Get started with Streamline

Version 8.8

Tutorial

Non-Confidential

Issue 00

102477_0808_00_en

Get started with Streamline

Tutorial

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	11 January 2019	Non-Confidential	Initial release.
0805-00	19 May 2023	Non-Confidential	Updated for Streamline 8.5 and general improvements.
0808-00	6 October 2023	Non-Confidential	Updated for Streamline 8.8 and general improvements.

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

This tutorial describes how to use Streamline to capture a profile of a debuggable application running on an unrooted Android device with an Arm GPU.

Follow the steps in each section to:

1. Complete the necessary Setup tasks
2. Capture a profile
3. Analyze the profile

You can also view these steps in this video.

See more about Streamline.

2. Setup tasks

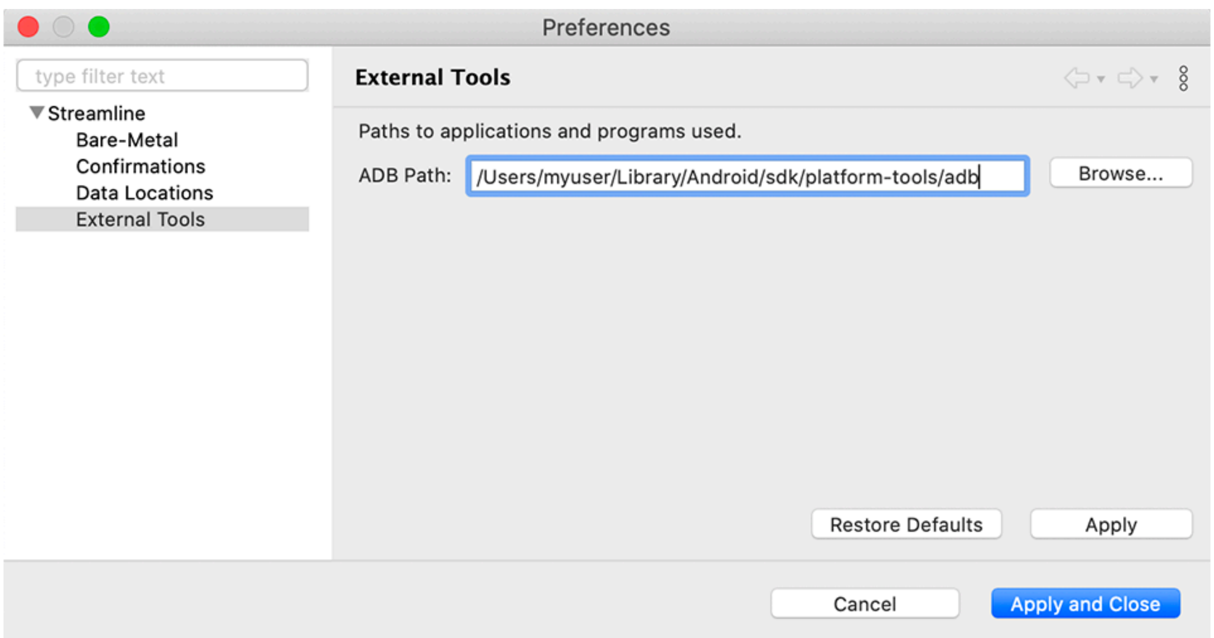
Follow these steps to set up your computer and device so that you can analyze your application with Streamline.

Before you begin

- Streamline uses Android Debug Bridge to connect to your device. Ensure you have ADB installed. ADB is available with the Android SDK platform tools, which are installed as part of Android Studio, or you can download them separately here.
- Edit your `PATH` environment variable to add the path to the Android SDK platform tools directory. This means that you can run ADB from any location on your machine, and that Streamline can automatically find ADB when attempting to connect to your device.

  If you decide not to do this, you must add the path to ADB in your Streamline preferences, under External Tools:

  Figure 2-1: Settting the path to ADB in Streamline



Procedure

1. Download Arm Mobile Studio and follow the installation instructions in the Arm Mobile Studio Release Note.
2. Connect your device to your computer through USB. Ensure that your device is set to Developer mode.
3. On your device, go to Settings > Developer Options and enable USB Debugging. If your device asks you to authorize connection to your computer, confirm this.

