NVIDIA® NSIGHT™ PERF SDK for Tegra Getting Started Guide

2023.5

Document Revision 5

Table of Contents

Introduction

The NVIDIA Nsight Perf SDK is a toolbox for collecting and analyzing GPU performance data, directly from application code. This guide covers the practical aspects of starting out, running SDK samples, and application integration.

Major features of the SDK are:

- Realtime Perf HUD utility classes provide a high-level, real-time performance metrics visualization, and demonstrate the use of the underlying Periodic Sampler and MiniTrace APIs.
- HTML Report Generator utility classes offer fast top-down performance triage, require minimal application code changes, and demonstrate the use of the underlying Range Profiler API.
- The Range Profiler and GPU Periodic Sampler utility classes provide lower-level access to GPU performance counter collection. Range Profiler offers collection for workloads of interest (with multi-pass collection to increase metric coverage), whereas Periodic Sampler collects metrics at a specified frequency.

System Requirements

Supported Operating systems

- Nvidia Drive OS Linux
- Nvidia Drive OS QNX
- Nvidia Jetson Linux

Supported Graphics API

- Vulkan®
- OpenGL (GLX and EGL)

Supported SOC

- DRIVE AGX Orin DevKit
- DRIVE AGX Orin Reference Board
- DRIVE Orin Nano low-cost Reference Board
- Jetson AGX Orin
- Jetson Orin NX
- Jetson Orin Nano

Release notes can be found in the NvPerf/doc subdirectory of the SDK package.

First Time Use

System Setup

On Linux, enable GPU profiling permissions by adding your user to the debug group using command on the board:
`sudo usermod -a -G debug $USER`

Copy the SDK Package

For Linux, cross compiling is not supported yet. All the steps below happen on the board.
For QNX, the build happens on the host.

Unzip the SDK Package

Linux: tar xvzf NVIDIA_Nsight_Perf_SDK_2023.5_[Public or Pro]_Linux.tar.gz
QNX: tar xvzf NVIDIA_Nsight_Perf_SDK_2023.5_Pro_QNX.tar.gz

You should see the following directory structure:

- `NvPerf`
  - `bin`
  - `doc`
  - `include`
  - `lib`
- `redist`
  - `include`
  - `NvPerfUtility`
- `Samples`
  - `cmake`
  - `NvPerfUtility`
  - `OGL`
  - `Tests (Linux only)`
  - `Vulkan (Linux only)`
  - `VulkanSC (Drive Linux only)`

If you use NVIDIA SDK Manager to install the SDK, the above folders will be placed in the installation directory on your host machine. You need to copy the NvPerf and Samples folders to the board and make sure they are under the same directory.

Proceed with building and running the SDK samples.

Vulkan Samples (Linux Only)

The Vulkan Samples are instrumented versions of the well-known Sascha Willems Vulkan Examples and the Nvidia nvpro ray tracing samples. To find profiler modifications in the sample source code, search for `NV_PERF_ENABLE_INSTRUMENTATION`, or diff against the original repository.

Building the SaschaWillems Samples

1. Acquire a copy of CMake
    a. Minimum Required Version: 3.7
2. Open a shell at the unzipped package directory location.
3. Linux:
    a. Install libvulkan-dev and g++.
    b. From a command shell:
    `cd Samples/Vulkan/SaschaWillems`
    `mkdir build`
    `cd build`
    `cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo`
    `make -j8`
    Or if make doesn't work, try the potentially slower
    `cmake --build .`
    c. On successful build, executables will appear in Samples/Vulkan/SaschaWillems/build/bin/
    `computenbody`
    `libnvperf_grfx_host.so`
    `triangle`
4. Note: if the above don't work, follow instructions at cmake-generators.

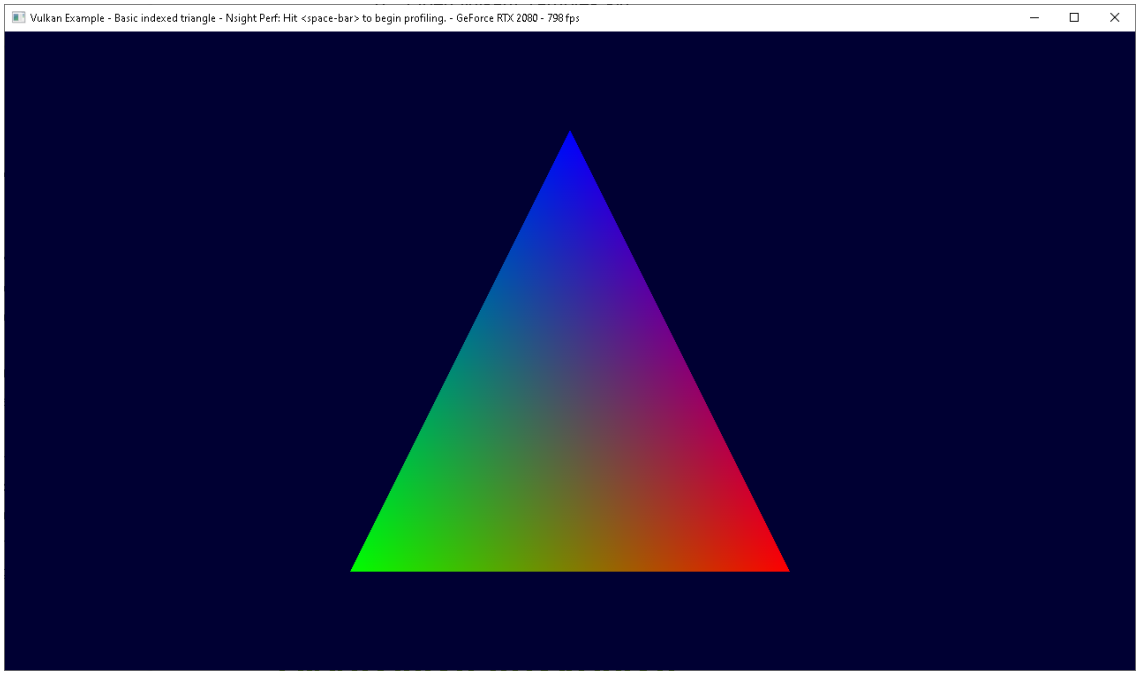
For reference: the original build instructions can be found here.

Range Profiler - Report Generator

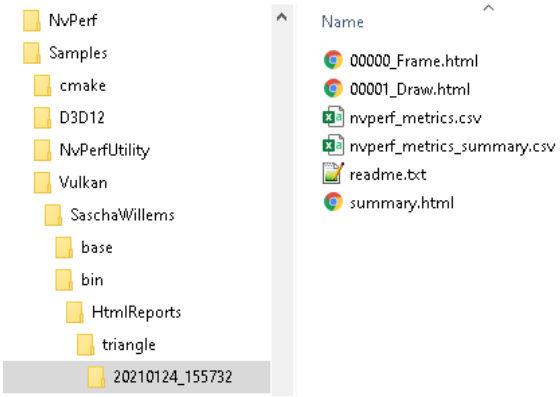
Triangle (3D)

1. **Linux:** set the LD_LIBRARY_PATH environment variable to point at nvperf_grfx_host.so, either in the command shell, or on the command line while running the executable. Since the .so file is copied into the samples' executable directory, the command would look like:
    `LD_LIBRARY_PATH=.`
    `./triangle`

2. Run the triangle program (see preceding section for executable location):



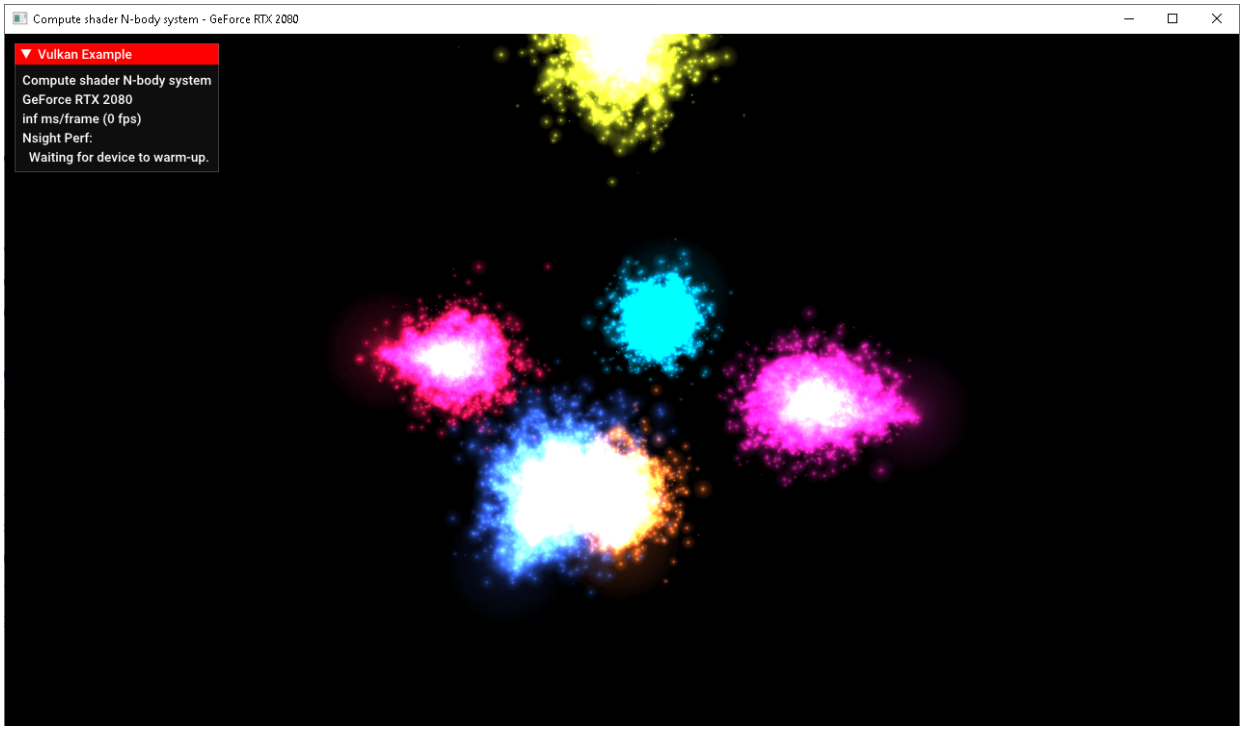
3. **From this step onward: If things don't work as expected, see Troubleshooting.**
4. Hit the **Spacebar** to collect a report.
    a. The title bar should change to say "Currently profiling the frame".
5. When the title bar returns to stating "Hit <space-bar>, it means an HTML profiler report has been written to disk.
    a. You may close the sample program now.
    b. In the current working directory, navigate to ./HtmlReports/triangle
    c. There should be a subdirectory with the current date and time. Navigate into it.



6. Open summary.html for an overview; from there, navigate to per-range reports.
7. See the HTML Report chapter for more information.

Compute N-Body

Follow the same procedure as Triangle (3D).
This sample has a built-in HUD that provides instructions and indicates status.



