NVIDIA® NSIGHT™ PERF SDK Getting Started Guide

2022.3

Document Revision 18

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 Platforms

- Windows® 10 x86_64
- Windows® 11 x86_64
- Linux x86_64 (Ubuntu Desktop 20.04 or similar distribution)

Supported Graphics API

- DirectX® 12
- Vulkan®
- DirectX® 11
- OpenGL®

Supported GPU Architectures

- NVIDIA Ada AD10x Architecture
- NVIDIA Ampere GA10x Architecture
- NVIDIA Turing TU10x Architecture
- NVIDIA Turing TU11x Architecture

Refer to the release notes for more information on supported drivers, and listings of supported GeForce, RTX, Quadro, and Data Center GPUs. Release notes can be found in the NvPerf/doc subdirectory of the SDK package.

Table of Contents

First Time Use

System Setup

Enable GPU profiling permissions by following the instructions here:
https://developer.nvidia.com/ERR_NVGPUCTRPERM
These steps can be performed at any time. See Profiling Permissions if you encounter trouble.

Unzip the SDK Package

Windows: unzip NVIDIA_Nsight_Perf_SDK_2022.3_[Public or Pro]_Windows.zip
Linux: tar xvzf NVIDIA_Nsight_Perf_SDK_2022.3_[Public or Pro]_Linux.tar.gz

You should see the following directory structure:

- `NvPerf`
  - `bin`
  - `doc`
  - `Include`
- `redist`
  - `include`
- `Samples`
  - `cmake`
  - `D3D11`
  - `D3D12`
  - `NvPerfUtility`
  - `OGL`
  - `Tests`
  - `Vulkan`

Proceed with building and running the SDK samples.

D3D12 Samples

The D3D12 Samples are instrumented versions of the well-known Microsoft DirectX Graphics Samples. To find profiler modifications in the sample source code, search for `NV_PERF_ENABLE_INSTRUMENTATION` or `m_nvperf`, or diff against the original repository.

The D3D12 samples have the following additional system requirements, from the original repo:

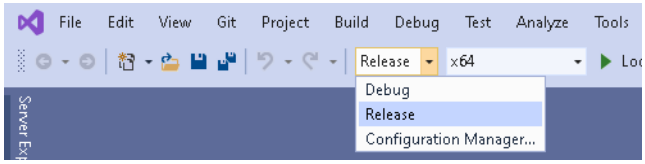
- Windows 10 version 2004 (no new features were added in version 20H2)
- Visual Studio 2019 with the Windows 10 SDK version 2004(19041)

Note that the Nsight Perf SDK works on any Windows 10/11 version where an NVIDIA driver is supported, and is compatible with any C++11-compliant development environment.

Realtime Perf HUD

Multithreading

1. Navigate to Samples\D3D12\D3D12Multithreading
2. Open D3D12Multithreading.sln in Visual Studio
3. Select the Release configuration



4. Build | Build Solution
5. On successful build, files will appear in Samples\D3D12\D3D12Multithreading\bin\x64\Release
    a. `D3D12Multithreading.exe`
    `nvperf_grfx_host.dll`
    `shaders-mt.hlsl`
    `SquidRoom.bin`
6. Debug | Start Debugging

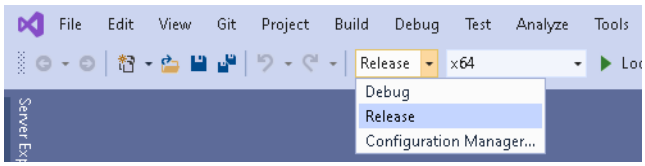
7. Check out the on-screen HUD
8. See the Realtime Perf HUD chapter for more information.

Report Generator

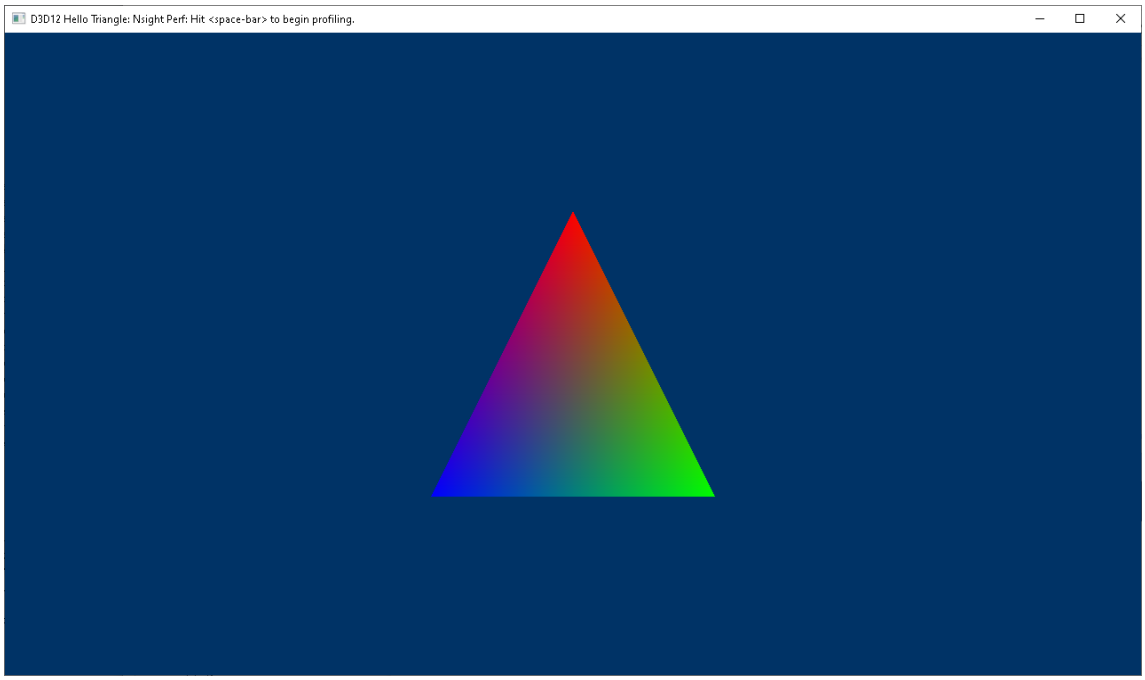
Hello Triangle (3D)

This sample demonstrates Range Profiler counter collection on a 3D Queue.

1. Navigate to Samples\D3D12\D3D12HelloTriangle
2. Open D3D12HelloTriangle.sln in Visual Studio
3. Select the Release configuration



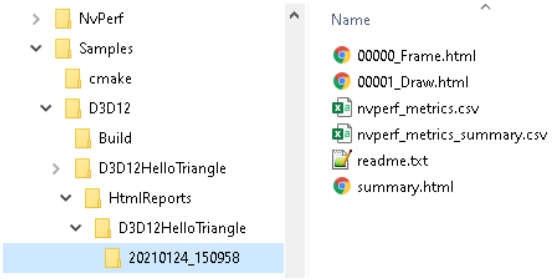
4. Build | Build Solution
5. On successful build, files will appear in Samples\D3D12\D3D12HelloTriangle\bin\x64\Release
    a. `D3D12HelloTriangle.exe`
    `nvperf_grfx_host.dll`
    `shaders.hlsl`
6. Debug | Start Debugging



    a. Notice the title bar contains instructions.
7. **From this step onward: If things don't work as expected, see Troubleshooting.**
8. Hit the **Spacebar** to collect a report.
    a. The title bar should change to the following message.

9. 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. Navigate to Samples\D3D12\D3D12HelloTriangle\HtmlReports\D3D12HelloTriangle
c. There should be a subdirectory with the current date and time. Navigate into it.



d. Note: if you ran the program directly from the executable's directory, the output directory would appear under the current working directory instead.

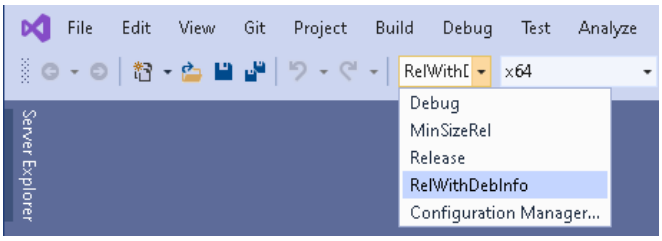
10. Open summary.html for an overview; from there, navigate to per-range reports.
11. See the HTML Report chapter for more information.

Vulkan Samples

The Vulkan Samples are instrumented versions of the well-known Sascha Willems Vulkan Examples. To find profiler modifications in the sample source code, search for `NV_PERF_ENABLE_INSTRUMENTATION` or `m_nvperf`, 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 command prompt or shell at the unzipped package directory location.
3. Windows:
    a. From a command prompt:
    ```
    cd Samples/Vulkan/SaschaWillems
    mkdir build
    cd build
    cmake .. -G "Visual Studio 16 2019"
    ```
    b. Build from Visual Studio:
        i. Open vulkanExamples.sln
        ii. Change configuration to RelWithDebInfo



        iii. Build | Build Solution

    c. Build from Commandline:
        i. `cmake --build . --config RelWithDebInfo`
    d. On successful build, executables will appear in Samples\Vulkan\SaschaWillems\build\bin\RelWithDebInfo
    `computenbody.exe`
    `nvperf_grfx_host.dll`
    `triangle.exe`
4. Linux:
    a. Install libvulkan-dev.
    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`
5. Note: if the above don't work, follow instructions at cmake-generators.

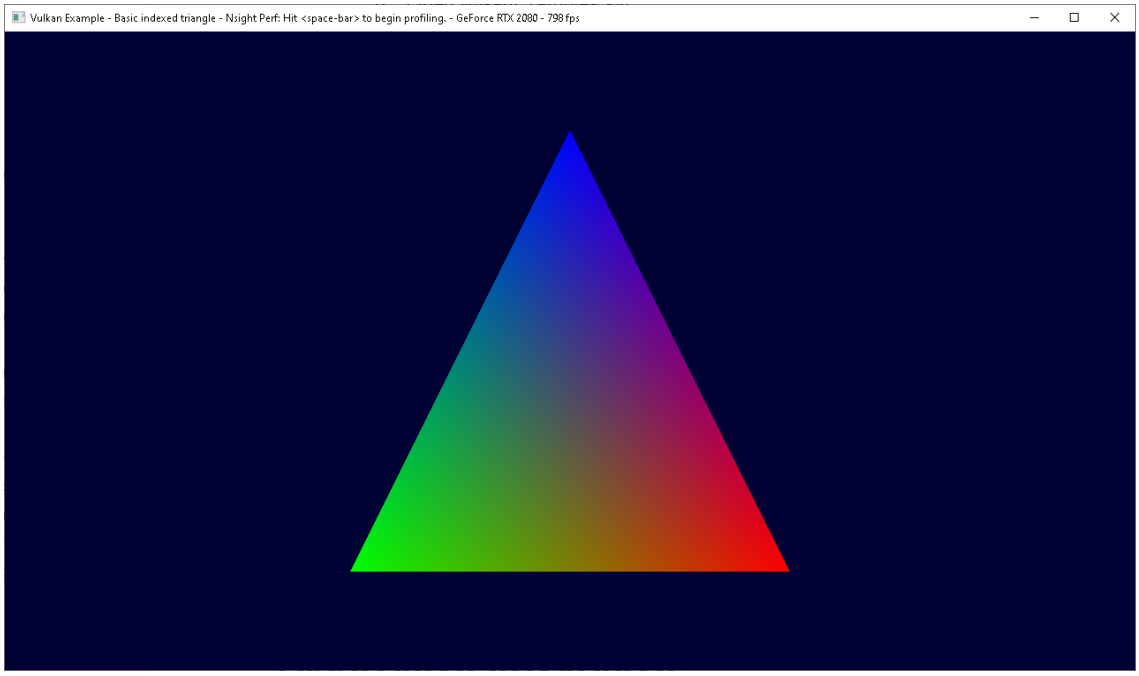
For reference: the original build instructions can be found here.