You can test the connection by entering the `adb devices` command in a command terminal. If successful, the command returns the device ID.

```
adb devices
List of devices attached
ce12345abcdf1a1234        device
```

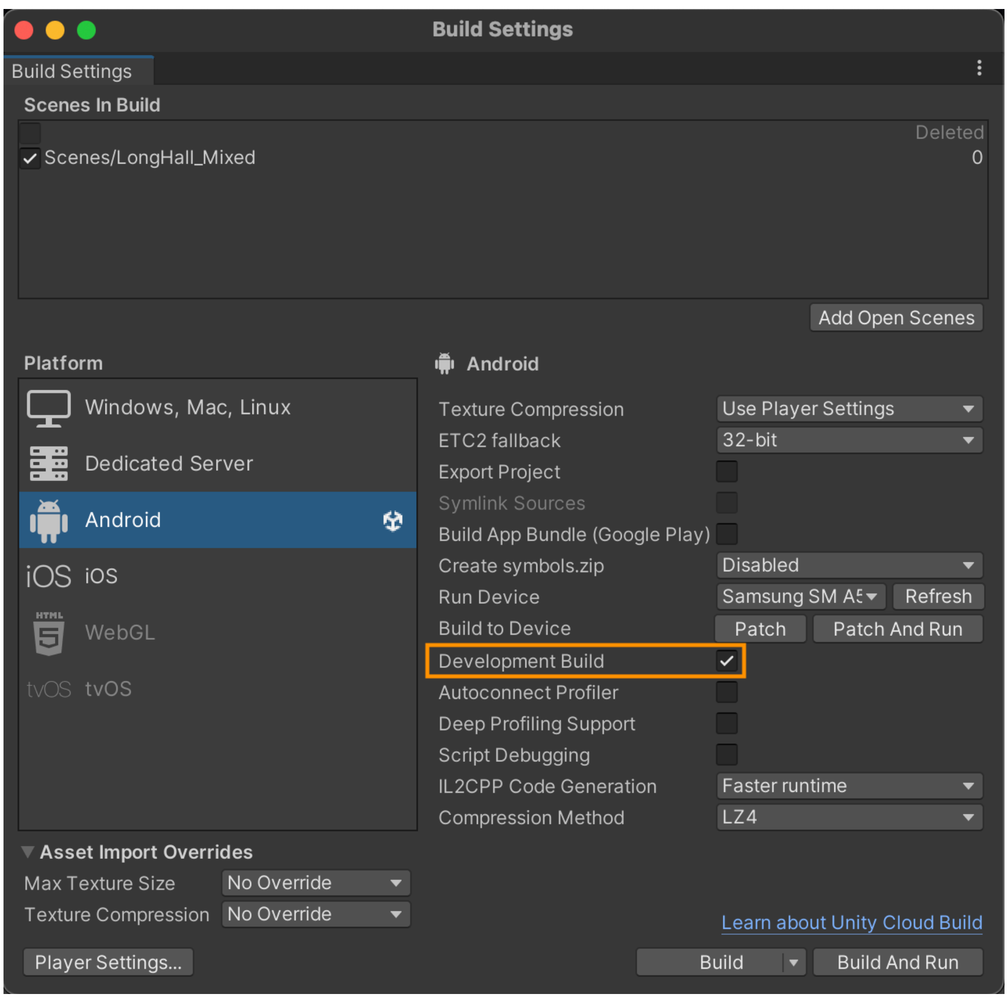
If you see that the device is listed as unauthorized, try disabling and re-enabling USB debugging on the device, and accept the authorization prompt to enable connection to the computer.

4. Install a debuggable build of the application you want to profile on the device.

   To do this in Android Studio, create a build variant that includes `debuggable true` in the build configuration. Or you can set `android:debuggable=true` in the application manifest file.

   To do this in Unity, select Development Build under File > Build Settings when building your application.

Figure 2-2: Unity Build Settings



Next steps

Now that your computer and device are connected and set up, the next step is to [Connect Streamline to your device].

