Gaussian Splatting的代码中用到了glm::vec3（三维向量）, glm::vec4（四维向量）, glm::mat3（3×3矩阵）和glm::dot（向量点积）。

主要逻辑写在diff-gaussian-rasterization/cuda\_rasterizer/rasterizer\_impl.cu

实现了：

**4.3 快速可微光栅化（Tile-based Rasterizer）**

渲染方法仅依靠Splatting无法达到较高的实时渲染帧率，本文通过Tile-based Rasterizer

（1）、把整个图像划分为**16\*16**个tiles，每个tile视锥内挑选可视的3D Gaussian；  
（2）、每个视锥内只取执行度大于**99%的高斯，并按深度排序**；（  
（3）、**并行**地在每个tile上splat；  
（4）、有像素的**不透明度达到饱和**就停止对应线程；  
（5）、反向传播误差时按**tile**对高斯进行索引。

取执行度大于99%的高斯后实例化为高斯对象，对象中包含所在tile的ID以及所在对应视域下的深度，通过这些信息对高斯对象进行排序；（将（2）中高斯按其到图像平面的深度值的排序顺序从近到远的tile上做splat，把splat留下的痕迹做堆叠累积直到不透明度饱和即为；每个tile都单独为一个线程块，所以可认为所有tile上的光栅化是并行运行的（光栅化指从堆叠的splat痕迹中去划分像素网格来生成像素值）

(6) 渲染前传 CUDA 实现部分: submodules/diff-gaussian-rasterization/cuda\_rasterizer/rasterizer\_impl.cu中的int CudaRasterizer::Rasterizer::forward函数;

(7) 渲染反传 CUDA 实现部分: submodules/diff-gaussian-rasterization/cuda\_rasterizer/rasterizer\_impl.cu中的void CudaRasterizer::Rasterizer::backward函数;

2. 高斯排序和合成顺序（rasterizer\_impl.cu）

计算每个高斯的前后顺序（Alpha合成）：当多个高斯分布重叠在同一区域时，需要确定它们在图像上的渲染顺序，这通常基于它们距离摄像机的远近来决定。

*/ Forward rendering procedure for differentiable rasterization*

*// of Gaussians.*

**int** CudaRasterizer**::**Rasterizer**::**forward(

std**::**function**<char\*** (size\_t)**>** geometryBuffer,

std**::**function**<char\*** (size\_t)**>** binningBuffer,

std**::**function**<char\*** (size\_t)**>** imageBuffer,

**const** **int** P, **int** D, **int** M,

**const** **float\*** background,

**const** **int** width, **int** height,

**const** **float\*** means3D,

**const** **float\*** shs,

**const** **float\*** colors\_precomp,

**const** **float\*** opacities,

**const** **float\*** scales,

**const** **float** scale\_modifier,

**const** **float\*** rotations,

**const** **float\*** cov3D\_precomp,

**const** **float\*** viewmatrix,

**const** **float\*** projmatrix,

**const** **float\*** cam\_pos,

**const** **float** tan\_fovx, **float** tan\_fovy,

**const** **bool** prefiltered,

**float\*** out\_color,

**int\*** radii,

**bool** debug)

{

**const** **float** focal\_y **=** height **/** (2.0f **\*** tan\_fovy); *// 垂直方向的焦距 focal\_y*

**const** **float** focal\_x **=** width **/** (2.0f **\*** tan\_fovx); *// 水平方向的焦距 focal\_x*

size\_t chunk\_size **=** required**<**GeometryState**>**(P); *// 计算存储所有3D gaussian的各个参数所需要的空间大小*

**char\*** chunkptr **=** geometryBuffer(chunk\_size); *// 给所有3D gaussian的各个参数分配存储空间, 并返回存储空间的指针*

GeometryState geomState **=** GeometryState**::**fromChunk(chunkptr, P); *// 在给定的内存块中初始化 GeometryState 结构体, 为不同成员分配空间，并返回一个初始化的实例*

**if** (radii **==** **nullptr**)

{

radii **=** geomState.internal\_radii; *// 指向radii数据的指针*

}

*// 定义了一个三维网格（dim3 是 CUDA 中定义三维网格维度的数据类型），确定了在水平和垂直方向上需要多少个块来覆盖整个渲染区域*

dim3 **tile\_grid**((width **+** BLOCK\_X **-** 1) **/** BLOCK\_X, (height **+** BLOCK\_Y **-** 1) **/** BLOCK\_Y, 1);

