**Skia vulkan和OpenGL不同渲染后端调用性能比较分析**

**SkiaVK, SkiaGL背景介绍**

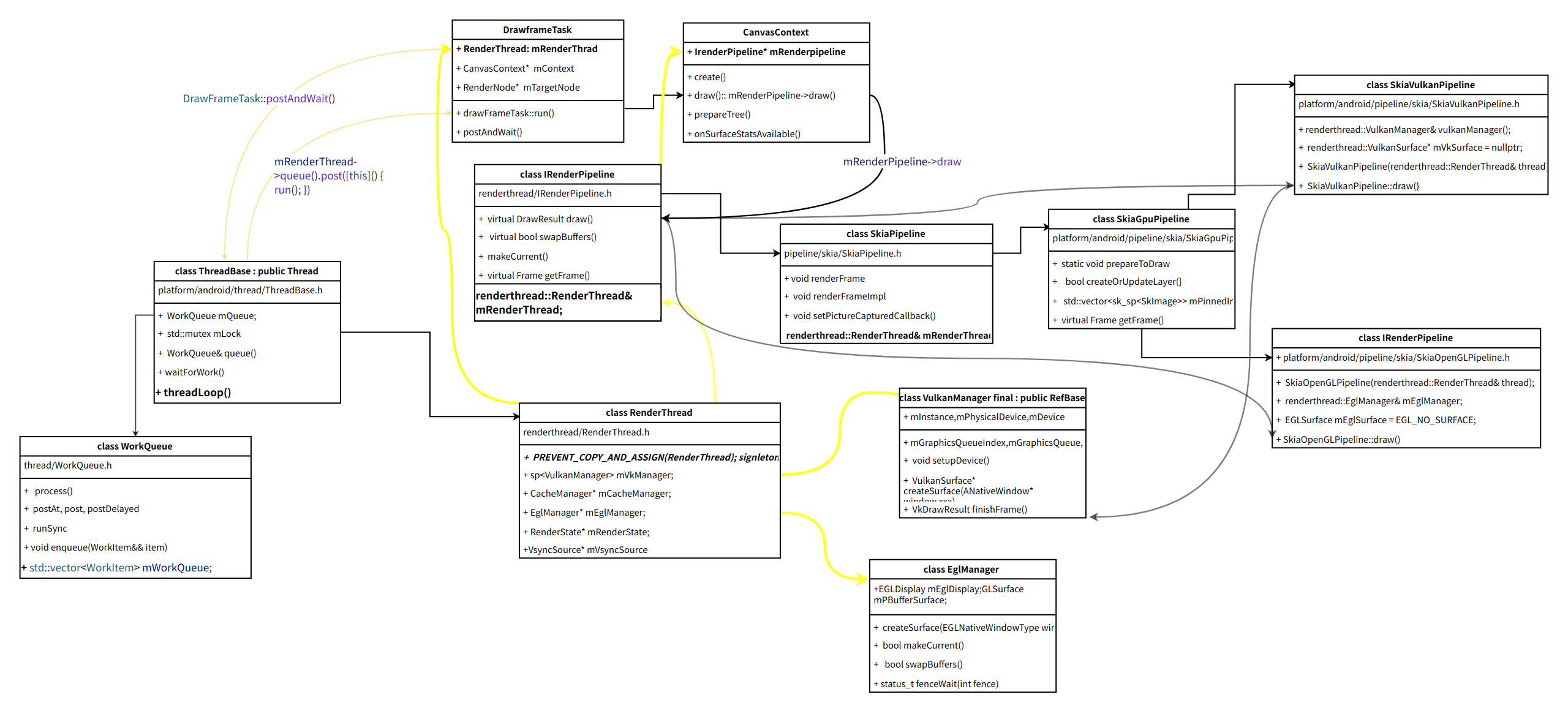
从Android 8.0开始就支持对硬件加速渲染设置不同的渲染管道，目前有三种opengl、Skiagl、Skiavk，Android 8.0系统默认使用是opengl渲染管道S，Android 9.0系统默认使用Skiagl渲染管道；从Android 10.0开始不支持opengl渲染管道，只支持Skiagl、Skiavk，根据系统属性ro.hwui.use\_vulkan值决定默认使用Skiavk或是Skiagl, 通过指令 adb shell "getprop debug.renderengine.vulkan" 可以看到小米手机是否支持SkiaVK.

**Skiavk和SkiaGL调用流程对比分析**

Skia是Google一个底层的图形、图像、动画、SVG、文本等多方面的图形库，是Android中图形系统的引擎。Skia作为第三方软件放在目录：external/skia/。Skia是HWUI 的核心图形库，OpenGL（openGLES), Vulkan是HWUI与GPU之间的桥梁，负责将Skia生成的绘制命令转化为GPU能够执行的指令序列。vulkan相比于OpenGL是较新的API，目的是为了给使用者更多的GPU控制权限和更低的CPU开销，为了验证从SkiaGL切换到SkiaVK是否引起性能退化，因此着重分析两种渲染流程的性能。

RenderThread渲染线程，是一个单例，也就是说，一个进程中只有一个，所有的绘制操纵都必须在这个线程中完成。应用端很多绘制操作，都以RenderTask的形式post到RenderThread线程中完成。

CanvasContext，顾名思义是画布上下文，表现在屏幕上就是一块区域，可以在上面使用各种API绘制想要的东西。

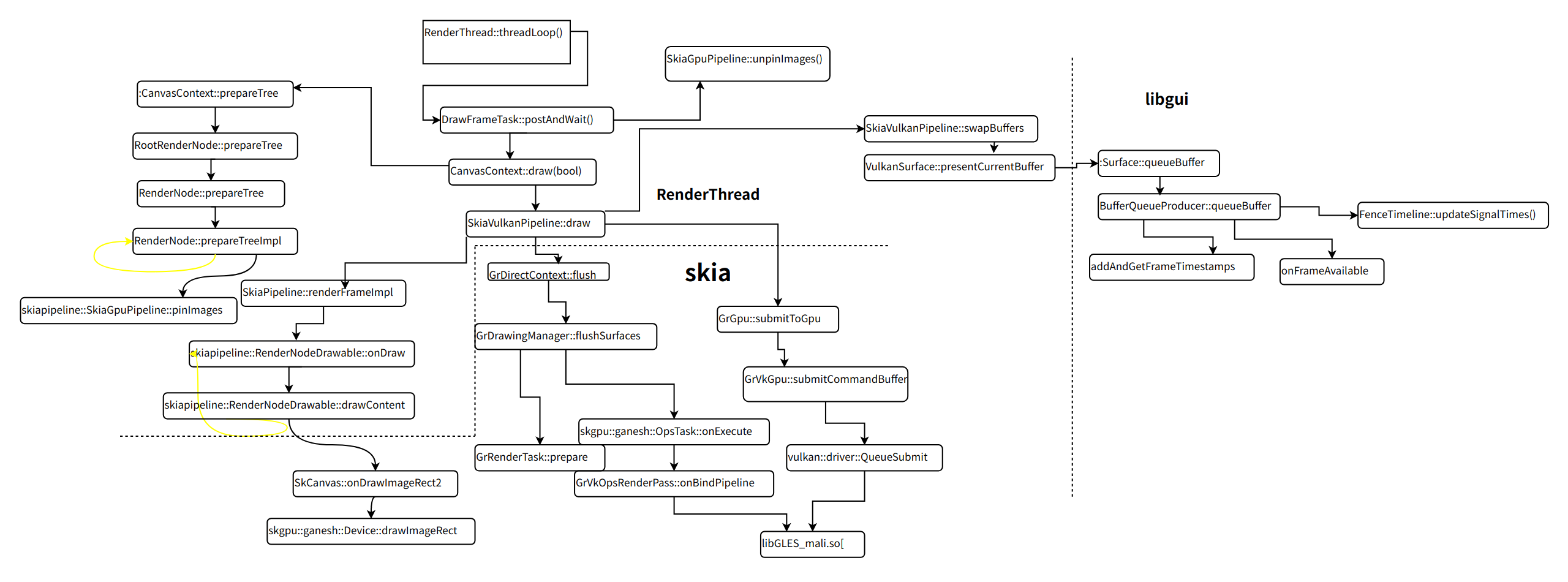


上图表示的是libhwui 调用drawFrames draw函数的流程，以及 renderThread, Canvas, SkiaPipeline, SkiaGPUPipeline, SkiaVulkanPipeline 以及manager之间的UMD图关系，最终绘制命令下发到Skia 渲染引擎，并通过渲染API vulkan or OpenGLES组装渲染命令在GPU绘制目标图层，由于google Skia引擎调用 通过libhwui.so，因此着重分析libhwui.so 以及libGLES\_mali.so的调用次数来分析性能指标。

**SkiaVK情况下Frame drawcall 命令调用简析**

如下图：主要表示RenderThread， Skia, Vulkan, libgui， libGLES\_mali之间 在drawFrame函数中的调用流程与对应关系。 Renderthread workqueue中接收到新任务，触发后将会准备绘制命令并且经由Skia组装下发到GPU执行，prepareTree则是准备渲染节点信息的刷新，其调用prepareTreeImpl函数进行准备工作，遍历渲染节点，并根据damage叠加器更新屏幕需要重新绘制的信息，准备好frame所需的绘制信息内容后，通过SkiaPipeline drawContent 调用Skia对应的接口，并更新绘制内容的指令。做完准备工作后， SkiaVulkanPipeline下发绘制命令，Skia引擎会组装转换命令为vulkan格式， 并等到submit 指令触发后将组装好的指令下发到GPU执行。

（执行vulkan::driver::QueueSubmit后调用libGELS\_mali.so，有待分析）



**SkiaGL情况下frame drawcall 命令调用简析**

如下图：（暂时未完成）

由于本wiki对Skia引擎的具体实现不做具体分析，如果对Skia渲染引擎有更多兴趣，可以访问官方网站https://Skia.org。

**测试条件**

使用工具PerfTool组件，分别使用perfetto 和simpleperf 两个工具测试一小段时间（6s), 锁定cpu1, pum\_power, cpum\_perf, cpub ddr, gpu, L3 cache各个IP的频率，其中除了GPU设置为中等偏上672000000（672M左右）外，其余均设各个IP的物理最小频率。测试过程以及测试结束均要检验并核实频率未变,抓取输出文件，包括但不限于trace file, perf.data， perf.html, perf.txt等。

使用工具Arm streamline, 需保证在相同配置文件， 主要是configureation.xml, session.xml文件, 保证手机无其他App 后台运行， 打开抖音并一个视频循环播放，配置采样时长16s, 视频循环播放不少于6s,抓取数据并分析。

其中perfetto, simpleperf 测试时锁频的主要IP频率信息如下：

cpul\_curfreq: 334230

cpum\_power\_curfreq: 417790

cpum\_perf\_curfreq: 691200

cpub\_curfreq: 1891200

ddr\_curfreq: 836000000

gpu\_curfreq: 672000000

l3\_curfreq: 238738000

测试前后保证频率未发生任何改变。

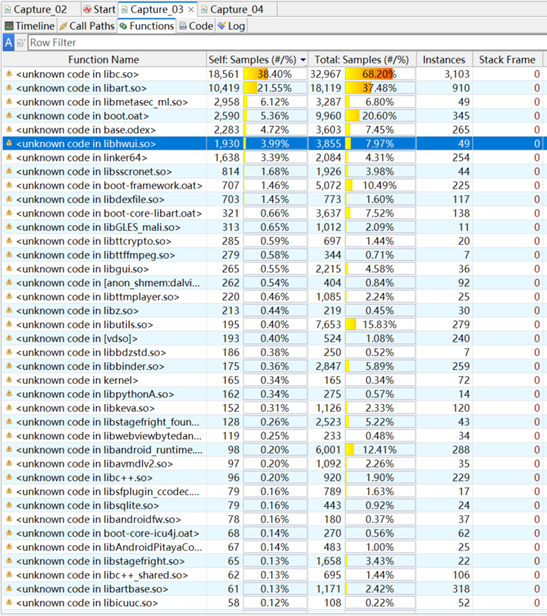
stremline时arm 官方提供的测试工具，其能够对app启动到运行进行全过程调用的分析。Streamline测试需保证在相同配置文件， 主要是configureation.xml, session.xml文件，分别测试Skiavk, SkiaGL 环境下抖音同一个视频循环播放，由于streamline具有启动app的权限，操作的时候需要首先退出所有运行的app，包括抖音本身，配置文件中设置duration 为16s, 选取视频播放处的1s 作为统计时间段。

**实验数据分析对比**

**Streamline数据分析对比**

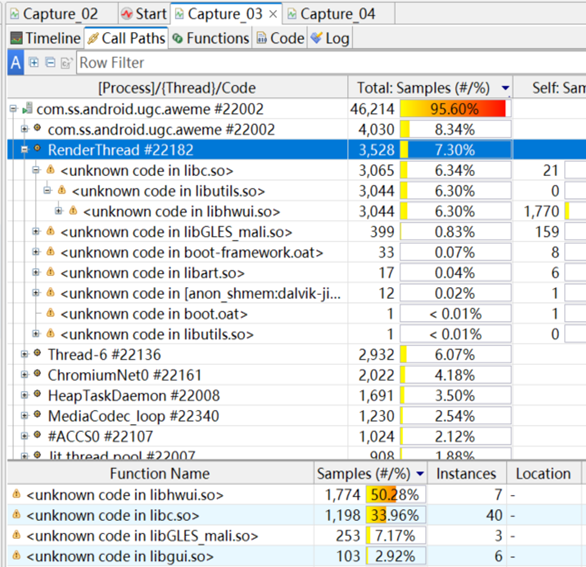
**SkiaVK统计数据**

SkiaVK 模式下各个库的调用统计，以及renderthread线程内部的具体调用如下图



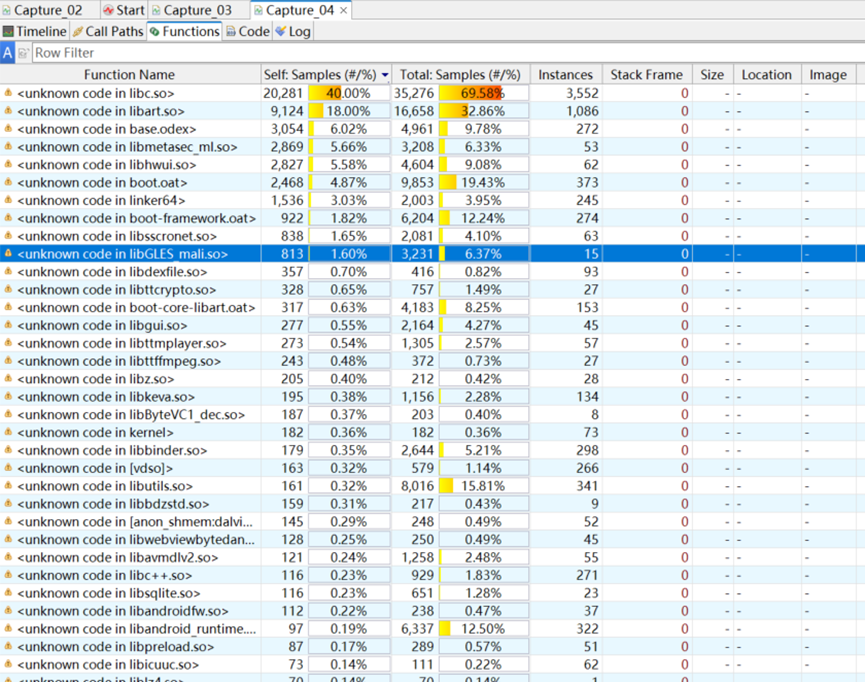
libGLES\_mali.so占比2.09%， libhwui.so占比7.79%。

下图时renderThread中 主要库的调用占比，分别列举了libhwui.so, libGLES\_mali.so, libc.so, libgui.so,



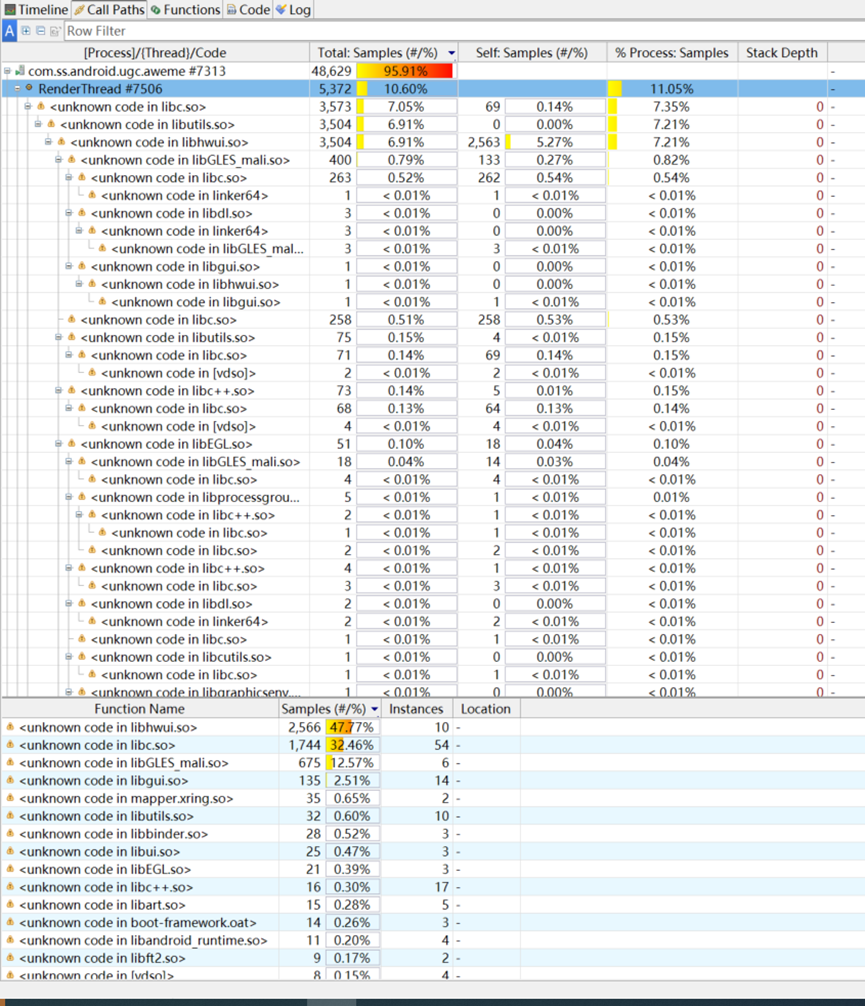
**SkiaGL统计数据**

下图时function的整体调用占比:



其中libGLES\_mali.so占总体比例6.37%， libhwui.so占总体9.78%，高于SkiaVK下的2.09%，7.79%。

下图是RenderThread中各个库的调用占比，如下图所示：



着重分析libhwui.so, libGLES\_mali.so, 分别占比47.77%，12.57%, 总共占比60.3%， 高于SkiaVK情况下的57.4%， 因此SkiaVK优于SkiaGL。

为了兼顾整个测试阶段的数据的完整性和代表性，截取了RenderThead线程起始到实验截止之间15s时间的调用情况数据并作对比，分别记录分析抖音进程和RenderThread线程下libhwui, libGLES\_mali 库调用数量，占比情况对比列出如下表格：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| sampless(#/%) | 抖音 | libhwui | libGLES\_mali | RenderThread | libhwui | libGLES\_mali |
| SkiaGL | 48629（95.91%） | 4604（9.08%） | 3231（6.37%） | 5372（10.60%） | 2566（47.77%） | 675（12.75%） |
| SkiaVK | 46214（95.60%） | 3855（7.97%） | 1012（2.09%） | 3528 （7.30%） | 1774（50.28%） | 253 （7.17%） |

从数量上分析，libhwui, libGLES\_mali, RenderThread 的调用采样数量在SkiaVK情况下都少于SkiaGL, 且libGLES\_mali库的调用分线减少； RenderThread线程内部分析，虽然libhwui在RenderThread中占比增加， 但是SkiaVK 情况下libhwui,libGLES\_mali的数量都有明显减少， 尤其是libGLES\_mali的数量减少更多，且占比更低。因此我们可以认为SkiaVk相比与SkiaGL管线性能并没有退化，甚至还有显著提升。

**Simpleperf 数据分析对比**

SkiVK simplerperf测试数据如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **线程名/函数** | **CPUcycles** | **总线程占比（weme抖音）** | **子线程占比（Render Thread）** |
| **Process: 2093** **(com.ss.android.ugc.aweme)** | 4078615271 | NA | NA |
| **Thread: 5540**  **(RenderThread)** | 359444494 | 8.8129% | 100.0000% |
| **Library:**  **/vendor/lib64/egl**  **/libGLES\_mali.so** | 17605255 | NA | 4.8979% |
| **Library: /system/**  **lib64/libhwui.so** | 209967520 | NA | 58.4144% |

SKiGL simpleperf 测试数据如下：

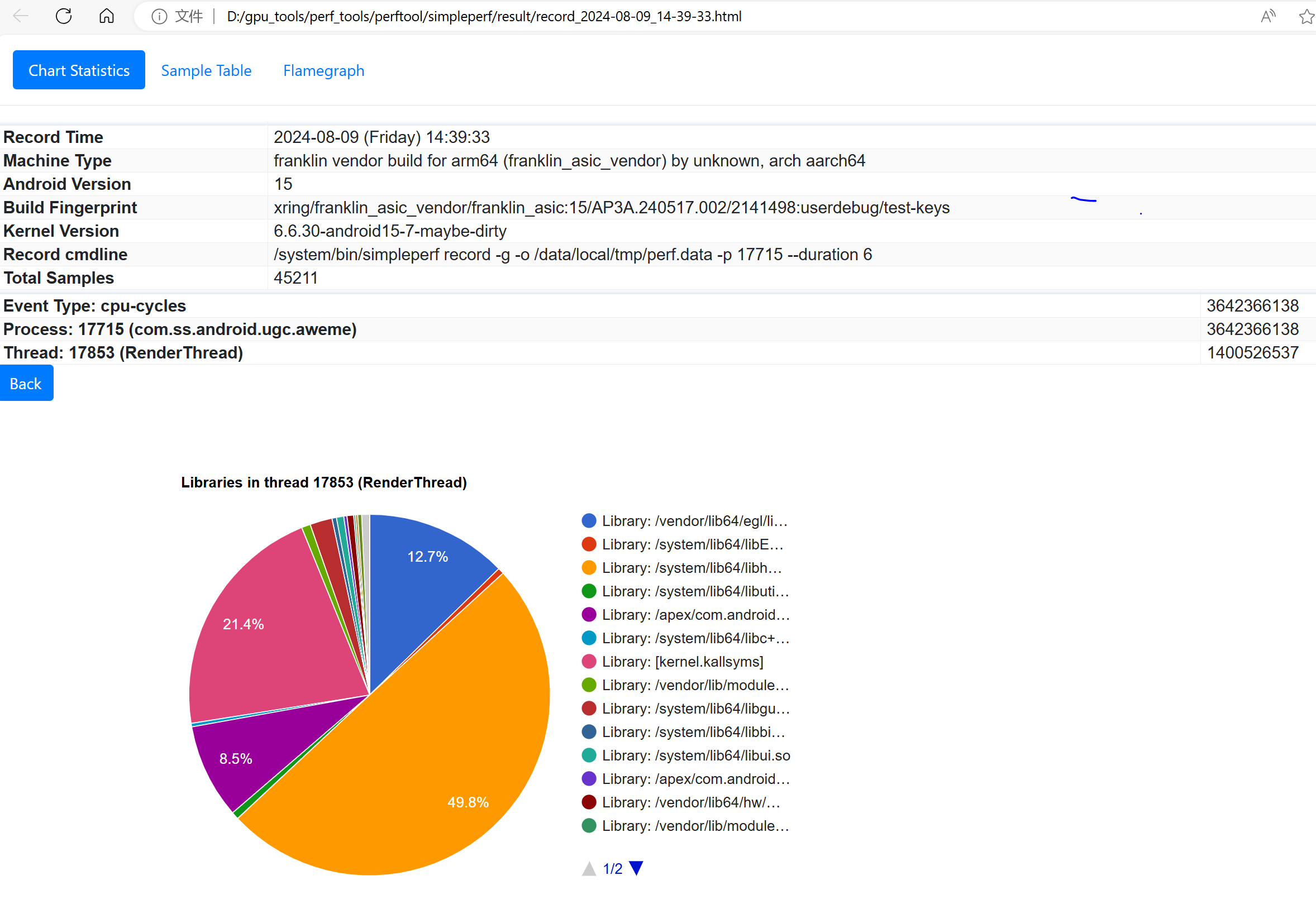
|  |  |  |  |
| --- | --- | --- | --- |
| **线程名/函数** | **CPUcycles** | **总线程占比**  **（weme抖音）** | **子线程占比（Render Thread）** |
| **Process: 2093**  **(com.ss.android.ugc.aweme)** | 3642366138 | NA | NA |
| **Thread: 5540 (RenderThread)** | 1400526537 | 38.4510% | 100.0000% |
| **Library: /vendor/lib64/egl/libGLES\_mali.so** | 177537295 | NA | 12.6765% |
| **Library: /system/lib64/libhwui.so** | 697542467 | NA | 49.8057% |

针对simpleperf的数据分析结果如下RenderThread cycle数占抖音进程的比38.4510%， 远高于SkiaVK中的8.8129%，怀疑是测试过程中的偏离值，分析了strealine中的这个数是10.6%.

着重分析RenderThread调用中libhwui.so,libGLES\_mali.so，对比发现skiGL libGLES\_mali.so的调用比例高于SkiaVK, 但是libhwui.so调用比例低，但是在整个抖音进程中 renderThread中两个库的调用cycle占比都要高于SkiaVK情况。

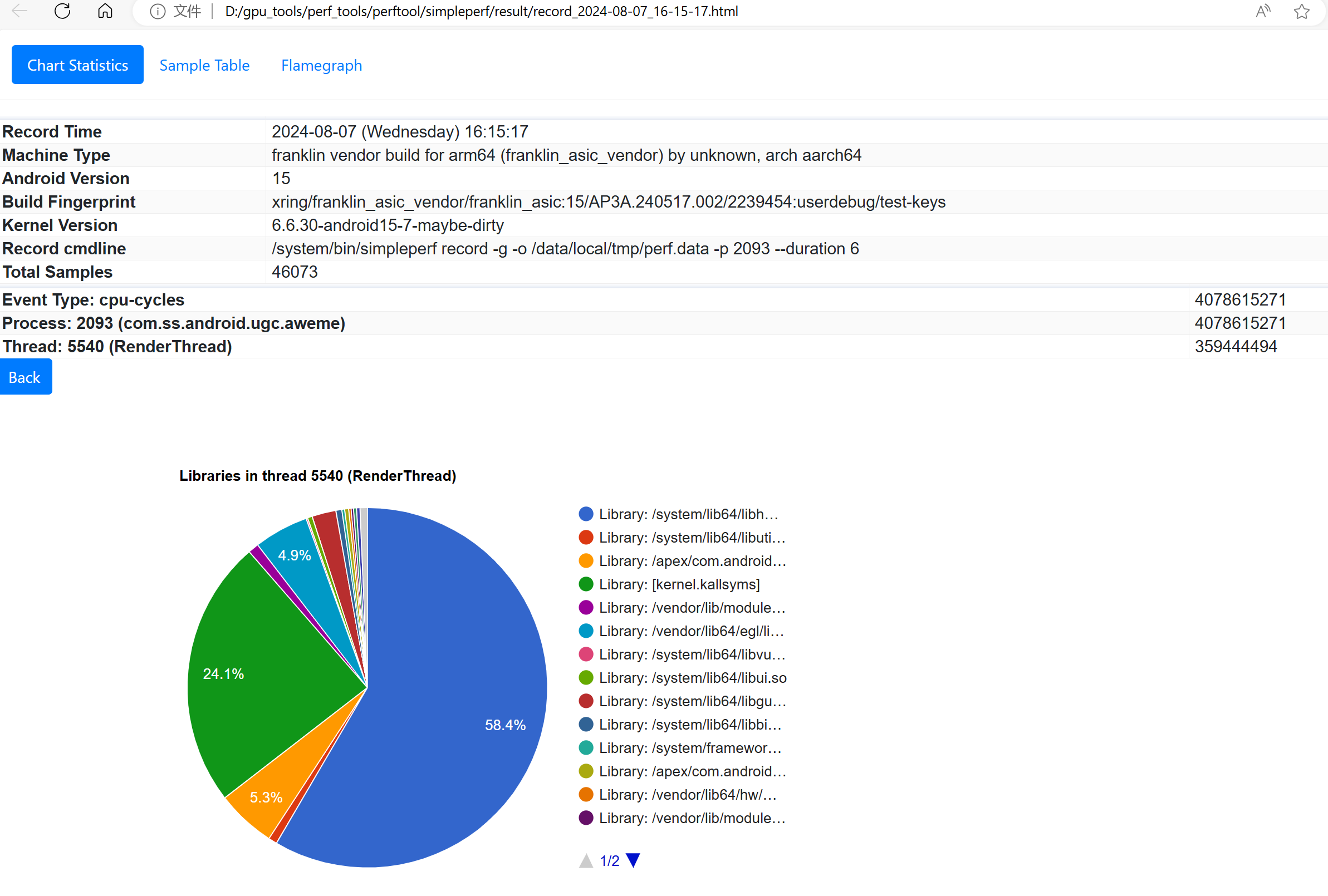
**饼图展示：**

**SkiaGL renderThread 各个函数调用饼图统计图**



其中橘黄色表示libhwui.so, 蓝色libGLES\_mali.so.

**SkiaVK renderThread各个函数调用饼图统计图**

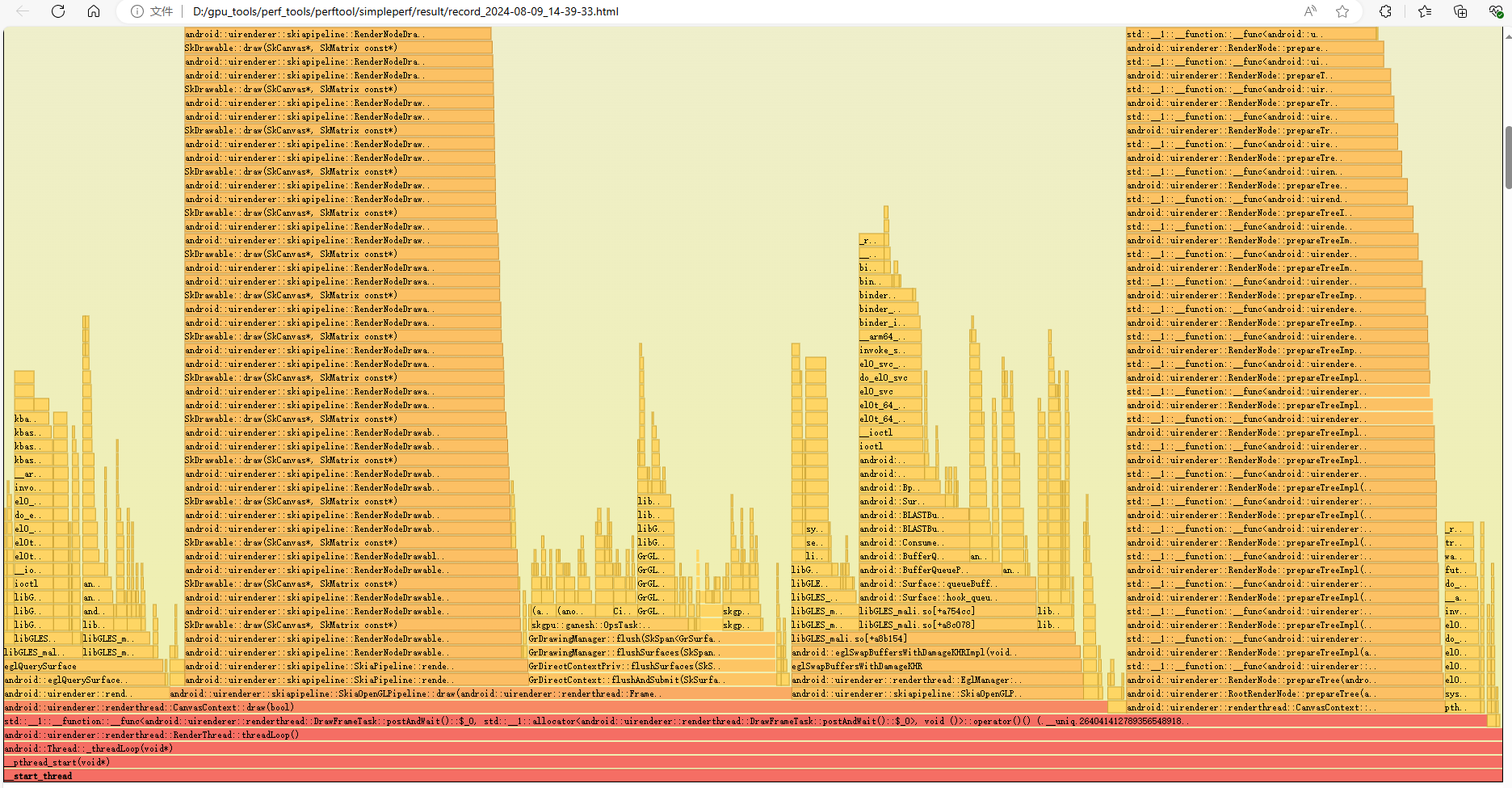


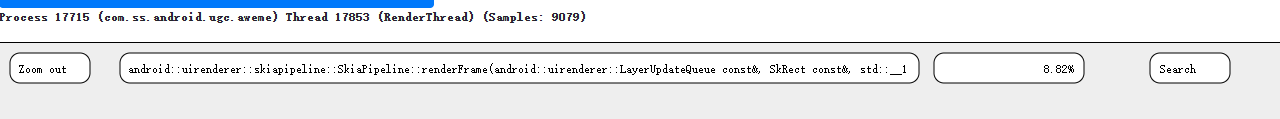
其中蓝色代表libhwui.so(58.04%), 浅蓝色部分4.9%表示libGLES\_mali.so。

**火焰图**

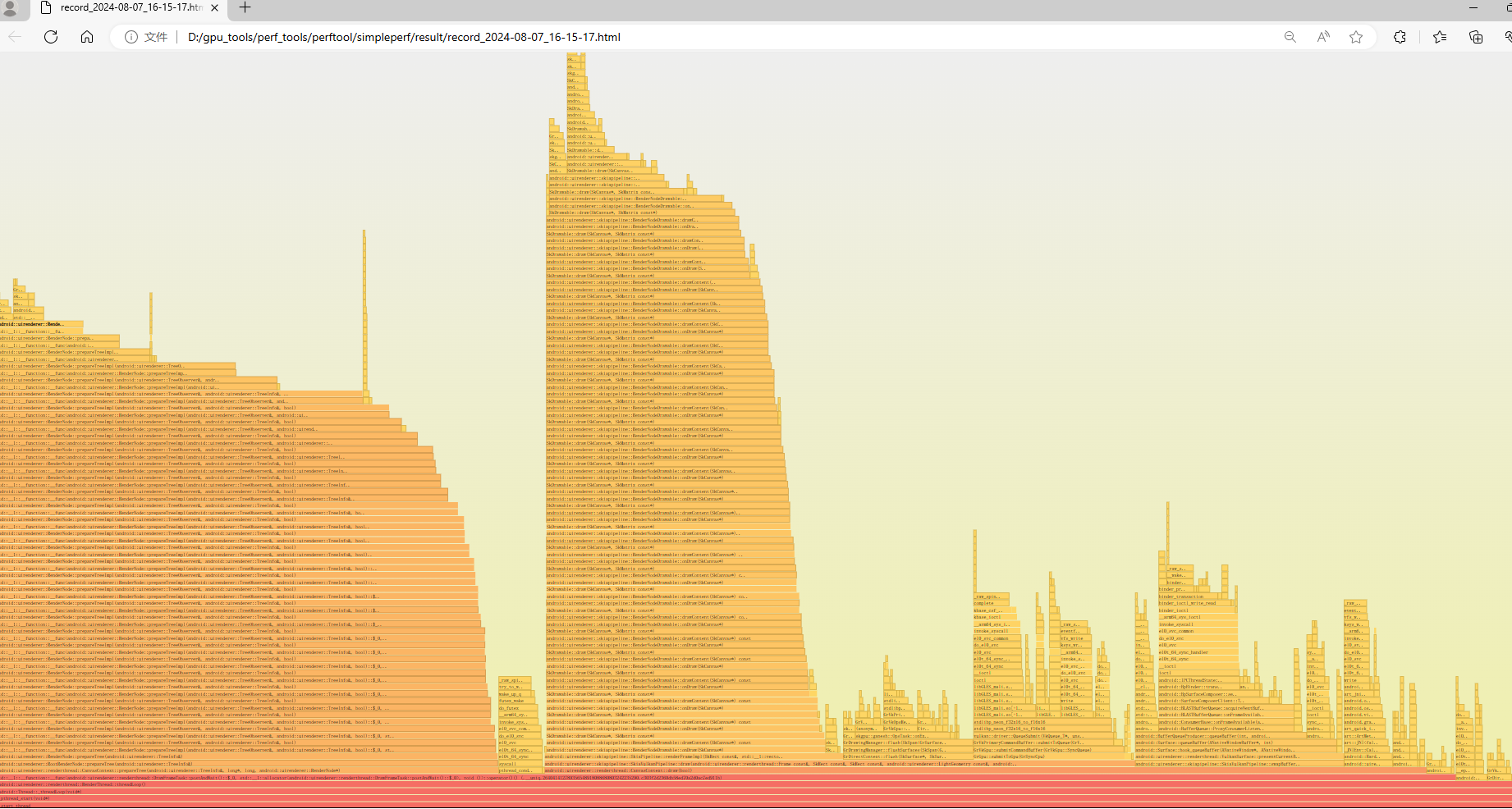
火焰图不仅能够展示各个函数调用占比，还能看到调用之间的关系。下面针对DrawFrameTask::postAndWait函数的统计结果作为展示：

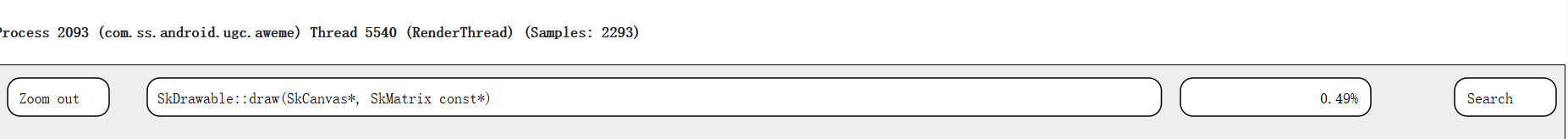
**SkiaGL如下图所示**





**SkiaVK postAndWait 火焰图**





以上就是SkiaGL, SkiaVK renderThread线程中postAndWait的调用火焰图，囊括renderthread调用Canvas, Skiapipeline，SkiaVulkanPipeline, SkiaOpenGLpipeline，经由Skia 渲染引擎调用，到KMD的调用等，由于图像太大熬制一些细节不清晰，大家可在自己的浏览器中打开分析。此函数的调用比例分别是vk:0.49%, GL 8.82%. 明显GL的调用占比高于VK.

**结论**

SkiaVK流程下RenderThread, libhwui, libLGES\_mali的调用cycle数量上少于SkiaGL流程下的统计值， libhwui, libGLES\_mali 在整个抖音线程中cycle数的占比对比可以看出，其次在RenderThread线程内部看，虽然SkiaVK流程下libhwui的占比高于SkiaGL情况，但是libGELS\_mali的占比低于SkiaGL情况，且两个库的总站比 SkiaVK是要写于SkiaGL的。

**perffetto数据对比分析**

pefetto工具在进程启动后通过进程号抓取trace文件， 命令格式如下:

adb shell "cat /data/local/tmp/perfetto.conf | perfetto --txt -c - -o /data/misc/perfetto-traces/trace\_2024-08-07\_15-36-53"

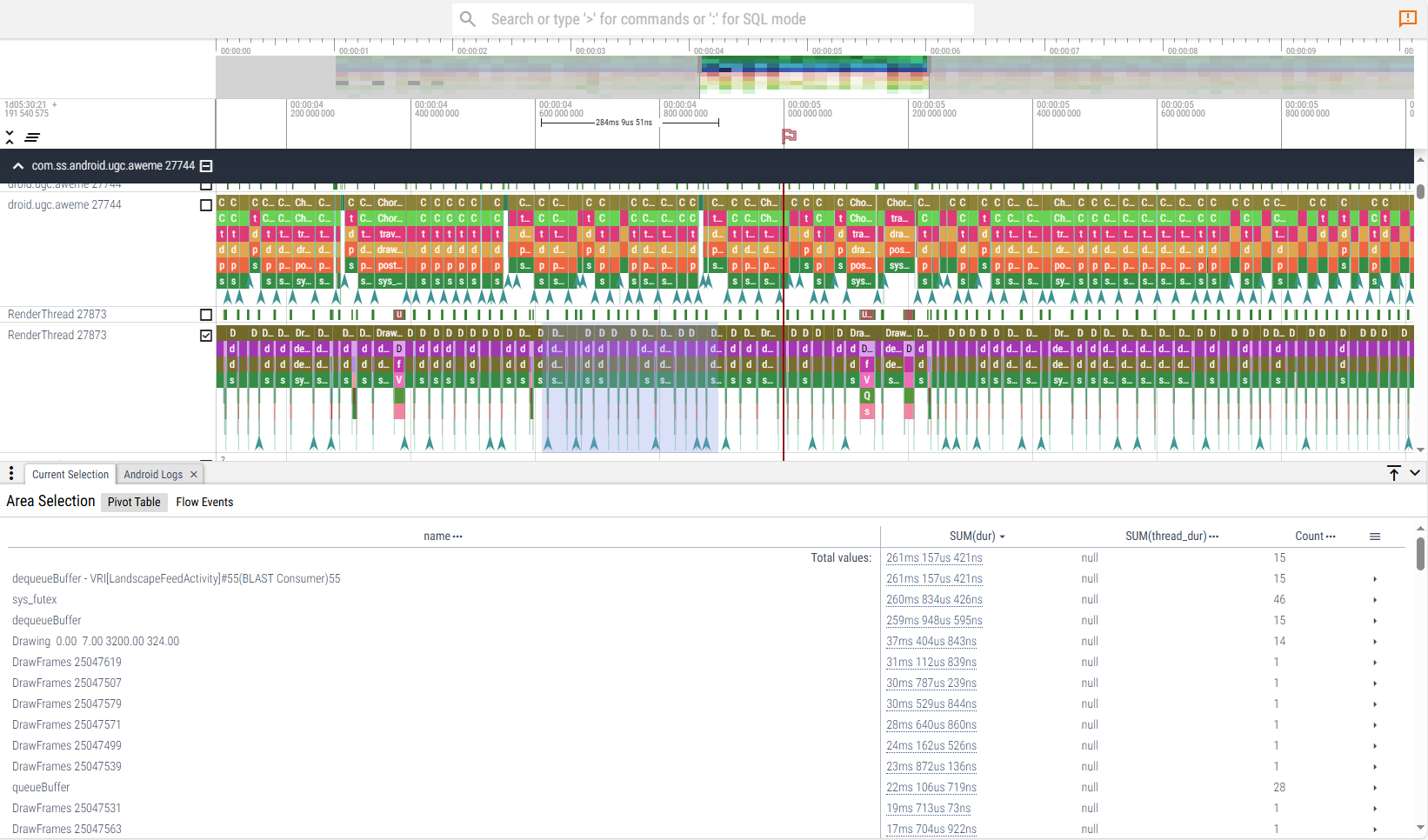
trace文件可以通过网页https://ui.perfetto.dev/ 拖入并在线分析，测试前需要设置锁频，跟上simpleperf一样。

分析时侧重单个drawcall, 也会统计一段时间内的平均值。

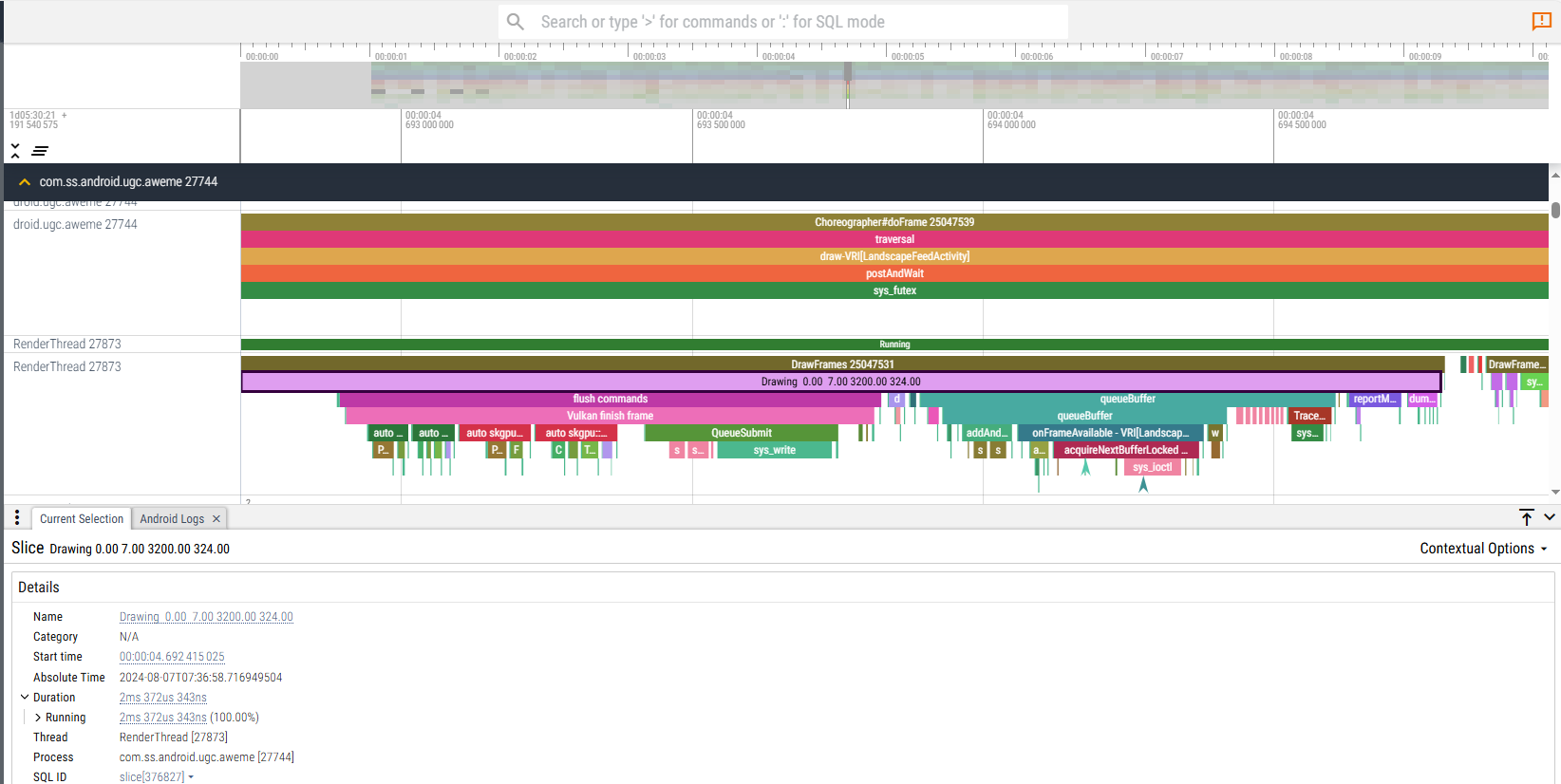
SkiaVK 或者 SkiaGL下一个drawcall的调用，主要分析RenderThread线程，DrawFrames函数执行绘制命令，具体调用关系请看上面对应的Frame drawcall 命令调用简析。下面分别摘取renderThread一段时间（280ms左右），以及其中一个drawcall命令的截图进行分析。

