

Performance Advisor

Version 1.6

User Guide



Performance Advisor

User Guide

Copyright © 2020, 2021 Arm Limited or its affiliates. All rights reserved.

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.
0103-00	27 November 2020	Non-Confidential	New document for v1.3.
0104-00	12 March 2021	Non-Confidential	New document for v1.4.
0105-00	21 May 2021	Non-Confidential	New document for v1.5.
0106-00	26 August 2021	Non-Confidential	New document for v1.6.

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2020, 2021 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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.

Web Address

developer.arm.com

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.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

Performance Advisor User Guide

Preface

About this book	7
-----------------------	---

Chapter 1

Introduction to Performance Advisor

1.1 Overview of Performance Advisor	1-10
1.2 Performance report example	1-12
1.3 Performance Advisor workflows	1-14

Chapter 2

Before you begin

2.1 Set up your host machine	2-17
2.2 Set up your device	2-18
2.3 Integrate Performance Advisor with your application	2-19

Chapter 3

Quick start guide

3.1 Connect Streamline to your device	3-25
3.2 Choose a counter template	3-27
3.3 Capture a Streamline profile	3-28
3.4 Generate a performance report	3-29
3.5 Setting performance budgets	3-31
3.6 Generate a custom report	3-33

Chapter 4

Running Performance Advisor in continuous integration workflows

4.1 Generate performance reports automatically	4-37
4.2 Export performance data as a JSON file	4-39

	4.3	Generate multiple report types	4-42
	4.4	Generate a JSON diff report	4-43
Chapter 5		Capturing a slow frame	
	5.1	Capturing slow frame rate images	5-45
	5.2	Tagging slow frames	5-47
Chapter 6		Adding semantic input to the reports	
	6.1	Send annotations from your application code	6-49
	6.2	Specify a CSV file containing the regions	6-52
	6.3	Clip unwanted data from the capture	6-53
Appendix A		Command-line options	
	A.1	The pa command	Appx-A-55
	A.2	The lwi_me.py script options	Appx-A-58

Preface

This preface introduces the *Performance Advisor User Guide*.

It contains the following:

- [About this book on page 7.](#)

About this book

This document 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 Command-line options

This appendix explains the command-line options that are available for the `pa` command and the `lwi_me.py` script.

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_0106_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® Documentation](#).
- [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 on page 1-10.](#)
- [1.2 Performance report example on page 1-12.](#)
- [1.3 Performance Advisor workflows on page 1-14.](#)

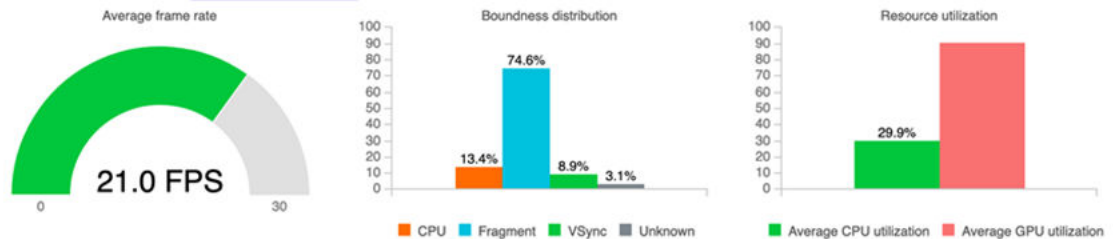
1.1 Overview of Performance Advisor

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

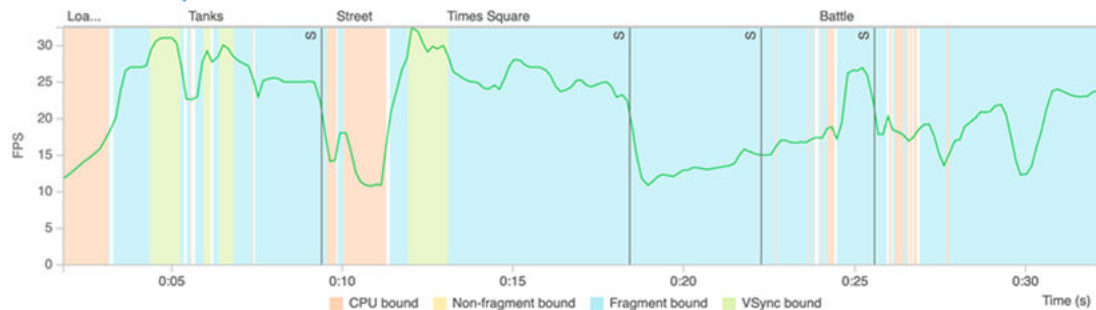
The capture summary shows whether you are achieving your target frame rate, the distribution of time spent by each processing unit, and your CPU and GPU utilization.

Capture summary

 You are hitting your performance target for 9% of the time within your application. For the frames below target you are predominantly fragment bound. Read our [optimization advice](#).



Frame rate analysis



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

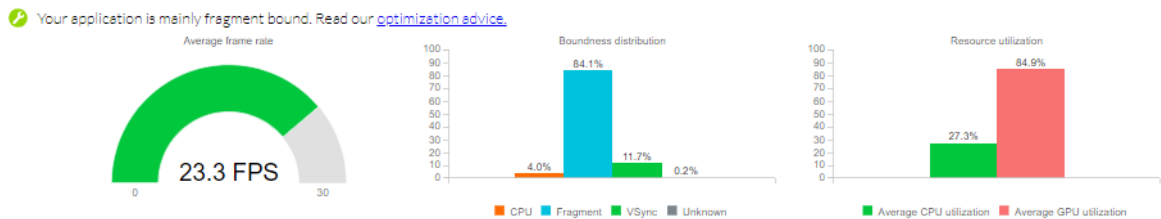
In this example, we will look at the charts in the Performance Advisor report to review the performance of an application. See how you can use the report to investigate problems with any scenes in your application that are not performing well.

We have generated a Performance Advisor report from a Streamline capture file.

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.

Capture summary

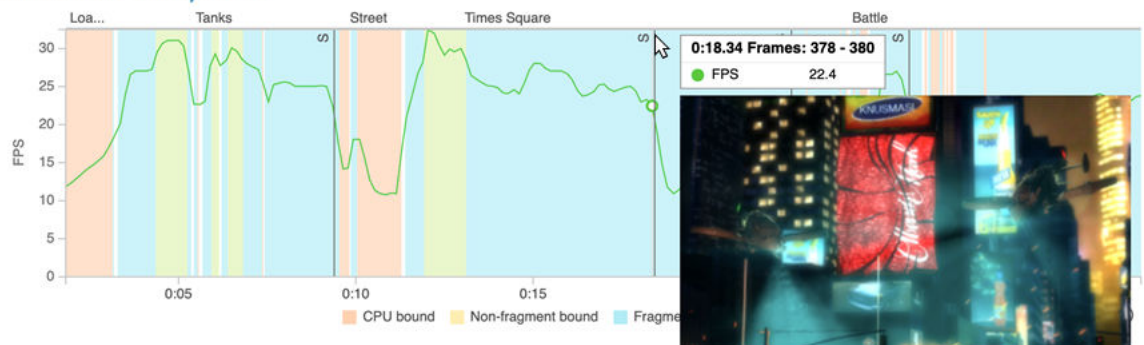


Here, we can see that the average frame rate for the capture is not achieving the configured target of 30fps. When we check the boundness distribution, we can see that the application is fragment bound. The utilization chart confirms that a graphical problem is causing this drop in frame rate.

