Performance Advisor

Version 1.2

User Guide

102009_0102_00_en

Performance Advisor

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	28 February 2020	Non-Confidential	New document for v1.0.
0101-00	29 May 2020	Non-Confidential	New document for v1.1.
0102-00	26 August 2020	Non-Confidential	New document for v1.2.

Confidentiality Status

Product Status

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

Web Address

developer.arm.com

Contents

Performance Advisor User Guide

Preface

This preface introduces the *Performance Advisor User Guide*.

It contains the following:
- *About this book* on page 7.

About this book

This book describes how to install and use Arm® Performance Advisor to generate reports from your Arm Streamline capture data.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction to Performance Advisor
This section introduces the Performance Advisor tool and the workflows that it is designed to handle.

Chapter 2 Before you begin
Set up Arm Mobile Studio and integrate Performance Advisor with your application by following the steps in this section.

Chapter 3 Quick start guide
Performance Advisor runs on a capture file generated from Streamline. Follow the steps in this section when you are ready to perform an interactive capture.

Chapter 4 Running Performance Advisor in continuous integration workflows
Regular performance reports enable you to get instant feedback throughout your development cycle. With an Arm Mobile Studio Professional license, you can integrate Performance Advisor into your continuous integration workflow. This workflow enables you to automatically generate daily reports that help your team monitor how changes during the development cycle impact performance. Also, you can automatically generate machine-readable JSON reports that you can import into your existing performance regression tracking systems.

Chapter 5 Capturing a slow frame
Identify slow frames by using the lightweight interceptor (LWI) in different modes. Before you can use the LWI, you must first integrate it with your application.

Chapter 6 Adding semantic input to the reports
Performance Advisor can use semantic information that the application provides as key input data when generating the analysis reports.

Appendix A Analytics

Appendix B Command-line options

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<and>`
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:
- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Performance Advisor User Guide*.
- The number 102009_0102_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- *Arm® Developer*.
- *Arm® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Introduction to Performance Advisor

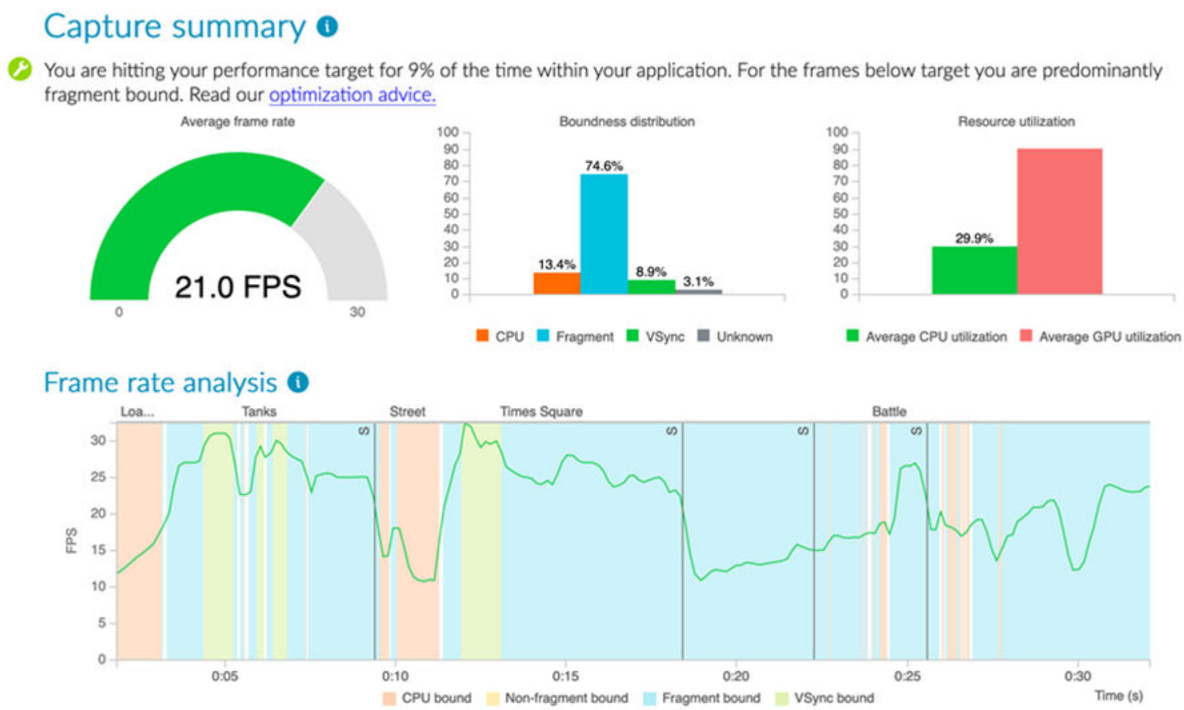
This section introduces the Performance Advisor tool and the workflows that it is designed to handle.

It contains the following sections:

1.1 Overview of Performance Advisor

Performance Advisor analyzes performance data from your Streamline capture, and generates a report that shows how your application is performing on your mobile device.

The summary at the top of the report shows whether your application is non-fragment or fragment bound. See how efficiently your CPU and GPU are running, your boundness split, and whether you are achieving your required frame rate.



To help you further understand how your application is performing over time, you can analyze key metrics shown on a series of charts:

Overdraw per pixel
Identify problems caused by transparency or rendering order, by monitoring the number of times pixels are shaded before they are displayed.

Draw calls per frame
To identify CPU workload inefficiencies, check the absolute number of draw calls per frame.

Primitives per frame
See how many input primitives are being processed per frame, and how many of them are visible in the scene.

Pixels per frame
See the total number of pixels being rendered per frame. This metric helps you to rule out problems caused by changes in the application render pass configuration. For example, extra passes for new shadow casters or post-processing effects.

Shader cycles per frame
The total number of shader cycles per frame, broken down by pipeline, so that you can see which workloads are occupying the GPU.

GPU cycles per frame
See how the GPU is processing non-fragment and fragment workloads, and whether the shader core resources are balanced.

GPU bandwidth per frame

Monitor the distribution of GPU bandwidth, including the breakdown between reads and writes, so that you can minimize external memory accesses to save energy.

CPU cycles per frame

See the consumption of CPU cycles per rendered frame. This metric helps you to validate improvements and regressions, which might not be visible in the CPU utilization charts.

Running the Performance Advisor report regularly enables you to get performance feedback throughout the development cycle. You can also integrate Performance Advisor in your performance regression workflows, by generating machine-readable JSON reports that you can import into other tracking systems.

Performance Advisor can identify scheduling issues that prevent you from achieving your target frame rate, and provide advice on how to resolve it. See *3.4 Generate a performance report* on page 3-29 for more information.

Related concepts

1.2 Performance report example on page 1-12

Related references

Chapter 2 Before you begin on page 2-16
Chapter 3 Quick start guide on page 3-24

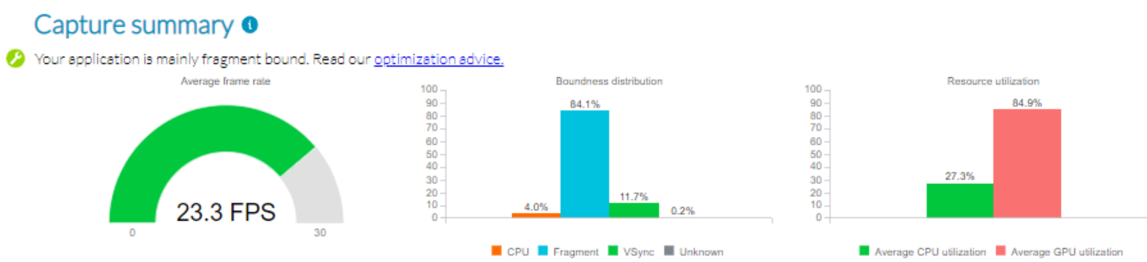
1.2 Performance report example

In this example, we will look at the charts in the Performance Advisor report to review the performance of your application. See how to use the report to investigate problems with any scenes in your application that are not performing well.

We have generated a Performance Advisor report from a Streamline capture file, and saved it as an HTML file.

Report summary

First look at the charts at the top of the report. These three charts provide a summary of how your application is performing for the duration of your capture. To identify any changes to your application throughout your development process, we recommend that you monitor these charts regularly.



Here, we can see that the average frame rate for the capture is not achieving the target of 30fps. When we check the boundness distribution, we can see that the application is fragment bound. The utilization chart confirms that a graphical problem is causing this drop in frame rate.

Analyze frame rate

To see how the frame rate changes throughout the duration of your capture, check the **FPS analysis** chart.

Note

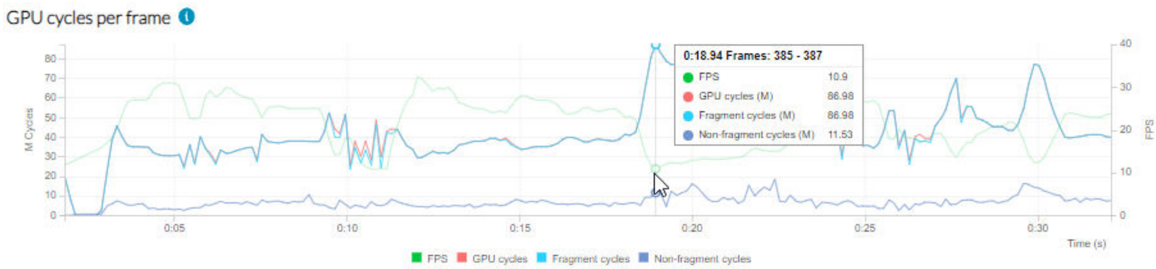
In this capture, we have used the `lwi_me.py` script to take a screenshot if the frame rate goes below 20fps. We have also specified a number of frames between captures to ensure that we do not capture too many images.

The background color of this chart is blue, indicating that the GPU in the device is struggling to process fragment workloads. We can also see that the frame rate has dropped below the target threshold of 20 in three places, so Performance Advisor has captured these frames. To see an image of the frame, hover the cursor on the screen capture icon ⟲. In the image, you might be able to see which graphical element is causing the frame rate to drop. To get a better understanding about what is happening in your application, continue your analysis by looking at the GPU behavior metrics.

