# 2. 前向渲染

forward.cu里包含了前向渲染函数，按照论文设计，主要包含两个部分：1. 对每个gaussian进行预处理，得到他在当前的相机下的半径、覆盖了哪些tile。2. 对每一个tile，使用splatting技术渲染得到RGB

## 2.1 预处理

 \* 结构：

 \* 定义预处理

 \* 定义渲染

 \* 调用预处理

 \* 调用渲染

**1. 预处理和投影（preprocessCUDA）**

**计算投影圆圈的半径**：在3D空间中的高斯分布投影到2D图像平面时，它通常会形成一个圆圈（实际上是椭圆，因为视角的影响）。这个步骤涉及计算这个圆圈的半径。

**计算圆圈覆盖的像素数**：这涉及到将图像平面分成许多小块（tiles），并计算每个高斯分布投影形成的圆圈与哪些小块相交。这是为了高效地渲染，只更新受影响的小块。

preprocessCUDA函数是前向渲染的核心部分，用于预处理每个gaussian。下面是该函数的执行流程：

1. 每个线程首先获取其在 CUDA 栅格中的线程索引 `idx`，用于确定要处理的gaussian的索引。初始化radii和tiles\_touched，用于跟踪半径和被覆盖的tile数

2. 用in\_frustum检查gaussian是否在视锥内，如果不在视锥内则不进行后续处理。

3. 进行坐标变换，得到图像上的中心坐标（投影坐标）、2D屏幕空间的协方差矩阵，协方差矩阵的逆

获得椭圆半径

4. 计算屏幕空间的像素覆盖范围（计算当前的2D gaussian落在哪几个tile上）

5. 计算和存储其他信息，包括：颜色（从球谐系数生成），深度，透明度

depths[idx] = p\_view.z;

radii[idx] = my\_radius;半径

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

conic\_opacity[idx] = { conic.x, conic.y, conic.z, opacities[idx] };  /逆2D协方差和不透明度巧妙地组合成

tiles\_touched[idx] //碰到了哪些tile

1. 预处理和投影（preprocessCUDA）

计算投影圆圈的半径：在3D空间中的高斯分布投影到2D图像平面时，它通常会形成一个圆圈（实际上是椭圆，因为视角的影响）。这个步骤涉及计算这个圆圈的半径。

计算圆圈覆盖的像素数：这涉及到将图像平面分成许多小块（tiles），并计算每个高斯分布投影形成的圆圈与哪些小块相交。这是为了高效地渲染，只更新受影响的小块。

渲染（renderCUDA）

计算每个像素的颜色：这是最终的渲染步骤。图像被分割成很多图块，每个图块由多个线程处理，每个线程负责一个像素。在这个过程中，每个像素的颜色是通过考虑所有影响该像素的高斯分布来计算的。

## 2.2 渲染

renderCUDA函数用于执行主要的光栅化操作。以下是执行流程：

1.每个线程块（block）代表一个tile，确定当前block所对应的tile的位置范围，即 pix\_min 和 pix\_max。每个线程块将处理该范围内的像素。

+加载点云数据处理范围：

2. 初始化用于跟踪渲染数据的变量，包括透明度 T、贡献像素数 contributor、最后一个贡献者的编号 last\_contributor 和颜色通道数据 C。

3. 将与当前线程块处理的像素相关的高斯函数数据从全局内存中复制到共享内存中，以便后续线程可以更快速地访问这些数据进行渲染计算。这有助于减少对全局内存的访问，提高了渲染操作的效率。这个操作非常细节，值得学习。

4. 遍历当前要处理的像素，这里就是论文中对应的计算过程了，也和nerf的volume render过程接近：

- （计算点在屏幕上的位置与当前像素的距离，并基于协方差矩阵进行采样。）

- 计算点的透明度（alpha）以及最新的 T 值。

- 如果透明度小于某个阈值，认为该点不再影响像素的颜色，退出循环。

- 更新颜色通道 C，（考虑透明度和颜色数据。）

- 更新 T 值，（考虑透明度。）

5. 最后，每个线程块将计算出的颜色通道数据存储到输出缓冲区 out\_color，并将最终的 T 值存储到 final\_T中，同时记录有多少个高斯函数贡献到该像素。

