

SIFT CPU及其优化思路

1. 程序及程序单元测试流程

- python读取图片（为什么使用Python：因为C++的opencv有点难装，Python方便展示结果）
 - 转为灰度值np数组（当sift算法精度为double时，其类型dtype应转换为float64）
- python调用共享库（在linux下为.so文件）
 - 共享库由测试程序的cpp、h文件编译得到（使用-fPIC -shared 选项，对于nvcc，还要使用-Xcompiler）
 - 使用ctypes进行数据转换¹
 - 声明共享库函数的argtypes（对于numpy数组，使用np.ctypeslib.ndpointer）

```
test.sift.argtypes = [np.ctypeslib.ndpointer(dtype=gray.dtype, ndim=2,
shape=gray.shape, flags='C_CONTIGUOUS'),
                        np.ctypeslib.ndpointer(dtype=res.dtype, ndim=2,
shape=res.shape, flags='C_CONTIGUOUS'),
                        ctypes.c_int, # n
                        ctypes.c_int, # m
                        ctypes.c_int, # kr
                        ctypes.c_int, # ks
                        ctypes.c_int, # ko
                        ctypes.c_int, # S
                        ctypes.c_double, # sigma_init
                        ctypes.c_double, # contrast_threshold
                        ctypes.c_double, # edge_response_threshold
                        ctypes.c_int, # max_interpolation
                        np.ctypeslib.ndpointer(dtype=time_arr4.dtype, ndim=1,
shape=time_arr4.shape)
]
```

- 调用时直接传递Python变量（除了传指针：需要使用ctypes.POINTER声明argtypes，用ctypes.pointer传参）

```
test.sift(gray, res, n, m, kr, ks, ko, S,
          sigma_init, contrast_threshold,
          edge_response_threshold, max_interpolation, time_arr4)
```

- test.ipynb文件包含了单元测试的例子（绘制高斯金字塔的前几层、绘制特征点）

2. SIFT on CPU

SIFT相关原理可以参考SIFT论文²，以及我在计算机视觉课程中编写的SIFT学习笔记³：

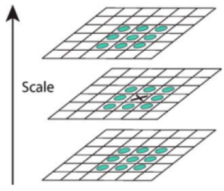


Figure 2. The extrema in DoG space and its 28 neighbors.

The extrema in DoG space, as shown in figure 2, are the interest points that have different scales and therefore provide scale invariant information. The extrema, $\{(x_e, y_e, z_e, i_e, j_e)\}$, are taken from the S DoG layers in the middle of each octave, $E = \{D_{ij} | i, j \in z, 1 \leq i \leq K, 2 \leq j \leq S + 1\}$, when $D(x_e, y_e, \sigma_{i_e j_e})$ is larger than all the 28 neighbors in the scale space:

$$D(x_e, y_e, \sigma_{i_e j_e}) > D(x_e + \Delta x, y_e + \Delta y, \sigma_{i_e(j_e + \Delta j)}),$$
$$\forall \Delta x, \Delta y, \Delta j \in \{-1, 0, 1\}, (\Delta x, \Delta y, \Delta j) \neq (0, 0, 0)$$

, or smaller than all the 28 neighbors.

The condition 1 ensures that the scales of DoG layers E , which generate extremas, increase smoothly at a constant scale of $2^{1/S}$:

can modify the initial smoothing degree. To compensate the loss of highest spatial frequencies in the input image cause by the initial smoothing, the input image is expanded by a factor of 2 using bilinear interpolation, prior to building the pyramid.

3.2. Accurate Keypoint Localization

3.2.1 Interpolated Location of the Extremas

The scale space shown in figure 1 is discrete, so the extrema extracted from it can change to one of its neighbors when the original scale changes. In 2002, Brown and Lowe [3] proposed an approach to determine the interpolated location of the extremas, which can improve the stability of the extremas. The approach uses the Taylor expansion to fit the 3D quadratic function to the scale space function, shifted so that the origin is at the sample point (i.e., the previous detected extrema):

$$\tilde{D}(\mathbf{x}) = D + \frac{\partial D}{\partial \mathbf{x}}^T \mathbf{x} + \frac{1}{2} \mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2} \mathbf{x}$$

where $\mathbf{x} = (x, y, \sigma)^T$ is the offset from this point and D , $\frac{\partial D}{\partial \mathbf{x}}$ and $\frac{\partial^2 D}{\partial \mathbf{x}^2}$ are the value at the sample point of the original function and its derivatives. Then the extremum $\hat{\mathbf{x}}$ is determined by setting the derivative of the function with respect

下述代码为多核课程大作业期间原创：

2.1. 构建高斯金字塔

这一步需要构建高斯金字塔，作为图像的尺度空间表示。高斯金字塔由多个octave组成，每个octave包含S+3层图像，第一个octave的第一层图像由原图2倍上采样得到，此后每个octave的第一层图像由上一个octave的第S层1/2倍下采样得到，每个octave中，下一层由上一层进行高斯模糊得到，它们相对于原图的高斯模糊参数是以 $2^{1/S}$ 为间隔连续的。高斯差分金字塔的每个octave由高斯金字塔的每个octave的每相邻两层的后一层减去前一层得到。

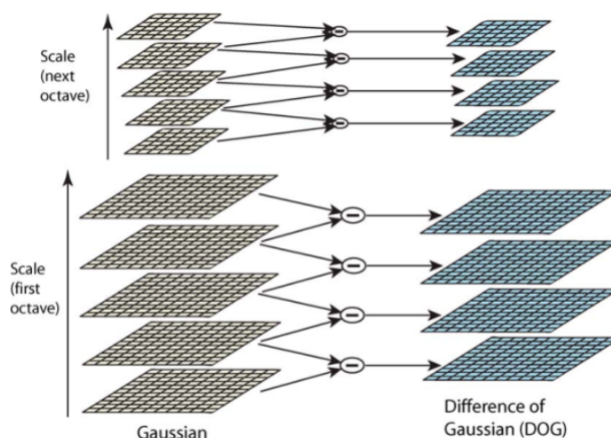


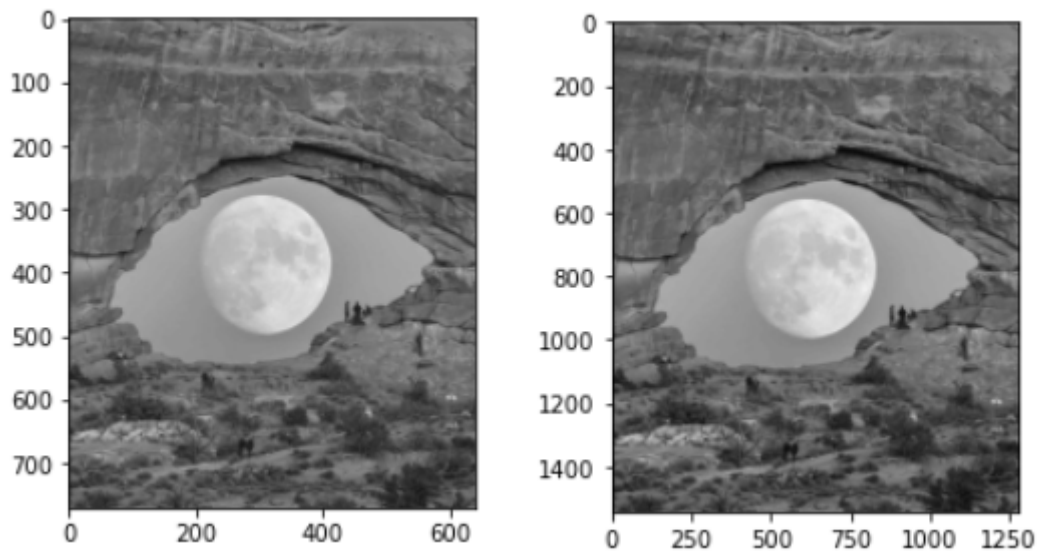
Figure 1. The Gaussian Pyramid to represent the scale space.