*// 确定了每个块在 X（水平）和 Y（垂直）方向上的线程数*

dim3 **block**(BLOCK\_X, BLOCK\_Y, 1);

*// Dynamically resize image-based auxiliary buffers during training*

size\_t img\_chunk\_size **=** required**<**ImageState**>**(width **\*** height); *// 计算存储所有2D pixel的各个参数所需要的空间大小*

**char\*** img\_chunkptr **=** imageBuffer(img\_chunk\_size); *// 给所有2D pixel的各个参数分配存储空间, 并返回存储空间的指针*

ImageState imgState **=** ImageState**::**fromChunk(img\_chunkptr, width **\*** height); *// 在给定的内存块中初始化 ImageState 结构体, 为不同成员分配空间，并返回一个初始化的实例*

**if** (NUM\_CHANNELS **!=** 3 **&&** colors\_precomp **==** **nullptr**)

{

**throw** std**::**runtime\_error("For non-RGB, provide precomputed Gaussian colors!");

}

*// Run preprocessing per-Gaussian (transformation, bounding, conversion of SHs to RGB)*

CHECK\_CUDA(FORWARD**::**preprocess(

P, D, M, *// 3D gaussian的个数, 球谐函数的次数, 球谐系数的个数 (球谐系数用于表示颜色)*

means3D, *// 每个3D gaussian的XYZ均值*

(glm**::**vec3**\***)scales, *// 每个3D gaussian的XYZ尺度*

scale\_modifier, *// 尺度缩放系数, 1.0*

(glm**::**vec4**\***)rotations, *// 每个3D gaussian的旋转四元组*

opacities, *// 每个3D gaussian的不透明度*

shs, *// 每个3D gaussian的球谐系数, 用于表示颜色*

geomState.clamped, *// 存储每个3D gaussian的R、G、B是否小于0*

cov3D\_precomp, *// 提前计算好的每个3D gaussian的协方差矩阵, []*

colors\_precomp, *// 提前计算好的每个3D gaussian的颜色, []*

viewmatrix, *// 相机外参矩阵, world to camera*

projmatrix, *// 投影矩阵, world to image*

(glm**::**vec3**\***)cam\_pos, *// 所有相机的中心点XYZ坐标*

width, height, *// 图像的宽和高*

focal\_x, focal\_y, *// 水平、垂直方向的焦距*

tan\_fovx, tan\_fovy, *// 水平、垂直视场角一半的正切值*

radii, *// 存储每个2D gaussian在图像上的半径*

geomState.means2D, *// 存储每个2D gaussian的均值*

geomState.depths, *// 存储每个2D gaussian的深度*

geomState.cov3D, *// 存储每个3D gaussian的协方差矩阵*

geomState.rgb, *// 存储每个2D pixel的颜色*

geomState.conic\_opacity, *// 存储每个2D gaussian的协方差矩阵的逆矩阵以及它的不透明度*

tile\_grid, *// 在水平和垂直方向上需要多少个块来覆盖整个渲染区域*

geomState.tiles\_touched, *// 存储每个2D gaussian覆盖了多少个tile*

prefiltered *// 是否预先过滤掉了中心点(均值XYZ)不在视锥（frustum）内的3D gaussian*

), debug)

*// Compute prefix sum over full list of touched tile counts by Gaussians*

*// E.g., [2, 3, 0, 2, 1] -> [2, 5, 5, 7, 8]*

CHECK\_CUDA(cub**::**DeviceScan**::**InclusiveSum(geomState.scanning\_space, geomState.scan\_size, geomState.tiles\_touched, geomState.point\_offsets, P), debug)

*// Retrieve total number of Gaussian instances to launch and resize aux buffers*

**int** num\_rendered; *// 存储所有的2D gaussian总共覆盖了多少个tile*

*// 将 geomState.point\_offsets 数组中最后一个元素的值复制到主机内存中的变量 num\_rendered*

CHECK\_CUDA(cudaMemcpy(**&**num\_rendered, geomState.point\_offsets **+** P **-** 1, **sizeof**(**int**), cudaMemcpyDeviceToHost), debug);

size\_t binning\_chunk\_size **=** required**<**BinningState**>**(num\_rendered);