3. 渲染（renderCUDA）

计算每个像素的颜色：这是最终的渲染步骤。图像被分割成很多图块，每个图块由多个线程处理，每个线程负责一个像素。在这个过程中，每个像素的颜色是通过考虑所有影响该像素的高斯分布来计算的。

下面详细看看renderCUDA。其核心逻辑如下

通过计算当前线程所属的 tile 的范围，确定当前线程要处理的像素区域。

判断当前线程是否在有效像素范围内，如果不在，则将 done 设置为 true，表示该线程不执行渲染操作。

使用 \_\_syncthreads\_count 函数，统计当前块内 done 变量为 true 的线程数，如果全部线程都完成，跳出循环。

在每个迭代中，从全局内存中收集每个线程块对应的范围内的数据，包括点的索引、2D 坐标和锥体参数透明度。

对当前线程块内的每个点，进行基于锥体参数的渲染，计算贡献并更新颜色。

所有线程处理完毕后，将渲染结果写入 final\_T、n\_contrib 和 out\_color。

**template<int** C**>**

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

**const** **float\*** orig\_points,

**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,

float2**\*** points\_xy\_image,

**float\*** depths,

**float\*** cov3Ds,

**float\*** rgb,

float4**\*** conic\_opacity,

**const** dim3 grid,

**uint32\_t\*** tiles\_touched,

**bool** prefiltered)

{

*// 每个线程处理一个3D gaussian, index超过3D gaussian总数的线程直接返回, 防止数组越界访问*

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

**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;

*// 判断当前处理的3D gaussian的中心点(均值XYZ)是否在视锥（frustum）内, 如果不在则直接返回*

*// Perform near culling, quit if outside.*

float3 p\_view; *// 用于存储将 p\_orig 通过视图矩阵 viewmatrix 转换到视图空间后的点坐标*

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

**return**;

*// Transform point by projecting*

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

*// 将当前3D gaussian的中心点从世界坐标系投影到裁剪坐标系*

float4 p\_hom **=** transformPoint4x4(p\_orig, projmatrix);

**float** p\_w **=** 1.0f **/** (p\_hom.w **+** 0.0000001f);

*// 将当前3D gaussian的中心点从裁剪坐标转变到归一化设备坐标（Normalized Device Coordinates, NDC）*

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**

{

*// 根据当前3D gaussian的尺度和旋转参数计算其对应的协方差矩阵*

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

cov3D **=** cov3Ds **+** idx **\*** 6;

}

*// 将当前的3D gaussian投影到2D图像，得到对应的2D gaussian的协方差矩阵cov*

*// Compute 2D screen-space covariance matrix*

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

*// 计算当前2D gaussian的协方差矩阵cov的逆矩阵*

*// Invert covariance (EWA algorithm)*

**float** det **=** (cov.x **\*** cov.z **-** cov.y **\*** cov.y);

**if** (det **==** 0.0f)

**return**;

**float** det\_inv **=** 1.f **/** det;

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

*// 计算2D gaussian的协方差矩阵cov的特征值lambda1, lambda2, 从而计算2D gaussian的最大半径*

*// 对协方差矩阵进行特征值分解时，可以得到描述分布形状的主轴（特征向量）以及这些轴上分布的宽度（特征值）*

*// 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));

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

*// 将归一化设备坐标（Normalized Device Coordinates, NDC）转换为像素坐标*

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

uint2 rect\_min, rect\_max;

*// 计算当前的2D gaussian落在哪几个tile上*

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

*// 如果没有命中任何一个title则直接返回*

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

**return**;

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

*// spherical harmonics coefficients to RGB color.*

**if** (colors\_precomp **==** **nullptr**)

{

*// 从每个3D gaussian对应的球谐系数中计算对应的颜色*

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;

radii[idx] **=** my\_radius;

points\_xy\_image[idx] **=** point\_image;

*// 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);

}

*// 根据当前3D gaussian的尺度和旋转参数计算其对应的协方差矩阵*

*// Forward method for converting scale and rotation properties of each*

*// Gaussian to a 3D covariance matrix in world space. Also takes care*

*// of quaternion normalization.*

\_\_device\_\_ **void** **computeCov3D**(**const** glm**::**vec3 scale, **float** mod, **const** glm**::**vec4 rot, **float\*** cov3D)

{

*// Create scaling matrix*

glm**::**mat3 S **=** glm**::**mat3(1.0f); *// 初始化了一个3x3的单位阵*

S[0][0] **=** mod **\*** scale.x;

S[1][1] **=** mod **\*** scale.y;

S[2][2] **=** mod **\*** scale.z;

*// Normalize quaternion to get valid rotation*

glm**::**vec4 q **=** rot;*// / glm::length(rot);*

**float** r **=** q.x;

**float** x **=** q.y;

**float** y **=** q.z;

**float** z **=** q.w;

*// Compute rotation matrix from quaternion*

glm**::**mat3 R **=** glm**::**mat3(

1.f **-** 2.f **\*** (y **\*** y **+** z **\*** z), 2.f **\*** (x **\*** y **-** r **\*** z), 2.f **\*** (x **\*** z **+** r **\*** y),

2.f **\*** (x **\*** y **+** r **\*** z), 1.f **-** 2.f **\*** (x **\*** x **+** z **\*** z), 2.f **\*** (y **\*** z **-** r **\*** x),

2.f **\*** (x **\*** z **-** r **\*** y), 2.f **\*** (y **\*** z **+** r **\*** x), 1.f **-** 2.f **\*** (x **\*** x **+** y **\*** y)

);

glm**::**mat3 M **=** S **\*** R;

*// Compute 3D world covariance matrix Sigma*

glm**::**mat3 Sigma **=** glm**::**transpose(M) **\*** M;

*// Covariance is symmetric, only store upper right*

cov3D[0] **=** Sigma[0][0];

cov3D[1] **=** Sigma[0][1];

cov3D[2] **=** Sigma[0][2];

cov3D[3] **=** Sigma[1][1];

cov3D[4] **=** Sigma[1][2];

cov3D[5] **=** Sigma[2][2];

}

\_\_device\_\_ glm::vec3 computeColorFromSH(

int idx, // 该线程负责第几个Gaussian

int deg, // 球谐的度数

int max\_coeffs, // 一个Gaussian最多有几个傅里叶系数

const glm::vec3\* means, // Gaussian中心位置

glm::vec3 campos, // 相机位置

const float\* shs, // 球谐系数

bool\* clamped) // 表示每个值是否被截断了（RGB只能为正数），这个在反向传播的时候用

{

// The implementation is loosely based on code for

// "Differentiable Point-Based Radiance Fields for

// Efficient View Synthesis" by Zhang et al. (2022)

glm::vec3 pos = means[idx];

glm::vec3 dir = pos - campos;

dir = dir / glm::length(dir); // dir = direction，即观察方向

...

// RGB colors are clamped to positive values. If values are

// clamped, we need to keep track of this for the backward pass.

clamped[3 \* idx + 0] = (result.x < 0);

clamped[3 \* idx + 1] = (result.y < 0);

clamped[3 \* idx + 2] = (result.z < 0);

return glm::max(result, 0.0f);

}

[【计算机视觉】Gaussian Splatting源码解读补充（二）\_colmap free gaussians splatting github-CSDN博客](https://blog.csdn.net/qaqwqaqwq/article/details/136886008)

