

# 基于 CUDA 加速的改进 Mean Shift 图像分割算法

---

组长: 16337180 麦显忠

16313018 李沐晗, 16337242 韦博耀, 16337259 谢江钊, 16337179 麦金杰

2019 年 7 月 26 日

## 目录

<b>1 Introduction</b>	<b>3</b>
1.1 Mean Shift 算法概述 . . . . .	3
1.1.1 Mean Shift 滤波 . . . . .	3
1.1.2 聚类和分割 . . . . .	5
1.2 课程实验 . . . . .	6
1.2.1 Mean shift 滤波 . . . . .	7
1.2.2 聚类 (Flooding) . . . . .	7
1.2.3 区域合并 . . . . .	8
<b>2 Implementation</b>	<b>10</b>
2.1 Mean Shift 滤波的并行化 . . . . .	10
2.1.1 并行优化思路 . . . . .	10
2.1.2 算法流程 . . . . .	10
2.1.3 共享内存的读入 . . . . .	12
2.1.4 优化总结 . . . . .	14
2.1.5 加速结果 . . . . .	14

2.2	聚类 flooding 的并行化 . . . . .	15
2.2.1	并行优化思路 . . . . .	15
2.2.2	加速结果 . . . . .	16
2.3	区域合并的并行化 . . . . .	17
2.3.1	CUDA 加速 . . . . .	17
2.3.2	Pthread 加速 . . . . .	22
<b>3</b>	<b>Realtime test</b>	<b>25</b>
<b>4</b>	<b>Conclusion</b>	<b>26</b>
<b>A</b>	<b>个人报告</b>	<b>28</b>
A.1	16337180 麦显忠 . . . . .	28
A.2	16313018 李沐晗 . . . . .	28
A.3	16337242 韦博耀 . . . . .	29
A.4	16337259 谢江钊 . . . . .	29
A.5	16337179 麦金杰 . . . . .	30

# 1 Introduction

## 1.1 Mean Shift 算法概述

Mean Shift 算法用于图像分割最早是由 (Comaniciu and Meer 2002) 提出的. 算法的主要步骤可以分为三步:

1. Mean Shift 滤波
2. 将像素点聚类为区域
3. 进行标签分配

下文介绍这篇 PAMI 论文提出的使用 Mean Shift 进行图像分割的原理及具体实现.

### 1.1.1 Mean Shift 滤波

Mean Shift 滤波即 mode 搜索过程. Mean Shift 算法将图像分割视为一个聚类问题. 要进行聚类, 需要先找到聚类中心, 也就是所称的 mode, 再由此出发进行聚类的过程. 在原文中, 作者先将原图映射到特征空间, 再从这个映射得到的特征空间中搜索 mode 点.



图 1: 原图

我们参照了原文作者的方法, 将 LUV 颜色空间作为待搜索的特征空间. LUV 空间中像素点的分布如图2所示, 原始输入图像位于 RGB 空间, 使用 LUV 空间的优点为, LUV 为均匀的色度空间, 色差可用欧式距离计算.

以下我们将描述 mode 点搜索过程, 为方便讨论, 我们仅考虑三维的 LUV 特征空间的 L 和 U 维, 如图2, 图3为图2b的密度表示, 若将邻域内点的密度作为搜索聚类中心 mode 的标准, 那么图3中的红色点即我们需要搜索的聚类中心.

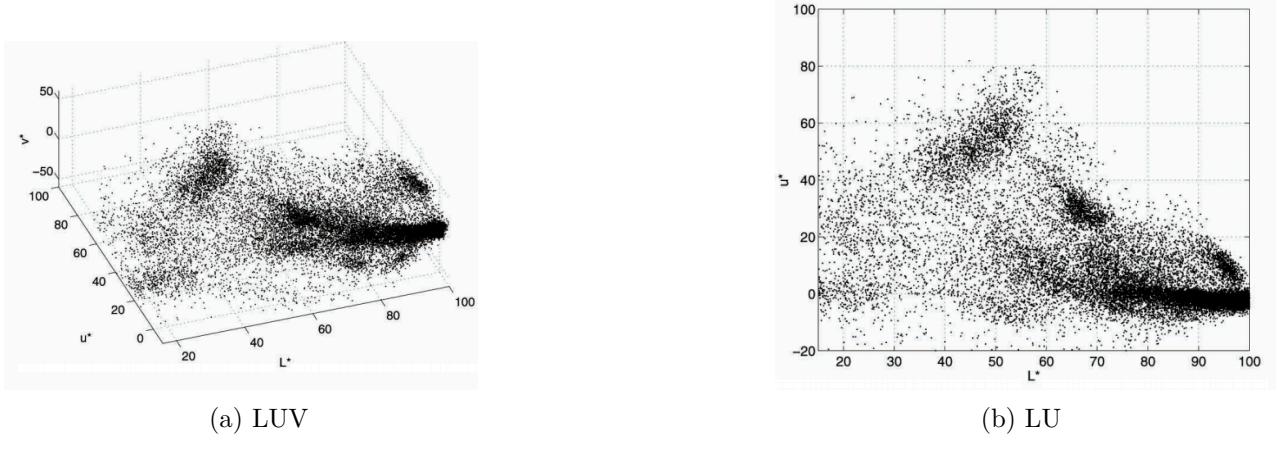


图 2: LUV 和 LU

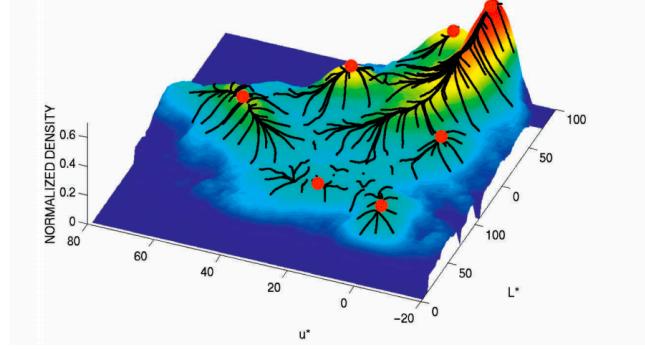


图 3: Mode

在原来 Mean shift 算法的实现中, 作者采用了基于核密度估计的策略来寻找 mode 点. 首先, 给出以下核函数公式:

$$\hat{f}(x) = \frac{1}{nh^d} K\left(\frac{x - x_i}{h}\right) \quad (1)$$

其中  $x$  代表输入数据,  $n$  是数据点数,  $h$  是带宽,  $d$  是数据的维度. 基于公式1, 作者提出了新的核密度估计方法. 首先有:

$$\hat{f}(x) = \frac{c_{k,d}}{nh^d} k\left(\left\|\frac{x - x_i}{h}\right\|^2\right) \quad (2)$$

$$\hat{\nabla} f_{h,K}(x) = \frac{2c_{k,d}}{nh^{d+2}} \sum_{i=1}^n (x - x_i) k'\left(\left\|\frac{x - x_i}{h}\right\|^2\right) \quad (3)$$

$$g(x) = -k'(x) \quad (4)$$

$$G(x) = c_{g,d} g(\|x\|^2) \quad (5)$$

这里我们用5中的  $G(x)$  作为新的核函数. 而根据上面的符号我们可以将重写 2的导数为

$$\begin{aligned}
 \hat{\nabla} f_{h,K}(x) &= \frac{2c_{k,d}}{nh^{d+2}} \sum_{i=1}^n (x_i - x) g\left(\left\|\frac{x-x_i}{h}\right\|^2\right) \\
 &= \frac{2c_{k,d}}{nh^{d+2}} \left[ \sum_{i=1}^n g\left(\left\|\frac{x-x_i}{h}\right\|^2\right) \right] \left[ \frac{\sum_{i=1}^n x_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}{g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)} - x \right]
 \end{aligned} \tag{6}$$

因此, 算法中的第二项就是我们算法中所称的 Mean shift  $m_{h,G}(x)$ , 由上面的推导可以得到:

$$\begin{aligned}
 m_{h,G}(x) &= \frac{\sum_{i=1}^n x_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}{g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)} - x \\
 &= \frac{1}{2} h^2 c \frac{\hat{\nabla} f_{h,K}(x)}{\hat{f}_{h,G}(x)}
 \end{aligned} \tag{7}$$

从式 7我们就可以看到, Mean shift 与数据密度的梯度方向相同, 这就证明了从任意点开始, 只要朝 mean shift 的方向移动, 就会慢慢的走向密度估计最大的地方, 而这也正是我们需要找的聚类中心, 也就是之前提到的 mode 点. 一个更加形式化的说明见图4, 可以看到, 不断朝着 mean shift 的方向进行移动, 最终到达了密度最大处.