Report Generator

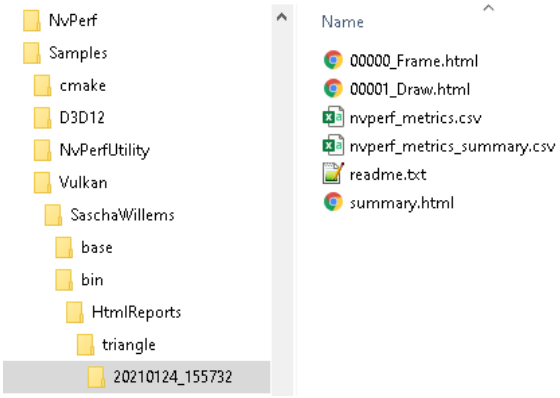
Triangle (3D)

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):



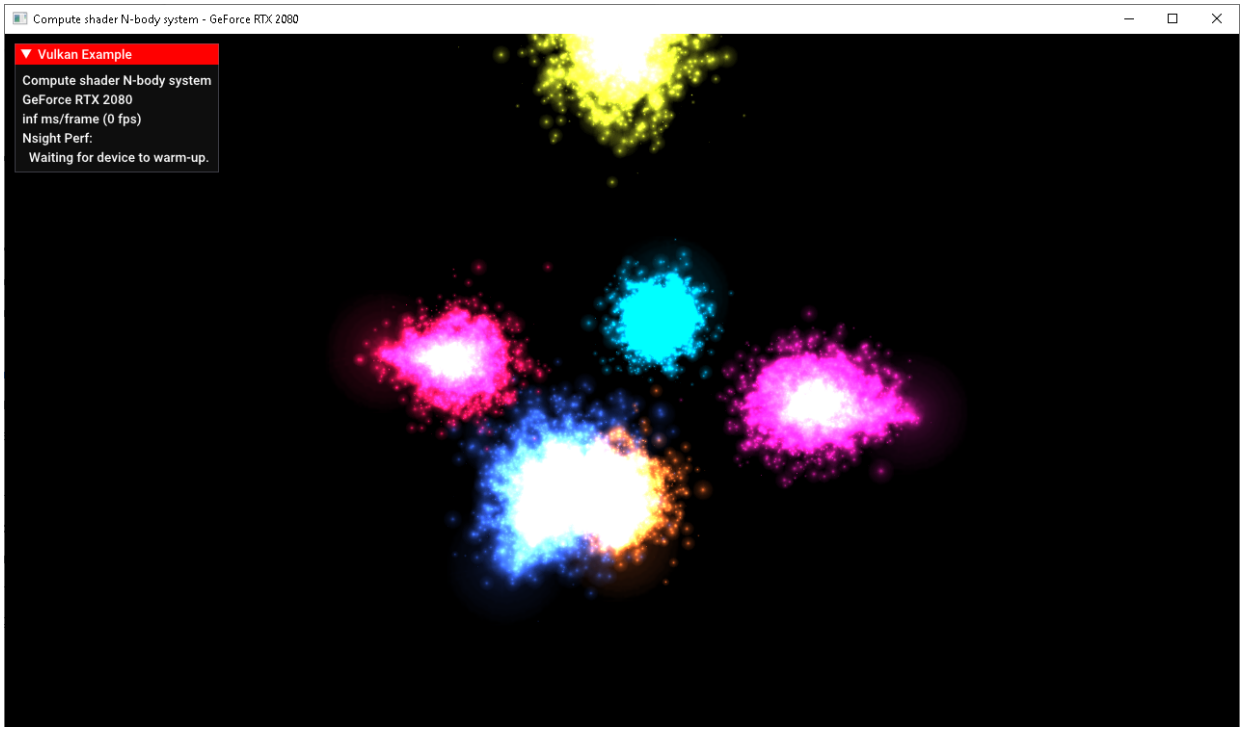
3. **From this step onward: If things don't work as expected, see Troubleshooting.**
4. Hit the **Spacebar** to collect a report.
    a. The title bar should change to say "Currently profiling the frame".
5. When the title bar returns to stating "Hit <space-bar>, it means an HTML profiler report has been written to disk.
    a. You may close the sample program now.
    b. If you ran from Visual Studio: navigate to Samples\Vulkan\SaschaWillems\bin\HtmlReports\triangle
    c. Otherwise, in the current working directory, navigate to ./HtmlReports/triangle
    d. 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

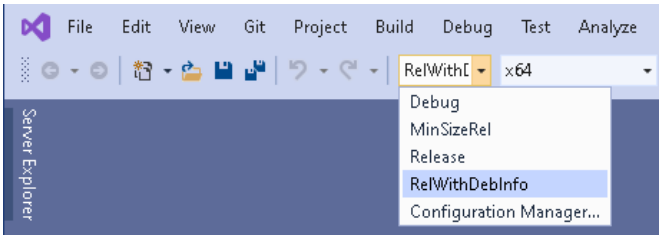
Follow the same procedure as Triangle (3D).
This sample has a built-in HUD that provides instructions and indicates status.



Building the NvPro Ray Tracing Samples

1. Acquire a copy of CMake
    a. Minimum Required Version: 3.11
2. Install Vulkan SDK from the website https://vulkan.lunarg.com/sdk/home. Unlike other vulkan samples, the ray tracing samples require Vulkan SDK version 1.2.170 or above for ray tracing headers and tools.
3. Open a command prompt or shell at the unzipped package directory location.
4. Windows:
    a. From a command prompt:
    ```
    cd Samples/Vulkan/nvpro-samples/vk_raytracing_tutorial_KHR/
    mkdir build
    cd build
    cmake .. -G "Visual Studio 16 2019"
    ```
    b. Build from Visual Studio:
        i. Open vk_raytracing_tutorial.sln

ii. Change configuration to RelWithDebInfo



iii. Build | Build Solution

c. Build from Commandline:
   i. `cmake --build . --config RelWithDebInfo`
d. On successful build, executables will appear in Samples\Vulkan\nvpro-samples\vk_raytracing_tutorial_KHR\build\RelWithDebInfo:

```
nvperf_grfx_host.dll
shared_sources_vk.lib
shared_sources_vk.pdb
vk_ray_tracing_animation_KHR
vk_ray_tracing_animation_KHR.exe
vk_ray_tracing_animation_KHR.ilk
vk_ray_tracing_animation_KHR.pdb
vk_ray_tracing_indirect_scissor_KHR
vk_ray_tracing_indirect_scissor_KHR.exe
vk_ray_tracing_indirect_scissor_KHR.ilk
vk_ray_tracing_indirect_scissor_KHR.pdb
```

5. Linux:
   a. Install libx11-dev, libxrandr-dev, libxinerama-dev, libxcursor-dev, libxi-dev, libgl1-mesa-dev, libxxf86vm-dev
   b. From a shell:

```
cd Samples/Vulkan/nvpro-samples/vk_raytracing_tutorial_KHR
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo
make -j8
```

   c. On successful build, executables will appear in Samples/Vulkan/nvpro-samples/vk_raytracing_tutorial_KHR/build/RelWithDebInfo:

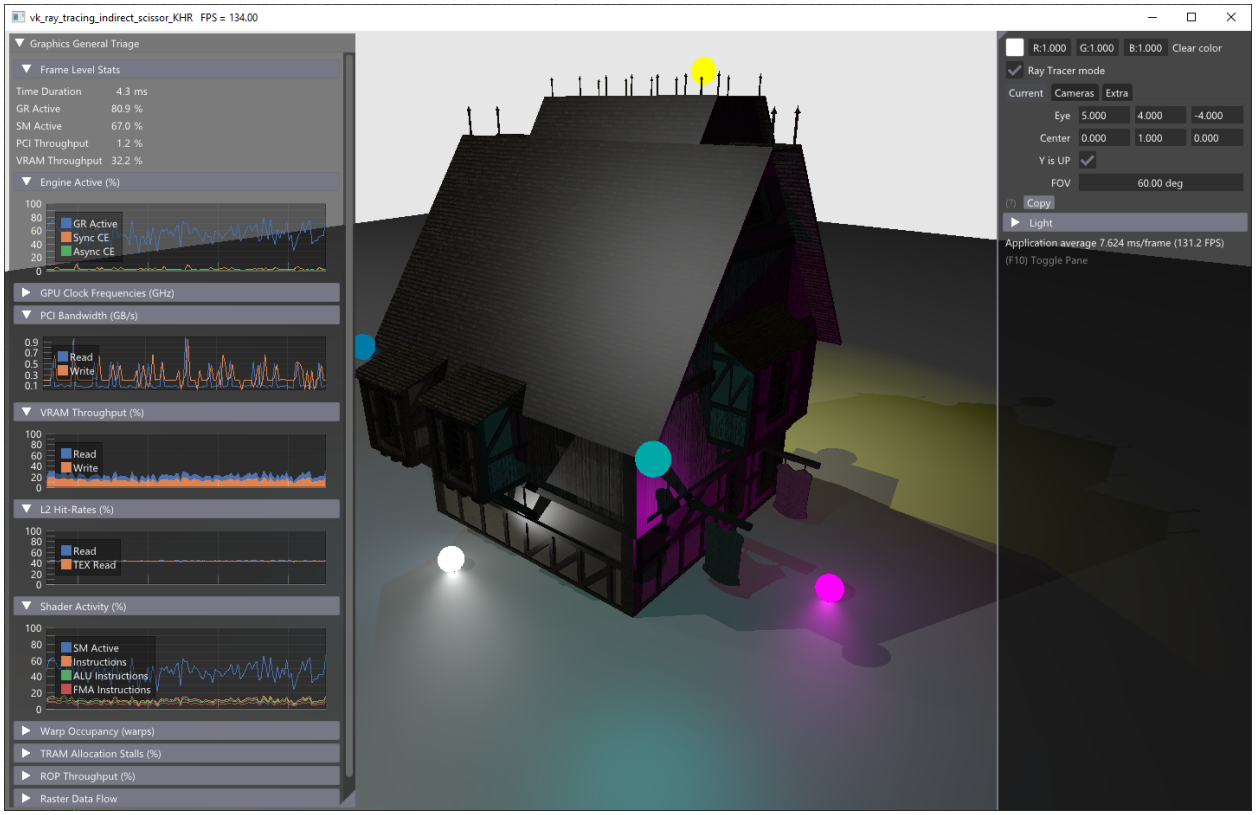
```
libnvperf_grfx_host.so -> libnvperf_grfx_host.so.2022.3.0
libnvperf_grfx_host.so.2022.3.0
libshared_sources_vk.a
vk_ray_tracing_animation_KHR
vk_ray_tracing_animation_KHR.exe
vk_ray_tracing_indirect_scissor_KHR
vk_ray_tracing_indirect_scissor_KHR.exe
```

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

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.



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.

D3D11 Samples

The D3D11 Samples are instrumented versions of the well-known DirectX SDK Samples for D3D11. To find profiler modifications in the sample source code, search for NV_PERF_ENABLE_INSTRUMENTATION or diff against the original repository.

The D3D11 Samples have the following additional system requirements:

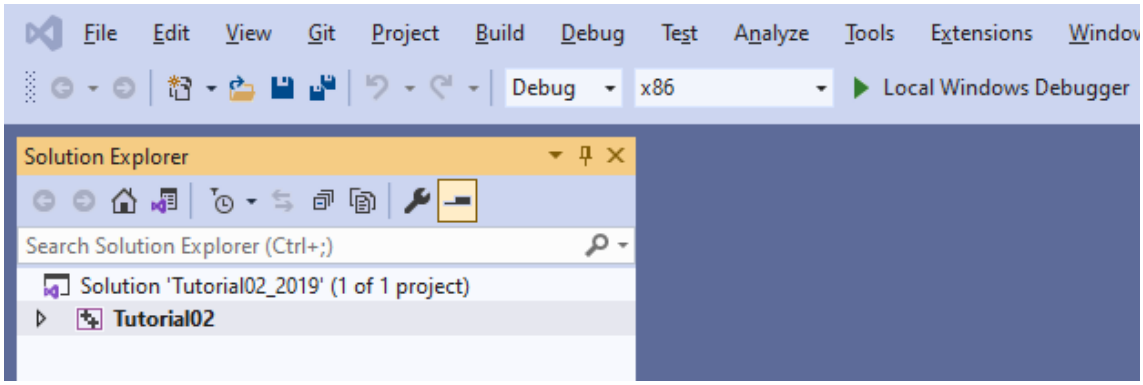
- Visual Studio 2019

Note that the Nsight Perf SDK works on any Windows 10/11 version where an NVIDIA driver is supported, and is compatible with any C++11 compliant development environment.

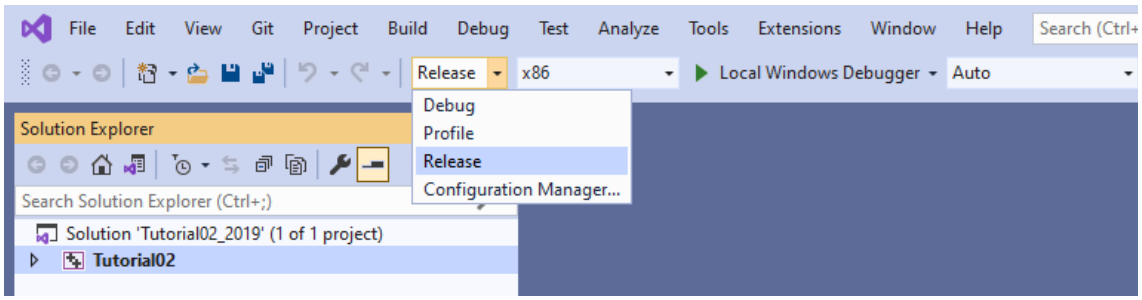
Tutorial02 (Hello Triangle)

This sample demonstrates collection in a 3D graphics context.