2.1.1. 上采样

SIFT的高斯金字塔的第一个octave的第一层表示已经是经过高斯模糊的，为了保留原图的最高频信息，可以在模糊之前对原图进行一次上采样（双线性插值）：

```
/* *****  
/* 函数: double_sample  
/* 函数描述: 对图像进行2倍上采样（图像的双线性插值）  
/* 参数描述:  
/*      img_src: 原图像地址  
/*      img_dst_ptr: 保存上采样图像的地址的指针  
/*      n、m: 图像高、宽，更新为上采样图像的大小  
/* *****/  
void double_sample(const gray_t* img_src, gray_t** img_dst_ptr, int* n, int* m) {  
    int scale_x = 2, scale_y = 2;  
    int nv = *n, mv = *m;  
    int nn = scale_x * nv, nm = scale_y * mv;  
    gray_t* img_dst = new gray_t[nn * nm];  
    *img_dst_ptr = img_dst;  
    for (int dst_x = 0; dst_x < nn; ++dst_x) {  
        for (int dst_y = 0; dst_y < nm; ++dst_y) {  
            // 中心对齐  
            double src_x = (dst_x + 0.5) / scale_x - 0.5;  
            double src_y = (dst_y + 0.5) / scale_y - 0.5;  
            int src_i = int(src_x);  
            int src_j = int(src_y);  
            // 双线性插值 原理参考https://blog.csdn.net/qq\_37577735/article/details/80041586  
            img_dst[dst_x * nm + dst_y] = \  
                (src_i + 1 - src_x) * (src_j + 1 - src_y) * img_src[src_i * mv + src_j] \  
                + (src_i + 1 - src_x) * (src_y - src_j) * img_src[src_i * mv + src_j + 1] \  
                + (src_x - src_i) * (src_j + 1 - src_y) * img_src[(src_i + 1) * mv + src_j] \  
                + (src_x - src_i) * (src_y - src_j) * img_src[(src_i + 1) * mv + src_j + 1];  
        }  
    }  
    *n = nn;  
    *m = nm;  
}
```

- 单元测试：



- 单元测试结果分析：左图为原图，右图为上采样后的图片，可见图片内容一致，而图片的高度、宽度都变为2倍（见横纵坐标），与预期结果一致。

2.1.2. 下采样

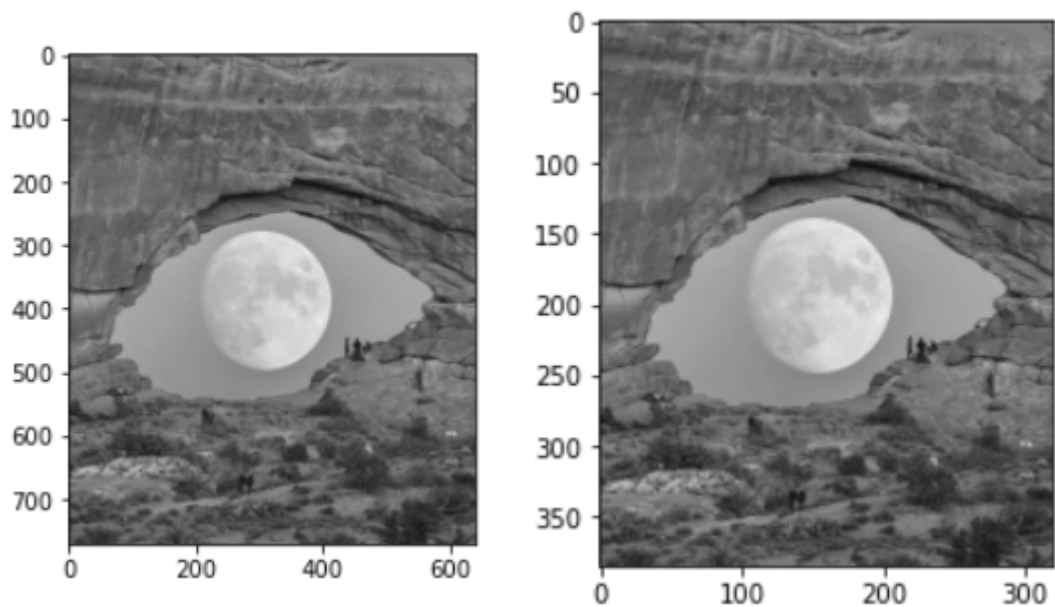
```

/*****
/* 函数: half_sample
/* 函数描述:对图像进行1/2下采样
/* 参数描述:
/*      img_src: 原图像地址
/*      img_dst_ptr: 保存下采样图像的地址的指针
/*      n、m: 图像高、宽，更新为下采样图像的大小
/*****/

void half_sample(const gray_t* img_src, gray_t** img_dst_ptr, int* n, int* m) {
    int nv = *n, mv = *m;
    int nn = nv / 2, nm = mv / 2;
    gray_t* img_dst = new gray_t[nn * nm];
    *img_dst_ptr = img_dst;
    for (int i = 0; i < nn; ++i) {
        for (int j = 0; j < nm; ++j) {
            // SIFT中的1/2下采样方法: 每个维度上每隔两个像素取一个像素
            img_dst[i * nm + j] = img_src[(i << 1) * mv + (j << 1)];
        }
    }
    *n = nn;
    *m = nm;
}

```

- 单元测试：



- 单元测试结果分析：左图为原图，右图为下采样后的图片，可见图片内容一致，而图片的高度、宽度都变为1/2倍（见横纵坐标），与预期结果一致。

2.1.3. 高斯模糊

在高斯模糊的实现中，使用两次一维卷积代替二维卷积，可将复杂度从 $O(m*n*filter_size*filter_size)$ 降为 $O(m*n*filter_size)$ ，当然还有进一步优化的空间，见3.1节。

```

/*****
/* 函数: gaussian_smooth
/* 函数描述:对图像进行高斯模糊，平滑参数为sigma，结果保存到img_dst
/* 参数描述:
/*     img_src: 原图像地址
/*     img_dst_ptr: 保存平滑图像的地址的指针
/*     n、m: 图像高、宽（平滑操作不改变图像大小）
/*     sigma: 高斯模糊参数
/*****/

void gaussian_smooth(const gray_t* img_src, gray_t** img_dst_ptr, int n, int m, double sigma) {
    gray_t* img_dst = new gray_t[n * m];
    *img_dst_ptr = img_dst;

    // 卷积核: 用两次一维卷积分离实现二维卷积 复杂度从 O(m*n*filter_size*filter_size) 降为 O(m*n*filter_size)
    // 1. 根据sigma确定卷积核大小 原理参考https://www.cnblogs.com/shine-lee/p/9671253.html “|1”是为了取邻近的奇数
    int filter_size = int(sigma * 3 * 2 + 1) | 1;
    gray_t* filter = new gray_t[filter_size];

    // 2. 根据高斯分布确定卷积核参数
    int mid = filter_size >> 1;
    double total = 0;
    for (int i = 0; i < filter_size; ++i) {
        filter[i] = 1 / (sqrt(2 * PI) * sigma) * exp((- (i - mid) * (i - mid)) / (2 * sigma * sigma));
        total += filter[i];
    }

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