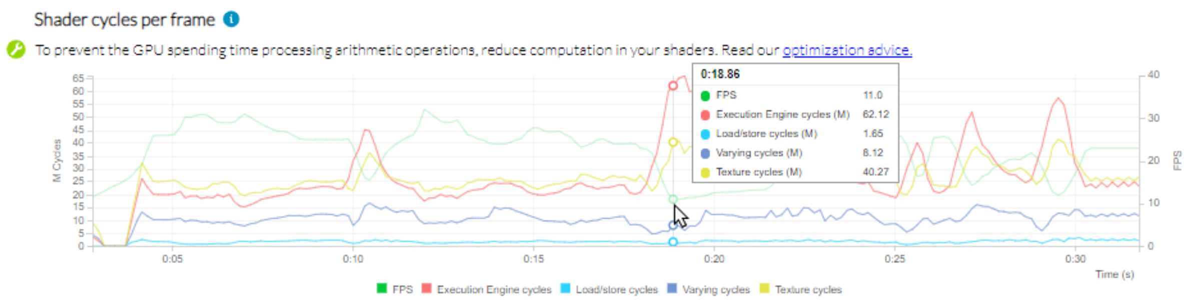
Investigate GPU behavior

Scroll through the GPU behavior charts to find any strong correlation between the GPU metric and a drop in the frame rate. Performance Advisor provides advice above a chart where it finds a potential problem. You can also get further advice on optimizing your code by clicking the accompanying link to our developer website.

The **GPU cycles per frame** chart shows that the frame rate drops when the number of fragment cycles increases.

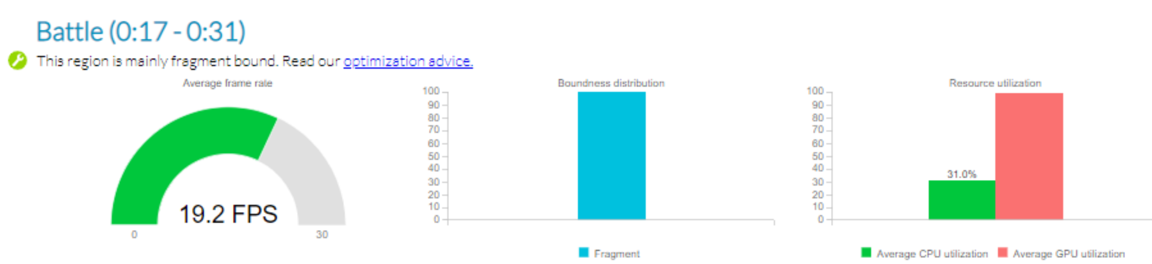


The **Shader cycles per frame** chart shows that the drop in frame rate correlates with high numbers of execution engine cycles.



This chart shows that the GPU is busy with arithmetic operations. We need to reduce the complexity of the shaders, and textures that we used. From here, we can click through to read *optimization advice* about how to improve shader performance.

We annotated the capture with region names to help us identify what is happening at different parts of the application. If we scroll down the report, we can analyze in more detail the specific region that we are interested in.



Next steps

When you have identified a performance problem with Performance Advisor, use the other tools in the Arm Mobile Studio suite to explore your problem in more detail.

Related information

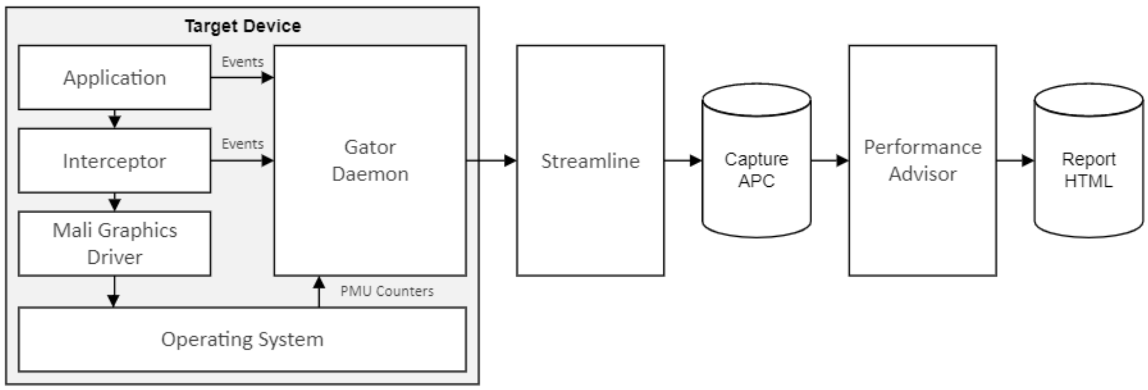
Get started with Arm Mobile Studio

1.3 Performance Advisor workflows

You can use Performance Advisor with Streamline in several different workflows, enabling you to solve multiple different types of problem.

Interactive capture with Performance Advisor report

You can use Performance Advisor to assist with a manual debug session. Manually connect to a target and capture data using Streamline. Use Performance Advisor to post-process the dataset to provide an initial quick analysis.

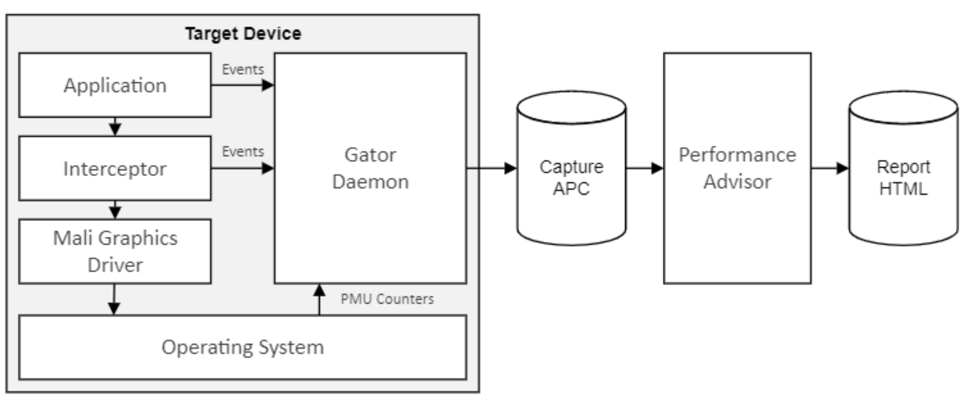


Automated capture with Performance Advisor report

Note

This feature is license managed and might not be available with some editions of Arm Mobile Studio. For more information, see *Arm Mobile Studio Professional Edition*.

You can use Performance Advisor as part of a continuous integration (CI) workflow. To capture data from automated game tests, without using the Streamline GUI on the host, integrate the gator daemon from Streamline into a nightly test system. Use Performance Advisor to generate a report, which can be published automatically. This workflow enables a QA team to review the status each morning.



Automated capture with Performance Advisor data export

Note

This feature is license managed and might not be available with some editions of Arm Mobile Studio. For more information, see *Arm Mobile Studio Professional Edition*.

You can use Streamline and Performance Advisor to generate a machine-readable JSON report. You can import data from the JSON report into other QA test reporting systems, allowing automated regression tracking of in-depth workload metrics. See *Chapter 4 Running Performance Advisor in continuous integration workflows* on page 4-31 for more information.

The APC data file that the CI workflow creates is a full Streamline capture that you can import into the Streamline GUI. Arm recommends that you store the APC data file alongside other build artifacts. If Performance Advisor reports a problem, it is then immediately available for manual investigation in Streamline.

For more information about using Streamline for profiling graphical applications running on Mali GPUs, see the Arm Community blog *Accelerating Mali GPU analysis using Arm Mobile Studio*.

Using Streamline and Graphics Analyzer for further deep-dive analysis

The Performance Advisor report shows where your application is causing a problem. You can then use the other tools in Arm Mobile Studio suite to investigate any problems in more detail.

Streamline

Capture a profile of your application running on a mobile device and see where your system spends most of its time. Use interactive charts and comprehensive data visualizations to identify whether CPU processing or GPU rendering are causing any performance bottlenecks.

Graphics Analyzer

Graphics Analyzer enables you to evaluate all the OpenGL ES or Vulkan API calls your application makes, as it runs on an Android device. Explore the scenes in your game frame-by-frame, draw call-by-draw call, to identify rendering defects, or opportunities to optimize performance. For more information, see *Graphics Analyzer* on the Arm Developer website.

Chapter 2
Before you begin

Set up Arm Mobile Studio and integrate Performance Advisor with your application by following the steps in this section.

It contains the following sections:

2.1 Set up your host machine

To use Performance Advisor, download and install the Arm Mobile Studio suite, then install the necessary software and set up environment variables on your host machine.

Procedure

1. Download Arm Mobile Studio from *https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/downloads*.
2. Install Arm Mobile Studio using the instructions at *https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/installation*.
3. Install Python 3.6 (or higher). Arm Mobile Studio uses Python to run the provided `lwi_me.py` and `gator_me.py` script, which uses the `gatord` agent to connect Streamline to your Android target.
4. Install Android Debug Bridge (adb). Arm Mobile Studio uses the adb utility to connect to the target device. Download the latest version of adb from the Android SDK platform tools (*https://developer.android.com/studio/releases/platform-tools*).
5. Edit your `PATH` environment variable to add the paths to the Performance Advisor, Python3, and Android SDK platform tools directories.

Next Steps

See *2.2 Set up your device* on page 2-18 for information about preparing your device for profiling your application.

2.2 Set up your device

To use Performance Advisor, set up your device with the application you want to profile.

Note

A list of the recommended devices that support Arm Mobile Studio is available from *https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/support/supported-devices*.

Procedure

1. Set your device to *Developer Mode*.
2. Select **Settings** > **Developer options** and enable **USB debugging**.
3. Connect the device to the host machine through USB. If the connection is successful, running the `adb devices` command on the host returns your device ID:

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

4. For devices running Android 9 or earlier, you need to add a library file to your application, to enable Performance Advisor to collect frame rate and graphics API call counts. See *2.3 Integrate Performance Advisor with your application* on page 2-19 for instructions on how to do this.
5. Install a debuggable build of your application on the device:
   - If you are not using Unity, enable the `android:debuggable` setting in the application manifest file, as described in *https://developer.android.com/guide/topics/manifest/application-element*.
   - In Unity, when building your application, select the **Development Build** option in **Build Settings**.