*// Main rasterization method. Collaboratively works on one tile per*

*// block, each thread treats one pixel. Alternates between fetching*

*// and rasterizing data.*

*// "Alternates between fetching and rasterizing data":*

*// 线程在读取数据（把数据从公用显存拉到block自己的显存）和进行计算之间来回切换，*

*// 使得线程们可以共同读取Gaussian数据。*

*// 这样做的原因是block共享内存比公共显存快得多。*

template <uint32\_t CHANNELS> *// CHANNELS取3，即RGB三个通道*

\_\_global\_\_ void \_\_launch\_bounds\_\_(BLOCK\_X \* BLOCK\_Y)

renderCUDA(

const uint2\* \_\_restrict\_\_ ranges, *// 每个tile对应排过序的数组中的哪一部分*

const uint32\_t\* \_\_restrict\_\_ point\_list, *// 按tile、深度排序后的Gaussian ID列表*

int W, int H, *// 图像宽高*

const float2\* \_\_restrict\_\_ points\_xy\_image, *// 图像上每个Gaussian中心的2D坐标*

const float\* \_\_restrict\_\_ features, *// RGB颜色*

const float4\* \_\_restrict\_\_ conic\_opacity, *// 解释过了*

float\* \_\_restrict\_\_ final\_T, *// 最终的透光率*

uint32\_t\* \_\_restrict\_\_ n\_contrib,

*// 多少个Gaussian对该像素的颜色有贡献（用于反向传播时判断各个Gaussian有没有梯度）*

const float\* \_\_restrict\_\_ bg\_color, *// 背景颜色*

float\* \_\_restrict\_\_ out\_color) *// 渲染结果（图片）*

{

*// Identify current tile and associated min/max pixel range.*

auto block = cg::this\_thread\_block();

uint32\_t horizontal\_blocks = (W + BLOCK\_X - 1) / BLOCK\_X; *// x方向上tile的个数*

uint2 pix\_min = { block.group\_index().x \* BLOCK\_X, block.group\_index().y \* BLOCK\_Y };

*// 我负责的tile的坐标较小的那个角的坐标*

uint2 pix\_max = { min(pix\_min.x + BLOCK\_X, W), min(pix\_min.y + BLOCK\_Y , H) };

*// 我负责的tile的坐标较大的那个角的坐标*

uint2 pix = { pix\_min.x + block.thread\_index().x, pix\_min.y + block.thread\_index().y };

*// 我负责哪个像素*

uint32\_t pix\_id = W \* pix.y + pix.x;

*// 我负责的像素在整张图片中排行老几*

float2 pixf = { (float)pix.x, (float)pix.y }; *// pix的浮点数版本*

*// Check if this thread is associated with a valid pixel or outside.*

bool inside = pix.x < W && pix.y < H; *// 看看我负责的**像素有没有跑到图像外面去*

*// Done threads can help with fetching, but don't rasterize*

bool done = !inside;

*// Load start/end range of IDs to process in bit sorted list.*

uint2 range = ranges[block.group\_index().y \* horizontal\_blocks + block.group\_index().x];

const int rounds = ((range.y - range.x + BLOCK\_SIZE - 1) / BLOCK\_SIZE);

*// BLOCK\_SIZE = 16 \* 16 = 256*

*// 我要把任务分成rounds批，每批处理BLOCK\_SIZE个Gaussians*

*// 每一批，每个线程负责读取一个Gaussian的信息，*

*// 所以该block的256个线程每一批就可以读取256个Gaussian的信息*

int toDo = range.y - range.x; *// 我要处理的Gaussian个数*

*// Allocate storage for batches of collectively fetched data.*

*// \_\_shared\_\_: 同一block中的线程共享的内存*

\_\_shared\_\_ int collected\_id[BLOCK\_SIZE];

\_\_shared\_\_ float2 collected\_xy[BLOCK\_SIZE];

\_\_shared\_\_ float4 collected\_conic\_opacity[BLOCK\_SIZE];