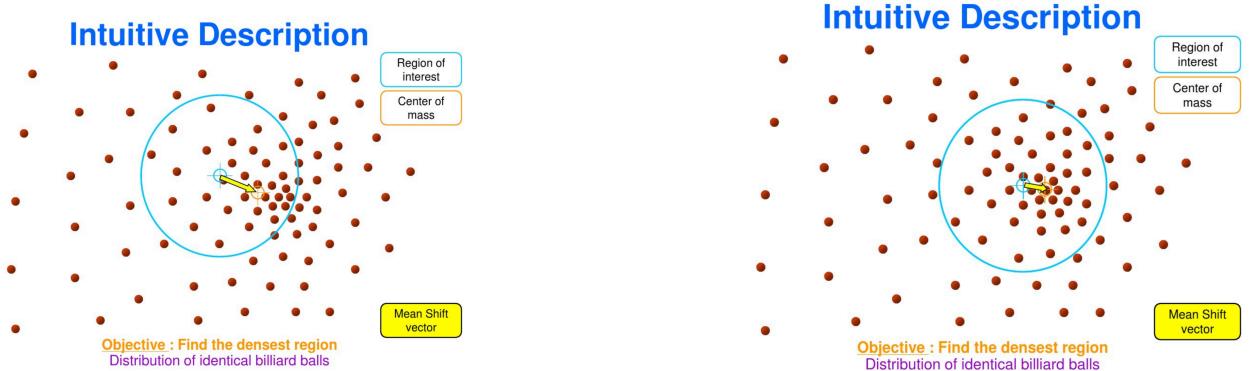


图 4: 搜索

### 1.1.2 聚类和分割

经过上面的步骤后, 对于输入数据中的每个点 (如图片中的像素点), 我们都能找到它的 mode 点, 即聚类中心. 接下来我们就可以开始围绕着聚类中心进行聚类. 原文中的具体方法是根据空间距离和颜色距离这两个度量指标, 采取类似于种子生成的策略一步步的将 mode 点附近的像素加入到各个 mode 当中, 最后对这些以 mode 为中心的, 零碎的小区域进行平滑和合并, 就得到了最终的分割图像. 比较直观的可视过程可以参见下图5.

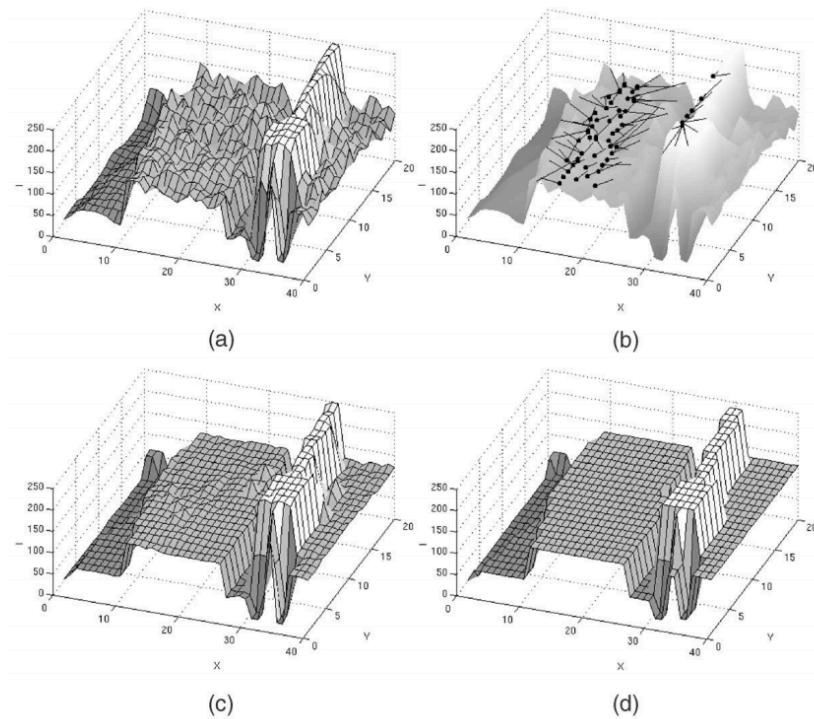


图 5: 流程

其中 a 图即是原图像经映射后得到的特征空间, b 图表示了从图像的各个点出发, 同时寻找对应的 mode 点的过程. c 图表示了根据距离相似性和色彩空间相似性得到的聚类结果, d 图即是经过了平滑, 极小区域擦除后的最终分割图像. 一个完整的例子可以参见下图6.



图 6: 例子

## 1.2 课程实验

因此, 我们组决定参照此篇论文提出的基于 Mean shift 的图像分割算法, 进行一定的复现和改进. 我们计划实现的算法大体框架如下:

### 1.2.1 Mean shift 滤波

---

**Algorithm 1:** Meanshift 算法

---

**Input:** 图片 I

**Output:** Meanshift 分割后的图片  $I'$

```
1 for  $I$  中的每个像素点  $(x,y)$  do
2   while 没有达到迭代次数上限 do
3     像素点  $(x,y)$  当前的 LUV 值为  $I_{x,y}$ 
4     设置 color_sum = [0,0,0], index_sum = [0, 0], summed_pixels = 0
5     for 对以带宽  $dh$  为半径的圆或正方形邻域内的像素点  $(x',y')$  do
6       计算  $\Delta LUV = ||I_{x,y} - I_{(x',y')}||$ 
7       if  $\Delta LUV > color\_max\_diff\_range$  then
8         color_sum +=  $I_{x',y'}$ 
9         index_sum +=  $(x', y')$ 
10        summed_pixels ++
11         $cs\_avg = \frac{color\_sum}{summed\_pixels}$ 
12         $is\_avg = \frac{index\_sum}{summed\_pixels}$ 
13        shift =  $|cs\_avg|^2 + |is\_avg|^2$ 
14        更新  $I'_{x,y} = cs\_avg$ 
15        if  $shift < shift\_range$  then
16          break
17 return 输出图像  $I'$ 
```

---

和原算法一致, 这一步同样是为了找到聚类中心. 在本次实验中我们采取了稍有不同的实现. 首先, 我们只采用 LUV 颜色空间作为特征, 据此先给出算法的伪代码1.

可以看到, 在我们的实现伪代码1中, 每个点的像素值不断地朝区域平均值的方向移动, 也即 mean shift. 待到算法终止时, 每个像素点的像素值都 mean shift 到了某一簇像素.

### 1.2.2 聚类 (Flooding)

---

**Algorithm 2:** Flooding 洪泛搜索

---

**Input:** 经过 Mean shift 后的图片 I

**Output:** 图片 I 上每个像素点对应的 label

```
1 for  $I$  中的每个像素点  $(x,y)$  do
2   分配一个唯一的  $label_{x,y} = y * width + x$ 
3   while 没有达到迭代次数上限 do
4     for 对以带宽  $dh$  为半径的圆或正方形邻域内的像素点  $(x',y')$  do
5       if  $\Delta LUV < color\_max\_diff\_range$  then
6         更新  $label_{x',y'} = label_{x,y}$ 
7 return 图像 I 对应的 label
```

---

完成初步的滤波后, 我们还需要对每一个像素都分配一个特定的标签, 以此完成图像分割的任务

要求. 而由于 Mean shift 方法本质上是无监督的算法, 因此我们只能给每个像素分配没有实际意义的标签. 实践中, 我们使用坐标给予每个像素点一个独特的初始标签, 这一步骤的具体过程如伪代码2所示.

在以上步骤中, 不难发现, 如果对上述过程进行并行化, 可能会有多个像素点抢一个像素点的情况 (赋给自己的 label), 这会由线程间的竞争决定最后谁赋值成功. 也就是说, 对于相同的输入, 标签分配算法并不能保证得到完全相同的输出. 但需要说明的是, 这里线程的竞争导致的标签不一致, 可以在后续步骤的区块合并中得到解决.

但在实际实验中, 我们发现如果对每个像素都进行并行化, 会造成比较密集的冲突和重复赋值, 降低并行化的效率. 为了解决这一情况, 我们在实际中进行了优化. 在某一部分像素点钟, 只让一个像素点线程进行洪泛搜索. 这些搜索部分的划分可以由行和列确定, 例如每个  $n * n$  的子图像只让一个像素点进行标签洪泛. 在循环中采取类似卷积移动滑动窗口的方式, 对下一个部分进行标签分配.