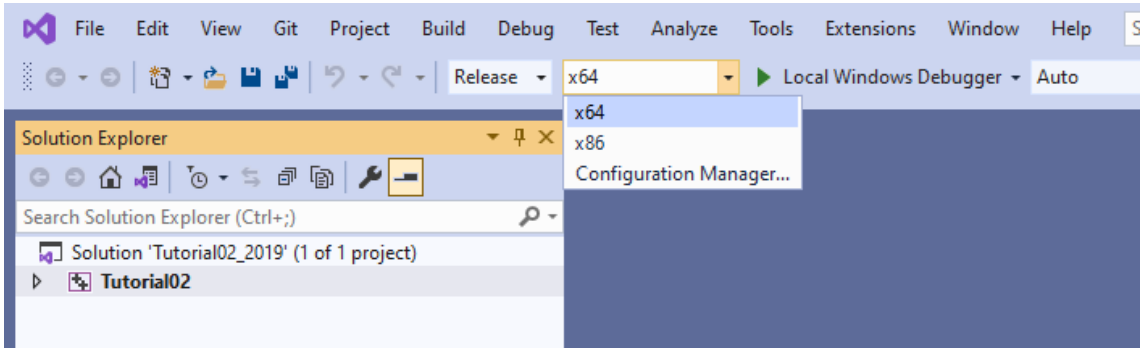
1. Navigate to Samples\D3D11\Direct3D11Tutorials\Tutorial02
2. Open Tutorial02_2019.vcxproj in Visual Studio 2019.
3. Visual Studio will automatically create a solution for this project, which is visible in the Solution Explorer window.



4. Select the Release configuration

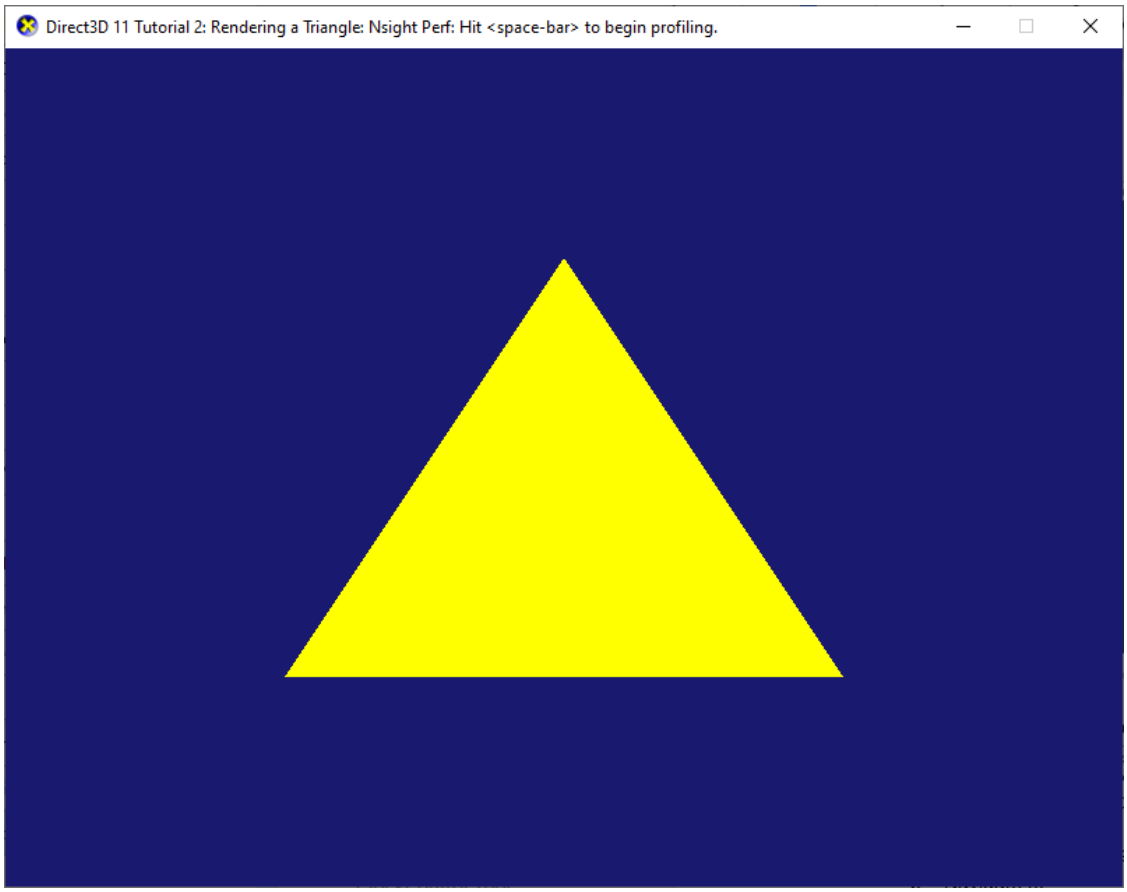


5. (Optionally) Select the x64 configuration

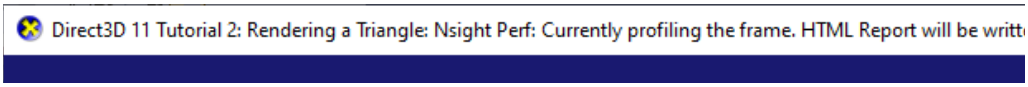


6. Build | Build Solution

7. On successful build, files will appear in
   (x64 builds) Samples\D3D11\Direct3D11Tutorials\Tutorial02\x64\Release
   a. Tutorial02.exe
      nvperf_grfx_host.dll
8. Caution: This sample requires the current working directory to be the project directory:
      Samples\D3D11\Direct3D11Tutorials\Tutorial02
   Running the executable from its own directory will fail.
   The visual studio project is preconfigured with the required working directory.
9. Debug | Start Debugging

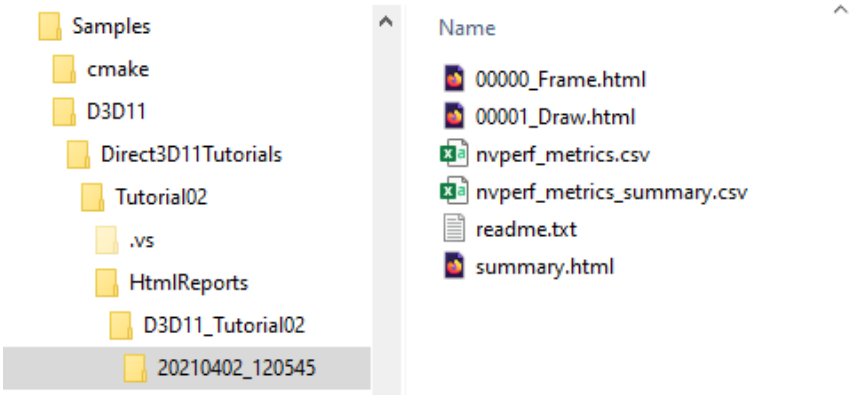


   a. Notice the title bar contains instructions
10. **From this step onward: If things don't work as expected, see Troubleshooting.**
11. Hit the **spacebar** to collect a report
    a. The title bar should change to the following message



12. 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. Navigate to
       Samples\D3D11\Direct3D11Tutorials\Tutorial02\HtmlReports\D3D11_Tutorial02

c. There should be a subdirectory with the current date and time. Navigate into it.



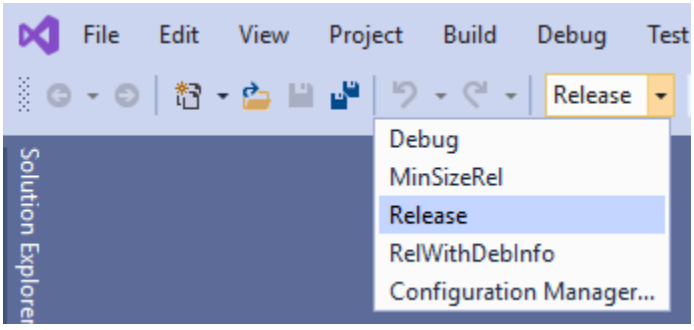
13. Open summary.html for an overview; from there, navigate to per-range reports.
14. See the HTML Report chapter for more information.

OpenGL Samples

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 `m_nvperf`, 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. Windows
    a. From Developer Command Prompt for VS 2019:
    ```
    cd Samples/OGL/gtruc/ogl-samples
    mkdir build
    cd build
    cmake .. -G "Visual Studio 16 2019"
    ```
    b. Build from Visual Studio:
        i. Open ogl-samples.sln
        ii. Change configuration to Release



        iii. Build | Build Solution

c. Build from command prompt:
    i. `cmake --build . --config Release`
d. On successful build, executables will appear in Samples\OGL\gtruc\ogl-samples\build\build\Release
`FreeImage.dll`
`gl-420-draw-base-instance.exe`
`gl-430-program-compute.exe`
`nvperf_grfx_host.dll`

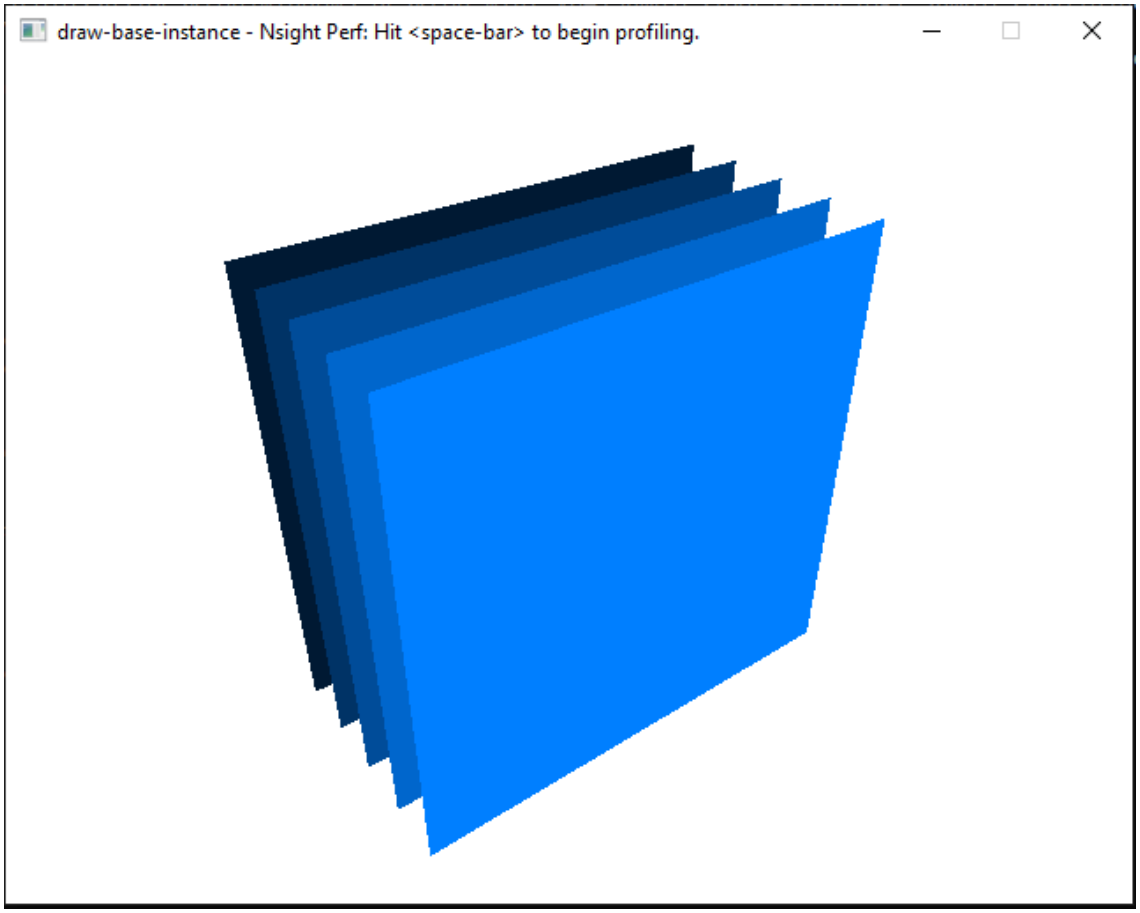
4. Linux
    a. Install these libraries (these names correspond to Ubuntu 20.04):
        i. libglu1-mesa-dev
        ii. libgl1-mesa-dev
        iii. libxrandr-dev
        iv. libxinerama-dev
        v. libxcursor-dev
    b. From a command shell:
    `cd Samples/OGL/gtruc/ogl-samples`
    `mkdir build`
    `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`
6. Note: if the above don't work, follow instructions at cmake-generators.

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

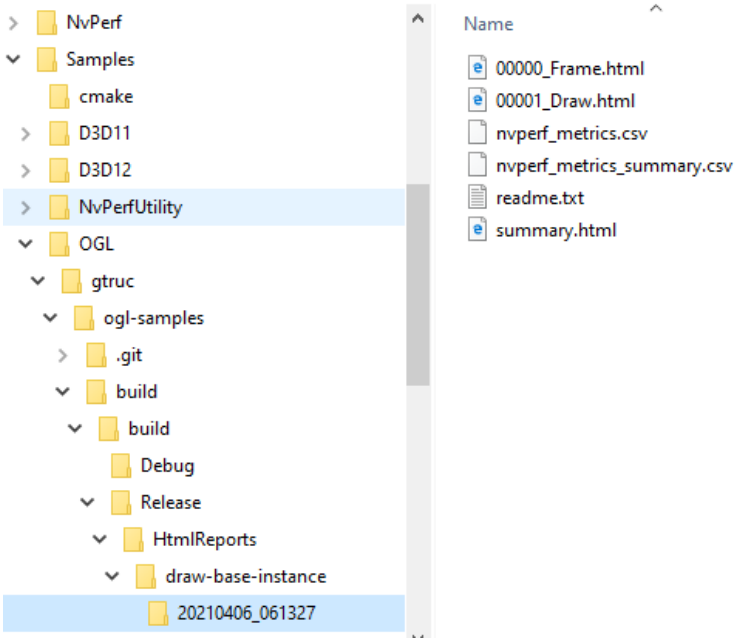
gl-420-draw-base-instance (3D)

1. **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=.`
    `./gl-420-draw-base-instance`

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. If you ran from Visual Studio: navigate to Samples\OGL\gtruc\ogl-samples\build\samples\HtmlReports\draw-base-instance
    c. Otherwise, in the current working directory, navigate to ./HtmlReports/draw-base-instance
    d. 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.