3. Capture a profile

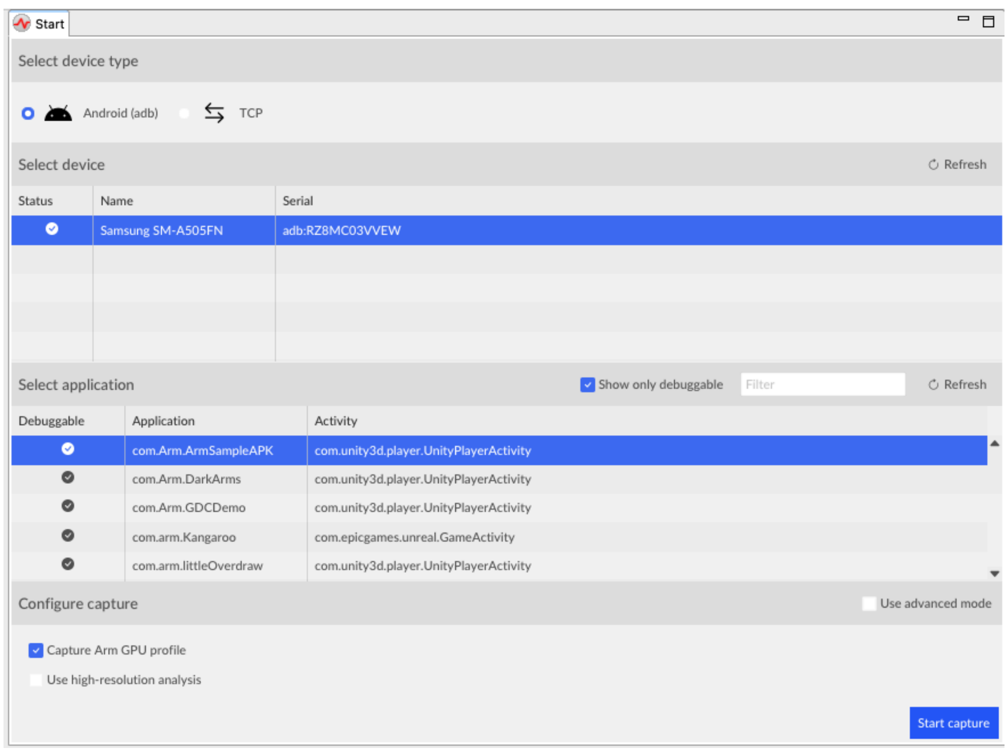
Follow these steps to capture a profile:

1. Launch Streamline:
    - On Windows, from the Start menu, navigate to the Arm Mobile Studio folder, and select the Streamline shortcut.
    - On macOS, go to the <install_dir>/streamline folder, and double-click the `Streamline.app` file.
    - On Linux, go to the <install_dir>/streamline folder, and run the `Streamline` file:

    ```
    cd <install_dir>/streamline
    ./Streamline
    ```

2. In the Start view in Streamline, ensure Android (adb) is selected. Select your device and the application you want to profile from the lists.

Figure 3-1: Streamline Start Tab

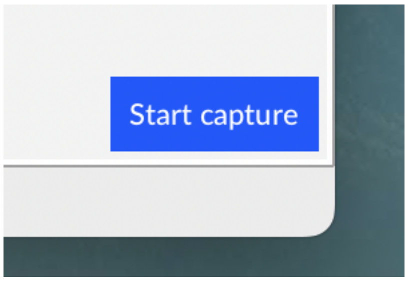


If you don’t see your device listed, check that your computer and device are configured correctly, as described in Setup tasks.

With Capture Arm GPU profile enabled in the Configure capture section, Streamline will automatically select a counter template appropriate for the GPU in your device. If you would rather build your own counter configuration, select Use advanced mode, and choose Select counters.

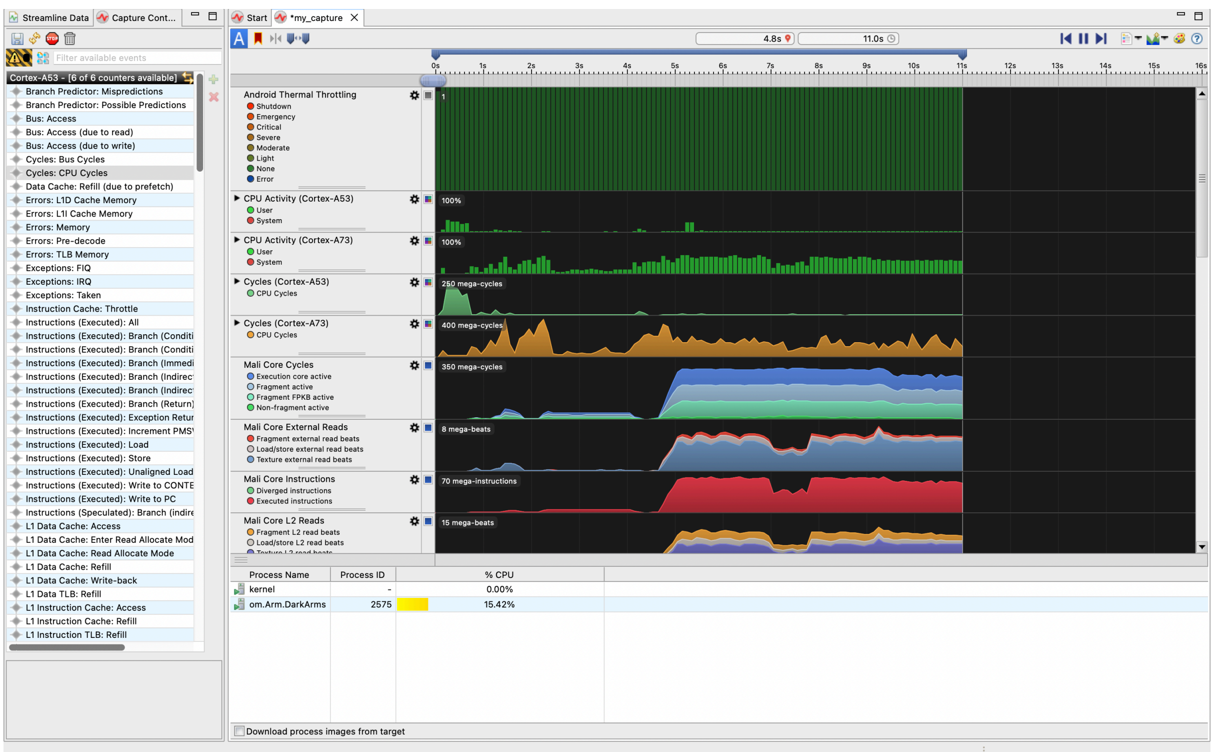
3. Select Start Capture.

