

# seam carving 报告

李晨昊 2017011466

2019-5-26

## 目录

1 概述	2
2 算法细节	3
2.1 缩小	3
2.2 放大	5
2.3 区域保护和删除	5
2.4 多线程加速	7

# 1 概述

本项目实现了去年的三个图形处理作业之一的 seam carving，关于这个作业内容，我已经提前向助教申请过了。

我完成了基础要求：图像双向缩小，以及提高要求：区域保护和删除 + 放大。计算能量矩阵所用到的算子，我实现了 Sobel 算子，Laplace 算子和 Roberts Cross 算子，经测试三者的区别不大，不过多数情况下 Laplace 算子还是略有优势，因此默认的编译选项采用 Laplace 算子。

我使用了lodepng和cmdline这两个第三方库，分别用于 png 文件的读写和命令行参数的处理，它们的代码都已经附在本项目中，无需额外安装依赖。

我使用了 openmp 来进行并行优化，具体细节可参见最后一个节。相比于光线追踪大作业，对于这个算法实现并行优化难度要大一些，最终的并行效率也要差一些，但是还是能够有明显的加速的。

linux 环境下直接 make 即可编译程序。为了程序效率，我用到了一些 linux 的系统调用，因此保证不兼容 windows。编译选项中对于 main.cpp 的编译有两个宏定义，分别是用来控制采用哪种算子和是否输出带有 seam 标记的图片信息。这些东西之所以做成了编译参数而不是运行时决定，主要是为了减少没必要的判断，提高一点效率（当然，其实都没太大区别，只是个人写代码的习惯罢了）。

运行编译好的程序，会显示以下帮助信息：

```
[mashplant@mashplant seam_carving]$ ./seam_carving
usage: ./seam_carving --type=string --input=string --output=string --seam=string [options] ...
options:
  -t, --type      which process to exec[carve, extend, remove, protect] (string)
  -i, --input     path of input .png file (string)
  -o, --output    path of output .png file (string)
  -s, --seam      path of output seam info .png file (string)
  -w, --w         the factor on w axis (float [=1])
  -h, --h         the factor on h axis (float [=1])
  -?, --help      print this message
```

我没有怎么考虑代码复用性，所以写的有点冗余，基本上是为每个功能都写了一个独立的函数。但是我相当注重代码的运行速度，例如需要用到二维数组的地方，我都使用了一维数组 + 计算偏移量的方法来代替；为了减少计算能量图和动态规划中的边界条件判断，很多数组我都预留了大小为 1 的 padding。基于这些原因，最后我的代码看起来可能可读性较差，不过这也算是符合我一贯的代码风格。

## 2 算法细节

### 2.1 缩小

图片的横向缩小十分简单，首先使用选定的算子为每个像素的位置都计算能量值，动态规划求出一条从上往下，能量值之和最小的道路即可。找到这条道路后，把右侧的像素平过来覆盖掉它们，这就完成了一次 seam carve。完成后，能量图中很少一部分项需要重新计算 ( $2h$  个)，但是有不少需要一起移动位置。好在 memmove 函数的效率非常高，所以图片和能量图的更新代价都不大。