Analyze frame rate

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

Frame rate analysis



Note

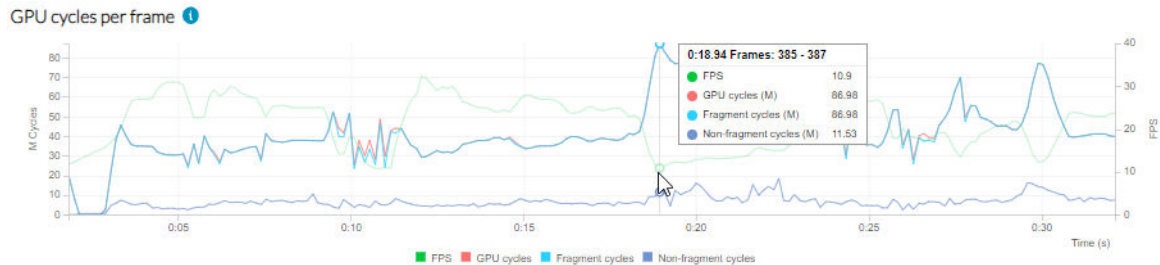
In this capture, we have used the `lwi_me.py` script to take a screenshot if the frame rate goes below 20fps. We have also specified a number of frames between captures to ensure that we do not capture too many images.

The majority 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 the application, we continue our analysis below by looking at the GPU behavior metrics.

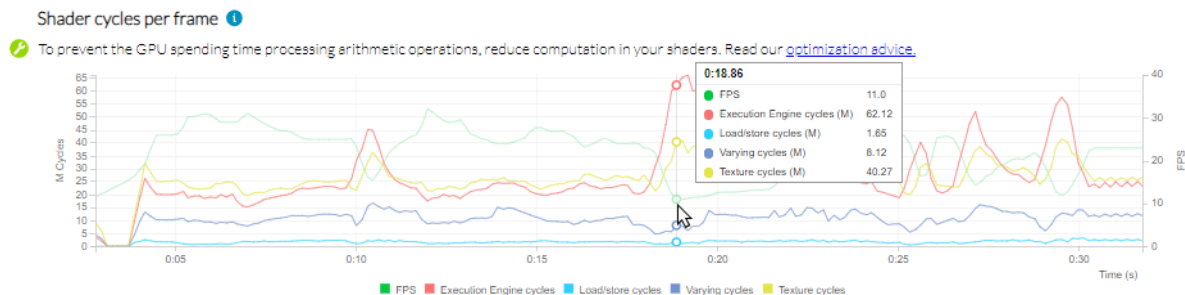
Investigate GPU behavior

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.

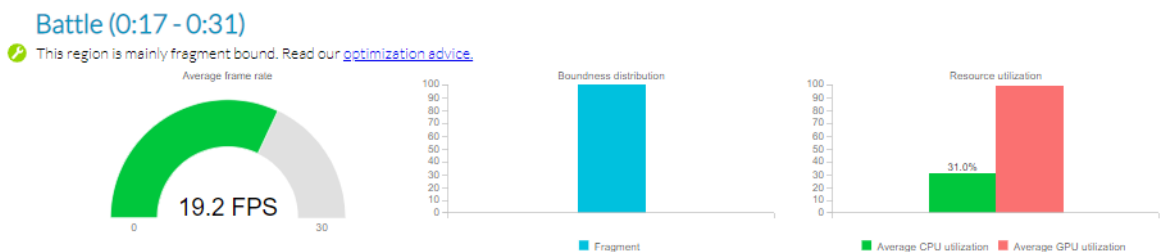


The **Shader cycles per frame** chart shows that the drop in frame rate correlates with high numbers of execution engine cycles.



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

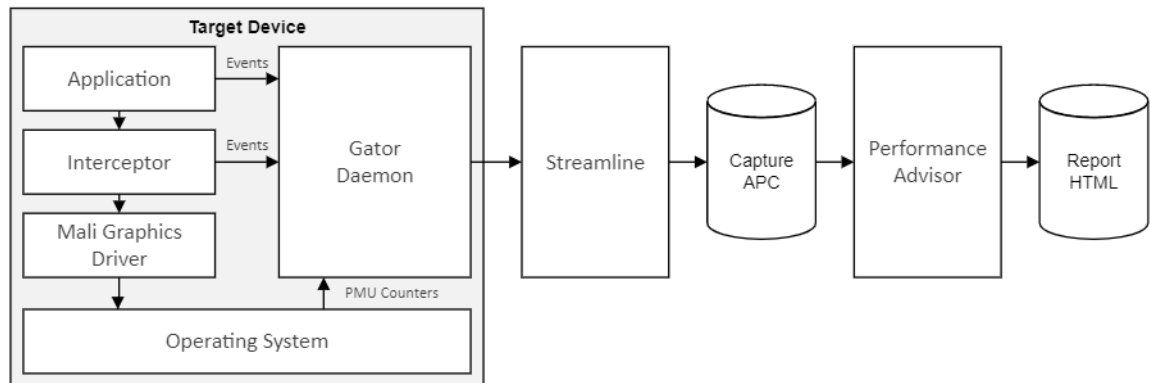
[Get started with Arm Mobile Studio](#)

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.

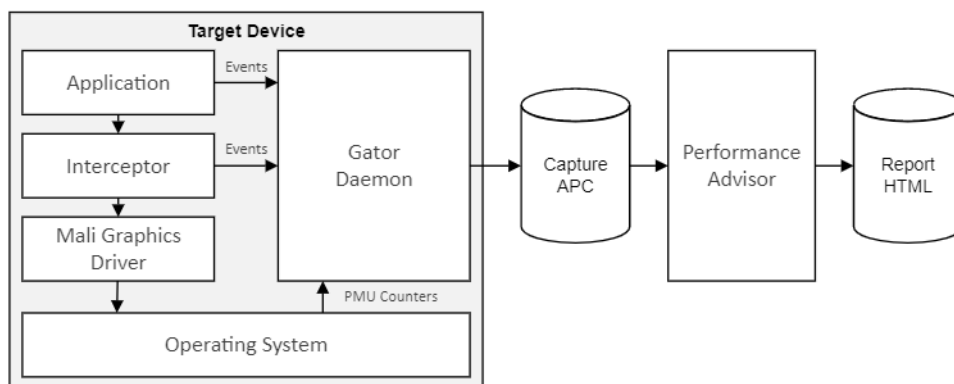


Automated capture with Performance Advisor report

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 4-36 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 on page 2-17.](#)
- [2.2 Set up your device on page 2-18.](#)
- [2.3 Integrate Performance Advisor with your application on page 2-19.](#)

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 `gator` 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 path to the Performance Advisor directory.