### 1.2.3 区域合并

---

#### Algorithm 3: 区域合并算法

---

**Input:** 经过 Mean shift 后的图片  $I$   
**Input:** 经过 Flooding 的 label  
**Output:** 最终经过聚合的 label

- 1 创建栈 stack
- 2 **for**  $I$  中的每个像素点  $(x, y)$  **do**
- 3   分配一个唯一的  $label_{x,y} = y * width + x$
- 4   stack.push( $I_{x,y}, label_{x,y}$ )
- 5 **while**  $!stack.empty()$  **do**
- 6   从 stack 中取出像素值  $x_i, y_i$  和标签  $x_l, y_l$
- 7   计算  $\Delta LUV = ||x_i - y_i||$
- 8   **if**  $\Delta LUV < color\_max\_diff\_range$  **then**
- 9     UnionFind( $x_l, y_l$ )
- 10 **return** 图像  $I$  对应的 label

---

最后一步, 根据上一步得到的聚类结果, 我们再进行最后一步后处理得到最后的分割. 对于过小的区块和相似的区块, 我们将它和邻近的进行合并, 并对分割图像进行一定的平滑和滤波处理. 即得到了分割结果.

在这一部分中, 我们采用了并查集算法 (UnionFind), 并查集算法在 (Galler and Fisher 1964) 中可以找到详细的描述, 区域合并算法的过程如伪代码3所示。

我们采用了 pthread 的多线程并行和 CUDA 并行对上面的算法进行优化, 因为 CUDA 并行结构和 CPU 多线程并行的细节之处十分不同, 因此此处只介绍主要的并行化思想, 在实现部分将具体介绍使用的算法:

1. 将待聚合标签数据进行分块，进行本地聚合处理
2. 对所有标签数据进行全局聚合处理
3. 统计区域数量，并对标签进行重新映射

## 2 Implementation

本节中，我们将对算法各个部分的具体实现进行详细描述。

### 2.1 Mean Shift 滤波的并行化

#### 2.1.1 并行优化思路

1. Mean shift 算法在迭代过程中会用到固定范围内的二维矩阵元素，这种读取方式如果采用一维读入二维转换的方法读取，性能不会特别好，而相比之下，纹理内存的存在可以极大地提高这种二维相邻的读取速度，充分利用缓存。
2. 同一个 block 内的线程下的元素，它们在执行算法循环的同时，使用到的数据大多是重复的，也就是说一个 block 内不需要把所有数据全部读取上来，可以使用共享内存进一步进行加速。
3. 对于循环内部，因为没有存在读写数据依赖，可以采用循环展开进行加速。
4. 对于浮点数值，可以添加 f 作为后缀说明，减少强制转换的开销。

#### 2.1.2 算法流程

1. CImg 读入图像数据，以 RRRGGGBBB.. 的形式保存为 float\* 数组。使用 cudaMallocPitch 函数和 cudaMemcpy2D() 函数在 device 开辟二维空间，读入该数组。
2. 通过上述内存空间创建纹理对象。
3. Block 内创建二维共享内存，第一个维度为通道，第二个维度为 BLock 内需要访问的每个通道的数据大小，大小为 BLock 的长宽分别加上 dis\_range 的乘积。
4. 共享内存写好后，执行算法的主要部分，迭代漂移到区域内的某个颜色值的相近点，最后将结果输出。

下面给出的算法的详细伪代码.

---

**Algorithm 4:** Mean Shift 并行算法

---

**Input:** Cimg 读入的图片 in\_tex, 最大颜色区别 color\_range, 图片尺寸 height, width,  
**Output:** 经过滤波的 output  
**Declare:** pixels[][] 位于共享内存

```
1 col, row ← 2D global thread_id
2 if col >= width or row >= height then return
3
4 patch_height, patch_width ← blk_w + 2 * dis_range, blk_h + 2 * disrange
5 共享内存读入; // read data
6 syncthreads()
7 cur_row, cur_col ← threadIdx.y + dis_range, threadIdx.x + dis_range
8 L, U, V ← pixels[[cur_row * patch_width + cur_col]
9 shift ← 5
10 col_from, col_to ← threadIdx.x, threadIdx.x + 2 * dis_range + 1
11 row_from, row_to ← threadIdx.y, threadIdx.y + 2 * dis_range + 1
12 for iter 0 to max_iter and shift < 3 do
13     old_row, old_col ← cur_row, cur_col // loop to update L,U,V
14     old_L, old_U, old_V ← L, U, V
15     avg_row, avg_col, avg_L, avg_U, avg_V, num ← 0
16     for r row_from to row_to do
17         for c col_from to col_to do
18             L2, U2, V2 = pixel[[r*patch_width+c]] // scan area data
19             d_L, d_U, d_V ← L2, V2, U2 - L, U, V
20             if (d_L, d_U, d_V)2 <= color_range then
21                 avg_row, avg_col += r, c
22                 avg_L, avg_U, avg_V += L2, U2, V2
23             num ++
24     num_ ← 1 / num // calculate avg
25     L, U, V ← avg_ * num
26     cur_row, cur_col ← round(avg_row * num_), round(avg_col * num_)
27     d_row, d_col ← cur_row, cur_col - old_row, old_col
28     d_L, d_U, d_V ← L, U, V - old_L, old_U, old_V
29     shift ← (d_row, d_col, d_L, d_U, d_V)2
30 output[row,col] ← L, U, V
```

---

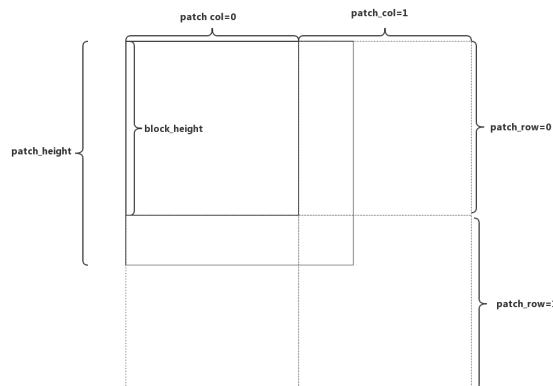
其中关于 Mean shift 算法的主要部分, 流程如下:

1. 对于每一个数据点, 获取自己的 LUV 颜色值, 命名为 L, U, V, 获取自己的 row, col 坐标值
2. 数据点需要迭代的区域范围为  $[threadIdx.x][threadIdx.y]$  到  $[threadIdx.x + 2 * disrange + 1][threadIdx.y + 2 * disrange + 1]$ 。

3. 遍历区域范围内的每一个点, 对每一个数据点, 获取它的 LUV, 命名为 L2, U2, V2, 计算与 L, U, V 的三维上的距离是否小于 color\_range, 如果成立则向 avg\_row, avg\_col, avg\_L, avg\_U, avg\_V 中加入该点的数据, 另外记录加入的点的数目 num++。
4. 迭代结束后, avg\_\* 变量中的值实际上为符合条件的点的数值的和, 因此使用 num\_ 变量作为 num 变量的倒数减少除法计算, 随后利用 avg\_\* 与 num\_ 的乘积计算出平均的 LUV 和坐标值, 并赋值到 L, U, V 变量和坐标值中去, 并根据原 L,U,V 和新 L,U,V 的二维距离 + 原 row, col 和新 row, col 的距离计算 shift
5. 以上 3-5 作为一个循环, 反复计算, 也就是不断进行漂移, 计算新的 LUV 和 row, col 值。当 shift<=3 或 iters>=max\_iter 时跳出循环。
6. 最终获取到漂移结束的 LUV 值, 将该值赋给线程处理的像素点, 输出到结果, 也就完成了算法的主要内容。

### 2.1.3 共享内存的读入

对于共享内存的读入这一方面, 采用了更好的方法:



共享内存需要读入的数据, 也就是 block 对应在图像中的位置延伸 dis\_range 的大矩形的数据, 对于这样的数据, 我们让 block 中的所有线程每次读取若干个元素, 元素数量跟 dis\_range 的范围有关, 即可完成数据的读取.

由于这个部分较为复杂, 以下结合具体代码进行分析.

blk\_x\_idx, blk\_y\_idx 为线程所在 block 在全局中对应的位置, 也是在图像中对应的位置.