Building the NvPro Ray Tracing Samples

1. Acquire a copy of CMake
    a. Minimum Required Version: 3.11
2. Open a command prompt or shell at the unzipped package directory location.
3. Linux:
    a. Install libx11-dev, libxrandr-dev, libxinerama-dev, libxcursor-dev, libxi-dev, libgl1-mesa-dev, libxxf86vm-dev, zlib1g-dev, locales
    b. Make sure locale en_US.UTF-8 is supported.
        i. If it's not, run command locale-gen en_US.UTF-8
    c. From a shell:
    ```
    cd Samples/Vulkan/nvpro-samples/vk_raytracing_tutorial_KHR
    mkdir build
    cd build
    cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo
    make -j8
    ```
    d. On successful build, executables will appear in Samples/Vulkan/nvpro-samples/vk_raytracing_tutorial_KHR/build/RelWithDebInfo:
    ```
    libnvperf_grfx_host.so -> libnvperf_grfx_host.so.2023.5.0
    libnvperf_grfx_host.so.2023.5.0
    libshared_sources_vk.a
    vk_animation_KHR
    vk_animation_KHR.exe
    ```

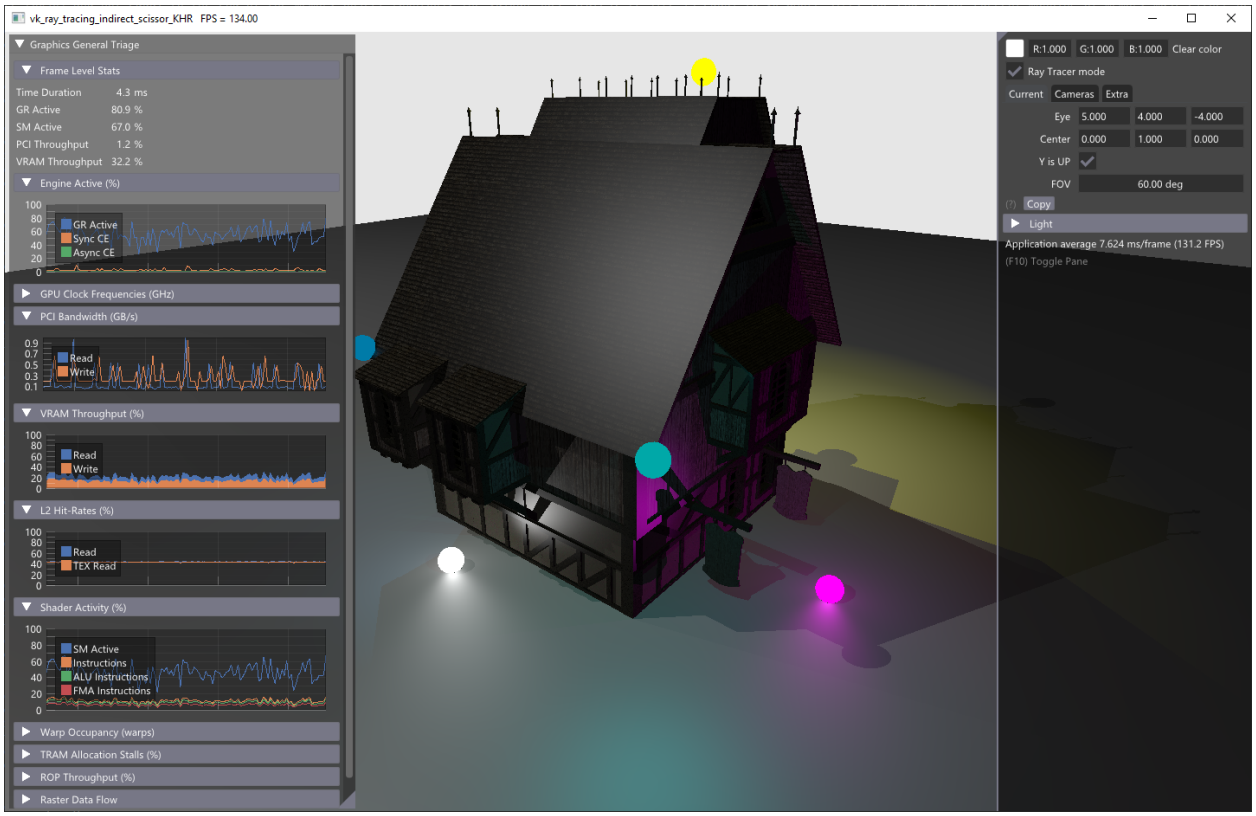
`vk_indirect_scissor_Continuous_KHR`
`vk_indirect_scissor_Continuous_KHR.exe`
`vk_indirect_scissor_HUD_KHR`
`vk_indirect_scissor_HUD_KHR.exe`

4. Note: if the above don't work, follow instructions at cmake-generators. For reference: the original build instructions can be found here.

Periodic Sampler - Realtime Perf HUD

Trace Rays Indirect

Use the steps from Triangle (3D) to launch `vk_ray_tracing_indirect_scissor_KHR.exe`.
See the Realtime Perf HUD chapter for more information.
If things don't work as expected, see Troubleshooting.



Periodic Sampler - Continuous Collection

Trace Rays Indirect

Use the steps from Triangle (3D) to launch `vk_indirect_scissor_Continuous_KHR.exe`.

This sample demonstrates how to use the Periodic Sampler APIs within NvPerfUtility to collect profiling data in a continuous manner, without relying on the perf HUD classes. Instead, the collected data is output directly to the console.

It starts by writing the header line that describes the data format, followed by a line for each sample collected. By utilizing the Periodic Sampler APIs, the sample provides a simple yet powerful way to monitor the performance of an application over time.

```
StartTime, EndTime, Duration, gpc__cycles_elapsed.avg.per_second, sys__cycles_elapsed.avg.per_second, lts__cycles_elapsed.avg.per_second
1679895728345033216, 1679895728353344480, 8311264, 1443433874, 1211023014, 1350332474
1679895728353344480, 1679895728361677824, 8333344, 1408802597, 1184899197, 1334893963
1679895728361677824, 1679895728370011136, 8333312, 1409344494, 1184996037, 1334996803
1679895728370011136, 1679895728378344480, 8333344, 1408891949, 1184928597, 1334925643
1679895728378344480, 1679895728386677792, 8333312, 1409316078, 1184989557, 1334988763
1679895728386677792, 1679895728395011104, 8333312, 1408901237, 1184937597, 1334936203
1679895728395011104, 1679895728403344448, 8333344, 1409345550, 1184982597, 1334984443
1679895728403344448, 1679895728411677760, 8333312, 1408940021, 1184920797, 1334918803
1679895728411677760, 1679895728420011072, 8333312, 1409252022, 1184980797, 1334979763
1679895728420011072, 1679895728428344416, 8333344, 1408833797, 1184910477, 1334911603
1679895728428344416, 1679895728436677728, 8333312, 1409258142, 1184974557, 1334974123
1679895728436677728, 1679895728445011072, 8333344, 1408998246, 1184935797, 1334936923
1679895728445011072, 1679895728453344384, 8333312, 1409401782, 1184987037, 1334988163
```

If things don't work as expected, see Troubleshooting.

Range Profiler - Report Generator

Animation

Follow the same procedure as Triangle (3D) to launch `vk_ray_tracing_animation_KHR.exe`.
See the HTML Report chapter for more information.
If things don't work as expected, see Troubleshooting.

OpenGL Samples (Linux Only)

The OpenGL Samples are instrumented versions of the well-known g-truc OpenGL Samples Pack. To find profiler modifications in the sample source code, search for `NV_PERF_ENABLE_INSTRUMENTATION`, or diff against the original repository.

Building the Samples

1. Acquire a copy of CMake.
    a. Minimum Required Version: 2.8.
2. Open a command prompt or shell at the unzipped package directory location.
3. Linux
    a. Install these libraries (these names correspond to Ubuntu 20.04):
        i. g++
        ii. libglu1-mesa-dev
        iii. libgl1-mesa-dev
        iv. libegl1-mesa-dev
        v. libxrandr-dev
        vi. libxinerama-dev
        vii. libxcursor-dev
        viii. libxi-dev
    b. From a command shell:
    `cd Samples/OGL/gtruc/ogl-samples`
    `mkdir build`
    `cd build`
    `cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo`
    `make -j8`
    Or if make doesn't work, try the potentially slower
    `cmake --build .`
    c. On successful build, executables will appear in Samples/OGL/gtruc/ogl-samples/build/build/Release
    `gl-420-draw-base-instance`
    `gl-430-program-compute`
    `libnvperf_grfx_host.so`

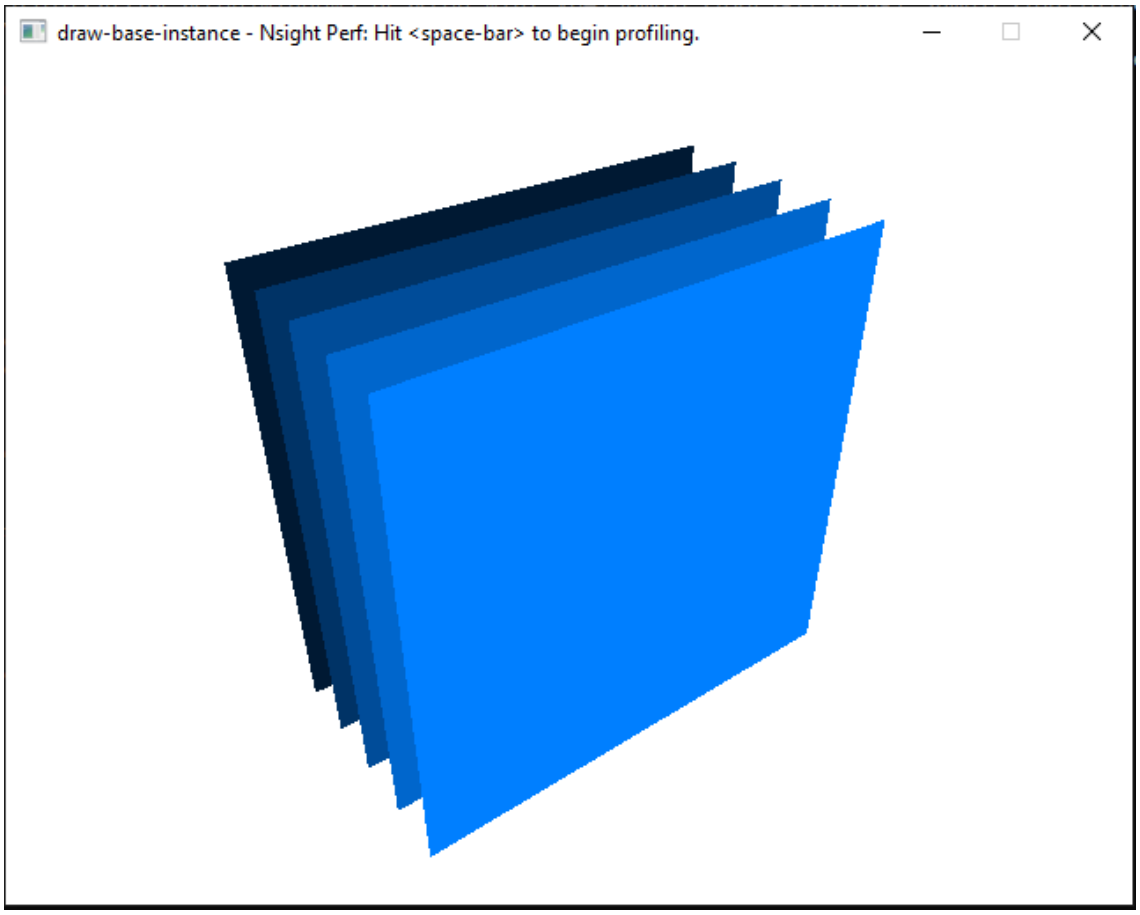
Note: if the above don't work, follow instructions at cmake-generators.
For reference: the original build instructions can be found here.

gl-420-draw-base-instance (3D)

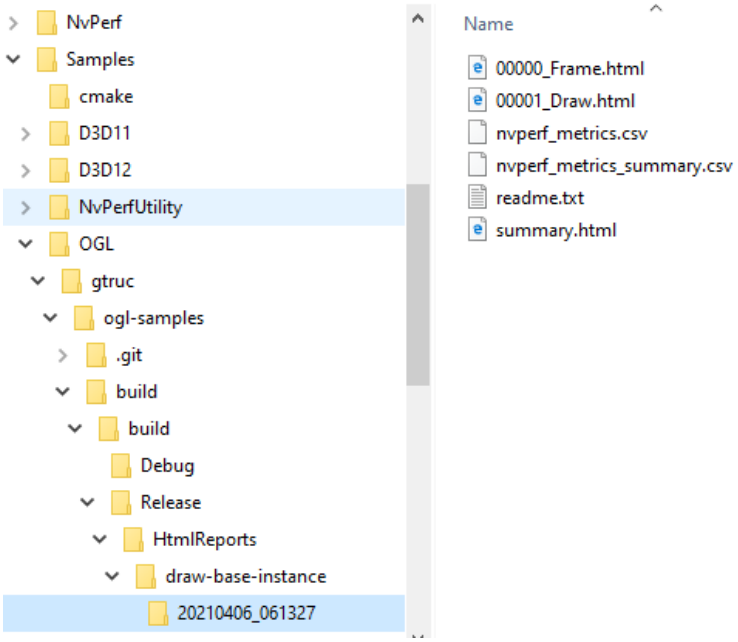
1. set the LD_LIBRARY_PATH environment variable to point at nvperf_grfx_host.so, either in the command shell, or on the command line while running the executable. Since the .so file is copied into the samples' executable directory, the command would look like:
    `LD_LIBRARY_PATH=. ./gl-420-draw-base-instance`
    This command launches the application and the OpenGL context is created from GLX by default. To create the OpenGL context via EGL, use this command:

```
LD_LIBRARY_PATH=. ./gl-420-draw-base-instance --egl
```

2. Run the triangle program (see preceding section for executable location):



3. **From this step onward: If things don't work as expected, see Troubleshooting.**
4. Hit the **Spacebar** to collect a report.
    a. The title bar should change to say "Currently profiling the frame".
5. When the title bar returns to stating "Hit <space-bar>, it means an HTML profiler report has been written to disk.
    a. You may close the sample program now.
    b. Otherwise, in the current working directory, navigate to ./HtmlReports/draw-base-instance
    c. There should be a subdirectory with the current date and time. Navigate into it.