Next Steps

3.1 Connect Streamline to your device on page 3-25

2.3 Integrate Performance Advisor with your application

For devices running Android 9 or earlier, package the lightweight interceptor library (LWI) with your application. Performance Advisor uses the LWI to collect performance data, such as frame rate and API call counts, from your application.

For devices running Android 10 or later, you do not need to package the library file with your application. Instead use the `--lwi-gles-layer-lib-path` or `--lwi-vk-layer-lib-path` options to specify the layer library file when you *3.1 Connect Streamline to your device* on page 3-25.

The LWI enables you to capture performance data automatically from your application, such as frame rate and frame captures. It is a lighter version of the Graphics Analyzer interceptor.

The LWI enables you to automatically capture data in the following situations:
- To automatically detect frame boundaries, or other API statistics, instead of manually embedding frame markers into the application.
- To identify slow parts of your application, you can capture a screenshot when your application goes below a threshold value that you configure.

Note

If you want to analyze your application with Graphics Analyzer, you must use a different library file, packaged in the same way as the LWI.
- For Unity applications, see *Prepare your Unity application*.
- For applications not using Unity, see *Get started with Graphics Analyzer*.

If you intend to capture frames when the frame rate goes below a specified value, you must use the LWI instead.

OpenGL ES

For OpenGL ES applications, package the required library file `libMGD.so`, which is provided in your Arm Mobile Studio package:

```
<install_directory>/performance_advisor/lwi/target/android/arm/unrooted/
```

Two versions of the library are provided:
- For 64-bit targets, use the library file located in the `arm64-v8a` directory.
- For 32-bit targets, use the library file located in the `armeabi-v7a` directory.

Note

You can package one or both interceptor libraries depending on the requirements of your application.

Vulkan

For Vulkan applications, package the required Vulkan layer file, which is provided in your Arm Mobile Studio package:

```
<install_directory>/performance_advisor/lwi/target/android/arm/rooted/
```

Note

If your target device is running Android 9 or above, you do not need to package the Vulkan layer with the application. Instead specify the path to the Vulkan layer when running the target connection script.

Two versions of the library are provided:
- For 64-bit targets, use the library file located in the `arm64-v8a` directory.
- For 32-bit targets, use the library file located in the `armeabi-v7a` directory.

Next steps

Continue with the appropriate instructions for your project:

- *2.3.1 Prepare your Unity project* on page 2-20
- *2.3.2 Prepare your Android Studio project* on page 2-23

2.3.1 Prepare your Unity project

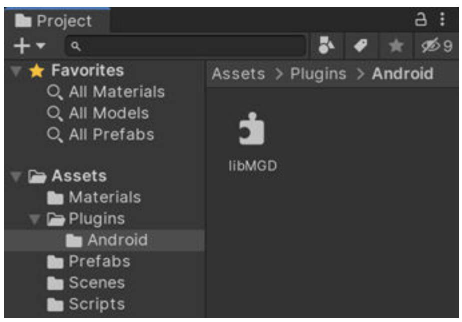
Copy the library file or Vulkan layer file into Unity, and set the necessary attributes and settings. Then build your APK and install it on your device. You are then ready to perform a capture.

Prerequisites

Locate the required library file or Vulkan layer file, as described in *2.3 Integrate Performance Advisor with your application* on page 2-19.

Procedure

1. Copy the required `libMGD.so` file or Vulkan layer into the `Assets/Plugins/Android/` directory in your Unity project. Create this directory if it does not exist.

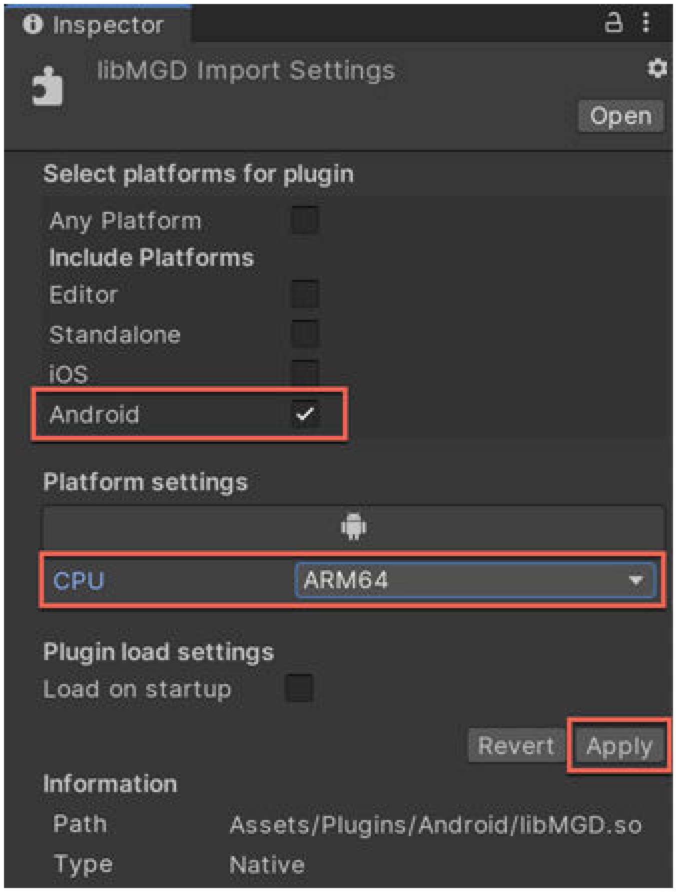
   If you are packaging both interceptor libraries:

   - Create two directories in the `Assets/Plugins/` directory. For example, `armv7` and `armv8`.
   - Create a directory called `Android` in each of these directories.
   - Copy each `libMGD.so` file into the appropriate `Android` directory.

2. Select the library in Unity and set the following attributes in the Inspector:

   - Under **Select platforms for plugin**, select **Android**.
   - Under **Platform settings**, set the **CPU** architecture to **ARM64** for 64-bit applications, or **ARMv7** for 32-bit applications.

   Click **Apply**.

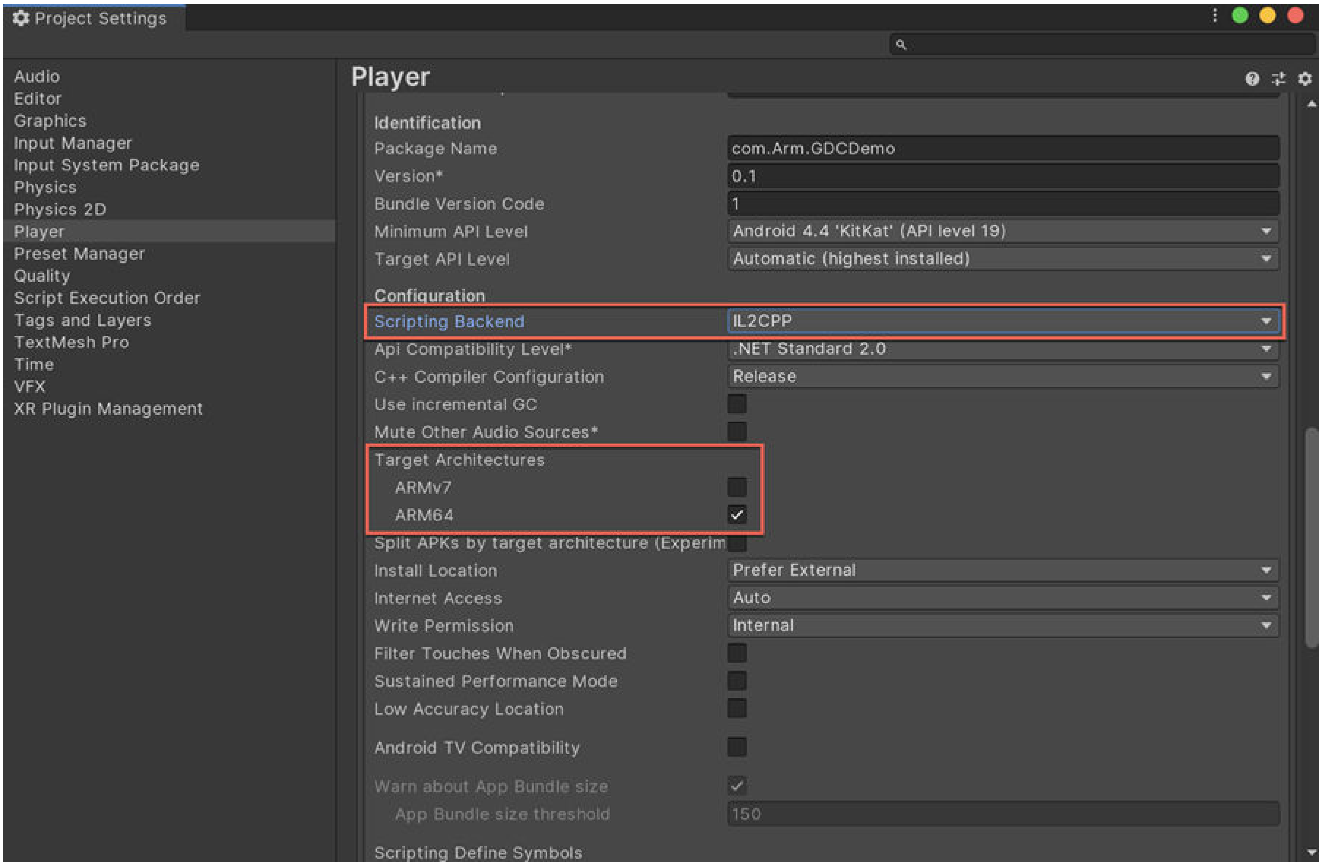


3. Select **File** > **Build Settings**, then select **Player Settings**.
4. Under **Identification**, set **Target API Level** to the required Android version.

Note

By default, **Target API Level** is set to the latest version of the Android SDK tools that you have installed. If you change to a lower API level, ensure that you have the SDK tools for that version installed. If you build for a higher API version later, change this setting accordingly.