*// Initialize helper variables*

float T = 1.0f; *// T = transmittance，透光率*

uint32\_t contributor = 0; *// 多少个Gaussian对该像素的颜色有贡献*

uint32\_t last\_contributor = 0; *// 比contributor慢半拍的变量*

float C[CHANNELS] = { 0 }; *// 渲染结果*

*// Iterate over batches until all done or range is complete*

for (int i = 0; i < rounds; i++, toDo -= BLOCK\_SIZE)

{

*// End if entire block votes that it is done rasterizing*

int num\_done = \_\_syncthreads\_count(done);

*// 它首先具有\_\_syncthreads的功能（让所有线程回到同一个起跑线上），*

*// 并且返回对于多少个线程来说done是true。*

if (num\_done == BLOCK\_SIZE)

break; *// 全干完了，收工*

*// Collectively fetch per-Gaussian data from global to shared*

*// 由于当前block的线程要处理同一个tile，所以它们面对的Gaussians也是相同的*

*// 因此合作读取BLOCK\_SIZE条的数据。*

*// 之所以分批而不是一次读完可能是因为block的共享内存空间有限。*

int progress = i \* BLOCK\_SIZE + block.thread\_rank();

if (range.x + progress < range.y) *// 我还有没有活干*

{

*// 读取我负责的Gaussian信息*

int coll\_id = point\_list[range.x + progress];

collected\_id[block.thread\_rank()] = coll\_id;

collected\_xy[block.thread\_rank()] = points\_xy\_image[coll\_id];

collected\_conic\_opacity[block.thread\_rank()] = conic\_opacity[coll\_id];

}

block.sync(); *// 这下collected\_\*\*\* 数组就被填满了*

*// Iterate over current batch*

for (int j = 0; !done && j < min(BLOCK\_SIZE, toDo); j++)

{

*// Keep track of current position in range*

contributor++;

*// 下面计算当前Gaussian的不透明度*

*// Resample using conic matrix (cf. "Surface*

*// Splatting" by Zwicker et al., 2001)*

float2 xy = collected\_xy[j]; *// Gaussian中心*

float2 d = { xy.x - pixf.x, xy.y - pixf.y };

*// 该像素到Gaussian中心的位移向量*

float4 con\_o = collected\_conic\_opacity[j];

float power = -0.5f \* (con\_o.x \* d.x \* d.x + con\_o.z \* d.y \* d.y) - con\_o.y \* d.x \* d.y;

*// 二维高斯分布公式的指数部分（见补充说明）*

if (power > 0.0f)

continue;

*// Eq. (2) from 3D Gaussian splatting paper.*

*// Obtain alpha by multiplying with Gaussian opacity*

*// and its exponential falloff from mean.*

*// Avoid numerical instabilities (see paper appendix).*

float alpha = min(0.99f, con\_o.w \* exp(power));

*// Gaussian对于这个像素点来说的不透明度*

*// 注意con\_o.w是”opacity“，是Gaussian整体的不透明度*

if (alpha < 1.0f / 255.0f) *// 不透明度太小，不渲染*

continue;

float test\_T = T \* (1 - alpha); *// 透光率是累乘*

if (test\_T < 0.0001f)

{ *// 当透光率很低的时候，就不继续渲染了（反正渲染了也看不见）*

done = true;

continue;

}

*// Eq. (3) from 3D Gaussian splatting paper.*

for (int ch = 0; ch < CHANNELS; ch++)

C[ch] += features[collected\_id[j] \* CHANNELS + ch] \* alpha \* T;

*// 计算当前Gaussian对像素颜色的贡献*

T = test\_T; *// 更新透光率*

*// Keep track of last range entry to update this*

*// pixel.*

last\_contributor = contributor;

}

}

*// All threads that treat valid pixel write out their final*

*// rendering data to the frame and auxiliary buffers.*

if (inside)

{

final\_T[pix\_id] = T;

n\_contrib[pix\_id] = last\_contributor;

for (int ch = 0; ch < CHANNELS; ch++)

out\_color[ch \* H \* W + pix\_id] = C[ch] + T \* bg\_color[ch];

*// 把渲染出来的像素值写进out\_color*

}

}