**char\*** binning\_chunkptr **=** binningBuffer(binning\_chunk\_size);

BinningState binningState **=** BinningState**::**fromChunk(binning\_chunkptr, num\_rendered);

*// 将每个3D gaussian的对应的tile index和深度存到point\_list\_keys\_unsorted中*

*// 将每个3D gaussian的对应的index（第几个3D gaussian）存到point\_list\_unsorted中*

*// For each instance to be rendered, produce adequate [ tile | depth ] key*

*// and corresponding dublicated Gaussian indices to be sorted*

duplicateWithKeys **<<** **<**(P **+** 255) **/** 256, 256 **>>** **>** (

P,

geomState.means2D,

geomState.depths,

geomState.point\_offsets,

binningState.point\_list\_keys\_unsorted,

binningState.point\_list\_unsorted,

radii,

tile\_grid)

CHECK\_CUDA(, debug)

**int** bit **=** getHigherMsb(tile\_grid.x **\*** tile\_grid.y);

*// 对一个键值对列表进行排序。这里的键值对由 binningState.point\_list\_keys\_unsorted 和 binningState.point\_list\_unsorted 组成*

*// 排序后的结果存储在 binningState.point\_list\_keys 和 binningState.point\_list 中*

*// binningState.list\_sorting\_space 和 binningState.sorting\_size 指定了排序操作所需的临时存储空间和其大小*

*// num\_rendered 是要排序的元素总数。0, 32 + bit 指定了排序的最低位和最高位，这里用于确保排序考虑到了足够的位数，以便正确处理所有的键值对*

*// Sort complete list of (duplicated) Gaussian indices by keys*

CHECK\_CUDA(cub**::**DeviceRadixSort**::**SortPairs(

binningState.list\_sorting\_space,

binningState.sorting\_size,

binningState.point\_list\_keys\_unsorted, binningState.point\_list\_keys,

binningState.point\_list\_unsorted, binningState.point\_list,

num\_rendered, 0, 32 **+** bit), debug)

*// 将 imgState.ranges 数组中的所有元素设置为 0*

CHECK\_CUDA(cudaMemset(imgState.ranges, 0, tile\_grid.x **\*** tile\_grid.y **\*** **sizeof**(uint2)), debug);

*// 识别每个瓦片（tile）在排序后的高斯ID列表中的范围*

*// 目的是确定哪些高斯ID属于哪个瓦片，并记录每个瓦片的开始和结束位置*

*// Identify start and end of per-tile workloads in sorted list*

**if** (num\_rendered **>** 0)

identifyTileRanges **<<** **<**(num\_rendered **+** 255) **/** 256, 256 **>>** **>** (

num\_rendered,

binningState.point\_list\_keys,

imgState.ranges);

CHECK\_CUDA(, debug)

*// Let each tile blend its range of Gaussians independently in parallel*

**const** **float\*** feature\_ptr **=** colors\_precomp **!=** **nullptr** **?** colors\_precomp : geomState.rgb;

CHECK\_CUDA(FORWARD**::**render(

tile\_grid, *// 在水平和垂直方向上需要多少个块来覆盖整个渲染区域*

block, *// 每个块在 X（水平）和 Y（垂直）方向上的线程数*

imgState.ranges, *// 每个瓦片（tile）在排序后的高斯ID列表中的范围*

binningState.point\_list, *// 排序后的3D gaussian的id列表*

width, height, *// 图像的宽和高*

geomState.means2D, *// 每个2D gaussian在图像上的中心点位置*

feature\_ptr, *// 每个3D gaussian对应的RGB颜色*

geomState.conic\_opacity, *// 每个2D gaussian的协方差矩阵的逆矩阵以及它的不透明度*

imgState.accum\_alpha, *// 渲染过程后每个像素的最终透明度或透射率值*

imgState.n\_contrib, *// 每个pixel的最后一个贡献的2D gaussian是谁*

background, *// 背景颜色*

out\_color), debug) *// 输出图像*

**return** num\_rendered;

}

*// Generates one key/value pair for all Gaussian / tile overlaps.*

*// Run once per Gaussian (1:N mapping).*

\_\_global\_\_ **void** **duplicateWithKeys**(

**int** P,

**const** float2**\*** points\_xy,

**const** **float\*** depths,

**const** **uint32\_t\*** offsets,

**uint64\_t\*** gaussian\_keys\_unsorted,

**uint32\_t\*** gaussian\_values\_unsorted,

**int\*** radii,

dim3 grid)