经过测试，主要的时间瓶颈在于动态规划的过程，为了减少它的计算量，我将路径的计算推迟到了动态规划结束后，这个优化相当于：

$$c_{big} * n^2 + c_{small} * n \rightarrow c_{small} * n^2 + c_{big} * n$$

实测结果也表明速度明显提升了。

图片的纵向缩小只需要将图片矩阵转置后执行横向缩小，再转置回来即可。矩阵转置就是最朴素的写法，虽然效率不高，但是这显然不是性能瓶颈，所以也没必要优化。

一个例子如下：



图 1: 原图



图 2: 双向缩小 20%



图 3: 带 seam 标记

绿色的线表示被切割掉的像素，可见切割线很好地避开了图像中信息量较大的部分，例如青蛙的眼睛；而对于信息量较小的部分，例如露珠和黑色的阴影，则被大量切割。

另一个例子如下：

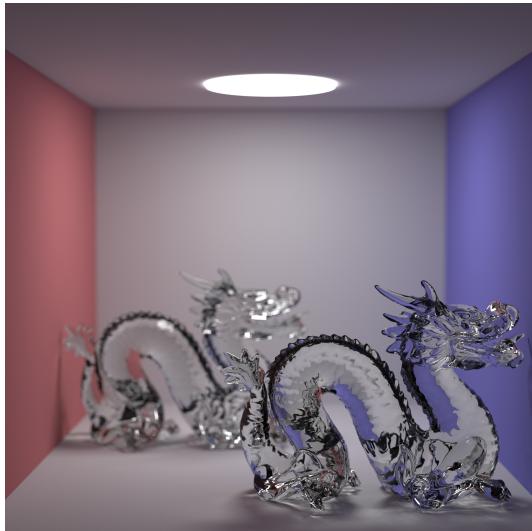


图 4: 原图



图 5: 横向缩小 20%

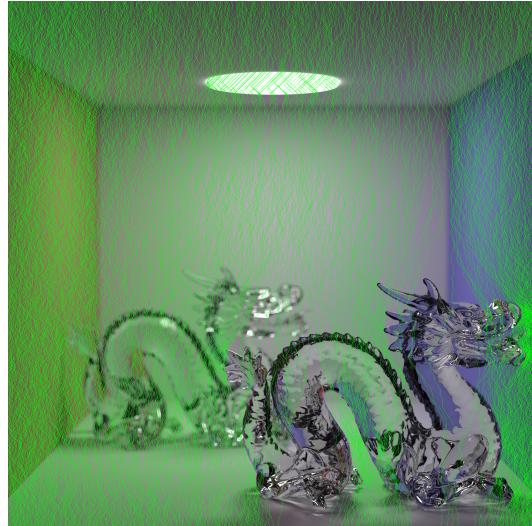


图 6: 带 seam 标记

原图来源于我的光线追踪大作业中的带焦散的龙。图片后方的龙是模糊的，因此计算出来能量值较小，被切割的更多；而图片前方的龙是其清晰的，切割线很好地避开了图像较复杂的区域。

## 2.2 放大

图片的放大是缩小的逆过程，也是计算出能量值最小的路径，然后将路径处填上左右像素颜色的平均值。然而论文中也指出，如果多次这样操作，那么能量值较小的路径倾向于集中在最初找到的那条路径上，从而不断重复同一条像素，产生严重的 artifact、解决方案是每次扩充多条能量值前  $k$  小的路径，多次执行直到完成预定的放大目标。

关于每次选择多少条路径用于扩充，需要两方面的考量：如果太少，则仍然可能会出现类似于反复扩充同一条路径的现象；如果太大，则失去了计算能量值的意义，没办法保证只扩充图像中信息量较小的部分。我选择的策略是每次扩充 15% 的像素，这是一个可调节的参数，也许对于不同的图片，最符合人的感官的比例并不相同。

具体操作时，每计算出一次能量值后进行多次动态规划，每次都选择一条路径并存储下来，将这条路径上的能量值都设为一个极大的值，表示已经选择过，之后不再使用。

一个例子如下：



图 7: 原图



图 8: 横向放大 20%



图 9: 带 seam 标记

## 2.3 区域保护和删除

通过一个辅助的 python 脚本来获取用户输入的坐标信息。理论上是可以获取任意形状的区域的，C++ 那边处理任意形状的保护和删除也不难实现，不过为了简便起见我只实现了矩形区域。

具体实现时，只需要把对应的区域内的能量值都设成一个极小/极大的值即可。由于这里我使用 32 位有符号整数来存储能量值，因此这个极小/极大值也不能太极端，否则会发生整数溢出而得到错误的结果。

删除的例子如下：



图 10: 原图



图 11: 删除人



图 12: 带 seam 标记

矩形选中了人，于是切割线优先经过人所在的矩形，不需要太多切割线就可以将人完全去除了。

保护的例子如下：



图 13: 原图



图 14: 保护塔底, 横向删除 20%



图 15: 带 seam 标记

选择的保护位置为塔底部的矩形区域。虽然看起来塔与蓝天是一个较大的变化，但是仔细看图可以发现塔下方的草地的颜色变化比其它地方少一些，因此如果正常切割，没有保护的话，会有一些分割线经过塔底部，造成一定的 artifact(从上面删除人的例子可以看出)。