6. Open summary.html for an overview; from there, navigate to per-range reports.
7. See the HTML Report chapter for more information.

OpenGL ES Samples (QNX Only)

Building the Samples

1. Install QNX Software center on a Linux host.
2. Install QNX SDP version 7.1 in QNX software center.
3. Set these enviroment variables:
    a. QNX_HOST = <QNX_7.1_INSTALL_DIR>/host/linux/x86_64
    b. QNX_TARGET = <QNX_7.1_INSTALL_DIR>/target/qnx7
    c. QCC_CONF_PATH = <QNX_7.1_INSTALL_DIR>/host/linux/x86_64/etc/qcc
4. Acquire a copy of CMake.
    a. Minimum Required Version: 3.10.
4. Open a command prompt or shell at the unzipped package directory location.
    a. From a command shell:
    `cd Samples/OGL/QNX`
    `mkdir build`
    `cd build`
    `cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo`
    `-DCMAKE_TOOLCHAIN_FILE=qnx_toolchain.cmake`
    `make -j8`
    Or if make doesn't work, try the potentially slower
    `cmake --build .`
    b. On successful build, executables will appear in Samples/OGL/QNX/build:
    `triangle`

`libnvperf_grfx_host.so`

Note: if the above don't work, follow instructions at cmake-generators.

5. Copy both binaries to the board:
   `scp triangle <user_name>@<board_IP>:/tmp`
   `scp libnvperf_grfx_host.so <user_name>@<board_IP>:/tmp`

6. Make sure the screen service is running in the background. Or the sample cannot create any OpenGL context.
7. Connect to the board via ssh using root. cd to a folder where mkdir is allowed. And then launch the triangle application:
   `cd /storage  # this is the place that allows the user to create folders`
   `LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/tmp /tmp/triangle`
8. The application starts to collect HTML reports on frame 10 automatically. It will exit on frame 300.
9. After the app exits, HTML reports can be found at /storage/HtmlReports/triangle.There should be a subdirectory with the current date and time. Navigate into it.
10. Open summary.html for an overview; from there, navigate to per-range reports.
11. See the HTML Report chapter for more information.

Vulkan SC sample (Drive Linux only)

The Vulkan SC sample is the instrumented sample from the drive OS. To find profiler changes in the sample source code, search for `NV_PERF_ENABLE_INSTRUMENTATION`, or diff against the original repository.

Building the Samples on the board

1. Install the package nv-driveos-linux-vksc-ecosystem-*_amd64.deb to the PDK and targetfs, then flash the board.
   a. This installs the Vulkan SC loader and ICD json config files to the board.
   b. The pipeline cache controller is also installed to the host PDK (path <PDK_ROOT>/drive-linux/vulkansc/pcc/Linux_x86-64/pcc).
2. Use scp or rsync to copy the Vulkan SC headers from the host path <PDK_ROOT>/drive-linux/include/VulkanSC to the board path /usr/include/VulkanSC
3. Acquire a copy of CMake.
   a. Minimum Required Version: 3.18.
   b. The apt version of cmake on the Ubuntu 20.04 board is 3.16. Please download the source code at: https://cmake.org/download/, and build/install it on the board.
4. Install these packages from APT:
   a. libvulkan-dev
5. Open a command prompt or shell at the unzipped package directory location.
   a. From a command shell:
      `cd Samples/VulkanSC/vulkan-sc-samples/`
      `mkdir build`

```
cd build
cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo
make -j8
```
Or if make doesn't work, try the potentially slower
```
cmake --build .
```

b. On successful build, executables will appear in Samples/VulkanSC/vulkan-sc-samples/build/bin:
`libnvperf_grfx_host.so`
`vk_01tri`
`vksc_01tri`

c. Note: if the above don't work, follow instructions at cmake-generators.

d. For reference: the original build instructions can be found here.

Running vksc_01tri

1. Run vk_01tri on target to generate the JSON files as follows:
```
cd <VulkanSC-Samples>/build/bin
export VK_LAYER_PATH=/etc/vulkansc/icd.d
export VK_JSON_FILE_PATH=$PWD/data/pipeline/vksc_01tri/
./vk_01tri
```
Upon successful execution, the following files are generated in VK_JSON_FILE_PATH:
   - `vk_01tri_pipeline_0.json`
   - `vk_01tri_pipeline_0.vert.spv`
   - `vk_01tri_pipeline_0.frag.spv`
2. Mount the path <VulkanSC-Samples> with read and write permission to the host using nfs or sshfs.
3. Generate the pipeline cache by running the PCC tool on the host:
```
cd <VulkanSC-Samples>/build/bin/data/pipeline/vksc_01tri
<PDK_ROOT>/drive-linux/vulkansc/pcc/Linux_x86-64/pcc -chip ga10b -path
./ -out pipeline_cache.bin
```
4. Run vksc_01tri on the board:
```
cd <VulkanSC-Samples>/build/bin
export LD_LIBRARY_PATH=<VulkanSC-Samples>/build/bin
./vksc_01tri
```
5. The application starts to collect HTML reports on frame 10 automatically. It will exit on frame 300.
6. After the app exits, HTML reports can be found at <VulkanSC-Samples>/HtmlReports/triangle.There should be a subdirectory with the current date and time. Copy it to the desktop computer to review.
7. Open summary.html for an overview; from there, navigate to per-range reports.
8. See the HTML Report chapter for more information.

NvPerfUtility Tools (Linux Only)

The NvPerfUtility tools are standalone applications that query and modify GPU and driver state, via the NvPerf API. They also act as examples of NvPerf API usage.

To build the suite of tools:

1. Linux:
    a. Install these app/libraries (exact names from Ubuntu 20.04):
        i. g++
        ii. libglu1-mesa-dev
        iii. libgl1-mesa-dev
        iv. libxrandr-dev
        v. libxinerama-dev
        vi. libxcursor-dev
        vii. libvulkan-dev
    b. From a command shell:
    `cd Samples/NvPerfUtility/tools`
    `mkdir build`
    `cd build`
    `cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo`
    `make -j8`
    Or if make doesn't work, try the potentially slower
    `cmake --build .`
    a. On successful build, executables will appear in Samples/NvPerfUtility/tools/build/bin/:
    `ClockControl`
    `GpuDiag`
    `libnvperf_grfx_host.so`
2. Note: if the above don't work, follow instructions at cmake-generators.

ClockControl Tool

On the DRIVE platforms, the GPU runs at a fixed frequency. On the Jetson platforms, GPU frequency changes based on load. Clock Control tools and APIs are not supported.

GpuDiag Tool

GpuDiag is short for GPU and system diagnostics. This tool is intended to help you troubleshoot profiler issues, and for sharing information in bug reports. It allows you to collect GPU information and system information relating to your video card and graphics profiling.

Command	Description
--html	By default this tool will print JSON to the console. Use "--html path_to_html_file" to generate an html file. The default "path_to_html_file" is GpuDiag.html in the current working directory.

A copy of its JSON Schema has been included in GpuDiag/GpuDiagSchema.json

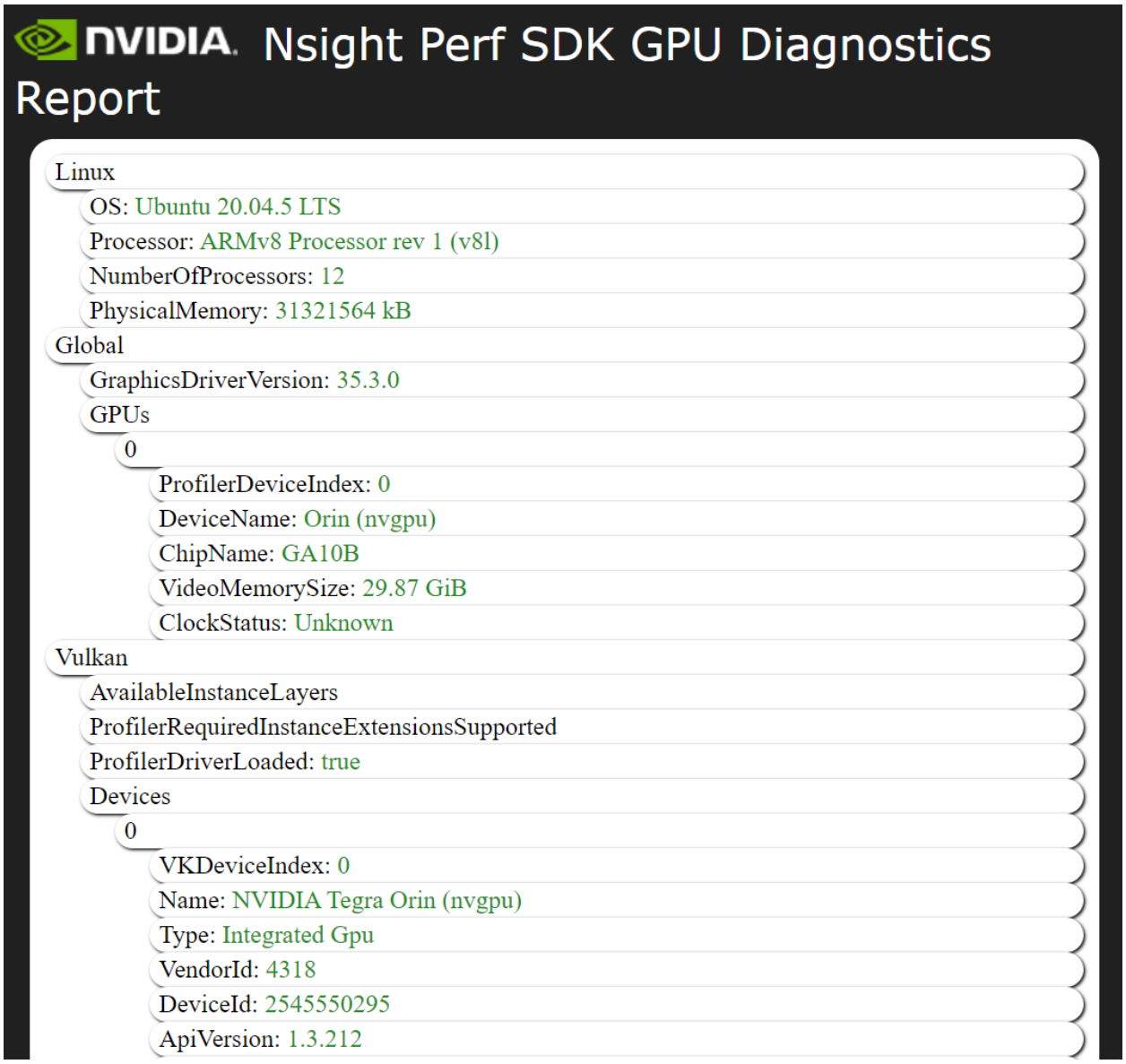
Example output (JSON):

GpuDiag
(console output)


```
{
    "Linux": {
        "OS": "Ubuntu 20.04.5 LTS",
        "Processor": " ARMv8 Processor rev 1 (v8l)",
        "NumberOfProcessors": "12",
        "PhysicalMemory": "31321564 kB"
    },
    "Global": {
        "GraphicsDriverVersion": "35.3.0",
        "GPUs": [
            {
                "ProfilerDeviceIndex": 0,
                "DeviceName": "Orin (nvgpu)",
                "ChipName": "GA10B",
                "VideoMemorySize": "29.87 GiB",
                "ClockStatus": "Unknown"
            }
        ]
    },
    "Vulkan": {
        "AvailableInstanceLayers": [],
        "ProfilerRequiredInstanceExtensionsSupported": {},
        "ProfilerDriverLoaded": true,
        "Devices": [
            {
                "VKDeviceIndex": 0,
                "Name": "NVIDIA Tegra Orin (nvgpu)",
                "Type": "Integrated Gpu",
                "VendorId": 4318,
                "DeviceId": 2545550295,
    ...
```

Example output (HTML):

```
GpuDiag --html
      (Opening the generated GpuDiag.html)
```



NvPerfUtility Tests (Linux Only)

The NvPerfUtility tests are standalone executables for regression testing. They also act as examples of NvPerf API usage.