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
ce12345abcdef1a1234    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.

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.

OpenGL ES

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

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

Two versions of the library are provided:

- For 64-bit targets, use the library file in the **64-bit** directory.
- For 32-bit targets, use the library file in the **32-bit** 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/
```

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 in the **64-bit** directory.
- For 32-bit targets, use the library file in the **32-bit** directory.

Next steps

Continue with the appropriate instructions for your project:

- [2.3.1 Prepare your Unity project on page 2-19](#)
- [2.3.2 Prepare your Android Studio project on page 2-22](#)

2.3.1 Prepare your Unity project

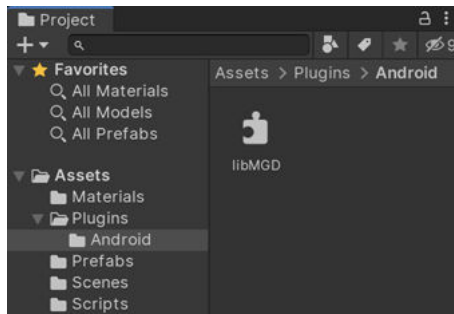
Copy the library file or Vulkan layer file into Unity, and set the necessary attributes and settings. Then build your APK and install it on your device. You are then ready to perform a capture.

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

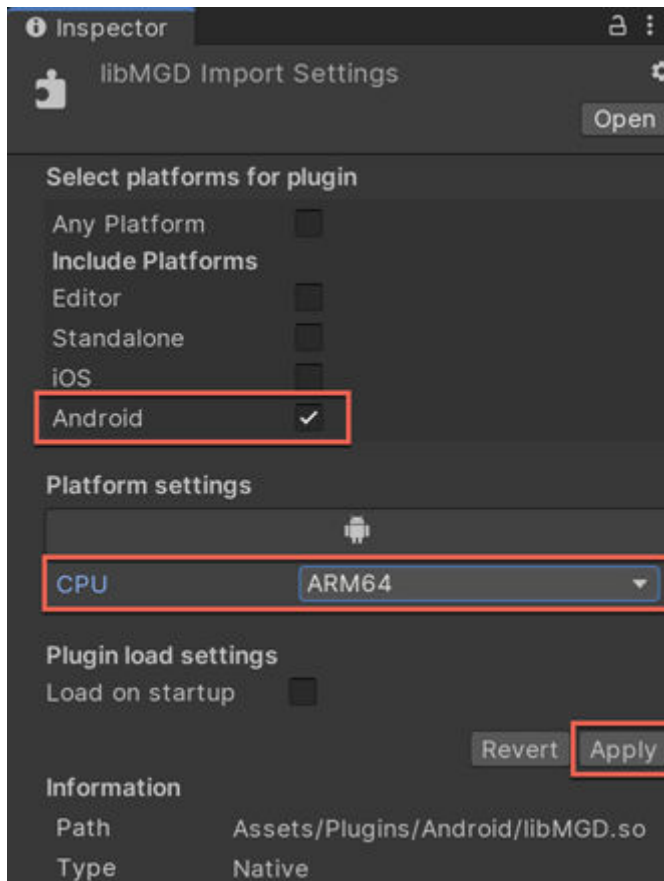
1. Copy the required libLWI.so file or Vulkan layer into the Assets/Plugins/Android/ directory in your Unity project. Create this directory if it does not exist.



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 libLWI.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**.

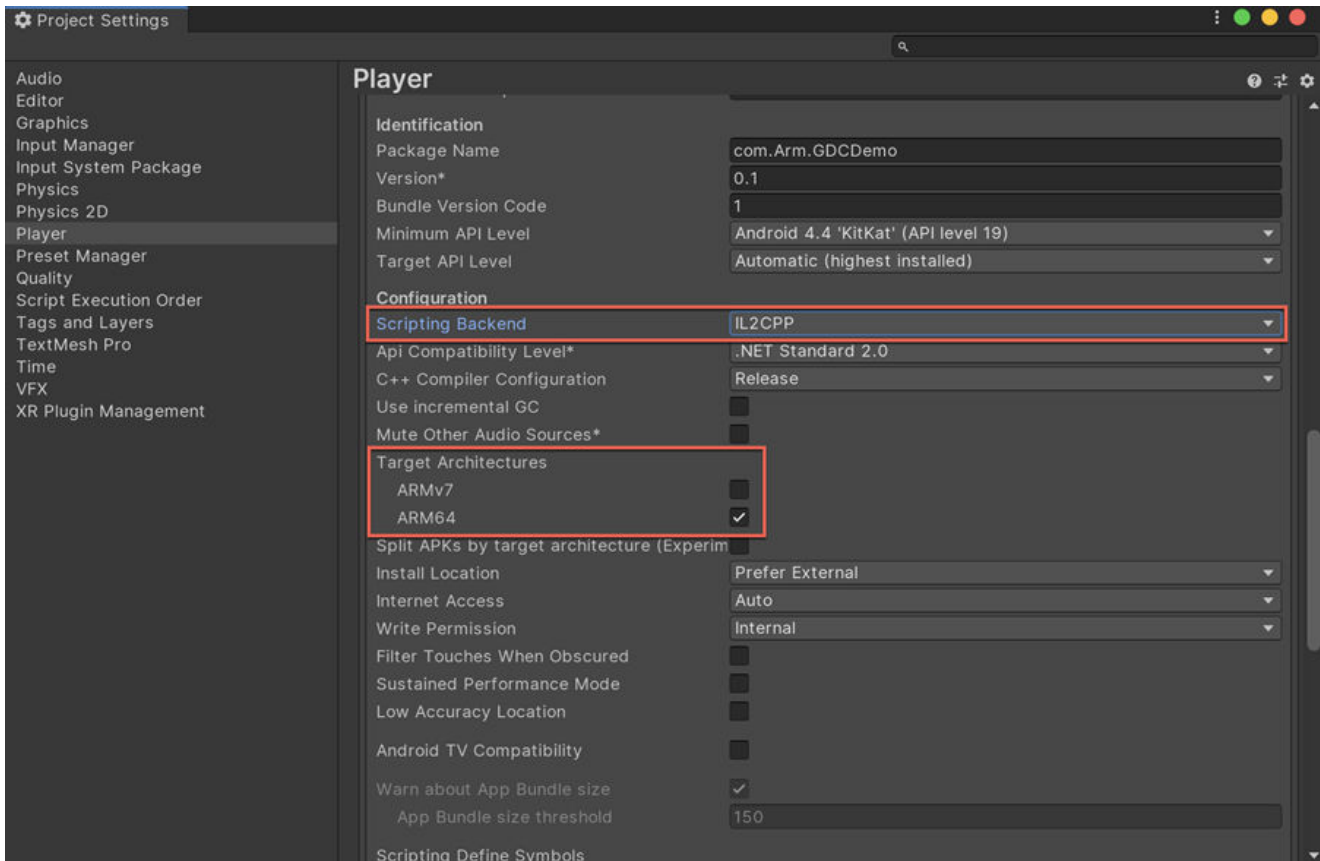


3. Select **File > Build Settings**, then select **Player Settings**.
4. Under **Identification**, set **Target API Level** to the required Android version.

————— **Note** —————

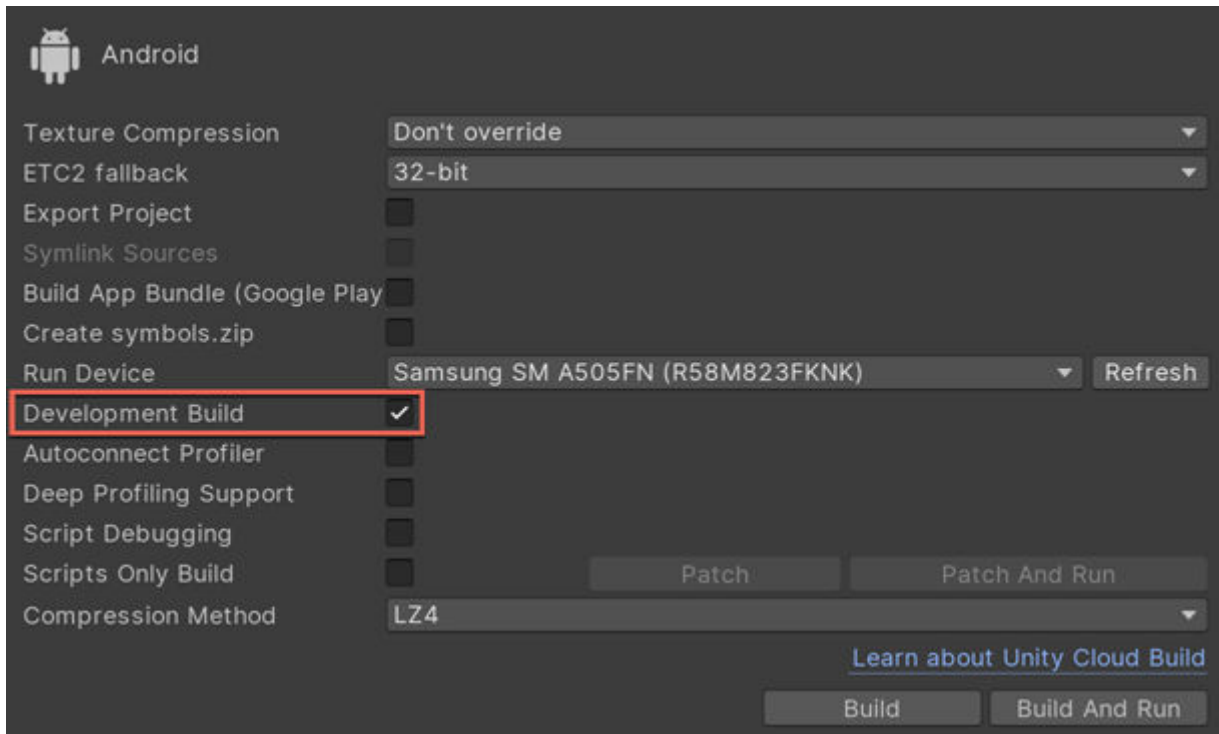
By default, **Target API Level** is set to the latest version of the Android SDK tools that you have installed. If you change to a lower API level, ensure that you have the SDK tools for that version installed. If you build for a higher API version later, change this setting accordingly.

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](#).
- For a full tutorial on using Arm Mobile Studio with Unity, see [Integrate Arm Mobile Studio with Unity](#) on the Arm Developer website.

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 files in your applications gradle file.