Figure 3-2: Streamline start capture button



4. Specify the name and location of the capture file that Streamline will create when the capture is complete.
5. The application will start on the device, and the charts in Streamline will update in real time to show the data being captured.

Figure 3-3: Streamline live view



6. Unless you specified a capture duration in Advanced Settings, click Stop capture to end the capture:

Figure 3-4: Streamline stop capture button

Streamline saves the capture file in the location you specified and then prepares the capture for analysis.

Next steps

Now that you have captured some data from your game running on your device, you can Analyze the profile.

4. Analyze the profile

View the captured data in Streamline to see how the GPU and CPU in the device handled the workloads from your application. The charts show the performance counter activity for the counters in the selected template. Below the charts area is the details panel, which provides further metrics from your capture. Both the charts and details panel are aligned on the timeline.

This section describes:

- How to Navigate the data in Streamline
- How to Analyze performance using the different charts.

Note

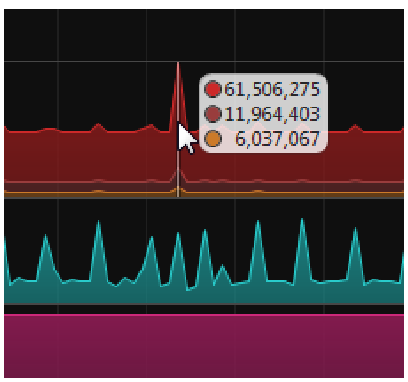
For detailed descriptions of all of the available counters for each Mali GPU, refer to the Mali GPU counter reference.

Navigate the data

Follow these steps to learn how to view and focus in on the charts data in Streamline.

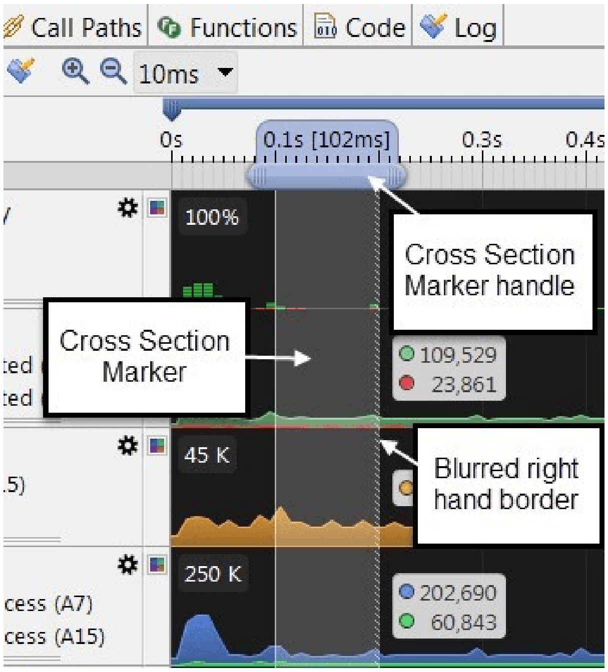
1. Control the granularity of the data by selecting the time unit. For example, if you choose 50ms, every color-coded unit in the details panel represents data captured during a 50ms window.
2. Hover over a chart to see the values at that point on the timeline.