```

```

        filter[i] /= total;
    }

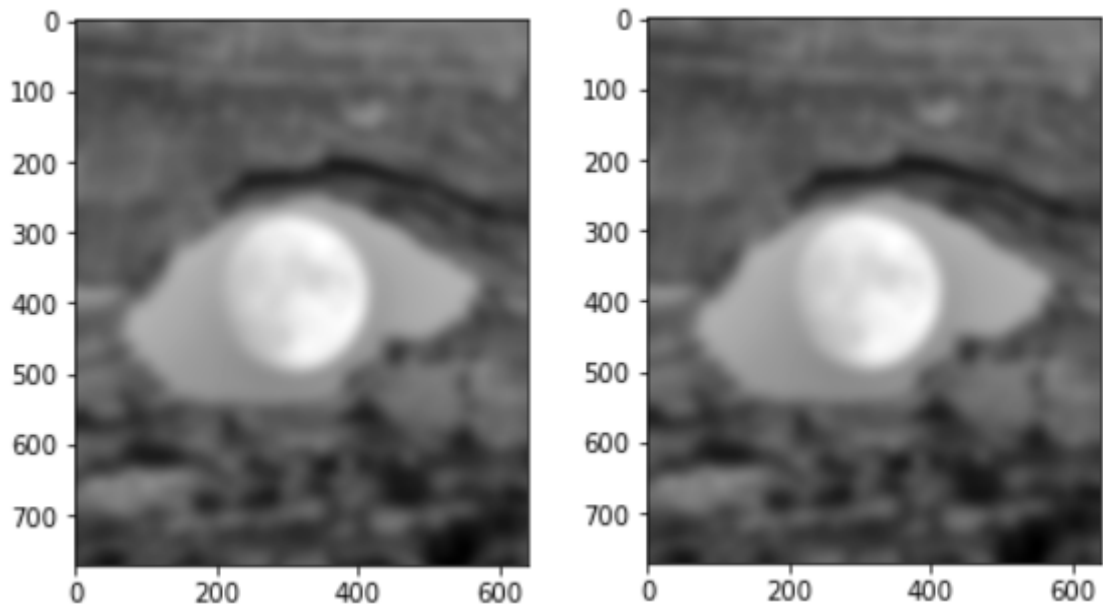
    // 卷积（卷积核越界部分使用边界填充，保持图片大小不变）
    gray_t* temp_res = new gray_t[n * m]; // 存储进行第一维卷积后的结果
    // 1. 进行第一维卷积
    for (int j = 0; j < m; ++j) {
        for (int i = 0; i < n; ++i) {
            int pos = i * m + j;
            temp_res[pos] = 0;
            for (int fi = 0; fi < filter_size; ++fi) {
                int xi = i + (fi - mid);
                xi = xi < 0 ? 0 : xi;
                xi = xi >= n ? n-1 : xi;
                temp_res[pos] += filter[fi] * img_src[xi * m + j];
            }
        }
    }

    // 2. 进行第二维卷积
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            int pos = i * m + j;
            img_dst[pos] = 0;
            for (int fi = 0; fi < filter_size; ++fi) {
                int yi = j + (fi - mid);
                yi = yi < 0 ? 0 : yi;
                yi = yi >= m ? m-1 : yi;
                img_dst[pos] += filter[fi] * temp_res[i * m + yi];
            }
        }
    }

    delete[] filter;
    delete[] temp_res;
}

```

- 单元测试：



- 单元测试结果分析：取 $\sigma=10$ ，左图为上述程序结果，右图为opencv结果，可见图

片内容比较接近 `np.mean(blur-res)`，说明上述实现基本没有问题，且实测当

`0.01623240015636605`

σ 较大时，用两次一维卷积代替一次二维卷积的速度提升很明显。

2.1.4. 高斯金字塔

```

/*****
/* 函数: build_Gauss_pyramid
/* 函数描述: 构建图像的高斯金字塔层
/* 参数描述:
/*     double_sample_img: 输入图像
/*     Gauss_pyramid: 保存高层的vector容器
/*     n,m: 输入图像大小
/*     S: 中间层数(每个octave是S+3层)
/*     sigma_init: 第一层使用的高斯模糊参数
/*****/

void build_Gauss_pyramid(gray_t* gray_img, int n, int m, std::vector<Layer>& Gauss_pyramid, int S,
double sigma_init) {
    // 至少3*3
    double sigma = sigma_init; // 记录下一层的相对于 原始上采样图像 的平滑参数
    double rela_sigma; // 下一层由当前层以rela_sigma的高斯平滑得到 相当于以sigma从 原始上采样图像 的高斯平滑得到
    double s_rt_2 = pow(2, 1.0 / S); // sigma(i+1) = sigma(i) * s_rt_2 注意1/S是整数0
    double s_mul = sqrt(s_rt_2 * s_rt_2 - 1); // rela_sigma = sqrt((sigma(i) * s_rt_2) ^ 2 - sigma(i) ^
2) = sigma(i) * s_mul
    gray_t* cur_img;
    for (int octave = 0; n >= 3 && m >= 3; ++octave) {
        if (octave == 0) {
            // 第一个octave的第一层 将输入图像上采样后进行高斯模糊
            double_sample(gray_img, &gray_img, &n, &m);
            gaussian_smooth(gray_img, &cur_img, n, m, sigma);
            delete[] gray_img; // 删除上采样图像
        }
    }
}

```

```

        gray_img = nullptr;

        Gauss_pyramid.push_back(Layer(cur_img, n, m, sigma));
    }
    else {
        // 第二个及之后octave的第一层 从上一个octave的第S层下采样得到
        int last_oct_S = octave * (S + 3) - 3;
        half_sample(Gauss_pyramid[last_oct_S].img, &cur_img, &n, &m);
        sigma = Gauss_pyramid[last_oct_S].sigma;
        Gauss_pyramid.push_back(Layer(cur_img, n, m, sigma));
    }

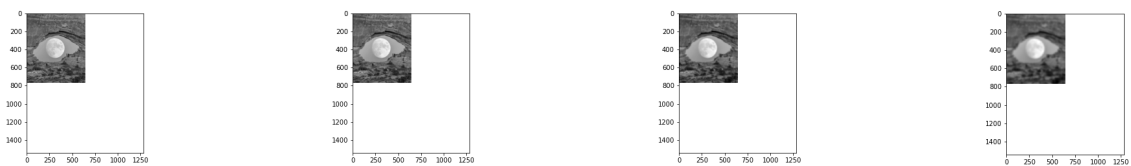
    rela_sigma = sigma * s_mul;
    sigma *= s_rt_2;
    // 每个octave的后续layer
    for (int layer = 1; layer < S + 3; ++layer) {
        gaussian_smooth(cur_img, &cur_img, n, m, rela_sigma);
        Gauss_pyramid.push_back(Layer(cur_img, n, m, sigma));
        rela_sigma = sigma * s_mul;
        sigma *= s_rt_2;
    }
}
}

```

- 单元测试：S=1，即octave大小为S+3=4所构建金字塔的前3个octave



第一个octave



第二个octave



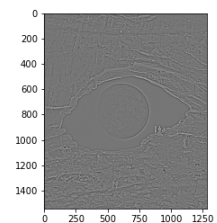
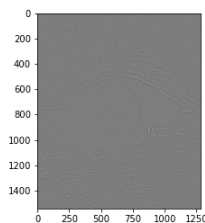
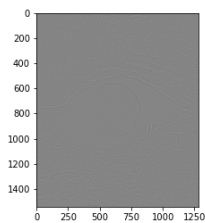
第三个octave

2.1.5. 高斯差分金字塔

```
/* *****
/* 函数: build_DoG_pyramid
/* 函数描述: 构建图像的高斯差分金字塔层
/* 参数描述:
/*      Gauss_pyramid: 输入高斯金字塔层
/*      DoG_pyramid: 保存高斯差分层的vector容器
/*      S: 中间层数 (每个octave是S+3层)
/* *****/

void build_DoG_pyramid(std::vector<Layer>& Gauss_pyramid, std::vector<Layer>& DoG_pyramid, int S) {
    int n_layer = Gauss_pyramid.size();
    for (int layer = 0; layer < n_layer; layer += S + 3) {
        int n = Gauss_pyramid[layer].n, m = Gauss_pyramid[layer].m;
        for (int layeri = layer + 1; layeri < layer + S + 3; ++ layeri) {
            gray_t* dog_img = new gray_t[n * m];
            for (int i = 0; i < n * m; ++i) {
                dog_img[i] = Gauss_pyramid[layeri].img[i] - Gauss_pyramid[layeri - 1].img[i];
            }
            DoG_pyramid.push_back(Layer(dog_img, n, m, Gauss_pyramid[layeri - 1].sigma));
        }
    }
}
```