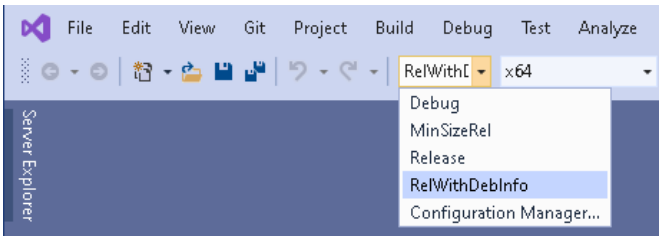
gl-430-program-compute

Follow the same procedure as gl-420-draw-base-instance.

NvPerfUtility Tools

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. Windows:
    a. From a command prompt:
    `cd Samples/NvPerfUtility/tools`
    `mkdir build`
    `cd build`
    `cmake .. -G "Visual Studio 16 2019"`
    b. Build from Visual Studio:
        i. Open NvPerfTools.sln in Visual Studio
        ii. Change configuration to RelWithDebInfo
        

        iii. Build | Build Solution
    c. Build from Commandline:
        i. `cmake --build . --config RelWithDebInfo`
    d. On successful build, executables will appear in Samples/NvPerfUtility/tools/build/bin/RelWithDebInfo:
    `ClockControl.exe`
    `GpuDiag.exe`
    `nvperf_grfx_host.dll`
2. 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
    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`

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

ClockControl Tool

ClockControl can be used to query and modify GPU clock states.

Command	Description
status	displays the current clock setting
lock	locks clocks to stable settings, within the rated power envelope (“Rated TDP”)
unlock	restores the GPU to system-default clock settings

Example output:

```
ClockControl status
[0] GeForce RTX 2080  - NVPW_DEVICE_CLOCK_STATUS_BOOST_ENABLED
ClockControl lock
[0] GeForce RTX 2080  - Locked to rated TDP
ClockControl status
[0] GeForce RTX 2080  - NVPW_DEVICE_CLOCK_STATUS_LOCKED_TO_RATED_TDP
ClockControl unlock
[0] GeForce RTX 2080  - Unlocked
ClockControl status
[0] GeForce RTX 2080  - NVPW_DEVICE_CLOCK_STATUS_BOOST_ENABLED
```

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

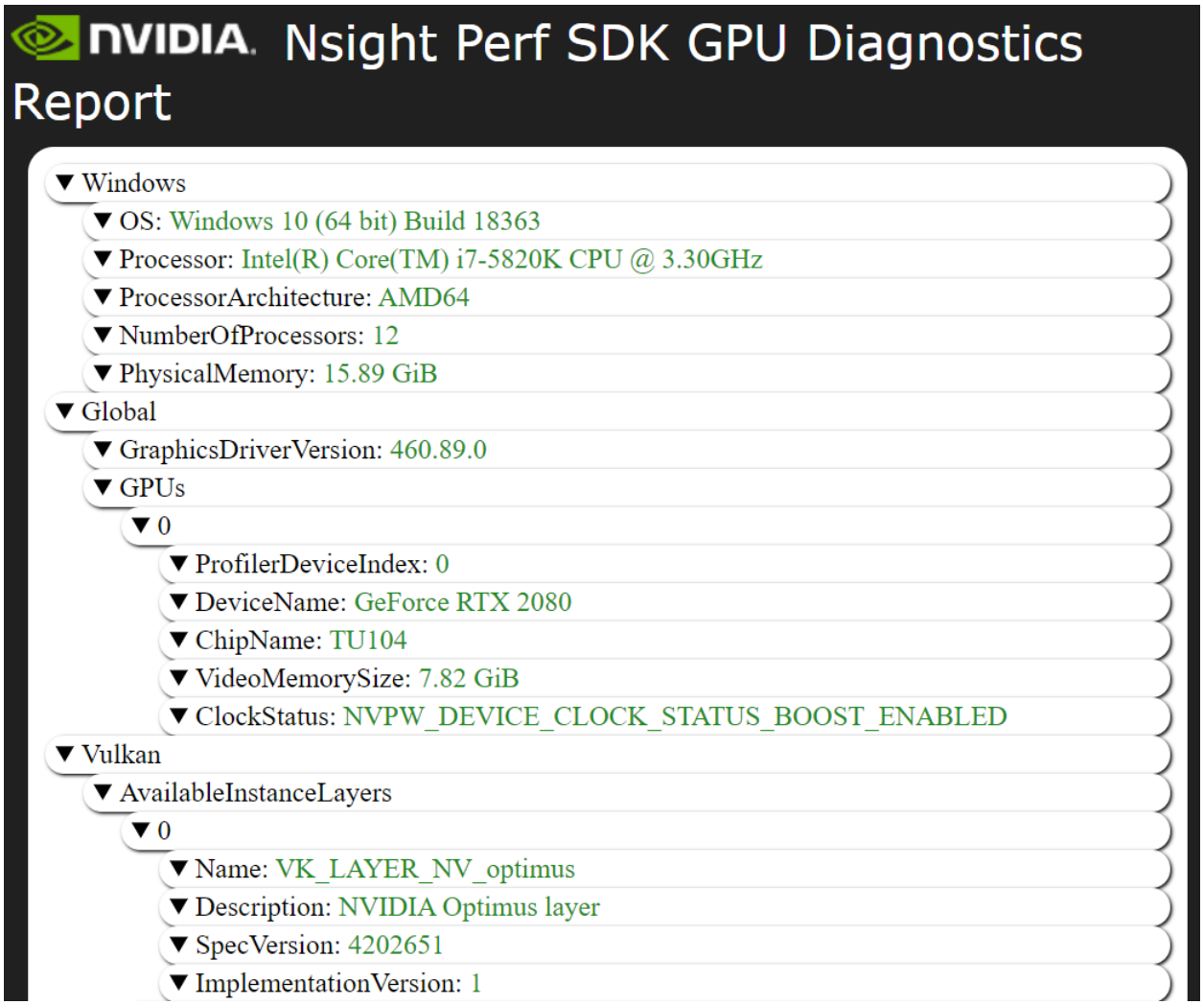
Example output (JSON):

GpuDiag
(console output)


```
{
    "Windows": {
        "OS": "Windows 10 (64 bit) Build 18363",
        "Processor": "Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz",
        "ProcessorArchitecture": "AMD64",
        "NumberOfProcessors": 12,
        "PhysicalMemory": "15.89 GiB"
    },
    "Global": {
        "GraphicsDriverVersion": "460.89.0",
        "GPUs": [
            {
                "ProfilerDeviceIndex": 0,
                "DeviceName": "GeForce RTX 2080",
                "ChipName": "TU104",
                "VideoMemorySize": "7.82 GiB",
                "ClockStatus": "NVPW_DEVICE_CLOCK_STATUS_BOOST_ENABLED"
            }
        ]
    },
    "Vulkan": {
        "AvailableInstanceLayers": [
            {
                "Name": "VK_LAYER_NV_optimus",
                "Description": "NVIDIA Optimus layer",
                "SpecVersion": 4202651,
                "ImplementationVersion": 1
            },
    ...
```

Example output (HTML):

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



The device information indicates whether the underlying GPU is supported for profiling, and whether it is in a usable configuration. If the GPU architecture and configuration is supported, GpuDiag will attempt to launch a profiling session, and report any resultant errors. The following example shows the output when profiling permissions were not granted.

```
..
"Devices": [
  {
      "VKDeviceIndex": 0,
      "Name": "GeForce RTX 2080",
      "Type": "Discrete Gpu",
      "VendorId": 4318,
      "DeviceId": 7815,
      "ApiVersion": "1.2.155",
```

```
                "DeviceUUID": "4e7d-e181-7cf8-85fa-2657-8fcb-a523-9c35",
                "DeviceLUID": "8fbe-3515-0000-0000",
                "DeviceNodeMask": 0,
                "IsNvidiaDevice": true,
                "ProfilerDeviceIndex": 0,
                "ProfilerIsGpuSupported": {
                    "IsSupported": true,
                    "GpuArchitectureSupported": true,
                    "SliSupportLevel": true,
                    "Advice": ""
                },
                "ProfilerIsSessionSupported": {
                    "IsSupported": false,
                    "Advice": "Profiling permissions not enabled. Please follow these
instructions:
https://developer.nvidia.com/nvidia-development-tools-solutions-ERR_NVGPUCTRPERM-permission-
issue-performance-counters"
                },
                "ProfilerRequiredDeviceExtensionsSupported": {
                    "VK_KHR_timeline_semaphore": true
                }
            }
        ]
```

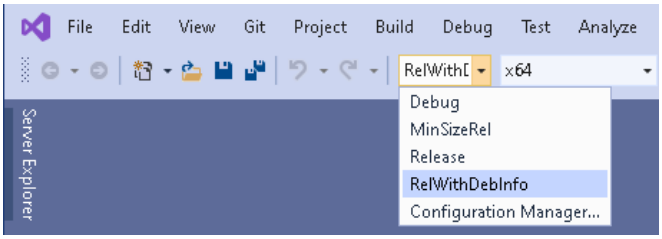

Here it can be seen that the GPU was supported, but profiling permissions were not enabled, which prevented the profiling session from succeeding.

NvPerfUtility Tests

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

To Build

1. Windows:
    b. From a command prompt:
    `cd Samples/Tests/Modules`
    `mkdir build`
    `cd build`
    `cmake .. -G "Visual Studio 16 2019"`
    c. Build from Visual Studio:
        i. Open NvPerfTests.sln in Visual Studio
        ii. Change configuration to RelWithDebInfo



        iii. Build | Build Solution
    d. Build from Commandline:
        i. `cmake --build . --config RelWithDebInfo`
    e. On successful build, executables will appear in `Samples/Tests/Modules/build/bin/RelWithDebInfo:`
    `NvPerfD3D11Test.exe`
    `NvPerfD3D12Test.exe`
    `NvPerfOfflineTest.exe`
    `NvPerfOpenGLTest.exe`
    `NvPerfVulkanTest.exe`
    `nvperf_grfx_host.dll`
2. 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
    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/`:
`NvPerfOfflineTest`
`NvPerfOpenGLTest`
`NvPerfVulkanTest`
`libnvperf_grfx_host.so`

3. 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.

- Windows: by default, errors are sent to OutputDebugString. View them in tools like:
    - Visual Studio Output View, Debug messages
    - SysInternals DebugView
- 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`

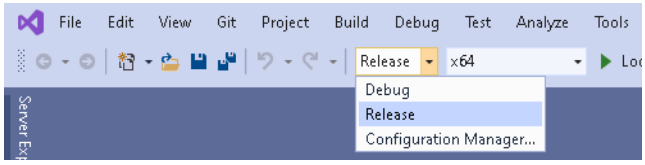
In addition, you can use the GpuDiag tool for quick diagnostics.

Additional Samples

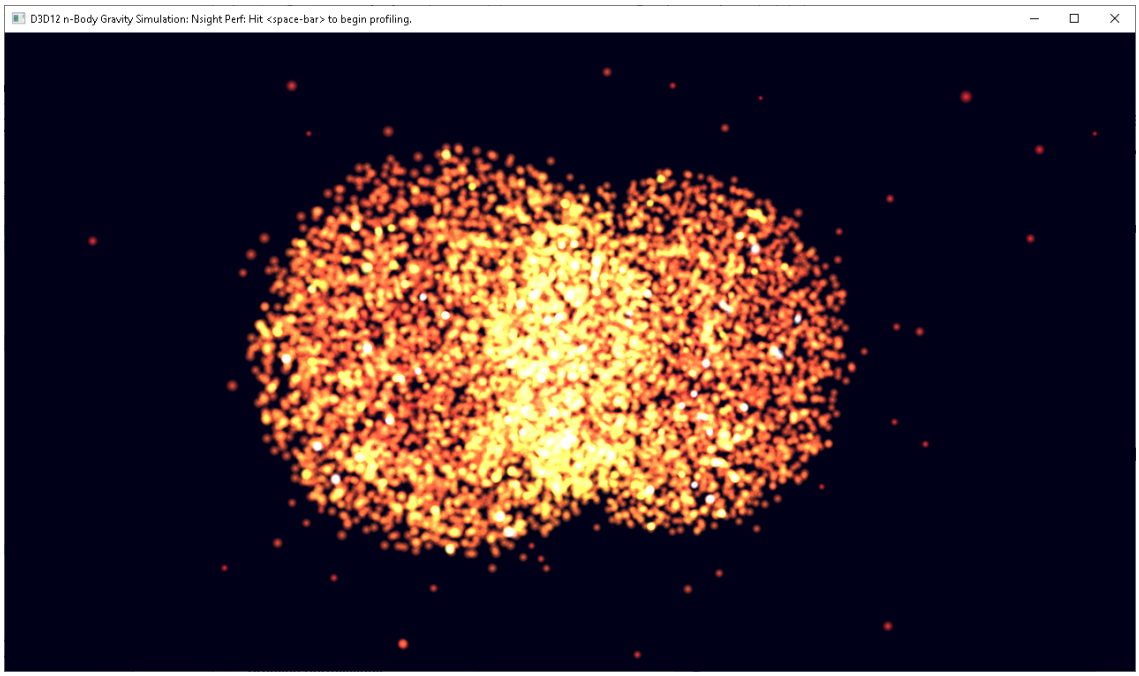
D3D12 Report Generator

nBodyGravity (Async Compute)

This sample demonstrates counter collection on an async compute queue.

1. Navigate to Samples\D3D12\D3D12nBodyGravity
2. Open D3D12nBodyGravity.sln in Visual Studio
3. Select the Release configuration
4. 

5. Build | Build Solution
6. On successful build, files will appear in Samples\D3D12\D3D12HelloTriangle\bin\x64\Release
    a. `D3D12nBodyGravity.exe`
    `nBodyGravityCS.hlsl`
    `nvperf_grfx_host.dll`
    `ParticleDraw.hlsl`
    `WinPixEventRuntime.dll`
7. Debug | Start Debugging

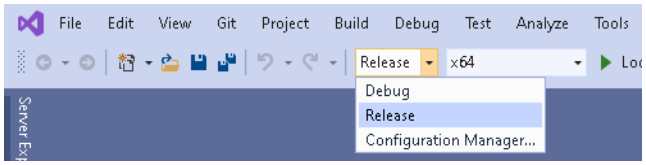


8. **From this step onward: If things don't work as expected, see Troubleshooting.**
9. Hit the **Spacebar** to collect a report.

    a. The title bar should change to say “Currently profiling the frame”.
10. 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. Navigate to Samples\D3D12\D3D12nBodyGravity\HtmlReports\D3D12nBodyGravity
    c. There should be a subdirectory with the current date and time. Navigate into it.
    d. Note: if you ran the program directly from the executable’s directory, the output directory would appear under the current working directory instead.
11. Open summary.html for an overview; from there, navigate to per-range reports.
12. See the HTML Report chapter for more information.

Real Time Denoised Ambient Occlusion (Raytracing)

This sample demonstrates counter collection on acceleration structure updating and ray tracing.

1. Make sure the operating system is Windows 10 version 2004 or above, and the GPU is AD10X, GA10X or TU10X.
2. Navigate to Samples\D3D12\D3D12Raytracing
3. Open D3D12Raytracing.sln in Visual Studio
4. Select the Release configuration
5. 

6. Build | Build Solution
7. On successful build, files will appear in Samples\D3D12\D3D12Raytracing\RTDAO\bin\x64\Release
8. Debug | Start Debugging

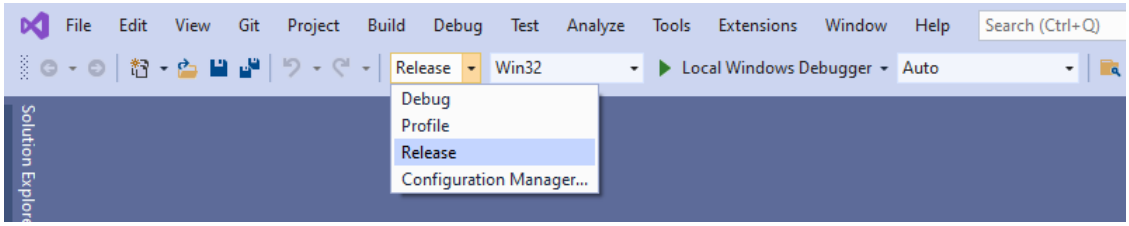
9. **From this step onward: If things don't work as expected, see Troubleshooting.**
10. Hit the **Spacebar** to collect a report.
    a. The title bar should change to say "Currently profiling the frame".
11. When the title bar returns to stating "Hit <space-bar>, it means an HTML profiler report has been written to disk.
    a. You may close the sample program now.
    b. Navigate to Samples\D3D12\D3D12Raytracing\RTDAO\HtmlReports\D3D12RaytracingRealTimeDenoisedAmbientOcclusion
    c. There should be a subdirectory with the current date and time. Navigate into it.
    d. Note: if you ran the program directly from the executable's directory, the output directory would appear under the current working directory instead.
12. Open summary.html for an overview; from there, navigate to per-range reports.
13. See the HTML Report chapter for more information.

D3D11 Report Generator

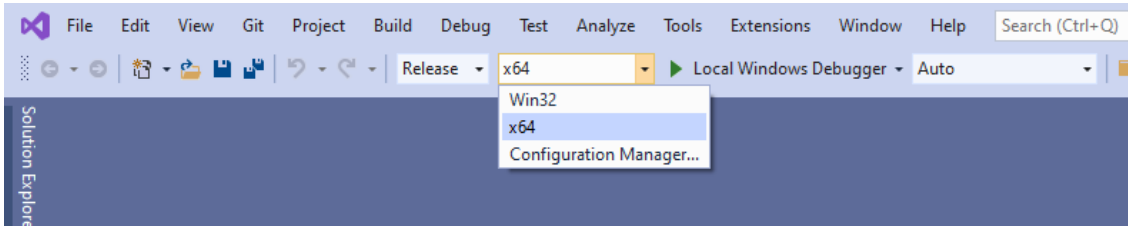
NBodyGravityCS11

This sample demonstrates collection in a compute context.

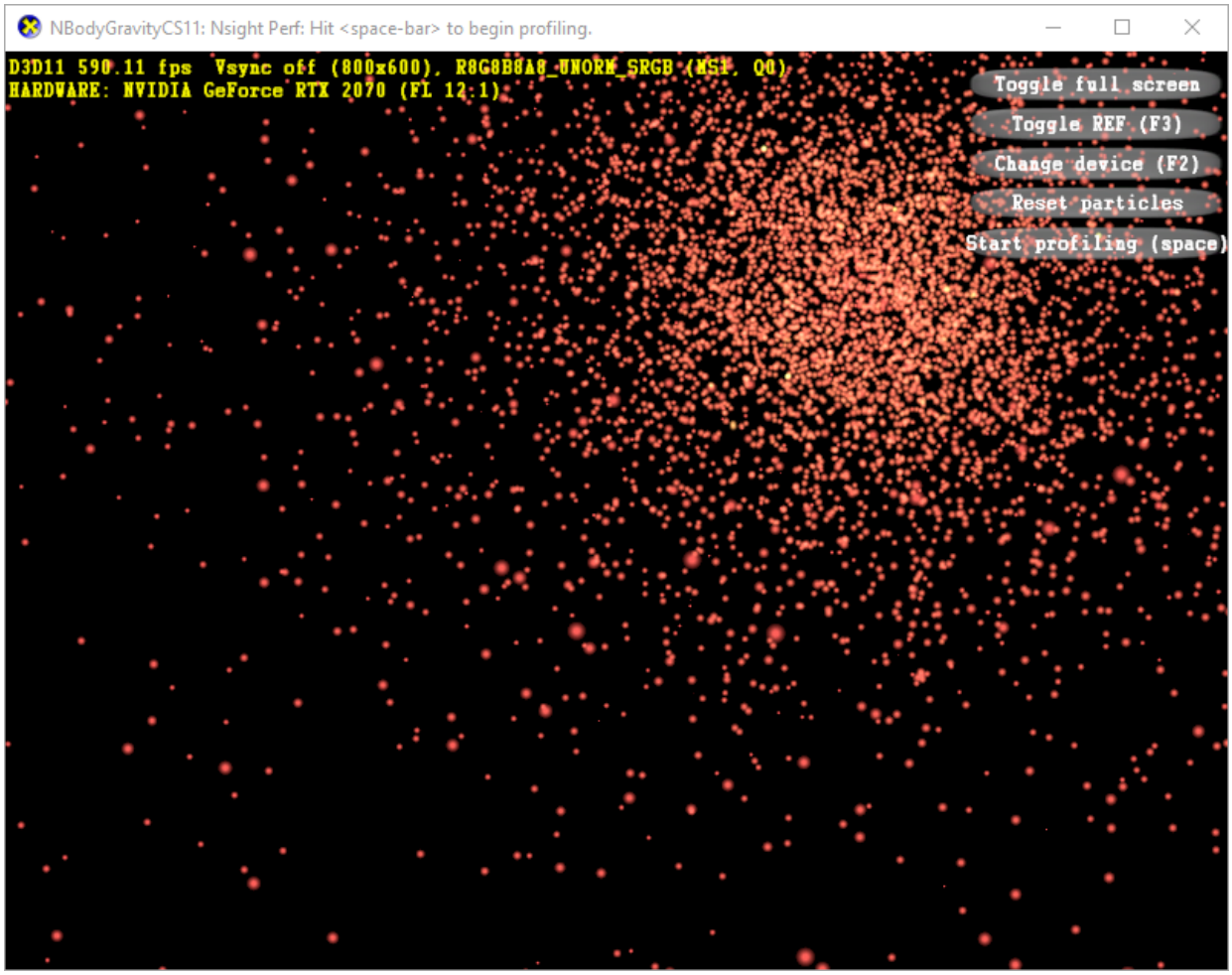
1. Navigate to Samples\D3D11\NBodyGravityCS11
2. Open NBodyGravityCS11_2019_Win10.sln in Visual Studio 2019
3. Select the Release configuration



4. (Optionally) Select the x64 configuration

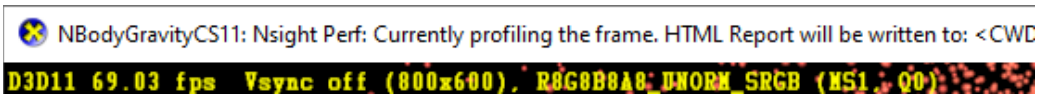


5. Build | Build Solution
6. On successful build, files will appear in (x64 builds) Samples\D3D11\NBodyGravityCS11\x64\Release
    a. NBodyGravityCS11.exe
       nvperf_grfx_host.dll
7. Debug | Start Debugging



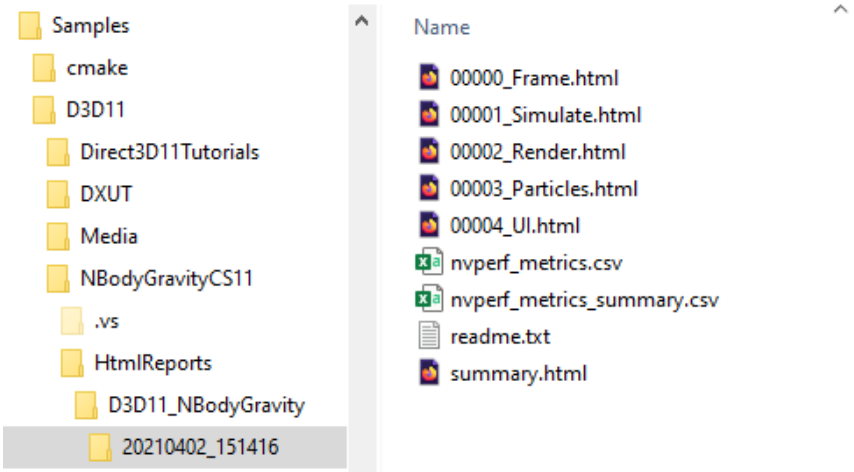
   a. Notice the title bar and on-screen HUD contain instructions

8. Hit the **spacebar** to collect a report
   a. The title bar should change to the following message



9. 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. If you ran from Visual Studio: navigate to Samples\D3D11\NBodyGravityCS11\HtmlReports\D3D11_NBodyGravity
   c. Otherwise, in the current working directory, navigate to HtmlReports\D3D11_NBodyGravity

d. 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.

See the HTML Report chapter for more information.

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 PeriodicSamplerTimeHistoryD3D12 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
- Windows additional paths:
    - NvPerf/include/windows-desktop-x64
- Linux additional paths:
    - NvPerf/include/linux-desktop-x64

NvPerfUtility is a header-only library.

At runtime, only one DLL/DSO is required:

- Windows 64-bit: nvperf_grfx_host.dll
- Linux 64-bit: libnvperf_grfx_host.so

nvperrf_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/windows-desktop-x64/nvperf_grfx_host_impl.h`
- `NvPerf/include/linux-desktop-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.

MSBuild .props file for D3D12

Copy the .props file into your project directory

Samples\NvPerfUtility\buildNvPerfSDK.props

Add this statement to your .vcxproj

```
<Import Project="NvPerfSDK.props" />
```

Add the following to compiler include paths (IncludePath variable):

```
$(NvPerfUtilityPath)/include
$(NvPerfSdkPath)/include
$(NvPerfSdkPath)/include/windows-desktop-x64
```

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.

Additional Dependencies for the Realtime Perf HUD

The core classes of the Realtime Perf HUD depend only on RapidYAML, while the HUD Renderer also depends on ImGui and ImPlot.