```
android {
    sourceSets {
        main {
            jniLibs.srcDirs += '<install_directory>/performance_advisor/lwi/target/
android/arm/<32-bit or 64-bit>'
        }
    }
}
```

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 on page 3-25](#).
- [3.2 Choose a counter template on page 3-27](#).
- [3.3 Capture a Streamline profile on page 3-28](#).
- [3.4 Generate a performance report on page 3-29](#).
- [3.5 Setting performance budgets on page 3-31](#).
- [3.6 Generate a custom report on page 3-33](#).

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
```

The `lwi_me.py` script defaults to capturing a 64-bit OpenGL ES application. To capture a 32-bit application, use the `--32bit` option. To capture a Vulkan application, use the `--lwi-api vulkan` option.

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/64-bit/libGLLES_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 2021.0** and select **Arm MS Streamline 2021.0**.
 - 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 target device type. Then select your device from the list of detected targets, or enter the address of your target.

6. Android users only, select the package you want to profile from the list of packages available on the selected device.
7. TCP users only, optionally enter the details for any command you want to run on the application.

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

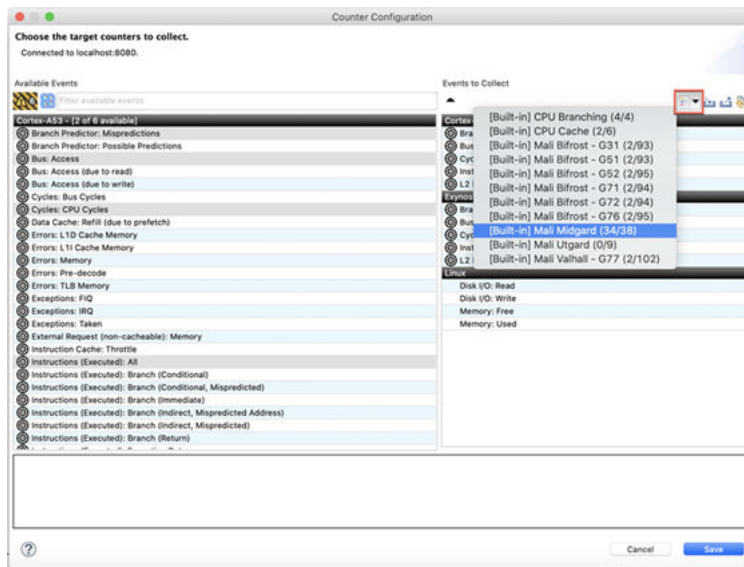
Counter templates are pre-defined sets of counters that enable you to review the performance of both CPU and GPU behavior. Choose the most appropriate template for the GPU in your target device.

Prerequisites

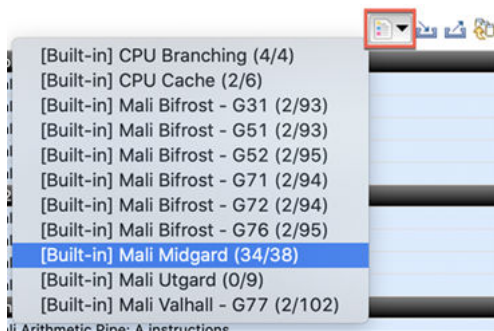
Follow the instructions detailed in [3.1 Connect Streamline to your device on page 3-25](#) before you choose your counter template.