{

**auto** idx **=** cg**::**this\_grid().thread\_rank();

**if** (idx **>=** P)

**return**;

*// Generate no key/value pair for invisible Gaussians*

**if** (radii[idx] **>** 0)

{

*// Find this Gaussian's offset in buffer for writing keys/values.*

**uint32\_t** off **=** (idx **==** 0) **?** 0 **:** offsets[idx **-** 1];

uint2 rect\_min, rect\_max;

getRect(points\_xy[idx], radii[idx], rect\_min, rect\_max, grid);

*// For each tile that the bounding rect overlaps, emit a*

*// key/value pair. The key is | tile ID | depth |,*

*// and the value is the ID of the Gaussian. Sorting the values*

*// with this key yields Gaussian IDs in a list, such that they*

*// are first sorted by tile and then by depth.*

**for** (**int** y **=** rect\_min.y; y **<** rect\_max.y; y**++**)

{

**for** (**int** x **=** rect\_min.x; x **<** rect\_max.x; x**++**)

{

**uint64\_t** key **=** y **\*** grid.x **+** x;

key **<<=** 32;

key **|=** **\***((**uint32\_t\***)**&**depths[idx]);

gaussian\_keys\_unsorted[off] **=** key;

gaussian\_values\_unsorted[off] **=** idx;

off**++**;

}

}

}

}

*// 计算当前的2D gaussian落在哪几个tile上*

\_\_forceinline\_\_ \_\_device\_\_ **void** **getRect**(**const** float2 p, **int** max\_radius, uint2**&** rect\_min, uint2**&** rect\_max, dim3 grid)

{

rect\_min **=** {

min(grid.x, max((**int**)0, (**int**)((p.x **-** max\_radius) **/** BLOCK\_X))),

min(grid.y, max((**int**)0, (**int**)((p.y **-** max\_radius) **/** BLOCK\_Y)))

};

rect\_max **=** {

min(grid.x, max((**int**)0, (**int**)((p.x **+** max\_radius **+** BLOCK\_X **-** 1) **/** BLOCK\_X))),

min(grid.y, max((**int**)0, (**int**)((p.y **+** max\_radius **+** BLOCK\_Y **-** 1) **/** BLOCK\_Y)))

};

}

*// 寻找给定无符号整数 n 的最高有效位（Most Significant Bit, MSB）的下一个最高位*

*// Helper function to find the next-highest bit of the MSB*

*// on the CPU.*

**uint32\_t** **getHigherMsb**(**uint32\_t** n)

{

**uint32\_t** msb **=** **sizeof**(n) **\*** 4;

**uint32\_t** step **=** msb;

**while** (step **>** 1)

{

step **/=** 2;

**if** (n **>>** msb)

msb **+=** step;

**else**

msb **-=** step;

}

**if** (n **>>** msb)

msb**++**;

**return** msb;

}

*// 识别每个瓦片（tile）在排序后的高斯ID列表中的范围*

*// 目的是确定哪些高斯ID属于哪个瓦片，并记录每个瓦片的开始和结束位置*

*// Check keys to see if it is at the start/end of one tile's range in*

*// the full sorted list. If yes, write start/end of this tile.*

*// Run once per instanced (duplicated) Gaussian ID.*

\_\_global\_\_ **void** **identifyTileRanges**(**int** L, **uint64\_t\*** point\_list\_keys, uint2**\*** ranges)

{

**auto** idx **=** cg**::**this\_grid().thread\_rank();

**if** (idx **>=** L)

**return**;

*// Read tile ID from key. Update start/end of tile range if at limit.*

**uint64\_t** key **=** point\_list\_keys[idx];

**uint32\_t** currtile **=** key **>>** 32;

**if** (idx **==** 0)

ranges[currtile].x **=** 0;

**else**

{

**uint32\_t** prevtile **=** point\_list\_keys[idx **-** 1] **>>** 32;

**if** (currtile **!=** prevtile)

{

ranges[prevtile].y **=** idx;

ranges[currtile].x **=** idx;

}

}

**if** (idx **==** L **-** 1)

ranges[currtile].y **=** L;

}

// Generates one key/value pair for all Gaussian / tile overlaps.

// Run once per Gaussian (1:N mapping).