To Build

1. Linux:
    a. Install these libraries (exact names from Ubuntu 20.04):
        i. libglu1-mesa-dev
        ii. libgl1-mesa-dev
        iii. libxrandr-dev
        iv. libxinerama-dev
        v. libxcursor-dev
        vi. libvulkan-dev
        vii. libxi-dev
        viii. libxxf86vm-dev
    b. From a command shell:
    `cd Samples/Tests/Modules`
    `mkdir build`
    `cd build`
    `cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo`
    `make -j8`
    Or if make doesn't work, try the potentially slower
    `cmake --build .`
    c. On successful build, executables will appear in `Samples/Tests/Modules/build/bin/`:
    `NvPerfEGLTest`
    `NvPerfOfflineTest`
    `NvPerfOpenGLTest`
    `NvPerfVulkanTest`
    `NvPerfVulkanTest_HideSymbols`
    `libnvperf_grfx_host.so`
2. Note: if the above don't work, follow instructions at cmake-generators.

To Run

Each graphics API has its own executable that can be directly executed; there is one additional executable for offline testing. The test-framework is based on doctest.

Example Output

`NvPerfOfflineTest`

```
[doctest] doctest version is "2.4.5"
[doctest] run with "--help" for options
===============================================================================
```

```
[doctest] test cases:    31 |    31 passed | 0 failed | 0 skipped
[doctest] assertions: 93875 | 93875 passed | 0 failed |
[doctest] Status: SUCCESS!
```

For a google-test style output:

NvPerfOfflineTest -r=gtest,console

```
[doctest] doctest version is "2.4.5"
[doctest] run with "--help" for options
[----------]
[ RUN      ] RingBuffer.CounterData
[       OK ] RingBuffer.CounterData (0 ms)
[ RUN      ] FrameLevelSampleCombiner.CounterData
[       OK ] FrameLevelSampleCombiner.CounterData (0 ms)
[ RUN      ] Single Frame Behavior - With Buffer.CpuMarkerTrace
[       OK ] Single Frame Behavior - With Buffer.CpuMarkerTrace (0 ms)
[ RUN      ] Single Frame Behavior - With Complex UserData.CpuMarkerTrace
[       OK ] Single Frame Behavior - With Complex UserData.CpuMarkerTrace (0 ms)
[ RUN      ] Single Frame Behavior.CpuMarkerTrace
[       OK ] Single Frame Behavior.CpuMarkerTrace (0 ms)
[ RUN      ] Multiple Frame Behavior.CpuMarkerTrace
[       OK ] Multiple Frame Behavior.CpuMarkerTrace (0 ms)
[ RUN      ] Negative Tests.CpuMarkerTrace
[       OK ] Negative Tests.CpuMarkerTrace (0 ms)
[ RUN      ] UserData.CpuMarkerTrace
[       OK ] UserData.CpuMarkerTrace (0 ms)
[ RUN      ] ChipSupport.HtmlReport
[       OK ] ChipSupport.HtmlReport (0 ms)
[ RUN      ] MetricsConfiguration.HtmlReport
[       OK ] MetricsConfiguration.HtmlReport (91965 ms)
[ RUN      ] RingBuffer.HudDataModel
[       OK ] RingBuffer.HudDataModel (0 ms)
[ RUN      ] BoolFromYaml.HudDataModel
[       OK ] BoolFromYaml.HudDataModel (0 ms)
[ RUN      ] Color::FromYaml.HudDataModel
[       OK ] Color::FromYaml.HudDataModel (0 ms)
[ RUN      ] StyledText::FromYaml.HudDataModel
[       OK ] StyledText::FromYaml.HudDataModel (0 ms)
[ RUN      ] MetricSignal::FromYaml.HudDataModel
[       OK ] MetricSignal::FromYaml.HudDataModel (1 ms)
[ RUN      ] Panel::FromYaml.HudDataModel
[       OK ] Panel::FromYaml.HudDataModel (0 ms)
[ RUN      ] WidgetFromYaml.HudDataModel
[       OK ] WidgetFromYaml.HudDataModel (0 ms)
[ RUN      ] ScalarText::FromYaml.HudDataModel
[       OK ] ScalarText::FromYaml.HudDataModel (1 ms)
[ RUN      ] Separator::FromYaml.HudDataModel
[       OK ] Separator::FromYaml.HudDataModel (0 ms)
[ RUN      ] TimePlot::FromYaml.HudDataModel
[       OK ] TimePlot::FromYaml.HudDataModel (0 ms)
[ RUN      ] HudConfiguration::FromYaml.HudDataModel
[       OK ] HudConfiguration::FromYaml.HudDataModel (0 ms)
[ RUN      ] HudPresets.HudDataModel
[       OK ] HudPresets.HudDataModel (10 ms)
[ RUN      ] HudDataModel.HudDataModel
[       OK ] HudDataModel.HudDataModel (7977 ms)
[ RUN      ] MetricsEnumeration.MetricsEvaluator
[       OK ] MetricsEnumeration.MetricsEvaluator (12 ms)
[ RUN      ] MetricEvalRequestTwoWayConversions.MetricsEvaluator
```

```
[       OK ] MetricEvalRequestTwoWayConversions.MetricsEvaluator (1 ms)
[ RUN      ] DimUnit.MetricsEvaluator
[       OK ] DimUnit.MetricsEvaluator (3 ms)
[ RUN      ] DimUnitToString.MetricsEvaluator
[       OK ] DimUnitToString.MetricsEvaluator (0 ms)
[ RUN      ] Description.MetricsEvaluator
[       OK ] Description.MetricsEvaluator (1 ms)
[ RUN      ] HwUnit.MetricsEvaluator
[       OK ] HwUnit.MetricsEvaluator (1 ms)
[ RUN      ] Unwind.ScopeExitGuard
[       OK ] Unwind.ScopeExitGuard (0 ms)
[ RUN      ] Dismiss.ScopeExitGuard
[       OK ] Dismiss.ScopeExitGuard (0 ms)
===============================================================================
[doctest] test cases:    31 |    31 passed | 0 failed | 0 skipped
[doctest] assertions: 93875 | 93875 passed | 0 failed |
[doctest] Status: SUCCESS!
[----------]
[==========] 31 test(s) ran.
[  PASSED  ] 31 test(s).
[  SKIPPED ] 0 test(s).
```

Troubleshooting

If things don't work as expected, check for error log messages.

- Linux: by default, errors are sent to stderr.
  - Starting the program from a console should reveal these messages.
- To log to a file instead (on all platforms), set the following environment variable with any file path of your choice. For example:
  - `NV_PERF_LOG_ENABLE_FILE=nvperf_log.txt`
  - `NV_PERF_LOG_ENABLE_FILE=/home/myusername/temp/nvperf_log.txt`
- Set a breakpoint in `NvPerfInit.h nv::perf::UserLog()` to capture callstacks of log events from a debugger.
- Linux: if you receive this `ERR: "NVPW_InitalizeHost failed"`, it is likely you have not run the application by first setting `LD_LIBRARY_PATH` to point to a directory containing the `libnvperf_grfx_host.so`. This library is deployed with all samples, tools, and tests. It is recommended to always run an application like this:
  - `cd bin`
  - `LD_LIBRARY_PATH=. ./application`

In addition, you can use the GpuDiag tool for quick diagnostics.

Application Integration

GPU Periodic Sampler Programming Model

The GPU Periodic Sampler records performance counters at a specified frequency for the whole GPU regardless of the context or chosen graphics API. You can also manually trigger a new sample via CpuTrigger. The list of counters to measure is specified at the start of the session[1]; all counters are measured for every sample.

The profiling hardware has a limited number of counter registers. Since Periodic Sampler offers no multi-pass / replay collection, the counter configuration (ConfigImage) needs to fit within one pass. This limits breadth of observation, but improves accuracy and collection speed.

GPU Periodic Sampler requires no interaction with the graphics API. By the nature of this approach, sampled performance metric data does not automatically correlate to frames. The PeriodicSamplerTimeHistoryVulkan utility class demonstrates how to use the Mini Trace API to get the required events and GPU timestamps, and use the Counter Data Combiner API to merge samples into per-frame data.

Range Profiler Programming Model

The Range Profiler gathers performance counters per annotated *performance marker range*. To create a measurement, the application inserts `PushRange("name")` and `PopRange()` commands into the API command stream, bracketing workloads like draw calls and compute dispatches. Ranges can be nested, forming a hierarchy. The list of counters to measure is specified at the start of the session[2]; all counters are measured for every measurement range.

The profiling hardware has a limited number of counter registers, whereas the API allows any number of counters to be specified; to overcome hardware limitations, the API is designed as a *multi-pass replay-based profiler*. That is, the API expects a *pass* (typically a frame) to be *deterministically replayed* until all counters have been collected, for all ranges. Frameless rendering engines can be profiled as well.

To configure counters, the API accepts a list of counters in string or programmatic-index format, and “compiles” them into an executable configuration (ConfigImage). Each configuration contains a list of passes, each pass containing a list of counters to collect. The number of passes in a configuration will be referred to as `NumConfigurationPasses` in this document.

---

[1] The SetConfig API allows changing the observed set of counters mid-session.
[2] Actually, the SetConfig API function can enqueue additional lists of counters within a session. The fact remains that all counters in the current configuration are collected for all ranges.

To keep measurements accurate, the range profiler API does not insert commands in the middle of a measurement[3]; ranges are isolated from each other, while executing with full parallelism within each range. To fulfill this quality guarantee, child ranges cannot be collected at the same time as parent ranges; instead, additional replays are required to collect each nesting depth. This appears in the API as numNestingLevels.

The API state of the profiled command queue is extended by the following:

- Current configuration (list of counters to collect).
- RangeStack: Stack of string names, updated by PushRange and PopRange commands.

Since the range stack is maintained only in the queue, CommandLists may legally contain unbalanced Push and Pop commands.

Ranges are implicitly popped at the end of each Pass.

See examples of measurements in the following sub-sections.

Example of Typical Usage

Given an abstract API stream, the following measurements will be taken:

```
        Draw(QQ)
PushRange("Player")          // range = Player
PushRange("Head")            // range = Player/Head
        Draw(AA)
PushRange("Eyes")            // range = Player/Head/Eyes
        Draw(BB)
PopRange()
PushRange("Mouth")           // range = Player/Head/Mouth
        Draw(CC)
PopRange() // Mouth
PopRange() // Head
        Draw(DD)
PopRange() // Player
        Draw(RR)
```

Range Hierarchy Name	Measured Draws (with natural parallelism)
Player	AA, BB, CC, DD
Player/Head	AA, BB, CC
Player/Head/Eyes	BB
Player/Head/Mouth	CC

[3] If the application inserts nested ranges, child ranges are deactivated while parent ranges are measured; the deactivated ranges incur GPU FrontEnd No-Operation overhead, on the order of 10s of cycles. For most applications the perturbation is in the noise floor (<< 1%), but if this is a real concern, the application should avoid the use of nested ranges, by managing its own state-machine and only serially emitting PushRange and PopRange commands.

unmeasured workloads	QQ, RR

Example with Unbalanced Ranges

Given an abstract API stream:

- CommandList Alpha
  - PushRange("Bound Shader Foo")
  - Draw(ABC)
- CommandList Beta
  - PopRange() // pop the previous bound shader
  - PushRange("Bound Shader Bar")
  - Draw(DEF)
- Queue
  - PushRange("Frame")
  - SubmitForExecution(Alpha)
  - SubmitForExecution(Beta)
  - PopRange()

Range Hierarchy Name	**Measured Draws (with natural parallelism)**
Frame	ABC, DEF
Bound Shader Foo	ABC
Bound Shader Bar	DEF

Compiling & Linking

General Setup

Add the following to the compiler include paths:

- All platforms:
  - NvPerf/include
  - Samples/NvPerfUtility/include
- Linux additional paths:
  - NvPerf/include/linux-l4t-a64

NvPerfUtility is a header-only library.

At runtime, only one DLL/DSO is required:

- Linux 64-bit: libnvperf_grfx_host.so

nvperf_host_impl.h

To avoid a static DLL/DSO dependency in the client application, NvPerf comes with a set of compilable shim libraries that dynamically load the DLL/DSO, and provide statically linkable symbols. These shim libraries are provided in single header form:

- `NvPerf/include/linux-l4t-x64/nvperf_grfx_host_impl.h`

Limitation: nvperf_grfx_host_impl.h can only be included in one translation unit per linkage unit. Or in other words, can only be included in one .cpp file. Inclusion in multiple translation units will result in a “duplicate symbol” linker error.