Figure 4-1: Hover over a chart in Streamline to see the values



3. Click anywhere on the timeline and drag the handles on the cross-section marker to select a range of time to investigate more closely. The information shown in the details panel when in Processes and Samples modes updates to show data for the window of time you have defined.

Figure 4-2: Cross-section marker

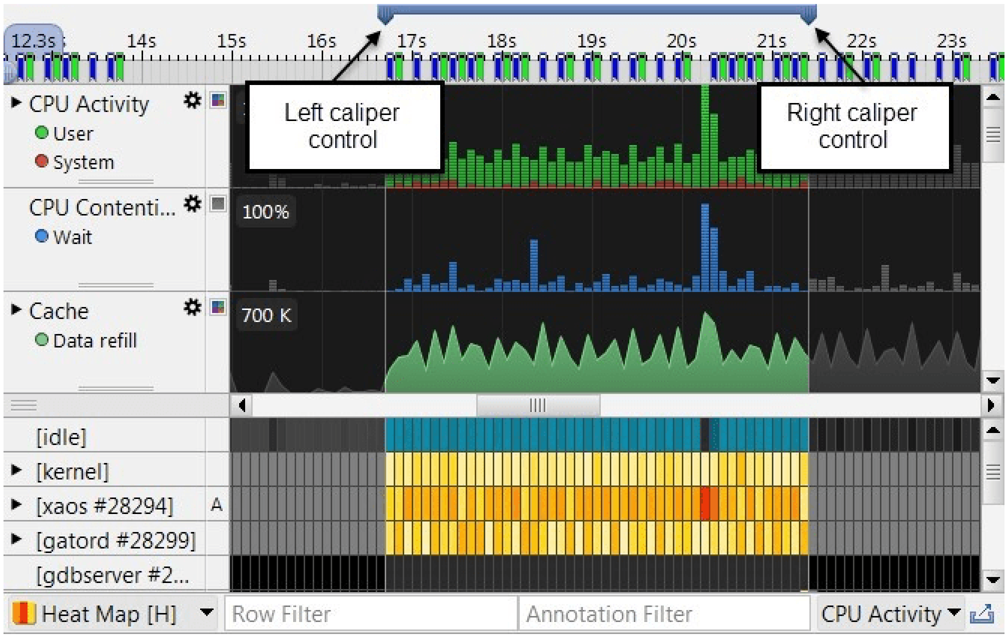


Note

If you define a region of time with the Cross-section marker, then change the view to a larger time unit where the Cross-section marker border can not sit precisely, the border is displayed as a blurred line.

4. Unlike the filter controls, moving and expanding the Cross-section marker does not affect the data in the other report views. To do this, drag the calipers to the required time region, or right-click on the timeline and select Set Left Caliper or Set Right Caliper. When you move the calipers, the Call Paths, Functions, and Code views update to show information for the selected region.

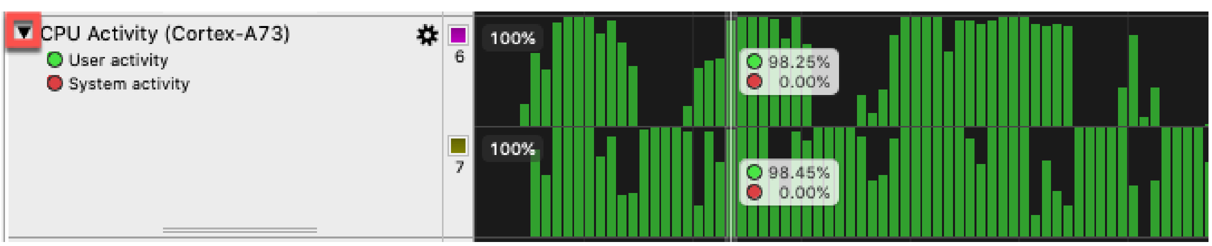
Figure 4-3: Select a time region with the calipers



Analyze performance

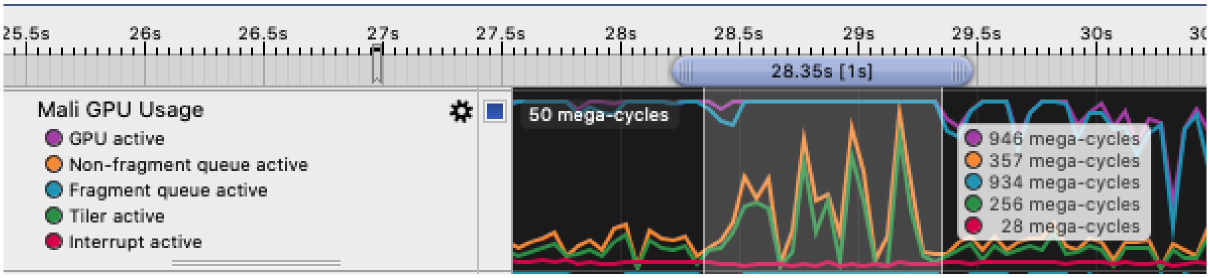
1. The CPU Activity charts show the activity of each processor cluster, presented as the percentage of each time slice that the CPU was running. Expand each chart to show the individual cores present inside the cluster. Note that this is the percentage of the time slice at the CPU frequency that is being used, not as a percentage of peak performance.

Figure 4-4: Streamline CPU activity chart



2. Use the GPU usage chart to check that the GPU is being kept busy, and the workload split between non-fragment and fragment processing. GPU workloads run asynchronously to the CPU, and the fragment and non-fragment queues can run in parallel to each other, provided that sufficient work is available to process.

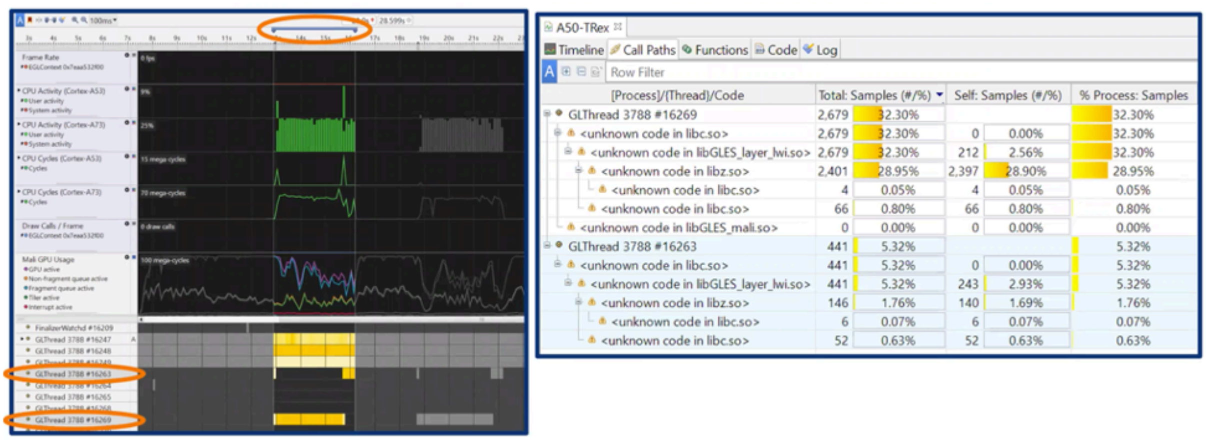
Figure 4-5: Streamline GPU usage



Look for areas where the GPU is active at the maximum frequency. This indicates that the application is GPU bound. These will look like flat lines, where there is no idle time. GPU active cycles will be approximately equal to the dominant work queue (non-fragment or fragment).

3. Check if drops in GPU activity correlate with spikes in CPU load, and whether those spikes are caused by a particular application thread. Use the calipers in Streamline to select the region where CPU spike occurs, then look at the Call Paths and Functions views to see which threads are active during the spike. If you don't have debug symbols, then you will only see library names in these views, but this can be enough to work out what's going on, because you can see which libraries are being accessed by each thread.

Figure 4-6: Filtering with the calipers



Tip

If you see a lot of time spent in `libGLES_mali.so`, this is driver overhead often caused by high draw call counts, bulk data upload, or shader compilation and linking.

---

4. Use the Mali Memory bandwidth chart to see the amount of memory traffic between the GPU and the downstream memory system. Minimizing GPU memory bandwidth is always a good optimization objective because memory accesses to external DRAM are very power intensive.

Figure 4-7: Mali memory bandwidth chart

5. Use the Mali Geometry usage and culling charts to check how efficiently objects are being rendered to the screen. Check how many primitives were sent to the GPU for processing, how many were visible on screen, and how many were culled.

Figure 4-8: Mali geometry usage

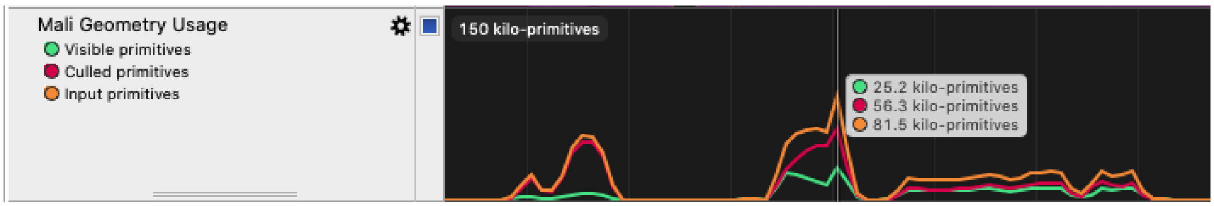
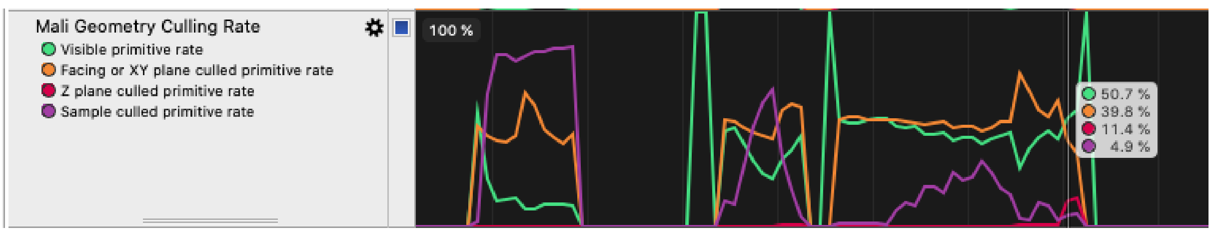


Figure 4-9: Mali geometry culling rate



For more information about culling, refer to the Android performance triage with Streamline tutorial.

6. Check how much work was discarded during early and late ZS (depth and stencil) testing. Early ZS testing is relatively inexpensive, because it happens before any pixels are colored in by fragment shading. Late ZS testing happens after all the fragment shading work has been done, therefore it is expensive and should be avoided.

Figure 4-10: Mali Early ZS Rate

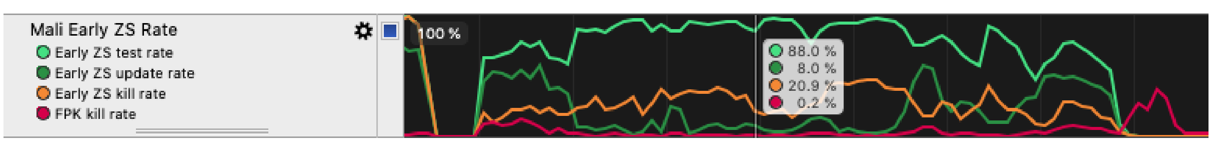
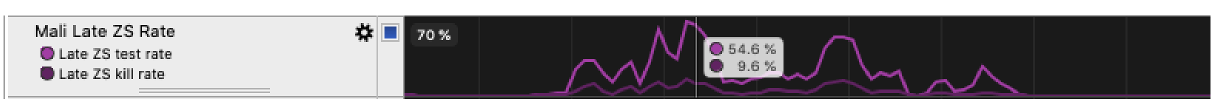


Figure 4-11: Mali Late ZS Rate



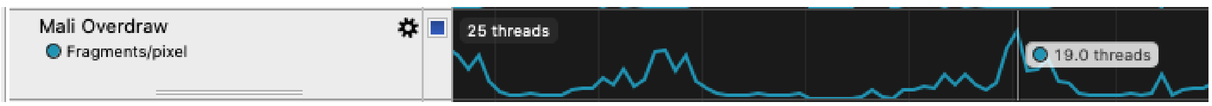
Tip

To get the benefit of early ZS, your application must pass in geometry in a front-to-back render order, starting at the point closest to the camera and moving further away.

---

7. Check the Mali overdraw chart to check whether the level of overdraw is acceptable. This chart shows the number of fragments shaded per output pixel. Ideally, this number should be less than 3.

   Figure 4-12: Mali overdraw chart in Streamline

   Refer to our optimization advice for reducing overdraw.

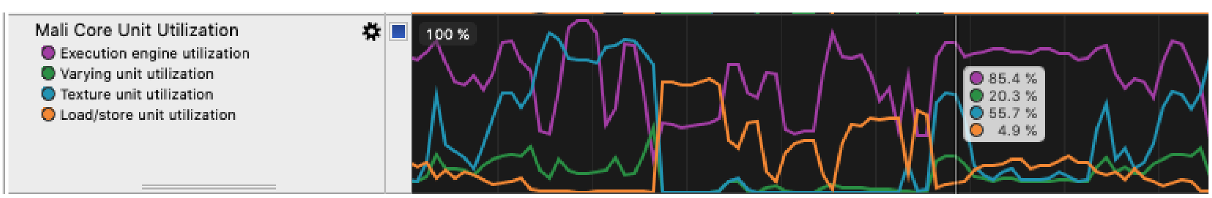
Tip

Use Graphics Analyzer to explore your object geometry in more detail. You can capture the level of overdraw in a scene, and explore it to see which objects are causing the problem.

---

8. There are a range of charts to help you understand how your shaders are performing. The Mali Core Unit Utilization chart shows the percentage utilization of the functional units inside the shader core; the execution engine, the varying unit, the texture unit and the load/store unit. The most heavily utilized functional unit should be the target for optimizations to improve performance, although reducing load on any of the units is good for energy efficiency.

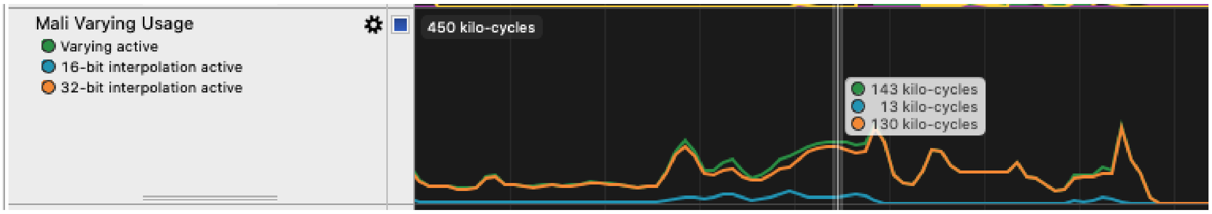
   Figure 4-13: Mali Core Unit Utilization

   If the texture unit utilization is a bottleneck, check the texturing charts to look for ways to optimize.

   In order to be efficient, shader cores within a GPU should execute calculations at moderate precision - in most cases, `mediump` (16-bit precision) is sufficient. The Mali Varying Usage chart shows the amount of interpolation processed by the varying unit, at 16-bit (`mediump`) or 32-bit (`highp`) precision.

Figure 4-14: Mali Varying usage



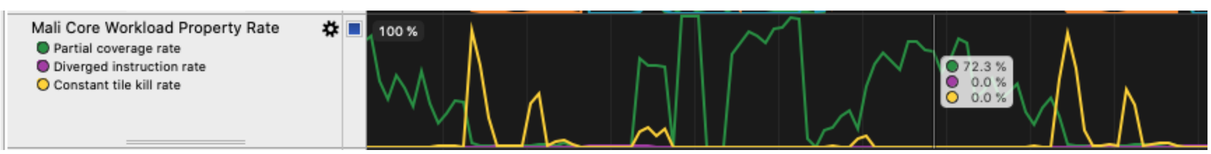
Tip

16-bit interpolation is twice as fast as 32-bit interpolation. It is recommended to use `mediump` (16-bit) varying inputs to fragment shaders, rather than `highp` whenever possible.

The Mali Core Workload Property Rate gives information about shader workload behavior that could be optimized:

- Partial coverage - Warps that contain samples with no coverage. A high number suggests that content has a high density of very small triangles, or microtriangles, which are disproportionately expensive to process. Check that the complexity of objects is appropriate for their position on screen. Use mesh LODs to reduce complexity when the object is far away from the camera.
- Diverged instructions - Instructions that have control flow divergence across the warp.
- Constant tile kill - Tile writes that are killed by the transaction elimination CRC check. A high number indicates that a significant part of the framebuffer is static from frame to frame.

Figure 4-15: Mali Core Workload Property rate



For information about further shader charts, refer to the Android performance triage with Streamline tutorial.

Next steps

Refer to Android performance triage with Streamline to learn more about how to interpret the data reported in the charts.

Full descriptions of all of the Mali GPU performance counters can be found in the GPU counter reference documentation.

For more information about the different views and capabilities of Streamline, see Analyze your capture in the Arm Streamline user guide.

Once you have discovered a problem in your game with Streamline, you can analyze the frames where the problem exists, with Graphics Analyzer.

5. Related information

Some useful further reading:

- Refer to Android performance triage with Streamline to learn more about how to interpret the charts in Streamline.
- Refer to the Streamline documentation for detailed topics.
- For a demo of Streamline, watch Episode 3.3 of the Arm Mali GPU training.
- For detailed descriptions of all of the available counters for each Mali GPU, refer to the Arm GPU performance counter reference guides.
- If you need a regular summary report for the data that Streamline collects, consider using Performance Advisor to create these.
- Once you have discovered a problem in your game with Streamline, you can analyze the frames where the problem exists, with Graphics Analyzer.