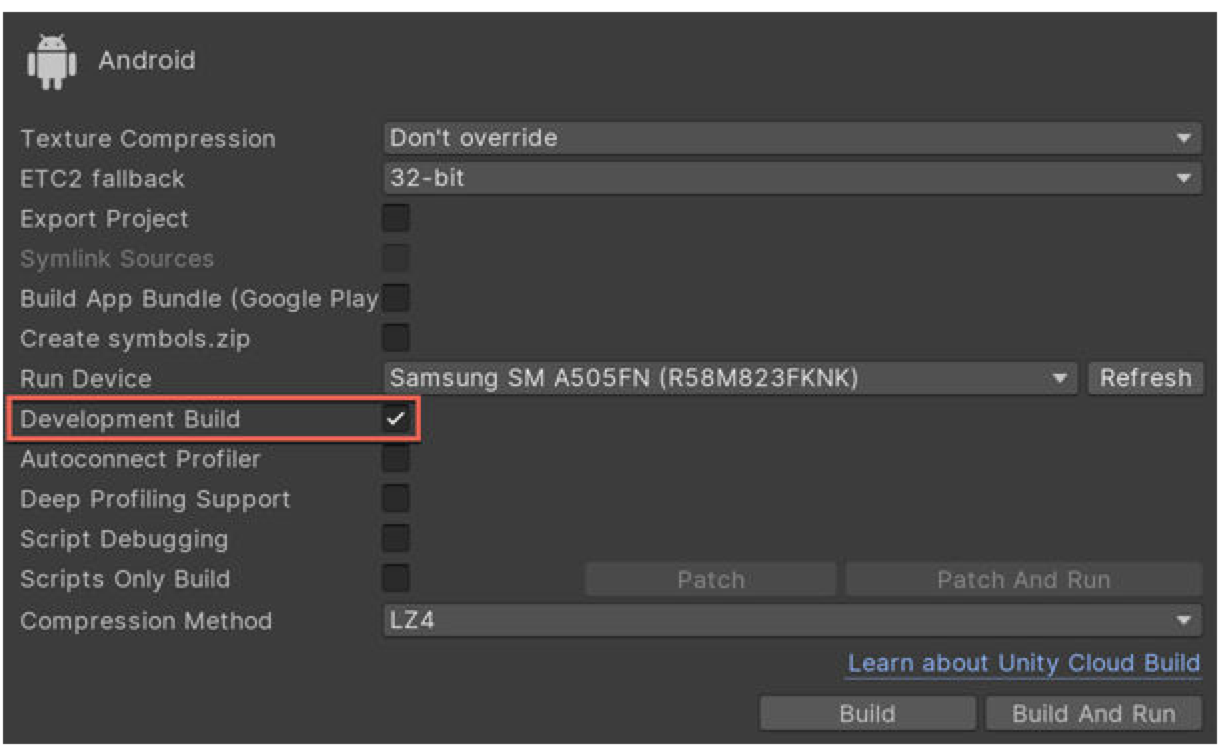
5. Under **Configuration**, set the following options to build a 64-bit application:
   1. Set the scripting backend in Unity to work with 64-bit targets. Set **Scripting Backend** to **IL2CPP**. For more information about IL2CPP, refer to the Unity documentation.
   2. Under **Target Architectures**, select **ARM64**.



To build a 32-bit application:

1. Leave the scripting backend at its default setting, **Mono**.
2. Under **Target Architectures**, select **ARM7**.

6. Close the **Player Settings**. In the **Build Settings**, select the **Development Build** checkbox. This option ensures that your application is marked as debuggable in the Android application manifest.



7. To build your APK and install it on your device in one step, select **Build and Run**. Alternatively, select **Build** to build the APK and then install it on your device using Android Debug Bridge:

```
adb install -r YourApplication.apk
```

Next Steps

Perform an interactive capture, see *3.1 Connect Streamline to your device* on page 3-25.

2.3.2 Prepare your Android Studio project

Supply the path to the library file or Vulkan layer file, and load the library in your code. Then build your APK and install it on your device. You are then ready to perform a capture.

Prerequisites

Locate the required library file or Vulkan layer file, as described in *2.3 Integrate Performance Advisor with your application* on page 2-19.

Procedure

1. Supply the path to the LWI library or Vulkan layer files in your applications gradle file.
   - For OpenGL ES applications, supply the path to the LWI library in your applications gradle file:

```
android {
    sourceSets {
        main {
            jniLibs.srcDirs += 'install_directory/performance_advisor/lwi/target/
android/arm/unrooted/'
        }
    }
}
```

   - For Vulkan applications, supply the path to the Vulkan layer files in your applications gradle file:

```
android {
    sourceSets {
        main {
            jniLibs.srcDirs += '<install_directory>/performance_advisor/lwi/target/
android/arm/rooted/'
        }
    }
}
```

2. Load the library in a static block in your code:

```
static
{
    try
    {
        System.loadLibrary("LWI");
    }
    catch (UnsatisfiedLinkError e)
    { ... }
}
```

3. Build your APK and install it on your device.

Next Steps

Perform an interactive capture, see *3.1 Connect Streamline to your device* on page 3-25.

Chapter 3
Quick start guide

Performance Advisor runs on a capture file generated from Streamline. Follow the steps in this section when you are ready to perform an interactive capture.

Note

If you already have the capture files, you can go straight to *3.4 Generate a performance report* on page 3-29.

You can also watch a demonstration of the steps on the *Android profiling with Performance Advisor* video on *YouTube* or *Youku*.

It contains the following sections:

3.1 Connect Streamline to your device

Arm provides a Python script, lwi_me.py that makes connecting to your device easy. Run the script so that Streamline can connect to your device, and collect data.

Procedure

1. Open a command terminal on your host machine and navigate to the Performance Advisor installation directory, <install_directory>/performance_advisor/lwi/helpers.
2. Run the lwi_me.py Python script:

```
python3 lwi_me.py --daemon <path_to_gatord> \
  [--lwi-gles-layer-lib-path | --lwi-vk-layer-lib-path \
  <path_to_Android10_layer_lib>]
```

   Use the --daemon option to specify the path to the gatord binary that you want to install on your device to collect data. This file is provided in your installation directory in two versions:

   - For 32-bit applications, use <install_directory>/streamline/bin/arm/gatord.
   - For 64-bit applications, use <install_directory>/streamline/bin/arm64/gatord.

   For Android 10, use --lwi-gles-layer-lib-path or --lwi-vk-layer-lib-path to specify the path to the OpenGL ES or Vulkan layer library file for Android 10 devices. These files are provided in your installation directory. Libraries for both 32-bit and 64-bit applications are stored in different folders; armeabi-v7a for 32-bit applications, and arm64-v8a for 64-bit applications.

   - The Open GL ES layer library file libGLES_layer_lwi.so is located in: <install_directory>/performance_advisor/lwi/target/android/arm/unrooted/{arm64-v8a|armeabi-v7a}/libGLES_layer_lwi.so
   - The Vulkan layer library file libVkLayerLWI64.so is located in: <install_directory>/performance_advisor/lwi/target/android/arm/rooted/{arm64-v8a|armeabi-v7a}/libVkLayerLWI64.so

   For example:

```
python3 lwi_me.py --daemon ../../../streamline/bin/arm64/gatord \
  --lwi-gles-layer-lib-path ../target/android/arm/unrooted/arm64-v8a/libGLES_layer_lwi.so
```

   Tip

   To simplify command entry, copy the following files from the Arm Mobile Studio installation directory to a working directory:

   - <install_directory>/performance_advisor/lwi/helpers/lwi_me.py
   - <install_directory>/performance_advisor/lwi/helpers/gator_me.py
   - <install_directory>/streamline/bin/arm64/gatord
   - <install_directory>/performance_advisor/lwi/target/android/arm/unrooted/arm64-v8a/libGLES_layer_lwi.so

   Note that the lwi_me.py script requires that the accompanying gator_me.py script is in the same directory, so ensure you copy both files.

3. The script returns a numbered list of the Android package names for the debuggable applications that are installed on your device. Enter the number of the package you want to profile.

   The script identifies the GPU in the device, installs the daemon application, and waits for you to complete the capture in Streamline. Leave the terminal window open, as you must come back to it later to terminate the script.
4. Launch Streamline:

- On Windows, from the **Start** menu, navigate to **Arm MS 2020.2** and select **Arm MS Streamline 2020.2**.
- On macOS, go to the `<install_directory>/streamline` folder, and double-click the `Streamline.app` file.
- On Linux, go to the `<install_directory>/streamline` folder, and run the `Streamline` file:

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

Note

To launch Streamline with an Arm Mobile Studio professional license, you must open this file from within a Terminal shell that has the correct licensing environment variables set. For example:

```
cd /streamline/
open Streamline.app
```

Refer to *Adding a professional license* for instructions.

5. In the **Start** view, select your device from the list of detected targets.

Next Steps

Choose a counter template. For more information about how to find and select a counter template, see *3.2 Choose a counter template* on page 3-27.

3.2 Choose a counter template

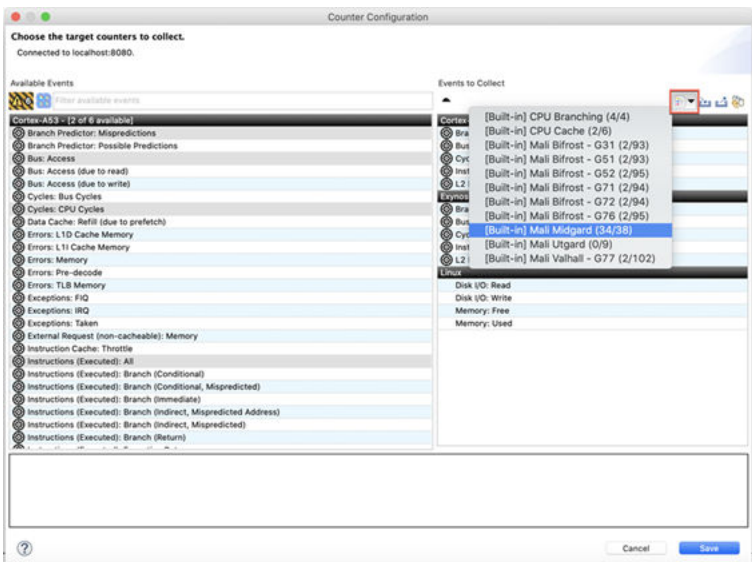
Counter templates are pre-defined sets of counters that enable you to review the performance of both CPU and GPU behavior. Choose the most appropriate template for the GPU in your target device.