CMake find_package

Copy these files to a subdirectory of your project:

```
Samples/cmake/NvPerfConfig.cmake
Samples/cmake/NvPerfUtilityConfig.cmake
```

Then add these statements to your CMakeLists.txt

```
option(NVPERF "Enable Nsight Perf instrumentation" ON)
IF(${NVPERF})
      find_package(NvPerf        REQUIRED PATHS ./cmake)
      find_package(NvPerfUtility REQUIRED PATHS ./cmake)
ENDIF()
```

Adjust the **./cmake** path to point at the location where you copied NvPerfConfig.cmake.

GPU Periodic Sampler, Hud Data Model and HUD Renderer

The GPU Periodic Sampler records performance metrics at a specified frequency without isolating workloads of interest. Because it is desirable to also relate the gathered per-sample

data to the frames during which they where recorded, the PeriodicSamplerTimeHistoryVulkan utility class incorporates a so-called Mini Trace which relays information about frame boundaries to the HudDataModel.

HudDataModel is a graphics-API-agnostic utility class that provides the following:

- A HudConfiguration abstraction that offers predefined sets of metrics suitable for concurrent, single-pass collection.
- An object hierarchy (like a GUI framework) for presentation.
- Logic that generates frame-level samples from temporal samples.

A HudConfiguration consists of multiple *panels* grouped by GPU subunit or topic. Each panel can contain a number of *scalar text* fields and *time plots*. Text fields display values either as per-frame sums or in a processed form (e.g. percentages and averages), whereas plots render values on a per-sample basis. In that case non-processed/total metric values depend on the sampling frequency.

Configurations are stored as YAML files in Samples\NvPerfUtility\build\HudConfigurations\, from which they can also be baked into the NvPerfUtility headers via the hud_configurations_generator.py script. The Realtime Perf HUD chapter has more information on the HUD configurations included in the SDK.

The HudImPlotRenderer utility class offers an ImGui/ImPlot-based visualization solution for HudDataModel.

Note that in the following subsections, the sampler, data model, and renderer do not need to be used together. You can use the sampler as is and process samples yourself. You can create your own renderer on top of HudDataModel. Or you can avoid all utility classes and use the low-level APIs directly.

Vulkan

State

Declare the following objects next to your VkDevice (1:1):

```
nv::perf::sampler::PeriodicSamplerTimeHistoryVulkan m_sampler;
nv::perf::hud::HudDataModel                         m_hudDataModel;
nv::perf::hud::HudImPlotRenderer                    m_hudRenderer;
```

Initialization

Initialize the sampler any time after VkDevice initialization.

```
m_sampler.Initialize(m_instance, m_physicalDevice, m_device);
```

Next, start a recording session and specify the sampling frequency, maximum decoding latency (explained below) and the number of concurrently unfinished frames (maxFrameLatency).

```
uint32_t samplingFrequencyInHz = 60;

uint32_t samplingIntervalInNs  = 1000000000 / samplingFrequencyInHz;
uint32_t maxDecodeLatencyInNs  = 1000000000;
uint32_t maxFrameLatency       = 5;
sampler.BeginSession(m_queue, m_queue, samplingIntervalInNs,
    maxDecodeLatencyInNs, maxFrameLatency);
```

Select a HUD configuration to record via the HudPresets class. Here we select *Graphics General Triage*.

```
nv::perf::hud::HudPresets hudPresets;
auto deviceIdentifiers = m_sampler.GetGpuDeviceIdentifiers();
hudPresets.Initialize(deviceIdentifiers.pChipName);
m_hudDataModel.Load(hudPresets.GetPreset("Graphics General Triage"));
```

Initialize the data model, choose a window of time to store in the TimePlots, and specify the sampling interval.

```
double plotTimeWidthInSeconds = 4.0;
m_hudDataModel.Initialize(1.0 / samplingFrequencyInHz,
    plotTimeWidthInSeconds);
```

Pass the newly created counter configuration to the sampler, and prepare the sample-to-frame data processing pipeline.

```
m_sampler.SetConfig(&m_hudDataModel.GetCounterConfiguration());
m_hudDataModel.PrepareSampleProcessing(m_sampler.GetCounterData());
```

Finally, initialize ImGui, ImPlot and pass the data model into HUD Renderer.

```
ImGui::CreateContext();
ImPlot::CreateContext();
ImGui_ImplVulkan_Init(...);
ImGui_ImplGlfw_InitForVulkan(...);
nv::perf::hud::HudImPlotRenderer::SetStyle();
m_hudRenderer.Initialize(m_hudDataModel);
```

Periodic Operations

Samples to be periodically fetched and processed by the sampler utility classes. Caveat: If this is not done, the sampler can fall into an irrecoverable state. Choose maxDecodeLatency to cover for a large-enough delay. Frame boundaries are recorded so that per-frame values, of e.g. draw call counts, can be shown as well as per-sample values.

```
m_sampler.DecodeCounters();
m_sampler.ConsumeSamples([&](const uint8_t* pCounterDataImage,
    size_t counterDataImageSize, uint32_t rangeIndex, bool& stop) {
        stop = false;
```

```
            return m_hudDataModel.AddSample(pCounterDataImage,
                counterDataImageSize, rangeIndex);
    });
    for (auto& frameDelimiter : m_sampler.GetFrameDelimiters())
    {
        m_hudDataModel.AddFrameDelimiter(frameDelimiter.frameEndTime);
    }
```

Per-Frame Operations

Inform the sampler about the frame end, and use the HUD renderer to visualize the populated data model.

```
m_sampler.OnFrameEnd();

ImGui_ImplGlfw_NewFrame();
ImGui::NewFrame();

ImGui::SetNextWindowSize(ImVec2(400, -1), ImGuiCond_Appearing);
ImGui::Begin("Graphics General Triage");
m_hudRenderer.Render();
ImGui::End();

ImGui::Render();
ImGui_ImplVulkan_RenderDrawData(ImGui::GetDrawData(), commandBuffer);
```

Clean-Up Operations

HudDataModel, HudImPlotRenderer and PeriodicSamplerTimeHistoryVulkan can be destroyed through their respective destructors. ImGui and ImPlot require the following tear-down:

```
ImGui_ImplVulkan_Shutdown();
ImGui_ImplGlfw_Shutdown();
ImPlot::DestroyContext();
ImGui::DestroyContext();
```

Threading Model

HudDataModel, HudImPlotRenderer and PeriodicSampler are a set of single-threaded classes; only one thread may call into them at any given time. If an application needs to call it on multiple threads, the application must apply its own synchronization (e.g. mutex lock).

Range Profiler

Vulkan

VkQueue

- A profiling session can be started on any 3D or Compute queue.
- Activity from all queues within the VkDevice may be measured.
- Ranges can be pushed and popped on the profiled queue, although this is discouraged.
    - Each queue-level push or pop incurs a VkQueueSubmitCommands call, and will not result in high quality measurements for small workloads.
    - Queue-level ranges exist for convenience, to collect entire frames, where the overhead of Submit is relatively small.
    - The range profiler has a MaxQueueRangesPerPass setting.
- Non-profiled queues cannot push or pop ranges; such commands are ignored.
- Only one queue can act as the controller for a profiling session.
- Only one profiling session is supported at a time per GPU.
- Limitation: Copy-only queues are not supported.

VkCommandBuffer

- PushRange and PopRange commands can be recorded into a CommandList.
- The recording of Range commands can occur at any time, including times outside of an active profiling session.
- Limitation: Secondary command buffers are not supported.
- Limitation: SIMULTANEOUS_USE_BIT is not supported.

ReportGenerator

State

Declare an instance of class ReportGeneratorVulkan next to your VkDevice (1:1).

```
nv::perf::profiler::ReportGeneratorVulkan m_nvperf;
```

Initialization

Initialize the report generator any time after VkDevice initialization. This step determines the list of counters. Specify additionalMetrics before calling InitializeReportGenerator. This is also a good time to decide whether a frame-level range is desirable.

```
m_nvperf.additionalMetrics = { "crop__write_throughput" };
m_nvperf.InitializeReportGenerator(instance, physicalDevice, device);
m_nvperf.SetFrameLevelRangeName("Frame");
m_nvperf.SetNumNestingLevels(10);
```

The number of required frame replays = `NumConfigurationPasses * NumNestingLevels`; choose the smallest viable number, to avoid excessively long data collection times. If unknown, a good heuristic for a deeply instrumented engine is 10.

Per-Frame Operations

Use the following call pattern around the vkQueuePresentKHR() call in your application:

```
m_nvperf.OnFrameEnd();
vkQueueWaitIdle(queue);  // workaround to avoid hang
vkQueuePresentKHR(queue, ...);
m_nvperf.OnFrameStart(queue);
```

OnFrameStart and OnFrameEnd will not perform any operation until collection is initiated.

The vkQueueWaitIdle() call in the code sequence above is a workaround for a driver/OS issue.

Initiating Report Collection

Initiate collection with the following call:

```
std::string outDir = ...; // wherever you decide
bool b = m_nvperf.StartCollectionOnNextFrame(outDir,
AppendDateTime::yes);
```

The above call may fail if the directory cannot be created or accessed (file system permissions).

Data collection begins on the subsequent call to `OnFrameBegin()`.

Threading Model

class ReportGeneratorVulkan is a single-threaded object; only one thread may call into it at any given time. If an application needs to call it on multiple threads, the application must apply its own synchronization (e.g. mutex lock).

The preceding statements do not apply to the `rangeCommands` and `deviceIdentifiers` members, which are initialized during `InitializeReportGenerator()`. After initialization, these fields remain immutable and safely usable from multiple threads until the next call to `InitializeReportGenerator`.

PushRange and PopRange

Struct VulkanRangeCommands provides a reliable set of function pointers, that are safe to call on any device (supported, unsupported, or non-NVIDIA). The ReportGeneratorVulkan has a member rangeCommands that is usable after calling `InitializeReportGenerator()`.

The recommended pattern is to insert both KHR DebugGroups and NvPerf range markers:

```
vkCmdBeginDebugUtilsLabelEXT(commandBuffer, ...);
m_nvperf.rangeCommands.PushRange(commandBuffer, "Particle System");
vkCmdDrawIndexed(commandBuffer, ...);
m_nvperf.rangeCommands.PopRange(commandBuffer);
vkCmdEndDebugUtilsLabelEXT(commandBuffer);
```

This pattern will ensure your application can be profiled by many GPU tools.

The command buffer must be in a recording state for PushRange/PopRange to succeed.

Frameless Renderers

The preceding sections described a typical pattern of instrumenting Present calls. However, this is not a technical requirement; frameless rendering is fully supported.

Profiling a frameless renderer requires additional synchronization by the application, because the profiler has a limited number of buffers to operate on. That number is specified in `SessionOptions::numTraceBuffers`. Typically the Present() operation includes some synchronization, preventing the CPU from running too far ahead of the GPU.

The typical solutions (from easiest to most difficult) are:

1. Insert a CPU/GPU sync before OnFrameEnd().
    a. Vulkan: `vkQueueWaitIdle()`
2. Increase the numTraceBuffers argument, and carefully ensure the CPU never enqueues more than that many replays before synchronizing.

Vulkan Code Pattern

```
m_nvperf.OnFrameStart(queue);
RenderScene(queue);
vkQueueWaitIdle();
m_nvperf.OnFrameEnd();
```

Interop with other Nsight Products

Other NVIDIA tools like Nsight Graphics, Nsight Compute, and Nsight Systems use the NvPerf library to take measurements. To ensure those tools continue to work with your application, take the following measures:

1. Make sure there is a way to disable the use of NvPerf at a whole application level.
2. Consider adding a dynamic toggle, to allow disabling in-application use of NvPerf just before activating the other Nsight tool's profiler. Only one tool can use the profiling hardware at a time (per GPU), but alternating between tools is supported.

Ensuring Stable Measurements

Range profiling requires the same frame to be replayed repeatedly, while collecting a different subset of the counters on each pass. For measurements to be stable and self-consistent, the replayed frames must be identical. Frame-to-frame variability and non-determinism will result in counter values having a higher error margin. Each subsection below describes a different source of variability, and ways to mitigate it.

GPU Clocks - Lock Clock

Force GPU clocks to stable values, by locking the GPU at a fixed frequency.

1. Stable clock values may be lower than usual. This setting also prevents the GPU from entering “boost” clock frequencies. The goal is stable and comparable measurements run-to-run, not absolute peak performance.
2. The clock setting is system global, and is not controlled by application.