**SkiaVK peffetto数据截取如下：**

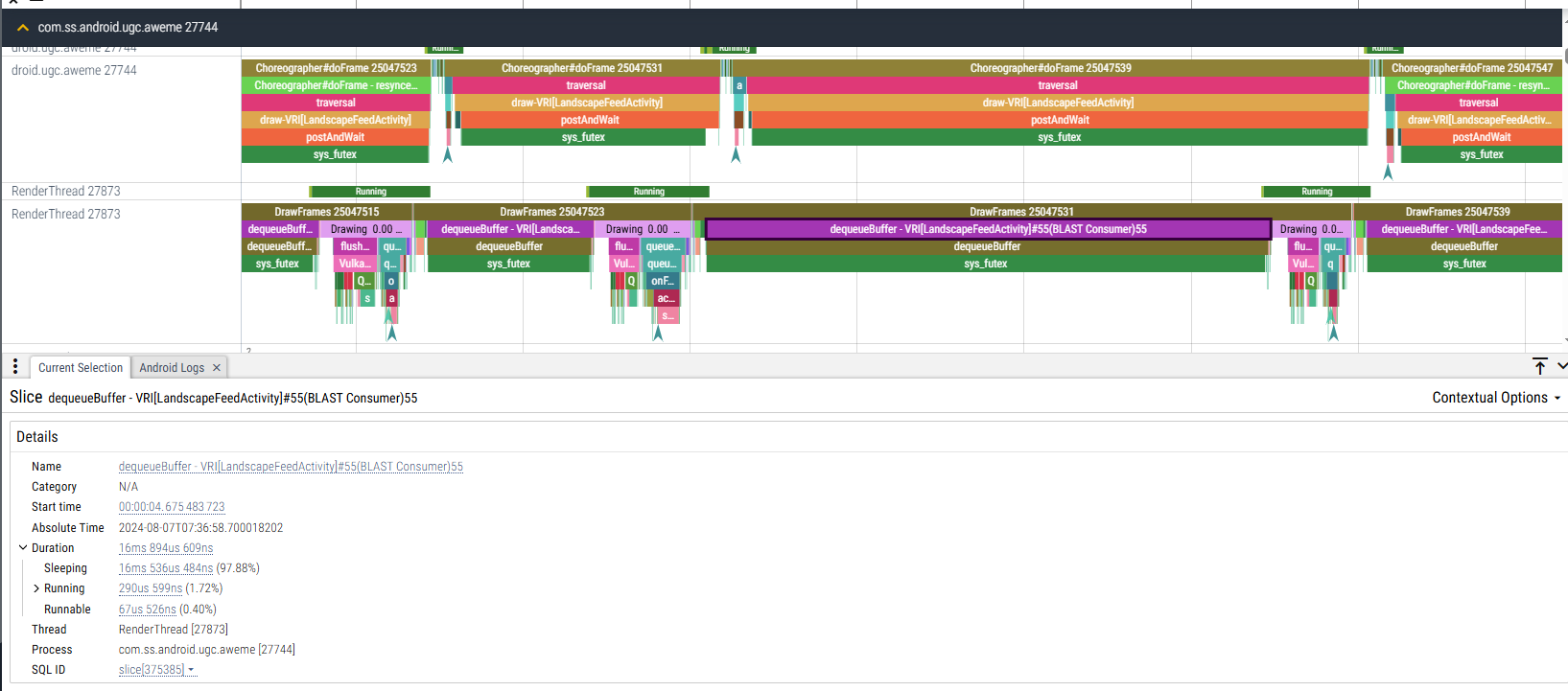
1. 280ms 左右的统计图



1. Drawcall统计图



1. Dequeue buffer 统计图

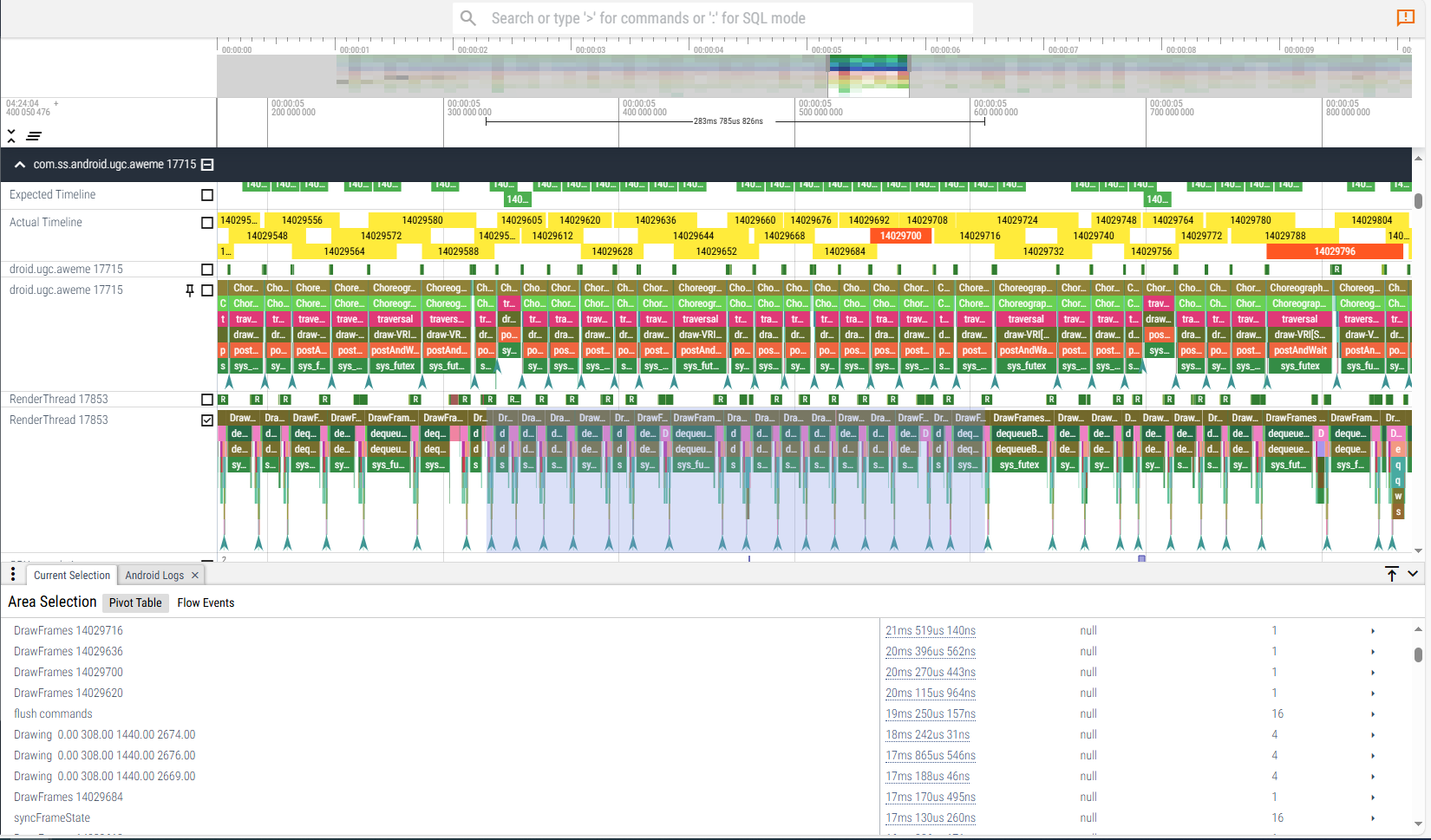


分析： Drawing 0.00 xxx部分则是drawcall命令执行部分，14次调用用时37ms404us843ns， 平均2.7ms左。主要有flush commands 和queue buffer两部分,这两部分running并没有阻塞，达到100%。且能明显看到调用了vulkan finish frame，queueBuffer从侧面证明走入了SkiaVK渲染流程。

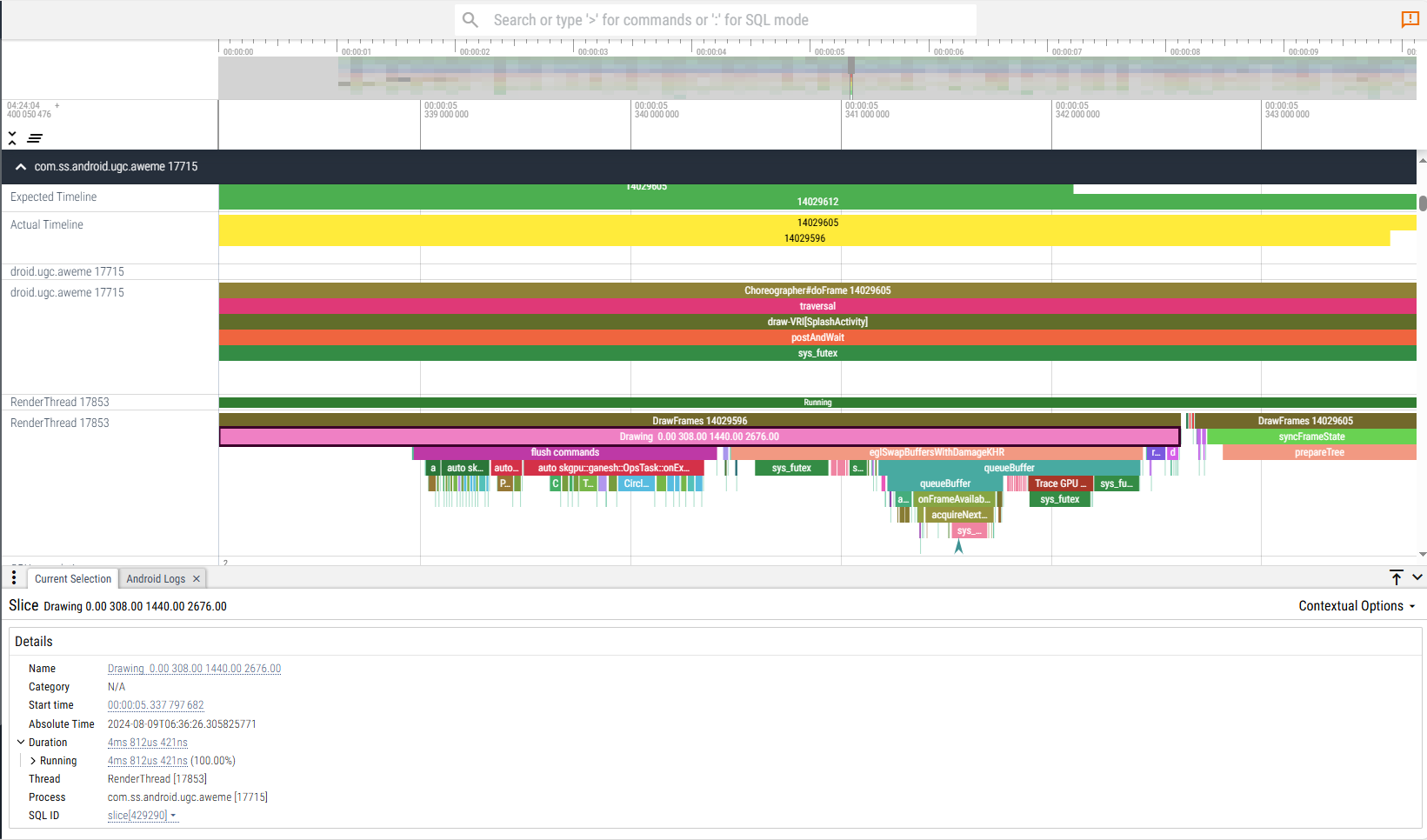
DrawFrames另外一部分是dequeueBuffer调用，这部分的用时较长，与正常情况有出入，已经分析正常版本情况并跟测试同事对齐，是与DPU交互过程中sync同步有问题，导致渲染进程拿到surface buffer用时比较长， 3. Dequeue buffer 统计图显示所示用时16ms 894us 609ns,其中线程处于sleeping时间16ms536us484ns, 显然是有问题的。