- 单元测试: S=1时的高斯差分金字塔的第一个octave



2.2. 极值点检测

```
/* *****
/* 函数: detect_keypoints
/* 函数描述: 从高斯差分金字塔中检测极值点
/* 参数描述:
/*      DoG_pyramid: 输入的高斯差分金字塔
/*      keypoints: 保存极值点
/*      S: 中间层数 (每个dog octave是S+2层)
/*      contrast_threshold: 用于低对比度样本点过滤的阈值大小 如0.03
/*      edge_response_threshold: 用于边缘响应样本点过滤的阈值大小 如10
/*      max_interpolation: 最大变换插值次数
/* *****/

void detect_keypoints(std::vector<Layer>& DoG_pyramid, std::vector<KeyPoint>& keypoints, int S, double
contrast_threshold, double edge_response_threshold, int max_interpolation) {
```

```

int n_layer = DoG_pyramid.size();
int octave = 0;

for (int layer = 0; layer < n_layer; layer += S + 2) {
    int n = DoG_pyramid[layer].n, m = DoG_pyramid[layer].m;
    for (int layeri = layer + 1; layeri < layer + S + 1; ++layeri) {
        // 刚好s个中间层
        gray_t* prev_img = DoG_pyramid[layeri - 1].img;
        gray_t* cur_img = DoG_pyramid[layeri].img;
        gray_t* next_img = DoG_pyramid[layeri + 1].img;
        for (int i = 1; i < n-1; ++i) {
            for (int j = 1; j < m-1; ++j) {
                gray_t cur = cur_img[i * m + j];
                int x = 0, y, l;
                // 如果小于它的9+8+9=26个邻居
                if (cur < cur_img[(i - 1) * m + j - 1] && cur < cur_img[(i - 1) * m + j] && cur <
cur_img[(i - 1) * m + j + 1]\
                    && cur < cur_img[i * m + j - 1] && cur < cur_img[i * m + j + 1]\
                    && cur < cur_img[(i + 1) * m + j - 1] && cur < cur_img[(i + 1) * m + j] &&
cur < cur_img[(i + 1) * m + j + 1]) {
                    if (cur < prev_img[(i - 1) * m + j - 1] && cur < prev_img[(i - 1) * m + j] &&
cur < prev_img[(i - 1) * m + j + 1]\
                        && cur < prev_img[i * m + j - 1] && cur < prev_img[i * m + j] && cur <
prev_img[i * m + j + 1]\
                            && cur < prev_img[(i + 1) * m + j - 1] && cur < prev_img[(i + 1) * m +
j] && cur < prev_img[(i + 1) * m + j + 1]) {
                                if (cur < next_img[(i - 1) * m + j - 1] && cur < next_img[(i - 1) * m + j]
&& cur < next_img[(i - 1) * m + j + 1]\
                                    && cur < next_img[i * m + j - 1] && cur < next_img[i * m + j] && cur <
next_img[i * m + j + 1]\
                                        && cur < next_img[(i + 1) * m + j - 1] && cur < next_img[(i + 1) * m +
j] && cur < next_img[(i + 1) * m + j + 1]) {
                                            x = i, y = j, l = layeri - layer;
                                        }
                                    }
                                }
                            }
                        }
                    }
                // 或者大于它的9+8+9=26个邻居
                else if (cur > cur_img[(i - 1) * m + j - 1] && cur > cur_img[(i - 1) * m + j] &&
cur > cur_img[(i - 1) * m + j + 1]\
                    && cur > cur_img[i * m + j - 1] && cur > cur_img[i * m + j + 1]\
                    && cur > cur_img[(i + 1) * m + j - 1] && cur > cur_img[(i + 1) * m + j] &&
cur > cur_img[(i + 1) * m + j + 1]) {
                    if (cur > prev_img[(i - 1) * m + j - 1] && cur > prev_img[(i - 1) * m + j] &&
cur > prev_img[(i - 1) * m + j + 1]\
                        && cur > prev_img[i * m + j - 1] && cur > prev_img[i * m + j] && cur >
prev_img[i * m + j + 1]\
                            && cur > prev_img[(i + 1) * m + j - 1] && cur > prev_img[(i + 1) * m +
j] && cur > prev_img[(i + 1) * m + j + 1]) {
                                if (cur > next_img[(i - 1) * m + j - 1] && cur > next_img[(i - 1) * m + j]
&& cur > next_img[(i - 1) * m + j + 1]\
                                    && cur > next_img[i * m + j - 1] && cur > next_img[i * m + j] && cur >
next_img[i * m + j + 1]\
                                        && cur > next_img[(i + 1) * m + j - 1] && cur > next_img[(i + 1) * m +
j] && cur > next_img[(i + 1) * m + j + 1]) {
                                            x = i, y = j, l = layeri - layer;
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

        && cur > next_img[i * m + j - 1] && cur > next_img[i * m + j] && cur >
next_img[i * m + j + 1]\
        && cur > next_img[(i + 1) * m + j - 1] && cur > next_img[(i + 1) * m +
j] && cur > next_img[(i + 1) * m + j + 1]) {
            x = i, y = j, l = layeri - layer;
        }
    }
}
// 添加极值点
if (x != 0) {
    interpolate_keypoints(DoG_pyramid, S, x, y, l, layer, contrast_threshold,
edge_response_threshold, max_interpolation);
    if (x != 0) {
        keypoints.push_back(KeyPoint(x, y, n, m, octave, l, DoG_pyramid[layer +
1].sigma));
    }
}
}
}
}
}
++octave;
}
}

```

其中的样本点插值函数如下：（其中的三阶矩阵求导经验证，为正确）

```

/*****
/* 函数: interpolate_keypoints
/* 函数描述: 对检测到的极值点进行插值更新、过滤
/* 参数描述:
/*     DoG_pyramid: 输入的高斯差分金字塔
/*     S: 中间层数（每个dog octave是S+2层）
/*     x, y, layeri, layer: 样本点所在二维坐标、所在DoG层的octave内编号、与其所在octave的第一层在金字塔中的编号
/*     contrast_threshold: 用于低对比度样本点过滤的阈值大小 如0.03
/*     edge_response_threshold: 用于边缘响应样本点过滤的阈值大小 如10
/*     max_interpolation: 最大变换插值次数
*****/
void interpolate_keypoints(std::vector<Layer>& DoG_pyramid, int S, int& x, int& y, int& layeri, int
layer, double contrast_threshold, double edge_response_threshold, int max_interpolation) {
    int n = DoG_pyramid[layer + layeri].n, m = DoG_pyramid[layer + layeri].m;
    double ex_val; //极值
    double ratio; // 边缘响应
    double ratio_threshold = (edge_response_threshold + 1) * (edge_response_threshold + 1) /
edge_response_threshold; // 边缘响应阈值
    double fxy1[3]; // 偏移量
    double he[9]; // hessian矩阵
    double he_inv[9]; // hessian矩阵的逆
    double dxy1[3]; // 一阶导
    gray_t* img[3];

