并行计算 II 2023 春季第三次作业：优化 Gaussian Blur 算子

1900017702 朱越

0. 概要

本作业选择使用单纯OpenMP技术对Gaussian Blur算子进行优化，最终总时长约**0.06s**，加速比约**89倍。**

1. Gaussian Blur算子

使用3*3卷积核

$$\frac{1}{16}\begin{pmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{pmatrix}$$

对N*M矩阵作卷积，不进行padding操作，得到(N-2)*(M-2)矩阵

2. 串行计算

源码见gaussian_blur.cpp，核心部分如下：

```
for (int j = 1; j < width - 1; ++j)
{
    for (int i = 1; i < height - 1; ++i)
    {
        double pixel = 0.0;
        for (int l = -1; l <= 1; ++l)
            for (int k = -1; k <= 1; ++k)
                pixel += image[i + k][j + l] * KERNEL[k + 1][l + 1];
        blurredImage[i][j] = pixel / 16.0;
    }
}
```

总运行时间大约是5.4s

3. 思考题

- Gaussian blur是属于运算密集型还是访存密集型应用？为什么？
- 答：是访存密集型应用。为了分析该问题，需要作这样的假设：w和h很大，且不考虑数据时空局部性。在一次迭代中，有9次计算（8加1乘），10次访存，计算密度AI=9/10。即使考虑了数据时空局部性，总体的访存次数和计算次数也至少是w*h数量级的，故计算密度总是一个非零常数。例：在滑动窗口算法中，访存次数可以减少到3wh，而计算次数可以减少到6wh（4加2乘），AI=2。

4. 并行计算

1. 编译优化
    - 使用-Ofast编译优化选项，平均用时减少到2.4s
2. 节点与核心，进程与线程

> 先考虑使用OpenMP作多线程并行，这在编程上最简单。测试发现，单节点16线程运行时间只有约0.5s，而且是在十张图片上的总时长，平均每张图片上的用时只有50ms。在这么多节点上的同步反而更浪费时间（总时长约1s），所以本次任务尽量使用单节点进行。
>
> 集群使用的物理节点是双路12核的E5-2650 v4，如果在一个节点上申请超过12个核心，就不得不分配到两个插槽上，不同插槽的共享内存访问比同一插槽的慢，这造成的性能损失比更多核心带来的计算负载减小更显著。因此，本次任务使用12个CPU核心，可以加速到约0.4s。
>
> 在该问题中，恰当的任务划分可以使得各工作流之间不存在除RAR外的数据冲突，只需要对同一片共享内存的不同位置各自写入。多进程并行由于内存隔离性好而不适合这种情况（需要作显式的通信同步），而多线程并行则正好对应这样共享内存写的情况。因此，本次任务使用单进程多线程并行。

- 使用单节点12核心作单进程多线程并行，可以加速到约0.4s

3. 遍历顺序初步

> 对两重循环遍历i和j作工作分配时，可以选择遍历的顺序和划分方式。
>
> 首先，应该对外层循环作并行划分。划分外层和内层的区别在于所开辟的并行区是持久的还是临时的，临时的并行区需要每次初始化，而持久化的并行区则可以复用这部分重复工作。
>
> 其次，输入和输出矩阵都是按行存储的。按行连续访问可以利用缓存的空间局部性，因此按列遍历放在外层，按行遍历放在内层的配置可以最大程度地利用空间局部性。

- 使用i-j的遍历顺序，并对外层i作并行划分，可以加速到约0.07s

4. 循环展开

    1. 核展开

    > 由于本问题中的Gaussian核只有3*3大小，因此考虑直接把要累加的9项展开。实际测得并无明显效果，可能是编译器已经自动对定长且较短的循环作了展开。进一步在无编译优化选项的条件下进行测试，发现循环展开确实会带来效果。这证实了编译优化会作循环展开。
    >
    > 另一方面，我尝试了直接把Gaussian核的数值写入到循环中，试图减少访存消耗的时间，然而这并没有带来任何改变。这是由于开启编译优化的情况下，编译器会尽量把在编译时刻就能确定的计算提前在编译阶段完成，本问题中的对于常量矩阵KERNEL的元素取值就可以在编译阶段直接确定。
    >
    > 此外，核的大小和数值未必总是题目中指定的，上述两项尝试降低了可扩展性。

    2. 列展开

    > 在按列遍历时展开循环，或许可以减少跳转次数，并更精细化地利用到空间局部性。然而实测表明这种展开方式并不是一个理想的方向，其可能原因如下：如果不展开核所在的循环，单次循环工作量已经比较大，跳转带来的流水线损失不大，没有显著效果；如果展开核所在的循环（核展开），可以进一步利用数据的时间局部性，但反而造成了寄存器满溢，速度变慢。

    3. 行展开

    > 按行展开的情况比较特殊，现有的优化工作是建立在按行划分上的，因此展开已划分的循环相当于修改了划分。在下一小节中详细讨论划分情况。

- 在这一部分中，并没有取得任何进展

5. 工作划分

> 截止到现在，只改变了循环的遍历顺序i-j，并对行指标i作OpenMP支持的自动划分。因此有两个优化可能性：调节划分方式（手动配置并行区）；调节循环顺序（核循环外移）。

1. 划分方式

> 最常见的两种调度方式：Round-Robin和range。测试发现range的表现略微好点，这是符合直觉的：因为在换行时跳转到下一行能利用到空间局部性，而远距离的跳转则不可以。