Enforce Application Determinism

1. While profiling the 3D Queue, do not allow Async Compute queues to arbitrarily execute simultaneously with 3D. Use a trace tool or realtime activity to investigate those scenarios instead.
2. “Flattening” execution onto one queue, or serializing execution across queues with additional fences, will prevent random overlap and improve the quality of results.
3. Always instrument at the CommandList / CommandBuffer level. Queue-level instrumentation is inherently variable, with noise coming from WDDM and/or hardware schedulers.
4. Prevent simulation timesteps from ticking forward.
    a. Simulation state managed entirely on the GPU may be difficult; many modern particle systems work this way. If possible, either support a timestep of zero, or save/restore input buffers (outside of the measurement range).
5. Be aware of cache-warming effects across workloads. It is better to replay a sequence of workloads while measuring the “middle” workloads, than to replay only a single workload by itself.

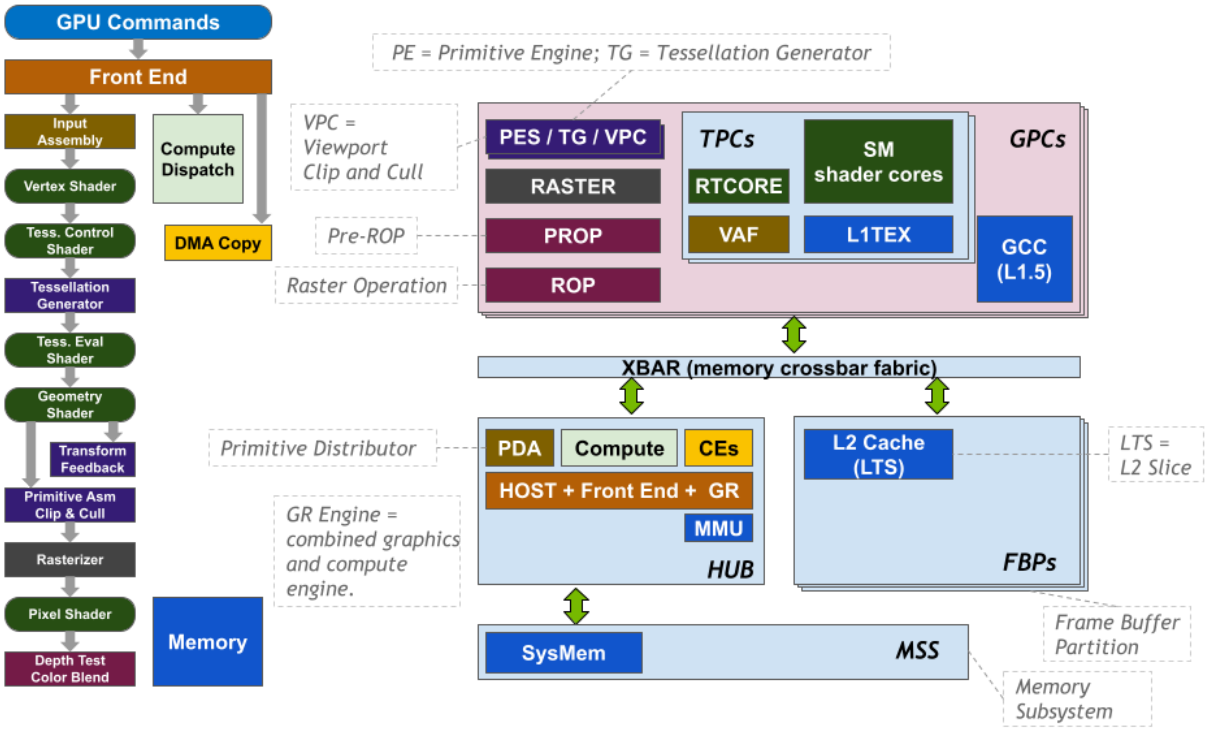
Avoid Running Background Tasks

Background tasks may interfere with GPU scheduling, and introduce non-determinism.

1. Using a web browser, playing videos in the background, etc. will interfere with profiling results, by causing the GPU to timeslice. This affects determinism.
2. Running other graphical programs in the background will cause GPU time slicing and change GPU scheduler behavior.
3. Even moving the mouse may cause CPU interrupts that subtly influence system behavior.

GPU Block Diagram

The following diagram contains a logical graphics pipeline on the left, and a physical block diagram representing an NVIDIA GPU on the right. The diagrams are color-coded, so that logical elements share the same color as their corresponding physical blocks.



Hardware Block	Description	Metrics
GR Engine	Combined 3D graphics and compute engine.	gr__
FE	The Front End unit, that fetches GPU commands from memory and dispatches them within the GR engine.	fe__
TPC	Texture Processing Cluster. Contains SM, L1TEX, and a portion of the Primitive Engine.	tpc__
SM	Streaming Multiprocessor. This unit contains the shader cores (warp schedulers), and instruction pipelines.	sm__

RTCORE	RT Core. Dedicated hardware units for accelerating Bounding Volume Hierarchy (BVH) data structure traversal, and performing the ray-triangle and ray-bounding box intersection testing calculations that are critical for ray tracing.	rtcore__
L1TEX	Combined L1 Data and Texture cache. It has two parallel pipelines: the LSU or load/store unit, and TEX for texture lookups and filtering.	l1tex__
GCC (L1.5)	The L1.5 Constant Cache for shader instructions, constant buffers, and texture/sampler descriptor headers.	*gcc*
L2 Cache	The GPU's L2 cache is its central point of coherence; nearly all GPU memory traffic flows through it.	lts__
MMU	Memory Management Unit. Responsible for GPU virtual to physical address translation.	*mmu*
PDA	Primitive Distributor, that fetches from 3D index buffers.	pda__
VAF	Vertex Attribute Fetch unit. The portion of the Primitive Engine that fetches Vertex Shader attributes.	vaf__
PES	Primitive Engine Shared. Contains the Transform Feedback logic, in addition to Tess Gen. and VPC units.	pes__
TG	Tessellation Generator unit.	
VPC	Viewport Clip and Cull unit, corresponding to Primitive Assembly before the rasterizer.	vpc__
RASTER	Rasterizer units that convert primitives into screen-space fragments and depth samples. Includes ZCULL.	raster__
ZCULL	Coarse resolution depth testing, that occurs earlier in the pipe from either Early-Depth-Test (EarlyZ) or Late-Depth-Test (LateZ).	raster__zcull_
PROP	Pre-ROP orchestrates the flow of screen-space work, ensuring API ordering rules are followed. The raster data flow through EarlyZ, Pixel Shading, LateZ, and surviving Color samples can be observed at this stage.	prop__
ROP	Raster Operation Stage, containing ZROP and CROP.	rop__
ZROP	Z-ROP performs the final depth-test, stencil-test, and related buffer updates.	zrop__
CROP	Color ROP performs the final color blend into bound render targets.	crop__

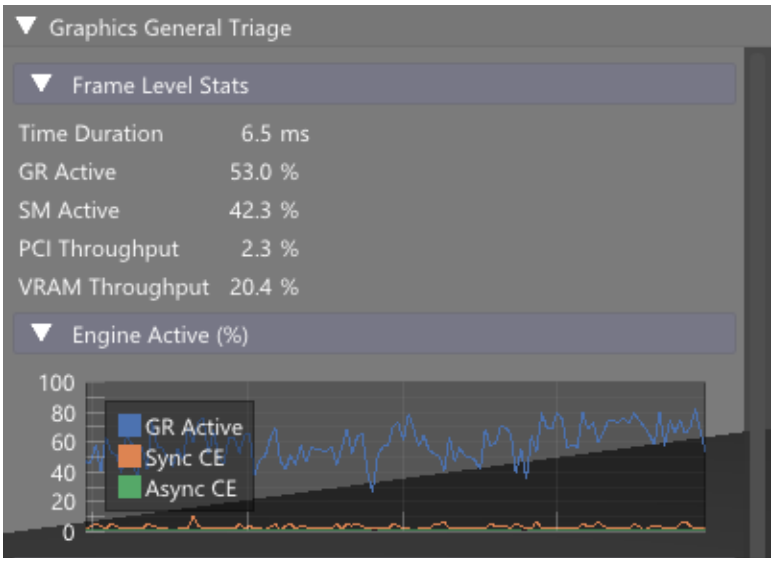
Additional resources

1. Introducing NVIDIA Nsight Perf SDK: A Graphics Profiling Toolbox (GTC 2021)
2. GPU Performance Analysis and Improvements utilizing the NVIDIA Nsight Perf SDK (GTC 2022)
3. Life of a triangle - NVIDIA's logical pipeline
4. The Peak-Performance-Percentage Analysis Method for Optimizing Any GPU Workload
5. Nsight Graphics Advanced Learning for GPU Trace Metrics
6. Nsight Compute Kernel Profiling Guide

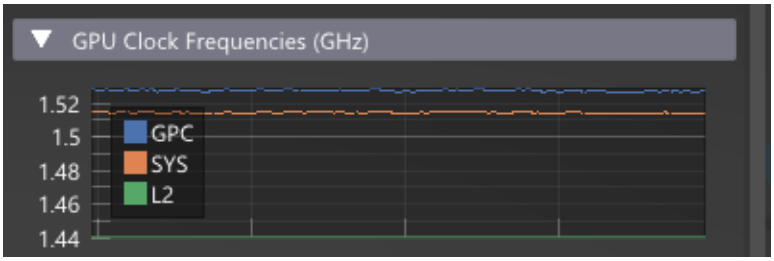
Realtime Perf HUD

The SDK includes one *Graphics General Triage* HUD configuration for the Ampere GA10B GPU architectures. This configuration provides a broad overview of GPU subunit activity. For that it presents various throughputs in a top-down fashion from external memory down to the shader cores.

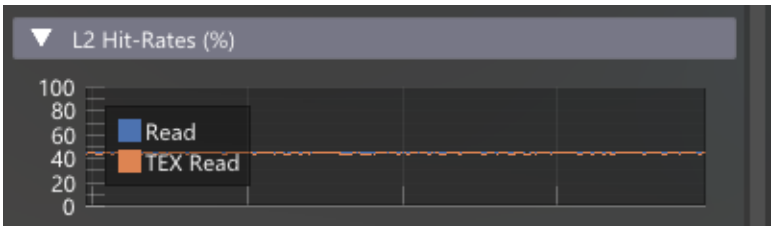
The Frame Level Stats and Engine Active panels offer top-level info on core throughputs including Frame time and Copy Engine activity.



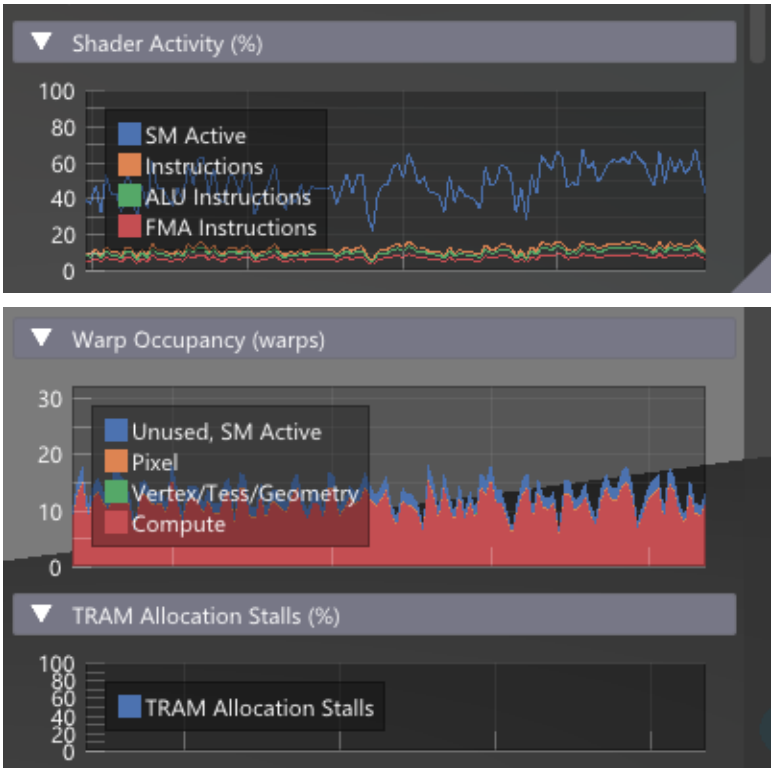
Clock frequencies plotted over time help understand throttling behavior (of particular interest on laptops), and provide context for the other plots.



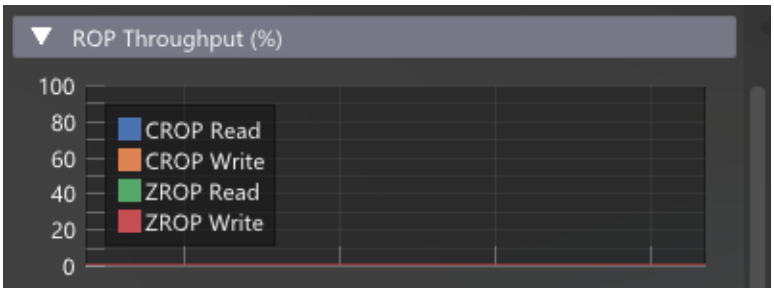
L2 Hit-Rates offer core performance data on the interconnect and memory performance.