```

```

for (int i = 0; i < max_interpolation; ++i) {
    img[0] = DoG_pyramid[layer + layeri - 1].img;
    img[1] = DoG_pyramid[layer + layeri].img;
    img[2] = DoG_pyramid[layer + layeri + 1].img;
    // 计算二阶导 (hessian矩阵) 原理参考https://blog.csdn.net/saltriver/article/details/78990520
    int xy = x * m + y;
    he[0] = img[1][xy - m] + img[1][xy + m] - 2 * img[1][xy]; // Dxx
    he[4] = img[1][xy + 1] + img[1][xy - 1] - 2 * img[1][xy]; // Dyy
    he[8] = img[0][xy] + img[2][xy] - 2 * img[1][xy]; // D11
    he[1] = he[3] = img[1][xy + m + 1] - img[1][xy + 1] - img[1][xy + m] + img[1][xy]; //Dxy
    he[2] = he[6] = img[2][xy + m] - img[2][xy] - img[1][xy + m] + img[1][xy]; //Dx1
    he[5] = he[7] = img[2][xy + 1] - img[2][xy] - img[1][xy + 1] + img[1][xy]; //Dy1
    // 计算hessian矩阵的逆 公式见https://blog.csdn.net/feixia\_24/article/details/41644335
    double det = he[0] * (he[4] * he[8] - he[5] * he[7]) \
        - he[3] * (he[1] * he[8] - he[2] * he[7]) \
        + he[6] * (he[1] * he[5] - he[2] * he[4]);
    // assert det != 0
    he_inv[0] = (he[4] * he[8] - he[5] * he[7]) / det;
    he_inv[1] = (he[2] * he[7] - he[1] * he[8]) / det;
    he_inv[2] = (he[1] * he[5] - he[2] * he[4]) / det;
    he_inv[3] = (he[5] * he[6] - he[3] * he[8]) / det;
    he_inv[4] = (he[0] * he[8] - he[2] * he[6]) / det;
    he_inv[5] = (he[3] * he[2] - he[0] * he[5]) / det;
    he_inv[6] = (he[3] * he[7] - he[4] * he[6]) / det;
    he_inv[7] = (he[1] * he[6] - he[0] * he[7]) / det;
    he_inv[8] = (he[0] * he[4] - he[3] * he[1]) / det;
    // 计算一阶导
    dxyl[0] = img[1][xy + m] - img[1][xy]; // dx
    dxyl[1] = img[1][xy + 1] - img[1][xy]; // dy
    dxyl[2] = img[2][xy] - img[1][xy]; // d1
    // 计算偏移量
    fxyl[0] = - (he_inv[0] * dxyl[0] + he_inv[1] * dxyl[1] + he_inv[2] * dxyl[2]);
    fxyl[1] = - (he_inv[3] * dxyl[0] + he_inv[4] * dxyl[1] + he_inv[5] * dxyl[2]);
    fxyl[2] = - (he_inv[6] * dxyl[0] + he_inv[7] * dxyl[1] + he_inv[8] * dxyl[2]);
    // 计算极值
    ex_val = img[1][xy] + 0.5 * (dxyl[0] * fxyl[0] + dxyl[1] * fxyl[1] + dxyl[2] * fxyl[2]);
    // 计算边缘响应
    ratio = (he[0] + he[4]) * (he[0] + he[4]) / (he[4] * he[0] - he[3] * he[1]);
    // 1. 如果某一维大于0.5 则更新样本点后重新插值
    if (myabs(fxyl[0]) > 0.5 || myabs(fxyl[1]) > 0.5 || myabs(fxyl[2]) > 0.5) {
        x += int(fxyl[0] + 0.5);
        y += int(fxyl[1] + 0.5);
        layeri += int(fxyl[2] + 0.5);
        // 检查是否在合法样本点范围 如果不在 取消该样本点
        if (x < 1 || x > n - 2 || y < 1 || y > m - 2 || layeri < 1 || layeri > S) {
            break;
        }
    }
}
// 2. 否则 如果极值小于阈值 (低对比度) 取消该样本点

```

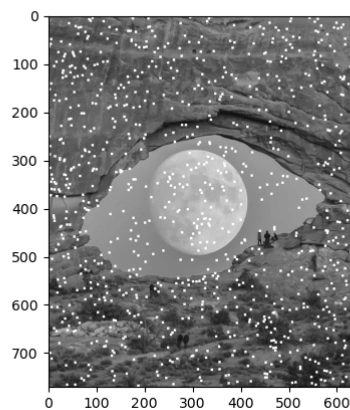
```

        else if (ex_val < contrast_threshold) {
            break;
        }
        // 3. 否则 如果边缘响应过大 取消该样本点
        else if (ratio > ratio_threshold) {
            break;
        }
        // 4. 否则 保留该样本点
        else {
            return;
        }
    }

    x = 0;
}

```

- 单元测试：检测、筛选后的样本点如下



2.3. 方向赋值

```

/*****
/* 函数: assign_orient
/* 函数描述: 为样本点赋值主方向
/* 参数描述:
/*     Gauss_pyramid: 输入的高斯金字塔
/*     keypoints: 已检测到的样本点
/*     S: 金字塔中每隔octave的中间层数
*****/

void assign_orient(std::vector<Layer>& Gauss_pyramid, std::vector<KeyPoint>& keypoints, int S) {
    int old_size = keypoints.size();
    for (int ki = 0; ki < old_size; ++ki) {
        double bins[36] = {0};
        double sigma = 1.5 * keypoints[ki].sigma;
        int layer = keypoints[ki].octave * (S+3) + keypoints[ki].layer;
        gray_t* img = Gauss_pyramid[layer].img;
        int n = Gauss_pyramid[layer].n, m = Gauss_pyramid[layer].m;
        int x = keypoints[ki].x, y = keypoints[ki].y;
    }
}

```

```

int win_radius = (int)(sigma * 3 * 2 + 1) | 1) >> 1; // 高斯权重窗口
// 统计高斯窗口内的梯度方向分布（将360度划分为36个bins），每个像素的梯度权重为其梯度大小乘以高斯权重大小
for (int i = - win_radius; i <= win_radius; ++i) {
    for (int j = - win_radius; j <= win_radius; ++j) {
        int xi = i + x, yj = j + y;
        if (xi > 0 && xi < n - 1 && yj > 0 && yj < m - 1) {
            double dx = img[(xi + 1) * m + yj] - img[(xi - 1) * m + yj];
            double dy = img[xi * m + yj + 1] - img[xi * m + yj - 1];
            double magnitude = sqrt(dx * dx + dy * dy);
            double gaussian = 1 / (2 * PI * sigma * sigma) * exp(-(i * i + j * j) / (2 * sigma
* sigma));

            // dy有可能等于0 需要加上eps数值稳定
            int theta = (int)((atan(dx / (dy + EPS)) * 180 / PI) + 180);
            bins[theta / 10] += gaussian * magnitude;
        }
    }
}

// 梯度值最大的作为主方向
double max_theta_val = bins[0];
int max_theta = 0;
for (int i = 1; i < 36; ++i) {
    if (bins[i] > max_theta_val) {
        max_theta = i * 10; // 单位：度
        max_theta_val = bins[i];
    }
}
keypoints[k].ori = max_theta;

// 增强稳定性：大于主方向的梯度值的80%的那些方向也用于创建新的样本点 分别以这些方向为主方向
for (int i = 1; i < 36; ++i) {
    if (bins[i] > 0.8 * max_theta_val) {
        KeyPoint dup_key(keypoints[k]);
        dup_key.ori = i * 10;
        keypoints.push_back(dup_key);
    }
}
}
}

```

2.4. 特征生成

```

/*****
/* 函数: generate_features
/* 函数描述: 生成样本点的特征描述
/* 参数描述:
/*     Gauss_pyramid: 高斯金字塔
/*     keypoints: 已检测到的样本点
/*     kr: 样本点所在region的大小为kr x kr

```

```

/*      ks: region划分为ks x ks个subregion
/*      ko: 描述每个subregion的方向直方图的bins个数（均匀划分360度）
/*      S: 金字塔中每隔octave的中间层数
/*****/

void generate_features(std::vector<KeyPoint>& keypoints, std::vector<Layer>& Gauss_pyramid, int kr, int
ks, int ko, int S) {
    int key_size = keypoints.size();
    // assert kr % ks = 0

    int rs = kr / ks; // subregion为rsxrs大小
    int radius = kr >> 1; // region的半径 即region为(2radius)x(2radius)
    int f_size = ks * ks * ko; // 特征的维度
    double bs = 360.0 / ko; // 方向直方图每个bin的区间长度
    for (int ki = 0; ki < key_size; ++ki) {
        int layer = keypoints[ki].octave * (S+3) + keypoints[ki].layer;
        gray_t* img = Gauss_pyramid[layer].img;
        int n = keypoints[ki].n, m = keypoints[ki].m;
        int x = keypoints[ki].x, y = keypoints[ki].y;
        int ori = keypoints[ki].ori;
        double sigma = 1.5 * kr;
        double* feature = new double[f_size];
        // 初始化所有直方图
        for (int i = 0; i < f_size; ++i) {
            feature[i] = 0;
        }
        for (int si = 0; si < ks; ++si) {
            for (int sj = 0; sj < ks; ++sj) {
                // 第sub个（行优先）subregion
                double* sub_feature = feature + (si * ks + sj) * ko;
                // 左上角坐标为(xsi, ysj)
                int xsi = x - radius + si * rs;
                int ysj = y - radius + sj * rs;
                // 统计该subregion中的梯度方向信息 所有subregion的直方图拼接为feature
                for (int i = xsi; i < xsi + rs; ++i) {
                    for (int j = ysj; j < ysj + rs; ++j) {
                        if (i > 0 && i < n - 1 && j > 0 && j < m - 1) {
                            // 当前像素的图像梯度
                            double dx = img[(i + 1) * m + j] - img[(i - 1) * m + j];
                            double dy = img[i * m + j + 1] - img[i * m + j - 1];
                            double magnitude = sqrt(dx * dx + dy * dy);
                            // 当前像素的高斯权重
                            int di = i - x, dj = j - y;
                            double gaussian = 1 / (2 * PI * sigma * sigma) * exp(-(di * di + dj * dj) /
(2 * sigma * sigma));

                            // 当前像素的相对于主方向的方向
                            // dy有可能等于0 需要加上eps数值稳定
                            int theta = atan(dx / (dy + EPS)) * 180 / PI + 180 - ori;
                            int b = int(theta / bs + 0.5);
                            sub_feature[b] += magnitude * gaussian;
                        }
                    }
                }
            }
        }
    }
}

```

```

    }

    }

    }

    // 归一化
    double total = 0;
    for (int i = 0; i < f_size; ++i) {
        total += feature[i];
    }
    for (int i = 0; i < f_size; ++i) {
        feature[i] /= total;
    }
    // 限制最大值为0.2
    for (int i = 0; i < f_size; ++i) {
        feature[i] = feature[i] > 0.2 ? 0.2 : feature[i];
    }
    // 再归一化
    total = 0;
    for (int i = 0; i < f_size; ++i) {
        total += feature[i];
    }
    for (int i = 0; i < f_size; ++i) {
        feature[i] /= total;
    }

    keypoints[ki].feature = feature;
    keypoints[ki].feat_len = f_size;
}
}

```

2.5. SIFT调用接口与其他函数

```

/*****
/* 函数: sift
/* 函数描述: 提取图像的sift特征 将其绘制在res_img上
/* 参数描述:
/*     gray_img: 原图像地址, 图像为二维数组, 每个元素为灰度整数值
/*     n、m: 图像高、宽
/*     kr: 特征描述: 样本点所在region的大小为kr x kr
/*     ks: 特征描述: region划分为ks x ks个subregion
/*     ko: 特征描述: 描述每个subregion的方向直方图的bins个数(均匀划分360度)
/*     S: 金字塔: 中间层数(每个octave是S+3层)
/*     sigma_init: 金字塔: 第一层使用的高斯模糊参数
/*     contrast_threshold: 样本点检测: 用于低对比度样本点过滤的阈值大小 如0.03
/*     edge_response_threshold: 样本点检测: 用于边缘响应样本点过滤的阈值大小 如10
/*     max_interpolation: 样本点检测: 最大变换插值次数
/*     time_arr4: 记录每个阶段(金字塔构建、样本点检测、主方向赋值、特征生成)的用时, 单位为秒

```



```

/*****/
extern "C" void sift(gray_t* gray_img, gray_t* res_img, int n, int m, int kr, int ks, int ko, int S,
double sigma_init, double contrast_threshold, double edge_response_threshold, int max_interpolation,
double* time_arr4) {
    // 高斯金字塔
    time_arr4[0] = - get_time();
    // 1. 高斯层
    std::vector<Layer> Gauss_pyramid;
    build_Gauss_pyramid(gray_img, n, m, Gauss_pyramid, S, sigma_init);
    // 2. 高斯差分层
    std::vector<Layer> DoG_pyramid;
    build_DoG_pyramid(Gauss_pyramid, DoG_pyramid, S);
    time_arr4[0] += get_time();
    printf("Build Pyramid: %.3lf s\n", time_arr4[0]);
    // 极值点检测
    time_arr4[1] = - get_time();
    std::vector<KeyPoint> keypoints;
    detect_keypoints(DoG_pyramid, keypoints, S, contrast_threshold, edge_response_threshold,
max_interpolation);
    time_arr4[1] += get_time();
    printf("Keypoints Detect: %.3lf s\n", time_arr4[1]);
    // 主方向提取
    time_arr4[2] = - get_time();
    assign_orient(Gauss_pyramid, keypoints, S);
    time_arr4[2] += get_time();
    printf("Orientation Assignment: %.3lf s\n", time_arr4[2]);
    // 描述生成
    time_arr4[3] = - get_time();
    generate_features(keypoints, Gauss_pyramid, kr, ks, ko, S);
    time_arr4[3] += get_time();
    printf("Descriptor Generation: %.3lf s\n", time_arr4[3]);
    // 绘制结果
    draw_keypoints(gray_img, res_img, n, m, keypoints);
    // 释放空间
    free_space(Gauss_pyramid);
    free_space(DoG_pyramid);
    //free_space_feat(keypoints);
}

```

```

// 其他函数
/*****
/* 函数: draw_keypoints
/* 函数描述: 将样本点绘制在图片上
/* 参数描述:
/*     src_img: 原图
/*     res_img: 结果图
/*     n, m: 图片大小
/*     keypoints: 已检测到的样本点
/*****

```

```

void draw_keypoints(gray_t* src_img, gray_t* res_img, int n, int m, std::vector<KeyPoint>& keypoints);

/*****
/* 函数: free_space
/* 函数描述: 释放金字塔空间
/* 参数描述:
/*     pyramid: 金字塔
*****/

void free_space(std::vector<Layer>& pyramid);

/*****
/* 函数: free_space_feat
/* 函数描述: 释放样本点特征
/* 参数描述:
/*     keypoints: 样本点信息
*****/

void free_space_feat(std::vector<KeyPoint>& keypoints);

/*****
/* 函数: get_time
/* 函数描述: 返回gettimeofday的时间 单位为秒
/* 参数描述:
*****/

double get_time();

```

3. 优化思路

通过测试发现，第一阶段（高斯金字塔的构建）耗时远高于其他阶段，因此我们主要优化该阶段中的操作。

卷积、扫描、规约的gpu实现的单元测试见test_scan.cu文件

3.1. 卷积的CPU优化

在高斯金字塔的构建中，每个octave需要一次上采样或下采样操作，以及S+2次高斯模糊（卷积），而上采样、下采样的复杂度为 $O(nm)$ ，使用两次一维卷积代替二维卷积的优化后的高斯模糊的复杂度仍有 $O(mnk)$ ，其中 k 为卷积核大小，由高斯参数 σ 计算得到：

$$k = \lfloor 6\sigma + 1 \rfloor$$

而第 o 个octave的第一层相对于原始图像的高斯模糊参数为：

$$\sigma_o = 2^o \sigma_0$$

所以卷积中的 k 也随着 o 的增长而指数增长，对此SIFT也采取了措施，即每个octave的图像大小 m, n 都是上一个octave的一半，所以 z 高斯模糊的复杂度 $O(mnk)$ 不会随着 o 的增长而指数增长，而是指数下降。