**SkiaGL peffetto 数据截取如下：**

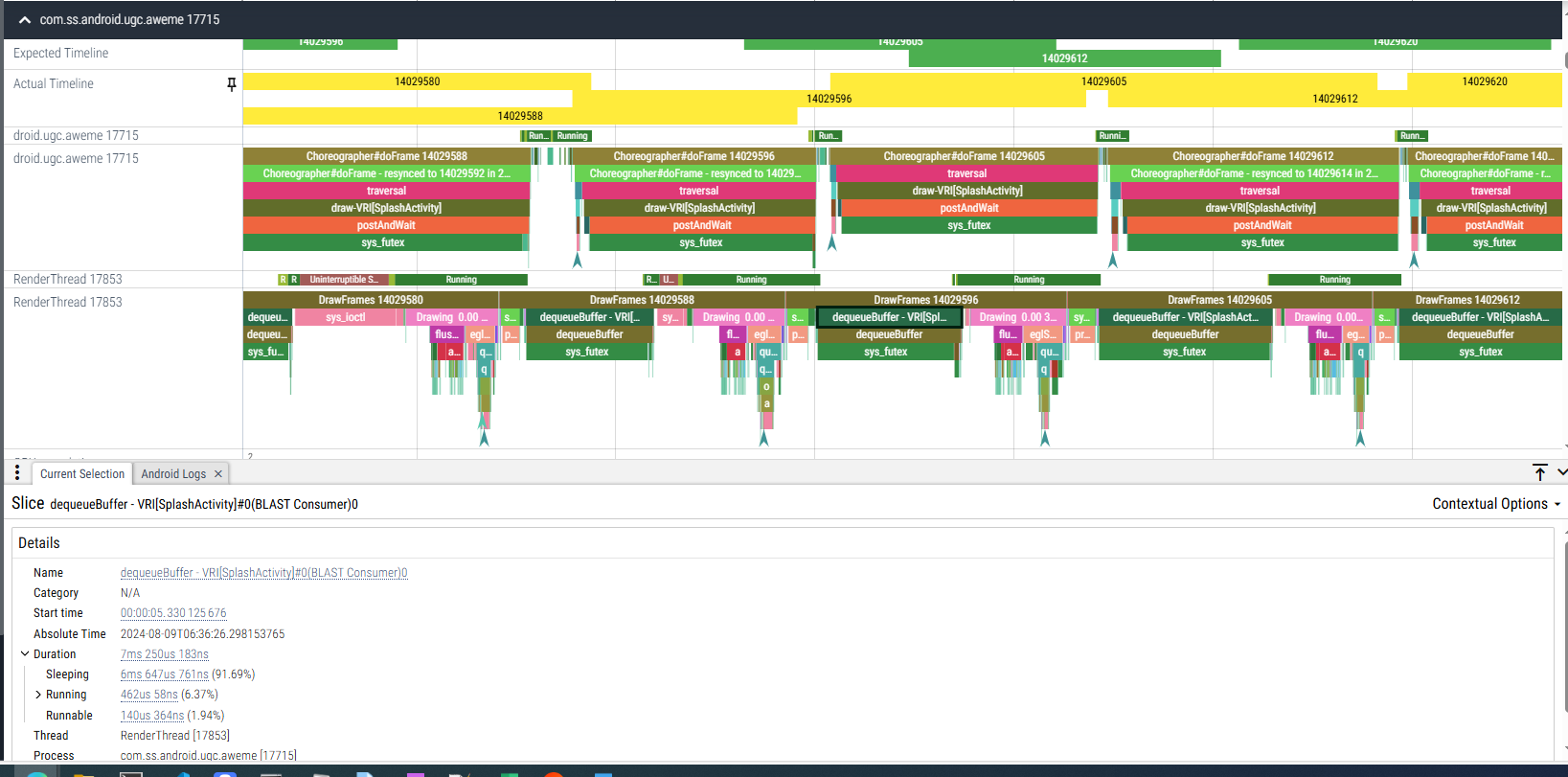
1. 280ms 左右的统计图



1. Drawcall统计图



1. Dequeue buffer 统计图



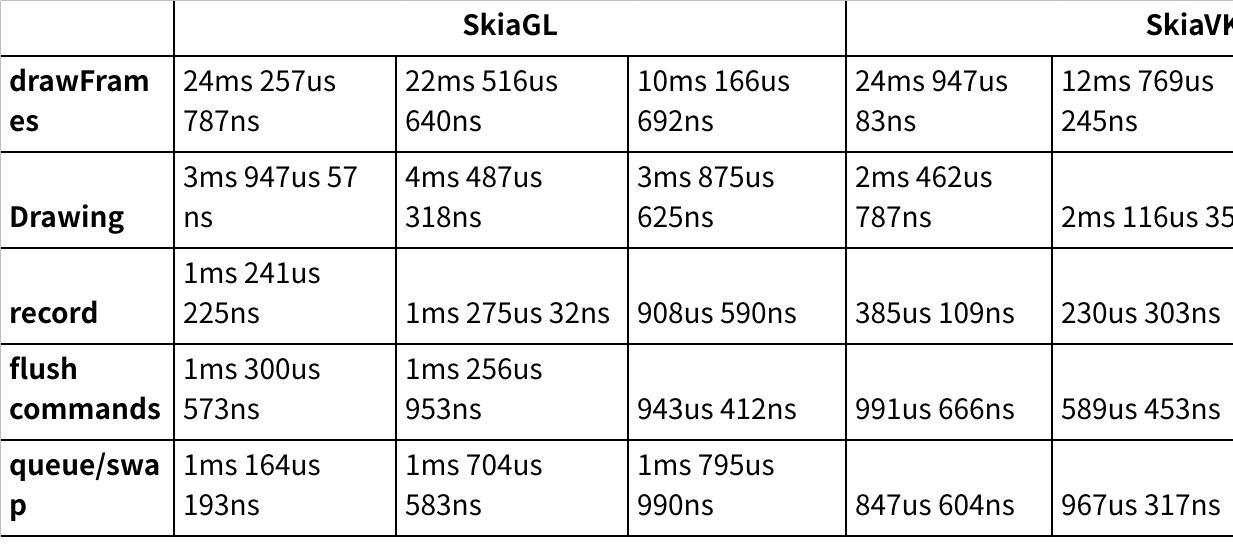
分析： Drawing 0.00. {2674， 2676， 2669} 每个调用四次，时间长分别为 18ms 242us 31ns, 17ms 865us 546ns, 17ms 188us 46ns, 平均每次调用时长4.4ms左右，图22. Drawcall 统计信息显示一个drawing 0.00 308.00 xxx.2676xxx duration 4ms 812us 421ns, 无任何阻塞，其中主要调用时skigpu::ganesh::opsTask::onExecute, eglSwapBuffersWithDamageKHR 印证渲染后端走SkiaGL, EGL窗口通道。

为了避免样本偏差，统计了SkiaGL和SkiaVK trace文件Drawing 0.00 xxx.xxx命令的时长信息，求取平均值。如下表所示分别通了388, 417个 drawing 指令样本，比较平均值发现，SkiaVK 明显时长较短。



**点击图片可查看完整电子表格**

下图是针对drawing 0.00 xx.xxx command执行个例的运行时间的统计对比:



**点击图片可查看完整电子表格**

上面相同drawFrames 时间的情况下，排除DeuqueBuffer的时长不同的影响，SkiaVK 在drawing调用中 command record过程占用时间相对较少，也能从侧面证明SkiaVk 相较于能减少CPU侧的负载。

DeuqueBuffer统计图中，dequeueBuffer 有6ms 647us 761ns的sleeping时长，总运行时长7ms 250us 183ns， 分析其他drawcall状况类似，相比与SkiaVK中dequeuebuffer用时将近17ms而言已经少了很多(不排除GL dequeue 用时超过VK dequeue流程的情况，只是作为一个横向比较，突出dequeue sleeping时间太长， 后面需要解决）。

**结论**

结合每个drawcall 中drawFrames调用中Drawing 0.00 xxx.xx调用时长对比，排除dequeueBuffer用时较长的情况，注重对比drawing 0.00 xxx.xx的用时， 分别从单个调用时长以及多个平均调用时长分析对比，SkiaVk 情况下的平均时长3ms 41us 121ns相比与SkiaGL的4ms 592us 907ns有显著的提升。

**总结**

综合streamline，simpleperf, perfetto 的测试数据分析，排除dequeueBuffer SkiaVK比SkiaGL阻塞（sleeping）时间更长的情况，SkiaVK相比与SkiaGL 在frame drawcall函数调用上cycle更少，在关键库libhwui, libGLES\_mali的调用数量上更少，且整体数量占renderThread线程调用数的占比更低。 因此总体而言抖音APP在系统切换到SkiaVK渲染后端的情况相比SkiaGL后端情况有所改善。

**遗留问题**

1. SkiaGL 和SkiaVK两种情况dequeueBuffer sleeping时间过长，需要继续定位分析。
2. 软件调用流程图未完成， vulkan指令下发调用libGLES\_mali.so需要继续分析。