## 2.4 多线程加速

显然这个图像处理的过程不像光线追踪那么简单就可以并行，最外层的循环有严格的先后关系，因此完全不能并行。内层有几个循环是与顺序无关的，包括能量图的计算和一些移动数据的循环，这些加一句 `omp parallel for` 就可以并行，效果也较理想。但是测试表明时间的瓶颈并不是它们，而是动态规划。动态规划的代码如下：

```
for (i32 i = 0; i < h; ++i) {
    u32 *p1 = dp + (i - 1) * pad_w, *p2 = dp + i * pad_w, *p3 = e_map + i * w;
    for (i32 j = 0; j < wx; ++j) {
        p2[j] = std::min(p1[j - 1], std::min(p1[j], p1[j + 1])) + p3[j];
    }
}
```

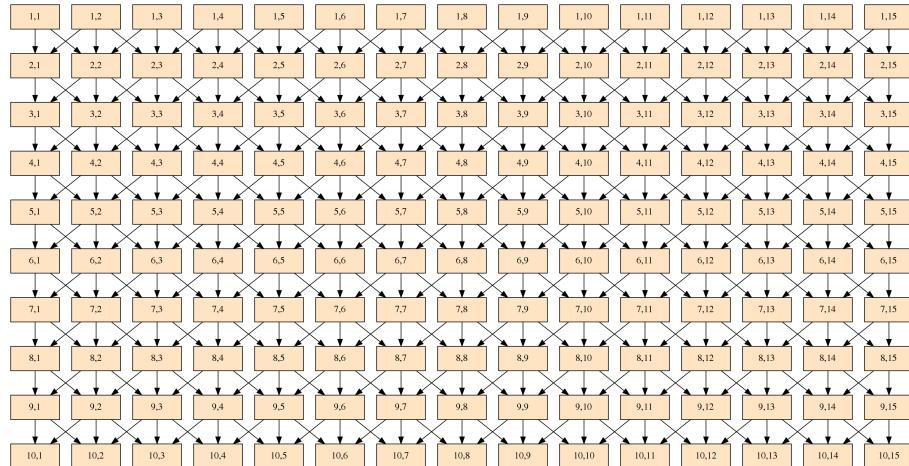
}

一个直观的想法是直接将内层循环改成并行，为了减少开启线程的开销，可以在外层循环开启 `omp parallel`，在内层循环使用 `omp for`，代码如下：

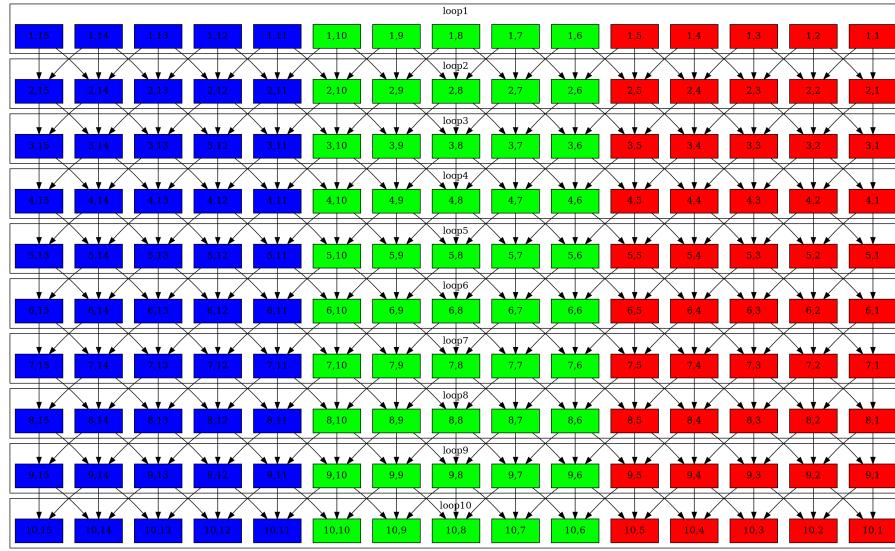
```
#pragma omp parallel
for (i32 i = 0; i < h; ++i) {
    u32 *p1 = dp + (i - 1) * pad_w, *p2 = dp + i * pad_w, *p3 = e_map + i * w;
#pragma omp for
    for (i32 j = 0; j < wx; ++j) {
        p2[j] = std::min(p1[j - 1], std::min(p1[j], p1[j + 1])) + p3[j];
    }
}
```