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

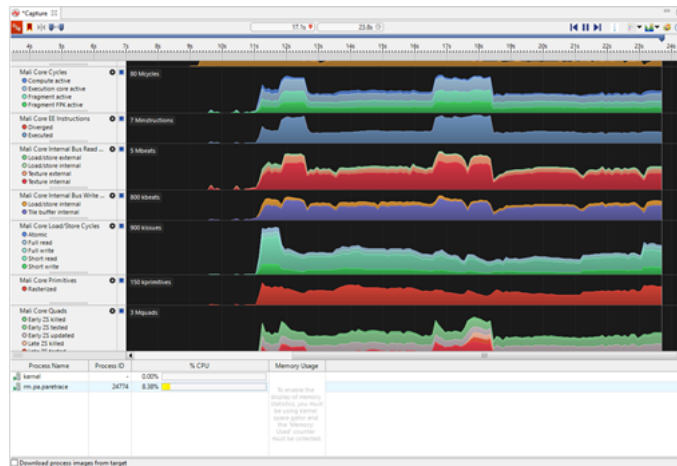
Start a capture session to profile data from your application in real time. When the capture session ends, Streamline automatically opens a report for you to analyze later.

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 `gator` 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 `gator` 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]
```

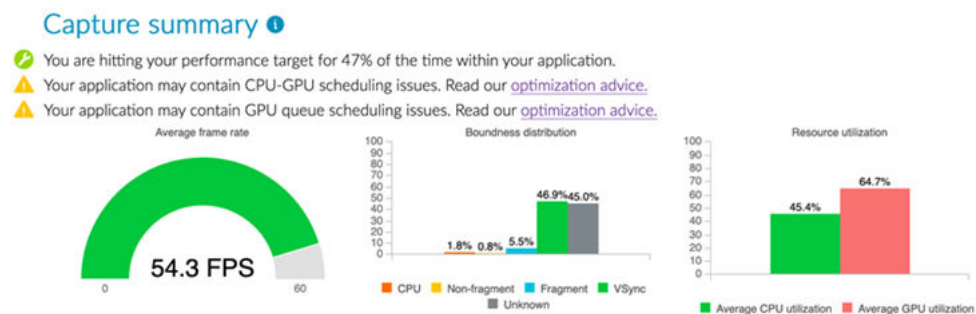
To control how the `pa` command runs, you can pass various options to it. See [A.1 The `pa` command on page Appx-A-55](#) for detailed descriptions of all the available options. You can also add multiple command-line options to a file that you pass to the `pa` command, see [A.1.1 `pa` command-line options file on page Appx-A-57](#) 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>
```

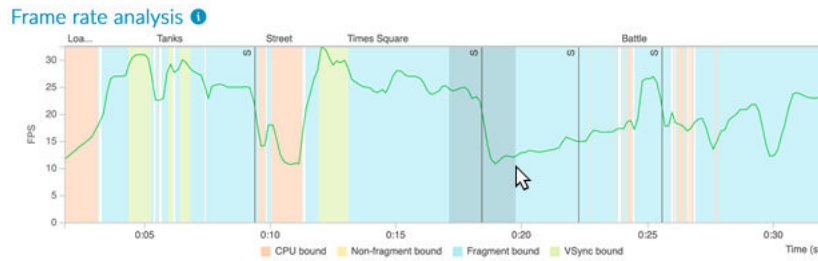
If any scheduling issues are detected, Performance Advisor shows an indicator at the top of the report.



- 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 4-39

[4.3 Generate multiple report types](#) on page 4-42

Related references

[A.1 The pa command](#) on page Appx-A-55

Related information

[Optimization advice](#)

3.5 Setting performance budgets

As different target devices have different performance expectations, it is a good idea to set your own performance budgets based on the expected GPU performance.


If you know the top frequency for the GPU, and you have a target frame rate, you can calculate the maximum GPU cost per frame:

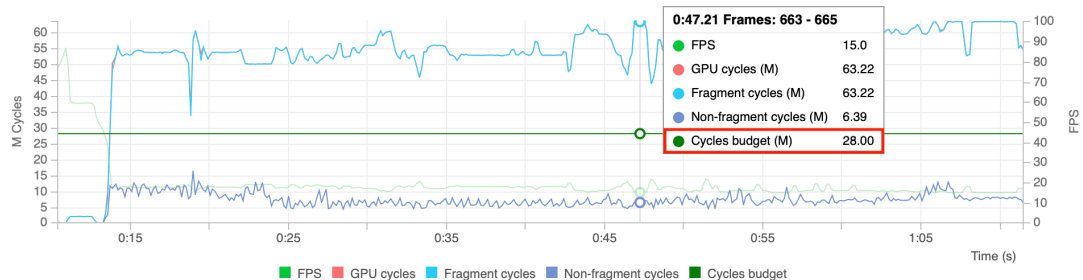
GPU maximum frequency / frame rate = maximum GPU cycles per frame

For example, if you want a minimum frame rate of 30fps on a device with a GPU with a maximum frequency of 940MHz, you can assume that the device can handle 31 million GPU cycles per frame.

940MHz / 30fps = 31.3M

When you generate Performance Advisor reports for this device, you can specify a maximum budget for GPU cycles per frame with the `--gpu-cycles-budget=<value>` command-line option to the `pa` command. This budget is then shown on the GPU cycles per frame chart, making it easy to see when the application has broken the budget. Here, we set a budget of 28 million GPU cycles per frame but the number of fragment cycles is significantly higher than 28 million. This difference means the application is fragment bound.

GPU cycles per frame 



All the per-frame charts in a Performance Advisor report can display a budget in this way.

This section contains the following subsection:

- [3.5.1 Generating a report with per-frame performance budgets on page 3-31.](#)

3.5.1 Generating a report with per-frame performance budgets

To generate a Performance Advisor report where the charts show your own performance budgets for a device, use the relevant command-line options with the `pa` command.

Command-line option	Budget
<code>--bandwidth-budget=<value></code>	Threshold for read/write bytes.
<code>--cpu-cycles-budget=<value></code>	Threshold for CPU cycles.
<code>--draw-calls-budget=<value></code>	Threshold for draw calls.
<code>--gpu-cycles-budget=<value></code>	Threshold for GPU cycles.
<code>--overdraw-budget=<value></code>	Threshold for overdraw.
<code>--pixels-budget=<value></code>	Threshold for pixels.
<code>--primitives-budget=<value></code>	Threshold for primitives.
<code>--shader-cycles-budget=<value></code>	Threshold for shader cycles.
<code>--vertices-budget=<value></code>	Threshold for vertices.

For example:

```
pa mycapture.apc -gpu-cycles-budget=28000000
```

To make it easy to pass in several budgets, you can create a file containing your budget options. Pass this file directly to the `pa` command when generating the report. Refer to [A.1.1 *pa* command-line options file](#) on page Appx-A-57 for detailed instructions.

3.6 Generate a custom report

To focus on the metrics that are most important to you, define which charts are included, and where they are shown, on the Performance Advisor report.

The charts available for you to include in your report are based on a subset of Streamline charts that are suitable for processing as "per-frame" data.

Prerequisites

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. Specify which charts you want to include in the report:
 - Use the `--chart-list-output` option to generate a JSON custom report definition file, containing all possible charts that you can plot on the report. Remove the charts that you do not want to appear on the report. Fixed format charts, from the standard report, appear at the top of report definition generated by the `--chart-list-output` option
 - Alternatively, create your own JSON custom report definition file containing the names of the charts that you want to see on the report.

Note

Some sample report definition files are available in the `examples` folder.

Example custom report definition file:

```
{
  "groups": [
    {
      "title": "Memory Usage",
      "description": "This group shows the system memory usage charts.",
      "charts": [
        {
          "chart": "Mali Memory Bandwidth",
          "title": "Memory bandwidth per frame",
          "description": "This chart shows the distribution of GPU bandwidth. Minimize external memory access to reduce energy consumption.",
          "threshold": 100000000
        },
        {
          "chart": "Mali Core External Memory Reads",
          "title": "External memory reads per frame"
        }
      ]
    },
    {
      "title": "Texture usage",
      "charts": [
        {
          "chart": "Mali Core Texture Cycles"
        }
      ]
    }
  ]
}
```

2. Enter the Streamline chart name exactly as it is shown in the `--chart-list-output`. The chart name is the only required field.
3. Enter information for the following fields:
 - The charts in your report must be contained within at least one group. The `groups` field enables you to group the charts in your report into different sections. If required, you can add a heading

for each section using `title`. You can also add an introduction that appears on the report, and a `description`, which you reveal on the report using the drop-down icon.

- To add information about the charts in your custom report, you can add a `title` and `description`.
- To show how you are performing against your set per-frame budget, add a `threshold` value.

————— **Note** —————

The `title`, `description` and `threshold` fields are ignored for fixed-format charts, because the standard report format is used.

4. Run Performance Advisor using the following command:

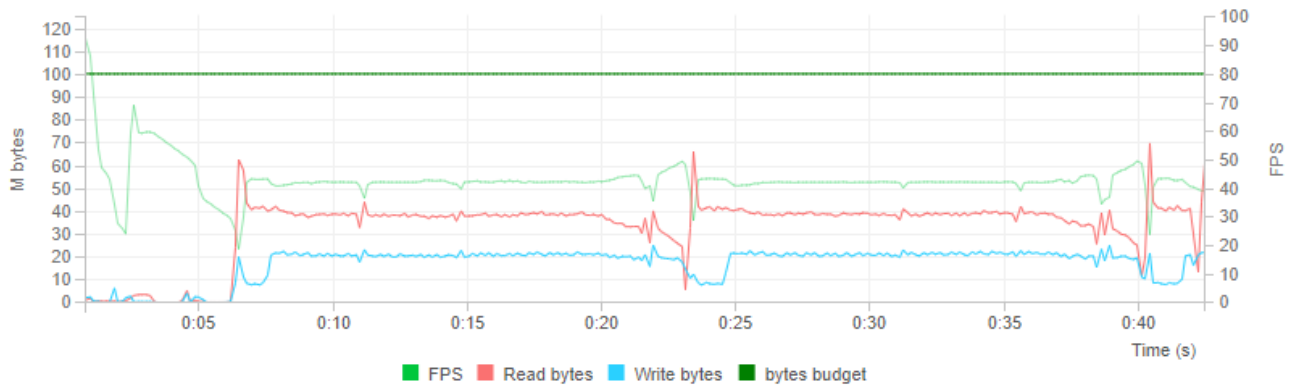
```
pa <filename>.apc --custom-report <path to configuration file> [options]
```

Results:

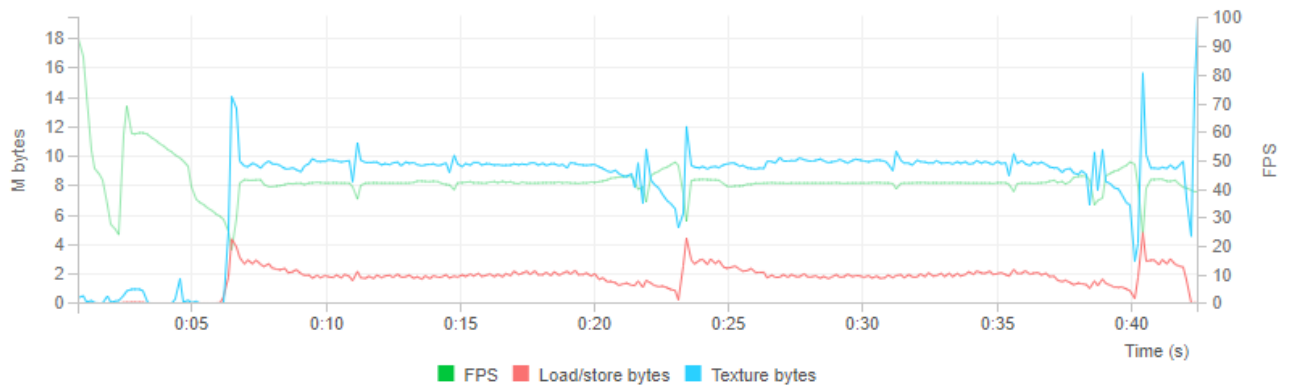
Performance Advisor generates a custom report containing the charts specified in the custom report definition file, and any [pa command on page Appx-A-55](#) options specified. For example:

Memory Usage

Memory bandwidth per frame

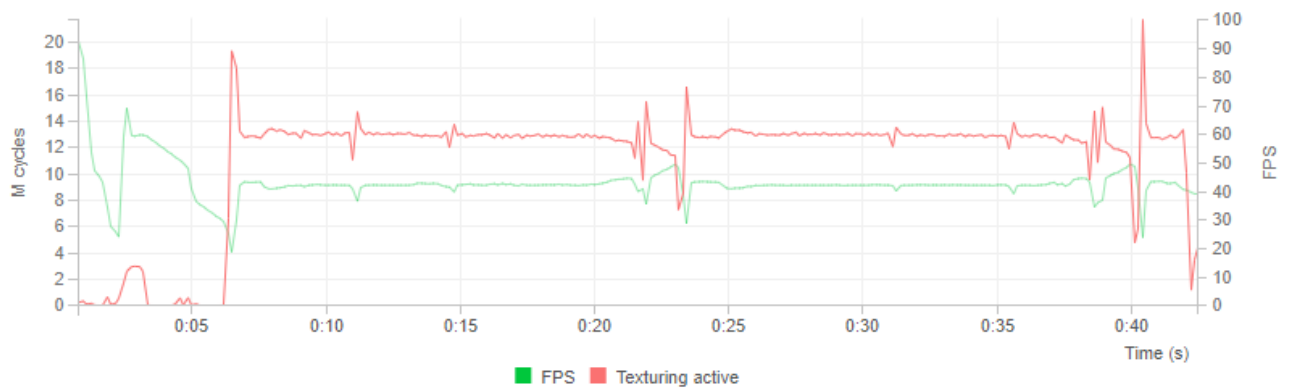


External memory reads per frame



Texture usage

Mali core texture cycles per frame



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 on page 4-37.](#)
- [4.2 Export performance data as a JSON file on page 4-39.](#)
- [4.3 Generate multiple report types on page 4-42.](#)
- [4.4 Generate a JSON diff report on page 4-43.](#)