Prerequisites

Follow the instructions detailed in *3.1 Connect Streamline to your device* on page 3-25 before you choose your counter template.

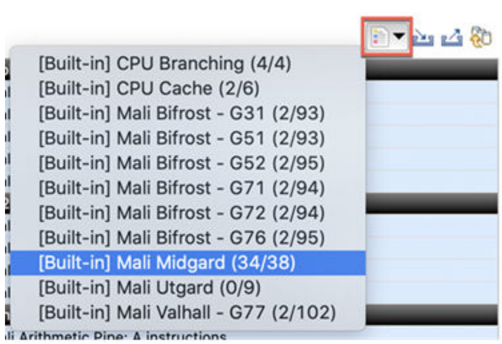
Procedure

1. In the **Start** view, click **Configure Counters**.
2. Click **Add counters from a template** to see a list of available templates.



3. Select a counter template appropriate for the GPU in your target device, then **Save** your changes.

   The number of counters in the template that your target device supports is shown next to each template. Choose the template with the highest number of supported counters. For example, here, 34 of the 38 available counters in the Mali Midgard template are supported in the connected device.



4. Optionally, in the **Start** view, click **Advanced Settings** to set more capture options, including the sample rate and the capture duration (by default unlimited). Refer to *Set capture options* in the *Arm Streamline User Guide*.

Next Steps

Capture a profile using Streamline. For more information about how to capture the behavior of your CPU and GPU performance using Streamline, see *3.3 Capture a Streamline profile* on page 3-28.

3.3 Capture a Streamline profile

Start a capture session to profile data from your application in real time. When the capture session ends, Streamline automatically opens a report for you to analyze later.

Prerequisites

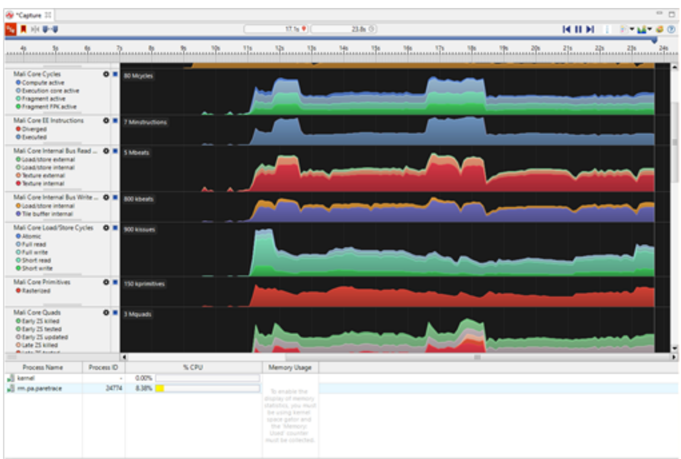
Before you capture a profile in Streamline, you must *3.1 Connect Streamline to your device* on page 3-25 and *3.2 Choose a counter template* on page 3-27.

Procedure

1. In the **Start** view, click **Start Capture** to start capturing data from the target device.

   Specify the name and location on the host for the capture file that Streamline creates when the capture is complete. Streamline then switches to **Live** view and waits for you to start the application on the device.
2. Start the application that you want to profile.

   The **Live** view shows charts for each counter that you selected. Below the charts is a list of running processes in your application with their CPU usage. The charts now start updating in real time to show the data that `gatord` captures from your running application.

   
3. Unless you specified a capture duration, in the **Capture Control** view, click **Stop capture and analyze**  to end the capture.

   Streamline stores the capture file in the location that you specified previously, and then prepares the capture for analysis. When complete, the capture appears in the **Timeline** view.
4. IMPORTANT: Switch back to the terminal running the `lwi_me.py` script and press any key to terminate it. The script kills all processes that it started and removes `gatord` from the target.

Next Steps

- *3.4 Generate a performance report* on page 3-29
- To analyze performance with Streamline, see *Analyze your capture* in the *Arm Streamline User Guide*.

3.4 Generate a performance report

Generate an HTML performance report from an existing Streamline capture.

Prerequisites

To generate a report, you must first *3.1 Connect Streamline to your device* on page 3-25, *3.2 Choose a counter template* on page 3-27, and *3.3 Capture a Streamline profile* on page 3-28.

Procedure

1. Open a terminal in the directory containing your APC file.

   Note

   The APC file can be a zip file or an uncompressed `.apc` directory.

2. Run Performance Advisor using the following command:

   ```
   pa <filename>.apc [options]
   ```

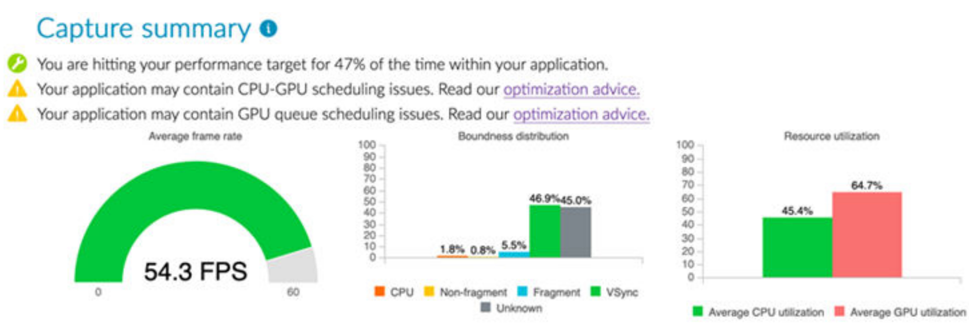
   To control how the `pa` command runs, you can pass various options to it. See *B.1 The pa command* on page Appx-B-51 for detailed descriptions of all the available options. You can also add multiple command-line options to a file that you pass to the `pa` command, see *B.1.1 pa command-line options file* on page Appx-B-53 for details.

   Note

   - For example, to include build and device information in the report summary, include the `--build-name`, `--build-timestamp`, and `--device-name` command-line options.
   - To show any CPU and GPU scheduling issues with your application, include the `--main-thread` option and specify the thread that you want to analyze:

     ```
     --main-thread=<thread-name>
     ```

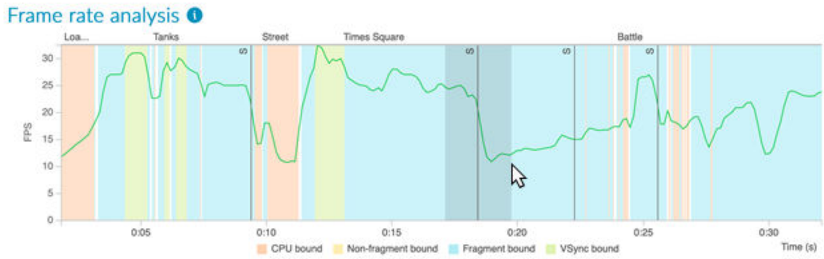
     If any scheduling issues are detected, Performance Advisor shows an indicator at the top of the report.



   - To check whether your application exceeds certain threshold values, include options for setting a per-frame budget.

Performance Advisor saves an HTML file to the current directory. Alternatively, you can specify a different directory using the `--directory` option. The file contains the results of the performance analysis, and links to advice on how to improve the performance.

The summary section shown at the top of the report is based on the duration of your capture. To take a closer look at a specific area of interest, click and drag the cursor over the region to select it.



Click anywhere on the chart when you are ready to go back to the original capture duration.

You can zoom in to any line chart in the report in the same way, by clicking and dragging over the area of interest. When you zoom in on one chart, all other charts in the same section zoom in to the same point so you can easily compare them.

If you set any per-frame budgets, a solid line appears on the relevant charts so you can check whether your application remains below it.

To get help on overcoming graphics problems and optimizing your application, click the *advice links* on the report.

Related tasks

4.2 Export performance data as a JSON file on page 4-34

4.3 Generate multiple report types on page 4-37

Related references

B.1 The pa command on page Appx-B-51

Related information

Optimization advice

Chapter 4

Running Performance Advisor in continuous integration workflows

Regular performance reports enable you to get instant feedback throughout your development cycle. With an Arm Mobile Studio Professional license, you can integrate Performance Advisor into your continuous integration workflow. This workflow enables you to automatically generate daily reports that help your team monitor how changes during the development cycle impact performance. Also, you can automatically generate machine-readable JSON reports that you can import into your existing performance regression tracking systems.

It contains the following sections:

4.1 Generate performance reports automatically

If your development team uses a CI (continuous integration) system to merge daily code changes, you can run nightly automated on-device performance testing across multiple devices.

Note

CI functionality is only available with *Arm Mobile Studio Professional Edition*.

Use a CI tool such as Jenkins, TeamCity, or Buildbot to send the following instructions to the host machines for each device in your device farm.

Prerequisites

Generate a `configuration.xml` file by *connecting Streamline to your device* on page 3-25, *choosing your counter configuration or counter template* on page 3-27, and then *exporting a configuration file*.

Procedure

1. Change to the `<install_directory>/performance_advisor/lwi/helpers` directory, or copy the following files to your working directory:
   - `<install_directory>/performance_advisor/lwi/helpers/lwi_me.py`
   - `<install_directory>/performance_advisor/lwi/helpers/gator_me.py`
   - `<install_directory>/streamline/bin/arm64/gatord`
   - `<install_directory>/performance_advisor/lwi/target/android/arm/unrooted/arm64-v8a/libGLES_layer_lwi.so`
   - `configuration.xml`
2. Run the `lwi_me.py` script with the `--headless` option, and specify the path to the configuration file:

   ```
   python3 lwi_me.py --package <app.package.name> \
     --headless <path_to_directory>/<filename>.apc \
     --daemon <install_directory>/streamline/bin/arm64/gatord \
     --config <path_to_config_file>/configuration.xml
   ```

   For Android 10, add one of the following options:

   - `--lwi-gles-layer-lib-path <path_to_GLES_layer_lib>`
   - `--lwi-vk-layer-lib-path <path_to_Vulkan_layer_lib>`

   Add any other optional arguments you require, refer to *B.2 The lwi_me.py script options* on page Appx-B-54 for details.

   Note

   If you built your application with Unity, include the Unity player activity in `<app.package.name>`, for example:

   ```
   com.arm.mygame/com.unity3d.player.UnityPlayerActivity
   ```

3. Add a wait period of at least one minute, to allow the script to prepare the device for profiling.
4. Start the application on the target device. For example:

   ```
   adb shell am start -n <app.package.name>
   ```

5. To stop profiling, exit the application in one of the following ways:
   - Set your application test case to exit after a certain length of time.
   - Forcefully kill the application using:

     ```
     adb shell am force-stop <app.package.name>
     ```

The Streamline capture file is saved to the location you specified with the `--headless` command-line option.

Note

Instead of exiting the application, you can specify a `--headless-timeout <seconds>` value. This method is not ideal for test scenarios with variable performance.

---

6. Generate Performance Advisor reports in HTML and JSON formats:

```
pa <capture_filename.apc> -p <app.package.name> -d <output_directory> -t
html:<file_name>.html,json:<file_name>.json
```

For the full list of available command-line options, refer to *B.1 The pa command* on page Appx-B-51.

Next Steps

Push the HTML reports to a centrally visible location for your team to analyze each day. Push the JSON reports to any JSON-compatible database and visualization tool, such as *ELK Stack*.

For more information, refer to *Integrate Arm Mobile Studio into a CI workflow* on the Arm Developer website.

4.2 Export performance data as a JSON file

Generate a JSON report that you can import into other tools. Use reports from multiple test runs to track performance over time.

Note

JSON reports are only available with *Arm Mobile Studio Professional Edition*.

JSON reports provide a raw data export that you can import into other tools, such as a NoSQL database, to compare different test runs. For example, you can track the average number of visible primitives per frame between builds.

Procedure

1. Open a terminal in the directory containing your APC file.

   Note

   The APC file can be a Streamline archive (.zip) or an uncompressed `.apc` directory.

2. Run Performance Advisor using the following command:

   ```
   pa <capture.apc.zip> -p <app.package.name> -d <optional output dir> -t json
   ```

   To change the output file name, append it to the `-t` argument using a colon:

   ```
   -t json:your_file_name.json
   ```

The JSON report output is packed by default, to make it compatible with most third-party database and visualization tools. If you want to view the data in a more human-readable format, use the `--pretty-print` option.

The following example shows part of a JSON report that was output with the `--pretty-print` option:

```
{
    "deviceInfo": {
        "build": null,
        "device": "Example board",
        "processors": "Cortex-A55 MP4, Mali-G72"
    },
    "allCapture": {
        "averageFrameRateFps": 19.4,
        "boundnessSplitPercentage": {
            "fragment": 0.0,
            "non-fragment": 0.0,
            "vsync": 0.0,
            "cpu": 98.5,
            "unknown": 1.5
        },
        "averageUtilizationPercentage": {
            "averageGpuUtilization": 19.0,
            "averageCpuUtilization": 62.7
        }
    },
    "fpsBoundness": {
        "frameRate": {
            "average": 19.4,
            "max": 21.1,
            "min": 17.9,
            "centiles": {
                "80": 20.0,
                "98": 21.1,
                "95": 20.7
            }
        },
        "vsync": {
            "target": 60,
            "percentageTimeUnderTarget": 100
        }
    },
    "overdrawPerPixel": {
```

```
        "overdraw": {
            "average": 0.3,
            "max": 0.4,
            "min": 0.1,
            "centiles": {
                "80": 0.4,
                "98": 0.4,
                "95": 0.4
            }
        }
    },
    "gpuUsagePerFrame": {
        "nonfragmentCycles": {
            "average": 1707767.6,
            "max": 2039630.8,
            "min": 770117.5,
            "centiles": {
                "80": 1917112.6,
                "98": 2039630.8,
                "95": 2039630.8
            }
        },
        "gpuCycles": {
            "average": 4157114.0,
            "max": 4897026.6,
            "min": 1587167.6,
            "centiles": {
                "80": 4649032.8,
                "98": 4897026.6,
                "95": 4897026.6
            }
        },
        "fragmentCycles": {
            "average": 2449346.8,
            "max": 2911080.0,
            "min": 608306.8,
            "centiles": {
                "80": 2857394.4,
                "98": 2911080.0,
                "95": 2911080.0
            }
        }
    },
    "drawCallsPerFrame": {
        "drawCalls": {
            "average": 456.0,
            "max": 456.0,
            "min": 456.0,
            "centiles": {
                "80": 456.0,
                "98": 456.0,
                "95": 456.0
            }
        }
    },
    "primitivesPerFrame": {
        "totalPrimitives": {
            "average": 290318.2,
            "max": 331233.8,
            "min": 114309.3,
            "centiles": {
                "80": 325304.5,
                "98": 331233.8,
                "95": 331233.8
            }
        },
        "visiblePrimitives": {
            "average": 89856.7,
            "max": 102210.2,
            "min": 34685.2,
            "centiles": {
                "80": 100151.9,
                "98": 102210.2,
                "95": 102210.2
            }
        }
    },
    "pixelsPerFrame": {
        "pixels": {
            "average": 4669783.4,
            "max": 5315129.7,
            "min": 3197000.8,
            "centiles": {
                "80": 5165539.5,
```

```
                "98": 5315129.7,
                "95": 5315129.7
            }
        }
    },
…
```

Related tasks

3.4 Generate a performance report on page 3-29

4.3 Generate multiple report types on page 4-37

Related references

B.1 The pa command on page Appx-B-51

4.3 Generate multiple report types

Generate an HTML performance report and a JSON performance report from an existing Streamline capture.

Prerequisites

Before you can generate a report, you must have a Streamline capture file. For help on creating a capture, see *3.3 Capture a Streamline profile* on page 3-28.

Procedure

1. Open a terminal in the directory containing your APC file.

   Note

   The APC file can be a zip file or an uncompressed `.apc` directory.

2. Run Performance Advisor using the following command:

   ```
   pa <capture.apc.zip> -p <app.package.name> -d <optional output dir> -t html,json
   ```

   To change the output file names, append each file name to the corresponding type argument using a colon:

   ```
   -t html:your_file_name.html,json:your_file_name.json
   ```

Related tasks

3.4 Generate a performance report on page 3-29

4.2 Export performance data as a JSON file on page 4-34

Related references

B.1 The pa command on page Appx-B-51

Chapter 5
Capturing a slow frame

Identify slow frames by using the lightweight interceptor (LWI) in different modes. Before you can use the LWI, you must first integrate it with your application.

It contains the following sections:

5.1 Capturing slow frame rate images

Use Performance Advisor to continuously monitor frame rate and trigger frame captures when a slow part is detected.

Arm provides the helper script `lwi_me.py` to enable you to capture data from your device using the lightweight interceptor. This script is located in <install_directory>/performance_advisor/lwi/helpers.

Procedure

1. Create an empty directory for the frame capture images.

   Note

   Frame captures might not have completed writing at the point of application exit, which can lead to incomplete frame captures. Performance Advisor ignores these incomplete frame captures, and only shows complete frame captures in the report.

2. In a terminal, navigate to <install_directory>/performance_advisor/lwi/helpers, where the Python script `lwi_me.py` is located.
3. Run the `lwi_me.py` script with the options you need for your frame capture.

   The script configures your device so that Performance Advisor can collect data from it. Specify the directory you created in step 1 so the frame capture images are saved there.

   Note

   This directory must be empty.

   For example, to capture a frame when the frame rate goes below 30fps, and allow at least 100 frames between captures:

```
python3 lwi_me.py --daemon <path_to_gatord> --lwi-fps-threshold 30 \
  --lwi-frame-gap 100 --lwi-mode capture \
  --lwi-out-dir <path_to_frame_captures_directory> \
  [--lwi-gles-layer-lib-path <path_to_GLES_layer_lib>]
```

   The `gatord` binaries are in the Arm Mobile Studio installation directory:

   - For 32-bit applications, set `--daemon` to <install_directory>/streamline/bin/arm/gatord
   - For 64-bit applications, set `--daemon` to <install_directory>/streamline/bin/arm64/gatord

   The `--lwi-gles-layer-lib-path` option is required for Android 10 devices to specify the path to the OpenGL ES layer library file. This file is supplied in the Arm Mobile Studio installation directory. For example:

```
--lwi-gles-layer-lib-path \
  ../target/android/arm/unrooted/arm64-v8a/libGLES_layer_lwi.so
```

   See *B.2 The lwi_me.py script options* on page Appx-B-54 for details of all the available command-line options.

   Note

   Capturing frames can affect performance. If you notice a decrease in performance when capturing images, tag the slow frames instead. See *5.2 Tagging slow frames* on page 5-41 for more information.

4. If there are multiple debuggable packages on your device, the script lists them. Enter the number of the package you want to analyze and follow the instructions to take a Streamline capture, as described in *3.3 Capture a Streamline profile* on page 3-28.

   You do not need to run the `gator_me` script as it is called by the `lwi_me` script.

   Important

   When Streamline prompts you to save the capture file, do not save it to the frame captures directory that you specified in step 1. The contents of this directory are replaced when the frame capture images are written there.

   Note

   During the capture, images are saved to the output directory every time the FPS drops below the target threshold. By default, images stop being captured after the first 500 frames. You can adjust the end frame by specifying a different number using the `-lwi-frame-end` option when running the `lwi_me.py` script. You can also specify the frame number at which to start the capture, using the `-lwi-frame-start` option.

5. Use the `pa` command to generate an HTML report, specifying the location where you saved the frame capture images in step 1. Optionally specify a directory in which to save the HTML report, otherwise the HTML report is saved to the current directory.