这虽然意味着我们不需要对卷积操作的复杂度进行数量级上的优化，但是并不妨碍我们对其进行常数级的优化。比如，在每次一维卷积中，当卷积范围在图片中越界时，越界部分都采用边界值 b 填充，假设在一边有 c 个参数 $f[1] \dots f[c]$ 越界，那么本需要计算 $\sum_i^c (f[i] \cdot b)$ ，即 c 次乘法和加法，如果预先计算好 $\sum_i^c f[i]$ ，就只需要一次乘法。此外，由于高斯卷积核的对称性，卷积核只需原始的一半空间。具体优化如下：

```
void gaussian_smooth(const gray_t* img_src, gray_t** img_dst_ptr, int n, int m, double sigma) {
    gray_t* img_dst = new gray_t[n * m];
    *img_dst_ptr = img_dst;

    // 卷积核：用两次一维卷积分离实现二维卷积 复杂度从 O(m*n*filter_size*filter_size) 降为 O(m*n*filter_size)
    // 1. 根据sigma确定卷积核大小 原理参考https://www.cnblogs.com/shine-lee/p/9671253.html “|1”是为了取邻近的奇数
    int filter_size = int(sigma * 3 * 2 + 1) | 1;

    // 2. 根据高斯分布确定卷积核参数
    int mid = filter_size >> 1;
    gray_t* filter = new gray_t[mid + 1]; // 因为高斯卷积核的对称性 所以只存储前半加一个参数
    gray_t* pre_filter = new gray_t[mid + 1]; // pre_filter[i]表示sum(filter[0], ..., filter[i])
    double total = 0;
    for (int i = 0; i < mid + 1; ++i) {
        filter[i] = 1 / (sqrt(2 * PI) * sigma) * exp((- (i - mid) * (i - mid)) / (2 * sigma * sigma));
        total += 2 * filter[i];
    }
    total -= filter[mid];
    for (int i = 0; i < mid + 1; ++i) {
        filter[i] /= total;
    }
    pre_filter[0] = filter[0];
    for (int i = 1; i < mid + 1; ++i) {
        pre_filter[i] = filter[i] + pre_filter[i - 1];
    }

    // 卷积（卷积核越界部分使用边界填充，保持图片大小不变）
    gray_t* temp_res = new gray_t[n * m]; // 存储进行第一维卷积后的结果
    // 1. 进行第一维卷积
    for (int j = 0; j < m; ++j) {
        for (int i = 0; i < n; ++i) {
            int pos = i * m + j;
            temp_res[pos] = 0;
            int i_sta = i - mid;
            int i_end = i + mid;
            // 当前行的卷积范围：[i-mid, i+mid]
            if (i - mid < 0) {
                // 合并计算[i-mid, 0]部分，即原filter的前mid-i个参数与mid-i个填充值（列首元素）的点乘
                temp_res[pos] += pre_filter[mid - i - 1] * img_src[j];
                i_sta = 0;
            }
            if (i + mid >= n) {
                // 合并计算(n-1, i+mid]部分，即原filter的后xx=i+mid+1-n个参数（由于对称性 等价于前xx个）与xx个填充值（行尾元素）的点乘
            }
        }
    }
}
```

```

        temp_res[pos] += pre_filter[i + mid - n] * img_src[(n - 1) * m + j];
        i_end = n - 1;
    }
    for (int xi = i_sta; xi <= i_end; ++xi) {
        // 第xi个元素离卷积中心i的距离为xi-i 使用的是距离卷积核中心mid距离为xi-i的卷积参数
        temp_res[pos] += filter[mid - myabs(i - xi)] * img_src[xi * m + j];
    }
}
}

// 2. 进行第二维卷积
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
        int pos = i * m + j;
        img_dst[pos] = 0;
        int j_sta = j - mid;
        int j_end = j + mid;
        // 当前行的卷积范围: [j-mid, j+mid]
        if (j - mid < 0) {
            // 合并计算[j-mid, 0)部分, 即原filter的前mid-j个参数与mid-j个填充值(行首元素)的点乘
            img_dst[pos] += pre_filter[mid - j - 1] * temp_res[i * m];
            j_sta = 0;
        }
        if (j + mid >= m) {
            // 合并计算(m-1, j+mid]部分, 即原filter的后xx=j+mid+1-m个参数(由于对称性 等价于前xx个)与xx个填充值(行尾元素)的点乘
            img_dst[pos] += pre_filter[j + mid - m] * temp_res[i * m];
            j_end = m - 1;
        }
        for (int yj = j_sta; yj <= j_end; ++yj) {
            // 第yj个元素离卷积中心j的距离为yj-j 使用的是距离卷积核中心mid距离为yj-j的卷积参数
            img_dst[pos] += filter[mid - myabs(j - yj)] * temp_res[i * m + yj];
        }
    }
}

delete[] filter;
delete[] temp_res;
}

```

该优化使单次卷积的复杂度降为 $O(mn \cdot \max(\min(m, k), \min(n, k)))$

3.2. 卷积的CUDA优化思路

卷积在CPU上的上述优化也为CUDA优化提供了便利, 比如, 如果将 n, m 上的外层循环用CUDA并行化, 那么原始算法的内存循环次数 k 过多, 使得单个CUDA线程的任务量过大, 占用资源过多, 从而降低活跃线程数数量, 不足以掩盖延迟。而优化后的算法显著降低了内层循环次数, 因为在前面的octave中, σ 很小, 所以 k 也比较小, 在后面的octave中, n, m 很小, 因

此内层循环次数 $\max(\min(m, k), \min(n, k))$ 始终较小。

3.3. 单个octave构建的OpenMP优化思路

在上述实现的高斯金字塔构建算法中，每一个octave的第一层由上一个octave的第S层下采样得到，因此下一个octave的构建依赖于前一个octave的构建。由于上一个octave的第S层是由上一个octave的第1层不断高斯模糊得到的，所以不能在构建完上一个octave的第S层之前开始构建下一个octave，因此无法并行开始每个octave的构建。

在每个octave内部，第*i*层由第*i* - 1层进行高斯模糊得到（参数为

$\sigma_{oi} = \sqrt{(2^{i/S} \sigma_o)^2 - (2^{(i-1)/S} \sigma_o)^2} = 2^{(i-1)/S} \sigma_o \sqrt{2^{2/S} - 1}$ ），（这里使用了高斯卷积的级联性质，参考⁴）使得其相对于原始上采样图片的模糊参数分别为：

$$\sigma_o, 2^{1/S} \sigma_o, 2^{2/S} \sigma_o, 2^{3/S} \sigma_o, 2^{4/S} \sigma_o, \dots$$

这样每一层的生成都对前一层的生成有依赖，无法并行。

为此，我们可以让octave内部，第*i*层由第1层直接进行高斯模糊得到（参数为

$\sigma_{oi} = \sqrt{(2^{i/S} \sigma_o)^2 - \sigma_o^2} = \sigma_o \sqrt{2^{2i/S} - 1}$ ），这样虽然使得每个 σ_{oi} 增大了（比例为 $\frac{\sqrt{2^{2i/S} - 1}}{\sqrt{2^{2i/S} - 2^{2(i-1)/S}}} = \sqrt{\frac{2^{2/S} - 2^{-2(i-1)/S}}{2^{2/S} - 1}}$ ），使得每次卷积的复杂度增加，但是好处是每一层可以独立生成，由于生成的计算比较复杂，所以可以考虑用OpenMP进行并行，带来几倍加速。我们期待octave内部层数*S*较大时，该并行方法可以实现加速。

```
//每个octave的后续layer: 前后独立实现 每一层直接从octave的第一层模糊得到
for (int layer = 1; layer < S + 3; ++layer) {
    Gauss_pyramid.push_back(Layer(nullptr, n, m, sigma));
    sigma *= s_rt_2;
}

int oct_layer = octave * (S + 3);

for (int layer = 1; layer < S + 3; ++layer) {
    gaussian_smooth(Gauss_pyramid[oct_layer].img, &Gauss_pyramid[oct_layer + layer].img, n, m,
    Gauss_pyramid[oct_layer].sigma * sqrt(pow(2, 2.0 * layer / S) - 1));
}
```

3.4. 显存拷贝优化思路

当我们把卷积计算放到CUDA上时，需要将被卷积的图片拷贝到设备上，将卷积结果拷贝回主机，这样虽然卷积计算的时间减少了，但是额外增加了内存拷贝的时间。仔细分析高斯金字塔构建的过程，可以发现，如果一开始就把图层放到显存上（每个layer记录一个img指针指向显存），那么每次卷积都不需要内存拷贝。这还带来一个好处是其他操作也可以在GPU上加速（而免受内存拷贝带来的负优化），比如我们也将下采样操作在GPU上实现。

不过由于我们只对高斯金字塔的构建进行加速，后续的操作都需要在CPU上进行，所以在高斯金字塔构建完成之后，还需要将所有图层从显存拷贝回来。即使这样，拷贝次数也缩减为原来的一半。如果有机会将后续操作都放在GPU上实现，那么就不用拷贝回来（我们只需要最后的样本点及其特征描述）。拷贝回来的操作如下：

```
void copy_back (std::vector<Layer>& Gauss_pyramid) {
    int g_size = Gauss_pyramid.size();
    for (int gi = 0; gi < g_size; ++gi) {
        gray_t* device_img = Gauss_pyramid[gi].img;
        Gauss_pyramid[gi].img = new gray_t[Gauss_pyramid[gi].n * Gauss_pyramid[gi].m];
        checkCudaErrors(cudaMemcpy(Gauss_pyramid[gi].img, device_img, sizeof(gray_t) *
(Gauss_pyramid[gi].n * Gauss_pyramid[gi].m), cudaMemcpyDeviceToHost));
        checkCudaErrors(cudaFree(device_img));
    }
}
```

3.5. 卷积核计算的规约与扫描优化

前面提到，在后面的octave中，卷积核大小会比较大，此时可以将卷积核参数的计算也用CUDA加速，额外的好处是卷积核可以直接在显存上申请，无需内存拷贝即可用于卷积。