\_\_global\_\_ void duplicateWithKeys(

int P,

const float2\* points\_xy,

const float\* depths,

const uint32\_t\* offsets,

uint64\_t\* gaussian\_keys\_unsorted,

uint32\_t\* gaussian\_values\_unsorted,

int\* radii,

dim3 grid)

{

auto idx = cg::this\_grid().thread\_rank(); // 线程索引，该显线程处理第idx个Gaussian

if (idx >= P)

return;

// Generate no key/value pair for invisible Gaussians

if (radii[idx] > 0)

{

// Find this Gaussian's offset in buffer for writing keys/values.

uint32\_t off = (idx == 0) ? 0 : offsets[idx - 1];

uint2 rect\_min, rect\_max;

getRect(points\_xy[idx], radii[idx], rect\_min, rect\_max, grid);

// 因为要给Gaussian覆盖的每个tile生成一个(key, value)对，

// 所以先获取它占了哪些tile

// For each tile that the bounding rect overlaps, emit a

// key/value pair. The key is | tile ID | depth |,

// and the value is the ID of the Gaussian. Sorting the values

// with this key yields Gaussian IDs in a list, such that they

// are first sorted by tile and then by depth.

for (int y = rect\_min.y; y < rect\_max.y; y++)

{

for (int x = rect\_min.x; x < rect\_max.x; x++)

{

uint64\_t key = y \* grid.x + x; // tile的ID

key <<= 32; // 放在高位

key |= \*((uint32\_t\*)&depths[idx]); // 低位是深度

gaussian\_keys\_unsorted[off] = key;

gaussian\_values\_unsorted[off] = idx;

off++; // 数组中的偏移量

}

}

}

}

// Check keys to see if it is at the start/end of one tile's range in

// the full sorted list. If yes, write start/end of this tile.

// Run once per instanced (duplicated) Gaussian ID.

\_\_global\_\_ void identifyTileRanges(

{

auto idx = cg::this\_grid().thread\_rank();

if (idx >= L)

return;

// Read tile ID from key. Update start/end of tile range if at limit.

uint64\_t key = point\_list\_keys[idx];

uint32\_t currtile = key >> 32; // 当前tile

if (idx == 0)

ranges[currtile].x = 0; // 边界条件：tile 0的起始位置

else

{

uint32\_t prevtile = point\_list\_keys[idx - 1] >> 32;

if (currtile != prevtile)

// 上一个元素和我处于不同的tile，

// 那我是上一个tile的终止位置和我所在tile的起始位置

{

ranges[prevtile].y = idx;

ranges[currtile].x = idx;

}

}

if (idx == L - 1)

ranges[currtile].y = L; // 边界条件：最后一个tile的终止位置

}

// Perform initial steps for each Gaussian prior to rasterization.

template<int C>

\_\_global\_\_ void preprocessCUDA(int P, int D, int M,

const float\* orig\_points, // 也就是CudaRasterizer::Rasterizer::forward的means3D

const glm::vec3\* scales,

const float scale\_modifier,

const glm::vec4\* rotations,

const float\* opacities,

const float\* shs,

bool\* clamped,

const float\* cov3D\_precomp,

const float\* colors\_precomp,

const float\* viewmatrix,

const float\* projmatrix,

const glm::vec3\* cam\_pos,

const int W, int H,

const float tan\_fovx, float tan\_fovy,

const float focal\_x, float focal\_y,

int\* radii,

/\*上面这些参数的含义都提到过\*/

float2\* points\_xy\_image, // Gaussian中心在图像上的像素坐标

float\* depths, // Gaussian中心的深度，即其在相机坐标系的z轴的坐标

float\* cov3Ds, // 三维协方差矩阵

float\* rgb, // 根据球谐算出的RGB颜色值

float4\* conic\_opacity, // 椭圆对应二次型的矩阵和不透明度的打包存储

const dim3 grid, // tile的在x、y方向上的数量

uint32\_t\* tiles\_touched, // Gaussian覆盖的tile数量

bool prefiltered)

{

auto idx = cg::this\_grid().thread\_rank(); // 该函数预处理第idx个Gaussian

if (idx >= P)

return;

// Initialize radius and touched tiles to 0. If this isn't changed,

// this Gaussian will not be processed further.

radii[idx] = 0;

tiles\_touched[idx] = 0;

// Perform near culling, quit if outside.