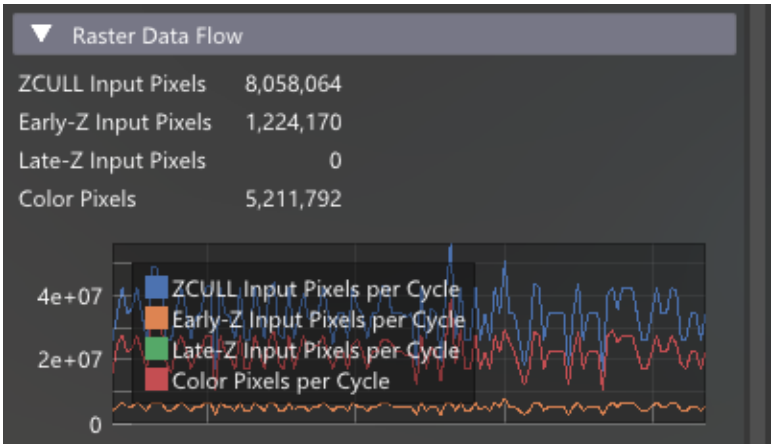
Shader Activity offers insight on SM utilization. If SM utilization is low, identify whether the limiter is low Warp Occupancy. Further identify if low Warp Occupancy was caused by excessive pixel shader attributes, indicated by high “TRAM Allocation Stalls”.



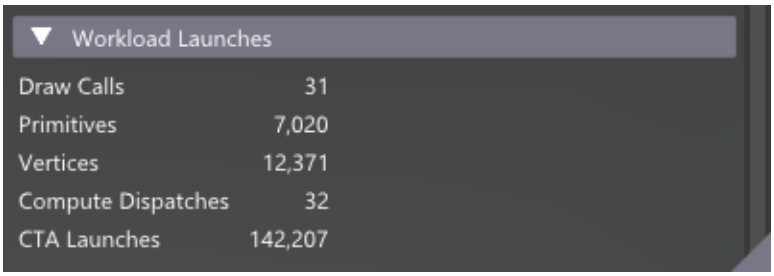
The ROP Throughput panel summarizes CROP and ZROP utilization.



Raster Data Flow gives a front-to-back view of the raster pipeline. How many pixels enter it. How many of these enter early and late depth-tests. And how many pixels are finally sent to CROP for blending. All metrics are shown per-frame and are plotted over time as per-cycle values.



Finally, the Workload Launches panel offers global per-frame counts of relevant API launch events.



Caveats

If the sampling rate is close to the frame rate, plots can look choppy because subsequent samples cover different intervals of subsequent frames. On the other hand if the sampling rate is high, a natural choppiness arises as well, because different parts of a frame execute unrelated workloads. Visually in the current renderer implementation, best results are achieved with a sampling rate below or around the frame rate.

Due to hardware limitations, some counters have inherent variability while measuring small workloads. For example, if a workload does not execute on all SMs (shader cores), the frame-to-frame work distribution within the GPU will vary, potentially impacting the counter's quality. This problem diminishes on bigger workloads.

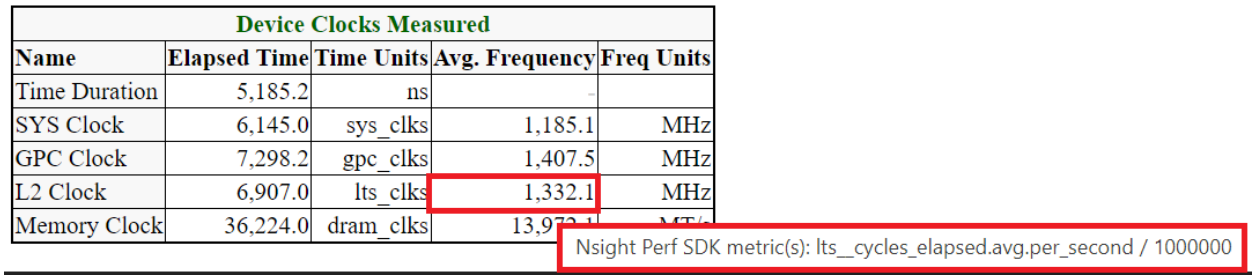
Expect percentages to be in the range 0 - 105% in typical runs, where the inherent variability of frame execution causes minor inconsistencies across measurements. Related topic - ensuring stable measurements.

HTML Report

The HTML reports are designed for fast performance triage, in a top-down fashion.

Interpreting the Tables

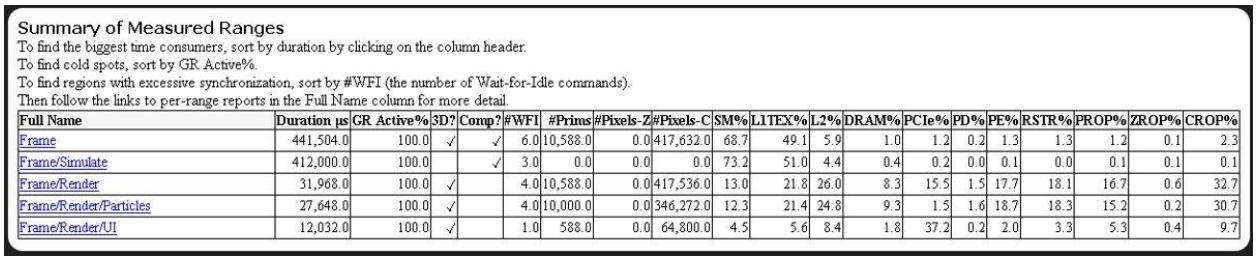
To identify the metrics associated with a specific cell in the table, simply hover over the cell of interest. A tooltip will appear, revealing the underlying metric formula. If you are developing your own collector, rather than utilizing the HTML report generator, this feature enables you to selectively choose relevant metrics from the HTML report for integration into your collector.



Device Clocks Measured				
Name	**Elapsed Time**	**Time Units**	**Avg. Frequency**	**Freq Units**
Time Duration	5,185.2	ns	-	
SYS Clock	6,145.0	sys_clks	1,185.1	MHz
GPC Clock	7,298.2	gpc_clks	1,407.5	MHz
L2 Clock	6,907.0	lts_clks	1,332.1	MHz
Memory Clock	36,224.0	dram_clks	13,972.1	MT/s

Summary Page

Every report directory contains a `summary.html` page. This is the starting point for analysis. `summary.html` is the only predictably named HTML file, if you wish to generate a link from another page into a report directory.



Summary of Measured Ranges

To find the biggest time consumers, sort by duration by clicking on the column header.
To find cold spots, sort by GR Active%.
To find regions with excessive synchronization, sort by #WFI (the number of Wait-for-Idle commands).
Then follow the links to per-range reports in the Full Name column for more detail.

Full Name	Duration µs	GR Active%	3D?	Comp?	#WFI	#Prims	#Pixels-Z	#Pixels-C	SM%	L1TEX%	L2%	DRAM%	PCIe%	PD%	PE%	RSTR%	PROP%	ZROP%	CROP%
Frame	441,504.0	100.0	✓	✓	6.0	10,588.0	0.0	417,632.0	68.7	49.1	5.9	1.0	1.2	0.2	1.3	1.3	1.2	0.1	2.3
Frame/Simulate	412,000.0	100.0		✓	3.0	0.0	0.0	0.0	73.2	51.0	4.4	0.4	0.2	0.0	0.1	0.0	0.1	0.1	0.1
Frame/Render	31,968.0	100.0	✓		4.0	10,588.0	0.0	417,536.0	13.0	21.8	26.0	8.3	15.5	1.5	17.7	18.1	16.7	0.6	32.7
Frame/Render/Particles	27,648.0	100.0	✓		4.0	10,000.0	0.0	346,272.0	12.3	21.4	24.8	9.3	1.5	1.6	18.7	18.3	15.2	0.2	30.7
Frame/Render/UI	12,032.0	100.0	✓		1.0	588.0	0.0	64,800.0	4.5	5.6	8.4	1.8	37.2	0.2	2.0	3.3	5.3	0.4	9.7

Click on any column header to sort by it.
Click on the link in the Full Name column to navigate to the corresponding per-range report.

Useful sort operations:

- Sort by Full Name to return to the natural order at any time. (“depth first sort”)
- Sort by Duration to find the biggest time consumers.
- Sort by GR Active% to find cold spots; but make sure the duration is significant.
- Sort by Comp? to find all ranges containing compute dispatches.
- Sort by WFI to find ranges with large amounts of synchronization. In D3D12 and Vulkan, these are typically heavyweight ResourceBarriers.

- Sort by Prims to find regions with high amounts of geometry sent from vertex buffers.
- Sort by Pixels-Z to find regions with fixed-function depth processing.
- Sort by Pixels-C to find regions with fixed-function color processing.

Troubleshooting

If this page is empty, check whether the application contains performance markers, and whether NvPerf PushRange and PopRange commands are being submitted to the GPU.

Per-Range Report

The per-range report organizes hundreds of GPU metrics into concise tables, for fast discovery of performance limiters. At first the report may seem daunting, but the trick is to first scan for the longest blue bars, and then go back and read the labels. Zeros and low values can generally be ignored and skimmed past, as they represent inactivity. Only if everything has a low value, conclude the range is starvation or latency-bound, which can be disambiguated with further inspection.

Otherwise, follow the instructions and links in the report, starting at the top.

The per-range report file has been designed with the following properties:

1. The file is standalone, and can be shared by attaching to emails or copying only the one file into a destination.
2. It can be loaded into a browser from disk without CORS errors.
3. Any portion can be copy/pasted into email or documents, as all contents are textual.
    a. Pro tip: make sure to select past a table extent when copying, as otherwise the browser sometimes does not copy over the table formatting.
4. Everything is searchable in the browser. There is no hidden text.

The following sections describe a few highlights from the report.

Device Section

This section displays static GPU properties, and dynamic time & clock measurements.

Caveat: There is a known issue with the “Time Duration” measuring too long a duration, which may cause the average clock frequencies to underreport. This issue tends to occur in longer ranges where background processes cause the GPU to be time sliced.

Device Properties	
Name	**Value**
Date & Time	Mar 8, 2023 17:40:17
GPU Name	Orin (nvgpu)
Chip Name	GA10B
Clock Locking Status	Unknown
# SMs	8
L2 Cache Size (KiB)	1024