在我们的卷积实现中，需要的卷积核参数及其计算过程、优化思路如下：

- filter：高斯卷积核的前 $\lfloor k/2 \rfloor + 1$ 个参数（后续参数与前一半对称）
 - 计算过程：计算每个位置高斯函数值，对所有高斯函数值求和（由于对称性，和为前 $\lfloor k/2 \rfloor$ 个值的和的两倍加上第 $\lfloor k/2 \rfloor + 1$ 个值），对已有值进行归一化
 - 优化思路：基于CUDA8.pdf的规约算法进行修改

```
__global__ void cal_filter(gray_t* filter, gray_t* total, int mid, double sigma) {
    extern __shared__ gray_t sdata[];

    // 计算filter
    int fi = blockIdx.x * blockDim.x + threadIdx.x; // not unsigned int
    int tid = threadIdx.x;

    if (fi <= mid) {
        filter[fi] = 1 / (sqrt(2 * PI) * sigma) * exp((- (fi - mid) * (fi - mid)) / (2 * sigma *
sigma));
    }

    if (tid == 0) {
        total[0] = 0.0;
    }

    __syncthreads(); // 确保当前线程块所需的filter计算完毕

    // 将filter加载到共享内存
    sdata[tid] = fi <= mid ? filter[fi] : 0;

    __syncthreads();

    // 规约：把当前线程块的filter之和存放到共享内存sdata[0]
```

```

for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
    if (tidx < s) {
        sdata[tidx] += sdata[tidx + s];
    }
    __syncthreads();
}

// 把所有线程块结果用原子加法累加到全局变量total[0]上
if (tidx == 0) {
    atomicAdd(&total, 2.0 * sdata[0]);
}

if (fi == mid) {
    atomicAdd(&total, - filter[mid]);
}
}

__global__ void div_filter(gray_t* filter, gray_t* total, int mid, int loop) {
    // 计算filter
    int fi = blockIdx.x * blockDim.x + threadIdx.x;
    int sta = fi * loop;
    int end = mymin(sta + loop, mid + 1);
    for (int i = sta; i < end; ++i) {
        filter[i] /= total[0];
    }
}

```

```

// 计算filter的完整调用

gray_t* device_total;
checkCudaErrors(cudaMalloc((void **) &device_total, sizeof(gray_t)));

cal_filter<<<diveup((mid + 1), 32), 32, sizeof(gray_t)*32>>>(device_filter, device_total, mid,
sigma);

checkCudaErrors(cudaDeviceSynchronize());
div_filter<<<diveup((mid + 1), 32 * 8), 32>>>(device_filter, device_total, mid, 8);
checkCudaErrors(cudaDeviceSynchronize());
checkCudaErrors(cudaFree(device_total));

```

- pre_filter: filter的前缀和
 - 优化思路: 使用CUDA7.pdf的扫描算法求前缀和

```

// from CUDA7.pdf
// 计算 每个block内部的前缀和 并记录block sum (block内前缀和数组的最后一个值) 的前缀和
__global__ void scan(gray_t* out, gray_t* block_sums, gray_t* data) {
    extern __shared__ gray_t s_data[];
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    s_data[threadIdx.x] = data[tid];
}

```

```

    for (int stride = 1; stride < blockDim.x; stride <= 1) {
        __syncthreads();
        gray_t val = (threadIdx.x >= stride) ? s_data[threadIdx.x - stride] : 0;
        __syncthreads();
        s_data[threadIdx.x] += val;
    }

    out[tid] = s_data[threadIdx.x];
    if (threadIdx.x == 0) {
        for (int i = blockIdx.x + 1; i < gridDim.x; ++i) {
            atomicAdd(&block_sums[i], s_data[blockDim.x - 1]);
        }
    }
}

// 将blocksum前缀和添加到block内部前缀和的每一项
__global__ void scan_update(gray_t* out, gray_t* block_sums) {
    __shared__ gray_t block_sum;
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (threadIdx.x == 0) {
        block_sum = block_sums[blockIdx.x];
    }

    __syncthreads();

    out[idx] += block_sum;
}

```

```

// 计算pre_filter的完整调用

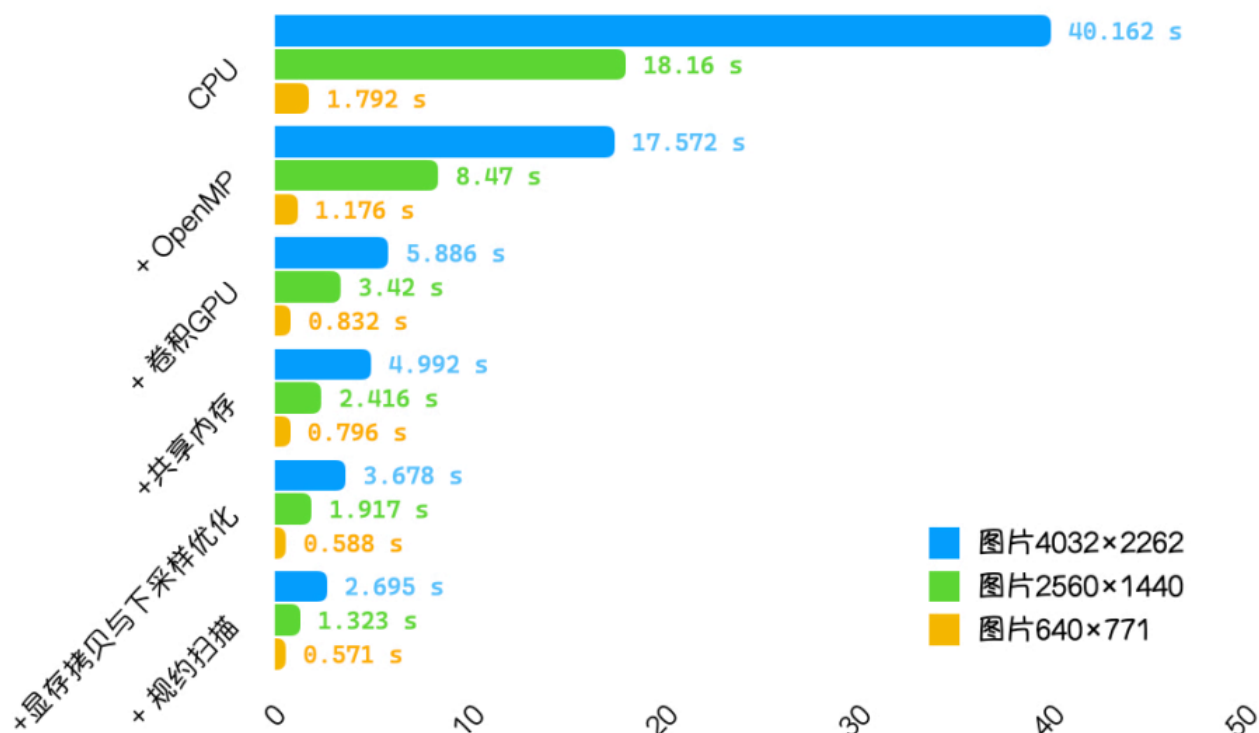
gray_t* block_sums;
int block_num = diveup(mid + 1, 32);
checkCudaErrors(cudaMalloc((void**) &block_sums, sizeof(gray_t) * (block_num)));
scan<<<block_num, 32, 32>>>(device_pre_filter, block_sums, device_filter);
checkCudaErrors(cudaDeviceSynchronize());
scan_update<<<block_num, 32>>>(device_pre_filter, block_sums);
checkCudaErrors(cudaDeviceSynchronize());
checkCudaErrors(cudaFree(block_sums));

```

上述代码中使用了双精度的atomicAdd，在V100上需要用-arch=sm_70编译

3.6. 最终优化效果

SIFT构建金字塔阶段时间对比



1. [ctypes — A foreign function library for Python — Python 3.9.5 documentation](#) ↵
2. D. G. Lowe. Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision, 2004. ↵
3. [SYSU/a lecture note of SIFT.pdf at master · gggxxl/SYSU \(github.com\)](#) ↵
4. [lecture10.pdf \(psu.edu\)](#) ↵