float3 p\_view; // Gaussian中心在相机坐标系下的坐标

if (!in\_frustum(idx, orig\_points, viewmatrix, projmatrix, prefiltered, p\_view))

return; // 不在相机的视锥内就不管了

// Transform point by projecting

float3 p\_orig = { orig\_points[3 \* idx], orig\_points[3 \* idx + 1], orig\_points[3 \* idx + 2] };

float4 p\_hom = transformPoint4x4(p\_orig, projmatrix); // homogeneous coordinates（齐次坐标）

float p\_w = 1.0f / (p\_hom.w + 0.0000001f); // 想要除以p\_hom.w从而转成正常的3D坐标，这里防止除零

float3 p\_proj = { p\_hom.x \* p\_w, p\_hom.y \* p\_w, p\_hom.z \* p\_w };

// If 3D covariance matrix is precomputed, use it, otherwise compute

// from scaling and rotation parameters.

const float\* cov3D;

if (cov3D\_precomp != nullptr)

{

cov3D = cov3D\_precomp + idx \* 6;

}

else

{

computeCov3D(scales[idx], scale\_modifier, rotations[idx], cov3Ds + idx \* 6);

cov3D = cov3Ds + idx \* 6;

}

// Compute 2D screen-space covariance matrix

float3 cov = computeCov2D(p\_orig, focal\_x, focal\_y, tan\_fovx, tan\_fovy, cov3D, viewmatrix);

// Invert covariance (EWA algorithm)

float det = (cov.x \* cov.z - cov.y \* cov.y); // 二维协方差矩阵的行列式

if (det == 0.0f)

return;

float det\_inv = 1.f / det; // 行列式的逆

float3 conic = { cov.z \* det\_inv, -cov.y \* det\_inv, cov.x \* det\_inv };

// conic是cone的形容词，意为“圆锥的”。猜测这里是指圆锥曲线（椭圆）。

// 二阶矩阵求逆口诀：“主对调，副相反”。

// Compute extent in screen space (by finding eigenvalues of

// 2D covariance matrix). Use extent to compute a bounding rectangle

// of screen-space tiles that this Gaussian overlaps with. Quit if

// rectangle covers 0 tiles.

float mid = 0.5f \* (cov.x + cov.z);

float lambda1 = mid + sqrt(max(0.1f, mid \* mid - det));

float lambda2 = mid - sqrt(max(0.1f, mid \* mid - det));

// 韦达定理求二维协方差矩阵的特征值

float my\_radius = ceil(3.f \* sqrt(max(lambda1, lambda2)));

// 原理见代码后面我的补充说明

// 这里就是截取Gaussian的中心部位（3σ原则），只取像平面上半径为my\_radius的部分

float2 point\_image = { ndc2Pix(p\_proj.x, W), ndc2Pix(p\_proj.y, H) };

// ndc2Pix(v, S) = ((v + 1) \* S - 1) / 2 = (v + 1) / 2 \* S - 0.5

uint2 rect\_min, rect\_max;

getRect(point\_image, my\_radius, rect\_min, rect\_max, grid);

// 检查该Gaussian在图片上覆盖了哪些tile（由一个tile组成的矩形表示）

if ((rect\_max.x - rect\_min.x) \* (rect\_max.y - rect\_min.y) == 0)

return; // 不与任何tile相交，不管了

// If colors have been precomputed, use them, otherwise convert

// spherical harmonics coefficients to RGB color.

if (colors\_precomp == nullptr)

{

glm::vec3 result = computeColorFromSH(idx, D, M, (glm::vec3\*)orig\_points, \*cam\_pos, shs, clamped);

rgb[idx \* C + 0] = result.x;

rgb[idx \* C + 1] = result.y;

rgb[idx \* C + 2] = result.z;

}

// Store some useful helper data for the next steps.

depths[idx] = p\_view.z; // 深度，即相机坐标系的z轴

radii[idx] = my\_radius; // Gaussian在像平面坐标系下的半径

points\_xy\_image[idx] = point\_image; // Gaussian中心在图像上的像素坐标

// Inverse 2D covariance and opacity neatly pack into one float4

conic\_opacity[idx] = { conic.x, conic.y, conic.z, opacities[idx] };

tiles\_touched[idx] = (rect\_max.y - rect\_min.y) \* (rect\_max.x - rect\_min.x);

// Gaussian覆盖的tile数量

}