---

```

1  int blk_x_idx = blockIdx.x * blockDim.x;
2  int blk_y_idx = blockIdx.y * blockDim.y;

```

```
3     int col = blk_x_idx + threadIdx.x;
4     int row = blk_y_idx + threadIdx.y;
```

---

start\_col,start\_row 为 blk\_x\_idx,blk\_y\_idx 分别减去 dis\_range 的数值，表示对于每个数据点，读取从自己的左上方开始，也正好是 block 整体需要的数据的边界。

```
1     int start_col = blk_x_idx - dis_range;
2     int start_row = blk_y_idx - dis_range;
3     const int patch_width = blk_w + 2 * dis_range;
4     const int patch_height = blk_h + 2 * dis_range;
```

---

patch\_cols,patch\_rows 是修正过后的 patch\_width/blk\_w 和 patch\_height/blk\_h 的值，用于计算出来共享内存尺寸和 block 尺寸的比值，因为 block 的线程数量有限，一个线程读取一个数据则只能覆盖好 block 范围内的点，这个做法的思想是把共享内存划分成若干个区域，通过这个值可以进行迭代，每个点去覆盖剩下的区域，确保所有区域被覆盖完全，也就把整个共享内存写入完全。

```
1     const int patch_cols = (patch_width + (blk_w - 1)) / blk_w;
2     const int patch_rows = (patch_height + (blk_h - 1)) / blk_h;
```

---

随后开始循环，设置循环变量 i,j，范围对应为 0 到 patch\_rows 和 patch\_cols，然后对于每一个区域，计算 p\_col,p\_row，对应将要读入的数据在共享内存中的坐标值，然后可以计算出共享内存的 p\_offset。另一方面，read\_col, read\_row 对应从 start\_col,start\_row 加上 p\_col,p\_row，对应的是在纹理内存中的位置。

```
1     for(int i=0; i<patch_rows; i++) {
2         for(int j=0; j<patch_cols; j++) {
3             const int p_col = j * blk_w + threadIdx.x;
4             const int p_row = i * blk_h + threadIdx.y;
5             const int read_col = start_col + p_col;
6             const int read_row = start_row + p_row;
7             if (p_col < patch_width && p_row < patch_height) {
8                 const int p_offset = p_row * patch_width + p_col;
9                 pixels[0][p_offset] = tex2D<float>(in_tex, read_col, read_row) * 100 / 255;
10                pixels[1][p_offset] = tex2D<float>(in_tex, read_col, read_row + height) - 128;
11                pixels[2][p_offset] = tex2D<float>(in_tex, read_col, read_row + height2) - 128;
```

```
12     }
13 }
14 }
```

---

#### 2.1.4 优化总结

算法在将像素点分配到线程实现并行加速的基础上，进行了以下若干优化：

1. 读取图像数据时，因为算法读取的数据点在二维上相近，使用纹理内存加速后续读取。
2. 每个 Block 内部的线程处理的像素点所需要的数据都在一定的范围内，使用共享内存来提高读取速度，另外对于共享内存的写入采用新的方法减少重复读取。
3. 算法内部的循环采用循环展开进行加速。

#### 2.1.5 加速结果

在 CPU 为 XEON E5 E5-2620 v4, GPU 为 TITAN X (Pascal) 平台上，块大小配置为 {32,32,1} 运行结果为：

图像大小	filter range	CPU 单线程版本 (ms)	GPU 版本 (ms)	加速比
640x480	2	315.9	0.469	673.5
640x480	4	942.1	0.879	1071.7
640x480	8	3539.3	5.237	675.8
640x480	16	13250.5	21.614	613.0
1920x1080	2	799.9	1.058	756.0
1920x1080	4	2332.4	4.873	478.6
1920x1080	8	8018.2	15.144	529.4
1920x1080	16	30611.5	61.714	496.0

表 1: Meanshift 运行时间表

产生的效果如图7所示：



图 7: meanshift result(dis\_range=16, color\_range=500)

## 2.2 聚类 flooding 的并行化

### 2.2.1 并行优化思路

从原始算法中我们得知洪泛搜索需要以像素点为中心，和固定半径的圆或正方形范围内的像素进行对比，若它们之间的差异小于阈值，那么我们应该将当前点的标签设为对比点的标签。原始算法中每个像素对应线程需要访问的像素点如图8所示

0	1	2
3	4	5
6	7	8

图 8: original access pattern( $r=1$ )

而且这仅仅是一次循环所需要访问的像素量，为了让更多的像素间产生相互联系，我们往往需要进行多次循环，通常采用的循环量为 iter=20.

为了减少运算量，我们对该算法稍微进行改动，变为每个像素点仅搜索左侧和上侧固定半径内的像素点，同时增加一个“标签冒泡”(label propagation)的步骤，让每个像素点顺着 flooding 产生的 label 链条向根部查找，直到查找到的像素的  $\Delta LUV > color\_range$ 。这样既方便并行，又有利于获得更好的空间局部性。

以上描述的算法中，每个线程块访问的像素点如图9所示。阴影区域为需要读取的，非本线程对应的像素。

0	1	...	31				
32	33	...	63				
64	65	...	95				
		...					
96 0	96 1	...	99 1				
99 2	99 3	...	10 23				

图 9: original access pattern( $r=4$ )

### 2.2.2 加速结果

1. 我们首先实现了 CPU 单线程版本作为基准，使用 640x480 的灰度图像作为测试数据，经过 profile，可得到 CPU 单线程版本的洪泛搜索耗时 165ms。
2. 然后我们将 CPU 单线程版本改为使用全局内存的 CUDA 版本，令每个 CUDA 线程处理一个像素点，经过 profile，可得到使用全局内存的 CUDA 版本的洪泛搜索耗时 490us，实现了约 337 倍的加速。
3. 接下来我们利用 GPU 的共享内存来存放子图像素，经过改进后，得到利用共享内存的 CUDA 版本的洪泛搜索，经过 profile，耗时 202us，实现了约 816 倍的加速。
4. 之后我们在翻阅代码的过程中发现，对比差异仅需进行一次，之后可以仅对标签进行冒泡，于是我们将代码重构，让洪泛搜索中的搜索和冒泡分离出来，经过 profile，耗时 215us，反而比没优化之前耗时更多。我们对这个现象感到好奇，经过一番对比和分析，最终找到了原因：我们每次冒泡都调用一次冒泡核函数，但由于显卡的机制，当显卡执行完当前核函数后，当前核函数所使用的缓存会被无效化，导致每次执行冒泡核函数都需要进行全局内存访问，因此拖累了运行时间。对此，我们将循环置于核函数内部，经过 profile，耗时 119us，最终实现了约 1387 倍的加速。

产生的效果如图10所示：

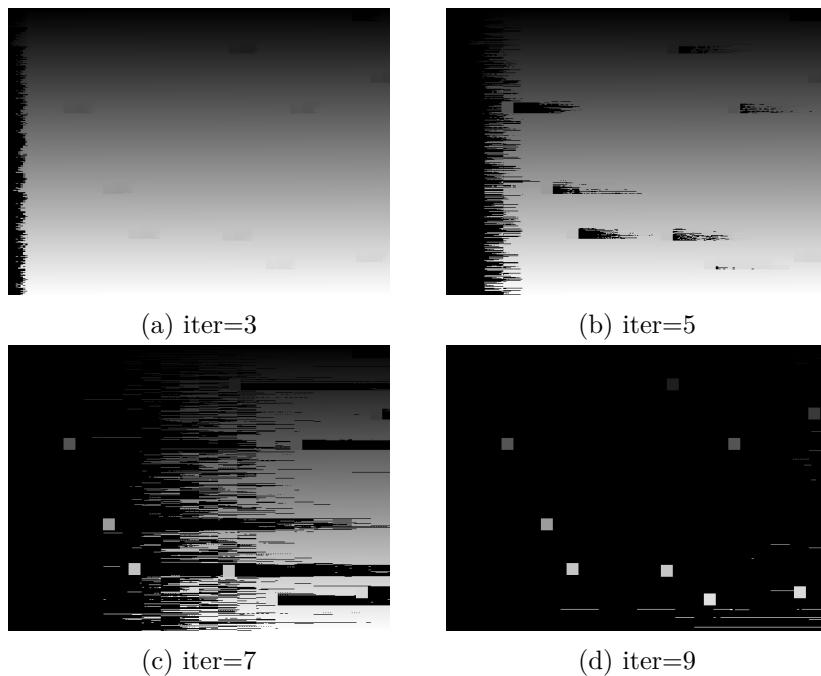


图 10: flooding result

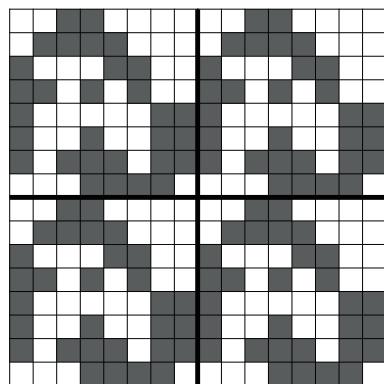
## 2.3 区域合并的并行化

### 2.3.1 CUDA 加速

本节中，我们将对我们方法的 CUDA 核函数进行描述，我们的算法基于 (Chen et al. 2017) 中描述的 UnionFind 算法进行改进。我们使用了三个核函数，每个核函数的功能如下：

1. 第一个核函数基于局部 UF 合并，通过使用共享内存来提升数据的访问速度，局部合并为最后的全局合并降低了复杂度
2. 第二个核函数对每块的边界进行分析和合并，建立块与块之间的连接关系
3. 第三个核函数对所有块进行全局合并，得到最终结果