MSBuild / Visual Studio

Add the following additional paths (IncludePath variable):

```
$(NvPerfUtilityPath)\imports\rapidyaml-0.4.0
$(NvPerfUtilityPath)\imports\imgui-1.87
$(NvPerfUtilityPath)\imports\imgui-1.87\backends
$(NvPerfUtilityPath)\imports\implot-0.13
```

Use the following format:

```
<ClCompile Include="file" />
```

To add the source files for ImGui and ImPlot:

```
$(NvPerfUtilityPath)\imports\imgui-1.87\backends\imgui_impl_dx12.cpp
$(NvPerfUtilityPath)\imports\imgui-1.87\backends\imgui_impl_win32.cpp
$(NvPerfUtilityPath)\imports\imgui-1.87\imgui.cpp
$(NvPerfUtilityPath)\imports\imgui-1.87\imgui_draw.cpp
$(NvPerfUtilityPath)\imports\imgui-1.87\imgui_tables.cpp
$(NvPerfUtilityPath)\imports\imgui-1.87\imgui_widgets.cpp
$(NvPerfUtilityPath)\imports\implot-0.13\implot.cpp
$(NvPerfUtilityPath)\imports\implot-0.13\implot_items.cpp
```

RapidYAML requires the following statement in one compilation unit / cpp file to add the necessary symbols:

```
#define RYML_SINGLE_HDR_DEFINE_NOW
#include <ryml_all.hpp>
```

CMake

Link the following libraries to your target:

```
NvPerf
NvPerfUtility
NvPerfUtilityImportsImGui
NvPerfUtilityImportsImPlot
NvPerfUtilityImportsRyml
```

RapidYAML requires the following statement in one compilation unit / cpp file to add the necessary symbols:

```
#define RYML_SINGLE_HDR_DEFINE_NOW
#include <ryml_all.hpp>
```

GPU Periodic Sampler, Hud Data Model and HUD Renderer

The GPU Periodic Sampler records performance metrics at a specified frequency without intrumenting workloads of interest. Because it is desirable to also relate the gathered per-sample data to the frames during which they where recorded, the PeriodicSamplerTimeHistoryD3D12 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.

D3D12

State

Declare the following objects next to your ID3D12Device (1:1):

```
nv::perf::sampler::PeriodicSamplerTimeHistoryD3D12 m_sampler;
nv::perf::hud::HudDataModel                        m_hudDataModel;
nv::perf::hud::HudImPlotRenderer                   m_hudRenderer;
```

Initialization

Initialize the sampler any time after ID3D12Device initialization.

```
m_sampler.Initialize(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;
m_sampler.BeginSession(m_commandQueue, 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_ImplWin32_Init(...);
ImGui_ImplDX12_Init(...);
nv::perf::hud::HudImPlotRenderer::SetStyle();
m_hudRenderer.Initialize(m_hudDataModel);
```

Periodic Operations

Samples need 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 requiring a session restart. 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_ImplDX12_NewFrame();
ImGui_ImplWin32_NewFrame();
ImGui::NewFrame();

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

ImGui::Render();
ImGui_ImplDX12_RenderDrawData(ImGui::GetDrawData(), m_commandLists);
```

Clean-Up Operations

HudDataModel, HudImPlotRenderer and PeriodicSamplerTimeHistoryD3D12 can be destroyed through their respective destructors. ImGui and ImPlot require the following tear-down:

```
ImGui_ImplDX12_Shutdown();
ImGui_ImplWin32_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).

Vulkan

The instructions for Vulkan only differ in how to use PeriodicSamplerTimeHistoryVulkan. HudDataModel and HudImPlotRenderer are graphics API agnostic.

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 PeriodicSamplerTimeHistoryD3D12 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

D3D12

ID3D12CommandQueue

- A profiling session can be started on a DIRECT or ASYNC_COMPUTE queue.
- Activity from all queues within the ID3D12Device may be measured.
- Ranges can be pushed and popped on the profiled queue, although this is discouraged.
    - Each queue-level push or pop incurs an ExecuteCommandList call, and will not result in high quality measurements for small workloads.
    - Queue-level push/pop exists for convenience, to collect entire frames, where the overhead of ECL is relatively small.
- 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.

ID3D12CommandList

- 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: Bundles are not supported.

ReportGenerator

State

Declare an instance of class ReportGeneratorD3D12 next to your ID3D12Device (1:1).

```
nv::perf::profiler::ReportGeneratorD3D12 m_nvperf;
```

Initialization

Initialize the report generator any time after ID3D12Device 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.

```
ID3D12Device* pDevice = ...; // from the renderer
m_nvperf.additionalMetrics = { "crop__write_throughput" };
m_nvperf.InitializeReportGenerator(pDevice);
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 Present() call in your application:

```
m_nvperf.OnFrameEnd();
pSwapChain->Present1(...);
m_nvperf.OnFrameStart(pCommandQueue);
```

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

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 ReportGeneratorD3D12 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 D3D12RangeCommands provides a reliable set of function pointers, that are safe to call on any device (supported, unsupported, or non-NVIDIA). The ReportGeneratorD3D12 has a member rangeCommands that is usable after calling `InitializeReportGenerator()`.

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

```
PIXBeginEvent(pCommandList, 0, L"Particle System");
m_nvperf.rangeCommands.PushRange(pCommandList, "Particle System");
pCommandList->DrawInstanced(...);
m_nvperf.rangeCommands.PopRange(pCommandList);
PIXEndEvent(pCommandList);
```

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

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

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. D3D12:
    ```
    pQueue->Signal(pFence, ++fenceValue);
    pFence->SetEventOnCompletion(fenceValue, hEvent);
    WaitForSingleObject(hEvent, 2000);
    ```
    b. 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 to Rated TDP

Force GPU clocks to stable values, by locking to rated TDP (thermal design power).

1. Search for uses of this constant in the sample code, to see example usage: NVPW_DEVICE_CLOCK_SETTING_LOCK_TO_RATED_TDP
2. 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.
3. The clock setting is system global, and is not automatically undone at application exit.
    a. After running a series of profiling sessions, it’s worth explicitly unlocking clocks using the ClockControl tool.
    b. The samples demonstrate a pattern of locking at device creation, and unlocking at device destruction; but abnormal program termination will not execute the device destruction code, causing the clock setting to be “leaked”.
4. Clock frequencies cannot change instantaneously.
    a. After requesting a clock setting change, insert a delay of at least 200 milliseconds, although 500 milliseconds is a safer delay. There is currently no programmatic way to detect whether a clock setting has taken effect in hardware.
    b. In a test framework, it’s worth using the ClockControl command line tool to bracket the entire test suite. This will reduce the testing run time, and avoid the risk of leaking the clock setting, in case an application abnormally terminates.
5. Comparison with ID3D12Device::SetStablePowerState()
    a. SetStablePowerState() is undone at D3D12Device destruction, whereas the NvPerf setting is not (it must be explicitly changed).
    b. SetStablePowerState() requires enabling Windows developer mode, whereas the NvPerf function does not have that requirement.
    c. SetStablePowerState() is reference counted at the system level; if multiple clients call it, the state is not undone until the last D3D12Device is destroyed, whereas the NvPerf setting is not reference counted at the system level.

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.

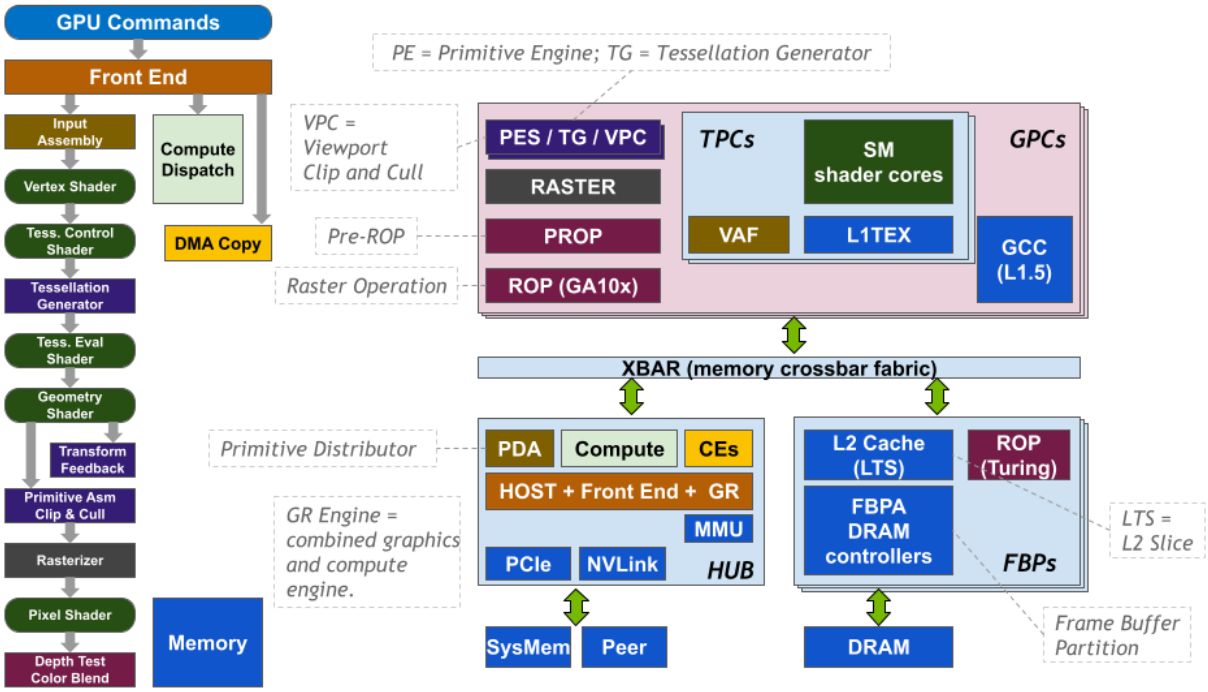
Avoid Running Background Tasks

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__
L1TEX	Combined L1 Data and Texture cache. Directly addressable shared	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__
DRAM	The GPU's onboard DRAM.	dram__
PCIe	PCI Express endpoint controller, which provides access to system memory. "Peer" accesses may also arrive from other peripherals over PCI.	pcie__
NVLINK	NVLINK endpoint controller.	nvlrx__ nvltx__
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__

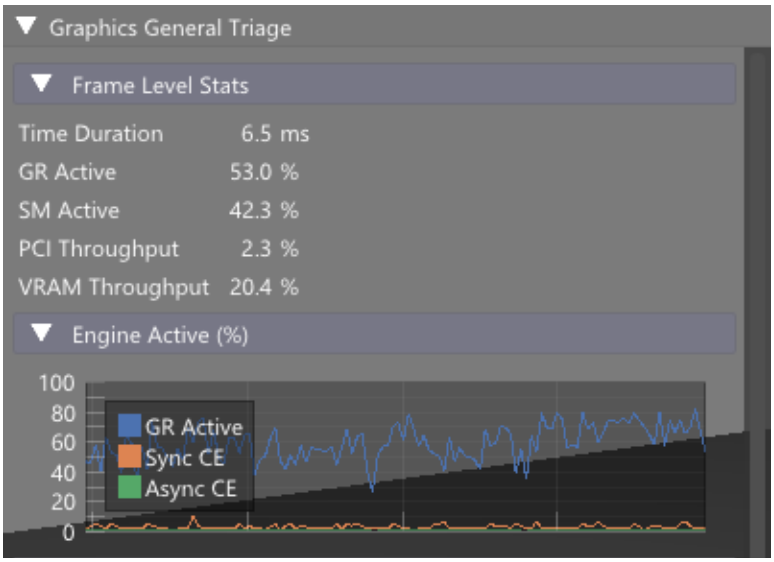
Additional resources

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

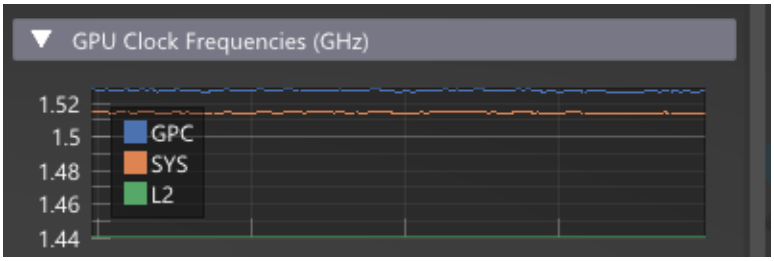
Realtime Perf HUD

The SDK includes one *Graphics General Triage* HUD configuration for each of the Ada, Ampere GA10x and Turing 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.

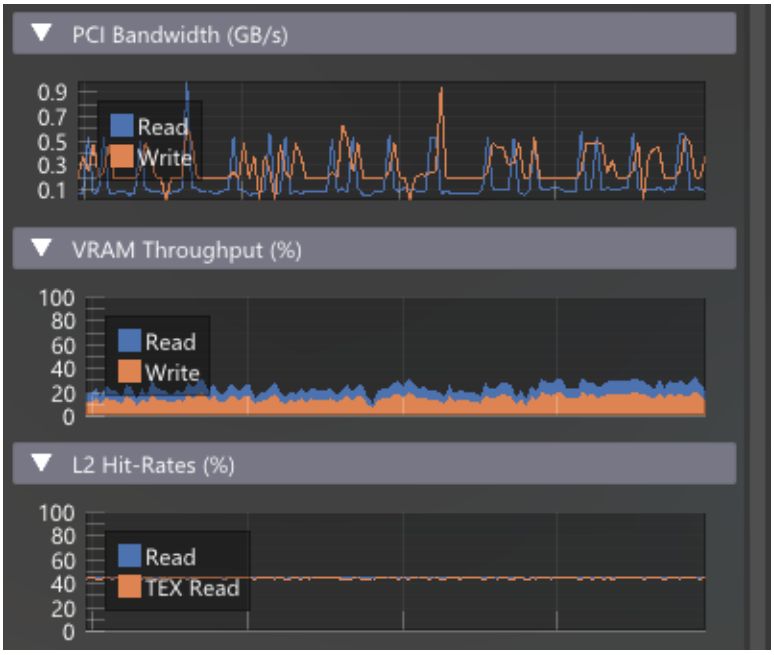
The Frame Level Stats and Engine Active panels offer top-level info on core throughputs including Frame time and Copy Engine activity.



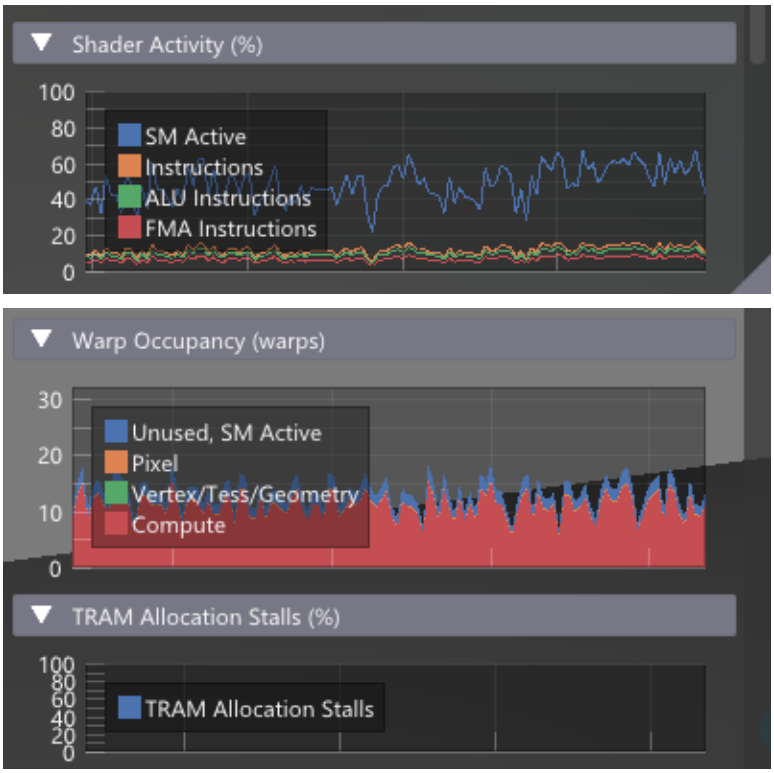
Clock frequencies plotted over time help understand throttling behavior (of particular interest on laptops), and provide context for the other plots.



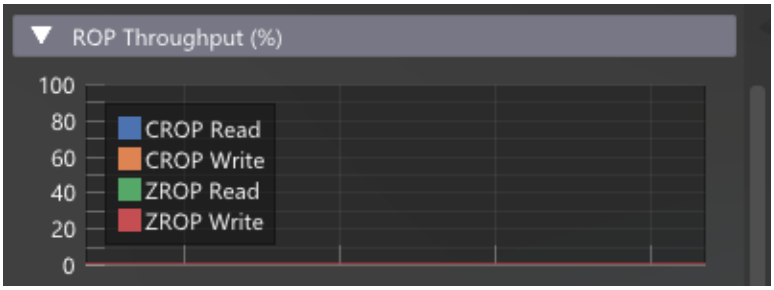
PCI Bandwidth, VRAM Throughput and L2 Hit-Rates offer core performance data on the interconnect and memory performance.



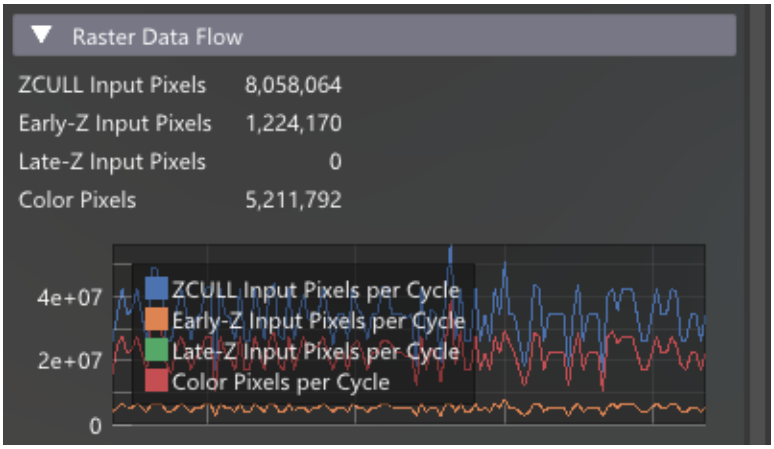
Shader Activity offers insight on SM utilization. If SM utilization is low, identify whether the limiter is low Warp Occupancy. Further identify if low Warp Occupancy was caused by excessive pixel shader attributes, indicated by high “TRAM Allocation Stalls”.



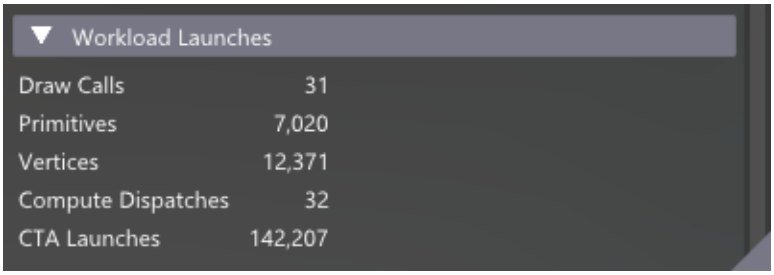
The ROP Throughput panel summarizes CROP and ZROP utilization.



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.

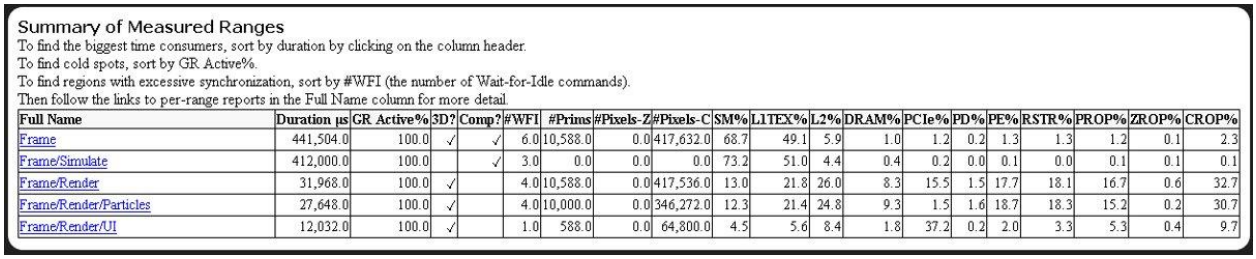
Expect percentages to be in the range 0 - 105% in typical runs, where the inherent variability of frame execution causes minor inconsistencies across measurements. Related topic - ensuring stable measurements.

HTML Report

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

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:

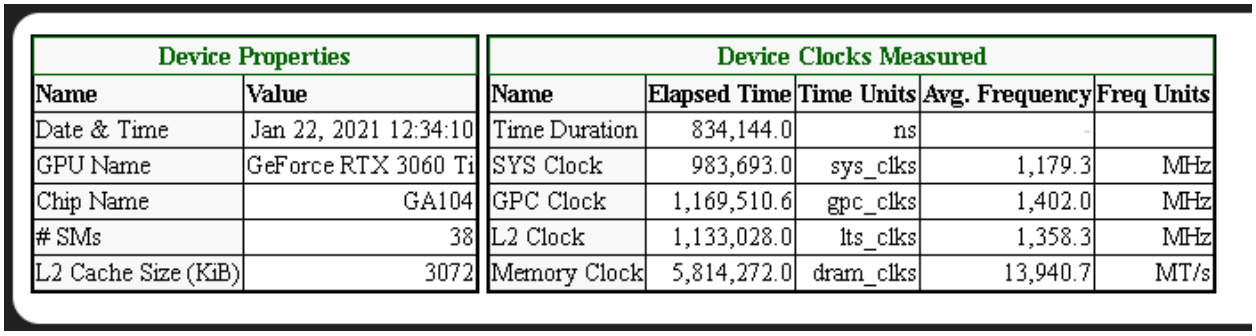
1. The file is standalone, and can be shared by attaching to emails or copying only the one file into a destination.
2. It can be loaded into a browser from disk without CORS errors.
3. Any portion can be copy/pasted into email or documents, as all contents are textual.
    a. Pro tip: make sure to select 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	Jan 22, 2021 12:34:10
GPU Name	GeForce RTX 3060 Ti
Chip Name	GA104
# SMs	38
L2 Cache Size (KiB)	3072

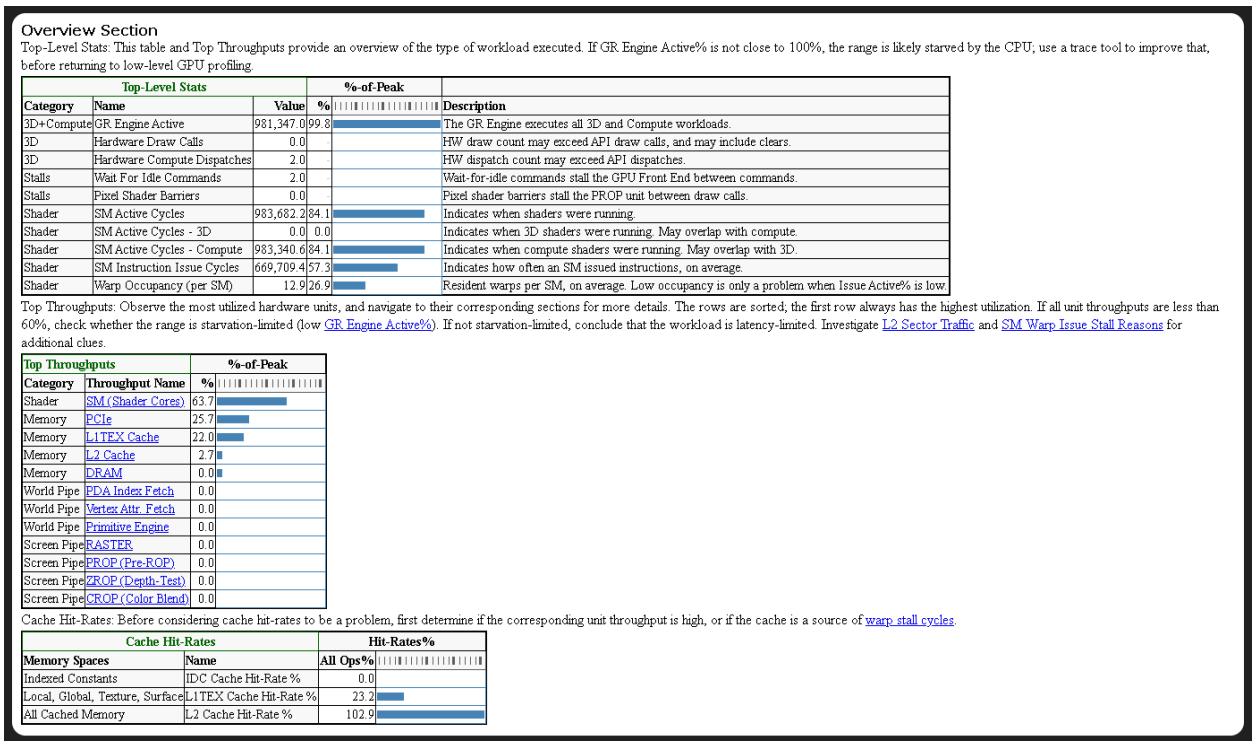
Device Clocks Measured				
Name	Elapsed Time	Time Units	Avg. Frequency	Freq Units
Time Duration	834,144.0	ns		