但是实际测试这样的效果非常糟糕，视我使用的编译器而定，速度有可能会慢一倍到几十倍。我看了一下程序的 profile 结果，发现绝大多数的时间都花在了线程同步上（而线程开启则只占很小比例）。这是容易预料到的，openmp 在每一次内层循环结束后都会加入一个 barrier 来同步所有线程，这样的计算顺序下，这个 barrier 是不可避免的。虽然一个 barrier 的开销不算大，但是 barrier 的数量是相当可观的 ( $\Delta wh$  个)。这样的结果完全不能接受。我现在采用了一种更好的计算方法。简要说明如下：

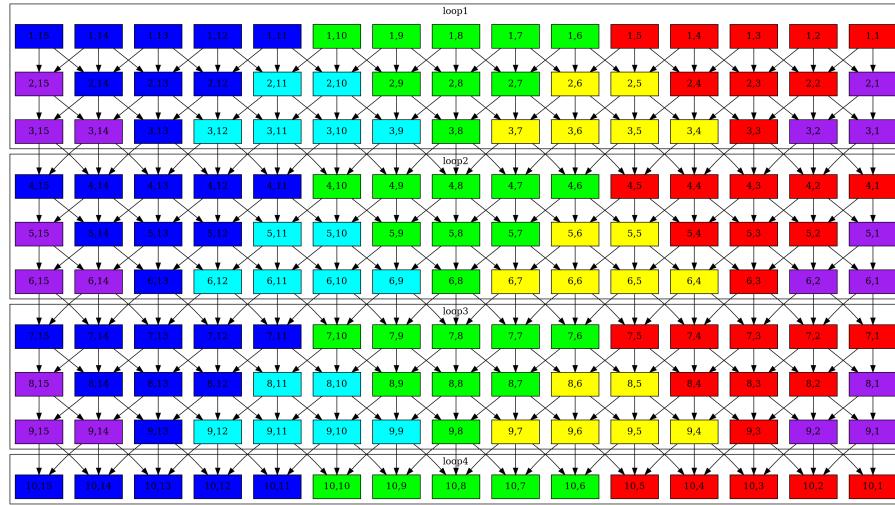
动态规划的计算过程是每个节点的计算依赖于上一层的三个节点（边界处两个）的值，用图形表示如下：



最初的这种并行方法的本质是每个线程负责一个固定的下标范围，每个内层循环处它们都有数据依赖，依赖于左边的最后一个结果和右边的第一个结果，虽然只有这两个数，但是所有的线程都得停下来等待同步。



这个例子中，这种计算顺序总计需要 10 次同步。我想出了一种同步次数更少的计算顺序，图示如下：



每次循环中，先执行红，绿，蓝三个线程，它们结束后同步一次，然后执行紫，黄，青三个线程，然后再同步一次，进入下次循环。具体的实现方法可参见代码。从图中可以看出，每个线程仅用到自身产生和本线程开始前就已经产生的数据，因此不会产生 data race。统计得，在这个例子中总计需要 7 次同步。虽然看起来差别不大，但是这里数据量很小，只是用来演示思想。分析得它的同步次数为：

$$O\left(\frac{\Delta wh}{\frac{w}{th}}\right) = O\left(\frac{th\Delta wh}{w}\right)$$

其中  $th$  为线程数。考虑到正常状态下这几个量的大小关系，这比之前的  $\Delta wh$  好了很多，并行程序的速度也终于超过了串行版本。但是同步的开销是随线程数增长的，也就是说线程数增长的时候，同步的耗时总是增加的（甚至不是保持不变的）。不过实际情况中线程数也不可能设的太大，我测试过 4 个线程，可以达到 2 左右的加速比，如果线程数继续增多，综合考虑到开启线程，线程同步，线程切换，cache miss 等因素，加速效果将明显下降，甚至有可能变慢，不过比串行版本慢几十倍这种事情再也没有发生过了。