#### A. 局部 UF 合并



(a) input data

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

(b) initialized local label map

图 11: Input data and initialized local label map

局部 UF 合并包括三个阶段：初始化，粗糙分类及行列归一化，局部 UF。它的伪代码如5所示。

---

**Algorithm 5:** 区域合并算法

**Input:** 经过 Mean shift 后的图片 I  
**Input:** 经过 Flooding 的 labelIn  
**Input:** 最大颜色区别 colorRange  
**Output:** 最终经过聚合的 labelOut  
**Declare:** int  $label_{sm}[], labelOrg_{sm}, pixelBuff_{sm}$  位于共享内存  
**Declare:** int x,y,tid,gid,temp

```
1 x,y ← 2D global thread id
2 if  $x >= imgWidth or y >= imgHeight$  then
3   return
  // read data
4 tid ← 1D block thread id
5 gid ← 1D global thread id
6  $label_{sm}[tid] = tid$ 
7  $labelOrg_{sm}[tid] = labelIn[gid]$   $pixelBuff_{sm}[tid] = I[x,y]$ 
8 syncthreads()
  // row scan
9 if  $||pixelBuff_{sm}[tid] - pixelBuff_{sm}[tid - 1]|| < colorRange$  then
10   $label_{sm}[tid] = label_{sm}[tid - 1]$ 
11 syncthreads()
  // column scan
12 if  $||pixelBuff_{sm}[tid] - pixelBuff_{sm}[tid - blockDim.x]|| < colorRange$  then
13   $label_{sm}[tid] = label_{sm}[tid - blockDim.x]$ 
14 syncthreads()
  // row-column unification
15 temp ← tid
16 while  $temp \neq label_{sm}[temp]$  do
17   $temp \leftarrow label_{sm}[temp]$ 
18   $sm[temp] = temp$ 
  // local union find
19 if  $||pixelBuff_{sm}[tid] - pixelBuff_{sm}[tid - 1]|| < colorRange$  then
20   $unionFind(label_{sm}[], tid, tid - 1)$ 
21 syncthreads()
22 if  $||pixelBuff_{sm}[tid] - pixelBuff_{sm}[tid - blockDim.x]|| < colorRange$  then
23   $unionFind(label_{sm}[], tid, tid - blockDim.x)$ 
24 syncthreads()
  // store result
25  $label[gid] = labelOrg[label_{sm}[tid]]$ 
26 return 图像 I 对应的 label
```

---

### 1) 初始化

我们将图片分割成和线程块一样大小的若干块，如图11a所示，并且将每块分配给不同的线程块处理，线程块通过共享内存来提升访问速度 (*CUDA Toolkit Documentation 2019*)，函数读取原始的 label 到共享内存  $labelOrg_{sm}$  中方便最后查找，并且初始化  $label_{sm}$  数组中元素为块内线程 id，一个初始化完毕的 8x8 大小的  $label_{sm}$  数组如11a所示。

### 2) 粗糙分类

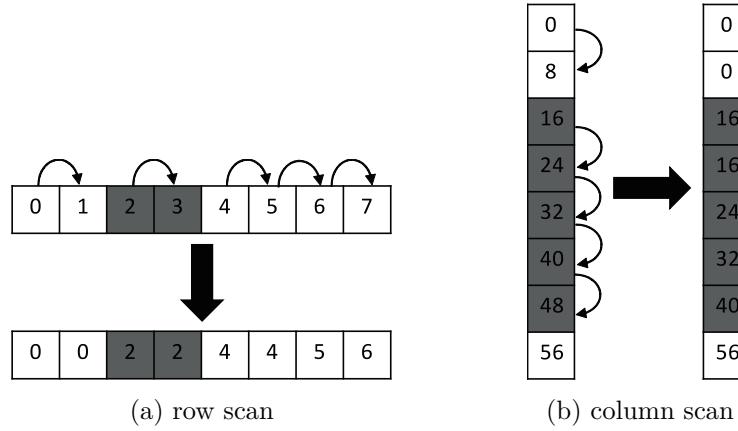


图 12: Scan model

在一个已经初始化的本地  $label_{sm}$  数组中，左方和上方的像素点的 label 值总是小于当前像素点的 label 值的，基于这个事实，我们可以先进行行扫描，融合同一行上相邻的像素点，然后进行列扫描，融合同一列上相邻的像素点，图12a描述了行扫描的过程，图12b描述了列扫描的过程。这两种内存访问方式不会发生 bank conflict，因此十分高效。

### 3) 行列归一化

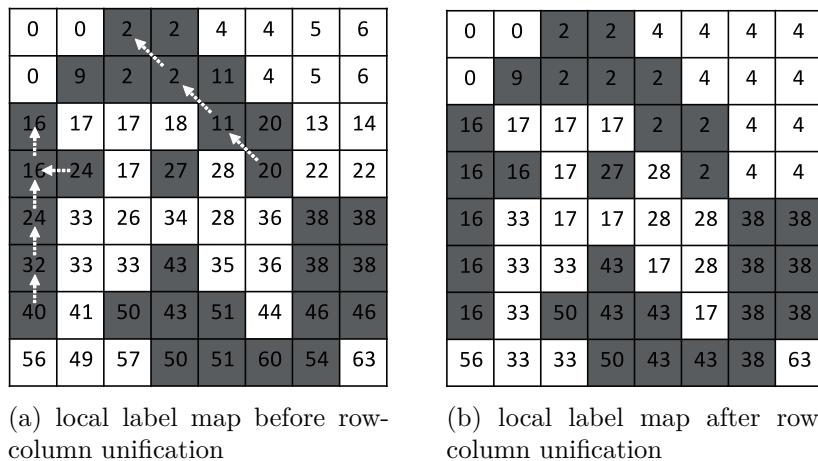


图 13: Coarse labeling

行列扫描后进行行列归一化，归一化通过对块内每个像素调用 *unionFind* 算法完成像素点的标

签统一, 此过程将根据行列扫描形成的邻接链条搜索最小局部 label, 并更新像素点的值为此 label. 因为线程块的大小受限于 CUDA runtime, 因此这个查找过程的深度也是受限的, 另外, 使用共享内存也提高了访问效率.

## B. 边界分析

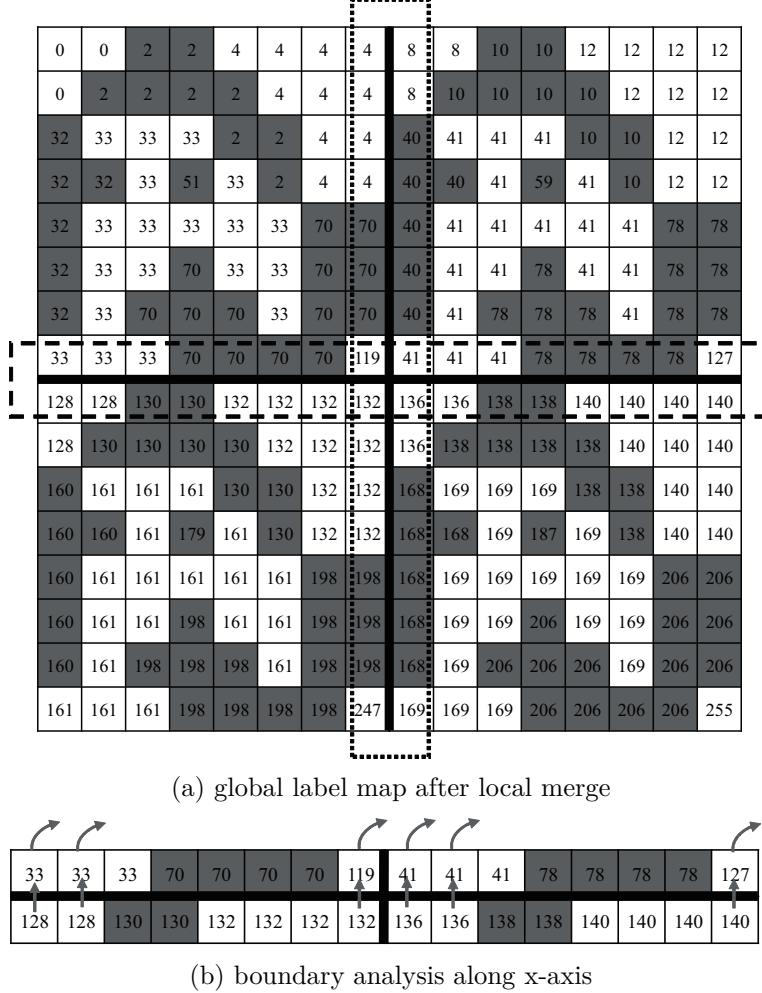


图 14: Coarse labeling

在边界分析阶段, 我们仅仅在每个块的邻接边上进行 *unionFind*(图14a虚线框内), 为了节省线程数目和简化程序, 我们将边界分析拆分为水平 (H) 和竖直 (V) 两个阶段, 两个阶段的线程数目分别和水平与竖直方向各自邻接边的长素之和相等, 假设输入图片的分辨率为  $N \times M$ , 线程块大小为  $\{b_x, b_y, 1\}$ , 则水平与竖直方向需要处理的像素数目 (即线程数目) 表示如下:

$$P_x = \lfloor N/b_x \rfloor * M \quad (8)$$

$$P_y = \lfloor M/b_x \rfloor * N$$

边界分析的伪代码如下:

---

**Algorithm 6:** 边界分析算法 (H)

```
Input: 经过 Mean shift 后的图片 I
Input: 经过 Local merge 的 label
Input: I 的尺寸 width,height
Input: 最大颜色区别 colorRange
Declare: int x,y,gid
1 x,y ← 2D global thread id
2 gid ← 1D global thread id
3 if  $x >= imgWidth$  or  $y >= imgHeight$  then
4   return
  // read data
5 if  $\|I[x, y] - I[x, y - 1]\| < colorRange$  then
6   unionFind(label[], gid, gid-width)
7 return 图像 I 对应的 label
```

---

**Algorithm 7:** 边界分析算法 (V)

```
Input: 经过 Mean shift 后的图片 I
Input: 经过 Local merge 的 label
Input: I 的尺寸 width,height
Input: 最大颜色区别 colorRange
Declare: int x,y,gid
1 x,y ← 2D global thread id
2 gid ← 1D global thread id
3 if  $x >= imgWidth$  or  $y >= imgHeight$  then
4   return
  // read data
5 if  $\|I[x, y] - I[x - 1, y]\| < colorRange$  then
6   unionFind(label[], gid, gid-1)
7 return 图像 I 对应的 label
```

---

### C. 全局合并

在进行完边界分析后, 我们在全局内存上对所有像素点进行 *unionFind* 操作, 此时互相独立的局部标签被关联为一个这前提, 我们使用和 (Yonehara and Aizawa 2015) 以及 (Oliveira and Lotufo 2010) 中一样的方法进行. 此算法更具体的描述可以参考 (Galler and Fisher 1964)

### 2.3.2 Pthread 加速

本节中我们使用 c 语言的 `<pthread.h>` 库来实现 union\_find 多线程并行。并行的思路跟 cuda 实现的 union\_find 类似。

---

#### Algorithm 8: Pthread 区域合并算法

---

**Input:** height,width  
**Input:** mode meanshift 后的图像像素，一维 float 数组  
**Input:** labels,color range,thread\_num  
**Output:** new\_labels

```
1 block_size=height*width/thread_num
2 pthread_data[thread_num]
   // img_divide and partial_union_find
3
4 for i in range(0,thread_num) do
5   start=i*block_size
6   end=min(height*width-1,i*(block_size+1)-1)
7   pthread_data[i]=createPthreadData(start,end)
8   pthread_create(pthread[1],partial_union_find_function,pthread_data[i])
   // wait for all thread end
9 for i in range(0,thread_num) do
10  pthread_join(pthread[i])
    // boundary analysis
11 for i in range(0,thread_num-1) do
12  end=min(height*width-1,i*(block_size+1)-1)
13  for j in range(0,width) do
14
15    if color_distance(mode[end-j],mode[end-j+width])<color_range then
16      union_find(end-j,end-j+width)
   // global_merge
```

---

#### A. 实现思路

1. 对于一个大小为 `height*width` 的图片，把图片分成 `height*width/THREAD_NUM` 大小的 `THREAD_NUM` 块，在每个子线程进行图片的局部并查集扩展
2. 主线程处理边界问题，把边界上 `color_distance` 相近的像素点所在的并查集合并起来
3. 主线程通过归并排序把所有子线程的局部并查集进行最终合并，并对图像做重新标注，更新 label

在完成区域合并的实现之后，我们还进行了性能评估，将我们所实现的并行版本的性能同串行版本的性能做了比较。我们选取了若干张典型图片15进行测试。

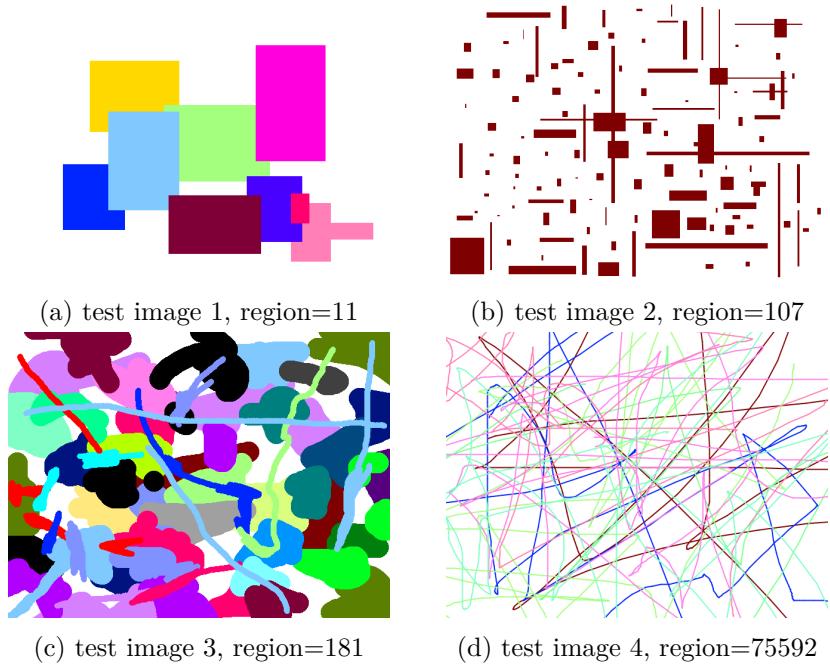


图 15: 输入图片

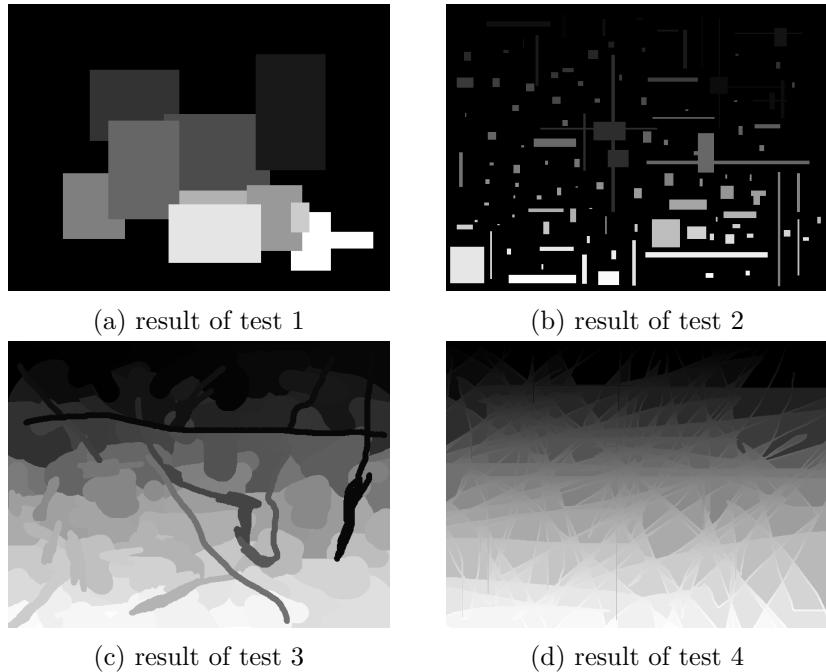


图 16: 输出图片

我们首先验证了两种算法的正确性, 输入图片经由区域合并算法的处理后, 都得到了如图16的输出. 经过程序核验, 可以确保所实现算法的正确性.

#### A. Pthread 加速效果

在 CPU 为 XEON E5 E5-2620 v4 的平台上, 对四张图片分别进行五次单线程跟多线程的 union\_find

并统计运行时间，结果如表2

测试图像	线程数目	测试 1(s)	测试 2(s)	测试 3(s)	测试 4(s)	测试 5(s)	平均时间 (s)
1	1	0.153	0.144	0.149	0.149	0.148	0.1486
1	2	0.144	0.153	0.151	0.141	0.148	0.1474
1	4	0.152	0.139	0.142	0.137	0.143	0.1426
1	8	0.181	0.17	0.175	0.186	0.174	0.1772
2	1	0.145	0.143	0.153	0.145	0.143	0.1458
2	2	0.144	0.142	0.14	0.14	0.141	0.1414
2	4	0.176	0.134	0.14	0.137	0.138	0.145
2	8	0.176	0.172	0.178	0.172	0.166	0.1728
3	1	0.154	0.162	0.153	0.161	0.156	0.1572
3	2	0.156	0.144	0.157	0.151	0.146	0.1508
3	4	0.144	0.14	0.156	0.14	0.146	0.1452
3	8	0.184	0.18	0.184	0.189	0.184	0.1482
4	1	0.3	0.299	0.29	0.284	0.281	0.2908
4	2	0.279	0.265	0.269	0.27	0.281	0.2728
4	4	0.267	0.256	0.252	0.25	0.264	0.2578
4	8	0.313	0.316	0.316	0.327	0.328	0.32

表 2: Pthread 运行时间表

由表格我们知道多线程在线程数目为 2 或者 4 的时候性能最佳，但当线程数目继续增加时，由于线程创建销毁同步所需的时间更长，反而造成性能的下降。接下来，我们使用了一系列测试，通过实验来测出我们所实现的并行算法的实际加速比. 上述若干张图片的运行时间如下表所示.

## B. CUDA 加速效果

在 GPU 为 TITAN X (Pascal) 平台上, 块大小配置为 {32,32,1}. CUDA 版本的 union find 效果如表3所示

测试图像	CPU 单线程版本 (ms)	GPU 版本 (ms)	加速比
1	233.2	0.124	1881.4
2	232.4	0.127	1829.9
3	243.1	0.126	1857.9
4	243.3	0.148	1643.9

表 3: UnionFind 运行时间表

### 3 Realtime test

我们在 CPU 为 i7 6700HQ, GPU 为 GTX 970m 的平台上运行了连续的实时视频流处理测试, 所有 CUDA 核的块大小均配置为 {32,32,1}, 在 radius=16, color\_range=1000, connect\_color\_range=300 的配置下, 单帧平均处理时间**28.32ms**, 运行结果如下

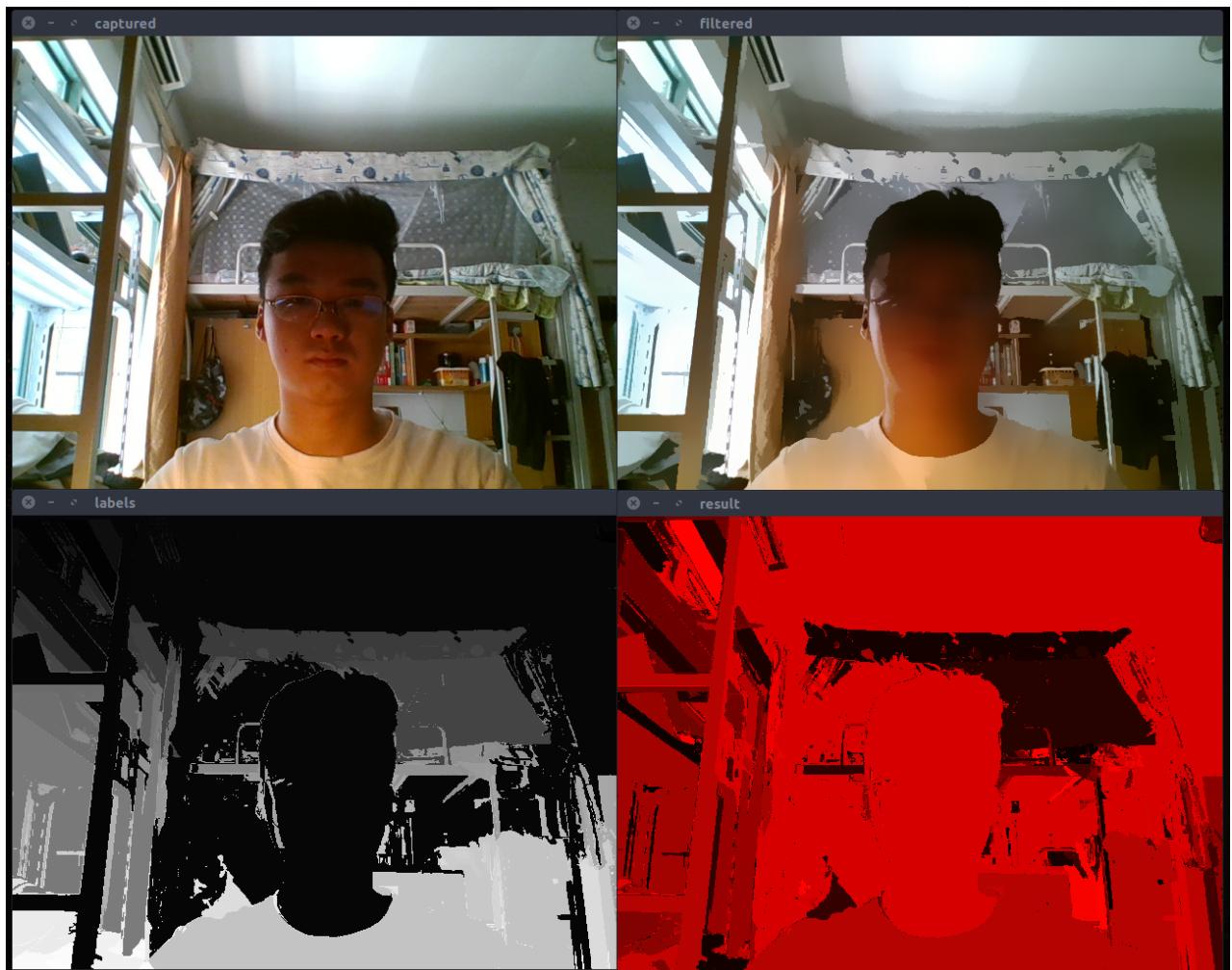


图 17: realtime test

## 4 Conclusion

本次实验中我们实现了并行化的 Mean Shift 算法, 但是还是有一些不足之处,

最后, 我们概括的总结一下我们小组在本次实验中的工作. 首先, 我们根据参考文献, 选取了 Meanshift 图像分割这一课题, 对它进行复现和并行化的优化. 在准备实验工作之前, 我们在开题报告时进行了充分的讨论和筹划, 并且在理论上重复了算法的推导过程, 验证了该算法的正确性和可靠性.

在实验中, 本小组成员分工明确, 配合紧密. 利用 Github 的 pull-request-merge-branch 等机能进行了高效的团队协作, 在共同拟定出的接口文档, 测试规范, 算法模型等文件的指导下很好的实现了模块化的分工和合作.

具体到实验设计上, 我们充分发挥了严谨认真的精神, 对于整个 Meanshift 算法的每个子模块, 都至少复现了串行版本和并行版本, 进行性能的对比实验. 有些部分还利用了 CPU 上的并行, 如某些模块还使用了 OpenMP 进行并行化等. 在每次实验中, 也充分发挥了科学对照的精神, 每个模块都要求进行测试, 验证正确性. 同时在每个模块的效率考察上, 也务必做到多次重复实验取平均值, 控制并调整参数以取得最优表现等.

更重要的是, 在实验内容上, 我们成功并正确的复现了 Meanshift 滤波, Flooding, Union Find 等一系列算法模块, 所有模块都通过了正确性测试. 许多模块的 CUDA 加速版本比起串行版本, 都取得了相当巨大的加速成绩, 这里的具体内容可以参照我们前文的实验报告. 而在 CUDA 加速的实现上, 我们软硬结合, 从算法考虑到 GPU 的内存, 架构采用了一系列优化措施. 比如有, 在 Meanshift 滤波中采用纹理内存, 共享内存; 在许多循环中使用了循环展开; 考量程序的时空局部性等等.

总的来说, 在本小组各成员的紧密配合, 有效协作下, 本次实验取得了比较完满的成功. 但也要看到, 我们的项目仍然有一些不足之处, 仍然有可以优化的空间. 比如不能自适应, 需要一些超参数, 不能很好适应不同大小和纹理的图片, 敏感度高分割会分出很多噪点, 敏感度低又会损失细节等. 在日后的学习工作中, 如果有机会, 我们将进一步完善我们的工作, 力求做得更好.

## References

- Chen, Jun et al. (2017). “An Optimized Union-Find Algorithm for Connected Components Labeling Using GPUs”. In: *CoRR* abs/1708.08180. arXiv: [1708.08180](https://arxiv.org/abs/1708.08180). URL: <http://arxiv.org/abs/1708.08180>.
- Comaniciu, Dorin and Peter Meer (2002). “Mean shift: A robust approach toward feature space analysis”. In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 5, pp. 603–619.
- CUDA Toolkit Documentation* (2019). <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>. [Online].
- Galler, Bernard A. and Michael J. Fisher (1964). “An Improved Equivalence Algorithm”. In: *Commun. ACM* 7.5, pp. 301–303. ISSN: 0001-0782. DOI: [10.1145/364099.364331](https://doi.acm.org/10.1145/364099.364331). URL: <http://doi.acm.org/10.1145/364099.364331>.
- Oliveira, Victor and Roberto Lotufo (2010). “A Study on Connected Components Labeling algorithms using GPUs”. In: vol. 2010.
- Yonehara, K. and K. Aizawa (2015). “A Line-Based Connected Component Labeling Algorithm Using GPUs”. In: *2015 Third International Symposium on Computing and Networking (CANDAR)*, pp. 341–345. DOI: [10.1109/CANDAR.2015.78](https://doi.org/10.1109/CANDAR.2015.78).

## A 个人报告

### A.1 16337180 麦显忠

本次项目中我负责了 unio\_find 模块的多线程并行模块，这一模块用 c 的 pthread 库来实现多线程，目的是为了跟 CUDA 版本的 union\_find 做性能比较。基本的并行化思路是对图片进行分割，在子线程对图像的局部做 label 的合并，最后在主线程进行边界分析跟区域合并。

在这次项目中自己首次接触了用 Cmake 来组织项目，Cmake 虽然比较繁琐，但对项目模块化十分有利，每个人只需要编译自己的 Cmake 文件就可以运行自己的模块，不会对队友的模块产生任何影响。

最后一点，感谢沐晗同学对我们的指导，虽然我是名义上的队长，但实际很多工作是沐晗在组织推进的。我自己也通过这次项目体会到一套完整的项目开发流程，像代码的组织、模块的划分、编程规范的规定、接口的规定、提交代码后他人的 review，希望以后在团队项目中能用上这次项目的开发经验，与他人有跟愉快的合作开发。

### A.2 16313018 李沐晗

本次实验中，我所做的工作如下：

1. 创建并维护项目的接口文档，编码规范等文档
2. 创建并维护项目的 cmake 文件，维护项目的文件结构
3. 为项目的每个模块查找合适的并行算法
4. 设计项目的各个模块及测试程序和测试脚本，设计各个算法并行方法
5. 编写项目的 CUDA union find 算法模块
6. review 其它同学的代码，提供详细的设计和优化建议

本次项目比较棘手的地方是，每个模块有多种算法可供选择，而算法与算法之间的效率差异非常显著，同时针对每一种算法，又需要不同的并行方法，因此设计的工作是比较复杂的。比如我负责的模块功能是区域连接（第三阶段），我初始想法是把这个问题处理成一个 min-cut 问题，并了解了 push-relabel 算法，但发现这种算法不适合修改来适应不确定的多个区域的分割和区域内融合的需求，所以放弃，后来了解到了适合 CUDA 并行的 union find 算法，发现非常适合我的模块需求，因此采用了这个算法来进行并行。对于第二阶段的 flooding 算法，我也考虑了一些时间，最后选择设计成“向左看，向上看” + “标签冒泡”这样的方法来提高循环的效率。

第二个比较麻烦的地方是，指导其它同学时，传达自己的想法比较困难，尤其是对于 CUDA 这种非常依赖于访问模式的并行化方法，经常需要花费大量时间来详细描述每个线程的访问模式，因此在 review 的过程中，我发现缺乏一种有效的工具来告诉他人如何来写。

最后一个地方就是多人合作 C++ 项目需要非常严格的接口规范和项目代码维护及检查，这一点上，我觉得我们做的还不够好，我虽然给出了初始的设计模板及代码规范，但后来发现在迭代过程中做了很多的修改，导致前面的设计完全不适用了，而且代码风格统一也需要比较多的精力，不过我通过加大 review 的力度及与其它同学的沟通交流解决了这个问题。

综上所述，在本次项目中我还是学习到了比较多的 CUDA 程序设计及合作经验的，希望以后的工作中能够用上本次项目的实践经验，做得更加完善。

### A.3 16337242 韦博耀

在本次项目中我负责了洪泛搜索模块的实现和优化。我对原始的 flooding 算法进行了部分改进，使之更加适合 CUDA 并行化。之后按顺序逐步实现了 CPU 和 GPU 版本，并对 GPU 版本进行了优化。在优化过程中，我了解到了许多有用的调试、profile 等工具，例如利用了 cuda-memcheck 来检查内存错误，利用 nvprof 来进行 profile，利用 cuda-gdb 来进行调试。除此之外，我在优化代码时对代码进行了重构，但结果却比优化前要差，这就促使我去寻找更加深层的原因，最终对 GPU 内部的结构有了更加深刻的认识，从而成功优化了代码，最终结果跟 CPU 版本相比实现了 1387 倍的加速。

除此之外，本次项目使我对怎样进行高效的 CUDA 变成有了更加深刻的认识，例如利用好 IDE 提供的功能，充分进行模块化组织，编写好单元测试与集成测试，同时还有怎样高效地查阅官方文档。在本次项目中，CUDA 官方文档给予了我巨大的帮助，使得我能够更好地编写 CUDA 代码。

还有一点就是 Cmake 的使用。虽然在日常学习中已经习惯了 CMake 的使用，但本次项目中 CMake 的使用方法还是让我眼前一亮，这样的组织方式和使用方法还是好评，让我学习到了许多新的操作，大大提升了我对使用 CMake 的经验。

最后，感谢沐晗同学在本次项目中对我的帮助。通过学习沐晗同学的项目组织、代码和优化思路，我学习到了许多新的东西，同时也得知了优秀的编程规范和项目组织方式，还有各种高效的编程方法。

### A.4 16337259 谢江钊

在本次项目中我主要负责了 Meanshift 模块的实现和后续的并行优化，这一部分的难点在于尽管它的算法并不复杂，但是并行的优化上来说还是有充分的思考空间的。从最开始的简单的一个像素一个线程，到后来考虑纹理内存提高内存的存取速度，然后发现 block 内部读取的范围一致使用共享内存提高速度，最后针对循环内部进行展开，以及还有减少强制转换提高速度，都做了很多各种各样的努力。而最终的结果也是非常优秀的，而最开始的没有做任何处理的优化，时间足足是现在的三十

倍。CUDA 内部的优化方法是丰富的，比起之前的小作业，这个大项目更加充分地让我掌握到一些优秀的优化技巧，同时也非常感谢沐晗同学的付出与指导，在这个项目中帮助了许多。

另外一点是这个项目采用了 Camke 来组织项目的配置，各个模块之间的调用，实现，声明都被区分开来，这种做法其实对我来说还是第一次接触，还有一些 C 调用 CUDA 的手段，利用模板去避免潜在的编译问题对我来说也是挺新颖的，因此在这些方面踩了不少的坑也学会了不少相关的知识。

最后一点是这次项目合作的地方我也觉得还有可以改进的地方，对接口的规范，文档组织和代码风格都可以进一步形成系统的解决方案。综上，我还是从这次项目中学习应用到了非常丰富的 CUDA 优化知识和团队合作经验，希望在以后的项目中能够吸收这次的经验。

## A.5 16337179 麦金杰

在本次项目中，我主要负责开题报告以及结题报告的撰写，整合，加工等工作，同时还参与了性能评测部分的工作。总的来说，因为我的分工中没有涉及到较多的实验和代码编写部分，因此相对来说我的任务在小组中是比较轻松的。因此在这里，首先要感谢其它负责实验的同学的付出，没有他们的付出和劳动，是不可能写出这样一份详尽而充实的实验报告的。

在编写开题报告的过程中，当时的主要参考 paper 是 Comaniciu and Meer 2002。为了撰写报告和参与之后的工作的需要仔细阅读了这篇 paper，对于我们在这次实验所复现的 Meanshift 算法在理论层面有了比较深的认识和体会。总的来看，原文的重心其实在于推导和论证，而并非算法的具体步骤或者实现。这其实就给了我们的后续实现比较大的优化空间。作者采用核密度估计的方法步步推导，论证了朝 Mean shift 方向进行移动，进行图像分割的理论基础和有效性。我的认识和体会就是，作为深度学时代之前方法，虽然它有着分割区块比较细碎，受噪声和图像局部信息影响大等等缺点，但它有着扎实的理论基础，可以达到相当快的运算速度。

在我参与性能评测的过程中，由于我们实现了独立而完整的测试模块，这就使得我进行性能评测时，调用各模块代码，与串行模块对比，检查输出和计时等变得相当方便。体会到了在项目中引入测试模块，基准评价体系的便捷和重要性。

在我完成结题报告的过程中，采取了在 Overleaf 线上写作 LATEX 文档的方式。这不仅使得各成员间可以进行有效的沟通，快速检查报告的内容和进度，同时基于 LaTex 的文档也让我们在算法的描述，报告内容组织上更加得心应手。而与此同时，在整合各成员的实验工作，与同伴进行算法和实验细节上的交流过程中，也令得我对于整个算法既有比较宏观的整体把握，也了解了更多项目实现过程中的具体细节，比方说实验中遇到的困难，各个模块间是如何优化的等等。从总结他人的实验工作中也受益良多。

最后，感谢所有人的付出!!!