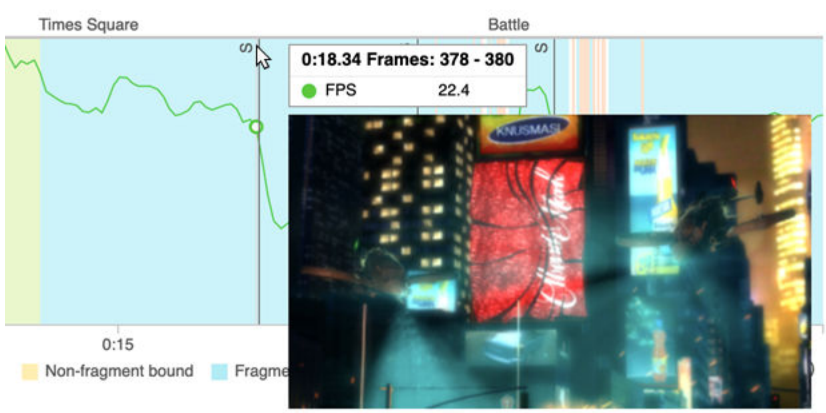
```
pa <my_capture.apc> --frame-capture=<path_to_frame_captures_directory> \
  [--directory=<path_to_output_directory>]
```

   You can use other options to specify metadata for your report, such as the build name, device name, and application name. See *B.1 The pa command* on page Appx-B-51 for all the available command-line options.

   For more information about generating an HTML report, see *4.3 Generate multiple report types* on page 4-37.

6. Open the HTML report in a browser.

   To see the captured frame, hover the cursor over the screen capture icon .



5.2 Tagging slow frames

If capturing frames directly impacts the performance of your application by reducing the frame rate, run the `lwi_me.py` command to capture the frame numbers in tag mode. Then run the `lwi_me.py` command to capture the frames in replay mode.

Procedure

1. Trace your application and output the capture to a specified folder.

   For example, use the following command to trace an OpenGL ES application, tagging a frame when the frame rate goes below 50fps:

   ```
   python3 lwi_me.py --daemon <path to gator> --package <app.package.name> \
     --lwi-fps-threshold 50 --lwi-mode tag --lwi-out-dir /some/folder
   ```

   Run the file with tagged frame numbers using `--lwi-mode replay` to capture the tagged frames.

   ```
   python3 lwi_me.py --daemon <path to gator> --package <app.package.name> \
     --lwi-fps-threshold 50 --lwi-mode replay --lwi-slow-frames /some/folder/slow-frames \
     --lwi-out-dir /some/folder
   ```

2. Manually capture a Streamline profile, as described in *3.3 Capture a Streamline profile* on page 3-28.

   Note

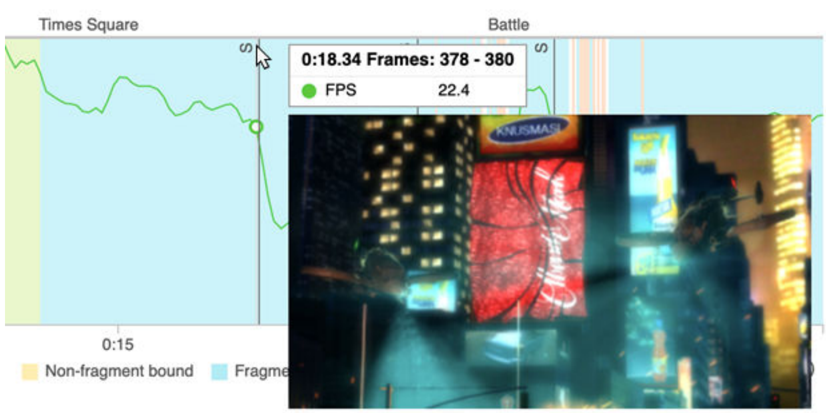
   During the Streamline capture, the captured resources are written in the target when the trace reaches the end frame. The default is to end the capture at frame 500. You can adjust the end frame by specifying an alternative value for the `FRAMEEND` parameter of the lwi_me.py script.

3. To export the capture to the HTML report, send the frame capture path to the output directory:

   ```
   pa [capture.apc] --package <app.package.name> --frame-capture=path [frame_capture_folder]
   ```

   For more information about generating an HTML report, see *4.3 Generate multiple report types* on page 4-37.

   To see the captured frame, hover the cursor over the screen capture icon.



Chapter 6
Adding semantic input to the reports

Performance Advisor can use semantic information that the application provides as key input data when generating the analysis reports.

The analysis reports support the use of region annotations to give context to the different frame ranges in a test scenario. Manually add these annotations into the application code. Alternatively, if manually adding annotations is not possible, or for quick debugging and extra analysis, specify a CSV file containing the regions. Give Performance Advisor the path to the CSV file using the `--regions` argument.

It contains the following sections:

6.1 Manually create annotations from your application

Streamline allows an application to manually create annotations, which can be used as an alternative source of frame boundary annotations.

Note

If you are hand-coding Performance Advisor annotations, use the `gator_me.py` script instead of `lwi_me.py` script. The script is located in the Arm Mobile Studio installation directory: `<install_directory>/performance_advisor/lwi/helpers/`

Native code

The native C code to include for generating annotations is located in the `<ms_install>/streamline/gator/annotate/` directory.

Unity plug-in code

Source code for a proof-of-concept plug-in for Unity 2018.2, which provides C# bindings for the Streamline annotation functions, is located here:

[GitHub] ARM-software Mobile Studio Unity Plugin

6.1.1 Generating frame boundary annotations

The Graphics Analyzer interceptor automatically adds frame boundary annotations. However, for more flexibility over how the frames are defined, you can manually specify the annotations. For example, if you only want to track certain frames or if you have multiple contexts.

To implement frame boundary annotations manually, generate Streamline marker annotations matching the following regular expression format, where the number is a monotonically incrementing frame number:

```
F(/d+)
```

For example:

```
F10
F11
F12
```

6.1.2 Generating region annotations

Region annotations enable you to define consecutive frames as a named group to help you see which part of your application is running when you review its performance. Performance Advisor produces a separate section in the report for each defined region so that you can analyze the metrics for each region in more detail.

To define a region in your code, manually generate Streamline marker annotations:

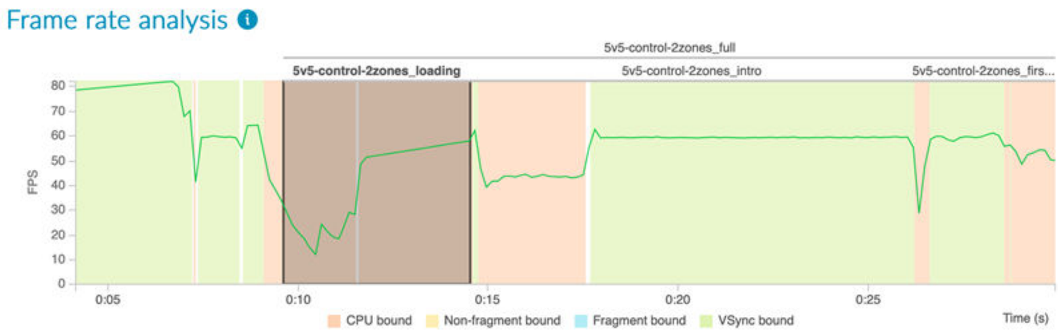
```
Region Start <region name>
Region End <region name>
```

Note

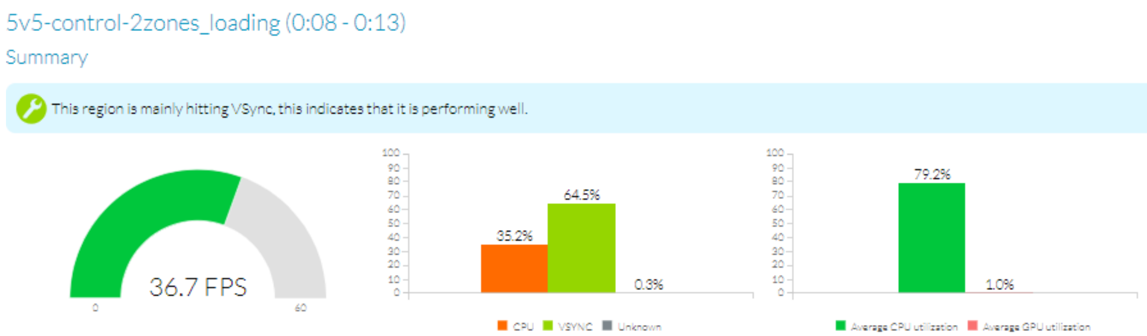
To prevent an error, ensure that region names are unique by appending a number to the end of the region name.

To ensure that conclusions are statistically significant, Arm recommends keeping regions and subregions relatively large. For example, at least two seconds in length.

The regions defined in your code are shown on the **FPS analysis** chart. Any regions that overlap are shown as subregions on the chart.



Click a region to zoom in and analyze it further on the **FPS analysis** chart, or scroll down the report to see metrics for each region.



6.2 Specify a CSV file containing the regions

If manually adding annotations is not possible, or for quick debugging and extra analysis, specify a CSV file containing the regions and use the `--regions` argument.

Create a CSV file using the following format, where each region is on a new line:

```
Region Name,Start,End
```

`Start` and `End` are a timestamp in milliseconds or a frame number followed by `f`.

For example, specify a region that starts at 500ms and ends at 15000ms with:

```
Test Region,500,15000
```

Specify a region that starts at the 500th frame and ends at the 15000th frame with:

```
Test Region,500f,15000f
```

To set the start to the start of the capture, or the end to the end of the capture, use a `*`. For example:

```
Test Region,*,15000
```

```
Test Region,5000f,*
```

Note

Performance Advisor ignores the region if you use `*` for both the start and the end, as this region is the whole capture.

Give Performance Advisor the path to the CSV file using the `--regions` argument.

6.3 Clip unwanted data from the capture

Specify the part of the capture that you want to include in the analysis report and discard the remaining data. For example, remove the loading and ending screens so they are not included in the report.

You can specify the start and end time with one of the following:

- A timestamp in milliseconds.
- A region name with `:start` or `:end` appended to it.

Procedure

1. Specify the start of the report with `--clip-start=<clipStartStr>`.

   If you do not specify a start, the report starts from the beginning of the capture.
2. Specify the end of the report with `--clip-end=<clipEndStr>`.

   If you do not specify an end, the report ends at the end of the capture.

Example 6-1 Clip sections of a capture

- Clip the capture so the report starts at two seconds and ends at 15 seconds:

```
--clip-start=2000 --clip-end=15000
```

- Clip the capture so the report starts at the end of the region named "loading screen":

```
--clip-start="loading screen:end"
```

- Clip the capture so the report starts at the end of the region "level one loading screen" and ends at the start of the region "level two loading screen":

```
--clip-start="level one loading screen:end" --clip-end="level two loading screen:start"
```

Related references

B.1 The pa command on page Appx-B-51

Appendix A
Analytics

It contains the following sections:

A.1 Data collection in Performance Advisor

Arm periodically collects anonymous information about the usage of our products to understand, and analyze, what components or features you are using. We use this information to improve our products and your experience with them.

Product usage analytics contain information such as system information, settings, and usage of specific features of the product. You can enable or disable the feature in the product settings.

The data that we collect through Performance Advisor is anonymous and does not include any personal information.

Host information includes:

- Operating system details, such as version number, platform, language, and architecture (for example 64-bit).
- CPU and GPU information.
- Java version and memory.
- Number of monitors.
- Screen pixels per inch (PPI) and monitor resolutions.

Product information includes:

- Version and build number of Performance Advisor.
- Edition of license that you are using (Starter, Evaluation, or Professional).
- Session time using Performance Advisor.

Feature information includes:
- Number, and type (HTML or JSON), of reports generated.
- Number of headless captures used.
- Number of captures containing user regions.
- Number of captures containing overdraw, or draw call information.
- Number of times screenshots were supplied.
- Total number of errors that you encountered, reported by type of error.
- Total number of licensing errors (unsupported/unknown license).
- Use of the `--mspf` option to display milliseconds per frame in the report.

A.2 Disable analytics data collection

Analytics collection is enabled by default. Use these options to disable the collection of analytics data in Performance Advisor.

- Set the command-line argument `--disable-analytics` when running Performance Advisor to disable it for the current invocation.
- Alternatively, set the `ARM_DISABLE_ANALYTICS` environment variable to any nonzero value before running Performance Advisor to disable analytics collection for all invocations.

Appendix B
Command-line options

It contains the following sections:

B.1 The pa command

The `pa` command runs Performance Advisor on a capture.

Syntax

```
pa [OPTIONS] <capture.apc>
```

Note

You can pass options to `pa` in a configuration file. See *B.1.1 pa command-line options file* on page Appx-B-53 for details.

Options

`<capture.apc>`
The path to the capture APC directory or zip file.

`--centiles=int[,int...]`
Comma-separated integer values specifying the percentiles to calculate for each data series. Default = 80,90,95.

`--clip-end=clipEndStr`
Specify the time that you want the report to end at. `clipEndStr` is the timestamp in milliseconds or the frame number followed by `f`. For example, `--clip-end=7000` ends the clip at 7000ms, or `--clip-end=7000f` ends the clip at the 7000th frame. Alternatively you can use the format `<region-name>:start` or `<region-name>:end` to use the start or end time of a region.

`--clip-start=clipStartStr`
Specify the time that you want the report to start from. `clipStartStr` is the timestamp in milliseconds or the frame number followed by `f`. For example, `--clip-start=500` starts the clip at 500ms, or `--clip-start=500f` starts the clip at the 500th frame. Alternatively you can use the format `<region-name>:start` or `<region-name>:end` to use the start or end time of a region.

`-d, --directory=path`
The output directory path for the reports.

`--disable-analytics`
Disable sending any analytics data to Arm.

`-f, --frame-capture=path`
The path to the frame captures directory.

`-h, --help`
Show command-line arguments and descriptions, and exit.

`-m, --main-thread=string`
The name of the main render thread to analyze.

`--mspf`
Display milliseconds per frame throughout the HTML report instead of FPS.

`--pretty-print`
Print the JSON output with whitespace, making it human readable.

`-p, --process=string`
The name of the process to inspect.

`--[no-]progress`
Whether to display progress bars or not.

-r, --regions=file
Takes a CSV file containing custom regions to add to the report, where each line of the CSV file is of the format `regionName,start,end`. `start` and `end` are a timestamp in milliseconds or a frame number followed by `f`. For example, `regionName,500,7000` starts the region at 500ms and ends it at 7000ms. `regionName,500f,7000f` starts the region at the 500th frame and ends it at the 7000th frame. See *6.2 Specify a CSV file containing the regions* on page 6-45.

-t, --type=type[:file][,type[:file]...]
A comma-separated list of report types, where the type is one of:

json
JSON CI report

html
Interactive html report

You can specify an output filename for each report.

--target-fps=int
The target frame rate in frames per second. Default = 60.

-V, --version
Print version information and exit.

Options for report metadata:

--application-name=string
The human readable name of the application being analyzed. For example, "Awesome Game". If the name contains whitespace, use quotes. This name becomes the report title. Default = "Performance Advisor Report".

--build-name=string
The build name of your application. For example, `nightly. fa34c92`.

--build-timestamp=string
The timestamp of your application build. For example, `Thu, 22 Aug 2019 12:47:30`.

--device-name=string
The name of the device that is used to obtain the capture.

Options for setting a per-frame budget:

--bandwidth-budget=<value>
Threshold for read/write bytes.

--cpu-cycles-budget=<value>
Threshold for CPU cycles.

--draw-calls-budget=<value>
Threshold for draw calls.

--gpu-cycles-budget=<value>
Threshold for GPU cycles.

--overdraw-budget=<value>
Threshold for overdraw.

--pixels-budget=<value>
Threshold for pixels.

--primitives-budget=<value>
Threshold for primitives.

--shader-cycles-budget=<value>
Threshold for shader cycles.

--vertices-budget=<value>
Threshold for vertices.

This section contains the following subsection:
- *B.1.1 pa command-line options file* on page Appx-B-53.

B.1.1 pa command-line options file

You can list command-line options in a file that you pass to the `pa` command.

For example, you might create a file for your budget thresholds called `budget` that contains the following options:

```
--cpu-cycles-budget 100000000
--gpu-cycles-budget 28000000
--shader-cycles-budget 20000000
--draw-calls-budget 350
--vertices-budget 1000000
```

Note

Use a space between an option and its value in the file, not an equals sign.

When you run Performance Advisor, specify the file with `@<filename>`, for example:

```
pa --type=html:output.html capture.apc @budget
```

B.2 The lwi_me.py script options

To see the possible options and their default values for the `lwi_me.py` command, run `python3 lwi_me.py -h`.

Syntax

```
python3 lwi_me.py --daemon <path_to_gatord> [OPTIONS]
```

Options

`--device` or `-E`
The target device name. If not specified, the script automatically detects the name from the device.

`--package` or `-P`
The application package name. If not specified, the script returns a list of debuggable packages on the device, and prompts you to choose one.

`--headless` or `-H`
Perform a headless capture, and write the result to a specified `<capture_path>`. Default = perform interactive capture.

`--headless-timeout` or `-T`
Exit the headless timeout after the specified number of `<seconds>`. Default = wait for process exit.

`--config` or `-C`
Specify the `<filename>` of the configuration XML file you want to use for a headless capture. Default = None for an interactive capture, or `configuration.xml` for a headless capture.

`--daemon` or `-D`
Mandatory. Specify the `<path>` to the `gatord` binary you want to use. The `gatord` binaries are supplied in the Arm Mobile Studio installation directory:

```
<install_directory>/streamline/bin/{arm|arm64}/gatord
```

Different folders contain different versions of `gatord` for 32-bit (`arm`) or 64-bit (`arm64`) applications.

`--no-clean-start`
Disable pre-run device cleanup. Default = enabled.

`--no-clean-end`
Disable post-run device cleanup. Default = enabled.

`--overwrite`
Overwrite an earlier headless output. Default = disabled.

`--verbose or -v`
Enable verbose logging. Default = disabled.

`--lwi` on | `off` | `alone`
Enable or disable the LWI. The `alone` mode bypasses gator. Default = `on`.

`--lwi-api` `gles` | `vulkan`
Select the API you want to listen to. Default = `gles`.

`--lwi-compress-img` or `-X`
Compress screenshots. Default = no.

`--lwi-gles-layer-name <name>`
The OpenGL ES layer name. Default = `libGLES_layer_lwi.so`.

--lwi-gles-layer-lib-path <path>
The path to the OpenGL ES layer library file. This file is supplied in the Arm Mobile Studio installation directory:

```
<install_directory>/performance_advisor/lwi/target/android/arm/{rooted|
unrooted}/{arm64-v8a|armeabi-v7a}/libGLES_layer_lwi.so
```

Different files are supplied for rooted or unrooted devices, and for 32-bit (`armeabi-v7a`) or 64-bit (`arm64-v8a`) applications.

--lwi-vk-layer-name <name>
The Vulkan layer name. Default = `VK_LAYER_ARM_LWI`.

--lwi-vk-layer-lib-path <path>
The Vulkan layer library path.

--lwi-fps-window or -W
Specify the `<number_of_frames>` for the sliding window used for FPS calculation. Default = 5.

--lwi-fps-threshold or -Th
Perform a capture if the FPS goes under a specified `<fps_value>`. Default = 30.

--lwi-frame-start or -S
Start tracking from a specified `<frame_number>`. Default = 1.

--lwi-frame-end or -N
End tracking at the specified `<frame_number>`. Default = 500.

--lwi-frame-gap or -G
Minimum `<number_of_frames>` between two captures. Default = 200.

--lwi-mode or -M
Specify in which mode you want the LWI to operate:
- `none` to not capture images or tag frames. This value is the default.
- `capture` or `c` to capture frame images when the fps goes below the specified `--lwi-fps-threshold <fps_value>`. You must specify an output directory for the captured images with `--lwi-out-dir`.
- `tag` or `t` to tag frame numbers when the fps goes below the specified `--lwi-fps-threshold <fps_value>`. You must specify an output directory for the tagged frames with `--lwi-out-dir`.
- `replay` or `r` to run the file of tagged frame numbers.

--lwi-out-dir or -o
Specify the path to a directory for the captured images or tagged frames. This directory must be empty.

--lwi-slow-frames <path>
Path to a file containing the indices of slow frames (required in `replay` mode). Generate this file using the LWI in `tag` mode.