2. 循环顺序

> 我想到了一种激进的策略：调换遍历循环和核循环的顺序，把核循环外移。即对每个结果元素，把Gaussian核切成三份，每次计算一行。

- 好处：每次只使用同一行的内存，避免了可能存在的缓存抖动（假设机器是直接映射缓存，如果图片一行的长度正好是该级缓存的大小，那么同时引用图片两行的同一列就会造成缓存的冲突失效）。
- 坏处：增加了写的次数。原先的策略在一次读入9个元素计算后总共进行一次写，现在的策略则是在读入3个元素后进行一次写，重复三次共进行三次写。

```
for (int i = begin; i < end; i++)
{
        for (int l = -1; l <= 1; ++l)       // 核循环
        {
    for (int j = 1; j < width - 1; ++j)     // 列循环
    {
        double pixel = 0.0;
            for (int k = -1; k <= 1; ++k)
            {
                pixel += image[i + k][j + l] * KERNEL[k + 1][l + 1];
            }
        blurredImage[i][j] += pixel / 16.0;
    }
        }
}
```

- 经过测试，核循环外移的策略时间要慢些。一方面，现代处理器的缓存相连度比较高，和直接映射的假设不符，针对Gaussian Blur这样一次引用三行的算子组相连缓存绰绰有余；另一方面，增加写的次数造成了性能损失。此外，在该尝试中需要预先把blurredImage矩阵初始化为0，这一步骤略微减少了时长，这和First Touch原则有关。

3. 遍历方式

> 对于二维矩阵的二重循环遍历，最简单的方式就是两个指标依次递增。尝试了一下“折线型遍历”（即在一行结束时移到正下方而不是从下一行的头部开始），或许可以在换行时复用一部分已读入的元素，减少缓存冲突的可能性，但测试发现没有什么用。。。

- 这一部分中，使用range的调度策略能略微优化，最终优化到平均用时约0.06s。

6. 滑动窗口

> 使用滑动窗口策略可以显式地利用数据的空间局部性，可能减少访存次数。并且由于核系数的特殊性，可以复用不同列（只需*2或/2），减少计算次数。
>
> 但在实际测试中发现并无显著效果。这也验证了原先的按行策略中已经隐式地利用了空间局部性，滑动窗口相比并没有减少访存次数。而由于计算密度较低，计算次数不是表现的瓶颈。

- 该策略没有能够进一步优化。

7. SIMD

> 使用编译选项-march=native和OpenMP的simd语句作SIMD化，发现效果并不明显。SIMD在硬件上只能对计算作并行，不能对访存作并行，对于访存密集型问题，SIMD的作用不大。

5. 最终方案

本次作业的最终方案源码见gaussian_blur_try.cpp，选择了如下配置：

- 在单节点的多核处理器E5-2650 v4（在作业脚本中使用lscpu > lscpu.out）上使用12个CPU
- 单纯OpenMP多线程编程技术，以单进程方式开启12个线程并行
- 编译器使用gcc 4.8.5（在登录节点上使用g++ -v &>>g++v.out）
- 开启编译优化选项-Ofast
- 循环方式为i-j顺序，外层遍历行指标i，内层遍历列指标j
- 划分方法为range调度按行划分，即每个线程负责矩阵的一段连续行的元素计算，占总体计算工作的约1/12

最终用时约0.06s，加速比约89倍，一次的运行结果如下（见job_217578_cu04.out）：

```
Opened input file: images/potm2209a.txt
Execution time: 104 ms
1.00691e+08
Opened input file: images/potm2209b.txt
Execution time: 650 ms
7.02887e+08
Opened input file: images/potm2209c.txt
Execution time: 653 ms
6.60097e+08
Opened input file: images/potm2210a.txt
Execution time: 197 ms
8.6829e+07
Opened input file: images/potm2210c.txt
Execution time: 162 ms
5.73649e+07
Opened input file: images/potm2210d.txt
Execution time: 161 ms
8.24852e+07
Opened input file: images/potm2211a.txt
Execution time: 659 ms
1.34824e+08
Opened input file: images/potm2211b.txt
Execution time: 857 ms
1.50042e+08
Opened input file: images/potm2211c.txt
Execution time: 859 ms
1.53024e+08
Opened input file: images/potm2212a.txt
Execution time: 1155 ms
6.08503e+08
Total time: 5.4570000000 s
Opened input file: images/potm2209a.txt
Execution time: 1 ms
1.00691e+08
Opened input file: images/potm2209b.txt
Execution time: 6 ms
7.02887e+08
Opened input file: images/potm2209c.txt
Execution time: 8 ms
6.60097e+08
```

```
Opened input file: images/potm2210a.txt
Execution time: 2 ms
8.6829e+07
Opened input file: images/potm2210c.txt
Execution time: 1 ms
5.73649e+07
Opened input file: images/potm2210d.txt
Execution time: 1 ms
8.24852e+07
Opened input file: images/potm2211a.txt
Execution time: 9 ms
1.34824e+08
Opened input file: images/potm2211b.txt
Execution time: 8 ms
1.50042e+08
Opened input file: images/potm2211c.txt
Execution time: 12 ms
1.53024e+08
Opened input file: images/potm2212a.txt
Execution time: 13 ms
6.08503e+08
Total time: 0.0610000000 s
```

6. 结果复现

编译

```
make clean
make
```

测试

```
sbatch run.slurm
```