SYS Clock	983,693.0	sys_clks	1,179.3	MHz
GPC Clock	1,169,510.6	gpc_clks	1,402.0	MHz
L2 Clock	1,133,028.0	lts_clks	1,358.3	MHz
Memory Clock	5,814,272.0	dram_clks	13,940.7	MT/s

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	981,347.0	99.8	The GR Engine executes all 3D and Compute workloads.
3D	Hardware Draw Calls	0.0		HW draw count may exceed API draw calls, and may include clears.
3D	Hardware Compute Dispatches	2.0		HW dispatch count may exceed API dispatches.
Stalls	Wait For Idle Commands	2.0		Wait-for-idle commands stall the GPU Front End between commands.
Stalls	Pixel Shader Barriers	0.0		Pixel shader barriers stall the PROP unit between draw calls.
Shader	SM Active Cycles	983,682.2	84.1	Indicates when shaders were running.
Shader	SM Active Cycles - 3D	0.0	0.0	Indicates when 3D shaders were running. May overlap with compute.
Shader	SM Active Cycles - Compute	983,340.6	84.1	Indicates when compute shaders were running. May overlap with 3D.
Shader	SM Instruction Issue Cycles	669,709.4	57.3	Indicates how often an SM issued instructions, on average.
Shader	Warp Occupancy (per SM)	12.9	26.9	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	%
Shader	SM (Shader Cores)	63.7
Memory	PCIe	25.7
Memory	L1TEX Cache	22.0
Memory	L2 Cache	2.7
Memory	DRAM	0.0
World Pipe	PDA Index Fetch	0.0
World Pipe	Vertex Attr. Fetch	0.0
World Pipe	Primitive Engine	0.0
Screen Pipe	RASTER	0.0
Screen Pipe	PROP (Pre-ROP)	0.0
Screen Pipe	ZROP (Depth-Test)	0.0
Screen Pipe	CROP (Color Blend)	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 %	23.2
All Cached Memory	L2 Cache Hit-Rate %	102.9

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%	%
DRAM	98.6	99.6	Reads	98.6	96.6
			Writes	97.2	3.0
			Atomics	0.0	0.0
			Reductions	100.0	0.1
System Memory	99.9	0.4	Reads	0.0	0.1
			Writes	100.0	0.1
			Atomics	0.0	0.0
			Reductions	0.0	0.0
Peer Memory	100.0	0.1	Reads	0.0	0.0
			Writes	0.0	0.0
			Atomics	0.0	0.0
			Reductions	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	98.7	96.4	L1TEX Cache	98.7	95.6	DRAM	98.7	95.6	Reads	98.8	94.0
									Writes	98.5	1.6
									Atomics	0.0	0.0
									Reductions	0.0	0.0
						Peer Memory	0.0	0.0			
						System Memory	0.0	0.0			
			Primitive Engine	0.0	0.0						
			other GPC units	83.3	0.9	DRAM	83.3	0.9	Reads	83.3	0.9
									Writes	0.0	0.0
						Peer Memory	0.0	0.0			
						System Memory	0.0	0.0			
FBP Units	∞	0.0									
HUB Units	99.9	0.5	all HUB Units	99.9	0.5	DRAM	0.0	0.0			
						Peer Memory	100.0	0.1	Reads	0.0	0.0
									Writes	0.0	0.0
						System Memory	99.9	0.4	Reads	0.0	0.1
									Writes	100.0	0.1

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	23.2	100.0		Global	23.2	100.0		Global Load	16.3	91.2	
								Global Store	94.5	8.8	
								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. 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.

Profiling Permissions

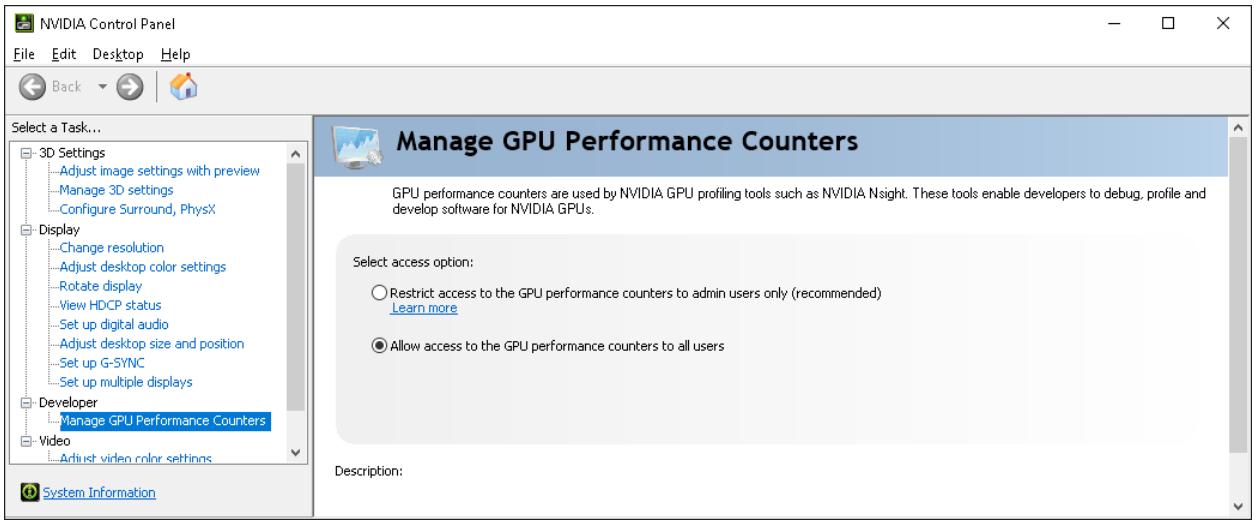
Follow the instructions here to enable GPU profiling permissions:
https://developer.nvidia.com/ERR_NVGPUCTRPERM

If the instructions do not work for you, the following recourses are available:

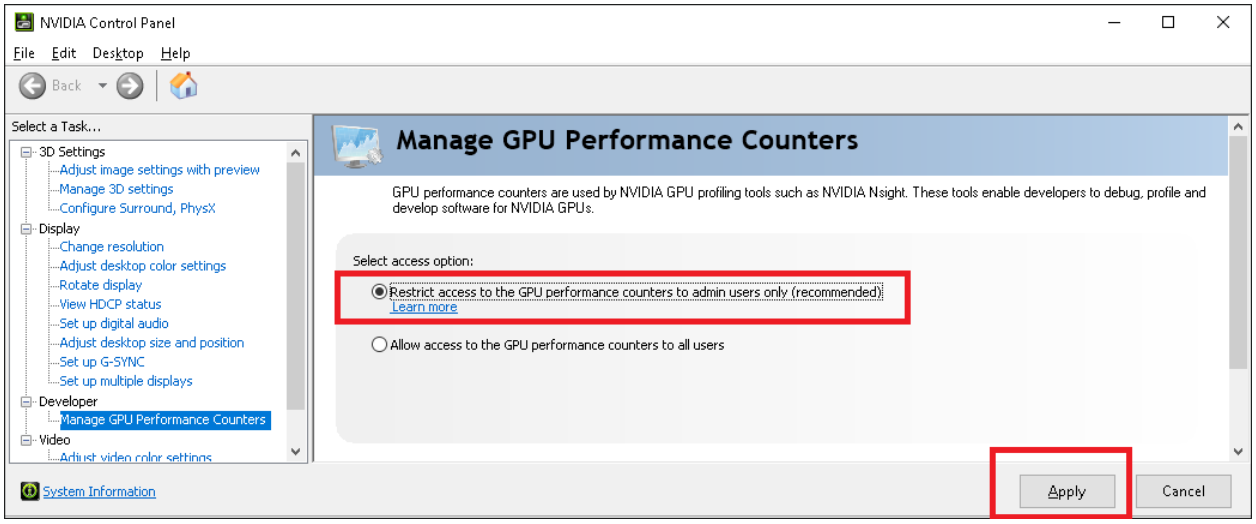
1. On Windows, toggle the setting off and back on again. (yes really)
    - This is sometimes needed after installing a new NVIDIA driver.
2. Run with root or Administrator privileges.
    - On Windows, this requires “Run as Administrator” or “elevated privileges”. A user account in the Administrator group will by default run programs with non-elevated privileges.
    - On Linux, run with sudo, or from a root command shell.
3. On Windows, attempt to manually set registry values.

Windows Control Panel Off/On Workaround

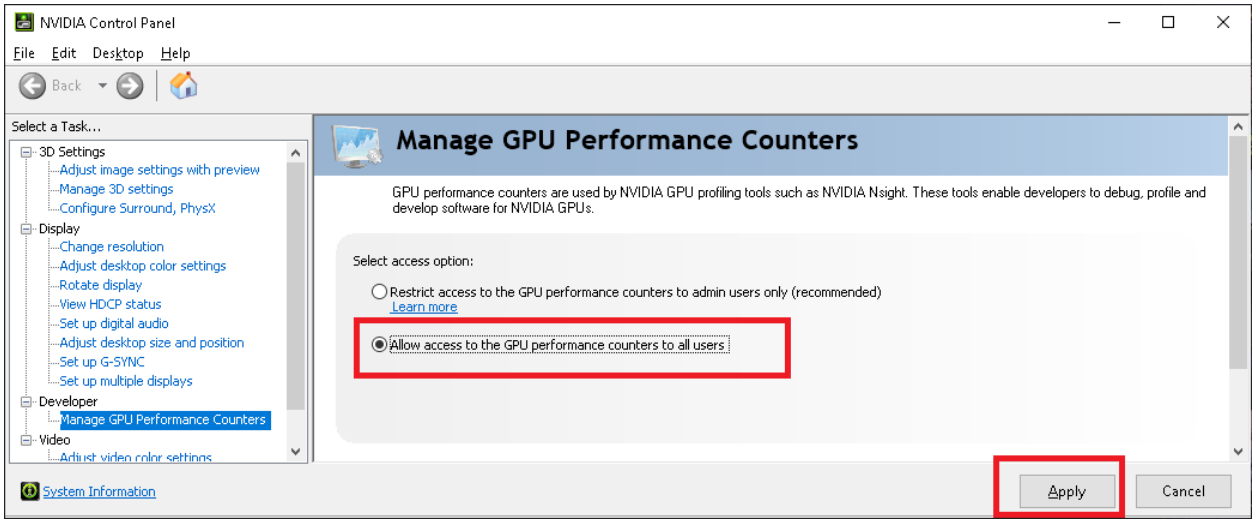
If you receive an error message from the profiler regarding permissions, but the Control Panel looks like this ...



… first toggle it off, then hit Apply.



… then toggle it back on, then hit Apply again.



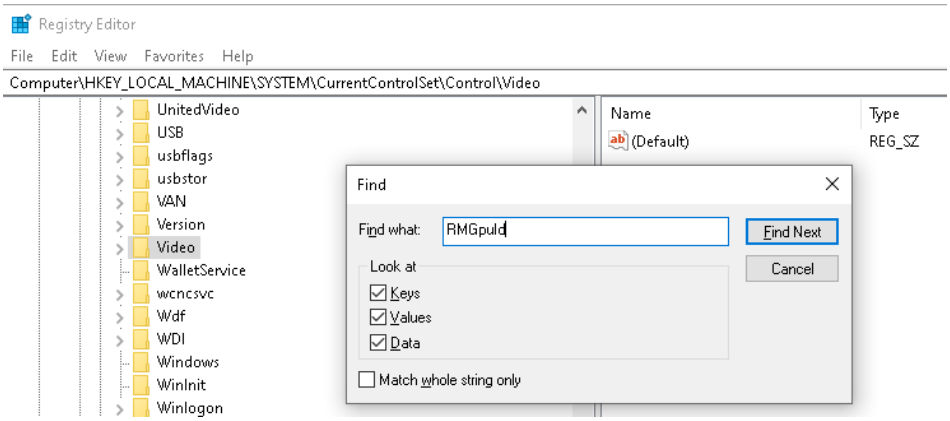
You will know the setting took effect if the screen flickered on each click of Apply, indicating a driver reload. Merely changing the radio button setting without clicking Apply will not fix the problem; you must click Apply each time.

Windows Registry Workaround

On Windows, if the NVIDIA Control Panel will not open, then it is possible to assign the equivalent settings in the Windows registry.

1. Open regedit.exe
2. Paste this URL in the top bar:
    a. Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Video
3. Search for **RMGpuId** to find the registry key (“directory”) for your first GPU.

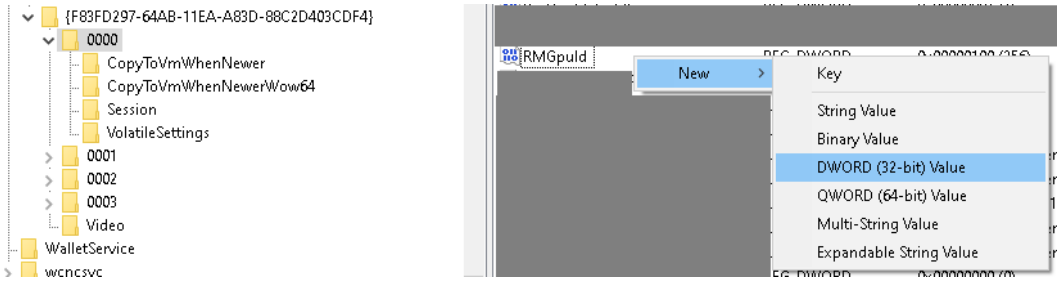
a. Ctrl+F or menu Edit | Find… opens the find dialog box.



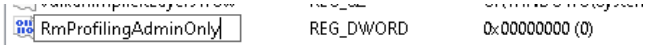
b. Found location (the IDs are machine-specific):

c. Confirm that it's a recognizable GPU

4. Create a new registry value **RmProfilingAdminOnly**
   a. Create a new DWORD (32-bit) Value



   b. Name it **RmProfilingAdminOnly**



   c. No need to alter the value. Zero grants permission to ordinary users.
5. Repeat steps 3 and 4 until all instances of RMGpuId under the Video registry key have been addressed.
6. Restart the graphics driver, to ensure the changes take effect.
   a. Option: Restart Windows.
   b. Option: Disable, then re-enable the display adapter via the Device Manager.

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, GeForce, and GeForce 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