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/64-bit/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
```

The `lwi_me.py` script defaults to capturing a 64-bit OpenGL ES application. To capture a 32-bit application, use the `--32bit` option. To capture a Vulkan application, use the `--lwi-api vulkan` option. Add any other options you require, refer to [A.2 The lwi_me.py script options on page Appx-A-58](#) for information.

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 [A.1 The pa command](#) on page Appx-A-55.

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  
    }  
  ],  
  ...  
}
```

Note

To aid writing parsers, JSON Schema definitions are provided in the `performance_advisor/json_schemas` directory.

Related tasks

[3.4 Generate a performance report on page 3-29](#)

[4.3 Generate multiple report types on page 4-42](#)

Related references

[A.1 The `pa` command on page Appx-A-55](#)

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 4-39](#)

Related references

[A.1 The pa command on page Appx-A-55](#)

4.4 Generate a JSON diff report

To see how changes in your application affect performance, generate a diff report between two JSON reports to compare differences in performance metrics.

Prerequisites

You must have already generated two JSON reports. For help on exporting data as a JSON file, see [4.2 Export performance data as a JSON file on page 4-39](#).

Procedure

1. Generate a JSON diff report using the following command:

```
./pa --diff-report path/to/previous_json_report.json path/to/current_json_report.json
```

This command subtracts the values in `previous_json_report.json` from the values in `current_json_report.json`.

Performance Advisor generates a file called `performance_advisor_diff-<timestamp>.json`, for example `performance_advisor_diff-210128-105937.json`. To specify a location for this file, use the `--directory` option.

Alternatively, to specify the filename of the JSON diff report, use the following command:

```
./pa --diff-report-output mydiffreport.json path/to/previous_json_report.json \
path/to/current_json_report.json
```

To specify a location for the report, include the path in the filename or use the `--directory` option (see example).

Note

JSON diff reports can be validated against the JSON schema in `performance_advisor/json_schemas/pa_json_diff_report_schema.json`.

Example 4-1 Generate a report called `mydiffreport` in `myoutputdir`

There are two ways to specify the location of the diff report that `--diff-report-output` generates.

- Include the path to the output directory with the filename:

```
./pa --diff-report-output myoutputdir/mydiffreport.json previous.json current.json
```

- Specify the output directory with the `--directory` option:

```
./pa --diff-report-output mydiffreport.json previous.json current.json \
--directory myoutputdir
```

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 on page 5-45.](#)
- [5.2 Tagging slow frames on page 5-47.](#)

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`.

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.

Procedure

1. In a terminal, navigate to `<install_directory>/performance_advisor/lwi/helpers`, where the Python script `lwi_me.py` is located.
2. 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.

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_gator> --lwi-fps-threshold 30 \
--lwi-frame-gap 100 --lwi-mode capture \
--lwi-out-dir <path_to_frame_captures_directory>
```

The script defaults to configuring a capture of a 64-bit application. To capture a 32-bit application, use the `--32bit` option. Also, the script defaults to capturing OpenGL ES applications. To capture a Vulkan application, use the `--lwi-api vulkan` option. See [A.2 The `lwi_me.py` script options](#) on page Appx-A-58 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 5-47 for more information.
- If you experience problems capturing slow frames on Vulkan applications, refer to the FAQ [Slow PA capture on Vulkan apps](#).

3. 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.


4. 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.

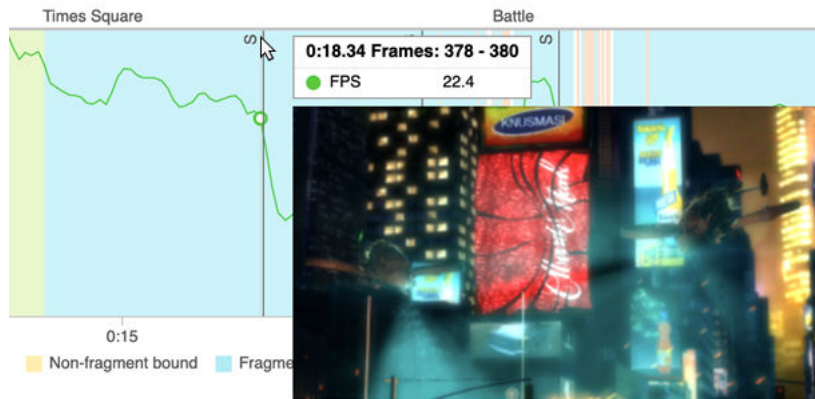
```
pa <my_capture.apc> --frame-capture=<path_to_frame_captures_directory> \
[--directory=<path_to_output_directory>]
```

You can use other options to specify metadata for your report, such as the build name, device name, and application name. See [A.1 The *pa* command on page Appx-A-55](#) for all the available command-line options.

For more information about generating an HTML report, see [4.3 Generate multiple report types on page 4-42](#).

5. 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 --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 --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

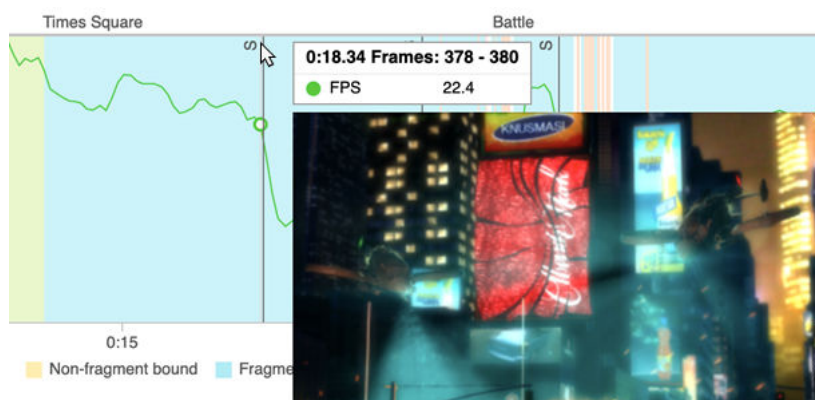
During the Streamline capture, the captured resources are written in the target when the trace reaches the end frame. The default is to end the capture at frame 500. You can adjust the end frame by specifying an alternative value for the `FRAMEEND` parameter of the `lwi_me.py` script.

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 4-42](#).

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 Send annotations from your application code on page 6-49.](#)
- [6.2 Specify a CSV file containing the regions on page 6-52.](#)
- [6.3 Clip unwanted data from the capture on page 6-53.](#)

6.1 Send annotations from your application code

You can send annotations from your application code using the Streamline annotations library.

Procedure

1. Add frame or region boundaries depending on your use case:

You want to avoid adding the lightweight interceptor to your application.

The lightweight interceptor adds annotations to your Streamline capture that identify when frames begin and end. These annotations are then used by Performance Advisor to generate its analysis. If you avoid using the lightweight interceptor, Performance Advisor no longer knows when frames begin and end, and is not able to generate a report. Add frame boundaries yourself from your application code by calling:

```
ANNOTATE_MARKER_STR(FRAME_STR);
```

Where FRAME_STR takes the form of a monotonically incrementing frame number in the following regular expression format:

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

For example:

```
F10  
F11  
F12
```

Note

If you are using `lwi_me` to generate your capture, use the `lwi=off` option to disable the lightweight interceptor.

You want to specify a region from your application code.

Performance Advisor supports regions, which are subsets of time within the capture that represent a particular portion of the game. For example, a region can be a loading screen or a fight level scene within the capture. You can send this information from your application code by calling:

```
ANNOTATE_MARKER_STR(REGION_STR);
```

Where REGION_STR takes the form of:

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

For example:

```
ANNOTATE_MARKER_STR("Region Start Loading Screen");  
...  
ANNOTATE_MARKER_STR("Region End Loading Screen");
```

Performance Advisor creates a region in the report named "Loading Screen" for the time between the two markers.

2. To enable the use of ANNOTATE_MARKER_STR, include the Streamline annotations library in your application using the relevant steps for your code:
 - [Native code on page 6-49](#)
 - [Unity plug-in code on page 6-50](#)
 - [Unreal Engine code on page 6-50](#)

6.1.1 Include the Streamline annotations library in native code

Copy the necessary files into your project and include in the source files where you want annotations.

Prerequisites

The native C code for generating annotations in <mobile_studio_install>/streamline/gator/annotate.

Procedure

1. Include the code in your project by completing one of the following sets of steps.
 1. Copy `streamline_annotate.c` and `streamline_annotate.h` into your projects directory.
 2. Add the following line to any source file where you want to create annotations:

```
#include "streamline_annotate.h"
```

Or

1. Use `make` to compile a `libstreamline_annotate` library build using the makefile within the `annotate` directory.
2. Copy `libstreamline_annotate` into your projects directory.
3. Add the following line to any source file where you want to create annotations:

```
#include "libstreamline_annotate"
```

2. To start a thread to allow annotation for your program, add this line to one of your C files:

```
ANNOTATE_SETUP;
```

6.1.2 Include the Streamline annotations library in Unity plug-in code

Import the Mobile Studio plug-in and set up a define so you can easily remove the plug-in from release builds.

Procedure

1. Open the package manager in Unity.
2. Click + in the toolbar and select **Add package from git URL**.
3. Import the Mobile Studio plug-in from GitHub into your project.

See [Mobile Studio integration for Unity](#) for more information.

Arm recommends that you set up a define so you can easily remove the plug-in from release builds without leaving errors in your code from plug-in usage. To set up the define, follow these steps:

4. If you do not have an `asmdef` file for scripts that reference the Mobile Studio API, create one.
5. In the `asmdef` file, under **Assembly Definition References**, add `MobileStudio.Runtime`.
6. In the `asmdef` file, under **Version Defines**, add a rule:
 1. Set **Resource** to `com.arm.mobile-studio`.
 2. Set **Define** to `MOBILE_STUDIO`.
 3. Set **Expression** to `1.0.0`.

This rule makes Unity define `MOBILE_STUDIO` if the `com.arm.mobile-studio` package is present in the project and its version is greater than `1.0.0`.

7. In your code, wrap `MOBILE_STUDIO` around the Mobile Studio API:

```
#if MOBILE_STUDIO
// Plug-in usage
#endif
```

You can now easily add and remove the plug-in without breaking your project, which avoids errors in release builds.

6.1.3 Include the Streamline annotations library in Unreal Engine code

Copy the necessary files into your project and include in the source files where you want annotations. You might require some additional libraries to compile the code.

Prerequisites

A C++ based project. Blueprint-based projects do not allow you to include external code.

Procedure

1. Follow the instructions in [6.1.1 Include the Streamline annotations library in native code](#) on page 6-49.

Some libraries that are required to compile the given code are not included with many compilers for Windows or within Microsoft Visual Studio. To download these packages within Visual Studio, complete the following steps:

2. Right-click on your project name within the **Solution Explorer** and select **Manage NuGet Packages for <project_name>....**
3. Click **Browse**.
4. Select the **pthread** package.
5. Select all the checkboxes.
6. Click **Install**.

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

[A.1 The pa command on page Appx-A-55](#)

Appendix A

Command-line options

This appendix explains the command-line options that are available for the `pa` command and the `lwi_me.py` script.

It contains the following sections:

- [A.1 The `pa` command on page Appx-A-55.](#)
- [A.2 The `lwi_me.py` script options on page Appx-A-58.](#)

A.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 [A.1.1 pa command-line options file](#) on page Appx-A-57 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.

-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 6-52](#).
- 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
 - customhtml**
Interactive html report containing custom charts
- 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.

Options for creating a custom chart:

--custom-report=path

The path to the JSON report containing the custom chart definitions.

--chart-list-output=path

Output location of the file containing chart names for the Streamline capture.

Options for creating a diff report:

--diff-report-output=path

Output location for the diff report.

This section contains the following subsection:

- [A.1.1 pa command-line options file on page Appx-A-57.](#)

A.1.1 pa command-line options file

You can list command-line options in a file that you pass to the pa command. Specify one option per line and use "=" to assign values.

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

```
--build-name=8.2
--build-timestamp=3rd March 2021
--application-name=My Awesome Game
--cpu-cycles-budget=100000000
--gpu-cycles-budget=28000000
--shader-cycles-budget=20000000
--draw-calls-budget=350
--vertices-budget=1000000
```

For options that accept a string, such as `--build-name`, `--build-timestamp`, or `--application-name`, note that the string does not need to be enclosed within quotes when it contains multiple words.

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

```
pa capture.apc "@budget"
```

A.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 [OPTIONS]
```

Options

- device or -E**
The target device name. Default = auto detected.
- package or -P**
The application package name. Default = auto detected.
- 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. Default = None for an interactive capture, or `configuration.xml` for a headless capture.
- daemon or -D**
Specify the `<path>` to the `gator` binary you want to use if it is not found automatically.
- no-clean-start**
Disable pre-run device cleanup. Default = enabled.
- no-clean-end**
Disable post-run device cleanup. Default = enabled.
- 32bit**
Specify a 32-bit application.
- 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 y | n**
Compresses images taken when capturing frames, to reduce file size. Default = n
- 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.
- 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.