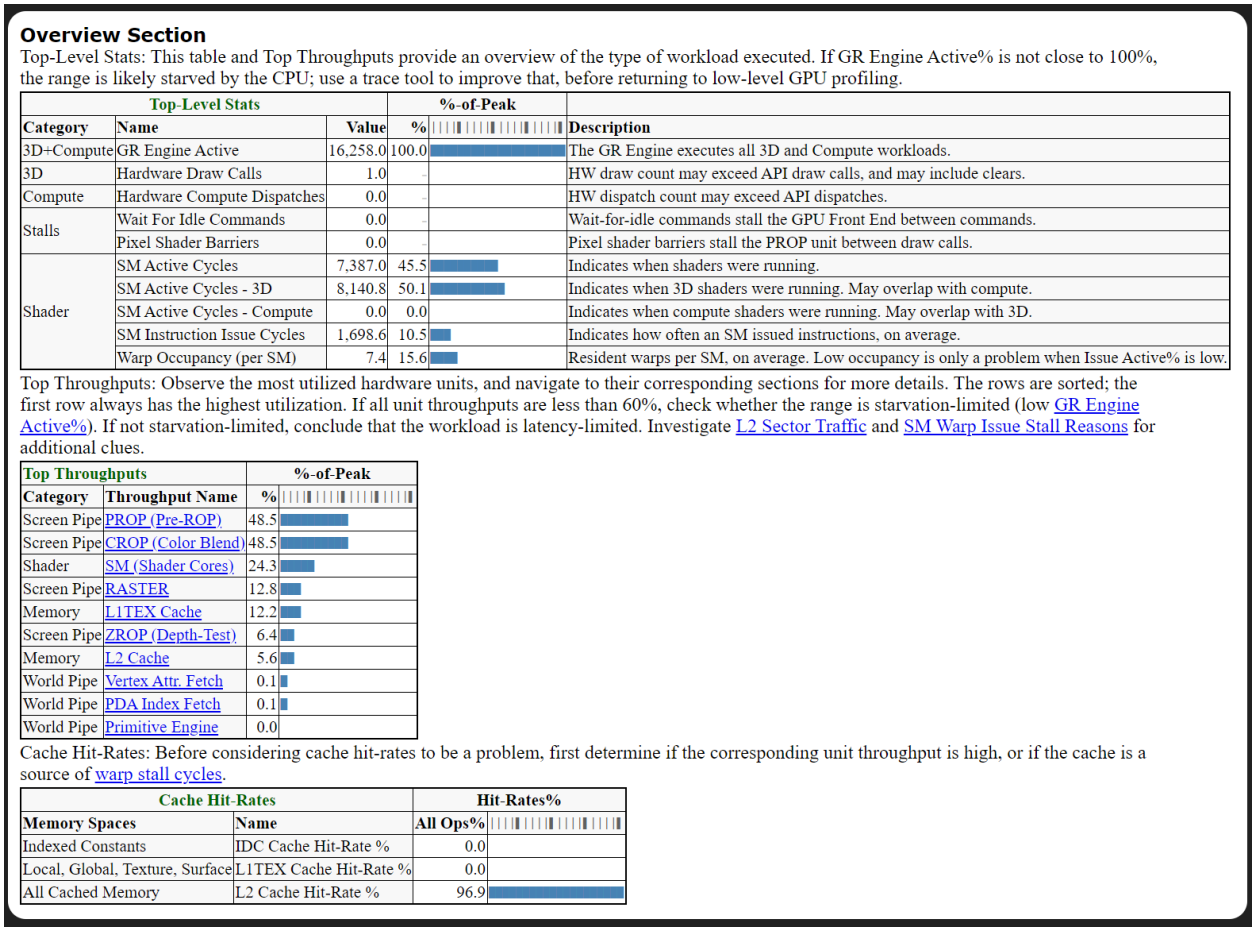
Device Clocks Measured				
Name	**Elapsed Time**	**Time Units**	**Avg. Frequency**	**Freq Units**
Time Duration	17,824.0	ns	-	
SYS Clock	16,258.0	sys_clks	912.1	MHz
GPC Clock	16,258.0	gpc_clks	912.1	MHz
L2 Clock	16,258.0	lts_clks	912.1	MHz

Overview Section

The overview section characterizes the workload, making it quick to determine:

1. Is the GR Engine being utilized, or is there starvation?
2. Was there a large amount of synchronization? (FrontEnd Wait-for-Idle commands)
3. What kind of workload was measured? (SM Active - 3D, Compute)
4. Which pipeline stages were most heavily utilized? (Top Throughputs)

Follow the instructions and links on the page.



Overview Section

Top-Level Stats: This table and Top Throughputs provide an overview of the type of workload executed. If GR Engine Active% is not close to 100%, the range is likely starved by the CPU; use a trace tool to improve that, before returning to low-level GPU profiling.

Top-Level Stats			%-of-Peak	
Category	**Name**	**Value**	**%**	**Description**
3D+Compute	GR Engine Active	16,258.0	100.0	The GR Engine executes all 3D and Compute workloads.
3D	Hardware Draw Calls	1.0		HW draw count may exceed API draw calls, and may include clears.
Compute	Hardware Compute Dispatches	0.0		HW dispatch count may exceed API dispatches.
Stalls	Wait For Idle Commands	0.0		Wait-for-idle commands stall the GPU Front End between commands.
	Pixel Shader Barriers	0.0		Pixel shader barriers stall the PROP unit between draw calls.
Shader	SM Active Cycles	7,387.0	45.5	Indicates when shaders were running.
	SM Active Cycles - 3D	8,140.8	50.1	Indicates when 3D shaders were running. May overlap with compute.
	SM Active Cycles - Compute	0.0	0.0	Indicates when compute shaders were running. May overlap with 3D.
	SM Instruction Issue Cycles	1,698.6	10.5	Indicates how often an SM issued instructions, on average.
	Warp Occupancy (per SM)	7.4	15.6	Resident warps per SM, on average. Low occupancy is only a problem when Issue Active% is low.

Top Throughputs: Observe the most utilized hardware units, and navigate to their corresponding sections for more details. The rows are sorted; the first row always has the highest utilization. If all unit throughputs are less than 60%, check whether the range is starvation-limited (low GR Engine Active%). If not starvation-limited, conclude that the workload is latency-limited. Investigate L2 Sector Traffic and SM Warp Issue Stall Reasons for additional clues.

Top Throughputs		%-of-Peak
Category	**Throughput Name**	**%**
Screen Pipe	PROP (Pre-ROP)	48.5
Screen Pipe	CROP (Color Blend)	48.5
Shader	SM (Shader Cores)	24.3
Screen Pipe	RASTER	12.8
Memory	L1TEX Cache	12.2
Screen Pipe	ZROP (Depth-Test)	6.4
Memory	L2 Cache	5.6
World Pipe	Vertex Attr. Fetch	0.1
World Pipe	PDA Index Fetch	0.1
World Pipe	Primitive Engine	0.0

Cache Hit-Rates: Before considering cache hit-rates to be a problem, first determine if the corresponding unit throughput is high, or if the cache is a source of warp stall cycles.

Cache Hit-Rates		Hit-Rates%
Memory Spaces	**Name**	**All Ops%**
Indexed Constants	IDC Cache Hit-Rate %	0.0
Local, Global, Texture, Surface	L1TEX Cache Hit-Rate %	0.0
All Cached Memory	L2 Cache Hit-Rate %	96.9

Memory Performance Section

If the Top Throughputs section indicates that L2 Cache utilization is high, then look at the L2 Sector Traffic tables. Simply look for the largest blue bars, to pinpoint the sources, destinations, and operation types incurring the most memory bandwidth. These tables are organized in a tree-like fashion to show a hierarchical breakdown.

L2 Sector Traffic by Memory Aperture		%-of-Sectors		Op		%-of-Sectors	
To Memory	Hit-Rate%	%		Op	Hit-Rate%	%	
System Memory	96.9	99.8		Reads	68.5	9.9	
				Writes	100.0	89.0	
				Atomics	0.0	0.0	
				Reductions	0.0	0.0	
Peer Memory	0.0	0.0					

L2 Sector Traffic by Source Breakdown		%-of-Sectors		Unit Breakdown		%-of-Sectors		Memory Aperture		%-of-Sectors		Op		%-of-Sectors	
From Source	Hit-Rate%	%		From Unit	Hit-Rate%	%		To Memory	Hit-Rate%	%		Op	Hit-Rate%	%	
GPC Units	96.9	98.1		L1TEX Cache	0.0	0.0									
				L1.5 Constant Cache	0.0	3.1		Peer Memory	0.0	0.0					
								System Memory	0.0	3.1		Reads	0.0	3.1	
				Primitive Engine	97.3	1.5		Peer Memory	0.0	0.0					
								System Memory	97.3	1.5		Reads	97.0	1.4	
												Writes	100.0	0.1	
				Raster	100.0	0.3		Peer Memory	0.0	0.0					
								System Memory	100.0	0.3		Reads	100.0	0.3	
												Writes	0.0	0.0	
				ZROP	100.0	30.8		Peer Memory	0.0	0.0					
								System Memory	100.0	30.8		Reads	100.0	2.6	
												Writes	100.0	28.2	
				CROP	100.0	62.4		Peer Memory	0.0	0.0					
								System Memory	100.0	62.4		Reads	100.0	2.6	
												Writes	100.0	59.8	
FBP Units	0.0	0.0													
HUB Units	99.3	2.0		all HUB Units	99.3	2.0		Peer Memory	0.0	0.0					
								System Memory	99.3	2.0		Reads	0.0	0.1	
												Writes	100.0	1.1	

If the L2 Sector Traffic table shows “from L1TEX Cache” as the topmost contributor, then inspect the L1TEX Sector Traffic table (below). Start at the right side, looking for long bars, then work your way to the left. This table is also a tree, showing the decomposition of traffic across (Pipeline, Memory Space, Operation). Knowing which Memory Space + Operation is incurring the most bandwidth can help to narrow down the lines of shader code that are limiting performance.

- Local Load and Local Store imply either heavy register pressure (too many live variables), or dynamically indexed thread-local arrays.
- The other operations map directly to shader-visible constructs.

Note that compute-shared memory and 3D attributes are not represented in the table, because they are not “tagged” memories. See the (L1TEX Throughput, LSU column, Data Cycles row) table cell for an indication of those memory spaces’ impact on performance; that cell includes (local + global + shared + 3D attributes).

L1TEX Sector Traffic: Determine which shader memory spaces and operations are incurring the most L1TEX sector bandwidth. Hit-rates are calculated per pipe/memory/op combination (both numerator and denominator are specific to the pipe/memory/op). Low hit-rates are only a problem if the corresponding %-of-Sectors is high. Texture traffic is decomposed in two ways: by memory, and by surface format. All %-of-Sectors values are relative to the total sectors processed by the L1TEX cache. The sum of "LSU by Mem" and "TEX by Mem" will add up to 100% in each column.

L1TEX Sector Traffic				Per-Memory Space %-of-Total				Per-Op %-of-total			
Pipe Breakdown	Hit-Rate%	%-of-Sectors		Memory Space	Hit-Rate%	%-of-Sectors		Op	Hit-Rate%	%-of-Sectors	
LSU by Mem	0.0	0.0		Global	0.0	0.0		Global Load	0.0	0.0	
								Global Store	0.0	0.0	
								Global Atom	0.0	0.0	
								Global Red	0.0	0.0	
				Local	0.0	0.0		Local Load	0.0	0.0	
								Local Store	0.0	0.0	
TEX by Mem	0.0	0.0		Texture	0.0	0.0		Texture Fetch	0.0	0.0	
								Texture Load	0.0	0.0	
				Surface	0.0	0.0		Surface Load	0.0	0.0	
								Surface Store	0.0	0.0	
								Surface Atom	0.0	0.0	
								Surface Red	0.0	0.0	
TEX by Format	0.0	0.0		1D Buffer	0.0	0.0					
				1D or 2D Tex/Surf	0.0	0.0					
				2D Tex/Surf, no Mipmaps	0.0	0.0					
				3D Tex/Surf	0.0	0.0					
				Cubemap	0.0	0.0					

Caveats

Reports generated from Queue-level PushRange/PopRange commands will have inherent variability, and may show some out-of-range percentage values (exceeding 100%). This is more typical of small workloads, where the noise introduced by queue-level work submission is a significant fraction of the workload's execution time. Related topic - ensuring stable measurements.

Due to hardware limitations, some counters have inherent variability while measuring small workloads. For example, if a workload does not execute on all SMs (shader cores), the frame-to-frame work distribution within the GPU will vary, potentially impacting the counter's quality. The other example is the out-of-range "Hit-Rate%" because the metrics which collect the "hit" and "total" access counts may be collected in different frames. In that case, the "total" could be smaller than "hit" and leads to a >100% and even an infinite Hit-Rate%. This problem diminishes on bigger workloads.

Expect percentages to be in the range 0 - 105% in typical runs, where the inherent variability of frame execution causes minor inconsistencies across measurements. Related topic - ensuring stable measurements.

Enhancing Diagnostic Capabilities

By setting `ReportOutputOptions::writeCounterConfigImage` and `ReportOutputOptions::writeCounterDataImage`(in NvPerfReportGenerator.h) to true programmatically, for instance, in Samples\D3D12\D3D12HelloTriangle_HtmlReport\D3D12HelloTriangle.cpp

```
m_nvperf.outputOptions.directoryName = "HtmlReports\...";
m_nvperf.outputOptions.writeCounterConfigImage = true;
```

```
m_nvperf.outputOptions.writeCounterDataImage = true;
```

, two primary Range Profiler binaries will be generated along with the HTML files: counter config image and counter data image. The former specifies which counters are to be collected, while the latter stores all collected counters. When submitting a bug report, providing both the binaries will greatly simplify the diagnostic procedures.

Notice

The information provided in this specification is believed to be accurate and reliable as of the date provided. However, NVIDIA Corporation (“NVIDIA”) does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This publication supersedes and replaces all other specifications for the product that may have been previously supplied.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and other changes to this specification, at any time and/or to discontinue any product or service without notice. Customer should obtain the latest relevant specification before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer. NVIDIA hereby expressly objects to applying any customer general terms and conditions with regard to the purchase of the NVIDIA product referenced in this specification.

NVIDIA products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on these specifications will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this specification. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this specification, or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this specification. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA. Reproduction of information in this specification is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

Trademarks

NVIDIA, the NVIDIA logo, Jetson, Jetson Orin, NVIDIA AGX, NVIDIA DRIVE and NVIDIA Orin are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright