fp[3];

    double t1 = omp\_get\_wtime();

    #pragma omp parallel for

    for(int i = 0; i < 2; i++){

        fp[i] = fopen(name[i], "rb");

    }

    for(int i = 0; i < 2; i++){

        fseek(fp[i], 0, SEEK\_END);

        int len = ftell(fp[i]);

        fseek(fp[i], 0, SEEK\_SET);

        unsigned char\* inputImage = new unsigned char[len]();

        unsigned char\* outputImage = new unsigned char[len]();

        fread(inputImage, 1, len, fp[i]);

        fclose(fp[i]);

        memcpy(outputImage, inputImage, len);

        #pragma omp parallel for

        for (int i = 1; i < H - 1; i++)

        {

            // int thread\_num = omp\_get\_thread\_num();

            // cout << "thread num: " << thread\_num << endl;

            for (int j = 1; j < W - 1; j++)

            {

                float sum = 0;

                for(int a = -1; a < 2; a++){

                    for(int b = -1; b < 2; b++){

                        sum += inputImage[(i+a)\*W+j+b]\*imageKernel[a+1][b+1];

                    }

                }

                int num = Limit(int(sum));

                outputImage[i\*W + j] = num;

            }

        }

        fp[i] = fopen(tar[i], "wb+");

        fwrite(outputImage, 1, len, fp[i]);

        fclose(fp[i]);

    }

    double t2 = omp\_get\_wtime();

    cout << t2-t1 << endl;

    system("pause");

    return 0;

}

### 4.3 进行编译

g++ -fopenmp conv.cpp

## 5. 实验分析

运行结果数据：

|  |  |  |  |
| --- | --- | --- | --- |
| 处理数量 | 1 | 2 | 3 |
| 串行 | 0.072 | 0.144 | 0.215 |
| 并行 | 0.016 | 0.032 | 0.048 |

运行结果可视化：

根据对处理图片数量以及是否使用OpenMP实现并行化的设置，运行程序得到程序运行的时间，再将获得的数据进行综合分析，最后得到如上所示的表格和折线图。根据实验结果，我们可以发现并行所用的时间显著小于串行所用时间，并且随着处理数量的增大，两者所用时间的差异越来越大，因此在实际的大规模化生产中，并行化能极大地提升程序运行的效率，节约时间成本。

## 6. 问题与讨论

1）最初我打算搭建c++的opencv环境，使用opencv中的imread()和imwrite()来实现图片的读取和写入，但是最终在加入并行化时出现了编译错误，在花费了大量时间无法解决后，我最终使用c++读写文件的形式来读取和写入图片。

2）最初我使用clock()来计算运行时间，但是发现计算所得串并行时间相差不大，最终使用<omp.h>里的omp\_get\_wtime()解决了问题。

3）关于openmp，并行计算机可以简单分为共享内存和分布式内存，共享内存就是多个核心共享一个内存，不管是只有一个多核CPU还是可以插多个CPU，它们都有多个核心和一个内存，一般的大型计算机结合分布式内存和共享内存结构，即每个计算节点内是共享内存，节点间是分布式内存。想要在这些并行计算机上获得较好的性能，进行并行编程是必要条件，而在我处理的这个项目中，采用的是openmp来共享内存结构。

4）openmp与mpi的比较： openmp是基于内存共享的并行处理机制，而mpi是基于消息传递的并行处理机制。openmp代码编写比较简单，只需要特定的编译引导语句，就可以自动的分解多线程执行，而mpi较为复杂，需要考虑到进程间的相互通信。 openmp是线程级的并行编程技术，mpi是进程级的并行编程技术，两者有各自优点和缺点，进行程序的最优化处理时应考虑机器的特性，openmp由于是共享存储，可扩展性较差，不适合集群，mpi扩展性好，适合各种机器，但较为复杂，调试麻烦。

## 参考资料：

（1）http://www.openmp.org